

Machine d'état client — login, chargement et navigation

Architecture

Le démarrage et la navigation principale utilisent des `GameState`. Chaque état effectue une opération asynchrone puis retourne `Result.TransitionTo(...)` lorsque la condition de transition est remplie.

```
flowchart TD
  INIT[InitBackendState] --> AUTH[AuthState]
  AUTH --> USER[LoadUserState]
  USER -->|prof sans école| TS[TeacherSetSchoolState]
  USER -->|prof| TLS[TeacherLoadSimStubsState]
  TLS --> TLOAD[TeacherLoadSimState]
  TLOAD --> TSCENE[TeacherLoadSceneState]
  USER -->|étudiant sans école| CHOOSE[StudentChooseTeacherAndSimState]
  USER -->|étudiant| SLOAD[StudentLoadSimState]
  SLOAD --> WAIT[StudentWaitForSimStartedState]
  SLOAD --> SCEN[Scenario State]
  SLOAD --> TEAM[StudentChooseTeamState]
  SLOAD --> STUD[StudentState]
```

InitBackendState

Sélectionne l'implémentation backend, récupère l'heure globale, dérive le username si nécessaire, puis passe à `AuthState`.

AuthState

Appelle `Game.Backend.Login()`. En cas de succès, passe à `LoadUserState`. En Editor, un échec peut revenir vers `EditorLoginState`.

LoadUserState

Charge `UserDetails`, crée `Teacher` ou `Student`, applique langue et `GameUid`, initialise heartbeat, met à jour textes/écoles sur WebGL puis choisit le prochain état.

StudentLoadSimState

Après chargement de la simulation, l'état choisit entre attente de démarrage, scénario By Year, scénario By Module, activité non commencée, choix d'équipe ou entrée directe dans `StudentState`.

Teacher flow

Le professeur charge d'abord les simulation stubs. S'il en possède, `TeacherLoadSimState` charge la simulation sélectionnée; sinon il va directement vers la scène de gestion.

Debug

Un client “bloqué” sur un écran n'est pas nécessairement un bug UI. Identifier d'abord le `GameState` courant et la condition de son `Update()` qui n'est jamais satisfaite.

Créer un nouvel état de A à Z

Un `GameState` doit représenter une **étape de navigation ou de chargement**, pas une logique métier arbitraire. Avant d'en créer un, vérifier si le besoin est réellement une transition d'état ou simplement une méthode d'un controller/UI.

1. Choisir la classe de base

Les états sont sous `Assets/Src/Game/GameStates/`. Reprendre le state le plus proche du contexte : étudiant, professeur, login, simulation, etc. Le state doit retourner des `Result` de la machine commune.

2. Mettre l'initialisation dans `TransitionIn()`

`TransitionIn()` sert à lancer l'opération asynchrone une seule fois : backend request, ouverture d'un panel, initialisation d'un flag. Éviter de déclencher la même requête dans `Update()`.

```
public class MyLoadState : GameState
{
    private bool _loaded;

    public override void TransitionIn()
    {
        _loaded = false;
        Game.Backend.SomeRequest(() => _loaded = true);
    }

    public override Result Update()
    {
        if (_loaded)
            return Result.TransitionTo(new NextState());

        return Result.Continue;
    }
}
```

3. Garder `Update()` déterministe

`Update()` doit essentiellement tester les conditions de sortie. Un state qui envoie une requête chaque frame, crée un popup continuellement ou démarre plusieurs tutoriels est presque toujours mal structuré.

4. Brancher la transition depuis le state précédent

La nouvelle classe n'est utile que si un state existant peut la retourner. Rechercher `Result.TransitionTo(...)` dans le flow concerné et brancher la nouvelle condition au bon endroit.

5. Nettoyer dans `TransitionOut()/Dispose` si nécessaire

Si le state attache des callbacks temporaires, listeners UI ou abonnements globaux, ils doivent être retirés lorsqu'il sort. Sinon un retour ultérieur dans le flow peut exécuter le callback plusieurs fois.

6. Tester toutes les branches

- succès backend;
- réponse vide ou utilisateur sans simulation/équipe;
- retour arrière / relog;
- professeur vs étudiant si le state est partagé;
- By Module vs By Year si le flow dépend du `SimType`;
- WebGL, où les erreurs de message sont plus sévèrement protégées.

Debug d'un state bloqué

1. Identifier la classe courante dans la machine.
2. Mettre un breakpoint/log dans `TransitionIn()` pour confirmer qu'il n'est appelé qu'une fois.
3. Inspecter chaque bool/condition testée dans `Update()`.
4. Vérifier que le callback backend est réellement exécuté.
5. Vérifier qu'un popup/tutoriel n'empêche pas la condition UI attendue.
6. Si le state dépend du modèle, vérifier que le controller a reçu son RTT avant de changer de state.

Revision #2

Created 2026-09-18 19:23:38 UTC by Mathis

Updated 2026-09-18 20:07:22 UTC by Mathis