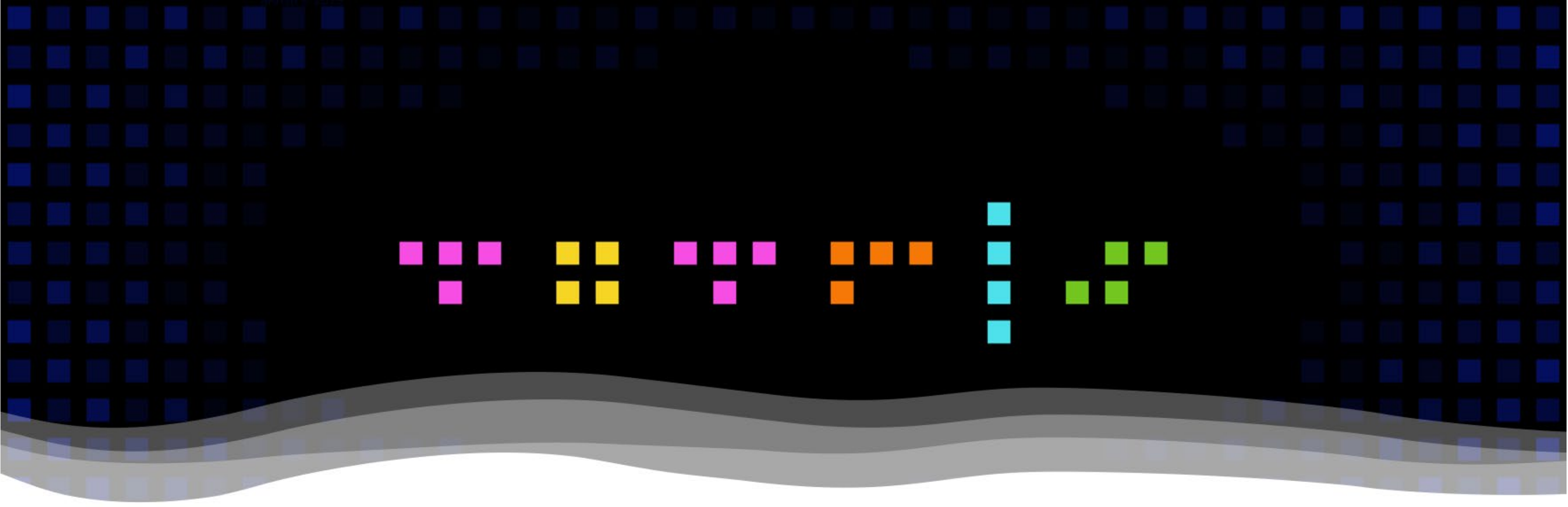


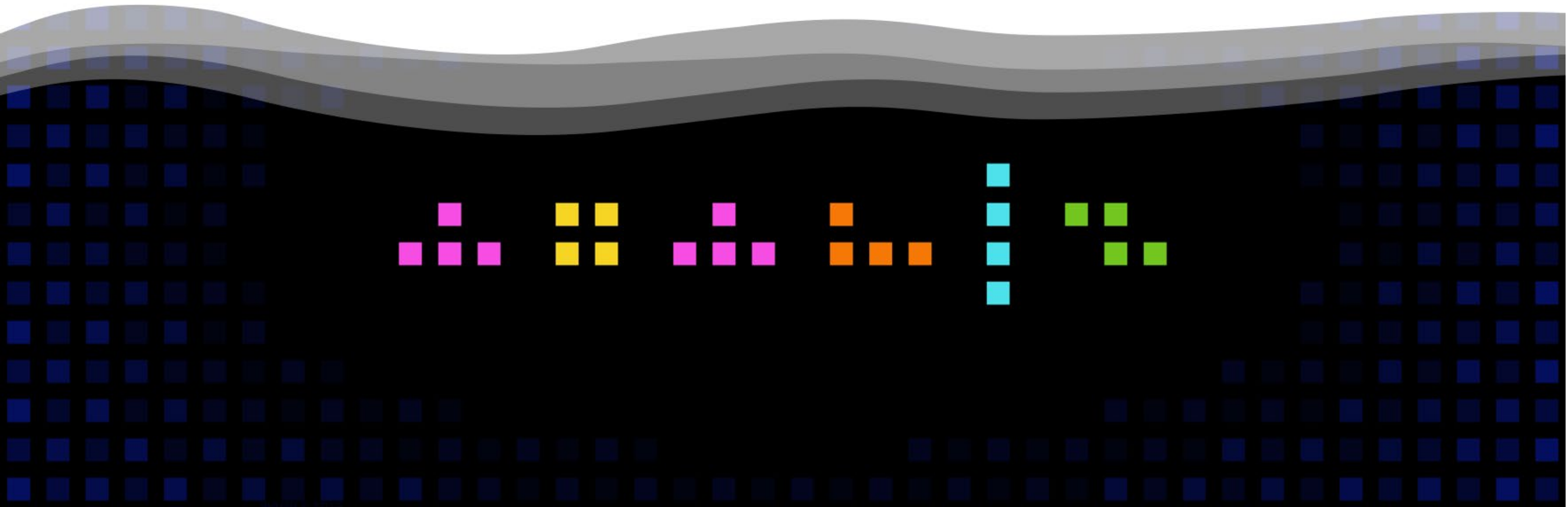
Seys Matthias

Gameboy Tetris System

Graduation work 2020-2021

Digital Arts and Entertainment

Howest.be



CONTENTS

ABSTRACT

In this paper we'll be creating a system that should be able to play Tetris while only having access to a Gameboys screen and buttons. We'll only have access to the pixel buffer and only being able to set the state of the buttons.

The focus of the paper is gathering the needed data from the pixels. We'll be avoiding the Tetris ROM data and only use what we can see. We'll use an algorithm to find the best move where we can get the necessary move set from to give to our system.

The system should be split up into clear parts so each part can be handled separately and then all comes together for the playing of the game itself.

INTRODUCTION

As games grow in complexity and use more pixels than someone could count, it is time to look back when games were smaller. We'll have a look at Tetris on the original Gameboy and see what we can do without modifying anything related to the game itself.

Many papers have done this before with focusing on the algorithm to best play Tetris, but most papers have modified and used the data from the ROM instead or they created their own Tetris to experiment on.

Instead of doing what others have done, we'll be following the WYSIWYG principle, which stands for "What You See Is What You Get". When you look at a Gameboy, all you can see is the screen and the buttons, which is all we will be using for our Tetris System that should play the game.

For this research, we'll be using the Gameboy Emulator created by Brecht Uytterschaut where we can access what we see as well as emulating the button presses.

If you wish to learn more about algorithms, this is not the paper for it as we'll be focusing on how we can get all the necessary data from the pixels. There's some good tutorials and papers in the References where I also took some inspirations and ideas from.

TECHNICAL DETAILS

The System only runs on x64 due to the emulator not working on x86. The system does not fully run correctly on Debug when it is calculating the move, the slow speed might interfere as the Gameboy Emulator might run multi-threaded which makes the system fall behind when trying to get data from it.

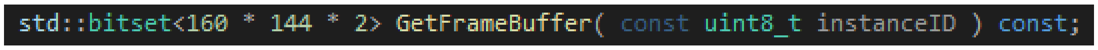
You can find the project and a build at this Github repository:
https://github.com/MSeys/GameboyTetrisSystem

RESEARCH

1. GAMEBOY EMULATOR

1.1. PIXEL BUFFER

The Gameboy Emulator gives us access to a Pixel Buffer that represents the screen. The buffer is in the format of a bitset with as size 160 by 144 by 2. The screen is 160 by 144 pixels and each pixel is represented as 2 bits which allows us to mimic the four “colors” the Gameboy can show – a value from 0 to 3. The original Gameboy also had multiple editions which lead to two different color variants, a green-tinted one and a grayscale one.



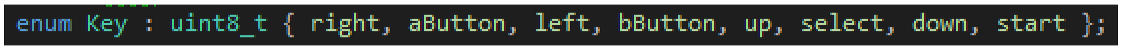
```
std::bitset<160 * 144 * 2> GetFrameBuffer( const uint8_t instanceID ) const;
```

1.1a Gameboy Emulator – Frame Buffer

An interesting finding is that the colors on the Gameboy happened due to how much “power” you send to the pixel to turn it on. Giving full power to the pixel would give you the brightest color while giving no power would give you the darkest. Giving a third or two thirds would give you two in between colors.

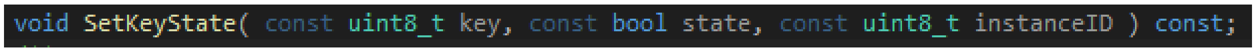
1.2. INPUT

The Gameboy Emulator gives us access to a simple function and an enumerator – which contains all the buttons – which allows us to set the state for them very easily. By looking through a demo, it would seem to state also resets automatically and isn’t handled in the demo itself. This allows us to not worry about having to change states to released again.



```
enum Key : uint8_t { right, aButton, left, bButton, up, select, down, start };
```

1.2a Gameboy Emulator – Key Enum



```
void SetKeyState( const uint8_t key, const bool state, const uint8_t instanceID ) const;
```

1.2b Gameboy Emulator – Set Key State

1.3. POSSIBLE ISSUES

WHITE LINES

White lines seem to appear throughout the pixel buffer almost randomly. The frequency depends on the build version (Debug vs Release). The white lines come and go in just one frame but can still tamper with the data we need to get from the buffer. Of course, we can just look at multiple pixels to keep our data correct.

PIXEL BUFFER OFFSET

There’s also a small vertical offset in the pixel buffer which requires us to handle a few things slightly differently as there is a small loss. Of course, we can also work around this as there are still more than enough pixels left to get data from and the top row – which is missing – isn’t entirely useful.

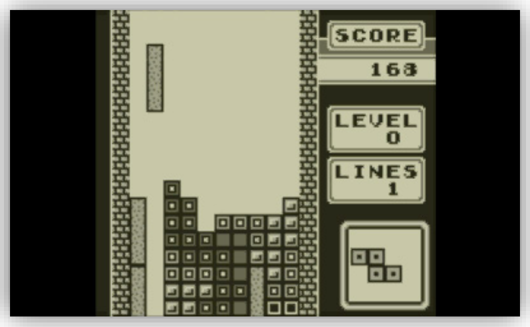
2. TETRIS

2.1. INTRODUCTION

Tetris was a very popular game back in the day. You need to fill rows by using various shapes of pieces that you can rotate. The goal was to keep making as many lines as possible and aim for the highest score. This version also contains another mode where you must clear a specific number of lines, but we'll be avoiding that mode for this research as there is not a clear data point that we can use to have our data updated correctly.

2.2. PLAYFIELD AND CURRENT PIECE

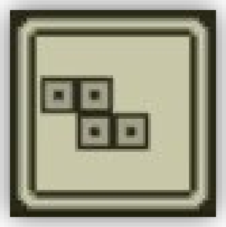
The playfield is a grid of 10 by 18 blocks, each block taking up 8 by 8 pixels. The playfield itself therefore takes up 80 by 144 pixels of the screen. The current piece is also located on the playfield grid and takes up a maximum of 2 rows and 4 columns. It is spawned in the 2nd and 3rd row and uses the middle 4 columns.



2.2a Tetris on Gameboy. (Taken from Nintendo)

2.3. NEXT PIECE

The next piece is in the bottom right corner of the screen. The piece is also positioned in a 4 by 4 grid. However, only the middle rows are used. Also, something that isn't very noticeable is that there is a small white border – one pixel wide – is between the black border and the Tetris piece.



2.3a Next Piece on Gameboy. (Taken from Nintendo)

2.4. SCORING

Score is given based on clearing lines; this value increases per level. There's also an additional way of getting score, which is soft dropping the block. Soft dropping means the player is dropping the block by pressing down (continuously), halting this or do tricks at the end (e.g. quickly putting a Z-Piece in a hole at the end) denies you the score, this however could be a bug or an unforeseen circumstance in the original Gameboy version.

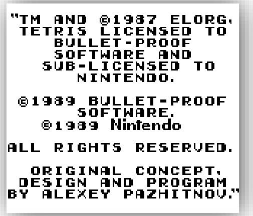
3. TETRIS SYSTEM

3.1. MENU SCREENS

The system will start when the game starts so we'll need to make sure the system can also control the menu's it has to go through. I'll be going through every menu / screen type to explain possible differences and other interesting findings.

CREDITS SCREEN

The Credits screen is only using the darkest and brightest colors (0 & 3). There's no grey-tinted colors seen on this screen, which makes it very noticeable when it is this screen.



"TM AND ©1987 ELORG, TETRIS LICENSED TO BULLET-PROOF SOFTWARE AND SUB-LICENSED TO NINTENDO.

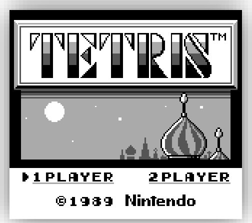
©1989 BULLET-PROOF SOFTWARE. ©1989 Nintendo

ALL RIGHTS RESERVED.

ORIGINAL CONCEPT, DESIGN AND PROGRAM BY ALEXEY PAZHITNOV."

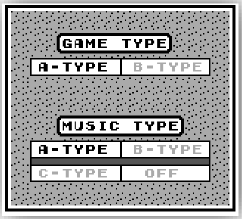
START SCREEN

The Start screen uses all (four) available colors and contains a nice pixelated palace. When it comes to differences, this is the only screen where the first row is completely black.



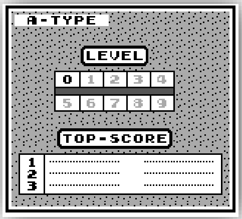
GAME MODE SCREEN

The Game Mode screen also has a completely white first row, like the credits screen. However, this screen also is full of grey-tinted colors. However, the Game Mode screen is very similar to the Level Select screen.



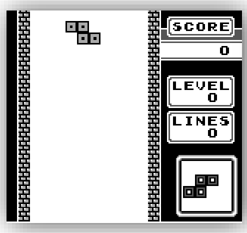
LEVEL SELECT SCREEN

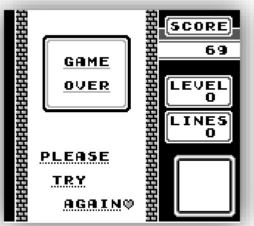
The Level Select screen is very similar to the Game Mode screen, but there's a very nice spot we can use to be able to identify the difference. In the top left corner, there is a part that only consists of the brightest and darkest color. We can use that to our advantage. One noticeable finding is, when returning from the playfield screen, we must first confirm our score and then we can select a level again. The previously selected level is still selected.



PLAYFIELD SCREEN / GAME OVER

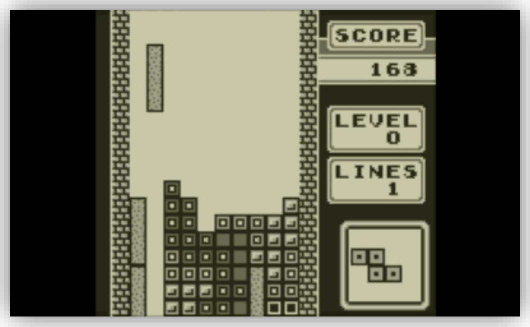
The Playfield screen and Game Over screen are very similar in structure with only few differences. The major difference is the empty "next piece" block which is never empty while playing. To see whether it is either of these screens, we can use the top row again. The number of white pixels is at least 10 times 8 pixels as that's the size of the playfield area itself, but also still contains black and grey-tinted pixels.





3.2. PLAYFIELD

The playfield is 10 blocks wide and 18 blocks high. We can figure out at what pixel the playfield exactly starts and ends by looking into the pixel buffer. By looking at the pixels starting from the start point, we can figure out where exactly a block is or not. That way we can convert it to our own data format. There's only one thing similar between every block and that is the border, which is black. And the only purely similar thing is the corners as with some blocks, some borders are missing.



3.2a Tetris on Gameboy. (Taken from Nintendo)

POSSIBLE ISSUES

The current piece is also part of the playfield; however, we don't want that in our data representation of the playfield. There's several ways we can solve this.

A. We keep track of the current piece position and remove it from our data every time.
B. We update our data once every x time.

Option B is more tempting as it would be less calculations per frame, and it would give less issues.

There's also a slight issue of the vertical offset in the pixel buffer, a possible solution would be to skip out on the top row of the playfield as once a piece would reach the spawning location, it would already mean game over.

DATA REPRESENTATION

The playfield would be represented by a 2D container that contains whether there is a block there or not – true or false. The initial data would also include the current piece but will be removed from the data afterwards. Removing the current piece might however tamper with the data as there is multiple ways to remove it, some not as optimized or as good as others.

3.3. CURRENT PIECE

The current piece spawns on the 2nd and 3rd row and can take up the middle four columns – at a maximum – which leads to a maximum of 8 blocks we need to check to know what Tetris piece it is. We know where the data is so we, again, can check the corners of where the blocks are and see if there is one there.

POSSIBLE ISSUES

Due to only having access to the pixels, we can only use what we get. However, that can lead into trouble when the blocks start going higher and higher and ultimately reach the spawning area of the current piece. Blocks in that zone will interfere with the data. The system should be able to handle that and make sure it knows the game is over.

White lines also tend to appear on the current piece which means we'll need to make sure to not require too many corners and make sure to check enough corners. Requiring too many corners or not checking enough corners could lead to incorrect data.

DATA REPRESENTATION

Initially, the current piece would be made up as a 2D container of 2 rows and 4 columns, as big as the possible area of the current piece. Using that we can manually find out what piece it is by checking what blocks are true or false. However, the result of the current piece is an enumerator value which says what piece type it is. Using that, we'll later get other manually made data for the algorithm.

3.4. NEXT PIECE

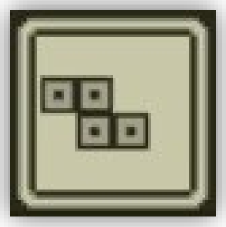
The next piece spawns in a grid-like shape in the bottom right corner of the screen. There's a 1-pixel white line between the black border and the first possible corner of a piece. We can notice only the 2nd and 3rd row are used for the pieces as none of the pieces in their 0-rotation form take up 3 or 4 rows. That way we again have a maximum of 8 blocks necessary to check to know what piece it is.

POSSIBLE ISSUES

White lines also appear across this part of the screen, so we'll need to make extra sure our data stays correct no matter where the lines appear.

DATA REPRESENTATION

The next piece would also be made up as a 2D container of 2 rows and 4 columns – the same as the current piece. We could then use the same way we've done for the current piece to find out what piece type it is and use that later for other data we need for the algorithm.


3.4a Next Piece on Gameboy. (Taken from Nintendo)

3.5. UPDATING THE DATA

Currently there’s multiple possibilities as there’s a lot of data we can get from the pixels.

A. We could update when the score changes, checking a range of pixels for any changes. However, score isn’t always added when doing certain moves. The question is if the algorithm / system will ever have that issue.

B. We could update when the next piece changes, but there is the possibility of a duplicate and then we could be stuck.

C. A final option would be to use the current piece but check if there is a new Tetris piece after an x amount of time or frames as we know the area the pieces spawn at. However, this can lead to issues further down the line as pieces on the field can reach the spawning location and could interfere.

4. TETRIS ALGORITHM

4.1. BLOCK POSITION PREDICTION

A major part before beginning the algorithm itself, is being able to predict where a block would land with a certain rotation and movement to the left or right. There's a few possibilities to do this when it comes to creating the block data necessary to "drop it down".

A. We could manually create the rotation containers with how many times it can move left and right.
B. We rotate the initial 2D container we created.

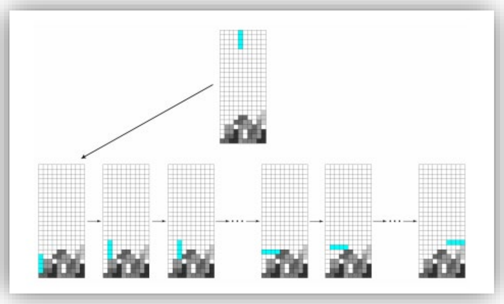
While option B might be tempting, it's harder to know how many times it can move left or right which is necessary to know. Option A would lead to more manual work, but it would follow the rules of the Tetris blocks which wouldn't suddenly change. There also seems to be less rotations for specific pieces than in newer Tetris games.

4.2. MAKING THE BEST MOVE

To create the best move, we need to investigate several parts of the algorithm that will help us define this. We'll be using several heuristics that are widely used for Tetris AI. One thing to note is that we'll "create" the possible moves breadth-first based as we need to check all moves and then go deeper (at least once, due to knowing the next piece), this is also used for most Tetris AI.

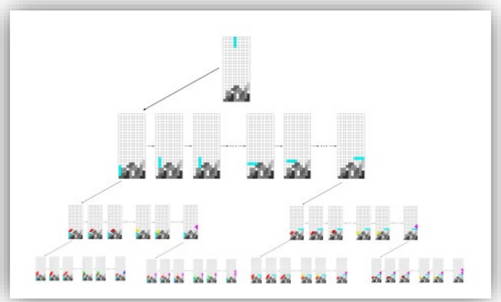
BREADTH-FIRST

Breadth-first means we'll be focusing on doing more "widespread" moves rather than going deeper on one possible move to see what results it could give. Tetris has a limited field and placement and rotation matters when it comes to pieces, so a breadth-first solution is the way to go as proven by several papers creating an AI to play Tetris.



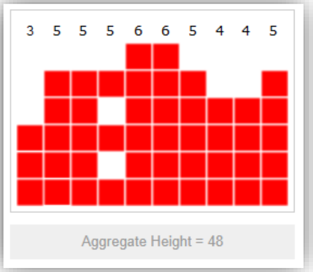
HOW DEEP CAN WE GO?

While we are going for a breadth-first solution, we can still add some depth to it as we know the next piece that would spawn. Using this, we can add another layer and see what move would be better. While a move might not seem interesting at first, could be the better move when a next piece is involved. The deeper we go, the more data we can get and the better move we could find. However, the deeper we go, the longer it will take to find the best move. The deeper we go, the more moves we must check and compare.



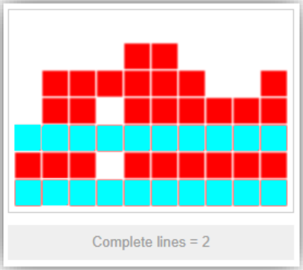
AGGREGATE HEIGHT (HEURISTIC)

The aggregate height means the sum of the height of all columns, which includes the holes. When playing Tetris, you need to try to keep the columns as low as possible, which means we want to keep this value low.



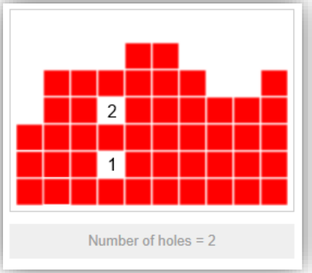
COMPLETE LINES (HEURISTIC)

This heuristic is very straightforward. It is the amount of completed lines in this move. The goal is to maximize this value as it is the goal of the game to create lines and it clears up space on the field.



HOLES (HEURISTIC)

A hole is defined when there is at least one block above it in the same column. Holes should be kept to a minimum as holes are harder to clear.

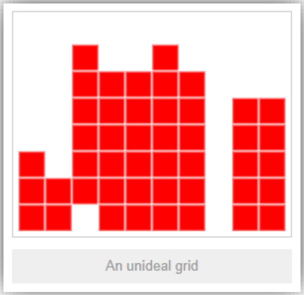


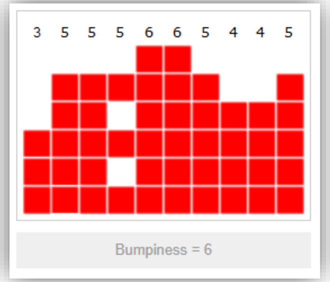
BUMPINESS (HEURISTIC)

The bumpiness tells us the variation between columns. A well should be avoided as it is harder to clear – only an I-Piece would be able to.

The bumpiness is the sum of the absolute difference between adjacent columns.

$$bumpiness = 6 = |3-5| + |5-5| + \ldots + |4-4| + |4-5|$$





PUTTING IT ALL TOGETHER

The final step to calculate a score is to put it together. For this step the researcher of the AI has made a linear combination of the four heuristics. This includes four constant parameters that he/she produced using a Genetic Algorithm. (If you wish to learn about this process, see "*Tetris AI – The (Near) Perfect Bot*")

The researcher assumed as we want to keep aggregate height, holes and bumpiness to a minimum, the constant parameters for those would be negative. The opposite happens for the completed lines as we want to maximize that.

$$a \times (AggregateHeight) + b \times (CompleteLines) + c \times (Holes) + d \times (Bumpiness)$$

$$\begin{gathered} a = -0.510066 \\ b = 0.760666 \\ c = -0.35663 \\ d = -0.184483 \end{gathered}$$

THE PLAN

The initial plan for this system would to calculate the moves for the current piece, use those moves to calculate the moves for the possible next move and calculate the score. This could be extended by also picking a random piece afterwards that might come after the next piece to possibly get an even better move, but that would also be up to chance.

WHAT ABOUT TRICK MOVES?

The algorithm does not account for trick moves as there's a lot of edge cases and as proven by research of the Tetris game, score is also not given when doing these kind of trick moves.

CASE STUDY

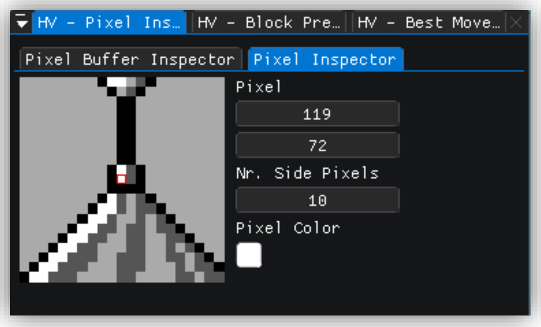
1. SYSTEM DEVELOPMENT

1.1. GETTING THE PIXELS

The first step is to create a way to get specific pixels or areas from the Gameboys screen. The buffer in the emulator stores values in an optimized, but not entirely readable way. However, we could get a value between 0 to 3 and store that in our representation which would be a 2D container so we can use an X and Y value to access a pixel. The value can also be used as an index, for example to know what color to use to render.

PIXEL INSPECTOR

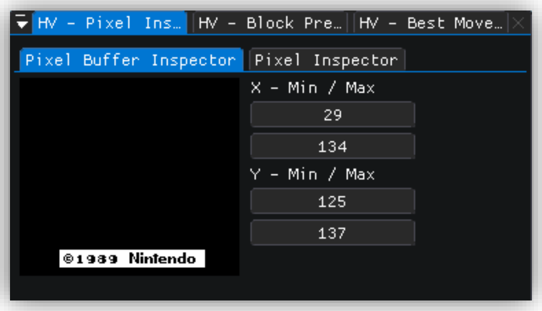
We can use the value in our pixel buffer as an index to know what color we need to render the pixel as. The color itself is merely used for rendering.
This Helper View (HV) was created to see a pixel and its surrounding ones. It was mainly used to find the starting pixel and ending pixels of needed areas such as playfield, current piece, next piece, etc. It was also used to check the top row and see what values / colors it contained.



Pixel
119
72
Nr. Side Pixels
10
Pixel Color

PIXEL BUFFER INSPECTOR

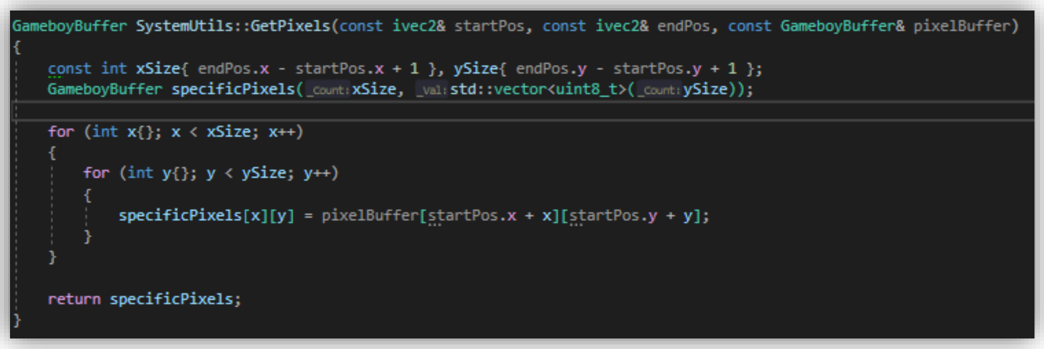
This Helper View was created to see a specific part of the pixel buffer. This view was made to see if the buffer was made correctly. This view doesn't use the additional function that would return the specific area of pixels as the unneeded pixels were just rendered as dark in here. The additional function would purely return the needed pixels and is used for other parts.



X - Min / Max
29
134
Y - Min / Max
125
137

GET PIXEL AREA

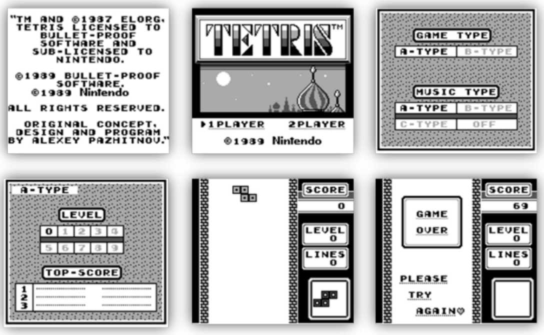
While for one pixel, it is easy enough to use the pixel buffer. However, for an area I created a function that will give me the specific pixel area requested in the same format as our pixel buffer. This function is also usually accompanied by a function that counts a specific value in this 2D container.



```cpp
GameboyBuffer SystemUtils::GetPixels(const ivec2& startPos, const ivec2& endPos, const GameboyBuffer& pixelBuffer)
{
    const int xSize{ endPos.x - startPos.x + 1 }, ySize{ endPos.y - startPos.y + 1 };
    GameboyBuffer specificPixels(_Count: xSize, _Val: std::vector<uint8_t>(_Count: ySize));

    for (int x{}; x < xSize; x++)
    {
        for (int y{}; y < ySize; y++)
        {
            specificPixels[x][y] = pixelBuffer[startPos.x + x][startPos.y + y];
        }
    }

    return specificPixels;
}
```

1.2. RECOGNIZE THE SCREENS

To find out what screen it is, there's several parts that are useful to us. The most important one is the top row. Despite the pixel buffer offset, the top row is still the most useful to use to find out what screen it could be.



Based solely on the first row, we can split the screens in 3 groups. The ones with a fully white row, the ones with a fully black row and the ones with a specific number of white pixels. There's also an additional "escape" for special cases which we can ignore.

COMPLETELY BLACK

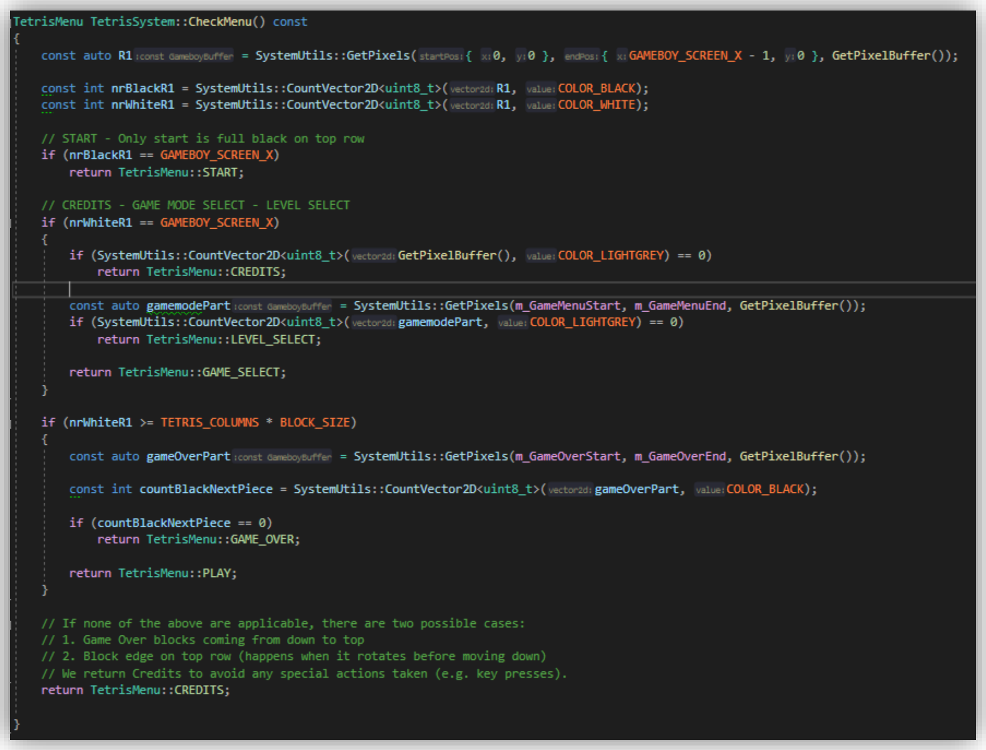
It might be surprising at first glance, but only the start screen (the one with the large TETRIS) is the only one with only black pixels in its top row. Although the cause might be the pixel buffer offset when it comes to the game select and level select not being under this group.

COMPLETELY WHITE

The most obvious screen falling under this group is the Credits screen as it only consists out of white and black pixels. That's how we can know it is the Credits screen rather than the other possible screens.
As explained above, the game and level select fall under this group. The most notable difference is the white and black area on the level select letting you know what the previously selected game mode is.

MIXED

The only ones mixed are the "playing" screen and the game over screen. They are incredibly similar, but the main difference is that the "next piece" location is completely white when the game is over.

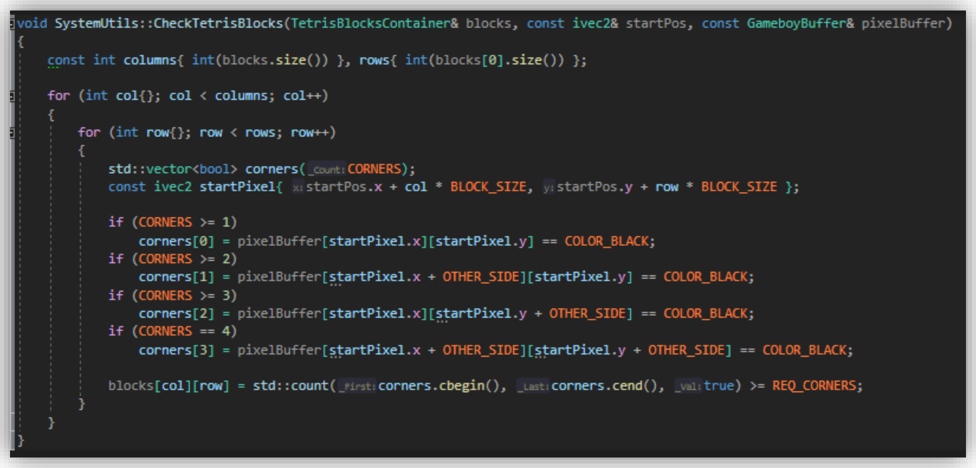


```cpp
TetrisMenu TetrisSystem::CheckMenu() const
{
    const auto R1 = SystemUtils::GetPixels({ 0, 0 }, { GAMEBOY_SCREEN_X - 1, 0 }, GetPixelBuffer());

    const int nrBlackR1 = SystemUtils::CountVector2D<uint8_t>(R1, COLOR_BLACK);
    const int nrWhiteR1 = SystemUtils::CountVector2D<uint8_t>(R1, COLOR_WHITE);

    // START - Only start is full black on top row
    if (nrBlackR1 == GAMEBOY_SCREEN_X)
        return TetrisMenu::START;

    // CREDITS - GAME MODE SELECT - LEVEL SELECT
    if (nrWhiteR1 == GAMEBOY_SCREEN_X)
    {
        if (SystemUtils::CountVector2D<uint8_t>(GetPixelBuffer(), COLOR_LIGHTGREY) == 0)
            return TetrisMenu::CREDITS;

        const auto gamemodePart = SystemUtils::GetPixels(m_GameMenuStart, m_GameMenuEnd, GetPixelBuffer());
        if (SystemUtils::CountVector2D<uint8_t>(gamemodePart, COLOR_LIGHTGREY) == 0)
            return TetrisMenu::LEVEL_SELECT;

        return TetrisMenu::GAME_SELECT;
    }

    if (nrWhiteR1 >= TETRIS_COLUMNS * BLOCK_SIZE)
    {
        const auto gameOverPart = SystemUtils::GetPixels(m_GameOverStart, m_GameOverEnd, GetPixelBuffer());

        const int countBlackNextPiece = SystemUtils::CountVector2D<uint8_t>(gameOverPart, COLOR_BLACK);

        if (countBlackNextPiece == 0)
            return TetrisMenu::GAME_OVER;

        return TetrisMenu::PLAY;
    }

    // If none of the above are applicable, there are two possible cases:
    // 1. Game Over blocks coming from down to top
    // 2. Block edge on top row (happens when it rotates before moving down)
    // We return Credits to avoid any special actions taken (e.g. key presses).
    return TetrisMenu::CREDITS;

}
```

1.3. RECOGNIZE BLOCKS AND PIECES

RECOGNIZE BLOCKS

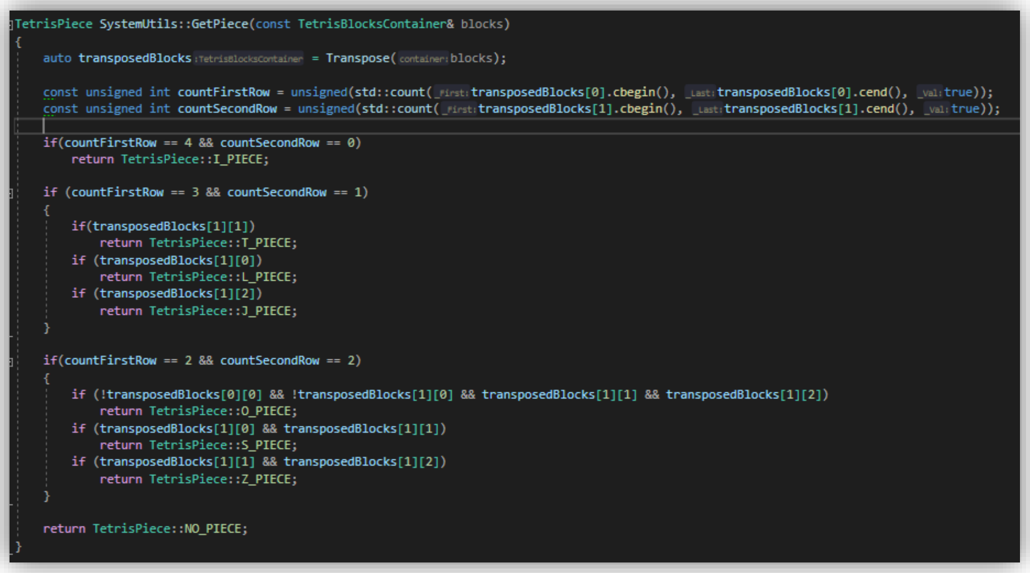
Recognizing blocks such as in the Playfield, Current Piece and Next Piece are handled slightly differently. To know if a block is there, we need to check the borders, specifically the corners. We need to watch out for white lines that appear through the Gameboys pixel buffer as that also affects ours. Through testing, I figured out checking all 4 corners and requiring at least 2 to be black, is the perfect balance and we get the data we need correctly. The function takes in the container that will fill in the blocks, the starting pixel and the pixel buffer from the system. The blocks get filled, but the container should already have its necessary size as it gets used to see how far we need to check. The block size is already pre-defined globally as a Tetris Block (one individual block) is 8 pixels wide and high.



```cpp
void SystemUtils::CheckTetrisBlocks(TetrisBlocksContainer& blocks, const ivec2& startPos, const GameboyBuffer& pixelBuffer)
{
    const int columns{ int(blocks.size()) }, rows{ int(blocks[0].size()) };

    for (int col{}; col < columns; col++)
    {
        for (int row{}; row < rows; row++)
        {
            std::vector<bool> corners(_Count: CORNERS);
            const ivec2 startPixel{ x: startPos.x + col * BLOCK_SIZE, y: startPos.y + row * BLOCK_SIZE };

            if (CORNERS >= 1)
                corners[0] = pixelBuffer[startPixel.x][startPixel.y] == COLOR_BLACK;
            if (CORNERS >= 2)
                corners[1] = pixelBuffer[startPixel.x + OTHER_SIDE][startPixel.y] == COLOR_BLACK;
            if (CORNERS >= 3)
                corners[2] = pixelBuffer[startPixel.x][startPixel.y + OTHER_SIDE] == COLOR_BLACK;
            if (CORNERS == 4)
                corners[3] = pixelBuffer[startPixel.x + OTHER_SIDE][startPixel.y + OTHER_SIDE] == COLOR_BLACK;

            blocks[col][row] = std::count(_First: corners.cbegin(), _Last: corners.cend(), _Val: true) >= REQ_CORNERS;
        }
    }
}
```

RECOGNIZE PIECE

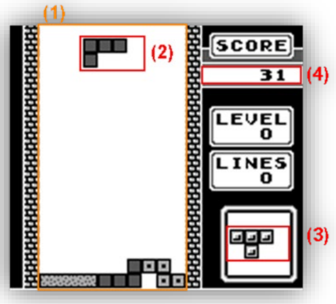
Recognizing a piece is mainly manual work by checking where a block is. The Current Piece and Next Piece are always in a container of 4 columns by 2 rows. With just that data, we know enough to figure out what block it is. For example, the only piece with the first row filled, is the I-Piece.
Note: The container is transposed as it is easier to count based on the rows, rather than columns in this case.



```cpp
TetrisPiece SystemUtils::GetPiece(const TetrisBlocksContainer& blocks)
{
    auto transposedBlocks :TetrisBlocksContainer = Transpose(container: blocks);

    const unsigned int countFirstRow = unsigned(std::count(_First: transposedBlocks[0].cbegin(), _Last: transposedBlocks[0].cend(), _Val: true));
    const unsigned int countSecondRow = unsigned(std::count(_First: transposedBlocks[1].cbegin(), _Last: transposedBlocks[1].cend(), _Val: true));

    if(countFirstRow == 4 && countSecondRow == 0)
        return TetrisPiece::I_PIECE;

    if (countFirstRow == 3 && countSecondRow == 1)
    {
        if(transposedBlocks[1][1])
            return TetrisPiece::T_PIECE;
        if (transposedBlocks[1][0])
            return TetrisPiece::L_PIECE;
        if (transposedBlocks[1][2])
            return TetrisPiece::J_PIECE;
    }

    if(countFirstRow == 2 && countSecondRow == 2)
    {
        if (!transposedBlocks[0][0] && !transposedBlocks[1][0] && transposedBlocks[1][1] && transposedBlocks[1][2])
            return TetrisPiece::O_PIECE;
        if (transposedBlocks[1][0] && transposedBlocks[1][1])
            return TetrisPiece::S_PIECE;
        if (transposedBlocks[1][1] && transposedBlocks[1][2])
            return TetrisPiece::Z_PIECE;
    }

    return TetrisPiece::NO_PIECE;
}
```

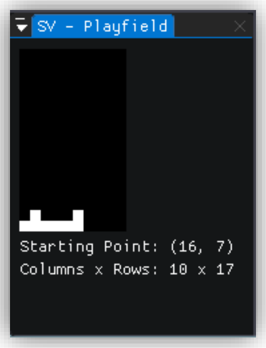
1.4. PLAY DATA

SCORE (4)



We'll start with the score. The score might not seem important at first, but due to its nature we can use it to know when we need to update our data. When we "soft-drop" a piece, a score is given depending on the number of blocks it was soft-dropped. The exact number doesn't matter, but knowing the score changed is more than enough. That way when a new piece arrives, we can update everything and calculate the next best move.

PLAYFIELD (1)



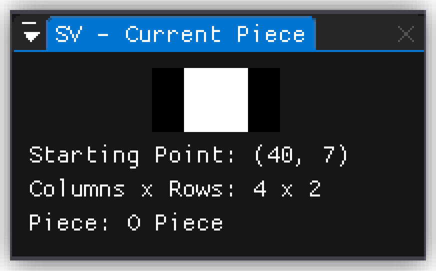
SV - Playfield

```
Starting Point: (16, 7)
Columns x Rows: 10 x 17
```

The playfield data is stored in a 2D container of 10 columns by 17 rows. We skip the first row of the playfield due to a pixel buffer offset from the Gameboy emulator itself. The first row wouldn't have been very useful anyway as the Current Piece spawns from the second row. We call the function with our container and the start position (**Recognize Blocks**). Without the pixel buffer offset, the y-position would've been 8.

In the playfield data, it also counts the current piece. To remove this, we set the middle 4 columns of the two first rows (in our data) to false. While it might remove actual blocks not part of the current piece, it is pointless to fix it for those edge cases because it would mean game over anyway.

CURRENT PIECE (2)

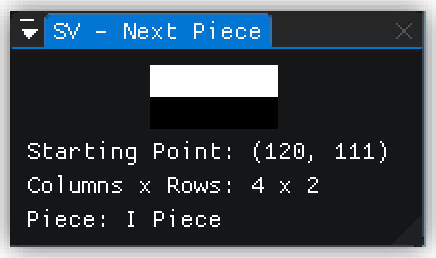


SV - Current Piece

```
Starting Point: (40, 7)
Columns x Rows: 4 x 2
Piece: O Piece
```

The current piece is located on the playfield, but the piece always spawns at a specific position. The piece specifically uses the middle 4 columns and the top two rows (from our data). To be able to find what piece it is, we first get the blocks based on the container and position we give it (**Recognize Blocks**). We can then find what Tetris piece it is based on those blocks (**Recognize Piece**).

NEXT PIECE (3)



SV - Next Piece

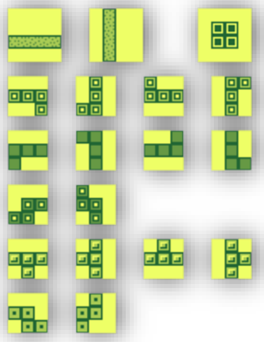
```
Starting Point: (120, 111)
Columns x Rows: 4 x 2
Piece: I Piece
```

The next piece is in the bottom right corner of the screen in its own little box. To get the needed data is very similar to the current piece. The main – and probably the only – difference is the start position is different. We still use the two functions to get the blocks (**Recognize Blocks**) and to recognize the piece (**Recognize Piece**).

2. ALGORITHM DEVELOPMENT

2.1. CALCULATING A MOVE

The algorithm is all about calculating a score based on what a move would do, but first we'll need to be able to place a possible move.

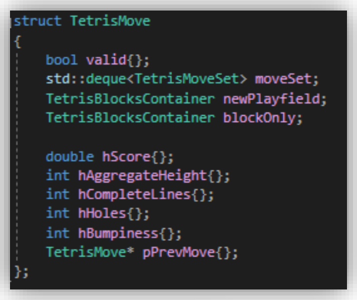


I manually created all the possible rotations of every piece in the form of a 2D container and the amount that specific rotation can move left and right. It would be more calculations to rotate it and it would be even more work to find how many times it can move left or right. Initializing it manually spares us possible issues there.

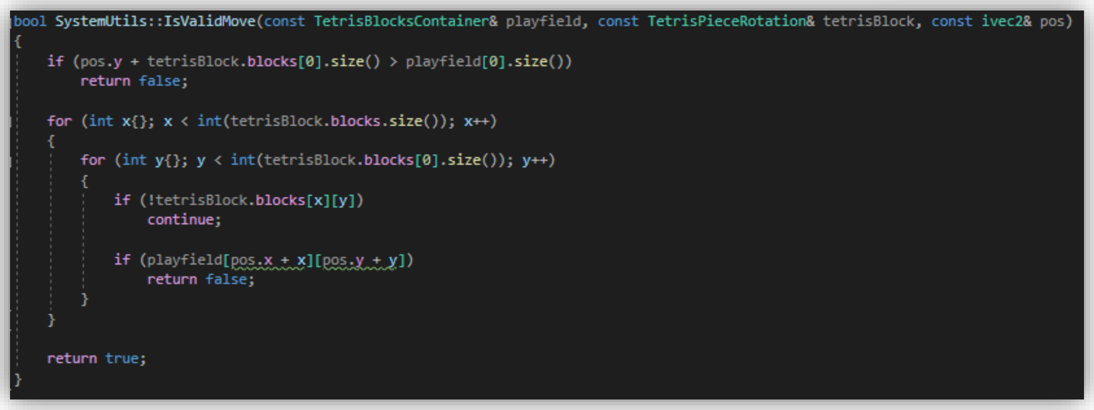
We'll focus on creating a move with a given movement and rotation.

A VALID MOVE

We first need a way to check if a move is valid. We can do this by comparing our rotated piece container with the positions on the playfield. If all the positions where a block would be are empty, we can assume we can place the piece there. We can keep calling this function with a further down row to find the final position we can place the Tetris piece in.



```
struct TetrisMove
{
    bool valid{};
    std::deque<TetrisMoveSet> moveSet;
    TetrisBlocksContainer newPlayfield;
    TetrisBlocksContainer blockOnly;

    double hScore{};
    int hAggregateHeight{};
    int hCompleteLines{};
    int hHoles{};
    int hBumpiness{};
    TetrisMove* pPrevMove{};
};
```

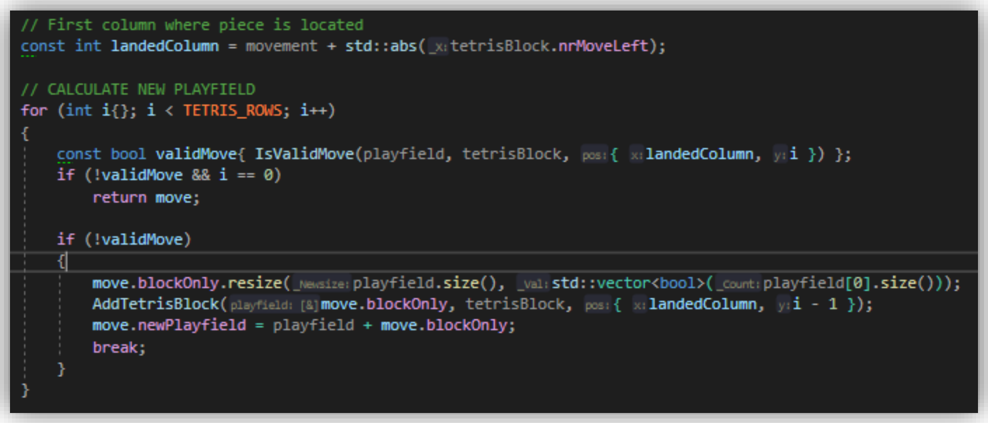


```
bool SystemUtils::IsValidMove(const TetrisBlocksContainer& playfield, const TetrisPieceRotation& tetrisBlock, const ivec2& pos)
{
    if (pos.y + tetrisBlock.blocks[0].size() > playfield[0].size())
        return false;

    for (int x{}; x < int(tetrisBlock.blocks.size()); x++)
    {
        for (int y{}; y < int(tetrisBlock.blocks[0].size()); y++)
        {
            if (!tetrisBlock.blocks[x][y])
                continue;

            if (playfield[pos.x + x][pos.y + y])
                return false;
        }
    }

    return true;
}
```

NEW PLAYFIELD

By going down the rows, checking if a move is valid, we can find the final position the piece could be located in. We can add our piece to the existing playfield (a copy representing the move).

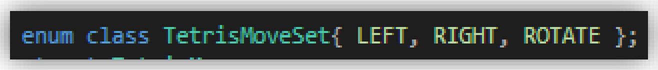
Note: blockOnly is a container purely containing the position of the block and is helpful to render the piece in a different color.



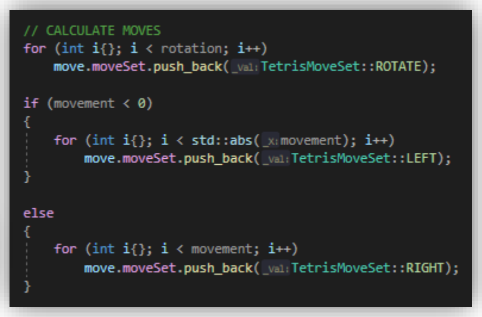
```cpp
// First column where piece is located
const int landedColumn = movement + std::abs(tetrisBlock.nrMoveLeft);

// CALCULATE NEW PLAYFIELD
for (int i{}; i < TETRIS_ROWS; i++)
{
    const bool validMove{ IsValidMove(playfield, tetrisBlock, { landedColumn, i }) };
    if (!validMove && i == 0)
        return move;

    if (!validMove)
    {
        move.blockOnly.resize(playfield.size(), std::vector<bool>(playfield[0].size()));
        AddTetrisBlock(move.blockOnly, tetrisBlock, { landedColumn, i - 1 });
        move.newPlayfield = playfield + move.blockOnly;
        break;
    }
}
```

CALCULATING THE MOVESET

The moveset is very simple. It's pushing a simple enum value to a deque which will later be converted to the keys the system will press.



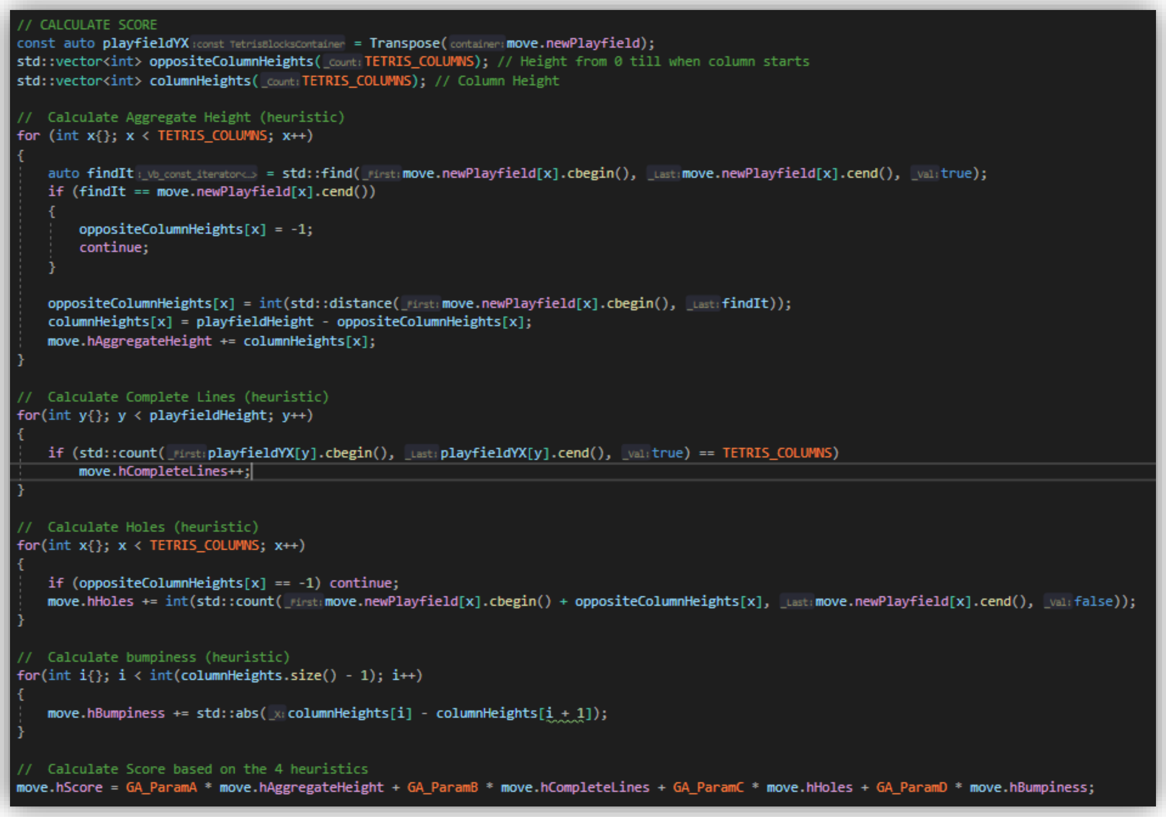
```cpp
enum class TetrisMoveSet{ LEFT, RIGHT, ROTATE };
```



```cpp
// CALCULATE MOVES
for (int i{}; i < rotation; i++)
    move.moveSet.push_back(TetrisMoveSet::ROTATE);

if (movement < 0)
{
    for (int i{}; i < std::abs(movement); i++)
        move.moveSet.push_back(TetrisMoveSet::LEFT);
}

else
{
    for (int i{}; i < movement; i++)
        move.moveSet.push_back(TetrisMoveSet::RIGHT);
}
```

CALCULATING THE SCORE

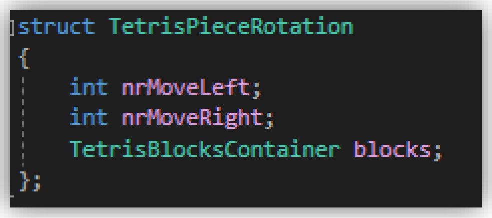
The most important part is to calculate the score based on the heuristics we've seen for the algorithm while researching. However, we can already calculate some data in one heuristic that would be useful for another to not waste many calculations as calculating a lot of moves would already take some time.



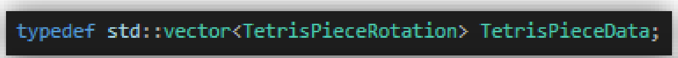
```
// CALCULATE SCORE
const auto playfieldYX = Transpose(move.newPlayfield);
std::vector<int> oppositeColumnHeights(TETRIS_COLUMNS); // Height from 0 till when column starts
std::vector<int> columnHeights(TETRIS_COLUMNS); // Column Height

//  Calculate Aggregate Height (heuristic)
for (int x{}; x < TETRIS_COLUMNS; x++)
{
    auto findIt = std::find(move.newPlayfield[x].cbegin(), move.newPlayfield[x].cend(), true);
    if (findIt == move.newPlayfield[x].cend())
    {
        oppositeColumnHeights[x] = -1;
        continue;
    }

    oppositeColumnHeights[x] = int(std::distance(move.newPlayfield[x].cbegin(), findIt));
    columnHeights[x] = playfieldHeight - oppositeColumnHeights[x];
    move.hAggregateHeight += columnHeights[x];
}

//  Calculate Complete Lines (heuristic)
for(int y{}; y < playfieldHeight; y++)
{
    if (std::count(playfieldYX[y].cbegin(), playfieldYX[y].cend(), true) == TETRIS_COLUMNS)
        move.hCompleteLines++;
}

//  Calculate Holes (heuristic)
for(int x{}; x < TETRIS_COLUMNS; x++)
{
    if (oppositeColumnHeights[x] == -1) continue;
    move.hHoles += int(std::count(move.newPlayfield[x].cbegin() + oppositeColumnHeights[x], move.newPlayfield[x].cend(), false));
}

//  Calculate bumpiness (heuristic)
for(int i{}; i < int(columnHeights.size() - 1); i++)
{
    move.hBumpiness += std::abs(columnHeights[i] - columnHeights[i + 1]);
}

//  Calculate Score based on the 4 heuristics
move.hScore = GA_ParamA * move.hAggregateHeight + GA_ParamB * move.hCompleteLines + GA_ParamC * move.hHoles + GA_ParamD * move.hBumpiness;
```

2.2. GETTING ALL MOVES

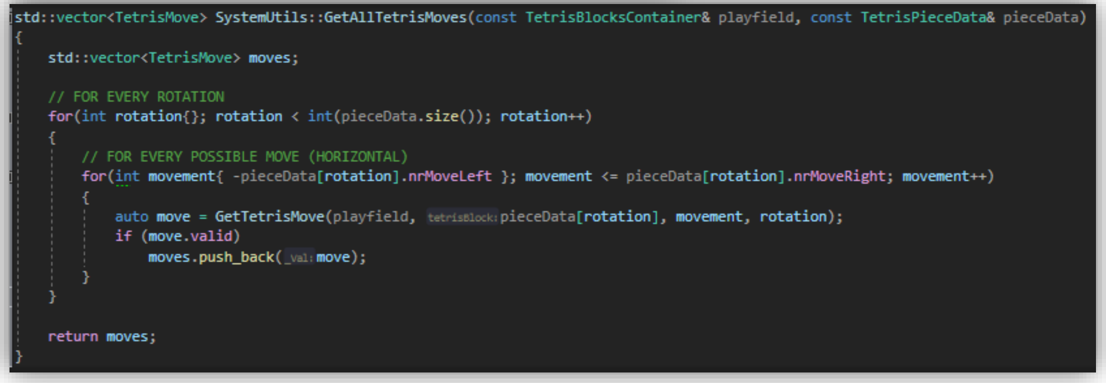
The obvious next step would be to calculate all moves for a specific piece. We can easily request the needed Piece Data as we manually created them before. Using that we easily know how many rotations there are (vector size), the amount it can move left and right and the 2D container form of the block itself.



```
struct TetrisPieceRotation
{
    int nrMoveLeft;
    int nrMoveRight;
    TetrisBlocksContainer blocks;
};
```



```
typedef std::vector<TetrisPieceRotation> TetrisPieceData;
```



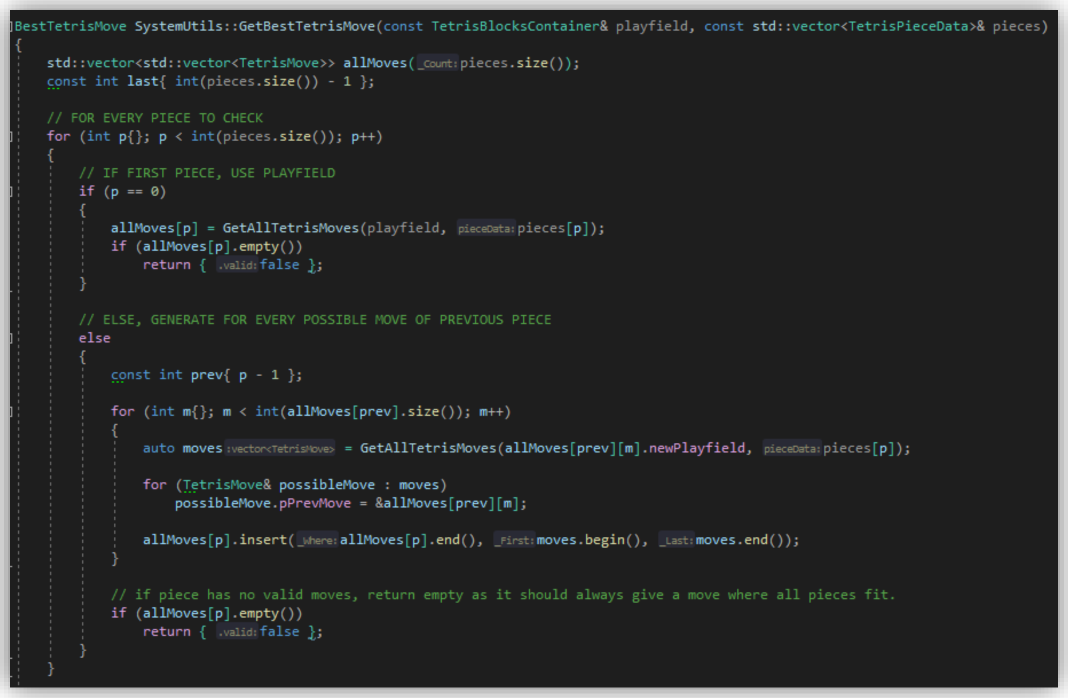
```
std::vector<TetrisMove> SystemUtils::GetAllTetrisMoves(const TetrisBlocksContainer& playfield, const TetrisPieceData& pieceData)
{
    std::vector<TetrisMove> moves;

    // FOR EVERY ROTATION
    for(int rotation{}; rotation < int(pieceData.size()); rotation++)
    {
        // FOR EVERY POSSIBLE MOVE (HORIZONTAL)
        for(int movement{ -pieceData[rotation].nrMoveLeft }; movement <= pieceData[rotation].nrMoveRight; movement++)
        {
            auto move = GetTetrisMove(playfield, pieceData[rotation], movement, rotation);
            if (move.valid)
                moves.push_back(move);
        }
    }

    return moves;
}
```

2.3. GETTING THE BEST MOVE

GENERATING ALL MOVES ACROSS DEPTH

We'll first need to generate all the moves. However, now it isn't just for one piece but for the next piece and possible next ones as well. (this function was made with multiple next pieces in mind to possibly get even better results)

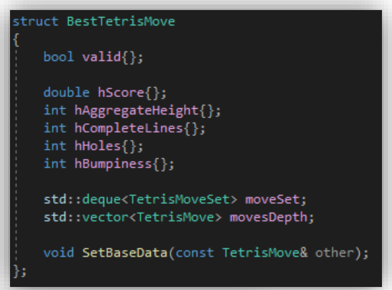
For the first piece, we'll still use the initial playfield given, but for the next pieces we'll use the previously calculated moves and keep going like that. However, if a piece can't find a valid move anymore, the function will drop out and there's no possible move for the series of pieces. This could happen if the game is near end and the playfield would be almost full.



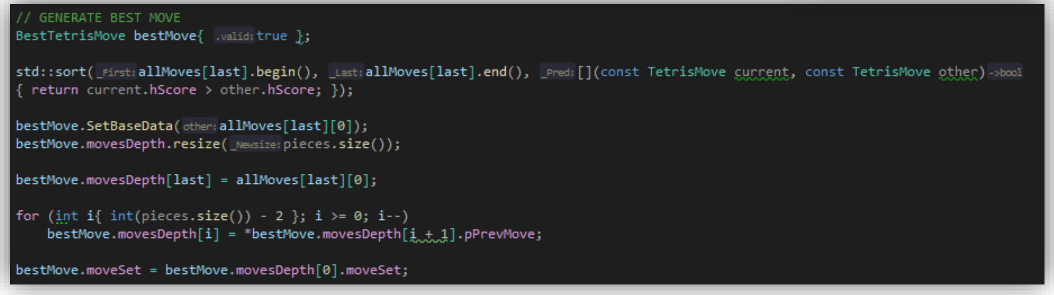
```cpp
BestTetrisMove SystemUtils::GetBestTetrisMove(const TetrisBlocksContainer& playfield, const std::vector<TetrisPieceData>& pieces)
{
    std::vector<std::vector<TetrisMove>> allMoves(pieces.size());
    const int last{ int(pieces.size()) - 1 };

    // FOR EVERY PIECE TO CHECK
    for (int p{}; p < int(pieces.size()); p++)
    {
        // IF FIRST PIECE, USE PLAYFIELD
        if (p == 0)
        {
            allMoves[p] = GetAllTetrisMoves(playfield, pieces[p]);
            if (allMoves[p].empty())
                return { false };
        }

        // ELSE, GENERATE FOR EVERY POSSIBLE MOVE OF PREVIOUS PIECE
        else
        {
            const int prev{ p - 1 };

            for (int m{}; m < int(allMoves[prev].size()); m++)
            {
                auto moves = GetAllTetrisMoves(allMoves[prev][m].newPlayfield, pieces[p]);

                for (TetrisMove& possibleMove : moves)
                    possibleMove.pPrevMove = &allMoves[prev][m];

                allMoves[p].insert(allMoves[p].end(), moves.begin(), moves.end());
            }

            // if piece has no valid moves, return empty as it should always give a move where all pieces fit.
            if (allMoves[p].empty())
                return { false };
        }
    }
```

CREATING THE BEST MOVE

Now that we have all moves of the final piece, we can sort it by the best score. The structure of the best Tetris move is slightly different as it also contains the moves depth so we can visually see what moves are in between. The score data will be that of the final move, but the move set is that one of the first move.



```cpp
struct BestTetrisMove
{
    bool valid{};

    double hScore{};
    int hAggregateHeight{};
    int hCompleteLines{};
    int hHoles{};
    int hBumpiness{};

    std::deque<TetrisMoveSet> moveSet;
    std::vector<TetrisMove> movesDepth;

    void SetBaseData(const TetrisMove& other);
};
```

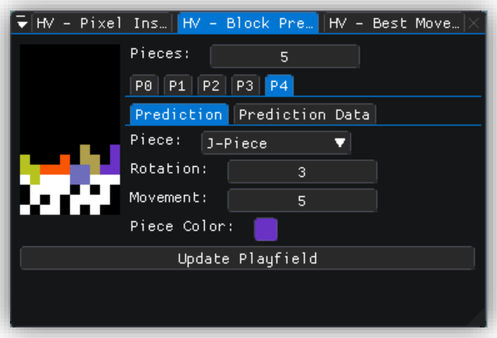


```cpp
// GENERATE BEST MOVE
BestTetrisMove bestMove{ true };

std::sort(allMoves[last].begin(), allMoves[last].end(), [](const TetrisMove current, const TetrisMove other)
{ return current.hScore > other.hScore; });

bestMove.SetBaseData(allMoves[last][0]);
bestMove.movesDepth.resize(pieces.size());

bestMove.movesDepth[last] = allMoves[last][0];

for (int i{ int(pieces.size()) - 2 }; i >= 0; i--)
    bestMove.movesDepth[i] = *bestMove.movesDepth[i + 1].pPrevMove;

bestMove.moveSet = bestMove.movesDepth[0].moveSet;
```

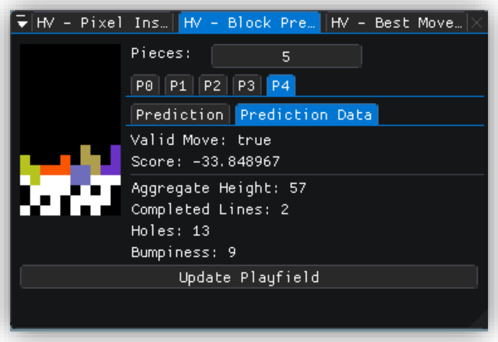
2.4. VISUAL REPRESENTATION

A POSSIBLE MOVE

This is a Helper View created to check multiple moves after each other where we give in the manual values. Each move is built upon the other (if valid). This was used to compare the scores to the best move Helper View to look for bugs. You can also use the current playfield (by pressing the button) to check for yourself when testing a possible move.



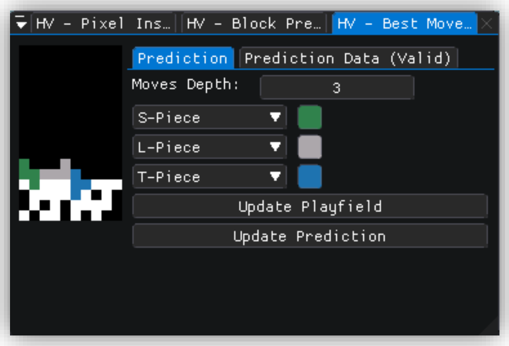
Pieces: 5
Piece: J-Piece
Rotation: 3
Movement: 5
Piece Color:



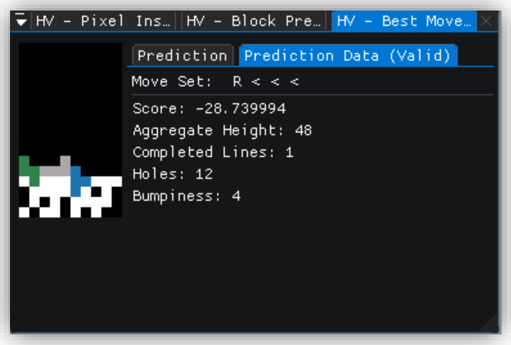
Pieces: 5
Valid Move: true
Score: -33.848967
Aggregate Height: 57
Completed Lines: 2
Holes: 13
Bumpiness: 9

THE BEST MOVE

The Helper View that will give you the best move based on the algorithm on the current playfield (if updated) with the given pieces (up to 5 as 3 can already take a few seconds to calculate). With this, you can predict a possible future, at least when it comes to Tetris.



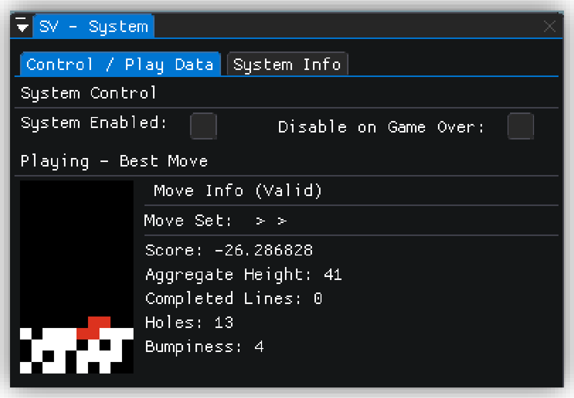
Moves Depth: 3
S-Piece
L-Piece
T-Piece



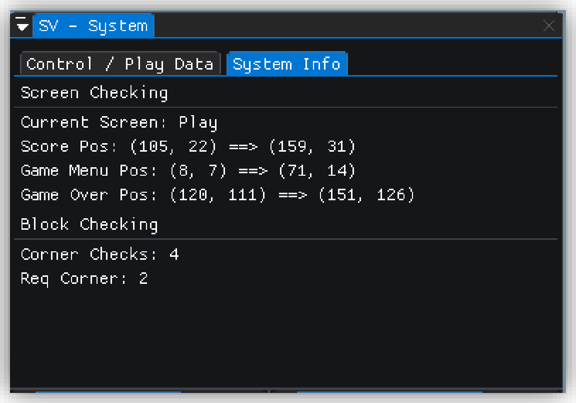
Move Set: R < < <
Score: -28.739994
Aggregate Height: 48
Completed Lines: 1
Holes: 12
Bumpiness: 4

SYSTEM CONTROL

The System itself also shows some valuable info such as the move the system would do and other info.



System Control
System Enabled:
Disable on Game Over:
Playing - Best Move
Move Info (Valid)
Move Set: > >
Score: -26.286828
Aggregate Height: 41
Completed Lines: 0
Holes: 13
Bumpiness: 4



Screen Checking
Current Screen: Play
Score Pos: (105, 22) ==> (159, 31)
Game Menu Pos: (8, 7) ==> (71, 14)
Game Over Pos: (120, 111) ==> (151, 126)
Block Checking
Corner Checks: 4
Req Corner: 2

3. A FINAL NOTE

This concludes the detailed info about the system. It doesn’t show everything that’s been coded but you can find everything on the Github repository in case you wish to delve deeper into the code behind the system. The paper mainly shows the important parts that really make up the heart of the system.

4. LIMITATIONS

By only being able to use the pixels, there’s some limitations when it comes to the system.
As time goes on and the level increases of the Tetris game, the game speeds up and the system seems to start failing to recognize the current piece.

The system still sometimes fails to press the button albeit being rare, perhaps the FPS still does not match up with the Gameboy emulator as the emulator perhaps runs multi-threaded.

5. FUTURE WORK

There could be so much more done with this, such as creating additional heuristics and being able to tweak them while the system continues to play Tetris. You could also have an actual robot press in the buttons based on what the system could give. (this would also avoid system-based errors)

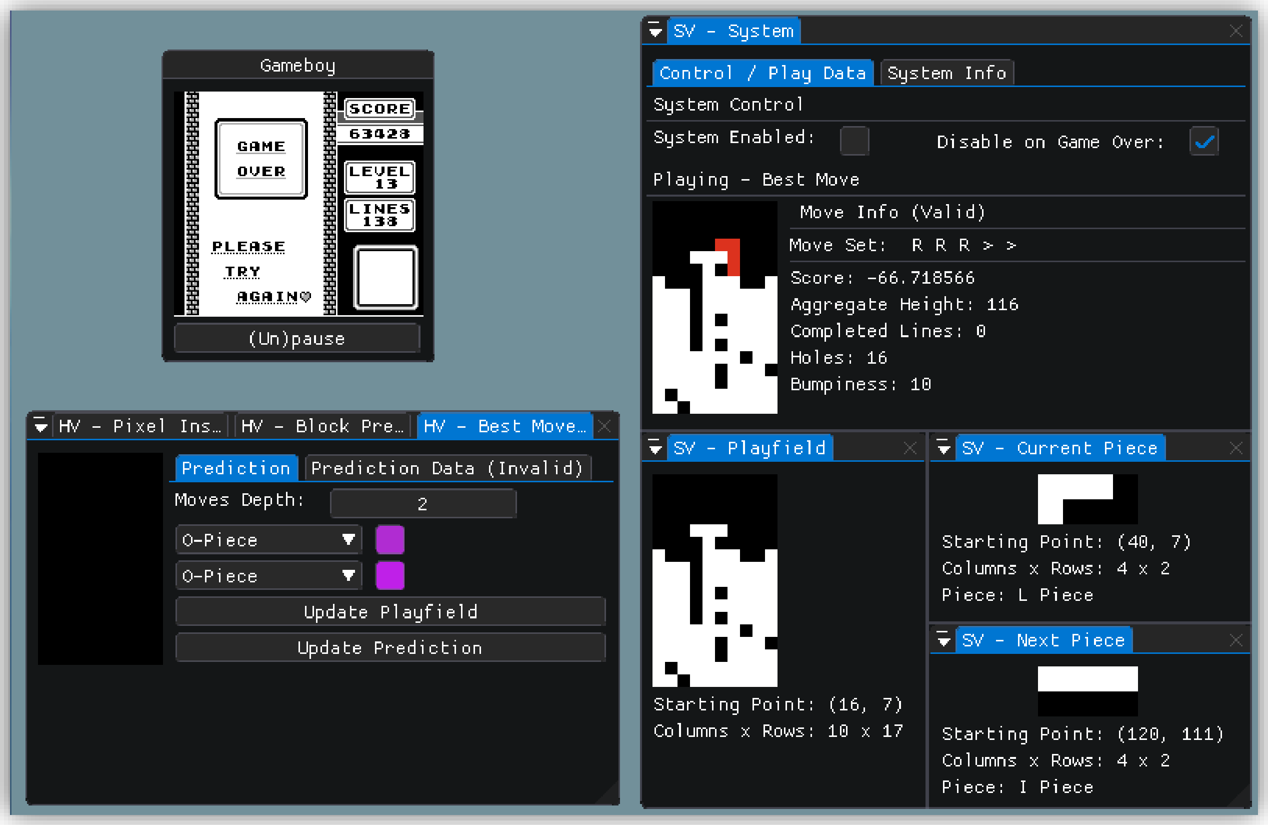
There could still be more work done on the system itself as improving the performance and fixing later issues or limitations that could somehow be avoided.

CONCLUSION

We have been able to handle every part of the system without a lot of effort and even with the several bugs in the emulator, we could bypass them. The algorithm and making sure the data from the system was updated at the right moment was a lot tougher as we couldn't keep generating the move or keep updating the data as we would lose track of the current piece as it moves down.

However, the limitations of only having access to the pixels are quite visible as the game keeps going. As the level increases, so does the speed and at some point, the system struggles to keep the current piece correct. Even if the system would run at a higher framerate, the issue still occurs. Perhaps the emulator is the issue as it might run multi-threaded or perhaps there wasn't enough testing of the system at higher speeds.

In the end, we have a working Tetris system with limitations, but it can still reach quite the high score without sweating. It was still a nice experiment even with the limitations.



Moves Depth: 2
O-Piece
O-Piece

System Control
System Enabled: ☐ Disable on Game Over: ✓
Playing - Best Move
Move Info (Valid)
Move Set: R R R > >
Score: -66.718566
Aggregate Height: 116
Completed Lines: 0
Holes: 16
Bumpiness: 10

Starting Point: (16, 7)
Columns x Rows: 10 x 17

Starting Point: (40, 7)
Columns x Rows: 4 x 2
Piece: L Piece

Starting Point: (120, 111)
Columns x Rows: 4 x 2
Piece: I Piece

A result and snap from the Tetris System.

REFERENCES

GAMEBOY EMULATION

Nazar, I. (n.d.). *GameBoy Emulation in JavaScript*. Retrieved from imrannazar: http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Graphics

TETRIS

Tetris (Game Boy) | Tetris Wiki | Fandom. (z.d.). Tetris Wiki. https://tetris.fandom.com/wiki/Tetris_(Game_Boy)

TETRIS®. (2020, 20 december). Nintendo of Europe GmbH. https://www.nintendo.nl/Games/Game-Boy/TETRIS--275924.html#Galerie

Scoring | Tetris Wiki | Fandom. (z.d.). Tetris Wiki. https://tetris.fandom.com/wiki/Scoring

TETRIS SYSTEM

da Silva, R. S., & Stubs Parpinelli, R. (2017). Playing the Original Game Boy Tetris Using a Real Coded Genetic Algorithm. 2017 Brazilian Conference on Intelligent Systems (BRACIS), 0. https://www.researchgate.net/publication/322321608_Playing_the_Original_Game_Boy_Tetris_Using_a_Real_Coded_Genetic_Algorithm

TETRIS ALGORITHM

Lee, Y. (2017, 14 juni). Tetris AI – The (Near) Perfect Bot. Code My Road. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

L. (z.d.). LeeYiyuan/tetrisai. GitHub. https://github.com/LeeYiyuan/tetrisai/blob/gh-pages/js/ai.js

Bergmark, M. (2015). [PDF] Tetris: A Heuristic Study : Using height-based weighing functions and breadth-first search heuristics for playing Tetris | Semantic Scholar. SemanticScholar. https://www.semanticscholar.org/paper/Tetris:-A-Heuristic-Study-:-Using-height-based-and-Bergmark/17644cdf501333a2495d0a05c65d2c0423c6fa6e