Rapport de projet — *deepdiep*

Antoine CARREZ Antoine PASCAL

Février 2026

Section 3.1 — Prise en main et premières modifications

Question 1.1 — Balles détruites en sortie d'écran

La vérification est centralisée dans `Entity::update()` : toute entité dont la position dépasse les limites du monde est détruite via `kill()`.

```
if (position_.x < -radius_ || position_.x > world_->getWidth()
 || position_.y < -radius_ || position_.y > world_->getHeight()) {
    kill();
}
```

Comme `Bullet` hérite d'`Entity`, ce comportement s'applique automatiquement sans duplication de code.

Question 1.2 — Joueur stoppé aux bords

À la fin de `Tank::update()`, la position est contrainte pour que le tank reste dans les limites du monde en tenant compte de son rayon :

```
if (position_.x - radius_ < 0.f)
    position_.x = radius_;
else if (position_.x + radius_ > world_->getWidth())
    position_.x = world_->getWidth() - radius_;
// idem pour y
```

Le clamping est appliqué *après* le calcul du déplacement (mouvement + recul) afin de ne pas fausser la physique.

Question 1.3 — Base sûre

Lors de la génération du monde dans `World::World()`, chaque astéroïde candidat est rejeté si sa distance au tank initial est inférieure à $4r_{\text{tank}}$:

```
if ((pX-tank->getPosition().x)*(pX-tank->getPosition().x)
  + (pY-tank->getPosition().y)*(pY-tank->getPosition().y)
  < tank->getRadius() * tank->getRadius() * 16) {
    i--;  // rejeter ce candidat
}
```

Nous avons choisi une zone circulaire de rayon $4r$ centrée sur le spawn plutôt qu'une zone rectangulaire en haut à gauche, car cette approche est plus générique : elle s'adapte sans modification au mécanisme de respawn (`spawnPlayer`), qui fait appel à la même logique.

Question 1.4 — Cooldown — cadence de tir

Le champ `reload_time_` (en ticks) est ajouté à `Tank`. La méthode `fire()` compare `world->getTick() - last_fire_` à `reload_time_` avant de créer un projectile, puis met à jour `last_fire_`. La valeur par défaut est 15 ticks ; les sous-classes la surchargent (MachineGun : 5, Destroyer : 60).

Question 1.5 — Recul des armes

Un champ `impulsion_` (`sf::Vector2f`) est ajouté à `Tank`. À chaque tir, une impulsion opposée à la direction du canon est appliquée :

```
impulsion_.x -= cos(angle * M_PI / 180.f) / inertia_;
impulsion_.y -= sin(angle * M_PI / 180.f) / inertia_;
```

L'impulsion est plafonnée à `speed_/2` puis amortie à chaque tick par décroissance linéaire (`inertia_/6`). Elle s'additionne à `move_` lors de la mise à jour de position, ce qui permet de se propulser en tirant derrière soi.

Question 1.6 — Score

Le champ `xp_` existe déjà dans `Entity`. Quand une entité est détruite, l'XP est transmis au tireur via le parent de la balle :

```
Bullet* b = dynamic_cast<Bullet*>(other.get());
if (b) b->getParent()->addXp(xp_);
```

Les carrés jaunes (`AsteroidSquare`) rapportent 10 XP. L'affichage est une barre de progression verte en bas de chaque vue, normalisée par `goalScore`. Lorsque le seuil est atteint, un canon supplémentaire est ajouté et le seuil est multiplié par 1,5.

Question 1.7 — Accélérations et frottement (optionnel)

Plutôt que de modifier directement la position, le mouvement passe par le vecteur `move_` hérité d'`Entity`. L'entrée incrémente `move_` proportionnellement à `inertia_` (accélération) avec un plafond à `speed_` ; en l'absence d'entrée `move_` décroît de `inertia_` par frame (frottement). Le vecteur `impulsion_` est séparé de `move_`, les deux effets se superposant indépendamment.

Section 3.2 — Plus de tanks

Question 2.1 — Tanks aux changements quantitatifs

Nous avons implémenté deux sous-classes de `Tank` en surchargeant uniquement les champs quantitatifs dans leur constructeur :

- **MachineGun** : `reload_time_` = 5 (tir rapide), `damage_` réduit à 4,9, balles plus petites (`bullet_size_` /= 1,5), inertie élevée (1,1) pour une meilleure maniabilité.
- **Destroyer** : `reload_time_` = 60 (tir lent), `damage_` = 21, balles deux fois plus grandes, `bullet_speed_` réduite, inertie faible (0,3) pour un tank lourd.

```
MachineGun::MachineGun(World* world, sf::Vector2f pos, int team)
    : Tank(world, pos, team) {
    damage_ = 4.9f;  reload_time_ = 5;
    bullet_size_ /= 1.5f;  setInertia(1.1f);
}
Destroyer::Destroyer(World* world, sf::Vector2f pos, int team)
    : Tank(world, pos, team) {
    damage_ = 21;  reload_time_ = 60;
    bullet_size_ *= 2;  bullet_speed_ /= 1.5f;  setInertia(0.3f);
}
```

L'organisation retenue est l'héritage simple : chaque sous-classe ne redéfinit que les paramètres qui changent, sans dupliquer la logique de déplacement, de tir ou de recul. Cette approche est justifiée car les changements sont purement quantitatifs ; si des comportements qualitativement différents étaient nécessaires, la composition serait préférable. La sélection du tank (base/MachineGun/Destroyer) est proposée au joueur via le menu de démarrage.

Question 2.2 — Tanks aux changements qualitatifs

Le changement qualitatif implémenté est le **système multi-canons**. Lorsqu'un tank accumule suffisamment d'XP, un canon supplémentaire lui est ajouté via `addCanon()`. Le comportement de tir change alors qualitativement : les n canons tirent simultanément à des angles régulièrement répartis ($360°/n$ entre chaque), formant une gerbe circulaire.

```
for (int i = 0; i < cNumber; i++) {
    float angle = getAngle() + 360*i/cNumber;
    world_->push(new Bullet(..., angle, ...));
}
```

Le rendu est cohérent : chaque canon est dessiné à la bonne orientation autour du tank.

```
for (int i = 0; i < cNumber; i++) {
    shape->setRotation(s->getAngle() + 360/cNumber*i);
    window_->draw(*shape);
}
```

Ce changement est qualitatif car il modifie la nature de l'attaque : un tank à 3 canons ne peut plus cibler précisément une cible, il couvre au contraire 360° et se comporte comme un tank défensif, similaire au *Tri-Angle* ou au *Triplet* du jeu original.

Question 2.3 — Nouvelles formes de polygones

Nous avons ajouté quatre types d'astéroïdes supplémentaires couvrant les polygones de 3 à 6 côtés, déjà présents dans le code de base : `AsteroidTriangle` (rouge, 25 XP), `AsteroidPentagon` (bleu, 130 XP) et `AsteroidHexagon` (cyan, 1 500 XP). Les formes sont définies une fois dans `View` par composition, et les entités y font référence — conformément au pattern déjà utilisé pour les tanks. Les objets gris incassables et rebondissants n'ont pas été implémentés.

Section 3.3 — Améliorations graphiques

Question 3.1 — Zoom de la vue

L'infrastructure de zoom est en place : chaque tank possède un champ `zoom_` (initialisé à 1,0) et la taille de la vue SFML est calculée en divisant les dimensions de la fenêtre par ce facteur. `World::getCameraPos()` tient compte du zoom pour rester dans les bords du monde. Il suffit d'exposer un `setZoom()` et de le brancher sur `sf::Event::MouseWheelScrolled` pour un zoom interactif complet ; cette liaison n'a pas encore été branchée.

Question 3.2 — Sans souris (optionnelle)

Deux modes de visée sans souris sont implémentés, configurables indépendamment par joueur via la structure `KeySet`.

Mode clavier (`AKeys`). Un curseur virtuel `keyMousePos_` est maintenu dans le `KeySet`. Il se déplace avec les touches de visée (`Z/Q/S/D` par défaut), est plafonné à un rayon de $4r$ autour du tank, et est affiché à l'écran par un disque rouge :

```
if (tank->getKeyset()->aim_type == KeySet::Aim::AKeys) {
    target.setPosition(tank->getPosition()
                       + (sf::Vector2f)tank->getKeyset()->keyMousePos_);
    window_->draw(target);
}
```

Mode manette (`AJoystick`). `GameCmd` lit directement les axes du joystick (axes R et U pour la visée par défaut) via `sf::Joystick::getAxisPosition()`. Le numéro de manette et les axes sont configurables par joueur dans son `KeySet`.

```
if (ks->aim_type == KeySet::Aim::AJoystick)
    return sf::Vector2i(
        sf::Joystick::getAxisPosition(ks->joystick, ks->haim_joy),
        sf::Joystick::getAxisPosition(ks->joystick, ks->vaim_joy)
    );
```

Question 3.3 — Multi-joueurs en écran partagé

Architecture. L'écran partagé repose sur deux mécanismes découplés : la gestion des viewports (classe `Layout`) et la gestion des entrées (classe `KeySet` par joueur).

Viewports. `Layout` pré-calcule 7 configurations de viewports SFML couvrant 1 à 4 joueurs. Lors de l'ajout d'un joueur, `addPlayerToLayout()` réaffecte les viewports à l'ensemble des joueurs pour que leur partition reste cohérente :

```
void Layout::addPlayerToLayout(std::shared_ptr<Tank> player) {
    players_.push_back(player);
    int n = players_.size();
    for (int i = 0; i < n; i++)
        players_[i]->setViewport(views_[n - 1 + i]);
}
```

Entrées. Un unique objet `GameCmd` est créé par frame et capture l'état global des entrées. Chaque tank possède son propre `KeySet*` ; lors de `tank->update(&cmd)`, il filtre `GameCmd` avec son `KeySet` pour n'en extraire que ses commandes. Ce design est extensible : ajouter un joueur revient à instancier un tank avec un `KeySet` distinct et à appeler `addPlayerToLayout()`.

Fonctionnalités supplémentaires

Menu et configuration

Nous avons ajouté un système de menus hiérarchiques navigables au clavier (flèches + Entrée) et à la souris. Les items sont affichés avec un fond semi-transparent, mis en évidence en vert au survol. Cinq menus distincts couvrent l'intégralité du flux de jeu :

- **Menu principal** : lancer la partie, accéder aux options, quitter.
- **Options** : ajouter ou retirer un joueur (1 à 4), accéder au configurateur de touches.
- **Sélection du joueur** : choisir quel joueur on va reconfigurer.
- **Configuration des touches** : modifier chaque touche de déplacement et de visée, basculer entre les modes `Keyboard` / `Joystick` pour le déplacement et `Mouse` / `Keyboard` / `Joystick` pour la visée.
- **Fin de partie** : rejouer (en conservant la configuration), retourner aux options, quitter.

La configuration se fait en cliquant sur le binding voulu : le menu disparaît, la boucle d'événements attend le prochain `sf::Event::KeyPressed` et met à jour le `KeySet` du joueur sélectionné. L'affichage est rechargé via `updateMenuFromKeySet()`. Les configurations (touches, mode de visée, équipe) sont persistées entre les parties : `startGame()` transfère les `KeySet` des tanks sortants vers les nouveaux tanks.

Système d'équipes

Chaque joueur se voit attribuer un numéro d'équipe unique (1, 2, 3, 4 selon l'ordre d'ajout). Ce numéro a trois effets concrets :

1. Collisions filtrées par équipe. Dans `World::update()`, deux entités ne se heurtent que si elles appartiennent à des équipes différentes. Cela empêche un joueur de se blesser avec ses propres balles, et ouvre la porte à un mode coopératif.

```
if (a->getTeam() != b->getTeam() && a->collides(b)) { ... }
```

2. Couleur des balles. Dans le rendu, chaque vue colore les balles selon leur origine : rouge pour les balles ennemies, bleu pour les balles alliées (y compris les siennes). Cela aide le joueur à lire rapidement la situation.

```
if (e->getTeam() != tank->getTeam())
    shape->setFillColor(sf::Color::Red);
else
    shape->setFillColor(sf::Color::Blue);
```

3. Couleur des tanks. Les tanks ennemis sont affichés en rouge, les alliés (et soi-même) en bleu, avec la même logique de comparaison d'équipe.

Section 3.4 — Gestion des collisions

Question 5.1 — Découpage spatial en grille

Cette question n'a pas été implémentée.

Question 5.2 — Performances (optionnelle)

Deux outils d'analyse ont été ajoutés. Un **compteur de FPS** (touche F5) affiche en temps réel le nombre de frames par seconde dans le coin de la fenêtre, permettant d'observer l'impact du nombre d'entités sur les performances. Un **mode stress test** (argument `./deepdiep stressTest`) peuple le monde avec un nombre d'entités bien supérieur à la normale (densité $\times 20$) et spawne de nombreux tanks automatiques, afin de saturer la détection de collisions en $O(n^2)$ et de mesurer le seuil à partir duquel le framerate chute.

Bibliographie

- Documentation SFML 2.6 : https://www.sfml-dev.org/documentation/2.6.0/
- Référence du jeu original : https://diep.io
- cppreference C++17 : https://en.cppreference.com