MINI PROJECT

NAME: Ayan Guchhait

ROLL NO: 16900123028

SUBJECT: Operating System Lab (PCC-CS592)

Project 1: Snake Game in Shell Script Problem Statement: Design and implement a Snake Game using a Shell Script. The game should provide an interactive terminal-based user interface where the snake can move in different directions. The snake will grow in length upon eating food, and the game will end when the snake collides with itself or the boundary. The script should handle user input, game logic, and screen updates within the shell environment.

Algorithm & Implementation Steps:

1. Initialize the game area (grid) and the snake's starting position.
2. Place the initial food at a random location within the grid.
3. Continuously read user input (W/A/S/D for directions).
4. Update the snake's position based on its current direction.
5. Check for collisions with boundaries and self.
6. Check for food consumption; grow and increase score if eaten.
7. Redraw the grid with each move.
8. End the game when a collision occurs or upon user quit.

Key Features:

- Terminal-based game interface.
- Keyboard controls (w/a/s/d for movement).
- Growing snake logic and random food placement.
- Handles collisions against walls.

Controls: W (Up), A (Left), S (Down), D (Right), Q (Quit)

Sample Code:

```
GAME_WIDTH=20
GAME_HEIGHT=15
INITIAL_SNAKE_LENGTH=3
```

```
SNAKE_HEAD="○"
SNAKE_BODY="●"
FOOD="□"
EMPTY=" "
WALL="▒"
snake=()
snake_direction="right"
food_x=0
food_y=0
score=0
game_over=0
init_game() {
    stty -echo -icanon time 0 min 0
    local start_x=$((GAME_WIDTH / 2))
    local start_y=$((GAME_HEIGHT / 2))
   for ((i=0; i<INITIAL_SNAKE_LENGTH; i++)); do
        snake+=("$((start_x - i)):$start_y")
    done
   place_food
    draw_screen
}
place_food() {
    while true; do
        food_x=$((RANDOM % (GAME_WIDTH - 2) + 1))
        food_y=$((RANDOM % (GAME_HEIGHT - 2) + 1))

        local valid_position=1
```

```
        for segment in "${snake[@]}"; do
            local seg_x=${segment%:*}
            local seg_y=${segment#*:}
            if [[ $seg_x -eq $food_x && $seg_y -eq $food_y ]]; then
                valid_position=0
                break
            fi
        done

        if [[ $valid_position -eq 1 ]]; then
            break
        fi
    done
}
draw_screen() {
    clear
        echo "╔═══════════════════════════════════════════════════════════╗"
    echo "║          SHELL SNAKE          ║"
    echo "║    Score: $score              ║"
    echo "╚═══════════════════════════════════════════════════════════╝"
    echo "Controls: W (Up), A (Left), S (Down), D (Right), Q (Quit)"
    echo ""
     printf "╔"
    for ((i=0; i<GAME_WIDTH; i++)); do
        printf "═"
    done
    printf "╗\n"
```

```
    for ((y=0; y<GAME_HEIGHT; y++)); do
    printf "║"
    for ((x=0; x<GAME_WIDTH; x++)); do
        local cell_content=$EMPTY
        local is_snake_segment=0
        for segment in "${snake[@]}"; do
            local seg_x=${segment%:*}
            local seg_y=${segment#*:}
            if [[ $x -eq $seg_x && $y -eq $seg_y ]]; then
                if [[ $x -eq ${snake[0]%:} && $y -eq ${snake[0]#:} ]]; then
                    cell_content=$SNAKE_HEAD
                else
                    cell_content=$SNAKE_BODY
                fi
                is_snake_segment=1
                break
            fi
        done
        if [[ $is_snake_segment -eq 0 && $x -eq $food_x && $y -eq $food_y ]];
       then
            cell_content=$FOOD
        fi


        printf "%s" "$cell_content"
    done
    printf "║\n"
done
 printf "╚"
```

```
    for ((i=0; i<GAME_WIDTH; i++)); do
        printf "═"
    done
    printf "╝\n"
}
update_game() {
    local head_x=${snake[0]%:*}
    local head_y=${snake[0]#*:}
   case $snake_direction in
        "up")    new_head_y=$((head_y - 1)); new_head_x=$head_x ;;
        "down")  new_head_y=$((head_y + 1)); new_head_x=$head_x ;;
        "left")  new_head_x=$((head_x - 1)); new_head_y=$head_y ;;
        "right") new_head_x=$((head_x + 1)); new_head_y=$head_y ;;
    esac
    if [[ $new_head_x -lt 0 || $new_head_x -ge $GAME_WIDTH ||
         $new_head_y -lt 0 || $new_head_y -ge $GAME_HEIGHT ]]; then
        game_over=1
        return
    fi
        for ((i=1; i<${#snake[@]}; i++)); do
        local seg_x=${snake[i]%:*}
        local seg_y=${snake[i]#*:}
        if [[ $new_head_x -eq $seg_x && $new_head_y -eq $seg_y ]]; then
            game_over=1
            return
        fi
    done
```

```
    local new_head="$new_head_x:$new_head_y"
     if [[ $new_head_x -eq $food_x && $new_head_y -eq $food_y ]]; then
     snake=("$new_head" "${snake[@]}")
     score=$((score + 10))
      place_food
    else
       snake=("$new_head" "${snake[@]}")
       unset 'snake[${#snake[@]}-1]'
    fi
}
handle_input() {
    local input
    IFS= read -rs -n1 -t 0.1 input

    case ${input,,} in
       "w") if [[ $snake_direction != "down" ]]; then snake_direction="up"; fi ;;
       "s") if [[ $snake_direction != "up" ]]; then snake_direction="down"; fi ;;
       "a") if [[ $snake_direction != "right" ]]; then snake_direction="left"; fi ;;
       "d") if [[ $snake_direction != "left" ]]; then snake_direction="right"; fi ;;
       "q") game_over=1 ;;
    esac
}
show_game_over() {
    clear
    echo "╔═══════════════════════════════════════════════════╗"
    echo "║         GAME OVER!          ║"
    echo "║                             ║"
```

```
    echo "╠      Final Score: $score        ╣"
    echo "╠                                  ╣"
    echo "╠     Press any key to exit    ╣"
    echo "╚═══════════════════════════════════════════════════════╝"
    read -n1
}
cleanup() {
    stty echo icanon
    clear
}
main() {
    trap cleanup EXIT
   echo "Starting Snake Game..."
    sleep 1
   init_game
   while [[ $game_over -eq 0 ]]; do
        handle_input
        update_game
           if [[ $game_over -eq 0 ]]; then
            draw_screen
            sleep 0
fi
done
  show_game_over
}
```

Project 2: Running Clock in Shell Script Problem Statement: Write an executable bash script (clock.sh) that continuously prints the current time at the center of the terminal in ASCII format and updates the display at regular

intervals (every second or every half a second). The current time and date should be obtained using the Unix command date.

Algorithm & Implementation :

Steps:

1. Retrieve and format current date/time using the date command.

2. Convert time digits into ASCII-art representations.

3. Center the clock display within the terminal.

4. Update the clock display at regular intervals by clearing and redrawing the output.

Key Features:

● Live clock display in ASCII art.

● Centered date and time, refreshing automatically.

● Uses standard Unix utilities for time retrieval.

Sample Code:

```
#!/bin/bash
 ascii_digit() {
case $1 in
0) echo -e " __ \n| |\n|__|";;
1) echo -e " \n |\n |";;
2) echo -e " __ \n __|\n|__ ";;
3) echo -e " __ \n __|\n __|";;
4) echo -e " \n|__|\n |";;
5) echo -e " __ \n|__ \n __|";;
 6) echo -e " __ \n|__ \n|__|";;
7) echo -e " __ \n |\n |";;
8) echo -e " __ \n|__|\n|__|";;
 9) echo -e " __ \n|__|\n __|";;
 :) echo -e " \n o \n o ";;
```

```
esac
}
draw_clock() {
local time_now=$(date +"%H:%M:%S")
    local date_now=$(date +"%Y-%m-%d")
   clear
    echo
    echo
    local cols=$(tput cols)
    local ascii_lines=("" "" "")

    for (( i=0; i<${#time_now}; i++ )); do
        local char="${time_now:$i:1}"
        local digit_ascii=$(simple_ascii_digit "$char")
        IFS=$'\n' read -d '' -r -a digit_lines <<< "$digit_ascii"

        for (( j=0; j<3; j++ )); do
            ascii_lines[j]="${ascii_lines[j]} ${digit_lines[j]}"
        done
    done

    for line in "${ascii_lines[@]}"; do
        printf "%*s\n" $(( (cols + ${#line}) / 2 )) "$line"
    done

    printf "\n%*s\n" $(( (cols + ${#date_now}) / 2 )) "$date_now"
    printf "\n%*s\n" $(( (cols + 20) / 2 )) "Press Ctrl+C to exit"
```

```
}
trap 'clear; echo "Clock stopped.";
exit 0' INT
while true; do
    draw _clock
    sleep 1
done
```

Conclusion:

This project showcases how to render and refresh a running digital clock in ASCII format using only shell scripting, incorporating terminal manipulation and real-time updates.