P\_Bulles

Snake en JavaScript

Une image contenant texte, capture d’écran, Police, Graphique

Le contenu généré par l’IA peut être incorrect.

Matias Denis

ETML – FID2

Vennes – Lausanne

Table des matières

[Introduction du projet 3](#_Toc212445630)

[Objectifs du jeu 3](#_Toc212445631)

[Technologies utilisées 3](#_Toc212445632)

[Architecture et conception 4](#_Toc212445633)

[Installation et exécution 4](#_Toc212445634)

[Implémentation par module 5](#_Toc212445635)

[Journal du déroulement 8](#_Toc212445636)

[Tests et résultats 9](#_Toc212445637)

[Difficultés rencontrées et solutions 9](#_Toc212445638)

[Contrôle de version (Git/GitHub) 9](#_Toc212445639)

[Recours à l’IA 10](#_Toc212445640)

[Conclusion 10](#_Toc212445641)

# Introduction du projet

Ce projet a pour but de réaliser une réplique du jeu Snake en JavaScript. Le jeu est dessiné sur un canevas HTML (<canevas>).

Le code JavaScript utilise la syntaxe des modules ES (ECMAScript Modules, ESM) pour organiser le code en plusieurs fichiers avec des responsabilités distinctes.

Le serveur de développement est géré par Vite qui est un outil moderne qui permet un rechargement rapide et une configuration minimale.

Le projet respecte le cahier des charges fourni avec un code organisé, des commentaires, des tests manuels et des bonnes pratiques de versionnement.

# Objectifs du jeu

Les objectifs du jeu sont :

* Contrôler un serpent qui se déplace sur une grille.
* Manger de la nourriture (des pommes) pour grandir.
* Eviter de se heurter aux murs ou à son propre corps.
* Le score augmente à chaque fois que le serpent mange de la nourriture.

# Technologies utilisées

**Vite**

Le projet utilise Vite comme serveur de développement car c’est un outil moderne qui offre une expérience de développement rapide et fluide. Il permet un rechargement automatique à chaque modification du code.

**ECMAScript Modules (ESM)**

Le projet est structuré en utilisant la syntaxe ESM, qui permet d’importer des fonctions et des variables entre les différents fichiers JavaScript. Cela permet de séparer différentes responsabilités du code et de maintenir une structure propre et modulable.

**JSDoc**

Le code est documenté avec JSDoc, un standard de documentation pour JavaScript. JSDoc permet la génération automatique d’une documentation à partir des commentaires du code, ce qui améliore la lisibilité et la maintenabilité du projet.

# Architecture et conception

Arborescence

└── src/

├── main.js // boucle de jeu, orchestration et rendu global

├── snake.js // init du serpent, calcul du déplacement, dessin

├── food.js // génération et rendu de la nourriture

├── controls.js // gestion clavier et anti-demi-tour

├── collision.js // détection des collisions (murs, corps)

└── score.js // affichage du score (HUD)

Modèle de données et constantes

* snake : tableau de segments {x : number, y : number}, snake[0] est la tête.
* box : taille d’une case en pixels (ex.20). Toutes les coordonnées sont des multiples de box.
* gameSpeed : durée d’un tick en ms (ex. 200ms)

# Installation et exécution

Il est nécessaire d’installer Node.js afin d’avoir un environnement d’exécution JavaScript.

Depuis le dossier racine du jeu, il faut taper la commande **npm install** dans l’invite de commandes.

Pour accéder à la page du jeu, il est nécessaire de lancer le serveur avec la commande **npm run dev** dans l’invite de commandes depuis le dossier racine du jeu.

Le lien <http://localgost:3000/> donne accès à la page du jeu.

# Implémentation par module

Cette section décrit ce qui est implémenté et pourquoi. Elle s’appuie sur le code du projet.

7.1 snake.js

Rôle : gérer le modèle du serpent (création, déplacement, rendu).

* initSnake()
  + But : créer l’état initial du serpent.
  + Implémentation : retourne un tableau avec un seul segment centré ({ x: 200, y: 200 }).
  + Pourquoi : point de départ clair, aligné sur la grille (20×20 si box=20).
* moveSnake(snake, direction, box)
  + Signature : Array<{x,y}>, "LEFT|UP|RIGHT|DOWN", number → {x,y}
  + Idée clé : ne pas muter le tableau directement. La fonction calcule la nouvelle tête à partir de la tête actuelle puis renvoie un nouvel objet {x,y}.
  + Algorithme :
    1. lire la tête (snake[0]),
    2. copier x/y dans nx/ny,
    3. ajuster selon direction de ±box,
    4. retourner { x: nx, y: ny }.
  + Complexité : O(1).
  + Design choice : séparation calcul (pur) / mutation (orchestration) → code testable et lisible.
* drawSnake(ctx, snake, box)
  + Rendu : boucle sur les segments et dessine un carré plein box×box par segment. La tête (index 0) est dessinée dans une couleur différente (#10b981 vs #34d399) pour la lisibilité.

Invariants & contraintes

* Les coordonnées {x,y} sont toujours des multiples de box.
* snake[0] est toujours la tête.
* Le déplacement se fait case par case (pas de diagonales).

7.2 food.js

Rôle : gérer la nourriture (génération aléatoire alignée sur la grille + dessin).

* generateFood(box, canvas)
  + But : placer la nourriture dans les limites du canvas et alignée sur la grille.
  + Implémentation : calcule maxX = floor(width/box) et maxY = floor(height/box), puis choisit un couple aléatoire, multiplié par box pour garantir l’alignement.
  + Remarque : dans une version avancée, on peut éviter de générer sur le corps du serpent (tirage avec rejet ou set des cases libres).
* drawFood(ctx, food, box)
  + Rendu : carré rouge (#e11d48) de taille box aux coordonnées {x,y}.

7.3 controls.js

Rôle : traduire les entrées clavier en directions valides.

* handleDirectionChange(event, currentDirection)
  + But : empêcher les demi-tours immédiats (ex. aller à gauche si on va à droite).
  + Implémentation : switch(event.key) sur ArrowLeft/Up/Right/Down. Retourne soit la nouvelle direction, soit la direction courante si la nouvelle est opposée.
  + Bénéfice gameplay : évite que le joueur se morde instantanément.

7.4 collision.js

Rôle : détection des conditions de fin.

* checkWallCollision(head, canvas, box)
  + But : savoir si la tête sort du canevas.
  + Implémentation : collision si x < 0, y < 0, x >= canvas.width ou y >= canvas.height. (Les coordonnées étant en pixels, box n’est pas nécessaire pour la comparaison, mais la signature le prévoit pour rester cohérent avec l’appelant.)
* checkCollision(head, snakeArray)
  + But : savoir si la tête touche le corps.
  + Implémentation : on compare head à chaque segment du corps à partir de l’index 1. Si une égalité (x,y) est trouvée → collision.

7.5 score.js

Rôle : affichage du HUD score.

* drawScore(ctx, score)
  + Rendu : texte Score: <valeur> en haut à gauche (police Arial 20px, noir).
  + Accessibilité : contraste lisible sur fond clair ; on peut renforcer via un fond semi-transparent derrière le texte.

7.6 Orchestration (dans main.js)

Cycle de jeu :

1. Initialisation (startGame) : création du serpent et de la nourriture, remise à zéro des variables de pause (pausedTime, pauseStart, isPaused), démarrage du chronomètre (startTime = Date.now()), puis lancement de la boucle de rendu via setInterval(draw, gameSpeed).
2. Boucle draw():
   * Effacer le canevas puis redessiner la nourriture, le serpent et le score (affiché dans le canvas ou, en option, dans le DOM via #scoreDisplay).
   * Pause : si isPaused est vrai, afficher un overlay “PAUSE” centré et sortir de draw() sans mettre à jour l’état (aucun mouvement, aucun calcul, aucun collision check).
   * Sinon, calculer la prochaine tête avec moveSnake, puis tester les collisions (checkWallCollision / checkCollision).
3. Manger : si nextHead est sur la nourriture → score++, pas de pop() (le serpent grandit) et régénération de la nourriture via generateFood. Sinon → snake.pop() pour un déplacement à taille constante.
4. Game Over : en cas de collision, arrêter l’intervalle, calculer le temps écoulé hors pauses (elapsedTime = (Date.now() - startTime - pausedTime)), puis afficher un bandeau/overlay centré avec des lignes de texte (message, score, temps, consigne de relance). L’overlay a une hauteur dynamique selon le nombre de lignes affichées.

Contrôles :

* keydown :
  + Espace → togglePause() : bascule pause/reprise. À la reprise, on additionne la durée de la pause à pausedTime pour que le chrono final n’inclue pas le temps de pause.
  + Flèches → mise à jour de la direction via handleDirectionChange (anti-demi-tour).

# Journal du déroulement

*Séquence 1*

Le code de ce projet n’étant pas encore possible dû au manque de connaissance technique concernant JavaScript, il a été nécessaire de trouver de la documentation ainsi que des tutoriels. Les liens de ces derniers sont disponibles dans le journal de travail.

Séquence 2

Cette séquence a servi à encore renforcer les connaissances techniques concernent le code en JavaScript. Il a également été créé un début de rapport de projet avec l’introduction, les objectifs, les technologies utilisées, l’installation et l’exécution, l’accès à la page du jeu.

*Séquence 3*

Cette séquence n’a malheureusement pas eu lieu à cause d’un accident à la main qui a empêché le travail.

*Séquence 4*

A partir d’un tutoriel Snake trouvé sur internet, il a été appris et compris beaucoup de choses. Il a été possible de code a fonction draw() du module main.js en ajoutant la logique de dessin du serpent, de la nourriture, de la vérification des collisions.

*Séquence 5*

Lundi du Jeûne fédéral, il y a eu congé ce jour-là.

*Séquence 6*

Il a été réécrit le code de la méthode draw() de main.js car le code du tutoriel trouvé était trop long et moins optimal. Les trois méthodes initSnake, moveSnake et drawSnake ont été codées grâce à une inspiration de code trouvé en ligne. Les étapes ont été rédigées dans le rapport de projet.

*Séquence 7*

Cette séquence n’a malheureusement pas eu lieu pour cause de maladie qui a empêché le travail.

*Séquence 8*

Ajout de la fonctionnalité de la pause.

Finalisation du projet en ajoutant les dernières modifications sur le rapport de projet ainsi que sur le code.

De part les nombreuses périodes d’absence et de congé, il a été obligatoire d’étudier ainsi que de coder en dehors des heures de travail, par exemple après les cours et pendant les vacances.

# Tests et résultats

Scénarios testés :

1. Affichage initial : canevas + nourriture, score = 0.
2. Déplacements : 4 directions OK, pas de demi-tour instantané.
3. Alimentation : contact tête/nourriture -> score +1 nouvelle nourriture, croissance du serpent.
4. Collisions : mur et corps -> message Game Over + arrêt du tick.
5. Redémarrage : F5 relance une partie propre.

Résultat : la version 1.0 est jouable, fluide et conforme au cahier des charges de base.

# Difficultés rencontrées et solutions

* Canevas vide : ordre de rendu corrigé (premier draw() avant setInterval).
* Message «head is not defined» : signatures des fonctions de collision corrigées pour accepter head.

# Contrôle de version (Git/GitHub)

* Dépôt privé GitHub
* Commits atomiques et messages normés (feat :, fix :, doc :).

# Recours à l’IA

L’IA a été utilisée comme aide pédagogique et technique pour :

* Expliquer des notions de code (grille box, canevas, déstructuration).
* Déboguer des erreurs (imports/exports, collisions, Live Server)
* Proposer une structure modulaire et des bonnes pratiques.
* Faciliter et accélérer la rédaction du rapport avec des phrases explicatives claires et précises.

Les décisions et validations finales ont été réalisées par l’apprenti.

# Conclusion

Générale

En réalisant ce Snake, l’objectif fixé a été atteint : livrer un jeu fonctionnel, clair et modulaire. Il a été su structurer un projet en JavaScript moderne avec des modules ESM, utiliser Vite pour accélérer le cycle de développement, et maîtriser le rendu 2D sur canevas. Une progression a eu lieu sur des aspects concrets de code en JavaScript comme la gestion des entrées clavier, mouvement par case, détection des collisions et séparation entre calcul et affichage. Les phases de débogage ont conduit à adopter des bonnes pratiques : fonctions courtes, JSDoc et tests manuels réguliers. Le résultat est une base saine et lisible, prête à évoluer.

Personnelle

Ce projet m’a appris à rester simple et régulier dans ma façon de travailler. J’ai pris l’habitude d’avancer par petites étapes, de tester souvent et de corriger calmement quand quelque chose ne marche pas. Malgré les nombreuses périodes de projet ratées à cause de absences, j’ai appris à rattraper mon retard en dehors des heures de projets. J’ai gagné en confiance, au lieu de me bloquer sur une erreur, je cherche, j’essaie, et le plus souvent je finis par trouver. C’était un bon projet.