Dossier Technique Projet Arduino

Générateur de labyrinthe

Sommaire

[INTRODUCTION 3](#_Toc484104031)

[Présentation de notre projet 3](#_Toc484104032)

[Schéma électronique 3](#_Toc484104033)

[CAHIER DES CHARGES 4](#_Toc484104034)

[CODE COMPLET ET EXPLICATION 6](#_Toc484104035)

[TESTS DU PROGRAMME 17](#_Toc484104036)

[CONCLUSION 18](#_Toc484104037)

# INTRODUCTION

## Présentation de notre projet

Le but de notre projet d’Intelligence Ambiante est de réaliser un générateur aléatoire de labyrinthe affiché sur la matrice LED connectée à la plaquette Arduino Unau.

Pour cela, nous allons avoir besoin de la plaquette Arduino et de la Magic LED Matrix 2.0 connectée à cette dernière selon le schéma électronique qui va suivre.

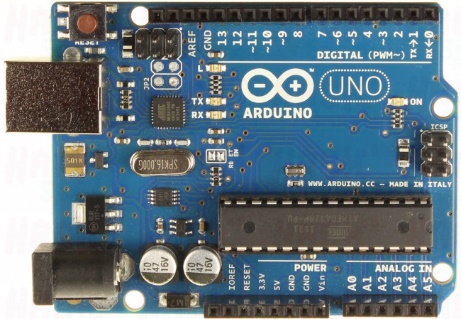
Nous voulons générer des labyrinthes parfaits grâce à la technique des ilôts. Chaque cellule/ilôt du labyrinthe est modélisée par 9 LED avec à chaque fois la LED du milieu haut NORD, la LED du milieu gauche OUEST, et ainsi de suite pour EST et SUD. Ces quatre LEDs sont en commun avec la cellule adjacente, par exemple, la cellule de gauche aura la même LED EST que la LED OUEST de la cellule de droite. Pour représenter le labyrinthe, les murs seront les 8 LEDS extérieurs de nos ilôts. Une ouverture du chemin du labyrinthe verra une des LEDs NORD, OUEST, EST, SUD éteinte selon la direction choisie aléatoirement par le programme.

## Schéma électronique

Voici le schéma électronique de notre montage : Matrice LED + Plaquette Arduino :

Connexion via Port USB à l’ordinateur

Plaquette Arduino



Matrice LED

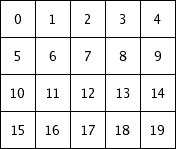
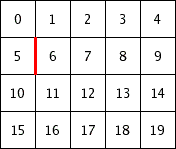
Connexion de la Plaquette Arduino à la matrice LED avec les branchements :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **M. LED** | D | WR | CLK | CS | GND |
| **Arduino** | SLK | SDO | X | BUZZER | GND |

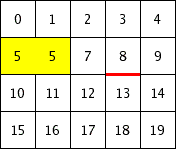
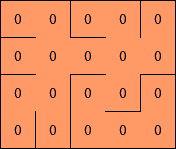
# CAHIER DES CHARGES

* Le labyrinthe doit être généré assez rapidement (moins d’1 min serait correct).
* Le matrice LED doit afficher un message lorsque le programme est en train de fonctionner pour créer le chemin du labyrinthe et montrer que le programme est en train de travailler.
* Le labyrinthe doit pouvoir être extensible en hauteur et en largeur de 3\*3 LEDs jusqu’à 31\*15 LEDs.
* Une cellule de départ et une cellule d’arrivée doivent être affichés pour pouvoir résoudre ce labyrinthe.

# Principe de fontionnement de l’algorithme

On donne à chaque case un numéro de composante, on choisi ensuite une case de départ et une direction aléatoirement.

On casse le mur entre ces deux cases et on unifie leurs composantes.

On obtiendra un labyrinthe parfait quand toutes les composantes seront identiques.

# CODE COMPLET ET EXPLICATION

#define LARGEUR 15 // Choix des dimensions du labyrinthe (3x3 -> 15x7)

#define HAUTEUR 7

typedef struct {

byte composante; // numéro de composante connexe = j \* LARGEUR + i

byte murs; // murs ouverts ou fermes

} T\_Cellule;

// Interpretation de l'octet "murs" eg, 0000 a3 a2 a1 a0

// 0000 ne servent à rien

// 0 : ferme ; 1 : ouvert

// puis Nord a0, Est a1, Sud a2, Ouest a3

// 0000 1001 signifie : murs nord et ouest ouverts, est et sud fermes

#define NORD 0

#define EST 1

#define SUD 2

#define OUEST 3

T\_Cellule labyrinthe[LARGEUR \* HAUTEUR];

//Declaration d’un tableau contenant L\*H elements

// var. globale initialisée à 0 partout : murs sont tous fermes

void initComposantes() {

// place une composante differente dans chaque case

byte i;

for (i = 0; i < LARGEUR \* HAUTEUR; i++) {

labyrinthe[i].composante = i;

}

}

//Tableaux qui nous serviront à visualiser plus clairement les résultats

char \* sDirs[] = {"NORD", "EST", "SUD", "OUEST"};

byte dir[] = {NORD, EST, SUD, OUEST};

byte dirOpposees[] = {SUD, OUEST, NORD, EST};

char \* descriptionMurs[] = {

"O S E N", //0000

"O S E .", //0001

"O S . N", //0010

"O S . .", //0011

"O . E N", //0100

"O . E .", //0101

"O . . N", //0110

"O . . .", //0111

". S E N", //1000

". S E .", //1001

". S . N", //1010

". S . .", //1011

". . E N", //1100

". . E .", //1101

". . . N", //1110

". . . ." //1111

};

void setup () {

ht1632\_setup(); // initialise the wire library and hardware

Serial.begin(9600);

Wire.begin();

setup7Seg();

initComposantes(); //On initialise le tableau de composantes au debut

cls(); //Puis on éteint la matrice

}

void detruire\_mur(byte i, byte j, byte direction) {

// détruire deux murs, celle de la cellule a composante n et sa voisine

byte n = j \* LARGEUR + i, nVoisin;

byte iVoisin = i, jVoisin = j;

switch (direction) { // On acquiert les coordonnées de la cellule voisine en fonction de la direction

case NORD:

jVoisin--;

break;

case EST:

iVoisin++;

break;

case SUD:

jVoisin++;

break;

case OUEST:

iVoisin--;

break;

}

nVoisin = jVoisin \* LARGEUR + iVoisin;

//Ci-dessous on détruit le mur de la cellule suivant la direction choisie par la fonction aléatoire

//On peut faire cela à l’aide de l’opérateur « OU » qui ajoutera 1 a la position souhaitée  
// Rappel : 1 signifie mur cassé

labyrinthe[n].murs = labyrinthe[n].murs | 1 << direction;

//Ainsi qu’ici on détruit la mur de la cellule voisine suivant la direction opposée choisie

labyrinthe[nVoisin].murs = labyrinthe[nVoisin].murs | 1 << (dirOpposees[direction]);

fusionner(labyrinthe[n].composante, labyrinthe[nVoisin].composante);

}

void fusionner(byte c1, byte c2) {

// change les composantes des cases c2 en c1

Serial.print("Unification composantes"); Serial.print(c1); Serial.print(" et "); Serial.println(c2);

byte n;

for (n = 0; n < LARGEUR \* HAUTEUR; n++) {

if (labyrinthe[n].composante == c2)

labyrinthe[n].composante = c1;

}

}

void dump() {

//Fonction permettant affichage du labyrinthe sous forme de données

byte n;

for (n = 0; n < LARGEUR \* HAUTEUR; n++) {

//Décrit les caractériqtiques de chaque case du tableau labyrinthe

Serial.print("n="); Serial.print(n);

Serial.print(" i="); Serial.print(n % LARGEUR);

Serial.print(" j="); Serial.print(n / LARGEUR);

Serial.print(" c="); Serial.print(labyrinthe[n].composante);

Serial.print(" murs="); Serial.println(descriptionMurs[labyrinthe[n].murs]);

}

}

void genererLabyrintheParfait() {

byte nbComposantes = LARGEUR \* HAUTEUR;

while (nbComposantes > 1 ) {

//Le labyrinthe est dit parfait quand toutes les composantes des cellules sont identiques

// Tirer au sort une case et une direction

byte dirAleatoire = dir[random(0, 3)];

byte j = random(0, HAUTEUR ); //ordonnés

byte i = random(0, LARGEUR ); //abscises

Serial.print("Direction="); Serial.println(sDirs[dirAleatoire]);

Serial.print("i="); Serial.println(i);

Serial.print("j="); Serial.println(j);

int nAleatoire = j \* LARGEUR + i;

int nAleatoireVoisin = j \* LARGEUR + i;

switch (dirAleatoire) { // on determine les coordonnées du voisin aléatoire conséquent du rand

case NORD:

nAleatoireVoisin = (j - 1) \* LARGEUR + i;

break;

case EST:

nAleatoireVoisin = j \* LARGEUR + (i + 1);

break;

case SUD:

nAleatoireVoisin = (j + 1) \* LARGEUR + i;

break;

case OUEST:

nAleatoireVoisin = j \* LARGEUR + (i - 1);

break;

}

// On ne doit pas casser un mur exterieur

if ((dirAleatoire == NORD) && (j == 0)) continue;

if ((dirAleatoire == EST) && (i == LARGEUR - 1)) continue;

if ((dirAleatoire == SUD) && (j == HAUTEUR - 1)) continue;

if ((dirAleatoire == OUEST) && (i == 0)) continue;

// SI cases autour du mur sont de la meme composante on continue

if (labyrinthe[nAleatoire].composante == labyrinthe[nAleatoireVoisin].composante) {

Serial.println("cases deja connexes !");

continue;

} else {

Serial.println("on Casse !");

switch (dirAleatoire) { //Sinon on casse le mur

case NORD:

detruire\_mur(i, j, NORD);

break;

case EST:

detruire\_mur(i, j, EST);

break;

case SUD:

detruire\_mur(i, j, SUD);

break;

case OUEST:

detruire\_mur(i, j, OUEST);

break;

}

// chaque opération de suppression d'un mur supprime une composante

fusionner(labyrinthe[nAleatoire].composante, labyrinthe[nAleatoireVoisin].composante);

nbComposantes--; //on décrémente le nombre de composantes après chaque suppréssion

}

}

dump() ; // Nous permet d’afficher l’opération faite sur le serial monitor (Soit ça casse ou ça passe)

}

void printLab() {

//Nous visualisions avant notre labyrinthe sous forme de données uniquement, on veut maintenant le rendre plus visuel grâce a printLab

byte i = 0, j = 0;

byte n;

byte x, y;

for (i = 0; i < LARGEUR; i += 2) Serial.print("\_\_"); //On trace le mur de haut

Serial.println("\_");

for (j = 0; j < HAUTEUR; j++) {

Serial.print('|'); //Puis le mur de gauche

for (i = 0; i < LARGEUR; i++) {

if ( (labyrinthe[j \* LARGEUR + i].murs & B00000100) == 0) Serial.print('\_');

//On vérifie à chaque cellule, si elle a un mur de bas

else Serial.print(' ');

if ( (labyrinthe[j \* LARGEUR + i].murs & B00000010) == 0) Serial.print('|');

//De meme pour le mur de droite

else Serial.print(' ');

}

Serial.println("");

}

ht1632\_plot(1 , 1 , ORANGE); //On affiche la case départ

ht1632\_plot(2 \* LARGEUR - 1 , 2 \* HAUTEUR - 1 , RED); //Et la case d’arrivée

//On suit la même méthode pour afficher le labyrinthe, cette fois sur la matrice LED

for (i = 0; i < 2 \* LARGEUR ; i += 2) { //mur NORD

ht1632\_plot( i - 1, 0, GREEN);

ht1632\_plot( i, 0, GREEN);

ht1632\_plot( i + 1, 0, GREEN);

}

for (j = 0; j < 2 \* HAUTEUR - 1; j += 2 ) { // mur OUEST

ht1632\_plot(0 , j - 1, GREEN);

ht1632\_plot(0 , j, GREEN);

ht1632\_plot(0 , j + 1, GREEN);

}

for (j = 0; j < HAUTEUR; j++) {

for (i = 0; i < LARGEUR; i ++) {

//Puisque i et j ne correspondent pas au coordonnées sur la matrice LED on déclare deux variable qui joueront un role d’adaptation

x = 2 \* i + 1 ;

y = 2 \* j + 1 ;

if ((labyrinthe[j \* LARGEUR + i].murs & B00000100) == 0) { //mur SUD

ht1632\_plot(x - 1 , y + 1 , GREEN);

ht1632\_plot(x , y + 1 , GREEN);

ht1632\_plot(x + 1 , y + 1 , GREEN);

}

if ( (labyrinthe[j \* LARGEUR + i].murs & B00000010) == 0) { //mur EST

ht1632\_plot(x + 1, y - 1, GREEN);

ht1632\_plot(x + 1, y, GREEN);

ht1632\_plot(x + 1, y + 1, GREEN);

}

}

}

}

//La fonction ce dessus (generer\_labyrintheparfait) nécessite une trentaine de seconde à se réaliser, on a donc pensé à afficher un écran de chargement le temps que le labyrinthe se charge

void display\_chargement() {

//Cet ecran indiquera la règle du jeu : il affichera deux points, départ et arrivée, avec une flèche qui pointe vers ce dernier.

byte i, j,k;

// on trace un cadre pour notre écran de chargement

for (i = 0; i <= 2 \* LARGEUR; i++) {

ht1632\_plot(i , 0 , GREEN); // ligne horizentale en haut

ht1632\_plot(i , 2 \* HAUTEUR , GREEN); //et en bas

}

for (j = 0; j <= 2 \* HAUTEUR; j++) {

ht1632\_plot(0 , j , GREEN); // De même mais en vertical cette fois

ht1632\_plot(2 \* LARGEUR , j , GREEN);

}

ht1632\_plot( floor(LARGEUR / 2) , HAUTEUR , ORANGE); //Point depart

ht1632\_plot( floor(3 \* LARGEUR / 2) , HAUTEUR , RED); //Point arrivée

ht1632\_plot(LARGEUR - 3 + k , HAUTEUR , GREEN); //Affichage de la flèche

ht1632\_plot(LARGEUR - 2 + k , HAUTEUR , GREEN);

ht1632\_plot(LARGEUR - 1 + k, HAUTEUR , GREEN);

ht1632\_plot(LARGEUR + k , HAUTEUR , GREEN);

ht1632\_plot(LARGEUR + 1 + k , HAUTEUR , GREEN);

ht1632\_plot(LARGEUR + k , HAUTEUR + 1 , GREEN);

ht1632\_plot(LARGEUR - 1 + k , HAUTEUR + 2 , GREEN);

ht1632\_plot(LARGEUR + k , HAUTEUR - 1 , GREEN);

ht1632\_plot(LARGEUR - 1 + k, HAUTEUR - 2 , GREEN);

}

void clear\_chargement () {

//Clear chargement est exactement la meme fonction que display\_chargement sauf qu’elle eteint cet écran

//Toutes les couleurs sont à BLACK

byte i, j;

for (i = 0; i <= 2 \* LARGEUR; i++) {

ht1632\_plot(i , 0 , BLACK);

ht1632\_plot(i , 2 \* HAUTEUR , BLACK);

}

for (j = 0; j <= 2 \* HAUTEUR; j++) {

ht1632\_plot(0 , j , BLACK);

ht1632\_plot(2 \* LARGEUR , j , BLACK);

}

ht1632\_plot( floor(LARGEUR / 2) , HAUTEUR , BLACK); //pt depart

ht1632\_plot( floor(3 \* LARGEUR / 2) , HAUTEUR , BLACK); //pt arrive

ht1632\_plot(LARGEUR - 3 , HAUTEUR , BLACK); //fleche

ht1632\_plot(LARGEUR - 2 , HAUTEUR , BLACK);

ht1632\_plot(LARGEUR - 1 , HAUTEUR , BLACK);

ht1632\_plot(LARGEUR , HAUTEUR , BLACK);

ht1632\_plot(LARGEUR + 1 , HAUTEUR , BLACK);

ht1632\_plot(LARGEUR , HAUTEUR + 1 , BLACK);

ht1632\_plot(LARGEUR - 1 , HAUTEUR + 2 , BLACK);

ht1632\_plot(LARGEUR , HAUTEUR - 1 , BLACK);

ht1632\_plot(LARGEUR - 1 , HAUTEUR - 2 , BLACK);

}

void loop () {

display\_chargement();

genererLabyrintheParfait();

clear\_chargement();

printLab();

while (1) {

//Boucle infinie

}

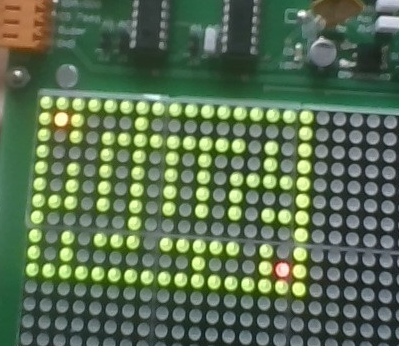
}

# TESTS DU PROGRAMME

Test pour 3x3 (résolution minimale) :

# 

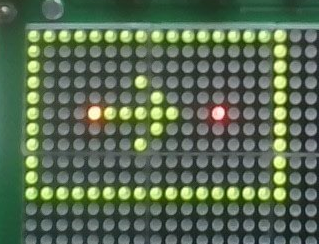
Test pour 8x5 :



Test pour 15x7 (résolution maximale) :



Ecran de chargement :



# CONCLUSION

Ce projet que nous menons depuis le début du module d’IAM commencé fin février nous a permis d’assimiler les techniques utilisées dans le codage de machines concrètes et non virtuelles. Cela est donc bien différent de ce que nous avons pu faire à l’ITEEM auparavant avec l’ADA, ou la programmation web par exemple. On relie en effet programmation et électronique dans ce cas du codage Arduino. Aussi, nous avons pu commencer à apprendre un autre langage de programmation, le C, qui est utilisé pour le codage de cette carte.