Student Number: 10137114

Name: Parampreet Singh

Unit Code: PHYS30762&63762

Word count: 2449 (Excluding Front Page, Titles, Captions and Appendix)

Abstract

The purpose of this report is to discuss the design and development challenges which were faced in recreating the game Tetris in the C++ command line. This required the development of classes which broke the game down into sections and algorithms which allowed member functions of the classes to run sequentially. The game functions correctly with some visual errors and does not include any colour as it is entirely made from ASCII characters.

Introduction

The project chosen was to develop a board game, in this case Tetris. Although this is not technically a board game it is a 2D game which is what board games essentially are. They operate on tile by tile basis with specific rules. Tetris was developed by outlining the boundaries of the board within the console and creating rules as functions which specify how the game is to be played. A file linked to the game was also developed to store the name and score of every user who plays the game. A breakdown of the how game works and how the classes interact can be seen in Figure 1.

This report goes over the reasoning behind implementing the classes and the algorithms within the classes. It does not highlight every aspect of the classes, which is documented in the code. The end of this report is a discussion of how this project could be extended and improved.
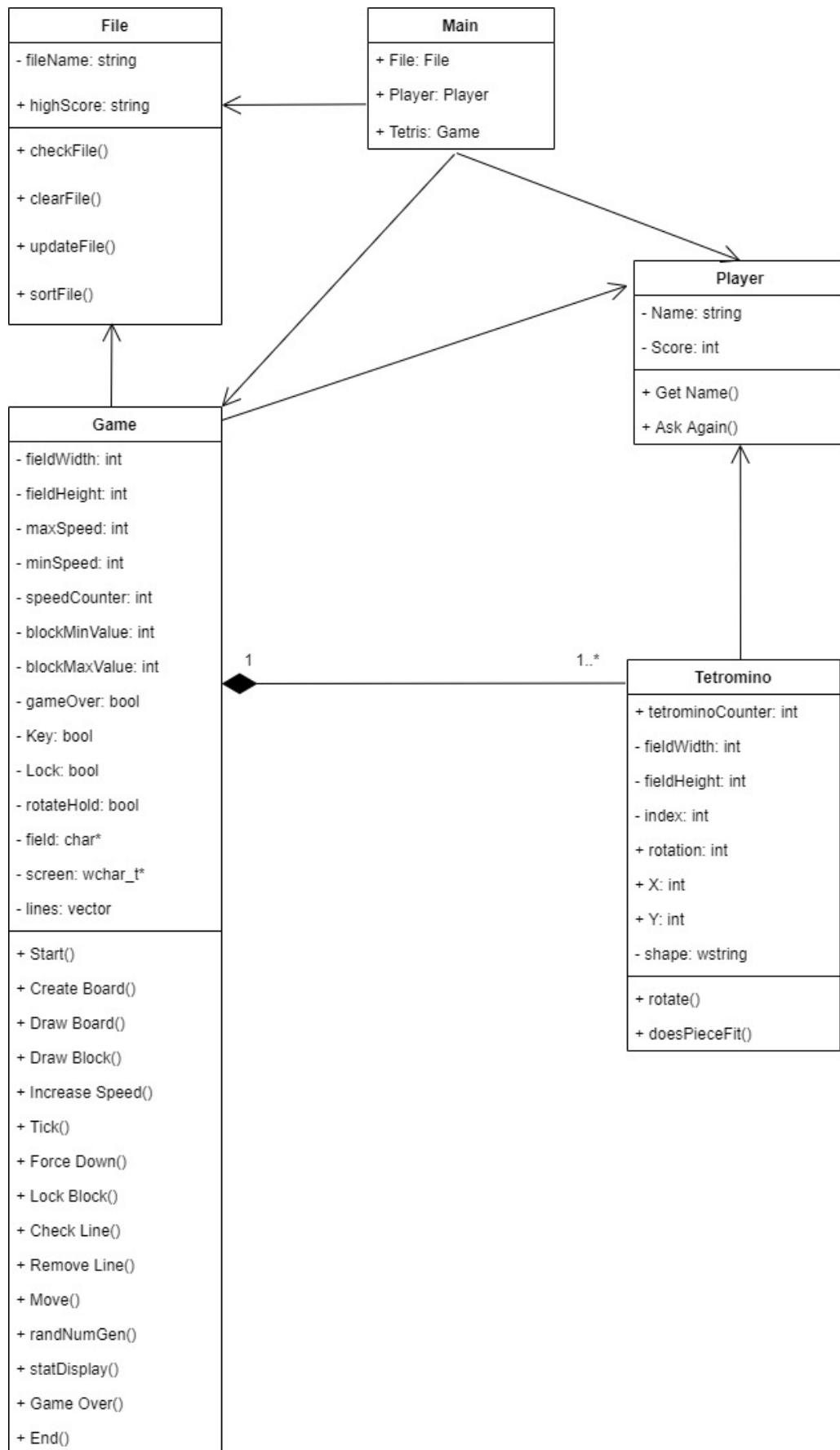
File

- fileName: string
+ highScore: string

+ checkFile()
+ clearFile()
+ updateFile()
+ sortFile()

Main

+ File: File
+ Player: Player
+ Tetris: Game

Player

- Name: string
- Score: int

+ Get Name()
+ Ask Again()

Game

- fieldWidth: int
- fieldHeight: int
- maxSpeed: int
- minSpeed: int
- speedCounter: int
- blockMinValue: int
- blockMaxValue: int
- gameOver: bool
- Key: bool
- Lock: bool
- rotateHold: bool
- field: char*
- screen: wchar_t*
- lines: vector

+ Start()
+ Create Board()
+ Draw Board()
+ Draw Block()
+ Increase Speed()
+ Tick()
+ Force Down()
+ Lock Block()
+ Check Line()
+ Remove Line()
+ Move()
+ randNumGen()
+ statDisplay()
+ Game Over()
+ End()

Tetromino

+ tetrominoCounter: int
- fieldWidth: int
- fieldHeight: int
- index: int
+ rotation: int
+ X: int
+ Y: int
- shape: wstring

+ rotate()
+ doesPieceFit()

1   1..*

Figure 1 | UML diagram for Tetris.

Code design and implementation

Class: Player

The class *Player* was created in order to store the user's information. *Player* is accessed by *Game* in order to display the user's information and accessed by *Tetromino* to update the user's score.

Class: File

The class *File* was created in order to keep track of all users who play the game. *File* stores the user's information in a text file. This prevents the user's information from being lost when the application is closed as the data is no longer stored in the temporary memory and is independent to the console. Making the file exclusive to the class *File* allows it to be repurposed in the future for similar applications. Also allowing one to store, clear and sort the file without affecting any other classes. *File* is accessed by the class *Game* in order to update the file with the users final score and to display the highest score in the file whilst the game is played.

- *sortFile*() reads the file and turns each line into a string and then appends those strings to a vector called *fileString*. Each character of a string, str, in *fileString* is then read. As the name of each player can only be 36 characters long the first 36 characters of str are then assigned to a string called *playerName*. The rest of the characters are stored in a string called *playerScore* which is then converted to type int. *playerName* and *playerScore* are then assigned to a pair called *playerInfo* which is then 'pushed_back' onto a vector called *data*. *data* can then be sorted from largest score to smallest using a lambda function and then written to the file.

```
for (int i{ 1 }; i < fileString.size() - 1; i++) // Skips first line (Header) of fileString
{
    for (int c{}; c < fileString[i].size(); c++) // Goes through each character of a string in fileString and separates name from score
        if (c < 35) // Names are 36 characters long
            playerName += fileString[i][c];
        else if (fileString[i][c] != ' ') // Remainder of string has to be score
            playerScore += fileString[i][c];

    // Creates pair for name and score
    playerInfo.first = playerName;
    if (!playerScore.empty())
        playerInfo.second = std::stoi(playerScore);

    // Push back pair onto new vector of pairs
    data.push_back(playerInfo);
    playerName.clear();
    playerScore.clear();
}

// Lambda function to sort vector of pairs by largest score
std::sort(data.begin(), data.end(), [](auto& left, auto& right) { return left.second > right.second; });
file.close();
```
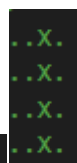
Figure 2 | Section of code from *sortFile()*.

Class: Tetromino

The class *Tetromino* was created in order to create blocks for the game. Separating the blocks into their own class allows one to be able to keep track of the number of blocks which are created. It also allows one to modify the blocks without affecting any other classes. Keeping this separate from the class *Game* is vital as it allows there to be a distinct separation between the blocks and the game logic, which makes implementing and debugging their interactions a lot simpler. *Tetromino* is only constructed within *Game* as it has no functionality outside of this class.

- *Tetromino*() is a constructor which takes a class *Player* by reference in order to update the players score. It also takes in an integer which determines the block's shape dependent on a switch statement. The shape is stored as a 1D list of characters to create a 2D block. The equation of turning a 2D index into 1D can be seen in equation 1.

```
                                          ..X.
                                          ..X.
                                          ..X.
case 0: shape = L"..X...X...X...X."; break;    ..X.
```

Figure 3 | Section of code from *Tetromino()* constructor depicting how a 1D list of characters will be shown on the console. The dots are not printed to the console.

- *Rotate*() takes in an X and Y coordinate for the block and an integer for the rotation, R, and rotates the block using a switch statement and returns the rotated index where an R of: 0 = no rotation, 1 = 90° rotation, 2 = 180° rotation and 3 = 270° rotation.

0°		X			90°		X			180°		X			270°		X		
		0	**1**	**2**	**3**			**0**	**1**	**2**	**3**			**0**	**1**	**2**	**3**			**0**	**1**	**2**	**3**

0°		**0**	**1**	**2**	**3**	90°		**0**	**1**	**2**	**3**	180°		**0**	**1**	**2**	**3**	270°		**0**	**1**	**2**	**3**
	0	0	1	2	3		**0**	12	8	4	0		**0**	15	14	13	12		**0**	3	7	11	15
Y	**1**	4	5	6	7	**Y**	**1**	13	9	5	1	**Y**	**1**	11	10	9	8	**Y**	**1**	2	6	10	14
	2	8	9	10	11		**2**	14	10	6	2		**2**	7	6	5	4		**2**	1	5	9	13
	3	12	13	14	15		**3**	15	11	7	3		**3**	3	2	1	0		**3**	0	4	8	12

Figure 4 | Shows x and y coordinate for block when rotated by 0, 90, 180 and 270°.

$$i = yw + x \qquad (1)$$

$$i = 12 + y - (4x) \qquad (2)$$

$$i = 15 - (4y) - x \qquad (3)$$

$$i = 3 + 4xy \qquad (4)$$

- Equations 1, 2, 3 and 4 were used in *Rotate()* to return the index of the block when rotated by 0, 90, 180 and 270°, respectively. Where i is the index, w, is the width (The width in this example is 4) and x and y are the coordinates of the block.

- *DoesPieceFit()* loops over the block and obtains the index of the field, *fi*, and block, *pi,* and checks if the block is attempting to occupy taken space. As *shape* is 16 characters long it will only read the 4 characters which are marked with 'X'. If the index of the field is not free (i.e. non-zero) it will return false.

```
// Collision detection: Reads only 'X' points of shape and checks if field is empty
if (this->shape[pi] == L'X' && pField[fi] != 0)
    return false; // fail on first hit
```

Figure 5 | Section of code from *DoesPieceFit().*

Class: Game

The class *Game* was created in order to deal with the logic of the game. It creates the game board and the rules of the game within member functions. It then goes on to handle the interactions of different classes, acting as a service layer. It also uses algorithms to make the game flow in the correct sequence. *Game* accesses all other classes in order to implement Tetris.

- *createBoard()* loops over the game board and assigns numerical values to sections of the board in *pField*.

9	0	0	9
9	0	0	9
9	0	0	9
9	0	0	9
9	0	0	9
9	10	10	9

Figure 6 | Shows a down scaled version of the game board. The left image shows how the data is stored in *pField* whilst the right image shows how it is displayed on the console.

- *updateScreen()* loops over the game board and assigns the *screen* to *pField* values with an offset of 2 to move the game board 2 characters to the right of the console. Effectively transferring the game board to the *screen* whilst assigning numerical values to the characters.

Number	Character	Purpose
0	" "	Blank space
1	A	Block
2	B	Block
3	C	Block
4	D	Block
5	E	Block
6	F	Block
7	G	Block
8	*	Line
9			Edge
10	T	Ground

Figure 7 | Table showing the numerical number character equivalent and purpose.

- *DrawString*() allows me to write to the console screen by looping over the string argument and writing each character to the *screen*. This is needed as I have taken control over the console and so can no longer use *std::cout.*
- *drawBlock()* loops over the block and only draws the sections in *shape* which are denoted by 'X'. *screen* is assigned to the block index + 65 as 65 is the ASCII equivalent to the character 'A'. Thus, each block will then be represented by a letter in the alphabet as the index increases from 0 to 6.

- *Tick*() causes the thread to sleep for 50 milliseconds which essentially pauses the screen, whilst increasing speedCounter by 1 each *Tick().*
- *lockBlock()* updates the *pField* with the current block's position and assigns the Boolean flag *bLock* to true.
- *forceDown*() checks to see if the position below is vacant and will force the block down by one character if it is. If however the space is not free this must mean that it is occupied and *lockBlock()* is called. After every iteration of *forceDown()*, *nSpeedCounter* is set to 0.
- *increaseSpeed*() checks the number of blocks created and if it is a multiple of 10 it will decrease *nMaxSpeed*.
  - **Interaction:** *forceDown*() will only activate once the Boolean flag *bForceDown* becomes true. This flag only becomes true after *nSpeedCounter* becomes equal to *nMaxSpeed*. *nSpeedCounter* only increases by 1 each *Tick()*. Thus, making *forceDown*() occur after every 20 *Ticks*() (1 second) at the beginning of the game. Decreasing *nMaxSpeed* causes *forceDown()* to occur more frequently as less game ticks occur, thus, giving the illusion that the game is speeding up.

- *checkLine*() loops only over the y coordinates of the current block. It checks to see if the current block causes a line. A line is identified by checking if there is a row of non-zeros in the game board. If this is identified, then that row of non-zeros will all be assigned to the numerical value of 8 ('*') and the vector, *vLine*, will store the y coordinate for the location of the line.
- *removeLine*() reads from *vLine* and assigns all the characters in that line to 0 (' ') to cause the line to disappear. However, there is delay of 0.4 ms to show the line as '*' characters. The row above the line is assigned to the line and this is looped from the row the line sits on, causing the game board to shift down until it reaches the line. Thus, creating the illusion that the line disappeared, and all the blocks moved down by one character. When a line is removed the score also gets updated. The score increase is exponential so for one line you get 100 points but for 4 lines you get 1600 points. Causing the user to take risks and try to eliminate stacks of lines at once.
- *randomNumberGenerator*() uses *#include <random>* and creates an object of type *std::mt19937* which defines a random engine known as mersenne twister. The random number is then seeded and an integer from a specified range is extracted from a uniform distribution.

```
for (int k{}; k < 4; k++)                                    //  R   L    D Spacebar
    bKey[k] = (0x8000 & GetAsyncKeyState((unsigned char)("\x27\x25\x28\x20"[k]))) != 0; // Virtual Key Codes

// Change coordinate of tetromino block dependent on key pressed
Block.get_nPosY() += (bKey[2]) && (Block.DoesPieceFit(Block.get_nRotation(), Block.get_nPosX(), Block.get_nPosY() + 1, pField)) ? 1 : 0; // Down Key

if (bKey[0]) // Right Key
{
    Block.get_nPosX() += (!bRightHold && (Block.DoesPieceFit(Block.get_nRotation(), Block.get_nPosX() + 1, Block.get_nPosY(), pField))) ? 1 : 0;
    bRightHold = true;
}
else
    bRightHold = false;

if (bKey[1]) // Left Key
{
    Block.get_nPosX() -= (!bLeftHold && (Block.DoesPieceFit(Block.get_nRotation(), Block.get_nPosX() - 1, Block.get_nPosY(), pField))) ? 1 : 0;
    bLeftHold = true;
}
else
    bLeftHold = false;

if (bKey[3]) // SpaceBar Key
{
    Block.get_nRotation() += (!bRotateHold && (Block.DoesPieceFit(Block.get_nRotation() + 1, Block.get_nPosX(), Block.get_nPosY(), pField))) ? 1 : 0;
    bRotateHold = true; // Stops from constant spinning when held
}
else
    bRotateHold = false;
```

Figure 8 | Section of code from *Move*()

- *Move*() uses *GetAsyncKeyState* which returns true if a specified key is pressed. *bKey* stores an array of bool values to store the return value of *GetAsyncKeyState*. Thus, if a button is pressed and the block can move to that position the block position is updated. Bool flags are put into place to ensure that one button press is not read multiple times. This flag has not been included for the down key allowing the user to quickly lock a block.

```
    HANDLE hConsole = CreateConsoleScreenBuffer(GENERIC_READ | GENERIC_WRITE, 0, NULL, CONSOLE_TEXTMODE_BUFFER, NULL);
    SetConsoleActiveScreenBuffer(hConsole);
    DWORD dwBytesWritten{};
```

Figure 9 | Section of code from *Move()*.

- In Figure 9 a handle to the console buffer is created and set as the active screen buffer. Displaying to the console is now done by *WriteConsoleOutputCharacter()*. The console only writes the characters stored in *screen.* So the console is updated either by directly assigning characters to *screen* or by assigning characters to *pField* which is then assigned to screen via *updateScreen().*

```
// Algorithm 1 - Will allow block to keep moving and cycle through functions untill bLock becomes true in forceDown()
Tick();
updateScreen();
drawBlock(Block); // Constantly updating position of tetromino block relative to board
forceDown(Block);
checkLine(Block);
displayStats(Player, file);
WriteConsoleOutputCharacter(hConsole, screen, nScreenWidth * nScreenHeight, { 0,0 }, &dwBytesWritten);
removeLine(Player, file);
```

Figure 10 | Algorithm 1 from *Move*()

- Algorithm 1 cycles through all the listed functions allowing the block to constantly move whilst checking for lines and moving the block down periodically. This cycle will not end until the block is locked and *bLock* becomes true.
  - *removeLine()* has been placed after the console has been written to, to ensure that the 0.4 ms delay for the line turning into '*' is seen by the user.

- *checkLine()* is called before *removeLine()* to count the number of lines created so the score gets updated correctly.
- *forceDown()* occurs after *drawBlock()* to ensure that it is checking below the correct current position of the block for vacancy.

```
}
// Algorithim 2 - Causes recursion so function never ends and keeps creating new blocks untill game ends.
bLock = false;
vLines.clear();
Tetromino nBlock(randomNumberGenerator(), Player);
increaseSpeed(nBlock);
drawBlock(nBlock);
gameOver(nBlock); // Checks to see if game should be ended
Move(nBlock, Player, file);
```

Figure 11 |Algorithm 2 *Move*().

- Algorithm 2 resets variables and cycles through the listed functions. If the game has not ended it will then call *Move()*, leading to recursion. Thus, blocks will keep spawning after the one before is locked and this process will not stop until a spawned block cannot move leading to the game to end.
  - *vLines* is cleared once the old block is locked to ensure that the correct number of points are awarded for the number of lines created in a single block getting locked.
  - *gameOver()* is checked just before *Move()* is recalled to prevent the block from spawning in a non-allowed position.

Class: main

- Creates three objects *Game*, *Player* and *File* through default constructors and checks that the "Score.txt" file exists and then calls *Start*() from *Game*.
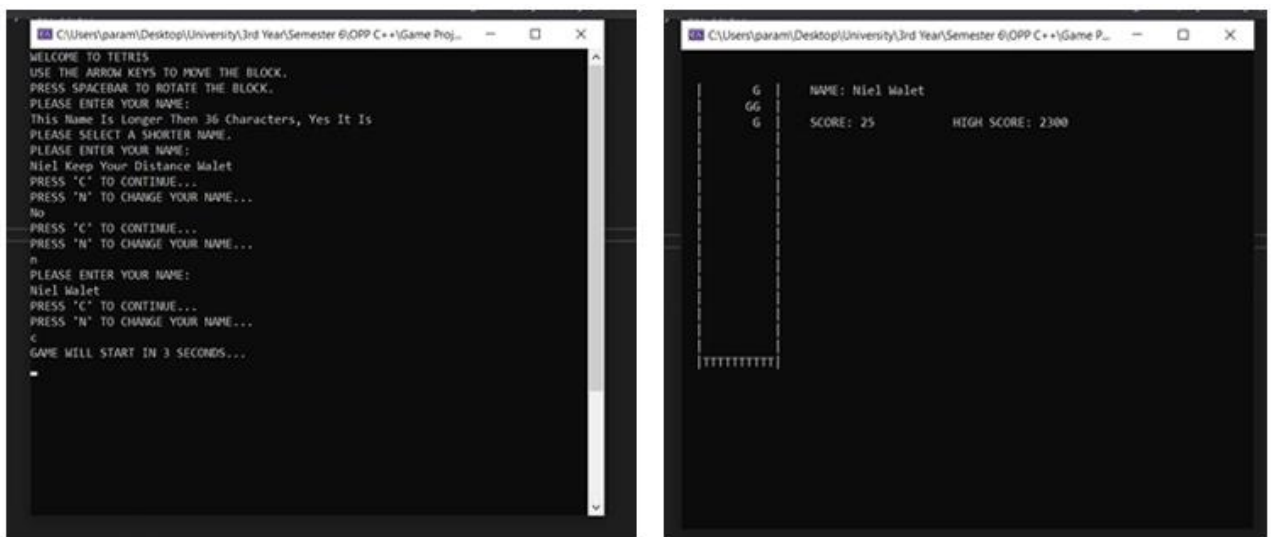
Results

Figure 12 | Start screen and beginning of game.

Figure 12 shows how the game is presented when "Score.txt" is in the correct directory. It also shows input validation when the name is too long or when instructions aren't correctly followed. The high score is taken from "Score.txt" and presented at the top right.
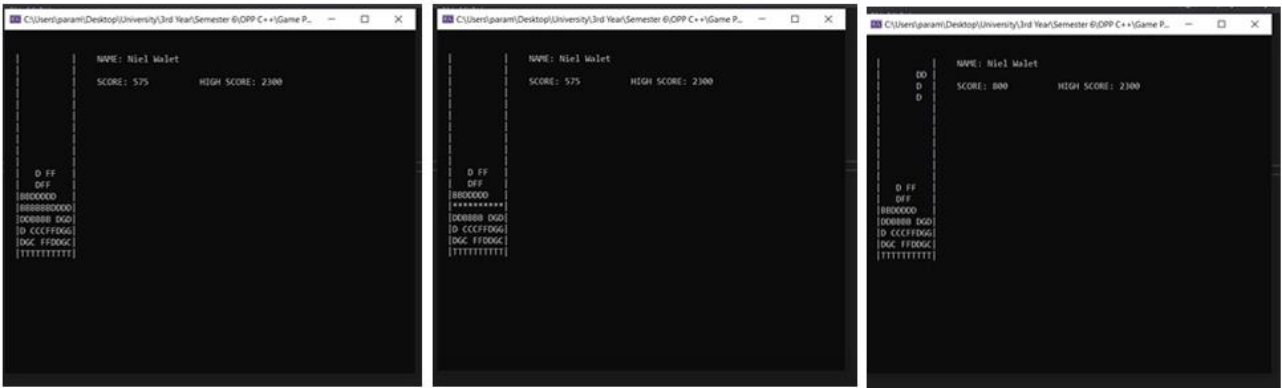

Figure 13 | Screen in process of making a line in the game.

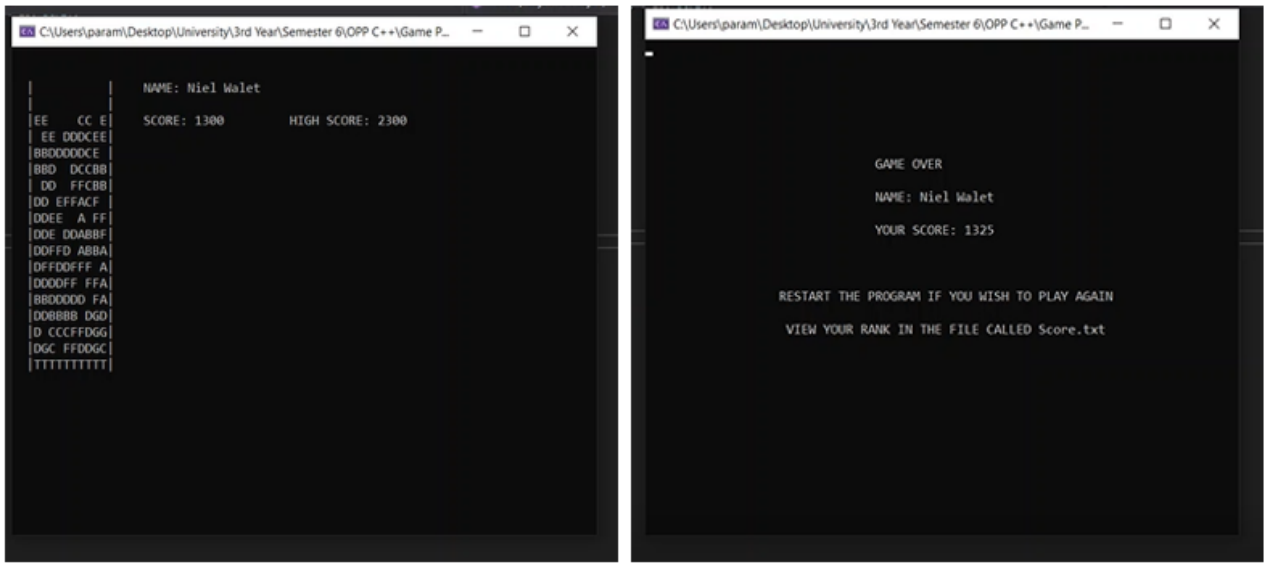Figure 13 shows how the game responds when a line is created.

Figure 14 | Screen at end of game.

Figure 14 shows the screen before and after the game ends. The block will no longer be able to move when it spawns and so the end screen is presented.
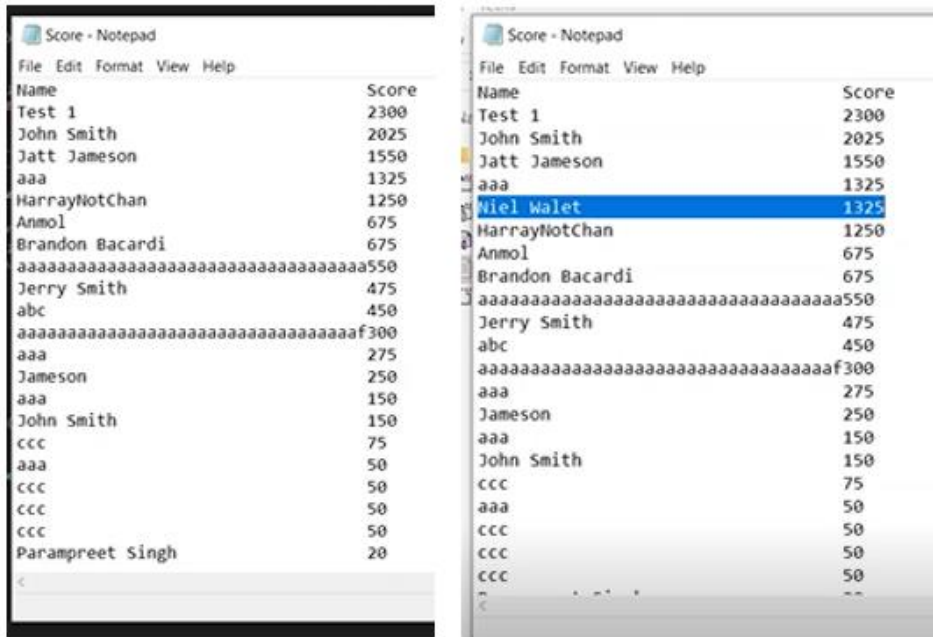
Figure 15 | File before (left) and after (right) game has ended.

Figure 15 shows the file "Score.txt" before and after it is updated. It sorts the names from highest to lowest score. If you are successful in getting the highest score, then your score will be presented as the high score the next time the game is started.
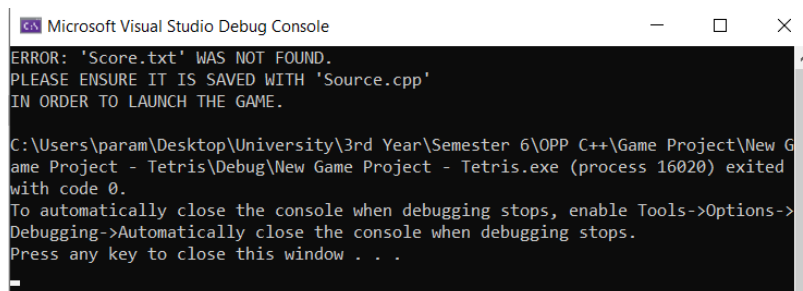

Figure 16 |Console screen when "Score.txt" cannot be found.

Figure 16 shows the console when "Score.txt" is not found. The game does not start and the application needs to be restarted.

Discussion

One way the project could be improved is by fixing the update lag caused when each block is locked. This could be solved if *hConsole* did not create and set a text buffer each iteration of *Move()*. However, to implement this would require *Move()* to have access to *hConsole* to update the display but *hConsole* to initialise outside of the function. When implemented *SetConsoleActiveScreenBuffer()* would error and only work when inside a function. Further debugging is required to understand the root cause of this issue, but it might be raised by the memory access through the pointers.

Another way the project could be improved is by displaying the next block to the user before it spawns. This could be implemented by generating a random number for the next block before it spawns and then displaying the shape which would be generated from the random number under the players score. This number would then have to be stored and then used for the new block's constructor. The variable storing this number would then be assigned to the next random number once the new block is generated. Time constraints prevented me from implementing this.

To extend the project I could have used *CHAR_INFO* instead of *whcar_t* for my type for *screen*. *CHAR_INFO* can be split into two parts which are glyph and palette, where glyph represents the character and palette represents the colour. This would allow me to add colour to the game for blocks, background and the board. The colour was deemed to be secondary at this stage, and the efforts went into defining the correct underlying logic.

Another way to extend the project is by adding a play again feature. I tried to implement this but found when I tried to access *std::cin* after I created a text buffer for the console the system would crash and not reopen until I restarted visual studios. The most likely root cause of this is related to the memory buffers involved, which need to be re-instantiated. Initially this was thought to be caused by not explicitly closing the handle with *CloseHandle*() function, however, when implementing this it made no difference. As the error would lead to the crash of visual studio, I was unable to quickly debug the problem.

Extensive testing is still required around exception management and other areas which would be achieved through unit-testing, but this was deemed to be out-of-scope for this project at this stage.

Appendix 1: Impact of COVID19 on the work completed and report

Please describe the effect of the COVID19 under the following headings.
If there were difficulties of a personal nature, such as in your home situation, that you would rather communicate directly to [prof. Walet](#) or [physics.support@manchester.ac.uk](#), please do so!

Time lost travelling home:

Some of you may have had substantial difficulties travelling home, and lost a lot of time on the way. Please give details here

Lack of access to computers and the internet:

We understand that not all students have access to sufficient IT resources. Please specify what the limitation of resources is.

Limitation in the outcome:

Please describe how this has impacted both the extent of the results presented, and the quality of the report.

Staying in Manchester alone over the course of many weeks took its toll on my mental wellbeing. Being somebody who has been suffering with mental health for a couple of years, I know that going to the gym and regular social contact are key aspects for me staying well. Having these taken away whilst simultaneously being forced to stay in Manchester to finish assignments. I can say that I certainly felt my mental wellbeing decay as time progressed and this can be seen on my performance on assignments.

Covid-19 also disrupted my studies to the extent where no matter of my performance this semester my overall grade could not change. As a student on the border of a 2:2 and 2:1 I decided to dedicate my entire final semester in the library and work really hard to ensure that I obtain the necessary grades in my final semester to end up with a 2:1. The measures brought into place after Covid-19 meant that no matter how well I performed in my final semester it would not carry enough weight to change my grade due to the lack of examination. Realising this, it was very difficult to find the motivation to perform well on any of the remaining assessments. Regardless, I still tried my very best to perform on my remaining modules hoping that I could show future employers specific module grades. However, I can say with certainty that my performance on my final assignments could have been far greater if I knew that they would carry weight in my degree.