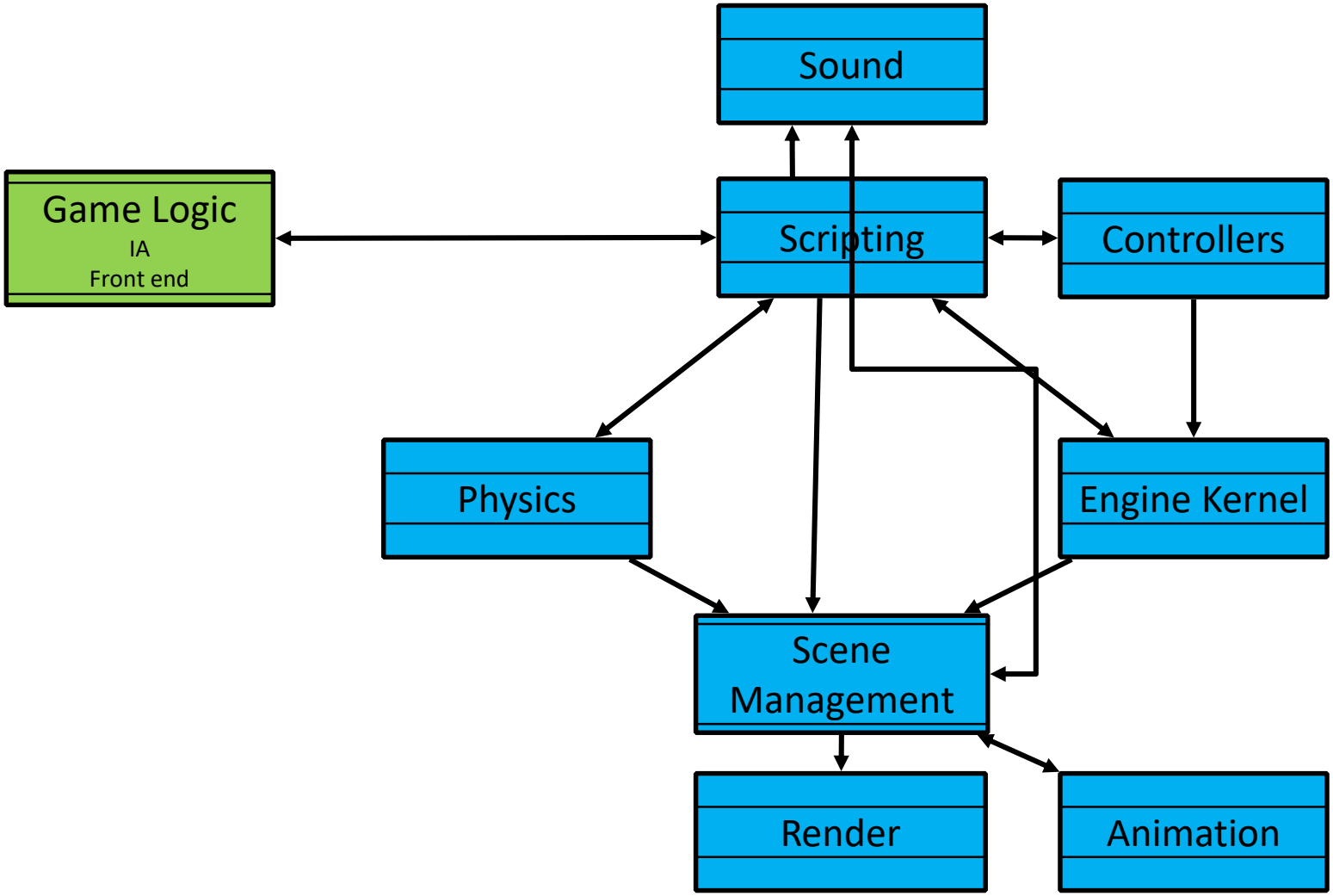
Moteurs de jeux,

Structure d'un moteur de jeu

- « Moteur de jeu » : l'ensemble des composants logiciels fournissant tous les services nécessaires à l'évolution et l'affichage d'un **univers interactif**, à vocation ludique.

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


Sound
Game Logic
IA
Front end
Scripting
Controllers
Physics
Engine Kernel
Scene Management
Render
Animation

- Jeu moderne :
  - moyens techniques et humains importants,
  - publication du jeu sur **plusieurs machines** pour maximiser la rentabilité.

- PB : développement différent sur chaque type de machine :
  - Organisation du code (ex: multi-coeurs vs multi-processeurs dédiés)
  - Organisation et capacité de la mémoire

- Le challenge moteur multi-plateformes :
  - maximiser la mise en commun des composants logiciels d'une plateforme à l'autre
  - minimiser le nivelage par le bas

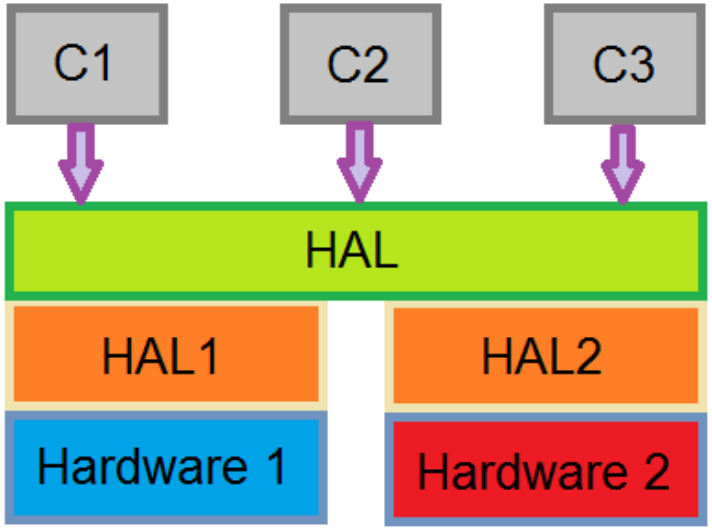
- **Idée** : construire l'ensemble des composants logiciels (génériques) du moteur de jeu sur une base logicielle dédiée (donc spécifique) à chaque plateforme,

→ couche d'abstraction : **« hardware abstraction layer » (HAL)**.

Standardisation de manipulation du matériel informatique tout en **cachant les détails techniques** de la mise en œuvre

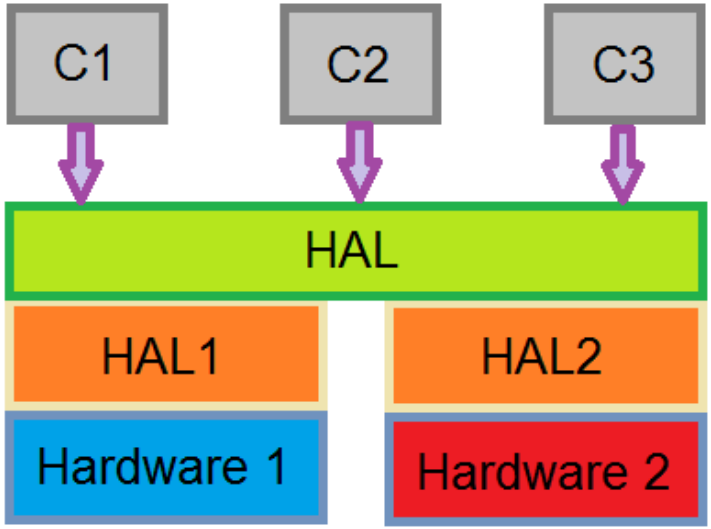
- Les avantages sont multiples:
  - développement du moteur et du jeu (quasi-) indépendants de la plateforme cible
  - développements des différents composants (relativement) décorrélés et donc parallélisables

Inconvénient: comment éviter le nivelage par le bas ?



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

- Stratégies :
  - 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)
  - Couche d'abstraction matérielle (I/O, rendu, multi-threading)
  - Gestion d'un **pool de ressources** "dédiées" (ex : les icones représentant les boutons du pad)



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

Notifications

Certains composants **doivent** communiquer avec les différentes entités du jeu (IA, joueur, etc…).

→ **Notifications** : signalent un événement précis (changement d'état, lecture d'un son, événement IHM, …).

- *Message* : moteur envoie un message dans une file (FIFO) pour notifier de l'événement → **traité séquentiellement**.
- *Polling* : entité requête auprès du moteur l'état de déclenchement des évènements l'intéressant → **« attente active »**.
- *Callback* : utilisateur du moteur enregistre une fonction auprès du moteur pour une notification donnée (ex: fin de lecture d'un son). Evénement détecté par le moteur →**callback utilisateur** appelé pour le notifier.

Gestion mémoire

- Elément clef de l'écriture d'un moteur de jeu :
  - Contrôler **précisément la quantité mémoire utilisée**
  - Eviter la fragmentation

- Stratégies courantes :
  - 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**
  - Système de gestion mémoire personnalisé : limite la fragmentation et accélère les allocations/libérations. Adaptation au types de mémoire présents sur la plateforme cible (mémoires dédiées).
  - Insertion d'informations de débuggage

- **NB** : Quantités mémoire attribuées pour chaque tâche dépend du jeu (rendu, effets spéciaux, …) → configuration facile de la balance des pools mémoire (i.e. non fixés par le moteur).

Game loop

- Le cœur d'un moteur de jeu est la **game loop** : boucle principale (active) se chargeant d'appeler l'intégralité des traitements.

Une version haut-niveau :

```
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

- Rendu en général parallélisé avec la mise à jour de la scène (multi-threading).

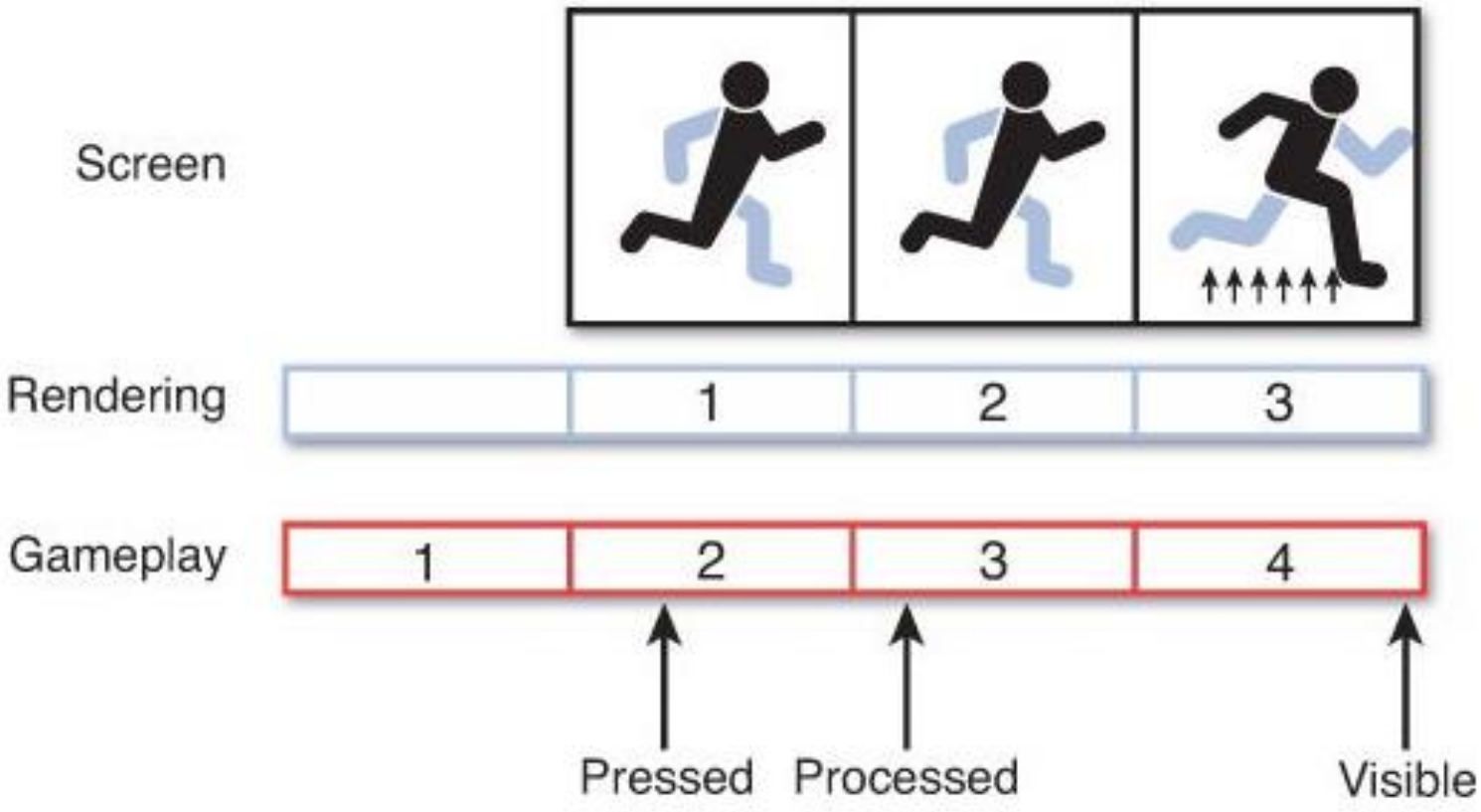
- Parallélisation du **traitement des primitives graphiques avec la gestion de la scène** (double-buffers selon les architectures).

→ image affichée à un temps T est le résultat de la mise à jour (T-1).
rendu d'une image ≠ son affichage réel à l'écran.

- « 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**!

Game loop


Screen
Rendering
1
2
3
Gameplay
1
2
3
4
Pressed
Processed
Visible

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
```

Contrôleurs

- Etat des périphériques de jeu

- Pouvoir accéder aux **différents états des périphériques** :
  - L'état de transition d'une touche (la touche vient de changer d'état appuyé/relâché)
  - Touche 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

- Mise en cache de 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

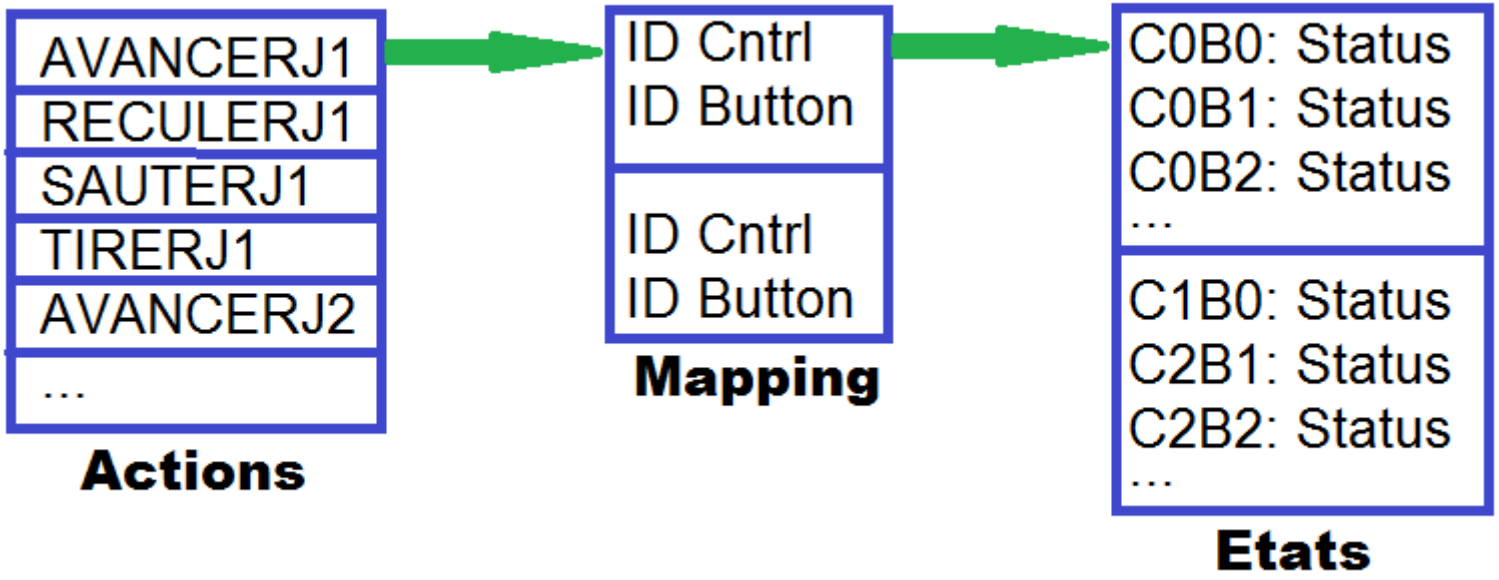
- Au niveau de la lecture des périphériques :
  - Branchement de périphériques de **types différents** pour le contrôle du jeu (ex: volant *vs* joystick)
  - Lecture indifférenciée d'évènements de plusieurs périphériques différents pour **contrôler une même action** (ex: clavier + joystick déclenchent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

Périphériques et contrôleurs

- Stratégie de gestion possible, 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 fait le lien (table de mapping) 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)

- Utilisation d'une table de mapping permet de 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).*

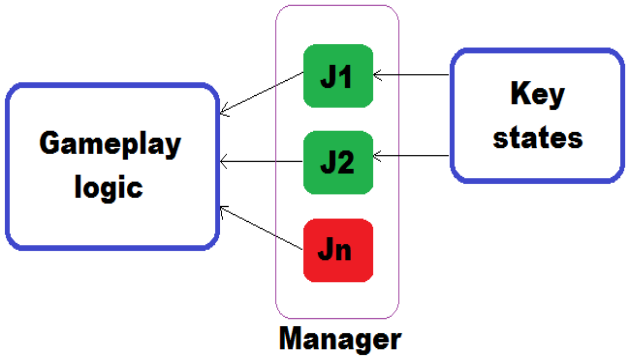
Périphériques et contrôleurs

Gestionnaire d'actions

Pousser plus loin le concept d'abstraction des périphériques d'entrée (eg, être capable de remplacer à la volée un joueur humain par un joueur IA dans un jeu multi-joueurs)

→ utiliser un **troisième niveau d'indirection** simulant un périphérique d'entrée, et sur lequel viendra se brancher un **gestionnaire d'actions**.

- Joueur humain au contrôle : lecture du périphérique d'entrée directe (**passthrough**)
- Gestionnaire "joueur" remplacé par un gestionnaire « IA » : IA simule un périphérique d'entrée en se basant sur la table de mapping des actions.



Gestion des accès disques

- Contraintes suivantes à respecter :
  - 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 Technical Requirements Checklist - **TRC**)

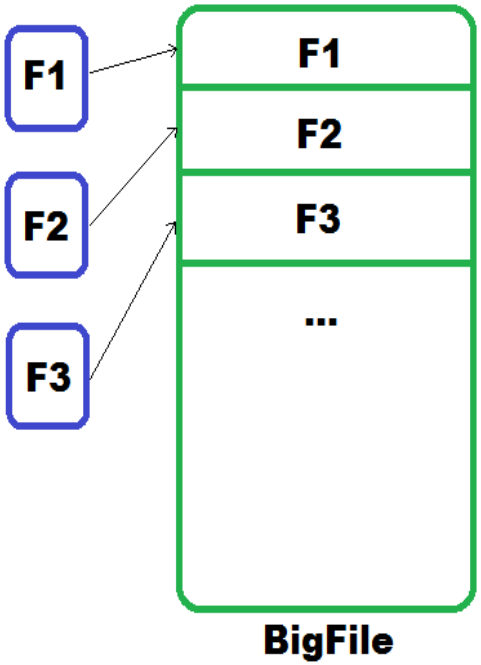
TRC : liste de tests fixée par une plateforme qu'un jeu doit passer afin d'y être publié

Gestion des fichiers

Organisation commune sous forme de « **bigfile** » : intégralité des fichiers de données concaténée dans un seul gros fichier.

Avantages

- premier niveau de protection contre l'accès au contenu
- agencement des fichiers à l'intérieur du *bigfile* en **fonction des accès aux données** → optimisation des déplacements de la tête du lecteur
- format compatible avec les contraintes courantes d'organisation établies par les **TRC** (nombre de fichiers présents sur le disque)
- Pendant les phases de développement, facile d'accéder et de transporter les dernières données du jeu.
- Idéalement, un serveur génère un nouveau *bigfile* incrémental chaque fois que de nouveaux *assets* sont disponibles.

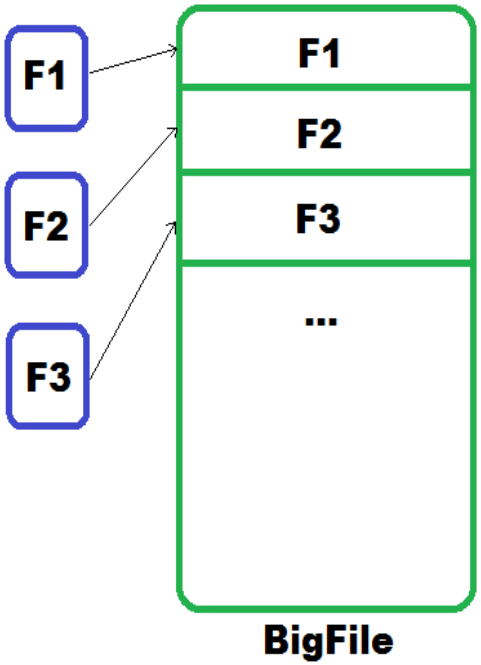


*Organisation d'un **bigfile***

Gestion des fichiers

Organisation commune sous forme de « **bigfile** » : intégralité des fichiers de données concaténée dans un seul gros fichier.

- Surcharge des fonctions de lecture/écriture des fichiers au niveau du moteur : possible de travailler de manière transparente sur **une vraie hiérarchie de fichiers** sur disque pendant la phase de développement.



*Organisation d'un **bigfile***

Timers

Mise à jour des différentes entités d'un jeu nécessite d'être **synchronisée à l'aide de timers** :

- objets permettant de signaler, en fonction de leur **fréquence propre**, à quel moment une mise à jour est nécessaire.

- plusieurs timers différents : gestion d'effets spécifiques → freeze de scène (bullet time), mise en pause du gameplay, ralentis, etc…

- si grand nombre d'**acteurs dynamiques** (IA) dans la scène :
  - mise à jour distribuées (allégement de la charge processeur),
  - tout particulièrement lorsque la mise à jour est lourde en calculs (par exemple : physique).

Timers

- Mises à jour distribuées :
  - L'idée sous-jacente :
    - mettre en place un mécanisme de mise à jour des acteurs ne traitant qu'une **sous-partie d'entre eux** à chaque nouvelle boucle d'affichage
    - Le reste des acteurs (non traités) mis à jour par interpolation.
  - Exemples d'usages :
    - mise à jour du décor,
    - physique des objets secondaires,
    - IA des PNJ secondaires,
    - systèmes de particules, etc…

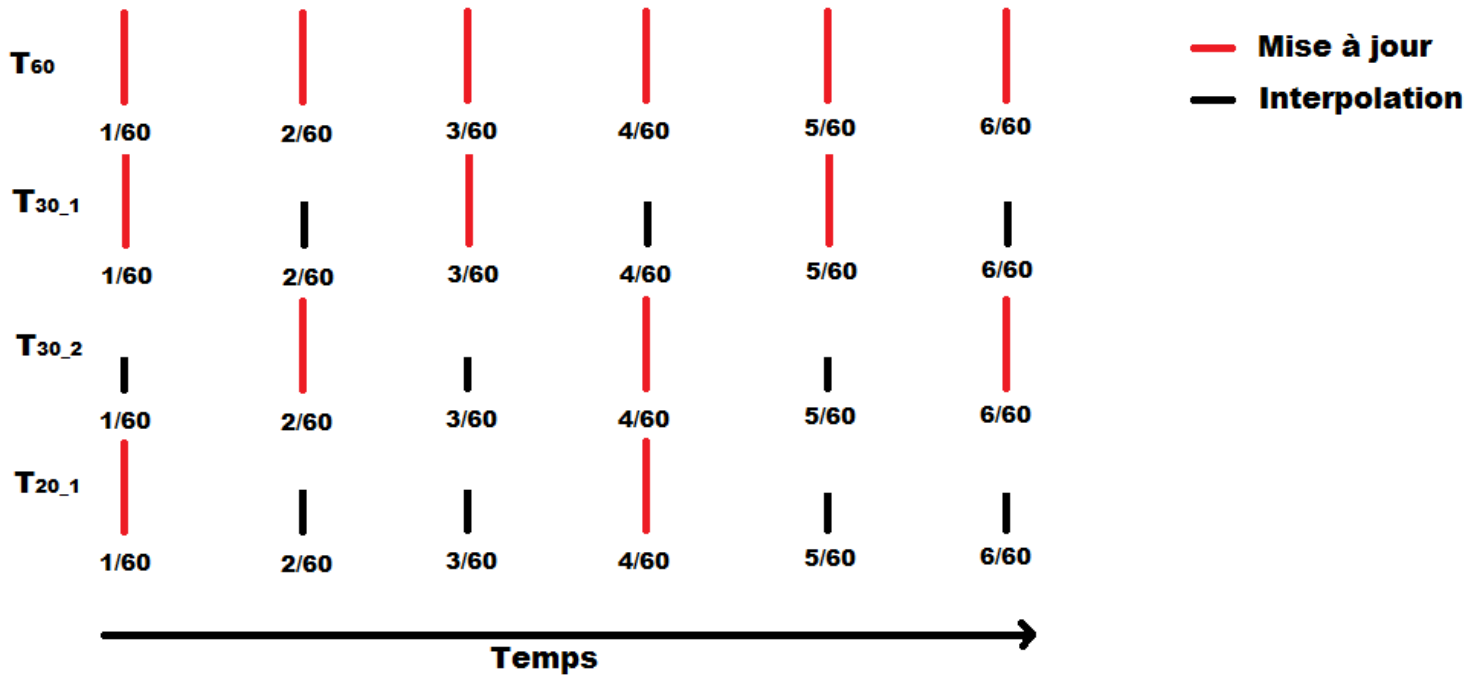
Timers

- La technique basée sur un système de **timers multi-fréquences** :
  - fréquence cible de mise à jour des acteurs de la scène fixée (eg 60 Hz).
  - ensemble de timers diviseurs de cette fréquence cible : T60, T30_1, T30_2, T20_1, T20_2, T20_3, T15_1, T15_2, T15_3, T15_4, etc…

  → acteurs associés par le code du jeu (de manière manuelle et/ou automatique) aux groupes de timers désirés (charge CPU, priorité, etc)

- A chaque boucle de rafraîchissement, mise à jour des acteurs se base sur la fréquence de référence (60Hz, soit dT = 1/60).
  - temps cumulé entre 2 mises à jour est **multiple de la fréquence du timer associé à l'acteur** : mise à jour complète de l'acteur
  - sinon, la position/orientation de l'acteur est **interpolée** (position/orientation).

Timers



Un biais de ce système : affichage à l'écran 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

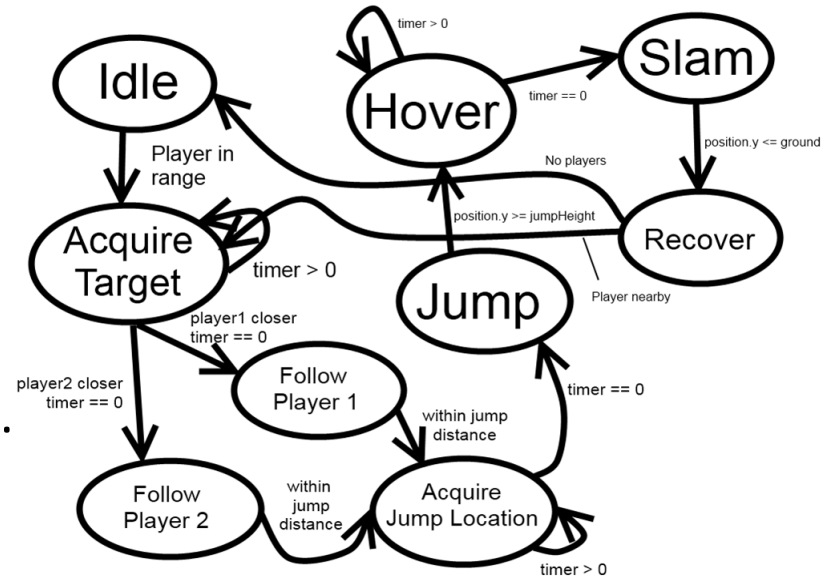
Les modèles comportementaux les plus courants sont les **graphes d'états**.

Le comportement d'une entité est modélisé par un
- **ensemble de nœuds** : états mutuellement exclusifs de cette entité,
- 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 !

Intérêt de ce type de structure :
- facile à implémenter,
- monopolise en général peu de ressources.



Exemple de machine à états

IA et comportements

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

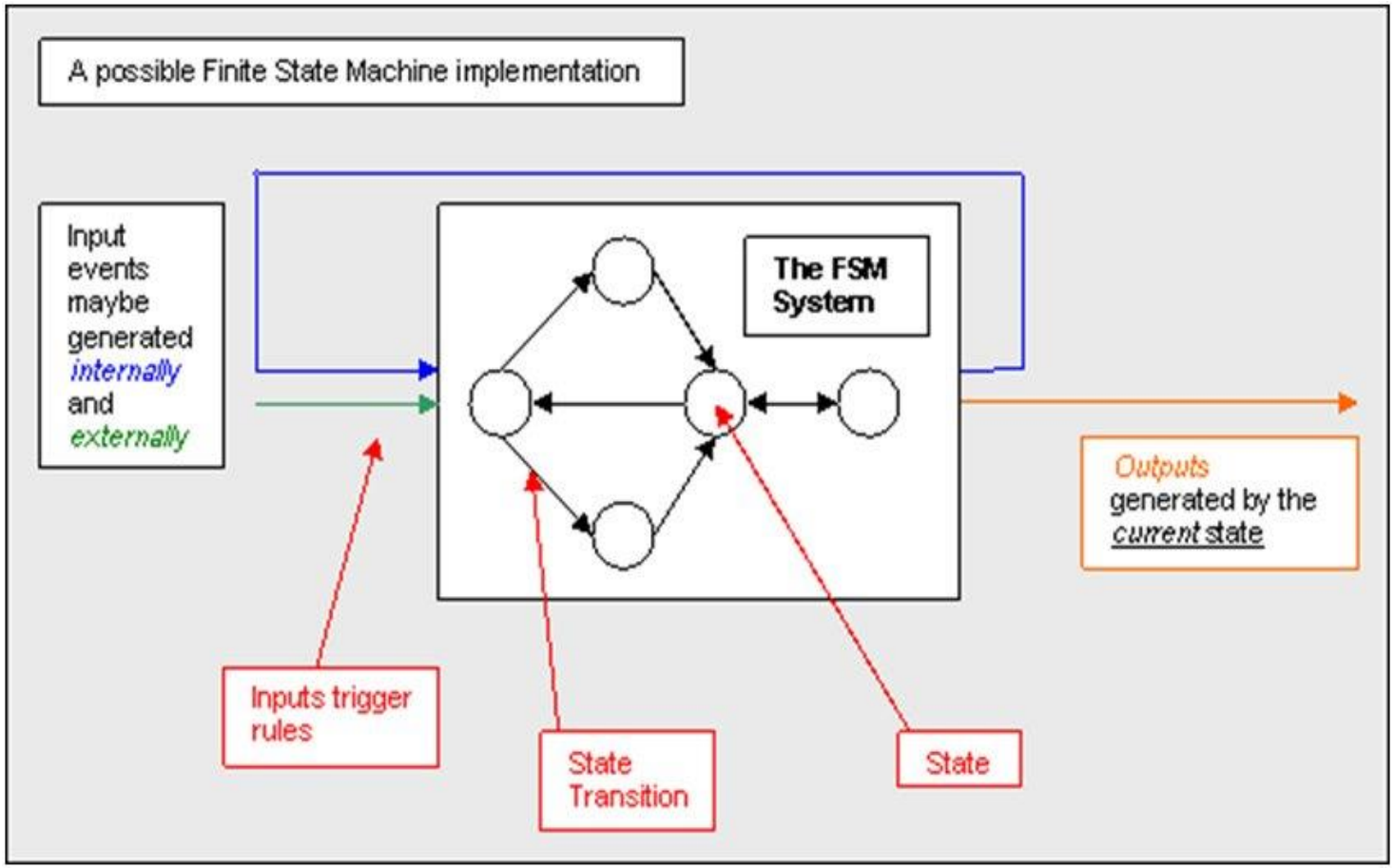


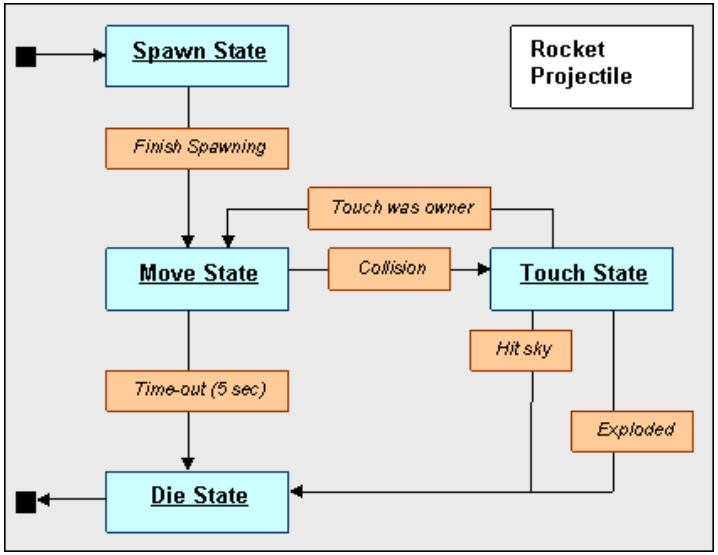
Schéma global d'une machine à états

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 :

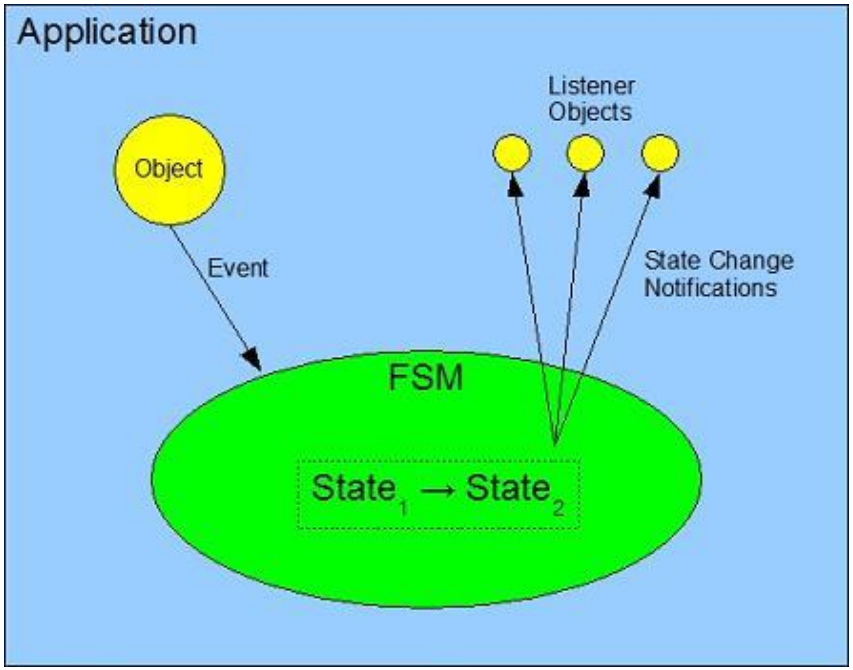
- Le graphe d'états définit le comportement au niveau d'une entité
- Un gestionnaire de décisions définit le comportement haut-niveau (la volonté, en quelque sorte) de l'ensemble des entités



IA et comportements

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

- Positions 3D précises dans la scène :
  - Points de démarrage
  - Positionnement d'entités (bonus, ennemis, interrupteur, émetteur de particules, émetteur sonore...)
  - Nœuds des chemins

- Très utile de pouvoir les exporter :
  - Fait par l’éditeur/exporteur de données
  - Gérant le format de l'export (si éditeur non dédié : 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** → entités du jeu 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 les évènements déclenchés au cours du jeu.

- La grande majorité des évènements déclenchée par le **franchissement d'une zone de la scène** → objets "déclencheurs"

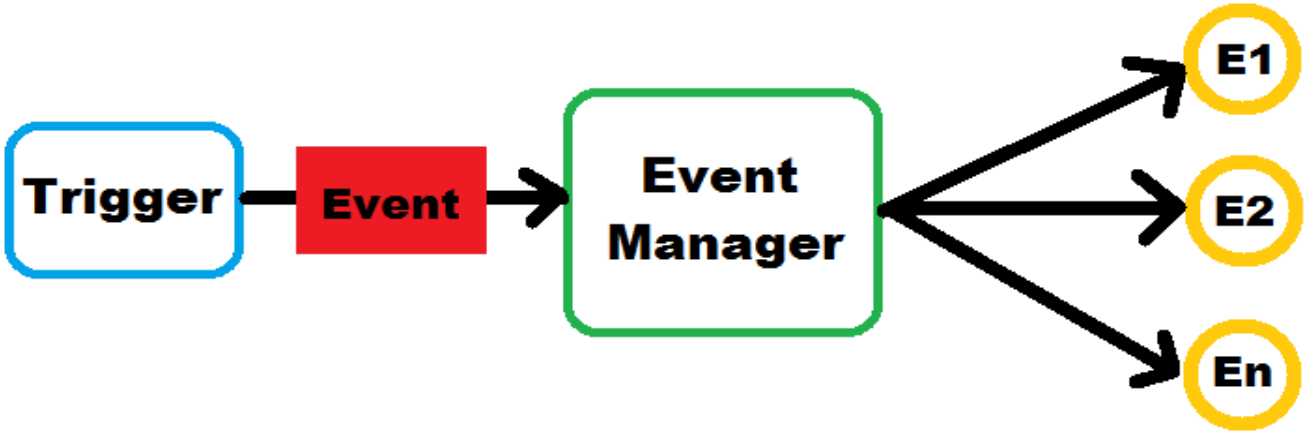
- Propriétés d'objets "déclencheurs" :
  - Topologie : plan, sphère, boîte englobante, …
  - Déclenchement du/des événements associés sur pénétration/occupation/sortie de la zone
  - Déclenchement unique/récurrent
  - Déclenchement associé à un type d'entités (PNJ, joueur, ennemis, alliés, …)

Gestion des événements

Un gestionnaire d'évènements global :

- collecte l'ensemble des évènements déclenchés,
- les mettre à disposition de l'ensemble des entités.
- les entités du jeu peuvent s'enregistrer auprès du gestionnaire d'évènements → notification du déclenchement des évènements les intéressant (**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, permettant 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.

- Les polygones de la scène forment souvent un ensemble difficilement utilisable par les routines de gestion de collisions :
  - 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, ...)
  - Les polygones de la scène ne forment pas obligatoirement un support solide (ex: neige poudreuse sur le sol, étendue d'eau, ...)
  - Accélération des calculs : représenter certaines entités physiques par un modèle ponctuel (collision : représentation polygonale de l‘acteur interpénètrerait les polygones de la scène)

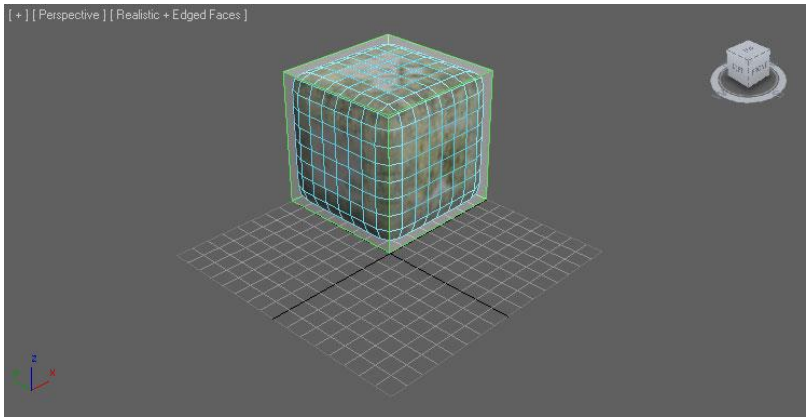
Interactions physiques / topologiques

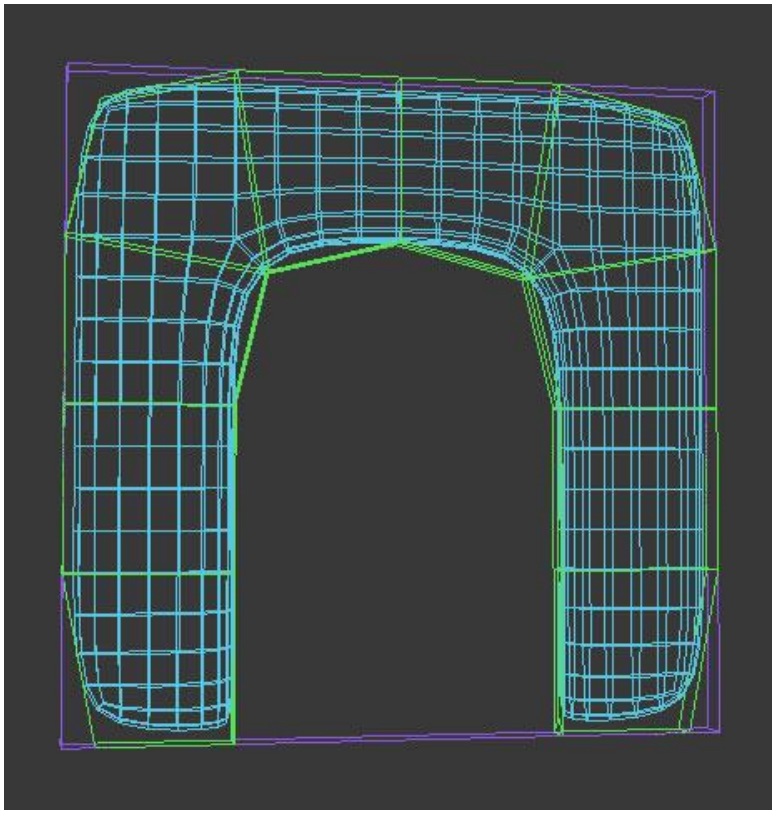
- Pour pallier à ces problèmes, on utilise une scène simplifiée, invisible, adaptée aux contraintes du moteur physique

  → la **carte des collisions** (collision map).

- Dans le même esprit, un maillage de collision (**collision mesh**) est associé aux entités de la scène pour simplifier le traitement des collisions et de la physique

Boites de collision





Matériaux et comportements

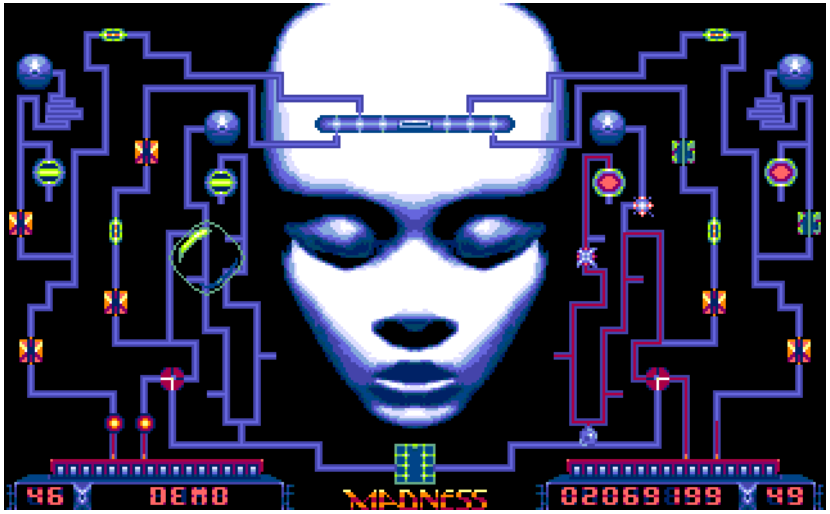
- Il est également nécessaire de pouvoir **réagir aux types de matériaux** de la scène:
  - Bruit des pas, émission de particules, impacts, …
  - Effets sur les acteurs (gain/perte d'énergie, ...)
  - Modification des paramètres physiques du contact (frottement, inertie, ...)
- Identifiant global permettant 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 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.
- La liste exhaustive des matériaux peut au choix être fixée par le moteur, soit par le jeu lui-même

Musique interactive

Capacité à **jouer la musique en relation avec une situation de jeu**. La difficulté : enchaînement correct des séquences musicales.

Propriétés souhaitées :

- Lecture contextuelle de séquences sonores constituant 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 définissant les points de transition possibles (à définir avec le musicien)



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

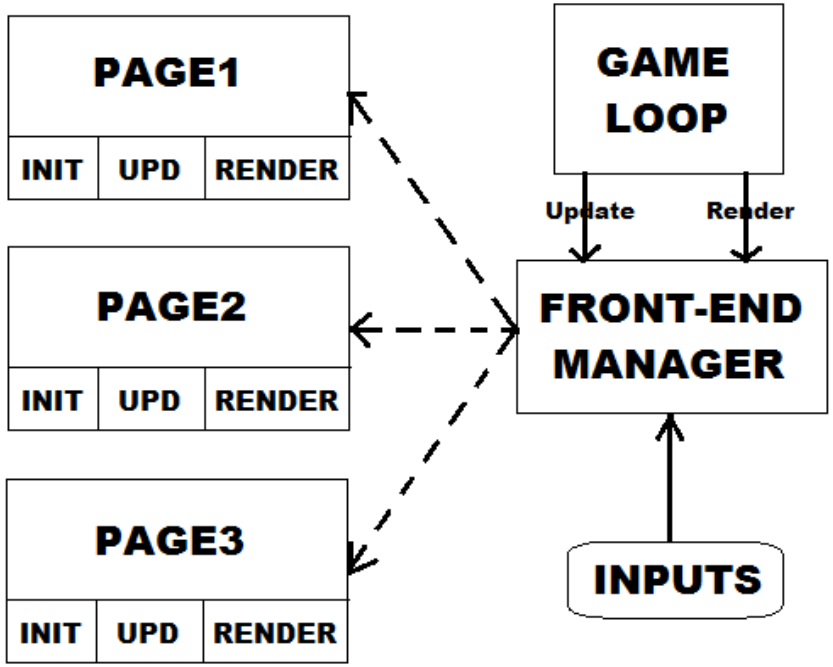
*Autre référence moderne en musiques interactives: **Journey (PS3)***

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, …)

Garder une architecture modulaire :

- 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

	Avantages	Inconvénients
Programmation	• Le langage natif est utilisé, il n'y a donc pas d'overhead dû à la machine virtuelle • Pas de perte de temps pour l'écriture d'un langage de script et de la machine virtuelle associée • Accès direct à l'intégralité des objets du moteur	• Les gameplay designers doivent savoir progammer dans le langage (ou travailler en binôme avec un programmeur) • Le jeu nécessite d'être recompilé à chaque modification majeure (possibilité de limiter les changements par le biais de fichiers de configuration externes) • Effets de bord possibles
Scripting	• Le langage est en général bien simplifié par rapport à un langage de programmation traditionnel • L'exécution du script est très cloisonnée • Le gameplay peut être modifié à la volée, sans recompiler voire recharger le jeu	• Développement long de la machine virtuelle • Overhead à l'exécution

Gameplay

- La tendance penche tout de même en faveur de l'utilisation de scripts.

- La programmation directe :
  - privilégiée dans le cas de micro-projets,
  - moyens et le temps imparti sont réduits (sauf si technologie déjà existante).

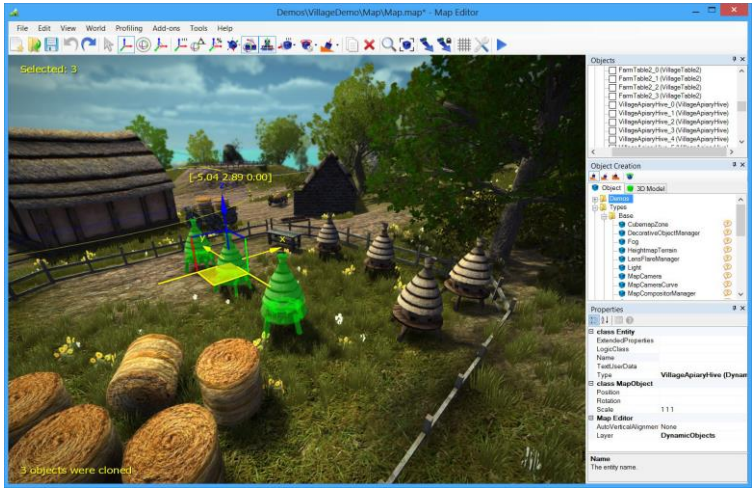
- Fournir (si possible) une paramétrisation externe des constantes de gameplay (ex: paramètres physiques, timings, etc...) :
  - par le biais de fichiers de configuration texte.
  - les gameplay designers peuvent ajuster le gameplay sans intervention d'un développeur.

Qles principaux moteurs de jeux

- Unity 3D
- Ogre3D
- XNA
- Unreal engine
- Cry engine
- Source engine
- Blender Game Engine (Armory Engine)

- Ogre 3D
  - OO interface en C++
  - Un Framework extensible
  - Un moteur haute performance
  - Multi plateforme
  - D3D et OpenGL

  - Contient:
    - Un gestionnaire de scène
    - Un gestionnaire de ressources
    - Un gestionnaire d'animation
    - Un gestionnaire de rendu
    - Des extensions



- XNA
  - Moteur de jeu en C#
  - Support de DirectX
  - Windows, Xbox

- Contient:
  - Les classes de bases pour un jeu
  - Un gestionnaire audio
  - Des fonction graphiques 2D 3D
  - La gestion de devices (manette xbox, clavier)
  - Gestionnaire de ressources
  - Gestionnaire de scène

Unreal Engine

- Unreal engine 3/4

  Open source, PC/Macos

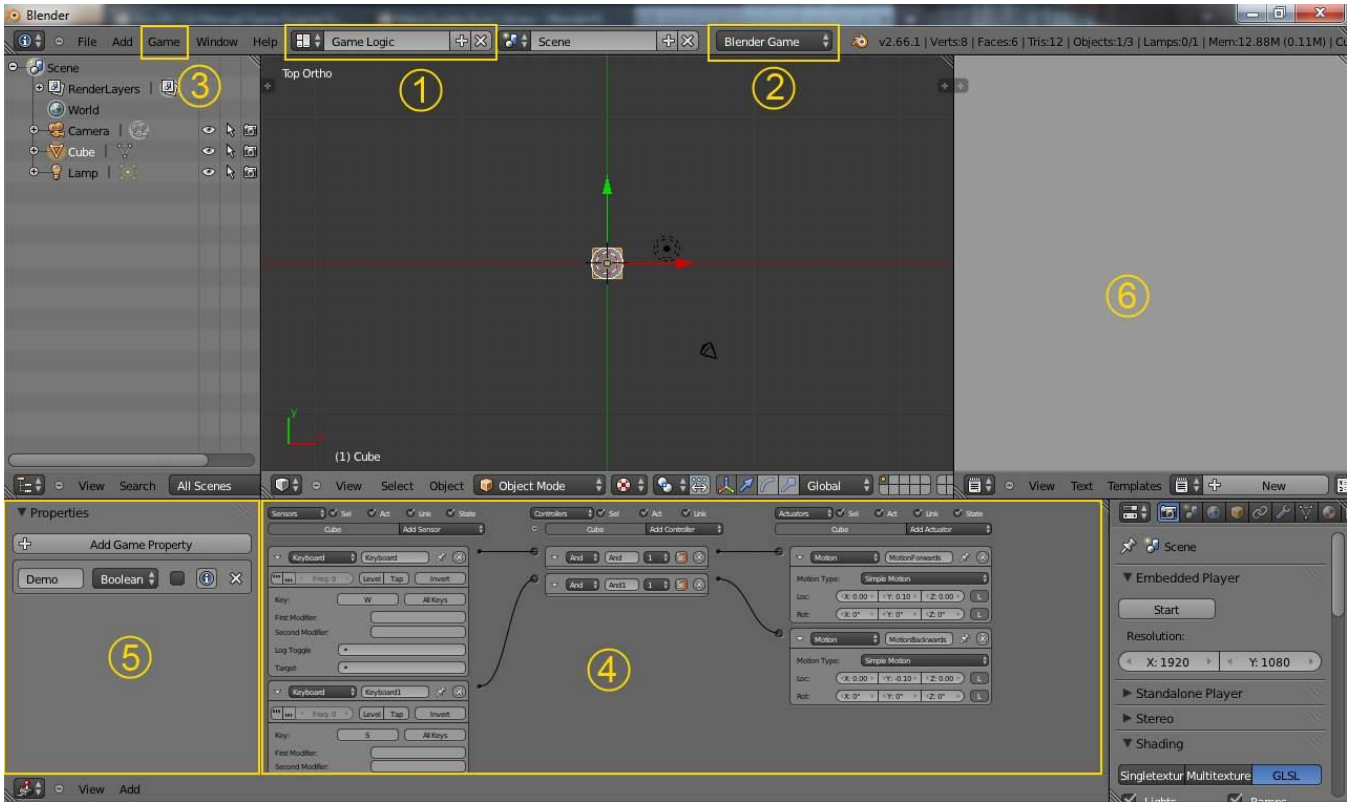
- Des outils :
  - ✓ UnrealEd, l'éditeur de niveaux;
  - ✓ Unreal Kismet, éditeur de scripts (en Flowgraph) ;
  - ✓ Unreal PhAT, éditeur pour la physique dans le jeu (collisions, ragdolls etc) ;
  - ✓ Unreal Matinee, éditeur de cinématiques ;
  - ✓ Unreal Swarm, pour la distribution de calculs ;
  - ✓ L'éditeur SpeedTree, pour créer arbres, feuilles, herbes etc ;
  - ✓ FaceFX, pour l'animation des visages.

- Cry engine/Cinebox (Crytek)
  - What You See Is What You Play Sandbox editor
  - Cinebox for special effects / cinematics
  - Vehicles and physics, scripting, advanced lighting (including real time, moving shadows)
  - Character inverse kinematics and animation blending, Deferred Lighting, Normal Maps & Parallax Occlusion Maps
  - Dynamic music and Advanced Modular AI System.

- Source 1 and 2 engine (Valve)
  - Counter-strike, Half-life, Dota 2
  - PC/Linux/Macos

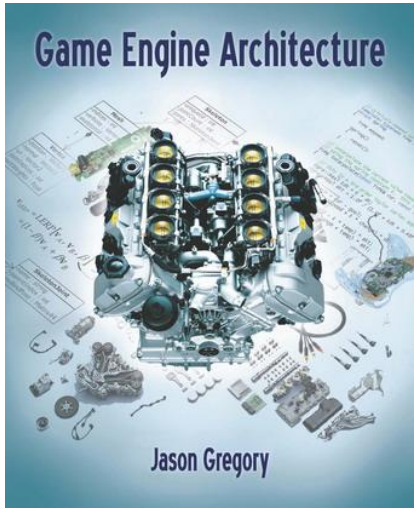
- Blender Game Engine (open source)
  - Architecture de capteurs, contrôleurs et actuateurs
  - Programmables en python

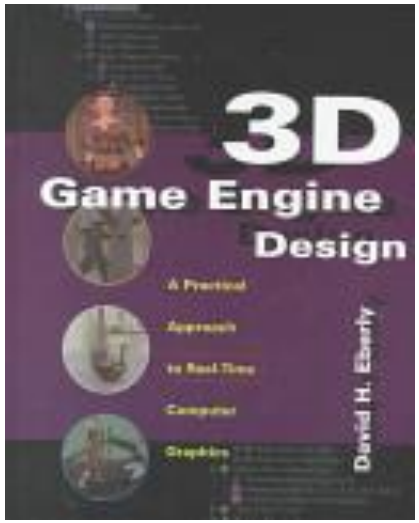
  Supprimé de la dernière version de blender (alternative armory 3D)



- Nombreux outils dans le Blender Game Engine
  - Recast - a state of the art navigation mesh construction tool set for games.
  - Detour - a path-finding and spatial reasoning toolkit.
  - Bullet - a physics engine featuring 3D collision detection, soft body dynamics, and rigid body dynamics
  - Audaspace - a sound library for control of audio. Uses OpenAL or SDL

Livres utiles pour compléter ce cours





Voir aussi les conférences GDC (game developers conference), SIGGRAPH, Eurographics, Motion in Games

Cours en ligne : 3D Game Engine Development

- https://sonarlearning.co.uk/coursepage.php?topic=game&course=ext-bb-3d-ged
- https://www.3dgep.com/courses/