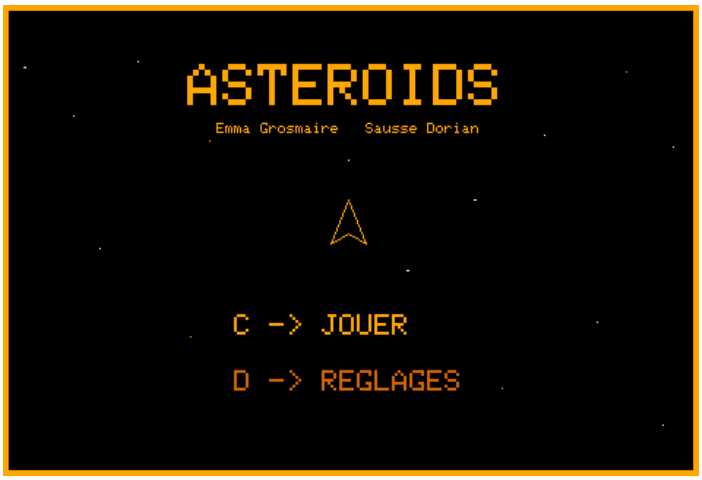
RAPPORT DE PROJET
SAE (2e semestre)

Conception et réalisation d'un système embarqué interactif:

Jeu vidéo « ASTEROIDS » sur microcontrôleur PIC16F18877



Auteurs : Emma Grosmaire & Dorian Sausse

Période : 2e semestre & Année universitaire 2025–2026

Date de rédaction : juin 2026

Outils et technologies : PIC16F18877 · MPLAB X IDE · compilateur XC8 · MPLAB Code Configurator (MCC) · écran TFT ILI9488 (SPI) · KiCad 9 (carte son LM386) · SolidWorks (modélisation 3D)

Résumé

Ce rapport présente la conception et la réalisation complète d'un système embarqué interactif prenant la forme d'un jeu vidéo de type *Asteroids*. Le projet, mené en binôme, intègre quatre sous-ensembles articulés autour d'un microcontrôleur **PIC16F18877**: une **interface d'affichage** reposant sur un écran **ILI9488** (480 × 320), une **manette** combinant un joystick analogique lu par le convertisseur **ADC** et des boutons poussoirs, une **carte son** dédiée conçue sous KiCad autour de l'amplificateur **LM386** par nos soin, et le **logiciel embarqué** développé sous MPLAB X ainssi que MCC pour les empreintes.

Il détaille l'ensemble de la chaîne d'ingénierie : la modélisation mécanique 3D sous SolidWorks, la conception électronique de la carte son (schéma, choix des composants, routage du circuit imprimé 45 × 45 mm, fabrication et brasage), la mise en œuvre des périphériques du microcontrôleur et, surtout, le développement logiciel expliqué **fonction par fonction**. Une attention particulière est portée aux **fondements mathématiques** transposés en langage C: l'orientation du vaisseau par normalisation vectorielle (qui fournit directement le couple cosinus / sinus sans calcul d'angle), la rotation par formules d'addition d'angles avec constantes précalculées, la trajectoire des astéroïdes vers le centre, et la détection de collisions par comparaison de distances au carré. .

Le prototype final est fonctionnel: il valide l'intégration matérielle et logicielle de l'ensemble des sous-systèmes.

Table des matières

Liste des figures (images)

1. Introduction et contexte du projet

1.1 Contexte du projet

Ce rapport présente une situation d'apprentissage et d'évaluation (SAE) menée au cours du deuxième semestre. L'objectif pédagogique d'une telle SAE est de confronter les étudiants à un projet technique complet, de la conception à la réalisation, en mobilisant simultanément les compétences acquises en électronique, en informatique embarquée et en conception assistée par ordinateur. Le travail a été réalisé en binôme par Emma Grosmaire et Dorian Sausse (groupe C).

Le système demandé est un système embarqué autonome intégrant trois grands ensembles : une interface utilisateur (écran graphique et manette), une carte son dédiée, et un microcontrôleur jouant le rôle d'unité centrale. Afin de donner un fil conducteur concret et motivant à l'ensemble des développements, le binôme a choisi de réaliser un jeu vidéo de type *Asteroids* comme support pédagogique. Ce choix n'est pas anodin : un jeu d'arcade impose des contraintes fortes et représentatives du génie embarqué, à savoir l'affichage graphique en temps réel, la lecture de capteurs (joystick, boutons), la génération sonore, et une boucle de traitement qui doit rester fluide malgré des ressources matérielles limitées.

Le principe du jeu est simple à énoncer mais riche à implémenter. Un vaisseau triangulaire reste fixé au centre de l'écran et pivote sur place pour viser, l'orientation étant commandée par le joystick analogique. Le joueur tire des projectiles à l'aide du bouton B. Des astéroïdes, représentés par des octogones, surgissent des bords de l'écran et foncent vers le centre. Détruire un astéroïde rapporte dix points au score ; en revanche, si un astéroïde atteint le vaisseau, la partie se termine. L'application est organisée autour de quatre écrans gérés par une machine à états : un menu d'accueil (ETAT_MENU), un écran de réglages décrivant les commandes (ETAT_REGLAGES), l'écran de jeu proprement dit (ETAT_JEU), et un écran de fin de partie ou *game over* (ETAT_GAMEOVER).

Figure 1 — vue d'ensemble du système : microcontrôleur PIC, écran ILI9488, manette à joystick et carte son.

1.2 Objectifs

Le projet poursuit plusieurs objectifs imbriqués, qui couvrent l'ensemble de la chaîne de conception d'un système embarqué :

- **Concevoir l'architecture matérielle globale** : définir comment les différents sous-ensembles (calculateur, affichage, commandes, audio, alimentation) communiquent entre eux, et répartir les fonctions sur les broches du microcontrôleur.
- **Développer une carte son dédiée** : concevoir et fabriquer un amplificateur audio basse tension autour du circuit intégré LM386, depuis le schéma jusqu'au circuit imprimé.
- **Mettre en œuvre le microcontrôleur PIC16F18877** : configurer ses périphériques (convertisseur analogique-numérique ADCC, bus de communication série SPI, oscillateur numériquement commandé NCO) et écrire le programme applicatif.
- **Réaliser une interface utilisateur** combinant un écran graphique couleur TFT et une manette dotée d'un joystick analogique et de boutons-poussoirs.
- **Développer le logiciel embarqué** : programmer la logique de jeu, le rendu graphique et la gestion du son dans un environnement contraint.
- **Intégrer l'ensemble** en un prototype fonctionnel, testé et démontrable, où matériel et logiciel coopèrent de façon fiable.

Ces objectifs traduisent la volonté de produire non pas une simple maquette de laboratoire, mais un système cohérent et utilisable, illustrant la démarche d'ingénierie : analyse du besoin, conception, réalisation, intégration et validation.

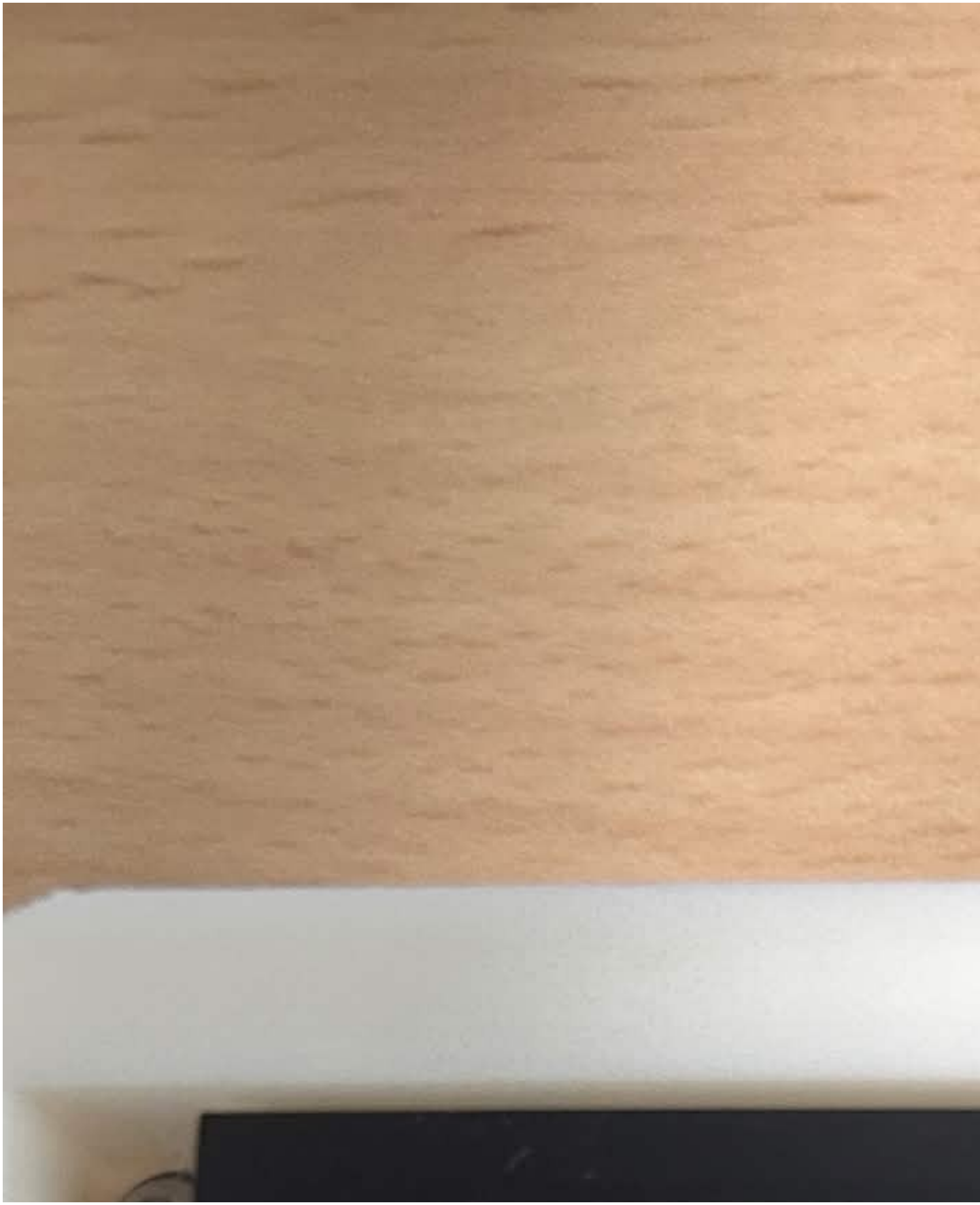


Figure 2 — prototype en fonctionnement : le menu d'accueil affiché à l'écran.

1.3 Cahier des charges et contraintes

Le cahier des charges peut se décliner en exigences fonctionnelles, matérielles et de contexte.

Exigences fonctionnelles. Le système doit offrir une jouabilité fluide : la rotation du vaisseau, le tir des projectiles et le déplacement des astéroïdes doivent réagir sans latence perceptible. L'interface doit être lisible (bande de score, ou HUD, en haut de l'écran, retours visuels et sonores) et la navigation entre les quatre écrans doit être intuitive, à l'aide de boutons-poussoirs dédiés (jouer, réglages, retour au menu).

Exigences matérielles. Le matériel est imposé et fixe le cadre technique du projet. L'unité centrale est un microcontrôleur PIC16F18877 de Microchip, en boîtier 40 broches PDIP, programmé sous MPLAB X avec le compilateur XC8 et l'outil de configuration MCC (MPLAB Code Configurator), qui génère l'initialisation des périphériques. L'affichage est confié à un écran TFT ILI9488 de 480 × 320 pixels en mode paysage, piloté par le bus SPI (MSSP1 en maître, mode 0) avec une couleur codée sur 18 bits. La commande est assurée par un joystick analogique deux axes lu par le convertisseur analogique-numérique ADCC sur 10 bits (valeurs de 0 à 1023), complété par des boutons-poussoirs actifs à l'état bas (résistance de tirage *pull-up*). Enfin, le son est produit par le périphérique NCO1 (oscillateur numériquement commandé) du PIC, dont le signal est amplifié par la carte son à base de LM386.

Contraintes liées au microcontrôleur. Ce sont elles qui structurent l'ensemble des choix logiciels et constituent le fil rouge du projet. Le PIC16F18877 est un cœur 8 bits cadencé par l'oscillateur interne HFINTOSC à FOSC = 32 MHz, soit un cycle d'instruction Tcy = FOSC/4 = 8 MHz (125 ns par cycle). Surtout, il ne dispose d'aucune unité de calcul en virgule flottante matérielle (pas de FPU) : toute opération sur des nombres réels (cosinus, sinus, racine carrée) est émulée par logiciel et donc lente. La mémoire vive disponible est par ailleurs réduite, ce qui motive l'emploi de types entiers de taille fixe (uint8_t, uint16_t) pour économiser la RAM. Ces deux limites — absence de FPU et peu de RAM — imposent une discipline de programmation rigoureuse, déclinée en quatre principes que l'on retrouvera tout au long du rapport : éviter au maximum les calculs flottants en cours de jeu (pas de atan2, cos ni sin en boucle, mais le vecteur normalisé du joystick et des constantes précalculées) ; ne jamais

bloquer inutilement le déroulement du jeu ; comparer les distances au carré plutôt que de calculer des racines carrées ; et n'effectuer le rendu graphique que sur les éléments qui changent réellement, selon le cycle effacer / déplacer / redessiner.

Contraintes de coût, d'encombrement et de consommation. Le projet vise un coût matériel modéré, cohérent avec un projet de formation, et privilégie des composants traversants (THT) faciles à braser à la main. L'encombrement reste celui d'un prototype de paillasse, avec une carte son compacte conçue pour tenir sur un petit circuit imprimé double face de 45 mm × 45 mm. La consommation, alimentée en basse tension (le PIC est alimenté en +5 V), doit rester compatible avec une alimentation de laboratoire ou un bloc autonome simple.

1.4 Organisation du projet et répartition des tâches

Le projet a été conduit en binôme selon une méthode itérative : plutôt que de figer entièrement le système avant de commencer, le binôme a procédé par cycles courts (conception partielle, réalisation, test, correction), ce qui permet d'identifier tôt les difficultés — notamment celles liées aux performances du microcontrôleur — et d'ajuster les choix en conséquence.

La répartition des tâches s'est appuyée sur les affinités de chacun tout en maintenant une part importante de travail commun, en particulier sur l'intégration et la validation. Une répartition cohérente avec le déroulement du projet est la suivante (répartition indicative, à confirmer) :

- **Conception mécanique et CAO (SolidWorks)** : modélisation du support et de l'enveloppe accueillant l'écran, la manette et la carte son, vérification des dimensions et des dégagements (cotes exactes à confirmer).
- **Conception électronique (KiCad)** : schéma et routage de la carte son LM386 sous KiCad 9, génération des fichiers de fabrication (Gerbers : couches cuivre F_Cu et B_Cu, masques, sérigraphie, perçages), nomenclature des composants.
- **Programmation (MPLAB X)** : configuration des périphériques du PIC via MCC, développement de la logique de jeu, du rendu graphique et de la génération sonore.
- **Intégration et tests** : assemblage des sous-ensembles, câblage, contrôle visuel, recherche de courts-circuits et mesure de continuité sur la carte son, essais de jouabilité et corrections.

Dans la pratique, les phases de programmation et d'électronique ont été menées de front, le

travail étant régulièrement mis en commun afin que chacun maîtrise l'ensemble du système. Les tâches d'intégration et de test, par nature transversales, ont été partagées entre Emma Grosmaire et Dorian Sausse.

Un mini-planning prévisionnel peut être décomposé en cinq phases successives, qui se recouvrent partiellement du fait de l'approche itérative :

1. **Analyse et architecture** : étude du besoin, choix de l'architecture matérielle et du brochage du PIC.
2. **Conception électronique et mécanique** : tracé de la carte son sous KiCad, modélisation du support sous SolidWorks.
3. **Fabrication et câblage** : brasage des composants traversants, assemblage de la manette, raccordement de l'écran et de la carte son.
4. **Développement logiciel** : mise en place des bibliothèques d'affichage (pilote bas niveau ILI9488 et bibliothèque graphique GFX), programmation de la boucle de jeu, intégration du son.
5. **Intégration, tests et finalisation** : validation du prototype complet, corrections, rédaction du présent rapport.

Cette organisation, couplée à la démarche itérative, a permis de faire converger progressivement les sous-ensembles vers un prototype fonctionnel, tout en gardant à l'esprit la contrainte centrale du projet : tirer le meilleur parti d'un microcontrôleur modeste pour obtenir un jeu réactif et agréable à jouer.

2. Architecture generale du systeme

Ce chapitre presente une vue d'ensemble du systeme realise pour le jeu de type Asteroids embarque. L'objectif est de decrire comment les differents elements materiels s'organisent autour d'un microcontroleur unique, comment l'information circule entre eux et selon quel principe logiciel le jeu est pilote. Cette description constitue le socle sur lequel s'appuieront les chapitres suivants, plus detailles, consacres au logiciel graphique, aux mathematiques du jeu et a la carte son.

2.1 Presentation generale

Le systeme est construit autour d'un microcontroleur PIC16F18877 de Microchip, un composant 8 bits en boitier 40 broches PDIP, qui joue le role de chef d'orchestre. Toute l'intelligence du jeu reside dans ce seul circuit : il lit les commandes du joueur, calcule la position des objets a l'ecran, dessine l'image et declenche les sons. Le microcontroleur fonctionne sur son oscillateur interne HFINTOSC a une frequence FOSC de 32 MHz, ce qui correspond a un cycle d'instruction de FOSC/4 = 8 MHz, soit un temps de cycle Tcy de 125 ns. Il faut garder a l'esprit que ce PIC16 ne possede pas d'unite de calcul en virgule flottante materielle : les operations sur les nombres reels (cos, sin, sqrt) sont donc emulees de maniere logicielle et restent lentes. Cette contrainte est le fil rouge de toute la conception, comme on le verra dans les chapitres dedies au logiciel.

Autour de ce coeur, on distingue quatre sous-ensembles relies au PIC :

- l'ecran TFT ILI9488, qui affiche le jeu, pilote par le bus SPI ;
- la manette de commande (joystick a deux axes plus boutons poussoirs), lue par le convertisseur analogique-numerique et des entrees tout-ou-rien ;
- la carte son a base de LM386, attaquee par la sortie du peripherique NCO1 ;
- l'alimentation 5 V, qui fournit l'energie a l'ensemble.

Le schema bloc ci-dessous resume cette organisation et place le PIC au centre des echanges.

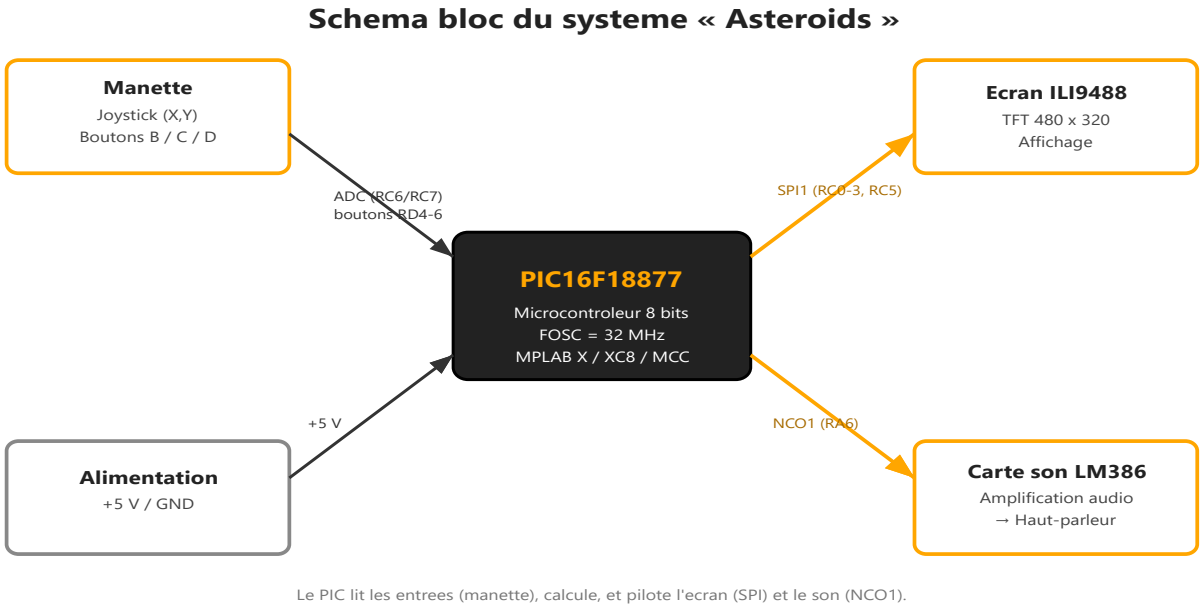


Figure 3 — Architecture generale : le PIC16F18877 au centre relie a l'ecran, a la manette, a la carte son et a l'alimentation.

2.2 Les sous-systemes

Module de controle (le PIC16F18877). C'est l'unite centrale du systeme. Programme en langage C sous l'environnement MPLAB X IDE avec le compilateur XC8, il a vu l'initialisation de ses peripheriques generee a l'aide de l'outil MCC (MPLAB Code Configurator). Le microcontroleur execute en continu la boucle de jeu : lecture des entrees, mise a jour de l'etat des objets, rendu graphique et gestion du son. Il est alimente en +5 V (pattes 11 et 32), sa masse VSS est reliee aux pattes 12 et 31, et son entree de reinitialisation MCLR se trouve sur la patte 1. Le recours a MCC est un choix methodologique important : l'outil produit automatiquement les fonctions d'initialisation (SYSTEM_Initialize, configuration de l'ADCC, ouverture du SPI1, definition du NCO1 et des macros de broches), ce qui fiabilise le parametrage des registres et laisse au binome le temps de se concentrer sur la logique du jeu.

Module d'affichage (l'ecran TFT ILI9488). L'ecran est un afficheur couleur de resolution native 320x480 pixels, utilise ici en mode paysage (setRotation(3)) soit 480x320 pixels exploitables. La couleur est transmise sur 18 bits, sous la forme de trois octets correspondant aux composantes rouge, verte et bleue. L'ecran est commande par le bus SPI1 (peripherique MSSP1) configure en maitre, en mode SPI 0. La communication est a sens unique : seul le PIC ecrit vers l'ecran via la ligne MOSI. La ligne MISO de retour existe physiquement sur le

module mais n'est pas exploitee par le programme, ce qui simplifie le cablage et le code. Au-dela des trois lignes du bus serie proprement dit (SCK, MOSI et la selection de boitier CS), l'afficheur reclame deux signaux de controle particuliers : la ligne DC (Data/Command), qui indique si l'octet transmis est une commande ou une donnee, et la ligne RST de reinitialisation materielle, utilisee lors de la sequence d'initialisation.

Module de commande (la manette). Le joueur agit sur le jeu au moyen d'un joystick analogique a deux axes et de boutons poussoirs. Les deux axes du joystick sont lus par le convertisseur ADCC sur 10 bits, fournissant des valeurs comprises entre 0 et 1023 : l'axe X est cable sur RC7 (patte 22) et l'axe Y sur RC6 (patte 21). Les boutons poussoirs sont actifs a l'etat bas, grace a une resistance de tirage (pull-up) : un bouton relache renvoie 1, un bouton appuye renvoie 0. Le bouton B (RD6, patte 29) commande le tir, le bouton C (RD5, patte 28) sert a jouer, revenir ou retourner au menu, et le bouton D (RD4, patte 27) ouvre les reglages. Le bouton A (RD7, patte 30) n'est pas utilise dans cette version.

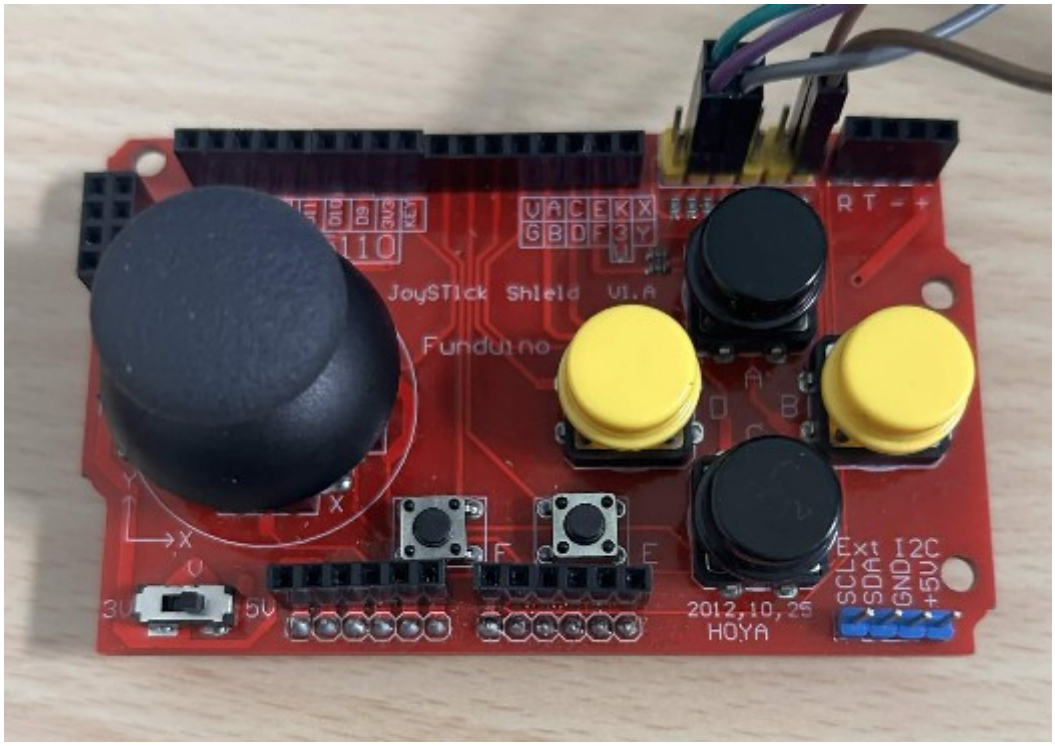


Figure 4 — Manette du jeu cablee : joystick a deux axes et boutons poussoirs relies au microcontroleur.

Module audio (la carte son). Le son est produit par le peripherique NCO1 (Numerically Controlled Oscillator) du PIC, dont la sortie est dirigee vers la patte RA6 (patte 14). Ce signal

carre est ensuite amplifie par une carte son dediee, concue specifiquement pour le projet sous KiCad 9. Il s'agit d'un amplificateur audio classique a base du circuit integre LM386 (boitier DIP-8), avec reglage de volume par un potentiometre de 10 kOhm en entree et un reseau de stabilisation en sortie. Le gain de l'etage est fixe a 20, valeur par defaut du LM386 obtenue en laissant les broches 1 et 8 non pontees. La carte, de format 45 mm x 45 mm en double face, regroupe quatre blocs fonctionnels : l'alimentation et son decouplage, l'etage d'entree avec couplage et reglage de volume, l'amplification proprement dite et l'etage de sortie. La photographie de rendu 3D ci-dessous montre cette carte avant brasage.

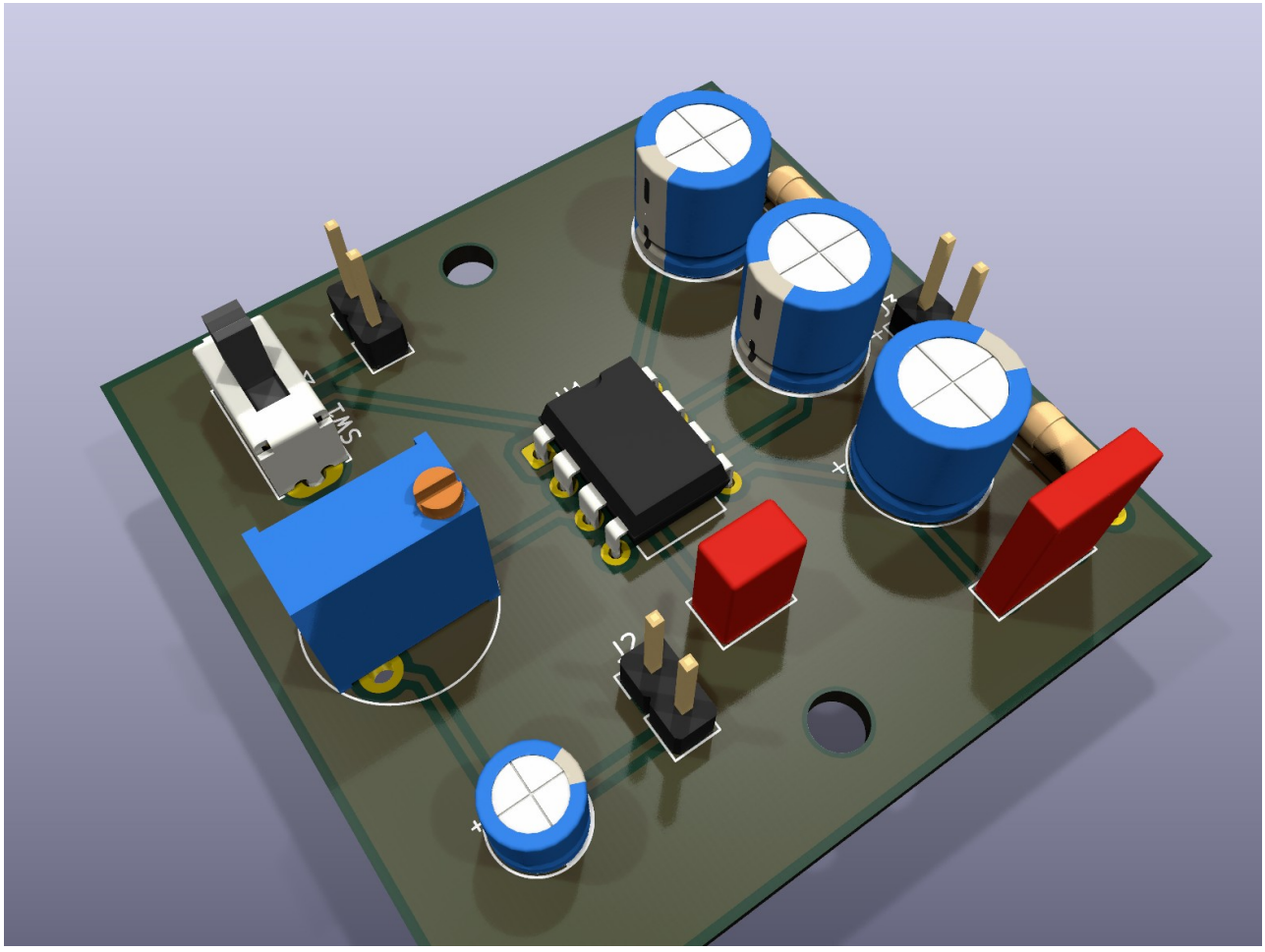


Figure 5 — Vue 3D isometrique de la carte son a base de LM386, generee sous KiCad.

2.3 Flux de donnees et interconnexions

Le fonctionnement du systeme repose sur une circulation ordonnee de l'information entre la manette, le PIC, l'ecran et la carte son. On peut le decrire selon le sens des echanges.

En entree, la position du joystick est transformee par le convertisseur analogique-numerique ADCC en deux valeurs numeriques (axe X et axe Y). Le programme exploite ces valeurs pour determiner directement la direction de visee du vaisseau, sans recourir a des fonctions trigonometriques couteuses : le vecteur issu du joystick, une fois normalise, fournit deja le couple (cos, sin) de l'angle de pointage. Toujours en entree, les boutons poussoirs sont lus comme de simples entrees tout-ou-rien, et leur etat bas declenche les actions correspondantes (tir, navigation entre les ecrans, acces aux reglages).

En sortie, le PIC produit deux flux distincts. Le premier est le flux video : le microcontroleur envoie, via le bus SPI, les commandes et les donnees de pixels vers l'ecran ILI9488, qui affiche le jeu. Le second est le flux audio : le PIC pilote le peripherique NCO1, dont le signal sort sur RA6, traverse la carte son (couplage d'entree, reglage de volume, amplification par le LM386, reseau de Zobel et couplage de sortie) avant d'attaquer le haut-parleur. L'alimentation 5 V, transverse, fournit l'energie a l'ensemble et n'intervient pas dans le traitement de l'information.

Le peripherique NCO1 merite une precision sur le plan numerique. Sa source d'horloge est FOSC/4, soit 8 MHz, et son accumulateur est code sur 20 bits. La frequence de sortie obeit a la relation :

`f = (8 000 000 x NCO1INC) / 2^20 = NCO1INC x 7,6294 Hz`

Pour calculer rapidement la valeur a charger dans le registre, le code ecrit `NCO1INC = frequence >> 3`, c'est-a-dire une division par 8 par simple decalage de bits. Le ton effectivement joue vaut alors environ `frequence x 0,9537`, donc tres proche de la frequence demandee. Le decalage `>>3` (qui revient a multiplier par 0,125) est une approximation rapide du facteur exact 0,1311, choisie parce qu'un decalage de bits s'execute en une instruction la ou une vraie division serait nettement plus couteuse sur ce microcontroleur 8 bits. On retrouve ici le souci constant de menager un processeur lent.

Le tableau ci-dessous recapitule l'ensemble des liaisons entre le PIC et ses sous-systemes.

Sous-systeme	Type de liaison	Broches du PIC (patte)
Ecran ILI9488	Bus SPI1 (maitre, mode	CS=RC0(15), RST=RC1(16), DC=RC2(17),

Sous-systeme	Type de liaison	Broches du PIC (patte)
	0)	SCK=RC3(18), MOSI=RC5(20)
Joystick (axes analogiques)	Entrees ADC 10 bits	X=RC7(22), Y=RC6(21)
Boutons poussoirs	Entrees TOR, actives a l'etat bas (pull-up)	B/TIR=RD6(29), C/MENU=RD5(28), D/REGLAGES=RD4(27), A non utilise=RD7(30)
Carte son	Sortie NCO1	SON=RA6(14)
Alimentation	Continue 5 V	VDD=11 et 32, VSS=12 et 31, MCLR=1

2.4 Principe de fonctionnement du jeu et machine a etats

Le jeu reprend le principe d'Asteroids adapte a la contrainte d'un affichage centre. Le vaisseau, represente par un triangle, reste fixe au centre de l'ecran (coordonnees CX=240, CY=172) et ne fait que pivoter sur place pour viser, sous la commande du joystick. Lorsque le joueur appuie sur le bouton de tir, le vaisseau emet des projectiles dans la direction visee. Des asteroides, dessines sous forme d'octogones, surgissent des bords de l'ecran et foncent vers le centre. Detruire un asteroide rapporte 10 points au score ; en revanche, si un asteroide atteint le vaisseau, la partie est perdue.

L'ensemble du deroulement est organise par une machine a etats comportant quatre ecrans distincts, decrits par le type enumere `EtatJeu` :

- ETAT_MENU : l'ecran d'accueil, ou le joueur peut lancer une partie ou consulter les reglages ;
- ETAT_REGLAGES : l'ecran qui rappelle les commandes du jeu ;
- ETAT_JEU : la partie proprement dite, ou s'execute la boucle de rendu et de simulation ;
- ETAT_GAMEOVER : l'ecran de fin de partie, affichant le resultat.

La declaration de ce type enumere se presente ainsi :

```
typedef enum { ETAT_MENU, ETAT_REGLAGES, ETAT_JEU, ETAT_GAMEOVER } EtatJeu;
```

Cette ligne definit un nouveau type `EtatJeu` qui ne peut prendre que l'une des quatre valeurs enumerees. En interne, le compilateur leur attribue des entiers consecutifs (0 a 3), ce qui permet de stocker l'etat courant dans une simple variable de petite taille et de comparer ou commuter dessus tres efficacement. Ce choix evite de manipuler des chaines de caracteres ou des indicateurs multiples, et reste donc econome en RAM et en temps de calcul, conformement au fil rouge du projet.

La navigation entre ces etats s'effectue au moyen des boutons C (jouer, retour, menu) et D (reglages). Cette structure en machine a etats permet de separer clairement les differentes phases de l'application et de n'executer, a chaque instant, que le code pertinent pour l'ecran courant. Elle constitue l'ossature logicielle du programme, dont le detail (boucle de jeu, rendu incremental, detection des collisions) sera developpe dans les chapitres suivants. Conformement au fil rouge du projet, cette organisation vise toujours le meme objectif : tirer le meilleur parti d'un microcontroleur 8 bits aux ressources limitees, en evitant les calculs inutiles et en ne redessinant que ce qui change reellement a l'ecran. Cette derniere idee, le rendu incremental, est essentielle : plutot que de repeindre toute l'image a chaque tour de boucle (ce qui saturerait le bus SPI), le programme efface, deplace puis redessine uniquement les objets ayant change de position, ce qui limite le volume de donnees transmis a l'ecran et preserve la fluidite du jeu.

3. Le microcontroleur PIC16F18877 et sa configuration (MCC)

Le coeur du systeme embarque est le microcontroleur PIC16F18877 du fabricant Microchip. C'est lui qui execute le programme du jeu, lit le joystick et les boutons, pilote l'ecran TFT et genere le son. Avant d'aborder le logiciel proprement dit, ce chapitre decrit le composant choisi, son horloge, les peripheriques materiels mobilises et l'outil de configuration MPLAB Code Configurator (MCC) qui en facilite la mise en oeuvre. Cette presentation est essentielle pour comprendre une contrainte qui structure tout le reste du projet : le PIC16F18877 est un microcontroleur 8 bits relativement lent, depourvu d'unite de calcul flottant, et l'ensemble du programme est concu pour menager ses ressources.

3.1 Presentation du PIC16F18877

Le PIC16F18877 appartient a la famille PIC16F1 de Microchip. Il s'agit d'un microcontroleur 8 bits, ce qui signifie que son unite de traitement manipule nativement des donnees codees sur 8 bits (un octet). Le composant retenu se presente dans un boitier de 40 broches au format PDIP (Plastic Dual In-line Package), c'est-a-dire un boitier traversant a deux rangees de pattes, parfaitement adapte au prototypage sur carte a souder. Ce grand nombre de broches offre un confort appreciable : il reste suffisamment d'entrees/sorties pour cabler simultanement l'ecran (bus SPI plus les lignes de controle), le joystick analogique (deux axes), les boutons poussoirs et la sortie son, sans avoir a multiplexer les signaux entre plusieurs fonctions.

Comme tout microcontroleur, le PIC16F18877 integre sur une meme puce un processeur, de la memoire et des peripheriques. Son architecture est de type Harvard : la memoire programme et la memoire de donnees sont physiquement separees et accessibles par des bus distincts. La memoire programme est une memoire Flash non volatile, dans laquelle le code reste grave apres coupure de l'alimentation ; la memoire de donnees (RAM) sert aux variables pendant l'execution et perd son contenu a la mise hors tension. Ces deux memoires sont limitees au regard d'un ordinateur classique, ce qui impose une certaine sobriete. C'est notamment pour economiser la RAM que le code emploie systematiquement des types entiers de taille fixe (`uint8_t`, `uint16_t`, `int8_t`) issus de `<stdint.h>` : choisir explicitement un entier sur 8 bits la ou un octet suffit (par exemple l'indicateur `active` d'un projectile) evite de consommer inutilement des octets precieux. Le jeu d'instructions du coeur est par ailleurs reduit : c'est un

processeur de type RISC, qui ne dispose que d'operations elementaires sur des entiers (chargement, addition, decalage, operations logiques, branchements). Il n'existe aucune instruction materielle de multiplication etendue ou de calcul flottant : toute operation plus complexe est decomposee en une suite d'instructions simples par le compilateur.

Le point le plus determinant pour ce projet est justement l'absence d'unite de calcul flottant materielle (FPU). Le PIC16 ne sait pas calculer directement sur des nombres a virgule flottante : chaque operation en `float` (addition, multiplication, et plus encore `cos`, `sin`, `sqrt`) est emulee par une bibliotheque logicielle, donc lente, car traduite en de longues sequences d'instructions entieres. Cette limitation est le fil rouge de toute la conception logicielle : on cherche en permanence a eviter les calculs flottants pendant le jeu, a precalculer les constantes trigonometriques et a comparer des distances au carre plutot que d'extraire des racines carrees.

3.2 L'horloge du systeme

Le rythme d'execution du microcontroleur est donne par son horloge. Le projet utilise l'oscillateur interne haute frequence HFINTOSC, ce qui evite d'ajouter un quartz externe et simplifie la carte. La frequence de l'oscillateur, notee FOSC, est reglee a 32 MHz.

Sur l'architecture PIC16, une instruction ne s'execute pas en un seul battement d'horloge : le cycle d'instruction vaut FOSC divise par 4. On a donc :

```
F_cycle = FOSC / 4 = 32 MHz / 4 = 8 MHz
```

La duree d'un cycle d'instruction, notee Tcy, est l'inverse de cette frequence :

```
Tcy = 1 / 8 MHz = 125 ns
```

Autrement dit, le coeur execute la majorite de ses instructions en 125 nanosecondes. Cela peut paraitre rapide, mais il faut le comparer au cout d'un calcul flottant : une fonction comme `cos` ou `sqrtf`, emulee en logiciel sur 8 bits, demande un grand nombre de cycles. A 8 MHz, repeter de tels calculs a chaque tour de la boucle de jeu et pour chaque objet a l'ecran couterait beaucoup trop cher en temps et ferait chuter la fluidite. C'est la consequence directe et concrete de l'horloge choisie : la rapidite percue du jeu depend de notre capacite a

remplacer les flottants par des entiers et par des valeurs precalculees. On retrouve ainsi le meme fil conducteur que pour l'absence de FPU. A noter que cette horloge de 8 MHz (FOSC/4) sert egalement de source de cadencement au peripherique de son, comme on le verra plus bas.

3.3 Les peripheriques utilises et configures via MCC

Le PIC16F18877 embarque de nombreux peripheriques materiels. Quatre d'entre eux, plus les ports d'entree/sortie, sont mis a profit dans ce projet ; tous sont parametres a l'aide de MCC.

- **L'oscillateur** : configure pour fournir les 32 MHz du HFINTOSC interne decrits ci-dessus. C'est la base de temps de tout le systeme, dont decoulent a la fois le cycle d'instruction et l'horloge des peripheriques.
- **Le module SPI1 (MSSP1)** : il pilote l'ecran TFT ILI9488. Il est configure en mode Maitre (c'est le PIC qui impose l'horloge SCK et cadence les echanges) et en SPI Mode 0 (polarite d'horloge au repos a l'etat bas et echantillonnage sur le premier front, conformes a ce qu'attend l'ecran). La communication est a sens unique, du PIC vers l'ecran : seule la ligne MOSI transporte des donnees. La ligne MISO de l'ecran existe physiquement mais n'est pas exploitee par le programme, car le jeu n'a jamais besoin de relire l'etat de l'afficheur. Le SPI etant un bus serie cadence materiellement, le module se charge de serialiser chaque octet bit a bit a la frequence SCK : cela permet d'envoyer rapidement les nombreux octets de couleur necessaires au remplissage des formes, en confiant au peripherique un travail qui serait beaucoup trop lent a realiser bit par bit en logiciel.
- **Le convertisseur ADCC** : il numerise les deux tensions issues du joystick analogique. La conversion se fait sur 10 bits, ce qui donne des valeurs entieres comprises entre 0 et 1023 pour chaque axe. L'axe X est lu sur la broche RC7 et l'axe Y sur la broche RC6. Le convertisseur etant unique, le programme selectionne tour a tour le canal voulu avant chaque conversion : on lit ainsi successivement les deux axes a partir d'un seul bloc analogique-numerique.
- **Le NCO1 (Numerically Controlled Oscillator)** : ce peripherique genere le son sur la broche RA6. Sa source d'horloge est FOSC/4, soit 8 MHz. Le NCO fonctionne par accumulation sur 20 bits : a chaque cycle il ajoute une valeur d'increment NCO1INC a un accumulateur de 20 bits, et c'est le debordement periodique de cet accumulateur qui

fait basculer la sortie, produisant une frequence proportionnelle a l'increment. La relation est :

```
f = (8 000 000 x NCO1INC) / 2^20 = NCO1INC x 7,6294 Hz
```

Pour obtenir un ton donne sans calcul flottant, le code ecrit `NCO1INC = frequence >> 3`, c'est-a-dire une division par 8 realisee par un simple decalage de bits (operation tres rapide pour le coeur 8 bits, qui evite une multiplication ou une division logicielle). Le ton effectivement joue vaut alors environ `frequence x (8e6 / (8 x 2^20))`, soit `frequence x 0,9537` : on reste tres proche de la frequence demandee. Le decalage `>>3` (diviser par 8, c'est-a-dire multiplier par 0,125) est en realite une approximation rapide du facteur exact 0,1311, jugee parfaitement suffisante pour un effet sonore de jeu ou la justesse absolue n'a pas d'importance. On reconnait encore le fil rouge : remplacer une division flottante exacte par un decalage entier approche, mais quasi gratuit. - **Les ports d'entree/sortie** : ils gerent les boutons poussoirs et les lignes de controle de l'ecran. Les boutons sont configures en entree avec resistance de tirage interne (pull-up), si bien qu'au repos la broche lit 1 et qu'a l'appui elle lit 0 : c'est une logique active a l'etat bas. Les lignes de controle de l'ecran (CS pour la selection, RST pour le reset, DC pour distinguer commande et donnee) sont configurees en sortie numerique. Inversement, les broches du joystick (RC6 et RC7) sont configurees en entrees analogiques afin d'etre routees vers le convertisseur.

3.4 Le role de MPLAB Code Configurator (MCC)

La chaine de developpement repose sur l'environnement MPLAB X IDE, le compilateur XC8 et l'outil MPLAB Code Configurator (MCC). MCC est un assistant graphique : plutot que d'ecrire a la main les valeurs de tous les registres de configuration des peripheriques (oscillateur, SPI, ADCC, NCO, broches), on les regle dans une interface visuelle, et MCC genere automatiquement le code d'initialisation correspondant dans le dossier `mcc_generated_files`.

Concretement, MCC produit la fonction `SYSTEM_Initialize`, qui est appelee au demarrage du programme et qui met en place l'horloge et tous les peripheriques d'un seul coup. Il genere egalement des fonctions specifiques comme `ADCC_Initialize` (et les fonctions de conversion associees) pour le convertisseur, ainsi que `SPI1_Open` pour ouvrir le canal SPI vers l'ecran. MCC fabrique enfin des macros lisibles pour manipuler les broches : par exemple

`BTN_B_GetValue` pour lire l'etat du bouton de tir, ou `TFT_CS_Pin_SetLow` pour abaisser la ligne de selection de l'ecran. Ces noms parlants rendent le code du jeu plus clair et evitent de manipuler directement les registres binaires des ports, sources frequentes d'erreurs.

L'interet est double : on gagne du temps et on reduit les risques d'erreur de configuration, tout en gardant un code genere coherent et documente. Le programmeur peut alors se concentrer sur la logique du jeu plutot que sur les details bas niveau du silicium. C'est d'autant plus appreciable que le couplage avec les bibliotheques de l'ecran (le pilote `ili9488` et la bibliotheque graphique `GFX_Library`) se fait au travers de ces memes macros et fonctions generees.

3.5 Tableau de brochage complet

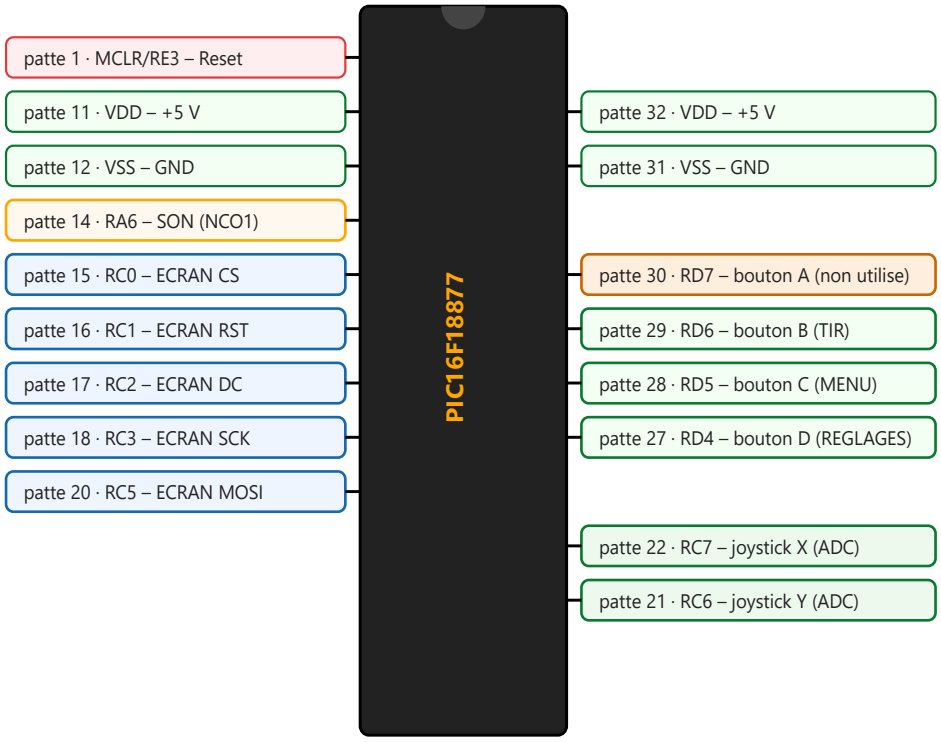
Le tableau ci-dessous reprend l'affectation des broches utilisees par le projet, du point de vue du boitier 40 broches PDIP : pour chaque signal sont indiques le port du PIC, le numero de patte et la fonction.

Port	Patte (PDIP 40)	Fonction
RA6	14	Sortie SON (NCO1) vers la carte son / buzzer
RC0	15	Ecran : CS (selection)
RC1	16	Ecran : RST (reset)
RC2	17	Ecran : DC (Data/Command)
RC3	18	Ecran : SCK (horloge SPI)
RC5	20	Ecran : SDI/MOSI (donnees SPI)
RC6	21	Joystick axe Y (entree ADC)
RC7	22	Joystick axe X (entree ADC)

Port	Patte (PDIP 40)	Fonction
RD4	27	Bouton D (REGLAGES)
RD5	28	Bouton C (JOUER / RETOUR / MENU)
RD6	29	Bouton B (TIR)
RD7	30	Bouton A (non utilise)

La figure suivante represente le brochage complet du composant et permet de visualiser la repartition des signaux autour du boitier.

Brochage utilise du PIC16F18877 (boitier 40 broches PDIP)



En bleu : liaison ecran (SPI). En orange : son. En vert : manette et alimentation. Les broches non listees ne sont pas utilisees.

Figure 6 — Brochage du PIC16F18877 et affectation des signaux du projet.

On remarque que les signaux sont regroupes de maniere logique : la sortie son est isolee sur le port A (RA6), tout ce qui concerne l'ecran et le joystick se concentre sur le port C (RC0 a RC7), et les quatre boutons occupent le haut du port D (RD4 a RD7). Ce regroupement par fonction facilite a la fois le cablage et la relecture du programme.

Pour terminer, il faut citer les broches indispensables au fonctionnement du composant mais qui ne portent aucune fonction applicative. L'alimentation positive VDD (+5 V) est appliquee sur les pattes 11 et 32, et la masse VSS (GND) sur les pattes 12 et 31 : ce doublement des broches d'alimentation est habituel pour bien decoupler le circuit et stabiliser la tension. Enfin, la broche MCLR (patte 1) est l'entree de reset (Master Clear) : maintenue a l'etat haut en fonctionnement normal, elle force le redemarrage du microcontroleur lorsqu'elle est tiree a la masse. Ces trois signaux constituent le minimum vital qui doit etre cable avant meme de raccorder le moindre peripherique.

4. Conception mecanique et modelisation 3D (SolidWorks)

La realisation du jeu de type *Asteroids* ne se limite pas a l'electronique et au logiciel embarque : pour obtenir un objet manipulable et fiable, il faut un support physique qui maintienne l'ensemble des organes du systeme. Le binome a donc mene en parallele une etude de conception mecanique sous SolidWorks, afin de modeliser en trois dimensions les pieces de structure necessaires. Cette partie decrit la demarche suivie et le role des pieces concues.

> Remarque de transparence : les trois fichiers SolidWorks d'origine (`ecran.SLDPRT`, `Piece2.SLDPRT`, `Piece3.SLDPRT`) sont enregistres dans un format proprietaire chiffre qui n'a pas pu etre relu automatiquement. Les dimensions exactes des pieces n'ont donc pas pu etre extraites du modele. Le present chapitre decrit par consequent la *demarche* de conception et le *role* de chaque piece, sans avancer de cotes precises. Chaque fois qu'une valeur dimensionnelle serait attendue, la mention **(cotes a confirmer sur le modele)** est portee explicitement.

4.1 Etude preliminaire et besoins mecaniques

Avant toute modelisation, il convient d'identifier les contraintes que la structure doit satisfaire. Ces besoins decoulent directement des composants utilises dans le projet et des conditions d'utilisation du jeu.

Le premier besoin est le **positionnement et le maintien de l'ecran TFT ILI9488**. Cet ecran de 480x320 pixels en mode paysage constitue la surface de jeu visible par le joueur : il doit etre tenu fermement, bien a plat et a la verticale, pour rester lisible et eviter tout porte-a-faux qui solliciterait les soudures de la nappe de connexion. L'ecran ne dispose que de huit bornes utilisees au dos (VCC, GND, SCLK, MOSI, MISO, CS, RES, DC, soit les bornes 1 a 8 du module), les cinq autres (BKL, SCL, SDA, INT, SDCS) restant inutilisees : le support doit donc preserver l'acces a ces bornes et menager le passage des fils vers la carte du microcontroleur.

Figure 7 — Face arriere de l'ecran TFT ILI9488 montrant les bornes a maintenir accessibles, contrainte directe pour la piece de support.

Le deuxieme besoin est la **protection des circuits**. Le systeme comporte plusieurs cartes a composants fragiles : la carte de la manette portant le PIC16F18877 (boitier 40 broches PDIP) et la carte son realisee sous KiCad (PCB double face de 45 mm x 45 mm, composants traversants). Ces circuits, leurs soudures et leurs connecteurs doivent etre proteges des chocs et des contacts accidentels.

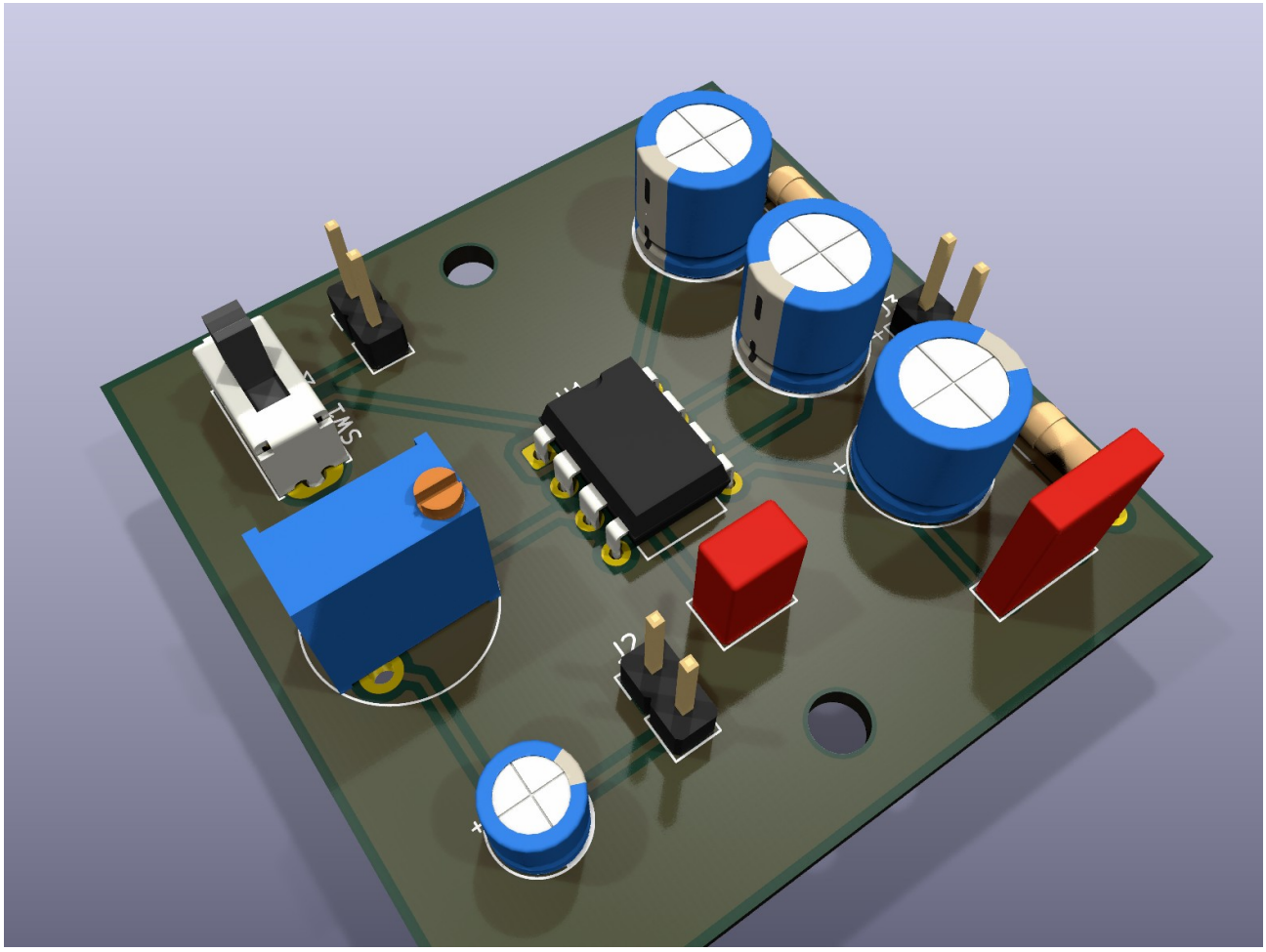


Figure 8 — Rendu 3D isometrique de la carte son (45 mm x 45 mm) dont l'encombrement doit etre pris en compte dans le modele mecanique.

Le troisieme besoin concerne **l'organisation du passage des cables**. Entre l'ecran, la carte de commande et la carte son circulent de nombreuses liaisons : le bus SPI vers l'ecran (SCK sur RC3, MOSI sur RC5, CS sur RC0, DC sur RC2, RST sur RC1), les deux axes analogiques du joystick (axe X sur RC7, axe Y sur RC6), les lignes des boutons poussoirs et le signal audio issu de la broche RA6 (sortie du NCO1) vers l'entree Audio_In de la carte son. Un cheminement ordonne evite les tractions sur les connecteurs et les courts-circuits.

Le quatrieme besoin est **l'accessibilite des organes de commande** : le joystick analogique a deux axes et les boutons poussoirs (BTN_B = TIR sur RD6, BTN_C = JOUER/RETOUR/MENU sur RD5, BTN_D = REGLAGES sur RD4) doivent rester atteignables et confortables a actionner, sans gener la lecture de l'ecran. Le support mecanique conditionne donc directement l'ergonomie du jeu.

4.2 Presentation de SolidWorks

SolidWorks est un logiciel de **conception assistee par ordinateur (CAO) en trois dimensions, de type parametrique**. Le principe parametrique signifie que la geometrie est pilotee par des cotes et des relations : on dessine d'abord une **esquisse** (un profil en deux dimensions) sur un plan, puis on la **cote** (longueurs, diametres, angles) et on lui impose des contraintes geometriques (parallelisme, tangence, coincidence, symetrie). Modifier une cote met automatiquement a jour toute la piece, ce qui rend la conception souple et reversible.

A partir de ces esquisses cotees, on cree le volume par des **fonctions** : principalement l'**extrusion** (on donne une epaisseur a un profil pour obtenir un solide), mais aussi l'enlevement de matiere (percages, poches), les conges et les chanfreins. Chaque fonction s'inscrit dans un **arbre de creation** qui conserve l'historique parametre de la piece : on peut revenir sur n'importe quelle etape sans tout recommencer. Plusieurs pieces independantes sont ensuite regroupees dans un **assemblage**, ou elles sont positionnees les unes par rapport aux autres au moyen de **contraintes** (coincidence de faces, alignement d'axes, distances imposees). L'assemblage permet de verifier que les pieces s'emboitent correctement et de detecter d'eventuelles interferences. Ce flux de travail (esquisse cotee, fonction volumique, assemblage contraint) structure l'ensemble de la conception mecanique du projet, et offre un parallele interessant avec la demarche logicielle : dans les deux cas, on construit a partir d'elements parametres et reutilisables plutot que de valeurs figees.

4.3 Les pieces modelisees

Le modele du projet repose sur trois pieces distinctes. En l'absence de relecture des fichiers, leurs roles sont decrits de maniere fonctionnelle et plausible, coherents avec les besoins du paragraphe 4.1.

La piece `ecran` constitue le **support et cadre de maintien de l'ecran TFT ILI9488**. Sa fonction est d'accueillir l'ecran, d'en bloquer le pourtour pour l'empecher de glisser et de presenter la dalle au joueur dans le bon sens (mode paysage, conforme au `setRotation(3)` utilise dans le pilote logiciel). Le cadre menage en principe une ouverture correspondant a la zone active de l'affichage et laisse libre l'acces aux bornes situees au dos **(cotes a confirmer sur le modele)**.

Les pieces Piece2 et Piece3 sont des **elements de structure** completant le cadre d'ecran. Des roles plausibles, coherents avec un petit boitier de jeu, sont les suivants : support de carte (plateforme ou pattes destinees a fixer la carte du microcontroleur ou la carte son), facade ou paroi (fermeture du boitier, percee pour laisser depasser le joystick et les boutons), et entretoise (piece intercalaire fixant l'ecartement entre deux niveaux et garantissant la rigidite de l'ensemble). La repartition exacte de ces fonctions entre Piece2 et Piece3 reste **(a confirmer sur le modele)**.

Figure 9 — Carte de la manette (vue de dessus) que les pieces de structure doivent positionner et fixer.

Le tableau suivant recapitule les pieces et le type de fonction SolidWorks mobilise pour chacune.

Piece	Role mecanique (plausible)	Type de fonction SolidWorks principal
ecran	Support / cadre de maintien de l'ecran ILI9488	Esquisse cotee + extrusion + poche (ouverture d'affichage)
Piece2	Element de structure : support de carte ou facade	Esquisse cotee + extrusion + percages de fixation
Piece3	Element de structure : facade ou entretoise	Esquisse cotee + extrusion + percages de fixation

4.4 Methode de conception

La modelisation suit une demarche descendante, partant des dimensions connues des composants pour aboutir a un ensemble assemble.

La premiere etape est **l'esquisse cotee a partir des dimensions des composants connus**. On part des encombrements physiques mesurables : l'ecran TFT (gabarit de la dalle et position des bornes), le PCB de la carte son dont le format est connu (45 mm x 45 mm), et le microcontroleur PIC16F18877 dans son boitier 40 broches PDIP. Ces references donnent les cotes de depart des esquisses : largeur et hauteur de l'ouverture du cadre, entraxes des trous de fixation, hauteur des entretoises. Les valeurs precises de ces esquisses ne pouvant etre relues, elles sont **(a confirmer sur le modele)**.

La deuxieme etape est **l'extrusion** : chaque profil esquisse recoit une epaisseur pour devenir un solide. L'epaisseur est choisie comme un compromis entre la rigidite recherchee et l'encombrement, et reste **(a confirmer sur le modele)**.

La troisieme etape consiste a **ajouter les percages de fixation** : trous traversants pour les vis qui solidarisent les cartes et les pieces entre elles, et eventuellement degagements pour les connecteurs et le passage des cables identifies au paragraphe 4.1. Le caractere parametrique de SolidWorks prend ici tout son sens : un entraxe de trous defini comme une cote pilotee

peut etre ajuste en une saisie si la mesure reelle sur la carte differe legerement de la valeur estimee.

La quatrieme etape est **l'assemblage et la verification**. Les pieces `ecran`, `Piece2` et `Piece3` sont reunies dans un assemblage et positionnees par des contraintes (coincidences de faces, alignements de percages). On procede alors a un **controle des collisions** (recherche d'interferences entre solides) et a une verification de **l'encombrement** general, pour s'assurer que l'ecran, les cartes et les organes de commande tiennent ensemble sans conflit. Cette etape exploite les contraintes vues au paragraphe 4.2 pour figer la position relative des pieces et simuler le montage reel.

4.5 Integration et apport du modele 3D

L'interet de cette modelisation depasse la simple representation : le modele 3D **sert a valider la faisabilite mecanique avant fabrication**. En verifiant numeriquement que les pieces s'assemblent, que les cartes trouvent leur place, que les percages sont alignes et que rien n'entre en collision, on detecte les erreurs de conception tant qu'elles ne coutent que quelques minutes de modification a l'ecran, plutot qu'une piece a refaire. C'est la meme philosophie de prudence que celle qui guide le reste du projet, ou l'on cherche a anticiper les difficultes plutot qu'a les subir : ici, l'assemblage virtuel joue le role de repetition generale avant le montage reel, exactement comme l'assemblage virtuel de la carte son sous KiCad precede son brasage.

Le modele constitue egalement une reference dimensionnelle commune au binome et un document de communication clair sur l'architecture mecanique de l'objet. En conclusion, et bien que les cotes exactes n'aient pu etre extraites des fichiers proprietaires, la demarche reste complete et methodique : analyse des besoins, choix d'un outil de CAO parametrique adapte, definition du role de chaque piece, conception par esquisses cotees et extrusions, puis assemblage et controle avant fabrication. Une relecture ulterieure des fichiers `ecran.SLDPRT`, `Piece2.SLDPRT` et `Piece3.SLDPRT` dans SolidWorks permettrait de completer ce chapitre par les cotes definitives **(a confirmer sur le modele)**.

5. Conception électronique : la carte son (schéma KiCad)

Le jeu produit des effets sonores (tir, destruction d'astéroïde, etc.) grâce au périphérique NCO1 (Numerically Controlled Oscillator) du PIC16F18877, qui génère un signal sur la broche RA6 (patte 14). Ce signal logique, disponible en sortie du microcontrôleur, n'est toutefois pas exploitable tel quel par un haut-parleur : son courant de sortie est trop faible. Une carte d'amplification dédiée, conçue sous KiCad 9 et réalisée sous la forme d'un circuit imprimé (PCB) double face de 45 mm x 45 mm à composants traversants (THT), a donc été développée pour mettre en forme et amplifier ce signal. Ce chapitre détaille la démarche de conception du schéma électronique, bloc par bloc, et justifie le rôle de chaque composant.

5.1 Analyse des besoins audio

Le point de départ de la conception est l'analyse précise du besoin. Le NCO1 est configuré pour produire un signal carré dont la fréquence est pilotée par le programme. Sa source d'horloge est FOSC/4 = 8 MHz et son accumulateur est codé sur 20 bits, de sorte que la fréquence de sortie vaut :

> f = (8 000 000 x NCO1INC) / 2^20 = NCO1INC x 7,6294 Hz

Pour une fréquence cible donnée, le code écrit `NCO1INC = frequence >> 3`, ce qui revient à une division par 8 par décalage de bits (0,125). La fréquence réellement jouée vaut alors environ `frequence x (8e6 / (8 x 2^20)) = frequence x 0,9537`, soit une valeur très proche de la fréquence demandée. Le décalage `>> 3` est en effet une approximation rapide, et peu coûteuse pour un microcontrôleur lent, du facteur exact 0,1311. Le signal obtenu est donc une onde rectangulaire dont les niveaux logiques oscillent entre 0 V et la tension d'alimentation, riche en harmoniques.

Trois besoins fonctionnels se dégagent de cette situation. Premièrement, il faut **amplifier** ce signal : la sortie d'un port du microcontrôleur ne peut fournir qu'un courant très limité, insuffisant pour faire vibrer la membrane d'un haut-parleur avec un volume audible. Il est donc nécessaire d'intercaler un étage d'amplification de puissance capable de débiter du courant dans une charge de faible impédance. Deuxièmement, l'utilisateur doit pouvoir **régler le volume**, afin d'adapter le niveau sonore au confort d'écoute. Troisièmement, il faut prévoir une

fonction **marche/arrêt**, permettant de couper l'alimentation de la carte (et donc le son) sans agir sur le reste du système.

À ces besoins s'ajoute une contrainte de simplicité : la carte devant être fabriquée et brasée à la main par le binôme, le circuit doit rester à composants traversants (THT) et reposer sur un nombre réduit de composants externes, pour limiter le risque d'erreur de câblage et faciliter les contrôles.

5.2 Choix de l'amplificateur LM386

Le composant central de la carte est l'amplificateur de puissance audio **LM386** (référence U1, boîtier DIP-8). Ce choix répond directement aux contraintes énoncées ci-dessus.

Le LM386 est un amplificateur audio basse tension, conçu spécifiquement pour les applications de petite puissance alimentées en faible tension, ce qui correspond bien à un montage embarqué fonctionnant sous une alimentation modeste. Son principal atout pour ce projet est sa **simplicité de mise en œuvre** : il ne requiert que très peu de composants externes pour fonctionner. Là où un amplificateur construit à partir de transistors discrets ou d'un amplificateur opérationnel générique imposerait un réseau de polarisation et de contre-réaction relativement lourd, le LM386 intègre déjà l'essentiel de ces fonctions dans son boîtier.

Par ailleurs, le LM386 possède un **gain fixé à 20 par défaut**. Ce gain est obtenu sans aucun composant supplémentaire : il suffit de laisser les broches 1 et 8 (broches de réglage du gain) non pontées. Ce gain de 20 est largement suffisant pour amener le signal du NCO1 à un niveau exploitable par un petit haut-parleur. Le boîtier DIP-8 traversant se brase aisément au fer à souder et autorise, le cas échéant, l'usage d'un support de circuit intégré, ce qui protège le composant pendant le brasage et facilite son remplacement. Le LM386 constitue ainsi un compromis idéal entre performance suffisante, faible encombrement et facilité de réalisation.

5.3 Le schéma bloc par bloc

L'architecture de la carte se décompose naturellement en quatre blocs fonctionnels, présentés ci-dessous. Le schéma de principe synthétise leur enchaînement, du signal d'entrée jusqu'au haut-parleur.

Figure 10 — Décomposition fonctionnelle de la carte son : alimentation, entrée/volume, amplification et sortie.

Bloc alimentation et découplage. Ce bloc regroupe le connecteur d'alimentation J1 (« Power supply », 2 points), l'interrupteur à glissière SW1 (SPST, marche/arrêt) et les condensateurs de découplage C2 (100 nF) et C3 (10 µF). L'interrupteur SW1 répond directement au besoin de marche/arrêt : placé en série avec l'alimentation, il coupe la tension d'alimentation de l'ensemble de la carte. Les condensateurs C2 et C3 stabilisent cette tension au plus près du circuit intégré, comme on le détaillera au paragraphe 5.4.

Bloc entrée et volume. Le signal sonore provenant de RA6 arrive par le connecteur J2 (« Audio_In », 2 points). Il traverse d'abord le condensateur de couplage C1 (1 µF), puis le potentiomètre RV1 (10 kΩ) qui assure le réglage du volume. En agissant sur le curseur du potentiomètre, l'utilisateur prélève une fraction plus ou moins grande du signal d'entrée avant son amplification : c'est ainsi que le volume est réglé, en faisant varier l'amplitude du signal présenté à l'amplificateur. La résistance R1 (1 kΩ) participe à la mise en forme du signal d'entrée et à la polarisation.

Bloc amplification. Le cœur du montage est l'amplificateur U1 (LM386). Le signal utile est appliqué sur l'entrée non inverseuse, **broche 3 (+)**, tandis que l'entrée inverseuse, **broche 2 (−)**, est reliée à la masse. L'alimentation positive V+ est appliquée sur la **broche 6**, et la masse (GND) sur la **broche 4**. La **broche 7 (BYPASS)** est découplée par le condensateur C4 (10 µF) : ce découplage interne contribue à la stabilité et à la propreté du signal amplifié. Comme indiqué précédemment, les **broches 1 et 8 ne sont pas pontées**, ce qui fixe le gain à sa valeur par défaut de 20.

Bloc sortie. La sortie du LM386 se prend sur la **broche 5**. On y trouve d'abord un réseau dit de Boucherot (ou réseau de Zobel), constitué de la résistance R2 (100 Ω) en série avec le condensateur C5 (47 nF), placé entre la sortie et la masse ; son rôle est d'assurer la stabilité de l'amplificateur. Le signal amplifié est ensuite transmis au haut-parleur à travers le condensateur de liaison C6 (220 µF), qui débouche sur le connecteur J3 (« Audio_out », 2 points). Les rendus 3D ci-dessous, générés par KiCad à partir du schéma et du routage, donnent un aperçu de l'implantation physique des composants sur le circuit imprimé double

face de 45 mm x 45 mm.

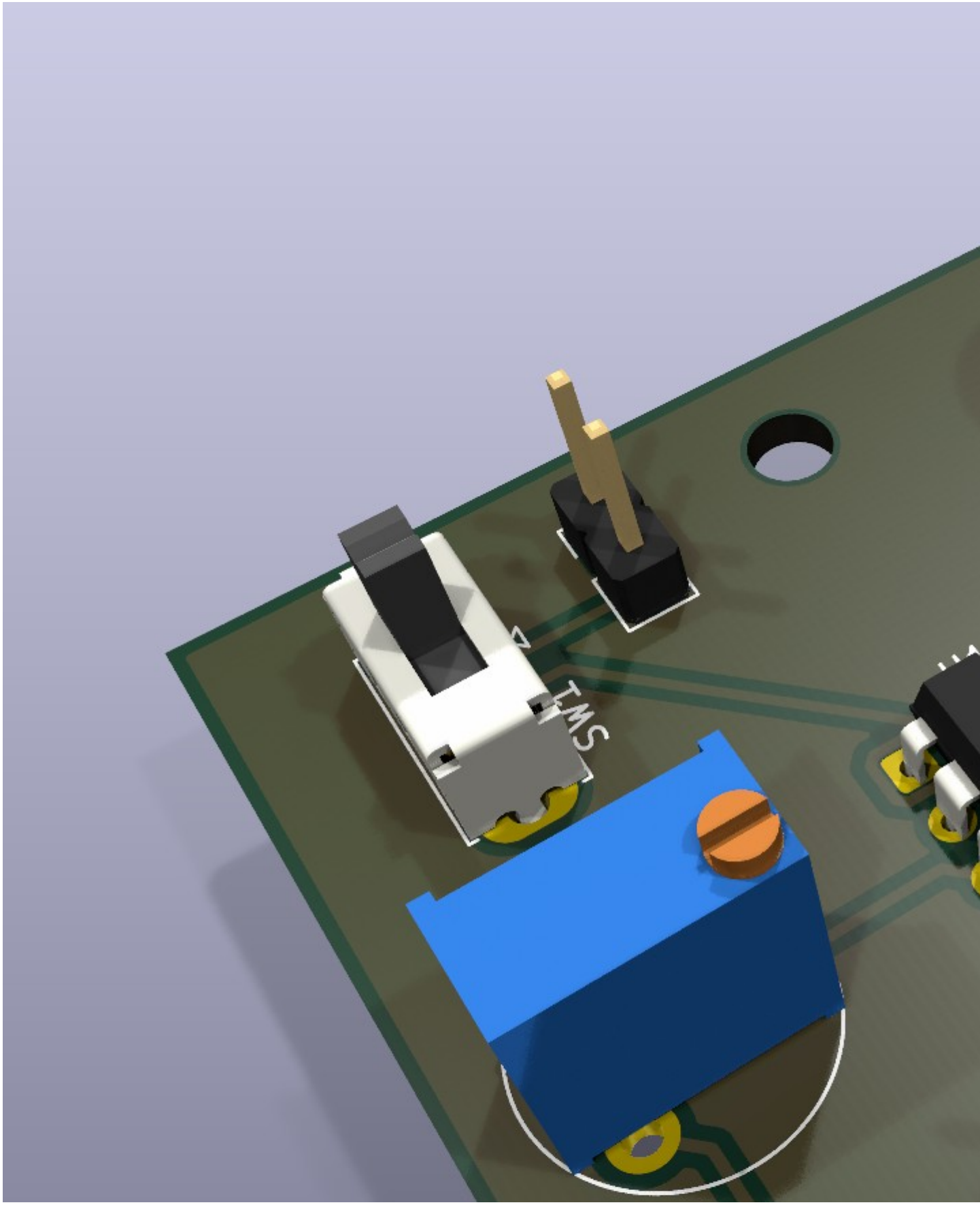

TMS

Figure 11 — Rendu 3D isométrique de la carte son sous KiCad.

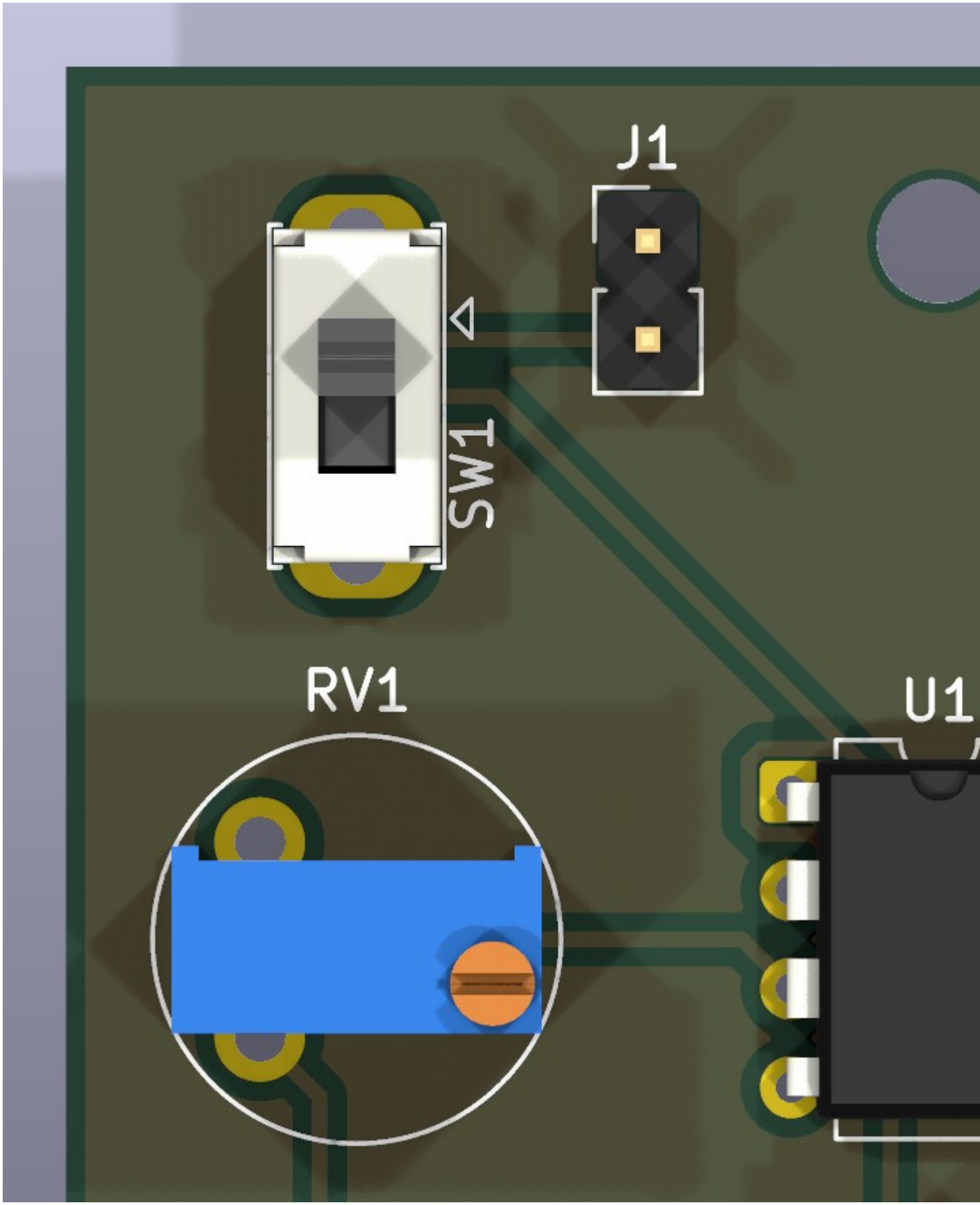

J1
SW1
RV1
U1

Figure 12 — Rendu 3D de la carte son vue de dessus, montrant l'implantation des composants.

5.4 Justification du rôle de chaque composant

Le tableau suivant récapitule la nomenclature (BOM) de la carte son, avec pour chaque composant sa référence, sa valeur et son rôle fonctionnel.

U1	LM386 (DIP-8)	Amplificateur de puissance audio, gain 20 par défaut
RV1	10 kΩ	Potentiomètre de réglage du volume (en entrée)
R1	1 kΩ	Résistance d'entrée / polarisation
R2	100 Ω	Réseau de Boucherot/Zobel (avec C5), stabilité
C1	1 µF	Condensateur de couplage / liaison d'entrée
C2	100 nF	Découplage de l'alimentation
C3	10 µF	Découplage / réservoir d'alimentation
C4	10 µF	Découplage de la broche BYPASS (broche 7)
C5	47 nF	Réseau de Zobel (avec R2), anti-oscillation
C6	220 µF	Condensateur de liaison de sortie vers le haut-parleur
SW1	SPST	Interrupteur marche/arrêt

J1	2 points	Connecteur d'alimentation (Power supply)
J2	2 points	Entrée du signal audio (Audio_In, depuis RA6)
J3	2 points	Sortie vers le haut-parleur (Audio_out)

Trois familles de composants méritent une explication détaillée, car elles illustrent des principes classiques de l'électronique analogique audio.

Les condensateurs de couplage (C1 et C6). Un condensateur de couplage, aussi appelé condensateur de liaison, a pour fonction de **bloquer la composante continue** du signal tout en laissant passer la composante variable (alternative) qui porte l'information sonore. En effet, un condensateur se comporte comme un circuit ouvert vis-à-vis du courant continu et comme un passage de plus en plus aisé à mesure que la fréquence augmente. À l'entrée, C1 (1 µF) empêche la tension continue présente sur le signal du NCO1 de perturber le point de fonctionnement de l'amplificateur. À la sortie, C6 (220 µF) isole le haut-parleur de la tension continue présente sur la broche 5 : seul le signal alternatif utile est transmis à la bobine, ce qui évite qu'un courant continu permanent ne traverse le haut-parleur. La valeur élevée de C6 s'explique par la faible impédance d'un haut-parleur : pour laisser passer les fréquences basses sans atténuation, le condensateur de liaison doit présenter une capacité importante.

Les condensateurs de découplage (C2, C3 et C4). Le découplage consiste à placer des condensateurs entre l'alimentation et la masse, au plus près du circuit intégré, afin de **filtrer l'alimentation**. Lorsque l'amplificateur débite des pointes de courant pour reproduire le son, il sollicite brutalement la ligne d'alimentation ; sans découplage, ces appels de courant provoqueraient des creux et des oscillations de tension susceptibles de dégrader le signal, voire de rendre l'amplificateur instable. Le condensateur C2 (100 nF) filtre les variations rapides (hautes fréquences), tandis que C3 (10 µF) joue le rôle de petit réservoir d'énergie pour les variations plus lentes. Le condensateur C4 (10 µF) découple spécifiquement la broche 7 (BYPASS) du LM386, ce qui réduit le bruit et améliore la stabilité interne de l'amplificateur.

Le réseau de Boucherot / Zobel (R2 et C5). Ce réseau série, composé de R2 (100 Ω) et C5 (47 nF), est placé entre la sortie de l'amplificateur (broche 5) et la masse. Sa fonction est d'**éviter les oscillations** parasites à haute fréquence. La charge présentée par un haut-parleur est en réalité inductive, ce qui peut entraîner un comportement instable de l'étage de sortie. Le réseau de Zobel présente, en haute fréquence, une impédance résistive maîtrisée qui « voit » une charge stable du point de vue de l'amplificateur : il compense ainsi le caractère inductif du haut-parleur et empêche l'apparition d'oscillations indésirables.

5.5 Du schéma au circuit fabriqué

Une fois le schéma validé, KiCad permet de réaliser le routage du circuit imprimé puis de générer les fichiers de fabrication, appelés Gerbers : couches de cuivre (F_Cu pour la face avant, B_Cu pour la face arrière), masques de soudure, sérigraphie et fichiers de perçage (PTH pour les trous métallisés, NPTH pour les trous non métallisés). Le PCB nu, de 45 mm x 45 mm, a ensuite été reçu puis garni à la main : les composants traversants ont été insérés et brasés au fer à souder. La phase de vérification a comporté un contrôle visuel des soudures, une recherche systématique de courts-circuits entre pistes voisines et une mesure de continuité afin de s'assurer que chaque liaison prévue au schéma était bien établie. Les photographies ci-dessous montrent la carte son une fois brasée et câblée à l'ensemble du système.

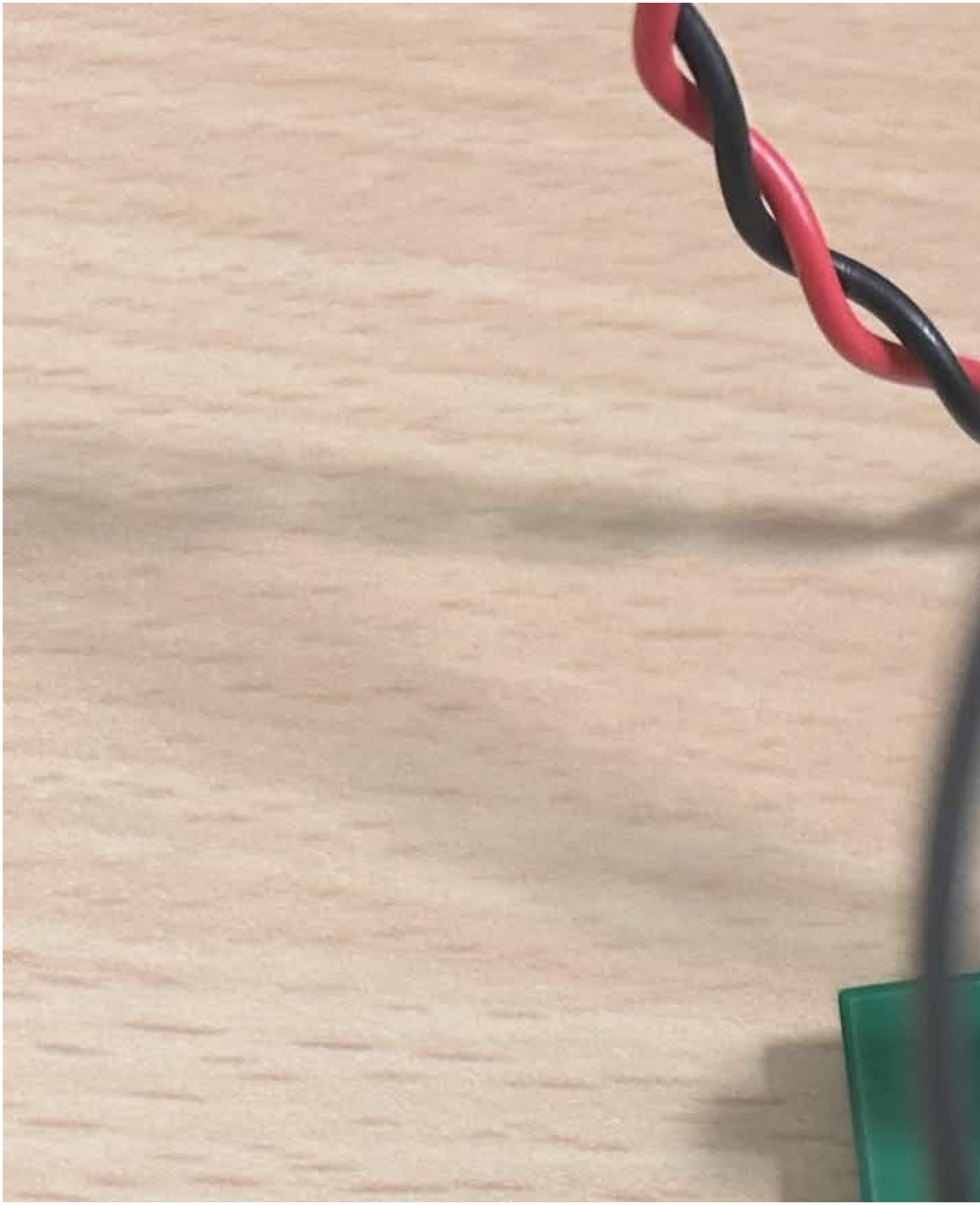


Figure 13 — Carte son brasée et câblée à l'ensemble du système.

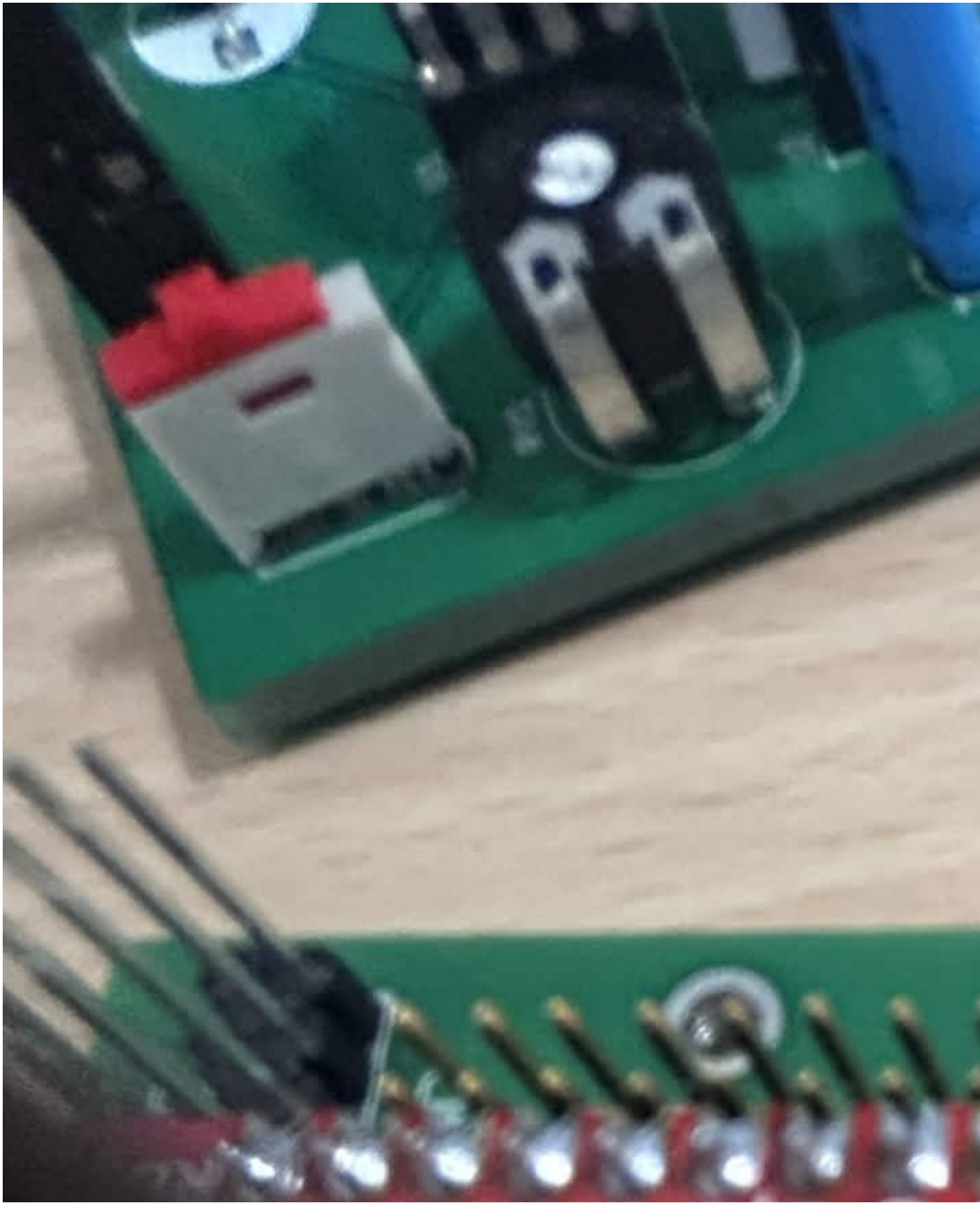


Figure 14 — Détail du câblage de la carte son.

En synthèse, le schéma de la carte son repose sur un découpage clair en quatre blocs et sur un choix de composant central, le LM386, particulièrement adapté à une réalisation embarquée simple. Chaque composant périphérique remplit une fonction précise — couplage, découplage, stabilité, réglage du volume ou commutation — qui contribue à transformer le signal carré brut du NCO1 en un son propre et de niveau réglable, restitué par le haut-parleur.

6. Conception du PCB et fabrication de la carte son

La carte son constitue l'organe matériel chargé de transformer le signal numérique émis par le microcontrôleur en un son audible. Le PIC16F18877 génère, par l'intermédiaire de son périphérique NCO1 (Numerically Controlled Oscillator) sur la patte RA6 (patte 14), un signal carré dont la fréquence correspond à la note jouée. Ce signal est de faible puissance : il ne peut pas attaquer directement un haut-parleur. La carte son a donc pour rôle de l'amplifier au moyen d'un amplificateur audio basse tension de référence LM386, tout en ménageant un réglage de volume en entrée et un filtrage destiné à garantir la stabilité du montage. Ce chapitre décrit l'ensemble du flux de travail, depuis la saisie du schéma jusqu'à la carte brasée et validée, en suivant l'enchaînement classique d'un projet de conception électronique sous KiCad 9.

6.1 Conception du PCB sous KiCad

La première étape consiste à saisir le schéma électrique dans l'éditeur de schématique de KiCad, en plaçant les symboles des composants et en traçant les liaisons (fils et bus). Chaque symbole reçoit une référence normalisée : U1 pour l'amplificateur LM386, RV1 pour le potentiomètre de volume, R1 et R2 pour les résistances, C1 à C6 pour les condensateurs, SW1 pour l'interrupteur à glissière, et J1, J2, J3 pour les trois connecteurs deux points (alimentation Power supply, entrée Audio_In et sortie Audio_out vers le haut-parleur). Une fois le schéma cohérent, on procède à l'association schéma vers empreintes : à chaque symbole logique on attribue une empreinte physique (footprint) correspondant au boîtier réel. L'ensemble du circuit a été conçu en composants traversants (THT, Through-Hole Technology), c'est-à-dire des composants dont les pattes traversent le PCB et sont brasées sur la face opposée. Ce choix, plus volumineux que le montage en surface (CMS), facilite grandement le brasage manuel au fer, ce qui est appréciable dans un cadre pédagogique.

La carte a été dimensionnée en double face (deux couches de cuivre, F_Cu en dessus et B_Cu en dessous) au format compact de 45 mm par 45 mm. Le format carré et réduit impose une discipline de placement et de routage, mais reste largement suffisant pour le nombre modeste de composants du montage.

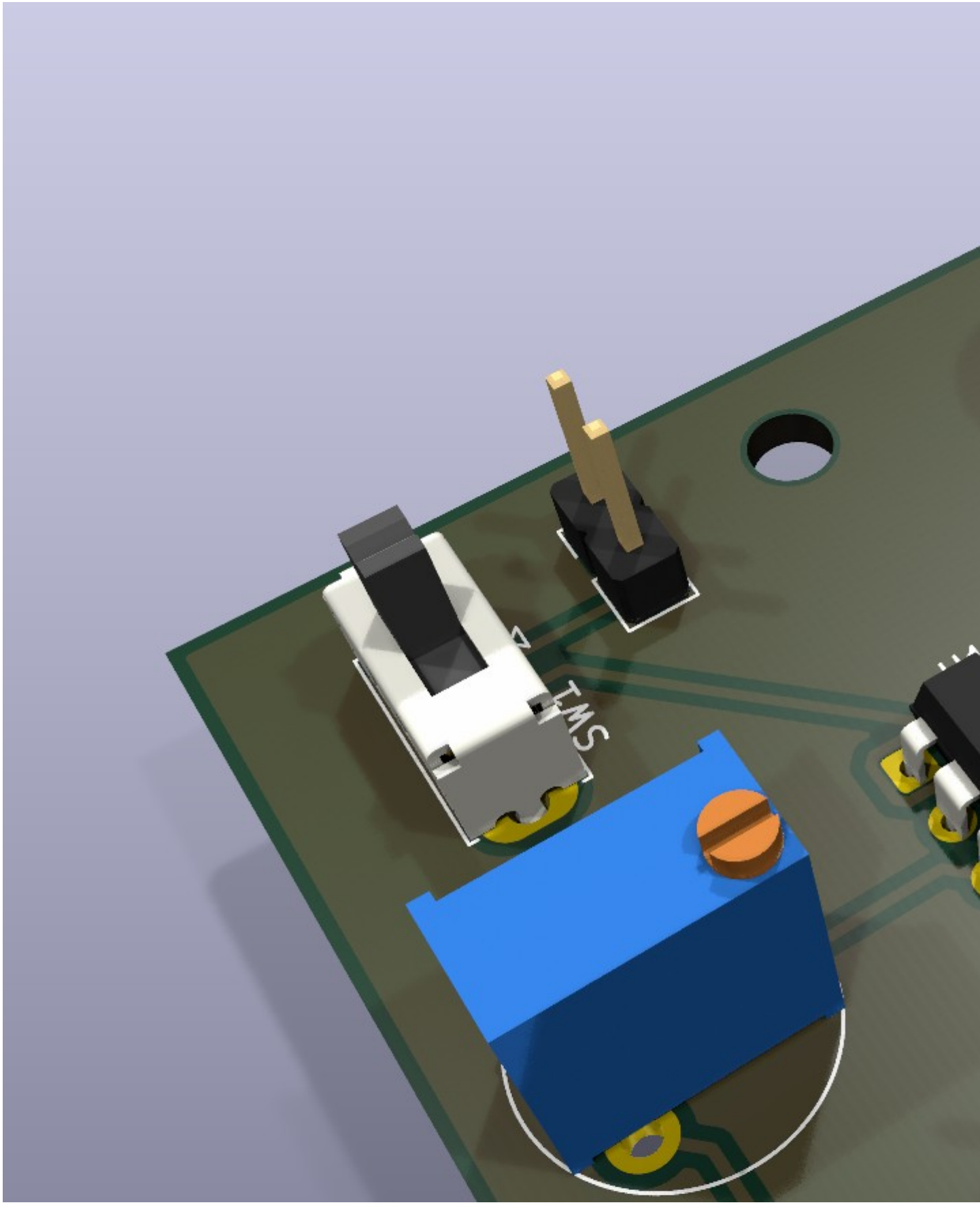

TMS

Figure 15 — rendu tridimensionnel isométrique de la carte son généré par KiCad à partir du fichier de PCB.

6.2 Placement des composants

Le placement constitue une étape déterminante pour la qualité finale du circuit. La règle directrice retenue a été de regrouper les composants par bloc fonctionnel, afin de garder ensemble les éléments qui travaillent de concert et de raccourcir les liaisons. On distingue quatre blocs.

Le premier bloc est l'alimentation et son découplage : il rassemble le connecteur d'alimentation J1, l'interrupteur à glissière SW1 (marche/arrêt, de type SPST), le condensateur de découplage C2 de 100 nF qui filtre les parasites haute fréquence sur l'alimentation, et le condensateur réservoir C3 de 10 µF qui lisse les variations lentes de tension.

Le deuxième bloc est l'entrée et le réglage de volume : le connecteur Audio_In J2 reçoit le signal venant de la patte RA6 du PIC ; le condensateur de couplage d'entrée C1 de 1 µF bloque toute composante continue tout en laissant passer le signal alternatif ; le potentiomètre RV1 de 10 k dose la fraction du signal transmise à l'amplificateur, formant ainsi le réglage de volume. La résistance R1 de 1 k assure une fonction d'entrée et de polarisation.

Le troisième bloc est l'amplification proprement dite, autour du LM386 (U1). Sur ce boîtier DIP-8, l'entrée non inverseuse (broche 3) reçoit le signal, l'entrée inverseuse (broche 2) est mise à la masse, l'alimentation positive V+ arrive sur la broche 6, la masse GND est sur la broche 4 et la broche BYPASS (broche 7) est découplée par le condensateur C4 de 10 µF. Le gain est laissé à sa valeur fixe de 20, car les broches 1 et 8 (qui permettraient de l'augmenter) ne sont pas pontées.

Le quatrième bloc est la sortie : le réseau de Zobel (ou réseau de Boucherot) formé par R2 (100 Ohm) et C5 (47 nF), placé de la sortie (broche 5) vers la masse, stabilise l'amplificateur en présence d'une charge inductive et prévient les oscillations ; le condensateur de liaison de sortie C6 de 220 µF bloque la composante continue avant le haut-parleur tout en laissant passer l'audio vers le connecteur Audio_out J3.

Une précaution majeure a guidé ce placement : éloigner l'entrée de la sortie. L'entrée

manipule un signal faible et donc sensible, tandis que la sortie véhicule un signal amplifié vingt fois plus grand. Si l'entrée et la sortie sont voisines, le signal de sortie risque de se recoupler sur l'entrée et de provoquer des oscillations parasites (effet Larsen interne). Le LM386 a donc été placé au centre de la carte, faisant office de pivot entre la chaîne d'entrée d'un côté et la chaîne de sortie de l'autre.

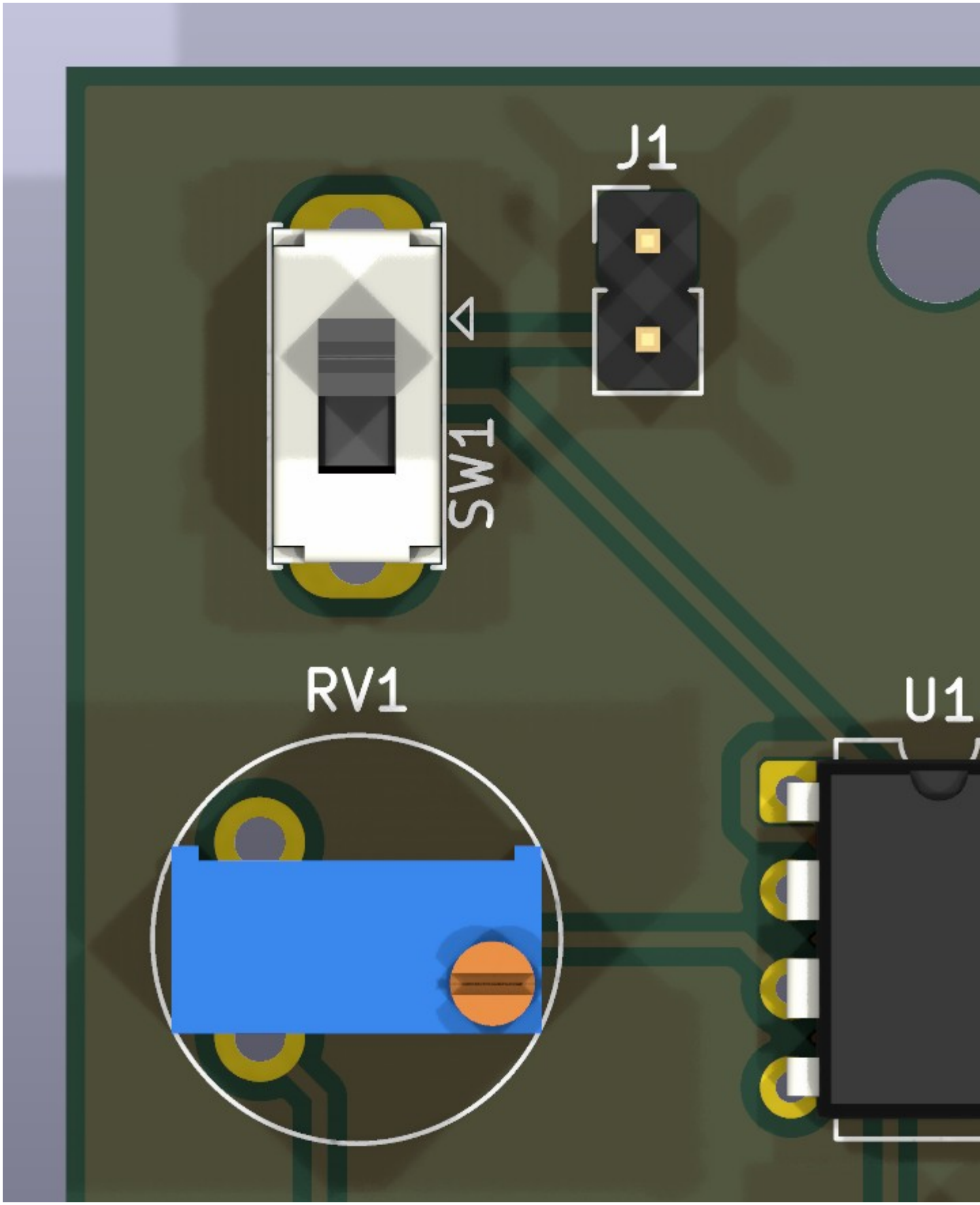

J1
SW1
RV1
U1

Figure 16 — rendu tridimensionnel de la carte son généré par KiCad, vue de dessus, montrant l'implantation des composants par bloc fonctionnel.

6.3 Routage

Le routage consiste à tracer les pistes de cuivre qui relient physiquement les pastilles selon les liaisons du schéma. Plusieurs principes ont été appliqués pour soigner le comportement audio du montage. D'abord, les pistes de puissance et de masse, qui transportent des courants plus importants, ont été tracées plus larges que les pistes de signal, afin de limiter leur résistance et la chute de tension associée. Ensuite, un plan de masse a été ménagé : une large surface de cuivre reliée à la masse offre un retour de courant à faible impédance et constitue un blindage naturel contre les perturbations.

Enfin, et c'est essentiel pour la qualité du son, on s'est attaché à éloigner les pistes de signal des pistes de puissance et de sortie. Une piste de sortie qui longerait de trop près la piste d'entrée induirait par couplage capacitif un bruit audible et, là encore, un risque d'oscillation. Le respect de ces règles de bon voisinage contribue à un signal de sortie propre, sans ronflement ni sifflement.

6.4 Vérifications électriques

Avant toute fabrication, deux contrôles automatiques offerts par KiCad ont été lancés. L'ERC (Electrical Rules Check, contrôle des règles électriques) s'exécute sur le schéma : il détecte les incohérences logiques, telles qu'une broche non connectée, deux sorties reliées entre elles ou une alimentation laissée flottante. Le DRC (Design Rules Check, contrôle des règles de dessin) s'exécute, lui, sur le circuit imprimé : il vérifie que les distances d'isolation entre pistes sont respectées, que les largeurs de pistes sont conformes aux règles fixées et qu'aucune piste ne chevauche une autre ou ne déborde du contour de la carte. Le passage sans erreur de l'ERC puis du DRC constitue la garantie que le fichier est sain et prêt à être envoyé en fabrication.

6.5 Génération des fichiers de fabrication et visualisation 3D

Une fois les vérifications validées, KiCad permet de produire les fichiers Gerber, format standard accepté par tous les fabricants de circuits imprimés. Chaque fichier décrit une

couche : le cuivre supérieur (F_Cu) et le cuivre inférieur (B_Cu), les masques de soudure (qui protègent le cuivre et n'ouvrent que les pastilles à braser), la sérigraphie (le marquage blanc des références et des contours), ainsi que les fichiers de perçage distinguant les trous métallisés PTH (Plated Through Hole, qui assurent une liaison électrique entre les deux faces) des trous non métallisés NPTH (trous mécaniques, comme les fixations). La visualisation 3D intégrée à KiCad, montrée aux figures précédentes, a permis de contrôler visuellement l'aspect final de la carte, la cohérence des empreintes et l'absence d'erreur grossière de placement avant d'engager toute dépense de fabrication.

6.6 Réception et brasage

À la réception du circuit imprimé nu, le travail d'assemblage a commencé par un inventaire des composants : on vérifie que chaque élément de la nomenclature (BOM) est présent et conforme, en contrôlant notamment les valeurs des résistances et des condensateurs ainsi que la référence et l'orientation du LM386. Le brasage des composants traversants a ensuite été réalisé au fer à souder, en commençant par les composants les plus bas (résistances) puis en progressant vers les plus hauts (condensateurs électrolytiques, connecteurs, potentiomètre). Une attention particulière a été portée à la polarité des composants polarisés, en particulier les condensateurs électrolytiques C3, C4 et C6, ainsi qu'à l'orientation du repère du LM386.

Chaque étape de brasage a été suivie d'un contrôle visuel des soudures : une bonne soudure présente un aspect lisse et brillant, en forme de cône régulier reliant la pastille à la patte. On a ensuite procédé à une recherche systématique de courts-circuits, suivie d'une mesure de continuité au multimètre pour vérifier que les liaisons attendues étaient bien établies et qu'aucune liaison parasite ne s'était créée entre deux pastilles voisines.

Figure 17 — face supérieure de la carte son après brasage des composants traversants.

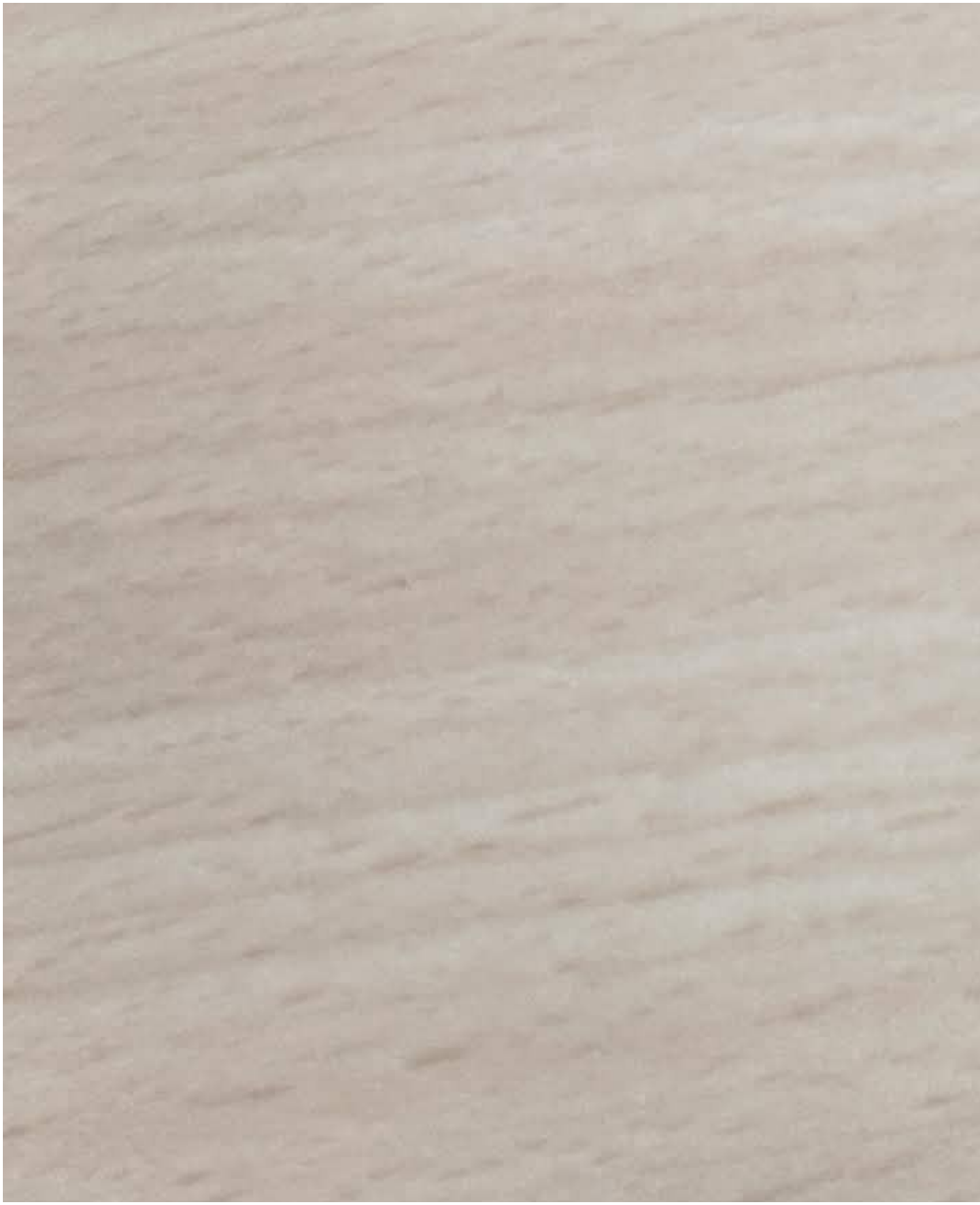


Figure 18 — face inférieure de la carte, permettant le contrôle visuel des soudures.

6.7 Difficultés du brasage et validation fonctionnelle

Le format compact de 45 mm par 45 mm impose des composants relativement proches les uns des autres. Cette densité a constitué la principale difficulté du brasage : lorsque deux pastilles sont voisines, un excès d'étain peut créer un pont de brasure, c'est-à-dire un court-circuit involontaire entre deux pistes. Le risque est d'autant plus élevé autour du LM386, dont les huit broches du boîtier DIP-8 sont rapprochées. Le brasage de ce composant a donc demandé un soin particulier : dosage de l'étain, panne propre et bien chauffée, et vérification systématique de l'absence de pont entre broches adjacentes après chaque point.

La validation finale a combiné l'inspection visuelle, la recherche de courts-circuits et les mesures de continuité décrites plus haut. La carte a ensuite été câblée à l'ensemble du montage, le connecteur Audio_In J2 recevant le signal issu de la patte RA6 du PIC et le connecteur Audio_out J3 alimentant le haut-parleur. Les figures suivantes montrent la carte son raccordée. La mise sous tension via l'interrupteur SW1, suivie de l'écoute du son produit par le jeu et du réglage du niveau à l'aide du potentiomètre RV1, a confirmé le bon fonctionnement de l'ensemble. La carte son est ainsi passée avec succès du schéma au circuit fabriqué, puis validé en conditions réelles.

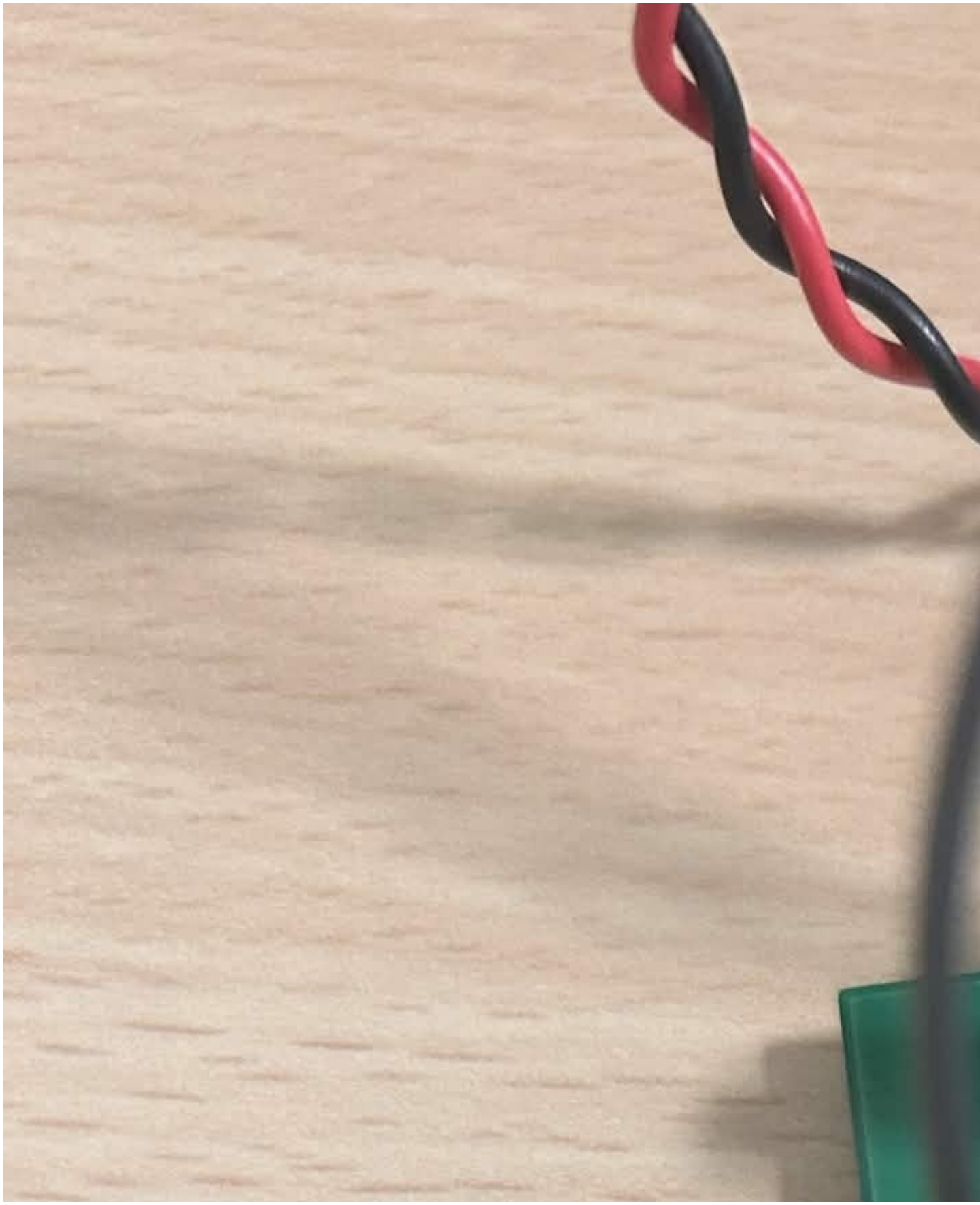


Figure 19 — carte son raccordée au reste du montage pour la validation fonctionnelle.

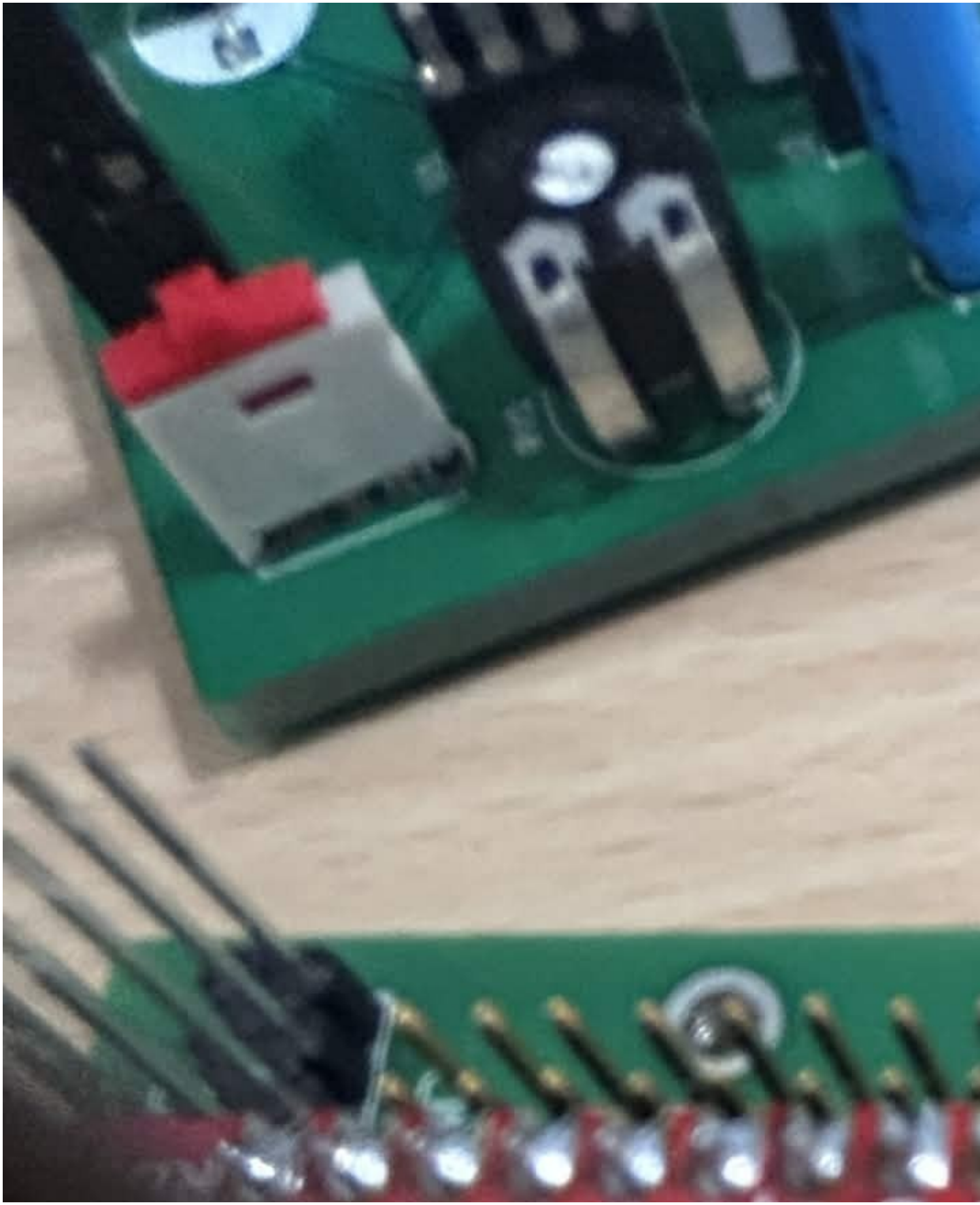


Figure 20 — détail du raccordement des connecteurs d'alimentation, d'entrée et de sortie de la carte son.

7. L’ecran ILI9488 et son pilote bas niveau (SPI)

L’affichage est le coeur visible du jeu : c’est sur lui que se dessinent le vaisseau, les asteroides, les projectiles et le score. Pour le piloter, le programme s’appuie sur une bibliotheque bas niveau, ili9488.c/ili9488.h, qui parle directement au controleur de la dalle. Cette section decrit le materiel, le bus de communication, l’origine du code et les fonctions essentielles, en mettant en regard chaque choix avec l’idee directrice du projet : menager un microcontroleur lent et depourvu de calcul flottant materiel.

7.1 L’ecran ILI9488 et ses 8 bornes utilisees

L’ecran retenu est une dalle TFT de 480 x 320 pixels en mode paysage, pilotee par le controleur ILI9488. La resolution native du module est 320 x 480 en portrait ; le programme la fait basculer en paysage par logiciel (voir 7.5). La couleur est transmise au controleur en 18 bits, soit trois octets correspondant aux composantes Rouge, Vert et Bleu. La liaison entre le PIC16F18877 et l’ecran est un bus SPI a sens unique : le microcontroleur envoie des donnees vers l’ecran (ligne MOSI), mais ne lit jamais en retour.

Au dos du module, treize bornes sont disponibles ; seules les huit premieres sont utilisees ici, dans l’ordre : VCC (alimentation), GND (masse), SCLK (horloge SPI), MOSI (donnees), MISO (retour, present mais non cable cote programme), CS (selection du composant), RES (reset) et DC (Data/Command). Les bornes BKL, SCL, SDA, INT et SDCS, situees au-dela de la huitieme, ne servent pas dans ce projet : la dalle est commandee exclusivement par son interface SPI a quatre fils, sans recourir a l’eventuelle interface I2C ni au lecteur de carte micro-SD integre.

Figure 21 — Face arriere de l’ecran : reperage des bornes VCC, GND, SCLK, MOSI, MISO, CS, RES et DC.

Le cablage effectif entre les bornes de l’ecran et le PIC est le suivant : CS sur RC0 (patte 15), RES sur RC1 (patte 16), DC sur RC2 (patte 17), SCLK sur RC3 (patte 18) et MOSI sur RC5 (patte 20). La ligne MISO de l’ecran existe mais reste inutilisee, puisque le programme n’interroge jamais le controleur.

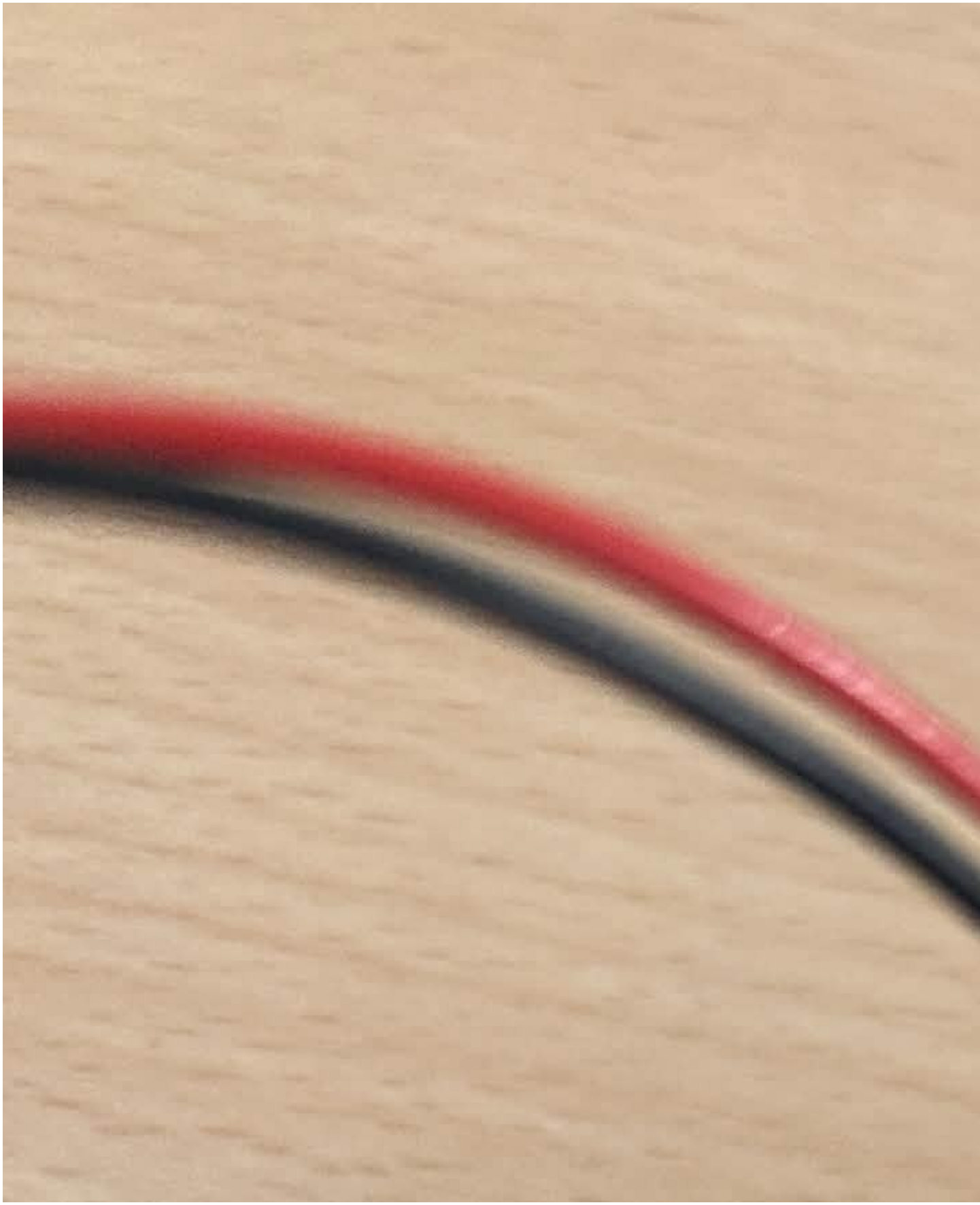


Figure 22 — Raccordement des huit bornes de l'ecran au PCB de la manette.

7.2 Le bus SPI et le role de chaque ligne

La communication s'effectue par le peripherique SPI1 (MSSP1) du PIC, configure en mode Maitre, SPI Mode 0. Le SPI est un bus synchrone : une horloge cadence l'echange et chaque front transfere un bit. Cinq lignes interviennent.

Figure 23 — Lignes SCK, MOSI, CS, DC et RST entre le PIC (maitre) et le controleur ILI9488 (esclave).

- **SCK (horloge)** : generee par le PIC, elle rythme la transmission. Chaque coup d'horloge fait avancer un bit. En mode 0, la donnee est echantillonnee sur le front montant, l'horloge etant au repos a l'etat bas.
- **MOSI (Master Out, Slave In)** : transporte les donnees du PIC vers l'ecran. C'est l'unique sens utile ici.
- **CS (Chip Select)** : selectionne l'ecran. Maintenue a l'etat bas pendant un echange, elle indique au controleur que les bits qui defilent lui sont destines.
- **DC (Data/Command)** : distingue une commande (DC = 0) d'une donnee (DC = 1). Le meme octet sur MOSI sera interprete comme un registre a configurer ou comme un pixel a afficher selon l'etat de DC.
- **RST (Reset)** : reinitialise le controleur au demarrage.

La communication est donc a sens unique (PIC vers ecran). On ne relit jamais le controleur : cela simplifie le code et evite des temporisations couteuses, au prix de l'impossibilite de verifier l'etat interne de la dalle. C'est un compromis assume, coherent avec le fil rouge du projet, qui privilegie la rapidite et la simplicite.

7.3 Origine de la bibliotheque

Le pilote bas niveau n'a pas ete ecrit integralement par le binome. Il provient d'une bibliotheque libre publiee par l'auteur timagr615, prevue a l'origine pour un microcontroleur STM32 utilisant la couche d'abstraction materielle HAL. Le professeur E. Nativel l'a portee vers l'environnement PIC/MCC : les appels HAL de STM32 ont ete remplaces par les

fonctions generees par MCC (MPLAB Code Configurator), notamment l'echange SPI et le pilotage des broches. Cette base, fournie par l'encadrant, constitue la couche `ili9488.c/.h` ; le travail du binome a consiste a l'utiliser et a batir au-dessus la logique du jeu.

7.4 Les fonctions cles du pilote

Envoyer une commande ou une donnee

Tout repose sur deux briques : envoyer un octet en mode commande, ou en mode donnee. La difference tient uniquement a l'etat de la ligne DC, le reste de la sequence etant identique : abaisser CS, transmettre l'octet par SPI, relever CS.

```
void ILI9488_SendCommand(uint8_t cmd) {
    DC_SetLow();            // DC = 0 : l'octet est une commande
    CS_SetLow();            // selection de l'ecran
    SPI1_ExchangeByte(cmd);   // emission de l'octet sur MOSI
    CS_SetHigh();           // fin de l'echange
}

void ILI9488_SendData(uint8_t data) {
    DC_SetHigh();           // DC = 1 : l'octet est une donnee
    CS_SetLow();
    SPI1_ExchangeByte(data);
    CS_SetHigh();
}
```

Ligne par ligne : `DC_SetLow()` / `DC_SetHigh()` positionnent la ligne Data/Command, ce qui dicte l'interpretation cote controleur. `CS_SetLow()` ouvre la transaction ; `SPI1_ExchangeByte()` pousse les huit bits sur MOSI (le PIC etant maitre, c'est lui qui cadence l'horloge) ; `CS_SetHigh()` referme la transaction. On notera que `SPI1_ExchangeByte` est nominalement bidirectionnelle (elle renvoie

aussi un octet recu), mais sa valeur de retour est ici ignoree : le sens PIC vers ecran est le seul exploite. Le choix d'encadrer chaque octet par CS rend les appels independants et robustes, au prix de quelques basculements de broche supplementaires.

Initialiser le controleur

`ILI9488_Init` envoie la sequence d'initialisation recommandee par le fabricant. Elle enchaine, via `ILI9488_SendCommand` puis `ILI9488_SendData`, plusieurs registres : les courbes de gamma positive et negative (0xE0 et 0xE1) qui ajustent le rendu des niveaux, les controles d'alimentation (Power Control 0xC0, 0xC1 et VCOM 0xC5), l'orientation memoire MADCTL (0x36), et surtout le format de pixel (0x3A) regle a 0x66 pour selectionner le mode 18 bits par pixel. La sequence se termine par la sortie de veille SLPOUT puis l'allumage de l'affichage DISPON. Sans cette sequence, le controleur reste muet ou affiche du bruit : c'est elle qui amene la dalle dans un etat connu et exploitable.

Definir la fenetre d'adressage

Avant d'ecrire des pixels, il faut indiquer au controleur dans quel rectangle de sa memoire ils vont aller. C'est le role de `setAddrWindow` :

```
void setAddrWindow(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1){
    ILI9488_SendCommand(ILI9488_CASET);
    ILI9488_SendData(x0 >> 8);
    ILI9488_SendData(x0 & 0xFF);
    ILI9488_SendData((x0+x1-1) >> 8);
    ILI9488_SendData((x0+x1-1) & 0xFF);
    ILI9488_SendCommand(ILI9488_PASET);
    ILI9488_SendData(y0 >> 8);
    ILI9488_SendData(y0 & 0xff);
    ILI9488_SendData((y0+y1-1) >> 8);
```

```
    ILI9488_SendData((y0+y1-1) & 0xff);
    ILI9488_SendCommand(ILI9488_RAMWR);
}
```

Le registre CASET (0x2A, Column Address Set) fixe la plage de colonnes ; PASET (0x2B, Page Address Set) celle des lignes. Comme les adresses tiennent sur 16 bits mais que le bus transmet des octets, chaque coordonnee est coupee en deux : `x0 >> 8` extrait l'octet de poids fort par decalage de huit bits vers la droite, et `x0 & 0xFF` isole l'octet de poids faible par un masque (ET binaire avec 0xFF). On envoie donc poids fort puis poids faible. L'adresse de fin vaut `x0 + x1 - 1`, ce qui revient a interpreter le quatrieme parametre `x1` comme une largeur (idem `y1` comme une hauteur) : le pixel de fin est inclus, d'ou le « -1 ». Enfin RAMWR (0x2C, Memory Write) signale que les octets suivants seront des donnees de pixels : toute donnee envoyee ensuite remplit la fenetre, colonne apres colonne, le controleur incrementant tout seul l'adresse interne. Ce mecanisme de fenetre est central pour l'optimisation du jeu : il permet de ne reecrire qu'un petit rectangle (un projectile, un asteroide) sans toucher au reste de l'ecran.

Convertir une couleur 16 bits en 18 bits

Le code de plus haut niveau manipule des couleurs au format RGB565 (16 bits : 5 bits de rouge, 6 de vert, 5 de bleu). Or le controleur attend ici 18 bits, soit trois octets pleins R, G, B (six bits utiles par composante). `write16BitColor` realise la conversion par une regle de trois sur les plages respectives 31, 63, 31 :

```
void write16BitColor(uint16_t color){
    uint8_t r = (color >> 11) & 0x1F;   // 5 bits de rouge   (0..31)
    uint8_t g = (color >> 5)  & 0x3F;   // 6 bits de vert    (0..63)
    uint8_t b =  color        & 0x1F;   // 5 bits de bleu    (0..31)
    ILI9488_SendData((r * 255) / 31);   // remise a l'echelle sur 0..255
    ILI9488_SendData((g * 255) / 63);
    ILI9488_SendData((b * 255) / 31);
```

```
}
```

Les decalages et masques extraient chaque composante a sa place dans le mot de 16 bits : le rouge occupe les bits 11 a 15, le vert les bits 5 a 10, le bleu les bits 0 a 4. La multiplication par 255 suivie d'une division par la plage d'origine (31 pour le rouge et le bleu, 63 pour le vert) etire chaque valeur sur un octet complet 0..255, le controleur n'en conservant que les six bits de poids fort. On garde des entiers du debut a la fin : aucune operation flottante, conformement au fil rouge du projet.

7.5 setRotation(3) : passer en paysage 480 x 320

L'ecran etant nativement portrait (320 x 480), le jeu force le mode paysage par `setRotation(3)`. Cette fonction ecrit dans le registre MADCTL (0x36, Memory Access Control), qui controle le sens de balayage et l'echange des axes X/Y. La valeur correspondant a l'orientation 3 inverse et echange les axes de sorte que l'origine se place dans le coin attendu et que l'espace utile devienne 480 x 320. Tout le reste du programme raisonne ensuite avec `SCREEN_W = 480` et `SCREEN_H = 320`, place la bande de score sur une hauteur `HUD_H = 24` en haut, et fixe le centre du jeu en (CX = 240, CY = 172), c'est-a-dire au milieu de la zone de jeu situee sous le HUD.

7.6 Pourquoi 18 bits = 3 octets, et l'impact sur le debit SPI

En mode 18 bits, chaque pixel est decrit par six bits de rouge, six de vert et six de bleu. Comme le bus SPI transmet par paquets de huit bits, le controleur attend un octet par composante, soit **trois octets par pixel** (les deux bits inutiles de chaque octet sont ignores). C'est plus simple a transmettre que de tasser 18 bits a cheval sur des octets, mais cela gaspille six bits sur vingt-quatre transmis.

La consequence est directe sur le debit. Remplir l'ecran entier represente 480 x 320 = 153 600 pixels, donc 153 600 x 3 = 460 800 octets a faire transiter sur MOSI, chacun necessitant au moins huit coups d'horloge SPI, soit plus de 3,6 millions de bits pour un seul rafraichissement complet. Sur un microcontroleur cadence a FOSC = 32 MHz (cycle instruction de 125 ns), repeindre tout l'ecran a chaque tour de boucle serait prohibitif. C'est precisement ce qui justifie la strategie de rendu du jeu : ne jamais repeindre tout l'ecran en boucle, mais effacer puis redessiner uniquement les petits objets qui bougent, en s'appuyant sur la fenetre d'adressage de `setAddrWindow` pour ne streamer que les quelques octets d'un projectile ou d'un asteroide.

Le pilote bas niveau et la contrainte du materiel dictent ainsi l’architecture graphique de tout le programme.

8. La bibliotheque graphique (GFX) et les fonctions de trace

La partie precedente a decrit le pilote bas niveau ili9488.c, qui sait piloter l'ecran ILI9488 sur le bus SPI : envoyer une commande (DC=0), envoyer une donnee (DC=1), ouvrir une fenetre d'adressage (CASET 0x2A, PASET 0x2B, RAMWR 0x2C) et allumer un pixel. Ce pilote est cependant trop primitif pour ecrire directement le jeu : il ne sait dessiner ni un cercle, ni un triangle, ni du texte. C'est le role de la couche intermediaire GFX_Library.c/.h, portage de la celebre bibliotheque libre Adafruit GFX, qui fournit les primitives geometriques de plus haut niveau. Ce chapitre detaille cette couche, ses principaux algorithmes de trace (ligne de Bresenham, cercle par le point milieu, texte par police bitmap) et les bibliotheques C standard sur lesquelles le jeu s'appuie.

8.1 Role de GFX_Library par-dessus le pilote ILI9488

La bibliotheque graphique se place exactement entre le code du jeu (main.c) et le pilote materiel (ili9488.c). Elle expose un jeu de fonctions display_* independantes du materiel : display_drawLine, display_drawRect, display_drawCircle et display_fillCircle, display_drawTriangle et display_fillTriangle, display_drawRoundRect, display_printText et display_drawChar, ou encore display_drawBitmap. L'idee fondamentale est la separation des responsabilites : GFX raisonne en geometrie (un cercle de rayon r centre en (x0, y0)), puis traduit cette geometrie en une suite d'allumages de pixels, et delegue chaque pixel au pilote. Si l'on changeait un jour d'ecran, il suffirait de reecrire le pilote ; la couche GFX et le jeu resteraient inchanges.

Cette traduction repose sur quelques macros qui font le pont vers le pilote : display_drawPixel renvoie vers drawPixel, display_fillScreen vers fillScreen, display_fillRect vers fillRect et display_setRotation vers setRotation. Toutes les primitives de GFX finissent donc, directement ou indirectement, par appeler drawPixel (allumage unitaire) ou fillRect (remplissage rapide d'un rectangle en streamant les octets RGB sur le SPI). Cette organisation est cruciale pour un microcontroleur lent comme le PIC16F18877 : l'oscillateur interne HFINTOSC delivre FOSC = 32 MHz, soit une horloge d'instruction FOSC/4 = 8 MHz (Tcy = 125 ns). Chaque pixel coute cher en transmission SPI (la couleur est envoyee sur 18 bits, soit 3 octets R, G, B), donc les algorithmes doivent etre frugaux et n'allumer que les pixels strictement utiles.

8.2 Le trace de ligne par l'algorithme de Bresenham

Tracer une droite entre deux points entiers (x0, y0) et (x1, y1) parait trivial avec un calcul de pente y = a*x + b, mais cette approche imposerait une division et des multiplications flottantes pour chaque colonne. Sur un PIC16 depourvu d'unite de calcul flottant materielle, ce serait redhibitoire. La fonction writeLine emploie donc l'algorithme de Bresenham, qui ne manipule que des entiers et une simple addition par pixel.

Le principe est le suivant. On suppose d'abord que la pente est faible (la ligne avance surtout horizontalement). On parcourt alors x de x0 a x1 en allumant un pixel par colonne. A chaque pas, on accumule l'ecart vertical dans un terme d'erreur entier ; lorsque cet ecart depasse une demi-colonne, on incremente y d'une unite et on retranche la largeur totale de l'erreur. Formellement, en posant dx = |x1 - x0| et dy = |y1 - y0|, l'erreur est initialisee a dx/2 puis decrementee de dy a chaque colonne ; lorsqu'elle devient negative, on avance y et on lui rajoute dx. Aucun flottant n'intervient : la decision se prend uniquement sur le signe d'un entier.

Pour traiter une ligne de forte pente (presque verticale), l'algorithme utilise un indicateur de pente raide, le drapeau steep. Si |y1 - y0| depasse |x1 - x0|, on echange mentalement les roles de x et de y (on echange x0 avec y0, x1 avec y1), de sorte que la boucle principale itere toujours sur l'axe le plus long. Au moment d'allumer le pixel, on inverse de nouveau les coordonnees pour steep, ce qui revient a refleter la ligne par la premiere bissectrice. On garantit ainsi qu'il y a exactement un pixel allume par valeur de l'axe dominant, sans trou ni doublon. Cet algorithme est exactement celui qui trace les trois aretes du vaisseau triangulaire (display_drawTriangle) et les huit cotes de l'octogone des asteroides.

Deux cas particuliers tres frequents beneficient d'un chemin rapide dedie : les lignes parfaitement horizontales et verticales. drawFastHLine et drawFastVLine court-circuitent Bresenham car la coordonnee constante ne demande aucun calcul d'erreur ; elles remplissent un segment d'un coup, ce qui est nettement plus efficace. Ce sont elles qui dessinent par exemple la bande de score (HUD, de hauteur HUD_H = 24 pixels) en haut de l'ecran et, comme on va le voir, le remplissage des cercles.

8.3 Le cercle plein par l'algorithme du point milieu

Les projectiles tires par le vaisseau (touche TIR, bouton B sur RD6, patte 29) sont de petits disques de rayon `BULLET_RADIUS` = 2 pixels. Ils sont dessines par `display_fillCircle`, dont voici le code :

```
void display_fillCircle(uint16_t x0, uint16_t y0, uint16_t r, uint16_t color){
    drawFastVLine(x0, y0-r, 2*r+1, color);
    display_fillCircleHelper(x0, y0, r, 3, 0, color);
}
```

Analysons ce code ligne par ligne. La premiere ligne, `drawFastVLine(x0, y0-r, 2*r+1, color)`, trace le diametre vertical du cercle : un segment vertical qui part du sommet (y0 - r) et descend sur 2*r + 1 pixels de haut, c'est-a-dire le diametre complet (r pixels au-dessus du centre, r en dessous, plus le pixel du centre). On profite ici du chemin rapide vertical de la section precedente pour ce trait central, plutot que de l'allumer pixel par pixel. La seconde ligne, `display_fillCircleHelper(x0, y0, r, 3, 0, color)`, remplit les deux moities laterales du disque autour de ce diametre. Le parametre `3` est un masque de quadrants : ses deux bits actifs (binaire 11) designent simultanement les cotes droit et gauche, ce qui produit un disque complet. Le `0` final est un decalage (delta) ajoute aux segments, inutile ici pour un cercle plein mais essentiel pour dessiner les coins de rectangles arrondis (`display_drawRoundRect`), ou le meme helper sert a tracer un quart de disque ecarte du centre.

Le remplissage repose sur l'algorithme du point milieu, qui calcule le contour du cercle uniquement avec des entiers, sans aucune racine carree ni sinus. Il s'appuie sur trois variables entieres : un terme de decision `f`, et deux increments `ddF_x` et `ddF_y`. On part du sommet du cercle et l'on avance d'un pixel a la fois sur l'axe x ; le signe de `f` indique s'il faut aussi descendre d'un pixel sur l'axe y pour rester au plus pres du cercle ideal d'equation x^2 + y^2 = r^2. A chaque iteration, `f` est mis a jour par de simples additions de `ddF_x` et `ddF_y` (typiquement ddF_x augmente de 2 et ddF_y de 2 a chaque pas). Ce schema evite totalement l'evaluation de `sqrt` et de fonctions trigonometriques, conformement au fil rouge du projet : on menage un microcontroleur lent en bannissant le flottant.

L'autre astuce de l'algorithme est l'exploitation de la symetrie en huit octants : un cercle est symetrique par rapport a ses deux axes et a ses deux diagonales. Il suffit donc de calculer un seul huitieme du contour, puis de reporter chaque point calcule dans les sept autres octants par simple changement de signe et echange de coordonnees. Pour le remplissage, le helper ne dessine pas des points isoles mais des segments verticaux (a l'aide de `drawFastVLine`) reliant les points symetriques de haut en bas : on balaie ainsi la surface du disque colonne par colonne. C'est ce mecanisme qui produit les projectiles a l'ecran. L'octogone des asteroides, lui, n'utilise pas ce cercle plein : il est forme de huit sommets precalcules astX[8] = {10,7,0,-7,-10,-7,0,7} et astY[8] = {0,7,10,7,0,-7,-10,-7} (rayon 10, mis a l'echelle par AST_RADIUS/10 = 12/10), relies par les segments de Bresenham.

8.4 Le texte : police bitmap 5x7

L'affichage du score (+10 par asteroide detruit), des libelles du menu et du compte a rebours passe par `display_printText`, qui appelle `display_drawChar` pour chaque caractere de la chaine. La police est une police bitmap stockee dans le tableau `font[256][5]`, indexe par le code ASCII du caractere. Chaque caractere occupe 5 colonnes ; chaque colonne est un octet de 8 bits dont les bits a 1 indiquent les pixels a allumer de haut en bas. On obtient une matrice de 5 pixels de large sur 7 utiles (la police est dite 5x7), la huitieme ligne servant d'espacement vertical.

`display_drawChar` lit donc les 5 octets du caractere voulu, puis, pour chaque colonne et chaque bit, allume ou non le pixel correspondant. L'agrandissement est gere par un parametre de taille `size` : au lieu d'allumer un pixel unitaire, la fonction dessine un petit carre plein de `size` x `size` pixels par `display_fillRect`. Ainsi, un texte de taille 2 occupe quatre fois plus de surface qu'un texte de taille 1, sans qu'il faille stocker une seconde police : un seul jeu de donnees 5x7 suffit pour toutes les tailles, ce qui economise la memoire programme du PIC.

8.5 Les bibliotheques C standard utilisees par le jeu

Le fichier `main.c` inclut un ensemble restreint de bibliotheques standard, chacune choisie pour un usage precis et compatible avec les contraintes du microcontroleur :

- `<stdint.h>` fournit les types entiers de taille fixe (`uint8_t`, `uint16_t`, `int8_t`). Les utiliser plutot que `int` permet de dimensionner au plus juste chaque variable et donc d'economiser la RAM, ressource rare sur un PIC16 8 bits : un compteur qui ne depasse jamais 255 tient

dans un seul octet.

- <stdbool.h> apporte le type bool et les valeurs true/false, qui rendent le code lisible pour les drapeaux d'etat (objet actif ou non, touche pressee ou non).
- <stdlib.h> est inclus pour rand(), generateur de nombres pseudo-aleatoires qui sert a faire surgir les asteroides depuis des bords et avec des trajectoires variables.
- <stdio.h> est inclus pour sprintf, qui fabrique les chaines de caracteres du score (conversion d'un entier en texte avant l'affichage par display_printText).
- <math.h> fournit sqrtf et la constante M_PI. Conformement au fil rouge, ces outils flottants sont evites autant que possible pendant le jeu ; lorsqu'une distance doit etre testee (collision projectile-asteroide ou asteroide-vaisseau), on compare des distances au carre pour ne pas appeler sqrtf.
- enfin mcc_generated_files/mcc.h rassemble les fonctions generees par MCC (MPLAB Code Configurator) : SYSTEM_Initialize, les fonctions ADCC_* pour la lecture du joystick, SPI1_Open pour ouvrir le bus de l'ecran, ainsi que les macros d'acces aux broches. C'est l'interface vers les peripheriques materiels.

8.6 Le principe effacer / deplacer / redessiner

Toutes ces primitives ne servent leur but que si le rendu reste fluide sur un materiel lent. Comme l'ecran TFT n'a pas de double tampon et que chaque pixel envoye coute trois octets sur le SPI, le jeu n'efface jamais l'ecran entier a chaque image. Il applique au contraire un cycle local pour chaque objet mobile (vaisseau, projectiles, asteroides, etoiles de fond) : on efface l'objet a son ancienne position (en le redessinant avec la couleur du fond), on met a jour ses coordonnees, puis on le redessine a sa nouvelle position. Seuls les quelques pixels reellement modifies transitent sur le bus, ce qui maintient une cadence acceptable.

Figure 24 — Principe du rendu incremental : effacer l'objet a l'ancienne position, deplacer, puis redessiner a la nouvelle position.

Ce pipeline incremental boucle la presentation de la couche graphique : la bibliotheque GFX fournit des primitives geometriques economes (lignes de Bresenham, cercles par le point milieu, texte bitmap reutilisant une seule police), le pilote ILI9488 les transforme en octets SPI, et le code du jeu orchestre le tout en ne redessinant que ce qui change. L'ensemble illustre la

philosophie constante du projet : produire un jeu reactif tout en menageant un microcontroleur 8 bits sans calcul flottant materiel.

9. Architecture logicielle et machine a etats

Le programme main.c constitue le coeur du jeu : il orchestre l'initialisation du microcontroleur, l'affichage, la lecture des commandes et le deroulement de la partie. Sa structure a ete pensee autour d'une contrainte centrale rappelee tout au long de ce rapport : le PIC16F18877 est un microcontroleur 8 bits cadence a FOSC = 32 MHz (l'horloge interne HFINTOSC), soit un cycle d'instruction de FOSC/4 = 8 MHz, c'est-a-dire 125 ns par cycle. Surtout, ce coeur ne dispose d'**aucune unite de calcul flottant materielle** : les operations float (cos, sin, sqrt) sont emulees par logiciel et donc lentes. Le fil rouge de toute la conception logicielle decoule de ce constat : eviter les calculs flottants pendant le jeu, ne jamais bloquer la boucle inutilement, ne redessiner que ce qui change, et comparer les distances au carre pour eviter les racines carrees. Chaque choix d'architecture vise ainsi la lisibilite du code et l'economie des ressources. Ce chapitre presente, de haut en bas, l'organisation de main.c.

9.1 Environnement de developpement : MPLAB X, XC8 et MCC

Le programme est ecrit, compile et televerse depuis l'environnement de developpement integre **MPLAB X IDE** de Microchip. La compilation est assuree par le compilateur **XC8**, specialise pour les microcontroleurs PIC 8 bits, qui traduit le code C en code machine optimise pour l'architecture du PIC16.

Une partie de l'initialisation n'est pas ecrite a la main mais generee automatiquement par le **MCC (MPLAB Code Configurator)**. Cet outil graphique permet de configurer les peripheriques (oscillateur HFINTOSC, convertisseur ADCC, bus SPI1/MSSP1, generateur de son NCO1, broches d'entree-sortie) via une interface, puis produit un ensemble de fichiers ranges dans le repertoire mcc_generated_files. C'est ce dossier qui fournit notamment la fonction SYSTEM_Initialize, les fonctions du convertisseur ADCC_*, l'ouverture du bus SPI SPI1_Open, ainsi que les macros d'acces aux broches (par exemple BTN_B_GetValue()). L'interet de cette approche est double : elle reduit le risque d'erreur dans la configuration des registres (souvent fastidieuse et source de bogues) et elle permet de se concentrer sur la logique du jeu plutot que sur les details du materiel. Le fichier d'en-tete mcc_generated_files/mcc.h est donc inclus dans main.c pour donner acces a toutes ces fonctions et macros.

A cote de cet en-tete genere, main.c inclut plusieurs bibliotheques C standard, chacune choisie

pour un role precis et compatible avec les contraintes de la cible : `<stdint.h>` pour les types entiers de taille fixe (`uint8_t`, `uint16_t`, `int8_t`) qui permettent de doser finement l'occupation de la RAM ; `<stdbool.h>` pour le type `bool` ; `<stdlib.h>` pour `rand()`, utilise pour faire apparaitre les asteroides de maniere aleatoire ; `<stdio.h>` pour `sprintf`, qui fabrique les chaines de caracteres du score ; et `<math.h>` pour `sqrtf` et la constante `M_PI`. Ces fonctions flottantes restent volontairement cantonnees aux phases non critiques (preparation, affichage du score) et non au coeur de la boucle de jeu.

9.2 Les `#define` de reglage

Le programme regroupe en tete de fichier un ensemble de constantes definies par directive `#define`. Plutot que de disperser des nombres magiques au fil du code, on leur donne un nom parlant. Les principales constantes sont :

```
#define SCREEN_W 480
#define SCREEN_H 320
#define HUD_H 24
#define CX 240          // centre X = SCREEN_W/2
#define CY 172          // centre Y = HUD_H + (SCREEN_H-HUD_H)/2
#define STAR_COUNT 18
#define COL_ORANGE 0xFD20
#define COL_ORANGE_DARK 0xCB20
#define JOY_DEADZONE 32
#define JOY_Y_SCALE 1.04
#define ANGLE_REDRAW_DOT 0.9997
#define SHIP_FRONT 18
#define SHIP_BACK 11
#define SHIP_RADIUS 18
```

```
#define MAX_BULLETS 4
#define BULLET_SPEED 9.0
#define BULLET_RADIUS 2
#define BULLET_LIFE 60
#define MAX_ASTEROIDS 4
#define AST_RADIUS 12
#define AST_SPEED 4.0
#define SPAWN_DELAY 19
```

Le recours systematique aux `#define` repond a trois objectifs precis. D'abord la **lisibilite** : lire `if (dist2 < AST_RADIUS*AST_RADIUS)` est beaucoup plus parlant que `if (dist2 < 144)`. Ensuite le **reglage en un seul endroit** : pour rendre le jeu plus difficile, il suffit de modifier `AST_SPEED` ou `SPAWN_DELAY` a un unique emplacement, sans risquer d'oublier une occurrence ailleurs dans le code. Enfin l'**economie de memoire** : une directive `#define` est traitee par le preprocesseur avant la compilation. Le nom est purement et simplement remplace par sa valeur dans le code source ; il ne consomme donc **aucun octet de RAM ni de ROM** a l'execution, contrairement a une variable globale qui occuperait une case memoire. Sur un microcontroleur dont la memoire est comptee, cet aspect n'est pas anecdotique.

Ces constantes traduisent egalement directement les regles du jeu et la geometrie de l'ecran. `SCREEN_W = 480` et `SCREEN_H = 320` decrivent la dalle TFT en mode paysage ; `HUD_H = 24` reserve une bande de score en haut ; `CX = 240` et `CY = 172` materialisent le fait que le vaisseau reste fixe au centre de la zone de jeu, sous la bande de score (`CY = HUD_H + (SCREEN_H - HUD_H)/2`). `STAR_COUNT = 18` fixe le nombre d'etoiles du fond. Les limites `MAX_BULLETS = 4` et `MAX_ASTEROIDS = 4` bornent les tableaux d'objets et donc la RAM mobilisee, tandis que `JOY_DEADZONE = 32` definit la zone morte du joystick (en deca de laquelle on ignore le mouvement pour ne pas faire tourner le vaisseau en permanence) et `ANGLE_REDRAW_DOT = 0.9997` fixe le seuil de produit scalaire au-dela duquel l'orientation est consideree comme inchangee : si le vaisseau n'a tourne que de moins d'environ 1,4 degre, on ne le redessine pas, ce qui evite un travail graphique inutile.

9.3 Les types : enum d'etat et structures

Le jeu fonctionne comme une machine a quatre etats. Pour representer proprement cet etat, le programme definit un type enumere :

```
typedef enum { ETAT_MENU, ETAT_REGLAGES, ETAT_JEU, ETAT_GAMEOVER } EtatJeu;
```

Un `enum` associe un nom symbolique a une valeur entiere (ici 0, 1, 2, 3 dans l'ordre). Manipuler `etat == ETAT_JEU` est a la fois plus lisible et plus sur qu'un test sur une valeur numerique brute, car le compilateur connait l'ensemble des valeurs possibles.

Les objets mobiles du jeu sont decrits par deux structures qui regroupent les donnees liees a un meme objet :

```
typedef struct { float x, y, vx, vy; uint8_t active, life; } Bullet;
```

```
typedef struct { float x, y, vx, vy; uint8_t active; } Asteroid;
```

Une structure rassemble dans une seule entite la position (`x`, `y`), le vecteur vitesse (`vx`, `vy`) et l'etat de l'objet. Le projectile (`Bullet`) possede en plus un champ `life`, une duree de vie en nombre de tours de boucle (limitee a `BULLET_LIFE = 60`) au-dela de laquelle il disparait. Le champ `active` indique si l'emplacement est occupe par un objet vivant. On remarque le choix du type `uint8_t` (entier non signe sur 8 bits, issu de `<stdint.h>`) pour `active` et `life` : un seul octet suffit a coder ces valeurs, ce qui economise la RAM par rapport a un `int` classique. Regrouper ainsi les donnees facilite la manipulation : on peut parcourir un tableau d'objets et traiter chacun comme un tout coherent.

9.4 Les variables globales

Plusieurs variables sont declarees au niveau global du fichier, donc accessibles par toutes les fonctions :

- `etat` : l'etat courant de la machine (de type `EtatJeu`) ;
- `bullets[4]` et `asteroids[4]` : les tableaux des projectiles et des asteroides, dimensionnes par `MAX_BULLETS` et `MAX_ASTEROIDS` ;
- `shipCos`, `shipSin` : les deux composantes du vecteur unitaire d'orientation du vaisseau ;

- `shipX1..shipX3` et `shipY1..shipY3` : les coordonnees ecran des trois sommets du triangle du vaisseau, memorisees pour pouvoir effacer l'ancienne position ;
- `score` : le score courant du joueur (chaque asteroide detruit rapporte 10 points).

Ces variables sont declarees globales parce qu'elles sont **partagees par plusieurs fonctions** appelees a des moments differents. Par exemple, `etat` est modifie dans `main()` mais lu egalement dans `boucleJeu()` ; les tableaux `bullets` et `asteroids` sont remplis lors d'un tir ou d'une apparition, deplaces a chaque tour de boucle, puis testes pour les collisions. Faire transiter toutes ces donnees en parametres de fonction alourdirait inutilement le code et consommerait de la pile a chaque appel. La memorisation des anciens sommets du vaisseau (`shipX1..shipY3`) illustre directement le principe d'optimisation graphique du projet : pour deplacer le vaisseau sans rafraichir tout l'ecran, on efface le triangle precedent grace a ses coordonnees memorisees, puis on dessine le nouveau. De meme, le couple `shipCos`/`shipSin` evite tout recours a `cos()` ou `sin()` pendant le jeu : ces deux composantes sont issues directement du vecteur normalise du joystick et representent deja le cosinus et le sinus de l'angle de visee.

9.5 La fonction `main()` et la boucle infinie

La fonction `main()` suit le squelette caracteristique des programmes pour microcontroleur : une phase d'**initialisation** executee une seule fois, suivie d'une **boucle infinie** `while(1)` qui ne s'arrete jamais (un microcontroleur n'a pas de systeme d'exploitation vers lequel rendre la main).

```
void main(void){
    SYSTEM_Initialize();
    ADCC_Initialize();
    SPI1_Open(SPI1_DEFAULT);
    ILI9488_Init();
    setRotation(3);
    NCO1INC = 0;
    afficherMenu();
```

```
    while(1){
        if(etat == ETAT_MENU){
            if(appuiC()){ initJeu(); etat = ETAT_JEU; }
            if(appuiD()){ afficherReglages(); etat = ETAT_REGLAGES; }
        }
        else if(etat == ETAT_REGLAGES){
            if(appuiC()){ afficherMenu(); etat = ETAT_MENU; }
        }
        else if(etat == ETAT_JEU){
            boucleJeu();
        }
        else if(etat == ETAT_GAMEOVER){
            if(appuiC()){ afficherMenu(); etat = ETAT_MENU; }
        }
    }
}
```

La phase d'initialisation se lit ligne par ligne. `SYSTEM_Initialize()` configure l'horloge et les peripheriques generes par le MCC. `ADCC_Initialize()` prepare le convertisseur analogique-numerique 10 bits pour la lecture des deux axes du joystick. `SPI1_Open(SPI1_DEFAULT)` ouvre le bus SPI1 en mode maitre avec la configuration par defaut (SPI Mode 0), indispensable au dialogue a sens unique PIC vers ecran. `ILI9488_Init()` envoie a l'ecran la sequence d'initialisation (gamma, controle d'alimentation, registre MADCTL, format de pixel 18 bits, sortie de veille, allumage). `setRotation(3)` place l'ecran en mode paysage 480x320 en ecrivant le registre d'orientation MADCTL. `NCO1INC = 0` met a zero l'incrementeur du generateur de son (NCO1) afin de garantir le silence au demarrage : la frequence produite valant `NCO1INC x 7,6294 Hz`, une

valeur nulle correspond a l'absence de signal. Enfin `afficherMenu()` dessine l'ecran d'accueil avant l'entree dans la boucle.

La boucle `while(1)` implemente la machine a etats sous la forme d'une cascade de tests `if / else if`. A chaque tour, le programme regarde la valeur de `etat` et n'execute que le bloc correspondant. Les **transitions** sont declenchees par les boutons : depuis `ETAT_MENU`, un appui sur le bouton C lance une partie (`initJeu()` reinitialise le jeu, puis `etat` passe a `ETAT_JEU`) et un appui sur le bouton D ouvre les reglages ; depuis `ETAT_REGLAGES`, l'appui sur C ramene au menu ; en `ETAT_JEU`, le bloc delegue tout le travail a `boucleJeu()` ; en `ETAT_GAMEOVER`, l'appui sur C renvoie au menu. On observe que chaque transition associe systematiquement un changement d'affichage et une mise a jour de la variable `etat`, ce qui garantit la coherence entre ce qui est dessine et l'etat logique du jeu. Le passage en `ETAT_GAMEOVER`, lui, n'est pas declenche par un bouton mais a l'interieur de `boucleJeu()`, lorsqu'un asteroide entre en collision avec le vaisseau.

Figure 25 — Les quatre etats du jeu et les transitions declenchees par les boutons.

9.6 La gestion des boutons par detection de front

Les boutons sont actifs a l'etat bas : une resistance de tirage (pull-up) maintient l'entree a 1 au repos, et un appui force l'entree a 0. Pour ne reagir qu'une seule fois au moment ou l'utilisateur enfonce le bouton (et non en continu tant qu'il reste appuye), le programme detecte le **front descendant** 1 vers 0. La fonction `appuiB()` illustre ce mecanisme :

```
bool appuiB(void){
    static uint8_t ancien = 1;
    uint8_t actuel = BTN_B_GetValue();
    bool appui = (ancien == 1 && actuel == 0);
    ancien = actuel;
    return appui;
```

```
}
```

La variable `ancien` est declaree `static` : elle conserve sa valeur d'un appel a l'autre, contrairement a une variable locale ordinaire qui serait reinitialisee a chaque entree dans la fonction. Elle memorise donc l'etat du bouton lors du tour de boucle precedent, initialise a 1 (repos). A chaque appel, `actuel` recupere l'etat present via `BTN_B_GetValue()`, la macro generee par le MCC qui lit la broche RD6 (patte 29, le bouton de tir). La condition `(ancien == 1 && actuel == 0)` n'est vraie que lors de la transition relache vers appui, c'est-a-dire le front descendant : un seul `true` est renvoye au moment precis de l'enfoncement, meme si le doigt reste pose plusieurs tours de boucle. La ligne `ancien = actuel` met a jour la memoire pour le prochain tour. Cette technique constitue un **anti-rebond logique** simple et joue le role d'une detection d'evenement : elle evite, par exemple, qu'un seul appui sur le bouton de tir ne genere une rafale ininterrompue de projectiles. Les fonctions `appuiC()` (bouton C sur RD5, patte 28 : JOUER/RETOUR/MENU) et `appuiD()` (bouton D sur RD4, patte 27 : REGLAGES) reposent exactement sur le meme principe pour leurs boutons respectifs. On retrouve ici, une fois encore, le souci d'un code leger et reactif, adapte a un microcontroleur lent : la detection de front ne coute que la comparaison de deux octets et une affectation, sans aucun calcul flottant ni temporisation bloquante.

10. Le joystick : lecture ADC et orientation du vaisseau

La fonction `lireJoystick()` constitue le cœur des commandes du jeu. C'est elle qui transforme le geste de la main du joueur en une orientation du vaisseau triangulaire affiché au centre de l'écran. Dans le jeu, le vaisseau ne se déplace pas : il reste au centre (CX = 240, CY = 172) et pivote sur place pour viser. C'est donc uniquement l'angle d'orientation qui est piloté par le joystick, et toute la difficulté consiste à le calculer de façon fiable sans surcharger un microcontrôleur lent. Ce chapitre détaille la chaîne complète, de la mesure des deux tensions analogiques jusqu'à l'astuce mathématique qui évite tout appel coûteux aux fonctions trigonométriques.

10.1 Le joystick analogique et le convertisseur ADC

Un joystick analogique est constitué de deux potentiomètres montés à angle droit, l'un pour l'axe X (horizontal) et l'autre pour l'axe Y (vertical). Chaque potentiomètre se comporte comme un diviseur de tension : la position du manche fixe une tension comprise entre 0 V (butée d'un côté) et la tension d'alimentation (butée de l'autre côté). Au repos, un ressort de rappel ramène le manche au centre, ce qui correspond théoriquement à la moitié de la plage.

Ces deux tensions sont mesurées par le convertisseur analogique-numérique du PIC16F18877, le périphérique ADCC, configuré en résolution 10 bits. Une mesure produit donc un nombre entier compris entre 0 et 1023 (soit 2^10 − 1). Sur la carte, l'axe X est câblé sur la broche RC7 (patte 22) et l'axe Y sur la broche RC6 (patte 21). Dans le code, ces deux canaux sont désignés par les symboles `poussoire_x` (axe X) et `poussoire_y` (axe Y).

La lecture d'un canal suit toujours le même protocole en trois temps : lancer la conversion, attendre qu'elle se termine, puis récupérer le résultat. C'est exactement ce que font les trois appels suivants, utilisés à plusieurs reprises dans ce chapitre :

```
ADCC_StartConversion(poussoire_x);
while(!ADCC_IsConversionDone());
int adcX = ADCC_GetConversionResult();
```

La première ligne déclenche la conversion du canal X. La deuxième est une boucle d'attente active (`while`) qui ne fait rien tant que la conversion n'est pas terminée : le `!` inverse le booléen, donc la boucle tourne « tant que la conversion n'est PAS finie ». La troisième ligne lit la valeur numérisée (0 à 1023) et la range dans `adcX`. Ces fonctions `ADCC_*` sont fournies par le code généré par MCC (MPLAB Code Configurator), qui se charge d'avoir configuré au préalable le multiplexeur d'entrée, l'horloge de conversion et le format du résultat. Cette attente est très courte (la conversion dure quelques microsecondes) et constitue l'un des rares moments où le programme se permet de bloquer, car il n'a rien d'utile à faire entre-temps : c'est une exception assumée au principe « ne jamais bloquer inutilement ».

10.2 La calibration : `calibrerJoystick()`

Un joystick réel ne renvoie presque jamais exactement la valeur centrale de 512 au repos. Les tolérances mécaniques des potentiomètres, l'usure du ressort de rappel et de légères différences d'alimentation décalent ce point de repos. Si l'on supposait naïvement que le centre vaut 512, le vaisseau dériverait en permanence vers une direction, même manche relâché. La calibration mesure donc le vrai centre du joystick au démarrage, et le mémorise dans deux variables `joyCx` (centre X) et `joyCy` (centre Y).

```
void calibrerJoystick(void){
    long sx=0, sy=0;
    for(uint8_t i=0;i<12;i++){
        ADCC_StartConversion(poussoire_x);
        while(!ADCC_IsConversionDone());
        sx += ADCC_GetConversionResult();
        ADCC_StartConversion(poussoire_y);
        while(!ADCC_IsConversionDone());
        sy += ADCC_GetConversionResult();
        __delay_ms(2);
```

```
    }
    joyCx = sx/12;
    joyCy = sy/12;
  }
```

Analysons ce code bloc par bloc. Les accumulateurs `sx` et `sy` sont déclarés en type `long` (entier signé large), et non en `int`. Ce choix est volontaire : on additionne 12 mesures pouvant chacune atteindre 1023, soit un total maximal de 12 × 1023 = 12 276. Un `int` 16 bits signé monte jusqu'à 32 767 et suffirait techniquement, mais le `long` garantit l'absence de débordement quelle que soit la configuration du compilateur, sans aucun risque. C'est une marge de sécurité peu coûteuse.

La boucle `for` itère 12 fois (le compteur `i` est un `uint8_t`, un entier 8 bits non signé, suffisant pour compter jusqu'à 12 et économe en RAM). À chaque tour, on lit l'axe X et on l'ajoute à `sx`, puis l'axe Y et on l'ajoute à `sy`. Le `__delay_ms(2)` introduit un délai de 2 ms entre deux mesures : il étale les lectures dans le temps afin que la moyenne capture le bruit réel du signal plutôt que 12 mesures quasi instantanées et corrélées.

Enfin, `joyCx = sx/12` et `joyCy = sy/12` calculent la moyenne arithmétique des 12 relevés. Moyenner lisse le bruit électrique : les petites fluctuations aléatoires positives et négatives se compensent, et l'on obtient une estimation stable du point de repos. Cette calibration suppose que le joueur ne touche pas au joystick pendant l'opération ; elle est donc réalisée une fois, au lancement.

10.3 `lireJoystick()` : la trigonométrie sans trigonométrie

C'est ici que se concentre l'idée la plus élégante du programme. Le PIC16 ne possède pas d'unité de calcul flottant matérielle : les fonctions `cos`, `sin`, `atan2` ou `sqrt` y sont émulées en logiciel et sont lentes. Or, pour orienter le vaisseau, on a besoin du cosinus et du sinus de l'angle visé. L'astuce consiste à les obtenir sans jamais appeler ces fonctions trigonométriques.

```
bool lireJoystick(void){
```

```
    ADCC_StartConversion(poussoire_x);
    while(!ADCC_IsConversionDone());
    int adcX = ADCC_GetConversionResult();
    ADCC_StartConversion(poussoire_y);
    while(!ADCC_IsConversionDone());
    int adcY = ADCC_GetConversionResult();
    int dx = (adcX - joyCx) * JOY_X_SIGN;
    int dy = -(adcY - joyCy);
    long mag2 = (long)dx*dx + (long)dy*dy;
    if(mag2 < (long)JOY_DEADZONE*JOY_DEADZONE) return false;
    float fx=(float)dx;
    float fy=(float)dy*JOY_Y_SCALE;
    float mag=sqrtf(fx*fx+fy*fy);
    if(mag<1.0f) return false;
    float newCos=fx/mag;
    float newSin=fy/mag;
    float dot=newCos*shipCos+newSin*shipSin;
    shipCos=newCos;
    shipSin=newSin;
    return (dot < ANGLE_REDRAW_DOT);
}
```

(a) L'écart au centre. Après avoir lu `adcX` et `adcY` comme expliqué plus haut, on calcule la position relative du manche par rapport au centre calibré. La ligne `int dx = (adcX - joyCx) *`

JOY_X_SIGN soustrait le centre mémorisé : si le manche est poussé d'un côté, dx est positif ; de l'autre, négatif ; au repos, proche de zéro. Le facteur JOY_X_SIGN est une constante de signe (+1 ou −1) qui permet d'inverser le sens de l'axe X selon l'orientation physique du joystick sur la carte (valeur à confirmer selon le câblage). Pour l'axe Y, int dy = -(adcY - joyCy) applique systématiquement un signe négatif. Cette inversion est nécessaire car, sur l'écran, l'axe vertical des pixels est orienté vers le bas (Y croît vers le bas), alors qu'en convention mathématique l'axe Y croît vers le haut. Le signe « − » remet l'axe Y dans le sens mathématique attendu, de sorte que pousser le manche vers le haut fasse pointer le vaisseau vers le haut de l'écran.

(b) La zone morte. Au repos, même après calibration, dx et dy oscillent légèrement à cause du bruit résiduel. Sans précaution, le vaisseau tremblerait en permanence. On calcule donc le carré de l'amplitude du déplacement : mag2 = dx*dx + dy*dy. Géométriquement, c'est le carré de la distance entre la position courante du manche et son centre. On compare ce carré au carré du seuil JOY_DEADZONE (qui vaut 32), c'est-à-dire 32 × 32 = 1024. Si mag2 < 1024, on considère que le manche est au repos et la fonction renvoie false sans rien modifier : le vaisseau ne bouge pas. Comparer les carrés (mag2 < JOY_DEADZONE²) plutôt que les distances elles-mêmes (mag < JOY_DEADZONE) évite une racine carrée coûteuse, tout en donnant exactement le même résultat puisque la fonction carré est strictement croissante sur les nombres positifs : c'est un principe directeur appliqué dans tout le code. On notera aussi les transtypages (long) qui forcent les multiplications à se faire en arithmétique large, là encore pour prévenir tout débordement entier (un dx de l'ordre de 500 donne déjà dx*dx = 250 000, bien au-delà de la capacité d'un int 16 bits).

(c) La normalisation, cœur de l'astuce. On convertit ensuite dx et dy en flottants fx et fy, en appliquant au passage le facteur JOY_Y_SCALE (1,04) sur l'axe Y pour compenser une légère asymétrie de course entre les deux axes du joystick. On calcule alors la norme du vecteur : mag = sqrtf(fx*fx + fy*fy), c'est-à-dire mag = $\sqrt{(fx^2 + fy^2)}$. Cette racine carrée, contrairement à celle que l'on a évitée à l'étape précédente, est indispensable ici, mais elle n'est exécutée que lorsque le manche est réellement poussé (le test de zone morte a déjà filtré tous les cas au repos). Le garde-fou if(mag < 1.0f) return false évite une division par zéro si la norme est négligeable.

Vient ensuite la double division newCos = fx/mag et newSin = fy/mag. Géométriquement, diviser un

vecteur par sa norme le ramène à une longueur de 1 : on obtient un **vecteur unitaire** pointant dans la direction du joystick. Or, sur le cercle trigonométrique de rayon 1, un point situé à l'angle θ a précisément pour coordonnées (cos θ, sin θ). La définition même du cosinus est cos θ = côté adjacent / hypoténuse, et celle du sinus sin θ = côté opposé / hypoténuse ; dans notre triangle rectangle, l'hypoténuse est `mag`, le côté adjacent (horizontal) est `fx` et le côté opposé (vertical) est `fy`. Par conséquent :

newCos = fx / mag = cos θ et newSin = fy / mag = sin θ

Les composantes du vecteur unitaire **sont** donc directement le cosinus et le sinus de l'angle visé. On obtient (cos θ, sin θ) sans jamais calculer θ ni appeler `atan2`, `cos` ou `sin`. La direction normalisée porte à elle seule toute l'information angulaire dont le vaisseau a besoin pour être dessiné. C'est l'illustration la plus pure du fil rouge du projet : exploiter une identité géométrique pour supprimer un calcul que le PIC16, dépourvu de FPU, exécuterait lentement.

Figure 26 — Du déplacement brut du joystick au vecteur unitaire d'orientation : la normalisation fournit directement cos et sin de l'angle visé.

(d) Le produit scalaire et le seuil de redessin. Une fois le nouveau vecteur d'orientation obtenu, faut-il réellement effacer puis redessiner le vaisseau ? Si l'orientation n'a quasiment pas changé, le faire serait du gaspillage de temps de calcul et provoquerait un scintillement. On compare donc la nouvelle direction à l'ancienne (`shipCos`, `shipSin`) à l'aide d'un produit scalaire : `dot = newCos*shipCos + newSin*shipSin`. Pour deux vecteurs unitaires, le produit scalaire vaut exactement le cosinus de l'angle qui les sépare. En effet, si l'ancienne direction est l'angle α et la nouvelle l'angle β, alors :

dot = cos β · cos α + sin β · sin α = cos(β − α)

ce qui n'est autre que la formule d'addition cos(a − b) = cos a cos b + sin a sin b appliquée à l'écart angulaire (β − α). Ainsi, si la direction n'a pas bougé, `dot` vaut cos(0) = 1 ; plus l'écart angulaire grandit, plus `dot` diminue. La constante `ANGLE_REDRAW_DOT` vaut 0,9997, ce qui correspond à un écart d'environ 1,4 degré (puisque arccos(0,9997) ≈ 1,4°). La fonction met d'abord à jour `shipCos` et `shipSin` avec la nouvelle orientation, puis renvoie `dot < ANGLE_REDRAW_DOT` : le résultat vaut `true` (il faut redessiner) seulement si le vaisseau a tourné

de plus de 1,4 degré environ, et `false` sinon. On évite ainsi des redessins inutiles tout en gardant une rotation parfaitement fluide à l'œil. Une nouvelle fois, c'est un produit scalaire — deux multiplications et une addition en flottant — qui remplace un calcul d'angle bien plus coûteux.

10.4 Bilan : une fonction taillée pour un microcontrôleur lent

`lireJoystick()` illustre à elle seule les quatre principes directeurs du projet. Elle évite les fonctions flottantes lourdes en exploitant l'identité entre vecteur unitaire et couple (cos, sin). Elle ne bloque que le strict nécessaire, le temps des conversions ADC. Elle ne déclenche un redessin que lorsque l'orientation change réellement, grâce au produit scalaire interprété comme un cosinus d'écart angulaire. Enfin, elle compare des distances au carré pour la zone morte afin d'épargner une racine carrée. La trigonométrie est ainsi entièrement « cachée » dans la géométrie du vecteur normalisé : c'est précisément ce qui permet au PIC16F18877, dépourvu de calcul flottant matériel, de faire pivoter le vaisseau de manière à la fois réactive et stable.

11. Le vaisseau : geometrie et rotation sans cos/sin

11.1 Description : un vaisseau immobile qui ne fait que pivoter

Le vaisseau du joueur est represente par un triangle isocele en forme de fleche. Contrairement a la version arcade originale d'Asteroids, ou le vaisseau se deplace librement, le choix de conception retenu ici fige sa position : le vaisseau reste en permanence au centre geometrique de la zone de jeu, defini par les constantes `CX = 240` et `CY = 172`. Ce point correspond a la moitie de la largeur de l'ecran (`SCREEN_W/2`, soit `480/2`) et au centre vertical de la zone situee sous la bande de score (`HUD_H + (SCREEN_H - HUD_H)/2`, avec `HUD_H = 24`). Le vaisseau ne se translate jamais : sa seule liberte est la rotation sur place, commandee par le joystick analogique.

Ce parti pris a deux merites. D'une part, il simplifie radicalement le rendu : puisque le centre du triangle ne bouge pas, seuls les trois sommets se deplacent autour d'un pivot fixe, ce qui limite la surface a effacer et a redessiner a chaque rotation. D'autre part, il colle au principe du jeu : les asteroides surgissent des bords et foncent vers le centre, le joueur doit donc orienter sa pointe vers la menace et tirer. La rotation seule suffit a viser dans toutes les directions.

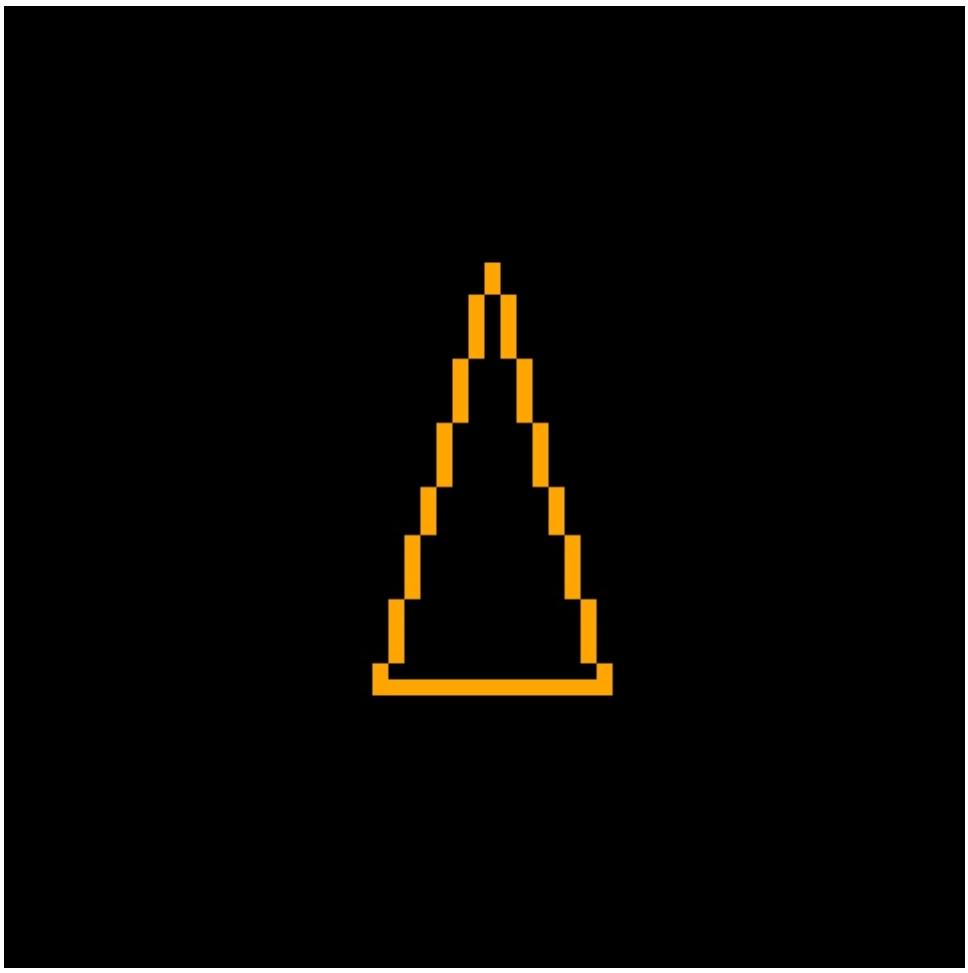


Figure 27 — Le vaisseau triangulaire affiche au centre de l'ecran, oriente par le joystick.

L'orientation du vaisseau est stockee non pas sous forme d'angle, mais sous forme d'un vecteur unitaire (shipCos, shipSin). Ce couple de valeurs flottantes represente directement le cosinus et le sinus de l'angle de visee. Ce detail, qui peut sembler anodin, est en realite la cle de toute l'optimisation decrite plus loin : on ne manipule jamais l'angle lui-meme, mais toujours son cosinus et son sinus. On evite ainsi les appels couteux a atan2, cos et sin sur un PIC16F18877 depourvu d'unite de calcul flottant materielle, ou ces fonctions sont realisees logiciellement.

11.2 La fonction calculerSommetsVaisseau()

La geometrie du triangle est entierement reconstruite par une seule fonction. La pointe avant est placee dans la direction de visee, a une distance SHIP_FRONT = 18 pixels du centre. Les deux coins arriere sont obtenus en faisant tourner la direction de visee de plus et moins 2,45 radians, puis en multipliant le resultat par SHIP_BACK = 11 pixels. Voici le code integral :

```
void calculerSommetsVaisseau(void){
    shipX1 = CX + (int)(shipCos * SHIP_FRONT);
    shipY1 = CY + (int)(shipSin * SHIP_FRONT);
    float bx = shipCos*COS245 - shipSin*SIN245;
    float by = shipSin*COS245 + shipCos*SIN245;
    shipX2 = CX + (int)(bx * SHIP_BACK);
    shipY2 = CY + (int)(by * SHIP_BACK);
    float cx = shipCos*COS245 + shipSin*SIN245;
    float cy = shipSin*COS245 - shipCos*SIN245;
    shipX3 = CX + (int)(cx * SHIP_BACK);
    shipY3 = CY + (int)(cy * SHIP_BACK);
}
```

Detaillons bloc par bloc.

Les deux premieres lignes calculent la pointe avant `(shipX1, shipY1)`. Le vecteur de direction `(shipCos, shipSin)` etant unitaire, le multiplier par `SHIP_FRONT` donne un deplacement de 18 pixels exactement dans la direction de visee. On ajoute ce deplacement au centre `(CX, CY)`. Le transtypage `(int)` convertit le resultat flottant en coordonnee entiere de pixel, format que l'ecran attend.

Le deuxieme bloc calcule le premier coin arriere. Les variables intermediaires `bx` et `by` representent le vecteur de direction tourne d'un angle de plus 2,45 radians. On reconnait la formule de rotation d'un vecteur (developpee en 11.3), ou `COS245` et `SIN245` sont le cosinus et le sinus de cet angle, precalcules une fois pour toutes. Le vecteur tourne `(bx, by)` reste unitaire ; on le multiplie donc par `SHIP_BACK = 11` pour placer le coin arriere a 11 pixels du centre, puis on l'ajoute au centre.

Le troisieme bloc est rigoureusement symetrique : `cx` et `cy` representent la direction tournee de

moins 2,45 radians. Le signe affecte au terme en sinus s'inverse (on lit `+ shipSin*SIN245` au lieu de `-`, et `- shipCos*SIN245` au lieu de `+`), ce qui correspond a une rotation dans l'autre sens. C'est cette symetrie autour de l'axe avant qui donne au triangle sa forme isocele : les deux ailes arriere sont ecartees du meme angle de part et d'autre de l'axe de visee.

On notera qu'aucune fonction trigonometrique n'est appelee : la fonction ne contient que des multiplications, des additions et des soustractions de flottants. C'est exactement l'objectif recherche.

11.3 Les mathematiques de la rotation

La formule employee est celle de la rotation d'un vecteur dans le plan. Soit un vecteur unitaire `(c, s)` correspondant a un angle de depart, que l'on souhaite tourner d'un angle `alpha`. Les coordonnees du vecteur tourne sont :

```
x' = c * cos(alpha) - s * sin(alpha)
```

```
y' = s * cos(alpha) + c * sin(alpha)
```

Ces deux expressions decoulent directement des formules d'addition des angles. En effet, si `(c, s) = (cos theta, sin theta)`, alors le vecteur tourne de `alpha` a pour angle `theta + alpha`, et :

```
cos(theta + alpha) = cos theta cos alpha - sin theta sin alpha
```

```
sin(theta + alpha) = sin theta cos alpha + cos theta sin alpha
```

On retrouve mot pour mot les lignes de code. Le point essentiel est que l'angle de rotation `alpha = 2,45 rad` est une **constante** du programme : il ne depend ni de la position du joystick, ni du temps. Par consequent, `cos(2,45)` et `sin(2,45)` peuvent etre evalues une seule fois, a la main ou par un calcul prealable, et stockes dans des constantes :

```
COS245 = -0.7705
```

```
SIN245 =  0.6374
```

La rotation se reduit alors a quatre multiplications et deux additions par coin, soit uniquement

des operations rapides. C'est la difference fondamentale avec une approche naive qui appellerait `cos(alpha)` et `sin(alpha)` a chaque image : en l'absence de FPU sur le PIC16F18877, ces fonctions, realisees logiciellement, coutent un grand nombre de cycles. En precalculant les constantes, on transforme un calcul trigonometrique en simple combinaison lineaire.

On verifie la coherence des constantes : 2,45 radians correspondent a environ 140,4 degres (puisque `2,45 x 180 / pi` vaut a peu pres 140,4). Le cosinus d'un angle de 140 degres est bien negatif, son sinus bien positif, ce qui est conforme aux valeurs `-0,7705` et `0,6374`.

11.4 La fonction `tracerVaisseau(c)`

Une fois les six coordonnees memorisees dans les variables globales `shipX1..shipX3` et `shipY1..shipY3`, le dessin est trivial. La fonction `tracerVaisseau` se contente de relier les trois sommets par trois segments, a l'aide de la primitive `display_drawLine` de la bibliotheque graphique :

```
void tracerVaisseau(uint16_t c){
    display_drawLine(shipX1, shipY1, shipX2, shipY2, c);
    display_drawLine(shipX2, shipY2, shipX3, shipY3, c);
    display_drawLine(shipX3, shipY3, shipX1, shipY1, c);
}
```

Le parametre `c` est la couleur du trace. Pour afficher le vaisseau, on appelle la fonction avec la couleur orange du jeu ; pour l'effacer, on rappelle exactement la meme fonction avec la couleur noire, ce qui repasse les memes pixels en couleur de fond. Aucun calcul flottant n'est effectue ici : la fonction ne manipule que des coordonnees entieres deja calculees. Le cout du dessin est donc reduit au trace de trois lignes par l'algorithme de Bresenham, sans la moindre operation trigonometrique.

11.5 La separation calcul / dessin

L'architecture du vaisseau illustre un principe recurrent du programme : separer le calcul de la position du dessin proprement dit. La fonction `calculerSommetsVaisseau()` n'est appelee que

lorsque l'orientation change reellement, c'est-a-dire lorsque le joueur tourne le joystick suffisamment pour franchir le seuil de redessin ANGLE_REDRAW_DOT = 0.9997. Ce seuil est un produit scalaire entre l'ancienne et la nouvelle direction : tant qu'il reste superieur a 0,9997, la rotation est inferieure a environ 1,4 degre et l'on considere que le vaisseau pointe toujours dans la meme direction. Ses sommets demeurent alors valides et il est inutile de les recalculer.

La fonction tracerVaisseau(), elle, peut etre appelee plusieurs fois sans recalcul : une premiere fois en noir pour effacer l'ancien triangle, une seconde fois en orange pour dessiner le nouveau. Ce decoupage "calculer une fois, dessiner ou effacer plusieurs fois" evite a la fois les recalculs inutiles et le scintillement, tout en menageant le microcontroleur.

Figure 28 — Geometrie du vaisseau : pointe avant a SHIP_FRONT du centre, ailes arriere ecartees de plus ou moins 2,45 rad de l'axe de visee.

Pour conclure, revenons sur la raison profonde pour laquelle un vecteur (cos, sin) normalise suffit a decrire une direction. Un vecteur unitaire est, par definition, un point du cercle de rayon 1 ; or tout point de ce cercle s'ecrit (cos theta, sin theta) pour un certain angle theta. Le couple (shipCos, shipSin) encode donc une direction sans qu'il soit jamais necessaire de connaitre theta lui-meme. Comme le joystick fournit deja, apres normalisation, un tel vecteur unitaire pointant vers la cible (voir le chapitre sur le joystick), la direction de visee est disponible "gratuitement". Pour les ailes, on ecarte cette direction de plus ou moins 2,45 rad par rapport a l'axe de visee. Chaque aile fait ainsi un angle d'environ 140 degres avec la pointe, ce qui revient a la placer a environ 40 degres de l'axe arriere : les deux coins se retrouvent largement derriere le centre, de part et d'autre de l'arriere du vaisseau, et le triangle prend son allure de fleche. Un angle proche de 180 degres ramenerait les ailes presque dans l'axe arriere et donnerait une fleche tres effilee, tandis qu'un angle proche de 90 degres produirait un triangle large et trapu. La valeur de 2,45 rad realise un compromis visuel satisfaisant, tout en restant une constante que l'on peut precalculer, fidele au fil rouge du projet : eviter le calcul flottant en temps reel pour menager un processeur lent.

12. Les projectiles (tirs)

Les projectiles constituent le moyen d'action principal offert au joueur : depuis le centre de l'ecran, le vaisseau triangulaire tire des disques orange qui filent en ligne droite vers les bords pour detruire les asteroides (chaque asteroide detruit rapporte +10 au score). Ce chapitre detaille la maniere dont les tirs sont representes en memoire, comment ils naissent (fonction `tirer`), comment ils evoluent a chaque tour de boucle (fonction `majTirs`) et comment la cadence de tir est limitee. Comme partout dans ce projet, la conception repond a une contrainte permanente : le PIC16F18877 est un microcontroleur 8 bits dont le cycle instruction vaut FOSC/4 = 8 MHz (soit un temps de cycle Tcy de 125 ns), et qui est depourvu d'unite de calcul flottant materielle. Chaque appel a `cos`, `sin`, `atan2` ou `sqrtf` se traduit donc par une routine logicielle lente. Chaque choix decrit ci-dessous vise a economiser le temps de calcul et la memoire vive.

12.1 Representation en memoire : un tableau fixe de quatre tirs

Un tir est decrit par la structure suivante :

```
typedef struct { float x, y, vx, vy; uint8_t active, life; } Bullet;
```

Les champs `x` et `y` portent la position courante du projectile en pixels sur l'ecran, et `vx`, `vy` sa vitesse, c'est-a-dire le deplacement applique a chaque tour de boucle. Le champ `active` indique si la case est occupee par un tir vivant (1) ou si elle est libre (0). Le champ `life` est un compteur de duree de vie : il decroit a chaque tour et provoque la disparition du tir lorsqu'il atteint zero, ce qui evite que des projectiles n'errent indefiniment a l'ecran. Les deux derniers champs sont declares en `uint8_t` (entier non signe sur 8 bits, plage 0 a 255) : cela suffit largement pour un drapeau booleen et pour une duree de vie de 60, tout en occupant le minimum de RAM possible. La position et la vitesse, en revanche, restent en `float` parce que le deplacement de 9 pixels par tour combine a une direction quelconque produit des increments fractionnaires (par exemple 9 x cos(a)) : les arrondir a chaque tour deformerait la trajectoire. On accepte donc le cout des additions flottantes pour la position, mais on bannit les fonctions trigonometriques flottantes, bien plus couteuses.

Les tirs sont stockes dans un tableau global de taille fixe :

```
Bullet bullets[MAX_BULLETS];   // MAX_BULLETS = 4
```

Le choix d'un **tableau de taille fixe** plutot que d'une allocation dynamique (malloc/free) est un parti pris essentiel sur cible embarquee. L'allocation dynamique est lente, peut echouer si la memoire est fragmentee, et expose a des fuites de memoire si une liberation est oubliee. Sur un microcontroleur disposant de quelques kilo-octets de RAM seulement, ces risques sont inacceptables. La solution retenue consiste a reserver une fois pour toutes quatre emplacements et a les recycler en jouant simplement sur le drapeau active : un tir « disparait » non pas en liberant de la memoire, mais en repassant active a 0, ce qui rend la case immediatement reutilisable pour un futur projectile. Le plafond MAX_BULLETS = 4 borne aussi le nombre de projectiles a dessiner par tour, donc le temps de calcul de la boucle de jeu reste previsible et constant : on sait a l'avance qu'on ne tracera jamais plus de quatre disques de tir par image.

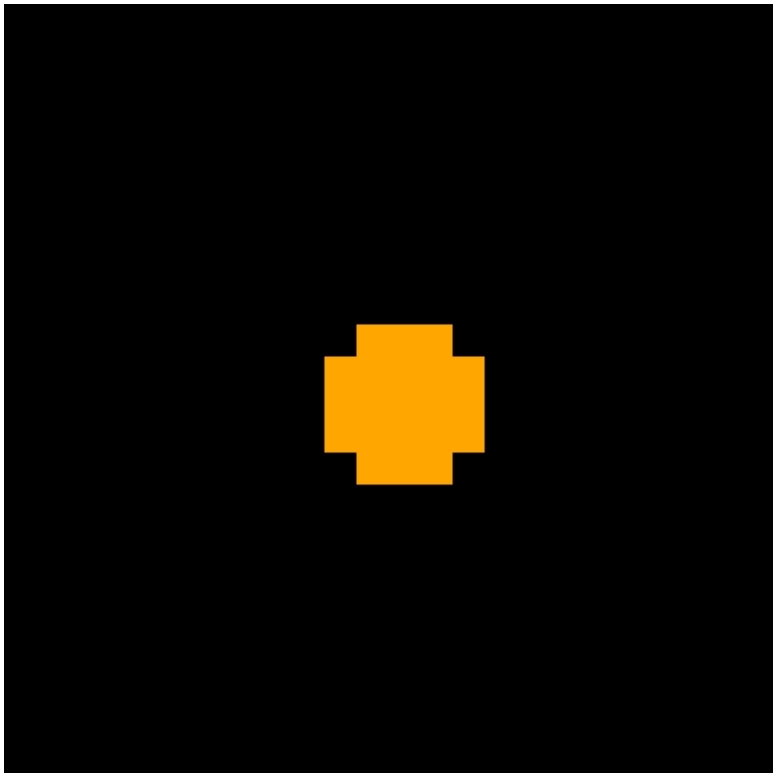


Figure 29 — projectile orange tel qu'il apparait a l'ecran, un petit disque de rayon BULLET_RADIUS.

12.2 La fonction tirer() : creation d'un projectile

Lorsque le joueur appuie sur le bouton de tir (BTN_B, sur RD6, patte 29), la fonction tirer est

appelee. Son role est de trouver un emplacement libre dans le tableau et d'y inscrire un nouveau projectile, oriente comme le vaisseau.

```
void tirer(void){
    for(uint8_t i=0;i<MAX_BULLETS;i++){
        if(!bullets[i].active){
            bullets[i].x = CX + shipCos*SHIP_FRONT;
            bullets[i].y = CY + shipSin*SHIP_FRONT;
            bullets[i].vx = shipCos*BULLET_SPEED;
            bullets[i].vy = shipSin*BULLET_SPEED;
            bullets[i].life = BULLET_LIFE;
            bullets[i].active = 1;
            display_fillCircle((uint16_t)bullets[i].x,(uint16_t)bullets[i].y,BULLET_RADIUS,COL_ORANGE);
            son_tir();
            break;
        }
    }
}
```

La boucle `for` parcourt les quatre cases. Le test `if(!bullets[i].active)` selectionne la **premiere case libre** rencontree (drapeau a 0). Si les quatre tirs sont deja actifs, la boucle se termine sans rien faire : il n'est tout simplement pas possible de tirer une cinquieme fois tant qu'un projectile n'a pas disparu. C'est une limitation voulue, coherente avec la taille fixe du tableau.

Une fois une case libre trouvee, le projectile est positionne **a la pointe avant du vaisseau** :

- `bullets[i].x = CX + shipCos*SHIP_FRONT;`
- `bullets[i].y = CY + shipSin*SHIP_FRONT;`

Ici `CX` et `CY` sont les coordonnees du centre de l'ecran (CX = 240, CY = 172), point fixe ou se trouve toujours le vaisseau. Le couple `(shipCos, shipSin)` est le **vecteur unitaire d'orientation** du vaisseau, deja calcule a partir du joystick : `shipCos` est le cosinus de l'angle de visee, `shipSin` son sinus. Multiplier ce vecteur unitaire par la distance `SHIP_FRONT = 18` (la longueur du nez du vaisseau) et l'ajouter au centre donne exactement la position de la pointe. Mathematiquement, on calcule le point situe a la distance SHIP_FRONT du centre dans la direction de visee :

pointe = (CX + cos(a) x SHIP_FRONT, CY + sin(a) x SHIP_FRONT)

Le tir part donc visuellement du canon du vaisseau, et non de son centre.

La vitesse reprend la **meme direction**, mise a l'echelle par la cadence souhaitee :

- `bullets[i].vx = shipCos*BULLET_SPEED;`
- `bullets[i].vy = shipSin*BULLET_SPEED;`

avec `BULLET_SPEED = 9.0`. Comme `(shipCos, shipSin)` est un vecteur de norme 1 (cos(a)^2 + sin(a)^2 = 1), multiplier par 9 donne un vecteur vitesse de norme 9 pixels par tour, dirige exactement comme le vaisseau. C'est l'avantage decisif de reutiliser l'orientation deja connue : aucun appel a `cos`, `sin` ou `atan2` n'est necessaire au moment du tir, alors que ces fonctions flottantes seraient tres couteuses sur le PIC. On recycle un resultat deja disponible, calcule une seule fois par tour lors de la lecture du joystick.

Le champ `life` est initialise a `BULLET_LIFE = 60` : le projectile vivra au plus 60 tours de boucle. Le drapeau `active` passe a 1 pour marquer la case occupee. La ligne `display_fillCircle(...)` dessine immediatement le disque orange (`COL_ORANGE = 0xFD20`) de rayon `BULLET_RADIUS = 2` a la position de depart. La conversion explicite `(uint16_t)bullets[i].x` tronque les coordonnees flottantes vers l'entier inferieur, car la primitive graphique (qui s'appuie sur `display_fillCircle`, lui-meme bati sur l'algorithme du point milieu de la bibliotheque GFX) attend des coordonnees de pixels entieres. Enfin, `son_tir()` declenche l'effet sonore via le peripherique NCO1 (Numerically Controlled Oscillator), qui genere une frequence sur la patte RA6 sans solliciter le processeur pendant la lecture du son, et le `break` quitte la boucle : **un seul tir est cree par appel**, meme si plusieurs cases etaient libres. Sans ce `break`, un seul appui remplirait d'un coup les quatre emplacements disponibles.

12.3 La fonction majTirs() : effacer, deplacer, redessiner

A chaque tour de boucle de jeu, la fonction `majTirs` (mise a jour des tirs) fait evoluer tous les projectiles actifs. Elle applique la technique fondamentale du rendu sur cet ecran : **effacer, deplacer, redessiner**. Comme on ne dispose pas de double tampon (framebuffer) en RAM et que reafficher tout l'ecran de 480 x 320 pixels via le bus SPI serait beaucoup trop lent, on ne touche qu'aux quelques pixels qui changent reellement.

Pour chaque tir actif, la sequence est la suivante :

1. **Effacer** : on redessine un disque de la couleur du fond (noir) a la position actuelle, ce qui « gomme » le projectile la ou il etait.
2. **Deplacer** : on met a jour la position par `x += vx` et `y += vy`. C'est une cinematique de mouvement rectiligne uniforme : a chaque tour, la position s'incremente du vecteur vitesse. Sur un tour, le projectile avance donc de 9 pixels dans sa direction (norme de la vitesse). Aucune acceleration n'intervient, la trajectoire est une droite parcourue a vitesse constante exprimee en pixels par tour de boucle.
3. **Decrementer la duree de vie** : `life` diminue de 1. Le tir est desactive (`active = 0`) si `life` atteint 0 ou si la nouvelle position sort de l'ecran (coordonnees hors des bornes 0..SCREEN_W et 0..SCREEN_H, soit 0..480 en x et 0..320 en y). La case redevient alors libre pour un futur tir.
4. **Redessiner** : si le tir est toujours vivant, on retrace le disque orange a sa nouvelle position.

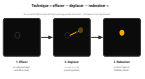


Figure 30 — principe effacer puis deplacer puis redessiner applique a chaque element mobile.

Ce cycle explique pourquoi chaque element mobile (tirs, asteroides, vaisseau) memorise sa position courante dans sa structure : il faut savoir ou effacer avant de redessiner ailleurs. La double condition de desactivation (sortie d'ecran ou duree de vie ecoulee) garantit qu'aucun projectile ne reste actif inutilement. On peut d'ailleurs verifier la coherence des constantes : un tir vit au plus 60 tours et avance de 9 pixels par tour, soit une portee maximale theorique de 60 x 9 = 540 pixels. Comme la plus grande dimension de l'ecran est 480 pixels, un tir parti du centre atteint toujours un bord avant l'epuisement de sa duree de vie ; c'est donc le plus

souvent la sortie d'ecran qui le desactive en premier, le compteur life ne servant que de garde-fou. Dans tous les cas, la case est liberee rapidement, ce qui borne la charge de calcul de la boucle.

12.4 L'anti-rafale : le compteur shootCooldown

Si la fonction tirer etait appelee a chaque tour ou le bouton est maintenu enfonce, le joueur viderait instantanement les quatre cases et la cadence dependrait de la vitesse de la boucle, ce qui serait peu maitrisable. Pour eviter cette rafale incontrolable, la boucle de jeu (boucleJeu) gere un compteur de temporisation, shootCooldown. Le principe est le suivant : un tir n'est autorise que lorsque shootCooldown est revenu a zero. Au moment ou un projectile est cree, le compteur est recharge a une valeur de delai, puis il decroit d'une unite a chaque tour de boucle. Tant qu'il n'est pas nul, les nouveaux appuis (ou le maintien du bouton, rappelons que BTN_B est actif a l'etat bas grace a une resistance de tirage) sont ignores. On obtient ainsi une cadence de tir reguliere et independante des micro-variations de duree de la boucle, et l'on protege en meme temps le nombre limite d'emplacements disponibles. Cette temporisation par compteur logiciel est preferee a toute attente bloquante : la boucle de jeu ne doit jamais s'arreter pour patienter, sous peine de figer le deplacement des asteroides et la reactivite du joystick.

12.5 Synthese

La gestion des tirs illustre parfaitement le fil conducteur du projet. Le tableau fixe de quatre cases bannit l'allocation dynamique au profit d'un recyclage par drapeau active, sur et previsible. La direction du projectile reutilise le vecteur unitaire (shipCos, shipSin) deja calcule pour le vaisseau, evitant tout appel trigonometrique flottant au moment du tir. Le deplacement se reduit a deux additions par tour, mouvement rectiligne uniforme a vitesse constante de 9 pixels par tour. Le rendu n'efface et ne retrace que les pixels concernes, et le compteur shootCooldown discipline la cadence sans jamais bloquer la boucle. Chaque decision menage un microcontroleur lent tout en offrant un comportement de jeu fluide et coherent.

13. Les asteroides (meteorites)

Les asteroides constituent la menace centrale du jeu : ce sont des octogones qui surgissent des bords de l'ecran et foncent vers le vaisseau immobilise au centre. Chaque asteroide detruit rapporte 10 points au score, tandis qu'un contact avec le vaisseau provoque la fin de partie (GAME OVER). Ce chapitre detaille leur representation en memoire, leur apparition aleatoire, leur trace graphique et leur mise a jour image par image. Comme partout dans le projet, l'objectif transversal est de menager le PIC16F18877, un microcontroleur 8 bits depourvu d'unite de calcul flottant materielle : on cherche donc a limiter le nombre d'operations en virgule flottante et a ne redessiner que ce qui change.

13.1 Representation en memoire

Un asteroide est decrit par une structure simple, declaree dans `main.c` :

```
typedef struct {float x,y,vx,vy; uint8_t active;} Asteroid;
```

Les champs `x` et `y` representent la position du centre de l'asteroide a l'ecran (en pixels, mais stockee en `float` car le calcul de trajectoire produit des valeurs non entieres). Les champs `vx` et `vy` sont les deux composantes du vecteur vitesse : a chaque tour de boucle, on ajoute `vx` a `x` et `vy` a `y` pour deplacer l'asteroide. Enfin, `active` est un drapeau (`uint8_t`, soit un seul octet) qui vaut 1 si l'asteroide existe a l'ecran et 0 si la case est libre. Le choix de `uint8_t` plutot qu'un `bool` ou un `int` est deliberement econome : sur ce PIC, la RAM est limitee, et un octet suffit largement pour un indicateur logique. La structure comporte donc cinq champs : quatre flottants (position et vitesse) et un octet de statut.

Les asteroides sont ranges dans un tableau global de taille fixe :

```
Asteroid asteroids[MAX_ASTEROIDS];
```

avec `MAX_ASTEROIDS=4`. Ce dimensionnement fixe est un principe recurrent du systeme embarque : on n'utilise pas d'allocation dynamique (`malloc`), couteuse et risquee sur un petit microcontroleur. Le tableau est reserve une fois pour toutes au demarrage, et le drapeau `active` sert de gestion d'occupation : une case « libre » est simplement une case dont `active` vaut 0.

Limiter le nombre d'asteroides simultanes a quatre borne aussi la charge de calcul de la boucle de jeu, ce qui aide a conserver une cadence reguliere. Le meme procede (tableau fixe + drapeau `active`) est utilise pour les projectiles (`bullets[MAX_BULLETS]`, avec `MAX_BULLETS=4`), ce qui confere au code une logique homogene et previsible.

Visuellement, chaque asteroide est un octogone (polygone a huit cotes), choisi comme compromis entre un rendu « rocheux » credible et un cout de trace faible (huit segments seulement).

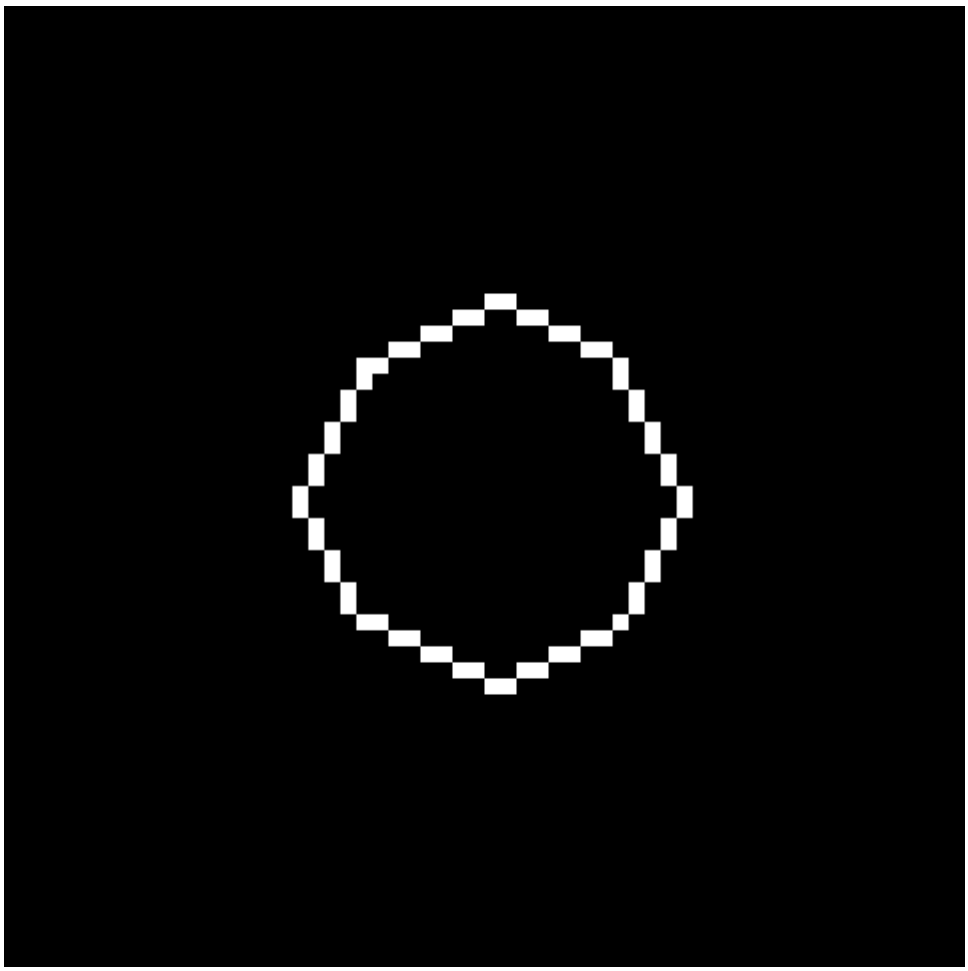


Figure 31 — rendu d'un asteroide a l'ecran : un octogone trace au trait.

13.2 Apparition : `spawnAsteroide()`

La fonction `spawnAsteroide()` fait naitre un nouvel asteroide sur un bord de l'ecran, avec une trajectoire dirigee vers le centre. Voici le code complet :

```
void spawnAsteroide(void){
    uint8_t i;
    for(i=0;i<MAX_ASTEROIDS;i++){ if(!asteroids[i].active) break; }
    if(i==MAX_ASTEROIDS) return;
    uint8_t bord = rand()%4;
    if(bord==0){ asteroids[i].x = rand()%SCREEN_W; asteroids[i].y = HUD_H+1; }
    else if(bord==1){ asteroids[i].x = rand()%SCREEN_W; asteroids[i].y = SCREEN_H-1; }
    else if(bord==2){ asteroids[i].x = 0; asteroids[i].y = HUD_H + rand()%(SCREEN_H-HUD_H); }
    else { asteroids[i].x = SCREEN_W-1; asteroids[i].y = HUD_H + rand()%(SCREEN_H-HUD_H); }
    float dx = CX - asteroids[i].x;
    float dy = CY - asteroids[i].y;
    float d = sqrtf(dx*dx+dy*dy);
    if(d<1.0f) d=1.0f;
    asteroids[i].vx = (dx/d)*AST_SPEED;
    asteroids[i].vy = (dy/d)*AST_SPEED;
    asteroids[i].active = 1;
}
```

Recherche d'une case libre. La boucle `for` parcourt le tableau et s'arrete (`break`) sur la premiere case dont `active` vaut 0 (l'operateur `!` inverse la valeur logique). Si aucune case n'est libre, l'indice `i` atteint `MAX_ASTEROIDS` apres la boucle : la condition `if(i==MAX_ASTEROIDS) return;` quitte alors la fonction sans rien creer. C'est ce mecanisme qui garantit qu'on ne depasse jamais quatre asteroides et qu'on n'ecrit jamais hors du tableau (pas de debordement memoire).

Choix du bord. `rand()%4` renvoie un entier dans {0,1,2,3}, ce qui designe respectivement le

bord haut, bas, gauche ou droite. La fonction `rand()`, issue de `<stdlib.h>`, fournit le hasard necessaire pour que les vagues d'asteroides ne soient pas previsibles. Selon le bord tire :

- bord 0 (haut) : abscisse aleatoire sur toute la largeur (`rand()%SCREEN_W`, soit 0 a 479), ordonnee juste sous le HUD (`HUD_H+1`) pour ne pas empieter sur la bande de score ;
- bord 1 (bas) : abscisse aleatoire, ordonnee `SCREEN_H-1` (derniere ligne, 319) ;
- bord 2 (gauche) : abscisse `0`, ordonnee aleatoire dans la zone de jeu `HUD_H + rand()%(SCREEN_H-HUD_H)` ;
- bord 3 (droite) : abscisse `SCREEN_W-1` (479), ordonnee aleatoire dans la meme zone.

On remarque que les positions verticales sont systematiquement comprises entre `HUD_H` (24) et `SCREEN_H` (320), ce qui interdit aux asteroides d'apparaitre dans la bande de score `HUD_H=24` pixels situee tout en haut de l'ecran. Cette precaution evite que des asteroides ne viennent recouvrir le compteur de points.

Calcul de la vitesse vers le centre. Une fois la position de depart fixee, on veut une vitesse qui pointe vers le centre du jeu `(CX, CY)`, soit `(240, 172)`. On forme d'abord le vecteur deplacement vers le centre :

```
dx = CX - x
```

```
dy = CY - y
```

Ce vecteur indique la bonne direction, mais sa longueur depend de la distance au centre : un asteroide ne du coin irait beaucoup plus vite qu'un autre apparu pres du centre. Pour que tous se deplacent a la meme vitesse `AST_SPEED` (ici 4,0), on **normalise** le vecteur : on le divise par sa propre longueur `d` pour obtenir un vecteur unitaire (de longueur 1), puis on le multiplie par la vitesse voulue. La longueur se calcule par le theoreme de Pythagore :

```
d = sqrt(dx^2 + dy^2)
```

```
vx = (dx/d) * AST_SPEED
```

```
vy = (dy/d) * AST_SPEED
```

La ligne `if(d<1.0f) d=1.0f;` est une garde de securite : si l'asteroide naissait quasiment sur le

centre, `d` serait proche de zero et la division `dx/d` produirait une valeur enorme, voire une division par zero. En forcant `d` a valoir au moins 1, on evite ce cas degenere et on garantit que `vx` et `vy` restent bornes.

C'est le seul appel a la racine carree `sqrtf` (issue de `<math.h>`) du cycle de vie d'un asteroide. Or `spawnAsteroide()` n'est appelee que rarement (au plus une fois tous les 19 tours de boucle, cadence reglee par `SPAWN_DELAY=19`), et au plus quatre asteroides coexistent. Le cout de ce flottant lent est donc parfaitement acceptable, parce qu'il est paye une seule fois a l'apparition et jamais pendant le deplacement. Cette repartition « calcul lourd a l'apparition, calcul leger en jeu » illustre la strategie generale du projet face a un processeur sans virgule flottante materielle : les seuls appels couteux (racine carree, division flottante) sont confines aux evenements rares, jamais a la boucle de rendu de chaque trame.

13.3 Trace de l'octogone : `dessinerAsteroide()`

Le dessin s'appuie sur deux tableaux de constantes precalculees representant un octogone de rayon 10, centre sur l'origine :

```
astX[8] = {10, 7, 0, -7, -10, -7,  0,  7}
astY[8] = { 0, 7,10,  7,   0, -7,-10, -7}
```

Ces huit couples decrivent les sommets d'un octogone parcouru dans le sens trigonometrique : chaque sommet est a une distance d'environ 10 de l'origine (par exemple sqrt(7^2 + 7^2) = sqrt(98) ~ 9,9, proche de 10). Le fait que ces valeurs soient figees une fois pour toutes evite tout appel a `cos` ou `sin` au moment du dessin. La fonction de trace transforme ces huit sommets puis les relie :

```
void dessinerAsteroide(Asteroid *a, uint16_t c){
    int px[8]; int py[8];
    for(uint8_t k=0;k<8;k++){
        px[k] = (int)a->x + (AST_RADIUS*astX[k])/10;
        py[k] = (int)a->y + (AST_RADIUS*astY[k])/10;
```

```
    }
    for(uint8_t k=0;k<8;k++){
        uint8_t j=(k+1)&7;
        display_drawLine(px[k],py[k],px[j],py[j],c);
    }
}
```

Le parametre `Asteroid *a`. La fonction recoit un *pointeur* sur un asteroide, et non une copie de la structure. La notation `a->x` est un raccourci pour `(*a).x` : la fleche signifie « le champ `x` de la structure pointee par `a` ». Passer un pointeur evite de recopier les cinq champs de la structure a chaque appel ; on transmet simplement une adresse, ce qui est plus econome en pile et en temps. Le second parametre `c` est la couleur du trait, codee sur 16 bits (`uint16_t`, format RGB565) ; le pilote bas niveau de l'ecran la convertira ensuite en trois octets 18 bits avant l'envoi sur le bus SPI.

Mise a l'echelle et translation. La premiere boucle calcule les huit sommets a l'ecran. Le facteur `(AST_RADIUS*astX[k])/10` met l'octogone a l'echelle : comme les constantes ont ete definies pour un rayon 10, multiplier par `AST_RADIUS` (12) puis diviser par 10 redimensionne la forme au rayon reel souhaite. Le calcul reste **entier** (les `astX`/`astY` sont des entiers et `AST_RADIUS` aussi), donc rapide : aucune virgule flottante n'intervient ici. On ajoute ensuite la position du centre `(int)a->x` et `(int)a->y` : la conversion `(int)` tronque la partie decimale de la position flottante pour obtenir des coordonnees de pixels entieres.

Trace des cotes et l'astuce `& 7`. La seconde boucle relie chaque sommet `k` au suivant `j`. Le calcul `j = (k+1)&7` est le point cle : `&7` est un ET binaire avec le masque 7 (`0b111`), qui equivaut a un modulo 8. Lorsque `k` vaut 7, `k+1` vaut 8, et `8 & 7` donne 0 : le dernier sommet se relie donc automatiquement au premier, fermant l'octogone. On evite ainsi un `if` particulier pour le dernier segment. Sur un PIC, un ET binaire est nettement moins couteux qu'une operation de modulo `%` (qui implique une division), d'ou ce choix. Chaque cote est trace par `display_drawLine`, qui s'appuie sur l'algorithme de Bresenham fourni par la bibliotheque graphique.

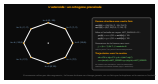


Figure 32 — construction de l'octogone a partir des huit sommets precalcules et fermeture par l'index (k+1) & 7.

13.4 Mise a jour : majAsteroides()

A chaque tour de boucle de jeu, la fonction majAsteroides() applique le schema general du rendu adopte dans tout le projet : effacer / deplacer / desactiver si necessaire / redessiner. Concretement, pour chaque asteroide actif :

1. **Effacer** l'octogone a sa position courante, en le redessinant avec la couleur du fond (typiquement noir). On ne rafraichit ainsi qu'une petite zone, sans effacer tout l'ecran, ce qui serait beaucoup trop lent sur la liaison SPI (chaque pixel coute trois octets de couleur a transmettre).
2. **Deplacer** l'asteroide : a->x += a->vx; et a->y += a->vy;. Comme vx et vy ont ete calcules une fois pour toutes a l'apparition, le deplacement ne coute que deux additions flottantes, sans aucune racine carree ni trigonometrie.
3. **Desactiver si trop loin** : si l'asteroide a depasse le centre et continue hors de la zone de jeu, on remet active = 0 pour liberer la case. Celle-ci redevient alors disponible pour un futur spawnAsteroide().
4. **Redessiner** l'octogone a sa nouvelle position avec sa couleur normale (les teintes orangees COL_ORANGE (0xFD20) / COL_ORANGE_DARK (0xCB20) du jeu).

Ce cycle effacer/deplacer/redessiner n'agit que sur les pixels reellement concernes : il evite de reconstruire toute l'image a chaque trame, ce qui economise un nombre considerable d'octets envoyes sur le bus SPI vers l'ecran ILI9488. C'est dans cette meme fonction, ou juste apres, que sont egalement testees les collisions (asteroide contre projectile pour marquer +10, asteroide contre vaisseau pour declencher le GAME OVER), comparaisons effectuees sur les distances au carre afin d'eviter de nouvelles racines carrees.

Retour sur la normalisation vectorielle. Le point mathematique central de la trajectoire d'asteroide est la normalisation d'un vecteur. Un vecteur (dx, dy) peut s'ecrire comme une direction multipliee par une longueur. La direction seule s'obtient en divisant par la norme d = sqrt(dx^2 + dy^2) : le vecteur (dx/d, dy/d) a alors une longueur exactement egale a 1, c'est le *vecteur unitaire*. En le multipliant par AST_SPEED, on impose la vitesse de deplacement

souhaitee tout en conservant la direction vers le centre. Sans cette normalisation, la vitesse dependrait de la distance de naissance et les asteroides lointains traverseraient l'ecran plus vite que les proches, ce qui serait visuellement incoherent et rendrait le jeu injouable. La meme idee de vecteur unitaire se retrouve ailleurs dans le programme (orientation du vaisseau via `shipCos`/`shipSin`, qui forment eux aussi un vecteur de longueur 1), preuve de la coherence de l'approche geometrique retenue tout au long du developpement.

14. La detection des collisions

La detection des collisions constitue le coeur de l'interaction physique du jeu : c'est elle qui decide si un projectile detruit un asteroide (et fait monter le score) et si un asteroide percute le vaisseau (et provoque le GAME OVER). Sur un microcontroleur PIC16F18877 cadence a 32 MHz (FOSC = 32 MHz, soit un cycle instruction Tcy = 125 ns) et depourvu d'unite de calcul flottant materielle, le defi n'est pas seulement de detecter correctement les contacts, mais de le faire sans operation couteuse. La strategie retenue applique fidelement le fil rouge du projet : modeliser chaque objet par un cercle et comparer des distances au carre pour eviter toute racine carree. Ce chapitre detaille le test elementaire `collision()`, la fonction d'orchestration `collisions()`, puis la justification mathematique du modele en cercles.

14.1 Le test de base : la fonction collision()

Le principe geometrique est simple. Deux disques se touchent (ou se chevauchent) lorsque la distance qui separe leurs centres est inferieure ou egale a la somme de leurs rayons. Si l'on note les deux centres (x1, y1) et (x2, y2), de rayons r1 et r2, la condition de contact s'ecrit :

distance <= r1 + r2

avec distance = sqrt((x1 - x2)^2 + (y1 - y2)^2).

Le calcul direct de cette condition demanderait une racine carree (`sqrtf`), operation tres lente sur un PIC8 bits sans FPU, et appelee potentiellement plusieurs fois par tour de boucle. Chaque appel a `sqrtf` se traduit en effet par une routine logicielle de la bibliotheque mathematique de XC8, soit des dizaines voire des centaines de cycles, la ou une multiplication entiere ou flottante simple reste bien plus economique. L'optimisation cle consiste donc a elever les deux membres de l'inegalite au carre. On compare alors des distances au carre, sans jamais extraire de racine. La condition devient :

(x1 - x2)^2 + (y1 - y2)^2 <= (r1 + r2)^2

Cette transformation est exacte : elle ne change ni le resultat logique, ni la precision, elle supprime simplement l'operation couteuse. La fonction C s'ecrit ainsi :

```
bool collision(float x1,float y1,float r1,float x2,float y2,float r2){
    float dx=x1-x2;
    float dy=y1-y2;
    float r=r1+r2;
    return (dx*dx+dy*dy) <= (r*r);
}
```

Detaillons ligne par ligne ce que fait ce bloc et pourquoi chaque choix a ete fait.

- `bool collision(float x1,float y1,float r1,float x2,float y2,float r2)` declare une fonction qui renvoie un booleen (le contact a lieu ou non) et recoit les centres et rayons des deux objets a tester. Le passage par valeur convient ici : les six parametres sont de petits scalaires, et la fonction ne modifie rien a l'exterieur.
- `float dx=x1-x2;` calcule l'ecart horizontal entre les deux centres. On stocke ce resultat dans une variable locale plutot que de le recalculer, ce qui evite de refaire deux soustractions identiques plus loin.
- `float dy=y1-y2;` calcule de meme l'ecart vertical. A ce stade, le vecteur (dx, dy) joint le centre 2 au centre 1.
- `float r=r1+r2;` precalcule la somme des rayons. C'est la distance de contact recherchee. La mettre dans une variable evite de l'additionner deux fois et rend la ligne de retour plus lisible.
- `return (dx*dx+dy*dy) <= (r*r);` evalue d'un cote le carre de la distance reelle (dx*dx* + *dy*dy) et de l'autre le carre de la distance de contact (r*r), puis renvoie le booleen `true` si le premier est inferieur ou egal au second. Aucune racine carree n'apparait : c'est tout l'interet de comparer les carres.

Les coordonnees sont des `float` car les positions des projectiles et des asteroides evoluent par petits increments de vitesse (BULLET_SPEED = 9.0, AST_SPEED = 4.0) et seraient mal representees par des entiers. En revanche, les seules operations effectuees ici sont trois soustractions/additions, trois multiplications et une comparaison, toutes nettement moins penalisantes qu'une racine carree. La fonction reste donc parfaitement utilisable a chaque

iteration de la boucle de jeu.

14.2 La fonction collisions() : deux blocs de tests

La fonction `collision()` ne teste qu'une seule paire d'objets. C'est la fonction `collisions()` qui balaie toutes les paires utiles du jeu et declenche les consequences. Elle se compose de deux blocs distincts : d'abord les tirs contre les asteroides, ensuite le vaisseau contre les asteroides.

```
void collisions(void){
    for(uint8_t b=0;b<MAX_BULLETS;b++){
        if(!bullets[b].active) continue;
        for(uint8_t a=0;a<MAX_ASTEROIDS;a++){
            if(!asteroids[a].active) continue;
            if(collision(bullets[b].x,bullets[b].y,BULLET_RADIUS,
                         asteroids[a].x,asteroids[a].y,AST_RADIUS)){
                display_fillCircle((uint16_t)bullets[b].x,(uint16_t)bullets[b].y,BULLET_RADIUS,ILI9488_BLACK);
                dessinerAsteroide(&asteroids[a],ILI9488_BLACK);
                bullets[b].active=0;
                asteroids[a].active=0;
                score+=10;
                break;
            }
        }
    }
    for(uint8_t a=0;a<MAX_ASTEROIDS;a++){
        if(!asteroids[a].active) continue;
```

```
        if(collision(CX,CY,SHIP_RADIUS,
                asteroids[a].x,asteroids[a].y,AST_RADIUS)){
            son_gameover();
            afficherGameOver();
            etat=ETAT_GAMEOVER;
            return;
        }
    }
}
```

Bloc 1 : tir contre asteroide. Une double boucle imbriquee teste chaque tir contre chaque asteroide. La boucle externe parcourt les MAX_BULLETS (4) projectiles possibles, la boucle interne les MAX_ASTEROIDS (4) asteroides possibles ; les compteurs `b` et `a` sont declares en `uint8_t`, le type le plus econome en RAM pour des valeurs aussi petites. Les deux lignes `if(!bullets[b].active) continue;` et `if(!asteroids[a].active) continue;` court-circuitent immediatement le test lorsque l'objet n'existe pas a l'ecran : il est inutile de calculer une collision pour un projectile eteint ou un asteroide deja detruit. Cette garde economise des calculs et constitue une bonne hygiene de boucle. Dans le pire des cas, le bloc effectue 4 x 4 = 16 appels a `collision()`, ce qui reste tres faible.

Lorsque `collision()` renvoie `true` pour le couple (tir, asteroide), quatre actions s'enchainent. D'abord `display_fillCircle(...)` redessine le projectile en noir (ILI9488_BLACK) pour l'effacer de l'ecran a sa derniere position connue ; la conversion `(uint16_t)` ramene les coordonnees flottantes a des entiers de pixels, car l'ecran n'adresse que des positions entieres. Ensuite `dessinerAsteroide(&asteroids[a],ILI9488_BLACK)` efface de meme l'octogone de l'asteroide en le redessinant en noir : on suit ici le principe effacer / deplacer / redessiner propre au moteur de rendu, qui ne repeint que ce qui change plutot que tout l'ecran. Puis `bullets[b].active=0;` et `asteroids[a].active=0;` desactivent logiquement les deux objets, qui ne seront plus ni affiches ni testes. Enfin `score+=10;` ajoute les 10 points associes a la destruction d'un asteroide.

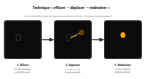


Figure 33 — principe effacer / deplacer / redessiner applique lors d'une destruction.

Le `break;` final est important : une fois qu'un projectile a touche un asteroide, il est consomme. Inutile de continuer a le comparer aux autres asteroides puisqu'il vient d'etre desactive ; on quitte donc la boucle interne pour passer au projectile suivant. Cela evite qu'un seul tir ne detruise plusieurs asteroides dans la meme passe et economise des iterations.

Bloc 2 : vaisseau contre asteroide. Une boucle simple teste cette fois le vaisseau, modelise par un cercle fixe centre en (CX, CY) = (240, 172) et de rayon SHIP_RADIUS = 18, contre chaque asteroide actif. Le vaisseau restant au centre de l'ecran, ses coordonnees sont des constantes, ce qui simplifie encore le calcul puisque (CX, CY) n'a jamais besoin d'etre recalcule. Des qu'un asteroide entre en contact avec ce cercle, le jeu bascule : `son_gameover()` joue le signal sonore de fin via le NCO1 (sortie sur RA6, patte 14), `afficherGameOver()` affiche l'ecran correspondant, et `etat=ETAT_GAMEOVER;` fait transiter la machine a etats vers l'ecran de fin de partie. Le `return;` interrompt immediatement la fonction : la partie est terminee, il est superflu de tester les asteroides restants.

L'ordre des deux blocs n'est pas anodin. On traite d'abord les destructions par tir, puis seulement la collision fatale avec le vaisseau. Ainsi, un asteroide detruit par un projectile dans le premier bloc est deja desactive (`asteroids[a].active=0`) et ne pourra pas, dans le second bloc, declencher a tort un GAME OVER : la garde `if(!asteroids[a].active) continue;` l'ecarte.

14.3 Pourquoi un modele en cercles et la demonstration des carres

Le vaisseau est en realite un triangle et l'asteroide un octogone (huit sommets precalcules dans les tableaux astX et astY, de rayon 10, mis a l'echelle x AST_RADIUS/10). On aurait pu calculer des intersections exactes entre ces polygones, mais ce serait inutilement complexe et couteux. Le choix d'un cercle englobant (boundary circle) est ici parfaitement justifie : il est simple a coder, tres rapide a evaluer, et suffisamment fidele pour un jeu d'arcade ou les objets sont petits et les contacts francs. Un cercle de rayon bien choisi (SHIP_RADIUS = 18, AST_RADIUS = 12) approxime convenablement l'encombrement reel de chaque forme, et l'imprecision residuelle reste imperceptible en pratique. Ce compromis entre exactitude geometrique et cout de calcul est exactement ce qu'impose un microcontroleur lent.

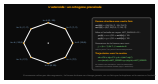


Figure 34 — forme reelle de l'asteroide (octogone) et son cercle englobant de rayon AST_RADIUS.

Reste a justifier rigoureusement que comparer les carres equivaut a comparer les distances. Soit d la distance entre les centres et r la somme des rayons. Par construction, d est une distance donc d >= 0, et r est une somme de rayons donc r >= 0. On veut montrer que :

d <= r si et seulement si d^2 <= r^2

La fonction qui a un nombre positif associe son carre est strictement croissante sur l'ensemble des reels positifs ou nuls : si l'on augmente une grandeur positive, son carre augmente aussi. Plus formellement, on peut ecrire la difference des carres comme un produit :

r^2 - d^2 = (r - d)(r + d)

Le facteur (r + d) est positif ou nul puisque d et r le sont. Le signe de r^2 - d^2 est donc entierement determine par le signe de (r - d). Si d <= r, alors (r - d) >= 0 et le produit (r - d)(r + d) >= 0, donc r^2 >= d^2, c'est-a-dire d^2 <= r^2. Reciproquement, si d^2 <= r^2, alors (r - d)(r + d) >= 0 ; comme (r + d) >= 0, le facteur (r - d) ne peut etre que positif ou nul, donc d <= r. Les deux conditions sont bien equivalentes.

Cette equivalence n'est valable que parce que les deux membres sont positifs : elever au carre une inegalite n'est pas anodin si des nombres negatifs sont en jeu. Ici, distances et rayons etant par nature positifs, la condition (dx*dx* + *dy*dy) <= (r*r) donne exactement le meme verdict que la condition geometrique d <= r, mais sans la racine carree. Le test elimine ainsi l'unique operation flottante couteuse tout en restant mathematiquement irreprochable : un compromis ideal entre rigueur et performance, parfaitement adapte aux contraintes du PIC16F18877.

15. La generation du son (NCO1)

Le son occupe une place modeste mais essentielle dans le ressenti d'un jeu de type Asteroids : un bruit sec accompagne chaque tir, et une descente sonore grave signale la fin de la partie. Sur le PIC16F18877, cet habillage sonore est produit par un peripherique materiel dedie, le NCO1, sans mobiliser le processeur. Ce choix architectural prolonge le fil rouge du projet : menager un microcontroleur lent, depourvu d'unite de calcul flottant materielle. Ce chapitre explique d'abord le principe du NCO et la formule qui relie la valeur ecrite dans son registre a la frequence reellement entendue, puis detaille les deux fonctions sonores du jeu, et enfin le cheminement du signal jusqu'au haut-parleur via la carte son.

15.1 Le peripherique NCO (Numerically Controlled Oscillator)

Le NCO (oscillateur a commande numerique) est un peripherique du PIC16F18877 capable de generer un signal carre dont la frequence est determinee par une simple valeur entiere. Son interet, pour un microcontroleur lent et sans materiel de calcul flottant, est qu'il fonctionne de maniere totalement autonome : une fois la valeur ecrite, le materiel produit le signal sans aucune intervention du processeur. La sortie du NCO1 est routee sur la broche RA6 (patte 14), qui rejoint ensuite l'entree de la carte son.

Le principe interne repose sur un accumulateur de phase. A chaque coup d'horloge, le NCO ajoute une valeur d'increment, stockee dans le registre `NCO1INC`, a un accumulateur de 20 bits. Lorsque cet accumulateur deborde (c'est-a-dire passe au-dela de 2^20), la sortie change d'etat. Plus l'increment est grand, plus l'accumulateur deborde souvent, et plus la frequence du signal carre est elevee. La frequence de sortie est donc directement proportionnelle a la valeur `NCO1INC` : il suffit d'ecrire un entier pour choisir une note. Cette configuration des peripheriques (source d'horloge, mode du NCO, affectation de la sortie sur RA6) est mise en place a l'initialisation par le code genere par MCC (MPLAB Code Configurator), via la fonction `SYSTEM_Initialize`.

Cette approche colle parfaitement au fil rouge du projet : menager un microcontroleur lent. La generation du signal sonore est entierement deportee sur le materiel ; le programme se contente d'ecrire une valeur, ce qui represente une seule operation. Le processeur reste donc libre de gerer la logique du jeu pendant que le son est produit.

15.2 La formule exacte et l'approximation par decalage de bits

La source d'horloge du NCO1 est fixee a FOSC/4, soit 8 MHz (8 000 000 Hz), puisque FOSC vaut 32 MHz (oscillateur interne HFINTOSC). L'accumulateur etant code sur 20 bits, le facteur de division est 2^20 = 1 048 576. La frequence de sortie obeit donc a la relation suivante :

```
f = (8 000 000 x NCO1INC) / 2^20
```

Le rapport 8 000 000 / 2^20 vaut environ 7,6294. La formule se resume alors a :

```
f = NCO1INC x 7,6294 Hz
```

Autrement dit, pour obtenir une frequence donnee, il faudrait theoriquement ecrire `NCO1INC = f / 7,6294`, c'est-a-dire `NCO1INC = f x (2^20 / 8 000 000) = f x 0,1311`. Or une multiplication par un facteur fractionnaire impose un calcul flottant, lent et couteux sur ce PIC. Le code contourne ce probleme par une astuce : il remplace la multiplication par 0,1311 par un simple decalage de bits vers la droite de 3 positions, ce qui equivaut a une division par 8 (soit une multiplication par 0,125).

```
NCO1INC = frequence >> 3;
```

Un decalage de bits est l'une des operations les plus rapides du processeur : elle se fait en quelques cycles seulement, sur des entiers, sans aucun recours au flottant. Verifions maintenant l'erreur introduite. La valeur ecrite est `NCO1INC = frequence / 8` (division entiere). La frequence reellement produite vaut donc :

```
f_reelle = (frequence / 8) x 7,6294 = frequence x (7,6294 / 8) = frequence x 0,9537
```

On constate que le ton entendu vaut environ `frequence x 0,9537`, c'est-a-dire seulement 4,6 % en dessous de la valeur visee. Pour un effet sonore de jeu, cet ecart est totalement inaudible : on ne cherche pas une note juste, mais un bruitage. Le facteur exact aurait ete 0,1311 ; le decalage `>> 3` (egal a 0,125) en est une approximation rapide et largement suffisante. A cet ecart de proportionnalite s'ajoute une legere erreur d'arrondi due a la division entiere par 8 (les bits perdus ne sont pas reportes), mais elle reste de l'ordre de quelques hertz et ne s'entend

pas davantage. Le gain est net : un decalage de bits sur entier au lieu d'une multiplication flottante, en parfaite coherence avec la philosophie d'economie de ressources du projet.

15.3 La fonction `son_tir()`

Le tir d'un projectile s'accompagne d'un « piou » tres court. La fonction qui le produit est volontairement minimaliste :

```
void son_tir(void){
    NCO1INC = 1300 >> 3;
    __delay_ms(6);
    NCO1INC = 0;
}
```

La premiere ligne ecrit dans `NCO1INC` la valeur `1300 >> 3`, soit 162 (1300 divise par 8). D'apres la formule etablie plus haut, la frequence entendue vaut environ 1300 x 0,9537 ≈ 1240 Hz : un son aigu, bref et percutant, adapte a un tir. La deuxieme ligne, `__delay_ms(6)`, maintient ce son pendant environ 6 millisecondes. La troisieme ligne, `NCO1INC = 0`, coupe immediatement le signal : avec un increment nul, l'accumulateur ne deborde plus, la sortie reste figee et le silence revient.

Ce delai de 6 ms est bloquant : pendant ces 6 ms, le programme n'execute rien d'autre. C'est un compromis assume. Une duree aussi courte est imperceptible pour le joueur a l'echelle de la boucle de jeu, et la simplicite du code prime ici sur l'optimisation. Un mecanisme non bloquant (a base de compteur decremente a chaque tour de boucle) aurait alourdi inutilement la gestion du tir pour un gain insignifiant. La regle « ne jamais bloquer le jeu inutilement » est donc respectee dans l'esprit : 6 ms ne perturbent pas la fluidite percue.

15.4 La fonction `son_gameover()`

La fin de partie merite un effet plus marque. La fonction `son_gameover()` produit un son d'extinction de type « power-down » : un balayage descendant en frequence, suivi de deux notes graves, evoquant un appareil qui s'eteint.

```
void son_gameover(void){
    for(int f=950; f>=120; f-=22){
        NCO1INC = f>>3;
        __delay_ms(9);
    }
    NCO1INC=0;
    __delay_ms(70);
    NCO1INC = 150>>3;
    __delay_ms(150);
    NCO1INC = 95>>3;
    __delay_ms(280);
    NCO1INC = 0;
}
```

La boucle `for` realise le balayage : la variable `f` part de 950 Hz et descend jusqu'a 120 Hz par pas de 22 Hz. A chaque iteration, `NCO1INC = f >> 3` fixe la note courante (toujours via le decalage rapide) et `__delay_ms(9)` la maintient 9 ms. L'oreille percoit ainsi une glissade continue vers les graves, sur environ trente-huit paliers, soit a peu pres 340 ms au total.

Apres la boucle, `NCO1INC = 0` impose un court silence de 70 ms (`__delay_ms(70)`), qui marque une rupture. Viennent ensuite deux notes graves distinctes : `NCO1INC = 150 >> 3` produit une note d'environ 150 x 0,9537 ≈ 143 Hz tenue 150 ms, puis `NCO1INC = 95 >> 3` une note encore plus grave d'environ 95 x 0,9537 ≈ 90 Hz tenue 280 ms. Le `NCO1INC = 0` final coupe definitivement le son. L'enchainement « descente, silence, deux coups graves » donne l'impression caracteristique d'un systeme qui s'arrete.

L'usage de delais bloquants est ici parfaitement justifie. Lorsque cette fonction est appelee, la partie est terminee (etat GAME OVER) : plus rien ne doit bouger a l'ecran, aucun asteroide ni

projectile n'a a etre mis a jour. Bloquer le programme le temps de jouer la sequence sonore n'a donc aucune consequence sur la jouabilite. C'est l'exception qui confirme la regle : on ne bloque jamais le jeu inutilement, mais ici l'attente est utile car le jeu est justement arrete.

On remarquera aussi que l'attente est decoupee en plusieurs petits delais (9 ms, 70 ms, 150 ms, 280 ms) plutot qu'en une seule longue temporisation. La duree maximale acceptee par la macro `__delay_ms` est en effet limitee sur le PIC, car elle est traduite par le compilateur XC8 en une boucle de comptage dont le nombre de cycles depend de la frequence d'horloge et reste borne. Decouper en petits delais respecte cette contrainte tout en composant librement la duree totale de l'effet.

15.5 Le lien avec la carte son

Le signal carre genere sur la broche RA6 n'attaque pas directement un haut-parleur : il est d'abord amplifie. Il rejoint le connecteur J2 (Audio_In) de la carte son, batie autour de l'amplificateur audio LM386. Celui-ci eleve le niveau du signal avec un gain de 20 (valeur par defaut, les broches 1 et 8 du LM386 n'etant pas pontees), puis l'envoie au haut-parleur via le connecteur de sortie J3 (Audio_out). Le potentiometre RV1 (10 kOhm) place en entree permet de regler le volume, tandis que le condensateur de couplage C1 (1 uF) isole l'entree en continu. L'architecture de cette carte est detaillee dans le chapitre consacre a la carte son ; on en rappelle ci-dessous le decoupage fonctionnel.

Figure 35 — decoupage fonctionnel de la carte son LM386 recevant le signal de RA6.

Un dernier detail temoigne du soin apporte a l'experience sonore : au demarrage du programme, `NCO1INC` est mis a 0. Cette precaution coupe tout son des l'initialisation et evite qu'un bourdonnement parasite ne se fasse entendre avant la premiere note volontaire. Le silence est ainsi l'etat par defaut, et le son n'apparait qu'au moment choisi par le jeu : un tir, ou la fin de la partie.

16. L’affichage des ecrans, le HUD et la boucle de jeu

Apres avoir decrit separement le joystick, le vaisseau, les projectiles, les asteroides et les collisions, il reste a assembler ces briques dans une experience coherente. Ce chapitre traite de deux aspects complementaires : d'une part la fabrication des ecrans fixes du jeu (menu, reglages, fin de partie) et de l'element d'interface persistant qu'est le compteur de score (HUD) ; d'autre part la `boucleJeu()`, qui orchestre image apres image l'ensemble des traitements pendant une partie. Comme dans tout le projet, le fil rouge demeure le menagement du PIC16F18877 : un microcontroleur 8 bits cadence a FOSC = 32 MHz (cycle instruction Tcy = 125 ns), depourvu d'unite de calcul flottant materielle, qui ne peut se permettre ni des calculs lourds inutiles, ni des reaffichages complets de l'ecran a chaque tour. Pour memoire, l'ecran ILI9488 affiche 480 x 320 pixels en mode paysage et chaque pixel envoye coute trois octets (couleur 18 bits) transmis sur le bus SPI : economiser des pixels reaffichis revient donc directement a economiser du temps machine.

16.1 Le fond etoile : `dessinerFondEtoiles()`

Tous les ecrans du jeu partagent un meme decor : un ciel noir parseme d'etoiles. La constante `STAR_COUNT=18` fixe le nombre d'etoiles. Plutot que de tirer ces positions au hasard a chaque affichage (ce qui ferait scintiller le decor et consommerait des appels a `rand()`), les 18 etoiles occupent des positions fixes, decidees une fois pour toutes. Une etoile sur trois est dessinee en orange (`COL_ORANGE=0xFD20`), les autres en blanc, ce qui donne un leger relief visuel sans surcout. Le trace lui-meme se reduit a un point, voire un tres petit motif, place via les primitives `display_drawPixel` ou `display_fillRect` de la bibliotheque graphique. Le choix de positions fixes illustre directement le principe « ne calculer que le strict necessaire » : un decor immobile n'a aucune raison d'etre recalcule, on se contente de le redessiner a l'identique lorsqu'on repeint un ecran complet.

16.2 Le menu d'accueil : `afficherMenu()`

La fonction `afficherMenu()` compose le premier ecran rencontre par le joueur. Elle commence par remplir l'ecran en noir, puis appelle `dessinerFondEtoiles()` pour poser le decor. Vient ensuite un cadre orange epais, obtenu non pas par une primitive de bordure dediee mais par quatre rectangles imbriques traces successivement : en superposant quatre contours de tailles

legerement decroissantes, on obtient visuellement un trait epais et net, alors que display_drawRect ne dessine qu'un trait d'un pixel. Cette astuce evite d'avoir a remplir une zone pleine puis a la recreuser, operation qui serait plus couteuse en pixels ecrits sur le bus SPI.

A l'interieur du cadre figurent le titre ASTEROIDS, les credits du binome (Emma Grosmaire Sausse Dorian) et un petit vaisseau decoratif, dessine avec les memes primitives de triangle que le vaisseau de jeu. Enfin, deux lignes rappellent les commandes de navigation : la touche C lance une partie (C -> JOUER) et la touche D ouvre l'ecran des reglages (D -> REGLAGES). Ces libelles correspondent au brochage reel : BTN_C est cable sur RD5 (patte 28) et BTN_D sur RD4 (patte 27), tous deux actifs a l'etat bas (resistance de tirage pull-up : relache = 1, appuye = 0).

Figure 36 — ecran d'accueil : cadre orange, titre, credits et choix JOUER ou REGLAGES.

16.3 L'ecran des commandes : afficherReglages()

La fonction afficherReglages() reprend la meme trame (fond noir, etoiles, cadre) et y affiche la notice des commandes sous le titre COMMANDES. Trois lignes resument l'usage : le bouton B sert a tirer (.B -> TIR), le joystick sert a orienter le vaisseau (.JOYSTICK -> VISER) et la touche C ramene au menu (C -> RETOUR). Cet ecran est purement informatif : aucune logique de jeu n'y tourne, le programme se contente d'attendre l'appui sur C pour revenir a l'etat ETAT_MENU. Sa presence repond au besoin pedagogique de rendre le jeu auto-explicatif, sans documentation

externe.

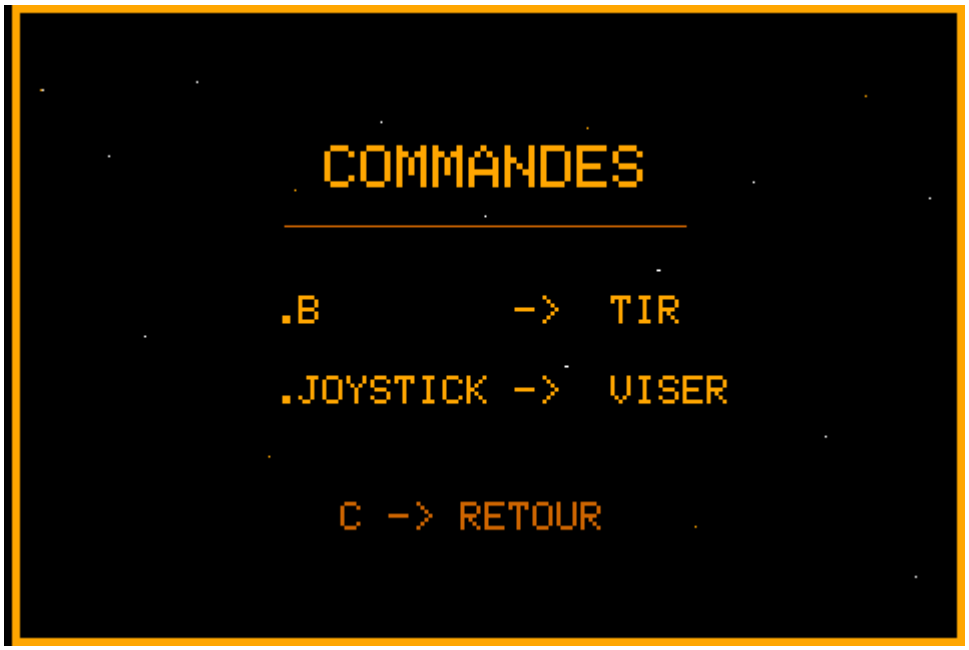


Figure 37 — ecran des commandes : correspondance entre boutons, joystick et actions.

16.4 La fin de partie : afficherGameOver()

Lorsqu'un asteroide percute le vaisseau, le jeu bascule dans l'etat ETAT_GAMEOVER et appelle afficherGameOver(). Cet ecran affiche le message GAME OVER ainsi que le score final. Le score, qui est une valeur entiere variable, ne peut pas etre ecrit en dur : on le met en forme dans une chaine de caracteres a l'aide de sprintf :

```
sprintf(buf, "SCORE : %d", score);
```

sprintf ecrit dans le tampon buf le texte litteral SCORE : suivi de la valeur entiere de score (format %d). La chaine ainsi fabriquee est ensuite affichee par display_printText. Le format specifie %d (entier decimal) est coherent avec le type de score. Enfin, un rappel C -> MENU indique au joueur comment relancer le cycle. Cet usage de sprintf provient de la bibliotheque standard <stdio.h> incluse dans main.c.

Figure 38 — ecran de fin de partie : message GAME OVER et score final.

16.5 Le compteur de score : `afficherScore()` et son optimisation

Pendant une partie, le score s'affiche en permanence dans une bande superieure (le HUD, de hauteur `HUD_H=24` pixels, courant sur toute la largeur `SCREEN_W=480`). Le rafraichir entierement a chaque tour de boucle, alors qu'il ne change que lorsqu'un asteroide est detruit (gain de +10 points), provoquerait un clignotement disgracieux et gaspillerait du temps de calcul. La fonction `afficherScore()` resout ce probleme par une optimisation classique : ne reafficher que si la valeur a reellement change.

```
void afficherScore(void){
    char buf[10];
    if(score==oldScore) return;
    oldScore=score;
    display_fillRect(82,4,65,18,ILI9488_BLACK);
    sprintf(buf,"%d",score);
    display_printText(buf,82,4,COL_ORANGE,ILI9488_BLACK,2);
```

```
}
```

La premiere ligne declare un tampon `buf` de dix caracteres, largement suffisant pour representer le score. Le test `if(score==oldScore) return;` est le coeur de l'optimisation : `oldScore` memorise la derniere valeur affichee, et tant que le score n'a pas bouge, la fonction sort immediatement sans rien redessiner. Des qu'un changement survient, on met a jour `oldScore=score` pour le prochain tour. On efface ensuite l'ancien nombre en peignant un rectangle noir aux coordonnees fixes (`display_fillRect(82,4,65,18,ILI9488_BLACK)`) : ce rectangle, de 65 pixels de large sur 18 de haut et place a partir du point (82, 4), couvre exactement la zone du HUD reservee a la valeur numerique. On reconstruit alors la chaine du score avec `sprintf(buf,"%d",score)` (ici sans libelle, juste le nombre), puis on l'affiche en orange sur fond noir, en taille 2, via `display_printText(buf,82,4,COL_ORANGE,ILI9488_BLACK,2)`. Effacer puis reecrire uniquement la zone du nombre, et seulement quand c'est utile, evite a la fois le clignotement et l'ecriture inutile d'octets sur le bus SPI.

Figure 39 — bande de score (HUD) en haut de l'ecran de jeu.

16.6 Le compte a rebours : `compteARebours()`

Avant que l'action ne demarre, un compte a rebours affiche successivement les chiffres 4, 3, 2 puis 1, a raison d'un chiffre par seconde, pour laisser au joueur le temps de saisir le joystick. La difficulte technique vient de la macro `__delay_ms` du compilateur XC8 : sa duree maximale est bornee, et l'on ne peut pas lui demander directement une attente d'une seconde pleine. La solution consiste a fractionner chaque seconde en dix attentes de 100 ms (soit 10 x 100 ms = 1000 ms) : on boucle dix fois sur un `__delay_ms(100)` pour totaliser une seconde par chiffre. Entre deux chiffres, on efface l'ancien et on dessine le suivant, toujours selon le principe « effacer puis redessiner » applique a une zone restreinte de l'ecran.

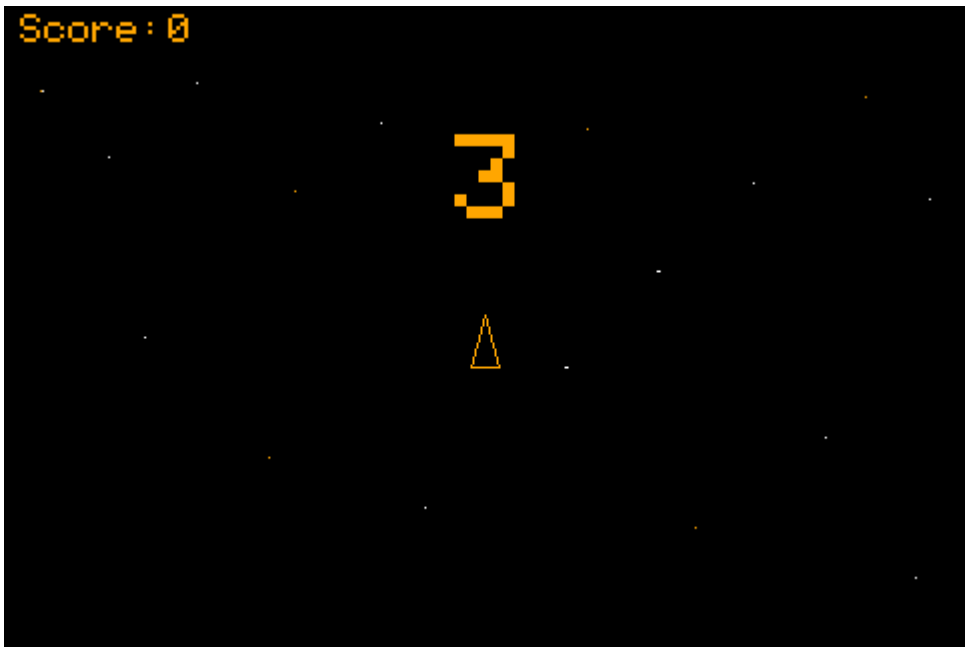


Figure 40 — affichage du compte a rebours (ici le chiffre 3) avant le lancement de la partie.

16.7 La reinitialisation : initJeu()

Chaque nouvelle partie doit repartir d'un etat propre. La fonction initJeu() realise cette remise a zero complete : elle repeint l'ecran de jeu, remet le score (et oldScore) a zero, reinitialise les compteurs internes (cadence de tir shootCooldown, compteur d'apparition d'asteroides spawnCounter, compteur de redessin du vaisseau shipDrawCounter), coupe le son en cours et oriente le vaisseau vers le haut en imposant shipCos=0 et shipSin=-1. Ce vecteur unitaire (0, -1) correspond a une direction « plein haut » : a l'ecran, l'axe Y croit vers le bas, donc une composante verticale negative pointe vers le sommet de l'ecran. Tous les tirs et tous les asteroides sont desactives (drapeaux active remis a 0). La fonction recalibre egalement le joystick au repos, ce qui fixe la reference du centre pour la zone morte JOY_DEADZONE=32, puis lance le compte a rebours decrit ci-dessus. A son issue, le jeu peut entrer dans ETAT_JEU avec des variables parfaitement coherentes. La coupure du son passe par le peripherique NCO1 (qui pilote la sortie sonore sur RA6, patte 14) : remettre l'increment NCO1INC a zero arrete la generation de frequence, evitant qu'un bip de la partie precedente ne se prolonge.

16.8 Le chef d'orchestre : boucleJeu()

La fonction boucleJeu() est appelee en continu tant que l'on reste dans l'etat ETAT_JEU. Elle enchaine les traitements dans un ordre precis, chaque tour de boucle correspondant a une

image du jeu.

```
void boucleJeu(void){
    afficherScore();
    bool angleChange = lireJoystick();
    if(angleChange){
        tracerVaisseau(ILI9488_BLACK);
        calculerSommetsVaisseau();
        tracerVaisseau(COL_ORANGE);
        shipDrawCounter=0;
    }
    if(shootCooldown>0) shootCooldown--;
    if(appuiB() && shootCooldown==0){ tirer(); shootCooldown=3; }
    spawnCounter++;
    if(spawnCounter>=SPAWN_DELAY){ spawnCounter=0; spawnAsteroide(); }
    majAsteroides();
    majTirs();
    shipDrawCounter++;
    if(shipDrawCounter>=12){ tracerVaisseau(COL_ORANGE); shipDrawCounter=0; }
    collisions();
}
```

L'ordre des etapes n'est pas arbitraire. On commence par `afficherScore()`, qui ne fait rien si le score est inchange (cf. 16.5). On lit ensuite le joystick : `lireJoystick()` renvoie un booleen `angleChange` indiquant si l'orientation du vaisseau a suffisamment varie pour justifier un

redessin. Le seuil est gouverne par la constante `ANGLE_REDRAW_DOT=0.9997` : on compare le produit scalaire de l'ancien et du nouveau vecteur d'orientation (tous deux unitaires) a cette valeur. Comme pour deux vecteurs unitaires le produit scalaire vaut le cosinus de l'angle entre eux, redessiner uniquement quand ce produit descend sous 0,9997 revient a exiger un ecart angulaire superieur a arccos(0,9997), soit environ 1,4 degre. En deca, le vaisseau bougerait si peu que le redessin serait imperceptible et inutilement couteux. Si le seuil est franchi, on applique le triptyque effacer / deplacer / redessiner : `tracerVaisseau(ILI9488_BLACK)` efface l'ancien vaisseau en le repeignant en noir, `calculerSommetsVaisseau()` recalcule ses trois sommets a partir du nouveau vecteur `(shipCos, shipSin)`, puis `tracerVaisseau(COL_ORANGE)` le redessine a sa nouvelle orientation. On remet aussi `shipDrawCounter` a zero, car le vaisseau vient d'etre repeint.

Vient la gestion du tir. La variable `shootCooldown` impose un delai minimal entre deux tirs : a chaque tour, si elle est positive, on la decremente. Lorsque le bouton B est enfonce (`appuiB()`, cable sur RD6, patte 29) et que le cooldown est nul, on declenche `tirer()` puis on recharge `shootCooldown=3`. Ce mecanisme empeche une rafale ininterrompue tant que le joueur maintient le bouton, sans avoir besoin de detecter un front montant complexe.

On gere ensuite l'apparition des asteroides via un compteur : `spawnCounter` augmente d'une unite par tour, et lorsqu'il atteint `SPAWN_DELAY=19`, on le remet a zero et l'on fait surgir un asteroide avec `spawnAsteroide()`. Un asteroide nait donc tous les dix-neuf tours de boucle, ce qui regule la difficulte (jusqu'a `MAX_ASTEROIDS=4` simultanement). Puis `majAsteroides()` et `majTirs()` deplacent et redessinent respectivement les asteroides et les projectiles encore actifs.

Le bloc suivant illustre un point subtil du rendu incrementiel. Comme les asteroides et les tirs effacent au passage les pixels du vaisseau (chacun gerant sa propre zone par effacement/redessin), le vaisseau finirait par se trouver partiellement « grignote ». Pour reparer ces degats sans repeindre le vaisseau a chaque image, on incremente `shipDrawCounter`, et tous les douze tours seulement on force un `tracerVaisseau(COL_ORANGE)` puis on remet le compteur a zero. C'est un compromis economique : redessiner le vaisseau en permanence couterait cher, ne jamais le redessiner laisserait des trous, le faire periodiquement maintient un rendu propre a moindre frais. Enfin, `collisions()` clot le tour en testant les rencontres entre tirs et asteroides (gain de 10 points, destruction) et entre asteroides et vaisseau (declenchement du GAME OVER), en comparant des distances au carre pour eviter les racines carrees couteuses.

Figure 41 — organigramme de la boucle de jeu : ordre des etapes executees a chaque image.

Cet ordonnancement resume a lui seul la philosophie du projet : lire les entrees, mettre a jour les objets mobiles, ne redessiner que ce qui change ou ce qui a ete abime, et reserver les calculs couteux aux seuls cas ou ils sont indispensables. C'est ce qui permet a un microcontroleur 8 bits sans virgule flottante materielle de faire tourner un jeu d'action fluide.

17. Cablage et integration materielle

Une fois les trois sous-ensembles fabriques et valides separement (la carte porteuse du microcontroleur, l'ecran TFT et la carte son), l'etape d'integration consiste a les relier entre eux pour former un systeme unique et fonctionnel. Cette phase, en apparence simple, conditionne pourtant la fiabilite de l'ensemble: une inversion de fil, une masse oubliee ou une alimentation mal repartie suffit a empecher tout demarrage, voire a endommager un composant. On procede donc methodiquement, sous-systeme par sous-systeme, en s'appuyant systematiquement sur le brochage du PIC16F18877 etabli precedemment. L'idee directrice reste la meme que pour le logiciel: maitriser chaque liaison pour ne laisser aucune place a l'ambiguite, et reduire les sources d'erreur la ou le microcontroleur lent ne pourrait pas les compenser.

17.1 Preparation et reperage des connexions

Avant tout cablage, on reporte sur papier l'ensemble des liaisons a realiser a partir du brochage complet du microcontroleur. Le PIC16F18877 est un boitier 40 broches PDIP, 8 bits: chaque patte recoit une affectation unique (entree analogique, sortie SPI, entree bouton, sortie son), et il est imperatif de ne jamais confondre deux pattes voisines. On etablit donc un plan de cablage qui associe, pour chaque signal, le nom logique (par exemple SCK ecran), le port du PIC (RC3) et le numero de patte physique (18). Ce plan devient ensuite la reference unique pour le brasage, la verification et le depannage.

Pour rendre le cablage plus propre et plus robuste autour du microcontroleur, le montage repose sur deux cartes empilees. La carte rouge porte le PIC16F18877 lui-meme ainsi que les composants strictement necessaires a son fonctionnement (decouplage, ligne de reset MCLR, alimentation). La carte verte, posee en regard, joue le role de carte d'adaptation: elle ajoute des borniers a vis qui prolongent les pattes du PIC vers l'exterieur. Plutot que de souder directement des fils volants sur les broches fragiles du circuit integre, on visse les conducteurs dans ces borniers, ce qui facilite le reperage, autorise les modifications et limite les risques de faux contact. Cette separation en deux cartes presente un avantage pratique: la carte rouge peut rester intacte pendant que l'on remanie le cablage exterieur sur la carte verte.

Figure 42 — Ensemble des deux cartes: la carte rouge porte le microcontroleur, la carte verte ajoute les borniers de cablage.

Le materiel preparatoire se limite a des fils de cablage de couleurs differenciees, un tournevis fin pour les borniers, ainsi qu'un multimetre pour les controles ulterieurs. Le choix d'un code couleur coherent (par exemple le gris reserve aux masses) facilite considerablement la verification visuelle et le depannage: une masse oubliee se reperant alors d'un simple coup d'oeil.

17.2 Cablage de l'ecran TFT ILI9488

L'ecran se connecte au PIC par le bus SPI1 (MSSP1) configure en maitre, en mode SPI 0. Le module presente huit bornes utiles, dans l'ordre suivant: VCC, GND, SCLK, MOSI, MISO, CS, RES et DC. Le tableau ci-dessous donne la correspondance exacte avec le PIC.

1 VCC	Alimentation	VDD +5V	11 / 32	Alimentation logique de l'ecran
2 GND	Masse	VSS	12 / 31	Reference de masse commune
3 SCLK	Horloge SPI	RC3	18	Cadence les bits envoyes
4 MOSI	Donnees SPI	RC5	20	PIC vers ecran (SDI)
5 MISO	Retour SPI	---	---	Non utilise (sens unique)
6 CS	Selection	RC0	15	Active le dialogue avec l'ecran
7 RES	Reset	RC1	16	Reinitialisation materielle
8 DC	Data/	RC2	17	Distingue commande (0) et

Command donnee (1)

La communication est volontairement a sens unique: le PIC ecrit vers l'ecran (MOSI seul) et ne lit jamais sa reponse. La ligne MISO de l'ecran existe physiquement mais n'est pas raccordee, ce qui economise une patte et simplifie le cablage. Cette economie est sans risque ici, car le pilote bas niveau ne fait jamais de lecture de registre de l'ecran. La ligne DC est essentielle: combinee a CS, elle permet au pilote de basculer entre l'envoi d'une commande (DC=0, fonction ILI9488_SendCommand) et l'envoi d'une donnee (DC=1, fonction ILI9488_SendData). En pratique, CS abaisse selectionne l'ecran, l'octet est cadence sur SCK (RC3) et sort sur MOSI (RC5), et DC indique si cet octet doit etre interprete comme une instruction ou comme une donnee de pixel. Les bornes 9 a 13 du module (BKL, SCL, SDA, INT, SDCS) ne sont pas raccordees car elles concernent des fonctions inutilisees ici (retroeclairage commande, interface tactile I2C, lecteur de carte SD).

Figure 43 — Liaison de l'ecran ILI9488 au PIC: huit bornes cablees, bus SPI et alimentation.

17.3 Cablage de la manette (joystick et boutons)

La manette regroupe le joystick analogique deux axes et les boutons poussoirs. Les deux axes du joystick sont lus par le convertisseur analogique-numerique ADCC (10 bits, valeurs de 0 a 1023): l'axe X est raccorde a RC7 (patte 22) et l'axe Y a RC6 (patte 21). Le joystick recoit egalement son alimentation VCC et sa masse GND, communes au reste du systeme. Ce sont ces deux valeurs analogiques, une fois centrees et normalisees par le logiciel, qui fournissent directement le vecteur d'orientation (cos, sin) du vaisseau, sans aucun calcul d'angle.

Les boutons poussoirs sont cables en logique active a l'etat bas: une resistance de tirage (pull-up) maintient l'entree a 1 au repos, et l'appui force l'entree a 0. Le programme teste donc l'etat 0 pour detecter un appui. Trois boutons sont effectivement utilises:

B	RD6	29	Tir d'un projectile
C	RD5	28	Jouer / Retour / Menu
D	RD4	27	Acces aux reglages
A	RD7	30	Non utilise

Le bouton A (RD7, patte 30) est cable ou laisse disponible mais n'est associe a aucune action dans le code. Ce choix de logique active-bas est classique et robuste: il evite les etats flottants et garantit une lecture stable des appuis, meme avec des fils de quelques dizaines de centimetres entre la manette et la carte.

Figure 44 — Cablage de la manette: axes X (RC7) et Y (RC6) du joystick et boutons actifs-bas.

17.4 Cablage de la carte son

La carte son, batie autour de l'amplificateur LM386, recoit le signal audio genere par le peripherique NCO1 du PIC, disponible sur la patte RA6 (patte 14). Ce signal carre est achemine vers le connecteur d'entree J2 (Audio_In) de la carte. Sur la photographie, on distingue deux cables marron: l'un transporte le signal audio depuis la patte RA6 vers le PCB, l'autre amene le VCC, raccorde a proximite de l'interrupteur a glissiere SW1. Les fils gris correspondent aux masses, conformement au code couleur retenu.

Le cheminement du signal sur la carte est le suivant: J2 amene le signal vers le condensateur de couplage C1 (1 uF), puis vers le potentiometre de volume RV1 (10 k), avant d'attaquer l'entree non inverseuse du LM386 (broche 3, la broche 2 etant a la masse). La sortie de l'amplificateur (broche 5) passe par le reseau de Zobel (R2 de 100 ohms et C5 de 47 nF) qui stabilise l'etage et previent les oscillations, puis par le condensateur de liaison C6 (220 uF) vers le connecteur de sortie J3 relie au haut-parleur. L'alimentation arrive par J1 et l'interrupteur SW1, decouplee par C2 (100 nF) et C3 (10 uF); la broche BYPASS (7) du LM386 est decouplee par C4 (10 uF). Le gain de l'amplificateur est fixe a 20 par defaut, car les broches 1 et 8 ne sont pas pontees. Les masses des trois cartes sont reunies en un point commun, condition indispensable pour que le signal carre de RA6 soit interprete correctement par l'etage d'entree.

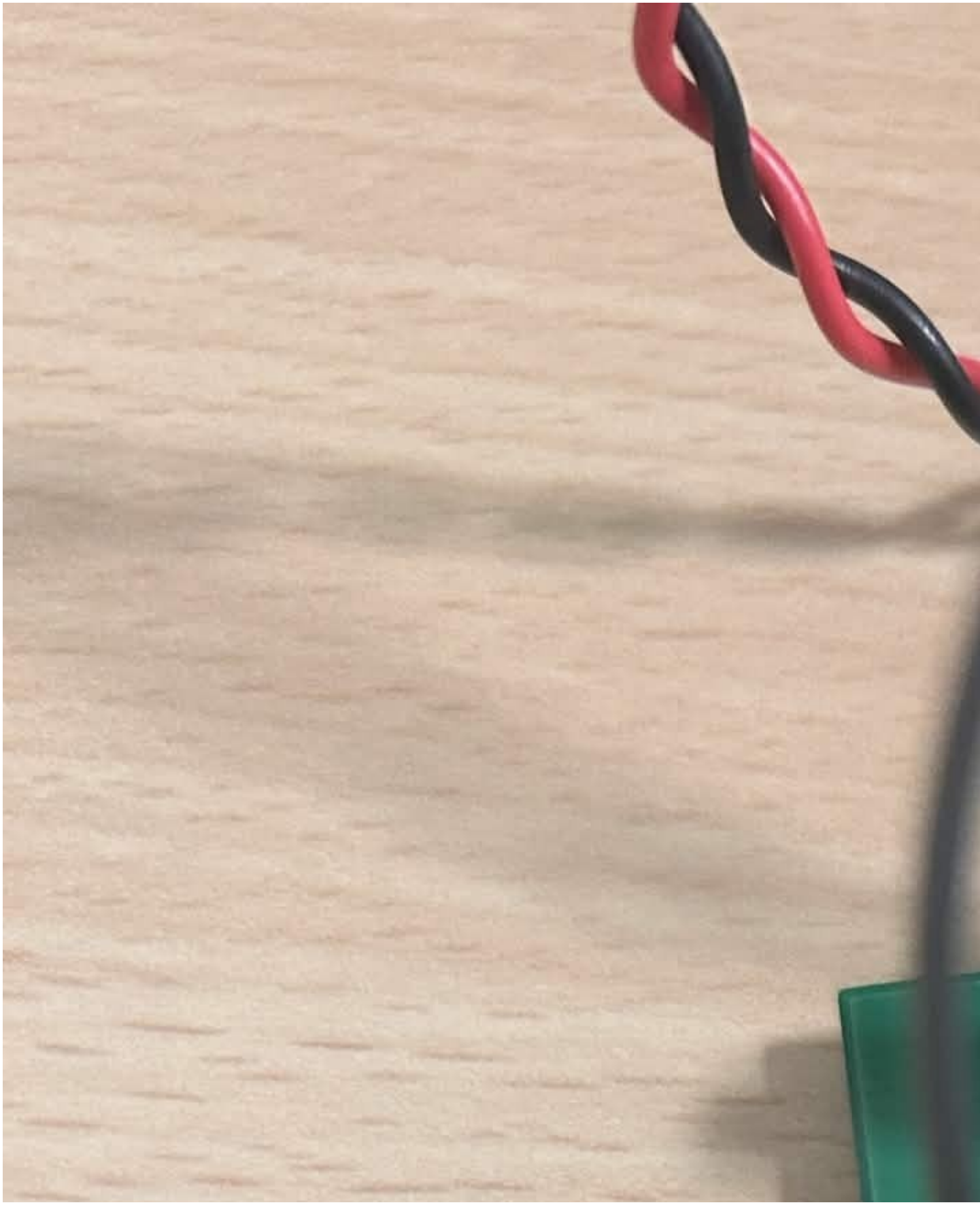


Figure 45 — Carte son raccordee: entree audio depuis RA6, alimentation pres de l'interrupteur, masses en gris.

17.5 Verification des connexions

Avant la premiere mise sous tension complete, on procede a un controle systematique, en plusieurs etapes. On commence par un controle visuel de toutes les liaisons, en confrontant chaque fil au plan de cablage. On realise ensuite des tests de continuite au multimetre, hors tension: on verifie que chaque signal relie bien la patte attendue du PIC a la borne attendue du sous-systeme, et qu'aucun court-circuit n'existe entre deux pattes voisines ou entre un signal et la masse. On confirme en particulier que toutes les masses sont communes (continuite entre les GND de l'ecran, de la manette et de la carte son). Enfin, sous tension, on mesure la tension d'alimentation: VDD doit valoir +5 V sur les pattes 11 et 32, VSS etant la reference de masse sur les pattes 12 et 31, MCLR (patte 1) etant la ligne de reset qui doit etre maintenue a l'etat haut pour autoriser l'execution.

Le tableau recapitulatif ci-dessous reunit, patte par patte, l'ensemble des connexions du systeme. Il sert de reference unique lors de la verification et du depannage.

1	MCLR	PIC	Reset
11	VDD	PIC	Alimentation +5V
12	VSS	PIC	Masse (GND)
14	RA6	Carte son	Sortie son NCO1 (vers J2 Audio_In)
15	RC0	Ecran	CS (selection)
16	RC1	Ecran	RST (reset)
17	RC2	Ecran	DC (Data/Command)

18	RC3	Ecran	SCK (horloge SPI)
20	RC5	Ecran	SDI / MOSI (donnees SPI)
21	RC6	Manette	Joystick axe Y (ADC)
22	RC7	Manette	Joystick axe X (ADC)
27	RD4	Manette	Bouton D (REGLAGES), actif-bas
28	RD5	Manette	Bouton C (JOUER/RETOUR/MENU), actif-bas
29	RD6	Manette	Bouton B (TIR), actif-bas
30	RD7	Manette	Bouton A (non utilise)
31	VSS	PIC	Masse (GND)
32	VDD	PIC	Alimentation +5V

Pour replacer ces liaisons dans leur contexte, le schema bloc rappelle l'architecture globale du systeme: le PIC16F18877 au centre, l'ecran ILI9488 sur le bus SPI1, la manette en entree (ADCC et boutons) et la carte son en sortie sur RA6.

Figure 46 — Rappel de l'architecture: microcontroleur central, ecran, manette et carte son.

A l'issue de ces controles, le systeme est pret pour les essais fonctionnels: affichage du menu, lecture du joystick, reaction des boutons et restitution sonore. Toute anomalie constatee au

demarrage renvoie immediatement au tableau de cablage, qui constitue le point d'entree naturel du diagnostic. Cette discipline de reperage et de verification, menee patte par patte, est ce qui transforme trois sous-ensembles independants en un produit unique et fiable.

18. Tests, validation et difficultes rencontrees

Un prototype embarque ne se valide pas d'un seul bloc. La demarche adoptee a consiste a verifier d'abord chaque sous-systeme isolement (l'ecran, la carte son, le joystick, les boutons, le generateur de son), puis a integrer l'ensemble et a observer le jeu complet en fonctionnement. Cette progression du composant vers le systeme permet de localiser rapidement l'origine d'un dysfonctionnement : un defaut detecte tot, sur un module isole, coute bien moins cher a corriger qu'un defaut decouvert dans le jeu assemble. Ce chapitre presente les tests unitaires par sous-systeme, les tests d'integration, une synthese qualitative des resultats, puis un inventaire raisonne des difficultes rencontrees, classees par nature et accompagnees pour chacune de leur cause, de leur consequence et de la solution appliquee.

18.1 Tests unitaires par sous-systeme

Ecran TFT ILI9488. Le premier test consiste a executer l'initialisation `ILI9488_Init` puis a remplir tout l'ecran d'une couleur unie via `display_fillScreen`. Si le panneau s'allume et affiche un aplat homogene, c'est que la sequence d'init (sleep out, display on, format de pixel 18 bits 0x66) et la liaison SPI fonctionnent. On teste ensuite l'adressage avec `display_fillRect` pour verifier que la fenetre `setAddrWindow` cible les bonnes colonnes (CASET 0x2A) et lignes (PASET 0x2B) avant l'ecriture en memoire video (RAMWR 0x2C). Enfin, l'affichage de texte par `display_printText` valide la police 5x7 et l'orientation paysage (`setRotation(3)`, soit 480x320 pixels). Un ecran qui resterait blanc ou afficherait du bruit oriente immediatement vers le cablage SPI (SCK sur RC3, MOSI/SDI sur RC5, CS sur RC0, DC sur RC2, RST sur RC1) ou vers l'alimentation du module.

Carte son. Le module amplificateur a base de LM386 (U1) se teste en injectant un signal connu sur l'entree J2 (Audio_In), c'est-a-dire la sortie NCO1 du PIC sur RA6 (patte 14). Apres mise sous tension par le connecteur J1 et l'interrupteur a glissiere SW1, on ecoute la presence du son dans le haut-parleur relie a J3 (Audio_out) et on agit sur le potentiometre RV1 (10 kOhm) pour verifier que le volume varie de facon continue. On controle l'absence de souffle excessif ou d'oscillation parasite, signe que le decouplage (C2 100 nF, C3 10 uF, et C4 10 uF sur la broche BYPASS du LM386) et le reseau de Zobel (R2 100 Ohm + C5 47 nF) jouent bien leur role de stabilisation.

Joystick analogique. On verifie d'abord que le convertisseur ADCC (10 bits) renvoie une valeur coherente sur la plage 0 a 1023 pour chacun des deux axes (axe X sur RC7 / patte 22, axe Y sur RC6 / patte 21). Au repos, les deux axes doivent se situer pres du milieu de course. La routine de calibration moyenne plusieurs releves pour fixer le centre `joyCx`/`joyCy` ; on s'assure ensuite que pousser le manche dans une direction fait croitre ou decroitre la valeur de maniere monotone, et que la zone morte `JOY_DEADZONE` (valeur 32) supprime bien le tremblement au repos.

Boutons poussoirs. Les boutons etant actifs a l'etat bas (resistance de tirage pull-up : relache = 1, appuye = 0), on verifie la lecture de chaque entree : BTN_B (RD6 / patte 29, tir), BTN_C (RD5 / patte 28, jouer/retour/menu) et BTN_D (RD4 / patte 27, reglages). Le bouton BTN_A (RD7 / patte 30) n'est pas utilise. Le test porte surtout sur la detection de front : un appui ne doit declencher qu'une seule action, meme si le doigt reste pose. On confirme donc qu'un tir part une fois par appui et non en rafale tant que le bouton est maintenu.

Generateur de son NCO1. On valide enfin que la frequence demandee correspond bien au ton entendu. Le NCO1, dont l'horloge source est FOSC/4 = 8 MHz et l'accumulateur fait 20 bits, produit une frequence de sortie :

```
f = (8 000 000 x NCO1INC) / 2^20 = NCO1INC x 7,6294 Hz
```

Le code ecrit `NCO1INC = frequence >> 3`, c'est-a-dire une division par 8 (0,125) realisee par simple decalage de bits, beaucoup plus rapide qu'une vraie division sur un coeur 8 bits. En combinant l'ecriture et la formule de sortie, le ton effectivement joue vaut environ `frequence x 0,9537`, donc tres proche de la valeur demandee. L'ecart provient du fait que le decalage `>>3` (0,125) n'est qu'une approximation du facteur theorique exact 0,1311 ; il reste inaudible a l'oreille : un bip aigu demande reste un bip aigu.

18.2 Tests d'integration

Une fois chaque brique validee, le jeu complet est assemble et eprouve. On verifie d'abord l'enchainement des quatre etats de la machine (ETAT_MENU, ETAT_REGLAGES, ETAT_JEU, ETAT_GAMEOVER) : depuis le MENU, le bouton REGLAGES (RD4) ouvre l'ecran des commandes, le bouton JOUER (RD5) lance la partie apres un compte a rebours, et une collision conduit au GAME OVER d'ou l'on revient au menu. La jouabilite est evaluee

manuellement : rotation du vaisseau a la commande du joystick, depart des projectiles au bouton de tir, surgissement des asteroides depuis les bords vers le centre. On observe la fluidite (absence de scintillement, rotation continue a l'oeil), la justesse des collisions (un tir bien place detruit l'octogone, un asteroide qui atteint le centre provoque la fin de partie) et la mise a jour correcte du score (+10 par asteroide detruit, affiche dans le bandeau HUD de hauteur HUD_H = 24 pixels).

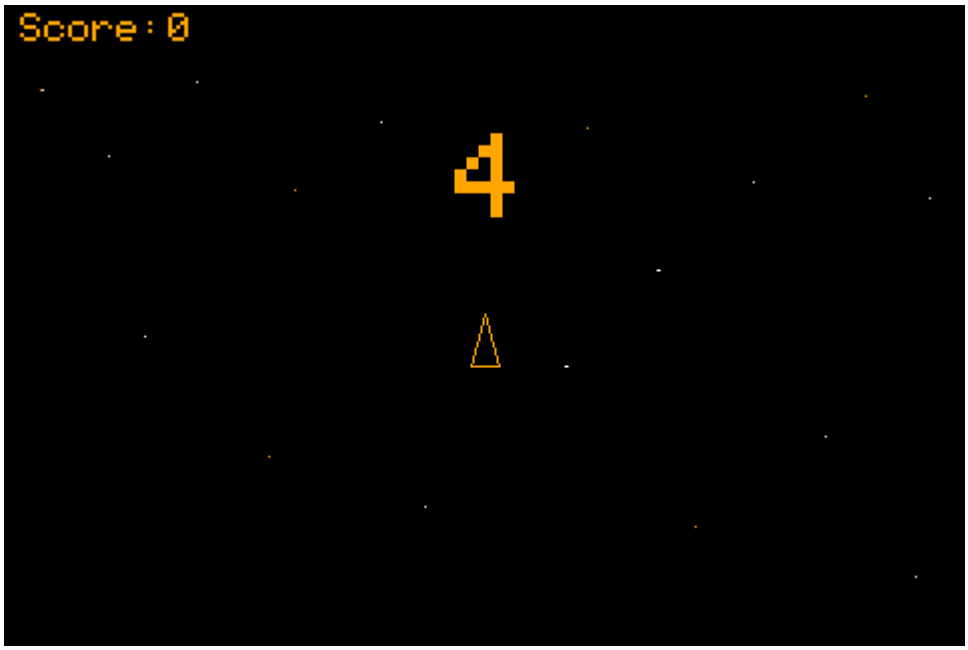


Figure 47 — Compte a rebours affiche lors du passage a l'etat JEU, verifie pendant les tests d'enchainement des etats.

18.3 Resultats obtenus

A l'issue de ces essais, le prototype est juge fonctionnel : le jeu est jouable de bout en bout, le son est present et reglable, et l'affichage restitue fidelement les formes attendues (vaisseau triangulaire au centre, asteroides octogonaux, projectiles, HUD). Le tableau ci-dessous synthetise la campagne de tests de maniere qualitative, sans chiffre mesure.

Ecran : init et remplissage	`display_fillScreen` couleur unie	Aplat homogene plein ecran	Conforme

Ecran : texte	`display_printText` police 5x7	Caracteres lisibles, bonne orientation	Conforme
Carte son : signal	Injection NCO1 sur J2, ecoute sur J3	Son audible, volume reglable par RV1	Conforme
Joystick : lecture ADC	Releve des axes X/Y (0-1023)	Variation monotone, centre stable	Conforme
Joystick : zone morte	Manche au repos	Vaisseau immobile, sans tremblement	Conforme
Boutons : front	Appui unique sur B/C/D	Une seule action par appui	Conforme
NCO : frequence	Comparaison ton demande / entendu	Ton proche de la consigne	Conforme (ecart inaudible)
Enchainement des etats	Parcours MENU -> JEU -> GAME OVER	Transitions correctes	Conforme
Collisions et score	Tirs et impacts en jeu	Destruction (+10) et GAME OVER	Conforme

L'affichage final, photographie ecran allume, confirme la fidelite du rendu observe au cours des tests.

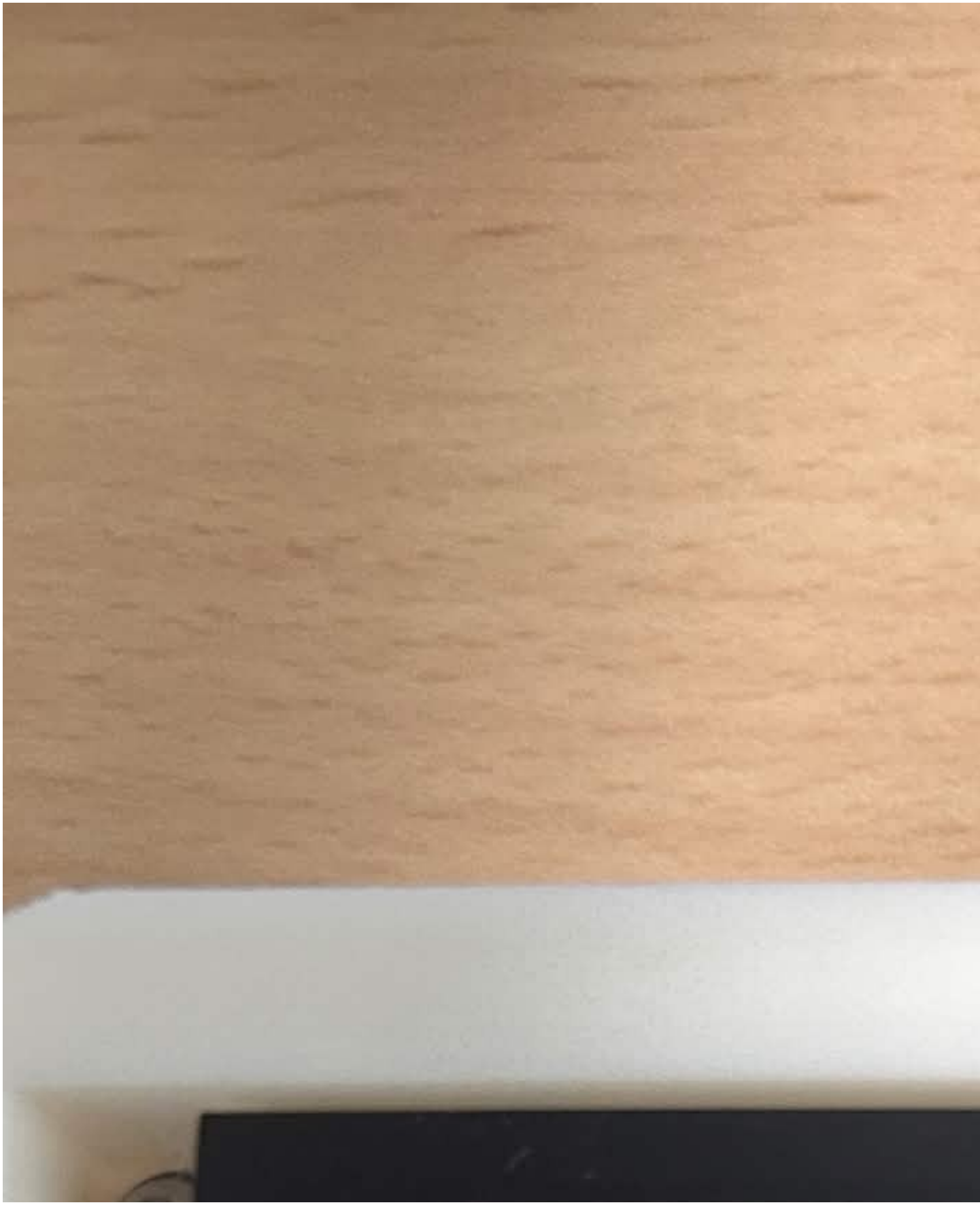


Figure 48 — Menu du jeu affiche sur l'ecran reel, conforme au rendu attendu apres integration complete.

18.4 Difficultes rencontrees

Difficultes mecaniques. Le principal point d'attention a porte sur l'ajustement des supports et l'encombrement : il a fallu loger ensemble la carte du PIC, la carte son (PCB de 45 mm x 45 mm) et l'ecran sans contrainte sur les connecteurs. *Cause* : plusieurs cartes a fixer dans un volume reduit. *Consequence* : risque de tension sur les fils et de mauvais maintien mecanique. *Solution* : adaptation des supports pour repartir les modules et degager les zones de cablage (cotes exactes a confirmer).

Difficultes electroniques. Le brasage de la carte son, entierement composee de composants traversants (THT) au fer a souder, a demande du soin. *Cause* : nombreuses soudures rapprochees autour du LM386 (boitier DIP-8) et des connecteurs. *Consequence* : risque de pont de soudure et de court-circuit, ainsi que de bruit audio. *Solution* : controle visuel systematique, recherche de courts-circuits et mesure de continuite apres brasage ; cote conception, la maitrise du bruit repose sur le decouplage de l'alimentation (C2 100 nF, C3 10 uF) et de la broche BYPASS du LM386 (C4 10 uF), ainsi que sur le reseau de Zobel (R2 100 Ohm + C5 47 nF) qui previent les oscillations en sortie, le condensateur C6 (220 uF) assurant la liaison vers le haut-parleur.

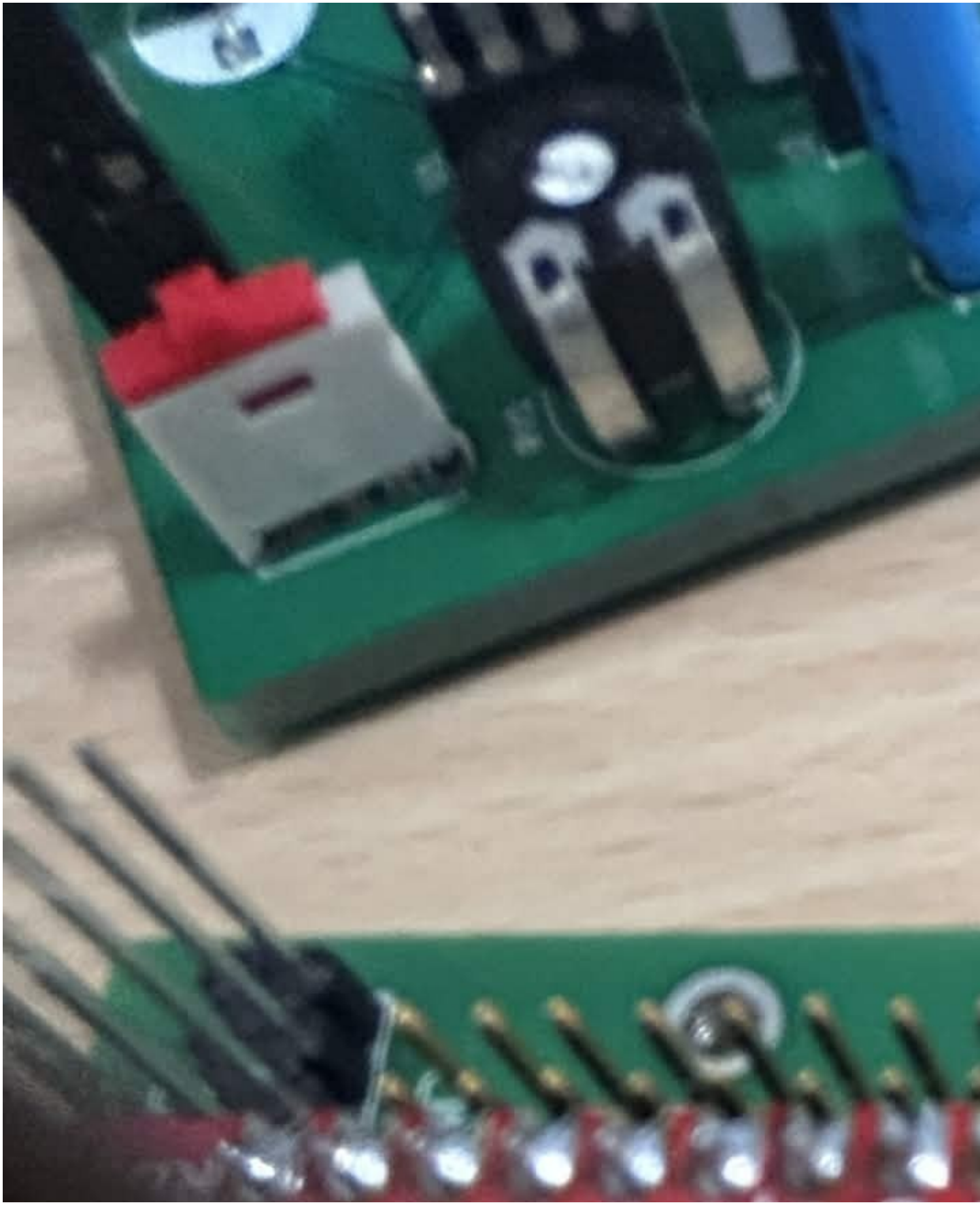


Figure 49 — Carte son raccordee lors des essais d'integration : verification des liaisons d'alimentation, d'entree et de sortie audio.

Difficultes logicielles. Plusieurs obstacles propres au PIC16F18877 ont oriente l'ecriture du code, dont le coeur 8 bits cadence a Tcy = 125 ns ne dispose d'aucune unite de calcul flottant materielle.

- *Calculs flottants lents* : depourvu de FPU, le PIC16 emule logiciellement `cos`, `sin`, `atan2` et `sqrtf`, operations couteuses en pleine boucle de jeu. *Consequence* : risque de ralentissement et de saccades. *Solution* : eviter ces fonctions en exploitant directement le vecteur normalise du joystick, dont les composantes valent deja (cos, sin) de l'angle vise (`shipCos`, `shipSin`), et comparer les distances au carre lors des collisions plutot que d'extraire une racine carree.
- *Clignotement de l'image* : tout redessiner a chaque tour faisait scintiller l'affichage. *Cause* : effacement integral de l'ecran a chaque trame. *Solution* : ne redessiner que ce qui change (effacer l'ancienne position, deplacer, redessiner), le seuil `ANGLE_REDRAW_DOT` (0,9997, soit environ 1,4 degres) evitant meme de retracer le vaisseau lorsqu'il n'a quasiment pas tourne.
- *Duree maximale de* `__delay_ms` : cette macro est bornee par la frequence d'horloge, ce qui interdit les longues temporisations en un seul appel. *Solution* : decouper les attentes (par exemple le compte a rebours) en plusieurs appels successifs plus courts.
- *Sens des axes du joystick* : selon le cablage, un axe pouvait etre inverse. *Consequence* : commande percue a l'envers. *Solution* : prevoir des reglages logiciels (`JOY_X_SIGN` pour le signe de l'axe X, et la prise en compte de l'orientation verticale du type `TOP_IS_LOW`) afin d'aligner le sens percu sur le sens attendu sans retoucher le materiel.

Difficultes d'integration. Relier trois sous-ensembles a impose un cablage propre. *Cause* : multiples liaisons (SPI vers l'ecran, audio de RA6 vers la carte son, alimentations +5 V). *Consequence* : risque de masses flottantes et de mauvais contacts. *Solution* : soin du cablage, mise en commun rigoureuse des masses (GND partages entre les cartes) et exploitation du fait que la liaison SPI vers l'ecran est a sens unique (PIC vers ecran, ligne MISO inutilisee par le programme), ce qui simplifie le raccordement et reduit les sources d'erreur.

19. Bilan, perspectives et conclusion

Ce dernier chapitre referme le rapport en prenant de la hauteur sur l'ensemble du travail mené. Après avoir détaillé, tout au long des chapitres précédents, chaque sous-ensemble matériel et chaque mécanisme logiciel, il s'agit ici de dresser le bilan global du projet, de confronter méthodiquement le résultat au cahier des charges, de rappeler le fil rouge technique qui a guidé tous les choix, puis d'ouvrir sur les améliorations envisageables. Le chapitre se conclut sur le retour personnel du binôme Emma Grosmaire et Dorian Sausse, du groupe C, au terme de cette SAE de deuxième semestre.

19.1 Bilan du projet : compétences mises en œuvre

La SAE avait pour ambition de couvrir l'intégralité de la chaîne de conception d'un système embarqué, depuis l'idée jusqu'au prototype démontrable. À ce titre, le projet a effectivement mobilisé un large éventail de compétences techniques, complémentaires les unes des autres.

La **conception mécanique** a été menée sous SolidWorks, pour modéliser le support et l'enveloppe destinés à accueillir l'écran, la manette et la carte son, et pour vérifier la cohérence dimensionnelle de l'assemblage (cotes exactes à confirmer). La **conception électronique** s'est appuyée sur KiCad 9 : tracé du schéma de la carte son autour de l'amplificateur LM386, routage du circuit imprimé double face de 45 mm × 45 mm en composants traversants (THT), puis génération des fichiers de fabrication (Gerbers : couches cuivre F_Cu et B_Cu, masques de soudure, sérigraphie, perçages PTH et NPTH).

La **réalisation et le brasage du PCB** ont prolongé naturellement cette phase : à la réception du circuit nu, le binôme a brasé au fer à souder les composants traversants, effectué un contrôle visuel des soudures, recherché d'éventuels courts-circuits et mesuré la continuité des pistes. Le **développement embarqué** a été conduit sous MPLAB X IDE avec le compilateur XC8 et l'outil MCC (MPLAB Code Configurator), depuis la configuration des périphériques du PIC16F18877 — convertisseur ADCC, bus SPI1 (MSSP1) et oscillateur numériquement commandé NCO1 — jusqu'à l'écriture de la logique de jeu, du moteur de rendu graphique et de la génération sonore.

Enfin, le **câblage et l'intégration** ont relié physiquement l'ensemble — écran ILI9488, manette à base de PIC, carte son et alimentation — et la **validation expérimentale** a consisté

à éprouver la jouabilité réelle, à corriger les défauts observés et à confirmer que matériel et logiciel coopèrent de façon fiable. Cette diversité de compétences, exercée sur un même objet cohérent, constitue sans doute l'apport le plus précieux du projet.

19.2 Objectifs atteints

Il convient à présent de reprendre point par point les objectifs fixés dans le cahier des charges et de vérifier qu'ils sont effectivement remplis.

- **Architecture matérielle complète.** Le système intègre les trois grands ensembles prévus — interface utilisateur (écran et manette), carte son dédiée et microcontrôleur jouant le rôle d'unité centrale — avec une répartition claire des fonctions sur les broches du PIC. Objectif atteint.
- **Carte son fonctionnelle.** L'amplificateur audio basse tension à base de LM386 (boîtier DIP-8, gain fixé à 20) a été conçu, fabriqué et câblé. Il reçoit le signal issu de la broche RA6 (sortie du NCO1, patte 14) sur l'entrée J2 (Audio_In), l'amplifie, et le restitue vers le haut-parleur via J3 (Audio_out), le tout réglable en volume par le potentiomètre RV1 de 10 kΩ. Objectif atteint.
- **Mise en œuvre du PIC16F18877.** Le microcontrôleur 8 bits, boîtier 40 broches PDIP, cadencé par l'oscillateur interne HFINTOSC à FOSC = 32 MHz (cycle d'instruction Tcy = 125 ns), a été configuré via MCC — ADCC 10 bits pour le joystick, SPI1 maître en mode 0 pour l'écran, NCO1 pour le son — et exécute l'intégralité du programme applicatif. Objectif atteint.
- **Interface écran et manette.** L'écran TFT ILI9488 480 × 320 en mode paysage (`setRotation(3)`) affiche le jeu en temps réel, tandis que la manette — joystick analogique deux axes (axe X sur RC7, axe Y sur RC6) et boutons-poussoirs actifs à l'état bas — assure la commande du vaisseau, le tir (bouton B, RD6) et la navigation entre écrans (bouton C, RD5). Objectif atteint.
- **Logiciel embarqué.** La logique de jeu, organisée autour d'une machine à états à quatre écrans (MENU, RÉGLAGES, JEU, GAME OVER), le moteur graphique reposant sur les trois niveaux de bibliothèques (pilote bas niveau `ili9488`, librairie graphique `GFX_Library`, code applicatif `main.c`) et la génération sonore fonctionnent de concert. Objectif atteint.
- **Prototype intégré.** L'ensemble forme un prototype fonctionnel, testé et démontrable,

où l'on joue effectivement à un *Asteroids* fluide : détruire un astéroïde rapporte +10 au score, tandis qu'une collision avec le vaisseau déclenche le GAME OVER. Objectif atteint.

L'ensemble des exigences du cahier des charges est donc satisfait : le projet aboutit non pas à une simple maquette de laboratoire, mais à un système cohérent et utilisable.

19.3 Le fil rouge technique : ménager un microcontrôleur lent

S'il fallait retenir une seule idée directrice de ce projet, ce serait celle-ci : tout le logiciel a été pensé pour tirer le meilleur parti d'un microcontrôleur modeste. Le PIC16F18877 ne possède aucune unité de calcul en virgule flottante matérielle ; chaque opération sur des réels (cosinus, sinus, racine carrée) est émulée par logiciel et donc coûteuse en cycles. Trois stratégies, déclinées tout au long du rapport, en découlent directement.

Éviter le flottant en cours de jeu. Plutôt que de calculer l'orientation du vaisseau par des fonctions trigonométriques, le programme exploite le fait que le vecteur normalisé du joystick vaut déjà (cos, sin). On dispose ainsi gratuitement du couple `shipCos`, `shipSin` sans appeler `atan2`, `cos` ni `sin`. De même, le demi-angle arrière du vaisseau est figé sous forme de constantes précalculées (COS245 = −0,7705 et SIN245 = 0,6374, soit le cosinus et le sinus de 2,45 rad), ce qui évite tout calcul trigonométrique répété à l'affichage. Cette approche illustre une règle d'or de l'embarqué : déplacer le calcul du temps réel vers le temps de compilation.

Ne redessiner que ce qui change. Le moteur de rendu n'efface jamais l'écran entier à chaque image. Il applique le principe effacer / déplacer / redessiner : un objet est repeint en noir à son ancienne position, puis dessiné à sa nouvelle position. Le seuil ANGLE_REDRAW_DOT = 0,9997 (un produit scalaire correspondant à un écart d'environ 1,4°) évite même de redessiner le vaisseau tant que son orientation n'a pas suffisamment varié. On économise ainsi un grand nombre d'écritures coûteuses sur le bus SPI, où chaque pixel coûte trois octets (couleur 18 bits).

Comparer les distances au carré. Pour la détection des collisions, comparer une distance à une somme de rayons exigerait une racine carrée. En élevant les deux membres au carré, on remplace cette opération coûteuse par de simples multiplications, sans rien perdre en exactitude — l'équivalence $d \leq r \Leftrightarrow d^2 \leq r^2$ étant rigoureusement vraie pour des grandeurs

positives. Concrètement, le test d'impact ne s'écrit pas avec `sqrtf` mais sous la forme suivante :

```
float dx = ax - bx;
float dy = ay - by;
float r  = AST_RADIUS + BULLET_RADIUS;
if (dx*dx + dy*dy <= r*r) {
    /* collision : asteroide touche */
}
```

Ligne par ligne : `dx` et `dy` calculent l'écart de coordonnées entre les deux objets (par exemple un astéroïde et un projectile) ; `r` est la somme des deux rayons, c'est-à-dire la distance en dessous de laquelle les deux disques se chevauchent ; enfin le test compare le carré de la distance `dx*dx + dy*dy` au carré du seuil `r*r`. On n'extrait jamais la racine carrée : trois multiplications et une comparaison suffisent. Ce compromis entre exactitude géométrique et coût de calcul résume à lui seul l'esprit du projet.

On retrouve la même philosophie côté son : le NCO1, dont l'accumulateur 20 bits est cadencé par FOSC/4 = 8 MHz, délivre une fréquence f = (8 000 000 × NCO1INC) / 2^{20} ≈ NCO1INC × 7,6294 Hz. Pour fixer un ton, le code écrit `NCO1INC = frequence >> 3`, c'est-à-dire une division par 8 réalisée par simple décalage de bits. Le facteur exact serait 0,1311, mais 1/8 = 0,125 en est une excellente approximation, instantanée pour le processeur — un nouvel exemple où l'on préfère un décalage entier à une multiplication flottante.

19.4 Perspectives d'amélioration

Le prototype étant fonctionnel, plusieurs axes d'amélioration peuvent être envisagés pour une version ultérieure.

- **Miniaturisation.** L'intégration mécanique gagnerait à être resserrée : boîtier plus compact, regroupement des cartes, soin apporté au cheminement des câbles afin d'obtenir un objet plus proche d'un produit fini.
- **Optimisation logicielle.** Le moteur de rendu et la boucle de jeu pourraient être affinés

davantage, par exemple en réduisant encore les zones redessinées ou en remplaçant certaines opérations flottantes résiduelles (positions et vitesses des projectiles et astéroïdes, aujourd'hui en `float`) par de l'arithmétique entière en virgule fixe.

- **Améliorations audio.** La génération sonore, aujourd'hui limitée à un canal unique piloté par le NCO1, pourrait évoluer vers plusieurs voix simultanées ou la lecture de courtes mélodies, pour enrichir l'ambiance sonore du jeu.
- **Nouvelles fonctionnalités de jeu.** Le gameplay pourrait s'étoffer : introduction de niveaux de difficulté croissante, système de vies, astéroïdes se fragmentant en plus petits morceaux lorsqu'ils sont touchés, ou encore sauvegarde du meilleur score pour ajouter un enjeu de progression.

Ces pistes confirment que l'architecture retenue est extensible et que le projet peut servir de base solide à des développements futurs.

19.5 Conclusion personnelle du binôme

Au terme de cette SAE, Emma Grosmaire et Dorian Sausse retiennent une expérience particulièrement formatrice. Le projet a eu le mérite de relier entre elles des compétences souvent abordées séparément en formation : la conception mécanique sous SolidWorks, l'électronique analogique (la carte son à LM386), la fabrication d'un circuit imprimé et la programmation embarquée se sont ici rejointes au service d'un même objectif concret et motivant. Contraints par les limites d'un microcontrôleur 8 bits dépourvu de calcul flottant matériel, ils ont compris de façon tangible pourquoi certaines optimisations — éviter le flottant, ne redessiner que l'essentiel, raisonner sur des distances au carré — ne sont pas de simples astuces mais des réflexes d'ingénieur.

Le travail en binôme a également constitué un apprentissage en soi : répartition des tâches selon les affinités, mise en commun régulière des avancées, partage des phases d'intégration et de test, et résolution conjointe des difficultés rencontrées. Voir le prototype fonctionner — un vaisseau qui pivote au centre de l'écran, des projectiles qui détruisent les astéroïdes, un score qui grimpe de dix en dix et un son qui accompagne l'action — a procuré une réelle satisfaction, à la hauteur de l'investissement consenti.

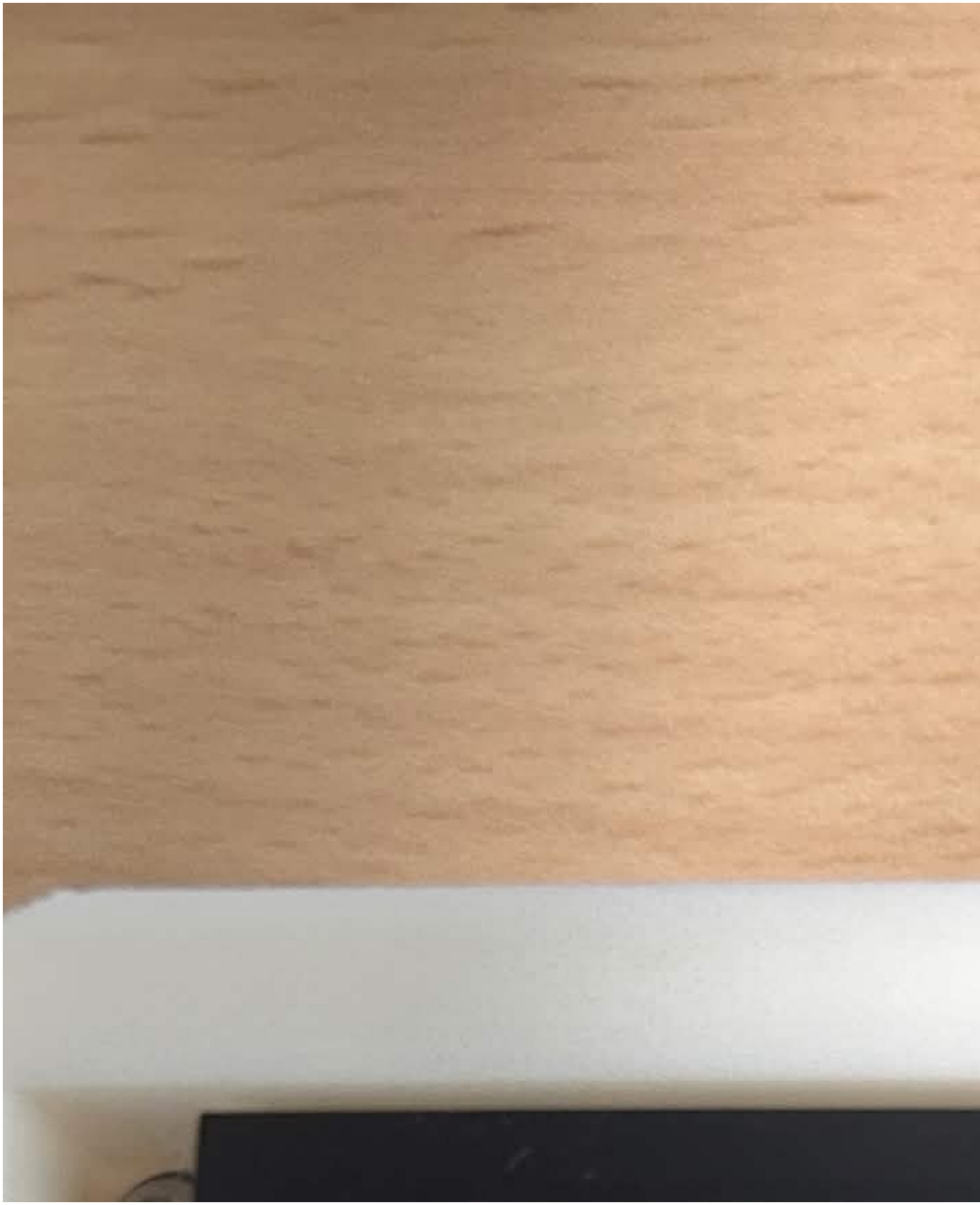


Figure 50 — le prototype du jeu Asteroids en fonctionnement, écran de menu sur l'écran ILI9488.