KEY BREAKER

Programmer:	Almog Hamdani
Grade:	10D
School:	Mekif Yod Alef - Ashdod
ID:	212940191
Teacher:	Anatoly Peymer
Project Name:	Key Breaker

Contents

Introduction

Files

Source files

KEY_BREA.ASM
DRAW.INC
PICS.INC

Game files

DIGIT.PCX
GAME.PCX
LOST.PCX
MENU.PCX
ABOUT.PCX
EXPLAIN.PCX
OPEN.PCX
WIN.PCX
LOCK.PCX
0 - 9.PCX
EXIT.PCX
HELP.PCX

Info

Workspace: Turbo Assembler

Development environment: Atom

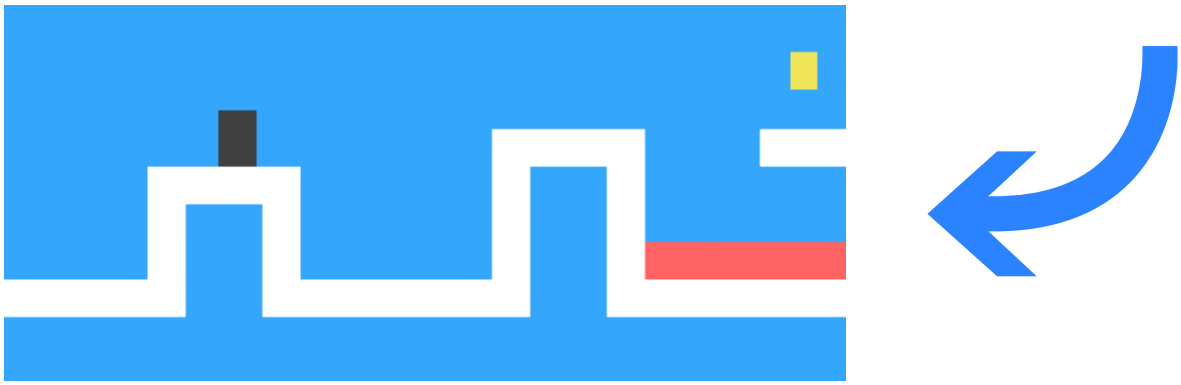
Running environment: DOSBox

Executable: KEY_BREA.EXE

The Idea

My game was inspired by a game concept that I found online when I was trying to decide which game or program I should make. The picture as you can see below is describing a game where you have a player that need to pass obstacles and collect coins to reach it's target.
I used this idea to make my game itself and came up with the idea of the story of the game.
Before I found the concept of the game, I wanted to make a game where the user has a key and it needs to pass mini-games in order to unlock each digit, my game has the same idea just without the mini-games, each digit requires the user to pass a level of the game where the first digit is the easiest to pass and the last digit is the hardest.

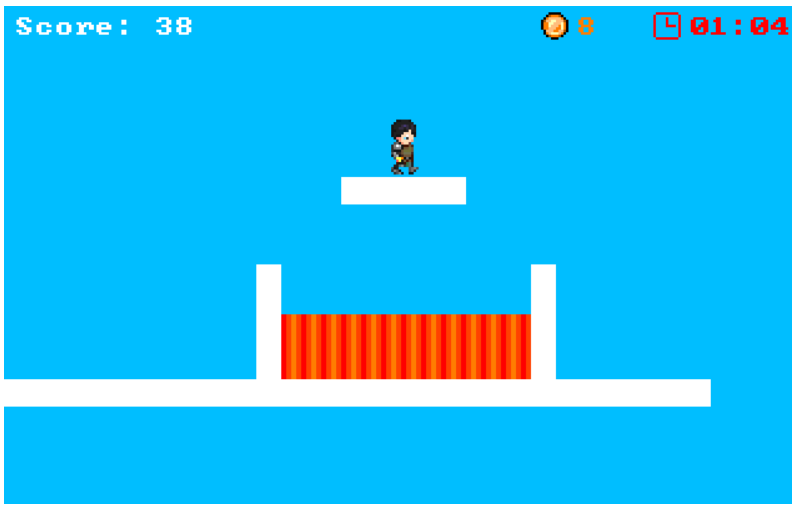
The game concept



The Game

Key Breaker is a single-player game. The story of the game is based on breaking a key, a 6-digit combination that is changing every run of the game. At first, all the 6 digits are locked, and you're given a game to pass to unlock each one of them. Each level(unlock of a digit) has it's own target score and target coins that the player need to reach in order to get the key for the digit and unlock it. In addition, each level has a max time to pass it that is based on the level's difficulty.
The game is about a player that need to pass obstacles and collect coins, for each coin the user collect or obstacle the user pass, he earns score which using that he passes the level.
The game consists 6 levels based on the amount of digits in the key.

Gameplay



How To Play

In order to play the game you must have a computer running DOS or a computer with any firmware of Windows, Linux and MacOS using the free DOS emulator called DOSBox.
In your DOS environment, locate the game' executable KEY_BREA.EXE and launch it. (Make sure all game files are available in the same folder!)
After you've started the game and started playing, the key menu will show up, in this menu you'll have the progress on your unlocking adventure. In addition, in the key menu, you'll see the current level info, target coins, target score and max time.
After starting the level, the game itself will start and your character will appear, the movement of the character is being done using the left and right arrow keys to move it horizontally and using space to jump vertically.
While exploring the game's map, you'll encounter obstacles that can lead to your death and coins that will progress you to finishing the level. Once you reach the target score and target coins of the level, the key will appear on the game's map and you'll have to collect it. After collecting the key you'll be back in the key menu and a digit will be unlocked, under it will be the time it took you to unlock it. Unlock all 6 digits to win the game.

Versions

This version

This version of the game includes all of my goals when I started this game, 6 stages, a random 6-digit key and the game itself from the game concept. With that, this game has a lot more functionality that I intended this game to have, it has a coins that are collectible by the player (coin animation), it has a timer that limits the user time to complete the level, it has a player that accelerate to it's speed and slow down to stop and many more features..

Improvements

I wanted to add to the game a scoreboard which will rank the top 10 players with the highest score using a database in a file. In addition, I wanted to add to the game more obstacles and make a wide variety of obstacles so that the obstacles would not repeat them self many times in a row. Moreover, I wanted to add a detailed story that will make the game more interesting.

Logic

Menu

The game's menu is made of a PCX image that contains all the the user's options. In the menu a circle is used to define the current selected option, and using the down and up arrow keys, the user can navigate between the different options. The circle is a custom circle with a color pattern on it that is a bitmap in the code. The user chooses an option by pressing enter and the option executes.

Key Menu

The key menu is the menu where the user seeing his progress in unlocking the key. The key menu show each digit for the key using PCX files that are image of each digit (0 - 9), if the digit is still locked, it shows a PCX image that has a lock in it instead. In the key menu there are the information for the current level (target score, coins and time) which are printed using a custom print method that allows custom background and foreground color in graphic mode. This menu leads to the game itself.

Logic

Player

In the game there is a map and a player, the player is a bitmap of a male character. The player can move using the arrow keys and jump using space. When the player starts to move, it accelerates to it's max speed using delay in moving the player, when the velocity of the player increases the delay of the player decreases and vice versa. This way the player "increases" and "decreases" it's speed and smooth animations are displayed.
In the vertical axis, the player performs the same that he performed as in the horizontal axis, but without the requirement of the user to hold the key, instead, the player jumps until he reaches it's max height and starts falling down until he reaches the ground. On the movement of the player there are checks that are made to insure that he doesn't move through walls, and checks to see if the user collected a coin or a key and finished the level. If the y position of the player reaches the end of the screen, the game detects that the player has lost and display him the lost screen.

Logic

Map

The map of the game is based on the obstacles that are currently need to be displayed. At first, when the level starts, the first obstacle is just a flat floor (a flat floor isn't really an obstacle but it's still counts as one in the game and score is given by passing it as well). The map moves by adding a block at the start of each part and removing part at the end of each part, this creates a smooth moving transition without actually moving the part and redrawing the screen. Each time the first part of the map is gone and cannot be seen anymore, a new random part is being generated as the third part, and the previous third part becomes the second part, and the second part becomes the first part. Every time a new part is being generated, the user get's score for the part he passed, each obstacle has it's own score value, for example, flat part, has a score value of 1. When the player reached the target score and target coins, the next part to be generated automatically becomes a flat part, so that the key can be drawn above it and the player can collect it to finish the level.

Logic

Coins

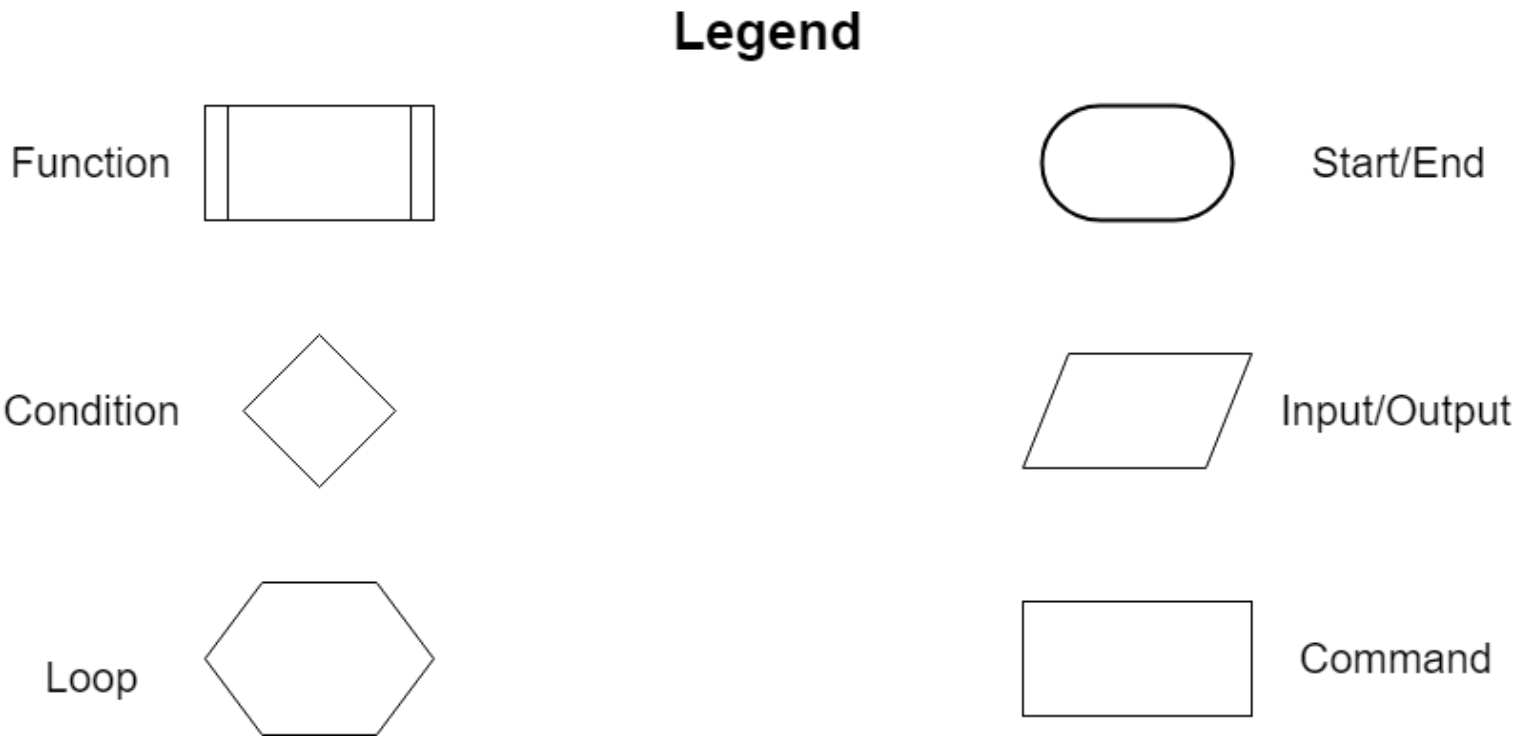
Coins are a collectible item that the player must collect in order to pass the game, each level has it's own target coins that the user need to reach. The coins position is being determined by the obstacle that they are appearing on, each obstacle has it's random range of coin's x and coin's y which the coin system uses in order to generate the coin's position in the map. The coin animation is being performed by a set of predefined frames of a spinning coin that were extracted from a GIF file. The coin system keeps track of the current coin frame and is in charge of changing it, the change of the current coin is being done using a specific delay that makes the coin's animation smooth and in good pace. The coin system is being ran in the game loop so it's redrawing the coins on the map each time and that is why the coins are being moved with the map itself. The coin's frames are surrounded with a special black border that has a unique color index to it, a color that is not being used in the game at all. Using the unique black border, the game is able to identify whenever the player touches a coin and calculates which coin using the player's location.

Logic

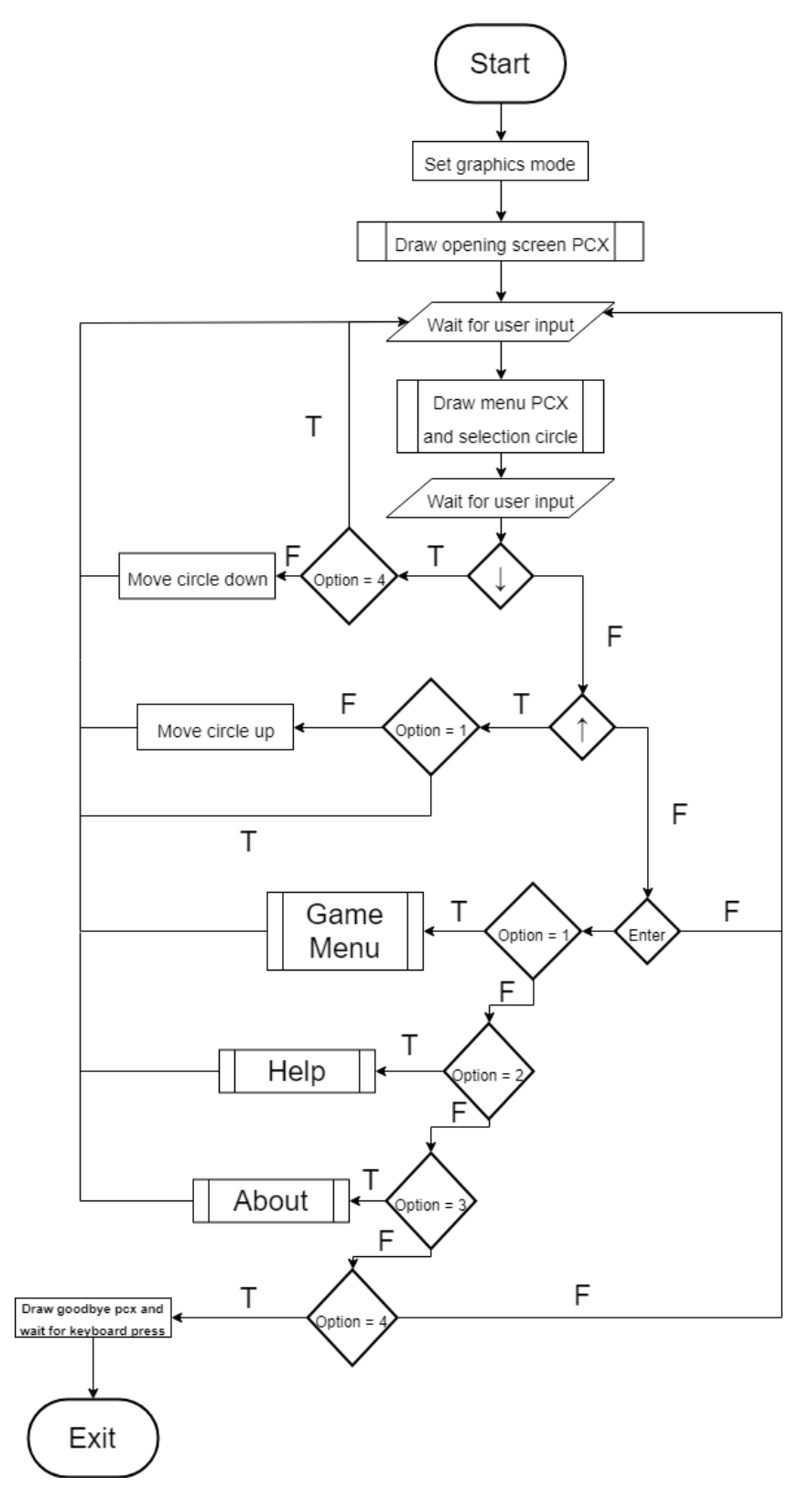
Sound

Sound is implemented in the game in many places, the menu's dot movement sound, the coin collection sound, the win sound and the lost sound. All of those sounds are being done using the sound system. The sound system works by playing sound frequencies with static delay between them. The system works by initializing it with an array of sound frequencies and a delay between them. Now the system handler need to be called in a loop, for example, in the coin collection sound, the sound system handler need be called in the game loop in order for the sound to work. Behind the scenes, the sound system handler, send each time the delay runs out, a sound frequency from the sound array by it's order, to the audio port, each channel of the sound at a time. If the sound needs to ended, the sound frequencies array has reached it's end, the handler closes the audio port and by that shutting down the sound at the user's speakers.

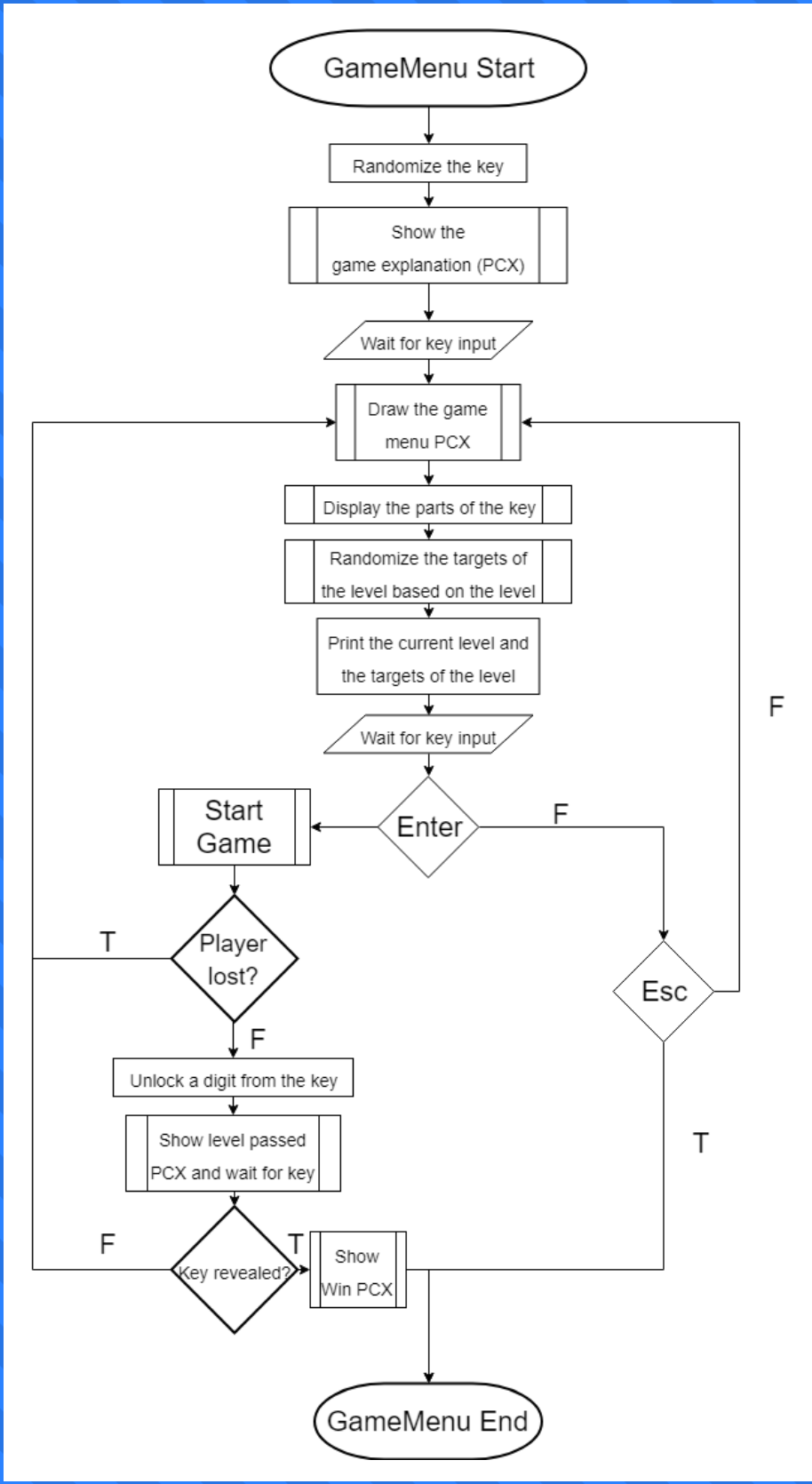
Flow Chart


Legend
Function
Start/End
Condition
Input/Output
Loop
Command

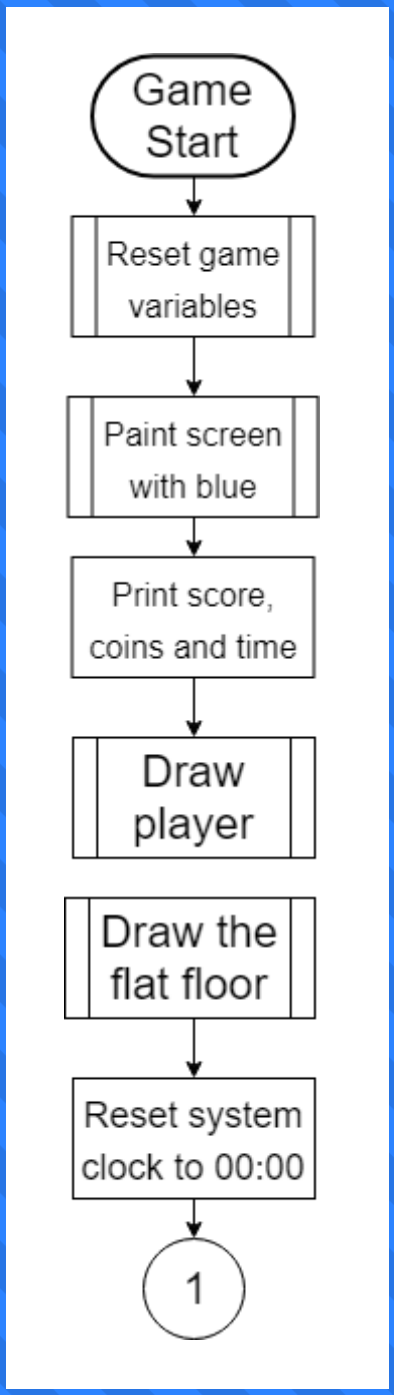
Flow Chart


Start
Set graphics mode
Draw opening screen PCX
Wait for user input
Draw menu PCX and selection circle
Wait for user input
T
F
T
Option = 4
Move circle down
↓
F
F
T
Option = 1
Move circle up
↑
F
T
F
T
Option = 1
Enter
F
Game Menu
F
T
Option = 2
Help
F
T
Option = 3
About
F
T
Option = 4
F
Draw goodbye pcx and wait for keyboard press
Exit

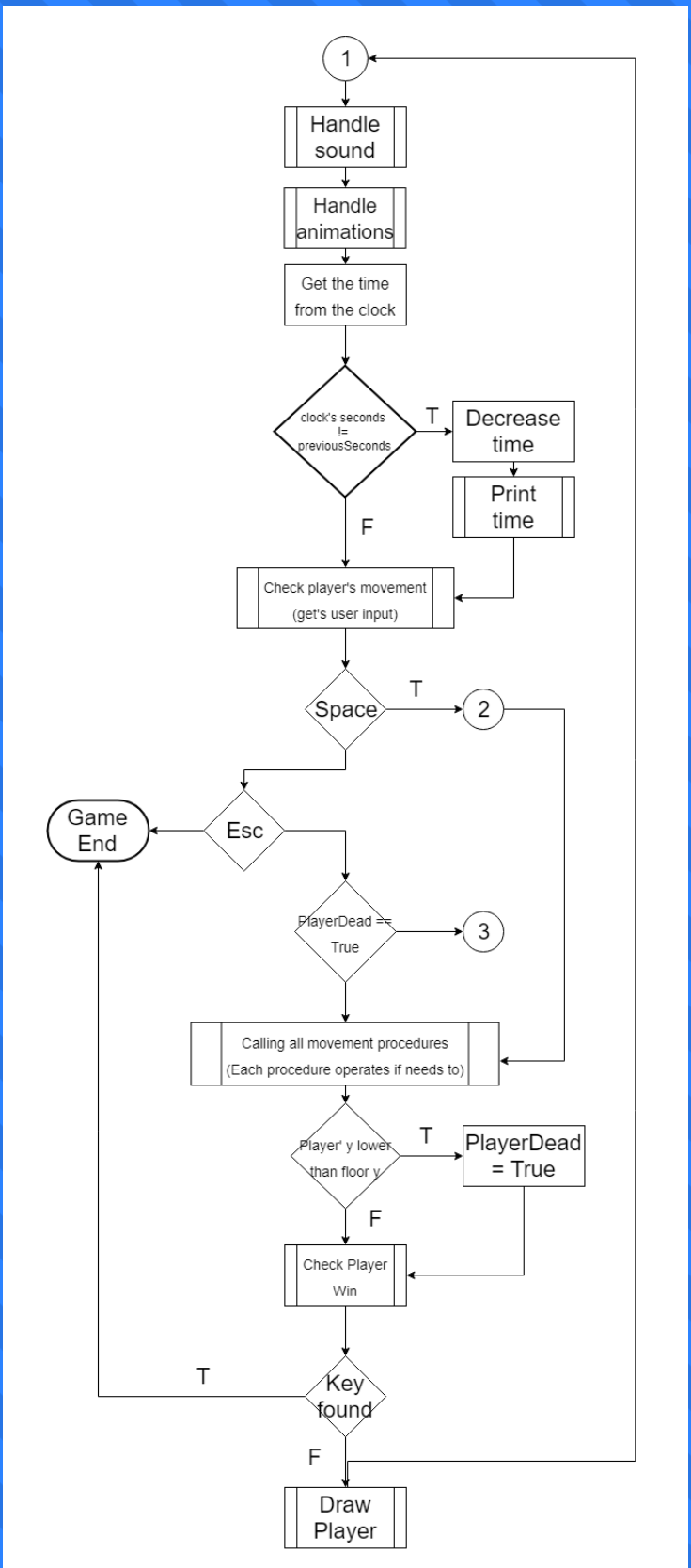
Flow Chart


GameMenu Start
Randomize the key
Show the game explanation (PCX)
Wait for key input
Draw the game menu PCX
Display the parts of the key
Randomize the targets of the level based on the level
Print the current level and the targets of the level
Wait for key input
Enter
F
Start Game
Player lost?
T
F
Esc
F
T
Unlock a digit from the key
Show level passed PCX and wait for key
Key revealed?
F
T
Show Win PCX
GameMenu End

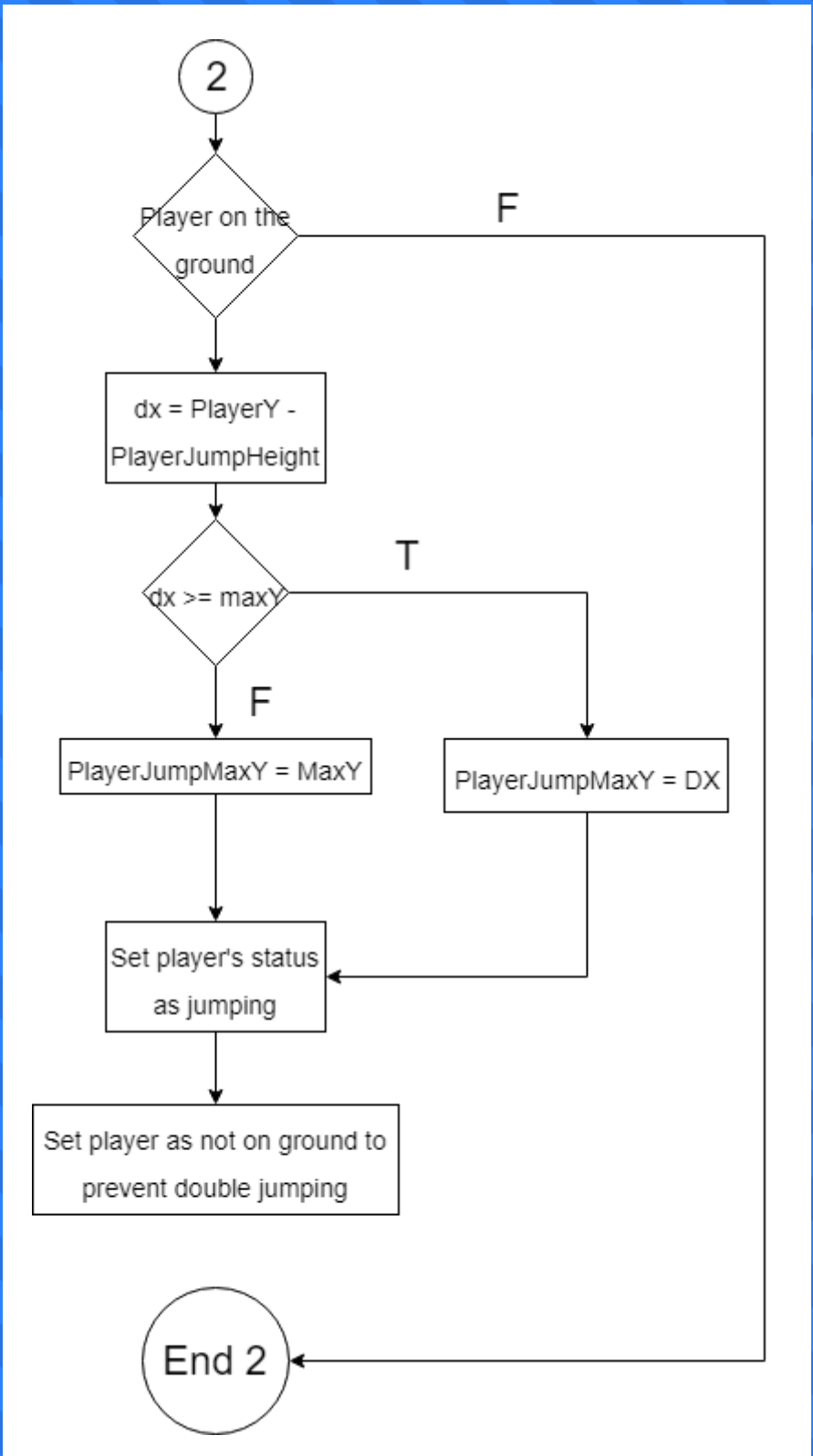
Flow Chart


Game Start
Reset game variables
Paint screen with blue
Print score, coins and time
Draw player
Draw the flat floor
Reset system clock to 00:00
1

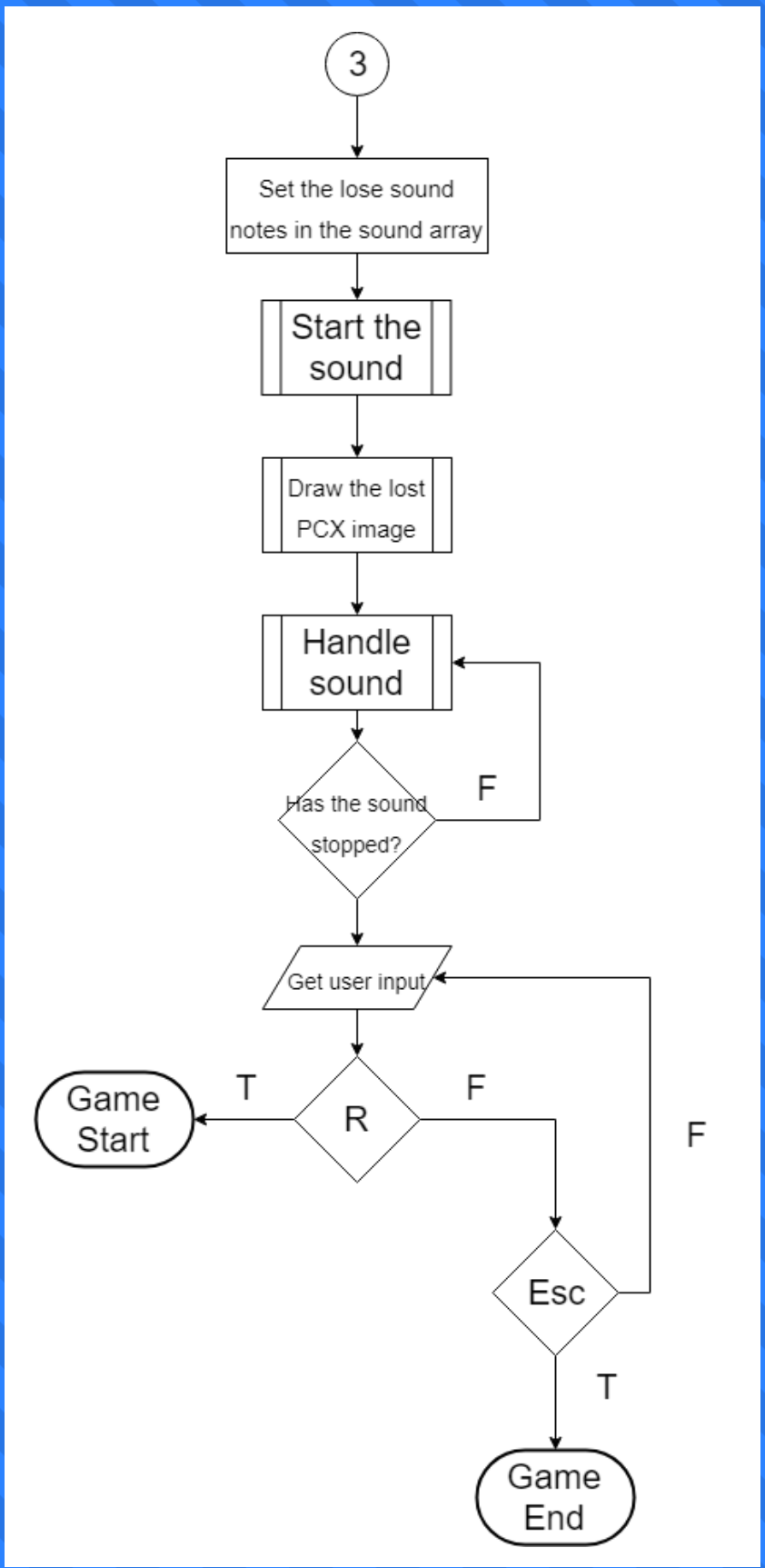
Flow Chart


1
Handle sound
Handle animations
Get the time from the clock
clock's seconds != previousSeconds
T
Decrease time
Print time
F
Check player's movement (get's user input)
Space
T
2
Esc
Game End
PlayerDead == True
3
Calling all movement procedures (Each procedure operates if needs to)
Player' y lower than floor y
T
PlayerDead = True
F
Check Player Win
Key found
T
F
Draw Player

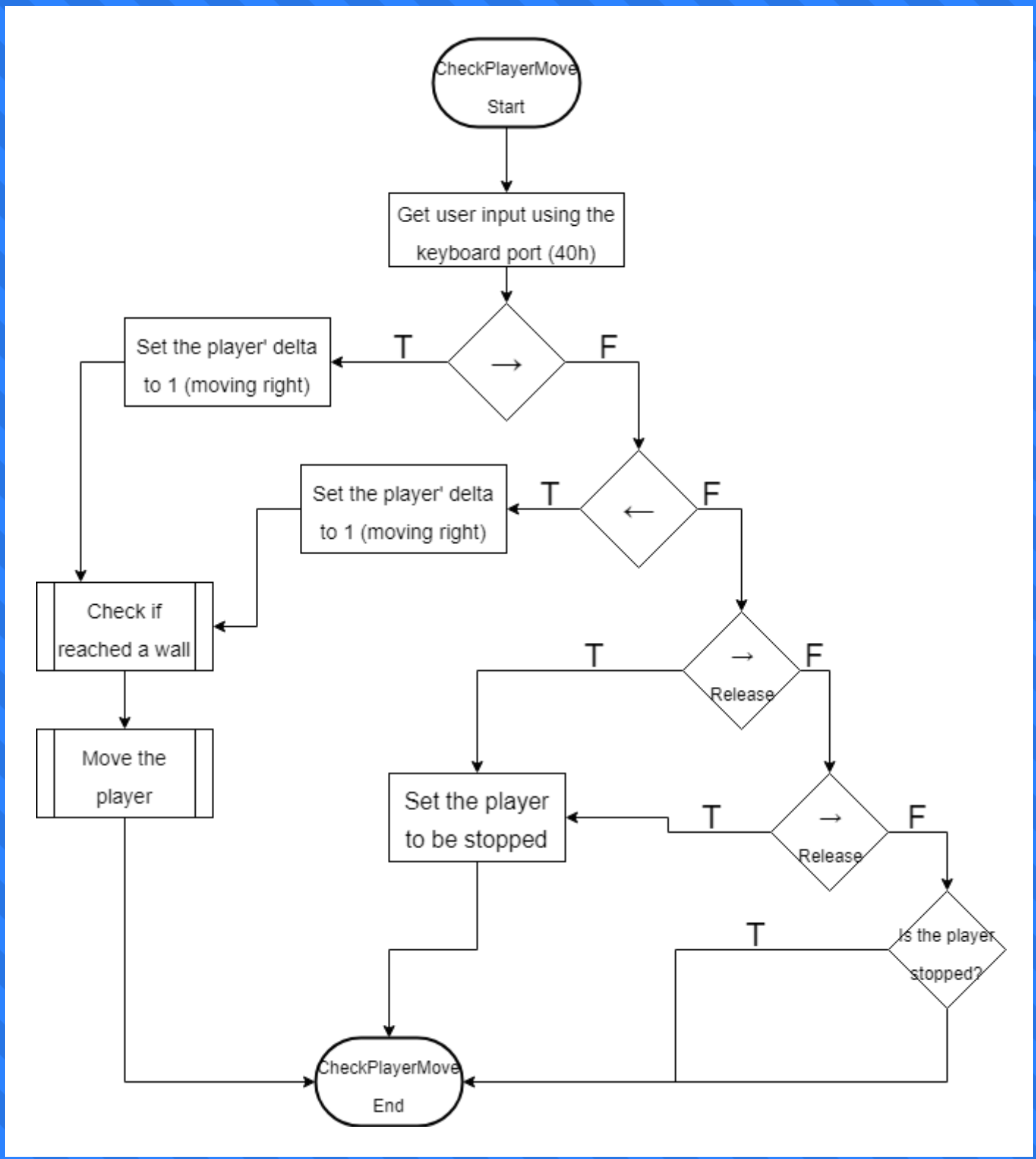
Flow Chart


2
Player on the ground
F
dx = PlayerY - PlayerJumpHeight
dx >= maxY
T
F
PlayerJumpMaxY = MaxY
PlayerJumpMaxY = DX
Set player's status as jumping
Set player as not on ground to prevent double jumping
End 2

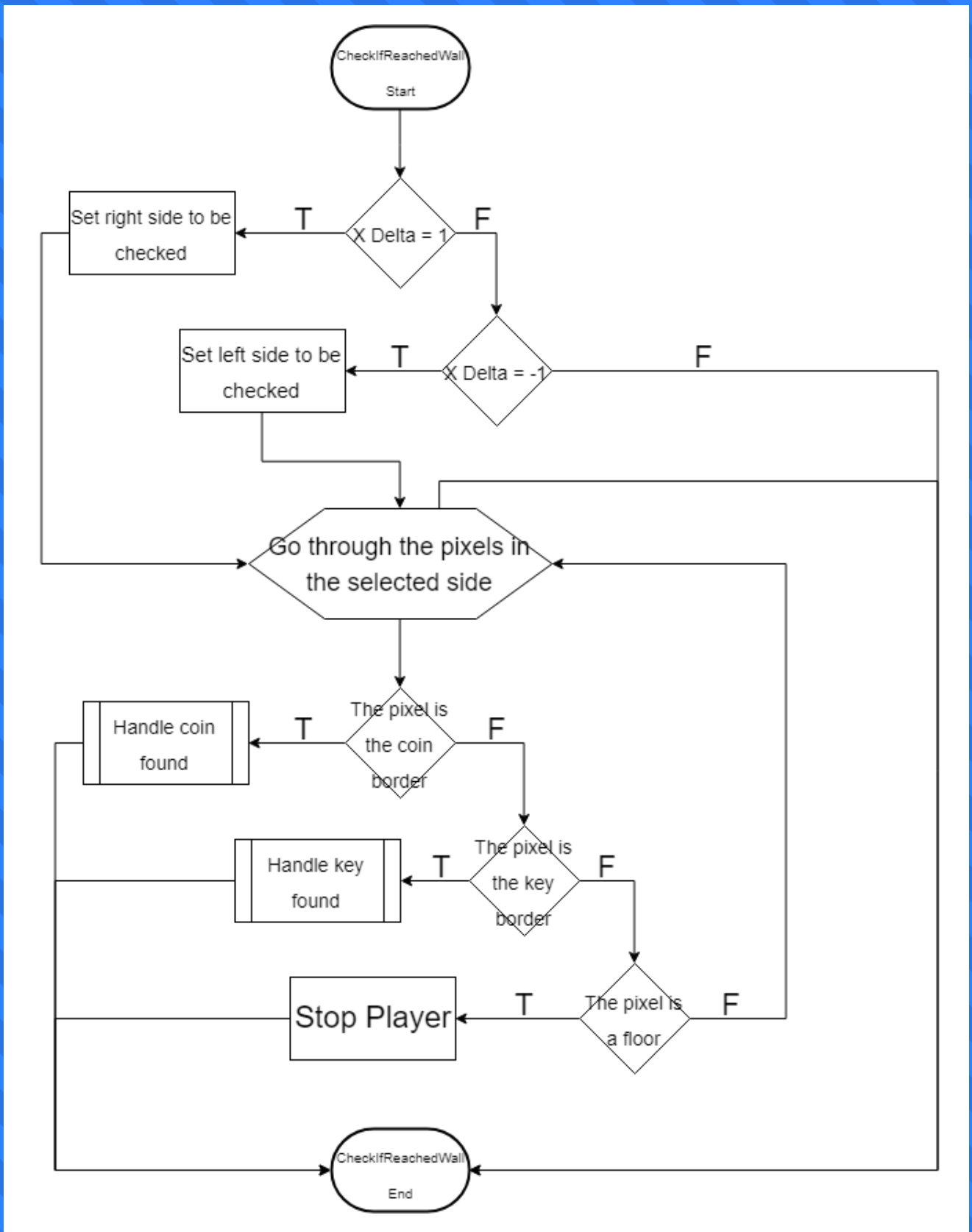
Flow Chart


3
Set the lose sound notes in the sound array
Start the sound
Draw the lost PCX image
Handle sound
Has the sound stopped?
F
Get user input
Game Start
T
R
F
F
Esc
T
Game End

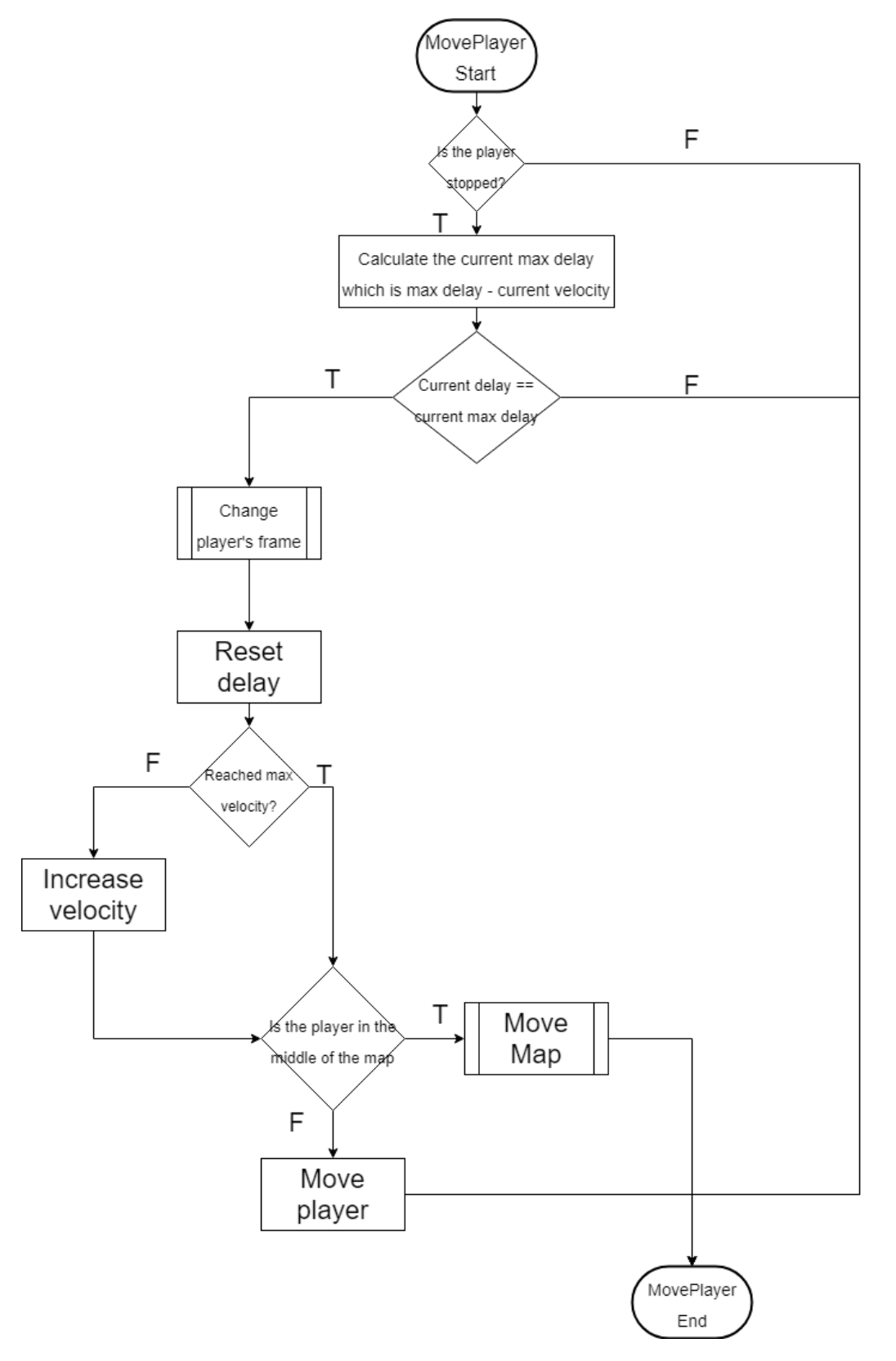
Flow Chart


CheckPlayerMove
Start
Get user input using the keyboard port (40h)
→
T
F
Set the player' delta to 1 (moving right)
←
T
F
Set the player' delta to 1 (moving right)
Check if reached a wall
→
Release
T
F
Move the player
Set the player to be stopped
→
Release
T
F
Is the player stopped?
T
CheckPlayerMove
End

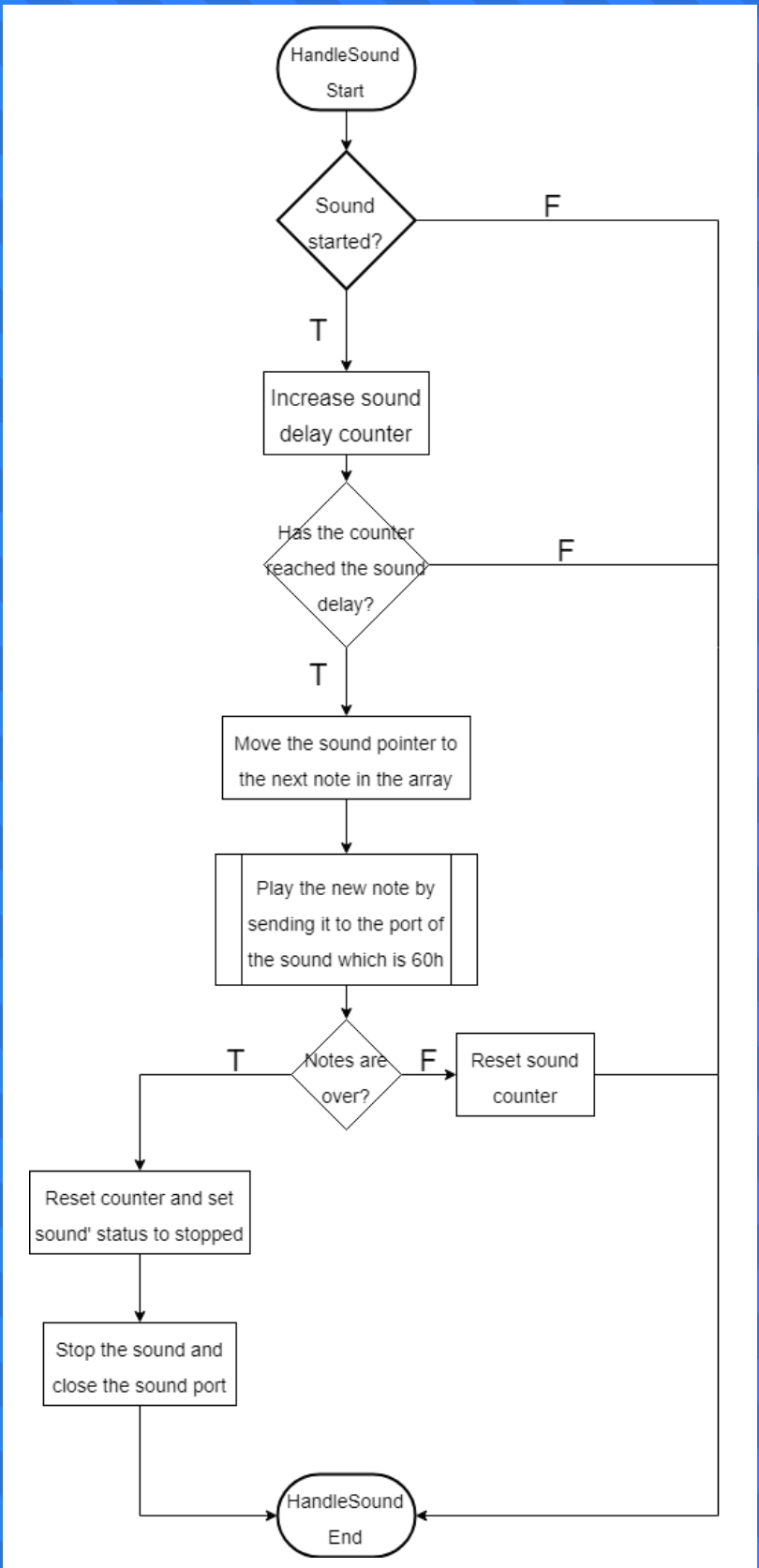
Flow Chart


CheckIfReachedWall
Start
X Delta = 1
T
F
Set right side to be checked
X Delta = -1
T
F
Set left side to be checked
Go through the pixels in the selected side
The pixel is the coin border
T
F
Handle coin found
The pixel is the key border
T
F
Handle key found
The pixel is a floor
T
F
Stop Player
CheckIfReachedWall
End

Flow Chart


MovePlayer Start
Is the player stopped?
F
T
Calculate the current max delay which is max delay - current velocity
Current delay == current max delay
T
F
Change player's frame
Reset delay
Reached max velocity?
F
T
Increase velocity
Is the player in the middle of the map
T
Move Map
F
Move player
MovePlayer End

Flow Chart


HandleSound Start
Sound started?
F
T
Increase sound delay counter
Has the counter reached the sound delay?
F
T
Move the sound pointer to the next note in the array
Play the new note by sending it to the port of the sound which is 60h
Notes are over?
T
F
Reset sound counter
Reset counter and set sound' status to stopped
Stop the sound and close the sound port
HandleSound End

Procedures

GameMenu

Task: This is handling the menu of the game and shows the key to the user.
Input: None.
Output: The game and it's menu.
Registers: AX, SI

RandomizeDigitInfo

Task: This is giving a random score and coins based on the specific digit (level).
Input: CurrentDigit
Output: TargetScore and TargetCoins for the level.
Registers: BX, CX, SI, DI

PrintTimeForDigit

Task: This is drawing the time for a given digit.
Input: SI <- Pointer to the current digit in key.
Output: The time under the digit.
Registers: AX, CX, DX, BP, SI

Procedures

DrawCurrentDigitInfo

Task: This is drawing the info for the current digit (level) and the total score.
Input: CurrentDigit
Output: The info on the screen.
Registers: SI, AX, DX, CX, BP

DrawKeyNumbers

Task: This is drawing the digits of the key or a lock if not unlocked yet.
Input: None
Output: The digits or the lock on the screen in the key position.
Registers: BX, CX, SI, DI

RandomizeKey

Task: This is initializing the variables of the game.
Input: None.
Output: The game variables initiated.
Registers: AX, ES

Procedures

Game

Task: This proc is starting the game for the current level.
Input: TargetScore, TargetCoins and Time
Output: The game playable to the user.
Registers: AX, BX, CX, DX, SI, DI, ES

DrawKeyIfNeeded

Task: This is drawing the key if the player reached the target.
Input: PlayerFinishedLevel <- Did the player reach the target.
Output: The key on the map if needed.
Registers: AX, SI

CheckPlayerWin

Task: This is checking if the player reached the target score and coins
Input: Score, Coins.
Output: PlayerFinishedLevel <- If the player finished it is true, else false
Registers: AX

Procedures

CheckPlayerDeath

Task: This is checking if the player should die.
Input: PlayerX, PlayerY <- Position of the player.
Output: PlayerDead <- Is the player dead or alive.
Registers: None.

AnimationHandler

Task: This is handling the animations in the game.
Input: None.
Output: The animations in the game.
Registers: None.

DrawCoinsForMap

Task: This is drawing the coins in the map.
Input: None.
Output: The coins on the screen.
Registers: AX, CX, SI, DI

Procedures

ChangePlayerFrame

Task: This is changing the player' pic frame if needed.
Input: PlayerXDelta <- Is the player moving left or right (-1, 1)
Output: The player frame changed.
Registers: AX, CX, SI, DI

DrawCoin

Task: This is drawing a coin in a given position on the screen.
Input: StartX, StartY <- Coin's position
Output: The coin on the screen.
Registers: SI, DX, AX, CX

TimeHandler

Task: This is handles the time, it is called every time a second passed.
Input: DH <- New seconds
Output: The time changed on the screen
Registers: None

Procedures

PrintMap

Task: This is printing the initial map on the screen.
Input: None.
Output: The map on the screen.
Registers: None.

RandomData

Task: This is generation a random data in a given range.
Input: BX <- Min, CX <- Max, DI <- The pointer to the data.
Output: Random data.
Registers: AX, BX, CX, DX, DI

MoveMap

Task: This is moving the map left and adding a new part if needed.
Input: None.
Output: The map will be moved but not redrawn.
Registers: AX, BX, CX, DX, SI, DI

Procedures

PrintScore

Task: This is printing the score on the screen.
Input: Score.
Output: The score on the screen.
Registers: None.

PrintCoins

Task: This is printing the coins on the screen.
Input: Coins.
Output: The coins on the screen.
Registers: None.

PrintTime

Task: This is printing the time on the screen.
Input: Time.
Output: The time on the screen.
Registers: AX, CX, DX, SI, BP

Procedures

NumToStr

Task: This is converting a number to a string.
Input: AX <- Number, BP <- The minimum characters,
SI <- Pointer to the string that will hold the number.
Output: The number in the given string.
Registers: AX, BX, SI, DX, BP

FixMap

Task: This is fixing the map on movement.
Input: None.
Output: The map on the screen fixed after movement.
Registers: AX, BX, SI

FixTallPart

Task: This is fixing the tall part on movement.
Input: BX <- The address of the map offset,
Temp <- The part' number.
Output: The tall part on the screen fixed.
Registers: AX, BX, CX, DI

Procedures

FixHolePart

Task: This is fixing the hole part on movement.
Input: BX <- The address of the map offset,
Temp <- The part' number.
Output: The hole part on the screen fixed.
Registers: AX, CX, DI

FixLavaPart

Task: This is fixing the lava part on movement.
Input: BX <- The address of the map offset,
Temp <- The part' number.
Output: The lava part on the screen fixed
Registers: AX, DX, CX, DI

HandleLavaColor

Task: This is handles the lava color and returns it.
Input: None.
Output: LavaCurrentColor <- The lava color
Registers: AX

Procedures

SendNoteSound

Task: This is sending the current sound note.
Input: SoundPtr <- Pointer to current sound note.
Output: The sound note playing
Registers: AX, SI

HandleCoinFound

Task: This is handling the coin collection.
Input: (X, Y) <- Interaction point with the coin
Output: The coin removed from the screen and increased amount of coins.
Registers: AX, BX, CX, DX

StartSound

Task: This is starting the sound.
Input: SoundNotes <- The notes of the sound.
Output: The sound playing in the speaker.
Registers: AX

Procedures

HandleSound

Task: This is handling the sound and switches notes when needed.

Input: SoundNotes <- The notes of the sound,
SoundDelay <- The delay between the sounds

Output: The sound playing in the speaker

Registers: AX, SI

PlayerJump

Task: This is making the player jump.

Input: PlayerJumping <- 1 To start jumping or 0 otherwise

Output: The player jumping.

Registers: AX, CX, DX

PlayerStopJump

Task: This is making the player to stop jumping smoothly.

Input: PlayerStopJump <- 1 To stop jumping or 0 otherwise.

Output: The player stops jumping.

Registers: AX, CX, DX

Procedures

PlayerGravity

Task: This is making the player to fall down

Input: None.

Output: The player falling down.

Registers: AX, CX, DX

HandleKeyFound

Task: This is handling a key collection.

Input: None.

Output: KeyFound <- True and the sound of a level pass.

Registers: None.

CheckPlayerMove

Task: This is checking if the player has moved and moving it if needed.

Input: None.

Output: The player to be moved but not redrawn

Registers: AX

Procedures

MovePlayer

Task: This is handling whatever the player should move or just the map and it's moving what's necessary + it's handling the acceleration of the player.
Input: PlayerXDelta <- The delta of the player, right or left.
Output: The player moved.
Registers: DX

CheckIfReachedWall

Task: This is checking if the player hit a wall.
Input: PlayerXDelta <- The delta of the player, right or left.
Output: The player stopped or if a coin found.
Registers: CX, DX

Procedures

StopPlayer

Task: This is stopping the player using acceleration.
Input: PlayerStop <- Should the player stop or not.
Output: The player stopped on the screen.
Registers: DX

PrintColorfulText

Task: This is printing text in a specific background and foreground color in graphic mode.
Input: SI <- Pointer to the string,
DL <- Background color,
DH <- Foreground color,
StartX, StartY <- The position of the text
Output: The text in the colors wanted on the screen
Registers: AX, CX, BX, DI, DX, ES

Procedures

Help

Task: This is displaying the help screen.
Input: None.
Output: The help screen displayed.
Registers: AX

About

Task: This is displaying the about screen.
Input: None.
Output: The about screen displayed.
Registers: AX

DrawRectangle

Task: This is drawing a rectangle on the screen.
Input: (StartX, StartY) <- Position of the rectangle,
(SizeWidth, SizeHeight) <- Size of the rectangle,
Color <- The color of the rectangle
Output: The rectangle drawn on the screen
Registers: AX, CX

Procedures

PaintScreen

Task: Paints the entire screen in 1 color.
Input: Color <- The color to paint with.
Output: The entire screen with the color.
Registers: None.

ReadPCX

Task: Reads a PCX file.
Input: FileName <- The PCX file name.
Output: FILEBUF <- The contents of the file,
FileSize <- The size of the file
Registers: AX, BX, CX, DX

DrawPCX

Task: Draws a PCX image to a specific location on the screen.
Input: StartX <- The X to start draw in,
StartY <- The Y to start draw in,
FileName <- The PCX file name
Output: The PCX on the screen.
Registers: AX, CX

Procedures

ClearSprite

Task: Removes a sprite from the screen.

Input: X, Y, Color,
CX <- Sprite' height, DX <- Sprite' width

Output: The sprite removed from the screen.

Registers: None.

PrintSprite

Task: Prints a sprite to the screen.

Input: StartX, StartY, SizeWidth, SizeHeight, SI <- Sprite' offset
Carry flag on -> Negative and Overflow check
ImageFlipped -> Is the image should be drawn flipped

Output: The sprite drawn on the screen.

Registers: AX, SI, DX

Procedures

PutPixel

Task: Write pixel to the screen.
Input: X, Y, Color
Output: The pixel on the screen.
Registers: AX, DI, ES

GetPixel

Task: Get a pixel from the screen.
Input: X, Y
Output: Color <- The color of the pixel.
Registers: AX, DI, ES

Code

File Name: KEY_BREA.ASM

Contents: The file contains all the logic of the game including the main menu, the game menu, the game itself and a lot more..

```
;-----------------------------------------
; PURPOSE : Final Project - Key Breaker
; SYSTEM  : Turbo Assembler Ideal Mode
; AUTHOR  : Almog Hamdani
;-----------------------------------------

%TITLE "Key Breaker"

                IDEAL

                MODEL small

                STACK 256

                P386

;------------------------
; ClearScreen - Clears the screen in graphic mode
;------------------------
; Input:
;       None
; Output:
;       None
; Registers:
;       AX
;------------------------
MACRO ClearScreen
        mov ax, 13h
        int 10h
ENDM ClearScreen

;------------------------
; CopyString - Copy a string to string
;------------------------
; Input:
;       STR1 - The src string, STR2 - The dst string, LEN - The length of the
src string
; Output:
;       The src in the dst string
; Registers:
;       AX, DX, ES, SI, DI, CX
;------------------------
MACRO CopyString STR1, STR2, LEN
        mov dx, es ; Save extra segment original

;-----  Set es as data segment
        mov ax, ds
        mov es, ax

        lea si, [STR1]
        lea di, [STR2]
        mov cx, LEN
        rep movsb

        mov es, dx
ENDM CopyString
```

```
;------------------------
; DrawRect - Draws a rectangle on the screen
;------------------------
; Input:
;       X, Y, SizeHeight, SizeWidth
; Output:
;       The rectangle on the screen
; Registers:
;       None
;------------------------
MACRO DrawRect X, Y, Width, Height, Clr
        mov [Color], Clr
        mov [StartX], X
        mov [StartY], Y
        mov [SizeWidth], Width
        mov [SizeHeight], Height

        pusha
        call DrawRectangle
        popa
ENDM DrawRect

;------------------------
; FillArray - Fills an array with a specific value
;------------------------
; Input:
;       Array, Value, Len, Size <- The size of each element in the array,
Offset <- The distance from each element to element
; Output:
;       The values in the array
; Registers:
;       DI, CX
;------------------------
MACRO FillArray Array, Value, Len, Size, Offset
        local ValueLoop
        lea di, [Array] ; Point to array
        mov cx, Len ; Set the amount of values to be changed

ValueLoop:
        mov [Size di], Value ; Set the value

        add di, Offset
        loop ValueLoop
ENDM FillArray
```

```
;------------------------
; PrintStringWithNumbers - Fills an array with a specific value
;------------------------
; Input:
;        String <- The name of the string var
;        Number <- The name of the number var
;        BG <- The background color
;        FG <- The foreground color
;        Numoffset <- Where to put the number in the string
;        MinChars <- Minimum amount of characters
;        Position: X, Y
; Output:
;       The values in the array
; Registers:
;       DI, CX
;------------------------
MACRO PrintStringWithNumbers String, Number, BG, FG, NumOffset, MinChars, X,
Y
;-----  Copy the number to the string by converting it
        lea si, [String]
        add si, NumOffset ; Set where to set the score in the string
        mov ax, [Number]
        mov bp, MinChars ; Minimum amount of chars
        call NumToStr
        mov [byte si], 0 ; Set end of the string

;-----  Print the string onto the screen
        lea si, [String]
        mov dl, BG ; Background color
        mov dh, FG ; Foreground color

;-----  Set position of the text
        mov [StartX], X
        mov [StartY], Y

        call PrintColorfulText
ENDM PrintStringWithNumbers
```

```
;------------------------
; DrawPlayer - Draws the player
;------------------------
; Input:
;       PlayerX, PlayerY
; Output:
;       The player on the screen
; Registers:
;       AX, CX, DX, SI, ES
;------------------------
MACRO DrawPlayer
;-----  Set player' position

        mov ax, [PlayerX]
        mov [StartX], ax

        mov ax, [PlayerY]
        mov [StartY], ax

;-----  Set the size of the player
        mov [SizeHeight], PlayerHeight
        mov [SizeWidth], PlayerWidth

        lea si, [PlayerPic]

;-----  Calculate the frame offset using mul with the current frame and the
size of each frame
        xor ax, ax
        mov al, [PlayerCurrentFrame]
        xor dx, dx
        mov cx, PlayerHeight * PlayerWidth
        mul cx

        add si, ax

;-----  Set whatever the player should be drawn flipped
        mov al, [PlayerFlipped]
        mov [ImageFlipped], al

        clc ; Clear carry flag so that the function won't check for overflow
and negetive position
        call PrintSprite
ENDM
```

```
;----Constants-----

PlayerMaxFrameDelay             equ 7

DotJump                         equ 33
DotStartX                       equ 105
DotStartY                       equ 75

ScreenWidth                     equ 320
GameBackgroundColor             equ 53
FloorY                          equ 150
DistanceFromWall                equ 5
MidScreen                       equ 160
EndScreen                       equ 320
MaxX                            equ (ScreenWidth - PlayerWidth)
MinX                            equ 0
MaxY                            equ 20

PlayerFloorDist                 equ 0
PlayerJumpHeight                equ (100 - PlayerHeight)

FloorColor                      equ 15
FloorSize                       equ 10
FlatFloor                       equ 1
FlatFloorCoinXRangeMin          equ 20
FlatFloorCoinXRangeMax          equ MidScreen - 20
FlatFloorCoinYRangeMin          equ (FloorY - PlayerJumpHeight) + 20
FlatFloorCoinYRangeMax          equ (FloorY - CoinMinDist - CoinPicHeight)

TallFloor                       equ 2
TallFloorHeightRangeMin         equ 40
TallFloorHeightRangeMax         equ 65
TallFloorWidthRangeMin          equ 30
TallFloorWidthRangeMax          equ 50
TallLowerFlatLength             equ 50

HoleFloor                       equ 3
HoleFloorWidthRangeMin          equ 40
HoleFloorWidthRangeMax          equ 80
HoleFloorCoinYRangeMin          equ (FloorY - PlayerJumpHeight)
HoleFloorCoinYRangeMax          equ (FloorY - CoinMinDist - CoinPicHeight -
65)

LavaFloor                       equ 5
LavaHeight                      equ 25
LavaColorRangeStart             equ 40
LavaColorRangeEnd               equ 42
LavaColorBlocksChange           equ 2
LavaLowerFlatLength             equ 20
LavaLength                      equ (160 - LavaLowerFlatLength * 2 -
FloorSize * 2)
LavaUpperWidthRangeMin          equ 20
LavaUpperWidthRangeMax          equ (LavaLength - 20)
LavaUpperHeightRangeMin         equ 80
LavaUpperHeightRangeMax         equ 105
LavaWallsHeight                 equ 45
```

```
MaxHVelocity                    equ 15
MaxHDelay                       equ 19
MaxVVelocity                    equ 8
MaxVDelay                       equ 10

False                           equ 0
True                            equ 1

ScoreX                          equ 5
ScoreY                          equ 5

TimeX                           equ (ScreenWidth - 45)
TimeY                           equ 5

CoinsX                          equ (ScreenWidth - 90)
CoinsY                          equ 5

EscapeScanCode                  equ 1
RScanCode                       equ 13h
UpArrowScanCode                 equ 72
DownArrowScanCode               equ 80
RightArrowScanCode              equ 77
LeftArrowScanCode               equ 75
ReleasedRightArrowScanCode      equ 11001101b
ReleasedLeftArrowScanCode       equ 11001011b
EnterScanCode                   equ 28
SpaceScanCode                   equ 57
ReleasedSpaceScanCode           equ 10111001b

AnimationCountMax               equ 15
CoinMinDist                     equ 5

KeyRangeStart                   equ 0
KeyRangeEnd                     equ 9

CurrentDigitTextX               equ 18
CurrentDigitTextY               equ Midscreen - 55

TargetScoreTextX                equ 18
TargetScoreTextY                equ CurrentDigitTextY + 15

TargetCoinsTextX                equ EndScreen - 140
TargetCoinsTextY                equ Midscreen - 55

TimeTextX                       equ EndScreen - 140
TimeTextY                       equ Midscreen - 40

TotalScoreTextX                 equ MidScreen - 60
TotalScoreTextY                 equ MidScreen - 4

DigitTimeY                      equ 80

CoinYellowColor                 equ 42
WhiteColor                      equ 15
GameMenuBGColor                 equ 21
RedColor                        equ 40
```

```
CoinValue                       equ 2
CoinBorderColor                 equ 0

KeyBorderColor                  equ 17

KeyInitialX                     equ EndScreen + MidScreen + 20
KeyInitialY                     equ FloorY - KeyPicHeight - 10

SoundCoinDelayMax               equ 40
SoundKeyDelayMax                equ 0A000h
SoundMenuDelayMax               equ 6000h

;------------------

                DATASEG

PCXErrorMSG                 db 'An error occurred during drawing PCX file!
Please try again!$'
FileHandle                  dw ?
FileName                    db 30 dup (?)
FileSize                    dw ?
ImageHeight                 dw ?
ImageWidth                  dw ?
ImageFlipped                db False
StartX                      dw ?
StartY                      dw ?

X                           dw ?
Y                           dw ?
Color                       db ?

SizeHeight                  dw ?
SizeWidth                   dw ?
SkipColumns                 dw 0

Temp                        dw ?
TempByte                    db ?

MapOrder                    db FlatFloor, FlatFloor, FlatFloor
MapOffset                   dw 0, MidScreen, EndScreen
MapDataHeight               dw 0, 0, 0
MapDataWidth                dw 0, 0, 0
MapCoin                     db False, False, False
MapCoinX                    dw 0, 0, 0
MapCoinY                    dw 0, 0, 0

LavaColorBlocksCounter      db 0
LavaCurrentColor            db LavaColorRangeStart
LavaCurrentColorOffset      db 1

PlayerFrameCurrentDelay     db 0
PlayerCurrentFrame          db 1

PlayerFlipped               db False
PlayerX                     dw ?
PlayerY                     dw ?
PlayerDead                  db False
```

```
PlayerXDelta                dw 0
PlayerHVelocity             dw 0
CurrentHDelay               dw 0
PlayerStop                  db False

PlayerVVelocity             dw 0
CurrentVDelay               dw 0
PlayerJumpMaxY              dw 0
PlayerJumping               db False
PlayerStopJumping           db False
PlayerOnGround              db True

OpeningFileName             db 'visuals\open.pcx$'
OpeningNameLen              equ 17

MenuFileName                db 'visuals\menu.pcx$'
MenuNameLen                 equ 17

LostFileName                db 'visuals\lost.pcx$'
LostNameLen                 equ 17

GameMenuFileName            db 'visuals\game.pcx$'
GameMenuNameLen             equ 17

HelpFileName                db 'visuals\help.pcx$'
HelpNameLen                 equ 17

ExitFileName                db 'visuals\exit.pcx$'
ExitNameLen                 equ 17

DigitUnlockFileName         db 'visuals\digit.pcx$'
DigitUnlockNameLen          equ 18

AboutFileName               db 'visuals\about.pcx$'
AboutNameLen                equ 18

WinFileName                 db 'visuals\win.pcx$'
WinNameLen                  equ 17

GameExplantionFileName      db 'visuals\explain.pcx$'
GameExplantionNameLen       equ 21

DotOffset                   dw 0
OptionSelected              db 1

ScanCode                    dw ?

HelpText                    db "Get help here!", 13, 10, '$'
AboutText                   db "Made by Almog Hamdani!", 13, 10, '$'

Score                       dw 0
ScoreText                   db "Score: ", 20 dup(0)

LevelTime                   dw 0
Time                        dw 0
TimeText                    db 20 dup(0)
```

```
Coins                       dw 0
CoinsText                   db 20 dup(0)

TextBitmap                  dw 4 dup(?)

PreviousSeconds             db 0

AnimationCounter            db 0
CoinFrame                   dw 0

Key                         dw 6 dup(0)
KeyUnlocked                 db True, 5 dup(True)
KeyTimes                    dw 6 dup(?)

TargetScore                 dw 150
TargetScoreText             db 'Target score : ', 10 dup(?)

TargetCoins                 dw 15
TargetCoinsText             db 'Target coins: ', 10 dup(?)

TargetTimeText              db 'Time: ', 10 dup(?)

CurrentDigitText            db 'Current digit: ', 5 dup(?)
CurrentDigit                dw 1

TotalScore                  dw 0
TotalScoreText              db 'Total score: ', 10 dup (?)

DigitX                      dw ?
DigitY                      dw ?

GameScoreRangeStart         dw 10, 10, 20, 20, 60, 60
GameScoreRangeEnd           dw 20, 20, 60, 60, 120, 120

GameCoinsRangeStart         dw 2, 2, 5, 5, 15, 15
GameCoinsRangeEnd           dw 5, 5, 15, 15, 30, 30

GameTimes                   dw 120, 90, 300, 240, 480, 390

KeyX                        dw KeyInitialX
KeyY                        dw KeyInitialY

KeyFound                    db False
PlayerFinishedLevel         db False

SoundNotes                  dw 20 dup(-1)
SoundCounter                dw 0
SoundPtr                    dw offset SoundNotes
SoundStarted                db False
SoundDelay                  dw ?

KeyDigitFileName            db 'digits\0.pcx$'
KeyDigitNameDigitOffset     equ 7
KeyDigitNameLen             equ 13

LockFileName                db 'digits\lock.pcx$'
```

```
LockNameLen                  equ 16

INCLUDE 'PICS.INC'

                CODESEG
Start:
        ; Set data segment
        mov ax, @data
        mov ds, ax

        ; Set video memory as extra segment
        mov ax, 0A000h
        mov es, ax

        ; Set graphic mode
        mov ax, 13h
        int 10h

PrintOpeningScreen:
        CopyString OpeningFileName, FileName, OpeningNameLen
        mov [StartX], 0
        mov [StartY], 0

        call DrawPCX

        ; Wait for input
        xor ah, ah
        int 16h

PrintMenu:
        ClearScreen

        CopyString MenuFileName, FileName, MenuNameLen
        mov [StartX], 0
        mov [StartY], 0

        call DrawPCX

PrintSelectDot:
        ; Set dot' position
        mov [StartX], DotStartX
        mov [StartY], DotStartY

        mov ax, [DotOffset]
        add [StartY], ax

        ; Set dot' size
        mov [SizeHeight], DotPicHeight
        mov [SizeWidth], DotPicWidth

        lea si, [DotPic] ; Set dot pic offset

        clc ; Clear carry flag so that the function won't check for overflow
and negetive position
        mov [ImageFlipped], False
        call PrintSprite ; Print dot
```

```
CheckArrows:
        call HandleSound

        ; Get input from keyboard
        mov ah, 1
        int 16h

        jz CheckArrows

        push ax
        mov ah, 0Ch
        mov al, 0
        int 21h
        pop ax

        cmp ah, EnterScanCode ; Check if enter is pressed
        je JumpToSelected

        cmp ah, UpArrowScanCode ; Check if up arrow was pressed
        je HandleUp

        cmp ah, DownArrowScanCode ; Check if down arrow was pressed
        je HandleDown

        jmp CheckArrows

HandleUp:
        ; If current option is the first, don't go up
        cmp [OptionSelected], 1
        je CheckArrows

        ; Set the notes' frequency
        mov [SoundNotes], 4063
        mov [SoundNotes + 2], -1 ; End of notes
        mov [SoundDelay], SoundMenuDelayMax
        call StartSound

        ; Set dot' position
        mov [X], dotStartX
        mov [Y], DotStartY

        mov ax, [DotOffset]
        add [Y], ax

        ; Set dot' size
        mov cx, DotPicHeight
        mov dx, DotPicWidth

        call ClearSprite

        dec [OptionSelected] ; Set option to be the previous
        sub [DotOffset], DotJump ; Set currect offset

        jmp PrintSelectDot ; Print dot again

HandleDown:
        ; If current option is the last, don't go down
```

```
        cmp [OptionSelected], 4
        je CheckArrows

        ; Set the notes' frequency
        mov [SoundNotes], 4063
        mov [SoundNotes + 2], -1 ; End of notes
        mov [SoundDelay], SoundMenuDelayMax
        call StartSound

        ; Set dot' position
        mov [X], dotStartX
        mov [Y], DotStartY

        mov ax, [DotOffset]
        add [Y], ax

        ; Set dot' size
        mov cx, DotPicHeight
        mov dx, DotPicWidth

        call ClearSprite

        inc [OptionSelected] ; Set option to be the next
        add [DotOffset], DotJump ; Set currect offset

        jmp PrintSelectDot ; Print dot again

JumpToSelected:
;-----  Set text mode
        mov ax, 13h
        int 10h

;-----  Jump to the selected option
        cmp [OptionSelected], 1
        je StartGame

        cmp [OptionSelected], 2
        je ShowHelp

        cmp [OptionSelected], 3
        je ShowAbout

        jmp Exit

StartGame:
        call GameMenu
        jmp PrintMenu

ShowHelp:
        call Help
        jmp PrintMenu

ShowAbout:
        call About
        jmp PrintMenu

Exit:
```

```
;-----  Stop sound - Get the current status from port 61h
        in al, 61h

;-----  Turn off bits 0 and 1 in order to stop the sound and send back to the
port
        and al, 11111100b
        out 61h, al

;-----  Draw the exit PCX image
        CopyString ExitFileName, FileName, ExitNameLen
        mov [StartX], 0
        mov [StartY], 0

        call DrawPCX

;-----  Wait for keyboard press
        xor ah, ah
        int 16h

;-----  Set text mode
        mov ax, 3h
        int 10h

;-----  Exit
        mov ax, 4C00h
        int 21h

;----------PROC----------

;------------------------
;GameMenu - This is handling the menu of the game and shows the key to the
user.
;------------------------
;Input:
;       None
;Output:
;       The game and it's menu
;Registers:
;       AX, SI
;------------------------
PROC GameMenu
;-----  Initialize variables
        mov [TotalScore], 0
        mov [CurrentDigit], 1
        FillArray KeyUnlocked, False, 6, byte, 1
        FillArray KeyTimes, 100, 6, word, 2

;-----  Randomize key
        call RandomizeKey

@@GameExplantion:
;-----  Print the game explantion picture
        CopyString GameExplantionFileName, FileName, GameExplantionNameLen
        mov [StartX], 0
        mov [StartY], 0
        call DrawPCX
```

```
        call DrawKeyNumbers

;------        Wait for key
        xor ah, ah
        int 16h

@@DrawUI:
;-----  Print the game menu picture
        CopyString GameMenuFileName, FileName, GameMenuNameLen
        mov [StartX], 0
        mov [StartY], 0
        call DrawPCX

;-----  Print UI
        call DrawKeyNumbers
        call RandomizeDigitInfo
        call DrawCurrentDigitInfo

@@GetKey:
;-----  Wait for keyboard press
        xor ah, ah
        int 16h

;-----  Check if enter is pressed
        cmp ah, EnterScanCode
        je @@StartGame

;-----  If escape is pressed, return to main menu
        cmp ah, EscapeScanCode
        je @@Return

        jmp @@GetKey

@@StartGame:
;-----  Get offset
        mov si, [CurrentDigit] ; Get the current digit
        dec si ; The array of ranges is starting with 0
        shl si, 1 ; Get the offset in words so mul by 2

;-----  Get the time for the digit and set it in the level time
        mov ax, [GameTimes + si]
        mov [LevelTime], ax

        call Game

;-----  Clean keyboard buffer
        mov ah, 0Ch
        mov al, 0
        int 21h

;-----  Check if the key was found, if it is found, reveal the digit, else
just return to menu
        cmp [KeyFound], True
        je @@AfterGame

        jmp @@DrawUI
```

```
@@AfterGame:
;-----  Print the digit unlock picture before adding the new one
        CopyString DigitUnlockFileName, FileName, DigitUnlockNameLen
        mov [StartX], 0
        mov [StartY], 0
        call DrawPCX

;-----  Skip the note that was played during the print
        ; Set in the counter the sound delay - 1 becuase it is increasing it
to skip the note
        mov ax, [SoundDelay]
        dec ax
        mov [SoundCounter], ax

@@WaitForKeyAndSound:
        call HandleSound

;-----  Check if the sound is finished, if it isn't return again
        cmp [SoundStarted], False
        jne @@WaitForKeyAndSound

;-----  Get if the keyboard was pressed
        mov ah, 1
        int 16h

        jz @@WaitForKeyAndSound

;-----  Clean keyboard buffer
        mov ah, 0Ch
        mov al, 0
        int 21h

;-----  Get offset for the digit
        mov si, [CurrentDigit] ; Get the current digit
        dec si ; The array of ranges is starting with 0
        mov [KeyUnlocked + si], True

;-----  Get the time that the level took for the user
        mov ax, [LevelTime]
        sub ax, [Time]

        shl si, 1 ; Get the offset in words so mul by 2
        mov [KeyTimes + si], ax ; Set time
        inc [CurrentDigit] ; Move to next digit

;-----  Add score to total score
        mov ax, [Score]
        add [TotalScore], ax

        cmp [CurrentDigit], 6 + 1
        jge @@FinishGame

        jmp @@DrawUI

@@FinishGame:
;-----  Print the win picture
        CopyString WinFileName, FileName, WinNameLen
```

```
        mov [StartX], 0
        mov [StartY], 0
        call DrawPCX

;-----  Printing keys and total score
        call DrawKeyNumbers
        PrintStringWithNumbers TotalScoreText, TotalScore, GameMenuBGColor,
RedColor, 15, 0, TotalScoreTextX, TotalScoreTextY

;-----  Wait for keyboard press
        xor ah, ah
        int 16h

@@Return:
        ret
ENDP GameMenu

;------------------------
;RandomizeDigitInfo - This is giving a random score and coins based on the
specific digit (level)
;------------------------
;Input:
;       CurrentDigit
;Output:
;       TargetScore and TargetCoins for the level
;Registers:
;       BX, CX, SI, DI
;------------------------
PROC RandomizeDigitInfo
;-----  Get offset
        mov si, [CurrentDigit] ; Get the current digit
        dec si ; The array of ranges is starting with 0
        shl si, 1 ; Get the offset in words so mul by 2

;-----  Get random score for level
        mov bx, [GameScoreRangeStart + si] ; Min
        mov cx, [GameScoreRangeEnd + si] ; Max
        lea di, [TargetScore]
        call RandomData

;-----  Get random coins for level
        mov bx, [GameCoinsRangeStart + si] ; Min
        mov cx, [GameCoinsRangeEnd + si] ; Max
        lea di, [TargetCoins]
        call RandomData

        ret
ENDP RandomizeDigitInfo

;------------------------
;PrintTimeForDigit - This is drawing the time for a given digit
;------------------------
;Input:
;       SI <- Pointer to the current digit in key
;Output:
;       The time under the digit
;Registers:
```

```
;       AX, CX, DX, BP, SI
;------------------------
PROC PrintTimeForDigit
;-----  Get the time for the digit
        sub si, offset Key ; Get the offset of the current key
        add si, offset KeyTimes ; Add the var to the offset to get the time
        mov ax, [word si] ; Take the time

;-----  Get minutes and seconds from time
        xor dx, dx ; Reset dx for div
        mov cx, 60 ; Divide by 60 seconds
        div cx

;-----  Convert minutes to text
        push dx ; Save seconds
        mov bp, 2 ; Amount of chars
        lea si, [TimeText] ; Point to the text holder of the Time
        call NumToStr

        mov [byte si], ':' ; Put divider between minutes and seconds
        inc si

;-----  Convert minutes to text
        pop ax ; Get seconds from stack
        mov bp, 2 ; Amount of chars
        call NumToStr

        mov [byte si], 0 ; Set end of the string

;-----  Get the x for the time from the digit x
        mov ax, [DigitX]
        add ax, 5
        mov [StartX], ax

        mov [StartY], DigitTimeY

;-----  Print time
        lea si, [TimeText]
        mov dl, GameMenuBGColor ; Set text background
        mov dh, WhiteColor ; Set foreground color, red color
        call PrintColorfulText

        ret
ENDP PrintTimeForDigit

;------------------------
;DrawCurrentDigitInfo - This is drawing the info for the current digit
(level) and the total score
;------------------------
;Input:
;       CurrentDigit
;Output:
;       The info on the screen
;Registers:
;       SI, AX, DX, CX, BP
;------------------------
PROC DrawCurrentDigitInfo
```

```
        PrintStringWithNumbers CurrentDigitText, CurrentDigit,
GameMenuBGColor, GameBackgroundColor, 15, 0, CurrentDigitTextX,
CurrentDigitTextY
        PrintStringWithNumbers TargetScoreText, TargetScore, GameMenuBGColor,
WhiteColor, 15, 0, TargetScoreTextX, TargetScoreTextY
        PrintStringWithNumbers TargetCoinsText, TargetCoins, GameMenuBGColor,
CoinYellowColor, 14, 0, TargetCoinsTextX, TargetCoinsTextY
        PrintStringWithNumbers TotalScoreText, TotalScore, GameMenuBGColor,
RedColor, 13, 0, TotalScoreTextX, TotalScoreTextY

;-----  Get offset
        mov si, [CurrentDigit] ; Get the current digit
        dec si ; The array of ranges is starting with 0
        shl si, 1 ; Get the offset in words so mul by 2

;-----  Get the time for the digit
        mov ax, [GameTimes + si]

;-----  Get minutes and seconds from time
        xor dx, dx ; Reset dx for div
        mov cx, 60 ; Divide by 60 seconds
        div cx

;-----  Convert minutes to text
        push dx ; Save seconds
        mov bp, 2 ; Amount of chars
        lea si, [TargetTimeText + 6] ; Point to the text holder of the Time
        call NumToStr

        mov [byte si], ':' ; Put divider between minutes and seconds
        inc si

;-----  Convert minutes to text
        pop ax ; Get seconds from stack
        mov bp, 2 ; Amount of chars
        call NumToStr

        mov [byte si], 0 ; Set end of the string

;-----  Set the location of the text
        mov [StartX], TimeTextX
        mov [StartY], TimeTextY

;-----  Print time
        lea si, [TargetTimeText]
        mov dl, GameMenuBGColor ; Set text background
        mov dh, WhiteColor ; Set foreground color, red color
        call PrintColorfulText
        ret
ENDP DrawCurrentDigitInfo

;------------------------
;DrawKeyNumbers - This is drawing the digits of the key or a lock if not
unlocked yet
;------------------------
;Input:
;       None
```

```
;Output:
;       The digits or the lock on the screen in the key position
;Registers:
;       AX, BP, DI, SI
;------------------------
PROC DrawKeyNumbers
        lea di, [KeyUnlocked] ; Point to the array that contains the whatever
the digit is unlocked or not
        lea si, [Key] ; Point to the current key' digit
        mov [Temp], 6 ; Move 6 times as there is 6 digits

;-----  Set first digit position
        mov [DigitX], 18
        mov [DigitY], 17

@@DigitLoop:
;-----  Check if the current digit is unlocked, if it is draw it
        cmp [byte di], True
        je @@DrawDigit

@@DrawLock:
;-----  Set position of the lock to be drawn
        mov ax, [DigitX]
        mov [StartX], ax

        mov ax, [DigitY]
        mov [StartY], ax

;-----  Draw the lock
        pusha
        CopyString LockFileName, FileName, LockNameLen ; Set the name of the
file
        popa

        pusha
        call DrawPCX
        popa

        jmp @@LoopCheck

@@DrawDigit:
;-----  Set position of the question mark to be drawn
        mov ax, [DigitX]
        mov [StartX], ax

        mov ax, [DigitY]
        mov [StartY], ax

;-----  Get the file name for the digit
        pusha
        mov ax, [si] ; Take the current key' digit
        mov bp, 0 ; No need for leading zeros
        lea si, [KeyDigitFileName + KeyDigitNameDigitOffset] ; Get where to
put the digit
        call NumToStr
        popa
```

```
;-----  Draw the digit
        pusha
        CopyString KeyDigitFileName, FileName, KeyDigitNameLen ; Set the name
of the file
        popa

        pusha
        call DrawPCX
        popa

;-----  Print time for the digit
        pusha
        call PrintTimeForDigit
        popa

@@LoopCheck:
        add [DigitX], 47

        inc di
        add si, 2

;-----  Check if the finished 6 digits to print
        dec [Temp]
        jnz @@DigitLoop

        ret
ENDP DrawKeyNumbers

;------------------------
;RandomizeKey - This is generation a random 6-digit key
;------------------------
;Input:
;       None
;Output:
;       The random key in the key var
;Registers:
;       AX, BX, CX, DI
;------------------------
PROC RandomizeKey
        lea di, [Key] ; Point to the key holder

@@GenerateKey:
;-----  Set min and max for the digit
        mov bx, KeyRangeStart
        mov cx, KeyRangeEnd

        call RandomData ; Generate random digit

        add di, 2 ; Move to next digit

;-----  Get the last byte that contains the key
        lea ax, [Key]
        add ax, 5 * 2

;-----  Loop the generator until finished 6 digits
        cmp di, ax
        jbe @@GenerateKey
```

```
        ret
ENDP RandomizeKey

;-------------------------
;RandomizeKey - This is initializing the variables of the game
;-------------------------
;Input:
;       None
;Output:
;       The game variables initiated
;Registers:
;       AX, ES
;-------------------------
PROC InitGameVariables
;-----  Set video memory
        mov ax, 0A000h
        mov es, ax

;-----  Initialize variables
        mov [PlayerDead], False
        mov [Score], 0
        mov [Coins], 0

        mov ax, [LevelTime]
        mov [Time], ax

        mov [KeyFound], False
        mov [PlayerFinishedLevel], False

        mov [KeyX], KeyInitialX
        mov [KeyY], KeyInitialY

        mov [PlayerXDelta], 0
        mov [PlayerHVelocity], 0
        mov [CurrentHDelay], 0
        mov [PlayerStop], False

        mov [PlayerVVelocity], 0
        mov [CurrentVDelay], 0
        mov [PlayerJumpMaxY], 0
        mov [PlayerJumping], False
        mov [PlayerStopJumping], False
        mov [PlayerOnGround], True

        ; Reset parts
        mov [MapOrder], FlatFloor
        mov [MapOrder + 1], FlatFloor
        mov [MapOrder + 2], FlatFloor

        ; Reset offsets
        mov [MapOffset], 0
        mov [MapOffset + 2], MidScreen
        mov [MapOffset + 4], EndScreen

        ; Reset data
        mov [MapDataHeight], 0
```

```
        mov [MapDataHeight + 2], 0
        mov [MapDataHeight + 4], 0

        ; Reset data
        mov [MapDataWidth], 0
        mov [MapDataWidth + 2], 0
        mov [MapDataWidth + 4], 0

        ; Reset data
        mov [MapCoin], False
        mov [MapCoin + 1], False
        mov [MapCoin + 2], False

        mov [PlayerCurrentFrame], 1 ; Set the second frame of the player to
be first

        ret
ENDP InitGameVariables

;------------------------
;Game - This proc is starting the game for the current level
;------------------------
;Input:
;       TargetScore, TargetCoins and Time
;Output:
;       The game playable to the user
;Registers:
;       AX, BX, CX, DX, SI, DI, ES
;------------------------
PROC Game
@@InitGame:
        call InitGameVariables

;-----  Game
        ; Setting background color
        mov [Color], GameBackgroundColor
        call PaintScreen

;-----  Printing UI
        call PrintScore
        call PrintCoins
        call PrintTime

;-----  Printing the coin besides the coins amount
        mov [StartY], CoinsY - 3
        mov [StartX], CoinsX - CoinPicWidth - 3
        mov [SizeWidth], CoinPicWidth
        mov [SizeHeight], CoinPicHeight
        lea si, [CoinPic]
        clc ; Clear carry so that additional checks will not be executed
        mov [ImageFlipped], False ; Print the coin normal without flip
        call PrintSprite

;-----  Printing the time icon besides the coins amount
        mov [StartY], TimeY - 3
        mov [StartX], TimeX - TimePicWidth - 3
        mov [SizeWidth], TimePicWidth
```

```
        mov [SizeHeight], TimePicHeight
        lea si, [TimePic]
        clc ; Clear carry so that additional checks will not be executed
        mov [ImageFlipped], False ; Print the coin normal without flip
        call PrintSprite

        call PrintMap

;-----  Set player' position
        mov [PlayerFlipped], False
        mov [PlayerX], 50
        mov [PlayerY], FloorY - PlayerHeight - PlayerFloorDist

        DrawPlayer

;-----  Reset real time clock to 00:00:00
        mov ah, 03h
        xor cx, cx
        xor dx, dx
        int 1Ah

@@Loop:
        call AnimationHandler
        call HandleSound

;-----  Get real time clock
        mov ah, 02h
        int 1Ah

;-----  Check if the seconds were changed, if it wasn't, countinue, else,
handle time
        cmp dh, [PreviousSeconds]
        je @@Key
        call TimeHandler

@@Key:
        call CheckPlayerMove

;-----  Check if the player is dead
        cmp [PlayerDead], True
        je @@ShowLostPic

;-----  Check if the user pressed to jump, if he did, go to check if he could
jump
        cmp al, SpaceScanCode
        je @@JumpCheck

;-----  Checking if escapse is pressed, for now it's leaving the game
        dec al
        jz @@Return

@@Normal:
;-----  Calling all movement functions because they all know how to handle
according to boolean variables
        call StopPlayer
        call PlayerJump
        call PlayerStopJump
```

```
        call PlayerGravity

;-----  Checking if the player is death or he passed the level
        call CheckPlayerDeath
        call CheckPlayerWin

;-----  If the key was found, return
        cmp [KeyFound], True
        je @@Return

        DrawPlayer

        jmp @@Loop

@@JumpCheck:
;-----  Check if the player is on the ground, if it isn't skip the jump to
prevent multiple jumps and bugs.
        cmp [PlayerOnGround], True
        jne @@Normal ; Return if not on ground

;-----  Check if the current y - jump height is after max y
        mov dx, [PlayerY]
        sub dx, PlayerJumpHeight

        cmp dx, MaxY
        jge @@Jump

        mov dx, MaxY ; Set max jump y as max y

@@Jump:
        mov [PlayerJumpMaxY], dx ; Set the jump max as dx, defined before
        mov [PlayerJumping], True
        mov [PlayerOnGround], False

        jmp @@Normal ; Do as normal as well

@@Return:
;-----  Clean keyboard buffer
        mov ah, 0Ch
        mov al, 0
        int 21h

        ret

@@ShowLostPic:
;-----  Play key sound
        ; Set the notes' frequency
        mov [word SoundNotes], 1140
        mov [word SoundNotes + 2], 1207
        mov [word SoundNotes + 4], 1355
        mov [word SoundNotes + 6], 1436
        mov [word SoundNotes + 8], -1 ; End of notes
        mov [SoundDelay], SoundKeyDelayMax
        call StartSound

;-----  Print the you lost picture
        CopyString LostFileName, FileName, LostNameLen
```

```
        mov [StartX], 0
        mov [StartY], 0

        call DrawPCX

;-----  Skip the note that was played during the print
        ; Set in the counter the sound delay - 1 becuase it is increasing it
to skip the note
        mov ax, [SoundDelay]
        dec ax
        mov [SoundCounter], ax

;-----  Clear buffer
        mov ah, 0ch
        mov al, 0h
        int 21h

@@WaitForKeyAndSound:
        call HandleSound

;-----  Check if the sound is finished, if it isn't return again
        cmp [SoundStarted], False
        jne @@WaitForKeyAndSound

;-----  Get a key from the user, don't wait so that we can handle the sound
        mov ah, 1
        int 16h

;-----  While nothing was pressed just handle the sound
        jz @@WaitForKeyAndSound

;-----  Restart game
        cmp ah, RScanCode
        je @@InitGame

;-----  Return to menu
        cmp ah, EscapeScanCode
        je @@Return

        jmp @@WaitForKeyAndSound
ENDP Game

;------------------------
;DrawKeyIfNeeded - This is drawing the key if the player reached the target
;------------------------
;Input:
;       PlayerFinishedLevel <- Did the player reach the target
;Output:
;       The key on the map if needed
;Registers:
;       AX, SI
;------------------------
PROC DrawKeyIfNeeded
;-----  Check if player finished the level
        cmp [PlayerFinishedLevel], True
        jne @@Return
```

```
;-----  Set position
        mov ax, [KeyX]
        mov [StartX], ax

        mov ax, [KeyY]
        mov [StartY], ax

;-----  Draw the key
        mov [SizeWidth], KeyPicWidth
        mov [SizeHeight], KeyPicHeight
        lea si, [KeyPic]
        stc ; Check for overflows
        call PrintSprite

        dec [KeyX]

@@Return:
        ret
ENDP DrawKeyIfNeeded

;------------------------
;CheckPlayerWin - This is checking if the player reached the target score and
coins
;------------------------
;Input:
;       Score, Coins
;Output:
;       PlayerFinishedLevel <- If the player finished it is true, else false
;Registers:
;       AX
;------------------------
PROC CheckPlayerWin
;-----  Check if player hasn't reached the target score yet
        mov ax, [TargetScore]
        cmp [Score], ax
        jl @@Return

;-----  Check if player hasn't reached the target coins yet
        mov ax, [TargetCoins]
        cmp [Coins], ax
        jl @@Return

@@Finished:
        mov [PlayerFinishedLevel], True
        ret

@@Return:
        mov [PlayerFinishedLevel], False
        ret
ENDP CheckPlayerWin

;------------------------
;CheckPlayerDeath - This is checking if the player should die (checking if
fell of map)
;------------------------
;Input:
;       PlayerX, PlayerY <- Position of the player
```

```
;Output:
;       PlayerDead <- Is the player dead or alive
;Registers:
;       None
;------------------------
PROC CheckPlayerDeath
        cmp [PlayerY], FloorY + 20
        jg @@PlayerDeath
        ret

@@PlayerDeath:
        mov [PlayerDead], True
        ret
ENDP CheckPlayerDeath

;------------------------
;AnimationHandler - This is handling the animations in the game (The spinning
coin - gif)
;------------------------
;Input:
;       None
;Output:
;       The animations in the game
;Registers:
;       None
;------------------------
PROC AnimationHandler
;-----  Increase the animation delay counter until reached max
        inc [AnimationCounter]
        cmp [AnimationCounter], AnimationCountMax
        je @@Animate
        jmp @@Return

@@Animate:
        mov [AnimationCounter], 0

        call DrawCoinsForMap

;-----  Move to the next coin' frame
        inc [CoinFrame]
        cmp [CoinFrame], 6
        je @@ResetCoinFrame
        jmp @@Return

@@ResetCoinFrame:
        mov [CoinFrame], 0

@@Return:
        ret
ENDP AnimationHandler

;------------------------
;DrawCoinsForMap - This is drawing the coins in the map
;------------------------
;Input:
;       None
;Output:
```

```
;       The coins on the screen
;Registers:
;       AX, CX, SI, DI
;------------------------
PROC DrawCoinsForMap
;-----  Init
        lea si, [MapCoin]
        mov di, 0
        mov cx, 3 ; Only three parts on screen

@@CheckForCoin:
;-----  Checking if need to draw a coin in this current part
        cmp [byte si], True
        jne @@Next ; Skip the print

;-----          Draw the wanted coin
        ; Set the coin' x
        mov ax, [MapCoinX + di]
        add ax, [MapOffset + di] ; Add the part' offset
        mov [StartX], ax

        ; Set the coin' y
        mov ax, [MapCoinY + di]
        mov [StartY], ax

;-----  Save registers
        pusha

        call DrawCoin

;-----  Return registers
        popa

@@Next:
        inc si ; Move to next map coin boolean indicator
        add di, 2 ; Move to next coin' info
        loop @@CheckForCoin

        ret
ENDP DrawCoinsForMap

;------------------------
;ChangePlayerFrame - This is changing the player' pic frame if needed
;------------------------
;Input:
;       PlayerXDelta <- Is the player moving left or right (-1, 1)
;Output:
;       The player frame changed
;Registers:
;       AX, CX, SI, DI
;------------------------
PROC ChangePlayerFrame
;-----  Check if the player is going left
        cmp [PlayerXDelta], -1
        je @@FlipPlayer
```

```
        mov [PlayerFlipped], False ; The normal state of the player is not
flipped
        jmp @@FrameDelay

@@FlipPlayer:
        mov [PlayerFlipped], True ; Set player flipped if delta is -1, which
means the player is going left

@@FrameDelay:
        inc [PlayerFrameCurrentDelay] ; Increase current delay

;-----  If reached max frame delay, change frame and reset
        cmp [PlayerFrameCurrentDelay], PlayerMaxFrameDelay
        je @@ChangeFrame

        jmp @@Return

@@ChangeFrame:
        xor [PlayerCurrentFrame], 1 ; Change frame using xor (0 -> 1 and 1 ->
0)
        mov [PlayerFrameCurrentDelay], 0 ; Reset delay

@@Return:
        ret
ENDP ChangePlayerFrame

;-------------------------
;DrawCoin - This is drawing a coin in a given position on the screen
;-------------------------
;Input:
;       StartX, StartY <- Coin's position
;Output:
;       The coin on the screen
;Registers:
;       SI, DX, AX, CX
;-------------------------
PROC DrawCoin
        lea si, [CoinPic] ; Point to the coin' frames array

;-----  Get Offset for current frame
        xor dx, dx ; Reset dx for mul
        mov ax, CoinPicHeight * CoinPicWidth ; Set the size of each frame
(height * width)
        mov cx, [CoinFrame]
        mul cx

        add si, ax ; Add the calculated offset to the pointer to get the
wanted frame

;-----  Draw the coin to the screen
        mov [SizeWidth], CoinPicWidth
        mov [SizeHeight], CoinPicHeight
        stc ; Turn on carry flag so that overflow and negetive will be
checked for better optimization
        mov [ImageFlipped], False
        call PrintSprite
        ret
```

```
ENDP DrawCoin

;------------------------
;TimeHandler - This is handles the time, it is called every time a second
passed
;------------------------
;Input:
;       DH <- New seconds
;Output:
;       The time changed on the screen
;Registers:
;       None
;------------------------
PROC TimeHandler
;-----  Set the new previous seconds
        mov [PreviousSeconds], dh

;-----  Decrease the time and print it again
        dec [Time]
        call PrintTime

        cmp [Time], 0
        je @@KillPlayer

        jmp @@Return

@@KillPlayer:
        mov [PlayerDead], True

@@Return:
        ret
ENDP TimeHandler

;------------------------
;PrintMap - This is printing the initial map on the screen
;------------------------
;Input:
;       None
;Output:
;       The map on the screen
;Registers:
;       None
;------------------------
PROC PrintMap
        ; Draw 2 flat parts
        DrawRect 0, FloorY, MidScreen, FloorSize, FloorColor
        DrawRect MidScreen, FloorY, MidScreen, FloorSize, FloorColor
        ret
ENDP PrintMap

;------------------------
;RandomData - This is generation a random data in a given range
;------------------------
;Input:
;       BX <- Min, CX <- Max, DI <- The pointer to the data
;Output:
;       Random data
```

```
;Registers:
;       AX, BX, CX, DX, DI
;------------------------
PROC RandomData
;-----  Safe check - Check if the minimum is bigger or equal than the
maximum, if it is set the rnd to be the minimum
        cmp bx, cx
        jb @@NormalRnd

;-----  Set the random number as the minimum
        mov [di], bx
        ret

@@NormalRnd:
;------        Random height
        in ax, 40h ; Random number from clock

;-----  Set in cx the max - min
        sub cx, bx

        xor dx, dx ; Reset dx for result
        div cx ; Divide ax in dx, the modulo will be in dx

        add dx, bx ; Add minimum for fixed range

        mov [di], dx ; Move random number to data

        ret
ENDP RandomData

;------------------------
;MoveMap - This is moving the map left and adding a new part if needed
;------------------------
;Input:
;       None
;Output:
;       The map will be moved but not redrawn
;Registers:
;       AX, BX, CX, DX, SI, DI
;------------------------
PROC MoveMap
;-----  Decrease all parts' offsets
        lea bx, [MapOffset]
        dec [word bx]

        add bx, 2
        dec [word bx]

        add bx, 2
        dec [word bx]

;-----  Check if the middle part is now the first part
        sub bx, 2
        cmp [word bx], 0
        je @@NewMapPart

        ret
```

```
@@NewMapPart:
;-----  Take the part of the last part and add it as a score
        xor ax, ax
        mov al, [MapOrder]
        add [Score], ax

        call PrintScore

;-----  Move all parts by 1 backwards and randomize the new one
        mov al, [MapOrder + 1]
        mov [MapOrder], al

        mov al, [MapOrder + 2]
        mov [MapOrder + 1], al

;-----  Move all parts' data by 1 backwards and randomize the new one
        mov ax, [MapDataHeight + 2]
        mov [MapDataHeight], ax

        mov ax, [MapDataHeight + 4]
        mov [MapDataHeight + 2], ax

;-----  Move all parts' data by 1 backwards and randomize the new one
        mov ax, [MapDataWidth + 2]
        mov [MapDataWidth], ax

        mov ax, [MapDataWidth + 4]
        mov [MapDataWidth + 2], ax

;-----  Move all parts' coin indicator by 1 backwards and randomize the new
one
        mov al, [MapCoin + 1]
        mov [MapCoin], al

        mov al, [MapCoin + 2]
        mov [MapCoin + 1], al

;-----  Move all parts' coin x by 1 backwards and randomize the new one
        mov ax, [MapCoinX + 2]
        mov [MapCoinX], ax

        mov ax, [MapCoinX + 4]
        mov [MapCoinX + 2], ax

;-----  Move all parts' coin y by 1 backwards and randomize the new one
        mov ax, [MapCoinY + 2]
        mov [MapCoinY], ax

        mov ax, [MapCoinY + 4]
        mov [MapCoinY + 2], ax

        cmp [PlayerFinishedLevel], True
        je @@FinishFlat

;-----  Random new part, 0 - 1 for now
        in ax, 40h ; Get number from time port
```

```
        xor dx, dx
        mov cx, 3 + 1 ; Between 0 - 3 (3 Types of parts)
        div cx ; Divide ax in cx, the modulo will be in dx

;-----  Set in the new part, is coin available to true
        mov [MapCoin + 2], True

@@PartCheck:
        cmp dl, 0
        je @@Flat

;-----  Check if the new part is a tall floor
        cmp dl, 1
        je @@Tall

;-----  Check if the new part is a lava floor
        cmp dl, 2
        je @@Lava

;-----  Check if the new part is a hole floor
        cmp dl, 3
        je @@Hole

@@Flat:
        mov [MapOrder + 2], FlatFloor ; Set new part as flat floor

;-----  Get a random x for the coin
        lea di, [MapCoinX + 4] ; Set pointer to coin' x
        mov bx, FlatFloorCoinXRangeMin ; Set range min
        mov cx, FlatFloorCoinXRangeMax ; Set range max
        call RandomData

;-----  Get a random y for the coin
        lea di, [MapCoinY + 4] ; Set pointer to coin' y
        mov bx, FlatFloorCoinYRangeMin ; Set range min
        mov cx, FlatFloorCoinYRangeMax ; Set range max
        call RandomData

        jmp @@ResetOffsets ; If none of the above, just jump to reset offsets

@@Tall:
        mov [MapOrder + 2], TallFloor ; Set new part as tall floor

;-----  Get a random height for the new part
        lea di, [MapDataHeight + 4] ; Set pointer to data
        mov bx, TallFloorHeightRangeMin ; Set min
        mov cx, TallFloorHeightRangeMax ; Set max
        call RandomData

;-----  Get a random width for the new part
        lea di, [MapDataWidth + 4] ; Set pointer to data
        mov bx, TallFloorWidthRangeMin ; Set min
        mov cx, TallFloorWidthRangeMax ; Set max
        call RandomData

;-----  Set x as the middle of the part
        mov [MapCoinX + 4], (MidScreen / 2) - (CoinPicHeight / 2) - 15
```

```
;-----  Get a random y for the coin
        lea di, [MapCoinY + 4] ; Set pointer to coin' y
        mov bx, (MaxY + CoinMinDist) ; Set min as the max y in the game + the
minimum distance of the coin
        mov cx, [MapDataHeight + 4] ; Get the part' random height
        sub cx, (CoinPicHeight + CoinMinDist) ; Subtracte from it the height
and the distance of the coin to get the max
        call RandomData

        jmp @@ResetOffsets

@@Lava:
        mov [MapOrder + 2], LavaFloor ; Set new part as lava floor

;-----  Get a random height for the new part
        lea di, [MapDataHeight + 4] ; Set pointer to data
        mov bx, LavaUpperHeightRangeMin ; Set min
        mov cx, LavaUpperHeightRangeMax ; Set max
        call RandomData

;-----  Get a random width for the new part
        lea di, [MapDataWidth + 4] ; Set pointer to data
        mov bx, LavaUpperWidthRangeMin ; Set min
        mov cx, LavaUpperWidthRangeMax ; Set max
        call RandomData

;-----  Set x as the middle of the part
        mov [MapCoinX + 4], (MidScreen / 2) - (CoinPicHeight / 2)

;-----  Get a random y for the coin
        lea di, [MapCoinY + 4] ; Set pointer to coin' y
        mov bx, (MaxY + CoinMinDist) ; Set min as the max y in the game + the
minimum distance of the coin
        mov cx, FloorY
        sub cx, [MapDataHeight + 4] ; Get the part' random height which is
the floor y - the random height
        sub cx, (CoinPicHeight + CoinMinDist) ; Subtracte from it the height
and the distance of the coin to get the max
        call RandomData

        jmp @@ResetOffsets

@@Hole:
        mov [MapOrder + 2], HoleFloor ; Set new part as hole floor

;-----  Get a random width for the new part
        lea di, [MapDataWidth + 4] ; Set pointer to data
        mov bx, HoleFloorWidthRangeMin ; Set min
        mov cx, HoleFloorWidthRangeMax ; Set max
        call RandomData

;-----  Set x as the middle of the part
        mov [MapCoinX + 4], (MidScreen / 2) - (CoinPicHeight / 2)

;-----  Get a random y for the coin
        lea di, [MapCoinY + 4] ; Set pointer to coin' y
```

```
        mov bx, FlatFloorCoinYRangeMin ; Set range min
        mov cx, FlatFloorCoinYRangeMax ; Set range max
        call RandomData

        jmp @@ResetOffsets

@@ResetOffsets:
;-----  Reset offsets
        mov [MapOffset], 0
        mov [MapOffset + 2], MidScreen
        mov [MapOffset + 4], EndScreen

        ret

@@FinishFlat:
        mov dx, 0 ; Set flat
        mov [MapCoin + 2], False ; Set no coin at the end
        jmp @@PartCheck
ENDP MoveMap

;-------------------------
;PrintScore - This is printing the score on the screen
;-------------------------
;Input:
;       Score
;Output:
;       The score on the screen
;Registers:
;       None
;-------------------------
PROC PrintScore
        PrintStringWithNumbers ScoreText, Score, GameBackgroundColor,
WhiteColor, 7, 0, ScoreX, ScoreY
        ret
ENDP PrintScore

;-------------------------
;PrintCoins - This is printing the coins on the screen
;-------------------------
;Input:
;       Coins
;Output:
;       The coins on the screen
;Registers:
;       None
;-------------------------
PROC PrintCoins
        PrintStringWithNumbers CoinsText, Coins, GameBackgroundColor,
CoinYellowColor, 0, 1, CoinsX, CoinsY
        ret
ENDP PrintCoins

;-------------------------
;PrintTime - This is printing the time on the screen
;-------------------------
;Input:
;       Time
```

```
;Output:
;       The time on the screen
;Registers:
;       AX, CX, DX, SI, BP
;------------------------
PROC PrintTime
;-----  Get minutes and seconds from time
        xor dx, dx ; Reset dx for div
        mov ax, [Time]
        mov cx, 60 ; Divide by 60 seconds
        div cx

;-----  Convert minutes to text
        push dx ; Save seconds
        mov bp, 2 ; Amount of chars
        lea si, [TimeText] ; Point to the text holder of the Time
        call NumToStr

        mov [byte si], ':' ; Put divider between minutes and seconds
        inc si

;-----  Convert minutes to text
        pop ax ; Get seconds from stack
        mov bp, 2 ; Amount of chars
        call NumToStr

        mov [byte si], 0 ; Set end of the string

        mov [StartX], TimeX
        mov [StartY], TimeY
        lea si, [TimeText]
        mov dl, GameBackgroundColor ; Set text background
        mov dh, RedColor ; Set foreground color, red color
        call PrintColorfulText

        ret
ENDP PrintTime

;------------------------
;NumToStr - This is converting a number to a string
;------------------------
;Input:
;       AX <- Number, BP <- The minimum characters, SI <- Pointer to the
string that will hold the number
;Output:
;       The number in the given string
;Registers:
;       AX, BX, SI, DX, BP
;------------------------
PROC NumToStr
        mov cx, 0 ; Reset counter

@@DigitLoop:
;-----  Divide in 10
        mov bx, 10
        xor dx, dx
        div bx
```

```
        add dx, '0' ; Add the zero in ascii to the digit, to get the digit in
ascii

        push dx ; Save char in stack
        inc cx ; Increase counter

;-----  Check if the number isn't 0
        cmp ax, 0
        jne @@DigitLoop

;-----  Check if has more digits than minimum digits
        cmp cx, bp
        jge @@DigitCharLoop

        sub bp, cx ; Get the amount of needed leading zero's

@@AddLeadingZero:
;-----  Add leading zero to string
        mov [byte si], '0'
        inc si

;-----  Check if done adding leading zero's
        dec bp
        jnz @@AddLeadingZero

@@DigitCharLoop:
        pop dx ; Get char from stack

        mov [byte si], dl ; Set in the string the character
        inc si ; Move to next char

        loop @@DigitCharLoop
        ret
ENDP NumToStr

;-------------------------
;FixMap - This is fixing the map on movement
;-------------------------
;Input:
;       None
;Output:
;       The map on the screen fixed after movement
;Registers:
;       AX, BX, SI
;-------------------------
PROC FixMap
        call DrawCoinsForMap
        call DrawKeyIfNeeded

;-----  Init
        lea si, [MapOrder]
        lea bx, [MapOffset]
        mov [Temp], 0

@@Fix:
;-----  If current part is a flat part
```

```
        cmp [byte si], FlatFloor
        je @@Flat

;-----  If current part is a tall part
        cmp [byte si], TallFloor
        je @@Tall

;-----  If current part is a lava part
        cmp [byte si], LavaFloor
        je @@Lava

;-----  If current part is a hole part
        cmp [byte si], HoleFloor
        je @@Hole

        ret

@@Flat:
;-----  Add a part in the start
        mov ax, [bx]
        DrawRect ax, FloorY, 1, FloorSize, FloorColor

;-----  Remove a part from the end
        mov ax, [bx]
        add ax, MidScreen
        DrawRect ax, FloorY, 1, FloorSize, GameBackgroundColor

        jmp @@Increase

@@Tall:
        call FixTallPart
        jmp @@Flat ; Do as flat as well to fix the flat sides of the part

@@Lava:
        call FixLavaPart
        jmp @@Flat ; Do as flat as well to fix the flat sides of the part

@@Hole:
        call FixHolePart
        jmp @@Flat ; Do as flat as well to fix the flat sides of the part

@@Increase:
;-----  Point to next map part
        inc si
        add bx, 2
        inc [Temp]

        cmp [Temp], 3
        jne @@Fix

        ret
ENDP FixMap

;------------------------
;FixTallPart - This is fixing the tall part on movement
;------------------------
;Input:
```

```
;       BX <- The address of the map offset, Temp <- The part' number
;Output:
;       The tall part on the screen fixed
;Registers:
;       AX, BX, CX, DI
;------------------------
PROC FixTallPart
;-----  Set x
        mov ax, [bx]
        add ax, TallLowerFlatLength + 1

;-----  Set di to point to current map info (height)
        lea di, [MapDataHeight]
        shl [Temp], 1 ; Mul by 2
        add di, [Temp] ; Temp holds the current part number
        shr [Temp], 1 ; Return original value, divide by 2

;-----  Set y
        mov cx, FloorY
        sub cx, [di]

;-----  Set di to point to current map info (width)
        lea di, [MapDataWidth]
        shl [Temp], 1 ; Mul by 2
        add di, [Temp] ; Temp holds the current part number
        shr [Temp], 1 ; Return original value, divide by 2

;-----  Set hole size
        mov dx, [di]

        DrawRect ax, cx, 1, FloorSize, FloorColor ; Add a part

        add ax, dx ; Set x as in the end of the hole which is the length of
the upper part

        DrawRect ax, cx, 1, FloorSize, GameBackgroundColor ; Remove a part

;-----  Set y as the floor y and remove a part there to create the hole
        mov cx, FloorY

        DrawRect ax, cx, 1, FloorSize, FloorColor ; Add a part

        sub ax, dx ; Return x to start of hole

        DrawRect ax, cx, 1, FloorSize, GameBackgroundColor ; Remove a part

        ret
ENDP FixTallPart

;------------------------
;FixHolePart - This is fixing the hole part on movement
;------------------------
;Input:
;       BX <- The address of the map offset, Temp <- The part' number
;Output:
;       The hole part on the screen fixed
;Registers:
```

```
;       AX, CX, DI
;-------------------------
PROC FixHolePart
;-----  Set di to point to current map info (width)
        lea di, [MapDataWidth]
        shl [Temp], 1 ; Mul by 2
        add di, [Temp] ; Temp holds the current part number
        shr [Temp], 1 ; Return original value, divide by 2

;-----  Set hole size
        mov dx, [di]

;-----  Get the size of the lower part
        mov ax, MidScreen
        sub ax, dx
        shr ax, 1

;-----  Add the offset of the part
        add ax, [bx]

;-----  Set y as the floor y and remove a part there to create the hole
        mov cx, FloorY

        DrawRect ax, cx, 1, FloorSize, GameBackgroundColor ; Remove a part

        add ax, dx ; Return x to start of hole

        DrawRect ax, cx, 1, FloorSize, FloorColor ; Add a part

        ret
ENDP FixHolePart

;-------------------------
;FixLavaPart - This is fixing the lava part on movement
;-------------------------
;Input:
;       BX <- The address of the map offset, Temp <- The part' number
;Output:
;       The lava part on the screen fixed
;Registers:
;       AX, DX, CX, DI
;-------------------------
PROC FixLavaPart
;-----  Adding a part before first wall
        mov ax, [bx]
        add ax, LavaLowerFlatLength + 1 ; Set x

        mov dx, FloorY - LavaWallsHeight - 1 ; Set y as under the first flat

        DrawRect ax, dx, 1, LavaWallsHeight, FloorColor ; Add a part

;-----  Adding a lava part after first wall
        add ax, FloorSize ; Set x

        mov dx, FloorY - LavaHeight - 1 ; Set y
        mov cl, [LavaCurrentColor]
```

```
        DrawRect ax, dx, 1, LavaHeight, cl ; Add a part

;-----  Removing a part on top the last part in case there is floor color
there
        mov cx, LavaWallsHeight - LavaHeight ; Set height as the diff between
the lava height and the wall
        sub dx, cx ; Set y
        dec cx

        DrawRect ax, dx, 1, cx, GameBackgroundColor ; Remove a part

;-----  Add a part before second wall
        add ax, LavaLength ; Set x as after the lava
        mov dx, FloorY - LavaWallsHeight - 1 ; Set y

        DrawRect ax, dx, 1, LavaWallsHeight, FloorColor ; Add a part

;-----  Remove a part after second wall
        add ax, FloorSize ; Set x as after the wall

        DrawRect ax, dx, 1, LavaWallsHeight, GameBackgroundColor ; Remove a
part

;-----  Set di to point to current map info and get the width of the upper
part
        lea di, [MapDataWidth]
        shl [Temp], 1 ; Mul by 2
        add di, [Temp] ; Temp holds the current part number
        shr [Temp], 1 ; Return original value, divide by 2

;-----  Add a part in the upper flat
        ; Setting start x
        mov ax, [bx] ; Get part' offset
        add ax, LavaLowerFlatLength + FloorSize + LavaLength / 2 ; Set x in
the middle of the lava
        mov cx, [di] ; Set upper length
        shr cx, 1 ; Div the length by 2
        sub ax, cx ; Set the x axis of the upper flat (Middle Lava - Upper
length / 2)
        shl cx, 1 ; Return original value

        mov dx, FloorY ; Set y axis as the floor

;-----  Set di to point to current map info and get the height of the upper
part
        lea di, [MapDataHeight]
        shl [Temp], 1 ; Mul by 2
        add di, [Temp] ; Temp holds the current part number
        shr [Temp], 1 ; Return original value, divide by 2

        sub dx, [di] ; Set the y as the floor height - the height of the
upper flat

        DrawRect ax, dx, 1, FloorSize, FloorColor ; Add a part

;-----  Remove a part from the end of the upper flat
        add ax, cx ; Add the size of the part to the x to get the end
```

```
        DrawRect ax, dx, 1, FloorSize, GameBackgroundColor ; Remove a part

        call HandleLavaColor

        ret
ENDP FixLavaPart

;------------------------
;HandleLavaColor - This is handles the lava color and returns it
;------------------------
;Input:
;       None
;Output:
;       LavaCurrentColor <- The lava color
;Registers:
;       AX
;------------------------
PROC HandleLavaColor
;-----  Increase and check if reached the maximum blocks to change color
        inc [LavaColorBlocksCounter]
        cmp [LavaColorBlocksCounter], LavaColorBlocksChange
        je @@ChangeColor
        ret

@@ChangeColor:
        mov [LavaColorBlocksCounter], 0 ; Reset counter

;-----  Add the current color offset to the color
        mov al, [LavaCurrentColorOffset]
        add [LavaCurrentColor], al

;-----  Check if reached the end of the color range of the lava, if we did,
set offset to -1 (go back)
        cmp [LavaCurrentColor], LavaColorRangeEnd
        je @@GoBack

;-----  Check if reached the start of the color range, if yes, set offset to
1 (go forward)
        cmp [LavaCurrentColor], LavaColorRangeStart
        je @@GoForward

        ret

@@GoBack:
        mov [LavaCurrentColorOffset], -1 ; Go back next time
        ret

@@GoForward:
        mov [LavaCurrentColorOffset], 1 ; Go forward next time
        ret
ENDP HandleLavaColor

;------------------------
;SendNoteSound - This is sending the current sound note
;------------------------
;Input:
```

```
;       SoundPtr <- Pointer to current sound note
;Output:
;       The sound note playing
;Registers:
;       AX, SI
;------------------------
PROC SendNoteSound
;-----  Prepare speaker for receiving a note
        mov al, 182
        out 43h, al

;-----  Set the note in ax to disassemble it to high and low
        push si
        mov si, [SoundPtr]
        mov ax, [si]
        pop si

;-----  Send to the sound port the high and low of the note
        out 42h, al
        mov al, ah ; Move the high to the low so that it could be sent
        out 42h, al

        ret
ENDP SendNoteSound

;------------------------
;HandleCoinFound - This is handling the coin collection
;------------------------
;Input:
;       (X, Y) <- Interaction point with the coin
;Output:
;       The coin removed from the screen and increased amount of coins
;Registers:
;       AX, BX, CX, DX
;------------------------
PROC HandleCoinFound
;-----  Play coin sound
        ; Set the notes' frequency
        mov [SoundNotes], 4063
        mov [SoundNotes + 2], 2415
        mov [SoundNotes + 4], -1 ; End of notes
        mov [SoundDelay], SoundCoinDelayMax
        call StartSound

        inc [Coins] ; Increase the amout of coins found
        add [Score], CoinValue ; Add to the score to value of the coin

;-----  Check if the coin is in the first part of the screen
        mov ax, [MapOffset]
        add ax, MidScreen
        cmp [X], ax

        jg @@SecondPart

        mov [MapCoin], False
        jmp @@UndrawCoin
```

```
@@SecondPart:
;-----  If the coin isn't in the first part, it must be in the second part
because the player cannot reach the coin in the third part so just remove the
coin
        mov [MapCoin + 1], False

@@UndrawCoin:
;-----  Get the x and y of the block that will be twice the size of the coin
        mov ax, [X]
        mov bx, [Y]

        sub ax, CoinPicWidth
        sub bx, CoinPicHeight

;-----  Set block size as twice the size of the coin
        mov cx, CoinPicWidth * 2
        mov dx, CoinPicHeight * 2

;-----  Drawing the block that is twice that size of the coin where the
middle of the block will be the interaction point so that the block will
remove the coin
        DrawRect ax, bx, cx, dx, GameBackgroundColor

        call PrintCoins
        call PrintScore

        mov cx, 0FFFFh
        ret
ENDP HandleCoinFound

;------------------------
;StartSound - This is starting the sound
;------------------------
;Input:
;       SoundNotes <- The notes of the sound
;Output:
;       The sound playing in the speaker
;Registers:
;       AX
;------------------------
PROC StartSound
        mov [SoundPtr], offset SoundNotes ; Set the pointer to the notes list
        call SendNoteSound

;-----  Get the current status from port 61h
        in al, 61h

;-----  Set 0 and 1 bits to start the note sound and send back to the port
        or al, 00000011b
        out 61h, al

        mov [SoundStarted], True ; Set sound status to started
        mov [SoundCounter], 0

        ret
ENDP StartSound
```

```
;------------------------
;HandleSound - This is handling the sound and switches notes when needed
;------------------------
;Input:
;       SoundNotes <- The notes of the sound, SoundDelay <- The delay between
the sounds
;Output:
;       The sound playing in the speaker
;Registers:
;       AX, SI
;------------------------
PROC HandleSound
;-----  Check if the sound hasn't started, if not, return
        cmp [SoundStarted], False
        je @@Return

;-----  Check if reached max sound delay, if didn't just increase and return
        mov ax, [SoundDelay]
        inc [SoundCounter]
        cmp [SoundCounter], ax
        jne @@Return

;-----  Move the pointer of the notes to the next note
        add [SoundPtr], 2
        call SendNoteSound

;-----  Get current note
        push si
        mov si, [SoundPtr]
        mov ax, [si]
        pop si

;-----  Check if reached the end of the list
        cmp ax, -1
        je @@Stop

;-----  Reset counter
        mov [SoundCounter], 0
        jmp @@Return

@@Stop:
        mov [SoundCounter], 0 ; Reset sound counter
        mov [SoundStarted], False ; Set sound to be stopped

;-----  Get the current status from port 61h
        in al, 61h

;-----  Turn off bits 0 and 1 in order to stop the sound and send back to the
port
        and al, 11111100b
        out 61h, al

@@Return:
        ret
ENDP HandleSound

;------------------------
```

```
;PlayerJump - This is making the player jump
;------------------------
;Input:
;       PlayerJumping <- 1 To start jumping or 0 otherwise
;Output:
;       The player jumping
;Registers:
;       AX, CX, DX
;------------------------
PROC PlayerJump
;-----  Check if need to jump, if not return, else continue
        cmp [PlayerJumping], False
        je @@Return

;-----  Set in dx, the current delay(max delay - currrent velocity)
        mov dx, MaxVDelay
        sub dx, [PlayerVVelocity]

;-----  Check if the current delay reached the needed delay, if it didn't
increase and return, else continue
        cmp dx, [CurrentVDelay]
        je @@VelocityCheck

;-----  Increase and return
        inc [CurrentVDelay]
        ret

@@VelocityCheck:
;-----  Resetting the current delay
        mov [CurrentVDelay], 0

;-----  Check if reached max velocity, if reached just move without
increasing speed, else continue
        cmp [PlayerVVelocity], MaxVVelocity
        je @@CheckJump

        inc [PlayerVVelocity] ; Increasing velocity

@@CheckJump:
        mov cx, PlayerWidth ; Set the amount of points that are needed to be
checked which is the width of the player

;-----  Check if reached floor, first point, start of player
        mov ax, [PlayerX]
        mov [X], ax

        mov ax, [PlayerY]
        dec ax
        mov [Y], ax

@@CheckLoop:
        call GetPixel ; Get the current pixel

;-----  Check if the current pixel is a wall, if it is, stop the player
        cmp [Color], FloorColor
        je @@StopJumping
```

```
;-----  Check if touched black which is the border of a coin
        cmp [Color], CoinBorderColor
        je @@CoinFound

;-----  Check if took key
        cmp [Color], KeyBorderColor
        je @@KeyFound

        inc [X] ; Go to next point
        loop @@CheckLoop

@@Jump:
        dec [PlayerY] ; Jumping

;-----  Changed if reached the max y jump
        mov dx, [PlayerJumpMaxY]
        cmp [PlayerY], dx
        je @@StopJumping

@@Return:
        ret

@@StopJumping:
        mov [PlayerJumping], False ; Stop jumping
        mov [PlayerStopJumping], True ; Make player to stop jumping smoothly
        jmp @@Return

@@CoinFound:
        call HandleCoinFound
        jmp @@Jump

@@KeyFound:
        call HandleKeyFound
        jmp @@Return
ENDP PlayerJump

;-------------------------
;PlayerStopJump - This is making the player to stop jumping smoothly
;-------------------------
;Input:
;       PlayerStopJump <- 1 To stop jumping or 0 otherwise
;Output:
;       The player stops jumping
;Registers:
;       AX, CX, DX
;-------------------------
PROC PlayerStopJump
;-----  Check if need to stop jumping, if not return, else continue
        cmp [PlayerStopJumping], False
        je @@Return

;-----  Set in dx, the current delay(max delay - currrent velocity)
        mov dx, MaxVDelay
        sub dx, [PlayerVVelocity]

;-----  Check if the current delay reached the needed delay, if it didn't
increase and return, else continue
```

```
        cmp dx, [CurrentVDelay]
        je @@VelocityCheck

;-----  Increase and return
        inc [CurrentVDelay]
        ret

@@VelocityCheck:
;-----  Resetting the current delay
        mov [CurrentVDelay], 0

;-----  Check if reached zero velocity, if reached stop, else continue
        cmp [PlayerVVelocity], 0
        je @@StopJumping

        dec [PlayerVVelocity] ; Decreasing velocity

@@CheckJump:
        mov cx, PlayerWidth ; Set the amount of points that are needed to be
checked which is the width of the player

;-----  Check if reached floor, first point, start of player
        mov ax, [PlayerX]
        mov [X], ax

        mov ax, [PlayerY]
        dec ax
        mov [Y], ax

@@CheckLoop:
        call GetPixel ; Get the current pixel

;-----  Check if the current pixel is a wall, if it is, stop the player
        cmp [Color], FloorColor
        je @@StopJumping

;-----  Check if touched black which is the border of a coin
        cmp [Color], CoinBorderColor
        je @@CoinFound

;-----  Check if took key
        cmp [Color], KeyBorderColor
        je @@KeyFound

        inc [X] ; Go to next point
        loop @@CheckLoop

@@Jump:
        dec [PlayerY] ; Jumping

        cmp [PlayerY], MaxY ; Changed if reached the top
        je @@StopJumping

@@Return:
        ret

@@StopJumping:
```

```
        mov [PlayerStopJumping], False ; Make player to stop jumping smoothly
        jmp @@Return

@@CoinFound:
        call HandleCoinFound
        jmp @@Jump

@@KeyFound:
        call HandleKeyFound
        jmp @@Return
ENDP PlayerStopJump

;------------------------
;PlayerGravity - This is making the player to fall down
;------------------------
;Input:
;       None
;Output:
;       The player falling down
;Registers:
;       AX, CX, DX
;------------------------
PROC PlayerGravity
;-----  Check if jumping, if yes, return, else continue
        cmp [PlayerJumping], False
        jne @@MidReturn

;-----  Check if stop to jumping, if stopping, return, else continue
        cmp [PlayerStopJumping], False
        jne @@MidReturn

;-----  Set in dx, the current delay(max delay - currrent velocity)
        mov dx, MaxVDelay
        sub dx, [PlayerVVelocity]

;-----  Check if the current delay reached the needed delay, if it didn't
increase and return, else continue
        cmp dx, [CurrentVDelay]
        je @@VelocityCheck

;-----  Increase and return
        inc [CurrentVDelay]

@@MidReturn:
        ret

@@VelocityCheck:
;-----  Resetting the current delay
        mov [CurrentVDelay], 0

;-----  Check if reached max velocity, if reached just move without
increasing speed, else continue
        cmp [PlayerVVelocity], MaxVVelocity
        je @@GravityCheck

        inc [PlayerVVelocity] ; Increasing velocity
```

```
@@GravityCheck:
        mov cx, PlayerWidth ; Set the amount of points that are needed to be
checked which is the width of the player

;-----  Check if reached floor, first point, start of player
        mov ax, [PlayerX]
        mov [X], ax

        mov ax, [PlayerY]
        add ax, PlayerHeight + PlayerFloorDist
        mov [Y], ax

@@CheckLoop:
        call GetPixel ; Get the current pixel

;-----  Check if the current pixel is a wall, if it is, stop the player
        cmp [Color], FloorColor
        je @@StopGravity

;-----  Check if touched black which is the border of a coin
        cmp [Color], CoinBorderColor
        je @@CoinFound

;-----  Check if took key
        cmp [Color], KeyBorderColor
        je @@KeyFound

        cmp [Color], LavaColorRangeStart
        je @@PlayerDie

        inc [X] ; Go to next point
        loop @@CheckLoop

@@Gravity:
        mov [PlayerOnGround], False ; Set player to be not on ground, just
found it using the conditions
        inc [PlayerY] ; Go down

@@Return:
        ret

@@StopGravity:
        mov [PlayerVVelocity], 0 ; Reset velocity
        mov [CurrentVDelay], 0 ; Reset MaxHDelay
        mov [PlayerOnGround], True

        jmp @@Return

@@CoinFound:
        call HandleCoinFound
        jmp @@Gravity

@@PlayerDie:
        mov [PlayerDead], True
        jmp @@Return

@@KeyFound:
```

```
        call HandleKeyFound
        jmp @@Return
ENDP PlayerGravity

;------------------------
;HandleKeyFound - This is handling a key collection
;------------------------
;Input:
;       None
;Output:
;       KeyFound <- True and the sound of a level pass
;Registers:
;       None
;------------------------
PROC HandleKeyFound
;-----  Play key sound
        ; Set the notes' frequency
        mov [word SoundNotes], 1436
        mov [word SoundNotes + 2], 1355
        mov [word SoundNotes + 4], 1207
        mov [word SoundNotes + 6], 1140
        mov [word SoundNotes + 8], -1 ; End of notes
        mov [SoundDelay], SoundKeyDelayMax
        call StartSound

;-----  Set the key status to found
        mov [KeyFound], True
        ret
ENDP HandleKeyFound

;------------------------
;CheckPlayerMove - This is checking if the player has moved and moving it if
needed
;------------------------
;Input:
;       None
;Output:
;       The player to be moved but not redrawn
;Registers:
;       AX
;------------------------
PROC CheckPlayerMove
        in al, 60h ; Get scan code from keybaord port
        push ax

;-----  Check if the right arrow is pressed
        cmp al, RightArrowScanCode
        je @@MoveRight

;-----  Check if the left arrow is pressed
        cmp al, LeftArrowScanCode
        je @@MoveLeft

;-----  Check if the left arrow is released
        cmp al, ReleasedLeftArrowScanCode
        je @@StopMoving
```

```
;-----  Check if the left arrow is released
        cmp al, ReleasedRightArrowScanCode
        je @@StopMoving

        cmp [PlayerStop], 1
        je @@Return

;-----  If the player has velocity but none of the above was pressed or
released, move by the x delta
        cmp [PlayerXDelta], 1
        je @@MoveRight

        cmp [PlayerXDelta], -1
        je @@MoveLeft

        jmp @@Return

@@MoveRight:
        mov [PlayerStop], False

        mov [PlayerXDelta], 1 ; Set delta, go right
        call CheckIfReachedWall

        call MovePlayer
        jmp @@Return

@@MoveLeft:
        mov [PlayerStop], False

        mov [PlayerXDelta], -1 ; Set delta, go left
        call CheckIfReachedWall

        call MovePlayer
        jmp @@Return

@@StopMoving:
        mov [PlayerStop], True

@@Return:
        pop ax ; Restore al to be used for checks later
        ret
ENDP CheckPlayerMove

;------------------------
;MovePlayer - This is handling whatever the player should move or just the
map and it's moving what's necessary + it's handling the acceleration of the
player
;------------------------
;Input:
;       PlayerXDelta <- The delta of the player, right or left
;Output:
;       The player moved
;Registers:
;       DX
;------------------------
PROC MovePlayer
;-----  Check if have delta x
```

```
        cmp [PlayerXDelta], 0
        je @@Return

;-----  Set in dx, the current delay(max delay - currrent velocity)
        mov dx, MaxHDelay
        sub dx, [PlayerHVelocity]

;-----  Check if the current delay reached the needed delay, if it didn't
increase and return, else continue
        cmp dx, [CurrentHDelay]
        je @@VelocityCheck

;-----  Increase and return
        inc [CurrentHDelay]
        ret

@@VelocityCheck:
        call ChangePlayerFrame

;-----  Resetting the current delay
        mov [CurrentHDelay], 0

;-----  Check if reached max velocity, if reached just move without
increasing speed, else continue
        cmp [PlayerHVelocity], MaxHVelocity
        je @@MapCheck

        inc [PlayerHVelocity] ; Increasing velocity

@@MapCheck:
        cmp [PlayerX], ScreenWidth / 2 - PlayerWidth / 2 ; Check if player in
middle of screen, if it is, move map
        je @@Map

@@Move:
        mov dx, [PlayerXDelta]
        add [PlayerX], dx ; Move player according to the delta
        jmp @@Return

@@Map:
;-----  Check if the player is going right, if it is so move map, else just
regular move
        cmp [PlayerXDelta], 1
        jne @@Move

;-----  Move map and fix it on the screen
        call MoveMap
        call FixMap
        jmp @@Return

@@Return:
        ret
ENDP MovePlayer

;------------------------
;CheckIfReachedWall - This is checking if the player hit a wall
;------------------------
```

```
;Input:
;       PlayerXDelta <- The delta of the player, right or left
;Output:
;       The player stopped or if a coin found
;Registers:
;       CX, DX
;------------------------
PROC CheckIfReachedWall
        mov cx, PlayerHeight ; Check all the points in the player' side(the
amount it's the player's height)

;-----  Check if moving right
        cmp [PlayerXDelta], 1
        je @@Right

;-----  Check if moving left
        cmp [PlayerXDelta], -1
        je @@Left

        jmp @@Return

@@Right:
;-----  Prepare for check by setting the x of the points and the initial y
axis
        mov dx, [PlayerX]
        add dx, PlayerWidth + DistanceFromWall
        mov [X], dx

        mov dx, [PlayerY]
        mov [Y], dx

        jmp @@CheckLoop

@@Left:
;-----  If reached the end of the screen (left side), stop player
        cmp [PlayerX], MinX + DistanceFromWall
        je @@Stop

;-----  Prepare for check by setting the x of the points and the initial y
axis
        mov dx, [PlayerX]
        sub dx, DistanceFromWall
        mov [X], dx

        mov dx, [PlayerY]
        mov [Y], dx

@@CheckLoop:
;-----  Get the current pixel
        call GetPixel

;-----  If touched a wall, stop the player
        cmp [Color], FloorColor
        je @@Stop

;-----  Check if touched black which is the border of a coin
        cmp [Color], CoinBorderColor
```

```
        je @@CoinFound

;-----  Check if took key
        cmp [Color], KeyBorderColor
        je @@KeyFound

        inc [Y] ; Go to next point
        loop @@CheckLoop
        jmp @@Return

@@Stop:
        mov [PlayerStop], False ; Setting player to be stopped
        mov [PlayerXDelta], 0 ; Resetting delta x
        mov [PlayerHVelocity], 0 ; Reset velocity
        mov [CurrentHDelay], 0 ; Reset delay

        jmp @@Return

@@CoinFound:
        call HandleCoinFound
        jmp @@Return

@@KeyFound:
        call HandleKeyFound

@@Return:
        ret
ENDP CheckIfReachedWall

;-------------------------
;StopPlayer - This is stopping the player using acceleration
;-------------------------
;Input:
;       PlayerStop <- Should the player stop or not
;Output:
;       The player stopped on the screen
;Registers:
;       DX
;-------------------------
PROC StopPlayer
        call CheckIfReachedWall

;-----  Check if should stop player, if shouldn't return, else continue
        cmp [PlayerStop], False
        je @@Return

;-----  Set in dx, the current delay(max delay - currrent velocity)
        mov dx, MaxHDelay
        sub dx, [PlayerHVelocity]

;-----  Check if the current delay reached the needed delay, if it didn't
increase and return, else continue
        cmp dx, [CurrentHDelay]
        je @@VelocityCheck

;-----  Increase and return
        inc [CurrentHDelay]
```

```
        ret

@@VelocityCheck:
;-----  Resetting the current delay
        mov [CurrentHDelay], 0

;-----  Check if reached zero velocity, if reached stop, else continue
        cmp [PlayerHVelocity], 0
        je @@Stop

        dec [PlayerHVelocity] ; Decreasing velocity

@@MapCheck:
        cmp [PlayerX], ScreenWidth / 2 - PlayerWidth / 2 ; Check if player in
middle of screen, if it is, move map
        je @@Map

@@Move:
        mov dx, [PlayerXDelta]
        add [PlayerX], dx ; Move player according to the delta
        jmp @@Return

@@Map:
;-----  Check if the player is going right, if it is so move map, else just
regular move
        cmp [PlayerXDelta], 1
        jne @@Move

;-----  Move map and fix it on the screen
        call MoveMap
        call FixMap
        jmp @@Return

@@Stop:
        mov [PlayerStop], False ; Setting player to be stopped
        mov [PlayerXDelta], 0 ; Resetting delta x
        mov [PlayerHVelocity], 0 ; Reset velocity
        mov [CurrentHDelay], 0 ; Reset delay

@@Return:
        ret
ENDP StopPlayer

;------------------------
;Help - This is displaying the help screen
;------------------------
;Input:
;       None.
;Output:
;       The help screen displayed
;Registers:
;       AX
;------------------------
PROC Help
;-----  Draw the help PCX image
        CopyString HelpFileName, FileName, HelpNameLen
        mov [StartX], 0
```

```
        mov [StartY], 0

        call DrawPCX

;-----  Wait for keyboard press
        xor ah, ah
        int 16h

        ret
ENDP Help

;-------------------------
;About - This is displaying the about screen
;-------------------------
;Input:
;       None
;Output:
;       The about screen displayed
;Registers:
;       AX
;-------------------------
PROC About
;-----  Draw the about PCX image
        CopyString AboutFileName, FileName, AboutNameLen
        mov [StartX], 0
        mov [StartY], 0

        call DrawPCX

;-----  Wait for keyboard press
        xor ah, ah
        int 16h

        ret
ENDP About

INCLUDE 'DRAW.INC'

                END Start
```

Code

File Name: DRAW.INC

Contents: The file contains the draw functions of the game like drawing PCX images, drawing rectangles, sprites and more..

```
;-----------------------------------------
; PURPOSE : Key breaker' draw tools
; SYSTEM  : Turbo Assembler Ideal Mode
; AUTHOR  : Almog Hamdani
;-----------------------------------------

;-----  This segment will hold the contents of the file we open
SEGMENT FILEBUF para public
        DB 65200 dup (?)
ENDS

;------------------------
;PrintColorfulText - This is printing text in a specific background and
foreground color in graphic mode
;------------------------
;Input:
;       SI <- Pointer to the string, DL <- Background color, DH <- Foreground
color, StartX, StartY
;Output:
;       The text in the colors wanted on the screen
;Registers:
;       AX, CX, BX, DI, DX, ES
;------------------------
PROC PrintColorfulText
;-----  Set in x and y the start x and start y
        mov ax, [StartX]
        mov [X], ax

        mov ax, [StartY]
        mov [Y], ax

@@CharLoop:
;-----  Set extra segment as ascii table in memory
        mov ax, 0F000h
        mov es, ax

;-----  Set in bx, the character * 8 + ascii table offset (Each character
takes up 8 bytes)
        mov bx, 0FA6Eh ; Initial ascii table in memory
        xor cx, cx
        mov cl, [byte si]
        shl cx, 3
        add bx, cx

;-----  Load the bitmap to a variable
        lea di, [TextBitmap] ; Set in di the text bitmap where we hold the
bitmap to

        mov cx, 4 ; Get the rows (2 at the time)

@@GetBitmap:
        mov ax, [es:bx] ; Get two rows from the bitmap

;-----  Fix the bug where the two rows are swapped, so swap them back
        mov [TempByte], al
        mov al, ah
        mov ah, [TempByte]
```

```
        mov [di], ax ; Put in the bitmap var the 2 rows

        add bx, 2
        add di, 2

        loop @@GetBitmap

;-----  Set extra segment as video memory
        mov ax, 0A000h
        mov es, ax

        mov bx, 8 / 2 ; Set as the bitmap' rows size / 2 (Printing 2 rows at
1 time)
        lea di, [TextBitmap] ; Set di to point at the bitmap

@@BitmapPrint:
        mov cx, 2 ; Set as the rows in one word

@@BitmapRowPrint:
        shl [word di], 1 ; Shift left the bitmap
        jc @@Foreground ; If the carry is on then it means we need to print
the foreground color

;-----  Print background color if carry is 0
        mov [Color], dl

        push di
        call PutPixel
        pop di

        jmp @@IncreaseX

@@Foreground:
;-----  Print foreground color
        mov [Color], dh

        push di
        call PutPixel
        pop di

@@IncreaseX:
        inc [X] ; Increase the x axis

;-----  Check if the line is finished
        mov ax, [StartX]
        add ax, 8
        cmp [X], ax

        jb @@BitmapRowPrint

;-----  Reset x
        mov ax, [StartX]
        mov [X], ax

        inc [Y]
```

```
        loop @@BitmapRowPrint ; If not finished 2 rows go back again

;-----  Check if finished printing character
        add di, 2
        dec bx
        jnz @@BitmapPrint

@@NextChar:
        inc si ; Move to next char

        add [StartX], 8 ; Adding to the start x the size of each character
        add [X], 8 ; Set x in the next char space

;-----  Reset y axis
        mov ax, [StartY]
        mov [Y], ax

;-----  Check if reached the end of the string
        cmp [byte si], 0
        je @@Return
        jmp @@CharLoop

@@Return:
        ret
ENDP PrintColorfulText

;-------------------------
;DrawRectangle - This is drawing a rectangle on the screen
;-------------------------
;Input:
;       (StartX, StartY) <- Position of the rectangle, (SizeWidth,
SizeHeight) <- Size of the rectangle, Color <- The color of the rectangle
;Output:
;       The rectangle drawn on the screen
;Registers:
;       AX, CX
;-------------------------
PROC DrawRectangle
;-----  Check if start point is out of bounds, if yes return
        cmp [StartX], ScreenWidth
        jb @@CheckNegX
        ret ; If out of bounds, return

@@CheckNegX:
;-----  Check if the x is negetive
        cmp [StartX], 0
        jl @@NegX
        jmp @@CheckOverflow

@@NegX:
;-----  If negetive reset it and remove it's value from the length
        mov ax, [StartX]
        add [SizeWidth], ax
        mov [StartX], 0

@@CheckOverflow:
        mov ax, [StartX]
```

```
        add ax, [SizeWidth]
        cmp ax, ScreenWidth

        jbe @@SetY

        mov ax, ScreenWidth
        sub ax, [StartX]
        mov [SizeWidth], ax

@@SetY:
;-----  Set Y as the start of the rectangle
        mov ax, [StartY]
        mov [Y], ax

@@Rectangle:
;-----  Set X as the start of the rectangle
        mov ax, [StartX]
        mov [X], ax

        mov cx, [SizeWidth] ; Set for loop

@@DrawHLine:
        call PutPixel
        inc [X] ; Set next pixel
        loop @@DrawHLine

        inc [Y] ; Set next line

;-----  Check if didn't reach last line
        mov ax, [SizeHeight]
        add ax, [StartY]
        cmp [Y], ax
        jbe @@Rectangle

        ret
ENDP DrawRectangle

;------------------------
;PaintScreen - Paints the entire screen in 1 color
;------------------------
;Input:
;       Color <- The color to paint with
;Output:
;       The entire screen with the color
;Registers:
;       None
;------------------------
PROC PaintScreen
        mov [X], 0
        mov [Y], 0

@@Paint:
        call PutPixel

        inc [X]
        cmp [X], 320 * 200
        jne @@Paint
```

```
        ret
ENDP PaintScreen

;-------------------------
;ReadPCX - Reads a PCX file
;-------------------------
;Input:
;       FileName <- The PCX file name
;Output:
;       FILEBUF <- The contents of the file, FileSize <- The size of the file
;Registers:
;       AX, BX, CX, DX
;-------------------------
PROC ReadPCX
;-----  Try open file using interrupt
        mov ah, 3Dh ; Interrupt entry
        mov al, 0h ; Open for read only
        lea dx, [FileName]
        int 21h

;-----  Error occurred
        jc @@Error

        mov [FileHandle], ax

;-----  Set file pointer to it's end
        mov ah, 42h  ; Interrupt Entry
        mov al, 2h ; Setting offset from end of file
        mov bx, [FileHandle]

        ; Set offset to 0:0
        xor dx, dx
        xor cx, cx

        int 21h

;-----  Error occurred
        jc @@Error

;-----  Set file size
        mov [FileSize], ax

;-----  Return file pointer to start
        mov ah, 42h  ; Interrupt Entry
        mov al, 0h ; Setting offset from start of file
        mov bx, [FileHandle]

        ; Set offset to 0:0
        xor dx, dx
        xor cx, cx

        int 21h

;-----  Error occurred
        jc @@Error
```

```
;-----  Read all file into FILEBUF segment
        push ds ; Save original data segment

        mov cx, [FileSize] ; Tell interrupt to read all the file
        mov bx, [FileHandle]

        ; Set data segment as FILEBUF segment
        mov ax, FILEBUF
        mov ds, ax
        xor dx, dx ; Set offset 0

        mov ah, 3Fh ; Interrupt entry

        int 21h

;-----  Error occurred
        jc @@Error

        pop ds ; Return data segment

;-----  Close file
        mov ah, 3Eh ; Interrupt entry
        mov bx, [FileHandle]

        int 21h

;-----  Error occurred
        jc @@Error
        ret

@@Error:
;-----  Set text mode
        mov ax, 3h
        int 10h

;-----  Print error
        mov ah, 9h
        lea dx, [PCXErrorMSG]
        int 21h
        jmp Exit
ENDP

;-------------------------
;DrawPCX - Draws a pcx to a specific location on the screen (Only supports
default pallete)
;-------------------------
;Input:
;       StartX <- The X to start draw in, StartY <- The Y to start draw in,
FileName <- The PCX file name
;Output:
;       The PCX on the screen
;Registers:
;       AX, BX, CX, DX, SI, DI, ES
;-------------------------
PROC DrawPCX
        call ReadPCX
```

```
;-----  Set extra segment as file buffer
        mov ax, FILEBUF
        mov es, ax
        mov si, 128 ; Set si to point at the start of the image data

;-----  Get image' width
        mov ax, [es:8h] ; Width is in the 8h pos
        inc ax ; Plus 1
        mov [ImageWidth], ax

;-----  Get image' height
        mov ax, [es:0Ah] ; Height is in the Ah pos
        inc ax ; Plus 1
        mov [ImageHeight], ax

;-----  Set starting position
        mov ax, [StartX]
        mov [X], ax
        mov ax, [StartY]
        mov [Y], ax

@@GetByte:
        mov al, [es:si]
        mov [Color], al
        inc si ; Point to next byte
        cmp [Color], 192 ; Check if there is a seq
        jb @@DrawNormal

        sub [Color], 192 ; In cx there is the amount of pixels to write
        xor ch, ch ; Reset cx
        mov cl, [Color] ; Set in cx the length to use for loop

        mov al, [es:si]
        mov [Color], al
        inc si ; Point to next byte

@@DrawSeq:
;-----  Set extra segment as video memory
        mov ax, 0A000h
        mov es, ax

        call PutPixel

;-----  Set extra segment as file buffer
        mov ax, FILEBUF
        mov es, ax

        inc [X] ; Set next x

;-----  Check if we got to the end of the line
        mov bx, [StartX]
        add bx, [ImageWidth]
        cmp bx, [X]
        je @@NewLine

        loop @@DrawSeq
```

```
        jmp @@GetByte

@@DrawNormal:
;-----  Set extra segment as video memory
        mov ax, 0A000h
        mov es, ax

        call PutPixel

;-----  Set extra segment as file buffer
        mov ax, FILEBUF
        mov es, ax

        inc [X] ; Set next x

;-----  Check if we got to the end of the line
        mov bx, [StartX]
        add bx, [ImageWidth]
        cmp bx, [X]
        je @@NewLine

        jmp @@GetByte

@@NewLine:
        mov ax, [StartX]
        mov [X], ax ; Set x to start

        inc [Y] ; Set next line

;-----  Check if the width of the image is odd or even, if it is odd,
increase ptr by 1 because of 0 in the end, else continue
        mov ax, [ImageWidth]
        and ax, 1
        cmp ax, 1
        jne @@LineCheck

        inc si

@@LineCheck:
;-----  Check if the line was the last line
        mov bx, [StartY]
        add bx, [ImageHeight]
        cmp bx, [Y]
        je @@End

        jmp @@GetByte

@@Error:
;-----  Set text mode
        mov ax, 3h
        int 10h

;-----  Print error
        mov ah, 9h
        lea dx, [PCXErrorMSG]
        int 21h
        jmp Exit
```

```
@@End:
;-----  Return video memory
        mov ax, 0A000h
        mov es, ax

        ret
ENDP DrawPCX

;-------------------------
;ClearSprite - Removes a sprite from the screen
;-------------------------
;Input:
;       X, Y, Color, CX <- Sprite' height, DX <- Sprite' width
;Output:
;       The sprite removed from the screen
;Registers:
;       CX, DX
;-------------------------
PROC ClearSprite
        push dx ; Save sprite' original width

@@Cycle:
        call PutPixel
        inc [X] ; Move pixel' x by 1
        dec dx ; Decrease width by 1
        jnz @@Cycle ; While we didn't reach 0 (End of line), keep on cycle

        pop dx ; Return sprite' original width
        push dx ; Save width in stack

        sub [X], dx ; Set ax to start of line again
        inc [Y] ; Point to a new line

        dec cx ; Decrease sprite' height
        jnz @@Cycle

        pop dx ; Clean stack
        ret
ENDP ClearSprite

;-------------------------
;PrintSprite - Prints a sprite to the screen
;-------------------------
;Input:
;       StartX, StartY, SizeWidth, SizeHeight, SI <- Sprite' offset
;       Carry flag on -> Negative and Overflow check
;       ImageFlipped -> Is the image should be drawn flipped
;       Note: Flipped image and image checks ain't working together, you can
only use one of of them
;Output:
;       The sprite drawn on the screen
;Registers:
;       AX, SI, DX
;-------------------------
PROC PrintSprite
```

```
        mov [SkipColumns], 0 ; Reset the amount of columns that needed to be
skipped

        jnc @@SetStartPosition

;-----  Check if start point is out of bounds, if yes return
        cmp [StartX], ScreenWidth
        jl @@OutOfBounds
        ret ; If out of bounds, return

@@OutOfBounds:
        mov [ImageFlipped], False ; Reset image flipped

;-----  Check if the drawen sprite will not appear on the screen by adding to
it's x value it's width
        mov ax, [StartX]
        add ax, [SizeWidth]
        cmp ax, 0
        jg @@NegX
        ret

@@NegX:
;-----  Check if we got a negative x, if we did, we need to handle it
        cmp [StartX], 0
        jg @@CheckOverflow

@@FixNegX:
;-----  Get the absolute number of the amount of columns needed to be skipped
        mov ax, [StartX]
        xor ax, 0FFFFh
        sub ax, 0FFFFh

        mov [SkipColumns], ax ; Set the amount of columns needed to be
skipepd

;-----  Remove from the width the wanted columns that needed to be skipped
        sub [SizeWidth], ax

        mov [StartX], 0 ; Reset the x value to 0

        add si, [SkipColumns] ; Skip the amount of columns needed from the
start
        jmp @@SetStartPosition

@@CheckOverflow:
;-----  Check for overflow, that the sprite is drawing in the other side as
well
        mov ax, [StartX]
        add ax, [SizeWidth]
        cmp ax, ScreenWidth
        jle @@SetStartPosition

@@FixOverflow:
;-----  Get the amount of columns that need to be skipped by subbing the
screen size from the overflow amount
        sub ax, ScreenWidth
        mov [SkipColumns], ax
```

```
;-----  Get the new width of the sprite without the columns that are going to
be skipped
        sub [SizeWidth], ax

@@SetStartPosition:
;-----  Set the starting position of printing the sprite
        mov ax, [StartY]
        mov [Y], ax

        mov ax, [StartX]
        mov [X], ax

        mov dx, [SizeWidth] ; Set width for line

        mov [Temp], 1 ; Set normal offset

;-----  If the image is flipped, reverse offset
        cmp [ImageFlipped], True
        jne @@Cycle

        mov [Temp], -1
        add si, [SizeWidth] ; Start from the end of the line
        dec si

@@Cycle:
        mov al, [si]
        mov [Color], al ; Get pixel' color and set in cl
        call PutPixel
        add si, [Temp] ; Point to next pixel' color by adding the offset that
is saved in the temp byte var
        inc [X] ; Move pixel' x by 1

        dec dx ; Decrease width by 1
        jnz @@Cycle ; While we didn't reach 0 (End of line), keep on cycle

        cmp [ImageFlipped], True
        jne @@Normal

@@HandleFlipped:
        add si, [SizeWidth] ; Go back to end of current line
        add si, [SizeWidth] ; Start from the end of the new line

@@Normal:
        add si, [SkipColumns] ; Skip the amount of columns needed

        mov ax, [StartX]
        mov [X], ax ; Set ax to start of line again
        inc [Y] ; Point to a new line

        mov dx, [SizeWidth] ; Set width for line

        dec [SizeHeight] ; Decrease sprite' height
        jnz @@Cycle

        ret
ENDP PrintSprite
```

```
;------------------------
;PutPixel - Write pixel
;------------------------
;Input:
;       X, Y, Color
;Output:
;       The pixel on the screen
;Registers:
;       AX, DI, ES
;------------------------
PROC PutPixel
        push ax
        mov di, [Y]
        mov ax, [Y]
        shl di, 8
        shl ax, 6
        add di, ax
        add di, [X]
        mov al, [Color]
        mov [es:di], al
        pop ax
        ret
ENDP PutPixel

;------------------------
;GetPixel - Read pixel
;------------------------
;Input:
;       X, Y
;Output:
;       Color
;Registers:
;       AX, DI, ES
;------------------------
PROC GetPixel
        push ax
        mov di, [Y]
        mov ax, [Y]
        shl di, 8
        shl ax, 6
        add di, ax
        add di, [X]
        mov al, [es:di]
        mov [Color], al
        pop ax
        ret
ENDP GetPixel
```

Code

File Name: PICS.INC

Contents: The file contains the sprites that are used in the game like the character, the select dot, the coin' frames and more..

```
TimePic db 53,53,53,53,53,53,53,53,53,53,53,53,53
        db 53,53,40,40,40,40,40,40,40,40,40,53,53
        db 53,40,53,53,53,53,53,53,53,53,53,40,53
        db 53,40,53,53,53,53,40,53,53,53,53,40,53
        db 53,40,53,53,53,53,40,53,53,53,53,40,53
        db 53,40,53,53,53,53,40,53,53,53,53,40,53
        db 53,40,53,53,53,53,40,40,40,40,53,40,53
        db 53,40,53,53,53,53,53,53,53,53,53,40,53
        db 53,40,53,53,53,53,53,53,53,53,53,40,53
        db 53,40,53,53,53,53,53,53,53,53,53,40,53
        db 53,40,53,53,53,53,53,53,53,53,53,40,53
        db 53,53,40,40,40,40,40,40,40,40,40,53,53
        db 53,53,53,53,53,53,53,53,53,53,53,53,53

TimePicHeight equ 13
TimePicWidth  equ 13


PlayerPic          db 53,53,53,53,53,53,53,53,53,53,53,53,53
                   db 53,53,53,17,17,18,18,18,17,53,53,53,53
                   db 53,53,17,18,18,18,18,18,18,18,53,53,53
                   db 53,17,18,18,18,226,129,226,18,226,17,53,53
                   db 53,17,18,18,226,226,18,226,18,226,18,53,53
                   db 53,17,17,18,18,18,17,18,90,18,18,53,53
                   db 53,17,18,18,18,18,90,90,90,17,17,53,53
                   db 53,17,18,17,17,90,90,15,90,90,18,53,53
                   db 53,53,17,64,65,90,77,19,90,90,53,53,53
                   db 53,53,53,18,65,90,90,90,90,65,53,53,53
                   db 53,53,22,24,17,65,90,90,136,53,53,53,53
                   db 53,19,25,15,28,17,185,185,138,209,53,53,53
                   db 53,19,24,25,25,17,138,138,140,138,53,53,53
                   db 53,19,17,17,17,138,140,140,140,138,53,53,53
                   db 53,22,27,19,209,138,164,164,164,209,53,53,53
                   db 53,17,25,28,17,209,138,138,138,185,53,53,53
                   db 53,53,22,14,14,42,17,24,17,185,53,53,53
                   db 53,53,53,12,14,17,138,185,138,185,53,53,53
                   db 53,53,53,185,17,17,185,145,172,18,53,53,53
                   db 53,53,53,18,172,26,243,145,172,18,53,53,53
                   db 53,53,17,17,18,172,18,18,18,17,53,53,53
                   db 53,53,17,25,8,17,53,17,24,25,17,53,53
                   db 53,53,53,22,25,22,53,17,8,19,25,28,53
                   db 53,53,53,53,53,53,53,53,53,53,53,53,53

                   db 53,53,53,53,53,53,53,53,53,53,53,53,53
                   db 53,53,53,17,17,18,18,18,17,53,53,53,53
                   db 53,53,17,18,18,18,18,18,18,18,53,53,53
                   db 53,17,18,18,18,226,129,226,18,226,17,53,53
                   db 53,17,18,18,226,226,18,226,18,226,18,53,53
                   db 53,17,17,18,18,18,17,18,90,18,18,53,53
                   db 53,17,18,18,18,18,90,90,90,17,17,53,53
                   db 53,17,18,17,17,90,90,15,90,90,18,53,53
                   db 53,53,17,64,65,90,77,19,90,90,53,53,53
                   db 53,53,53,18,65,90,90,90,90,65,53,53,53
                   db 53,53,22,24,17,65,90,90,136,53,53,53,53
                   db 53,19,25,15,28,17,185,185,138,209,53,53,53
                   db 53,19,24,25,25,17,138,138,140,138,53,53,53
                   db 53,19,17,17,17,138,140,140,140,138,53,53,53
```

```
                db 53,22,27,19,209,138,164,164,164,209,53,53,53
                db 53,17,25,28,17,209,138,138,138,185,53,53,53
                db 53,53,22,14,14,42,17,24,17,185,53,53,53
                db 53,53,53,12,14,17,138,185,138,185,53,53,53
                db 53,53,53,185,17,17,185,145,172,18,53,53,53
                db 53,53,18,18,172,26,243,145,172,18,53,53,53
                db 53,22,17,18,53,53,53,53,22,24,25,22,53
                db 53,19,25,8,17,53,53,53,22,24,19,53,53
                db 53,53,19,22,25,17,53,53,17,17,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

PlayerHeight    equ 24
PlayerWidth     equ 13


CoinPic         db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,0,0,0,0,0,53,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,0,42,66,66,66,66,66,42,0,53,53
                db 53,0,6,66,66,92,92,66,66,66,6,0,53
                db 53,0,42,42,92,92,66,66,66,92,42,0,53
                db 53,0,42,42,92,66,66,66,42,92,42,0,53
                db 53,0,42,42,66,66,66,42,42,92,42,0,53
                db 53,53,0,66,66,66,42,42,66,66,6,0,53
                db 53,53,0,42,66,92,92,92,66,42,0,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,53,53,53,0,0,0,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

                db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,0,0,0,0,53,53,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,0,6,42,66,66,66,66,6,0,53,53
                db 53,53,0,42,66,92,66,66,66,42,0,53,53
                db 53,53,0,42,92,92,66,66,92,66,0,53,53
                db 53,53,0,42,92,66,66,66,66,42,0,53,53
                db 53,53,0,42,66,66,66,42,92,42,0,53,53
                db 53,53,0,42,66,66,42,66,66,6,0,53,53
                db 53,53,0,6,66,92,92,92,66,6,0,53,53
                db 53,53,53,0,6,42,42,42,0,53,53,53,53
                db 53,53,53,53,53,0,0,0,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

                db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,0,42,42,42,0,53,53,53,53
                db 53,53,53,0,6,42,66,66,6,0,53,53,53
                db 53,53,53,0,6,42,92,66,6,0,53,53,53
                db 53,53,53,0,6,42,92,66,6,0,53,53,53
                db 53,53,53,0,6,42,92,42,6,0,53,53,53
                db 53,53,53,0,6,42,92,42,6,0,53,53,53
                db 53,53,53,0,6,42,66,42,6,0,53,53,53
                db 53,53,53,0,6,66,66,6,0,53,53,53,53
                db 53,53,53,53,0,42,6,6,0,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53
```

```
                db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,53,0,42,0,53,53,53,53,53
                db 53,53,53,53,53,0,66,0,53,53,53,53,53
                db 53,53,53,53,53,0,66,0,53,53,53,53,53
                db 53,53,53,53,53,0,66,0,53,53,53,53,53
                db 53,53,53,53,53,0,42,0,53,53,53,53,53
                db 53,53,53,53,53,0,42,0,53,53,53,53,53
                db 53,53,53,53,53,0,42,0,53,53,53,53,53
                db 53,53,53,53,53,0,42,0,53,53,53,53,53
                db 53,53,53,53,53,0,6,0,53,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

                db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,0,42,42,6,0,53,53,53,53
                db 53,53,53,0,6,66,66,42,6,0,53,53,53
                db 53,53,53,0,6,66,42,66,6,0,53,53,53
                db 53,53,53,0,6,66,42,66,6,0,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,53,53,0,6,42,42,6,0,53,53,53
                db 53,53,53,53,0,6,6,6,0,53,53,53,53
                db 53,53,53,53,53,53,0,53,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

                db 53,53,53,53,53,53,53,53,53,53,53,53,53
                db 53,53,53,53,53,0,0,0,53,53,53,53,53
                db 53,53,53,0,6,42,42,42,6,0,53,53,53
                db 53,53,0,6,66,42,66,66,66,6,0,53,53
                db 53,53,0,42,66,66,92,66,66,42,0,53,53
                db 53,53,0,66,66,92,66,66,66,92,0,53,53
                db 53,53,0,42,66,66,66,66,42,92,0,53,53
                db 53,53,0,42,42,66,66,42,42,92,0,53,53
                db 53,53,0,6,66,66,66,42,66,42,0,53,53
                db 53,53,0,6,42,66,92,92,66,6,0,53,53
                db 53,53,53,53,0,42,42,42,6,0,53,53,53
                db 53,53,53,53,53,0,0,0,53,53,53,53,53
                db 53,53,53,53,53,53,53,53,53,53,53,53,53

CoinPicHeight   equ 13
CoinPicWidth    equ 13

DotPic          db 0,0,0,0,0,0,0,0,0,0,0,0,0,0
                db 0,0,0,0,48,48,48,49,49,49,50,0,0,0,0
                db 0,0,0,48,48,48,49,49,50,50,50,51,0,0,0
                db 0,0,48,48,48,49,49,50,50,51,51,51,52,0,0
                db 0,48,48,48,49,49,50,50,51,51,51,52,52,53,0
                db 0,48,48,49,49,50,50,51,51,52,52,52,53,53,0
                db 0,48,49,49,50,50,51,51,52,52,53,53,53,54,0
                db 0,49,49,50,50,51,51,52,52,53,53,53,54,54,0
                db 0,49,50,50,51,51,52,52,53,53,53,54,54,55,0
                db 0,50,50,51,51,52,52,53,53,54,54,54,55,55,0
                db 0,50,51,51,52,52,53,53,54,54,54,55,55,55,0
                db 0,0,51,52,52,53,53,54,54,54,55,55,56,0,0
```

```
                db 0,0,0,52,53,53,54,54,54,55,55,56,0,0,0
                db 0,0,0,0,53,53,54,54,55,55,56,0,0,0,0
                db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

DotPicWidth        equ 15
DotPicHeight        equ 15

KeyPic db 53,53,53,53,53,53,53,53,53,53,53,53,53,53,53
       db 53,53,53,17,17,17,17,17,17,17,17,53,53,53
       db 53,53,53,17,43,14,14,14,14,92,17,53,53,53
       db 53,53,17,17,43,14,14,43,43,14,17,17,53,53
       db 53,17,17,43,43,116,116,116,116,43,14,17,17,53
       db 53,17,43,43,17,17,17,17,17,17,17,14,17,53
       db 53,17,43,44,17,53,53,53,53,53,17,43,17,53
       db 53,17,116,44,17,53,53,53,53,53,17,43,17,53
       db 53,17,43,44,17,53,53,53,53,53,17,43,17,53
       db 53,17,43,92,17,53,53,53,53,53,17,43,17,53
       db 53,17,116,6,17,17,17,17,17,17,17,43,17,53
       db 53,17,17,116,43,14,14,43,43,43,43,17,17,53
       db 53,53,17,17,116,43,43,43,43,43,116,17,53,53
       db 53,53,17,17,190,188,43,43,188,188,17,17,53,53
       db 53,53,53,17,17,17,43,44,17,17,17,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,43,14,17,53,53,53,53,53
       db 53,53,17,17,17,17,43,14,17,53,53,53,53,53
       db 53,53,17,44,44,44,43,14,17,53,53,53,53,53
       db 53,53,17,17,17,17,45,14,17,53,53,53,53,53
       db 53,53,53,53,53,17,17,17,17,53,53,53,53,53
       db 53,53,53,53,53,53,53,53,53,53,53,53,53,53

KeyPicHeight equ 29
KeyPicWidth  equ 14
```

Images

Opening Screen

Menu

Images

Help

About

Images

Game Explanation

Hey there!
You have to unlock the key, the key is a 6-digit number, to unlock each digit you have to complete a level of the game.

Press any key to start now!

Current Level

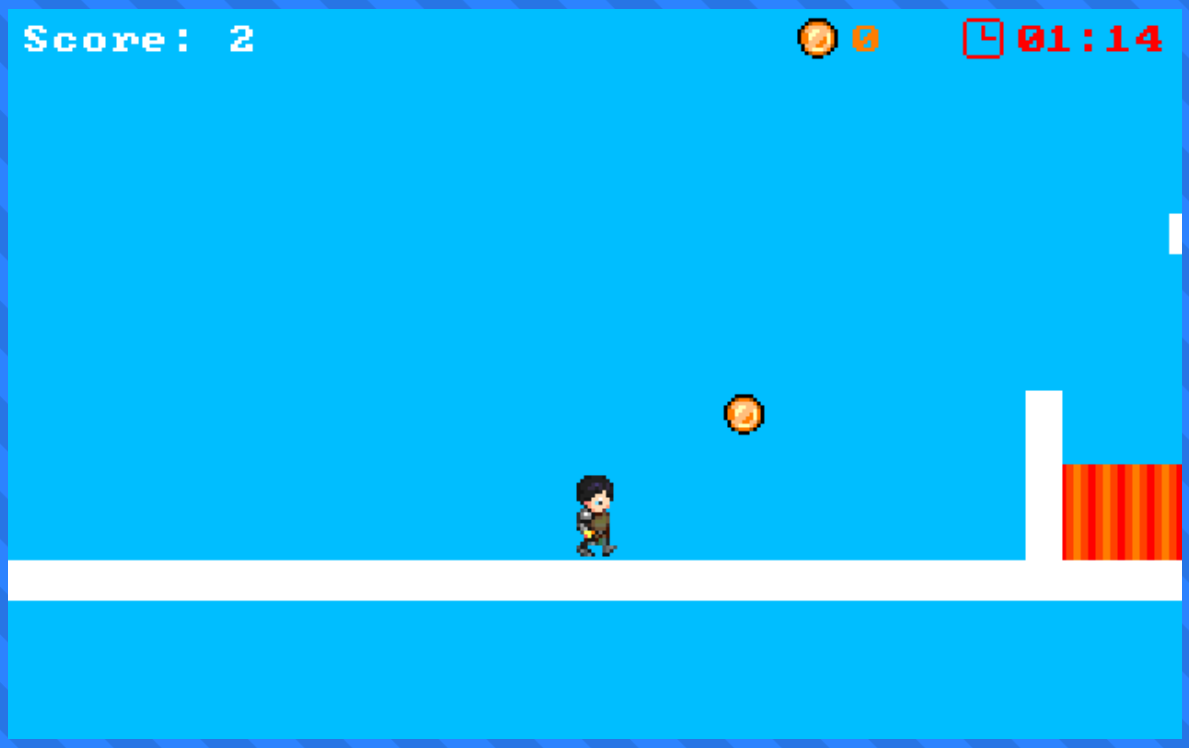
Current digit: 1 Target coins: 2
Target score : 16 Time: 02:00

Total score: 0

Press enter to start unlocking..

Images

Game Examples



Images

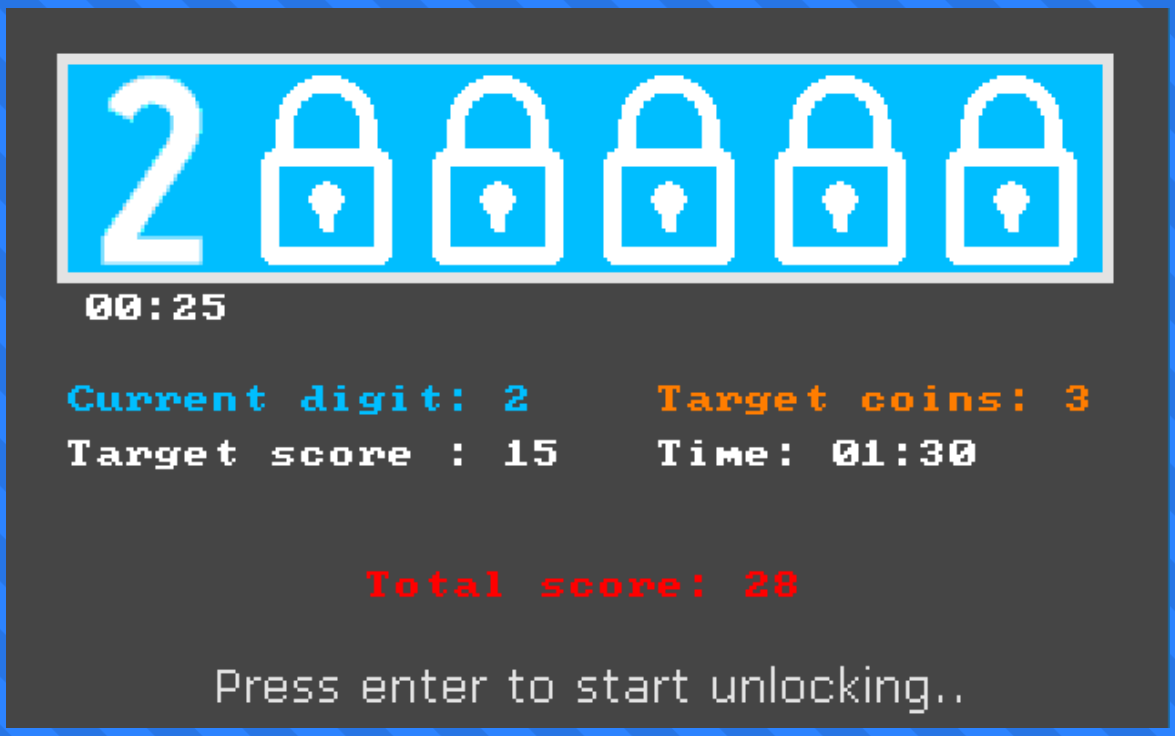
Game Examples

The Key

Images

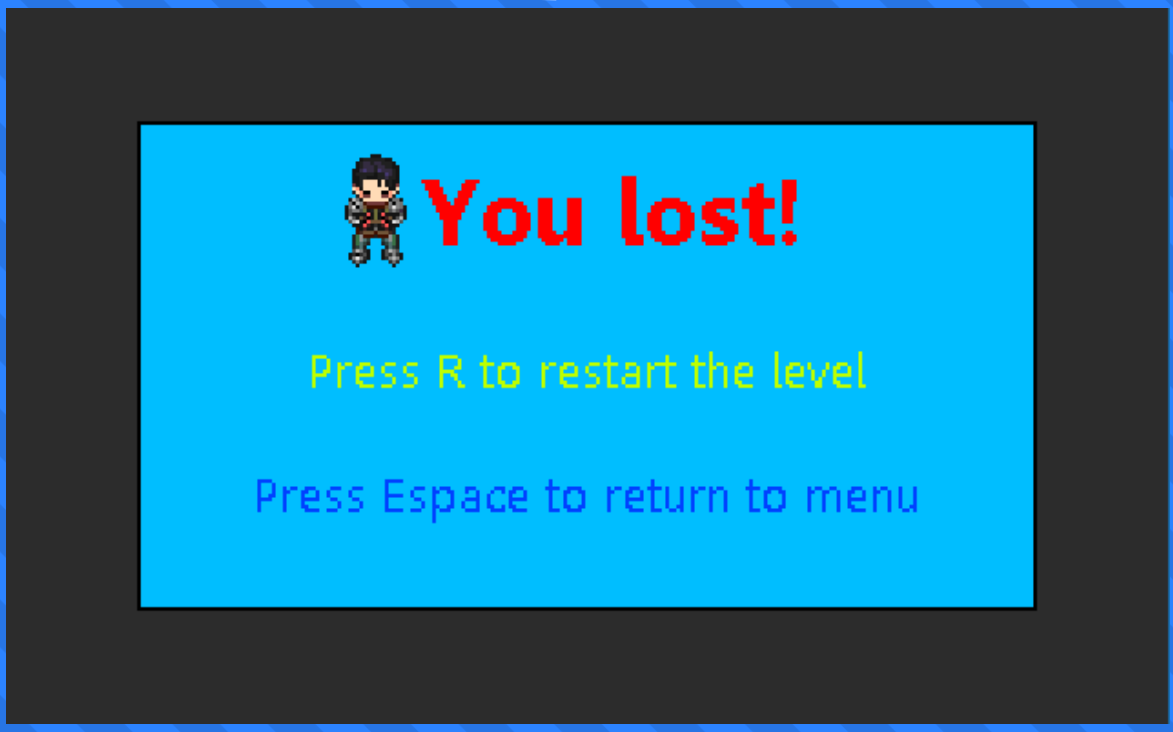
Passing a Level

A digit unlocked

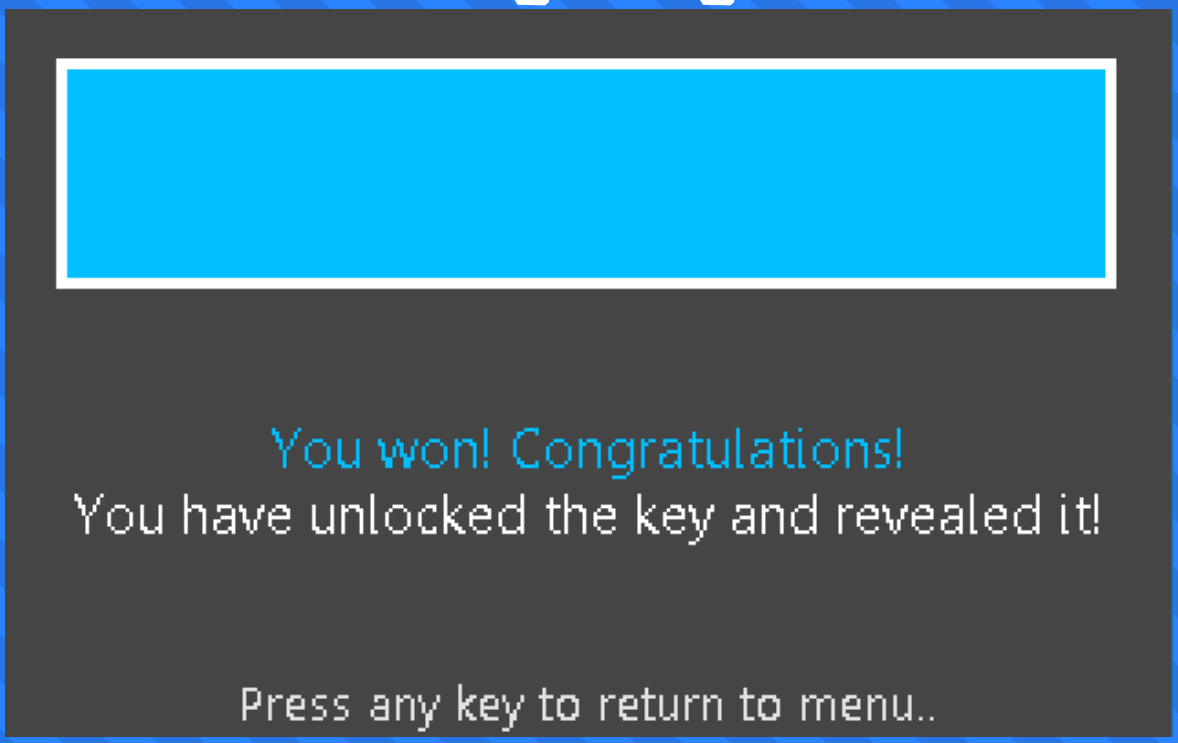


Images

Failing a Level



Winning the game



Summary

My project, Key Breaker, was my biggest project I have ever made. The experience on working on it was amazing and very fun. I learned a lot about how the computer and programming works, assembly is a language that is a word presentation of the hardware capabilities and using it teaches you about the combination of the software and hardware. When I started this project I never I would be able to create a game that is functional as my game with animations and a lot more. I learned during this process of making this project about the following:

- I learned how time in computer works, and how the processor handles it.
- I learned about how simple animation can be made using a couple of frames that were extracted from a GIF file.
- I learned about how graphic works in the basic form of it, that each sprite and character I see, is actually a set of pixels combined together to create it.
- I learned about how sound works, that it is separated into notes and using delay you can make simple sound that will make the user experience more fun and smooth.

I want to thank my teacher, Anatoly Peymer, for helping me with all I needed and to my friends, Omer Benisty and Itay Benvenisti for the help with ideas.