Migration vers une implémentation Multi-Agent

Anthony Hommais
Corentin Fay

Novembre 2025

1 Plan de migration vers un système multi-agent coopératif

Objectif général

L'objectif est de passer d'un unique *Cuisinier* à plusieurs agents (IA et/ou joueurs) capables de coopérer sur les mêmes stations sans corruption d'état ni conflits d'accès. L'approche est incrémentale : d'abord sécuriser les ressources via une API de réservation, ensuite introduire un gestionnaire d'agents, refactorer l'agent, puis ajouter les contrôleurs joueurs, les outils de debug et enfin les tests et la documentation.

Phase A — API de réservation pour les ressources

Mettre en place une API standard de réservation dans les scripts de ressources (`spawner.gd`, `pile_assiettes.gd`, `table_travail.gd`, `table_coupe.gd`, `fourneau.gd`, `zone_livraison.gd`) afin d'empêcher l'utilisation simultanée d'une même station par deux agents.

- Ajouter les membres :
  - `var reserved_by: int = -1`
  - `func reserve(agent_id: int) -> bool`
  - `func release(agent_id: int) -> void`
  - `func is_reserved() -> bool`
  - (optionnel) `func reserve_with_timeout(agent_id:int, timeout_sec:float) -> bool`
- **Comportement** : `reserve` réussit uniquement si la ressource est libre, `release` ne libère que si l'identité correspond, `is_reserved` indique si quelqu'un détient la ressource.
- **Compatibilité** : si une ressource ne définit pas cette API, l'agent la considère comme non réservée.

Phase A.1 — Intégration de la réservation dans give/receive

Adapter les opérations concrètes de transfert d'ingrédients pour respecter la réservation.

- Modifier `give_ingredient` et `receive_ingredient` pour vérifier que, si `reserved_by != -1`, seul l'agent correspondant peut appeler la méthode.
- Ajouter explicitement `agent_id` en paramètre (par exemple `func give_ingredient(agent_id: int)`).
- Libérer la réservation à la fin de l'opération (ou via un appel explicite à `release(agent_id)`).

Phase B — Manager d'agents et instanciation dynamique

Introduire un `AgentManager` centralisant la création, l'identité et la recherche de ressources pour les agents.

- Créer `agents/agent_manager.gd` (singleton ou node de la scène principale) avec des méthodes du type : `spawn_agents`, `register_agent`, `get_nearest_free`, `get_all_agents`, `find_agent_by_id`.
- Ajouter les ressources aux groupes appropriés (ex. `"Spawner"`, `"TableCoupe"`, etc.).
- Modifier `main.gd` pour qu'il utilise le manager afin de créer plusieurs instances de *Cuisinier*.

Phase B.1 — Vérification des scènes et tscn

S'assurer que `cuisinier.tscn` est instanciable plusieurs fois sans dépendances globales fragiles.

- Éliminer les chemins absolus vers d'autres noeuds (pas de `$Cuisinier` global).
- Vérifier que chaque instance possède ses propres composants (labels, point de main, etc.).

Phase C — Refactor du Cuisinier pour le multi-instance

Rendre l'agent conscient de son identité et de son mode de contrôle, et le faire interagir via le `AgentManager` et l'API de réservation.

- Ajouter des propriétés exportées : `@export var agent_id: int`, `@export var control_mode: String`
- Pour trouver des stations génériques, utiliser le manager : demander `get_nearest_free(group_name, global_position)` puis tenter `reserve(agent_id)`.

- Avant de se déplacer vers une station, réserver la ressource ; en cas d'échec, choisir une autre station ou réessayer plus tard.
- À la fin d'une action, libérer la ressource via `release(agent_id)`.
- Propager `agent_id` dans les appels à `give_ingredient`, `receive_ingredient`, `pickup`, `drop`, `deliver`, etc.
- Adapter la construction de recettes : la planification décrit des actions abstraites (par ex. `["pickup", "tomate"]`), et la station concrète est choisie au moment de l'exécution via le manager et la réservation.

Phase C.1 — Files d'actions et logique asynchrone

Adapter la gestion de la file d'actions aux contraintes de réservation.

- Lors d'un pickup , si aucune station n'est libre, l'agent attend un court délai puis réessaie ou tente une autre station.
- Mettre en place un nombre maximal de tentatives ou un timeout pour éviter les blocages.
- Mettre à jour l'`ActionLabel` pour indiquer les états d'attente ou d'échec de réservation.

Phase D — UI / HUD et outils de debug

Améliorer la lisibilité et le debug en contexte multi-agent.

- S'assurer que chaque agent possède un `ActionLabel`.
- Ajouter un HUD léger indiquant l'identifiant de l'agent, son mode (AI/Joueur) et son état courant.
- Ajouter, dans l'`AgentManager`, des logs optionnels pour tracer les réservations et libérations de ressources.

Phase E — Timeouts et robustesse

Gérer les agents bloqués et les réservations qui fuient.

- Implémenter `reserve_with_timeout` pour limiter la durée d'une réservation.
- Ajouter un watchdog dans l'`AgentManager` pour détecter les agents inactifs ou bloqués et relâcher leurs ressources si nécessaire.
- Définir des stratégies de retry pour les agents en attente prolongée.

Phase F — Tests et scènes de validation

Valider le comportement multi-agent de manière systématique.

- Créer des scènes de tests, par exemple :
  - `test_multi_ai.tscn` : deux agents IA sur une même recette.
- Ajouter des tests unitaires ou des scènes scriptées pour vérifier les comportements de `reserve`, `give_ingredient`, etc.
- Critères d'acceptation : aucune ressource utilisée simultanément par deux agents, livraisons correctes, récupération après timeout ou fuites de réservation.

Phase G — Documentation, PR et revue

Finaliser la livraison du travail.

- Documenter l'API des ressources (`reserve`, `release`, `give_ingredient(agent_id)`, `receive_ingredient(agent_id, ingredient)`).
- Expliquer comment ajouter un joueur supplémentaire ou de nouveaux agents, ainsi que les conventions de groupes et d'InputMap.
- Organiser les commits par phase et préparer une Pull Request propre et commentée.

Vue d'ensemble

En pratique, les premières phases (A, A.1, B) fournissent rapidement un socle multi-agent fonctionnel : plusieurs cuisiniers instanciés, partage contrôlé des ressources et absence de conflits d'accès. Les phases suivantes ajoutent progressivement la robustesse, le contrôle joueur, l'ergonomie et la qualité logicielle (tests et documentation).