KartoMNT v2.0

Documentation du logiciel.

Ce document a pour but d'expliciter les grandes lignes et les points importants du fonctionnement du logiciel KartoMNT. Il est destiné à toute personne qui souhaiterait comprendre ou modifier les sources de KartoMNT.
Cette documentation ne couvre pas avec précision l'intégralité des fonctionnalités mais essaie d'apporter le maximum de clarté sur le programme.
Elle est ouverte à toute correction, précision, ou extension si il en est jugé utile.

Auteurs : *Jean BRESSON*
Roland DERHI
03-2003

1. Fonctionnement général de l'application

1.1. Interface

A la base du programme, la classe *Main* démarre l'application en ouvrant la fenêtre principale. Il s'agit de la classe *MainWindowSDI*. Dans cette fenêtre se trouve une grande partie de l'interface de KartoMNT.

Dans KartoMNT, il existe deux types de documents : les documents Karto et les documents MNT. Un document Karto est composé d'une image de carte, de lignes de niveaux et de données de calibration. Il est utilisé pour le travail de construction de Modèles Numériques de Terrain à partir d'une carte. Un document MNT représente lui un MNT, c'est à dire une carte altimétrique d'un rectangle de terrain. Il est issu de la génération à partir d'un document Karto mais peut aussi provenir de sources externes. Nous décrirons en détail ces deux types de documents par la suite.

Une *MainWindowSDI* contient donc un document Karto ou MNT et propose des actions spécifiques à chacun, mais également des actions communes.

Il s'agit en réalité d'un *Container* (*DocumentKartoContainer* ou *DocumentMNTContainer*) qui permet de gérer une liste de fenêtres à l'aide d'un vecteur statique *mainWindowSDIlist*.

Une partie importante de cette classe est donc consacrée à la gestion des fenêtre et à la création de l'interface (réalisée en *Swing*).

1.2. Gestion des actions

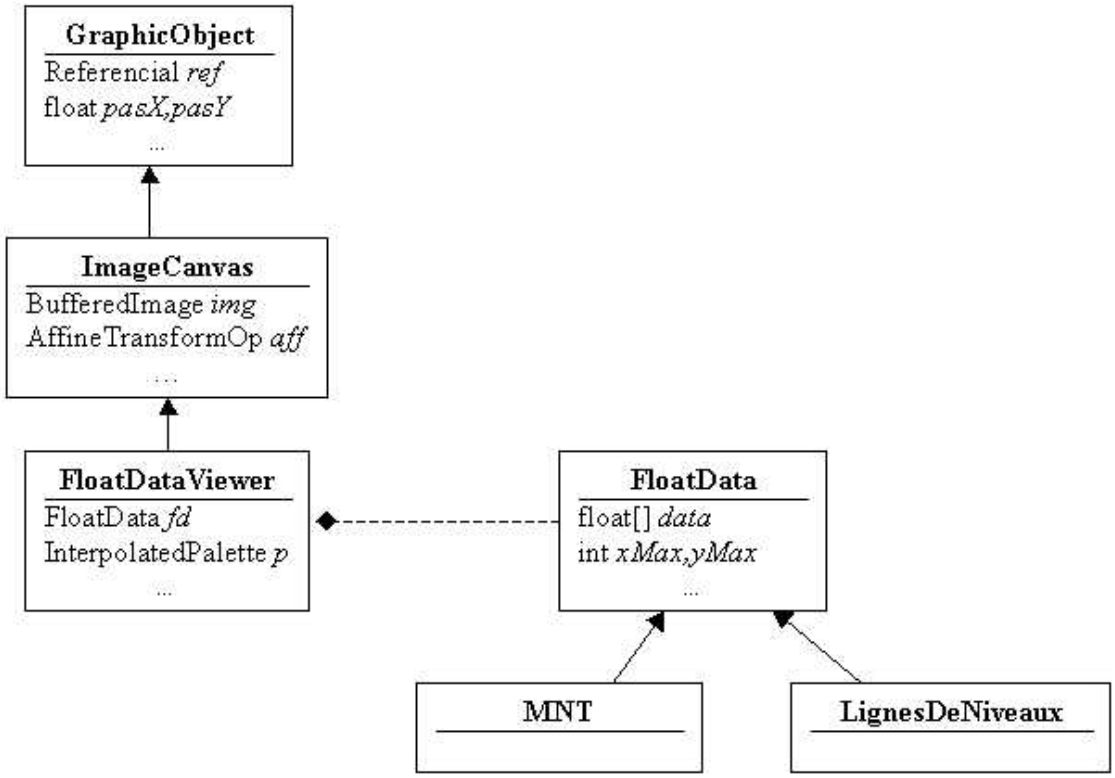
L'interface propose de nombreuses actions, qui sont disponibles dans les menus ou par des boutons (ou les deux à la fois). Chaque action est modélisée par une classe *ActionXXX*, dérivant de la classe *AbstractAction* de Java, ou *AbstractAction2*, redéfinie dans KartoMNT. Les classes *Action* et *AbstractAction* de Java permettent de centraliser une partie de l'état d'une fonctionnalité (nom de la commande icône utilisée, état sélectionné ou non) dans un unique objet. Par exemple, si on a un item de menu et un bouton dans une barre d'outil, ils peuvent tous les deux partager la même icône et le même nom. Si on appelle la fonction *setEnabled* sur l'action les deux composants sont mis a jour. Toutefois, cette représentation ne permet pas de centraliser l'état "selected" utilisé par JCheckBox et JcheckBoxMenuItem. C'est pour cela que les classe *Action2* et *AbstractAction2* ont été ajoutées dans KartoMNT, et permettent par exemple, lorsqu'on clique sur un item du menu, que le bouton correspondant dans la barre d'outils s'enfonce également. Des super classes d'actions permettent de centraliser d'autres fonctionnalités, comme la classe *DocumentAction*, qui permet de désactiver l'action quand il n'y a pas de document ouvert, la classe *DocumentMNTAction*, qui spécifie ce comportement pour les documents MNT, la classe *ActionTool*, qui regroupe les caractéristiques des outils de travail sur les lignes de niveaux, ou encore *ActionGenericDraw* qui concerne plus précisément les outils de dessin.

La plupart des actions sont localisées dans le menu. Pour ajouter une action, il suffit donc, sur le modèle des autres, de rajouter la ligne correspondante dans la méthode *createMenu* de MainWindowSDI et de créer la classe d'action qui convient.

1.3. Structures et représentation des données

Les données sont manipulées à l'aide de la classe *FloatData*. Celle-ci représente un tableau bidimensionnel de flottants. Elle est utilisée pour les MNT pour les lignes de niveaux. Les deux classes *MNT* et *LignesDeNiveaux* étendent donc *FloatData*. Dans les documents (Karto ou MNT), on utilise en fait une classe *FloatDataViewer*, qui permet de visualiser un *FloatData*. En particulier, la classe *FloatDataViewer* a donc un attribut de type *FloatData*, et une palette de couleurs interpolées (*InterpolatedPalette*), qui permet la visualisation des *FloatData* avec des couleurs indexées sur les valeurs de celui-ci. *FloatDataViewer* est une extension de *ImageCanvas*, qui lui même étend *GraphicObject*.

Voici donc comment s'organisent les données :



1.4. Affichage

Les techniques d'affichage sont gérées en majorité par les classes *ImageCanvas* et *GraphicObject* citées précédemment. Elles permettent de manipuler les transformations sur les images.

L'affichage se fait par superposition de calques, qui sont des *FloatDataViewers*, pour les MNT et les Lignes de niveaux, des images, pour les cartes, ou des graphiques simples (lignes de carroyage, points de calibration), sur un même Panel.

Les zooms sont gérés grâce à un objet de type *ObjectReferencial*, en principe commun aux différents calques d'un même document, et à partir duquel est calculé l'affichage de pixels. Selon le choix de l'utilisateur, le calcul des pixels lors du zoom peut être fait :

- par "plus proche voisin" un pixel calculé prend la couleur du pixel connu le plus proche. On a ainsi des carrés qui représentent les pixels grossis.
- par interpolation bilinéaire entre les pixels de couleur connue. On obtient alors un effet lissé quelque soit la résolution.

1.5. Internationalisation

L'internationalisation du logiciel a été réalisée avec les outils d'internationalisation classiques de Java. Une classe *ResourceManager*, appelée de manière statique, est initialisée en début d'application. Elle va chercher dans le fichier *karto.ini* la propriété *"language"*, qui lui donne un descriptif du langage utilisé selon la norme ISO Language Code (ex: "fr"="français", "en"="anglais", etc.). Ces codes peuvent être trouvés à l'adresse :
http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt.

Ce code permet d'initialiser un objet de type *Locale* qui est utilisé en Java pour caractériser une langue ou une région géographique :

```
Locale myLocale = new Locale("fr");
```

On utilise ensuite la classe *ResourceBundle* pour fixer le fichier de ressources qui sera utilisé :

```
ResourceBundle myResources =ResourceBundle.getBundle("KartoMNTResources", myLocale);
```

Dans notre cas, on utilise donc le fichier *kartoMNTResources_fr.properties* . C'est un mécanisme automatique dans Java.

Ce fichier contient, dans la langue correspondante, un identifiant et une chaîne de caractères correspondant à tous les messages, labels, etc.. utilisés dans le logiciel. On fera appel à ce "dictionnaire" à tout moment par :

```
ResourceManager.get(identifiant);
```

Cette ligne est très importante puisqu'elle sera substituée à tous les *Strings* destinés à être lus par l'utilisateur.

Evidemment les dictionnaires doivent être actualisés en conséquence : tout identifiant doit avoir une correspondance dans les fichiers *properties*. Si une langue donnée dans *karto.ini* n'est pas gérée (le dictionnaire correspondant n'existe pas) ou si un des mots n'y figure pas, le *ResourceBundle* va chercher dans la langue par défaut du système et, en dernier recours dans le fichier *kartoMNTResources.properties* (sans identifiant de Locale), qui est utilisé par défaut.

2. Le document Karto

2.1. Généralités

Un document Karto (classe *DocumentKarto)* permet donc, à partir d'une carte scannée, de générer un MNT, en passant par un certain nombre d'étapes.

Au départ, une image doit être ouverte. Le document est ordonné à l'aide de calques : la carte de départ, un calque de résolution égale sur lequel seront dessinées les lignes de niveaux utilisées pour le calcul de MNT, et deux calques pour les données de calibration : les points de calibration et le quadrillage de calibration ("carroyage"). L'opacité et la visibilité des calques peuvent être réglés par l'utilisateur.

On a donc un attribut de type *FloatDataViewer*, qui contient les *LignesDeNiveaux*, un attribut *ImageCanvas* pour l'image de la carte, un ensemble de points de calibrations (*PointCalibration*) stockés dans un *PointsCalibrationSet*, un *Carroyage* pour la grille de carroyage, un objet de type *Proj* qui modélise la projection, un *ObjectReferencial*, etc…

2.2. Lignes de niveaux

Les ligne de niveaux sont, comme on l'a dit un tableau bidimensionnel de données. En d'autre termes, c'est une image dans laquelle sont stockées non pas des informations de couleur mais d'altitudes. Il existe des valeurs spéciales, comme *EMPTY*, qui symbolise le fait qu'un pixel ne soit occupé par aucune ligne, *LINE_WITH_NO_ALTITUDE* qui indique qu'il y a une ligne mais d'altitude indéterminée, ou *TEMP* qui est utilisé dans les algorithmes.

A l'aide d'une palette (*InterpolatedPalette*), ces données sont représentées par différentes couleurs et affichées sous forme d'images.

2.3. Détection des Lignes de niveaux

La première étape pour un utilisateur est de détecter les lignes de niveaux sur la carte qu'il a ouverte.

Il y a plusieurs méthodes de détection de contours : norme du gradient, seuillage dans l'espace de couleur RGB ou YUV.

Norme du gradient :

Pour calculer le gradient on applique un filtre de Sobel horizontal, puis vertical. Il suffit ensuite de prendre la norme. On garde ensuite les valeurs qui dépassent un certain seuil.

Seuillage :

La détection de contours est réalisée par seuillage en testant si les 3 composantes sont comprises à l'intérieur d'un intervalle autour d'une couleur de référence. Cette détection peut se faire dans l'espace de couleur RGB ou YUV. Le passage de l'espace RGB à l'espace YUV se fait grâce à la transformation linéaire suivante

Y = 0.257 * R + 0.504 * G + 0.098 * B

U = -0.148 * R - 0.291 * G + 0.439 * B + 128

V = 0.439 * R - 0.368 * G - 0.071 * B + 128

La composante Y peut s'interpréter comme l'intensité lumineuse

On constate que la détection de contours est de meilleure qualité dans l'espace YUV si on prend un intervalle d'erreur plus important pour Y que pour U et V. Cette méthode permet d'éviter de prendre en compte les ombres de la carte qui correspondent essentiellement à des variations de Y.

A l'issue du seuillage, ou de la norme du gradient, les valeurs des lignes de niveaux détectées sont fixées à *LINE_WITH_NO_ALTITUDE*.

2.4. Outils de travail sur les lignes de niveaux

Outils de dessin

Des outils (crayon, lignes, gomme, etc..) permettent de dessiner des lignes et corriger les erreurs de la détection.

Affinage des lignes de niveaux

Un des problèmes de la détection de contours est que les contours obtenus sont trop épais, ce qui peut provoquer des défauts sur le MNT interpolé. La méthode d'affinage des lignes de niveaux est basée sur le principe suivant :

Soit X un ensemble de points (les lignes de niveaux dans notre cas) On efface de l'ensemble X les points dont l'effacement ne déconnecte pas l'ensemble, c'est à dire ne transforme aucune partie connexe en plusieurs parties connexes. Une région connexe est une région pour laquelle tout couple de points peut être relié par un arc entièrement contenu dans la région.

On fait également en sorte de ne pas effacer les points se trouvant à une extrémité d'une courbe pour éviter de réduire une ligne en un point.

L'algorithme est le suivant :

On numérote les points entourant un point P1 comme ceci :

P3	P2	P9
P4	P1	P8
P5	P6	P7

On suppose qu'un point P est à 1 s'il est dans l'ensemble et à 0 sinon.

On note Z0(P1) le nombre de passages de 0 à 1 dans l'ensemble ordonné P2, P3, P4, P5, P6, P7, P8, P9 et P2.

On note NZ(P) le nombre de voisins de P à 1 (8 voisinage)

Le point P1 est effacé si $2 \leq NZ(P1) \leq 6$

et $Z0(P1) = 1$

et($P2.P4.P8 = 0$ ou $Z(P_2) \neq 1$)

et($P2.P4.P6 = 0$ ou $Z(P_4) \neq 1$)

L'algorithme est appliqué à l'ensemble des points de l'ensemble X. On réitère le processus jusqu'à ce que l'ensemble X ne change plus.

Elimination des points isolés

Lorsque l'on détecte les lignes de niveaux, il y a souvent du bruit. C'est pourquoi une méthode permet d'éliminer les points isolés. La méthode est très simple. Pour chaque point, on regarde ses 8 voisins et si aucun des 8 voisins ne fait partie d'une ligne le point est effacé

Raboutage semi-automatique

Il existe également d'autres outils en particulier un rabouteur semi-automatique de lignes de niveaux. Celui-ci détecte lorsque la souris passe sur une ligne, l'extrémité de celle ci, et la sélectionne au clic de l'utilisateur. Si celui-ci se déplace près d'une autre ligne, son extrémité est sélectionnée selon le même principe, et si l'utilisateur clique une deuxième fois, une ligne droite est tracée entre le deux extrémités.

L'outil est implanté par la classe *ToolSemiAuto*. Pour détecter les extrémités, il considère qu'une extrémité d'une ligne de niveaux est un pixel de la ligne de niveaux ne possédant qu'un seul voisin possédant la même altitude que lui. Cela est vrai seulement si les lignes de niveaux ont été affinées auparavant. Après avoir sélectionner la première extrémité, on recherche la plus proche extrémité autour du curseur de la souris et on affiche une ligne temporaire joignant les deux extrémités. Un clic à ce moment là rend la ligne réelle.

2.5. Détermination des altitudes

Pour déterminer l'altitude des lignes, l'utilisateur a le choix entre les sélectionner une par une et leur fixer des altitudes, ou déterminer un axe, avec une altitude initiale et un incrément, afin de déterminer automatiquement de toutes les lignes croisant cet axe.

Lorsque l'altitude d'un point est fixée, celle-ci se répand sur toute la ligne de niveaux par propagation.

Il est possible de donner l'altitude en des points isolés (sommets).

2.6. Calibration

La calibration est une opération qui consiste à mettre en correspondance les coordonnées de l'espace image (ici les pixels d'une carte de randonnée) avec les coordonnés de l'espace réel. Dans le cas qui nous intéresse, cela revient à déterminer, pour chaque point de la carte, ses coordonnées géographiques. Cela se fait par le calcul d'une matrice de projection.

KartoMNT utilise les sources du logiciel Karto, qui permettent déjà d'effectuer la calibration (calculer la matrice de projection).
L'utilisateur sélectionne à la souris des points de référence (souvent visibles sur la carte, marqués par de grandes lignes), entre leurs coordonnées géographiques, et lance la calibration.

La calibration n'est pas nécessaire pour la détection des lignes de niveaux mais sera nécessaire pour l'interpolation car les dimensions du MNT et le pas d'échantillonnage seront donnés en coordonnées terrain. On a donc une relation de projection connue entre les images et données des lignes de niveaux et les dimensions et positions réelles du terrain.

2.7. Génération de MNT

Une fois qu'on dispose de toutes les lignes de niveaux et de toutes les altitudes, il est possible de générer le MNT en interpolant au niveaux des points entre les lignes de niveaux.

La méthode utilise une **Equation aux Dérivées Partielles :**

Soit L l'ensemble des points appartenant aux lignes de niveaux et E l'ensemble des points de la carte (domaine ouvert)

Les lignes de niveaux vont être interpolées par la solution de l'équation :

$$\forall (x,y) \in E - L, \Delta\Psi(x,y) = 0 \quad (1)$$

avec la condition aux limites :

$$\forall (x,y) \in L, \Psi(x,y) = f(x,y) \quad (2)$$

Il est nécessaire d'avoir des conditions aux limites aux bords de la carte. Comme on ne peut pas imposer d'altitude au bord de la carte on impose que le flux soit nul sur la frontière de la carte (sur $\partial E - L$ *) :*

$$grad\Psi\vec{n} = 0 \quad \textit{avec } \vec{n} \textit{ normale au domaine E}$$

Une des propriétés de l'équation (1) est qu'elle vérifie le principe du maximum.

Principe du maximum :

Soit Ω *un ouvert borné de* $\Re^n$, $\partial\Omega$ *sa frontière et* Ψ *une fonction continue sur* $\overline{\Omega}$ *et de classe* C^2 *sur* Ω

si $\forall x \in \Omega, \Delta\Psi(x) = 0$ *alors* $\forall x \in \Omega, \min_{x \in \partial\Omega} \Psi(x) \le \Psi(x) \le \max_{x \in \partial\Omega} \Psi(x) = 0$

Donc si on a un domaine compris entre deux lignes de niveaux l'ensemble des valeurs prises à l'intérieur du domaine est comprise entre la valeur minimale et la valeur maximale atteinte sur le bord du domaine. Ceci est bien une propriété qu'on attend d'une interpolation

Nous allons vérifier dans un cas simple que cette équation réalise bien une interpolation.

Considérons le cas unidimensionnel :

$$\Delta\Psi = \frac{\partial^2\Psi}{\partial x^2} = 0 \quad \textit{sur }]A,B[$$
$$\Psi(A) = a \textit{ et } \Psi(B) = b$$

on a $\frac{\partial\Psi}{\partial x} = c$ *donc* $\Psi(x) = cx + d$

donc la solution réalise bien une interpolation linéaire

Dans le cas général bidimensionnel, il est difficile de démontrer que l'équation réalise bien une interpolation car il faudrait déjà définir ce qu'est une interpolation en 2D (ce qui peut être défini de plusieurs manière) Nous allons ici définir l'interpolation comme la solution Ψ *vérifiant les équations de (1) et (2), ce qui nous dispense de démonstration.*

L'équation (1) va être résolue par une méthode de point fixe :

$$\Delta\Psi(x,y,t) = \frac{\partial\Psi}{\partial t} \quad (2)$$

La solution de cette équation converge vers la solution de l'équation (1) ($\frac{\partial \Psi}{\partial t} = 0$ à l'équilibre)

Cette EDP est discrétisée par une méthode de type différence finie

Méthode numérique de résolution

Approximation de la dérivée seconde :

$$\frac{\partial^2 \Psi(x,y)}{\partial x^2} \approx \frac{\frac{\Psi(x+d,y)-\Psi(x,y)}{\Delta x} - \frac{\Psi(x,y)-\Psi(x-d,y)}{\Delta x}}{\Delta x}$$

$$\frac{\partial^2 \Psi}{\partial x^2} \approx \frac{\Psi(x+d,y)+\Psi(x-d,y)-2.\Psi(x,y)}{\Delta x^2}$$

Approximation du Laplacien :

$$\Delta\Psi(x,y) = \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2}$$

$$\Delta\Psi(x,y) \approx \frac{\Psi(x+\Delta x,y)+\Psi(x-\Delta x,y)-2\Psi(x,y)}{\Delta x^2} + \frac{\Psi(x,y+\Delta y)+\Psi(x,y-\Delta y)-2.\Psi(x,y)}{\Delta y^2}$$

Développement limité par rapport au temps en t1

$$\Psi(x,y,t+\Delta t) = \Psi(x,y,t) + \frac{\partial \Psi(x,y,t)}{\partial t'}(t_1).\Delta t + o(\Delta t)$$

En substituant :

$$\Psi(x,y,t+\Delta t) \approx \Psi(x,y,t) + \frac{\Delta t}{\Delta x^2}.(\Psi(x+\Delta x,y,t)+\Psi(x-\Delta x,y,t)-2\Psi(x,y,t))$$

$$+\frac{\Delta t}{\Delta y^2}(\Psi(x,y+\Delta y,t)+\Psi(x,y-\Delta y,t)-2\Psi(x,y,t))$$

soit

$$\Psi(x,y,t+\Delta t) \approx \left(1-2\Delta t\left(\frac{1}{\Delta x^2}-\frac{1}{\Delta y^2}\right)\right)\Psi(x,y,t) + \frac{\Delta t}{\Delta x^2}.(\Psi(x+\Delta x,y,t)+\Psi(x-\Delta x,y,t))$$

$$+\frac{\Delta t}{\Delta y^2}(\Psi(x,y+\Delta y,t)+\Psi(x,y-\Delta y,t))$$

On peut montrer que ce schéma est stable sous la condition

$$\frac{\Delta t}{\Delta x^2} + \frac{\Delta t}{\Delta y^2} \leq \frac{1}{2}$$

L'équation précédente peut se réécrire sous la forme (en supposant $\Delta x = \Delta y$) :

$$\Psi_{i,j,t+1} = (1-2a)\Psi_{i,j,t} + a.(\Psi_{i+1,j,t} + \Psi_{i,j+1,t} + \Psi_{i-1,j,t} + \Psi_{i,j-1,t})$$

C'est cette équation qui sera utilisée pour actualiser les points qui ne font pas partie des lignes de niveaux.

Les points sur les lignes de niveaux n'ont pas besoin d'être actualisés. Les points aux bords de la carte seront actualisés d'une manière différente. Ils seront actualisés par simple copie (condition de flux nul):

$$\Psi_{i,0,t+1} = \Psi_{i,1,t}$$
$$\Psi_{i,M,t+1} = \Psi_{i,M-1,t}$$
$$\Psi_{0,j,t+1} = \Psi_{1,j,t}$$
$$\Psi_{iN,t+1} = \Psi_{i,N-1,t}$$

Le problème avec cette méthode est que le temps de calcul est assez long, car il faut attendre que les altitudes " diffusent " à partir des lignes de niveaux, c'est pour cela que nous avons utilisé une méthode multi-résolution (inspiré des méthodes multigrilles) Cette méthode consiste à résoudre d'abord l'équation sur un domaine de taille réduite puis à interpoler la solution pour obtenir une solution à la résolution supérieure. On a une suite de grilles I_0, I_1,..I_N . On passe d'une grille I_k à la suivante I_{k+1} en ne gardant qu'un échantillon sur 2. On résout d'abord l'équation sur la grille I_N puis on interpole la solution obtenue à la résolution supérieure sur la grille I_{N-1}. On part ensuite de cette solution comme condition initiale pour résoudre l'équation sur la grille I_{N-1}. Comme la condition initiale est proche de la solution la convergence est plus rapide. On continue de la même façon jusqu'à la résolution maximale sur la grille I_0

La méthode précédente est donc appliquée dans la classe *LignesDeNiveaux*. Celle-ci applique cet algorithme de diffusion des altitudes après avoir déterminé la portion "cible" des lignes à interpoler (la région sélectionnée par l'utilisateur).

Si le pas d'échantillonnage du MNT (choisi par l'utilisateur) est supérieur à celui des lignes de niveaux (résolution de l'image), c'est à dire qu'un point du MNT correspond à plusieurs points des lignes, alors l'altitude en un point est calculée en faisant la moyenne de tous les points des lignes de niveaux correspondants.

Dans le cas contraire, on calcule une interpolation entre les points des lignes de niveaux pour déterminer avec précision les altitudes sur lesquels on va réaliser l'interpolation du MNT.

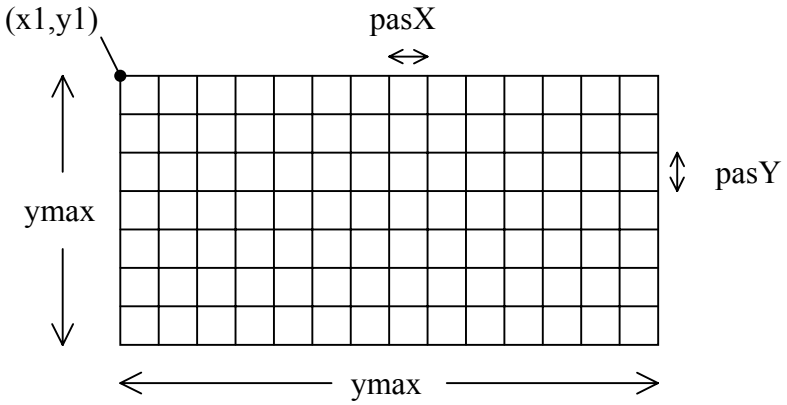
Dans tous les cas il y a donc deux étapes.

Afin d'éviter les effets de bords indésirables, le calcul est fait sur toute la carte puis le MNT est redécoupé aux dimensions voulues. En effet les points interpolés sur les bords sont calculés à partir des données d'un seul de leurs cotés et sont donc influencés par un seul des versants de la pente. C'est inévitable sur les bords de la carte mais pas si le MNT est sélectionné en laissant une marge tout autour.

3. Le Document MNT

3.1. Les MNT

Un MNT est une carte des altitudes sur un rectangle de terrain donné. Il possède donc comme attribut une origine (*x1,y1*), des pas d'échantillonnage (*pasX,pasY*) qui sont les distances séparant deux points de la grille formée par les données. Ces attributs sont des attributs permettant de mettre en rapport les données avec le terrain réel. Le *MNT* possède également les attributs de la classe *FloatData* dont il hérite : tableau de données et dimensions du tableau (*xMax,yMax*).



3.2. Visualisation du MNT

Le MNT est visualisé dans un *DocumentMNT* grâce à un *FloatDataViewer* dont il est l'attribut, et qui établit une correspondance entre les altitudes et les couleurs d'une palette (*InterpolatedPalette*). Par défaut celle-ci est étalonnée entre les altitudes minimales et maximales du MNT sur 5 couleurs, mais il est possible d'en ajouter.

Superposition des lignes de niveaux

Le MNT possède un attribut de type *Lignes*, qui est une extension de *FloatDataViewer*. Il permettra donc de visualiser un tableau de données selon le même principe que pour les lignes de niveaux, mais simplifié.

Le tableau (*FloatData*) de *Lignes*, est rempli à partir d'un objet *LignesDeNiveaux*, selon une palette simplifiée : pas de couleur (transparent dans les zones "*EMPTY*", et une valeur (1) que l'on associe à la couleur noire pour les lignes, quelle que soit leur altitude. Ainsi on peut superposer sur la représentation du MNT une carte des lignes de niveaux (noires). Lorsqu'un MNT est généré, les lignes de niveaux qui ont permis sa création sont passées à l'attribut *Lignes* du nouveau MNT. Ainsi, on peut superposer (si l'utilisateur le choisit) les lignes sur le MNT.

Ces données sur les lignes de niveaux ne font pas partie des données du MNT à proprement parlé et ne sont pas sauvegardées lors de l'exportation du MNT. Un MNT importé aura donc son attribut *Lignes* de valeur *null* et cette opération de superposition ne lui est pas permise.

4. Opérations sur les MNT

4.1. Les différents formats

KartoMNT gère l'ensemble des formats à l'aide de la classe *GestionnaireMntFormat* dédiée à ce but. Dès son chargement, elle commence par vérifier s'il existe des classes de formats compilées définies par l'utilisateur dans le répertoire *kartoMNT*. On utilise pour cela la fonction *listFiles* de *File* qui retourne la liste des fichiers et répertoires contenue dans un répertoire. Puis, on cherche parmi les fichiers listés ceux qui sont des .class valides commençant par Format. Enfin, on appelle la fonction *loadUrl* de *MNTClassLoader*. Celle-ci utilise *URLScriptClassLoader* pour charger la classe dynamiquement.

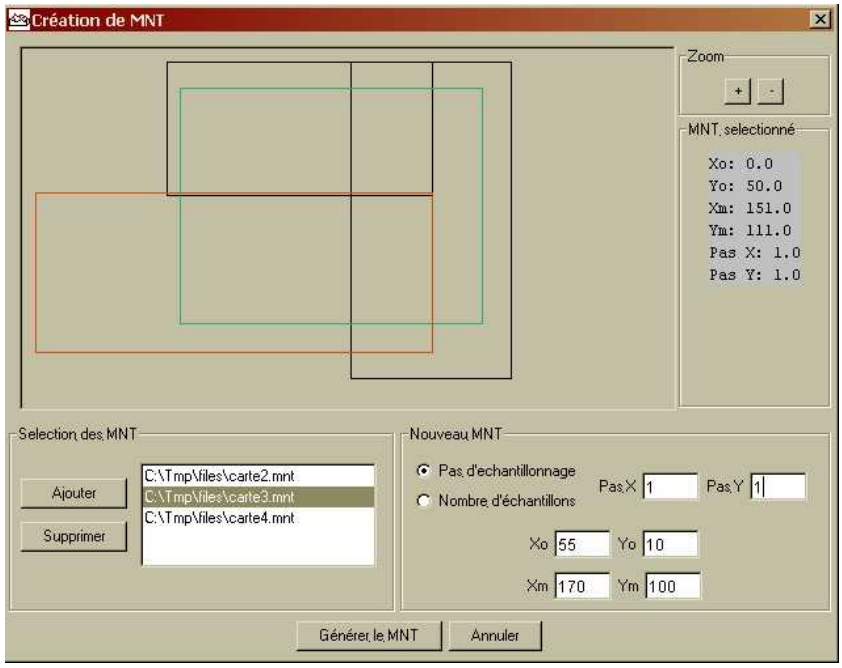
Ensuite, *GestionnaireMntFormat* charge les classes se trouvant dans le Jar. Comme le fonctionnement de chargement dans un Jar est un peu différent du chargement dans un répertoire normal, les deux classes n'ont pu être utilisées. Il faut, en effet, tout d'abord obtenir l'URL du fichier se trouvant dans le jar par la méthode *GestionnaireMntFormat.class.getResource,* ouvrir une JarURLConnection grâce à *url.openConnection(),* créer un JarFile par *getJarFile* de *JarURLConnection* pour enfin pouvoir enfin obtenir la liste des fichiers contenus dans le Jar (JarFile possède une fonction *entries* retournant une *Enumeration* de *JarEntry* dont on peut avoir le nom par getName…). On peut alors vérifier les .class valides commençant par « MntFormat » et charger la classe à partir de son nom à l'aide de la méthode statique *forName* de *Class* retournant un objet de type *Class* à partir duquel on peut construire dynamiquement un objet en utilisant *newInstance*. Simple, non ;-)

Une liste de formats est ainsi créée. Pour obtenir un *MntFormat* à partir du nom de fichier, une fonction statique *donneFormat* a été rajoutée. Elle vérifie simplement dans la liste des *MntFormat* si l'un d'entre eux gère l'extension du fichier donné.

Une autre fonction permet d'initialiser un *JFileChooser* avec l'ensemble des formats de MNT connus en import ou en export selon le type de *JFileChooser*.

4.2. Travail sur les MNT

Le dialogue *FusionMNTDialog* permet des opérations de fusion, découpage, ré-échantillonnage sur les MNT. Celui-ci contient un *Vector* d'objets de type *MNT*. L'utilisateur peut charger des MNT à partir de fichiers, ceux-ci sont stockés dans le vecteur. Une *JList* permet de contrôler les vecteurs enregistrés, on peut voir leur caractéristiques dans un panel d'information sur le dialogue, les supprimer, etc..



Un panel d'affichage (*JPanelMNT*), contient lui les rectangles correspondants aux MNT chargés et permet de les visualiser pour contrôler leurs positions et dimensions respectives.

L'utilisateur peut saisir les caractéristiques du MNT qu'il souhaite générer à partir du ou des MNT chargé(s) (position, dimensions, pas d'échantillonnage).

On fait alors appel à un *MNTGenerateur* pour créer le nouveau MNT. Un *MNTGenerateur* est construit à partir d'un *Vector* de *MNT*, d'une origine (*x1,y1*), dimension (*xmax, ymax*) et de pas d'échantillonnage (*pasX* et *pasY*).

A partir de ces données, il crée un nouveau MNT aux caractéristiques citées précédemment et aux altitudes nulles. Reste alors à déterminer l'altitude en chacun des points de ce MNT. Pour cela, on parcourt pour chaque point l'ensemble des MNT dont on dispose afin de savoir lequel pourra nous renseigner sur l'altitude de ce point. Si aucun MNT ne convient, une altitude nulle par défaut est donnée au point. Dans le cas contraire, on calcule l'altitude au point donné par interpolation bilinéaire entre les quatre points d'altitudes connues qui l'entourent.

On arrive ainsi à reconstruire un MNT complet.

5. Visualisation 3D

5.1. Java3D

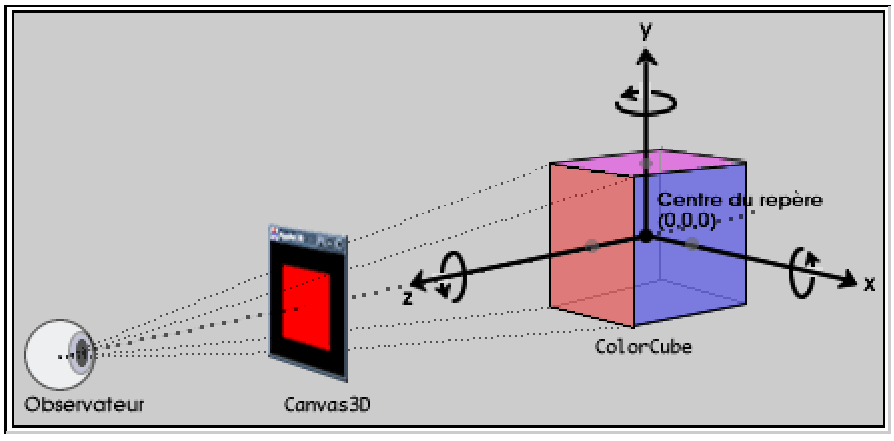
Java3D est une API orientée objet, extension du JDK (Java Development Kit), destinée à la création de scènes en 3D. Elle offre une bibliothèque d'objets de très haut niveau permettant la construction et la manipulation de scènes 3D. Elle bénéficie des avantages intrinsèques à la programmation Java (portabilité, indépendance par rapport au matériel et système d'exploitation, exécution dans un butineur).

L'API Java 3D reprend les idées des principales APIs existantes. Elle synthétise les meilleures idées des APIs bas niveau (Direct3D, OpenGL, ...) et reprend le concept de systèmes basés sur la construction d'un graphe de scène. Elle introduit de nouvelles fonctionnalités telles que le son 3D et la compression des géométries.

Le graphe de scène fournit un moyen simple et flexible de représenter une scène en 3D. Il regroupe les objets à visualiser, les informations de visualisation et les éventuels outils d'interaction entre l'utilisateur et les objets.

Le système de coordonnées

Par défaut, Java 3D utilise:
- le système de la main droite comme système de coordonnées
- l'axe Z positif orienté vers l'observateur



Il faut donc faire attention, car ce repère est différent du repère que l'on souhaite utiliser dans notre application : il semble plus naturel que les données altimétriques du MNT se dirigent vers le haut de la scène. Ainsi de nombreux changements ont du être nécessaires, passant d'un repère à l'autre, notamment à chaque fois que l'utilisateur veut rentrer une nouvelle position ou rotation de la caméra.

Présentation du graphe de scène

Un univers virtuel (*VirtualUniverse*) est défini comme un espace tridimensionnel dans lequel sont insérées une ou plusieurs scène(s) (*Locale*).

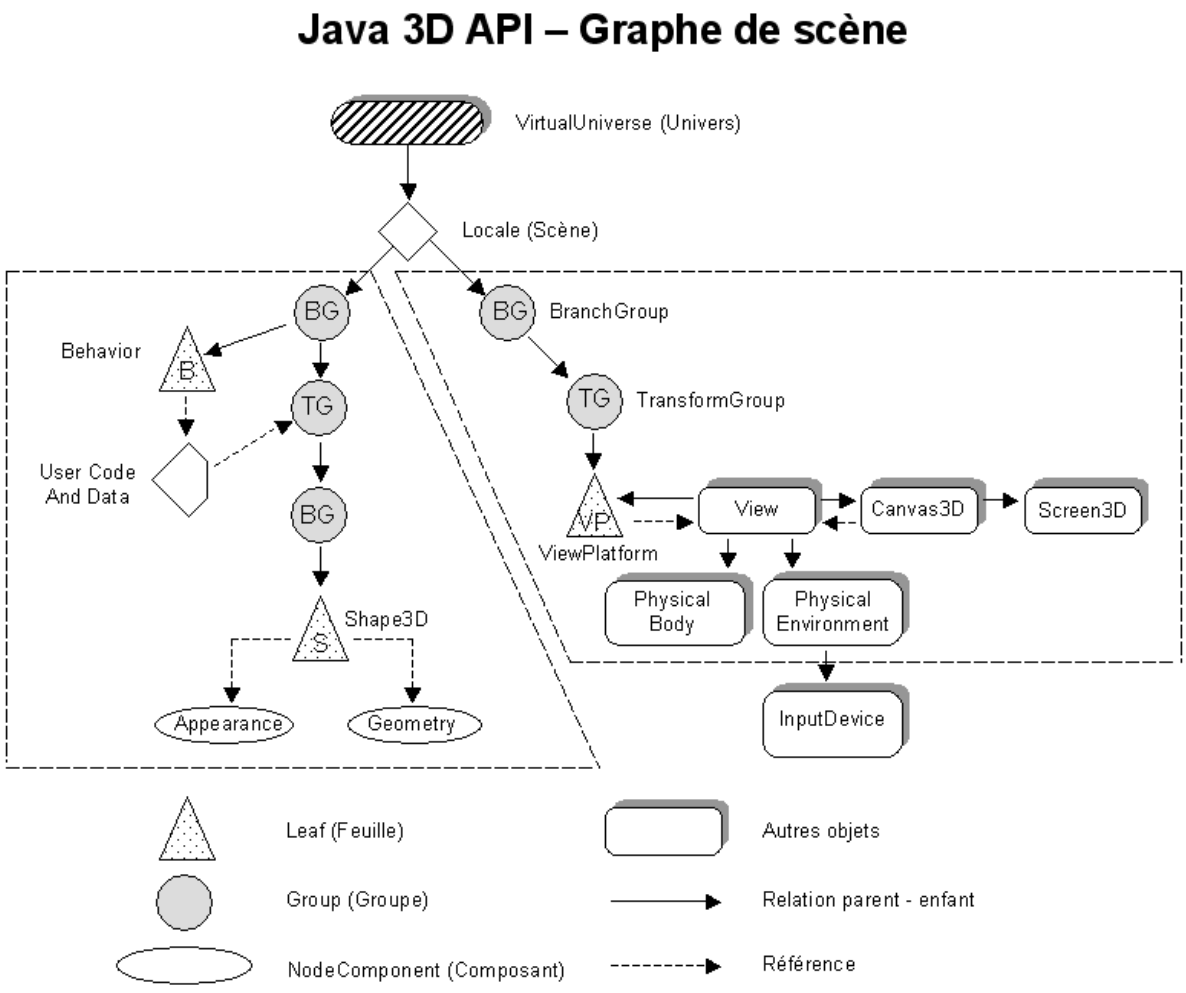
Construire une scène consiste à associer une branche de volume et une branche de visualisation.

La branche de volume constitue la partie gauche du graphe de scène (ci-dessous) et elle va contenir les différents objets physiques (Géométrie, Son, Texte) à visualiser.

La branche de visualisation forme la partie droite de la scène (ci-dessous) et elle contient les outils nécessaires à visualiser l'espace de volume (position du point de vue, projection, style d'affichage, ...). Afin de créer une visualisation plus réaliste, Java 3D permet également de tenir compte des caractéristiques physiques de la tête de l'utilisateur (PhysicalBody) (position des yeux, écartement, ...) et de l'environnement sensoriel (PhysicalEnvironment).

Chaque branche est un ensemble structuré de nœuds (*Node*) et de composants de nœuds (*NodeComponent*) liés par des relations. Le point de départ d'une branche est obligatoirement un objet *BranchGroup*.

La plupart des applications 3D n'utilisent qu'une branche visualisation (un seul point de vue) et qu'une seule branche de volume mais il est possible de concevoir des scènes complexes dans lesquelles seront intégrées plusieurs branches de volume et plusieurs branches de visualisation. Cela permet de créer des scènes multi-représentations et multi-points de vue.

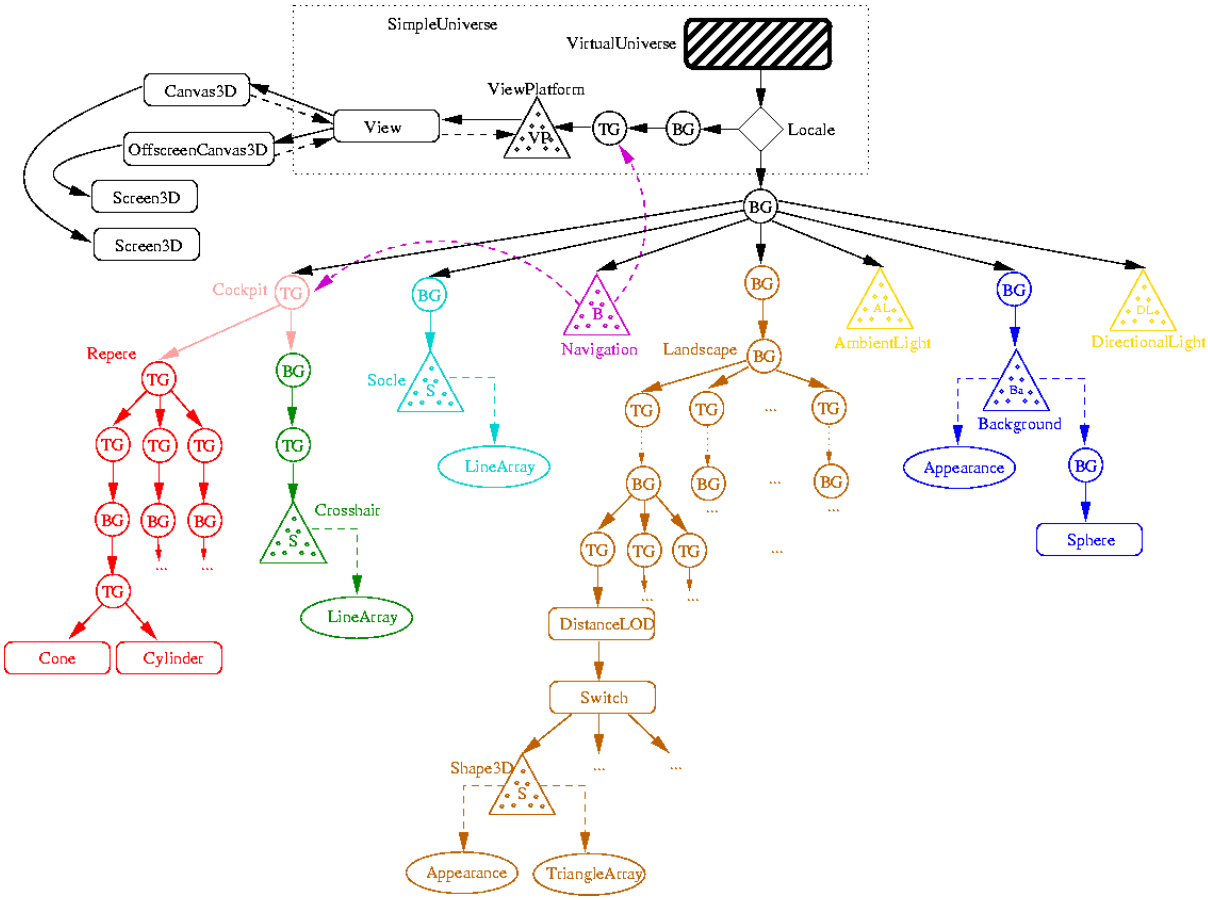


5.2. Aperçu de l'Application

Voici un bref descriptif des classes du visualisateur 3D :

- *kartoMNT.visu3D.BoucheTrouDialog* : fenêtre de dialogue permettant de boucher les trous dans la texture dus à la calibration.
- *kartoMNT.visu3D.Crosshair* : la croix verte centrale du cockpit
- *kartoMNT.visu3D.DialogVisu3DKartoMNT* : fenêtre de dialogue initiale gérant les paramètres du visualisateur 3D
- *kartoMNT.visu3D.JMainFrame2* : fenêtre de l'application principale, elle transforme une JApplet en JFrame.
- *kartoMNT.visu3D.LandScape* : le MNT en 3D
- *kartoMNT.visu3D.Navigation* : classe gérant la navigation de la caméra
- *kartoMNT.visu3D.OffScreenCanvas3D* : canvas 3D de capture d'écran haute qualité
- *kartoMNT.visu3D.Sequenceur* : thread se chargeant de la capture de séquences et de films
- *kartoMNT.visu3D.Socle* : socle du MNT
- *kartoMNT.visu3D.Variables* : classe utilitaire contenant l'ensemble des paramètres de l'application
- *kartoMNT.visu3D.Visu3DKartoMNT* : classe principale contenant l'interface, et le visualisateur.
- *kartoMNT.visu3D.gif.AnimatedGifEncoder* : encodeur de gifs animés
- *kartoMNT.visu3D.gif.GifDecoder* : décodeur de gifs
- *kartoMNT.visu3D.gif.LZWEncoder* : algorithme d'encodage de gif
- *kartoMNT.visu3D.gif.NeuQuant* : algorithme de quantisation du gif

Voici le graphe présentant l'application du point de vue Java3D.



Pour simplifier la réalisation, la classe *SimpleUniverse* a été utilisée. Cette classe permet au programmeur d'éviter la programmation de la branche de visualisation. Le constructeur d'un objet *SimpleUniverse* crée un graphe de scène contenant les objets *VirtualUniverse*, *Locale* et toute la branche de visualisation. L'univers est créé dans le constructeur de *Visu3DKartoMNT*.

La branche marron du graphe montre l'arborescence du *Landscape*. Celle-ci est rajoutée par la fonction *addLandscape* de *Visu3DKartoMNT*. La fonction crée un objet *LandScape* initialisé avec le MNT à afficher, construit la texture à appliquer sur le MNT, la donne à *LandScape* puis appelle la fonction *LODModel* chargée d'élaborer le *BranchGroup* contenant le MNT en 3D avec niveaux de détails. La méthode *LODModel* se contente d'appeler la fonction *LODSubdivide* qui est une fonction récursive récurant sur le nombre de divisions courant du LOD. S'il est plus grand que 0, on divise le tableau de données altimétriques du MNT en quatre sous-tableaux de tailles (approximativement) identiques et on réitère sur chacun de ces tableaux en diminuant le nombre de subdivisions courant de 1. Lorsque celui-ci atteint 0, on construit le LOD à l'aide de la fonction *createLOD*.

La méthode créée un objet *DistanceLOD* à partir d'un tableau des distances à laquelle on doit changer la qualité du MNT et d'un point dans l'espace 3D servant d'origines aux distances. On donne à l'objet *DistanceLOD,* également, un objet *Switch* qui contient l'ensemble des *Shape3D* représentant la triangulation du MNT à chaque niveau de détails.

C'est la fonction *convertToArray* qui se charge de créer la triangulation que l'on donne en paramètre au *Shape3D*. Celle-ci prend en paramètre un tableau de données altimétriques et le transforme en *TriangleArray*. Elle calcule ainsi pour chaque triangle les coordonnées 3D, les coordonnées de texture et les normales associées à ses sommets.

La branche bleu foncé du graphe représente l'arborescence du fond. Celui-ci constitue une immense sphère texturée englobant l'ensemble du monde. C'est la fonction *addBackground* qui le créée. Si l'utilisateur ne désire pas avoir de texture de fond, alors la branche n'est pas construite.

Les branches jaunes rajoutent la lumière au monde 3D. Il existe deux lumières, une source de lumière ambiante (pas très lumineuse) et une source lumière directionnelle simulant le Soleil (bien plus lumineuse). Les deux sources sont rajoutées par la fonction *addLights*.

La branche magenta désigne la classe *Navigation* s'occupant de la gestion de la souris, du clavier et des boutons de navigations de l'interface du visualisateur. C'est la méthode *addNavigation* qui a le rôle de son intégration dans le graphe. *Navigation* modifie les *Transform3D* contenus dans les *Transform3D* de la caméra et du cockpit.

5.3. Mécanisme de LOD

Plusieurs sortes de mécanismes de LOD existent. En particulier, on distingue ceux fondés :

- sur la distance (plus un objet est loin, moins ces détails sont perceptibles) ;
- sur la vitesse (un objet en mouvement est moins bien perçu par le système visuel humain qu'un objet fixe) ;
- et sur l'incidence (dans certains cas, un objet peut avoir une forme particulière selon l'angle de vue, par exemple une porte ou un tube vus de profil).

Cependant le seul mécanisme de LOD, fourni par Java3D (DistanceLOD), est fondé sur la distance de la caméra à l'objet.

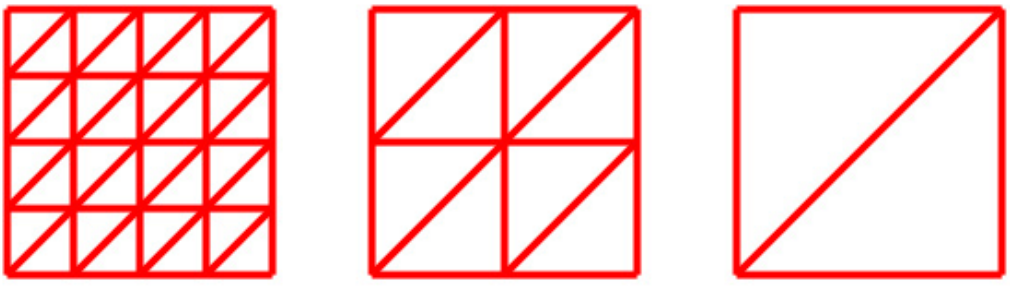
Chaque objet LOD a un, ou plusieurs, objets Switch comme cible. Un objet Switch est un groupe spécial comprenant zéro, un ou plusieurs fils dans le graphe de scène dont un seul au maximum peut être rendu dans la scène 3D à la fois. Avec un DistanceLOD, la sélection du fils du Switch est contrôlée par la distance de l'objet contenant le DistanceLOD à la vue en fonction d'un ensemble de distances seuils.

Les distances seuils sont spécifiées dans un tableau commençant avec la distance maximale à laquelle le premier fils du Switch sera utilisé. Il s'agit typiquement de l'objet avec la plus grande résolution. Lorsque la distance de l'objet contenant DistanceLOD est plus grande que la première distance seuil, le deuxième fils du Switch est utilisé. Les distances seuils doivent absolument être préalablement triées dans l'ordre croissant. De plus, il doit y avoir une distance seuil de moins dans le tableau qu'il n'y a de fils dans le Switch.

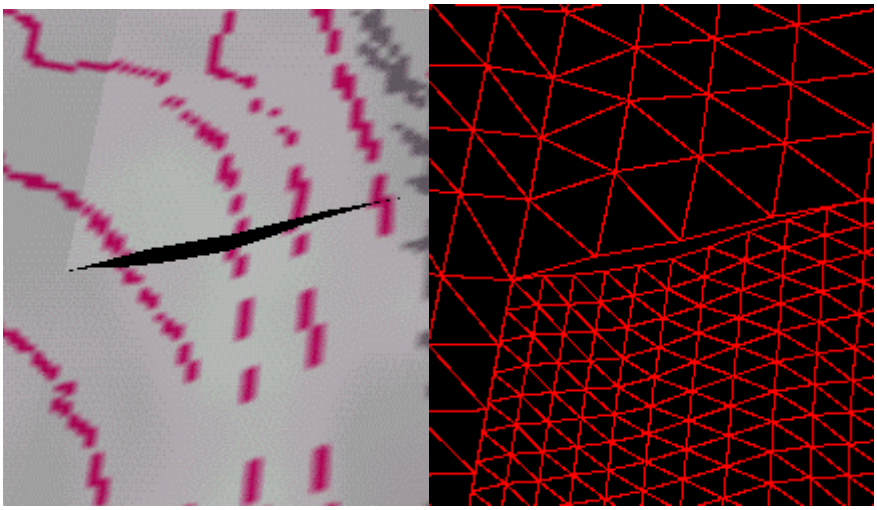
Ainsi, Java3D simplifiait beaucoup la gestion du LOD. Il restait cependant à :
- créer l'ensemble des résolutions différentes à placer dans le Switch ;
- subdiviser le MNT en plusieurs objets LOD différents. En effet, si l'on ne faisait pas cela, c'est l'ensemble du MNT qui changerait de résolution alors que ce que l'on désire, c'est que les parties éloignées du MNT soient affichées avec une résolution plus faible que les plus proches.

Le paramètre *LODDepth* contient le nombre de subdivisions par deux du MNT et c'est la fonction *LODSubdivide* qui se charge de découper le MNT en plusieurs sous-parties. On l'initialise avec la matrice de données altimétriques du MNT et le paramètre *LODDepth* comme nombre de subdivisions courant. Si le nombre de subdivisions courant est plus grand que zéro, la fonction construit quatre sous-matrices contenant les données altimétriques de la matrice donnée en paramètre. Comme la taille de la matrice de départ n'est pas forcément une puissance de 2, il faut gérer les cas impairs. La fonction réitère alors la fonction *LODSubdivide* sur chacune des 4 sous-matrices en diminuant le nombre de subdivisions courant de 1. Lorsque le nombre de subdivisions courant est égale à zéro, c'est alors que l'on peut créer l'objet LOD par la méthode *createLOD*.

Le paramètre *LODLevel* donne le nombre de niveaux de détails qu'il faut construire par sous-parties de MNT. Il a fallu déterminer une manière de construire les différents niveaux de détails. Celle qui a été choisie est sans doute la plus simple : il s'agit de la technique du sous-échantillonnage. A chaque étape, nous récupérons une valeur sur deux en X et Y des données altimétriques de la version précédente, divisant ainsi le nombre de polygones par quatre à chaque sous-échantillonnage effectué.



Chaque valeur récupérée est bien une valeur originale du MNT : nous n'effectuons absolument aucune moyenne. Cette technique fonctionne bien sur des zones continues du MNT. Cependant, de petites zones avec des grandes différences de hauteurs peuvent tout simplement disparaître avec cette méthode. Autre problème, des zones voisines possédant des niveaux de détails différents ont de grandes chances de provoquer des "craquement" du MNT.



Pour réaliser le LOD, la méthode *createLOD* construit le *Switch* et lui attache l'ensemble des niveaux de détails. Les matrices altimétriques sont sous-échantillonnées à l'aide de la fonction *newValues*. La triangulation des différentes matrices altimétriques est assurée par la méthode *convertToArray*. *createLOD* crée également la table des distances seuils et calcule la position du point central du LOD nécessaire à l'élaboration du *DistanceLOD* de la sous-partie du MNT. La table des distances seuils est élaborée à partir de la variable *LODDistance* que l'on rajoute à intervalle régulier. Pour augmenter ou diminuer la qualité du MNT à partir de l'interface, celle-ci appelle la méthode *changeQuality* qui change tout simplement la variable *LODDistance* et recalcule la table des distances seuils associée à chaque *DistanceLOD*.

convertToArray commence par créer le TriangleArray et lui rajoute l'ensemble des triangles et des cordonnées de texture associées à chaque sommet du triangle, la principale difficulté étant de réussir à les calculer correctement en fonction du nombre de subdivisions et du niveau de détails. Elle peut ensuite calculer les normales associées en chaque point en faisant la moyenne des normales des faces contenant ce point.

5.4. Texturage Calibré

L'élaboration de la texture calibrée s'effectue dans la méthode *initTextureCalibree* de *Visu3DKartoMNT*. Cette fonction prend en paramètre une *DocumentKarto* préalablement construite à partir du .karto contenant la texture calibrée. Facilitant grandement le travail, la fonction statique *MathUtil.getEnglobingRect* retourne le rectangle calibré qui englobe la texture. Le MNT étant aussi calibré, on peut donc retrouver le rectangle utile dans la texture qui sera appliqué sur le MNT. Si le rectangle est valide (si sa largeur et sa hauteur sont bien plus grandes que 0), on récupère les pixels utiles dans un tableau à l'aide de *PixelGrabber*. Si ce rectangle recouvre entièrement le MNT, on a fini et on peut transformer le tableau de pixels en image à l'aide de *MemoryImageSource* et de *createImage* de la classe *Toolkit*. Sinon, il faut auparavant boucher les trous et on demande à l'utilisateur comment il désire les boucher.

S'il choisit de les boucher par une couleur, on construit le tableau de pixels devant contenir la texture du MNT par la couleur sélectionnée. Si l'utilisateur préfère les boucher par une texture, on la charge et on remplit le tableau cette fois-ci avec les données de la texture que l'on répète si besoin est. Une fois cela fait, il suffit de trouver où doit s'appliquer le rectangle de texture calibrée utile dans la texture finale de MNT et de rajouter effectivement la texture. On a ainsi construit la texture calibrée comme il fallait. Seulement comme celle-ci peut parfaitement avoir une forme allant du carré parfait au rectangle fortement allongé, il arrivait que le *TextureLoader* de Java3D n'arrive pas à la charger ! Il a donc fallu retailler l'image afin qu'elle ait une taille que le *TextureLoader* puisse supporter, typiquement 512x512. Le retaillage s'effectue à l'aide de la classe *BufferedImage* qui peut construire un *Graphics2D*. Cette classe permet d'effectuer une interpolation Bi-Cubique de l'image afin d'obtenir une meilleure qualité d'image. L'image retaillée est ensuite acceptée par le *TextureLoader* sans aucun problème.

5.5. Capture d'Ecran

Capture d'Ecran Normale

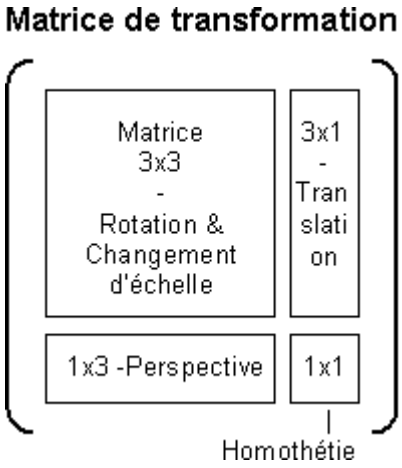
Le code réalisant la capture d'écran se trouve dans la fonction actionPerformed qui est appelée lorsque l'utilisateur appuie sur un bouton. Dans un premier temps, il avait été trouvé pour réaliser une capture d'écran la classe *java.awt.Robot* qui par l'entremise de la fonction *createScreenCapture* offre la possibilité de construire une *BufferedImage* contenant la capture d'écran d'un rectangle donné en paramètre de la fonction. Malheureusement, si *Robot* fonctionnait parfaitement sous Windows, elle posait problème sous Linux. Cela parce qu'elle a besoin des droits pour pouvoir lire dans le buffer Image de la carte graphique, droits que l'on n'a pas toujours sous Linux. Fort heureusement, une autre méthode, spécifique à Java3D, a été préférée. La classe *Raster* positionne un buffer Image dans le monde 3D. On peut également, grâce à la méthode *readRaster* de *GraphicsContext3D,* remplir ce buffer Image (en fait, une *ImageComponent2D*) associé à la classe *Raster* avec les données affichées sur le panel de visualisation 3D. Il suffit après de récupérer l'image contenue dans le buffer Image par la méthode *getImage* de *ImageComponent2D*.

Capture d'Ecran Grande Qualité

Le mécanisme pour réaliser une capture d'écran grande qualité est différent de la méthode normale car on ne peut pas lire les données du buffer Image de l'écran étant donné que la résolution que l'on désire obtenir peut être plus grande que celle de l'écran. Il faut donc créer un nouveau buffer image où sera restituée la capture grande qualité. Java3D permet de rendre une même scène sur plusieurs Canvas3D différents (afin d'avoir une vue stéréoscopique, par exemple). Certains de ces Canvas3D peuvent être construits non pas pour être affichés dans un composant AWT ou Swing mais uniquement comme un buffer Image permettant de récupérer la vue 3D. On dit alors que ces Canvas3D sont en mode Rendu Hors-Ecran (Off-Screen Rendering). La classe *OffScreenCanvas3D* s'occupe de la capture d'écran. Après avoir initialisé sa taille et l'avoir attachée à la *View* de Java3D, sa méthode *retourneImage* demande explicitement à la vue de lui dessiner la scène 3D par la méthode *renderOffScreenBuffer* héritée de *Canvas3D*. On peut ensuite lui demander l'image rendue à l'aide de *getOffScreenBuffer().getImage()*.

5.6. Les Transformations Géométriques

En image de synthèse, afin d'unifier le traitement des transformations géométriques d'une scène, on utilise des **coordonnées homogènes**. Les transformations géométriques sont codées sous la forme d'une matrice 4x4 : translation, rotation, homothétie, changement d'échelle et projection.



En 3D, un point *P(X,Y,Z)* a comme coordonnées homogènes *(X,Y,Z,1)*. Après la multiplication par la matrice de transformation on obtient le résultat *(XP, YP, ZP, H)*. Les coordonnées transformées de *P* sont alors *(XP/H, YP/H, ZP/P)*.

Dans Java 3D, la matrice des transformations géométriques est enregistrée dans un objet *Transform3D*.

TransformGroup est un groupe de transformation spatiale. C'est un objet *Group* enrichi d'un objet *Transform3D* qui va permettre de positionner, orienter ou mettre à l'échelle tous les enfants du groupe.

Comment obtenir la position de la camera en Java3D ?

Il suffit de récupérer le *TransformGroup* de la caméra par un appel aux méthodes *getViewingPlatform().getViewPlatformTransform()* de *VirtualUniverse* et depuis ce *TranformGroup* faire appel aux méthodes *transformGroup.getTransform(transform3D)* pour récupérer le *Transform3D* et *transform3D.get(vector3d)* pour obtenir le vecteur position dans le monde 3D. On peut ensuite convertir cette position selon la calibration du MNT. Cette opération est effectuée dans la méthode *getPosition* de *Navigation*.

Comment obtenir l'orientation de la caméra en Java3D ?

Trouver l'orientation de la caméra est bien plus compliqué que de localiser sa position. Fort heureusement, il a été découvert une méthode sur Internet permettant de la faire, qui nous a bien aidé. La fonction *getRotAngles* se trouvant sur http://jamaica.ee.pitt.edu/Eric/java3d/rotation.htm, retourne à partir d'un *Transform3D* les deux angles d'Euler possibles pour cette orientation. La méthode *getRotation* se sert de cette fonction pour l'affichage de la rotation en sélectionnant le bon angle d'Euler parmi les deux et en transformant en degrés les angles en radians.

Comment se recentrer sur le MNT ?

Cela est en fait très simple grâce à la méthode *lookAt* de *Transform3D*. Cette fonction permet de calculer la transformation nécessaire en prenant comme paramètre l'œil de la caméra et le centre vers lequel il regarde. Il suffit donc de donner comme œil la position de la caméra, et comme centre, le centre du MNT. Ne pas oublier d'inverser le *Transform3D* obtenu comme précisé sur la Javadoc avant de modifier le *TransformGroup* de la caméra.

Comment se remettre à l'horizontale ?

Cette opération a été faite par une méthode "maison". Peut-être en existe-t-il une plus efficace ? Se remettre à l'horizontale consiste en fait à réaliser une rotation en Z afin d'avoir le ciel au-dessus de sa tête (roulis nul) et ensuite à réaliser une rotation en X afin de se retrouver effectivement à l'horizontale (pente nulle). Aussi, il suffit de trouver les angles afin de pouvoir effectuer les rotations. Pour trouver l'angle en Z, on fait effectuer à la caméra un "pas de côté virtuel" afin de connaître le vecteur de déplacement dans le monde 3D. Bref, on multiplie la *Transformation3D* de la caméra par la *Transformation3D* représentant la translation (1,0,0).

On projette ce vecteur par rapport au plan horizontal (ce qui revient, tout simplement, à mettre à 0 la composante en Y du vecteur précédent) et ainsi l'angle que l'on désire trouver est l'angle entre les deux vecteurs que l'on peut obtenir en se servant de la méthode *angle* de *Vector3d*.

On procède de la même manière pour l'angle en X, en calculant l'angle entre le vecteur de déplacement, obtenu en multipliant la *Transformation3D* de la caméra par la *Transformation3D* représentant la translation (0,0,1) avec sa projeté horizontale (là aussi, il suffit de mettre à zéro la composante en Y).

Cette méthode a un petit inconvénient, dû au fait que l'angle calculé par la méthode *angle* de *Vector3d,* utilisant *acos*, retourne par conséquent un angle compris entre 0 et PI. C'est pour cela qu'il peut être nécessaire d'appuyer plusieurs fois sur le bouton de remise à l'horizontale.

Comment faire pour repositionner le repère ?

Avant d'expliquer cela, il faut déjà comprendre que le *TransformGroup* du repère, ainsi que le *TransformGroup* de la croix verte, sont des fils du *TransformGroup* de la caméra. Cela permet qu'à chaque déplacement de la caméra, la croix verte et le repère réalisent le même déplacement que la caméra. Ainsi, pour correctement positionner le repère, il faut juste effectuer les bonnes rotations depuis sa position centrale et à la fin rajouter une translation afin que le repère se retrouve en bas à gauche du panel du visualisateur 3D. Ce que fait la fonction *repositionneRepere* de Navigation.

Pour trouver les bonnes rotations à appliquer, il faut connaître à la fois les vecteurs de révolution dans l'espace et les angles de rotations à effectuer autour des ces vecteurs.

Pour calculer les vecteurs, on multiplie la *Transformation3D* de la caméra par la *Transformation3D* représentant la translation (1,0,0) pour connaître la transformée du vecteur (1,0,0) dans le monde 3D. On effectue la même opération avec le vecteur (0,1,0) puis avec le vecteur (0,0,1). Nous avons ainsi les 3 vecteurs du repère de la caméra dans le monde 3D.

Pour obtenir les angles à appliquer, on utilise tout simplement la méthode *getRotAngles* sans oublier que, comme le repère doit en fait effectuer des rotations opposées à la caméra, on prend les angles opposés à ceux rendus par *getRotAngles*. Possédant vecteurs et angles, on peut calculer la *Transformation3D* à appliquer en créant un objet *AxisAngle4d* pour chaque couple vecteur/angle, puis en utilisant la méthode *set(AxisAngle4d)* de *Transformation3D* pour les transformer en *Transformation3D* et, enfin, en multipliant les 3 *Transformation3D* entre elles dans le bon ordre. La *Transformation3D* résultante est celle que doit avoir le *TransformGroup* du repère.