

LED Wall

DII4 2018 - 2019



POLYTECH[®]

Apprentis :

LOCHE Jérémy
SIMMOU Soufiane
LE NEL Corentin

THOMAS Louis
DOROTHÉE Corentin
RICHARD Arthur



Encadrants : ESSWEIN Carl ROLLAND Alexis

| | |
|--|-----------|
| Remerciements | 5 |
| Introduction | 6 |
| Présentation & Contexte du projet | 7 |
| <i>Expression des besoins</i> | 7 |
| <i>Who's Who : parties prenantes</i> | 7 |
| <i>Analyse de l'existant: état de l'art</i> | 8 |
| <i>Cahier des charges</i> | 9 |
| Livrables | 9 |
| Budget | 10 |
| Conception et analyse | 10 |
| Changement d'architecture | 11 |
| <i>BlinkM vs WS2812b</i> | 11 |
| Solution BlinkM et cartes filles | 11 |
| Solution WS2812b et carte linux mère | 14 |
| <i>Performances théoriques</i> | 17 |
| <i>Conséquences du changement</i> | 18 |
| Architecture hardware | 20 |
| <i>Schéma fonctionnel de principe</i> | 20 |
| Carte Linux embarquée | 21 |
| Génération de signaux de pilotage | 23 |
| Architecture logicielle | 24 |
| <i>Schéma fonctionnel de principe</i> | 25 |
| Service Web | 26 |
| Front end | 26 |
| Back end | 26 |
| MiddleWare | 28 |
| Réalisations | 29 |
| Construction du mur de LEDs | 29 |
| <i>Production des LEDs WS2812b</i> | 29 |
| Chaîne de production | 29 |
| Banc de test | 32 |
| <i>Modification de la structure en bois</i> | 33 |
| <i>Câblage et alimentation</i> | 35 |
| Principe de câblage d'une LED | 35 |
| Schéma de chaînage | 36 |
| Schéma d'alimentation | 36 |
| <i>Tests in situ</i> | 37 |
| Mesures de continuités | 37 |
| Mesures de tensions | 37 |
| Vérification du chaînage | 38 |

| | |
|--|-----------|
| <i>Validation des couleurs et du pilotage</i> | 38 |
| Préparation d'une carte de test | 38 |
| Lien entre position (X,Y) et position dans la chaîne | 38 |
| Codage du programme d'animation (Arduino) | 39 |
| Système de contrôle et pilotage | 40 |
| Codage du site Web | 40 |
| Base de données NoSQL | 40 |
| Back-end NodeJS | 43 |
| Front-end Angular 7 et NGINX | 44 |
| Communication avec le MiddleWare | 49 |
| Version Framebuffer | 49 |
| Codage d'un simulateur de LED Wall | 50 |
| Version 1: P.O.C. architecture embarquée sur Raspberry PI 3B+ | 52 |
| Mise en place du système Raspbian | 52 |
| Configuration du système Linux | 52 |
| Configuration réseau | 53 |
| Configuration du projet | 54 |
| Conclusion POC | 54 |
| Version 2: Intégration système embarquée sur BeagleBone Black | 54 |
| Mise en place du système Debian | 54 |
| Configuration du système Linux | 54 |
| Installation sur la emmc | 54 |
| Test de la carte | 54 |
| Préparation du système : | 55 |
| Programmation des Programmable Real-time Units | 61 |
| Utilisation des PRU | 61 |
| Communication inter-processeurs | 62 |
| Installation et démarrage automatique du PRU | 65 |
| Génération de signaux WS2812b | 65 |
| Codage du MiddleWare | 67 |
| Implémentation du framebuffer fichier | 67 |
| Événement inotify | 67 |
| Structure du fichier | 67 |
| Adaptation bas niveau pour la génération de signaux | 67 |
| Librairie pour WS281x sur Raspberry PI | 67 |
| Dialogue inter-processeurs sur BeagleBone Black | 68 |
| Gestion de projet | 69 |
| Kanban et Trello | 69 |
| Répartition des tâches | 69 |
| Équipe de montage matériel | 69 |
| Équipe de développement Web | 70 |

| | |
|---|-----------|
| <i>Équipe intégration système</i> | 70 |
| <i>Équipe développement embarqué bas niveau</i> | 71 |
| Communication en équipe | 71 |
| Gdrive et Github | 71 |
| <i>GitFlow</i> | 71 |
| <i>Intégration continue</i> | 73 |
| Conclusion | 74 |
| Perspectives d'améliorations | 74 |
| Annexes | 75 |

Remerciements

Avant toute chose, nous souhaitons adresser nos remerciements à M.ESSWEIN et M.ROLLAND qui ont été nos responsables de projet. Ils ont su faire preuve d'ouverture d'esprit et ont été d'excellents clients rendant la progression sur ce projet agréable et valorisante.

Nous adressons également nos remerciements à S.BEAUFILS et G.MAILLAUD pour leur soutiens logistique et matériel.

Enfin, nous remercions l'ensemble de l'équipe pédagogique de Polytech Tours pour nous avoir permis de travailler sur ce projet.

Introduction

Depuis 2014, le département informatique de Polytech cherche à s'équiper d'un mur de LED installé au niveau de la cloison en verre de la bibliothèque. Ce projet, co-dirigé par Carl ESSWEIN et Alexis ROLLAND, a pour but promouvoir la formation lors des journées portes ouvertes.



Figure : Etat du LED Wall à la reprise (Septembre 2018)

Cette année 2018-2019 est l'occasion pour notre équipe de 6 apprentis de d'achever la réalisation du mur de LED dans le cadre du projet collectif de quatrième année.

Lorsque ce formidable outils de communication sera achevé, l'école pourra faire valoir les nombreuses compétences techniques et organisationnelles enseignées dans ses formations.

Il est important de montrer aux étudiants que la philosophie d'enseignement par projet, adoptée par le réseau Polytech, est un excellent moyen d'aider les étudiants à s'épanouir à travers des projets innovants. Promouvoir un projet vitrine tel que le LED Wall aux futurs étudiants sera un excellent moyen de le prouver.

Présentation & Contexte du projet

Pour démarrer ce chapitre de présentation, nous allons présenter le projet dans son ensemble. Nous commencerons par présenter l'expression des besoins énoncés par les tuteurs du projet. Nous continuerons cette description en présentant l'équipe de projet. Nous finirons par une analyse de l'existant et un point de cahier des charges.

Expression des besoins

En abordant ce projet, nous avons cherché à comprendre les motivations initiales de M.Rolland et M.Esswein sur ce projet en faisant l'effort de se détacher du travail effectué jusqu'ici.

Les attentes sur ce projet sont multiples. Les besoins énoncés par M.Rolland et M.Esswein sont les suivants:

1. Un mur de LEDs RGB intégralement câblé, testé et installé dans la bibliothèque du département informatique.
2. Le système de pilotage doit être capable de rafraîchir l'ensemble des LEDs jusqu'à une fréquence de 24Hz.
3. Le système doit disposer d'un logiciel de création de motifs à afficher sur le mur, permettant de créer des images, des animations et des programmes lumineux.
4. Le système de gestion du stockage et contrôle des programmes lumineux à afficher sur le LED Wall doit être fonctionnel.
5. L'ensemble des logiciels de gestion du mur de LED doivent être simple de façon à être utilisé par S.BELAIR et G.MAILLAUD.

Who's Who : parties prenantes

Ce projet a été repris cette année 2018-2019 par notre équipe de 6 apprentis du département informatique industrielle:

- LOCHE Jérémie
- THOMAS Louis
- RICHARD Arthur
- SIMMOU Soufiane
- LE NEL Corentin
- DOROTHÉE Corentin

En tant qu'équipe de travail sur ce projet, nous avons la mission de réaliser le montage, le câblage, les tests et le développement du logiciel de contrôle du mur de LEDs.

Nous sommes encadrés par le binômes de tuteurs:

- ESSWEIN Carl : encadrant, client et enseignant au département informatique
- ROLLAND Alexis : encadrant, client et référent technique en électronique

Enfin nous avons les membres du service logistique et gestion de Polytech Tours:

- BEAUFILS Sébastien : chargé des commandes pour l'approvisionnement matériel.
- MAILLAUD Gérald : technicien du département informatique et soutien logistique sur ce projet.

Analyse de l'existant: état de l'art

Depuis les 5 ans d'existence du projet, il a été repris par plusieurs équipes ayant chacunes apportés leurs lots d'avancements et améliorations.

En **2014** a commencé le projet LED Wall. Initialement, le mur était constitué de 20 colonnes et 10 lignes offrant un recouvrement important de la surface disponible au niveau de la bibliothèque du département informatique. Le fonctionnement de ce mur réside dans la technologie utilisée pour les LEDs. En effet, les équipes ayant travaillé sur le sujet ont proposé de réaliser un pixel du mur à l'aide de modules BlinkM. Ces modules, composés d'un micro-contrôleur et d'une LED RGB, sont adressable sur un bus I2C permettant piloter les couleurs. Avec l'aide de G.MAILLAUD, la première équipe de projet a réalisé une structure en bois composé de cubes permettant d'accueillir les modules BlinkM. Cependant, les étudiants ont essayé de piloter l'ensemble du mur à l'aide d'un seul bus I2C, ce qui n'est pas du tout faisable avec cette technologie.

En **2015**, l'étude d'une nouvelle architecture du LED Wall a été réalisée de façon à piloter le mur par tronçons. En effet, l'idée développée est de réaliser un ensemble de 10 cartes filles chargés de piloter un tronçon vertical de 20 leds commandés par I2C sur le mur.

Durant la même année, un logiciel de pilotage a été analysé mais malheureusement, les deux éléments matériels et logiciels ne se sont pas rencontré ce qui n'a pas permis d'aboutir à un ensemble fonctionnel.

Le projet a alors été poursuivi en **2016 - 2017**. L'équipe d'étudiants a alors repris l'architecture spécifiée en 2015 et a mis en place 2 colonnes de LEDs sur le mur (18 restantes). Malheureusement, à ce stade, nous n'avons pas noté de travail suffisant du côté matériel et logiciel permettant d'attester d'une avancée significative.

Ensuite, un stagiaire en BTS a réalisé une procédure de fabrication et tests des modules BlinkM.

Le mur n'était toujours pas terminé. C'est pour cette raison que le projet a été proposé de nouveau pour l'année **2017-2018**. L'équipe a fourni un travail

conséquent qui a permis de valider le fonctionnement de 2 tronçons et de terminer le câblage de 5 autres tronçons (non validés). Cette équipe a réussi à fournir des fiches de tests matériels et logiciels complètes. Ils ont mis en service deux cartes filles validés sur le point logiciel et matériel (bien qu'il subsiste encore quelques points brumeux sur leur fonctionnement, notamment au niveau de la mémoire de stockage EEPROM).

Pour le pilotage du mur de LED, cette équipe a réalisé une logiciel très élémentaire permettant de convertir une image bitmap en un ensemble de trames. Ces trames sont nécessaires au chargement des images en mémoire dans les cartes filles.

Suite à cette analyse, nous nous rendons compte que de 2015 à 2018, le projet n'a pas abouti et n'illumine toujours pas le hall du département informatique.

Pour cette raison, M.Rolland et M.Esswein, propose la **5ème année édition** du projet LED Wall pour mettre un terme à la réalisation de celui-ci. Ils ont donc proposé de tester le câblage réalisé l'année précédente puis de câbler et tester le reste des colonnes. Ensuite il aurait fallu fabriquer et tester de nouvelles cartes filles à partir des schémas, routages et fiches de tests préparés par les équipes précédentes. Enfin, il serait nécessaire de focaliser des ressources sur la création d'un logiciel de pilotage du mur.

Cahier des charges

En abordant ce projet, il ne nous a pas été formellement fourni de cahier des charges. Nos clients attendent de nous que nous terminions le LED Wall tel qu'il a été commencé 5 ans auparavant. Dans ce cas de figure, les éléments de cahier des charges auraient dut être les mêmes que pour les équipes précédentes.

Au final, nous avons convenu avec M.Esswein et M.Rolland de fournir une solution répondant à l'expression de leurs besoins décrite précédemment.

En effet, nous avons préféré nous concentrer sur les attentes fonctionnelles plus sur les contraintes techniques imposées par la poursuite de ce projet.

Nous disons cela car vous découvrirez dans la prochaine partie de ce rapport que la stratégie technique adoptée a changée radicalement pour cette année.

Livrables

Les livrables attendus sont liés aux attentes fonctionnelles du LED Wall mais aussi documentaires permettant de promouvoir le projet.

1. **Un mur de LEDs RGB** intégralement câblé, testé et installé dans la bibliothèque du département informatique capable d'afficher des animations d'une fréquence allant jusqu'à 24 Hz.

2. **Un logiciel de création** de motifs à afficher sur le mur, permettant de créer des images, des animations et des programmes lumineux et gérer l'affichage sur le mur.
3. **Deux posters** de promotions :
 - a. Un poster vulgarisé tout public format A3 à présenter aux visiteurs des journées portes ouvertes.
 - b. Un poster technique présentant l'architecture matérielle et logicielle, les technologies pour les informaticiens et électroniciens.
4. **Une ou deux vidéos** de **démonstration** bien finies à diffuser sur les réseaux sociaux.

Budget

Nous n'avons pas trouvé de chiffre réel sur l'investissement économique réalisé sur ce projet dans le rapport de l'équipe précédente. Cependant il a été convenu que le budget associé à la finalisation du LED Wall serait en conséquence des moyens à mettre en oeuvre pour y arriver.

En voyant le travail achevé par le passé et la quantité de matériel mis en oeuvre, **on estime** que l'école a **déjà dépensé entre 2000 et 4000 € de matériel** dans ce projet.

En comptant le **coût humain** d'environ 15 à 20 apprentis ayant travaillé sur le projet par le passé (70 h par apprentis à 6.5€/h), on estime que l'école aurait dépensé **l'équivalent de 8000 € de main d'oeuvre** pendant les **4 années précédentes**.

Pour cette année 2018-2019, on parle d'une enveloppe pouvant monter **jusqu'à 1000€ de matériel supplémentaire**. L'objectif étant d'arriver au bout de ce projet, nous sommes **6 apprentis** sur cette tâche. À raison d'un travail estimé de 70h par apprentis pour 6.5€/h, **le coût humain estimé** de ce projet s'élèverait à environ **2800 €**.

Le **budget global** de ce projet s'élève donc à **3800€**.

Conception et analyse

Nous entamons ce chapitre d'analyse et de conception par un point critique sur le travail réalisé jusqu'à maintenant. Lorsque nous avons repris le projet du LED Wall, nous avons dès la première semaine fait part de nos **inquiétudes** à M.Esswein et M.Rolland quant aux **chances de réussite du projet**. Nous n'étions pas convaincu par les choix réalisés par le passé. Nous avions en tête une architecture, plus simple et à priori plus performante *sur le papier*, basée sur des LEDs adressables WS2812b conçues pour ce genre d'application.

En présentant cette idée à M.Rolland et M.Esswein, ils n'ont au premier abord pas été séduit. En effet, proposer une nouvelle architecture revient à

remettre en question tout le travail réalisé jusque là par les équipes précédentes. Mais, après avoir énoncé quelques points forts de notre proposition, les encadrants se sont montrés ouvert à la discussion et nous ont demandés de réaliser un comparatif honnête des deux architectures.

Nous avons alors réalisé une étude de l'existant en cherchant de ses points forts et points faibles à la manière d'une analyse SWOT (forces faiblesses opportunités et menaces). Nous avons fait de même pour la solution des LEDs WS2812b.

Changement d'architecture

BlinkM vs WS2812b

Solution BlinkM et cartes filles

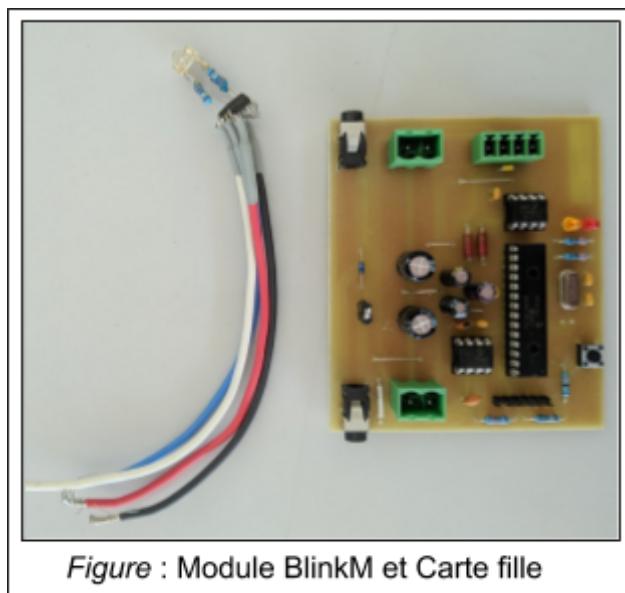


Figure : Module BlinkM et Carte fille

La figure ci-dessus présente un module BlinkM, brique élémentaire du système proposé par les étudiants précédents. A sa droite, vous trouverez une carte fille dédiée au pilotage des BlinkM grâce au bus I2C. Avec ce système, les cartes filles ont vocations à être sur un bus CAN. Chaque carte fille est équipée d'une intelligence basé sur un microcontrôleur PIC associé à un dispositif de stockage de type EEPROM.

Nous avons cherché à éclaircir les qualités et défauts de cette solution sur les éléments de ce système.

L'architecture hardware:

On commence par énoncer ce qui nous paraissent être les points forts et faiblesses d'un BlinkM. Pour rappel, le BlinkM est constitué de 3 résistances, 1 LED

RGB, un ATTiny45 ainsi que 4 fils de connexions. Il ne faut pas moins de 12 soudures pour fabriquer un simple module.

| Module BlinkM | |
|---|--|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> Couleur 24 bits sur 3 canaux PWM Bonne luminosité Connectique sur 4 fils Adressable unitairement sur le bus | <ul style="list-style-type: none"> Homogénéité des couleurs → Utilisation de résistances de limitation de courant Bus I2C : Haute fréquence difficile sur la distance 15 min de soudage + 4 min de programmation Soudure Fragiles 12 soudures pour 1 LED Gaine thermorétractable difficile à installer pour éviter les courts circuits. |

Comme on l'a vu, fabriquer 200 BlinkM prendrait à minima 7h à temps plein et ce sans compter les pertes qui peuvent être occasionnées par la fragilité des modules. Nous avons déploré plus de 30 BlinkM inutilisables pour des raisons de pâtes de l'ATTiny45 cassés ou des soudures (souvent des soudures 3 points) de mauvaise qualité. **Les BlinkM sont donc difficiles à fabriquer et fragiles.** Cependant, ils sont équipés de fonctionnalités intéressantes comme les couleurs 24 bits et la possibilité de les adresser unitairement sur le bus I2C.

On continue cette analyse avec l'architecture des cartes filles.

| Cartes filles | |
|---|--|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> Bus CAN: Sécurité et rapidité Bonne capacité de stockage Analysés, 2 existantes, BOM , Schéma et routage terminés | <ul style="list-style-type: none"> Bus I2C limité à 128 LEDs et en distance du fait de la fréquence, juste suffisant pour un tronçon Pas de synchronisation des cartes filles sans Maître CAN → perte de synchronisation à l'échelle d'une journée Difficile d'estimer le temps de fabrication et de test d'une carte fille : probablement 2h à 3h par cartes Prix de fabrication → Fabrication PCB, Composants... |

Nous avons trouvé que le travail réalisé sur les cartes filles est cependant très bien et qu'il y a au final peu de choses à redire de ce côté là. Cependant, l'architecture proposée et présentée par les équipes précédentes faisaient mention d'un PC de pilotage utile au chargement des motifs dans les cartes et à la synchronisation des affichages. Malheureusement, M.Rolland et M.Esswein n'ont pas validé cette solution et auraient préféré disposer d'une carte mère dédiée au pilotage. Cette carte est inexistante et n'a pas été mentionnée par les équipes précédentes. Les cartes filles seraient alors vouées à perdre la synchronisation au cours d'une journée. Ceci amène alors un décalage entre les tronçons du mur.

Nous avons énoncé nos points d'inquiétude en ce qui concerne l'architecture matérielle. Cependant, nous n'avons pas encore abordé le coté logiciel qui de l'architecture.

L'architecture logicielle

Nous avons souhaité poursuivre plus loin que la simple observation du système présenté par les autres étudiants en analysant la façon de programmer les LEDs, les cartes filles et le protocole de communication.

On commence par parler de la programmation d'un module BlinkM.
Les modules sont uniques et disposent d'une adresse I2C configurable.

| Programmation d'un module BlinkM | |
|--|---|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> Procédure fabrication et de test spécifiée et complète Protocole de communication I2C clair et spécifié | <ul style="list-style-type: none"> 1 logiciel spécialisé et 1 programmeur pour charger le binaire BlinkM 1 programme Arduino + 1 breadboard pour configurer l'adresse I2C 1 carte de test sur laquelle planter les BlinkM pour le test Remplacement de BlinkM difficile |

En terme de firmware du BlinkM, les équipes ont fait du bon travail. Cependant, ils n'ont pas facilité la vie des programmeurs et constructeurs en créant 3 programmeurs/testeurs pour préparer un module BlinkM. Ceci augmente considérablement le temps de fabrication d'un BlinkM. On augmente aussi énormément les chances de casser le BlinkM au moment de le planter dans le programmeur et le testeur. Ceci rend les modules BlinkM difficiles à remplacer en cas de panne. Ces modules sont déjà fragiles et la procédure de programmation

n'arrange pas les choses. Un point fort est que ces modules ont une adresse unique et la procédure de validation est claire et spécifiée.

Nous en venons maintenant aux cartes filles.

| Programmation d'une carte fille | |
|--|--|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> Firmware intègre la couche CAN → Bon travail de la part des équipes précédentes Protocole de communication clair et bien défini Dimensionnée au problème | <ul style="list-style-type: none"> “Adresse” CAN configurée par programmation du firmware → Remplacement difficile Maximum 15 cartes filles sur le bus → A cause de l’implémentation du CAN sur la cible → Maximum 300 LEDs adressables Surcharge protocolaire CAN 50-60% → Mauvais pour le débit de données utiles Écriture en EEPROM d'une image = 200ms Maximum 2000 images en EEPROM Impossible de faire de l'affichage dynamique sauf pixel par pixel → Limite de 24 images par secondes non respectée |

Nous avons remarqué que encore une fois, les équipes précédentes ont fourni un excellent travail de documentation sur leur système. Cependant, il apporte son lot de limitation qui pour nous se rapporte à de sévères problèmes. Un exemple concret est le chargement des animations en EEPROM. Si l'on prend le chargement de 2000 images créant une animation de 83 secondes (pour 24 img/s) à afficher sur le mur de LEDs, il ne faudra pas moins de 6 minutes 30 pour charger les images dans les cartes filles. Le temps de chargement étant plus long que le temps de l'animation, on ne peut pas effectuer d'affichage dynamique avec cette méthode.

Après avoir vu les points d'attention sur la technologie des BlinkM et leurs cartes filles, nous allons présenter notre nouvelle solution.

Solution WS2812b et carte linux mère

Lorsque nous avons présenté cette nouvelle architecture, nous avons pris soin de lister les points fort et points faibles.

Nous avons alors proposé une solution basée sur des LEDs adressables du commerce et un dispositif de commande basé sur une seule carte mère. Vous pouvez voir ci-dessous les LEDs et la carte mère Linux.

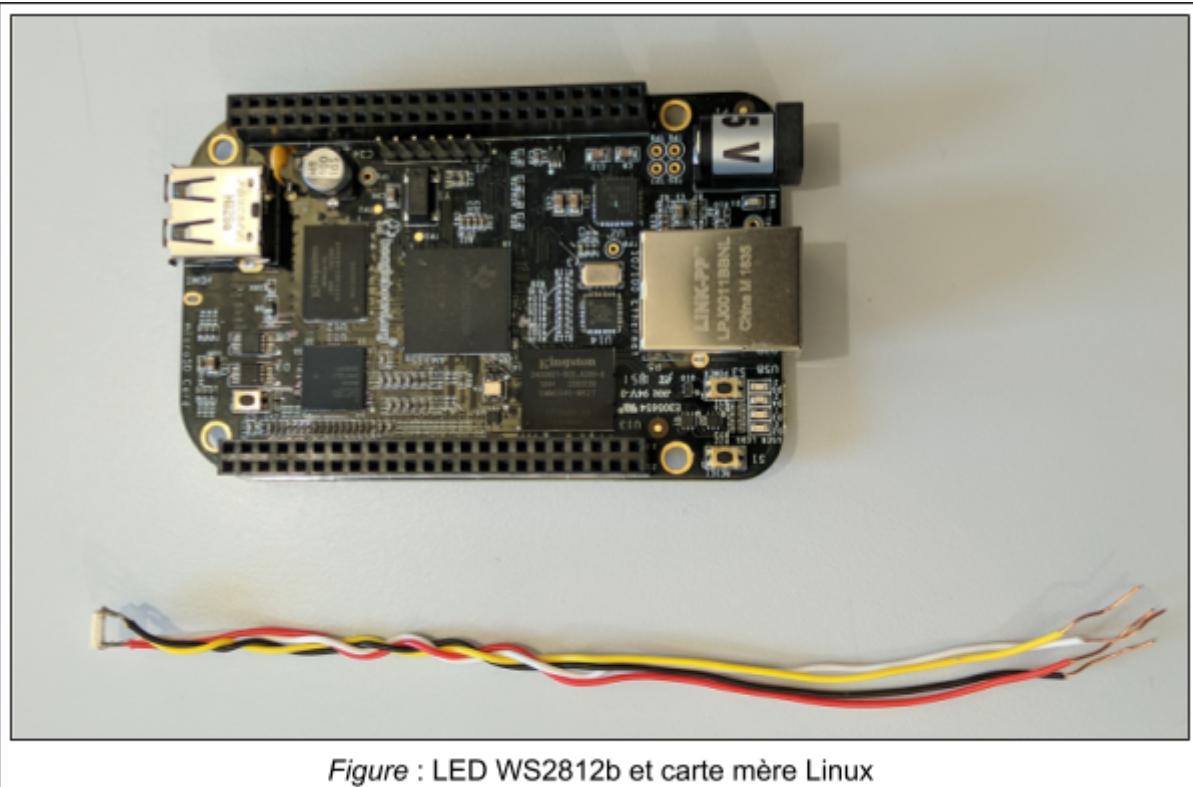


Figure : LED WS2812b et carte mère Linux

Nous avons proposé une système basé sur les LEDs WS2812b adressables. Ces LEDs disponibles au format 5050 sont alimentable en **5V** de la même manière que les modules BlinkM. Consommant moins qu'un module BlinkM, il n'est **pas nécessaire de changer l'alimentation dimensionnée par les équipes précédentes**. Pour fonctionner, ces LEDs ont besoin d'être chaînées les unes aux autres, créant ainsi une ligne de donnée parcourant les LEDs du début de la chaîne jusqu'à la dernière LED.



Figure : LED RGB WS2812b

Nous avons récapitulé dans un tableau l'ensemble des points positifs et négatifs à propos de ces LEDs.

| LED WS2812b | |
|--|---|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> 4 min de fabrication / soudage + 1 min de test Adaptable sur le LED Wall en | <ul style="list-style-type: none"> Luminosité légèrement plus faible qu'un BlinkM → Réduction de la |

| | |
|---|---|
| <p>10-15 minutes par tronçons.</p> <ul style="list-style-type: none"> • Adressable et programmée en usine • Couleur 24 bits • Chaînage → Câblage allégé à 3 fils VCC-GND-DATA • Reshaping de signal entre les LEDs d'une chaîne → Pas de dégradation de signal et 10 m de portée entre 2 LEDs • Couleurs calibrés en usine et pilotage des LEDs en courant → Homogénéité des couleurs. • Pas d'adresse fixe → position dans la chaîne définit l'adresse | <p>consommation.</p> <ul style="list-style-type: none"> • Chaînage: Une LED est en panne, les LEDs suivantes ne reçoivent plus le signal de donnée → Cependant, facilite le diagnostic, le signal s'arrête à la première LED HS • Rafraîchir 1 LED implique de rafraîchir l'ensemble de la chaîne |
|---|---|

Vous l'aurez compris, ces LEDs WS2812b peuvent remplacer avantageusement les BlinkM. Elles n'ont pas une connectique plus compliquée et elles possèdent, entre autres, des propriétés intéressantes d'homogénéité et de consommation garanties par la production industrielle en grande série. Les diagnostics et les remplacements en cas de pannes sont facilités par le fait que les LEDs n'ont pas d'adresse fixe. Ceci était un point faible pour les BlinkM, et non des moindres.

Ne pas avoir d'adresse fixe permet de produire les LEDs très rapidement car il n'y a aucune programmation à prévoir et le câblage se résume à effectuer 4 soudures simples sur le composant 5050. On peut ainsi, produire l'intégralité des 200 LEDs WS2812b comparé à 60 modules BlinkM pour la même durée.

On pourrait penser que devoir rafraîchir l'ensemble des LEDs du mur est bien cher payé pour mettre à jour une seule LED. Cependant, ce propos est à nuancer lorsque l'on regarde de plus près la fréquence de rafraîchissement atteignable en fonction du nombre de LEDs.

Un autre intérêt de ces LEDs est qu'elles peuvent être pilotées à partir d'une simple carte mère. On oublie donc les cartes filles avec cette solution.

Nous proposons alors d'utiliser une simple carte fille Linux possédant un cœur temps réel pour la gestion des signaux du mur de LEDs.

| Carte mère Linux | |
|---|--|
| Forces | Faiblesses |
| <ul style="list-style-type: none"> • Cœur temps réel : pour la gestion des signaux WS2812b • Héberge un site Web de pilotage et de création de motifs pour le mur | <ul style="list-style-type: none"> • Plus de modularité en tronçons. |

- **Faible consommation** < 10 cartes filles
- 1 seule carte = 1 seul problème
- **6 ms pour rafraîchir le mur entier** = 1.25 us/bit WS2812 → **165 img/s** meilleure fréquence atteignable en théorie.
- 1 GPIO pour le bus de 200 LEDs → Câblage extrêmement simple
- **Moins cher** que de produire des cartes filles

Cette nouvelle solution à base de carte Linux ouvre la porte à de très intéressantes possibilités. En effet, en utilisant une carte unique pour l'intégralité du mur de LEDs, on réduit énormément la surcharge protocolaire et donc on augmente nos chances de pouvoir réaliser du pilotage dynamique. De même, utiliser une carte Linux dédiée permet d'héberger un site Web de pilotage et ainsi fournir à l'école un système minimaliste clé en main pour utiliser le LED Wall. Cette architecture est, dans son ensemble, plus facile à maintenir et à déployer que la solution BlinkM.

A la suite de la présentation de cette nouvelle architecture, M.Rolland, sceptique de la performance du système basé des WS2812b, nous a mis au défi de faire fonctionner le mur. Défi que nous avons bien évidemment relevé.

Cette solution, bien que risquée, puisqu'elle n'est pas modulaire, a donné l'impression à nos tuteurs que nous proposons une architecture qui passe ou qui casse. Malgré les preuves en vidéos de projets réalisés à partir des LEDs WS2812b et les performances théoriques garanties par les constructeurs, il a fallu faire preuve de ténacité pour montrer à nos tuteurs que cette technologie est robuste, réalisable et qu'elle répond à l'expression des besoins.

Parlons un peu de performances et des évolutions apportées par cette nouvelle architecture.

Performances théoriques

On a vu que ces LEDs doivent être chaînées pour fonctionner. Le constructeur garantie qu'une chaîne de 1024 LEDs peut être mise à jour à une fréquence de 30 Hz. Pour illustrer la variation de la fréquence de rafraîchissement en fonction du nombre de LEDs, nous avons tracé le graphique suivant. En partant des données constructeurs, on sait qu'un bit prend 1.25 us sur la liaison de données de la chaîne. Etant donné que 2 rafraîchissements sont séparés par une période de **reset** de 50 us, on obtient l'équation suivante pour le temps de rafraîchissement de N leds:

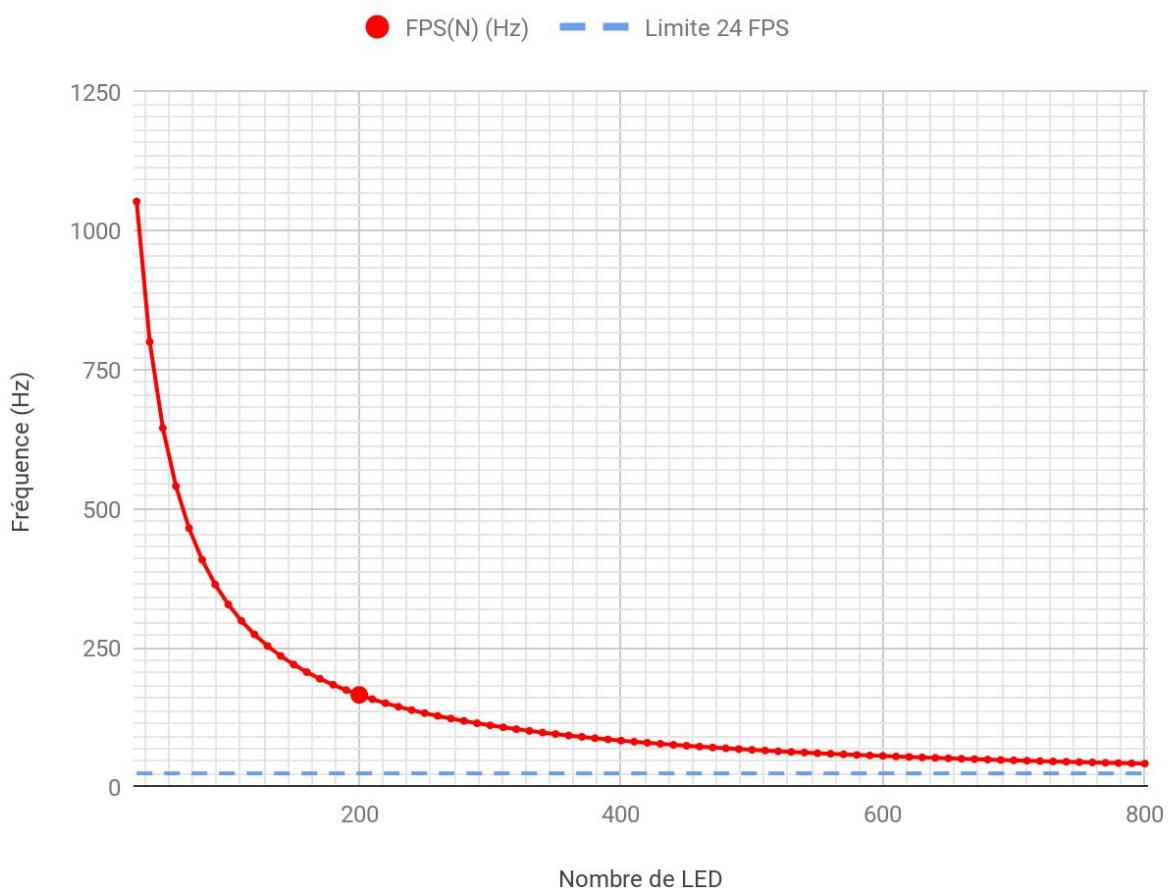
$$T_{refresh}(N) = N \times 24 \text{ bits} \times 1.25 \mu\text{s/bit} + 50 \mu\text{s} = 30\mu\text{s/led} \times N + 50\mu\text{s}$$

La fréquence de rafraîchissement théorique maximale est donc de:

$$F_{refresh}(N) = \frac{1}{T_{refresh}(N)}$$

On obtient alors le graphique suivant:

Fréquence de rafraîchissement (Hz) en fonction du nombre de LED



On voit que jusqu'à 800 LEDs sur une seule ligne de données, on atteint une fréquence de 41 Hz, toujours supérieure à la limite de 24 images par seconde imposée.

Conséquences du changement

Le changement d'architecture implique au final très peu de modifications sur la structure existante. En effet, la structure en bois permettant de supporter les modules BlinkM pourra être utilisée telle quelle.

De plus, nous pouvons réutiliser l'ensemble des câbles d'interconnexions confectionnés pour connecter les modules BlinkM, donc il n'y a aucune dépense

supplémentaire à ce niveau. À cela, nous ajoutons la réutilisation de l'alimentation 90W utilisée par les équipes précédentes, qui n'a pas besoin d'être changée. Les économies d'énergie réalisées sur cette partie nous permettent même d'alimenter la carte Linux depuis cette alimentation. On simplifie le câblage et on diminue alors les coûts et la main d'oeuvre.

Il y a cependant un point négatif à tout cela. Nous n'utilisons plus les modules BlinkM pour le mur de LED ainsi que les cartes filles. Cela implique un effort sur leur revalorisation. Nous pensons notamment à une utilisation en TP d'électronique et microcontrôleurs pour les cours des DII3.

Du point de vue budgétaire, le changement fait **tomber la facture de 700€ pour terminer la solution BlinkM à 350€ pour réaliser la solution WS2812b** intégralement (sans considérer la récupération des fils et de l'alimentation).

| Solution WS2812b | |
|--|--------------------|
| Matériel utilisé | Prix Unitaire HT € |
| 200 m de jarretière téléphonique jaune/noir | 32,15 |
| 200 m de jarretière téléphonique rouge/blanc | 30,05 |
| 200 LEDs SK6812/WS2812b 4.58/10pcs | 91,60 |
| Fils interconnection Red, Blue, Black * 20cm * 200 | 75,00 |
| Alimentation 5V 18A 90W | 16,43 |
| Carte Linux | 100,00 |
| Total | 345,23 |

Changer de stratégie sur ce projet nous permet de simplifier l'architecture et de dégager plus de temps pour le développement d'outils de gestion du mur de LED.

En effet, grâce aux atouts de cette nouvelle structure, nous pouvons envisager un système de pilotage plus complet que ce que nous aurions pu faire avec les BlinkM. Le matériel pouvant être opérationnel plus rapidement, ceci nous permet de placer des ressources sur le développement.

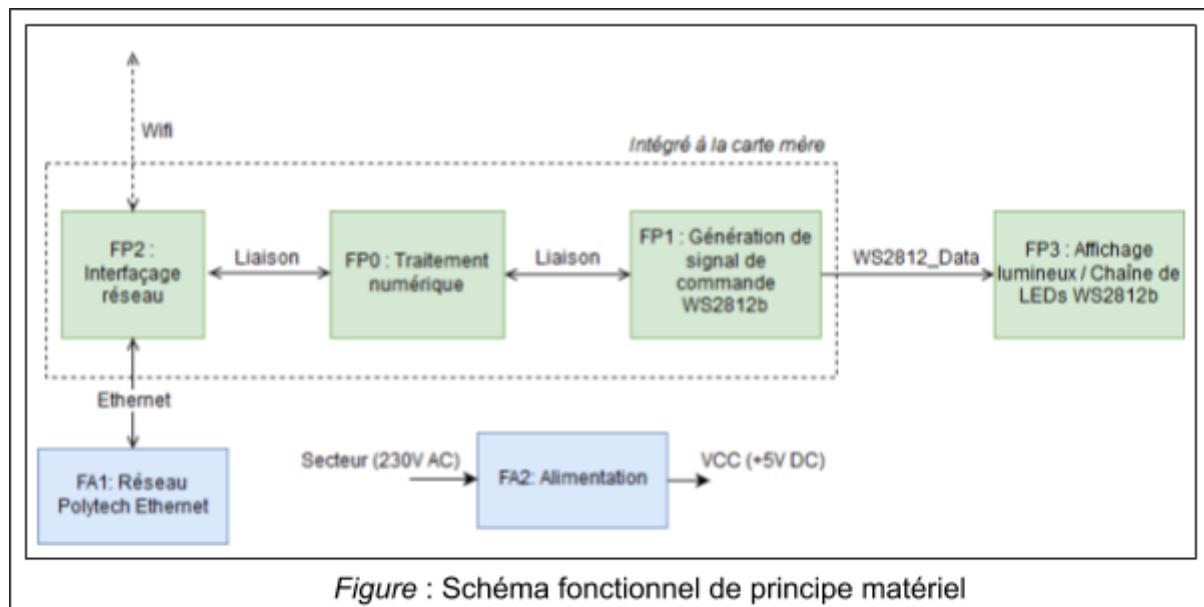
Fort heureusement, nous avons réussi à valider l'ensemble des points de cette nouvelle architecture avec nos encadrants et elle est devenu de facto la nouvelle architecture de base pour ce système.

Nous allons désormais rentrer dans les détails de cette architecture. Nous verrons comment nous l'avons conçue puis nous parlerons des signaux mis en place pour le pilotage.

Architecture hardware

Pour vous présenter l'architecture du système, nous passerons par une revue du schéma fonctionnel de principe. Nous vous parlerons ensuite de l'architecture nécessaire au niveau de la carte Linux. Enfin nous aborderons un point technique de pilotage des LEDs WS2812b.

Schéma fonctionnel de principe



L'architecture épurée contient un ensemble de 4 fonctions principales ainsi que 2 fonctions annexes.

Nous avons besoin d'une fonction **FP0** de traitement numérique capable d'héberger un site WEB et générer des motifs à envoyer sur le mur de LEDs.

Nous avons aussi besoin d'une fonction **FP1** de génération de signaux de commandes pour les WS2812b. Cette fonction doit respecter les timings préconisés par la documentation des LEDs WS2812b et devra communiquer avec la fonction de traitement numérique chargée de générer les motifs.

Ensuite, nous avons besoin de la fonction **FP3** d'affichage lumineux qui correspond aux LEDs à installer dans le mur.

Enfin, nous terminons par la fonction **FP2** d'interface réseau qui nous permet de communiquer avec l'extérieur à travers le réseau Polytech ou bien depuis un point d'accès sans fil.

La fonction annexe **FA2** d'alimentation est réalisée par le bloc d'alimentation 5V fourni. Il délivrera une puissance supérieure ou égale à 62.5W permettant

d'alimenter 200 LED consommant 60mA chacune ainsi qu'une carte Linux consommant jusqu'à 500mA à la tension de 5V.

$$P_{LEDWall} = (200 \times 0.06A + 0.5A) \times 5V = 12.5A \times 5V = 62.5W$$

Enfin, la fonction annexe **FA1** d'accès réseau est réalisée par l'infrastructure réseau du bâtiment Portalis de Polytech Tours.

On remarque que si l'on choisit bien la carte Linux embarquée, on peut faire tenir les fonctions **FP0**, **FP1** et **FP2** sur la même carte électronique.

Nous allons notamment revenir un peu plus en détail sur ce point. Ensuite nous passerons en revue les caractéristiques des fonction **FP1** et **FP3** en détaillant le principe de contrôle des LEDs WS2812b et le chaînage des LEDs sur le mur.

Carte Linux embarquée

Nous avons vu précédemment que pour choisir de la manière la plus efficace notre cible Linux embarquée, il faut s'assurer que celle-ci puisse répondre aux besoins des fonctions FP0, FP1 et FP2. Il nous faut donc au choix une interface réseau Ethernet ou Wi-Fi, un processeur capable de faire tourner un serveur Web et un dispositif de génération de signaux pour les WS2812b.

En ce qui concerne les aspects d'hébergement du site web et l'interface réseau, la plupart des cartes Linux embarquées possèdent le processeur, la RAM et le support de stockage disponible pour faire fonctionner un serveur. Nous avons cependant fait attention au support de la carte Linux à choisir et notamment la *disponibilité ou non d'une distribution suffisamment standard et complète* pour nous aider dans le développement.

Bien que nous ayons appris à compiler une distribution Yocto, le temps nécessaire au développement et la configuration du système nous ont paru être un projet de trop grande envergure pour réussir à développer un système complet et fonctionnel, dans le temps imparti.

Nous avons donc orienté nos recherches sur une carte disposant d'une distribution bien supportée telle que **Arch Linux ou Debian**.

Un second point important est la partie génération de signaux pour le mur de LED. Cette partie ayant une forte contrainte temps-réel, elle ne peut pas être réalisée par le processeur dédiée à faire tourner l'OS Linux. L'idéal est de disposer ici d'un système fortement prédictible et ne dépendant pas d'un OS (Bare metal).

On pourrait déporter cette tâche sur un microcontrôleur externe à la carte Linux et dialoguer en haute vitesse à travers une liaison série synchrone de type SPI ou bien une liaison moins rapide asynchrone.

Cette solution aurait l'avantage de fonctionner sur n'importe quelle carte Linux disposant d'un périphérique SPI par exemple.

Cependant, nous avons appris pendant nos cours de systèmes embarqués que des cartes Linux embarquées possédant des processeurs à **architecture hétérogènes** dédiés à ces cas d'utilisations existent.

Nous aurions pu choisir pour cela la carte **Udoo Neo** qui possède une **coeur A9** pour la partie Linux et un **coeur M4** pour la partie temps réel.

Bien que cette carte comporte sur le papier des caractéristiques intéressantes, les seules distributions fournies par le site Udoo sont basées sur une version d'**Ubuntu 14.04 datant de 2014** ou bien sur Android 6.0.1. De plus il s'agit ici d'une distribution modifiée par Udoo, donc nous sommes dépendants du bon vouloir des ingénieurs pour disposer de paquets à jour.

Nous nous sommes alors penché du côté d'autres cartes industrielles bien mieux supportées.

Nous avons trouvé l'ensemble des cartes BeagleBoard et nous nous sommes notamment attardé sur la carte **BeagleBone Black**. Cette carte a plusieurs atouts. Elle dispose d'un processeur hétérogène **TI-Sitara AM3358** disposant de **deux coeurs temps réels** indépendant (PRU: Programmable Real-time Unit) ainsi qu'un cœur **ARM Cortex A8 pour l'OS Linux**. Elle dispose de 512 Mo de RAM et de 4Go de mémoire **flash eMMC**.

Cette carte s'inscrit clairement au sein des cartes industrielles avec des fonctionnalités très pertinentes pour notre utilisation.

Une mémoire **flash eMMC** nous permet d'éviter les problèmes de corruption très fréquents sur les cartes micro-SD utilisés sur les Raspberry PI et la carte Udoo.

De plus, cette carte dispose d'une distribution **Debian** avec un **Kernel mainline à jour**. On peut donc aisément flasher la mémoire eMMC avec une distribution **Debian Stretch IOT 9.8** sortie en **Avril 2019**.

Les coeurs temps réels sont programmables et utilisables à travers le framework **remoteproc** présent à partir du Kernel linux 4.1 comme API de référence pour la gestion des coeurs temps réels. Cette carte est aussi économique en énergie car elle arrive à une consommation de 1.5W pour une utilisation normale et jusqu'à 0.5W en mode idle. Notre choix s'est porté sur cette carte pour l'intégration du projet en phase finale. Nous discuterons des détails techniques d'implémentation dans le prochain chapitre du rapport.

Bien que nous ayons choisi la carte **BeagleBone Black** pour l'intégration, nous avons tout de même choisi de mettre à profit les capacités d'une **carte Raspberry PI 3B+ pour valider le concept**. En effet, la carte Raspberry PI est capable de piloter les LEDs WS2812b à travers une bibliothèque développée par la société **Adafruit** mettant à profit un périphérique PWM. Ceci nous permet de valider nos idées sans trop se soucier des détails d'implémentation sur la BeagleBone Black.

La carte Raspberry Pi étant bien supportée pour tous les projets de bricolage et notamment ceux contenant des LEDs WS2812 et des services WEB, nous avons considéré qu'elle serait un bon point de départ pour accélérer les développements.

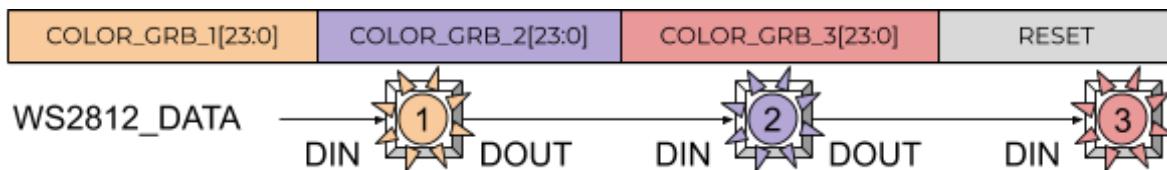
Génération de signaux de pilotage

Les LEDs WS2812b sont des LEDs chainables ([Datasheet WS2812b](#) et [Datasheet SK6812](#)) pilotable via un seul GPIO au travers d'une liaison série asynchrone de type MLI (modulation de largeur d'impulsion)/PWM.

La couleur de chaque LEDs est définie par un flux de 24bits sur la liaison série.

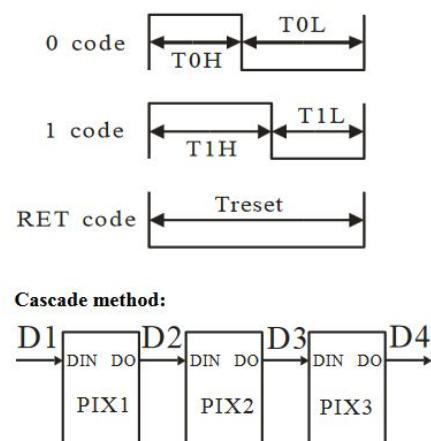


Les informations de couleurs transmises sur la liaison sont chaînées les unes après les autres. Les 24 premiers bits sont pour la première LED dans la chaîne, puis les seconds 24 bits pour la deuxième LED et ainsi de suite. Une fois les 24 bits des LEDs envoyés, un signal de RESET ≥ 50 us est envoyé sur la ligne.



Pour les signaux 24 bits, ils sont mis en forme de bits '**1**' et '**0**' de la façon suivante :

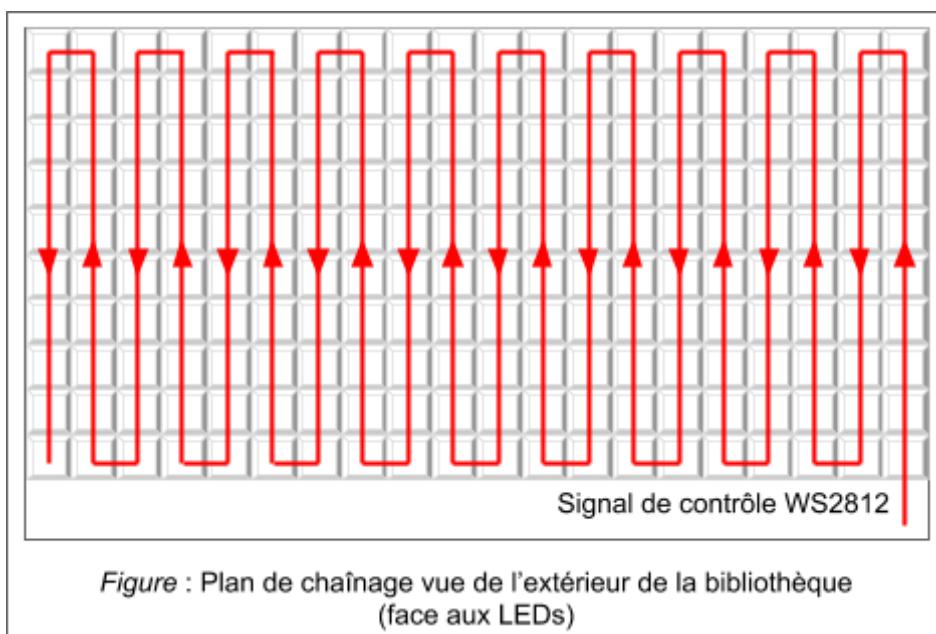
| Temps de référence | |
|--------------------|--------------------|
| T _{1H} | 0.6us ± 150 ns |
| T _{1L} | 0.6us ± 150 ns |
| T _{0H} | 0.3us ± 150 ns |
| T _{0L} | 0.9us ± 150 ns |
| RESET | ≥ 50 us |



On sait désormais comment les signaux de pilotage des LEDs sont générés. On peut tout à fait implémenter une commande à partir d'un périphérique PWM cadencé à **800kHz** pour lequel le rapport cyclique est de **50%** pour envoyer un '**1**'

et **25%** pour envoyer '**0**'. Il est aussi possible de générer ces signaux en utilisant du bit-banging pour générer logiciellement les timings. Il faut cependant pouvoir produire un code qui s'exécute de manière très prédictive et fiable. C'est le cas sur un cœur temps réel, un microcontrôleur ou un FPGA.

Nous allons maintenant parler du style de chaînage choisi pour le mur de LED. Nous sommes partis sur une technique en zig-zag permettant de réaliser un chaînage par tronçons. Ainsi, la ligne de donnée monte depuis la partie inférieure droite du mur de LED pour redescendre sur la colonne suivante (Cf schéma ci-dessous). Ainsi, on peut même imaginer agrandir ou rétrécir le mur de un ou plusieurs tronçons comme c'était rendu possible à l'origine par la solution des modules BlinkM.

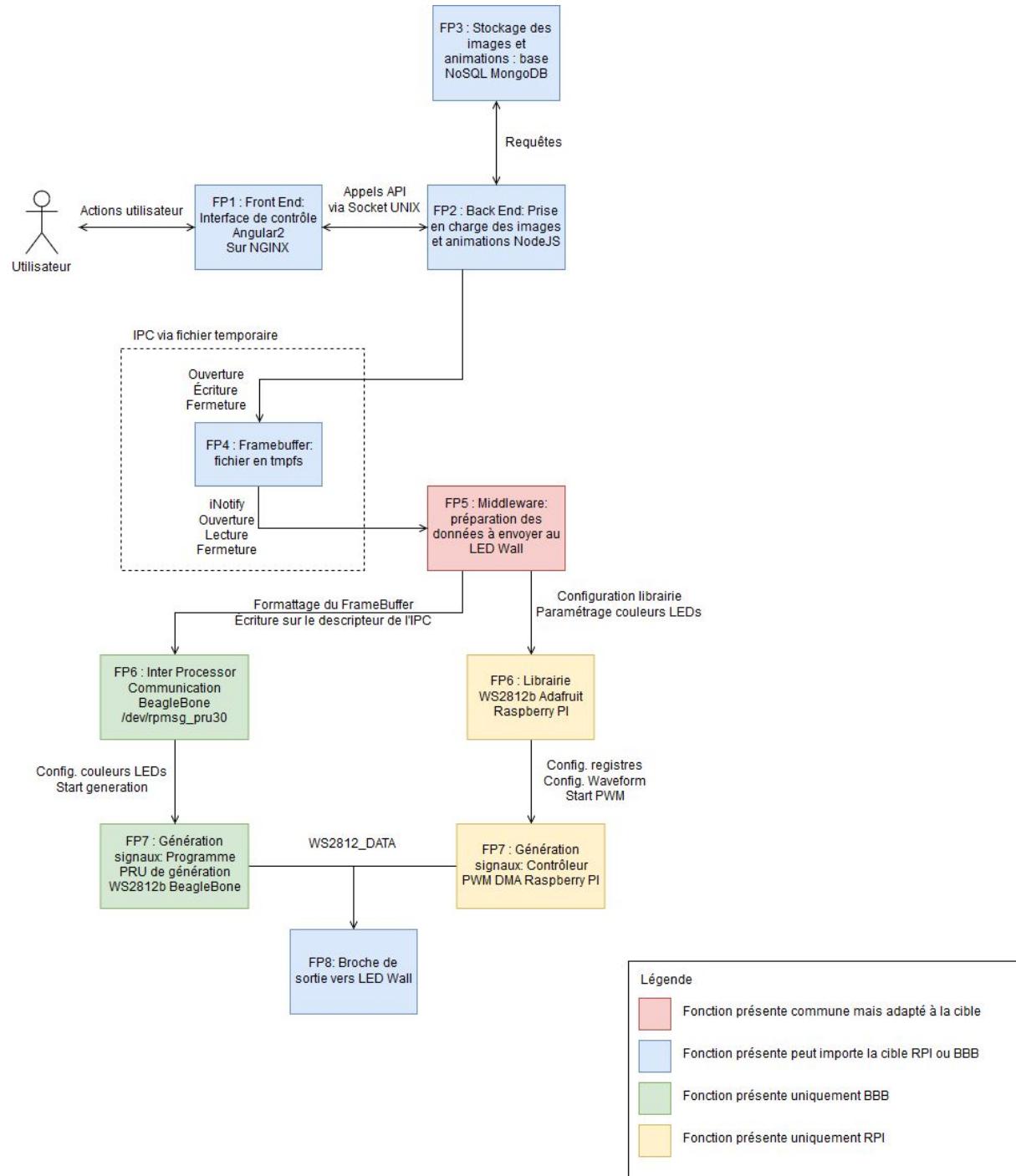


Architecture logicielle

Pour accompagner le système matériel, nous avons conçu une architecture logicielle permettant de réaliser le pilotage du mur. Nous avons opté pour une solution intégralement embarquée sur la carte Linux à base de services Web.

Nous passerons en revue le schéma de principe de cette architecture puis nous parlerons de la composition des services web. Enfin nous aborderons le fonctionnement de l'application Middleware d'adaptation bas-niveau.

Schéma fonctionnel de principe



Sur ce schéma de principe, nous retrouvons un ensemble de 8 fonctions principales à maîtriser. Pour que l'utilisateur interagisse avec le logiciel de pilotage du LED Wall, il faut créer un **front end** (**FP1**) qui sera le site web façade. Ce logiciel servira uniquement d'interface homme machine à travers un PC ou un smartphone. La fonction FP1 n'est qu'une façade et il est nécessaire de disposer d'un système de gestion des images et données à afficher sur le front end. Pour cela, on crée la fonction **FP2** servant de **back end** pour la gestion des données.

Le service back end étant un gestionnaire de données, il est nécessaire de rajouter une fonction de stockage de données **FP3**. Cette fonction correspond à une base de données à laquelle on doit accéder pour visionner les données de motifs et animations.

Nous avons voulu dissocier la partie web de la partie contrôle du mur. Ainsi, il est nécessaire de disposer d'un logiciel d'adaptation bas niveau pour la couche matérielle. Cette fonction est réalisée le MiddleWare **FP5**.

Pour communiquer avec le Middleware, nous avons besoin d'un mécanisme de communication inter-processus **FP4**.

Il y a plusieurs manières de réaliser la communication entre processus. On peut imaginer un pipe, un socket unix, un fichier temporaire, une message queue... Dans notre cas, nous avons choisi quelque chose de simple et robuste, à savoir un fichier temporaire.

En effet, la fonction **FP4** de framebuffer peut être réalisée par un simple fichier. On se retrouve alors dans une configuration de producteur-consommateur. La partie **back end** est le **producteur** des framebuffers à afficher à partir du contenu de la base de données et les poussent dans le fichier temporaire.

Ensuite, le **middleware**, qui se place en **consommateur** de cette donnée, peut être averti de la présence d'un nouveau framebuffer à l'aide du déblocage d'un sémaphore ou de la fonction de l'OS Linux inotify.

Pour finir, le **middleware** se charge de faire la liaison avec le matériel à l'aide des fonctions **FP6 et FP7**. Ces fonctions sont très liées à la cible et à la manière dont on génère les signaux. Ces fonctions attaquent concrètement le mur de LEDs par la fonction **FP8** représentant une broches à connecter aux LEDs WS2812b.

Service Web

Front end

L'IHM permettant de modifier les animations du LED Wall est destiné à des personnes ne possédant pas forcément des compétences techniques en informatique. Celui-ci doit alors être simple d'utilisation et doit proposer les fonctions principales suivantes :

- Créer/afficher un Pixel Art
- Créer/afficher un Word Art

Back end

Tout d'abord, le site web doit pouvoir enregistrer les Pixels Art créés par les utilisateurs (ou artistes). Pour cela, nous avons décidé de nous tourner vers une base de données NoSql.

Les avantages sont :

- Scalabilité (= schéma flexible)

On peut stocker toutes les informations d'un Pixel Art sans se soucier des relations. Si jamais le schéma du Pixel Art change, on peut le changer rapidement et facilement.

- Performances

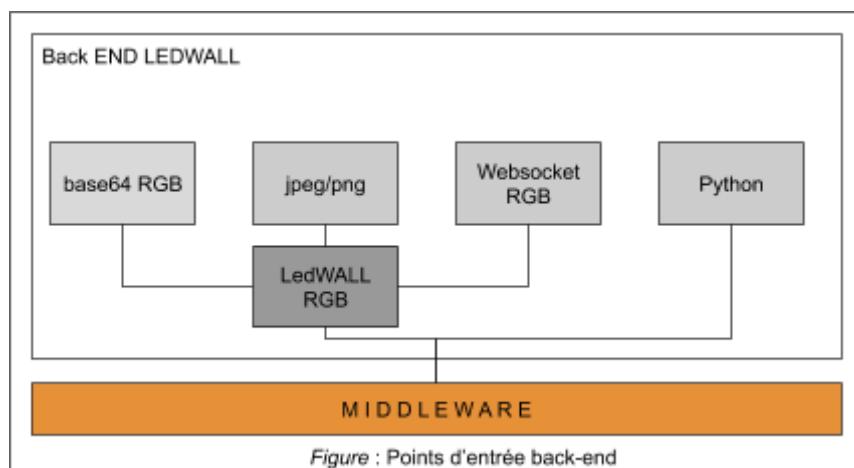
Nous avons très peu de relation et les requêtes sont simples, de ce fait, les performances sont élevés.

- Facilité de mise en oeuvre et moderne

Les bases de données NoSql sont rapides à mettre en place et sont souvent un choix par défaut dans certains frameworks modernes tels que express.

Le NoSql est de plus une technologie émergente qu'on rencontre de plus en plus souvent. Par conséquent, c'est intéressant pour ce projet pédagogique de découvrir ce type de base de données.

Le back-end de notre application contient notre API, mais aussi la partie communicant directement avec le LED Wall. Il y a plusieurs manières de communiquer avec le LED Wall comme indiqué ci-dessous.



Ainsi, notre backend n'est pas limité à une communication avec notre front-end. Nous avons donc des possibilités d'évolutivité, dans le cadre de futurs projets. Comme vous pouvez le constater sur le schéma ci-dessus, nous disposons aussi

d'une installation Python 3. Ceci afin de développer des applications dynamiques sans pourtant autant toucher au code du site web.

Voici quelques possibilités appartenant cependant au domaine de l'hypothétique:



MiddleWare

Afin de communiquer des données au middleware, il a été nécessaire de se mettre d'accord sur un formattage des données dans le framebuffer. La solution évidente, et celle qui fut donc implémentée, est la description des données au format RGB brut, comme montré ci-dessous.

```
65 6e 73 65 6d 62 6c 65 20 73 69 6e 67 65 73 20 66 6f 72 74 73 20 67 6e 65 75 20 67 6e ...
```

Figure : Format du framebuffer

Où chaque trio d'octets représente la couleur d'un pixel. Le tableau de pixels est par ailleurs dessiné ligne par ligne. Voici le pseudo code utilisé dans le middleware pour afficher une image du framebuffer:

```
for y in range(LED_WALL_HEIGHT):
    for x in range(LED_WALL_WIDTH):
        idx_color += 3
        drawPix(x,y,fram[idx_color],fram[idx_color+1],fram[idx_color+2])
```

Où drawPix est une fonction fictive prenant en argument la coordonnée X et la coordonnée Y d'un pixel, puis les valeurs de couleur rouges, vertes et bleues respectivement de ce pixel, d'après la lecture du framebuffer, représenté ici par le tableau **fram**.

Réalisations

Construction du mur de LEDs

Production des LEDs WS2812b

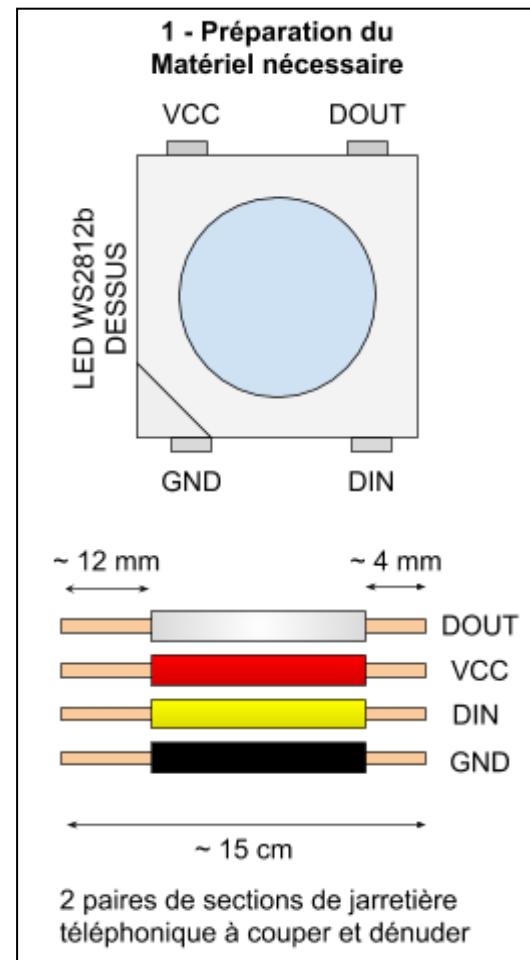
Chaîne de production

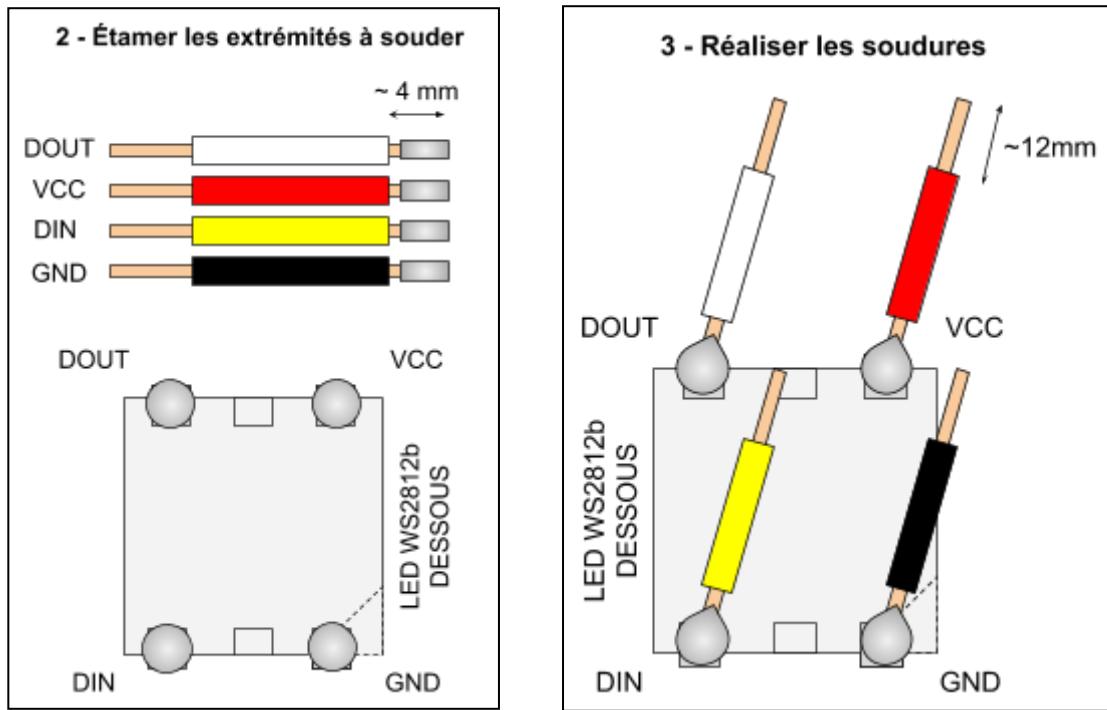
Pour produire les LEDs, nous avons réalisé une chaîne de production sur laquelle on peut affecter des ressources parallèles. Il y a quatres étapes dans la production des LEDs. La première consiste à préparer le matériel nécessaire. Nous aurons donc besoin de 4 fils de couleurs rouge, blanche, jaune et noire. C'est fils sont obtenus à partir d'une bobine de jarretière téléphonique à 2 fils. Ainsi, on disposera des deux paires rouge/blanche et jaune/noire.

Etape 1: Préparation du matériel

L'étape 1 correspond à la découpe et au dénudage des 2 paires de section de jarretière téléphonique.

Nous aurons besoin de sections de 15 cm dénudé d'un côté de 12 mm environ et de l'autre de 4mm. Le côté 12 mm est destiné à être connecté à l'aide des connecteurs WAGO présent sur le mur de LED. L'extrémité de 4 mm devra être soudée sur les pads de la LED.



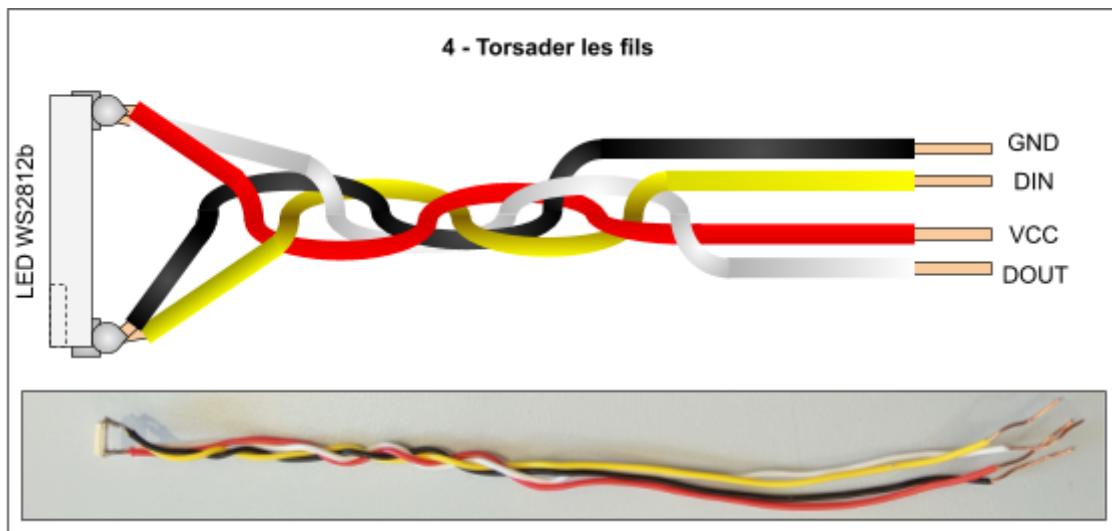


Etape 2 : Étamer les extrémités à souder

Dans cette étape, il faudra étamer les extrémités des 4 fils au niveau du cuivre dénudé sur 4 mm. Il faudra faciliter l'étape de soudure en étamant les pads de la LED WS2812b. Vous pouvez vous aider de l'illustration ci-dessus. **Attention, la température du fer à souder en doit pas dépasser 300° !**

Etape 3 : Réaliser les soudures LED/Fils

Dans cette étape, il faudra réaliser la connexion des fils précédemment étamé en suivant le schéma numéro 3 ci-dessus. Veillez à souder les fils à perpendiculaire au dos de la LED. Si la phase d'étamage est correcte, inutile de rajouter de l'étain à ce niveau. Chauffer simplement la bille présente sur le pad de la LED en y mettant en contact le fil étamé suffira à le souder convenablement. Pour faciliter l'étape de soudure et d'étamage de la LED, nous vous recommandons d'utiliser un étaux pour cartes électronique et d'y pincer la LED. **Attention à ne pas serrer, la LED doit tenir mais ne pas être déformée au risque d'être endommagée.**



Etape 4 : Torsader les fils

Afin de diminuer les chances de casse des soudures et améliorer la répartition des efforts mécaniques appliqués à la structure, nous recommandons de torsader les paires de fils soudés en étape 3. Pour ce faire, pincez les paires de fils à des environ 15 mm des soudures au niveau de la LED. Maintenir fermement et torsader les paires de fils du côté dénudés de 12mm. Le résultat obtenu doit ressembler à celui de la figure 4 ci-dessus.

Parallélisation des tâches:

Pour accélérer la production, nous avons réparti les tâches de production. Il faut une ressource sur chacune des tâches suivantes:

1. Couper les fils
2. Dénuder les fils
3. Étamer les fils
4. Étamer la LEDs et souder les fils
5. Torsader la LED
6. Tester la LED

Si l'ensemble de ces tâches sont réalisés par une seule ressource, comptez environ **4 minutes** pour un technicien formé pour réaliser la construction d'une LED.

Si vous passez par la chaîne de production, vous pouvez espérer une diminution de ce temps à **1 à 1 min 30 s** par LED.

Nous avons produit ces LEDs en utilisant la méthode de la chaîne de production. En effet, il nous aura fallu **5h** pour produire et tester l'ensemble des 200 LEDs. La simplicité du process de fabrication nous a permis d'avoir seulement **3 LED HS sur les 200 fabriqués** soit un **taux de pièces refoulées de 1.5%**. C'est un excellent résultat.

Nous allons justement en venir la création du banc de test et de la procédure mise en place pour valider la production des LEDs.

Banc de test

Afin de tester nos LEDs, nous avons créé un banc de test basé sur une carte Arduino, une platine breadboard, une LED WS2812b fonctionnelle et un programme de test.

//INSERER SCHEMA DU BANC ET IMAGE/DIAGRAMME

//EXPLIQUER LE PROGRAMME ARDUINO

Pour le déroulement du test, le technicien doit suivre l'organigramme suivant afin de conclure sur le fait qu'une LED est opérationnelle ou HS:

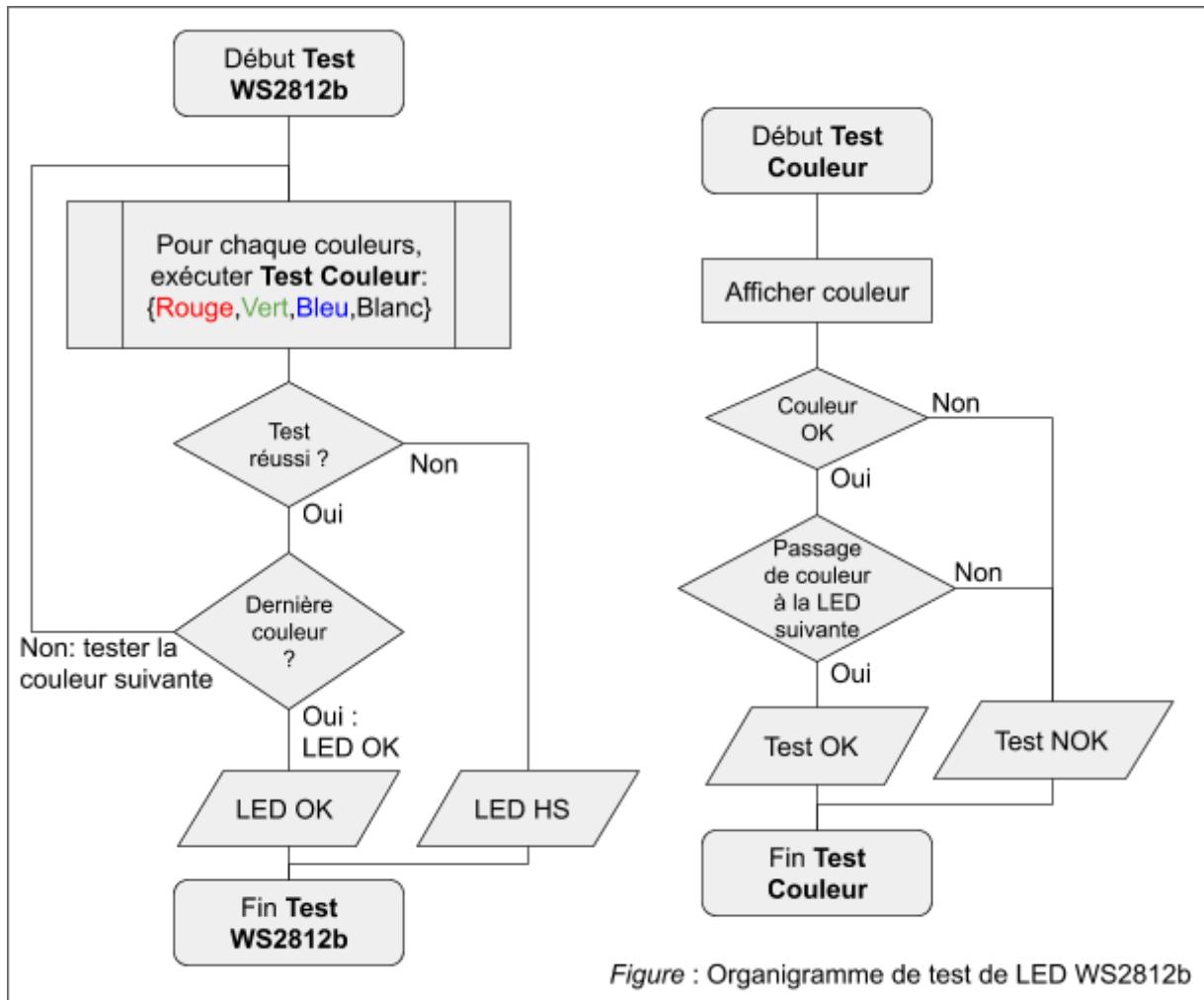


Figure : Organigramme de test de LED WS2812b

L'objectif de ce test est de vérifier que les couleurs sont bien affichés par la LED même lorsque l'on affiche du blanc. En effet, la LED doit être capable d'afficher individuellement les couleurs Rouge, Vert, Bleu mais aussi la couleur Blanche. Il faut aussi vérifier que la LED fait bien suivre les signaux à la LED suivante. Pour ce faire, le programme de test implanté sur le banc est conçu pour piloter 2 LEDs WS2812b et afficher des couleurs identiques sur les 2 LEDs. Il faut s'assurer que pour chaque couleurs testés, la LED suivante est bien allumée avec la même couleur que la LED à tester.

Lors de nos tests, nous avons décelé **3 LED HS** pour les causes suivantes:

- **2 LEDs ne s'allument** pas du tout peut importe la couleur
- **1 LED s'allume correctement** mais **ne fait pas suivre le signal** aux LEDs suivantes

Modification de la structure en bois

Nous avons dû faire face à un **changement de programme imprévu en Décembre 2018**. En effet, le **service d'administration de Polytech Tours a validé le projet de construction** d'une salle de visioconférence et travail de groupe à l'intérieur de la bibliothèque du département informatique. À ce stade, la volonté du service administratif était de tout simplement détruire le LED Wall et définitivement le retirer de la bibliothèque. La situation était d'autant plus accablante que M.Esswein et M.Rolland ne semblaient pas avoir été prévenu de cette décision.

Il a donc fallu proposer des alternatives pour permettre au LED Wall de survivre aux travaux. Après 2 semaines de réflexions, il a été décidé que la solution serait de **réduire la taille du mur de LED à 16 colonnes et 10 lignes** et de légèrement le déplacer pour laisser de la place à la nouvelle salle visio. On perd donc 4 colonnes passant d'une **Résolution de 20 x 10 à 16 x 10**.

Nous avons accepté cette solution et nous avons agit vite pour faire les modifications du mur de LED. Avec l'aide de **G.Maillaud** et de toute l'équipe du LED Wall, nous avons **déplacé et découpé la structure en bois**.

En retirant la structure, nous nous sommes rendu compte que les équipes précédentes avaient installé du papier calque sur les carreaux de la bibliothèque. Malheureusement, ceux-ci étant **mal collés et déchirés**, ils se sont arrachés lors du déplacement du mur de LEDs.



Nous avons donc **pris l'initiative de nettoyer les vitres et appliquer de nouveaux calques**. Cette fois-ci nous avons utilisé des bandes de calques de **37.5 cm de largeur** permettant de couvrir **2 lignes de carreaux** en verre d'un seul tenant. Nous avons réalisé la fixation des calques à l'aide d'un **scotch transparent de 40 mm** de largeur (Cf. figure ci-dessus). Le résultat est plus propre et robuste que le calque appliqué auparavant.

En réalisant ce **travail rapidement**, nous avons laissé le **champ libre pour la construction** de la salle visio et tout s'est déroulé pour le mieux. Nous avons pu garder le LED Wall dans la bibliothèque sans impacter les travaux prévus.

Nous avons profité de devoir déplacer le mur de LED pour faire ces finitions.

En effet, une fois la salle visio construite, il aurait été très difficile de les réaliser, notamment pour le calque et le nettoyage des vitres.



Cette nouvelle nous a beaucoup inquiété car si aucune solution n'était trouvée, ceci aurait probablement signé la mort du projet. Heureusement, nous n'en sommes pas arrivé là.

Cette partie de modification est venue avant la partie de câblage. Cependant, nous avions déjà câbler les LEDs avant que la cloison de la salle visio ne soit construite. Nous allons maintenant vous exposer la réalisation du câblage.

Câblage et alimentation

Principe de câblage d'une LED

Pour câbler le mur de LEDs, nous avons récupéré les fils utilisés précédemment pour les modules BlinkM. Nous avons aussi récupéré les connecteurs **WAGO 3 points**. Ainsi, les 4 fils VCC, GND,DIN,DOUT d'une LED trouvent naturellement leurs place dans les 4 connecteurs prévus initialement pour les BlinkM (Cf. illustration ci-dessous). Nous avons ensuite utilisé les 2 emplacements de connexion restants sur chacuns des connecteurs pour distribuer l'énergie et réaliser le chaînage.

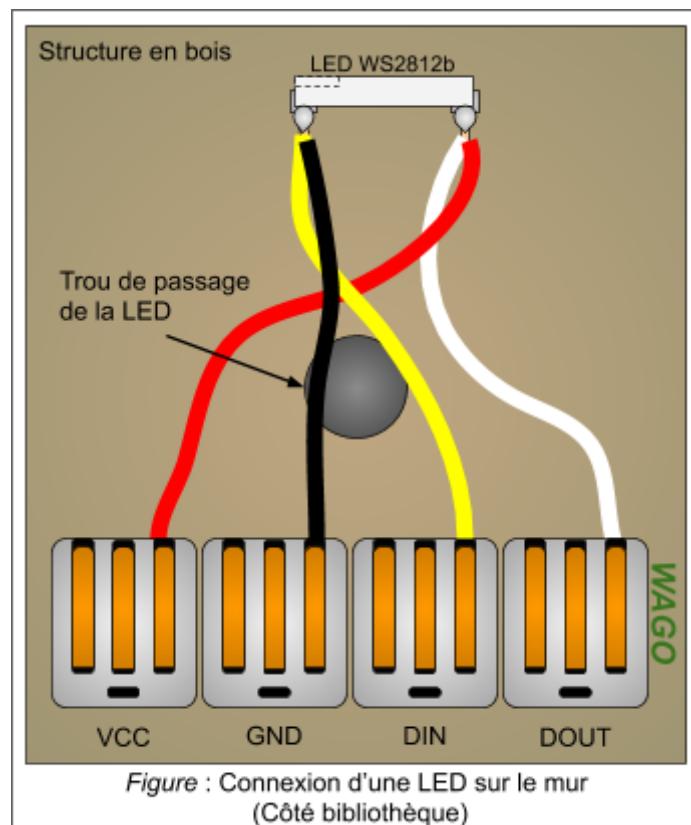
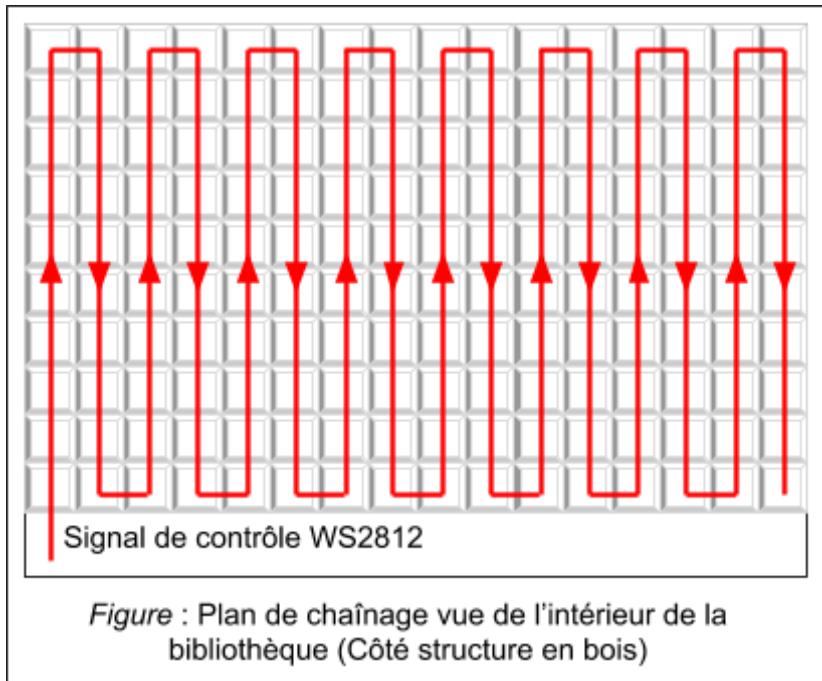


Schéma de chaînage



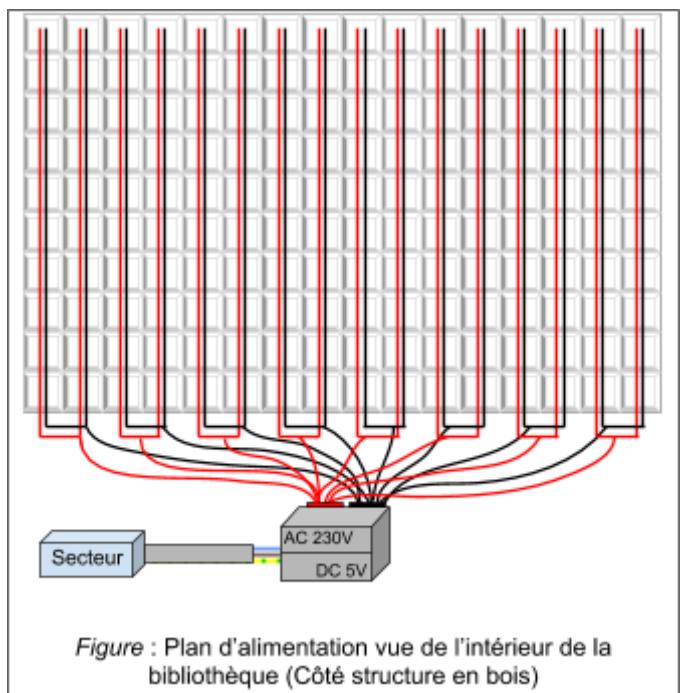
Le schéma de chaînage réalisé a été expliqué dans la partie précédente du rapport. Il a simplement fallu relier les connecteurs WAGO dans le sens DIN → DOUT en suivant le schéma de chaînage présenté ci-dessus à l'aide de fils de **25 cm environ**. Le câblage a été très rapide, il aura fallu **3h et 3 ressources** pour réaliser intégralement le câblage. Pour réaliser ces connexions, nous avons **recyclés les fils auparavant prévu pour les modules BlinkM**.

Schéma d'alimentation

Afin de répartir le courant utilisé par les LEDs et éviter autant que possible les **pertes dans les fils dût au fort courant** consommé sur la ligne 5V, nous avons réalisé un **câblage en étoile**. Ainsi, chaque tronçon dispose de ses **lignes d'alimentations** (VCC et GND) arrivant **jusqu'aux borniers du bloc d'alimentation** (Cf schéma ci-après).

Il n'est pas possible de chaîner l'alimentation des LEDs de la même manière que le signal de commande car ceci induit une **chute de tension en bout de fil pouvant aller jusqu'à 2.5V** (confirmé expérimentalement).

Avec une alimentation en étoile, la longueur des fils de chaque branches



est limité et on ne branche pas plus de 20 LEDs par sections d'alimentation.

Tests *in situ*

Mesures de continuités

Afin de s'assurer que les connexions ont été correctement réalisés, il est nécessaire de réaliser des tests de continuités sur les lignes d'alimentations. En effet, il faut éviter les court-circuits et vérifier que chaque LED peut être alimentée en énergie correctement.

Pour ce faire, nous avons testé notre câblage à l'aide d'un multimètre équipé de la fonction **test de continuité**. Cette fonction permet de vérifier qu'un câble n'est pas sectionné et est aussi utile pour suivre des pistes sur un circuit imprimé.

Pour isoler rapidement les anomalies de câblage, il faut tester chaque tronçons du mur de LED. Pour ce faire, il ne faut pas connecter les fils VCC et GND des tronçons au bloc d'alimentation pendant la phase de test.

La procédure utilisée est la suivante:

1. Vérifier la continuité de la ligne VCC depuis le bas du tronçon jusqu'à l'extrémité haute du tronçon.
2. Vérifier la continuité de la ligne GND de la même façon
3. Vérifier qu'il n'y a pas de continuité entre la ligne VCC et GND sur le tronçon en bas du tronçon, en haut et au milieu.

Si des modifications de câblage doivent être réalisées pour être conforme au schéma, il est nécessaire d'effectuer les vérifications à nouveau.

Une fois l'ensemble des continuités vérifiées, on va pouvoir passer aux tests de tensions.

Mesures de tensions

Avant de connecter les fils, il est nécessaire de vérifier le réglage de **la tension de sortie de l'alimentation**.

Il faut s'assurer que rien n'est branché sur le bloc d'alimentation hormis le **câble secteur 230V**. Après mise sous tension du bloc, vérifier la tension sur les borniers de sortie +5V et 0V à l'aide d'un Voltmètre.

La tension de sortie doit être comprise **entre 4.95V et 5.10V**. Si besoin, ajuster à l'aide du potentiomètre de réglage.

Ensuite, on peut **connecter les lignes VCC et GND des tronçons** au bloc d'alimentation en prenant soin de réaliser **cette opération hors-tension**.

Vérification du chaînage

Pour vérifier le chaînage, nous avons dû générer des signaux de pilotage des LEDs et observer le résultat au niveau des couleurs affichés. La difficulté du test réside dans la création d'un motif de test permettant la vérification du chaînage efficacement.

Validation des couleurs et du pilotage

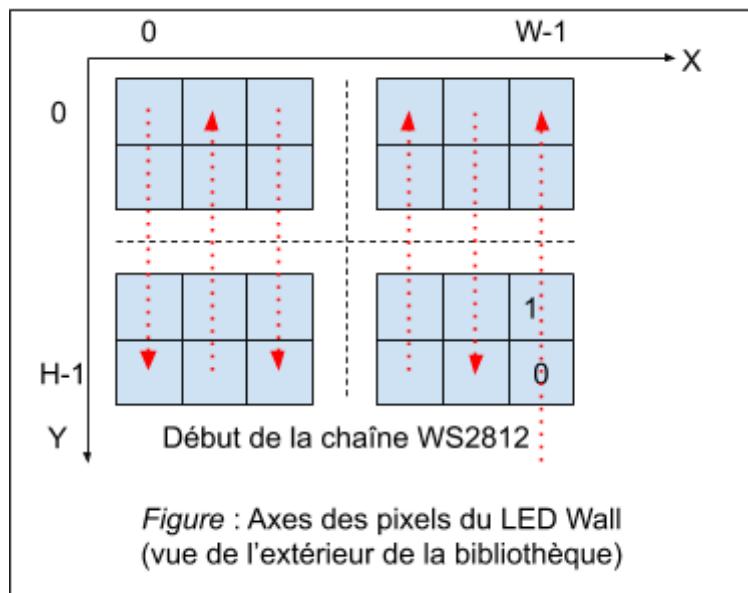
Pour tester le chaînage du mur de LED, nous avons dû concevoir une carte de pilotage associée à un programme

Préparation d'une carte de test

La carte de test utilisée comme cerveau de pilotage est une carte Arduino UNO accompagné d'un programme de démonstration pour le pilotage de bandes de LEDs WS2812b.

Lien entre position (X,Y) et position dans la chaîne

Pour piloter les LEDs WS2812b, il est nécessaire de transformer le tableau bidimensionnel des couleurs des LEDs en une chaîne correspondant au câblage du mur de LEDs. Nous avons fait en sorte que le LED Wall soit vu comme un écran de pixels classique avec l'axe X allant de gauche à droite et Y de haut en bas.



L'algorithme réalisant le lien entre position (x,y) et indice dans la chaîne est le suivant:

$x = \text{position le long de l'axe } x \text{ dans l'intervalle } [[0, W]]$ avec W le nombre de pixels en largeur.

$y = \text{position le long de l'axe } y \text{ dans l'intervalle } [[0, H]]$ avec H le nombre de pixels en hauteur.

PixelId(x , y) :

Si ((W - 1 - x)%2 == 1) Alors
retourner (W - 1 - x) * H + y

Sinon

retourner (W - 1 - x) * H + (H - 1) - y

Codage du programme d'animation (Arduino)

A l'aide de l'algorithme de mappage, nous avons mis en place un programme de test Arduino permettant d'afficher des motifs sur le LED Wall. A l'aide de l'IDE Arduino et de la librairie FastLED, nous avons pu programmer les premiers motifs à afficher sur le LED Wall. Ce motif est animé et se déplace de gauche à droite sur le LED Wall. Il est intégralement calculé à la volé par la carte Arduino.

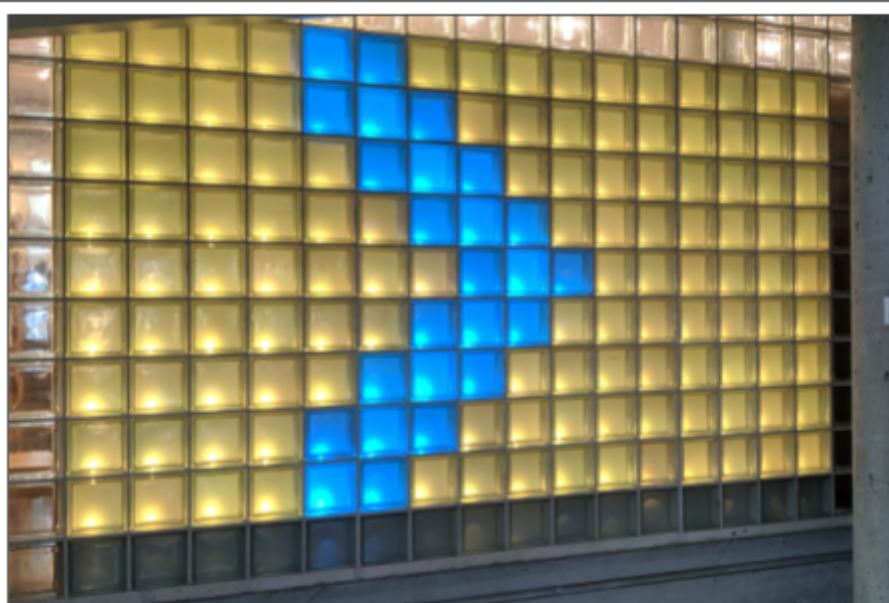


Figure : Motif de démonstration (fonctionnant sur une carte Arduino)

Système de contrôle et pilotage

Codage du site Web

Base de données NoSQL

Afin d'implémenter la base de données NoSql, nous avons décidé de nous tourner vers **MongoDB**, un système de stockage de données NoSQL très populaire. Celui-ci comporte une grande communauté avec une librairie maintenue à jour.

Nous avons créé un schéma pour chaque objets (ou collections) que l'on stocke :

- Pixel Art

Pour l'édition de Pixels Art, nous nous sommes appuyé sur un projet Github existant que nous aborderons plus en détail dans la partie [Front-end Angular 7 et NGINX](#). Celui-ci enregistre et charge des Pixels Art sous format JSON. De ce fait, nous avons repris en partie ce format pour l'objet "Pixel Art".

```
3  let Schema = mongoose.Schema({
4    modelVersion: String,
5    base64Thumb: String,
6    piskel: {
7      name: String,
8      description: String,
9      fps: Number,
10     height: Number,
11     width: Number,
12     layers: [String]
13   },
14 },
15 {
16   timestamps: true
17});
```

Figure : Schéma du modèle PixelArt

Dans ce schéma, nous retrouvons la miniature du pixel art (affiché sur la page principale du site), et des informations sur le pixel art en question tel que sa taille, son nom, sa description, les FPS (si celui-ci est un gif), et pour finir, les couches du pixel arts (si celui-ci est un gif, il comporte plusieurs couches).

- Animations

```
3  let Schema = mongoose.Schema({
4      name: String,
5      animationItems: [
6          {
7              time: Number, // seconds
8              pixelart: {
9                  type: mongoose.Schema.Types.ObjectId,
10                 ref: 'Pixelart'
11             }
12         },
13     ],
14     timestamps: true
15 }) ;
```

Figure : Schéma du modèle Animation

Pour l'objet “animation”, nous avons le nom de l'animation et un tableau “animationItems” comportant des pixels art et leurs temps d'affichage.

- Word Art

```
3  let Schema = mongoose.Schema({
4      text: String,
5      textColor: [Number],
6      bgColor: [Number]
7  },
8  {
9      timestamps: true
10});
```

Figure : Schéma de modèle WordArt

Ensuite, nous avons le schéma “wordart” qui se compose d'un texte, d'une couleur de texte et d'une couleur de fond.

- Script

```
3   let Schema = mongoose.Schema({  
4     path: String,  
5     extension: String,  
6     filename: String,  
7     name: String  
8   },  
9   {  
10     timestamps: true  
11   });
```

Figure : Schéma du modèle Script

Pour finir, nous avons commencé le prototype de la fonction “script python” dans le site web et pour cela, nous lui avons créé un schéma également comportant les informations : chemin, extension, nom du fichier enregistré et nom du fichier.

Back-end NodeJS

Nous avons décidé de nous tourner vers un framework populaire de nodejs : **ExpressJS**. Celui-ci est très léger et rapide à implémenter.

Le back est ainsi structuré de cette manière :



Nous utilisons ce framework en tant qu'API qui sera appelée par le front-end. De ce fait, nous l'avons sécurisé avec 2 modules :

- **helmet** : un middleware de sécurité qui gère plusieurs types d'attaques dans les protocoles HTTP / HTTPS. C'est un ensemble de neuf middlewares internes chargés de traiter des paramètres HTTP (ClickJacking, Cross-Site Scripting, sniffing avec Mime Type...).
- **CORS** : permet de définir le ou les domaines ayant accès à l'url de l'API. Nous avons défini 2 types de lancement l'application (production - développement). Si celui-ci est lancé en développement, alors n'importe quelle url peut accéder à l'api. En revanche si c'est lancé en production, uniquement l'url du front est autorisée.

Pour afficher les images créées par l'utilisateur, nous avons décidé de développer une bibliothèque de fonctions appelée **ws2812.js**. Ainsi, nous disposons de fonction nous permettant entre autre de:

- Convertir des images JPEG/PNG en tableau RGB.
- Afficher les images et animations
- Afficher les WordArts
- Stopper les animations

Il est ainsi très simple de piloter notre LedWALL depuis le back-end de notre application. Voici quelques exemples pour confirmer nos dires:

Initialisation et affichage de texte rouge sur fond noir:

```
anim_interval_id = ws2812.WS2812RunWordArt("Coucou", '#FF0000', '#000000',  
anim_interval_id);
```

où **anim_interval_id** est une variable permettant de sauvegarder l'id de la fonction **setTimeout()**, propre au langage **Javascript**. Ainsi, nous pouvons stopper un motif à n'importe quel moment où endroit du programme.

Front-end Angular 7 et NGINX

- Angular

Nous avons décidé d'utiliser **Angular 7** pour construire le front-end. C'est un framework front-end créé par Google, à la mode et avec une large communauté. Cette technologie fonctionne sous forme de "composants". Un composant peut être utilisé plusieurs fois et nous pouvons imbriquer un composants dans un autre.

Nous avons également ajouté au front-end le framework populaire **Bootstrap** et les composants **Angular Material** afin d'avoir une interface **responsive** et donc utilisable sur téléphone mobile.

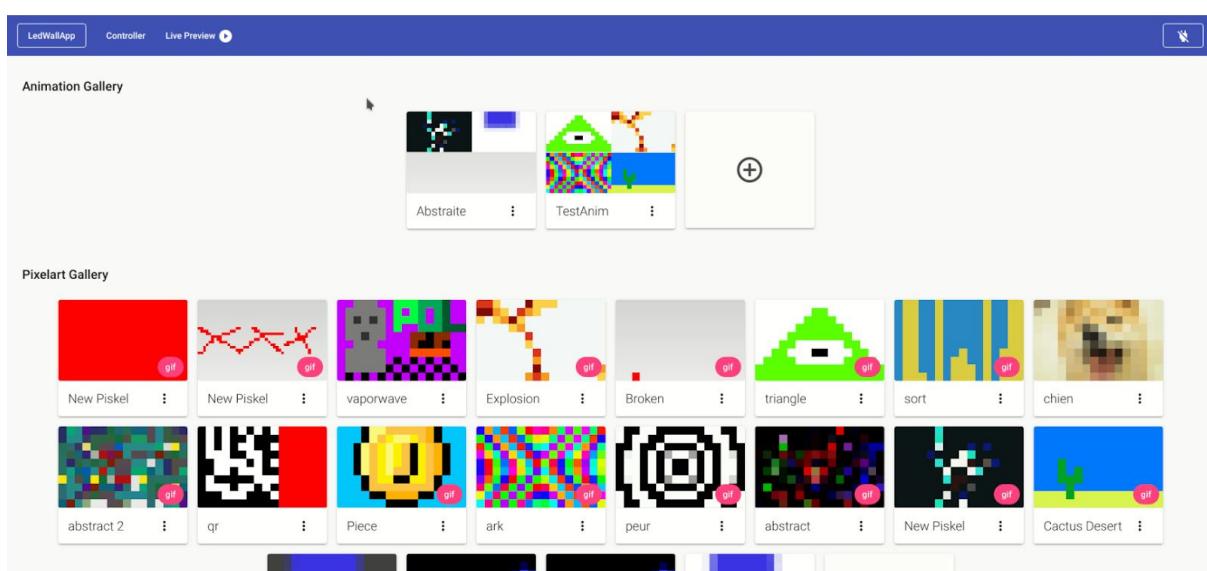
A. Les composants

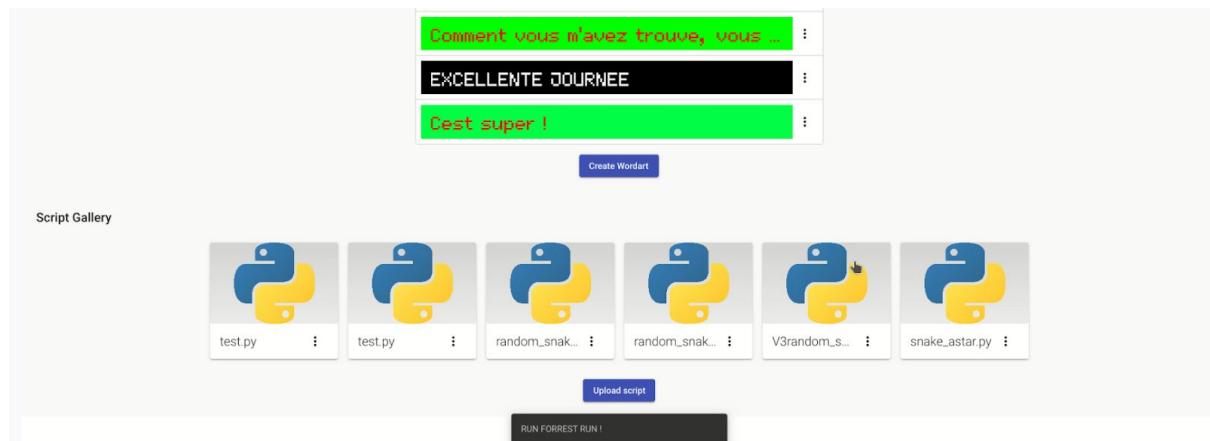
Voici les composants que nous avons créés :

1. Les pages principales

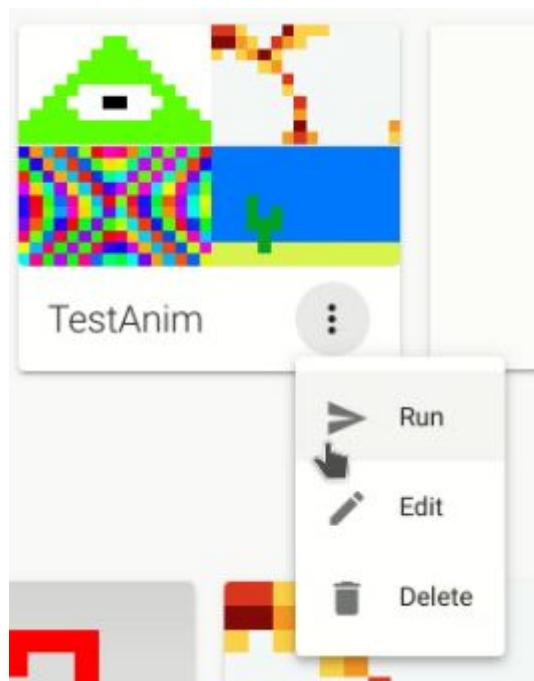
a. Composant "gallery"

La page principale du site web est le composant "gallery". Celui-ci se charge d'afficher les éléments **Animations**, **Pixels Art**, **Worlds Art** et **Scripts** enregistrés en base de données.





Nous retrouvons dans ce composant, les fonctions permettant de jouer, supprimer ou éditer un des éléments.

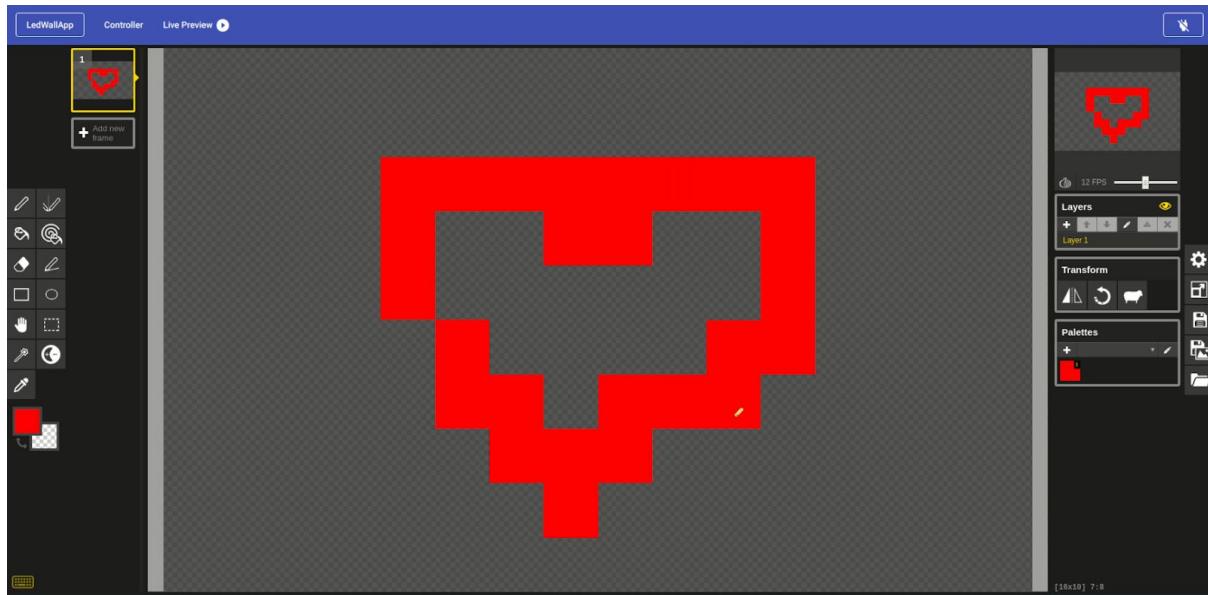


b. Composant “editor”

Ce composant permet l'édition et la **création de Pixel Art** (gif ou non).

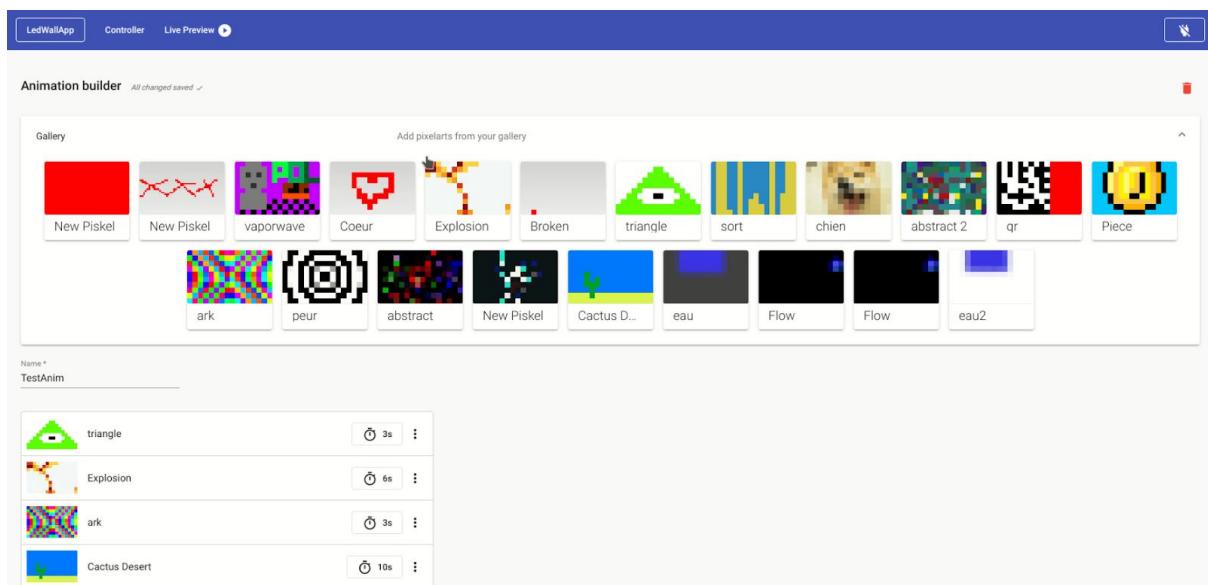
Afin d'avoir un éditeur de Pixel Art performant et complet, nous avons réussi à intégrer un éditeur déjà existant nommé “Piskel” (<https://www.piskelapp.com/>). Nous l'avons adapté à notre site web en redimensionnant la taille d'un Pixel Art, en enlevant certaines fonctionnalités non nécessaires et en ajoutant les appels API de création et modification d'un Pixel Art.

DII4 - Projet Collectif - LED Wall



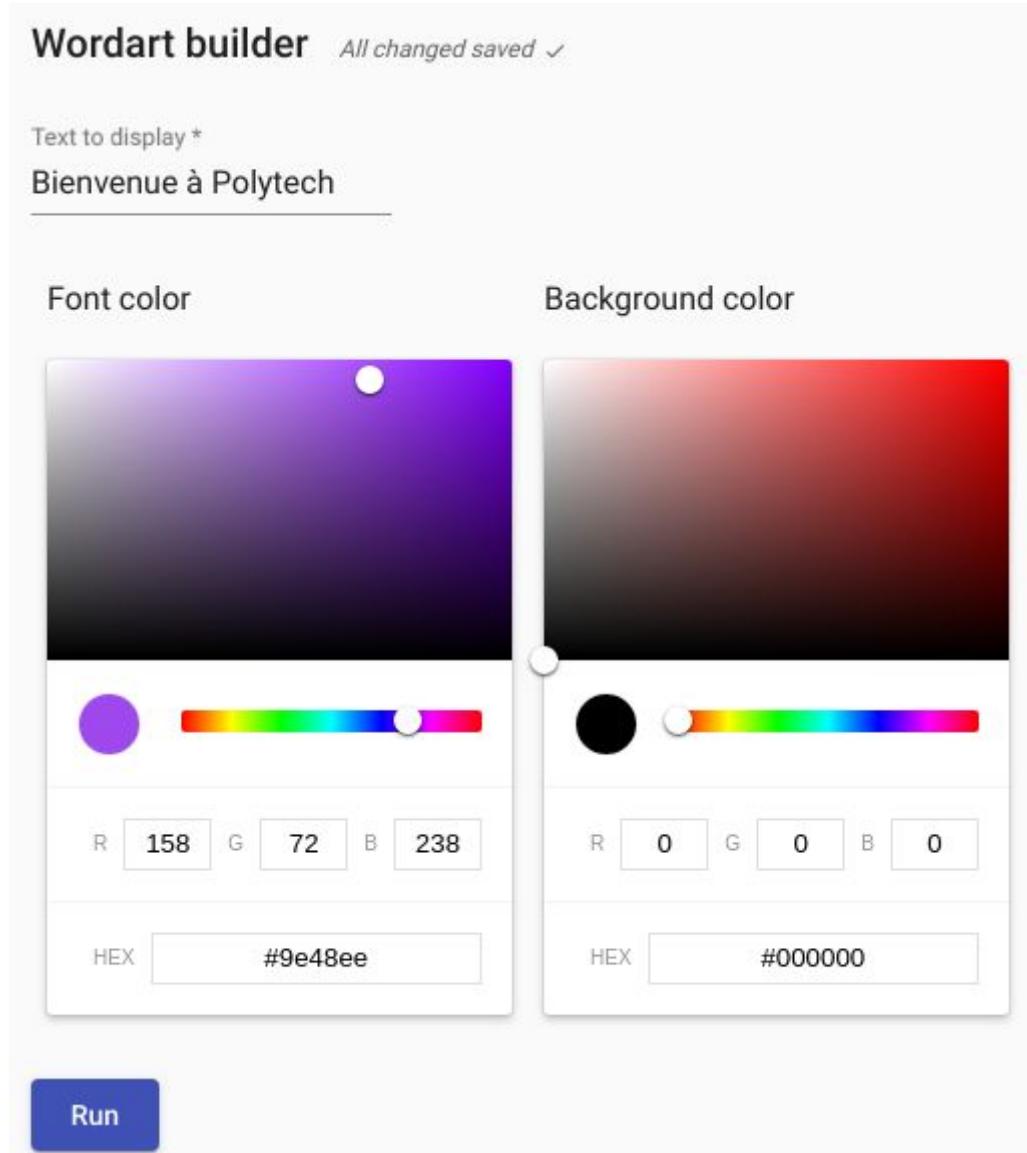
c. Composant “animations”

Il y a aussi la possibilité de **créer des animations de Pixel Arts**. Durant la création d'une animation, nous avons pouvons aller chercher les Pixel Arts préalablement créés, de les ajouter à une playlist, de les ordonner et de modifier leurs temps d'affichage.



d. Composant “wordarts”

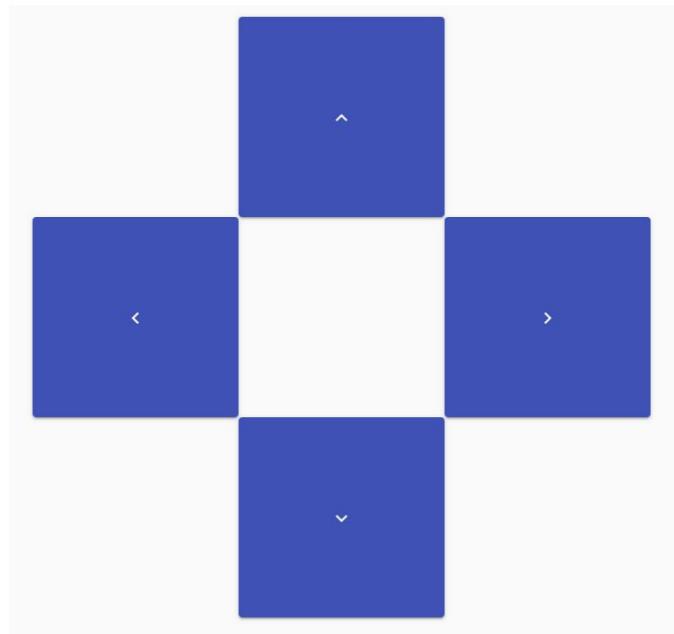
Le composant “wordarts” permet de **créer un texte défilant** en ayant la possibilité de choisir la couleur du texte, et la couleur de fond.



e. Composant “controller”

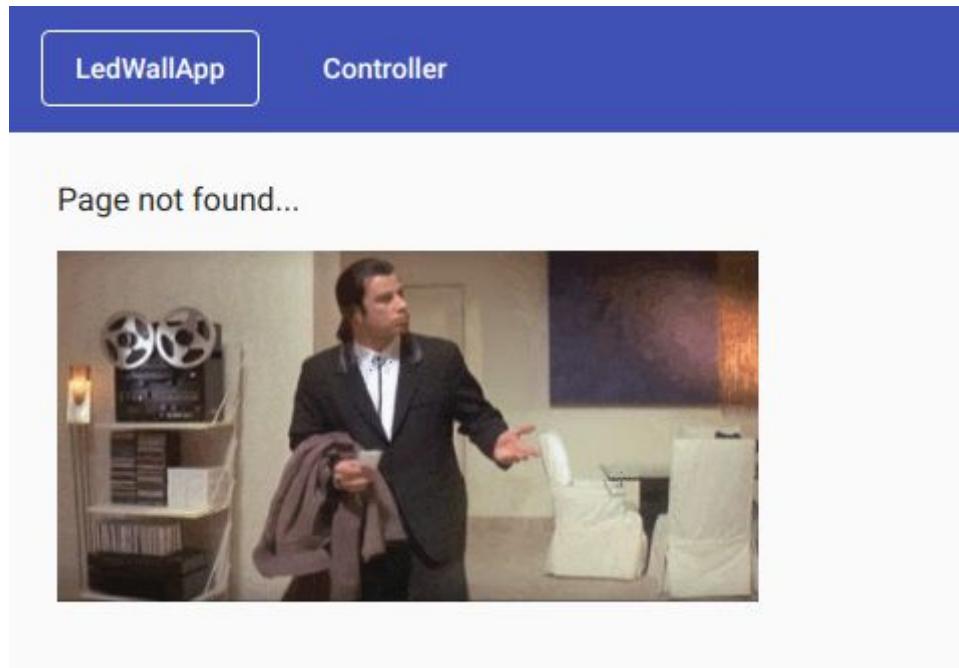
Cette page affiche simplement 4 boutons : haut, bas, gauche, droite. Ces boutons envoient des appels API qui simulent les touches “fléchés” du clavier à l'aide d'un “hook”. Cela permettra d'ajouter des **interactions durant l'exécution d'un script python**. (ex : jeu snake, changer d'interface...)

Dans l'avenir, afin d'avoir plus de réactivité, l'implémentation d'un websocket serait nécessaire.



f. Composant “notfound”

Un composant très simple affichant une **page 404** composée d'un gif.

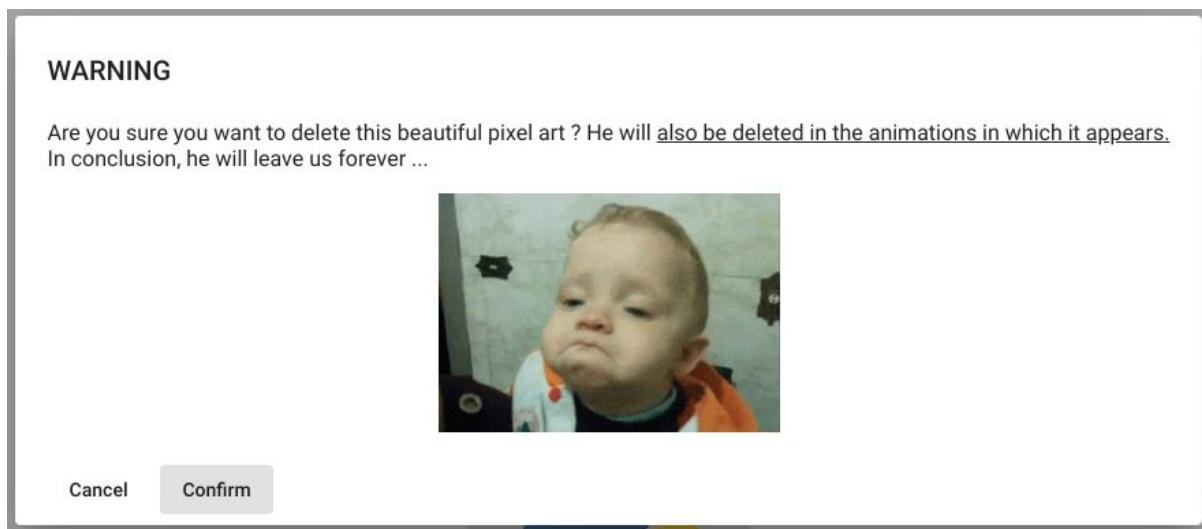


2. Modals

a. Composant “confirm-dialog”

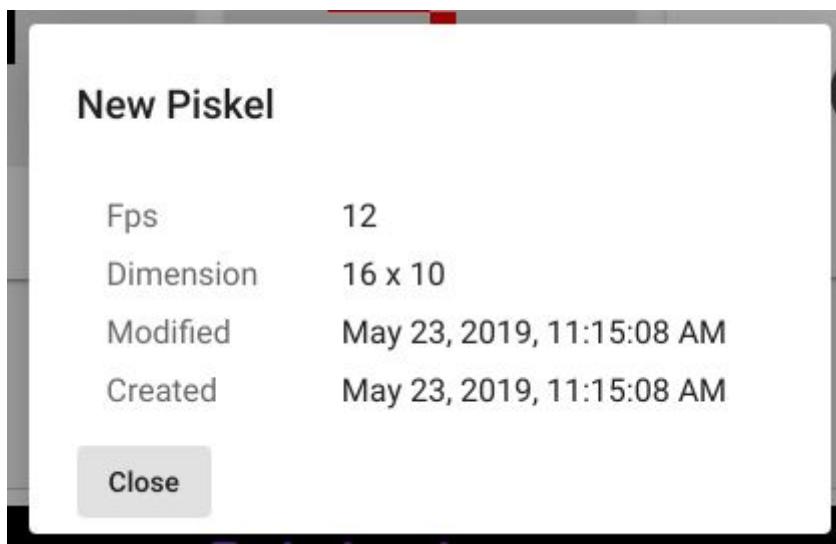
Ce composant se charge d'afficher un modal de confirmation qui prend en entré un gif, un texte et est composé de 2 boutons : “Confirm” et “Cancel”.

Celui-ci est principalement utilisé pour la **confirmation de la suppression d'un élément.** (ex: Word Art...)



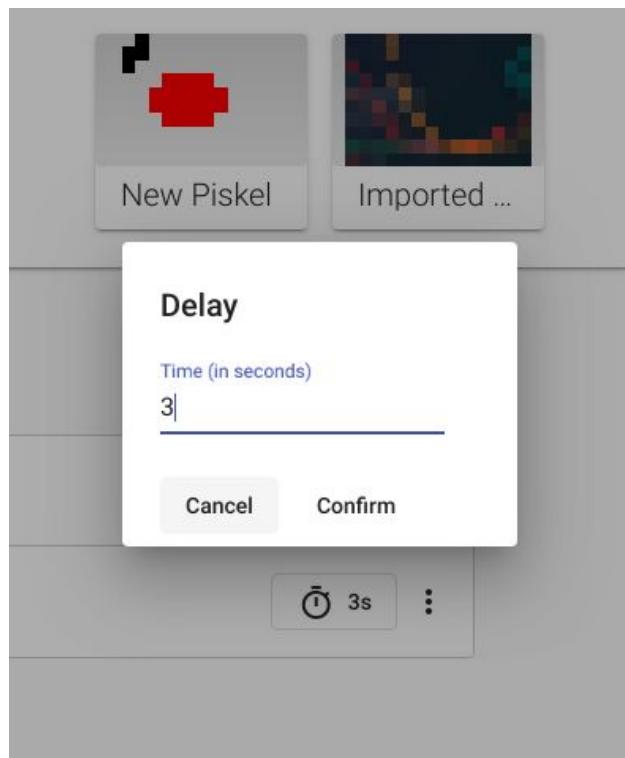
b. Composant “pixelart-information-dialog”

Ce composant affiche un modal composé des **informations d'un Pixel Art.**



c. Composant “time-dialog”

Ce composant est utilisé dans la page “Animations” lors de la **modification du temps d'affichage** d'un élément.



3. Autres composants

a. Composant "footer"

Comporte le footer de la page.

b. Composant "saving"

Ce composant permet la **sauvegarde automatique** des Pixels Art, des Worlds Art et des Animations.

A partir du moment où l'élément à un nom, il est automatiquement enregistré. A chaque modification, un "**debounceTime**" (limite la fréquence d'appels API) est appelé pour aller enregistrer l'élément en base de données. Cela permet d'éviter que l'utilisateur oublie ou tout simplement est la nécessité d'appuyer sur un bouton pour sauvegarder.

L'inspiration de ce composant est venu de la suite Google.

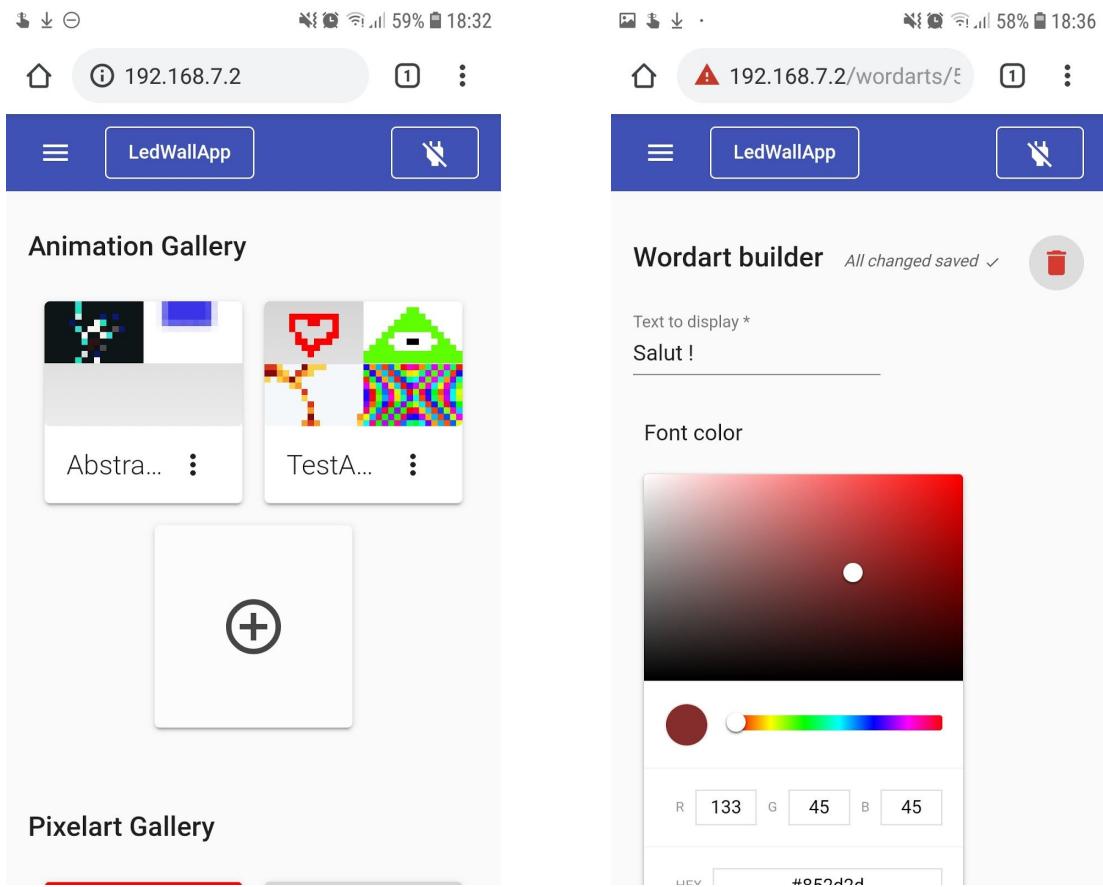
Animation builder

Saving...

Animation builder

All changed saved ✓

B. Démonstration sous format mobile



Nous pouvons observer que le site web est adapté au format mobile. Vous pouvez changer d'animation très facilement. Toutes les fonctionnalités sont incluses à part la création ou modification d'un pixel art qui n'est pas géré par ce type de format.

- Nginx

Au démarrage du projet, nous étions partis du principe que notre backend **nodeJS** allait servir absolument tout, y compris l'hébergement de nos pages web embarquées.

Cette solution avait bien des avantages. En effet, **expressJS**, notre framework de routage, nous mettait à disposition une solution très simple pour servir nos pages web. Nous nous sommes cependant rendus compte après des analyses approfondies de performances que cette solution était très gourmande en ressources CPU et mémoire. En effet, cela pouvait s'observer lors du chargement d'une page Web durant la lecture d'une animation.

Rapidement, l'expérience aidant, nous avons décidé de prendre avantage de notre solution Linux en installant le logiciel de service web **Nginx**. Cette solution,

mondialement utilisée, se prête tout aussi bien à l'installation sur notre carte. Étant développée par de véritables développeurs soucieux de la performance et de la sécurité, nous pouvons lui faire confiance.

Nginx étant aussi un standard de l'hébergement de pages webs, une pléthore de tutoriels et de documentations nous est offerte.

La configuration de nginx est fait comme ci-dessous :

```
1 server {
2     listen 80;
3
4     auth_basic      "Administrator's Area";
5     auth_basic_user_file /etc/nginx/.htpasswd;
6
7     server_name localhost;
8     root /home/user/Public/LedWallApp/front/dist/ledwall-app;
9     location / {
10        try_files $uri $uri/ /index.html;
11    }
12 }
```

Nous mettons à disposition le site sur le port par défaut : 80 à l'adresse localhost. L'accès est sécurisé par un système d'authentification utilisateur/mot de passe stocké dans le fichier .htpasswd.

Communication avec le MiddleWare

Version Framebuffer

On rappelle que le middleware est un logiciel réalisant l'adaptation bas niveau du pilotage du LED Wall. L'idée est que l'utilisateur dispose d'un simple fichier sur le système de fichier pour piloter le LED Wall.

Ainsi, au niveau utilisateur, il suffit d'**ouvrir** le fichier **/tmp/lwfb** y **écrire**, en écrasant les données, **les octets de couleurs** comme décrit dans la partie analyse puis **fermer** le fichier.

Compte-tenu de la nature **unidimensionnelle** d'un flux d'octets, il faut trouver une méthode pour décrire des données intrinsèquement multidimensionnelles sous cette forme. Fort heureusement, avec un minimum d'algèbre, et une grande quantité d'expérimentations, nous avons pu en déduire la solution décrite en pseudo-code ci-dessous.

```
for (var y = 0; y < data.height; y++)
{
    for (var x = 0; x < data.width; x++)
    {
        /* 2D coordinates to 1D array index */
```

```

        var idx = (data.width * y + x) << 2;
        frame_RGB.push(data.data[idx]);           //R
        frame_RGB.push(data.data[idx + 1]);       //G
        frame_RGB.push(data.data[idx + 2]);       //B
    }
}

```

Où **frame_RGB** représente notre buffer unidimensionnel. La “magie” opère sur la ligne en surbrillance rouge. En effet, c'est ici que nous calculons l'index auquel un pixel de coordonnées X et Y doit se situer dans le fichier **lwfb**.

Puisque nous ne disposions pas de LED Wall fonctionnel en début de projet, nous avons fait le choix de rapidement développer un simulateur permettant d'afficher ce framebuffer.

Codage d'un simulateur de LED Wall

Afin de pouvoir tester le site WEB et la génération correcte des images et animations par le back-end dans le framebuffer fichier, nous avons eu besoin d'un simulateur graphique sur PC.

Le site web ayant été développé en parallèle du matériel, nous avons dû trouver un moyen de simuler le LED Wall. Pour cela, nous avons conçu un logiciel venant s'interfacer sur le fichier framebuffer de la même façon que le middleware. Nous avons conçu deux versions, une première version en Python et une version Java optimisée par la suite. Le choix de ces langages est majoritairement dû aux contraintes de la compatibilité multi-OS. En effet, les développeurs WEB travaillent sous Linux et Windows, offrir un simulateur portable en JAVA et Python permet de rapidement effectuer des tests.



Figure : Simulateur de LED Wall (Python 3)

La première version fut réalisée en Python 3 et n'a nécessité aucune dépendance externe. Cependant, Python 3 ne dispose pas de fonctionnalités de surveillance de fichier cross plateforme native, ce qui signifie que nous devons

surveiller le timestamp du fichier pour décider de la mise à jour ou non de la fenêtre. Grâce à ce petit utilitaire, la validation des fonctionnalités de dessin et d'animation provenant du site ont pu être validées au fur et à mesure de leur développement, sans attendre la réalisation finale de la structure électronique de notre LED Wall.

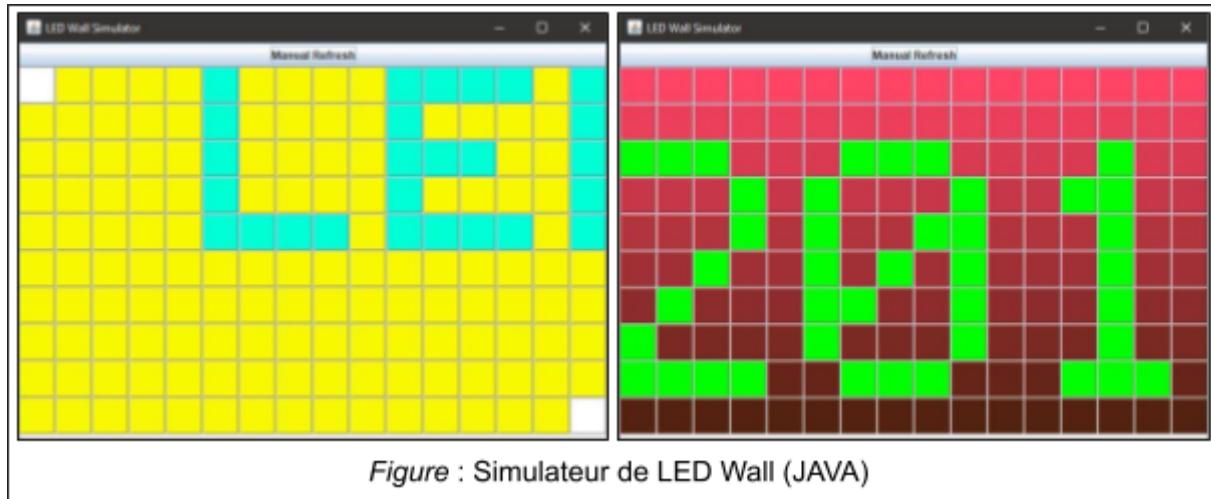


Figure : Simulateur de LED Wall (JAVA)

L'interface graphique est très simple, on affiche les 16 x 10 cases du LED Wall dans une fenêtre redimensionnable.

Grâce à l'API JAVA 7 on dispose de fonctions d'observation des modifications sur un fichier à travers le package **java.nio.file.***.

Ainsi, on peut enregistrer un déclenchement d'événements sur les modifications ou créations de fichiers dans un dossier donné du système de fichier. A l'ouverture du simulateur, une boîte de dialogue JFileChooser nous demande de choisir un répertoire à observer. On peut choisir n'importe quel dossier mais celui-ci devra contenir le fichier **lwfb** contenant le framebuffer du LED Wall. Le **WatchService** de l'API Java.NIO déclenche alors la lecture du fichier **/tmp/lwfb** à chaque fois qu'il est **ouvert → modifié → fermé** par un **autre processus**. Lorsque cet événement est détecté, on effectue un **décodage du framebuffer** et on lance un rafraîchissement de l'objet **LedWallPanel** affichant l'ensemble des **carrés (pixels)** du LED Wall simulé.

La **partie graphique** du simulateur est réalisée avec l'API **Swing de JAVA**. L'objet **LedWallPanel** est un objet héritant des attributs de **JPanel** pour lequel est surchargée la méthode **paint**. Ainsi on peut utiliser l'objet **graphics** passé de cette méthode pour réaliser le dessin des pixels du Led Wall. Un rafraîchissement se fait alors à chaque déformation de la fenêtre d'affichage ou bien lors de l'appel consécutif des méthodes **revalidate** et **repaint** héritées de l'objet JPanel.

Ainsi, si l'on compare le temps de développement de cet outil avec le temps qu'il nous a fait gagner, on peut sans trop s'avancer dire qu'il s'agissait d'un choix judicieux.

Version 1: P.O.C. architecture embarquée sur Raspberry PI 3B+

Mise en place du système Raspbian

Suite à notre analyse sur le changement de technologie, il nous a fallu montrer que celle-ci permettrait de répondre aux attentes de notre client.

Cette première de **preuve de concept** devait être réalisée rapidement afin de permettre d'avancer rapidement sur la suite et de la faire valider par notre client. Pour ce faire, nous avons réalisé un premier essai sur **une carte Raspberry PI 3B+**.

L'ensemble des étapes de configurations sont gérées à partir d'un script d'installation à lancer à la première utilisation.

Configuration du système Linux

Afin de pouvoir exploiter la carte, nous avons utilisé le système d'exploitation **Raspbian Stretch lite**. C'est une version modifiée de la distribution linux Debian avec seulement l'interface en ligne de commande.

Afin de communiquer avec la Raspberry, il faut embarquer un fichier permettant la communication ssh pour pouvoir piloter la carte depuis un ordinateur via un câble un RJ45. On pilote ensuite la Raspberry depuis un émulateur de console tel que **PuTTY**.

Le but est ensuite de configurer la carte afin que l'on puisse y accéder par **wifi**. On va alors modifier les configurations du réseau.

Configuration réseau

On commence par donner une adresse fixe à notre carte. Pour ce faire, on commence par installer les logiciels **dnsmasq** et **hostapd**.

On commence par modifier le fichier de configuration du dhcpcd (/etc/dhcpcd.conf) pour y entrer l'adresse ip souhaitée :

```
interface wlan0
    static ip_address=192.168.4.1/24
    nohook wpa_supplicant
```

On redémarre ensuite le **système dhcpcd** avec la commande suivante :

```
sudo systemctl restart dhcpcd
```

On va ensuite modifier la configuration du serveur DHCP grâce au logiciel dnsmasq. On commence par modifier le fichier de config du dnsmasq **/etc/dnsmasq.conf**.

On va utiliser l'interface sans fil wlan0 et on va fixer une adresse IP entre 192.168.4.2 et 192.168.4.20.

```
interface=wlan0
dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h
```

On précise ici les 24h pour que l'adresse ne change pas.

On relance ensuite le **système dnsmasq** avec la commande suivante :

```
sudo systemctl reload dnsmasq
```

On va ensuite créer un point d'accès, pour pouvoir y accéder par wifi. Pour ce faire, on va créer le fichier /etc/hostapd/hostapd.conf. On y spécifie l'ensemble des caractéristiques de notre point d'accès.

```
interface=wlan0
driver=n180211
ssid=LedWall2019
hw_mode=g
channel=7
wmm_enabled=0
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=*****
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

On précise ensuite au système où trouver notre fichier de configuration du point d'accès. On modifie alors le fichier /etc/default/hostapd en modifiant la ligne suivante :

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

Configuration du projet

Afin de pouvoir lancer le projet sur la carte, il faut installer les différents programmes et cloner le projet Git sur la carte.

Conclusion POC

Nos estimations sur les performances WS2812 se sont avérées correctes.

En contrepartie, nous avons été limité par les performances de la carte raspberry. Lors de la mise en place d'une animation à 24 fps, certaines LEDs n'étaient pas bien pilotées. Pour la suite, nous sommes passé sur une carte industrielle BeagleBone Black.

Version 2: Intégration système embarquée sur BeagleBone Black

Mise en place du système Debian

Configuration du système Linux

Pour des raisons de maintenabilité, on a choisi la dernière version de debian (9.8 IOT) à installer dans la BeagleBoard black.

Installation sur la emmc

La BeagleBone était un choix naturel vu qu'elle dispose d'une mémoire, emmc qui nous permet de se passer de la carte SD qui peut avec le temps lâcher et perdre tous les informations.

Pour installer la distribution il faut donc télécharger la version 9.8, ensuite graver l'image sur une carte sd. Une fois terminé, insérez la carte SD dans la BeagleBoard, le firmware devrait s'installer automatiquement. Les leds clignotent en chenillard durant l'installation, et l'arrêt de clignotement signifie la fin de l'installation, vous pouvez donc débrancher la carte et enlever la carte SD

La version de debian installée dans la carte est surchargée avec des fonctionnalités qu'on va désactiver dans la partie qui suit.

Test de la carte

Connectez la carte beaglebone, avec votre ordinateur à l'aide du câble USB, après quelques secondes, une interface RNDIS (Remote Network Driver Interface Specification) doit s'afficher dans votre gestionnaire de périphériques.

Cette interface représente un port Ethernet over USB, est vous permet de se connecter à la carte en SSH.

L'adresse IP par défaut 192.168.7.2, pour se connecter une session par défaut est accessible avec le nom d'utilisateur : **debian**, et le mot de passe : **temppwd**.

Préparation du système :

Cette procédure d'installation est disponible le dépôt Github ainsi que l'intégralité des fichiers nécessaires dans la rubrique **BeagleBoneBlackLinux** du dépôt.

Avant d'installer l'application LedWall, on va suivre les étapes de nettoyage suivantes :

1. Désactiver le port HDMI:

Exécutez la commande suivante pour ouvrir le fichier uEnv

```
sudo nano /boot/uEnv.txt
```

Pour désactiver le HDMI, il faut retirer le “ # ” devant les lignes suivantes :

```
disable_uboot_overlay_video=1  
disable_uboot_overlay_audio=1
```

Enregistrez vos modifications et quittez nano.

2. Charger /temp en RAM:

Exécutez la ligne suivante pour ouvrir le fichier **fstab**

```
sudo nano /etc/fstab
```

Ajoutez la ligne suivante dans le fichier

```
tmpfs /tmp tmpfs rw,size=16M,noexec,nodev,nosuid,noatime,mode=1777 0 0
```

Enregistrez vos modifications, fermez nano et redémarrez la carte.

```
sudo reboot
```

3. Supprimer le serveur Apache2

L'application LED Wall utilise NGINX comme serveur pour tourner, donc on supprime Apache en exécutant les commandes suivantes :

```
#Supprimer Apache2:  
sudo service apache2 stop  
sudo apt-get purge apache2  
sudo apt-get purge apache2-mpm-worker  
sudo apt-get purge apache2-utils  
sudo apt-get purge apache2.2-bin  
sudo apt-get purge apache2.2-common  
sudo apt-get autoremove #get rid of no-longer needed dependencies
```

```
whereis apache2
```

```
#supprimer tout les répertoires listés.  
sudo rm -rf /etc/apache2  
sudo rm -rf /usr/lib/apache2  
sudo rm -rf /usr/sbin/apache2  
sudo rm -rf /usr/share/apache2
```

4. Supprimer la page d'accueil

Une fois apache supprimé on supprime la page d'accueil par defaut de la BeagleBoard en exécutant les commandes qui suivent :

```
sudo systemctl stop cloud9.service          #stop working copy
sudo systemctl stop cloud9.socket           #stop working copy
sudo systemctl disable cloud9.service       #disable autorun
sudo systemctl disable cloud9.socket        #disable autorun
sudo rm -rf /var/lib/cloud9                #installed binaries and such
sudo rm -rf /opt/cloud9                   #source download and build directory
sudo rm /etc/default/cloud9               #environment variables
sudo rm /lib/systemd/system/cloud9.*      #systemd scripts
sudo systemctl daemon-reload              #restart/reload systemctl deamon
```

5. Supprimer BoneScript Library

BoneScript est une librairie qui permet de faire d'appel fonctions similaires aux fonctions arduino, qui facilite la prise en main de la carte. Cette fonctionnalité n'est pas nécessaire pour notre besoin on va la supprimer pour libérer le port 80 que cette fonctionnalité occupe :

```
sudo systemctl stop bonescript-autorun.service  #stop currently running copy
sudo systemctl stop bonescript.service
sudo systemctl stop bonescript.socket
sudo systemctl disable bonescript-autorun.service  #purge autorun scripts
sudo systemctl disable bonescript.service
sudo systemctl disable bonescript.socket
sudo rm /lib/systemd/system/bonescript*          #startup scripts
sudo rm -rf /usr/local/lib/node_modules/bonescript #binaries
sudo systemctl daemon-reload                      #restart/reload systemctl
deamon
```

ATTENTION

Ne pas toucher au contenu du dossier **/opt/source** vu qu'il contient des scripts utiles pour la gestion des GPIOs de la carte.

6. Installation de NGINX:

Comme évoqué avant l'application LedWall tourne sur un serveur NGINX, et pour l'installer il faut suivre les étapes suivantes :

En premier temps installer les dependances de nginx-full 1.10.3

```
sudo apt-get update
```

```
sudo apt-get install nginx
```

En second temps configurez la page par défaut de Nginx:

```
cd /etc/nginx/sites_available  
sudo nano default
```

Dans la partie serveur modifiez les lignes suivantes :

```
server{  
    listen 80;  
    listen [::]:80 ipv6only=on default_server;  
    # ... reste du fichier  
}
```

Enregistrez vos modifications, et fermez nano.

Arrêtez le service et redémarrez le pour prendre en compte la nouvelle configuration

```
#Arreter nginx  
sudo nginx -s stop  
  
#demarrer nginx  
sudo nginx
```

7. Installation de MongoDB

Pour enregistrer les images et wordart, l'application LedWall utilise MongoDB comme base de données.

MongoDB n'est pas compatible avec la dernière version de debian, donc on peut pas l'installer directement avec un paquet debian.

Pour l'installer en premier temps il faut copier les fichiers core et tools manuellement dans le système de fichiers de la beaglebone .

```
#extraire mongodb_stretch_3_0_14_core.zip  
#extraire mongodb_stretch_3_0_14_tools.zip  
#copier les binaires via SCP dans les répertoires suivant  
# mongo mongod mongos  
mkdir ~/mongodb  
mkdir ~/mongodb/core  
mkdir ~/mongodb/tools  
#Copier les binaires dans les dossiers correspondants  
chmod 755 ~/mongodb/core/*  
chmod 755 ~/mongodb/tools/*
```

```
#Copier les binaires vers /usr/local/bin pour y avoir accès dans le PATH  
cp ~/mongodb/core/* /usr/local/bin  
cp ~/mongodb/tools/* /usr/local/bin  
  
#Créer le dossier /data/db pour Mongo  
sudo mkdir /data  
sudo mkdir /data/db
```

Après exécutions de ces lignes de commandes, Mongo peut être tester.
On commence par démarrer la base de données

```
sudo mongod
```

Mongod doit s'initialiser et attendre sur le port 27017
Ouvrez une nouvelle sessions SSH et lancez mongo

```
sudo mongo
```

Le shell doit se connecter au serveur mongod.

8. Installation du programme WS2812b PRU

L'installation du WS2812b se fait en suivant les commandes suivantes:

```
cd /home/debian  
mkdir ~/PRU_WS2812  
cd ~/PRU_WS2812
```

Copiez les fichiers suivants en scp dans le dossier

```
AM335x_PRU.cmd Makefile resource_table_0.h setup.sh neo_pixel_pru_firmware.c  
README.txt  
examples/neo_blink_open.c examples/neo_rainbow_open.c  
examples/neo_rainbow.py
```

Après que la copie soit terminée, lancez les deux commandes suivantes :

```
source setup.sh  
make
```

9. Installation du LedWallAPP

Avec votre navigateur préféré (internet explorer peut causer le lancement d'une tête nucléaire du SNLE **Le Terrible** ! à vos risques et périls), téléchargez la dernière release du [LedWallAPP](#).

Créez un dossier LedWallApp et copiez les binaires dedans.

```
mkdir ~/LedWallApp  
  
#Apres la fin de la copie  
  
chmod 755 LedWall_Middleware
```

Téléchargez le [middleware](#) et placez le dans ~/LedWallApp .

Ensute il faut changer le fichier .conf, pour cela il faut exécuter les commandes suivantes :

```
cd /etc/nginx/sites_available/  
sudo touch ledwallapp.conf
```

Ecrivez la configuration suivante dans le fichier

```
server {  
    listen 80;  
    server_name localhost;  
    root /home/debian/LedWallApp/front/dist/ledwallapp;  
    location / {  
        try_files $uri $uri/ /index.html;  
    }  
}
```

You almost there ! Il faut maintenant ajouter le lien symbolique pour activer le Site LedWallApp en exécutant la commande suivante :

```
sudo ln -s /etc/nginx/sites-available/ledwallapp.conf  
/etc/nginx/sites-enabled/ || exit 0
```

Supprimez la page par défaut de nginx et relancez le .

```
sudo rm /etc/nginx/sites-enabled/default  
  
#Relancer nginx  
sudo nginx -s stop  
sudo nginx
```

10. Lancement des services

Tous les composants du LedWallAPP sont installées et pour les lancer, exécutez les instructions suivantes.

```
cd ~/PRU_WS2812  
source setup.sh
```

```
make  
sudo mongod  
sudo ~/LedWallApp/LedWall_Middleware  
node ~/LedWallApp/back/server.js  
sudo nginx
```

11. Automatisation des services

```
#Ajouter les scripts de demarrage systemd  
mkdir ~/LedWallApp/startup  
#Copier launchMiddleWare.sh ledwallapp-api.service  
ledwallapp-middleware.service mongodb.service  
#dans le dossier startup  
#Creer des liens symboliques des fichiers .service dans /lib/systemd/system/  
  
#Configurer le host ap Wifi pour la BB wireless !  
sudo nano /var/lib/connman/settings  
  
#Dans la rubrique [WIFI]  
#Tethering=true  
#Tethering.Identifier=DII-LedWall  
#Tethering.Passphrase=AxrCewDII  
#sources : https://diyevil.com/projects/beaglebone-wifi-access-point/
```

Programmation des Programmable Real-time Units

Nous vous avons parlé du choix d'un processeur hétérogène précédemment et qu'il est adapté à ce que nous cherchons à accomplir. On dispose de deux PRU, sur notre cible TI Sitara-AM3358. Nous allons voir comment nous les avons utilisés et comment nous avons créé un driver de LEDs WS2812b sur celui-ci.

Utilisation des PRU

Les PRU font parties intégrante du composant AM3358 de TI. Ces architectures **disposant coeurs temps réels** se sont développées de plus en plus et nous avons vu apparaître dans le noyaux Linux un framework permettant de les utiliser.

Ce framework s'appelle **remoteproc**. Il a été conçu pour unifier et simplifier la gestion des coeurs temps réels. Ainsi, sur la carte BeagleBone, on dispose de 3 coeurs temps réels, un cortex M3 dédié à la gestion de l'alimentation et deux PRU pour les utilisateurs. Les PRU ont été créés pour répondre à la demande fortement temps réels des systèmes industriels sur lesquels on souhaite désormais embarquer des IHM graphiques et des couches réseaux utilisés sur des PC.

Ainsi, avec les PRU, on dédie les coeurs A8 à l'OS Linux et aux IHM Web ou tactiles et on dialogues avec les PRU qui peuvent contrôler des machines ou dialoguer sur des réseaux de terrain industriels ayant des contraintes temps réelles.

Ce PRU est cadencé à 200Mhz, ce qui permet d'exécuter rapidement du code bare metal.

On utilise un PRU comme on utilise un microcontrôleur 32 bits. On peut l'utiliser avec ou sans RTOS avec ou non une librairie d'abstraction matérielle.

Il y a un excellent article écrit par TI expliquant le fonctionnement des PRU.

http://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components_PRU-ICSS_PRU_ICSSG.html

Nous allons aller à l'essentiel de l'utilisation du **driver Linux remoteproc** pour voir ce qu'il est possible de faire.

On peut donc accéder à un PRU à travers le système de fichier Linux:

On dispose d'un répertoire par processeur déporté :

/sys/class/remoteproc/remoteprocX

X=0 → Cortex M3 pour la gestion de l'énergie

X=1 → PRU 0

X=2 → PRU 1

On peut donc le contenu de ces dossier pour agir avec les PRU.

Dans le dossier **remoteproc1** on trouve les fichiers:

1. firmware → Contient le **nom** du binaire à charger sur le PRU.
2. state → **Etat** du PRU, allumé, éteint, déconnecté

On peut donc changer le **nom du fichier binaire** à exécuter à l'aide de la commande suivante :

```
echo 'am335x-pru0-fw' > /sys/class/remoteproc/remoteproc1/firmware  
echo 'am335x-pru1-fw' > /sys/class/remoteproc/remoteproc2/firmware
```

Il faut ensuite placer le binaire à exécuter sous le nom **am335x-pru0-fw** dans le dossier :

/lib/firmware/

Enfin, on peut concrètement renseigner le firmware binaire à charger par une simple copie:

```
cp myPRU0bin /lib/firmware/am335x-pru0-fw  
cp myPRU1bin /lib/firmware/am335x-pru1-fw
```

Désormais, le PRU dispose d'un binaire à exécuter. Pour lancer l'exécution du PRU, il suffit d'écrire **start** dans le fichier **state** du PRU à démarrer:

```
echo 'start' > /sys/class/remoteproc/remoteproc1/state
echo 'start' > /sys/class/remoteproc/remoteproc2/state
```

On peut aussi stopper l'exécution en écrivant **stop**.

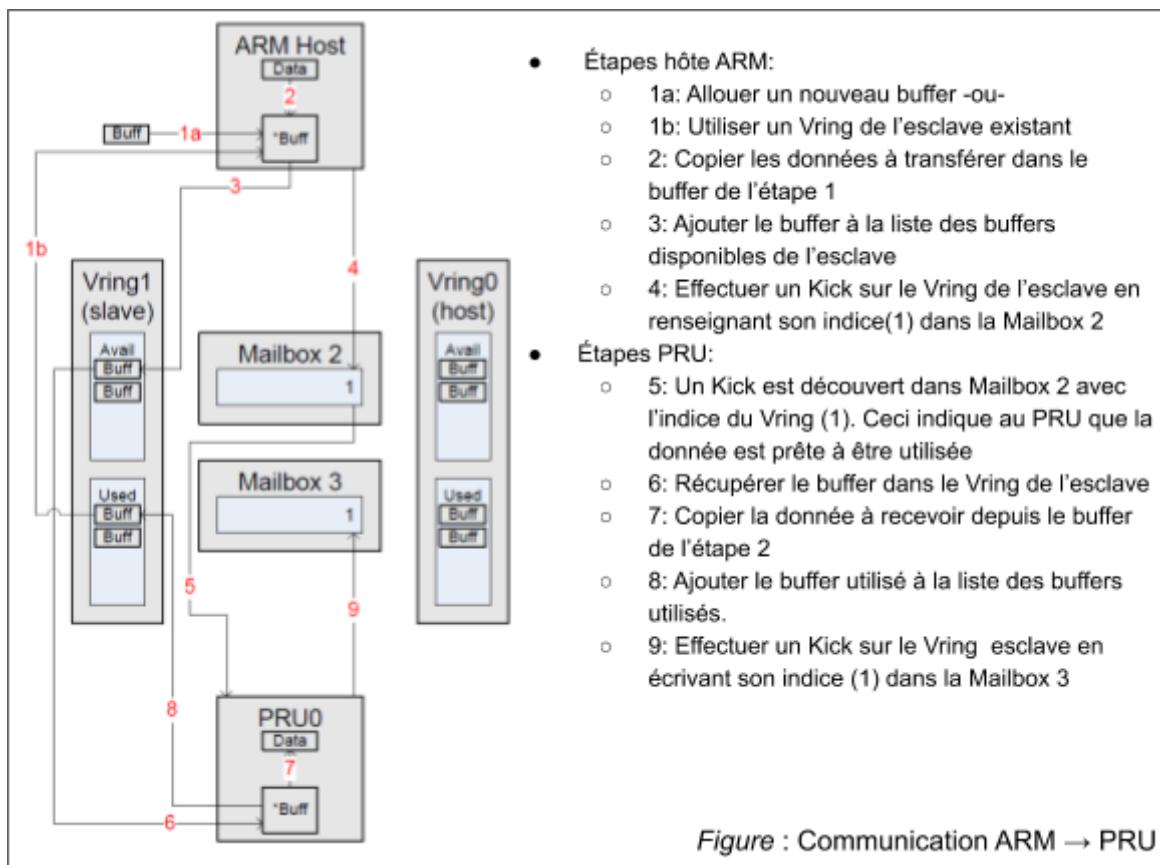
La procédure pour charger un binaire est la suivante:

1. STOP: echo 'stop' > /sys/class/remoteproc/remoteproc1/state
2. INSTALL: cp myPRU0bin /lib/firmware/am335x-pru1-fw
3. START: echo 'start' > /sys/class/remoteproc/remoteproc1/state

C'est tout ce qui est nous est nécessaire pour faire fonctionner un PRU.

Communication inter-processeurs

Le framework **remoteproc** fournit un dispositif de dialogue entre CPU et PRU à l'aide d'une mémoire partagée et un système de **mailbox**. Les détails du fonctionnement sont indiqués sur le site de TI et reprises ci-dessous.



Heureusement, toutes ces actions sont prises en charges du côté Linux grâce à un driver **VirtIO**. Ainsi, un fichier **/dev/rpmsg_pru30** est créé lorsque le programme PRU fait appel à la fonction d'initialisation de création du flux inter-processeurs (**pru_rpmsg_init** et **pru_rpmsg_channel**). Les nom du fichier

ainsi que le numéro de canal rpmsg est renseigné dans le programme PRU lors des appels des fonction init et channel.

On peut ensuite aisément envoyer des paquets de données à l'aide d'une simple écriture dans le fichier.

Et c'est tout pour la partie Linux ! Il suffit d'ouvrir le fichier **/dev/rpmsg_pru30** pour dialoguer avec le PRU 0 en effectuant un **write** dans ce fichier.

Cependant, le fichier **/dev/rpmsg_pru30** n'est instantié dans le système de fichier que si le PRU fait appel aux fonctions nécessaires dans la librairie "**pru_rpmsg.h**".

Du côté PRU, il y a cependant un peu plus de travail mais nous disposons d'une librairie conçue pour mettre à profit le driver.

Pour construire notre application, nous nous sommes basés sur l'excellent dépôt Github : <https://github.com/MarkAYoder/PRUCookbook>

On y trouve l'ensemble des exemples permettant de réaliser la communication avec le PRU. Nous avons adapté le code créé par le développeur pour mettre en place le dialogue entre ARM et PRU et le pilotage des LEDs.

<https://github.com/MarkAYoder/PRUCookbook/blob/master/docs/05blocks/code/neo4.c>

Nous avons allégé et complété le protocole de communication utilisé par le créateur original et réalisés quelques optimisations de code.

Il avons pensé le PRU comme un pilote d'une chaîne de LEDs d'une taille maximale de 200 LEDs.

Les fonctions sont les suivantes:

1. Configurer la couleur d'une LED dans la chaîne
2. Configurer une correction de couleur à appliquer sur toute la chaîne.
3. Rafraîchir les LEDs (envoyer les signaux)

Les dialogues inter-processeurs suivant ont été mis en places:

| Fonction | Trame | Exemple: Code C correspondant |
|---|---|---|
| Configurer le couleur d'une LED dans la chaîne | "Id Color\n" Id = id LED (0 à 199) Color = couleur 24 bits RGB en Hexadécimal | uint32_t colorRGB=0xFF3322; int16_t idled = 0; dprintf(fdrpmsg,"%d %06X\n",idled,colorRGB); |
| Configurer une correction de couleur à appliquer sur toute la chaîne. | "-2 Corr\n" Corr = correction 24 bits RGB en Hexadécimal | //Rouge 100% Vert 69% Bleu 65% uint32_t corrRGB=0xFFB0A5; dprintf(fdrpmsg,"-2 %06X\n",corrRGB); |

| | | |
|--|--|---|
| Rafraîchir les LEDs (envoyer les signaux) | <pre>"-1 NbLeds\n" NbLeds = nb de LED à rafraîchir depuis le début de la chaîne en Hexadécimal</pre> | <pre>#define NB_LEDS 2 //Première LED en rouge dprintf(fdrpmmsg,"%d %06X\n",0,0xFF0000); //Deuxième LED en vert dprintf(fdrpmmsg,"%d %06X\n",1,0x00FF00); //Rafraîchir les deux LEDs dprintf(fdrpmmsg,"-1 %06X\n",NB_LEDS);</pre> |
|--|--|---|

Attention cependant, lors de l'utilisation du driver **/dev/rpmsg_pru30**, il est nécessaire de travailler avec les appels système **open**, **write**, **read** et **close** pour éviter d'être sujet aux mécanismes de bufferisations de la librairie File (**fopen,fprintf,fread...**). Ainsi il faut ouvrir le fichier de la manière suivante:

```
int fdrpmmsg = open("/dev/rpmsg_pru30",O_WRONLY);
```

On sait désormais comment communiquer avec le PRU en utilisant notre protocole conçu pour l'occasion. Nous allons désormais discuter de la génération de signaux pour les WS2812b à l'aide du PRU.

Installation et démarrage automatique du PRU

Afin de faciliter la phase de compilation, démarrage et gestion du programme PRU, nous avons écrit un makefile qui est lancé automatiquement au démarrage de la carte Linux. Le binaire du PRU est compilé grâce au compilateur C propriétaire fourni par TI dans la distribution Debian Stretch IOT. Le programme PRU étant très léger, la compilation est réalisée en quelques seconde.

Cependant, cette tâche est réalisée une seule fois et le binaire est conservé après compilation et est réutilisé à chaque chargement du programme des PRU.

Le programme PRU est constitué d'un fichier **.c**, et deux fichiers **ressource_table_0.h**, **AM335x_PRU.cmd** utiles au pour l'éditeur de liens.

Le Makefile réalise alors les étapes de configuration **d'arrêt du PRU, chargement du firmware et démarrage du PRU**. On fournit en supplément un script bash permettant de configurer automatiquement la GPIO à utiliser pour la génération de signaux.

On utilise alors un script **setup.sh** réalisant l'appel du programme **config-pin** expliqué ci-dessous et configure les **variables d'environnement** (numéro de PRU et fichier .c à compiler) nécessaire à la compilation via le Makefile. Ensuite le **Makefile** réalise la compilation et le chargement du programme.

Il suffit donc de faire les commandes suivantes pour charger le programme:

```
source setup.sh
make
```

Génération de signaux WS2812b

Pour pouvoir piloter la GPIO de sortie à l'aide du PRU, il faut configurer le multiplexeur afin de router les actions du PRU sur une GPIO accessible sur la carte.

Pour ce faire, nous disposons du programme **config_pin** sur la distribution **Debian** de la Beaglebone permettant d'affecter une ou plusieurs pins aux PRU.

En effet, dans notre cas, nous utilisons la broche **P9_29** du connecteur **P9** de la BeagleBone Black comme sortie de signal pour la commande des LEDs.

Cette broche étant une sortie du PRU, il faut la configurer en **prout** pour que les registres des PRU agissent sur celle-ci.

Cette action est réalisée à l'aide de la commande :

config-pin P9_29 prout

Il est possible de piloter et régler le niveau de cette pin grâce au **bit 1 du registre R30** du PRU0 (Définition des pin du BeagleBone Black :

<https://vadl.github.io/images/bbb/P9Header.png>). Ainsi, en écrivant ce bit à 1, la broche passe à +3.3V et en écrivant 0 la broche passe à 0V.

Pour la génération des signaux de contrôle des LEDs, on a utilisé la technique du **bit-banging** très utilisé sur les cartes Arduino pour générer les signaux. On génère les signaux **logiciellement** par des boucles d'attentes bien calibrées dépendant de la fréquence du CPU.

Le PRU étant dédié uniquement à cette tâche, les timings peuvent être très précis et nous n'utilisons pas d'interruptions pouvant allonger les temps et créer des glitches dans les signaux.

L'algorithme d'envoi d'un bit est alors le suivant:

EnvoyerBit(bit)

Si bit = 1 Alors

```
    WS2812b_DATA ← 1  
    Attendre(600ns)  
    WS2812b_DATA ← 0  
    Attendre(600ns)
```

Sinon

```
    WS2812b_DATA ← 1  
    Attendre(300ns)  
    WS2812b_DATA ← 0  
    Attendre(900ns)
```

FinSi

Ensuite, il suffit d'exécuter la procédure d'envoi d'un bit pour les 24 bits de couleurs.

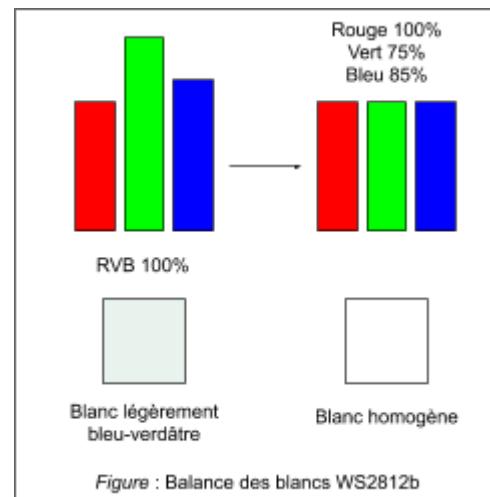
On a vu précédemment que la couleur doit être arrangée dans le sens GRB pour être accepté par les LEDs WS2812b. Les 3 octets de couleurs sont transmis dans le sens MSB first c'est à dire du bit 7 en premier au bit 0 en dernier.

Un point sur la correction de couleur :

En fonction du format dans lequel on achète les LED WS2812b, c'est à dire sous forme de ruban de LED ou de guirlande, il est parfois nécessaire d'appliquer une correction de couleur. En effet, lorsque l'on demande 100% de puissance sur chaque couleurs à la LED WS2812b, la teinte obtenue est un blanc teinté de vert et de bleu du à la balance des couleurs étant différente par nature. Pour corriger cette couleur, on peut utiliser une algorithme de correction.

L'idée est d'aligner les intensités des couleurs à la couleur dont la puissance est la plus faible (Cf schéma ci-après). Pour cela, on limite la valeur maximale du vert et du bleu d'une fraction donnée de la puissance maximale.

Par exemple, si on veut **100% de rouge, 69% de vert et 65% de bleu** pour faire du blanc on appliquera la correction avec les valeurs **0xFFB0A5**.



Le calcul des couleurs ROUGE, BLEU, VERT à affichés sur les LEDs à partir d'une couleur demandée rouge, vert, bleu est le suivant:

$$ROUGE = rouge \times FF / FF$$

$$BLEU = bleu \times B0 / FF$$

$$VERT = vert \times A5 / FF$$

Les calculs sont effectués avec les valeurs hexadécimales. L'utilisation de la correction de couleurs limite la puissance lumineuse globale de la LED mais permet une meilleure fidélité des couleurs.

Codage du MiddleWare

Implémentation du framebuffer fichier

Événement inotify

Nous utilisons le mécanisme inotify du noyau linux pour détecter la modification du framebuffer.

Le principe de ce mécanisme est le suivant : On s'abonne aux événements du type IN_CLOSE_WRITE pour le fichier framebuffer situé à /tmp/lwfb, puis on attend d'en recevoir un. Pendant ce temps là, le processus du middleware est bloqué. Il n'y a donc pas d'attente active. Le processus sera de nouveau actif dès lors que le fichier framebuffer sera fermé suite à une écriture dessus.

Donc, lorsque par exemple, le back écrit une nouvelle frame dans le framebuffer, le middleware se réveille pour transmettre la frame.

Structure du fichier

Nous avons construit le fichier framebuffer de la façon la plus simple. Chaque pixel est représenté par 3 octet qui représentent respectivement le canal rouge, vert et bleu du pixel. On répète ces trois octets autant de fois que de led du ledwall. Donc la taille en octet de notre framebuffer pour notre mur de 16 par 10 est de $16*10*3 = 480$ octets.

Adaptation bas niveau pour la génération de signaux

Librairie pour WS281x sur Raspberry Pi

Lorsque nous étions en expérimentation avec le Raspberry Pi, nous avions utilisés une librairie pour WS281x¹. Elle utilise une sortie PWM en coordination avec un canal DMA². De cette façon, il est possible de générer le signal qui pilote l'ensemble des leds. Même avec l'utilisation du DMA, le processeur était utilisé (5 à 10 %). En effet, le processeur devait mettre à jour le canal DMA à chaque bit transmis.

Dialogue inter-processeurs sur BeagleBone Black

Comme la BeagleBone Black possède des PRU³, nous en utilisons un pour la génération du signal de pilotage des leds. le protocole qui est utilisé pour la communication avec le PRU est celui défini plus haut dans la section [Communication inter-processeurs](#).

¹ https://github.com/jgarff/rpi_ws281x

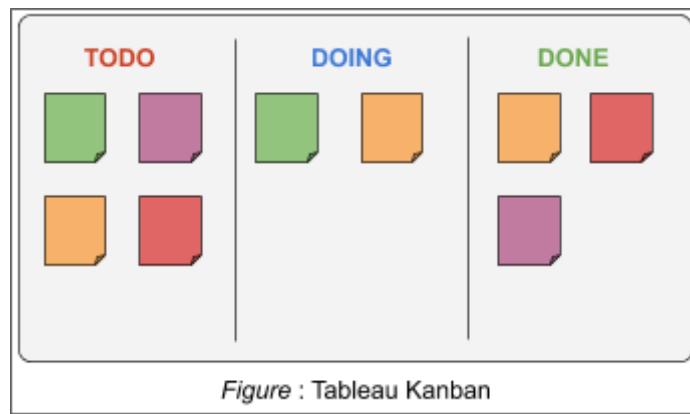
² Direct Memory access

³ Programmable real-time unit

Gestion de projet

Kanban et Trello

Pour gérer les ressources, l'assignation des tâches et suivre l'avancement des tâches au sein des équipes, nous nous sommes inspiré des méthodes agiles et notamment la méthode Kanban. Dans la méthode Kanban, chaque membre de l'équipe voit affecter une tâche dans la liste des tâches à faire. Ainsi, chacun peut indexer sa progression en faisant passer ses tâches d'une colonne à l'autre du tableau.



Pour que le suivi soit régulier et centralisé, nous avons implémenté ce tableau sur le logiciel en ligne **Trello**.

Nous avons utilisé ce logiciel pour publier les **ordres du jours des réunions** organisés avec les clients et pour les réunions internes. Les compte rendus et notes de réunions sont alors disponible dans la tuile Trello de la réunion.

Pour préparer au mieux nos réunions, nous avons eu recours aux logiciels **Xmind Zen** et **Coggle.it** pour créer des cartes mentales et réaliser nos séances de brainstorming.

Répartition des tâches

Pour la réalisation de ce projet, nous avons créé des équipes de travail en cherchant à affecter les ressources aux tâches en fonction des compétences.

Équipe de montage matériel

Nous avons eu besoin de toutes les ressources en début de projet pour la production des LEDs et le montage sur le LED Wall.

Ainsi, nous avons mobilisé les **6 ressources du projet sur la chaîne de production** des LEDs en début de projet pendant la semaine 50 et 51. Ensuite, il a fallu apporter des modifications à la structure du mur en raison des travaux de la bibliothèque.

Six personnes n'ont pas été de trop pour déplacer le mur, nettoyer les vitres, appliquer les nouveaux calques et réaliser le câblage des LEDs fraîchement produites. Grâce à cette force de frappe, nous avons réussi à terminer le câblage du mur de LED pour la mi-janvier.

Avec le complément de l'équipe embarquée bas niveau, il a été possible d'interfacer une carte Arduino pour **générer des motifs à présenter pendant la journée portes-ouvertes de l'école !** Cette présentation a eu lieu pendant la semaine 9 de 2019.

L'équipe composée de :

- LOCHE Jérémy
- THOMAS Louis
- RICHARD Arthur
- SIMMOU Soufiane
- LE NEL Corentin
- DOROTHÉE Corentin

Équipe de développement Web

Il nous a fallu des **ressources compétentes** pour le développement de la solution **WEB** aussi bien au niveau front-end qu'au niveau back-end. Pour cette tâche, nous avons assignés des ressources doués en la matière. **Louis THOMAS travaille au quotidien** dans le milieu et a été un **excellent référent** en terme de technologies et techniques d'implémentation. **Arthur RICHARD** a utilisé les **technologies WEB** employé durant ce projet à mainte reprise et est très à l'aise avec le **JavaScript**. Nous avions ici un excellent duo pour accomplir la réalisation du back-end et front-end. Ils se sont avérés être extrêmement efficace dans leur travail.

Ainsi, dès la soutenance de mi-parcours (semaine 16), nous avions déjà de très bonnes bases pour établir une architecture web complète.

L'équipe composée de :

- THOMAS Louis
- RICHARD Arthur

Équipe intégration système

Lors de la mise en place des systèmes sur nos **cartes Linux**, il a fallu disposer de ressources capable de réaliser l'intégralité de l'installation et la configuration des cibles. **Corentin LE NEL** et **Soufiane SIMMOU** travaillant régulièrement sur **l'OS Linux** ont été à même de réaliser cette tâche. Leur rôle a été primordiale durant **l'intégration de la solution** et le test sur les cartes Linux (Raspberry PI et BeagleBoard). Installer les OS, s'assurer d'installer les versions correctes des dépendances, automatiser le démarrage des services et **déployer les versions de programme** a été très efficacement réalisé par ce binôme.

L'équipe composée de :

- Corentin LE NEL
- Soufiane SIMMOU

Équipe développement embarqué bas niveau

Enfin, nous avons eu besoin de deux ressources compétentes en **développement bas-niveau et en électronique** pour réaliser le pilotage des LEDs et les test électriques. Sur cette tâche, **Jérémy LOCHE** et **Corentin DOROTHEE** ont réalisés les **programmes de pilotages bas-niveau**. Jérémy ayant développé un *driver de LED WS2812b basé sur un FPGA* pendant les cours de *programmation HDL* est devenu un spécialiste de ce type de LEDs. Ils travaillent tous deux sur des cibles embarqués et produisent **des programmes en C au quotidien en entreprise**. Corentin, étant très familier avec le monde Linux, a développé le Middleware très efficacement. De même, Jérémy a su produire les **programme bas niveau** installé dans les **PRU** de la carte BeagleBoard.

L'équipe composée de :

- Corentin DOROTHÉE
- Jérémy LOCHE

Communication en équipe

La communication a été un point essentiel pour la gestion de ce projet. Nous avons utilisé le service de messagerie instantanée **Messenger** en créant un groupe de discussion **LED WALL - DII4**. Ce service nous a permis de nous passer des mails pour la plupart des discussion internes. Nous avons aussi réalisé plusieurs vidéoconférences pour faire des réunions d'avancement grâce à cet outil.

Gdrive et Github

Pour le partage de données générales et la rédaction du rapport de projet, nous avons utilisé la suite Google avec notamment **Google Drive, Docs et Slide**.

Un point très intéressant cependant est la méthode de **gestion de version** et **l'intégration continue** mise en place sur ce projet.

GitFlow

En effet, afin de bien nous organiser dans le code, nous avons utiliser le modèle de branche **GitFlow** qui est adapté à la collaboration.

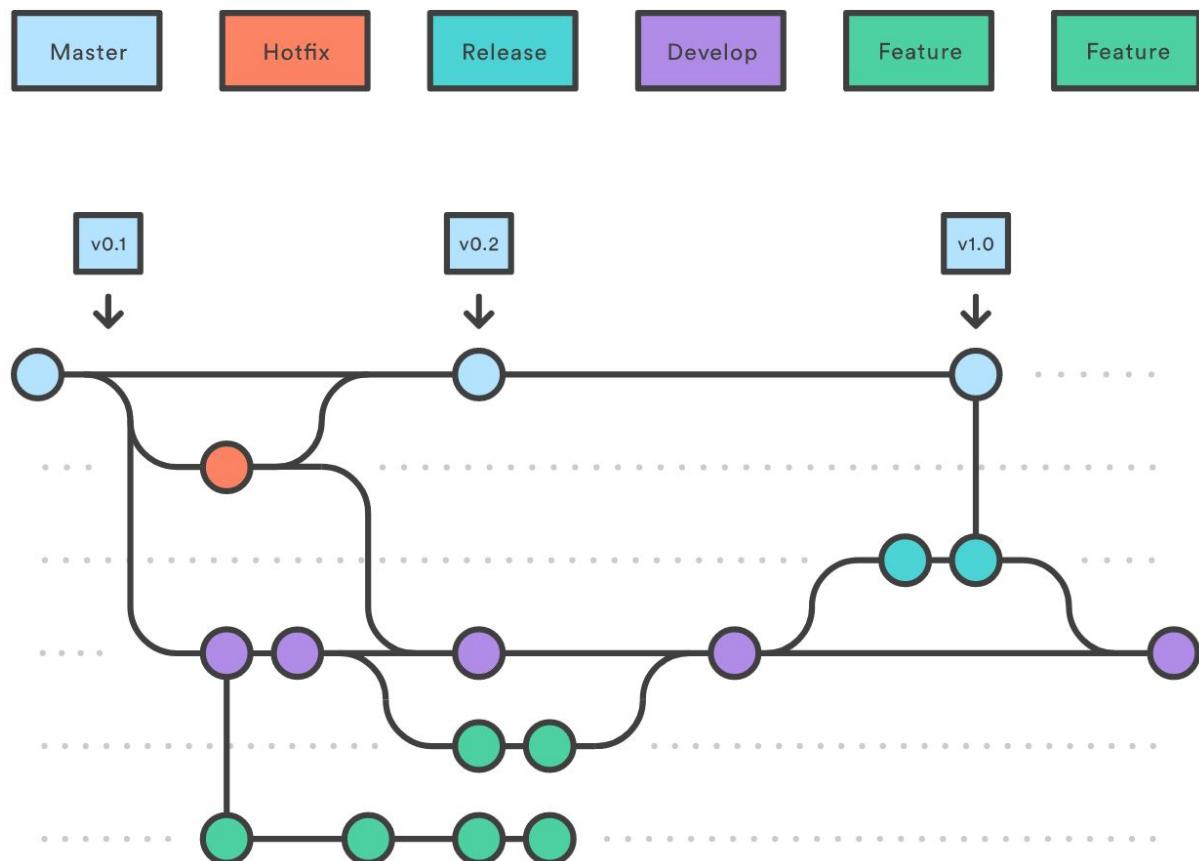


Figure : Image explicative du git flow

GitFlow présente plusieurs branches :

- Master : branche de production
- Develop : branche de pré-production
- Release : branche créée avant une production programmé. Seuls les correctifs de bugs, la génération de documentation et d'autres tâches orientées version doivent figurer dans cette branche
- Hotfix : branche Master créé pour une correction rapide d'un bug en production. Merge ensuite vers Master et Develop
- Feature : branche de Develop pour une nouvelle fonctionnalité

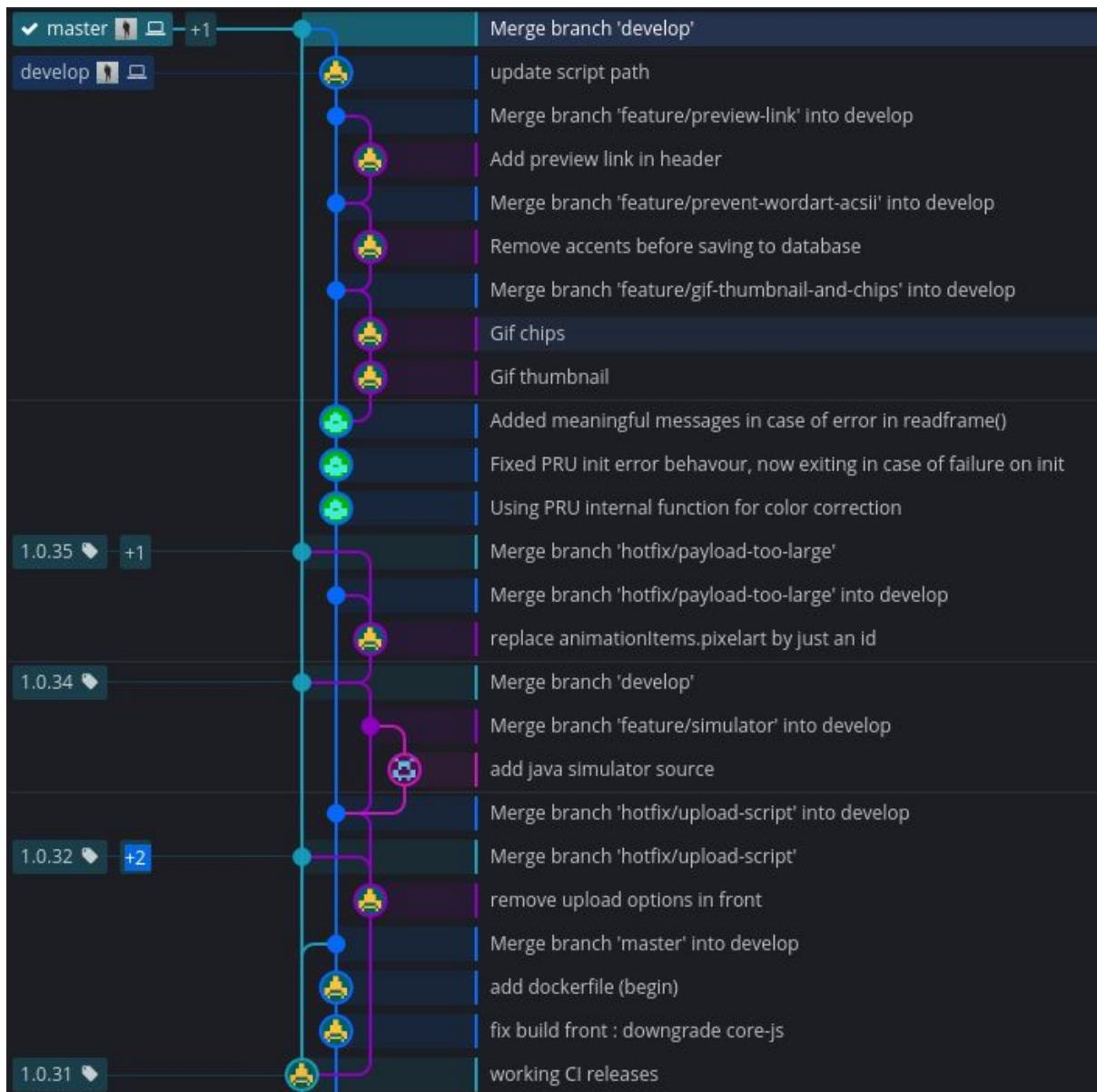


Figure : Screenshot du git tree

Intégration continue

L'intégration continue permet de préparer le site web à la production sans avoir besoin de builder manuellement. Pour cela, nous avons utilisé le service Travis qui se charge de déclencher un événement lors d'un "push" sur la branche "Master".

Par la suite, il lance une machine virtuelle venant cloner le projet, construire les fichiers de production, les publier sur Github release et affecter une nouvelle version au projet.

The screenshot shows a Travis CI build log for a project named 'louisthomasp / LedWallApp'. The build status is 'build passing'. The log details a merge from branch 'hotfix/cors-config' into 'master'. It includes commit information (commit cba8620, compare dec39f7..cba8620), a branch master, and a user Louis. The build ran for 2 min 47 sec, completed 3 days ago. A 'Restart job' button is visible.

Screenshot Travis build

The screenshot shows a GitHub release page for version 1.0.35. The release was made by louisthomasp 17 hours ago, merging the 'hotfix/payload-too-large' branch. It includes five assets: back.zip (13.9 MB), front.zip (21.6 MB), LedWall_Middleware (68.4 KB), Source code (zip), and Source code (tar.gz).

Screenshot de github release

Conclusion

Après 6 mois de dur labeur, nous avons réussi à fournir l'ensemble des éléments fonctionnels attendus sur ce projet. Le mur est intégralement câblé et fonctionnel et la procédure de fabrication est complète. Nous avons fourni un ensemble composé d'un service web complet allant de la base de donnée jusqu'à la page de contrôle.

L'ensemble est cohérent et opérationnel. Bien que l'avenir du projet ait été menacé à de multiples reprises, nous avons su garder la tête froide et mettre en œuvre les moyens nécessaires à l'achèvement du projet.

Le travail s'est bien étalé sur l'ensemble de la période et les équipes ont fourni un excellent effort.

Nous sommes fier du travail accompli. Nous avons mis à l'épreuve l'ensemble des connaissances abordées en cours et bien plus encore. Les compétences de chacun nous ont permis de balayer une importante fraction du spectre de l'informatique industrielle. L'utilisation des dernières technologies en terme de matériel et d'intégration continue montrent que ce projet est l'air du temps.

Nous terminerons ce rapport en évoquant d'intéressantes perspectives d'évolution pour ce projet qui pourraient donner lieu à de nouveaux projets étudiants.

Perspectives d'améliorations

- Ajouter un Websocket pour la page controller, afin d'interagir plus rapidement avec les scripts python.
- Ajouter des Words Art et scripts python aux animations.
- Ajouter l'enregistrement automatique pendant l'édition d'un Pixel Art
- Afficher un logo de chargement.
- Affichage en temps réel lors de l'édition d'un Pixel Art.
- Redimensionnement automatique en 16x10 lors de l'importation d'un Pixel Art ou gif.
- Ajouter un suivi temps réel du LED Wall sur le site web afin de connaître son état (animation en cours, motif affiché...).
- Implémenter un unix socket pour la relation entre le middleware et le back end nodejs (nouveau type de communication inter-process).
- Mise à jour des logiciels automatique lorsqu'il y a une nouvelle release (ex : package debian)
- Afficher une preview du World Art lors de sa création
- Générer un Pixel Art thumbnail automatique lors de l'importation d'un fichier python
- Informer l'utilisateur de l'état en cours d'un script python (est-il en erreur ?)
- Afficher l'animation gif dans la galerie

Annexes

- Projet github (sources et releases Site Web, Simulateur, PRU, Systemd, Middleware...) : <https://github.com/louisthomasp/LedWallApp>

DII4 - Projet Collectif - LED Wall

