



Open Electronics

Projet: Stepmania Duel

PHY_52064_EP

Mars 2024

**Léo Gabriel
Krah Espérance Marc Kouadio
Gabriel Pereira de Carvalho**

Introduction

Les jeux Stepmania sont des jeux de rythme où le joueur doit appuyer en accord avec le tempo sur les boutons flèches à ses pieds. Les bornes d'arcade Dance Dance Revolution de Konami sont les plus répandues, on peut en trouver dans des salles d'arcades partout dans le monde. Une borne Stepmania est également présente au Bataclan de l'École polytechnique. Alliant sport cardio, réflexes et amusement, ces jeux ont rassemblé une vaste communauté qui a produit de nombreuses créations, comme des émulateurs pour y jouer sur ordinateur ou encore des musiques personnalisées.



Figure 1: Borne d'arcade
Dance Dance Revolution

Souhaitant s'inspirer de ces jeux, le projet vise à créer un duel interactif entre deux joueurs, l'un pouvant créer les flèches et l'autre devant les toucher en rythme. Le projet se compose de deux éléments : une application pygame fonctionnant sur un Raspberry Pi 3 Model B+ ou ordinateur, et une manette sans fil utilisant un microcontrôleur ESP32 avec des boutons LED et un boîtier 3D réalisé avec Autodesk Fusion360. Une deuxième manette a aussi été partiellement fabriquée, mais les problèmes de communication évoqués plus tard nous ont fait envisager d'autres possibilités pour le joueur 2. Un joueur a également la possibilité de jouer contre l'aléatoire, car ici les *timings* tombent plus sur le tempo.

En tant que jeu de rythme, un enjeu significatif se porte sur la latence. Nous avons expérimenté différents protocoles de communication sans fil pour trouver une solution de latence minimale, et qui varie peu: l'exécution d'un broker MQTT sur Raspberry Pi avec un modèle de communication Pub-Sub, l'utilisation du protocole Bluetooth classique et des notifications GATT avec Bluetooth Low Energy (la solution qui a été choisie pour notre version finale). Notre priorité



INSTITUT
POLYTECHNIQUE
DE PARIS

était de maximiser le taux de rafraîchissement du jeu et de rendre les interactions des joueurs en temps réel aussi transparentes que possible.

On a aussi étudié la possibilité d'intégrer un deuxième joueur qui serait responsable pour placer les flèches du jeu. Ce joueur remplacerait l'approche randomisée qui est mise en place pour le jeu classique. Une section du rapport est dédiée aux difficultés rencontrées lors de ces tentatives et les différentes architectures et protocoles explorées pour l'implémenter.

Tout le code et les détails de la mise en œuvre du projet sont disponibles sur [Github](#). Nous avons également préparé une [vidéo de démonstration sur Youtube](#) où vous pouvez voir le projet en action! (mais sans musique)

Choix de technologie et architecture pour l'application démo

La technologie de communication utilisée pour le projet est le Bluetooth Low Energy (BLE), technologie de communication sans fil conçue pour transmettre des données sur de courtes distances tout en consommant très peu d'énergie. Il fonctionne sur un modèle client-serveur.

Dans l'optique de mettre en oeuvre notre jeu, du côté serveur nous avons :

- un ESP32 qui joue le rôle de serveur;
- 4 boutons LED reliés au serveur, qui correspondent aux différentes directions (gauche, droite, haut, bas) fonctionnant de telle sorte que chaque bouton pressé envoie un message via BLE au client tout en allumant la led correspondante au bouton.

Du côté client nous notons :

- Un Raspberry Pi jouant le rôle du client sur lequel est exécuté l'instance de jeu, qui génère des séquences de flèches aléatoires qui défilent à l'écran, reçoit les messages BLE du serveur, vérifie si le bouton a été pressé au bon moment et enregistre le score.

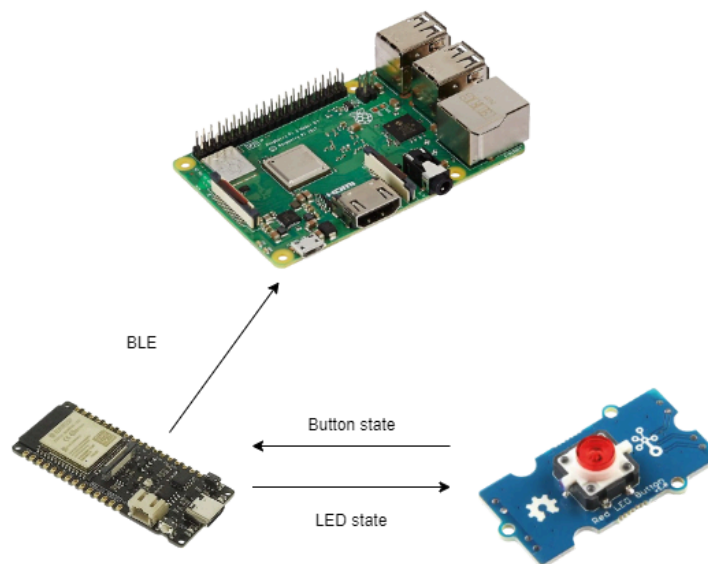


Figure 2: Architecture pour l'application de base

Les problèmes que nous voulions aborder dans le cadre de notre projet étaient les suivants :

- Effectuer une analyse comparative de la latence entre différents protocoles de communication sans fil.
- Travailler avec des interruptions liées à l'état des pins GPIO pour arrêter le flux d'exécution normal sur le contrôleur ESP32 et envoyer un message avec une latence minimale lorsqu'un bouton est pressé.

Au départ, nous pensions que pour maintenir une consistance dans les *timings*, il fallait que l'ESP puisse traiter les mouvements indépendamment du Raspberry Pi qui exécute le jeu principal ("simulation dans la manette"). Cette approche permet en théorie de s'affranchir des problèmes d'inconsistances de latence, car le Raspberry Pi enverrait les informations associées à chaque flèche à l'ESP32, mais l'ESP déciderait si un mouvement est bon ou non sans avoir besoin d'envoyer des messages au Raspberry Pi.

Le problème de la synchronisation sans fil des horloges fait l'objet d'une abondante littérature [1] [2] [3]. En effet, même si les deux appareils commencent à mesurer le temps à partir d'un même *offset* t_0 , les horloges auront toujours tendance à dériver parce que leurs taux ne sont pas constants (*clock drift*) et peuvent varier en raison de paramètres imprévisibles tels que la température ambiante et même la charge de travail effectuée sur l'appareil. La solution la plus courante consiste à obtenir des mises à jour périodiques du décalage à partir d'une horloge de haute précision via un serveur (*Network Time Protocol NTP* [3]).

Cependant, nos tests avec Bluetooth Low Energy ont été si bons en termes de latence (voir plus bas) que le temps d'envoyer une notification entre le Raspberry Pi et l'ESP32 était plus que suffisant pour effectuer le traitement des mouvements localement dans le Raspberry Pi. C'est pourquoi nous avons décidé de simplifier notre architecture et de ne pas recourir à des techniques de synchronisation d'horloge.

Au total, nous avons testé 3 protocoles différents pour la communication entre l'ESP32 et le Raspberry Pi : MQTT(implémenté sur WiFi), Bluetooth Classic et Bluetooth Low Energy.

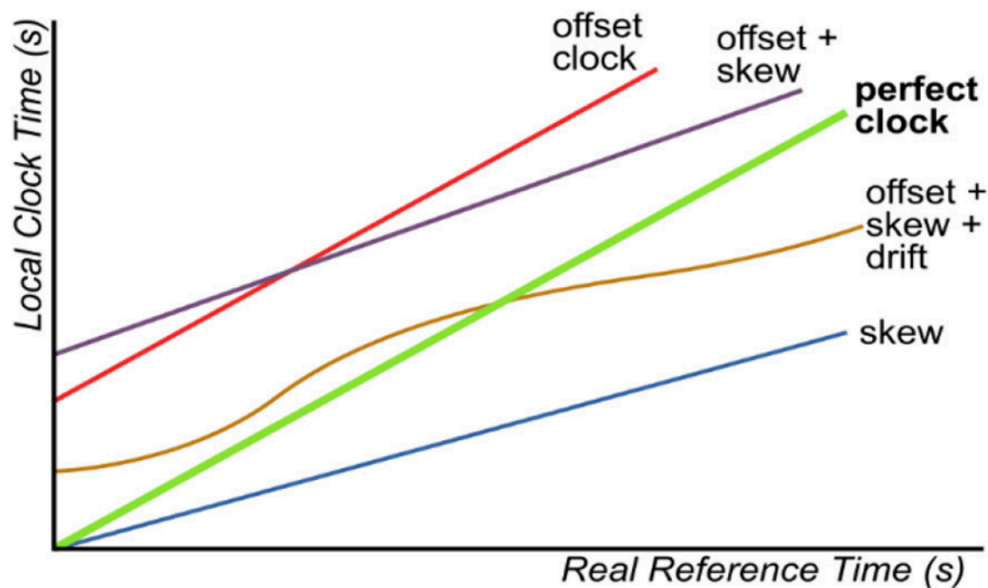


Figure 3: Plusieurs facteurs peuvent causer la dérive des clocks [2]

Nous avons utilisé la bibliothèque *Mosquitto* pour mettre en place un broker MQTT sur notre carte Raspberry Pi. MQTT est un protocole fonctionnant par messages. Les différents dispositifs peuvent s'abonner à un type de message ou en envoyer. Dans le modèle PubSub de MQTT, nous avons un *topic* pour chaque bouton/colonne du jeu. L'ESP32 agit en tant qu'éditeur sur chacun de ces topics, et les applications python se sont abonnées à chacun d'entre eux. De cette manière, les mouvements du joueur sont traités dans la fonction de réponse (*callback*) associée à chaque sujet.

Cette approche pose deux problèmes principaux :

- MQTT n'est pas un protocole fiable, des paquets pouvant être perdus occasionnellement, il est donc possible que les mouvements des joueurs soient perdus.
- la latence des messages MQTT était assez importante (de l'ordre de plusieurs centaines de millisecondes), mais surtout variait beaucoup (pouvant être de 150ms à 500ms de façon incontrôlée, ce qui constituait un gros problème pour notre application où le rythme est crucial).

Par la suite nous avons essayé de connecter les appareils via Bluetooth Classic avec la bibliothèque *bleak*, ce qui nous a permis d'obtenir des latences très faibles pour les messages (autour de 100ms), ce qui est correct surtout que cette latence est très constante.

Nous avons enfin étudié l'utilisation de **notifications GATT** dans le protocole Bluetooth Low Energy toujours possible via *bleak*, qui par ailleurs est conçu pour améliorer l'efficacité énergétique des appareils embarqués.

Quelles sont les différences entre le Bluetooth Classic et le Bluetooth Low Energy ?

La conception de Bluetooth Low Energy vise à fournir une portée de communication similaire à celle du Bluetooth Classic tout en diminuant la consommation d'énergie. Naturellement, cela implique des compromis en termes de débit de données et de sécurité.

La latence moyenne sur Bluetooth Classic est d'environ 100 ms alors que pour la Bluetooth Low Energy elle est d'ordre de 6 ms (imperceptible). L'avantage de l'utilisation du Bluetooth Classic est le débit de données, qui est d'environ 3 Mbps contre 1 Mbps pour Bluetooth Low Energy. C'est pourquoi Bluetooth Classic est toujours utilisé dans les applications audio et vidéo[4]. En conclusion, étant donné que nous transmettons très peu de données et que la latence est notre principale priorité, Bluetooth Low Energy est la technologie idéale pour notre projet.

Ajout d'un deuxième joueur

Une fois que notre démo a fonctionné avec un seul joueur. Nous avons décidé d'ajouter un deuxième joueur, chargé de contrôler les flèches dans le jeu. Ainsi, au lieu d'une fonction aléatoire générant périodiquement de nouvelles flèches, le jeu serait compétitif, un joueur essayant de battre l'autre avec des combinaisons de flèches de plus en plus difficiles.

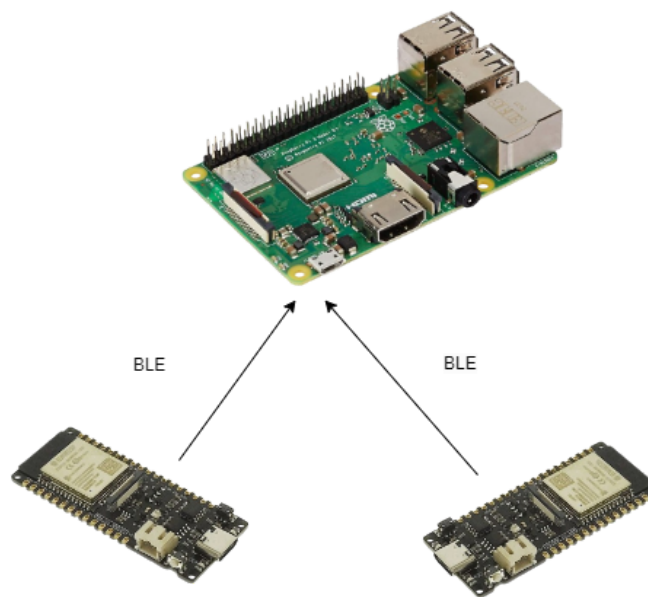


Figure 4: Dans la première architecture essayée, le deuxième joueur se connecte au jeu.

Nous avons commencé par utiliser une version légèrement modifiée du script que nous avons pour le premier joueur, qui était également basé sur une connexion BLE au Raspberry Pi. Cependant, en testant cette solution, nous avons réalisé que le Raspberry ne pouvait pas maintenir les deux connexions simultanément. Lorsqu'un joueur se connecte au jeu, la connexion avec le deuxième joueur semble être difficile.

En effet, le protocole Bluetooth permet de connecter plusieurs appareils à un seul appareil hôte simultanément même si le nombre d'appareils connectés peut varier en fonction de la

version du Bluetooth et des capacités matérielles de l'appareil, il devrait être possible de gérer deux connexions à la fois sur la Raspberry Pi. Nous avons plusieurs hypothèses quant à la raison de cela que nous n'avons pas forcément pu investiguer en détail:

- peut-être que la raison provient d'une maladresse d'implémentation côté ESP32 (qui est monthread)
- peut-être du côté du Raspberry Pi, le programme utilisant notamment des threads afin d'exécuter le jeu ainsi que les instances bluetooth en parallèle, tandis que notre implémentation du bluetooth fait usage de *asyncio*, bibliothèque de programmation non bloquante très contre intuitive.

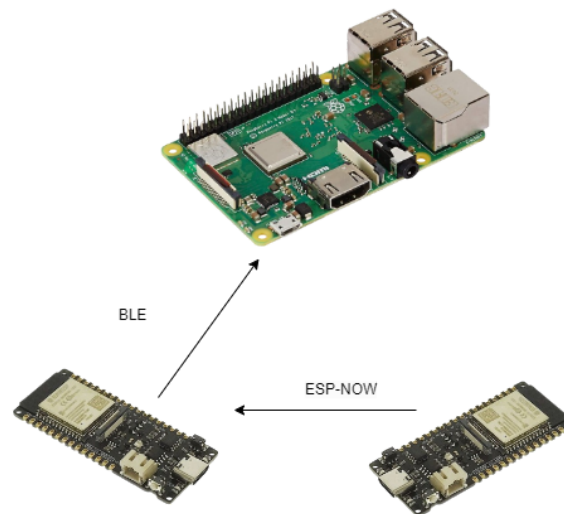


Figure 5: Dans la deuxième architecture essayée, une seule ESP32 reste connectée à la Raspberry Pi.

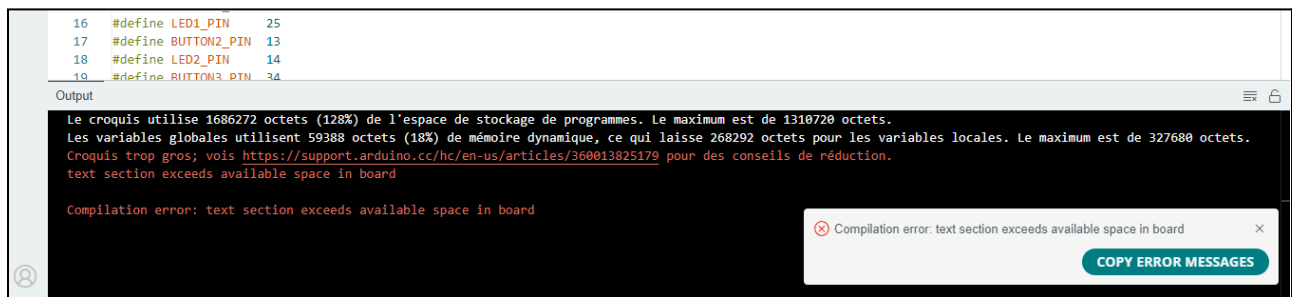
Pour contourner ce problème, nous avons décidé de changer l'architecture de notre application. Le deuxième joueur pourrait transmettre ses actions au premier via un protocole de communication sans fil et le premier devient responsable de transmettre l'ensemble de messages à la Raspberry Pi. La différenciation entre les deux ESP32 dans le script du jeu se fait par le préfixe du message, qui peut être facilement configuré et même utilisé pour ajouter encore d'autres joueurs à l'application.

Nous avons choisi d'utiliser le protocole ESP-NOW pour faire la communication entre les deux ESP32. Il s'agit d'un protocole de *layer 2*, qui a moins d'*overhead* que les protocoles

classiques normalement utilisés au niveau d'une application en *layer 5* et il a été spécifiquement conçu et optimisé pour l'ESP 32 (évite les surcouches permettant le transfert via canal quelconque).

Il est intéressant d'observer que l'adresse MAC utilisée ici par le protocole ESP-NOW n'est pas la même que l'adresse MAC utilisée pour la communication via Bluetooth (l'adresse MAC permet d'identifier les appareils bluetooth). Pour cette raison, nous étions bien convaincus de ne pas retrouver le même problème de l'architecture antérieure.

Alors qu'on a réussi à tester l'envoi et la réception de messages via ESP-NOW avec nos deux ESP 32, une fois cette fonctionnalité ajoutée dans notre script avec les bibliothèques nécessaires pour le Bluetooth Low Energy, la limite de mémoire du code de l'ESP 32 a été dépassée. Sans avoir à réécrire une partie significative du code afin d'économiser de l'espace mémoire, cette implémentation allait être ardue.



```
16 #define LED1_PIN 25
17 #define BUTTON2_PIN 13
18 #define LED2_PIN 14
19 #define BUTTON3_PIN 34
```

Output

Le croquis utilise 1686272 octets (128%) de l'espace de stockage de programmes. Le maximum est de 1310720 octets.
Les variables globales utilisent 59388 octets (18%) de mémoire dynamique, ce qui laisse 268292 octets pour les variables locales. Le maximum est de 327680 octets.
Croquis trop gros; voir <https://support.arduino.cc/hc/en-us/articles/360013825179> pour des conseils de réduction.
text section exceeds available space in board

Compilation error: text section exceeds available space in board

✖ Compilation error: text section exceeds available space in board

COPY ERROR MESSAGES

Figure 6: Erreur lors de la compilation.

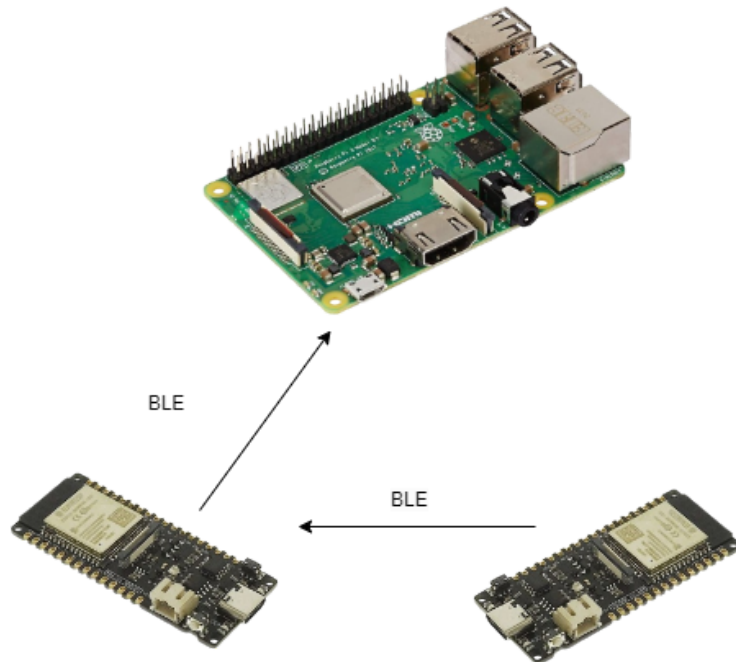


Figure 7: Troisième architecture, connexion indirecte mais via BLE aussi.

Dans une dernière tentative pour essayer d'ajouter le deuxième ESP32 au projet, nous avons étudié l'utilisation d'une connexion Bluetooth Low Energy au lieu du protocole ESP-NOW qui avait excédé la limite de mémoire. Cependant, ici on s'est retrouvé avec le même problème de notre architecture originale. L'ESP 32 du premier joueur n'arrive pas à se connecter à la fois au deuxième joueur et au Raspberry Pi.

Malgré tout, un deuxième joueur peut tout à fait participer en utilisant le clavier de l'ordinateur.

<pre> Start advertising Device Connected Device disconnected. Start advertising Device Connected Device disconnected. Start advertising Device Connected </pre>	<pre> clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00 mode:DIO, clock div:1 load:0x3fff0030,len:4688 load:0x40078000,len:15460 ho 0 tail 12 room 4 load:0x40080400,len:4 load:0x40080404,len:3196 entry 0x400805a4 Connecting to the server... </pre>
---	---

Figure 8 :Les multiples échecs de double connexion

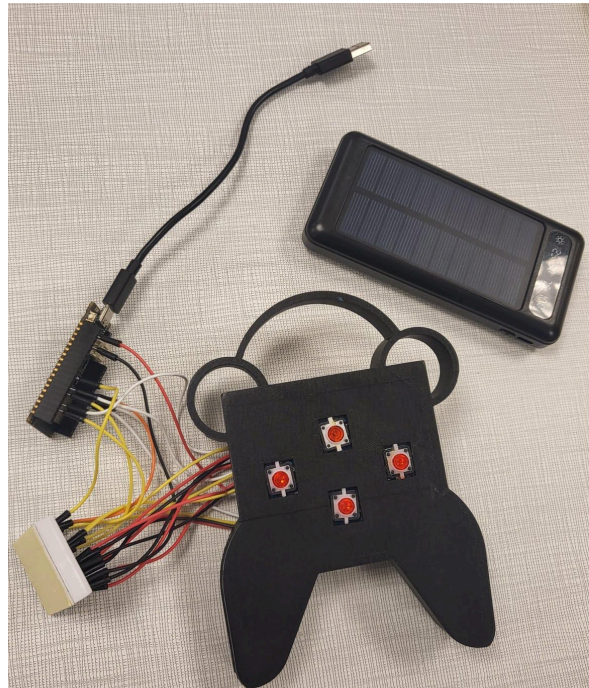


Figure 9 : Manette du joueur

Quelques détails sur les implémentations logicielles

Détails sur le code python

Le code du jeu, et plus généralement le code s'exécutant sur l'ordinateur ou le Raspberry

Pi, a été en majorité conçu sous python3. Cela a permis de très rapidement obtenir des prototypes, usant du confort apporté par les bibliothèques ainsi que la possibilité d'exécuter le même code sur ordinateur ou Raspberry Pi. Usant abondamment de l'orienté objet, le code a été pensé dès le début pour au maximum modulaire, permettant l'ajout ou l'activation/désactivation de fonctionnalités facilement.

Tout d'abord, le coeur de la logique et la boucle d'action est programmé au sein de la classe *Stepmania*, où est instancié les principaux objets (contrôleur bluetooth, images et audio,...), ainsi que la bibliothèque *pygame* qui implémente en particulier l'affichage d'images dans une fenêtre.

Une autre partie majeure du code concerne la connexion (facultative) des manettes. Ces connexions sont paramétrées dans la classe *BluetoothClient*, qui permet de se connecter à un appareil bluetooth, envoyer des messages ainsi que d'en recevoir. Sous réception d'un message, une fonction de "callback" peut être appelée permettant l'exécution d'une action, par exemple dans notre projet d'envoyer l'information qu'une flèche a été touchée. En terme de bibliothèques, *pybluez* permettait de facilement détecter les appareils bluetooth, *bleak* permettait de communiquer avec les appareils, et du code *asyncio* était requis pour le client *bleak*. La programmation asynchrone via *asyncio* est très particulière, il se peut qu'une mauvaise gestion

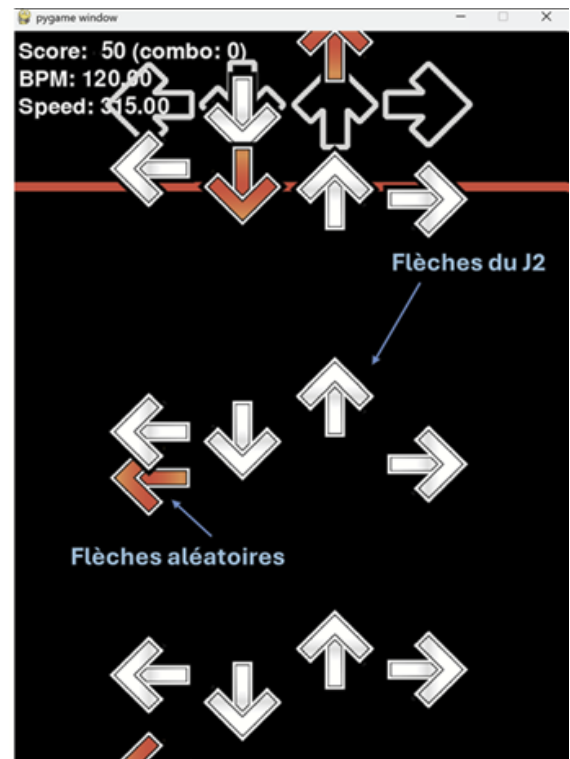


Figure 10 : Capture du jeu créé

du blocage par des tâches *scheduled* explique les difficultés de connexions que nous avons eu avec les manettes.

Finalement, afin d'avoir le jeu qui s'exécute tout en ayant des processus communiquant avec les manettes par exemple, la bibliothèque *threading* a été abondamment utilisée afin de créer des processus à l'exécution parallèle.

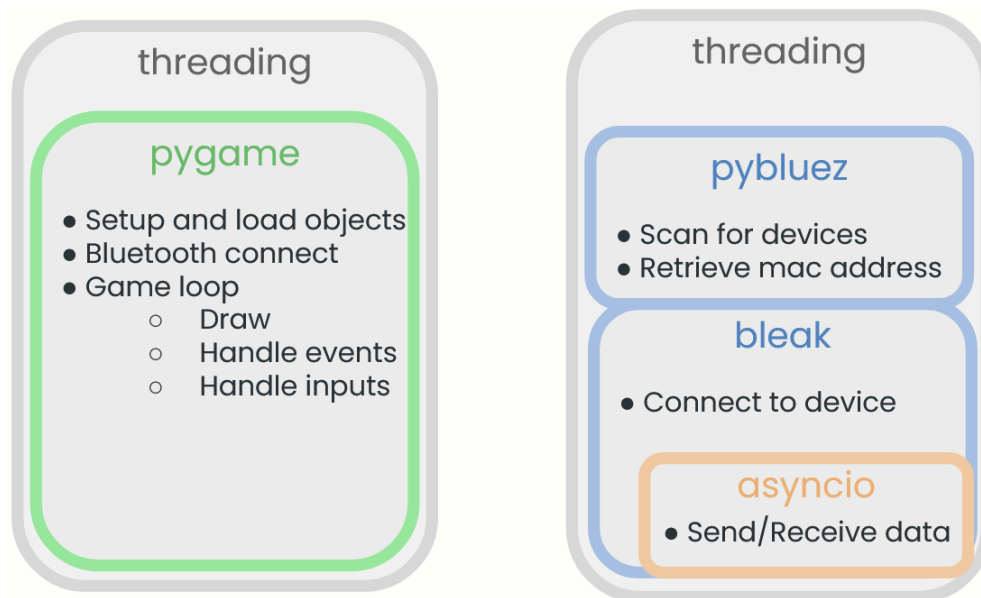


Figure 11 : Architecture du jeu python, ainsi que les bibliothèques utilisées

Détails sur le script exécuté sur ESP32

Le script exécuté dans l'ESP32 est beaucoup plus simple que l'application pygame que nous venons de décrire. Nous avons utilisé la bibliothèque *ESP32 BLE Arduino* pour avoir une implémentation de Bluetooth Low Energy sur l'ESP32. Une fois que nous avons découvert l'adresse MAC de l'ESP32 et défini des UUID uniques pour la transmission et la réception des données, il a été facile d'établir et de tester la connexion.

Pour traiter les mouvements du joueur avec le moins de latence possible, nous avons relié des interruptions aux broches GPIO du signal du bouton. De cette manière, la boucle du programme est immédiatement interrompue lorsque l'état du bouton change afin d'envoyer une notification au Raspberry Pi.

```
pinMode(BUTTON1_PIN, INPUT_PULLUP);
attachInterrupt(digitalPinToInterrupt(BUTTON1_PIN), handleButton1,
FALLING);
void IRAM_ATTR handleButton1() {
    sendMessage("HIT 1");
}
```

Références

- [1] Ulkar Aslanova, Leander Seidlitz, Jonas Andre (2023). **Wireless Time Synchronization in IEEE 802.11**. School of Computation, Information and Technology, Technical University of Munich, Germany.
- [2] Jack Henderson (2022). [Clock synchronisation over WiFi networks](#). University of Canberra.
- [3] David L. Mills (1991). **Internet Time Synchronization: The Network Time Protocol**. IEEE Transactions on Communications.
- [4] Nordic Semiconductors (2021). [The Difference Between Classic Bluetooth and Bluetooth Low Energy](#). Get Connected Blog.

Bibliothèques clés utilisées

python3

- *pygame*: Python Game Development, Version 2.6.1 [Online]. Available: <https://github.com/pygame/pygame>
- *pybluez* : Bluetooth Python extension module, Version 0.46 [Online]. Available: <https://github.com/pybluez/pybluez>
- *bleak*: Cross platform BLE Client for Python using asyncio, Version 0.22.3 [Online]. Available: <https://github.com/hbldh/bleak>
- bibliothèques “built-in”: *numpy*, *threading*, *pathlib*, *asyncio*, *time*, *typing*
- bibliothèque [ESP32 BLE Arduino](#)