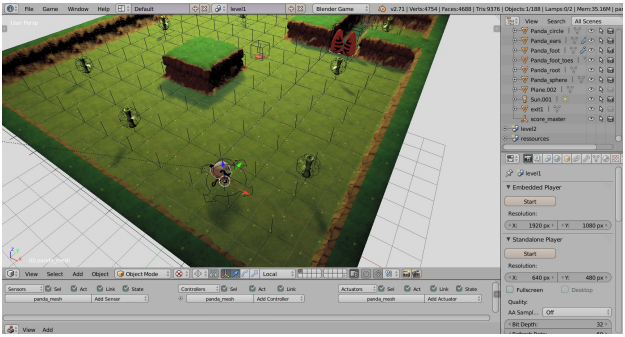
# BLENDER POUR LE JEU VIDEO

**Published :** 2017-06-21

# INTRODUCTION

Le jeu est une activité humaine universellement partagée, et le jeu vidéo a changé la donne depuis l'arrivée de la 3D dans l'expression graphique et scénaristique. Depuis l'avènement du Web et des tablettes, il connait une expansion d'autant plus grande. Les types de jeux produits sont très variés, et leur quantité ne cesse de s'accroitre exponentiellement.

Les moteurs de jeu permettant de créer soi-même un jeu vidéo existent depuis longtemps, mais la nécessité d'avoir recours à la programmation est une contrainte majeure pour tous ceux et celles qui souhaitent créer et cela reste une étape technique difficile à franchir. Pour d'autres, ce sera la partie graphique ou généralement créative qui sera un blocage. Bref, le jeu vidéo étant un produit culturel complexe, demandant diverses compétences, il fait souvent appel à des équipes plus ou moins nombreuses, sans pour autant empêcher des individus touche-à-tout de tenter leur chance.



Depuis longtemps, le monde du logiciel libre dispose avec **Blender** d'un formidable outil de création de jeux. Avant les années 2000, il était déjà conçu comme un outil complet de création 3D, incluant l'interactivité, ce qui permettait d'emblée de l'utiliser pour la création de jeux. Cette utilisation n'a pas toujours été mise en avant, mais un regain d'intérêt récent pour le développement de jeux révèle ce potentiel. Néanmoins, le logiciel ne remplace en rien les compétences des collaborateurs qui devront le développer :

- artistes numériques variés (graphismes, sons, 3D) ;
- scénaristes ;
- programmeurs.

De plus, toute autre(s) compétence(s) peut(vent) être utile(s) selon la taille du jeu et ses ambitions.

Le jeu, en tant que mélange d'histoire, d'interaction, d'univers, a besoin de tous ces éléments pour susciter l'intérêt des joueurs. L'un des avantages de **Blender** est de regrouper en un seul logiciel les éléments de création importants du jeu : le visuel, l'interaction avec les objets et les interfaces, et la programmation. Les différentes parties du travail sont intégrées et augmentent la fluidité du travail d'équipe.

## OBJECTIF DE CE LIVRE

Ce livre a pour ambition de fournir au lecteur un aperçu des possibilités de **Blender** dans la création de jeux vidéo et d'en montrer de façon progressive les principaux tenants. Il abordera des besoins récurrents dans la création de jeux et montrera comment procéder avec le logiciel pour y répondre. Il espère combler un manque de documentation francophone sur le travail interactif dans **Blender**, de manière à aider au mieux les amateurs et professionnels à tirer parti de sa puissance.

Ce livre n'a pas pour objectif de vous faire produire un jeu de A à Z. Il n'a pas non plus l'ambition de vous dire comment réaliser un bon jeu. Dans ce cas, reportez-vous plutôt à un livre traitant du *game design*. Enfin, il n'a pas la prétention de fournir des recettes qui seront applicables dans tous les contextes. Chaque jeu aura une logique propre et le contenu de cet ouvrage devra être adapté aux différents projets voire complété par d'autres ressources et recherches qui accompagnent nécessairement l'analyse de faisabilité d'un jeu.

D'une certaine façon, ce livre a une ambition plus grande encore : vous rendre autonome dans votre capacité à utiliser les outils interactifs de **Blender**, quel qu'en soit l'usage final. Si nous parlons ici de jeux vidéo, il sera utile dans de nombreux autres domaines comme la visualisation architecturale interactive, la robotique, le contrôle à distance d'objet (comme en médecine ou en exploration spatiale), ou encore dans la création artistique interactive et plus récemment dans la réalité augmentée.

## PRÉ-REQUIS

Le sujet de ce livre étant déjà très vaste, il nous a semblé important de nous focaliser ici sur les points clés, de manière à ne pas encombrer l'esprit de détails moins importants. Nous essayons d'expliquer le plus clairement possible les fonctionnalités interactives. Nous partons aussi du principe que notre lecteur dispose déjà d'un certain nombre de prérequis :

- Être correctement équipé : disposer d'un clavier et d'une souris trois boutons (la molette faisant office de bouton central), un pavé numérique pour changer facilement les vues, voire d'un écran suffisamment grand pour travailler confortablement ;
- Savoir modéliser un minimum avec **Blender**, pour créer des personnages, objets et décors, texturer, gérer des contraintes de base comme attribuer un parent à un objet, éventuellement créer un *rig* et animer. Si ce n'est pas le cas n'hésitez pas à vous reporter aux autres manuels Flossmanuals sur **Blender** ou d'autres documentations qui sont assez nombreuses sur ces sujets ;
- Connaitre les bases de la programmation. Le langage Python est utilisé dans **Blender** et, au vu des besoins éventuels en programmation dans le développement du jeu, il n'est pas inutile d'y avoir recours. Ce livre ne présente aucune base en langage Python. Nous considérons que celles-ci sont connues, même partiellement. Si ce n'est pas le cas, nous vous invitons à lire au préalable ou à garder sous les yeux une documentation Python comme celle de Flossmanuals et d'avoir à l'œil la documentation officiel de Python et l'API Python de Blender.

*Pour ce livre, nous avons changé le thème de l'interface de Blender pour utiliser celui de la 2.4, et changé la couleur de fond de la vue 3D pour avoir des captures d'écran plus contrastées, et plus lisibles.*

Les auteurs de ce livre ont utilisé la version 2.71 de Blender. Ce logiciel évoluant à un rythme soutenu, nous vous incitons à vous inscrire sur cette plateforme pour apporter toute correction concernant des changements qui seraient intervenus ultérieurement.

# INTRODUCTION

**1.** POURQUOI UTILISER LE BLENDER GAME ENGINE ?
**2.** SPÉCIFICITÉS DU GAME ENGINE
**3.** NOTIONS DE GAME DESIGN
**4.** BIEN PRÉPARER SON PROJET

# 1. POURQUOI UTILISER LE BLENDER *GAME ENGINE* ?

Est-ce que le **Blender *Game Engine* (BGE)** est adapté pour créer votre projet ? Après tout, il y a beaucoup d'autres moteurs de jeux vidéo, et votre temps est probablement limité pour pouvoir tous les tester.

## QUE PEUT-ON FAIRE AVEC LE BGE ?

Le moteur temps réel de Blender permet de réaliser beaucoup de choses différentes.

### Du jeu vidéo

Comme son nom l'indique, le *Game Engine* nous permet de créer des jeux vidéo et de le faire assez facilement. En quelques minutes, il est possible de créer un jeu basique, de le lancer et d'avoir une première interaction avec le jeu comme celle de déplacer un objet. En poussant ses capacités, il est possible de créer des jeux ou des simulations très complexes, voir photoréalistes, avec une logique avancée et des interactions sophistiquées.

*A priori*, tous les types de jeux sont réalisables avec le BGE. Néanmoins, vous ne pourrez pas produire de MMORPG* de nouvelle génération, mais d'un autre côté, vous ne disposez probablement pas d'une équipe de 200 personnes ! Si vous n'avez pas les yeux plus gros que le ventre, votre idée pourra certainement être réalisée.

Même si vous êtes un professionnel du jeu vidéo ou un développeur aguerri, le BGE pourra vous intéresser pour réaliser un prototype de jeu et le tester rapidement. Vous pourrez ensuite éventuellement passer à un autre moteur de jeu plus adapté pour votre concept. Ainsi, vous pourriez ne passer que quelques jours à le concevoir avec le BGE, plutôt qu'un mois à coder avant de tester votre idée.

### Des présentations, des visites virtuelles et de la visualisation architecturale

Le BGE est souvent utilisé tout simplement pour permettre à un utilisateur (joueur) de se déplacer dans un univers virtuel, en 3D ou en 2D. Si vous êtes architecte, vos clients seront ravis de visiter leur future maison.

Le BGE permet également de présenter un objet ou un produit sous plusieurs angles et de le faire réagir aux clics des utilisateurs.

### Des simulations et des prototypes scientifiques

Il est possible de faire des simulations 3D, avec de la physique rigide, comme une chute d'objets ou une simulation de voiture.

Si vous exercez un métier dans le domaine de la réalité virtuelle et que vous cherchez à tester vos concepts, le BGE est peut-être une bonne solution. Des médecins ont utilisé Blender pour simuler des opérations chirurgicales avec retour de force.

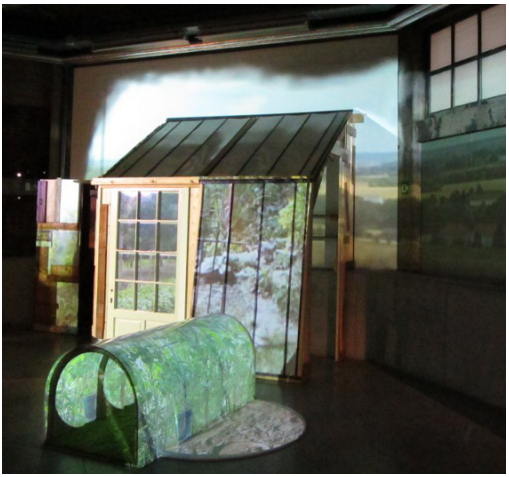
### En pédagogie pour des animations interactives ou des *Serious Game*

Le monde de l'éducation et de la formation cherche toujours de nouvelles solutions pour la diffusion de contenu et surtout pour rendre ses contenus plus attrayants. Les logiciels éducatifs pour enfants mais aussi les MOOC* ou plate-formes de formation à distance, nécessitent la réalisation d'animations interactives qui permettent aux apprenants de tester. Blender peut alors se prêter à cette utilisation autant que d'autres. On peut également imaginer réaliser des *Serious Game* avec le BGE, c'est à dire des jeux à but pédagogique qui reproduisent une situation réelle qui serait dangereuse (apprendre à ne pas jouer avec le gaz de la cuisine par exemple).

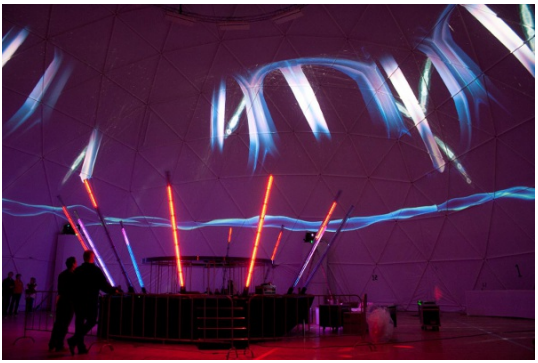
### Des installations artistiques interactives et en temps réel

Le BGE est aussi utilisé pour des installations artistiques, intégrant de l'interaction avec le public. La possibilité d'intégrer tant du contenu 3D, 2D, que du son et des capteurs (grâce au développement en Python) permet de développer des projets complexes et variés.

L'intégration de la texture vidéo permet également de concevoir des environnements appliquant la vidéo sur des surfaces non planes. Ceci permettant ce qu'on appelle maintenant le *mapping*, c'est-à-dire le fait de texturer des objets réels par des projections vidéo s'adaptant à la géométrie de l'espace réel, redressant des perspectives et jouant sur des illusions d'optique.



Perma-Cabane est une installation vidéo interactive dans un musée, avec *mapping* dynamique contrôlé par Kinect (http://www.ogeem.be/doku/doku.php?id=fr:perma_cabane). L'image est produite avec seulement 2 projecteurs vidéo installés de part et d'autre de l'espace.



Cosmic Sensations proposait une projection en dôme d'un système permettant de visualiser les rayons cosmiques de manière créative (http://download.blender.org/documentation/bc2010/cosmicsensation_slides.pdf ). L'image est projetée à une taille de 1000x4000 pixels, grâce à plusieurs projecteurs et à un *mapping* en dôme.

Blender est également extensible grâce à Python, et peut s'interfacer facilement avec d'autres logiciels, d'image ou de son par exemple. Une méthode courante pour interfacer avec Blender est l'utilisation du protocole OSC (Open Sound Control), sorte de successeur du MIDI, très courant dans les logiciels de travail du son. Ceci permet par exemple de créer un système sonore indépendant du BGE mais synchronisé avec lui. Ce protocole est également souvent utilisé pour intégrer des contrôleurs externes, et compléter la palette d'interface physique disponible (par exemple Kinect, Wiimote, détection caméra, senseurs avec Arduino, *etc.*). Dernièrement on a pu voir aussi qu'il était simple d'utiliser le BGE avec les casques de réalité virtuelle de type Oculus Rift.

## QUELQUES EXEMPLES DE JEUX UTILISANT LE BGE ?

Pas convaincu(e) ? Voici des exemples de jeux réalisés avec le Blender *Game Engine*.

### Yo Frankie!

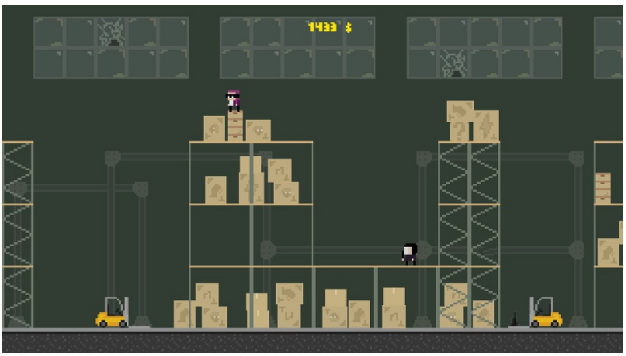
Il s'agit d'un jeu vidéo gratuit et libre édité par la Blender Foundation en 2008 dont le personnage principal, Frank, est issu du court-métrage Big Buck Bunny.



Url du projet : http://www.yofrankie.org

### Running Guys

Un jeu de **course/plateforme**, gagnant d'un concours de création de jeux vidéo en juin 2014. Il est possible de jouer à deux sur la même machine avec le clavier ou des manettes et de jouer en **multi-joueur** (les serveurs peuvent gérer plusieurs parties en parallèles). Le code et toutes les ressources (images et musiques) sont entièrement **libres**. Il a été réalisé par seulement deux personnes en deux semaines. Depuis il continue d'être amélioré. Des fonctionnalités et du contenu sont ajoutés tous les mois.


Accessible sur http://rg.osxia.org

### Dead Cyborg

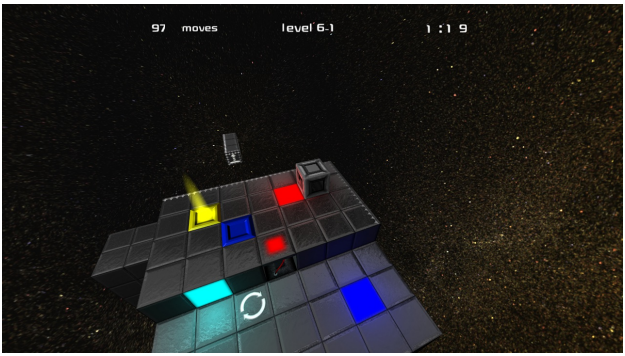
Un jeu complet d'aventure SF, une histoire à propos du sens de la vie et de la mort. Il est disponible sur le Steam Greenlight.

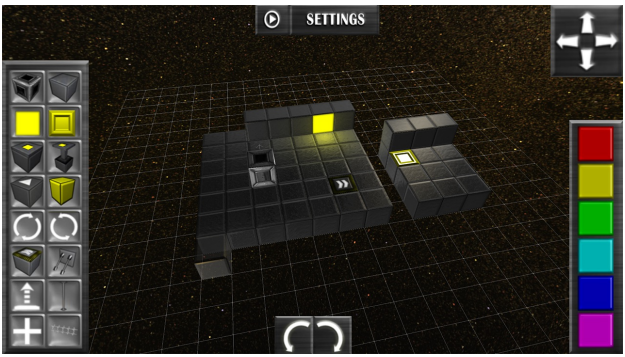

Accessible sur http://deadcyborg.com

**ColorCube**

ColorCube est un jeu de puzzle/réflexion basé sur le roulement d'un cube et le fait d'attraper et de déposer une face colorée. Le jeu a été entièrement conçu à l'aide de Blender, que ce soit pour la modélisation ou la logique interne, utilisant le moins possible de scripts Python. La toute première version du jeu n'utilisait d'ailleurs que des briques logiques, afin de participer à un concours basé sur ce principe. Après l'enthousiasme encourageant des joueurs testeurs, une version améliorée incluant un éditeur de niveau a été proposé à petit prix, dans l'optique de faire un test de jeu commercial utilisant le BGE. Le jeu a connu un succès raisonnable, principalement au niveau de la communauté Blender.

Développeur : ColorCubeStudio
Genre : puzzle/réflexion 3D
Licence : propriétaire / Démo libre disponible
Prix : 4,50€




## POURQUOI PAS D'AUTRES MOTEURS DE JEU ?

Il existe de nombreux moteurs de jeu. Beaucoup d'entre eux sont d'ailleurs libres dont l'un des plus prometteur pourrait être Godot dont le statut évolue (http://okamstudio.com/tech/).

En attendant Godot, ou qu'un autre moteur indépendant se révèle, il est important d'avoir de nombreux critères pour en choisir un. Tout d'abord, le *Game Engine* est intégré à Blender : si vous connaissez déjà Blender, il sera plus facile de commencer les jeux vidéos avec votre outil préféré ! Si vous ne connaissez pas Blender, avoir un moteur de jeu directement intégré à votre outil de création 3D est clairement un plus : pas besoin de changer d'outil pour tester son jeu, un simple appui sur P permet de lancer le jeu. Cela représente aussi un avantage non négligeable sur les jeux créés en langages compilés.

De plus, vous n'avez pas besoin de savoir coder pour commencer à utiliser le BGE : les briques logiques (`Logic Bricks`) permettent d'aller assez loin dans la création et l'interactivité du jeu.

À l'inverse, si vous savez coder en Python, vous pourrez accéder à absolument toutes les possibilités du BGE avec votre code, comme c'est le cas dans Blender en général.

Enfin, c'est un moteur de jeu libre, sous licence GPL* : vous avez le droit de l'utiliser sans restrictions, de le partager autour de vous, de l'étudier, et même de le modifier pour l'améliorer !

## UN OUTIL, ÇA OBLIGE À FAIRE DES CHOIX

Bien sûr, comme tous les outils, le BGE ne sait pas absolument tout faire. Il possède ses propres spécificités et limites, dont voici les principales. Un outil ça s'utilise, ça se contourne, et cela a parfois besoin d'adaptation pour fonctionner comme désiré. Libre à vous de contribuer à son amélioration si vous avez des demandes spécifiques !

### Jeux en réseau

Il est possible de faire des jeux en réseau avec le BGE mais il n'y a pas de fonction toute prête pour le faire rapidement : il faudra utiliser le langage Python et une librairie dédiée au réseau, donc avoir les compétences nécessaires en programmation. En revanche, avoir plusieurs joueurs sur le même ordinateur est plus accessible, nous l'aborderons dans la section **Développer l'ergonomie**.

### Performances

Il y a plus rapide que le *Game Engine* de Blender, et, malgré des progrès dans ce sens, vous pourrez atteindre des limites de performance plus tôt qu'avec d'autres moteurs. Cela est principalement dû à l'utilisation du langage Python, qui apporte beaucoup d'avantages et de simplicité, mais qui n'est pas conçu pour le cas spécifique des performances dans un environnement 3D.

Cependant, tout jeu doit toujours être pensé en fonction de ces contraintes de calcul et cela a un impact systématique, y compris dès le départ sur la modélisation. Optimiser son jeu dès le début et prendre de bonnes habitudes ici sera utile dans tous les moteurs ! Nous traiterons de ces habitudes à prendre dans tout le livre, et nous aborderons plus particulièrement les techniques dédiées à l'amélioration des performances dans la section **Optimisations**.

### Le *Game Engine* ne possède pas toutes les fonctions de Blender

Le *Game Engine* est une partie à part de Blender. Le calcul en temps réel réclame d'autres solutions techniques que celles utilisées pour faire du rendu 3D. De ce fait, tout ce que vous savez faire dans Blender ne sera pas forcément possible dans le BGE, à moins de passer par une étape de conversion aux solutions dédiées au « temps réel ». Si votre ambition est de faire des jeux, vous concevrez vos projets directement dans cette optique et bientôt ces contraintes n'apparaitront plus.

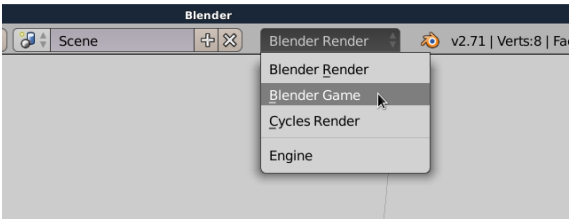
# 2. SPÉCIFICITÉS DU *GAME* ENGINE

Blender est une suite graphique 3D complète. Ses usages vont de l'impression 3D aux films à effets spéciaux ou d'animation. Une utilisation pour le *Game Engine* implique donc des spécificités.

## ADAPTER SON INTERFACE

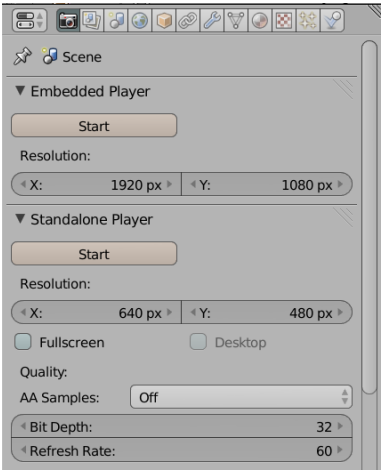
L'interface par défaut est adaptée au moteur de rendu *Blender Internal* destiné à du film d'animation.

Il faut changer `Blender Render` dans le menu de la barre `Info` et opter pour le choix `Blender Game` afin de travailler avec l'interface du moteur de jeu.




Un changement immédiatement visible est la modification de l'onglet `Render`.

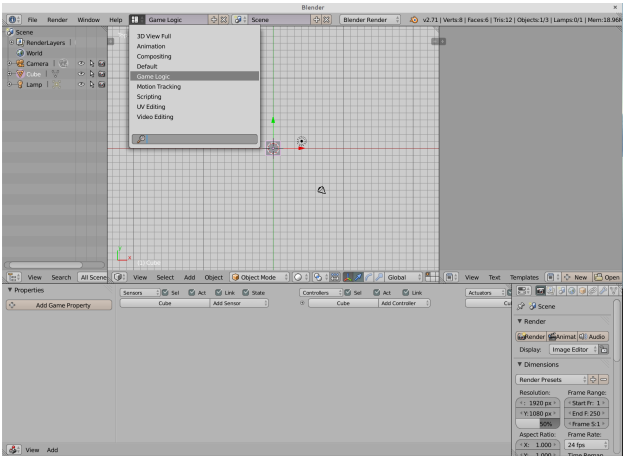
Les panneaux `Render` et `Dimensions` (qui permettent de lancer le rendu d'une image à une certaine dimension) ont été remplacés par des panneaux `Embedded Player` et `Standalone Player` (destinés à définir l'emplacement, la taille de la fenêtre dans laquelle va s'exécuter le jeu).




Les différentes options de ces nouveaux panneaux seront précisées dans le chapitre **Bien préparer son projet**, à la fin de cette section.

## ADAPTER SON AGENCEMENT DE FENÊTRE

Nous utilisons prioritairement certaines fenêtres (ou éditeurs) en fonction du type de travail à réaliser. Bien souvent, un utilisateur confirmé aura défini son propre agencement de fenêtre, et Blender permet facilement de faire apparaitre une fenêtre ou une autre en fonction des besoins. Néanmoins, Blender propose par défaut un agencement appelé ***Game Logic***. C'est celui-ci que nous préfèrerons parce qu'il positionne de manière claire les fenêtres les plus importantes lorsque l'on pratique de la création de jeu vidéo.

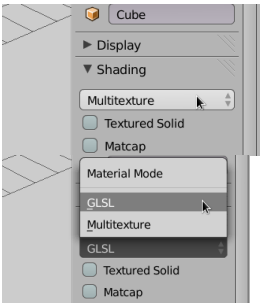



Ces fenêtres sont (en allant de gauche à droite et de haut en bas):

- `Outliner` qui nous sert à avoir une vue en liste de nos éléments de jeux ;
- `3D View` qui servira entre autre à prévisualiser le jeu ;
- `Text Editor` où nous écrirons du code Python ;
- *Logic Editor* où nous assemblerons et éditerons nos briques logiques ;
- `Properties` bien connue des utilisateurs de Blender.

## SPÉCIFICITÉS GRAPHIQUES

Les capacités de l'affichage 3D des ordinateurs ont évolué au cours du temps, ce qui se traduit actuellement par deux méthodes de rendu.
Dans le panneau ***Shading***, le choix `Multitexture`est actif par défaut. Il correspond simplement à la possibilité d'afficher des matériaux utilisant plusieurs textures.

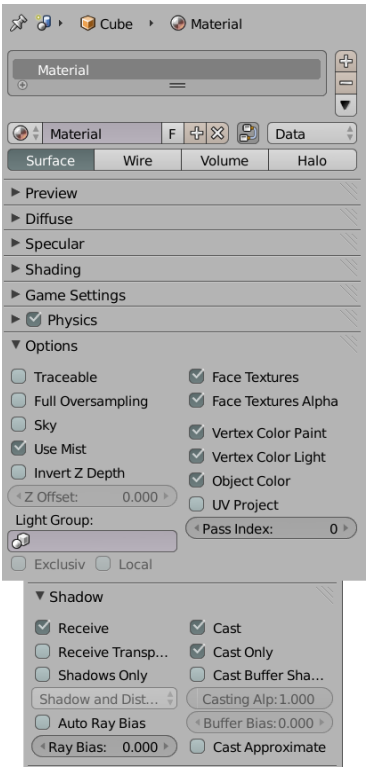


Le choix ***GLSL*** est une option plus intéressante graphiquement car elle permet de nombreux effets avec les éclairages, les matériaux, les effets de texture, et d'autres améliorations graphiques.
La plupart des ordinateurs récents supportent les capacités *GLSL*. Néanmoins, si vous destinez votre jeu à une large diffusion et que la qualité graphique du jeu est mineure (ou par nécessité d'optimisation), vous pourrez opter pour le choix *Multitexture*.

### Les matériaux

L'interface de l'onglet des matériaux dans le *Blender Game* est dérivée de celle du *Blender Render*, avec quelques subtilités. Puisque le moteur de rendu *Cycles* est désormais souvent le premier moteur de rendu appris par un graphiste utilisant *Blender*, un petit récapitulatif s'impose sur l'onglet `Matériaux` et ses différents panneaux étant donné que les matériaux et les textures doivent être gérés différemment.

- Comme dans le `Blender Render`, pour un matériau donné, les panneaux ***Diffuse***, ***Specular***, ***Shading*** permettent de définir les paramètres de base du matériau (couleur, intensité, dureté, auto-illumination, sensibilité à la lumière ambiante, etc.).
- Les panneaux `Mirror` et `Subsurface Scattering` n'existent plus. Ce sont des fonctionnalités non présentes dans le *Game Engine*. Il sera possible de simuler un miroir avec des techniques avancées, mais ce n'est plus une option de matériau.
- Le panneau ***Options*** permet d'activer des fonctions de colorisation supplémentaires. Malheureusement, quelques-unes de ces options n'ont aucun effet dans le *Game Engine* (par exemple: *Traceable*, *Full Oversampling*, *Sky* ou encore *Invert Z Depth*). Par contre d'autres peuvent s'avérer très utiles pour l'optimisation de jeux :
  - L'option ***Object Color*** correspond à une couleur propre à un objet. Deux objets peuvent utiliser le même matériau mais avoir une couleur différente définie par cette option.
  - Les options ***Vertex Color Paint*** et ***Vertex Color Light*** correspondent à une collection de couleurs variant selon les *vertices* d'un maillage. Pour le jeu vidéo, cela sert souvent à colorer, illuminer ou assombrir des parties d'objet, pour améliorer la qualité graphique d'un objet sans alourdir son matériau.
  - L'option ***Face Textures*** permet à la texture brute de se substituer au résultat de tous les réglages du matériau. Cette option était fort utilisée avant l'arrivée du *GLSL* afin de s'assurer de l'affichage des textures dans le *Game Engine*.
- Le panneau ***Shadow*** permet de gérer les ombres à partir des matériaux. Ces options doivent être pensées en relation avec les lumières. Certaines options n'ont aucun effet dans le *Game Engine* comme `Receive Transparent` ou `Shadows Only`.




*Sur cette image, les cases cochées correspondent aux options supportées par le Game Engine.*

### Les particules

Les particules des moteurs de rendu *Blender Render* ou *Cycles* ne sont pas accessibles dans l'interface *Blender Game* car leur physique complexe n'est pas supportée. Par des effets de textures animées, il est possible de générer en temps réel le même genre d'effets créés avec des particules (feu, fumée) ou pourquoi pas en utilisant un générateur automatique d'objets sur lequel nous aurons un chapitre spécifique.

Un *add-on* appelé ***Easy Emit*** permet de gérer la génération de plans texturés, avec une interface semblable à celle du système standard des particules. Voir http://blenderartists.org/forum/showthread.php?241656-easyEmit-*Update*-13-06-2013

## SPÉCIFICITÉS ET LIMITATIONS LIÉES À CERTAINES TECHNIQUES DE MODÉLISATION ET D'ANIMATION

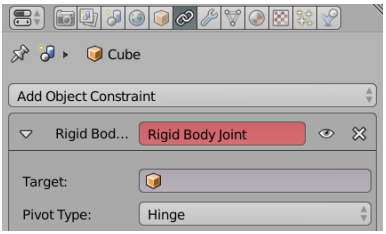
Pour animer des éléments dans Blender, certains ont peut-être l'habitude d'utiliser des fonctionnalités avancées comme les contraintes d'objet (`Object constraintS`) pour conserver ou changer les propriétés ou le maillage de leur modèle, pour s'aider dans l'animation de l'une ou l'autre partie de leur personnage ou tout simplement pour lui faire suivre un objet particulier de la scène. Malheureusement, certaines de ces aides et méthodes ne sont pas disponibles dans le *Game Engine*. Il faudra utiliser des alternatives pour arriver à un résultat identique ou approchant.

Les techniques utilisant les contraintes d'os (`Bone constraints`) sont heureusement bien supportées, mais ne sont pas toutes disponibles.

Et pour finir, la règle générale pour l'utilisation des **modificateurs**(***modifiers***), c'est que seuls les modificateurs s'appliquant sur des **maillages** (***mesh***) et qui ne dépendent pas d'un facteur temps sont supportés. Quoi qu'il en soit, on préfèrera appliquer les effets de tous les modificateurs avant la publication du jeu de manière à alléger les calculs autant que possible.

### Contraintes d'objet

Il est important de savoir que la plupart des contraintes d'objet ne sont pas supportées par le *Game Engine*. L'unique contrainte d'objet supportée est le `Rigid Body Joint`, à l'exception du joint de type *Ball*. La contrainte de type *Rigid Body Joint* ne fonctionne que pour les objets de type ***mesh***, pour autant que l'objet ait un modèle physique (voir le chapitre **Le comportement des objets de notre monde** dans la section **Comportement des objets et personnages**).




Les joints sont créés automatiquement au démarrage du jeu, mais ne peuvent être détruits ou créés au cours du jeu que par une des commandes Python spécialisées.

*L'absence de presque toutes les contraintes dans le BGE peut sembler rédhibitoire mais il est souvent possible de compenser cette perte par diverses techniques :*

- Utiliser une relation de parent entre objets à la place de la contrainte `Copy Location` ;
- Utiliser la brique logique `Edit Object`et son option`Track To` à la place de la contrainte `Track To`.

*En dernier recours, on peut toujours tenter de reproduire l'effet d'une contrainte par du code Python.*

### Contrainte d'os

Pour les armatures, un certain nombre de contraintes d'os (***Bone Constraint***) sont supportées. En voici la liste exhaustive:

- `Track To`
- *Damped Track*
- *Inverse Kinematics*
- *Copy Rotation*
- *Copy Location*
- *Floor*
- *Copy Scale*
- *Locked Track*
- *Stretch To*
- *Clamp To*
- *Transformation*
- *Limit Distance*
- *Copy Transform*

### Modificateurs

Davantage de modificateurs (*modifiers*) sont supportés dans le BGE. En premier lieu, le *modifier* `Armature` qui permet l'animation par armature d'un objet. Pour le reste, les *modifiers* qui n'agissent que sur le *mesh* de l'objet et qui ne dépendent pas du temps sont supportés. Voici une liste de *modifiers* qui dépendent du temps et donc **ne sont pas supportés** :

- *Build*
- *Displace*
- *Wave*
- *Cloth*
- *Fuild Simulation*
- *Explode*
- *Smoke*
- Mesh Cache

En cas d'hésitation, le plus simple est de tester directement dans le BGE si un *modifier* est supporté avec le *Standalone Player*. En général, l'effet du *modifier* est supporté mais pas son animation.

Toutes ces remarques sur les `modifiers` ne valent que pour ceux non appliqués avant le lancement du jeu. L'application d'un modificateur (en appuyant sur le boutton `Apply`) transforme de manière permanente le maillage de l'objet (le `modifier` disparait d'ailleurs de l'interface). Et donc la question du support de ce `modifier` dans le BGE ne se pose plus, puisque le maillage a subi des transformations irréversibles et maintenant visibles dans le *Game Engine*.

Si vous avez l'habitude d'animer directement un *modifier* (en ajoutant des clés d'animation sur ses paramètres) ou indirectement (par le biais de l'animation d'un objet référence ou de celles de *weight groups*), la parade à adopter sera d'appliquer le *modifier* en tant que *ShapeKey* (avec le bouton ***Apply as Shape Key***). Le BGE supporte les animations de *ShapeKeys*.

*L'utilisation de modifiers dans le jeu ne se justifie que pendant la phase de développement. Une fois en production, il est généralement préférable d'appliquer le modifier pour améliorer les performances* ***à l'exception de l'armature****.*

## PAS DE GASPILLAGE DE RESSOURCES

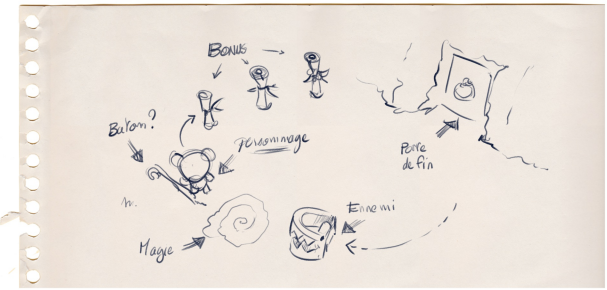
Sans rentrer dans des détails qui seront approfondis plus tard, il faut retenir que l'optimisation est au cœur de la conception d'application en temps réel. Voici quelques points de repères et de bonnes pratiques.

- Un jeu vidéo est considéré comme parfaitement fluide s'il est capable d'afficher enviiron **60 images par seconde**. En dessous de 50, le jeu commence à sembler saccadé mais reste utilisable. En dessous de 25 images par seconde, l'illusion de mouvement ne fonctionne plus, et l'expérience ludique est fortement dégradée.
- Il est important de limiter le nombre de faces de chaque objet. De nombreux facteurs entrent en compte, mais disons qu'il est déraisonnable de dépasser **quelques milliers de triangles pour un objet important**, et quelques centaines pour un objet secondaire.
- Attention, Blender indique le nombre de faces, pas le nombre de triangles. Dans la majorité des cas, il faut donc multiplier par deux le nombre indiqué.
- Pour des raisons techniques liées à la rapidité d'affichage en jeu, les textures devraient toutes respecter une règle simple : être des images **carrées** dont le côté est une **puissance de 2**. C'est à dire 128x128 pixels, ou 256x256 pixels, 512x512 pixels et ainsi de suite. Les formats préconisés sont le **PNG** et le **TGA**.
- Gardez à l'esprit que vos futurs joueurs ne disposent pas forcément d'un ordinateur très puissant ou d'un système très performant. De manière générale, **optimisez, même si ça fonctionne bien sur votre machine**. De plus, testez votre jeu sur la machine la moins puissante que vous pourrez trouver, ou celle que vous aurez définie comme "configuration minimum".

## CONCLUSION

Le fait de produire une création qui sera vue en temps réel, et donc constamment calculée pour le rendu, a un impact sur différents aspects de la création 3D tant du point de vue de la modélisation, de la texture ou de l'animation que du rendu global des éléments du jeu. Cela peut être déroutant pour une personne ayant l'habitude des mode *Blender Render* ou *Cycles*. Mais comme dans tout acte créatif, ses limitations sont non seulement imposées par le médium (le temps réel, les capacités du *Game Engine* et les capacités de l'ordinateur qui le fait tourner), mais elles peuvent aussi être vues comme autant de défis créatifs à relever ou à contourner.

# 3. NOTIONS DE GAME DESIGN




Un jeu est un tout. Un jeu c'est à la fois une histoire, des personnages, des ressources graphiques, une logique, des énigmes. Un jeu c'est aussi une ambiance sonore, un univers et différents moyens pour le joueur d'agir. Le *game design* : c'est l'art de prendre les bonnes décisions concernant votre jeu. Faire du *game design* : c'est prendre des dizaines de petites décisions, décisions qui en s'ajoutant les unes aux autres donneront corps à votre jeu. Votre interface graphique doit-elle afficher les points de vie de votre personnage ? Allez vous choisir une vue à la première ou à la troisième personne ? La musique sera-t-elle présente tout au long de votre jeu ou ne se fera-t-elle entendre qu'en annonce de passage bien précis ?

Toutes ces questions sont des questions de *game design*. Y répondre vous fera prendre vos premières décisions de *game designer*. Décider de faire un jeu est en soit une réponse à la première et l'une des plus importantes questions : Est ce que je veux faire un jeu ? Félicitations donc, vous venez de devenir *game designer*. Le chemin qui amène le *game designer* vers la fin de son jeu est long (mais la voie est libre). Afin de vous aider à le suivre jusqu'au bout en évitant le plus possible les embuches, nous allons voir ensemble quelques notions importantes.

## QU'EST CE QU'UN JEU ?

Cela peut paraître trivial de poser cette question ici. Nous savons tous ce qu'est un jeu. Un jeu, c'est quelque chose auquel on joue. Mais encore ? Ne pourrait-on pas définir les choses plus précisément ? Ne pourrait-on donc pas ensuite utiliser cette définition du jeu pour pouvoir en tirer les règles qui nous permettraient immédiatement de définir ce qu'est un bon jeu ? Et donc, par réciprocité, de définir un mauvais jeu ? Ce qui nous permettrait d'être sûr de pouvoir toujours faire un bon jeu. Nous allons voir qu'il y a plusieurs définitions d'un jeu. Par contre, il n'y a malheureusement pas de règles magiques pour savoir si un jeu est bon ou pas. C'est bien malheureux, mais c'est aussi cette difficulté qui donne son intérêt à la création de jeux vidéo.

Il y a cependant quelques critères qui nous permettent d'analyser si un jeu est bon. Cette petite théorie "*Toy, Immersive, Goal*" a été introduite par Carsten Wartmann et Michael Kauppi dans le livre *The Blender GameKit* (Licence Creative Commons Attribution 3.0). À divers degrés, un jeu doit remplir plus ou moins les critères suivants.

**L'aspect jouet (*Toy*)** : ce critère reflète le plaisir immédiat que l'on a à jouer. Pas besoin de réfléchir trop, on commence à jouer comme on le faisait quand on était enfant. Pas besoin de lire le manuel, on sait intuitivement comment s'amuser. Cela ne veut pas dire que ces jeux ne peuvent pas devenir difficiles à jouer et qu'il ne faudra pas un certain degré de compétences pour arriver au bout, mais on a un plaisir immédiat à y jouer.

**L'aspect immersif (*Immersive*)** : ce critère reflète à quel degré vous oubliez que vous êtes en train de jouer à un jeu, ce qui a aussi été appelé la "suspension de l'incrédulité". Cela vient peut-être du fait que les jeux sont si réalistes que l'on se croit dans la réalité, mais aussi parce que l'univers est si riche, si cohérent et consistant que l'on y croit complètement, et que l'on se croit dans le monde du jeu. Bien sûr, le joueur sait toujours qu'il joue, mais jusqu'à un certain point, il va se projeter dans le monde du jeu. Par exemple, un phénomène assez fréquent quand on joue est de perdre la notion du temps, parce qu'on se met mentalement au rythme du jeu.

**L'aspect objectif (*Goal*)** : ce critère reflète à quel point le jeu vous emmène vers l'accomplissement d'un objectif, d'un but à atteindre, en passant par un certain nombre d'étapes plus ou moins difficiles et complexes. Le fait d'avoir un objectif peut être lié au scénario, ou simplement être l'aboutissement du principe de jeu (finir la course). Il peut y avoir plusieurs objectifs, et ceux-ci construisent la motivation du joueur à aller au bout.

Un bon jeu aura donc atteint un **équilibre** entre ces trois aspects, en fonction de ce qu'il est. Idéalement les trois aspects seront très présents, mais vous devrez probablement sacrifier pourtant un peu de l'un pour en renforcer un autre. Par exemple, plus les objectifs sont complexes et la stratégie élaborée, plus il est difficile de conserver l'aspect jouet, mais ça ne veut pas dire que cet aspect doit être complètement absent.

### Mais alors, qu'est-ce qu'un jeu ?

Beaucoup de *game designers* se sont essayés à donner une définition. Nous allons en citer deux. Pour George Santayana, « *Jouer est tout ce qui se fait spontanément et pour le simple plaisir de jouer* ». Tracy Fullerton définit un jeu comme « *un système formel, fermé, qui engage les joueurs dans un conflit structuré, et se termine dans une issue inégale* ». Nous pourrions en donner d'autres. Vous pourriez essayer de trouver votre propre définition. Il y a en fait quasiment autant de définitions du mot jeu que de personnes essayant d'en donner une. Il y a pourtant des points communs. On retrouve très souvent les notions de conflit au sens large, de plaisir, d'initiative personnelle et de résolution de problèmes. Gardez ces points à l'esprit lorsque vous allez créer votre jeu et tenez en compte lorsque vous prenez des décisions.

*Article en français Wikipédia sur George Santayana :*
*http://fr.wikipedia.org/wiki/George_Santayana.*
*Article Wikipédia en anglais sur Tracy Fullterton :*
*http://en.wikipedia.org/wiki/Tracy_Fullerton.*

## QU'EST-CE QU'UN JOUEUR ?

Nous avons beaucoup parlé du jeu. Mais un jeu sans joueur n'est pas vraiment un jeu. Il faut qu'on joue à un jeu pour qu'il prenne vie. Bien entendu, chaque créateur de jeu voudra que son jeu plaise à toutes et tous, ou au moins au plus grand nombre. Pourtant faire un jeu qui plaira à tous revient souvent à réduire les originalités de son jeu pour en faire quelque chose de plus lisse et donc souvent plus fade.

En fait, on classifie souvent les joueurs en groupe. Les classifications changent en fonction du type de jeu que l'on adresse. Vous allez par exemple trouver la classification de Richard Bartle qui s'axe plus sur les joueurs de MMORPG ou de jeux de rôles. Elle permet de définir que des joueurs sont plus intéressés par l'axe d'exploration du monde des jeux auxquels ils jouent tandis que d'autres préfèreront suivre l'histoire et en découvrir toutes les ramifications. Dans un autre domaine, Amy Jo Kim a, quant à elle, défini une taxonomie des joueurs de jeux ayant une composante sociale. On voit alors que plusieurs groupes se détachent, certains joueurs étant très réceptif à la compétition, d'autres préférant largement les systèmes de collaboration entre joueurs.

Nous avons parlé du jeu en lui-même ainsi que des joueurs. Il nous reste une dernière chose à voir avant de passer aux éléments constitutifs des jeux eux-mêmes, c'est l'expérience de jeu.

*Article Wikipédia en anglais sur Richard Bartle :*
*http://en.wikipedia.org/wiki/Richard_Bartle*
*Article Wikipédia en anglais sur Amy Jo Kim :*
*http://en.wikipedia.org/wiki/Amy_Jo_Kim*

## QU'EST-CE QU'UNE EXPÉRIENCE DE JEU ?

L'expérience de jeu est ce qui naît de la rencontre entre le joueur et le jeu. Créer une expérience de jeu est le but ultime de votre jeu. C'est la finalité de celui-ci. L'expérience utilisateur va regrouper tout ce que ressent votre joueur. Normalement, en jouant à votre jeu, il va éprouver des sentiments. Il pourra éprouver de la joie, de la peur, de la tristesse. Il pourra éprouver de la colère ou un peu de frustration. Toutes ces émotions seront créées par son interaction avec votre monde. Toutes ces émotions sont intéressantes et vous devez rechercher la mise en place de ces émotions. La seule chose que votre joueur ne devrait pas ressentir est l'ennui.

Lorsque vous concevez votre jeu n'oubliez pas que le but final de ce dernier est de faire vivre une expérience à ses joueurs. L'une des premières décisions que vous allez avoir à prendre sera donc de définir l'expérience que vous allez vouloir offrir. Est-ce que vous allez vouloir mettre en place un *survival horror* où les joueurs auront la peur de leur vie à chaque déplacement ? Est-ce qu'au contraire vous allez vouloir leur proposer un univers onirique et mignon plein de couleurs et de bons sentiments ?

Toutes vos décisions futures doivent renforcer la construction de l'expérience que vous voulez proposer. Un zombie en état de décomposition avancé sera parfait pour un *survival horror*, bien qu'un peu classique. Un joli lapin souriant, lui, ira bien dans un univers mignon et coloré, mais pas vraiment pour votre *survival horror*. A moins qu'il soit en train de se faire dévorer par le zombie !

Maintenant que nous avons vu ces trois notions importantes mais très théoriques nous allons nous attacher à détailler quelques composants importants des jeux.

## DÉCOUPAGE TEMPOREL D'UN JEU

On découpe généralement le temps passé à jouer à un jeu en trois parties.

### L'*onboarding*

Au début d'un jeu, on retrouve ce que l'on appelle l'*onboarding*. L'embarquement est une assez bonne traduction. C'est en effet la partie où votre joueur va découvrir votre jeu. C'est pendant ce laps de temps qu'il va décider si cela vaut le coup de continuer ou pas. Si votre jeu va réussir à l'embarquer dans son monde.

Une méthode d'embarquement classique est de commencer le jeu par un tutoriel. Votre joueur peut alors se familiariser avec la manière de jouer, les possibilités de son personnage, etc. L'important ici est de proposer une découverte progressive des possibilité que votre joueur aura. Cela peut aussi passer par des systèmes d'énigmes simples à résoudre qui mettent, chacune, en lumière, une des possibilités du jeu. Dans *Plants VS Zombies* (un jeu de *tower defense* où vous devez protéger votre maison d'une invasion de zombies en plantant des végétaux qui les détruiront), les premiers niveaux ne sont là que pour introduire une par une les nouvelles plantes. Lorsque vous avez vu toute la palette possible des unités végétales, alors le jeu démarre réellement.

Les mécanismes de mise en place de la découverte d'un jeu sont vraiment multiples. Ne vous laissez pas limiter par ce dont vous avez l'habitude et innovez. Dans *Bastion* par exemple, la découverte du jeu est mise en place par une voix *off* qui va proposer au joueur de tester différents mécanismes. Il lui suffit de se laisser guider. C'est une manière de faire très astucieuse qui permet de réduire un peu l'artificialité que certains joueurs reprochent aux mécanismes d'*onboarding*. Certains ont pour parti pris de ne pas du tout mettre en place d'*onboarding*. Le joueur est alors directement catapulté dans le vrai jeu et doit tout découvrir seul, par mécanisme d'essai et d'erreur. Un certain nombre de joueurs sont très friands de cela. Pour d'autres, c'est au contraire déroutant. A vous de faire les choix en conscience.

### Le corps du jeu

C'est la partie la plus longue du jeu. Le joueur va vivre l'expérience que vous proposez en parcourant votre jeu. Peu à peu, il progressera dans sa compréhension et dans les mécanismes, et il deviendra de plus en plus habile à résoudre les conflits et les énigmes que vous lui proposerez.

Il vous faudra donc doser les choses avec doigté. La difficulté de votre jeu doit accompagner la montée en compétences de votre joueur. Si la difficulté ne croit pas assez vite, votre joueur s'ennuiera. Et rappelez-vous, l'ennui est la seule chose que vous ne voulez pas que votre joueur ressente. Par contre, si vous corsez les choses trop rapidement, votre joueur ressentira très rapidement de la frustration. Et, autant ressentir une légère frustration est important, — c'est l'aiguillon qui pousse en avant beaucoup de joueurs — autant ressentir trop de frustration sera vécu comme une injustice par votre joueur qui ne comprendra par la raison de cet acharnement. Et cela pourrait bien l'arrêter de jouer à votre jeu.

### La fin de votre jeu

Ici, les choses sont vraiment très différentes en fonction du type de jeu que vous souhaitez mettre en place. Un jeu avec une fin définie et connue (comme arriver à la fin de tous les niveaux d'un *puzzle game* ou libérer le prince qui s'est bêtement fait kidnapper par une sorcière) possède une fin en soi.

Il y a toutefois des jeux qui n'ont pas de fin. C'est par exemple le cas des jeux en ligne multijoueur ou d'un jeu qui peut se reparcourir de multiples fois. Dans tous les cas, la problématique est la même. Le joueur a atteint un niveau de compétences suffisant pour arriver au bout du jeu où son personnage a atteint le maximum de l'évolution proposée par le système. Il faut lui donner une raison de ne pas passer à un autre jeu. Il faut donc lui offrir de nouveaux *challenges*. Soit du nouveau contenu à explorer, soit la possibilité d'entrer en compétition avec les autres ou lui-même. Pourra-t-il refaire le jeu en moins de 10 minutes ? Pourra-t-il le faire sans perdre une seule vie ? Pourra-t-il trouver une autre fin plus satisfaisante pour lui à l'histoire que lui raconte votre jeu ? Ce sont des ressorts possibles. À vous de trouver ceux qui correspondront bien à votre jeu pour donner envie à votre joueur de le relancer.

### Interface entre le jeu et le joueur

On réduit souvent l'interface entre le jeu et le joueur à l'interface graphique. Mais, l'interface graphique est loin d'être la seule interface que le joueur aura avec le jeu. En effet, pour jouer à votre jeu, votre joueur va devoir agir sur celui-ci. Et en réponse aux actions du joueur, le jeu va devoir réagir et faire connaître au joueur quel est le nouvel état global du jeu. De même vous pouvez choisir d'informer votre joueur de la valeur de certaines données relatives au personnage de votre joueur, au monde de votre jeu, etc. C'est tout cela que représente l'interface du jeu.

En réfléchissant sur l'interface de votre jeu, vous allez vous rendre compte que c'est un point central dans la construction de l'expérience de votre joueur. Imaginons que vous conceviez un jeu qui va proposer des quêtes, allez-vous afficher sur une carte les lieux où il pourra les résoudre ou allez-vous simplement mettre à disposition un journal des quêtes décrivant les choses et les lieux et en lui laissant la charge de les trouver ? Si le personnage de votre joueur a des points de vie, allez-vous lui donner une indication précise numérique ou allez-vous, alors, le laisser dans le flou en ajoutant simplement un filtre rouge sang se fonçant selon la gravité des blessures reçues ? Si vous développez un jeu de tir, allez-vous afficher un réticule de visée ou pas ? Allez-vous auréoler les obstacles d'une couleur permettant aux joueurs de savoir s'ils sont surmontables ou allez-vous le laisser dans l'inconnu ?

Réfléchir sur l'interface entre votre jeu et votre joueur est donc un point important qui est bien souvent malheureusement négligé et qui est donc la source de beaucoup de frustrations inutiles. Combien de fois dit-on d'un jeu que bien qu'il soit excellent, l'interface graphique est tellement mal conçue qu'elle détruit tout le plaisir du jeu ? Enfin n'oubliez pas que la partie visuelle n'est pas la seule interface possible. Périphérique à retour de force ou ambiance sonore sont des interfaces possibles. Des jeux entiers reposent sur le fait que les sons entendus par le joueur lui donnent des informations vitales pour sa progression.

## OBJECTIFS DANS LE JEU

Pourquoi les joueurs jouent-ils à des jeux ? Ou plutôt pourquoi continuent-ils à jouer une fois la phase de découverte passée ? Pour finir le jeu, bien entendu. Mais pourquoi est-ce que ce besoin de finir le jeu se fait-il sentir ? Parce qu'un jeu bien construit défie le joueur. Il lui propose des objectifs à remplir. Vous devez trouver le trésor, récolter des pièces d'or ou tout simplement comprendre ce qui se passe. Comprendre pourquoi la situation de départ du jeu était ce qu'elle était.

Mais il ne suffit pas à un jeu d'avoir des objectifs pour être prenant. Il faut que les objectifs soient motivants pour le joueur. Un facteur de motivation important est de mettre en place des objectifs qui ont une valeur endogène à votre jeu, ou qui sont reliés à des objets ayant une valeur endogène. Par endogène on veut dire que l'objectif ou les objets ont une valeur en eux-mêmes dans le jeu. Cela peut-être de trouver les quinze pièces d'or cachées pour atteindre un niveau secret par exemple. En général, on reconnaît un bon objectif au fait qu'il a plusieurs caractéristiques.

Un bon objectif sera concret, atteignable et gratifiant. La gratification se manifeste de différentes manières mais elle doit exister. Cela peut être un bonus de point, un nouveau niveau, un gain d'objet ou tout simplement le gain d'un accomplissement dans son profil (on parlera d'*achievement*).

Enfin un objectif doit être clair. Mais attention, clair ne veut pas forcément dire qu'il doit être défini explicitement par le jeu. De même, il n'est pas inintéressant de fournir des objectifs de plusieurs niveaux. Cela permet au joueur de choisir s'il se contente de valider les objectifs principaux ou s'il tente de tous les remplir. Un bon exemple d'un tel mécanisme d'objectif est présent dans le jeu *Limbo*. Lorsqu'on y joue, l'objectif principal est de comprendre, comprendre la situation initiale du personnage et ce qui se passe. Cela implique de finir le jeu. Au cours du parcours, s'il explore un peu, le joueur peut trouver un œuf. S'il l'écrase il gagnera automatiquement un accomplissement. Cela crée automatiquement un nouvel objectif, secondaire, récolter tous les œufs. L'objectif en lui même est parfaitement clair. On écrase un œuf, on gagne un accomplissement. Mais il a tout de même fallu que le joueur le découvre par lui-même. La découverte d'objectifs cachés devient en elle-même un objectif qui peut-être un puissant moteur pour le joueur.

## ÉQUILIBRAGE

On va pouvoir équilibrer plusieurs choses dans un jeu. On peut toutefois regrouper le tout en deux grands ensembles, l'équilibrage joueur contre joueur et l'équilibrage du jeu.

### Équilibrage joueur contre joueur

On va ici pouvoir à nouveau découper en deux ensembles différents, à savoir, si l'on a des systèmes symétriques ou des systèmes asymétriques.

#### Système de jeux symétriques

Le jeu d'échec est par exemple un système symétrique. Tout comme la plupart des sports, chaque joueur possède les mêmes possibilités. Dans le jeu vidéo, on pourra citer des jeux tels que *Towerfall* ou *Samourai Gunn*. Il n'y a là du coup rien à équilibrer vu que les possibilités sont exactement identiques. On n'équilibre pas les pions blancs comparés aux pions noirs aux échecs. Il s'agit alors de pouvoir proposer des solutions pour équilibrer des différences importantes de compétence de jeu entre plusieurs joueurs. On ne peut le faire qu'en donnant des bonus aux joueurs les plus faibles ou en infligeant des handicaps au plus puissants. Dans *Towerfall* par exemple, le joueur le plus faible se voit automatiquement offrir des bonus en début de partie.

#### Système de jeux asymétriques

Beaucoup de jeux proposent un fonctionnement différent basé sur des façons de jouer asymétriques. On peut citer tous les jeux de type 0 A.D., *Warcraft* ou *Starcraft* qui proposeront plusieurs races jouables différentes. Cela peut également être des jeux tels que *Nexuiz* ou *Xonotic*. Ici, ce sont les armes qu'il faudra équilibrer pour être sûr que, dans une même catégorie, il n'y a pas d'arme plus puissante qu'une autre et qu'il n'y a pas non plus d'arme ultime dont l'utilisation donnerait un avantage irrémédiable à un joueur. On doit aussi par exemple équilibrer les capacités des différents karts disponibles dans *SuperTuxKarts*.

Comment réussir à équilibrer des choses asymétriques ? C'est en effet un exercice assez difficile. La solution la plus sûre consiste à mettre en place des systèmes de notation. Il vous faudra définir les caractéristiques importantes du système qui permet d'évaluer l'équilibre. Pour une arme cela pourra être ses dégâts, son temps de rechargement, sa précision, sa portée et le temps entre deux tirs. Une fois votre grille de notation prête, il vous faudra noter chaque objets à équilibrer. Une fois cela fait, vous n'avez plus qu'à vérifier que les moyennes de tous vos objets sont à peu près équivalentes.

### Équilibrage du monde.

On va retrouver, encore une fois, deux grandes catégories dans ce que nous appelons équilibrage du monde. Nous allons commencer avec l'équilibrage de la difficulté et nous finirons avec celui des choix.

#### Équilibrage de la difficulté

On l'a vu au début de ce chapitre, pendant tout le corps du jeu, il faut augmenter petit à petit la difficulté des *challenges* qui seront proposés aux joueurs. C'est l'équilibre entre réussite et frustration, difficulté et plaisir qui fera que les joueurs continueront à jouer. Mais comment doser cette difficulté ? Comment savoir quel levier augmenter ?

Là aussi, la méthode la plus sûre pour y arriver est celle de la notation « multi-critère ». Imaginons que vous créiez un jeu d'escarmouche. Votre joueur va devoir se battre contre des équipes ennemies. Il va vous falloir doser la puissance des équipes en question. Une façon simple est alors de définir une note pour chaque créature pouvant composer l'équipe. Pour construire des *challenges* plus difficiles et donc des équipes plus fortes, il vous suffira de vous autoriser à chaque fois un peu plus de points pour la construction d'une équipe. La première équipe que le joueur affrontera devra coûter moins de douze points tandis que la dernière pourra dépasser les deux cents.

On peut avoir des jeux faisant intervenir des opposants non joueurs. Cela peut par exemple être un jeu de *karting* où il va vous falloir modéliser le comportement des concurrents et simuler une courbe de progression de ces compétiteurs artificiels pour que le joueur ait toujours une concurrence à peu près à son niveau. Ici, l'astuce est de faire comme lorsque deux joueurs de niveau différents s'affrontent en sport, mettre en place des handicaps pour les bots, handicaps qui seront très importants au départ et qu'il faudra alléger au fur et à mesure.

#### Équilibrage des choix

Un jeu est une succession de choix. Mais pour être intéressants, pour être stimulants, il faut que les choix soient utiles et impactants. Noyer le joueur sous une avalanche de choix n'ayant aucun poids sur le déroulement du jeu ou de l'histoire est totalement inutile, voir contre-productif. Après tout, si c'est pour être passif devant un écran sans avoir de réelle possibilité d'intervention, la plupart de vos joueurs préfèreront aller regarder un bon film. *The walking dead*, le jeu saison 1, est un bon exemple de cela. Les choix que l'on vous proposera auront un impact direct et immédiat. Dans *The walking dead*, vous vous retrouverez parfois à choisir qui va survivre et donc qui va mourir entre deux des personnages que vous côtoyez.

Une pratique intéressante lorsque l'on construit des choix est de tenter d'aller plus loin que le simple choix binaire, et de proposer des choix jouant sur une ambivalence : risque limité avec petite récompense, gros risque lié à grosse récompense. Allez-vous prendre le chemin de droite beaucoup plus rapide, mais vous permettant de collecter moins de pièces ou celui de gauche qui vous fera faire un détour, mais vous permettra de faire le plein de trésors ? Un bon conseil à propos de la mise en place de choix est de toujours essayer d'éliminer le concept de bon et de mauvais choix.

Il nous reste maintenant une dernière chose à voir dans cette courte introduction au *game design*, les récompenses et leur pendant, les punitions.

## RÉCOMPENSES ET PUNITIONS

Les jeux sont des systèmes visant à résoudre des conflits, des énigmes. Mais qui dit résolution, dit gain, récompenses. Un joueur qui réussit une épreuve s'attend en effet à être récompensé. Bien entendu, vous pouvez totalement prendre ce principe à contre-pied et proposer un jeu n'ayant pas de récompense. Mais si vous décidez d'intégrer le fait qu'une résolution réussie offre une récompense, il faudra alors vous pencher sur la mise en place ou pas d'un système de punitions en cas d'échec.

Mais pourquoi voudriez-vous mettre en place un système de punitions ? Là encore, ce n'est pas une obligation et de nombreux joueurs n'apprécient pas du tout d'être punis par leur jeu. Mais la punition dans un jeu vidéo est toutefois un ressort important. Du simple fait de sa présence, elle augmente mécaniquement la valeur que vos joueurs vont accorder aux récompenses que vous leur offrez. S'ils savent qu'un échec peut leur valoir de perdre le magnifique objet qu'ils viennent de gagner, le *challenge* n'en devient que plus important. Les décisions que vous leur demanderez de prendre deviendront des décisions importantes, dont ils pèseront vraiment l'impact. Mieux encore, l'échec peut alors devenir mémorable et générer un vrai souvenir dont on reparle ensuite entre joueurs.

Il n'y a bien entendu pas que la punition par perte d'objet ou de ressource qui soit possible. Dans certains cas, on peut imaginer ce qu'on appelle la mort véritable, en clair : le jeu est fini. Il faut tout recommencer depuis le début. On peut aussi voir la punition comme un ressort presque comique. Si l'on reprend *Bastion* comme exemple, la voix *off* lorsque vous échouez une action d'une manière vraiment importante se moque de vous. L'effet est garanti.

Ce dernier point sur les récompenses et les punitions clôt donc cette présentation de ce que l'on appelle le *Game Design*. Il y aurait encore beaucoup à dire sur le sujet et si vous souhaitez creuser les choses, ce n'est pas la littérature qui manque. Vous en savez toutefois maintenant largement assez pour pouvoir imaginer et construire votre jeu. Et n'oubliez pas, pour être un *game designer*, il suffit de créer un jeu. Alors n'attendez plus, tournez la page, apprenez à mettre en place votre environnement *Blender Game Engine* et créez !

# 4. BIEN PRÉPARER SON PROJET

Créer un jeu basique ne pose pas de soucis d'organisation, mais dès que notre projet va grandir, nous aurons des images et des sons qui y seront associés, des fichiers textes contenant le code, *etc.*

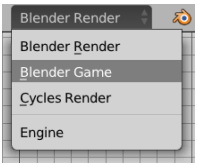
Une bonne pratique est de se préparer dès le départ et de travailler de façon méthodique. Cela nous permettra d'éviter et/ou de corriger des erreurs plus facilement.

Il appartient à chacun de définir l'organisation optimale pour son travail, mais nous vous proposons ici les éléments qui nous semblent les plus importants. Seul le premier point est indispensable pour démarrer avec le *Blender Game Engine (BGE*).

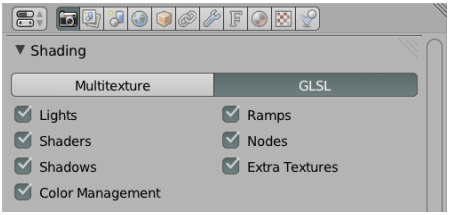
## CHANGER LE MODE UTILISÉ

Blender est par défaut réglé pour faire du rendu avec son moteur de rendu interne (`Blender Render`), mais nous voulons faire du jeu, donc nous allons changer le mode de Blender.

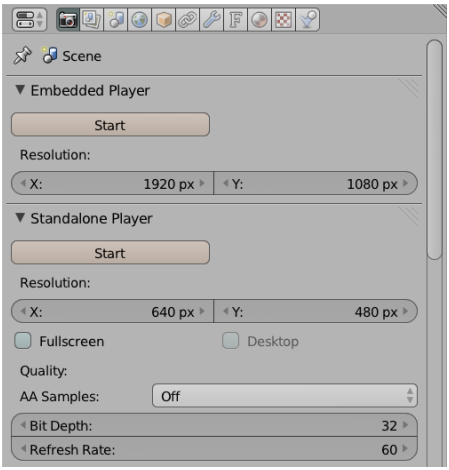
Ce changement s'opère en haut à droite de Blender, choisissons ***`Blender Game`***.




Nous choisirons aussi, dans le panneau dédié, le *shading*`GLSL` dans l'onglet`Render` pour nous offrir plus de possibilités et de réalisme dans l'affichage en temps réel.




Remarquons également que l'onglet `Render` a sensiblement changé depuis la sélection du mode`Blender Game`. C'est ici que se trouvent les configurations du *BGE*. Nous pouvons déjà lancer le moteur de jeu en positionnant la souris au-dessus de la vue 3D en appuyant sur **`P`**, même si pour l'instant rien ne se passe. `Échap` ou `Esc` est la touche par défaut pour fermer le mode interactif et retrouver les fonctions d'édition de Blender.




Nous avons alors deux options de rendu qui sont placées dans des onglets séparés :

- Le `Embedded Player` (lecteur intégré) permet de lancer le moteur dans Blender directement (comme `P` dans la vue 3D).
- Le `Standalone Player` (lecteur autonome) permet de lancer le jeu dans une fenêtre séparée avec davantage d'options (résolution précise, plein écran, l'antialiasing). Le jeu, une fois exporté en exécutable indépendant se comportera au plus proche de l'aperçu obtenu dans ce mode (il ne faudra pas oublier d'empaqueter les ressources avant, nous l'aborderons dans la section *Créer son premier jeu*, chapitre *Partager son jeu*).

*Attention, il faut **toujours sauvegarder le .blend avant de lancer le jeu**. Pendant sa conception, un jeu plante très souvent et sans sauvegarde au préalable, le travail serait perdu.*

## BLENDER SUR PLUSIEURS MONITEURS

Si vous avez la possibilité d'utiliser plusieurs écrans, il peut être assez confortable de dédier votre second moniteur à la vue 3D, et au test de votre jeu, pour profiter de tout l'espace disponible sur l'écran principal pour le réglage des briques logiques et l'écriture du code (ou tout l'inverse !).

Pour cela, il faut ouvrir une nouvelle fenêtre à l'intérieur de Blender.

Au lieu d'attraper (cliquer-glisser) l'angle supérieur droit ou inférieur gauche d'un éditeur (les trois traits diagonaux) pour le séparer en deux, ou le joindre à un autre, il suffit d'appuyer sur `Maj` et de la maintenir appuyée avant de cliquer-glisser sur cet angle. Blender détache alors une nouvelle fenêtre avec le même éditeur, par exemple la vue 3D, que vous pouvez placer sur un autre écran.

## CRÉER UN RÉPERTOIRE POUR CONTENIR NOTRE PROJET

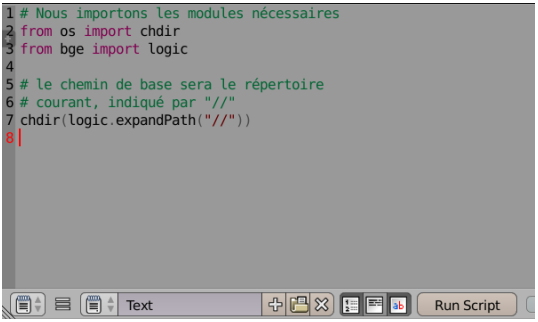
Fondamentalement, la première bonne habitude à prendre pour le projet lui-même est d'organiser ses ressources dans un dossier propre au jeu (avec différents sous-dossiers pour les images, les sons, *etc.*). Cela facilitera la recherche des fichiers et simplifiera l'importation des éléments dans Blender par la suite.

## RÉGLAGE DE L'ENVIRONNEMENT DE DÉVELOPPEMENT PYTHON

Votre jeu, s'il est un peu évolué, contiendra certainement des scripts Python. Il serait en effet fastidieux de vouloir utiliser systématiquement les outils graphiques de Blender : s'obstiner à tout définir en briques logiques peut s'avérer contre-productif. Les outils graphiques mettent à disposition des options courantes, et passé les premiers temps vous pourrez vous sentir contraints.

## LES SCRIPTS PYTHON

Les jeux dans Blender se programment en Python. C'est un langage largement utilisé dans les applications graphiques puisqu'il est aussi utilisé par exemple dans Gimp, Inkscape ou Scribus. Il représente donc un bon investissement pour de nombreuses personnes, y compris celles venant du monde de la création visuelle.




*L'éditeur de texte interne de Blender*

Pour enregistrer notre code, deux options s'offrent à nous :

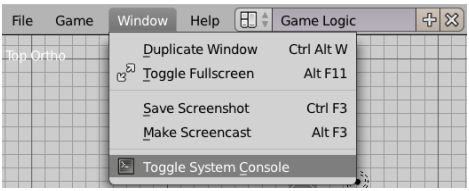
- **Utiliser l'éditeur de texte interne de Blender** : Les scripts seront enregistrés directement dans le .blend. Cette méthode est la moins compliquée, mais n'offre pas autant de malléabilité ni de sécurité : Par exemple, si Blender (ou le moteur) plante et que le .blend n'a pas été sauvegardé, les scripts (et autres données) seront perdus. En affichage `Blender Game`, l'éditeur de texte est directement disponible dans l'écran de travail *Scripting*. On pourra ainsi en créer autant que nécessaire. Il suffit alors de cliquer sur le bouton `New` pour créer un nouveau texte. Pour rendre le texte plus agréable à lire et plus compréhensible pour la programmation, il est possible d'activer la coloration syntaxique et la numération des lignes via deux des trois boutons dédiés dans le *header* de l'éditeur de texte (juste à gauche de `Run Script`).
- **Utiliser un éditeur de texte (ou plutôt de code) externe** : en cas de plantage imprévu de Blender, les dernières modifications sur les scripts ne seront pas perdues. Cela facilite aussi le travail avec des collaborateurs qui peuvent alors utiliser leur outil d'écriture favori. En revanche, il faut placer les scripts e t *packages* dans le même répertoire que celui du .blend. Si nous empaquetons le jeu (pour l'exporter par exemple), les scripts ne seront pas ajoutés dans le .blend (contrairement aux images et sons). Il faudra donc les ajouter à la main en les ouvrant depuis l'éditeur de texte de Blender.

### La console

Pour le développement (en Python), la console sera notre outil indispensable. Elle permet de faire apparaître les messages au fur et à mesure du jeu : les messages d'erreurs, ceux sur l'évolution d'une action ou d'une variable. Ces informations seront utiles pour débugger et essayer de résoudre les problèmes éventuels. Certains éditeurs Python intègrent des consoles et on pourrait imaginer lancer Blender à partir d'eux. Certains terminaux comme Quake permettent un coup d'œil rapide, voire, si vous utilisez une console standard, de la configurer pour qu'elle reste au premier plan.

#### Sous Windows

Si vous utilisez Blender sous Windows, dans le menu, nous choisissons `Window > Toggle System Console`.




Et une belle fenêtre apparaît.

**Sous Linux**

Sous Linux, il y a deux manières d'installer Blender et donc deux manières de le lancer. Vous pouvez l'installer en utilisant votre gestionnaire de paquet. C'est la façon la plus simple et la plus rapide. Il faut toutefois faire attention, suivant la distribution que vous utilisez, la version de Blender que vous installerez pourra être plus ou moins vieille. Une fois Blender installé, il vous suffit de lancer une console et de simplement taper la commande `blender` suivie d'un appui sur la touche `Entrée`.

Vous pouvez aussi décider de télécharger une archive de la dernière version de Blender et de la décompresser quelque part sur votre disque dur. Une fois Blender installé de cette façon, il faudra pour le lancer que vous ouvriez une console, que vous alliez dans le répertoire contenant Blender (avec la commande `cd`) et que vous tapiez la commande `./blender` suivie d'un appui sur `Entrée`.

**Sous Mac**

Sous Mac, pour démarrer Blender en mode console, vous devrez démarrer un terminal et lancer Blender en ligne de commande. *A priori*, nous considérons que Blender est installé dans vos`Applications`.

L'application`Terminal`(c'est la console de Mac) se trouve dans le dossier`Utilitaires`. Lancez-la. Dans le terminal, écrire la ligne suivante et ensuite appuyer sur `Entrée` pour démarrer Blender

```
/Applications/Blender/blender.app/Contents/MacOS/blender
```

*Si cette commande ne fonctionne pas, vérifiez le chemin vers votre application Blender. En fonction de votre version ou de la façon dont vous avez installé Blender, le chemin pourrait être différent. Faites bien attention à la casse (majuscules et minuscules) également.*

## Gérer les ressources externes, modification du répertoire courant en Python

Un autre point important est la gestion du répertoire courant. Par défaut, lorsque nous lançons Blender depuis un raccourci bureau par exemple, puis chargeons le fichier .blend contenant notre jeu, le répertoire courant (celui où Blender ira chercher les ressources) sera celui où se trouve l'exécutable de Blender (par exemple pour Windows C:\Program Files\Blender Foundation\Blender).

Au contraire, si nous lançons Blender simplement en double-cliquant sur le fichier .blend de notre jeu, le répertoire courant sera celui qui contient le .blend.

Il est donc important de mieux contrôler le contexte de recherche des ressources, en particulier lorsqu'on travaille avec des fichiers externes.

Pour être sûrs que notre fichier .blend fonctionne dans toutes les conditions, nous utiliserons ce petit script au début du jeu pour redéfinir le répertoire courant (attention, il faut bien enregistrer le fichier .blend avant, sinon il est impossible de trouver son répertoire courant) :

```
# Nous importons les modules nécessaires
from os import chdir
from bge import logic

# on va dans le répertoire où se trouve notre blend, indiqué par "//"
chdir(logic.expandPath("//"))
```

Mais que fait ce script en détail ?

Nous commençons par importer deux fonctions **`chdir`** du module `os` et **`logic`** du module `bge`. Il faut que nous indiquions à Blender où aller chercher nos fichiers ressources. Nous allons donc lui demander d'aller dans le répertoire où est sauvegardé notre fichier .blend.

Bien entendu, on ne veut pas écrire directement le chemin absolu de notre fichier. Si nous faisions cela, les ressources ne seraient trouvées que sur notre propre ordinateur mais pas sur celui des autres collaborateurs. Heureusement, Blender propose une syntaxe pour indiquer « le répertoire où se trouve le fichier .blend courant ». On utilise pour cela la chaîne de caractère `//`. C'est ce que fait la ligne de code **`chdir(logic.expandPath("//"))`**. On demande à Blender de nous donner, d'une manière compréhensible par le Python, l'endroit où se trouve le fichier .blend et ensuite on s'y déplace.

Maintenant tous les fichiers ressources que vous déposerez dans le même répertoire que votre fichier .blend seront facilement utilisables dans vos scripts Python en utilisant un simple chemin relatif se basant sur le répertoire de votre .blend. Quelle que soit la façon dont nous démarrons Blender, nous sommes sûrs que notre code fonctionnera !

## PRÉPARER LA PHYSIQUE DE SES OBJETS

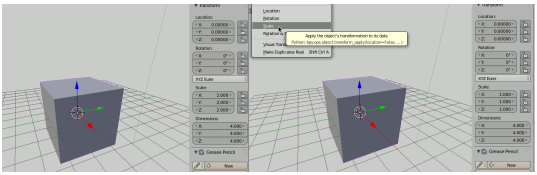
Quand nous modélisons pour du rendu classique, nous avons tendance à ne pas trop nous soucier de la taille des objets. Il suffit que les objets d'une scène aient une taille cohérente entre eux pour que le rendu soit correct.

### Choisir les bonnes unités

Dans un jeu, il y a un moteur physique qui calcule la dynamique des objets parallèlement au moteur graphique. Le moteur physique du BGE est réglé par défaut de telle sorte qu'une unité de longueur Blender soit équivalente à 1 mètre et une unité de masse à 1 kilogramme. Nous conseillons fortement de donner aux objets une taille réaliste, et pour les objets soumis à la gravité une masse réaliste (voir dans la section **Comportement des objets et personnages**, le chapitre **Le comportement des objets de notre monde** pour définir la masse). En respectant ces unités, le comportement physique des objets sera plus conforme à la réalité.

### Éviter les facteurs d'échelle

Il faut éviter impérativement les facteurs d'échelle non-uniforme (taille différente en x, y ou z) car ils ne sont pas supportés par le moteur physique. D'une manière générale, il est préférable d'éviter tout facteur d'échelle quand c'est possible. Pour cela, appliquer le *scale*de votre objet (`Apply Scale` avec le raccourci `Ctrl+A`).

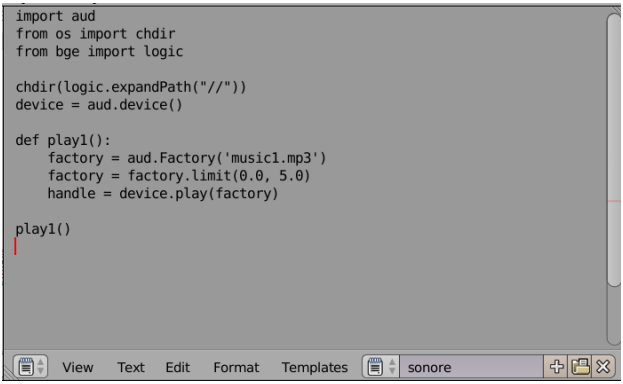


## RETOUR SUR LE DÉVELOPPEMENT DES SCRIPTS PYTHON

Ces paragraphes s'adressent plus particulièrement à ceux qui pensent que leur jeu contiendra une quantité importante de code Python. Si vous prévoyez de ne taper que quelques lignes de Python, vous pouvez sans remords passer au chapitre suivant. En revanche, si vous avez l'intention d'utiliser du Python de manière répétée, il est important de prendre le temps de lire ce qui va suivre.

Nous avons vu précédemment qu'il y a deux manières d'écrire du Python pour Blender : soit directement dans l'éditeur de texte interne, soit dans des modules Python externes. Quelle que soit la méthode que vous déciderez d'utiliser, il faudra que vous connectiez ensuite votre code Python à votre jeu. Vous verrez dans la section **Créer son premier jeu** au chapitre **Déplacer le personnage** qu'il vous faudra pour cela une brique logique `Controller`de type`Python`. Mais pour l'instant retenez simplement que votre script Python doit passer par une brique logique pour fonctionner.

Cette brique permet deux modes de configuration pour le Python. Si vous choisissez le mode *Script*, vous devez donner un nom au script que vous avez écrit dans l'éditeur de texte interne. Ce nom n'a pas besoin d'être construit comme un nom de fichier, c'est tout simplement un nom que vous définissez vous-même en bas de la fenêtre de votre éditeur interne. Par exemple, dans la capture d'écran qui suit, le nom du script est"sonore". Si vous aviez ce texte interne dans votre fichier Blender, vous mettriez donc"sonore"dans la case *Script* de votre brique logique d'activation de code Python.




Vous pouvez aussi décider d'écrire vos scripts Python dans un éditeur spécialement fait pour développer du Python. Dans ce cas, il faut impérativement que vous sauvegardiez vos fichiers Python (que l'on appelle alors des modules) dans le même répertoire que votre fichier .blend.

Dans votre brique logique vous allez alors choisir *Module*et non plus*Script*. Ce n'est plus le nom du script que vous allez devoir saisir mais le nom du module, suivi du nom de la fonction Python que vous voulez utiliser. Les deux doivent être séparés par un point. Imaginons par exemple que nous ayons un script Python qui se trouve dans un fichier *son.py*. Dans ce fichier, nous avons plusieurs fonctions, dont la fonction **`splitch()`**. Pour connecter la fonction **`splitch()`** à la brique logique, il faut sélectionner *Module*et taper **`son.splitch`**.

### Fichier externe ou script interne, comment choisir ?

Nous avons vu en détail qu'il y a deux façons d'intégrer du Python dans un fichier Blender. Quelle méthode choisir ? Disons le tout de suite, dès que nous dépassons le simple fichier de test, la bonne pratique semble être de ne jamais utiliser la fonctionnalité de script interne.

Pourquoi ? Imaginons que l'on ait plusieurs choses très similaires à faire en Python. Par exemple, on veut jouer plusieurs fois le même son mais avec des effets différents. Dans ce cas, il est préférable de regrouper les choses pour plus de facilité. Pour cela, il faut définir différentes fonctions et exécuter la bonne, en accord avec la situation de jeu.

Toutefois, lorsqu'une brique logique Python est configurée en mode *Script*, on ne peut pas choisir un nom de fonction à exécuter, le contenu du script est lancé en totalité. Si nous reprenons notre exemple avec cinq façons de lancer le même son, il faudrait donc créer cinq scripts internes différents.

Maintenant, si nous voulons changer de son pour trois des cinq scripts, il va falloir ouvrir ces trois script et faire trois fois la même modification de nom. Ce qui était simple au départ devient donc fastidieux à gérer, et prend trop de temps.

De plus, il peut s'avérer utile d'avoir un fichier Python séparé pour l'envoyer à quelqu'un et lui demander un avis, par exemple. Envoyer un petit fichier texte sera plus simple et plus léger qu'un fichier Blender, surtout si la personne en question ne connaît pas Blender !

Par ailleurs, et même si cela dépasse largement le sujet de cet ouvrage, une bonne pratique en développement logiciel consiste à faire le moins de répétition possible. L'utilisation de fichier externe rend possible cela en rassemblant plusieurs fonctions dans le même fichier par exemple. On appelle ça factoriser son code.

Enfin, et malgré ce que nous venons de dire, il est tout a fait acceptable d'utiliser les scripts internes pour mettre en place des petites fonctionnalités rapides et simples. Tout au long de ce livre, nous utiliserons donc les deux solutions, en fonction des situations.

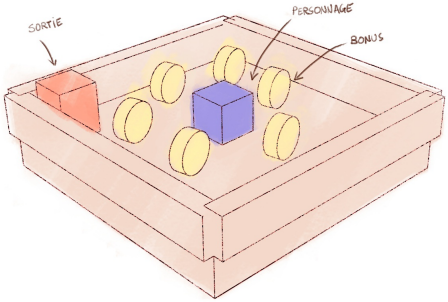
# CRÉER SON PREMIER JEU

# 5. DESCRIPTION DU JEU

Commençons par créer notre premier jeu. Il sera simple, épuré, sans son et peut-être pas très élaboré graphiquement, ou en tout cas pas autant que nous l'espérerions. Il sera toutefois possible de l'étoffer par la suite avec des sons, des graphismes plus riches et davantage d'interactions, mais ces points seront abordés dans les prochaines sections. Il nous semble ici plus important de présenter une première base opérationnelle, un peu comme un prototype. Dans toute production, on aura tendance à séparer ainsi les différents secteurs de travail. Cela nous permettra de nous focaliser sur les fonctions interactives de base et de bien appréhender l'utilisation du moteur temps réel.

*Pour notre premier jeu, nous utiliserons les formes et fonctionnalités intégrées de Blender afin de ne pas avoir à utiliser des ressources extérieures. Vous pourrez personnaliser cela autant que vous le souhaitez. Faites attention cependant à ne pas vous noyer d'emblée dans des projets trop complexes, ce qui pourrait diminuer l'intelligibilité du projet.*

## NOTRE HISTOIRE




Notre personnage, un cube bleu, se trouve dans un espace fermé par des murs, une place orangée. Son objectif sera de parcourir cet espace pour arriver à atteindre le cube rouge. Il pourra au passage récupérer des bonus jaunes dispersés dans l'espace ce qui viendra améliorer le score. Sortir en touchant le cube rouge arrêtera notre niveau en quittant le jeu.

L'histoire est simpliste, mais reprend des principes essentiels souvent utilisés dans des jeux :

- déplacer un personnage ;
- interagir avec l'environnement (cogner et/ou récupérer des objets) ;
- stocker et afficher des données (le score) ;
- déclencher un événement (ouvrir une porte, sortir du jeu).

## LES MODÈLES

Pour réaliser ce jeu, nous allons partir d'objets simples : cinq grands cubes orangés pour le terrain, un petit cube bleu pour notre personnage, des cylindres aplatis jaunes pour représenter les objets à récupérer, un cube rouge pour la porte. Bref, des modèles simples seulement issus des primitives de base de Blender. Si vos compétences en graphisme dépassent ce stade, n'hésitez pas à jouer dessus. Sinon vous pourrez toujours importer des modèles déjà conçus pour le *Game Engine*. Mais ne passez pas trop de temps sur ces aspects dans un premier temps car ce n'est pas une priorité pour cette étape-ci.

Nous avons simplement colorisé ces objets en utilisant la couleur du panneau ***Diffuse*** dans l'onglet `Material` et aussi en réduisant le ***Specular*** à zéro pour supprimer des brillances superflues et finalement en ajoutant un peu d'***Emit*** pour ne pas avoir d'ombres trop noires. Pour ce qui est des textures, nous nous en passerons pour le moment. Il vaut mieux se référer (ou attendre d'arriver) aux chapitres qui traitent en détail des spécificités de leur affichage dans le *Game Engine*.

## CE QUE NOUS APPRENDRA CET EXERCICE

La réalisation de ce mini-jeu nous permettra de parcourir des fonctions fondamentales du BGE :

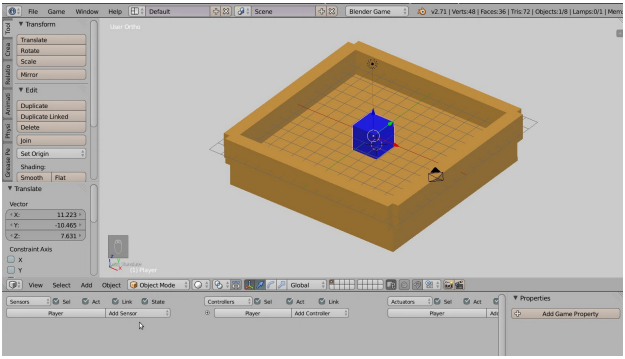
- lancer le jeu ;
- récupérer des événements clavier pour déplacer un personnage à l'aide de briques logiques ;
- manipuler un objet texte pour afficher un score grâce à la création d'une propriété ad-hoc ;
- gérer une collision entre des acteurs du jeu ;
- déclencher un événement comme ouvrir une porte ou sortir du jeu.

Nous en profiterons pour utiliser autant que possible les briques logiques (`logic bricks`) qui sont les éléments de l'interface graphique qui permettent la programmation du jeu. Nous verrons aussi en parallèle comment de petits morceaux de code (plus précisément des scripts Python) peuvent être utilisés pour réaliser les mêmes choses.

# 6. DÉPLACER LE PERSONNAGE

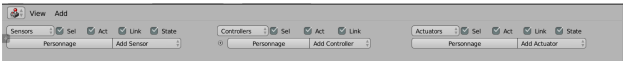
Pour notre jeu, nous avons créé une pièce, représentée par cinq grands cubes orangés et un petit cube bleu pour notre personnage. Une vidéo, *01a_preparation-deplacements.webm*, résume le processus de fabrication de cette scène.

Lorsque notre personnage est sélectionné, nous pouvons alors passer dans l'éditeur ***`Logic Editor`*** pour lui attacher des actions. C'est dans le ***`Logic Editor`*** que nous allons lier des événements-clavier, comme le fait d'appuyer sur la `flèche vers le haut` pour activer le déplacement vers l'avant du personnage.



## INTRODUCTION AU *LOGIC EDITOR*

Dans le ***`Logic Editor`***, il y a trois colonnes :



- La colonne de gauche, concerne les ***sensors***, c'est-à-dire tout ce qui détecte et déclenche une action. Par exemple, le fait que le joueur appuie sur une touche du clavier, bouge la souris ou utilise une manette, mais également quand un personnage du jeu cogne un objet ou quand le scénario du jeu requière le déclenchement d'un événement.
- La colonne de droite, appelée ***`actuators`***, s'occupe d'activer les événements dans le jeu. Comme changer de scène, déplacer un objet, lancer une animation, modifier certaines propriétés du jeu, etc.
- La colonne du milieu, les ***`controllers`***, détermine comment lier les déclencheurs (les ***`sensors`***) et les activateurs d'événements (les***`actuators`***). Par exemple, lorsque mon personnage cogne un objet (***sensor***), cela lance l'animation d'un personnage (***`actuator`***). Il faut donc lier une brique ***`sensor`*** qui détecte la collision avec une brique ***`actuator`*** qui lance l'animation via une brique ***`controller`***.

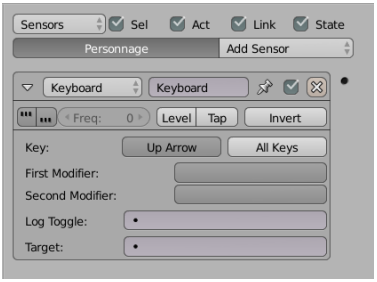
## CRÉER LES TROIS PREMIÈRES *LOGIC BRICKS*

Nous allons donc créer notre toute première logique de jeu !

### Récupérer la frappe au clavier

Pour faire avancer notre personnage, nous allons d'abord créer un ***sensor*** de type clavier.

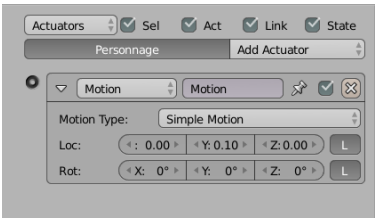
1. Pour cela, appuyons sur le bouton ***Add Sensor*** et choisissons l'option ***Keyboard***.
2. Dans ce ***sensor*** nouvellement créé, il y a une option appelée ***Key*** avec un bouton vide et un bouton ***All Keys***. Cliquons sur le bouton vide et ensuite sur la `flèche vers le haut`. Le bouton s'appelle maintenant ***Up Arrow*** pour bien montrer que c'est lorsque le joueur appuiera sur cette touche du clavier que le ***sensor*** sera activé.




### Activer un déplacement d'objet

Nous ajoutons ensuite dans la colonne de droite, un ***actuator*** qui va effectivement déplacer notre personnage vers l'avant.

1. En cliquant sur ***Add Actuator*** et en choisissant l'option ***Motion***, nous créons cet ***actuator*** de déplacement, qui par défaut se trouve en ***Simple Motion***, ce qui nous convient pour cet exercice.
2. Les deux lignes suivantes représentent les valeurs en x, y et z qui seront ajoutés à la position (***Loc***) ou à la rotation (***Rot***) du personnage dans l'espace lorsque cet ***actuator*** sera activé. En valeur de $y$, nous allons entrer **0.10**, ce qui correspond à un déplacement d'un dixième d'unité Blender.




### Lier frappe au clavier et déplacement d'objet

Nous allons maintenant lier l'appui sur la `flèche vers le haut` avec le déplacement dans l'espace. Pour cela, nous utilisons un contrôleur (***controller***). Les contrôleurs peuvent être de types différents, et vérifient que tout ce qui leur est envoyé en entrée (venant de la colonne de gauche) est actif avant d'activer sa sortie (tout ce qui se trouve en colonne de droite).

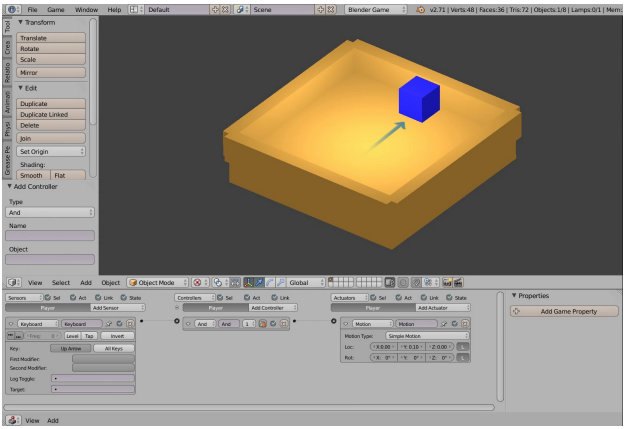
1. Dans la colonne du milieu, nous cliquons sur ***Add Controller*** et choisissons l'option ***And***.
2. Dans notre cas, nous lions la sortie de *sensor* ***keyboard*** avec ce *controller* ***And*** en cliquant et glissant un lien entre la sortie du `sensor` et l'entrée du *controller* à l'aide des petits points situés à droite du premier et à gauche du second. Le lien entre les deux sera ensuite représenté par une ligne liant les deux éléments (voir capture d'écran).
3. Ensuite, nous lions, de la même manière, la sortie du *controller* ***And***avec l'entrée de l'*actuator* ***Motion***.

## Tester le déplacement dans le jeu

Nous pouvons maintenant vérifier que notre personnage avance bien comme on le désire en lançant une pré-visualisation du jeu.

Le meilleur affichage de la vue pour le *Game Engine* est texturé. selectionnez le mode d'ombrage texturé.

Déplacez le curseur de la souris au-dessus de la`vue 3D`et appuyez sur **`P`** pour lancer le *Game Engine*. La souris va disparaître et la fenêtre changer de couleur, signe que le *Game Engine* a démarré. Appuyons maintenant sur `flèche vers le haut` et vérifions que le personnage avance bien. **`Échap`** permet de sortir du jeu.



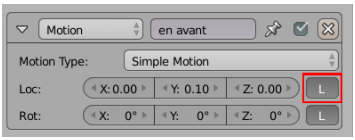
## Résoudre les problèmes

Lorsque nous avons lancé le jeu et appuyé sur `la flèche du haut`, il se pourrait que rien ne se produise. Nous pouvons alors nous placer dans une logique de recherche d'erreur sur différents niveaux.

- Dans un premier temps vérifions que c'est bien `Up Arrow` qui est enregistrée dans le *sensor*`keyboard`.
- Si le réglage précédent est bon, nous pouvons tester une autre touche du clavier en reconfigurant ce *sensor* pour tester uniquement la validité de la touche.
- Vérifions également que nos différents *sensors*, *controllers* et *actuators* sont liés correctement.

Il s'agit d'évaluer à chaque fois les éléments qui ne vont pas et de trouver tout ce qui intervient éventuellement dans le dysfonctionnement et d'évaluer leur validité, un par un, pour les éliminer jusqu'à trouver le bon.

Il est possible aussi que notre personnage ne se soit pas déplacé vers l'avant, mais qu'il avançait de côté, ou même reculait. Peut-être qu'il avançait beaucoup trop vite ou alors très lentement. Retournons voir les paramètres de *Loc* de l'*Actuator Motion*. Si notre personnage n'avançait pas, mais reculait, c'est qu'il faut changer le signe de notre valeur en$_y$et placer **-0.10**. S'il avançait trop lentement, il faut encore augmenter cette valeur à **0.5** par exemple. S'il avançait de côté, il faut plutôt changer les valeurs de position en x ou celles en$_z$. Retenons, que par défaut, ces changements de position se font par rapport aux coordonnées locales de l'objet auquel ils s'appliquent (lorsque le bouton **L**, à droite est coché).

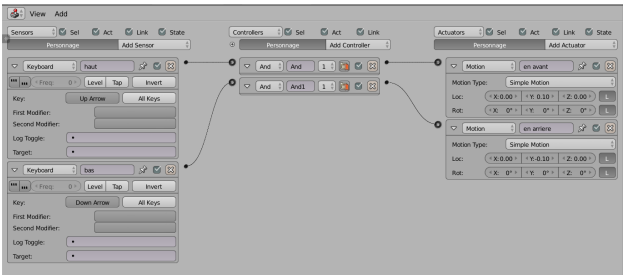



## AJOUTER D'AUTRES DIRECTIONS DE DÉPLACEMENT

À cette étape, notre personnage va dans une seule direction. Bien que cela puisse être déjà un jeu amusant en soi, nous serons vite tentés d'ajouter d'autres degrés de liberté de mouvement, comme la possibilité de reculer, tourner à gauche, tourner à droite, courir, voler, *etc*. En bref, tout ce que nous avons défini dans la phase de conception du jeu (*game design*).

### Faire reculer le personnage

Nous devons ajouter un nouveau *sensor* qui va récupérer l'événement quand le joueur appuiera sur `flèche vers le bas` du clavier. Nous devons créer un nouvel *actuator* `Motion`qui appliquera un déplacement négatif en $_y$à notre objet, par exemple **-0.10**. Et nous devons créer un *controller* `And` qui liera cet événement clavier avec le déplacement de l'objet vers l'arrière.

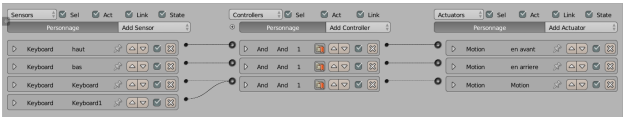



### Organiser le *Logic Editor*

Nous sentons bien que lorsque nous allons ajouter de multiples briques logiques pour gérer les différentes façons de déplacer notre personnage, nous allons vite avoir un grand nombre d'éléments. Il va falloir organiser. C'est pourquoi il est important de nommer les briques de manière claire et descriptive. Nommer nos *sensors*, *actuators* et *controllers* nous sera bien utile lorsque nous commencerons à utiliser des scripts Python afin de les identifier et de récupérer leurs paramètres au sein de notre code.

En cliquant sur le champ textuel à côté de la liste décrivant le type de *sensor* en haut de la brique, renommons **en haut** le *sensor Keyboard* qui récupère la touche clavier `flèche vers le haut` . Renommons **en avant** l'*actuator Motion* qui fait avancer le personnage, et ainsi de suite pour les autres : **en bas** et **en arriere**.

En cliquant sur la petite flèche située à gauche dans l'en-tête de la brique, il est possible de la réduire pour qu'elle occupe moins d'espace et ainsi avoir un meilleur aperçu global.

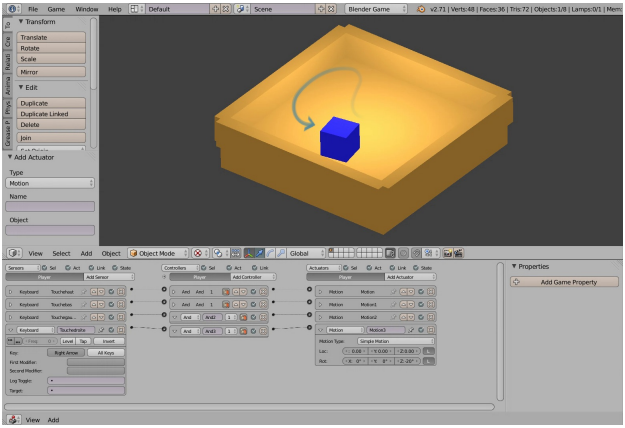



*Une fois réduites, seules les briques qui ont été renommées sont compréhensibles, comme le montre cette capture d'écran de comparaison. Renommer un controller* `And` *n'a pas d'importance pour l'instant. Mais il sera impératif de le faire pour des cas plus complexes où l'entrée du* `And` *sera reliée à plusieurs sensors.*

### Faire tourner le personnage

Nous sommes maintenant capable d'ajouter deux nouveaux *sensors* `Keyboard` pour récupérer les frappes sur `flèche gauche` et `flèche droite` du clavier et les lier via des *controllers* aux *actuators Motion* qui feront tourner l'objet dans un sens ou dans l'autre. Introduisons une valeur de **0.20°** en$_z$ dans la ligne ***Rot*** pour faire tourner notre objet à gauche ou **-0.20°** en $_z$ pour faire tourner notre objet à droite.

*Notre conseil est de tester à nouveau les ajouts, et cela aussi souvent que possible, en particulier tant que vous n'êtes pas encore très expérimentés. Cela permet de détecter les erreurs au plus vite et de faciliter la recherche en cas de problème, ainsi que de tester le jeu, ce qui reste un élément essentiel.*



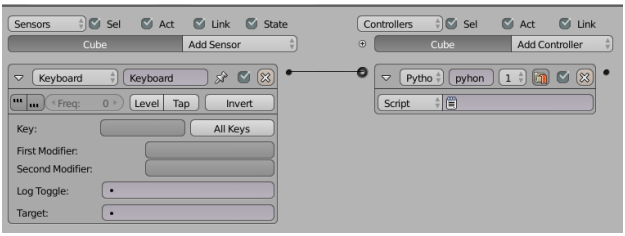
## SCRYPTHON !

Nous allons aborder ici très simplement comment déplacer le personnage, mais cette fois-ci en utilisant un script Python. Il s'agit de notre première utilisation du Python. Coder dans ce cas n'est pas nécessaire, mais cela permet d'offrir d'autres voies d'explorations et d'interaction.

### Python ou briques logiques

De manière générale, toutes les `logic bricks` (briques logiques) peuvent être remplacées par des scripts Python, mais l'inverse n'est pas nécessairement vrai. Les scripts permettent, avec le temps et les connaissances nécessaires, de créer des fonctions de jeu beaucoup plus complexes. Il est donc important de se familiariser très tôt avec leur fonctionnement.

Néanmoins, les scripts Python ont au moins toujours besoin de deux briques logiques pour fonctionner. Un `sensor`, qui déterminera quand activer le script Python et un `controller` de type `Python` qui détermine quel script est à activer à cet instant là. D'autres `sensors` et `actuators` peuvent être présents et l'on peut combiner l'utilisation de scripts et de `logic bricks` dans tout projet, sans restriction. Ce n'est pas parce qu'on décide d'utiliser des scripts Python pour telle ou telle fonction que l'on doit à partir de ce moment-là, tout coder en Python.




### Créer un script Python

Dans le *Text Editor*, créons un nouveau texte que nous appelons **avancer.py**. L'extension ".py" est communément celle des fichiers textes qui représentent du code Python. Pour les fichiers textes inclus dans le .blend, il n'est pas nécessaire d'ajouter une extension, mais, de manière générale, c'est une bonne pratique. Comme ça, sans ouvrir le fichier, nous avons une idée de ce qu'il contient.

### Charger un module

La première ligne que nous devons (et devrons) toujours écrire, c'est celle qui détermine quel module Python nous devons importer. Il existe beaucoup de modules Python différents. Chaque module regroupe une collection de fonctions et méthodes particulières, regroupées autour d'un sujet. Le module *render*, par exemple, regroupe toutes les méthodes liées au rendu du jeu.

Pour faire avancer notre personnage, nous devons importer le module *logic*. Ce module contient entre autre les fonctions nécessaires pour manipuler des objets dans le *Game Engine*. Voici comment cela s'écrit en Python :

```
from bge import logic
```

### Récupérer le *controller*

Le script Python ne "sait" pas qui ou quoi l'active. Pour cela, il existe une méthode qui permet de récupérer le *controller* qui l'a lancé. Nous la stockons dans une variable nommée `cont`.

```
cont = logic.getCurrentController()
```

### Appliquer le déplacement

Une fois que nous avons notre *controller*. Nous pouvons savoir quel est l'objet auquel il est attaché, ou, en d'autres termes, quel est son propriétaire. Dans ce cas-ci, comme nos *logic bricks* sont attachées à notre objet personnage, c'est bien lui que nous récupérons.

```
personnage = cont.owner
```

Nous pouvons donc lui appliquer un déplacement de 0.10 en coordonnée locale y, comme nous le faisions avec le premier *actuator* de notre exemple.

```
personnage.localPosition.y += 0.10
```

En pratique, ce que nous faisons ici, c'est ajouter 0.10 à la coordonnée y de notre objet.

### Activer le script à chaque frappe du clavier

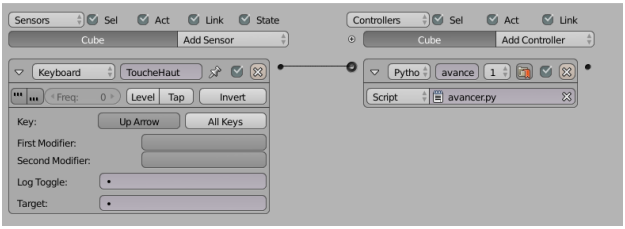
Voici notre script complet :

```
from bge import logic

# Recuperer le controller qui active le script
cont = logic.getCurrentController()

# Recuperer l'objet attache personnage = cont.owner #
Deplacer l'objet vers l'avant personnage.localPosition.y += 0.1
```

Il doit bien porter le nom "avancer.py" dans le *Text Editor*.

Dans le *Logic Editor*, nous avons aussi un *sensor Keyboard* configuré pour répondre lorsque l'on presse la touche `flèche vers le haut` et ce *sensor* est relié à un *controller Python*.

Dans notre *controller* *Python*, devant le menu déroulant qui affiche par défaut *Script*, cliquons dans la boîte vide et nous voyons une liste des scripts et autres textes disponibles dans notre projet Blender. Choisissons "avancer.py".



En lançant maintenant le jeu (`p`), s'il n'y a pas de faute de frappe et que tout est lié correctement, le personnage devrait faire un micro-déplacement vers l'avant à chaque appui sur la touche `flèche vers le haut`.

## DES AMÉLIORATIONS PROGRESSIVES

Tout travail sur des mécaniques de jeu (que ce soit des jeux vidéo ou d'autres genre de jeux comme des jeux de société) demande de longues heures de tests et d'essais. Nous ne trouverons peut-être pas tout de suite les bonnes valeurs de rotation ou de déplacement. L'ajout de fonctions et de paramètres d'un côté ou d'un autre du jeu demande souvent de revoir des choses définies plus tôt. Un jeu est un assemblage subtil de règles et de paramètres pour trouver un équilibre agréable ou parfois même juste utilisable. Il est donc recommandé, après chaque changement de paramètre ou ajout de fonction de tester son jeu. Vous gardez ainsi le fil, en comprenant plus facilement quelle option a cassé ce qui marchait avant ou de comprendre comment tel paramètre influe sur la jouabilité du projet.

# 7. COLLISIONS

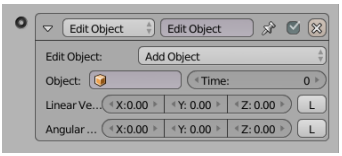
Un espace 3D virtuel est constitué de points, lignes et surfaces qui n'ont pas d'existence physique. Des surfaces peuvent donc s'interpénétrer sans autre conséquence, et notre univers 3D peut sembler être, sur ce point, passablement irréel, fantomatique.

Pour permettre l'interaction entre les éléments du jeu, le moteur de physique intégré de Blender (*Bullet*) est utilisé. Il permet de prendre en compte les volumes et surfaces des objets pour déclencher des événements lorsque ceux-ci entrent en collision (virtuelle), en s'interpénétrant.

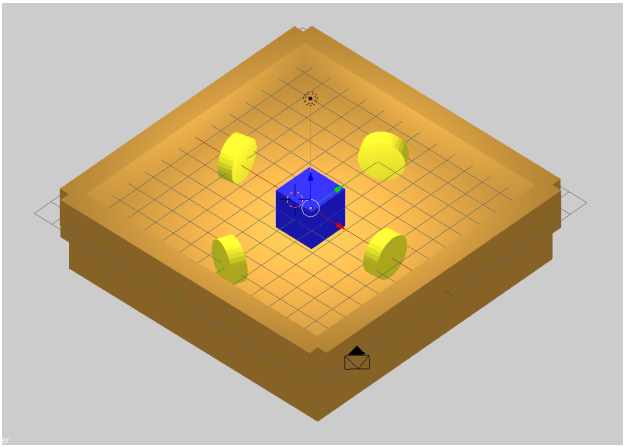
La détection de collision se fait par le *sensor* ***Collision***, très simple d'utilisation. Par défaut, toute collision le déclenchera, mais l'option ***property*** permet de filtrer quels objets déclencheront le *sensor* en se limitant seulement à ceux possédant une propriété du nom choisi (le type de la propriété n'a pas d'importance). Le bouton M/P permet de filtrer les objets collisionnant selon un matériau donné plutôt qu'une propriété. Si un objet possède plusieurs matériaux, il suffit alors que le matériau choisi apparaisse au moins une fois sur l'objet pour qu'il réagisse.


*Il est aussi possible de gérer les objets qui collisionnent avec des groupes de collision tels que décrits dans la section Comportement des objets et personnages au chapitre intitulé Comportement des objets de notre monde.*

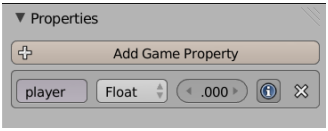
Nous en profiterons pour introduire un nouvel *actuator*, ***Edit Object***. Il permet d'effectuer différentes actions sur l'objet : le détruire, ajouter un nouvel objet, etc. et sera extrêmement utile dans les jeux.




Améliorons notre jeu en ajoutant une fonctionnalité très simple : le joueur doit ramasser des bonus, que nous représenterons par des cylindres aplatis jaunes. Nous allons devoir détecter la collision entre le bonus et le joueur, et faire disparaître le bonus.

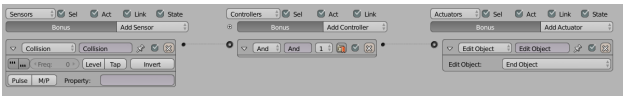


Sélectionnons notre objet joueur et dans le *Logic Editor*, cliquons sur ***Add Game Property***. Nous détaillerons son utilisation dans le chapitre **Compter et afficher le score**. Pour le moment elle sert simplement d'étiquette pour l'*actuator* ***Collision*** : préoccupons-nous donc simplement de lui donner le nom ***player*** (joueur).



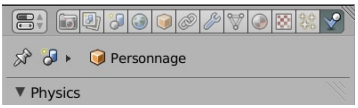

Dans le *Logic Editor* ajoutons cette configuration de *Logic Brick* sur nos objets bonus :

1. un *sensor* de type ***Collision***, avec dans le champ ***Property***, la valeur qui identifie le joueur (***player***) ;
2. lié à un *controller* ***And*** ;
3. lié à un *actuator* de type ***Edit Object***, réglé via sa liste déroulante pour être en mode ***End Object***.

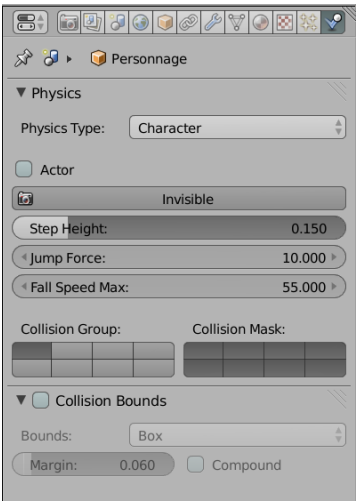



Pour aller vite, faisons l'opération sur un objet, sélectionnons tous les bonus et utilisons le menu `Object > Game > Copy Logic Bricks` pour recopier les briques existantes sur tous les objets sélectionnés.

En appuyant sur `P`et en déplaçant le joueur sur le bonus, nous constatons que ça ne fonctionne pas. C'est normal, la détection des collisions étant une fonction coûteuse en ressource, elle n'est pas activée par défaut, par souci d'optimisation. Cette option est à activer uniquement quand cela est nécessaire.

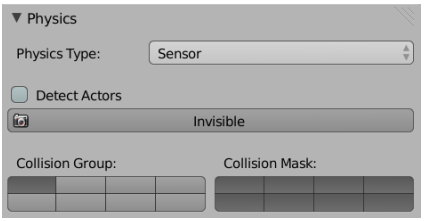



Sélectionnons notre objet joueur et rendons-nous dans l'onglet ***Physics***et son panneau***Physics***. Assurez-vous de bien avoir sélectionné le mode ***Blender Game*** sinon le panneau *Physics* n'affichera pas les mêmes options que dans la copie d'écran ci-dessous. Dans `Physics Type`, sélectionnons ***Character***, qui convient très bien pour un personnage. Cette fois, en appuyant sur **P**, le joueur est capable de ramasser le bonus !
On remarquera également quelque chose d'important, le joueur est à présent soumis à la gravité (il tombe) et aux collisions avec les objets statiques (il ne passe pas à travers le sol et les murs de la salle).

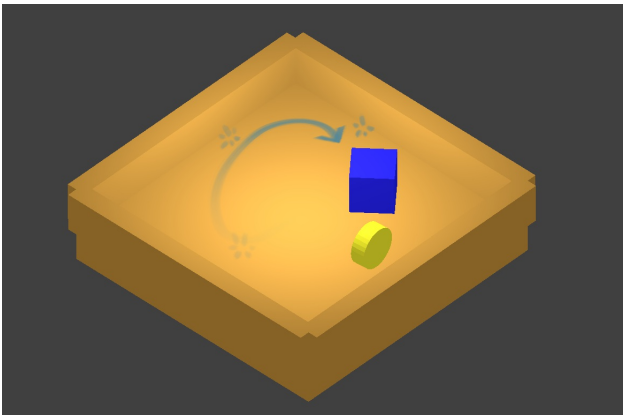



A l'inverse des murs, les bonus ne doivent pas empêcher le joueur d'avancer. Néanmoins, ils doivent être capables de détecter les collisions (pour déclencher une action). Il existe un type de physique dédié à ce cas, le type ***Sensor***.

Sélectionnons donc un objets bonus et dans le panneau *Physics*, et dans *Physics Type*, choisissons ***Sensor***.
Utilisons le menu *Object > Game > Copy Physics Properties* pour recopier ce *Physics Type* sur tous les autres objets bonus.




À présent, le bonus disparaît bien lorsque le joueur le touche ! C'est un bon début, mais un tel bonus n'est pas franchement motivant. Dans le chapitre suivant, nous verrons comment récompenser le joueur en ajoutant des points à son score.



## SCRYPTHON !

Il est parfois plus utile et plus facile de mettre en place du Python pour remplacer un *controller* ou un *actuator*. Il peut par contre être intéressant d'avoir un *sensor* simple en brique logique auquel on ajoutera un *controller* Python qui fera des traitements plus fins que ce qu'aurait pu faire un *sensor* seul .

Voici un exemple de code qui fonctionne ainsi. Pour le tester, vous pouvez créer un *controller* Python que vous pourrez accrocher au *sensor Collision* de l'objet bonus.

```
from bge import logic

# Recupere le controller
cont = logic.getCurrentController()

# Recupere le premier sensor attache au controller
sensor = cont.sensors[0]

# Si le Player se trouve dans la liste des objets entrés en collision
if "Player" in [obj.name for obj in sensor.hitObjectList]:
    # Destruction de l'objet bonus
    cont.owner.endObject()
```

Nous commençons par importer le module `logic` du Blender *Game Engine*. Nous récupérons ensuite le *controller* Python courant que nous plaçons dans une variable `cont`. Tous les *controllers* permettent de récupérer la liste des *sensors* qui leurs sont rattachés grâce à leur attribut *sensors*.

```
sensor = cont.sensors[0]
```

Par cette commande, nous récupérons le premier *sensor*, dans notre cas le seul des *sensors,* celui qui détecte les collisions.

La condition qui suit est un peu plus technique au niveau Python.

```
if "Player" in [obj.name for obj in sensor.hitObjectList]:
    co.owner.endObject()
```

Nous parcourons la totalité des éléments de **`sensor.hitObjectList`**. Cette liste contient tous les éléments qui sont en collisions avec l'objet auquel est rattaché le *sensor*. À partir de cette liste, nous construisons une liste ne contenant que les noms des objets (les noms étant stockés dans l'attribut **`name`** des objets). Nous testons ensuite grâce au mot-clé **`in`** si un des objets a pour nom *`Player`*. Si c'est le cas nous effectuerons alors le code qui se trouve dans le **`if,`** à savoir, lancer la fonction **`endObject()`** de l'objet auquel est attaché le *sensor Collison*. C'est-à-dire que nous allons le faire disparaître.

*L'utilisation des Lists Comprehension (compréhension de liste) est à la fois courante et recommandée en Python. Si vous n'avez pas l'habitude de les utiliser, voici une version plus explicite.*

```
# Nous créons une variable temporaire
del_object = False

# Nous bouclons sur la liste des objets en collision
for obj in sensor.hitObjectList:
    # Si un objet porte le nom Player
    if obj.name == "Player" :
```

```
        # La variable temporaire est changée
        del_object = True
        # Et on peut quitter la boucle
        break

# Si la variable temporaire a été changée
if del_object:
    # Destruction de l'objet bonus
    cont.owner.endObject()
```

### Gestion des listes, les spécificités de Blender

Un bon nombre des listes d'objets, que va vous fournir Blender, ne sont pas des listes classiques Python. Ce sont des listes, mais légèrement modifiées par Blender. On les appelle des **`CListValue`**. On peut, avec une **`CListValue`**, récupérer un objet par son index (**liste_test[0]**) mais aussi en utilisant son nom(**list_test["Player"]**). En utilisant cette spécificité Blender, notre code d'exemple devient beaucoup plus court.

```
from bge import logic

cont = logic.getCurrentController()

sensor = cont.sensors[0]

if "Player" in sensor.hitObjectList:
    cont.owner.endObject()
```

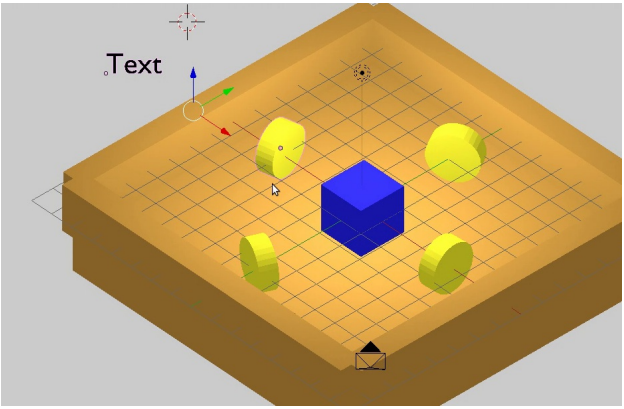
# 8. COMPTER ET AFFICHER LE SCORE

Que serait un jeu sans les systèmes de comptage ou les scores ? Il s'agit là d'éléments importants dans les mécaniques de jeu, le joueur pouvant même décider de jouer en valorisant plus tel ou tel aspect. Globalement, l'affichage d'informations textuelles sur l'écran sera essentiel, et le cas échéant, ces informations devront être calculées. Dans ce chapitre nous verrons comment ajouter un score à notre jeu et l'afficher.

## CRÉER UN OBJET TEXTE POUR STOCKER ET AFFICHER LE SCORE

Nous allons commencer par créer un objet de type *Text*, qui servira à la fois à stocker la valeur du score et à l'afficher à l'écran.

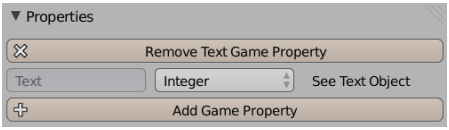
Dans la vue 3D, créons un objet de type *Text* (`Maj + A` > `Text`). Dans les propriétés de l'objet en bas à gauche, cochons ***Align to View***.




Nommons l'objet ***score*** (dans l'onglet *Object*). N'oublions pas de le placer dans une zone visible par la caméra, et de lui donner les dimensions de notre choix.

Pour l'instant, le texte est statique, il ne pourra jamais afficher autre chose que 0. Nous allons le rendre accessible depuis le *BGE*.

Vérifions que l'objet texte est bien sélectionné et, dans le *Logic Editor*, cliquons sur le champ ***Add Text Game Property***. Notre score sera un nombre entier, choisissons ***Integer*** à la place de `String`.Lorsque nous déclenchons le jeu, le texte change pour`0quand on lance le jeu avec P`.




Les ***Properties*** permettent de stocker différents types d'informations que vous pourrez manipuler plus tard dans votre jeu. (Quel est le score du joueur ? Quel est son nom ? *etc.*). À ce titre, elles ressemblent beaucoup à des variables existant dans de nombreux langages de programmation. Elles peuvent être de différents types :

- `Boolean` : Vrai ou Faux (par exemple le joueur est-il actif (Vrai) ou inactif (Faux) ?).
- `Integer`: un nombre entier (par exemple quel est le score ? 1000.).
- `Float`: un nombre à virgule (par exemple combien lui reste-t-il de vie ? 8,5.).
- `String`: une chaîne de caractères (par exemple quel est le nom du joueur ? Suzanne.).
- `Timer`: un compteur de temps (par exemple depuis quand la partie a commencé ? 10 secondes.).

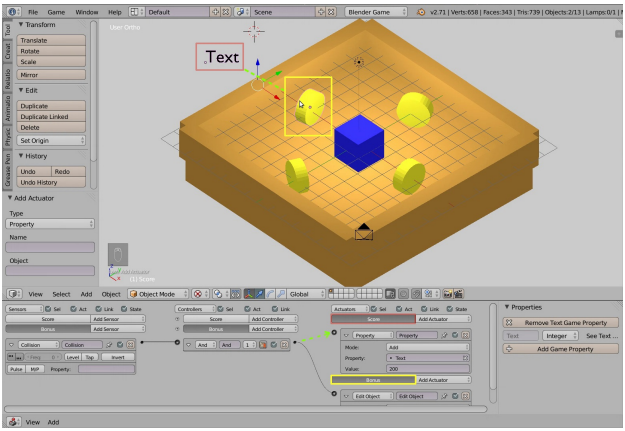
## AUGMENTER LE SCORE LORSQUE LE JOUEUR RAMASSE UN BONUS

Dans le chapitre précédent, nous avons vu comment détecter une collision entre le joueur et un bonus. Cette collision déclenche la disparition du bonus. Nous aimerions qu'elle déclenche également l'augmentation du score.

1. Ajoutons à l'objet score un *actuator* de type ***Property***, associé à la *property* `Text`.

2. Sélectionnons à la fois notre objet `Score` et notre objet `Bonus`en maintenant enfoncée la touche **`Maj`** durant la sélection.

3. Dans le *Logic Editor*, nous voyons à présent à la fois les briques appartenant aux deux objets, nous pouvons donc relier le *controller* `And`(déclenché par la collision détectée par`Bonus`), à l'*actuator*`Property` de l'objet `Score`.

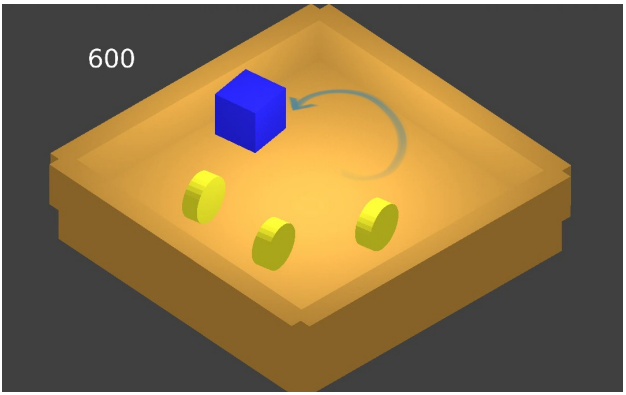



L'*actuator* `Property`permet d'agir sur la valeur de `Text` :

- `Assign` lui assigne une valeur fixée ;
- `Add` lui ajoute une valeur (un nombre négatif peut également être ajouté afin d'opérer une soustraction) ;
- `Copy` copie la valeur d'une autre *property* ;
- `Switch` la bascule, dans le cas d'une valeur booléenne (comme un interrupteur : *on* et *off*, Vrai et Faux, 0 et 1).

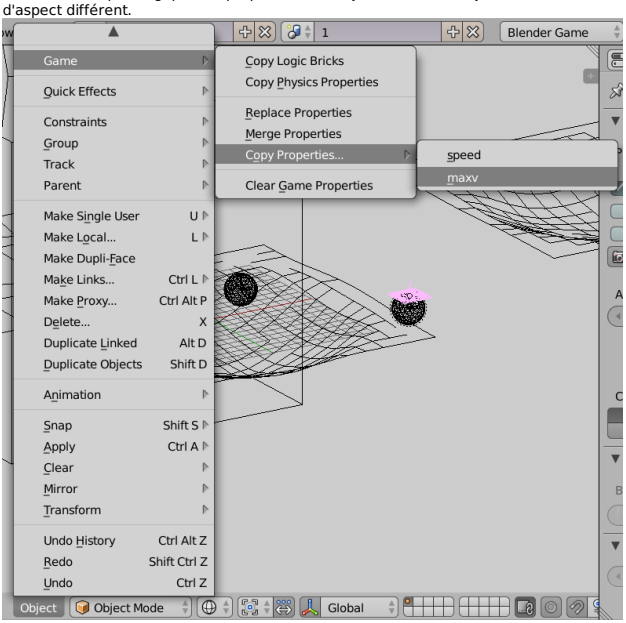
4. Dans notre cas, nous voulons ajouter des points au *score*, donc choisissons `Add`, et pour `Value`, le nombre de points que nous souhaitons attribuer au joueur pour cette réussite.

À présent, lorsque le personnage ramasse le bonus, le score est bien augmenté ! Il est maintenant aisé de dupliquer le bonus paramétré à l'aide de la combinaison de touche **`Maj+D`** afin d'en proposer plusieurs dans la même scène.




*Sur cette image, notre cube personnage a fait disparaître trois bonus placés sur sa course. Le score est passé à 600 points, soit trois fois 200 points par bonus.*

Lorsque vos projets deviendront plus complexes, utiliser le menu`Object` et le sous-menu `Game` vous permettra de copier facilement briques logiques et propriétés de l'objet actif à des objets d'aspect différent.




## SCRYPTHON !

Agir sur le texte sera une action courante dans le jeu. Les objets de type texte (**`KX_FontObject`**) ont un attribut **`text`** (qui correspond à leur *Game Property* `Text`). Il est possible d'utiliser cet attribut pour modifier à la volée ses valeurs sous la forme **`owner.text`** (ou **`owner["Text"]`**).

```
from bge import logic

cont = logic.getCurrentController()

own = cont.owner own.text = "10"
```

Remarquez que même si la propriété est définie comme *Integer*, il faudra respecter le fait que Python gère la valeur comme une chaîne et faire les conversions qui s'imposent.

```
from bge import logic cont = logic.getCurrentController() own = cont.owner own.text =
str(int(own.text)+10)
```

### Comment accéder à des *Game Property* en Python ?

Nous pourrions tout d'abord vouloir changer la valeur d'une *Game Property*, voici un exemple de code pour le faire :

```
from bge import logic

cont = logic.getCurrentController()

own = cont.owner

own["Text"] = "Hello World!"
```

Nous pourrions aussi vouloir récupérer la valeur d'une *Game Property*. Dans ce cas-là, une façon évidente de faire est d'utiliser la même syntaxe.

```
from bge import logic

cont = logic.getCurrentController()

own = cont.owner

# Affiche en console la valeur de la property "Text"
print(own["Text"])
```

Cette manière de faire fonctionne parfaitement sauf dans un cas, celui où la *Game Property* n'existe pas. Dans ce cas-là, Python va nous renvoyer une erreur sous la forme d'une exception de type ***KeyError***. Dans le cas où vous n'êtes pas sûr de l'existence de votre *Game Property*, il faut donc utiliser la fonction **`get()`** qui permet en plus de définir une valeur à renvoyer si justement la *Game Property* n'existe pas.

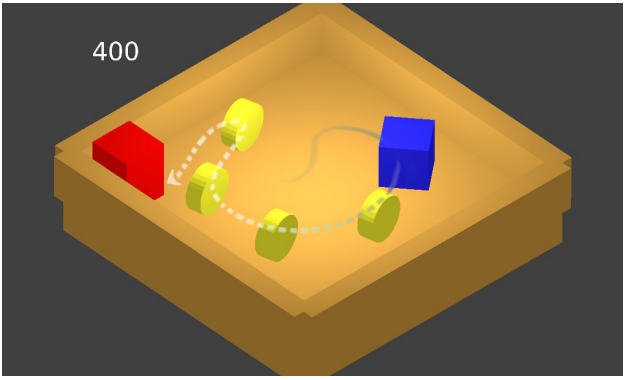
Notre code devient alors :

```
from bge import logic

cont = logic.getCurrentController()

own = cont.owner

print(own.get("Text", "Sans Game Property"))
```

Le **`print`** affichera soit la valeur de la *Game Property* appelée `"Text"`, soit `"Sans Game Property"`.

# 9. QUITTER LE JEU

Une fois notre objectif atteint (ou l'envie de faire une pause) nous voulons pouvoir quitter le jeu.

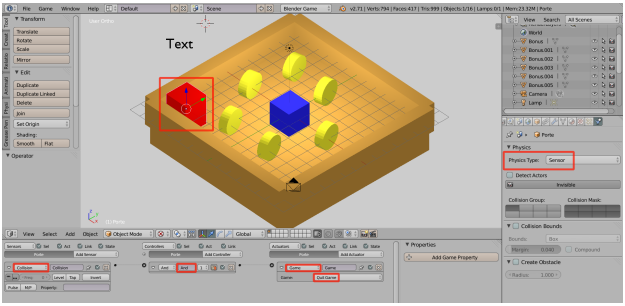



Dans le cadre de notre mini-jeu, nous voulons que notre personnage quitte le jeu après avoir atteint la porte. L'objet porte est pour l'instant un simple cube rouge. Pour procéder au paramétrage de cette porte, nous allons, comme vu dans le chapitre précédent, le gérer par une simple collision. Seul l'événement de fin changera pour terminer la partie.

Sélectionnons notre cube rouge renommé "Porte" et ajoutons lui les *Logic Brick* suivantes :

- le ***Sensor*** de type ***Collision ;***
- le ***Controller*** de type ***And ;***
- l' ***Actuator*** de type ***Game***, paramétré pour exécuter un ***Quit Game***.

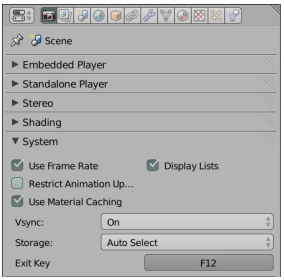
Il ne faudra pas oublier de changer le ***Physics Type*** de l'objet pour ***Sensor***.




Voila, nous avons notre fin du jeu ! Le procédé est similaire pour le cas où la fin du jeu serait déclenchée par une collision avec un ennemi. On peut également proposer de relancer le jeu en changeant l'*actuator Game* en lui mettant l'option *Restart Game*.

## QUITTER À L'AIDE D'UNE TOUCHE DE CLAVIER

Par défaut, la touche permettant de quitter est **Échap**. Mais si par exemple, nous préférons afficher un menu de pause plutôt que de quitter le jeu abruptement avec cette touche, pour sortir du jeu à tout moment, nous pouvons redéfinir une autre touche (comme `F12`) en changeant la valeur *Exit Key* dans *Properties* > *Render* > *System*.



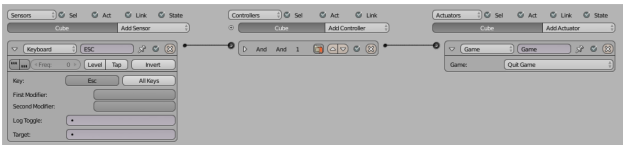

Maintenant que nous avons redéfini la touche par défaut, nous allons assigner une nouvelle touche via les *Logic Brick*.

1. Nous allons donc commencer par ajouter une brique *Sensor* de type *Keyboard* qui sera la touche à actionner pour quitter le jeu.

2. Relions cette brique à une brique *Controller* de type *And* qui sera elle même branchée sur une brique *Actuator* de type *Game*.

3. Ensuite, il nous reste juste à changer l'action à effectuer dans la brique *Game* pour mettre *Quit Game*.

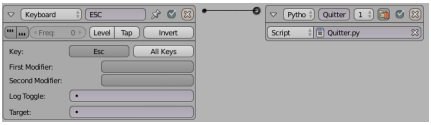
Cet *actuator* pourrait également lancer une scène plutôt que de quitter, par exemple pour afficher un menu de pause. Ceci sera discuté dans le chapitre **Créer un Menu** dans la section **Développer l'ergonomie**.



## SCRYPTHON!

Il nous suffit juste de faire appel au code suivant avec une brique *Controller* de type Python :

```
from bge import logic

logic.endGame()
```



Nous avons juste un petit morceau de code, mais nous pouvons aller bien plus loin en gérant avec Python l'événement qui fera quitter le jeu.

```
from bge import logic, events
# On recupere l'objet qui lance le script
cont = logic.getCurrentController() own = cont.owner

# On assigne les evenements du clavier à une variable kboard = logic.keyboard.events
# Si la touche echap est appuyee if kboard[events.ESCKEY] == logic.KX_INPUT_JUST_ACTIVATED :
 # Quitter le jeu logic.endGame()
 # Si le score est de minimum 5 if own['score'] >= 5 :
 # Quitter le jeu logic.endGame()
```

Il ne nous reste plus qu'à remplacer la brique *sensor* qui lançait le script par une brique *sensor Always* pour laquelle on enclenche le *Pulse mode* (bouton `"..."` de gauche) afin que le script soit exécuté en permanence.


Le *Pulse mode* sert à envoyer un signal à chaque *image (frame)* pour que le script soit toujours exécuté.

# 10. REMPLACER LES PRIMITIVES PAR DES RESSOURCES

Dans le chapitre précédent, nous avons obtenu un jeu fonctionnel, mais il faut reconnaître qu'il n'est pas très joli. La prochaine étape sera donc de remplacer nos objets temporaires, par des objets que nous aurons modélisés et texturés, ou téléchargés sur un site de ressources libres, comme BlendSwap.com ou OpenGameArt.org. Attention de bien choisir des objets adaptés au temps réel (voir la section précédente, chapitre Spécificités du *Game Engine*).

Il est conseillé de créer chaque objet dans un fichier .blend distinct, ainsi il sera facile de se constituer progressivement une bibliothèque d'objets (on les appelle souvent *assets*), réutilisable dans de futurs projets.

Pour ce chapitre, nous pourrons utiliser les ressources qui sont associées à ce manuel.

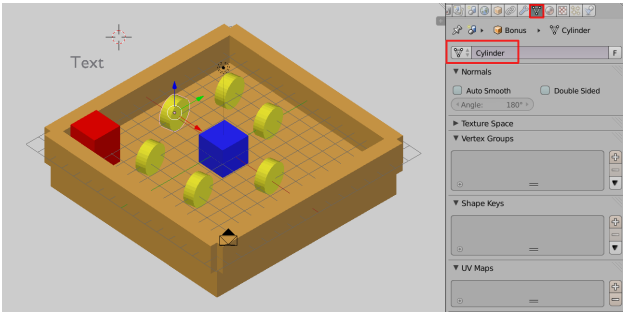
Selon le type d'objet et ses conditions d'utilisation, différentes techniques s'offrent à nous, chacune ayant ses avantages et inconvénients.

## REMPLACEMENT MANUEL DU *MESH* (MAILLAGE)

C'est sans doute la méthode la plus simple et la plus rapide. Elle conviendra parfaitement pour des objets simples, comme des éléments de décor. Ici, nous allons remplacer la sphère qui figure un bonus, par un *asset* représentant un bambou.

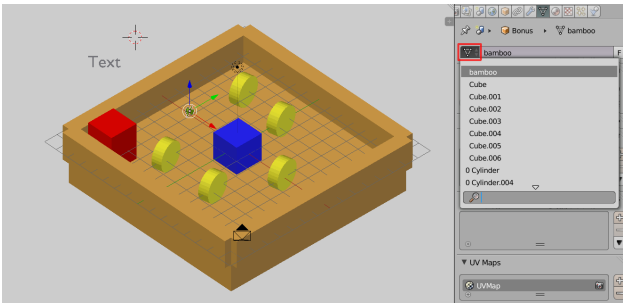
Commençons par importer le *mesh* de l'*asset* dans notre fichier blend : ***File*** > ***Append*** > bambou.blend > ***Mesh*** > Bambou.

Sélectionnons l'objet `Bonus` et rendons-nous dans le panneau ***Mesh***.



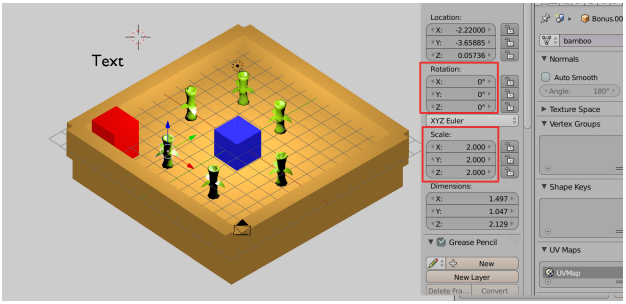

Il suffit de remplacer le nom de notre *mesh* temporaire (`Cylinder`), par le nom du *mesh* de notre *asset* (`bamboo`). Nommer correctement ses *meshs* et ses objets est évidemment indispensable !

Notre *Mesh* importé va hériter de la taille et de la rotation de notre objet précédent, ce qui a de fortes chances de le déformer.

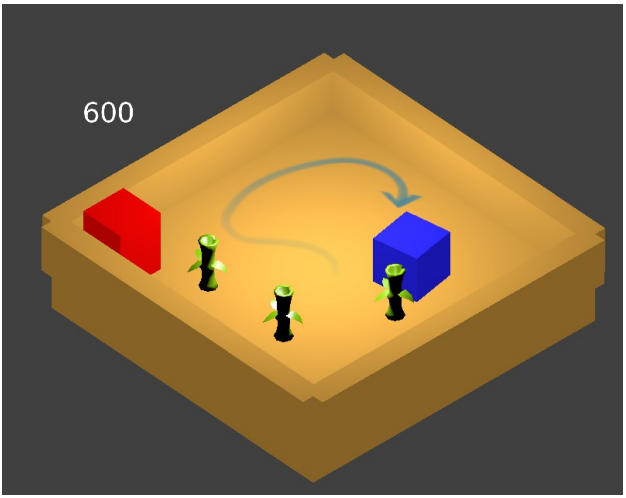



Nous changerons donc pour chaque élément les propriétés de tailles et de rotations des objets pour les adapter à notre scène via le panneau des propriétés de la vue 3D (touche `N`).

*Nous pourrions aussi appliquer les transformations réalisées sur le maillage de base avant de le remplacer. Pour cela, nous utiliserions `Ctrl + A` puis choisirions l'option à appliquer.*




Dans le jeu, nos bonus ont à présent l'apparence de bambous.




**Faites des économies d'échelle!**

En temps réel, la gestion des ressources est cruciale : elle fait la différence entre un jeu fluide et un jeu saccadé (ou même entre un jeu qui fonctionne, et un jeu qui plante au démarrage).

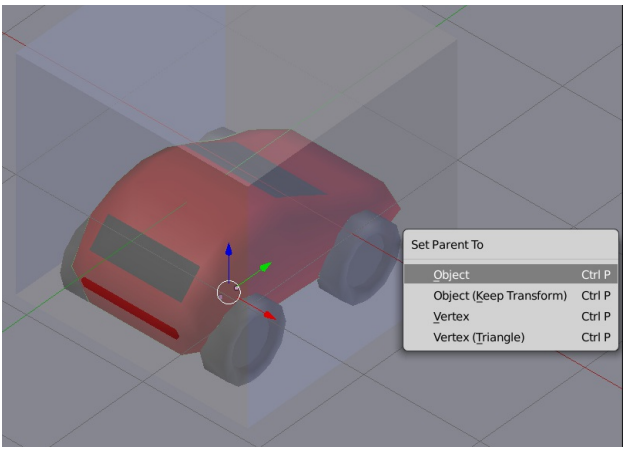
Il est important de comprendre que si deux objets partagent le même *mesh* (maillage), ce dernier ne sera chargé qu'une seule fois en mémoire. Il est donc très avantageux de réutiliser autant que possible les mêmes *mesh*. Ce **principe fondamental** s'applique également pour les textures, les sons, etc.

## PARENTER UNE RESSOURCE

Lorsque l'*asset* est plus complexe (typiquement, un groupe composé de plusieurs objets, comprenant des animations, *etc.*), la première méthode ne sera pas suffisante. Par contre, on peut attacher (parenter) notre *asset* à notre objet basique et masquer ce dernier. Ici, nous allons "remplacer" le cube représentant le joueur, par un *asset* plus sympathique.

Importons le Group contenant notre *asset*:`File` > `Append` >(...)`voiture.blend` > `Group` > `Bambou`.

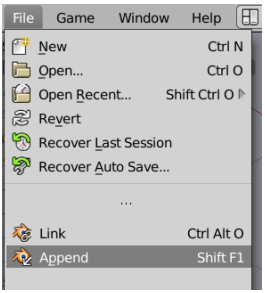
Positionnons-le de manière à ce qu'il s'inscrive dans notre cube, puis parentons-le (sélectionner le groupe, puis le cube, `CTRL + P` > `Set parent to Object`).




Nous avons toujours besoin de notre cube (il contient les briques logiques et gère les collisions), mais nous ne souhaitons pas le voir s'afficher : dans l'onglet ***Physics***, cochons ***Invisible***.
À l'inverse, nous souhaitons voir l'*asset*, mais nous ne souhaitons pas qu'il gère les collisions : on vérifiera donc que ***No Collision*** est bien coché dans le panneau ***Physics*** de chaque objet du groupe.

### Gestion des ressources

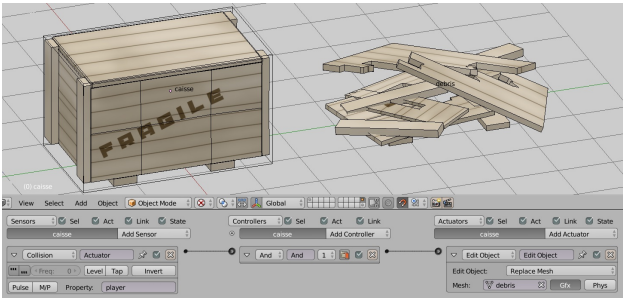
Pour des projets plus complexes, on préférera **lier** les *assets* plutôt que les **importer** (utiliser ***Link*** plutôt que ***Append***). Ainsi, une modification de l'*asset* sera automatiquement répercutée à l'ensemble du projet. Cela facilite également le travail en équipe. Dans ce cas, il convient de placer les *assets* dans un sous-dossier de votre projet, et d'utiliser des chemins relatifs, comme pour un projet Blender classique.




## REMPLACEMENT DYNAMIQUE AVEC LA BRIQUE LOGIQUE *REPLACE MESH*

Parfois, il est utile de remplacer dynamiquement un *asset* au cours du jeu ; par exemple, une caisse en bois sera remplacée par des débris si le joueur la percute.

1. Importons les deux objets représentant la caisse et les débris : ***File*** > ***Append*** > (...)caisse.blend > ***Object*** > Caisse Debris;
2. La caisse a besoin de pouvoir détecter les collisions, nous allons donc lui donner une physique simple : dans l'onglet ***Physics*** > ***Physic Type***, sélectionnons ***Dynamics*** ;
3. La partie logique sera très simple également, avec trois briques logiques :
   1. u n *sensor* ***Collision***, qui détecte la *property* *Player* (vérifier que notre objet joueur possède bien cette *property*)
   2. lié à un *controller* ***And***
   3. lié à un *actuator* ***Edit Object***
4. L'***actuator*** ***Edit Object*** permet différentes actions, ici c'est ***Replace Mesh*** qui nous intéresse. Dans le champ ***Mesh***, entrons le nom du ***mesh*** qui représente les débris : Debris;
5. L'objet qui contient le ***mesh*** Debrisdoit exister dans notre scène pour que le ***Replace Mesh*** fonctionne. Néanmoins, on ne souhaite pas qu'il soit réellement présent au début du jeu. Il suffit de le déplacer dans un calque inactif ;
6. Lançons le jeu et dirigeons le joueur vers la caisse : elle se brise en morceaux.



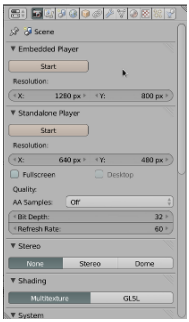

L'effet mériterait bien entendu d'être amélioré, par exemple en projetant quelques débris (section Comportement des objets et personnages au chapitre Génération d'objets) qui rebondiront dans le décor et en agissant sur le joueur (section **Comportement des objets et personnages** chapitre **Comportement des objets dans notre monde**).

# 11. PARTAGER SON JEU

Une fois le projet finalisé, nous souhaitons le partager. Blender permet d'exporter les scènes de manière à les lire sans lancer Blender. Le logiciel **Blenderplayer** permet de lire les fichiers .blend comme des séquences interactives autonomes afin de les distribuer.

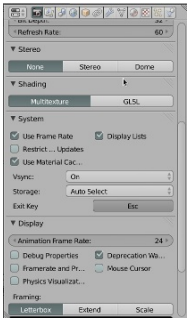
## CONFIGURATION DE L'AFFICHAGE DU JEU

Blender permet depuis l'interface de tester le comportement du jeu. Dans le panneau `Render`, lorsqu'on est en mode *Game Engine*, les onglets `Embedded Player` et `Standalone Player` apparaissent. Ils permettent de tester le jeu en le lançant dans l'interface de Blender pour `Embedded Player` ou dans une fenêtre séparée avec `Standalone Player`.




Les paramètres de cette fenêtre permettent de configurer le jeu pour son export :

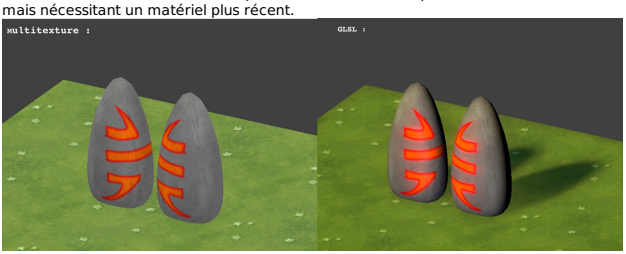
- ***Resolution*** : permet d'adapter la taille de l'image du jeu.
- ***Fullscreen*** : lance le jeu en plein écran. Lorsque cette case est cochée, l'option *Desktop* devient disponible. Si cette option est cochée, le jeu sera lancé dans la résolution actuelle de l'écran, outrepassant le réglage *Resolution*.
- ***Quality*** : *AA Samples* : permet de configurer l'anti-crénelage. Par contre, il faut faire attention car cette option nécessite plus de ressources graphiques.
- ***Bit Depth*** : règle le nombre de couleurs. Généralement, on garde `32 bit` pour une qualité des couleurs idéale sur les machine actuelles.
- ***Refresh Rate*** : nombre maximum d'images par seconde. La plupart des écrans d'ordinateurs actuels fonctionnent à 60 Hz, qui est l'option par défaut.



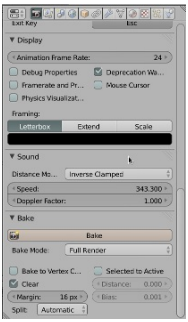

L'onglet ***Stereo*** : permet de faire un rendu stéréoscopique (3D) suivant plusieurs procédés (par lunettes teintées, côte-à-côte) ou d'activer le rendu anamorphique en *Dome*, permettant de projeter l'image sur un dôme (ex. planétarium) ou éventuellement d'autres géométries...

L'onget ***Shading*** :

- L'option `Multitexture` permet de faire un rendu simplifié, adapté à des cartes graphiques ne supportant pas OpenGL 2.0, par exemple anciennes ou dédiées aux *SOC** (systèmes embarqués), comme les smartphones et les tablettes.
  *Le BGE ne permet pas actuellement d'exporter directement pour les plate-formes ne supportant qu'OpenGLES 2.0 comme Android ou iOS. Par contre, on peut tout à fait utiliser Blender et le BGE pour concevoir des ressources exportables vers des moteurs dédiés à ces plateformes.*
- L'option **GLSL** active le rendu OpenGL2.0 avec les effets de texture et de lumière avancés, permettant un rendu optimal mais nécessitant un matériel plus récent.




- L'onglet ***System*** : `Use Frame Rate`, utilise la fréquence de rafraîchissement actuellement utilisée par l'écran.
- ***Restrict Animation Update*** : contraint le rafraîchissement des animations à la fréquence de rafraîchissement de l'écran. Cette option permet d'optimiser certaines performances en réduisant le calcul des animations, mais peut provoquer des incohérences dans certains mouvements rapides ou complexes.
- ***Use Material Caching*** : option d'optimisation permettant de minimiser les lectures disque des textures.
- ***Vsync***: synchronise le rendu au rafraîchissement de l'écran (l'option `Adaptive` est recommandée si vous ne connaissez pas).
- ***Storage Mode*** : option d'optimisation graphique. Le mode par défaut (*auto-select*) n'est à changer qu'en cas de problème.
- ***Exit key*** : définition de la touche permettant de sortir du jeu.

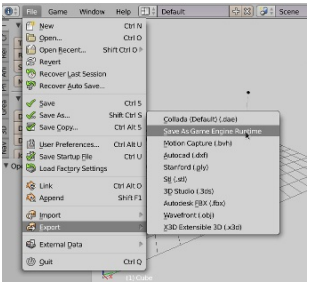



- L'onglet ***Display*** concerne le rendu dans l'interface d'édition, et ses paramètres n'ont pas d'influence lors de l'export d'un jeu en dehors du paramètre `Animation Frame Rate` que nous aborderons dans la section **Comportement des objets et personnages** dans le chapitre **Animer des personnages** dans le *Game Engine*. Cet onglet comporte donc des options de débogage, permettant de visualiser les paramètres internes du jeu, comme les propriétés des objets, la fréquence de rafraîchissement de l'image réelle, les contraintes physiques, les avertissements de dépréciation pour le code Python (pour ne pas utiliser des fonctions Python anciennes et non supportées) et l'affichage du curseur de la souris.
- Le ***Framing*** permet de configurer la vue camera dans l'éditeur.
- L'onglet ***Sound*** permet de régler la diffusion du son, plus précisément sa simulation physique.
- L'onglet ***Bake*** concerne une optimisation des textures et du rendu, qui sera détaillée dans la section **Optimisations** dans le chapitre **Pré-calcule des textures et de la lumière**.

## PUBLIER UN BINAIRE

Lorsque nous souhaitons exporter une scène, l'entièreté du jeu peut être contenue avec le *Player* dans un même fichier exécutable. De la sorte, tout le jeu est présent dans un seul fichier (accompagné de bibliothèques DLL), et donc plus simple à partager. Sous cette forme, le code et le contenu sont soumis à la licence GPLv3*, et nous sommes dès lors obligé de publier également le fichier .blend de manière séparée.

Pour cela, allons dans le menu ***File > Export > Save as Game Engine Runtime.*** *Cet export est à activer au préalable dans l'onglet* ***addons****, se trouvant dans* ***user settings.***




Nous pouvons également choisir de sauver le contenu sous la forme d'un fichier .blend, et de le lire avec le binaire **Blenderplayer**. De cette manière le contenu artistique n'est pas soumis à la licence GPL (mais reste lisible par ailleurs avec Blender).

Il suffit donc simplement de mettre le fichier .blend dans le même répertoire que l'exécutable **Blenderplayer**.

Pour lancer le jeu, il faut alors lancer **Blenderplayer** en lui spécifiant quel fichier lire.

sur GNU-linux et MacOS :

```
user@desktop:~/mondossier/$: ./blenderplayer monjeu.blend
```

sur Windows, utiliser la ligne de commande :

```
C:\mondossier\> blenderplayer.exe monjeu.blend
```

Pour créer un lanceur il faut créer un raccourci, ou un fichier de script *batch* avec cette commande.

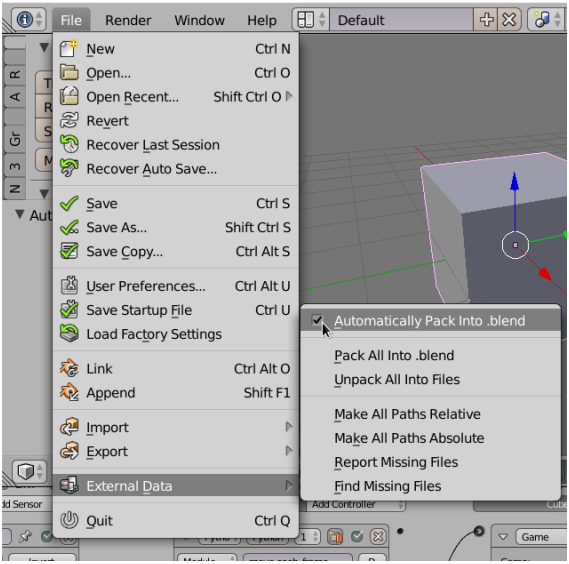
Certains ont développé des scripts permettant de crypter le fichier .blend, de sorte à en protéger le contenu, mais ceci dépasse le cadre de ce manuel.

A noter que si le contenu artistique n'est pas soumis ici à la licence GNU GPL*, le code Python faisant référence à l'API du *Blender Game Engine* reste lui sous cette licence. Il y a lieu donc de publier les scripts, séparément au besoin.

## EMPAQUETER SON CONTENU

Pour être certain de ne rien perdre entre les manipulations de copie et d'installation du jeu, nous pouvons empaqueter le contenu directement dans le .blend (par exemple, textures et sons).

La façon la plus simple est de cocher l'option d'empaquetage automatique. De la sorte les données externes sont intégrées dans le fichier .blend à chaque enregistrement. Sinon, il faut s'assurer que les données soient toujours dans le bon répertoire (répertoire du .blend ou sous-dossier).




Une fois les éléments empaquetés, il est possible de les dés-empaqueter et donc de les retravailler ultérieurement avec l'option `Unpack all files`. L'empaquetage n'est donc pas une mesure de protection du contenu, simplement une méthode facile pour conserver les données accessibles au jeu dans un même fichier.

## EXPORT VERS D'AUTRES MOTEURS DE JEU

Blender permet d'exporter le contenu et parfois (une partie de) la logique. Ceci dépend de la plate-forme ou du moteur de jeu visé, et la plupart du temps on utilise les *plugins* d'exportateurs adaptés, soit pour exporter directement (par exemple, l'animation pour le moteur Unreal - *plugin* à activer dans les préférences utilisateurs) soit en passant par un format intermédiaire (par exemple, .dxf ou .obj pour les objets, collada pour géométrie et animations, *etc.*).

## NOTIONS DE LICENCES

Les créations de l'esprit sont soumises, dans nos contrées, à un régime de propriété intellectuelle. Le droit d'auteur (et *copyright*) s'exerce *a priori* sur toute création et a priori restreint les droits des utilisateurs, lecteurs, spectateurs à une consultation privée. Toute autre utilisation est normalement soumise à autorisation des ayants droits (auteur, héritiers de l'auteur ou éditeur).

Blender, en tant que logiciel libre est distribué avec une licence GNU GPLv3*. Ceci implique que tout code lié au code de Blender est lui-même soumis à cette licence.

La licence GPLv3 est une licence garantissant les libertés des utilisateurs, en obligeant à publier les sources des logiciels, et permettant aux utilisateurs de modifier et d'utiliser ce logiciel comme bon leur semble, à la seule condition que toute modification soit également publiée sous cette licence.

Elle garantit tant la liberté de l'utilisateur que le droit d'auteur du créateur, en lui assurant la citation de son nom et le fait qu'il bénéficiera des modifications ultérieures au programme.

Qui dit publication, dit distribution. Si les créations ne sont pas distribuées, elle ne doivent pas nécessairement divulguer leurs sources (par exemple, dans le cas d'une installation interactive sur une machine spécifique pour un événement ou un musée).

Il y a eu beaucoup de discussions sur ce sujet et souvent beaucoup de confusion, voire même de peur de voir son travail pillé par d'autres parce que sous licence libre. Ceci est souvent le fait de personnes ne comprenant pas les implications ni la protection offerte par la licence GNU GPL.

De manière générale, cette question n'a finalement pas beaucoup d'importance pour l'usage qui est généralement fait du BGE, entre application à usage interne, exemples didactiques et prototypage.

Parce que ce qui compte généralement dans un jeu, c'est le contenu et non le code, certains ont par ailleurs publié des jeux de manière commerciale sans problème, sous ces conditions de licence, sans en souffrir. Tout comme certaines grosses sociétés de jeux vidéo qui ont libéré le code de leur moteur de jeu (Quake Engine par exemple). Les contenus eux peuvent rester protégés pour pouvoir être vendus.

Les contenus peuvent être également libres pour pouvoir être modifiés par d'autres et réutilisés comme Yo Frankie! ou sur http://opengameart.org/.

Ainsi, être libre n'empêche nullement le financement, mais oblige parfois à revoir le modèle économique envisagé.

# DÉVELOPPER L'UNIVERS DU JEU

**12.** LUMIÈRES ET OMBRES
**13.** INTRODUCTION AUX MATÉRIAUX ET TEXTURES
**14.** MATÉRIAUX ET TEXTURES AVANCÉS
**15.** MATÉRIAUX DYNAMIQUES ET TEXTURES ANIMÉES
**16.** AJOUTER DU SON
**17.** FILTRES DE RENDU

# 12. LUMIÈRES ET OMBRES

Maintenant que nous avons les éléments de base de notre jeu, nous pouvons aller plus loin dans la création de l'univers qu'il va faire explorer à nos joueurs. La première chose à prendre en compte est l'éclairage.

Un espace est perceptible grâce à sa géométrie, mais également par les ombres et lumières qui en dessinent les volumes et y apportent une ambiance. Nous allons voir dans ce chapitre comment illuminer notre espace et y dessiner des ombres.

*Nous ne rentrons pas ici en détail dans l'apprentissage de l'éclairage puisqu'il s'agit d'une spécialité en soi qui n'est pas l'objet principal de ce livre.*

## DIFFÉRENTS TYPES D'ÉCLAIRAGE

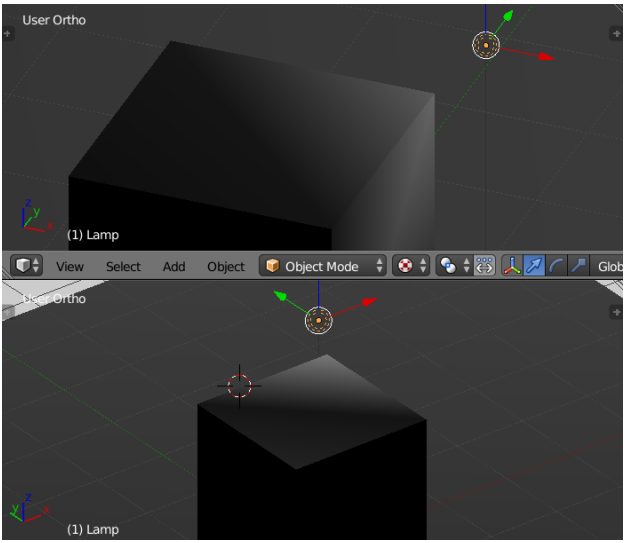
L'éclairage par défaut du mode jeu est basé sur les *vertices* : chaque *vertex* prend une teinte en fonction de sa distance par rapport aux lampes environnantes. La couleur des faces est déduite par interpolation. Cette technique montre ses limites assez rapidement, car l'interpolation est faussée ou rend visibles certaines arrêtes.

Il s'agit du mode d'éclairage le plus simple et historiquement le plus ancien. Il est aussi appelé éclairage Gouraud, du nom de son inventeur.

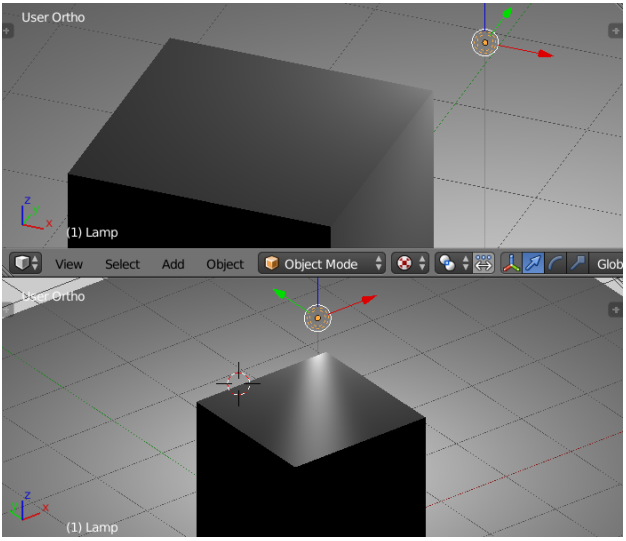
Cette technique ne permettra pas non plus de produire des ombres portées.


Pour ces raisons, il est souvent préférable de changer de technique de *shading*. L'option se trouve dans le panneau ***Render***, sous l'onglet ***Shading***. On peut choisir entre ***Multitexture*** (par défaut) et ***GLSL***. Ce dernier dévoile plus de boutons, pour activer et désactiver certaines options (les lampes, les ombres, etc.).

En mode GLSL, l'éclairage n'est plus calculé à partir des *vertices* mais sur la face toute entière, pixel par pixel. Cela permet de révéler des dégradés plus fins et des reflets spéculaires fidèles à la forme d'origine. Les changements sont aussi bien visibles dans la *3D View* que dans le jeu.
Dans l'exemple ci-contre, nous avons placé une lampe au niveau d'un des angles du cube, (légèrement décalée) et nous avons pris deux points de vue différents de manière à bien voir le reflet. Le résultat est parlant : en **multitexture**, le rendu est extrêmement triangulé et les spéculaires sont faibles, ce qui n'est pas le cas en **GLSL**.

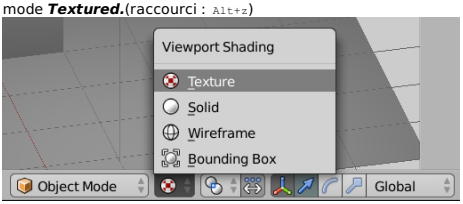



*Multitexture*




*GLSL*

Pour profiter du mode GLSL, il faut penser à configurer la vue 3D en mode ***Textured.***(raccourci : `Alt+z`)

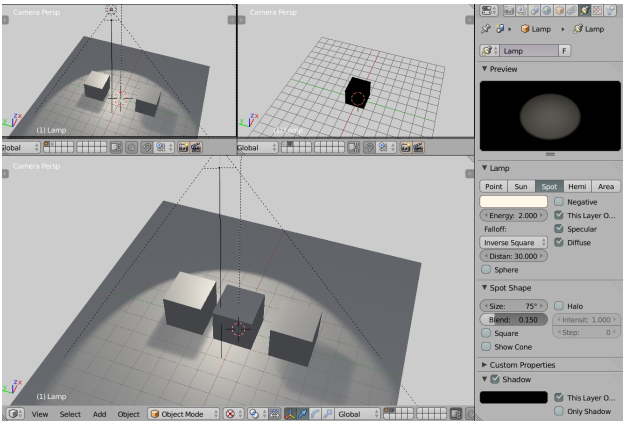



## OPTIONS DES LAMPES

Pour ajouter une lampe, il suffit de passer par le menu ***Add*** (`Maj + A`), sous l'option ***Lamp***. Le seul mode non supporté actuellement est la lampe de type *Area*. Une fois une lampe ajoutée, on peut modifier son type ainsi que diverses options dans le panneau ***Lamp***. Les options disponibles ne présentent pas de différence majeure avec le mode ***Blender Render***, mais nous allons les passer en revue brièvement.

Les options communes à tout type de lampe sont tout d'abord la couleur et l'intensité (***Energy***). ***Negative***permet d'avoir une lampe dont l'éclairage va en diminuant (idéal pour fausser une ombre, mais non physiquement réaliste).

En cochant l'option ***This Layer Only***, une lampe se trouvant dans un calque n'éclaire que les objets présents dans ce même calque. Cette option peut être utile si d'autres *layers* sont activés simultanément : on peut avoir une lampe qui éclaire un objet mais pas les autres.

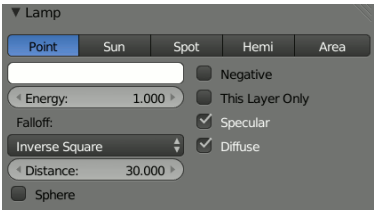



Enfin, ***Specular*** et ***Diffuse*** spécifient si la lampe produit des reflets spéculaires et de la lumière ambiante.

### Lampe *Point*

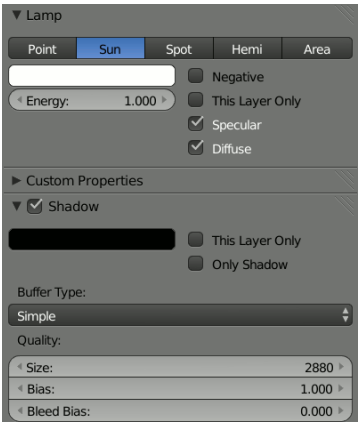
Ce type de lampe est utile pour imiter les sources lumineuses qui diffusent de la lumière dans toutes les directions depuis un point fixe et central, comme une torche ou une bougie, avec une rapide diminution d'intensité. Il n'y pas d'ombre portée avec ce type de lampe.

La manière dont diminue l'intensité peut être changée avec l'option ***Falloff***. L'option par défaut, `Inverse Square`, correspond à la diminution réaliste d'une lampe ponctuelle. L'option ***Distance*** indique la distance à laquelle il faut se situer pour être exposé à la moitié de l'intensité maximale, elle-même définie dans ***Energy***.



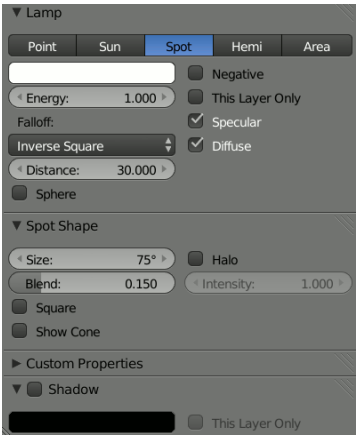

### Lampe *Sun*

Le type *Sun* est une lampe directionnelle, permettant de simuler un point éclairant la scène depuis une très grande distance et avec des rayons parallèles, comme le ferait le soleil. Seule la rotation de l'objet lampe permet de changer la provenance de la lumière, sa position n'est pas prise en compte. L'onglet *Shadow* permet de régler la projection d'ombre. Ce point sera traité spécifiquement dans la suite de ce chapitre. Le calcul de l'ombre est limité à un certain angle, défini par *Frustum*.



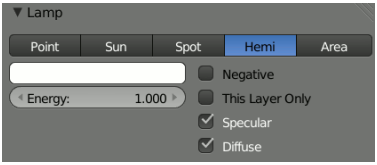

### Lampe *Spot*

Le spot limite l'éclairage à un cône et permet de projeter les ombres portées. La forme du cône est paramétrable sous l'onglet *Spot Shape*. Comme pour le *Sun*, l'option *Shadow* est disponible.



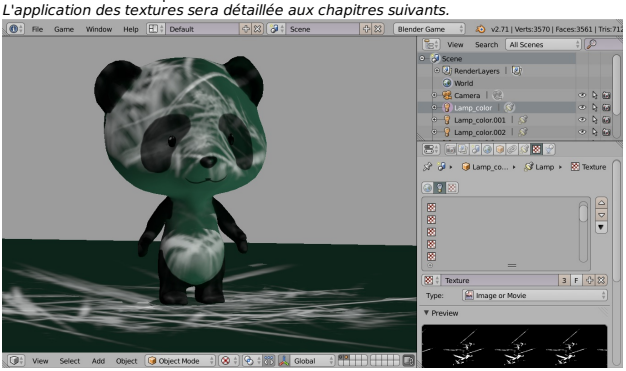
### Lampe *Hemi*

Le type `Hemi` est un autre type de lampe directionnelle, sans possibilité de produire des ombres. Elle est différente de la lampe de type `Sun`, car la lumière provient de tous les côtés d'une demi-sphère englobant la scène. Elle peut par exemple simuler l'éclairage diffus produit par un ciel nuageux.



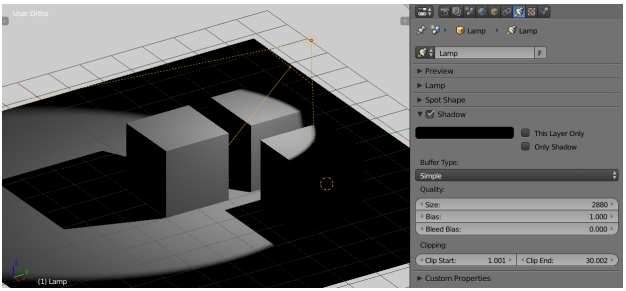

### Texturer une lampe

Comme pour les matériaux, il est possible d'affecter des textures à chaque paramètre en lien avec la lumière. Nous pouvons modifier le *World* qui sert de couleur de fond et agit aussi sur la couleur ambiante des matériaux, mais également sur les lampes qui peuvent ainsi varier d'intensité pour simuler l'ambiance d'un environnement.
*L'application des textures sera détaillée aux chapitres suivants.*

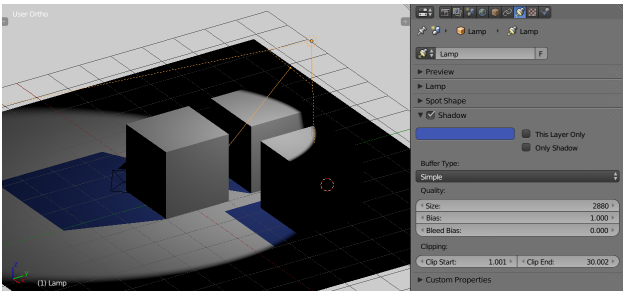


## LES OMBRES PORTÉES

Commune aux lampes ***Spot*** et ***Sun***, les options d'ombres sont importantes à comprendre pour une bonne qualité d'éclairage et de bonnes performances graphiques. En effet, le calcul des ombres portées dynamiques peut vite devenir gourmand en ressources, et doit être utilisé avec parcimonie. Cela reste malgré tout une option de choix pour ajouter réalisme et ambiance à une scène.

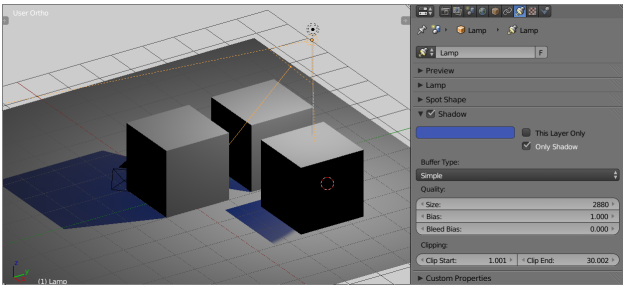


On peut modifier la couleur de cette ombre pour simuler une ambiance. Ce qui est à l'ombre de la lampe est éclairé par une lumière d'ambiance colorée.

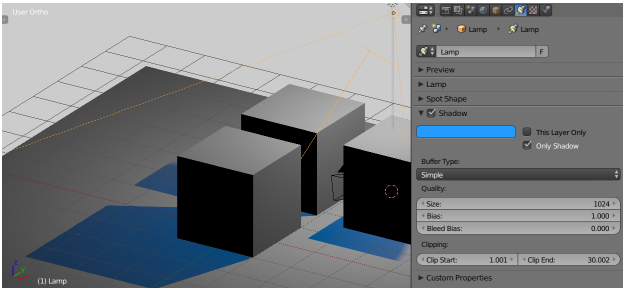



L'option ***This Layer Only*** permet de limiter la production d'ombre aux seuls objets présents sur le même calque que la lampe.

L'option ***Shadow Only*** permet de produire une ombre sans ajouter de lumière à la scène. En fait, la lampe ne produit que les ombres. Dans l'exemple ci-dessous, nous avons dû ajouter une autre lumière pour l'éclairage général en complément du spot qui produit des ombres bleues, selon son rayon d'éclairage.




L'option ***Buffer Type*** est importante : elle détermine le mode de production de l'ombre. Une ombre est en fait une texture ajoutée par-dessus les autres. Elle a donc une définition. Le type ***Buffer Simple*** produit simplement une texture à partir de la scène, le type ***Variance*** produit lui une texture adoucie, plus floue, qui permet d'obtenir des ombres douces, au prix d'un calcul supplémentaire.

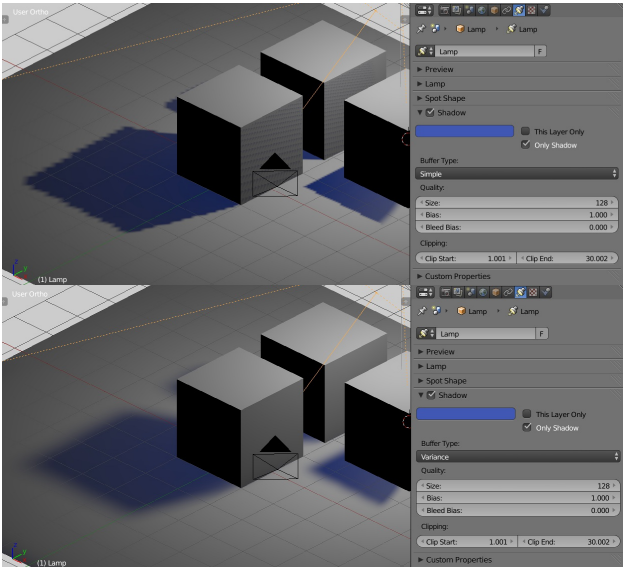



L'option ***Quality : Size*** est importante également : c'est elle qui détermine la taille de la texture utilisée pour l'ombre (toujours carrée). Ceci est particulièrement visible dans le cas d'une ombre de type ***Simple***. Une ombre de petite taille (ex. 128) produira une ombre pixelisée alors qu'une ombre plus détaillée (512 ou 1024) sera plus précise et aura des bords plus nets.

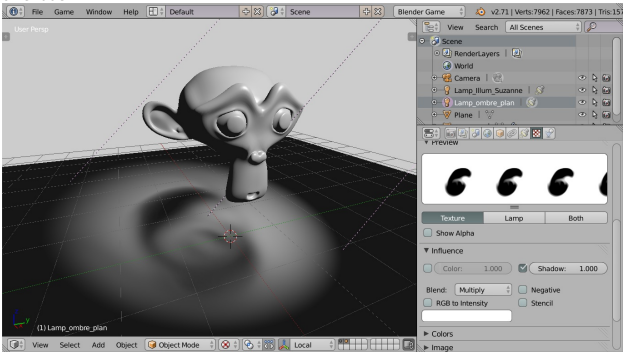
*Pour la taille des images, il est préférable d'utiliser des puissances de 2 pour optimiser le calcul. 128, 256, 512, 1024 ou 2048 sont les échelles habituellement utilisées.*

Cette option est à considérer avec attention, parce qu'elle joue un rôle aussi bien sur la performance que sur l'esthétique. Des ombres de grandes tailles, en grand nombre, risquent de réduire fortement le taux de rafraîchissement de l'image, augmentant donc la qualité du rendu mais diminuant la qualité de l'animation.

***Variance*** semble être une bonne option, permettant d'utiliser plus de lampes avec un rendu adouci tout en conservant une fréquence élevée de l'affichage.




En ajoutant, une texture à une lampe, il est possible de lui demander d'agir sur une ombre. Suivant l'effet souhaité, cela peut être nécessaire de jouer avec le positionnement des objets et des lampes sur plusieurs calques de la scène. Ici, il s'agit d'une distorsion : un objet fixe (Suzanne) ayant une ombre tout à fait différente de l'ombre attendue.



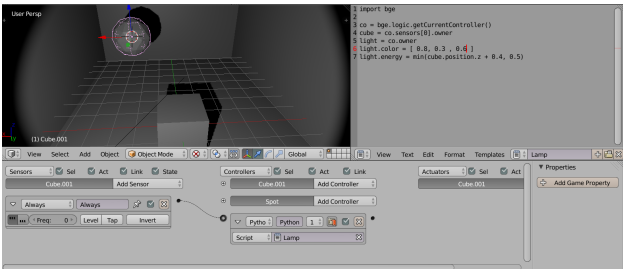

Multiplier les lampes pour améliorer l'éclairage ou ajouter des effets peut être tentant. Il faut cependant résister à cette envie lorsque des ombres dynamiques sont utilisées et bien réfléchir à l'importance d'avoir une ombre dynamique pour chaque lampe. Il est possible de produire un éclairage de bonne qualité et de figer (*baker)* des ombres dans les textures des objets statiques. Ceci sera détaillé dans la section **Travailler le rendu du jeu** dans le chapitre **Pré-calcul des textures et de la lumière**.

L'espace dans lequel les ombres seront produites peut également être limité, en contrôlant la distance de leur apparition (***clipping***). ***Clip Start*** et ***Clip End*** sont respectivement les distances minimales et maximales entre lesquelles l'objet doit se trouver pour produire une ombre. La distance des objets sur lesquels l'ombre sera éventuellement projetée peut sortir de cette intervalle. En limitant cette distance, les interactions entre lampes facilitent ainsi les calculs du moteur de jeu. Ces distances sont représentées dans la vue 3D par un segment de droite affiché dans l'axe de la lampe, ce qui aide à les régler finement.

## SCRYPTHON !

Interagir avec les lumières en Python est également possible. L'objectif va être de faire varier l'intensité de la lumière en fonction de la hauteur d'un objet 3D qui va rebondir sur le sol.

Les captures d'écran ci-dessous montrent la scène dans sa situation de départ ainsi que le code Python que nous allons utiliser. Vous pouvez également voir les deux briques logiques que nous utilisons pour la mise en place de la scène. La première est la brique *Always*. Elle est configurée en mode *pulse*, ce qui permet de s'assurer que l'on aura une génération d'événement constants. Attaché à cette génération d'événement, nous exécutons le script python *Lamp*.




Étudions maintenant en détail le script Python que nous utilisons.

```
from bge import logic

cont = logic.getCurrentController()

ball = cont.sensors[0].owner
light = cont.owner

light.color = [ 0.8, 0.3 , 0.6 ]
light.energy = min(ball.position.z + 0.4, 0.5)
```

La première ligne correspond, comme toujours, à l'import des modules recevables.
On récupère ensuite le contrôleur qui est relié à ce script Python. Ici le contrôleur est rattaché à notre lampe, or nous avons besoin de connaître la position en z de notre boule.

```
ball = cont.sensors[0].owner
```

Avec cette ligne de code, nous allons récupérer notre boule par l'intermédiaire de notre ***sensors Always*** (premier et unique *sensor* attaché à notre *controller*) que nous avons opportunément accroché à notre objet. Chaque contrôleur possède une liste appelée *sensors* qui répertorie la totalité des *sensors* qui lui sont rattachés. Ici, il n'y a qu'un seul *sensor*, on peut donc sans problème utiliser la notation **[0]** pour récupérer le premier élément de la liste.

```
light = cont.owner
```

Nous récupèrons la lumière de la manière classique, en récupérant le *owner* du contrôleur courant.

```
light.color = [ 0.8, 0.3 , 0.6 ]
```

Ici, nous nous amusons à modifier la couleur de la lumière pour la rendre violette.
La ligne suivante est un peu plus compliquée. On va ici modifier la puissance de notre lumière.

```
light.energy = min(ball.position.z + 0.4, 0.5)
```

Pour cela, on change la valeur de l'attribut `energy` de notre *spot*. La fonction `min` en Python nous permet de récupérer la plus petite valeur d'un ensemble, qui se limite ici à la valeur de la coordonnée en z de notre balle rebondissante à laquelle on ajoute 0.4 ou à 0.5). On va donc obtenir un effet 'jour , nuit, jour , nuit' tant que la balle rebondit. Lorsqu'elle finira par s'arrêter notre scène sera alors plongée dans le noir.

# 13. INTRODUCTION AUX MATÉRIAUX ET TEXTURES

Les matériaux et leurs textures sont les futures couleurs du monde de notre jeu ! Et, puisque nous devons limiter le nombre de polygones pour des raisons de performance du moteur de jeu, les matériaux et textures seront donc très importants pour habiller nos objets et personnages. Nous allons voir dans ce chapitre deux approches pour donner à nos surfaces des aspects variés et enrichir l'aspect global de notre environnement.

De manière générale, matériaux et textures sont intimement liés. Il est possible d'avoir des matériaux sans textures, mais cela limite fortement les effets visuels possibles. Par contre, il n'existe pas de texture visible sans qu'elle soit au moins attachée à un matériau.

Il est important de noter aussi que la gestion des matériaux et textures pour le *Game Engine* se fait différemment qu'en mode de rendu Blender classique. Les différences sont tant dans la gestion des *nodes*, pour ceux qui ont l'habitude d'aborder cette pratique sous cet angle, que dans les possibilités offertes par le mode de texture plus ancien, sans utilisation de *nodes*.

## LES DEUX FAÇONS DE CRÉER DES MATÉRIAUX

Les matériaux dans le *Game Engine* peuvent être définis de deux manières différentes, qui se complètent d'une certaine façon. Ceci est dû à l'historique du logiciel.

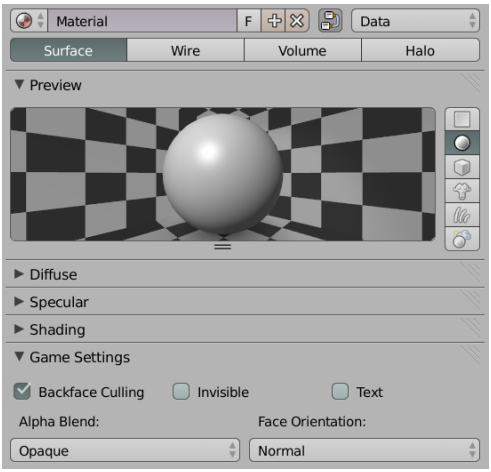
- La méthode ancienne, héritant de la méthode des versions 2.4x, est de définir le matériau dans les panneaux des onglets matériaux et textures. C'est aussi la seule méthode qui fonctionnera lorsque nous serons en mode de rendu *Multitextures*.
- La méthode actuellement développée utilise le système des nœuds de matériaux, qui tire utilement parti des capacités de calcul des cartes graphiques modernes. Pour utiliser ce dernier système, il faudra néanmoins initialiser certains paramètres dans les panneaux de matériaux et textures. Cette méthode est la méthode conseillée, permettant de construire des textures complexes et dynamiques au besoin. Cette méthode ne fonctionne que si le mode de rendu *GLSL* est activé.

## LES PARAMÈTRES D'AFFICHAGE DES MATÉRIAUX

Nous allons décrire dans cette partie les différents panneaux présents dans l'onglet `Material` de la fenêtre `Properties`.

Les options `Diffuse`, `Specular` et `Shading` sont a priori connues. Rien de mystérieux concernant leurs utilisations, car elles ne diffèrent pas de ***Blender Internal*** ou de ***Blender Cycle***.

Une des spécificités du mode *Game Engine*, c'est que nous avons maintenant un panneau *Game Settings* avec quelques options importantes.




Le *Backface Culling*sert uniquement à calculer les faces dont la normale pointe vers la caméra. Techniquement, si l'on coche cette option, si une normale est dans le même sens que la caméra elle n'est pas affichée et donc l'objet devient transparent. On peut utiliser cette méthode pour le cas où la caméra entre dans un objet afin de ne pas avoir un écran noir.

Le paramètre *Invisible*, quant à lui, fait disparaître totalement le matériau en jeu et tous les objets auxquels il est appliqué.

Le bouton *Text*sert uniquement à afficher le matériau sur un objet texte, mais seuls les objets textes sont concernés.

Les modes d'*Alpha Blend* indiquent de quelle manière gérer la transparence :

- *Opaque* : c'est l'option par défaut. Dans ce mode, toutes les faces sont opaques.
- *Add*: la transparence de la face dépend de sa couleur, plus elle est proche du blanc, plus le matériau est opaque et plus elle est proche du noir, plus il est transparent.
- *Alpha clip* : ce mode supporte un canal de transparence *Alpha* (*RGBA*). On l'utilise pour avoir de la transparence quand on utilise les textures. Cependant, la transparence est limitée : on a soit complètement transparent (0), soit complètement opaque (1).
- *Alpha* : sûrement le mode le plus adapté pour la transparence car il gère les textures qui ont une transparence avec un canal alpha et supporte la semi-transparence (de 0 à 255). Par contre, ce mode demande plus de calculs, et n'est pas conseillé pour des faces trop nombreuses.

## LES PARAMÈTRES DES TEXTURES

Comme nous le disions plus haut, il est très rare d'avoir un matériau qui n'est pas lié à une texture. De manière générale, une texture donnera toujours au minimum une couleur plus complexe à notre objet, que simplement la présence de l'option *Diffuse* dans l'onglet *Material*.

Notez que les textures procédurales ne sont pas supportées par le *Game Engine*. Si vous les avez utilisées pour modifier l'apparence de votre objet, il vous faudra passer une technique nommée *baking* qui permet de faire la conversion de ces textures en images bitmap. Cette technique est d'ailleurs décrite au chapitre Pre-calcul des textures et de la lumière de la section **Travailler sur le rendu**.

Vous l'aurez compris, le travail se fera donc uniquement avec des textures image ou par une programmation plus complexe en GLSL.

*Concernant les images et pour des raisons de performance, il sera toujours préférable de limiter le nombre de textures utilisées ainsi que leur taille. Par défaut, une image dans Blender est un carré de 1024 pixels de côté. Le Game Engine n'est cependant pas restreint à*

*l'affichage d'images carrées pour les textures. Mais il est recommandé de prendre cette habitude pour faciliter les calculs liés à leur affichage. Une texture idéale sera donc un carré dont la longueur de côté, en pixel, sera une puissance de 2 (exemple: 128x128, 256x256, 512x512,...).*

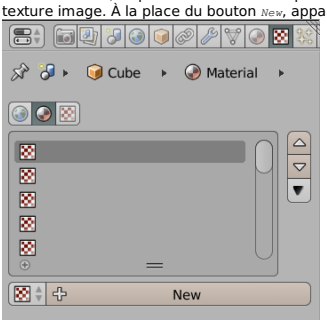
### Créer une nouvelle texture

La création des textures de type Image liées à un matériau pour le *Game Engine* est identique au *Blender Internal,* mais nécessite la connaissance de quelques subtilités pour qu'elle s'affiche correctement dans le jeu.

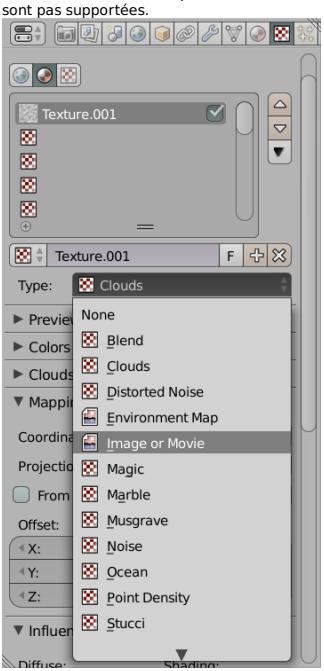
Pour les utilisateurs de ***Cycles***, une petite explication s'impose. Par défaut, les matériaux du *Blender Internal* et du *Game Engine* n'utilisent pas de *nodes*. Donc, contrairement à ***Cycles***, il n'y a pas un système nodal commun aux textures et aux matériaux. Dans le ***Node Editor***, les nœuds des textures et les nœuds de matériaux sont donc différents. Ils ont chacun leur propre onglet. Le nœud **Texture** permet d'introduire une texture créée de façon traditionnelle et de la lier avec un nœud matériau. Il est donc impératif d'initialiser une texture à la manière de *Blender Internal*(avec l'onglet `Texture`de la fenêtre`Properties`) avant de pouvoir l'utiliser dans les *nodes* de matériaux.

De façon traditionnelle (à la manière de *Blender Internal*), les textures sont donc disposées dans une liste où l'ordre détermine la priorité d'une texture par rapport à une autre. Ci-dessous, voici la méthode pour ajouter une texture à cette liste :

1. Sous cette liste, cliquez sur le bouton `New` pour créer une nouvelle texture image. À la place du bouton `New`, apparaît son nom.



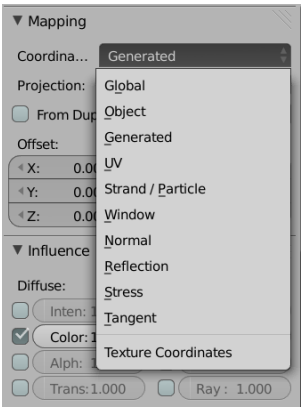

2. En dessous du nom, il faut préciser le type *Image ou Movie*. C'est le seul type fonctionnel dans le *Game Engine*.
*Depuis la version 2.72 de Blender, il s'agit du type par défaut à la création d'une texture.*
Comme on vous le disait plus haut, les textures procédurales ne sont pas supportées.




3. Cliquez alors sur le bouton `Open` dans l'onglet `Image` placé en dessous de manière à pouvoir charger une image.

### Définir le mode de projection

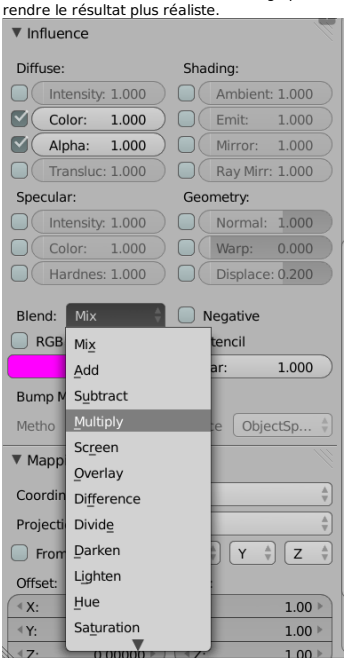
Le panneau `Mapping` est configuré par défaut en mode coordonnées automatiques `Generated` pour mapper la texture selon une projection plane (`Flat`), mais une bonne pratique est d'utiliser systématiquement des coordonnées `UV` pour tout *mesh* (volume ou plan). Seule la projection de type `Flat` fonctionne dans le *Game Engine*, les autres types de projections ne sont pas supportées.




### Définir l'influence

Le panneau ***`Influence`*** indique comment la texture va influencer les paramètres du matériau. Ce panneau ne s'affiche que si on n'utilise pas le mode *node* pour gérer la texture de notre matériau.

Dans notre capture d'écran, nous avons activé le *slider* ***`Color`*** et ***`Alpha`***. Ce qui veut dire que notre image de texture influence l'apparence colorée de notre objet et joue aussi sur sa transparence. Nous aurions pu activer uniquement l'influence *Alpha*. La couleur n'aurait alors pas été affectée, juste la transparence de l'objet. Il est ainsi possible d'activer ou de désactiver un seul ou tous les réglages du panneau. Il n'est pas rare aussi d'affecter la géométrie de manière à utiliser les variations de luminosité de notre image pour simuler du relief et rendre le résultat plus réaliste.



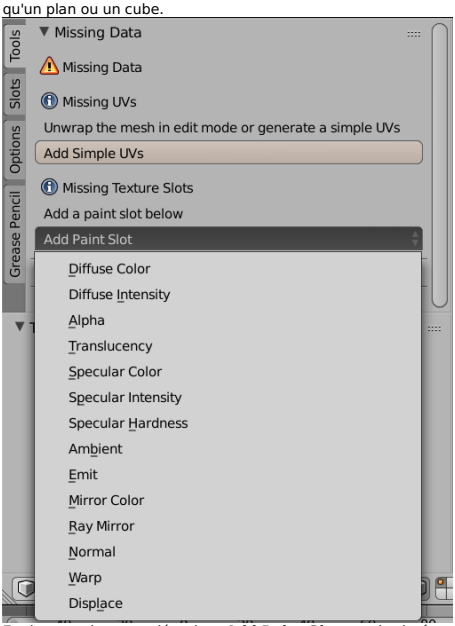

Les autres panneaux de l'onglet`Texture` sont des notions supposées connues.

*Lors de l'utilisation de textures répétées, il est important de bien travailler l'image initiale de manière à ce que les raccords ne soient pas visibles dans le jeu. La fonction* `Filtres > Mappage > Rendre raccordable` *de Gimp permet d'aider et marche parfaitement sur les textures de matériaux. Dans Krita, appuyer sur* `v` *après une sélection fera un remplissage automatiquement raccordé de la sélections sur toute sa surface.*

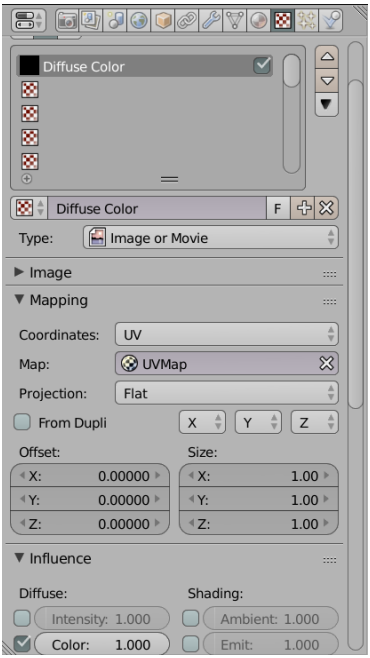
### Création d'une texture depuis le mode de peinture

Depuis la version 2.72 de Blender, le mode de peinture ou ***Texture Paint Mode*** est devenu très intéressant et vous serez peut-être plus tenté de l'utiliser plutôt que de passer par un logiciel tiers pour créer vos textures.
Dans ce cas vous pourrez directement y créer vos textures.

Dans l'onglet ***Tools*** de la nouvelle colonne d'outils de ce mode, il y a un premier bouton ***Add simple UVs*** qui créera automatiquement des coordonnées UVs faisant se répéter l'image peinte sur chaque face du mesh.
Le dépliage UVs du maillage en ***Edit Mode*** sera donc toujours une phase indispensable si vous souhaitez texturer un objet plus complexe qu'un plan ou un cube.

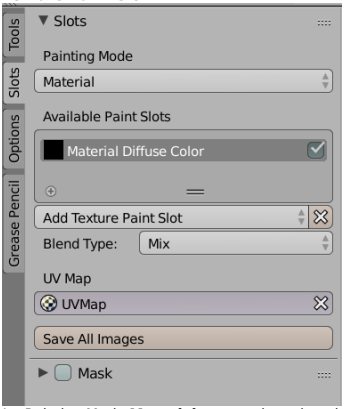



En dessous, le menu déroulant ***Add Paint Slot*** est plus intéressant.
Il crée, paramètre et nomme directement une texture *Image* avec un mapping UVs agissant sur le paramètre d'Influence choisi comme expliqué précédemment.



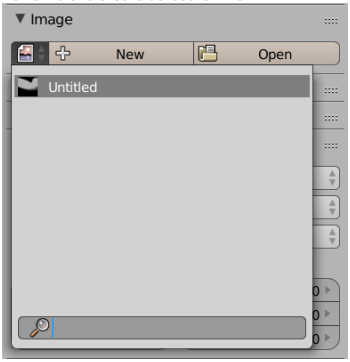

Dès qu'une texture est correctement paramêtrée pour être peinte; l''onglet ***Tool*** change d'aspect et n'affiche plus que le ***brush*** actif.

Pour créer une seconde texture, il faut afficher l'onglet ***Slots*** de la colonne d'outils.
Vous pouvez alors préciser l'UV Map et le Blend Type de l'influence de la future texutre, puis désormais utiliser le menu nommé ***Add Texture Paint Slot***.




Le *Painting Mode* ***Material*** permet donc de peindre les textures en pensant à leur influence sur le matériau.
Cette démarche d'une texture par paramêtre sied mieux à un rendu *Blender Render* qu'à un jeu. Cependant, cela reste pertinent si l'on envisage d'en faire une seule texture issu d'un *baking* ou rendu en texture. Vous êtes toujours libre de modifier le panneau *Influence* après coup. La création automatique d'une texture vous facilitera le travail même si vous souhaitez l'exploiter avec les nodes. Il vous suffira de la supprimer du matériau standard avant de le changer utilisant des noeuds et de la réutiliser en tant que noeud Texture.

Si vous souhaitez peindre une image sans l'intégrer immédiatement au matériau, il faudra alors utiliser le *Painting Mode* ***Image***. Dans ce cas, il faudra ensuite faire le paramêtrage de la texture vous-même comme indiqué précédemment.
Alors au lieu de charger la texture en cliquant sur le bouton *Open*, il suffira de la récupérer depuis la liste des images présentes dans le fichier .blend à côté du bouton ***New***.



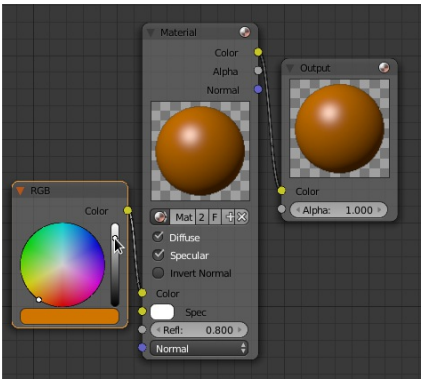

## LES NŒUDS MATÉRIAUX

Pour gérer nos matériaux et textures, nous pouvons aussi utiliser les *nodes*: une pratique qui tend à se généralise peu à peu dans l'ensemble de Blender. Nous vous encourageons d'ailleurs à utiliser cette méthode parce que nous pensons que les nœuds de matériaux exposent une arborescence de matériau et textures plus simple à comprendre que les relations établies par l'ordre de la liste et les panneaux `Influence`de chaque texture (méthode plus attachée à une pratique de Blender 2.4x).

*Nous tenons à rappeler que cette façon de travailler ne fonctionne que si vous avez GLSL activé.*

### Un matériau simple avec des nœuds

Voici un exemple simple de matériel fait avec les *nodes*.

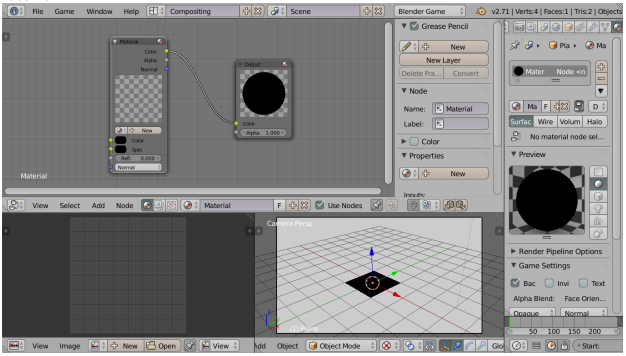



Ici, un *node*`RGB` envoie une donnée de couleur vers l'entrée couleur du matériau, dont la sortie nourrit un *node*`Output` qui donnera le résultat sur l'objet.

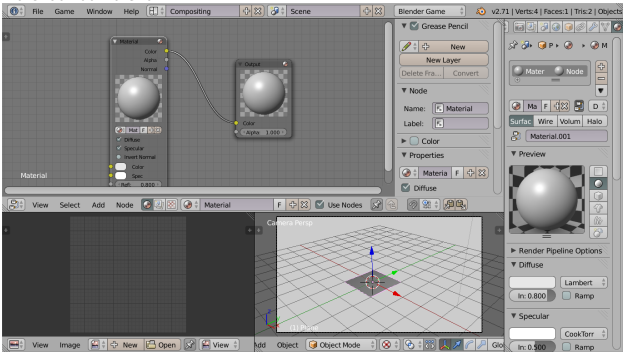
### Matériau avec une texture dans le `Node Editor`

Voici, en détail, la procédure pour créer des *nodes* de matériaux et les lier avec des textures :

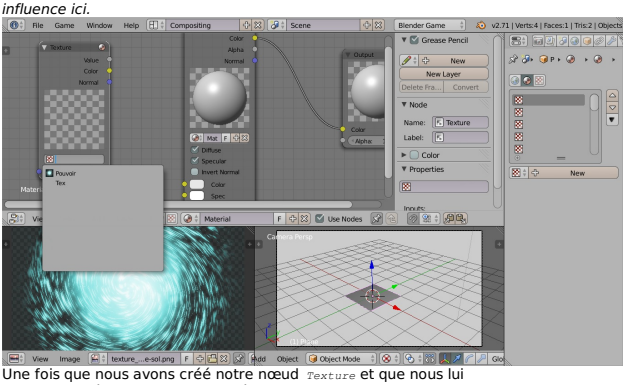
1. Affichez les nœuds de matériaux dans le **`Node Editor`**.
2. Activez les nœuds de matériaux en cochant la case **`Use Nodes`** dans le `Header` du `Node Editor`. Une fois réalisé, l'objet devient complètement noir et les panneaux changent dans le panneau **`Material`**.



3. Cliquez sur le bouton **`New`** pour donner un nouveau matériau au nœud `Material`. L'objet n'est plus noir. Si le matériau à utiliser avait déjà été créé, il suffit de le choisir dans la liste des matériaux du *Material*.



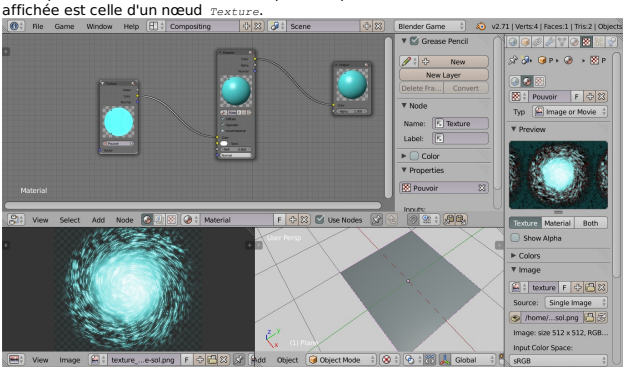
4. Appuyons ensuite sur **`Maj+A`** pour créer un nœud `Input > Texture`. Assignons-lui une texture créée au préalable en la choisissant dans la liste.
*Référez-vous au sous-chapitre **Créer une texture** décrit plus haut si vous ne savez pas comment faire, mais ne tenez pas compte du panneau **`Influence`** qui n'aura effectivement aucune influence ici.*



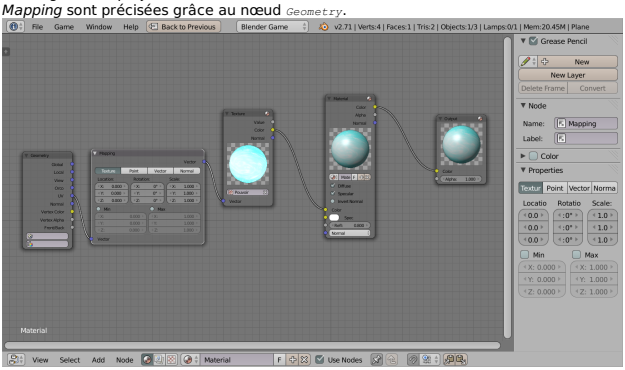
5. Une fois que nous avons créé notre nœud `Texture` et que nous lui avons assigné notre bloc de données *Texture*,nous devons supprimer ce bloc de donnée de la liste qui se trouve dans

l'onglet *Texture* de la fenêtre *Properties*, sinon la texture sera appliquée de façon traditionnelle en plus de son utilisation par les *nodes*.

6. Le paramétrage de notre nœud texture est accessible dans l'onglet ***Textures*** à chaque fois que vous sélectionnez le nœud correspondant. L'absence de liste indique bien que la texture affichée est celle d'un nœud *Texture*.



7. Pour appliquer votre texture en mode *UV* sur votre objet, vous devez maintenant connecter au minimum un nœud ***Geometry*** à partir de sa sortie *UV* dans l'entrée ***Vector*** de notre nœud *Texture*. Comme dans *Cycles*, nous pouvons aussi utiliser un nœud ***Mapping*** qui permet de modifier la taille, la rotation et les décalages de la projection de la texture. Les coordonnées de *Mapping* sont précisées grâce au nœud *Geometry*.



*L'information essentielle est que les textures doivent avoir une des coordonnées UVs pour s'afficher. Les autres modes de Mapping ne fonctionnent pas dans le Game Engine.*

# 14. MATÉRIAUX ET TEXTURES AVANCÉS

Dans ce chapitre, nous verrons quelques effets avancés liés aux matériaux et textures. Il s'agit principalement de jouer sur des effets de relief, de transparence ou de brillance de nos objets afin de donner plus de réalisme ou de corps à notre environnement. Ces effets sont généralement utilisés pour palier à la simplification de la géométrie due à notre besoin de performance. C'est aussi un des canaux importants de l'expression du style graphique du jeu.

Nous verrons 3 aspects importants combinant propriétés des matériaux et usage avancés des textures : les effets de brillance, de relief et de transparence.

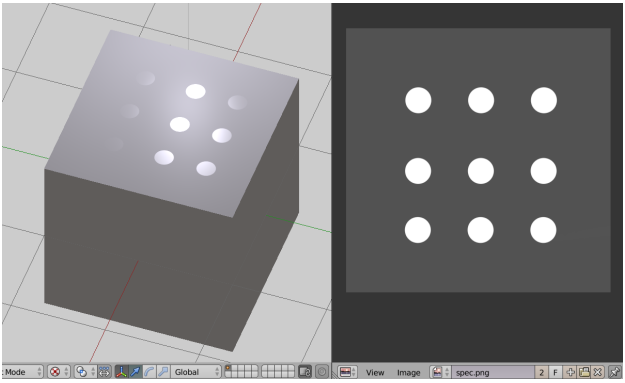
*Vous devez avoir compris le chapitre précédent, Introduction aux matériaux et textures, pour pouvoir suivre celui-ci.*

## GÉRER LA BRILLANCE OU LES REFLETS DE LUMIÈRE SUR NOTRE MATÉRIAU

Le canal spéculaire décrit la brillance. Il peut s'agir d'une couleur uniforme (pour un matériau à la surface régulière, plus ou moins brillante), ou d'une image en niveaux de gris servant à distinguer les zones mates (caoutchouc, tissu...) des zones brillantes (métal, verre, graisse...), avec toutes les nuances intermédiaires (caoutchouc mouillé, métal sale...).

*On peut l'obtenir généralement en se basant sur la texture de couleur, que l'on modifie dans un éditeur d'images comme Gimp ou Krita.*

Dans l'exemple suivant, une texture agit sur le canal spéculaire pour que le cube brille uniquement dans certaines zones.




En méthode *Blender Internal*(pas d'utilisation de nodes), il faut définir dans l'onglet`Texture` de notre matériau, ajouter une texture dans notre liste qui n'a d'influence que sur les options sous *Specular* dans le panneau `Influence`.

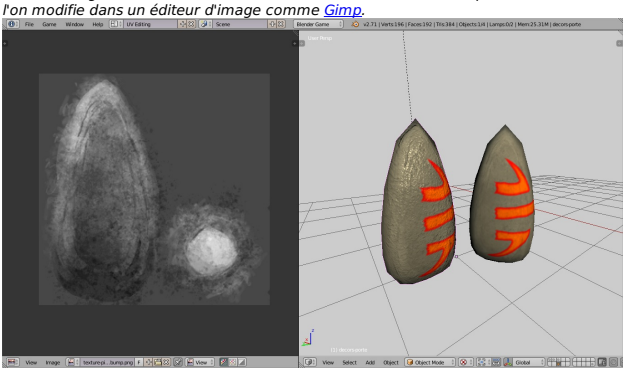
En mode *node*, ajoutez un nœud `Texture` qui se connectera uniquement à notre entrée `Specular` de notre nœud`Material`.

## DONNER DU RELIEF

Comme nous travaillons sur une application temps réel, les performances sont une question cruciale. Nous utiliserons donc des maillages composés de peu de faces (*low poly*), et par conséquent peu détaillés. Pour pallier ceci, nous pouvons simuler un faible relief à l'aide d'une texture de relief. C'est un effet très intéressant, car il utilise peu de ressources, pour un résultat qui peut être saisissant s'il est bien utilisé.

Une texture de relief (*Bump map*) est une image en niveau de gris qui contient des informations de profondeur : si les pixels sombres sont des "creux", les pixels clairs sont des "bosses". L'effet convient bien à des reliefs peu prononcés et dont la précision n'est pas fondamentale (une route goudronnée, du tissu...).

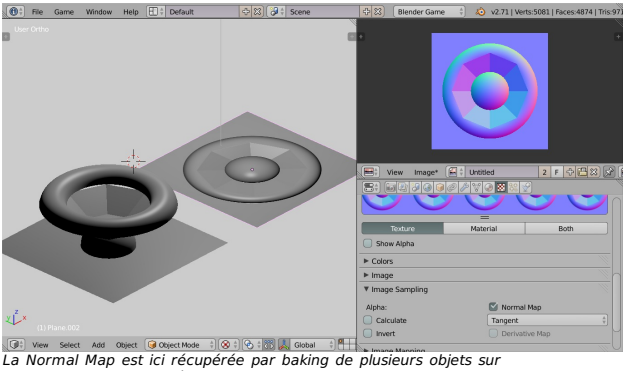
*On l'obtient généralement en se basant sur la texture de couleur, que l'on modifie dans un éditeur d'image comme Gimp.*



*A gauche, la texture Bump. Elle est utilisée dans le menhir de gauche et est absente dans le cas du menhir de droite.*

Une *Normal Map* est un peu plus complexe. Elle décrit de manière très précise le relief de l'objet, elle permet vraiment de donner l'illusion d'un maillage dense. On l'obtient généralement en comparant un modèle low poly et un modèle high poly (cette technique est traitée dans le chapitre **Optimisation de la texture et de la lumière : Baking**), ou à défaut, à partir d'une *bump map* et d'un outil comme Insane Bump (plugin Gimp).

Dans le BGE, elles s'utilisent de la même manière que pour un matériau de rendu, soit en utilisant un node texture nourrissant une entrée ***Normals*** d'un matériau, soit de manière traditionnelle en cochant la case ***Normal*** (panneau ***Textures*** / ***Influence*** / ***Geometry***) et en réglant la puissance de l'effet. Pour une *normal map*, il faudra également, dans ***Textures*** / ***Image Sampling***, cocher ***Normal Map***.

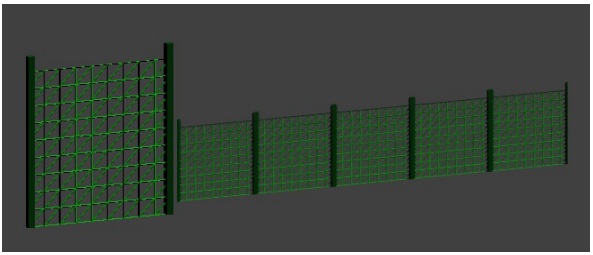



*La Normal Map est ici récupérée par baking de plusieurs objets sur l'objet actif (le plan) grâce l'option Selected to Active du panneau Baking dans l'onglet Render.*

## L'UTILISATION DE LA TRANSPARENCE

Outre l'usage évident de transparence pour imiter des matériaux de type eau, verre, etc., la transparence est utilisée dans les jeux vidéo pour simplifier la géométrie. On peut par exemple simuler une forme complexe plane par un plan carré et une texture partiellement transparente.

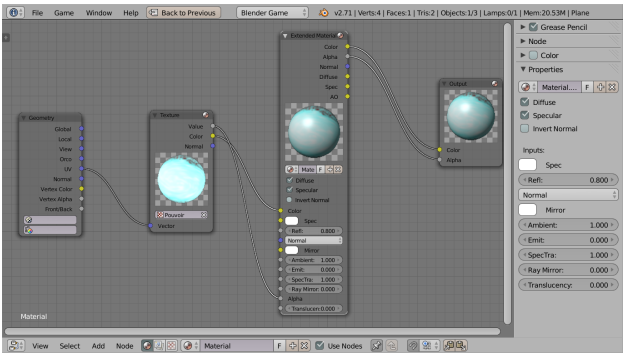
C'est une façon d'éluder la création de géométrie pour définir une forme. Dans un décor naturel, les arbres, les feuilles, les grillages, par exemple impliqueraient de créer une géométrie très lourde, mais ramenés à des plans texturés avec de la transparence, un projet d'un décor familier devient réalisable avec beaucoup moins de faces.



Le *terme Alpha* (en référence à la couche ou au canal Alpha d'une image) où qu'il soit mentionné dans l'interface de *Blender* est relatif à de la transparence. Dans l'*UV/Image Editor*, vous pouvez afficher une image avec sa transparence, sans sa transparence ou uniquement sa couche *Alpha*. Vous pouvez la modifier avec les outils de peinture. Plutôt que de dédier une image en noir et blanc à la transparence, l'usage le plus fréquent est d'utiliser la couche *Alpha* de l'image. Le réglage de la transparence de façon traditionnelle n'est pas évident car les options par défaut n'aident pas ou sont disséminées dans différents recoins de l'interface. Pour information, la méthode traditionnelle implique de cocher la case du panneau *Transparency* dans l'onglet des matériaux. Les boutons pour les différents types de transparence sont relatifs au *Blender Internal*. Seule, la glissière *Alpha* a une importance sur la transparence globale du matériau. Passez sa valeur à 0.

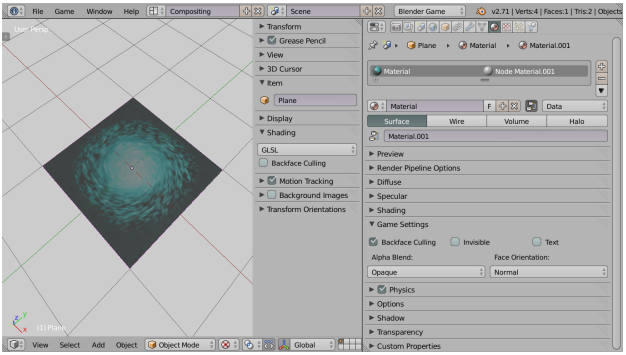
Dans l'onglet des textures, pour qu'une texture modifie la transparence, il faudra activer également une glissière *Alpha* dans le panneau Influence. Par défaut, le *blend mode* de ce panneau est *Mix*. La valeur de la glissière *Alpha* du matériau est à adapter au *blend mode* utilisé. Avec un *blend mode Mix,* la valeur à indiquer est zéro car les valeurs de la couche alpha de l'image vont s'ajouter à la valeur *Alpha* nulle du matériau. Mais pour un *blend mode Multiply,* il faudra cocher l'option *RGB to Intensity* et utiliser une valeur *Alpha* du matériau égale à 1. (Si vous ouvrez de très vieux fichiers créés avant l'apparition du *blend mode Mix,* il est probable que le *blend mode* utilisé soit *Multiply)*. La méthode nodale permet de faire abstraction du *blend mode* et d'une activation du panneau de la transparence dans l'onglet du matériau. Il suffit simplement d'utiliser un nœud *Extended Material* possédant un *socket* d'entrée *Alpha* au lieu du nœud *Material* par défaut qui ne comprend qu'une sortie *Alpha*.

Pour créer un nouveau matériau pour ce nœud, cliquez sur le bouton *New* du noeud *Extended Material.* Déterminez les coordonnées de la texture en connectant la sortie *UV* d'un noeud *Geometry* à l'entrée *Vector* du noeud *Texture*. Faites agir la texture sur la transparence du matériau. Dans le cas où vous souhaitez utiliser la couche alpha d'une image, connectez la sortie *Value* du nœud Texture à l'entrée *alpha* du nœud *Extended Material.* Dans le cas où vous utilisez la texture comme un masque noir et blanc à partir d'un image sans couche alpha, connectez la sortie *Color* du nœud Texture à l'entrée *alpha* du nœud *Extended Material.*

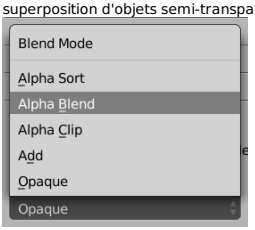


*Attention : pour que la transparence du matériau soit effective, il faut absolument que la sortie alpha du nœud Extended Material soit connectée à l'entrée alpha du nœud Output.*

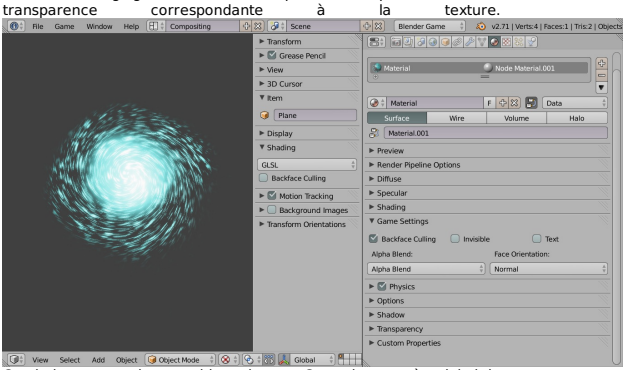
Bien que l'effet de transparence soit effectif dans la prévisualisation (preview) du matériau, il n'est pas flagrant dans le *viewport*. C'est à cause du réglage par défaut de l'option *alpha blend* du panneau *Game Settings* sur le choix *Opaque* (ignorant volontairement la transparence pour garder toutes les faces du mesh opaques).



Dans l'onglet du matériau, ce panneau *Game Settings* possède un paramètre *Alpha blend mode* à modifier pour adapter la superposition d'objets semi-transparents.



Passez ce réglage sur le choix *Alpha Blend* pour avoir une transparence correspondante à la texture.




Ce choix respecte les transitions douces. Contrairement à celui-ci, le choix *Alpha Clip* n'accepte pas les valeurs intermédiaires et force une transition abrupte entre une partie totalement transparente ou totalement opaque. Lorsque des surfaces se chevauchent, le choix *Alpha Sort* donnera un effet de profondeur, d'épaisseur en éclairant plus fortement les éléments les plus proches. Pour générer des effets lumineux comme du feu, des *sprites* lumineux peuvent se superposer et alors le choix *Add* permettra d'augmenter l'intensité lumineuse d'après la transparence aussi appelée de la transparence additive.

A présent, il reste une différence entre le résultat visible dans la vue 3D et le résultat dans le *viewport* du jeu. La surface n'est visible que d'un côté pour accélérer l'affichage. Par défaut, l'option *Backface Culling* du matériau est activée. Il faudra alors la désactiver pour créer un élément de décor fixe visible des deux côtés comme un grillage. Pour accorder la vue 3D au choix fait pour un matériau, la même option *Backface Culling* est accessible dans le panneau *Shading* des propriétés de la vue 3D.

Les autres paramêtres de ce panneau ne sont pas directement en relation avec la transparence. Cependant lorsque l'on créé un plan texturé, c'est soit pour faire un sprite soit un *billboard* toujours dirigé dans la même direction (souvent vers la caméra). Le menu *Face Orientations* offre plusieurs choix de direction automatique.

# 15. MATÉRIAUX DYNAMIQUES ET TEXTURES ANIMÉES

Dans ce chapitre, nous allons voir trois méthodes pour faire varier les matériaux et textures dans le temps. Ces trois méthodes sont très différentes les unes des autres, mais sont regroupées ici parce qu'elles concernent la modification en direct de l'apparence de nos objets.

Nous verrons premièrement une technique bien connue des créateurs de jeux 2D, c'est l'utilisation de *sprites* pour rendre des effets d'animation. La deuxième passe par un processus de remplacement, à la volée, dans la carte graphique elle-même, des images attachées aux textures. Et la dernière méthode concerne la fabrication d'un matériau qui se modifie en fonction d'un évènement particulier activé en cours de jeu.


## TEXTURES ANIMÉES PAR *SPRITES*

L'effet de transparence est souvent utilisé avec les *sprites* animés. Il s'agit de réduire les ressources en utilisant une seule image regroupant les quelques frames d'une animation courte.

Il vous faut considérer l'image comme une texture normale au niveau du matériau.

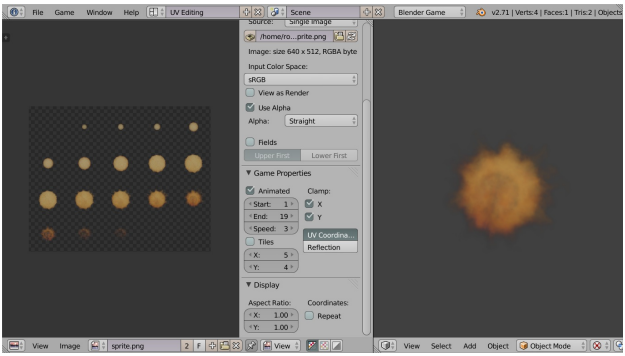
Le réglage de l'animation se fait dans l'*UV/Image Editor*.

Texturez un simple plan avec l'image correspondant à collection des étapes de l'animation.

Elle doit être formée de tuiles de taille identique. Chaque tuile correspondant à une *frame* de l'animation.
Affichez l'image dans l'*UV/Image Editor*.

Les UVs de la face du plan doivent recouvrir l'ensemble de l'image.

Faire un *reset* des UVs est un moyen simple de s'en assurer.




Affichez le panneau *Game Properties* dans la colonne des propriétés de l'image. Afin de recouper le nombre et la disposition des tuiles, dans la colonne des propriétés de l'image, il faut définir un nombre de colonnes avec la valeur x et de lignes avec la valeur y de l'encart *Tiles*.
Cochez le *Clamp* en x et en y selon l'aspect de votre image mais laissez l'option *Tiles* désactivée pour l'animation. Pour indiquer la tuile de départ et celle de la fin de l'animation, précisez les valeurs *Start* et *End* de l'encart *Animated*. Les tuiles sont indexées de gauche à droite, de haut en bas. Pour que l'animation se lance dans le *Game Engine*, cochez la case *Animate*d du panneau *Game Properties*. L'animation tourne en boucle lorsque vous pressez la touche **P**. À ce stade, il reste à adapter la vitesse de défilement des tuiles lorsque vous lancez le jeu. Pour cela, le plus simple est de faire des aller-retours successifs entre le lancement du jeu et modification de la valeur *Speed*.

## TEXTURES ANIMÉES PAR REMPLACEMENT

La technique présentée dans cette partie est très différente. Nous allons remplacer une texture dans la carte graphique par une image ou un flux d'image de provenance variée : fichier image, fichier vidéo, *webcam*, bloc mémoire ou un mélange de ce qui précède. Le terme fichier est à prendre au sens large : il peut s'agit d'un fichier ou d'un flux vidéo accessible sur internet. Il suffit pour cela de spécifier une *URL* plutôt qu'un nom de fichier.

En plus nous pouvons appliquer un effet sur l'image avant qu'elle ne soit envoyée au *GPU* : incrustation vidéo de type écran vert, gamme de gris, *normal map*. Cette technique avancée n'est accessible que par Python via le module `bge.texture`. La magie derrière ce module est la librairie libre *FFmpeg*. La plupart des formats et des codecs supportés par *FFmpeg* sont utilisables dans ce module. En voici une liste non exhaustive :

- AVI
- Ogg
- Xvid
- Theora
- dv1394 camera
- carte d'acquisition compatible avec video4linux (cela inclut de nombreuses webcams)
- jpg
- flux RTSP

Le principe est simple. Nous identifions une texture sur un objet existant en utilisant la fonction `bge.texture.materialID`. Ensuite nous créons un objet texture dont le contenu sera modifié dynamiquement suivant l'une ou l'autre méthode évoquée plus haut et nous échangeons les deux textures dans le GPU. Le moteur graphique du BGE n'est pas averti de la substitution et continue à afficher les objets comme d'habitude, sauf que nous avons maintenant le contrôle de la texture. Quand l'objet texture est détruit, l'ancienne texture est restaurée.

### Image fixe

Ce script Python change la texture d'un objet par une image fixe.

```
import bge
from bge import texture
from bge import logic
```

```
def createTexture(cont):
 obj = cont.owner

 # Obtention de l'index de la texture initiale
 ID = texture.materialID(obj, 'IMoriginal.png')

 # Creation de la nouvelle texture
 object_texture = texture.Texture(obj, ID)

 # il faut garder une reference permanente
 obj.attrDict["tex"] = object_texture

 # chemin complet vers le fichier image
 url = logic.expandPath("//newtex.png")

 # chargeons la texture en memoire
 new_source = texture.ImageFFmpeg(url)

 # swap de texture
 object_texture.source = new_source

 # remplacement dans le GPU
 object_texture.refresh(False)

def removeTexture(cont):
     obj = cont.owner
 try:
 del obj.attrDict["tex"] except: pass
```

Voyons ligne par ligne le fonctionnement de ces fonctions.

```
import bge
from bge import logic
from bge import texture
```

Ces 3 lignes nous permettent d'accéder aux modules bge.logic et bge.texture par leurs noms. Nous avons besoin de texture pour créer la texture animée et de logic pour des fonctions d'intendance.

```
def createTexture(cont):
 obj = cont.owner
```

La fonction *createTexture* sera appelée par un *controller* Python, l'argument est donc la référence du controller. Nous avons besoin de l'objet car c'est lui qui possède la texture que nous voulons modifier.

```
 # Obtention de l'index de la texture initiale
 ID = texture.materialID(obj, 'IMoriginal.png')
```

Nous supposons dans cet exemple que l'objet possède un matériel qui utilise une texture original.png. Le préfixe IM signifie que nous cherchons une image et pas un matériel. Nous pouvons directement rechercher un matériel par son nom en utilisant le prefixe 'MA'.

```
 # Creation de la nouvelle texture
 object_texture = texture.Texture(obj, ID)
```

Une fois le matériel identifié nous créons un objet Texture en spécifiant l'identifiant de matériel trouvé précédemment. Le premier canal de texture du matériel sera automatiquement remplacé par cette nouvelle texture.

```
 # il faut garder une reference permanente
 obj.attrDict["tex"] = object_texture
```

L'objet texture doit impérativement rester en mémoire tant que nous voulons utiliser la nouvelle mémoire, sinon le *garbage collector* de Python supprimera l'objet dès la fin de la fonction et la texture sera supprimée. La méthode que nous utilisons ici est de conserver la référence l'objet logic en l'ajoutant au dictionnaire **attrDict** de l'objet qui persiste tant que l'objet n'est pas détruit (ce qui se produit au plus tard à la fin du jeu). Nous aurions pu sauver la référence dans le dictionnaire global de l'objet logic (`logic.globalDict`) pour le conserver tout au long du jeu même si l'objet est détruit.

```
 # chemin complet vers le fichier image
 url = logic.expandPath("//newtexture.jpg")
```

Nous avons besoin du chemin complet de la texture. Cette ligne suppose qu'une image 'newtexture.jpg' est présente dans le répertoire du fichier blend. Les '//' sont remplacés par le chemin complet du blend.

```
 # chargeons la texture memoire
 new_source = texture.ImageFFmpeg(url)
```

La fonction ImageFFmpeg() du module bge.texture permet de charger une image fixe. Nous utiliserions une autre fonction pour charger une vidéo ou pour créer un bloc mémoire.

```
 # swap de texture
 object_texture.source = new_source
```

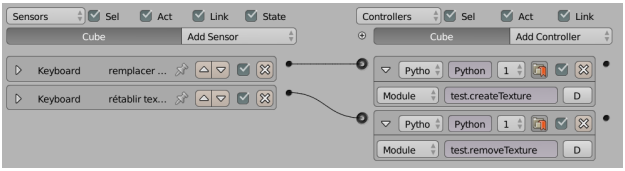
Jusqu'à présent, notre texture était vide. En lui donnant une source, on lui donne du contenu, ici notre image préalablement chargée.

```
 # remplacement dans le GPU
 object_texture.refresh(False)
```

Le chargement de la nouvelle texture dans le GPU se fait par la fonction refresh. L'argument de refresh est un booléen qui indique si la texture doit être recalculée (et donc ré-envoyée au GPU) au prochain refresh. Dans le cas d'une image fixe, ce n'est pas nécessaire mais pour une vidéo, ce serait indispensable, de même qu'il serait indispensable d'appeler *refresh* très fréquemment pour que les différentes images de la vidéo soit extraites et envoyées au GPU (tout se passe dans refresh). Pour utiliser ce script nous aurons besoin de deux briques *controllers* qui appellent la fonction `createTexture`au début du jeu pour effectuer le remplacement. Et une à la fin qui appelle la fonction *removeTexture* pour restaurer la texture.

*La fonction removeTexture est optionnelle car la texture sera d'office restaurée à la fin du jeu grâce au fait que nous utilisons le dictionnaire de l'objet pour sauver la référence: ce dictionnaire est effacé à la fin du jeu ou dès que l'objet est détruit.*

Comme vous le voyez sur la capture d'écran, nous avons choisi la méthode Module et non pas la méthode Script pour ajouter notre fonctionnalité python. Cela implique donc que nous devions utiliser la syntaxe **NomModule.NomFonction** pour indiquer à Blender où se trouve le code python à utiliser.



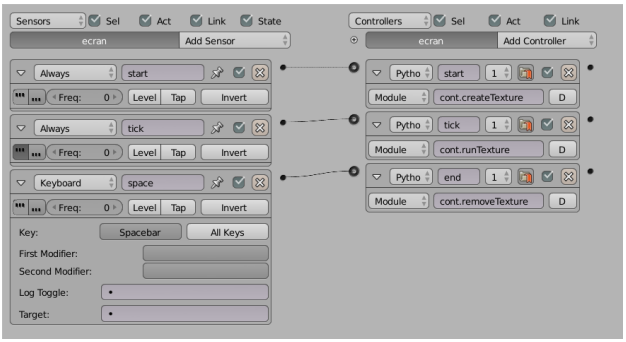

### Image vidéo

Dans cet exemple que nous ne détaillerons pas autant, nous remplaçons la texture d'un objet par une vidéo, ici une vidéo en ligne.

```
import bge
from bge import texture
from bge import logic

def createTexture(cont):
 obj = cont.owner

 # Obtention de l'index de la texture initiale
 ID = texture.materialID(obj, 'IMoriginal.png')

 # Creation de la nouvelle texture
 object_texture = texture.Texture(obj, ID)

 # il faut garder une reference permanente
 obj.attrDict["tex"] = object_texture

 # chargeons la texture en memoire
 new_source = texture.VideoFFmpeg("http://mirrorblender.top-ix.org/movies/sintel-1024-
surround.mp4")

 # swap de texture
 object_texture.source = new_source

 # demarrons la video new_source.play()
 # remplacement dans le GPU object_texture.refresh(True)

 def runTexture(cont): obj = cont.owner if "tex" in obj: obj.attrDict["tex"].refresh(True)
print("1") def removeTexture(cont): obj = cont.owner try: del obj.attrDict["tex"] except:
pass
```

Ce script se distingue du précédent par deux points :
1. L'objet source est de type VideoFFmpeg au lieu de ImageFFmpeg (si nous avions utilisé ImageFFmpeg nous n'aurions obtenu que la première image du film). Cet objet a une API adaptée à la lecture de fichier vidéo (play, pause, stop). Ici nous utilisons *play()* pour lancer la vidéo.

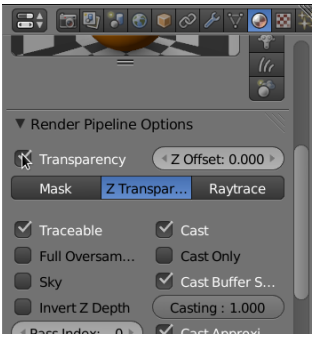
2. Nous devons rafraîchir la texture fréquemment pour que les images du film soient successivement envoyées au GPU. Pour cela, la fonction *runTexture* doit être appelée fréquemment. Nous utiliserons pour cela un sensor **Always** en mode *tick*:



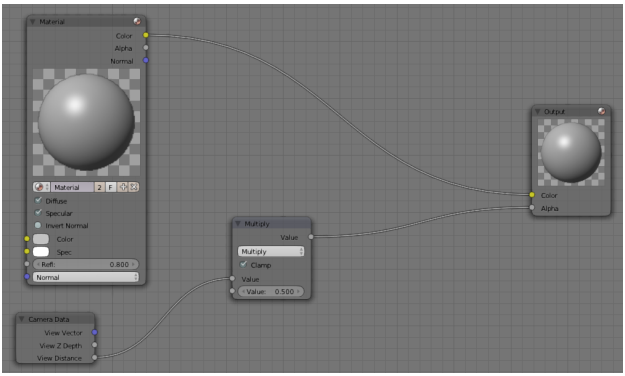

## MATÉRIAUX DYNAMIQUES AVEC LES *NODES*

Utiliser les *nodes* ouvre la porte à une multitude d'effets poussés basés sur des paramètres d'entrée (*inputs*) qui seraient normalement inaccessibles. Comme exemple, fabriquons un matériau qui devient transparent s'il est observé de près : cela peut servir par exemple à voir à travers une partie du décor si la caméra se rapproche trop. En partant du *material* basique vu sous forme de *node* que nous avons préparé précédemment, il n'y a que très peu de changements à effectuer.

Pour commencer, il faut activer la transparence du *material*en cochant l'option ***Transparency*** dans l'onglet ***Render Pipeline Options*** du *material*.




Ensuite, dans le`Node Editor`, il suffit de jouer avec le paramètre***Alpha*** du ***node Output***. Dans notre cas, nous devons relier cet `Alpha` à la distance de la caméra, disponible via le *node* ***Camera Data***, en exploitant l'entrée ***View Distance***. Lorsque cette distance atteint ou dépasse 1, l'*alpha*prendra sa valeur maximale (opaque). Pour un réglage plus fin, insérer un ***node Math***en mode *Multiply* permet de modifier cette distance critique. L'option `Clamp` est une bonne pratique pour limiter les valeurs de sortie entre 0 et 1, puisque l'*alpha* va ignorer des valeurs extérieures à cet intervalle.


Material
Color
Alpha
Normal
Material
2
F
Diffuse
Specular
Invert Normal
Color
Spec
Refl
0.800
Normal
Multiply
Value
Multiply
Clamp
Value
Value:
0.500
Output
Color
Alpha
Camera Data
View Vector
View Z Depth
View Distance

# 16. AJOUTER DU SON

À côté de l'aspect visuel, un jeu vidéo (ou une application interactive) a souvent besoin d'un univers sonore pour être complet. Nous allons voir dans ce chapitre comment ajouter du son et de la musique dans notre univers visuel.

## LES FORMATS AUDIOS

Considérons deux cas de figures concernant les sons dans un jeu, les sons d'ambiance et les sons générés par des actions. Pour les sons courts, les petits bruitages, nous choisirons des fichiers au format .wav ou .aif, non compressés, qui seront plus rapides à lancer pour le Game Engine et assureront une bonne synchronisation avec les interactions. Par contre, si les fichiers sont plus gros (par exemple des musiques d'ambiance), nous pourrons prendre un autre format compressé (.ogg ou .mp3) pour éviter d'alourdir le .blend. Concernant le fichier, les formats supportées par le BGE sont wav, mp3, ogg, aif et flac.

## LOGIC BRICK

Il est très simple de jouer un son via les briques logiques, pour réagir à un événement du jeu. Pour lancer des sons via les *Logic brick*, nous devons utiliser l'*actuator* ***Sound***. Dans cet *actuator*, nous avons donc le fichier à lire, le mode de lecture, le volume du son, qui varie de 0 à 2 et le *Pitch* qui sert à modifier la fréquence du son.


## LES MODES DE LECTURE

Selon les cas, il peut être intéressant de jouer un son de différentes manières : une seule fois si le personnage se cogne contre un mur, plusieurs fois en boucle tant qu'il marche sur un sol bruyant, une seule fois sauf si on l'arrête, etc.

### Jouer le son en entier

En utilisant ***Play End*** à la première impulsion, le son se lance et se joue complètement. En cas de seconde impulsion pendant que le son est encore en train de se jouer, il recommencera à 0. Cela correspond bien pour des sons d'événements comme les chocs ou les actions du personnage.

### Jouer le son en entier, sauf si on l'arrête

En mode ***Play Stop***, tant que l'entrée de la brique est active, le son est joué. Mais dès que l'entrée n'est plus active, le son s'arrête. En cas d'arrêt (fin du son ou entrée inactive), le prochain lancement se fera au début du son. Attention toutefois, même si l'entrée reste active, le son ne se relancera pas de lui-même à la fin de sa lecture.

### Jouer le son en boucle, pour toujours

Avec ***Loop End***, le son joue complètement et va se relancer en boucle après une impulsion. En cas de seconde impulsion, le son recommence à 0. Cette approche est parfaite pour une musique d'ambiance.

### Jouer le son en boucle, sauf si on l'arrête

En mode ***Loop Stop***, c'est le même principe que le *Play Stop* sauf que le son se jouera en boucle tant que l'entrée n'est pas relâchée. Idéal pour des sons fonctionnels ou de processus, comme des bruits de machine et moteur, de l'eau qui coule, etc.

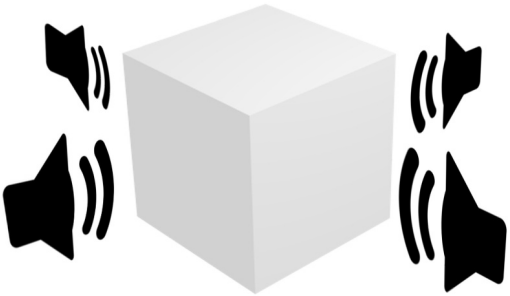
### Jouer le son dans les deux sens

Le ***Loop Bidirectional*** joue le son du début jusqu'à la fin et si l'entrée est encore active à la fin du son, celui ci est alors joué à l'envers.

### Jouer le son dans les deux sens, sauf si on l'arrête

Le ***Loop Bidirectional Stop*** joue le son tant que l'entrée est active mais s'arrête dès lors que l'entrée est inactive. Et comme le *Loop Bidirectional*, si l'entrée est active jusqu'à la fin du son, alors celui-ci se joue à l'envers, jusqu'à ce que l'entrée logique soit mise à Faux.

## EFFET 3D



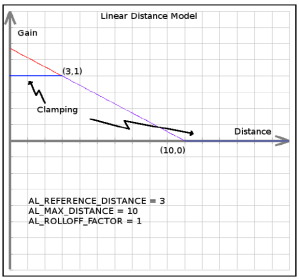
Ajouter un peu de profondeur et d'espace aux sons permet de créer une meilleure immersion dans le jeu. Par exemple si un ennemi se trouve loin du personnage, on l'entendra à peine ; mais s'il surgit juste derrière le personnage, le son sera très fort !

Nous devons en premier changer le type du ***Listener*** de la scène (dans l'onglet `Scene > Audio > Listener`), choisir la valeur ***Linear Clamped*** (le modèle de distance pour le calcul des atténuations). Occupons nous de la brique logique `Sound` : cochons la case ***3D Sound***. Les valeurs de ***Reference Distance*** (distance de référence), ***Maximum Dist*** (distance maximale) et ***Rolloff*** sont les paramètres définissant la diffusion du son.

À partir de la distance de référence, le son commence à diminuer. La distance maximale est la distance à laquelle on pourra entendre le son. Le ***Rolloff*** est le paramètre de la courbe d'atténuation du son. Avec la valeur 1 la diminution du son est linéaire suivant la distance entre la distance de référence et la distance maximum. Pour ceux que ça intéresse, l'audio est gérée par **OpenAL** et selon la documentation de la bibliothèque, on a :

```
distance = max(distance, AL_REFERENCE_DISTANCE)
distance = min(distance, AL_MAX_DISTANCE)
gain = (1 – AL_ROLLOFF_FACTOR * (distance –
            AL_REFERENCE_DISTANCE) /
             (AL_MAX_DISTANCE – AL_REFERENCE_DISTANCE))
```

Ce graphique présente la diffusion du son pour des valeurs données.




Avec une valeur de 200 pour le Rolloff nous avons à peu près une valeur de 2000 pour le Maximum Distance, ce qui correspond +-10 Unités Blender (donc si nous passons le Maximum Distance à 4000, le son sera entendu jusqu'à 20 Unités Blender, etc.). Par contre, cela ne marche plus très bien avec des valeurs trop petites (< 1).

On peut aussi jouer avec le ***Reference Distance*** pour définir quand effectuer l'effet d'atténuation du son.

Pour conclure, dans une scène on change `Scene > Audio > Listener` en mode Linear Clamped

On place un cube avec l'actuator sound :

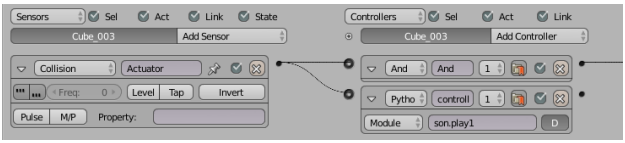
- `Maximum Distance` = 2000 (donc une décroissance sur 10 Blender Units !)
- `Rolloff` = 200
- `Reference Distance` = 20

Voilà ce qui se passera :

- à 1 `Blender Unit` : on entendra normalement le son (100%)
- à 10 `Blender Units` : on entendra normalement le son (100%)
- à 15 `Blender Units` : on entendra normalement le son (100%)
- à 20 `Blender Units` : on entendra normalement le son (100%)
- à 23 `Blender Units` : on entendra le son un peu moins fort...
- à 28 `Blender Units` : on entendra le son très bas...
- à 30 `Blender Unites` : on n'entendra pratiquement plus le son.

## SCRYPTHON !

Concernant la gestion du son, le Python amène vraiment une nouvelle palette de fonctionnalités pour traiter les sons. Cette fois, nous utilisons un autre module Python : aud. Pour ajouter du son lors d'une collision entre deux objets, nous utilisons la configuration suivante.




Rappelons que nous avons choisi la méthode Module et non pas la méthode Script du contrôleur pour appler le script son.py suivant.

```
import aud
from os import chdir
from bge import logic

# on n'oublie pas de se placer dans le bon repertoire
chdir(logic.expandPath("//"))
device = aud.device()

def play1():
     factory = aud.Factory('music1.mp3')
     factory = factory.limit(0.0, 5.0)
     handle = device.play(factory)

def play2():
     factory = aud.Factory('music1.mp3')
     factory = factory.fadein(0.0, 5.0)
     factory = factory.fadeout(7.0, 9.0)
     handle = device.play(factory)

def splitch():
     factory = aud.Factory('splitch.mp3')
     factory = factory.limit(0.0, 5.0)
     handle = device.play(factory)
     handle.loop_count = 2
```

Les trois premières lignes nous permettent d'importer nos modules et les trois suivantes servent à bien définir le répertoire courant (voir le chapitre **Bien préparer son projet**).

Nous commençons par créer une variable **device** qui nous permettra d'agir sur la carte son. Ensuite nous déclarons trois fonctions, **play1**, **play2** et **splitch**. Dans la capture d'écran nous demandons à Blender d'utiliser la fonction play1, mais nous pourrions très bien utiliser les autres fonctions à d'autre moment dans notre projet. Les trois fonctions commencent par la création d'une **aud.Factory**, un lien vers un fichier son que l'on va pouvoir configurer, avant de le jouer grâce à la méthode **play()** de notre *device*.

Regardons donc cette fonction **play1**. Une fois la Factory créée, nous allons utiliser sa méthode **limit()** pour sélectionner uniquement une plage de temps du fichier son. La fonction **limit()** prend deux paramètres float : **start** et **end**, qui définissent en secondes le début et la fin de notre sélection. Une fois que l'on a configuré ce que l'on veut, on demande à notre *device* (qui, rappelez-vous est un lien vers la carte son) : ici la carte son va jouer les 5 premières secondes de notre fichier audio.

Dans la fonction **play2()**, nous souhaiterions aller plus loin en ajoutant des fondus, pour rendre plus fluides le début et la fin de la lecture. Une fois la Factory créée, nous allons la configurer en appelant successivement deux fonctions (lignes 2 et 3 de **play2()**) : **fadein()** pour un fondu en ouverture, puis **fadeout()** pour un fondu en fermeture. On peut en effet mettre en place plusieurs effets en enchaînant les appels à des fonctions de configuration. Il ne nous reste qu'à jouer le son avec device.play().

Terminons maintenant avec la fonction **splitch()**. Vous aurez remarqué que lorsque nous utilisions la méthode **play()**, nous récupérions un *handle*. Voyons comment en tirer profit. On peut par exemple modifier son attribut **loop_count**, qui permet de jouer plusieurs fois le son. Ici nous avons donné 1 comme valeur, il sera donc joué 1 fois à la 4e ligne de la fonction.

Petite précision intéressante, si vous donnez une valeur négative à **loop_count**, votre son se répétera continuellement. Le handle que vous renvoie la fonction **play()** permet d'agir sur la lecture du son une fois celle-ci en cours. Vous pouvez par exemple la mettre en pause (avec la fonction **pause()** ), la reprendre (avec la fonction **resume()** ) ou arrêter définitivement la lecture (avec la fonction **stop()** ). Concernant la fonction **stop()**, elle arrête bien totalement la lecture. On ne peut la reprendre qu'en relançant la fonction **play()** de notre périphérique.

# 17. FILTRES DE RENDU

Jusqu'à présent, nous avons étudié un ensemble de techniques qui permettent d'agir sur le contenu de l'image avant son rendu. Nous verrons dans ce chapitre qu'il est possible d'agir après le rendu, afin de personnaliser le style visuel du jeu, ou d'agir de manière très pointue sur l'affichage.

## FILTRE 2D

La brique logique 2D Filter Actuator permet d'ajouter facilement un shader d'affichage, afin d'obtenir des effets comme du flou, ou une coloration sépia par exemple. Il s'agit de filtres 2D (comparables à ceux d'un éditeur d'image comme Gimp) qui s'appliquent sur chaque frame, après leur rendu et avant leur affichage. Voici un exemple de jeu BGE sans filtre 2D.

Le même avec un filtre Sepia.

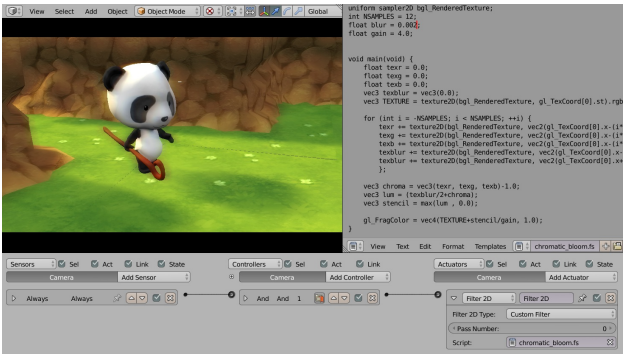
Ou un filtre Dilatation (voir le détail des textures).

La mise en place est très simple, puisqu'il suffit d'ajouter un actuator 2D Filter et de sélectionner un filtre. On peut insérer cette brique logique dans un objet quelconque comme la caméra.


Les options Enable et Disable permettent de contrôler les filtres dynamiquement. On peut par exemple imaginer un flou de mouvement qui ne s'active qu'au-delà d'une certaine vitesse de déplacement, ou encore un filtre que le joueur peut désactiver dans un menu pour améliorer les performances. Ici, deux touches du clavier permettent d'activer ou de désactiver le filtre.


L'option Pass Number permet de contrôler les filtres de manière indépendante. Ici, le clavier permet d'agir sur le filtre Sepia défini sur la passe 1, mais pas sur le filtre Blur défini sur la passe 0.


Quelques filtres simples sont fournis avec le BGE, mais ils ne répondront sans doute pas à tous vos besoins. Il est naturellement possible d'utiliser des filtres personnalisés (via l'option Custom Filter et un simple fichier texte) que l'on aura téléchargé ou écrit soi même.



Le filtre Chromatic Bloom visible sur cette capture est disponible ici : http://mattline1.blogspot.co.uk/2012/08/blender-2d-filters.html

Il existe un Add-on regroupant un belle collection de ces filtres (parmi d'autres shaders utiles pour le BGE) : http://urfoex.blogspot.be/2013/03/bge-glsl-glsl-shader-repository-addon.html

Cette série de tutoriels explique les bases nécessaires pour écrire vos propres shaders : http://solarlune-gameup.blogspot.co.uk/search/label/GLS

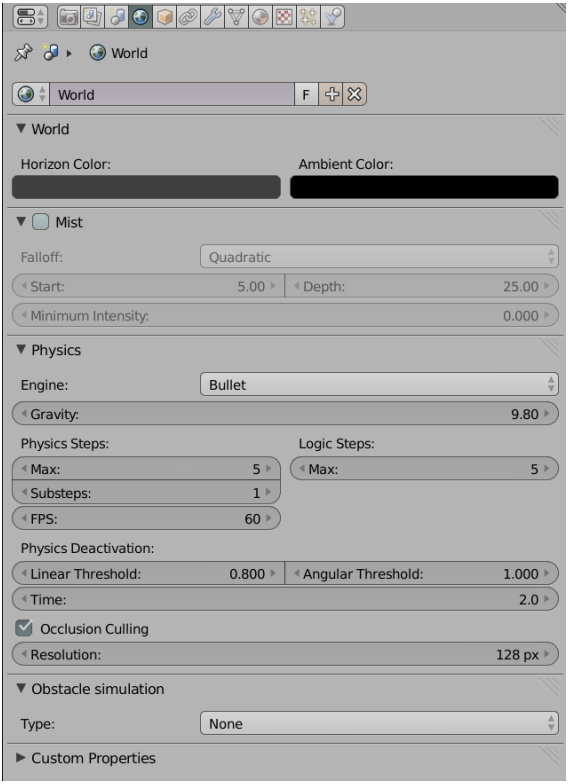
# COMPORTEMENT DES OBJETS ET PERSONNAGES

**18.** DÉFINIR LES CARACTÉRISTIQUES DU MONDE
**19.** LE COMPORTEMENT DES OBJETS DE NOTRE MONDE
**20.** DES PERSONNAGES AU BON COMPORTEMENT
**21.** DES ANIMATIONS DANS NOTRE JEU
**22.** ANIMER DES PERSONNAGES DANS LE GAME ENGINE
**23.** INVERSION DE CINÉMATIQUE DANS LE GAME ENGINE
**24.** GÉNÉRATION D'OBJETS
**25.** RECHERCHE DE CHEMIN AUTOMATIQUE

# 18. DÉFINIR LES CARACTÉRISTIQUES DU MONDE

À la base, notre espace de jeu est simplement constitué d'un certain nombre de polygones virtuels, flottant dans le vide. Pour leur donner de la consistance et de la "matière", le moteur de jeu incorpore un simulateur de "lois physiques" incluant notamment la gravité, les collisions, des unités, et beaucoup d'autres paramètres et comportements. Dans le mode ***Blender Game***, l'onglet ***World***est radicalement différent de celui que l'on trouve lorsque qu'on est en mode `Blender Render`. C'est là que se trouvent les options concernant le moteur physique, sous le panneau ***Physics***.

Dans l'onglet `World`, on peut également changer la couleur du fond (***Horizon color***), donner un éclairage de base ou une couleur d'ambiance (***Ambient Color***), et on peut même ajouter un effet de brouillard dans l'onglet ***Mist***. Mais au-delà de ces options habituelles, il faudra définir le comportement général de notre univers de jeu.




## CHANGER LES LOIS DU MONDE : LE MOTEUR PHYSIQUE

La première option de l'onglet ***Physics*** est tout simplement de choisir le moteur physique. Blender ne supporte actuellement que le moteur ***Bullet****, le seul autre choix ***None***est donc de désactiver la physique. Ce choix a du sens si le jeu ne nécessite aucune sorte de physique : le panneau ***Physics*** des objets existera toujours, mais les valeurs ne seront pas utilisées dans le jeu.

*Si l'interface utilise une liste qui peut se dérouler au lieu d'une simple case à cocher, c'est pour permettre de compléter la liste avec d'autres moteurs physique. Comme nous avons le choix entre une interface Cycles ou Blender Internal, dans des anciennes versions, il était possible de sélectionner le moteur physique Sumo ou dans des versions spéciales d'utiliser le moteur Open Dynamics Engine.*

## DÉFINIR LA GRAVITÉ

La gravité a un impact direct sur certaines capacités de mobilité dans le jeu. La valeur par défaut affichée par Blender est 9.80 unités, ce qui correspond à la gravité terrestre de 9,80 m/s². Le résultat sera donc le plus naturel pour les joueurs. Il s'agit de définir les forces fondamentales que le monde impose aux personnages et aux objets pour définir la pesanteur. Ainsi, pour un jeu qui se passerait dans l'espace ou sur une autre planète, le changement de gravité pourrait être un paramètre particulièrement actif et affecter fondamentalement la jouabilité. Plus la gravité est faible plus les chutes seront perçues comme lentes. Une gravité à 0 fera que les objets ne chuteront pas. Ce réglage n'affectera pas les objets de type *static*. Pour les autres, la modification de la masse dans les propriétés physiques pourra être un moyen de compenser individuellement la force qu'impose la gravité selon les situations.

## LA FLUIDITÉ DE L'AFFICHAGE : GÉRER LA FRÉQUENCE DE RAFRAÎCHISSEMENT

Le ***FPS*** (*Frame Per Second*) est une option essentielle qui définit le nombre d'images que le jeu va afficher par seconde pour donner au joueur l'illusion du mouvement, de la même façon qu'on le fait pour un film. La valeur par défaut, **60 images par seconde**, est souvent acceptée comme standard pour garantir un confort de jeu. Il est bien sûr possible de la diminuer si nécessaire.

Cependant, il arrive que le jeu soit graphiquement trop gourmand en ressources et qu'une telle fréquence de rafraîchissement d'image soit impossible à atteindre. Dans de telles situations, Blender va donner en priorité les ressources de calcul au moteur physique et à la logique, quitte à perdre quelques images par seconde. Rassurez-vous, la sensation d'images saccadées (ou de "*lag*") n'est perceptible qu'en dessous de 24 ou 25 images par seconde, ce qui nous laisse un peu de marge. Les options ***Max*** de ***Physics steps*** et ***Logic steps*** déterminent combien de rafraîchissements physiques et logiques vont être intercalés entre deux rafraîchissements d'image, en cas de manque de ressources.

Il faut comprendre que le moteur physique va calculer les événements de manière séparée de l'affichage : le paramètre ***Substeps*** d e ***Physics steps*** détermine le nombre d'itérations* des calculs de physique entre le calcul des images. Plus il y aura d'étapes calculées, plus la simulation physique sera précise, mais aussi plus votre processeur aura du travail. Par exemple, si *Substeps* est réglé à 2, il y aura 2 calculs de physique entre chaque image, donc 120 par seconde en laissant les *FPS* à 60. Ces options sont donc à considérer dans l'optique de l'optimisation du projet et elles sont généralement laissées à leur valeur par défaut pour commencer.

## OPTIMISER LA PHYSIQUE POUR ACCÉLÉRER LES CALCULS

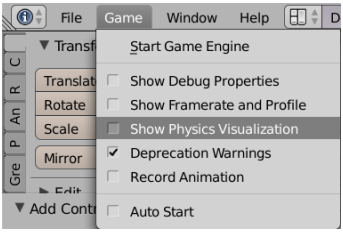
Les deux options suivantes concernent l'optimisation du moteur.

**Occlusion Culling** est une technique avancée qui permet de ne pas afficher des objets cachés par d'autres. Bien qu'il s'agisse d'une technique graphique, elle utilise accessoirement le moteur physique, ce qui justifie la présence de cette option dans ce panneau. Cette technique est décrite plus en détail dans le chapitre **Ignorer les objets hors champ** de la section **Travailler le rendu du jeu**.

**Physics Deactivation** est une option qui permet de choisir en-dessous de quelle vitesse, linéaire et angulaire, un objet peut être considéré comme inactif. Un objet inactif n'est plus mis à jour par le moteur physique : il s'arrête brusquement. Par défaut les limites de vitesses sont assez élevées: 0.8m/s en vitesse linéaire et 1 radian/s en vitesse angulaire (soit environ 1 tour en 6s). L'effet peut être assez déroutant : un objet qui est nettement encore en train de bouger se fige subitement. Pour éviter cela nous pouvons soit réduire les vitesses de seuil, soit mettre les principaux objets qui pourraient être affectés par ce problème (par exemple le personnage du jeu) en mode **No Sleeping** (cette option fait partie des paramètres physiques de l'objet). Un objet dont cette option est cochée ne devient jamais inactif et les objets qui sont en contact avec lui restent actifs tant qu'ils restent en contact.

# 19. LE COMPORTEMENT DES OBJETS DE NOTRE MONDE

Un objet dans le *Game Engine* possède généralement une double représentation : d'une part la représentation graphique qui est identique à ce que l'on voit dans la `3D View` et d'autre part une représentation physique qui est spécifique au *Game Engine*. Dans le cas de la représentation physique, on parle de modèle physique. Celle-ci est gérée automatiquement par **`Bullet`**, le moteur physique du *BGE*, et détermine le comportement de l'objet dans l'environnement du jeu, c'est-à-dire ses interactions avec les autres objets et personnages. L'ensemble des modèles physiques constitue le ***Physics World*** (monde physique) qui est spécifique à chaque scène.




L'affichage de la physique n'est pas visible sauf si l'on active l'option **`Show Physics Visualization`** du menu Game dans barre de menu principale. C'est une option de développement (*debugging*) qui sera souvent indispensable pendant la création du jeu. Alors que le moteur graphique du *BGE* s'appuie sur l'énorme puissance de calcul des cartes graphiques, le moteur physique tourne uniquement sur le processeur central de l'ordinateur du joueur et la question de l'optimisation est donc déterminante. Beaucoup d'options physiques ont un impact sur la performance du jeu et choisir les bonnes options pour les bons objets est crucial.

*Comme nous l'avons vu dans le chapitre précédent, l'utilisation du moteur physique est indispensable pour toute détection de contact ou de collision.*

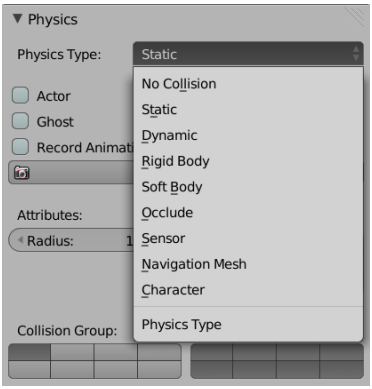
## CRÉER UNE RÉALITÉ AVEC LES TYPES DE PHYSIQUE D'OBJET

Nous créons donc véritablement une simulation de monde : l'enjeu va être de lui donner assez de "réalité" pour que le joueur y croit, et que l'immersion dans l'univers du jeu ne soit pas cassée, tout en demandant au moteur physique le minimum de calculs, ce qui nous garantit une fluidité de jeu. Un jeu trop gourmand devenant injouable, à cause des ralentissements qu'il provoque, retombe sur le même défaut : le joueur "décroche" parce que l'immersion ne fonctionne plus et qu'il est distrait par le problème technique.

Nous visons donc un compromis technique : juste assez de réalisme physique pour donner l'illusion que le monde est crédible ! Pour cela, nous devrons choisir avec attention le "type" de physique que nous attribuons à chacun de nos objets.

Afin de définir le meilleur type possible, il est important de bien définir à l'avance ce que l'on attend de l'objet en question. Par exemple :

- L'objet est-il un élément qui intervient dans le jeu ?
- L'objet fait-il partie de l'interface ?
- L'objet est-il un élément de décor ?
- Est-ce un personnage ?
- L'objet doit-il pouvoir être déplacé, et si oui avec quelle méthode ?




Les options physiques se trouvent dans l'onglet ***Physics*** de l'éditeur des propriétés(***Properties***. Le point de départ est le type d'objet (***Physics Panel*** > ***Physics Type***), qui va changer la façon dont se comportent nos objets.

### Des objets indépendants et sans interactions avec *No Collision*

Ce type est le plus simple : l'objet n'a aucune représentation physique. Cela signifie qu'un tel objet ne peut interagir avec aucun autre objet du jeu, car il est "invisible" au moteur physique. Exemples d'utilisation typiques : affichage d'information, détails graphiques secondaires, version *high poly* (maillage dense) d'un objet dont on utilise une version *low poly* (maillage léger) pour le moteur physique.

### Des éléments de décor avec une physique *Static*

Nous utiliserons ce type pour des objets immobiles ou animés qui ne sont pas influencés par les autres objets de l'environnement. Exemple : le sol et les murs d'une pièce ou un ascenseur.

D'un point de vue technique, on peut considérer que ces objets ont une masse infinie : aucune force ne peut les faire bouger. En particulier, ils sont insensibles à la gravitation. En pratique, cela veut dire que ces objets ne bougent que s'ils sont animés ou déplacés à l'aide d'*actuators* ou de scripts Python. Le moteur physique tient compte de leurs nouvelles positions pour calculer les effets sur les autres objets.
Attention, il y a des optimisations du moteur physique spécifiques aux objets statiques. L'une d'elles est que les objets statiques ne se collisionnent pas entre eux : une collision éventuelle ne provoque aucun effet, et il n'est donc pas possible de détecter le contact d'un objet statique avec un autre objet statique.

### Des objets mobiles en type *Dynamic*

Ce type s'applique à des objets qui ont une masse et peuvent bouger sous l'effet de la gravité ou de forces, mais ne peuvent pas tourner ou basculer, du moins pas sous l'effet des contacts avec les autres objets du monde physique. Plus précisément, ils restent parallèles à eux-même tant que l'on ne leur impose aucune rotation via d'une brique logique ou via Python.

Ce n'est donc pas un comportement physiquement réaliste, mais c'est utile pour les objets de type joueur ou PNJ* (Personnage Non Joueur) qui *a priori* ne doivent pas basculer, ou des objets secondaires tels que par exemple, une caisse ou un tonneau que le joueur peut pousser mais qui ne se renverseront pas.

Le fichier dynamics.blend montre ce genre d'objet en action.

### Des objets réalistes avec la physique *Rigid Body*

Ce type d'objets est le plus physiquement réaliste : ils roulent et basculent comme les objets réels. Ils sont aussi les plus coûteux en calculs. À n'utiliser que pour des détails qui rendent le jeu réaliste : par exemple des projectiles rebondissant sur les murs de la salle, ou les briques d'un mur qui s'écroulent (attention, très gourmand en calcul !).

Ils sont explicités dans le fichier rigidbodies_vaults.blend

### Des objets mous avec *Soft Body*

La physique *Soft Body* convient pour des objets mous, par exemple des tissus ou des choses organiques, qui se déforment sous un impact.

À utiliser avec précaution car ce type d'objets a de nombreuses limitations :

- Techniquement, le moteur physique considère ces objets comme des surfaces déformables et non pas comme des volumes mous. Cela convient pour une tenture mais beaucoup moins pour une masse molle.
- Aucune détection n'est possible sur ces objets : ils collisionnent avec l'environnement mais les détecteurs ne les voient pas.
- Ils n'ont pas de position ni d'orientation précise.
- Ils ont la fâcheuse tendance à traverser les sols !

L'exemple softbodies1.blend montre l'utilisation pour un tissu.
L'exemple softbodies2.blend montre une utilisation pour des objets gélatineux.

### Des objets occultant avec *Occlude*

Ce type d'objets fait partie des options avancées d'optimisation que nous ne décrirons pas ici.

Référez-vous au chapitre **Ignorer les objets hors champ** de la section **Travailler le rendu du jeu** pour voir cet aspect plus en détail. Sachez juste qu'un objet de type *Occlude* cache les objets qui sont derrière lui et évite d'envoyer au moteur graphique les objets qui sont ainsi entièrement cachés.

### Des objets détecteurs d'autres objets avec *Sensor*

Plus haut, nous avons dit que les objets statiques ne peuvent pas entrer en collision entre eux pour des raisons d'optimisation. Cependant, les objets de type *Sensor* sont semblables aux objets statiques (ils ne sont pas soumis à la gravité) tout en étant capables de détecter les objets statiques et dynamiques. Ces objets sont uniquement utilisés pour la détection : ils n'opposent aucune résistance et seront en général rendus invisibles.

Exemples d'utilisations :

- Une zone de détection de passage dans le jeu (comme une porte).
- Une ou plusieurs zones de "contacts" autour du joueur, par exemple pour détecter à quel endroit celui-ci est touché par un projectile lancé par des ennemis. Dans ce cas nous définirons plusieurs objets *Sensors* parentés au joueur, par exemple un pour la tête, un pour le torse et un pour les jambes. Un système de *score* déduira plus ou moins de points suivant le *sensor* qui aura détecté le contact avec le projectile. Un autre exemple est d'avoir un *sensor* qui définit à partir de quelle distance les PNJ* (ou seulement les ennemis) repèrent le personnage joueur.

Dans le fichier dynamics.blend, la sortie est un sensor qui met fin au jeu.

### Définir un chemin de déplacement avec *Navigation Mesh*

Un objet de type *Navigation Mesh* (abrégé *NavMesh*) est utilisé pour la navigation des PNJ. Il sera associé à un autre objet et lui servira de carte de manière à l'aider à trouver un chemin dans un niveau. Nous traiterons plus amplement de l'utilisation de cette option dans le chapitre **Recherche de chemin automatique** à la fin de cette section.

### Des personnages avec la physique *Character*

Le type `Character` est spécialement conçu pour les personnages joueurs et non joueurs (PNJ) : le moteur physique applique une mécanique simplifiée qui évite de nombreux soucis, comme par exemple traverser le sol. Le chapitre suivant est entièrement dédié à ce type très important pour beaucoup de jeux.

## MODIFIER LA PHYSIQUE DANS LE DÉTAIL, AVEC LES OPTIONS DES TYPES D'OBJETS

Les objets physiques ont plus ou moins d'options, suivant le type utilisé, qui nous permettront d'affiner leur comportement.

### Définir la masse de nos objets, avec l'attribut `Mass`

Ce réglage nous permet de définir la masse d'un objet dynamique, son unité est le kilogramme. Plus un objet a de masse, plus il est lourd, ce qui va bien sûr modifier la façon dont il va réagir à certaines forces du monde ou d'objets environnants.

### Des objets "fantômes" avec la case à cocher `Ghost`

Lorsque l'option *Ghost* est activée, l'objet permet à la logique de collision de fonctionner mais la collision ne produit pas de résultat visible. L'objet ne renvoie alors aucune force aux objets avec lesquels il entre en collision. Autrement dit, la collision ne produit pas de choc, et les autres objets passent au travers des objets *Ghost*.

*Il ne faut pas confondre cette option avec le type `No Collision` car l'objet est présent dans le monde physique et peut donc être détecté par des `Sensors` de collision (rappelez-vous, il y a collision même si celle-ci ne produit aucune force).*

Nous utilisons rarement l'option *Ghost* pour des objets statiques, car le type *Sensor* est normalement plus adapté et déjà tout prêt. Nous nous en servons donc plutôt pour des effets sur des objets qui ont déjà des physiques plus complexes. Ainsi, un exemple d'utilisation est l'effet passe-muraille : le personnage joueur acquiert temporairement un super pouvoir qui lui permet de traverser un obstacle, il va retrouver sa physique habituelle, mais temporairement il n'entre plus en collision avec le monde.

*Vous l'aurez deviné, en Python les objets `Sensors` sont d'office de type `Ghost`.*

```
# True rend l'objet invisible pour le moteur physique,
# False rend l'objet visible pour le moteur
my_object.game.use_ghost = True
```

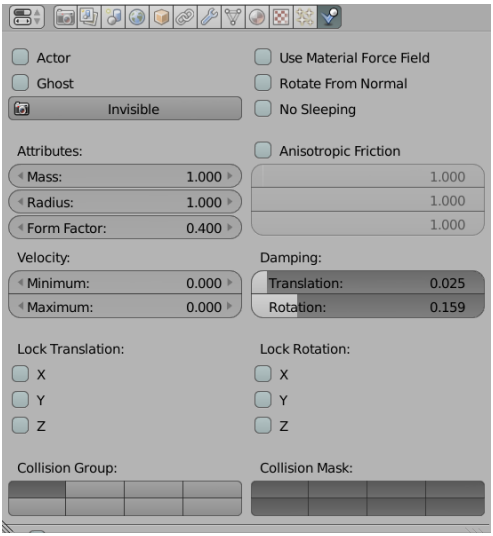
### Des objets que l'on heurte mais que l'on ne voit pas avec le bouton `Invisible`

Cette option désactive la représentation graphique de l'objet : seule la représentation physique existe. L'objet n'est donc pas affiché dans le jeu, mais il peut entrer en collision avec d'autres. Parmi les objets dont la représentation graphique n'est pas souhaitable, on peut citer :

- L'objet *low poly* (maillage léger) qui représente le modèle physique d'un objet graphique *high poly* (maillage dense).
- Un volume de détection d'intrusion, typiquement un objet *Sensor*.

L'invisibilité d'un objet n'est pas définitive. Cette option ne détermine l'invisibilité qu'au moment du démarrage du jeu. Ensuite, l'invisibilité se contrôle grâce à l'*actuator* ***Visibility*** ou grâce à Python :

```
# True rend l'objet invisible, False rend l'objet visible
my_object.hire_render = True
```




### Des objets détectables avec la case à cocher *Actor*

L'objet est détecté par les *Sensors* Near et Radar. Mais cette option est obsolète, car on peut faire mieux et plus finement avec l'option *Collision Mask* que nous allons expliquer ci-dessous.

### Une gestion des collisions plus fine avec les *Collision Group* et *Collision Mask*

Cette série de boutons fait partie des options avancées du moteur de jeu de Blender : Ils permettent de choisir finement quels objets entrent en collision avec quels autres.

L'intérêt d'utiliser les groupes de collision est double :

- Ils réduisent le nombre de collisions parasites détectées par les *Sensors*.
- Ils réduisent sensiblement le travail du moteur physique, puisqu'il doit calculer moins de collisions.

Les boutons *Collisions Group*(groupes de collisions) définissent à quel groupe l'objet appartient : c'est nous qui choisissons à notre guise la signification des groupes en sélectionnant des cases. Les boutons*Collision Mask* (masques de collision) définissent avec quels autres groupes d'objets notre objet sélectionné peut entrer en collision. Une collision ne peut avoir lieu entre deux objets que si au moins un groupe de l'un est également présent dans le masque de l'autre et réciproquement.

On peut donc définir ainsi des groupes d'objets qui ne réagissent qu'à certains autres groupes (ceux qui sont présents – case enfoncée – dans leurs masques). Par défaut, les objets appartiennent tous au premier groupe et entrent en collision avec tous les masques de groupes. Autrement dit, tous les objets se heurtent. Un cas pratique, pour un personnage joueur, serait par exemple d'avoir un groupe de collision qui détecte les PNJ amicaux, et un autre qui détecte les PNJ hostiles.

### Des dynamiques de forces pour les jeux avec des véhicules

Les paramètres de *Velocity* (vitesse),*Damping* (amortissement) et *Anisotropic Friction* (anisotropie frictionnelle) pour les objets de type *Dynamic*et *Rigid Body*vous permettront d'affiner la physique de vos véhicules, et sont particulièrement utiles dans des jeux de courses de véhicules.

## DES DÉTECTIONS DE COLLISIONS

## OPTIMISÉES AVEC LES *COLLISION BOUNDS*

Ce sous-onglet permet de définir la forme exacte du modèle physique de notre objet. Cela veut dire que le modèle physique ne doit pas nécessairement correspondre au modèle graphique. L'idée est de choisir une forme simplifiée pour le modèle physique pour simplifier considérablement le calcul des collisions mais qui correspond suffisamment bien au modèle graphique, pour conserver une cohérence entre les deux.

### Des formes physiques simplifiées

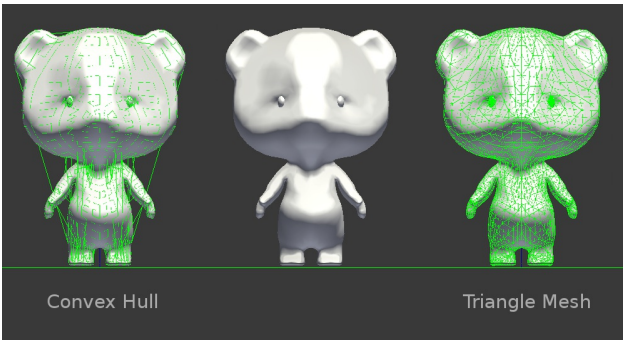
Le choix des formes simplifiées est limité : boîte, sphère, cône, cylindre, capsule (un cylindre complété par 2 demi-misphères). Dans le cas de la sphère, le rayon est déterminé par l'option `Radius`, dans le sous-onglet précédent, en-dessous de l'attribut `Mass`. Dans les autres cas, le *BGE* calcule automatiquement la dimension de la forme qui remplit au mieux le volume de l'objet.

### Les formes physiques complexes dites de type *Mesh*

Dans certains cas, une forme simplifiée n'est pas possible, par exemple pour le sol et les murs d'une pièce, ou un objet dynamique de forme complexe. Dans ce cas le modèle physique est dit de type *Mesh* (maillage) et se rapproche du maillage de l'objet.

Deux types de formes complexes sont disponibles :

- `Convex hull` : cette forme est appropriée pour des objets dynamiques, car plus simple. Le modèle physique « emballe » le modèle graphique pour former une surface convexe.
- `Triangle mesh` : à réserver au objets statiques dont c'est par ailleurs le choix par défaut, car le modèle physique va reproduire exactement le modèle graphique. Il y a donc la meilleure précision possible au prix de plus de calculs, d'où l'importance de simplifier au maximum le maillage de l'objet lorsque l'on choisit ce type.




### Corrigeons les imperfections avec une marge sur notre modèle physique

Quelle que soit la forme, simple ou complexe, nous pouvons de plus jouer sur le paramètre ***Margin***: c'est une épaisseur qui est ajoutée sur toute la surface du modèle physique pour l'augmenter. Cette marge nous permet de corriger les imperfections du moteur physique. Le but principal est d'éviter qu'un objet de petite taille ne passe à travers une paroi s'il est animé d'une grande vitesse. Par défaut la marge vaut 0.06, soit 6 centimètres. Mais il faut bien constater que le paramètre `Margin` ne résout pas tout : s'il y a vraiment un problème physique, il faudra probablement changer le type de physique de notre objet, ou jouer avec les groupes de collisions, comme nous l'avons vu plus haut.

## NOTE SUR L'USAGE DES UNITÉS DANS LE *BGE*

*Par défaut, le moteur physique est réglé de telle sorte que l'unité de longueur est le mètre (une unité Blender = 1 m) et l'unité de masse est le kilogramme. Définir une valeur réaliste pour votre objet donne en général de meilleurs résultats.*

# 20. DES PERSONNAGES AU BON COMPORTEMENT

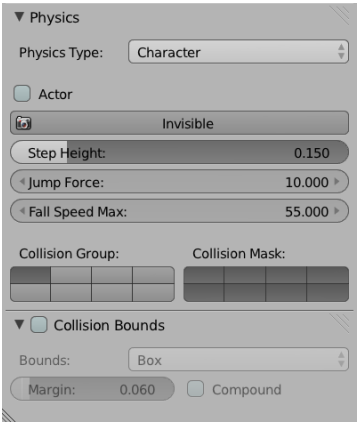
Dans le *Blender Game Engine*, il est apparu que les contrôles de physique et d'animation classiques n'étaient pas efficaces pour des entités adaptées à certains comportements courants, ceux des personnages de jeux et ceux des séquences interactives. Des options spécifiques sont prévues à cet effet dans le *BGE*.

## LA PHYSIQUE DES PERSONNAGES

Comme nous l'avons vu dans le chapitre précédent, la physique plus ordinaire des objets statiques et dynamiques ne convient en général pas pour les personnages-joueurs. Les objets statiques sont insensibles aux forces et les objets dynamiques ont aussi leurs problèmes.

- Ils peuvent être bloqués par de minuscules obstacles (par exemple un accroc dans le sol).
- Ils peuvent parfois traverser les parois, y compris le sol, s'ils ont une vitesse élevée.
- Les objets de type `Rigid Body` roulent et basculent facilement, ce qu'on peut vouloir faire pour un personnage, mais qu'il est plus facile de contrôler avec une animation.

Pour ces raisons, un type de physique dédié aux personnages a été spécialement prévu dans le moteur physique, le type ***`Character`***. Ce type est semblable à `Dynamic`, car il n'a pas de rotation mais il résout les problèmes cités précédemment et permet un contrôle fin des mouvements du personnage joueur et des PNJ via des options spécifiques.




***`Step Height`*** : hauteur de marche maximale que peut franchir le joueur dans son mouvement. Cela permet entre autres de grimper un escalier.

*Attention, la capacité à franchir un obstacle dépend également de la vitesse du personnage face à cet obstacle, plus la vitesse est lente et plus le franchissement sera difficile.*

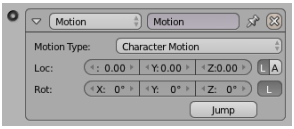
***`Jump Force`***: Littéralement force de saut. Le nom de ce paramètre est mal choisi, car il s'agit en fait d'une vitesse verticale (dont l'unité est le mètre par seconde) appliquée au joueur quand l'action `Jump` est exécutée via l'`Actuator` spécialisé (voir juste après). Une fois propulsé, le joueur retombe sous l'effet de la gravité.

***`Fall Speed Max`***: Ce paramètre règle la vitesse maximale que le joueur peut atteindre s'il tombe en chute libre, ce qui permet de simuler le frottement de l'air.

## USAGE DE CETTE PHYSIQUE DANS LES

## BRIQUES LOGIQUES

Une brique logique spécifique est prévue pour déplacer un joueur de type ***Character***, l'*actuator* ***Motion>Character Motion*** (Mouvement > Mouvement de personnage).




Il faut utiliser cet *actuator* de préférence en ***Motion>Simple Motion*** (mouvement simple) si nous voulons bénéficier de la physique ***Character***. Les options sont identiques, excepté l'option ***Jump***qui applique l'impulsion spécifiée dans ***Jump Force*** (évoqué précédemment). Le **`Simple Motion`**fonctionne avec le type **`Character`** mais ne permet pas de franchir les obstacles.

*Et attention, le* ***`Servo Control`*** ***ne*** *fonctionne* ***pas*** *avec le type* ***`Character`***.

## SCRYPTHON !

La brique logique ***Motion*** ne permet qu'un contrôle assez grossier du personnage : la vitesse est définie en incrémentant la position et est appliquée instantanément (il n'y a pas de phase d'accélération ni de décélération). Pour un contrôle fin, il faut accéder par Python à l'objet et utiliser les *API* spécifiques du type *Character*.

```
from bge import logic, constraints

# Recuper le controleur et son objet cont = logic.getCurrentController()
own = cont.owner

# Recupere le Character Physique
c = constraints.getCharacter(own)
```

Après import des modules nécessaires, nous récupérons le contrôleur courant (la brique logique qui a lancé le script). Ce contrôleur possède des *sensors* et des *actuators* reliés à lui mais surtout un **`owner`** (littéralement le propriétaire) : c'est l'objet qui possède la brique logique. Nous récupérons le **`owner`** de notre contrôleur dans une variable `own` car c'est lui qui contient tout notre objet, et donc aussi sa physique.

À la dernière ligne, nous récupérons l'expression python correspondante à la physique *character* de notre objet **`types.KX_CharacterWrapper`**, et nous avons ainsi accès à toutes ses méthodes et attributs.

- **`onGround`** : un booléen indiquant si le personnage est actuellement sur le sol.
- **`gravity`** : la valeur de la gravité utilisée par le personnage (par défaut : 29.4).
- **`maxJumps`** : le nombre maximum de sauts autorisés au personnage avant de devoir retoucher le sol (par défaut : 1 ; 2 pour un double sauts, *etc.*).
- **`jumpCount`** : le nombre de sauts en cours.
- **`walkDirection`** : la vitesse et la direction des déplacements du personnage en **coordonnées globales [x, y, z]**. On **ne** doit **pas** utiliser les autres méthodes de déplacement des objets classiques (`localPosition`, `worldPosition`, `applyMovement`, etc.).
- **`jump()`** : fait sauter le personnage.

Et voici un code complet de déplacement de personnage, avec une classe créée pour l'occasion :

```
from bge import logic, events, types, constraints
from mathutils import Vector

def keyDown(kevt):
 return logic.keyboard.events[kevt] == logic.KX_INPUT_ACTIVE
 def keyHit(kevt):
 return logic.keyboard.events[kevt] == logic.KX_INPUT_JUST_ACTIVATED

class Panda(types.KX_GameObject):
 def __init__(self, own):
 # facteur vitesse du personnage
 self.speed = .1
 # facteur sqrt(2) pour les diagonales
self.factor = 0.70710678
self.KX_Char = constraints.getCharacter(self)
def main(self):
self.move()
def move(self):
 # vecteur y = 0, 1 ou -1
 y = keyDown(events.UPARROWKEY) - keyDown(events.DOWNARROWKEY)
# vecteur x = 0, 1 ou -1
```

```
 x = keyDown(events.RIGHTARROWKEY) - keyDown(events.LEFTARROWKEY)
 # diagonal
 if x and y:
self.KX_Char.walkDirection = [x*self.speed*self.factor, y*self.speed*self.factor, 0.0]
# Nord, Sud, Est, Ouest
 elif x or y:
self.KX_Char.walkDirection = [x*self.speed, y*self.speed, 0.0]
else:
self.KX_Char.walkDirection = [0.0, 0.0, 0.0]
 # vecteur representant la direction de deplacement du personnage
vec = Vector((x, y, 0.0))
if x != 0 or y != 0:
 # on aligne le personnnage au vecteur
self.alignAxisToVect(vec, 0)

def main(cont):
own = cont.owner
if not "init" in own:
 own["init"] = True
Panda(own)
else: own.main()
```

1. Nous commençons, comme toujours, par importer nos modules.
2. Nous définissons deux fonctions qui nous seront utiles pour simplifier le code par la suite. **`keyHit(events.AKEY)`** renverra **`True`** si **`A`** vient juste d'être pressée (**`KX_INPUT_JUST_ACTIVATED`**). De même, **`keyDown(events.AKEY)`** renverra **`True`** si la **`A`** est enfoncée (**`KX_INPUT_ACTIVE`**).
3. Nous déclarons une classe **`Panda`** qui hérite de **`types.KX_GameObject`** (l'objet élémentaire du *BGE*).
4. Dans le **`__init__`** nous déclarons les attributs (nous récupèrons un **`KX_CharacterWrapper`** et nous le gardons dans le **`self.KX_Char`**).
5. Nous définissons la méthode **`main`**, qui contiendra toutes les autres méthodes appelées (ici il n'y en a qu'une, mais nous pourrions en imaginer d'autres comme **`attack`**, **`anim`**, **`heal`**, *etc.*).
6. Nous définissons la méthode **`move`**. Ici elle récupère les *inputs* (touches fléchées), déplace et oriente le joueur dans le bon vecteur correspondant. Arrive en dernier la fonction **`main`**, qui sera appelée par une brique logique.
7. Nous récupérons le **`owner`** (**`KX_GameObject`**). Si celui-ci n'a pas été initialisé, nous le remplaçons par **`Panda(own)`**, sinon le **`owner`** a maintenant été changé et on peut appeler sa méthode **`main`**.

Vous trouverez un exemple de fichier .blend de personnage animé à la fin du chapitre **Animer des personnages** dans **le Game Engine** de cette même section.

# 21. DES ANIMATIONS DANS NOTRE JEU

Dans un jeu, nous voulons nous amuser ! Des animations ajoutent de la vie et du *fun*, et de plus, dans un jeu, il est souvent essentiel de pouvoir déplacer certains objets de façon prédictible (ou « scriptée »).

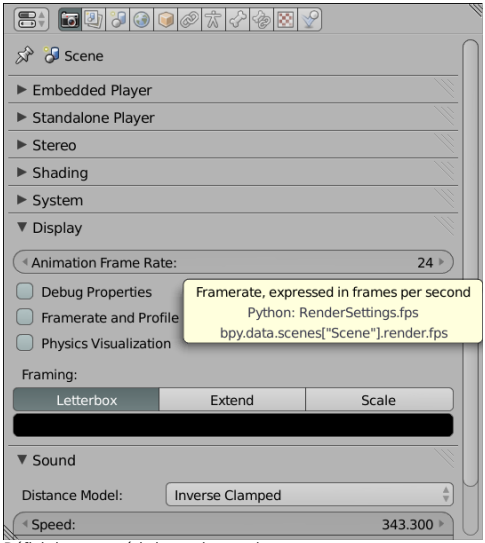
Blender possède un système d'animation permettant de changer une multitude de propriétés des objets (position, rotation, couleur, *etc.*), et également un système de *rigging* (de construction de squelettes) permettant de déformer un *mesh* (objet maillé) complexe à l'aide d'armatures (ou squelettes). Les deux systèmes sont complètement utilisables en dehors du moteur de jeu et la plupart de ces types d'animations fonctionnent également dans le moteur de jeu, à l'aide de l'***actuator Action***.

## LES IMAGES CLÉS D'ANIMATION (*KEYFRAME)* DANS BLENDER

Dans Blender comme dans la plupart des logiciels d'animation, les animations sont gérées par un système d'images-clés qui enregistre à un instant T la position, l'orientation et la taille des objets (ou des os −*bones*− dans le cas des armatures). Les images intermédiaires, qui ne sont donc pas définies précisément, sont calculées automatiquement par Blender (ce qui s'appelle du *tweening*).

L'image courante (*frame)* est indiquée entre parenthèses en bas à gauche dans le *viewport* de la vue 3D et elle est colorée en jaune si l'objet sélectionné dispose d'une *keyframe* (image-clé) sur la *frame* active. L'image de départ (*Start Frame)* par défaut est la *frame* 1, mais il vaut mieux utiliser la *frame* 0 pour nos animations car, par défaut, l'*actuator* `action` prend comme *frame* de départ 0. Pour changer la *frame* courante, il nous suffit d'appuyer sur les touches fléchées gauche et droite. Mais pour définir celle de départ il faut aller à la numéro 0 dans la *timeline* (ligne de temps) et appuyer sur **s** (pour *Start* - départ).

Ce qui va définir la vitesse de nos animations est le paramètre ***Animation Frame Rate*** (vitesse d'animation par image), défini dans le panneau `Render`sous l'onglet `Display`.



Définir les caractéristiques du monde

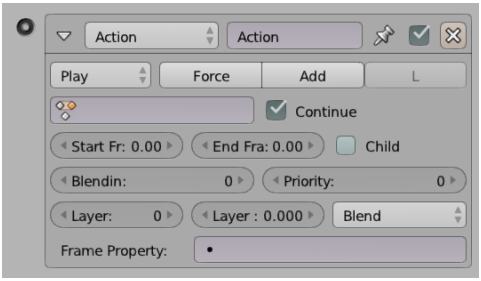
Ce *Frame Rate* est séparé de celui auquel le jeu se rafraîchit (voire le chapitre **Définir les caractéristiques du monde** de cette même section). L'animation sera jouée dans le jeu en respectant la *Frame Rate* de l'animation et non celle du jeu, à condition cependant que le jeu soit fluide, sinon la vitesse de l'animation est ralentie proportionnellement au ralentissement du jeu. Par défaut, elle est réglée à 24 image par seconde, qui est le standard que l'on retrouve au cinéma. Nous utiliserons les mêmes techniques d'animation pour faire un court-métrage dans Blender que pour faire un jeu utilisant le *Game Engine*. Pour plus d'informations sur l'animation par images-clés dans Blender, reportez-vous à un ouvrage dédié. Dans le *Blender Store* en particulier, il existe un livre de Tony Mullen en anglais *Introducing Character Animation, 2.5 edition*. Pour apprendre les concepts du *rigging*, le DVD de Nathan Vegdahl en anglais *Humane Rigging* est l'une des rares références les explicitant en utilisant Blender. Il est également disponible dans le *Blender Cloud*.

## RELIER NOS ANIMATIONS À NOTRE JEU AVEC L'*ACTUATOR ACTION*

L'*actuator* possède beaucoup d'options, mais les plus importantes sont :

1. La sélection de l'action à jouer.
2. La définition des images de début et de fin de l'animation avec `Start Frame` et `End Frame` pour contrôler la plage de lecture de cette action.
3. Le type d'animation.

Nous allons détailler ces types d'animations un par un.




### Les différents types d'animation

La première liste de l'*actuator* `Action` présente les différentes possibilités (ou types d'animations) offertes par Blender pour jouer les animations que nous auront réalisées ou récupérées.

***`Play`***

Joue l'animation du début jusqu'à la fin. Déclencher à nouveau l'*actuator* n'a d'effet que lorsque l'animation est terminée, il faudra donc attendre sa fin pour déclencher une autre animation du même type.

***`Ping Pong`***

Permet d'alterner des lectures à l'endroit et à l'envers : le premier déclenchement joue l'animation à l'endroit, le suivant à l'envers, etc. Comme pour le `Play`, on ne peut pas l'interrompre lorsqu'il est en cours.

***`Flipper`***

Joue l'animation tant que l'entrée est active, mais dès que l'entrée est à l'état logique Faux (**`False`**), l'animation se joue a l'envers pour revenir à l'état initial. Si l'entrée est réactivée alors que l'animation n'est pas revenue à l'image de départ (*Start Frame*), alors celle-ci reprend à l'endroit où elle était au moment de la réactivation.

***`Loop Stop`***

Tant que l'entrée est active, l'animation est jouée en continu, mais si l'entrée passe à Faux (**`False`**), alors l'animation se stoppe à l'image (*frame*) courante.

***Loop End***

L'animation se joue en continu du début à la fin tant que l'entrée est active, si l'entrée passe à Faux (**`False`**), l'animation ne se stoppe pas tout de suite, elle va jusqu'à la dernière *frame* d'animation puis s'arrête.

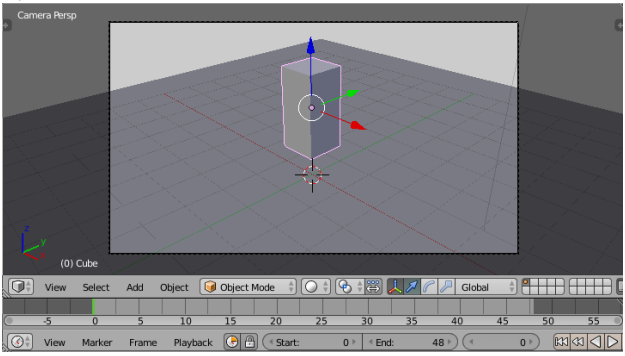
***Property***

Joue l'animation de l'image (*frame*) passée en paramètre de l'*actuator*, via une propriété du jeu `Property`, qui a été définie par ailleurs.

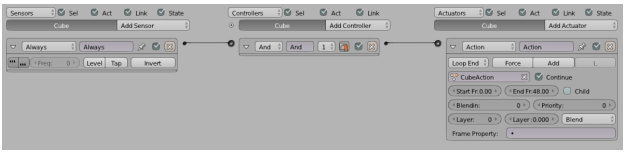
## NOTRE PREMIÈRE ANIMATION EN JEU

Nous allons créer un objet flottant, avec un mouvement vertical plutôt lent de la forme d'un cycle de montée et descente. Prenons un simple cube avec 3 *Keyframes* : à 0, 24 et 48. De cette façon, à 24 images par secondes, chaque étape prendra exactement 1 seconde. Sur ces 3 *Keyframes*, la position ou la rotation peut varier, mais les premières et dernières doivent être identiques pour que l'animation puisse boucler.

1. Élevons légèrement le cube par défaut et en `Edit mode`, réduisons de moitié sa largeur et sa profondeur avec les raccourcis claviers `SXY.5`.
2. Plaçons un plan au centre du monde et agrandissons-le. Nous percevrons ainsi un peu mieux le mouvement, en particulier grâce aux ombres.
3. Dans le *layout* `Blender Game`, éventuellement, créons une nouvelle vue par le bas dans la fenêtre centrale et choisissons l'éditeur de type ***`Timeline`***.
4. Changeons la valeur ***`Start`*** à `0` et ***`End`*** à `48`. L'intervalle d'animation apparaît en plus clair dans la ligne de temps. Zoomons dessus en tournant la molette pour qu'elle occupe bien l'espace disponible.
5. Passons à la première image de l'animation en utilisant le raccourci `Maj + flèche gauche` :0doit apparaître dans le champ placé à droite de **`End`**, et que nous aurions pu renseigner à la main.




6. Vérifions que le cube est bien sélectionné, plaçons nous dans la vue 3D et appuyons sur `i` puis choisissons `Location`, pour mémoriser l'emplacement du cube à cet instant de l'animation.
7. Déplaçons la *Timeline* à l'image 24 et modifions légèrement la hauteur du cube avec `g` puis `z` puis appuyons à nouveau sur `i` > `Location`.
8. À l'image 48, replaçons le cube à sa position initiale et faisons encore `i` > `Location`. Pour nous faciliter la tâche nous aurions pu faire l'image 48 avant l'image 24 puisque les deux ont la même position, ou nous aurions aussi pu dupliquer la clé avec `Maj + d` dans le`Dopesheet Editor`.
9. Nous pouvons jouer l'animation en faisant `Alt + a`. Appuyons sur `Echap` pour sortir.
10. Pour que notre animation joue en permanence dans le jeu, créons un ***sensor*** `Always`.
11. Créons un ***actuator*** `Action`de type ***`Loop end`*** et dans le champ situé en-dessous sélectionnons l'action`Cube Action`automatiquement généré par Blender.
12. Enfin, paramétrons le début et la fin des images à montrer :`Start Frame` doit être à 0 et`End Frame` à 48.




## TOUT PEUT S'ANIMER ! ANIMONS DONC UNE LAMPE

Gardez en mémoire que Blender est conçu pour pouvoir tout animer ! Ainsi Blender permet de créer des images clés (***Keyframe***) sur presque tous les paramètres, et il est possible d'enregistrer la valeur ***Energy*** d'une lampe au cours du temps en pressant la touche de capture ↓ lorsque notre curseur de souris survole le paramètre *Energy* d'une lampe. Une variation très simple de l'exemple d'animation ***Loop end*** précédent est d'animer une lampe avec les mêmes briques logiques, afin de la faire clignoter.

Cette option est très pratique pour la création d'atmosphère dans notre jeu, ou permet de mettre en évidence des objets ou des effets. Les animations et les lumières attirent le regard du joueur, combiner les deux est donc très efficace pour capter son attention !

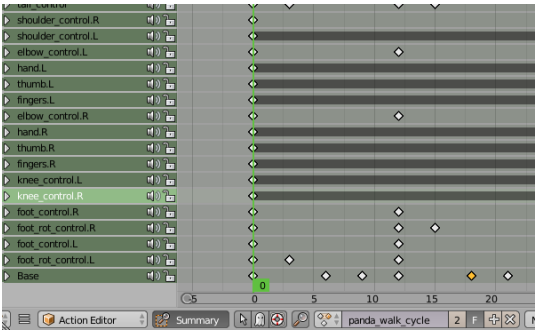
# 22. ANIMER DES PERSONNAGES DANS LE *GAME ENGINE*

Les objets animés, c'est intéressant, mais ce qui va vraiment enrichir notre jeu, c'est de donner vie à notre personnage ! Vous vous en doutez, il faudra utiliser une armature : grâce à elle, il va pouvoir courir, sauter et même danser. Nous devrons donc tout d'abord lui construire un squelette (***Armature***), composé d'os (***Bones***), puis lier notre maillage à ce squelette (***Skinning*** et ***Weight Painting***), comme nous pourrions le faire pour l'animation de personnages dédiés aux films. L'animation avec les armatures est donc identique à celle des objets, sauf que nous animons le maillage en déplaçant les *bones* ou leurs contrôleurs.

## CRÉER UNE BIBLIOTHÈQUE D'ANIMATIONS POUR NOTRE PERSONNAGE

L'idée principale pour les animations dans les jeux est de séparer nos animations en pistes différentes : une animation pour marcher, une animation pour sauter, une autre pour danser, etc. Cette méthode va nous permettre par la suite de mixer nos animations.

Le `Dopesheet Editor`va nous permettre de créer ces différentes actions qui pourront être jouées les unes à la suite des autres, voire même simultanément. Pour cela, nous devons aller dans l'`Action Editor` qui nous permettra de donner des noms aux différentes actions : en effet, il est indispensable des les nommer pour s'y retrouver par la suite. Ainsi, à chaque piste une animation : marcher, courir, sauter, etc.



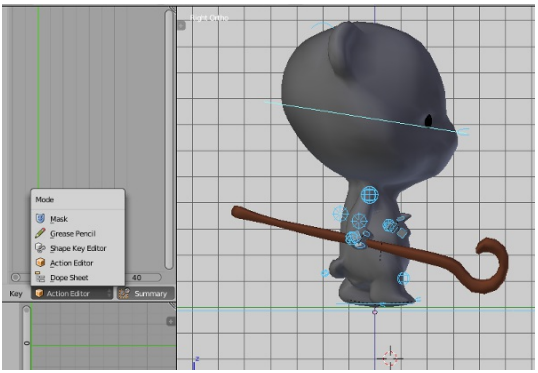

Donc à chaque nouvelle action du personnage, on crée une action dans le `Dopesheet Editor` qui sera appelée via l'*actuator*`Action`.

*Attention, quand on crée plusieurs pistes d'animation, il faut toujours cocher le petit F (Fake User*) à la droite du nom de l'animation, sinon celle-ci peut ne pas être enregistrée (si elle n'est pas utilisée).*

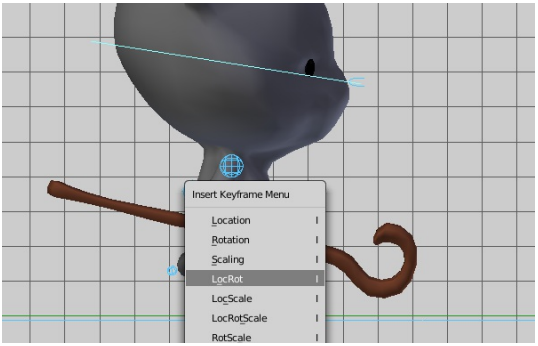
## UN EXEMPLE AVEC UN CYCLE DE MARCHE

Nous allons commencer par ajouter une action dans le `Dopesheet Editor`, que nous allons appeler *walk_cycle* (cycle de marche), qui sera le mouvement de déplacement du personnage. Tout d'abord, nous allons mettre le personnage dans une position de départ qui correspond à l'action. Dans le cas de la marche, nous ne pouvons pas commencer avec les deux pieds l'un à coté de l'autre sur le sol, car cela ne se peut pas lorsque nous marchons.

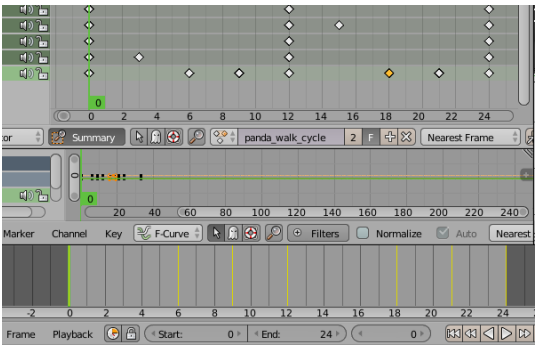
*Si vous n'avez jamais fait d'animation, ou si n'avez pas de connaissances théoriques sur l'animation, nous vous recommandons la lecture du livre de référence dans le genre : Techniques d'animation pour le dessin animé, l'animation 3D et le jeu vidéo de Richard Williams.*




Une fois dans la bonne position, nous allons sélectionner tous les *bones* et *shapes*de notre personnage et insérer une clé d'animation avec `Insert KeyFrame` (`i` dans la vue 3D), puis choisir `LocRot` (position et rotation).




Une fois la *KeyFrame* (clé d'animation) enregistrée, nous allons nous déplacer dans la *timeline* (ligne de temps) afin de poser d'autres *KeyFrames*, comme on le ferait pour une animation standard. Le `Dopesheet editor` nous permet d'affiner l'animation en dupliquant ou en supprimant des clés par exemple.



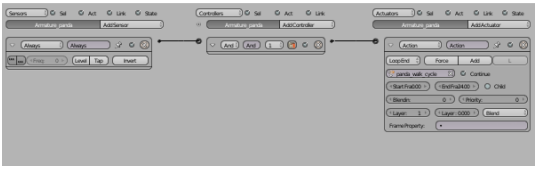

Une fois cette animation finie, nous pouvons en créer une autre à partir de la première en cliquant sur le + à coté du nom de l'animation. Cela duplique notre animation : nous pouvons alors la renommer et la modifier à votre guise.

Résumé en vidéo : vidéo

*Le `Dopesheet Editor` possède des outils comme les marqueurs, ou différentes visualisations des keyframes, qui aident à organiser l'animation.*

## LIRE L'ANIMATION DE MARCHE DANS LE JEU

Pour jouer une action avec les briques logiques, il nous suffit d'ajouter un *actuator* de type `Action` et de lui mettre en paramètre le nom de l'animation que l'on vient de créer, *walk_cycle*. Comme notre animation doit être jouée en continu jusqu'au relâchement du bouton, nous allons mettre son `Play mode` en `Loop End` (comme nous l'avons détaillé dans le chapitre précédent à propos des objets).



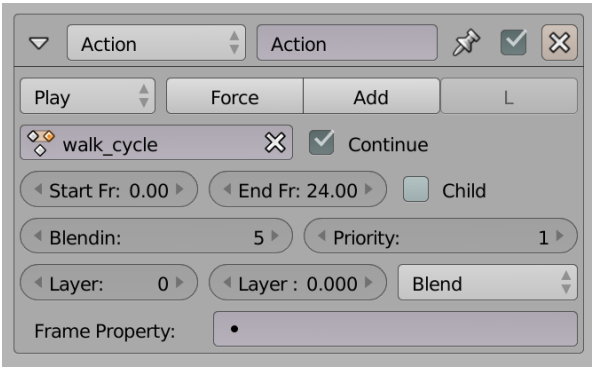
Ensuite, il nous suffit de brancher directement l'*actuator* d'animation a u *sensor* de la touche de déplacement, et notre personnage se déplace en marchant. Le *controller* de type `And` va être ajouté automatiquement.

## SI VOTRE ANIMATION NE SEMBLE PAS FONCTIONNER DANS LE JEU

Il est possible que vous ne voyez pas votre animation en lançant le jeu, et que le personnage ne bouge pas. Si c'est le cas, c'est probablement parce que le modifier `Armature` n'est pas le premier de la liste. Mais comme nous l'avons expliqué dans le chapitre **Spécificités du *Game Engine* dans Blender** de la section **Introduction**, il est conseillé d'appliquer vos modificateurs avant de publier votre jeu, afin d'avoir de meilleures performances.

## DE LA FLUIDITÉ ET DES TRANSITIONS ENTRE DEUX ANIMATIONS

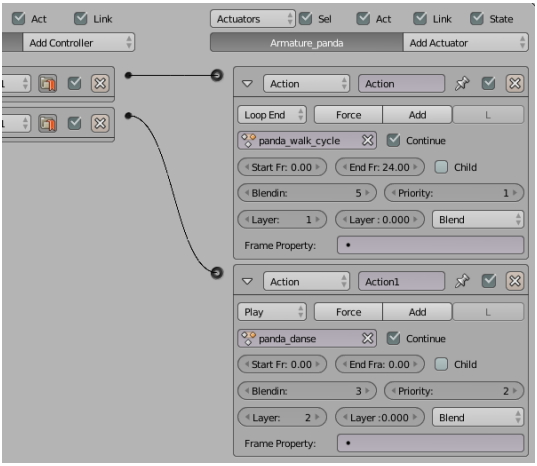
Blender est capable de changer d'animation de façon fluide. Cela rend les mouvements d'un personnage beaucoup plus naturels et évite l'effet saccadé (l'impression de sauts d'images) lors du changement d'animation. Pour les mettre en œuvre, nous allons avoir recours aux propriétés `Blendin` et `Priority` qui se trouvent dans l'*actuator* de type `Action`.




Le `Motion blending` (mixage) indique sur combien d'images (*frames*) la transition entre les deux animations se déroulera. Cela permet de bien lisser le changement. La priorité, quant à elle, sert à faire un choix : si deux animations sont actives en même temps sur une armature, c'est la priorité avec la valeur la plus faible qui passe en premier.

## ET MÊME PLUSIEURS ANIMATIONS SIMULTANÉMENT !

Pour jouer deux animations, par exemple pour gérer le haut et le bas d'un personnage séparément, il nous suffit de mettre deux *actuators* jouant les actions voulues et de mettre les actions sur différents calques (les *layers* de l'*actuator*). Nous pouvons utiliser jusqu'à 8 calques (*Layer* de 0 à 7).




## SCRYPTHON !

Nous pouvons également lire des actions en Python. Le script doit être lancé depuis une armature.

```
from bge import logic

def anim1():
    own = logic.getCurrentController().owner
    own.playAction("boy_idle", 4, 188, 1, 2, 6,
logic.KX_ACTION_MODE_LOOP)
    if not any(logic.keyboard.active_events):
        own.stopAction(6)
        own.stopAction(4)

def anim2():
    own = logic.getCurrentController().owner
    own.stopAction(1)
    own.playAction("boy_walk", 5, 62, 0, 1, 12,
logic.KX_ACTION_MODE_PLAY)
```

Ressource : animation_python.blend

L'idée est de jouer deux animations différentes : une animation de marche et une animation d'attente statique (*idle*). Nous allons jouer les animations sur des calques (*layers*) différents. On utilise pour cela la méthode `play Action` :

```
self.playAction(
                name,
                start_frame,
                end_frame,
                layer=0,
                priority=0,
                blendin=0,
                play_mode=KX_ACTION_MODE_PLAY,
                layer_weight=0.0,
                ipo_flags=0,
                speed=1.0,
                blend_mode=KX_ACTION_BLEND_BLEND
               )
```

On peut ensuite stopper toutes les animations jouées sur un calque avec `stopAction(layer)`.

## AJOUTER DES DÉPLACEMENTS

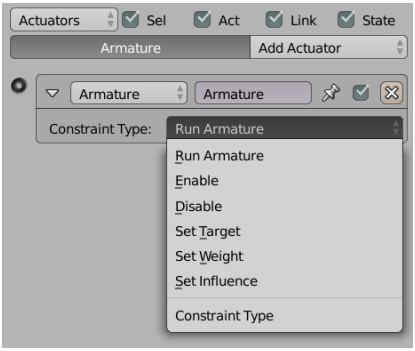
Nous venons de voir comment faire jouer des animations à un personnage, Nous pouvons maintenant lui ajouter les fonctions de déplacement vues dans le chapitre précédent **Les personnages** pour le terminer.

Ressource : personnage.blend

# 23. INVERSION DE CINÉMATIQUE DANS LE *GAME ENGINE*

L'***inverse kinematics*** (IK dans le reste de ce chapitre) ou en français inversion de cinématique est le terme consacré qui désigne le procédé mathématique pour calculer la pose d'une armature qui doit réaliser un objectif imposé à son extrémité (on parle de contrainte). Cette technique est généralement perçue comme un truc pratique pour faciliter l'animation : nul besoin de contrôler un à un tous les os d'une chaîne, il suffit de contraindre le dernier et l'algorithme se charge de disposer les autres.

Ce serait cependant une erreur de croire que l'IK dans le BGE se résume à une aide à l'animation. C'est aussi un moyen puissant pour faire interagir nos personnages avec l'environnement. L'IK peut être en effet exécuté indépendamment de toute action, il suffit pour cela d'utiliser l'*actuator* `Armature`:

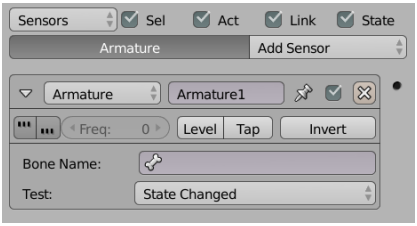



Le mode ***Run Armature*** exécute simplement l'IK avec les contraintes et les objectifs actuels (par défaut ceux qui étaient actifs dans le *3D view* avant de lancer le jeu).

*Si l'actuator n'est pas actif, l'IK n'est pas exécuté et l'armature garde la pose qu'elle avait au début du jeu ou lorsque l'actuator a cessé d'être actif. Cet actuator* ***n'est pas*** *nécessaire si l'IK est exécuté dans le cadre d'une Action (voir le chapitre* ***Des aminations dans notre jeu*** *de cette section).*

Les autres mode de l'*actuator* permettent de manipuler les contraintes d'IK : **Enable**, **Disable**, **Set Target**, **Set Weight**. Ces termes font directement référence aux paramètres de l'IK dans Blender : il est possible de les modifier dans le BGE en temps réel. L'intérêt de l'IK en tant qu'outil d'interaction est qu'il est tout à fait possible de choisir comme *target* un objet extérieur au personnage. Par exemple, si nous considérons un personnage avec une contrainte d'IK sur son squelette de tête qui lui fait de tourner la tête vers un objectif et si nous lui donnons comme objectif la caméra du jeu, nous obtiendrons un personnage qui regarde en permanence la caméra même si celle-ci se déplace.

Pour améliorer encore l'interaction, il existe un *sensor* `Armature` qui détecte des changements dans une contrainte de l'IK.




- ***State Changed*** : la contrainte est devenue active ou inactive. La sortie du *sensor* est l'état de la contrainte.
- ***Lin (Rot) Error bellow (above)*** : détecte que la contrainte remplit ou ne remplit pas l'objectif en angle ou en position. Cela permet de déclencher un comportement lorsque l'objectif est atteint ou est hors d'atteinte de l'armature. Dans l'exemple précité du personnage qui regarde la caméra, nous pourrions déclencher un mouvement si la tête atteint les limites des articulations. Note : ce test ne marche qu'avec iTaSC (voir ci-après).

Toutes les actions réalisées par ces deux briques sont accessibles par Python. Il est également possible de contrôler une armature entièrement par Python en donnant directement les angles de chaque articulation.

## ITASC COMME IK PRIVILÉGIÉ DANS LE BGE

Il existe deux algorithmes d'IK dans Blender : ***Standard*** et **iTaSC**. Les deux sont utilisables dans le BGE. Le premier est l'algorithme historique de Blender qui convient bien aux animations et aux armatures simples. iTaSC est un algorithme plus récent et plus élaboré qui possède des propriétés particulièrement intéressantes pour le BGE:

- Mémoire d'état
  Contrairement à `Standard`, iTaSC garde en mémoire l'état de l'armature d'une *frame* à l'autre. Cela garantit une bien meilleure continuité des mouvements pendant le jeu et une bien meilleure performance puisque quelques itérations de l'algorithme sont suffisantes pour suivre le déplacement de l'objectif depuis la pose précédente.
  *Note : nous devons choisir le mode **Simulation** pour obtenir ce comportement.*
- Multi-contraintes
  Contrairement à`Standard` qui ne considère véritablement que la dernière contrainte d'une armature (la contrainte la plus éloignée du début de la chaîne), iTaSC est capable de satisfaire simultanément plusieurs contraintes sur la même armatures. Si ces contraintes ne peuvent pas être toutes réalisées en même temps, iTaSC réalise un compromis que nous pouvons régler avec les paramètres **`Weight`**.
  *Note : une limitation dans iTaSC impose que toutes les chaînes d'os contraints qui ont des os en communs partagent le même os d'origine. Dans ce cas il n'est pas absurde que la totalité de l'armature soit contrainte.*
- Contraintes exotiques
  Au contraire de *Standard* qui ne peut résoudre qu'une contrainte de position (et optionnellement d'orientation), iTaSC accepte une plus grande variété de contraintes :
  - contrainte de position et d'orientation partielle : seules certaines composantes de la position et de l'orientation sont contraintes avec le choix de la référence (armature ou objectif). Cela permet d'intéressants effets d'alignement sur un plan, sur un axe ou de parallélisme.
  - contrainte de distance : l'os est maintenu à une certaine distance de l'objectif.
  - contrainte d'articulation : un effet de ressort peut être appliqué sur les articulations pour les contraindre à rester au plus près de la pose de repos.

  D'autres types de contraintes pourraient être ajoutés dans le futur car iTaSC introduit un concept de contrainte généralisée.
- Armature complexe
  Contrairement à *Standard*, qui donne des résultats souvent médiocres (lent et incorrect) quand il est utilisé sur une armature entière, iTaSC produit généralement des poses correctes et permet un contrôle global d'une armature avec un minimum de points de contrôle.
- Dynamique réaliste
  Bien qu'iTaSC soit un algorithme cinématique (sans notion de force ou d'inertie), on peut obtenir un comportement pseudo dynamique en jouant avec le paramètre **`Max Velocity`** qui limite la vitesse de rotation des articulations : on obtient une armature « lente » qui peut prendre du retard sur l'objectif.
- Retour d'information
  Contrairement à *Standard*, iTaSC produit un retour d'information qui permet d'évaluer si une contrainte est réalisée ou non et dans quelle mesure. Cette information est utilisable comme entrée dans le *sensor* `Armature` prévu à cet effet.

*Malheureusement, un bogue interdit d'utiliser iTaSC dans le BGE dans les versions 2.70 et 2.71. Ce bogue est déjà corrigé mais la correction ne sera disponible qu'à partir de la version 2.72.*

Il sort du cadre de ce livre d'expliquer iTaSC en détail. Une documentation détaillée est disponible en anglais:
http://wiki.blender.org/index.php/Dev:Source/GameEngine/RobotIKSolver

Ce .blend fait la démonstration du mode interactif d'iTaSC : l'objet central se déforme et tourne sur lui-même si nécessaire pour suivre l'objectif. [lien vers : game_armature.blend].

# 24. GÉNÉRATION D'OBJETS

Dans ce chapitre, nous allons voir comment générer des objets à la volée. Typiquement, c'est le cas du gorille qui lance toutes les 10 secondes un tonneau vers notre héros, des caisses de munitions que l'on retrouve à intervalle régulier dans l'arène, ou des lemmings qui arrivent par groupe de 20 dans notre grotte, etc. En somme, il n'y avait rien, puis soudainement, quelque chose est là.

## DÉFINITION DU *SPAWNING*

Le fait de créer un objet ou un personnage en direct et en cours de jeu est ce que l'on appelle, dans le jargon du jeu vidéo, du *spawning*. L e *spawning* est donc le fait de typiquement faire apparaître un élément de jeu comme s'il venait d'être créé. Ces éléments existent donc bien dans l'univers du jeu et ont été créés par les concepteurs, mais ils n'apparaissent à l'écran et n'interagissent avec l'environnement du jeu qu'à partir du moment où la logique du jeu le nécessite. L'avantage des méthodes de *spawning*, c'est que l'on peut faire apparaître tout le temps de nouvelles copies d'objets, indéfiniment ou tant que c'est nécessaire.

Dans un jeu de survie par exemple, dont le but est de rester en vie le plus longtemps possible devant une attaque de zombies, peu importe combien de zombies tuera le héros, il y aura toujours de nouveaux zombies qui apparaîtront à l'horizon. En pratique, le créateur de jeu aura donc défini plusieurs modèles de zombies possibles (avec des apparences, des capacités et des modes de déplacement différents). Puis la logique de jeu ira pêcher dans cette bibliothèque d'ennemis pour créer cette invasion constante de morts-vivants.

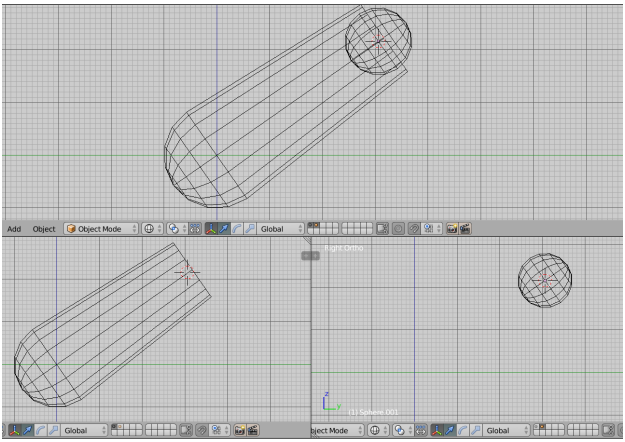
Ce qu'il faut comprendre ici, c'est que nous avons donc un modèle, un objet de référence, et le *spawning* est la technique qui va permettre de créer des copies de ce modèle aux bons endroits du jeu. Toutes les copies, qu'on peut comprendre comme des clones, ou qu'on appelle aussi des instances, ont leur vie propre à partir du moment où elles sont créées. Ces instances, bien qu'elles partagent les mêmes propriétés lors de leur création, évoluent de façon indépendantes une fois dans le jeu.

Pour ce chapitre, nous utiliserons un exemple très simple d'un canon qui tire des boulets lorsque le joueur appuie sur `la barre d'espace`. Le nombre de boulets est infini. Le joueur peut en tirer autant qu'il veut.

## PLACER DES OBJETS DANS UN CALQUE NON VISIBLE

Préparons notre scène de manière à être plus à l'aise au moment de la génération. Créons les objets de base et organisons les de manière à y accéder facilement sans qu'ils gênent le jeu. De plus, il est impossible pour le BGE d'ajouter à la volée des objets présents dans un calque actif, il faudra en tenir compte.

1. Dans notre scène, de manière visible, nous n'avons qu'un canon, représenté par un cylindre. Nous allons ranger dans un calque (*layer*) non visible du projet, les éléments que nous ne voulons pas voir apparaître dans le jeu, mais que nous désirons générer à la volée. Dans ce cas précis, il s'agira de nos boulets de canon. Si nous oublions cette étape, le BGE ne sera pas capable de générer l'objet et renverra une erreur.
2. Vérifions les propriétés de physique du canon et assurons-nous que `Ghost` est coché de manière à ce que le canon ne retienne pas les boulets.
3. Créons maintenant un objet sphère qui sera notre boulet de canon. Nous donnons à cet objet le nom : Boulet. Cela simplifiera la manière de le retrouver lorsque nous voudrons créer de nouveaux boulets pendant le jeu.
4. Concernant toujours notre objet Boulet, dans l'onglet `Physics` de la fenêtre `Properties`, nous lui donnons une physique de type `Dynamic` afin que notre boulet se comporte comme un projectile réel.
5. Revenons ensuite dans notre calque principal où nous avons placé notre canon. Assurons-nous que le calque où se trouve le boulet est bien désactivé ou, autrement dit, invisible.



## GÉNÉRER DES INSTANCES

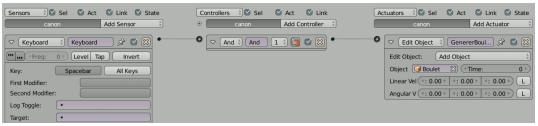
Nous allons attacher à notre objet canon la logique qui permet de tirer des boulets de canon lorsque le joueur appuie sur la `barre d'espace`.

### Créer un objet

1. Sélectionnons le canon et ajoutons un *`sensor`* de type *`keyboard`* que l'on configure pour s'activer lorsque l'on appuie sur la `barre d'espace`.

2. Relions-le à un *`controller`* de type *`And`* que l'on relie ensuite à un *`actuator`* de type ***`Edit Object`***.

3. Par défaut, cet *`actuator`*est configuré pour ajouter des objets à la scène. C'est le mode ***`Add Object`***. Nous devons ensuite lui spécifier le nom de l'objet qu'il doit ajouter à la scène. En cliquant dans la case après *`Object:`*, nous retrouvons facilement notre Boulet, nommé précédemment. Nous rappelons que cet objet doit être sur un calque invisible pour que cela fonctionne.



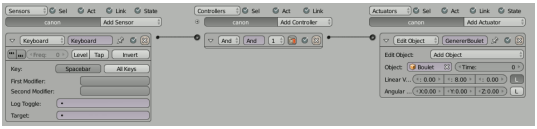
### Donner un mouvement

Une instance sera toujours générée aux coordonnées d'origine de l'objet qui la crée. Dans ce cas-ci, nos boulets seront générés au milieu de notre canon.

4. Si besoin, replaçons l'origine pour que les boulet partent bien de l'emplacement souhaité (pour cela, nous pouvons positionner correctement le boulet dans le canon, y caler le curseur 3D avec `Maj + s` > ***`Cursor to selected`***, puis sur le canon, utiliser ***`Set Origin > Origin to 3D cursor`*** de la boîte à outil.

5. Nous pouvons donner une vitesse de départ à nos instances en modifiant les paramètres ***`Linear Velocity`***. Dans notre exemple, nous allons appliquer une vitesse de **8.00** en **y** selon les coordonnées locales du canon en cochant le bouton `L` en bout de ligne. Pour bien choisir les axes, passez la vue 3D en mode local également.

Nous avons choisi la coordonnée locale du canon parce que nous pourrions ensuite placer ce canon sur une tourelle qui tourne sur elle-même ou faire tenir ce canon par un personnage se déplaçant. Nous nous assurons ainsi que les boulets sortiront toujours par le bon côté du canon et iront dans la direction vers laquelle il pointe.




<lien vers blend>

### Donner une durée de vie

Toujours dans cet *actuator*, nous voyons l'option ***Time***. Il s'agit de la durée de vie que nous allons donner à chaque instance que nous allons générer. Au delà de cette durée, l'instance est détruite. Cela peut servir dans plusieurs cas comme simplement celui de limiter dans la durée la présence de cette instance dans le jeu. Mais, au delà des règles du jeu, limiter cette présence peut devenir réellement important si l'on compte, par exemple, générer un très grand nombre d'instances. Parce que si aucun autre mécanisme ne vient les détruire, elles resteront présentes dans le moteur de jeu tant qu'on n'est pas sorti du jeu ou qu'on a pas changé de scène. Cela peut donc devenir rapidement très lourd pour le moteur physique de gérer autant d'objets. Donner une durée de vie limite permet ainsi de conserver un nombre d'instances raisonnable par rapport aux performances de la machine.

Par défaut, l'`actuator` a une valeur de `Time` égale à 0. Cela veut dire que l'instance générée aura par défaut une durée de vie infinie.

## SCRYPTHON !

L e *spawning*peut bien évidemment se faire en Python. Gardons notre`sensor Keyboard`défini précédemment mais relions le plutôt à un `controller` de type Python pointant vers un script nommé tirer_un_boulet.py. Dans ce script nous écrirons.

```
from bge import logic

# Recuperer la scene

scene = logic.getCurrentScene()

# Recuperer le canon

canon = scene.objects["Canon"]

# Recuperer le boulet sur son calque invisible

boulet = scene.objectsInactive["Boulet"]

# Creer une instance de boulet

instance = scene.addObject(boulet, canon, 0)

# Donner une vitesse à l'instance

instance.localLinearVelocity.y = 8.0
```

Après import du module `logic` et récupération de la scène active, nous créons deux variables qui représentent nos objets "canon" et "boulet" présents dans la scène. Remarquons qu'on accède différemment à des objets se trouvant dans un calque non visible : `scene.objectsInactive["Boulet"]` qu'à des objets présents dans un calque visible : `scene.objects["Canon"]`. Dans les deux manières, nous utilisons le nom le l'objet précédemment défini.

Nous créons ensuite des instances de notre objet *Boulet* qui apparaîtront aux coordonnées de l'origine de notre objet canon de cette façon :

```
instance = scene.addObject(boulet, canon, 0)
```

Le troisième paramètre, ici mis à 0, est la durée de vie de l'objet. Observez aussi que cette méthode `addObject()` renvoie l'objet nouvellement créé. Nous conservons ceci dans une variable nommée instance afin de pouvoir lui appliquer des fonctions supplémentaires, comme dans ce cas-ci lui ajouter une vitesse.

```
instance.localLinearVelocity.y = 8.0
```

L'avantage d'utiliser Python ici par rapport aux briques logiques, c'est que l'on pourrait dynamiquement changer la durée de vie de nos instances, générer des instances différentes pêchées aléatoirement dans une liste d'objets cachés, ou encore la faire apparaître à des endroits différents de la scène en utilisant comme point de génération des objets vides (*empty*) placés à des endroits clés de notre scène de jeu.

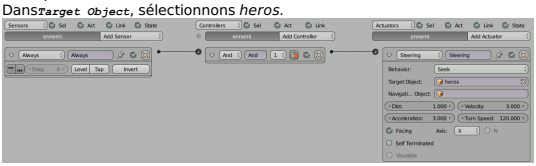
# 25. RECHERCHE DE CHEMIN AUTOMATIQUE

Si nous voulons donner à nos PNJ* une intelligence, histoire de corser le jeu, la première chose à faire est de leur permettre de trouver leur chemin dans le niveau. Heureusement cette tâche ardue est grandement facilitée par le logiciel ***Recast & Detour*** de Mikko Mononen (https://github.com/memononen/recastnavigation) qui est intégré dans Blender et dans le BGE.

## SUIVRE LE HÉROS

Une action fréquente de jeu est de faire des poursuites ou de mettre le héros en danger. Le principe est bien sûr de complexifier le jeu, d'en augmenter l'intérêt en contraignant le joueur à comprendre la logique de personnages non-joueurs (PNJ). Avant de passer en revue les possibilités de l'assistance automatisée aux déplacements, commençons par un premier exemple qui va permettre de prendre les choses en main. Il s'agira simplement de suivre le héros sur une scène. Pour cela nous utiliserons l'*actuator* ***steering***.

1. Supprimons le cube par défaut avec `x` puis ajoutons un plan `Maj + a`***>Plane***.
2. Attribuons un matériau de base de couleur verte.
3. Créons une pyramide à partir d'un Cube `Maj + a`***>Cube***, sélectionnons les *vertices* de l'un des côtés et fusionnons les avec `Alt + m`***>At center***. Attribuons un matériau rouge à cet objet et nommons-le ennemi.
4. Ajoutons à présent le héros avec une simple sphère à laquelle nous pouvons attribuer un matériau jaune et le nom *heros*.
5. Redimensionnons `s` et déplaçons `g` ces 2 personnages de manière à les espacer sur le terrain de jeu.
6. Les bases du suivi étant à présent réalisées, sélectionnons l'objet ennemi puis passons en mode *Blender Game* si ce n'était pas encore le cas.
7. Créons un *sensor* ***Always*** et relions-le à un *actuator* ***Steering***.
8. Dans la liste ***Behavior*** (comportement), vérifions que ***Seek*** (chercher) est bien sélectionné. Cette option définit l'option de base sur laquelle se basera le personnage pour rechercher sa cible. Ici, il fait une recherche directe à l'inverse de *navigation Path* que nous verrons ultérieurement.
9. Dans ***Target Object***, sélectionnons *heros*.

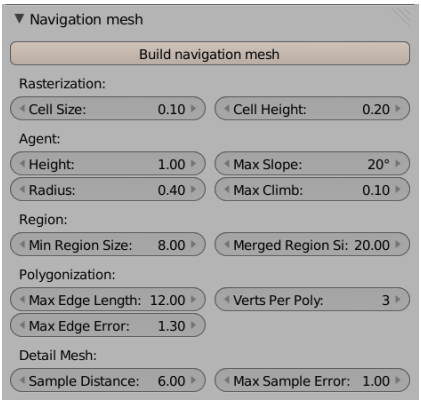



10. Vous pouvez tester le jeu. L'ennemi doit se diriger vers le héros.
11. Maintenant, lorsqu'il l'atteint, il s'arrête un peu trop tôt car il ne le touche pas vraiment, ce qui n'est pas réaliste. Dans les paramètres de l'*actuator*, jouons sur les paramètres du ***steering***. Dans notre cas diminuons ***Dist*** à une valeur qui rendra correctement en fonction des dimensions de nos objets.
12. Vous pouvez enfin rendre le héros déplaçable à la souris et les choses deviennent alors très intéressantes.

## CONSTRUIRE UN PLAN DE NAVIGATION

*Nous utiliserons ce blend d'exemple comme support de ce chapitre : [recherche_de_chemin_automatique.blend].*

La partie ***Recast*** s'exécute dans Blender et consiste à construire un plan de navigation (***Navmesh*** dans le jargon Blender) à partir des différents objets qui constituent la partie fixe du niveau (les murs, le sol, les escaliers, les colonnes, etc). Dans notre exemple, il n'y a qu'un seul objet qui constitue le niveau, mais on pourrait en avoir plusieurs, il suffit juste de les sélectionner tous.




Ensuite nous ouvrons le panneau ***Navigation Mesh***(***Properties > Scene***) et nous choisissons les options en fonction des caractéristiques de notre PNJ. Il y a de nombreuses options mais les principales sont :

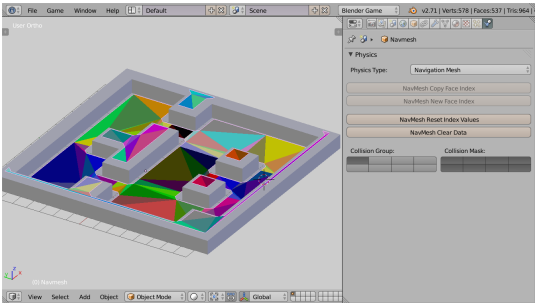
- ***Polygonization : Verts Per Poly***. Cette option **doit** être mise à **3** pour que l'algorithme de nagivation fonctionne correctement.
- ***Agent : Radius***. Spécifiez ici le rayon du PNJ en considérant le cylindre qui englobe son mesh visuel. L'algorithme utilise cette donnée pour ménager une marge par rapport aux murs du niveau de sorte que le PNJ ne se cogne pas au mur.
- ***Agent*** **: autres attributs**. Les autres options*Agent*définissent plus précisément les capacités de notre PNJ. Quel pente est-il en mesure de grimper ? Quelle hauteur d'obstacle peut-il franchir ? de quelle hauteur de plafond minimale a-t-il besoin pour entrer dans un espace ? Ces options sont à régler en fonction de la taille du *mesh* et des caractéristiques physiques de l'objet que nous avons choisies (voir le chapitre Des personnages au bon comportement de cette section).

Les autres options sont très techniques et généralement configurées correctement. Si besoin, référez vous à la documentation de l'algorithme *Recast* avant d'y toucher. Ensuite nous cliquons sur ***Build navigation mesh*** et un nouveau *mesh* est ajouté dans la scène. Ce *mesh* se nomme *navmesh* par défaut et a les caractéristiques suivantes :

- Il est constitué de triangles qui couvrent la surface accessible à un agent ayant les caractéristiques précitées.
- En mode solide les triangles sont colorés automatiquement pour que nous puissions voir clairement la position des vertex car ce seront les points de passage du PNJ dans le jeu.
- Son type physique est mis à ***Navigation Mesh***, ce qui, soit dit en passant, active la coloration automatique.
- Il peut être divisé en plusieurs parties non connectées. Dans notre exemple, des surfaces de navigation sont créées aux sommets de certains murs. Ce n'est pas gênant pour notre PNJ.

Si nous ne sommes pas satisfaits du résultat, nous pouvons soit modifier le *mesh* à la main comme tout *mesh* (en veillant à n'ajouter que des faces triangulaires), soit supprimer l'objet et en relançant l'utilitaire après avoir modifié les paramètres (et ré-sélectionné tous les objets).

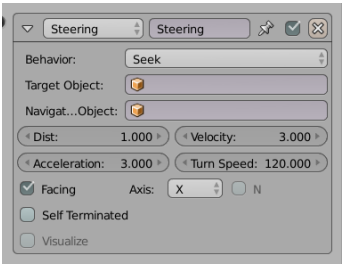
Il est tout à fait possible de créer plusieurs plans de navigation, pour différentes scènes, voire pour la même scène mais pour différents agents ayant des caractéristiques différentes. Les briques du BGE qui contrôlent la navigation permettent de définir quel *navmesh* sera utilisé par quel PNJ.




*Le **Navmesh** n'est visible que dans la Vue 3D (3D View), il est automatiquement invisible dans le jeu. Le panneau `Physics` d'un Navmesh montre des boutons spécifiques à ce type d'objets. Au cas ou nous aurions modifié le mesh manuellement, il est indispensable d'utiliser le bouton `NavMesh Reset Index Values` pour s'assurer de la cohérence du mesh.*

## UTILISER UN PLAN DE NAVIGATION PAR BRIQUE LOGIQUE

L'*actuator* `Steering` permet d'utiliser un *mesh* de navigation dans le jeu.




Lorsque cet *actuator*est activé pour un objet, il en prend le contrôle et le déplace automatiquement jusqu'à atteindre l'objectif qui est la position de l'objet spécifié dans `Target Object`. Nous utiliserons le terme **agent** pour désigner un objet qui passe sous le contrôle d'un *actuator* `Steering`.

*Si l'objectif se déplace, l'actuator actualise la trajectoire de l'agent, c'est idéal pour un comportement de poursuite.*

L'option `Behavior` détermine le comportement de l'agent :

- `Seek`: l'agent cherche à atteindre l'objectif en ligne droite sans tenir compte des obstacles et du *navmesh* (nous pouvons omettre de spécifier un *navmesh* dans ce mode).
- `Flee` : cette option est l'exacte opposée de `Seek`, l'agent cherche à fuir en ligne droite en partant dans la direction opposée de la cible.
- `Path Following` : l'agent utilise le *navmesh* spécifié dans `Navigation Mesh` pour trouver le meilleur chemin pour atteindre l'objectif. Ce comportement est celui qui nous intéresse car il donne une réelle intelligence à l'agent.

Les autre options définissent les aspect dynamiques du mouvement :

- **Dist** : la poursuite s'arrête si l'agent s'approche de l'objectif à une distance inférieure à cette valeur (en unités Blender, c'est-à-dire en mètres).
  *La poursuite pourra éventuellement reprendre si l'objectif se déplace à nouveau. L'actuator ne définit pas en soi ce qui se passe lorsque l'objectif est atteint, c'est la logique du jeu qui le fait.*
- **Velocity** : l'agent poursuit l'objectif à cette vitesse (en m/s).
  *Malheureusement, aucune accélération n'est prévue dans l'actuator, l'agent se met en mouvement instantanément.*
- **Acceleration** & **Turn Speed** : ces options sont utilisées uniquement pour l'évitement des obstacles mobiles que nous n'aborderons pas ici.
- **Facing/Axis/N** : ces options définissent l'orientation de l'agent pendant la poursuite. Pour obtenir un agent qui se tourne dans le sens du chemin, nous cocherons l'option **Facing** et choisirons dans **Axis** l'axe local qui pointe vers l'avant de l'agent. L'option **N** n'est utile que si le navigation *mesh* n'est pas horizontal.
  *De même que pour la vitesse, aucune rotation progressive n'est prévue. L'agent se réoriente instantanément à chaque angle de la trajectoire.*
- **Self Terminated** : en cochant cette option l'*actuator* se désactive automatiquement si l'objectif est atteint. Autrement dit, la poursuite s'arrête même si l'objectif se déplace à nouveau hors de la distance d'arrêt. Cette option est très utile pour faire une logique chaînée à peu de frais. Cette technique est expliquée plus loin dans ce chapitre.
- **Update period** : cette option définit intervalle de temps en millisecondes entre deux re-calculs de la trajectoire pour tenir compte du déplacement de l'objectif. La valeur par défaut est 0, ce qui implique un re-calcul à chaque *frame*. Ça ne dérange pas pour notre petit test, mais pour des jeux plus complexes avec beaucoup d'agents, c'est très certainement excessif. Une valeur de 100 à 500 est plus raisonnable, à tester au cas par cas.
- **Visualise**: Permet de visualiser en temps réel dans le jeu la trajectoire de l'agent sous la forme d'une ligne rouge.

## UN EXEMPLE DE COMPORTEMENT COMPLEXE PAR BRIQUES LOGIQUES

Nous savons maintenant comment faire un agent qui poursuit inlassablement un objectif qui se déplace dans un niveau (dans notre jeu, ce sera le piège qui poursuit le panda) mais nous aimerions lui donner un comportement plus typique d'un méchant dans les jeux. Nous voulons que l'agent fasse une ronde mais que si l'objectif s'approche à une certaine distance, différente par devant et par derrière, il se mette à sa poursuite jusqu'à ce que l'objectif soit atteint ou qu'il s'éloigne au-delà d'une certaine distance, auquel cas l'agent reprend sa ronde.

### Utilisation d'une machine à état

Nous allons utiliser les notions de machine à état que nous avons vues au chapitre précédent pour modéliser l'agent que nous venons de décrire. Dans notre exemple nous tenterons de nous limiter à deux états plus un état terminal :

- la ronde ;
- la poursuite ;
- la fin du niveau.

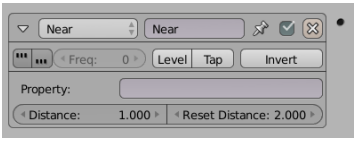
Les événements sont :

- 1->2 : l'objectif s'approche assez près de l'agent ;
- 2->1 : l'objectif est trop loin de l'agent ;
- 2->3 : l'agent atteint l'objectif.

*Les termes assez près et trop loin sont volontairement flous car ils n'influent pas sur la logique. Nous réglerons les valeurs précises dans les sensors de détections que nous utiliserons pour produire les événements.*

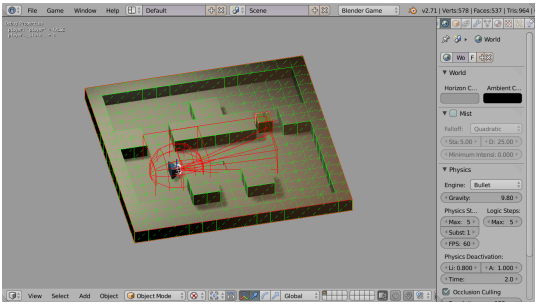
### Événement de proximité par brique logique

Les détections de proximité, dont nous avons besoin pour produire les événements de notre machine à état, seront effectuées par des *sensors* **Near** et**Radar**. Le *sensor* **Near** détecte la présence d'objet dans un rayon déterminé par le paramètre **Distance**.




Le *sensor* `Radar`détecte la présence d'un objet dans un cône orienté dont la pointe est centrée sur l'objet. On peut choisir l'ouverture du cône avec le paramètre `Angle`, la hauteur avec le paramètre `Distance` et l'orientation avec l'option `Axis` (les axes sont ceux de l'objet). Bien orienté, ce type de *sensor* simule la vue d'un agent.

*Les sensors* ***Near*** *et* ***Radar*** *ne détectent que les objets dont l'option* ***Actor****est activée dans le panneau Physics ! C'est un oubli fréquent qui peut causer des soucis. Si nous avons un doute sur la taille et l'orientation des zones de détections, il nous suffit d'activer la visualisation de la physique (menu principal*`> Game > Show Physics Visualization`*).*



Ces deux *sensors*peuvent être rendus très sélectifs grâce au paramètre `Property`. En mettant un nom dans ce champ, seuls les objets qui ont une propriété (`Game Property`) de ce nom seront détecté par le *sensor*.

### Comment faire exécuter une ronde à un agent, à peu de frais ?

Pour les états 1 et 2 nous pouvons utiliser l'*actuator*`Steering` pour déplacer notre agent, il suffit de choisir l'objectif : pour la ronde ce sera un simple ***Empty*** qui sert de point de passage (inutile de surcharger le BGE avec un objet solide alors que nous avons juste besoin d'une balise) et pour la poursuite, ce sera le joueur (avec une vitesse plus rapide pour simuler la course). Cependant pour la ronde, un problème se pose: un seul point de passage n'est pas suffisant, il en faut au moins 3 pour faire une boucle (2 pour un aller-retour). Nous avons donc besoin de 3 *actuators* (1 par point de passage) et nous devons trouver le moyen de les activer en séquence pour que l'agent aille d'un point de passage à l'autre.

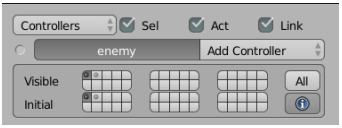
*A priori* il nous faudrait plus d'états pour réaliser cela mais une solution élégante est possible grâce au *sensor*`Actuator`. Il s'agit d'un *sensor* qui surveille l'état d'un *actuator* en particulier (d'où son nom) et il produit une impulsion logique, positive ou négative, lorsque l'*actuator* en question devient respectivement actif ou inactif.

*L'actuator sous surveillance est nécessairement un actuator de l'objet. On le sélectionne par son nom, d'où l'intérêt de choisir des noms parlants pour les briques logiques.*

Ce type de *sensor* est générique, il fonctionne avec tous les *actuators*, en particulier avec ceux qui se désactivent spontanément comme `Action` et `Constraint > Force Field`. Il permet d'une manière générale de chaîner les actions: lorsqu'un *actuator* termine son travail, un autre peut s'activer grâce au *sensor*.

### États visibles et initiaux

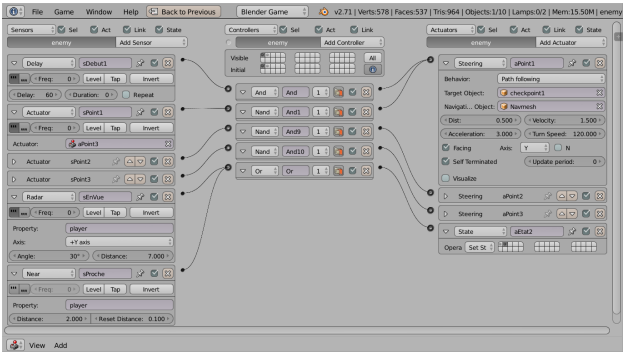
Si nous changeons le *state* d'un *contoller*sans autre précaution, nous aurons la surprise de le voir disparaître ainsi que toutes les briques auxquelles il était connecté. Pas de panique, les briques n'ont pas été supprimées, elles sont simplement invisibles car Blender n'affiche par défaut que les briques de l'état 1. Nous pouvons choisir quels états seront affichés grâce à l'outil de sélection des états. L'outil est lui-même invisible par défaut, il faut cliquer sur le petit▫à coté de l'entête des *controllers* pour le voir.




Les boutons en face de `Visible` indiquent quels sont les états actuellement affichés et ceux en face d'`Initial` indiquent les états actifs au démarrage du jeu (il en faut au moins un). Dans notre exemple, ce sera l'état 1.

### Etat 1 : le résultat

Nous avons maintenant toutes les données en main pour réaliser notre état 1 : la ronde. Nous avons choisi de donner un nom qui commence par 's' pour tous les *sensors* et 'a' pour tous les *actuators*. Comme un dessin vaut mieux qu'un grand discours voici le résultat.



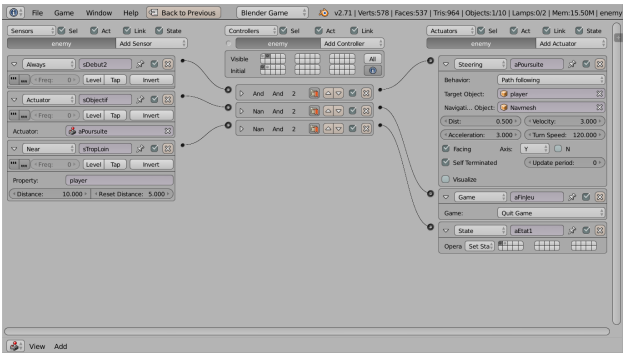
Nous avons mis un *sensor sDebut1* qui est un `Delay` pour démarrer le mouvement après une petite pause pour faire joli. Cette pose sera visible au démarrage mais aussi chaque fois que l'état 1 sera réactivé par la suite.

Ensuite les 3 *actuators* `Steering`, *aPoint1*, *aPoint2* et *aPoint3* pour les 3 points de passages et les 3 *sensors sPoint1, sPoint2 et sPoint3* associés qui forment une boucle, comme expliqué plus haut. Remarquons que les *controllers* connectés aux sensors sont des `Nand` (c'est-à-dire qu'ils inversent le signal logique). En effet, il faut démarrer le segment suivant dans la ronde (donc il faut une impulsion logique positive) quand le segment précédent s'arrête (ce qui produit une impulsion négative dans le *sensor*), d'où le besoin d'inverser le signal logique.

Les *sensors sEnVue et sProche* sont à l'affût du joueur: notons que nous avons écrit *player*dans le paramètre`Property` car le joueur à précisément une propriété du même nom et que c'est le seul objet dans ce cas: seul le joueur sera détecté (il faut aussi l'option `Physics > Actor` pour le joueur sinon ça ne marche pas !). Les 2 *sensors* sont connectés au même *controller* `Or` car il suffit que l'un ou l'autre détecte le joueur pour que l'événement de transition de l'état 1 vers l'état 2 se produise. La transition est effectuée par l'*actuator sEtat2*.

### Etat 2 : le résultat

Voici notre état 2.



Notons le *sensor* `Always` qui active immédiatement le comportement de poursuite. Les options de l'*actuator aPoursuite* sont calculées pour aller plus vite que dans la ronde mais moins vite que le joueur, sinon nous n'aurions aucune chance de gagner. Le *sensor sObjectif* avec son *controller* `Nand` détecte la fin de *aPoursuite* et donc la fin du jeu puisque le joueur a été rejoint. Nous voyons que l'état 3 ne doit pas être réellement implémenté, il suffit d'un *actuator* pour terminer le jeu. Le *sensor sTropLoin* détecte quand le joueur est proche de l'agent, or nous avons besoin du contraire pour générer l'événement qui nous ramènera à l'état 1, d'où l'inversion par un *controller Nand*.

### Conclusion et derniers conseils

Nous sommes arrivés au bout de notre tâche avec un minimum de briques logiques. Cet exemple montre que des comportements assez évolués sont réalisables facilement. Cependant la complexité peut augmenter rapidement et il devient nécessaire de comprendre ce qui se passe pendant le jeu. Nous avons la possibilité d'afficher des valeurs dans la fenêtre de jeu très simplement :

1. Menu principal > `Game`>`Show Debug Properties`.
2. Pour afficher l'état courant d'un objet.


3. Pour afficher la valeur d'une propriété d'un objet.

4. Penser à utiliser l'*actuator*`Property` pour changer la valeur d'une propriété pendant le jeu en vue du débogage.

## SCRYPTHON !

L'algorithme de navigation est accessible par Python. Il est possible de ré-implémenter tout ce qui précède entièrement en Python et nous aurions en prime la possibilité de palier aux limitations de l'*actuator*`Steering` : par exemple obtenir une accélération progressive et des rotations douces.

Imaginons une situation assez classique avec un ennemi poursuivant notre personnage. Celui-ci pour tenter de s'échapper active alors un pouvoir rendant collant le sol derrière lui. Lorsque l'ennemi marcherait sur ce sol collant (on détectera cela avec une collision), sa vitesse se réduirait. Une fois ressorti de la zone collante, il reprendra alors sa vitesse de départ.

Concernant la mise en place, on a décidé de détecter la collision du côté ennemi. Nous avons donc commencé par créer un *sensor* `Collision` qui ne s'active que lorsque notre ennemi entre en collision avec un décor de type sol magique (nous avons réussi cela en dotant notre décor sol magique d'une propriété et en filtrant le *sensor* avec cette propriété).

```
from bge import logic

cont = logic.getCurrentController()
enemy = cont.owner

e_type = enemy['enemy_type']
```

```
sensor = cont.sensors["magic_collision"]
player_steering = enemy.actuators["Steering_player"]

if not sensor.positive:
     player_steering.velocity = enemy.normal_velocity
else:
     player_steering.velocity = enemy.normal_velocity - 1.0
```

Nous commençons, très classiquement, par l'import des modules. Nous récupérons ensuite le type d'ennemi ainsi que le *sensor* de collision qui a, dans notre exemple le nom de "*magic_collision*". Comme nous devons modifier la vitesse de poursuite de notre ennemi, il nous faut la référence de l'*actuator* de poursuite.

Ensuite nous n'avons plus qu'à modifier la vitesse de celui-ci. La valeur **`sensor.positive`** indique si la collision commence ou si elle vient de se terminer. Si elle commence, on réduit la vitesse, si elle vient de se terminer, on rend à l'ennemi sa vitesse normale. Tous les *sensors* possèdent d'ailleurs l'attribut **`positive`** qui indique si l'événement détecté est réalisé ou pas. Ceci est décrit plus en détails dans le chapitre Comprendre les briques logiques de la section **Développer l'ergonomie**.

# DÉVELOPPER L'ERGONOMIE

**26.** GÉRER LES PÉRIPHÉRIQUES
**27.** GESTION DE SCÈNE
**28.** RÉALISER UN MENU
**29.** LOGIQUE DE JEU
**30.** COMPRENDRE LES BRIQUES LOGIQUES
**31.** POINT DE VUE DU JOUEUR
**32.** AFFICHER PLUSIEURS POINT DE VUES

# 26. GÉRER LES PÉRIPHÉRIQUES

Une partie très importante de la création d'un jeu vidéo est l'interface avec le joueur, ou IHM (Interface Homme Machine). Définir comment le joueur va interagir avec notre jeu, c'est penser une ergonomie : si le joueur a l'impression que le jeu ne réagit pas à ses commandes ou au contraire si le jeu réagit trop vite, il risque de ne pas arriver à jouer ou bien de se désintéresser du jeu. Nous devrons probablement passer du temps à tester et à affiner nos réglages pour que le jeu donne un bon "feeling" au joueur, qui est là pour s'amuser !

Expliquer au joueur comment jouer lors du premier niveau, faire des niveaux avec une difficulté progressive, sont également des principes à garder en tête : nous avons conçu notre jeu, donc nous savons comment y jouer... mais le futur joueur lui ne le sait pas, ce n'est pas évident pour lui. La première chose à considérer est donc quel matériel votre joueur va utiliser pour jouer. En fonction des périphériques du joueur, nous ne ferons pas les mêmes types de jeux : concevoir un jeu jouable au clavier et à la souris n'a rien à voir avec un jeu tactile sur un smartphone, ou un jeu qui suppose que le joueur fasse de grands gestes devant un détecteur de mouvements.

## EN PRATIQUE

Dans le BGE, on récupère les informations envoyées par les périphériques via les briques logiques de type *sensor* (*Keyboard*, *Mouse*, ou *Joystick*) ou bien via des scripts Python, grâce aux objets `logic.keyboard`, `logic.mouse` ou `logic.joysticks`, ou encore par le protocole OSC (voir plus loin).

En Python, nous devrons donc importer en début de code les modules *`logic`* et *`events`*. Ce dernier contient les constantes correspondant à toutes les entrées possibles du clavier et de la souris. Pour toutes les touches et boutons, qu'il s'agisse du clavier ou de la souris, il existe 3 statuts différents :

- `logic.KX_INPUT_NONE` : le statut par défaut, bouton au repos, non actif ;
- `logic.KX_INPUT_JUST_ACTIVATED` : si une touche ou un bouton vient juste d'être pressé ;
- `logic.KX_INPUT_ACTIVE` : si une touche ou un bouton est actuellement pressé.

## LE CLAVIER

Le clavier est le périphérique qui contient le plus d'entrées. Si votre jeu nécessite un grand nombre de commandes différentes, comme souvent dans les jeux de stratégie ou les jeux à la première personne avec de multiples actions possibles du héros, vous aurez certainement besoin du clavier.

### Briques logiques

Dans la première section de ce manuel, pour déplacer notre personnage, nous avons vu comment récupérer les frappes clavier avec le *sensor*`Keyboard`. Référez-vous à cette partie si vous avez des doutes : section **Créer son premier jeu**, chapitre **Déplacer le personnage**.

### Scrypthon !

Pour écrire un script Python qui récupère les frappes au clavier, voici comment procéder:

```
from bge import logic, events

clavier = logic.keyboard

if clavier.events[ events.UPARROWKEY ] == logic.KX_INPUT_ACTIVE :
    print("La touche up active")
```

Etudions ce code ligne par ligne.

```
from bge import logic, events
```

L'import des modules Python est l'entête obligé de tout **script** Python dans le BGE. Nous aurons besoin de *`logic`* pour accéder au clavier et *`events`* pour les constantes du clavier.

```
clavier = logic.keyboard
```

Cette technique classique en programmation Python consiste à assigner la référence d'un objet accessible par une expression complexe à variable locale pour plus de clarté et d'efficacité.

```
if clavier.events[ events.UPARROWKEY ] == logic.KX INPUT ACTIVE :
```

L'objet clavier a un attribut *events* qui est un dictionnaire qui répertorie le statut courant de toutes les touches (actif, tout juste activé ou inactif). Le test vérifie si le statut de la touche flèche vers le haut (events.UPARROWKEY) est active (statut logic.KX_INPUT_ACTIVE).

```
    print("La touche up active")
```

Et si c'est le cas, le test imprime ce message en console.

*Notons que pour que ce simple script fonctionne, il faut au minimum l'attacher à un sensor Always dont le pulse mode TRUE level triggering est actif. Autrement dit, il faut que ce script soit exécuté à chaque frame.*

## LA SOURIS

La souris est l'extension la plus importante et la plus courante après le clavier. Qu'elle se pende ou non au bout d'un fil, qu'on l'appelle *trackpad* ou mulot1. Qu'elle possède un, deux ou trois boutons, c'est l'outil de base du joueur pour pointer des éléments sur un écran. C'est précis, sensible et encore bien ancré dans des pratiques de jeu.

### Briques logiques

La brique logique permettant de récupérer les événements liés à la souris est la bien nommée *sensor* ***Mouse***. Elle est assez simple à configurer puisqu'il n'y a qu'une seule option : c'est l'action de la souris qui active ce *sensor*. Ce sont bien sûr les clics de boutons mais aussi le simple fait de déplacer la souris (***Movement***).

L'autre type d'action intéressant est l'événement ***Mouse Over*** qui correspond au passage de la souris au dessus de l'objet 3D auquel ce *sensor* est attaché. Cette option peut se révéler très utile pour créer un menu ou sélectionner des éléments dans une scène.

### Scrypthon !

Voici comment procéder en Python :

```
from bge import logic, events

souris = logic.mouse

if souris.events[events.LEFTMOUSE] == logic.KX_INPUT_JUST_ACTIVATED :
    print("Clic gauche")

# Affiche en console la position de la souris
print( souris.position )

# Afficher le curseur pendant le jeu
souris.visible = True
```

Comme pour le clavier, nous assignons l'objet *logic.mouse* à une variable "souris" pour plus de clarté.

L'objet souris possède également un attribut events qui répertorie le statut de toutes les actions souris. Cette fois, nous détectons si le bouton de gauche (events.LEFTMOUSE) est "tout juste activé" (logic.KX_INPUT_JUST_ACTIVATED). Cette condition sera vérifiée lorsque le joueur aura tout juste appuyé sur le bouton gauche de sa souris. S'il maintient le bouton appuyé, cette condition ne sera plus vérifiée.

## MANETTE

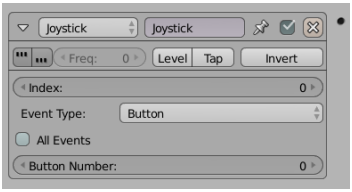
La manette, le joystick, le volant, le *gamepad*, etc., sont autant d'accessoires bien connus des joueurs. Ils ont la particularité de renvoyer davantage d'informations que le simple fait de cliquer sur tel ou tel bouton. Ces accessoires ont des formes multiples, parfois en lien étroit avec le type de jeu auxquels ils sont apparentés (les volants pour les courses de voiture par exemple). En revanche, ils partagent tous la caractéristique d'envoyer des informations précises de direction, parfois variables en intensité et souvent représentées sur des axes x et y.

Dans la suite de ce chapitre, nous utiliserons le terme manette pour désigner la manette de jeu dans son ensemble et joystick pour désigner un manche en particulier. Les joysticks peuvent être à axe simple (par exemple avant-arrière) ou multiple (par exemple avant-arrière, gauche-droit). Contrairement à la souris qui renvoie une position dans un rectangle (la fenêtre de jeu). Ici, il s'agit plutôt de déterminer ici l'angle du joystick par rapport à sa position au repos.

Nous pouvons utiliser ce .blend pour tester si notre manette préféré est supportée par le BGE: [référence vers blends/SCA_PythonJoystick/joystick_test.blend + 8bitoperator.ttf]

### Briques logiques

Pour utiliser une manette, nous nous servirons de la brique logique *sensor* ***Joystick***.




Plusieurs manettes peuvent être connectées à l'ordinateur. Le BGE numérote automatiquement les manettes en partant de 0. C'est ce numéro que nous indiquons dans le champ ***Index***pour spécifier quelle manette nous intéresse. Lorsqu'il n'y a qu'une seule manette connectée, elle aura d'office l'index 0. Mais dans le cas de plusieurs manettes, nous ne pouvons malheureusement pas savoir à l'avance quel seront leurs numéros.

Cette question est importante lorsque nous publions un jeu : quel index choisir dans les *sensors* et comment permettre à l'utilisateur de choisir la manette qui sera utilisée dans le jeu ? La solution de facilité est de fixer l'index à 0 dans tous les *sensors*, ce qui force l'utilisateur à prendre une manette en particulier, ou l'oblige à débrancher ses autres manettes pour utiliser sa manette préférée. La solution complexe consiste à proposer un choix par menu à l'utilisateur et à modifier par Python le champs index des *sensors*, voire de se passer complètement des *sensors*, car tous les événements manette sont récupérables par Python (voir plus bas).

L'option ***event Type***détermine le type d'événement que nous allons récupérer : simple bouton pressé (***Button***), action sur une croix directionnelle (***Hat***), déplacement d'un joystick à axe simple ou multiples (***Axis***). Malheureusement le même problème se pose pour les boutons et les joysticks des manettes: ils sont numérotés automatiquement et l'ordre peut varier fortement d'un fabriquant de manette à l'autre.

### Scrypthon !

Voici un extrait de script permettant de récupérer facilement les indices des manettes connectées.

```
from bge import logic

# Affichons les manettes connectées et leur indice
indice = 0
for manette in logic.joystick :
    if manette != None :
        print( manette.name, " a l'indice ", indice )
    indice += 1
```

L'objet `logic.joystick` contient la liste de manettes connectées à l'ordinateur dans l'ordre où elles ont été numérotées par le BGE. La boucle `for` passe en revue les éléments de cette liste. Si un élément n'est pas vide (`None`), il représente une manette actuellement connectée. Nous pouvons accéder à ses propriétés, comme par exemple son nom (ici par l'affichage en console de `manette.name`). La variable `indice` sert à connaître l'index de chaque manette car elle est incrémentée à partir de 0. En affichant le nom de la manette et son index, nous savons maintenant quelle valeur nous devons mettre dans le champ ***Index*** des *sensors* ***Joystick***.

Voici les propriétés d'une manette accessible par Python :

- `name` : le nom de la manette.
- `activeButtons` : une liste d'entiers représentant les boutons actuellement pressés.
- `axisValues` : une liste contenant les valeurs des axes du joystick.
- `hatValue` : statut de la croix directionnelle (0:None, 1:Up, 2:Right, 4:Down, 8:Left, 3:Up-Right, 6: Down-Right, 12: Down-Left, 9:Up-left).

En modifiant notre code précédent; nous affichons les données envoyées par les joysticks lorsqu'on appuie sur leurs boutons :

```
from bge import logic

for manette in logic.joysticks :
    if manette != None :
        # Affichons les boutons actifs
        for bouton in manette.activeButtons :
            print( bouton )
```

En plus de la boucle sur les joysticks, nous bouclons sur la liste `active Buttons`de chaque joystick et nous imprimons son contenu. Les éléments de cette liste sont de simples nombres entiers qui représentent le numéro des boutons actuellement pressés. Ce sont ces numéros que nous utiliserons dans le champ ***`Button Number`*** des *sensors* pour détecter tel ou tel bouton. La liste est vide si aucun bouton n'est pressé.

Pour les autres propriétés, la démarche sera similaire.

- `AxisValue` : retourne une liste de nombres décimaux compris entre -1 et +1 qui représentent l'inclinaison d'un axe de joystick (0 est la position de repos). A noter qu'un joystick double axe prend deux positions successives dans cette liste.
- `HatValues` : retourne une liste de nombres entiers qui représentent la valeur de chaque croix directionnelle.

## PÉRIPHÉRIQUES EXOTIQUES

Il est tout à fait possible d'utiliser d'autres périphériques avec Blender que ceux cités plus haut. Seulement, il n'existe pas de brique logique pour cela. C'est grâce à l'utilisation de scripts Python et du protocole OSC que nous pourrons étendre à l'infini ces possibilités d'interface. Par exemple, la télécommande Wiimote, le senseur Kinect, des capteurs sur Arduino, des contrôleurs provenant d'autres logiciels comme Pure-Data, un clavier MIDI, etc. peuvent tous servir comme interface de contrôle.

Pour la Kinect360, l'usage de OSCeleton permet de facilement récupérer les données des squelettes des utilisateurs détectés par le senseur. Ces données pourront ensuite être traitées dans le BGE. Il existe plusieurs méthodes pour utiliser le protocole OSC dans le Game Engine :

- soit en important le module pyOSC.
- soit en faisant appel à la librairie pylibloc, qui elle-même dépendant de liblo (pour GNU-Linux ou OSX seulement).

Étant donné la diversité des possibilités à ce niveau, nous nous contenterons d'un exemple simple avec pyOSC. Pour les besoins de l'exemple, nous utiliserons 2 instances du Blenderplayer plutôt qu'un contrôleur physique externe, de sorte que n'importe qui puisse comprendre le fonctionnement, sans dépendre d'un matériel spécifique. Le principe est de contrôler le mouvement d'un cube dans un *player* à partir d'un autre *player*.

<OSC_send.blend>

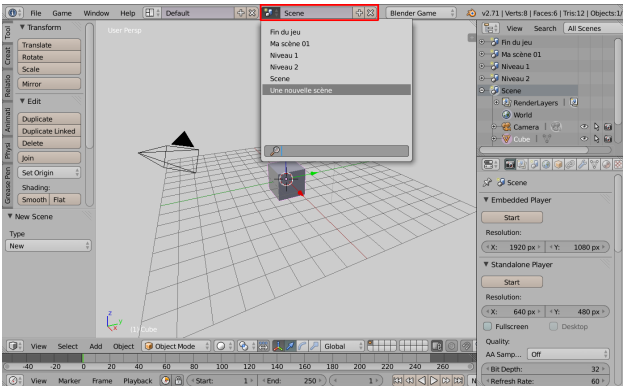
Dans ce premier fichier, on se contente de récupérer les événements du clavier, et d'assigner des commandes à certaines touches. Ces commandes sont envoyées par le script en OSC sur le réseau. Ici en envoyant les infos vers l'ordinateur lui-même, en utilisant l'adresse interne ou *localhost* (127.0.0.1). On peut également spécifier une autre adresse réseau pour envoyer les messages OSC vers une machine distante.

<OSC_receive.blend>

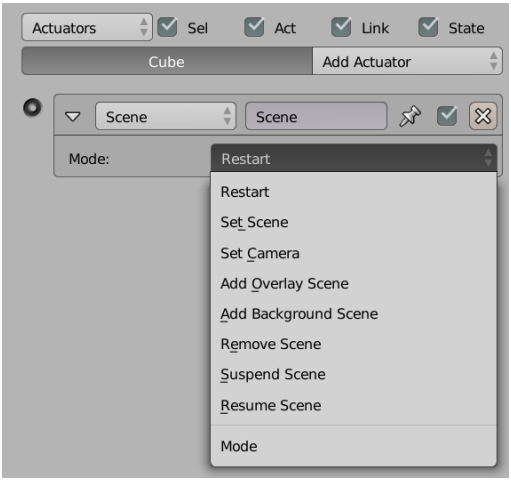
Dans ce fichier, le script Python appelle le module pyOSC et écoute les messages qu'il reçoit pour ensuite utiliser ces données pour ajuster la position du cube en xyz. Une subtilité importante est la fonction `quit()` qui éteint le serveur à la sortie du jeux (sinon, à la prochaine relance, il risque d'y avoir une erreur de port réseau resté ouvert). La touche clavier `Echap` est ici également reliée explicitement à une action *`Game > Quit`*, de sorte à quitter le *Game Engine* (sinon, la touche est réassignée à l'action du script et le jeu ne quitte plus...).

# 27. GESTION DE SCÈNE

Blender permet de travailler sur plusieurs scènes qui peuvent se succéder ou se superposer. Dans le cadre d'un jeu, cette méthode est idéale pour séparer plusieurs niveaux ou chapitres d'une histoire, ou créer des menus et informations à l'écran. Créer, supprimer ou sélectionner des scènes se fait dans la fenêtre `Info`, par défaut tout en haut de l'écran.




La scène qui est active dans Blender sera également la première scène du jeu. Pour gérer d'autres scènes durant le jeu, il faut passer par l'*Actuator* `Scene`. Il possède différents modes que nous allons expliquer.




## CHANGER DE SCÈNE

Pour changer de niveau durant le jeu, l'*actuator* doit être en mode `Set Scene`. L'option `Scene`détermine vers quelle scène basculer (ou plutôt, quelle sera la nouvelle scène active). Il est nécessaire de placer une caméra dans la scène cible pour obtenir une vue de la scène une fois le changement effectué. La scène qui a provoqué le changement est détruite ainsi que tous les objets qu'elle contient. Ce mode est idéal pour charger un niveau donné avec tous les objets dans leur état initial.

## LANCER PLUSIEURS SCÈNES

Il est parfois nécessaire d'avoir plusieurs scènes fonctionnant simultanément : par exemple, afin d'afficher une interface graphique par dessus l'écran (cette technique sera décrite en détail au chapitre suivant). Les modes ***Add Overlay Scene*** et ***Add Background Scene*** permettent respectivement d'ajouter une scène à l'avant ou à l'arrière de la scène active. Comme une scène en *Background* est affichée avant la scène active, elle détermine le fond de l'écran par ses objets et par ses paramètres de l'onglet *World*. Une scènes en *Overlay* est affichée après la scène active et ses objets sont visibles par dessus tous les objets de la scène. Seul le fond de la scène en *Overlay* sera transparent et laissera apparaître les éléments de la scène active.

Le mode *Overlay* est utile pour afficher des éléments en surimpression de la scène principale, comme un menu (voir chapitre suivant), des informations comme les points de vie ou le score (voir le chapitre **Point de vue du joueur** de cette même section).

## AGIR SUR UNE SCÈNE DÉJÀ LANCÉE

Le mode ***Suspend Scene*** sert à mettre en pause la scène visée. Attention, une scène en pause ne peut se sortir elle-même de pause car sa logique interne est à l'arrêt ! La mise en pause et la sortie de pause d'une scène doivent provenir de commandes présentes dans une autre scène, par exemple, le bouton pause d'un menu présent sur une scène en *Overlay*.

Le mode ***Remove Scene*** permet de supprimer une scène et tout ce qu'elle contient. On utilise habituellement cette commande pour enlever les scènes placées en *Overlay* ou en *Background* d'une scène active. Si la scène active se retire elle-même, cela provoque la fin du jeu.

Le mode ***Restart*** redémarre la scène courante comme si le jeu venait de démarrer : elle retrouve tous ses paramètres initiaux.

## SCRIPTHON !

Pour ajouter une scène nous utiliserons la fonction **addScene()** de **bge.logic**. Cette fonction prend deux arguments, le premier obligatoire est le nom de la scène que l'on veut ajouter, le deuxième indique si la scène doit être ajoutée en mode *Overlay* (argument à 1) ou en *Background* (argument à 0).

Récupérons la scène courante et remplaçons-la par une autre :

```
from bge import logic

# Recupere un objet
KX_Scene scene = logic.getCurrentScene()
scene.replace("Nom de la nouvelle scene")
```

Récupérons une autre scène et et mettons-la en pause :

```
from bge import logic

# Liste les scènes uniquement en cours d'execution
scene_list = logic.getSceneList()
# Parcours de la liste
for scene in scene_list:
     if scene.name == "Game":
          scene.suspend()
```

Réactiver une scène en pause :

```
if scene.suspended:
     scene.resume()
```

Fermer une scène :

```
scene.end()
```

Voir :
http://www.blender.org/documentation/blender_python_api_2_71_release/bge.types.KX_Scene.html#bge.types.KX_Scene

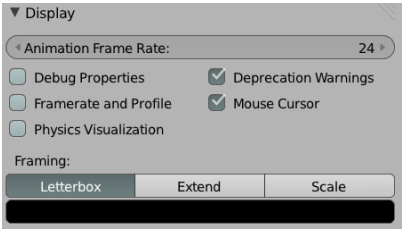
# 28. RÉALISER UN MENU

Démarrer une nouvelle partie, quitter le jeu, consulter les crédits : ce type d'actions passe souvent par des boutons regroupés dans un menu.

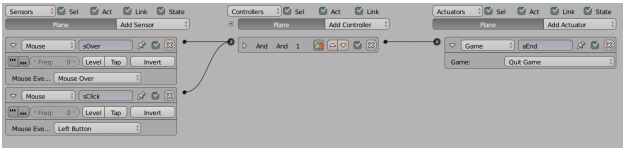
Nous découvrirons dans ce chapitre quelques techniques pour créer un menu grâce aux briques logiques. Notez que nous détaillerons le fonctionnement des options communes à tous les *sensors*(***Tap***, ***Level***, ***Invert***, ***Pulse***) dans le chapitre suivant.

## UN BOUTON FONCTIONNEL

Les menus classiques fonctionnent généralement avec la souris. Nous devons détecter si une souris survole un objet et détecter un événement souris. La première nécessité est d'afficher le *cursor*: ***Properties*** > ***Render*** > ***Display*** > ***Mouse Cursor***.

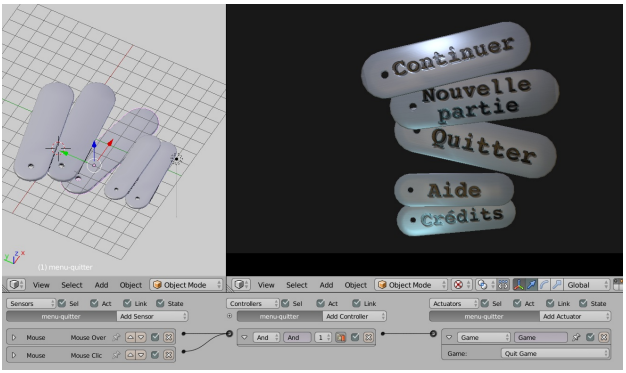



Pour notre premier bouton, nous ajoutons un plan, sur lequel nous mettrons une texture de bouton avec du texte indiquant son rôle, dans cet exemple ce sera quitter le jeu. Nous ajouterons un *actuator*de type ***Game***, le but étant de l'activer avec la souris.



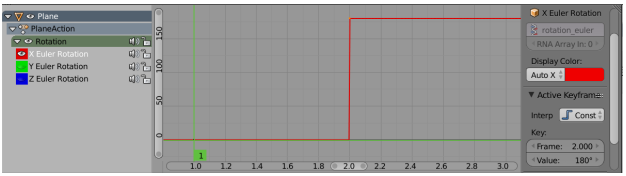

Le premier *sensor* `Mouse` configuré en`Mouse Over` envoie *True* lorsque l'objet (ici le bouton) est survolé par la souris. Le deuxième *sensor*`Mouse`, configuré en `Left Button`, envoie *True*lorsqu'il détecte l'événement clic gauche de la souris. Le contrôleur logique lié aux deux `sensors Mouse` est ici de type `And` : parce qu'il faut que tous ses *sensors* soient *True* pour que les *actuators* auxquels il est connecté soient activés. Ces deux conditions sont en effet nécessaires pour que cela corresponde à un clic du bouton. L'*actuator* termine le jeu.

Nous pouvons ensuite dupliquer ce plan (changer sa texture) et modifier l'*actuator* pour choisir une autre action, par exemple changer la scène courante vers une scène appelée Options.




## EMBELLIR LE MENU

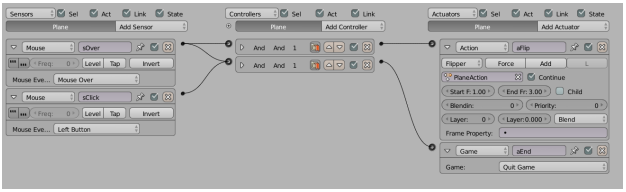
Nous souhaitons améliorer l'aspect visuel du menu et le rendre plus interactif, avec par exemple des boutons qui changent d'aspect lorsqu'on les survole ou qui jouent des animations (ils sautillent, tournent, grossissent, etc.). La façon la plus économe et de faire un bouton qui change de couleur est de placer deux plans dos à dos, et faire tourner l'ensemble de 180° lorsqu'il est survolé par la souris. Pour obtenir une rotation précise et reproductible, nous utiliserons une animation.



Notons que nous avons choisi l'interpolation constante de sorte que le *flip* soit instantané sans étape intermédiaire.

*Le BGE tourne avec un fps de 60, soit bien plus que le fps habituel de 24 ou 25 des animations en cinéma ou vidéo. Pour que les animations se jouent néanmoins à la vitesse normale, le BGE affiche des frames intermédiaires. Avec une interpolation de type Bézier, nous verrions le bouton brièvement tourner, ce qui n'est pas l'effet recherché.*

Ensuite avec une brique **`Action`** en mode **`Flipper`**, nous connectons simplement cet *actuator* au *sensor* `Mouse Over`.



Vous pouvez télécharger cette demo ici: [menu_bouton_flip.blend]

L'explication courte est la suivante : le mode`Flipper` joue l'animation dans le sens normal lors de l'activation par le *controller* et dans le sens inverse lors de la désactivation : le bouton retrouve sa rotation normale lorsque la souris quitte le bouton. Dans le chapitre suivant nous décrirons comment faire la même chose avec Python après avoir expliqué la mécanique interne du moteur logique.

## FAIRE APPARAÎTRE UN MENU PENDANT LE JEU

Comme nous l'avons vu dans le chapitre précédent avec la gestion de plusieurs scènes, il suffit d'ajouter une scène en *Overlay* (la scène menu) et de mettre en pause la scène courante. Ensuite, lorsqu'on supprime le menu, il ne faut pas oublier de faire un *resume scene* pour relancer la scène courante mise en pause précédemment.

# 29. LOGIQUE DE JEU

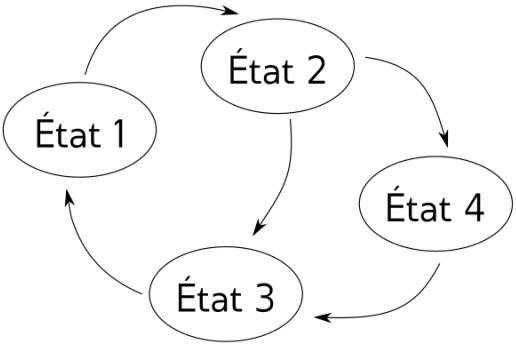
Lors de la création d'un jeu, la mise en place d'un système d'interaction complexe entre des éléments du jeu et le joueur arrive rapidement. Par exemple, on cherche à mettre en place un ennemi montant la garde, et tentant d'attraper le joueur ou de lui tirer dessus à son approche. Bien entendu, le garde peut adopter d'autres états et adopter d'autres réactions. Imaginons un jeu de cuisine avec des plats à mettre au four. Cela demande la gestion d'état de la cuisson du plat, comprenant des états crus, légèrement cuits, bien cuit, trop cuits, totalement brûlés. Nous pouvons tout aussi bien imaginer un changement en cascade, comme avoir dans un jeu, un interrupteur qui s'actionne par le joueur et ouvre une porte au loin. Ce chapitre va être l'occasion d'étudier deux mécanismes permettant de mettre en place ce principe en utilisant les machines à états et l'envoi de message.

## LES MACHINES À ÉTATS

Les objets avec lesquels le joueur doit interagir auront assez régulièrement besoin de stocker leur état. Les **machines à états** formalisent cela.

### Principes généraux d'une machine à états

Une machine à états est composée de deux choses, les états et les transitions permettant de passer d'un états à un autre. Représentons une machine à états sous la forme d'un graphe : les états sont les nœuds du graphe et les événements sont les flèches qui relient les nœuds.




Pour que la machine à états soit cohérente, il faut bien sûr éviter les états orphelins (aucune flèche n'y mène) et les états cul-de-sac (aucun chemin n'en sort), sauf s'il s'agit d'un état terminal qui termine le jeu ou détruit l'objet. En principe, un seul état est actif dans une machine à état. Le système d'états du BGE permet d'avoir plusieurs états actifs en même temps mais nous n'aborderons pas cette possibilité car cela se résume souvent à considérer plusieurs machines à états indépendantes.

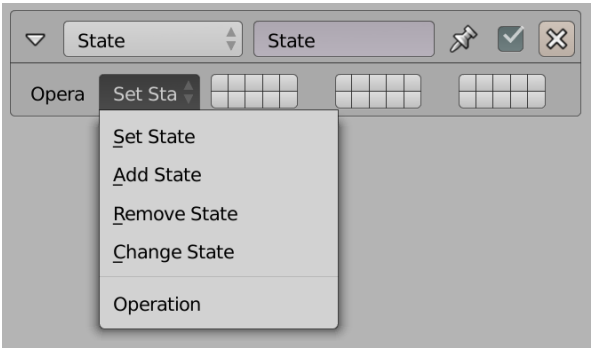
### Machine à états dans le BGE

Pour ce comportement complexe, nous aurons besoin d'utiliser le système des états du BGE. Cette technique avancée que nous n'avons pas abordé jusqu'à présent consiste à regrouper les briques par état. Ensuite, en activant tel ou tel état, nous obtenons différents groupes de briques actives, et donc différents comportements. Dans le BGE, les états sont portés par les *controllers* : nous choisissons l'état auquel appartient un *controller* grâce à ce bouton.


Les *sensors* et *actuators* qui sont connectés à un *controller* font automatiquement partie de l'état auquel appartient le *controller*. Le changement d'état se fait grâce à l'*actuator* State.


Chacun des 30 petits boutons correspond à un état. L'*actuator* permet de manipuler plusieurs états simultanément. Les états peuvent être actifs ou inactifs et l'*actuator* permet de changer cela. Le type de manipulation est déterminé par l'option Operation. Nous nous limiterons à l'opération la plus simple qui consiste à activer un état et désactiver tous les autres. Par exemple pous activer l'état 2, nous utiliserons l'actuator en mode Set State.


*Pour info, les autres opérations sont : inversion de certains états (Change State), activation de certains états sans changer ceux qui étaient déja actifs (Add State), désactivation de certains états (Remove State). Ces opérations sont utiles dans le cas ou plusieurs machines à états coexistent pour un objet.*



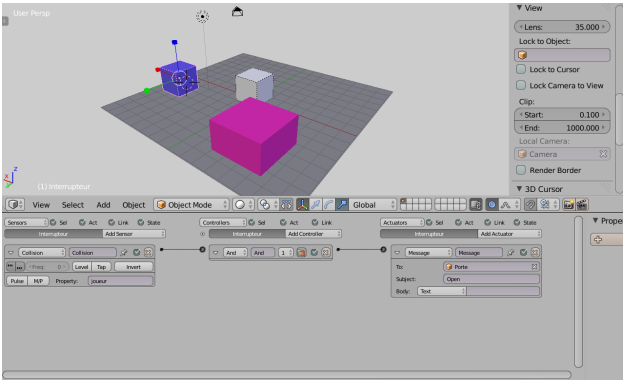

Pour rappel, la brique qui permet de changer l'état étant un actuator, il faut nécessairement un controller et un sensor pour l'activer. Dès qu'un état est activé, toutes les briques de cet état deviennent fonctionnelles comme si le jeu venait de démarrer. En particulier les sensors *Always* et *Delay* produisent une impulsion logique, ce qui permet de démarrer la logique de l'état.

Les briques appartenant à un état inactif n'utilisent aucune ressource du BGE. Une machine à états bien pensée est donc également un outil d'optimisation.

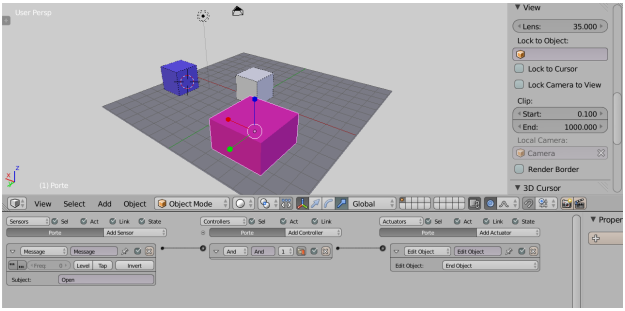
## COMMUNICATION ENTRE OBJETS, UTILISATION DES MESSAGES

Le *Blender Game Engine* permet de faire communiquer les objets entre eux grâce à des messages. Il faut voir un message comme un *email* que l'on pourrait envoyer. Un message se décompose en plusieurs éléments. Tout d'abord un message est émis par un objet précis, son émetteur. Ensuite un message contient un sujet et un corps de message. Enfin un message peut être envoyé à quelqu'un ou bien envoyé à tout le monde.

Pour illustrer l'utilisation de messages nous allons mettre en place une mécanisme interrupteur - porte. Nous allons donc avoir un objet représentant le joueur (le cube gris), un objet représentant l'interrupteur (le cube bleu) et un pavé représentant la porte (le mesh rose).



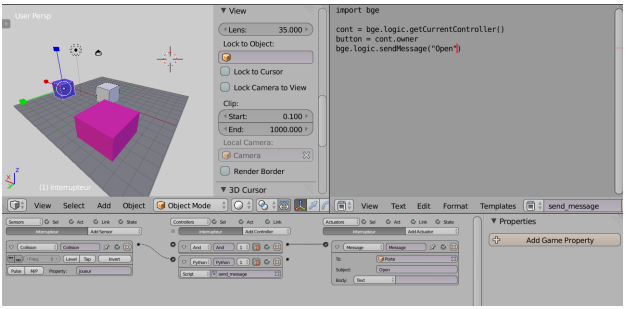
Expliquons maintenant les Briques que nous utilisons. En *Sensor*, nous utilisons tout d'abord un détecteur de collision branché sur l'interrupteur. Nous ajoutons ici un filtre sur les propriétés. En effet, l'interrupteur est en collision perpétuelle avec le sol et nous ne voulons ouvrir la porte que si le joueur touche à l'interrupteur. Bien entendu, nous avons pris soin d'ajouter une propriété de type *string* et ayant pour valeur `joueur` à notre cube gris représentant le joueur. Le contrôleur est très classiquement un contrôleur de type `And`. C'est dans l'*actuator* que nous mettons en place l'envoi du message. Il y a pour cela un *Actuator* spécial, l'actuator de type message. On définit le récepteur du message avec la partie *To* (Nous aurions très bien pu ne pas renseigner cette partie, le message aurait alors été envoyé à tous les objets). Définissons ensuite un sujet avec la partie *Subject* et le corps du message avec la partie *Body*. Ici nous laissons vide la partie *body* dont nous n'avons pas l'utilité. Maintenant que notre interrupteur sait envoyer un message à notre porte, il faut que notre porte soit capable de recevoir le message et de le comprendre afin de s'ouvrir. Voici la suite de briques logiques qui permet de faire en sorte que la porte reçoive les messages qu'on lui envoie.



Pour la porte, les choses sont encore plus simples que pour l'interrupteur. Il faut que la porte détecte un message. Nous allons donc utiliser un nouveau sensor spécifiquement dédié à cela, le *sensor Message*. On peut filtrer sur un seul critère, le sujet du message. Ici nous voulons réagir sur la réception d'un message d'ouverture de porte, donc nous filtrons sur le sujet Open.

### Scrypthon !

Intéressons-nous à l'envoi d'un message en Python. Comme vous pouvez le voir sur la capture d'écran suivante, la connexion entre le *sensor* de collision et le *controller And* précédent a été supprimé. Un contrôleur Python a ensuite été rajouté puis relié au *sensor* de collision. Le script Python s'occupera d'envoyer le message.




Comme vous pouvez le constater dans la capture, le code est vraiment très court. Le revoici ici pour plus de lisibilité.

```
import bge

cont = bge.logic.getCurrentController()
button = cont.owner
bge.logic.sendMessage("Open")
```

La première ligne est toujours celle de l'import du BGE. Nous récupérons ensuite l'objet relié au contrôleur (dans cet exemple ce n'est pas obligatoire, mais c'est souvent utile). Pour l'instant, la ligne suivante est celle qui envoie réellement le message. La fonction `sendMessage()` fournie par le BGE est utilisée. Ce code se contente d'envoyer un message avec uniquement un sujet : `Open`. La fonction `sendMessage()` nous permet toutefois de faire bien mieux que cela. Les autres arguments de cette fonction sont body, to et message_from. Pour dire explicitement que ce bouton envoie un message, le code aurait été :

```
bge.logic.sendMessage("Open", message_from=button)
```

D'où l'intérêt d'avoir récupéré le button précédemment.

Nous avons vu un exemple relativement simple des messages dans le *Blender Game Engine*. Cependant, il sont surtout intéressant dans des usages plus poussés, comme envoyer des informations variant en fonction de l'émetteur et de son état dans les corps de message.

# 30. COMPRENDRE LES BRIQUES LOGIQUES

Dans ce chapitre nous étudierons en détail le fonctionnement du moteur logique du BGE. Jusqu'à présent, nous avons utilisé les briques sans trop nous soucier de la mécanique interne mais il nous faut en savoir plus si nous voulons faire des machines à états élaborées et efficaces. Commençons par décrire les caractéristiques communes à toutes les briques.

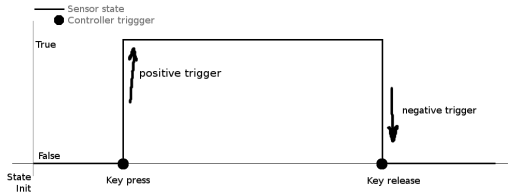
### Principes de base

Les *sensors* ont tous un état interne qui est un simple booléen : Vrai (*True*) si la condition qu'ils détectent est réalisée ou Faux (*False*) dans le cas contraire. Les *sensors* produisent en outre des impulsions (*Trigger*) qui activent les *controllers* auxquels ils sont connectés.

*Par la suite nous parlerons d'**impulsion positive** si l'impulsion est générée alors que le sensor est dans l'état **Vrai** et d'impulsion négative dans le cas contraire. Cependant il faut bien comprendre que les controllers sont activés de la même manière par une impulsion, qu'elle soit positive ou négative.*

Un *sensor* produit toujours une impulsion lorsque son état change. D'autres impulsions peuvent être générées en fonction des options communes à tous les *sensors*, nous y reviendrons.

Prenons comme exemple le *sensor* **Keyboard** car il est facile à utiliser et représentatif de tous les *sensors*. Voici le comportement de base du *sensor* :




Cette image représente l'état du *sensor* au cours du temps depuis le démarrage du jeu (ou l'activation de l'état auquel appartient le sensor). Les boules noires représentent les moments ou une impulsion est générée. Nous voyons qu'un impulsion (positive) est générée lorsque la touche est pressée et une autre (négative) lorsqu'elle est relâchée. Un *controller* connecté à ce *sensor* sera donc activé deux fois.

Un *controller* activé sous l'effet d'une impulsion examine l'état de tous les *sensors* auxquels il est connecté (qu'ils aient ou non généré une impulsion) et selon les conditions logiques qui lui sont propres peut envoyer aux *actuators* auxquels il est connecté une impulsion positive ou négative.

Les *controllers* logiques envoient une impulsion positive à tous les *actuators* si la condition logique est vérifiée, sinon ils envoient une impulsion négative. Cette liste donne les conditions logiques pour chaque type de *controller* :

- **And** : tous les *sensors* doivent être positifs.
- **Nand** : au moins un *sensor* doit être négatif.
- **Or** : au moins un *sensor* doit être positif.
- **Nor** : tous les *sensors* doivent être négatifs.
- **Xor** : un seul *sensor* doit être positif.
- **Xnor** : zéro, deux ou plus de *sensors* doivent être positifs.
- **Expression** : l'expression logique est écrite en python.

*Le controller Python n'est pas un controller logique : le script doit explicitement indiquer quels actuators reçoivent une impulsion et s'il s'agit d'une impulsion positive ou négative. Le script peut même se passer entièrement d'actuator en agissant directement sur les objets via l'API* Python.*

Une fois transmise à un *actuator*, une impulsion positive l'active et une impulsion négative le désactive (ou ne fait rien s'il était déjà inactif). Le temps pendant lequel un *actuator* reste actif en l'absence d'impulsion varie d'un type à l'autre. Nous distinguons 3 type d'*actuators*.

- Ceux qui restent actifs tant qu'ils ne reçoivent pas d'impulsion négative. Les *actuators* suivants appartiennent à cette catégorie :
  - ***Armature*** (en mode ***Run***)
  - ***Camera***
  - ***Constraint***
  - ***Motion***
- Ceux qui restent actifs un certain temps et se désactivent automatiquement sous certaines conditions :
  - ***Action***
  - ***Steering***
  - ***Sound***
- Ceux qui ne restent pas actifs : ils exécutent une opération et se désactivent immédiatement.
  - ***Edit Object***
  - ***Filtre 2D*** (le filtre, une fois en place, reste actif)
  - ***Game***
  - ***Message***
  - ***Mouse*** (dans le cas de *Mouse Look*, il faut envoyer constamment des impulsions pour actualiser l'orientation de la caméra)
  - ***Parent***
  - ***Property***
  - ***Random***
  - ***Scene***
  - ***State***
  - ***Visibility***

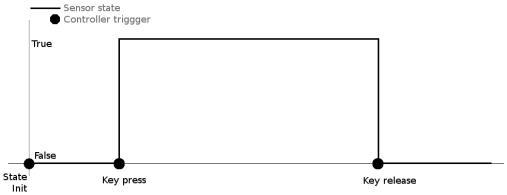
### Mode avancé des *sensors*

Nous pouvons voir que tous les *sensors* ont une ligne de boutons communs.


Ces boutons influencent le nombre et la polarité des impulsions. Voici ce qui se passe lorsque le bouton Tap est sélectionné :


Nous voyons que l'impulsion négative est générée immédiatement après l'impulsion positive, indépendamment du fait que la touche soit toujours enfoncée. L'écart de temps entre les deux impulsions est toujours de 1 *frame*. Ce mode est souvent utilisé pour obtenir des déplacements ponctuels : un *actuator*`Motion`qui reçoit ces deux impulsions s'activera pendant exactement 1 *frame*.

Le bouton ***Level*** fonctionne comme le mode normal excepté qu'une impulsion est toujours générée au démarrage du jeu ou lorsque qu'un *controller* devient actif suite à un changement d'état. Dans ce cas, seuls les *controllers* nouvellement actifs reçoivent l'impulsion.



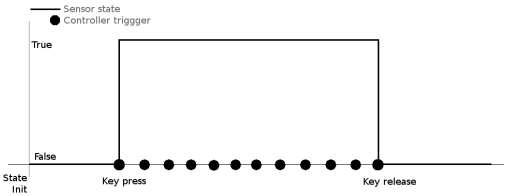

Ce mode est utile pour la programmation de machine à états car il permet une initialisation correcte de l'état (les *controllers* sont garantis de recevoir une impulsion au démarrage de l'état). Le bouton ***Invert*** inverse la logique du *sensor*. Par exemple, dans le cas d u ***Keyboard***, le *sensor* sera positif tant que la touche n'est pas enfoncée et négatif sinon :


*Une impulsion est générée au début du jeu (ou de l'état) car le sensor passe directement positif (nous supposons que la touche n'est pas pressée initialement), ce qui constitue une transition.*

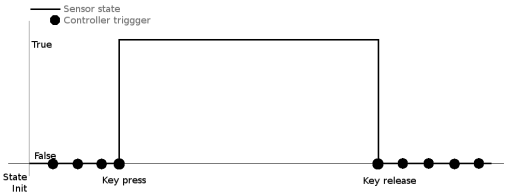
Les boutons***Pulse*** +et ***Pulse*** -produisent des impulsions supplémentaires tant que le *sensor* est respectivement positif ou négatif. Le paramètre ***Freq*** est l'intervalle, exprimé en *frame*, inséré entre deux impulsions, 0 signifiant qu'une impulsion est envoyée à chaque *frame*.

*Il ne faut pas abuser du mode pulse car il augmente rapidement la charge de travail du BGE.*

Voici ce qui se passe quand le bouton pulse + est sélectionné avec *Freq* = 1 :




Et avec pulse - :



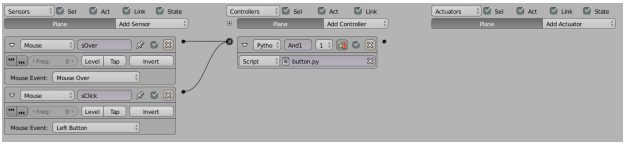

Les boutons peuvent être combinés à l'exception de ***Tap*** qui est incompatible avec ***Level***et ***Pulse***. Voici par exemple ce qui se passe si*Tap*et*Invert* sont sélectionnés en même temps :


## SCRYPTHON !

Pour reproduire le bouton du chapitre précédent nous utiliserons deux attributs communs à tous les *sensors*:

- ***triggered*** : *True* si le *sensor* vient d'envoyer une impulsion
- ***positive*** : *True* si le *sensor* est positif, *False* s'il est négatif

Nous conservons les deux *sensors* car il est plus efficace de demander au BGE de surveiller les boutons de la souris que de le faire en Python. Nous remplaçons les deux *controllers* par un seul *controller* Python et nous nous passerons d'*actuator* pour montrer que nous pouvons agir sur le jeu directement par Python :




Nous utilisons le script suivant :

```
from bge import logic
import math

# Recupere le controller, l'objet et les sensors
cont = logic.getCurrentController()
obj = cont.owner
sOver = cont.sensors["sOver"]
sClick = cont.sensors["sClick"]

# Triggered est True si la souris entre ou sort du bouton
if sOver.triggered:
    # Calcule la rotation selon l'axe X
    rot = math.pi if sOver.positive else 0
    # Ecrire un vecteur d'Euler change la rotation de l'objet
    obj.orientation = [rot, 0, 0]

# Logique equivalente au controller And
if sOver.positive and sClick.positive:
    logic.endGame()
```

Expliquons la partie qui concerne la rotation du bouton. Nous utilisons l'attribut `triggered` du ***Mouse Over*** pour détecter le moment où la souris entre ou sort du bouton : ce *sensor* ne génère une impulsion qu'à ces moments-là. Nous modifions directement la rotation de l'objet en écrivant dans l'attribut `orientation` qui accepte différents types de données. Ici, nous passons un vecteur dont les composantes sont les angles d'Euler. Comme les angles sont en radian, nous utilisons `math.pi`(3,1415) pour indiquer une rotation de 180 degrés.

# 31. POINT DE VUE DU JOUEUR

La vue dans un jeu est défini par des caméras virtuelles. Ces caméras peuvent être contrôlées de bien des façons pour donner des effets scénaristiques. La gestion de la (ou des) caméra(s) est tellement importante qu'on catégorise même les jeux par ce critère. Nous allons voir ici comment définir le point de vue dans différents contextes.

## POINT DE VUE À LA 1ÈRE PERSONNE OU CAMÉRA SUBJECTIVE

Popularisé par des jeux comme Doom (ou Open Arena), souvent utilisé en visualisation architecturale, ce mode de jeu est très immersif puisqu'il place le joueur selon le même point de vue que le personnage qu'il incarne. L'immersion est habituellement renforcée en ajoutant à l'écran des éléments comme les bras du personnage.

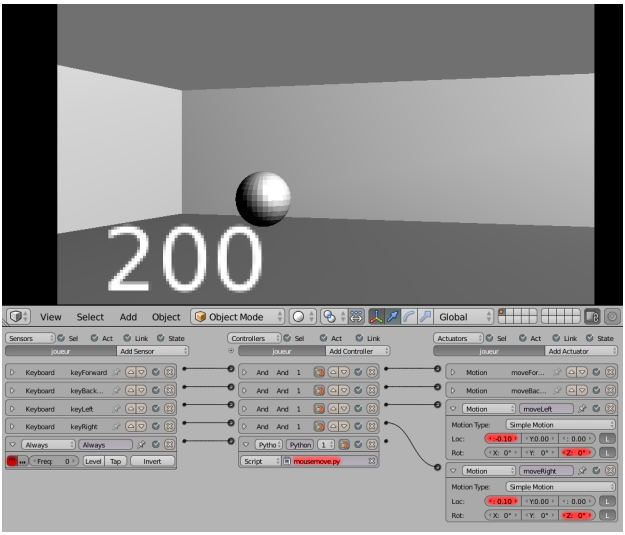
### Mise en place

Reprenons notre exemple du chapitre Créer son premier jeu, et positionnons la caméra exactement à l'emplacement des yeux du personnage. L'objet qui représente notre personnage gêne évidemment la vue, nous allons donc le rendre invisible (panneau ***Physics***, cocher ***Invisible***). Il nous reste à "parenter" la caméra au personnage pour qu'elle colle à ses déplacements : sélectionnons la caméra, puis le personnage, `Ctrl + p`, ***Set Parent to Object***.

### Contrôler le point de vue avec la souris

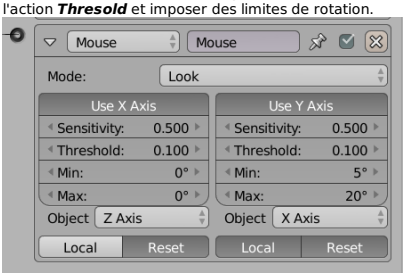
Dans ce type de jeu, la direction dans laquelle regarde le personnage est généralement contrôlée à la souris. Un script bien pratique est dédié à résoudre cette problématique, il s'agit de mousemove.py, partagé sous licence Creative Commons (CC-BY-3.0) par riyuzakisan. Nous allons voir comment le mettre en place en quelques instants.

1. Téléchargeons le script à cette adresse :
   http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Game_Engine/FPS_Mouselook
2. Copions le contenu du fichier .py dans l'éditeur de texte de Blender, et nommez le bloc de texte "mousemove.py"
3. Dans le panneau `Game Logic` de notre objet "joueur", ajoutons deux briques logiques :
   un *sensor* `Always`, en *Pulse Mode (True)*
   un *controller* Python, lié au script mousemove.py
4. Adaptons nos déplacements latéraux pour qu'ils conviennent mieux à ce type de contrôles : Réinitialisons les valeurs de nos *actuators* "*moveLeft*" et "*moveRigth*". Remplaçons les par un mouvement de type `Simple Motion` sur l'axe X, respectivement de -0.1 et 0.1.




La méthode de déplacement est maintenant comparable à celle d'un jeu à la première personne.

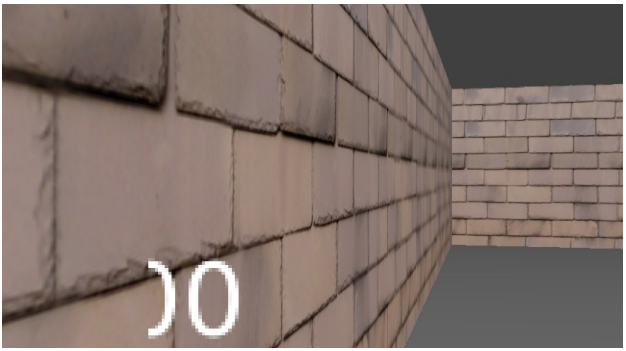
Depuis la 2.72, un *actuator Mouse* a fait son apparition.
En mode ***Look***, connectée à un *sensor Always,* elle a le même effet que le script.
On peut modifier sa réactivité aux mouvements de la souris en modifiant sa senbilité ***Sensitivity*** et le seuil de déclenchement de l'action ***Thresold*** et imposer des limites de rotation.



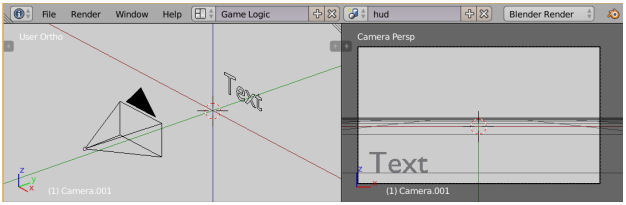

### Éléments proches de la caméra

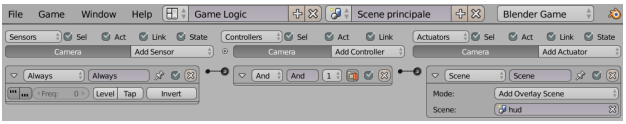
De la même manière, plaçons l'objet qui affiche le score quelques centimètres devant la caméra, et "parentons"-le à la caméra.

Passons en vue caméra (Num0), lançons le jeu et vérifions que tout se passe bien. C'est normalement le cas, à une exception près : lorsque l'on s'approche d'un mur, le score disparaît ! En réalité, il traverse le mur.



Une parade efficace est de créer une nouvelle scène contenant simplement une caméra avec notre objet score, puis de l'ajouter en mode *Overlay* à notre scène principale, comme décrit dans le chapitre précédent sur la gestion de scènes.



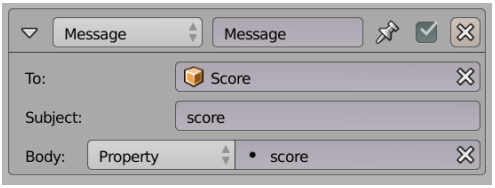




### Communication entre scènes

Il reste à régler la communication entre les deux scènes car elles n'ont rien en commun à l'exception de deux choses :

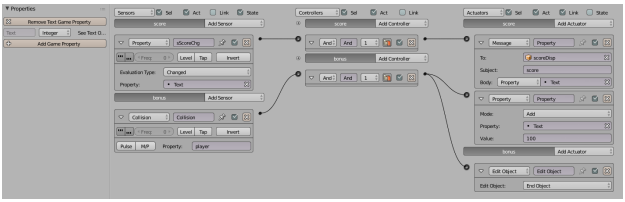
- le système de messagerie interne : un message envoyé d'une scène peut être reçu dans une autre, il suffit que le nom de l'objet destinataire corresponde.
- le répertoire global du moteur logique (`logic.globalDict`). Les objets Python qui sont stockés dans ce dictionnaire persistent tout au long du jeu et sont accessibles de toutes les scènes.

Nous pourrions utiliser le dictionnaire global pour stocker le score et envoyer un message à la scène *Overlay* pour ordonner le rafraîchissement de l'affichage mais nous pouvons aussi envoyer le score directement dans un message grâce à l'*actuator* ***Message*** :



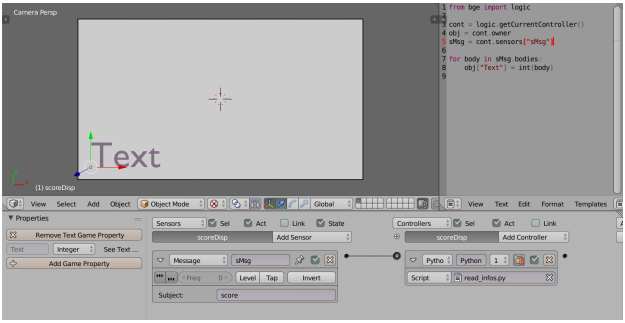

Lorsque l'option ***Body*** est sur ***Property***, nous spécifions le nom d'une ***Game Property*** de l'objet et la valeur de cette propriété est automatiquement envoyée, sous forme de texte, dans le corps du message.

Voici notre logique de jeu pour la mise à jour du score :



Nous voyons que le contact du joueur avec un objet bonus modifie le score, cela n'a pas changé. Mais nous avons ajouté une logique qui détecte toute modification du score et envoie un message à l'objet *scoreDisp*avec le nouveau score. C'est grâce au sensor ***Property > Changed***que cela est possible : ce type de *sensor* surveille une propriété de l'objet et génère une impulsion dès qu'il détecte un changement.

L'objet *scoreDisp* est dans la scène *hud* que avons ajoutée au début du jeu. Pour récupérer le message, nous utilisons un *sensor* ***Message*** :




Tel qu'il est configuré, le *sensor* détecte les messages qui sont envoyés à l'objet (plus précisément au nom de l'objet, ici `scoreDisp`) et dont le sujet est "*score*". Comme la nouvelle valeur du score est dans le corps du message, nous avons besoin de Python pour le lire car les briques ne permettent pas d'y avoir accès. Le script suivant transfère le contenu du message dans la propriété *Text* de l'objet, ce qui a pour effet de modifier l'affichage.

```
from bge import logic
 cont = logic.getCurrentController() obj = cont.owner sMsg = cont.sensor["sMsg"]

for body in sMsg.bodies:
    obj["Text"] = int(body)
```

Notons que le score est stocké à l'origine dans une propriété de type entier (*Integer*) car c'est nécessaire pour l'arithmétique du score. Il se transforme cependant en texte une fois copié dans le message car c'est le type obligé pour tous les messages (String). Comme la propriété *Text* de l'objet *scoreDisp* est aussi un entier nous devons reconvertir le message en entier avec l'expression `int(body)`. Nous aurions pu tout aussi bien choisir une propriété de type *String* pour *scoreDisp* de façon à éviter la conversion.

Grâce à la scène en *Overlay*, le score est toujours visible :


**Scrypthon !**

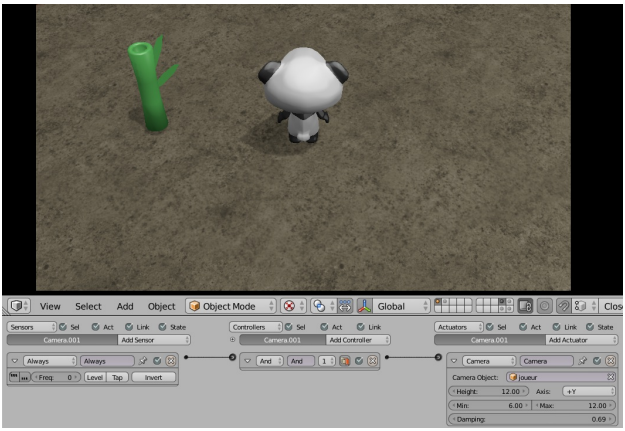
Comme nous l'avons vu au chapitre **Logique du jeu**, il est possible d'envoyer un message directement d'un script Python. Nous pourrions remplacer l'*actuator*`Message` par un *controller* Python qui exécute ce script :

```
from bge import logic

cont = logic.getCurrentController()
obj = cont.owner

# arguments: sujet, corps, destinataire
logic.sendMessage("score",str(obj["score"]),"Score")
```

## POINT DE VUE À LA 3ÈME PERSONNE

Popularisé par des jeux comme Tomb Raider (ou YoFrankie!), ce mode est peut être moins immersif mais permet au joueur d'admirer le personnage, ses animations, ses interactions avec le décor. La caméra flotte à quelques mètres du personnage, ce qui implique qu'il faudra porter un soin particulier à ce qu'elle ne traverse pas un mur, qu'un objet ne se place pas entre les deux, que les mouvements ne soient pas saccadés.

L'***`actuator Camera`***fournit une première approche. Sélectionnons notre caméra, ajoutons lui un actuator Camera déclenché par un *sensor Always*. Renseignons le nom de la cible (l'objet qui sera suivi par la caméra). Les noms des paramètres sont explicites, mais leur réglages demandent quelques tâtonnements.



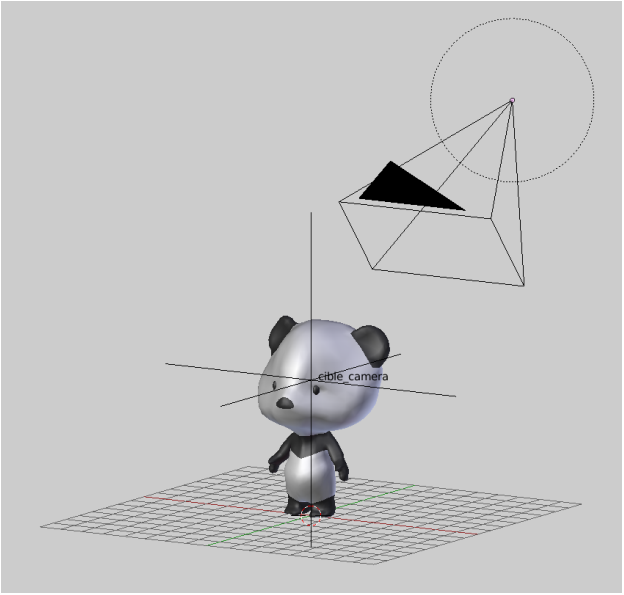

Toutefois on s'aperçoit vite que quels que soient les réglages, plusieurs problèmes restent présents :

- La caméra vise le **centre** de l'objet. Dans notre cas, elle cadre les pieds du personnage.
- La caméra passe à travers les murs
- Il arrive qu'un mur s'intercale entre la caméra et le personnage, bouchant la vue.

Voyons comment contourner ces petits soucis. Il s'agit souvent d'utiliser des techniques déjà abordées dans des chapitres précédents.

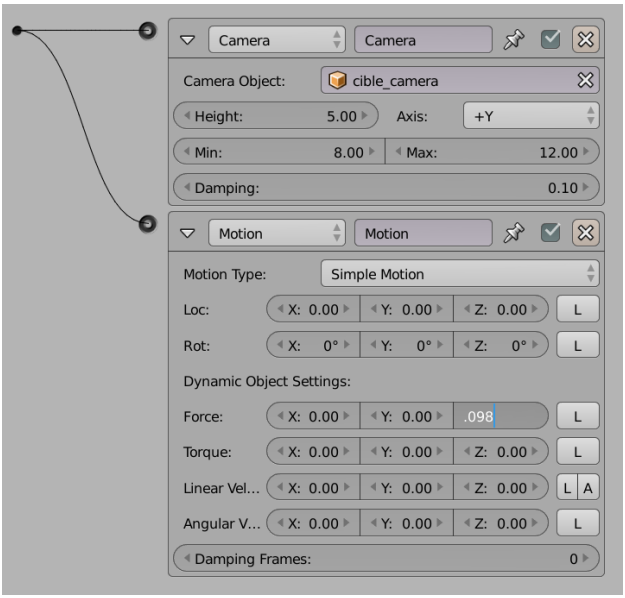
**Contrôler le cadrage**

On peut parenter un objet quelconque (un ***Empty*** convient parfaitement) à notre personnage, et déclarer cet objet dans l'***actuator*** `Camera`. Il est ensuite facile de le déplacer, pour régler finement le cadrage de la caméra.




**Éviter de traverser les murs**

On peut activer les collisions dans l'onglet ***Physics*** de la Caméra. Le mode ***Dynamic*** devrait convenir. Ces points sont détaillés dans la section Comportement des objets et personnages au chapitre Le comportement des objets de notre monde. Attention, la caméra devient alors sensible à la gravité, ce qui n'est généralement pas souhaitable. Pour pallier à cela, il est courant de lui donner constamment une force contraire (un *actuator*de type `Motion` avec une `Force` de 0.098 en Z).




**Déboucher la vue**

On peut utiliser les nœuds de matériaux, sur le matériel des murs, pour rendre transparent les parties proches de la caméra. Cette technique est détaillée dans le chapitre Matériaux dynamiques et textures animées de la section **Développer l'univers du jeu**.

## CHANGEMENT DE POINT DE VUE

L'idée ici est que plusieurs caméras ont été placées à des endroits clés d'une scène de jeu. Différentes fonctions, règles de jeu ou commandes de l'utilisateur déclenchent le passage de l'une à l'autre. Par exemple, il est courant, dans un jeu de voiture de pouvoir choisir sa vue, alors même que la voiture roule déjà, ou de rapidement regarder en arrière, lorsqu'on n'a pas de rétroviseurs. Il y a d'autres types de jeux qui activent le changement de caméra en fonction de la place du joueur sur la scène de jeu.

### Mise en place

Nous avons placé dans notre scène un certain nombre de caméras. Par soucis de clarté, nous avons donné à chacune un nom qui la caractérise bien. L'avantage de cette méthode est qu'on peut, dans le cas de caméras fixes, régler finement leurs points de vues.

### Briques Logiques

Pour passer d'une caméra à une autre, il faut juste créer un *actuator* ***Scene*** et le mettre en mode ***Set Camera***. Très simplement, il faut juste alors lui indiquer le nom de la nouvelle caméra à partir de la quelle on désire voir la scène. L'événement qui déclenchera cet *actuator* peut être une collision, une frappe au clavier ou le changement d'une *Game Property*, tout cela dépend des règles du jeu et de la jouabilité que l'on cherche à développer, mais le résultat activera la vue de la scène à partir de la caméra dont le nom est inscrit dans l'*actuator*.

### Scrypthon !

Voici un code pour lister toutes les caméras présentes dans la scène et passer à la suivante dans la liste des caméras disponibles lorsque le code est activé :

```
from bge import logic
scene = logic.getCurrentScene()

# Quel est l'indice de la camera active
indice = scene.cameras.index( scene.active_camera )

# Incrémenter cet indice
indice += 1

# Vérifier que l'indice n'est pas hors limites
if indice >= len( scene.cameras ) :
    indice = 0

# Changer la caméra active
scene.active camera = scene.cameras[indice]
```

Après import des modules et récupération de la scène courante, nous recherchons l'indice de la caméra active (`scene.active_camera`) dans la liste de toutes les caméras présentes dans la scène (`scene.cameras`). Nous incrémentons ensuite cet indice. Et nous vérifions que cette nouvelle valeur n'est pas hors limite par rapport au nombre de caméras disponibles dans la scène. Sinon, nous remettons l'indice à 0. En bref, nous bouclons sur toutes les caméras disponibles. Pour modifier la caméra courante, il suffit d'assigner la nouvelle caméra à l'attribute `scene.active_camera`.

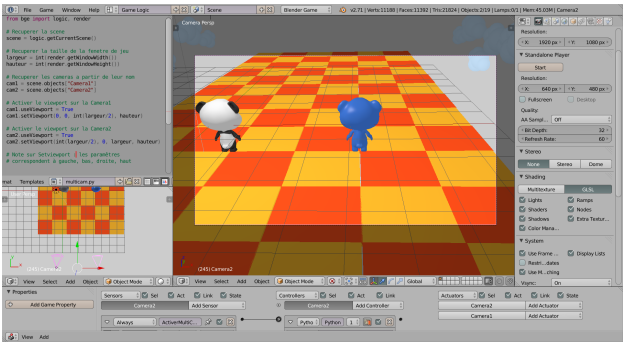
# 32. AFFICHER PLUSIEURS POINT DE VUES

En fonction du type de jeu et du type d'interactivité, il est possible et parfois même souhaitable d'avoir plusieurs points de vues affichés dans la même fenêtre. Par exemple, lorsque le jeu propose à deux joueurs de s'affronter dans un mode appelé écran partagé ou que nous désirons afficher plusieurs points de vue sur une même scène pour créer des effets graphiques intéressants.

Dans les faits, nous installerons deux caméras dans notre scène avec l'intention d'afficher ces deux points de vue dans la fenêtre principale, qui pour l'occasion sera divisée verticalement en deux parties égales.

## ÉCRAN PARTAGÉ

Le terme technique qui définit la zone d'affichage du jeu dans l'écran est le ***viewport***. Il n'existe malheureusement pas de `logic brick` pour activer ces fonctionnalités, nous utiliserons donc un script Python.



Le script peut être activé à partir de n'importe quel objet et doit être exécuté juste une fois : une fois en place les *viewport* persistent jusqu'à ce qu'ils soit supprimés explicitement. Le script doit désigner les caméras qui se partageront l'écran et définir les zones d'affichage pour chacune.

Voici le script, que nous détaillerons ensuite ligne par ligne.

```
from bge import logic, render
# Recuperer la scene
 scene = logic.getCurrentScene()
# Recuperer les cameras a partir de leur nom
 cam1 = scene.objects["Camera1"]
 cam2 = scene.objects["Camera2"]
# Recuperer la taille de la fenetre de jeu
 largeur = int(render.getWindowWidth())
 hauteur = int(render.getWindowHeight())
# Activer le viewport sur la Camera1
 cam1.useViewport = True
 cam1.setViewport(0, 0, int(largeur/2), hauteur)
# Activer le viewport sur la Camera2
 cam2.useViewport = True
 cam2.setViewport(int(largeur/2), 0, largeur, hauteur)
```

Après l'import des modules Python nécessaires, nous récupérons la scène dans laquelle se trouve l'objet.

```
 scene = logic.getCurrentScene()
```

Cet objet `scene` nous permet ensuite de récupérer les deux caméras grâce à leurs noms. Dans notre cas, nous avons au préalable choisi d'appeler les caméras Camera1 et Camera2.

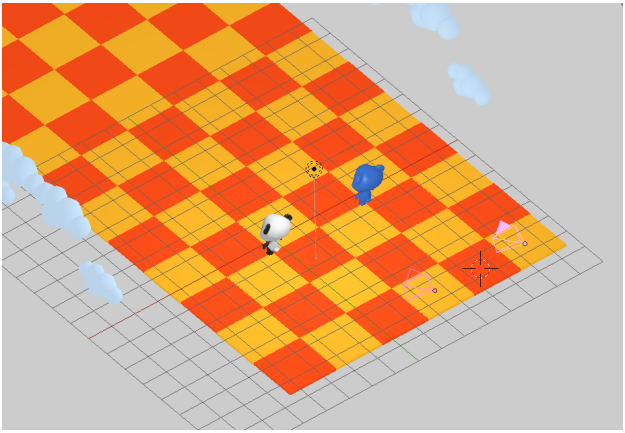
```
cam1 = scene.objects["Camera1"]
cam2 = scene.objects["Camera2"]
```

Il faut ensuite récupérer la taille de la fenêtre dans laquelle s'affiche le jeu. Que que le joueur soit en plein écran ou pas et que nous ayons défini une taille de fenêtre fixe ou pas, il est préférable de récupérer cette valeur directement par du code Python. C'est une méthode plus souple et plus sûre pour avoir toujours un écran partagé proportionné.

```
largeur = int(render.getWindowWidth())
```

```
hauteur = int(render.getWindowHeight())
```

Dans l'exercice, nous cherchons à diviser l'écran verticalement en deux parties égales. Camera1 utilise le système de *viewport* et affiche son point de vue dans un rectangle qui occupe la moitié gauche de l'écran. Camera2 suit le même principe, mais dans la moitié droite.



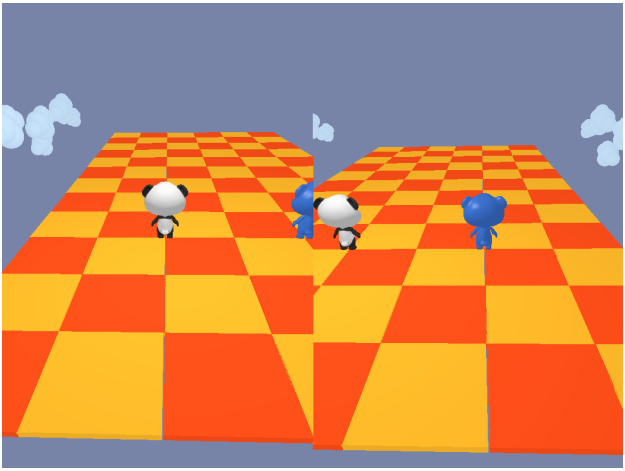
La méthode ***setViewport()*** ne tolère que des nombres entiers dans ses paramètres, c'est-à-dire les mesures en pixel des limites du rectangle dans lequel s'affiche la caméra. Attention, ces paramètres doivent suivre **un ordre très précis** : gauche, bas, droite, haut. Ce script indique que la première caméra prend toute la moitié gauche de l'écran.

```
cam1.useViewport = True
cam1.setViewport(0, 0, int(largeur/2), hauteur)
```

La deuxième caméra remplit l'espace restant.

```
cam2.useViewport = True cam2.setViewport(int(largeur/2), 0, largeur, hauteur)
```

En utilisant les valeurs de hauteur et de largeur de notre fenêtre de jeu, nous nous assurons ainsi de remplir tout l'espace de cette même fenêtre.



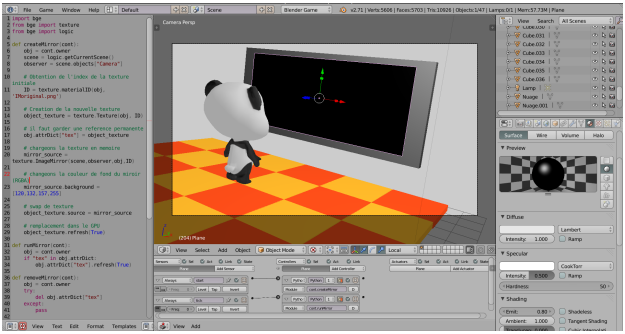
Voici quelques détails techniques à propos des *viewports* qu'il est utile de connaître:

- Si nous laissons la caméra courante inchangée et que nous activons une caméra secondaire en mode *viewport*, la caméra principale continue de remplir tout l'écran et la caméra secondaire vient en surimpression dans la zone du *viewport*.
- Il n'est pas obligatoire que les *viewports* soient jointifs comme dans notre exemple. Ils peuvent se chevaucher mais dans ce cas l'ordre de chevauchement n'est pas prévisible. Heureusement l'objet caméra possède une méthode `setOnTop` qui permet de placer cette caméra au sommet de l'empilement de *viewports*. En appliquant cette méthode sur toutes les caméras en commençant par celle qui doit être la plus en arrière, nous pouvons forcer un ordre de chevauchement.
- Les *viewports* n'effacent pas l'image déjà présente dans le *frame buffer*; ils s'affichent toujours en surimpression. Dans le cas de *viewports* jointifs comme dans notre exemple, cela ne porte pas à conséquence. Mais en cas de chevauchement, avec la caméra principale ou avec un autre viewport, l'image sous-jacente sera visible partout où il n'y a pas d'objet dans le *viewport*.

Pour supprimer un *viewport*, il suffit de mettre la propriété `useViewport` de la caméra correspondante à `False`.

Fichier référence : ecran_partage-panda.blend

## LE MIROIR



Le miroir est un cas particulier qui peut être utile par exemple dans le cas d'un jeu de voiture pour simuler un rétroviseur. L'idée est d'afficher le point de vue d'une caméra de la scène comme une texture d'un objet.

Pour cela, nous utiliserons un script Python écrit comme suit :

```
from bge import texture
from bge import logic

def createMirror(cont):
    obj = cont.owner
    scene = logic.getCurrentScene()
    observer = scene.objects["Camera"]

    # Obtention de l'index de la texture initiale
    ID = texture.materialID(obj, 'IMoriginal.png')

    # Creation de la nouvelle texture
    object_texture = texture.Texture(obj, ID)

    # il faut garder une reference permanente
    obj.attrDict["tex"] = object_texture

    # creons le miroir
    mirror_source = texture.ImageMirror(scene,observer,obj,ID)

    # changeons la couleur de fond du miroir (RGBA)
    mirror_source.background = [50,0,0,255]

    # echange de texture
    object_texture.source = mirror_source

    # remplacement dans le GPU
    object_texture.refresh(True)

def runMirror(cont):
    obj = cont.owner
    if "tex" in obj.attrDict:
        # rafraichissons l'image du miroir
        obj.attrDict["tex"].refresh(True)

def removeMirror(cont):
```

```
    obj = cont.owner
    try:
        del obj.attrDict["tex"]
    except:
        pass
```

À deux différences près, ce script ressemble beaucoup à celui présenté dans le chapitre Matériaux dynamiques et textures animées de la section Développer l'univers du jeu.

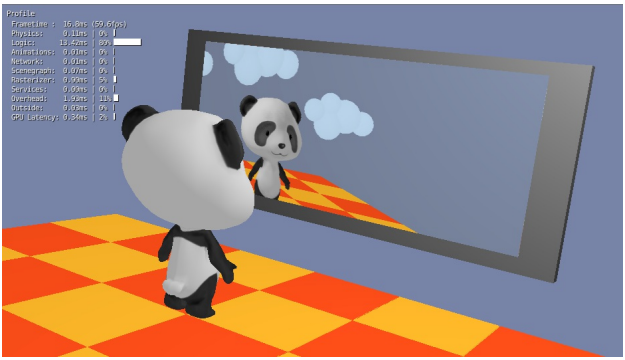
L'objet image est de type ***ImageMirror*** :

```
mirror source = texture.ImageMirror(scene,observer,obj,ID)
```

Le constructeur requiert les paramètres suivants :

- la scène courante ;
- l'objet qui observe le miroir (l'image du miroir est prise de son point de vue), généralement ce sera la caméra courante pour que le reflet ait l'air naturel ;
- l'objet sur lequel l'effet miroir sera projeté ;
- l'ID du matériel dont la texture sera remplacée.

Il est nécessaire de rafraîchir l'image du miroir fréquemment pour tenir compte du déplacement de l'observateur. Ceci est réalisé par la fonction `runMirror` qui doit être exécuté à chaque *frame* ou chaque fois que l'observateur bouge.




Pour réaliser cet effet miroir, le BGE effectue un rendu spécial à partir d'un point symétriquement opposé à l'observateur par rapport au miroir. Ce rendu est appliqué à l'objet via la texture. Pour que le reflet ne soit pas déformé, il faut que la texture recouvre au mieux la surface de l'objet miroir. Nous obtiendrons ce résultat grâce à la fonction ***UV mapping > Project from View (Bounds)***.

Fichier référence : mirroir-panda.blend

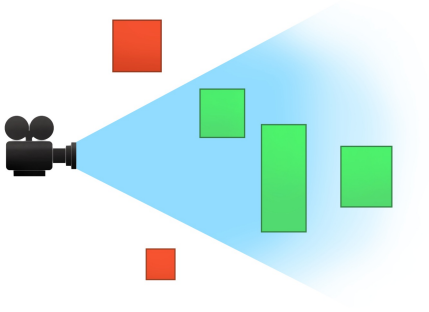
# TRAVAILLER LE RENDU DU JEU

**33.** IGNORER LES OBJETS HORS CHAMP
**34.** PRÉ-CALCUL DES TEXTURES ET DE LA LUMIÈRE
**35.** PROGRAMMER UNE SURCOUCHE GRAPHIQUE

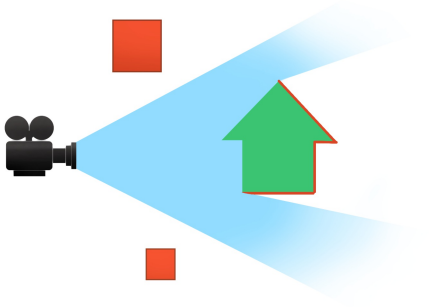
# 33. IGNORER LES OBJETS HORS CHAMP

Le terme *culling* signifie la suppression ciblée de l'affichage de certains éléments graphiques dont on est certain qu'ils ne sont pas visibles.

Deux techniques simples sont toujours actives dans le BGE pour cela.

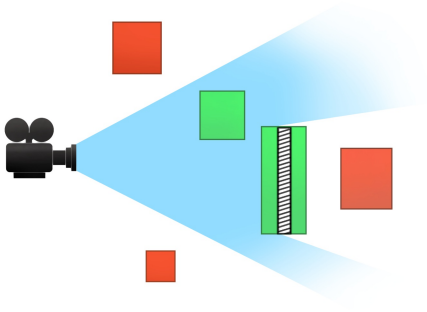


- ***frustrum culling***. Les objets qui sont entièrement hors du champ de la caméra ne sont pas envoyés au processeur graphique.



- ***back face culling***. Les faces dont les normales ne sont pas dirigées vers la caméra ne sont pas rendues par le processeur graphique (mais elles sont néanmoins envoyées à la carte).

La technique d'optimisation appelée *Occlusion Culling* consiste à éviter d'envoyer au processeur graphique les objets qui sont dans le champ de la caméra mais invisibles parce qu''ils sont entièrement cachés par des objets spéciaux appelés *occluders*.



Pour être efficace nous devons choisir comme *occluders* des objets de grande taille susceptibles de cacher beaucoup d'autres objets. Un choix idéal seraient les murs et la toiture d'un bâtiment qui cacheraient tous les objets situées à l'intérieur lorsqu'on se trouve à l'extérieur et vice et versa.

Les *occluders* sont des objets marqués comme tels dans le panneau `Physics (Properties > Physics > Physics Type > Occlude)`. Ce choix désactive toute autre option physique car les *occluders* sont d'office des objets sans physique. Pour cette raison les *occluders* doivent, de préférence, être construits spécialement à cet effet. Par exemple, dans le cas du bâtiment cités plus haut, plutôt que de choisir le *mesh* du bâtiment, nous créerons un volume simple qui épouse la forme du bâtiment mais qui est situé au sein des murs de sorte que le bâtiment lui-même soit à la fois à l'intérieur et à l'extérieur de l'*occluder* mais que les autres objets soient dedans ou dehors.

Cette technique est nettement plus sophistiquée que les autres méthodes de *culling*, car le BGE doit faire une sorte de rendu simplifié de la scène pour voir quels objets sont cachés. Pour cette raison il faut faire des tests pour évaluer l'intérêt de la méthode au cas par cas.

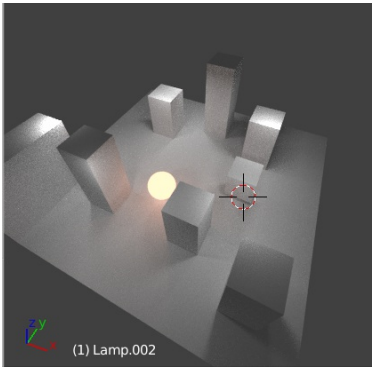
L'`Occlusion Culling` est activée dans le panneau `Properties > World > Physics`. L'option `Resolution` spécifie la résolution du rendu simplifié. La valeur par défaut (128) est sans doute adéquate pour la plupart des cas. Un valeur trop basse diminue les calculs mais augmente les marges ce qui réduit le nombre d'objets occultés. Une *value* trop élevée augmente inutilement le calcul, sans gain supplémentaire d'occultation.

Au minimum un `occluder` doit être activé dans la scène pour activer cette fonctionnalité.

# 34. PRÉ-CALCUL DES TEXTURES ET DE LA LUMIÈRE

Le *baking*, ou cuisson des textures, est une méthode permettant de simplifier le calcul des textures et des lumières par le moteur de rendu. Cette technique fige la surface des objets telle qu'elle est rendue par le moteur non temps réel de Blender (*Internal* ou *Cycles*) et en fabrique une texture. Par exemple, cela permet de diminuer le nombre de lumières dynamiques nécessaires pour illuminer un espace, simplifier la texture ou la géométrie d'un personnage détaillé. De plus, depuis la version 2.71, le *baking* est capable d'utiliser le rendu de *Cycles*, ce qui permet un mode d'illumination des scènes très réaliste, et donne à des scènes simples un rendu de grande qualité, impossible à obtenir avec le rendu classique du *Game Engine*.

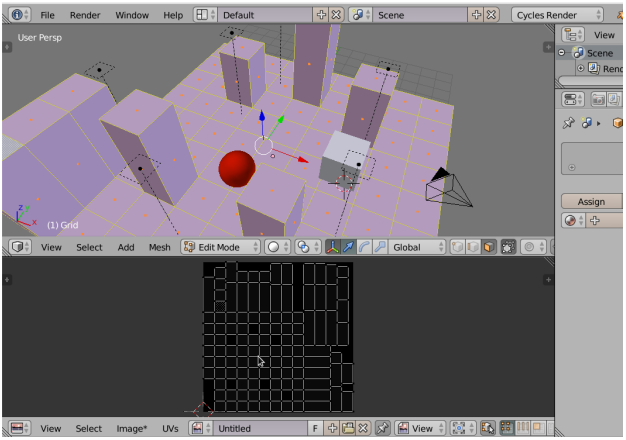
Le principe du *baking* d'un objet commence par le texturage et l'éclairage de la scène en mode rendu ***Cycle***. La cuisson se réalise sur une texture mapée en UV. Ensuite cette texture est utilisée à la place du matériau original. L'éclairage de la scène ainsi que le texturage des objets se réalise avec les précisions souhaitées. Nous allons voir ici comment faire pour l'illumination d'un espace.



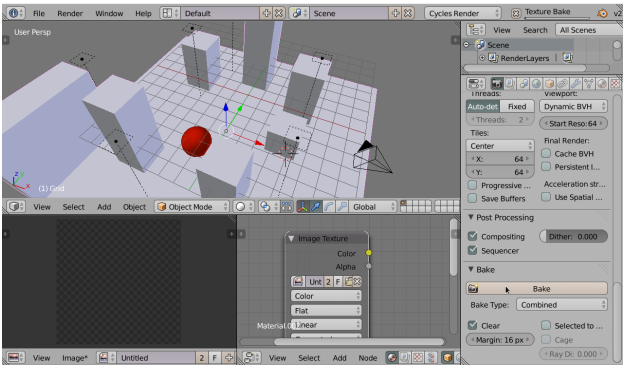

*Preview* d'un rendu *Cycle* d'une scène comportant 6 lampes de type *area*, ainsi qu'une sphère dont le matériau émet de la lumière.

## PRÉPARATION DE LA SCÈNE

La scène doit être installée comme dans un rendu classique. Lorsque l'effet d'éclairage désiré est obtenu, le *baking* se réalise en ajoutant une carte UV dédiée et assignée à une nouvelle texture.



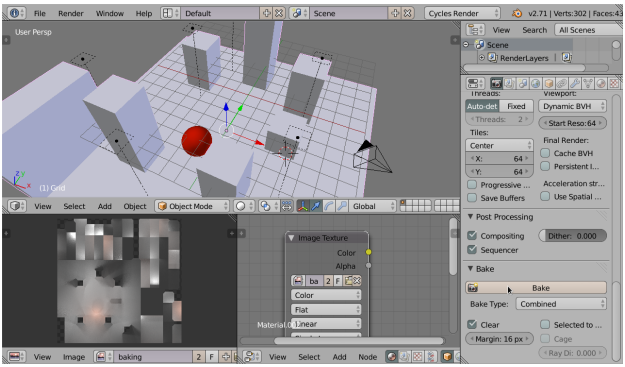

Ajoutons la texture dans son matériau (fenêtre `nodes matériau`) pour le *baking*.




## *BAKING*

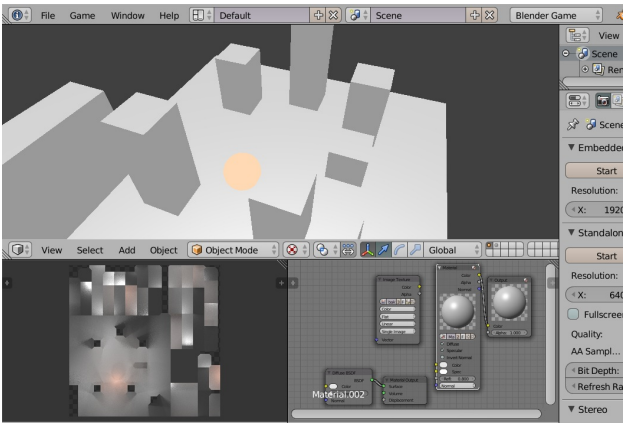
Sélectionnons notre objet puis dans la fenêtre des `nodes` de son matériau, sélectionnons la texture qui doit être remplie.

Dans l'onglet`Rendu`, section `Baking`, enclenchons le`bouton` ***Bake***. Il faut être patient, car le rendu n'est pas visible et peut prendre un certain temps. Une fois terminée, la texture prend les couleurs de l'éclairage complexe. Enregistrons-la pour ne pas la perdre.

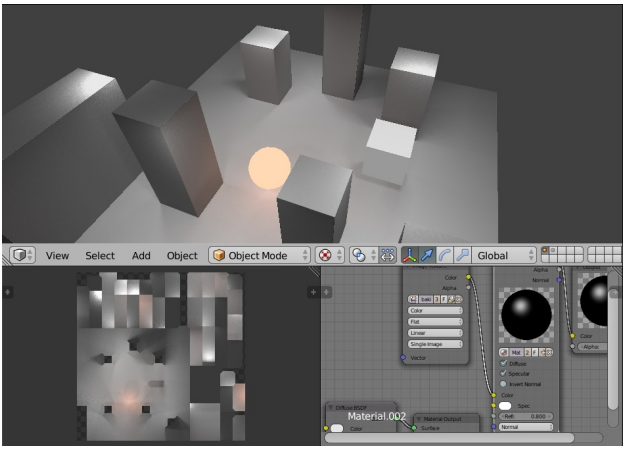



## DANS LE GAME ENGINE

Repassons en mode *Game Engine*et modifions le matériau dans la fenêtre`node`, en lui ajoutant un *input* matériau, ainsi qu'un `node` **`output`**.




Enfin relions la texture du *baking* à l'entrée couleur de la texture. Le résultat est maintenant visible dans le BGE.




## AUTRES POSSIBILITÉS

Il est également possible de *baker* des textures procédurales non supportées par le *Game Engine*.

Dans le `node editor`, une texture générée par association de différentes *patterns* grâce aux *nodes* mais non supportée dans le *Game Engine* est transformée en image *bitmap* utilisée dans le processus de *baking.*

Un autre usage est de simplifier la géométrie d'un objet en faisant le *baking* des reliefs dans une *normal map* qui sera utilisée sur un modèle de moindre détail. La surface aura l'air d'être détaillée alors qu'elle ne comporte que peu de faces. Ceci est très utile pour les personnages par exemple.

# 35. PROGRAMMER UNE SURCOUCHE GRAPHIQUE

Les *hooks* (crochets en français) font partie des techniques avancées d'affichage. Ils permettent d'exécuter du code à chaque *frame* alors que le contexte OpenGL est prêt pour l'affichage. Cela signifie qu'il est possible d'ajouter des éléments ou des effets graphiques dans la fenêtre de jeu par voie de programmation sans passer par le moteur graphique du jeu. Les possibilités sont vastes mais le prix à payer est qu'il faut passer par OpenGL. Voyons plus en détails comment cela se passe.

## ANATOMIE D'UN HOOK

U n *hook* est une fonction Python que nous devons écrire et dans laquelle nous mettrons du code OpenGL, soit directement grâce au module `bgl` de Blender qui exporte la plupart des fonctions, soit indirectement via notre API Python préférée pour générer du graphisme OpenGL.

Nous devons enregistrer cette fonction auprès de la scène courante :

- Fonctions à exécuter avant le rendu des objets.

```
scene.pre_draw.append(my_function)
```

- Fonctions à exécuter après le rendu des objets.

```
scene.post_draw.append(my_function)
```

*Les attributs `pre_draw` et `post_draw` sont des listes pour permettre à différentes parties du code d'enregistrer leur fonctions. Pour supprimer une fonction hook existante, il suffit d'utiliser la méthode `remove` : `scene.post_draw.remove(my_function)`*

Une fois enregistrée auprès de la scène, la fonction sera appelée par le moteur graphique à chaque *frame* avec les modalités suivantes :

- **Pas d'argument**.

  La fonction est appelée sans argument. Cependant, comme elle est appelée dans le cadre de la scène où elle a été enregistrée, elle a accès à la scène de manière habituelle (via la fonction `logic.getCurrentScene()`) et si nécessaire à tous les objets qu'elle contient.

- **Fonctions `pre_draw` : matrice chargée, *frame buffer* vide.**

  Le contexte OpenGL est prêt pour des commandes d'affichage, le *frame buffer* est vide et les matrices OpenGL actives sont telles que les *vertices* que nous enverrons à OpenGL devront être en coordonnées `world`. Nous avons bien sûr la possibilité de charger de nouvelles matrices mais nous devrons impérativement rétablir l'état OpenGL avant le `return`. Tout ce que nous mettrons dans le *frame buffer* sera combiné avec le rendu des objets.

- **Fonctions `post_draw`: matrice indéterminée, *frame buffer* rempli**.

  Les fonctions `post_draw` héritent d'un *frame buffer* contenant le rendu de tous les objets de la scène plus le résultat des éventuels filtres 2D. Elle ne peuvent compter sur des valeurs particulières de matrices OpenGL, elle devront donc positionner les matrices selon leurs besoins sans nécessité de rétablir l'état OpenGL avant le `return`.

- **En cas de`viewport`**

  Les fonctions *hook* sont compatibles avec les *viewports* dans les limites suivantes :

  - Les fonctions `pre_draw` sont exécutées pour chaque `viewport`. Elles héritent du contexte OpenGL du *viewport* courant. Cela permet d'agir dans chaque `viewport`. Cependant, une fonction qui ne devrait écrire que dans un `viewport` devrait d'abord tester le `viewport` avec la fonction OpenGL appropriée (il n'existe aucun moyen de connaître le `viewport` courant avec l'API du BGE).
  - Les fonctions `post_draw` ne sont exécutées qu'une seule fois par *frame* à la fin du rendu de tous les *viewports*. Elles héritent de la totalité de la fenêtre OpenGL.

### Exemple : dessiner un rectangle en OpenGL.

Nous allons afficher une sorte de barre rouge et semi transparente à l'écran. Cette routine doit s'effectuer après le rendu de l'image. Pour être sûr que la routine s'effectuera bien après le rendu de l'image la fonction **`write()`** s'enregistre dans la liste **`post_draw`** contenu dans l'objet **`scene`** récupéré juste avant. **`post_draw`** est une liste de fonctions (plus exactement `post_draw` est une liste de ***callable***) qui seront exécutées les unes après les autres, et cela après chaque rendu.

```
from bge import logic, render
import bgl

def init():
 scene = logic.getCurrentScene()
 scene.post_draw = [write]

def write():
 """ Write on screen """
 scene = logic.getCurrentScene()
 width = render.getWindowWidth()
 height = render.getWindowHeight()

# OpenGL setup
bgl.glMatrixMode(bgl.GL_PROJECTION)
bgl.glLoadIdentity()
bgl.gluOrtho2D(0, width, 0, height)
bgl.glMatrixMode(bgl.GL_MODELVIEW)
bgl.glLoadIdentity()

# Draw a 2D rectangle to make the fonts stand out
bgl.glEnable(bgl.GL_BLEND)# Enable alpha blending
bgl.glBlendFunc(bgl.GL_SRC_ALPHA, bgl.GL_ONE_MINUS_SRC_ALPHA)
view_buf = bgl.Buffer(bgl.GL_INT, 4)
bgl.glGetIntegerv(bgl.GL_VIEWPORT, view_buf)
view = view_buf
bgl.glMatrixMode(bgl.GL_PROJECTION)
bgl.glLoadIdentity()
bgl.gluOrtho2D(0, view[2], 0, view[3])
bgl.glMatrixMode(bgl.GL_MODELVIEW)
bgl.glLoadIdentity()
bgl.glBegin(bgl.GL_QUADS)
bgl.glColor4f(.4, 0, 0, 0.4)
bgl.glVertex2f(5, (height/2))
bgl.glVertex2f(width -5, (height/2))
bgl.glVertex2f(width - 5, (height/2) + 21)
bgl.glVertex2f(5, int(height/2) + 21)
bgl.glEnd()

def main(cont):
 own = cont.owner
 if not "init" in own:
 own["init"] = True
 init()
```

Ressource : OpenGL_postdraw.blend

Il est aussi possible d'afficher du texte de la même façon, via le module blf.

# ANNEXES

# 36. GLOSSAIRE

Actuator
: Élément logique qui une fois activé par un *controller* effectue les actions dans le jeu.

Algorithme
: Suite d'instructions permettant de résoudre un problème

API
: Interface de programmation qui fournit des méthodes et des fonctions pour exploiter les fonctionnalités Blender

Asset
: Ressource numérique (graphique, sonore,...), communément un objet.

Baking
: En jeu vidéo, il s'agit de pré-calcul (les ombres la plupart du temps).

BGE
: Blender Game Engine ou moteur de jeu de Blender. Le BGE est interne à Blender et existe aussi sous forme d'un programme autonome: le Blenderplayer.

Binaire
: Fichier non-lisible (par un humain) sans le logiciel adapté (à l'inverse d'un fichier texte)

Blender Game
: Mode d'édition qui active les panneaux et options spécifiques au BGE. Ce mode doit être utilisé de préférence à `Blender Render` lors de la création d'un jeu.

Blenderplayer
: (De son petit nom *player*) Version réduite de Blender qui ne contient que le BGE et les routines nécessaires pour exécuter un jeu. Lorsque le *player* ouvre un fichier .blend, il lance automatiquement le BGE. Le Blenderplayer existe sous forme d'un exécutable autonome livré avec Blender ou sous forme d'un exécutable qui combine le *player* avec le .blend courant. Cette version empaquetée du *player* est produite par Blender grâce à l'add-on *Save as runtime*.

Blender Render
: Mode d'édition qui active les panneaux et options spécifiques au moteur de rendu classique de Blender (`Blender Internal`).

Brique Logique ou *Logic Brick*
: Éléments logique simples pouvant être connectés pour former une logique complexe. Les briques logiques se répartissent en trois catégories: détecteurs (ou *sensors*), contrôleurs (ou *controllers*), actionneurs (ou *actuator*).

Bullet
: Moteur physique libre utilisé par Blender

Controller
: Élément logique activé par les *sensors* et dont le rôle est de transmettre, après traitement éventuel, les impulsions logiques aux *actuators*.

CPU
: *Central Processor Unit* ou processeur central. C'est le processeur de l'ordinateur sur lequel Blender et l'ensemble des programmes s'exécutent. Le CPU a généralement beaucoup moins de puissance de calcul que le GPU.

Cycles Render
: Mode de rendu qui active les panneaux et options spécifiques au moteur de rendu *Cycles*.

Embedded Player
: Le nom donné au BGE lorsque celui-ci s'exécute à l'intérieur de Blender (dans la vue 3D).

Game Engine
: Moteur de jeu, incluant généralement un moteur graphique et un moteur physique

Game Logic
: L'ensemble des briques logiques.

GLSL
: Langage de programmation pour les *shaders* (API OpenGL).

Gnu GPL :
: Gnu General Public Licence, ou licence publique générale Gnu, la licence libre la plus connue et la plus utilisée : http://fr.wikipedia.org/wiki/Licence_publique_générale_GNU. Ce manuel est publié sous cette licence.

GPU
: *Graphical Processor Unit* ou processeur graphique. Le GPU réside dans la carte graphique et est spécialisé dans le calcul de pixels. Il développe généralement une puissance de calcul considérable pour les calculs matriciels.

Incrémenter
: Ajouter une valeur entière (1 pour une valeur numérique )

iTaSC
: Génère des mouvements en respectant des contraintes.

Itération
: Voir itérer

Itérer
: Répéter une action

Keyframe
: Définit une image-clé ou clé d'animation permettant de définir les différents états essentiels d'un objet lors de son animation.

Logic Editor
: L'éditeur logique est l'éditeur type contenant les outils graphiques permettant de mettre en place l'interactivité.

Mesh
: Maillage d'un objet

MMORPG (Massively Multiplayer Online Role Playing Game)
: Jeu de rôle multijoueur en ligne. Les « MMO » les plus connus sont World of Warcraft, Ultima online, Everquest, Anarchy Online, Eve Online.

MOOC (Massive Online Open Course)
: Formation en ligne ouverte à tous.

Multitexture
: Gestion de plusieurs couches de textures (permettant l'utilisation de *normal map* par exemple).

Sensor
: Éléments logiques qui détectent des événements et produisent des impulsions logiques, positives ou négatives à destination des *controllers*.

Nœuds matériaux
: Utilisation de l'OpenGL sans programmation.

Normal map
: Texture de relief utilisée afin d'augmenter le détail des objets.

OSC
: Protocole de transmission de données

Parenter
: Créer un lien parent-enfant afin que les modifications sur le parent influencent également sur l'enfant.

Pitch
: Hauteur du son.

PNJ
: Personnage non joueur. Décrit les personnages du jeu non manipulé par un joueur, autrement dit, par des algorithmes.

RTSP (Real Time Streaming Protocol*)*
: Protocole de *streaming* en temps-réel

SOC
: System on Chip, système embarqué.

Solver
: Résout des calculs afin de bien positionner les os d'une armature en fonction de contraintes.

Sprite
: Éléments graphiques, généralement déclinés en plusieurs versions, afin d'être disposés en mosaïques pour réaliser des éléments graphiques plus grands ou arrangés en séquences pour produire des animations.

Standalone Player
: Voir Blendeplayer.

Texture animée
: Opération qui consiste à faire varier une texture au cours du temps par remplacement de l'image dans le GPU.

Texture procédurale
: Image directement créée par un algorithme, souvent utilisée pour simuler un élément naturel comme du bois, de la pierre ou du métal.

URL
: Permet de localiser des ressources (locales ou en ligne)

.blend
: Extension et format de fichier de Blender

.wav
: Extension et format de fichier audio (avec pertes)

.mp3
: Extension et format de fichier audio compressé (avec pertes)

.ogg
: Extension et format de fichier audio libre, compressé en Vorbis (avec pertes)

.aif/.aiff
: Extension et format de fichier audio

.flac
: Extension et format de fichier audio libre et sans pertes

# 37. À PROPOS

Le cœur et la structure de l'ouvrage de plus de 200 pages ont été réalisés en 5 jours dans le cadre d'un Libérathon qui s'est tenu à Bruxelles du 4 au 8 août 2014, dans les locaux de F/LAT, grâce à l'initiative et avec le soutien de l'Organisation internationale de la Francophonie (http://www.francophonie.org).

Expérimentée et popularisée par la **Floss Manuals Foundation** dans le cadre de ses activités de création de manuels multilingues sur les logiciels et pratiques libres, la méthodologie du **Libérathon** permet de rédiger en un temps très court des livres de qualité. **Floss Manuals Francophone** est une association loi 1901 dont l'objectif est d'organiser et de faciliter l'écriture ou la traduction de documentations en français.

Un groupe de quatorze co-auteurs (graphistes 3D, modeleurs, animateurs, illustrateurs, développeurs) provenant de France et de Belgique ont travaillé ensemble en veillant à représenter la diversité des utilisatrices et utilisateurs francophones.

Co-rédacteurs et facilitatrice présents lors du booksprint :

- Jean-Michel Armand, developpeur Django (http://j-mad.com/blog/).
- Camille Bissuel, graphiste (http://nylnook.com/fr).
- Benoît Bolsée, développeur Blender.
- Quentin Bolsée, développeur de jeu.
- Elisa de Castro Guerra, formatrice et facilitatrice (http://activdesign.eu).
- Julien Deswaef, artiste numérique (http://xuv.be/).
- Ronan Ducluzeau, graphiste.
- Cédric Gémy, graphiste et formateur (http://activdesign.eu).
- Olivier Meunier, artiste numérique et formateur (http://f-lat.org/).
- Lucile Patry, étudiante graphiste.
- David Revoy, illustrateur et concept artiste (http://www.davidrevoy.com/).
- Thibaud Sertier, graphiste (http://thibaudsertier.com/).
- Maxime Gourgoulhon, étudiant en informatique (http://maxime.gourgoulhon.fr/).
- Jimmy Etienne, étudiant en informatique (http://osxia.org/).

## UN MANUEL LIBRE DISPONIBLE SOUS PLUSIEURS FORMATS ET SUPPORTS

Écrit en collaboration, ce manuel d'initiation au **Blender Game Engine** a été inspiré par les valeurs du libre. Il est disponible depuis le site de **Floss Manuals** sous plusieurs formes : livre imprimé, pages web, PDF et ePub, ce dernier format permettant de le consulter facilement sur des appareils portatifs.

Par ailleurs, la version électronique de cet ouvrage évoluera encore au fur et à mesure des contributions et des avancées du logiciel. Pour consulter la dernière version actualisée, nous vous invitons à visiter régulièrement le volet francophone de Floss Manuals sur le site http://fr.flossmanuals.net/

N'hésitez pas à votre tour à améliorer ce manuel en nous faisant part de vos commentaires dans la liste de diffusion francophone de Floss Manuals. Si vous avez des talents de rédacteur et une bonne connaissance du *Game Engine* de Blender, l'envie d'ajouter une remarque ou un détail, inscrivez-vous en tant que contributeur pour proposer la création de nouveaux chapitres ou améliorer les chapitres existants.

Vous consultez l'édition publiée le 16 septembre 2014.

# 38. RESSOURCES

Une archive complète des exemples utilisés dans ce livre est téléchargeable à partir de cet endroit: https://github.com/flossmanualsfr/exemples-blender-pour-le-jeu-video/archive/master.zip

Certaines captures d'écran de ce livre ont été faites à partir (ou avec des éléments provenant) du jeu "Péril en Daiza". Ce jeu est disponible sur le site http://perilendaiza.com/. Vous êtes également cordialement invités à participer à son développement en proposant des mises-à-jour sur https://github.com/flossmanualsfr/peril-en-daiza.

Sur le site officiel de blender dans la section *Support*, il est possible de télécharger des fichiers d'exemple dans l'encart *Demo Files & FAQ*. L'archive *regression suite* contient plus de 70 fichiers d'exemples pour tous les usages de Blender. Les fichiers relatifs au *Blender Game Engine* y sont regroupés en 3 dossiers : *gameengine_logic*, *gameengine_physics*, *gameengine_visual*.
http://www.blender.org/support/

*Ces trois dossiers, spécifiques au Game Engine, ont été ajoutés au dossier d'exemples fournis avec ce livre. Ils sont aussi consultables individuellement ici: https://github.com/flossmanualsfr/exemples-blender-pour-le-jeu-video/tree/master/07_Annexes*