



TER

LEDSpinner

Membres du groupe :

Victor JUNG
Steve MALALEL

Encadrants :

Marie PELLEAU
André ANGLADE

4 février 2020

Table des matières

1	État de l'art	2
1.1	Persistance rétinienne	2
1.2	Méthodes de déplacements des LEDs	2
2	Théorie : Comment afficher une image sur une bande tournante ?	3
2.1	Affichage d'une image	3
2.1.1	Image fixe ou image tournante	3
2.1.2	Comment obtenir une image fixe	4
2.2	Construction et compression de l'image	4
2.2.1	Présentation du problème	4
2.2.2	Étapes de conversion	4
3	Conception et fabrication	6
3.1	Base	7
3.2	Support tournant	7
4	Implémentation	9
4.1	Présentation de la bibliothèque FastLED	9
4.2	Algorithme Arduino	10
4.2.1	Définition des constantes et variables globales	10
4.2.2	Setup et routine de démarrage	10
4.2.3	Affichage de l'image	11
4.3	Logiciel de conversion d'image	12
4.3.1	Extraction	13
4.3.2	Réduction des couleurs	14
4.3.3	Compression	14
4.3.4	Assemblage	15
4.3.5	Génération de code	15
4.4	Limites observées du matériel	16
5	Problèmes rencontrés	16
6	Résultats	17
6.1	Conversion d'une image	17
6.2	Affichage sur le LEDSpinner	17
7	Perspectives et réflexions personnelles	18
8	Remerciements	18

Résumé

L'objectif de ce TER est de réussir à afficher une image grâce à une bande de LED et un moteur en se basant sur l'effet de persistance rétinienne. Nous allons voir les différentes étapes afin d'arriver à un tel résultat, en détaillant les idées les plus importantes (côté logiciel) et les branchements des composants (côté matériel). Enfin, nous aborderons les problèmes rencontrés durant ce TER ainsi que les pistes d'amélioration que nous estimons comme faisables et pertinentes.

1 État de l'art

1.1 Persistance rétinienne

Comme dit dans l'introduction nous nous basons sur la persistance rétinienne pour afficher notre image, il faut donc savoir comment cela fonctionne exactement. La persistance rétinienne est un phénomène qui permet aux cellules de la rétine de garder une image en mémoire pendant environ 50ms (cette valeur dépend des sources, mais cela semble être le consensus). Cela signifie qu'une LED doit faire un tour complet en moins de 50ms pour que l'image persiste, et donc que la bande de LEDs doit tourner à une vitesse (minimum) de 20 tours par seconde. Or il y a deux méthodes qui peuvent faire évoluer cette limite lorsque l'on travail avec une seule bande de LEDs.

La première est la symétrie centrale de la bande lors de la rotation (si on choisit de disposer les LEDs à la manière d'une hélice), ce qui veut dire qu'un demi-tour permet d'afficher de nouveau une même couleur au même endroit, la limite devient donc de 10 tours par seconde.

La deuxième méthode vient du fait que plus une image est lumineuse plus la persistance rétinienne va permettre de garder l'image en mémoire longtemps, et cela permet donc de baisser encore la vitesse minimum si on augmente l'intensité des LEDs (mais il faut faire attention car cela peut devenir dangereux pour les yeux).

1.2 Méthodes de déplacements des LEDs

Lors de nos recherches on a pu relever deux principales méthodes de déplacement des LEDs.

La première est celle où la bande de LEDs est posée sur une hélice puis celle-ci est mis en rotation : elle présente l'avantage de permettre de placer les LEDs de manière symétrique, ou encore d'augmenter la résolution de l'image en décalant légèrement la bande de manière à ce que les deux côtés de l'hélice se complètent lors de la rotation.

La deuxième utilise aussi une hélice mais cette fois-ci la bande est placée à une des extrémités de celle-ci de manière perpendiculaire (on peut aussi mettre une bande à chaque extrémité de l'hélice), ce qui forme un cylindre lorsque ça tourne, l'image

s'affichant sur la face courbée de ce cylindre. Cette méthode a l'avantage de faciliter la conversion de l'image car on reste en coordonnées cartésienne pour afficher les pixels.

2 Théorie : Comment afficher une image sur une bande tournante ?

Dans cette section, nous allons faire abstraction du côté matériel : nous allons admettre que notre matériel est fonctionnel, et nous voulons savoir comment peut-on afficher une image sur une bande de LEDs tournante.

Habituellement, une image est représentée comme une matrice : la première ligne de la matrice correspond à la première ligne de l'image, et la première colonne de la matrice correspond à la première colonne de l'image. Nous sommes donc dans une représentation cartésienne : la couleur d'un pixel est obtenue à l'aide de ses coordonnées cartésiennes (disons $(0,0)$ pour le premier pixel en haut à gauche). Cependant, dans le cadre de notre TER, nous ne sommes justement PAS dans une représentation cartésienne mais polaire, c'est-à-dire dans notre cas que la couleur d'un pixel est obtenue grâce à la distance du pixel par rapport à l'origine et son angle.

Dans cette section, nous allons tout d'abord parler de comment afficher une image sur la bande, puis de comment obtenir cette image (passage du cartésien au polaire et compression).

2.1 Affichage d'une image

On suppose dans cette partie que l'on dispose d'une image dont il est possible d'obtenir la couleur d'un pixel à l'aide de sa distance et de son angle.

2.1.1 Image fixe ou image tournante

Lorsque l'on décide d'afficher une image sur le LEDSpinner, nous avons deux options : soit obtenir une image "tournante", soit obtenir une image "fixe".

Image tournante On décide d'afficher les pixels les uns après les autres sans se soucier de la vitesse de rotation. Cette solution est la plus simple car elle nous permet de savoir que, lorsqu'on se trouve à l'étape i , l'étape suivante est l'étape $i + 1$. Elle permet également d'obtenir une très bonne compression de l'information, car on peut se permettre d'allumer (ou pas) une LED en se basant sur son état à l'étape précédente. Cependant, cela aura comme effet d'avoir l'impression que l'image "tourne", à cause de la vitesse de rotation et de la vitesse de calcul.

Image fixe Cette option est un peu plus complexe que la précédente. En effet, cette fois on se base sur la position de la bande de LED pour savoir quelle suite de couleur afficher. Cela signifie donc que l'on abandonne le postulat qui est que l'étape i est celle qui précéde l'étape $i + 1$. Cela signifie que nous n'avons plus la possibilité

de compresser autant l'image qu'avant, et que nous devons également effectuer plus de calculs. Nous aborderons plus en détail cette option dans la section suivante.

2.1.2 Comment obtenir une image fixe

La première étape pour obtenir une image fixe est de savoir approximativement combien de temps met le LEDSpinner pour faire un tour complet. On suppose qu'il est possible de savoir combien de temps dure un tour. En sachant le temps qu'il est lors du début du tour, nous sommes en mesure d'estimer, en nous basant sur la durée du tour précédent, à quel pourcentage de la rotation nous sommes à un temps donné.

$$\theta = \frac{\text{current_turn_time}}{\text{last_rotation_time}}$$

Sachant cela, la seconde étape consiste à afficher la ligne de LED correspondante. On transforme ce pourcentage en index par rapport à la taille du tableau représentant l'image. Pour finir, on affiche la suite de couleur correspondant à la suite de couleur à l'index $\lfloor \theta \times \text{image_size} \rfloor - 1$, avec image_size = la taille de l'image générée (en pixel).

Remarque 1 Il arrive que θ soit supérieur à 1 dans le cas où la vitesse de rotation décroît par rapport au tour précédent. Dans ce cas, on borne θ à 1.

Remarque 2 La véritable formule pour obtenir l'index est $\lfloor \theta \times \text{image_size} \times 2 \rfloor - 1$, car nous avons seulement besoin de 180° , tandis qu'un tour est de 360° (l'image correspondant à $(180 + x)^\circ$ est l'inverse de l'image correspondant à x°).

2.2 Construction et compression de l'image

2.2.1 Présentation du problème

Notre but est d'arriver à convertir une image de manière à ce qu'elle soit affichable sur notre LEDSpinner. Il faut donc trouver un moyen d'obtenir la couleur d'un pixel en fonction d'un angle et d'une position au lieu de deux positions : en d'autres termes, passer l'image de coordonnées cartésiennes à des coordonnées polaires.

De plus, nous sommes limités à l'affichage de n pixels, n étant la taille de la bande de LED (c'est-à-dire le nombre de LEDs présentes sur la bande).

2.2.2 Étapes de conversion

Passage du cartésien au polaire

Notre première étape est de transformer une image de coordonnées cartésiennes en une image de coordonnées polaires. Pour se faire, nous avions tout d'abord pensé à extraire ligne par ligne en utilisant l'algorithme de Bresenham. L'idée est d'obtenir à partir de la ligne de degré θ le pixel de départ (s_1, s_2) et le pixel de fin (t_1, t_2). On obtient ensuite une ligne traversant l'image du point de départ vers le point d'arrivée : tous les pixels appartenant à la ligne sont extraits. L'inconvénient de cette technique est qu'on effectue une conversion complète de l'image de cartésien

vers polaire, ce qui est inutile étant donné que nous avons seulement un certain nombre pixels à échantillonner.

La deuxième idée que nous avons eu, la plus simple et celle que nous avons gardé, est tout simplement de calculer les pixels à l'aide de la fonction suivante :

$$\forall x \in [0, n/2[, \forall \theta \in [0, 180[:$$

$$pos(x, \theta) = (\cos\theta \times x \times r + half, \sin\theta \times x \times r + half)$$

avec θ = l'angle, $r = \frac{image_size}{2n}$, $half = \frac{image_size}{2}$, $image_size$ = la taille de l'image et n = le nombre de LEDs.

Nous obtenons ainsi une matrice de taille $180 \times n$.

Compression des couleurs

Afin de ne pas avoir de couleurs approximatives, nous décidons d'effectuer une compression sur les couleurs. De plus, cela permet également de compresser la taille de l'information, ce qui est toujours une bonne chose. Pour cela, nous avons plusieurs fonctions :

- Arrondir chaque valeur R, G et B selon un paramètre k .
Soit la fonction $compress(c, k) = \min(\lfloor c/k \rfloor * k, 255)$ avec $c \in [0, 255]$.
- Ramener chaque valeur à 0 ou 255 selon un seuil.
Soit la fonction $3bit(c) = \lfloor c/127 \rfloor * 255$.
- Transformer l'image en noir et blanc.
Soit la fonction

$$bw(R, G, B) = \begin{cases} 255 & \text{if } R+G+B > 381 \\ 0 & \text{sinon} \end{cases}$$

La fonction $3bit(c)$ est la fonction qui pose le moins de soucis à l'affichage, car la fonction renvoie un maximum de 8 couleurs différentes et que les couleurs sont "élémentaires" (et donc facile à différencier et afficher), tout en restant plus fidèle à l'image de base qu'avec une transformation en noir et blanc.

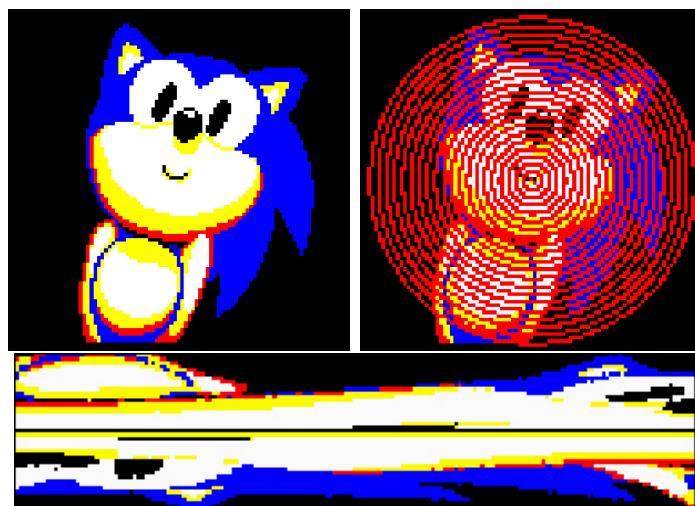


FIGURE 1 - (1) Compression couleur (2) Points extraits (3) Image obtenue

Échantillonnage de l'image polaire

La dernière étape de la création de l'image est l'échantillonnage. Comme nous l'avons dit précédemment, nous avons une matrice de taille $180 \times n$. Mais est-ce vraiment utile ?

En étudiant la chose plus attentivement, on remarque qu'il y a une redondance de l'information. Le cas le plus facile à voir est celui de la LED du centre : quelque soit l'angle de rotation, la couleur reste la même (étant donné que la distance est de 0 par rapport au centre, par définition) (FIGURE 1 - (3)).

Dans un cas plus général, on remarque que plus la distance est élevée, plus on a de diversité dans l'information. Cette diversité évolue de manière linéaire ; nous avons donc décidé d'effectuer un échantillonnage linéaire du nombre de pixel en fonction de la distance par rapport au centre de l'image.

Problème Le problème de cette méthode et que nous ne pouvons plus représenter l'image sous forme de matrice, car le nombre d'informations dépend du numéro de la LED. Nous avons donc pensé à un moyen de représenter l'image à l'aide de deux tableaux : l'un contenant toutes les informations (tous les différents pixels échantillonnés) et l'un contenant l'index du début de la portion pour chaque numéro de LED.

Le tableau *index* est donc de taille $n + 1$ et l'autre (*pixels*) est d'une taille variable (en fonction de la taille de l'échantillonnage et du nombre de LED).

Pourquoi $n + 1$? Si on veut savoir le nombre de pixels pour chaque LED, on peut effectuer le calcul suivant : $nb_i = index_{i+1} - index_i$. Si $i = n$, alors il nous faut bien $n + 1$ valeurs pour ne pas avoir une erreur de limite d'indexation.

Remarque $index_{n+1}$ correspond à la taille du tableau *pixels*.

Ainsi, pour obtenir le pixel p correspondant à la LED i et d'angle θ , nous devons effectuer le calcul suivant :

$$p = pixels[\lfloor \theta \times (index[i + 1] - index[i] - 1) \rfloor + index[i]]$$

3 Conception et fabrication

Dans cette section nous allons parler du matériel utilisé pour accueillir notre code. N'ayant que peu de compétences en électronique, un premier modèle créé par Marie Pelleau et André Anglade nous a été fourni au début de nos travaux pour que l'on puisse y implémenter notre programme. Nous avons par la suite remplacé certaines parties de ce modèle pour améliorer les performances et possibilités disponibles, mais la majeure partie des branchements sont restés les mêmes.

Nous allons principalement détailler les branchements effectués pour que tout soit fonctionnel. Notre LEDSpinner est composé de deux principales parties : le support tournant avec la bande de LEDs et la base sur laquelle va être posé ce support pour qu'il puisse tourner.

3.1 Base

Tout d'abord voici un schéma qui représente les branchements de la base :

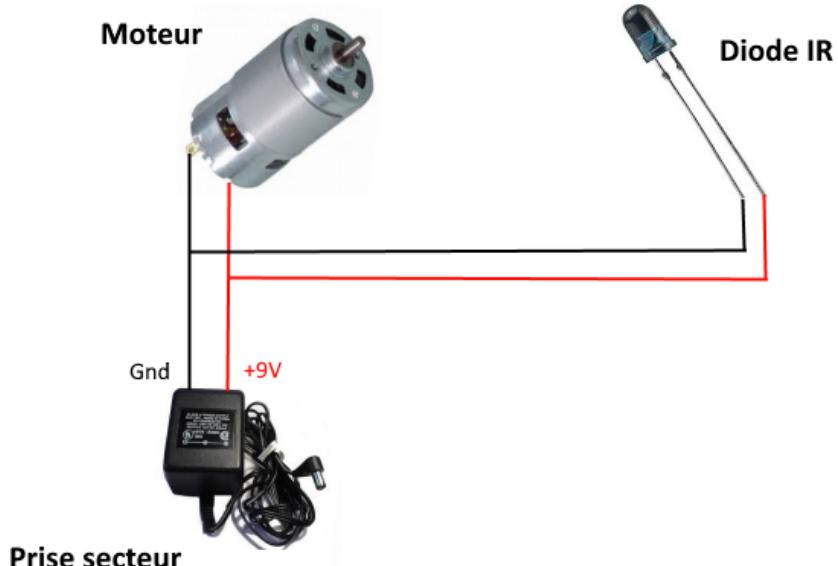


FIGURE 2 - Branchements base

Comme on peut le voir une prise secteur va alimenter le moteur pour faire tourner le support, mais aussi une diode pour qu'elle émette un rayonnement infrarouge, nous verrons dans la partie du support tournant l'utilité de cette diode.

Grâce à une imprimante 3D nous avons pu recréer une base destinée à accueillir tous ces éléments :



FIGURE 3 - (1) Base (2) Base vu de dessous

Elle est légèrement plus haute que l'ancienne base car les fils passent désormais à l'intérieur.

3.2 Support tournant

Le support est plus complexe : il doit contenir tout un ensemble d'éléments lui permettant d'être autonome en terme d'alimentation, doit pouvoir détecter quand

est-ce qu'il a fait un tour complet et enfin doit bien évidemment afficher les LEDs comme il faut. Voici le détail des branchements :

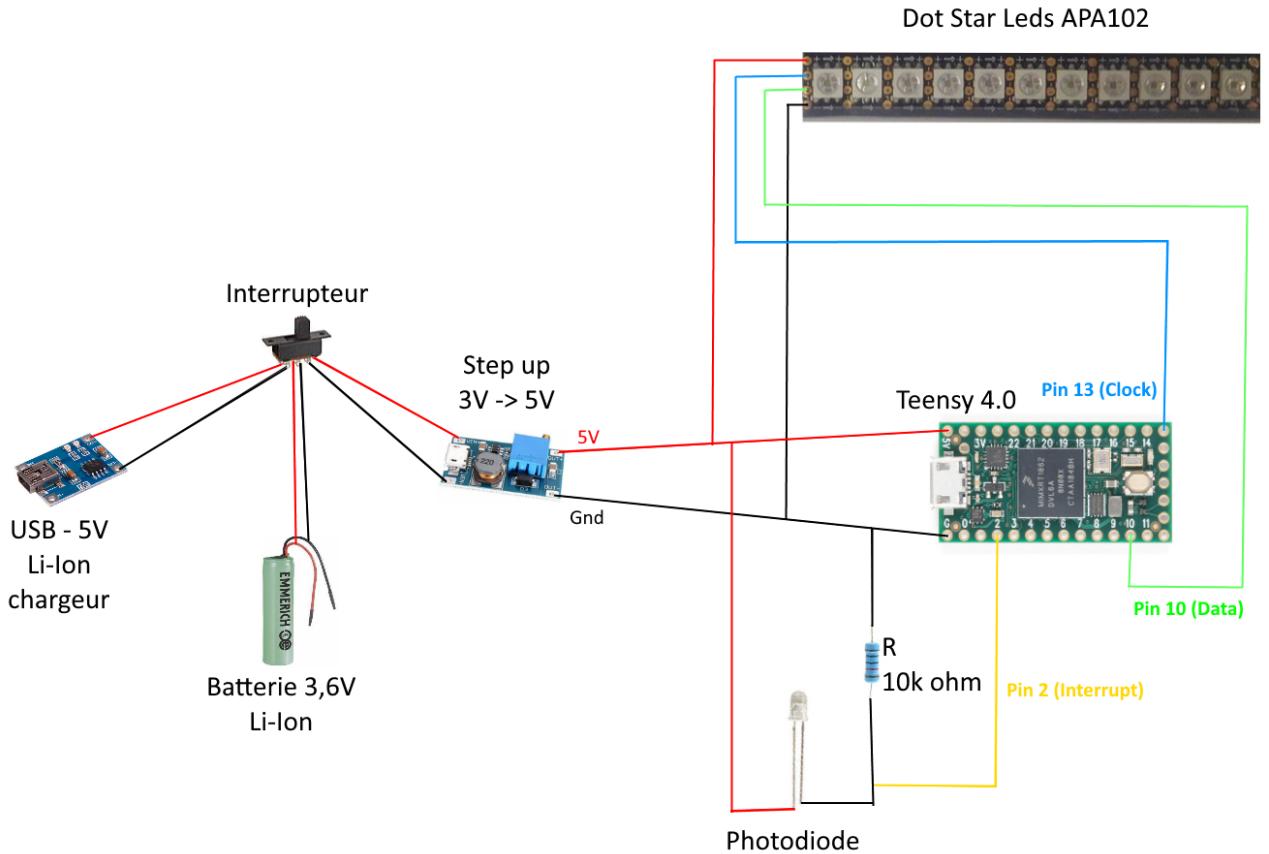


FIGURE 4 - Branchements support tournant

L'alimentation Sur la partie gauche du schéma se trouve les composants concernant l'alimentation. On a bien évidemment une batterie pour que tous les éléments puissent marcher sans être branchés sur secteur, mais aussi un chargeur USB pour pouvoir recharger cette batterie. Un interrupteur est présent pour permettre l'alimentation du reste du circuit en temps voulu, et un Step up permet ensuite d'alimenter le reste des composants avec une puissance de 5V.

Au niveau de leur disposition, la batterie étant l'élément le plus lourd elle est placée au milieu du support pour éviter un déséquilibre et faire en sorte que le centre de gravité soit le plus au milieu possible. L'interrupteur et le chargeur sont placés de manière à être accessibles depuis l'extérieur.

La détection d'un tour Afin de savoir si le support a fait un tour complet, une photodiode est reliée à la Teensy et lui enverra un signal quand elle même recevra un signal provenant de la diode infrarouge présente sur la base. La photodiode est située en dessous du support et au même niveau que la diode infrarouge par rapport à l'éloignement du centre de rotation.

L'affichage des leds Une bande de LEDs de type APA102 est reliée à la Teensy par les PINs 13 pour l'horloge et 10 pour les données (à noter que pour les données

c'est le PIN 11 qui aurait du être utilisé, mais suite à un dysfonctionnement de celui-ci c'est finalement le PIN 10 qui est utilisé car il permet lui aussi l'envoi des données vers la bande de LEDs).

La Teensy se situe à l'intérieur du support et est accessible pour pouvoir téléverser du code dessus. La bande de leds se situe sur la partie supérieur du support, recouvrant tout le circuit présent dans celui-ci.

Nous avons décidé de réutiliser le même support qui nous a été fourni, celui-ci a été aussi imprimé en 3D :



FIGURE 5 - (1) Support vide (2) Support monté (3) Photodiode en dessous du support

4 Implémentation

Nous allons dans cette section présenter l'implémentation du code en deux parties. Premièrement, le code Arduino (en C++), et ensuite le code de création d'image (en Python).

L'ensemble du code est disponible sur notre Github.

4.1 Présentation de la bibliothèque FastLED

FastLED est une bibliothèque Arduino permettant de gérer l'interaction entre la carte Arduino ou Teensy et la bande de LED. Cela nous permet de choisir quelle LED allumer et de quelle couleur, et de gérer automatiquement les échanges entre les composants. Il suffit de déclarer le nombre de LEDs et les différents PINs, de lier les deux par l'appel d'une fonction et c'est tout.

4.2 Algorithme Arduino

Comme nous l'avons vu dans les sections précédentes, il nous suffit pour afficher une image d'avoir :

- Un accesseur pour obtenir la couleur d'un pixel en fonction de sa distance et d'un angle (donné par notre méthode de conversion),
- Savoir quand on vient d'effectuer un tour (donné grâce à la diode)

Afin de simplifier le code, nous avons découpé celui-ci en deux parties : une partie principale chargée de gérer l'affichage de l'image (ou des images), et une autre partie qui contient cette ou ces images (que l'on *include* dans la première partie).

Nous allons ici nous intéresser plus particulièrement à la première partie, car la deuxième est directement générée lors de la création de l'image (que nous verrons plus tard).

4.2.1 Définition des constantes et variables globales

Constantes Les constantes correspondent aux numéros des PINs utilisés pour l'envoi d'informations (DATA_PIN), à l'horloge (CLOCK_PIN) et enfin à l'interruption (pour la diode, INTERRUPT_PIN).

Variables globales Comme on a pu le présenter plus haut, nous avons besoin de connaître différentes informations : le temps du début du tour (*turn_start*), la durée de la dernière rotation (*last_rotation*) et le temps passé depuis le début du tour (*current_turn*). De plus, on définit deux autres variables globales : le tableau de LEDs (*leds*) et l'index de l'image qu'on est en train d'afficher (IMAGE, utile seulement dans le cas d'une image animée).

4.2.2 Setup et routine de démarrage

Setup Le setup est la fonction la plus importante, car c'est dans celle-ci que l'on va lier les différents composants. Dans notre cas, nous avons deux choses à lier : la bande de LEDs et la diode.

`FastLED.addLeds<APA102, DATA_PIN, CLOCK_PIN, BGR>(leds, NUM_LEDS)` nous permet de lier la bande de LED à la carte. On déclare que la bande de LED est une bande APA102 (1), qu'elle est branchée sur le PIN DATA_PIN (2) et CLOCK_PIN (3), et que le mode de couleur utilisé est BGR (Blue Green Red). Enfin, on précise quelle est la variable correspondante aux LEDs (4) et combien on a de LEDs (5).

`attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), oneTurn, RISING)` nous permet de lier un signal reçu sur le PIN INTERRUPT_PIN (1) à une fonction oneTurn (2) qui sera automatiquement appelée lors de la réception du signal (et qui aura la priorité sur tout le reste). Enfin, on déclare que nous déclençons le signal lors du début de la réception (RISING (3)).

Routine de démarrage Afin de tester le fonctionnement des LEDs, nous effectuons une petite routine de démarrage consistant à allumer puis éteindre les LEDs une par une afin de nous assurer dans un premier temps que les lumières fonctionnent, et dans un

second temps que la transmission d'information entre la bande et la carte fonctionne. Cette fonction est la fonction `LED_test()`.

4.2.3 Affichage de l'image

Obtenir l'angle Comme nous l'avons déjà expliqué dans la section théorique, pour obtenir l'angle on se base sur la durée du tour précédent (variable `last_turn`) et le temps passé depuis le début du tour (`current_turn`). Nous obtenons la variable $\theta = \text{current_turn}/\text{last_turn}$, qui n'est pas vraiment un angle mais le pourcentage de complétion du tour (situé normalement entre 0 et 1, mais nous avons déjà discuté de ceci précédemment).

Obtenir la couleur d'un pixel Afin d'optimiser l'espace mémoire, nous avons créé deux tableaux pour obtenir la couleur : un contient chaque couleur de l'image (`COLORS`), et l'autre contient l'index de la couleur du pixel (`PIXELS`, le tableau comme décrit dans la section théorique). Cela permet de ne pas avoir plusieurs objets créés pour une seule et même couleur. On peut donc obtenir la couleur d'un pixel en faisant l'accès suivant : `COLORS[PIXELS[image][index]]`, avec `image` l'index de l'image actuelle (pour les images animées) et `index` l'index du pixel que l'on veut afficher.

Obtenir l'index d'un pixel En sachant le pourcentage de complétion du tour θ , on peut obtenir l'index d'un pixel dans le tableau `PIXELS` avec la formule suivante : $p = \text{PIXELS}[|\theta \times (\text{POSITIONS}[i + 1] - \text{POSITIONS}[i] - 1)| + \text{POSITIONS}[i]]$.

Cette formule a déjà été présentée dans la section théorique. Cependant, il y a une précision à apporter concernant θ . En effet, nous avons dit qu'à partir de 180° (soit $\theta = 0.5$, la moitié d'un tour) il suffisait de “renverser” l'image ; $180 + x = \text{inv}(x)$. Ce qui signifie que pour nous, $180^\circ \equiv 1$, et non 0.5. Pour obtenir le réel pourcentage de complétion, il faut effectuer le calcul suivant :

```

si theta > 1:
    theta := 1
si theta > 0.5:
    theta = (theta - 0.5) * 2
    afficherImageInverse()
sinon:
    theta = theta * 2
    afficherImage()

```

Dans notre cas, “inverser” l'image signifie attribuer au pixel i la couleur du pixel $n - i - 1$ (avec n le nombre de LEDs). `leds[i] = COLORS[p]` permet ensuite d'allumer la LED i avec la couleur `COLORS[p]`.

On appelle cette fonction `drawPicture(θ)`, et on crée une fonction `draw()` qui effectue le calcul de θ et qui appelle la fonction `drawPicture(θ)`.

Obtenir θ Pour obtenir le pourcentage de complétion d'un tour, il faut déjà savoir combien de temps prend un tour. Pour cela, on se base à chaque fois sur le temps du tour précédent ; c'est le plus simple, mais pas forcément le plus efficace. En effet, si la vitesse de rotation varie de façon significative entre deux tours, alors la valeur θ sera fausse et l'image ne s'affichera pas correctement. Cependant, la vitesse de rotation semble rester stable (sauf lors du lancement où la vitesse croît fortement, mais cela ne dure même pas une seconde), donc cette méthode de calcul de θ marche dans les faits. On mesure le temps

à l'aide de la fonction millis() qui renvoie le temps en milisecondes.

On appelle cette fonction computeDegree().

Mettre à jour l'image à afficher (animation) Dans le cas d'une image animée, on définit un nombre d'images par seconde (IPS ou FPS en anglais pour Frames Per Second) à afficher. Cependant, ce qui nous intéresse nous est le nombre de secondes par image. On effectue donc (en amont, puisqu'il s'agit d'une constante) le calcul de l'inverse des IPS $IMAGE_TIME = 1/FPS$ et on regarde à l'aide d'un timer si le temps écoulé depuis le début de l'affichage de l'image dépasse cette quantité. Si oui, alors on passe à l'image suivante. Une animation est cyclique, il faut donc, lorsque l'on arrive à la fin, faire attention à repartir depuis le début.

Cette fonction s'appelle updateImage().

Mettre à jour les variables de tour Comme on l'a vu dans la partie setup, on a lié une fonction oneTurn avec la réception d'un signal au niveau de la diode : c'est cette fonction qui va mettre à jour le temps mesuré pour le dernier tour. $last_rotation = current_turn$, $current_turn = 0$. On n'oublie évidemment pas de mettre à jour le timer du début de tour, $turn_start = millis()$.

Cette fonction s'appelle updateTimers().

Boucle principale La fonction loop() est la fonction principale qui s'exécute à chaque tic d'horloge. Cette fonction loop() consiste à appeler les fonctions updateTimers(), updateImage() et draw().

4.3 Logiciel de conversion d'image

Afin de convertir des images de sorte à ce qu'on puisse ensuite les afficher sur notre bande de LEDs, nous avons créé un petit logiciel permettant :

- D'ajouter (ou supprimer) les images à convertir
- De spécifier le nombre de LEDs sur la bande
- De choisir le nombre d'échantillons maximum
- De choisir de compresser les couleurs (méthode *compress* et *3bit*, voir partie théorique)
- De choisir la vitesse d'animation (FPS) et d'avoir un aperçu de cette dernière.

Ce logiciel est simple d'utilisation, et propose de sauvegarder directement en “image.h”, comme attendu par le code Arduino. Il suffit de placer le fichier généré au même endroit que le code Arduino. C'est ce code qui contient les différentes variables utilisées dans le code principal, à savoir : POSITIONS, COLORS, PIXELS, NUM_LEDS, N_IMAGES et IMAGE_TIME.

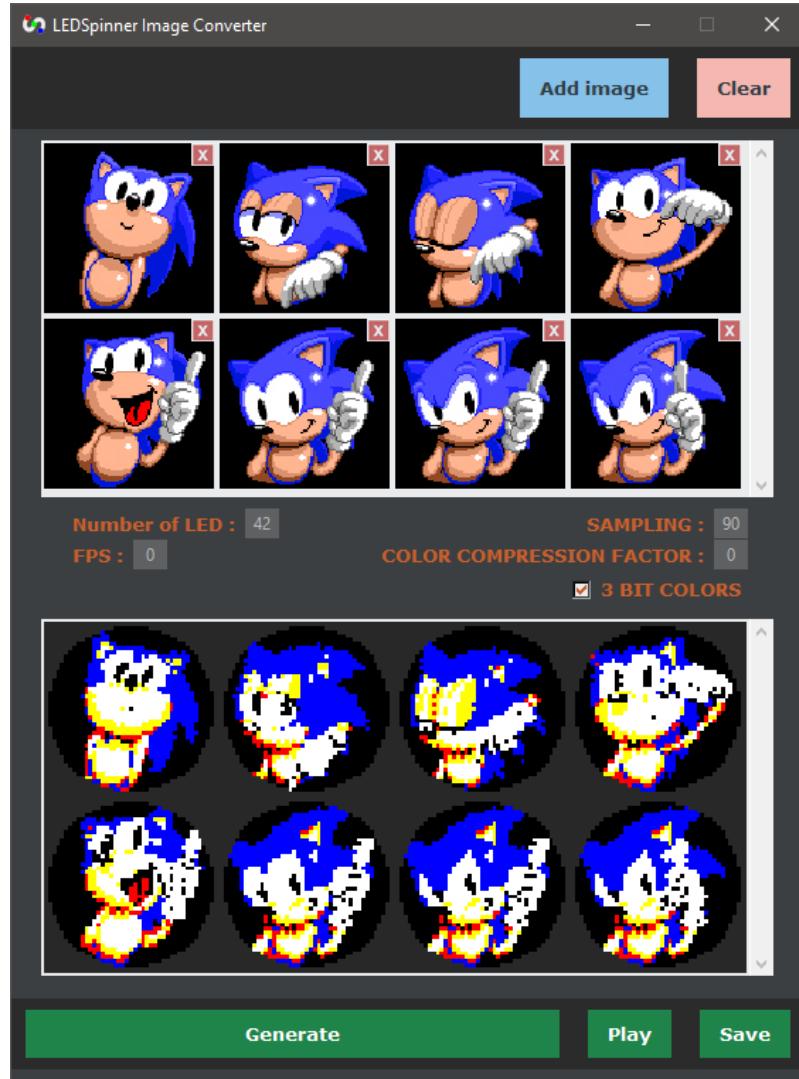


FIGURE 6 - Interface du logiciel

Nous allons dans cette section aborder les différentes étapes de la conversion. Étant donné que nous avons déjà expliqué les principes fondamentaux dans la partie théorique, nous n'aborderons pas les méthodes utilisées très en détail.

4.3.1 Extraction

L'algorithme d'extraction ne change pas de celui présenté dans la partie théorique. La méthode utilisée reste la même. On implémente une légère optimisation au niveau de l'extraction en profitant du calcul de la position d'un pixel, ici $(xPos1, yPos1)$ pour obtenir le pixel "inverse" $(xPos2, yPos2)$. Autrement, rien ne change.

```
def extract(img, LED=SIZE):
    size = min(len(img), len(img[0]))
    ratio = size / LED
    res = []
    HALF = LED // 2
    for i in range(180):
        radian = i / 180 * np.pi
        cos = np.cos(radian)
        sin = np.sin(radian)
        res.append([[0, 0, 0] for o in range(LED)])
    for j in range(HALF):
```

```

        k = j * ratio
        xPos1 = int(np.round(cos * k + size / 2))
        yPos1 = int(np.round(sin * k + size / 2))
        xPos2 = size - xPos1
        yPos2 = size - yPos1
        res[i][j + HALF] = mean(img, xPos1, yPos1, 0)
        res[i][HALF - j - 1] = mean(img, xPos2, yPos2, 0)
    return res

```

4.3.2 Réduction des couleurs

Comme expliqué dans la partie théorique, on dispose de plusieurs fonctions afin de réduire le nombre de couleurs. Principalement, nous avons décidé d'utiliser la méthode $compress(c, k)$ et la méthode $3bit$. L'implémentation n'est pas très intéressante, nous avons donc décidé de ne pas montrer de code. Nous en parlons cependant car il s'agit d'une option dans le logiciel que nous avons créé.

4.3.3 Compression

Nombre d'échantillons L'implémentation de l'algorithme de compression est assez simple. On commence tout d'abord par générer un tableau qui, pour chaque LED, nous donne le nombre d'échantillons à garder. On en profite également pour générer le tableau POSITIONS au même instant. Étant donné que notre bande de LEDs est symétrique par rapport au centre, on se permet de générer seulement la moitié du tableau à chaque fois puis de concaténer ce même tableau renversé. Comme nous l'avons déjà expliqué, nous utilisons une méthode d'échantillonage linéaire.

```

def linear(num_led, max_size):
    mid = int(np.ceil(num_led / 2))
    index_size = [round(i * (max_size / mid) + 1) for i in range(mid)]
    return build_positions(index_size)

```

```

def build_positions(sizes):
    sizes = symmetry(sizes)
    index_position = [0]
    for i in range(1, len(sizes) + 1):
        index_position.append(index_position[i - 1] + sizes[i - 1])
    return sizes, index_position

```

Compression Étant donné le tableau généré précédemment, pour chaque LED, on échantillonne le nombre de pixels correspondant au nombre donné dans le tableau. Pour rappel, l'échantillonage est effectué sur l'image “extraite”, c'est à dire de taille $180 \times n$.

```

def compress(img, SAMPLING_SIZE=45, size=SIZE):
    compression_size, positions = linear(size, SAMPLING_SIZE)
    compressed = []
    for i in range(size):
        s = compression_size[i]
        for j in range(s):
            index = round(180 * j / s)
            compressed.append(img[index][i])
    return compressed, positions

```

4.3.4 Assemblage

On définit la fonction compute comme étant la fonction qui génère l'image et les différents tableaux nécessaires à la génération de code. Il s'agit d'un assemblage des différentes fonctions présentées précédemment.

```
def compute(image, led_num=42, compress_color=False,
           color_approx_factor=50, compressed_index_method=LINEAR,
           sampling_size=90, c=(0, 0, 0), bit3=True):

    new_img, img = extract(image, led_num)
    if bit3:
        new_img = bit(new_img)
    elif compress_color:
        new_img = compress_img_color(new_img, color_approx_factor)
    compressed, positions = compress(new_img, compressed_index_method,
                                      sampling_size, led_num)
    reconstruction = reconstruct_img(compressed, positions, led_num, c)

    return new_img, compressed, reconstruction, positions
```

4.3.5 Génération de code

Après avoir généré tous les tableaux nécessaires, il ne nous reste plus qu'à générer le code prêt à être utilisé par l'Arduino. Tout d'abord, on commence par construire le code du tableau de l'image (PIXELS) et du tableau de couleurs (COLORS). Étant donné qu'il est possible d'avoir plusieurs images, on doit à chaque fois garder le même tableau de couleurs. Ainsi, pour générer le tableau de l'image, on regarde si la couleur du pixel est déjà présente dans le tableau couleur : si non, alors on ajoute la couleur au tableau, si oui on ne fait rien. On ajoute ensuite l'index de cette couleur à la suite du code généré.

```
def generate_code_array(image, colors):
    code = "{"
    for pixel in image[:-1]:
        color = "CRGB(%s, %s, %s)" % (pixel[0], pixel[1], pixel[2])
        if color not in colors:
            colors[color] = len(colors)
            code += "%s," % colors[color]
    pixel = image[len(image)-1]
    color = "CRGB(%s, %s, %s)" % (pixel[0], pixel[1], pixel[2])
    if color not in colors:
        colors[color] = len(colors)
        code += "%s" % colors[color]
    return code+"}"
```

Une alternance de blanc/noir donnera par exemple :

```
PIXELS = {0, 1, 0, 1, 0, 1, 0, 1}
COLORS = {CRGB(255, 255, 255), CRGB(0, 0, 0)}
```

En répétant ceci pour chaque image, et en ajoutant les constantes du nombre de LEDs et du nombre d'images, on arrive à générer un fichier comme celui-ci :

```

#define NUM_LEDS 10
#define N_IMAGES 3
const int POSITIONS[11] = {0, 3, 5, 7, 8, 9, 10, 11, 13, 15, 18};
const int PIXELS[N_IMAGES][18] = {
    {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
const CRGB COLORS[2] = {CRGB(0, 0, 0), CRGB(255, 255, 255)};
const float IMAGE_TIME = 200.0;

```

4.4 Limites observées du matériel

Mémoire Arduino Nous avons rencontré très vite une première limitation du côté matériel : la mémoire. En effet la mémoire disponible sur l’Arduino ne permet pas de sauvegarder dans une variable une très grande quantité de donnée (alors que nous en avons besoin pour afficher une image nette avec des couleurs varié). Typiquement, on peut stocker un tableau de taille 32×45 . De plus, le nombre de LED présente sur la bande limite la résolution de notre image, mais cela reste moins contraignant que le problème de mémoire.

Pour résoudre ce problème, nous avons dans un premier temps passé le code sous forme de fonctions, afin d’utiliser la mémoire “sketch” d’Arduino (qui correspond à l’espace mémoire dans lequel les fonctions sont gardées, environ 30 KB) qui est plus élevée que la mémoire “dynamique” (espace mémoire dans lequel les variables globales sont gardées, environ 2 KB). Ainsi, on pouvait afficher une animation constituée de 6 images avec un échantillonnage de 45 ($6 \times (42 \times 45)$). Dans un second temps, nous avons remplacé la carte Arduino par une carte Teensy ayant beaucoup plus de mémoire (environ 1000 fois plus, dynamique et sketch).

Vitesse de rotation et IPS Lors d’une animation, on définit un nombre d’IPS (image par seconde). Cependant, si la vitesse de rotation ne permet pas d’afficher au moins complètement l’image avant de passer à la suivante, alors l’affichage sera désastreux. Il en va de même pour l’effet de persistance rétinienne (comme nous en avons parlé dans l’état de l’art).

5 Problèmes rencontrés

En plus des limites observées nous avons rencontrés des dysfonctionnements au niveau du matériel. Comme dit plus haut dans la partie électronique, le PIN 11 ne fonctionne pas comme il faut, mais ça n’a pas été simple de déterminer que le problème venait de là. Lorsque nous avons remplacé l’Arduino par la Teensy, nous avons eu un problème au niveau des LEDs : l’affichage ne se faisait pas du tout comme il fallait. Au début nous avons pensé que le problème pouvait venir de plusieurs sources : la bande de LEDs, l’alimentation, une mauvaise installation de la bibliothèque Teensy sur l’ordinateur, ou encore que la Teensy elle-même ne fonctionnait plus. Après plusieurs expérimentations et grâce à André Anglade notre sauveur, nous avons pu voir à l’aide d’un oscilloscope que le message envoyé par le PIN 11 était trop rapide pour la bande de leds, et un remplacement par un autre PIN, le 10, a réglé le problème.

Cependant un autre problème est survenu et provoque le fait que le LEDSpinner ne fonctionne pas forcément : nous ne savons pas à l’heure actuelle quelle en est l’origine mais

nous pensons que c'est dû à un faux contact car le problème disparaît et réapparaît sans que l'on fasse quoi que ce soit.

6 Résultats

6.1 Conversion d'une image

Concernant la conversion d'une image, nous sommes très satisfait du résultat. En effet, on lorsque l'on génère les images avec un nombre de LEDs élevé (disons la taille de l'image de base), on est en mesure de clairement identifier l'image, mais avec une résolution deux fois moindre (et en polaire).

De plus, lorsque l'on compare une image échantillonnée de manière constante (c'est-à-dire qu'on échantillonne le même nombre de pixel pour chaque LED) et une image échantillonnée de manière linéaire (comme présenté précédemment) de même taille (typiquement, si c est le nombre d'échantillons constant, alors on échantillonne de manière linéaire jusqu'à $2c - 1$), alors on constate que l'échantillonnage linéaire est bien supérieur. En réalité, on perd de la qualité au niveau du centre de l'image (ce qui n'est pas très important car moins d'informations) mais on gagne en qualité au niveau des extrémités (ce qui est plus important car plus d'informations).



FIGURE 7 - Image de base (1) Echantillonnage constant $c = 45$ (2)
Echantillonnage linéaire $c = 89$ (3)

6.2 Affichage sur le LEDSpinner

Comme évoqué durant les problèmes rencontrés, nous n'avons pas pu obtenir de résultats sur la nouvelle carte Teensy. Cependant, nous avions pu obtenir des images sur la version précédente du prototype (et donc également sur une version précédente du code, malheureusement). La seule chose ayant changé est la nouvelle méthode de compression ; le code Arduino reste sensiblement le même (quelques modifications pour coller au nouveau style de représentation de l'image, c'est-à-dire de compression) et les méthodes d'extraction et de réduction des couleurs restent exactement les mêmes. Nous restons donc assez confiants malgré les problèmes électroniques rencontrés quant aux résultats que l'on pourrait obtenir.

Voici donc le lien vers deux vidéos que nous avons pu tourner. Ces vidéos ont été tournées avec une caméra bénéficiant d'un bon taux de rafraîchissement (60 FPS), ce qui explique que l'effet de persistance rétinienne n'est pas forcément visible.

VIDÉO 1 - Image statique

VIDÉO 2 - Image animée

7 Perspectives et réflexions personnelles

Dans cette section, nous allons partager les idées que nous avons eu et que nous estimons comme réalisables et intéressantes, et que nous aurions aimé faire si nous avions eu plus de temps.

Décalage de la bande Une première façon d'améliorer le prototype est de décaler la bande de LED de façon à ce qu'il n'y ait plus de centre (décalage de 75% de la largeur d'une LED). De cette façon, lorsque la bande tourne, elle ne passe pas deux fois par le même endroit (qui est une propriété que nous avons exploitée pour "renverser" l'image passé les 180°). Cela demanderait de refaire une nouvelle méthode d'échantillonnage sur 360° (au lieu de 180°).

Réaliser ce décalage permettrait donc de doubler la résolution d'une image ; mais il existe une autre façon d'y arriver.

Ajout d'une deuxième bande En ajoutant une deuxième bande de LED avec un décalage d'1/2 LED par rapport à la première, cela permettrait également de doubler la résolution de l'image. De plus, cela permet de garder les mêmes méthodes d'échantillonnage et de compression (il suffit d'introduire un décalage lors de l'échantillonnage). Il faudra cependant faire attention à la communication entre bandes et carte.

Ajout d'une carte mémoire Nous avons eu l'idée d'ajouter une carte SD afin de pouvoir stocker les informations des images sans pour autant se reposer sur la mémoire de la carte (qui peut être limitée, par exemple avec la première carte Arduino). Cependant, cela donnerait lieu à de nouveaux problèmes, comme la vitesse de lecture sur la carte SD, ou bien le simple fait de trouver la place pour ajouter un lecteur de carte SD.

Communication WiFi Une dernière piste que nous aurions aimé explorer est l'ajout d'un module WiFi communiquant avec la carte Arduino (ou Teensy). Ce module WiFi permettrait de recevoir les informations de l'image à afficher, ce qui permettrait de résoudre le problème de mémoire. De plus, cela permettrait de choisir à la volée quelle image afficher, et également de ne pas recompiler le code à chaque fois que l'on veut changer l'image (mais on devra changer le code quand même à chaque fois que l'on veut changer de réseau!). Cependant, cela ajoute dans l'équation les problèmes inhérents à la communication (combien de temps pour recevoir l'information, est-ce que toute l'information a été reçue etc...).

8 Remerciements

Nous tenons à remercier nos encadrants Marie Pelleau et André Anglade pour le matériel fourni et surtout pour nous avoir aidé et enseigné des notions concernant la partie électronique qui nous était moins familière. Nous voulons aussi remercier le FabLab qui nous a permis d'accéder à tous les outils dont nous avions besoin pour réaliser notre TER.