

FACULTÉ DES SCIENCES ET TECHNIQUES

Atelier de programmation

Simulation comportementale d'un aquarium en C/GTK4 avec configuration externe

Réalisé par :

ELMAOUI Yahya

EL ASSALI Saad

JOUNAID Ayoub

BICHRI Yasser

6 août 2026

Résumé

Ce mémoire présente la conception et l'implémentation d'un aquarium virtuel animé écrit en C avec la bibliothèque graphique GTK4. Le système simule en temps réel un écosystème de poissons multi-espèces à partir d'un modèle comportemental dérivé des travaux de Reynolds sur le *flocking* (séparation, alignement, cohésion), enrichi de relations prédateur/proie, d'un cycle de vie complet (naissance, mort, respawn) et d'une mécanique d'alimentation basée sur la position de la *bouche* des prédateurs.

L'apport principal du projet est la mise en place d'un *système de configuration externe* fondé sur le format TOML. Ce système permet à un utilisateur non développeur d'ajouter une nouvelle espèce, d'en supprimer une, ou d'ajuster finement n'importe quel paramètre de simulation (vitesses, rayons de vol en banc, populations, prédateurs, etc.) en modifiant un seul fichier texte : `aquarium.conf`. Le système gère les chemins de recherche standards (`./`, `$XDG_CONFIG_HOME/`, `/etc/`), valide les références croisées entre espèces, fournit des messages d'erreur lisibles, et autorise le rechargement à chaud depuis l'interface graphique.

Sur le plan technique, le rapport détaille : l'architecture modulaire (modules `main`, `world`, `entity`, `species`, `conf`, `vec2`) ; la boucle de simulation à pas fixe basée sur `g_timeout_add` ; la gestion d'un pool d'entités à capacité bornée ; la résolution des références d'espèces par index dynamique permettant l'ajout d'espèces depuis le fichier de configuration sans recompilation ; l'utilisation de transformations CSS GTK4 pour la rotation des sprites ; et enfin l'intégration de la bibliothèque `tomlc99` comme parseur TOML embarqué.

Mots-clés : simulation comportementale, flocking, boids, GTK4, langage C, configuration TOML, prédateur/proie, écosystème virtuel.

Table des matières

Résumé	1
1 Introduction générale	8
1.1 Contexte	8
1.2 Problématique	8
1.3 Objectifs	9
1.4 Plan du mémoire	9
2 État de l’art	11
2.1 Le modèle de Reynolds : flocking et boids	11
2.1.1 Les trois règles canoniques	11
2.1.2 Combinaison et limites	11
2.2 Comportements de pilotage : au-delà des boids	12
2.3 Intégration à pas fixe vs pas variable	12
2.4 GTK4 comme moteur de rendu	13
2.5 Formats de configuration	13
2.6 Synthèse	14
3 Analyse et spécifications	15
3.1 Périmètre	15
3.2 Exigences fonctionnelles	15
3.3 Exigences non fonctionnelles	16
3.4 Cas d’utilisation	17
3.5 Glossaire	17
4 Conception	19
4.1 Architecture globale	19
4.2 Modèle de données	20
4.2.1 Entity	20
4.2.2 SpeciesConfig	20
4.2.3 World	21
4.3 Séquence d’un tick	21
4.4 Cycle de vie d’une entité	22
4.5 Graphe prédateur/proie	23
4.6 Forces de pilotage	23
4.7 Choix de conception remarquables	24
5 Implémentation	25
5.1 Environnement et chaîne de compilation	25
5.2 Pipeline d’assets	25
5.3 Algorithmes clés	26

5.3.1	Boucle de tick	26
5.3.2	Forces de Reynolds	26
5.3.3	Force de fuite avec adrénaline	27
5.3.4	Alimentation par la bouche	27
5.3.5	Cycle de vie : despawn et respawn	27
5.3.6	Famine	27
5.4	Système de configuration TOML	27
5.4.1	Structure du fichier	27
5.4.2	Parseur <code>tomlc99</code>	28
5.4.3	Résolution des références <code>prey</code>	28
5.4.4	Chemins de recherche et fallback	28
5.4.5	Variables globales et runtime	28
5.4.6	Rechargement à chaud	29
5.5	Rendu GTK4	29
5.5.1	Hiérarchie des widgets	29
5.5.2	Rotation par CSS	30
5.5.3	Effet de mort	30
5.6	Récapitulatif modulaire	31
6	Tests et résultats	38
6.1	Méthodologie	38
6.2	Scénarios fonctionnels	38
6.2.1	S1 – Démarrage avec configuration par défaut	38
6.2.2	S2 – Modification d’un paramètre simple	38
6.2.3	S3 – Suppression d’une espèce	39
6.2.4	S4 – Ajout d’une nouvelle espèce	39
6.2.5	S5 – Configuration invalide : référence inconnue	39
6.2.6	S6 – Configuration invalide : syntaxe TOML	39
6.2.7	S7 – Absence du fichier	40
6.2.8	S8 – Fallback XDG	40
6.2.9	S9 – Pause et reprise	40
6.2.10	S10 – Plein écran	40
6.3	Synthèse des résultats	41
6.4	Limites identifiées	41
7	Conclusion et perspectives	42
7.1	Bilan	42
7.2	Limites techniques	42
7.3	Perspectives	43
7.4	Apport personnel	44
	Bibliographie	45
A	Fichier de configuration complet	46

B	Instructions de build et d'exécution	51
B.1	Dépendances système	51
B.2	Commandes principales	51
B.3	Variables d'environnement utiles	51
C	Arborescence du projet	52
D	Tableau récapitulatif des espèces	53
E	Définition de la structure SpeciesConfig	54
F	Signature de l'API de configuration	56

Table des figures

3.1	Diagramme de cas d'utilisation.	17
4.1	Architecture modulaire et dépendances d'inclusion.	19
4.2	Séquence d'un tick de simulation.	21
4.3	Machine à états du cycle de vie d'une entité.	22
4.4	Chaîne alimentaire de l'écosystème par défaut.	23
4.5	Forces de séparation, alignement et cohésion appliquées à un boid.	24

Liste des tableaux

2.1	Comparaison des formats de configuration courants.	14
5.1	Volume de code par module.	31
6.1	Synthèse des scénarios de test.	41
7.1	Statut des exigences fonctionnelles.	42
D.1	Récapitulatif des huit espèces par défaut.	53

List of Listings

5.1	La fonction <code>world_tick()</code> (extrait de <code>src/world.c</code>).	26
5.2	Calcul des trois forces de banc (extrait de <code>src/entity.c</code>).	32
5.3	Force de fuite avec boost d'adrénaline (extrait de <code>src/entity.c</code>).	33
5.4	Calcul du point de bouche d'un prédateur.	34
5.5	Détection des repas (extrait de <code>src/world.c</code>).	34
5.6	Despawn d'une entité (extrait de <code>src/world.c</code>).	35
5.7	Logique de respawn (extrait de <code>src/world.c</code>).	36
5.8	Vérification de la famine.	37
5.9	Extrait de <code>aquarium.conf</code> : section globale et bloc <code>goldfish</code>	37

Chapitre 1

Introduction générale

1.1 Contexte

Les simulations comportementales sont au cœur de nombreux domaines de l’informatique : jeux vidéo, films d’animation, recherche en biologie artificielle, robotique en essaim, et applications grand public comme les *screensavers* animés. Elles cherchent à reproduire, par des règles locales simples appliquées à un grand nombre d’agents, des comportements globaux *émergents* qui paraîtraient, sans cela, extrêmement coûteux à scénariser explicitement.

Le cas particulier de la simulation d’un *aquarium* concentre plusieurs de ces problématiques : déplacement fluide d’agents hétérogènes, formation et dissolution de bancs, interactions prédateur/proie, gestion d’une population qui naît et meurt, et tout cela en temps réel avec un budget de calcul modeste. C’est un terrain d’expérimentation idéal pour mettre en pratique le modèle *boids* de Reynolds [7], qui reste, près de quarante ans après sa publication, la référence pour ce type de système.

Sur le plan technique, ce projet a également pour objectif de développer une application graphique native sous Linux à l’aide de la bibliothèque GTK4 [4]. Contrairement à un moteur de jeu généraliste qui imposerait son propre cadre, GTK est un *toolkit* d’interface graphique de bureau : l’utiliser pour piloter une simulation animée oblige à exploiter astucieusement ses primitives (positionnement absolu via `GtkFixed`, transformations CSS pour la rotation, flux multimédia pour la vidéo de fond) et à respecter son modèle d’événements basé sur la boucle principale GLib.

1.2 Problématique

Construire un aquarium virtuel crédible suppose de répondre à plusieurs questions techniques imbriquées :

1. **Comportement émergent.** Comment obtenir, à partir de règles locales (chaque poisson ne voit que ses voisins immédiats), un comportement global cohérent : bancs qui se forment, se déplacent ensemble, évitent les obstacles, paniquent à l’approche d’un prédateur ?
2. **Stabilité numérique.** Comment intégrer les forces qui s’exercent sur un poisson sans que la simulation diverge ou devienne saccadée ? La réponse passe par une boucle à pas fixe et un contrôle strict des accélérations et des vitesses.
3. **Écosystème.** Comment représenter de façon généralisable les relations alimentaires entre espèces ? Comment les espèces naissent, meurent et se renouvellent ?
4. **Adaptabilité.** Et surtout : comment permettre à un utilisateur non programmeur de modifier l’écosystème sans recompiler le programme ? L’expérience montre qu’un système qui n’expose pas ses paramètres devient rapidement figé, donc inutilisable pour l’expérimentation.

C’est sur ce dernier point que ce projet se distingue d’une implémentation académique classique du modèle de Reynolds : l’enjeu n’est pas seulement de simuler, mais de fournir un système *ouvert* où la liste des espèces, leurs paramètres comportementaux et les chaînes alimentaires sont décrits dans un fichier texte modifiable à chaud.

1.3 Objectifs

Les objectifs assignés au projet sont les suivants :

- Implémenter en C un moteur de simulation à pas fixe pour un écosystème de poissons, intégrant les trois forces de Reynolds (séparation, alignement, cohésion), une force de fuite face aux prédateurs, une force de chasse vers les proies, et une force d’évitement entre individus de la même espèce.
- Construire l’interface utilisateur avec GTK4 : fenêtre plein écran possible, vidéo de fond en boucle, contrôles latéraux (vitesse globale, pause, ajout et retrait précis de poissons, sélection des espèces du *pool*, rechargement de la configuration).
- Concevoir un **système de configuration externe** en TOML [6] qui décrit l’intégralité du paramétrage : toutes les espèces, toutes leurs propriétés, et les constantes globales du moteur. Cette configuration doit :
 - permettre l’ajout/suppression d’une espèce sans toucher au code C ;
 - résoudre dynamiquement les références entre espèces (**prey** = liste d’identifiants) ;
 - proposer des chemins de recherche standards et un mécanisme de *fallback* ;
 - être rechargeable à chaud depuis l’interface.
- Garantir la *robustesse* de l’application face à une configuration invalide : erreurs de syntaxe TOML, identifiants manquants, références **prey** inconnues, valeurs hors plage. Le programme doit échouer proprement avec un message lisible plutôt que de produire un comportement indéfini.

1.4 Plan du mémoire

Ce rapport est organisé en sept chapitres :

- Le **chapitre 2** présente l’état de l’art utile à la compréhension du projet : le modèle de Reynolds, l’évolution vers des comportements de pilotage plus riches, les choix d’intégration numérique et un panorama comparatif des formats de configuration usuels.
- Le **chapitre 3** formalise les spécifications fonctionnelles et non fonctionnelles du système et présente les principaux cas d’utilisation.
- Le **chapitre 4** décrit l’architecture modulaire retenue, le modèle de données, la séquence d’un *tick* de simulation, le cycle de vie d’une entité et le graphe prédateur/proie.
- Le **chapitre 5** détaille l’implémentation : environnement et chaîne de compilation, pipeline d’assets, algorithmes clés (forces de Reynolds, chasse, fuite, alimentation par la bouche, gestion des respawns), et enfin le système de configuration TOML.

- Le **chapitre 6** présente les scénarios de test menés et leurs résultats.
- Le **chapitre 7** dresse le bilan du projet et propose des pistes d'amélioration.
- Une bibliographie et plusieurs annexes (fichier de configuration complet, instructions de *build*, arborescence du projet, tableau récapitulatif des espèces) closent le document.

Chapitre 2

État de l’art

2.1 Le modèle de Reynolds : flocking et boids

En 1987, Craig W. Reynolds publie au SIGGRAPH un article fondateur intitulé *Flocks, herds and schools : a distributed behavioral model* [7]. Il y propose un modèle de simulation de groupes d’animaux (volées d’oiseaux, troupeaux, bancs de poissons) qui repose sur trois constats :

1. Le comportement de groupe ne provient pas d’un *leader* qui dicterait la direction commune.
2. Chaque agent (que Reynolds appelle un *boid*) prend ses décisions à partir d’observations *purement locales* : il ne voit que ses voisins dans un voisinage défini.
3. Les règles locales sont peu nombreuses et simples, mais le comportement global qui en émerge est riche et crédible.

2.1.1 Les trois règles canoniques

Reynolds définit trois règles, appliquées indépendamment puis combinées en une accélération unique :

Séparation. Évite les collisions avec les voisins trop proches. Mathématiquement, on accumule une force inversement proportionnelle à la distance entre l’agent et chacun de ses voisins proches :

$$\vec{F}_{\text{sep}} \propto - \sum_{i \in V_{\text{proches}}} \frac{\vec{r}_i}{\|\vec{r}_i\|^2}$$

où $\vec{r}_i = \vec{p}_i - \vec{p}_{\text{self}}$ est le vecteur allant de l’agent au voisin i .

Alignement. Aligne sa propre vitesse sur la moyenne des vitesses des voisins :

$$\vec{F}_{\text{ali}} \propto \vec{v}_V - \vec{v}_{\text{self}}$$

Cohésion. Tend à se rapprocher du *centre de masse* local des voisins :

$$\vec{F}_{\text{coh}} \propto \vec{p}_V - \vec{p}_{\text{self}}$$

2.1.2 Combinaison et limites

Les trois forces sont additionnées, chacune pondérée par un coefficient (`separation_weight`, etc.), puis l’accélération résultante est bornée par une *force maximale* `max_force` afin d’éviter des changements brusques. La vitesse résultante est elle-même bornée par `max_speed`, et un seuil minimal `min_speed` maintient les agents en mouvement.

Cette construction présente plusieurs propriétés intéressantes :

- Elle est *distribuée* : chaque boid décide pour lui-même, sans coordination centrale.
- Elle est *paramétrable* : en jouant sur les rayons et les poids, on obtient des comportements visuellement très différents (banc serré, nuée diffuse, individus solitaires).
- Elle a une complexité naïve en $O(N^2)$ (chaque agent examine chaque autre), améliorable par des structures spatiales de type *spatial hashing* ou *octree* [3], mais cette complexité reste acceptable pour $N \lesssim 200$ comme dans notre cas.

2.2 Comportements de pilotage : au-delà des boids

Le modèle initial de Reynolds décrit trois forces de groupe, mais laisse de côté les interactions inter-espèces. Dans un article ultérieur [8], Reynolds généralise sous le nom de *steering behaviors* (comportements de pilotage) une famille plus large de forces utiles pour les agents autonomes : *seek* (aller vers une cible), *flee* (s’éloigner), *pursue* (intercepter une cible mobile), *wander* (errer aléatoirement), *obstacle avoidance*, etc. Millington et Funge approfondissent cette taxonomie dans le contexte du jeu vidéo [5].

Pour notre aquarium, les forces ajoutées au-delà des trois forces classiques sont les suivantes :

- **Fuite (*flee*)**. Une proie s’éloigne d’un prédateur visible dans son rayon de peur `fear_radius`, avec un *boost* d’adrénaline `ADRENALINE_BOOST` qui multiplie temporairement sa vitesse maximale.
- **Chasse (*seek*)**. Un prédateur se dirige vers sa proie la plus proche dans son rayon de chasse `hunt_radius`, à condition que l’espèce de la proie figure dans son masque `prey_mask`.
- **Évitement intra-espèce**. Une force d’écartement supplémentaire entre individus d’une même espèce, au-delà de la séparation locale, modélisée par `avoid_same_radius`. Cette force évite que des prédateurs solitaires (requin, barracuda) se collent les uns aux autres.

2.3 Intégration à pas fixe vs pas variable

Toute simulation continue échantillonnée doit choisir un schéma d’intégration numérique. Les deux approches usuelles sont :

Pas variable. À chaque image, on calcule le délai Δt écoulé depuis l’image précédente, puis on intègre les positions et vitesses sur cet intervalle. Avantage : la simulation suit le rythme réel, peu importe que le rendu prenne 16 ms ou 50 ms. Inconvénient majeur : le comportement devient *non déterministe* et numériquement instable lorsque Δt varie fortement (typiquement à cause d’un *stutter* système).

Pas fixe. On choisit un pas Δt constant (par exemple 16 ms, soit environ 60 Hz) et on appelle la fonction de mise à jour à intervalles réguliers. Avantage : déterminisme parfait, stabilité numérique, code simple. Inconvénient : si le rendu prend plus longtemps que Δt , le système peut prendre du retard. Cette limitation se gère par un mécanisme de *rattrapage* (faire plusieurs ticks de simulation par image de rendu) si nécessaire.

Pour notre projet, le compromis bénéfice/complexité est net : le pas fixe est implémenté trivialement avec `g_timeout_add(TICK_MS, ...)` de GLib [2], et la simulation reste fluide tant que le coût de calcul d’un tick reste inférieur à `TICK_MS`.

2.4 GTK4 comme moteur de rendu

GTK4 [4] est le *toolkit* graphique du projet GNOME. Il s’agit d’une bibliothèque d’interface utilisateur de bureau, et non d’un moteur de jeu. L’utiliser pour piloter une simulation animée est un choix non standard qui mérite quelques explications.

Arguments en faveur de GTK4. Bibliothèque native, écrite en C, intégration parfaite avec l’environnement Linux, gestion automatique de la fenêtre, du plein écran, des entrées clavier, et accès à un système de *CSS* pour styler les widgets, y compris des transformations affines (rotation, mise à l’échelle). La présence du widget `GtkPicture` pour afficher une image fixe et la possibilité de placer ces widgets en coordonnées absolues via `GtkFixed` permettent d’obtenir un rendu 2D animé sans dépendre d’OpenGL ou d’une bibliothèque tierce.

Arguments contre. GTK n’est pas optimisé pour des centaines de widgets manipulés à 60 Hz. Chaque entité de l’aquarium étant un widget GTK distinct, on charge le système avec N recalculs de style CSS par image. En pratique, à 95 poissons, les performances restent acceptables sur du matériel moderne, mais ce choix limiterait la simulation à quelques centaines d’agents au plus. Une version future utilisant Cairo ou GTK *snapshot* pour dessiner directement les sprites en une seule passe gagnerait significativement en performance.

2.5 Formats de configuration

Le cœur du projet est de permettre la modification de paramètres sans recompilation. Cela impose de choisir un format de configuration. Quatre familles ont été envisagées :

INI. Très répandu, simplissime à parser, lisible. Faiblesses : pas de structures imbriquées standard, pas de listes natives, types implicites (tout est chaîne).

JSON. Universel, beaucoup d’outils, types stricts (nombres, booléens, listes). Faiblesses pour un fichier *édité à la main* : pas de commentaires, virgules trompeuses, guillemets obligatoires partout, parenthèses imbriquées qui rendent la structure visuelle confuse.

YAML. Lisible, permet les commentaires, listes natives. Faiblesses : spécification très complexe (plusieurs ambiguïtés célèbres comme *Norway problem*), parseur lourd, sensibilité à l’indentation par espaces qui pénalise les éditeurs non-techniques.

TOML. *Tom’s Obvious, Minimal Language* [6], format conçu spécifiquement pour les fichiers de configuration. Avantages décisifs pour notre cas :

— Commentaires natifs avec `#`.

- Sections claires `[global]`, tableaux de tables `[[species]]`.
- Types stricts (entier, flottant, booléen, chaîne, tableau, date).
- Spécification courte et sans ambiguïté.
- Parseurs disponibles dans la plupart des langages, en particulier `tomlc99` [9], une implémentation en C99 distribuée sur deux fichiers (`toml.h` et `toml.c`), facile à *vendor* iser dans un projet C sans dépendance externe.

Le tableau 2.1 récapitule cette comparaison.

Critère	INI	JSON	YAML	TOML
Commentaires natifs	oui	non	oui	oui
Listes natives	non	oui	oui	oui
Types stricts	non	oui	oui	oui
Tables imbriquées	non	oui	oui	oui
Édition manuelle	++	–	+	++
Parseur C minimaliste	oui	oui	non	oui
Spécification courte	oui	oui	non	oui

TABLE 2.1 – Comparaison des formats de configuration courants.

2.6 Synthèse

Le projet se positionne donc à l’intersection de trois traditions : le modèle de Reynolds pour le comportement, l’écosystème de développement GTK pour le rendu, et la philosophie *Unix* de configuration par fichier texte (avec chemins de recherche en `./`, `~/.config/` et `/etc/`) pour l’extensibilité. Chacun de ces piliers est suffisamment stable et documenté pour servir de fondation ; la valeur ajoutée du projet réside dans leur intégration cohérente.

Chapitre 3

Analyse et spécifications

3.1 Périmètre

L'application cible est un *screensaver* interactif présentant un aquarium peuplé de plusieurs espèces de poissons. Elle s'exécute sur un poste Linux avec un environnement graphique compatible GTK4. Elle est mono-utilisateur, mono-fenêtre, et n'a pas de persistance : chaque démarrage recharge la configuration depuis le disque et réinitialise la simulation.

3.2 Exigences fonctionnelles

Les exigences fonctionnelles, numérotées EF1..EF12, décrivent ce que le système doit faire.

EF1 – Simulation de banc.

L'application doit simuler le comportement de banc d'au moins sept espèces de poissons distinctes, en appliquant à chacune les trois forces de Reynolds (séparation, alignement, cohésion) avec des paramètres propres.

EF2 – Prédateur/proie.

Certaines espèces sont des prédateurs et chassent d'autres espèces désignées comme leurs proies. La consommation d'une proie est effective lorsque la *bouche* du prédateur entre en collision avec le centre de la proie.

EF3 – Fuite des proies.

Toute proie qui détecte un prédateur dans son rayon de peur (`fear_radius`) doit s'enfuir en sens opposé, avec un *boost* de vitesse d'adrénaline.

EF4 – Cycle de vie.

Une entité passe par les états `ALIVE`, `DYING` (transition visuelle de ~ 0.22 seconde) puis `INACTIVE` (invisible, en attente). Une entité `INACTIVE` peut être réactivée au-delà du délai `respawn_delay`.

EF5 – Famine.

Un prédateur qui n'a rien mangé depuis `STARVATION_SECONDS` secondes meurt de faim.

EF6 – Configuration externe.

L'intégralité des paramètres (constantes globales et tableau des espèces) est lue au démarrage depuis un fichier TOML `aquarium.conf`.

EF7 – Recherche multi-niveaux.

Le fichier de configuration est cherché successivement dans le répertoire courant, puis dans `~/config/gtkaqua/`, puis dans `/etc/gtkaqua/`.

EF8 – Ajout/suppression d'espèces.

Ajouter une espèce consiste à ajouter un bloc `[[species]]` au fichier de configuration et

à fournir le sprite correspondant dans le répertoire `assets/`, sans modifier le code C ni recompiler.

EF9 – Validation.

Le chargement doit échouer proprement en cas d'erreur de syntaxe TOML, d'identifiant manquant, ou de référence `prey` pointant vers une espèce inexistante, en affichant un message lisible.

EF10 – Rechargement à chaud.

Un bouton *Reload config* dans l'interface relit le fichier de configuration et relance la simulation avec les nouveaux paramètres, sans redémarrer l'application.

EF11 – Contrôles utilisateur.

L'interface latérale fournit : une case à cocher par espèce pour décider qui peuple le *pool* initial, un sélecteur d'espèce et un compteur pour ajouter/retirer un nombre précis de poissons à chaud, un bouton pause, un curseur de vitesse globale, un bouton *Initialize pool*, le bouton *Reload config*.

EF12 – Plein écran.

La touche F11 bascule en mode plein écran. La touche Échap quitte le plein écran.

3.3 Exigences non fonctionnelles

ENF1 – Performance.

La simulation doit tenir 60 Hz avec au moins 100 entités sur un poste de bureau moderne (CPU quad-core, GPU intégré).

ENF2 – Portabilité.

Cible Linux x86_64 avec GTK4 et GLib installés. Pas d'objectif de portage Windows ou macOS.

ENF3 – Lisibilité de la configuration.

Le fichier `aquarium.conf` doit être éditable par un utilisateur qui n'est pas développeur : noms de paramètres explicites, commentaires de section, valeurs typées (entiers, flottants, listes de chaînes), pas de syntaxe spécifique au langage C.

ENF4 – Robustesse mémoire.

Pas de fuite de mémoire en régime stationnaire. Les widgets GTK des entités `INACTIVE` sont conservés et masqués plutôt que détruits et recréés, pour éviter la fragmentation et les coûts d'allocation répétés.

ENF5 – Modularité.

Chaque module C correspond à un couple `.h/.c` clairement délimité, et expose une API minimale. La taille moyenne d'un module reste raisonnable (quelques centaines de lignes).

ENF6 – Reproductibilité du build.

La compilation se fait par `make` sans télécharger de dépendances réseau au moment du build. Le seul *vendor* embarqué est la bibliothèque `tomlc99`, copiée dans `vendor/tomlc99/`.

ENF7 – Diagnostic.

En cas d'erreur de chargement de configuration, un message clair indique le chemin essayé

et la cause (clé manquante, identifiant de proie inconnu, etc.).

3.4 Cas d'utilisation

Cinq cas d'utilisation principaux structurent l'interaction avec l'application. Ils sont représentés graphiquement dans la figure 3.1.

- **UC1 – Lancer la simulation.** L'utilisateur démarre l'application. Le programme lit `aquarium.conf`, instancie les espèces, peuple le *pool* initial selon `flock_size` de chaque espèce, démarre la boucle de simulation.
- **UC2 – Mettre en pause.** L'utilisateur clique sur *Pause*. Le tick continue d'être appelé mais `world_tick` n'est plus exécuté.
- **UC3 – Modifier l'écosystème via configuration.** L'utilisateur édite `aquarium.conf` (modification d'un paramètre, ajout/suppression d'un bloc `[[species]]`), enregistre, puis clique sur *Reload config* ou redémarre l'application.
- **UC4 – Ajouter/retirer des poissons à chaud.** L'utilisateur sélectionne une espèce dans la combobox, saisit un nombre, et clique sur *+ Add* ou *- Remove*.
- **UC5 – Recharger la configuration.** L'utilisateur clique sur *Reload config*. Le système relit le fichier, valide, et relance la simulation. En cas d'erreur, une boîte de dialogue s'affiche.

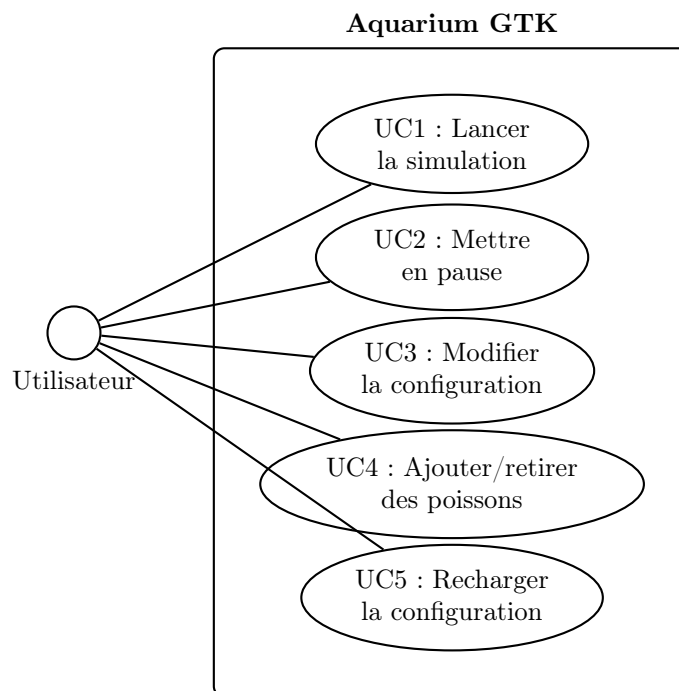


FIGURE 3.1 – Diagramme de cas d'utilisation.

3.5 Glossaire

Boid.

Agent autonome dans le modèle de Reynolds. Dans notre projet, un boid est une instance

de `Entity`.

Flocking.

Comportement de banc émergeant de l'application des trois règles de Reynolds.

Mouth-based eating.

Approche d'alimentation où la collision est calculée entre la *bouche* du prédateur (point décalé vers l'avant du sprite, à `mouth_forward_ratio` $\times$ `sprite_width`) et la position centrale de la proie. Plus réaliste qu'une collision centre-à-centre.

Lifespan.

Durée de vie maximale d'une entité, exprimée en secondes. Atteinte la durée, l'entité passe à `DYING`.

Respawn.

Réactivation d'une entité `INACTIVE` après expiration de `respawn_delay` et tant que la population vivante de l'espèce est inférieure à `flock_size`.

Pool.

Ensemble des espèces effectivement peuplées au démarrage, déterminé par les cases à cocher de l'interface.

Pred mask.

Bitmask 32 bits où le bit i indique que l'espèce d'index i est une proie de l'espèce courante.

Chapitre 4

Conception

4.1 Architecture globale

L'application est découpée en six modules C, plus une bibliothèque *vendor* pour le parsing TOML. Chaque module possède un fichier d'en-tête (dans `include/`) qui expose son API publique, et un fichier source (dans `src/`) qui contient l'implémentation. La figure 4.1 présente le graphe des inclusions.

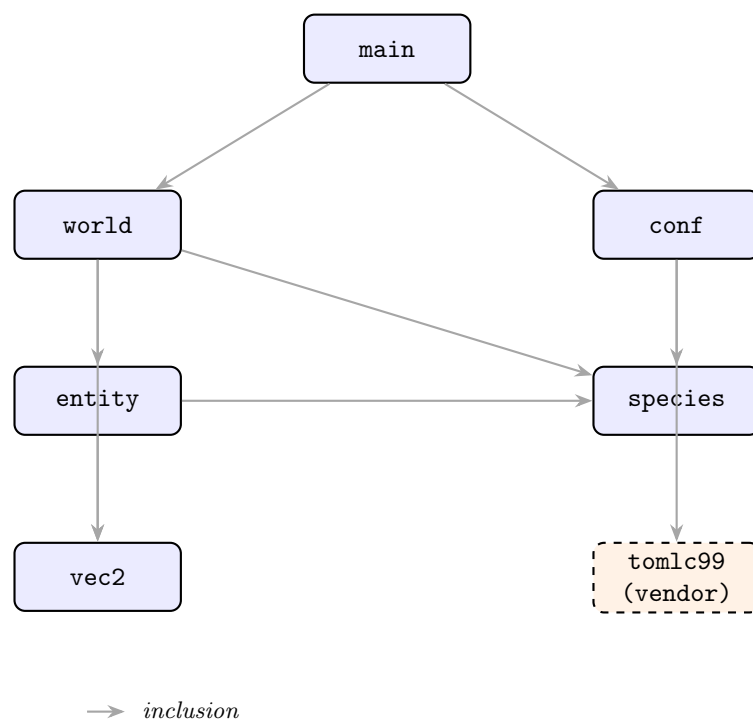


FIGURE 4.1 – Architecture modulaire et dépendances d'inclusion.

Rôle de chaque module :

main. Point d'entrée de l'application. Crée la fenêtre GTK, met en place les contrôles latéraux, charge la configuration via `conf`, initialise le `World` et enregistre le *tick* via `g_timeout_add()`.

world.

Possède le pool d'entités, gère leur spawn/despawn, orchestre la séquence de mise à jour (déplacement, alimentation, vieillissement, respawn), et expose la fonction `world_tick()` appelée à chaque image.

entity.

Représente une entité individuelle. Implémente les forces de pilotage (boids, fuite, chasse,

évitement intra-espèce), l'intégration physique semi-implicite, le lissage de l'angle de cap, et la mise à jour visuelle (positionnement et CSS).

species.

Registre dynamique des **SpeciesConfig**. Stocke en mémoire le tableau des espèces issu de la configuration et fournit les fonctions de recherche par index ou par identifiant.

conf. Lit **aquarium.conf** via le parseur TOML, valide le contenu, résout les listes **prey** en bitmask, et publie les paramètres globaux dans des variables **extern**.

vec2. Bibliothèque *header-only* de vecteurs 2D : addition, soustraction, multiplication par scalaire, normalisation, limitation, *wrapping* torique pour le monde, et calcul de delta minimal en distance enveloppée.

tomlc99 (vendor).

Parseur TOML en C99 distribué en deux fichiers (**toml.h**, **toml.c**), embarqué dans **vendor/tomlc99/**.

4.2 Modèle de données

Trois structures principales portent l'état du système.

4.2.1 Entity

Une **Entity** représente une instance vivante (ou en sursis) dans l'aquarium. Elle agrège trois groupes de champs :

- **Identité et appartenance** : **id** unique, **species** (index dans le registre).
- **État physique** : position **pos**, vitesse **vel**, accélération **acc**, angle de cap **angle** (lissé), facteur de variation individuel **speed_scale**.
- **Cycle de vie et rendu** : **state** (ALIVE, DYING, INACTIVE), **age**, **last_eaten**, **respawn_at**, **death_timer**, plus le widget GTK et le **css_provider** associés.

4.2.2 SpeciesConfig

Cette structure regroupe l'intégralité des paramètres gouvernant le comportement d'une espèce : identifiant **id**, nom affiché, chemin du sprite, dimensions du sprite, ratio de bouche, vitesses min/max, force max, rayons et poids des trois forces de Reynolds, rayon et poids de fuite, rayon et poids de chasse, rayon et poids d'évitement intra-espèce, masque de proie, durée de vie, taille initiale du banc, plafond de population, délai de respawn, et drapeau **is_apex**.

L'ensemble des paramètres comportementaux est ainsi capturé dans un *seul* type, ce qui simplifie considérablement le code consommateur : **entity_tick()** reçoit un pointeur **const SpeciesConfig *** et lit les paramètres dont il a besoin, sans avoir à connaître le système de configuration.

4.2.3 World

Le **World** contient le pool d'entités en tableau statique **Entity entities[MAX_ENTITIES]**, le compteur courant, les dimensions de la zone simulée, le pointeur vers le conteneur GTK, le pas de temps courant **dt**, le temps écoulé **elapsed**, le générateur d'identifiants **next_id**, ainsi que deux tableaux indexés par espèce : **alive_counts[MAX_SPECIES]** et **respawn_ready[MAX_SPECIES]**.

4.3 Séquence d'un tick

À chaque pas de simulation, **world_tick()** orchestre une séquence ordonnée d'étapes. La figure 4.2 représente cet enchaînement.

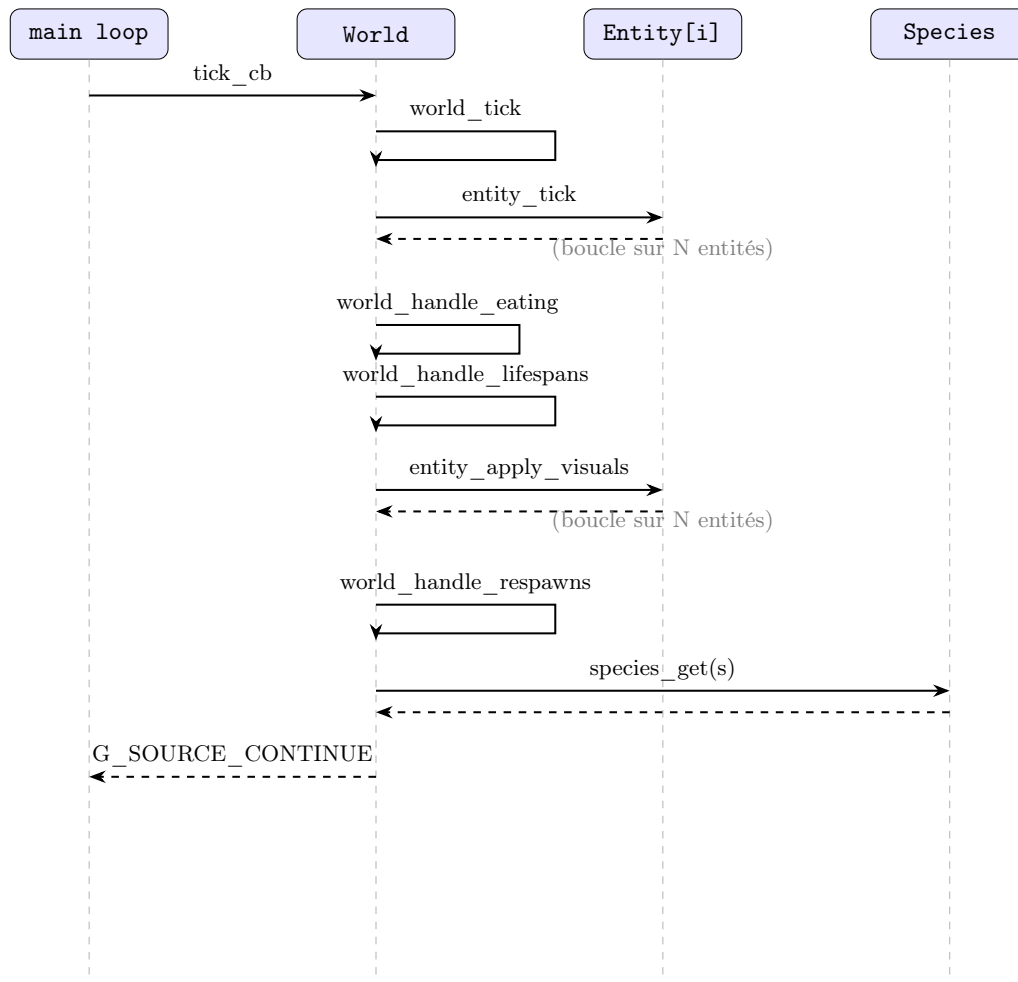


FIGURE 4.2 – Séquence d'un tick de simulation.

Chronologiquement :

1. Avancement du temps : **elapsed += dt**.
2. Pour chaque entité **ALIVE**, calcul des forces et mise à jour de la position et de la vitesse via **entity_tick()**.
3. Détection et application des *repas* : **world_handle_eating()** parcourt tous les couples prédateur/proie compatibles et déclenche un **world_despawn()** si la bouche du prédateur entre dans le rayon de hit de la proie.

4. Vérification de la famine : une entité qui n'a pas mangé depuis `STARVATION_SECONDS` secondes est *despawnée*.
5. Pour chaque entité, `entity_apply_visuals()` actualise la position du widget et le CSS de rotation. Si l'entité est en `DYING`, le timer décroît jusqu'à `INACTIVE`.
6. Tentative de respawn pour chaque espèce dont `alive_counts` est inférieur à `flock_size`.

L'ordre des étapes est important. La détection des repas *après* l'intégration physique garantit que la collision est calculée sur les positions à jour. La mise à jour visuelle *après* la détection des repas permet d'afficher le flash de mort dès le tick suivant la consommation. Le respawn en dernière étape évite de réactiver une entité qui aurait été mangée le même tick.

4.4 Cycle de vie d'une entité

La figure 4.3 formalise les trois états et leurs transitions.

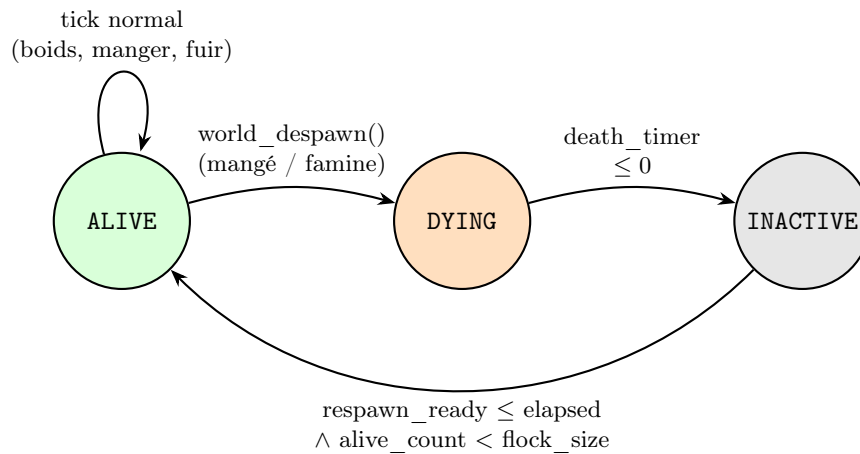


FIGURE 4.3 – Machine à états du cycle de vie d'une entité.

ALIVE → ALIVE.

Cas nominal. L'entité est mise à jour à chaque tick.

ALIVE → DYING.

Déclenché par `world_despawn()`, qui peut être appelé soit par `world_handle_eating()` (mort par prédation) soit par `world_handle_lifespans()` (mort par famine). Le compteur `alive_counts[species]` est décrémenté, et `death_timer` est initialisé à `DEATH_FLASH_DURATION`.

DYING → INACTIVE.

À chaque tick en `DYING`, `death_timer` décroît de `dt`. Lorsqu'il atteint zéro, le widget est masqué et l'état passe à `INACTIVE`.

INACTIVE → ALIVE.

Si la population vivante est inférieure à `flock_size` *et* que `elapsed` dépasse `respawn_ready[species]`, `world_handle_respawns()` réactive une entité `INACTIVE` de cette espèce : position et vitesse aléatoires, état `ALIVE`, widget rendu visible, et `respawn_ready` repoussé de `respawn_delay`.

Le choix d'un *recyclage* d'entités plutôt qu'une allocation/destruction permet de borner statiquement le tableau `entities` à `MAX_ENTITIES` et d'éviter toute fragmentation : les widgets GTK sont créés une fois pour toutes au moment du *spawn*, puis réutilisés.

4.5 Graphe prédateur/proie

La configuration par défaut implémente une chaîne alimentaire à trois niveaux représentée dans la figure 4.4.

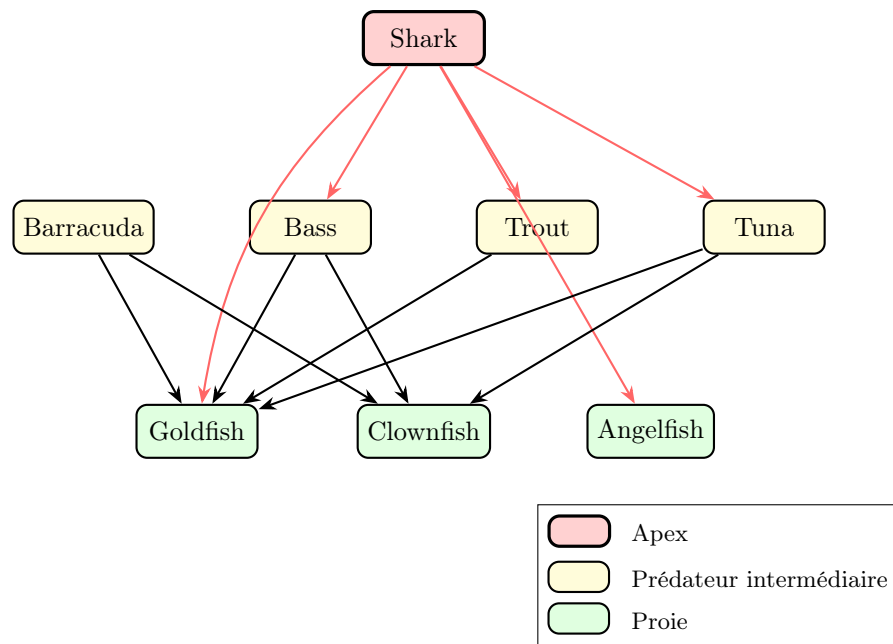


FIGURE 4.4 – Chaîne alimentaire de l'écosystème par défaut.

Le requin (`shark`) est l'unique prédateur *apex* (drapeau `is_apex = true`, aucun `fear_radius`). Il chasse cinq espèces. Quatre prédateurs intermédiaires (`barracuda`, `bass`, `trout`, `tuna`) chassent les petites proies (`goldfish`, `clownfish`) et craignent le requin. Trois espèces de proies pures (`goldfish`, `clownfish`, `angelfish`) ne chassent personne.

L'angelfish a un statut particulier : il n'est chassé que par le requin, ce qui en fait une proie résiliente, intermédiaire entre les petites proies (massivement chassées) et les prédateurs.

4.6 Forces de pilotage

La figure 4.5 illustre comment les trois forces de Reynolds s'appliquent à un agent. Les forces supplémentaires (fuite, chasse, évitement intra-espèce) sont construites sur le même schéma vectoriel mais avec des voisinages et des cibles différents.

Toutes les forces partagent le même *pattern* de calcul, ce qui se reflète directement dans le code de `src/entity.c` (chapitre 5) :

1. Parcourir les voisins compatibles dans un rayon donné.
2. Accumuler un vecteur *désiré* selon la règle.
3. Calculer un vecteur de *steering* = désiré – vitesse courante, borné par `max_force`.

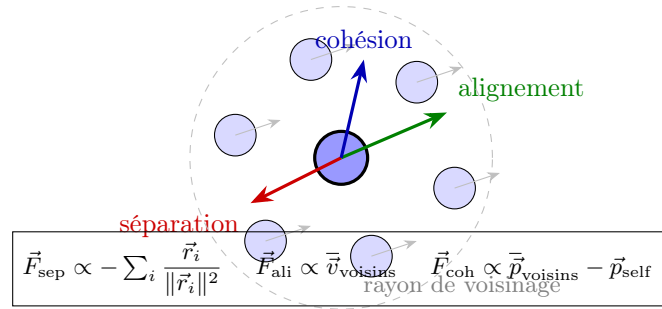


FIGURE 4.5 – Forces de séparation, alignement et cohésion appliquées à un boid.

4. Pondérer par le poids de la force et accumuler dans l'accélération totale.

L'accélération totale est ensuite bornée à $4 \times \text{max_force}$ pour absorber les pics, intégrée dans la vitesse, qui est elle-même bornée à `max_speed`.

4.7 Choix de conception remarquables

Position centre-based. La position d'une entité est le centre de son sprite, pas son coin haut-gauche. Le coin de dessin est dérivé au moment du rendu. Cela simplifie tous les calculs de distance et de collision.

Bouche dérivée du cap. Le point de bouche est calculé par `pos + sprite_width * mouth_forward_ratio * dir`, où `dir` est le vecteur unitaire de la vitesse. Cela rend l'alimentation visuellement crédible (le poisson doit *faire face* à sa proie pour la manger), sans coût de calcul significatif.

Indices d'espèces dynamiques. À l'origine, les espèces étaient identifiées par une `enum SpeciesKind` définie au moment de la compilation. Cette approche figeait la liste des espèces. Le passage à un index entier dynamique (`0..species_count-1`) résolu par `species_find_by_id()` a permis de découpler la liste des espèces du code C : ajouter une espèce ne demande plus de toucher à `species.h`.

Bitmask 32 bits pour prey_mask. Avec `MAX_SPECIES = 32`, un `int` suffit pour décrire l'ensemble des proies d'un prédateur. Le test "cette entité j est-elle une proie de l'entité i ?" se réduit alors à `prey_mask_i & (1 << species_j)`, donc à une opération en temps constant. C'est essentiel car ce test est exécuté à chaque couple (i, j) d'entités, soit en $O(N^2)$ par tick.

Vendoring de tomlc99. La bibliothèque TOML n'est pas une dépendance système : elle est copiée dans `vendor/tomlc99/` et compilée avec le projet. Cela garantit la reproductibilité du *build* et l'absence de téléchargement réseau au moment de la compilation.

Chapitre 5

Implémentation

5.1 Environnement et chaîne de compilation

Le projet est écrit en C 11 et compilé avec `gcc` (ou `clang`) en utilisant un `Makefile` GNU classique. Les options de compilation par défaut sont :

```
1 CFLAGS ?= -std=c11 -O2 -Wall -Wextra -Wpedantic
```

Les drapeaux GTK4 sont obtenus dynamiquement via `pkg-config` :

```
1 GTK_CFLAGS := $(shell $(PKG_CONFIG) --cflags gtk4)
2 GTK_LIBS   := $(shell $(PKG_CONFIG) --libs  gtk4)
```

L'arborescence du *build* est :

- `build/obj/` reçoit les fichiers objets ;
- `build/bin/aquarium` est le binaire final ;
- `vendor/tomlc99/` contient `toml.h`, `toml.c` et la licence MIT du parseur.

Une cible `install-conf` du `Makefile` copie `aquarium.conf` dans `~/.config/gtkaquar/` pour faciliter l'usage du *fallback* système.

5.2 Pipeline d'assets

Les sprites bruts vivent dans `Fish/` (formats `webp` ou `png`). La cible `make assets` les normalise vers `assets/` en garantissant qu'ils sont tous orientés vers la gauche. Pour les fichiers d'origine déjà gauche, la commande se limite à un redimensionnement ; pour les fichiers déjà droite, un `flip horizontal` est appliqué via `ffmpeg` [1] :

```
1 ffmpeg -i $(FISH_DIR)/shark.webp -vf "scale=100:-1" \
2     -frames:v 1 -y $(ASSET_DIR)/shark.png
3 ffmpeg -i $(FISH_DIR)/fish1Texture.png -vf "hflip" \
4     -frames:v 1 -y $(ASSET_DIR)/goldfish.png
```

Cette discipline d'orientation est documentée dans le `README.md` : au moment du runtime, le code suppose que tous les sprites pointent à gauche. La rotation de chaque sprite est ensuite produite par une transformation CSS GTK4 sur le widget, sans réchantillonnage de l'image.

5.3 Algorithmes clés

Cette section présente les fonctions centrales du moteur de simulation. Pour des raisons de place, on extrait les passages significatifs ; le code complet est disponible en annexe et dans le dépôt.

5.3.1 Boucle de tick

La fonction `world_tick()` orchestre une itération complète de la simulation. Elle est listée dans le code 5.1.

```
298 }
299
300 void world_tick(World *w) {
301     w->elapsed += w->dt;
302
303     for (int i = 0; i < w->entity_count; i++) {
304         entity_tick(w, i);
305     }
306
307     world_handle_eating(w);
308     world_handle_lifespans(w);
309
310     for (int i = 0; i < w->entity_count; i++) {
311         entity_apply_visuals(w, i);
312     }
313 }
```

Listing 5.1 – La fonction `world_tick()` (extrait de `src/world.c`).

L'ordre est important : déplacement physique, puis interactions (repas, famine), puis mise à jour visuelle, puis gestion des respawns. Cette séquence évite les races logiques entre événements du même tick.

5.3.2 Forces de Reynolds

Le calcul des trois forces de banc (séparation, alignement, cohésion) constitue le cœur algorithmique du moteur. Le code parcourt tous les voisins de la *même* espèce dans le rayon correspondant à chaque force, accumule les contributions, puis construit un vecteur de *steering* pondéré par les coefficients de la `SpeciesConfig`.

Trois remarques sur ce code :

- Les distances sont comparées au carré (`dist_sq < radius * radius`), ce qui évite un `sqrt` par voisin et constitue une optimisation classique.
- L'*enveloppement torique* du monde est pris en compte via `vec2_delta_wrap()`, qui choisit le delta minimum entre deux points à travers les bords.
- Le vecteur de séparation est pondéré par $1/\|\vec{r}\|^2$, ce qui rend la force d'autant plus forte que les voisins sont proches : c'est la formulation classique de Reynolds.

5.3.3 Force de fuite avec adrénaline

La fuite face à un prédateur est implémentée par `force_flee()`. Lorsqu’au moins un prédateur entre dans le `fear_radius` de l’agent, sa vitesse maximale et sa force maximale sont multipliées par le facteur `g_adrenaline_boost` (2.5 par défaut), ce qui rend la fuite beaucoup plus rapide que le déplacement nominal.

Le test `EATS()` sur la ligne pivot de cette fonction reflète exactement la définition donnée dans `species.h` :

```
1 #define EATS(predator_cfg, prey_index) \
2     ((predator_cfg).prey_mask & (1 << (prey_index)))
```

5.3.4 Alimentation par la bouche

Plutôt que de tester une collision centre-à-centre entre prédateur et proie, le projet utilise une *mouth-based collision* : le point d’attaque est dérivé du cap du prédateur, selon le ratio `mouth_forward_ratio`.

Ce point est ensuite comparé à la position centrale de chaque proie compatible. La proie est mangée si la distance enveloppée est inférieure à `g_eat_radius + prey_hit`, où `prey_hit` dépend de la taille du sprite de la proie. La formule pour `prey_hit` accorde un rayon de hit proportionnel à la plus petite dimension du sprite, mais avec un plancher de 8 pixels pour ne pas pénaliser les sprites microscopiques.

5.3.5 Cycle de vie : despawn et respawn

`world_despawn()` fait passer une entité de `ALIVE` à `DYING`, décrémente le compteur d’espèce, arme le timer de flash de mort, et programme la prochaine fenêtre de respawn.

Symétriquement, `world_handle_respawns()` parcourt les espèces dont la population est inférieure à `flock_size`, attend la fenêtre `respawn_ready`, puis recycle une entité `INACTIVE` en lui assignant une nouvelle position et une nouvelle vitesse aléatoires.

5.3.6 Famine

La famine est délibérément simple : un prédateur qui n’a pas mangé depuis `g_starvation_seconds` secondes est despawnée. Pour les espèces non-prédatrices, le champ `last_eaten` est initialisé à l’instant courant et n’est jamais mis à jour ; à moins que `g_starvation_seconds` ne soit anormalement court, ces espèces ne meurent donc pas de faim. Cela laisse intacte la mécanique de respawn pour les proies, qui meurent exclusivement par prédation.

5.4 Système de configuration TOML

5.4.1 Structure du fichier

Le fichier `aquarium.conf` comporte une section `[global]` pour les constantes du moteur, et autant de blocs `[[species]]` qu’il y a d’espèces. La syntaxe TOML est strictement définie par la

spécification 1.0.0 [6].

5.4.2 Parseur tomlc99

La bibliothèque `tomlc99` [9] expose une API en deux étapes : `toml_parse_file()` retourne un `toml_table_t *` si le fichier est syntaxiquement valide, puis des accesseurs typés (`toml_int_in()`, `toml_double_in()`, `toml_bool_in()`, `toml_string_in()`) lisent les valeurs par clé. Pour les tableaux de tables, `toml_array_in()` récupère le tableau, puis `toml_table_at()` accède aux sections individuelles.

Un point d'attention : les valeurs `string` retournées par `tomlc99` sont allouées dynamiquement et doivent être libérées avec `free()` (et non pas `toml_free()`, qui sert à libérer la table racine). Cette subtilité a été identifiée à la première compilation et corrigée.

5.4.3 Résolution des références prey

Le champ `prey` de chaque espèce est un tableau d'identifiants. Comme les indices de la table `species` ne sont connus qu'après lecture de toutes les espèces, la résolution se fait en **deux passes** :

1. **Passe 1.** Lecture des champs scalaires de chaque `[[species]]` : identifiant, nom, sprite, dimensions, paramètres comportementaux. Le `prey_mask` est laissé à zéro.
2. **Passe 2.** Pour chaque espèce, parcours du tableau `prey`, recherche linéaire de l'identifiant dans la table déjà construite, et activation du bit correspondant dans `prey_mask`. Une référence inconnue déclenche une erreur de chargement nommant l'espèce coupable et la proie absente.

Cette double passe est exécutée pendant que la `toml_table_t` racine est encore vivante en mémoire, ce qui permet de simplement re-parcourir l'arbre TOML sans avoir à mémoriser les chaînes brutes entre les deux passes.

5.4.4 Chemins de recherche et fallback

`conf_load()` essaie successivement plusieurs chemins, dans l'ordre suivant :

1. `override_path` explicite passé en argument (réservé à un futur `-config`).
2. `./aquarium.conf` (répertoire courant).
3. `$XDG_CONFIG_HOME/gtkaqua/aquarium.conf`, ou à défaut `~/.config/gtkaqua/aquarium.conf`.
4. `/etc/gtkaqua/aquarium.conf` (configuration système).

Le premier chemin existant et lisible (`stat()` qui renvoie un fichier régulier) est utilisé. Si aucun n'est trouvé, un message d'erreur précis liste les chemins essayés.

5.4.5 Variables globales et runtime

Les paramètres globaux issus de la configuration sont publiés sous forme de variables `extern` déclarées dans `include/config.h` et définies dans `src/conf.c` :

```

1 extern double g_eat_radius;
2 extern double g_adrenaline_boost;
3 extern double g_rotation_smooth;
4 extern double g_death_flash_duration;
5 extern double g_starvation_seconds;
6 extern int g_start_width;
7 extern int g_start_height;
8 extern int g_start_fullscreen;
9 extern int g_tick_ms;
10 extern const char *g_assets_dir;

```

Ce choix simplifie la consommation : `src/world.c` et `src/entity.c` référencent ces variables sans avoir besoin de connaître la `AquariumConfig`. Le couplage est unidirectionnel : `conf` écrit, tout le reste lit.

5.4.6 Rechargement à chaud

Le bouton *Reload config* de la barre latérale appelle un gestionnaire qui :

1. Construit un `AquariumConfig` temporaire et appelle `conf_load()` dessus.
2. Si l'opération échoue, affiche une boîte de dialogue d'erreur GTK avec le message d'erreur retourné par `conf_error()`, et abandonne sans toucher à l'état courant.
3. Si l'opération réussit, copie le tableau d'espèces dans le registre runtime via `species_init()`, puis vide le monde (`world_clear()`), reconstruit dynamiquement l'interface des espèces (cases à cocher du *pool*, combobox *Add precise*), et appelle `world_populate()` pour repeupler avec la nouvelle configuration.

Cette opération préserve la fenêtre, la pause, et tous les contrôles non liés aux espèces.

5.5 Rendu GTK4

5.5.1 Hiérarchie des widgets

L'application repose sur une hiérarchie GTK simple :

- `GtkApplicationWindow`.
- `GtkBox` horizontal racine.
- À gauche, un `GtkScrolledWindow` contenant la barre latérale (`GtkBox` vertical) avec les contrôles.
- À droite, un `GtkOverlay` qui empile la vidéo de fond (`GtkPicture` avec `GtkMediaFile`) et un `GtkFixed` servant de toile pour les sprites.

`GtkFixed` est un conteneur qui place ses enfants en coordonnées absolues. À chaque tick, `entity_apply_visuals()` appelle `gtk_fixed_move()` pour mettre à jour la position de chaque sprite.

5.5.2 Rotation par CSS

La rotation des sprites est confiée à GTK via une transformation CSS appliquée à chaque widget. Chaque entité possède un nom de classe CSS unique (`entity-N`) et un `GtkCssProvider` dédié. À chaque tick, le CSS est régénéré :

```
1 snprintf(css, sizeof(css),
2         "%s { transform: rotate(%.1fdeg); opacity: 1.0; }",
3         e->css_class, e->angle);
4 gtk_css_provider_load_from_string(e->css_provider, css);
```

L'angle est lissé entre l'angle courant et la cible (`heading_deg_from_vel`) selon le facteur `g_rotation_smooth` :

```
1 double diff = target_deg - e->angle;
2 while (diff > 180.0) diff -= 360.0;
3 while (diff < -180.0) diff += 360.0;
4 e->angle += diff * g_rotation_smooth;
```

Cela évite les changements brusques de direction visuelle.

5.5.3 Effet de mort

L'état DYING active un effet visuel pulsé combinant rotation, mise à l'échelle et variation d'opacité :

```
1 double pulse = fmod(e->death_timer * 24.0, 2.0) < 1.0 ? 1.0 : 0.0;
2 double scale = 1.15 + (0.35 * pulse);
3 double opacity = 0.35 + (0.55 * pulse);
4 snprintf(css, sizeof(css),
5         "%s { transform: rotate(%.1fdeg) scale(%.2f); "
6         "opacity: %.2f; }",
7         e->css_class, e->angle, scale, opacity);
8 e->death_timer -= w->dt;
9 if (e->death_timer <= 0.0) {
10     e->state = ENTITY_INACTIVE;
11     gtk_widget_set_visible(e->widget, FALSE);
12 }
```

Le calcul `fmod(e->death_timer * 24.0, 2.0) < 1.0` produit un battement à 12 Hz qui alterne entre deux états visuels, donnant un effet de *flash* caractéristique pendant `g_death_flash_duration` secondes.

Fichier	Lignes (approx.)
src/main.c	560
src/world.c	335
src/entity.c	380
src/species.c	35
src/conf.c	305
include/*.h	200
aquarium.conf	240
Total projet	≈ 2050

TABLE 5.1 – Volume de code par module.

5.6 Récapitulatif modulaire

Au final, le projet livre une base de code de l'ordre de 1500 lignes de C (hors *vendor*), répartie selon le tableau 5.1.

Cette répartition reflète la philosophie du projet : un coût fixe modeste pour chaque module sauf `main.c` (qui porte toute l'interface utilisateur GTK) et `entity.c` (qui porte les algorithmes physiques et de pilotage).

```

56 static Vec2 force_boids(World *w, int index) {
57     Entity *self = &w->entities[index];
58     const SpeciesConfig *cfg = species_get(self->species);
59
60     Vec2 separation = {0.0, 0.0};
61     Vec2 alignment = {0.0, 0.0};
62     Vec2 cohesion_sum = {0.0, 0.0};
63     int sep_count = 0;
64     int ali_count = 0;
65     int coh_count = 0;
66
67     for (int i = 0; i < w->entity_count; i++) {
68         if (i == index) {
69             continue;
70         }
71         Entity *other = &w->entities[i];
72         if (other->state != ENTITY_ALIVE || other->species != self->species) {
73             continue;
74         }
75
76         Vec2 d = vec2_delta_wrap(self->pos, other->pos, w->width, w->height);
77         double dist_sq = vec2_len_sq(d);
78         if (dist_sq < 1e-6) {
79             continue;
80         }
81
82         if (dist_sq < cfg->separation_radius * cfg->separation_radius) {
83             Vec2 away = vec2_scale(d, -1.0 / dist_sq);
84             separation = vec2_add(separation, away);
85             sep_count++;
86         }
87
88         if (dist_sq < cfg->alignment_radius * cfg->alignment_radius) {
89             alignment = vec2_add(alignment, other->vel);
90             ali_count++;
91         }
92
93         if (dist_sq < cfg->cohesion_radius * cfg->cohesion_radius) {
94             Vec2 local = vec2_add(self->pos, d);
95             cohesion_sum = vec2_add(cohesion_sum, local);
96             coh_count++;
97         }
98     }
99
100     Vec2 out = {0.0, 0.0};
101
102     if (sep_count > 0) {
103         separation = vec2_scale(separation, 1.0 / (double) sep_count);
104         Vec2 desired = vec2_scale(vec2_normalize(separation), entity_max_speed(self, cfg));
105         Vec2 steer = steer_toward(self->vel, desired, cfg->max_force);
106         out = vec2_add(out, vec2_scale(steer, cfg->separation_weight));
107     }
108
109     if (ali_count > 0) {
110         alignment = vec2_scale(alignment, 1.0 / (double) ali_count);
111         Vec2 desired = vec2_scale(vec2_normalize(alignment), entity_max_speed(self, cfg));
112         Vec2 steer = steer_toward(self->vel, desired, cfg->max_force);
113         out = vec2_add(out, vec2_scale(steer, cfg->alignment_weight));
114     }
115
116     if (coh_count > 0) {
117         Vec2 center = vec2_scale(cohesion_sum, 1.0 / (double) coh_count);
118         Vec2 to_center = vec2_sub(center, self->pos);
119         Vec2 desired = vec2_scale(vec2_normalize(to_center), entity_max_speed(self, cfg));
120         Vec2 steer = steer_toward(self->vel, desired, cfg->max_force);
121         out = vec2_add(out, vec2_scale(steer, cfg->cohesion_weight));
122     }
123
124     return out;
125 }

```

```

127 static Vec2 force_flee(World *w, int index, gboolean *is_fleeing) {
128     Entity *self = &w->entities[index];
129     const SpeciesConfig *cfg = species_get(self->species);
130     Vec2 fear = {0.0, 0.0};
131     int fear_count = 0;
132
133     if (cfg->fear_radius <= 0.0 || cfg->fear_weight <= 0.0) {
134         *is_fleeing = FALSE;
135         return fear;
136     }
137
138     for (int i = 0; i < w->entity_count; i++) {
139         if (i == index) {
140             continue;
141         }
142         Entity *other = &w->entities[i];
143         if (other->state != ENTITY_ALIVE) {
144             continue;
145         }
146
147         const SpeciesConfig *other_cfg = species_get(other->species);
148         if (!EATS(*other_cfg, self->species)) {
149             continue;
150         }
151
152         Vec2 d = vec2_delta_wrap(self->pos, other->pos, w->width, w->height);
153         double dist_sq = vec2_len_sq(d);
154         if (dist_sq >= cfg->fear_radius * cfg->fear_radius || dist_sq < 1e-8) {
155             continue;
156         }
157
158         Vec2 away = vec2_scale(d, -1.0 / dist_sq);
159         fear = vec2_add(fear, away);
160         fear_count++;
161     }
162
163     if (fear_count == 0) {
164         *is_fleeing = FALSE;
165         return fear;
166     }
167
168     fear = vec2_scale(fear, 1.0 / (double) fear_count);
169     Vec2 desired = vec2_scale(vec2_normalize(fear), entity_max_speed(self, cfg) *
↪ g_adrenaline_boost);
170     Vec2 steer = steer_toward(self->vel, desired, cfg->max_force * g_adrenaline_boost);
171     *is_fleeing = TRUE;
172     return vec2_scale(steer, cfg->fear_weight);
173 }

```

Listing 5.3 – Force de fuite avec boost d'adrénaline (extrait de src/entity.c).

```

33 Vec2 entity_head_point(const Entity *e, const SpeciesConfig *cfg) {
34     Vec2 dir = vec2_normalize(e->vel);
35     if (vec2_len_sq(dir) < 1e-8) {
36         double rad = (e->angle - 180.0) * (M_PI / 180.0);
37         dir = (Vec2) {cos(rad), sin(rad)};
38     }
39     double forward = ((double) cfg->sprite_width * cfg->mouth_forward_ratio);
40     return vec2_add(e->pos, vec2_scale(dir, forward));
41 }

```

Listing 5.4 – Calcul du point de bouche d’un prédateur.

```

196     }
197 }
198
199 static void world_handle_eating(World *w) {
200     for (int i = 0; i < w->entity_count; i++) {
201         Entity *pred = &w->entities[i];
202         if (pred->state != ENTITY_ALIVE) {
203             continue;
204         }
205         const SpeciesConfig *pred_cfg = species_get(pred->species);
206         if (pred_cfg == 0 || pred_cfg->prey_mask == 0) {
207             continue;
208         }
209
210         Vec2 pred_mouth = entity_head_point(pred, pred_cfg);
211
212         for (int j = 0; j < w->entity_count; j++) {
213             if (i == j) {
214                 continue;
215             }
216             Entity *prey = &w->entities[j];
217             if (prey->state != ENTITY_ALIVE || !EATS(*pred_cfg, prey->species)) {
218                 continue;
219             }
220
221             const SpeciesConfig *prey_cfg = species_get(pre->species);
222             if (prey_cfg == 0) {
223                 continue;
224             }
225
226             double prey_hit = ((double) (prey_cfg->sprite_width < prey_cfg->sprite_height ?
227 ↪ prey_cfg->sprite_width : prey_cfg->sprite_height)) * 0.28;
228             if (prey_hit < 8.0) {
229                 prey_hit = 8.0;
230             }
231             double eat_dist = g_eat_radius + prey_hit;
232
233             double d2 = vec2_dist_sq_wrap(pred_mouth, prey->pos, w->width, w->height);
234             if (d2 < eat_dist * eat_dist) {
235                 pred->last_eaten = w->elapsed;
236                 world_despawn(w, j);
237             }
238         }
239     }
240 }

```

Listing 5.5 – Détection des repas (extrait de src/world.c).

```
116 }  
117  
118  
119 void world_despawn(World *w, int index) {  
120     if (index < 0 || index >= w->entity_count) {  
121         return;  
122     }  
123     Entity *e = &w->entities[index];  
124     if (e->state == ENTITY_ALIVE) {  
125         e->state = ENTITY_DYING;  
126         if (w->alive_counts[e->species] > 0) {  
127             w->alive_counts[e->species]--;  
128         }  
129         e->death_timer = g_death_flash_duration;  
130     }  
131  
132     const SpeciesConfig *cfg = species_get(e->species);  
133     if (cfg != 0) {  
134         e->respawn_at = w->elapsed + cfg->respawn_delay;  
135         if (w->respawn_ready[e->species] < e->respawn_at) {  
136             w->respawn_ready[e->species] = e->respawn_at;  
137         }  
138     }
```

Listing 5.6 – Despawn d’une entité (extrait de `src/world.c`).

```

252
253 static void world_handle_respawns(World *w) {
254     int sc = species_count();
255     for (int s = 0; s < sc; s++) {
256         const SpeciesConfig *cfg = species_get(s);
257         if (cfg == 0) {
258             continue;
259         }
260
261         if (w->alive_counts[s] >= cfg->flock_size) {
262             continue;
263         }
264
265         if (w->elapsed < w->respawn_ready[s]) {
266             continue;
267         }
268
269         for (int i = 0; i < w->entity_count; i++) {
270             Entity *e = &w->entities[i];
271             if (e->state != ENTITY_INACTIVE || e->species != s) {
272                 continue;
273             }
274             e->state = ENTITY_ALIVE;
275             e->age = 0.0;
276             e->last_eaten = w->elapsed;
277             e->respawn_at = 0.0;
278             e->pos = random_position(w);
279             e->vel = random_velocity_for(cfg);
280             e->acc = (Vec2) {0.0, 0.0};
281             e->angle = heading_deg_from_vel(e->vel);
282             e->speed_scale = rand_range(0.82, 1.30);
283             e->death_timer = 0.0;
284             e->anim_frame = 0;
285             e->anim_timer = 0.0;
286             if (e->widget != 0) {
287                 const SpeciesConfig *respawn_cfg = species_get(e->species);
288                 if (respawn_cfg != 0 && respawn_cfg->asset_path[0] != '\\0') {
289                     gtk_picture_set_filename(GTK_PICTURE(e->widget), respawn_cfg->asset_path);
290                 }
291                 gtk_widget_set_visible(e->widget, TRUE);
292             }
293             w->alive_counts[s]++;
294             w->respawn_ready[s] = w->elapsed + cfg->respawn_delay;
295             break;
296         }

```

Listing 5.7 – Logique de respawn (extrait de src/world.c).

```
239 }
240
241 static void world_handle_lifespans(World *w) {
242     for (int i = 0; i < w->entity_count; i++) {
243         Entity *e = &w->entities[i];
244         if (e->state != ENTITY_ALIVE) {
245             continue;
246         }
247         if (w->elapsed - e->last_eaten >= g_starvation_seconds) {
248             world_despawn(w, i);
249         }
250     }
}
```

Listing 5.8 – Vérification de la famine.

```
1  [global]
2  eat_radius = 10.0
3  adrenaline_boost = 2.5
4  rotation_smooth = 0.1
5  death_flash_duration = 0.22
6  starvation_seconds = 120.0
7  start_width = 1280
8  start_height = 720
9  start_fullscreen = false
10 tick_ms = 16
11 assets_dir = "assets/"
12
13 [[species]]
14 id = "goldfish"
15 name = "Goldfish"
16 sprite = "assets/goldfish.png"
17 sprite_width = 46
18 sprite_height = 23
19 mouth_forward_ratio = 0.40
20 max_speed = 85.0
21 min_speed = 30.0
22 max_force = 42.0
23 separation_radius = 24.0
24 alignment_radius = 120.0
25 cohesion_radius = 100.0
26 separation_weight = 1.6
27 alignment_weight = 1.35
28 cohesion_weight = 1.45
29 fear_radius = 190.0
30 fear_weight = 2.5
31 hunt_radius = 0.0
32 hunt_weight = 0.0
33 avoid_same_radius = 0.0
34 avoid_same_weight = 0.0
35 prey = []
36 lifespan = 120.0
37 flock_size = 45
38 max_population = 60
39 respawn_delay = 1.5
40 is_apex = false
```

Listing 5.9 – Extrait de `aquarium.conf` : section globale et bloc `goldfish`.

Chapitre 6

Tests et résultats

6.1 Méthodologie

Le projet ne dispose pas pour l’instant d’un *framework* de tests unitaires automatisés : la nature visuelle de la simulation et le couplage fort avec GTK rendent peu rentable la mise en place d’un harnais comme `cmocka` ou `check`. À la place, la validation s’est appuyée sur :

- **Tests fonctionnels manuels** reproductibles, listés dans la section suivante. Chaque scénario est défini par : une condition initiale (état du fichier de configuration), une action utilisateur, un résultat attendu, et un résultat observé.
- **Tests de chargement de configuration** avec des cas volontairement erronés : typo dans une clé, identifiant de proie inconnu, fichier introuvable.
- **Compilation avec drapeaux stricts** : `-Wall -Wextra -Wpedantic` pour détecter les constructions douteuses.

Cette approche reste qualitative : aucune mesure quantitative de performance (FPS moyen, latence d’un tick) n’a été instrumentée, ce qui constitue une limite identifiée et reportée dans les perspectives (chapitre 7).

6.2 Scénarios fonctionnels

6.2.1 S1 – Démarrage avec configuration par défaut

Préconditions. Le binaire `build/bin/aquarium` est construit. Le fichier `aquarium.conf` contient les huit espèces de la configuration de référence avec leurs valeurs documentées.

Action. Lancer `./build/bin/aquarium`.

Attendu. La fenêtre s’ouvre en 1280×720, la barre latérale présente huit cases à cocher (une par espèce), la combobox *Add precise* liste les huit noms, le pool initial spawn environ 95 entités au total (45 goldfish, 18 clownfish, 14 angelfish, 10 par prédateur intermédiaire, 4 tuna, 2 shark).

Observé. Conforme. Les bancs se forment en quelques secondes, les prédateurs commencent à chasser après le premier croisement avec une proie.

6.2.2 S2 – Modification d’un paramètre simple

Préconditions. Application en cours d’exécution.

Action. Éditer `aquarium.conf`, remplacer `max_speed = 85.0` par `max_speed = 200.0` dans le bloc `[[species]]` `goldfish`, sauvegarder, et cliquer sur *Reload config*.

Attendu. Les goldfish se déplacent visiblement plus vite (vitesse maximale environ $2,4\times$ supérieure). Les autres espèces conservent leur comportement nominal.

Observé. Conforme. Les goldfish atteignent visuellement une vitesse plus élevée. Le banc paraît plus dispersé car la cohésion à pondération inchangée a moins de prise sur des individus rapides.

6.2.3 S3 – Suppression d’une espèce

Préconditions. Application en cours d’exécution.

Action. Commenter intégralement le bloc `[[species]] angelfish` dans la configuration en préfixant chaque ligne par `#`, puis cliquer sur *Reload config*.

Attendu. L’espèce angelfish disparaît : aucun spawn, plus de case à cocher, plus d’entrée dans la combobox. Les autres espèces continuent normalement. Aucun crash.

Observé. Conforme. Les références `prey` pointaient déjà vers angelfish dans la configuration du requin : l’absence d’angelfish est correctement gérée car le bit correspondant n’est simplement pas mis dans `prey_mask`.

Note. Si une espèce dont l’identifiant figure encore dans une liste `prey` d’un autre prédateur est supprimée, le chargement échoue avec un message clair (cf. scénario S5).

6.2.4 S4 – Ajout d’une nouvelle espèce

Préconditions. Un fichier PNG `piranha.png` orienté vers la gauche est placé dans `assets/`.

Action. Ajouter à la fin de `aquarium.conf` un bloc `[[species]]` avec `id = "piranha"` et un sous-ensemble cohérent de paramètres (vitesse moyenne, rayon de chasse, `prey = ["goldfish"]`). Cliquer sur *Reload config*.

Attendu. Une neuvième case à cocher apparaît, une neuvième entrée dans la combobox, et des piranhas spawnent dans la simulation.

Observé. Conforme. Aucun changement de code C n’a été nécessaire.

6.2.5 S5 – Configuration invalide : référence inconnue

Préconditions. Application en cours d’exécution.

Action. Modifier `prey = ["goldfish"]` en `prey = ["sardine"]` dans le bloc `trout`, puis cliquer sur *Reload config*.

Attendu. La boîte de dialogue d’erreur s’affiche avec un message du type « species 'trout' : unknown prey 'sardine' ». La simulation courante n’est pas affectée (rollback implicite : l’ancienne configuration reste en place).

Observé. Conforme.

6.2.6 S6 – Configuration invalide : syntaxe TOML

Préconditions. Application en cours d’exécution.

Action. Introduire volontairement une erreur de syntaxe (par exemple : oubli des guillemets autour d’une chaîne, ou crochet ouvert non refermé) puis cliquer sur *Reload config*.

Attendu. La boîte d’erreur affiche le chemin du fichier suivi du message d’erreur retourné par `tomlc99`, qui inclut typiquement le numéro de ligne.

Observé. Conforme. Le message est précis car `tomlc99` fournit lui-même un `errbuf` explicite.

6.2.7 S7 – Absence du fichier

Préconditions. Renommer temporairement `aquarium.conf` en `aquarium.conf.bak` et s’assurer qu’aucune copie de *fallback* n’existe en `~/.config/gtkaqua/` ni en `/etc/gtkaqua/`.

Action. Lancer l’application.

Attendu. Échec propre avec le message « no aquarium.conf found (checked ./aquarium.conf, ~/.config/gtkaqua/aquarium.conf, /etc/gtkaqua/aquarium.conf) ». Pas de *segfault*.

Observé. Conforme.

6.2.8 S8 – Fallback XDG

Préconditions. `aquarium.conf` retiré du répertoire courant. Une copie placée dans `~/.config/gtkaqua/aquarium.conf` via `make install-conf`.

Action. Lancer l’application depuis un répertoire arbitraire qui ne contient pas `aquarium.conf`.

Attendu. L’application démarre normalement, en utilisant la copie XDG.

Observé. Conforme.

6.2.9 S9 – Pause et reprise

Préconditions. Simulation en cours.

Action. Cliquer sur *Pause*, attendre quelques secondes, observer le comportement, puis cliquer sur *Resume*.

Attendu. Pendant la pause, les sprites restent immobiles, les widgets sont à leur dernière position. À la reprise, le mouvement reprend sans saut.

Observé. Conforme. `world_tick` n’est plus appelé pendant la pause, mais le timer GTK continue, donc `elapsed` ne progresse pas.

6.2.10 S10 – Plein écran

Préconditions. Simulation en cours.

Action. Appuyer sur F11.

Attendu. La fenêtre passe en plein écran. Les sprites sont libres d’occuper toute la zone, qui est reconfigurée automatiquement (`world.width`, `world.height` sont mises à jour à partir de `gtk_widget_get_width/height` dans le `tick_cb`).

Observé. Conforme. Appuyer sur Échap restaure le mode fenêtré.

6.3 Synthèse des résultats

Le tableau 6.1 récapitule les dix scénarios.

Scénario	Fonctionnalité testée	Résultat
S1	Démarrage avec config par défaut	OK
S2	Modification d'un paramètre simple	OK
S3	Suppression d'une espèce	OK
S4	Ajout d'une nouvelle espèce	OK
S5	Référence <code>prey</code> inconnue	OK
S6	Erreur de syntaxe TOML	OK
S7	Absence totale du fichier	OK
S8	Fallback XDG	OK
S9	Pause/Resume	OK
S10	Plein écran	OK

TABLE 6.1 – Synthèse des scénarios de test.

6.4 Limites identifiées

- **Pas de mesure quantitative de performance.** Le coût d'un tick n'est pas instrumenté. Pour une évaluation rigoureuse, il faudrait mesurer la durée moyenne et le 99e centile de `world_tick()` en fonction du nombre d'entités, et tracer le résultat.
- **Complexité $O(N^2)$.** Toutes les forces parcourent la liste complète des entités à chaque tick. À population modeste (100 entités) cela ne pose pas de problème, mais le coût croît rapidement : 200 entités impliquent quatre fois plus de comparaisons.
- **Pas de tests automatisés.** Les régressions sont actuellement détectées *a posteriori* par observation visuelle. Un harnais de tests sur les fonctions pures (parsing TOML, résolution de `prey_mask`, `vec2_*`()) serait peu coûteux à mettre en place et procurerait une garantie utile.
- **Couverture du parseur de configuration.** Toutes les clés sont optionnelles côté parseur : si une clé obligatoire est manquante (par exemple `sprite`), le chargement détecte le cas, mais d'autres clés silencieusement par défaut à zéro pourraient produire des comportements dégénérés (vitesse nulle, rayons nuls). Une validation *post-load* renforcée serait souhaitable.

Chapitre 7

Conclusion et perspectives

7.1 Bilan

Le projet a livré un aquarium animé fonctionnel sous Linux et GTK4, peuplé par défaut de huit espèces et offrant des interactions prédateur/proie crédibles. Les objectifs initiaux sont atteints :

- Le moteur de simulation à pas fixe est stable et déterministe, avec les trois forces de Reynolds correctement implémentées et complétées par des comportements de fuite, de chasse et d'évitement intra-espèce.
- Le cycle de vie complet des entités (**ALIVE**, **DYING**, **INACTIVE**) est en place, avec un recyclage qui borne statiquement la consommation mémoire.
- Le *système de configuration externe* en TOML permet effectivement à un utilisateur non développeur d'ajuster la simulation, d'ajouter ou de retirer des espèces, sans recompilation. Les chemins de *fallback*, le rechargement à chaud et la gestion d'erreurs lisible répondent aux attentes posées au chapitre 3.
- L'interface graphique propose les contrôles essentiels : pause, vitesse globale, sélection du *pool*, ajout et retrait précis de poissons, plein écran, et rechargement de la configuration.

Le tableau 7.1 synthétise les exigences fonctionnelles et leur statut.

Réf.	Exigence	Statut
EF1	Simulation de banc multi-espèces	livré
EF2	Prédateur/proie avec mouth-based eating	livré
EF3	Fuite des proies + adrénaline	livré
EF4	Cycle de vie 3 états	livré
EF5	Famine	livré
EF6	Configuration TOML externe	livré
EF7	Recherche multi-niveaux	livré
EF8	Ajout/suppression sans recompilation	livré
EF9	Validation et messages d'erreur	livré
EF10	Rechargement à chaud	livré
EF11	Contrôles utilisateur	livré
EF12	Plein écran	livré

TABLE 7.1 – Statut des exigences fonctionnelles.

7.2 Limites techniques

Le projet, dans son état actuel, présente plusieurs limites identifiées :

Complexité quadratique. Toutes les détections de voisinage parcourent l'ensemble des entités, ce qui rend le coût $O(N^2)$ par tick. Au-delà de quelques centaines d'entités, cela deviendrait limitant. Une structure spatiale (grille uniforme, *spatial hashing*) ramènerait le coût à $O(N \log N)$ voire $O(N)$ dans le cas amorti.

Rendu widget par widget. Chaque entité est un widget GTK distinct, soumis à un cycle complet de *layout* et de *styling* CSS à chaque tick. Cela limite le nombre maximal d'entités tenable à 60 Hz. Un rendu via `GtkSnapshot` ou un `GtkDrawingArea` unique avec `cairo` serait beaucoup plus efficace.

Validation post-chargement minimale. Le parseur TOML ne vérifie pas la cohérence sémantique des paramètres : un `max_speed` à zéro, un `flock_size` négatif ou un `respawn_delay` infini ne déclencheraient pas d'erreur. Une passe de validation explicite avec plages tolérées serait utile.

Pas de tests automatisés. Les régressions sont détectées par observation visuelle. Un harnais sur les fonctions pures (`vec2_*`, parsing TOML, résolution `prey_mask`) serait peu coûteux à mettre en place.

Pas d'instrumentation de performance. Aucun compteur FPS, aucune mesure du coût d'un tick.

7.3 Perspectives

Plusieurs pistes d'évolution se dégagent naturellement :

Compteur FPS et statistiques temps réel. Ajouter un panneau optionnel affichant le FPS instantané, le coût du tick, et le nombre d'entités vivantes par espèce. Cela faciliterait le *tuning* et la détection de régressions de performance.

Profil de configuration. Permettre la sélection d'un fichier de configuration alternatif via un argument de ligne de commande `-config calme.conf`, et fournir quelques presets dans `presets/` (`calme.conf`, `chaos.conf`, `predator-heavy.conf`).

Sauvegarde et restauration d'état. Permettre de *geler* l'état complet de la simulation (positions, vitesses, états) et de le restaurer ultérieurement, pour reproduire des scénarios intéressants.

Spatial hashing. Implémenter une grille uniforme indexée sur la position des entités, mise à jour à chaque tick. Les recherches de voisinage deviennent localisées à quelques cellules.

Rendu Cairo unique. Remplacer les widgets `GtkPicture` par un seul `GtkDrawingArea` dessiné en Cairo, ce qui éliminerait le coût de *layout* et permettrait d'atteindre des populations beaucoup plus importantes.

Mode multijoueur. Distribuer la simulation sur plusieurs instances connectées par socket UDP, avec une synchronisation périodique de l'état. Le déterminisme du pas fixe rend cet exercice tractable.

Portage Wayland natif et Windows. GTK4 supporte nativement Wayland ; un portage Windows via les paquets MSYS2 est en théorie possible. La principale difficulté serait le *packaging* et la gestion des chemins de configuration.

Validation TOML renforcée. Définir un schéma explicite des plages acceptables pour chaque paramètre, et émettre des avertissements quand des valeurs sortent du raisonnable (par exemple `max_speed > 1000`).

7.4 Apport personnel

Au-delà du résultat livré, le projet a été l'occasion d'explorer en pratique :

- Le modèle de Reynolds, depuis l'article original de 1987 jusqu'à ses extensions modernes pour les jeux vidéo [5].
- L'écriture d'un parseur de configuration robuste en C, avec gestion mémoire stricte et messages d'erreur lisibles.
- Les particularités de GTK4 utilisé hors de son contexte habituel (interface de bureau classique) pour piloter une simulation animée temps réel.
- La discipline de *vendoring* d'une bibliothèque tierce (`tomlc99`) pour garantir la reproductibilité du *build* sans dépendance externe.

Le *refactoring* qui a fait passer la liste des espèces d'une `enum` compile-time à un registre dynamique runtime a en particulier été instructif : il a fallu identifier précisément tous les points de couplage du code à l'enum, et les remplacer par des indices entiers résolus à l'exécution. Cette transformation, en apparence cosmétique, est en réalité ce qui a rendu possible l'objectif central du projet (configuration externe).

Bibliographie

- [1] Ffmpeg : A complete, cross-platform solution to record, convert and stream audio and video, 2024.
- [2] Glib reference manual, 2024.
- [3] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2004.
- [4] The GNOME Project. *GTK 4 Reference Manual*, 2024.
- [5] Ian Millington and John Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2 edition, 2009.
- [6] Tom Preda and contributors. TOML : Tom’s obvious, minimal language, version 1.0.0, 2021.
- [7] Craig W. Reynolds. Flocks, herds and schools : A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’87)*, pages 25–34, New York, NY, USA, 1987. ACM.
- [8] Craig W. Reynolds. Steering behaviors for autonomous characters, 1999. Game Developers Conference 1999.
- [9] C. K. Tan. tomlc99 : Toml parser and serializer in c99, 2024.

Annexe A

Fichier de configuration complet

Le fichier ci-après est la configuration de référence livrée avec le projet. Il fixe les valeurs identiques à celles utilisées en interne avant l'introduction du système de configuration externe, garantissant ainsi une parfaite compatibilité descendante.

```
1  # GTK Aquarium Configuration
2  # Edit this file to tune the simulation, add/remove species, or change behavior.
3  # Lines starting with # are comments.
4  #
5  # After editing, restart the aquarium for changes to take effect.
6  # If you see an error on launch, check the terminal output.
7
8  [global]
9  eat_radius = 10.0
10 adrenaline_boost = 2.5
11 rotation_smooth = 0.1
12 death_flash_duration = 0.22
13 starvation_seconds = 120.0
14 start_width = 1280
15 start_height = 720
16 start_fullscreen = false
17 tick_ms = 16
18 assets_dir = "assets/"
19
20 # --- Species -----
21 # Each entry under [[species]] defines one type of fish.
22 # To remove a fish, delete (or comment out) its entire [[species]] block.
23 # To add a new fish, copy an existing block, change the id, name, sprite path,
24 # and tune the values below.
25 #
26 # prey = [ "id1", "id2", ... ]    + list of species this fish hunts (by id)
27 # is_apex = true/false           + apex predators show a special label in the UI
28
29 [[species]]
30 id = "goldfish"
31 name = "Goldfish-ds"
32 sprite = "assets/goldfish.png"
33 sprite_width = 46
34 sprite_height = 23
35 mouth_forward_ratio = 0.40
36 max_speed = 85.0
37 min_speed = 30.0
38 max_force = 42.0
39 separation_radius = 24.0
40 alignment_radius = 120.0
41 cohesion_radius = 100.0
42 separation_weight = 1.6
43 alignment_weight = 1.35
44 cohesion_weight = 1.45
45 fear_radius = 190.0
46 fear_weight = 2.5
47 hunt_radius = 0.0
```

```
48 hunt_weight = 0.0
49 avoid_same_radius = 0.0
50 avoid_same_weight = 0.0
51 prey = []
52 lifespan = 120.0
53 flock_size = 45
54 max_population = 60
55 respawn_delay = 1.5
56 is_apex = false
57
58
59
60 [[species]]
61 id = "clownfish"
62 name = "Clownfish"
63 sprite = "assets/clownfish.png"
64 sprite_width = 72
65 sprite_height = 42
66 mouth_forward_ratio = 0.40
67 max_speed = 90.0
68 min_speed = 32.0
69 max_force = 45.0
70 separation_radius = 34.0
71 alignment_radius = 115.0
72 cohesion_radius = 125.0
73 separation_weight = 1.7
74 alignment_weight = 1.35
75 cohesion_weight = 1.45
76 fear_radius = 195.0
77 fear_weight = 2.6
78 hunt_radius = 0.0
79 hunt_weight = 0.0
80 avoid_same_radius = 0.0
81 avoid_same_weight = 0.0
82 prey = []
83 lifespan = 120.0
84 flock_size = 10
85 max_population = 18
86 respawn_delay = 1.5
87 is_apex = false
88
89 [[species]]
90 id = "angelfish"
91 name = "Angelfish"
92 sprite = "assets/angelfish.png"
93 sprite_width = 78
94 sprite_height = 74
95 mouth_forward_ratio = 0.40
96 max_speed = 95.0
97 min_speed = 34.0
98 max_force = 44.0
99 separation_radius = 38.0
100 alignment_radius = 130.0
101 cohesion_radius = 142.0
102 separation_weight = 1.5
103 alignment_weight = 1.25
104 cohesion_weight = 1.3
105 fear_radius = 210.0
106 fear_weight = 2.5
107 hunt_radius = 0.0
108 hunt_weight = 0.0
```

```
109 avoid_same_radius = 0.0
110 avoid_same_weight = 0.0
111 prey = []
112 lifespan = 150.0
113 flock_size = 14
114 max_population = 14
115 respawn_delay = 2.0
116 is_apex = false
117
118 [[species]]
119 id = "barracuda"
120 name = "Barracuda"
121 sprite = "assets/barracuda.png"
122 sprite_width = 96
123 sprite_height = 38
124 mouth_forward_ratio = 0.42
125 max_speed = 130.0
126 min_speed = 40.0
127 max_force = 60.0
128 separation_radius = 56.0
129 alignment_radius = 100.0
130 cohesion_radius = 112.0
131 separation_weight = 1.3
132 alignment_weight = 0.8
133 cohesion_weight = 0.6
134 fear_radius = 230.0
135 fear_weight = 1.5
136 hunt_radius = 300.0
137 hunt_weight = 2.7
138 avoid_same_radius = 90.0
139 avoid_same_weight = 0.8
140 prey = [ "goldfish", "clownfish" ]
141 lifespan = 200.0
142 flock_size = 10
143 max_population = 10
144 respawn_delay = 4.0
145 is_apex = false
146
147 [[species]]
148 id = "bass"
149 name = "Bass"
150 sprite = "assets/bass.png"
151 sprite_width = 84
152 sprite_height = 52
153 mouth_forward_ratio = 0.42
154 max_speed = 112.0
155 min_speed = 36.0
156 max_force = 53.0
157 separation_radius = 48.0
158 alignment_radius = 96.0
159 cohesion_radius = 108.0
160 separation_weight = 1.4
161 alignment_weight = 0.8
162 cohesion_weight = 0.7
163 fear_radius = 245.0
164 fear_weight = 1.7
165 hunt_radius = 290.0
166 hunt_weight = 2.4
167 avoid_same_radius = 80.0
168 avoid_same_weight = 0.7
169 prey = [ "goldfish", "clownfish" ]
```

```
170 lifespan = 180.0
171 flock_size = 10
172 max_population = 10
173 respawn_delay = 4.5
174 is_apex = false
175
176 [[species]]
177 id = "trout"
178 name = "Trout"
179 sprite = "assets/trout.png"
180 sprite_width = 84
181 sprite_height = 48
182 mouth_forward_ratio = 0.42
183 max_speed = 110.0
184 min_speed = 36.0
185 max_force = 52.0
186 separation_radius = 44.0
187 alignment_radius = 94.0
188 cohesion_radius = 106.0
189 separation_weight = 1.4
190 alignment_weight = 0.9
191 cohesion_weight = 0.7
192 fear_radius = 240.0
193 fear_weight = 1.8
194 hunt_radius = 270.0
195 hunt_weight = 2.25
196 avoid_same_radius = 75.0
197 avoid_same_weight = 0.7
198 prey = [ "goldfish" ]
199 lifespan = 180.0
200 flock_size = 10
201 max_population = 10
202 respawn_delay = 4.5
203 is_apex = false
204
205 [[species]]
206 id = "tuna"
207 name = "Tuna"
208 sprite = "assets/tuna.png"
209 sprite_width = 90
210 sprite_height = 54
211 mouth_forward_ratio = 0.42
212 max_speed = 125.0
213 min_speed = 40.0
214 max_force = 56.0
215 separation_radius = 52.0
216 alignment_radius = 100.0
217 cohesion_radius = 114.0
218 separation_weight = 1.3
219 alignment_weight = 0.8
220 cohesion_weight = 0.6
221 fear_radius = 250.0
222 fear_weight = 1.7
223 hunt_radius = 320.0
224 hunt_weight = 2.5
225 avoid_same_radius = 85.0
226 avoid_same_weight = 0.8
227 prey = [ "goldfish", "clownfish" ]
228 lifespan = 200.0
229 flock_size = 4
230 max_population = 8
```

```
231 respawn_delay = 5.0
232 is_apex = false
233
234 [[species]]
235 id = "shark"
236 name = "Shark"
237 sprite = "assets/shark.png"
238 sprite_width = 190
239 sprite_height = 96
240 mouth_forward_ratio = 0.43
241 max_speed = 145.0
242 min_speed = 44.0
243 max_force = 62.0
244 separation_radius = 76.0
245 alignment_radius = 130.0
246 cohesion_radius = 140.0
247 separation_weight = 1.1
248 alignment_weight = 0.6
249 cohesion_weight = 0.4
250 fear_radius = 0.0
251 fear_weight = 0.0
252 hunt_radius = 430.0
253 hunt_weight = 3.2
254 avoid_same_radius = 220.0
255 avoid_same_weight = 3.4
256 prey = [ "tuna", "bass", "trout", "angelfish", "goldfish" , "sardine" ]
257 lifespan = 300.0
258 flock_size = 2
259 max_population = 8
260 respawn_delay = 6.0
261 is_apex = true
262
263 [[species]]
264 id = "sardine"
265 name = "sardine"
266 sprite = "assets/sardine.png"
267 sprite_width = 190
268 sprite_height = 96
269 mouth_forward_ratio = 0.43
270 max_speed = 145.0
271 min_speed = 44.0
272 max_force = 62.0
273 separation_radius = 76.0
274 alignment_radius = 130.0
275 cohesion_radius = 140.0
276 separation_weight = 1.1
277 alignment_weight = 0.6
278 cohesion_weight = 0.4
279 fear_radius = 0.0
280 fear_weight = 0.0
281 hunt_radius = 430.0
282 hunt_weight = 3.2
283 avoid_same_radius = 220.0
284 avoid_same_weight = 3.4
285 prey = []
286 lifespan = 1.0
287 flock_size = 20
288 max_population = 30
289 respawn_delay = 6.0
290 is_apex = true
```

Annexe B

Instructions de build et d'exécution

B.1 Dépendances système

- GCC ou Clang (C11)
- `make` (GNU make)
- `pkg-config`
- GTK4 et ses headers de développement
- FFmpeg (pour le pipeline d'assets uniquement)
- Bear (optionnel, pour générer `compile_commands.json`)

B.2 Commandes principales

```
1 # Préparer les sprites à partir des fichiers bruts (à faire une fois)
2 make assets
3
4 # Compiler le projet
5 make
6
7 # Lancer l'aquarium
8 ./build/bin/aquarium
9
10 # Nettoyer les artefacts de build
11 make clean
12
13 # Installer aquarium.conf dans ~/.config/gtkaqua/
14 make install-conf
```

B.3 Variables d'environnement utiles

`XDG_CONFIG_HOME` contrôle où l'application cherche son fichier de configuration utilisateur. Si cette variable n'est pas définie, la valeur par défaut `~/.config/` est utilisée.

Annexe C

Arborescence du projet

```
gtkqua/
|-- Makefile
|-- README.md
|-- aquarium.conf          <- configuration externe
|-- include/
|   |-- config.h           <- variables globales extern
|   |-- conf.h             <- API du loader TOML
|   |-- entity.h
|   |-- species.h
|   |-- vec2.h             <- vecteurs 2D (header-only)
|   `-- world.h
|-- src/
|   |-- conf.c             <- loader TOML + validation
|   |-- entity.c           <- forces et integration physique
|   |-- main.c             <- GTK, UI, tick principal
|   |-- species.c          <- registre dynamique
|   `-- world.c            <- pool d'entites, orchestration
|-- vendor/
|   `-- tomlc99/           <- parseur TOML vendored
|       |-- toml.c
|       |-- toml.h
|       `-- LICENSE
|-- Fish/                  <- sprites bruts
|-- assets/                <- sprites normalises + video de fond
|-- spritesheets/          <- references non utilisees
|-- docs/                  <- documentation projet
`-- build/
    |-- obj/               <- fichiers .o
    `-- bin/aquarium       <- binaire final
```

Annexe D

Tableau récapitulatif des espèces

Le tableau [D.1](#) résume les principales caractéristiques de chaque espèce telles que définies dans `aquarium.conf`.

Espèce	Vit. max	Vit. min	Force	Flock	Pop. max	Durée	Proies
Goldfish	85	30	42	45	60	120	–
Clownfish	90	32	45	18	18	120	–
Angelfish	95	34	44	14	14	150	–
Barracuda	130	40	60	10	10	200	goldfish, clownfish
Bass	112	36	53	10	10	180	goldfish, clownfish
Trout	110	36	52	10	10	180	goldfish
Tuna	125	40	56	4	8	200	goldfish, clownfish
Shark	145	44	62	2	8	300	tuna, bass, trout, angelfish, goldfish

TABLE D.1 – Récapitulatif des huit espèces par défaut.

Les vitesses sont exprimées en pixels par seconde, la force maximale en pixels par seconde au carré (accélération), et la durée de vie en secondes.

Annexe E

Définition de la structure SpeciesConfig

Pour référence, voici la définition complète du type qui porte l'intégralité du paramétrage d'une espèce, telle qu'elle figure dans `include/species.h` :

```
1  #ifndef SPECIES_H
2  #define SPECIES_H
3
4  #include <stdbool.h>
5
6  #define MAX_SPECIES 32
7
8  typedef struct {
9      char id[64];
10     char name[64];
11     char asset_path[256];
12     int sprite_width;
13     int sprite_height;
14     double mouth_forward_ratio;
15     double max_speed;
16     double min_speed;
17     double max_force;
18     double separation_radius;
19     double alignment_radius;
20     double cohesion_radius;
21     double separation_weight;
22     double alignment_weight;
23     double cohesion_weight;
24     double fear_radius;
25     double fear_weight;
26     double hunt_radius;
27     double hunt_weight;
28     double avoid_same_radius;
29     double avoid_same_weight;
30     int prey_mask;
31     double lifespan;
32     int flock_size;
33     int max_population;
34     double respawn_delay;
35     bool is_apex;
36 } SpeciesConfig;
37
38 #define EATS(predator_cfg, prey_index) ((predator_cfg).prey_mask & (1 << (prey_index)))
39
40 int species_count(void);
41 const SpeciesConfig *species_get(int index);
```

```
42 int species_find_by_id(const char *id);
43 void species_init(const SpeciesConfig *configs, int count);
44 void species_destroy(void);
45
46 #endif
```

Annexe F

Signature de l'API de configuration

L'interface publique exposée par le module de configuration :

```
1  #ifndef CONF_H
2  #define CONF_H
3
4  #include "species.h"
5
6  typedef struct {
7      double eat_radius;
8      double adrenaline_boost;
9      double rotation_smooth;
10     double death_flash_duration;
11     double starvation_seconds;
12     int start_width;
13     int start_height;
14     int start_fullscreen;
15     int tick_ms;
16     char assets_dir[256];
17 } GlobalConfig;
18
19 typedef struct {
20     GlobalConfig global;
21     SpeciesConfig species[MAX_SPECIES];
22     int species_count;
23     char last_error[512];
24 } AquariumConfig;
25
26 int conf_load(AquariumConfig *cfg, const char *override_path);
27 const char *conf_error(const AquariumConfig *cfg);
28
29 #endif
```