



SUPINFO
International University

INSTITUTE OF INFORMATION TECHNOLOGY

Class Group Project Architecture

Delivery

Group members
Documentation

Version 1.0

Last update: 06/06/2013

Use: Students

Author: Samuel CUELLA

Conditions d'utilisations : SUPINFO International University vous permet de partager ce document. Vous êtes libre de :

- Partager — reproduire, distribuer et communiquer ce document
- Remixer — modifier ce document

A condition de respecter les règles suivantes :

Indication obligatoire de la paternité — Vous devez obligatoirement préciser l'origine « SUPINFO » du document au début de celui-ci de la même manière qu'indiqué par SUPINFO International University – Notamment en laissant obligatoirement la première et la dernière page du document, mais pas d'une manière qui suggérerait que SUPINFO International University vous soutiennent ou approuvent votre utilisation du document, surtout si vous le modifiez. Dans ce dernier cas, il vous faudra obligatoirement supprimer le texte « SUPINFO Official Document » en tête de page et préciser notamment la page indiquant votre identité et les modifications principales apportées.

En dehors de ces dispositions, aucune autre modification de la première et de la dernière page du document n'est autorisée.

NOTE IMPORTANTE : Ce document est mis à disposition selon le contrat CC-BY-NC-SA Creative Commons disponible en ligne <http://creativecommons.org/licenses> ou par courrier postal à Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA modifié en ce sens que la première et la dernière page du document ne peuvent être supprimées en cas de reproduction, distribution, communication ou modification. Vous pouvez donc reproduire, remixer, arranger et adapter ce document à des fins non commerciales tant que vous respectez les règles de paternité et que les nouveaux documents sont protégés selon des termes identiques. Les autorisations au-delà du champ de cette licence peuvent être obtenues à support@supinfo.com.

© SUPINFO International University – EDUCINVEST - Rue Ducale, 29 - 1000 Brussels Belgium . www.supinfo.com

Table of contents





TABLE DES MATIERES

1	RESUME DU GROUPE	4
2	RAPPORT DU PROJET.....	5
3	MANUEL D'UTILISATION.....	5
3.1	<i>SOMMAIRE.....</i>	6
3.2	<i>INTRODUCTION.....</i>	7
3.3	<i>CONFIGURATION REQUISE.....</i>	8
3.4	<i>INSTRUCTIONS D'INSTALLATION</i>	8
3.5	<i>COMMENT JOUER ?.....</i>	9
3.6	<i>SUPPORT TECHNIQUE</i>	9
4	DOCUMENTATION TECHNIQUE	10
4.1	<i>FONCTIONNEMENT.....</i>	10
4.1.1	Déplacement du curseur	10
4.1.2	Appui sur une touche	10
4.1.3	Déplacement du joueur	11
4.1.4	Objets et clés	12
4.1.5	Portes.....	13
4.1.6	Score et écran de fin.....	15
4.2	<i>AFFICHAGE.....</i>	16
4.2.1	<i>Exemples de « sprites » envisagés</i>	16

1 RESUME DU GROUPE

Campus: **Strasbourg**

Classe : **A.Sc.1**

ID Open Campus	Nom de famille	Prénom	Photo
296188	GRUBER	Quentin	
295006	BIER	Maxime	
291960	CUSIMANO	Laurent	
280120	DUPREZ	Aurélie	

2 RAPPORT DU PROJET

Quentin a pris en main le fonctionnement du jeu en programmant les mécaniques centrales du jeu : les déplacements du joueur, les collisions et interaction du joueur, et l'affichage de l'inventaire du joueur et des dialogues.

Aurélie a programmé le système de score du jeu, ainsi que l'affichage du menu et des différents écrans de fin.

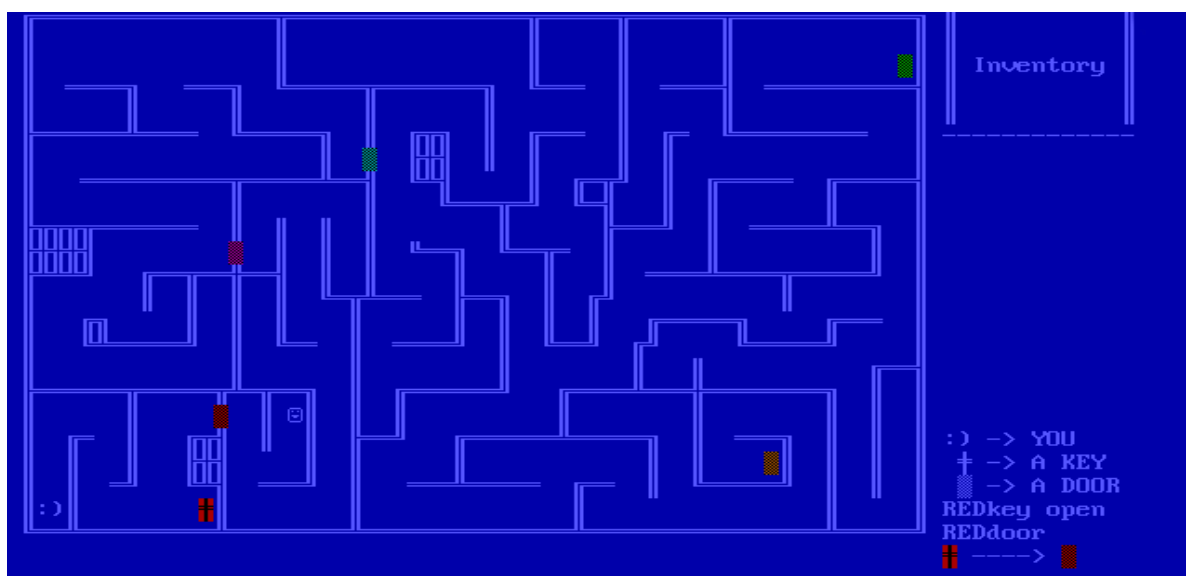
Maxime a conçu dans son intégralité le labyrinthe du jeu, et a participé à sa génération.

Laurent a programmé la génération du labyrinthe dans le jeu.

3 MANUEL D'UTILISATION

MAZE GAME

Retro'GameZ



3.1 SOMMAIRE

INTRODUCTION	7
CONFIGURATION REQUISE	8
INSTRUCTIONS D'INSTALLATION	8
COMMENT JOUER ?.....	9
SUPPORT TECHNIQUE.....	9

Maze Game – Retro'GameZ

3.2 INTRODUCTION

Maze Game est un jeu de labyrinthe où l'on incarne un personnage souriant. Le but du jeu est de s'échapper du labyrinthe en ouvrant successivement les 5 portes qui nous séparent de la liberté, et ce en récupérant pour chaque porte une clé de la couleur qui lui correspond.

Une fois échappé du labyrinthe, le joueur est informé du nombre de déplacements effectués. L'objectif est maintenant de s'échapper du labyrinthe en un minimum de mouvements.

Il existe également une manière cachée de gagner la partie, à vous de la trouver !

3.3 CONFIGURATION REQUISE

- Un ordinateur permettant de lancer l'émulateur DOSBox
- Un clavier
- Un écran

3.4 INSTRUCTIONS D'INSTALLATION

1. Lancer DOSBox
2. Entrer l'instruction « *mount c C:\[votre chemin d'installation]* »
3. Entrer l'instruction « *C:* » pour se déplacer sur le disque C :
4. Entrer l'instruction « *cd [votre chemin d'installation]* » pour naviguer jusqu'au dossier contenant le jeu.
5. Entrer l'instruction « *MAZEGAME.COM* » pour lancer le jeu

Maze Game – Retro'GameZ

3.5 COMMENT JOUER ?

Pour se déplacer, le joueur peut utiliser deux ensembles de touches selon sa préférence : ➤ Les flèches directionnelles :

- ↑ a
- ↓ b
- ← c
- → d

➤ Les touches de déplacement habituelles :

- Z (haut)
- S (bas)
- Q (gauche)
- D (droite)

3.6 SUPPORT TECHNIQUE

En cas de problème, n'hésitez pas à contacter le support technique à l'adresse suivante :

296188@supinfo.com

4 DOCUMENTATION TECHNIQUE

4.1 FONCTIONNEMENT

4.1.1 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
```

4.1.2 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

```

4.1.3 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 caractere 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.

4.1.4 Objets et clés

Les clés apparaissent 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
    
```

4.1.5 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_nokmessage
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

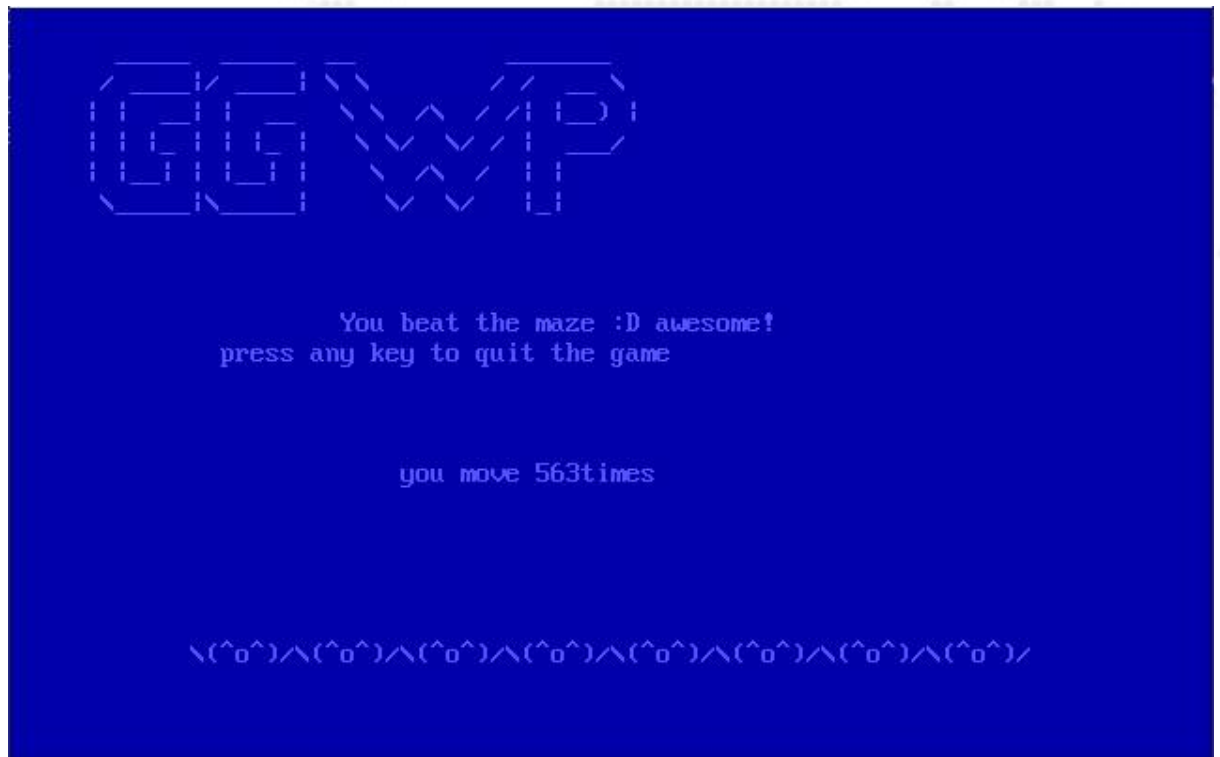
    PRINT 'red door'

    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
```

4.1.6 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...

4.2 AFFICHAGE

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

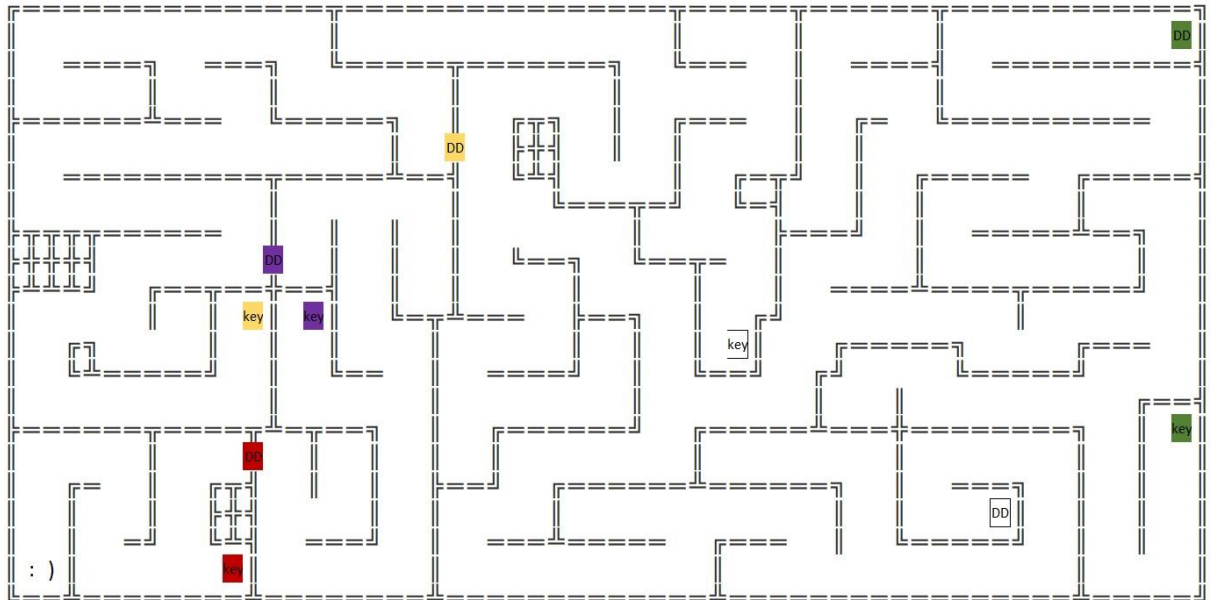


4.2.1 Exemples de « sprites » envisagés

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 :

	=	└┐	┌┐	└┐	┌┐	└┐	┌┐	└┐	┌┐	└┐	┌┐	└┐
186	205	188	187	201	200	204	185	202	203	206	216	177

Résultat en jeu :

