

COSC 2410: Final Lab Problem

Name: _____

Dr. Mirkovic, Spring 2004

Due: 4/13/2004

Part I – Programming

You will program a scaled down version of the game Breakout. For those of you who are unfamiliar with Breakout, it was invented in 1976. It was a variation of the game Pong that came out earlier from Atari. What you need to do is program an interface to accept keyboard input that will update the location of a paddle, which the ball bounces off of. You are given several constants which are specified in the file “constants.inc.”

Ball starts on the screen at position “STARTX,” and “STARTY.” The ball will always start from this position if it falls off the screen. It will fall off of the screen if it ever goes beyond the last row on the screen. Every time it falls off the screen, you need to reset the ball position to “STARTX,” and “STARTY.” The ball has a x velocity “STARTVELX,” and a y velocity “STARTVELY.” As the game progresses, the ball moves. To calculate the new position of the ball, simply add the x and y velocity to the current ball position to obtain the new column and row position of the ball. If the ball hits either side of the screen (left or right) it will bounce back in the opposite direction and continue to move in the opposite direction (reverse x velocity). The same applies if the ball hits the top of the screen, it will reverse direction. If the ball is about to hit the paddle at the bottom of the screen, then it will also reverse the y direction of the ball. As stated, the only time you worry about the ball not bouncing back will be if it’s the bottom of the screen. If the ball is about to hit the paddle then its y direction needs to change. Furthermore, if the paddle is also directly under or directly above the paddle, then the direction of the ball reverses. When the ball moves, you need to check to see if the ball is currently located in a position where a brick is. The bricks are what you are aiming for. Here is really where our version of breakout differs. For the sake of simplicity we are only going to worry about changing the y direction of the ball if it hits a block. So, if the ball is currently on top of a block then change the current y direction of the ball. So when it hits a brick, it immediately changes direction. The positions of the bricks are specified with the constants in the “constants.inc,” file and specify a large block of bricks in a rectangular fashion. When a brick is hit, that brick disappears

The paddle bar will be controlled by the user using the arrow keys and will of a length specified by the constant “PADDLELEN,” which can be found in the file “constants.inc.” The start position of the paddle is specified by constants “PADDLEX,” and “PADDLEY.”

You need to display these bricks to the screen as well as the ball and the paddle all the time and constantly update their position. If the right arrow key is pressed, then the paddle moves right. If the left arrow key is pressed, then the paddle moves left. If the down arrow is pressed, then paddle moves down. If the up arrow is pressed, then the arrow moves up. The paddle may not move beyond the left or right side of the screen. The paddle can move up and down and must stay between rows 13 and 23 (inclusive – row count starting from row 0). If there is no input for the movement of the paddle, then it doesn’t move and everything else progresses as always. This means that the ball still needs to be updated and newest position displayed to the screen.

If the number of bricks displayed on the screen is 0, then the program ends. If the user ever hits the escape key, then the program ends.

You are NOT allowed to use any procedures from “util.lib,” or any pre-made macros. You may NOT use the interrupts for writing the screen, you must do it MANUALLY.

You may write to the video segment (text mode) at segment 0B800h. This constant is located in the “constants.inc,” file. Every screen position in this segment is composed of 2 bytes (1 word). The high part is the color attributes and the low part is the ASCII character to display.

Use the keyboard interrupts and OS interrupts to perform checking of the keyboard buffer and to perform processing on any input if present or to continue program execution if no input is currently available. Once everything is done, update the screen and start the process all over again.

You will find skeleton code for the main loop on the lab web page. There is one stipulation, you must keep track of the screen in an array called back buffer so that you might learn how to manipulate an array. In general, many games actually use a back buffer to prevent flickering of the screen, but it isn’t needed for this in this case. As stated, you are using a back buffer so that you can learn how memory is organized. This means that there is one required function “swapbuffer” which will copy the back buffer to the screen.

NOTE: You may need to add slowdown code so the cube movement is visible in an acceptable manner.

Grade distribution:

Part 1	
Ball movement	20
User input	20
Swapbuffer	20
Following Instructions	10%
Comments	
Program Description	5%
Code Comments	5%
Misc./Efficiency	10%


```

                int 16h
                @Cls
                @Setcsrpos,0,0
        .exit
main endp

;*****
;Function: drawscreen
;Description:
;   This function clears the backbuffer, draws
;   the remaining blocks to the backbuffer,
;   draws the ball to the backbuffer, draws the
;   the paddle to the backbuffer, and then
;   finally draws the backbuffer to the screen
;Input:
;   None
;Output:
;   None
;Return:
;   None
;*****
drawscreen proc c uses ax bx cx dx
    invoke clearbuffer,addr backbuff

    ;draw remaining blocks
    invoke writebuffer,0,0,addr blockbuff,2000,0FFh, addr backbuff

    ;draw ball
    invoke writebuffer,ballx,bally,addr paddlebar, 1, ballcolor, addr backbuff

    ;draw paddle
    invoke writebuffer,xpos,ypos,addr paddlebar,paddlelen,0Fh, addr backbuff

    ;swap buffer
    call swapbuffer

    ret
drawscreen endp

checkybrick proc c uses bx cx dx

    mov dx, 0
    mov ax, VIDCOL
    mov bx, bally
    mul bx
    add ax, ballx
    shl ax, 1
    lea bx, blockbuff
    add bx, ax
    mov ax, [bx]
    cmp ah, 00h
    jz checkunderpaddle
    invoke writebuffer, ballx, bally, addr paddlebar, 1, 00h, addr blockbuff
    inc score
    mov ax, TRUE
    jmp continue
checkunderpaddle:
    mov ax, bally
    inc ax
    cmp ax, ypos
    jnz setfalse
    mov dx, 0
    mov ax, VIDCOL
    mov bx, bally
    inc bx
    mul bx
    add ax, ballx
    shl ax, 1
    lea bx, backbuff
    add bx, ax
    mov ax, [bx]
    cmp ah, 00h
    jz checkleftpaddle
    mov ax, TRUE
    jmp continue
checkleftpaddle:
    mov dx, 0
    mov ax, VIDCOL
    mov bx, bally
    inc bx
    mul bx
    add ax, ballx
    inc ax
    shl ax, 1
    lea bx, backbuff
    add bx, ax
    mov ax, [bx]
    cmp ah, 00h
    jz checkrightpaddle
    mov ax, TRUE
    jmp continue
checkrightpaddle:
    mov dx, 0
    mov ax, VIDCOL
    mov bx, bally
    inc bx
    mul bx

```

```

        add ax, ballx
        dec ax
        shl ax, 1
        lea bx, backbuff
        add bx, ax
        mov ax, [bx]
        cmp ah, 00h
        jz setfalse
        mov ax, TRUE
        jmp continue
setfalse:
        mov ax, FALSE
continue:
        ret
checkybrick endp

```

updateball proc c uses ax bx cx dx

```

        cmp moveball, 350
        jnz continue
        mov moveball, 0
updatex:
        mov ax, ballx
        add ax, ballvelx
        mov ballx, ax
        cmp ax, VIDCOL
        jb updatey
        mov ax, -1
        mov bx, ballvelx
        mov dx, 0
        imul bx
        mov ballvelx, ax
updatey:
        mov ax, bally
        add ax, ballvely
        mov bally, ax
        cmp ax, VIDROW
        jb adjustygreater
        mov ax, -1
        mov bx, ballvely
        mov dx, 0
        imul bx
        mov ballvely, ax
adjustygreater:
        cmp bally, 24
        jng adjustylower
        mov bally, STARTY
        mov ballx, STARTX
        mov ballvely, STARTVELY
        mov ballvelx, STARTVELX
        jmp continue
adjustylower:
        cmp bally, 0
        jge checkycoll
        mov bally, 0
checkycoll:
        call checkybrick
        cmp ax, FALSE
        jz adjustxgreater
        mov ax, -1
        mov bx, ballvely
        mov dx, 0
        imul bx
        mov ballvely, ax
        jmp continue
adjustxgreater:
        cmp ballx, 79
        jng adjustxlower
        mov ballx, 79
        jmp continue
adjustxlower:
        cmp ballx, 0
        jge continue
        mov ballx, 0
continue:
        inc moveball
        ret

```

updateball endp

```

;*****
;Function: clearbuffer
;Description:
;   This function sets the contents of a buffer
;   to all 0's
;Input:
;   buff - address of buffer to clear
;Output:
;   buff - cleared buffer
;Return:
;   None
;*****

```

```

clearbuffer proc c uses ax bx cx dx, buff:word
        mov di, buff           ;move address of buffer into di
        mov cx, 2000          ;mov length of buffer into cx
again:
        mov WORD PTR [di], WORD PTR 0000h;move 00 into location pointed to by di
        add di, 2              ;increment buffer pointer

```

```

                loop again                ;if cx!=0, jump to again
                ret
clearbuffer endp

;*****
;Function: swapbuffer
;Description:
;   This function moves the contents of the
;   backbuffer to the video segment
;Input:
;   None
;Output:
;   None
;Return:
;   None
;*****
swapbuffer proc c uses ax bx cx dx
    pushf                ;save flags
    mov ax, 80           ;mov rows ot ax
    mov bx, 25          ;mov columns to bx
    mul bx              ;multiply ax and cx
    mov cx, ax          ;mov 80*25 into cx
    push VIDSEG         ;push video segment onto stack
    pop es              ;pop video segment into es register
    mov di, 0           ;set di to 0
    lea si, backbuff ;move address of backbuffer into si
again:
    push WORD PTR [si]  ;push backbuff onto stack
    pop WORD PTR es:[di] ;pop backbuff into video segment
    add si, 2           ;increment backbuff pointer
    add di, 2           ;increment video segment pointer
    loop again         ;if cx!=0, jump to again
    popf               ;restore flags
    ret               ;return flow of control to caller
swapbuffer endp

;*****
;Function: keyread
;Description:
;   This function reads the keyboard buffer and
;   updates the paddle position
;Input:
;   s - address to write scan code to
;   k - address to write ascii code to
;Output:
;   s - address with written scan code
;   k - address with written ascii code
;Return:
;   None
;*****
keyread proc c uses ax bx cx dx, s:word, k:word
    pushf                ;save the flags
    mov ah, 0Bh         ;check to see if there is a character in the keyboard buffer
    int 21h            ;request interrupt from OS
    cmp al, 0FFh       ;check to see if there is atleast on character in the buffer
    jnz noadjust       ;al!=0FFh, nothing in the buffer, jump to noadjust
done:
    mov ah, 0000h      ;read character from the keyboard buffer
    int 16h           ;request interrupt from keyboard
    mov bx, s          ;move address of s into bx
    mov [bx], ah       ;store ah, in variable s
    mov bx, k          ;move address of k into bx
    mov [bx], al       ;store al, in variable k
    mov bx, xpos       ;move xpos into bx
    mov oldxpos, bx   ;store current xpos as oldxpos
    mov bx, ypos       ;move ypos into bx
    mov oldypos, bx   ;store current ypos as oldypos
left:
    cmp ah, 04Bh       ;compare scan code to left arrow code
    jnz right         ;scan code is not for left arrow, jump to label right
    sub xpos, 2        ;scan code is for left, decrement xposition
    cmp xpos, 80       ;compare xpos to 80
    jb noadjust       ;if xpos is less than 80, then do nothing, jump to label noadjust
    mov xpos, 0        ;if xpos is greater than 80, then set xpos to 0
    jmp noadjust       ;jump to label noadjust
right:
    cmp ah, 04Dh       ;compare scan code to right arrow
    jnz up            ;scan code is not for right, jump to label up
    add xpos, 2        ;scan code is for right, increase xpos
    cmp xpos, VIDCOL-PADDLELEN;compare xpos to offset from left of screen, paddlen length
    jb noadjust       ;if below, jump to noadjust
    mov xpos, VIDCOL-PADDLELEN;mov xpos to maximum position
    jmp noadjust       ;jump to noadjust
up:
    cmp ah, 048h       ;compare ah to up arrow scan code
    jnz down         ;if ah does not equal up, then jump to label down
    sub ypos, 1        ;subtract current row position by 1
    cmp ypos, 13       ;compare current row position by 13,
    jae noadjust       ;if above, do nothing, jump to noadjust
    mov ypos, 13       ;if above, then resnet ypos to maximum position (min position on screen)
    jmp noadjust       ;jump to label noadjust
down:
    cmp ah, 050h       ;compare ah to down arrow scan code
    jnz noadjust       ;if ah!=050h, do nothing, jump to noadjust
    add ypos, 1        ;ah==050h, increment ypos
    cmp ypos, 23       ;compare ypos to maximum row (min row in screen)
    jb noadjust       ;if below, do nothing, jump to noadjust

```

```

        mov ypos, 23                ;set ypos to minimum row position (max position on screen)
noadjust:
        popf                        ;restore the flags
        ret
keyread endp

;*****
;Function: writebuffer
;Description:
; This function writes a series of characters
; of arbitrary length and color to a specified
; buffer of length and width no larger than the
; size of the screen 80x25
;Input:
; x - column position to write to
; y - row position to write to
; s - address to character array to write to
;     buffer
; a - color attributes to write to buffer
;Output:
; buff - address of buffer to write to
;Return:
; None
;*****
writebuffer proc c uses ax bx cx dx, x:word, y:word, s:word, l:word, a:byte, buff:word
        pushf                        ;save flags
        mov si, s                    ;move address into si
        mov cx, l                    ;move length to cx
again:
        mov di, buff                ;move address of buffer into di
        mov ax, VIDCOL              ;move buffer width into ax
        mul y                        ;multiply ax by row position
        add ax, x                    ;add column position to ax
        shl ax, 1                    ;multiply ax by 2 to get offset
        add di, ax                  ;add offset to buff address in di
        cmp a, 0FFh                 ;compare attribute to 0FFh
        jnz assigncolor             ;if a!=0FFh then jump to label assigncolor
        mov ax, [si]                ;move character in source into ax
        add si, 2                    ;increment source pointer
        jmp continue                ;jump to label continue
assigncolor:
        mov al, [si]                ;move character from source into al
        mov ah, a                    ;move attributes into ah
        inc si                        ;increment source pointer
continue:
        mov [di], ax                ;mov ah,al pair into destination buffer
        inc x                        ;increment column position
        dec cx                        ;decrement cx, number of characters left
        jnz again                    ;if cx==0, we are done, jump to label again
done:
        popf                        ;restore flags
        ret                          ;return flow of control to caller
writebuffer endp

;*****
;Function: blockbuffinit
;Description:
; This function initializes the blocks that
; that will be displayed to the screen
; that are left on the screen
;Input:
; STARTBRICKX (GLOBAL CONST) - starting column
;     position of the bricks
; STARTBRICKY (GLOBAL CONST) - starting row
;     position of the bricks
; ENDBRICKX (GLOBAL CONST) - the ending column
;     position of the bricks
; ENDBRICKY (GLOBAL CONST) - the ending row
;     position of the bricks
;Output:
; None
;Return:
; None
;*****
blockbuffinit proc c uses ax bx cx dx
        local lx:word, ly:word, k:byte

        pushf                        ;save flags
        mov ax, 00DBh
        mov ly, STARTBRICKY
        mov lx, STARTBRICKX
        mov k, 6
col:
        invoke writebuffer, lx,ly,addr paddlebar,1,k,addr blockbuff
        inc k
        mov al, k
        mov ah, 0
        mov bx, 16
        mov dx, 0
        div bx
        add dx, 3
        mov k, dl
        inc lx
        cmp lx, ENDBRICKX
        jle col
row:

```

```

        inc ly
        mov lx, STARTBRICKX
        cmp ly, ENDBRICKY
        jbe col

        mov ax, ENDBRICKX
        sub ax, STARTBRICKX
        mov bx, ENDBRICKY
        sub bx, STARTBRICKY
        mul bx
        mov blockcount, ax
        popf                                ;restore flags
        ret

blockbuffinit endp

;*****
;Function: checkblockcount
;Description:
;   This function counts the number of blocks
;   that are left on the screen
;Input:
;   blockcount (GLOBAL) - the number of blocks
;   left on the screen
;Output:
;   blockcount (GLOBAL) - the number of blocks
;   left on the screen
;Return:
;   None
;*****
checkblockcount proc c uses ax bx cx dx
    pushf                                ;save flags
    mov blockcount, 0;clear blockcount variable
    lea di, blockbuff;load address of blockbuff
    mov cx, 2000                          ;move 2000 into cx, the size of blockbuff

again:
    cmp cx, 0                             ;compare cx to 0
    jz done                                ;if cx==0, then we are done checking
    mov ax, [di]                           ;move character pointed to by di into ax
    add di, 2                               ;increment di by 2 bytes
    dec cx                                  ;decrement cx, number of buffer spaces to check
    cmp ah, 00h                             ;compare ah, to color black
    jz again                                ;if ah==0 then check next block
    inc blockcount                          ;if ah!=0 then found a block, increment blockcount
    jmp again                               ;jump to label again, start process again

done:
    popf                                    ;restore flags
    ret                                     ;return flow of control to caller
checkblockcount endp

;*****
;Function: putchar
;Description:
;   This function displays a character to the
;   string, at an arbitrary postion, with any
;   color attributes
;Input:
;   x - column position to begin writing
;   character to
;   y - row position to begin writing character
;   to
;   k - character to write to screen
;   a - color attributes of string
;Output:
;   None
;Return:
;   None
;*****
putchar proc c uses ax di es, x:word, y:word, k:byte, a:byte
    pushf                                ;store flags
    mov di, vidseg                         ;move video segment into di
    mov es, di                             ;move di into es (video segment)
    mov ax, vidcol                          ;move ax, the number of video columns
    mul y                                   ;multiply ax by the number row location
    add ax, x                               ;add the column offset
    shl ax, 1                              ;multiply by 2 to get correct byte to write to
    mov di, ax                              ;move segment offset into di
    mov al, k                               ;move display character to al
    mov ah, a                               ;move display character attribute into ah (color, etc)
    mov es:[di], ax                         ;write character to vid memory es:di
    popf                                    ;restore flags
    ret                                     ;return flow of control to caller
putchar endp

;*****
;Function: displaystring
;Description:
;   This function displays a string of arbitrary
;   length to the screen. It will do thia at any
;   position with any color. This function
;   understands CR, LF and will skip to the next
;   line (beginning at x position) when encountered
;Input:
;   x - column position to begin writing string to
;   y - row position to begin writing string to
;   s - address of string to write out to screen
;   l - length of string to write out to screen
;   a - color attributes of string

```

```

;Output:
;      None
;Return:
;      None
;*****
displaystring proc c uses ax bx cx dx, x:word, y:word, s:word, l:word, a:byte
    local beginx:word

        pushf                ;save flags
        mov ax, x             ;mov col position into ax
        mov beginx, ax        ;mov ax into beginx, column position
        mov si, s             ;mov address of string into si
        mov cx, l             ;mov length of string into cx

again:
        cmp cx, 0             ;compare cx to 0
        jz done               ;if cx==0, then we are done writing string, finish
        mov al, [si]          ;not done writing string, move current char into ax
        cmp al, CR            ;compare ax to CR
        jnz continue1        ;if ax!=CR then write the character out to screen
        dec cx                ;ax=CR, decrement characters left,
        inc si                 ;increment si to next character
        inc y                  ;increase current row
        jmp again

continue1:
        mov al, [si]          ;move string character into ax
        cmp al, LF            ;compare ax to LF
        jnz continue2        ;if ax!=LF, then write the character out to screen
        dec cx                ;decrement character count
        inc si                 ;increment si to point to next character
        mov ax, beginx        ;move beginning x into ax
        mov x, ax             ;move ax into x
        jmp again             ;jump to label again, start process over again

continue2:
        invoke putchar, x, y, [si], a
        inc si                 ;increment si to point to next character
        inc x                  ;increment column count
        dec cx                 ;decrement cx, number of characters left in string
        jnz again            ;if cx!=0, jump to label again, start process over again

done:
        popf                  ;restore flags
        ret                    ;return flow of control to caller
displaystring endp

```

```

;*****
;Function: converttoascii
;Description:
;      This function takes an input number of size
;      16 bits (65535 maximum value) and converts
;      it to a string representation
;Input:
;      input - unsigned number that will be
;              converted to a string value
;      outputbuffer - pointer to a string that will
;              contain the ASCII representation of
;              the input number
;      outlen - the length of the outputbuffer in
;              characters
;Output:
;      outputbuffer - a pointer to a modified string
;              now contains the ASCII representation
;              of the input word
;Return:
;      None
;*****
converttoascii proc near c uses ax bx cx dx, input:word, outputbuffer:word, outlen:word
    local tempbuffer[128]:byte

        pushf                ;save flags
        mov ax, input         ;initialize AX
        mov bx, 10            ;initialize BX to base 2
        mov cx, 0             ;clear CX

convert:
        cmp cx, outlen        ;check if CX is over max string length
        je cleanup           ;if CX==outlen then finish
        mov dx, 0             ;clear DX
        div bx                 ;divide AX by BX, store in AX
        add dl, 030h          ;add 30h to remainder to convert to ASCII
        lea si, tempbuffer    ;load address of local buffer
        add si, cx             ;add offset to SI pointer
        mov BYTE PTR [si], dl ;move ASCII value into temporary array
        inc cx                 ;increment offset
        cmp ax, 0             ;check if AX is 0
        jne convert          ;if AX is not 0 then we are not done converting
        mov bx, outputbuffer   ;initialize BX to output array

cleanup:
        mov al, [si]          ;move last converted number into AL
        mov [bx], al          ;move AL into output array
        inc bx                 ;increment BX offset
        dec si                 ;decrement SI offset
        dec cx                 ;decrement number of characters left
        cmp cx, 0             ;check if CX is equal to 0
        jnz cleanup          ;if CX!=0 then we have more char's to move

done:
        popf                  ;restore flags
        ret                    ;return to caller
converttoascii endp

```

end