

Asteroids sur PIC16F18877 — Explication complète pour l'oral

Document de révision : chaque fonction expliquée, ce qu'elle fait et **pourquoi**. Matériel : microcontrôleur **PIC16F18877**, écran **ILI9488** (480×320) en **SPI**, joystick lu par l'**ADC**, son généré par le périphérique **NCO** sur une broche.

PARTIE 0 — Les rappels de C dont tu as besoin

Avant les fonctions, voici le vocabulaire qui revient partout. Si tu connais ces 8 points, tu comprends tout le reste.

- **#define NOM valeur** : ce n'est PAS une variable. Le compilateur remplace juste **NOM** par **valeur** dans tout le code, avant de compiler. Ça ne prend **aucune mémoire** et ça sert à donner des noms clairs aux nombres (ex : **SCREEN_W** au lieu d'écrire **480** partout).
- **uint8_t**, **uint16_t**, **int8_t** : des entiers de taille fixe. **uint8_t** = entier de 8 bits sans signe (0 à 255), **uint16_t** = 16 bits (0 à 65535), **int8_t** = 8 bits signé (−128 à 127). Sur un microcontrôleur on choisit la plus petite taille possible **pour économiser la mémoire** (le PIC en a très peu).
- **float** : un nombre à virgule. **Attention** : le PIC n'a pas de calculateur dédié aux virgules, donc les calculs en **float** (surtout cos, sin, racine) sont **lents**. Tout le programme cherche à les éviter.
- **static (devant une variable locale)** : la variable **garde sa valeur entre deux appels** de la fonction. Une variable locale normale est remise à zéro à chaque appel ; une **static** se souvient.
- **typedef enum { ... } Nom;** : crée une liste de noms (états) qui valent en réalité 0, 1, 2, 3... C'est juste plus lisible que des chiffres.
- **typedef struct { ... } Nom;** : un "paquet" qui regroupe plusieurs variables sous un seul nom (ex : un astéroïde a une position x, y et une vitesse vx, vy).
- **Le pointeur et la flèche ->** : **Asteroid *a** veut dire "a est l'adresse d'un astéroïde". Pour accéder à son champ x, on écrit **a->x**. Ça évite de recopier tout le paquet, on travaille directement sur l'original.
- **Les opérations rapides** : **>> 3** = décalage de bits = **diviser par 8**. **& 7** = **modulo 8** (reste de la division par 8). Ces opérations sont quasi instantanées pour le PIC, beaucoup plus que **/** ou **%**.

PARTIE 1 — Les **#define** et les variables globales

C'est le "réglage" du jeu. Tu n'as pas à tout citer, mais voici les importants.

Dimensions et repères - `SCREEN_W 480`, `SCREEN_H 320` : taille de l'écran. - `HUD_H 24` : hauteur de la bande du haut réservée au score (HUD = affichage tête haute). - `CX`, `CY` : le centre de la zone de jeu, là où le vaisseau reste fixe.

Réglages du jeu - `STAR_COUNT 18` : nombre d'étoiles du fond. - `MAX_BULLETS 4` : au maximum 4 tirs à l'écran en même temps. - `MAX_asteroids 4` : au maximum 4 astéroïdes. - `BULLET_SPEED`, `AST_SPEED` : vitesses. `SPAWN_DELAY 19` : un nouvel astéroïde tous les 19 tours de boucle. - `JOY_DEADZONE 32` : zone morte du joystick (on ignore les petits mouvements). - `ANGLE_REDRAW_DOT 0.9997f` : seuil pour redessiner le vaisseau seulement s'il a assez tourné (expliqué dans `lireJoystick`). - `COS245`, `SIN245` : le cosinus et le sinus de 2,45 radians, **précalculés une fois pour toutes** pour ne pas les recalculer en jeu.

Variables globales (accessibles par toutes les fonctions) : l'état courant `etat`, les tableaux `bullets[]` et `asteroids[]`, l'orientation du vaisseau (`shipCos`, `shipSin`), les sommets du vaisseau à l'écran (`shipX1` ...), le `score`, et les drapeaux des sons. Elles sont globales parce que **plusieurs fonctions doivent les partager**.

Question type du prof : « Pourquoi des `#define` plutôt que des nombres directement ? » → Pour la lisibilité et pour modifier un réglage à un seul endroit ; et ça ne consomme pas de mémoire.

PARTIE 2 — Les boutons : `appuiB`, `appuiC`, `appuiD`

Les trois fonctions sont identiques, juste un bouton différent. Exemple :

```
bool appuiB(void)
{
    static uint8_t ancien = 1;           // se souvient de l'état précédent
    uint8_t actuel = BTN_B_GetValue();   // lit le bouton maintenant (0 = appuyé)
    bool appui = (ancien == 1 && actuel == 0); // vrai SEULEMENT au moment où on enfonce
    ancien = actuel;                      // on mémorise pour le prochain tour
    return appui;
}
```

Ce que ça fait : détecte un **appui neuf** (le passage de "relâché" à "appuyé").

Pourquoi : le bouton est lu des centaines de fois par seconde. Si on testait juste "est-il appuyé ?", un seul appui du doigt déclencherait l'action des dizaines de fois. Ici, grâce à la variable `static ancien`, on ne renvoie `true` qu'**une seule fois par appui** : au front descendant (1 → 0). C'est ce qu'on appelle gérer le **front** plutôt que l'**état**.

Les boutons sont en logique inversée (**actifs à 0**) : 1 = relâché, 0 = appuyé. C'est classique avec une résistance de tirage (*pull-up*).

PARTIE 3 — Le son (NCO sur la broche)

Le périphérique **NCO** génère un signal carré sur une broche, qui fait du son sur un buzzer/haut-parleur. La variable `NC01INC` règle la **hauteur de la note** : plus elle est grande, plus le son est aigu. Le `>> 3` (diviser par 8) sert juste à convertir la fréquence voulue en valeur de réglage adaptée à leur horloge.

Il n'y a **qu'un seul canal de son**, donc une règle simple : **l'explosion est prioritaire sur le tir**.

`son_explosion()` et `son_tir()` — démarrer un son

Ces fonctions ne jouent PAS le son directement. Elles **arment** juste un son en mettant des drapeaux :

```
void son_explosion(void) {
    sfxTirActive = 0;           // l'explosion coupe un "piou" en cours
    sfxExplosionActive = 1;      // on active l'explosion
    sfxExplosionStep = 0;        // on repart du début
    sfxExplosionTimer = 0;
}
```

`son_tir()` fait pareil mais **refuse de démarrer si une explosion joue** (`if(sfxExplosionActive) return;`). C'est la règle de priorité.

`majSon()` — la fonction maligne (son non bloquant)

C'est le cœur du système son, et un **excellent point à expliquer**. Le problème : pour faire un son qui change de fréquence (un "boum" ou un "piou"), la solution naïve serait d'attendre entre chaque fréquence avec `__delay_ms`. Mais pendant une attente, **le jeu est figé** : le vaisseau ne bouge plus, les astéroïdes non plus. Inacceptable.

La solution : `majSon()` est une **petite machine à états** appelée à chaque tour de boucle. À chaque appel, elle fait juste **un tout petit bout de travail puis rend la main immédiatement** :

```
if(sfxExplosionActive) {
    if(sfxExplosionTimer > 0) { sfxExplosionTimer--; return; } // on patiente sans bloquer
    switch(sfxExplosionStep) {
        case 0: NC01INC = 650 >> 3; sfxExplosionTimer = 3; sfxExplosionStep++; break; //
grave-aigu
        case 1: NC01INC = 220 >> 3; sfxExplosionTimer = 4; sfxExplosionStep++; break;
        case 2: NC01INC = 70 >> 3; sfxExplosionTimer = 4; sfxExplosionStep++; break; //
très grave
        default: NC01INC = 0; sfxExplosionActive = 0; ... break; //
fin : silence
    }
    return;
}
```

Comment ça marche : au lieu d'attendre, on décrémente un compteur (`sfxExplosionTimer`) à chaque tour. Quand il atteint 0, on passe à l'étape suivante (`step`) qui charge une nouvelle fréquence. L'explosion descend de 650 Hz → 220 → 70 puis se coupe. Le tir ("piou") fait pareil mais plus vite : 1400 → 1050 → 750 → 520 puis silence.

Le point clé à dire : « le son ne bloque jamais le jeu, parce qu'à chaque appel la fonction fait une micro-étape et rend la main tout de suite. »

`son_gameover()` — le son de mort (lui bloque, et c'est normal)

```
for(int f = 950; f >= 120; f -= 22) { // descente continue type extinction
    NC01INC = f >> 3;
    __delay_ms(9);
}
NC01INC = 0; __delay_ms(70);
NC01INC = 150 >> 3; __delay_ms(150); // deux notes graves finales
NC01INC = 95 >> 3; __delay_ms(280);
NC01INC = 0;
```

Ici on utilise bien des `__delay_ms` (attentes bloquantes). **Pourquoi c'est permis ici alors qu'on l'a interdit ailleurs ?** Parce que **le jeu est terminé** : plus rien ne bouge, on peut donc se permettre de bloquer pour jouer un beau son de "power-down" (descente de 950 Hz à 120 Hz + deux notes graves).

À retenir : non bloquant *pendant* le jeu (`majSon`), bloquant accepté *quand le jeu est fini* (`son_gameover`).

PARTIE 4 — Le joystick (lu par l'ADC)

Le joystick donne deux tensions (X et Y) lues par le **convertisseur analogique-numérique (ADC)**, qui les transforme en nombres de 0 à 1023.

`calibrerJoystick()` — trouver le vrai centre

```
for(uint8_t i = 0; i < 12; i++) {
    ADCC_StartConversion(poussoir_x); while(!ADCC_IsConversionDone());
    sx += ADCC_GetConversionResult();
    ADCC_StartConversion(poussoir_y); while(!ADCC_IsConversionDone());
    sy += ADCC_GetConversionResult();
    __delay_ms(2);
}
joyCx = sx / 12; joyCy = sy / 12; // moyenne
```

Ce que ça fait : lit la position du joystick **au repos** 12 fois, et en fait la **moyenne**.

Pourquoi : au repos, un joystick ne renvoie jamais exactement la valeur du milieu (512). Il y a un petit décalage propre à chaque manette. On mesure donc son vrai centre (`joyCx` , `joyCy`) au démarrage de chaque partie, pour pouvoir ensuite calculer un écart juste. La moyenne sur 12 mesures **lisse le bruit** électrique. (`sx` est un `long` pour ne pas déborder en additionnant 12 valeurs.)

`lireJoystick()` — orienter le vaisseau SANS calculer d'angle

C'est **la fonction la plus importante du programme**. Elle décide où pointe le vaisseau et renvoie `true` s'il faut le redessiner.

```
int dx = (adcX - joyCx) * JOY_X_SIGN;    // écart par rapport au centre, en X
int dy = -(adcY - joyCy);                // idem en Y (signe selon le câblage)

long mag2 = (long)dx*dx + (long)dy*dy;
if(mag2 < JOY_DEADZONE*JOY_DEADZONE) return false;    // zone morte : on ignore
```

Étape 1 — la zone morte : on calcule la distance² du joystick à son centre. Si elle est trop petite, on ne fait rien (`return false`). Ça évite que le vaisseau tremble tout seul à cause du bruit quand on ne touche pas la manette.

```
float mag = sqrtf(fx*fx + fy*fy);    // longueur du vecteur
float newCos = fx / mag;              // direction normalisée...
float newSin = fy / mag;              // ...c'est DÉJÀ (cos, sin) !
```

Étape 2 — l'astuce géniale : on prend le vecteur (dx, dy) et on le divise par sa longueur. On obtient un vecteur de longueur 1. **Or, un vecteur de longueur 1 dans une direction, c'est exactement (cos θ, sin θ) de l'angle de cette direction.** Donc on n'a **jamais besoin de calculer l'angle** avec `atan2`, ni de repasser par `cos` / `sin` : la direction normalisée **EST** le cosinus et le sinus dont on a besoin. C'est ça qui évite des calculs lents au PIC.

```
float dot = newCos*shipCos + newSin*shipSin;    // produit scalaire avec l'ancienne direction
shipCos = newCos; shipSin = newSin;
return (dot < ANGLE_REDRAW_DOT);                // a-t-on assez tourné ?
```

Étape 3 — ne redessiner que si nécessaire : on compare la nouvelle direction à l'ancienne avec un **produit scalaire**. Si les deux directions sont presque identiques, le produit scalaire vaut presque 1 (`0.9997` $\approx 1,4^\circ$). Dans ce cas on renvoie `false` : inutile de gaspiller du temps à effacer et redessiner le vaisseau pour un déplacement minuscule. On ne le redessine que s'il a **vraiment** tourné.

Trois questions type du prof, trois réponses : 1. *Zone morte ?* → ignorer le bruit du joystick au repos. 2. *Pourquoi pas atan2/cos/sin ?* → pas de calculateur de virgule rapide, donc on utilise directement le vecteur normalisé qui vaut (cos, sin). 3. *Le produit scalaire ?* → mesurer si l'orientation a assez changé pour mériter un redessin.

PARTIE 5 — Le vaisseau

Le vaisseau est un triangle qui reste **au centre** de l'écran et ne fait que **tourner**.

calculerSommetsVaisseau() — les 3 coins, toujours sans cos / sin

```
shipX1 = CX + (int)(shipCos * SHIP_FRONT); // pointe avant = centre + direction × longueur
shipY1 = CY + (int)(shipSin * SHIP_FRONT);

float bx = shipCos*COS245 - shipSin*SIN245; // direction tournée de +2,45 rad
float by = shipSin*COS245 + shipCos*SIN245;
shipX2 = CX + (int)(bx * SHIP_BACK); // coin arrière gauche

float cx = shipCos*COS245 + shipSin*SIN245; // direction tournée de -2,45 rad
float cy = shipSin*COS245 - shipCos*SIN245;
shipX3 = CX + (int)(cx * SHIP_BACK); // coin arrière droit
```

La **pointe avant** est simple : centre + (cos, sin) × longueur.

Les deux coins arrière : il faut prendre la direction du vaisseau et la **faire pivoter de ±2,45 radians** (pour écarter les deux ailes). La formule de rotation d'un vecteur (c, s) d'un angle α est : $-x' = c \cdot \cos \alpha - s \cdot \sin \alpha$ - $y' = s \cdot \cos \alpha + c \cdot \sin \alpha$

Comme l'angle $\alpha = 2,45$ est **toujours le même**, on a précalculé `COS245` et `SIN245` une fois pour toutes (dans les `#define`). Résultat : on tourne le vaisseau **sans jamais appeler cos ni sin pendant le jeu**, juste avec des multiplications et additions — ce que le PIC fait vite. C'est le prolongement direct de l'astuce de `lireJoystick`.

tracerVaisseau(uint16_t c) — dessiner le triangle

```
display_drawLine(shipX1, shipY1, shipX2, shipY2, c);
display_drawLine(shipX2, shipY2, shipX3, shipY3, c);
display_drawLine(shipX3, shipY3, shipX1, shipY1, c);
```

Trois traits entre les trois coins déjà calculés. **Aucun calcul à virgule ici** : les coins sont déjà connus. Le paramètre `c` est la couleur : on passe **orange** pour afficher le vaisseau, ou **noir** pour l'effacer (dessiner par-dessus en couleur du fond).

L'idée à expliquer : on sépare le **calcul** des coins (`calculerSommetsVaisseau`) et le **dessin** (`tracerVaisseau`). Ça permet de redessiner vite sans recalculer, et d'effacer/réafficher facilement.

PARTIE 6 — L'affichage des écrans

dessinerFondEtoiles()

Boucle sur les 18 étoiles. Une étoile sur trois est orange, les autres blanches ; une sur sept reçoit un pixel blanc en plus à côté (pour varier la taille). C'est purement décoratif. Les positions sont fixées dans les tableaux `starX[]` et `starY[]`.

dessinerPetitVaisseauMenu(cx, cy, couleur)

Dessine un petit vaisseau (4 traits formant une forme de fusée avec une encoche en bas) à une position donnée. Sert juste de décoration au centre du menu.

dessinerBoutonMenu(...)

Dessine un bouton stylé (un cadre double + du texte au centre).

Honnêteté : dans le code actuel, les écrans dessinent les options "à la main" (avec `display_printText`) et **n'appellent pas** cette fonction. Elle est donc **présente mais inutilisée**. Si le prof le remarque, tu peux dire que c'est une fonction prévue mais finalement non utilisée — autant le savoir.

afficherMenu()

Efface l'écran en noir, dessine les étoiles, puis un **cadre orange épais** (boucle de 4 rectangles imbriqués pour faire une bordure de 4 pixels), le titre « ASTEROIDS », les crédits, le petit vaisseau, et les deux choix « C → JOUER » et « D → REGLAGES ».

afficherReglages()

Même principe : étoiles, cadre, titre « COMMANDES », une ligne de séparation, les commandes (« B → TIR », « JOYSTICK → VISER ») et « C → RETOUR ».

afficherGameOver()

Écran de fin : cadre, « GAME OVER », ligne, puis le score. Utilise `sprintf(buf, "SCORE : %d", score)` qui **fabrique un texte** à partir du nombre `score` (`%d` est remplacé par la valeur). Puis « C → MENU ».

afficherScore() — encore une optimisation

```
if(score == oldScore) return; // rien à faire si le score n'a pas changé
oldScore = score;
display_fillRect(82, 4, 65, 18, ILI9488_BLACK); // efface l'ancien nombre
sprintf(buf, "%d", score);
display_printText(buf, 82, 4, COL_ORANGE, ILI9488_BLACK, 2); // écrit le nouveau
```

Pourquoi : réécrire le score à chaque image ferait clignoter l'affichage et gaspillerait du temps. On ne le redessine **que s'il a changé** (on compare avec `oldScore`). Même esprit que partout : *ne redessiner que ce qui bouge*.

PARTIE 7 — `compteARebours()` : le décompte de départ

```
for(int s = 4; s >= 1; s--) {
    display_fillRect(216, 60, 48, 56, ILI9488_BLACK); // efface le chiffre précédent
    sprintf(b, "%d", s);
    display_printText(b, 225, 64, COL_ORANGE, ILI9488_BLACK, 6); // gros chiffre
    for(int k = 0; k < 10; k++) __delay_ms(100); // attend 1 seconde
}
display_fillRect(216, 60, 48, 56, ILI9488_BLACK); // efface le dernier
```

Ce que ça fait : affiche 4, 3, 2, 1 (un chiffre par seconde) avant que les astéroïdes arrivent, pour laisser le joueur se préparer.

Deux détails à connaître : - L'attente est bloquante, mais **c'est voulu** : avant le début du jeu, rien d'autre n'a besoin de tourner. - On attend 1 s en faisant **10 × 100 ms** au lieu d'un seul `__delay_ms(1000)`. C'est parce que `__delay_ms` a une **durée maximale** sur PIC ; on découpe donc en petits délais.

PARTIE 8 — `initJeu()` : préparer une nouvelle partie

C'est la "remise à zéro" complète avant de jouer. Dans l'ordre, elle : 1. efface l'écran et redessine les étoiles ; 2. remet `score = 0` et affiche « Score: » ; 3. remet à zéro tous les compteurs (cooldown de tir, compteur de spawn, etc.) ; 4. coupe et réinitialise tous les sons (`NC01INC = 0`) ; 5. remet l'orientation du vaisseau **vers le haut** (`shipCos = 0`, `shipSin = -1`, ce qui correspond à l'angle $-\pi/2$) ; 6. **désactive** tous les tirs et tous les astéroïdes (`active = 0`) pour repartir propre ; 7. **calibre le joystick** (`calibrerJoystick`) ; 8. affiche le score, dessine le vaisseau ; 9. lance le **compte à rebours**.

Pourquoi tout réinitialiser : si on relance une partie après un game over, il ne faut pas garder l'ancien score, ni d'anciens astéroïdes/tirs à l'écran. On repart d'un état parfaitement propre.

PARTIE 9 — Les tirs

`tirer()`

```
for(uint8_t i = 0; i < MAX_BULLETS; i++) {
    if(!bullets[i].active) {                // on cherche un tir libre
        bullets[i].x = CX + shipCos * SHIP_FRONT; // part de la pointe du vaisseau
        bullets[i].y = CY + shipSin * SHIP_FRONT;
        bullets[i].vx = shipCos * BULLET_SPEED; // vitesse dans la direction visée
        bullets[i].vy = shipSin * BULLET_SPEED;
        bullets[i].life = BULLET_LIFE;
        bullets[i].active = 1;
        display_fillCircle(...);           // dessine le tir
        son_tir();                          // "piou"
        break;                             // un seul tir par appui
    }
}
```

Ce que ça fait : cherche un **emplacement de tir libre** dans le tableau (parmi les 4), le place à la pointe du vaisseau, et lui donne une vitesse dans la direction visée (réutilise `shipCos / shipSin`). `life = BULLET_LIFE` limite sa durée de vie. Le `break` garantit **un seul tir par appui**.

Pourquoi un tableau de taille fixe : pas d'allocation dynamique (lente et risquée sur microcontrôleur) ; on réutilise toujours les mêmes 4 cases en les activant/désactivant.

majTirs()

Pour chaque tir actif : on l'**efface** (le redessine en noir), on le **déplace** (`x += vx`), on **diminue sa durée de vie**, on vérifie s'il sort de l'écran ou si sa vie atteint 0 (→ on le désactive), sinon on le **redessine** en orange à sa nouvelle position. C'est la technique "effacer → déplacer → redessiner" utilisée pour tout ce qui bouge.

PARTIE 10 — Les astéroïdes

spawnAsteroïde() — en faire apparaître un

```
uint8_t bord = rand() % 4; // choisit un bord au hasard : 0=haut 1=bas 2=gauche 3=droite
// ... place l'astéroïde sur ce bord, à une position aléatoire ...

float dx = CX - asteroids[i].x; // direction vers le CENTRE
float dy = CY - asteroids[i].y;
float d = sqrtf(dx*dx + dy*dy);
asteroids[i].vx = (dx / d) * AST_SPEED; // vitesse vers le centre
asteroids[i].vy = (dy / d) * AST_SPEED;
```

Ce que ça fait : cherche un emplacement libre, choisit un **bord au hasard** (`rand() % 4`), y place l'astéroïde, puis calcule sa vitesse pour qu'il **fonce vers le centre** (là où est le vaisseau). On normalise le vecteur (dx, dy) pour que tous les astéroïdes aillent à la même vitesse quelle que soit leur distance.

Ici on utilise bien `sqrtf` (racine), mais **une seule fois, au moment de l'apparition** (rare), pas à chaque image. C'est acceptable.

dessinerAsteroïde(Asteroid *a, couleur) — dessiner un octogone

```
for(uint8_t k = 0; k < 8; k++) {
    px[k] = (int)a->x + (AST_RADIUS * astX[k]) / 10; // 8 coins, forme précalculée
    py[k] = (int)a->y + (AST_RADIUS * astY[k]) / 10;
}
for(uint8_t k = 0; k < 8; k++) {
    uint8_t j = (k + 1) & 7; // (k+1) modulo 8 → relie 7 à 0
    display_drawLine(px[k], py[k], px[j], py[j], c);
}
```

Ce que ça fait : dessine un astéroïde en forme d'**octogone** (8 côtés). La forme de base est stockée dans `astX[] / astY[]` (8 points), on la **met juste à l'échelle** (`× AST_RADIUS / 10`) et on la décale à la position de l'astéroïde.

Deux points malins : - On passe un **pointeur** (`Asteroid *a`) pour ne pas recopier tout le paquet ; on lit ses champs avec `a->x` . - `(k + 1) & 7` est une astuce : `& 7` = modulo 8. Ça permet de relier le dernier point (7) au premier (0) pour fermer la forme, sans `if` spécial.

majAsteroides()

Pour chaque astéroïde actif : on l'**efface** (noir), on le **déplace**, on vérifie s'il est sorti **très** loin de l'écran (→ on le désactive), sinon on le **redessine** (blanc). Même schéma "effacer → déplacer → redessiner".

PARTIE 11 — Les collisions

collision(x1,y1,r1, x2,y2,r2) — la brique de base

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

Ce que ça fait : teste si deux **cercles** se touchent. Deux cercles se chevauchent si la distance entre leurs centres est inférieure à la somme de leurs rayons.

L'optimisation clé : on compare les **carrés** (`dx*dx + dy*dy` contre `r*r`) au lieu de calculer la vraie distance avec une racine carrée. Comparer les carrés donne **exactement le même résultat** que comparer les distances, mais **sans sqrt** (lente). À dire absolument à l'oral.

collisions() — appliquer les règles

Deux blocs :

Bloc 1 — tir contre astéroïde (double boucle : chaque tir × chaque astéroïde) :

```
if(collision(bullet, asteroid)) {
    // efface les deux, les désactive
    score += 10;
    son_explosion();
}
```

Si un tir touche un astéroïde : on efface et désactive les deux, on **ajoute 10 au score**, et on joue le son d'explosion.

Bloc 2 — vaisseau contre astéroïde :

```
if(collision(CX, CY, SHIP_RADIUS, asteroid)) {
    son_gameover();
    afficherGameOver();
    etat = ETAT_GAMEOVER; // on change d'état → fin de partie
}
```

Si un astéroïde touche le vaisseau (toujours au centre) : son de mort, écran de game over, et on **bascule la machine à états** en `ETAT_GAMEOVER`. C'est le lien entre le jeu et l'écran de fin.

PARTIE 12 — `boucleJeu()` : le chef d'orchestre

Appelée en boucle pendant le jeu. Elle enchaîne, dans l'ordre, **un cycle complet** :

```
afficherScore();           // 1. met à jour le score si besoin
majSon();                  // 2. fait avancer le son (micro-étape)

bool angleChange = lireJoystick(); // 3. lit le joystick
if(angleChange) {          // si on a assez tourné :
    tracerVaisseau(ILI9488_BLACK); // efface l'ancien vaisseau
    calculerSommetsVaisseau();    // recalcule les coins
    tracerVaisseau(COL_ORANGE);  // dessine le nouveau
}

if(shootCooldown > 0) shootCooldown--; // 4. délai entre deux tirs
if(appuiB() && shootCooldown == 0) {   // si bouton tir + délai écoulé
    tirer(); shootCooldown = 3;
}

spawnCounter++;           // 5. compteur d'apparition
if(spawnCounter >= SPAWN_DELAY) { spawnCounter = 0; spawnAsteroide(); }

majAsteroïdes();          // 6. déplace les astéroïdes
majTirs();                // 7. déplace les tirs

shipDrawCounter++;        // 8. toutes les 12 images...
if(shipDrawCounter >= 12) { // ...on redessine le vaisseau
    tracerVaisseau(COL_ORANGE); // pour réparer les pixels effacés
    shipDrawCounter = 0;        // par les astéroïdes qui passent dessus
}

collisions();             // 9. teste toutes les collisions
majSon();                 // 10. re-fait avancer le son
```

Les choses importantes à expliquer : - C'est une **séquence fixe** répétée très vite : lire les entrées → bouger les objets → tester les collisions → recommencer. C'est la structure de **toute boucle de jeu**. - Le `shootCooldown` empêche de tirer trop vite (un petit délai forcé entre deux tirs). - Le bloc `shipDrawCounter` toutes les 12 images : comme les astéroïdes passent parfois sur le vaisseau et effacent une partie de ses pixels, on le **redessine régulièrement** pour le "réparer". - `majSon()` est appelée **deux fois** (début et fin) pour faire avancer le son un peu plus souvent → un son plus fluide.

PARTIE 13 — `main()` : le point d'entrée

```
void main(void) {
    SYSTEM_Initialize(); // configure l'horloge, les broches, les périphériques (MCC)
    ADCC_Initialize();   // prépare le convertisseur ADC (joystick)
    SPI1_Open(SPI1_DEFAULT); // ouvre la liaison SPI vers l'écran
    ILI9488_Init();      // initialise l'écran
    setRotation(3);      // met l'écran en mode paysage

    afficherMenu();      // on commence sur le menu

    while(1) {           // boucle infinie : la machine à états
        if(etat == ETAT_MENU) { /* appui C → jouer, appui D → réglages */ }
        else if(etat == ETAT_REGLAGES) { /* appui C → retour menu */ }
        else if(etat == ETAT_JEU) { boucleJeu(); }
        else if(etat == ETAT_GAMEOVER) { /* appui C → retour menu */ }
    }
}
```

Ce que ça fait : 1. **Initialise le matériel** : l'horloge et les broches (`SYSTEM_Initialize` , généré par MCC), l'ADC pour le joystick, le SPI et l'écran ILI9488 (mode paysage avec `setRotation(3)`). 2. Affiche le menu. 3. Entre dans la **boucle infinie** `while(1)` qui ne s'arrête jamais : à chaque tour, elle regarde dans quel **état** on est et agit. Dans le menu/réglages/ game over, elle attend juste un appui de bouton pour changer d'état. Dans le jeu, elle appelle `boucleJeu()` .

C'est le squelette de tout programme sur microcontrôleur : **une initialisation, puis une boucle infinie**. Le PIC ne s'arrête jamais ; il n'y a pas de "fin de programme".

LE FIL ROUGE À RETENIR (ta conclusion d'oral)

Si tu ne devais retenir qu'**une idée**, c'est celle-ci :

Tout le programme est conçu pour ménager un microcontrôleur lent et limité.

Trois exemples concrets, et tu as gagné :

1. **Éviter les calculs à virgule** : pas de `cos` / `sin` / `atan2` en jeu. On utilise directement le vecteur normalisé du joystick (qui vaut déjà (cos, sin)) et des constantes précalculées pour tourner le vaisseau.
2. **Ne jamais bloquer le jeu** : les sons sont gérés par une machine à états (`majSon`) qui fait des micro-étapes, au lieu d'attendre.
3. **Ne redessiner que ce qui change** : on efface/redessine seulement les objets qui bougent (jamais tout l'écran), le vaisseau seulement s'il a assez tourné, le score seulement s'il a changé. Et on compare des distances **au carré** pour éviter les racines carrées.

Et la **structure générale** : une machine à états (menu / réglages / jeu / game over) pilotée par une boucle infinie, avec une boucle de jeu qui répète sans cesse "lire les entrées → bouger → tester les collisions".

Conseil : entraîne-toi à dire à voix haute la partie 2, la fonction `lireJoystick` et le fil rouge. Ce sont les trois morceaux qui font la différence à l'oral.