Implementing Tetris using Assembly language

A Project Report

BY

BOPPANA SAI SUCHEET(18BCE2322)

GOKUL RAJ (18BCE2308)

SHUBHAM MAITY (18BCE2304)

FACULTY: PROF. ANTHONY XAVIER GLITTAS X.

MICROPROCESSOR AND INTERFACING

CSE2006

SLOT: A1

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CERTIFICATE

This is to certify that the project work entitled ***"Implementing Tetris using Assembly language"*** that is being submitted by ***"Sucheet Boppana(18BCE2322),Gokul Raj(18BCE2308) and Shubham Maity (18BCE2304)"*** for Microprocessors and Interfacing (CSE2006).

ACKNOWLEDGEMENTS

We take immense pleasure in thanking **Dr. G. Viswanathan**, our beloved Chancellor, VIT University and respected Dean, **Dr. R. Saravanan**, for having permitted us to carry out the project.

We express gratitude to our guide**, PROF. ANTHONY XAVIER GLITTAS X.** , for guidance and suggestions that helped us to complete the project on time. Words are inadequate to express our gratitude to the faculty and staff members who encouraged and supported us during the project. Finally, we would like to thank our ever-loving parents for their blessings and our friends for their timely help and support.

Introduction

What is Tetris? It was invented in Russia and the main aim of the game is to bring down blocks from the top of the screen. The blocks fall at a certain rate, but you can make them fall faster manually and you can move the blocks around as well. Your objective is to get all the blocks to fill all the empty space in a line at the bottom of the screen and as you do that, the blocks will vanish and hence reward you points.

On paper it never looked like an idea that will be so popular beyond the year 1975 but due to its goals and rules, participating will become voluntary and there are unnecessary obstacles that keep the game challenging.

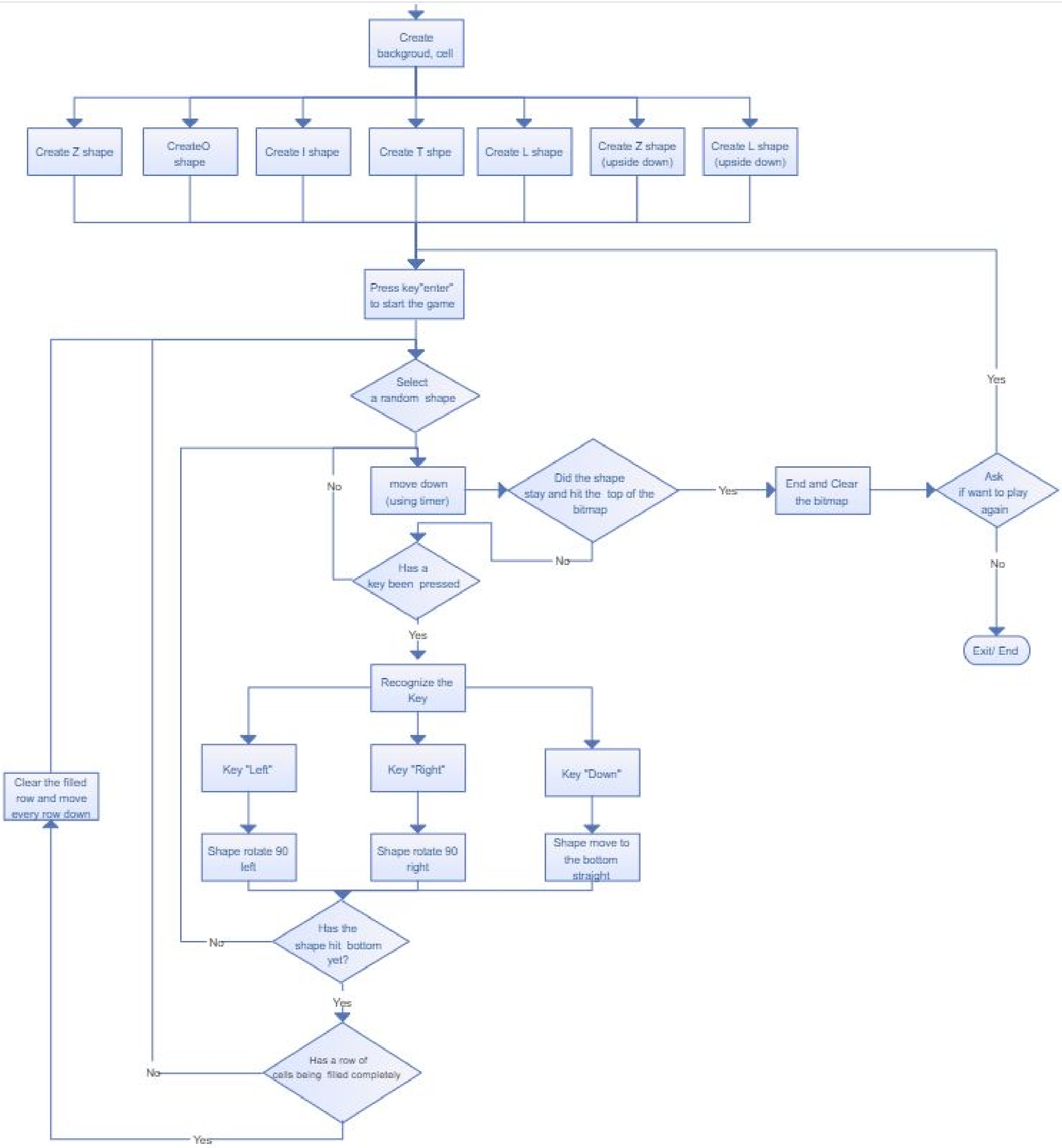
Proposed System Design

We are using Assembly language to implement the game and we are using Computer x86. We have tested the game on DOSBOX and NASM.

DOSBOX is an emulator program which is used to run old games and even used to run other MS-DOS applications. It's a free, open-source cross platform that uses the SDL library. Many IBM PC compatible graphics and sound cards are also emulated. DOSBOX is the standard way to play DOS games on modern computers.

The code is written for Computer x86 architecture and tested on DOSBOX and NASM

FLOW CHART


Create backgroud, cell
Create Z shape
CreateO shape
Create I shape
Create T shpe
Create L shape
Create Z shape (upside down)
Create L shape (upside down)
Press key"enter" to start the game
Select a random shape
move down (using timer)
Did the shape stay and hit the top of the bitmap
Yes
End and Clear the bitmap
Ask if want to play again
Yes
No
Exit/ End
No
No
Has a key been pressed
Yes
Recognize the Key
Key "Left"
Key "Right"
Key "Down"
Shape rotate 90 left
Shape rotate 90 right
Shape move to the bottom straight
Clear the filled row and move every row down
No
Has the shape hit bottom yet?
Yes
No
Has a row of cells being filled completely
Yes

Assembly Code:

```
org 100h
jmp initialization
    msg_author db "Welcome to Tetris$"
    msg_next db "Next$"
    msg_left db "A - Left$"
    msg_right db "S - Right$"
    msg_rotate db "SPC - Rotate$"
    msg_quit db "Q - Quit$"
    msg_lines db "Lines$"
    msg_game_over db "Game Over$"
    msg_asmtris db "Tetris$"

    delay_centiseconds db 5 ; delay between frames in hundredths of a second
    screen_width dw 320

    block_size dw 5 ; block size in pixels
    blocks_per_piece dw 4 ; number of blocks in a piece
 colour_cemented_piece dw 40, 48, 54, 14, 42, 36, 34 ; colours for pieces
                                             ; which have cemented
 colour_falling_piece dw 39, 47, 55, 44, 6, 37, 33 ; colours for pieces
                                            ; which are falling
pieces_origin:
    piece_t dw 1605, 1610, 1615, 3210 ; point down
            dw 10, 1610, 1615, 3210   ; point right
            dw 10, 1605, 1610, 1615   ; point up
            dw 10, 1605, 1610, 3210   ; point left
    piece_j dw 1605, 1610, 1615, 3215 ; point down
            dw 10, 15, 1610, 3210     ; point right
            dw 5, 1605, 1610, 1615    ; point up
            dw 10, 1610, 3205, 3210   ; point left
    piece_l dw 1605, 1610, 1615, 3205 ; point down
            dw 10, 1610, 3210, 3215   ; point right
            dw 15, 1605, 1610, 1615   ; point up
            dw 5, 10, 1610, 3210      ; point left
    piece_z dw 1605, 1610, 3210, 3215 ; horizontal z
```

```
        dw 15, 1610, 1615, 3210   ; vertical z
        dw 1605, 1610, 3210, 3215 ; horizontal z
        dw 15, 1610, 1615, 3210   ; vertical z
piece_s dw 1610, 1615, 3205, 3210 ; horizontal s
        dw 10, 1610, 1615, 3215   ; vertical s
        dw 1610, 1615, 3205, 3210 ; horizontal s
        dw 10, 1610, 1615, 3215   ; vertical s
piece_square dw 1605, 1610, 3205, 3210 ; a square
            dw 1605, 1610, 3205, 3210 ; another square
            dw 1605, 1610, 3205, 3210 ; nothing but
            dw 1605, 1610, 3205, 3210 ; squares here
piece_line dw 1600, 1605, 1610, 1615 ; horizontal line
          dw 10, 1610, 3210, 4810   ; vertical line
          dw 1600, 1605, 1610, 1615 ; horizontal line
          dw 10, 1610, 3210, 4810   ; vertical line

msg_score_buffer db "000$" ; holds the string representation of score
score dw 0 ; keeps score (representing total number of cleared lines)

current_frame dw 0 ; our global frame counter

delay_stopping_point_centiseconds db 0 ; convenience variable used by the
                                       ; delay subroutine
delay_initial db 0 ; another convenience variable used by the
                   ; delay subroutine

random_number db 0 ; incremented by various events
                   ; such as input, clock polling, etc.

must_quit db 0 ; flag indicating that the player is quitting the game

cement_counter db 0 ; number of frames during which a piece which
                    ; can no longer fall is allowed to still be
                    ; controlled by the player
;;variables
player_input_pressed db 0 ; flag indicating the presence of input

current_piece_colour_index dw 0 ; index of current colour in colours array
```

```
    next_piece_colour_index dw 0 ; used to display next piece
    next_piece_orientation_index dw 0 ; used to display next piece

    piece_definition dw 0 ; pointer to first of the group
                          ; of four piece orientations for this piece
                          ; (see above for an explanation)
    piece_orientation_index dw 0 ; 0 through 3, index of current orientation
                                 ; among all of the piece's orientations
                                 ; (see above for an explanation)

    piece_blocks dw 0, 0, 0, 0  ; stores positions of blocks of current piece

    piece_position dw 0    ; position of the top left corner
                        ; of the falling 4 by 4 piece
    piece_position_delta dw 0 ; frame-by-frame change in current piece position

initialization:
     ; enter graphics mode 13h, 320x200 pixels 8bit colour
      mov ax, 13h
      int 10h
    ; set keyboard parameters to be most responsive
    mov ax, 0305h
    xor bx, bx
    int 16h
    ; generate initial piece
    call procedure_random_next_piece
    ; display controls, play area, borders, etc.
     call procedure_draw_screen
; Main program
new_piece:
    ; since we're generating a new block, a piece has just cemented, which
    ; means that there may be updates to the score due to lines potentially
    ; being cleared by that last piece
     call procedure_display_score
    ; start falling from the middle of the top of the play area
      mov word [piece_position], 14550
        ; next piece colour index becomes current
      mov ax, [next_piece_colour_index]
    mov word [current_piece_colour_index], ax
    ; colours array and pieces array have corresponding entries, so use colours
    ; index to set the piece index as well, but it has to be offset by as many
    ; bytes as each piece occupies
    shl ax, 5 ; ax := ax * 32 ( 16 words for each piece )
    add ax, pieces_origin ; offset from first piece
    mov [piece_definition], ax ; piece_definition now points to the first of
                              ; four piece orientations of a specific piece
    ; next piece becomes current
    mov ax, [next_piece_orientation_index]
    mov word [piece_orientation_index], ax ; choose one of the
                                          ; four orientations
    call procedure_copy_piece
        ; can this piece even spawn?
   call procedure_can_piece_be_placed
    test al, al              ; did we get a 0, meaning "can move"?
    jne game_over     ; no, can't move down - game is over!
    test al,al
    je s1
    game_over:
     ; draw game over overlay panel, and hide left/right/rotate controls
      call procedure_display_game_over
        ; since we've just made next piece current, we need to generate a new one
 s1:
    call procedure_random_next_piece


; Temporarily make next piece current so that it can be displayed in the
; "Next" piece area
display_next_piece:
    ; erase old next piece by drawing a black 4x4 block piece on top
    mov di, 17805
    mov bx, 20
    mov dl, 0
    call procedure_draw_square ; erase old "next" piece
    ; save current piece
    push word [current_piece_colour_index]
    push word [piece_definition]
    push word [piece_orientation_index]
    push word [piece_position]
     ; make next piece current - colour index
      mov ax, [next_piece_colour_index]
    mov word [current_piece_colour_index], ax ; save colour index
     ; make next piece current - piece definition
     shl ax, 5 ; ax := ax * 32 ( 16 words for each piece )
    add ax, pieces_origin ; offset from first piece
    mov [piece_definition], ax ; piece_definition now points to the first of
                            ; four piece orientations of a specific piece
    ; make next piece current -  piece orientation index
    mov ax, [next_piece_orientation_index]
    mov word [piece_orientation_index], ax ; choose one of the
                                     ; four orientations
    call procedure_copy_piece
      ; temporarily move current piece to the Next display area
    mov word [piece_position], 17805 ; move piece to where next
                                ; piece is displayed
        ; set colour in dl
    mov word bx, [current_piece_colour_index]
    shl bx, 1
    mov byte dl, [colour_falling_piece + bx]
    call procedure_draw_piece
     ; revert current piece to what is truly the current piece
     pop word [piece_position]
    pop word [piece_orientation_index]
    pop word [piece_definition]
    pop word [current_piece_colour_index]
    call procedure_copy_piece
main_loop:
    ; advance frame
    mov word ax, [current_frame]
    inc ax
    mov word [current_frame], ax
    call procedure_delay
     ; reset position delta and input state
    mov word [piece_position_delta], 0
    mov byte [player_input_pressed], 0
```

```
       ; animate logo
    call procedure_display_logo
read_input:
    ; read input, exiting game if the player chose to
     call procedure_read_character
    cmp byte [must_quit], 0
    jne x1
     ; [piece_position_delta] now contains modification from input
handle_horizontal_movement:
      ; if the player didn't press left or right, skip directly to where we
    ; handle vertical movement
    mov ax, [piece_position_delta]
    test ax, ax
    jz handle_vertical_movement ; we didn't press left or right
    ; either left or right was pressed, so shift piece horizontally
    ; according to how delta was set
    call procedure_apply_delta_and_draw_piece
handle_vertical_movement:
     ; for each of the blocks in the current piece
      mov cx, [blocks_per_piece] ; each piece has 4 blocks
handle_vertical_movement_loop:
    ; position di to the origin of current block
    mov di, [piece_position] ; start from the origin of the piece
    mov bx, cx ; wish I could use cx as an index register...
    shl bx, 1 ; bx := bx * 2, since each block index is a word
    sub bx, 2 ; our index is zero-based, while cx/loop are one-based
    add di, word [piece_blocks + bx] ; shift position in the piece
                                        ; to the position of current block
    ; if current block cannot move down, then
    ; the whole piece cannot move down
    call procedure_can_move_down
    test al, al ; a non-zero indicates an obstacle below
    jnz handle_vertical_movement_loop_failure
     ; check next block
    loop handle_vertical_movement_loop
     ; all blocks can move down means that the piece can move down
       jmp handle_vertical_movement_move_down_success
x1:
   jnz x2
handle_vertical_movement_loop_failure:
    ; we get here when the piece can no longer fall
    mov byte al, [player_input_pressed]
    test al, al
     ; if no player input is present during this last frame, then cement right
    ; away, because the player isn't trying to slide or rotate the piece at the
    ; last moment, as it is landing ( shortly after ); this would ultimately
    ; introduced an unnecessary delay when the piece lands, when the player
    ; is already expecting the next piece
    jz handle_vertical_movement_cement_immediately
    ; decrement and check the cement counter to see if it reached zero
    ; if it did, then the piece landed a long enough time ago to be cemented
    ; in place
    mov byte al, [cement_counter]
    dec al
    mov byte [cement_counter], al
    test al, al ; if we reached zero now, it means the piece can finally cement
    jnz main_loop ; we haven't reached zero yet, so render next frame
    ; cement counter is now zero, which means we have to cement the piece
x2:
   jnz x3
; Current piece can now be "cemented" on whatever it landed
handle_vertical_movement_cement_immediately:
    ; since the cement counter isn't guaranteed to be zero, we should zero it
    mov byte [cement_counter], 0
        ; it cannot move down, so "cement it in place" by changing its colour
    ; by indexing in the cemented piece colours array
     mov word bx, [current_piece_colour_index]
    shl bx, 1 ; each colour is a word, so offset by double the index
    mov byte dl, [colour_cemented_piece + bx]
    call procedure_draw_piece


    ; remove possibly full lines
    xor dx, dx ; we'll accumulate number of lines cleared in dx
    mov cx, 20 ; we're clearing at most 4 lines, each
             ; having a height of 5 pixels
handle_vertical_movement_cement_immediately_attempt_clear_lines_loop:
    push dx
    call procedure_attempt_line_removal
    pop dx
; accumulate number of cleared lines in dx and continue to loop
     add dl, al
    loop handle_vertical_movement_cement_immediately_attempt_clear_lines_loop
update_score:
    ; dx now contains number of lines (not block lines!) cleared, so we must
    ; divide in order to convert to block lines (or actual "tetris" lines).
    mov ax, dx
    mov dl, [block_size]
    div dl ; al now contains number of block lines
    xor ah, ah
        ; add number of cleared lines to the score
    mov word dx, [score]
    add ax, dx
      ; if score reached 1000, it rolls back to 0
    cmp ax, 1000                    ; our scoring goes to 999, so restart at 0 if it goes over
    jl score_is_not_over_1000
    sub ax, 1000

x3:
   jnz x4
score_is_not_over_1000:
    mov word [score], ax
     ; spawn new piece
    jmp new_piece
; Current piece will now move down one pixel
handle_vertical_movement_move_down_success:
      ; re-start cement counter, in case the piece landed on something, but the
    ; player slid it off during the cementing period, causing it to start
    ; falling again, in which case we want to allow sliding again when it
    ; lands on something again
       mov byte [cement_counter], 10
     ; it can move down, and our delta will be one pixel lower
      mov ax, [screen_width]
    mov word [piece_position_delta], ax
     ; delta is now one row lower
     ; move piece down and display it
    call procedure_apply_delta_and_draw_piece
    ; render next frame
      jmp main_loop

; Game has ended because the screen has filled up (next piece can no
; longer spawn)
x4:
  jnz done
game_over_loop:
    ; still display logo
    call procedure_display_logo
    call procedure_delay
    ; advance frame, since we're still animating the logo
      mov word ax, [current_frame]
    inc ax
    mov word [current_frame], ax
     ; check whether any key is pressed
    mov ah, 1
    int 16h ; any key pressed ?
    jz game_over_loop ; no key pressed
    ; read key
    xor ah, ah
    int 16h
    cmp al, 'q'
    jne game_over_loop ; wait for Q to be pressed to exit the program
; Exit to the operating system
done:
     ; change video mode to 80x25 text mode
    mov ax, 3
    int 10h ; restore text mode
    ; return to the operating system
    ret

; Procedures
procedure_display_score:
    ; divide by 100 and convert to the character '0', '1', '2', ... , '9',
    ; storing it in the first position of our 3-digit string buffer
    mov word ax, [score]
    mov dl, 100
    div dl ; hundreds in al, remainder in ah
    mov cl, '0'
    add cl, al
    mov byte [msg_score_buffer], cl ; set hundreds digit
      ; divide by 10 and convert to the character '0', '1', '2', ... , '9',
    ; storing it in the second position of our 3-digit string buffer
    mov al, ah ; divide remainder again
    xor ah, ah
    mov dl, 10
    div dl ; tens in al, remainder in ah
    mov cl, '0'
    add cl, al
    mov byte [msg_score_buffer + 1], cl ; set tens digit

     ; convert remainder to the character '0', '1', '2', ... , '9',
    ; storing it in the third position of our 3-digit string buffer
    mov cl, '0'
    add cl, ah
    mov byte [msg_score_buffer + 2], cl ; set units digit
    ; display string representation of score
    mov bx, msg_score_buffer
    mov dh, 15
    mov dl, 26
    call procedure_print_at
    ret
; Print a string at the specified location
;
; Input:
;        dh = row
;        dl = column
;        bx = address of string

procedure_print_at:
    ; position cursor
    push bx
    mov ah, 2
    xor bh, bh
    int 10h
    ; output string
    mov ah, 9
    pop dx
    int 21h
    ret


; Create next piece
procedure_random_next_piece:
      call procedure_delay ; advance random number (or seed for the initial call)
     ; piece index will be randomly chosen from [0, 6] inclusive
    mov bl, 7
    call procedure_generate_random_number ; choose a piece (in ax)
    mov word [next_piece_colour_index], ax ; save colour index
     ; orientation will be randomly chosen from [0, 3] inclusive
     mov bl, 4
    call procedure_generate_random_number ; choose one of four piece
                                          ; orientations (in ax)
    mov word [next_piece_orientation_index], ax
    ret
procedure_attempt_line_removal:
    push cx
        ; start at bottom left position of play area
     mov di, 47815
    mov cx, 104 ; we'll check at most all but one lines of the play area
                ; there are 20 block lines, and each block line is 5 pixels
                ; tall with an additional top line to accommodate pieces with
                ; an empty top block line in some of their orientations
attempt_line_removal_loop:
    ; if this line is full (no black pixels), we will shift all lines above it
    ; down by one line each
    call procedure_is_horizontal_line_full
    test al, al
    jz attempt_line_removal_full_line_found
    ; this line isn't full (it has gaps), so continue with next line above
     sub di, [screen_width] ; move one line up
    loop attempt_line_removal_loop
      ; no completely full lines has been found
    jmp attempt_line_removal_no_line_found
attempt_line_removal_full_line_found:
     ; di now points to the left most pixel of the full line we're removing
    ; and cx takes our next loop to the second line from the top (inclusive)
  attempt_line_removal_shift_lines_down_loop:
        ; save outer loop (for each line, going upwards)
     push cx
    push di
    ; set source pointer for the memory copy operation to be one line above
    ; our current line
    mov si, di
    sub si, [screen_width] ; line above (source)
        ; destination pointer for the memory copy operation is in di, and is
    ; set to the current line, as it should be
    ; memory copy operation will execute 50 times, going pixel-by-pixel to the
    ; right, copying the line above current line into current line
    mov cx, 50
        ; execute memory copy operation within the video memory segment, restoring
    ; data segments after
     push ds
    push es
    mov ax, 0A000h  ; we'll be reading and writing within the video segment
    mov ds, ax                ; so source segment will be this segment as well
    mov es, ax                ; and so will the destination segment
    rep movsb
    pop es
    pop ds
      ; restore outer loop (for each line, going upwards)
    pop di
    pop cx
    ; next line (upwards)
     sub di, [screen_width] ; move one line up

    loop attempt_line_removal_shift_lines_down_loop
        ; after the last iteration of our shift-lines-down-by-one loop,
    ; di is at the beginning of the top most line; this is exactly where we
    ; need it in order to empty (set all pixels to black) the top-most line
    xor dl, dl
    mov cx, 50
    call procedure_draw_line ; empty the top most line
     ; return the fact that we did clear one line
      mov al, 1
    jmp attempt_line_removal_done

attempt_line_removal_no_line_found:
    ; return the fact that no lines were cleared
    xor al, al
attempt_line_removal_done:
    pop cx
    ret
; cx is preserved
; di is preserved
; Check a line to see whether it is full (meaning it contains no black pixels)
; Input:
;       di - position
; Output:
;       al - 0 if line is full
procedure_is_horizontal_line_full:
    push cx
    push di
       ; for each pixel, going to the right, starting at di
    mov cx, 50 ; width of play area is 10 blocks



is_horizontal_line_full_loop:
    ; if current pixel is black, then this line cannot be full
    call procedure_read_pixel
    test dl, dl ; is colour at current location black?
    jz is_horizontal_line_full_failure
    ; next pixel
    inc di ; next pixel of this line
    loop is_horizontal_line_full_loop
    ; if we got here, it means we haven't found any black pixels, so the line
    ; is full; ax is set accordingly to return the fact that the line is full
    xor ax, ax
    jmp is_horizontal_line_full_loop_done
is_horizontal_line_full_failure:
    ; return the fact that the line isn't full
    mov al, 1
is_horizontal_line_full_loop_done:
    pop di
    pop cx
    ret

procedure_generate_random_number:
    ; advance random number
     mov al, byte [random_number]
    add al, 31
    mov byte [random_number], al
    ; divide by N and return remainder
    div bl ; divide by N
    mov al, ah ; save remainder in al
    xor ah, ah
    ret
; Change current piece orientation to the
; orientation specified in [piece_orientation_index]
procedure_copy_piece:
    push ds
    push es
     ; both source and destination segments will be the same as the code segment
        mov ax, cs ; all code is within this segment
    mov ds, ax ; so source segment will be this segment as well
    mov es, ax ; and so will the destination segment
    ; destination of memory copy operation is the current piece blocks array
     mov di, piece_blocks ; pointer to current orientation (destination)
        ; source of memory copy operation is the current piece origin, offset by
    ; the orientation specified in [piece_orientation_index]
     mov ax, [piece_orientation_index] ; choose k-th orientation
                                   ; of this piece ( 0 through 3 )
    mov si, [piece_definition] ; piece_definition is a pointer to
                               ; first orientation of current piece (source)
    shl ax, 3 ; ax := ax * 8 ( 4 words for each orientation )
    add si, ax ; offset orientation within the current piece
      ; copy each of the four blocks
    mov cx, 4
    rep movsw ; perform copy
    pop es
    pop ds
     ret
  ; Applies a movement delta (causing either vertical or horizontal movement of
; the current piece
procedure_apply_delta_and_draw_piece:
    ; erase old piece
    mov dl, 0
    call procedure_draw_piece
    ; apply delta
     mov ax, [piece_position]
    add ax, [piece_position_delta]
    mov [piece_position], ax
    ; draw new piece
    mov word bx, [current_piece_colour_index]
    shl bx, 1 ; two bytes per colour
    mov byte dl, [colour_falling_piece + bx]
    call procedure_draw_piece
    ret
; Draw the blocks within the current piece at current position
; Input:
;       dl - colour
procedure_draw_piece:
    ; for each of the piece's four blocks
     mov cx, [blocks_per_piece]
draw_piece_loop:
    ; set di to the origin (top left corner) of this piece
    mov di, [piece_position]
    ; and then offset it to the origin (top left corner) of the current block
    mov bx, cx
    shl bx, 1 ; bx := bx * 2
    sub bx, 2 ; our index is zero-based, while cx/loop are one-based
    add di, word [piece_blocks + bx] ; shift position in the piece
```

```
                                  ; to the position of current block
     ; di now points to the origin of the current block, so we can draw it
    mov bx, [block_size]
    call procedure_draw_square
        ; next block of this piece
     loop draw_piece_loop
    ret
; Checks if current piece can be placed in its current position
; This can be used to check if we can still spawn pieces (whether the
; game has ended), or if we can rotate a certain piece (since existing
; "cemented" blocks could be in the way, or we could be too close to the
; edge or bottom)
; Output
;        al - 0 if piece can be placed at current location
procedure_can_piece_be_placed:
    ; for each of the piece's four blocks
    mov cx, [blocks_per_piece] ; each piece has 4 blocks
can_piece_be_placed_loop:
    ; set di to the origin (top left corner) of this piece
    mov di, [piece_position]
    ; and then offset it to the origin (top left corner) of the current block
    mov bx, cx
    shl bx, 1 ; bx := bx * 2
    sub bx, 2 ; our index is zero-based, while cx/loop are one-based
    add di, word [piece_blocks + bx] ; shift position in the piece
                                  ; to the position of current block
     ; preserve outer loop (for each block of current piece)
    push cx ; don't mess up the outer loop
    ; inner loop will check horizontal lines, so the pixel increment is 1
    mov bx, 1 ; horizontal lines
      ; di now points to the first horizontal line of this block
    ; for each of this block's horizontal lines
    mov cx, [block_size]
can_piece_be_placed_line_by_line_loop:
    ; if current line is not available, we cannot place this piece
    call procedure_is_line_available
    test al, al ; a non-zero indicates an obstacle
    jne can_piece_be_placed_failure
    ; next horizontal line of this block
    add di, [screen_width]
    loop can_piece_be_placed_line_by_line_loop
       pop cx


    loop can_piece_be_placed_loop
    xor ax, ax
    jmp can_piece_be_placed_success
can_piece_be_placed_failure:
    mov al, 1
    pop cx
can_piece_be_placed_success:
  ret
procedure_advance_orientation:
    mov word ax, [piece_orientation_index]
    inc ax
    and ax, 3 ; ax := (ax + 1) mod 4
    mov word [piece_orientation_index], ax
    call procedure_copy_piece
    ret
procedure_read_character:
    mov ah, 1
    int 16h ; any keys pressed?
    jnz read_character_key_was_pressed ; yes
    ret
read_character_key_was_pressed:
    mov ah, 0
    int 16h
    push ax
    mov ah, 6 ; direct console I/O
    mov dl, 0FFh ; input mode
    int 21h
    pop ax


handle_input:
     cmp al, 's'
    je move_right
    cmp al, 'a'
    je move_left
    cmp al, ' '
    je rotate
    cmp al, 'q'
    je quit
   ret

quit:
    mov byte [must_quit], 1
    ret
rotate:
      push word [piece_orientation_index]
     call procedure_advance_orientation
     call procedure_can_piece_be_placed
    test al, al ; did we get a 0, meaning ok
    jz rotate_perform ; yes!
        pop word [piece_orientation_index]
    call procedure_copy_piece
    ret
rotate_perform:
    pop word [piece_orientation_index]
    call procedure_copy_piece
       xor dl, dl ; black colour
    call procedure_draw_piece
      call procedure_advance_orientation
      mov al, byte [random_number]
    add al, 11
    mov byte [random_number], al
    ret


move_right:
    mov byte [player_input_pressed], 1
      mov cx, [blocks_per_piece]
move_right_loop:
     mov di, [piece_position]
        mov bx, cx
    shl bx, 1 ; bx := bx * 2
    sub bx, 2 ; our index is zero-based, while cx/loop are one-based
    add di, word [piece_blocks + bx] ; shift position in the piece
                                     ; to the position of current block
        add di, [block_size]
    mov bx, [screen_width]
    call procedure_is_line_available
     test al, al ; did we get a 0, meaning success ?
    jnz move_right_done ; no
        loop move_right_loop
    mov ax, [piece_position_delta]
    add ax, [block_size]
    mov [piece_position_delta], ax

move_right_done:
     mov al, byte [random_number]
    add al, 3
    mov byte [random_number], al
    ret
move_left:
     mov byte [player_input_pressed], 1
      mov cx, [blocks_per_piece]
move_left_loop:
      mov di, [piece_position]
      mov bx, cx
    shl bx, 1 ; bx := bx * 2
    sub bx, 2 ; our index is zero-based, while cx/loop are one-based
    add di, word [piece_blocks + bx] ; shift position in the piece
                                     ; to the position of current block
     dec di
     mov bx, [screen_width]
    call procedure_is_line_available
     test al, al ; did we get a 0, meaning success ?
    jnz move_left_done ; no
        loop move_left_loop
      mov ax, [piece_position_delta]
    sub ax, [block_size]
    mov [piece_position_delta], ax
```

```
move_left_done:
    mov al, byte [random_number]
    add al, 5
    mov byte [random_number], al
    ret

procedure_can_move_down:
    push cx
    push di
      mov cx, [block_size]
can_move_down_find_delta:
    add di, [screen_width]
    loop can_move_down_find_delta
    mov bx, 1
    call procedure_is_line_available
       test al, al ; did we get a 0, meaning success ?
    jnz can_move_down_obstacle_found ; no
    xor ax, ax
    jmp can_move_down_done

can_move_down_obstacle_found:
    mov ax, 1
can_move_down_done:
    pop di
    pop cx
    ret
procedure_is_line_available:
    push bx
    push cx
    push di
       mov cx, [block_size]
is_line_available_loop:
    call procedure_read_pixel
    test dl, dl ; is colour at current location black?
    jnz is_line_available_obstacle_found
    is_line_available_loop_next_pixel:
    add di, bx ; move to next pixel of this line
    loop is_line_available_loop
    xor ax, ax
    jmp is_line_available_loop_done
is_line_available_obstacle_found:
      push bx
    mov word bx, [current_piece_colour_index]
    shl bx, 1 ; two bytes per colour
    mov byte al, [colour_falling_piece + bx]
    cmp dl, al ; if obstacle is a falling block, treat it as a non-obstacle
    pop bx
    jne is_line_available_failure
    jmp is_line_available_loop_next_pixel

is_line_available_failure:
    mov al, 1

is_line_available_loop_done:
    pop di
    pop cx
    pop bx

    ret
procedure_delay:
    push bx
    push cx
    push dx
    push ax
    xor bl, bl
    mov ah, 2Ch
    int 21h
    mov byte al, [random_number]
    add al, dl
    mov byte [random_number], al
    mov [delay_initial], dh
    add dl, [delay_centiseconds]
    cmp dl, 100
    jb delay_second_adjustment_done
      sub dl, 100
    mov bl, 1
```

```
delay_second_adjustment_done:
       mov [delay_stopping_point_centiseconds], dl

read_time_again:
    int 21h

    test bl, bl ; will we stop within the same second?
    je must_be_within_same_second
     ; second will change, so we keep polling if we're still within
    ; the same second as when we started
       cmp dh, [delay_initial]
    je read_time_again
    push dx
    sub dh, [delay_initial]
    cmp dh, 2
    pop dx
    jae done_delay
     jmp check_stopping_point_reached
    must_be_within_same_second:
    cmp dh, [delay_initial]
    jne done_delay

check_stopping_point_reached:
    cmp dl, [delay_stopping_point_centiseconds]
    jb read_time_again

done_delay:
    pop ax
    pop dx
    pop cx
    pop bx
    ret
procedure_draw_square:
    mov ax, bx
    call procedure_draw_rectangle
    ret
```

```
procedure_draw_rectangle:
    push di
    push dx
    push cx
    mov cx, ax
draw_rectangle_loop:
    push cx
    push di
    mov cx, bx
    call procedure_draw_line
        pop di
    add di, [screen_width]
    pop cx
    loop draw_rectangle_loop
    pop cx
    pop dx
    pop di
    ret
procedure_draw_line_vertical:
    call procedure_draw_pixel
    ; move di one pixel down
    add di, [screen_width]
    ; next pixel
    loop procedure_draw_line_vertical
    ret
procedure_draw_line:
    call procedure_draw_pixel
     ; move di one pixel to the right
    inc di
    ; next pixel
    loop procedure_draw_line

    ret
procedure_draw_pixel:
    push ax
    push es
    mov ax, 0A000h
    mov es, ax
    mov byte [es:di], dl
    pop es
    pop ax
    ret
procedure_read_pixel:
    push ax
    push es
    mov ax, 0A000h
    mov es, ax
    mov byte dl, [es:di]
    pop es
    pop ax
    ret
procedure_draw_border:
    mov dl, 200 ; colour
    mov bx, 4
    mov ax, 200
    ; top left to bottom left
    xor di, di
    call procedure_draw_rectangle
    mov di, 316
    call procedure_draw_rectangle
    mov bx, 317
    mov ax, 4
    xor di, di
    call procedure_draw_rectangle

    mov di, 62720
    call procedure_draw_rectangle
    ret
procedure_draw_screen:
    call procedure_draw_border
draw_screen_play_area:
    mov dl, 27 ; colour

    mov cx, 52
    mov di, 14214
    call procedure_draw_line
    mov cx, 52
    mov di, 48134
    call procedure_draw_line
    mov cx, 105
    mov di, 14534
    call procedure_draw_line_vertical
    mov cx, 105
    mov di, 14585
    call procedure_draw_line_vertical

draw_screen_next_piece_area:
    mov di, 16199
    mov cx, 31
    call procedure_draw_line
    mov di, 25799
    mov cx, 31
    call procedure_draw_line
    ; top left to bottom left
    mov di, 16199
    mov cx, 31
    call procedure_draw_line_vertical
    mov di, 16230
    mov cx, 31
    call procedure_draw_line_vertical

draw_screen_strings:
    mov dh, 21
    mov dl, 4
    mov bx, msg_author
    call procedure_print_at
    mov dh, 11
    mov dl, 25
    mov bx, msg_next
    call procedure_print_at
    mov dh, 8
    mov dl, 4
    mov bx, msg_left
    call procedure_print_at
     mov dh, 10
    mov dl, 4
    mov bx, msg_right
    call procedure_print_at
    mov dh, 12
    mov dl, 4
    mov bx, msg_rotate
    call procedure_print_at
    mov dh, 14
    mov dl, 4
    mov bx, msg_quit
    call procedure_print_at
    mov bx, msg_lines
    mov dh, 16
    mov dl, 24
    call procedure_print_at
    mov bx, msg_asmtris
    mov dh, 3
    mov dl, 16
    call procedure_print_at

    ret

procedure_display_logo:
    mov word ax, [current_frame]
    and ax, 3 ; ax := ax mod 4
    jz display_logo_begin

    ret

display_logo_begin:
    mov di, 4905
    mov cx, 20
display_logo_horizontal_loop:
    mov word ax, [current_frame]
    and ax, 8
    shr ax, 3
    add ax, di
    and al, 1
    shl al, 3
    add al, 192
    mov dl, al
    mov bx, 5
    call procedure_draw_square
    push di
    add di, 6400
    call procedure_draw_square
       pop di
    add di, bx
    loop display_logo_horizontal_loop
      mov di, 4905
      mov cx, 5
display_logo_vertical_loop:
      mov word ax, [current_frame]
    and ax, 8
    shr ax, 3
    push ax
    mov ax, di
    mov bl, 160
    div bl
    xor ah, ah
    shr ax, 1
    and al, 1

     pop bx
    add al, bl
    and al, 1
     shl al, 3
    add al, 192
    mov dl, al
    mov bx, 5
    call procedure_draw_square
    push di
    add di, 100
    call procedure_draw_square
     pop di
    add di, 1600
    loop display_logo_vertical_loop
       ret
procedure_display_game_over:
    xor dl, dl
    mov ax, 45
    mov bx, 100
    mov di, 19550
    call procedure_draw_rectangle
    mov dl, 40
    mov ax, 16
    mov bx, 88
    mov di, 29560
    call procedure_draw_rectangle
    mov dh, 12
    mov dl, 16
    mov bx, msg_game_over
    call procedure_print_at

    ret
```

How to Run

1- Download this code and move the 'tetris' folder to C: directory.

2- Install DOSBOX from this link: Download DOSBOX Emulator

3- After complete installation, go to DOSBOX installation directory and run "DOSBox 0.74 Options.bat". This will save you from the pain of searching the configuration file yourself and will open that file for you. Copy these lines at the end of that file:

```
mount c: c:\tetris
```

```
c:
```

4- Now to run the code, run DOSBOX 0.74 and type

```
nasm tetris.asm -o tetris.com
```

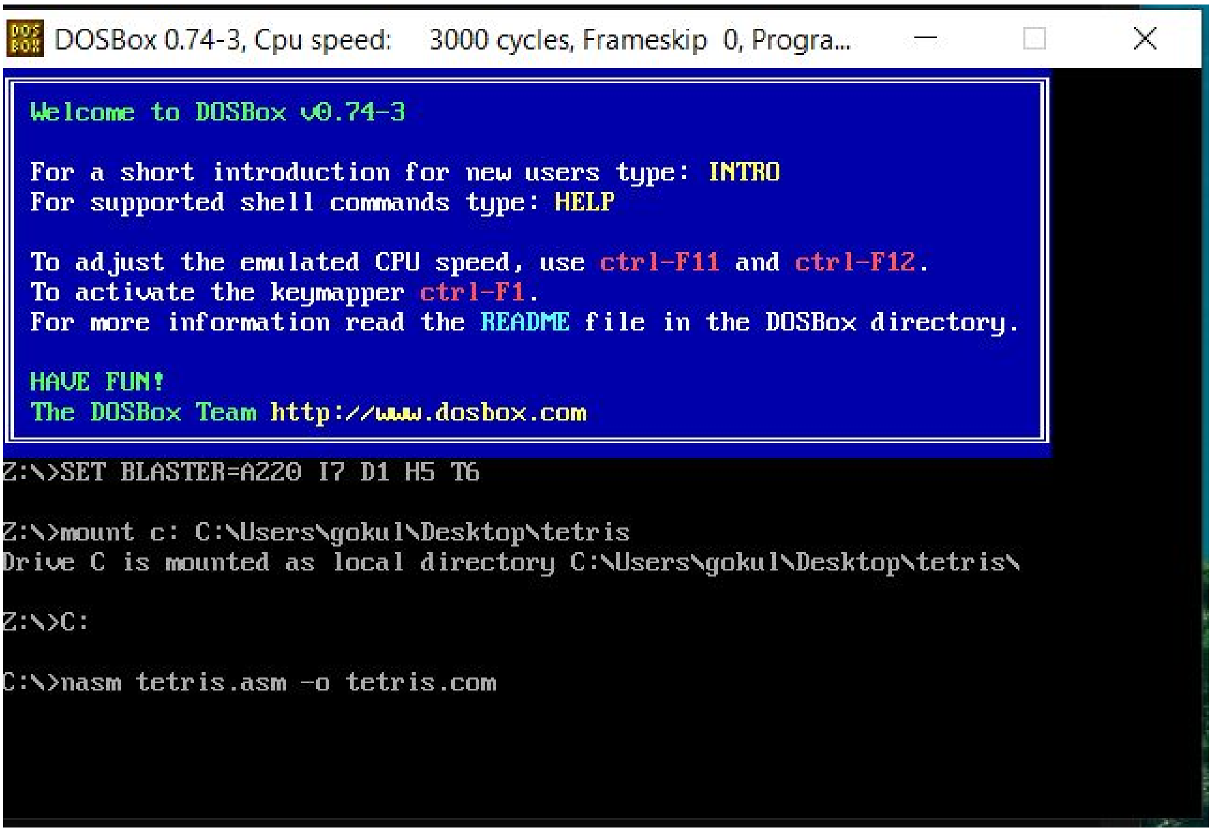
To run the stop watch, type:

```
tetris.com
```

To examine step by step working of the code, type

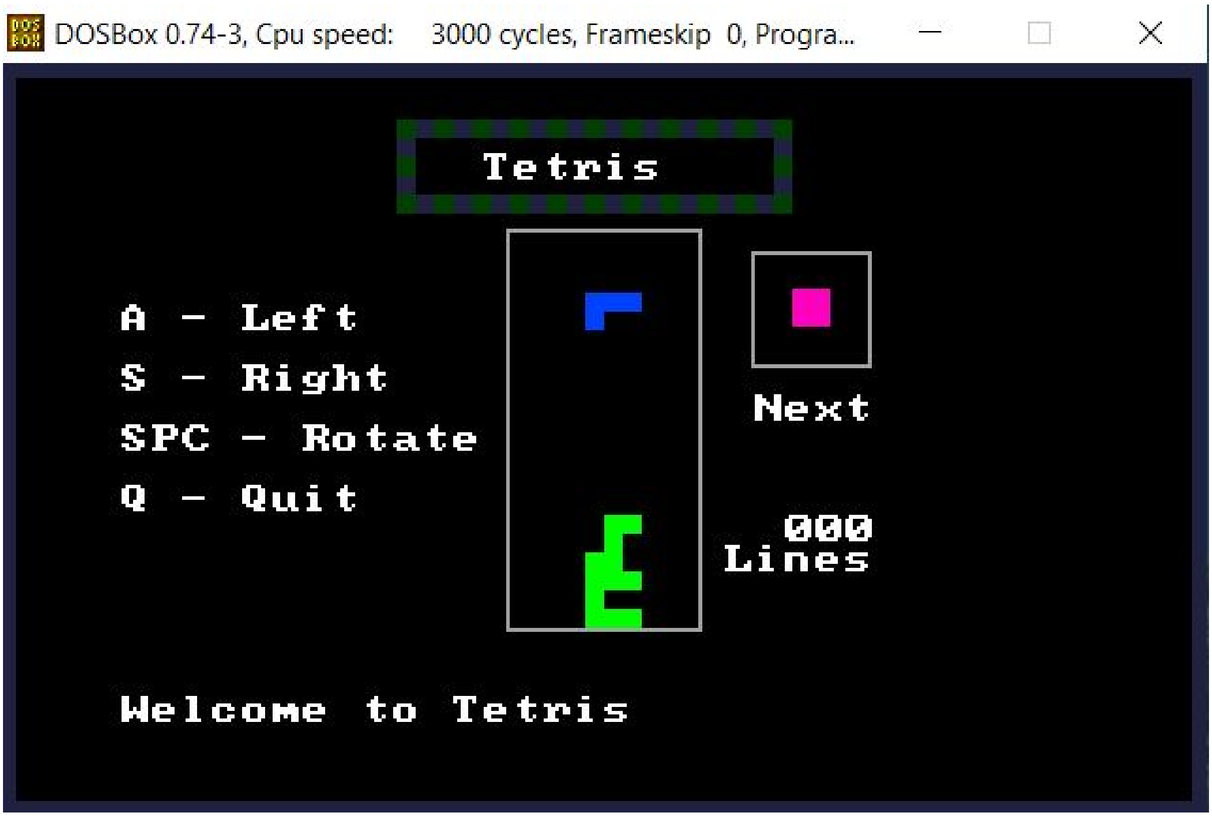
```
afd tetris.com
```

OUTPUTS



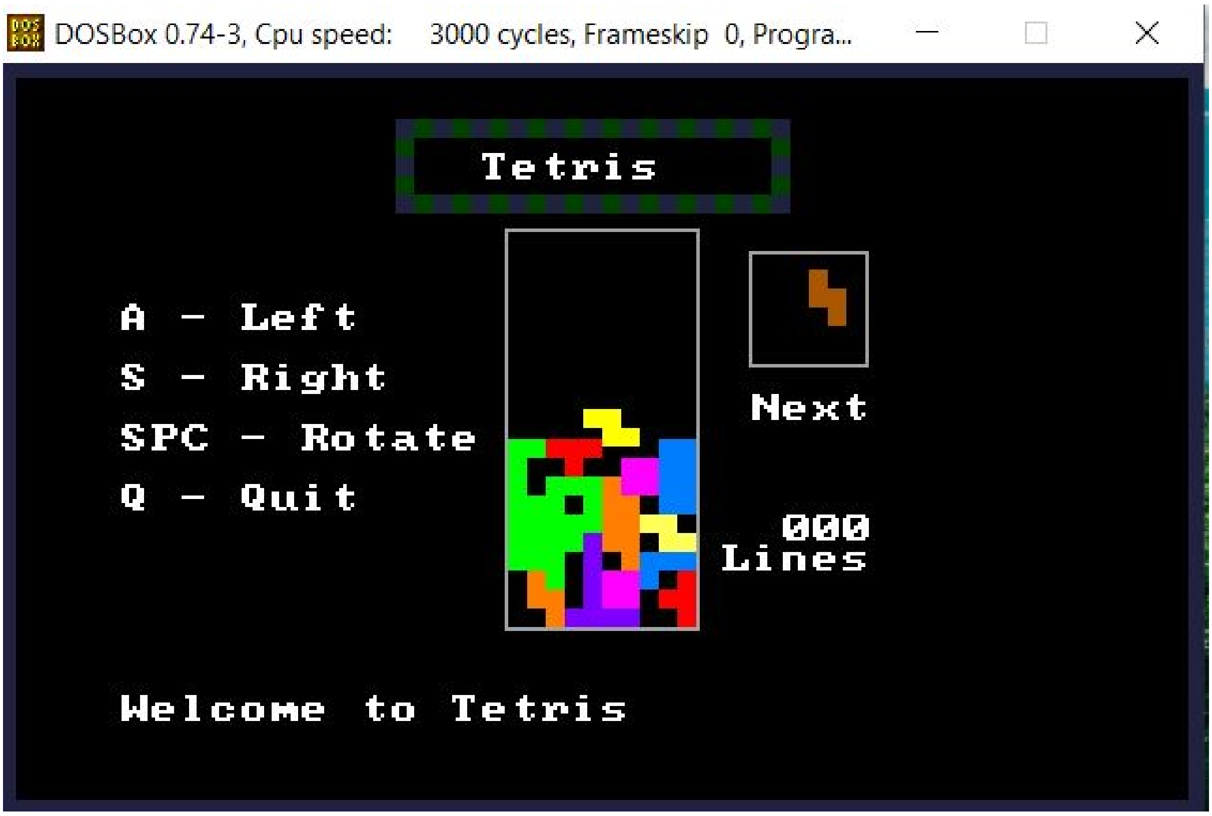
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...

```
Welcome to DOSBox v0.74-3

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: C:\Users\gokul\Desktop\tetris
Drive C is mounted as local directory C:\Users\gokul\Desktop\tetris\

Z:\>C:

C:\>nasm tetris.asm -o tetris.com
```



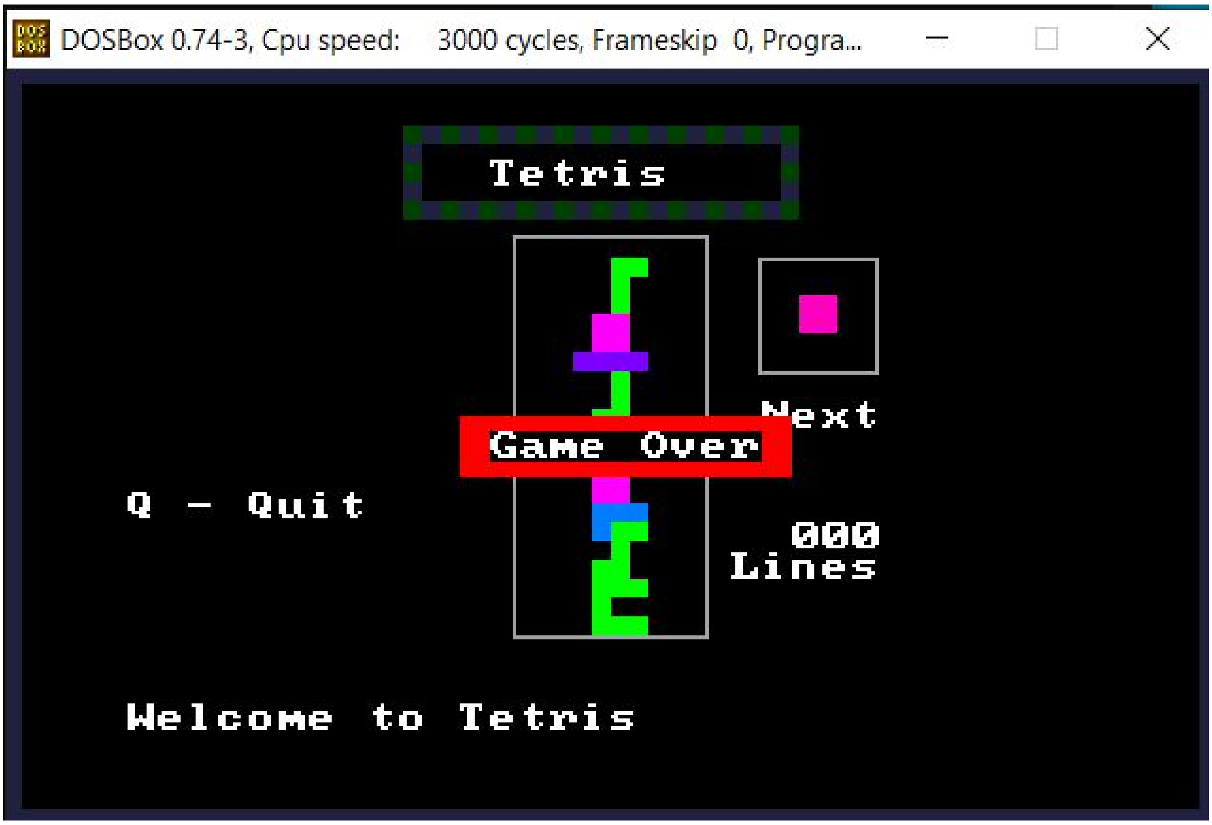
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...

```
Tetris

A   - Left
S   - Right
SPC - Rotate
Q   - Quit

Next

000
Lines

Welcome to Tetris
```



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...

```
Tetris

A - Left
S - Right
SPC - Rotate
Q - Quit

Next

000
Lines

Welcome to Tetris
```



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...

```
Tetris

Next
Game Over

Q - Quit

000
Lines

Welcome to Tetris
```