YILDIZ TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BBG Final Project: Tetris Game

Student Name Surname: Amirkia Rafiei Oskooei
Student Number: [redacted]
E-mail: amirkia.oskooei@std.yildiz.edu.tr

December 2023

Introduction

This project delves into the creation of a simplified Tetris game using the C programming language. Tetris, a classic game originating from Alexey Pajitnov in the 1980s, involves strategically arranging falling shapes (Shapes) to complete lines and prevent screen overflow.

The primary goal is to gain practical insights into essential computer science principles, such as data structures, algorithms, and user interface design. The Tetris game serves as a practical application, allowing us to implement theoretical knowledge in a real-world context. This report outlines the project's conceptualization, design choices, challenges faced, and insights gained, offering a comprehensive view of the development process and the intersection of theoretical concepts with practical implementation in computer science.

Problem definition

The task at hand involves designing a program that enables users to engage in the iconic Tetris game—an addictive puzzle challenge renowned for its appeal in organizing chaos into order. Originating from the creative prowess of Alexey Pajitnov in 1984, Tetris has secured its place as one of the most successful and recognizable video games, captivating players across various gaming platforms.

The primary objective of the program is to replicate the Tetris gaming experience by allowing users to interact with a customizable game board. Upon initiation, the program prompts users to specify the size of the game board, setting the stage for the dynamic placement of randomly generated Shapes—distinct shapes integral to the Tetris gameplay.

Key features of the program include:

1. Creation of a user-friendly menu for game navigation.
2. Dynamic generation of Shapes in each round to sustain game variability.
3. User selection of the placement area for Shapes through specified coordinates.
4. Automatic placement of Shapes at the bottom of the game board based on user input.
5. Implementation of line-clearing mechanics to sustain gameplay fluidity.
6. Real-time calculation and display of points obtained from cleared lines.
7. Maintenance of a record for the highest score achieved during gameplay.
8. Termination of the game upon any Shape reaching the top line of the game board or upon user request.

In addressing this problem, the program must navigate the intricate balance between providing an engaging Tetris experience and ensuring logical coherence in its mechanics. The challenge lies in the successful execution of Tetris gameplay dynamics, encompassing piece manipulation, line clearance, and score calculation. To facilitate the development process, starting with simpler Shape configurations is recommended, coupled with the use of printf statements to visualize intermediate steps and identify

potential logical errors. The ultimate goal is to deliver a functional Tetris program that captures the essence of the original game while adhering to the principles of simplicity and logical integrity.

Methodology

This code provides a comprehensive structure for a Tetris game, incorporating functions to handle various aspects of gameplay, ensuring a functional and engaging experience for the user.

1. Libraries and Constants:

The code begins by including essential libraries and defining constants. Libraries like **stdio.h** and **stdlib.h** are used for input/output operations and dynamic memory allocation, respectively. **time.h** is included for utilizing the **rand()** function. The constant **MAX** is set to 16, representing the maximum size of the game board with an additional row acting as an obstacle.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 16  // one extra row for the last row to act as an obstacle
#define SIZE_SHAPES 3
```

2. Function Prototypes:

Before the **main** function, several function prototypes are declared. These serve as forward declarations, indicating to the compiler that these functions will be defined later in the code. Function prototypes include those responsible for obtaining the size of the game board, initializing the board, displaying the board and shapes, creating and rotating shapes, finding the first collision, moving shapes, updating the score, clearing filled rows, and checking for a game over.

```c
//################################################################################ PROTOTYPES
int getRowSize();
int getColSize();
void initializeBoard(int board[MAX][MAX], int m, int n);
void displayBoard(int board[MAX][MAX], int m, int n);
void displayShape(int currentShape[SIZE_SHAPES][SIZE_SHAPES]);
void createShape(int currentShape[SIZE_SHAPES][SIZE_SHAPES]);
void rotateShape(int currentShape[SIZE_SHAPES][SIZE_SHAPES]);
int findFirstCollision(const int board[MAX][MAX], const int currentShape[SIZE_SHAPES][SIZE_SHAPES], int m, int inputCol);
int moveShape(int board[MAX][MAX], const int currentShape[SIZE_SHAPES][SIZE_SHAPES], int startingRow, int inputCol, int n);
int updateScore(int board[MAX][MAX], int m, int n);
void clearFilledRow(int board[MAX][MAX], int filledRow, int n);
int gameOver(const int board[MAX][MAX], int n);
```

3. Main Function:

The **main** function initializes various variables that control the game's flow. These include variables for user selection, game status, scores, board size, and the current shape. It then enters a **do-while** loop, allowing the player to make menu selections until they decide to exit or start a new game.

4. Game Logic:

Inside the loop, the program checks whether a new shape is needed. If so, a random Tetris shape is generated using the **createShape** function. The current state of the game, including the board and the incoming shape, is displayed. The user is prompted to choose from the menu: play, rotate the shape, or exit.

```
219     //---------------------------------------------------- SELECT RANDOM SHAPE
220         srand( Seed: time( Time: NULL));
221         int randomNum = rand() % 6;
```

- **Playing the Game:**
    - If the player chooses to play, they input a column where they want the shape to be placed.
    - The program utilizes the **findFirstCollision** function to determine the suitable row for the shape.

```
300 int findFirstCollision(const int board[MAX][MAX], const int currentShape[SIZE_SHAPES][SIZE_SHAPES], int m, int inputCol) {
301
302     int obstacleFound = 0, startingRow = 0;
303
304     // decrease the shape until it hits an obstacle
305     while (!obstacleFound) {
306         for (int i = 0; i < SIZE_SHAPES; ++i) {
307             for (int j = 0; j < SIZE_SHAPES; ++j) {
308                 if (currentShape[i][j] == 1 && board[startingRow + i][inputCol + j] == 1) {
309                     obstacleFound = 1;
310                 }
311             }
312         }
313
314         if(!obstacleFound)
315             startingRow++;
316     }
317
318     return --startingRow;
319 }
```

- The shape is then moved to the board using the **moveShape** function, and the program checks for game over or successful moves.

- **Rotating the Shape:**
    - If the player chooses to rotate the shape, it undergoes a rotation transformation using the **rotateShape** function.

```
281 void rotateShape(int currentShape[SIZE_SHAPES][SIZE_SHAPES]) {
282     int temp[SIZE_SHAPES][SIZE_SHAPES];
283
284     // Copy the original shape to a temporary matrix
285     for (int i = 0; i < SIZE_SHAPES; ++i) {
286         for (int j = 0; j < SIZE_SHAPES; ++j) {
287             temp[i][j] = currentShape[i][j];
288         }
289     }
290
291     // Rotate the shape by copying values from the temporary matrix to the rotated position
292     for (int i = 0; i < SIZE_SHAPES; ++i) {
293         for (int j = 0; j < SIZE_SHAPES; ++j) {
294             currentShape[i][j] = temp[SIZE_SHAPES - 1 - j][i];
295         }
296     }
297 }
```

5. Functions:

- **getRowSize and getColSize:** Prompt the user for the size of the game board, ensuring it does not exceed the defined maximum.
- **initializeBoard:** Sets up the initial state of the game board by assigning zeros and obstacles. (the bottom of the board acts as an obstacle that's why it is initialized with 1's)
- **createShape:** Generates a random Tetris shape for the player to control from a predefined set of shapes.
- **displayShape and displayBoard:** Print the current Tetris shape and the game board, respectively.

- **moveShape:** Moves the Tetris shape to the specified location on the board.

```
328 int moveShape(int board[MAX][MAX], const int currentShape[SIZE_SHAPES][SIZE_SHAPES], int startingRow, int inputCol, int n) {

330     // check if the 1's (the shape) fits in the board
331     for (int i = 0; i <SIZE_SHAPES; ++i) {
332         for (int j = 0; j < SIZE_SHAPES; ++j) {
333             if (currentShape[i][j] == 1 && inputCol + j >= n) {
334                     return 0;
335             }
336         }
337     }
338
339     // move and place the shape in the board
340     for (int i = 0; i <SIZE_SHAPES; ++i) {
341         for (int j = 0; j < SIZE_SHAPES; ++j) {
342             if (currentShape[i][j] == 1) {
343                 board[startingRow + i][inputCol + j] = 1;
344             }
345         }
346     }
347
348     return 1;
349 }
```

- **updateScore:** Checks for and clears filled rows, updating the score accordingly.

```
336 int updateScore(int board[MAX][MAX], int m, int n) {
337     int existsFilledRow = 1;
338     int score = 0;
339 
340     // finds the filled rows and adds to score
341     while (existsFilledRow) {
342         existsFilledRow = 0;
343 
344         for (int i = m-1; i >= 0; --i) {
345             int counter = 0;
346 
347             for (int j = 0; j < n; ++j) {
348                 if (board[i][j] == 1)
349                     counter++;
350             }
351 
352             if (counter == n) {
353                 existsFilledRow = 1;
354                 clearFilledRow(board, filledRow: i, n);
355                 score = score + n;
356             }
357 
358         }
359     }
360 
361     return score;
362 }
```

- **clearFilledRow:** Removes a filled row from the board, displaying a message about the cleared row and earned points.
- **gameOver:** Checks if any block has reached the top row, indicating the end of the game.

```
385 int gameOver(const int board[MAX][MAX], int n) {
386     for (int j = 0; j < n; ++j) {
387         // if the topmost row is reached, the game is over
388         if (board[0][j] == 1)
389             return 1;
390     }
391 
392     return 0;
393 }
```

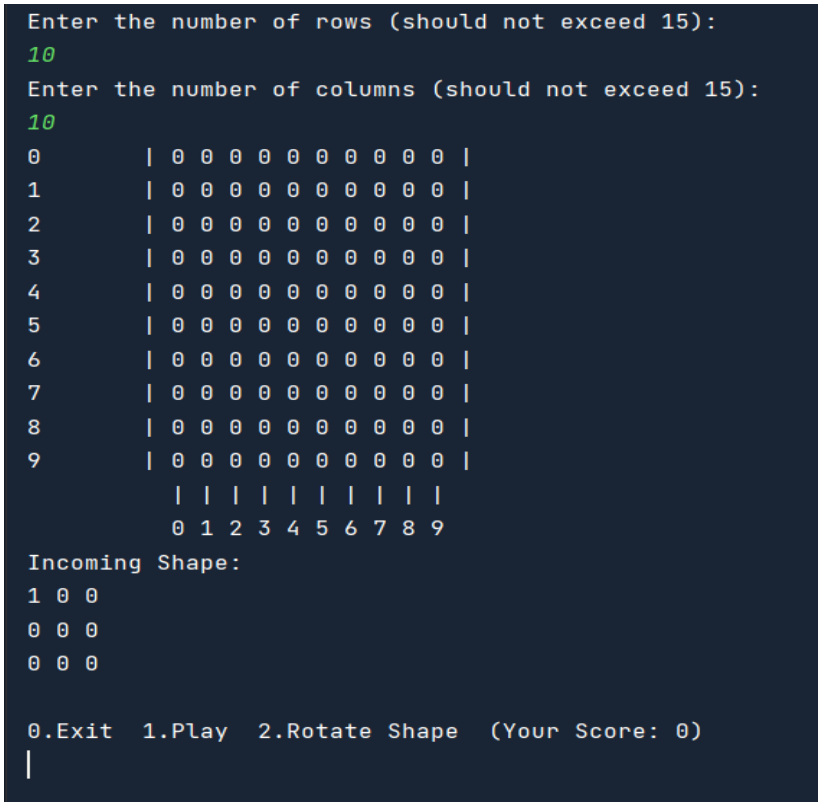
6. Loop Continuation:

The loop continues until the user chooses to exit or starts a new game. If the player opts for a new game, the board is reinitialized, and the process begins anew.

Code Structure, Design and Algorithm

The code demonstrates a well-structured design, modularizing functionalities into functions for clarity and reusability. Key game mechanics, such as generating shapes, moving them on the board, updating scores, and checking for game over, are systematically implemented. The menu-driven interface ensures user interaction, allowing them to make choices and control the flow of the game.

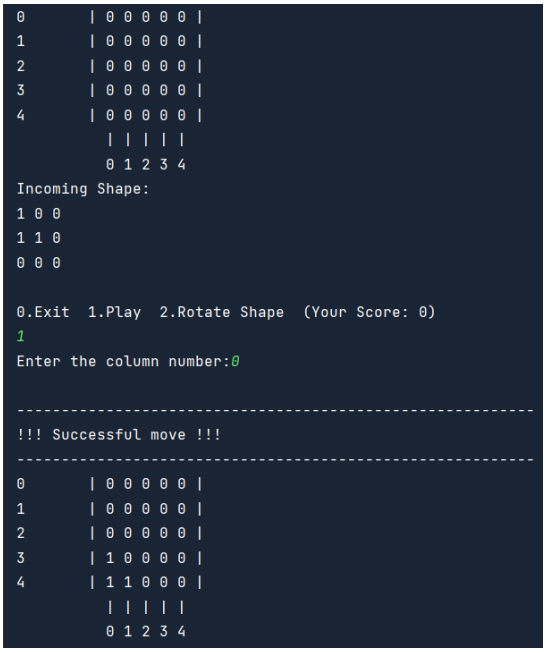
The main logic of the Tetris game code revolves around a loop that continuously prompts the user with a menu, allowing them to play, rotate a Tetris shape, or exit the game. Within this loop, the program dynamically generates Tetris shapes, displays the current state of the game board, and waits for user input. When the player chooses to play, they provide a column where the incoming shape should be placed. The code determines the suitable row for the shape using the **findFirstCollision** function, ensuring it aligns with the existing blocks on the board. The **moveShape** function is then employed to place the shape at the designated position. The game checks for successful moves, updates the score based on filled rows, and handles game over scenarios. The player can also rotate the Tetris shape, facilitated by the **rotateShape** function. The main algorithm involves user interaction, shape manipulation, and continual updating of the game board, creating an engaging and functional Tetris gaming experience.

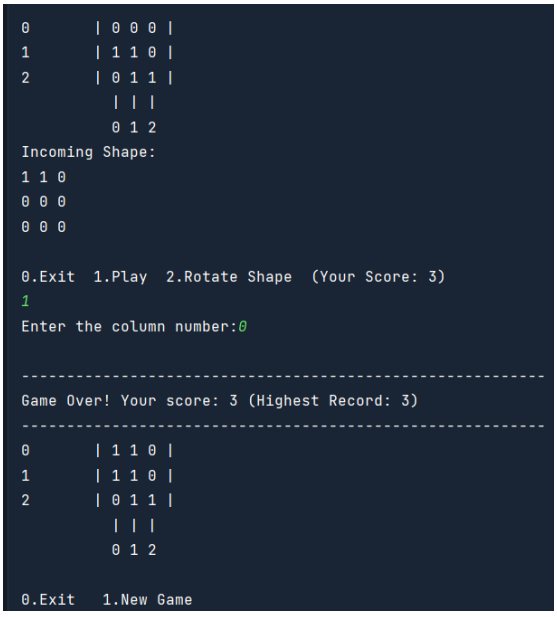


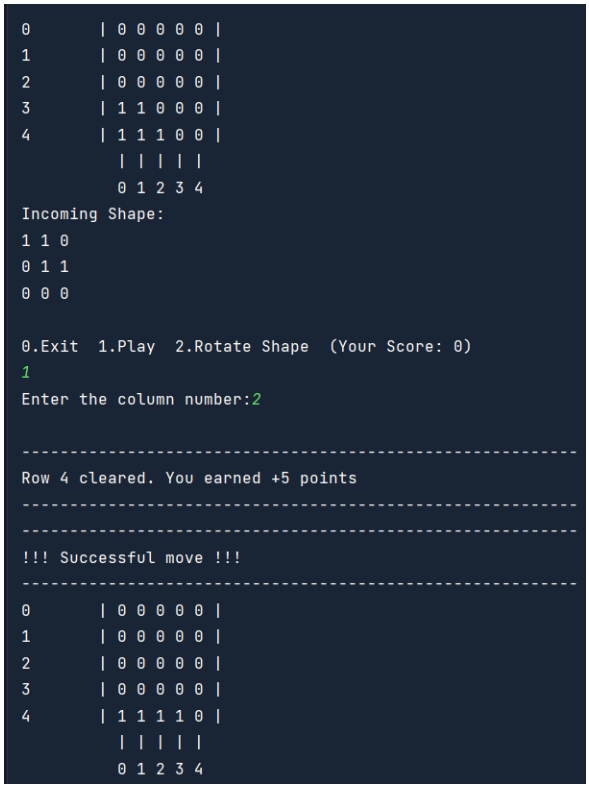
Results

The implemented Tetris game successfully provides users with an interactive gaming experience. Throughout the gameplay, users can control the placement and rotation of Tetris shapes within a customizable game board. The program effectively calculates and updates scores based on cleared rows, creating a dynamic and engaging environment. The game incorporates a user-friendly menu, allowing players to seamlessly navigate between playing, rotating shapes, and exiting the game.

Additionally, the code handles game over scenarios, resetting the board for new games. Screenshots of key moments during gameplay illustrate the visual aspects of the code, showcasing the evolving state of the game board and the dynamic placement of Tetris shapes. The results demonstrate a functional and enjoyable implementation of the classic Tetris game using the C programming language.







Video Link:

https://youtu.be/JB_64GgQMdE