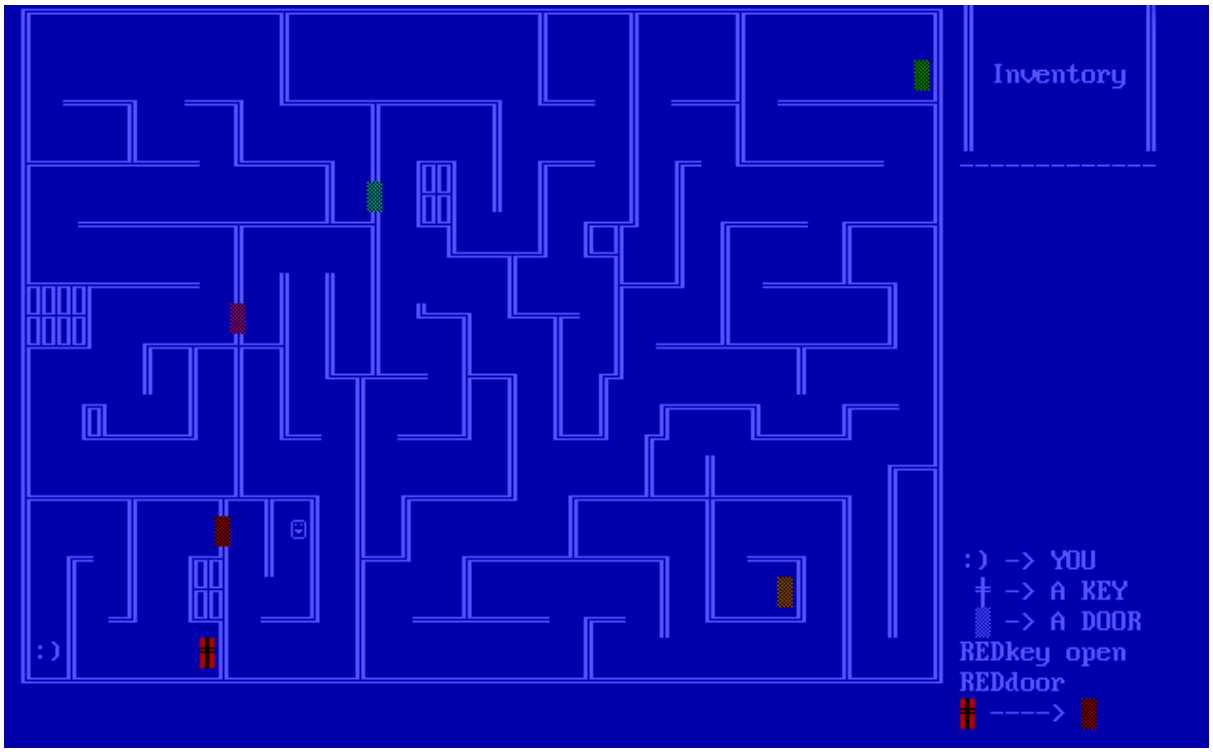
MAZE GAME

Retro’GameZ



Un jeu de Quentin GRUBER, Laurent CUSIMANO, Maxime BIER et Aurélie DUPREZ

SOMMAIRE

FONCTIONNEMENT 4

Déplacement du curseur 4
Appui sur une touche 4
Déplacement du joueur 5
Objets et clés 6
Portes 8
Score et écran de fin 10

AFFICHAGE 11

FONCTIONNEMENT

Déplacement du curseur

La gestion du jeu est basée sur le système de curseur de l'assembleur 8086. Celui-ci n'est pas visible dans le jeu grâce à la commande "CURSOROFF" présente dans la librairie "emu8086.inc ". Cette dernière est la seule librairie utilisée pour ce programme.

Pour déplacer le curseur, il faut modifier les registres dl et dh qui seront ses coordonnées et appeler notre fonction "SetCursor" qui met à jour la position du curseur.

```
SetCursor:
;update the cursor position:
    mov ah, 02h
    mov bh, 00
    int 10h
    ret
```

Appui sur une touche

Le programme attend la pression d'une touche de déplacement afin de lancer une fonction qui va déplacer le personnage dans la direction souhaitée.

```
 ;wait for a key to be pressed:
mov ah, 0h
int 16h

;controls with ZQSD
cmp al, 115  ;if "s"
je Down

cmp al, 122  ;if "z"
je Up

cmp al, 113   ;if "q"
je Left

cmp al, 100   ;if "d"
je Right

  ;controls with arrows
cmp ah, 50h  ;if downarrow
je Down

cmp ah, 48h   ;if uparrow
je Up

cmp ah, 4Bh  ;if leftarrow
je Left

cmp ah, 4Dh  ;if rightarrow
je Right

;other
cmp al, 27 ;if "ecs":
je give_up

jmp main
```

Déplacement du joueur

Pour effectuer un déplacement la fonction “clear_player” est appelée pour effacer le joueur. Par la suite, le curseur est déplacé dans la direction choisie et la commande “PRINT ‘:)’” affiche le joueur à sa nouvelle position.

```
Right:
    add dl, 2
    call SetCursor
    call TestColid
    sub dl,2
    call SetCursor
    ;test if a collider is detected:
    cmp ColliderDetected,'y'
    je main
    ;if not move the player to his new location:
    call clear_player
    add dl,1
    call SetCursor
    mov bh, 0
        mov ah, 0x2
        int 0x10
        mov cx, 2 ; nb char
        mov bh, 0
        mov bl, 0x19 ; color
        mov al, 0x20 ; blank char
        mov ah, 0x9
        int 0x10
        mov ah, 02h
        mov bh, 00
        int 10h

    PRINT ':)'
    add MovesCount,1
    jmp main
    ret
```

En revanche, avant chaque déplacement, le programme vérifie le contenu de la case dans laquelle le personnage s’apprête à se déplacer :

- Si cette dernière contient un caractère représentant l’un des murs du labyrinthe, il empêche le déplacement du personnage.
- Si elle contient un caractère avec lequel on doit interagir (porte, clé…), le programme va exécuter la fonction associée à ce caractère.

```
TestColid:
;test if next charactere is a "blacklist" one:
    mov ah, 08h
    int 10h

    cmp al,206
    je CollidYes

    cmp al,188
    je CollidYes

    cmp al,203
    je CollidYes

    cmp al,187
    je CollidYes
```

Ici, 206, 188, 203 et 187 sont des caractères représentant des murs.

Objets et clés

Les clés apparaissant au fur et à mesure de la partie et toujours dans le même ordre. La fonction “objectpickup”, qui est exécutée après un déplacement sur une clé, vérifie dans quelle partie du labyrinthe le joueur se situe et exécute la fonction de la clé correspondante.

```
objectpickup:
;look at what key we have picked up:
     key1check:
          cmp Event_key,1
          je key1pickup

     key2check:
          cmp Event_key,2
          je key2pickup
     key3check:
          cmp Event_key,3
          je key3pickup

     key4check:
          cmp Event_key,4
          je key4pickup

     key5check:
          cmp Event_key,5
          je key5pickup
```

Lors du ramassage d’un objets/l’ouverture d’une porte, un message est affiché sous l’inventaire du joueur :

Pour afficher ce dialogue, la fonction “clear_oldmessage” est appelée. Celle-ci va effacer le dernier message sur l’interface avant d’afficher le texte de remplacement via la commande “PRINT”.

```
key1pickup:
     call Save_PlayerLoc
     call clear_oldmessage
     mov dl,62
     mov dh,6
     call SetCursor

     ;message to let you know you collected an object:
     PRINT 'You have found'
     mov dl,62
     mov dh,7
     call SetCursor

     PRINT ' a redkey'
```

Chaque clé est associée à sa propre fonction. Elles suivent toutes la même procédure :

- Afficher le message d'obtention de la clé
- L'ajouter à l'inventaire du joueur
- Stocker l'information dans une variable

L'inventaire est mis à jour via l'appel de la fonction "UpdateInv". Cette fonction vérifie quelles clés sont possédées par le joueur et l'affiche dans son inventaire.

```
UpdateInv proc near
 ;update the inventory:
  testhavekey1:
            cmp whichKey,1
            je Draw_on_key1
            jmp testhavekey2
            Draw_on_key1:
            ;draw in inventory which key we have unlocked:
            mov dl,65
            mov dh,4
            mov bh, 0
            mov ah, 0x2
            int 0x10
            mov cx, 1 ; nb char
            mov bh, 0
            mov bl, 0x40 ; color
            mov al, 0x20 ; blank char
            mov ah, 0x9
            int 0x10
            mov ah, 02h
            mov bh, 00
            int 10h
            PRINT 216
            jmp end_UpdateInv
```

Portes

Lors de la collision avec une porte, une première fonction se lance : “keytest”. Elle va tester quelle(s) clé(s) sont dans l’inventaire du joueur.

```
keytest:
;look at what key we have:
        nokeytest:;look at first if you have a key
        cmp whichKey,0
        je display_nokeymessage
        jmp redkeytest

       ;test all key value:
       redkeytest:

                    cmp whichKey,1
                    je opendoor1
                    jmp bluekeytest


        bluekeytest:

                cmp whichKey,2
                je opendoor2
                jmp yellowkeytest

           yellowkeytest:

                cmp whichKey,3
                je opendoor3
                jmp OrangeAndGreenkeytest

          OrangeAndGreenkeytest:

                cmp whichKey,4
                je GreenOrOrange
                cmp whichKey,5
                je GreenOrOrange
```

La méthode de vérification de clé varie selon les portes afin de s’assurer que le joueur ouvre les portes dans l’ordre. Pour les trois premières portes, c’est très simple : si le joueur veut ouvrir une porte, il doit avoir la clé correspondante.

Pour les deux dernières, la logique varie. Ces deux portes devenant accessibles en même temps, il faut s’assurer que le joueur les ouvre dans le bon ordre, orange puis verte. Cette vérification s’effectue via la fonction “GreenOrOrange” qui vérifie la position du joueur.

- S’il se trouve face à la porte verte, la fonction “isthatgreen” vérifie si la clé verte est présente dans l’inventaire. Ceci n’est possible qu’après ouverture de la porte orange : c’est cet « évènement » qui déclenche l’apparition de la clé verte.
- Sinon, le joueur se trouve en face de la porte orange : l’ouverture suit la même logique que les autres portes.

```
OrangeAndGreenkeytest:

    cmp whichKey,4
    je GreenOrOrange

    cmp whichKey,5
    je GreenOrOrange
    jmp havenokey

    GreenOrOrange:
    cmp dl,60
    je isthatgreen
    jmp opendoor4

    isthatgreen:
    ;verify if the boolean "IsGreen" is true:
    cmp IsGreen,'y'
    je opendoor5
```

Chaque porte possède sa propre fonction qui effectue les opérations suivantes à son ouverture :

- Afficher un message au joueur à propos de l'ouverture de cette porte
- Déclarer via la variable "whichkey" l'utilisation de la clé de la zone précédente
- dessine la clé de la zone suivante.

```
opendoor1:
        call Save_PlayerLoc
        call clear_oldmessage

        mov dl,62
        mov dh,6
        call SetCursor

        PRINT 'you have open'
        mov dl,62
        mov dh,7
        call SetCursor

        PRINT 'red door'


        mov whichKey,0

          draw_purplekey:
              mov dl,17
              mov dh,12
              mov bh, 0
              mov ah, 0x2
              int 0x10
              mov cx, 1 ; nb char
              mov bh, 0
              mov bl, 0x50 ; color
              mov al, 0x20 ; blank char
              mov ah, 0x9
              int 0x10
              mov ah, 02h
              mov bh, 00
              int 10h
              PRINT 216

        ;return the cursor to its original position:
        call Load_PlayerLoc
        call SetCursor
        ret
        jmp inside_loop
```

Score et écran de fin

À chaque déplacement, le programme incrémente une variable. Cette variable est affichée sur l'écran de fin de partie, généré à l'ouverture de la dernière porte, et permet au joueur d'avoir un retour de sa performance.

Il existe cependant un deuxième écran de fin, caché dans le jeu…

AFFICHAGE

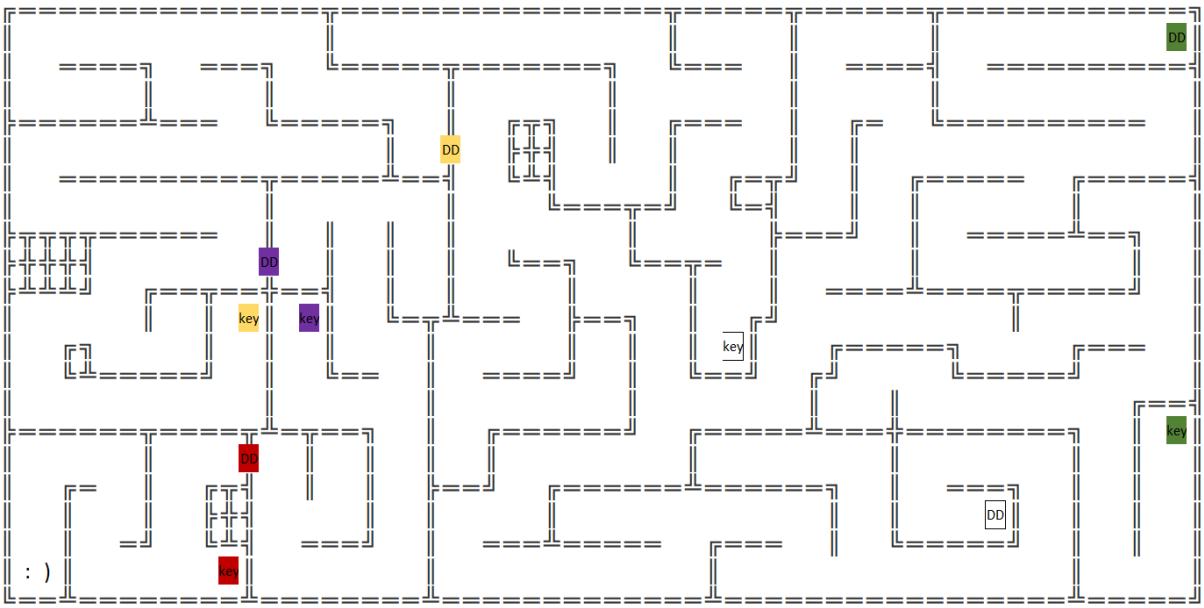
Pour l'affichage du jeu, la première solution envisagée a été d'afficher différents « sprites » pour les objets et le personnage :

Exemples de « sprites » envisagés

Cette solution a cependant été rapidement abandonnée, au vu de sa complexité d'implémentation et de maintenance.

La deuxième solution, celle-ci adoptée, est de modéliser le labyrinthe en caractères ASCII, afin de faciliter son affichage. L'équipe a donc choisi de réaliser un labyrinthe de dimensions 60x22, afin de tirer pleinement partie des possibilités de l'environnement imposé. Bien que le nombre de couleurs disponibles pour les éléments soit limité par les capacités techniques de l'assembleur 8086, ils possèdent tous leur propre couleur.

Avant d'implémenter le labyrinthe dans le jeu, un plan réalisé à l'aide d'un tableur a permis de le préparer et de pouvoir y apporter toutes les corrections nécessaires



Les caractères ASCII utilisés sont les suivants :

‖	═	╝	╗	╔	╚	╠	╣	╩	╦	╬	ï	▒
186	205	188	187	201	200	204	185	202	203	206	216	177

Résultat en jeu :