Semester Project

# Tetris

## Game info:

You must have played famous game named Tetris. It basically drops different shaped blocks from the ceiling of the screen and players can move left, right or rotate the block before it reaches the floor or deck onto other block. If at that time, certain rows are filled up, these rows will be erased.

## Background skills

This game will involve three programming skills, which you don’t need to grasp within this course. You are not required to code anything using these skills, but it will help you a lot in understanding the game design if you know what they are.

### Escape Sequence

As we know ‘\n’ is to make a new line, which is to move the cursor to the next row of the console window. In fact, we can move the cursor to any position of our console window using “escape sequence” if the console supports.

For example,

“**printf( "\033[2;1H");**” will move your cursor to the beginning of line 2. Here “\033” is the ASCII of “ESC” key in Oct mode.

“**printf( "\033[K");** “ will clear the content of the whole line where your cursor stays.

Using this technology, we could refresh same piece of console window periodically, such as 1.5 seconds, giving you the illusion that the words are elevating.

But windows 7 console window does not support that. We need to install a widget to enable it. The widget is called “**ansicon**”. To install it, you need to open your command window by typing “**cmd**” in your windows **start menu**. In command window, cd to “**ansi160\x64**” if your computer is 64 bits, or to “**ansi160\x86**” if your computer 32 bit, run “**ansicon -i**”. Then, this command window will supports ANSI mode.

### Thread

When you are using scanf, you program will suspend there waiting your input from stdin(keyboard. But we do require the block keeps dropping. So, in addition to our main program, which is also a thread (we just name it main thread), we create another thread named UI thread via **CreateThread**. The main thread will periodically refresh screen, in which blocks will keep dropping. The UI thread waits for user’s input. All these code is within **projmain.c**. You do not need to change anything there.

Since two threads are running at the same time, both of them can access our word list. We need to make sure when one of the threads is processing, the other will be waiting on there. This is why **EnterCriticalSection**, **LeaveCriticalSection** are used. These are already there in main, which you don’t have to change.

### Random Generating

**rand()** is to generate a random integer within the range you can control. This part of code is within .**lib**. You don’t have to worry about that either. This gives you a chance to learn how to link a non-system lib.

## Game Design

As mentioned early, the main thread will refresh our screen periodically. Our screen has the following grid layout. After printing fixed game title “Tetris” and ceiling, the main thread will refresh the blue **game area**. The content of the area will keep change as the game is running. We can use certain data structure to store the contents. The most intuitive way is using a 2D array like “int gameArea[ROWS][COLS];” However, this design is not so efficient when it comes to delete the whole row once filled up. Let’s say your row 20 has been filled up and you need to delete it, under this design, it has to copy everything from row 0 to 19 to row 1 to 20. Large memory copy is always being frowned upon.

So, we use link list to store the game area contents. Each node in the link is a struct designed like this

struct row{

char content[COLS + 1];

struct row \*next;

};

struct row \*head, \*tail;

Basically, row 0 is supposed to be the head of the link list, its next points to row one, and again row 1’s next point to row 2, till it goes to the last row. The contents[COLS] stores each column’s empty/occupied information. If it is 0, you printf space in the corresponding column, else you printf “\*”.

|  |
| --- |
| pCursor[0] |
| pCursor[2] |
| pCursor[3] |
| pCursor[4] |
| pCursor[5] |
| pCursor[6] |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tetris | | | | | | | | | |
| =================== | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| \* | \* |  |  |  |  |  |  |  | \* |
| \* | \* | \* | \* |  |  |  |  |  | \* |
| \* | \* | \* | \* | \* |  | \* |  | \* | \* |
| Total Score: | | | | | | | | | |

Screen layout

However, link list is not so easy to access any grid as 2D array. A lot of grid access is required when you calculate block moving, rotating. So we may define an array of pointers pointing to struct row. “struct row \*pCursor[6]”. Since maximum rows a block can span is 4 (bar in vertical position), plus the row above and below. So there are at most 6 rows of grids might need to access when calculating moving and rotating. Every time the block drops, update the pCursor also.

Each block has different shape; also same block can rotate. So we need some data structure to store all these information to make the code easily. There are various ways to do that. Here gives an example data structure looks like, you can welcome to use your own data structure.

Since a 4x4 matrix is enough to fit all kind of block. So for the T-Shape block below, using coordinate (row,column), only(2,0), (2,1), (2,2),(3,1) are filled. So we can have an declare an array to store that.

int ShapeT[16] = {2,0,2,1,2,2,3,1,-1); The reason to choose 16, is that it large enough to accommodate all shapes. If you need a more complex shape, for example 3\*3 square, you need more space. -1 marks the end of the array. However, we know when the block rotates, the coordinate will change. There are at most 4 rotate positions, so we need 4x16’s 2D array to store the shape information.

int ShapeT[4][16] = {

{2,0, 2,1, 2,2, 3,1, -1},

{1,0, 2,0, 2,1, 3,0, -1},

{2,1, 3,0, 3,1, 3,2, -1},

{1,1, 2,0, 2,1, 3,1, -1},

};

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |  | 0 | 1 | 2 | 3 |  | 0 | 1 | 2 | 3 |  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  | 0 |  |  |  |  | 0 |  |  |  |  | 0 |  |  |  |  |
| 1 |  |  |  |  | 1 | \* |  |  |  | 1 |  |  |  |  | 1 |  | \* |  |  |
| 2 | \* | \* | \* |  | 2 | \* | \* |  |  | 2 |  | \* |  |  | 2 | \* | \* |  |  |
| 3 |  | \* |  |  | 3 | \* |  |  |  | 3 | \* | \* | \* |  | 3 |  | \* |  |  |

You can adjust the data a little bit if you don’t want it be so “left-bottom” cornered.

When the block is moving left, right, down or rotate, you should check whether the block can move. If the any grid you are to move in has been occupied, you can’t move. To make code easier, we also need to store information about what grid to check for different shapes’ different rotates. The data structure is similar to shape data structure.

For example, we has the following data structure to check whether a bar block can move left. When a bar in vertical position, it needs to check the grids on the left of (0,0), (1,1), (2,0), (3,0), if it in horizontal position, we only need to check the grid on the left of (3,0).

int ShapeBarLeftDetectLUT[4][16] = {

{3,0, -1},

{0,0, 1,0, 2,0, 3,0, -1},

{3,0, -1},

{0,0, 1,0, 2,0, 3,0, -1},

};

All these lookup tables will be given. You can also design your own lookup table. If you want to add more shape, you are supposed to add new shapes’ lookup table. It is a good design if no more code to change except that.

The block is generated randomly. We need a struct to describe the block.

enum SHAPE

{

SHAPE\_SQUARE = 0,

SHAPE\_T,

SHAPE\_BAR,

SHAPE\_MAX

};

struct block {

int columnNo;

int rowNo;

int rotate;

enum SHAPE shape;

};

First, each block has its position. The member columnNo, rowNo gives the block’s position. However it is up to you to interpret the exact position based on your pivot selection. For example, if the previous screen layout demo, if you select top-left corner of 4x4 matrix as your pivot, the bar is at the position

For example, the bar in the above “screen layout” example has the position row 2, column 3. If you select top-right corner as your pivot, the bar is at row 2, column 6.

The rotate is the rotate angle, it is between 0 and 3.

Shape is defined as enum to make the code easy to read. Currently we supports three shapes.

## What you need to do?

## Writing all the functions below to support this game:

### void Init()

In this function, you need create a link list of rows. Also you may have to initialize your cache[6]. cache[0] points to the header, which is row 0, cache[1] pointes to row 1 and so forth.

### void freshScreen ()

This function is being called periodically. Basically, it just refreshes the game area based on the content of your link list of rows.

### void fillShape(struct block \*myBlock, int bFill)

This function will be used a lot when block is moving down, left, right, rotate.

### int processBlockDown(struct block \*myBlock)

This function is called when the user press “down” key, or periodically by the main thread.

1. It tries to move the block one row down. If the block can’t move down, it returns 1, or it returns 0.
2. If the block can’t move down, call checkRowsDelete function to delete filled up rows.
3. If the block can move, call fillShape with last parameter being 0 to empty the grids the block occupies, then increase the block’s rowNo by one, and call fillShape again with last parameter being 1 to occupy the new grids.
4. Update the pCache[].

### struct block \*processBlock (struct block \* pBlock)

This function is also called periodically. It does the following:

1. Call the processBlockDown function, which let the block drops one row each time.
2. If the processBlockDown returns 0, which means the pBlock can not drop down. You need free this pBlock. And call genNewBlock to generate new block, and return this block.
3. If the processBlockDown returns 1, it just return the original block.

### void moveLeft(struct block \*block)

This function is to move the block to left.

1. It checks whether the block can move block left.
2. If it can, first call fillShape with last parameter being 0 to empty the grids the block occupies, then decrease the block’s columnNo by one, and call fillShape again with last parameter being 1 to occupy the new grids.

### void moveRight(struct block \*block)

Same as moveLeft, except it moves the block to right.

### void moveDown(struct block \*\*block)

This function is little bit like processBlock, but it is called when user press “down” key five times in row.

1. It calls processBlockDown repeatedly until it returns 0.
2. Free the old block and generate new block.

Notice the function prototype is different from processBlock, to practice using double pointer.

### void rotate(struct block \*\*block)

Same as moveLeft, except it rotates the block.

### void display\_Statistics()

It is called after refreshScreen, you are suppose to print out the total score in the last row.

### void checkRowsDelete()

It is called after we found out the block can’t move down anymore.

1. You are suppose to check whether there are rows need to be deleted. If there are rows to be deleted, using the way you did in delete\_ith\_node to remove the rows.
2. After remove the rows, you need create new rows and append it at the head.
3. Update the total score.
4. Call appExit to terminate the application it the block touches the ceiling

### void appExit()

It is called before the game is over.

1. The highest history record is saved in a local file. If the totalScore is higher than history record, update the file with new total Score.
2. Free all the memory being used
3. Exit the game.

|  |  |
| --- | --- |
| Function | Score |
| Init | 10 |
| refreshScreen | 10 |
| processBlock | 10 |
| processBlockDown | 10 |
| moveDown | 5 |
| moveLeft | 5 |
| moveRight | 5 |
| Rotate | 5 |
| displayStatistics | 5 |
| fillShape | 10 |
| checkRowsDelete | 15 |
| appExit | 10 |
| SubTotal | 100 |
| Bonus: |  |
| Add two shapes | 15 |
| Fancy features | 15 |
| Total | 130 |
|  |  |