ChatGPT

Projet « Moteur de Simulation de Ferme » – Spécifications et Feuille de Route

1. Introduction et objectifs

L'objectif de ce projet est de concevoir et de développer **from scratch** un moteur de simulation modulaire permettant de mettre en scène différentes interactions dans un monde 2D. Le premier scénario visé est une **simulation de ferme** dans laquelle des personnages (des fermiers) produisent des ressources (par exemple du blé), gèrent leurs besoins (faim, fatigue), interagissent avec leur environnement et échangent des biens via un système économique simple. À terme, la base ainsi construite devra être suffisamment générique pour évoluer vers d'autres scénarios (aventure, guerre, société complète).

Le projet est pensé comme un **bac à sable communautaire** : les briques de base (appelées « nœuds ») doivent pouvoir être réassemblées pour créer des comportements variés. Tout doit être paramétrable via des fichiers de configuration déclaratifs pour faciliter la création et le partage de simulations.

2. Exigences générales

- **Modularité** : chaque entité ou mécanique doit être encapsulée dans un nœud indépendant. Les nœuds doivent être réutilisables dans différents contextes (par exemple, un `NeedNode` gérant la faim peut être utilisé pour des animaux ou des humains).
- **Extensibilité** : il doit être possible d'ajouter de nouveaux types de nœuds (plugins) sans modifier le noyau du moteur. Les plugins devront respecter des contrats d'interface clairement définis.
- **Rejouabilité** : la simulation doit pouvoir être rendue déterministe via un *seed* pour reproduire exactement une session de jeu.
- **Séparation du modèle et du rendu** : la logique de simulation (moteur) ne doit pas dépendre d'un moteur graphique spécifique. Un module de rendu pourra être ajouté (par exemple avec Pygame) mais restera optionnel.
- **Chargement déclaratif** : l'arbre de la simulation doit pouvoir être décrit dans un fichier YAML ou JSON. Le moteur instanciera l'arbre à partir de ce fichier.
- **Observabilité** : l'état de la simulation doit être facilement inspectable (dump d'état, affichage des stocks, niveaux des besoins, météo, etc.).

3. Architecture proposée

3.1 SimNode et bus d'événements

La structure de base est un arbre de **SimNodes** : chaque nœud peut avoir un parent et des enfants. Tous les nœuds disposent d'un **bus d'événements** qui permet d'émettre (`emit`) et de recevoir (`on_event`) des événements nommés. Les événements doivent pouvoir remonter ou descendre l'arbre selon des règles configurables (par défaut, propagation à tous les parents). Le bus d'événements doit permettre :

- l'enregistrement/désenregistrement de gestionnaires d'événements par type d'événement ;

- la propagation d'événements aux parents et/ou enfants, avec la possibilité de stopper la propagation ;
- l'horodatage ou l'identification des événements pour faciliter le debug.

Chaque nœud doit également fournir les méthodes suivantes :

- `update(dt)` : appelée à chaque tick de simulation pour mettre à jour l'état du nœud (en appelant `update` sur ses enfants par défaut) ;
- `serialize()` : retourne une représentation sérialisée du nœud et de ses enfants pour le stockage ou l'inspection ;
- `add_child(node)` et `remove_child(node)` : gestion de la hiérarchie.

3.2 Systèmes et boucles de simulation

Certains comportements globaux doivent être implémentés dans des **SystemNodes**, qui héritent de `SimNode` mais parcourent l'arbre ou écoutent des événements pour appliquer des règles globales. Exemples :

- `TimeSystem` : gère l'écoulement du temps (ticks, cycles jour/nuit, saisons) et émet des événements `tick`, `phase_changed` ;
- `EconomySystem` : écoute les événements d'achat/vente, effectue les transferts d'argent et met à jour les prix si besoin ;
- `WeatherSystem` : détermine la météo courante, déclenche des événements (`rain_started`, `drought`) et expose l'état de la météo aux autres nœuds.

3.3 Nœuds génériques

Pour modéliser la ferme et ses habitants, plusieurs nœuds génériques doivent être développés :

- `InventoryNode` : gère un stock d'objets ou de ressources. Doit permettre d'ajouter, de retirer et de transférer des ressources vers un autre inventaire. Peut émettre des événements `inventory_changed`.
- `NeedNode` : représente un besoin (par exemple faim ou fatigue). Contient une valeur courante, un ou plusieurs seuils et un taux d'accroissement/diminution. Émet des événements lorsque des seuils sont franchis (`need_threshold_reached` ou `need_satisfied`).
- `ResourceProducerNode` : produit une ressource à chaque tick en consommant éventuellement des intrants. Paramètres : type de ressource produite, taux de production, types d'intrants nécessaires (par exemple eau). Émet des événements `resource_produced`.
- `AIBehaviorNode` : décide des actions à entreprendre en fonction de l'état interne du personnage et des événements reçus. Pour un fermier, il écoutera `need_threshold_reached` (faim/fatigue) et les événements de temps (`phase_changed`) pour aller manger, travailler ou dormir.
- `TransformNode` (optionnel) : stocke la position et la vitesse d'un personnage dans l'espace si besoin.

3.4 Nœuds composés

Un personnage (« Fermier ») pourra être un `CharacterNode` qui combine :

- un `TransformNode` pour la position et la direction ;
- plusieurs `NeedNode` (faim, fatigue) ;

- un `InventoryNode` (stock personnel) ;
- un `AIBehaviorNode` (routine quotidienne).

Un bâtiment (« Ferme ») sera constitué d'un `InventoryNode` (stock de blé), d'un `ResourceProducerNode` (production de blé) et de propriétés fixes (position, taille). La ferme pourra consommer de l'eau ou un autre intrant si souhaité ultérieurement.

3.5 Registre de plugins et loader déclaratif

Les nœuds seront implémentés dans des modules Python et **enregistrés** dans un registre de plugins, de sorte qu'il soit possible d'instancier un nœud à partir d'un nom. Le loader lira un fichier YAML/JSON décrivant l'arbre : chaque entrée contiendra un champ `type` (nom du nœud enregistré), un champ `config` (dictionnaire de paramètres) et éventuellement un champ `children`.

Exemple de format YAML minimal :

```
world:
  type: WorldNode
  config:
    width: 200
    height: 200
  children:
    - type: FarmNode
      id: farm_001
      config:
        position: [50, 50]
        size: [20, 20]
      children:
        - type: InventoryNode
          config: { items: { wheat: 0 } }
        - type: ResourceProducerNode
          config: { resource: wheat, rate_per_tick: 1 }
    - type: CharacterNode
      id: farmer_001
      config:
        name: Jean
        position: [10, 10]
      children:
        - type: NeedNode
          config: { name: faim, threshold: 50, increase_rate: 1 }
        - type: NeedNode
          config: { name: fatigue, threshold: 70, increase_rate: 0.5 }
        - type: InventoryNode
          config: { items: {} }
        - type: AIBehaviorNode
```

4. Plan d'implémentation détaillé

Les étapes ci-dessous doivent être réalisées dans l'ordre. Chaque étape comporte des tâches à accomplir et des livrables attendus. Des tests unitaires devront accompagner chaque étape pour garantir la stabilité.

Étape 1 : Initialisation du projet

1. **Configurer le dépôt Git** : créer un nouveau dépôt GitHub, initialiser un environnement Python (version ≥ 3.9) et configurer un linter (par ex. `flake8`) ainsi que `pytest` pour les tests.
2. **Définir un guide de style** : adopter le style PEP 8, documenter les fonctions et classes avec des docstrings, utiliser des noms de variables explicites et de l'anglais pour le code.
3. **Ajouter un fichier** `README.md` décrivant le projet, son but et la structure de base à venir.
4. **Ajouter un fichier** `requirements.txt` listant les dépendances nécessaires (Python standard pour l'instant, Pygame pourra être ajouté plus tard pour le rendu).

Étape 2 : Implémentation de `SimNode` et du bus d'événements

1. **Créer la classe** `SimNode` dans `core/simnode.py` :
2. attributs `name`, `parent`, `children` ;
3. méthodes `add_child`, `remove_child`, `update`, `serialize` ;
4. mise en place d'un bus d'événements : table de listeners (`{ event_name: [handlers...] }`), méthodes `emit(event_name, **payload)`, `on_event(event_name, handler)`, `off_event(event_name, handler)` ;
5. propagation des événements à travers l'arbre (vers le parent ou vers les enfants, selon une politique simple documentée dans le code).
6. **Créer la classe abstraite** `SystemNode` dérivée de `SimNode` pour les systèmes globaux. Elle pourra redéfinir `update` pour exécuter une logique globale.
7. **Écrire des tests unitaires pour** `SimNode` : vérifier l'ajout/retrait d'enfants, la propagation des événements, l'invocation correcte des handlers.

Étape 3 : Mise en place du registre de plugins et du loader

1. **Créer** `core/plugins.py` :
2. fonction `register_node_type(name, cls)` pour enregistrer un nœud ;
3. fonction `get_node_type(name)` pour retrouver une classe à partir d'un nom ;
4. fonction `load_plugins(module_names)` pour importer dynamiquement des modules.
5. **Définir des contrats d'interface** pour les rôles (`NeedNode`, `InventoryNode`, etc.) et ajouter une fonction de validation dans `register_node_type` pour s'assurer que les classes enregistrées implémentent bien les méthodes requises.
6. **Créer** `core/loader.py` : implémente une fonction `load_simulation_from_file(path)` qui lit un fichier YAML/JSON, construit l'arbre via `get_node_type` et applique les paramètres `config`. Ajouter des tests couvrant des cas simples et des erreurs de format.

Étape 4 : Développement des nœuds génériques

4.1 `InventoryNode`

- Attributs : `items` (dictionnaire `{ nom: quantité }`).
- Méthodes :
- `add_item(name, qty)` et `remove_item(name, qty)` ;

- `transfer_to(other_inventory, name, qty)` ;
- émettre `inventory_changed` après toute modification.
- Tests : opérations d'ajout/retrait/transfer avec succès et échec (par exemple, retirer plus que la quantité disponible).

4.2 `NeedNode`

- Attributs : `name` , `value` , `threshold` , `increase_rate` , `decrease_rate` (optionnel).
- Méthodes :
- `update(dt)` : augmenter ou diminuer la valeur selon l'activité du personnage (paramètre optionnel) ;
- `satisfy(amount)` : réduire la valeur d'un certain montant (manger diminue la faim) ;
- émettre `need_threshold_reached` ou `need_satisfied` lorsque des seuils sont franchis.
- Tests : franchissement de seuil, diminution après satisfaction.

4.3 `ResourceProducerNode`

- Attributs : `resource` (nom), `rate_per_tick` , `inputs` (liste de ressources nécessaires), `output_inventory` (référence vers un `InventoryNode`).
- Méthodes :
- `update(dt)` : consommer les intrants s'ils existent dans l'inventaire, produire la ressource au taux spécifié et stocker le résultat dans l'inventaire ;
- émettre `resource_produced` .
- Tests : production avec suffisamment d'intrants, production bloquée lorsqu'un intrant manque.

4.4 `AIBehaviorNode`

- Attributs : paramètres de comportement (par exemple, priorités des besoins).
- Méthodes :
- `on_event(event_name, payload)` : réagir à des événements de besoins (`need_threshold_reached`), de temps (`phase_changed`) et de production (`resource_produced`) pour changer l'état du personnage (par exemple, se mettre en route vers la ferme, se reposer, consommer de la nourriture).
- `update(dt)` : éventuellement émettre des commandes régulières si nécessaire.
- Tests : écouter un événement et déclencher une action (par exemple, lorsqu'un besoin dépasse un seuil, l'AI choisit d'aller manger).

Étape 5 : Développement des systèmes globaux

5.1 `TimeSystem`

- Paramètres : durée d'un tick, longueur des phases, heure de début.
- Méthodes :
- `update(dt)` : incrémenter le compteur de ticks, déterminer la phase courante et émettre `tick` ou `phase_changed` si besoin.
- Tests : progression des phases, bon échantillonnage du temps.

5.2 `EconomySystem`

- Rôle : écouter les événements d'achat (`buy_request`) et de vente (`sell_request`), vérifier les conditions (prix, stock) et effectuer les transferts d'argent et d'objets. Optionnellement gérer un prix dynamique.

- Tests : succès et échec d'achats/ventes.

5.3 `WeatherSystem` **(facultatif pour la ferme de base)**

- Paramètres : état courant (ensoleillé, pluvieux…) et règles de transition (probabilités).
- Méthodes :
- `update(dt)` : déterminer si la météo change et émettre `weather_changed`.
- Tests : transitions et émision d'événements.

Étape 6 : Composition de la simulation de ferme

1. **Implémenter un** `WorldNode` : conteneur racine qui définit la taille de la carte, contient les `SystemNodes`, les bâtiments et les personnages. Le `WorldNode` délègue l'ajout d'enfants via `add_child`.
2. **Implémenter un** `FarmNode` (classe dérivée de `SimNode`) :
3. doit avoir un positionnement dans la carte ;
4. contient un `InventoryNode` pour le stockage de blé ;
5. contient un `ResourceProducerNode` paramétré pour produire du blé.
6. **Implémenter un** `CharacterNode` : combine un `TransformNode`, plusieurs `NeedNode` (faim, fatigue), un `InventoryNode` et un `AIBehaviorNode`. Les déplacements peuvent être gérés directement par le `TransformNode` ou par la logique de l'IA selon la distance à parcourir.
7. **Écrire un** fichier de configuration minimal** décrivant : un `WorldNode` de taille appropriée, une `FarmNode` et un personnage `CharacterNode` (fermier). Utiliser ce fichier comme base pour initialiser la simulation via le loader.
8. **Écrire des tests d'intégration** : s'assurer que, lorsqu'on lance la simulation, le `ResourceProducerNode` produit du blé, que le fermier va à la ferme, consomme du blé lorsqu'il a faim et se repose lorsqu'il est fatigué.
9. **Ajouter un module de rendu simple (optionnel pour l'instant)** : peut se contenter de logger les événements ou afficher dans la console l'état des stocks et des besoins. Un rendu Pygame plus élaboré pourra être développé ensuite.

Étape 7 : Documentation et guidelines

1. **Documenter chaque classe** dans des docstrings détaillées : rôle, paramètres, événements émis et écouteurs attendus.
2. **Mettre à jour le** `README.md` avec les instructions pour installer, configurer et lancer la simulation, ainsi qu'un exemple de fichier de configuration.
3. **Écrire un fichier** `docs/ARCHITECTURE.md` (ou utiliser ce document) expliquant l'architecture, les interactions entre nœuds et comment créer de nouveaux plugins.
4. **Fournir une "Checklist d'avancement"** à maintenir dans le dépôt (voir section suivante).

5. Checklist d'avancement

La checklist ci-dessous détaille les tâches clés à accomplir. Chaque point devra être marqué comme `[x]` une fois réalisé. Cette liste est volontairement exhaustive et pourra évoluer au fil du développement :

- [ ] Initialiser le dépôt Git, configurer l'environnement et les dépendances.
- [ ] Implémenter la classe `SimNode` avec bus d'événements et écrire des tests unitaires associés.
- [ ] Créer le `SystemNode` de base et écrire un test minimal.

- [ ] Mettre en place le registre de plugins (`core/plugins.py`) et le loader déclaratif (`core/loader.py`).
- [ ] Définir et implémenter `InventoryNode`, avec tests.
- [ ] Définir et implémenter `NeedNode`, avec tests.
- [ ] Définir et implémenter `ResourceProducerNode`, avec tests.
- [ ] Définir et implémenter `AIBehaviorNode`, avec tests.
- [ ] Implémenter les `SystemNodes` : `TimeSystem`, `EconomySystem` (et éventuellement `WeatherSystem`), avec tests.
- [ ] Implémenter `WorldNode` comme conteneur racine.
- [ ] Implémenter `FarmNode` avec `InventoryNode` et `ResourceProducerNode`.
- [ ] Implémenter `CharacterNode` en composant des nœuds génériques, avec logique simple de fermier.
- [ ] Écrire un fichier de configuration YAML minimal pour la simulation de ferme.
- [ ] Écrire des tests d'intégration vérifiant que le fermier travaille, mange et dort comme prévu.
- [ ] Mettre en place un rendu minimal (console ou graphique) pour observer la simulation.
- [ ] Documenter l'architecture et les nœuds, mettre à jour le `README`.
- [ ] Mettre en place un système de seed pour la reproductibilité (optionnel mais recommandé).
- [ ] Préparer les bases pour l'intégration future de nouveaux nœuds et scénarios (plugins supplémentaires).

6. Lignes directrices pour les contributions

- **Commits atomiques** : chaque tâche ou sous-tâche doit être réalisée dans un ou plusieurs commits dédiés, avec des messages explicites (en anglais) décrivant ce qui a été fait.
- **Tests en premier** : lorsque cela est pertinent, écrire les tests unitaires ou d'intégration avant (ou en parallèle) d'implémenter la fonctionnalité.
- **Revue et linting** : exécuter `pytest` et le linter avant de proposer une PR ou de pousser sur la branche principale.
- **Documentation** : tout nouveau nœud ou système doit être documenté, et son rôle décrit clairement dans les commentaires et/ou dans la documentation.
- **Évolution de la checklist** : mettre à jour la checklist à mesure que des tâches sont ajoutées ou complétées.

7. Prochaines étapes après la ferme

Une fois la simulation de ferme opérationnelle et stable, plusieurs pistes d'évolution sont envisageables :

- **Ajout de la météo et des saisons** (via `WeatherSystem`) pour influencer la production ;
- **Ajout d'animaux** (nœuds spécifiques avec besoins, reproduction, production de lait…) ;
- **Développement d'un éditeur visuel** pour assembler des nœuds et générer automatiquement des fichiers de configuration ;
- **Introduction de quêtes ou d'objectifs narratifs** en créant un `QuestNode` et un système de progression de scénario ;
- **Intégration d'armes et de combat** pour envisager des scénarios de défense de la ferme ou de guerre.
