OR 22-23 DN2 poročilo

Irinej Slapal VSŠ2

Uvod:

Za 2. domačo nalogo pri predmetu OR sem se odločil, da bom na Miško3 boardu dopolnil že obstoječ projekt in implementiral igro tetris v jeziku C.

Verzija tetrisa je zelo preprosta, ko je vrsta polna se ne izbriše ampak se le pobarva.
Igre je konec, ko se tetris kosi dosežejo vrh zaslona.
Link do videa demonstracije: https://www.youtube.com/watch?v=OU4xcbiVv8w

Projekt na katerem sem začel graditi:
https://github.com/LAPSyLAB/Misko3_Docs_and_Projects.git
V projektu so bili že implementirani api-ji za periferijo (LCD, joystick, tipke).
Za implementacijo sem uporabil CUBE Integrirano programsko okolje (CUBE IDE) od stm32.

API-ji, ki sem jih uporabil

avtorji so navedeni tako, kot so se podpisali na začetek datotek

ugui.h

Avtorkse pravice: Achimu Döblerju, 2015.
µGUI je odprtokoden generičen GUI modul za vgrajene sisteme.
Uporabil sem ga za tipe barv in za pisanje ascii nizov na zaslon.

lcd.h

Avtorske pravice: Nejc Bertoncelj, marec 2022.
Ta datoteka večinoma kliče prototipe datoteke *lcd_ili9341.h*.

Tukaj sem implementiral 3 svoje funkcije za prikaz tetris kosa in igralnega polja.

```
// Displays a Tetis piece on the LCD
void LCD_Tetromino(Tetromino t) {
        for (int y = 0; y < 4; y++) {
                for (int x = 0; x < 4; x++) {
                        int gridIndex = rotateTetromino(x, y, t.rotation);
                        uint16_t tetrominoValue = t.grid[gridIndex];
                        if (tetrominoValue != 0) {
                        // If the current cell of the tetromino is "full"
                        // display it on lcd
                        int pixelX = (t.x + x-1)*10;
                        int pixelY = (t.y + y)*10;
                        // display a 10x10 window at coordinates (pixelX,
pixelY)
                        ILI9341_SetDisplayWindow(pixelX, pixelY, 10, 10);
                        // set this window to certain color
                        ILI9341_SendRepeatedData(tetrominoValue, 10*10);
                        }
                }
        }
}

// Displays the playing field on the LCD
void LCD_PlayingField(uint16_t fieldArray[], int fullRow[]) {
        for (int y = 0; y < 25; y++) {
                for (int x = 0; x < 33; x++) {
                        uint16_t faValue = fieldArray[y * 34 + x+1];
                        if (fullRow[y] == 32) {
                                // if row is full, color it gold
                                ILI9341_SetDisplayWindow(x*10, y*10, 10,
10);
                                ILI9341_SendRepeatedData(C_GOLD, (10)*
(10));
                        }
                        else if (faValue != 0) {
                                ILI9341_SetDisplayWindow(x*10, y*10, 10,
10);
                                ILI9341_SendRepeatedData(faValue, (10)*
(10));
                        }
                }
        }
}

// Displays both at once
void LCD_TetrisFrame(Tetromino t, uint16_t field[], int fullRow[]) {
        LCD_Tetromino(t);
        LCD_PlayingField(field, fullRow);
}
```

kbd.h

Avtorske pravice: Gasper, Jan 19. 2022.
API za branje gumbov.

joystick.h

Avtorske pravice: marko, 23. feb 2022.
API za branje joysticka.

Tetris Kos (Tetromino)

Trenutni tetris kos na zaslonu (kos, ki ga moramo namestiti na polje), sem predstavil kot strukturo *Tetromino*, ki vsebuje podateke o rotaciji, obliki, poziciji na zaslonu.
Obliko Tetromina sem predstavil z linearnim poljem velikosti 16, ki predstavlja 4x4 matriko, ki vsebuje barvo tetris kosa.

```
typedef struct {
        int rotation;
        char shape;
        uint16_t grid[TETROMINO_SIZE]; // linearized representation
        int x;    // location of upper most-left corner of matrix on the
lcd
        int y;
} Tetromino;
```

Globalna spremenljivka tetrominoShapes, hrani matrike za vseh 7 različnih oblik:

```
int tetrominoShapes[TETROMINO_COUNT][TETROMINO_SIZE] = {
        {
        0, 0, 0, 0,
        1, 1, 1, 1,
        0, 0, 0, 0,
        0, 0, 0, 0
        },
        ...
        ...
};
```

Tako se generira tetris kos (Tetromino):

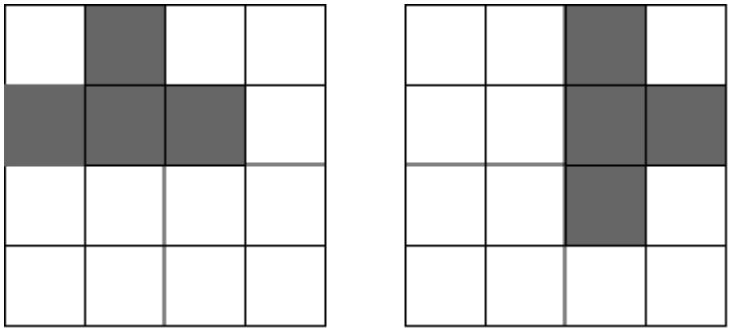
```
Tetromino generateTetromino(int s, int r, int p) {
        // s = shape
        // r = rotation
        // p = position
        Tetromino tetromino;  //define a tetromino
        uint16_t c;  // define color of tetromino
        switch (s) {
                case 0:
                        c = C_CYAN;
                        tetromino.shape = 'I';
                break;
                case 1:
                        c = C_PURPLE;
                        tetromino.shape = 'T';
                break;
                case 2:
                        c = C_YELLOW;
                        tetromino.shape = 'O';
                break;
                // and so on for all 7 shapes
        }

        for (int i = 0; i < 16; i++) {
                if (tetrominoShapes[s][i] == 1) {
                        tetromino.grid[i] = c;
                }
                else tetromino.grid[i] = 0;
        }
        tetromino.x = p;
        tetromino.y = 0;
        tetromino.rotation = r;
        return tetromino;
}
```

Rotacije Tetris kosa:

```
int rotateTetromino(int x, int y, int r) {
        switch (r % 4) {
                case 0: return (y * 4) + x; //0
                case 1: return 12 + y - (x * 4); //90
                case 2: return 15 - (y * 4) - x; //180
                case 3: return 3 - y + (x * 4); //270
        }
        return 0;
}
```

Ker Tetris kos predstavljen z 4x4 matriko, in so možne le 4 smeri rotacije, lahko obliko zarotiranega kosa preprosto izračunamo.
Vendar pa rotacije zato niso intuitivne, ker se kos rotira okoli središča njegove matrike in ne središča kosa:



rotacija za 90 stopinj

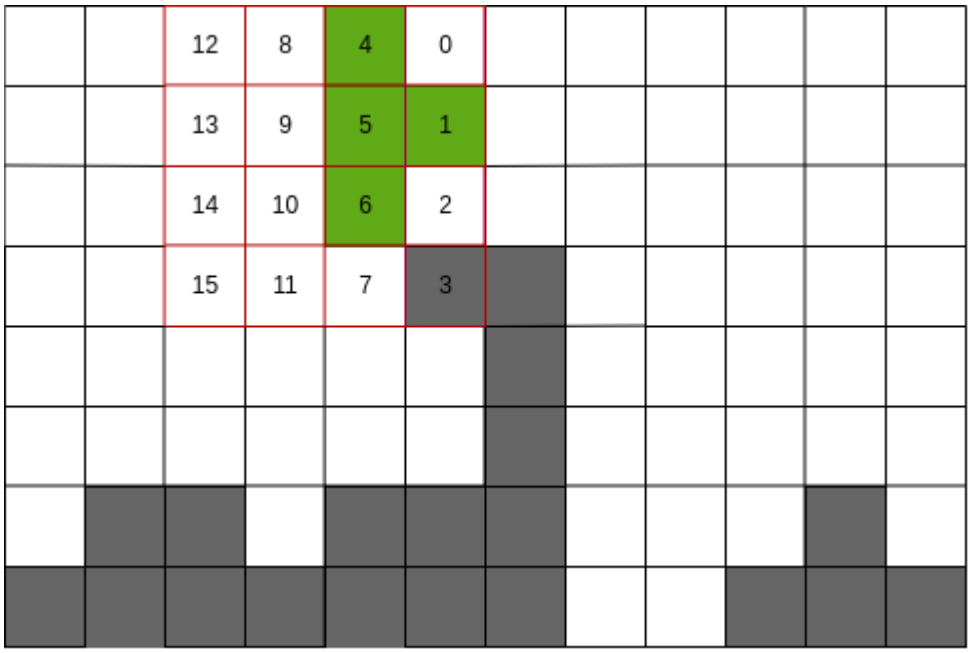
Tetris polje in Kolizije

Deklaracija tetris polja:

```
uint16_t fieldArray[34*25];
```

Tetris polje predstavlja neko mrežo čez zaslon, vsaka celica je velika 10x10 pikslov in je tipa *uitn16_t*, ki predstavlja barvo posamezne celice (tako lahko narišemo vsak kos z drugo barvo).
Ko tetris kos "pristane", se njegove polne celice prepišejo v tetris polje.
Tetris polje je inicializirano na 1, na stranskih robovih in na dnu, tako dobimo nek zid,
ki nam omogoči, da pri preverjanju kolizij ne rabimo gledati, če je tetris kos zašel iz polja.

Če postanemo živčni ko se nam tetris kos čudno obrne pri rotaciji (zaradi preslikovanja matrik), pa se lahko spomnimo, kako preprosto se da implementirati kolizije.



Pri preverjanju kolizij, lahko preprosto iteriramo čez matriko tetris kosa z dvema zankama, in po principu $(y \cdot \check{s}irina\ zaslona) + x$, izračunamo koordinate celic, ki jih matrika kosa prekriva.
Upoštevati moramo rotacijo kosa (glej sliko).
Če sta vrednosti obeh celic različni od 0, potem smo prišli do kolizije in se bo matrika tetris kosa prepisala v matriko tetris polja.

```
int collisions(Tetromino t, int tx, int ty, int rotation) {
        for (int y = 0; y < 4; y++) {
                for (int x = 0; x < 4; x++) {

                        uint16_t tValue = t.grid[rotateTetromino(x, y,
rotation)];
                        int fieldIndex = (ty + y) * 34 + (tx + x);
                        if (tx+x >= 0 && tx+x < 34) {
                                if (ty + y >= 0 && ty + y < 25) {
                                        if (tValue != 0 &&
fieldArray[fieldIndex] != 0) {
                                                // we get collision
                                                return 1;
                                        }
                                }
                        }
                }
        }
return 0;
}
```

Branje uporabnikovega vnosa

Branje gumbov:

Pri branju stanja gumbov, sem prišel do težave, ko je bil gumb prižgan takrat ko je bil pritisnjen.
To sem rešil z debouncing mehanizmom, ki generira pulz, ko je gumb pritisnjen, tako se bo gumb, če ni spuščen obnašal, kot da smo ga pritisnili le za n-tinko sekunde.

```
typedef struct {
        buttons_enum_t name;
        int previousState;
        uint32_t lastPressTime;
} ButtonState;

ButtonState up = {BTN_UP, 0, 0};
ButtonState right = {BTN_RIGHT, 0, 0};
ButtonState down = {BTN_DOWN, 0, 0};
ButtonState left = {BTN_LEFT, 0, 0};
ButtonState ok = {BTN_OK, 0, 0};
ButtonState esc = {BTN_ESC, 0, 0};
```

Tukaj preverjamo ali je gumb:

- trenutnem stanju pritisnjen,
- bil pritisnjen v prejšnjem stanju,
- ali je minilo dovolj časa od zadnjega pritiska gumba, da se gumb lahko sproži.

Stanje gumba moramo v funkcijo podati referenčno, saj tako posodobimo originalno strukturo in ne kopije.

```
int isButtonPressed(ButtonState* button) {
        int currentState = !KBD_get_button_state(button->name);
        uint32_t debounceInterval = 15;
        uint32_t currentTick = HAL_GetTick();
        int isPressed = 0; // false by default
        if (currentState && !button->previousState &&
        (currentTick - button->lastPressTime > debounceInterval)) {
        button->lastPressTime = currentTick; // Update the last press time
        isPressed = 1;
        }

        button->previousState = currentState;
        return isPressed;
}
```

Branje joysticka:

Globalni spremenljivki:

```
coord_t joystick_raw, joystick_out;
joystick_t joystick;
```

Branje vrendosti:

```
int handleJoystick() {
        HAL_ADC_Start(&hadc4);
        HAL_ADC_PollForConversion(&hadc4,10);// Waiting for ADC conversion
        joystick_raw.x=HAL_ADC_GetValue(&hadc4);

        HAL_ADC_Start(&hadc4);
        HAL_ADC_PollForConversion(&hadc4,10);// Waiting for ADC conversion
        joystick_raw.y=HAL_ADC_GetValue(&hadc4);
        HAL_ADC_Stop(&hadc4);

        joystick_get(&joystick_raw, &joystick_out, &joystick);
        if (joystick_out.x > 0) return 1;
        else if (joystick_out.x <-18) return -1;
        return 0;
}
```

Za branje joysticka se uporabi ADC (analogno-digitalna pretvorba), na vmesniku *hadc4*.
Na zaključek pretvorbe, se bo čakalo 10ms.
Po zaključku pretvorbe, se vrednosti shranijo v globalno spremenljivko *joystick_raw.*
Po spodnji fukciji se potem joystick_raw koorditate pretvorijo in zapišejo v globalno spremenljivko *joystick_out*, iz katere se potem v igralni zanki bere vrednost.

```
void joystick_get(coord_t *raw, coord_t *out, joystick_t *joystick) {
        out->x = joystick->x_k*(raw->x-joystick->n.x);
        out->y = -joystick->y_k*(raw->y-joystick->n.y);
}
```

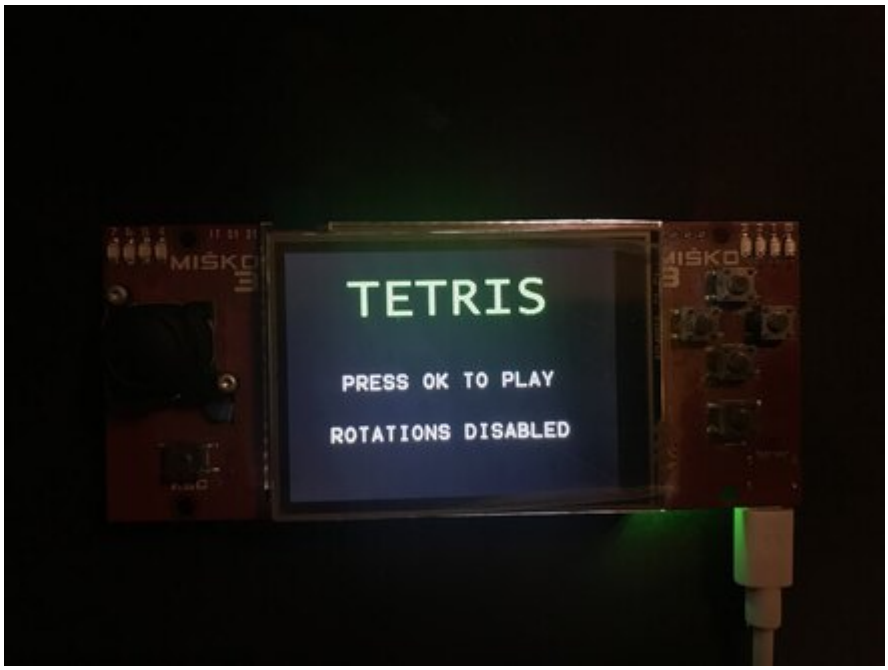
Igralna zanka

Igra ima 3 stanja:

```
typedef enum {
        START_SCREEN,
        GAMING_SCREEN,
        GAME_OVER_SCREEN
} GameState;
```

➪ začetno stanje:

- izriše začetno sliko,
- ponudi opcijo če bi igro igrali z rotacijami ali ne.



```
case START_SCREEN:
        score = 0;
        initFA();
        initFR();
        DrawStartScreen();

        int buttonPressed = handleButtons();
        if (buttonPressed < 4) {
                rotations = toggle(rotations);
                seed++;
                UG_FillScreen(C_BLACK);
        }
        if (!KBD_get_button_state(BTN_OK)) {
                UG_FillScreen(C_BLACK);
                t = generateTetromino(pseudoRandom(7), pseudoRandom(4),
16);
                currentState = GAMING_SCREEN;
                break;
        }
break;
```

- score resetira (spremenljivka beleži število vrst, ki jih je igralec uspel zapolniti),
- igralno polje in polje, ki šteje število zapolnjenih mest v vrsti se postavita v začetno stanje,
- na zaslon se izriše zgornje okno,
- ko igralec izbira med igranjem z rotacijami ali brez, se prišteva vrednost k seedu,
- seed se uporablja za računanje psevdonaključne pozicije, rotacije in oblike novega tetris kosa,
- ko igralec pritisne tipko *ok* se generira prvi tetris in program preide v igralno stanje.

➪ igralna zanka:

- sprejema vnos uporabnika,
- izrisuje tetris kos na zaslon,
- preverja kolizije,
- v primeru kolizije kos prepiše v tetrisPolje (kos pristane na tleh).

```
case GAMING_SCREEN:
        LCD_TetrisFrame(t, fieldArray, fullRow);
        HAL_Delay(10);
        int input = handleButtons();
        int inputJ = handleJoystick();
        if (input == 6) {
                LCD_ClearScreen();
                rotations = 0;
                UG_FillScreen(C_BLACK);
                currentState = START_SCREEN;
        }
        HAL_Delay(200);
        int tetrominoLanded = 0;
        rotationInput = t.rotation;
        if (rotations == 1 && t.shape != 'O') {
        rotationInput += (input < 4) ? 1 : 0;
        }
        // checks collisions
        t.x += (inputJ != 0 &&
        collisions(t, t.x+inputJ, t.y, rotationInput) == 1) ? 0 : inputJ;
        if (collisions(t, t.x, t.y+1, rotationInput) == 1) {
                t.rotation = rotationInput;
                writeToFA(t);
                tetrominoLanded = 1;
        }
        else {
                t.rotation = rotationInput;
                t.y++;
        }
        LCD_ClearScreen();
        if (tetrominoLanded == 1) {
                LCD_PlayingField(fieldArray, fullRow);
                if (t.y == 0) {
                        LCD_ClearScreen();
                        currentState = GAME_OVER_SCREEN;
                        break;
                }
```

```
                int s = pseudoRandom(7);
                int r = pseudoRandom(4);
                int p = pseudoRandom(28);
                // generates new tetromino
                t = generateTetromino(s, r, p+1);
        }
break;
```

- izriše se tetris kos, ki pada in tetris polje (vsi že pristali tetris kosi),
- preveri se vnos igralca
- Določi se dolžino prikaza na zaslonu, "frame-a" *HAL_Delay(200)*,
- preverjanje kolizij na robovih in potem na dnu.
- ko tetris kos pristane, se zapiše v tetris polje, nato se generira nov tetris kos
- program se vrača v to stanje, dokler je najvišji tetris kos pod zgornjim robom zaslona, ali dokler ne želi igralec ponovno začeti

➪ končno stanje:

- izriše končno sliko,
- ponudi možnost za vstop v začetno stanje.


```
case GAME_OVER_SCREEN:
        DrawGameOverScreen();
        if (!KBD_get_button_state(BTN_OK)) {
                UG_FillScreen(C_BLACK);
                currentState = START_SCREEN;
                break;
        }
break;
}
```