Rapport projet n°33: Maquette Sas et programmation en scratch

Mously DIAW
Antoine MAILLEFERT
Hamza MABROUK
Louis LERBOURG

Responsable : Mathias Voisin-Fradin

sommaire

Déroulement du projet....1
Avant-propos technique....1
Le contexte initial....1
Le cahier des charges....2
Les choix et les solutions envisagées....3
Résultats....7

Description technique de la réalisation....8
Les connexions entre les différentes entités....8
Le codage logiciel....9
L'électronique de la maquette....12
Guide pratique d'utilisation et de programmation de la maquette....14
Exemples d'exercices à caractère pédagogique....16

La continuité du projet....16
Conseils pour la prise en main....17
Finalisation et corrections de notre avancement....18
Ajouter de nouvelles fonctionnalités....19
Une version industrialisable....20

Bilan financier....20
Coûts et frais des composants électroniques:....20
Coûts estimés du prototype de la maquette:....21

Gestion du groupe....21
Retours d'expériences individuels....21
Analyse critique....23

Diagrammes de Gantt....23
Diagramme initial....23
Diagramme révisé à la semaine 13....23

Annexes....24

Index des illustrations....28

Déroulement du projet

Avant-propos technique

Avant toute présentation de notre projet, voici une brève liste des technologies que nous avons employées, et dont il sera question au cours de ce rapport. Il peut être utile de se renseigner à propos de ces notions pour une bonne compréhension par la suite.

- Raspberry pi
- Arduino
- Scratch
- Python
- Javascript
- Kicad

Le contexte initial

Le projet n°33 intitulé maquette SAS s'effectue dans la continuité du travail d'un groupe de l'année précédente. L'objectif était alors de transformer une ancienne maquette de TP de l'ense3 en support pédagogique à la programmation en Scratch. Nous avons donc récupéré une maquette sur laquelle on trouve plusieurs composants (une porte, des capteurs, des boutons, un digicode, etc.) qui ont pour but de simuler un sas bancaire, où pourrait se dérouler un scénario tel que le suivant : Une personne se présente devant le sas, est détectée, la porte s'ouvre, etc...

L'année passée, un montage électronique avait été réalisé, et la majorité des composants étaient contrôlables grâce à un code python. Tout le potentiel des fonctionnalités n'était cependant pas exploité, et le contrôle de la maquette depuis scratch n'avait pas abouti.

Dans un premier temps, nous avons dû nous approprier le travail de l'année passée et acquérir les connaissances nécessaires à l'utilisation des outils indispensables pour mener le projet à bien.

Notre objectif cette année était particulièrement la recherche et la mise en place de solutions pour le contrôle du prototype "maquette sas" depuis le logiciel scratch.

Comme nous le verrons par la suite, résoudre ce problème représente un réel défi, puisque scratch est un logiciel peu orienté hardware, et qui communique difficilement avec l'extérieur.

Il fallait aussi se placer dans un contexte de continuité de ce projet au-delà de notre période de travail. Fournir aux années suivantes la documentation pour pouvoir encore améliorer le prototype, et pourquoi pas leur donner les moyens d'en concevoir une version industrialisable. Nous devions réfléchir à la production de documents à l'intention des professeurs et des élèves qui voudraient tester notre maquette.
L'aboutissement de ce projet sur le long terme serait donc que la maquette sas puisse être un support pédagogique dans lequel des établissements scolaires pourraient investir dans le cadre de travaux pratique avec leurs élèves.

Le cahier des charges

Le point de départ

Voici l'état de la maquette, récupérée de l'année précédente, et après vérification des connexions entre les différents éléments.

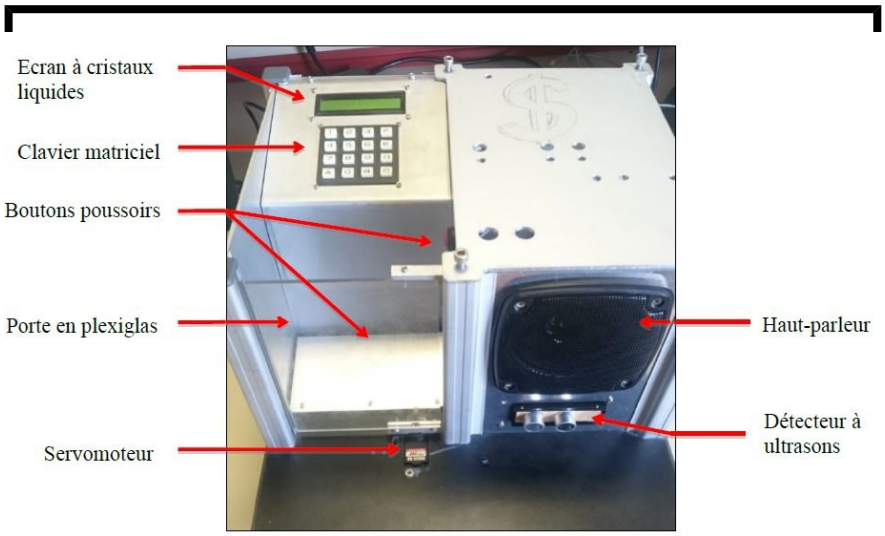


Illustration 2: maquette et composants

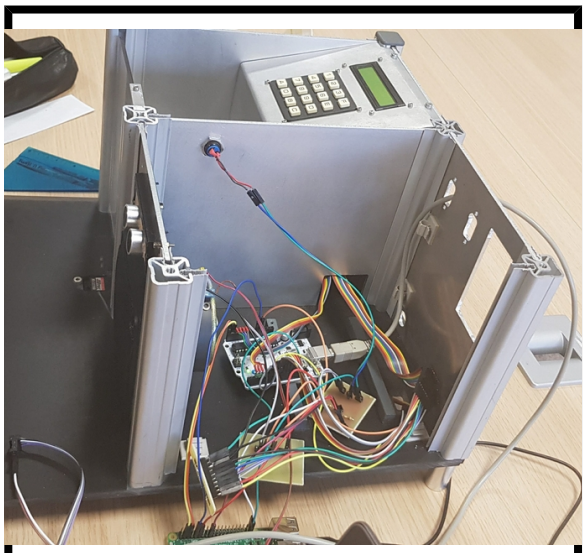


Illustration 1: maquette vue intérieure

Un programme python sur le raspberry pi commande un arduino via liaison i2c, et celui-ci commande la majorité des composants. Les points suivants étaient à corriger :

- Pas de contrôle des éléments depuis scratch
- Pas d'amplificateur audio sur le haut-parleur
- I2c entre le raspberry pi et l'arduino non-fonctionnel dans le sens arduino->raspberry.
- Pilotage de la maquette avec peu de fonctionnalités.
- Montage électronique peu fiable, et sujet aux faux contacts.

Au début du projet, nous avons donc fixé des objectifs de réalisation précis quant au livrable que nous allions avoir à produire.

L'interface scratch

Le plus important du projet se trouve bien ici. Nous devions mettre en place une solution afin de pouvoir piloter les différents éléments la maquette depuis scratch. Les critères d'aboutissement que nous nous sommes fixé étaient les suivants:

- La maquette est programmable depuis scratch.
- Le logiciel scratch tourne depuis un raspberry pi interne à la maquette.
- L'utilisateur peut programmer sur le raspberry, qui sera branché à un écran, un clavier et une souris.
- Depuis scratch, on peut ouvrir et fermer **la porte** du sas en contrôlant le servomoteur.
- Depuis scratch, on peut faire la mesure de l'état du **bouton rouge** (appuyé ou relâché).
- Depuis scratch, on peut faire la mesure de l'état de la **plaque de pression** (appuyé ou relâché).
- Depuis scratch, on peut afficher du texte sur les deux niveaux de l'**écran LCD**.

- Depuis scratch, on peut récupérer un code entré sur le **clavier matriciel**.
- Depuis scratch, on peut récupérer la mesure du **détecteur de distance**.
- Depuis scratch, on peut jouer du son sur le **haut-parleur** de la maquette.
- L'interface du sas proposée à l'utilisateur scratch doit pouvoir se mêler du mieux possible aux autres fonctions proposées par scratch.
- L'interface du sas proposée à l'utilisateur scratch doit masquer au mieux les contraintes techniques liées à l'électronique et au transfert de l'information.
- L'interface du sas proposée à l'utilisateur scratch doit être la plus intuitive possible, tout en laissant des libertés à l'utilisateur.
- Un élève qui a une connaissance de scratch peut aisément programmer la maquette.
- Un scénario type "sas bancaire" est programmé sous scratch, comme illustration de l'ensemble des possibilités du prototype.
- Un "guide de l'utilisateur" est rédigé.

L'amélioration matérielle

Nous avions des difficultés au début du projet à évaluer le temps qu'il allait nous rester une fois les objectifs énoncés ci-dessus accomplis. En effet, un premier gros travail qui nous attendait tenait de la **recherche** et de l'expérimentation. Nous ne souhaitions pas limiter ces deux étapes dans le temps, afin de favoriser en priorité la création d'une interface scratch fonctionnelle et réfléchie. C'est pourquoi l'amélioration matérielle de la maquette se voulait au départ comme un objectif secondaire. Ces objectifs étaient les suivants :

- Une solution pour améliorer la stabilité des connexions est mise en place (par exemple un shield : une plaque pcb qui prévoit toute les connexions, et sur laquelle on vient simplement ficher les composants et les cartes).
- Le haut-parleur est adapté en impédance à la sortie jack du raspberry pi grâce à un amplificateur audio.
- Un document technique est produit spécifiant les branchements à effectuer pour que la maquette soit fonctionnelle.
- De nouvelles fonctionnalités sont ajoutées comme une lampe, un micro, etc…
- Le prototype est amélioré esthétiquement parlant grâce à diverses décorations.

Les choix et les solutions envisagées

Après avoir assimilé le travail de nos prédécesseurs, nous avons commencé à rechercher les solutions à mettre en place pour répondre au cahier des charges.

Pourquoi donc arduino ET Raspberry pi?

Nous pouvons d'ores et déjà répondre à cette interrogation que vous avez peut-être. On peut en effet remarquer que cela fait beaucoup de moyens mis en œuvre, et c'est un bonne remarque. Nous avons récupéré le travail de l'année précédente, et nous nous somme posé la même question.

Le raspberry pi est indispensable, puisque nous voulons programmer directement sur la maquette. C'est depuis le système *raspbian* que l'utilisateur utilise scratch. Cela fait partie du cahier des charges, et cela évite le coût de passer par un ordinateur.

En théorie, le raspberry pi pourrait peut-être contrôler tous les composants depuis un code python avec les bonnes bibliothèques. Cependant, un arduino est spécialement conçu pour effectuer ces tâche de contrôle de composants. Notre choix est donc un pilotage de la

maquette grandement simplifié grâce à l'arduino, au prix de l'utilisation de protocoles de communication i2c pour relier l'arduino et le raspberry. Comme c'est l'interface scratch qui nous intéressait le plus, nous n'avons pas retravaillé cette question, d'autant plus que la communication i2c fonctionne très bien. On peut tout de même imaginer que dans les prochaines années, un effort soit fourni afin de se passer de l'arduino.
En fait, moins de transport d'information pourrait améliorer les performances de l'interface scratch, et diminuer la complexité du montage électronique ainsi que son coût.

Les boutons poussoirs

Nous continuons à utiliser le choix de nos prédécesseurs quant aux boutons poussoirs. Il est intéressant de noter l'emploi de résistances de "pull-up". Ce sont des résistances, situées entre la source d'alimentation et une ligne, et qui amènent cette même ligne à l'état haut. Aussi, les boutons sont les seuls capteurs à être reliés directement sur le raspberry pi : l'information est très facilement récupérée sur les ports GPIO, et exploitable dans le code python.

Des solutions pour l'interface scratch

Tout d'abord, il faut savoir que scratch existe en deux version. Scratch 1 ainsi que la dernière version, scratch 2. Ce sont des plate-formes d'apprentissage au codage, qui sont en premier lieu développées pour être utilisées en ligne, sur le site web de scratch. Il n'est pas vraiment possible d'accéder au hardware depuis un code "en ligne". En revanche, scratch 1 et 2 ont des versions "offline", sous forme de programmes, et il y a même des versions pour raspberry pi. La question était donc : Comment pouvons-nous accéder au hardware de notre machine depuis scratch1 offline ou scratch2 offline?

Après de longues heures de recherche et de tests, nous avions tiré quelques conclusions. Scratch1 possède une fonctionnalité appelée "remote sensor connections". Après activation, scratch1 ouvre un serveur sur lequel d'autres applications peuvent venir se connecter. On peut donc imaginer pouvoir échanger des variables et donc de l'information vers l'arduino. Nous poussons donc nos recherches un peu plus loin, et nous découvrons l'existence d'une bibliothèque python, scratchpy, qui offre directement une interface pour faire communiquer python et scratch1 via ce serveur. Du côté de scratch, on reçoit alors des messages sous forme de broadcasts.
À partir de ce moment, nous avons commencé à développer un code python qui effectuait le lien entre scratch1 et l'arduino, grâce à différentes bibliothèques. Nous sommes arrivés à un résultat parfaitement fonctionnel. Voici un extrait de code scratch qui montre à quoi ressemblait le résultat, côté utilisateur.

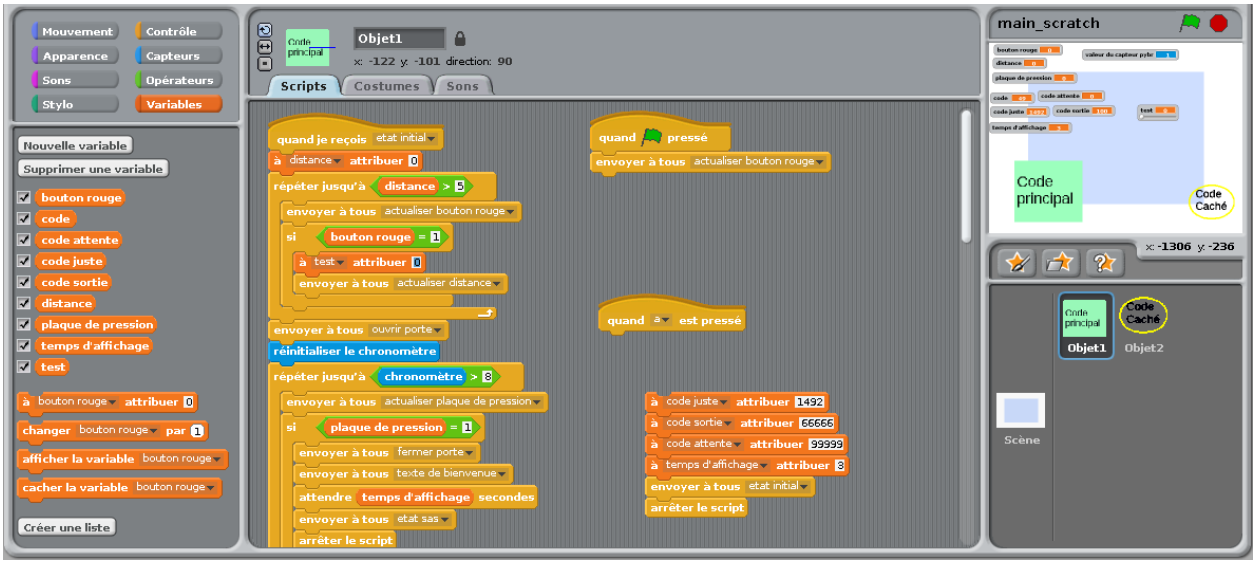


Illustration 3: Exemple de code scratch 1 avec notre première solution

En fait, cette façon de répondre au problème comportait plusieurs problèmes majeurs. La syntaxe était assez lourde pour l'utilisateur. Il fallait employer très souvent les blocs "envoyer à tous" et "quand je reçois", qui sont les blocs qui envoient et reçoivent les broadcasts. Ce n'était pas vraiment intuitif, et nous passions en plus par l'usage d'une "partie cachée" de code en scratch, qui effectuait divers réglages et qui était sensée être opaque au niveau de l'utilisateur.
Pendant longtemps, nous n'avons pas eu d'autre choix que de conserver cette solution. De plus, "remote sensor connections" est une fonctionnalité qui a été retirée dans scratch2, nous étions bloqués sur scratch1 avec cette solution. Le développement d'une extension scratch2 était pour nous à ce moment une solution que nous pensions trop irréaliste.

Nous avons alors exploré assez loin une possibilité d'amélioration de la solution en scratch1. En fait, depuis scratch1, il est possible d'accéder au "system browser". On y retrouve le code source de scratch1 ! Après avoir cherché tant bien que mal de la documentation sur cette notion, et de longues heures d'expérimentations délicates, nous étions en mesure de créer… De nouveaux blocs, en modifiant directement le code de scratch1 ! Cela aurait permis de simplifier grandement l'interface proposée à l'utilisateur, en proposant des blocs spécifiques à la maquette.

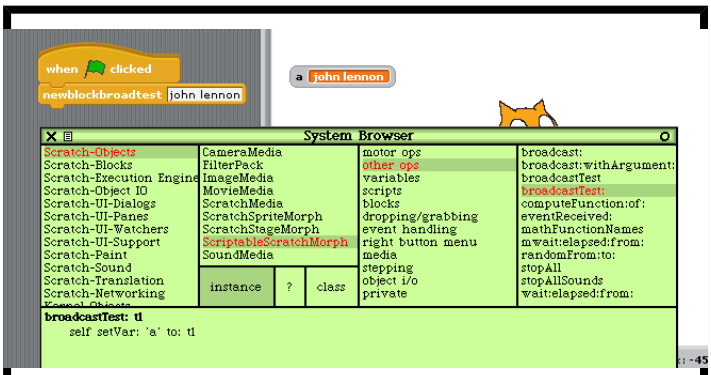


Illustration 4: à l'intérieur du system broswer de scratch 1

Cette solution allait être assez coûteuse à mettre en place, ainsi que difficile à débugger. De plus, il restait dommage que les élèves ne puissent pas programmer sous scratch2, la version la plus récente.

Nous nous sommes alors penchés plus sérieusement sur le développement d'une extension scratch2. Nous n'étions alors pas certain d'en être capable, mais après s'être bien documentés, nous avons souhaité tenter le coup, et cela s'est avéré fructifiant.

L'extension scratch2

La solution que nous avons mis en place est la suivante. Il est possible d'écrire une extension pour scratch2, et de l'ajouter à la liste des extensions. C'est une extension que l'on code en javascript, ce qui revient en fait à coder chaque nouveau bloc que l'on veut rajouter. C'est un processus un peu compliqué quand on ne l'a jamais fait, nous avons donc opté pour le rétro-engineering d'un code d'extension qui ajoute des blocs pour les GPIOs, dans le cadre d'un tutoriel. Voir ce lien.

Comme il reste très pratique de conserver un programme python pour communiquer avec la partie électronique, nous avons mis en place la stratégie suivante : Lors de l'appel dans scratch à une fonction de l'extension, le javascript communique avec le programme python via un serveur type websocket. Le programme python demande alors l'instruction correspondante à l'arduino. Ce protocole fonctionne parfaitement dans les deux sens (on récupère par exemple la mesure de la distance du détecteur de présence).

Pour plus de description technique, se référer à la partie correspondante plus loin dans le rapport.

Note intéressante : On pourrait remettre en question notre travail par le fait qu'une version 3.0 de scratch se prépare pour les années à venir. Après nous être renseignés, nous avons écarté ce problème futur puisque, encore plus que les versions précédentes, la version 3.0 sera orientée "online". Les connexions avec le hardware n'ont pas l'air d'intéresser tant que ça les développeurs, et une version "offline" 3.0 pour raspberry reste encore hypothétique. Notre travail restera donc sûrement un bon bout de temps pertinent avant de devenir obsolète.

Développer les blocs

Pour développer chaque bloc de l'extension, il faut coder son équivalent en javascript, en python et sur arduino, en respectant les protocoles de transfert entre chaque programme.

Il y a là un choix intéressant à visualiser : à quel point laisse-t-on de la liberté à l'utilisateur ? Par exemple, on pourrait hésiter entre un bloc "renvoi la distance à l'obstacle" et un bloc "renvoi 0 si obstacle < 5cm | 1 sinon". Le premier bloc laisse plus de liberté. Encore un exemple : "ouvre porte" ou "met l'angle du servomoteur à 90°" ?
Dans la description des résultats vous pourrez constater nos choix, qui sont pour la plupart arbitraires.

Argumentons simplement le bloc "demande de code".

Derrière ce bloc, il y a beaucoup de choses :
On affiche sur le LCD : "entrez le code:", on gère le dépassement (4 chiffres max), on gère la correction (touche C), la validation (touche B), ou la sortie (touche A) avec un code spécial

à 5 chiffres. Puis on renvoi le code.
En fait, c'est une opération assez délicate à gérer, avec beaucoup d'information à transférer et beaucoup de conditions (le code arduino est d'ailleurs assez complexe). C'est pour cela que nous avons choisi de laisser cette opération comme un bloc "tout en un", au lieu par exemple de faire des blocs "récupère touche", et de laisser l'utilisateur se débrouiller pour récupérer un code de manière sécurisée. Encore une fois, c'est un choix qui pourra être réétudié dans les années à venir.

La réalisation du shield

Le temps l'ayant permis après avoir implémenté la solution logicielle vue précédemment, nous avons pu réaliser un shield pour rendre plus fiable le montage électronique. L'objectif était aussi de supprimer le plus de câbles possible, et de simplifier les branchements par quelqu'un qui aurait la documentation pour cela.
Nous avons donc dimensionné un PCB sur KiCad. Nous avons choisi de séparer cette carte en trois cartes :

- Une carte "mère" qui ne comporte que des connectiques
- Une carte "fille" avec les résistances de "pull-up"
- une carte "fille" amplificateur audio pour le raspberry.

Ainsi, si une carte "fille" est défectueuse, pas besoin de changer l'ensemble du shield.
Le dimensionnement était un travail délicat : il fallait placer les composants à l'endroit exact pour que tout s'emboîte correctement, et que le tout soit intuitif à brancher.

Nous avons en plus ajouté une sortie "utilitaire au shield, afin de pouvoir récupérer facilement la masse, le +3V, le +5V, ainsi qu'un GPIO du raspberry si l'on veut ajouter facilement une fonctionnalité.

Résultats

Nous sommes plutôt satisfaits des résultats que nous avons obtenus. Les solutions trouvées et mises en œuvre correspondent au cahier des charges.

- Notre interface scratch pour piloter la maquette est fonctionnelle, facile d'utilisation et intéressante pédagogiquement parlant
- Nous avons pu améliorer la structure des connexions électroniques dans la maquette grâce au shield réalisé
- Le prototype est prêt à être testé dans un établissement par un enseignant d'abord, puis par des élèves.
- Ce rapport fournira à nos successeurs et aux personnes intéressées la documentation pour se servir du prototype et l'améliorer.

Les petits bémols :

- La plaque "fille" amplificateur audio ne fonctionnait pas, et nous n'avons pas eu le temps de la debugger. Si ce n'est pas un faux contact, ce doit être un problème de dimensionnement (voir description technique).
- Certains blocs de l'extension ne fonctionnent pas de manière totalement asynchrone. Voir les recommandations dans le guide de l'utilisateur. Cela n'empêche pas leur utilisation.

Illustration 5: L'extension dans scratch2

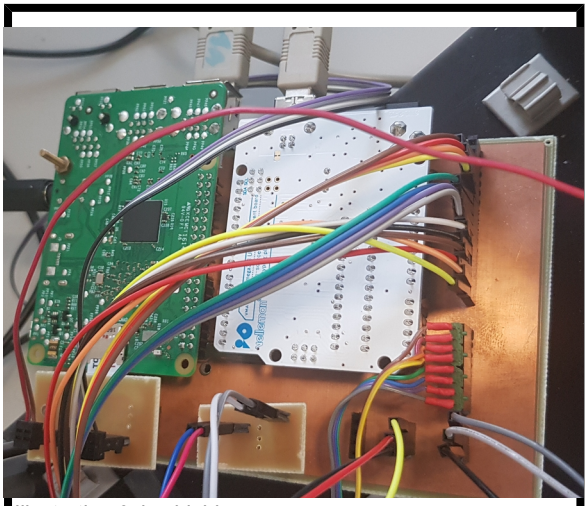


Illustration 6: le shield

Voir aussi :

- L'ensemble du code développé pour ce projet dans notre dépôt github.
- Une vue théorique du PCB en annexe

Description technique de la réalisation

Comment ça marche? Comment tout cela a-t-il été codé ? Dimensionné ?
Cette partie a pour vocation de répondre de manière assez précise à ces questions...

Les connexions entre les différentes entités

Ce schéma se suffit à lui même et explicite les liaisons et les interfaces entres les différentes entités qui composent la maquette.

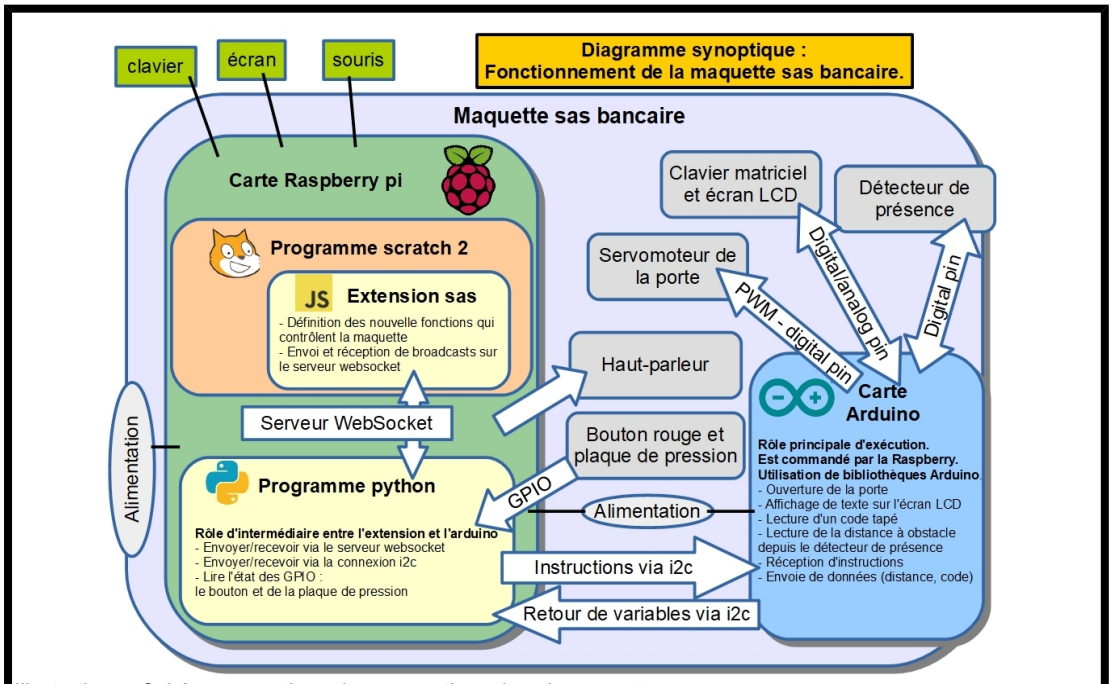


Illustration 7: Schéma synaptique des connections dans la maquette

Le codage logiciel

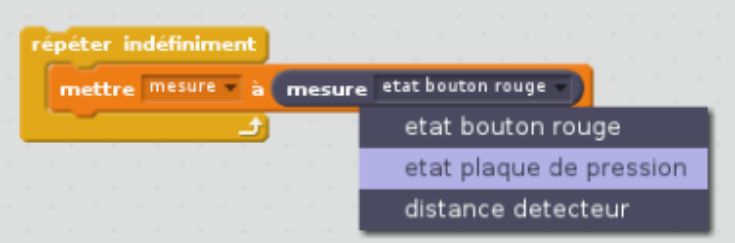
Nous ne pouvons pas dans ce rapport commenter chaque ligne de code, mais nous allons essayer dans cette partie d'expliciter la façon dont fonctionnent les différents codes que nous avons réalisé. Reportez-vous au code hébergé sur github si besoin.

Commandes et protocoles lors des échanges via i2c

Commande\protocole	python envoi via i2c	python envoi	python attend de recevoir	python lit	condition
ouvrir porte	0xa1				
fermer porte	0xa2				
détecte distance	0xa3		0xb3	mesure	
affiche texte à étage	0xa4	texte+étage			
affiche infos digicode	0xa5				
demande code	0xa6		!0xb0	0xb1 ou 0xb2	0xb1 : code==4*lecture du chiffre 0xb2 : code==66666
efface écran	0xa7				

Descendons pas à pas dans les couches de code, et notons les éléments principaux à la compréhension globale du codage de l'extension sas pour scratch 2. Nous prendrons l'exemple du bloc "mesure distance detecteur" puisqu'il est assez représentatif de la complexité globale du code mis en place.

Dans scratch2



C'est facile, intuitif, mais qu'y-a-t-il derrière tout cela?

L'extension en JavaScript

Pour apprendre à créer une extension scratch2, nous avons fait du retro-engineering sur le code proposé dans ce tutoriel. Nous avons pris ce qui nous intéressait, et modifié ou supprimé le reste.
Tout se passe dans le dossier */usr/lib/scratch2/scratch_extensions* du raspberry.
Il faut modifier le fichier "extension.json", et ajouter une ligne descriptive de l'extension, dans

notre cas:

```
{ "name":"Pi Sas", "type":"extension", "file":"sasExtension.js", "md5":"saslogo.png", "tags":["hardware"]  }
```

Notez que le champs “file” correspond au fichier javascript dans lequel sont codées les fonctions de l’extension.
Tout est donc dans ce fichier "sasExtension.js".
Ce lien indique comment ajouter des blocs. La particularité de notre extension, c’est que nous utilisons un serveur websocket pour communiquer avec python.
Revenons à notre fil rouge, la mesure de la distance.
On ajoute donc la ligne au descriptor:

```
["R", "mesure %m.capteur", "sensor_measure", "etat bouton rouge"],
```

“R” correspond à un bloc reporter de type asynchrone (on attend le retour de l’information).
Il faut paramétrer le menu du choix du capteur : %m.capteur

```
"capteur": ["etat bouton rouge", "etat plaque de pression", "distance detecteur"],
```

On écrit ensuite la fonction sensor_measure() qui prend en paramètre “sensor” (le capteur à interroger) et “callback()” la fonction qui renvoie l’information.

```
// Ask for measures of the sensors
ext.sensor_measure = function (sensor, callback) {
if (connected == false) {
        alert("Server Not Connected");
    }
console.log("sensors");
var msg = JSON.stringify({
    "command": 'measure', 'sensor': sensor
});
console.log(msg);
received_info = null;
    window.socket.send(msg);
received_info = null;
wait_callback(callback);
}
```

Illustration 8: sensor_measure JS

En résumé, le message (“command”: ‘measure’, ‘sensor’; sensor) est constitué, puis envoyé sur le serveur websocket, à destination du code python.
received_info est une variable globale qui prend pour valeur la donnée reçue par le serveur.
On entre donc dans la fonction que nous avons écrite “wait_callback()”, qui attend que received_info soit différent de NULL. Une fois que python nous répond avec la donnée, wait_callback() permet alors de sortir de la fonction en exécutant callback(received_info). La donnée reçue est envoyée à scratch, c'est la distance dans notre cas.

Le code python

Ce programme doit être lancé avec scratch2! Voyons comment il récupère le message envoyé sur le serveur websocket par javasScript.

```
def handleMessage(self):
    # get command from Scratch2
    payload = json.loads(self.data)
    print(payload)
    client_cmd = payload['command']

    if client_cmd == 'open_close_door':

    elif client_cmd == 'measure':
        sensor = payload['sensor']
```

Le message est récupéré dans payload, puis la commande dans client_cmd, puis le capteur dans sensor. Ainsi, python sait que l’on demande la mesure de la distance. Il met alors en

place le protocole associé (voir le tableau récapitulatif des protocoles que nous employons lors des échanges i2c).

```
elif sensor == 'distance detecteur':
    bus.write_byte(address, 0xa3)
    result = 0xb0
    while (result != 0xb3):
        time.sleep(0.05)
        result = bus.read_byte(address)
    result = bus.read_byte(address)
```

puis le result est renvoyé au code javascript avec self.sendMessage(msg).

```
payload = {'report': 'measure', 'result': result}
print('callback', payload)
msg = json.dumps(payload)
self.sendMessage(msg)
```

Code Arduino

Après la partie déclaration de constantes, voici le squelette du code Arduino :

```
// --- Fonctions de base ---

void setup() { // initialisation au démarrage de l'arduino
  Serial.begin(9600);
  monServo.attach(7); // Moteur branché sur la broche 7
  monServo.write(pos);
  lcd.begin(16, 2); // Initialise le LCD avec 16 colonnes x 2 lignes
  Wire.begin(SLAVE_ADDRESS); // lancement de l'i2c sur 0x12
  Wire.onReceive(receiveData); // définit la fonction pour la réception i2c
  Wire.onRequest(sendData); // définit la fonction d'envoi i2c
}

void loop() {
  if(demande_code){
    recup_code(); // cette fonction ne s'execute correctement que dans la loop!
    demande_code = 0;
  }
}



// --- fonctions à appeller lors de demandes de communication i2c ---

void receiveData(int byteCount) { //réception

void sendData() { // envoi

// --- fonctions actionneurs et capteurs ---

void ouvrir_porte() { // 0ax1

void fermer_porte() { // 0ax2

void detecte_presence() { // 0ax3

long microsecondsToCentimeters(long microseconds) {

void aff_info_code() { // 0xa5

void recup_code() { // 0xa6
```

Illustration 9: fonctions code arduino

Le point-clé de la communication entre l'arduino et le code python du raspberry se trouve dans *setup()*. On définit les fonctions *receiveData()* et *sendData()* comme fonctions respectivement de réception et d'envoi de donnée. Quand une donnée est reçue, *receiveData()* est appelée, quand une donnée demande à être lue, *sendData() est appelée.*

En fonction de l'instruction reçue, *receiveData()* redirige vers la fonction qui va directement faire une mesure ou actionner un composant OU se mettre dans un mode de réception de plusieurs mots. (envoi du texte à afficher par exemple).
Reprenons notre fil conducteur. DataReceived vaut ici 0xa3. On met donc en place le protocole, et dès que l'arduino a pu récupérer la mesure grâce à detecte_presence(), sendData() l'envoie au programme python.

Et voilà, la chaîne de l'information est déroulée.

L'électronique de la maquette

Nous allons dans cette partie décrire quelques aspects du dimensionnement du shield.

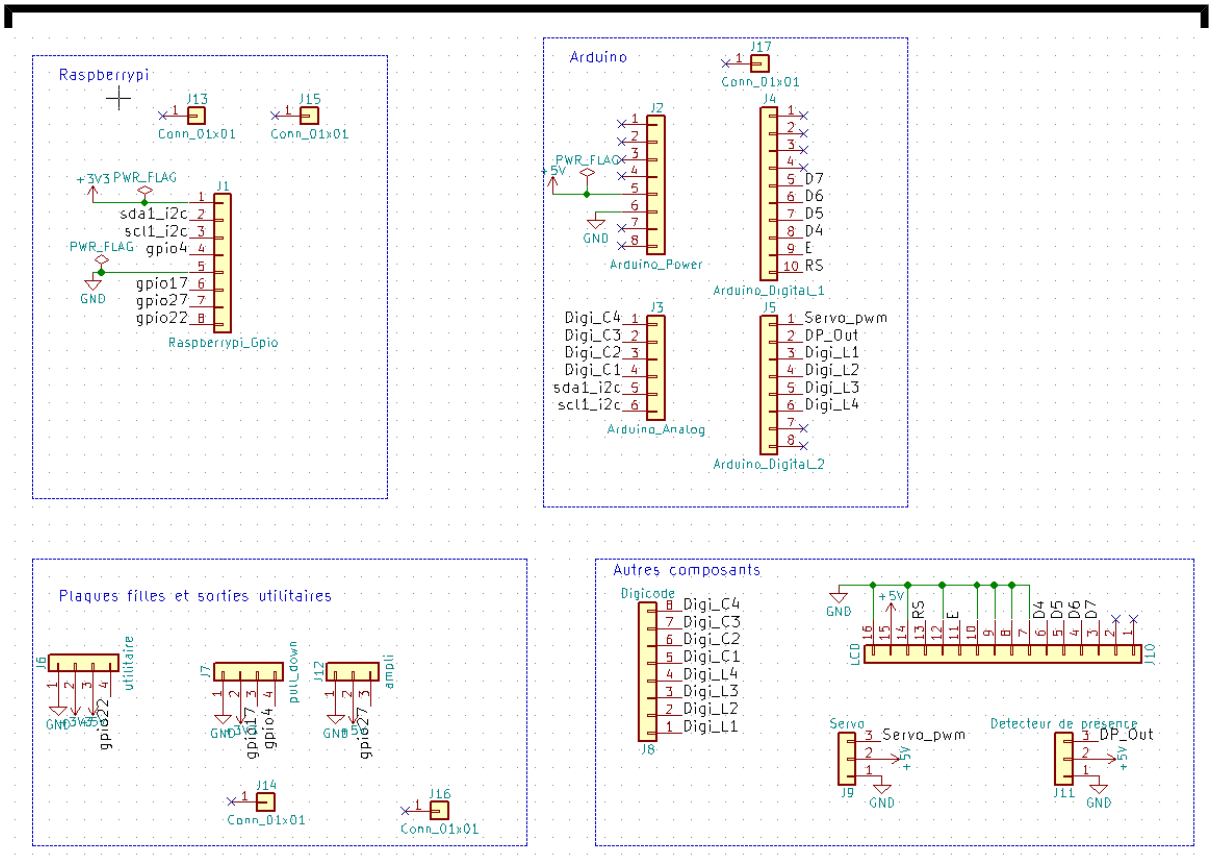


Illustration 10: Schéma carte mère

Sur cette capture de Kicad, on voit le schéma de la carte mère. Les liaisons composants/carte et carte/carte sont faite dans le PCB. Une personne qui souhaite câbler la maquette aura juste à brancher chaque composant sur la suite de connecteurs qui lui est attribuée.

Voir en annexe :

- la documentation concernant les pins du LCD du clavier matriciel.
- La documentation concernant les pins et les dimensions d'un raspberry pi et d'un arduino.
- L'ancien schéma de montage électronique

Le schéma suivant fait la synthèse du montage à effectuer :

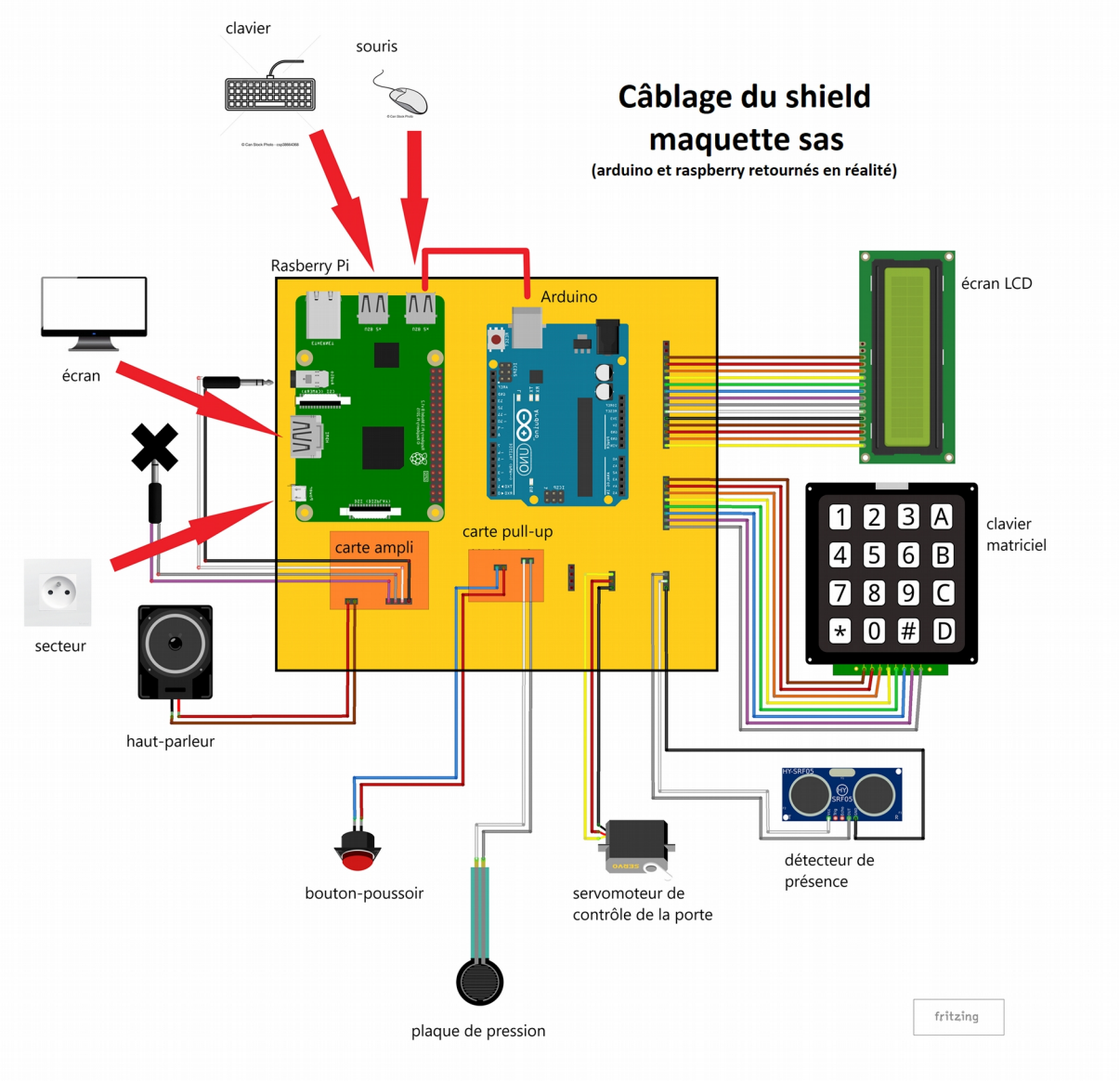


Illustration 11: schéma de câblage shield maquette

Quelques détails concernant la conception de la plaque amplificateur audio qui n'a pas fonctionné:

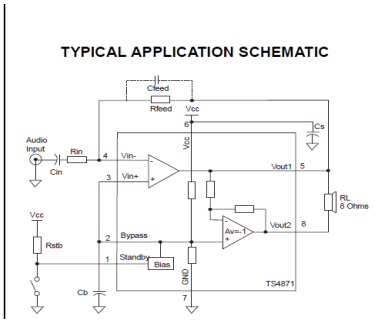


Illustration 12: extrait de datasheet TS4871

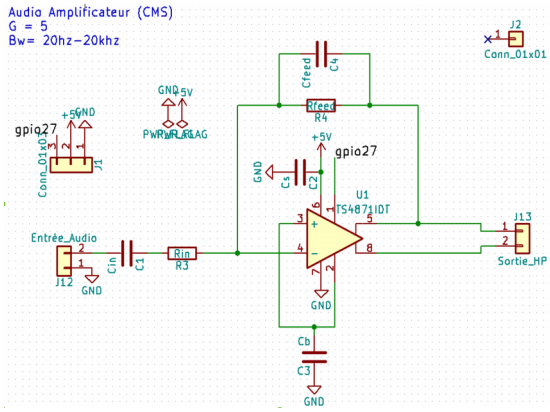


Illustration 13: shéma plaque fille amplificateur audio

Les valeurs des composants utilisés sont les suivantes:

- Cin – 1µ
- Rin – 10k
- Cfeed – 220p
- Rfeed – 47k
- Rstb – 1k
- Cb – 1µ
- Cs - 1µ

D'après nos calculs, on devrait avoir un gain de 4.7 idéal pour amplifier un niveau ligne à 5V, et une bande passante entre 20Hz et 20kHz.

Guide pratique d'utilisation et de programmation de la maquette

Beaucoup de travail a été mis en œuvre pour que la maquette SAS soit fonctionnelle. Mais comment l'utiliser en pratique lorsque l'on veut commencer à la programmer ?
Pas de panique ! Il suffit de suivre le tutoriel suivant, et il n'y a rien de bien compliqué !

Matériel

- la maquette SAS
- le chargeur usb
- une moniteur avec entrée hdmi
- une souris et un clavier

Branchements

Connectez tous les périphériques et le moniteur sur le raspberry pi interne à la maquette. Vous pouvez ensuite alimenter le raspberry. Celui-ci démarre sous raspbian, tout va bien.

Programmer la maquette

Le raccourcis Ctrl+Alt+T lance un terminal qui peut aussi être retrouvé dans *menu* > *accessoires*.
Entrez simplement la commande: *sas.*
Cela a pour effet de lancer scratch2, ainsi que le serveur python qui communique avec le reste de la maquette. Quand vous voudrez quitter, Ctrl+Alt+c dans le terminal termine le processus de ce serveur.
Une fois dans Sratch2, aller dans "ajouter blocs", puis "ajoute une extension".

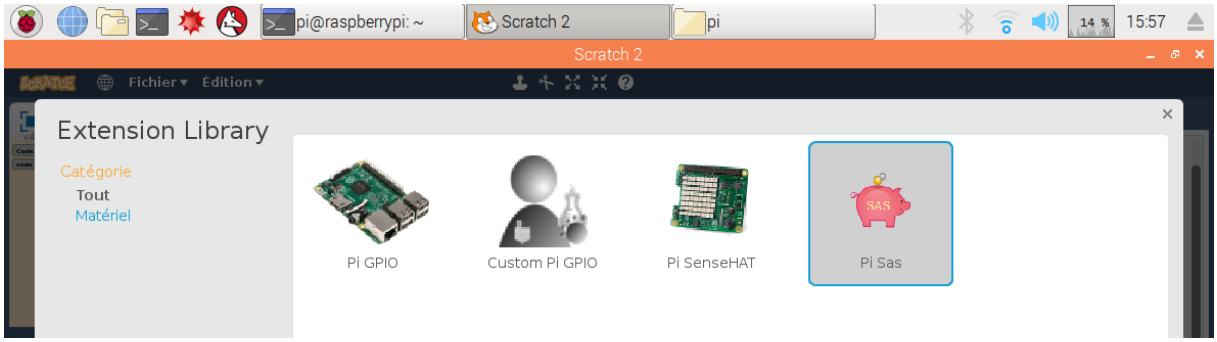


Illustration 14: choix de l'extension

Choisir alors l'extension "Pi Sas".

Illustration 15: extension sas

La liste des blocs de l'extension apparaît. Afin de pouvoir les utiliser, **toujours exécuter le bloc intitulé "connexion au serveur sas"**.

Le voyant à la droite de "extension sas" devient alors vert, on peut se mettre à programmer la maquette à l'aide des blocs de l'extension.

Les blocs de l'extension

Le tableau suivant récapitule les blocs qui permettent de programmer la maquette. Certains blocs sont soumis à des contraintes d'utilisation, il faut les utiliser avec précaution. Dans le cas où une utilisation abusive ferait planter la maquette, vous pouvez toujours la débrancher et la rebrancher.

Nom du bloc	Fonction du bloc	Précautions à employer
Connection au serveur sas	Connecte l'extension au serveur python	Exécuter avant tout appel à un bloc de l'extension
ouvrir/fermer la porte	Ouvre ou ferme la porte du sas	aucune
affiche "ce que vous voulez" à l'étage 0/1 du LCD	Affiche un texte de moins de 16 caractères sur l'étage 0 ou 1 du LCD.	Employer un délai entre deux appels ! 0.5s?
affiche instructions du digicode	À afficher avant "demande de code".	Employer un délai entre deux appels ! 0.5s?
effacer l'ecran	Efface les deux étages du LCD	Employer un délai entre deux appels ! 0.5s?
mesure etat bouton rouge/etat plaque de pression/distance detecteur	Renvoi le résultat de la mesure choisie. À stocker dans une variable?	aucune
demande de code	renvoi le code entré sur le digicode. À stocker dans une variable?	Ne pas exécuter d'autres blocs de l'extension pendant la récupération du code !

Et maintenant, à vous de jouer ! Vous avez toutes les clés en main pour commencer à programme la maquette SAS.
Une dernière recommandation. Si vous sauvegardez un nouveau projet avec l'extension sas chargée, l'extension du fichier sera ".sbx". Il est préférable de renommer ".sbx" en ".sb2". Vous pourrez alors retrouver correctement vos projets en faisant "ouvrir", depuis scratch2.

Exemples d’exercices à caractère pédagogique

Voilà ! Scratch 2.0 est ouvert et l’extension est chargée. Quels exercices pouvons nous alors proposer aux élèves ?

Familiarisation avec la maquette

En premier lieu, il est conseillé de prévoir le temps nécessaire pour découvrir la programmation les nouveaux blocs qui commandent la maquette. On peut alors actionner l’ouverture de la porte, écrire du texte à l’écran, entrer les mesures des capteurs dans des variables, et pourquoi pas tester les limites des blocs en cherchant à faire planter le serveur.

Avant de s’attaquer à un scénario plus complet, qui utiliserait toutes les fonctionnalités du sas, il peut être bon d’échauffer ses élèves avec un projet moins complexe:
Par exemple, un jeu de “+ ou -”. Il s’agit d’entrer un nombre sur le digicode, qui sera à faire deviner à une seconde personne, qui testera des nombres et sera aiguillé par l’information “c’est plus/moins!”. Cela fait utiliser le digicode, le lcd, le haut parleur, pourquoi pas les boutons ainsi que les notions d’algorithmique en scratch2.

Le scénario du sas bancaire

Ici, nous abordons vraiment la finalité de notre projet : L’écriture d’un code scratch2 qui permet une mise en scène dans le contexte où notre maquette représente un sas bancaire. Le scénario que nous proposons est le suivant :

“Un personnage se présente devant le détecteur de présence, la porte s’ouvre alors et il entre dans le sas. La porte se referme derrière lui, il entre son code bancaire. Si ce code est bon, il est félicité (pas encore de distributeur de billets fonctionnel !). Il est ensuite invité à sortir en pressant le bouton rouge. La porte s’ouvre, le personnage sort du sas et la porte se referme. Du son peut être joué à différents moments”

Bien entendu, il faut penser à tous les cas possibles : personne n’entre dans le sas après ouverture de la porte, code mauvais, etc. Nous avons bien sûr codé cet exercice. Vous pourrez retrouver cet exemple dans la maquette et charger directement le fichier scratch2.

- En annexe, vous trouverez le code scratch ainsi réalisé ainsi que le logigramme correspondant.

Vous êtes bien sûr libres d’inventer d’autres exercices avec notre maquette comme support, ou même de développer vos propres extensions en prenant exemple sur la notre, l’éventail des possibilités entre scratch2 et l'électronique est finalement assez large, comme nous venons de le montrer.

La continuité du projet

Bonjour ! Toi qui reprend peut-être notre projet là où nous l’avons laissé. Voici quelques notes concernant les perspectives que nous voyons pour ce projet, ainsi que quelques conseils. Ce n’est que notre avis sur la continuité du projet. Tu es libre d’utiliser ou d’ignorer ces conseils comme bon te semble.
Bien sûr, vous pourrez nous demander l’accès à tous les fichiers utiles au projet...

Conseils pour la prise en main

Il y a beaucoup de choses que nous avons appris pendant ce projet, ce serait dommage que tu perdes du temps aux mêmes endroits que nous ! Normalement, si tu as lu le rapport jusqu'ici, tu as déjà de quoi faire fonctionner correctement la maquette, l'améliorer et la modifier selon les objectifs de ton projet. Quelques conseils supplémentaires ne font tout de même pas de mal.

Le raspberry pi

Nous n'avions jamais utilisé de raspberry pi avant ce projet. Voici quelques pistes si c'est aussi ton cas.

- Raspbian est le système d'exploitation sur raspberry pi. C'est un système linux, il sera utile d'apprendre à utiliser le terminal correctement.
- **Si tu dois réinstaller un système raspbian :** https://raspbian-france.fr/installer-raspbian-premier-demarrage-configuration/
- Ne pas oublier d'activer l'i2C ainsi que le choix de la sortie audio dans les paramètres en exécutant: *sudo raspi-config*
- **Les commandes pour tester l'I2C:**

Pour plus d'informations, utiliser la commande *man* (manuel). Le "1" est le port i2c 1. Le 0 ne semble pas fonctionner.
Vérifier si l'arduino est bien connecté sur 0x12
i2cdetect -y 1

```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- 12 -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Illustration 16: commande test i2c

Écriture d'une donnée (commande?)
i2cset -y 1 0x12 0xa?
Lecture d'une donnée
i2cget -y 1 0x12

- **Télécharger des paquets**
  dans le cas où tu ne serais pas familier avec linux (c'est le moment de le devenir), on installe des paquets avec: *sudo apt-get install nom_du_paquet*
- Pour récupérer toutes nos sources :
  git clone https://github.com/lerbourl/phelma_projet_sas.git
- La commande *sas* que l'on exécute pour lancer le serveur python ET scratch2, est en fait un alias, que nous avons défini dans le .bashrc :

L'arduino

Il faut installer le logiciel arduino: https://www.arduino.cc/
Tu pourras alors éditer et télécharger les programmes dans l'arduino. Si tu n'as jamais

programmé en arduino, le code risque d'être assez lourd à comprendre. Il faut s'accrocher. Et modifier quand on est certain de ce que l'on fait.

L'électronique

- Il peut sembler bête de le préciser mais si jamais vous rencontrez un "bug inexpliqué" (plus fréquents qu'il n'y paraît !), vérifier toujours le câblage (ou les faux contacts si pcb). L'ajout du shield a grandement amélioré la stabilité des connexions mais on ne sait jamais.
- Lorsque l'on en vient à programmer les protocoles de transmission via i2c, toujours penser que le temps de transmission des bits peut être un facteur limitant (et un facteur de bugs) !

Finalisation et corrections de notre avancement

Par manque de temps, nous n'avons pas mis en oeuvre certaines corrections ou améliorations au niveau du code ou du shield. Si il est question de faire une version stable et propre de la maquette, voici ce qui pourrait être retravaillé.

L'électronique

Encore une fois, la partie "amélioration des connectiques par le développement d'un shield" est un objectif supplémentaire qui est arrivé plutôt vers la fin du projet. Le shield est globalement fonctionnel, mais il y a encore matière à l'améliorer. Voici la liste de ce que nous aurions voulu corriger sur le projet KiCad.

- Débugger la plaque ampli-audio, ou en refaire une nouvelle.
- Déplacer plaque audio : elle chevauche la carte SD du raspberry !
- Le port Ethernet du raspberry entre en collision avec le PCB, raspberry à déplacer.
- Enlever les plans de masse faciliterai les très nombreuses soudures et préviendrai les faux contacts.
- Prévoir des trous plus grands pour pouvoir utiliser les connecteurs que nous avions commandé à la base (identiques à ceux du raspberry).
- Prévoir un meilleur maintien mécanique.
- Prévoir plus de détrompeurs et de commentaires sur le PCB.
- Prévoir en général une meilleure organisation des éléments dans l'espace, par rapport à la provenance des câbles dans la maquette.

Le codage de l'extension

Nous n'avons pas non plus eu le temps de revenir sur certains défauts des fonctions de l'extension.

- Toutes les fonctions ne fonctionnent pas de manière totalement asynchrone.
- "demande de code" par exemple doit être utilisée seule ! Si on tente un envoi ou une lecture de donnée au même moment, il peut y avoir conflit.
- Idem pour l'affichage : Vouloir afficher deux textes successivement et sans délais provoque un conflit. Il faudrait coder ces fonctions de manière asynchrone !
- Une boucle infinie dans scratch, qui utilise une fonction de l'extension quand le serveur python est déconnecté peut faire crasher le raspberry ! C'est à cause du message d'erreur "serveur déconnecté" qui s'affiche en boucle.

- En reprenant toute la structure de “demande de code”, on pourrait la rendre programmable directement sur scratch! Il faudrait réussir à écrire la fonction en scratch: “récupère caractère”.

Remarquons qu’en prenant les bonnes précautions et en étant conscient de ce que l’on fait, ces défauts peuvent ne pas poser de problèmes.

Ajouter de nouvelles fonctionnalités

Il peut être assez intéressant de rajouter de nouvelles fonctionnalités à la maquette. Un nouveau capteur ? Un nouveau composant ? Pourquoi pas un micro ? Une lumière? Une reconnaissance d’empreinte digitale ? d’autres boutons ?
Dans tous les cas, il est “relativement” aisé d’ajouter un nouveau composant contrôlable. Il faut coder la transmission de l’information depuis scratch, jusqu’à l’arduino qui exécute la fonction (ou jusqu’à python si on veut simplement utiliser les GPIOs).
Si vous avez suivi l’explication technique jusque là, l’ajout de nouvelles fonctions à l’extension scratch ne devrait pas être un problème.

Dans l’arduino

Fichier “sas_arduino_2.0.ino”.
Ajouter la nouvelle fonction qui interagit avec le nouveau composant. Utiliser potentiellement une bibliothèque Arduino.

```
// traitement de la réception d
if (compteur_reception == 0){
  if (dataReceived == 0xa1) {
    ouvrir_porte();
  }
  if (dataReceived == 0xa2) {
    fermer_porte();
  }
  if (dataReceived == 0xa3) {
    detecte_presence();
  }
```

L’appel à cette fonction pourrait être codée par une réception de 0xa8 par exemple?
Bien entendu, des protocoles de transmission plus complexes que le simple envoi d’une commande peuvent être mis en place. Vous pouvez vous inspirer alors de la demande code, ou de l’envoi du texte à afficher.

Dans le fichier python intermédiaire

Fichier “pywebsocketserver.py”.

Il faut choisir une chaîne de caractères qui correspondra à notre fonction, et qui sera transmise depuis scratch par le serveur websocket. disons ‘new_f’ par exemple. Il suffit de rajouter une condition pour exécuter la transmission i2c ou l’écriture/lecture sur les ports GPIO, uniquement si client_cmd vaut ‘new_f’. S’inspirer ensuite des autres fonctions pour les protocoles de transmission avec l’arduino. Penser à retourner un callback si le bloc scratch attend un retour.

Dans le code de l’extension sas

Fichier “sasExtension.js”

```
118     ext.display = function (text_string, floor) {
134
135     ext.display_instr = function () {
146
147     ext.clear_lcd = function () {
158
159     // Asks for the code and return it
160     ext.ask_code = function (callback) {
173
174     // Block and block menu descriptions
175     var descriptor = {
176         blocks: [
177         // Block type, block name, function name
178         ["w", 'connection au serveur sas', 'cnct'],
```

Ici, deux opérations à faire. Ajouter le nouveau bloc en ajoutant une ligne au descriptor, et écrire la fonction javascript associée. ne pas oublier le “wait_callback()” qui permet d’attendre un retour de python de manière asynchrone si besoin.

Dans scratch 2.0

Et bien c’est tout. Chargez l’extension, votre nouveau bloc est là, prêt à être utilisé !

Une version industrialisable

Tout d’abord, il serait bénéfique d’avoir des retours du corps enseignant sur notre prototype. Une fois le dialogue engagé, vous pourrez optimiser le code et l’électronique, afin de développer un prototype “industrialisable”, qui conviendrait aux exigences pédagogiques. Il faudra réfléchir à l’achat de tous les composants séparés. Une rapide estimation nous donnais moins d’une centaine d’euros en coût d’achat des composants, ce qui est tout à fait concurrentiel par exemple vis-à-vis des LEGOs programmables. Et ce prix pourrait encore être revu à la baisse ! Pour ce qui est de la tenue mécanique pour remplacer la maquette, pourquoi ne pas dimensionner une coque imprimable en 3D ? Est-ce possible de se passer de l’arduino afin de baisser encore le coût de fabrication ?

Nous espérons fortement que ce projet que nous avons porté cette année soit poursuivi dans les années à venir...

Bilan financier

Cette année, nous n’avons pas eu besoin de commander beaucoup de matériel.

Coûts et frais des composants électroniques:

Les frais des composants électroniques (hors frais de port) de cette année sont assez faibles, ceci s’explique par le fait que notre travail s’est surtout focalisé sur la partie programmation de la maquette. Lorsque nous avons voulu commencer à fabriquer le shield, nous avons eu besoin de connecteurs type raspberry pi. Une commande en grosse quantité a été effectuée : ce sont des composants qui profiteront à d’autres groupes.
Par la suite, suite à une erreur de dimensionnement de la taille des trous de perçage, nous nous sommes retrouvés à ne quasiment pas utiliser ces composants! Ils ont été rendus à Phelma, ils ne seront pas perdus…

Composant	Coût du lot en euros
Connecteur carte-à-carte,2.54mm 10 contacts 2211S X 50	2.735
Connecteur carte-à-carte,2.54mm,10 contacts 2212S X 10	2.16
Frais de port	**6.90**
Total	**11.80**

Coûts estimés du prototype de la maquette:

La maquette sas a été récupérée d'une ancienne maquette de TP de l'ENS3. Une rapide recherche sur internet nous donne son coût si nous avions dû la fabriquer de toutes pièces.

Composant	Valeur estimée du composant (€)
Clavier matriciel	0.5
Ecran LCD	7.5
Servomoteur	5
Carte Arduino	10
Carte Raspberry Pi 3	30
Hauts parleurs	5
2 X boutons poussoirs	3
Capteur de distance ultrason	14
Chargeur	5
Electronique et tenue mécanique	10
Total	**90**

Gestion du groupe

Retours d'expériences individuels

Mously

Ce projet a été pour ma part une occasion d'acquérir de nouvelles connaissances aussi bien dans l'électronique que dans la programmation. En effet il m'a permis de me familiariser avec des composants électroniques tels que l'Arduino et la Raspberry Pi que je n'avais pas eu l'occasion d'utiliser auparavant. Même si nous n'avons finalement pas eu l'opportunité de

le présenter à des élèves, l’aspect pédagogique m’a beaucoup plu car il est intéressant de faire découvrir à un public non initié tout ce qu’il est possible de faire avec l'électronique. J’ai également gagné en expérience en ce qui concerne la gestion de projet car il fallait beaucoup coordonner avec les autres membres du groupe pour mener à bien le projet dans sa globalité.

Louis

Je garderai un très bon souvenir du projet sas auquel je me suis beaucoup identifié. J’étais très intéressé par l’aspect programmation, et je n’ai pas été déçu ! Nous avons dû concevoir de véritables protocoles de transfert de données et le codage réalisé dans les différents langages était tout de même assez poussé ! J’ai ainsi enrichi mes connaissance pratiques et théorique dans beaucoup de notions très utiles : La découverte de l’arduino et du raspberry pi, le codage javascript, l’utilisation d’un serveur websocket et d’un protocole i2c. Je suis d’ordinaire moins intéressé par l’électronique, mais le fait de pratiquer en dimensionnant le shield sur KiCad était très formateur. L’aspect pédagogique du projet m’a aussi attiré pour choisir ce projet, et en effet, c’est très motivant de savoir que le fruit de notre travail pourrait sortir du cadre d’un “simple projet 1A”, et qu’il pourrait servir de support d’apprentissage! Le travail en groupe a été aussi formateur, par les responsabilités qui incombent au chef de groupe. Tout cela aura été un projet très complet, j’espère être tenu au courant de sa poursuite, si poursuite il y a.

Hamza

Le projet du SAS bancaire était une expérience enrichissante en tout point: la pluridisciplinarité requise pour aboutir à la fin du projet m’a permise d’acquérir de nouvelles connaissances dans plusieurs domaines différents. En passant de la programmation sur Scratch à la programmation sur Arduino puis à la partie électronique de la maquette, le projet a surtout constitué une plateforme d’apprentissage et de mise en pratique de nos connaissances théoriques. Le travail en groupe m’a, pour sa part, permis de m’initier à coordonner mon travail avec celui de mes collègues, en répartissant les tâches pour chaque partie du projet.

Antoine

Ce projet aura été pour moi une expérience enrichissante, me mettant face à des notions de programmation et d’informatique en général auxquelles je n’ai jamais été confronté, et que je n’aurais sans doute jamais vu dans d’autres circonstances. J’avais au départ choisi ce projet pour son aspect pédagogique, et bien qu’au final nous n’ayons pas eu de rapport direct avec un corps enseignant, la volonté de toujours rendre l’utilisation plus simple et plus intuitive dans cette optique s’est révélé être, en plus de la multitude de nouvelles notions à assimiler, le véritable défi et intérêt de ce projet. La coordination avec les autres membres du group a également été formatrice, nos niveau de compréhension n’étant évidemment pas égale dans tous les domaines abordés.

Analyse critique

Au global, ce projet a été plutôt bien mené par tous ces acteurs. La communication avec notre tuteur était assurée par un des membres du groupe. M.Voisin a aussi joué un grand rôle en nous aiguillant et en nous conseillant. Le résultat est là, nous avons répondu au cahier des charges en proposant nos propres solutions, et le prototype est prêt à être testé en “situation réelle”. Toute cette réalisation a été effectuée dans les temps, nous avons correctement organisé notre planning. Bien sûr, la réalisation technique pourrait être encore poussée plus loin, et nous nous sommes efforcés dans ce rapport de laisser des pistes à l’intention d’un potentiel groupe l’année prochaine qui reprendrait le projet sas…

Diagrammes de Gantt

Le premier diagramme de Gantt montre la façon dont le travail était envisagé au départ, tandis que le second diagramme représente la façon dont nous avons modifié notre vision du projet en cours de route.

Diagramme initial

L'objectif initial du projet était de rendre la maquette entièrement pilotable depuis scratch donc avons donc tout d'abord cherché ensemble des solutions techniques pour faire le lien entre les différents langages de programmation, avant de nous répartir le codage une fois cette solution trouvée (sous la forme d'une bibliothèque python).
Nous avons atteint cet objectif beaucoup plus tôt que ce que nous avions planifié, puisque la maquette était pilotable au bout de la 12e séance.

Diagramme de Gantt – Groupe 33

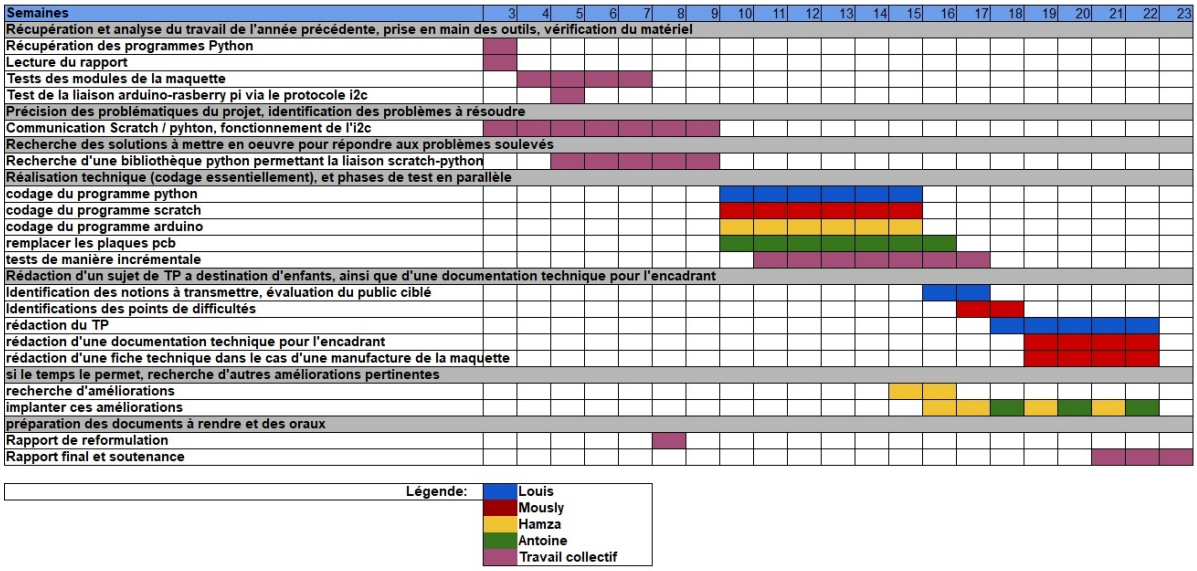
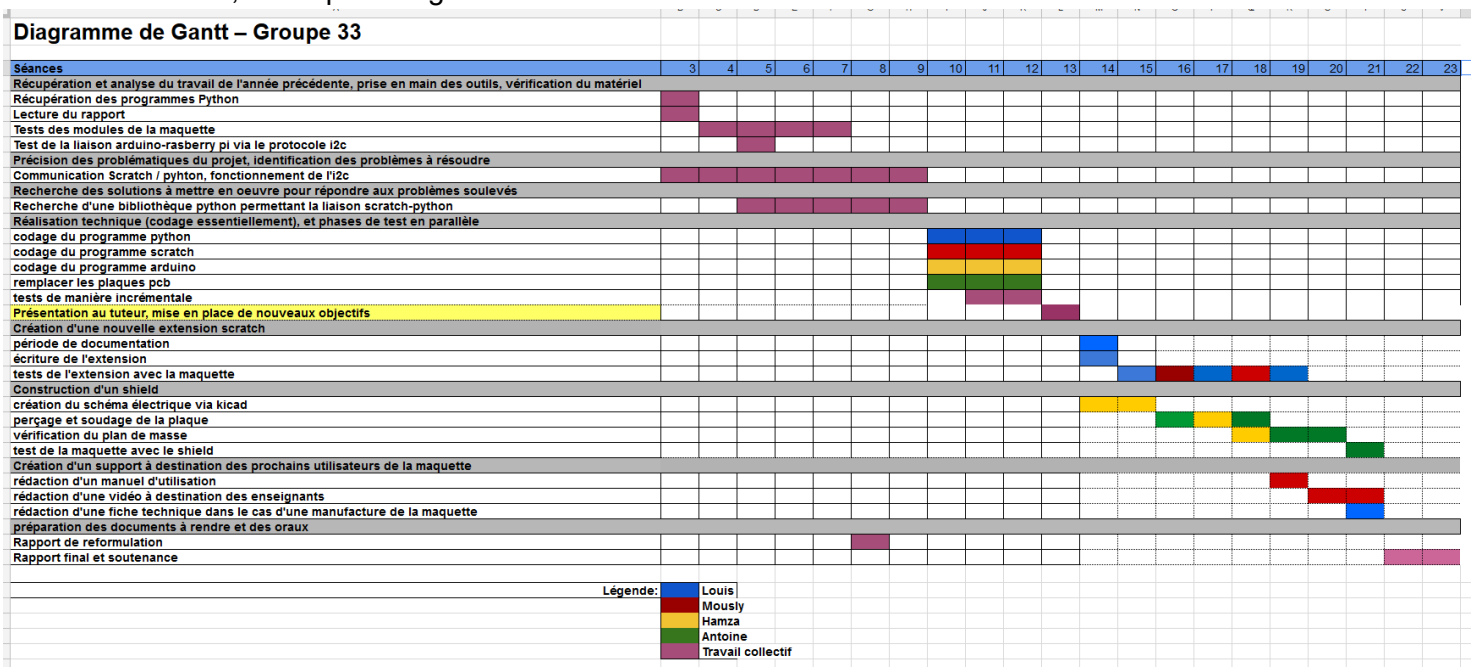


Diagramme révisé à la semaine 13

Une entrevue avec le tuteur a mis en avant deux nouveaux objectifs : améliorer la partie scratch, déjà fonctionnelle, pour rendre le contrôle encore beaucoup plus intuitif pour les enfants, et créer un shield pour regrouper toute la partie électronique de la maquette et la rendre plus fiable. C’est ici que nous avons commencé la programmation de l’extension

scratch 2.
Nous nous sommes donc répartis en deux groupes chargés de la réalisation de ces deux objectifs. Pour finir, nous avons cherché à produire des documents représentatifs de nos travaux, ainsi que des guides d'utilisation.



Annexes

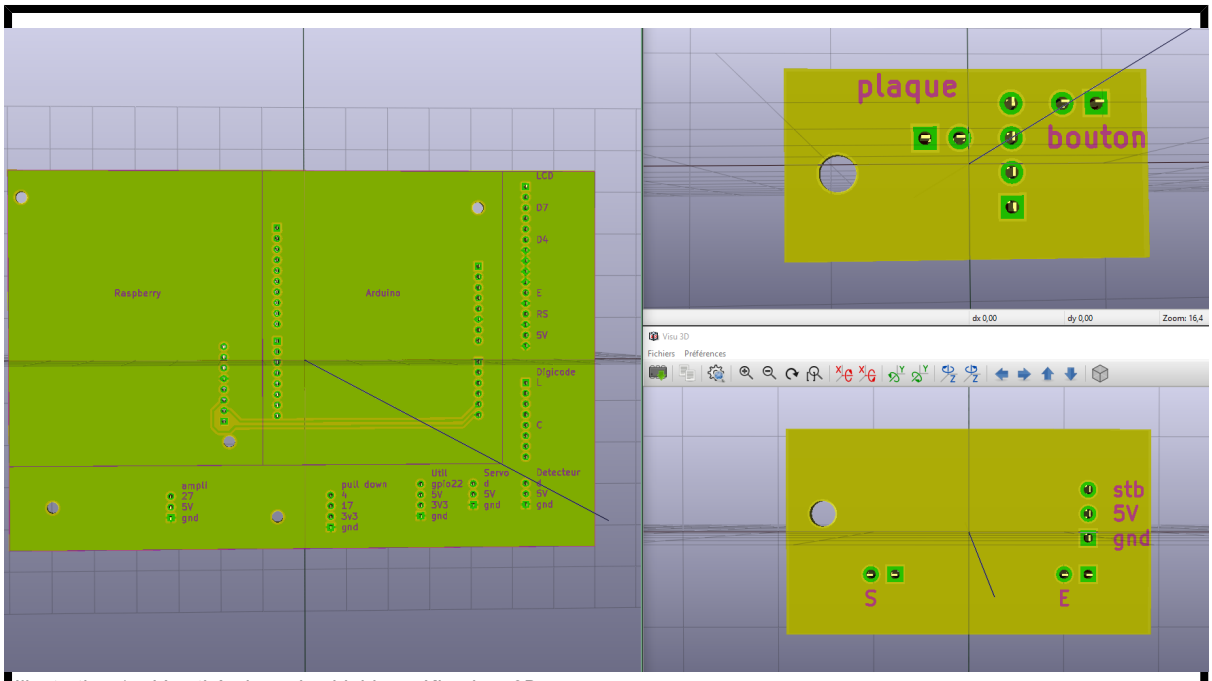


Illustration 17: Vue théorique du shield ave Kicad en 3D

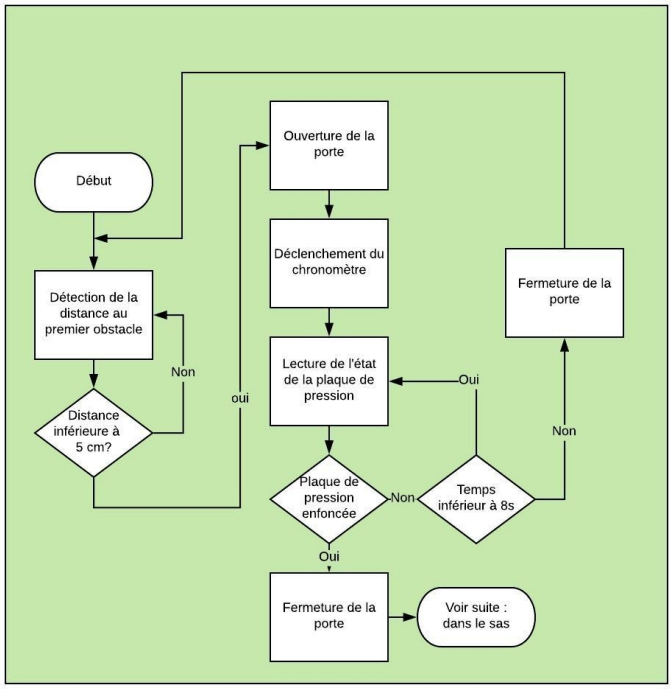


Illustration 20: logigramme scénario partie 1

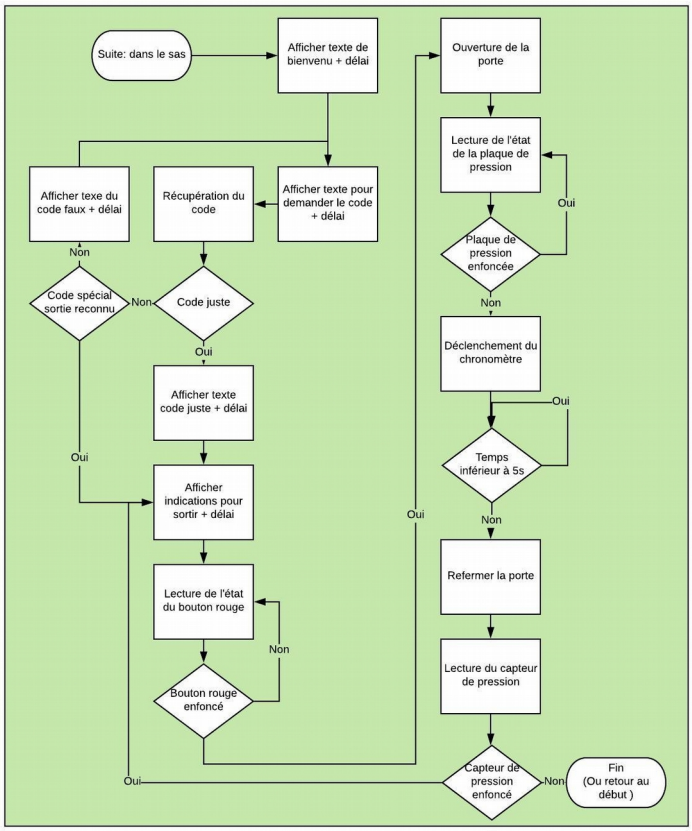


Illustration 18: logramme scénario part 2

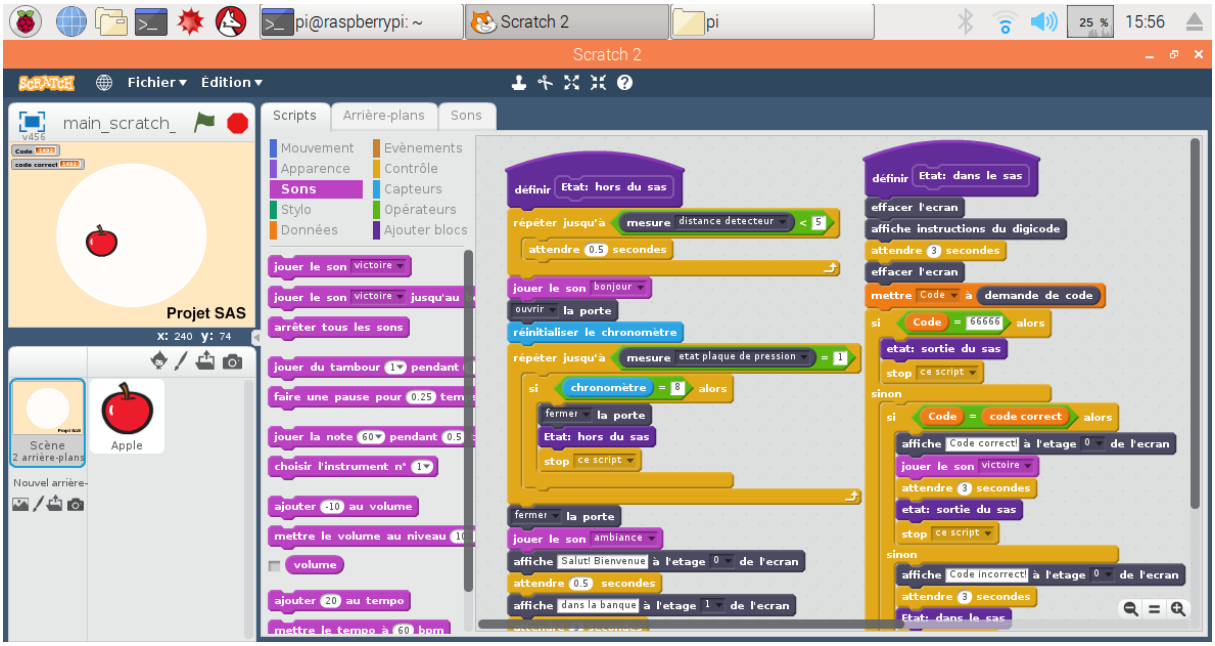


Illustration 19: code scratch scénario 1

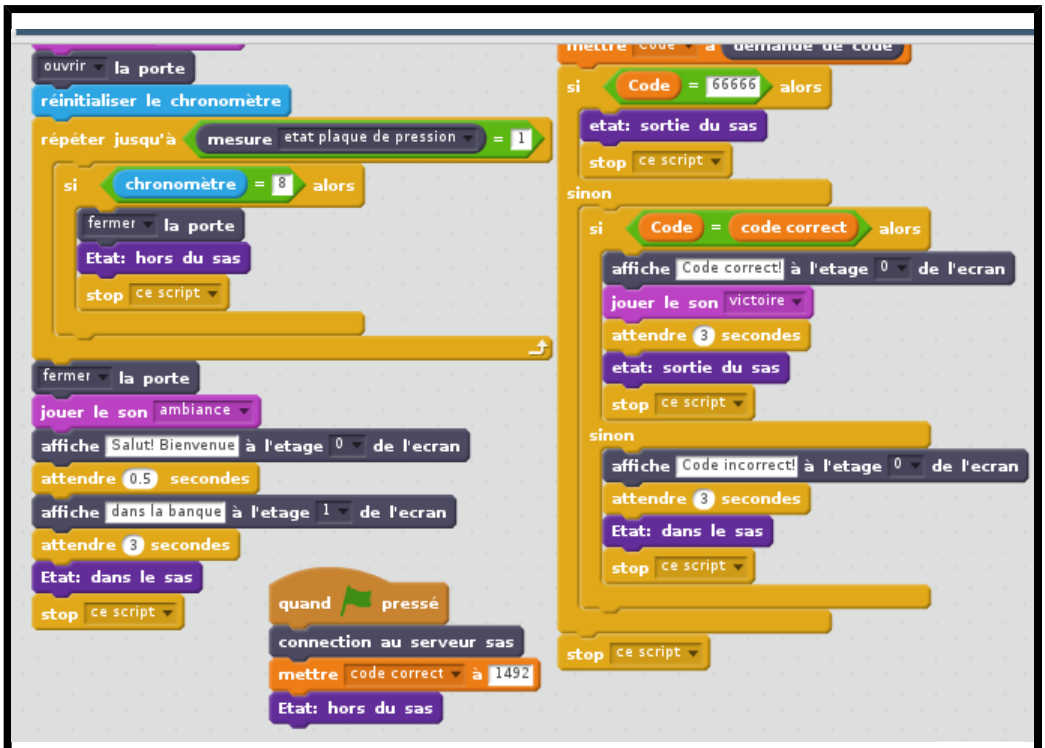


Illustration 21: Code scratch scénario 2

Illustration 22: Code scratch scénario 3

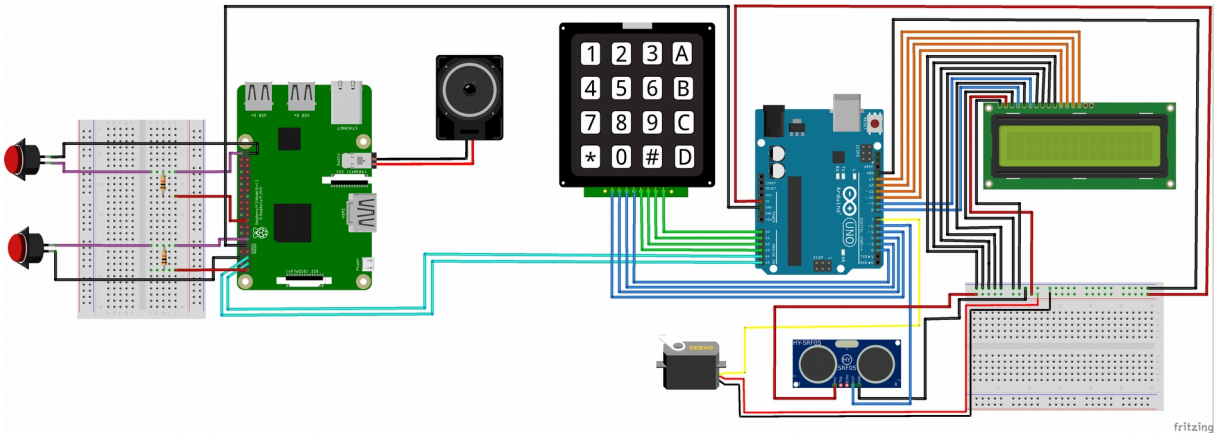


Illustration 23: Ancien montage électronique

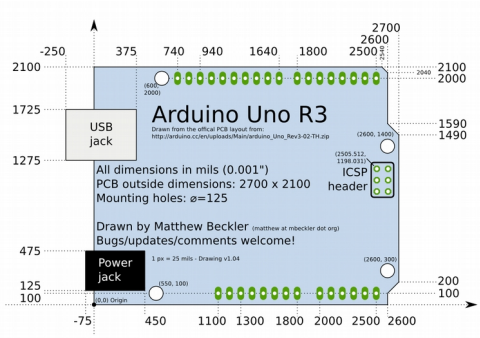


Illustration 24: Dimensions arduino

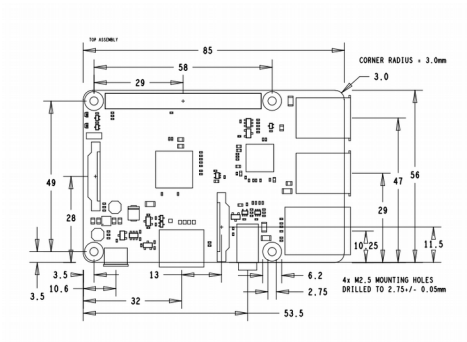


Illustration 25: Dimensions raspberry pi

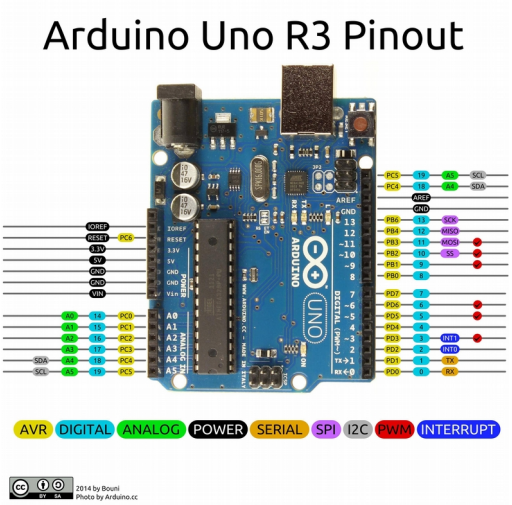


Illustration 27: pins arduino

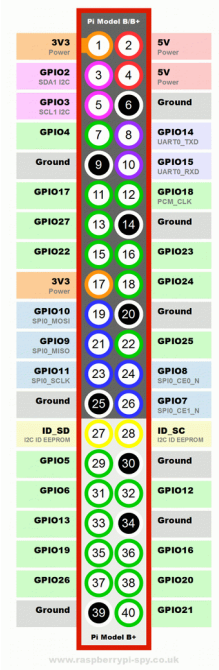


Illustration 26: pins raspberry

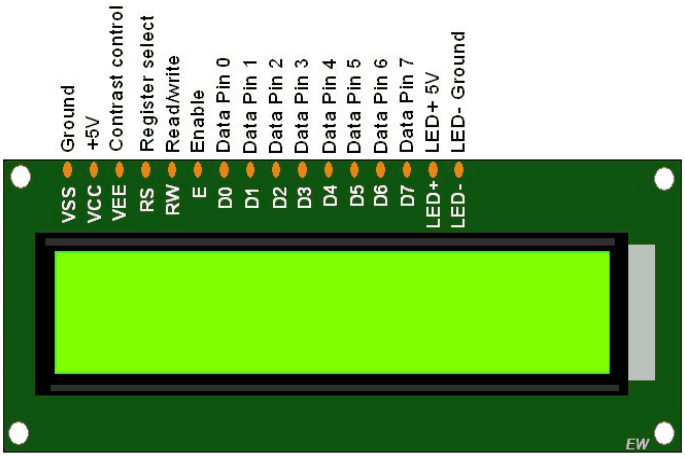


Illustration 29: documentation écran LCD

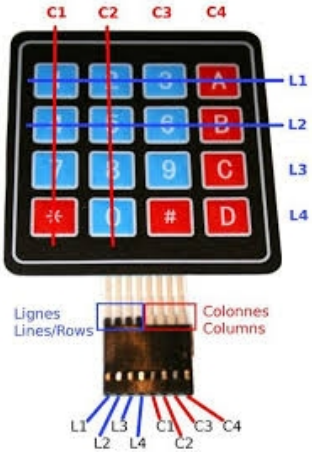


Illustration 28: Documentation calvier matriciel

Index des illustrations

Index des illustrations

Illustration 1: maquette vue intérieure....2
Illustration 2: maquette et composants....2
Illustration 3: Exemple de code scratch 1 avec notre première solution....5
Illustration 4: à l'intérieur du system broswer de scratch 1....5
Illustration 5: L'extension dans scratch2....8
Illustration 6: le shield....8
Illustration 7: Schéma synaptique des connections dans la maquette....8
Illustration 8: sensor_measure JS....10
Illustration 9: fonctions code arduino....11
Illustration 10: Schéma carte mère....12
Illustration 11: schéma de câblage shield maquette....13
Illustration 12: extrait de datasheet TS4871....13
Illustration 13: shéma plaque fille amplificateur audio....13
Illustration 14: choix de l'extension....14
Illustration 15: extension sas....15
Illustration 16: commande test i2c....17
Illustration 17: Vue théorique du shield ave Kicad en 3D....24
Illustration 18: logramme scénario part 2....25
Illustration 19: code scratch scénario 1....25
Illustration 20: logigramme scénario partie 1....25
Illustration 21: Code scratch scénario 2....26
Illustration 22: Code scratch scénario 3....26
Illustration 23: Ancien montage électronique....27
Illustration 24: Dimensions arduino....27
Illustration 25: Dimensions raspberry pi....27
Illustration 26: pins raspberry....27
Illustration 27: pins arduino....27
Illustration 28: Documentation calvier matriciel....28
Illustration 29: documentation écran LCD....28