Simulation de quadricoptère avec Blender

Benoît Saccomano & Michael Muré

27/05/2011

1 Introduction

Dans le cadre de notre projet nous avons été amenés à modéliser en 3D un quadricoptère de la marque Parrot™ sous Blender™. Ce document a pour but d'expliciter les différentes étapes que nous avons suivi afin de réaliser ce travail.

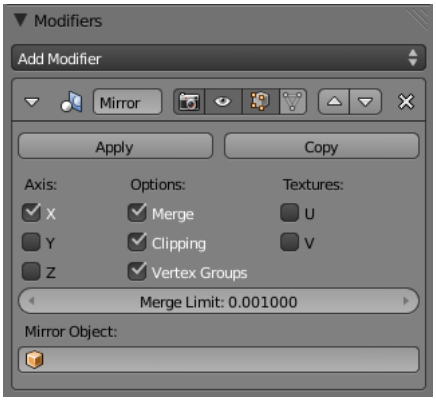
Dans un premier temps on reviendra sur la modélisation (au sens graphique) du modèle. Ensuite on décrira la méthode suivie afin d'intégrer à ce modèle un comportement physique « primitif », c'est à dire qui permet de commander indépendament les 4 moteurs.

On reviendra enfin sur les différentes extensions que nous avons ajoutées, à savoir un HUD (*head's up display*), un comportement plus évolué et la possibilité de contrôler le quadricoptère à l'aide d'un joystick.

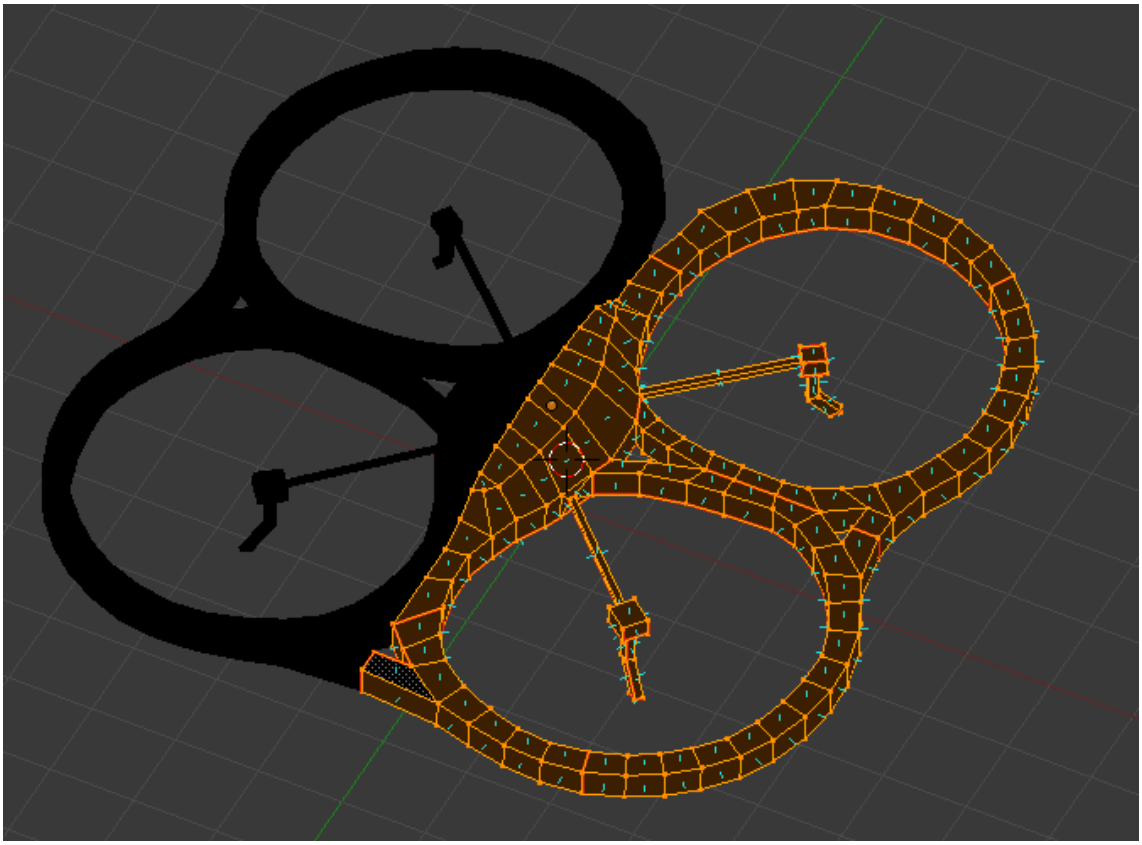
Quelques petites précisions supplémentaires avant de commencer : nous avons utilisé la version 2.57 de Blender, le design du modèle que nous avons pris est celui du drone vendu par Parroy™, le contrôle de ce modèle est apporté dans le moteur de jeu de Blender™ via des scripts (en Python) écrits par nos soins, en combinaison avec le moteur physique interne de Blender™, Bullet.

2 Modélisation

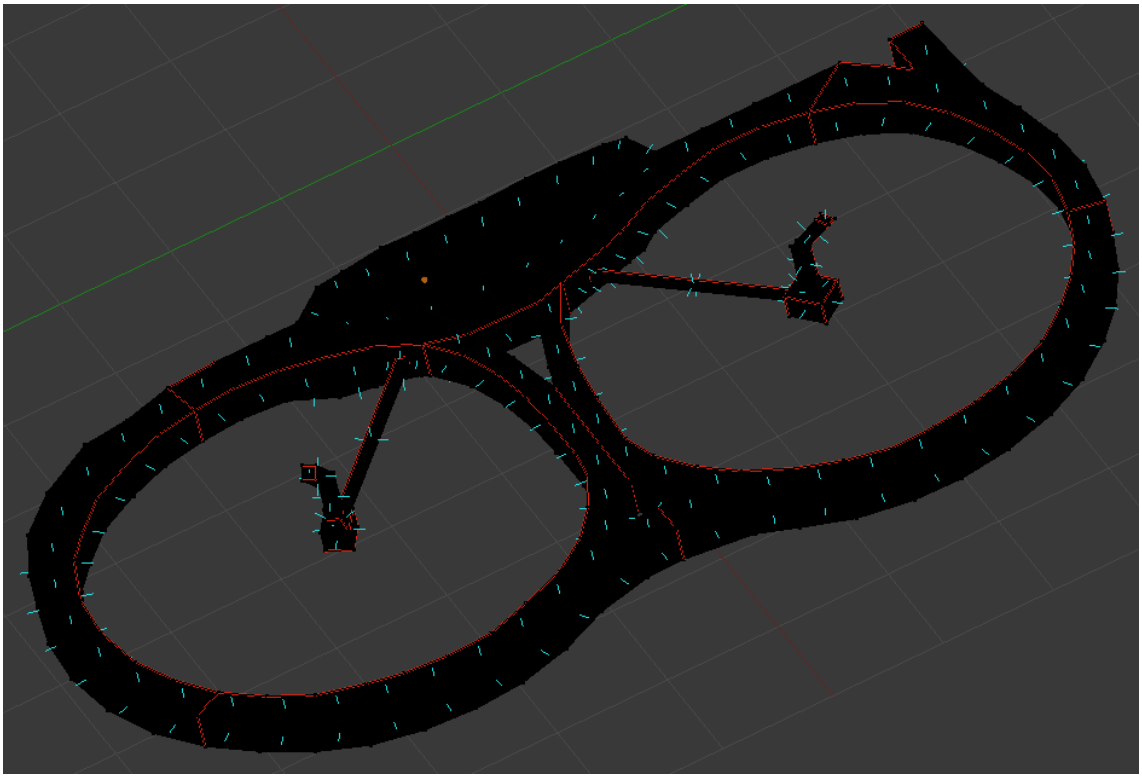
Nous avons modélisé le quadricoptère grâce à Blender. Comme le modèle est symétrique, nous avons utilisé un outil de miroir (*modifier mirror*), qui permet de ne modéliser qu'une moitié du modèle. La modélisation a été faite à partir d'une photo vue de dessus, trouvée dans un moteur de recherche. Nous avons mis cette photo en image de fond dans Blender, et modélisé par dessus pour avoir les bonnes proportions.



Le modèle final complet (les deux cotés) comporte 652 points (vertices), et 668 faces, ce qui est assez léger pour un affichage en temps réel.



Nous avons ensuite déplié en 2D, c'est à dire que pour chaque point du modèle est associée une coordonnée en 2D dans l'espace de la texture. Ces coordonnées s'appellent les coordonnées UV. Pour faire cela, le principe est de marquer certaines arrêtes (*edge*) du maillage comme étant des coutures (*seam*). Ces coutures sont visibles en rouge sur la capture d'écran suivante. Le dépliage se fait ensuite grâce à un algorithme interne de Blender qui va déplier (unwrap) au mieux ces surfaces.



Une fois le dépliage terminé, nous avons texturé le modèle. Ce texturage s'est fait en 3 étapes.

1. Nous avons projeté la photo vue de dessus du modèle réel sur le modèle virtuel grâce à Blender. Cela nous a permis d'obtenir les décorations du dessus du modèle facilement, déformées de la bonne façon pour coller à notre dépliage.
2. Nous avons ensuite complété la texture grâce au logiciel de traitement d'image The Gimp. Nous avons utilisé notamment l'outil de clonage pour dupliquer la texture de polystyrène noir partout où c'était nécessaire.
3. Nous avons calculé l'ombrage que le modèle fait sur lui même (*ambient occlusion*). C'est à dire que certaines parties du modèle en cachent d'autres, ce qui crée des zones plus sombres que les autres. Blender permet de projeter sur une texture ces zones d'ombre, que nous avons ensuite combinées au reste de la texture. Cela permet d'avoir un peu plus de réalisme pour un coût complètement nul au rendu.

La capture d'écran suivante présente la texture finale avec les différentes couches, ainsi que le dépliage du modèle.

3 Contenu de la scène

La scène finale comporte plusieurs objets :
- Le modèle du quadricoptère (Quadri)
- Un niveau basique où évolue le modèle (Level)
- Une caméra
- Un ensemble d'objets parenté à la caméra qui sert à afficher le HUD, que nous détaillerons plus loin.
- Un modèle haute résolution du quadricoptère (Quadri.high), qui n'est pas utilisé actuellement.
- Un modèle ultra-simplifié du quadricoptère (32 points, 30 faces), qui est utilisé comme modèle de collision pour le moteur physique (non actif actuellement).

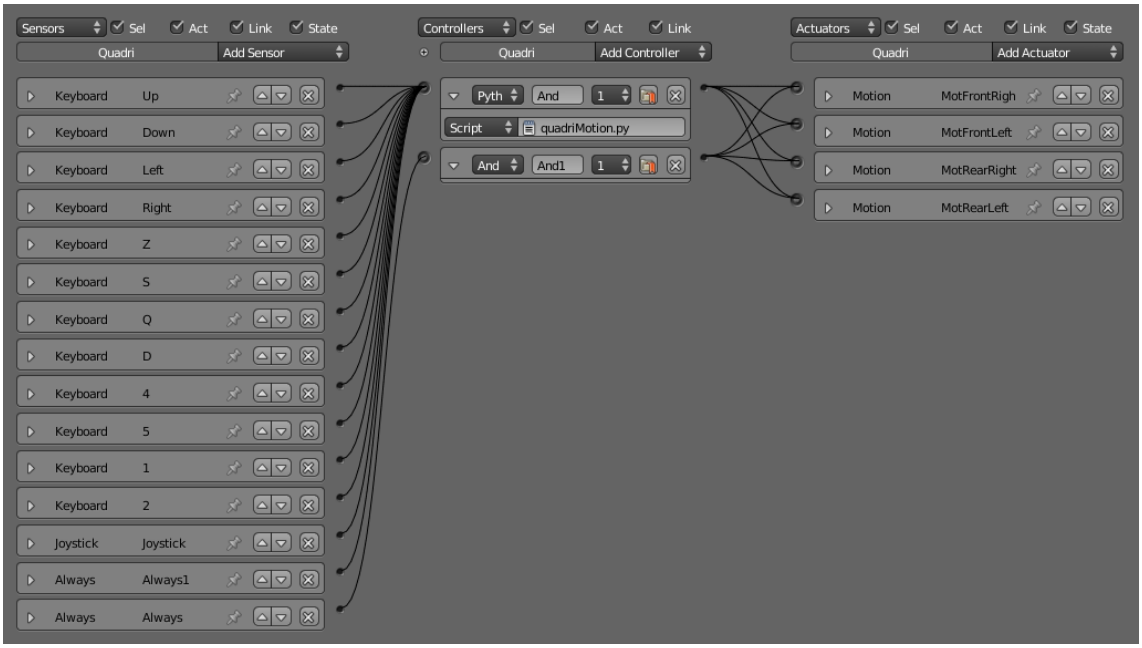
4 Simulation physique du quadricoptère

4.1 Principe global

Afin de pouvoir donner son comportement au modèle dans le mode de jeu de Blender™, il a fallu définir plusieurs types d'objets sur notre modèle de quadricoptère.
- les *sensors* qui détectent une action sur une entrée et génèrent un évènement. Dans notre cas cela correspond aux touches du clavier, ou à un joystick.
- les *controller* qui sont le plus souvent des expressions logiques (OR, NAND...) faisant la liaison entre les sensor et les controller. Dans notre cas, la complexité du comportement nous a poussé à utiliser des scripts Python en tant que controller.
- les *actuators* qui se chargent de modifier le modèle suivant les paramètres du controller. Dans notre cas, ils sont au nombre de 4, un pour chaque moteur du quadricoptère.

Ces différents éléments sont destinés à être reliés entre eux suivant le comportement que l'on souhaite définir. Petite précision sur les deux *sensors* « Always » visiblent sur l'image ci-dessous. Le premier est utilisé pour lire le script Python de manière périodique. Il est utile dans le cas où l'utilisateur controle le quadricoptère avec le joystick puisque celui ci ne transmet que continue les valeurs de ses axes (et donc aucun évènement du type « touche A pressée »).

Le deuxième est relié directement aux 4 *actuator* afin d'activir les 4 moteurs en permanence.



4.2 Comportement simple

Le but du comportement simple est de pouvoir agir via une touche du clavier sur un moteur.

Nous avons donc défini une variable pour chaque force appliquée à chaque moteur.

Ainsi nous avons 2 parties distinctes dans notre code pour gérer ce comportement.

Une qui détecte une pression sur une des 4 touches et qui incrémente la force correspondante.

La deuxième partie consiste à affecter les 4 forces au modèle. Il faut préciser qu'avec Blender™, toutes les forces s'appliquent, par défaut, au centre de gravité de l'objet considéré. Pour remédier à ce problème nous appliquons les moments qui sont nécessaires pour « déplacer » la force au point qui nous intéresse (en l'occurrence la position où se trouve le moteur).

4.3 Comportement évolué

Le comportement évolué reprend le comportement précédent mais offre la possibilité de piloter le quadricoptère.

En effet, nous avons dans ce cas 8 autres touches pour piloter le quadricoptère. Mais dans ce cas, une pression sur une touche aura pour effet de modifier plusieurs forces à la fois.

Ainsi on offre la possibilité de tangage et de lacet directement via une touche.

4.4 Joystick

Le comportement via joystick est légèrement différent de celui au clavier.

En effet, comme précisé précédemment, le joystick n'envoie pas directement des évènements mais juste un tableau de 4 valeurs correspondant aux valeurs des axes (3 axes de la manette plus l'axe de la molette des gaz).

On a donc ajouté un *sensor* « Always » qui permet de lire en continue les valeurs envoyées par le joystick.

4.4.1 Zone morte

Nous avons définit une « zone morte » , qui permet de rendre le Joystick moins sensible aux variations minimes. Cette sensibilité peut être modifiée via la constante THRESHOLD (il est à noter que la position d'un axe va de -32768 à 32767).

4.4.2 Gestion du pilotage

Le deuxième ajout est une aide au pilotage. Concrètement celle-ci consiste à repositionner le quadricoptère dans sa position initiale si l'utilisateur lâche le Joystick.

Pour cela on va récupèrer l'orientation de notre modèle par rapport à l'environnement (elle est représentée par une matrice 3x3), puis on applique des forces opposées à l'orientation actuelle afin de remettre le quadricoptère dans une position "neutre".

5 HUD

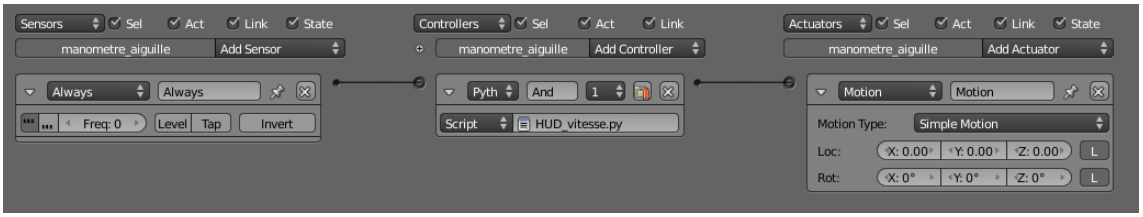
5.1 Principe global

Le HUD (head's up display) que nous avons implémenté est constitué d'un indicateur de vitesse et d'un indicateur d'altitude. Le principe est relativement simple. Pour chaque indicateur, deux plans sont positionnés parallèlement à la caméra et parentés à elle de sorte que si la caméra bouge, les indicateurs suivent. Ces plans sont texturés avec les deux textures suivantes :

Les matériaux pour ces deux plans sont un peu particuliers :

- Activation de la transparence (la texture affecte les canaux color et alpha, valeur de alpha par défaut : 0).
- Matériau en shadeless, c'est à dire pas de calcul d'ombrage ou d'éclairage. Seules les couleurs de la texture sont affichées.

Le centre de l'objet aiguille est placé au niveau de l'axe de rotation de l'aiguille, de sorte qu'on ait juste à appliquer une rotation à l'aiguille pour fixer la valeur. Les afficheurs prennent vie grâce à un script python simple que nous allons voir dans la partie suivante. Le réseau logique utilisé est le suivant :



Nous avons donc :
- Un *sensor* « Always » qui active le script à intervalles réguliers
- Un *controller* « Python » avec notre script
- Un *actuator* « Motion » qui ne sert strictement à rien, si ce n'est d'avoir un graphe valide (sinon le script ne se déclencherait pas).

5.2 Script python

```
1  import bge
2  from mathutils import Matrix
3
4  # récupère la scène et la liste des objets
5  scene = bge.logic.getCurrentScene()
6  List = scene.objects
7
8  # récupère le controleur, puis l'objet aiguille qui détient ce
       controleur
9  cont = bge.logic.getCurrentController()
10 aiguille = cont.owner
11
12 # récupère l'objet quadricoptère dans la liste des objets de la scène,
       puis sa vitesse
13 value = List['Quadri'].worldLinearVelocity.length
14
15 # affecte une matrice de rotation à l'objet aiguille, calculée d'après
       la valeur à afficher
16 aiguille.localOrientation = Matrix.Rotation(-value / 10.0, 4, 'Z').
       to_3x3()
```

6 Conclusion

Notre projet a permis de valider la solution technique de Blender pour la simulation, autant par ses capacités à fournir un affichage correct et fluide, qu'une simulation physique simple, correcte et extensible. Un future projet pour implementer dans le script de contrôle en python d'un vrai modèle et d'asservissement devrait aboutir à un comportement dans le moteur de jeu proche de la réalité.