
# Santa Clash

*Atelier de synthèse : C#, POO, LINQ, MonoGame et TU*

**ATPROG P4**

**Règles d'usage de l'IA**

- L'utilisation de l'**IA** (ChatGPT, Copilot, assistants intégrés, etc.) **est strictement interdite**, sauf indication contraire de l'enseignant.
- L'usage de documents de référence est autorisé (support cours, référence API...).
- **Chaque ligne de code doit être comprise** : vous devez être capables d'expliquer vos choix et votre implémentation.
- **Des questions ponctuelles** pourront être posées à tout moment pour vérifier votre compréhension. En cas de difficulté à expliquer votre travail, **l'évaluation sera adaptée**.
- Il est de **votre responsabilité** de désactiver les aides automatiques.

**Objectif** : progresser dans vos compétences en **comprenant réellement ce que vous codez**, afin de savoir utiliser les outils d'IA de manière pertinente à l'avenir.

**CFPT Ecole informatique**
stephane.garchery@edu.ge.ch

*2025 – 2026*

# Table des matières



# Contexte de l'atelier

**Objectifs**

Cet atelier de fin de semestre a pour objectifs principaux :

- ❏ mettre en pratique la programmation orientée objet (héritage, interfaces, encapsulation) ;
- ❏ structurer un petit jeu 2D avec MonoGame ;
- ❏ utiliser LINQ pour manipuler des collections d'objets (filtrage, tri, agrégation) ;
- ❏ écrire et exécuter des tests unitaires sur la logique métier du jeu ;
- ❏ travailler en petite équipe et présenter un jeu jouable à deux joueurs en compétition.

Vous disposez de **3 séances de 4 périodes** pour concevoir, implémenter et tester un mini-jeu complet répondant aux contraintes ci-dessous.

# 1 Pitch du projet « Santa Clash »

> **Santa Clash** est un jeu 2D dans lequel deux gardiens de Noël protègent le Père Noël contre des vagues d'ennemis. Les joueurs coopèrent pour empêcher le Père Noël de mourir, mais sont en compétition pour détruire le plus grand nombre d'ennemis.

**Principes de jeu :**

- ❏ Le **Père Noël** est placé au centre (ou en bas) de l'écran et possède un certain nombre de points de vie.
- ❏ Des **vagues d'ennemis** apparaissent de tous les côtés et se dirigent vers le Père Noël. Les ennemis ont un déplacement cahotique.
- ❏ Deux **joueurs humains** contrôlent chacun un gardien de Noël et peuvent se déplacer et tirer sur les ennemis pour les détruire.
- ❏ Si le Père Noël perd tous ses points de vie, la partie est **perdue pour les deux joueurs**.
- ❏ À la fin de la partie, on affiche au minimum :
  - ❏ le gagnant (joueur ayant détruit le plus d'ennemis) ;
  - ❏ quelques statistiques (ennemis détruits par joueur, éventuellement précision des tirs, etc.).

Des améliorations et variantes sont acceptées **SI** tous les critères évaluables ci-dessous son respectés ! !

Tous les groupes développent une version du jeu « Santa Clash » avec deux joueurs locaux, LINQ et tests unitaires.

# 2 Contraintes techniques

## 2.1 Architecture objet minimale

Votre solution doit respecter au minimum la structure suivante :

- ❏ une classe abstraite `GameObject` contenant au moins :
  - — la position 2D ;
  - — la vitesse ;
  - — une propriété `IsAlive` ;
  - — une méthode virtuelle `Update()` ;
  - — une méthode virtuelle `Draw()`.
- ❏ des classes dérivées de `GameObject` :
  - — `Player` (au moins deux instances distinctes pour Joueur 1 et Joueur 2) ;
  - — `Enemy` (classe de base) avec au moins **deux types d'ennemis** différents (par ex. lents/rapides, au sol/aériens) ;
  - — `Projectile` (tirs des joueurs) ;
  - — `Santa` (Père Noël).
- ❏ une classe de gestion des vagues d'ennemis (par ex. `WaveManager`) ;
- ❏ une classe de gestion des états du jeu (par ex. `GameStateManager`) avec au minimum : `Menu`, `Playing`, `GameOver` ;
- ❏ au moins une interface, par exemple `IDamageable` avec la méthode `ApplyDamage(int amount)`.

L'encapsulation doit être respectée (champs privés, propriétés publiques contrôlées) et le code doit être organisé de manière lisible . . .et je ne parle même pas des commentaires.

## 2.2 Gameplay et multi local

### 2.2.1 Multi local.

- ❏ Le jeu est jouable à **deux joueurs humains** sur le même ordinateur.
- ❏ Joueur 1 utilise une **manette** (joystick pour le déplacement, un bouton pour tirer).
- ❏ Joueur 2 utilise le **clavier** (ou une deuxième manette si disponible).
- ❏ Les deux joueurs sont visibles en permanence sur la même carte.

### 2.2.2 Boucle de jeu minimale.

Votre jeu doit au minimum gérer :

- ❏ l'apparition de vagues d'ennemis et leur déplacement vers le Père Noël ;
- ❏ les tirs des joueurs, avec collision tirs / ennemis et destruction des ennemis ;
- ❏ la perte de points de vie du Père Noël en cas de contact avec les ennemis ;
- ❏ la fin de partie si la vie du Père Noël atteint zéro ;
- ❏ un **score individuel** pour chaque joueur (nombre d'ennemis détruits).

### 2.2.3 Affichage.

- ❏ Le jeu est en **2D sprite** (pas besoin de 3D).
- ❏ Implémentez un **effet de parallaxe** simple : au moins deux couches d'arrière-plan qui défilent à des vitesses différentes.
- ❏ Les collisions peuvent être gérées par de simples rectangles (AABB).
- ❏ Un **HUD** affiche clairement :
  - ❏ la vie du Père Noël ;
  - ❏ le score de chaque joueur.

## 2.3 Utilisation de LINQ

LINQ doit être utilisé de manière visible et pertinente dans le code.

Au minimum :

- filtrer les ennemis « actifs » ou « dangereux » (par ex. proches du Père Noël) ;
- sélectionner les couples objets en collision (par ex. projectile / ennemi) ;
- produire des statistiques de fin de partie (nombre d'ennemis détruits par type, par joueur, etc.) ;
- déterminer le classement final des joueurs à partir de leurs scores.

Le sujet exige au minimum **quatre requêtes LINQ non triviales** (avec `Where`, `Select`, `OrderBy`, `GroupBy`, etc.).

## 2.4 Tests unitaires

La logique métier doit être testable sans lancer le jeu MonoGame.

- Isolez la logique métier (collisions, calculs de score, génération de vagues, etc.) dans une bibliothèque de classes (par ex. projet `Core`).
- Créez un **projet de tests unitaires** (MSTest, NUnit ou xUnit) qui référence ce projet `Core`.
- Implémentez au minimum **10 tests unitaires** couvrant par exemple :
  - une fonction de collision (deux rectangles qui se touchent / ne se touchent pas) ;
  - le calcul de la précision d'un joueur (tirs touchés / tirs effectués, avec gestion du cas 0 tir) ;
  - la génération d'une vague (nombre et type d'ennemis attendus) ;

- au moins une requête LINQ de statistiques ou de classement.

## Aides techniques (à utiliser ou non)

### Aide : mouvement « chaotique » du Père Noël vers le centre

**Idée d'implémentation**

L'idée est de considérer une position cible fixe (par exemple le centre de l'écran), et d'y ajouter un mouvement amorti avec une petite perturbation aléatoire.

- **Position cible** : par ex. `targetPosition = new Vector2(screenWidth / 2, screenHeight / 2);`
- **Vitesse** : garder un vecteur `velocity` pour le Père Noël (stocké comme champ).
- **Attraction** : à chaque `Update`, calculer un vecteur direction vers la cible et l'utiliser comme force d'attraction.
- **Bruit** : ajouter un petit vecteur aléatoire pour rendre le mouvement « cahotique » plutôt que parfaitement lisse.
- **Amortissement** : appliquer un facteur `damping` sur la vitesse pour éviter qu'il ne parte à l'infini.

```
// Champs dans la classe Santa
private Vector2 _position;
private Vector2 _velocity;
private readonly Vector2 _targetPosition;
private readonly Random _random = new Random();

private float _attractionStrength = 50f;   // force qui attire vers le centre
private float _noiseStrength = 20f;        // intensité du "chaos"
private float _damping = 0.90f;            // amortissement de la vitesse

public void Update(float dt)
{
  // Direction vers la position cible
  Vector2 toCenter = _targetPosition - _position;
  if (toCenter != Vector2.Zero)
    toCenter.Normalize();

  // Petit bruit aléatoire
  float noiseX = (float)(_random.NextDouble() - 0.5); // [-0.5, 0.5]
  float noiseY = (float)(_random.NextDouble() - 0.5);
  Vector2 noise = new Vector2(noiseX, noiseY);
  if (noise != Vector2.Zero)
    noise.Normalize();

  // Force totale = attraction vers le centre + bruit
  Vector2 acceleration = toCenter * _attractionStrength + noise * _noiseStrength;

  // Intégration simple
  _velocity += acceleration * dt;
  _velocity *= _damping;      // amortissement
  _position += _velocity * dt;
}
```

Listing 1 – Exemple de logique possible côté C#

**Points de vigilance**

- Pensez à **clamp** la position dans la zone de jeu pour éviter que le Père Noël ne sorte de l'écran.
- Vous pouvez faire varier le `noiseStrength` pendant la partie (plus « paniqué » quand la vie de Santa diminue, par exemple).
- Le paramètre `_damping` doit être dans $(0, 1)$ pour que la vitesse reste contrôlée.

### Aide : déplacement pseudo-aléatoire des ennemis vers le Père Noël

**Idée d'implémentation**

Les ennemis se dirigent globalement vers le Père Noël, mais avec un « zigzag » ou une trajectoire irrégulière. On peut partir d'une direction de base vers Santa et y ajouter une légère rotation aléatoire ou une petite composante latérale.

- **Direction de base** : vecteur normalisé `dirToSanta = (santaPos - enemyPos).Normalized()`.
- **Angle aléatoire** : ajouter un petit angle aléatoire $\theta$ à chaque mise à jour (ou toutes les N frames) pour dévier la trajectoire.
- **Vitesse constante** : garder une vitesse `speed` globale, pour éviter que certains ennemis soient immobilisés.
- **Option zigzag** : conserver une « direction latérale » (gauche/droite) et la changer de temps en temps pour créer des oscillations.

```
// Champs dans la classe Enemy
private Vector2 _position;
private float _speed = 80f;
private readonly Random _random = new Random();

// angle max de déviation en radians (par ex. + ou -15 degrés)
private float _maxAngleOffset = MathF.PI / 12f;

public void Update(float dt, Vector2 santaPosition)
{
  // Direction de base vers Santa
  Vector2 dir = santaPosition - _position;
  if (dir == Vector2.Zero) return;
  dir.Normalize();

  // Petit angle aléatoire à chaque update (ou toutes les N frames)
  float angleOffset = (float)(_random.NextDouble() - 0.5) * 2f * _maxAngleOffset;
  float cos = MathF.Cos(angleOffset);
  float sin = MathF.Sin(angleOffset);

  // Rotation 2D (x', y') = (x cos - y sin, x sin + y cos)
  Vector2 dirRotated = new Vector2(
    dir.X * cos - dir.Y * sin,
    dir.X * sin + dir.Y * cos
  );

  // Déplacement à vitesse constante
  _position += dirRotated * _speed * dt;
}
```

Listing 2 – Exemple simple avec une petite rotation aléatoire de la direction de base

**Variantes possibles**

- Mettre à jour l'angle aléatoire seulement **tous les N frames** pour éviter un mouvement trop « tremblotant ».
- Ajouter un **offset latéral** qui oscille avec un `MathF.Sin(time * freq)` pour faire comme des ennemis qui zigzaguent.
- Adapter `_maxAngleOffset` en fonction du type d'ennemi (ennemi très erratique vs ennemi presque « missile guidé »).

# 3 Organisation des trois sprints

## Sprint 1 : conception et mise en place

- Constitution des équipes (2 à 3 apprentis par projet).

- Description rapide du jeu (une demi-page) pour fixer les règles.
- Diagramme de classes simple basé sur la structure imposée.
- Création de la solution (projet `Game` + projet `Core`).
- Implémentation de base : affichage des deux joueurs et déplacement.

### Sprint 2 : gameplay et scoring

- Implémentation de plusieurs types d'ennemis et du gestionnaire de vagues.
- Gestion des tirs, collisions, destruction d'ennemis, mise à jour des scores.
- Gestion de la vie du Père Noël et de la fin de partie.
- Premier écran de fin de partie avec affichage du gagnant.

### Sprint 3 : LINQ, tests et finition

- Finalisation et nettoyage des requêtes LINQ (collisions, stats, classement).
- Complétion du projet de tests unitaires (au moins 10 tests en vert).
- Amélioration du HUD et du ressenti de jeu (lisibilité, petites corrections).
- Préparation d'une démonstration de 2 à 3 minutes.

# 4 Livrables et principe d'évaluation

## 4.1 Livrables

À la fin du projet, chaque équipe remet :

- ❏ la solution complète (projets C#, incluant `Game` et `Core`) et les tests unitaires avec tous les tests en vert ;
- ❏ un court fichier `README` expliquant :
  - comment lancer le jeu ;
  - les commandes des deux joueurs ;
  - les principales fonctionnalités implémentées.
- ❏ le code source au format pdf (LaTeX)

## 4.2 Évaluation

L'évaluation s'appuie sur :

- ❏ **l'implication individuel et le comportement professionnel de chaque participant ;**
- ❏ une auto-évaluation de l'équipe via une grille simple (cases à cocher) ;
- ❏ une évaluation par une autre équipe qui teste votre jeu et valide la grille ;
- ❏ une validation rapide par l'enseignant (lancement du jeu, vérification de quelques points clés).

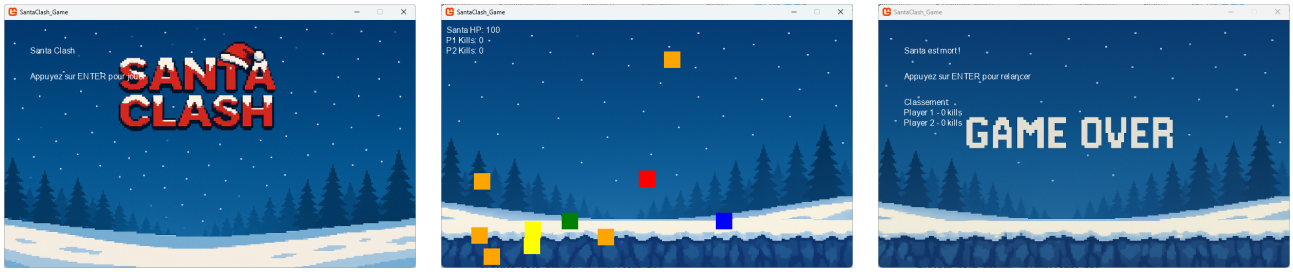
Les critères porteront également sur :

- ❏ la présence et le fonctionnement du multi local et du gameplay de base ;
- ❏ le respect des contraintes techniques (POO, LINQ, tests unitaires) ;
- ❏ la stabilité minimale du jeu (pas de crash immédiat, commandes utilisables).

> *Ce document est né des idées de l'auteur, qui l'a rédigé puis relu et validé. Des outils d'aide ont pu l'accompagner (correction, reformulation, clarification, mise en forme…), sans jamais se substituer à sa réflexion ni à ses choix de contenu.*

# 5 Exemple de visuels possibles. . .

Exemple d'écrans (sans les sprites),




(1a) Écran d'accueil (1b) Écran de jeu (1c) Écran de fin

FIGURE 1 – Maquettes des principaux écrans de l'application.

Vous avez le droit de faire mieux. . .