Skeg – Sex, Kinematics, Elegance and Glory
Projet Images MIM

David Baelde et Samuel Mimram

13 mai 2004

Table des matières

1 Objectifs

Nous avons voulu travailler sur l'animation. La cinématique inverse nous a paru être une technique intéressante, et facilitant la vie pour animer des personnages. Nous avons de plus ajouté des fonctions de rendu de metaballs à notre projet, afin de plaquer un aspect sympa sur les squelettes. Le but ultime était de fournir un environnement simple et puissant de développement d'animations.

2 Réalisations

2.1 La cinématique inverse

Le problème de la cinématique est de calculer le mouvement de points d'après des paramètres, translations, rotations. La cinématique inverse est le problème inverse : trouver des paramètres qui permettent un déplacement. Ensuite on

utilise ces paramètres pour effectuer le déplacement. L'intérêt est que le mouvement a été validé par le modèle.

Il est très pratique de disposer d'un tel outil quand on anime un squelette. Il suffit en effet de préciser la destination de certains points pour obtenir les déplacements en rotations qui correspondent au mouvement voulu, en accord avec les contraines imposées par le squelette.

Le domaine d'application est donc principalement l'animation, mais aussi les jeux vidéos, où les mouvement précalculés ne peuvent pas tout résoudre. Grâce à la cinématique inverse, Lara se penchera et se tournera un peu au besoin pour attraper un objet sur une table.

Cela peut aussi servir à interpoler un mouvement : si on veut que la main de A se retrouve sur la tête de B, une fois qu'on a les rotations des membres de A qui permettent ce déplacement, on peut les appliquer progressivement, en faisant varier la vitesse, pour obtenir une animation réaliste.

2.1.1 Modèle

On considère un squelette dont les os sont rigides. Les degrés de liberté sont les rotations des os les uns par rapport aux autres. Ce genre de modèle est beaucoup plus général qu'il n'y parait d'abord. En effet, on peut modéliser avec des systèmes d'os rigides un grand nombre de types de liaisons. Malheureusement nous verrons plus tard que nous allons devoir restreindre le modèle.

La relation entre la position des sommets et les orientations des os est trop compliquée pour être inversée. Par contre, on peut approximer au premier ordre les équations du mouvement pour obtenir un système linéaire. Il ne restera alors qu'à le résoudre.

Il y a deux catégories d'équations. D'abord les contraintes du squelette, pour tout os (i, j, d) de i vers j [1] et de taille d et orienté selon (θ, ϕ) :

$$\begin{pmatrix} dx_j \\ dy_j \\ dz_j \end{pmatrix} = \begin{pmatrix} dx_i \\ dy_i \\ dz_i \end{pmatrix} + d \times \begin{pmatrix} -d\theta \sin\phi \sin\theta & + & d\phi \cos\phi \cos\theta) \\ d\theta \sin\phi \cos\theta & + & d\phi \cos\phi \sin\theta \\ & - & d\phi \sin\phi \end{pmatrix}$$

Les autres équations sont les contraintes sur le déplacement demandées par l'utilisateur. Par exemple que i doit se déplacer de (dx, dy, dz) ou seulement (dans le cas du pingouin[2] metaball de la démo) que la tête doit avancer selon z de 0.5 pour suivre les pieds. Pour fixer des contraintes sur seulement certains axes il faut utiliser la fonction préfixée par `precise_`.

Bien sûr il a trainé une erreur de signe pendant longtemps dans nos équations, jusqu'à ce qu'on s'apercoive que le pingouin marchait dans la mauvaise direction. Mais l'erreur la plus grave, et longue à trouver, a été un autre problème de signe, dans le calcul des angles initiaux des os. En effet, le module

[1] Nous verrons plus tard pourquoi les os sont orientés. La limitation ne se situe pas à ce niveau puisqu'on pourrait considérer des os non orientés $(\{i, j\}, d)$ comme des paires $(i, j, d), (j, i, d)$.

[2] Ou barbapapa, *nose guy*, ...

est initialisé à partir d'une matrice d'adjacence et de la position initiale des points. Il trie les os et calcule leur orientation, ce qui consiste à passer de coordonnées cartésiennes à sphériques. Nous avons eu le malheur de copier sans réfléchir `http://mathworld.wolfram.com/SphericalCoordinates.html`, qui donne pour θ une formule qui ne marche que pour le secteur $x \geq 0$.

2.1.2 Résolution du système

Il existe plusieurs techniques de résolution, plus ou moins adaptées. Nous avons commencé par implémenter la technique la plus simple : utiliser le pivot de Gauss pour trouver une solution au système. Nous n'avons pas eu le temps d'en tester une autre.

Par exemple, il aurait été intéressant d'utiliser la technique des moindres carrés, qui cherche une solution par approximations successives. Il a en effet été montré [3] que les résultats sont plus fluides.

Nous n'avons pas trouvé de librairie résolvant les systèmes dont le rang n'est pas égal à la taille. C'est pourtant plus simple. Nous avons donc écrit l'algorithme de Gauss, où l'on essaie au plus vite d'éliminer des variables. Après cela on remonte les équations pour déduire la solution s'il y en a une. La méthode devrait marcher pour les systèmes comportant plus d'équations que de variables, mais nous n'avons pas eu affaire à ce cas et n'avons donc fait aucun test.

Si la théorie est simple, l'implémentation de ces algorithmes réserve quelques surprises. De plus, il est toujours difficile de repérer et comprendre une erreur : on ne peut pas afficher à l'écran les systèmes de 14 équations en 18 variables qui sont résolus toutes les secondes pour comprendre pourquoi tel système a échoué incompréhensiblement. Le problème principal rencontré est du à la précision forcément limitée du calcul. Un 0 peut être confondu avec un ϵ.

2.1.3 Application de mouvement

Une fois le système résolu, il faut appliquer les rotations. En fait, cela ne peut être fait simplement que dans le cas d'un squelette acyclique, dont on a déterminé un point fixe pendant le mouvement : la racine.

Dans ce cas, on trie préalablement les os du squelette selon un ordre topologique à partir de la racine. Ce tri est effectué dans `Ik.Acyclic` à chaque changement de racine. Ensuite on applique les rotations selon cet ordre.

2.1.4 API haut niveau

Nous disposons à ce stade du moyen de faire bouger un squelette vers une position proche. C'est la fonction `Ik.Gauss.elementary_move`. Plus la position est éloignée, moins l'approximation au premier ordre faite plus haut est pertinente, et au mieux les erreurs s'accumulent, au pire le système n'est plus faisable.

[3] http ://math.ucsd.edu/ sbuss/ResearchWeb/ikmethods

Pour faire un mouvement plus ample, nous avons implémenté la fonction `Ik.Gauss.move`. Elle appelle `elementary_move` pour s'approcher successivement de l'objectif fixé. Le nombre de pas peut être limité ou pas. Par défaut il vaut 10, on peut le changer, et une valeur négative signifie qu'il faut faire autant de pas que nécessaire pour atteindre l'objectif, à ϵ près. C'est avec un nombre de pas illimité qu'on a obtenu la bonne précision de la marche du bonhomme metaball de la démonstration.

2.2 Les metaballs

L'implémentation des metaballs que nous avons faite est inspirée de divers documents que nous avons pu trouver sur internet. Le code correspondant à ces metaballs se trouve dans `metaballs.c`. Nous les avons programmées en C et non pas en OCaml pour des raisons d'efficacité.

Le principe des metaballs est simple : il s'agit de dessiner une surface iso-potentiel. En pratique, le potentiel que nous avons choisi est une « gravité » *i.e.* un potentiel proportionnel à $1/r^2$ où r est la distance entre le point et la source. Cependant, l'affichage de telles surfaces s'avère difficile à effectuer en pratique. Il est en effet illusoire de vouloir en calculer une expression analytique car celle-ci serait extrêmement compliquée et peu exploitable, voire incalculable.

La technique que nous avons utilisée est la suivante. Nous découpons l'espace en une grille régulière (grille de cubes appelés *voxels*) et pour chaque voxel nous évaluons la valeur du potentiel en chacun des 8 sommets du voxel. Une fois que les potentiels ont été évalués en chaque sommet, suivant que cette valeur est supérieure ou inférieure à la valeur de l'iso-potentielle que nous voulons afficher, nous pouvons savoir quelle va être la forme de l'intersection du cube avec le voxel. Pour cela on regarde dans une table décrivant toutes les configurations possibles d'une intersection d'une iso-potentielle avec un voxel (technique des *marching cubes*). Pour chacune des possibilités, la table nous donne quels triangles il faut afficher et entre quelle et quelle arrête il faut les afficher. L'intersection de l'iso-potentielle avec les arrêtes du cube est calculée par interpolation linéaire entre les deux sommets adjacents à l'arrête. Cette approximation est relativement bonne (elle l'est d'autant plus que les voxels sont petits).

Enfin, pour accélérer encore l'exécution du programme, au lieu de calculer le potentiel en découpant tout l'espace en voxels, nous déterminons dynamiquement un parallélépipède qui délimite une zone dans laquelle l'iso-potentielle doit se trouver et nous ne faisons les calculs que sur le découpage en voxels de ce parallélépipède (et non pas de tout l'espace). Les bornes de ce parallélépipède sont les coordonnées des points extrêmes auxquelles on ajoute $n/\sqrt{p}$ (où n est le nombre de sommets et p la valeur de l'iso-potentielle que l'on veut afficher).

2.3 L'application `skeg`

La compilation nécessite `lablGL`. `make demo` ouvre une fenêtre avec un pingouin en metaballs. Les touches utilisables pour déplacer la caméra et le pingouin

s'affichent dans la console. `make interactive` lancera `skeg` en mode interactif, présentant un squelette que vous pouvez déformer à la souris.

Le mode interactif est le premier que nous avons créé. Le pingouin a été créé en vue de la démonstration et nous a permis de découvrir de nouveaux bugs. Malheureusement, le mode interactif est beaucoup plus capricieux depuis les dernières modifications. Souvent, la solution du système comporte des variations d'angle énormes, ce qui induit des mouvements incohérents. Il s'agit d'un bug du résolveur de système, qui renvoie une valeur qui n'est pas une solution au lieu de lancer l'exception correspondant aux cas d'erreur. Nous pensons qu'il s'agit d'un problème de système mal conditionné ou de précision du calcul flottant, car le vecteur constant, une fois la résolution terminée, est nul comme attendu.

2.3.1 Marche du pingouin

Dans sa version définitive chaque pression sur la touche `<espace>` commande les mouvement suivants du pingouin (fonction `Skeg.walk_step`) :

- Changer de pied racine.
- Avancer l'autre pied de $(0., 0., 1.)$. On ne fixe aucune limite sur le nombre de mouvement élémentaires nécessaires.
- Puis avancer (toujours sans limite de mouvement élémentaires) la tête de 0.5 selon z, et recaler dans le même mouvement le pied mobile sur sur axe, sans bouger selon z.

Des pressions trop rapides sur `espace` (laisser appuyer pour tester) entrainent une annulation du mouvement précédent et le début d'un autre mouvement. Le pingouin n'avance plus la tête, ses pieds sortent de leurs rails. Cependant, s'il ne s'est pas trop éloigné, il reviendra à sa place en en deux mouvements supplémentaires complets. Sinon il essaiera à l'infini de poser sur pied sur un rail, suspendu dans l'espace par son autre pied.

On voit que la spécification du mouvement a été facile. Le seul inconvénient est que la tête peut se baisser en pivotant autour du corps, tant qu'elle avance de la quantité requise. On ne pouvait pas la contraindre à rester à altitude constante. Il serait bon de pouvoir préciser des contraintes sous formes d'inégalités, pour demander par exemple que la tête ne se baisse pas trop. Cela semble faisable mais il faudrait alors résoudre des problèmes de programmation linéaire, ce qui est nettement plus compliqué qu'une simple résolution de système linéaire.

2.3.2 Vidéos

Nous avons réalisé à différentes étapes du développement des sessions de `skeg`. Elles sont disponibles sur

`http://perso.ens-lyon.fr/samuel.mimram/docs/skeg`

3 Conclusion

Nous sommes assez satisfait de la stabilité de la marche du pingouin, bien qu'elle soit du coup moins sympatique que les premières versions chaloupées et imprécises.

Il serait bon de comprendre (et corriger) pourquoi le mode interactif est si peu manipulable. D'autre part, il serait bon de développer une API encore plus haut niveau, avec un type de squelette plus facile à définir, notamment en donnant des noms aux nœuds. Enfin, les metaballs ne nous donnent pas entière satisfaction car elle n'ont pas toutes les caractéristiques d'un modèle physiquement réaliste. Par exemple, si le pingouin fait se toucher ses deux pieds, il fusionnent ! Il faudrait donc soit changer compétement de technique pour « mettre en chair » les squelettes, soit utiliser des metaballs améliorées sur lesquelles on pourrait mettre des contraintes qui éviteraient les fusions intempestives[4].

[4] les *zspheres* nous semblent être un bon outil dans cette direction