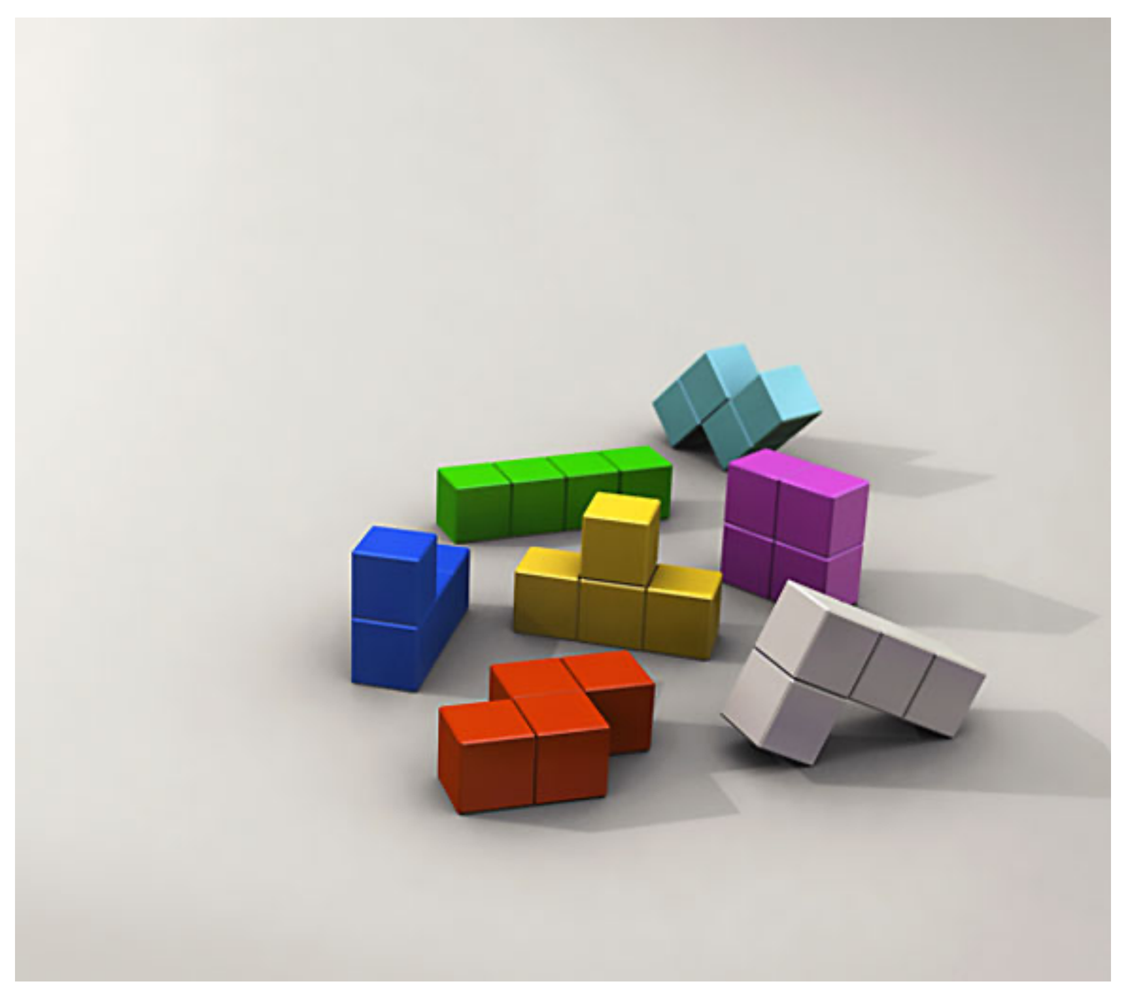
# *Building a Tetris Game in C*



*By Chad Jordan*

*July 18th, 2008*

# Table of Contents

Introduction ........ 1

A Brief Story of Tetris ........ 1

Formulating a Plan ........ 2

Understand the Problem ........ 2

Break Down the Components ........ 2

Design an Algorithm ........ 2

Plan the Program Structure ........ 3

Iterate and Test ........ 4

Finalize the Program ........ 4

C Programming for Game Development ........ 4

Why Procedural Programming? ........ 4

Setting Up the Project ........ 5

Preparing the C Project ........ 5

Implementing the Code ........ 6

Declaring Libraries & Defining Directives ........ 6

Declaring & Storing Data Types ........ 7

Clearing the Screen & Spawning Pieces ........ 9

Drawing the Next Piece & the Player Board ........ 10

Updating Score Logic ........ 12

Remove Full Lines Logic ........ 13

Implementing Collision Detection ........ 14

Collision with Rotation ........ 15

Keyboard Input & Resetting the Game ........ 16

Game Over & Finalizing ........ 17

Compiling & Done ........ 18

Conclusion ........ 19

# Introduction

In this guide, you will learn:

1) Algorithm design and decision making in order to formulate a Tetris game
2) A complete procedural programming approach using the C language for game development

Ever since its inception in 1984, Tetris has remained one of the most sought-after video games of all time selling more than 100 million copies world-wide. Alexey Pajitnov from Moscow, Russia was the original creator and programmer of Tetris who wrote the game in Pascal. This would later be sold to Nintendo for licensing rights and make him millions of dollars in sales. The success of Tetris would later pave the way for numerous other manifestations of the game across multiple consoles. The original NES and Game Boy release of the game would not happen until 1989.

My love of Tetris began in the Christmas of 1990 when I got it for Nintendo and Game Boy. The game was deeply addictive and all of my family members enjoyed it on these platforms. I never could understand how any of it worked behind the screen, I just knew I loved the game.

Even after all these years, I've still never been able to part with the game and I still enjoy playing them from time to time. After a number of years later, I had become determined to understand how games were made, and in 2001 I tried to start learning C programming. I only got so far on my own, but when I finally made it to college, I declared a major in computer science new media. It would still be a couple more years of hard work before I could build my first version of the block-dropping masterpiece, but on July $4^{th}$ of 2008, I had made a successful version of the game with all of the base-level features. At the time, I was not prepared to implement a graphics API so I figured colorful ASCii values in a console application would have to suffice. I wrote the program in C using the free Dev C++ compiler by Bloodshed Software on Windows. How is something like this done? It's time for a complete walkthrough of how I implemented my first Tetris game.

# Formulating a Plan

As any programmer knows, the general thought process of creating something first involves understanding the problem, then breaking down the components into an algorithm that will help you achieve the desired outcome, followed by implementation and testing. In the case of my Tetris game, I broke down the process as follows:

## 1. Understand the problem

- **Game Objectives:** *What is the goal of the game?*
  - Fitting Tetrominoes and clearing lines
  - Get the highest score
- **Game Rules:**
  - Tetrominoes fall from the top and can be moved, and rotated
  - Horizontal lines become cleared when filled across
  - As the score increases, levels increase, making the Tetrominoes drop faster
  - The game ends once the pieces stack to the top of the board

## 2. Break Down the Components

- **The Game Board:**
  - 2D Grid representing the playing field
  - Track where pieces are kept and cleared
- **Display on Screen:**
  - Title of the game
  - Current level
  - Next piece
  - Total score
- **Tetrominoes:**
  - 7 different shapes *(follows traditional tetromino shapes)*
  - Defined in a 4x4 grid for rotation in an array
- **Player Input:**
  - Allow movement left, right, down, and rotation
- **Game Logic:**
  - Collision detection for pieces
  - Line-clearing mechanism
  - Continually update game state
  - Handle timing such as rate of drop speed

## 3. Design an Algorithm

- **Board Initialization:**
  - Initialize a 2D array of size HEIGHT x WIDTH with all zeros.
- **Movement & Rotation of Pieces:**
  - **Movement –**
  - Check if the new position is valid (within bounds and not overlapping)
  - If valid, update the piece's position

- **Rotation –**
- Use matrix transposition and reversing rows/columns for 90° rotation
- Check collision after rotation

- <u>**Collision Detection:**</u>
  - Check for overlapped pieces within board boundaries

    *-(Pseudo Code)*
    ```
    For each cell in the pieces:
        Calculate the cell's position on the board.
        If the position is outside the board or overlaps an occupied cell:
            Return collision detected.
    ```
- <u>**Line Clearing:**</u>

  *-(Pseudo Code)*
  ```
  For each row on the board:
      Check if all cells are occupied.
      If yes:
          Clear the row.
          Shift rows above it down by one.
  ```
- <u>**Scoring and Level Progression:**</u>
  - Each cleared line increases the score
  - Every N lines cleared advances the level and increases the speed
- <u>**Game Over Detection:**</u>

  *-(Pseudo Code)*
  ```
  If a piece spawns and immediately collides with the board:
      Trigger the game over state
  ```

## 4. <u>Plan the Program Structure</u>

- <u>**Divide the program into modular functions:**</u>
  - Initialization
  - initBoard(): Sets up the game board
  - spawnPiece(): Generates a random Tetromino at the top
- <u>**Game Logic:**</u>
  - movePiece(direction): Handles movement
  - rotatePiece(): Rotates the piece
  - checkCollision(): Ensures valid moves
  - updateGame(): Advances the game state (piece falling, line clearing)
- <u>**Display:**</u>
  - drawBoard(): Renders the board.
  - drawNextPiece(): Displays the upcoming Tetromino
- <u>**User Input:**</u>
  - handleInput(): Processes player input

5. **Iterate and Test**
   - **Test Each Component Individually (IE, collision detection, line clearing)**
   - **Integrate Components Incrementally**
   - **Test Edge Cases:**
     - Rotate near boundaries
     - Clear single or multiple lines at once
     - Rapid input causing unexpected behavior

6. **Finalize the Program**
   - **Ensure the Game Runs Without Crashing or Freezing**
   - **Add Comments to the Code for Clarity and Good Practice**
   - **Execute and Package the Program**

## C Programming for Game Development

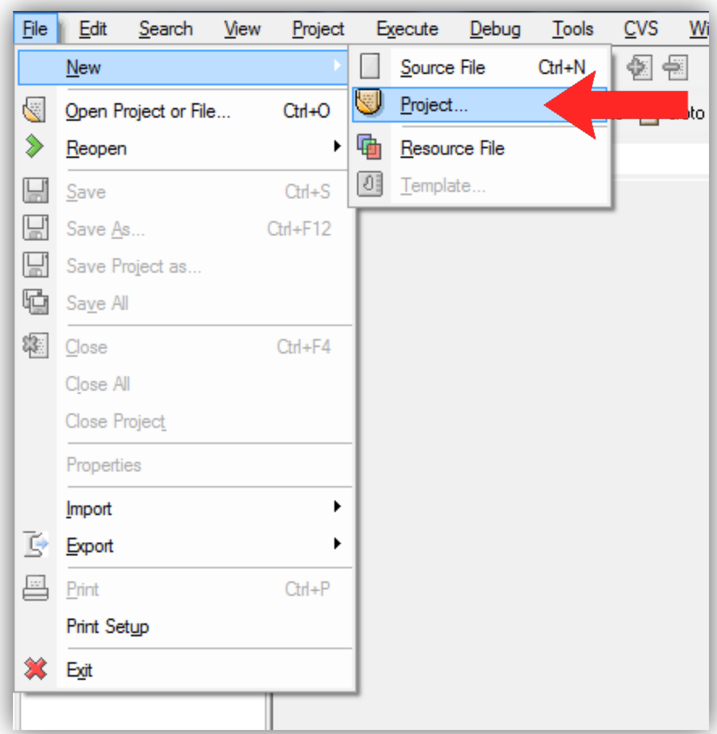
At this stage, the basis of the program structure has been drawn out in front of you for how the program will need to be approached, built and executed. Now you face the '*what'* and the '*why'* behind which programming language you will choose to write the program. I used the C language for several reasons over other languages:

1) **Advantage of simplicity and control**
   - Low-level access to memory and system resources, making it ideal for smaller games where fine-tuned control over hardware is crucial
   - A non-object-oriented process reduces overhead in memory and execution, leading to potentially faster performance
2) **Procedural Programming**
   - C has a more structured flow, which is better for small to medium-sized projects like a Tetris game
   - Modular design allows you to break down the game into smaller, separate files for more sustainable code
3) **Performance**
   - C has minimal runtime overhead. **Example:** Features like virtual tables *(used in C++ for polymorphism)* and complex object management are absent, resulting in leaner executables
4) **Control Over Resources**
   - C provides explicit control over memory allocation and deallocation using *malloc* and *free*. This avoids the added abstractions of C++ memory management, which may incur overhead
5) **Direct Hardware Access**
   - C is closer to assembly than C++, making it easier to optimize for specific hardware, which is often critical for embedded systems

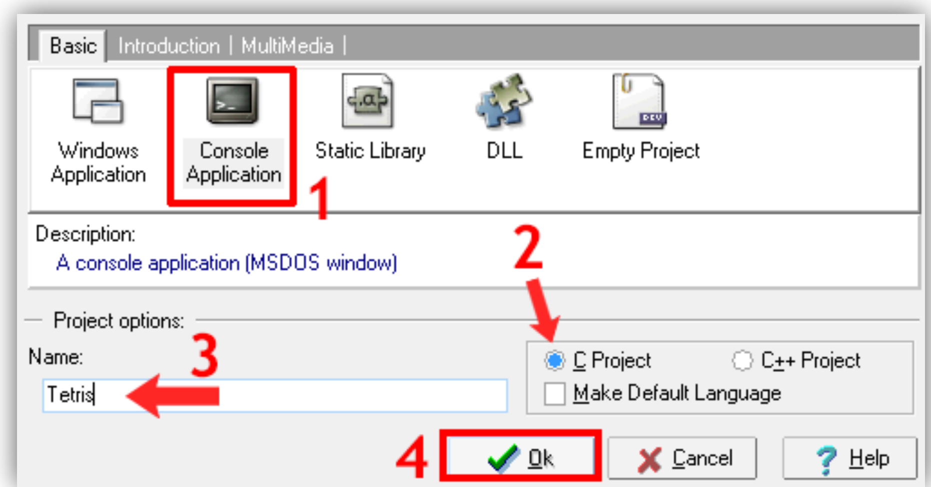
## Setting Up the Project

As mentioned in my introduction, I'm using the free Dev C++ compiler in Windows. I setup a standard Dev Project as a console application in the C language, and with no prior exposure or experience with graphics APIs, ASCii values would be a traditional go-to solution for simplicity and visualization.

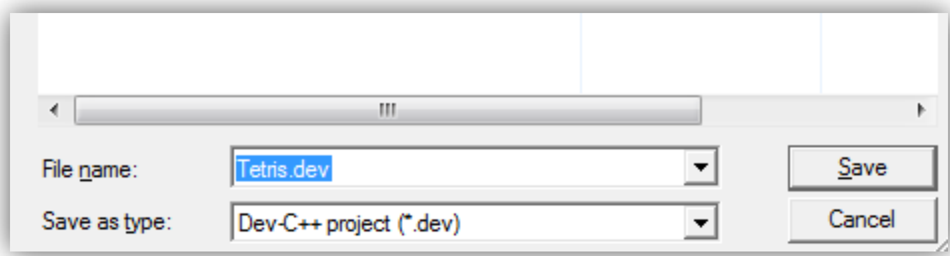
**Step 1.** With Dev C++ installed, it's simply a matter of setting up a basic project in C and this is done by opening up the application, and clicking on **File** > **New** > **Project**.




**Step 2.** Single-click **Console Application**, Select **C Project**, type a name in the **Name** field, and click **Ok**.



**Step 3.** On the next screen you'll save the dev project to whatever directory you have created, and select the filename you wish for your project and click **Save**.




**Step 4.** In the left window pane you will see a small hierarchy of the dev project that you can expand displaying the connected main.c file. This can be renamed to Tetris.c or any name you wish. With the default code provided in the project window to the right, you can simply highlight and delete from line 4, down if you want. You will still need the **stdio** and **stdlib** header files declared at the top.

Tetris
main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6
7   system("PAUSE");
8   return 0;
9 }
```

With all of these settings in place, it's time to start writing the code!

# Implementing the Code

Aside from the default libraries at the top, the ***time*** library seeds the random number generator that I'll be using for my score tracker, the rate in which the pieces fall, and other time-related calculations. ***Conio*** is short for *"console input/output"* which will handle various console-related operations. The ***Windows*** header file will serve as an interface to the windows API allowing function declarations with required data types and macros that allow C programs to interact with the Windows operating system. This file is important for a variety of other reasons and it is not a part of the standard C library.

```
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <time.h>
13 #include <conio.h>
14 #include <windows.h>
15
```

The ***#define*** directive serves as a preprocessor allowing us to make very specific declarations regarding data types that we use throughout the programming process when defining constants. The BOARD_WIDTH * 2 + 14 is adjusted to include space for "Next Piece" and BOARD_HEIGHT + 4 allows room for boarders and spacing. Defining BOARD_WIDTH 10 on **line 18** defines the width of the game board in cells, and BOARD_HEIGHT defines the height of the game board in cells. Beginning on **line 20** I define PIECE_SIZE at 4 because each Tetromino piece is made up of 4 smaller square shapes so we have to consider these pieces on a 4x4 grid to account for movement, rotation and angularity of each piece.

```
16 #define DISPLAY_WIDTH (BOARD_WIDTH * 2 + 14)
17 #define DISPLAY_HEIGHT (BOARD_HEIGHT + 4)
18 #define BOARD_WIDTH 10
19 #define BOARD_HEIGHT 20
20 #define PIECE_SIZE 4
```

Next, on **line 22** I define integer data types which will store data for the variables resulting in the next piece being used in loop sequences and then pass them into an array for the next one to spawn. **Line 23** will be used to store data for the color of the next piece.

```
22 int nextPiece[PIECE_SIZE][PIECE_SIZE];
23 int nextColor;
```

Next, on **line 25** I need to declare an integer type to hold all seven of the tetromino pieces in an array and this method represents them as 4x4 matrices.

```
25 int tetrominoes[7][PIECE_SIZE][PIECE_SIZE] = {
26     {
27         {1, 1, 1, 1},
28         {0, 0, 0, 0},
29         {0, 0, 0, 0},
30         {0, 0, 0, 0}
31     },
32     {
33         {1, 1},
34         {1, 1}
35     },
36     {
37         {0, 1, 0},
38         {1, 1, 1},
39         {0, 0, 0}
40     },
```

I

O

T

As I mentioned earlier, this 4x4 matrix allows the program to have proper movement, rotation, and angularity for each individual shape. Since each piece is made up of 4 smaller pieces, we have to consider how they rotate at 90 degrees in the real game and ensure that behavior matches as close as possible in this version.

Continuing on **line 41**, I'm creating the remaining shapes to store into the tetrominoes integer type. Once all of these are properly stored, the function can be closed.

```
41      {
42          {1, 1, 0},
43          {0, 1, 1},    =   Z
44          {0, 0, 0}
45      },
46      {
47          {0, 1, 1},
48          {1, 1, 0},    =   S
49          {0, 0, 0}
50      },
51      {
52          {1, 0, 0},
53          {1, 1, 1},    =   J
54          {0, 0, 0}
55      },
56      {
57          {0, 0, 1},
58          {1, 1, 1},    =   L
59          {0, 0, 0}
60      }
61  };
```

Now, I need to make more variable declarations. Beginning on **line 63** I am declaring all seven colors for each tetromino piece and storing them into an integer named ***colors***. On **line 64** I have to make sure in the beginning that the board is initialized to empty, and the current piece will be the active tetromino piece that is currently dropping on the player board followed by the color of the current tetromino. **Line 67** will hold the coordinates of the top-left corner of the current tetromino. The level will also need to be set to 1 in the beginning so this is declared as the starting point on **line 68** and **line 69.** This is exactly what you would expect as a data type to hold the lines cleared throughout each stage, and in the beginning we set it to 0. **Line 70** sets the initial drop speed to 800 milliseconds which we know is .8 seconds, and the next line begins the score at 0. The reason this is measured in milliseconds, is that it allows the game to have a much more precise way to control the timing between block movements, ensuring consistent and scalable gameplay as the game progresses. This is also done as the universal standard in programming as well as game engines.

```
63 int colors[7] = {1, 2, 3, 4, 5, 6, 7};
64 int board[BOARD_HEIGHT][BOARD_WIDTH] = {0};
65 int currentPiece[PIECE_SIZE][PIECE_SIZE];
66 int currentColor;
67 int currentX, currentY;
68 int level = 1;
69 int linesCleared = 0;
70 int dropSpeed = 800;
71 int score = 0;
```

This is where I write my first void function in this program. Void functions allow the user to perform actions that don't require returning a value of true or false (1 or 0). This is usually for the purpose of *'side effect'* actions such as printing to the console, modifying global variables, or updating data structures. In the instance of this program, on **line 73** I'm writing a micro function called *clearScreen*. Clearing the screen allows the programmer to update what the user sees without overlapping previous output. In other words, if we do not clear the screen, new information would merely stack on top of or next to the old content resulting in an unreadable display.

```
73 void clearScreen() {
74     system("cls");
75 }
76
77 void setColor(int color) {
78     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), color);
79 }
```

**Line 77** allows us to set the color of the text in the console, and without it, the program would not be able to make a connection between conditional checks for currentX and currentY coordinates and would fail to compile.

```
81 void spawnPiece() {
82     for (int y = 0; y < PIECE_SIZE; y++) {
83         for (int x = 0; x < PIECE_SIZE; x++) {
84             currentPiece[y][x] = nextPiece[y][x];
85         }
86     }
87     currentColor = nextColor;
88
89     //Generate a new next piece.
90     int type = rand() % 7;
91     for (int y = 0; y < PIECE_SIZE; y++) {
92         for (int x = 0; x < PIECE_SIZE; x++) {
93             nextPiece[y][x] = tetrominoes[type][y][x];
94         }
95     }
96     nextColor = colors[type];
97
98     currentX = BOARD_WIDTH / 2 - 2;
99     currentY = 0;
100
101    if (checkCollision(currentX, currentY)) {
102        gameOver();
103    }
104 }
```

The function on **line 81** spawns a random Tetromino at the top of the board as well as handle the next piece. The first for-loop on **line 82** copies the next piece into the current piece, and then we assign the color of the next piece to the current piece on **line 87** followed by generating the next piece at random between the seven pieces. Then, on **line 98** the BOARD_WIDTH operator centers the piece horizontally.

Next, on **line 101** we check if a game-over scenario is triggered in the event the piece collides immediately. If so, we print out a game-over message to the screen.

This function displays the next piece preview in the top-right corner of the player board area. Offset allows a push against the BOARD_WIDTH operator and offsetting Y positions a few lines below the top of the board. This is one of the cleaner methods of doing so in C programming.

```
107 void drawNextPiece() {
108     int offsetX = BOARD_WIDTH * 2 + 4;
109     int offsetY = 2;
110
111     for (int i = 0; i < offsetY; i++) {
112         printf("\n");
113     }
114     for (int i = 0; i < offsetX; i++) {
115         printf(" ");
116     }
117     printf("Next Piece:\n");
118
119     for (int y = 0; y < PIECE_SIZE; y++) {
120         for (int i = 0; i < offsetX; i++) {
121             printf(" ");
122         }
123         for (int x = 0; x < PIECE_SIZE; x++) {
124             if (nextPiece[y][x]) {
125                 setColor(nextColor);
126                 printf("[]");
127             } else {
128                 printf("  ");
129             }
130         }
131         printf("\n");
132     }
133     setColor(7);
134 }
```

The for-loop on **line 114** allows us to maintain the horizontal offset for each row. Otherwise, without this function we would experience problems from our *clearScreen* function from above.

This if-else statement on **line 124** allows us to draw the filled cell for the next piece and draw an empty space. We need this in order to differentiate between pieces that are drawn, as well as empty space. This function is essential for drawing the game board with the updated display area and score.

The if-statement on **line 154** is a conditional statement that checks if the current piece overlaps the cell. If we do not check for this, the program would crash every time the piece was spawned.

```
140 void drawBoard() {
141     clearScreen();
142     printf(" Tetris-C                    Level: %d\n", level);
143     //Top border
144     for (int i = 0; i < DISPLAY_WIDTH; i++) {
145         printf("*");
146     }
147     printf("\n");
148
149     for (int y = 0; y < BOARD_HEIGHT; y++) {
150         printf("* "); //Left border
151         for (int x = 0; x < BOARD_WIDTH; x++) {
152             int cell = board[y][x];
153
154             if (y >= currentY && y < currentY + PIECE_SIZE && x >= currentX && x < currentX + PIECE_SIZE) {
155                 int pieceY = y - currentY;
156                 int pieceX = x - currentX;
157                 if (currentPiece[pieceY][pieceX]) {
158                     cell = currentColor;
159                 }
160             }
```

Beginning on **line 162** this is the continuation of the conditional check from above. I've already been printing off the top and left borders of the player board.

```
161
162                 if (cell) {
163                     setColor(cell);
164                     printf("[]");
165                 } else {
166                     printf("  ");
167                 }
168             }
169             printf(" *     ");
170             if (y == 1) {
171                 printf("Next Piece:");
172             } else if (y >= 3 && y < 3 + PIECE_SIZE) {
173                 int nextY = y - 3;
174                 for (int x = 0; x < PIECE_SIZE; x++) {
175                     if (nextPiece[nextY][x]) {
176                         setColor(nextColor);
177                         printf("[]");
178                     } else {
179                         printf("  ");
180                     }
```

Beginning on **line 169** I'm printing spaces for alignment, the "Next Piece" and score area on the other side of the right border.

Setting color to *nextColor* on **line 176** is providing the program the ability to use the next pieces color, otherwise without making that function call we would simply repeat the same color. If this is false, and the color is passed the if-else condition prints the bracket for the tetromino piece and then the empty cell for the next piece preview. Continuing on, the else-if statement on **line 182** displays the score row underneath the "Next Piece" row, and then prints a new line to add a space.

```
181                 }
182             } else if (y == 8) { //Score display row
183                 printf("Score: %d", score);
184             }
185             printf("\n");
186         }
187
188         //Bottom border
189         for (int i = 0; i < DISPLAY_WIDTH; i++) {
190             printf("*");
191         }
192         printf("\n");
193
194         setColor(7);
195 }
```

Starting on **line 189**, this for-loop iterates through and prints out the bottom border of the play area, prints a new line, and then resets the color to default. All of these borders are more than just a visual aspect of the program. We cannot draw the border without knowing the height and the width, and we also can't write the logic for collision detection later without checking for the same parameters of BOARD_WIDTH & HEIGHT.

Now that the score area is displayed, we have to apply functionality and make sure it updates when the pieces drop. In the regular game, the player receives points every time the pieces hit even if they do not fit together or clear lines, and this program will be no different. The first thing to consider is how the score updates based on a few factors:

1) The current level
2) How many lines were cleared
3) What the current drop speed is

If the player removes a single line on level 1 then its 40 extra points. If the player removes two lines, then a score that is greater than or equal to 100 multiplied by the current level will be awarded. If three lines are cleared at once, then the player receives a score greater than or equal to 300 multiplied by the current level. If the player removes all four lines, then this awards a Tetris which is a score of greater than or equal to 1200 multiplied by the current level. This is accomplished using a switch statement.

```
198 void updateScore(int linesClearedInOneMove) {
199     switch (linesClearedInOneMove) {
200         case 1:
201             score += 40 * level;
202             break;
203         case 2:
204             score += 100 * level;
205             break;
206         case 3:
207             score += 300 * level;
208             break;
209         case 4:
210             score += 1200 * level;
211             break;
212         default:
213             break;
214     }
215 }
```

Even with the switch statement in place for updating the score this logic has to be combined with the understanding of how lines are removed. This next function proved to be one of the most complicated portions of writing this program, not because of the base function and removing lines themselves, but trying to customize it so that multiple lines would animate and disappear in a very specific way to the regular game.

However, there were continual problems with my code getting stuck in the loop of this *removeFullLines* function and not properly exiting the loop causing the program to freeze once the animation was triggered. After several hours of debugging with unsuccessful results, I ended up accepting that the lines would simply disappear rather than create an animation for all of them at once. Sometimes a programmer must either accept defeat, or head back to the drawing board to rework a new algorithm and rewrite everything from scratch.

This version of my *removeFullLines* function is a fairly straight forward implementation and ultimately the function that I ended up going with. It will remove multiple lines at once, but not with any fancy animated behavior. It simply removes the proper lines from the player board.

```
217 void removeFullLines() {
218     int linesClearedInThisMove = 0;
219
220     for (int y = 0; y < BOARD_HEIGHT; y++) {
221         int full = 1;
222         for (int x = 0; x < BOARD_WIDTH; x++) {
223             if (!board[y][x]) {
224                 full = 0;
225                 break;
226             }
227         }
228         if (full) {
229             linesCleared++;
230             linesClearedInThisMove++;
231
232             for (int yy = y; yy > 0; yy--) {
233                 for (int x = 0; x < BOARD_WIDTH; x++) {
234                     board[yy][x] = board[yy - 1][x];
235                 }
236             }
237             for (int x = 0; x < BOARD_WIDTH; x++) {
238                 board[0][x] = 0;
239             }
240         }
241     }
242     updateScore(linesClearedInThisMove);
243
244     if (linesCleared >= level * 5) {
245         level++;
246         dropSpeed -= 50;
247         if (dropSpeed < 100) {
248             dropSpeed = 100;
249         }
250     }
251 }
```

In another version of my Tetris game, the lines do blink just as they do in the Game Boy version before disappearing, but only clearing one line at a time, rather than blinking all at once before shifting them down. To me this was unacceptable, and I decided to go with something less complicated that would still function properly without crashing the program.

This next function is also very straight forward. Just like the regular version of the game, I want to award the player with points even when a piece is placed down, so we do this by writing a function that allows us to give them 5 points even when a piece is placed down each time.

```
256 void placePiece() {
257     for (int y = 0; y < PIECE_SIZE; y++) {
258         for (int x = 0; x < PIECE_SIZE; x++) {
259             if (currentPiece[y][x]) {
260                 board[currentY + y][currentX + x] = currentColor;
261             }
262         }
263     }
264     score += 5;
265 }
```

At this stage, I can now implement the logic for my collision detection. In this function, I declare the data type and give it an obvious name like *checkCollision* and then pass the parameters of newX and newY. Since we don't know yet what those will be during gameplay, we have to check for a possible collision and assign the values accordingly. At **line 270**, I'm checking if the cell is a part of the piece, and then at **line 274** I'm checking for the boundaries of the board. By returning a value of true or 1, then there is a collision with the border.

```
267 int checkCollision(int newX, int newY) {
268     for (int y = 0; y < PIECE_SIZE; y++) {
269         for (int x = 0; x < PIECE_SIZE; x++) {
270             if (currentPiece[y][x]) {
271                 int boardX = newX + x;
272                 int boardY = newY + y;
273
274                 if (boardX < 0 || boardX >= BOARD_WIDTH || boardY >= BOARD_HEIGHT) {
275                     return 1;
276                 }
277
278                 if (boardY >= 0 && board[boardY][boardX]) {
279                     return 1;
280                 }
281             }
282         }
283     }
284     return 0;
285 }
```

On **line 278** I'm checking for collision with existing blocks on the board, and if we return a value of true, then the collision is placed by the blocks. Otherwise, by returning zero or false, then there is no collision.

At this point, I've checked for collision with borders and blocks, but I haven't checked collision with rotation, nor have we implemented the ability to rotate the pieces 90 degrees. This function allows the player to rotate the active piece 90 degrees clockwise.

```
287 void rotatePiece() {
288     int temp[PIECE_SIZE][PIECE_SIZE] = {0};
289     for (int y = 0; y < PIECE_SIZE; y++) {
290         for (int x = 0; x < PIECE_SIZE; x++) {
291             temp[x][PIECE_SIZE - 1 - y] = currentPiece[y][x];
292         }
293     }
294
295     if (!checkCollision(currentX, currentY)) {
296         for (int y = 0; y < PIECE_SIZE; y++) {
297             for (int x = 0; x < PIECE_SIZE; x++) {
298                 currentPiece[y][x] = temp[y][x];
299             }
300         }
301         while (currentX < 0) currentX++;
302         while (currentX + PIECE_SIZE > BOARD_WIDTH) currentX--;
303     }
304 }
```

Within the function, we have to check if rotation causes a collision. This conditional is checked on **line 295**.

There also has to be a way to handle boundary violations after rotation. Remember, we are considering this behavior on a 4x4 matrix. This can be done by writing some while-loops. On **line 301** we shift right if the piece is out on the left, and on **line 302** we shift left if the piece is out on the right using incremental and decremental looping. During game loop executions, most games need to continuously update the screen, check for input, and update the current game state. While-loops ensure the game keeps running until the player exits.

At this point, we have to update the game status by moving the current piece or spawning a new one. This function checks for an attempt to move the current piece down by one cell, and move the piece down if no collision is detected. Otherwise, if the piece cannot move further down, we place the piece on the game board, check for and remove any completed lines, and then spawn a new piece to continue the game.

```
306 void update() {
307     if (!checkCollision(currentX, currentY + 1)) {
308         currentY++;
309     } else {
310         placePiece();
311         removeFullLines();
312         spawnPiece();
313     }
314 }
```

At this point, I have a plethora of logical functions written for the game, but at the current state I have no means of handling the input made by the user. This is where another void function comes into play. There must be a function for handling movement, and rotation of the current Tetronimo piece. In game development, the most common keyboard input is *W, A, S, D* so I will be following the same trend in my game as well. This function reads keyboard input, checks for valid moves, and updates the piece position or orientation accordingly. **Line 317** checks if a key has been pressed, and then gets the key value. The switch statement is the most common method for performing key input.

```
316 void handleInput() {
317     if (_kbhit()) {
318         char key = _getch();
319         switch (key) {
320             case 'a':
321                 if (!checkCollision(currentX - 1, currentY)) {
322                     currentX--;
323                 }
324                 break;
325             case 'd':
326                 if (!checkCollision(currentX + 1, currentY)) {
327                     currentX++;
328                 }
329                 break;
330             case 's':
331                 update();
332                 break;
333             case 'w':
334                 rotatePiece();
335                 break;
336         }
337     }
338 }
```

**Line 322** moves the piece one cell to the left if there is no collision, and **line 327** allows for movement to the right.

Calling the update function on **line 331** will perform a soft drop by moving the piece downward when the 'S' key is hit.

The 'W' key will allow us to rotate the active piece clockwise as it descends.

In the event that the user wishes to play the game again, we need to make sure there is a function in place to handle a game reset. This function will tell the program how to reset the game state for a new game, and then clear the board, reset the variables, and then spawn the first piece for the new game.

```
340 void resetGame() {
341     for (int y = 0; y < BOARD_HEIGHT; y++) {
342         for (int x = 0; x < BOARD_WIDTH; x++) {
343             board[y][x] = 0;
344         }
345     }
346     score = 0;
347     level = 1;
348     linesCleared = 0;
349     dropSpeed = 800;
350     spawnPiece();
351 }
```

Once the player has concluded gameplay, there needs to be a function to handle the game-over scenario and prompt the player to restart or exit the program.

```
353 void gameOver() {
354     clearScreen();
355     printf("Game Over!\n");
356     printf("Final Score: %d\n", score);
357     printf("Would you like to play again? (Y/N): ");
358
359     char choice;
360     while (1) {
361         choice = _getch();
362         if (choice == 'Y' || choice == 'y') {
363             resetGame();
364             return;
365         } else if (choice == 'N' || choice == 'n') {
366             printf("\nThank you for playing!\n");
367             exit(0);
368         }
369     }
370 }
```

Beginning on **line 359**, the character choice is placed into a while-loop and waits for valid input. If the choice is 'Y' for yes, the game restarts, and the *gameOver* function returns to the game loop.

If the choice is 'N' for no, the game thanks you for playing and exits the program.

To finalize the program, the *Main* function allows us to initialize everything. **Line 376** seeds the random number generator with the current time, and then **line 378** generates the first *"next piece"*. After that, I assign a color to the first next piece and then spawn the first piece to start the game.

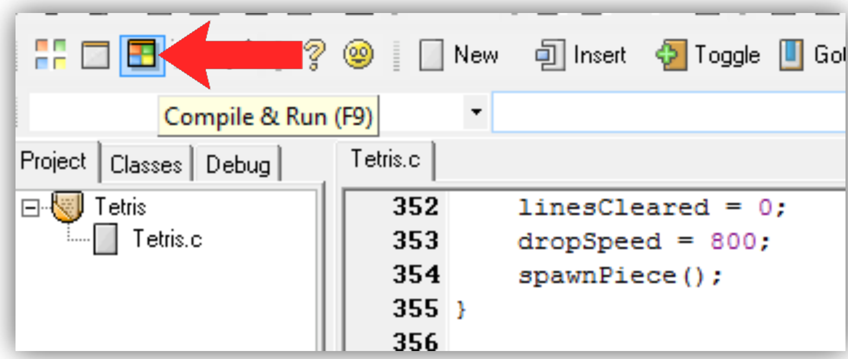
```
375 int main() {
376     srand(time(NULL));
377
378     int type = rand() % 7;
379     for (int y = 0; y < PIECE_SIZE; y++) {
380         for (int x = 0; x < PIECE_SIZE; x++) {
381             nextPiece[y][x] = tetrominoes[type][y][x];
382         }
383     }
384     nextColor = colors[type];
385     spawnPiece();
386
387     unsigned long lastUpdate = GetTickCount();
388     while (1) {
389         drawBoard();
390         handleInput();
391
392         unsigned long now = GetTickCount();
393         if (now - lastUpdate >= dropSpeed) {
394             update();
395             lastUpdate = now;
396         }
397         Sleep(50);
398     }
399     return 0;
400 }
```

Beginning on **line 387**, I use an unsigned long as a data type to represent a non-negative integer with a larger than regular unsigned integer. The unsigned property only stores positive numbers (0 and above). This allows me to track the time for game updates.

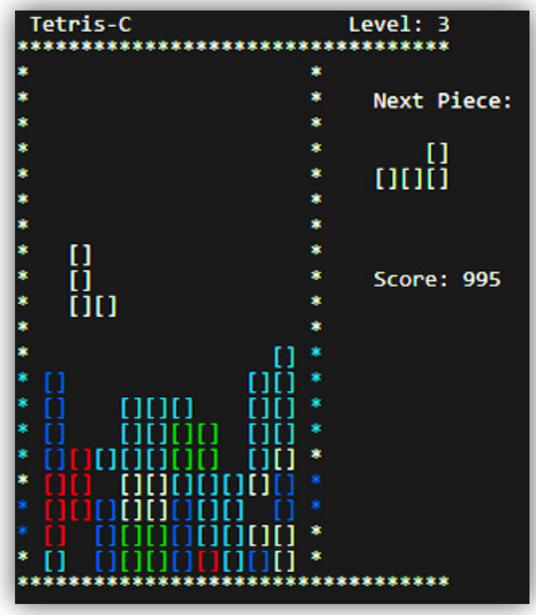
The while-loop on **line 388** enters the main game loop, renders the game board to the console, and processes player input to move or rotate the current piece.

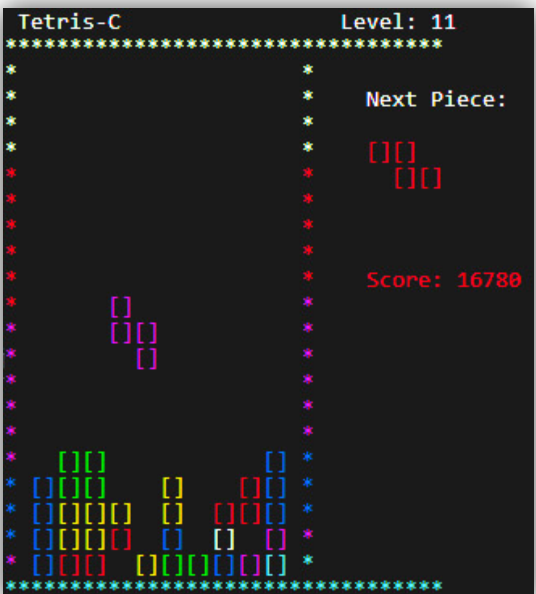
The next function on **line 392** will update the game state by either moving the piece or spawning a new one, and then reset the timer for the next update. The Sleep function on **line 397** will introduce a small delay to make the loop less CPU-intensive. This is a common practice in console applications in early development. *Return 0* on **line 399** will never be reached because the game is running in an infinite loop.

As a programmer, you obviously test your syntax by compiling the code throughout the development process, but with this program completely written, it can be compiled as-is. Simply click the following icon or hit **F9** to compile and run the program into an executable file.




That's it, and this concludes my first Tetris game!



# Conclusion

After extensive testing of running the game trying to crash it, I can confirm that this particular implementation is fully functional and behaves as the user would expect it to. That being said, while it does work, it is by no means perfect and does not behave to the extent *(desired)* that I tried to build it. I still wanted to see more out of it, but regarding the complications of my *removeFullLines* function, as long as a program works, and does the thing, then sometimes this can be enough. As my first implementation of Tetris, from formulating a plan of action to implementing all of the code, this proved to be a lot and I knew it would be for me. For a natural born programmer, an application like this can be made in a day or two. For me, it took weeks of planning, implementation and repeated testing. As I've mentioned in my *Intro to Game Logic* document, game logic is among the most complicated logic to be implemented into a program, and if you can master game logic, there is little that you can't create in this world. I hope this guide is useful to any and all for educational purposes and at nothing else, provides insight into what all goes into creating the base-level of a Tetris game. If you have any questions about this guide or any other general inquiries, you can email me at technologicguy@gmail.com

---

**Resources Used:**

- Cplusplus.com
- Tetris Wiki