UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Rafał Gosiewski

Memory of the final work

AEV

Classic game „Space Invaders" for NDS console.

1. Table of contents

2. Scope of work

1. Introduction

This project is intended for practicing skills in designing and programming games on console NDS. All works are developed during course *Arquitectura y Entornos de Desarrollo para Videoconsolas* during my studies on Universidad Politecnica de Valencia.

The idea is to develop similar version of classic game *Space Invaders*, which has been created for arcade games and released in 1978 year. *Space Invaders* is one of the earliest shooting game and the aim is to defeat waves of aliens with a laser cannon to earn as many points as possible.

2. Description

1. Rules

Space Invaders is a two-dimensional fixed shooter game, where player controls a spaceship with a laser cannon, which can be moved only horizontally across the bottom of the screen. The aim is to defeat five rows of eleven aliens – that move horizontally back and forth across the screen as they advance toward the bottom of the screen.

The player earns points by shooting it with the laser cannon. As more aliens are defeated, then alien's movement and the game's music both speed up. Defeating the aliens brings another wave that is more difficult. This is endless loop.

The aliens attempt to destroy the cannon by firing at it while they approach the bottom of the screen. If they reach the bottom, the alien invasion is successful, and the game ends.

A special *mystery ship* will occasionally move across the top of the screen and award bonus points if destroyed.

The laser cannon is partially protected by several stationary defense bunkers, that are

gradually destroyed by numerous blasts from the aliens or player. A game will also end if the player's last laser base is destroyed.

An invader can have a maximum of 3 missiles on screen at any one time. The player can have only 1 laser shoot on scree at any one time.

The invaders travel sideway and each time they touch the side they drop down 1 line. On screen 1 they need to drop 11 lines to reach the bottom and 'invade'. From screen 2 through to screen 9 they start progressively lower down the screen. At screen 10 the game reverts to the screen 1 start position and the cycle begins again.

2. Scoring

Players earn an extra life at either 1,000 or 1,500 points, but no extra lives thereafter. The high score is maxed at 9,990 points – players may exceed this score, but the game keeps the last four digits.

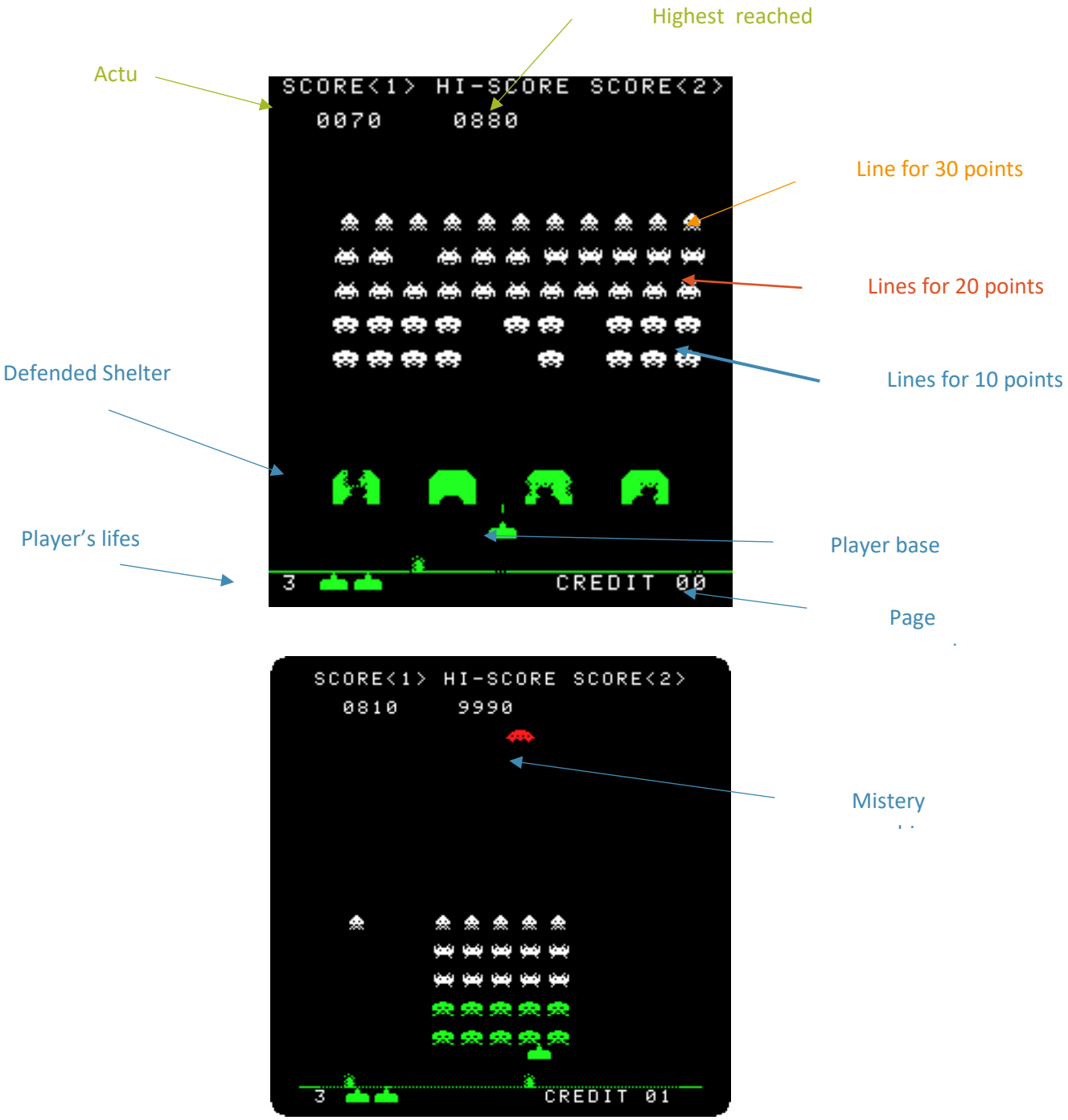
Alien invaders:

- 10 (top two rows),
- 20 (middle and second from top rows)
- 30 (bottom row)

Mystery ship:

- 100, 150, 200, 250, 300, 350 or 400 points (randomly)

3. Design of gameboard



3. Requirements

Following the above description, it is required to prepare the main elements of the game:

1. Graphics

All graphics are in pixel-style and all words too, so it is necessary to design them. Text has to be prepared in arrays in working memory.

2. Music

Soundtrack is important part of the gameplay, it creates atmosphere and improve experiences. It is necessary to implement usage of music with music volume control.

3. Gameplay

All basic elements and rules have been described in introduction above. It is necessary to implement main loop with 10 stages of *alien invasion*, use detection algorithm. All data about board have to been storage in matrix, and map have to been generated based on it.

To play we need 3 keys – arrows (left, right) for movement and one key for shooting from laser cannon. Another key may be implemented to end gameplay with reached score without dying.

At beginning to start player has to press any key, so all of them must been implemented.

4. Methodology

1. Research [5 hours]

In this stage I am going to explore web for some tutorials, help about rules, technics, and resources like music, graphics ideas.

2. Implementation [40 hours]

In this stage I am going to create game from scratch:

1. Creating board [5]
2. Implement keys usage [5]
3. Implement movement [5]
4. Implement player's and alien's shooting [5]
5. Implement Mistery ship event [5]
6. Implement scoring system [5]
7. Implement main loop with 10 stages [5]
8. Implement music [5]

3. Testing [2 hours]

In this stage I am going to test every possible action which can be done by players, and behaviors on original console.

4. Documentation [5 hours]

In this stage I am going to prepare documentation from my whole work attached results and all problems from all stages. All resources will be collecting during whole implementation process.

3. Memory of the work

1. Description

First, I need to mention, that originally, I declared to do game Space Invaders for console 3DS, but during the course I realized that all exercises were based on console NDS, so finally, according that I don't have so much time for learning difference between them, changed the console from 3DS to NDS.

1. Architecture

The project is created for console NDS, based on library libnds. Emulator which I used to test my application was Desmume. The GUI which I used was the simplest Programmer Notepad attached to devkitPro package.

2. First step

At the beginning of my work, I had to plan all steps, collect resources. Most of that information are included in the first part of that document – Scope of work. The other resources which I found are: the video which presents original gameplay[1], webpage[2] with more detailed information about the gameplay and some graphics and music resources.

2. Creating screens

1. Description

As you can see in the original gameplay, there were 3 main screens. First two, were only based on text mode to display welcome message, inform the player about the score for each invader ship, printing all words letter by letter.

The third screen was the main screen with game board, all ships, bases, lines. I decided to modify a little that screen – put all text information at the top of the lower screen and all sprites without any text on the top screen.

[1] Look for the link in the bibliography
[2] Webpage Classic Games

2. Structures, usages

To be able to use the console for the printing text the first thing which I had to done was to initialize console mode.

```
void initConsole() {
    consoleDemoInit();
}
```

To implement effect of spelling I needed to create some structs. First one called CMessage with flag state to know that message is printed whole or not, the array of characters to keep the message, the index of the actually printed character, the velocity of the animation (it is connected to the timer options), and the information about position of that message: line column where the cursor should be placed.

```
struct CMessage {
    u8 state;                       // running dialog flag
    char text[MAX_LINE_SIZE];       // buffer
    u8 current_char;                // index of current char in dialog_buffer
    u8 velocity;                    // writing velocity
    u8 cursor_col0;                 // initial column
    u8 cursor_line0;                // initial line
    u8 cursor_col;                  // writing column
    u8 cursor_line;                 // writing line
};
```

I also defined two different modes of printing message: character by character, and the second one to print whole message at once.

```
void printMessage(struct CMessage *message) {
    if(message->state) {
        iprintf("\x1b[%d;%dH%c", message->cursor_line,
            message->cursor_col, message->text[message->current_char]);
        message->current_char++;
        message->cursor_col++;
        message->state = (message->text[message->current_char] != '\0');
    }
}
```

```
void printWholeMessage(struct CMessage *message) {
    if(message->state) {
        iprintf("\x1b[%d;%dH%s", message->cursor_line,
            message->cursor_col, message->text);
        message->state = 0;
    }
}
```

Later I needed to implement one more type of printing message to display the score (in generally to display numbers, passing the reference to it).

```
void printScoreMessage(struct CMessage *message, int* score) {
    if(message->state) {
        iprintf("\x1b[%d;%dH%4d", message->cursor_line,
            message->cursor_col, *score);
        message->state = 0;
    }
}
```

I also needed to change color of the font, clear whole console, clear one line and change the text in the struct, I defined some extra method based on escape ASCII characters.

```
void setColor(u8 color) {
    iprintf("\x1b[3%d;1m", color);
}

void setText(char* text,struct CMessage *message) {
    strcpy(message->text, text);
}

void clearConsole() {
    iprintf("\x1b[2J");
}

void clearLine(u8 line) {
    iprintf("\x1b[%d;0H\x1b[K", line);
}
```

When I had defined messages and methods to print them on the console, I needed to create another struct for each screen. Each struct has another structs of CMessage type – as many as unique lines I wanted to print. The screen structs have also counter msg which defined how many CMessage has been printed and flag state to inform that whole screen has been printed or not.

All animations are based on the timer with the frequency based on the velocity from each CMessage struct.

All screens are initialized at the beginning of the program, before main loop, and depends of the game stage, the screen is displayed and animated or not (depends on their destiny).

3. Screen 1 (welcome screen)

```
struct Screen1 {
    struct CMessage msg_play;
    struct CMessage msg_title;
    struct CMessage msg_scoreHeader;
    struct CMessage msg_scoreMystery;
    struct CMessage msg_score30;
    struct CMessage msg_score20;
    struct CMessage msg_score10;
    struct CMessage msg_next;
    int msg;
    int state;
};
```

Struct Screen1 is assigned to variable title_screen. All methods witch are connected to that screen are:

```
void initTitleScreen();

void animateTitleScreen();

void changeTitleScreenVelocity(u8 v) {
    endAnimation();
    timerStart(timer_id, ClockDivider_1024, TIMER_FREQ_1024(v), animateTitleScreen);
}
```

Which is response for changing timer frequency based on the velocity from each message. That method allow me to display some lines faster or slower than another.

The last method is

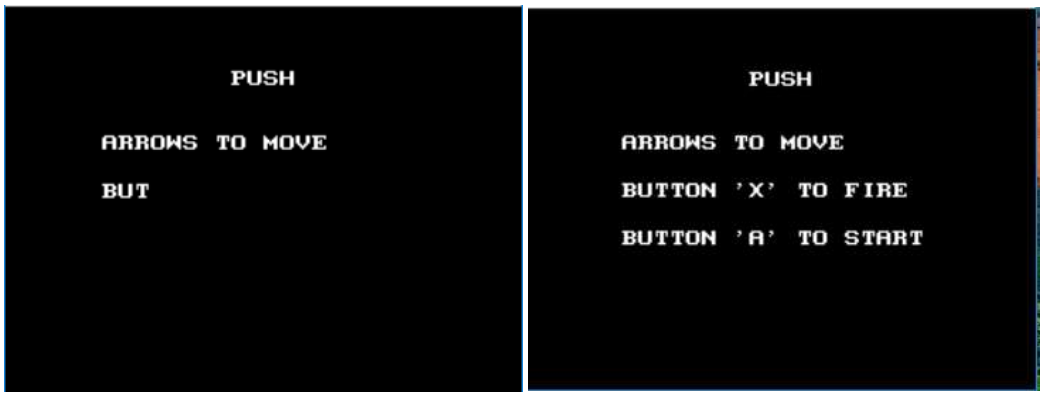
```
void endAnimation() {
    timerStop(timer_id);
}
```

Which is response to stop the timer at the end of printing whole screen or inside methods for changing velocity. The idea is to stop actually running timer and run it again in new frequency.

4. Screen 2 (control information screen)

```
struct Screen2 {
    struct CMessage msg_push;
    struct CMessage msg_arrows;
    struct CMessage msg_buttonX;
    struct CMessage msg_next;
    int msg;
    int state;
};
```

The idea of the struct is exactly the same as for the previous one, and all methods are similar, so I am not going to attached it here.



5. Screen 3 (game screen)

```
struct Screen3 {
    struct CMessage msg_scoreLabel;
    struct CMessage msg_hiScoreLabel;
    struct CMessage msg_score;
    struct CMessage msg_hiScore;
    int msg;
    int state;
    int velocity;
    int score;
    int hiScore;
};
```

Game screen has little different structure, because it is screen when the game is displayed during the main action. This screen is response to printing the labels for the score and the score value.

This screen is printed in the second mode which means that all messages are printed whole. Because the idea is to inform the player about the actual score which he reached playing game, this screen has to be refreshed. For that functionality I implemented method:

```
void refreshGameScreen() {
    game_screen.msg_score.state=1;
    game_screen.msg_hiScore.state=1;
    printWholeMessage(&game_screen.msg_scoreLabel);
    printWholeMessage(&game_screen.msg_hiScoreLabel);
    printScoreMessage(&game_screen.msg_score, &game_screen.score);
    printScoreMessage(&game_screen.msg_hiScore, &game_screen.hiScore);
}
```

For refresh I only need to set the state of messages score and hiScore and print them again. This method is set as timer callback.

To modify the score which is displayed I prepared method addScore which modify the value inside the struct depends on the value witch is passed as an argument. That method is called inside main loop, every time when invader's ship is destroyed.

```
void addScore(int i) {
    game_screen.score+= i;
}
```

6. Screen 4 (end game screen)

```
struct Screen4 {
    struct CMessage msg_end;
    struct CMessage msg_scoreLabel;
    struct CMessage msg_score;
    struct CMessage msg_newGame;
    int msg;
    int state;
    int velocity;
    int score;
};
```

This is the last screen struct. The idea is to display it after the game to passed the score which player reached and print it with the information how to start again.

At this stage I also update the hiScore which is reached during that game session and pass that information to the game screen (Screen3) which will be displayed again when the player wants to play again.

```
void finalScore() {
    end_screen.score=game_screen.score;
    game_screen.score=0;
}

void getHiScore() {
    if(game_screen.hiScore<end_screen.score)
        game_screen.hiScore=end_screen.score;
}
```

3. Implement key usage

1. Usage description

In the game there are only 4 keys which have influence on the game.

A key is response for starting game, going to the next screen.

X key is response for shooting from the player ship.

Arrows <left, right> are response for players moving.

```
void readKeys() {
    int held;
    scanKeys();
    held = keysHeld();
    if( held & KEY_X) {
        player_fire();
    }

    if( held & KEY_LEFT) {
        player_moveL();
    }

    if( held & KEY_RIGHT) {
        player_moveR();
    }

}
```

That method is called inside the main loop of the game and based on the keys it calls another methods for modify player object or bullets objects.

When the main part of the game is not started yet, there are another method which waits for the A key.

```
void waitForA() {
    int held;
    scanKeys();
    held = keysHeld();
    if( held & KEY_A) {
        active_screen++;
    }
}
```

4. Objects

1. Player

a. Struct

```
struct Player {
    int lifes;
    int x;
    int y;
    int width;
    int height;
    int moving;
    int id;
    Bullet* bullet;
};

struct Player ship;
```

The main struct which keeps all information about the player is showed above. The variable lifes keeps value from 0-3 and define how many lifes player has. The x, y describe the position on the map. Width, height describe how big the player sprite is (to be honest I don't remember which I am using that values or taking them from another method). The variable moving describe the state of moving (not sure too, if I am using it). The id keeps the index of that object in the array of sprites from the OAM. The last one is the pointer to the bullet object which is always player bullet, and can be only one on the stage.

b. Initialization

```
void initPlayer(){
    ship.lifes = 3;
    ship.x = BOARD_WIDTH/2 - 8;
    ship.y = BOARD_HEIGHT-P_Y_POSITION;
    ship.width= 16;
    ship.height= 8;
    ship.moving = 0;
    ship.id=0;
    ship.bullet=&bullets[0];

    ship.bullet->type=1;
    ship.bullet->dy=1;
}
```

To init the player object I defined the initPlayer method called once before the main loop. It sets all the variables inside the struct object and connect the bullet object with the player and set it's type and position too.

c. Updating

Updating player object is done by set of methods, called in different moments, depends on the context. 3 of the methods were mentioned before (moving in the left, right direction and shooting).

```
void player_moveL() {
    if(ship.x>0)
        ship.x--;
}

void player_moveR() {
    if(ship.x< (BOARD_WIDTH-ship.width))
        ship.x++;
}

void player_fire(){
    //any player's bullet
    if(ship.bullet->state==0) {
        //create new bullet (initialize their states)
        ship.bullet->x=ship.x+6;
        ship.bullet->y=ship.y+8;
        ship.bullet->state=1;
    }
}
```

There are also 2 methods where are used when the player's ship is destroyed.

```
void killPlayer() {
    ship.lifes--;
    explosionSounds=true;
    if(ship.lifes==0)
        //gameOver();
        game_stage++;
    removeBullets();
    centerPlayer();
    refreshLifes();
}

void centerPlayer() {
    ship.x =  BOARD_WIDTH/2 - 8;
}
```

The main part at this moment for us are the lines which modifies ship object.

2. Invader

a. Struct

```
typedef struct {
    int id;
    int type;
    int x;
    int y;
    int dy;
    int shoot;
    int movement;
    int state;
} Invader;
```

The struct Invader keep all necessary informations: the id which is the index in the sprite array, the type of the Invader (this define the sprite, points), position, state of shooting (probably not even used), movement state (probably not used too – done by static variable), and the state that describe is this Invader still 'alive' or not.

b. Initialization

Initialization is done once at the beginning of the game stage, and every time when the game is refreshed or the wave is done and the next came. It is simple hand-defined implementation of array of the objects.

```
Invader invaders[] = {
        {1, 0, 7, 15, 0, 0, 0,1 },
        {2, 0, 26, 15, 0, 0, 0,1 },
        {3, 0, 45, 15, 0, 0, 0,1 },
        {4, 0, 64, 15, 0, 0, 0,1 },
        {5, 0, 83, 15, 0, 0, 0,1 },
        {6, 0, 102, 15, 0, 0, 0,1 },
        {7, 0, 121, 15, 0, 0, 0,1 },
        {8, 0, 140, 15, 0, 0, 0,1 },
        {9, 0, 159, 15, 0, 0, 0,1 },
        {10, 0, 178, 15, 0, 0, 0,1 },
        {11, 0, 197, 15, 0, 0, 0,1 },

        {12, 0, 7, 34, 0, 0, 0,1 },
        {13, 0, 26, 34, 0, 0, 0,1 }
```

Also there are some global information about the Invaders which I had to set:

```
int invaderMoves = 0;
int invadersDirection = 1;
int invadersBullets = 0;
int killedInvaders = 0;
```

They are describe the number of displacements, the direction of the moving, how many invader's bullets are active at that moment and how many Invaders are killed by player.

c. Updating

Updating invaders is done by the method set as a timer callback.

```
void updateInvaders() {
    invadersSounds=true;

    if(invadersDirection==1) {
        if(invaderMoves<MAX_HOR_MOVES) {    //moving right
            for(int i=0; i<55; i++) {
                if(invaders[i].state==1)
                    invaders[i].x++;
            }
            invaderMoves++;
        } else {                            //moving down
            for(int i=0; i<55; i++) {
                if(invaders[i].state==1)
                    invaders[i].y+=3;
            }
            invaderMoves=0;
            invadersDirection=-1;
        }

    } else if(invadersDirection == -1) {

        if(invaderMoves<MAX_HOR_MOVES) {    //moving left
            for(int i=0; i<55; i++) {
                if(invaders[i].state==1)
                    invaders[i].x--;
            }
            invaderMoves++;
        } else {                            //moving down
            for(int i=0; i<55; i++) {
                if(invaders[i].state==1)
                    invaders[i].y+=3;
            }
            invaderMoves=0;
            invadersDirection=1;
        }

    }
```

The first part of that method define the way of moving the Invaders. Depends of the variables which I described before, all Invaders objects are modified. The first line will be describe later.

```
    int bulletChance = rand()%100;
    if((invadersBullets < MAX_I_SHOOTS) && (bulletChance < BULLET_I_CHANCE)) {
        invadersBullets++;
        int freeBullet = 0;
        for(int i=1; i<4; i++){
            if(bullets[i].state==0) {
                freeBullet=i;
                break;
            }
        }

        //find living free invader
        int k = 0;
        int invaderID =0;
        while(k==0) {
            invaderID = rand()%55;
            if(invaders[invaderID].state==1)
                k++;
        }


        bullets[freeBullet].state = 1;
        bullets[freeBullet].y = invaders[invaderID].y+4;
        bullets[freeBullet].x = invaders[invaderID].x+4;
    }


    updateMystery();

}
```

In the second part of that method I implemented invaders shooting. First I check the probability of the shoot. Then if there is any free bullet (only 3 shoots are allowed at the same time) I look for the Invader which shoot (it must be alive) and then create new bullet in the position of that Invader.

At the end I run also method updateMystery which I explain later.

3. Bullet

a. Struct

```
typedef struct {
    int id;
    int type;
    int x;
    int y;
    int dy;
    int state;
} Bullet;
```

The struct for the bullets is very simple. Each bullet has the id which describe the index from the sprite array, the type of the bullet (it define the sprite which will be used for the bullet), position information and the state, that it is active bullet or not.

b. Initialization

Initialization is done at the begin, before main method by hand, to let connect the player object with the bullet.

```
Bullet bullets[] = {
    {59,0,0,0,0,0},        //player shoot
    {60,0,0,0,0,0},        //1st invader shoot
    {61,0,0,0,0,0},        //2st invader shoot
    {62,0,0,0,0,0}         //3st invader shoot
};
```

c. Updating

Bullet's updating is done once only for player bullet and in another method for all Invaders bullet for easier distinguish the actions which have to be done when the bullet hit something.

```
void updatePlayerBullet() {
    int invaderId=1;
    int mysteryCol = 0;
    int bulletId = -1;
    //player bullet
    if(bullets[0].state==1) {
        //bullet sound
        bulletSounds=true;

        if(bullets[0].y>0) {
            bullets[0].y-=2;
            //detect collisions
            invaderId = checkCollisionWithInvaders(bullets[0].x, bullets[0].y);
            if(invaderId!= -1) {
                killInvaderById(invaderId);
                bullets[0].state=0;
                bulletSounds=false;
            }

            //detectCollision with mystery
            mysteryCol = checkCollisionWithMystery(bullets[0].x, bullets[0].y);
            if(mysteryCol == 1) {
                killMystery();
                bullets[0].state=0;
                bulletSounds=false;
            }

            //detectCollision with invadersBullets
            bulletId = checkCollisionWithBullets(bullets[0].x, bullets[0].y);
            if(bulletId!=-1) {
                killBulletById(bulletId);
                bullets[0].state=0;
                bulletSounds=false;
            }

        } else {
            //hit border
            bullets[0].state=0;
            bulletSounds=false;

        }
    } else {
        bullets[0].y=300;
        bulletSounds=false;
    }
}
```

I method updatePlayerBullet first I check that bullet is active, so it means that it's position must be updated. It also update the sound state. When the bullet is moving there are also checked all possible collisions (with the invaders, with the mystery ship and with another bullet). For each of that case the bullet is destroyed, the sound is turned off, and the special action is performed (for example the Invader is killed).

At last case there is checked the collision with the border – then the bullet is moved out the map and is killed.

```
void updateInvaderBullets() {
    int id = -1;
    //invader bullets
    for(int i=1; i<4; i++) {
        if(bullets[i].state==1) {
            if(bullets[i].y<ship.y+8) {
                bullets[i].y++;
                id = checkCollisionWithPlayer(bullets[i].x, bullets[i].y);
                if(id!=-1) {
                    killPlayer();
                    bullets[i].state=0;
                }

            } else {
                //hit border
                bullets[i].state=0;
                invadersBullets--;
            }
        } else {
            bullets[i].y=300;
        }
    }
}
```

The case for the invader bullet is simpler, because I only need to check the collision with the player and the border. The general logic is the same in the player bullet case. The difference is that there is no sound effect for the invaders bullets.

4. Mystery (vip invader)

a. Struct

```
struct Mystery {
    int id;
    int type;
    int x;
    int y;
    int state;
    int value;
};
```

The mystery struct keeps information about the id, which is the index in the sprite array, the type which define the sprite which will be used for that object, the position, state (is any mystery ship alive) and the score which player reach for shooting it down.

b. Initialization

```
void initMystery() {
    mystery.id=63;
    mystery.type=3;
    mystery.x = BOARD_WIDTH + ATTR1_SIZE_16;
    mystery.y = 2;
    mystery.state = 0;
    mystery.value = 0;
}
```

The initialization is done once before the main loop of the game.

c. Updating

Method updateMystery was mentioned before (it is called always inside updateInvaders method).

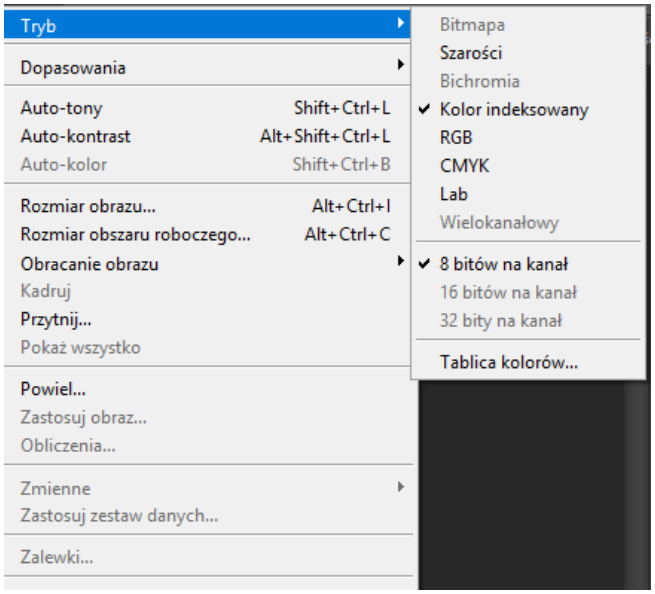
```
void updateMystery() {
    if(mystery.state==0) {
        mysterySounds=false;
        int r = rand() % 10000;
        if(r<10) {
            mystery.x=BOARD_WIDTH;
            int val = (rand() % 6) * 50 + 100;
            mystery.value = val;
            mystery.state=1;
        }
    } else {
        mysterySounds=true;
        //moving
        if(mystery.x>=-16) {
            mystery.x-=3;
        } else {
            mystery.state=0;
        }
    }
}
```

This method (if there is any mystery ship on the board) check the probability for the mystery appearance and if the test passed, the mystery ship status is set as alive. Then the sound status is set and mystery ship position is modified.

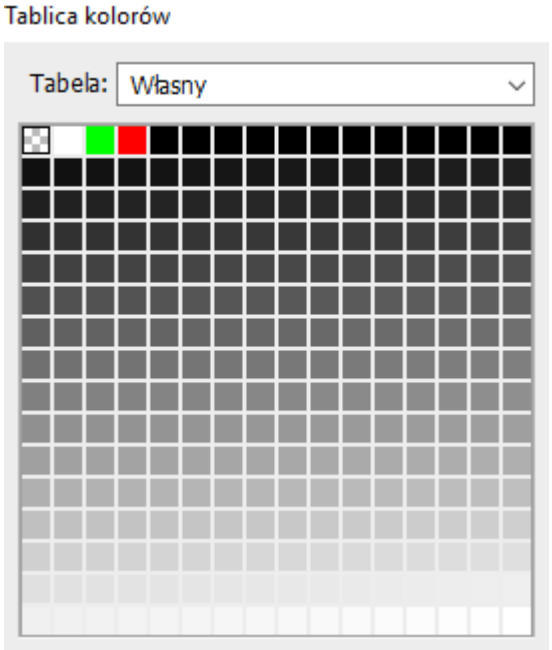
5. Sprites

1. Description

All sprites which I used are made by me in the graphic editor. The mode of the images is Indexed color (Tryb->Kolor indeksowany) with 8-bit color per canal (8 bitów na kanał).



The color pallete is defined with the opacity for the black color (because the background color is black):



All images are save in .png extension in size of 16x16 px but the real images are smaller. I needed to keep information about their real size in the game, so I defined special method with return each width and height, based on the type of the sprite. Here is one of that methods.

```
int getInvaderWidth(int type) {
    switch(type) {
        case 0:
            return 8;
            break;
        case 1:
            return 12;
            break;
        case 2:
            return 16;
            break;
        case 3:
            return 16;
            break;
        default:
            return 0;
    }
}
```

2. Structures, images

When the make is performing, based on all images and one gfx file, there are created headers files for each image using the grit script.

It was necessary to add following line to the MakeFile with the rule for the png files.

```
#---------------------------------------------
%.s %.h : %.png
    grit $< -ff../sprites/sprite.grit -o$*
#---------------------------------------------
```

This files are included at the beginning of the main.c file.

```
#include <gfx_ship.h>
#include <gfx_invader1.h>
#include <gfx_invader2.h>
#include <gfx_invader3.h>
#include <gfx_mystery.h>
#include <gfx_fire1.h>
#include <gfx_fire2.h>
#include <gfx_fire3.h>
```

3. Sprites initialization

To initialize sprites I need to define spriteEntry struct and create pointer to the place in memory responsible for the sprites. It is done in the code below (the comments belowe sprites array define which indices I will be using for all in game objects

```
typedef struct t_spriteEntry
{
    u16 attr0;          // position y (board size - sprite size)
    u16 attr1;          // position x (board size - sprite size) + attr1_size_16;
    u16 attr2;          // tile and pallete (tile + (pal << 12))
    u16 affine_data;    //
} spriteAttrEntry;

#define sprites ((spriteAttrEntry*)OAM)
//sprites[0] - ship
//sprites[1-55] - invaders
//sprites[56-58] - lives
//sprites[59-62] - shoots
//sprites[63] - mystery
```

I also need to define the tiles and the pallet indices

```
#define tiles_ship 0
#define tiles_invader1 1
#define tiles_invader2 2
#define tiles_invader3 3
#define tiles_fire1 4
#define tiles_fire2 5
#define tiles_fire3 6
#define tiles_mystery 7

#define pal_ship 0
#define pal_invader1 1
#define pal_invader2 2
#define pal_invader3 3
#define pal_fire1 4
#define pal_fire2 5
#define pal_fire3 6
#define pal_mystery 7
```

Also I needed to prepare some macros

```
#define tile2bgram(t) (BG_GFX + (t) *16)
#define pal2bgram(p) (BG_PALETTE + (p) * 16)
#define tile2objram(t) (SPRITE_GFX + (t) * 16)
#define pal2objram(p) (SPRITE_PALETTE + (p) * 16)
#define tile4(t) ((t) * 4)
#define backdrop_colour RGB8( 0, 0, 0 )
```

Which are helpful for copy data to the memory and to set the background color.

When it has been done, I need to setup graphic modes and copy all information to the memory.

```
void setupGraphics(void) {
    vramSetBankE( VRAM_E_MAIN_BG );
    vramSetBankF( VRAM_F_MAIN_SPRITE );

    dmaCopyHalfWords( 3, gfx_shipTiles, tile2objram(tile4(tiles_ship)), gfx_shipTilesLen );
    dmaCopyHalfWords( 3, gfx_invader1Tiles, tile2objram(tile4(tiles_invader1)), gfx_invader1TilesLen );
    dmaCopyHalfWords( 3, gfx_invader2Tiles, tile2objram(tile4(tiles_invader2)), gfx_invader2TilesLen );
    dmaCopyHalfWords( 3, gfx_invader3Tiles, tile2objram(tile4(tiles_invader3)), gfx_invader3TilesLen );
    dmaCopyHalfWords( 3, gfx_fire1Tiles, tile2objram(tile4(tiles_fire1)), gfx_fire1TilesLen );
    dmaCopyHalfWords( 3, gfx_fire2Tiles, tile2objram(tile4(tiles_fire2)), gfx_fire2TilesLen );
    dmaCopyHalfWords( 3, gfx_fire3Tiles, tile2objram(tile4(tiles_fire3)), gfx_fire3TilesLen );
    dmaCopyHalfWords( 3, gfx_mysteryTiles, tile2objram(tile4(tiles_mystery)), gfx_mysteryTilesLen );

    dmaCopyHalfWords( 3, gfx_shipPal, pal2objram(pal_ship), gfx_shipPalLen );
    dmaCopyHalfWords( 3, gfx_invader1Pal, pal2objram(pal_invader1), gfx_invader1PalLen );
    dmaCopyHalfWords( 3, gfx_invader2Pal, pal2objram(pal_invader2), gfx_invader2PalLen );
    dmaCopyHalfWords( 3, gfx_invader3Pal, pal2objram(pal_invader3), gfx_invader3PalLen );
    dmaCopyHalfWords( 3, gfx_fire1Pal, pal2objram(pal_fire1), gfx_fire1PalLen );
    dmaCopyHalfWords( 3, gfx_fire2Pal, pal2objram(pal_fire2), gfx_fire2PalLen );
    dmaCopyHalfWords( 3, gfx_fire3Pal, pal2objram(pal_fire3), gfx_fire3PalLen );
    dmaCopyHalfWords( 3, gfx_mysteryPal, pal2objram(pal_mystery), gfx_mysteryPalLen );


    //asign color to background
    BG_PALETTE[0] = backdrop_colour;
    videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE | DISPLAY_BG1_ACTIVE | DISPLAY_SPR_ACTIVE | DISPLAY_SPR_1D_LAYOUT );

    for(int n = 0; n < 128; n++ ) {
        sprites[n].attr0 = ATTR0_DISABLED;
    }
```

In that part I set the memory response for the background and the sprites, copied all tiles and pallet information to the memory.

```
    //ship (id 0)
    sprites[0].attr0 = ship.y;
    sprites[0].attr1 = ship.x + ATTR1_SIZE_16;
    sprites[0].attr2 = tile4(tiles_ship) + (pal_ship << 12);


    //invaders (id 1-55)
    for(int n = 0; n <55; n++) {
        int id = invaders[n].id;
        sprites[id].attr0 = invaders[n].y;
        sprites[id].attr1 = invaders[n].x + ATTR1_SIZE_16;

        if(invaders[n].type == 0)
            sprites[id].attr2 = tile4(tiles_invader1) + (pal_invader1 << 12);
        if(invaders[n].type == 1)
            sprites[id].attr2 = tile4(tiles_invader2) + (pal_invader2 << 12);
        if(invaders[n].type == 2)
            sprites[id].attr2 = tile4(tiles_invader3) + (pal_invader3 << 12);

    }

    //lifes (56-58)
    for(int i=0; i<ship.lifes; i++) {
        sprites[56+i].attr0 = BOARD_HEIGHT - 16;
        sprites[56+i].attr1 = 1+i*17 + ATTR1_SIZE_16;
        sprites[56+i].attr2 = tile4(tiles_ship) + (pal_ship << 12);
    }


    //bullets
        //sprites[59].attr0 = BOARD_HEIGHT;
        //sprites[59].attr1 = BOARD_WIDTH + ATTR1_SIZE_16;
    for(int i=0; i<4; i++) {
        sprites[59+i].attr0 = -20;
        sprites[59+i].attr1 = 0;
        sprites[59+i].attr2 = tile4(tiles_fire1) + (pal_fire1 << 12);
    }

    //mystery ship
    sprites[63].attr0 = 2;
    sprites[63].attr1 = BOARD_WIDTH + ATTR1_SIZE_16;
    sprites[63].attr2 = tile4(tiles_mystery) + (pal_mystery << 12);

}
```

In the second part of that method I initialize first positions for each sprite object.

The same code will be execute for updating sprites but with different values.

4. Sprites updating

```
void updateGraphics(void) {
    //player
    sprites[0].attr0 = ship.y;
    sprites[0].attr1 = ship.x + ATTR1_SIZE_16;

    //invaders (id 1-55)
    for(int n = 0; n <55; n++) {
        int id = invaders[n].id;
        if(invaders[n].state==1) {
            sprites[id].attr0 = invaders[n].y;
            sprites[id].attr1 = invaders[n].x + ATTR1_SIZE_16;
        } else {
            sprites[id].attr0 = -16;
            sprites[id].attr1 = 0;
        }
    }


    //player bullet
    sprites[59].attr0 = ship.bullet->y;
    sprites[59].attr1 = ship.bullet->x + ATTR1_SIZE_16;

    //invader bullets
    for(int i=1; i<=MAX_I_SHOOTS; i++) {
        sprites[59+i].attr0 = bullets[i].y;
        sprites[59+i].attr1 = bullets[i].x + ATTR1_SIZE_16;
    }

    //mystery
    if(mystery.state==1) {
        sprites[mystery.id].attr0 = mystery.y;
        sprites[mystery.id].attr1 = mystery.x  + ATTR1_SIZE_16;
    } else {
        sprites[mystery.id].attr1 = BOARD_WIDTH + ATTR1_SIZE_16;
    }


}
```

Here is the simple code for updating all sprites attributes (position and size) for each object.

5. Collisions detection

In my project I implemented two collision detection algorithms. First one which check the collision based on the distance between circles (the center point of each circle is inside of each sprite. This method is not the best with the long rectangle objects when it is used as only method to check the collision. I am using it to safe resources and if it detect collision only for the player bullet (because there is a lot invaders ship), then I check it again using checking pixel per pixel but only for the borders. Of course to check the collision with the border of the map

I only check the vertical position. In another cases I used simple collisions based on the corners of the object, treats all as a rectangle objects.

What I wanted to implement pixel per pixel collision detection mechanism with color comparation, but I haven't more time to learn how to extract that information from the memory or objects tiles.

```
int checkCollisionWithPlayer(int x,int y) {

    int bw, bh, ix, iy,px, py, pw, ph;
    pw = ship.width;
    ph = ship.height;

    for(int i = 1; i<3; i++) {
        //check if bullet still exists
        if(bullets[i].state==1) {
            ix = bullets[i].x;
            iy = bullets[i].y;

            bw = getBulletWidth(bullets[i].type);
            bh = getBulletHeight(bullets[i].type);

            px = ship.x;
            py = ship.y;

            if(( (ix >= px) && (ix <= px+pw) ) && (iy >= py+3) && (iy <= py + ph))
                return i;

            if(( (ix >= px) && (ix <= px+pw) ) && (iy+bh >= py+3) && (iy + bh <= py + ph))
                return i;

            if(( (ix + bw >= px) && (ix + bw <= px+pw) ) && (iy >= py+3) && (iy <= py + ph))
                return i;

            if(( (ix + bw >= px) && (ix + bw<= px+pw) ) && (iy+bh >= py+3) && (iy + bh <= py + ph))
                return i;
        }
    }
    return -1;
}
```

This part of the code is the example of simple collision detection based on the corners of the objects.

```
int checkCollisionWithInvaders(int x, int y) {
    int iw,ih, bw, bh, ir, br, ix, ixx, iy, iyy, bx, by, px1, px2, py1, py2;

    for(int i = 0; i<55; i++) {
        //check if invader still exists
        if(invaders[i].state==1) {
            ixx = invaders[i].x;
            iyy = invaders[i].y;

            //frist cirle collision detection
            //for little cost cutting
            //i know that bullet sprite is 8x8 sized but the
            //real image representation is 2x8 size and starts from left
            //it implicites that the center of that bullet is in (1,3) and
            //the radius is equal 4px
            //each invader has size
            iw=getInvaderWidth(invaders[i].type);
            ih=getInvaderHeight(invaders[i].type);
            bw=getBulletWidth(bullets[0].type);
            bh=getBulletHeight(bullets[0].type);

            //calculate center points
            ix = invaders[i].x + iw/2;
            iy = invaders[i].y + ih/2;
            bx = x+bw/2;
            by = y+bh/2;

            //calculate radius
            br = (int) sqrt( (bw*bw/4+bh*bh/4));
            ir = (int) sqrt( (iw*iw/4+ih*ih/4));
            //compare distance with radiuses
            if( sqrt (((ix-bx)*(ix-bx)) + ((iy-by)*(iy-by))) < (br+ir)) {
                //more detailed collision detect - rectangle collision (only borders)
                //the best one would be pixel collision detection with color comparing
                for(py1=0;py1<bh;py1++) {
                    //bottom, top
                    if((py1==0) || (py1==bh)) {
                        for(px1=0;px1<bw;px1++) {

                            //compare with border in invader
                            for(py2=0; py2<ih; py2++) {
                                //bottom,top
                                if((py2 == 0) || (py2==ih)) {
                                    for(px2=0; px2<iw; px2++) {
                                        //compare coliision

                                        if ((x + px1 == ixx + px2) &&
                                            (y + py1 == iyy + py2))
                                            //collision
                                            return i;

                                    }
                                } else {
```

This part is the first from the circle shape collision detection mechanism. First I need to extract all necessary information as positions, radiuses, center points of the circles to calculate the distance between them and compare It with the sum of the radiuses. This is the collision for the circle shaped objects. If this test pass, I start second collision detection which is modified pixel per pixel algorithm (I am only use the border pixels.

```
                    } else {

                        //invader left, right collision

                        if ((x + px1 == ixx) &&
                            (y + py1 == iyy + py2))
                                //collision
                                return i;

                        if ((x + px1 == ixx + iw) &&
                            (y + py1 == iyy + py2))
                                //collision
                                return i;


                    }
                }
            }
        } else {

            //left, right
            //compare collision with all borders in invader
            for(py2=0; py2<ih; py2++) {
                //bottom,top
                if((py2 == 0) || (py2==ih)) {
                    for(px2=0; px2<iw; px2++) {
                        //compare coliision for left
                            if ((x == ixx + px2) &&
                                (y + py1 == iyy + py2))
                                //collision
                                return i;

                        //compare coliision for right
                            if ((x + bw == ixx + px2) &&
                                (y + py1 == iyy + py2))
                                //collision
                                return i;
                    }
                } else {
```

This is the main idea of that algorithm. I am moving from top to the bottom of one object, and when I am comparing the first and last row I am moving throght all the pixels horizontally, if not, I am only using pixels from left and right border.

```
                } else {

                    //compare coliision for left
                    if ((x == ixx) &&
                        (y + py1 == iyy + py2))
                            //collision
                            return i;

                    //compare coliision for left
                    if ((x + bw== ixx) &&
                        (y + py1 == iyy + py2))
                            //collision
                            return i;

                    //compare coliision for right
                    if ((x == ixx + px2) &&
                        (y + py1 == iyy + py2))
                            return i;

                    //compare coliision for right
                    if ((x + bw == ixx + px2) &&
                        (y + py1 == iyy + py2))
                            return i;
                }
            }
        }
    }
    return -1;
}
```

In that part I am checking pixels from the first object from left and right border with the pixels from the second object placed also in the left and right border.

If the collision happened then I returned the id of the invader to call method for killing that Invader.

```
void killInvaderById(int id) {
    explosionSounds=!explosionSounds;
    invaders[id].state=0;
    addPointsByType(invaders[id].type);
    killedInvaders++;
    if(killedInvaders%10 == 0) {
        timerStop(invaderTimer_id);
        invaderTimer_vel+=5;
        timerStart(invaderTimer_id, ClockDivider_1024,
            TIMER_FREQ_1024(invaderTimer_vel), updateInvaders);

    }
}
```

Killing invader (or another object – the mechanism is the same) means to set it's status to inactive and add points, modifiy the sounds state. When player kill 10 Invaders then game speed up, so in that method I am also checking that.

6. Timers

1. Description and role

All timers are used to call callbacks method which may be described as main logical methods. They are updating positions of the objects, define when the sounds should be played, and they are responsible for refreshing the console with text informations.

In the project I am using all 4 hardware timers which are allowed.

Timer 0 updates the console.

Timer 1 updates all invaders and mystery objects.

Timer 2 updates all bullets

Timer 3 based on the flags modified in callback methods from another timers, plays sounds.

2. Initialization

```
//Timers.h
int invaderTimer_id = 1;
int invaderTimer_vel = 3;
int bulletTimer_id = 2;
int bulletTimer_vel = 80;
int soundsTimer_id = 3;
int soundsTimer_vel = 30;

//
void displayTitleScreen() {
    active_screen = 0;
    timer_id = 0;
    timerStart(timer_id, ClockDivider_1024,
        TIMER_FREQ_1024(title_screen.msg_play.velocity), animateTitleScreen);
}

//
void startTimers() {
    timerStart(invaderTimer_id, ClockDivider_1024,
        TIMER_FREQ_1024(invaderTimer_vel), updateInvaders);
    timerStart(bulletTimer_id, ClockDivider_1024,
        TIMER_FREQ_1024(bulletTimer_vel), updateBullets);
    timerStart(soundsTimer_id, ClockDivider_1024,
        TIMER_FREQ_1024(soundsTimer_vel), soundsCallback);
}
```

Because application can be split into two stages – only displaying text information and the second with the proper game I also split the timers initializations. The first timer with id 0 is initialized when the screen with text information have to be displayed, the other are initialized together, because all of them together update the game logic.

3. Management

Each timer has it' own velocity, which is modified inside another methods. When I was talking about screens there was one example (changeTitleScreenVelocity).

When I was killing the 10th Invader I was also modified the velocity of the Timer 1.

```
void refreshTimersFreq() {
    invaderTimer_vel = 3;
    bulletTimer_vel = 80;
    soundsTimer_vel = 30;
}

void stopTimers() {
    timerStop(invaderTimer_id);
    timerStop(bulletTimer_id);
    timerStop(soundsTimer_id);
}
```

This two methods are helpful to refresh the values of the variables (in new game case), or to stop all the timers (in game over case).

7. Sounds

1. Types

Because I didn't have more time, I decided to use only simple sounds. There are 2 types of sounds, the simple sound based on the frequency and the noise based on the frequency. It is possible to create nice sounds effects but it requires more time to experiment. So, in my case I am using sounds which are not the best one, but they are not intensive too.

2. Initialization

```
void initSounds() {
    soundEnable();

    /*

    int channel0 = 0;    //invaders moves
    int channel1 = 0;    //bullet sound
    int channel2a = 0;   //mystery noise
    int channel2b = 0;   //mystery sound
    int channel3 = 0;    //explosion
    */
    //0-127

    channel0 = soundPlayPSG(DutyCycle_50, invaderFreq, 4, 64);
    channel1 = soundPlayPSG(DutyCycle_50, bulletFreq, 2, 64);
    channel2a = soundPlayNoise(mysteryNoiseFreq, 10, 64);
    channel2b = soundPlayPSG(DutyCycle_50, mysterySoundFreq, 10, 64);
    channel3a = soundPlayNoise(explosionNoiseFreq, 10, 64);
    channel3b = soundPlayPSG(DutyCycle_50, explosionFreq, 10, 64);
}
```

To initialize the sounds I defined some channels and connect them with the sounds or noises.

```
void soundsCallback() {

    if(!invadersSounds) {
        soundPause(channel0);
    } else {
        soundResume(channel0);
        invadersSounds=!invadersSounds;
    }


    if(!bulletSounds) {
        soundPause(channel1);
    } else {
        soundResume(channel1);
    }

    if(!mysterySounds) {
        soundPause(channel2a);
        soundPause(channel2b);
        inverseMysteryFreq();
    } else {
        soundResume(channel2a);
        soundResume(channel2b);
    }

    if(explosionSounds) {
        soundPause(channel3a);
        soundPause(channel3b);
    } else {
        soundResume(channel3a);
        soundResume(channel3b);
        explosionSounds=!explosionSounds;
    }

}
```

This method is set as a callback for the sound timer and depends of the flags it displays or pause the sounds.

3. Management

```
void inverseMysteryFreq() {
    if(mysteryNoiseFreq==2500) {
        mysteryNoiseFreq = 100;
        mysterySoundFreq = 2500;
    } else {
        mysteryNoiseFreq = 2500;
        mysterySoundFreq = 400;
    }

    soundSetFreq(channel2a, mysteryNoiseFreq);
    soundSetFreq(channel2b, mysterySoundFreq);
}
```

I tried to create mystery sounds like a waving sound which is changing between to states, but the frequency of the timer is too high, so it sounds like one sound (without wave, mystery effect).

```
void soundsStop() {
    invadersSounds = false;
    bulletSounds = false;
    mysterySounds = false;
    explosionSounds = false;

    soundPause(channel0);
    soundPause(channel1);
    soundPause(channel2a);
    soundPause(channel2b);
    soundPause(channel3a);
    soundPause(channel3b);
}
```

To pause all sounds I created simple method. It is used mostly when the game is over.

4. Summary

1. What I missed

First, the most important thing which I missed in my version of that game is the board. I completely didn't have time for creating bases which protect the earth and the field around the earth.

I also didn't have time for implementing management of extra lifes for each 1000 or 1500 points.

What I mentioned before I also not implemented the best sounds effects and it is a big disadvantage especially because this is important part of the game experience.

The last thing which I wanted to do, and I didn't have time for that is to implement animated sprites when Invaders are moving and for the explosion when the bullet hit any object.

2. Last thoughts

It was amazing adventure, and great opportunity to try create any game, especially, without any experience.

5. Bibliography

[1.] Classic Games - http://www.classicgaming.cc/classics/space-invaders/play-guide

[2.] Space Invaders gameplay - https://www.youtube.com/watch?v=vYMent6mg_c

[3.] Documentation for libnds library

[4.] Grit GBA Raster Image Transmogrifier http://www.coranac.com/projects/grit/

[5.] Tutorial práctico para desarrollo de videojuegos sobre plataforma Nintendo NDS, Jose David Jaén Gomariz, UPV, 2015