Video Game Project

ANDRANGO SANCHEZ ANGELA NAYELI

I. Introduction

A. Goal

The main objective of this project was to design, develop, and implement a version of the classic video game Space Invaders using low-level programming. Through this project, we aimed to understand fundamental concepts of computer architecture, such as the use of registers, memory management, interrupt handling, and hardware control. The idea was to implement these theoretical concepts in a real programming environment.

B. Brief Description of the Game

The game developed was an adapted version of Space Invaders, which in this case was called Fisher Invaders. This game consists of a space ship controlled by the user that moves from left to right while shooting projectiles at enemies at the top of the screen. The player has a specific number of lives and the game ends when they lose them all. It also includes a start screen with a selection for the game difficulty and an end-of-game screen.

C. Programming language and environment

The video game was programmed in x86 assembly, designed for the 8086 microprocessor. Development took place in a DOS environment using EMU8086 and DOSBox. BIOS and DOS interrupts were used to handle keyboard input, display graphics, and control timing, providing an execution environment similar to real hardware.

II. Game Modifications

A. Visual Changes (Sprites, Colors, Themes)

- Enemies are not monsters but green fish with white eyes
- The player's ship is a magenta submarine
- Enemy and submarine bullets are different colors so they can be distinguished
- The ship flashes when it is hit
- There are two shields that decrease in size when damaged
- Mini-submarines are drawn in the upper left corner of the screen as life indicators

B. Gameplay Changes (Movement, Speed, Difficulty

- Intelligent enemy movement: synchronized group movement, configurable speed depending on game difficulty, and game reset when enemies reach the end of the screen
- Advanced shooting system: the player has double shots, enemies have random shots from anywhere, and there is a limit on simultaneous projectiles
- Difficulty mechanics: The game is divided into easy mode: slow enemies and shots with a longer delay, and hard mode: fast enemies and frequent shots, in addition to enemies having a variable level

C. New Features Added

- Complete menu system: there is a main menu with difficulty selection, integrated instructions, and Game Over and Victory screens
- Progression system: scoring with different values for each type of enemy, 3 initial lives with visual representation, and shields that protect the player
- Sound effects: Sound is implemented for player shots, explosions, and victory
- Invasion detection: Game Over if enemies reach the ship's height, with verification by rows of enemies

III. Technical Implementation

A. Program Structure

```
; Estructura de segmentos
STACK SEGMENT PARA STACK      ; Segmento de pila
DATA SEGMENT PARA 'DATA'      ; Variables del juego
CODE SEGMENT PARA 'CODE'      ; Cdigo ejecutable

; Estructura del main loop
MAIN PROC FAR
    ; Inicializacin
    ; Men principal
    ; Loop principal del juego:
    ;   1. Verificar tiempo
    ;   2. Actualizar lgica
    ;   3. Dibujar elementos
    ;   4. Verificar condiciones de fin
```

Procedures organized by functionality:
Movement:
`MOVE_ENEMIES`,
`MOVE_NAVE`,
`MOVE_DISPAROS`
Collisions:
`CHECK_COLLISIONS`,
`CHECK_ENEMY_BULLET_COLLISION`
Drawing:

DRAW_ENEMIES,
DRAW_NAVE,
DRAW_SHIELDS
Menus:
DRAW_MENU,
WAIT_MENU_INPUT
Sound:
PLAY_SHOOT_SOUND,
PLAY_EXPLOSION_SOUND

B. Input Handling

```
; Lectura de teclado sin espera
MOV AH, 01h
INT 16h
JNZ KEY_PRESSED

; Lectura de tecla especfica
MOV AH, 00h
INT 16h

; Teclas implementadas:
; - A/D: Movimiento izquierda/derecha
; - G: Disparar
; - Flechas: Navegar men
; - ESC: Salir
```

Implemented keys:
A/D: Move left/right
G: Shoot
Arrows: Navigate menu
ESC: Exit

C. Graphics Handling

```
; Modo video 13h (320x200, 256 colores)
MOV AH, 00h
MOV AL, 13h
INT 10h

; Dibujar pxel individual
DRAW_PIXEL_COLOR PROC NEAR
    MOV AH, 0Ch          ; Funcin escribir pxel
    MOV BH, 00h          ; Pgina 0
    INT 10h
    RET

; Dibujar sprites mediante composicin de pxeles
DRAW_LARGE_FISH PROC NEAR
    ; Dibuja pez lnea por lnea
    ; Usa bucles anidados para formas rectangulares
    ; Aplica colores especficos por elemento
```

Graphics explanation:
Video Mode 13h is used for 256-color graphics . Sprites are drawn using pixel-by-pixel composition with nested loops.

D. Collision Detection Logic

```
; Deteccin rectngulo-rectngulo (AABB)
CHECK_HIT PROC NEAR
    ; Verifica colisin punto (DI,SI) con enemigo (
    CX,DX)
    ; Compara lmites en ambos ejes

; Sistema de hits por enemigo
ENEMY_HITS DB 23 DUP(0)        ; Contador de
    impactos por enemigo
ENEMY_MAX_HITS DB ...          ; Resistencia mxima
    por enemigo

; Verificacin por filas con clculos de posicin
CHECK_DISPARO_VS_ENEMIES PROC NEAR
    ; Calcula posicin exacta de cada enemigo
    ; Considera offset por fila y centrado
```

Collision logic explanation:
Rectangle–rectangle (AABB) detection is used. Each enemy stores its individual hit counter. Position checks consider row offsets and centering.

E. Challenges Faced

Problems with Movement or Timing

1) Synchronization of enemies:
   Problem: Irregular movement when changing direction
   Solution: Use variable ENEMY_DIRECTION and uniform downward movement
2) Enemy shooting timing:
   Problem: Shots too frequent or clustered
   Solution: Implement ENEMY_SHOOT_TIMER with configurable delays
3) Smooth ship movement:
   Problem: Slow response to keyboard
   Solution: Check keys every frame without waiting

Bugs Related to Screen Drawing

1) Sprite flickering:
   Problem: Flicker during fast updates
   Solution: Clear the screen once per frame
2) Incorrect enemy alignment:
   Problem: Enemies not aligned properly in rows
   Solution: Precise calculations using ROW_SPACING and offsets
3) Shield drawing with damage:
   Problem: Incorrect width after multiple hits
   Solution: Implement UPDATE_SHIELD_WIDTH with proportional calculation

Memory or Register Management Issues

1) Register overflow:
   Problem: Multiplication results exceed 16-bit range
   Solution: Use IMUL and DIV carefully, verify ranges
2) Word array handling:

Problem: Incorrect access to `DISPARO_X[SI]` (word vs byte)
Solution: Use `SHL SI, 1` to convert index to offset

3) Register preservation:
Problem: Procedures modify registers without saving
Solution: Implement consistent `PUSH`/`POP` conventions

Debugging and Testing Difficulties

1) Assembly debugging:
Difficulty: Lacks modern debugging tools
Solution: Use "tracing" techniques by printing characters on screen
2) Collision testing:
Difficulty: Hard-to-reproduce edge cases
Solution: Implement exhaustive checks on all boundaries
3) Emulator compatibility:
Difficulty: Different behavior in DOSBox vs EMU8086
Solution: Standardize on well-documented BIOS interrupts

6. Conclusion

F. What Was Learned from the Project

Low-level programming:

- Direct hardware control through BIOS interrupts
- Optimization of limited resources (memory, CPU)
- Precise control of timing and synchronization

Classic game development:

- Game loop architecture
- Efficient collision systems
- State management and transitions

Software engineering in assembly:

- Modular organization using procedures
- Calling conventions and register preservation
- Handling of complex data structures

G. Possible Future Improvements

Graphical enhancements:

- Sprite animation (alternate frames)
- Particle effects for explosions
- Scrolling or parallax backgrounds

Extended gameplay:

- Multiple levels with different enemy patterns
- Power-ups and special weapons
- Boss enemies with unique behaviors

Advanced techniques:

- More complex PC Speaker sound effects
- Protected mode implementation for increased memory
- Optimization using 286/386 instructions

Modern features:

- Persistent high-score system
- Gameplay recording and playback
- Joystick or gamepad support

H. Final Reflection

This project demonstrates that even with the limitations of x86 assembly and older hardware, it is entirely possible to create complete and engaging gameplay experiences. The key lies in careful design, constant optimization, and a deep understanding of the underlying hardware.