Niveau de détail et cartes de hauteurs

1 Préambule

Ce document a pour but de traiter des différents algorithmes utilisés pour réduire la géométrie des cartes de hauteurs, c'est-à-dire diminuer le nombre de triangles affichés afin de pouvoir afficher des terrains conséquents en temps réel.

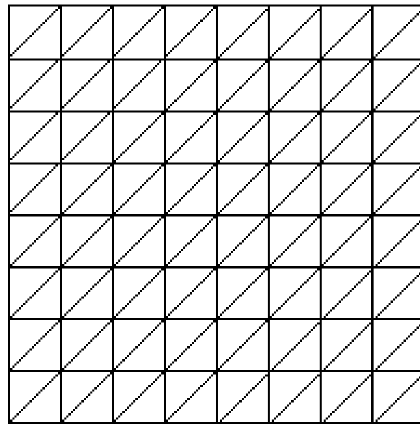
Les méthodes évoquées sont au nombre de deux : Le QuadTree et ROAM. Ce sont des techniques de niveau de détails particuliers, des CLODs (Continuous Level Of Detail).

Alors que les LODs proposent plusieurs niveaux de détails distincts pour un maillage, les CLODs sont de techniques qui permettent des variations du niveau de détails pour un même maillage.

Le but de ce document est également de proposer des solutions pour accélérer le rendu de cartes de hauteurs.

2 Présentation des cartes de hauteurs

Les cartes de hauteurs sont des maillages qui possèdent des caractéristiques particulières importantes. Au niveau du rendu en mode file de fer, nous observons toutes leurs singularités en vue de dessus :



Représentation d'un maillage régulier.

- Les distances en x et y sont équivalentes et en 2 puissance n ou 2 puissance n moins 1.
- La distance entre chaque vertex en x est identique
- La distance entre chaque vertex en y est identique

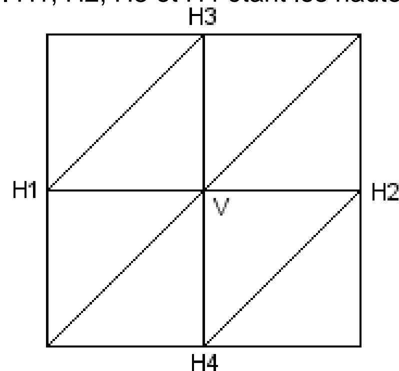
Ce type de maillage est dit régulier. Ces quelques propriétés facilités grandement le calcul des coordonnées de textures et celui des normales.

Les coordonnées de textures sont données par la relation :
vec2 TexCoord = PositionVertex * TailleMesh * Repete.

Repete : Scalaire indiquant le nombre de fois que la texture doit être répété sur le maillage.

TailleMesh : Vecteur à 2 dimensions indiquant le nombre de sommet en largeur (x) et en hauteur (y).

Le calcule d'une normale du sommet V est alors donnée par la formule : N = normalize(vec3(H2 – H1, H3 – H4, 2)). H1, H2, H3 et H4 étant les hauteurs des sommets associés.

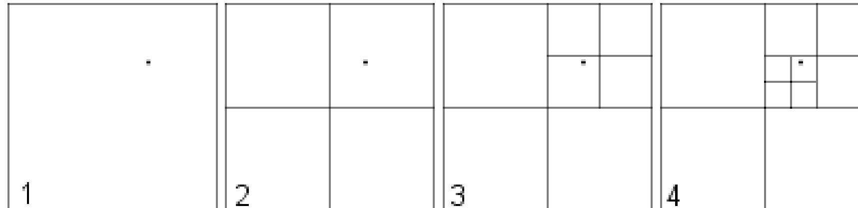


Pour tous les CLODs, l'idée est de n'afficher qu'une partie des sommets. Nous avons des solutions simples pour calculer les coordonnées de textures et les normales pour chaque sommet, l'application des différentes techniques est à présent parfaitement viable.

3 QuadTree

3.1 Principe

Le QuadTree est une structure 2D dont l'algorithme est une méthode récursive basée sur la subdivision d'un carré en quatre autre carré plus petit. Admettons que l'on recherche à localiser un point sur un carré à l'aide d'un QuadTree. Nous aurons alors le traitement suivant pour une profondeur de récursivité 3 :



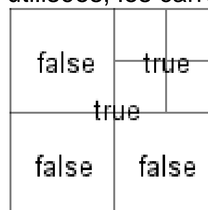
Résultat du travail d'une quadtree en 3 récursions.

Ce paragraphe ne nous permet pas une application telle que nous la souhaitons, il s'agit d'une simple explication. Le principe ne change pas avec les cartes de hauteur, c'est seulement l'application qui est différente.

3.2 Application sur une carte de hauteur

Pour la partie précédente, le critère de récursivité était un simple point localisé dans l'espace plan des réels. A présent, il s'agit d'une structure basé sur la carte de hauteur : la matrice de quadtree. Il s'agit en fait d'une matrice de la même taille que la carte de hauteur mais dont le contenu indique comment il faudra subdiviser la carte de hauteur en triangle au moment du rendu. Il existe deux stratégies principales. La première consiste à demander la subdivision suivant la position de la caméra sur la carte et la seconde consiste à subdiviser la carte suivant les variations du relief. Si une zone de la carte est plane alors les subdivisions sont peu nombreuses mais si une zone est accidentée, il y a davantage de divisions. La subdivision reste limité à la taille de la carte de hauteur et il est parfaitement possible de jouer sur la récursivité pour manipuler le ratio précision / vitesse dans les deux cas.

Concrètement la « quad matrix » contient des valeurs booléennes indiquant pour chaque sommet s'il doit y avoir subdivision ou non. Il y a en fait trois possibilités de valeur. La première, « true » en générale, indique qu'il faut subdiviser en ce point. La seconde « false » indique de ne pas subdiviser et la dernière est indéterminée. Il s'agit de toutes les valeurs qui exploité pendant le rendu. En effet, on commence comme précédemment par un carré s'inscrit sur la totalité de la carte de hauteur. Si l'on subdivise une première fois, nous avons quatre carrées et dans la matrice nous indiquons « true » pour le sommet central. Si trois d'un carré ne doivent pas être subdivisé alors les sommets centraux de la matrice prennent la valeur « false » et peu nous importe les autres valeurs de ces carrées puisse qu'elles ne seront pas utilisées, les carrés ne seront pas subdivisés de nouveaux.



Notons ici l'intérêt de la contrainte sur la taille de la carte de hauteur. Si la taille est en 2 puissances n + 1 alors on s'assure de trouver une valeur entière et unique pour l'entrée de subdivision de la quad matrice.

3.3 Subdivision suivant la camera

Précisons le critère de subdivision pour la génération de la « quad matrix » dans le cas d'une dépendance avec la position de la camera. Nous subdivisons uniquement si la variable de décision « f » est inférieure à 1. Nous obtenons la valeur de f avec la formule suivante :

f = 1 / (distance * ResolutionMinimum * max(ResolutionSouhait, 1))

distance : Distance « L1-Norm » entre la position de la camera et le centre de subdivision, se calculant ainsi : float distance = abs(Centre.x – Camera.x) + abs(Centre.y – Camera.y) + abs(Centre.z – Camera.z)

ResolutionMinimum : Résolution minimum du maillage affiché

ResolutionSouhait: Résolution souhaité du maillage affiché

Revenons un instant sur les résolutions. Elles indiquent la quantité de subdivisions et donc la quantité de détails du maillage en sortie. Augmenter de un la résolution, revient à multiplier par quatre le nombre de triangles nécessaires pour le rendu.

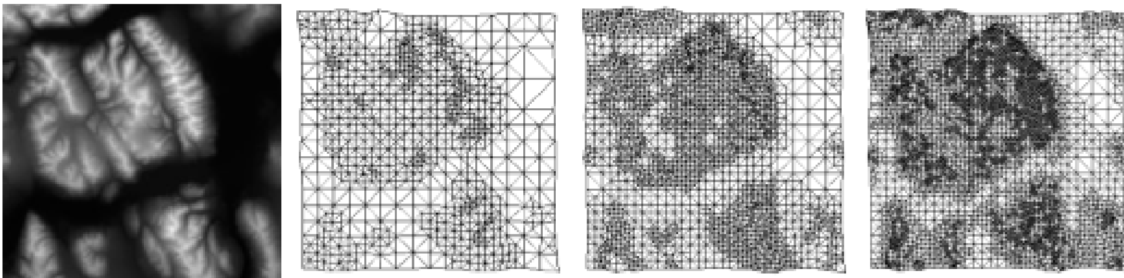
Notons que pour chaque rendu d'image, il faut réitéré tout l'algorithme d'affichage, nous pouvons au plus espérer mettre à jour la « quad matrix » seulement si la camera à été déplacé.

3.4 Subdivision suivant le relief

Un second critère de subdivision consiste à vouloir ajouter du détail dans les zones accidenté et à subdiviser nettement moins pour les surfaces planes. Pour chaque carré, nous calculons la différence entre le centre et chacun des quatre coins (dhi) puis nous choisissons la valeur maximum parmi les quatres. Ce nombre est alors divisé par la taille du carrée incriminé afin de conserver une homogénéité du critère en fonction du niveau de détail. Nommons cette valeur final d2.

d2 = max(fabs(dhi)) / distance

Ceci à pour conséquence de modifier le critère de subdivision du paragraphe précédent ainsi :
f = 1 / (distance * ResolutionMinimum * max(ResolutionSouhait * d2, 1))



Une carte de hauteur et trois maillages générés avec différents niveaux de détails suivant le relief

3.5 Section en T

La subdivision de la surface en carrée à partir d'une carte de hauteur donne un maillage irrégulier en sortie :

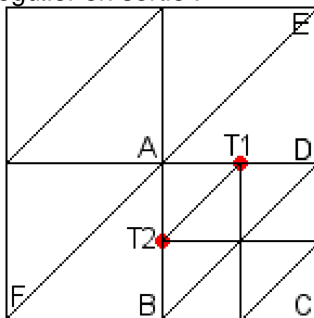


Figure 1

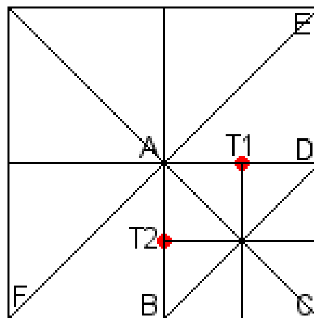


Figure 2

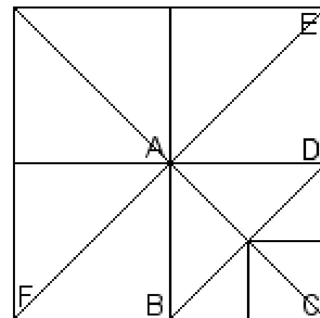
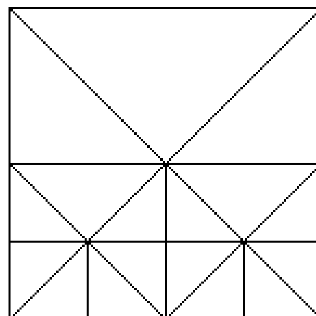


Figure 3

Ceci à pour conséquence de produire des bugs de continuité entre deux surfaces de niveau de détails différents. Sur l'exemple ci-dessus, si le point T1 n'appartient pas au plan AED, on observera un défaut de rendu. Pour résoudre ce problème, nous devrons supprimer les sections en « T ». Pour cela, il faut modifier le découpage triangulaire du maillage original pour le remplacer par une organisation en étoile (figure 2). En suite, suivant la profondeur des détails des triangles voisins on adapte le découpage du carré (figure 3). Ce découpage adaptatif peut donner la figure suivante dans un cas extrême :



En général, le maillage final sera loin d'être de cette forme car les sections en T sont des éléments particuliers et non usuels.

3.6 Algorithme de rendu

Par ce qu'un morceau de code en dit plus que de longs paragraphes dans certains cas, voici un algorithme d'écrivant le rendu d'un maillage avec un quadtree dans speudo code inspiré du C++.

```
// x : coordonné de découpage horizontale
// y : coordonné de découpage verticale
// longeur: taille du maillage : 512, 256, 128, 64, …
void Terrain::Node(float x, float y, int longeur)
{
	// Si l'on atteint un niveau de détail égale à celui de la carte de hauteur,
	// on ne peut plus subdiviser … alors on affiche.
	if(longeur <= 2)
	{
		AfficherTriangles();
		return;
	}

	// Affichage des triangles
	switch(AdapterAffichageSuivantSubdivisionVoisine())
	{
	case TOUS: … return; // Cas usuel
	case AUCUN: … break; // Cas usuel
	case ADAPTE: … break; // Cas particulier, on affiche ou non suivant chaque voisin
	}

	// Tous les nœuds, qui n'ont pas étés affichés, sont subdivisés
	for each(child)
	{
		// Appel récursif de la fonction pour un niveau de détails supérieur.
		Node(child_x, child_y, longuer / 2);
	}
}
```

L'algorithme précédent est relativement simple, cependant nous pouvons observer que le nombre d'appel à la fonction « Node » peut très rapidement augmenter. En effet, un appel génère au plus quatre nouveaux appels à « Node ». Ainsi sur une configuration de type Athlon XP 2400+ (2.0 Ghz), cet algorithme laisse espérer l'utilisation de carte de hauteur jusqu'en 256*256 voir 512*512 en temps réel mais difficilement au-delà. Cependant, il existe plusieurs techniques de culling permettant de gagner considérablement en performance. (cf partie 5)

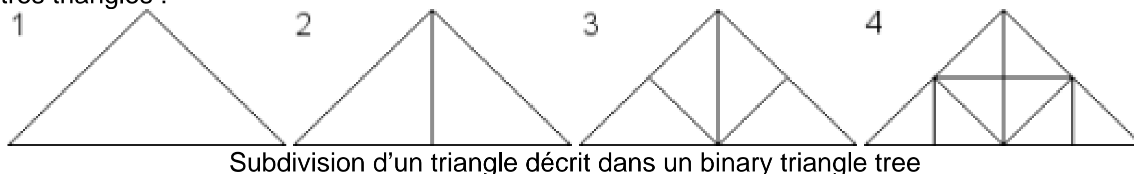
4 ROAM

4.0 Présentation

Le Quadtree est une méthode importante dans le rendu de carte de hauteur mais ce n'est pas la seule. L'algorithme Real-time Optimally Adapting Mesh (ROAM) en est une autre qui tend à prendre le pas sur le Quadtree, du fait des nombreuses optimisations permissent par cet algorithme.

4.1 Le binary triangle tree

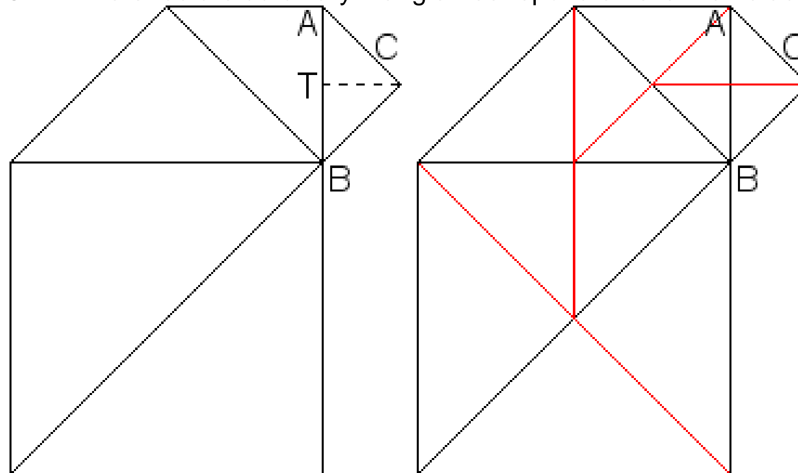
Alors que le quadtree repose sur un algorithme récursif et une matrice, ROAM repose sur une structure nommée « binary triangle tree ». On part d'un simple triangle que nous divisons en deux autres triangles :



Subdivision d'un triangle décrit dans un binary triangle tree

Pour chaque étape de raffinement, nous dessinons la bissectrice du triangle à partir de l'angle droit. Contrairement à ce que l'on pourrait penser le binary triangle tree ne contient aucune information sur les polygones, il s'agit simplement d'un arbre contenant des liens vers des noeuds voisins et des noeuds fils.

Pour le QuadTree nous avons vue que la méthode contenait un bug pour les jonctions en T, Quand est t'il avec ROAM ? La structure du binary triangle tree répond en elle-même au problème :

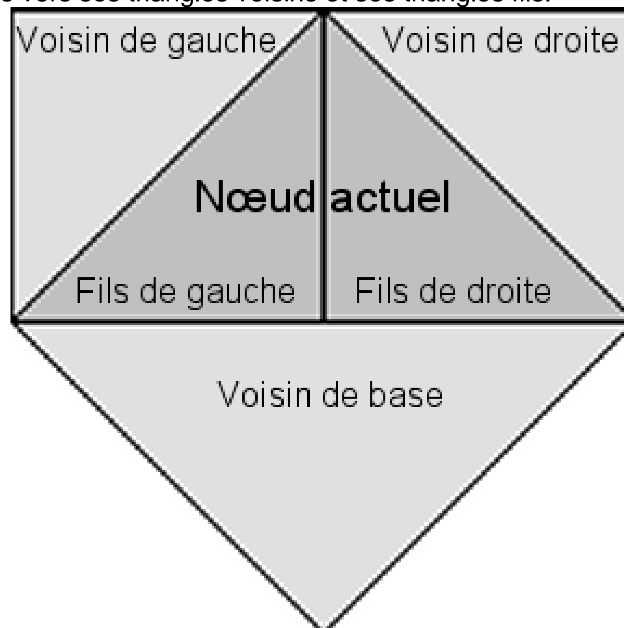


S'il l'on souhaite subdiviser le triangle ABC alors une jonction en T pourrait apparaître au point T sur le dessin ci-dessus. Cependant, la structure même du binary triangle tree permet de résoudre la problème en subdivisant tous les nœuds nécessaires, ce qui donne un maillage continue et donc sans jonction en T.

L'algorithme permettant la mise à jour de l'arbre repose sur deux files d'attentes : la « split queue » (la file de scission) et la « merge queue » (la file de fusion). Ces files vont conserver les priorités pour chaque triangle du maillage en commençant par le maillage grossier (non subdivisé) puis en forçant la scission et la fusion avec les triangles de plus grandes priorités. Il est important de maintenir le fait que la priorité d'un nœud parent doit être supérieur à celle d'un nœud enfant.

4.2 Les données du binary triangle tree

Comme nous l'avons dit précédemment, le binary triangle tree ne sauve pas de valeur géométrique. Mais que contient il alors ? Il ne s'agit que d'un arbre dont chaque élément, chaque triangle, dispose de liens vers ses triangles voisins et ses triangles fils.



En terme de code C/C++ cette structure peut-être implémentée ainsi :

```
struct Node
{
   Node* pLeftChild;
   Node* pRightChild;
   Node* pLeftNeighbor;
   Node* pRightNeighbor;
   Node* pBaseNeighbor;
};
```

4.3 « Split and merge » : Scission et fusion

Il existe deux méthodes pour utiliser ROAM. La première consiste à conserver la structure précédente et à la modifier. Avec cette méthode, suivant la position de la caméra on choisit de subdiviser ou fusionner les nœuds par rapport au critère de subdivision. Nous pourrions croire que la récupération des données est plus rapide cependant ce n'est pas toujours le cas. En effet, une méthode plus récente et plus rapide développé par Seumas McNally, propose de ne rien conserver et de régénérer la totalité de l'arbre en n'effectuant que des subdivisions. Le rendu avec ROAM s'effectue en deux étapes, on construit l'arbre puis on l'affiche.

4.4 Algorithme de rendu

Si dessous, nous avons l'algorithme permettant le rendu d'un terrain à partir d'un triangle binary tree. L'algorithme est particulièrement simple :

```
// Les paramètres '…' indique les coordonnées de chaque sommet du triangle actuel.
void CTerrain::RenderNode(Node *pCurrentNode, ...)
{
   // S'il y a fils gauche alors il y a aussi un fils droit
   if(pCurrentNode->pLeftChild)
   {
      RenderNode (pCurrentNode->pLeftChild, ...);
      RenderNode (pCurrentNode->pRightChild, ...);
      return;
   }

   // Sinon la subdivision est terminé, on affiche le triangle
   RenderTriangle();
}
```

4.5 Algorithme de création de l'arbre

Si dessous, nous avons la fonction récursive principale permettant la création de l'arbre :

```
// Les paramètres '…' indiquent les coordonnées de chaque sommet du triangle actuel.
void CTerrain::SubdiviseNode(Node *pCurrentNode, ...)
{
   // On continue la subdivision suivant le critère de subdivision
   if(Subdivise())
   {
      // Si le noeud ne dispose pas de fils, le triangle est subdivisé
      if(!pCurrentNode->pLeftChild)
         Split(pCurrentNode);

      if(pCurrentNode->pLeftChild)
      {
         SubdiviseNode(pCurrentNode->pLeftChild, ...);
         SubdiviseNode(pCurrentNode->pRightChild, ...);
      }
   }
}
```

4.6 Subdivision

Si le rendu et la fonction récursive sont très simple, la subdivision est elle plus risqué car il faut s'assurer que chacun des cinq points de chaque nœud soit prise en compte. Nous retrouvons ici tous les particularités d'implémentations des listes doublements chaînés :

```
void CTerrain::Split(Node *pCurrentNode)
{
   // Si le triangle créé une section en T alors on subdivise le triangle de base.
   if(pCurrentNode->pBaseNeighbor && (pCurrentNode->pBaseNeighbor->pBaseNeighbor !=
pCurrentNode))
```

```
        Split(tri->pBaseNeighbor);

    // Allocation des fils du noeud
    pCurrentNode->pLeftChild  = new Node;
    pCurrentNode->pRightChild = new Node;

    // On complète les informatiques sur ses voisons des fils avec celle des parents
    pCurrentNode->pLeftChild->pBaseNeighbor = pCurrentNode->pLeftNeighbor;
    pCurrentNode->pLeftChild->pLeftNeighbor = pCurrentNode->pRightChild;
    pCurrentNode->pRightChild->pBaseNeighbor = pCurrentNode->pRightNeighbor;
    pCurrentNode->pRightChild->pRightNeighbor = pCurrentNode->pLeftChild;

    // On lie le voisin de gauche avec le fils gauche
    if(pCurrentNode->pLeftNeighbor)
    {
        if(pCurrentNode->pLeftNeighbor->pBaseNeighbor == pCurrentNode)
            pCurrentNode->pLeftNeighbor->pBaseNeighbor = pCurrentNode->pLeftChild;
        else if(pCurrentNode->pLeftNeighbor->pLeftNeighbor == tri)
            pCurrentNode->pLeftNeighbor->pLeftNeighbor = pCurrentNode->pLeftChild;
        else if(pCurrentNode->pLeftNeighbor->pRightNeighbor == pCurrentNode)
            pCurrentNode->pLeftNeighbor->pRightNeighbor = pCurrentNode->pLeftChild;
        else // Erreur !
    }

    // On lie le voisin de droit avec le fils droit
    if(pCurrentNode->RightNeighbor)
    {
        if(pCurrentNode->RightNeighbor->BaseNeighbor == pCurrentNode)
            pCurrentNode->RightNeighbor->BaseNeighbor = pCurrentNode->RightChild;
        else if (pCurrentNode->RightNeighbor->RightNeighbor == pCurrentNode)
            pCurrentNode->RightNeighbor->RightNeighbor = pCurrentNode->RightChild;
        else if (tri->RightNeighbor->LeftNeighbor == pCurrentNode)
            pCurrentNode->RightNeighbor->LeftNeighbor = pCurrentNode->RightChild;
        else // Erreur !
    }

    // On lie le voisin de base avec les nouveaux fils
    if(pCurrentNode->pBaseNeighbor)
    {
        if(pCurrentNode->pBaseNeighbor->pLeftChild)
        {
            pCurrentNode->pBaseNeighbor->pLeftChild->pRightNeighbor = pCurrentNode->pRightChild;
            pCurrentNode->pBaseNeighbor->pRightChild->pLeftNeighbor = pCurrentNode->pLeftChild;
            pCurrentNode->pLeftChild->pRightNeighbor = pCurrentNode->pBaseNeighbor->pRightChild;
            pCurrentNode->pRightChild->pLeftNeighbor = pCurrentNode->pBaseNeighbor->pLeftChild;
        }
        else // Il faut diviser pour éviter les sections en T !
        {
            Split(pCurrentNode->BaseNeighbor);
        }
    }
    else
    {
        // Dans le cas d'un triangle de coin alors, il n'y a pas de voisins
        pCurrentNode->LeftChild->RightNeighbor = 0;
        pCurrentNode->RightChild->LeftNeighbor = 0;
    }
}
```

5 Optimisation

Il existe plusieurs techniques permettant d'améliorer encore la vitesse du rendu à partir des algorithme ROAM et QuadTree. En voici trois exemples.

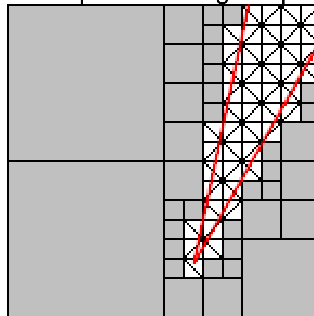
5.1 Face culling

La première, est le traditionnel « Face culling ». Il s'agit simplement d'une technique permettant de ne dessiner que sur un seul coté de chaque face. Pour cela, il faut dessiner chaque face de telle manière que les faces avant soient toutes du même cotées du maillage.

Rendu de l'image en 640 * 480 avec éclairage et 4 textures par face
Avec face culling : 316 images par seconde
Sans face culling : 176 images par seconde

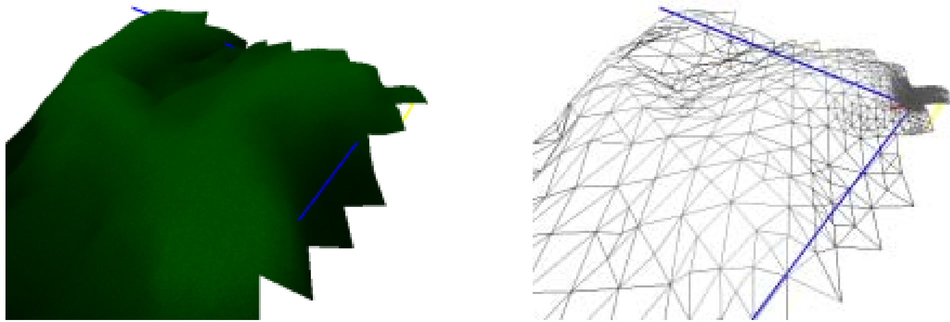
5.2 Frustum culling

La seconde technique est nommée « frustum culling ». Cette technique à un véritable impact sur la géométrie. Il s'agit de ne subdiviser et de n'afficher que ce qui est dans le champ de la caméra. Sur l'image si dessous, si nous considérons les traits rouges comme représentant du champ de vision, seuls les triangles blancs sont affichés. Nous gagnons sur deux tableaux : nous réduisons le nombre de triangle affichés et nous ne subdivisons que les triangles qui nous intéresse.



Frustum culling sur une carte de hauteurs.

Le frustum culling est particulièrement simple à implémenter. Il suffit simplement au début de la fonction « Node » de notre algorithme, de vérifier si le centre de subdivision ou ses voisins sont dans le champ de vision. Sinon, nous arrêtons le processus récursif.



Exemple OpenGL utilisant ROAM pour le rendu de carte de hauteur avec Frustum culling.

5.3 Occlusion culling

Une troisième technique encore en plein développement est nommée occlusion culling. Elle consiste à n'afficher que les triangles visibles à l'écran et repose sur les extensions OpenGL GL_ARB_occlusion_query, GL_NV_occlusion_query ou GL_HP_occlusion_test. Elle suppose également que les triangles à afficher, soient stockables car le rendu s'effectue en deux passes. La première exécute l'algorithme CLOD avec un rendu minimaliste, c'est-à-dire sans la moindre texture, sans éclairage ou tout autre effet graphique. Cette passe doit stocker les triangles et marquer ceux triangles qui sont effectivement visibles. La deuxième passe s'effectue avec tous les effets graphiques activés et n'affiche que les triangles marqués visibles. Cette technique sera particulièrement efficace pour des reliefs très marqués comme les montagnes. Notons que l'occlusion culling à l'avantage de reporter des calcules sur le GPU contrairement aux précédentes techniques.

Enfin, précisons que les trois techniques présentées sont parfaitement capables de travailler en coopération.

6 Conclusion

Nous avons vue deux méthodes importantes pour la réalisation de niveau de détails : Le QuadTree et ROAM. Alors que le QuadTree dispose d'un algorithme stable qui n'évolue pas, ROAM est en constante évolution. En effet, il existe de nombreuses optimisations de ROAM généralistes ou adapté à des besoins spécifiques. Par exemple, une nouvelle version de ROAM propose de remplacer le binary triangle tree par un diamond tree (une structure à base de losange) ce qui permet de rendre la stratégie « split and merge » à nouveau crédible en terme de performance.

Cependant, actuellement il y a un tel écart entre les performances des cartes graphiques et des processeurs que les algorithmes de LODs perdent de leurs intérêts. En effet, en chargement l'intégralité des triangles sur la carte graphique, en utilisant seulement les techniques de culling, nous déchargeons le travail du CPU en le renvoyant sur le GPU.

7 Bibliographie

- [1] Trent Polack. 3D Terrain Programming. Premier Press, 2003
- [2] Stefan Rottger. Real-Time Generation of Continuous Levels of Detail for Height Fields.
- [3] Jason Shankel, Fast Heightfield normal calculation in Game Programming Gems 3 Charles Rivers Media, 2003
- [4], Occlusion culling in Game Programming Gems 4 Charles Rivers Media, 2004
- [5] Eric Lengyel, OpenGL Extensions Guide, Charles Rivers Media, 2003
- [6] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Millery, Charles Aldrich, Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes.
- [7] Bryan Turner, Real-Time Dynamic Level of Detail Terrain Rendering with ROAM, www.gamasutra.com, 2000
- [8] Seumas McNally, Tread Marks, www.treadmarks.com
- [9] V-Terrain, Terrain LOD: Runtime Regular-Grid Algorithms, http://www.vterrain.org/LOD/Papers

26/03/2005 – Groove (groove'at'g-truc'point'net)