國 立 台 灣 科 技 大 學

微 算 機 概 論 實 習

指 導 教 授：陸敬互 教 授

微算機概論實習報告
(原創)

期末報告

班　　級：四電機二甲
學　　生：李品翰、陳建安
學　　號：B11107051、B11107053
建檔日期：2023/12/10

一、學習成果

此遊戲名稱為color clash : ballistic showdown，兩位玩家輪流行動，用滑鼠操控球的發射，球撞到牆壁會出現不同的特殊效果；程式說明分為五大部分：主程式、遊戲開始、 遊戲動畫迴圈、幫助介面與其他部分。

1.主程式

主程式的功能為取得系統CPU時脈週期與遊戲啟動的畫面:使用getCPUClockPeriod 函式取得當前系統的週期；我們選擇int 10h ah 13h作為畫面的顯示模式，先列印出封面的背景圖，再用printStringWithAttribute 函式列印出按鍵導覽，並等待玩家輸入按鍵，H為切換至幫助介面，Enter為開始遊戲。

```
main proc
    .startup

    ; video mode, 320 * 200, 256 colors
    mov ax, 13h
    int 10h

    ; random number seed
    rdtsc
    mov [randomNumber], eax

    call getCPUClockPeriod

    .while 1 ; game cycle
        ; cover image
        push ds
        mov ax, coverImage
        mov ds, ax
        lea si, image
        mov ax, 0a000h
        mov es, ax
        xor di, di
        mov cx, imageHeight
        @@:
            push cx
            mov cx, imageWidth
            rep movsb
            add di, 320 - imageWidth
            pop cx
```

```
            loop @b
        pop ds

        printStringWithAttribute enterToStartMessage, 28h, lengthof
enterToStartMessage, 2, 17
        printStringWithAttribute hForHelpMessage, 37h, lengthof
hForHelpMessage, 4, 19


        .while 1 ; wait user input
            mov ah, 00h
            int 16h
            .if al == 1bh ; esc
                call exitGame
            .elseif al == 'h' || al == 'H'
                call helpPage
                .break
            .elseif al == 0dh ; enter
                call gameStart
                .break
            .endif
        .endw
    .endw
main endp
```

本程式需要較精確的計時功能，測試後 dos 提供的計時功能精度太低，8254 晶片的 timer 則太容易溢位，只有 rdtsc 指令較為理想，而為了避免不同電腦的 CPU 時脈不同，首先需要用 getCPUClockPeriod 函式計算出 CPU 時脈週期。使用 8254 計數頻率為 1.19MHZ 的 timer0，等待 4096 個 CPU 時脈，計算期間 timer0 共計數了幾次，將此次數÷4096÷1193182即可算出 CPU 時脈週期，後續只要將 CPU 時脈數乘上時脈週期即可得到實際精確時間。

```
getCPUClockPeriod proc
    ; 1 tick = 1 / 1193182 s
    ; clock = rdtsc cpu clock
    ; clockPeriod
    ;     = time / clock
    ;     = time/tick * tick/clock
    ;     = tickIn4096Clock / 4096 / 1193182
    mov al, 00110000b
```

```
    out 43h, al
    xor al, al
    out 43h, al
    rdtsc
    mov ecx, eax
    in al, 40h
    mov bl, al
    in al, 40h
    mov bh, al
    .repeat
        rdtsc
        sub eax, ecx
    .until eax > 4096 ; clock > 4096
    push eax
    xor al, al
    out 43h, al
    in al, 40h
    mov cl, al
    in al, 40h
    mov ch, al
    pop eax
    sub bx, cx ; bx = tick in 4096 clock
    mov word ptr [fpuTemp], bx
    fild word ptr [fpuTemp] ; st: [tickIn4096Clock]
    mov word ptr [fpuTemp], 4096
    fidiv word ptr [fpuTemp] ; st: [tickIn4096Clock / 4096]
    mov dword ptr [fpuTemp], 1193182
    fidiv dword ptr [fpuTemp] ; st: [tickIn4096Clock / 4096 / 1193182 =
clockPeriod]
    fstp [clockPeriod] ; st: []

    ret
getCPUClockPeriod endp
```

printStringWithAttribute 功能為在特定位置列印有顏色的字串。

```
printStringWithAttribute macro string, color, length, x, y
    push bp
    mov ax, @data
    mov es, ax
    mov ah, 13h
    mov al, 0
    mov bh, 0
    mov bl, color
```

```
    mov cx, length
    mov dh, y
    mov dl, x
    lea bp, string
    int 10h
    pop bp
endm
```

2. 遊戲開始

遊戲開始時，兩顆球個別位於畫面的左上與右下角，初始速度為 0，接著 call generateWall 生成隨機效果的 26 面牆壁，以 rdtsc 指令更新 lastTimerCount，使用 nextRandomNumber 巨集取得亂數，決定紅藍方的先後手開局。

```
gameStart proc
    finit

    ; ball initialization
    mov [redBall.integerX], 50
    fild [redBall.integerX] ; st: [x]
    fstp [redBall.x] ; st: []
    mov [redBall.integerY], 50
    fild [redBall.integerY] ; st: [y]
    fstp [redBall.y] ; st: []
    mov [blueBall.integerX], 320 - 50
    fild [blueBall.integerX] ; st: [x]
    fstp [blueBall.x] ; st: []
    mov [blueBall.integerY], 200 - 50
    fild [blueBall.integerY] ; st: [y]
    fstp [blueBall.y] ; st: []
    mov [redBall.vx], 0
    mov [redBall.vy], 0
    mov [blueBall.vx], 0
    mov [blueBall.vy], 0
    mov [redBall.score], 0
    mov [blueBall.score], 0

    call generateWall

    mov ax, 03h
    int 33h
```

```
    mov [mouseButtonStatus], bl

    rdtsc
    mov [lastTimerCount], eax

    ; reset game status and choose random player to start
    mov [gameStatus], 0
    nextRandomNumber
    .if al & 10000000b
        or [gameStatus], 00000010b ; blue first
        and [gameStatus], 11111011b ; next is red
    .else
        and [gameStatus], 11111101b ; red first
        or [gameStatus], 00000100b ; next is blue
    .endif
    (下略)
gameStart endp
```

牆壁共有 6 種效果，以 6 種顏色表示，首先將第一面牆指定為加分，避免遊戲無法結束，剩下 25 面牆用 nextRandomNumber 巨集，以亂數決定效果，同時填入顏色，各效果機率如程式綠色註解所示；接著將 26 面牆洗牌，避免第一面牆總是加分效果。

```
generateWall proc
    mov [wallEffect + 0 * 2], increaseScore ; at least one increase score
    mov [wallColor + 0 * 1], 2fh ; green
    mov esi, 1
    .while esi != 26
        ;  increaseScore: 15% = ~ 644245094
        ; increase3Score: 2% = ~ 730144440
        ;  decreaseScore: 5% = ~ 944892805
        ;      swapBalls: 2% = ~ 1030792151
        ;      extraTurn: 2% = ~ 1116691497
        ;       noEffect: remain% ~ 4294967295
        nextRandomNumber
        .if [randomNumber] <= 644245094
            mov [wallEffect + esi * 2], increaseScore
            mov byte ptr [wallColor + esi * 1], 2fh ; green
        .elseif [randomNumber] <= 730144440
            mov [wallEffect + esi * 2], increase3Score
            mov byte ptr [wallColor + esi * 1], 34h ; aqua
        .elseif [randomNumber] <= 944892805
```

```
            mov [wallEffect + esi * 2], decreaseScore
            mov byte ptr [wallColor + esi * 1], 28h ; red
        .elseif [randomNumber] <= 1030792151
            mov [wallEffect + esi * 2], swapBalls
            mov byte ptr [wallColor + esi * 1], 22h ; purple
        .elseif [randomNumber] <= 1116691497
            mov [wallEffect + esi * 2], extraTurn
            mov byte ptr [wallColor + esi * 1], 2ch ; yellow
        .else
            mov [wallEffect + esi * 2], noEffect
            mov byte ptr [wallColor + esi * 1], 0fh ; white
        .endif

        inc esi
    .endw
    ; shuffle wall
    ;
https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algo
rithm
    mov bl, lengthof wallEffect ; 26, divisor
    mov ecx, lengthof wallEffect - 1 ; 25
    @@:
        nextRandomNumber
        xor ah, ah ; avoid division overflow
        div bl ; random / bl = al ... ah, 0 <= ah <= cx
        movzx edi, ah
        ; exchange wallEffect[ecx], wallEffect[edi]
        mov ax, [wallEffect + ecx * 2]
        xchg ax, [wallEffect + edi * 2]
        mov [wallEffect + ecx * 2], ax
        ; exchange wallColor[ecx], wallColor[edi]
        mov al, [wallColor + ecx * 1]
        xchg al, [wallColor + edi * 1]
        mov [wallColor + ecx * 1], al

        dec bl
        loop @b

    ret
generateWall endp
```

3. 遊戲動畫迴圈

本遊戲需要大量的物理量運算，所以會定期地讀取 CPU 時脈，時脈x週期取得計算物理量所需的基本單位時間(dt)；為了遊戲畫面完整度，本遊戲使用 VSync 與 backbuffer，以防止畫面撕裂與閃爍。迴圈開始時，清空 backbuffer，讀取當前時脈以計算 dt，並用 dt 計算球的下個位置與速度；clickHandler 函式判斷滑鼠狀態，使球向鼠標方向發射；wallCollision 判斷球是否碰到牆壁，反彈球並觸發各種牆壁效果；ballCollision 函式判斷球之間的碰撞並反彈；接著判斷回合是否結束；draw 函式將虛線、球、牆壁畫到 backbuffer；等待垂直掃描訊號後，將 backbuffer 輸出到畫面上；printScore 函式列印玩家當前的得分，最後檢查獲勝條件決定是否結束遊戲，檢查使用者輸入 ESC 或 H，則中斷或暫停遊戲進入 helpPage。若獲勝條件成立則宣告獲勝，檢查按鍵輸入 ESC 或 Enter，則離開遊戲或回到主選單。

```
gameStart proc
    (上略)
    .while 1
        ; clear backbuffer
        mov ax, backBuffer
        mov es, ax
        xor eax, eax
        xor di, di
        mov cx, (320 * 200) / 4
        rep stosd

        ; get dt
        rdtsc
        mov ebx, eax
        sub eax, [lastTimerCount]
        mov [lastTimerCount], ebx
        mov dword ptr [fpuTemp], eax
        fild dword ptr [fpuTemp] ; st: [dClock]
        fmul [clockPeriod] ; st: [dt]
        fstp [deltaT] ; st: []

        ; calculate next position and velocity
        fld [deltaT] ; st: [dt]
```

```
        invoke step, addr redBall
        invoke step, addr blueBall
        fstp st ; st: []

        call clickHandler

        ; wall collision
        fld [heightMinusRadius] ; st: [h - r]
        fld [widthMinusRadius] ; st: [w - r][h - r]
        fld [circleRadius] ; st: [r][w - r][h - r]
        invoke wallCollision, addr [redBall]
        invoke wallCollision, addr [blueBall]
        fstp st ; st: [w - r][h - r]
        fstp st ; st: [h - r]
        fstp st ; st: []

        call ballCollsion

        ; round not over yet && all velocity == 0: end of round
        mov eax, 7fffffffh ; 01111...111b, float has +0 and -0
        .if [gameStatus] & 1b && !([redBall.vx] & eax) && !([redBall.vy] &
eax) && !([blueBall.vx] & eax) && !([redBall.vy] & eax)
            xor [gameStatus], 00000001b ; end round
            .if [gameStatus] & 00000100b ; who's next
                or [gameStatus], 00000010b ; blue's turn
                and [gameStatus], 11111011b ; next is red
            .else
                and [gameStatus], 11111101b ; red's turn
                or [gameStatus], 00000100b ; next is blue
            .endif
        .endif

        call draw

        ; VSync
        ; https://stackoverflow.com/questions/67066755/assembly-x86-16-bit-
vsync-screen-tearing
        mov dx, 03dah
        @@:
            in al, dx
            test al, 1000b
            jnz @b
        @@:
            in al, dx
            test al, 1000b
```

```
            jz @b

        ; backbuffer -> video memory
        push ds
        mov ax, backBuffer
        mov ds, ax
        mov ax, 0a000h
        mov es, ax
        xor si, si
        xor di, di
        mov cx, (320 * 200) / 4
        rep movsd
        pop ds

        invoke printScore, [redball.score], 28h, 19, 1
        invoke printScore, [blueBall.score], 37h, 19, 23

        ; game over judgement
        .if sbyte ptr [redBall.score] >= 10
            lea si, [redBall]
            .break
        .elseif sbyte ptr [blueBall.score] >= 10
            lea si, [blueBall]
            .break
        .elseif sbyte ptr [redBall.score] <= -10
            lea si, [blueBall]
            .break
        .elseif sbyte ptr [blueBall.score] <= -10
            lea si, [redball]
            .break
        .endif

        mov ah, 06h
        mov dl, 0ffh
        int 21h
        .continue .if zero?
        .if al == 1bh ; esc
            ret
        .elseif al == 'h' || al == 'H'
            call helpPage
            rdtsc
            mov [lastTimerCount], eax
        .endif
    .endw
```

```
    ; print win message
    moveTextCursor 15, 12
    printString (circle ptr [si]).id
    printString winMessage
    ; print play again, exit message
    moveTextCursor (40 - (lengthof playAgainMessage - 1)) / 2, 16
    printString [playAgainMessage]
    moveTextCursor (40 - (lengthof exitMessage - 1)) / 2, 18
    printString [exitMessage]

    .repeat
        mov ah, 00h
        int 16h
        .if al == 1bh ; esc
            call exitGame
        .endif
    .until al == 0dh ; enter

    ret
gameStart endp
```

step 函式用 dt 計算球的新位置與速度，使用公式：$x_{n+1} = x_n + v_n\,dt + \frac{1}{2}a_n\,dt^2$可得到下一個位置，$\frac{1}{2}a_n\,dt^2$過小，此處忽略。假設正向力 n 與質量 m 為 1，k 為摩擦係數，摩擦力所造成的加速度推導如下：

$$f = m\,a = k\,n\,\frac{-v}{\|v\|}\,,a = -k\frac{v}{\|v\|}$$

$$v_{n+1} = v_n + a\,dt = v_n - k\frac{v_n}{\|v_n\|}dt = v_n(1 - k\frac{dt}{\|v_n\|})$$

其中，若因誤差導致$1 - k\frac{dt}{\|v_n\|} < 0$會使得球原地震動，因此若此項小於 0 則直接指定為 0。

```
step proc, pBall: near ptr circle ; st: [dt]
    mov si, [pBall]
    assume si: near ptr circle

    ; newX = x + vx * dt
    fld [si].vx ; st: [vx][dt]
    fmul st, st(1) ; st: [vx * dt][dt]
    fadd [si].x ; st: [x + vx * dt][dt] = [newX][dt]
    fist [si].integerX
```

```
    fstp [si].x ; st: [dt]
    ; newY = y + vy * dt
    fld [si].vy ; st: [vy][dt]
    fmul st, st(1) ; st: [vy * dt][dt]
    fadd [si].y ; st: [y + vy * dt][dt] = [newY1][dt]
    fist [si].integerY
    fstp [si].y ; st: [dt]

    ; friction:
    ;   f = m * a = k * n * (-v / ||v||), m = mass = 1, a = accelleration,
    ;   k = friction coefficient, n = normal force = 1
    ;   a = -k * n / m * (-v / ||v||) = -k * v / ||v||
    ;   v = v0 + a0 * dt = v0 - k * v0 / ||v0|| * dt
    ;     = v0 * (1 - k * dt / ||v0||), k > 0, 1 - k * dt / ||v0|| >= 0
    fld [si].vx ; st: [vx][dt]
    fmul st, st ; st: [vx^2][dt]
    fld [si].vy ; st: [vy][vx^2][dt]
    fmul st, st ; st: [vy^2][vx^2][dt]
    faddp st(1), st ; st: [vx^2 + vy^2][dt]
    fsqrt ; st: [||v||][dt]
    fdivr st, st(1) ; st: [dt / ||v||][dt]
    fmul [frictionCoefficient] ; st: [k * dt / ||v||][dt]
    fld1 ; st: [1][k * dt / ||v||][dt]
    fsubrp st(1), st ; st: [1 - k * dt / ||v||][dt]
    ftst ; if v < 0: v = 0
    fstsw ax
    fwait
    sahf
    jae @f ; 1 - k * dt / ||v|| >= 0
        fstp st ; st: [dt]
        fldz ; st: [0][dt]
    @@:
    fld [si].vx ; st: [vx][1 - k * dt / ||v||][dt]
    fmul st, st(1) ; st: [vx * (1 - k * dt / ||v||)][1 - k * dt /
||v||][dt]
    fstp [si].vx ; st: [1 - k * dt / ||v||][dt]
    fmul [si].vy ; st: [vy * (1 - k * dt / ||v||)][dt]
    fstp [si].vy ; st: [dt]

    assume si: nothing
    ret
step endp
```

gameStatus 第 0bit 代表回合正在進行或結束(1 或 0)，第 1bit 代表當前行動玩家為紅或藍(0 或 1)，發射球的條件為回合結束且左鍵放開(負緣)，速度為 2x(鼠標位置-球位置)，乘 2 為測試後較剛好的數值。

```
clickHandler proc
    mov ax, 03h
    int 33h
    shr cx, 1 ; X max: 640 -> 320
    ; wait for negative edge
    test [mouseButtonStatus], 001b
    jz @f ; not pressed
    test bl, 001b
    jnz @f ; pressed but not released
    test [gameStatus], 00000001b
    jnz @f ; round not over yet
        .if [gameStatus] & 00000010b
            lea si, [blueBall]
        .else
            lea si, [redBall]
        .endif
        assume si: near ptr circle
        or [gameStatus], 00000001b ; new round start
        mov word ptr [fpuTemp], cx
        fild word ptr [fpuTemp] ; st: [mouseX]
        fsub [si].x ; st: [mouseX - x1]
        fadd st, st ; st: [(mouseX - x1) * 2]
        fstp [si].vx ; st: []
        mov word ptr [fpuTemp], dx
        fild word ptr [fpuTemp] ; st: [mouseY]
        fsub [si].y ; st: [mouseY - y1]
        fadd st, st ; st: [(mouseY - y1) * 2]
        fstp [si].vy ; st: []
        assume si: nothing
    @@:
    mov [mouseButtonStatus], bl

    ret
clickHandler endp
```

wallCollision 分為左、右、上、下牆壁碰撞四個部分，四個部分結構相似，大致上為先判斷是否有碰撞，有則將 x(左右牆壁)或 y(上下牆壁)方向速度變號，並預判其反彈後新位置，接著計算撞到的是牆壁 0~25 中的哪一個區域(如下綠色註解)，再觸發該牆壁的效果。

```
wallCollision proc, pBall: near ptr circle
    ; st: [radius][width- radius][height - radius]
    ; 40 pixels per section
    ;
    ;     0  1  2  3  4  5  6  7
    ;    ┌─────────────────────┐
    ; 25 │                     │ 8
    ; 24 │                     │ 9
    ; 23 │                     │ 10
    ; 22 │                     │ 11
    ; 21 │                     │ 12
    ;    └─────────────────────┘
    ;     20 19 18 17 16 15 14 13

    mov si, [pBall]
    assume si: near ptr circle
    ; r = radius, w = width, h = height

    ; left wall
    fld [si].x ; st: [x][r][w - r][h - r]
    fcomi_ 1 ; cmp x, r
    ja @f ; x > r
    bt [si].vx, 31 ; sign bit of float
    jnc @f ; vx > 0
        ; x <= r && vx < 0
        btr [si].vx, 31 ; neg -> pos
        ; x = x + 2(r - x) = 2r - x
        fld st(1) ; st: [r][x][r][w - r][h - r]
        fadd st, st ; st: [2r][x][r][w - r][h - r]
        fsub st, st(1) ; st: [2r - x][x][r][w - r][h - r]
        fst [si].x
        fistp [si].integerX ; st: [x][r][w - r][h - r]

        ; wall effect index = 2 * (wall section)
        ;   = 2 * (25 - floor(y / 40))
        mov ax, [si].integerY
        mov bl, 40
        div bl ; y / 40 = al ... ah, floor(y / 40) = al
```

```
        xor ah, ah
        neg ax
        add ax, 25
        shl ax, 1
        mov bx, ax
        cmp [wallEffect + bx], noEffect
        je topBottomCollision
        invoke [wallEffect + bx]
        jmp topBottomCollision
    @@:
    ; st: [x][r][w - r][h - r]
    ; right wall
    fcomi_ 2 ; cmp x, w - r
    jb @f ; x < w - r
    bt [si].vx, 31 ; sign bit of float
    jc @f ; vx < 0
        ; x >= w - r && vx > 0
        bts [si].vx, 31 ; pos -> neg
        ; x = x - 2(x - (w - r)) = 2(w - r) - x
        fld st(2) ; st: [w - r][x][r][w - r][h - r]
        fadd st, st ; st: [2(w - r)][x][r][w - r][h - r]
        fsub st, st(1) ; st: [2(w - r) - x][x][r][w - r][h - r]
        fst [si].x
        fistp [si].integerX ; st: [x][r][w - r][h - r]

        ; wall effect index = 2 * (wall section)
        ;   = 2 * (8 + floor(y / 40))
        mov ax, [si].integerY
        mov bl, 40
        div bl ; y / 40 = al ... ah, floor(y / 40) = al
        xor ah, ah
        add ax, 8
        shl ax, 1
        mov bx, ax
        cmp [wallEffect + bx], noEffect
        je topBottomCollision
        invoke [wallEffect + bx]
    @@:

    topBottomCollision:
    ; top wall
    fld [si].y ; st: [y][x][r][w - r][h - r]
    fcomi_ 2 ; cmp y, r
    ja @f ; y > r
    bt [si].vy, 31 ; sign bit of float
```

```
    jnc @f ; vy > 0
        ; y <= r && vy < 0
        btr [si].vy, 31 ; neg -> pos
        ; y = y + 2(r - y) = 2r - y
        fld st(2) ; st: [r][y][x][r][w - r][h - r]
        fadd st, st ; st: [2r][y][x][r][w - r][h - r]
        fsub st, st(1) ; st: [2r - y][y][x][r][w - r][h - r]
        fst [si].y
        fistp [si].integerY ; st: [y][x][r][w - r][h - r]

        ; wall effect index = 2 * (wall section)
        ;   = 2 * floor(x / 40)
        mov ax, [si].integerX
        mov bl, 40
        div bl ; x / 40 = al ... ah, floor(x / 40) = al
        xor ah, ah
        shl ax, 1
        mov bx, ax
        cmp [wallEffect + bx], noEffect
        je return
        invoke [wallEffect + bx]
        jmp return
    @@:

    ; bottom wall
    fcomi_ 4 ; cmp y, h - r
    jb @f ; y < h - r
    bt [si].vy, 31 ; sign bit of float
    jc @f ; vy < 0
        ; y >= h - r && vy > 0
        bts [si].vy, 31 ; pos -> neg
        ; y = y - 2(y - (h - r)) = 2(h - r) - y
        fld st(4) ; st: [h - r][y][x][r][w - r][h - r]
        fadd st, st ; st: [2(h - r)][y][x][r][w - r][h - r]
        fsub st, st(1) ; st: [2(h - r) - y][y][x][r][w - r][h - r]
        fst [si].y
        fistp [si].integerY ; st: [y][x][r][w - r][h - r]

        ; wall effect index = 2 * (wall section)
        ;   = 2 * (20 - floor(x / 40))
        mov ax, [si].integerX
        mov bl, 40
        div bl ; x / 40 = al ... ah, floor(x / 40) = al
        xor ah, ah
        neg ax
```

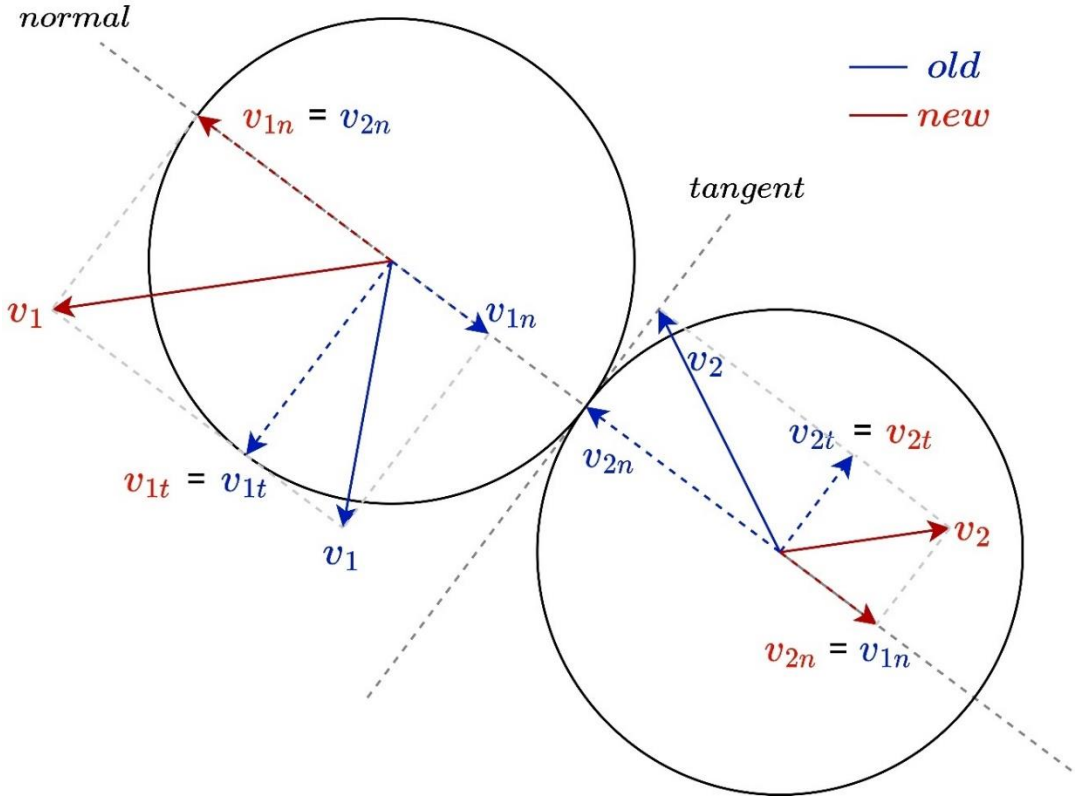
```
        add ax, 20
        shl ax, 1
        mov bx, ax
        cmp [wallEffect + bx], noEffect
        je return
        invoke [wallEffect + bx]
    @@:

    return:
        fstp st ; st: [x][r][w - r][h - r]
        fstp st ; st: [r][w - r][h - r]
        assume si: nothing
        ret
wallCollision endp
```

本遊戲有 6 種牆壁效果，除無效果的牆壁之外，以下為其他五種牆壁效果的程式，其中較為特別的 swapBalls 需要交換兩顆球的位置和速度，及 noEffect 無功能，需要個佔位符 nop。

```
; wall effect function
noEffect proc
    nop
noEffect endp
increaseScore proc
    inc (circle ptr [si]).score
    ret
increaseScore endp
increase3Score proc
    add (circle ptr [si]).score, 3
    ret
increase3Score endp
decreaseScore proc
    dec (circle ptr [si]).score
    ret
decreaseScore endp
swapBalls proc
    mov eax, redBall.x
    xchg eax, blueBall.x
    mov redBall.x, eax
    mov eax, redBall.y
    xchg eax, blueBall.y
    mov redBall.y, eax
    mov eax, redBall.vx
    xchg eax, blueBall.vx
```

```
    mov redBall.vx, eax
    mov eax, redBall.vy
    xchg eax, blueBall.vy
    mov redBall.vy, eax
    mov ax, redBall.integerX
    xchg ax, blueBall.integerX
    mov redBall.integerX, ax
    mov ax, redBall.integerY
    xchg ax, blueBall.integerY
    mov redBall.integerY, ax

    ret
swapBalls endp
extraTurn proc
    .if si == offset [redball]
        and [gameStatus], 11111011b ; next is red
    .else
        or [gameStatus], 00000100b ; next is blue
    .endif
    ret
extraTurn endp
```

ballCollision 首先判斷球是否碰撞(球之間距離≤兩倍半徑)，若碰撞則如下圖所示，假設球的質量相同，且為完全彈性(無能量損失)，初始分別擁有v_1與v_2的速度(藍色表示)，依碰撞面畫出切線與法線，則法線方向速度將會傳遞給對方，切線方向速度將保留給自己，驗證後確定此結論符合能量守恆定律與動量守恆定律。



程式部份透過旋轉矩陣將原本以 xy 座標表示的速度變為以 nt 座標表示，交換完速度後再將其以旋轉矩陣的反矩陣變回 xy 座標，完成運算。

```
ballcollision proc
    fld [redBall.y] ; st: [y1]
    fsub [blueBall.y] ; st: [y1 - y2 = dy]
    fld [redBall.x] ; st: [x1][dy]
    fsub [blueBall.x] ; st: [x1 - x2 = dx][dy]
    fld st(1) ; st: [dy][dx][dy]
    fmul st, st ; st: [dy^2][dx][dy]
    fld st(1) ; st: [dx][dy^2][dx][dy]
    fmul st, st ; st: [dx^2][dy^2][dx][dy]
    fadd ; st: [dx^2 + dy^2 = |n|^2][dx][dy]
    fld [circleRadius] ; st: [r][|n|^2][dx][dy]
```

```
    fadd st, st ; st: [2 * r][|n|^2][dx][dy]
    fmul st, st ; st: [(2 * r)^2][|n|^2][dx][dy]
    fcomip_ 1 ; st: [|n|^2][dx][dy]
    jb @f ; (2 * r)^2 < |n|^2: no collision
    ; n = (s1 - s2) / |n| = (nx, ny), unit normal vector
    ; t = (tx, ty) = (-ny, nx), unit tangent vector
    ; |n|^2 <= (2 * r)^2: collision occur
    ; /nx tx\   /nx -ny\ , NT coordinate => XY coordinate
    ; \ny ty/ = \ny  nx/ , orthonormal basis
    ; /nx -ny\-1  /nx -ny\T  / nx ny\ , XY coordinate => NT coordinate
    ; \ny  nx/  = \ny  nx/ = \-ny nx/
        fsqrt ; st: [|n|][dx][dy]
        fdiv st(2), st ; st: [|n|][dx][dy / |n| = ny]
        fdivp st(1), st ; st: [dx / |n| = nx][ny]
        fld st(1) ; st: [ny][nx][ny]
        fchs ; st: [-ny][nx][ny]
        fld [redBall.vy] ; st: [v1y][-ny][nx][ny]
        fld [redBall.vx] ; st: [v1x][v1y][-ny][nx][ny]
        ; / nx ny\ /v1x\ = /v1n\
        ; \-ny nx/ \v1y/   \v1t/
        matrix_x_vector 3, 2, 4, 3, 0, 1 ; st: [v1n][v1t][-ny][nx][ny]
        fstp [fpuTemp] ; st: [v1t][-ny][nx][ny]
        fld [blueBall.vy] ; st: [v2y][v1t][-ny][nx][ny]
        fld [blueBall.vx] ; st: [v2x][v2y][v1t][-ny][nx][ny]
        ; / nx ny\ /v2x\ = /v2n\
        ; \-ny nx/ \v2y/   \v2t/
        matrix_x_vector 4, 3, 5, 4, 0, 1 ; st: [v2n][v2t][v1t][-ny][nx][ny]
        ; /nx -ny\ /v2n\ = v1': v1 after collision
        ; \ny  nx/ \v1t/
        matrix_x_vector 4, 5, 3, 4, 0, 2 ; st: [v1x'][v2t][v1y'][-
ny][nx][ny]
        fstp [redBall.vx] ; st: [v2t][v1y'][-ny][nx][ny]
        fld [fpuTemp] ; st: [v1n][v2t][v1y'][-ny][nx][ny]
        ; /nx -ny\ /v1n\ = v2': v2 after collision
        ; \ny  nx/ \v2t/
        matrix_x_vector 4, 5, 3, 4, 0, 1 ; st: [v2x'][v2y'][v1y'][-
ny][nx][ny]
        fstp [blueBall.vx] ; st: [v2y'][v1y'][-ny][nx][ny]
        fstp [blueBall.vy] ; st: [v1y'][-ny][nx][ny]
        fstp [redBall.vy] ; st: [-ny][nx][ny]
    @@:
    finit ; st: []

    ret
ballcollision endp
```

畫圖共分三部分，畫虛線、畫圓形與畫牆壁。首先判斷當前回合行動的球，畫出球到鼠標之間的虛線，並透過 dashedlineFlow 變數操控虛線的流動效果；接著畫出紅藍兩顆球，最後繞著螢幕一圈，畫出 0~25 區域牆壁的顏色。

```
draw proc
    mov ax, backBuffer
    mov es, ax
    ; draw line
    mov ax, 03h
    int 33h
    shr cx, 1
    mov si, [dashedlineFlow]
    dec [dashedlineFlow]
    and [dashedlineFlow], 0111b ; cycle from 0 to 7
    .if !([gameStatus] & 00000001b) ; if round not start yet
        .if [gameStatus] & 00000010b ; if player2's turn
            invoke drawLine, [blueBall.integerX], [blueBall.integerY], cx,
dx
        .else
            invoke drawLine, [redBall.integerX], [redBall.integerY], cx, dx
        .endif
    .endif
    invoke drawCircle, [redBall.integerX], [redBall.integerY], 28h
    invoke drawCircle, [blueBall.integerX], [blueBall.integerY], 37h
    ; draw wall
    lea si, wallColor
    xor di, di
    .repeat
        mov al, [si]
        mov cx, 40
        rep stosb
        inc si
    .until si == offset [wallColor + 8]
    dec di
    .repeat
        mov al, [si]
        mov cx, 40
        @@:
            stosb
            add di, 320 - 1 ; width - stosb increment
            loop @b
        inc si
    .until si == offset [wallColor + 8 + 5]
```

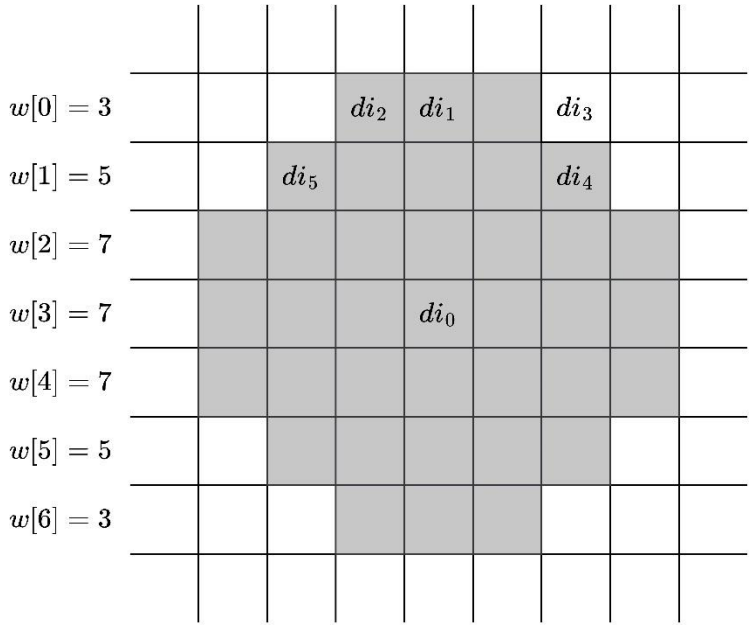
```
    sub di, 320
    std
    .repeat
        mov al, [si]
        mov cx, 40
        rep stosb
        inc si
    .until si == offset [wallColor + 8 + 5 + 8]
    inc di
    .repeat
        mov al, [si]
        mov cx, 40
        @@:
            stosb
            add di, -(320 - 1) ; width - stosb increment
            loop @b
        inc si
    .until si == offset [wallColor + 8 + 5 + 8 + 5]
    cld

    ret
draw endp
```

利用 Bresenham Line Algorithm，公式帶入起始點與終點，得出修正項，實現兩點畫一線。

```
drawLine proc, x0: word, y0: word, x1: word, y1: word ; si would be the
flashing clock
    local sx: word, sy: word, sydi: word, error: word, deltax: word,
deltay: word
    mov ax, x1
    sub ax, x0
    .if sign? ; x1 - x0 < 0, check sign flag
        neg ax
        mov deltax, ax
        mov sx, -1
    .else
        mov deltax, ax
        mov sx, 1
    .endif
    mov bx, y1
    sub bx, y0
    .if sign? ; y1 - y0 < 0
        mov deltay, bx
        mov sy, -1
        mov sydi, -320
```

```
    .else
        neg bx
        mov deltay, bx
        mov sy, 1
        mov sydi, 320
    .endif
    add ax, bx
    mov error, ax ; error = dx + dy

    mov ax, 320
    mul y0
    add ax, x0
    mov di, ax

    ; cx = x0, dx = y0, bx = e2
    mov cx, x0
    mov dx, y0
    .while 1
        mov al, [dashedlineColor + si]
        mov es:[di], al
        inc si
        and si, 0111b ; clear si, if si > 7
        .break .if cx == x1 && dx == y1
        mov bx, error
        add bx, bx
        .if sword ptr bx >= deltay
            .break .if cx == x1
            mov ax, deltay
            add error, ax
            add cx, sx
            add di, sx
        .endif
        .if sword ptr bx <= deltax
            .break .if dx == y1
            mov ax, deltax
            add error, ax
            add dx, sy
            add di, sydi
        .endif
    .endw

    ret
drawLine endp
```

本遊戲圓半徑固定為 12pixels，因此只要將圓在像素格中每一列的寬度計綠下來即可快速印出圓型。如下圖，以半徑為 3 推導，w[0]~w[6]為每一列的寬度，為了要由左印到右，由上印到下，依序要算出圖中的di_0~di_5，di 為暫存器，做為 VGA memory 的 index。首先要算出di_2，使用 rep stosb 畫完一列後會到di_3，再算出di_5，畫完一列後再移到下一列的起始位置，如此循環直到全部畫完。



$r = radius = 3$

$di_0 = 320 \times y_0 + x_0$

$di_1 = di_0 - 320 \times r$

$di_2 = di_1 - w[0] \div 2$

$di_4 = di_3 + 320$

$di_5 = di_4 - (w[0] + w[1]) \div 2$

```
drawCircle proc, centerX: word, centerY: word, color: byte
    ; di = (centerY - circleRadius) * 320 + (centerX - circleWidth[0] >> 1)
    mov ax, [centerY]
    sub ax, circleIntegerRadius
    mov bx, 320
    mul bx
    add ax, [centerX]
    mov bx, [circleWidth + 0]
    shr bx, 1
    sub ax, bx
    mov di, ax

    ; bx = si end = &circleWidth[0] + sizeof circleWidth
    lea si, [circleWidth]
    mov bx, si
    add bx, (circleIntegerRadius * 2 + 1) * 2 ; 2bytes * (2r + 1)

    mov al, [color]
```

```
    .while si != bx
        mov cx, [si]
        rep stosb

        ; di += 320 - circleWidth[i] / 2 - circleWidth[i + 1] / 2
        ;     = (640 - circleWidth[i] - circleWidth[i + 1]) >> 1
        mov cx, 640
        sub cx, [si]
        add si, 2
        sub cx, [si]
        shr cx, 1
        add di, cx
    .endw

    ret
drawCircle endp
```

若數字為 0 則印出 0 並結束，若為負號則先印出"-"，並將數字變正，接著將數字不停除以 10 可分別得到其個位、十位、百位、…，最後使用印出有屬性的字元的中斷功能將數字印出來。

```
printScore proc, score: byte, color: byte, x: byte, y: byte
    moveTextCursor x, y
    mov ah, 0eh
    xor bh, bh
    mov bl, color
    .if sbyte ptr [score] < 0
        neg [score]
        mov al, '-'
        int 10h
    .elseif sbyte ptr [score] == 0
        mov al, '0'
        int 10h
        ret
    .endif

    mov al, [score]
    xor ecx, ecx
    mov dl, 10
    .repeat
        xor ah, ah
        div dl
        or ah, 30h
```

```
        push ax
        inc ecx
    .until al == 0

    mov ah, 0eh
    @@:
        pop dx
        mov al, dh
        int 10h
        loop @b

    ret
printScore endp
```

4. 幫助畫面

本部分功能為提醒玩家遊戲玩法為何，於主畫面或遊戲中按下 H 則跳出幫助畫面，drawRectangle 函式畫出方形色塊以說明不同顏色牆壁的功能。

```
helpPage proc
    ; clear screen
    mov ax, 0a000h
    mov es, ax
    xor eax, eax
    xor di, di
    mov cx, (320 * 200) / 4
    rep stosd

    moveTextCursor 1, 1
    printString [gameIntroduction]

    invoke drawRectangle, 2 * 8, 12 * 8, 8, 8, 0fh ; white
    moveTextCursor 4, 12
    printString [normalWallMessage]
    invoke drawRectangle, 2 * 8, 14 * 8, 8, 8, 2fh ; green
    moveTextCursor 4, 14
    printString [increaseScoreMessage]
    invoke drawRectangle, 2 * 8, 16 * 8, 8, 8, 34h ; aqua
    moveTextCursor 4, 16
    printString [increase3ScoreMessage]
    invoke drawRectangle, 2 * 8, 18 * 8, 8, 8, 28h ; red
    moveTextCursor 4, 18
    printString [decreaseScoreMessage]
```

```
    invoke drawRectangle, 2 * 8, 20 * 8, 8, 8, 22h ; purple
    moveTextCursor 4, 20
    printString [swapBallsMessage]
    invoke drawRectangle, 20 * 8, 12 * 8, 8, 8, 2ch ; yellow
    moveTextCursor 22, 12
    printString [extraTurnMessage]

    mov ah, 00h
    int 16h

    ret
helpPage endp
```

本部分於幫助畫面中的對應位置畫出方形顏色圖標，以方便玩家隨時參照圖標，了解各種牆壁得效果。

```
drawRectangle proc, left: word, top: word, rectWidth: word, height: word,
color: byte
    mov ax, 320
    mul top
    add ax, left
    mov di, ax
    mov bx, 320
    sub bx, rectWidth
    mov al, color
    mov cx, height
    @@:
        push cx
        mov cx, rectWidth
        rep stosb
        add di, bx
        pop cx
        loop @b
    ret
drawRectangle endp
```

5. 其他函式

・隨機數字

使用當前 CPU 時脈做為種子，利用線性同餘生成器(LCG)取得偽亂數。

```
nextRandomNumber macro ;
https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use
    mov eax, 1664525
    mul [randomNumber]
    add eax, 1013904223
    mov [randomNumber], eax
endm
```

・矩陣乘法

$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$，用於計算球之間的碰撞。

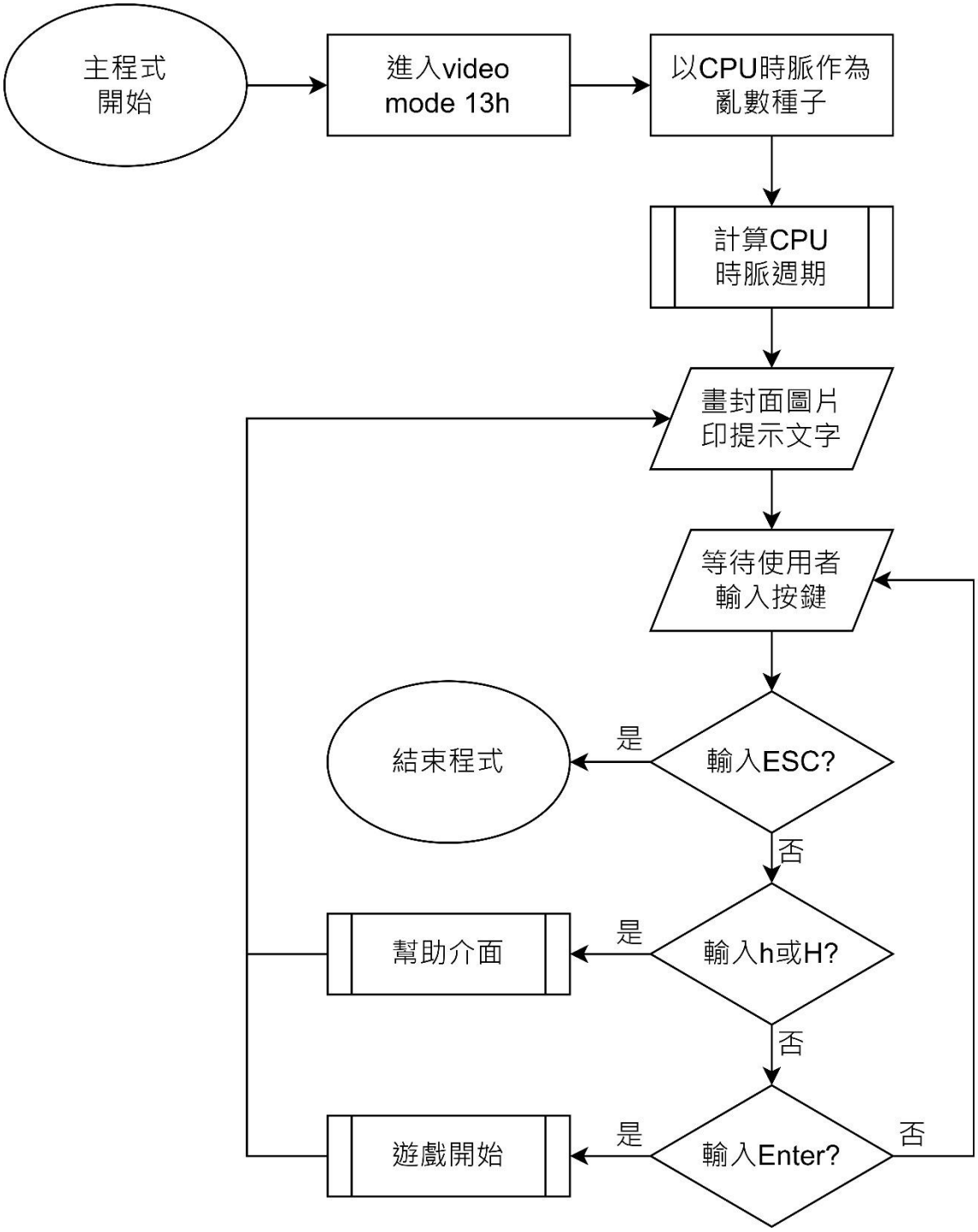
```
matrix_x_vector macro m11, m21, m12, m22, v1, v2 ; index of fpu stack, result stored in v1, v2
    ; /m11 m12\ /v1\ = /m11 * v1 + m12 * v2\
    ; \m21 m22/ \v2/   \m21 * v1 + m22 * v2/
    fld st(v1) ; st: [v1]...[v1][v2]
    fld st(v2 + 1) ; st: [v2][v1]...[v1][v2]
    fmul st, st(m12 + 2) ; st: [m12 * v2][v1]...[v1][v2]
    fxch st(v1 + 2) ; st: [v1][v1]...[m12 * v2][v2]

    fmul st, st(m11 + 2) ; st: [m11 * v1][v1]...[m12 * v2][v2]
    faddp st(v1 + 2), st ; st: [v1]...[m11 * v1 + m12 * v2][v2]

    fmul st, st(m21 + 1) ; st: [m21 * v1]...[m11 * v1 + m12 * v2][v2]
    fxch st(v2 + 1) ; st: [v2]...[m11 * v1 + m12 * v2][m21 * v1]

    fmul st, st(m22 + 1) ; st: [m22 * v2]...[m11 * v1 + m12 * m1v22][m21 * v1]
    faddp st(v2 + 1), st ; st: ...[m11 * v1 + m12 * v2][m21 * v1 + m22*v2]
endm
```

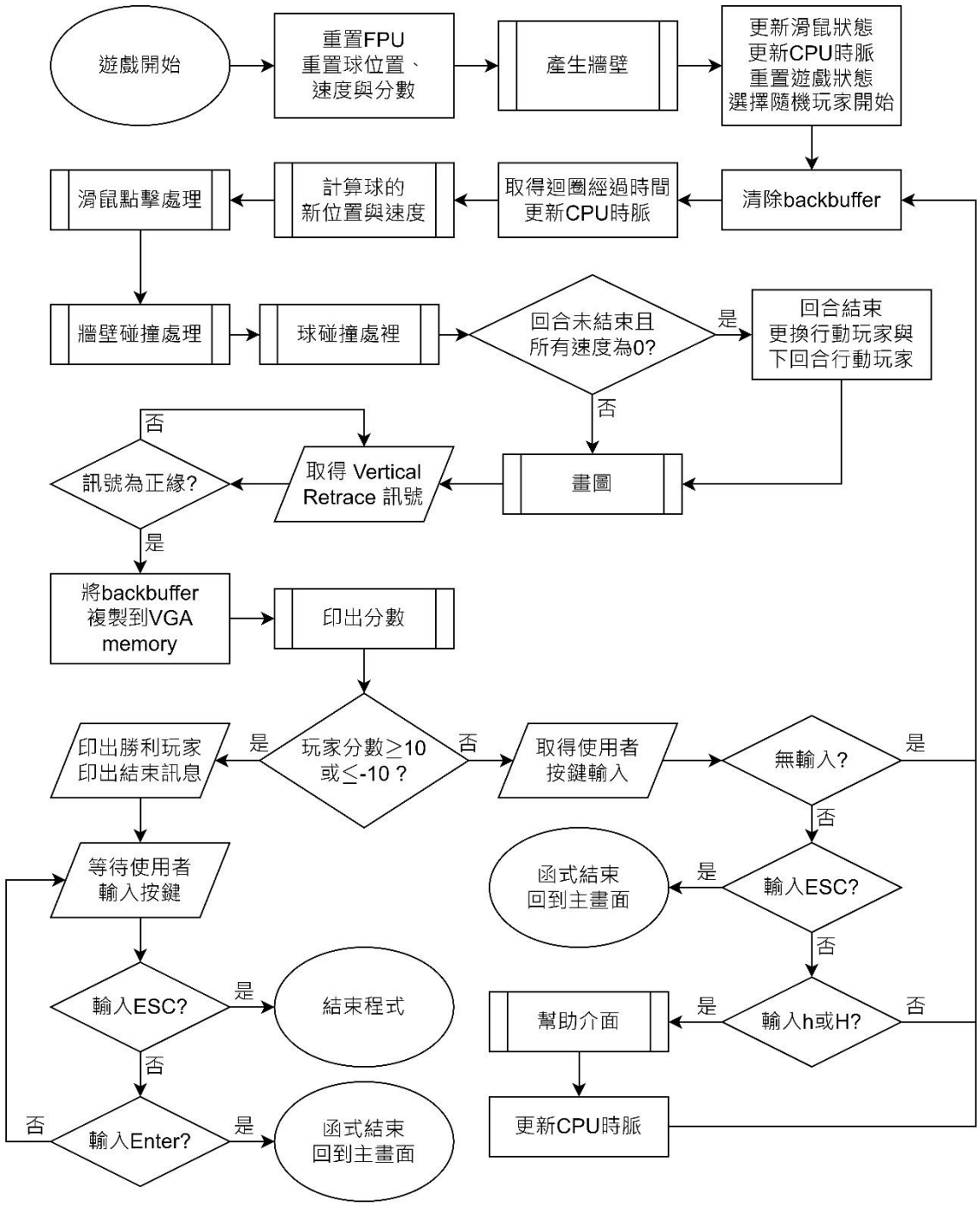
・退出遊戲

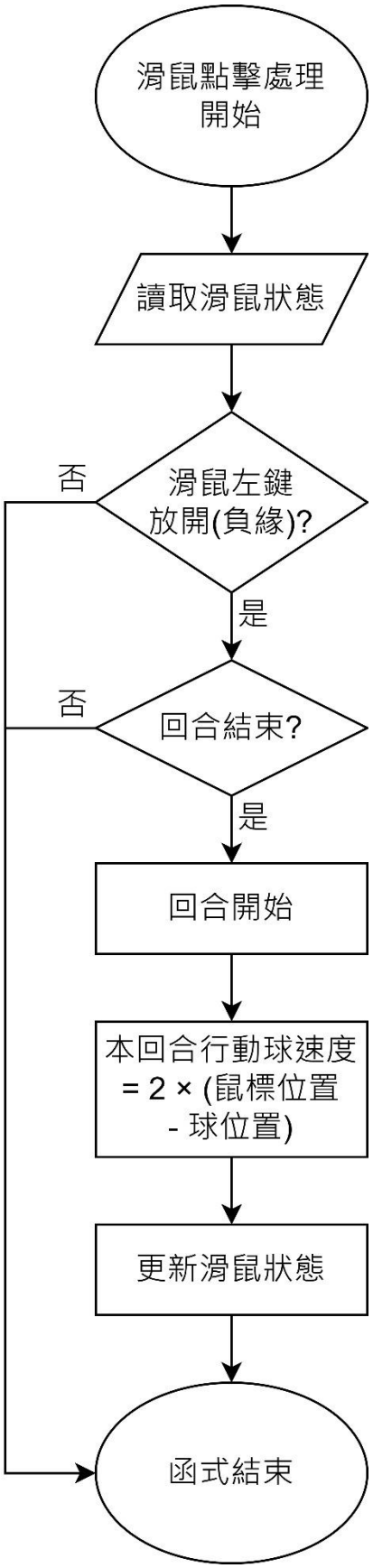
本函式功能為回到文字模式後退出程式。

```
exitGame proc
    mov ax, 03h
    int 10h
    .exit
exitGame endp
```

二、 流程圖


主程式
開始
進入video
mode 13h
以CPU時脈作為
亂數種子
計算CPU
時脈週期
畫封面圖片
印提示文字
等待使用者
輸入按鍵
輸入ESC?
是
結束程式
否
輸入h或H?
是
幫助介面
否
輸入Enter?
是
遊戲開始
否

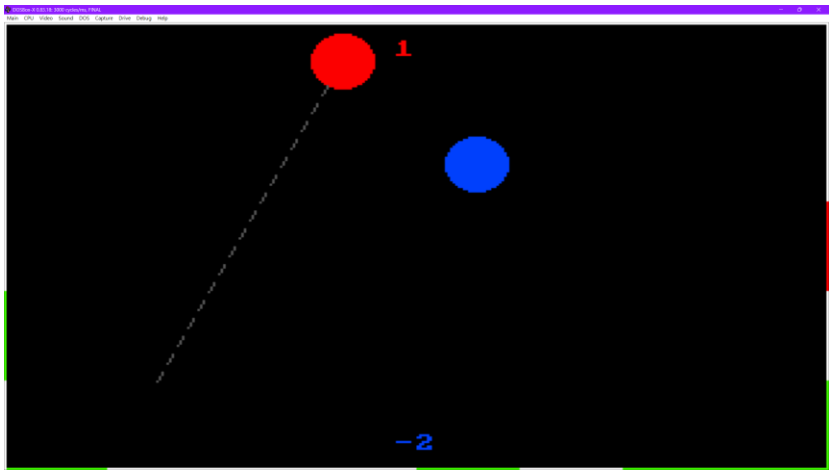

遊戲開始
重置FPU
重置球位置、
速度與分數
產生牆壁
更新滑鼠狀態
更新CPU時脈
重置遊戲狀態
選擇隨機玩家開始
清除backbuffer
取得迴圈經過時間
更新CPU時脈
計算球的
新位置與速度
滑鼠點擊處理
牆壁碰撞處理
球碰撞處裡
回合未結束且
所有速度為0?
是
回合結束
更換行動玩家與
下回合行動玩家
否
畫圖
取得 Vertical
Retrace 訊號
否
訊號為正緣?
是
將backbuffer
複製到VGA
memory
印出分數
玩家分數≥10
或≤-10？
是
否
印出勝利玩家
印出結束訊息
取得使用者
按鍵輸入
無輸入?
是
否
等待使用者
輸入按鍵
函式結束
回到主畫面
是
輸入ESC?
否
輸入ESC?
是
結束程式
否
幫助介面
是
輸入h或H?
否
否
輸入Enter?
是
函式結束
回到主畫面
更新CPU時脈


滑鼠點擊處理
開始
讀取滑鼠狀態
滑鼠左鍵
放開(負緣)?
否
是
回合結束?
否
是
回合開始
本回合行動球速度
= 2 × (鼠標位置
- 球位置)
更新滑鼠狀態
函式結束

三、 實習結果 (遊戲運作畫面)

・遊戲開始畫面

・遊戲過程



・遊戲結束

四、 心得

- B11107051李品翰：

  一開始決定題目時，只有想要做一個「可玩性高的遊戲」這樣模糊的想法，經過幾個禮拜的努力，我覺得這個想法真的實現了，這個遊戲需要玩家一定的技術、戰略，且具有一些隨機性，甚至可以產生無數張不同的地圖，可惜受時間限制，若是能加入更多牆壁效果的話這個遊戲就更完美了。

  為了做出這個遊戲，在這短短幾個禮拜中，我學習了無數的新知識。最一開始，當大家都透過重新進入圖形模式清除畫面時，我在網路上看到一個直接存取VGA memory來清除畫面的程式，打開了我新世界的大門，在了解mode 12h的四個色平面與mode 13h 1pixel = 1byte的友善操作方式後，我就決定使用mode 13h做為期末專題的模式了。後來發現了閃爍問題，了解到page flipping與double buffering等做法，然而mode 13h只有一頁，因此最後只能選擇相對較慢的後者。後來又發現了畫面撕裂的問題，查詢後了解到垂直同步的運作原理，沒想到一個經常在遊戲中聽到、感覺很專業的技術，有天也能夠用在自己的遊戲中。

  在遊戲設計前期，球的位置與速度運算都是用整數，有了加速度後，進行乘法運算會使的數字變極大，除法還要避免精度過低，最後變得礙手礙腳，難以計算，因此下定決心開始學習FPU指令。學習完畢修改程式後，效果十分顯著，後面加入摩擦力、球碰撞等功能也變得容易許多。不過FPU的運算相對整數來說真的十分麻煩，總感覺以後看見浮點數都會不由得肅然起敬了。

一開始使用backbuffer實現double buffering時，畫面上總是會出現一小排一直跳動的顏色，百思不得其解的我們開始猜測各種可能，或許是segment重疊？或許是backbuffer太大？或許是沒有初始化？最終透過code view觀察每一個segment的值，並計算各種資料的offset與實際位址，發現那排跳動的顏色其實是stack裡的資料，查詢後才知道.model預設會將ds與ss設定為相同的值，而backbuffer足足有64000bytes之大，接在data segment後面肯定會與stack重疊，當時正好沒有需要stack的功能，因此程式也沒有出現其他的問題，最後將.model加上farstack選項才順利解決問題。

遊戲設計後期，有朋友想要放封面圖片到程式裡面，但網路上找不到已經做好的功能，於是我用javascript寫了一個程式(連結)，能夠讀取圖片並轉換成masm裡資料宣告格式的文字，效果意外不錯，於是我們的遊戲裡也加入了封面圖片，遊戲完成度感覺瞬間高了起來。

這次的遊戲設計對我來說是各種第一次，第一次做圖形遊戲、第一次寫一個1000多行的大程式、第一次和別人一起用github合作，雖然最後幾天除了吃飯睡覺以外，幾乎都在寫程式，但看到自己寫的遊戲順利動起來的那刻，總感覺，一切都太值得了！

- 組員二(B11107053陳建安):
  相較於期中作業，完成期末作業讓我學到了很多，但更多的是心理上的震撼，學期初第一次接觸到組合語言，經過一學期，竟然有個期末作業是做一個Dos遊戲，在最後一堂實習課，我心中只有滿滿的訝異，當時的我頂多會印個三角形或當個記憶體搬運工之類的，根本無法相信我們可以自己完成遊戲，還一度懷疑是我少學了甚麼，亦或是助教少教甚麼，直到完成遊戲的現在，我只會說助教已經盡職了，今天完成遊戲所用的資料太龐大了，而且來源也很多元，課堂上的教學不可能支撐我們完成遊戲，反而我們自己尋找需要的資料還比較實際，這部分我的組員就是個老手，實現遊戲內物理量運算必須要使用的FPU浮點運算，是個很麻煩的運算工具，而我的組員就展現了超群的學習能力，了解了FPU的操作，才能將這款遊戲實現出來。
  實現這款遊戲，了解FPU只是基礎，遊戲的內容雖然很單純，只有兩顆球對撞，或者撞牆壁，但遊戲的動畫卻很麻煩，以畫線為例，在現實中畫線可能是件簡單的事，但在組合語言裡就不一樣了，我只知道起始點和終點，卻不知道路徑怎麼走，在現實有視覺可以做導引，但在組合語言裡沒有，所以我引用了BresenhamLine的演算法，BresenhamLine的概念就如同我們在遠處射擊，沒有命中目標，但我約略知道偏差多少，可以修正偏差，畫線時，則檢查斜率偏差，以此修正路徑，有趣的是，在wiki尋找pseudocode時，發現了不同語言的wiki會出現雖然不同版本但功能相同的pseudocode，在畫完線之後，我們就可以把線的長度當作力量、方向當作彈道發射球，當球發射出去，接著就是物理運算的世界了，想實現物理運算，除了FPU浮點運算，還需要一個單位時間，做物理量換算，這部

份我們也考慮過8254與rdtsc，礙於8254產生的是個脈波，更像是輸出給外部的控制裝置，rdtsc產生時脈的功能就相對符合我們的需求，我們也就不計較rdtsc可以顯示1/100秒精度卻是5/100秒的問題，解決完單位時間，接著就只需要用FPU實現基本的向量運算，就可以解決求發射後的碰撞與摩擦力問題，接著會發現球還是無法順暢地撞來撞去，因為畫面會出現撕裂和閃頻的問題，此時，我們找到了Vsync垂直同步，利用卡迴圈的方式，避開列印的週期防止畫面撕裂，還有Backbuffer，先將欲輸出的內容，寫到Backbuffer的這個區段內，在一次性地將記憶體一次輸出給VGA 記憶體的區段，避免閃頻的狀況。

從11/21號算起至截止日12/11，總共三週的時間，我覺得這款遊戲可以做到球體正常動作，有簡單的封面，以及完整的遊戲流程，就已經相當足夠了，相較於期中一人做一份作業，再合併，我覺得期末作業更能讓我體會到未來職場分工作業的氛圍。