Un moteur de terrain

Gabriel Peyré
nikopol0@altern.org
www.orion3d.fr.st

Le 19 septembre 2001

Table des matières

Introduction

- DOCUMENT EN COURS DE REDACTION -

Encore une fois, je vous engage à lire cet article avec un oeil critique, car tout ceci constitue l'état de mes réflexions sur le rendu de terrains, cette branche de la programmation 3D ne regorgeant pas d'algorithme universellement reconnus et utilisés.

Je vais tacher de vous faire comprendre à la fois les structures de données adéquates pour générer tout ce qui se trouve sous les pieds de vos personnages, ainsi que les algorithmes pour décider de façon heuristique à quel niveau de détail rendre chaque portion du paysage.

L'idée principale est de faire non seulement un moteur de terrain efficace, mais surtout polyvalent, ie. qui s'adapte aux différentes représentation de notre terrain[1].

1. Qu'il s'agisse de *NURBS*, *heightfield*, etc.

Chapitre 1

Un tesslateur de terrain universel

La question qui doit vous effleurer les lèvre est sans nul doute "mais quel genre de terrain afficher à l'écran ?". En fait, il s'agit plutôt de se demander par quelle formule mathématique, ou par quelle subtilité programmatoire définir notre surface. La réponse que je voudrais apporter serait : "peu importe !". Notre *tesslateur de terrain* (nom barbare mais expressif ...) se doit d'être *universel*.

1.1 Structure du moteur

Pour y parvenir, nous allons diviser le moteur de terrain en deux parties :

- Les surfaces définissant notre sol : elles dériveront toutes d'une même *interface*, qui leur permettrons d'être utilisé de façon transparente par le *tesslateur*.
- Le *tesslateur* proprement dit. On lui donne une surface lors de se creation, qu'il doit utiliser pour calculer les position, normales, etc. des points composant notre surface.

1.2 Les surfaces

Décrivons tout d'abord les caractérisitiques de nos surfacés, ou plutôt quelles fonctions ces surfaces doivent implémenter :

- `vector GetVertex(float u, float v)` : retourne la position d'un vertex.
- `vector GetNormal(float u, float v)` : retourne la normale à un vertex.
- `float GetCourbure(float u, float v)` : retourne la courbure à un vertex (cf. plus loin).

1.3 Le teslateur

– TODO –

Chapitre 2

Gestion du L.O.D.

Le but d'un moteur de terrain est principalement de pouvoir définir de façon "locale" la précision du rendu du sol. Pour y parvenir, nous alors utiliser des structures de données adéquats, dont la plus importante est certainement le *quad-tree*.

2.1 Les quad-trees

– TODO –

2.2 Structure du terrain

Le terrain peut se voir à deux échelles :

- Le sol est composé de plusieurs surfaces carrées (*patch*), collés les uns aux autres de façon lisse (*C1* : les tangentes se recollent). Chaque *patch* occupe une feuille d'un grand quad-tree.
- Chaque surface est à sont tour découpée en carré, de façon récursif. Ainsi, on commence par un grand carré de la taille de la surface, puis on le divise en quatres carrés de tailles égales. En continuant ainsi jusqu'à une certaine precision, on remplie un *quad-tree* remplie de carrés de plus en plus petits. Plus on s'enfonce dans l'arbre, plus le sol est rendu avec précision.

On a donc en quelque sorte un "*quad-tree de quad-tree*", ce qui symbolise les deux échelles où va se jouer le calcul de *L.O.D.*

2.3 Le quad-tree global

Intéressons nous tout d'abord au *quad-tree* global (celui qui est composé de surfaces *patch*). Ce *quad-tree* sert à faire des calculs de rapides, principalement de deux sortes :

- Clipping des *surfaces* : le *quad-tree* nous permet de déterminer rapidement quelles surfaces sont dans le champ de la camera, et ainsi de ne dessiner que celles qui sont nécessaires.
- Mise au point grossière du *LOD*[1] : pour chaque *surface*, en fonction de sa distance à la camera, on fixe de façon grossière la precision que l'on désire avoir pour l'ensemble de la surface. Ainsi, une surface éloignée sera dessinée avec peu de précision.

2.4 Les surfaces patch

Ensuite, précisons comment sont définies nos surfaces patch :

- Les surfaces seront en fait *C1 par morceau*, ce seront donc plusieurs *surfaces paramètrées*par un couple (u,v), qui variera sur un carre, les différents "carrés" se recollerons de façon "lisse" (chaque carré constitue donc un *patch* ...).

1. pour **L**evel **O**f **D**etail

- Le *quad-tree* utilise un système de pointeurs pour ne pas répéter les différents *vertex* communs. [chaque noeud est un carre, auquel il faut ajouter les 4 vertex correspondants].
- A chaque noeud est accroché aussi un `float`, qui représente le *niveau de precision*, qui depend de la courbure en ce point [plus la *surface* est *courbée*, plus elle a besoin d'être *teslatée* avec précision]. Grosso-modo, ça exprime la distance moyenne entre le *polygone teslaté* et la *surface*.

2.5 Calcul de la courbure

– TODO : définition mathématique –

C'est là le point critique, car il n'y a pas d'expression exacte de cette quantité. Pour un *nurbs* [2], y'a moyen de le calculer de façon + ou - exacte [plutôt moins que plus, d'ailleurs], mais je propose une alternative : on calcule la "courbure Moyenne", qui se calcule par $2.H = k_1 + k_2$, ou k_1 est la courbure minimale au niveau du point et k_2 la courbure max [ie. les valeurs propre de la matrice de la *Heissienne*, matrice des dérivées secondes, donc $H = \frac{1}{2} * trace(Heiss(f))$]. OK, j'arrête avec les maths. Juste pour votre culture, les surfaces telles que $H = 0$ partout sont les *surfaces minimales* [par exemple les *bulles de savon* ...].

2.6 Construire le quad-tree

Enfin, vient la question fatidique : comment calculer ce quad-tree ? Je propose de faire ça de façon progressive, et pour ne pas que la place mémoire "explose", de supprimer les arbres des carrés trop loin de la camera.

– TODO : préciser tout ça –

2.7 Affichage du terrain

Ensuite, vient la question de l'affichage du terrain. A chaque image, on fait un parcours de l'arbre, et on s'arrête quand la précision est suffisante, ie. quand la precision marquée dans l'arbre est plus petite qu'un seuil donné par la distance a la camera.

– TODO : préciser tout ça –

Comme ça, une partie de la surface qui est soit très courbée, soit très proche, sera *teslatée* avec une bonne precision.

Cependant, cette méthode ne va pas sans quelques problèmes : si on s'arrête n'importe ou dans l'arbre, il risque d'y avoir des "*trous*" dans la surface, qu'il faut boucher, par exemple par un parcours préliminaire. Ainsi, à chaque fois qu'un carré est calculé avec moins de précision que son voisin, on replace le point problématique par une *interpolation linéaire* savamment calculée.

– TODO : mettre un graphique –

2. pour **N**on **U**niform **B-S**pline

Chapitre 3

Détection de collision

Dand l'article intitulé "détection de collision", vous avez appris[1] à effectuer les tests de collisions entre les objets qui composent votre scène. Cependant, nous venons d'introduire de nouvelles structures de données[2], comment donc les utiliser pour réaliser une détection de collision efficace ? Et plus important encore : comment réagir à ces collisions pour que les objets ne traversent pas le sol?

– TODO –

1. Si si !
2. Principalement les quad-tree

Chapitre 4

Conclusion

Comme vous avez pu le remarquer, je ne suis pas rentré dans le détail des fonctions à utiliser pour définir nos surfaces. Un moyen simple est d'utiliser une heightmap[1], mais on peut penser à des fonctions mathématiques complexes, telles les *NURBS*. Une bonne idée serait d'utiliser des *surfaces de Bezier*, qui sont des courbes paramètrées de degrés 3. Elles sont moins précises que les *NURBS*, mais ont l'avantage de permettre des schémas de calcul optimisés[2].

– TODO : améliorations possibles –

1. Carte de hauteur
2. Par *dichotomie*, entre autre

Annexe A

Références

Voici un ensemble de resource on-line, avec une brève description.

- La page de James Stewart
  http://www.dgp.toronto.edu/people/JamesStewart/terrainSummary/
- La page personnelle de **Ming C. Lin**
  http://www.cs.unc.edu/~lin/
- Un tutorial sur les *quad-tree*
  http://www.gamedev.net/reference/programming/features/quadtrees/

Bibliographie