Documentation Technique

# **Introduction**

Dans le cadre de notre projet de développement de jeu vidéo, nous nous sommes interrogés sur le fonctionnement des **State Machines** et la manière de les implémenter efficacement dans un jeu. Ces modèles permettent de gérer des comportements complexes en structurant le système en **états** et en **transitions**, facilitant ainsi la gestion des différentes actions ou réactions du jeu.

Cette documentation explore en détail ce qu'est une **State Machine**, comment l'intégrer dans un projet de jeu vidéo, à quels moments son utilisation est appropriée, les différents types de machines à états, ainsi que les alternatives potentielles pour répondre à certaines contraintes spécifiques.

### **State Machine**

Une **“State Machine”** (ou machine à états) est un modèle qui permet de représenter le **comportement** d’un **système** en termes d’**états** et de **transitions** entre ces états. Ce modèle est utilisé lorsqu’un processus peut être divisé en plusieurs états distincts, avec des **conditions** ou **événements** qui déclenchent les passages d’un état à un autre. À un instant donné, le système se trouve dans un état unique, et il peut changer d’état selon des événements spécifiques, qu’on appelle des **transitions**.

### **Explication des Termes**

Voici une explication de quelques termes essentiels pour comprendre les **state machines** :

* **État** : Une situation ou une configuration particulière dans laquelle le système se trouve à un moment donné. Un état peut être associé à des **actions** ou à des **propriétés** spécifiques qui définissent ce que fait le système à cet instant.
* **Transition** : Le passage d’un état à un autre en réponse à un évènement. Les transitions indiquent comment le système évolue en fonction des conditions rencontrées, comme une commande utilisateur, l’atteinte d’un seuil, ou l’activation d’une fonction.
* **Action** (ou **Sortie**) : Les actions représentent ce que fait le système, soit lors d’une transition, soit lorsqu’il se trouve dans un certain état. Les actions peuvent être déclenchées en entrant dans un état (**action d’entrée**), en le quittant (**action de sortie**), ou au cours d’une transition elle-même.
* **État initial** : L’état de départ du système, où il commence à fonctionner. Il est souvent prédéterminé et sert de point de départ aux premières transitions.
* **État final** : L'état où le système s’arrête. Une fois atteint, il n’y a plus de transitions possibles, et le système cesse de fonctionner (il n’y a pas toujours d’état final).

### **Utilisation**

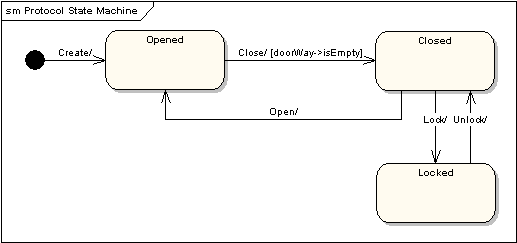
Les **state machines** sont utilisées dans divers domaines, notamment :

* **Systèmes embarqués** : Pour gérer des appareils ou logiciels qui doivent réagir en temps réel à des événements, comme le changement de mode d’un appareil électroménager ou d’un robot.
* **Jeux vidéo** : Pour gérer les **comportements** des **personnages non-joueurs**, des **intelligences artificielles**, ou les **cycles de jeu**, par exemple, le passage entre les états “Attente”, “Attaque”, ou “Patrouille”.
* **Langages de programmation** : Dans les **compilateurs** pour analyser la syntaxe du code et détecter les structures correctes.
* **Interfaces utilisateurs (UI)** : Dans la conception, les state machines peuvent modéliser les différents états d’un **bouton** (par exemple : “Normal”, “Survolé”, “Cliqué”).

En programmation, une state machine est souvent représentée par un **diagramme d’états**, où chaque nœud est un état et chaque flèche représente une transition. En **C++**, les state machines peuvent être implémentées à l’aide de **classes**, d’**énumérations**, ou de **fonctions** qui modélisent le comportement des états et des transitions.

### **Schéma**

Un schéma simple que l’on pourrait envisager pour illustrer une **state machine** serait celui d’un **système de porte**. Par exemple, une porte peut être dans un état **"Ouverte"** ou **"Fermée"**, et la transition entre ces états pourrait être déclenchée par une action de l’utilisateur.



### **Quand utiliser une State Machine ?**

L'utilisation de **machines à états** n'est pas toujours nécessaire. Par exemple, dans le jeu **Flappy Bird**, il n’y a qu’une seule action : voler en battant des ailes en appuyant sur un bouton pour éviter des obstacles. Dans ce cas, il n’y a pas de transitions entre différents comportements complexes. Le personnage est toujours en vol, et les actions se résument à la **gravité** et à l’**impulsion de vol** lors de la pression du bouton. Une simple **gestion d’événements** suffit alors pour contrôler la montée et la descente de l’oiseau.

Cependant, bien que les machines à états soient un outil puissant, elles présentent certaines **limites**. Dans des situations où les **comportements** sont plus complexes, d'autres méthodes peuvent s'avérer plus avantageuses. Par exemple, imaginons un **boss de jeu** avec un comportement complexe : il patrouille, attaque le joueur, esquive, change de phase après un certain seuil de santé, et réagit à plusieurs **stimuli**, comme les attaques du joueur ou des événements dans l'environnement. Dans ce cas, un **Behaviour Tree** (arbre de comportement) pourrait être préférable, car il permet de **prioriser** et de **hiérarchiser** les comportements de manière flexible.

Ainsi, si l’**ennemi** a une santé faible, le comportement de repli peut être prioritaire sur l’attaque, ce qui serait difficile à gérer efficacement avec une simple state machine. On peut imaginer une situation similaire avec un **compagnon du joueur**, qui pourrait prioriser le soin plutôt que l’exploration si le joueur est gravement blessé. Avec une state machine, il serait nécessaire de définir de nombreux états et transitions complexes, rendant la gestion des actions rigide et peu dynamique.

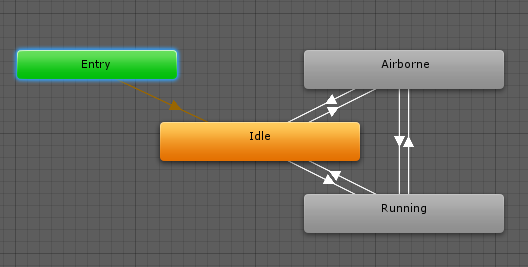
Ces exemples illustrent certaines **limites** des **state machines**, et plusieurs **alternatives**, comme les arbres de comportement, peuvent offrir des solutions plus adaptées à des situations complexes.

### **Exemple dans Unity**

Pour illustrer l'application des **state machines**, prenons un exemple bien connu dans **Unity** : le système d’**animation**. Dans Unity, pour gérer les **animations** et déterminer quand elles doivent se jouer, on crée généralement une **state machine**. Par exemple :

* **Idle State** : Le personnage reste immobile.
* **Running State** : Le personnage court.
* **Airborne State** : Le personnage est en l’air.

Les transitions entre ces états se produisent en fonction d'événements, comme l'entrée d'une commande utilisateur (par exemple, appuyer sur une touche pour attaquer ou courir). Ce système de state machine permet de gérer efficacement les différentes animations et d'assurer que le personnage réagit de manière fluide aux actions du joueur.



# **Les Différentes Techniques de State Machine**

Nous avons découvert qu'il existe plusieurs techniques de **State Machine**. Il est important de les connaître, car elles permettent de modéliser différents types de comportements dans un système. Parmi ces techniques, certaines sont plus répandues et essentielles que d'autres. Voici un aperçu des cinq types les plus notables :

#### **1. Finite State Machine (FSM)**

La **Finite State Machine** (FSM) est la forme la plus simple et courante d'une state machine. Elle est constituée d'un nombre fini d'états, et à chaque instant, le système se trouve dans un état donné. Les transitions entre les états sont déclenchées par des événements ou des conditions simples.

##### **Caractéristiques d'une FSM :**

* **Un état actif à la fois** : À tout moment, la machine est dans un état unique.
* **Transitions déterminées par des événements ou des conditions** : Le passage d'un état à un autre se produit en réponse à des stimuli spécifiques.
* **Modélisation de comportements relativement simples** : Les FSM sont idéales pour gérer des logiques simples et linéaires.

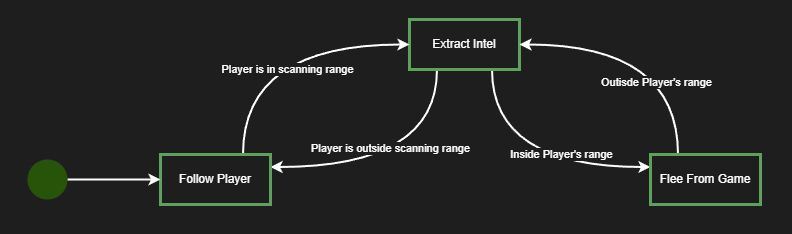
**Exemple** : Imaginons un monstre de type espion qui rôde sur la carte d'un jeu. Ce monstre possède un **état initial** et un comportement défini. Il peut avoir trois états distincts :

* **Follow Player** (Suivre le Joueur)
* **Extract Intel** (Extraire des Informations)
* **Flee from Game** (Se Retirer du Jeu)

Les transitions entre ces états se produisent en fonction de certaines conditions. Par exemple :

* **Follow Player** : Le monstre suit le joueur lorsqu'il est à portée.
* **Extract Intel** : Si le monstre détecte un objet ou un ennemi, il passe à cet état pour recueillir des informations.
* **Flee from Game** : Si le monstre subit trop de dégâts ou se trouve en danger, il choisit de se retirer du jeu.

Voici un schéma pour illustrer cet exemple :



### **2. Hierarchical State Machine (HSM)**

La **Hierarchical State Machine** (HSM) est une extension de la **Finite State Machine** (FSM) qui permet d'organiser les états de manière hiérarchique. Cette structure offre la possibilité de représenter des sous-états et des comportements plus complexes tout en évitant la duplication de code pour des comportements communs.

#### **Caractéristiques d'une HSM :**

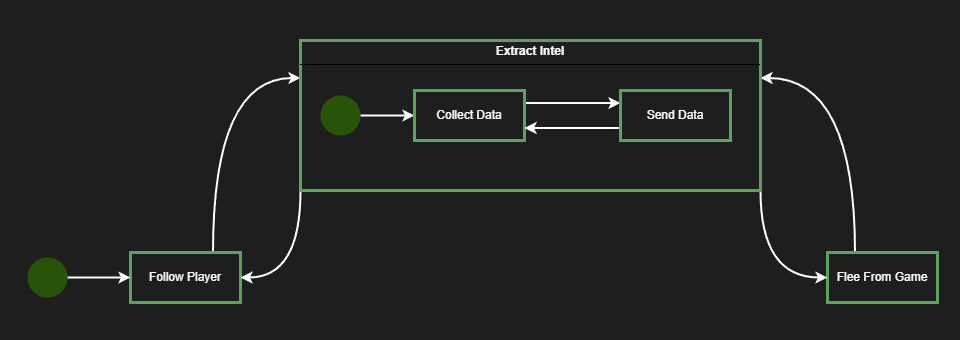
* **Organisation des états en niveaux** : Les états peuvent avoir des parents et des sous-états, ce qui permet de créer une hiérarchie logique des comportements.
* **Réutilisation du comportement** : Un état parent peut fournir un comportement commun à plusieurs sous-états, facilitant ainsi la gestion de la logique sans répétition.
* **Gestion simplifiée des comportements complexes** : En réduisant le nombre de transitions nécessaires, la HSM permet une gestion plus fluide et intuitive des états.

**Exemple** : Pour illustrer la HSM, reprenons l'exemple du monstre de jeu que nous avons précédemment étudié dans le cadre de la FSM. Dans ce cas, nous pouvons structurer l'état **Extract Intel** en tant qu'état parent, avec deux sous-états :

* **Collect Data** (Collecter des Données)
* **Send Data** (Envoyer des Données)

Ces sous-états permettent au monstre de gérer de manière plus spécifique le comportement d'extraction d'informations, tout en gardant une structure claire et hiérarchique.

Voici un schéma illustratif de cette organisation hiérarchique :



**3. Mealy et Moore sont des variantes de FSM**

Concernant les Mealy, les sorties dépendent de l’état actuel mais également de l’entrée. Ainsi, les états peuvent être influencés par l’entrée et pas seulement les transitions.

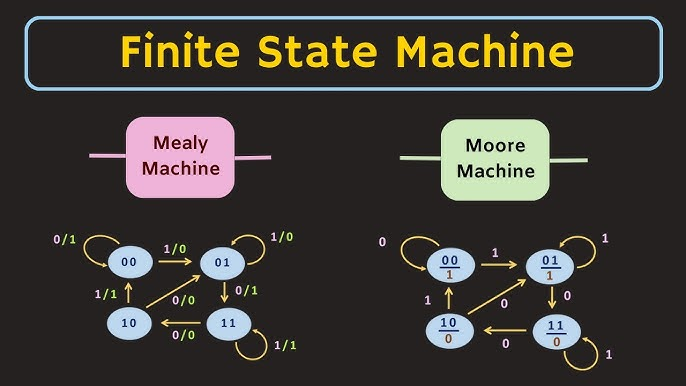
Une Mealy a plusieurs caractéristiques :

* Les actions (ou sorties) dépendent des états et des entrées.
* Plus réactifs aux changements d’entrées car les sorties peuvent changer immédiatement en fonction de l’entrée.

Concernant les Moore, les sorties dépendent uniquement de l’état actuel et non des entrées.Les actions associées à un état particulier ne changent qu’en changeant d’état.

Une Moore a plusieurs caractéristiques :

* Les actions (ou sorties) sont uniquement déterminées par l’état actuel.
* Les transitions d’état sont déclenchées par les entrées mais elles n'affectent pas directement les sorties.



*Dans Mealy, les entrées et les sorties sont notées comme ceci : ‘entrée/sortie’ au-dessus des transitions.  
Dans Moore, les entrées se trouvent sur les transitions et les sorties sont dans les états.*

### **4. Extended State Machine (ESM)**

L'**Extended State Machine** (ESM) est une version améliorée de la **Finite State Machine** (FSM) qui intègre des variables supplémentaires influençant les transitions et les actions. Cette fonctionnalité permet d'enrichir les comportements des systèmes sans nécessiter un nombre accru d'états.

#### **Caractéristiques d'une ESM :**

* **Transitions basées sur des variables** : Les transitions entre états peuvent dépendre non seulement d'événements spécifiques, mais également des valeurs de certaines variables. Cela permet une plus grande flexibilité dans la gestion des comportements.
* **Réduction du nombre d'états** : En utilisant des variables pour capturer des aspects supplémentaires (comme la santé, l'énergie, ou le temps), nous pouvons réduire le nombre total d'états nécessaires tout en maintenant une logique complexe.

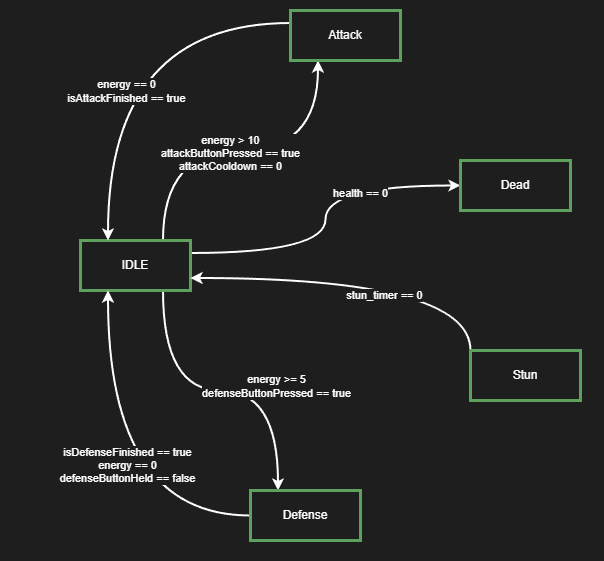
**Exemple** : Imaginons que nous souhaitons modéliser le comportement d'un ennemi pendant une phase d'attaque dans un jeu vidéo. Cet ennemi peut se trouver dans plusieurs états :

1. **Idle** (Inactif)
2. **Attack** (Attaque)
3. **Defend** (Défense)
4. **Stunned** (Immobilisé)
5. **Die** (Mort)

Dans ce contexte, pour passer d'un état à un autre, plusieurs variables doivent être prises en compte, notamment :

* **Health** (Santé) : La quantité de vie restante de l'ennemi.
* **Energy** (Énergie) : La quantité d'énergie disponible pour attaquer ou se défendre.
* **Time** (Temps) : Le temps écoulé depuis la dernière action.

Avec cette approche, nous pourrions créer un schéma de transition simplifié comme suit :



### **Autres Modèles de State Machine**

Bien qu'il existe plusieurs modèles de machines à états, certains sont moins courants et moins souvent présentés sous forme de diagrammes. Voici une explication de chacun d'eux :

#### **1. PDA (Pushdown Automaton)**

Le **Pushdown Automaton (PDA)** est une extension des **Finite State Machines (FSM)** qui utilise une pile pour stocker des informations sur les états précédents. Cela permet de gérer des comportements nécessitant une mémoire plus profonde, comme le retour à des états antérieurs ou la gestion de contextes imbriqués.

**Caractéristiques d'un PDA :**

* **Utilisation d'une pile** : Le PDA mémorise les états précédents grâce à une pile, ce qui permet de conserver l'historique des transitions.
* **Modélisation de comportements récursifs** : Grâce à sa capacité à gérer des informations contextuelles, le PDA est particulièrement utile pour modéliser des comportements récursifs ou des structures de langage imbriquées (comme les parenthèses).

**Exemple d'utilisation :** Un PDA peut être utilisé pour vérifier la validité des expressions relatives dans un langage de programmation.

#### **2. EDSM (Event-Driven State Machine)**

L'**Event-Driven State Machine (EDSM)** réagit spécifiquement à des événements externes. Cette technique est largement utilisée dans les systèmes interactifs, tels que les interfaces utilisateur ou les systèmes embarqués.

**Caractéristiques d'une EDSM :**

* **Transitions déclenchées par des événements externes** : Les changements d'état ne se produisent que lorsqu'un événement externe se produit, permettant une réponse dynamique à l'environnement.
* **Utilisation dans des systèmes asynchrones ou réactifs** : L'EDSM est idéale pour les systèmes qui doivent répondre rapidement à des événements externes, tels que les clics de souris, les entrées utilisateur ou les signaux de capteurs.

**Exemple d'utilisation :** Une EDSM peut être utilisée pour gérer l'état d'un bouton dans une interface utilisateur, où chaque état (normal, survolé, cliqué) est déterminé par les événements de l'utilisateur.

#### **3. AFSM (Asynchronous Finite State Machine)**

L'**Asynchronous Finite State Machine (AFSM)** fonctionne sans horloge centrale, contrairement aux FSM traditionnelles qui sont synchronisées avec un cycle d’horloge. Les transitions peuvent se produire à tout moment en fonction des événements externes.

**Caractéristiques d'une AFSM :**

* **Transitions déclenchées par des événements** : Les changements d'état peuvent se produire à tout moment, ce qui permet une plus grande flexibilité dans la gestion des événements.
* **Utilisation dans des systèmes asynchrones** : L'AFSM est particulièrement adaptée pour les systèmes composés de plusieurs composants fonctionnant à des vitesses différentes ou de manière asynchrone, comme les systèmes distribués ou les applications temps réel.

**Exemple d'utilisation :** Une AFSM peut être utilisée dans les protocoles de communication, où les réponses peuvent être reçues à des moments imprévisibles, nécessitant des transitions d'état immédiates.

Bien que ces modèles de state machines soient moins courants, ils offrent des solutions adaptées à des problèmes spécifiques nécessitant des comportements complexes, une gestion de l'historique, ou une réactivité aux événements. En choisissant le bon modèle, les développeurs peuvent concevoir des systèmes plus robustes et efficaces en fonction des besoins de leurs applications.

### **Application de la Finite State Machine (FSM) en C++**

Dans cette section, nous allons illustrer l'utilisation d'une **Finite State Machine (FSM)** simple en C++. L'objectif de cet exemple est de gérer le comportement d'un ennemi dans un jeu, qui réagit en fonction de la proximité du joueur.

#### **Description du Comportement**

* **État Initial :** Lorsque le jeu commence, l'ennemi est dans un état **"Idle"** (inactif).
* **Transition vers l'État de Fuite :** Si le joueur s'approche suffisamment de l'ennemi, celui-ci déclenche une transition vers l'état **"Run"** (fuite).
* **Transition vers l'État de Flash :** Si le joueur parvient à rattraper l'ennemi, celui-ci passe à l'état **"Flash"** et change de couleur pour signaler qu'il a été attrapé et s'arrête.

#### **Structure de la Classe**

Nous allons définir plusieurs classes pour représenter les états de l'ennemi, ainsi qu'une classe pour gérer la state machine.

1. **Classe de Base : State**
   * Représente l'état de base de la state machine. Elle contient des méthodes virtuelles que chaque état doit implémenter.
2. **Classes d'États Concrets :**
   * **IdleState :** Gère le comportement de l'ennemi lorsqu'il est inactif.
   * **RunState :** Gère le comportement de l'ennemi lorsqu'il fuit le joueur.
   * **FlashState :** Gère le comportement de l'ennemi lorsqu'il est attrapé et clignote.
3. **Classe StateMachine :**
   * Gère les transitions entre les différents états. Elle contient des méthodes pour changer d'état et mettre à jour l'état actuel.

# **Les alternatives**

Lorsque les **State Machines** (FSM) deviennent trop complexes ou rigides pour modéliser certains comportements d'IA, plusieurs alternatives plus dynamiques et flexibles peuvent être utilisées. Ces alternatives permettent de gérer des comportements complexes tout en offrant une meilleure adaptabilité aux conditions changeantes du jeu.

#### **1. Behaviour Tree (Arbre de Comportement)**

Les **Behaviour Trees** sont des structures hiérarchiques couramment utilisées pour contrôler les actions et décisions d'un personnage en IA. Un arbre de comportement est composé de **nœuds** et de **branches**, où chaque nœud représente soit une action, soit une décision à prendre. L'arbre est parcouru de haut en bas, avec des **sélecteurs** (qui choisissent parmi plusieurs actions) ou des **séquences** (qui exécutent des actions dans un ordre spécifique). L'IA sélectionne l'action à exécuter en fonction des conditions actuelles du jeu.

##### **Avantages :**

* **Flexibilité** : Permet de modéliser des comportements complexes tout en conservant une structure claire et facile à gérer.
* **Réactivité** : L'IA peut s'adapter rapidement aux changements dans l'environnement, tels que la santé du personnage, la proximité du joueur ou l'état de l'environnement.

##### **Utilisation :**

Les Behaviour Trees sont parfaits pour les IA qui doivent gérer plusieurs stimuli et ajuster leur comportement sans avoir à gérer une multitude de transitions comme dans une FSM. Ils sont particulièrement utilisés dans les jeux où l'IA doit changer de stratégie de manière fluide, par exemple dans les jeux d'action ou de tir à la première personne (FPS).

#### **2. Goal-Oriented Action Planning (GOAP)**

Le **GOAP** est une méthode d'IA où les agents, au lieu de suivre des transitions d'états prédéfinies, **planifient leurs actions** en fonction d'objectifs à long terme. L'agent évalue son état actuel et génère une séquence d'actions pour atteindre un objectif, comme attaquer un joueur ou se retirer en cas de danger.

##### **Avantages :**

* **Prise de décision dynamique** : Les agents évaluent en temps réel la meilleure séquence d'actions pour atteindre leurs objectifs, offrant ainsi des comportements plus intelligents et moins prévisibles.
* **Complexité des comportements** : Permet à l'IA de planifier plusieurs étapes à l'avance, en tenant compte des changements dans l'environnement.

##### **Utilisation :**

GOAP est utilisé dans des jeux nécessitant des **décisions tactiques**, comme les jeux de stratégie ou de tir. Par exemple, dans le jeu **"F.E.A.R."**, les ennemis prennent des décisions tactiques telles que tendre des embuscades ou se mettre à couvert, en évaluant leur environnement.

#### **3. Utility Systems**

Les **Utility Systems** permettent à un agent IA de choisir la meilleure action en fonction de **valeurs d’utilité** associées à différentes actions possibles. Chaque action est évaluée en fonction de son **importance contextuelle**, et l'IA sélectionne celle qui a la valeur la plus élevée. L’utilité d'une action peut dépendre de plusieurs facteurs, comme la distance à l'ennemi, le niveau de santé ou la proximité des objectifs.

##### **Avantages :**

* **Décision plus nuancée** : Les actions ne sont pas choisies de manière binaire, mais en fonction de priorités pondérées, ce qui permet à l'IA de s’adapter rapidement aux situations complexes.
* **Gestion d’actions concurrentes** : Très utile lorsque plusieurs actions potentielles doivent être évaluées et où les priorités changent dynamiquement.

##### **Utilisation :**

Les Utility Systems sont souvent utilisés dans des jeux de **gestion de ressources** ou des jeux de **simulation**. Un exemple célèbre est **"The Sims"**, où les besoins des personnages (faim, fatigue, sociabilisation) sont constamment évalués pour déterminer quelles actions doivent être entreprises en premier.

#### **4. Scripting Systems**

Les **Scripting Systems** sont des systèmes basés sur des **événements** où le comportement est déclenché par des actions ou des conditions spécifiques définies dans des scripts. L'IA réagit en fonction de certains événements comme l’apparition du joueur ou la proximité d’un objet, et exécute des actions prédéfinies.

##### **Avantages :**

* **Contrôle précis** : Permet de gérer des comportements narratifs ou des événements rares de manière très spécifique sans avoir à créer de nombreuses transitions d’états.
* **Flexibilité narrative** : Idéal pour des séquences complexes ou des événements uniques dans un jeu.

##### **Utilisation :**

Les Scripting Systems sont particulièrement adaptés aux **jeux narratifs** ou aux jeux avec une forte composante cinématographique, comme **"The Last of Us"**, où les interactions des personnages et des ennemis sont souvent déclenchées par des événements prédéfinis pour maximiser l’impact dramatique.

#### **5. Neural Networks / Machine Learning**

Les réseaux neuronaux et les systèmes basés sur le **Machine Learning** permettent à l’IA d’apprendre à partir des données, plutôt que de suivre des règles ou des transitions d’états rigides. L’IA peut s’entraîner à réagir à des situations complexes en fonction des résultats passés et ajuster son comportement pour s'améliorer au fil du temps.

##### **Avantages :**

* **Comportements sophistiqués et imprévisibles** : L'IA peut s’adapter de manière dynamique aux tactiques du joueur, rendant chaque partie unique.
* **Apprentissage continu** : L'agent peut ajuster ses actions en fonction des expériences accumulées, créant ainsi des comportements évolutifs.

##### **Utilisation :**

Le Machine Learning est employé dans des jeux nécessitant une IA hautement adaptative, comme "StarCraft II" où l'IA **AlphaStar** utilise des techniques d’apprentissage profond pour affiner ses stratégies.

**Conclusion**

Dans cette documentation, nous avons exploré le concept de **State Machine** (machine à états) et son utilisation dans différents domaines, notamment dans le développement de jeux vidéo. Nous avons défini les termes clés tels que **état**, **transition**, **action**, **état initial** et **état final**, qui permettent de modéliser efficacement les comportements d’un système.

Nous avons également analysé les différentes techniques de **State Machine**, notamment la **Finite State Machine (FSM)**, la **Hierarchical State Machine (HSM)**, ainsi que les variantes **Mealy** et **Moore**. Chacune de ces techniques apporte des solutions spécifiques pour modéliser des comportements variés, allant des plus simples aux plus complexes.

Nous avons souligné que bien que les **State Machines** soient un outil puissant, elles présentent certaines limites, surtout dans des situations où des comportements plus dynamiques et flexibles sont nécessaires. Dans ces cas, des alternatives comme les **Behaviour Trees**, le **Goal-Oriented Action Planning (GOAP)**, les **Utility Systems**, ou encore des systèmes basés sur le **Machine Learning** peuvent offrir des solutions plus adaptées.

Finalement, la **State Machine** reste une approche fondamentale et largement utilisée en programmation, particulièrement pour gérer des systèmes qui requièrent des transitions d'états bien définies. Cependant, il est important de comprendre ses alternatives afin de pouvoir choisir la meilleure approche en fonction des exigences spécifiques du projet.

# 

# 

# 

# 

# 

# 

# **Bibliographie**

* <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines>
* [https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work\*\*\*\*](https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work****)
* <https://www.gamedeveloper.com/programming/behavior-trees-and-the-future-of-intelligent-control-2>
* <https://www.gamedeveloper.com/design/deep-learning-for-pattern-recognition-in-movements-of-game-entities>
* <https://www.gamedeveloper.com/programming/visual-finite-state-machine-ai-systems>
* <https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm>
* <https://barrgroup.com/blog/introduction-hierarchical-state-machines>
* <https://home.cs.colorado.edu/~srirams/courses/fcps-spring17/lectures/l2.html>
* <https://excaliburjs.com/blog/goal-oriented-action-planning/>
* <https://en.wikipedia.org/wiki/Utility_system>
* <https://www.ibm.com/fr-fr/topics/neural-networks>
* <https://www.igi-global.com/dictionary/scripting-system/25939>