Application Web

Jeu Canvas

Alumni :
BOUTRIK Alexandre
MANILIUC David
LANDOULSI Aziz

Enseignant :
BUFFA Michel

Février 15, 2026

Table des Matière

1 Introduction Préliminaire

Nous avons déjà réalisé le menu principal, qui figure dans le directoire *menu* du repositoire. Toutefois, celui-ci n'est pas prévu pour le présent rendu; il fera l'objet de légers ajustements avant son intégration définitive la semaine prochaine.

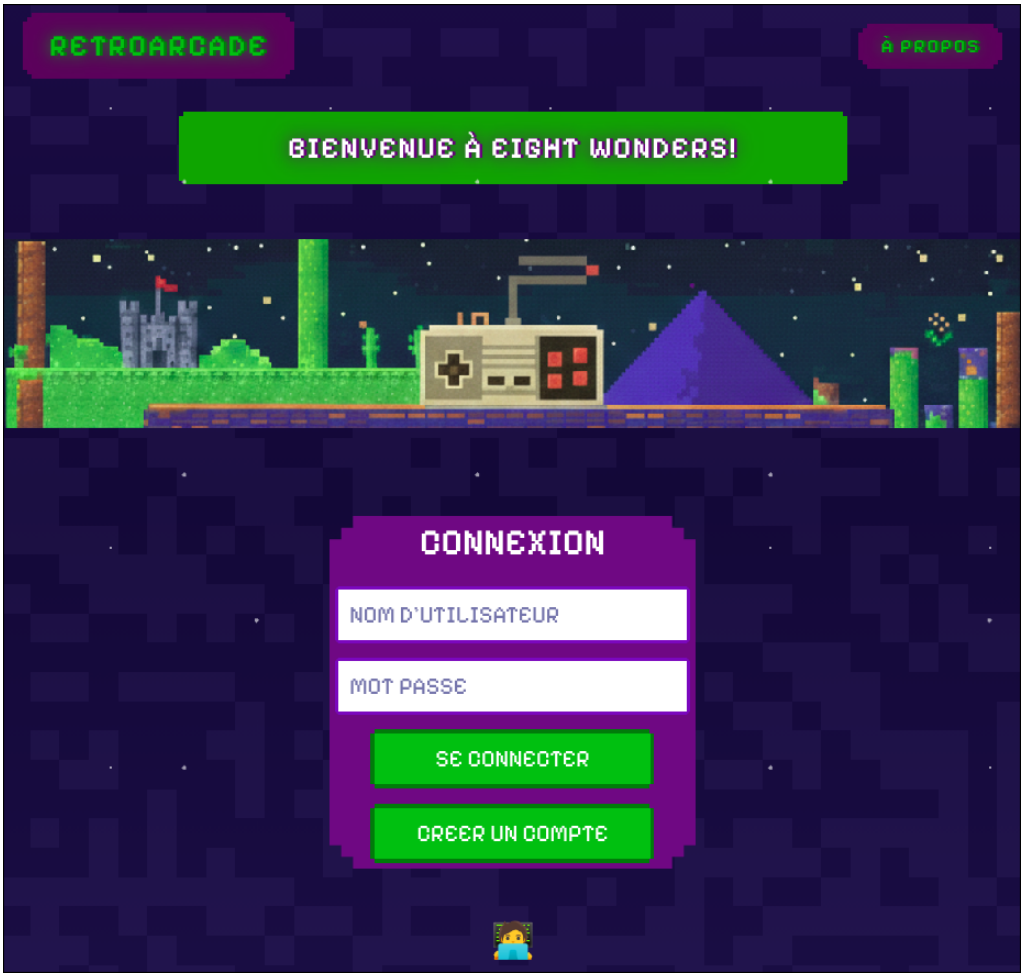

Fig. 1: Aperçu de l'interface du menu principal.

2 Vue d'ensemble du Jeu

Le gameplay s'inspire de classiques du genre « artillerie » tels que *Worms* et *DDTank*. Le joueur incarne un personnage (Mage ou Archer) et doit affronter une série d'ennemis contrôlés par l'ordinateur (simple bots, goblins, dragons) à travers différents niveaux. L'objectif est d'utiliser stratégiquement les déplacements, la visée et un éventail de capacités (projectiles soumis à la gravité, soins, téléportation) pour réduire les points de vie des adversaires à zéro avant que le joueur ne soit vaincu.

3 Architecture du Jeu

.

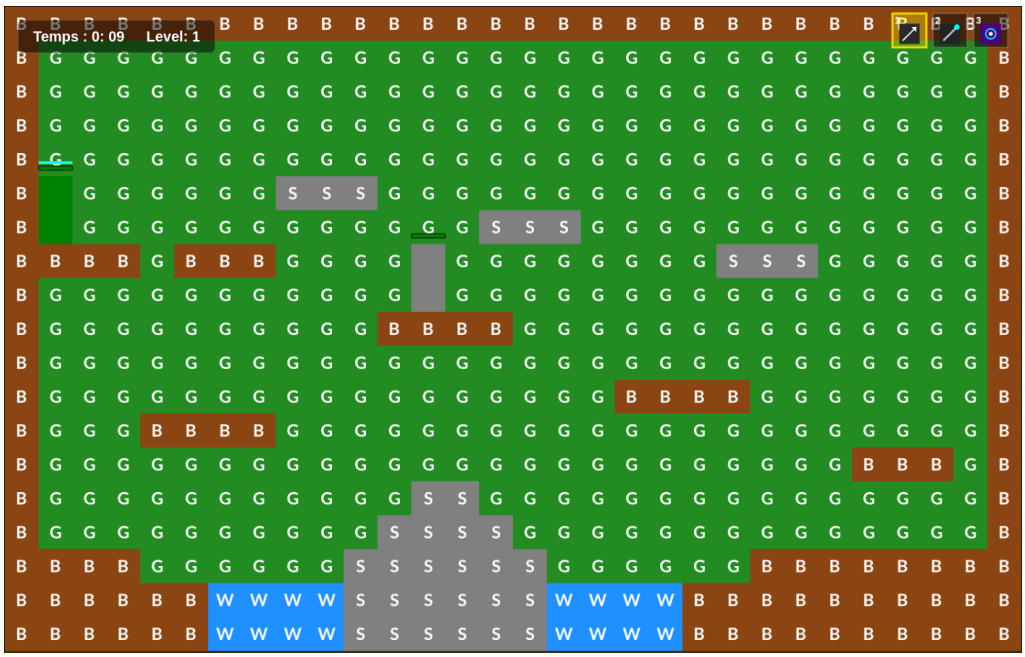

Fig. 2: Aperçu du niveau 1 : Interface de jeu, environnement et HUD.

```
|-- babylonjs
|-- canvas
|   |-- assets
|   |   |-- maps
|   |-- css
|   |-- documentation
|   |   |-- latex-src
|   |-- js
|       |-- abilities
|       |   |-- combat
|       |   |-- instant
|       |-- ai
|       |-- core
|       |-- players
|       |-- projectiles
|-- common
|-- dom
|-- menu
```

L'architecture technique du projet a été conçue pour être modulaire et extensible, séparant distinctement la logique de jeu, le rendu graphique et la gestion des données. Cette séparation se reflète directement dans l'organisation des fichiers sources. Le cœur du système réside dans le répertoire *js/core*, qui contient les chefs d'orchestre du jeu: *game.js* pour la boucle principale, *turn_manager.js* pour la gestion du tour par tour, et *level_manager.js* pour le chargement des niveaux. Les entités jouables et les ennemis sont définis dans *js/players*, tandis que leurs cerveaux (les stratégies d'intelligence artificielle) sont découplés dans le dossier *js/ai*.

Enfin, tout ce qui concerne les attaques et les compétences est regroupé sous *js/projectiles* et *js/abilities*, assurant que chaque aspect du gameplay possède son propre espace de définition.

3.1 Boucle Principale (Main Loop)

Le jeu repose sur une boucle d'animation gérée par la classe *Game*. L'exécution démarre depuis *js/main.js*, une fois le DOM chargé. La méthode *loop()* s'appuie sur `requestAnimationFrame` pour assurer un rafraîchissement fluide. À chaque itération, deux étapes sont exécutées: *update()*, qui met à jour la physique, les positions et la logique de jeu, puis *draw()*, qui efface le canvas et redessine la scène avec son état courant.

La boucle principale - js/core/game.js

```
loop() {
  this.update();
  this.draw();

  // sans setInterval
  requestAnimationFrame(this.loop);
}
```

3.2 Gestion des États (State Management)

Le déroulement du jeu est piloté par une machine à états finis. L'état actif est stocké dans *currentState* et correspond à une valeur de l'énumération *GAME_STATE* (par exemple *MENU*, *PLAYING*, *LEVEL_TRANSITION*, *GAME_OVER* ou *VICTORY*). À chaque frame, la boucle de rendu consulte cet état pour afficher l'interface appropriée : menu principal, HUD en jeu ou écrans de fin via le *UIManager*.

Gestion des etats dans draw() - js/core/game.js

```
switch (this.currentState) {
  case GAME_STATE.MENU:
  this.ui.drawMenu(); // 60 FPS Menu
  break;
  case GAME_STATE.PLAYING:
  if (this.map) this.map.draw(this.ctx);
  // ... drawing players ...
  break;
  case GAME_STATE.GAME_OVER:
  this.ui.drawGameOver();
  break;
}
```

3.3 Système d'Entités et Composants

Le jeu adopte une approche orientée objet stricte utilisant l'héritage pour structurer les acteurs du jeu. La classe fondamentale *GraphicalObject* gère les propriétés de bas niveau comme la position spatiale, les dimensions et le rendu graphique de base via les transformations du

contexte Canvas (*ctx.translate*). La classe *Player* étend (extends) cette base en y ajoutant la physique (gravité, sauts) et les mécanismes de combat. Les classes spécifiques, qu'il s'agisse du joueur humain (*Mage*, *Archer*) ou des ennemis contrôlés par l'ordinateur (*Bot*), héritent toutes de cette structure commune. Pour les ennemis, une couche supplémentaire de spécialisation permet de distinguer des comportements uniques, comme ceux du *Goblin* ou du *Dragon*, tout en réutilisant la logique d'IA générique.

Rendu avec transformations - js/graphical_object.js

```
draw(ctx) {
  ctx.save(); // Isolation du contexte

  // Deplacement de l'origine sur l'objet
  ctx.translate(this.x, this.y);

  ctx.fillStyle = this.color;
  // Dessin a (0,0) relatif
  ctx.fillRect(0, 0, this.width, this.height);

  ctx.restore(); // Restauration pour l'objet suivant
}
```

3.4 Gestion des Niveaux (Level Management)

La construction des niveaux est déléguée au *LevelManager*, qui agit comme une usine à entités. La configuration de chaque niveau, incluant le fichier de carte à charger et la position initiale des ennemis, est centralisée dans le fichier *js/core/levels.js*. Les cartes elles-mêmes sont stockées sous forme de fichiers CSV dans le dossier *assets/maps*. Le système parse ces fichiers pour générer une grille bidimensionnelle d'objets *Map*, où chaque cellule est identifiée par un ID correspondant à un type de terrain spécifique (herbe, brique, eau ou pierre), permettant ainsi une détection de collision précise basée sur les tuiles.

Configuration centralisee des niveaux - js/core/levels.js

```
export const LEVEL_CONFIG = [
{
  id: 1,
  mapFile: "assets/maps/01.csv",
  enemies: [
  { type: "bot", x: 600, y: 100 }
  ]
},
{
  id: 2,
  mapFile: "assets/maps/02.csv",
  enemies: [
  { type: "goblin", x: 1150, y: 100 },
  { type: "bot", x: 1728, y: 100 }
  ]
},
{
  id: 3,
  mapFile: "assets/maps/03.csv",
  enemies: [
  { type: "dragon", x: 500, y: 100 }
  ]
}
];
```

Parseur de fichiers de carte - ../common/csv-parser.js

```
export function parse_csv_from_string(str) {
  // Decoupage par ligne puis par virgule
  const rows = str.trim().split('\n');

  // Conversion des caracteres en nombres pour la grille de jeu
  return rows.map(row => row.split(',').map(Number));
}
```

4 Conception & Design Patterns

Les choix d'architecture ont été orientés vers la clarté du code et sa capacité d'évolution. Cette section présente notamment l'implémentation de l'IA via le patron Stratégie, la gestion du tour par tour et le système de collisions.

4.1 Système d'IA et Patron de Conception Stratégie

Les ennemis contrôlés par l'ordinateur utilisent le *Strategy Pattern*, ce qui permet de dissocier la prise de décision de l'entité elle-même. La classe *Bot* délègue ainsi ses actions à une instance de *AIStrategy*, interchangeable selon le comportement souhaité. Trois niveaux d'intelligence ont été implémentés.

Delegation via Strategy Pattern - js/players/bot.js

```
updateBotLogic(map, players) {
  if (!this.turnActive || !this.strategy) return false;

  return this.strategy.update(this, map, players);
}
```

La stratégie *DumbAI*, utilisée par les Goblins, adopte un comportement simple : elle se rapproche du joueur dès que possible, effectue des sauts aléatoires et tire sans tenir compte de la gravité, ce qui la rend surtout dangereuse à courte portée. À l'inverse, `SmartAI` privilégie une approche plus prudente. Elle tente de conserver une distance optimale (environ 350 pixels), évite les zones dangereuses et calcule une trajectoire parabolique pour ses tirs, même en présence d'obstacles. Enfin, la *StationaryAI*, réservée aux boss comme le Dragon, supprime toute logique de déplacement et transforme l'ennemi en tourelle fixe qui vise et tire à intervalles réguliers.

4.2 Gestion du Tir Allié

Une logique de prévention des tirs alliés a été théoriquement implémentée au sein de l'IA via la méthode *isFriendInLineOfFire*. Celle-ci a pour but d'utiliser une technique de « raycasting » pour analyser la trajectoire balistique et annuler le tir si un allié se trouve sur la ligne de visée.

Toutefois, il est important de préciser que cette fonctionnalité est actuellement inopérante. Malgré la présence du code, le mécanisme de détection échoue systématiquement et ne parvient pas à prévenir les tirs fratricides. Ce module est donc considéré comme buggé et non-fonctionnel dans la version actuelle du rendu.

Detection de friendly fire par Raycasting - js/ai/ai_strategy.js

```
isFriendInLineOfFire(bot, aimAngle, map, players, target) {
  const range = 800;
  const step = 20;
  const safetyMargin = 60; // Marge pour compenser la taille des
  projectiles

  for (let d = 10; d < range; d += step) {
    const px = (bot.x + bot.width / 2) + Math.cos(aimAngle) * d;
    const py = (bot.y + bot.height / 2) + Math.sin(aimAngle) * d;

    // Arret si on touche un mur
    if (map.getTile(Math.floor(px / map.tileSize), Math.floor(py / map.
  tileSize)) !== 0) return false;

    for (const p of players) {
      if (p !== bot && p.health > 0) {
        if (px >= p.x - safetyMargin && px <= p.x + p.width +
  safetyMargin &&
        py >= p.y - safetyMargin && py <= p.y + p.height + safetyMargin)
   {
          if (p === target) return false; // Cible legitime
          if (typeof p.strategy !== 'undefined') return true; // Ami
  detecte
        }
      }
    }
  }
  return false;
}
```

4.3 Gestion des Tours et Mécaniques d'Esquive

Le système de tour par tour, géré par le *TurnManager*, introduit une dynamique particulière. Contrairement à certains jeux d'artillerie où le tour se termine après l'impact du projectile, le contrôle passe ici au joueur suivant immédiatement après le tir. Le temps de trajet du projectile crée ainsi une courte fenêtre pendant laquelle l'adversaire peut tenter d'esquiver, ajoutant une dimension réactive au gameplay tactique.

Logique de transition de tour - js/core/game.js

```
update() {
  const activePlayer = this.turnManager.getCurrentPlayer(this.players);

  // Si le joueur vient de tirer, on change de tour immediatement
  if (activePlayer && activePlayer.hasFired) {
    activePlayer.hasFired = false;
    this.turnManager.nextTurn(this.players);
  }

  // Les projectiles continuent de vivre independamment du tour
  this.players.forEach(p => p.updateProjectiles(this.map, this.players))
  ;
}
```

4.4 Système de Temps et Score

Dans ce projet, la performance du joueur (score) n'est pas mesurée à l'aide d'un système de points classique, mais à travers son efficacité temporelle. Le score final correspond ainsi au temps total nécessaire pour terminer l'ensemble des niveaux, ce qui met davantage l'accent sur la rapidité d'exécution que sur l'accumulation d'actions ou de bonus.

Le suivi du temps est directement intégré au cœur du moteur de jeu dans *game.js*. Au début de chaque niveau, un instant de référence est enregistré dans la variable *startTime*, servant de point de départ pour le chronométrage. Une fois le niveau terminé, le temps écoulé est calculé en comparant l'horodatage courant à cette valeur initiale, puis ajouté à *accumulatedTime*, qui conserve la durée totale passée depuis le début de la partie.

L'affichage du temps est pris en charge par le *UIManager*, qui convertit la durée en secondes vers un format minutes-secondes lisible dans le HUD.

Calcul du temps accumule - js/core/game.js

```
advanceLevel() {
  const currentLevelTime = Math.floor((Date.now() - this.startTime) /
 1000);
  this.accumulatedTime += currentLevelTime; // Le score est le temps
 total cumule

  if (this.currentLevelIdx < LEVEL_CONFIG.length) {
    this.currentLevelIdx++;
    this.startLevel(this.currentLevelIdx);
  } else {
    this.currentState = GAME_STATE.VICTORY;
  }
}
```

4.4.1 Persistance des Données (Backend Node.js & MongoDB)

Afin de sauvegarder les performances des joueurs, une architecture client-serveur a été mise en place.

Lorsqu'une partie est remportée (état *VICTORY*), une interface dédiée invite le joueur à saisir son pseudonyme (*username*). Le système capture alors trois informations essentielles: le *Username* (saisi par le joueur), le *Score* (le temps accumulé final) et le *Timestamp* (l'horodatage actuel au format ISO).

Ces données sont transmises de manière asynchrone via la classe *ScoreService* à une API REST développée en *Node.js*. Le backend se charge ensuite de valider les données et de les stocker de manière persistante dans une base de données *MongoDB*. Cette approche permet de conserver un historique global des meilleurs temps réalisés.

Service d'envoi de score - js/services/score.js

```
export class ScoreService {
  /**
  * Envoie le score au backend Node.js/MongoDB.
  * @param {string} username
  * @param {number} score
  */
  static async submit(username, score) {
    try {
      const response = await fetch("http://localhost:3000/api/score", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          username,
          score, // Correspond a accumulatedTime
          timestamp: new Date().toISOString()
        })
      });

      if (!response.ok) {
        const errData = await response.json().catch(() => ({}));
        return { success: false, error: errData };
      }

      const data = await response.json();
      return { success: true, data };
    } catch (error) {
      return { success: false, error };
    }
  }
}
```

4.5 Système de Collisions

Les collisions sont traitées selon deux approches complémentaires. Pour l'environnement, une détection basée sur les tuiles est utilisée : la position d'une entité est convertie en coordonnées de grille afin de vérifier la présence d'obstacles ou de zones létales via la méthode *getTile*. En parallèle, les interactions entre projectiles et entités reposent sur des boîtes englobantes alignées sur les axes (AABB - Axis-Aligned Bounding Box). À chaque frame, le chevauchement entre rectangles est testé afin d'appliquer dégâts et recul.

Collision avec l'environnement - js/players/player.js

```
checkCollision(map, axis) {
  const startCol = Math.floor(this.x / map.tileSize);
  const endCol = Math.floor((this.x + this.width - 0.1) / map.tileSize);
  const startRow = Math.floor(this.y / map.tileSize);
  const endRow = Math.floor((this.y + this.height - 0.1) / map.tileSize)
  ;

  for (let row = startRow; row <= endRow; row++) {
    for (let col = startCol; col <= endCol; col++) {
      const tile = map.getTile(col, row);

      if (tile === tilesTypes.water) {
        this.health = 0; // Mort instantanee dans l'eau
        return;
      }
      if (tile !== 0) { // Collision avec un mur (brick/stone)
        if (axis === "x") this.x = (this.vx > 0) ? col * map.tileSize -
 this.width : (col + 1) * map.tileSize;
        else { this.y = (this.dy > 0) ? row * map.tileSize - this.height
  : (row + 1) * map.tileSize;
          if (this.dy > 0) { this.dy = 0; this.grounded = true; } }
      }
    }
  }
}
```

5 Mécaniques de Jeu

Le système de compétences du jeu repose sur une distinction architecturale entre les actions de combat physiques (*combat*) et les effets instantanés (*instant*). Les capacités offensives génèrent des instances de la classe *Projectile*, qui sont des objets graphiques soumis aux lois de la physique et aux collisions. En revanche, les capacités de soutien sont traitées comme des actions immédiates; par exemple, le soin (*Heal*) - modifie directement les points de vie sans exister dans l'espace de jeu, tandis que la téléportation (*Teleport*) altère instantanément les coordonnées x et y de l'entité sans traverser l'espace intermédiaire.

Capacite instantanee de soin - js/abilities/instant/heal.js

```
export class Heal extends Projectile {
  constructor(x, y, angle, owner) {
    super(x, y, angle, owner, 0);
    this.active = false; // Desactivation immediate (pas de rendu)

    // Application directe de l'effet
    const healAmount = Math.floor(owner.maxHealth * 0.2);
    owner.health = Math.min(owner.maxHealth, owner.health + healAmount);

    // Nettoyage des alterations d'etat
    owner.statuses = [];
  }
}
```

Actuellement, le joueur a le choix entre deux classes distinctes, le Mage et l'Archer, qui disposent chacune d'un arsenal fixe de trois compétences (*abilities*). La première compétence constitue l'attaque de base, la seconde est conçue pour infliger des dégâts massifs contre les boss comme le Dragon, et la troisième est une compétence utilitaire dédiée à la mobilité. Pour l'instant, cette compétence de mouvement est uniformisée sous la forme d'une téléportation pour les deux classes, permettant de se repositionner stratégiquement sur la carte.

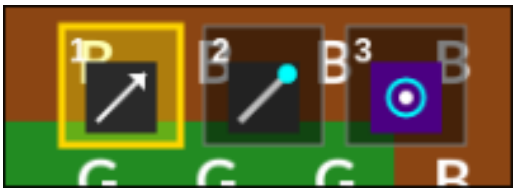

Fig. 3: Interface de sélection des capacités (*abilities*) affichée en jeu.

Le moteur physique qui régit ces interactions implémente une simulation de la gravité, de la friction et de la puissance de saut directement au sein de la classe *Player*. Chaque entité possède des vecteurs de vitesse verticale (dy) et d'inertie horizontale (vx), permettant des déplacements fluides et la gestion des chutes.

6 Améliorations Futures

Pour le rendu final du projet, plusieurs axes d'amélioration ont été identifiés afin d'enrichir l'expérience utilisateur et la profondeur stratégique. Sur le plan visuel, nous prévoyons de remplacer les formes géométriques primitives actuelles, dessinées via le contexte Canvas, par des sprites SVG ou des images détaillées pour donner une identité artistique plus marquée au jeu. L'immersion sera également renforcée par l'ajout de musiques d'ambiance, d'effets sonores et d'animations plus fluides pour les actions des personnages.

Nous envisageons également d'étendre le roster des personnages jouables avec des classes aux mécaniques radicalement différentes. Une classe "Fantôme" est à l'étude, possédant la capacité unique d'ignorer les collisions avec les murs lors de ses déplacements et d'infliger des dégâts uniquement par contact direct. De plus, une classe "Assassin" pourrait introduire une

dynamique de dissimulation, avec la possibilité de se déplacer pendant quelques secondes après avoir tiré un projectile avant de devenir invisible, forçant l'adversaire à deviner sa position.

Enfin, pour assurer la pérennité du jeu, nous souhaitons développer un éditeur de cartes intégré qui permettra aux joueurs de concevoir et de partager leurs propres niveaux personnalisés.

7 Déclaration d'utilisation d'IA

Dans un souci de transparence académique, nous tenons à déclarer que ce projet a bénéficié d'une utilisation extensive d'outils d'intelligence artificielle générative. Cette collaboration homme-machine s'est articulée autour d'une répartition claire des responsabilités: l'équipe a agi en tant qu'architecte et maître d'œuvre, tandis que l'IA a servi d'exécutant technique.

L'intégralité de la vision créative, incluant les idées originales, les mécaniques de jeu (système de compétences, gestion du tour par tour) et la structure globale du projet, a été planifiée et conçue par nos soins. L'intelligence artificielle a été sollicitée principalement pour la phase d'implémentation, générant une part significative du code source (classes, méthodes, logique de bas niveau) sur la base de nos spécifications. Plus précisément, nous avons utilisé Gemini Pro, Gemini Thinking et Gemini CLI.

Cependant, ce code n'a pas été intégré sans filtre. Nous avons exercé une supervision constante et rigoureuse sur le travail fourni par l'IA. Notre rôle a consisté à relire, tester et corriger les incohérences, ainsi qu'à imposer des directives strictes concernant l'architecture logicielle. Nous sommes intervenus manuellement pour rectifier des erreurs de conception et pour garantir que le code respecte les patrons de conception (Design Patterns) définis, tels que le *Strategy Pattern* pour les comportements des ennemis.

En somme, si l'IA a accéléré la production, la cohérence technique et la qualité finale du projet relèvent de notre entière responsabilité.