

# GMIN 317 – Moteur de Jeu

## *Structure d'une moteur de jeu*

### Université Montpellier 2

Rémi Ronfard

[remi.ronfard@inria.fr](mailto:remi.ronfard@inria.fr)

<https://team.inria.fr/imagine/remi-ronfard/>

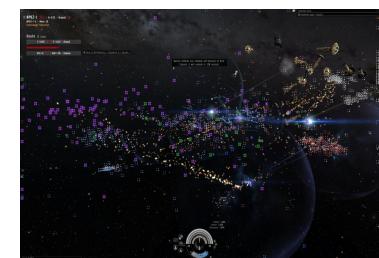
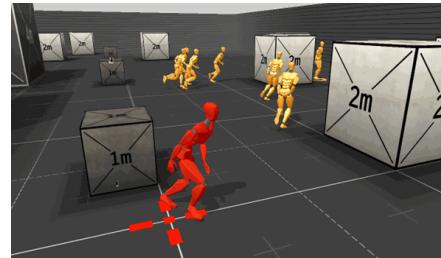
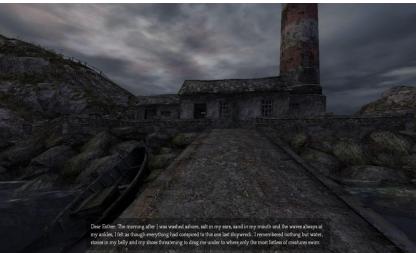
25 septembre 2015

Ce cours est largement inspiré des cours précédents de ***Marc Moulis et Benoit Lange***.

Le but de cette présentation est de fournir une vue globale du workflow de création d'un jeu vidéo et de la structure d'un moteur de jeu, puis de décrire les composants nécessaires à la programmation d'un moteur de jeu et leurs problématiques.

# What is a game engine ?

- Core set of components that facilitate game creation
  - Rendering
  - Physics
  - Sound
  - User input
  - Artificial intelligence
  - Networking
- No such thing as an engine that can support every type of game



# Real-time strategy (RTS) engines

- Large number of low-detail game units
- Multiple levels of AI (individual units as well as computer players)
- Heightmap-based terrain
- Client/server



# Minecraft

- Procedurally-generated block world (voxel-based)
- Simple, pixelated graphics
- Undirected multiplayer gameplay
- Players manipulate the shape of the world



# TP et mini-projets

Based on Qt framework

Huge, cross-platform framework

Qt handles application loop

Your code resides in callbacks

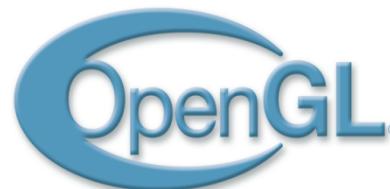
See comments in method stubs for more details

We strongly recommend separating your engine from QGLWidget  
(view.cpp)



- Different game engines define coordinate systems differently

Most of you will probably use the OpenGL coordinate system



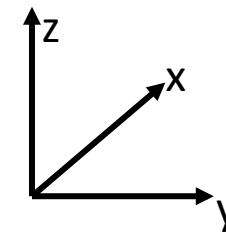
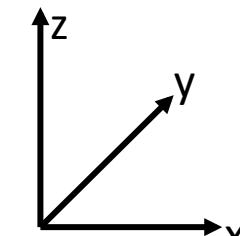
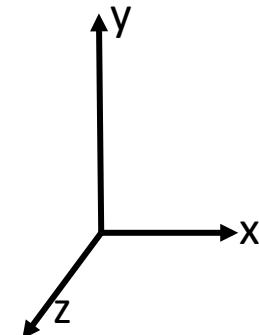
- TAs will strive to be coordinate-system independent

- “Horizontal plane”

Plane parallel to the ground (in OpenGL, the xz-plane)

- “Up-axis”

Axis perpendicular to horizontal plane (in OpenGL, the y-axis)



## Workflow

- Avant d'étudier en détail les composants d'un moteur de jeu, il est important de se familiariser avec le processus de développement d'un jeu.
- Selon l'ampleur et le type de projet, tous les corps de métiers ci-après peuvent ne pas être représentés, ou certaines personnes peuvent endosser plusieurs casquettes.

## • Artistique

- Directeur artistique
- Concept artist
- Graphiste (2D, 3D)
- Graphiste technique
- Animateur
- Designer sonore
- Musicien

## Technique

- Directeur technique
- Développeur

## Production

- Producteur
- Game designer,
- Réalisateur
- Level designer
- Opérateur mocap
- Testeur
- Chargé de production

- **Brainstorming**
  - Propositions de concepts
- **Pré-production**
  - Ecriture du game design document (GDD)
  - Ecriture du technical design document (TDD)
  - Recherches artistiques
  - Mise en place de la chaîne d'outils (export, éditeurs, ...)

## Production

- Ecriture du jeu
- Création des données (assets)
- Mise en place du gameplay
- Test panels
- Polishing

## Testing & validation

- Tests & déboggage
- Soumission pour les TRC
- Production du gold master

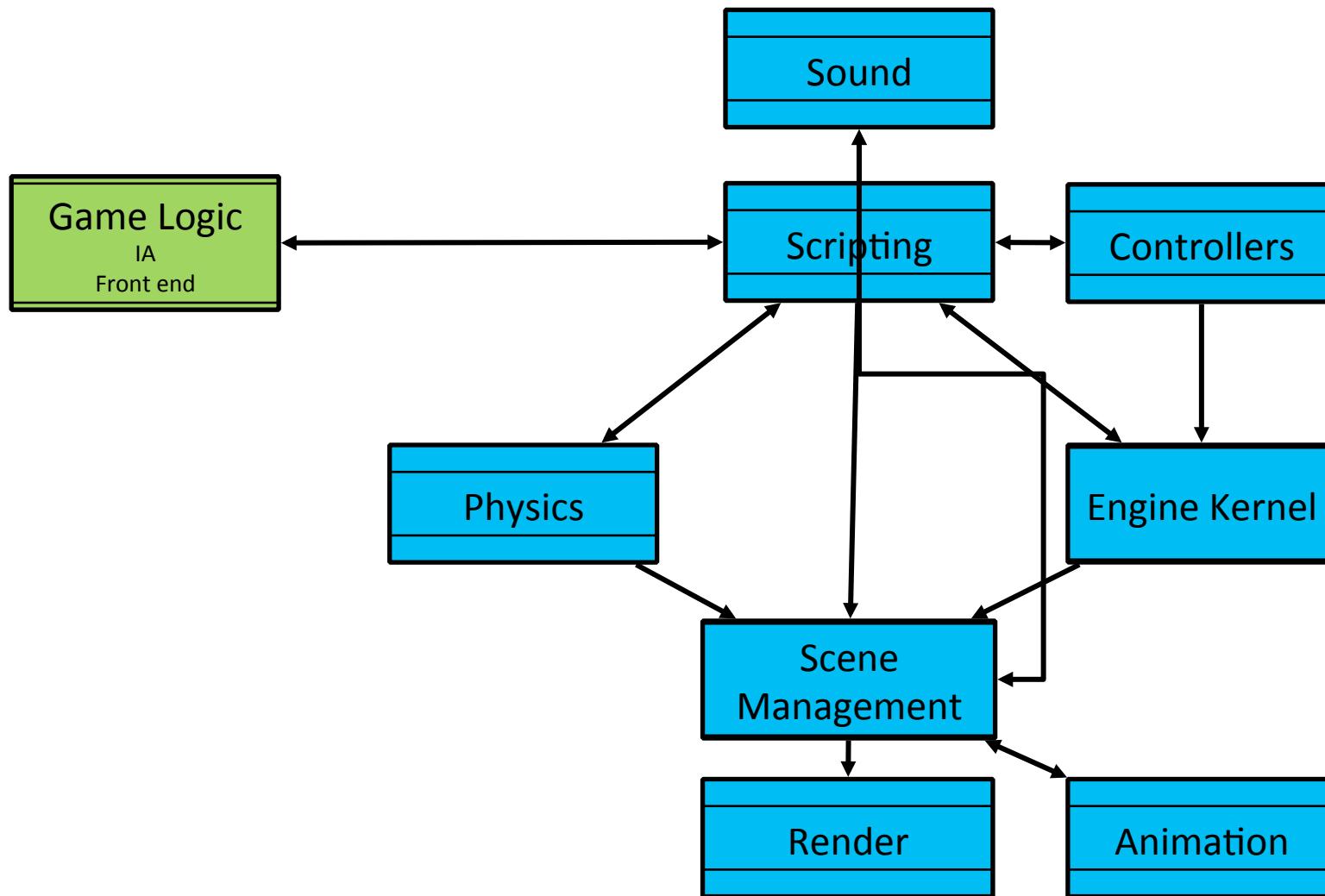
## Post-mortem

- Retours sur expérience

La liste de ces étapes n'est pas forcément exhaustive, et peut varier d'un projet à l'autre ou d'un studio à l'autre. Néanmoins on retrouvera à chaque fois la même structure dans le déroulement de la production.

- On appelle "moteur de jeu" l'ensemble des composants logiciels qui fournissent tous les services nécessaires à l'évolution et l'affichage d'un univers interactif, à vocation ludique.
- On fera la distinction entre:
  - Moteurs first-party: le moteur est tout ou majoritairement développé en interne par le développeur du jeu
  - Moteurs third-party: le moteur est acquis auprès d'une société tierce qui l'a développé (Ex: Unreal Engine, Frostbite, Unity...)

- On distingue globalement 2 courants:
- Moteurs généralistes: Les composants fournissent tous les services utiles pour la mise en œuvre de virtuellement n'importe quel type de jeu (c'est la tendance des moteurs third party : Renderware, Unreal Engine, Unity, Ogre, etc...)
- Moteurs dédiés: Les composants du moteur de jeu sont spécialisés pour un type de jeu précis: FPS, course, aventure, plateforme, RTS, etc...
- NB: *Dans le cadre de cette présentation, nous nous intéresserons principalement à la mise en place d'un framework qui pourrait servir de base commune tant à un moteur généraliste que dédié. Les spécificités techniques de chaque catégorie de jeu ne seront donc pas ou peu abordées.*

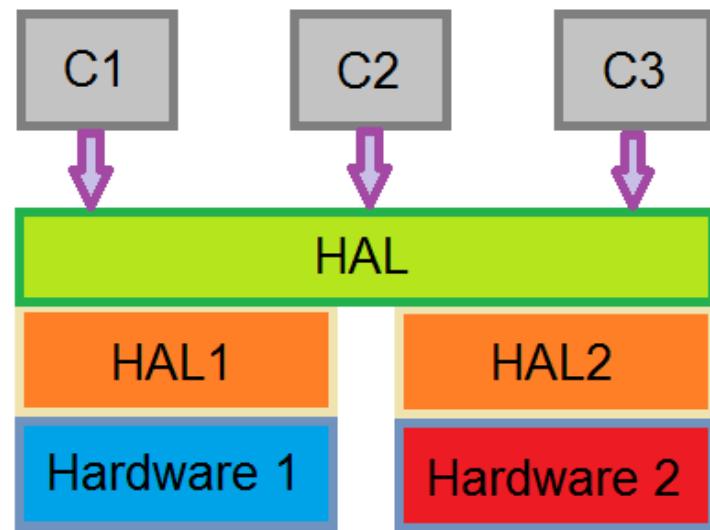


- Notifications
- Gestion mémoire
- Game loop
- Gestion des contrôleurs
- Gestion des données disque
- Gestion du temps
- IA & Comportements
- Interactions avec la scène
- Gestion du son
- Gestion du front end
- Gameplay

- Au vu des moyens techniques et humains nécessaires pour la production d'un jeu moderne, les développeurs font souvent le choix de publier le jeu sur plusieurs machines pour maximiser la rentabilité.
- Le problème est que le développement sur chaque type de machine est différent:
  - Organisation du code (ex: multi-coeurs vs multi-processeurs dédiés)
  - Organisation et capacité de la mémoire

- Le challenge lors de l'écriture d'un moteur multi-plateformes est donc:
  - De maximiser la mise en commun des composants logiciels d'une plateforme à l'autre
  - De minimiser le niveling par le bas
- L'idée est de construire l'ensemble des composants logiciels (génériques) du moteur de jeu sur une base logicielle dédiée (donc spécifique) à chaque plateforme, ce qu'on appelle un couche d'abstraction : « hardware abstraction layer » (HAL).
- Les avantages sont multiples:
  - Le développement du moteur et du jeu deviennent (quasi-)indépendants de la plateforme cible
  - Les développements des différents composants sont relativement décorrélés et donc parallélisables
- Inconvénient: comment éviter le niveling par le bas ?

- On pourra donc utiliser les stratégies suivantes:
  - Redéfinition des types de données de base
  - Surcharge de toutes les fonctions vitales (gestion mémoire, manipulation de chaînes, gestion des noms de fichiers, accès système)
  - Mise en place d'une couche d'abstraction matérielle (I/O, rendu, multi-threading)
  - Gestion d'un pool de ressources "dédiées" (ex: les icônes représentant les boutons du pad)



Composants (C1, C2, C3) et couches d'abstraction matérielle (H1, H2)

## Notifications

- Certains composants du moteur de jeu ont la nécessité de communiquer avec les différentes entités du jeu (IA, joueur, etc...). Ces communications prennent généralement la forme de notifications, qui signalent un événement précis (changement d'état, lecture d'un son, événement IHM, ...). On distingue:
  - *Callback*: L'utilisateur du moteur enregistre une fonction auprès du moteur pour une notification donnée (ex: fin de lecture d'un son). Lorsque l'événement est détecté par le moteur, la callback utilisateur est appelée pour le notifier de l'événement correspondant.

### **Mécanisme relativement optimal.**

- *Polling*: Chaque entité requête auprès du moteur l'état de déclenchement des évènements qui l'intéresse.

### **Equivalent à de « l'attente active ».**

- *Message*: Le moteur de jeu envoie un message dans une file (FIFO) de messages pour notifier de l'événement. Cette file sera lue par l'utilisateur du moteur et chaque message traité séquentiellement.

## Gestion mémoire

- La gestion correcte de la mémoire est un élément clef lors de l'écriture d'un moteur de jeu. Il faut pouvoir répondre aux problématiques suivantes:
  - Contrôler précisément la quantité mémoire utilisée
  - Eviter la fragmentation
- Les stratégies suivantes sont donc couramment mises en place:
  - Allocations de pools mémoire de tailles maîtrisées pour la réalisation de divers traitements (ex: génération de polygones à la volée) ou le stockage d'objets (ex: particules), et allocation des données directement à l'intérieur de ces pools
  - Mise en place d'un système de gestion mémoire personnalisé qui permette de limiter la fragmentation et accélère les allocations/libérations. Le système peut également s'adapter au types de mémoire présents sur la plateforme cible (mémoires dédiées).
  - Insertion d'informations de débogage
- NB: *Les réglages des quantités mémoire attribuées pour chaque tâche sont extrêmement dépendants du jeu (rendu, effets spéciaux, ...). Il faut donc pouvoir configurer facilement la balance des différents pools mémoire (i.e. ils ne doivent pas être fixés par le moteur).*

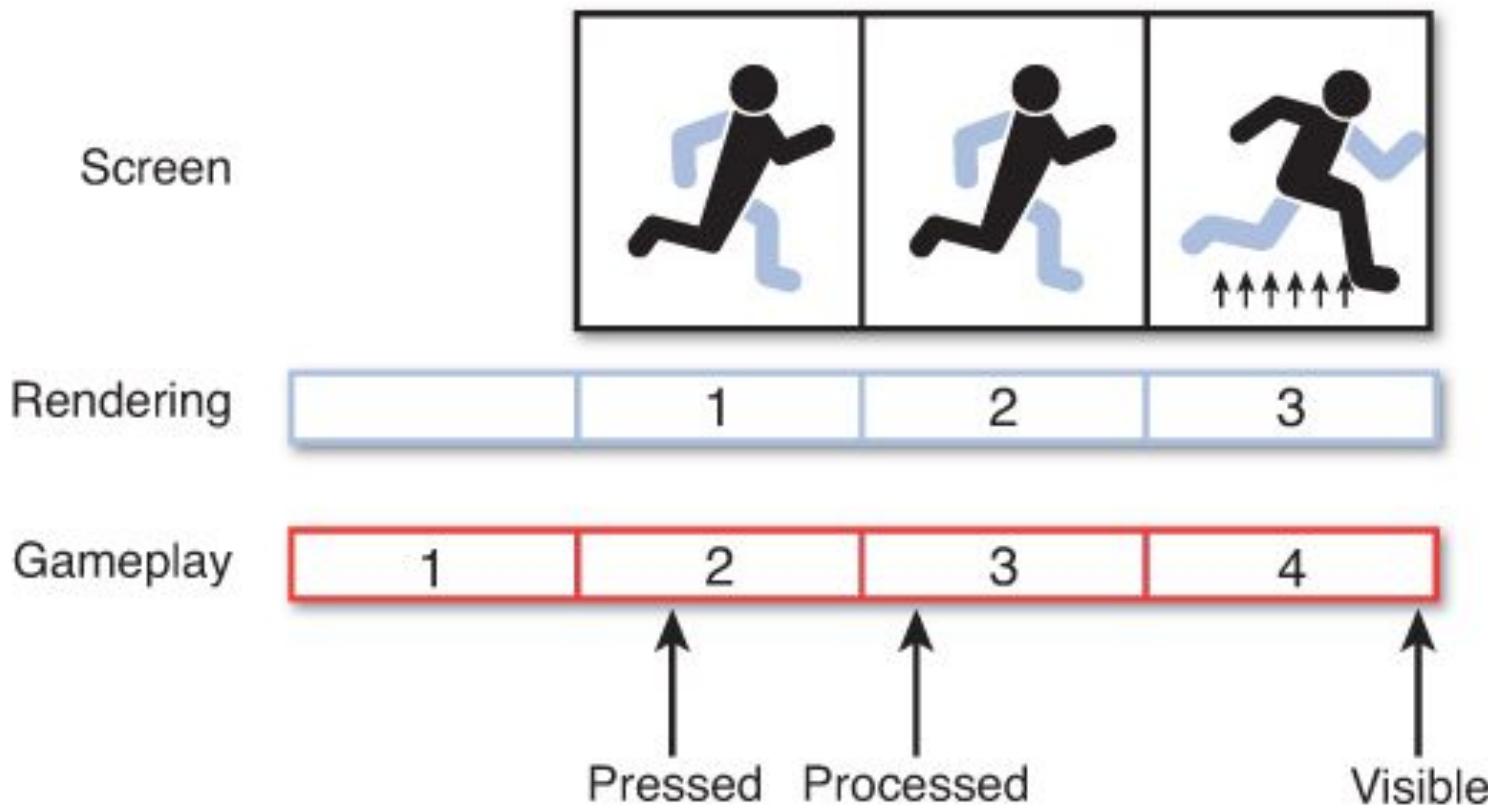
# Game loop

- Le cœur d'un moteur de jeu est la game loop. C'est la boucle principale (active) qui va se charger d'appeler l'intégralité des traitements. Une version haut-niveau a la forme suivante:
- Initialisations
- while (Run)
- {
- Rendu de la scène
- Gestion des évènements système
- Lecture des contrôleurs
- Mise à jour de la scène
- Affichage du rendu
- }
- Libérations
- Sortie

## Game loop

- **Le rendu est en général parallélisé avec la mise à jour de la scène.** Quand ce n'est pas explicitement le cas (multi-threading), on parallélise tout de même le traitement des primitives graphiques avec la gestion de la scène (peut nécessiter la gestion de double-buffers selon les architectures).
- Cela signifie donc que l'image affichée à un temps  $T$  est en fait le résultat de la mise à jour ( $T-1$ ). On différencie ainsi le rendu d'une image, et son affichage réel à l'écran.
- La ligne « mise à jour de la scène » regroupe l'intégralité des mises à jour : sons, chargement des données, physique, IA, gameplay, ....
- **La mise à jour des données et le rendu doivent impérativement être indépendants!** Il ne doit y avoir aucun appel au rendu pendant le processus de mise à jour et vice-versa.

# Game loop



## Exemple : PACMAN

---

```
while player.lives > 0
    // Process Inputs
    JoystickData j = grab raw data from joystick

    // Update Game World
    update player.position based on j
    foreach Ghost g in world
        if player collides with g
            kill either player or g
        else
            update AI for g based on player.position
        end
    loop

    // Pac-Man eats any pellets
    ...

    // Generate Outputs
    draw graphics
    update audio
loop
```

---

# Game loop générique

---

```
while game is running
    realDeltaTime = time since last frame
    gameDeltaTime = realDeltaTime * gameTimeFactor

    // Process inputs
    ...

    // Update game world
    foreach Updateable o in GameWorld.updateableObjects
        o.Update(gameDeltaTime)
    loop

    // Generate outputs
    foreach Drawable o in GameWorld.drawableObjects
        o.Draw()
    loop

    // Frame limiting code
    ...
loop
```

---

## Game objects

```
// Update-only Game Object
class UGameObject inherits GameObject, implements Updateable
    // Overload Update function
    ...
end

// Draw-only Game Object
class DGameObject inherits GameObject, implements Drawable
    // Overload Draw function
    ...
end

// Update and Draw Game Object
class DUGameObject inherits UGameObject, implements Drawable
    // Inherit overloaded Update, overload Draw function
    ...
end
```

---

## Controleurs

- Etat des périphériques de jeu
- Lors de l'écriture du gameplay, il est important de pouvoir accéder aux différents états des périphériques.  
Usuellement, il faut facilement pouvoir déterminer:
  - L'état de transition d'une touche (la touche vient de changer d'état appuyé/relâché)
  - Si une touche est actuellement pressée ou relâchée
  - Depuis combien de temps une touche est dans son état actuel
  - Selon le type de contrôleur, le niveau de pression d'une touche
- On va donc mettre en cache l'état de l'intégralité des touches utiles en début de mise à jour:
  - Coût fixe en cycles machine
  - Mise à jour unique des durées de pression/relâchement
  - Consistance des états tout au long de la boucle
  - Eventuellement parallélisable avec d'autres traitements

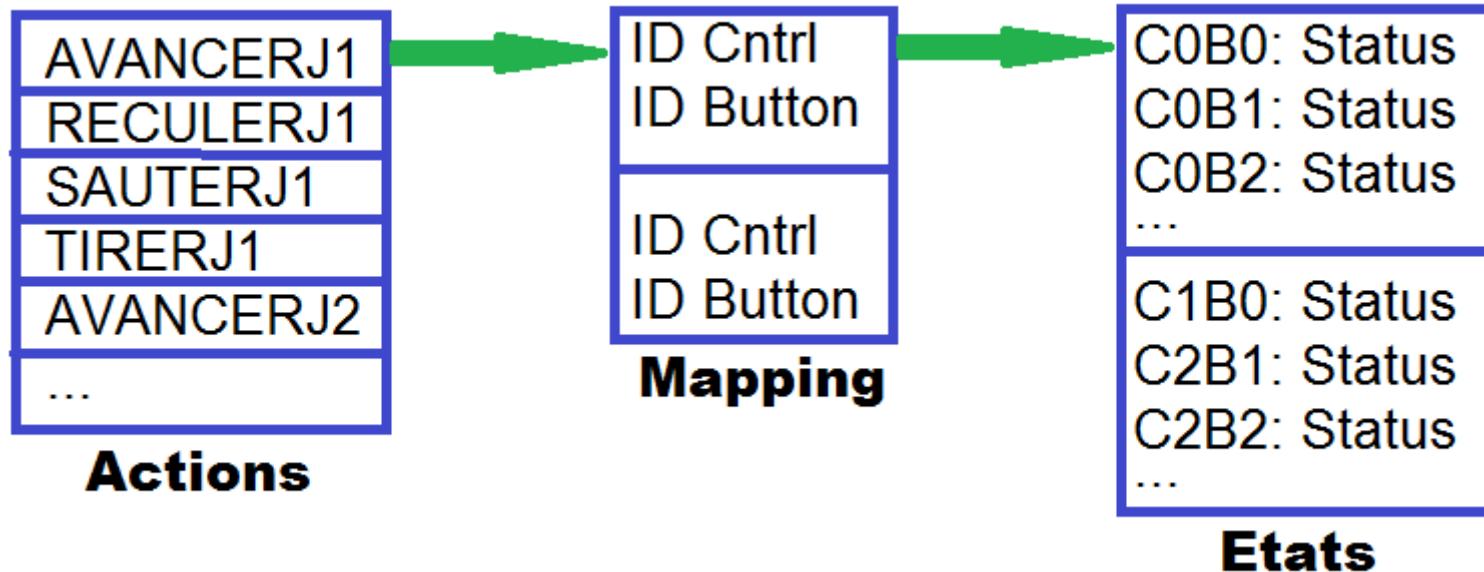
## Périphériques et contrôleurs

- Remapping des actions
- Au niveau de la lecture des périphériques, les cas de figure suivants sont hautement probables:
  - Branchement de périphériques de types différents pour le contrôle du jeu (ex: volant *vs* joystick)
  - Lecture indifférenciée d'évènements de plusieurs périphériques différents pour contrôler une même action (ex: clavier + joystick déclenchent le même événement)
  - Classes d'évènements différentes associées à une même action (ex: déplacement souris (continu) *vs* pression clavier (binaire))

## Périphériques et contrôleurs

- On peut donc envisager la stratégie de gestion suivante, à deux niveaux d'indirection:
  1. Définir au niveau du jeu (pas du moteur !) une liste d'actions unitaires (ex: AVANCERJ1, RECULERJ2, SAUTER, etc..)
  2. Enregistrer auprès du moteur de jeu une table de mapping entre les actions unitaires et les entrées de périphériques valides pour cette action
  3. Le moteur se charge, en se basant sur la table de mapping, de faire le lien entre les actions du jeu et les états bas-niveau des périphériques (combinaison de touches, normalisation des valeurs des inputs)
  4. Des paramétrages optionnels globaux peuvent être disponibles pour configurer la réactivité de certains périphériques (ex: zone neutre des sticks analogiques, sensibilité souris)
- L'utilisation d'une telle table de mapping a également l'avantage de pouvoir facilement redéfinir les touches, sans avoir à changer le code du jeu.

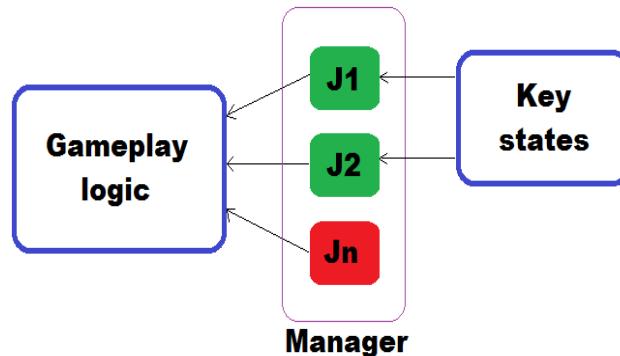
# Périphériques et contrôleurs



*Remapping d'actions pour la gestion des contrôleurs: chaque action pointe vers une liste de contrôleurs/boutons possibles (mapping), utilisée par le gestionnaire pour vérifier les états bas-niveau de chacun des contrôleurs (CnBn).*

# Périphériques et contrôleurs

- Gestionnaire d'actions
- Dans certains cas spécifiques, on peut vouloir pousser encore plus loin le concept d'abstraction des périphériques d'entrée.
- Par exemple, on peut vouloir être capable de remplacer à la volée un joueur humain par un joueur IA dans un jeu multi-joueurs.
- On va donc utiliser un troisième niveau d'indirection qui simulera un périphérique d'entrée, et sur lequel viendra se brancher un gestionnaire d'actions.
- Lorsque le joueur humain est au contrôle, la lecture du périphérique d'entrée est directe (passthrough)
- Lorsque le gestionnaire "joueur" est remplacé par un gestionnaire "IA", l'IA simule un périphérique d'entrée en se basant sur la table de mapping des actions.



*Gestionnaire d'actions*

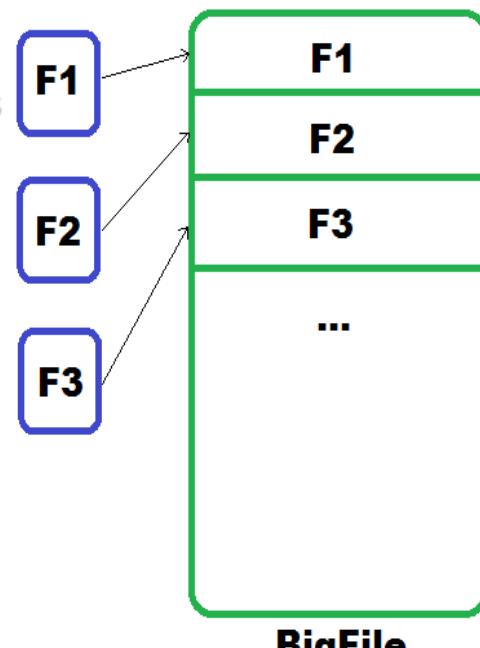
## Gestion des accès disques

- Généralement, les contraintes suivantes doivent être respectées:
  - Les chargements/sauvegardes doivent être **asynchrones** (i.e. non-bloquants)
  - Les chargements doivent être le moins longs possible (i.e. compression des données)
  - L'organisation des données sur disque doit suivre certaines règles (dépend du **TRC**)

# Gestion des fichiers

Une organisation courante des données disque est sous forme de "**bigfile**": l'intégralité des fichiers de données est concaténée dans un seul gros fichier. Les avantages sont multiples:

- Le fichier forme un premier niveau de protection contre l'accès au contenu
- L'agencement des fichiers à l'intérieur du *bigfile* peut être fait en fonction des accès aux données, afin d'optimiser les déplacements de la tête du lecteur
- Le format est compatible avec les contraintes courantes d'organisation des fichiers établies par les **TRC** (nombre de fichiers présents sur le disque)
- Pendant les phases de développement, il est facile d'accéder et de transporter les dernières données du jeu. Dans l'idéal, un serveur automatique peut même générer un nouveau *bigfile* incrémental chaque fois que de nouveaux *assets* sont disponibles.



En surchargeant les fonctions de lecture/écriture des fichiers au niveau du moteur, il est même possible de travailler de manière transparente sur une vraie hiérarchie de fichiers sur disque pendant la phase de développement.

*Organisation d'un bigfile*

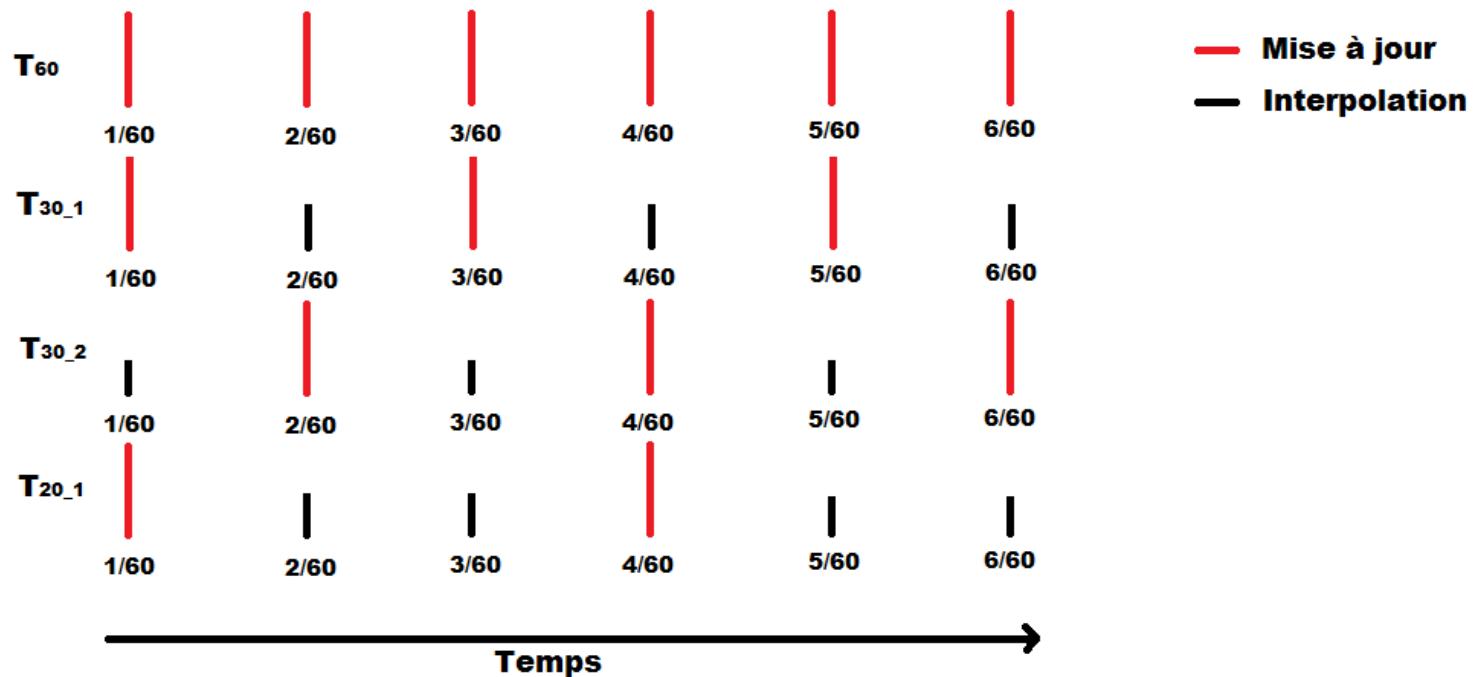
## Timers

- La mise à jour des différentes entités d'un jeu nécessite d'être synchronisée à l'aide de timers. Ces objets permettent de signaler, en fonction de leur fréquence propre, à quel moment une mise à jour est nécessaire.
- La mise à disposition de plusieurs timers différents va permettre la gestion d'effets spécifiques: freeze de scène (bullet time), mise en pause du gameplay, ralentis, etc...
- Lorsqu'une scène comprend un grand nombre d'**acteurs dynamiques** (préférablement IA), il peut être intéressant de mettre en place des stratégies de mises à jour distribuées, afin d'alléger la charge processeur, tout particulièrement lorsque la mise à jour est lourde en calculs (par exemple : physique).
- L'idée sous-jacente est de mettre en place un mécanisme de mise à jour des acteurs qui ne va traiter qu'une sous-partie d'entre eux à chaque nouvelle boucle d'affichage. Le reste des acteurs (non traités) est mis à jour par simple interpolation.
- Exemples d'usages : mise à jour du décor, physique des objets secondaires, IA des PNJ secondaires, systèmes de particules, etc...

# Timers

- La technique présentée ici se base sur un système de timers multi-fréquences. Le fonctionnement est le suivant:
  - On fixe une fréquence cible de mise à jour des acteurs de la scène (par exemple 60 Hz).
  - On détermine un ensemble de timers diviseurs de cette fréquence cible :  $T_{60}$ ,  $T_{30\_1}$ ,  $T_{30\_2}$ ,  $T_{20\_1}$ ,  $T_{20\_2}$ ,  $T_{20\_3}$ ,  $T_{15\_1}$ ,  $T_{15\_2}$ ,  $T_{15\_3}$ ,  $T_{15\_4}$ , etc...
  - Les acteurs sont associés par le code du jeu (de manière manuelle et/ou automatique) aux groupes de timers désirés (selon divers critères au choix : charge CPU, priorité de mise à jour, etc)
  - A chaque boucle de rafraîchissement, la mise à jour des acteurs se base sur la fréquence de référence (60Hz, soit  $dT = 1/60$ ). Si le temps cumulé entre 2 mises à jour est multiple de la fréquence du timer associé à l'acteur, alors la fonction de mise à jour complète de l'acteur est appelée. Sinon, la position/ orientation de l'acteur sont interpolées entre les 2 position/orientation calculées par les 2 mises à jour complètes précédentes.
  - Pour éviter les pertes de  $dT$ , on conserve le reste de la division du temps cumulé par la fréquence de référence (cf. le chapitre sur la mise à jour en temps constant).

# Timers



Un biais de ce système est que ce qui est affiché à l'écran est en retard de n images (n dépendant du timer de rafraîchissement) par rapport au calcul (mais en général ce n'est pas gênant).

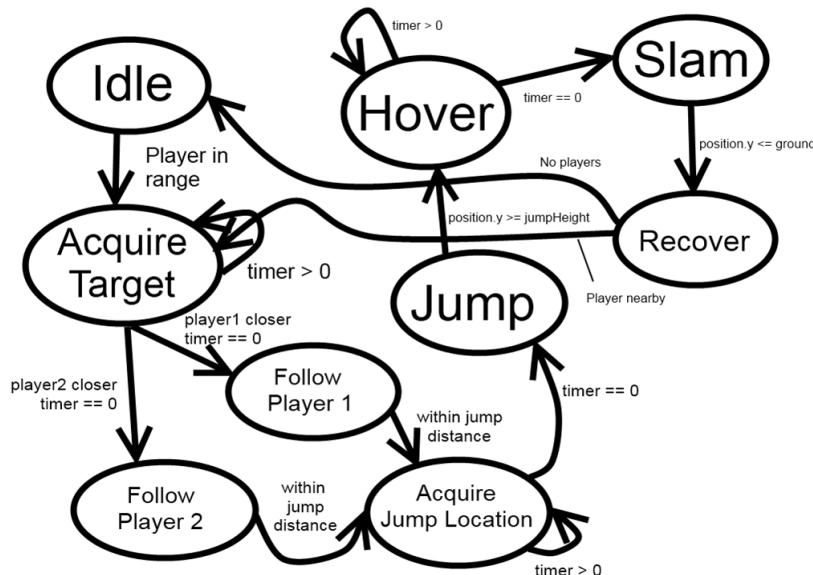
# IA et comportements

Les modèles comportementaux les plus courants sont les graphes d'états.

Le comportement d'une entité est modélisé par un ensemble de nœuds, représentant les états mutuellement exclusifs de cette entité, et par les transitions possibles entre ces états. A tout instant donné, un état (et un seul !) est actif dans le graphe.

***Il est impératif d'apprendre à réfléchir de manière incrémentale !***

L'intérêt de ce type de structure est qu'elle est très facile à implémenter, et monopolise en général très peu de ressources.



Exemple de machine à états

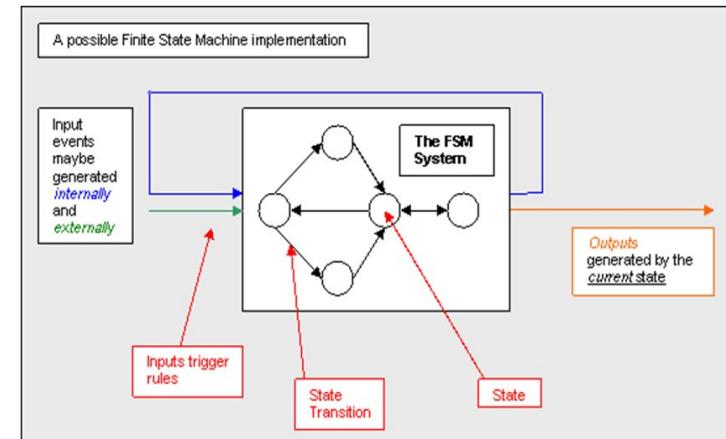
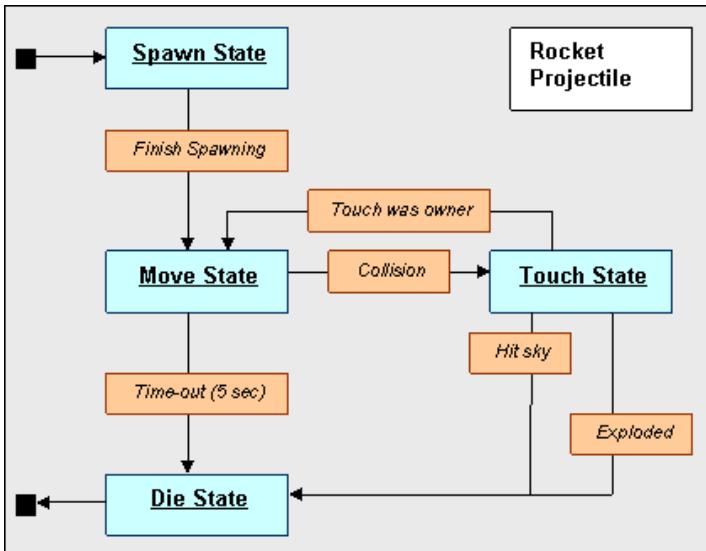


Schéma global d'une machine à états

# IA et comportements



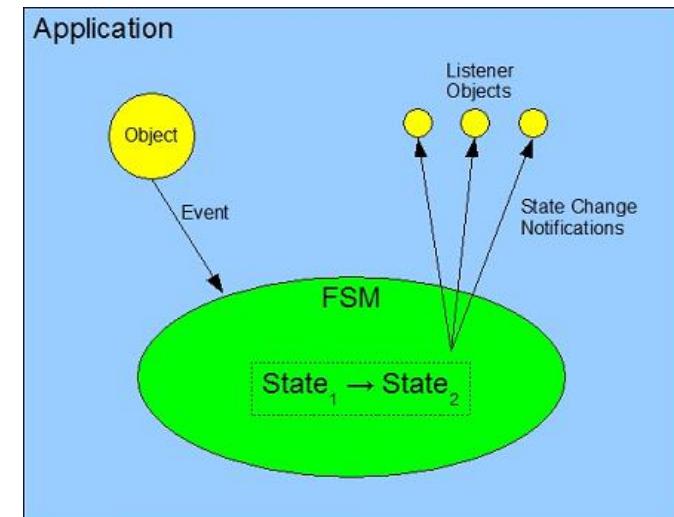
A chaque état peut être associée une ou plusieurs fonctions, dont le rôle sera de mettre à jour automatiquement le comportement de l'entité.

On peut enrichir le comportement de l'IA par une approche multi-échelles:

- Le graphe d'états définit le comportement au niveau d'une entité
- Un gestionnaire de décisions définit le comportement haut-niveau (la volonté, en quelque sorte) de l'ensemble des entités

Le gestionnaire de décision peut baser son modèle comportemental sur un ou plusieurs types de données:

- Une modèle statistique des actions à entreprendre
- Une réaction à l'environnement
- Une pondération aléatoire sur des comportements types
- ...



## Interactions

- On distingue plusieurs niveaux d'interaction avec la scène:
  - Utilisation de points remarquables
  - Déclenchement d'évènements
  - Réactions à la topologie, à la matière

## Points remarquables

- Il est très utile au niveau de l'éditeur (ou exporteur) d'être en mesure d'exporter des positions 3D précises dans la scène:
  - Points de démarrage
  - Positionnement d'entités (bonus, ennemis, interrupteur, émetteur de particules, émetteur sonore...)
  - Nœuds des chemins
- La gestion du format de l'export est laissée à la discrétion de l'éditeur/exporteur de données. En cas d'éditeur non dédié, une solution peut par exemple consister en l'utilisation d'une norme de nommage des objets dans l'outil de modélisation.

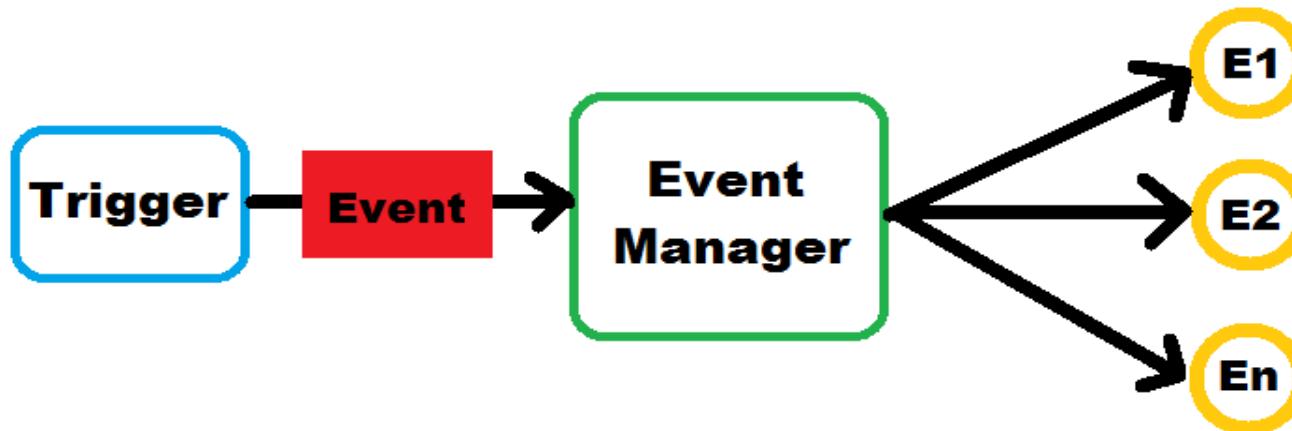
## Déclenchement et notification d'évènements

- Un univers de jeu est, par essence, un univers interactif. Pour répondre aux besoins du scénario, il faut que les entités du jeu soient en mesure de déclencher des évènements.
- En définissant un ensemble d'évènements uniques au niveau global du jeu (pas du moteur !), il est possible d'identifier sans équivoque les évènements qui sont déclenchés au cours du jeu.
- La grande majorité des évènements est le plus généralement déclenchée par le franchissement d'une zone de la scène. On peut donc envisager la mise en place d'objets "déclencheurs", placés dans la scène, qui auront les propriétés suivantes:
  - Topologie : plan, sphère, boîte englobante, ...
  - Déclenchement du/des événements associés sur pénétration/occupation/sortie de la zone
  - Déclenchement unique/récurrent
  - Déclenchement associé à un type d'entités (PNJ, joueur, ennemis, alliés, ...)

## Gestion des événements

Un gestionnaire d'évènements global peut se charger de collecter l'ensemble des évènements déclenchés, et de les mettre à disposition de l'ensemble des entités. En poussant plus loin le concept, on peut imaginer que les entités du jeu s'enregistrent auprès du gestionnaire d'évènements, afin d'être automatiquement notifiées du déclenchement des évènements qui les intéressent (**solution lourde si beaucoup d'entités**).

La notification d'un évènement peut être gérée comme un stimulus extérieur à la machine à état des entités, qui lui permet de passer dans un état spécifique pour la réponse à l'évènement.

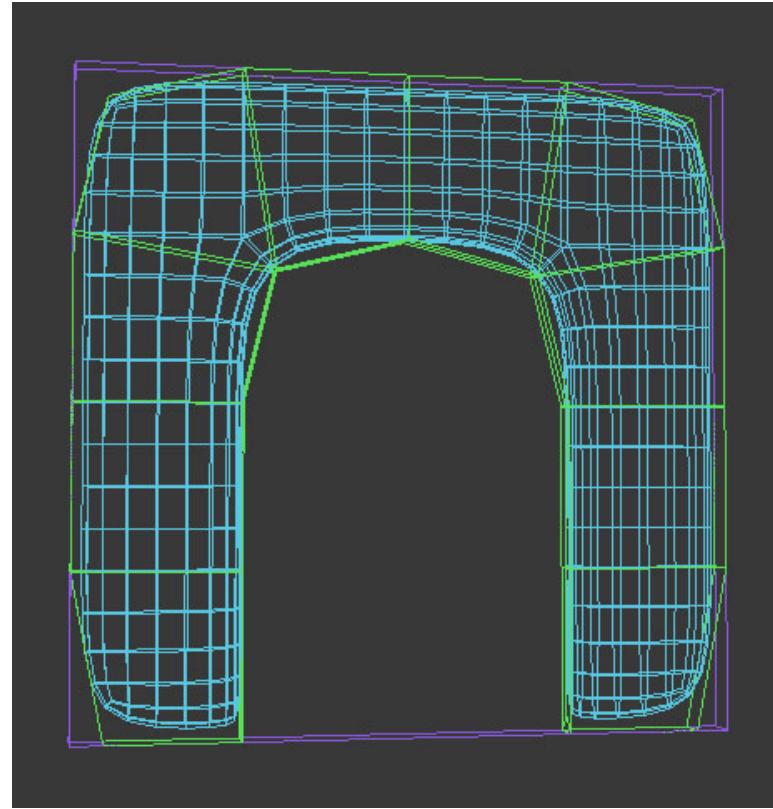
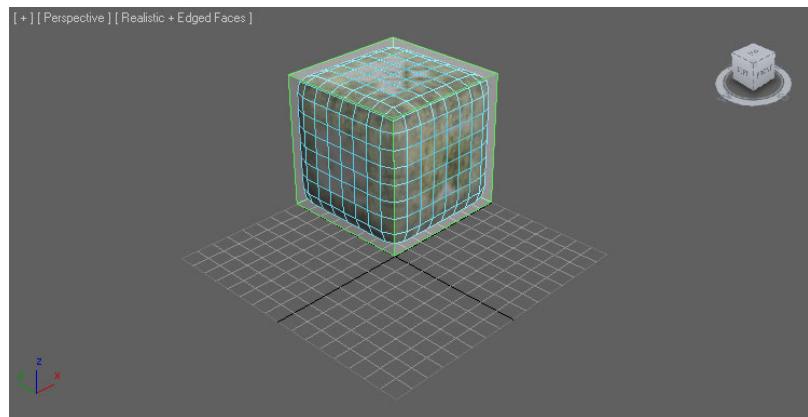
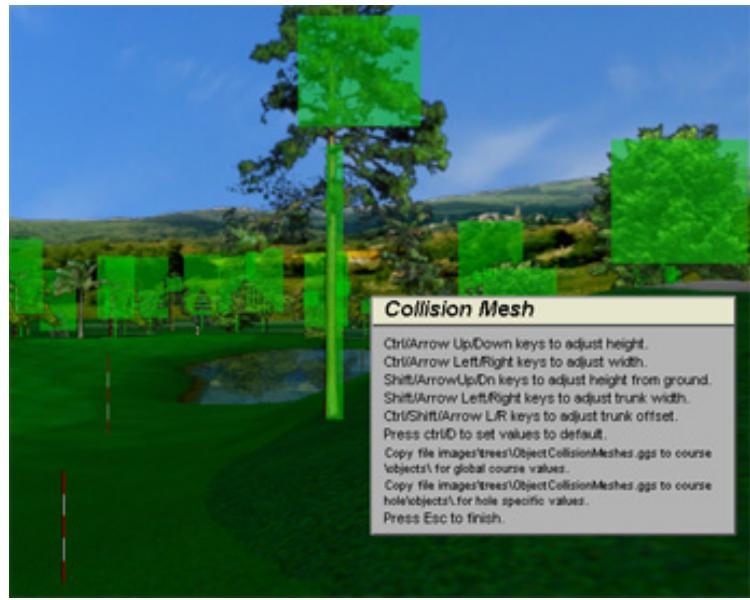


*Notification d'un évènement à toutes les entités concernées par le gestionnaire d'évènements*

# Interactions physiques / topologiques

- La physique des entités de la scène nécessite la détection et la réponse correcte aux collisions avec l'environnement. Cependant, il peut arriver que les polygones de la scène forment un ensemble difficilement utilisable par les routines de gestion de collisions, et ce pour les raisons suivantes:
  - La scène peut être extrêmement détaillée, et contenir des zones difficiles à gérer pour le moteur de collision, ou gourmandes en calculs inutiles (ex: un escalier, une sculpture, ...)
  - Pour accélérer les calculs, on peut représenter certaines entités physiques par un modèle ponctuel. Dans ce cas, la représentation polygonale de l'acteur interpenetrerait les polygones de la scène lors des collisions
  - Les polygones de la scène ne forment pas obligatoirement un support solide (ex: neige poudreuse sur le sol, étendue d'eau, ...)
- 
- Pour pallier à ces problèmes, on peut utiliser une seconde scène simplifiée, invisible, adaptée aux contraintes du moteur physique: la **carte des collisions** (collision map).
- **NB:** Dans le même esprit, un maillage de collision (**collision mesh**) peut être associé aux entités de la scène pour simplifier le traitement des collisions et de la physique, indépendamment de l'usage d'une collision map.

# Boites de collision



# Matériaux et comportements

- Il est également nécessaire de pouvoir réagir correctement aux types de matériaux de la scène:
  - Bruit des pas, émission de particules, impacts, ...
  - Effets sur les acteurs (gain/perte d'énergie, ...)
  - Modification des paramètres physiques du contact (frottement, inertie, ...)
- On utilise souvent un identifiant global qui permet de déterminer le type de matière avec laquelle une entité est en contact. Selon les besoins/contraintes, ce type d'information peut être stocké à différents endroits (face, objet, matériau), mais on le retrouve le plus usuellement au niveau des groupes de polygones (matériaux) de la collision map.
- Les routines de test d'intersection avec la géométrie peuvent alors optionnellement renvoyer l'identifiant matériau d'un polygone donné.
- **NB:** La liste exhaustive des matériaux peut au choix être fixée par le moteur, soit par le jeu lui-même

# Musique interactive

C'est la capacité à jouer la musique en relation avec une situation de jeu. La difficulté de l'exercice tient dans l'enchaînement correct des séquences musicales. Dans tous les cas, prévoir une ou plusieurs des propriétés suivantes:

- Lecture contextuelle de séquences sonores qui constituent la musique en s'ajoutant à une base de fond sonore
- Cross-fading d'une musique à l'autre
- Définition de marqueurs dans la musique qui définissent les points de transition possibles (à définir avec le musicien)



Le jeu vidéo *Extase* (Amiga) était une référence en matière de musique interactive

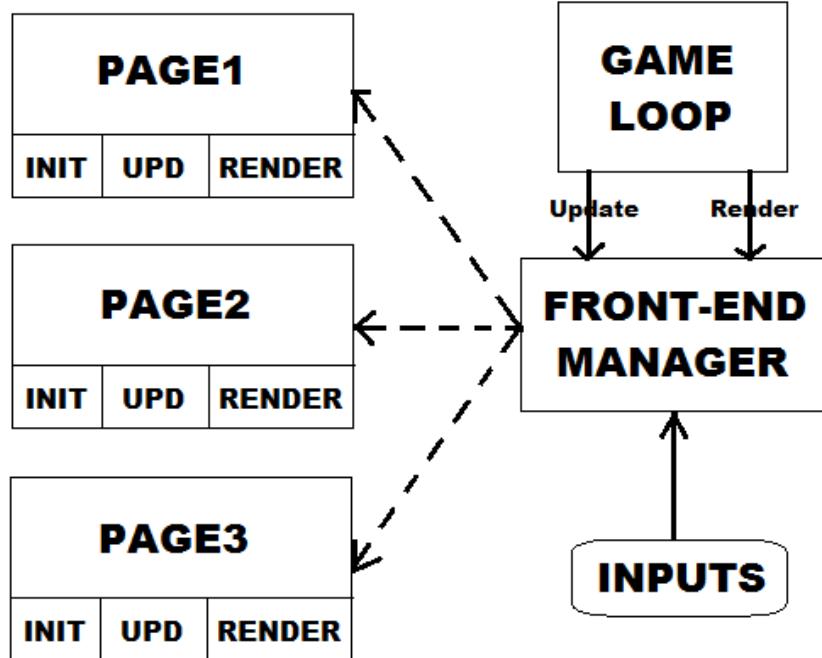


Autre référence moderne en musiques interactives: *Journey* (PS3)

## Interface utilisateur

- L'interface d'un jeu se limite rarement à l'affichage de la scène. Il est souvent nécessaire d'afficher des informations 2D supplémentaires, pour représenter:
  - le HUD (head-up display)
  - les menus de configuration, choix de niveau, ...
  - les menus internes au jeu (pause, inventaire, ...)

# Interface utilisateur



Afin de ne pas complexifier la gestion des interfaces, et de garder une architecture modulaire, on pourra envisager la mise en place de la structure suivante:

- On conserve la gestion de chaque page d'IHM dans un module séparé
- Chaque page d'IHM expose au minimum une fonction de construction, mise à jour, affichage, et destruction de son contenu
- Un manager d'IHM gère l'enchaînement des pages et les appels aux callbacks. Ce manager est intégré dans les appels depuis le moteur de jeu (mise à jour, rendu)

L'utilisation conjointe du manager de pages et des timers permettra la réalisation d'affichages d'IHM synchrones ou asynchrones.

Certains moteurs tiers (ex: Unreal Engine) proposent l'intégration directe de technologies type Flash pour la création/édition des IHM du jeu.

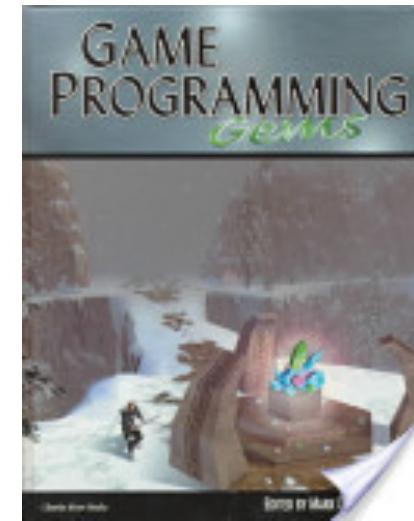
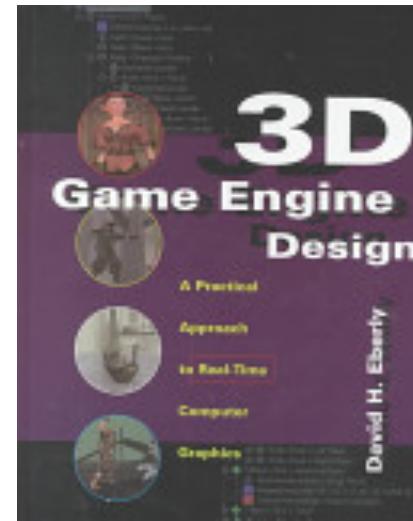
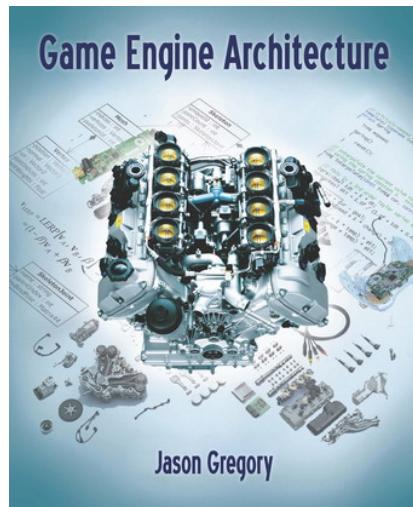
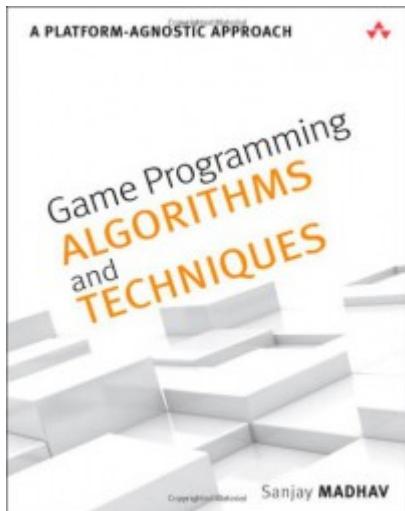
# Gameplay : programmation ou script

	<b>Avantages</b>	<b>Inconvénients</b>
<b>Programmation</b>	<ul style="list-style-type: none"><li>• Le langage natif est utilisé, il n'y a donc pas d'overhead dû à la machine virtuelle</li><li>• Pas de perte de temps pour l'écriture d'un langage de script et de la machine virtuelle associée</li><li>• Accès direct à l'intégralité des objets du moteur</li></ul>	<ul style="list-style-type: none"><li>• Les gameplay designers doivent savoir programmer dans le langage (ou travailler en binôme avec un programmeur)</li><li>• Le jeu nécessite d'être recompilé à chaque modification majeure (possibilité de limiter les changements par le biais de fichiers de configuration externes)</li><li>• Effets de bord possibles</li></ul>
<b>Scripting</b>	<ul style="list-style-type: none"><li>• Le langage est en général bien simplifié par rapport à un langage de programmation traditionnel</li><li>• L'exécution du script est très cloisonnée</li><li>• Le gameplay peut être modifié à la volée, sans recompiler voire recharger le jeu</li></ul>	<ul style="list-style-type: none"><li>• Développement long de la machine virtuelle</li><li>• Overhead à l'exécution</li></ul>

## Gameplay

- La tendance penche tout de même en faveur de l'utilisation de scripts. La programmation directe est probablement à privilégier dans le cas de micro-projets, où les moyens et le temps imparti sont réduits (à moins de bénéficier d'une technologie déjà existante).
- Prévoir également, à chaque fois que c'est possible, de fournir une paramétrisation externe des constantes de gameplay (ex: paramètres physiques, timings, etc...) par le biais de fichiers de configuration texte. Dans ce cas, les gameplay designers peuvent ajuster le gameplay sans intervention d'un développeur.

## Livres utiles pour compléter ce cours



- Voir aussi les conférences GDC (game developers conference) et SIGGRAPH