INP Grenoble - ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble

Rapport de projet de fin d'études

Université de Sherbrooke (Québec)

Filmage automatique des déplacements du bras-robot canadien

Benjamin Auder

3e année — Intelligence artificielle, filière III

Projet du 01/07 au 23/12 2007

Université de Sherbrooke
2500 boulevard de l'Université
J1K 3A8 Québec, Canada

Responsable à l'Université de Sherbrooke :
Froduald Kabanza
Tuteur de l'école :
Augustin Lux

Sommaire

Le présent mémoire propose une méthode de planification du mouvement des caméras sur la station spatiale internationale, afin de générer automatiquement des animations 3D destinées à rendre l'apprentissage de la manipulation du bras-robot canadien plus facile. Tout ceci est intégré au logiciel *RomanTutor* que nous présenterons brièvement. Après une revue de diverses techniques de filmage automatique d'une scène à l'aide de caméras virtuelles, nous indiquerons deux types de méthodes permettant de résoudre le problème posé : filmer de manière à satisfaire certaines contraintes image par image, ou, par analogie avec le monde du cinéma, filmer selon un découpage du film en scènes. Finalement, des tests de ces deux méthodes seront menés au sein de *RomanTutor*.

Notre système de filmage doit donc au final réaliser la même tâche que ferait un cinéaste, avec la difficulté supplémentaire que l'on ne sait pas toujours quelles actions on va filmer à l'avance.

Remerciements

Merci à Froduald Kabanza d'avoir accepté de superviser cette maîtrise et de m'avoir aidé dans la rédaction de ce document. Je remercie Khaled Belghith qui m'a aidé à comprendre le fonctionnement de Roman Tutor à partir du code source, ainsi que tous ceux qui ont contribué de près ou de loin à l'avancement de mes travaux.

Table des matières

Table des figures

Introduction

Afin de faciliter l'entraînement des astronautes devant manipuler le bras-robot canadien sur la station spatiale internationale, un prototype de système tutoriel intelligent est en cours de développement depuis quelques années au sein du laboratoire PLANIART de l'Université de Sherbrooke : *RomanTutor*, pour « Robotic Manipulation Tutorial ». Ce système comprendra un générateur automatique de démonstrations afin d'illustrer à l'apprenant une partie ou la totalité d'une tâche à accomplir. Cet outil permet actuellement aux élèves de simuler des tâches qu'ils auront à effectuer dans l'espace, mais ceux-ci n'ont pas de feedback sur leurs actions, et les déplacements du bras ne sont pas toujours correctement filmés. En effet, les caméras utilisées sont fixées à la station et ne peuvent ni pivoter ni se déplacer dans l'espace 3D. La figure 1 montre une capture d'écran du logiciel.

Ce dernier point fait l'objet de cette maîtrise, et constituera l'amorce du feedback donné à l'utilisateur, pour lequel le bon filmage de la trajectoire est indispensable. Le problème de trouver une trajectoire quasi-optimale du bras-robot a été résolu précédemment par K. Belghith *et al.* (107) avec l'algorithme FADPRM, non pertinent pour ce travail.

Des travaux concernant le filmage automatique d'une scène ont déjà été menés par le passé, notamment pour les jeux vidéos mais aussi pour des observations d'expériences complexes, des générations automatiques de manuels d'utilisation en 3D etc. Ceux-ci seront quelque peu détaillés dans la seconde partie. S'ensuivront quelques aspects techniques utiles à notre travail, puis la description des méthodes et algorithmes utilisés afin de mener à bien la génération d'animations.

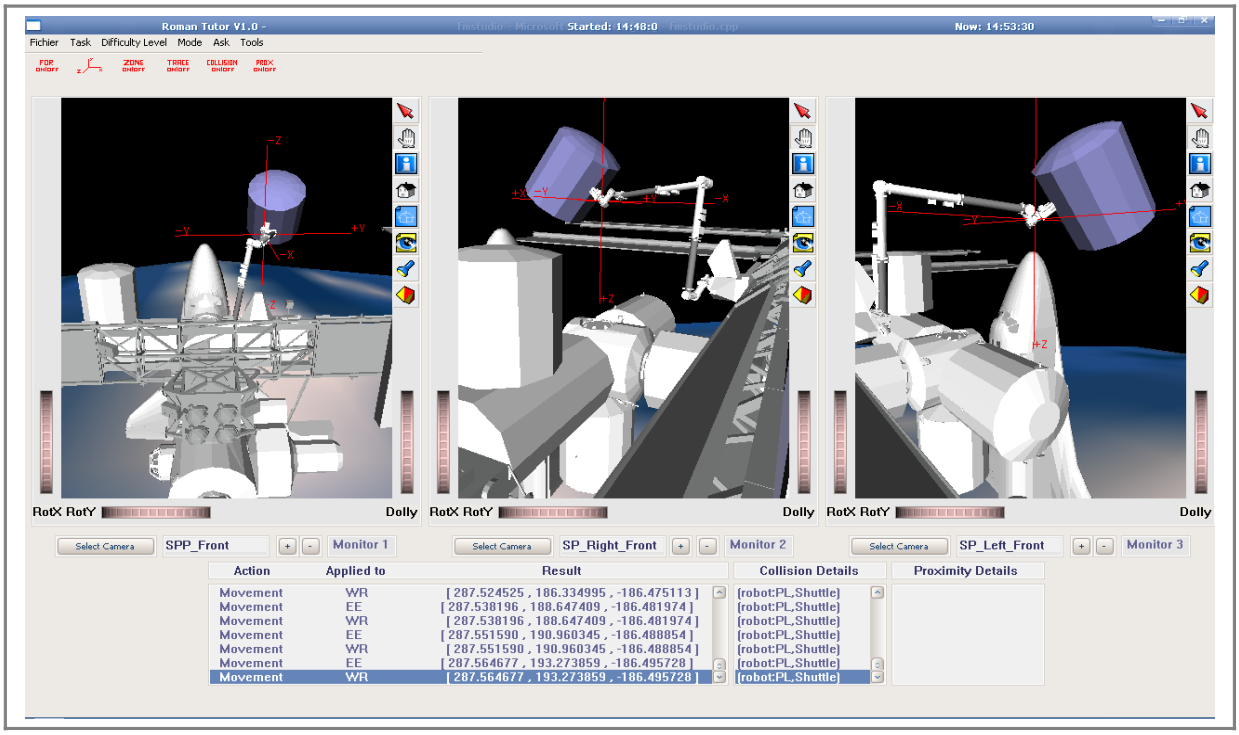


Figure 1 **Le simulateur *Roman Tutor***

Chapitre 1

Description du problème

Dans cette partie on se met en situation du problème à résoudre, présentant tous les éléments utiles.

1.1 La station spatiale

1.1.1 Constitution de l'ISS

La station spatiale internationale (*International Space Station*, en anglais, que nous noterons *ISS* en abrégé) est constituée de divers éléments présents sur l'illustration de la figure 2. Ces éléments aux géométries variables mais assez régulières (c-à-d cylindres, cubes etc) sont emboîtés sur une structure globalement cylindrique (si l'on excepte les panneaux solaires). Le bras lui-même est arrimé à la station et se déplace en translation le long d'un rail, parallèlement à la verticale du cylindre principal.

La géométrie de la station est assez complexe, d'où la difficulté à planifier une trajectoire, et a fortiori filmer cette trajectoire. Nous disposons cependant d'une carte 3D de la station (via une librairie graphique modélisant chaque objet) permettant de savoir avec une très bonne précision ce qui se trouve en chacun de ses points. Cette carte sera utile pour constituer l'image 2D que « voit » une caméra, et pour les tests de collisions.

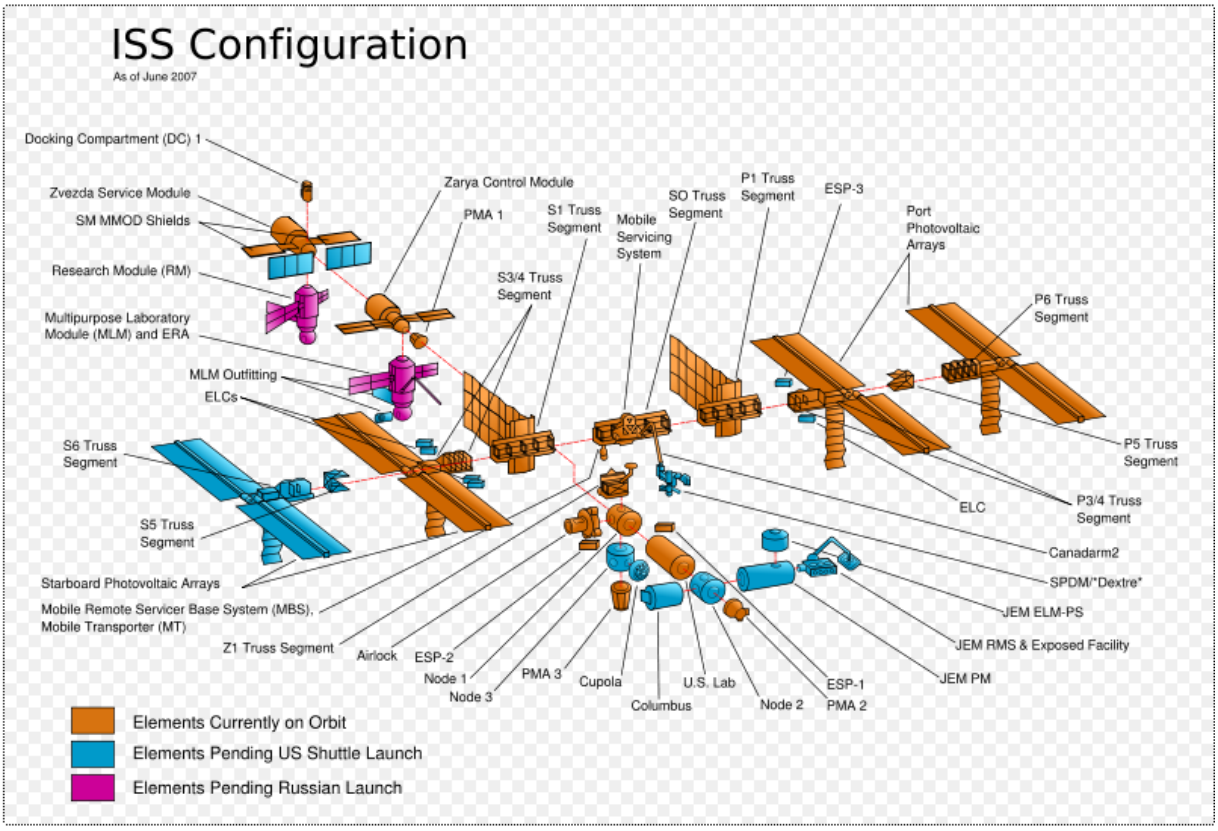


Figure 2 **Carte de la station spatiale**

1.1.2 Historique de la formation

Paragraphe adapté de *Wikipedia*. Voir 107 pour plus de détails.

La NASA amorce les premières réflexions sur un projet de station spatiale sur orbite terrestre dès le début des années 1960. Cette station est alors imaginée occupée en permanence par un équipage de dix à vingt astronautes et déjà, on prévoit de nombreuses applications : laboratoires, observatoire astronomique, ateliers de montage, dépôts de pièces et matériel, station-service, nœud et base de transport et de relais.

1.1.2.1 Naissance du projet

- En avril 1983, le Président Ronald Reagan demande que soit établi un projet de station spatiale par la NASA avant d'annoncer en 1984 la décision d'en entreprendre la construction dans un cadre international.
- Le 31 janvier 1985, l'Agence spatiale européenne (ESA) s'associe au projet, puis est suivie par le Canada le 16 avril et le Japon le 9 mai de la même année.
- Le 16 juillet 1988, le Président Ronald Reagan baptise la station du nom de Freedom.
- En 1993, l'administration Clinton invite la Russie à se joindre au projet qu'elle révise entièrement et redéfinit en suivant un concept dérivé des plans de Freedom et de la station russe Mir 2 qui devait succéder à Mir. Le projet est rebaptisé Alpha.
- Dès 1993, les Américains estiment nécessaire de profiter de la longue expérience de la Russie, maintenant alliée au projet, dans le domaine des longs séjours à bord de stations spatiales, dans le but d'éviter de reproduire certaines erreurs stratégiques ou technologiques susceptibles de provoquer de lourdes dépenses inutiles. Le 16 décembre, la NASA et la Rousskoye Kosmitcheskoye Agentsvo (RKA, l'agence spatiale russe) marquent leur accord pour 10 vols de navette vers Mir.

Le 14 octobre 1997, le Brésil rejoint l'équipe, et à Washington en 1998, ce sont 16 nations qui participent au projet : les États-Unis, 11 États de l'Union européenne, le Canada, le Japon, le Brésil, la Russie. La construction peut débuter. Le nom Alpha, qui ne plaît pas aux Russes car ils estiment que ce sont eux qui ont créé la véritable première station orbitale, est simplement dénommée Station spatiale internationale, ou International Space Station (ISS).

1.1.2.2 Mise en oeuvre

Le 20 novembre 1998, le premier élément de la Station Spatiale Internationale, le module Zarya, est mis en orbite par les Russes au moyen d'une fusée Proton lancée depuis Baïkonour. En octobre 2005, suite à l'échec du retour en mission des navettes spatiales américaines, la NASA a annoncé que seuls 18 vols auraient lieu avant la fin du programme. Ces 18 vols comprennent notamment l'envoi du module européen Columbus

et du Japanese experiment module (JEM). Deux importants modules : la plate-forme de puissance solaire russe et la centrifugeuse japonaise ne seront pas envoyées.
Notons que les coûts impliqués sont de l'ordre de plusieurs dizaines de milliards de dollars à chaque opération de grande envergure.

En conclusion, l'ISS représente un terrain d'expérimentation unique pour les sciences de la vie et de la matière, la physique fondamentale, mais aussi une plateforme d'observation de la Terre et de l'Univers. Depuis novembre 2000, deux ou trois spationautes occupent en permanence la station.

1.2 Le bras-robot CANADARM 2

1.2.1 Description du bras

Il s'agit d'une grande structure robotisée reproduisant grossièrement un bras humain (figure 3, la figure 4 est quant à elle une photo du bras sur la station).

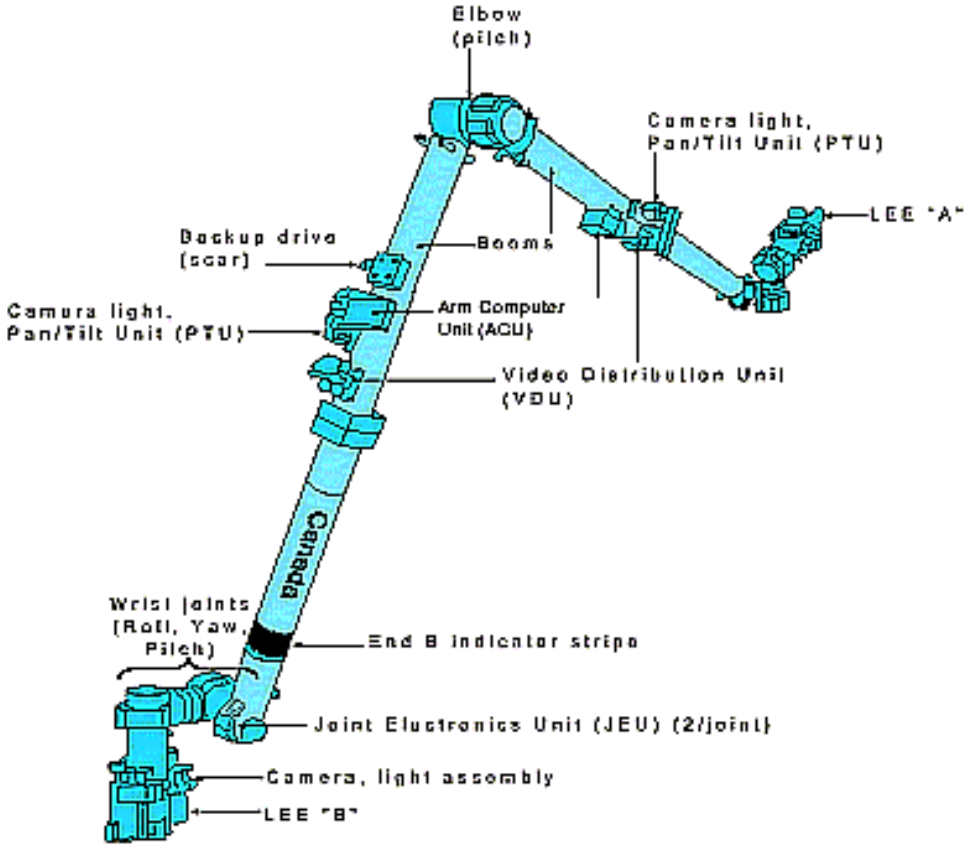


Figure 3 **Description du bras-robot canadien**

Le premier joint de l'épaule permet des mouvements de rotation entraînant le reste du bras (yaw and pitch), et un mouvement sur lui-même (roll). Le joint suivant, au coude et donc à la moitié de la longueur du bras ne permet que des mouvements rectilignes dans le même plan (rotation « pitch », comme chez un être humain). Enfin, le poignet est capable de trois sortes de déplacements qui ne nous intéresseront pas vraiment dans cette étude, sauf dans le cas où seul celui-ci bougera : on zoome alors sur le poignet. Notons que la main peut se désolidariser du reste du bras. Ajoutons à cela la translation le long du rail parallèle à la verticale de la station, et nous obtenons au final huit degrés de liberté pour le bras-robot.

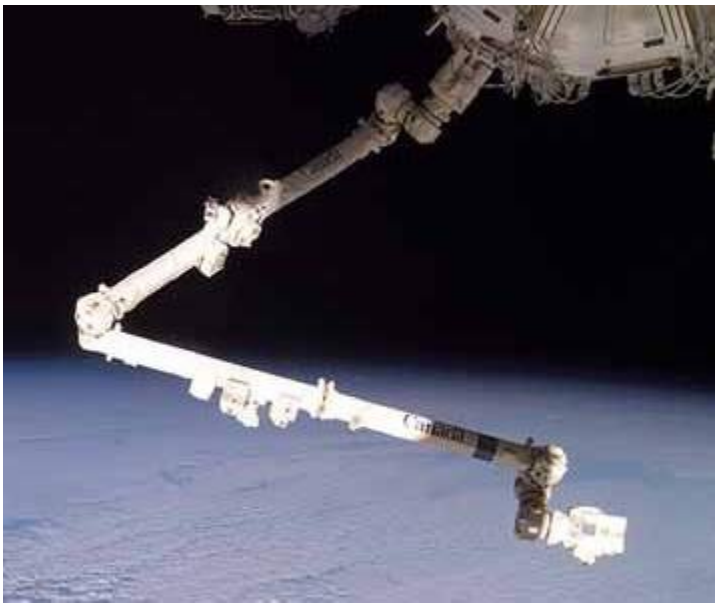


Figure 4 **Le bras-robot arrimé à la station**

Voir aussi 107 pour d'autres renseignements.

1.2.2 Fonction sur la station

Le bras-robot canadien (BRC) est une composante particulièrement importante de la station spatiale internationale (SSI). En effet, le BRC permet de déplacer des charges sur la station, d'effectuer des réparations de la station spatiale et une caméra placée à son extrémité permet aussi d'inspecter différentes

parties de la station. La figure 5 montre la station spatiale internationale et le BRC. La manipulation du BRC est une tâche très complexe : non seulement l'opérateur du BRC doit maîtriser parfaitement la géométrie du robot et ses différents modes de manipulation, mais il doit aussi avoir une grande connaissance de la station et ses différents composants. Il doit aussi accomplir toutes les manipulations avec la plus grande précision possible afin d'éviter toute collision avec les différents éléments de la station, souvent fort coûteux.

L'opérateur effectue ces différentes tâches à distance, à partir d'un ordinateur se trouvant à l'intérieur de la station. Il ne peut voir l'extérieur qu'à l'aide des trois moniteurs de cet ordinateur, chacun relié à l'une des quatorze caméras placées à différents endroits de la station spatiale. Chaque caméra n'offre qu'une vision partielle de la station, l'opérateur doit donc choisir correctement les caméras à utiliser, suivant la manipulation à effectuer.

La figure 6 résume la forme globale de la station ainsi qu'une esquisse des repérages utilisés. En effet la station est globalement cylindrique, et tous les paramètres du bras sont des angles variant entre $-\pi$ et π (à l'exception bien sûr de la translation le long du rail).
L'ensemble de ces paramètres est un vecteur décrivant la configuration du bras-robot.

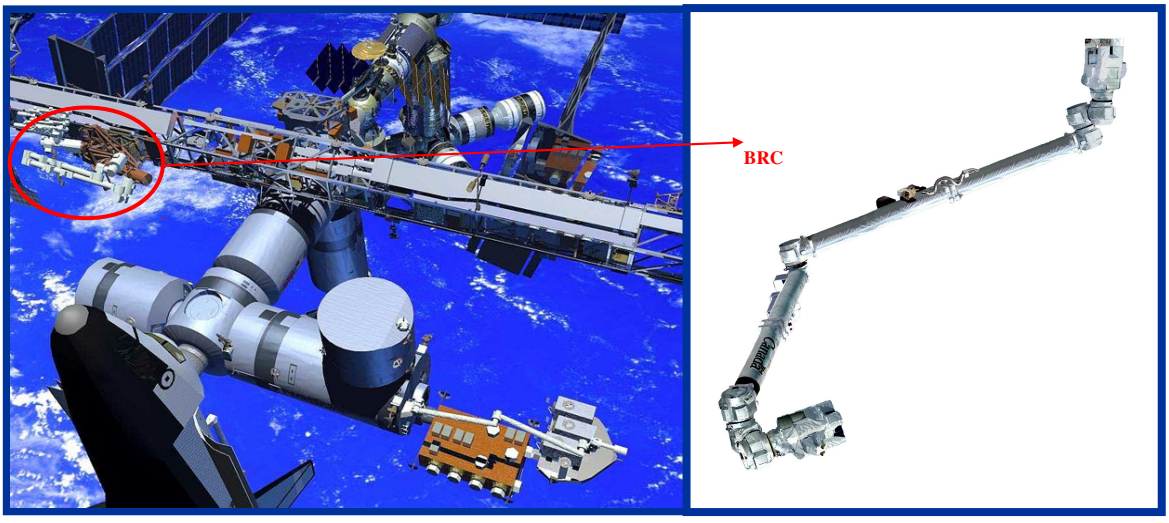


Figure 5 **La station spatiale internationale et le BRC**

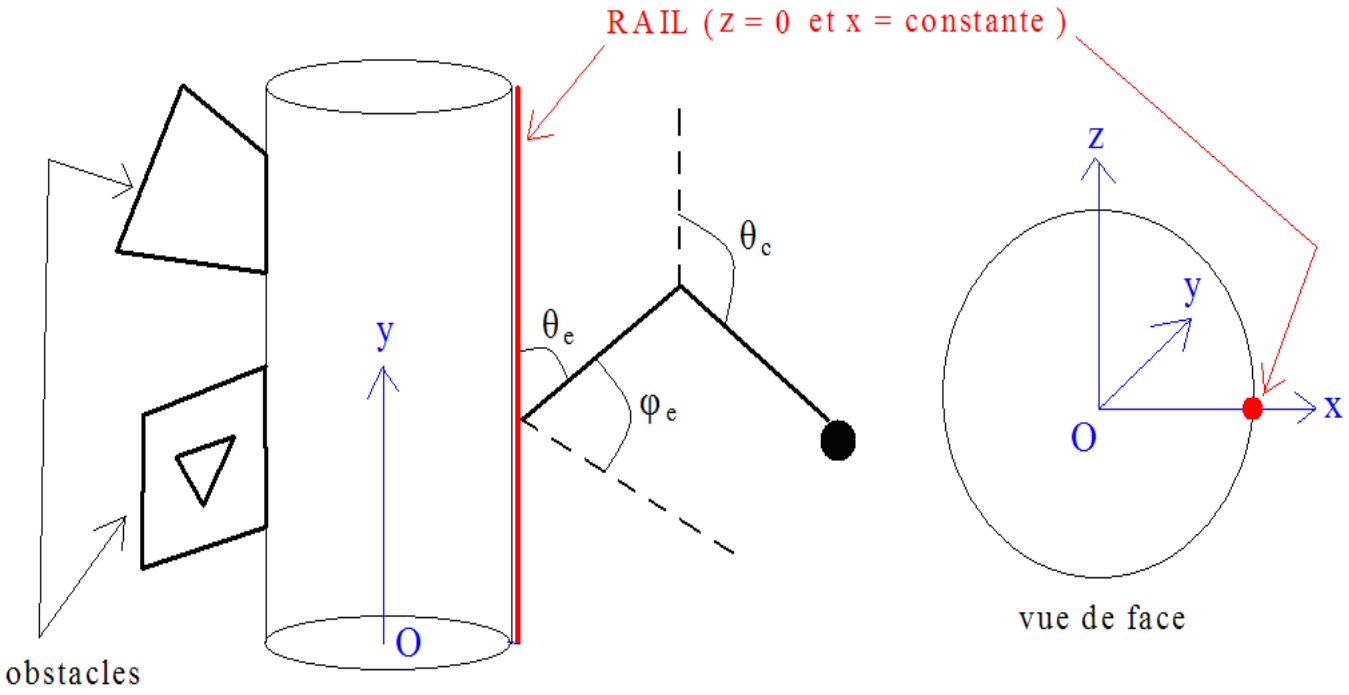


Figure 6 Schéma du bras et du repère global

1.3 Les caméras utilisées

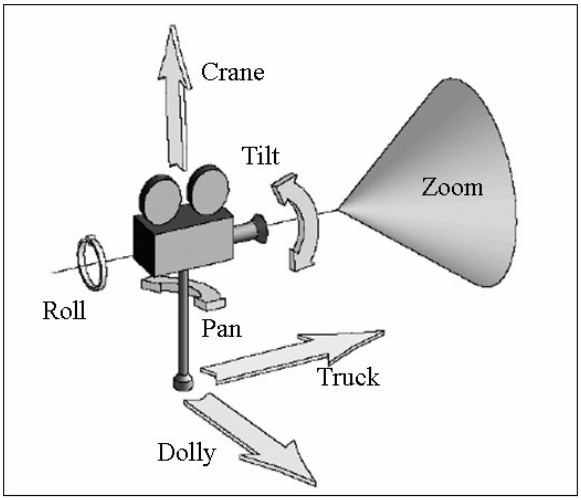
Lorsque l'astronaute utilise le bras-robot sur la vraie station spatiale (non simulée), il n'a donc à sa disposition que quatorze caméras fixées en différents endroits de la station : celles-ci sont seulement capables de pivoter sur elles-mêmes et de zoomer – et il ne peut visualiser que trois caméras à la fois dans son interface. En revanche, au sein de *RomanTutor* on peut présenter l'animation de la trajectoire du bras à l'aide d'un jeu de caméras virtuelles, libres, non contraintes par les spécifications de la station.

D'où deux modes de simulation évidents qui doivent être implémentés dans *RomanTutor* :

- Dans une première phase on peut utiliser des caméras virtuelles pour faciliter la manipulation du bras.
- Ensuite, l'astronaute doit s'habituer à ne voir que les champs visuels des caméras effectivement arrimées à différents endroits de la station.

1.3.1.1 Description d'une caméra

Chaque caméra possède sept degrés de liberté pour la position, l'orientation et le champ de vision, comme le montre la figure 7.



$\{X, Y, Z, R_x, R_y, R_z, \boldsymbol{fov}\} \in \Re^7$

Figure 7 **Les sept degrés de liberté d'une caméra**

Pour ce qui est des spécificités des caméras, on doit préciser l'étendue de zoom possible ainsi que les angles de vue. Toutes les caméras de la station sont identiques et possèdent les caractéristiques suivantes :

- **Zoom** : considéré comme infini dans le cas de caméras virtuelles ; information inconnue sinon.
- **Angle** : on prend π/2 comme angle au sommet du cône de vision ; choix arbitraire car information inconnue (pas d'influence sur les résultats).

1.3.1.2 Détermination du repère local

Lorsque la caméra vise un point (x,y,z) dans l'espace, nous nous intéressons naturellement aux coordonnées des objets de l'environnement du point de vue de la caméra. Pour cela, il faut faire le choix d'un repère centré sur la caméra. On en choisit un orthonormé direct comme suit :

Vecteur X ayant pour coefficients $\begin{pmatrix} \cos\phi\cos\theta \\ \sin\phi\cos\theta \\ -\sin\theta \end{pmatrix}$ dans le repère orthonormé direct global présenté figure 5

Vecteur Y ayant pour coefficients $\begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$ dans ce même repère ,

Les angles φ et θ étant ceux des coordonnées sphériques du point cible (x,y,z) dans le repère global translaté pour avoir la caméra comme origine.

Le **vecteur Z** a donc pour coefficients $\begin{pmatrix} \cos\phi\sin\theta \\ \sin\phi\sin\theta \\ \cos\theta \end{pmatrix}$ dans le repère global.

On vérifie ainsi que la base choisie (X,Y,Z) est orthonormée directe. Cela facilite le traitement ultérieur des rotations.

Finalement, quelques mots sur la structure d'un film, qui nous sera utile tout au long du travail.

1.4 Décomposition d'un film

Un film est d'abord découpé en scènes. Chaque scène présente une situation ou une action particulière. Chaque scène est ensuite découpée en prises (ou *shots* en anglais). Un *shot* est une séquence ininterrompue d'images filmées par une seule caméra, et dure en général une à dix secondes. Ces *shots* sont éventuellement liés les uns aux autres par des transitions. La figure 8 présente ces abstractions sous forme de hiérarchie.

Dans notre cas, le découpage est à calculer à chaque déplacement du bras-robot, une scène représentant une action du bras comme « contourner l'obstacle x » ou « soulever la charge y ». Bien sûr ce

découpage en scènes n'est pas immédiat et nécessite une analyse mathématique de la trajectoire, donnée sous la forme d'une suite de configurations du bras.

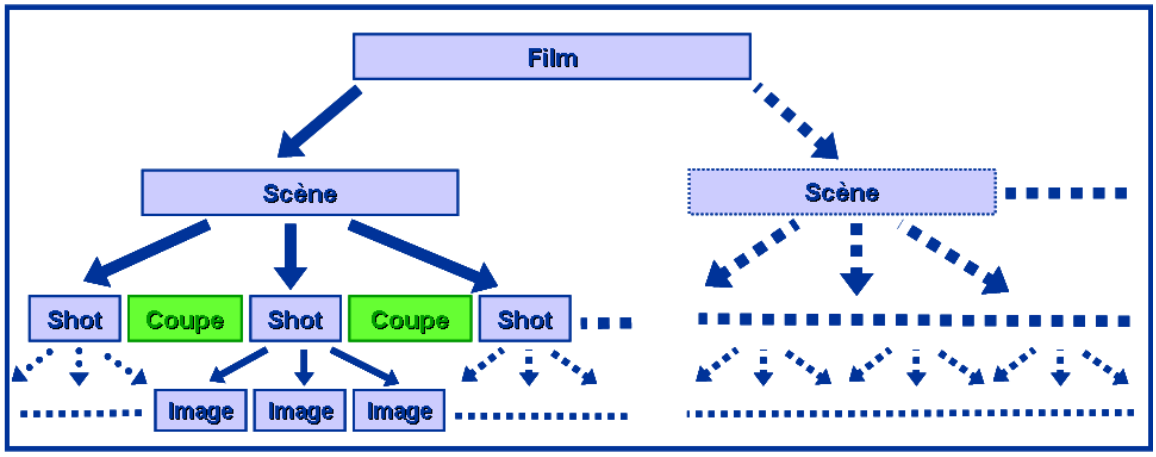


Figure 8 **Hiérarchie d'abstractions à l'intérieur d'un film**

Ensuite, chaque séquence ainsi déterminée doit être subdivisée en shots, et il faut donc déterminer les points de coupure pour changer de caméra. Une fois ces découpages effectués (faisant l'objet d'une première partie), on est à même de déterminer les « meilleures » façons de filmer chaque morceau, suivant leurs caractéristiques (c'est la seconde partie du travail). Tout cela est détaillé par la suite.

1.5 Objectifs du travail

Maintenant que nous avons clairement énoncé le cadre de cette maîtrise ainsi que la problématique, nous pouvons énoncer nos objectifs globaux précisément :

- Développer un système permettant de filmer le bras-robot au cours de son mouvement, à l'aide d'un jeu de caméras virtuelles dans un premier temps, puis fixées à la station.
- Intégrer ce système de filmage, supposé suffisamment informationnel pour l'apprenant, au logiciel *RomanTutor* dont nous avons déjà parlé.

La figure 9 présente les niveaux où nous allons intervenir dans le code de *RomanTutor*, qui est actuellement divisé en trois grandes parties

- Un module planifiant la trajectoire du bras-robot en fonction d'un certain nombre de points dans l'espace 8D des configurations par lesquels le bras doit passer.
- Un module planifiant le parcours des caméras compte-tenu de la suite de configurations du bras-robot donnée par le module précédent.
- Un module permettant l'affichage de la scène et des mouvements, à l'aide d'une librairie graphique.

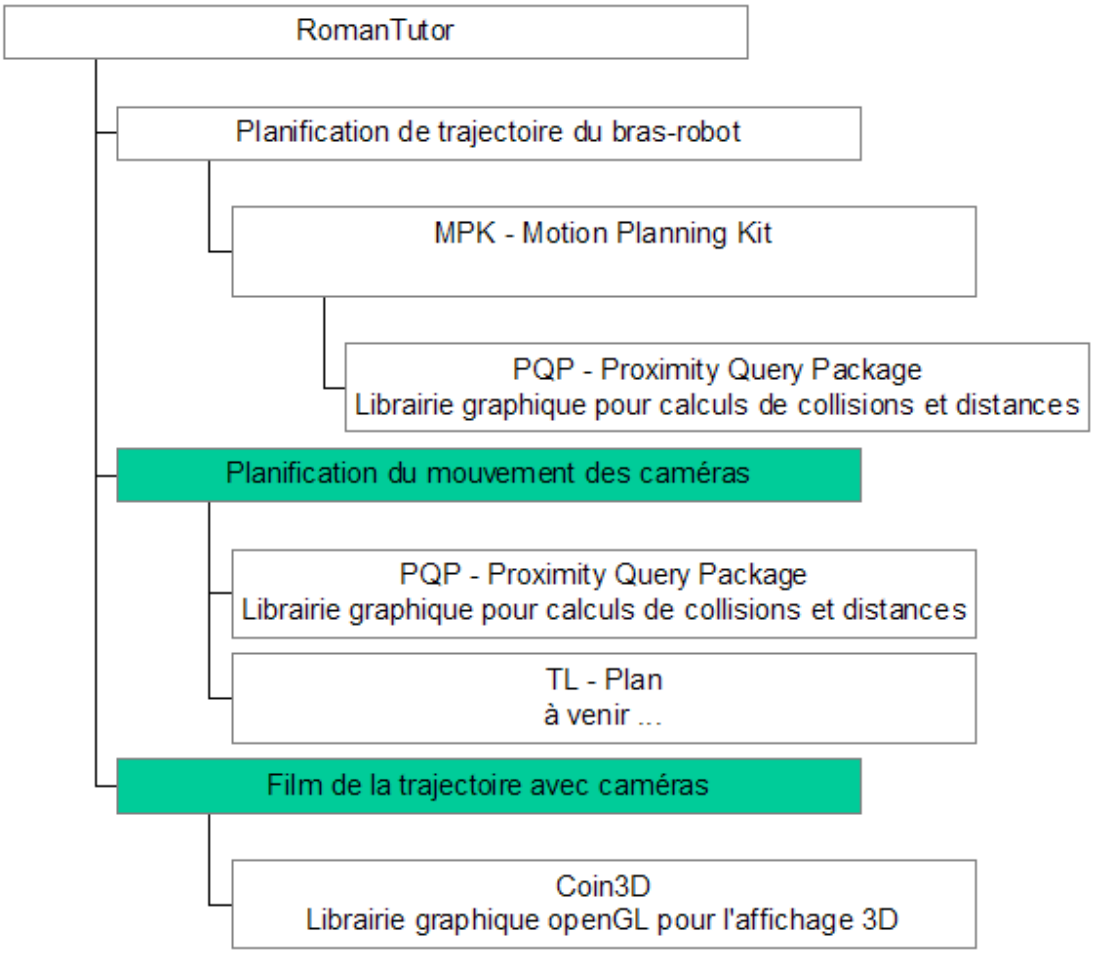


Figure 9 **Architecture de *RomanTutor***

Chapitre 2

Filmage automatique d'une scène : état de l'art

Au début de l'histoire du cinéma, une scène était filmée par une caméra fixe, avec un seul point de vue adopté vis-à-vis des acteurs : exactement comme si l'on filmait une pièce de théâtre. Depuis plusieurs années beaucoup de jeux vidéos proposent une vue à la première personne, mettant le joueur dans la peau de son personnage. C'est mieux, mais encore insuffisant dans le cas de jeux complexes, et a fortiori des animations devant être générées dans le cas de *RomanTutor*. Parmi les nouvelles approches de filmage automatique, nous distinguons deux groupes parmi lesquels nous avons sélectionné quelques algorithmes.

2.1 Approches purement géométriques

Ces méthodes proposent de suivre le mouvement de personnages ou choses animés en se basant uniquement sur la géométrie de la scène.

2.1.1 Résolution de contraintes algébriques

La méthode développée par F. Benhamou et al. 107 est à la base prévue pour résoudre des systèmes de contraintes non linéaires données sous la forme d'inéquations. Ils mettent donc en inéquations les contraintes devant être respectées par une caméra (par exemple « voir la sphère x des coordonnées (a,b) à (c,d) pendant 20 secondes à l'écran, sans obstruer la vision de la caméra »), puis déduisent le mouvement de la caméra en résolvant le système obtenu, la plupart du temps non linéaire.

2.1.1.1 Méthode utilisée

Disposant d'un système non linéaire de *m* équations à *n* inconnues, on commence par déterminer (si cela n'est pas donné) les domaines de variation de chaque variable : on obtient une union d'intervalles réels pour chaque variable considérée, notées $x_1,\ldots,x_n$. Les intervalles correspondant forment une grande « boîte » dans l'espace à *n* dimensions : c'est une première approximation par excès de l'ensemble des solutions. L'idée est ensuite d'utiliser divers algorithmes de réduction de cette boîte par partitionnements successifs (voir l'article pour les détails, pages 10 et 11). Ensuite, on recherche au contraire une approximation de l'ensemble des solutions par plusieurs « boîtes » *n*-dimensionnelles, par défaut : tous les points contenus dans ces boîtes sont des solutions du système (pour les détails voir pages 12 à 19 dans l'article). On obtient alors finalement un bon encadrement des solutions recherchées : il ne reste plus à l'utilisateur qu'à choisir celle qu'il préfère (en effet il y a souvent plusieurs, voire une infinité de solutions possibles, que le système ne peut pas juger).

2.1.1.2 Application au cas du filmage

L'objectif principal est de construire un outil de haut niveau permettant à l'artiste de spécifier les mouvements désirés de la caméra pour un shot, en utilisant des primitives cinématographiques. La description qui en résulte est ensuite traduite en un système de contraintes portant sur les paramètres des caméras, que l'on résout par les techniques évoquées au paragraphe précédent.

Remarquons qu'il suffit de planifier les déplacements de la caméra sur un shot, car un film est au final composé uniquement d'une multitudes de shots. On doit donc simplement planifier des mouvements élémentaires de la caméra. Contrairement aux approches optimisant la position de la caméra image par image, les auteurs paramétrisent ici le mouvement de la caméra, en mettant en équation (polynômes du troisième degré) ses déplacements élémentaires. La caméra est définie mathématiquement par sept réels : trois pour la position dans l'espace, trois pour l'orientation de la caméra et un pour la valeur de zoom, comme indiqué figure 10.

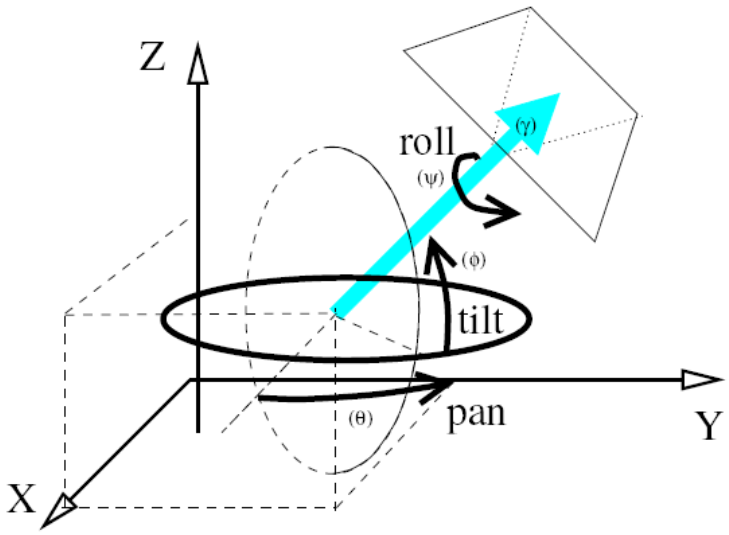


Figure 10 **Les degrés de liberté d'une caméra**

Tout est alors prêt pour mettre en inéquations les contraintes imposées par l'utilisateur quant à certaines propriétés que doit vérifier le film.

2.1.2 Roadmap probabiliste

D. Nieuwenhuisen et M. H. Overmars ont développé quant à eux un système de planification des mouvements de caméra à l'aide d'un roadmap probabiliste (107). L'idée est de construire un graphe de points 3D autour de la station spatiale (appelé « roadmap », dit probabiliste la plupart du temps car les méthodes exactes sont beaucoup trop lentes), considérant la caméra comme un point (en fait comme une sphère, mais seule la position de son objectif importe donc on se ramène au cas d'un point sauf lors des tests de collision). Une fois le graphe des lieux à filmer obtenu et assez dense, on est à même de trouver un chemin dans ce graphe de la position initiale souhaitée à la position finale, via un algorithme comme celui de Dijkstra ou A*. On obtient donc le parcours basique de la caméra.

Note : concernant le coût des arêtes pendant le passage dans l'algorithme de recherche de chemin, on introduit une pénalité d'autant plus élevée que l'angle entre les deux arêtes est aigu. En effet la caméra met

alors plus de temps à tourner, pour ne pas déstabiliser le spectateur. On préfère les virages plus doux, par exemple sur la figure 11 le chemin **a** est préférable.

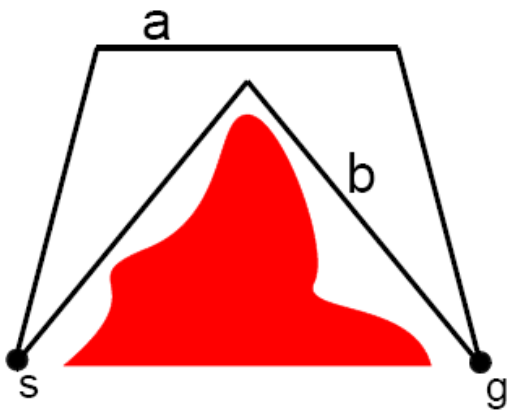


Figure 11 **Chemin le plus court versus chemin le moins escarpé**

Ensuite, différentes phases sont à considérer :

2.1.2.1 Amélioration du chemin

Le chemin obtenu est pour l'instant une succession de segments rectilignes, et donc inutilisable tel quel : on ne veut pas d'un film saccadé. Ce chemin va être rendu continûment dérivable en ajoutant simplement des arcs de cercles reliant deux segments adjacents, comme illustré sur la figure 12. Un raccordement de ce type est toujours possible car en prenant un modèle sphérique de caméra on est assuré que les segments soient toujours à distance strictement positive des obstacles, distance supérieure à un réel $\delta > 0$. On procède donc par dichotomie en commençant par un arc de cercle joignant les milieux des segments, puis les quarts etc, comme illustré figure 13.

Cette étape s'achève en ajustant la vitesse de la caméra le long du parcours, tenant compte des contraintes sur l'accélération et les vitesses maximales, puis en raccourcissant le chemin en testant si on peut relier deux points du trajet par un segment, puis arc de cercle sans collision (smoothing).

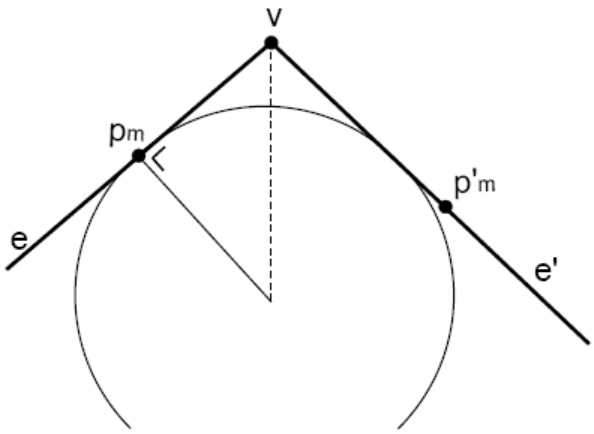


Figure 12 **Raccord entre deux segments par un arc de cercle**

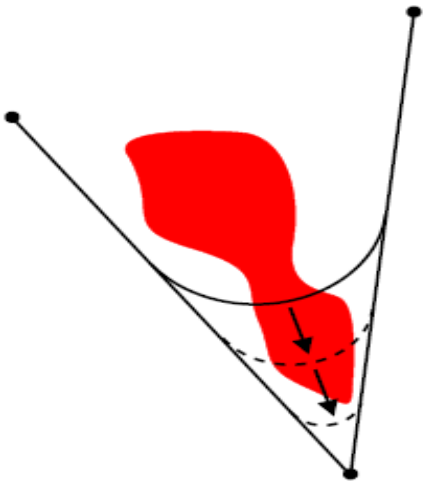


Figure 13 **Déplacements successifs d'un arc de cercle**

2.1.2.2 Orientation de la caméra

Jusqu'ici nous n'avons pas parlé de la direction dans laquelle la caméra devait filmer, ainsi que le zoom à appliquer : ce sont les quatre derniers paramètres à régler (trois seulement si l'on s'interdit le mouvement de rotation « roll », ou qu'on en tient pas compte). C'est ce qui va être déterminé maintenant.

Tout d'abord nous savons par l'expérience que la caméra doit regarder non pas dans le sens du mouvement actuel, mais anticiper d'un temps t_d la prochaine position de la caméra, et regarder dans cette direction. Une bonne valeur de t_d est une seconde. On montre ainsi que la direction de la caméra est une fonction du temps continûment dérivable (propriété non vérifiée si l'on regarde dans le sens du mouvement) : cela permet un affichage plus fluide et donc plus agréable. Voir la figure 14 pour une illustration. Le mouvement « roll » est évité le plus possible, semblant utile que lors de loopings par exemple.

Enfin, on règle la direction de départ de la caméra en l'orientant comme elle doit l'être au point initial trouvé lors du roadmap (en général différent mais proche du vrai point de départ, à moins que l'on ait ajouté les points initiaux et finaux au roadmap), avant que celle-ci effectue une translation pour se rendre au premier point calculé pour l'itinéraire de la caméra.

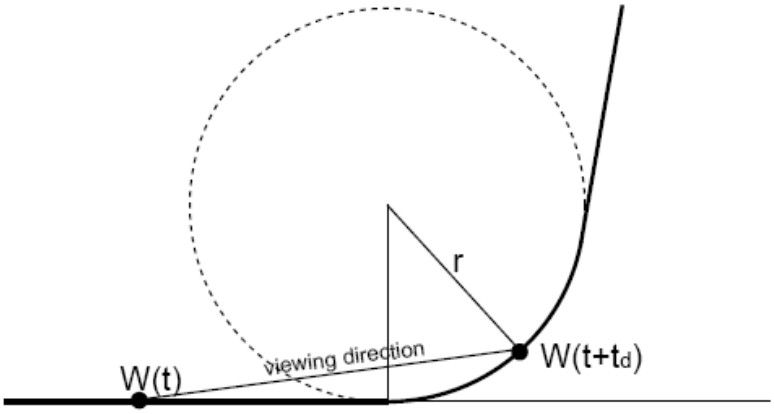


Figure 14 **Direction de la caméra**

Beaucoup de jeux vidéo utilisent un procédé rigide qui détermine à l'avance comment la caméra devra filmer le niveau en fonction de ses caractéristiques, dépendamment de l'emplacement du personnage du joueur. Tomb Raider, ainsi que les jeux de Lucas Art en sont des exemples. Cette façon de faire n'est pas souhaitable car on se retrouve trop souvent avec une vue qui n'est pas la meilleure pour l'utilisateur : il faut que les mouvements de la caméra s'accordent avec l'action actuelle du joueur, et donc avec ses intentions apparentes, tout en fournissant un bon effet visuel.

2.2 Les approches par satisfaction de contraintes cinématographiques

Ce sont les méthodes les plus intéressantes, utilisant des règles établies au fil du temps par l'expérience lors de la réalisation de films ; parmi celles-ci, on peut citer :

- La caméra ne devrait pas passer trop proche d'un obstacle : cela provoquerait une obstruction de l'écran tellement importante que le spectateur aurait l'illusion d'une collision. Donc on doit ménager un certain « espace vital » pour la caméra.
- L'horizon de la caméra doit rester le plus "droit" possible, c'est-à-dire que les objets ne doivent pas être observés de travers. Cela signifie que l'on doit le moins possible « roller » la caméra.
- Quand la caméra effectue un virage, sa vitesse doit diminuer, car sinon les objets bougeront beaucoup trop vite à l'écran.
- Des indices devraient permettre au spectateur de deviner où la caméra se dirige. En particulier, celui-ci devrait pouvoir prévoir une rotation. Ceci peut être réalisé en orientant non pas la caméra dans le sens du mouvement, mais vers la position où l'objet se retrouvera quelques instants plus tard.
- Ne pas traverser la ligne d'action : lorsque l'on a démarré une séquence avec l'impression d'un mouvement dans un sens, il ne faut pas passer brusquement de l'autre côté de la ligne pour voir alors le personnage aller dans l'autre sens apparent ; un shot neutre entre les deux prises est nécessaire.
- Le mouvement coupé : une scène illustrant un mouvement devrait typiquement être découpée en deux shots, chacun voyant l'objet ou être vivant parcourir la moitié de l'écran.

Voyons quelques systèmes développés en vue de satisfaire les contraintes précédentes, avec une certaine gradation.

2.2.1 Le « Camera Planning System » (CPS) de Christianson et al. 107

Définition : Un *idiome* est une succession de shots, arrangés de façon à montrer un passage particulier d'une scène. La rencontre de deux acteurs, par exemple. C'est donc une structure intermédiaire entre la scène et le shot.

Les auteurs ont utilisé un langage de description des scènes qui leur permet de formaliser les « façons de filmer » diverses scènes. Ils reprennent la structure arborescente du film décrite sur la figure 8, en associant à chaque scène différents idiomes possibles. Le Camera Planning System fonctionne en parcourant l'arbre du film, un peu comme on parcourrait un arbre dans le cas d'un algorithme alpha-beta : Il va jusque dans les feuilles pour trouver les meilleurs shots, puis remonte pour en déduire les cadres et idiomes appropriés pour chaque scène.

Ils utilisent beaucoup la « ligne d'intérêt » d'une scène, qui est dans la direction du regard ou du mouvement d'un ou plusieurs acteurs. Dans le cas du bras-robot on ne peut utiliser cette ligne, d'autres méthodes doivent donc être utilisées.

Toutefois, le langage utilisé pour décrire les mouvements de caméras est intéressant et mérite une description plus approfondie :

2.2.1.1 DCCL

Il y a quatre primitives de base dans le langage DCCL, qui sont les *fragments, views, placements et movement endpoints* (on garde la terminologie anglaise afin de ne pas perdre certaines nuances). Ces primitives sont combinées pour construire des structures de plus haut niveau comme les shots ou les idiomes.

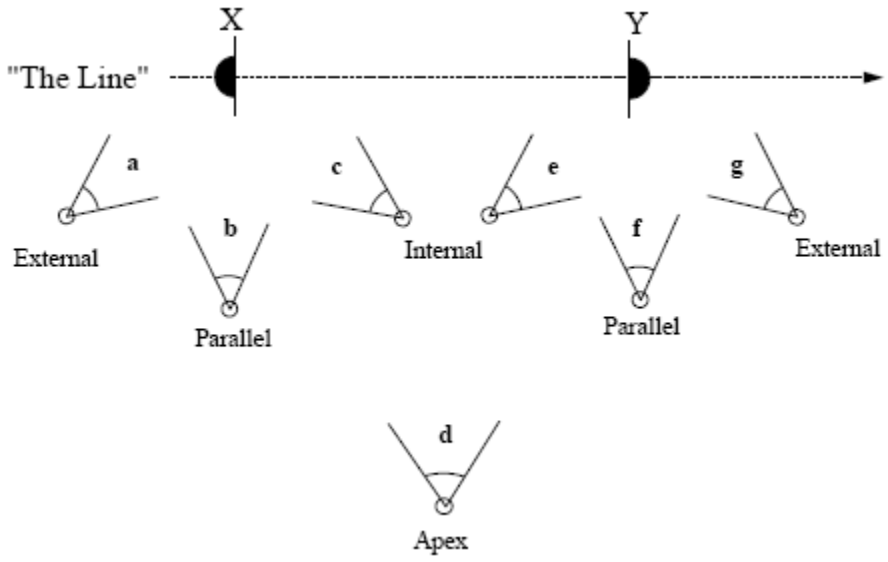


Figure 15 **Les différents points de vue d'une caméra**

Un *fragment* spécifie un intervalle de temps durant lequel la caméra effectue un mouvement élémentaire précis. Bien sûr pour qu'on puisse utiliser cette information, il faut aussi indiquer comment la caméra doit être placée par rapport à l'acteur : *internal, external, parallel* ou *apex*, voir la figure 15 ci-dessus, ainsi que le type de zoom :

- **extreme closeup :** zoom sur le visage
- **closeup :** image du buste
- **medium :** image du tronc coupé aux hanches
- **full :** image de la personne entière prenant tout l'écran
- **long :** la personne entière vue de plus loin.

Finalement, il faut donner aussi les temps de début et de fin de chaque filmage élémentaire, et on se retrouve avec un fragment de film entièrement spécifié.
La figure 16 indique les cinq types de fragments retenus par les auteurs.

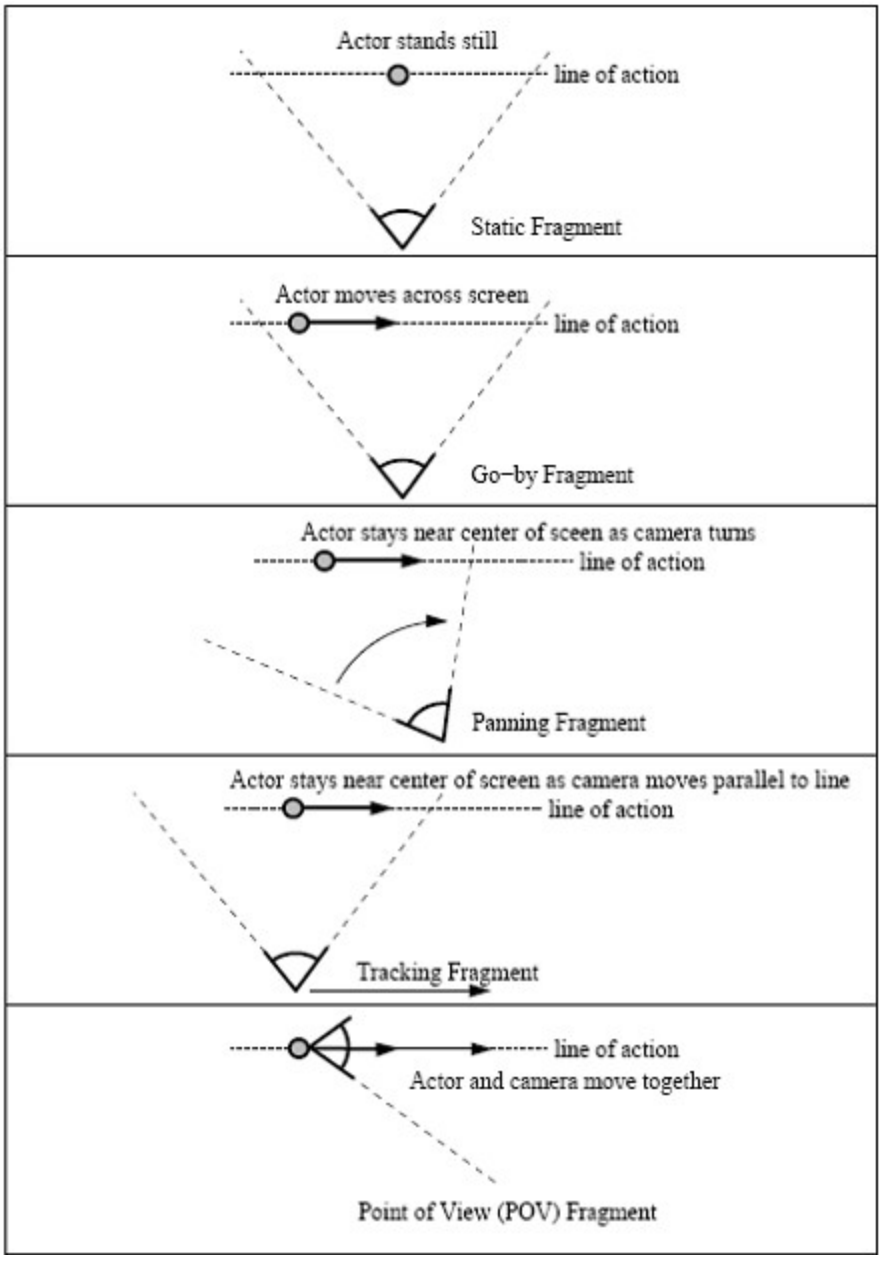


Figure 16 **Les cinq sortes de fragments**

2.2.1.2 Bilan de la méthode

La méthode utilisée se résume finalement ainsi, étant donné une trace d'animation (c'est-à-dire la suite des configurations des objets mobiles de la scène ; dans notre cas ce serait les configurations successives du bras-robot) :

1) Découpage en actions élémentaires (cette partie dépend de l'application).
2) Rechercher un idiome correspondant à chaque action élémentaire dans la base de données, tout en précisant les éléments nécessaires à la traduction en langage DCCL.
3) Déterminer les cadres de transition sur chaque point de coupure obtenu lors du découpage, si nécessaire (violation d'une contrainte cinématographique par exemple).
4) Finalement, décoder le script DCCL par un compilateur et exécuter le film.

Remarque 1 : Le point 2) implique que si l'on envisage d' utiliser cette méthode, il faut lister toutes les actions élémentaires du bras. Cela ne semble pas réalisable, compte tenu de la complexité des mouvements possibles.

Remarque 2 : La méthode présentée nécessite d'avoir dès le départ toute la trace d'animation, et n'est donc pas utilisable telle quelle dans le cas d'un filmage en direct du bras-robot.

2.2.2 L'algorithme UCAM par W. H. Bares et J. C. Lester 107

Ce système se base sur une représentation des préférences visuelles d'un spectateur virtuel créé afin de mieux cerner les meilleures vues pour les caméras ; ensuite, UCAM choisit les orientations, positions et mouvements des caméras en fonction de ce qu'il a déterminé que le spectateur aimerait voir. L'ajout de ce spectateur virtuel, qui en pratique est simplement l'utilisateur final qui dicte ses préférences visuelles, constitue l'amélioration majeure de cette méthode par rapport à la précédente.

Le choix effectué par les auteurs pour transcrire les préférences visuelles de l'utilisateur est assez simple : les préférences du type « plus ou moins la caractéristique x » se transforment en probabilités plus ou moins élevées de filmer d'une certaine manière conforme à la caractéristique souhaitée. Par exemple si

l'utilisateur souhaite voir un film sans coupures, on raccordera systématiquement deux shots par un mouvement continu de la caméra etc. Voir la figure 17 pour plus de précisions.

Ainsi, à chaque nouvelle image devant être affichée, on calcule la nouvelle position de la caméra en fonction des données et des préférences de l'utilisateur. Peu de détails supplémentaires sont donnés sur cet algorithme de visualisation d'une scène, qui ne sera de toutes façons pas utile car dans notre cas on cherche avant tout un point de vue informationnel ; il est inutile de proposer à l'élève le choix d'un style de filmage. Cela pourrait toutefois constituer une extension future intéressante.

```
loop
        3DEnvironment_ update-environment
        if user modifies visualization preferences then
                CinematicUserModel ← construct-UM (VisualizationPrefs)
        if NumFrames ← MinimumShotDuration then
                (* no change to CamShot but move camera to track object *)
                CamPosition ← select-new-position (3DEnvironment, null-transition)
        else {
                CamShot ← select-new-shot (CinematicUserModel, 3DEnvironment)
                NumFrames ← 0
                if AngleToNewShot ← MinimumCutAngle then
                        CamTransition ← pan to new position
                else
                        CamTransition ← cut to new position
        CamPosition ← select-new-position (3DEnvironment, CamTransition)}
        NumFrames ← NumFrames + 1
        NewFrame ← render (3DEnvironment, CamShot, CamPosition)
until user exits visualization
```

Algorithme 1 **UCAM de W. H. Bares et J. C. Lester**

Visualization Preference	*Value*	*Cinematographic User Model Camera Directives*
Viewpoint Style	Informational	Medium and long shots more probable Medium and high elevation shots more probable
	Dramatic	Close-up and near shots more probable Low and medium elevation shots more probable
Camera Pacing	Slow	Longer shot duration ($\geq$ 35 frames) Pan in increments of 1 °
	Fast	Shorter shot duration ($\leq$ 15 frames) Pan in increments of 4 °
Transition Style	Gradual	Always pan and track between different shots
	Jump	Cut if angular distance between shots $>$ 60 °

Figure 17 **Sémantique des préférences visuelles**

2.2.3 Le système de filmage de N. Halper et al. 107

Ce système de filmage automatique destiné aux jeux vidéos s'intègre en fait dans une hiérarchie de modules allant du niveau du script du jeu jusqu'au rendu de l'image, comme indiqué sur la figure 18. Le travail effectué par les auteurs concerne le module intermédiaire entre la couche de haut niveau flexible et facilement modifiable par les développeurs (toute la partie de conception du jeu) et celle de bas niveau contrainte à interpréter les instructions reçues toujours de la même façon afin d'effectuer le rendu visuel (éclairage et image finale).

2.2.3.1 Caractéristiques nécessaires au filmage automatique

Concernant ce module intermédiaire de contrôle de la caméra, nous retenons trois concepts de base :

- **Flexibilité** : Le codage doit être paramétrique et souple dans les spécifications de contraintes, afin de s'adapter à tous les événements pouvant venir du module d'action.
- **Information** : Plus la caméra connaît de choses sur le cours actuel des événements, mieux c'est. On a besoin des informations venant des acteurs, des motivations du joueur et des buts visuels.
- **Satisfaction** : On ne pourra pas toujours trouver une configuration optimale pour la caméra, il faut donc définir des degrés dans la satisfaction des solutions envisagées, afin d'afficher les points les plus importants en tout temps sans trop de perte d'information.

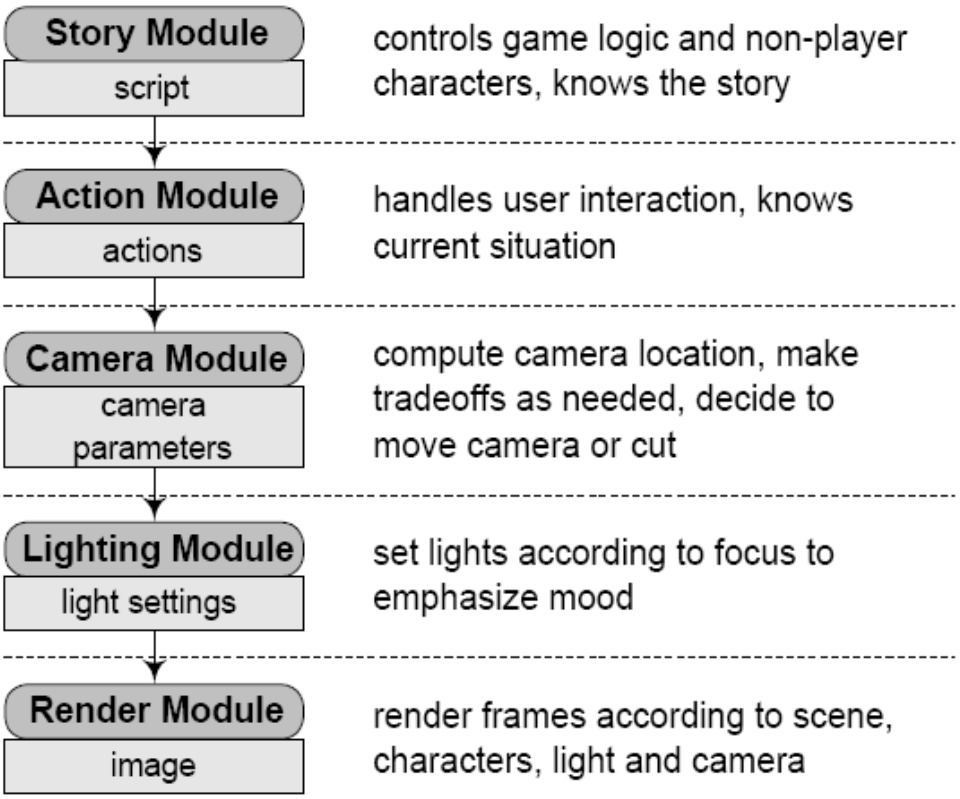


Figure 18 **Organisation hiérarchique des modules de jeu**

L'expérience des travaux précédents nous permet de dresser une liste des propriétés intéressantes qu'un shot doit vérifier, dans le cas d'un jeu vidéo ; retenons celles qui paraissent les plus importantes :

- Placer la caméra à la bonne hauteur avec un bon angle de vue.
- Au lieu de compter les pixels appartenant à un objet pour évaluer sa taille apparente et donc régler le zoom, préférer placer la caméra à un certain rayon fixe de l'objet : on évite ainsi des oscillations de la caméra.
- Faire en sorte que la cible à filmer reste toujours visible à l'écran.

2.2.3.2 Développement d'un système présentant les caractéristiques précédentes

Remarquons tout d'abord que les contraintes changent au fur et à mesure que les objets se déplacent, et qu'elles doivent donc être réévaluées à chaque instant. Ainsi on ne peut pas baser notre système

simplement sur ces contraintes, auquel cas la caméra sauterait brusquement d'une image à une autre sans aucun respect des règles cinématographiques énoncées précédemment, entraînant donc un rendu visuel désastreux. Il faut avant tout maintenir la cohérence des images successives pour obtenir un mouvement continu de la caméra. C'est dans cette optique que les auteurs proposent un schéma de résolution de contraintes pas à pas, image par image : la caméra s'ajuste progressivement pour satisfaire aux exigences cinématographiques telles que l'angle de vue, la distance au sujet filmé, la visibilité etc. La figure 19 indique les différentes étapes. Notons que l'ordre dans lequel elles sont effectuées minimise l'influence de la satisfaction d'une contrainte sur la précédente : ainsi l'application d'une contrainte ne modifie que très peu l'optimalité des paramètres des contraintes précédentes.

En plus de cela, le système de filmage proposé permet de résoudre des contraintes sur plusieurs objets simultanément (par des techniques que nous ne détaillons pas, celles-ci n'étant pas intéressantes dans le cadre de notre travail).

En ce qui concerne le problème de la visibilité, les auteurs ont choisi d'appliquer une méthode n'ayant pas les défauts de celles précédemment utilisées dans la littérature ; en effet aucune des méthodes existantes ne permettaient alors de résoudre le cas de la figure 20, où la cible à filmer est indiquée par « target ». La technique utilisée se base sur les « régions de visibilité potentielles » (PVR ou Potential Visibility Regions, en anglais) : l'idée est d'utiliser des polygones pour délimiter les régions plus ou moins désirables, ces polygones étant de couleur d'autant plus claire qu la zone est intéressante. Plus le nombre d'obstacles (ainsi que leur épaisseur) obstruant la vue de la caméra est élevé, moins la zone est désirable. On obtient ainsi une carte des zones préférentielles pour la caméra.

Enfin, il faut noter qu'afin d'obtenir un agréable rendu visuel, la caméra doit avoir quelques indications sur le futur mouvement probable de l'utilisateur. Pour cela, on calcule la prochaine position de la caméra un instant δt après la position courante à l'aide de sa vitesse, accélération, inertie etc : on peut alors deviner où l'objet se trouvera sur une image proche dans le futur, puis on réajuste nos calculs en fonction de la position effectivement observée. On peut aussi encoder certaines heuristiques comme « se déplacer dans la direction du mouvement » etc. La figure 21 présente un bilan des avantages acquis à l'aide de la méthode de filmage décrite précédemment.

Remarque : On n'utilise pas cette fois les concepts d'idiomes et shots d'une manière élaborée, pour une raison simple : filmer un jeu vidéo hautement interactif est très différent du tournage d'un film, on ne peut pas faire plusieurs prises pour déterminer quelle succession de shots est préférable. Ainsi, il semble plus naturel de repartir de zéro en définissant d'autres critères propres aux jeux vidéos (et à notre application, qui peut être considérée comme un jeu consistant à déplacer le bras d'un point à un autre en évitant les obstacles).

En conclusion, les principales améliorations apportées par ce travail sont la possibilité de prévoir les mouvements des sujets filmés afin de mieux ajuster la caméra, ainsi que la résolution de contraintes image par image, et enfin un nouveau système évitant l'occlusion de la caméra.

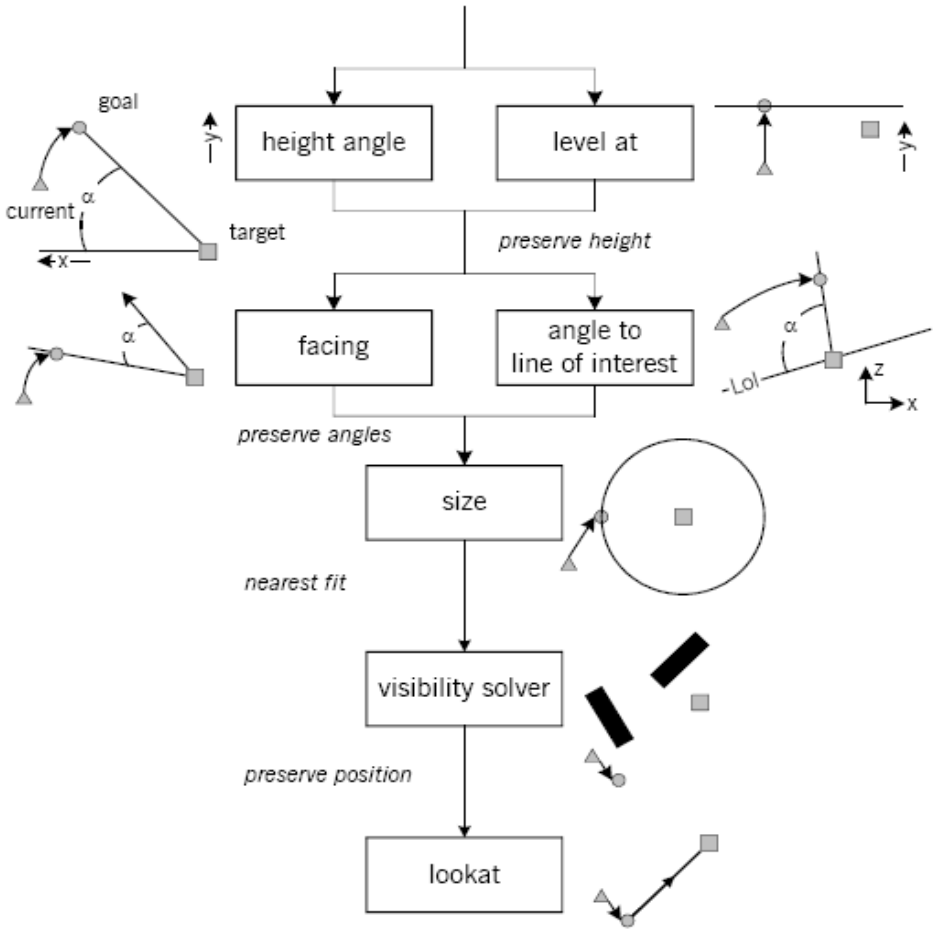


Figure 19 **Ordre d'application des contraintes**

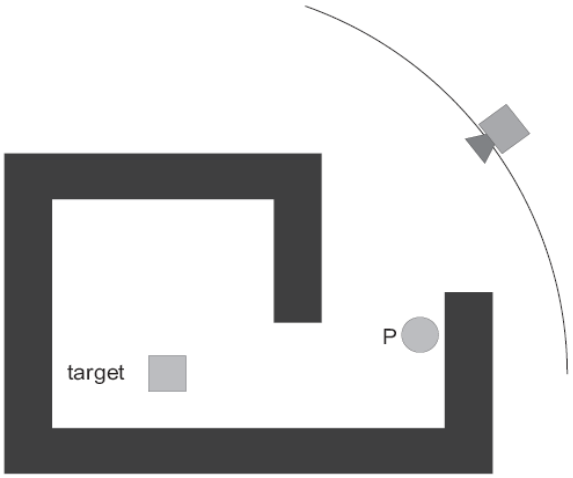


Figure 20 **Problème des méthodes classiques anti-obstruction lorsqu'un parcours des obstacles en profondeur est nécessaire**

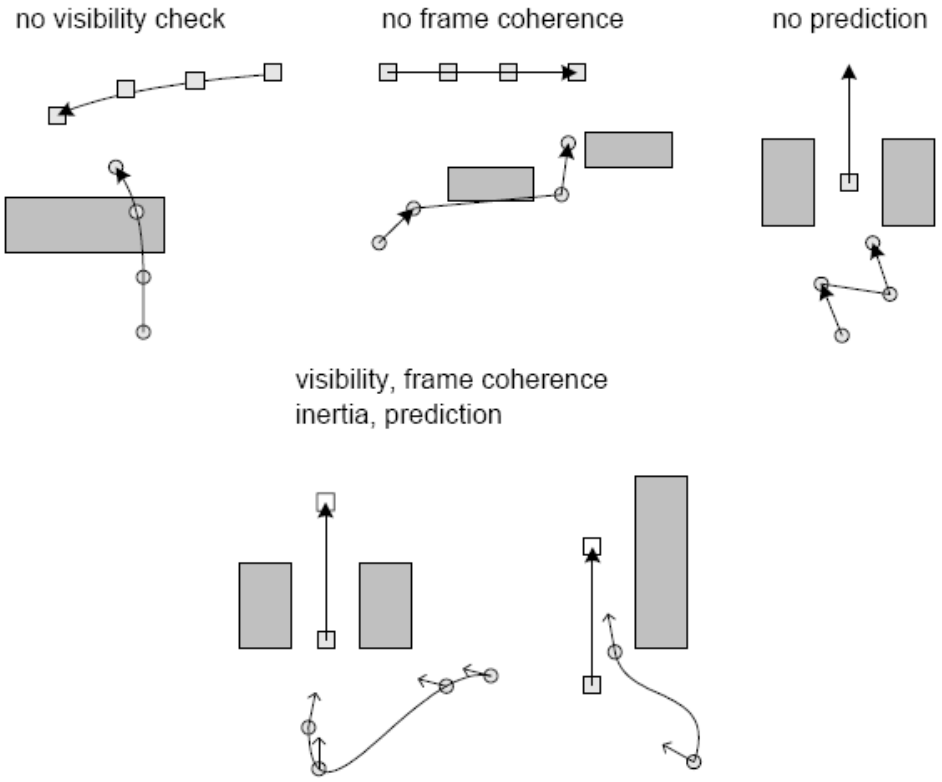


Figure 21 **Problèmes survenant lorsque certaines contraintes ne sont pas satisfaites**

Chapitre 3

Aspects techniques préliminaires

Nous détaillons ici quelques points techniques nécessaires à la suite du développement.

3.1 Obtention des coordonnées 3D du bras

On retranscrit ici les coordonnées du bras-robot en 3D afin de pouvoir calculer (entre autres) le centre de gravité, très souvent utile pour déterminer la direction de visée d'une caméra lors du tournage. Les calculs ici se basent sur une modélisation simpliste du bras par deux segments « épaule-coude » et « coude-poignet », car le but est avant tout de donner une idée générale de la façon dont sont obtenues les coordonnées. La méthode réelle prenant en compte chaque joint de rotation est plus longue et complexe, mais similaire.

Voici tout d'abord les significations de chaque paramètre d'une configuration *q* du bras-robot (la figure 22 les présente visuellement) :

- q[0] : valeur de translation le long du rail, allant de 0 (tout en bas) à L (longueur de la station). C'est la hauteur du point sur la figure précitée.
- q[1] : angle θ_e formé par l'axe Oz et le vecteur épaule→coude (pitch).
- q[2] : angle φ_e formé par l'axe Ox et le vecteur épaule→coude en projection sur le plan xOy (yaw).
- q[3] : angle ξ de rotation de l'axe épaule-coude sur lui-même (roll).
- q[4] : angle θ_c formé par l'axe Oz et le vecteur coude→poignet (pitch).
- q[5] à q[7] : paramètres de la main, non utiles pour notre travail.

Nous allons déterminer les positions des trois joints à partir de ces données.

On obtient naturellement la position de la base du bras (épaule) comme suit :

x_e = rayon de la station

y_e = q[0]

z_e = 0 (sur le rail, voir la figure 6**)**

La position du coude s'obtient facilement par des calculs en coordonnées sphériques :

$x_c = x_e + \cos(q[2])\ \sin(q[1])\ l_{ec}$

$y_c = y_e + \cos(q[1])\ l_{ec}$

$z_c = \sin(q[2])\ \sin(q[1])\ l_{ec}$,

où l_{ec} est la longueur du segment [épaule,coude].

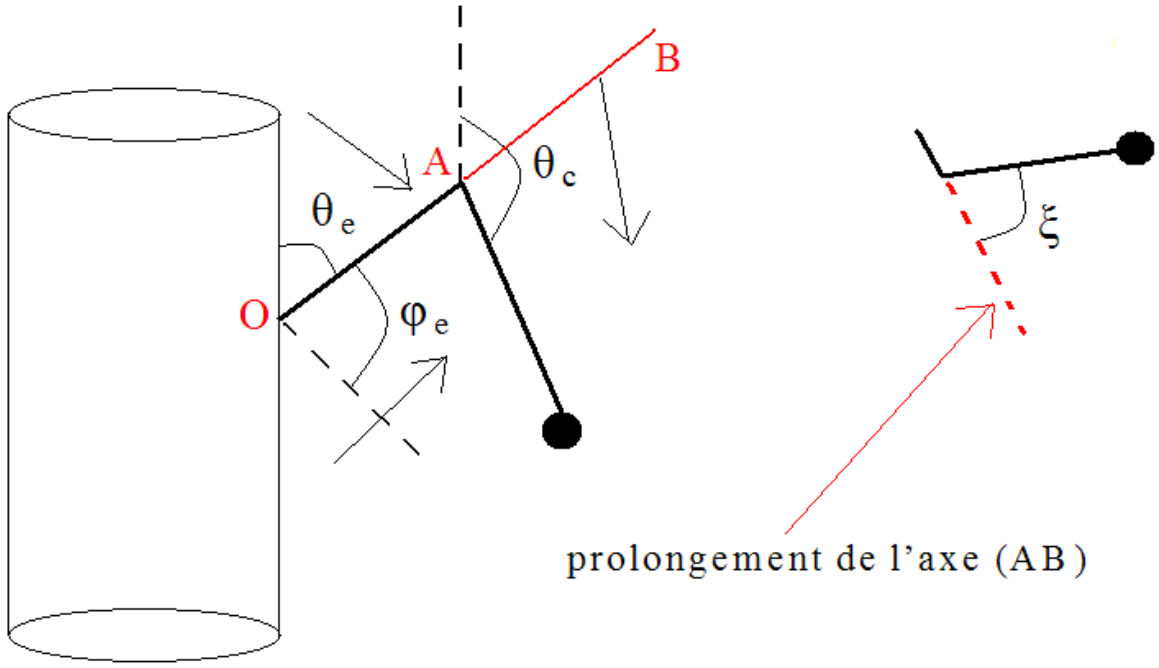


Figure 22 **La paramétrisation du bras-robot**

Pour le poignet c'est plus compliqué car il faut prendre en compte l'angle de rotation « roll » ξ. On commence donc par supposer que ξ = 0 et on applique ainsi une méthode similaire à la précédente :

$\mathbf{x_p = x_c + \cos(q[2])\ \sin(q[4])\ l_{cp}}$

$\mathbf{y_p = y_c + \cos(q[4])\ l_{cp}}$

$\mathbf{z_p = z_c + \sin(q[2])\ \sin(q[4])\ l_{cp}}$,

où l_{cp} est la longueur du segment [coude,poignet].

Ensuite, il faut appliquer une rotation d'angle ξ et d'axe épaule→coude au point terminal (poignet). Cela s'effectue techniquement de la manière suivante :

1) Calcul des coordonnées *Cp* du point terminal actuel dans le repère d'origine la base du bras, translaté du repère global.
2) Calcul de la matrice de passage **P** de changement de base [base du repère global ➔ base orthonormée ayant le vecteur épaule→coude comme axe Oz, choix arbitraire des axes Ox et Oy]. Dans cette dernière base la rotation d'angle ξ s'écrit très facilement sous la forme $\mathbf{R} = \begin{pmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$.
3) Calcul de $C_p' = \mathbf{P\ R\ {}^tP}\ C_p$
4) On obtient $\mathbf{(x_p, y_p, z_p) = (x_e, y_e, z_e) + C_p'}$

Voir le code dans le fichier ***Configuration.cpp***.

3.2 Découpage en séquences élémentaires

Afin de pouvoir filmer en accord avec la notion de scènes et d'idiomes par exemple, il est nécessaire de morceler la trajectoire du bras-robot en les parties correspondantes aux scènes et séquences. Pour cela, il faut déterminer des critères sur la trajectoire du bras.

3.2.1 Découpage selon les sens de déplacements projetés

Nous avons à notre disposition une suite de configurations du bras-robot, une configuration étant un point de l'espace dans lequel se déplace le bras, en l'occurrence $\mathbf{R}^8$. Il s'agit donc de découper cette suite de points (supposés assez rapprochés pour que la discrétisation n'entraîne pas de perte d'information sur le mouvement continu originel) afin d'obtenir des fragments de trajectoire cohérents.

La première idée venant à l'esprit est qu'un tel fragment de trajectoire doit voir chacune des composantes du vecteur de $\mathbf{R}^8$ – représentant les points successifs – monotone ; c-à-d : les projections de ses fonctions composantes dans **R** doivent être monotones, au sens mathématique du terme. En effet, il semble naturel qu'un mouvement élémentaire ne soit pas erratique, qu'il possède une certaine uniformité (penser au mouvements d'un lanceur de disque, marteau ou javelot en athlétisme par exemple). L'algorithme 2, prenant en paramètre un vecteur de configurations **C**, effectue ce travail; le champ 4 de la troisième configuration s'obtient par exemple par **C**[3][4].

1) Mémoriser les premières variations **U**[j] = **C**[1][j] – **C**[0][j]
2) Boucle : pour i allant de 2 à **C**.size() – 1, faire :
    a) Pour j parcourant chaque dimension, faire : **V**[j] = **C**[i][j] - **C**[i-1][j]
    b) S'il existe un indice k tel que **U**[k] ≠ **V**[k], marquer i comme point de rupture.
    c) **U** ← **V** ;

Algorithme 2 **Découpage de la trajectoire selon les mouvements monotones**

Ce premier algorithme prend en compte toutes les variations de trajectoire, donc est très sensible au bruit (mouvements parasites). On peut l'améliorer en calculant ensuite la moyenne des longueurs des segments de trajectoires, puis en fusionnant les segments parasites trop courts par rapport à la moyenne avec les segments adjacents aux variations similaires. Voir le code correspondant à cette dernière version améliorée dans ***DecoupMonotone.cpp***.

On obtient ainsi en particulier les segments sur lesquels seule la main bouge (en vérifiant que le sens de déplacement est neutre sur tous les autres joints), cela nous sera utile lorsqu'il faudra filmer en zoomant sur la main. Cependant, la structure des points calculés alors peut comporter des anomalies, pour diverses raisons à commencer par des imprécisions de mesure ou des mauvais mouvements de l'astronaute apprenant. Supposons par exemple que l'on obtienne ceci, où les points à **1** sont ceux où seule la main bouge, ceux à **0** représentant un autre type de filmage :

0000001001011110010101111100010000000000…

L'algorithme présenté en annexe A (97) permet de ramener ce vecteur à :

0000000001111111111111111100000000000000… ,

Nous disposons alors d'une suite ininterrompue de points où l'on doit filmer en zoomant sur la main. Le film sera beaucoup plus agréable à regarder comme cela.

3.2.2 Découpage suivant les proximités aux obstacles

On cherche à découper la trajectoire en morceaux le long desquels le bras est approximativement à la même distance des obstacles. Cette idée est motivée par le fait que si le bras passe dans une zone encombrée de l'espace, le filmage sera beaucoup plus contraint au niveau visuel, et on devra fort probablement changer d'angle de vue. Ainsi on repère à l'avance les points intéressants pour changer de caméra, dans la même optique qu'au paragraphe précédent.

Le problème réside dans la façon de détecter la proximité aux obstacles : on peut le faire de manière exacte en utilisant des des librairies graphiques comme **PQP** (Proximity Query Package), ou bien utiliser un algorithme probabiliste approchant les distances (algorithme 3). Ce dernier est suffisant pour estimer les distances dans le cas qui nous préoccupe, mais nous constatons expérimentalement qu'au-delà de 20 à 30 points échantillonnés **PQP** est plus rapide (et plus précis). L'idée dans cet algorithme est simplement d'échantillonner un certain nombre de points autour du bras-robot selon une loi gaussienne, puis de supposer que la plus courte distance à un obstacle est approximativement égale à la plus courte distance entre un point

échantillonné en collision et le bras. L'avantage de cette méthode est aussi que l'on peut régler le nombre de points échantillonnés en fonction de la vitesse souhaitée lors de l'animation.

Bien sûr les distances ainsi calculées par l'algorithme 3 sont parfois assez éloignées de la réalité, mais les tests indiquent qu'elles sont suffisamment correctes pour obtenir un découpage cohérent. Finalement, nous présentons l'algorithme décidant des points de coupure (= éventuels changements de caméra) selon les proximités aux obstacles : algorithme 4 ; « Recherche des points de coupure basés sur les distances aux obstacles », voir le code dans ***DecoupObstacles.cpp***.

Pour une image :

σ ← sigma_Min

Tant que σ < sigma_Max {

    Réaliser un échantillonnage gaussien le long des axes définis par l'avant-bras, le bras et le poignet (pouvant être considéré comme ponctuel)

    Pour chaque point (x,y,z) échantillonné :

        vérifier_collision ;

        si collision, alors :

            on considère que l'obstacle trouvé est le plus proche et approxime la distance à celui-ci par le plus court segment du bras à (x,y,z).

            break Tant que ;

    σ ← next_ σ_Value

Algorithme 3 **Calcul de distances par échantillonnage**

```
valeursDist ← vecteur de réels initialisé à la taille 1 avec la première distance calculée
moy ← première distance calculée
outsiders ← vecteur de distances (réelles) supposées dans un premier temps être absurdes
compteur ← 1
Boucle sur chaque image du film où l'on calcule la distance d aux obstacles :
        si ( |d – moy| > seuil ) :
                outsiders.add(d) ;
                si ( outsiders.size() = 4 ) :
                        points_de_coupure.add(compteur) ;
                    compteur ← compteur + 4 ;
                        valeursDist.clear() ;
                        valeursDist ← outsiders ;
                    moy ← moyenne de valeursDist ;
        sinon :
                compteur ← compteur + outsiders.size() + 1 ;
                outsiders.clear() ;
                moy ← ( moy + d ) / 2 ;
```

Algorithme 4 **Recherche des points de coupure basés sur les distances aux obstacles**

Remarque 1 : Le seuil dans la première instruction conditionnelle de ce dernier algorithme doit être très bien choisi, sous peine de considérer une augmentation monotone de la distance comme une stagnation dans la même zone.

Remarque 2 : il est possible que dans ce même algorithme un groupe de points « outsiders » contienne des valeurs de distances très différentes et donc que leur moyenne ne soit pas significative, mais ce cas est

rarissime (non observé sur tous les tests effectués). L'algorithme sert surtout à détecter les changements significatifs de distance aux obstacles, il ne vérifie pas la cohérence des distances.

3.2.3 Reconnaissance des mouvements du centre de gravité

On calcule le centre de gravité du bras-robot pour chaque point dicrétisé en assignant à chaque milieu de segment un poids correspondant à sa longueur (la main est considérée comme une sphère contrairement aux segments « épaule-coude » et « coude-poignet »). On obtient alors une trajectoire en 3D au lieu d'avoir à prendre en compte huit dimensions :
reconnaître un tel déplacement via des courbes de $\mathbf{R}^3$ est beaucoup plus simple, et c'est l'unique motivation de cette approche. En contrepartie on perd en précision, différentes postures du bras pouvant correspondre à la même position du centre de gravité (non injectivité de la transformation).

On peut par exemple reconnaître des déplacements dans un même plan, sur un même segment, ou encore des arcs de cercles etc. L'algorithme n'est pas facile à déterminer, mais on peut imaginer ceci : on regarde les trois premiers points suffisamment différents (en éliminant le bruit), puis on teste leur colinéarité ; dans le cas d'une réponse négative on ajoute
le quatrième point et on teste la coplanarité de tous ces points ; enfin, dans le cas d'une réponse négative à la dernière question, on doit poursuivre l'analyse plus profondément (ce pourrait être un mouvement vissé : déplacement dans un plan et déplacement de ce plan suivant un axe, par exemple, etc).

Remarque : on pourrait effectuer cette analyse en apprenant la topologie de la courbe 3D définie par le centre de gavité, on dispose en effet d'assez de littérature sur le sujet, mais comme on le verra plus loin c'est inutile car les autres méthodes sont plus intéressantes. En revanche, le simple parcours du centre de gravité en trois dimensions nous sera utile dans le paragraphe suivant.

3.2.4 Découpage suivant les zones de la station

Finalement, voici une méthode se proposant de découper la trajectoire non pas en fonction des déplacements du bras, mais en fonction des zones de la station dans lesquelles le bras se déplace successivement. Cette idée est justifiée par le fait qu'il est hautement probable que l'apprenant effectue souvent certaines actions particulières aux mêmes endroits.

Nous divisons assez grossièrement la station en six zones : deux proches des panneaux solaires de part et d'autre du milieu *M* du cylindre principal dans le sens de sa longueur, deux zones dégagées symétriques par rapport à *M* mais plus proches que les panneaux solaires, une zone entourant l'obstacle « navette spatiale » et enfin une entourant les obstacles collés à la navette spatiale. Une visualisation du morcelage effectué se trouve sur la figure 23.

Un découpage plus raffiné pourra être effectué par la suite, mais l'étude du filmage tenant compte du passage du bras dans certaines zones est reportée à une date ultérieure, faute de temps. L'idée est ensuite de définir des idiomes dépendamment de la zone dans laquelle se trouve le bras, puis de filmer la trajectoire du bras selon ces idiomes.

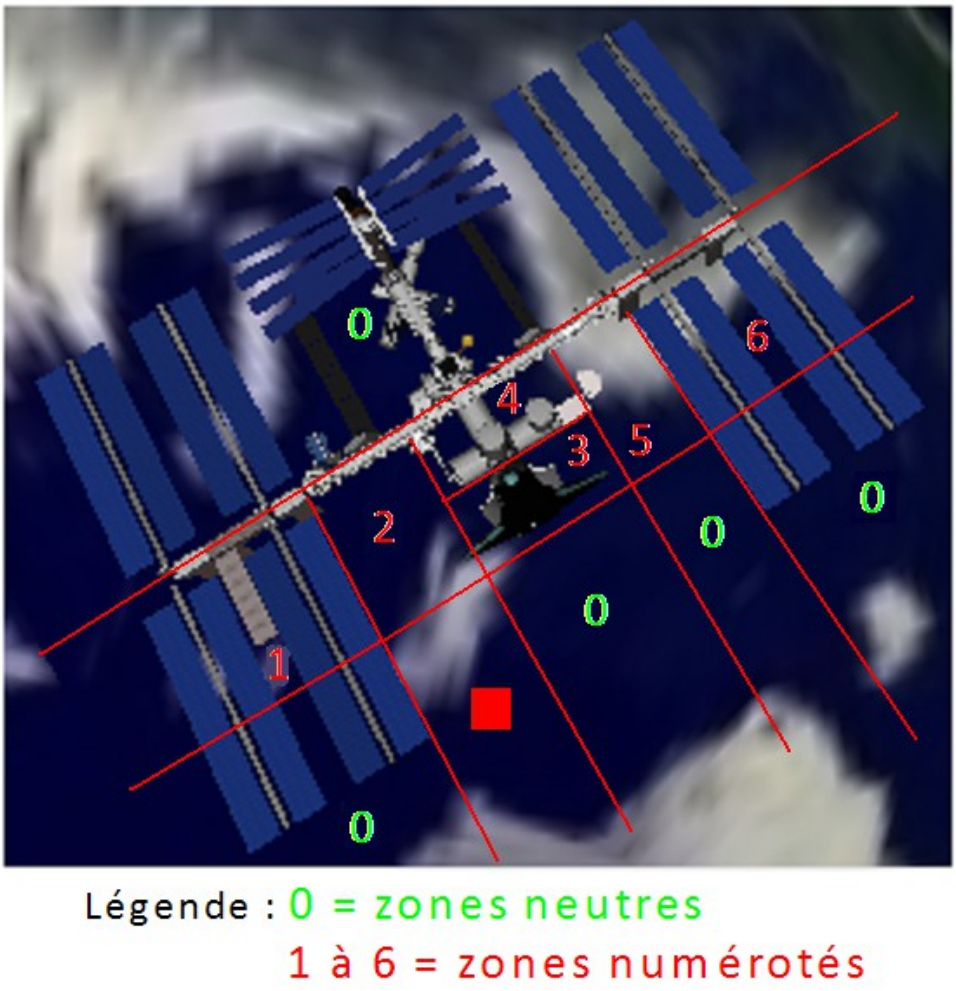


Figure 23 **Découpage en zones autour de la station**

3.2.5 Bilan

Considérons les deux mouvements suivants (voir la figure 24) :

1) Le joint du coude se plie vers le bas tandis que la base du bras ainsi que le joint du poignet restent aux mêmes positions. Le centre de gravité descend alors dans une légère courbe car la main influe sur le déplacement comme un satellite perturberait l'orbite d'une planète.
2) Seul le joint de la base du bras provoque un mouvement global du bras vers le bas, celui-ci restant donc parfaitement rectiligne globalement : on observe aussi une légère courbe en arc de cercle.

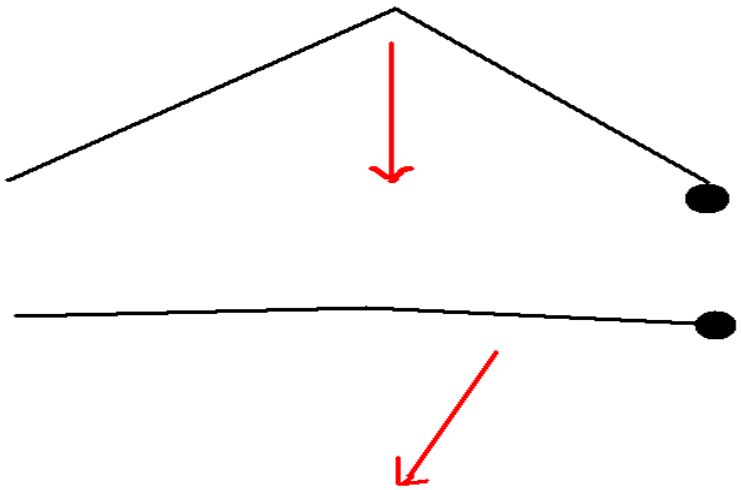


Figure 24 Deux mouvements du bras-robot

Dans les deux cas le mouvement détecté du centre de gravité est à peu près le même, et pourtant les déplacements du bras correspondants sont très différents. On utilisera donc pas un découpage suivant le mouvement du centre de gravité lorsque l'on base le morcellement uniquement sur la trajectoire du bras.

Vérifions que la méthode présentée en 3.2.2 (46) s'effectue bien sur un exemple simple (ne correspondant pas à la vraie géométrie de la station, pour simplifier l'interprétation des résultats). La figure 25 rappelle le repère local au voisinage du bras à la surface de la station, en regardant dans la direction de l'axe principal de la station (Oy).

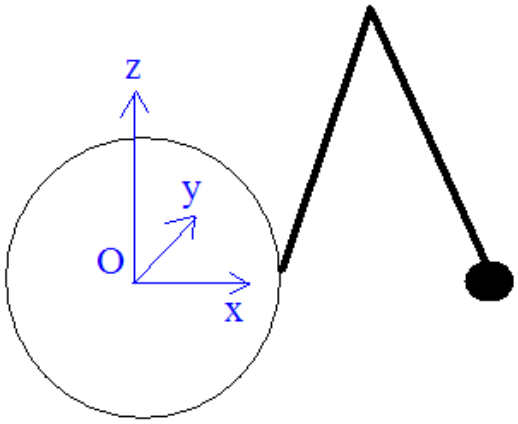


Figure 25 **Repère local près du bras**

- Il y aura quatre obstacles en plus de la station considérée cylindrique :
  - **obstacle 1**: {rayStation < x < rayStation + 5 ; 50 < y < 150 ; -4 < z < -1}
  - **obstacle 2**: {rayStation + 10 < x < rayStation + 15 ; 350 < y < 450; -2 < z < 2}
  - **obstacle 3**: {rayStation < x < rayStation + 5 ; 550 < y < 650 ; 1 < z < 4 }
  - **obstacle 4**: {rayStation +10 < x < rayStation + 15 ; 850 < y < 950 ; -4 < z < 1}

  où *rayStation* est le rayon du cylindre modélisant la station.
- Le bras démarre en y = 0 et termine en y = 1000, dans la configuration

  (0 ou 1000, π/4, 0, 0, 3π/4, 0, 0, 0)

  à chaque fois, *i.e.* : translation de 0 ou 1000 mètres par rapport à l'origine, vecteur épaule→coude faisant un angle de π/4 avec l'axe Oz, et vecteur coude→poignet faisant un angle de 3π/4 avec Oz.
- Sa trajectoire est déterminée à la main dans l'espace 8D des configurations, de manière à éviter les obstacles ; voici le détail de cette trajectoire, y = :
  - 0 à 300 : configuration = (y, π/4, 0, 0, 3π/4, 0, 0, 0) (pas de changements)
  - 300 à 350 : passage monotone jusqu'à la configuration (y, π/4, 0, 0, π, 0, 0, 0)
  - 350 à 450 : configuration = (y, π/4, 0, 0, π, 0, 0, 0) (pas de changements)
  - 450 à 500 : rétablissement progressif pour revenir à la configuration initiale

- 500 à 550 : passage monotone jusqu'à la configuration (y, π/2, 0, 0, π, 0, 0, 0)
- 550 à 650 : configuration = (y, π/2, 0, 0, π, 0, 0, 0) (pas de changements)
- 650 à 700 : rétablissement progressif pour revenir à la configuration initiale
- 700 à 800 : configuration = (y, π/4, 0, 0, 3π/4, 0, 0, 0) (pas de changements)
- 800 à 850 : passage monotone jusqu'à la configuration (y, π/4, 0, 0, π, 0, 0, 0)
- 850 à 950 : configuration = (y, π/4, 0, 0, π, 0, 0, 0) (pas de changements)
- 950 à 1000 : rétablissement progressif pour revenir à la configuration initiale

On obtient alors via l'algorithme de filmage en direct utilisant l'échantillonnage gaussien les points de coupure suivants (avec le seuil 3.0 dans l'algorithme 4) :

42, 155, 342, 458, 540

660, 664 → ces deux valeurs rapprochées sont une anomalie, due à l'incertitude probabiliste. Un bon choix du seuil permet toutefois de minimiser suffisamment ces défauts.

843, 956

La figure 26 donne les distances aux obstacles calculées pour chaque point de la trajectoire (une distance de 20 signifie qu'aucun obstacle n'a été détecté). Les coordonnées des obstacles considérés comme proches le long de la trajectoire ne sont pas vraiment pertinentes, et ne sont donc pas indiquées dans les résultats.
Les sauts dans ce graphique correspondent bien aux points de coupure calculés.

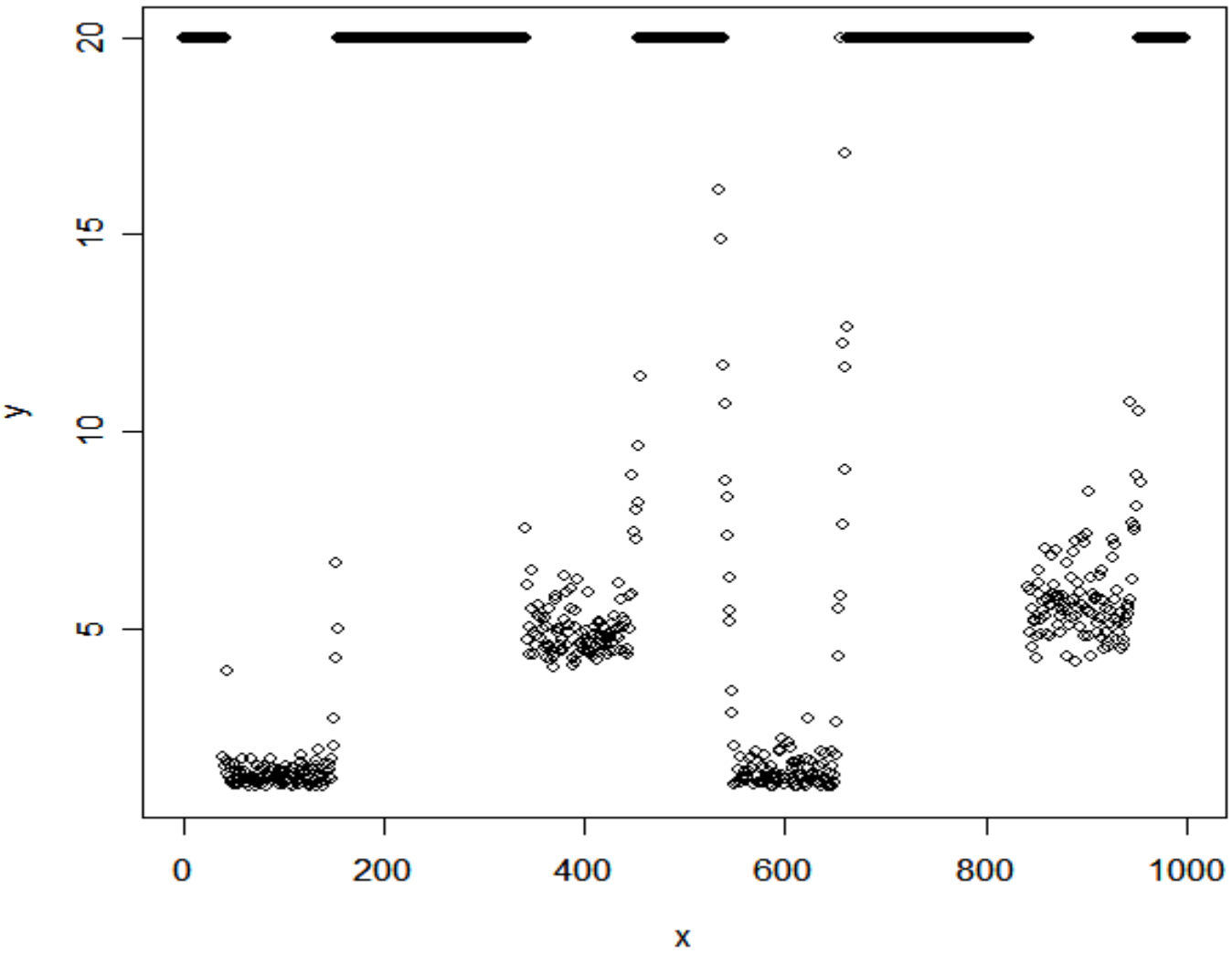


Figure 26 **Distances aux obstacles en fonction du point**

3.3 Codage des paramètres d'une caméra

Pour coder les sept paramètres d'une caméra nous avons deux possibilités : soit l'on utilise trois coordonnées de position, trois angles dont deux indiquant la direction de filmage (les angles φ et θ des coordonnées sphériques) et le zoom, soit trois coordonnées de positions, trois coordonnées pour un point cible (remplaçant les angles φ et θ dans la modélisation précédente), un angle (le roll) et le zoom. Les paramètres

de la première modélisation correspondent à ceux indiqués sur la figure 7, avec φ = pan, et θ = tilt, que nous rappelons ici :

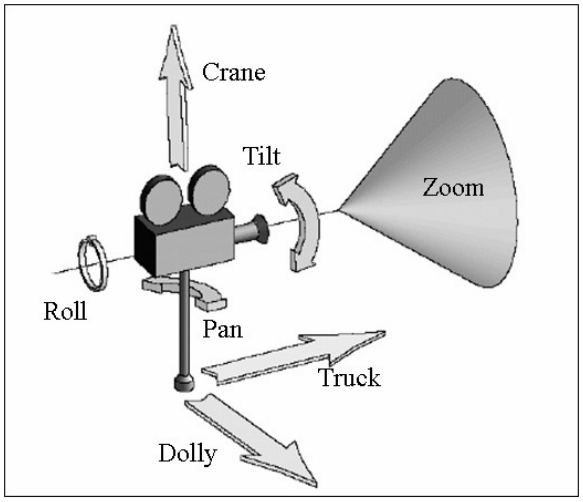


Figure 7 – **rappel – paramètres d'une caméra.**

Ces deux modélisations sont mathématiquement équivalentes (la seconde étant d'ailleurs redondante, nous n'avons pas besoin de la distance à la cible pour définir l'orientation), mais la librairie graphique **Coin3D** utilise systématiquement la modélisation avec un point cible car celle-ci facilite entre autres l'interpolation d'images. Nous nous fixons donc sur cette seconde modélisation dans le cadre de l'intégration du système de filmage à *RomanTutor*. Notons de plus qu'il est très facile d'obtenir les angles φ et θ de la caméra quand cela s'avère nécessaire, en cherchant les coordonnées spéhriques du vecteur caméra→point cible.

3.4 Détermination de la vue d'une caméra

Ce paragraphe intermédiaire a pour but de constituer l'image 2D qu'une caméra voit, et par là même pouvoir détecter les obstructions empêchant de voir le bras-robot ; ce sera utile tout au long du filmage, la contrainte la plus importante étant bien sûr celle-ci.

3.4.1 Méthode avec PQP

On vérifie d'abord que le bras-robot est bien dans le cône de vision de la caméra via des calculs géométriques (changements de repères), puis il suffit de vérifier que la projection du bras sur le plan de vision de la caméra contient tout le bras. En d'autres termes, il faut et il suffit que chaque segment issu d'un point de la structure du bras-robot et aboutissant à la caméra ne rencontre aucun obstacle ; et, si cela n'est pas possible, il faut être capable de fournir une solution la moins obstruée possible. On se trouve donc face à un problème d'optimisation.

3.4.1.1 Inclusion du bras-robot dans le cône de vision

Rappelons tout d'abord l'équation d'un cône de révolution d'axe (Sz) et de sommet *S* dans l'espace, dans un repère orthonormé adapté ayant (Sz) pour axe Oz :

$$x^2 + y^2 = z^2(\tan\ a)^2 ,$$

où *a* est l'angle du cône, formé par l'axe et une génératrice. Un point (x,y,z) est donc intérieur au cône si et seulement si $x^2 + y^2 \leq z^2 (\tan a)^2$. Le repère est facilement obtenu en se basant sur l'orientation et la position d'une caméra : il ne reste alors plus qu'à traduire la position du bras en coordonnées 3D, puis à vérifier l'inclusion du bras dans le cône. Le bras-robot ayant une structure rectiligne par morceaux, et le cône de vision de la caméra étant convexe, il suffit de vérifier que les trois joints principaux sont dans le champ théorique de vision de la caméra.

Le cône est l'approximation la plus simple que l'on peut faire de la vision d'une caméra, tout en étant assez réaliste pour l'application qui nous préoccupe. D'autres approches plus fines sont indiquées dans 108, chapitre 3.

3.4.1.2 Discrétisation de la structure du bras-robot

Bien sûr on ne va pas vérifier tous les segments issus de chaque point du bras-robot, mais en choisir un certain nombre intelligemment. Le bras est décrit schématiquement par deux cylindres de l'épaule au coude puis du coude au poignet, et par une sphère englobant le poignet. On va donc échantillonner un certain nombre de points (loi uniforme) sur chaque sous-partie du bras, puis tester seulement les segments issus de ces points.

On utilise 120 points échantillonnés, dont 50 entre l'épaule et le coude, 50 entre le coude et le poignet et 20 au voisinage de la main. Ce choix est assez arbitraire pour obtenir un bon compromis qualité / temps de calcul ; d'autres valeurs pourraient convenir aussi. Voir la figure 27 pour une illustration de tout cela. Sur cette dernière figure, les cercles en rouges correspondent aux zones reconnues comme en collision dans le paragraphe qui va suivre. La visibilité si l'on arrête le processus aux cinq segments indiqués serait donc de 3/5, correspondant au ratio des segments non obstrués sur ceux rencontrant un obstacle.

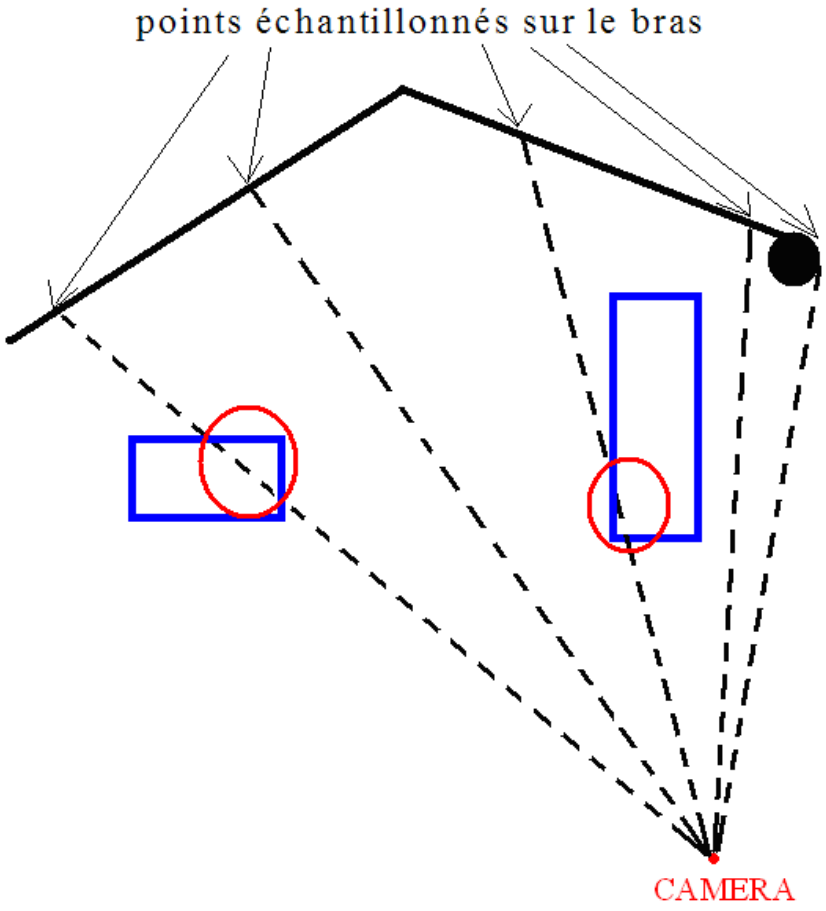


Figure 27 **Test des segments bras-caméra**

3.4.1.3 Tests de collision le long de chaque segment issu du bras

Supposons que l'on ait échantillonné un point P sur le bras-robot, et que la caméra soit située en C dans un repère cartésien de la station, arbitraire mais fixe. Alors on appelle la fonction élémentaire

PQP_Collide() avec en arguments un triangle de sommets P, P+ε, C (modélisant le segment [PC], ε << 1), ainsi que chaque obstacle de la station. Si cette fonction indique qu'il y a collision avec un des obstacles, alors on marque que le segment [PC] est en collision (en pratique on incrémente simplement un compteur).

La caméra a « très probablement » une vue claire seulement si « presque » aucun des points échantillonnés n'a été marqué comme occulté au travers l'algorithme précédent. Pour renforcer la précision sur la vue qu'a effectivement la caméra, on peut regrouper plusieurs images ensemble : le tout ne dure qu'ne fraction de seconde pour l'humain, mais permet de se rendre compte si plusieurs zones différentes du bras sont occultées, auquel cas il faut vite trouver une vue complètement différente de la caméra. Sinon, si seule une petite zone du bras

est occultée, on peut tenter de déplacer légèrement la caméra par optimisation locale discrète. Pour connaître l'importance de l'occlusion, on renvoit un réel entre 0 et 1 égal à 0 si un des joints n'est pas dans le champ de la caméra, et au ratio du nombre de points non occultés par le nombre total de points échantillonnés sinon. Voir le code de tout cela dans ***Camera.cpp***.

Remarque : Tout ceci est simplifié dans le cas où l'on zoome sur une région spécifique du bras, comme la main par exemple. Il suffit alors de restreindre l'algorithme aux régions considérées.

Note : cette méthode est en fait très lourde, compte-tenu de la multitude de tests de collision nécessaires via **PQP**. On pourrait redéfinir la géométrie de chaque obstacle sous forme de systèmes d'inéquations, et calculer beaucoup plus vite si les segments testés sont en collision, mais c'est hors de portée de cette étude. En revanche, une méthode basée sur la comparaison de buffers d'image est possible via la librairie **Coin3D**, et bien que conceptuellement plus lente (on doit tester chaque pixel au lieu d'un échantillon), elle s'avère beaucoup plus rapide car utilisant des méthodes beaucoup plus efficaces pour calculer le rendu visuel. Nous la présentons à présent.

3.4.2 Méthode avec deux buffers d'images

L'idée est cette fois d'utiliser une caméra virtuelle provenant directement de la librairie **Coin3D**, et d'afficher sur un écran virtuel deux images différentes : une comportant tous les obstacles et bien sûr le bras-

robot, et une ne comportant que le bras-robot seul. Ainsi, on obtient la visibilité en comptant les pixels appartenant au bras sur la première image, que l'on divise par le nombre maximal de pixels du bras-robot trouvé sur la seconde image.
Pour réaliser cela, on change simplement les couleurs des objets dans la scène de manière à ce que le bras Canadarm2 soit facilement distinguable, puis on obtient les deux buffers d'image par des appels aux fonction de la librairie **Coin3D**, auquels il suffit d'appliquer la méthode indiquée. Tout est donc beaucoup plus simple qu'au paragraphe précédent, car on délègue la formation de l'image aux classes adaptées et donc optimisées pour ce travail (via openGL).

Cette dernière méthode est beaucoup plus rapide que la précédente, on l'utilisera donc par la suite, bien que la technique présentée auparavant ait un grand intérêt théorique (elle devient en fait plus rapide si l'on est capable de tester les segments reliant la caméra au bras-robot aussi vite que l'affichage + comptage de pixels via **Coin3D**).

3.4.3 Tests

On place (artificiellement) le bras-robot dans une configuration standard : légèrement plié, relativement « redressé » par rapport au plan (xOz) global. On ajoute aussi un obstacle parallélépipédique d'assez petite taille, volontairement interposé entre la caméra et le bras. La Figure 28 Schéma du test effectué présente une visualisation de ce test. Plus précisément, la configuration du bras est :

(700, π/4, 0, 0, 3π/4, 0, 0, 0) ,

i.e. : translation de 700 mètres par rapport à l'origine sur l'axe Oy, le vecteur épaule→coude faisant un angle de π/4 avec l'axe Oz, et le vecteur coude→poignet un angle de 3π/4 avec l'axe Oz, le plan formé par les trois joints principaux du bras étant orthogonal à Oy.
L'obstacle est situé entre les valeurs suivantes de coordonnées :

x : rayStation + 3m < x < rayStation + 5m

y : 400m < y < 450m

z : -2m < z < 2m ,

où *rayStation* est le rayon du cylindre principal de la station spatiale. L'unité de mesure choisie comme référence est le mètre (m).

Enfin, la caméra est placée en **(x, y, z) = (rayStation+4, 200, 0)** , c'est-à-dire telle que le segment [caméra, centre de gravité] soit quasiment parallèle à l'axe Oy, avec l'obstacle interposé. Les angles de la caméra sont réglés de telle façon qu'elle pointe dans la direction de l'axe Oy (en théorie l'orientation devrait être très légèrement différente, car on prend le centre de gravité comme référence, mais la différence est négligeable).

On obtient alors une visibilité de ***0.683***, correspondant à l'intuition. Lorsque l'on restreint l'obstacle entre les abscisses rayStation + 4m et rayStation + 5m, on obtient une visibilité de 0.892. Les résultats sont donc bien cohérents.

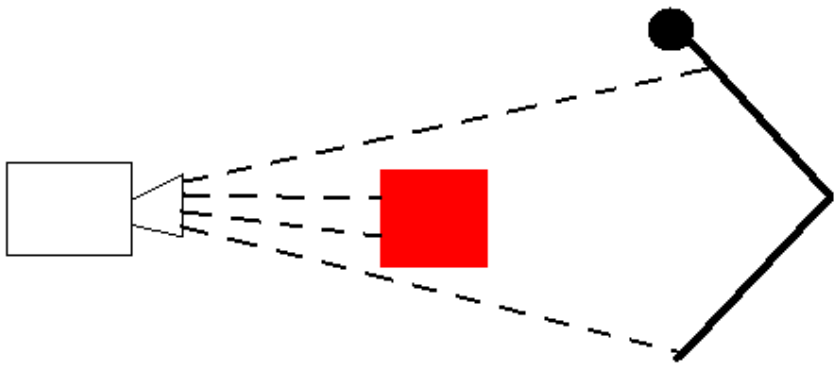


Figure 28 **Schéma du test effectué**

Chapitre 4

Génération d'une animation du bras-robot

Nous allons proposer quelques méthodes adaptées au filmage des déplacements du bras-robot canadien, en partie inspirées de tout ce que l'on a vu jusqu'à présent. Rappelons qu'une animation 3D n'est rien de plus qu'une séquence linéaire et bien agencée d'images (24 à 30 images par secondes). Il faut et il suffit donc de choisir les bonnes images.

4.1 Trajectoire connue : approche cinématographique

Dans ce cas on connaît dès le départ le parcours précis que l'on doit filmer. On peut donc se pemettre d'utiliser une approche cinématographique en utilisant un langage semblable à DCCL : il faut découper la trajectoire assez finement comme indiqué en 3.2 (44), puis, une fois ce découpage effectué, déterminer quels idiomes utiliser sur chaque sous-partie de trajectoire. Nous détaillerons une méthode basée sur le découpage selon les mouvements monotones (en effet un mouvement monotone dans toutes les directions est nécessairement une brique indivisible pour un filmage suivant les actions du bras).
On commence donc par une phase d'analyse des mouvements possibles du bras en définissant les idiomes correspondants.

L'autre méthode basée sur le découpage en zones autour de la station pourra ensuite étendre et raffiner la méthode présentée ici assez linéairement. Nous ne l'implémentons pas faute de temps, et surtout car la démarche serait exactement la même – création de la base d'idiomes, de leurs vues, transformation de la trajectoire et compilation – mais elle sera la première chose à réaliser à la suite de cette maîtrise.

4.1.1 Création de la base d'idiomes

Le bras possédant huit degrés de liberté, et chaque joint pouvant se déplacer de trois manières différentes (croissant, constant, décroissant), on a au total $3^8 = 6561$ mouvements élémentaires à prendre en considération. Les étudier tous un par un serait beaucoup trop long par rapport aux enjeux de ce travail, surtout qu'il faut en plus prendre en compte l'environnement immédiat du bras pour ajuster la caméra. On va donc chercher quels sont les mouvements clés du bras-robot :

- Bras s'élevant pour soulever une charge ou en prévision d'un contournement d'obstacle, et mouvement inverse.
- Mouvement latéral de rotation du bras pour amener une charge sur une trajectoire globalement circulaire.
- Mouvement de rotation du segment épaule-coude sur lui-même (roll).
- Mouvements du poignet pour agripper une charge ou effectuer une opération de maintenance.
- Mouvement de translation en déplacement le long du rail.

Voici les idiomes proposés, correspondants aux mouvements esquissés ci-dessus :

1) *Élévation du bras*

   On utilise (assez logiquement) un shot *pan* vertical.

2) *Mouvement latéral en rotation*

   On utilise un shot *pan* horizontal au dessus du bras (vue ¾).

3) *Rotation du segment épaule-coude*

   On utilise un shot *static* face au bras, c'est en général la vue qui donnera le plus d'informations dans ce cas.

4) *Mouvement du poignet*

   On zoom naturellement sur le poignet avec un shot *static*.

5) *Translation le long du rail :*

   On utilise logiquement un shot *track* avec une vue ¾ du bras-robot, car c'est la vue qui donne le plus d'information dans le cas général.

Remarque : les idiomes ainsi proposé sont constitués d'un unique shot ; il serait très facile de modifier pour avoir plusieurs shots, mais cela semble un peu artificiel dans la mesure où l'effet visuel provoqué reste difficile à évaluer. De plus, l'effet visuel final et la complexité en temps de calcul sont les mêmes, l'idiome est simplement un niveau d'abstraction supérieur.

Voir le code correspondant à la création des idiomes dans le fichier ***IdiomBase.cpp***.

4.1.2 Transformation de la trajectoire en une suite d'idiomes

On commence par découper la trajectoire de manière adéquate, en sous-parties qui correspondront aux idiomes et shots avec la méthode décrite dans le paragraphe 45 : découpage selon les sens de déplacement projetés.

Il suffit de reconnaître les quatre types de mouvements décrits précédemment dans chacune des sous-parties de trajectoire : on commence par regarder si seule la main bouge, auquel cas on applique l'idiome correspondant, puis s'il y a eu translation le long du rail. On regarde ensuite si l'épaule a bougé en rotation « pitch », et, enfin, on vérifie si les joints « yaw » du coude et/ou de l'épaule ont bougé. On filme alors avec l'idiome correspondant dans la base. Au cours du processus on indique les temps de début et de fin de chaque idiome (en fait les indices dans le vecteur de configurations du bras), utiles pour la génération finale de l'animation.

Voici un extrait du code réalisant cette tâche, recherchant si l'épaule a bougé en rotation « yaw » (le reste est similaire, dans ***TransfIdioms.cpp***) :

```
/* test si prédominance d'un mouvement angulaire de rotation, les variables depctRail,
* depctPhi, depctThetaE, depctThetaC et depctRoll représentant respectivements les
* déplacements  en valeur absolue le long du rail, puis des angles yaw et pitch du joint de de * l'épaule,
l'angle pitch du joint du coude et l'angle roll du joint de l'épaule.
* SEUIL_PREDOM est une constante réelle  entre 0 et 1 indiquant le ratio au-delà duquel on * considère que
le mouvement du joint est prédominant sur celui des autres. */
```

```
if (depctPhi>SEUIL_TOL && depctThetaE/depctPhi<SEUIL_PREDOM
                        && depctThetaC/depctPhi<SEUIL_PREDOM
                        && depctRoll/depctPhi<SEUIL_PREDOM
                        && depctRail/depctPhi<SEUIL_PREDOM) {
    // on ajoute un nouvel idiome que si le type du dernier mouvement repéré est
    // différent. Sinon, on fusionne naturellement les deux morceaux contenant le
    // même type de mouvement (note : ce n'est pas obligatoire, et plus délicat alors à
    // la compilation ; dans l'implémentation actuelle la fusion n'est pas utilisé)
    if (lastMouv!=1)  {
        // ajout de l'idiome et initialisation des indices de début du nouvel idiome
        // et fin du dernier idiome ajouté avant :
        resIdioms.back().shots.back().lastInd=decoupageM[i]-1 ;
        resIdioms.push_back(idiomsBase[1]) ;
        resIdioms.back().shots[0].firstInd=decoupageM[i] ;
        // initialisation des angles de vue privilegiés pour l'idiome :
        resIdioms.back().views.push_back(4) ;
        resIdioms.back().views.push_back(5) ;
        resIdioms.back().views.push_back(7) ;
        resIdioms.back().views.push_back(6) ;
        resIdioms.back().views.push_back(2) ;
        resIdioms.back().views.push_back(3) ;
        lastMouv=1 ;
    }
}
```

Note : Comme mentionné dans l'algorithme, on pourrait raffiner les résultats du découpage selon les mouvements monotones en regroupant les segments adjacents sur lesquels les mêmes joints se déplacent (mais probablement dans des directions opposées). Nous n'avons pas implémenté cela pour deux raisons :

- Si par exemple le bras s'élève pour effectuer une certaine tâche, puis s'abaisse en prévision de l'évitement d'un obstacle, il est logique de décomposer le mouvement en deux parties, sans les fusionner en un seul bloc.
- La fusion est beaucoup plus délicate à gérer à la compilation au niveau des mouvements de caméras, qu'elle dénaturerait quelque peu (raison pratique).

4.1.3 Compilation et exécution du film

Une fois que l'on dispose de la suite de shots à filmer, il reste à déterminer les positions exactes de la caméra le long de chaque shot : ce travail est effectuée par l'algorithme / compilateur suivant :

Cas du shot 1) :

On place la caméra sur la droite passant par le centre de gravité du bras-robot, parallèle à l'axe Oy, du côté tel que l'on voie le poignet plus proche que le coude (si ces deux joints sont équidistants, alors on choisit le côté arbitrairement). Ce placement semble plus agréable à regarder que l'autre, même si bien sûr cela reste très subjectif.
Ensuite on fait varier l'angle θ (tilt) de la caméra suivant le mouvement du centre de gravité, en maintenant constants tous les autres paramètres.

Cas du shot 2) :

La position de la caméra est cette fois initialisée légèrement décalée par rapport à la position du centre de gravité sur l'axe Oy, au dessus du bras (on comprend ce que cela signifie intuitivement, voir la figure 29) ; le terme « légèrement » correspond techniquement à un angle de $\pi/4$ par rapport au segment [épaule = point initial du bras (point *Ce*), centre de gravité du bras-robot (point *Cg*)] (vue ¾). Les coordonnées s'obtiennent par des rotations et homothéties du point terminal du vecteur physique $Ce \rightarrow Cg$. Il y a alors deux choix possibles : on se place du côté du plus petit angle entre l'axe Oy et le vecteur $Ce \rightarrow Cg$.

Cas du shot 3) :

C'est le type de shot le plus simple : on place la caméra face au bras (x >> abscisse du point terminal), à une distance suffisante pour voir toute l'amplitude du mouvement.

Cas du shot 4) :

On se place comme pour le shot 1), à la différence près que l'on ne bouge pas la caméra qui zoome d'un facteur adéquat sur la main du bras-robot (supposée effectuer une action intéressante pour l'apprenant).

Cas du shot 5) :

On place la caméra comme pour le shot 2), mais on la déplace ensuite en translation en suivant le mouvement de la base du bras (inutile de calculer la position du centre de gravité, même si ce calcul aura été fait de toutes façons).

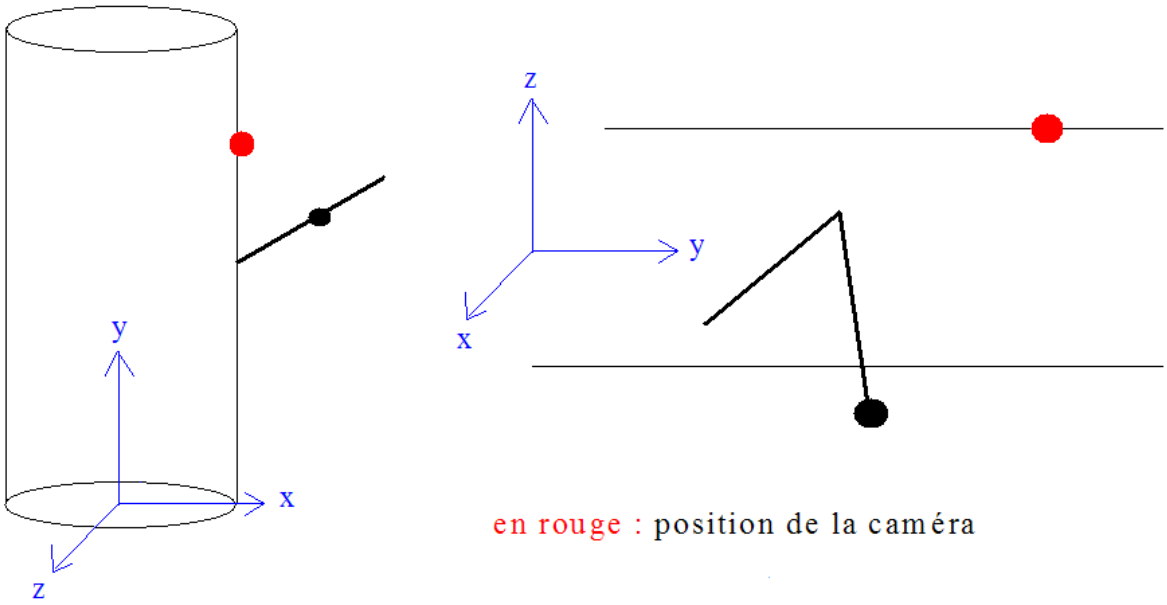


Figure 29 **Placement de la caméra pour le shot 2**

Voici le code correspondant à la recherche du pas angulaire dans le cas d'un mouvement de rotation « yaw » selon l'angle φ : (voir l'intégralité dans ***CompilIdioms.cpp***)

```
// Si l'idiome devant être retranscrit est un « pan horizontal », alors :
if ( strcmp ( idiomes[i].name, "panHori") == 0 ) {
    // on determine grossierement la position de la camera par rapport au bras pour
```

```
    // connaitre le signe de l'angle, initPR étant le point initial de visée de la caméra, et
    // currPR le point de visée de la caméra en fin de mouvement :
    int sgn ;
    if ( configCam1.posX > currPR[0] ) {
        if ( currPR[1] - initPR[1] < 0 )
            // déplacement "de la droite vers la gauche", donc etant donné le
            // demi-espace dans lequel se situe la caméra, le signe est positif.
            sgn = 1 ;
        else sgn = -1 ;
    }
    else {
        if ( currPR[1] - initPR[1] < 0 ) sgn = -1 ;
        else sgn = 1 ;
    }

   // calcul de l'angle global en fonction des 3 côtés |currPR[1]-initPR[1]|, distCamPR et
   // distCamPR (triangle isocèle, on considère que la distance à la cible ne varie pas) :
    pasPhi1 = sgn*2.0*asin(fabs(currPR[1]-initPR[1])/(2.0*distCamPR));
    // on normalise pour obtenir le pas :
    pasPhi1 /= (idiomes[i].shots.back().lastInd-idiomes[i].shots[0].firstInd+1) ;
}
```

Note sur le Raccord des sous-trajectoires entre deux idiomes :

Entre deux idiomes ou shots, la position de la caméra peut être très différente car on se base sur les proximités aux obstacles ; de plus, les façons de filmer chaque morceau de trajectoire sont très différentes. On pourrait donc relier les positions des caméras entre deux séquences par des parcours obtenus avec un roadmap « single query » (ie : on construit le graphe de recherche par expansion locale sans le stocker en mémoire),

comme dans la méthode de D. Nieuwenhuisen et M. H. Overmars 107. Cela ne semble toutefois pas vraiment utile, l'astronaute étant sensé connaitre suffisamment la station pour comprendre les changements de vue, mais pourrait faire partie d'une amélioration future.

4.1.4 Approche pour trois caméras : angles de vue privilégiés

Les deux caméras qui n'ont pas été prises en compte jusqu'alors peuvent être utilisés en adoptant d'autres points de vue, en théorie légèrement moins bons que celui choisi pour la caméra principale, mais intéressants tout de même. Nous définissons donc des hiérarchies de positions de caméras pour chaque type de shot décrit précédemment :

- Shot 1) :
    1) Placement proposé précédemment.
    2) Placement 1), du côté où l'on voit le poignet plus loin que le coude.
    3) Vue ¾ au dessus du bras, d'un côté ou de l'autre (les « côtés » sont les deux demi-espaces de part et d'autre du plan formé par les trois joints principaux du bras-robot).
    4) Vue face au bras : x >> abscisse du point terminal
    5) Vue quelconque pourvu que le bras soit dans le champ.
- Shot 2) :
    1) Placement proposé précédemment
    2) « Miroir » du placement 1) : on garde l'angle de π/4 mais de l'autre côté.
    3) Vue face au bras : x >> abscisse du point terminal
    4) Vue de dessus « aérienne »
    5) Vue parallèle à l'axe Oy, d'un côté ou de l'autre.
    6) Vue quelconque pourvu que le bras soit dans le champ.
- Shot 3) :
    1) Placement proposé précédemment.
    2) Vue de dessus (aérienne)
    3) Vue ¾ au dessus du bras, d'un côté ou de l'autre.
    4) Vue quelconque pourvu que le bras soit dans le champ.

- Shot 4) :
    1) Placement proposé précédemment.
    2) Vue quelconque pourvu que la main soit visible clairement par un zoom.
- Shot 5) :
    1) Placement proposé précédemment.
    2) « Miroir » du placement 1) : on garde l'angle de π/4 mais de l'autre côté.
    3) Vue orthogonale au plan du bras, d'un côté ou de l'autre.
    4) Vue de dessus (aérienne).
    5) Vue face au bras : x >> abscisse du point terminal
    6) Vue quelconque pourvu que le bras soit dans le champ.

Ensuite, on peut raffiner l'algorithme en cherchant le point de vue parmi ceux-là qui donne la meilleure visibilité du bras pour la caméra centrale, et pour la caméra de gauche le second meilleur point de vue, la caméra de droite obtenant la moins bonne des trois meilleures visibilités (on suppose un sens de lecture de l'écran naturel de gauche à droite, d'où ce choix). Toutes ces vues sont implémentées dans ***View.cpp***.

4.1.5 Tri des vues associées à un idiome

L'ordre des vues annoncé au paragraphe précédent ne peut pas toujours être satisfait, dans le cas où des vues théoriquement plus mauvaises ont une meilleure visibilité (en terme de surface visible du bras-robot) que d'autres en théorie meilleures. Nous allons donc réorganiser les vues associées à un idiome, afin de donner en premier celle qui a la meilleure visibilité etc. La méthode consiste à trier ces vues selon la moyenne d'un certain nombre d'évaluations d'images le long du parcours de la caméra.

Il y a trois catégories d'idiomes à considérer pour l'évaluation, regroupées en fonction de l'amplitude du mouvement du bras :

- Shot *static* ou *go-by* : on évalue une seule image avec l'unique placement de la caméra. On choisit d'évaluer la première image de l'idiome.

- N'importe quel shot sauf *track* : on évalue les images initiale et finale, ainsi que l'image au milieu du mouvement. En effectuant la moyenne de ces trois valeurs on obtient une bonne approximation de la qualité de la vue.
- Shot *track* : c'est le cas le plus complexe avec le plus de distance parcourue par le bras. On évalue $n > 3$ images choisies au hasard sur le parcours de la caméra, avec n d'autant plus grand que le parcours est long.

Une fois toutes les vues associées à un idiome évaluées avec la méthode précédente, il suffit de trier le vecteur de vues pour avoir les meilleures vues en premier. Voir le code dans le fichier ***EvalIdiomView.cpp***.

4.2 Trajectoire imprévisible : résolution de contraintes

Ne pouvant cette fois prédécouper la trajectoire, on va le faire en temps réel au fur et à mesure que l'astronaute déplace le bras. On utilise la méthode décrite en 3.2 (44), obtenant des portions de trajectoire sur lesquelles le bras-robot est environ à la même distance des obstacles, et des séquences durant lesquelles le bras a un mouvement monotone selon toutes les dimensions (pas de relations entre ces deux découpages a priori). On étudie le cas de trois caméras simultanées qu'à la fin, utilisant ce qui aura été dit avant.

4.2.1 Positionnement initial de la caméra par rapport au bras

On applique ici l'idée donnée dans le paragraphe 2.2.3 (36), en résolvant des contraintes pas à pas sur la position de la caméra. Voici l'algorithme utilisé pour initialiser la position de la caméra :

1) Calcul du centre de gravité du bras-robot.
2) Détermination du plan formé par les trois points (épaule, coude, poignet) formant les joints principaux du bras. Ce plan coupe l'horizontale locale de la station, qui est le plan tangent à la base du bras sur le cylindre central. Ces deux plans définissent deux zones de l'espace à la surface de la station.
3) On choisit la zone définie par les deux plans là où l'angle entre les deux est le plus grand ($\geq \pi/2$), et on s'intéresse à la demi-doite passant par le centre de gravité précédemment calculé, orthogonale au plan des joints du bras-robot.

4) La caméra devra se positionner sur cette derniere demi-droite, à une distance calculée de façon à voir l'intégralité du bras, ou seulement les parties qui sont en mouvement, ainsi que d'éventuels repères visuels. On ajuste ensuite la position de la caméra localement pour corriger la vue au cas où celle-ci serait trop mauvaise, de manière à conserver la distance à la cible le plus possible. Cet ajustement est nécessaire pour tenter d'éviter l'obstruction de la caméra.

Cette vue semble en effet donner le plus d'informations possible sur les mouvements du bras-robot, lorsqu'elle n'est pas obstruée bien sûr. Voir la pour une illustration de tout cela. Dans le cas où cette vue serait mauvaise, alors on va utiliser la même technique que lors du filmage des shots du paragraphe 4.1.1 (63) : on choisit en second choix la vue en miroir du côté du plus petit angle avec l'horizontale locale de la station, puis les vues parallèles à l'axe Oy, puis une vue ¾ au dessus du bras d'un côté ou de l'autre, puis une vue « aérienne » au dessus du bras, et enfin une vue aléatoire pourvu que le bras soit clairement visible.

Ensuite, on ajuste la position de la caméra image par image, en tenant compte des informations obtenues lors du découpage effectué en parallèle (proximité aux obstacles, mouvements monotones), et en recherchant avant tout un filmage cohérent tout en maintenant une vue claire par tatonnements successifs au voisinage de la position actuelle de la caméra. Voir ***PlaceCamera.cpp*** pour le code de tout cela, que nous détaillons quelque peu dans la suite.

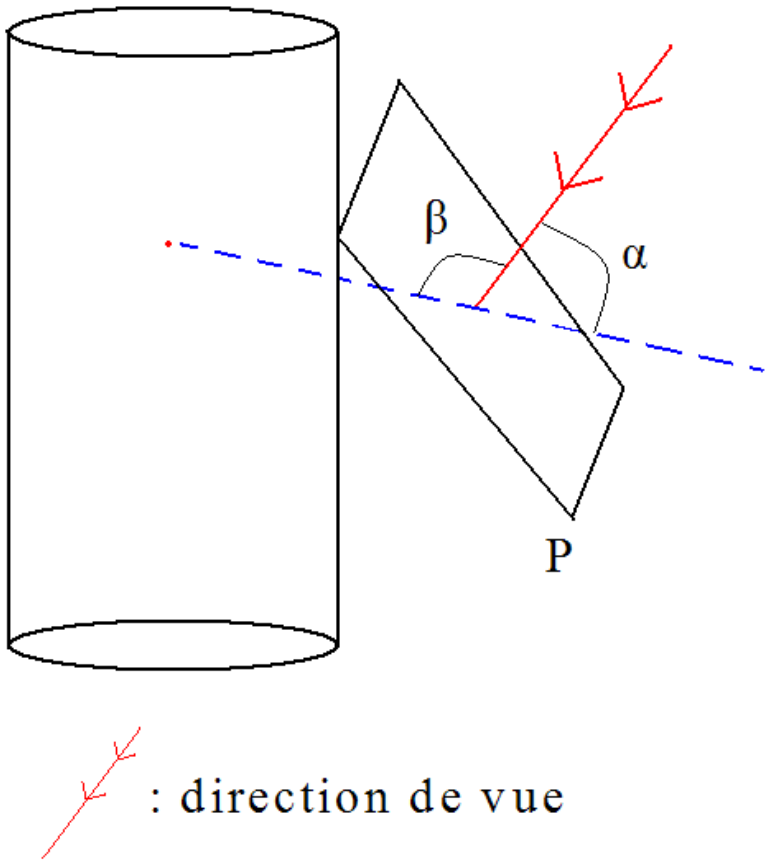


Figure 30 **Méthode d'initialisation**

4.2.2 Prise en compte du découpage de la trajectoire

On ne peut pas vraiment utiliser les points de coupure comme lors du filmage selon les idiomes, car on filme de manière continue en changeant d'angle que lorsque la vue devient trop obstruée. Cependant, les points de coupure fonctions des distances aux obstacles peuvent nous servir de checkpoints pour vérifier si l'on ne peut pas récupérer une meilleure vue dans la hiérarchie, ou simplement exiger une meilleure visibilité que si l'on était loin entre deux points de coupure (cela permet de garder plus longtemps une vue acceptable, et donc de maintenir la cohérence du film).

Ces opérations sont effectuées dans la fonction ***adjustCamPos****(int indexCam)*, qui est appelée à chaque calcul d'une nouvelle configuration de caméra.

Voici le code correspondant à la recherche d'une meilleure visibilité au voisinage d'un point de coupure :

```
// plus on est proche d'un point de coupure selon les proximités aux obstacles, et plus on
// va exiger une bonne vue de la camera :
int i=0 ;
double dist2CutInv=0.0 ; // "presque" l'inverse de la distance au point de coupure le plus proche
if (!decoupageO.empty()) {
        // on recherche l'indice i tel que | decoupageO[i]-indiceG | soit minimal,
        // indiceG désignant l'indice courant de parcours des configurations.
        while (i<(int)decoupageO.size() && decoupageO[i]<indiceG) i++ ;
        if (i==0) {
                dist2CutInv=1.0/(mini(indiceG,decoupageO[0]-indiceG)+2.1) ;
        }
        else if (i<(int)decoupageO.size()) {
                dist2CutInv=1.0/(mini(indiceG-decoupageO[i-1],decoupageO[i]-indiceG)+2.1) ;
        }
}

// Ainsi, plus cette distance inverse est grande (et donc plus on est proche d'un point de
// coupure), et plus on exige une bonne visibilité :
double visibilite=(*cam).getVisibilite() ;
if (visibilite<SEUIL_VIS_ADJUST+dist2CutInv) {
        initCamPos(indexCam) ;
        return;
}
```

4.2.3 Ajustement de la position de la caméra

Cet ajustement consiste simplement à faire effectuer de petits déplacements élémentaires à la caméra dans huit directions (le pas est choisi en fonction des caractéristiques de la scène), tangentes à la sphère de centre le point de référence sur le bras-robot (le centre de gravité ou la main dans notre application) et de rayon déterminé de façon à voir l'intégralité du bras-robot (ou de la main). On conserve alors la position de la caméra qui a donné la meilleure visibilité en ajustant la distance au point de référence, ce qui fait que la caméra ne suit pas exactement les déplacements du centre de gravité (d'où la nécessité du lissage effectué en 4.2.6 (79), quand la trajectoire est connue à l'avance). Afin d'effectuer une recherche non aveugle, il faudrait avoir à notre disposition les caractéristiques locales de la station, puis les interpréter pour guider la caméra. Cela pourrait faire l'objet d'une autre étude, et est hors de portée de ce travail.

Voici un extrait du code réalisant cela dans une direction :

```
// translation d'un pas élémentaire :
(*cam).posX+=elemStep*coeffsY[0] ;
(*cam).posY+=elemStep*coeffsY[1] ;
(*cam).posZ+=elemStep*coeffsY[2] ;
// obtention du degré de vsibilité entre 0 et 1 :
newVis = getVisibilite(indexCam) ;
// si ce dernier est meilleur que tout ce qu'on a testé jusqu'alors, on mémorise la
// direction pour déplacer la caméra avant de sortir de la fonction :
if (newVis > visibilite) {
        visibilite = newVis ;
        memStep[0]=0 ;
        memStep[1]=1 ;
}
```

4.2.4 Passage proche d'un obstacle

4.2.4.1 Réglage du zoom

Nous avons vu en 3.2.2 (46) comment juger de la proximité aux obstacles : il suffit alors de définir des seuils sur la distance aux obstacles, palliers progressifs indiquant que la caméra doit zoomer d'un certain facteur de plus en plus important. On applique alors le zoom en conséquence dans l'algorithme du paragraphe précédent, tout en changeant le point de visée de la caméra pour pointer vers la région du bras où la collision est proche. Notons que l'on zoome aussi sur la main dans le cas où seule celle-ci se déplace, mais dans ce cas on n'a pas besoin d'utiliser des informations sur les obstacles. En revanche le point de visée sera un point au milieu d'un segment reliant le bras à un obstacle proche, dans le cas où on passe assez près d'une collision pour que la caméra doive en tenir compte. Le calcul du point de visée se fait toujours en fonction des morcellements de trajectoire effectués.

Il faut alors calculer le zoom. Nous nous sommes rendu compte qu'il est plus simple (et équivalent) de rapprocher la caméra au lieu de faire varier le zoom. C'est cette technique que l'on utilise, calculant à chaque instant la distance correcte à l'objectif de la manière suivante, les constantes TAN_ANGLE et MARGE_VUE (≥ 1) représentant respectivement la tangente de l'angle du cône de vision de la camera et le facteur multiplicatif à appliquer pour obtenir le champ de vision filmé (par exemple si l'on calcule une amplitude de 13m pour le bras-robot, on filmera en fait une amplitude de 13*MARGE_VUEm pour un meilleur rendu visuel) :

```
// si l'on doit filmer seulement la main, on utilise la constante CHAMP_POIGNET
// donnant l'amplitude en mètres devant être filmée autour du poignet.
if (handPoints[indice]) {
    distCamPR=TAN_ANGLE*(CHAMP_POIGNET*MARGE_VUE)/2.0;
}
// si l'on doit filmer une partie proche d'une collision, alors la distance sera d'autant plus
// courte que la collision est proche (distObstacle[indice] donne cette distance à l'obstacle
```

```
// le plus proche).
else if (zoomObstPoints[indice]) {
        distCamPR=TAN_ANGLE*distObstacles[indice]*MARGE_VUE;
}
// Et, finalement, lorsque l'on doit filmer le bras en entier, on se base sur l'amplitude
// maximale que celui-ci peut avoir (variable maxEcart) :
else {
        // calcul de l'ecart maximal entre deux points du bras,
        // pour connaitre l'amplitude que l'on doit filmer :
        double maxEcart=sqrt((coord3D[6]-coord3D[0])*(coord3D[6]-coord3D[0])
                                + (coord3D[7]-coord3D[1])*(coord3D[7]-coord3D[1])
                                + (coord3D[8]-coord3D[2])*(coord3D[8]-coord3D[2]));
        if (LONG_EC>maxEcart) maxEcart=LONG_EC;
        if (LONG_CP>maxEcart) maxEcart=LONG_CP;
        // il faut voir maxEcart*MARGE_VUE, au moins :
        distCamPR=TAN_ANGLE*(maxEcart*MARGE_VUE)/2.0;
}
```

4.2.4.2 Positionnement de la caméra

Le bras-robot passe proche d'un obstacle, il est donc naturel de centrer la vue de la caméra sur le milieu **O** du segment joignant les deux points P_r sur le bras-robot et P_o sur un obstacle, ces deux derniers points étant tels que la distance $\| P_r - P_o \|$ soit minimale parmi toutes les distances $\| P - Q \|$ avec P point sur le bras-robot, et Q point sur un obstacle quelconque. Ce calcul est fait à l'aide de la libraire **PQP**.

Ensuite, il faut déterminer l'angle de vue de la caméra : on sait qu'elle est sur un cercle centré en **O**, dans le plan orthogonal au segment $[P_r , P_o]$ (pour avoir le plus de détails possibles sur le passage proche de l'obstacle), mais il reste à déterminer l'angle. Pour cela, on choisit de minimiser la valeur $\boldsymbol{min}\ (\alpha , \pi - \alpha)$, α

étant l'angle aigu entre le vecteur épaule→coude et l'orthogonal du plan formé par les trois joints principaux du bras, illustré sur la figure 31, où [Pr , Po] est représenté par le segment en vert.

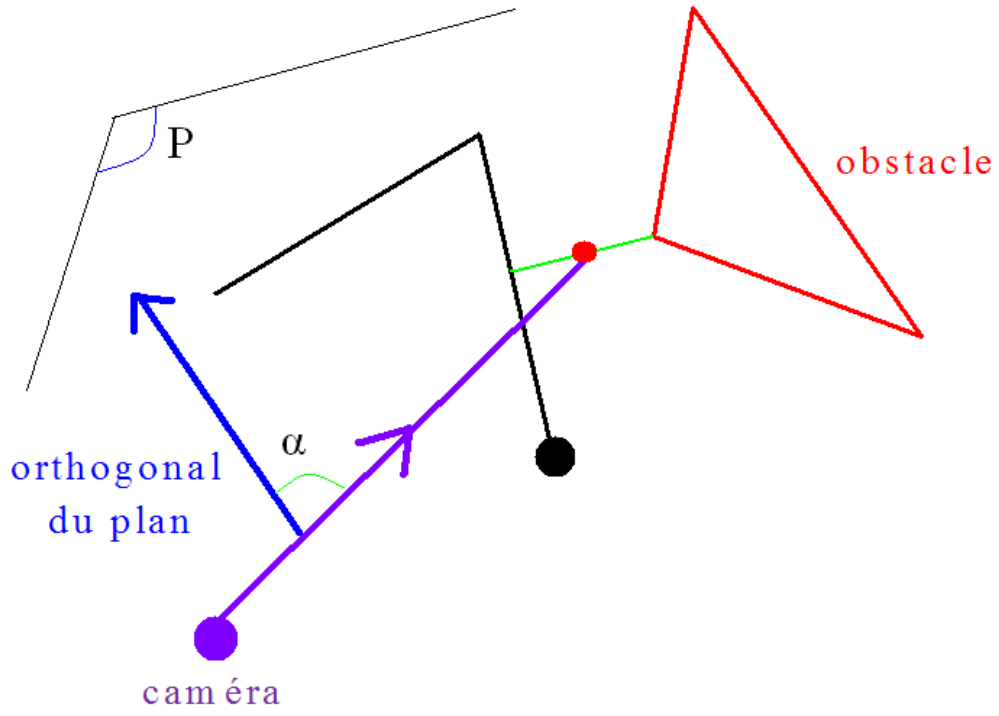


Figure 31 **Placement de la caméra lors du passage proche d'un obstacle**

Cette vue est celle qui maximise la surface de bras observée (on se place « le plus orthogonalement possible » au plan formé par le bras-robot), tout en gardant une bonne vision des événements survenant dans la zone entre le bras et l'obstacle.

4.2.5 Résolution du problème des trois caméras

On doit afficher trois écrans représentant trois vues différentes ; il semble intéressant que ces vues présentent la scène telle que les caméras correspondantes forment un triangle équilatéral approximatif en projection sur un plan. Cependant, les calculs à réaliser sont alors assez fastidieux, et on préfère placer les deux autres caméras aléatoirement sur la sphère de filmage, le plus loin possible les unes des autres (avec un

degré d'éloignement dépendant du nombre de points échantillonnés sur la sphère) ; toutefois, cette dernière situation ne survient que lorsqu'aucune des vues suivantes n'est satisfaisante et/ou disponible :

1) Vue indiquée en 4.2.1 (71) ; normalement réservée à la caméra centrale.
2) Vue en miroir de cette dernière, du côté du plus petit angle de l'orthogonal du plan avec l'horizontale locale de la station
3) Vue parallèle à l'axe Oy du côté où l'on voit le poignet plus proche que le coude
4) Vue symétrique de la précédente (le poignet plus loin que le coude sur l'axe Oy)
5) Vue ¾ au dessus du bras, d'un côté ou de l'autre
6) Vue « aérienne » au dessus du bras (mêmes coordonnées sur x et y, $z >> z_{bras}$)

Une fois ces deux autres caméras placées, on met à jour leurs positions par les mêmes méthodes déjà écrites pour la caméra principale (écran central).

Note : Tout ceci n'est utile que lorsque l'on filme le bras entier ou seulement la main, mais pas lorsque l'on se concentre ou sur une région spécifique proche d'une collision, car dans ce cas les critères de visibilité sont différents.

4.2.6 Lissage de la suite des configurations de caméras

Une fois que l'on a la suite des configurations de caméras, avant de passer au film il faut les réorganiser en une courbe assez lisse pour que le film soit moins erratique pour un observateur humain (le film serait probablement utilisable tel quel, étant donné le nombre d'images par seconde, mais on peut améliorer la qualité significativement comme on le fait ici). L'idée est de remplacer les coordonnées projetées sur les axes des configurations de caméras par les milieux des segments joignant deux valeurs de coordonnées, lorsque le déplacement n'est pas de trop grande amplitude. Dans ce dernier cas, on suppose avoir affaire à un changement radical de position de caméra (lorsque la visibilité devient trop médiocre par exemple), et donc on ne prend pas la moyenne mais on conserve la valeur calculée pour la coordonnée de la position de la caméra.

Voici un schéma visuel de l'algorithme (figure 32, code dans ***Lissage.cpp***), utilisé dans le cas d'un filmage d'une trajectoire connue. En effet, si la trajectoire est filmée en direct on n'a pas connaissance de la prochaine position de la caméra, et on ne peut donc réaliser cela :

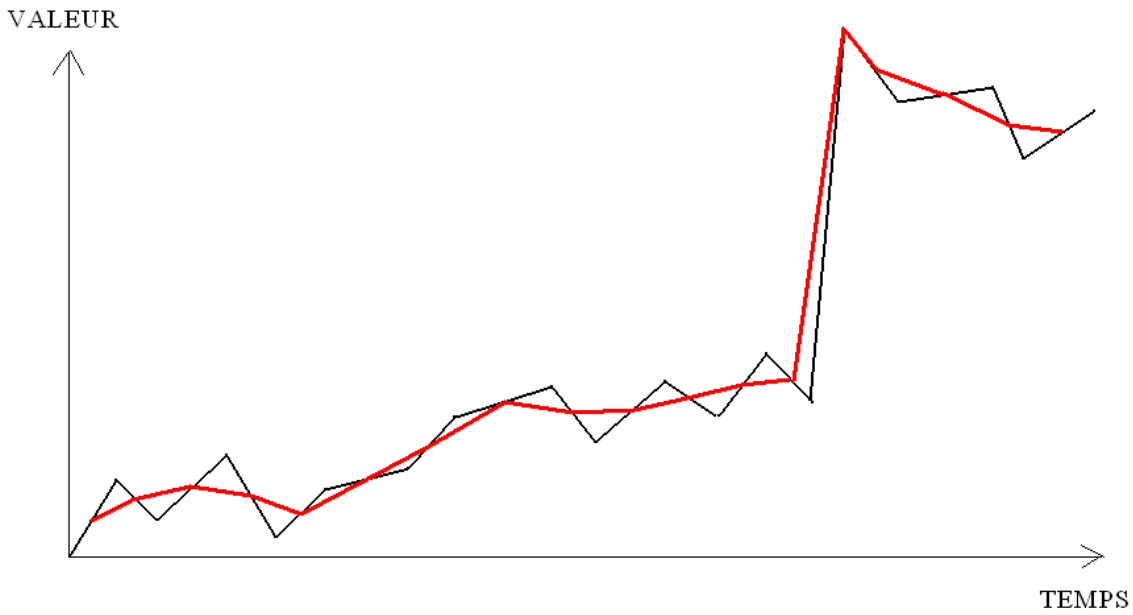


Figure 32 **Présentation visuelle de l'algorithme de lissage**

Une autre possibilité est de remplacer plusieurs valeurs de coordonnées par leur moyenne lors de petits segments monotones croissants ou décroissants. On a pas utilisé cette idée car elle entraîne des sauts encore plus importants dans le rendu final.

4.3 Caméras fixes

Dans cette partie, on évalue la qualité de l'image pour chaque caméra sur chaque nouvelle configuration du bras-robot, et on garde la meilleure caméra pour l'écran central, la seconde pour l'écran de gauche et enfin la troisième pour l'écran de droite. Pour évaluer la qualité d'une vue, on pointe simplement la caméra sur la cible et calcule ce qu'elle a dans son champ de vision. On change de caméra lorsque celle utilisée jusqu'alors a une trop mauvaise vue.

Le codage de cette partie est assez simple car les caméras ne sont plus capables de mouvements complexes, nous ne détaillerons donc pas plus cet algorithme, qui utilise des fonctions déjà présentées. La seule difficulté notable est le réglage du zoom (utile cette fois). On calcule celui-ci par la formule suivante :

$$\mathbf{zoom = dist_R / dist_M} ,$$

où $dist_R$ est la distance réelle de la caméra au point de repère, et $dist_M$ la distance à laquelle serait la caméra si celle-ci pouvait se déplacer dans l'espace.

Note : On suppose (faute de données) que chaque camera peut pivoter autant que l'on veut. Cela ne change guère la nature du filmage, car dire qu'une caméra peut pivoter autant qu'elle veut ne signifie pas qu'elle puisse mieux voir le bras si celui-ci est obstrué. Notons aussi que dans ce cas, seule une trajectoire imprévisible présente un intérêt ; on ne filmera en effet jamais une trajectoire connue avec des caméras fixes, étant donné que le but est alors de générer une animation 3D la plus informationelle possible. Il s'agit ici du mode le plus contraint.

Voir le code dans ***FilmageFixe.cpp***. Afin d'avoir une vision globale de tout cela, l'100 décrit l'utilisation des algorithmes présentés dans le système de filmage.

4.4 Génération de l'animation étant donnée la suite de configurations de caméras

Nous indiquons ici comment est générée l'animation tridimensionnelle finale que l'utilisateur pourra observer dans *RomanTutor*. Avant le présent travail, le logiciel en question ne fournissait que des vues fixes, les caméras ne bougeant pas et étant donc contraintes à regarder toujours dans les mêmes directions, aux mêmes endroits. Ainsi certaines situations étaient difficiles, voire impossible à filmer, sans compter que le film obtenu n'est pas très intéressant à regarder pour l'apprenant. Nous avons donc modifié le code responsable de la génération de l'animation.

Pour fabriquer une animation 3D, on dispose d'un objet de type *SoCamera*, lié à la bibliothèque graphique **Coin3D**. C'est en fait cet objet qui, lorsqu'on le modifie, va automatiquement donner le rendu graphique correspondant. Il suffit donc de passer les paramètres de sortie de notre programme (les configurations des caméras) à ces objets *SoCamera*, remplaçant la suite de vues fixes codée dans *RomanTutor*. En effet, *RomanTutor* utilise déjà **Coin3D** pour les animations, on se contente doonc de changer les appels de méthodes de cette librairie dans la classe principale du programme.

Voici un extrait du code rélisant cela, pour une caméra, à chaque image :

```
// Réglage de la position :
mpkGUI::rootCamera2-> position = SbVec3f (configsCam[1][curr_animstep].posX, configsCam[1]
[curr_animstep].posY, configsCam[1][curr_animstep].posZ);
// Réglage de l'orientation et du roll :
mpkGUI::rootCamera2->pointAt(SbVec3f(configsCam[1][curr_animstep].posCx,
                configsCam[1][curr_animstep].posCy,
        configsCam[1][curr_animstep].posCz),
                                SbVec3f(0,-1,0));
```

Chapitre 5

Expérimentations et Évaluation

Dans une première partie on teste le programme sur des trajectoires artificiellement générées pour avoir une idée claire de la façon dont les algorithmes se comportent. Ensuite on effectue le test « grandeur nature » avec le logiciel *RomanTutor*.

5.1 Tests sur trajectoires artificielles

Dans un premier temps on étudie les réponses du programme sur une trajectoire connue, puis imprévisible ensuite. Nous reprenons la trajectoire déjà présentée comme test au paragraphe 3.2.5 (52), en ajoutant des déplacement en rotation « yaw » de l'épaule, ainsi que des valeur d'angle « roll » non nulles, pour rendre le tout plus complet et intéressant. La trajectoire modifiée se trouve en annexe C (104).

5.1.1 Trajectoire connue

Deux modes de filmage sont implémentés dans ce cas. Nous les testons séparément.

5.1.1.1 Idiomes

La trajectoire est conçue pour que l'on reconnaisse et filme tous les types d'idiomes enregistrés. En effet, les tests indiquent que le programme « voit » successivement les shots *pan vertical*, *track*, *pan horizontal*, et *static*. La caméra est alors positionnée en conséquence, comme indiqué en 4.1.3. La succession de configurations de caméra ne présente alors aucun intérêt seule, mais devra être accompagnée d'une animation 3D. Voir le paragraphe correspondant dans les tests avec Roman Tutor.

5.1.1.2 Résolution de contraintes

Le graphique de la figure 33 présente les résultats obtenus, c'est-à-dire l'évolution des positions de deux caméras (celle de gauche et celle du centre) dans l'espace lorsque la configuration du bras-robot varie, en projection sur le plan yOz. La projection sur le plan xOz donne en effet le genre de graphique de la figure 34, inexploitable : c'est normal car les mouvements dans ce plan varient peu, et il y a 1500 points évoluant normalement en trois dimensions. Les angles des caméras ne sont pas pertinents, car on sait que celles-ci visent toujours le bras-robot. On obtient bien une séparation des trajectoires, et celles-ci sont cohérentes.

Remarque : Le graphe devrait être en 3D car il dépend de la configuration courante, mais comme les deux caméras se déplacent inexorablement parallèlement au rail de la station, leur coordonnées **y** varient plus ou moins similairement, ce qui permet à Maple de relier les points pour présenter la trajectoire, en deux dimensions.

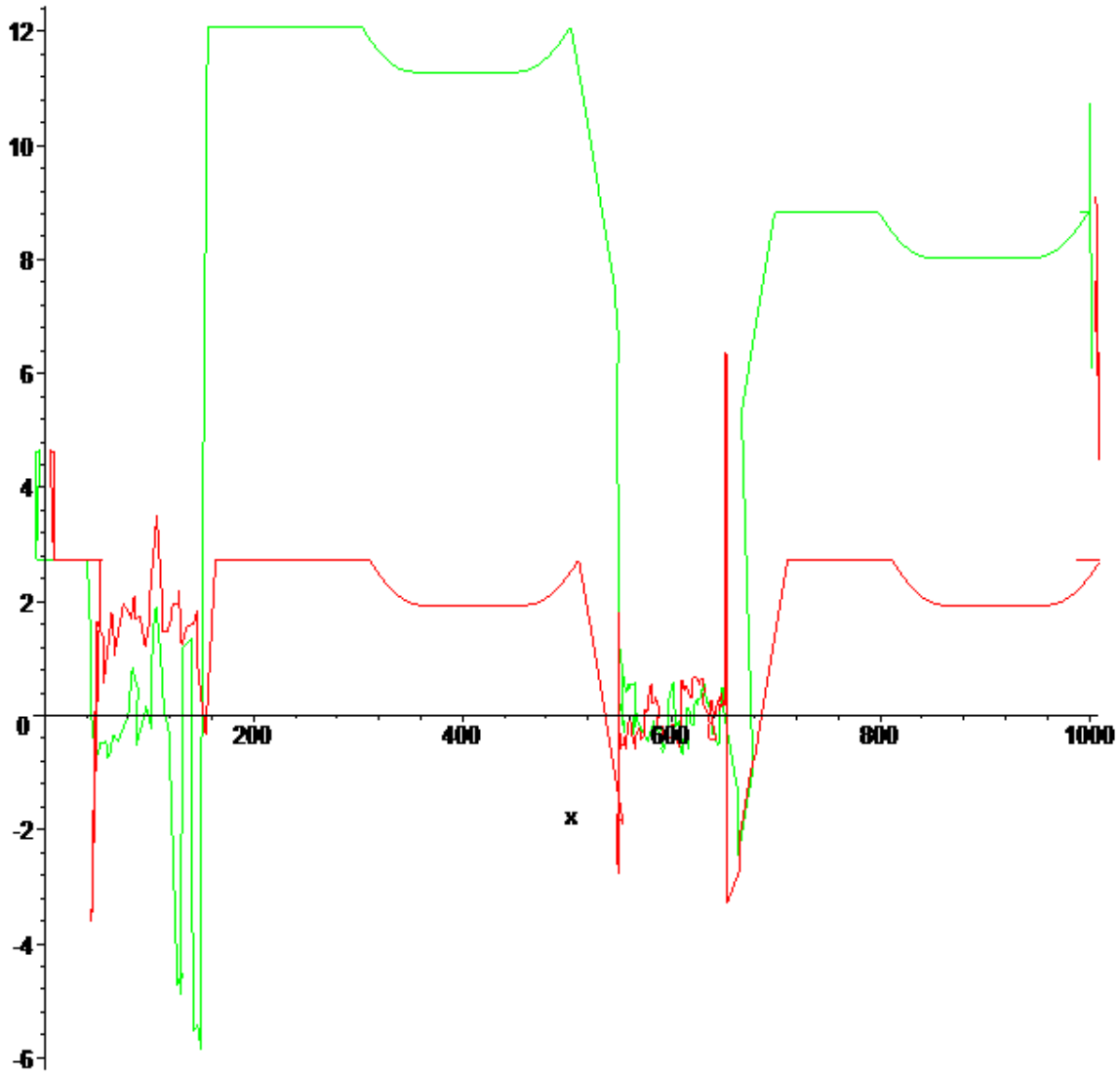


Figure 33 **Trajectoires « connues » de deux caméras dans le plan yOz**

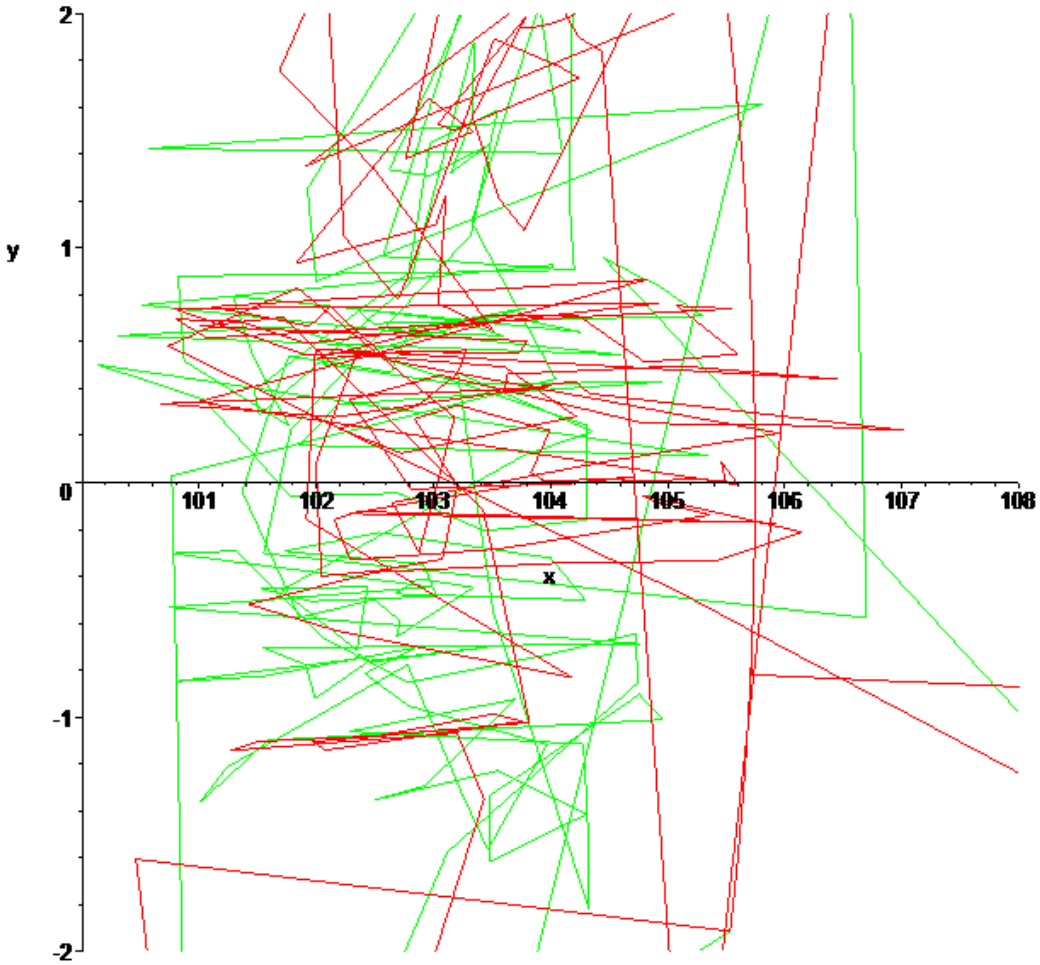


Figure 34 **Trajectoire quasi-chaotique dans le plan xOz**

5.1.2 Trajectoire imprévisible

On observe la suite des positions des caméras générées dans le cas d'un filmage en direct, avec la même trajectoire qu'au paragraphe précédent. La figure 35 montre les courbes obtenues en projection sur le plan yOz. Par rapport au filmage de la trajectoire connue, on remarque une différence assez marquée entre les configurations 40 et 160 environ, les positions des caméras étant moins bien discriminées. C'est normal d'observer cela, car les calculs des distances étant moins précis, la caméra a plus tendance à « hésiter » entre

deux placements au fur et à mesure des configurations du bras. De plus, on ne lisse pas le vecteur final des configurations de caméras dans ce cas.

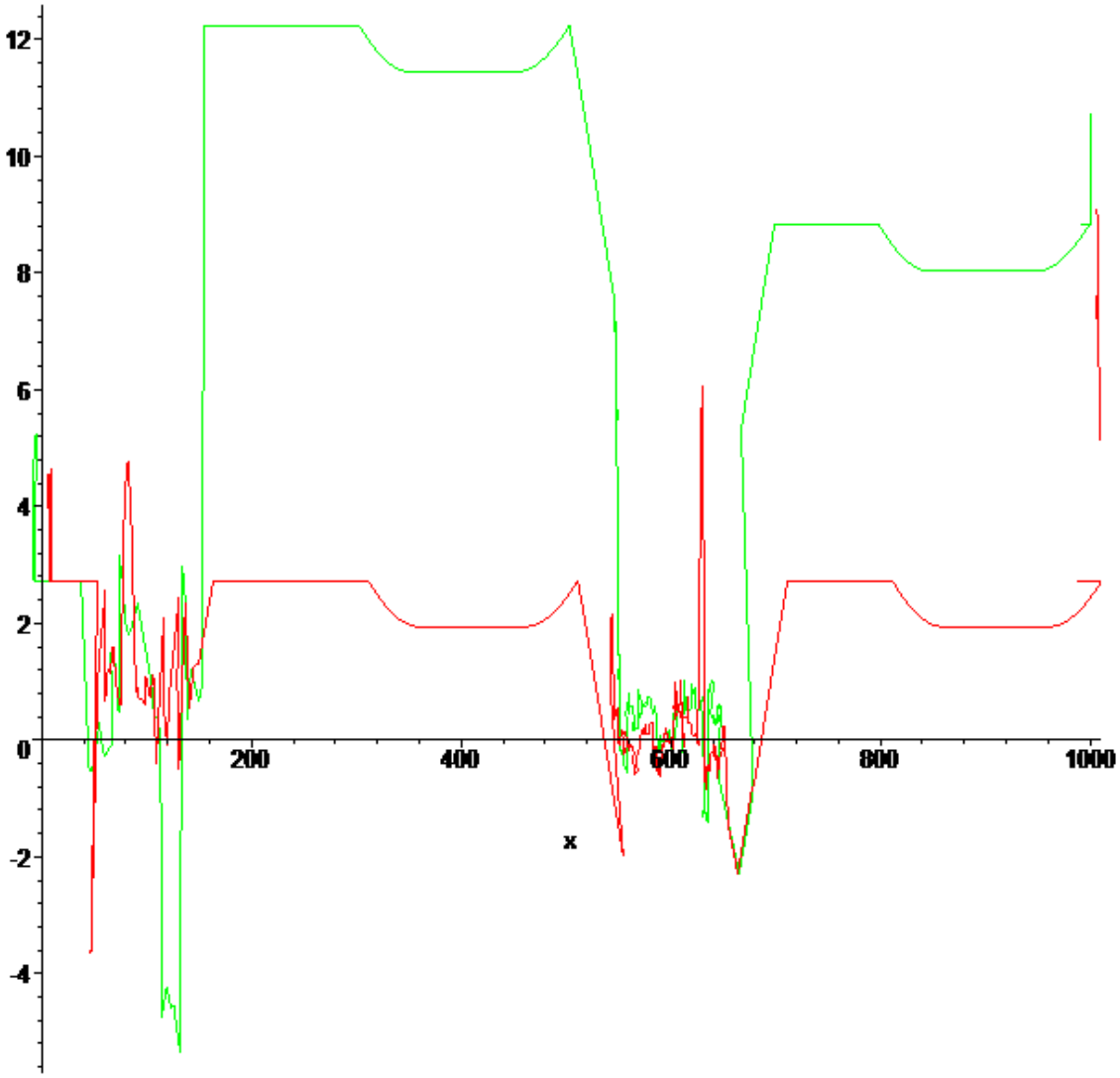


Figure 35 **Trajectoires « imprévisibles » de deux caméras dans le plan yOz**

5.2 Intégration dans *RomanTutor*

On utilise le programme décrit précédemment au sein de *RomanTutor*, afin de pouvoir observer l'animation générée. L'entrée du programme est la sortie du planificateur interne à *RomanTutor*, et la sortie est interprétable par *RomanTutor* via la librairie **Coin3D** en terme de génération d'images comme indiqué au paragraphe 4.4 (81). Il n'y a donc pas de gros problèmes d'intégration.

5.2.1 Trajectoire connue

Nous utiliserons ici à des fins de tests la trajectoire indiquée en rouge sur la figure 36 (générée par le planificateur pour des points initiaux et finaux particuliers), car elle comporte plusieurs éléments intéressants. En effet, elle commence par un mouvement vertical du bras pour le redresser, puis se déplace le long du rail tout en effectuant une courbe qui s'oriente peu à peu dans la direction de l'axe Ox.

5.2.1.1 Idiomes

On note que la détection des types d'idiomes n'est pas facile car le planificateur effectue des mouvements sur tous les joints en même temps pour optimiser la durée du parcours. En réglant les paramètres de manière adéquate on arrive cependant à forcer le programme à trouver les bons idiomes, puis on observe que les caméras se déplacent comme prévu, générant un effet visuel correct, à condition de bien régler le paramètre d'échantillonnage pour tester les shots de type *track*. La figure 37 présente une vue d'un shot pan vertical à côté du bras, filmant un mouvement vertical de ce dernier.

Temps nécessaire à l'algorithme : quelques secondes pour une trajectoire de plusieurs centaines de points, donc le temps d'exécution est négligeable.

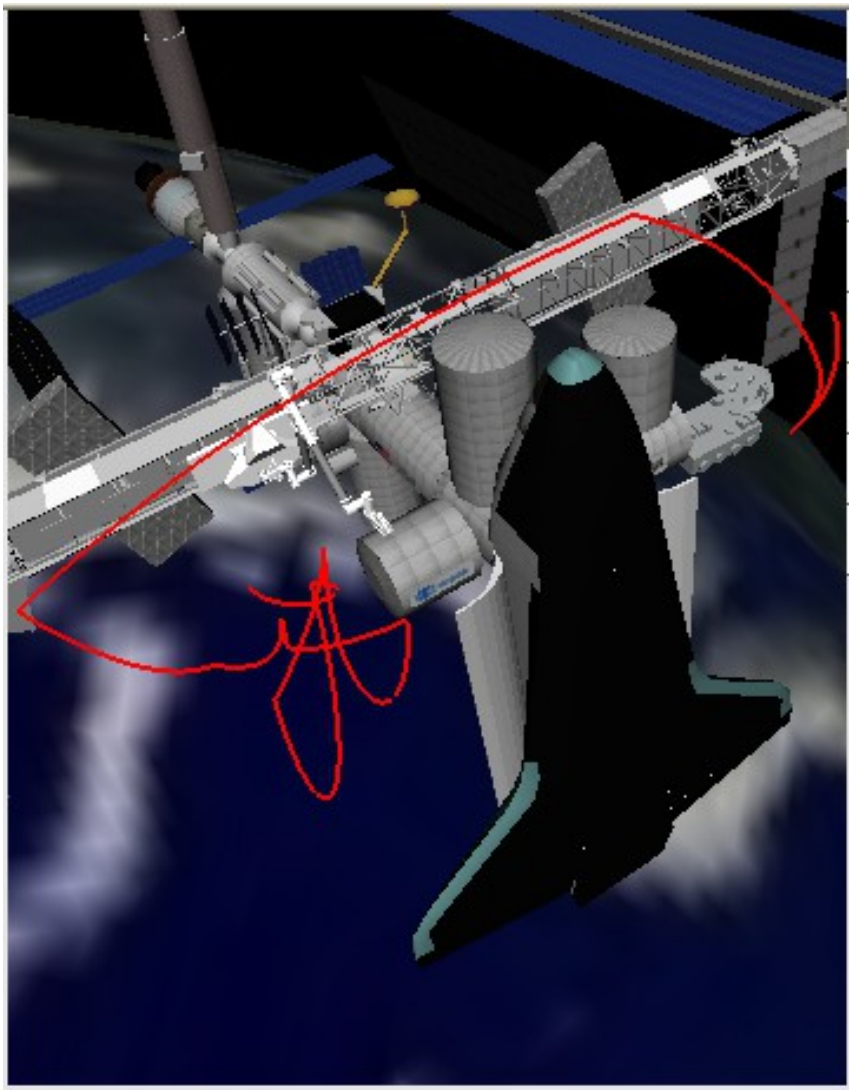


Figure 36 **Trajectoire du bras-robot pour le test dans *RomanTutor***

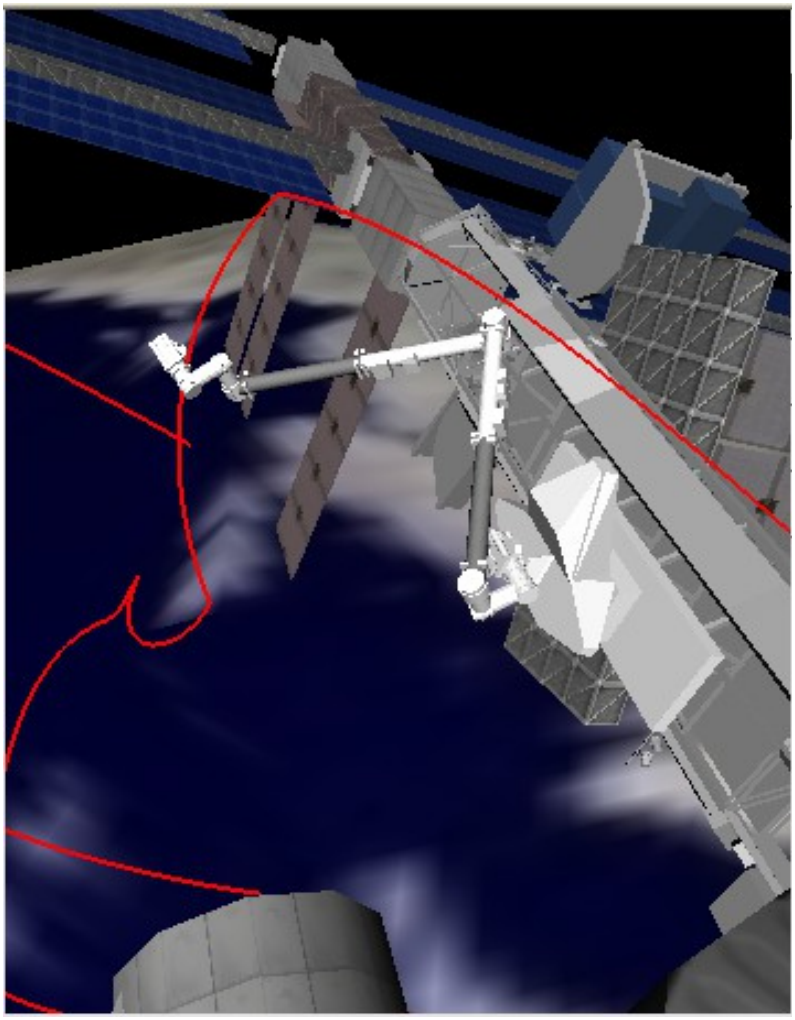


Figure 37 **Shot *pan vertical***

5.2.1.2 Résolution de contraintes

Les tests indiquent que le filmage obtenu par résolution de contraintes visuelles donne le même rendu que le filmage via les idiomes, avec une visibilité améliorée toutefois car on la teste cette fois à chaque image. L'unique défaut de cette approche semble être le temps nécessaire au calcul lorsque l'on utilise l'approche via **PQP** présentée au paragraphe 3.4.1 (57) ; cependant, en utilisant la méthode avec **Coin3D** du paragraphe 3.4.2 (59), on accélère nettement les temps de calcul pour rivaliser avec la méthode découpant la trajectoire en idiomes.

Temps nécessaire à l'algorithme : quelques secondes pour une trajectoire de plusieurs centaines de points, comme pour les idiomes, à condition d'utiliser la méthode avec **Coin3D** pour calculer la qualité d'une image. Sinon, il faut une dizaine de minutes pour tout calculer avec 900 points.

5.2.2 Trajectoire imprévisible

Nous reprenons la trajectoire de la figure 36, que l'on donne point par point au planificateur de caméra afin de simuler un filmage en direct.

5.2.2.1 Résolution de contraintes

Les résultats sont très proches de ceux obtenus dans le cas où la trajectoire est connue, pour des temps de calcul identiques, ce qui est encourageant car cela signifie que l'on ne perd que très peu de qualité lorsque l'on doit filmer la trajectoire du bras en direct. La figure 38 présente une vue aérienne du bras obtenue pendant quelques secondes avec ce type de filmage.

5.2.2.2 Caméras fixes

Finalement, dans un souci de complétude, nous testons aussi la viabilité de notre algorithme lorsque les caméras sont fixées à la station, toujours sur la même trajectoire. On obtient des résultats très satisfaisants par rapport à la trajectoire que l'on a testé, mais il faut garder à l'esprit que certaines trajectoires ne peuvent pas être filmées par les caméras fixes ; d'où la notion de degré de désirabilité des zones de la station, afin de guider l'utilisateur vers les endroits où l'on pourra filmer facilement (voir 107). La figure 39 présente finalement la vue d'une des caméras fixes, lorsque le bras s'éloigne en translation le long du rail.

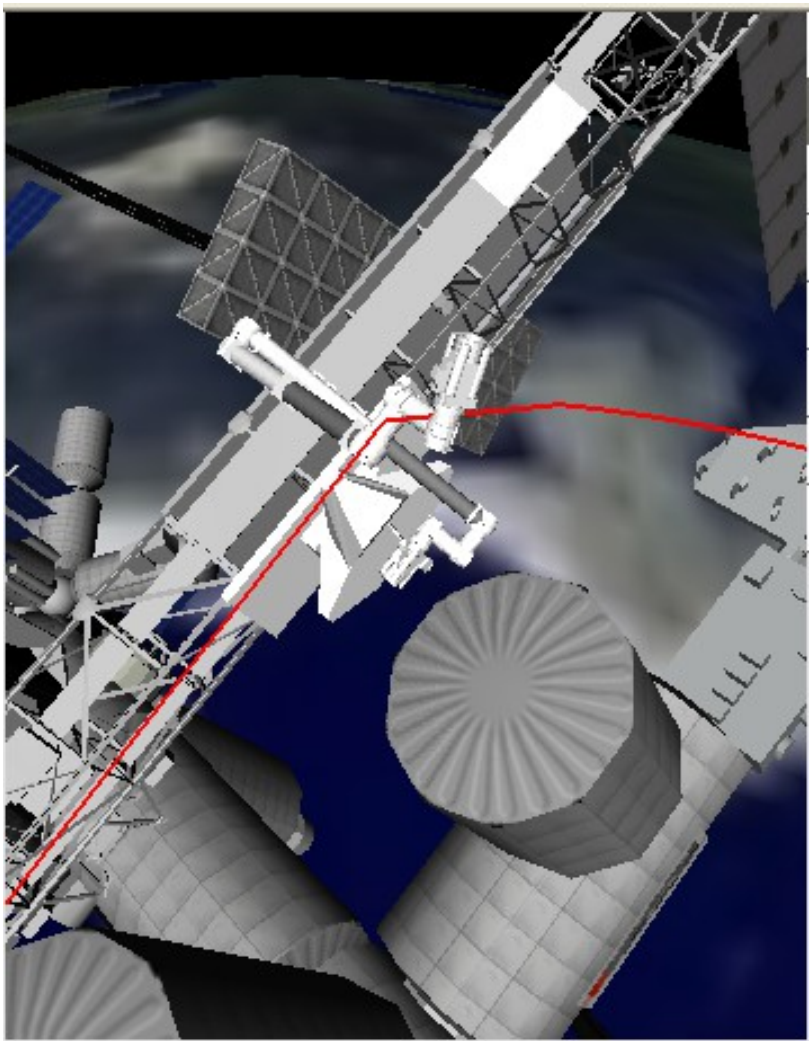


Figure 38 **Vue aérienne au dessus du bras-robot**

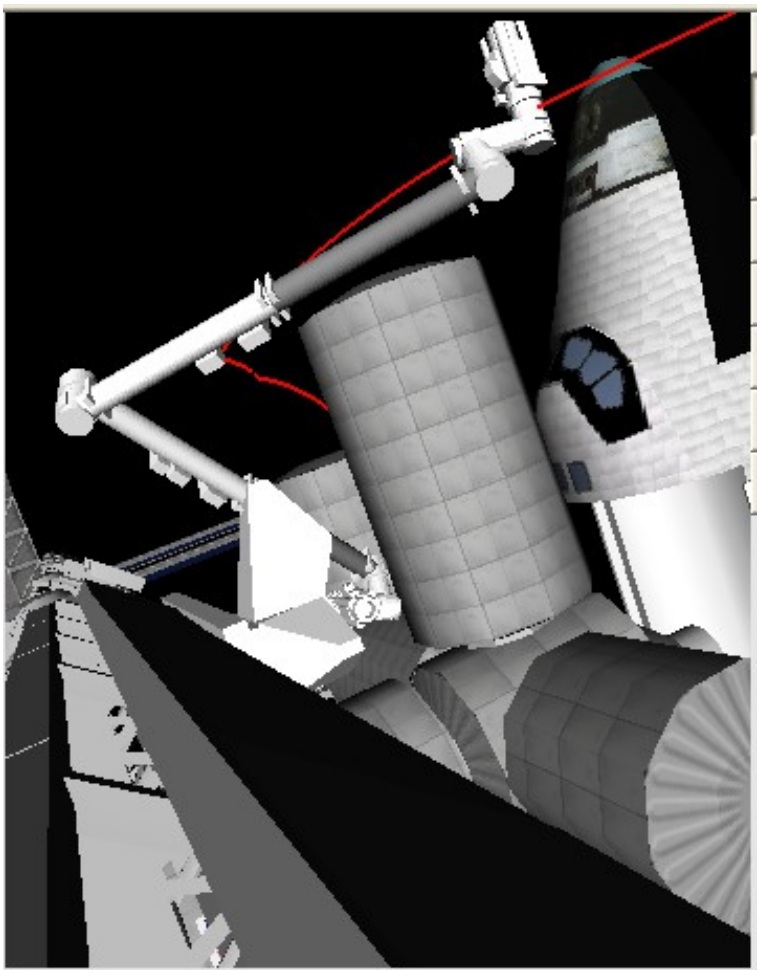


Figure 39 **Bras s'éloignant d'une caméra fixe**

Bilan :

Nous obtenons sensiblement les mêmes genres de film avec les approches idiomes et contraintes. On note donc que l'utilisation d'heuristique sur l'évaluation du trajet d'une caméra (en terme de visibilité du bras) est assez efficace dans le cas des idiomes, réduisant le temps de calcul par rapport à l'autre méthode. La qualité du film obtenu est assez bonne pour donner des indications pertinentes à l'utilisateur, bien qu'elle puisse très probablement être améliorée par le biais d'autres techniques. Par exemple, l'utilisation du planificateur TL-Plan (planificateur basé sur la logique temporelle, renvoyant une suite d'actions effectuées sur des états pour parvenir à un but depuis un état initial) est en cours d'étude, afin de déterminer des idiomes mieux adaptés.

Conclusion

Contributions

Le présent mémoire contribue au développement de *RomanTutor* et par là-même à l'apprentissage assisté par ordinateur en général, pour son côté génération d'animation explicatives. Par son côté cinématographie, ce travail constitue aussi une certaine avancée dans le monde du filmage automatique, en prenant des caractéristiques de quelques approches existantes, et en comparant deux types de filmage radicalement différents : résolution de contraintes au niveau de l'image, et idiomes au niveau des déplacements élémentaires des caméras. C'est d'ailleurs à notre connaissance la première tentative de comparaison des deux approches.

Critique du travail

Les contraintes de temps ayant beaucoup pesé sur l'achèvement de ce mémoire (pour cause d'un doctorat débutant en janvier 2008), certaines parties auraient mérité un développement plus poussé. En particulier, l'approche basée sur les idiomes utilisant les zones significatives pour l'apprenant n'a pu être implémentée dans les délais, et la compilation des idiomes existante pourrait être améliorée en raffinant les positions des caméras. Il faudrait aussi poursuivre la construction de la base d'idiomes en introduisant des idiomes en plusieurs shots, se rapprochant du vrai concept cinématographique. De plus, le code n'est que très peu optimisé. Plusieurs fonctionnalités sont néanmoins achevées et opérationnelles au sein de *RomanTutor*.

Futurs travaux de recherche

Nous commencerons par poursuivre le développement du programme afin de corriger entre autres les points mentionnés dans le paragraphe précédent et tout le long de ce mémoire, avant de passer à une phase de généralisation : tout ce que l'on a utilisé là pour le bras-robot canadien doit pouvoir être applicable à un autre robot mobile, moyennant quelques modifications. Un autre axe de continuation du travail est aussi le raffinement du procédé de filmage. En effet, on peut imaginer filmer une scène non seulement en fonction de l'objet filmé, mais aussi de son entourage, en recherchant des vues dépendant des positions relatives des objets – éventuellement mobiles, ce qui n'est pas le cas avec Canadarm2 –, de leurs formes et interactions.

Perspective

Le domaine de la génération automatique d'animation 3D est toujours très actif au moins grâce aux jeux vidéos qui comportent des intelligences artificielles et graphismes de plus en plus évolués. Ceux-ci ont donc de plus en plus besoin de planifier optimalement les déplacements des caméras, car celles-ci constituent l'interface indispensable entre le moteur de jeu et les perceptions humaines : la façon dont on filme dans un jeu détermine en grande partie le succès de celui-ci. Et, en dehors du domaine des jeux, la génération automatique d'animations trouve toujours sa place dans tous les systèmes interactifs d'apprentissage, sujet qui est loin d'être totalement exploré.

Annexe A

Lissage d'un ensemble de points par densité

Cet algorithme intervient lorsque l'on a obtenu une série de valeurs booléennes bruitée, c'est-à-dire que la plupart des valeurs sont correctes mais quelqus-unes sont fausses et interféreraient avec un algorithme non prévu pour les supporter.

On prend donc en entrée $\mathbf{u=(u_1, \ldots, u_n)}$, vecteur booléen, et l'on renvoit en sortie $\mathbf{v=(v_1, \ldots, v_n)}$, vecteur booléen tel que toutes les sous-suites de ce vecteur ne contenant que des valeurs **true** soient de taille plus grande qu'un nombre fixé (le nombre d'image par seconde, éventuellement multiplié par un certain facteur dans le cas qui nous intéresse).

L'idée est simple : on parcourt linéairement le vecteur **u**, supprimant les ilots de points isolés, et connectant les groupes trop petits (par rapport au nombre d'images par seconde) assez proche les uns des autres pour ne pas être supprimés (fusion). On réitère l'opération autant de fois que nécessaire pour ne plus avoir de petits groupes à supprimer.
Il reste donc à définir deux choses :

- Sur quel critère supprime-t-on un groupe de valeurs **true** consécutives ?
- Sur quel critère fusionne-t-on deux tels groupes consécutifs ?

La réponse à la seconde est en fait le complémentaire de la première réponse : on supprime un groupe $\mathbf{(u_i, \ldots, u_j)}$ si et seulement si toutes les valeurs $\mathbf{u_{i-k}}$ et $\mathbf{u_{j+k}}$ sont à **false**, avec k variant de 1 à j-i+2 (la valeur 2 est choisie de façon assez subjective mais semble donner de bons résultats). Ainsi, on relie deux groupes si le critère précédent n'est pas rempli par un des éléments frontaliers d'un des groupes.

Plus précisément, le pseudo-code est décrit dans l'algorithme 5 ci-dessous.

```
Tant qu'il existe un groupe de taille inférieure à la taille minimale, faire :
Tant que l'on supprime un groupe, faire :
                nouveau_groupe ← false ;
            Pour i allant de 1 à n faire :
                si u[i] = true alors :
                    si ( ! nouveau_groupe ) {
                        indice_gauche_groupe ← i ;
                        nouveau_groupe ← true ;
                    }
                sinon :
                    si ( nouveau_groupe ) {
                        indice_droite_groupe ← i-1 ;
                        Tester_critère_suppression ;
                        Supprimer ou non ;
                    }
            Fin faire ;
        Fin faire ;
        Fusion des groupes survivant selon le critère opposé ;
Fin faire ;
```

Algorithme 5 **Groupement des points par densité**

Observons l'exécution de l'algorithme sur le vecteur suivant, avec une taille minimale de 10 pour les groupes :

(00000**1**0**1**00**1**00**1111**00**1**0**1**000**11**000**111**00**1**0**1**000**1111**00000) (étape 0)

Après premier passage dans la boucle détectant les segments à supprimer (pour i allant de 1 à n ..), on obtient :

(00000**1**0**1**0000**1111**00**1**0**1**0000000**111**00**1**0**1**000**1111**00000)

Il n'y a plus rien à supprimer alors, et on passe donc à l'étape de fusion des segments adjacents :

(00000**111**0000**111111111**0000000**111111111111111**00000)

On effectue ensuite le second passage dans la boucle de suppression :

(00000000000000**111111111**0000000**111111111111111**00000)

Puis on fusionne enfin :

(00000000000000**111111111111111111111111111111111**00000)

Remarquons que ce dernier résultat est assez logique car la densité de l'ensemble de valeurs à **1** dans le vecteur initial sur la région délimitée par les **1** dans le vecteur final est égale à 19 / (19 + 15) > 0.5.

Annexe B

Structure du programme

Le programme de filmage est subdivisé en plusieurs modules, dont celui que l'on vient de détailler en 97 (*GroupDensite.cpp*), celui responsable du découpage de la trajectoire (*Decoupxxx.cpp*), de la transformation de la trajectoire en idiomes (*TransfIdiomes.cpp*) etc. Pour y voir plus clair, voici les principaux scénarios d'exécution pouvant survenir :

B.1 Méthode basée sur la résolution de contraintes image par image

Nous commençons par initialiser les variables globales (coordonnées 3D du bras-robot **coord3D**, point de visée **currPR**, distance adéquate d'une caméra à la cible **distCamPR**) via les fonctions suivantes :

```
// récupération des coordonnées 3D :
coord3D = configurations[0].conf2coord3D();  // fonction conf2coord3D() de
                                             // Configuration.cpp
PlaceCamera placeCam=PlaceCamera(); // l'objet servant a (dé)placer les caméras
// determination du point de repere courant :
getRepPoint(0); // fonction getRepPoint() de Functions.cpp
// calcul de la distance de vue :
getViewDistance(0); // fonction getViewDistance() de Functions.cpp
```

Puis l'on traite chaque caméra successivement :

```
// initialisation de la position de chaque caméra (milieu, puis gauche, puis droite)
```

```
placeCam.initCamPos(1); // fonction initCamPos(int) de PlaceCamera.cpp
placeCam.initCamPos(0);
placeCam.initCamPos(2);
// ajout des paramètres de caméras dans le vecteur résultat
configsCam1.push_back(placeCam.cameras[1].getConfig());
configsCam0.push_back(placeCam.cameras[0].getConfig());
configsCam2.push_back(placeCam.cameras[2].getConfig());
```

Enfin, on boucle sur chaque configuration configurations[i] en effectuant le même travail, à la différence près que l'on déplace la caméra puis ajuste sa position au lieu de l'initialiser. Les 6 lignes de code précédentes deviennent :

```
// déplacement de chaque caméra (milieu, puis gauche, puis droite) suivant le mouvement du // point de visée (calculé dans depctRepPoint)
placeCam.moveCamera(depctRepPoint,1); // fonction moveCamera(double*,int) de
                                      // PlaceCamera.cpp
placeCam.moveCamera(depctRepPoint,0);
placeCam.moveCamera(depctRepPoint,2);
// ajustement des positions nouvellement calculées :
placeCam.adjustCamPos(1); // fonction adjustCamPos(int) de PlaceCamera.cpp
placeCam.adjustCamPos(0);
placeCam.adjustCamPos(2);
// ajout des paramètres de caméras dans le vecteur résultat : même code qu'auparavant
```

Ensuite, les différences entre les trajectoires connues ou non s'observent à ce niveau uniquement via le découpage de la trajectoire, qui se fait d'un bloc dans le premier cas, et point par point sinon.

B.2 Méthode basée sur les idiomes

Dans ce cas la trajectoire est forcément connue à l'avance. Il faut d'abord disposer d'une base d'idiomes afin d'effectuer le codage de la trajectoire en une suite d'idiomes. Ensuite, on compile cette suite d'idiomes en placements de caméras, en utilisant beaucoup la classe *EvalIdiomView* responsable du classement des meilleures vues pour un idiome donné. Tout ceci est assez linéaire :

```
// Constitution de la base d'idiomes
buildIdiomsBase();
// Parcours de la trajectoire de chaque élément du bras, et choix d'un idiome sur chaque
// portion :
vector<Idiom> idioms=path2idioms(configurations); // fonction
                                                  // path2idioms(vector<Configuration>) // de
                                                  // TransfIdioms.cpp
// Parcours de la liste d'idiomes et génération de la suite de configurations des trois caméras
configsCam=compilation(idioms,configurations); // fonction compilation(vector<Idiom>,
                                               // vector<Configuration>) de
                                               // CompilIdioms.cpp
// on a alors obtenu les vecteurs configsCamx du paragraphe précédent.
```

B.3 Filmage à l'aide de caméras fixes

Comme indiqué dans le paragraphe 4.3 (80), on filme ici une trajectoire imprévisible. Pour une image, nous allons commencer par faire exactement les mêmes opérations que pour le filmage par résolution de contraintes, seules l'obtention des configurations de caméras via un objet de type *PlaceCamera* va changer ; en effet ces lignes sont modifiées par :

```
// obtention des configurations de camera :
configsCam=filmF.getCamConfig();
```

configsCam1.push_back(configsCam[1]);

configsCam0.push_back(configsCam[0]);

configsCam2.push_back(configsCam[2]); ,

où **filmF** est l'objet de type *FilmageFixe* à travers lequel on contrôles les paramètres des caméras.

Annexe C

Trajectoire du bras-robot pour les tests sur trajectoire artificielle

Voici la trajectoire utilisée dans les premiers tests sur trajectoire artificielle :

Sur les 100 premiers point on fait bouger uniquement le joint « pitch » de l'épaule, afin de reconnaître un shot *pan vertical*. La configuration du bras-robot vaut alors, en fonction du numéro *i* du point :

configuration = (0, π/4 $\left(1-\frac{i}{100}\right)$ **, 0.0, 0.0, 3π/4, 0, 0, 0)**

On rétablit ensuite la configuration initiale pour *i* allant de 100 à 200 :

configuration = (0, π/4 $\left(\frac{i}{100}-1\right)$ **, 0.0, 0.0, 3π/4, 0, 0, 0)**

Puis on effectue plusieurs tracks successifs, comme au paragraphe 3.2.5 (52) :

- *i* allant de 200 à 300 : **configuration = (3.(*i-200*), π/4, π/5** $\left(\frac{i}{100}-2\right)$ **, 0, 3π/4, 0, 0, 0)**
- *i* allant de 300 à 350 : **configuration = (*i*, π/4, π/5, 0, π/4** $\left(3+\frac{i-300}{50}\right)$ **, 0, 0, 0)**
- *i* allant de 350 à 450 : **configuration = (*i*, π/4, π/5, 0, π, 0, 0, 0)**
- *i* allant de 450 à 500 : **configuration = (*i*, π/4, π/5, 0, π** $\left(1-\frac{i-450}{200}\right)$ **, 0, 0, 0)**
- *i* allant de 500 à 550 : **configuration = (*i*, π/4** $\left(1+\frac{i-500}{50}\right)$ **, π/5, 0, 3π/4, 0, 0, 0)**

- *i* allant de 550 à 650 : **configuration = (*i*, π/2, π/5, 0, 3π/4, 0, 0, 0)**
- *i* allant de 650 à 700 : **configuration = (*i*, π/2** $\left(1-\frac{i-650}{100}\right)$ **, π/5, 0, 3π/4, 0, 0, 0)**
- *i* allant de 700 à 800 : **configuration = (*i*, π/4, π/5, 0, 3π/4, 0, 0, 0)**
- *i* allant de 800 à 850 : **configuration = (*i*, π/4, π/5, 0, π/4** $\left(3+\frac{i-800}{50}\right)$ **, 0, 0, 0)**
- *i* allant de 850 à 950 : **configuration = (*i*, π/4, π/5, 0, π, 0, 0, 0)**
- *i* allant de 950 à 1000 : **configuration = (*i*, π/4, π/5, 0, π** $\left(1-\frac{i-950}{200}\right)$ **, 0, 0, 0)**

Ensuite, on effectue un mouvement horizontal selon l'angle φ_e (yaw) de l'épaule, afin de reconnaître un shot *pan horizontal*, pour *i* variant de 1000 à 1100 :

configuration = (*i*, π/4, π/5 – π/2 $\left(\frac{i}{100}-10\right)$ **, 0, 3π/4, 0, 0, 0)**

Puis on ramène le bras en position précédent cette rotation :

configuration = (*i*, π/4, π/5 + π/2 $\left(\frac{i}{100}-12\right)$ **, 0, 3π/4, 0, 0, 0)**

Enfin, le bras subit un mouvement de roll autour du segment épaule→coude, qui fait tourner le segment coude→poignet. Pour *i* allant de 1200 à 1300 on a :

configuration = (*i*, π/4, π/5, π/2 $\left(\frac{i}{100}-12\right)$ **, 3π/4, 0, 0, 0)**

On rétablit alors la position initiale, pour *i* variant de 1300 à 1400 :

configuration = (*i*, π/4, π/5, π/2 $\left(14-\frac{i}{100}\right)$ **, 3π/4, 0, 0, 0)**

Pour achever ces mouvements, on déplace les joints de la main suivant des mouvements sinusoïdaux pendant 100 points. Pour *i* variant finalement de 1400 à 1500,

configuration = (*i*, π/4, π/5, 0, 3π/4, cos(i), sin(i), – cos(i))

Note : L'algorithme est conçu pour ne pas tenir compte des variations de mouvement de la main lors du découpage en segments monotones. On reconnaît que seule la main bouge, mais cela n'entraîne pas l'ajout d'un point de coupure à chaque instant.

Bibliographie

[1] K. Belghith. (2005). *FPRM : Une nouvelle approche Flexible de Planification de Trajectoire de Robot dans un Environnement complexe.* MsC Thesis, Université du Québec à Montréal. url : http://planiart.usherbrooke.ca/~khaled/papers_khaled/FPRM-Memoirekhaled.pdf

[2] *Informations sur la Station Spatiale Internationale*. Récupéré sur Wikipedia: http://fr.wikipedia.org/wiki/Station_spatiale_internationale

[3] *Informations surle bras-robot Canadarm2*. Récupéré sur Wikipedia: http://fr.wikipedia.org/wiki/Station_spatiale_internationale

[4] F. Benhamou, F. Goualard, E. Languenou et M. Christie. (1994). Interval Constraint Solving for Camera Control and Motion Planning. *International Symposium on Logic Programming (ILPS),* (pp. 124-138). url : http://tocl.acm.org/accepted/goualard.pdf

[5] D. Nieuwenhuisen, M. H. Overmars. (2003). Motion Planning for Camera Movements in Virtual Environments. *MOVIE project* at Utrecht University. url : http://www.cs.uu.nl/groups/AA/movie/publications/PDF/cameraMotions.pdf

[6] D. B. Christianson, S. E. Anderson, L-W. He, D. H. Salesin, D. S. Weld et M. F. Cohen. (1996). Declarative Camera Control for Automatic Cinematography. AAAI . (pp. 148-155). url : http://grail.cs.washington.edu/pub/papers/dccl-aaai96.pdf

[7] W. H. Bares, J. C. Lester. (1997). Cinematographic User Models for Automated Realtime Camera Control. (pp. 215-230). *User Modeling: Proceedings of the Sixth International Conference*. url : http://research.csc.ncsu.edu/intellimedia/papers/ucam-um-97.pdf

[8] N. Halper, R. Helbing et T. Strothotte. (2001). A camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence. *Computer Graphics Forum , Volume 20, Issue 3* (pp. 174-183). url : http://wwwisg.cs.uni-magdeburg.de/graphik/pub/files/Halper_2001_CEC.pdf

[9] G. Olague. (1998). *Planification du placement de caméras pour des mesures 3D de précision.* PhD Thesis, INPG Grenoble.
url : http://www.inria.fr/rrrt/tu-0549.html

[10] M. Christie, R. Machap, J-M. Normand, P. Olivier et J. H. Pickering. (2005). Virtual Camera Planning: A Survey. Dans *Smart Graphics* (pp. 40-52) aux éditions Springer Berlin / Heidelberg. url : http://ifgi.uni-muenster.de/~kruegera/sg05/37.pdf/

[11] N. Courty, E. Marchand. (2001). *Navigation et contrôle d'une caméra dans un environnement virtuel.* MsC Thesis, IRISA - INRIA Rennes. url : http://www.irisa.fr/lagadic/pdf/2001_tsi_courty.pdf

[12] N. Halper, P. Olivier. (2000). CamPlan : A Camera Planning Agent. *AAAI Spring Symposium* (pp. 92-100).
url : http://wwwisg.cs.uni-magdeburg.de/graphik/pub/files/Halper_2000_CCP.pdf

[13] J. H. Pickering. (2002). *Intelligent Camera Planning for Computer Graphics.* PhD Thesis, University of York. url : http://www.cs.york.ac.uk/ftpdir/reports/YCST-2003-02.pdf