Manual tecnico

Strucs:

```
BLOCK struct 2
    points dw 0;signed POINTS EN 0 SIGNIFICA NULL
    blockHitbox HITBOX { 0, 0, 15, 15 }
    colors db GRAY, BLACK
    tickAccumulator dw 0;para que cada n ticks baje
BLOCK ends
```

Describe los datos unicos de cada bloque en la pantalla, si puntuacion es 0 significa que es un bloque null y no se dibuja ni actualiza

```
HITBOX struct 2
    x0 dw 0
    y0 dw 0
    _width dw 0
    _height dw 0
HITBOX ends
```

Rectangulo, consisten de un par de coordenadas x y (esquina superior izquierda), ancho y altura. Se utilizan para calcular las colisiones de los objetos en el juego

```
;null cuando _name es 0
LEVEL struct 2
    _name db 8 dup(" ")
    time dw 10
    time_enemy dw 3
    time_prize dw 2
    points_enemy dw -5
    points_prize dw 5
    color db RED
    ;1 byte padding
LEVEL ends
```

Contiene los datos de cada nivel, se utiliza el archivo de entrada para hacer cada nivel

```
USR struct 2
    _name db 8 dup(0)
    password db 4 dup(0) ;Este no tiene que ser nul
    _type dw 0
    points dw 0
    time dw 0
    level db 8 dup(" ")
USR ends
```

Contiene todos los datos que puede tener un usuario en el sistema. Se guarda el struct literal en un archivo binario que se utiliza para que los usuarios ingresados persistan despues de terminar la ejecucion del programa.

Gameloop:

```
GAME_LOOP:
;beg checkPausa:
call mydirectConsoleInput
cmp al, 1bh ;ASCII: ESC
je ENTER_PAUSE
;end checkPausa

CONTINUE_GAME:
call playerUpdate;al todavia tienen el resultado de mydi
call gameUpdateTime
call blocksTryUpdate
jnz EXIT_GAME
call spawnerUpdate
call levelManagerTryUpdate
jnz EXIT_GAME

jmp GAME_LOOP
```

Primero chequeamos is el usuario presiono pausa, de no ser asi pasamos el input del usuario a playerUpdate atravez del registro al

Luego actualizamos todos los gameObjects, dependiendo del resultado de su actualizacion los dibujamos borramos o redibujamos en la pantalla.

```
call playerUpdate;al todavia
```

Movemos el jugador si el usuario presiono izquierda o derecha, de ser asi, borramos el dibujo del carro anterior y lo redibujamos con las coordenadas actualizadas

```
call gameUpdateTime
```

Actualizamos el tiempo transcurrido y actualizamos la cadena que se muestra en la parte superior derecha.

```
call blocksTryUpdate
jnz EXIT_GAME
```

Actualizamos todos los bloques que estan actualmente activos en el juego. Si colisionan con el hitbox del jugador se suman o restan puntos dependiendo del tipo de bloque. Si el puntaje llega a 0 se limpia la bandera zero y se termina el juego.

```
call spawnerUpdate
```

Verificamos si ha pasado suficiente tiempo para que aparesca otro bloque enemigo o amigo, de ser asi creamos ese nuevo bloque y ponemos sus coordenadas justo afuera de la pantalla en una coordenada x aleatoria.

```
call levelManagerTryUpdate
jnz EXIT_GAME
```

Verificamos si es tiempo de pasar al siguiente nivel, si ya no hay niveles entonces se pone en 0 la bandera zero y se termina el juego.

Generador de numeros aleatorios:

Se utiliza para poner los bloques nuevos en posiciones aleatorias:

Generador de semillas:

Genera una semilla laetoria usando el numero de ciclos del procesador desde media noche. Solo es necesario correr este procedimiento al principio del programa

```
generateSeed proc
    mov ah, 00h   ; interrupt to get system timer in cx:dx
    int 1ah
    mov mathRandomSeed, dx
    ret
generateSeed endp
```

Implementa un generador lineal congruencial

```
getRandomWord proc
    mov ax, 25173
    mul mathRandomSeed
    add ax, 13849
    mov mathRandomSeed, ax
    ret
getRandomWord endp
```

Almacenamiento de bloques (enemigos y premios)

```
blocks BLOCK 40 dup({ })

blocksCount dw 0
PBLOCK TYPEDEF PTR BLOCK
firstFreeBlock PBLOCK blocks[0]
```

BubbleSort:

```
LOOP_I:
    mov bx, cx

    mov si, ax ;si=usrs[0]
    mov di, ax
    add di, S_USR ;di = usrs[1]

    LOOP_J:
        mov dx, [si].points

        cmp dx, [di].points
        ja MAYOR
        jb MENOR
        jmp CONTINUE_J

        MAYOR:
        cmp direccion, ASCENDENTE
        je SWAP
        jmp CONTINUE_J

        MENOR:
        cmp direccion, DESCENDENTE
        je SWAP
        jmp CONTINUE_J

        SWAP:
        mSwapUsrSIDI
        mGraphUsrsPoints

    CONTINUE_J:
        add si, S_USR ;si = usrs[curr + 1]
        add di, S_USR ;di = usrs[curr + 1]

    dec bx
    jnz LOOP_J

loop LOOP_I
```

shellSort:

```
LOOP_GAP:
    cmp cx, 0;gap > 0
    je BREAK_GAP
    ;Nota: metemos a ax el valor gap * S_USR + offset usrs dos
    mov ax, cx ;ax = gap
    mov dx, S_USR
    mul dx ; ax = gap * S_USR
    add ax, offset usrs; ax = gap * S_USR + offset usrs
    mov bx, ax;i = gap * S_USR + offset usrs
    LOOP_I:
        cmp bx, offset usrs + S_USR * L_USRS;Condicion del loop
        jae BREAK_I
        cmp [bx]._name[0], 0;Condicion del loop i < N
        je BREAK_I

        mCopyUsr bx, offset tempUsr;temp = arr[i]

        mov di, bx;j = i
        LOOP_J:
            mov ax, cx ;ax = gap
            mov dx, S_USR
            mul dx ; ax = gap * S_USR
            add ax, offset usrs; ax = gap * S_USR + offset usrs
            cmp di, ax ; j >= gap * S_USR + offset usrs
            jb BREAK_J
            sub ax, offset usrs;ax = gap * S_USR
            ;&& arr[j - gap] > temp
            mov si, di
            sub si, ax;si: j - gap
            mov dx, [si].points
            cmp dx, tempUsr.points
            ja MAYOR
            jb MENOR
            jmp BREAK_J
```

```
            MAYOR:
            cmp direccion, ASCENDENTE
            je COPY
            jmp BREAK_J

            MENOR:
            cmp direccion, DESCENDENTE
            je COPY
            jmp BREAK_J

            COPY:
            mCopyUsrSIDI;si: j - gap
                       ;di: j
            mGraphUsrsPoints

        CONTINUE_J:
            sub di, ax ;j -= gap * S_USR
            jmp LOOP_J
        BREAK_J:
            mov si, offset tempUsr
            mCopyUsrSIDI;si: temp
                       ;di: j
            ;arr[j] = temp
            mGraphUsrsPoints

    CONTINUE_I:
        add bx, S_USR
        jmp LOOP_I
    BREAK_I:

CONTINUE_GAP:
    mov ax, cx
    mov dl, 2
    div dl ;ax = gap / 2
    movzx cx, al ;gap /= 2
    jmp LOOP_GAP
BREAK_GAP:
```

QuickSort:

```
;params:   [bp + 4]: low
;          [bp + 6]: high
quickSortPoints proc
    push bp
    mov bp, sp

    mov si, [bp + 4]
    mov di, [bp + 6]

    cmp si, di
    jae RETURN
    mPartitionPoints [bp + 4], [bp + 6];ax = partitioning index

    ;quickSort(low, pi - 1);  // Before pi
    push ax
    sub ax, S_USR ;ax = pi - 1
    push ax
    mov dx, [bp + 4]
    push dx
    call quickSortPoints
    pop ax

    ;quickSort(pi + 1, high); // After pi
    mov dx, [bp + 6]
    push dx
    add ax, S_USR;pi + 1
    push ax
    call quickSortPoints

RETURN:
    mov sp, bp;hella safe
    pop bp
    ret 4
```

```
LOOP_J:
    cmp di, bx;Condicion del loop j < high;
    jae BREAK_J

    cmp [di].points, ax
    ja MAYOR
    jb MENOR
    jmp CONTINUE_J

    MAYOR:
    cmp direccion, DESCENDENTE
    je SWAP
    jmp CONTINUE_J

    MENOR:
    cmp direccion, ASCENDENTE
    je SWAP
    jmp CONTINUE_J

    SWAP:
    add si, S_USR;i++
    cmp si, di
    je CONTINUE_J
    mSwapUsrSIDI;swap si y di
    cmp shouldQuickSortDraw, 0
    je CONTINUE_J
    mGraphUsrsPoints
CONTINUE_J:
add di, S_USR;j++
jmp LOOP_J
```