Tetris on ATMEGA328 for a VGA output

Marco Pisani

Contents

1 Introduction

The main goal for this project was to code tetris on a microcontroller (in this case an ATMEGA328P on the Arduino Uno board) for a VGA output.

2 Signals timing and resolution

The chosen resolution is 640 x 480 @ 60Hz (data taken from http://tinyvga.com/vga-timing/640x480@60Hz):

Scanline	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

Hsync frequency	31.46875 kHz
Pixel frequency	25.175 MHz

The board works with a clock of 16 MHz, so it won't be possible to reach the actual 640x480 resolution, however it is enough to reach a decent resolution.

3 Hsync & Vsync signals

The 16-bit timer was used to generate the Hsync signal.
The clock period is $0.0625\,\mu s$, so the timer was set on fast PWM, non inverting mode with a period of 508 clock cycles ($508 \times 0.0625\,\mu s = 31.75\,\mu s$, $\approx 31.49\,kHz$).

The duty cycle was set to 0x37, in order to adjust and center the picture properly. No prescaling was applied to the input clock of the counter.
The timer is set in the setup() function.

```
void setup()
{
    DDRB |= 1 << DDB1;      //OCA1, HSYHC
    DDRB |= 1 << DDB2;      //VSYNC

    DDRD |= 1 << PORTD0;    //RED
    DDRD |= 1 << PORTD1;    //GREEN
    DDRD |= 1 << PORTD2;    //BLUE

    DDRB &= ~(1 << PORTB3); //RIGHT button
    DDRB &= ~(1 << PORTB4); //LEFT  button
    DDRB &= ~(1 << PORTB0); //ROTATE button

    OCR1AH = 0x00;
    OCR1AL = STARTING_PIXEL;

    ICR1  = HSYNC_PERIOD;

    TCCR1A |= (1 << COM1A0);
    TCCR1A |= (1 << COM1A1);

    TCCR1A &= ~(1 << WGM10);
    TCCR1A |= 1 << WGM11;
    TCCR1B |= 1 << WGM12;
    TCCR1B |= 1 << WGM13;

    TCCR1B &= ~(1 << CS10);
    TCCR1B &= ~(1 << CS11);
    TCCR1B &= ~(1 << CS12);

    TIMSK1 |= 1 << OCIE1A;

    TCCR1B |= (1 << CS10);      //Timer start

    TCCR0B |= 5;
    sei();                      //enable interrupts

    utilities3.a=0;
    current_columns = COLUMNS;
    speed = 10;
    lines_completed = 0;
}
```

In the setup function the input and output ports are set, as well as the RGB

signal (3 bits so 8 colors in total) and other variables that will be described later.
To generate the Vsync signal, the code simply counts the lines and clears the Vsync port (PORTB2) after 490 lines and sets it back after 2 lines (pulse duration).

4 Image coloring

The entire frame is divided into 19X24 squares, these values are called ROWS and COLUMNS in the header file. A char variable is assigned to each square, in this case a matrix of 19X24.

```
    volatile char Pixel_color[ROWS][COLUMNS];
```

The square with coordinates [0][0] is the one on the top-left corner of the screen, while the square [18][23] is the one on the bottom-right corner.
The last column (coordinates [n][23]) will always be black, so that the RGB signal will be off at the end of each line.
While displaying an image, each line is repeated 25 times (BLOCK_HEIGHT), meaning for the first 25 lines, the variables Pixel_color[0][n] are displayed.
To display a frame, an interrupt is used for when the counter reaches the value in the ICR1A register.

```
ISR(TIMER1_COMPA_vect)      //interrupt routine
{

    if(line == VISIBLE + FRONT + PULSE)
        PORTB |= 1 << PORTB2;


    if(line < VISIBLE)
    {
        while( j < current_columns)
        {
            PORTD = Pixel_color[i][j++];
        }
        j=0;

        if(line_count == BLOCK_HEIGHT)
        {
            i++;
```

```
            line_count = 0;
        }

    }

    if(line == VISIBLE + FRONT)
        PORTB &= ~(1 << PORTB2);

    if((utilities3.a == speed) && (utilities2.enable))
    {
        position_update();
        utilities3.a=0;
    }

    if(line == LINES)          //line increment
    {
        line = 0;
        i=0;
        line_count = 0;
        utilities3.a++;
    }
    else
    {
        line_count++;
        ++line;
    }
}
```

Whenever the interrupt arrives, PORTD is updated with the values of the Pixel_color variables for a total of "current_columns" times. The current_columns value will vary depending if the game is displaying the title screen, the game over screen or during gameplay.
The line variable is used to count lines and to know in which part of the frame the microcontroller currently is (visible area, front porch, pulse or back porch).
The line_count will tell when to pass from displaying the n^{th} line to the $(n+1)^{th}$ line of the color matrix.
The function "position_update()", when called, will advance the current tetromino by one block toward the bottom part of the screen. The function is called every "speed" frames, so higher values of speed means lower difficulty. Utilities.a is increased after every frame.

The title_colors_setup() and game_over_screen() functions will just assign the values of the color matrix so that the words "PRESS START" and "GAME

OVER" will display.

```
void title_colors_setup()
{

  background = WHITE;

  utilities1.line_check=0;

  for(i = 0; i<ROWS; i++)
  {
    for(j=0; j<COLUMNS; j++)
    {
      Pixel_color[i][j] = BLACK;
    }
  }

  //P
  Pixel_color[2][0+2] = background;
  Pixel_color[2][1+2] = background;
  Pixel_color[3][2+2] = background;
  Pixel_color[3][0+2] = background;
  Pixel_color[4][0+2] = background;
  Pixel_color[4][1+2] = background;
  Pixel_color[5][0+2] = background;
  Pixel_color[6][0+2] = background;

  //R
  Pixel_color[2][4+2] = background;
  Pixel_color[2][5+2] = background;
  Pixel_color[3][6+2] = background;
  Pixel_color[3][4+2] = background;
  Pixel_color[4][4+2] = background;
  Pixel_color[4][5+2] = background;
  Pixel_color[5][4+2] = background;
  Pixel_color[6][4+2] = background;
  Pixel_color[5][6+2] = background;
  Pixel_color[6][6+2] = background;

  //E
  Pixel_color[2][8+2] = background;
  Pixel_color[2][9+2] = background;
  Pixel_color[2][8+2] = background;
  Pixel_color[2][10+2] = background;
  Pixel_color[3][8+2] = background;
  Pixel_color[4][8+2] = background;
  Pixel_color[4][9+2] = background;
  Pixel_color[5][8+2] = background;
```

```
47  Pixel_color[6][8+2] = background;
48  Pixel_color[4][10+2] = background;
49  Pixel_color[6][10+2] = background;
50  Pixel_color[6][9+2] = background;
51
52  //S
53  Pixel_color[2][12+2] = background;
54  Pixel_color[2][13+2] = background;
55  Pixel_color[2][12+2] = background;
56  Pixel_color[2][14+2] = background;
57  Pixel_color[3][12+2] = background;
58  Pixel_color[4][12+2] = background;
59  Pixel_color[4][13+2] = background;
60  Pixel_color[5][14+2] = background;
61  Pixel_color[6][12+2] = background;
62  Pixel_color[4][14+2] = background;
63  Pixel_color[6][14+2] = background;
64  Pixel_color[6][13+2] = background;
65
66  //S
67  Pixel_color[2][16+2] = background;
68  Pixel_color[2][17+2] = background;
69  Pixel_color[2][17+2] = background;
70  Pixel_color[2][18+2] = background;
71  Pixel_color[3][16+2] = background;
72  Pixel_color[4][16+2] = background;
73  Pixel_color[4][17+2] = background;
74  Pixel_color[5][18+2] = background;
75  Pixel_color[6][16+2] = background;
76  Pixel_color[4][18+2] = background;
77  Pixel_color[6][18+2] = background;
78  Pixel_color[6][17+2] = background;
79
80
81  //S
82  Pixel_color[8][2] = background;
83  Pixel_color[9][2] = background;
84  Pixel_color[10][2] = background;
85  Pixel_color[8][3] = background;
86  Pixel_color[8][4] = background;
87  Pixel_color[10][3] = background;
88  Pixel_color[10][4] = background;
89  Pixel_color[11][4] = background;
90  Pixel_color[12][4] = background;
91  Pixel_color[12][3] = background;
92  Pixel_color[12][2] = background;
93
94  //T
95  Pixel_color[8][6] = background;
```

```
  Pixel_color[8][7] = background;
  Pixel_color[8][8] = background;
  Pixel_color[9][7] = background;
  Pixel_color[10][7] = background;
  Pixel_color[11][7] = background;
  Pixel_color[12][7] = background;

  //A
  Pixel_color[8][11] = background;
  Pixel_color[9][10] = background;
  Pixel_color[9][12] = background;
  Pixel_color[10][10] = background;
  Pixel_color[10][12] = background;
  Pixel_color[11][10] = background;
  Pixel_color[11][12] = background;
  Pixel_color[12][10] = background;
  Pixel_color[12][12] = background;
  Pixel_color[10][11] = background;

  //R
  Pixel_color[8][14] = background;
  Pixel_color[8][15] = background;
  Pixel_color[9][14] = background;
  Pixel_color[10][14] = background;
  Pixel_color[11][14] = background;
  Pixel_color[12][14] = background;
  Pixel_color[9][16] = background;
  Pixel_color[10][15] = background;
  Pixel_color[11][16] = background;
  Pixel_color[12][16] = background;

  //T
  Pixel_color[8][18] = background;
  Pixel_color[8][19] = background;
  Pixel_color[8][20] = background;
  Pixel_color[9][19] = background;
  Pixel_color[10][19] = background;
  Pixel_color[11][19] = background;
  Pixel_color[12][19] = background;

  for(int g = 14; g<4 + 14; g++)
  {
    for(int y = 2; y<21; y++)
    {
      Pixel_color[g][y] = (g+y)%8 + 1;
    }
  }
}

```

```
145
146
147 void game_over_screen()
148 {
149   for(i = 0; i<ROWS; i++)
150   {
151     for(j=0; j<COLUMNS; j++)
152     {
153       Pixel_color[i][j] = 0;
154     }
155   }
156
157
158   //G
159   Pixel_color[2][0] = background;
160   Pixel_color[2][1] = background;
161   Pixel_color[2][2] = background;
162   Pixel_color[2][3] = background;
163   Pixel_color[2][4] = background;
164   Pixel_color[3][0] = background;
165   Pixel_color[4][0] = background;
166   Pixel_color[5][0] = background;
167   Pixel_color[6][0] = background;
168   Pixel_color[6][1] = background;
169   Pixel_color[6][2] = background;
170   Pixel_color[6][3] = background;
171   Pixel_color[6][4] = background;
172   Pixel_color[5][4] = background;
173   Pixel_color[4][4] = background;
174   Pixel_color[4][3] = background;
175
176   //A
177   Pixel_color[2][7] = background;
178   Pixel_color[2][8] = background;
179   Pixel_color[2][9] = background;
180   Pixel_color[3][6] = background;
181   Pixel_color[3][10] = background;
182   Pixel_color[4][6] = background;
183   Pixel_color[5][6] = background;
184   Pixel_color[6][6] = background;
185   Pixel_color[4][10] = background;
186   Pixel_color[5][10] = background;
187   Pixel_color[6][10] = background;
188   Pixel_color[4][9] = background;
189   Pixel_color[4][8] = background;
190   Pixel_color[4][7] = background;
191
192   //M
193   Pixel_color[2][12] = background;
```

```
194   Pixel_color[3][12] = background;
195   Pixel_color[4][12] = background;
196   Pixel_color[5][12] = background;
197   Pixel_color[6][12] = background;
198   Pixel_color[3][13] = background;
199   Pixel_color[3][15] = background;
200   Pixel_color[2][16] = background;
201   Pixel_color[4][14] = background;
202   Pixel_color[3][16] = background;
203   Pixel_color[4][16] = background;
204   Pixel_color[5][16] = background;
205   Pixel_color[6][16] = background;
206
207   //E
208   Pixel_color[2][18] = background;
209   Pixel_color[3][18] = background;
210   Pixel_color[4][18] = background;
211   Pixel_color[5][18] = background;
212   Pixel_color[6][18] = background;
213   Pixel_color[2][19] = background;
214   Pixel_color[2][20] = background;
215   Pixel_color[2][21] = background;
216
217   Pixel_color[4][19] = background;
218   Pixel_color[4][20] = background;
219
220   Pixel_color[6][19] = background;
221   Pixel_color[6][20] = background;
222   Pixel_color[6][21] = background;
223
224   //O
225
226   Pixel_color[9][0] = background;
227   Pixel_color[10][0] = background;
228   Pixel_color[11][0] = background;
229   Pixel_color[12][0] = background;
230   Pixel_color[13][0] = background;
231   Pixel_color[9][1] = background;
232   Pixel_color[9][2] = background;
233   Pixel_color[9][3] = background;
234   Pixel_color[9][4] = background;
235   Pixel_color[10][4] = background;
236   Pixel_color[11][4] = background;
237   Pixel_color[12][4] = background;
238   Pixel_color[13][4] = background;
239   Pixel_color[13][1] = background;
240   Pixel_color[13][2] = background;
241   Pixel_color[13][3] = background;
242
```

```
  //V
  Pixel_color[9][6] = background;
  Pixel_color[10][6] = background;
  Pixel_color[11][7] = background;
  Pixel_color[12][7] = background;
  Pixel_color[13][8] = background;
  Pixel_color[12][9] = background;
  Pixel_color[11][9] = background;
  Pixel_color[10][10] = background;
  Pixel_color[9][10] = background;

  //E
  Pixel_color[9][12] = background;
  Pixel_color[9][13] = background;
  Pixel_color[9][14] = background;
  Pixel_color[9][15] = background;
  Pixel_color[10][12] = background;
  Pixel_color[11][12] = background;
  Pixel_color[12][12] = background;
  Pixel_color[13][12] = background;
  Pixel_color[13][13] = background;
  Pixel_color[13][14] = background;
  Pixel_color[13][15] = background;
  Pixel_color[11][13] = background;
  Pixel_color[11][14] = background;

  //R
  Pixel_color[9][17] = background;
  Pixel_color[9][18] = background;
  Pixel_color[9][19] = background;
  Pixel_color[10][17] = background;
  Pixel_color[11][17] = background;
  Pixel_color[12][17] = background;
  Pixel_color[13][17] = background;
  Pixel_color[10][20] = background;
  Pixel_color[11][19] = background;
  Pixel_color[11][18] = background;
  Pixel_color[12][19] = background;
  Pixel_color[13][20] = background;
}
```

Figure 1: Title screen

Figure 2: Game over screen

5 Tetrominoes

5.1 Tetrominoes coordinates

There are seven Tetrominoes, each one composed by four blocks.
One of the blocks is called "reference block" and its position is absolute, so to store its coordinates two variables are needed, one for the row and one for the column.
The variables "reference_row" and "reference_column" are stored in registers r8 and r14.
The coordinates of the other three blocks are relative to the reference block, expressed as its coordinates plus a displacement that can vary from -2 to +2. This allows to use less bits for the other three blocks so that their coordinates can be stored directly into registers.
In figure 3, block n° 1 is the reference block, with coordinates (X, Y), block

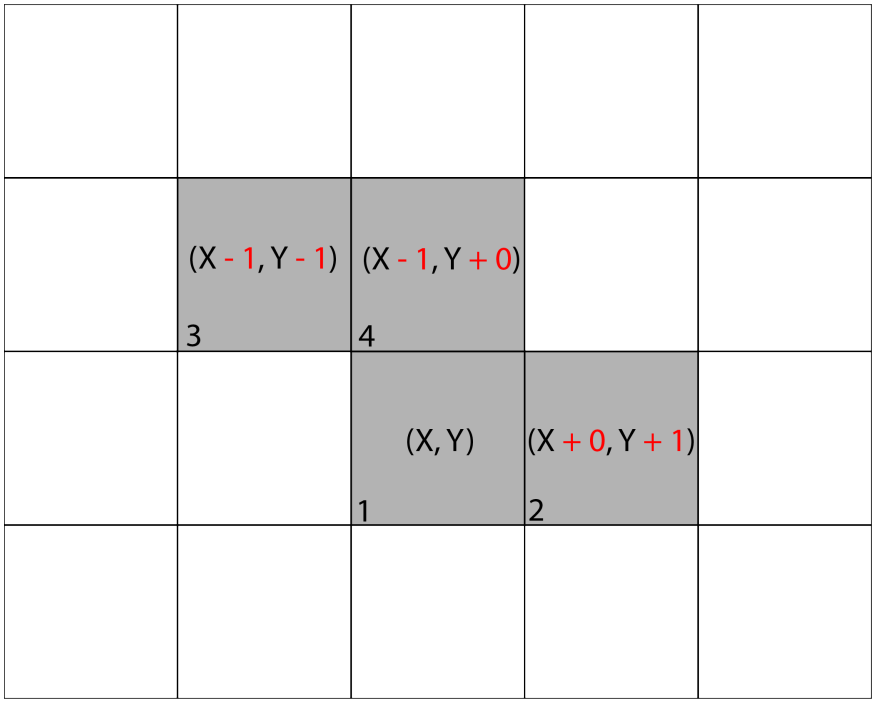

Figure 3: Inverse Skew tetromino

n° 2 can be expressed as (X + 0, Y + 1), so only the numbers (0, +1) are needed to identify that block.
The structures "second_third" and "fourth" are used to store these values.

```
    typedef struct secondary_blocks
    {
    //second block
        char block2_row   :2;
        char block2_column  :2;

    //third block
        char block3_row   :2;
        char block3_column  :2;

    }secondary_blocks;
    register secondary_blocks second_third asm("r9");

    typedef struct fourth_block
    {
    //fourth block, 3 bits so it can assume the value -2 or +2
        char block4_row   :3;
        char block4_column  :3;

    }fourth_block;
    register fourth_block fourth asm("r10");
```

5.2 Collisions

Consider the inverse Skew tetromino at Figure 3, in order to tell if the piece has reached the bottom, or there is another piece below, we have to check if the blocks below block n° 1, 2 and 3 are occupied or not.
To check if the inverse Skew tetromino can move one block to the left, we have to check if the blocks to the left of blocks n° 1 and 3 are occupied or not.
To check if the inverse Skew can move to the right we have to check block n° 2 and 4.
This information is stored in the "position_checking" variable, a 16 bit variable where each block is described by 4 bits, one for each direction.
The value relative to one direction is 1 if the block is to be considered when moving into that direction: for example, if we consider block 2 of the inverse Skew tetromino, it has to be considered when moving down, right and up but not when moving left. This means that block n° 2 has another block on the left that is part of the same tetromino.

```
typedef struct position_checking
{
    unsigned short left_block1 :1;
```

```
    unsigned short left_block2 :1;
    unsigned short left_block3 :1;
    unsigned short left_block4 :1;

    unsigned short low_block1 :1;
    unsigned short low_block2 :1;
    unsigned short low_block3 :1;
    unsigned short low_block4 :1;

    unsigned short right_block1 :1;
    unsigned short right_block2 :1;
    unsigned short right_block3 :1;
    unsigned short right_block4 :1;

    unsigned short high_block1 :1;
    unsigned short high_block2 :1;
    unsigned short high_block3 :1;
    unsigned short high_block4 :1;

}position_checking;
```

The order in which the bits are stored is not random.
When rotating a tetromino counter-clockwise, The "position_checking" variable just needs to be rotated by 4 bits. Consider the inverse Skew rotated

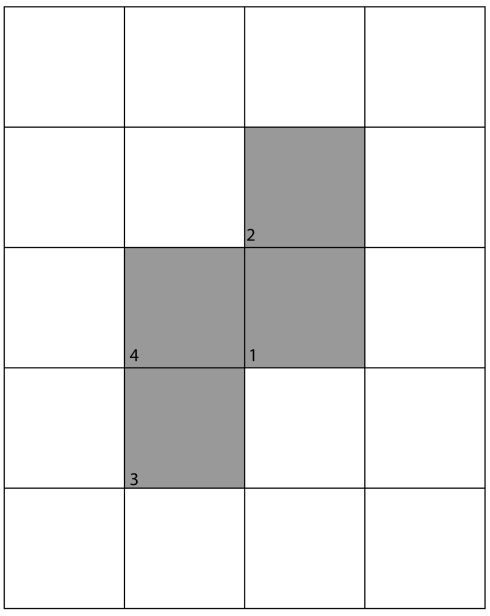

Figure 4: Rotated inverse skew

by 90° counter-clockwise at figure 4.
Now block 1 and 3 are to be considered when moving the tetromino toward the bottom, before the rotation they were to be considered when moving left. If rotated again, block 1 and 3 will be considered when moving to the right.

5.3 Types of tetromino

Every characteristic of each tetromino is written before the game begins, in the block_type_setup() function:

```
1   void block_types_setup()
2   {
3       //square tetromino, 0
4       blocks[0].second_third_position.block2_row = 0;
5       blocks[0].second_third_position.block2_column = -1;
6
7       blocks[0].second_third_position.block3_row = -1;
8       blocks[0].second_third_position.block3_column = 0;
9
10      blocks[0].fourth_position.block4_row = -1;
11      blocks[0].fourth_position.block4_column = -1;
12
13      starting_collisions[0].collisions.low_block1 = 1;
14      starting_collisions[0].collisions.low_block2 = 1;
15      starting_collisions[0].collisions.low_block3 = 0;
16      starting_collisions[0].collisions.low_block4 = 0;
17
18      starting_collisions[0].collisions.left_block1 = 0;
19      starting_collisions[0].collisions.left_block2 = 1;
20      starting_collisions[0].collisions.left_block3 = 0;
21      starting_collisions[0].collisions.left_block4 = 1;
22
23      starting_collisions[0].collisions.right_block1 = 1;
24      starting_collisions[0].collisions.right_block2 = 0;
25      starting_collisions[0].collisions.right_block3 = 1;
26      starting_collisions[0].collisions.right_block4 = 0;
27
28      starting_collisions[0].collisions.high_block1 = 0;
29      starting_collisions[0].collisions.high_block2 = 0;
30      starting_collisions[0].collisions.high_block3 = 1;
31      starting_collisions[0].collisions.high_block4 = 1;
32
33      //long tetromino, 1
34      blocks[1].second_third_position.block2_row = -1;
35      blocks[1].second_third_position.block2_column = 0;
36
37      blocks[1].second_third_position.block3_row = +1;
38      blocks[1].second_third_position.block3_column = 0;
```

```
39
40          blocks[1].fourth_position.block4_row = +2;
41          blocks[1].fourth_position.block4_column = 0;
42
43          starting_collisions[1].collisions.low_block1 = 0;
44          starting_collisions[1].collisions.low_block2 = 0;
45          starting_collisions[1].collisions.low_block3 = 0;
46          starting_collisions[1].collisions.low_block4 = 1;
47
48          starting_collisions[1].collisions.left_block1 = 1;
49          starting_collisions[1].collisions.left_block2 = 1;
50          starting_collisions[1].collisions.left_block3 = 1;
51          starting_collisions[1].collisions.left_block4 = 1;
52
53          starting_collisions[1].collisions.right_block1 = 1;
54          starting_collisions[1].collisions.right_block2 = 1;
55          starting_collisions[1].collisions.right_block3 = 1;
56          starting_collisions[1].collisions.right_block4 = 1;
57
58          starting_collisions[1].collisions.high_block1 = 0;
59          starting_collisions[1].collisions.high_block2 = 1;
60          starting_collisions[1].collisions.high_block3 = 0;
61          starting_collisions[1].collisions.high_block4 = 0;
62
63          //T tetromino, 2
64          blocks[2].second_third_position.block2_row = 0;
65          blocks[2].second_third_position.block2_column = -1;
66
67          blocks[2].second_third_position.block3_row = 0;
68          blocks[2].second_third_position.block3_column = 1;
69
70          blocks[2].fourth_position.block4_row = -1;
71          blocks[2].fourth_position.block4_column = 0;
72
73          starting_collisions[2].collisions.low_block1 = 1;
74          starting_collisions[2].collisions.low_block2 = 1;
75          starting_collisions[2].collisions.low_block3 = 1;
76          starting_collisions[2].collisions.low_block4 = 0;
77
78          starting_collisions[2].collisions.left_block1 = 0;
79          starting_collisions[2].collisions.left_block2 = 1;
80          starting_collisions[2].collisions.left_block3 = 0;
81          starting_collisions[2].collisions.left_block4 = 1;
82
83          starting_collisions[2].collisions.right_block1 = 0;
84          starting_collisions[2].collisions.right_block2 = 0;
85          starting_collisions[2].collisions.right_block3 = 1;
86          starting_collisions[2].collisions.right_block4 = 1;
87
```

```
88          starting_collisions[2].collisions.high_block1 = 0;
89          starting_collisions[2].collisions.high_block2 = 1;
90          starting_collisions[2].collisions.high_block3 = 1;
91          starting_collisions[2].collisions.high_block4 = 1;
92
93          //skew tetromino, 3
94          blocks[3].second_third_position.block2_row = 0;
95          blocks[3].second_third_position.block2_column = -1;
96
97          blocks[3].second_third_position.block3_row = -1;
98          blocks[3].second_third_position.block3_column = 1;
99
100         blocks[3].fourth_position.block4_row = -1;
101         blocks[3].fourth_position.block4_column = 0;
102
103         starting_collisions[3].collisions.low_block1 = 1;
104         starting_collisions[3].collisions.low_block2 = 1;
105         starting_collisions[3].collisions.low_block3 = 1;
106         starting_collisions[3].collisions.low_block4 = 0;
107
108         starting_collisions[3].collisions.left_block1 = 0;
109         starting_collisions[3].collisions.left_block2 = 1;
110         starting_collisions[3].collisions.left_block3 = 0;
111         starting_collisions[3].collisions.left_block4 = 1;
112
113         starting_collisions[3].collisions.right_block1 = 1;
114         starting_collisions[3].collisions.right_block2 = 0;
115         starting_collisions[3].collisions.right_block3 = 1;
116         starting_collisions[3].collisions.right_block4 = 0;
117
118         starting_collisions[3].collisions.high_block1 = 0;
119         starting_collisions[3].collisions.high_block2 = 1;
120         starting_collisions[3].collisions.high_block3 = 1;
121         starting_collisions[3].collisions.high_block4 = 1;
122
123         //inverse skew tetromino, 4
124         blocks[4].second_third_position.block2_row = 0;
125         blocks[4].second_third_position.block2_column = 1;
126
127         blocks[4].second_third_position.block3_row = -1;
128         blocks[4].second_third_position.block3_column = -1;
129
130         blocks[4].fourth_position.block4_row = -1;
131         blocks[4].fourth_position.block4_column = 0;
132
133         starting_collisions[4].collisions.low_block1 = 1;
134         starting_collisions[4].collisions.low_block2 = 1;
135         starting_collisions[4].collisions.low_block3 = 1;
136         starting_collisions[4].collisions.low_block4 = 0;
```

```
137
138         starting_collisions[4].collisions.left_block1 = 1;
139         starting_collisions[4].collisions.left_block2 = 0;
140         starting_collisions[4].collisions.left_block3 = 1;
141         starting_collisions[4].collisions.left_block4 = 0;
142
143         starting_collisions[4].collisions.right_block1 = 0;
144         starting_collisions[4].collisions.right_block2 = 1;
145         starting_collisions[4].collisions.right_block3 = 0;
146         starting_collisions[4].collisions.right_block4 = 1;
147
148         starting_collisions[4].collisions.high_block1 = 0;
149         starting_collisions[4].collisions.high_block2 = 1;
150         starting_collisions[4].collisions.high_block3 = 1;
151         starting_collisions[4].collisions.high_block4 = 1;
152
153         //L tetromino, 5
154         blocks[5].second_third_position.block2_row = 0;
155         blocks[5].second_third_position.block2_column = -1;
156
157         blocks[5].second_third_position.block3_row = 0;
158         blocks[5].second_third_position.block3_column = 1;
159
160         blocks[5].fourth_position.block4_row = -1;
161         blocks[5].fourth_position.block4_column = 1;
162
163         starting_collisions[5].collisions.low_block1 = 1;
164         starting_collisions[5].collisions.low_block2 = 1;
165         starting_collisions[5].collisions.low_block3 = 1;
166         starting_collisions[5].collisions.low_block4 = 0;
167
168         starting_collisions[5].collisions.left_block1 = 0;
169         starting_collisions[5].collisions.left_block2 = 1;
170         starting_collisions[5].collisions.left_block3 = 0;
171         starting_collisions[5].collisions.left_block4 = 1;
172
173         starting_collisions[5].collisions.right_block1 = 0;
174         starting_collisions[5].collisions.right_block2 = 0;
175         starting_collisions[5].collisions.right_block3 = 1;
176         starting_collisions[5].collisions.right_block4 = 1;
177
178         starting_collisions[5].collisions.high_block1 = 1;
179         starting_collisions[5].collisions.high_block2 = 1;
180         starting_collisions[5].collisions.high_block3 = 0;
181         starting_collisions[5].collisions.high_block4 = 1;
182
183         //inverse L tetromino 6
184         blocks[6].second_third_position.block2_row = -1;
185         blocks[6].second_third_position.block2_column = 0;
```

```
186
187         blocks[6].second_third_position.block3_row = 0;
188         blocks[6].second_third_position.block3_column = 1;
189
190         blocks[6].fourth_position.block4_row = 0;
191         blocks[6].fourth_position.block4_column = 2;
192
193         starting_collisions[6].collisions.low_block1 = 1;
194         starting_collisions[6].collisions.low_block2 = 0;
195         starting_collisions[6].collisions.low_block3 = 1;
196         starting_collisions[6].collisions.low_block4 = 1;
197
198         starting_collisions[6].collisions.left_block1 = 1;
199         starting_collisions[6].collisions.left_block2 = 1;
200         starting_collisions[6].collisions.left_block3 = 0;
201         starting_collisions[6].collisions.left_block4 = 0;
202
203         starting_collisions[6].collisions.right_block1 = 0;
204         starting_collisions[6].collisions.right_block2 = 1;
205         starting_collisions[6].collisions.right_block3 = 0;
206         starting_collisions[6].collisions.right_block4 = 1;
207
208         starting_collisions[6].collisions.high_block1 = 0;
209         starting_collisions[6].collisions.high_block2 = 1;
210         starting_collisions[6].collisions.high_block3 = 1;
211         starting_collisions[6].collisions.high_block4 = 1;
212
213         colors[0] = 1;
214         colors[1] = 2;
215         colors[2] = 3;
216         colors[3] = 4;
217         colors[4] = 5;
218         colors[5] = 6;
219         colors[6] = 0;
220     }
```

6 Buttons

Three buttons can be use: rotate (PINB0), move right (PINB3) and move left (PINB4).
Pressing a button will set a bit (such as utilities.rotate for the rotate button) and call the respective function (rotate()). The bit will return to 0 only when the button is released and the function won't be called again if the bit is 1.
Using this method, the code checking each button can be put in the main function that is looped as long as the player doesn't lose.

```
        //rotate button
        if( utilities2.rotate == 0 )
        {
            _delay_ms(10);

            if((PINB & 0x01) == 0x01 && (utilities2.enable))
            {
                utilities2.rotate = 1;  //button pressed
                rotate();
            }
        }
        if( (utilities2.rotate == 1) && (PINB & 0X01) == 0 )
        {
            utilities2.rotate = 0;
        }

        //right button
        if( utilities1.right == 0 )
        {
            _delay_ms(10);

            if((PINB & 0x08) == 0x08 && (utilities2.enable))
            {
                utilities1.right = 1; //button pressed
                move_right();
            }
        }
        if( (utilities1.right == 1) && (PINB & 0X08) == 0 )
        {
            utilities1.right = 0;
        }

        //left button
        if( utilities1.left == 0 )
        {
            _delay_ms(10);

            if((PINB & 0x10) == 0x10 && (utilities2.enable))
            {
            utilities1.left = 1;       //button pressed
            move_left();
            }
        }
        if( (utilities1.left == 1) && (PINB & 0X10) == 0 )
        {
        utilities1.left = 0;
        }
```

7 Moving tetrominoes

Each button will call a function that will first check if the piece can be moved and then modifies the color matrix.
In addition the piece moves automatically after some frames toward the bottom of the screen, if it can't go further then the function starting_position() will be called that randomly generates a new block on the top of the screen.
The utilities2.enable is cleared at the beginning of each function and set back at the end; when cleared it disables other move-like functions.

7.1 Move left & move right

Moving a piece to the left/right is done by simply subtracting/adding 1 to the reference_column variable after coloring the current occupied blocks with the background color.
Before moving the piece, each function will check the collision bit (left or right) of each block.

```
1  void move_right()
2  {
3      if(current_collisions.collisions.right_block1)
4      {
5          if(reference_column == (RIGHT_MARGIN - 1) || Pixel_color
       [reference_row][reference_column + 1] != background)
6          {
7              return;
8          }
9      }
10
11     if(current_collisions.collisions.right_block2)
12     {
13         if(reference_column + second_third.block2_column == (
       RIGHT_MARGIN - 1) || Pixel_color[reference_row + second_third
       .block2_row][reference_column + second_third.block2_column +
       1] != background)
14         {
15             return;
16         }
17     }
18
19     if(current_collisions.collisions.right_block3)
20     {
21         if(reference_column + second_third.block3_column == (
       RIGHT_MARGIN - 1) || Pixel_color[reference_row + second_third
       .block3_row][reference_column + second_third.block3_column +
```

```
    1] != background)
22          {
23              return;
24          }
25      }
26
27      if(current_collisions.collisions.right_block4)
28      {
29          if(reference_column + fourth.block4_column == (
    RIGHT_MARGIN - 1) || Pixel_color[reference_row + fourth.
    block4_row][reference_column + fourth.block4_column + 1] !=
    background)
30          {
31              return;
32          }
33      }
34      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] = background;
35      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] = background;
36      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = background;
37      Pixel_color[reference_row][reference_column++] = background;
38
39      Pixel_color[reference_row][reference_column] = current_color
    ;
40      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] =
    current_color;
41      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] =
    current_color;
42      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = current_color;
43  }
44
45  void move_left()
46  {
47      if(current_collisions.collisions.left_block1)
48      {
49          if(reference_column == (LEFT_MARGIN) || Pixel_color[
    reference_row][reference_column - 1] != background)
50          {
51              return;
52          }
53      }
54
55      if(current_collisions.collisions.left_block2)
56      {
```

```
57          if(reference_column + second_third.block2_column == (
    LEFT_MARGIN) || Pixel_color[reference_row + second_third.
    block2_row][reference_column + second_third.block2_column -
    1] != background)
58          {
59              return;
60          }
61      }
62
63      if(current_collisions.collisions.left_block3)
64      {
65          if(reference_column + second_third.block3_column == (
    LEFT_MARGIN) || Pixel_color[reference_row + second_third.
    block3_row][reference_column + second_third.block3_column -
    1] != background)
66          {
67              return;
68          }
69      }
70
71      if(current_collisions.collisions.left_block4)
72      {
73          if(reference_column + fourth.block4_column == (
    LEFT_MARGIN) || Pixel_color[reference_row + fourth.block4_row
    ][reference_column + fourth.block4_column - 1] != background)
74          {
75              return;
76          }
77      }
78      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] = background;
79      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] = background;
80      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = background;
81      Pixel_color[reference_row][reference_column--] = background;
82
83      Pixel_color[reference_row][reference_column] = current_color
    ;
84      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] =
    current_color;
85      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] =
    current_color;
86      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = current_color;
87  }
```

7.2 Rotate

Rotation is performed by multiplying each block's coordinates by the rotating matrix:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

90° counter-clockwise

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

90° clockwise

If the new blocks aren't already occupied or out of bounds then the function will color these blocks with the piece color, otherwise it will revert the rotation.
A successful rotation will also make the collision variable rotate by 4 bits.

```
1  void rotate()
2  {
3      utilities2.enable = 0;
4
5      //first empty the blocks
6      Pixel_color[reference_row + second_third.block2_row][
       reference_column + second_third.block2_column] = background;
7      Pixel_color[reference_row + second_third.block3_row][
       reference_column + second_third.block3_column] = background;
8      Pixel_color[reference_row + fourth.block4_row][
       reference_column + fourth.block4_column] = background;
9      Pixel_color[reference_row][reference_column] = background;
10
11     //rotation of block coordinates, counter-clockwise
12     //rotation matrix
13     /*
14
15         0 -1
16
17         1  0
18
19         */
20     //if the block has coordinates X and Y, the new coordinates
       are -Y and X
21
22     //clock wise
23     /*
24
```

```
25          0  1
26
27          -1   0
28
29      */
30      //if the block has coordinates X and Y, the new coordinates
        are Y and -X
31
32      temp = 0;
33      temp = second_third.block2_row;
34      second_third.block2_row = -second_third.block2_column;  //X
        = -Y
35      second_third.block2_column = temp;        //Y = X
36
37      temp = second_third.block3_row;
38      second_third.block3_row = -second_third.block3_column;  //X
        = -Y
39      second_third.block3_column = temp;        //Y = X
40
41      temp = fourth.block4_row;
42      fourth.block4_row = -fourth.block4_column;    //X = -Y
43      fourth.block4_column = temp;        //Y = X
44
45      //now checks if the new blocks aren't already full or out of
        bounds, if there are errors with the new block then temp = 1
46
47
48
49      temp = 0;
50
51      if(Pixel_color[reference_row + second_third.block2_row][
        reference_column + second_third.block2_column] != background
        || reference_column + second_third.block2_column >
        RIGHT_MARGIN-1)
52      {
53          temp = 1;
54      }
55
56      if(Pixel_color[reference_row + second_third.block3_row][
        reference_column + second_third.block3_column] != background
        || reference_column + second_third.block3_column >
        RIGHT_MARGIN-1)
57      {
58          temp = 1;
59      }
60
61      if(Pixel_color[reference_row + fourth.block4_row][
        reference_column + fourth.block4_column] != background ||
        reference_column + fourth.block4_column > RIGHT_MARGIN-1)
```

```
62      {
63          temp = 1;
64      }
65
66      if(temp == 1)   //error, perform a clockwise rotation to
    restore the values
67      {
68          temp = 0;
69
70          temp = second_third.block2_row;
71          second_third.block2_row = second_third.block2_column;
    //X = Y
72          second_third.block2_column = -temp;     //Y = -X
73
74          temp = second_third.block3_row;
75          second_third.block3_row = second_third.block3_column;
    //X = Y
76          second_third.block3_column = -temp;     //Y = -X
77
78          temp = fourth.block4_row;
79          fourth.block4_row = fourth.block4_column;   //X = Y
80          fourth.block4_column = -temp;       //Y = -X
81      }
82      else      //no errors
83      {
84      //block positions change
85      temp = 0;
86
87      temp = (current_collisions.temp >> 12) & 0x0f;        //
    saving the high_block values
88      current_collisions.temp = current_collisions.temp << 4;
    //shifting the positions values
89      current_collisions.temp = (current_collisions.temp | (temp &
        0x0f));  //restoring the high_values (now left values)
90      }
91
92      Pixel_color[reference_row][reference_column] = current_color
    ;
93      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] =
    current_color;
94      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] =
    current_color;
95      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = current_color;
96
97      utilities2.enable = 1;
98  }
```

7.3 Moving down

Each piece automatically moves down after a certain amount of frames, depending on the difficulty. When that happens, the function position_update() is called from within the interrupt handler.
If the block can't move down, then the current row is saved in the utilities1.row_pointer variable and the bit utilities2.reset_position is set.
The first will tell to check if that line is completed, the second will tell that a new piece is required.
Apart from this, position_update() works just as the move_left() and move_right functions.

```
1  void position_update()
2  {
3      utilities2.enable = 0;
4
5      if(current_collisions.collisions.low_block1)
6      {
7          if(reference_row == (ROWS-1) || Pixel_color[
       reference_row + 1][reference_column] != background)
8          {
9              utilities1.row_pointer = (reference_row > utilities1
       .row_pointer) ? reference_row : utilities1.row_pointer;
10             utilities2.reset_position=1;
11             utilities2.enable = 1;
12         }
13     }
14
15     if(current_collisions.collisions.low_block2)
16     {
17         if( reference_row + second_third.block2_row == (ROWS-1)
       || Pixel_color[reference_row + second_third.block2_row + 1][
       reference_column + second_third.block2_column] != background)
18         {
19             utilities1.row_pointer = (reference_row +
       second_third.block2_row > utilities1.row_pointer) ? (
       reference_row + second_third.block2_row) : utilities1.
       row_pointer;
20             utilities2.reset_position=1;
21             utilities2.enable = 1;
22         }
23     }
24
25     if(current_collisions.collisions.low_block3)
26     {
27         if( reference_row + second_third.block3_row == (ROWS-1)
       || Pixel_color[reference_row + second_third.block3_row + 1][
```

```
reference_column + second_third.block3_column] != background)
28      {
29          utilities1.row_pointer = (reference_row +
    second_third.block3_row > utilities1.row_pointer) ? (
    reference_row + second_third.block3_row) : utilities1.
    row_pointer;
30          utilities2.reset_position=1;
31          utilities2.enable = 1;
32      }
33  }
34
35  if(current_collisions.collisions.low_block4)
36  {
37      if( reference_row + fourth.block4_row == (ROWS-1) ||
    Pixel_color[reference_row + fourth.block4_row + 1][
    reference_column + fourth.block4_column] != background)
38      {
39          utilities1.row_pointer = (reference_row + fourth.
    block4_row > utilities1.row_pointer) ? (reference_row +
    fourth.block4_row) : utilities1.row_pointer;
40          utilities2.reset_position=1;
41          utilities2.enable = 1;
42      }
43  }
44
45
46  if(!utilities2.reset_position)
47  {
48      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] = background;
49      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] = background;
50      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = background;
51      Pixel_color[reference_row++][reference_column] =
    background;
52
53      Pixel_color[reference_row][reference_column] =
    current_color;
54      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] =
    current_color;
55      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] =
    current_color;
56      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = current_color;
57  }
58
```

```
    utilities2.enable = 1;
}
```

8 Line checking

Whenever utilities1.row_pointer is different from 0, then the main function will call the line_check() function.
It checks the colors of the blocks of that line and the 3 above. If one of these lines is completed (there is no background color) then all lines above are shifted to the bottom by one.

```
void line_check(char line_to_check)
{
    utilities2.enable = 0;

    for(char m = 0; m<4; m++)
    {
        for(int p = (LEFT_MARGIN); (p < (RIGHT_MARGIN)) && (utilities1.line_check != 1); p++)
        {
            if(Pixel_color[line_to_check][p] == background)
            {
                utilities1.line_check = 1;  //means the line is not completed
            }
        }

        if(utilities1.line_check == 0)  //line completed
        {
            for(char k = (line_to_check); k > 1; k--)
            {
                for(char h=LEFT_MARGIN; h < (RIGHT_MARGIN); h++)
                {
                    Pixel_color[k][h] = Pixel_color[k - 1][h];
                }
            }

            lines_completed++;
        }
        utilities1.line_check=0;
    }

    utilities2.enable = 1;
}
```

9 Generating new tetrominoes

The starting_position() function generates a new piece by looking at the value of the second counter and updates the current piece and the next piece accordingly.
It also checks if the game should end or not.

```
void starting_position()
{
    utilities2.enable = 0;    //disables other functions

    Pixel_color[next_block_row + blocks[next_block].
    second_third_position.block2_row][next_block_column + blocks[
    next_block].second_third_position.block2_column] = 0;

    Pixel_color[next_block_row + blocks[next_block].
    second_third_position.block3_row][next_block_column + blocks[
    next_block].second_third_position.block3_column] = 0;

    Pixel_color[next_block_row + blocks[next_block].
    fourth_position.block4_row][next_block_column + blocks[
    next_block].fourth_position.block4_column] = 0;

    Pixel_color[next_block_row][next_block_column] = 0;

    utilities2.block_type = next_block;

    while( ( next_block = (TCNT0 & 0x7) ) > 6);   //randomly
    generates a new piece

    second_third = blocks[utilities2.block_type].
    second_third_position;

    fourth = blocks[utilities2.block_type].fourth_position;

    current_collisions = starting_collisions[utilities2.
    block_type];

    reference_row = START_ROW;
    reference_column = START_COLUMN;

    current_color = colors[utilities2.block_type];


    //gameover check
    if(Pixel_color[reference_row][reference_column] !=
    background)
```

```
31      {
32          game_over = 1;    //game_over
33      }
34      else
35      {
36          if(Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] != background)
37          {
38              game_over = 1;    //game_over
39          }
40          else
41          {
42              if(Pixel_color[reference_row + second_third.
    block3_row][reference_column + second_third.block3_column] !=
     background)
43              {
44                  game_over = 1;    //game_over
45              }
46              else
47              {
48                  if(Pixel_color[reference_row + fourth.block4_row
    ][reference_column + fourth.block4_column] != background)
49                  {
50                      game_over = 1;    //game_over
51                  }
52              }
53          }
54      }
55
56
57
58      Pixel_color[reference_row][reference_column] = current_color
    ;
59      Pixel_color[reference_row + second_third.block2_row][
    reference_column + second_third.block2_column] =
    current_color;
60      Pixel_color[reference_row + second_third.block3_row][
    reference_column + second_third.block3_column] =
    current_color;
61      Pixel_color[reference_row + fourth.block4_row][
    reference_column + fourth.block4_column] = current_color;
62
63      Pixel_color[next_block_row + blocks[next_block].
    second_third_position.block2_row][next_block_column + blocks[
    next_block].second_third_position.block2_column] = background
    ;
64
65      Pixel_color[next_block_row + blocks[next_block].
    second_third_position.block3_row][next_block_column + blocks[
```

```
    next_block].second_third_position.block3_column] = background
    ;
66
67      Pixel_color[next_block_row + blocks[next_block].
    fourth_position.block4_row][next_block_column + blocks[
    next_block].fourth_position.block4_column] = background;
68
69      Pixel_color[next_block_row][next_block_column] = background;
70
71      utilities2.reset_position = 0;
72      utilities2.enable = 1;
73  }
```

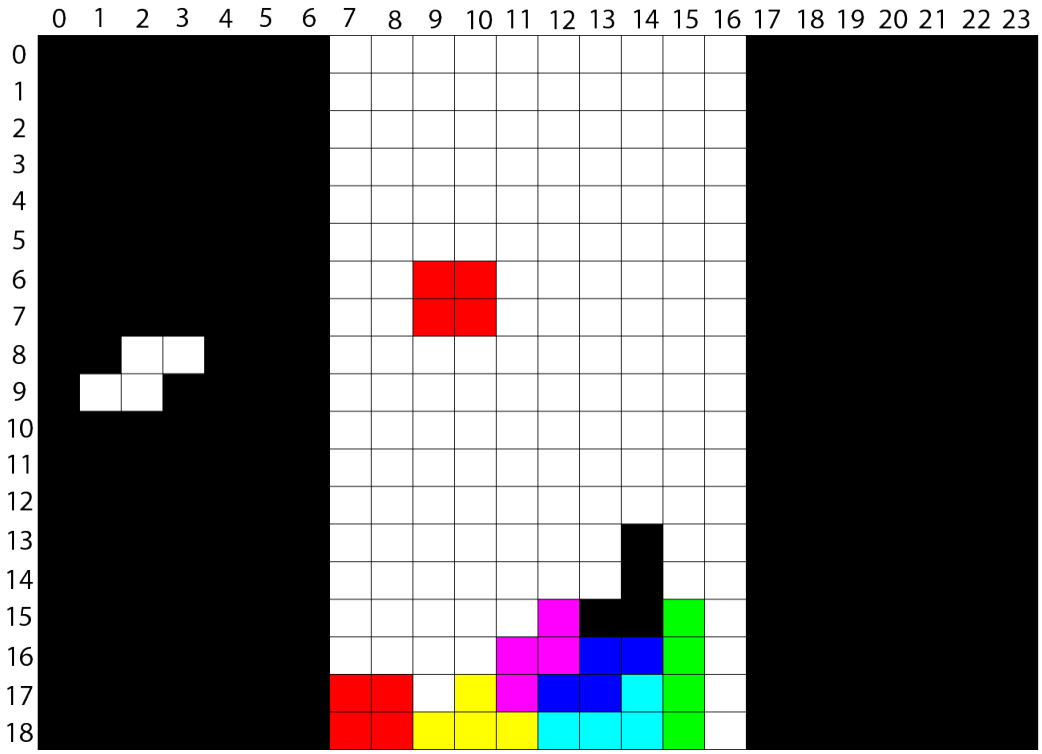

Figure 5: Gameplay