INSTITUT SUPERIEUR
D'INFORMATIQUE
DE MODELISATION
ET DE LEURS APPLICATIONS

COMPLEXE DES CEZEAUX
BP 125 - 63173 AUBIERE CEDEX

Rapport de projet 2ème année

MODELEUR DE PAYSAGE 3D

Présenté par : DEVIC François et GOUY Nicolas
Responsable ISIMA : Dr DUHAMEL Christophe

2002-2003

Remerciements

Nous tenons à remercier Christophe DUHAMEL pour l'encadrement du projet, le suivi régulier, et la disponibilité dont il a fait preuve.

Nous remercions également David RC Hill pour son aide concernant le simulateur de croissance des forêts du LIMOS.

Merci également à Vincent BARRA et Rémy MALGOUYRES pour leur aide technique pour le développement de la partie infographie du projet.

Enfin, merci à Sébastien DEGUY, pour son aide et sa documentation sur les fractales.

GLOSSAIRE

API : une API est une interface de programmation. Ainsi les API graphiques permettent une programmation plus aisée des fonctions graphiques. De plus, certaines fonctions existent déjà.

Fractale : une fractale est un motif géométrique qui se répète à l'infini. Ainsi, on peut zoomer à l'infini sur ce motif, on retrouvera le motif.

Implémentation : Adaptation et mise en service d'un logiciel donné sur un système informatique donné .

Implémenter : programmer un algorithme dans un langage informatique.

Itérateur : classe, variable « pointeur » ou mécanisme qui permet de repérer et de parcourir tous les éléments d'une collection sans se soucier de la structure sous-jacente des données.

Random : fonction délivrant un nombre aléatoire compris entre la borne inférieure et la borne supérieure passées en paramètres

Rendre : ici afficher

SIGGRAPH : grande exposition américaine sur tout ce qui concerne le graphisme sur ordinateur (http://www.siggraph.org/)

XLib : bibliothèque graphique bas niveau qui permet de créer des environnements graphiques sous UNIX

LISTES DES ABREVIATIONS

API : Application Programming Interface

Cg : C for graphics

DMA :Defense Mapping Agency

DEM : Digital Elevation Model

GLUT : OpenGL Utility Toolkit

LIMOS : Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes

MFC : Microsoft Foundation Class

MUI : Micro User Interface

OpenGL : OpenGraphics Library

USGS : United States Geological Survey

Table des figures

ANNEXES

TABLE DES MATIERES

TABLE DES ANNEXES

Résumé

Le LIMOS travaille sur de nombreuses simulations. L'une d'elles est notamment une simulation de forêts. Les résultats de cette simulation ne sont pour l'instant que des images figées. Pour avoir une vision plus globale des résultats les chercheurs voulaient une application qui leur permettrait de les visualiser en 3 dimensions.

Le but de notre projet était donc de créer cette application. De plus, comme nous commencions la programmation objet au début de cette année, ce projet devait nous confronter à des problèmes liés à la conception objet.

Donc avant de développer cette application, nous avons préalablement réalisé une analyse du projet, ce qui nous a permis d'anticiper certains problèmes.

Pour le développement en lui-même nous avons utilisé le langage C++ qui nous a semblé être le langage le plus approprié pour la réalisation du projet. Pour tout le développement de la partie graphique, nous avons utilisé la bibliothèque GLUT, qui est une sur-couche d'OpenGL. Nous avons programmé aussi bien sous Linux que sous Windows.

Le logiciel, sous sa forme actuelle, permet de lire des fichiers de type .pay, un format que nous avons développé et les fichiers DEM. Il permet aussi d'afficher des forêts. Le déplacement dans le paysage est aisé. De plus, la version Windows ainsi que la version Linux fonctionnent.

Mots-clés : simulation de forêts, conception objet, C++, GLUT

Abstract

The LIMOS laboratory is developing numerous simulations. In particular, one of them is a forest simulation. The results of this simulation are only fixed images for the moment. To have a more complete view of the results, resarchers wanted an application which would enable them to see results in 3 dimensions.

So the main goal of our project was to create this application. Furthemore, as we had started the Object Oriented Programmating at the beginning of this year, this project was an occasion to confront with problems involved in the object design.

Thus, before coding any application, we carried out an analysis of the project which enabled us to anticipate some problems.

For the development itself, we used the C++ language which has seemed us to be the more appropriate language for the implementation of our project. For the whole graphics programming, we used GLUT library which is an OpenGL add-on. We programmed either under Windows or under Linux.

The software package, in its current form, allows to read .pay files (a format that we had developed) and DEM files. It also allows to display forests. Travelling on the landscape is easy. Moreover, the Windows release as well as the Linux release work properly.

keywords : forests simulation, object design, C++, GLUT

INTRODUCTION

Dans le cadre de notre projet de deuxième année d'études à l'ISIMA, nous avons développé une application qui permet de visualiser des forêts se trouvant sur un terrain réel ou imaginaire.

Un simulateur de forêts est utilisé au LIMOS. Le but de notre projet est de pouvoir visualiser les forêts sur des terrains réels. Les informations géographiques ainsi que les forêts utilisés pour ce simulateur sont créés à partir de données réelles. Ces données sont recueillies sur le terrain et ensuite mémorisées dans la machine.

Le simulateur du LIMOS propose déjà une sortie graphique, mais elle présente l'inconvénient d'être statique. Les chercheurs du LIMOS voulaient une représentation dans laquelle il était possible de se déplacer mais aussi pouvoir rajouter des forêts, des terrains.

De nombreux modeleurs de terrains existent sur Internet mais aucun ne permet de visualiser un terrain à partir de coordonnées précises. De plus, il est impossible de rajouter des forêts. Nous devions donc tout recréer pour obtenir le résultat escompté.

Il nous fallait aussi développer une application qui soit facile à utiliser et la plus portable possible. En effet, une contrainte qui nous était imposée était de pouvoir utiliser cette application sur n'importe quelle machine. Ainsi, notre application, une fois recompilée devait pouvoir fonctionner aussi bien sous Linux que sous Windows.

Après un bref descriptif des besoins et de l'environnement de l'application, nous allons montrer comment nous sommes passés des coordonnées recueillies à l'affichage du paysage correspond. Nous verrons ensuite comment nous avons créé les arbres. Puis nous décrirons les différentes méthodes utilisées pour l'affichage du terrain et des forêts. Nous terminerons par une discussion sur l'outil final et ses extensions possibles.

Partie 1
Etude

A) CONTEXTE DE L'APPLICATION

Cette étude devait aboutir sur un logiciel permettant de représenter la simulation de croissance des forêts. Plusieurs solutions étaient envisagées : nous pouvions développer des modules complémentaires de logiciels existants, mais aucune ne convenait, il nous a donc fallu développer un produit de bout en bout.

Le logiciel devait donc fournir les fonctionnalités suivantes :

- récupérer des données provenant de diverses applications et dont le format pouvait varier (la gestion de différents standards était donc nécessaire)
- exploiter ces données en les retraitant grâce à différents algorithmes
- présenter ces informations sous un aspect compréhensible, la vision en 3D permettant le meilleur rendu possible
- permettre une interaction avec l'utilisateur
- s'assurer que l'application serait facilement exploitable par la suite, le projet ne permettant pas un développement complet autour des fonctionnalités attendues

B) GENERATEUR DE PAYSAGE

Un des premiers points à aborder était de pouvoir générer un terrain aléatoirement c'est-à-dire sans disposer de support réel. Nous avons utilisé des algorithmes issus de l'étude des fractales.

Nous allons étudier différentes méthodes.

I. La création du maillage

I. 1. Etude de l'algorithme du « MidPoint Displacement »

Pour s'imaginer la génération de paysages à l'aide de cette méthode, on va tout d'abord l'expliquer en 2 dimensions.

I. 1. 1. La méthode du MidPoint Displacement (2D)

Le MidPoint Displacement est en fait une série d'itérations au cours desquelles un segment AB est découpé en 2 au point I. On affecte à I un déplacement vertical comprise entre –deltaH et +deltaH. Le deltaH initial est tout d'abord initialisé à la moitié du segment AB (AB/2).

A chaque nouvelle itération, on va multiplier deltaH par 2^{-r} avec r la constante de rugosité. Cela permet de contrôler la planarité du terrain obtenu.

r>1: terrain sans trop de relief (aplati)

r<1: terrain rugueux (accidenté)

r=1: valeur magique, deltaH est divisé par 2, tout comme la longueur du segment.

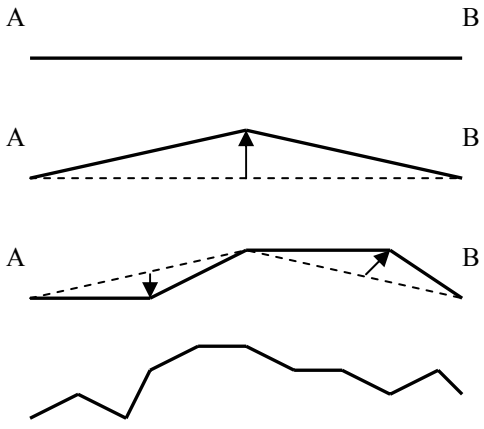


Figure 1 : application de l'algorithme à un segment

Ici, on a utilisé la valeur r=1. Elle semble donner des courbes naturelles, c'est-à-dire ni trop plate ni trop accidentée.

I. 1. 2. La méthode du Diamond-Square (3D)

Au lieu de partir avec un segment, on va utiliser un carré.

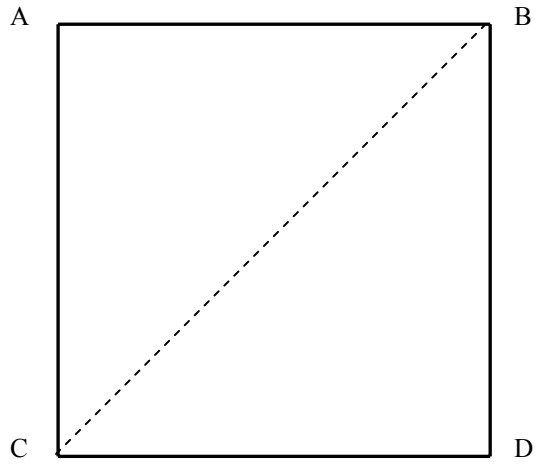


Figure 2 : algorithme appliqué à un carré : première étape

La première étape est celle du Diamant, durant laquelle on calcule la hauteur du centre de ce quadrilatère (noté E sur la figure ci-dessous).

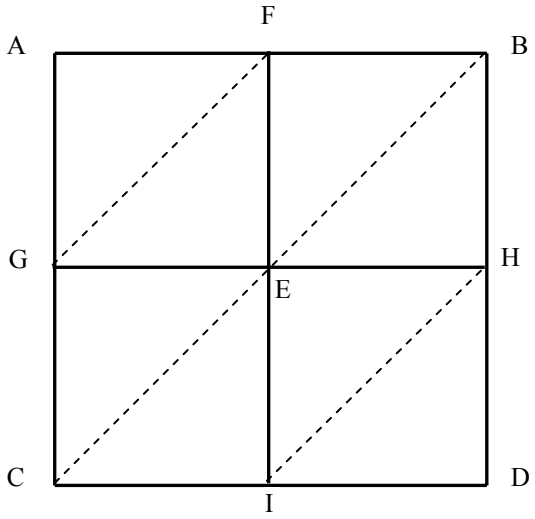


Figure 3 : algorithme appliqué à un carré : seconde étape

La hauteur du point E peut être obtenue en utilisant la fonction ci-dessous :

E = (A+B+C+D)/4 + Random(-deltaH, +deltaH)

On calcule donc la hauteur de E en faisant la moyenne des 4 sommets autour formant le rectangle. On ajoute ensuite une valeur aléatoire comprise entre –deltaH et +deltaH qui sera multipliée à chaque fois que l'on descendra dans les itérations.

La prochaine étape est appelée l'étape du Carré (Square). Il s'agit de calculer l'altitude des milieux des segments entourant le point E en faisant la moyenne des rectangles autour d'eux et en ajoutant encore une valeur aléatoire.

Par exemple, pour calculer F, on fait :

F = (A+B+E+E)/4 + Random(-deltaH, +deltaH)

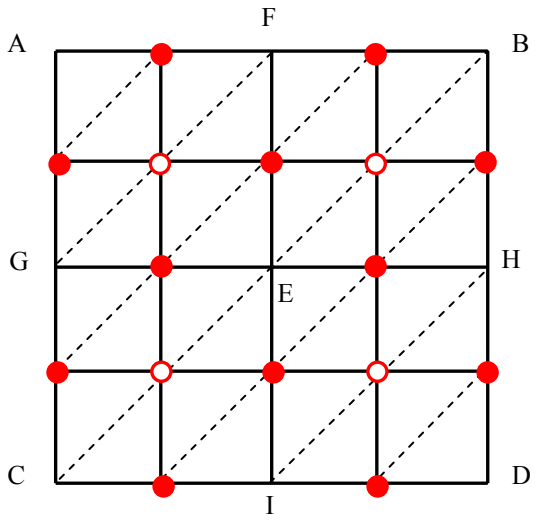


Figure 4 : algorithme appliqué à un carré : après plusieurs itérations

Les points blancs sont calculés dans l'étape Diamant de la prochaine itération et les rouges sont calculés dans l'étape Carré. Cet algorithme est répété jusqu'à ce qu'un niveau de détail suffisant soit atteint.

I. 1. 3. Limites de cette méthode

Nous avons étudié cette méthode et l'avons implémenté. Ce fut la première étape du projet, car c'est la méthode de base utilisée dans énormément de projets de génération de paysages. Cependant, nous y avons trouvé quelques limites.

En ce qui concerne le rendu graphique, on s'aperçoit que l'on voit des régularités dans la génération aléatoire, ceci étant dû au maillage trop régulier et carré.

Le maillage est forcément carré et ne s'applique pas à des listes de points pris au hasard sur une carte. Le maillage doit être carré et sa dimension doit être une dimension de taille 2^n. On ne peut intégrer facilement des points ad hoc que s'ils ont des coordonnées en puissance de 2.

L'interaction avec des données géographiques non uniformes est trop compliquée.

Nous avons donc gardé le même principe, mais utilisé une méthode plus souple afin de remédier à ce problème.

I. 2. Adaptation vers un algorithme plus souple

Une des solutions appropriées était d'utiliser des triangles.

Au lieu de déterminer le centre des carrés, on va prendre un point caractéristique d'un triangle se trouvant à l'intérieur et on découpe ce triangle en 3.

Nous avons choisi de prendre le barycentre qui est l'intersection des différentes médianes d'un triangle.

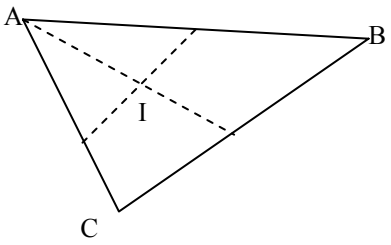


Figure 5 : calcul du barycentre

Ici, I est le barycentre. On crée 3 triangles à partir de cette figure (5) et on calcule l'altitude de I.

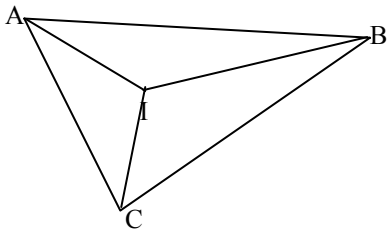


Figure 6 : la création de 3 nouveaux triangles

Le problème de l'altitude : pour l'algorithme du « MidPoint displacement », on utilisait un deltaH, qui était diminué à chaque itération. Cela convenait parfaitement car le maillage était régulier, donc la taille des carrés était la même pour une itération donnée. Ici, le problème est double : on veut faire varier l'altitude de I suivant la forme de ABC et également suivant sa taille (ou plutôt son aire).

Notre solution consiste à calculer la moyenne des distances de chaque sommet au côté opposé. Elle est en fait proportionnelle à l'aire et nous donne une bonne idée de la marge que l'on peut avoir pour l'altitude de I.

On a donc :

MoyenneDistance = DistanceSommetCote(A,BC) + DistanceSommetCote(B,AC) + DistanceSommetCote(C,AB)

I = (A+B+C)/3 + Rugosite*Random(MoyenneDistance,-MoyenneDistance)

On introduit également un coefficient de rugosité (le r vu plus haut).

De plus, on ajoute une partie aléatoire sur les x et y pour avoir moins uniforme.

Nous sommes donc capables de générer un terrain aléatoirement et la création de reliefs est assez aisée en « forçant » certains pics ou en jouant sur des paramètres. Maintenant se pose un problème plus compliqué : créer des arbres assez réalistes sans avoir de données réelles sur lesquelles s'appuyer.

II. Génération d'arbres

II. 1. Le L-System

Nous avons généré des arbres grâce au L-System (L pour Lindenmayer son concepteur) qui est une technique très utilisée pour générer des plantes, des arbres et toutes les structures végétales.

Cette technique est très simple à comprendre et repose sur le fait que les plantes ont la propriété d'auto-similarité. C'est à dire qu'une petite partie d'un arbre ressemble souvent à la structure globale de l'arbre.

En effet, un des aspects du développement des plantes est qu'un petit segment est transformé en d'autres segments plus complexes, comme illustré ci-dessous.

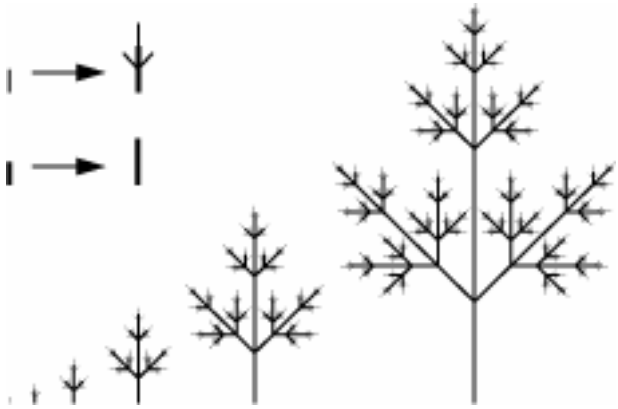


Figure 7 : l'aspect fractal d'un arbre

L'idée du L-System est donc d'utiliser :

- un axiome de départ
- des règles de production
- un nombre d'itérations

En effet, à partir de ces trois éléments on peut générer une structure complexe.
Cette technique rentre dans le domaine des algorithmes fractals.

Pour expliquer la technique, imaginons une tortue qui peut se déplacer librement dans l'espace. Cette tortue (allusion au LOGO, logiciel de dessin simple) peut lire des instructions et se déplacer en conséquence.

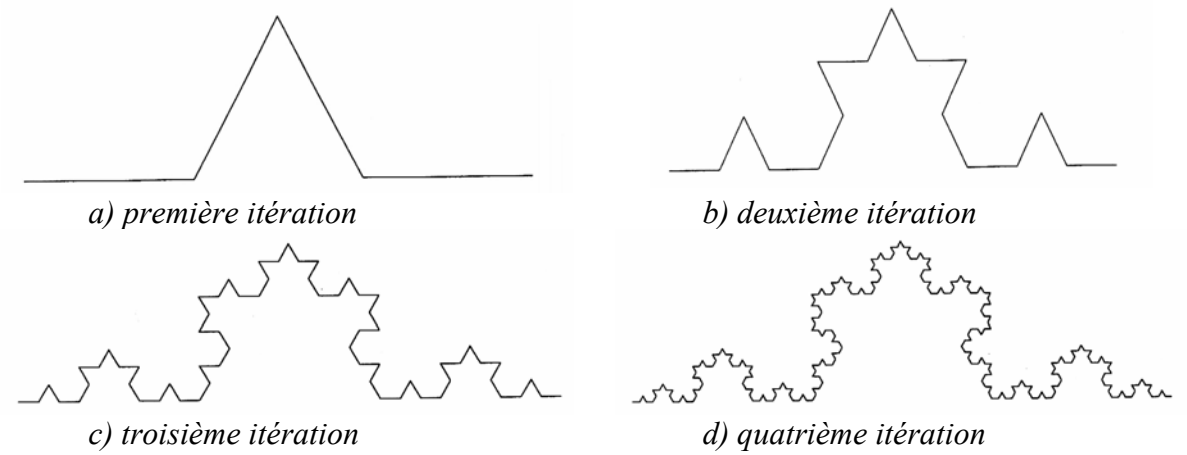
Prenons des instructions simples :

F : dessine une ligne en avançant
+ : tourne à droite de 30°
- : tourne à gauche de 30 °

Maintenant prenons un axiome "F" et la règle de grammaire suivante (règle de productions qui, à partir d'une chaîne, donne une autre chaîne) :

F → F-F++F-F

Itération 1 : (on remplace F par la partie de droite)
On obtient F-F++F-F
Itération 2 : (on remplace chaque F par la partie de droite)
On obtient F-F++F-F - F-F++F-F ++ F-F++F-F - F-F++F-F



a) première itération *b) deuxième itération*

c) troisième itération *d) quatrième itération*

Figure 8 : les 4 premières itérations pour créer le flocon de Von Koch

Ainsi, on peut obtenir la structure d'arbre suivante (capture d'écran du projet) :

Figure 9 : arbre obtenu grâce à l'algorithme du L-System

II. 2. L'implémentation avec une tortue

Cependant, différentes adaptations ont dû être nécessaires pour une tortue en 3D.
Une tortue est caractérisée par :

- une position
- un vecteur direction
- une longueur de déplacement

Nous avons donc fourni un vocabulaire complet permettant à la tortue de se déplacer dans l'espace grâce à des matrices de rotation 3D que l'on applique au vecteur.

II. 3. Les forêts

Un autre problème se posait : comment gérer des forêts ? Si chaque arbre était généré aléatoirement par un L-System, cela prendrait trop de temps de calcul. Il fallait trouver une solution optimisée car le projet n'avait pas pour but de rendre réaliste les arbres mais de montrer leur expansion géographique (et non leur croissance).

Ainsi, pour chaque forêt, on génère plusieurs patrons d'arbre (mais pas autant que d'arbres) et on a un fichier de positions qui nous permet de savoir où sont les arbres.

On parcourt toutes les positions pour placer un patron d'arbre choisi aléatoirement (que l'on modifie par homothétie, suivant sa hauteur, et rotation pour ne pas avoir une forêt trop uniforme).

Le L-System permet donc de générer plusieurs patrons d'arbres, ceci en partant de la même grammaire, en introduisant des coefficients aléatoires dans les règles de production.

Remarque concernant les positions : on utilise des positions (x, y) qui ne contiennent pas l'altitude et on effectue une interpolation sur un triangle qui va contenir le point. On fait une projection verticale sur ce triangle afin de trouver l'altitude z. Cette manière nous permet de déterminer de manière exacte le point de contact de l'arbre avec le sol.

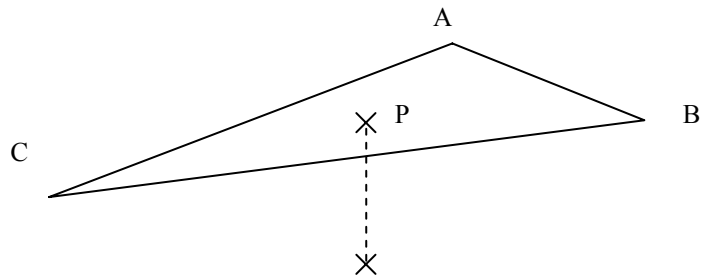


Figure 10 : projection de la racine d'un arbre sur un triangle

Explication du calcul de la projection :
Les vecteurs CA, CP et CB doivent être colinéaires. On calcule donc le déterminant de la matrice formée par ces 3 vecteurs et on le force égal à 0. La seule inconnue est Zp car les termes Xp et Yp sont les mêmes que le point à projeter. Ceci nous permet d'obtenir Zp.

Nous avons donc traité tous les problèmes de génération automatique de données (le terrain ainsi que les arbres). Cependant notre logiciel devait fournir une autre fonctionnalité importante : à partir d'une liste de points fournie par l'utilisateur, on devait créer un maillage, c'est à dire tous les triangles associés. Si l'on considère un ensemble de points, il est facile de créer une figure avec seulement des triangles. Mais ce problème devient compliqué lorsque

l'on doit générere plusieurs milliers de points. Nous avons donc étudié des algorithmes de triangulation.

III. La triangulation et les propriétés de Delaunay

III. 1. Le respect des propriétés de Delaunay

Que ce soit pour la triangulation ou lors d'itérations fractales, nous devons respecter les propriétés de Delaunay.

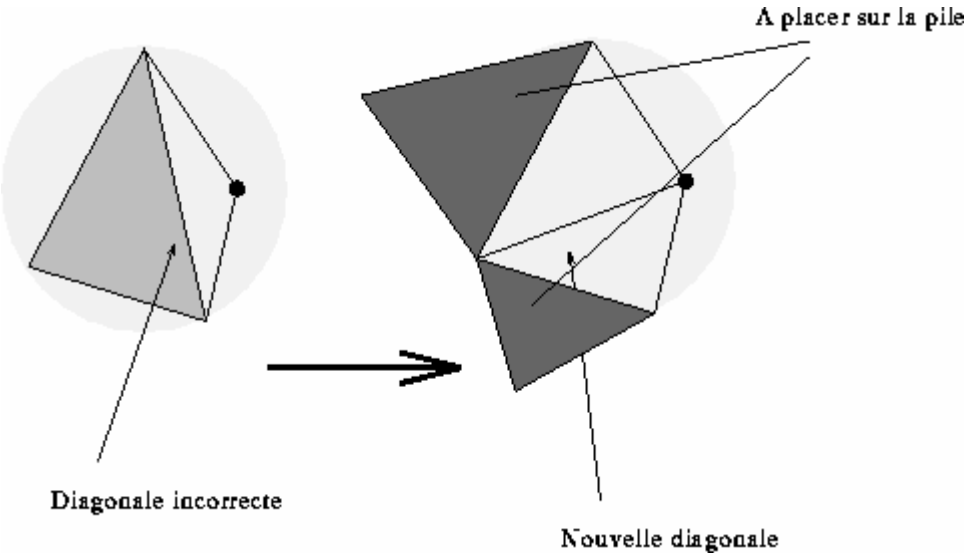


Figure 11 : illustration de la propriété de Delaunay pour éviter les « triangles plats »

Ces propriétés ont pour but de supprimer les triangles aplatis. En effet, les algorithmes ont tendance à générer des triangles trop plats si on ne les contrôle pas. Ceci nuit énormément au rendu fait par la suite ainsi qu'à l'efficacité de la triangulation.

Le cercle dessiné est le cercle circonscrit au triangle. On dit que la diagonale est incorrecte lorsqu'un point est à l'intérieur d'un triangle appartenant également au cercle circonscrit d'un triangle voisin. On transforme ces 2 triangles en 2 autres en utilisant l'autre diagonale du quadrilatère. Mais il faut continuer à tester sur les triangles adjacents, car cette propriété peut ne pas être respectée.

III. 2. La triangulation

Il existe de nombreux algorithmes permettant de trianguler un nuage de points.

La triangulation de Delaunay est de loin la plus populaire des méthodes de triangulation 2D et 3D.

Soit P un ensemble de points P_1, P_2,..., P_N du plan.

Nous allons étudier l'algorithme de Watson.

Algorithme de Watson [BARRA2000]

Il existe un grand nombre d'algorithmes pour construire une telle triangulation. La procédure proposée est principalement due à Watson et permet de réaliser cette triangulation avec une certaine efficacité. Cette méthode se résume comme suit. Tout d'abord, on crée trois nouveaux sommets formant un « super » triangle englobant tous les points à trianguler. La triangulation initiale est ce triangle. La procédure consiste ensuite à ajouter successivement tous les sommets à trianguler. Pour chaque sommet $x_{i,}$ on effectue les opérations suivantes :

- on recherche un triangle existant dans lequel x_i est inclus et on forme trois nouveaux triangles en reliant notre sommet aux trois côtés du triangle existant qui sera détruit. Le gain net en triangles est donc de deux unités.
- tous les triangles adjacents au triangle détruit sont placés dans une pile du type *Premier Entré Premier Sorti (PEPS)*. Pour chaque triangle t_j de la pile :
- on vérifie si x_i appartient au cercle circonscrit de t_j.
- si c'est le cas, cela signifie que t_j et l'un des nouveaux triangles forment un quadrilatère avec une diagonale placée dans le mauvais sens. Cette diagonale doit être remplacée par l'autre diagonale pour que la triangulation conserve la propriété de Delaunay. Il s'agit donc de remplacer les deux triangles existants par deux nouveaux triangles par une technique d'échange. Lorsque l'échange est effectué, il faut ajouter à la pile les triangles qui sont adjacents aux deux nouveaux triangles et qui n'ont pas x_i comme sommet.
- si ce n'est pas le cas, il n'y a rien à faire.

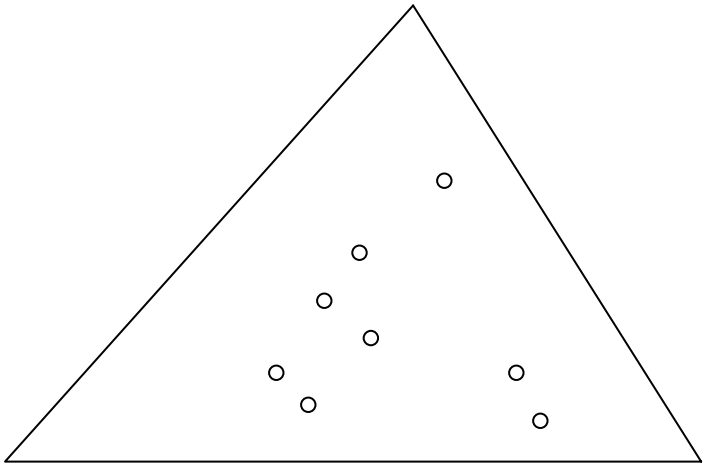


Figure 12 : algorithme de Watson : création du grand triangle englobant les points

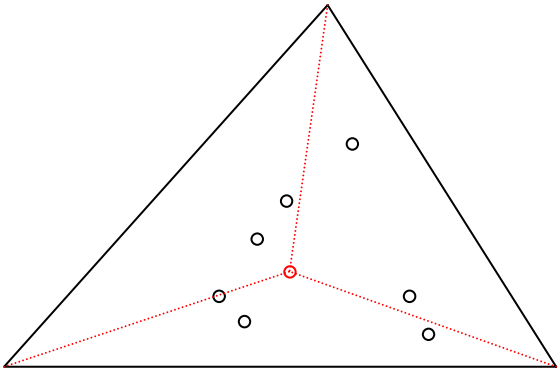


Figure 13 : première étape : on choisit un point et on construit 3 triangles

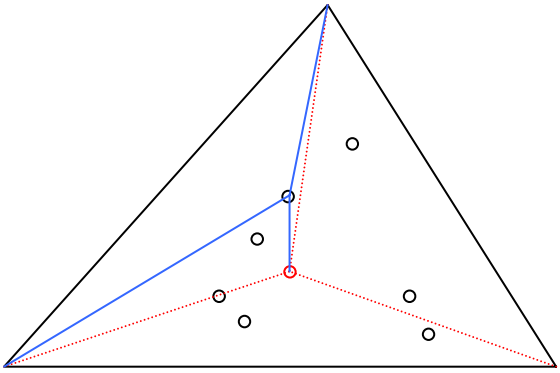


Figure 14 : sélection d'un nouveau point et création des 3 nouveaux triangles.

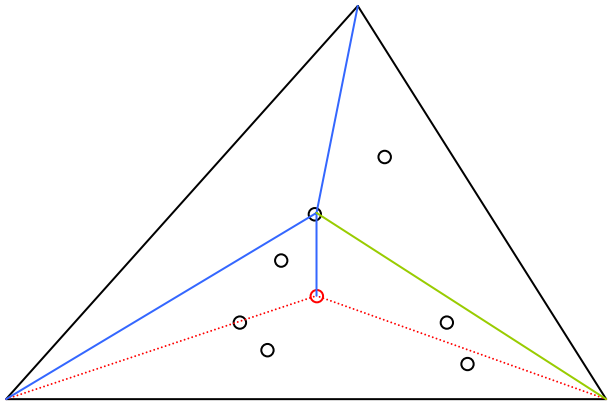


Figure 15 : on regarde les triangles adjacents pour savoir si la propriété de Delaunay est respectée. Elle ne l'est pas donc on redécoupe.

Le nombre final de triangles comprenant ceux contenant les sommets du « super » triangle est *2n + 1*. La triangulation finale est obtenue en détruisant tous les triangles qui contiennent un ou plusieurs sommets du « super » triangle.

On a ainsi généré les triangles à partir de la liste de points, ainsi que les adjacents qui sont renseignés en même temps que l'algorithme de triangulation.

III. 3. Problème de précision

Un problème est survenu lorsque nous avons fait des tests plus poussés. Lorsque le nombre de triangles devient important et que les points sont espacés, un problème lié à la précision des flottants dans la machine se pose.

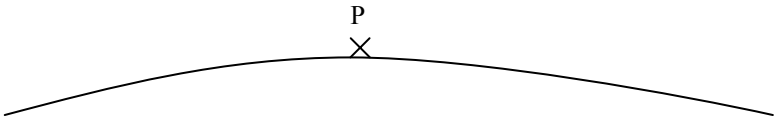


Figure 16 : illustration du problème de précision lorsqu'un point est trop près du cercle

Lorsque le cercle circonscrit est très grand (triangle aplati), on voit que la perte de précision peut engendrer une confusion, indiquant qu'un point est dans le cercle, alors qu'il n'y est pas. Ceci peut poser problème dans le cas suivant :

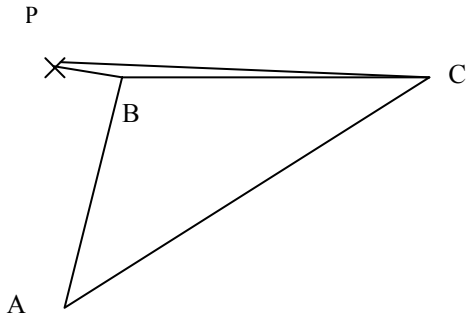


Figure 17 : illustration du problème dans le respect de Delaunay

Cette erreur nous amène à conclure que P est sur le cercle, on peut donc se retrouver avec des triangles qui ne le sont plus.

Nous avons remédié à ce problème en vérifiant que le segment PA coupait bien le segment BC avant d'appliquer le découpage de Delaunay.

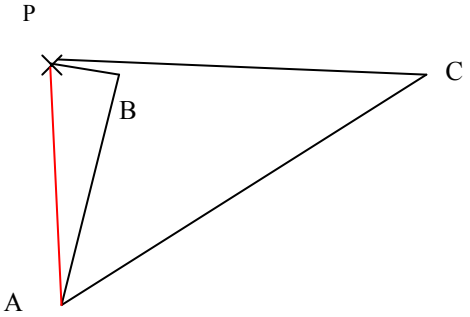


Figure 18 : voilà ce qu'on obtient si on ne tient pas compte de ce problème de précision

Concernant les problèmes pour la génération de paysages, il ne nous restait plus qu'à étudier les manipulations de points à partir de fichiers. Une fois ces données extraites, nous

pouvions appliquer les principes vus précédemment : faire une triangulation, des itérations fractales, des générations aléatoires et mettre en place des forêts sur les terrains.

IV. L'interaction avec des données géographiques

IV. 1. Le standard DEM

Un fichier DEM est en fait un simple tableau à 2 dimensions de niveaux d'élévations des positions d'un terrain. Il s'agit de mesurer des points régulièrement espacés sur un terrain, puis de leur associer une altitude.

Les fichiers DEM permettent à l'heure actuelle la description des Etats-Unis, de l'Alaska ainsi que d'Hawaï avec une précision de l'ordre du mètre.

Les modèles de base dont ils sont issus sont produits par ou pour la DMA qui est le département cartographie pour la défense américaine. Cependant la distribution est effectuée par la USGS qui produit le standard DEM. Les spécifications sont très complètes et formalisées. Le reformatage effectué par le distributeur n'affecte en rien les informations principales (en l'occurrence celles qui nous intéressaient pour le projet : des mesures d'altitudes).

Les données sont structurées sous la forme d'une liste de profils du terrain, comme le résume la figure ci-dessous. On voit ici un des deux formats de DEM qui sont traités par notre logiciel.

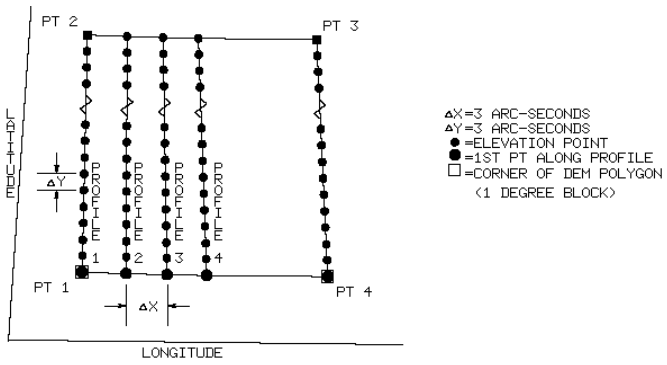


Figure 19 : illustration du format DEM

IV. 2. Traitement des fichiers

Nous avons utilisé un patron de conception (voir les concepts objets développés en section V.2) spécifique pour le traitement des fichiers générant un paysage.

Ce patron de conception a été conçu de manière à pouvoir « greffer » un nouveau standard très facilement. Nous avons développé une bibliothèque pour le format DEM ainsi que notre format PAY qui est tout simplement une liste de points avec leurs coordonnées dans l'espace (format utilisé pour l'interaction avec le simulateur).

Cette méthode de conception présente de nombreux avantages :

- une maintenance plus facile : le standard DEM n'est pas fixe dans le temps et dépend du distributeur. L'adaptabilité est donc accrue ;

- une extensibilité très grande car on peut définir un nouveau standard d'une manière transparente pour le reste du projet.

On peut noter que ce traitement générique des différents fichiers a été appliqué pour les standards de positionnement des forêts et de manipulation de différents formats de fichiers pour les textures.

Nous avons donc vu différents traitements. Certains étant complexes et permettant différentes implémentations, nous avons étudié les méthodologies objets afin de satisfaire aux spécifications concernant la réutilisabilité et l'adaptabilité du logiciel.

V. Les concepts objets

V. 1. L'encapsulation des objets

Ce projet réalisé en C++ devait respecter les concepts objets dont l'encapsulation (le fait de protéger les attributs de chaque instance. Les attributs ne devaient pas être accessibles de l'extérieur. De plus, ceci renforce la séparation entre l'interface d'un objet et son implémentation.

En effet, dans la conception classique, on travaille sur des copies d'objets. Ensuite on manipule ces copies ainsi on a une forte encapsulation en ne travaillant pas sur les originaux. Les quantités de mémoire utilisée deviennent rapidement très importantes (plusieurs centaines de Méga-octets) et des recopies fréquentes auraient vite saturé la mémoire vive de l'ordinateur.

Deux concepts ont donc été mis en place :

- les triangles et points sont stockés dans le maillage et ne sont visibles de l'extérieur que par des identifiants en ce qui concerne les créations et modifications ;
- les triangles et points sont parcourus et repérés par des itérateurs ne permettant pas de modification lorsqu'on les consulte.

Tout cela permet d'avoir des composants logiciels facilement manipulables. En effet, on peut très facilement changer l'environnement pour le rendu (OpenGL), cela n'affectera aucune partie du composant de génération de paysage.

Les structures internes du composant de génération de paysage peuvent être modifiées, cela ne demandera pas de modifications de la partie rendu car on utilise uniquement des itérateurs.

Cette conception nous a permis également de mieux nous répartir le travail et de développer d'une façon indépendante (les modifications internes de l'un ne touchant pas aux développements de l'autre).

V. 2. Les patrons de conception utilisés

V. 2. 1. La création des forêts

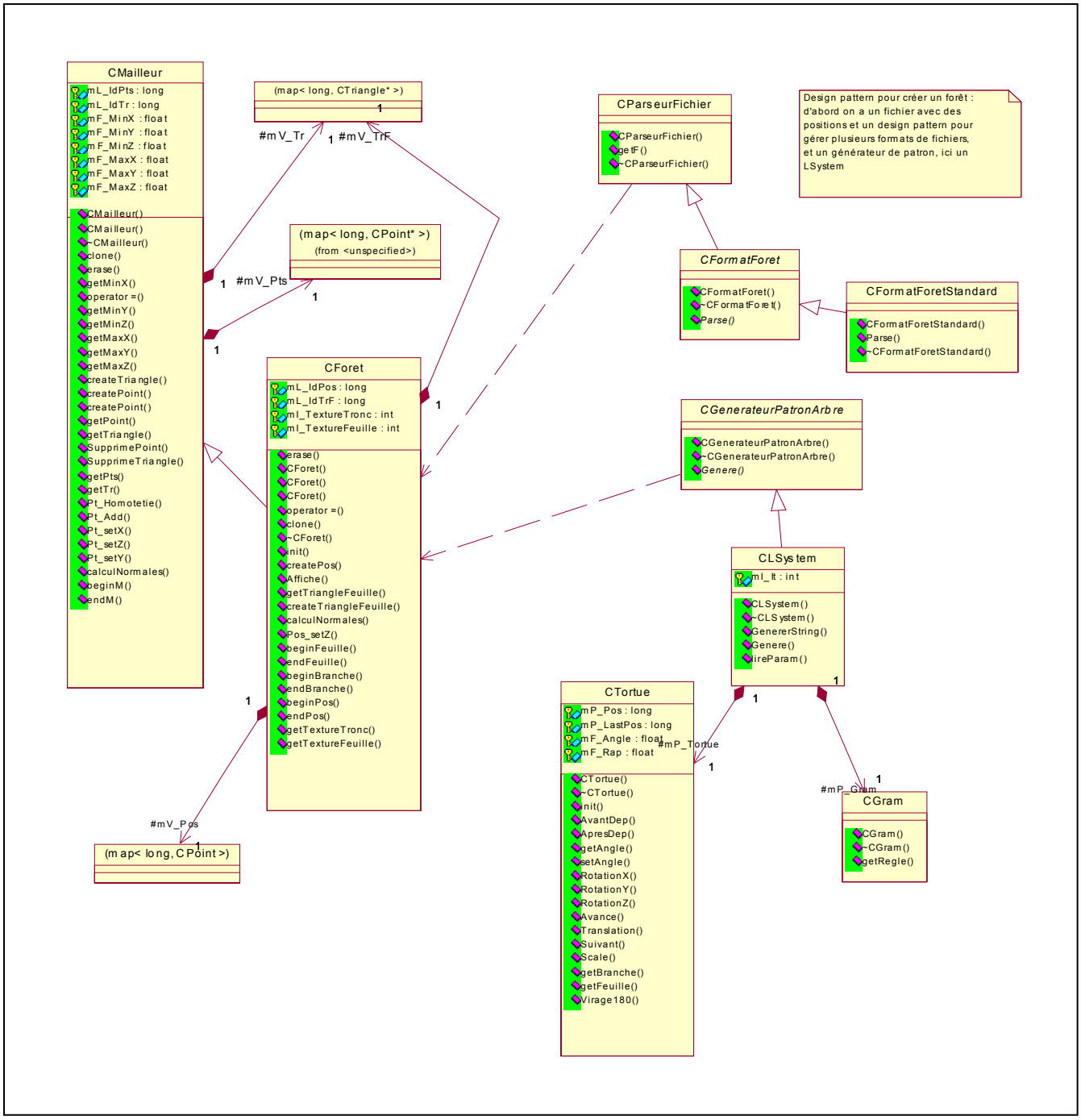


Figure 20 : patron de conception pour la gestion des forêts

V. 2. 2. Delaunay : triangulation et algorithme fractal

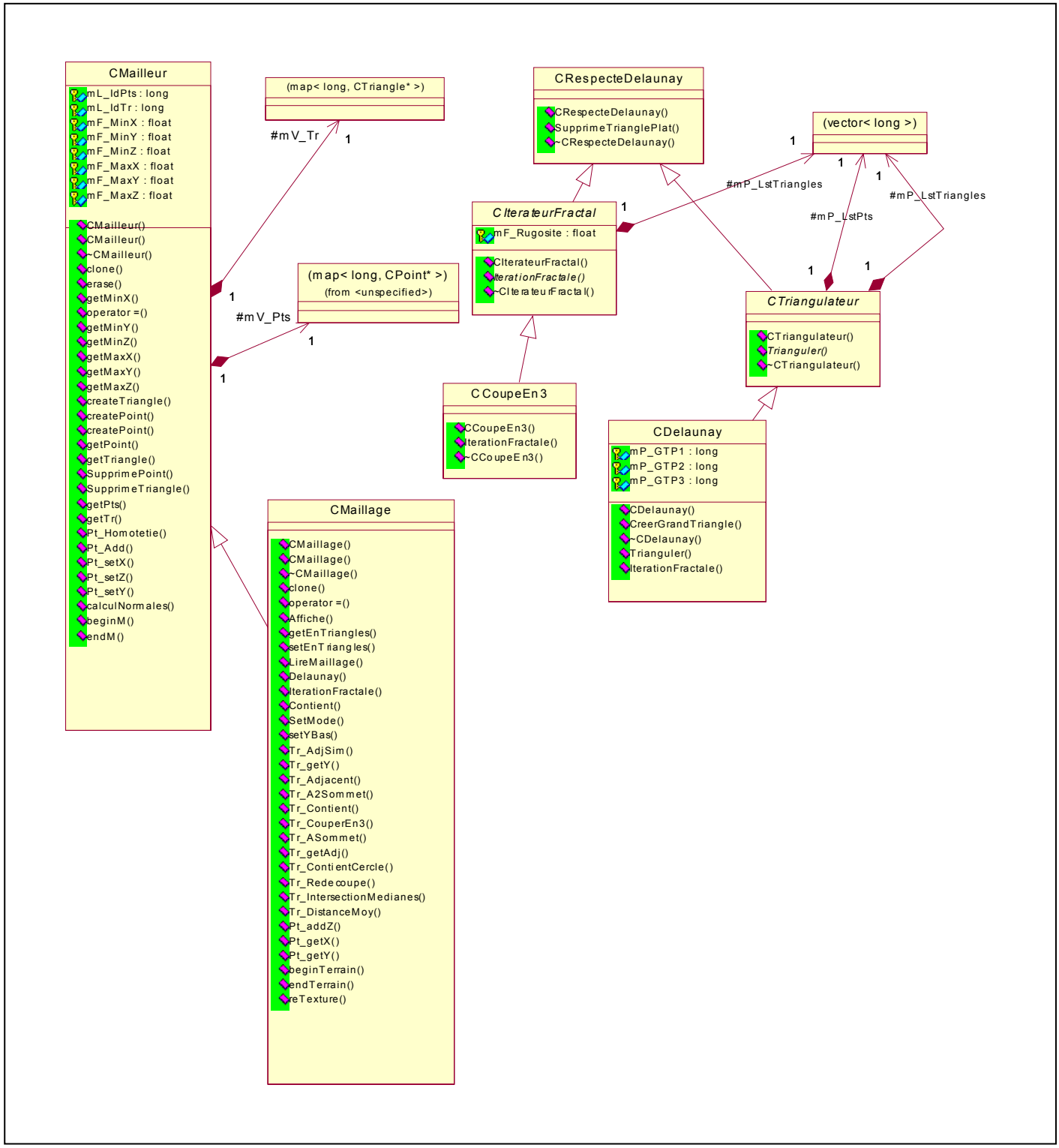


Figure 21 : patron de conception pour les propriétés de Delaunay et les itérations fractales

V. 2. 3. L'extraction des maillages dans les fichiers

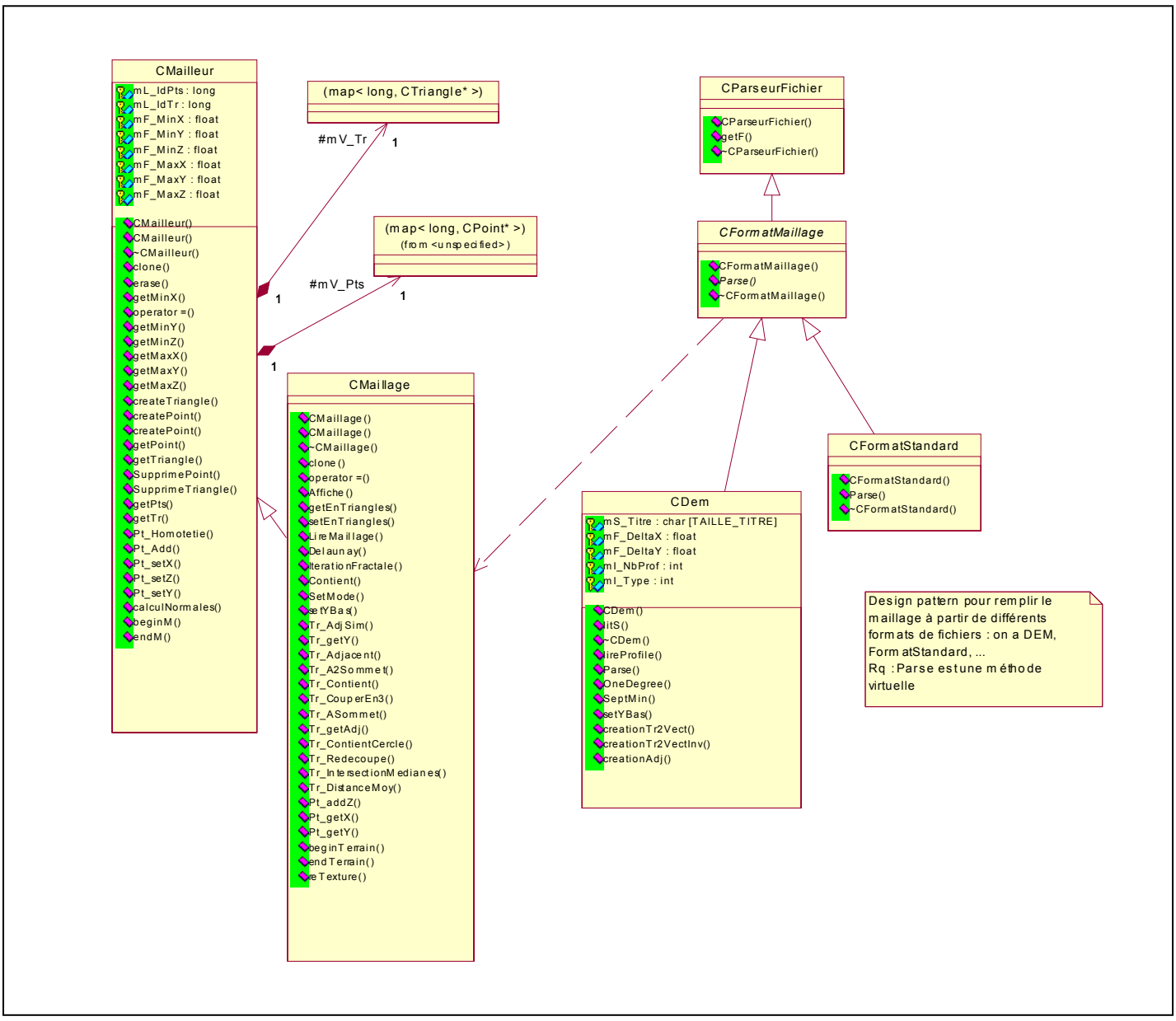


Figure 22 : patron de conception pour la création à partir des fichiers, on peut ajouter autant de standards fichiers que l'on désire

C) L’AFFICHAGE

Après avoir vu comment créer le paysage avec des triangles, nous allons voir comment les afficher. Pour voir le rendu du paysage, il est plus facile de voir des triangles que leurs coordonnées…

Nous allons donc voir dans cette partie pourquoi nous avons choisi OpenGL et plus particulièrement la GLUT. Ensuite nous allons parler de l'interface graphique puis enfin nous allons présenter les différentes techniques employées pour l'affichage.

I. Pourquoi la GLUT ?

Pour afficher le paysage, nous avions pour impératif : celui que le programme puisse s'exécuter sur plusieurs plates-formes différentes. En effet, dès le départ nous avions choisi de programmer un outil de visualisation qui puisse fonctionner aussi bien sur Windows que sur un environnement Unix. Nous allons donc faire une comparaison de toutes les API présentes sur le marché puis nous montrerons les différents problèmes rencontrés.

I. 1. Les différentes API

Actuellement, il y a 4 API principales sur le marché.

API				
Développeurs	OpenGL ARB (Architectural Review Board)	Microsoft	NVidia	Sun
Date et version de la dernière version	1.4 (juillet 2002) et GLUT 3.7 (mars 2000)	DirectX 9 (décembre 2002)	1.0 (décembre 2002)	Java 3D 1.3 (juin 2002)
Compatibilité	Tous les systèmes d'exploitation	Windows	Tous les systèmes d'exploitation	Tous les systèmes d'exploitation
Forme de l'API	bibliothèque	bibliothèque	langage	langage
Langage utilisé	C/C++ - Delphi	C/C++	Cg	Java – Java3D
Compilateur	gcc, g++, Borland C++, VisualC++,...	**Visual C++**[1], Borland C++	Cg compiler	langage interprété

Tableau 1 : comparaison entre les différentes API

Nous voyons donc dans le tableau 1 que seul DirectX est une API propriétaire. En effet, elle ne peut être utilisée que sous Windows. Nous avions donc le choix entre les 3 restantes. Le Cg est un nouveau langage basé sur le C. Il est dédié à la création d'effets spéciaux en 3D (surtout dans les jeux vidéo ou dans la synthèse d'images). De plus, le Cg repose sur l'OpenGL et sur DirectX. La première version définitive est sortie en décembre 2002. Nous n'avons donc pas retenu ce langage pour développer la partie *rendu* de ce projet.

[1] Microsoft précise que l'on peut utiliser DirectX avec tous les compilateurs disponibles sous Windows mais dans la pratique, il n'y a que sous Visual C++ que l'on peut compiler facilement des programmes DirectX.

Le Java 3D est une extension du langage Java. C'est donc un langage interprété. Malgré l'unanimité de la presse au SIGGRAPH 2002 à affirmer que c'est un langage rapide[2], nous avons préféré utiliser l'OpenGL. De plus, notre visualisateur demandait de nombreux calculs. Le langage C++ était alors d'autant plus approprié.

OpenGL est une bibliothèque très répandue dans le milieu informatique. C'est une bibliothèque très facile d'accès. Grâce à l'utilisation de la surcouche GLUT, la création de fenêtres pour l'application est très aisée et demeure portable sur tous les systèmes d'exploitation. Enfin, OpenGL peut s'interfacer avec le langage C.

I. 2. Problèmes rencontrés

OpenGL est une bibliothèque dont la première version date de 1992. Cette première version avait été entièrement développée en langage C. Nous avions pris comme option de développer notre application dans le langage C++. Pour intégrer l'OpenGL dans notre application, nous avons connu quelques problèmes. Les problèmes n'étaient pas dus à l'OpenGL mais plutôt aux surcouches GLUT utilisées pour la création des fenêtres (et MUI pour la création de l'interface graphique). Pour fonctionner, ces 2 surcouches utilisent des fonctions « call-back » ou fonctions « rappelées » en français. Par exemple, pour la GLUT, qui est un gestionnaire de fenêtres, il faut enregistrer une fonction qui va gérer le redimensionnement des fenêtres, une autre pour le rafraîchissement des fenêtres et une autre pour la gestion de la souris. Ces fonctions « rappelées » sont des fonctions enregistrées. Ces fonctions étaient des pointeurs vers des fonctions écrites en C. Nous avons d'abord essayé d'enregistrer des méthodes de la classe contenant les fonctions OpenGL que nous avions créées. Mais le compilateur refusait de fonctionner. En fait, nous avons dû utiliser le concept de fonctions amies du C++. Nous nous sommes servis de ce concept pour la gestion de la fenêtre de rendu ainsi que pour la fenêtre de l'interface graphique.

II. L'interface graphique

Pour que l'utilisateur ne soit pas perdu lors de l'utilisation de notre application, nous avons ajouté une interface graphique. Les premières versions de notre application ne possédaient que des raccourcis clavier. Il est plus agréable pour une tierce de travailler avec la souris qu'avec le clavier.

Voici donc notre interface graphique :

[2] http://www.sun.com/smi/Press/sunflash/2002-07/sunflash.20020722.3.html

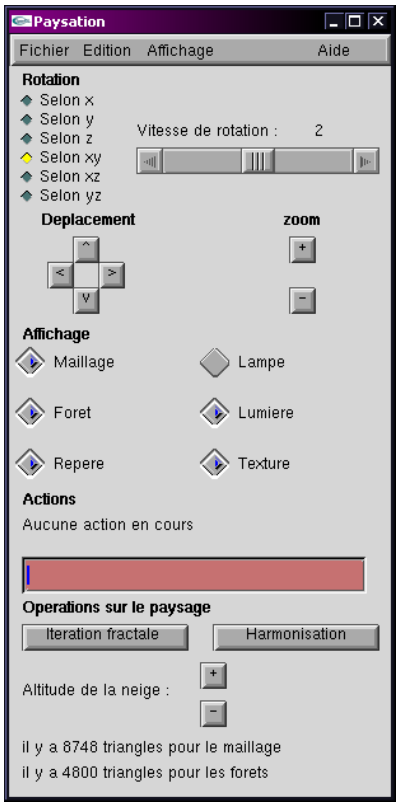


Figure 23 : l'interface graphique

De nombreuses options sont disponibles dans cette interface. On peut ainsi changer la sensibilité de la souris, enlever des éléments du paysage, etc.

Cette interface graphique a été créée grâce à MUI . Toute l'interface a été programmée « à la main ». Il n'existe pas encore d'environnements de développement tel que Visual C++ dans le cadre des MFC ou Borland C++ Builder pour créer cette interface rapidement. L'inconvénient est que si on veut changer un contrôle de place, il faut tout recoder. En contrepartie l'avantage est de pouvoir tout gérer de bout en bout.

Les différentes options sont définies dans l'annexe B.

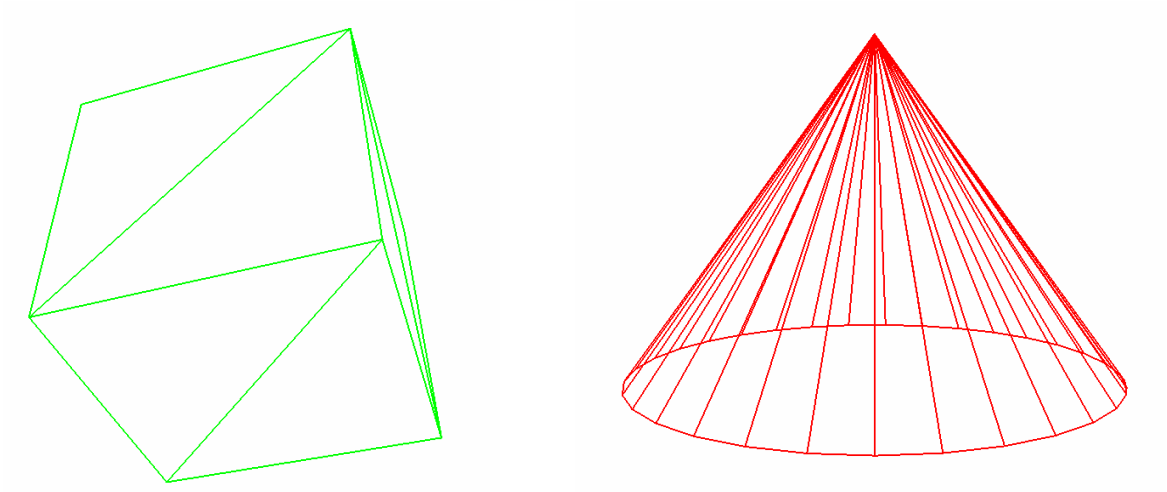
III. Les différentes techniques utilisées pour afficher le paysage

Le but de cette partie n'est pas d'être une explication de la programmation avec OpenGL. En effet de nombreux livres existent sur le sujet. OpenGL n'a été qu'un outil pour l'affichage.

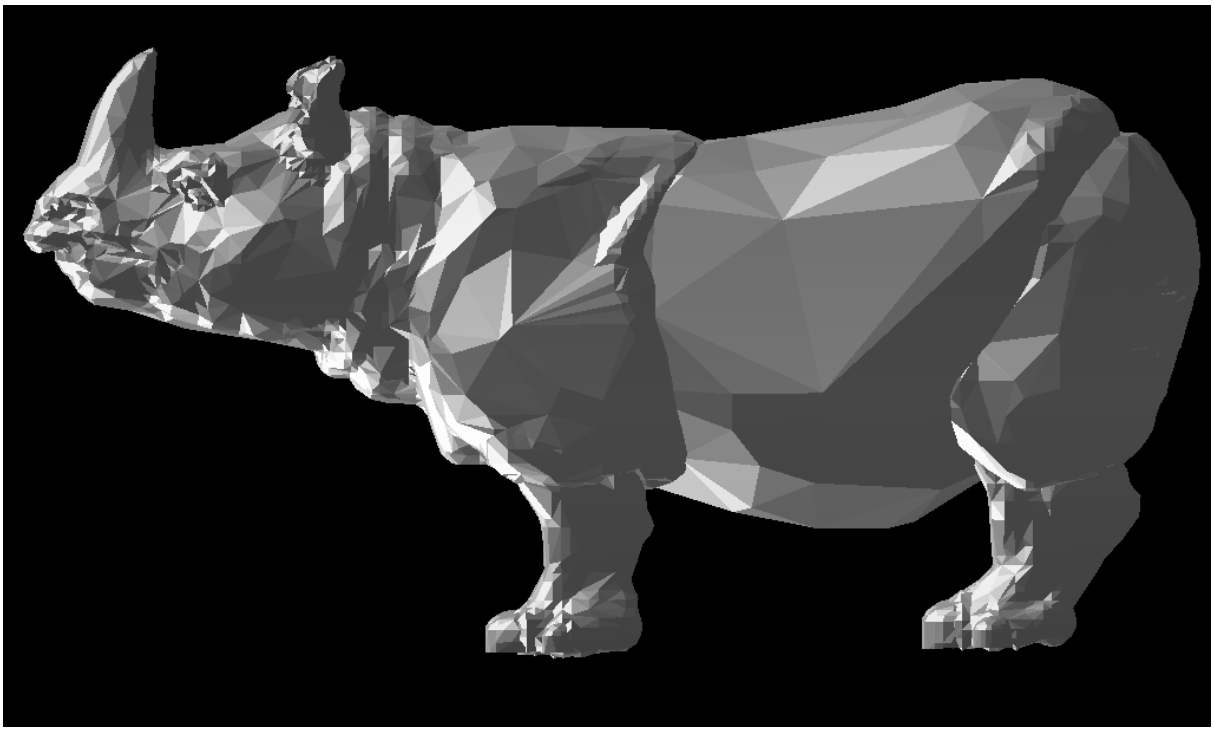
Dans ce paragraphe, nous allons voir pourquoi nous avons choisi les triangles comme forme de base. Ensuite nous verrons la technique d'illumination choisie. Enfin nous verrons l'application des textures. Pour toutes ces techniques nous nous sommes servis des principes décrits dans [MALGOUYRES2002] et [WOO2000 et coll]

III. 1. Les triangles

Pour afficher le paysage de nombreuses options s'offraient à nous. Comme le générateur de paysage crée des triangles, nous avons affiché le paysage sous forme de triangles. Mais il y a d'autres raisons à ce choix. En effet, grâce aux triangles nous pouvons rendre toutes les formes géométriques. De plus, si nous voulons utiliser sur une autre API pour effectuer l'affichage, le triangle est la forme de base choisie pour la plupart d'entre elles (cf DirectX).



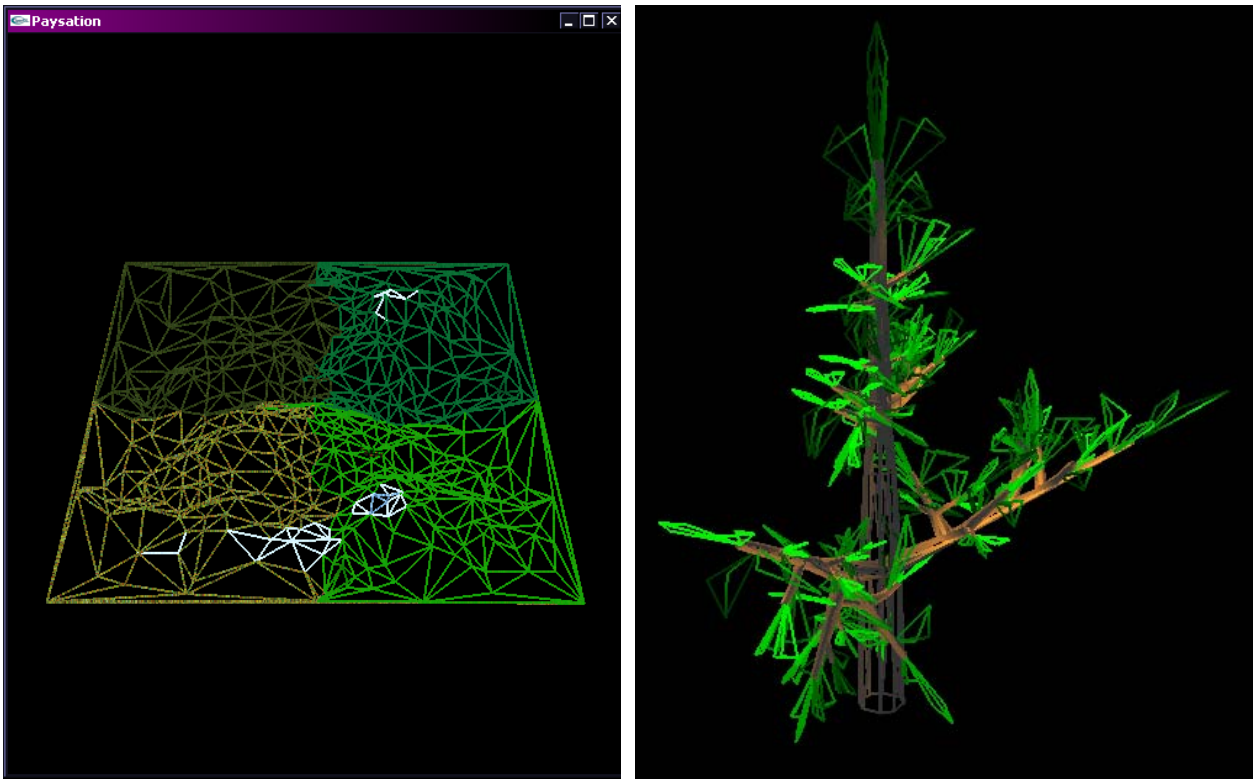
a) un cube tracé seulement avec des triangles *b) un cône composé uniquement de triangles*



c) un rhinocéros créé à partir de triangles.

Figure 24 : différentes figures tracées avec des triangles

Nous allons maintenant présenter des captures d'écrans de notre application où l'on voit que le paysage est affiché sous forme de triangles.



a : paysage affiché sous forme filaire *b : arbre affiché sous forme filaire*

Figure 25 : Différentes images de formes filaires

Dans ces 2 figures (figure 25), l'éclairage est activé. Pour le paysage, les textures sont également activées. Même si le rôle de cette partie n'est pas d'expliquer la programmation OpenGL, nous allons quand même montrer comment nous avons réalisé les différentes opérations mises en œuvre.

Pour tracer un triangle en OpenGL, il n'y a rien de plus facile :

- on indique à OpenGL qu'on va tracer des triangles ;
- on donne les points pour tracer les triangles, OpenGL va les regrouper par groupe de 3 ;
- on indique à OpenGL qu'on a fini de tracer notre figure.

Cette n'est donc pas compliqué.

III. 2. La lumière

Dans les figures ci-dessus, l'éclairage est appliqué. Pour pouvoir appliquer de l'éclairage à une figure, il faut que cette figure possède des composantes matérielles ambiantes, diffuses et spéculaires. Ainsi on pourra avoir des objets qui sont verts avec des reflets rouges par exemple. Un objet sans matériau sera d'une couleur uniforme sous n'importe quelle lumière et où que soit cette lumière.

- La composante ambiante donne l'aspect clair ou sombre à un objet. Plus la composante ambiante d'un objet est proche du noir, plus cet objet semble sombre.
- La composante diffuse peut être assimilée à la couleur de l'objet. C'est en effet cette composante qui rentre le plus dans la composition de la couleur d'un objet.
- La composante spéculaire correspond à la couleur du reflet renvoyé par l'objet.

- La spécularité est la grosseur de ce reflet. Plus ce coefficient est grand, plus le reflet est petit et concentré.

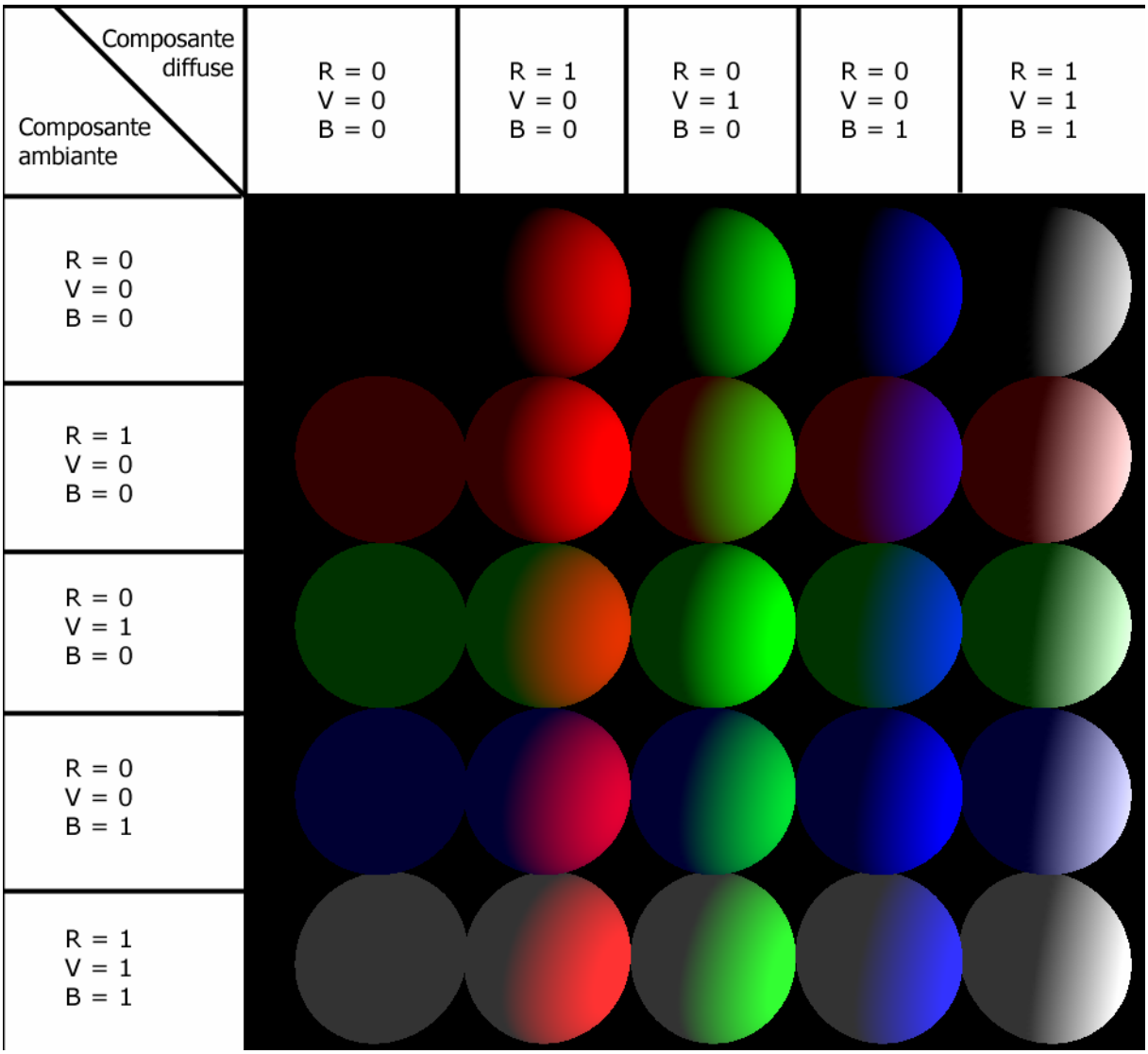


Figure 26 : couleurs obtenues sur une sphère en faisant varier les paramètres de couleur ambiante et diffuse.

Dans la figure 26, la couleur spéculaire reste toujours la même (noire) pour toutes les boules et la lumière est une lumière blanche. Le coefficient de spécularité du matériau est de 0 c'est-à-dire aucun reflet. On s'aperçoit facilement que c'est la composante diffuse qui donne le plus sa « couleur » à l'objet.

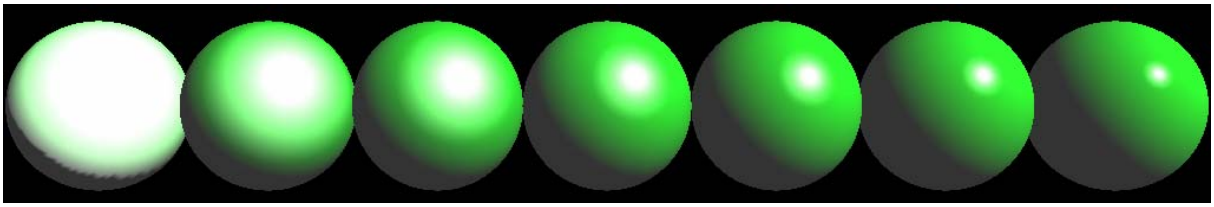


Figure 27 : influence de la spécularité.

Dans la figure 27, la couleur ambiante du matériau est blanche, sa couleur de diffusion est verte et sa couleur spéculaire est blanche. La lumière est blanche. Le coefficient de spécularité varie (1, 5,10, 25, 50, 100, 128 plus précisément). On remarque donc bien linfluence de ce coefficient.

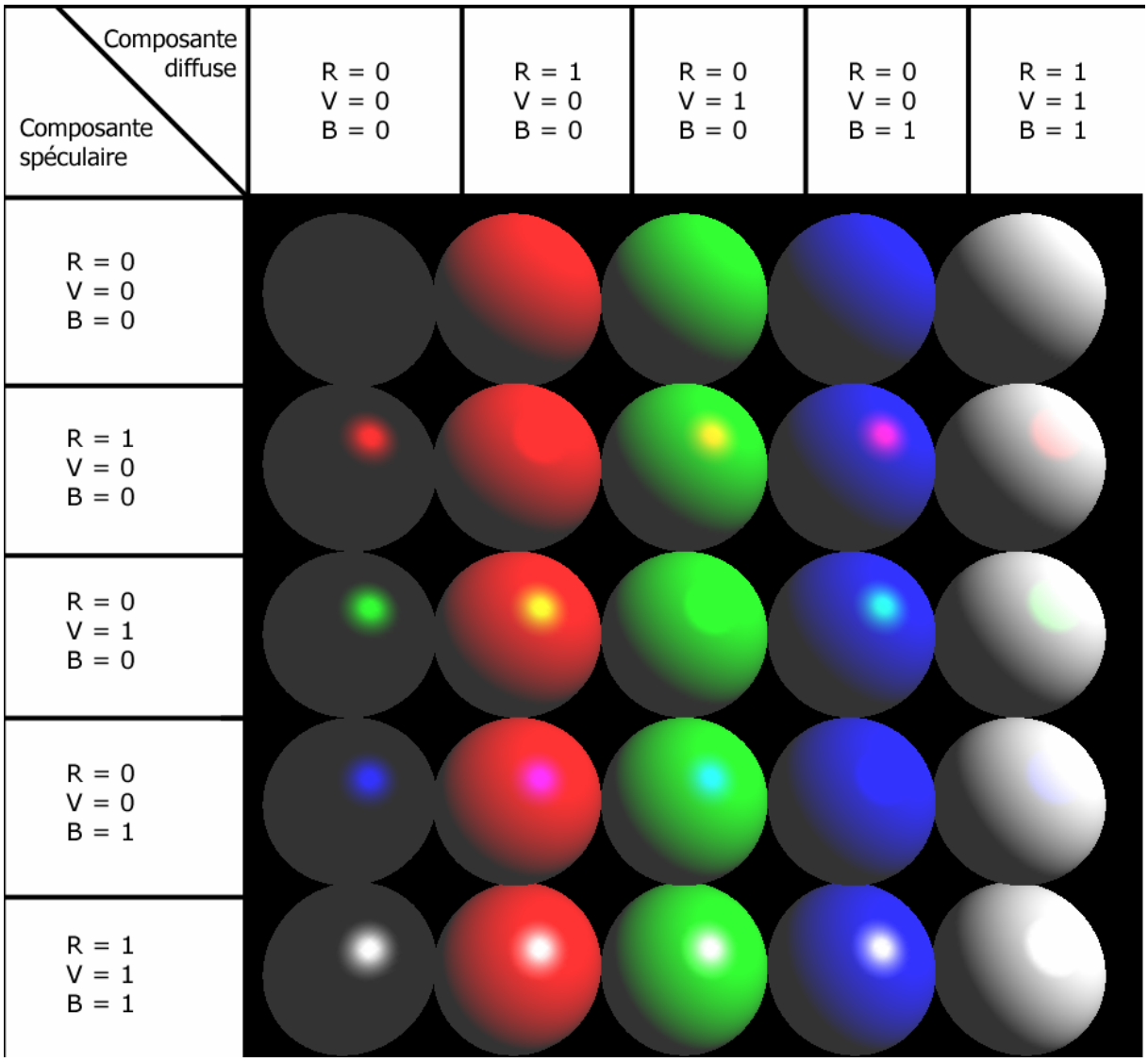


Figure 28 : influence de la couleur spéculaire et de la couleur de diffusion sur la couleur de l'objet obtenue.

Dans la figure 28, la couleur ambiante du matériau est blanche, la lumière est blanche et le coefficient de spécularité vaut 50. On remarque bien la couleur du reflet sur les boules noires et blanches.

La couleur d'un objet ne dépend pas que du matériau de l'objet Elle dépend aussi de la lumière qui éclaire cet objet. La lumière possède les mêmes propriétés que le matériau : une composante ambiante, une composante diffuse et une composante spéculaire.

- La composante ambiante détermine l'intensité de chaque composante (RVB) de la lumière.
- La composante diffuse donne sa couleur à la lumière.
- La composante spéculaire donne une couleur au reflet.

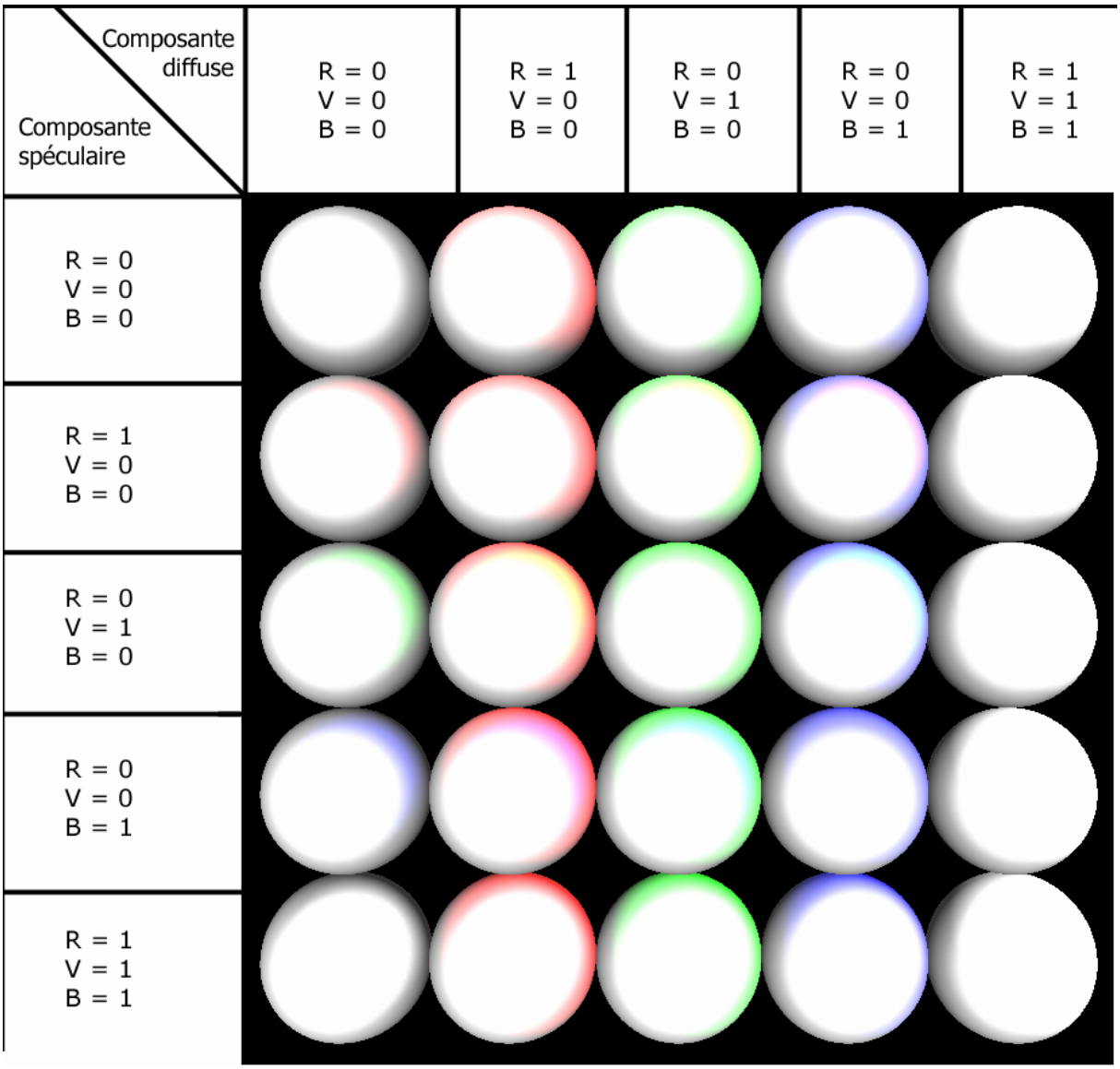


Figure 29 : influence des couleurs de diffusion et spéculaire de la lumière sur une boule blanche.

Sur la figure 29, on remarque bien l'influence de la couleur spéculaire de la lumière quand sa couleur de diffusion est noire (colonne de gauche).

Pour avoir un éclairage réaliste, apposer un matériau à l'objet et attribuer une lumière d'une couleur particulière ne suffit pas. Il faut en effet calculer les normales à chaque triangle. En plus, pour avoir un rendu de paysage qui soit à peu près lisse, il faut effectuer une interpolation sur les normales aux triangles. En fait cette interpolation se rapproche du lissage de Gouraud. Pour calculer la normale à un sommet du triangle, on détermine la moyenne des normales à chaque triangle auxquels appartient le sommet.

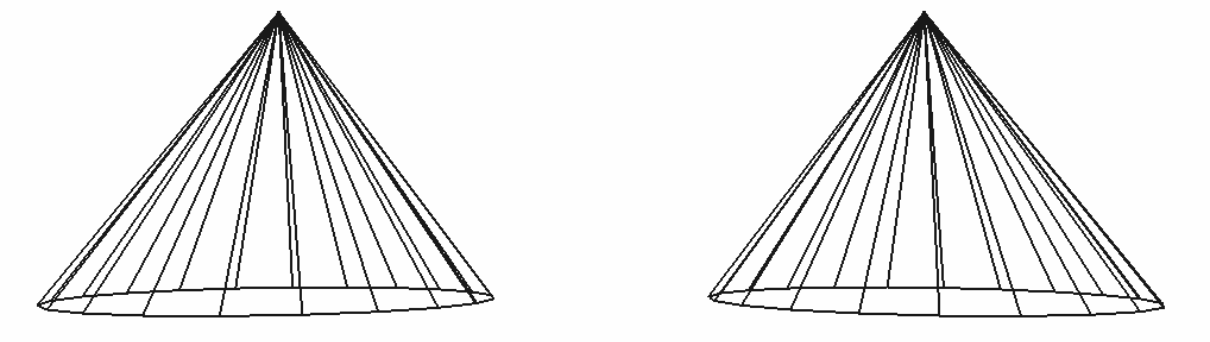


Figure 30 : 2 cônes créés à l'aide de triangles

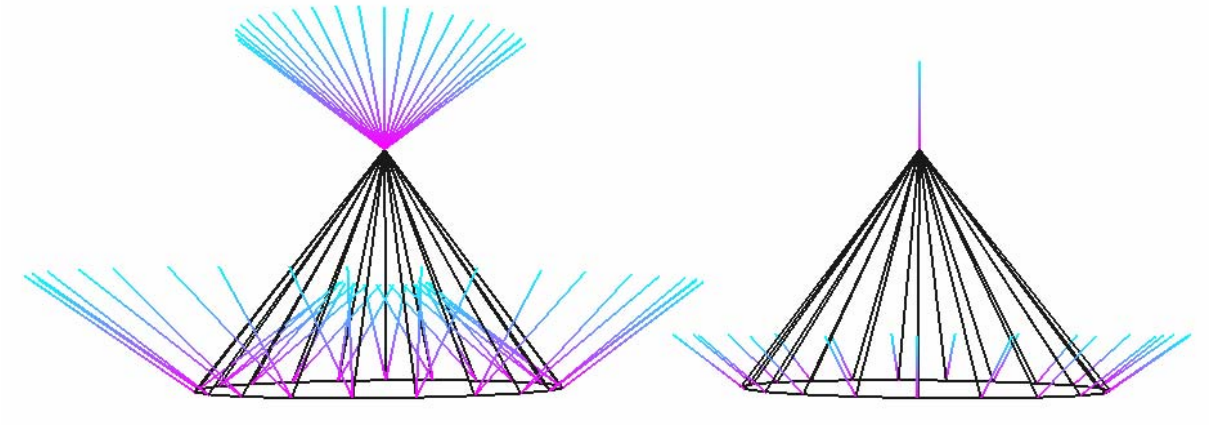


Figure 31 : les normales pour chaque cône (sans et avec moyenne)

Pour le cône de gauche de la figure 31, à chaque point correspond 2 normales, une pour chaque face alors que pour celui de droite, il n'y a qu'une normale par sommet

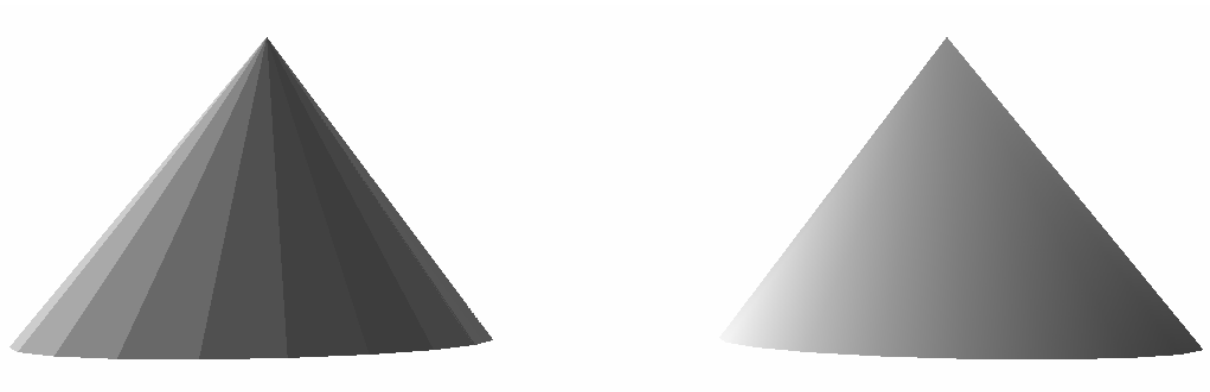


Figure 32 : résultat de l'application d'une lumière.

On remarque bien que toutes les faces sont visibles sur le cône de gauche alors que sur celui de droite, on obtient un cône lisse. Les cônes ont les mêmes propriétés de matériau. Le cône de droite est en fait un paysage.

Nous avons vu comment donner un rendu lisse du paysage et comment lui donner une couleur unie. Il reste maintenant à lui donner un aspect plus crédible en lui ajoutant des textures.

III. 3. Les textures

Avant d'expliquer comment appliquer une texture à un objet, nous allons définir ce qu'est une texture.

Une texture en 2 dimensions est une image qui permet d'ajouter des détails à un objet modélisé. Elle permet aussi d'avoir un objet d'une couleur non uniforme. Ainsi pour avoir un mur de briques modélisé sans texture, il faudrait le créer brique après brique alors que grâce à l'utilisation d'une texture 'mur de briques', un seul rectangle suffit. Une texture 2D est soit lue dans un fichier, soit générée à partir d'algorithmes (algorithme de Perlin par exemple).

Pour gérer le fait de pouvoir lire les textures dans un fichier ou bien de les générer à partir d'un algorithme, nous avons repris le patron composite développé pour les fichiers de terrains.

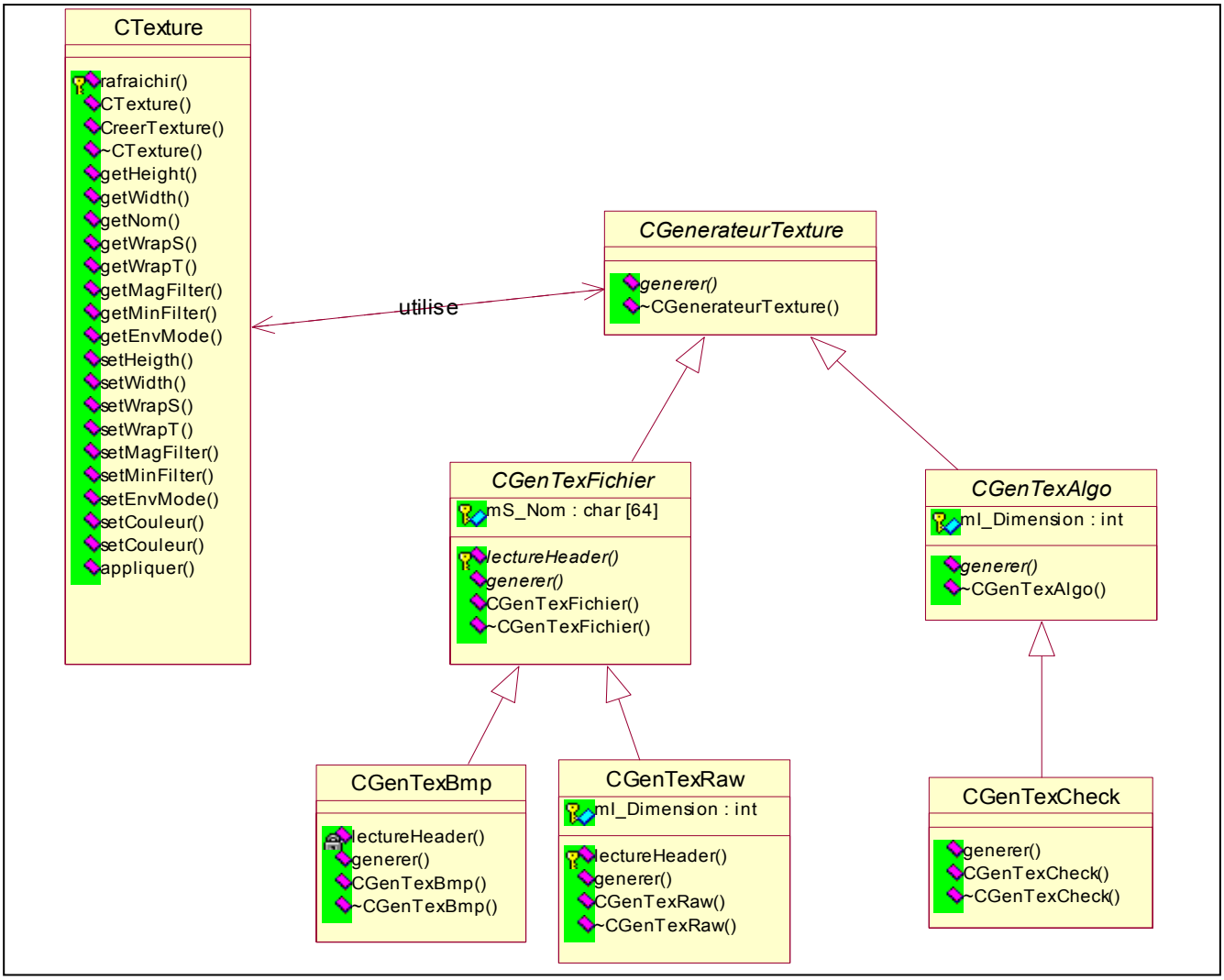
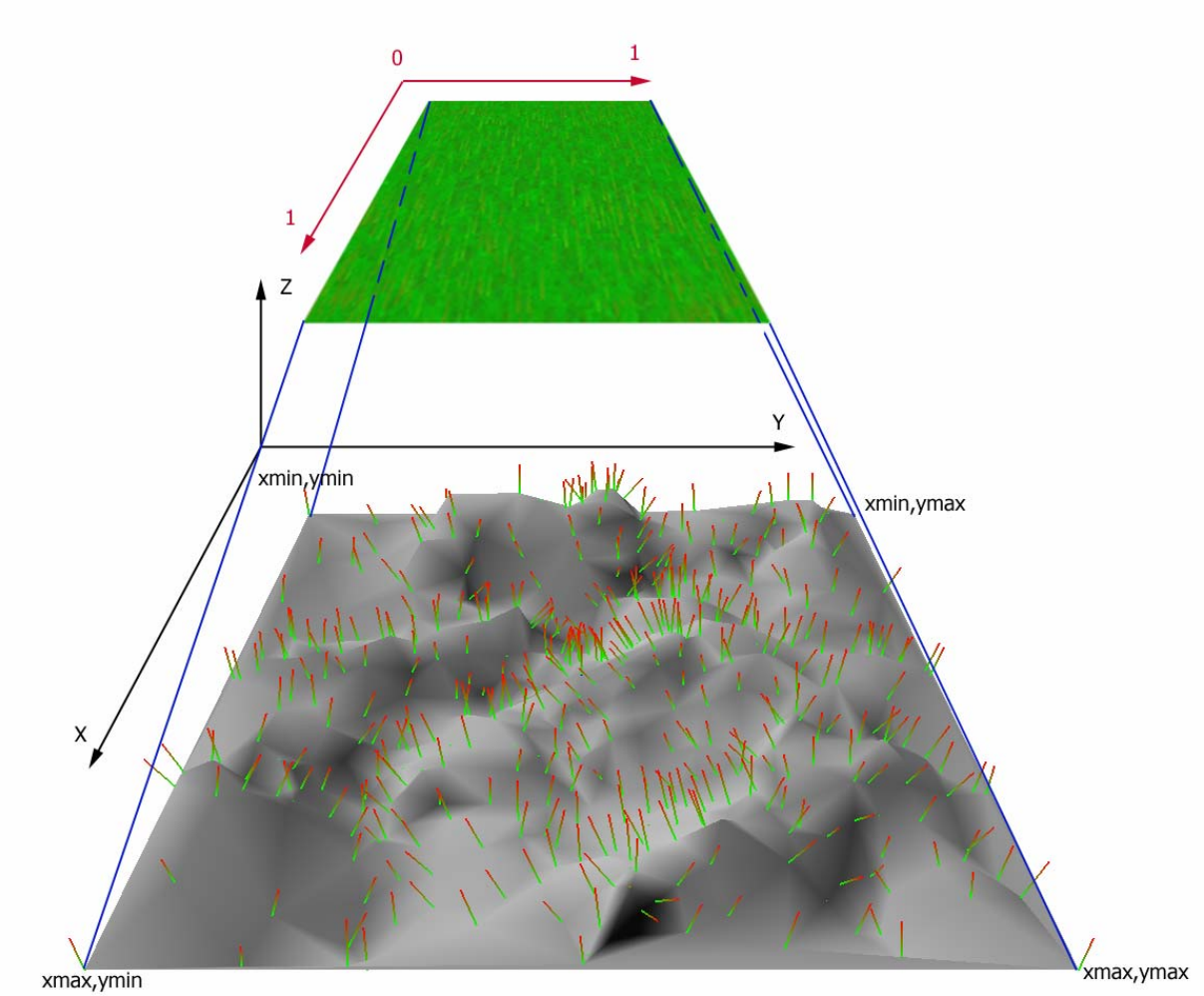


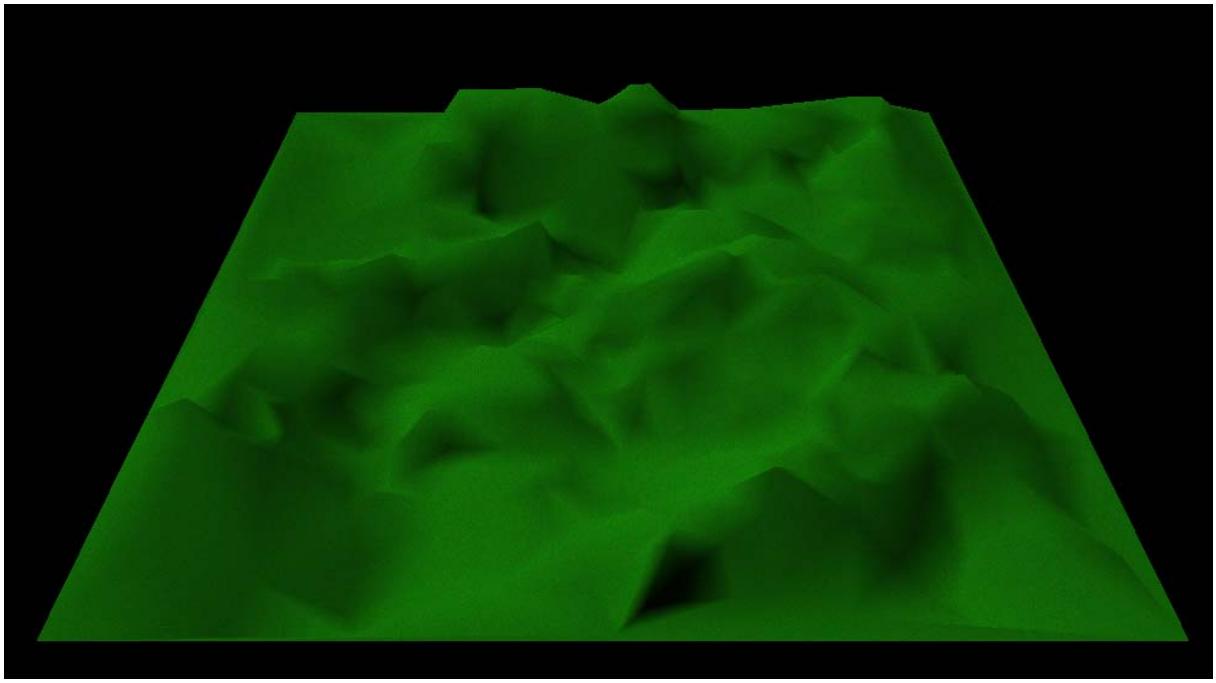
Figure 33 : le patron composite utilisé pour la gestion de textures

Une fois que notre texture est créée, il ne reste plus qu'à l'affecter aux triangles.

Pour cela, il suffit d'affecter des coordonnées de textures à chaque triangle. Pour le paysage, nous avons dit que le point de coordonnées (xmin, ymin) du paysage correspondait au point de coordonnées (0,0) de la texture et le point du paysage de coordonnées (xmax, ymax) correspondait au point (1,1) de la texture. Ainsi, chaque point du paysage possède des coordonnées dans 2 systèmes : dans le système cartésien pour le placement dans l'espace et dans le système de la texture pour pouvoir être texturé.



a) correspondance système cartésien – système de la texture

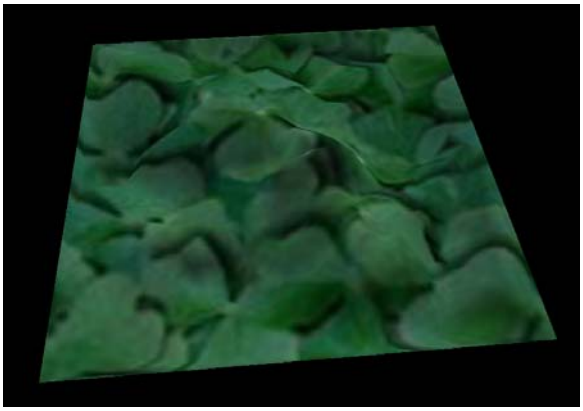


b) résultat obtenu

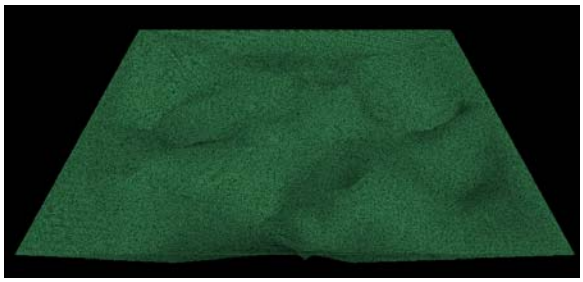
Figure 34 : correspondance entre les systèmes de mesures et résultat obtenu

En fait, le point (xmax, ymax) ne correspond pas au point (1,1) de la texture mais au point (512,512). En effet, nous lisons nos textures dans des fichiers. Le format de ces textures est au maximum de 512 pixels par 512 pixels. Donc pour avoir des textures qui ne soient pas trop étirées, nous allons les répéter.

Texture utilisée



a) résultat avec la texture étirée



b) résultat avec la texture répétée

Figure 35 : différence de résultat sur le même terrain avec une texture étirée et une texture répétée

Nous pouvons les répéter car elles ont été créées dans ce but. En effet, il n'y a pas d'effet de coupure sur les bords lorsque l'on met 2 textures l'une à côté de l'autre.

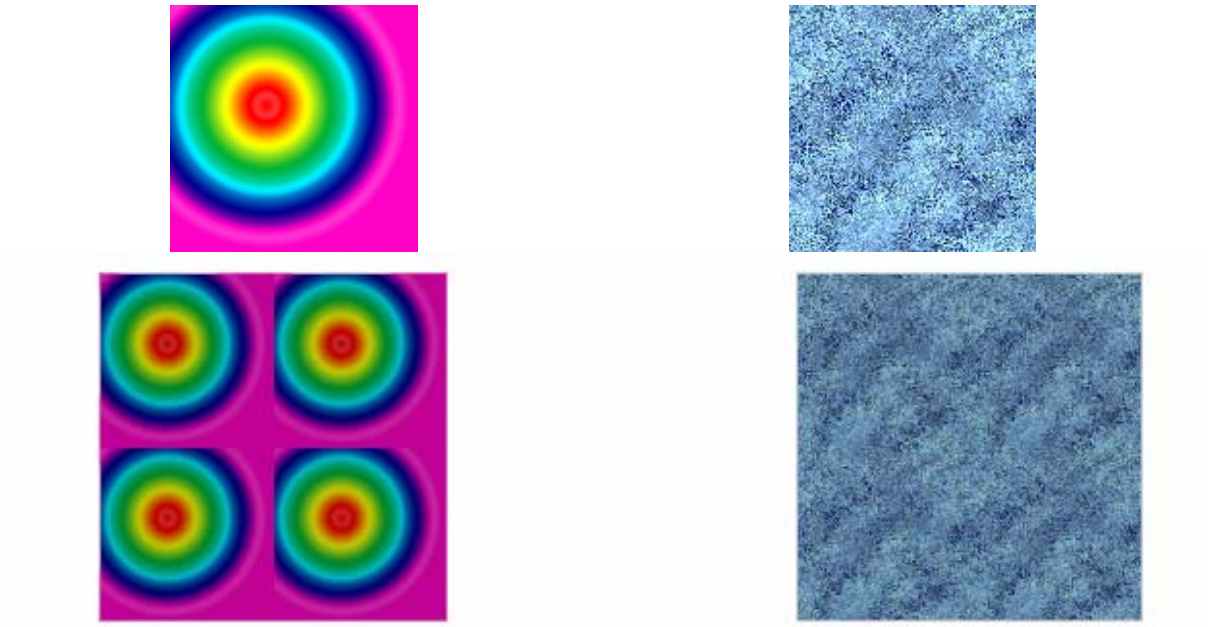
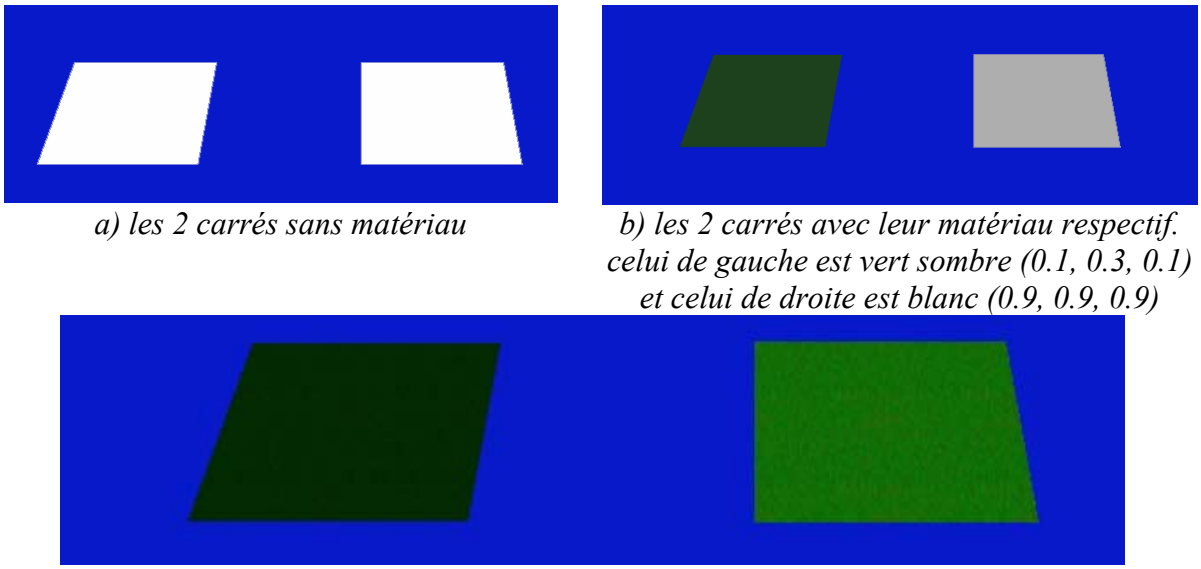


Figure 36 : différence entre une texture étudiée pour se répéter et une qui ne l'est pas.

Sur la figure de droite de la figure 36, il semble n'y avoir qu'une seule texture. C'est un principe désormais classique dans els jeux vidéo modernes.

Pour mettre de la neige, nous avons simplement fait un test sur l'altitude des points. Ensuite nous avons appliqué une texture de neige.

Lorsque le terrain est éclairé et qu'aucune texture n'est appliquée, ce dernier est blanc. Le matériau utilisé pour le terrain est blanc afin de faire ressortir les textures. Plus le matériau employé va être sombre, moins la texture va être visible et ce même si la lumière est très forte. Ceci est un gros problème avec les textures.



a) les 2 carrés sans matériau

b) les 2 carrés avec leur matériau respectif. celui de gauche est vert sombre (0.1, 0.3, 0.1) et celui de droite est blanc (0.9, 0.9, 0.9)

c) les 2 carrés texturés avec la même texture (texture de la figure12)

Figure 37 : résultat de l'application d'une même texture sur des matériaux différents

Nous avons donc présenté les différents éléments qui nous ont servi à afficher le terrain ainsi que les arbres. Bien que notre but n'était pas de présenter un guide OpenGL (comme précisé plus haut), nous allons tout de même présenter une particularité d'OpenGL : les listes d'affichage.

III. 4. Les listes d'affichage

Les listes d'affichage permettent de créer un motif pour pouvoir l'afficher n'importe où. Ainsi, il existe une liste d'affichage par arbre, puis nous déplaçons cette liste pour chaque arbre que nous voulons afficher. Il ne reste plus qu'à réaliser les homothéties et les rotations sur cette liste pour afficher l'arbre désiré. Cette liste d'affichage est normalement enregistrée directement dans la mémoire de la carte graphique et son accès est donc très rapide pour l'affichage.

Nous avons présenté dans cette partie les différentes techniques utilisées pour l'affichage. Ces techniques, exceptées les listes d'affichage, peuvent être utilisées avec les 3 autres API ainsi que les bibliothèques graphiques usuelles telles que la XLib sous Unix ou le GDI sous Windows.

Il faut noter que toutes les images de cette partie, exceptée le rhinocéros, ont été réalisées avec notre application.

PARTIE 2 :
RESULTATS ET DISCUSSION

I. Résultats

Nous avons abouti à un logiciel fini, dans le sens où les fonctionnalités ont été analysées, développées puis testées.

Les besoins définis au départ ont été remplis.

I. 1. Les maillages et les forêts

Nous sommes donc arrivés à modéliser différents paysages avec des forêts variées.

L'interaction avec l'utilisateur nous montre le fonctionnement par composant. En effet, l'utilisateur peut choisir différents formats de fichiers.

I. 2. Le rendu Graphique

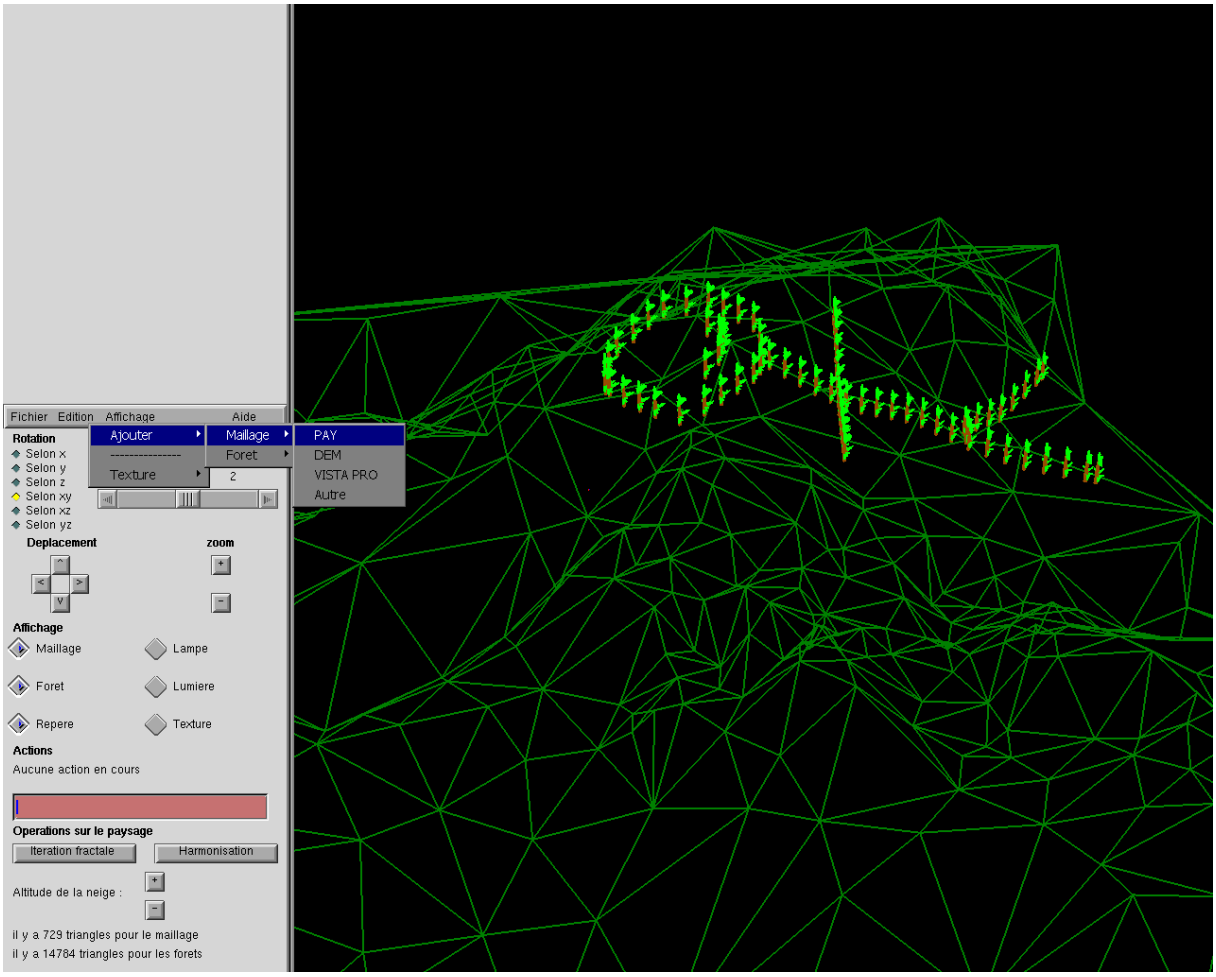


Figure 38 : un maillage et une forêt en mode filaire

On voit ici la structure d'un terrain (un maillage avec des triangles) et une forêt « collée sur ce terrain ». C'est l'illustration de la partie génération de paysages et de forêts.

Cependant ces résultats ne sont pas suffisants pour un rendu réaliste. Voyons maintenant ce que nous a apporté le travail de l'infographie (lumière, texture).

Le travail sur les lumières dans un premier temps nous a permis d'obtenir des paysages assez réalistes. Le jeu des ombres met bien en valeur le relief et les aspérités du terrain.

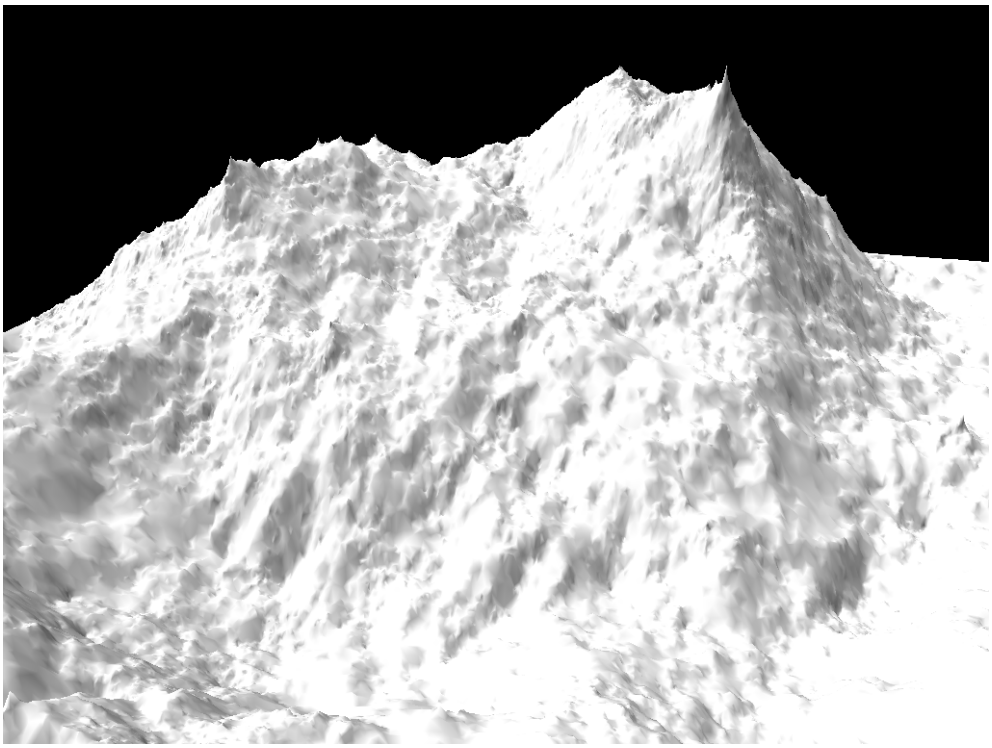


Figure 39 : un premier paysage rendu seulement avec la lumière d'activer

L'ajout des textures a ensuite permis de réaliser des paysages ainsi que des arbres beaucoup plus réalistes.

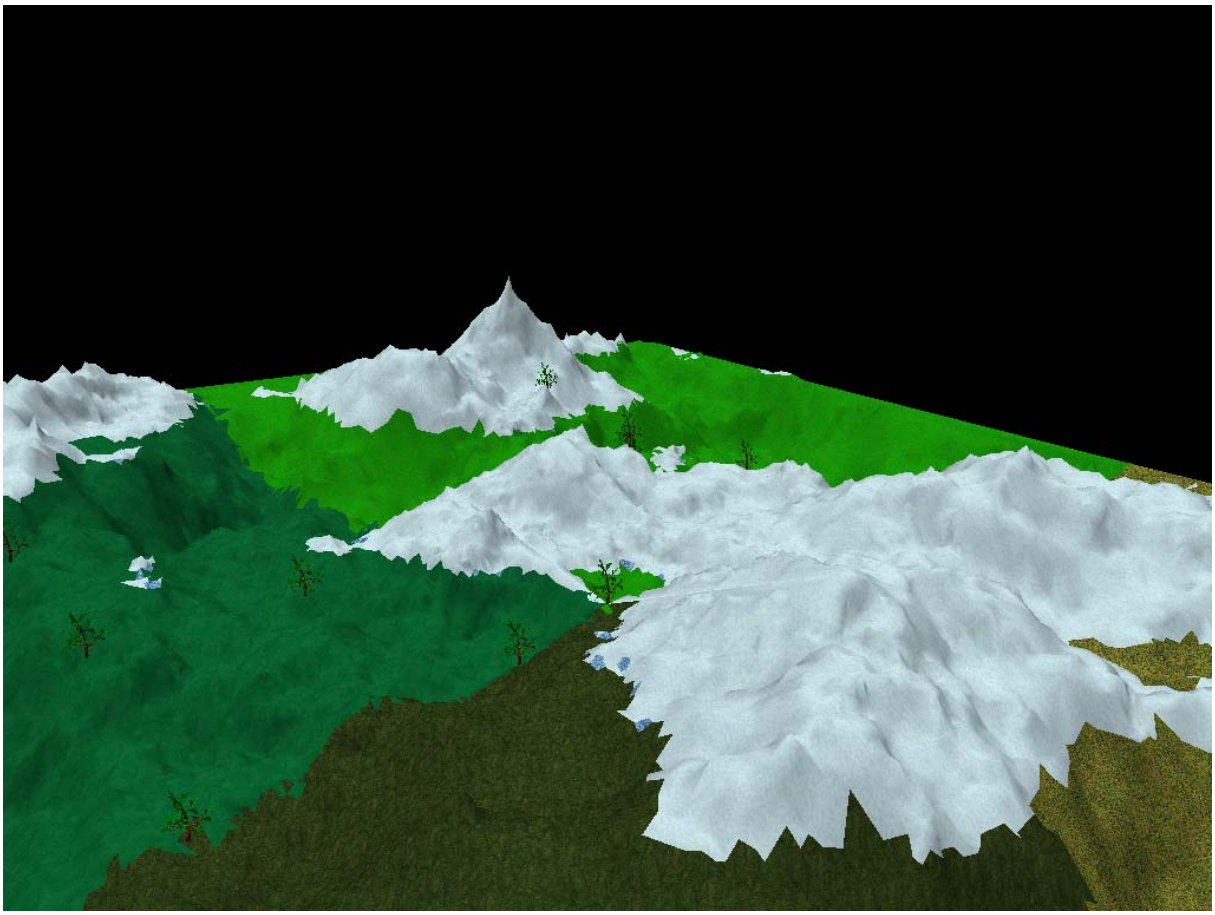


Figure 40 : un paysage avec un rendu graphique optimum

II. Discussion

Cependant, différentes extensions n'ont pas pu être mises en œuvre. Les développements que nous avons effectués ont bien entendu pris en compte cette dimension, et nous pensons notamment à :

- une interaction plus facile concernant le choix des textures pour les paysages : actuellement, on ne peut choisir à quel endroit on désire un type de terrain (choix par l'utilisateur en interactif ou par des standards de fichiers nous indiquant ici on trouve de l'herbe, des pierres…). Pour l'instant seule une altitude et une inclinaison déterminent la présence de neige ou de pierre ;
- la prise en compte de phénomènes physiques (érosion…) ou météorologiques (neige...) ou biologiques (vaches…) ;
- d'autres algorithmes fractals auraient pu être implémentés, afin d'étudier les différents rendus de ceux-ci. Cela est valable pour les terrains mais également pour les arbres ;
- en ce qui concerne les forêts nous avons décidé d'avoir un rendu réaliste avec des structures d'arbres, mais nous n'avons pas eu le temps de tester d'autres méthodes (textures montrant des arbres de loin, …) qui pouvaient s'avérer plus efficaces (cf Annexe C).

Concernant l'organisation du projet, elle s'est déroulée avec une petite équipe, donc la gestion de projet n'était pas trop lourde. Cependant, nous avons pu travailler d'une manière assez indépendante, en prenant les responsabilités des deux grandes parties, et en considérant que les composants logiciels de chacun « fournissaient des services » à l'autre. C'était un des intérêts du génie logiciel Objet (boîte noire).

Nous avons utilisé un processus itératif, en analysant d'une manière globale le problème. Ne connaissant pas précisément les difficultés liées aux logiciels d'infographie, c'était la seule méthode possible. Nous avons donc affiné progressivement des choix rencontrés.

Une refonte majeure a été nécessaire au milieu du projet, car nous étions en train d'assimiler les concepts objets, mais nous n'avions pas assez de recul. Cette refonte concernait plus l'organisation du logiciel que les fonctionnalités et algorithmes. Il nous a donc fallu un mois afin de repartir sur de bonnes bases.

CONCLUSION

Au travers de ce projet, nous avons pu développer deux aspects spécifiques. En effet, afin de créer un logiciel représentant l'aspect graphique d'une simulation de croissance de forêts, il a été nécessaire de travailler sur une génération de paysage et sur un rendu graphique.

Le générateur de paysage nous a permis de développer et d'illustrer beaucoup de mécanismes objets ainsi que d'algorithmes. Ceci nous a amené à des contradictions qu'il a fallu gérer. En effet, au cours du développement, il a fallu faire une véritable refonte de certaines classes, car elles ne validaient pas certains concepts.

En ce qui concerne le rendu graphique, nous avons étudié une technologie qui nous était inconnue et avons également dû étudier et tester différentes solutions afin d'avoir un outil optimal.

Nous disons optimal, mais ce n'est pas forcément le plus optimisé et performant, car nous avons dû effectuer certains compromis. En effet, nous nous sommes aperçus qu'il existe parfois une contradiction entre « développer objets » et « optimiser ». Nous avons cherché une optimalité en essayant d'allier ces deux aspects. Un des côtés les plus intéressants était donc de faire des choix autour de ces concepts.

Nous avons choisi certains fils conducteurs pour ce projet : d'abord le développer sous la forme de composants, comme s'il était amené à être utilisé par des personnes n'ayant pas participé (impact sur la conception, la maintenance, la documentation, la protection des composants = leur accessibilité). Un autre fil était de concevoir une solution indépendante. Nous avons en effet mis en valeur deux pôles dans ce projet (générateur de paysage et infographie) et ils se devaient d'être les plus indépendants possibles afin que leurs communications s'effectuent comme s'il s'agissait de deux boîtes noires.

En conclusion, nous pouvons dire que ce projet nous a apporté beaucoup d'éléments techniques. Il nous a permis de prendre conscience de l'importance de l'organisation d'un progiciel (surtout par projection avec l'idée que le logiciel serait utilisé, modifié…).

De nombreux autres aspects auraient pu être développés mais nous avons préféré terminer et valider tous les composants existant et livrer un résultat satisfaisant.

REFERENCES BIBLIOGRAPHIQUES

[BARRA2000], BARRA V., « Eléments pour la synthèse d'images », 2000-2001

[MALGOUYRES2002], MALGOUYRES Rémy, « Algorithmes pour la synthèse d'images et l'animation 3D », Dunod, Paris, 2002

[WOO2000 et coll], WOO M., NEIDER J., DAVIS T., SHREINER D., « OpenGL 1.2 », CampusPress, Paris, 2000

OUVRAGES CONSULTES

BARRA V., « Cours de bibliothèques graphiques », 2000-2001

BARNSLEY M., DEVANEY R., MANDELBROT B., et al, « The Science of Fractal Images », Springer Verlag, New York, 1988

GARCIA B., « La Librairie Standard du C++ »

McCUSKEY M., « Special Effects Game Programming with DirectX », Premier Press, Cincinatti, 2002, pp. 329-388

MICHELON X., « OpenGL : Génération de terrain », *Linux Magazine(Novembre et Décembre 2001*

NVIDIA CORPORATION, « Cg Toolkit : User's Manual, A Developer's Guide to Programmable Graphics », Decembre 2002,
http://developer.nvidia.com/docs/IO/3263/ATT/Cg_Toolkit.pdf

POLACK T., « Focus on Terrain Programming », Premier Press, Cincinatti, 2003

TURNER M., BLACKLEDGE J., ANDREWS P., « Fractal Geometry in Digital Imaging », Academic Press , 1998

ANNEXES

ANNEXE A :
GALERIE D'IMAGES

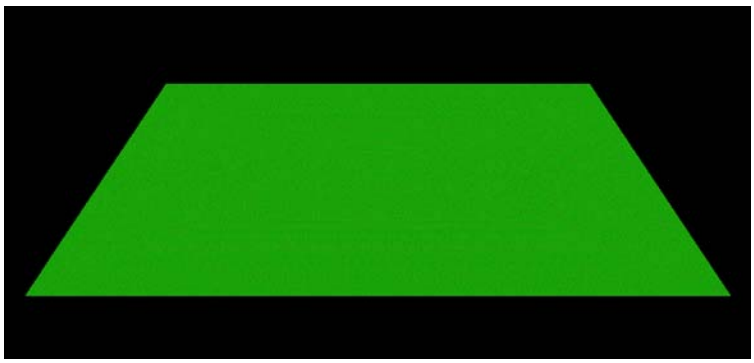


Figure 1 : Terrain de initial (12 triangles)

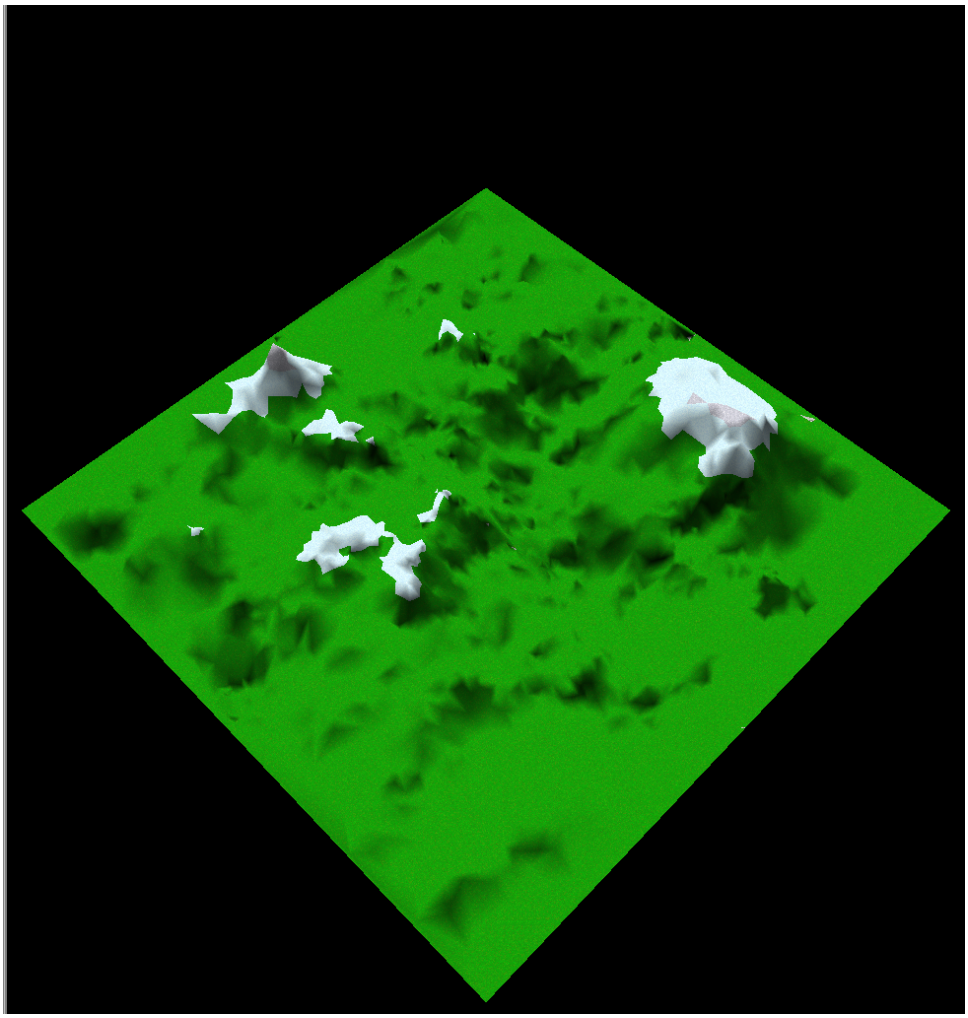


Figure 2 : le même terrain après quelques itérations (8748 triangles)

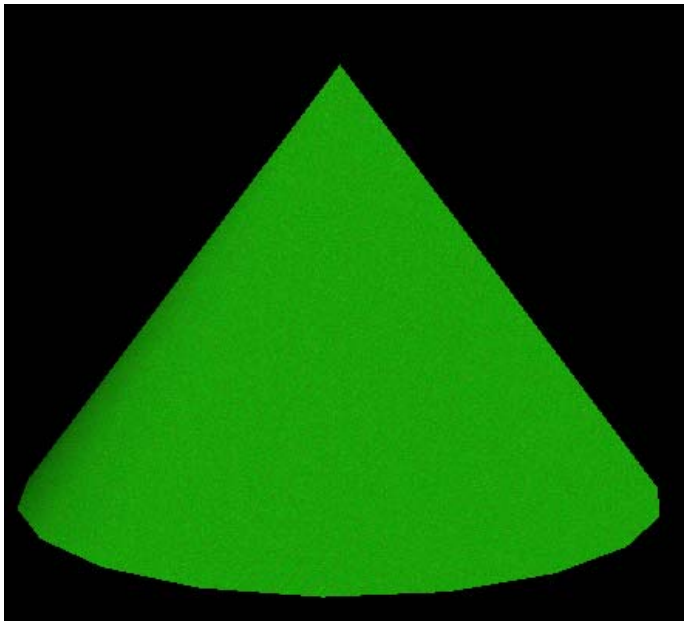


Figure 3 : le cône de la partie 3

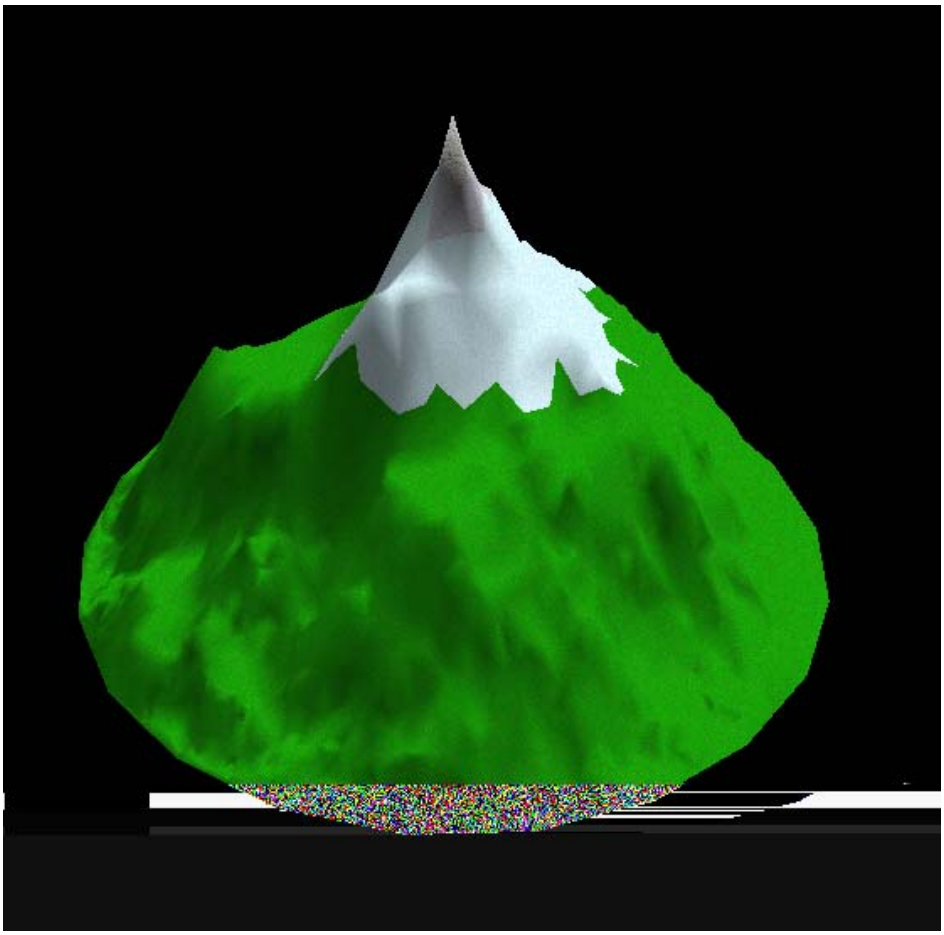


Figure 4 : le même cône après quelques itérations (5700 triangles)

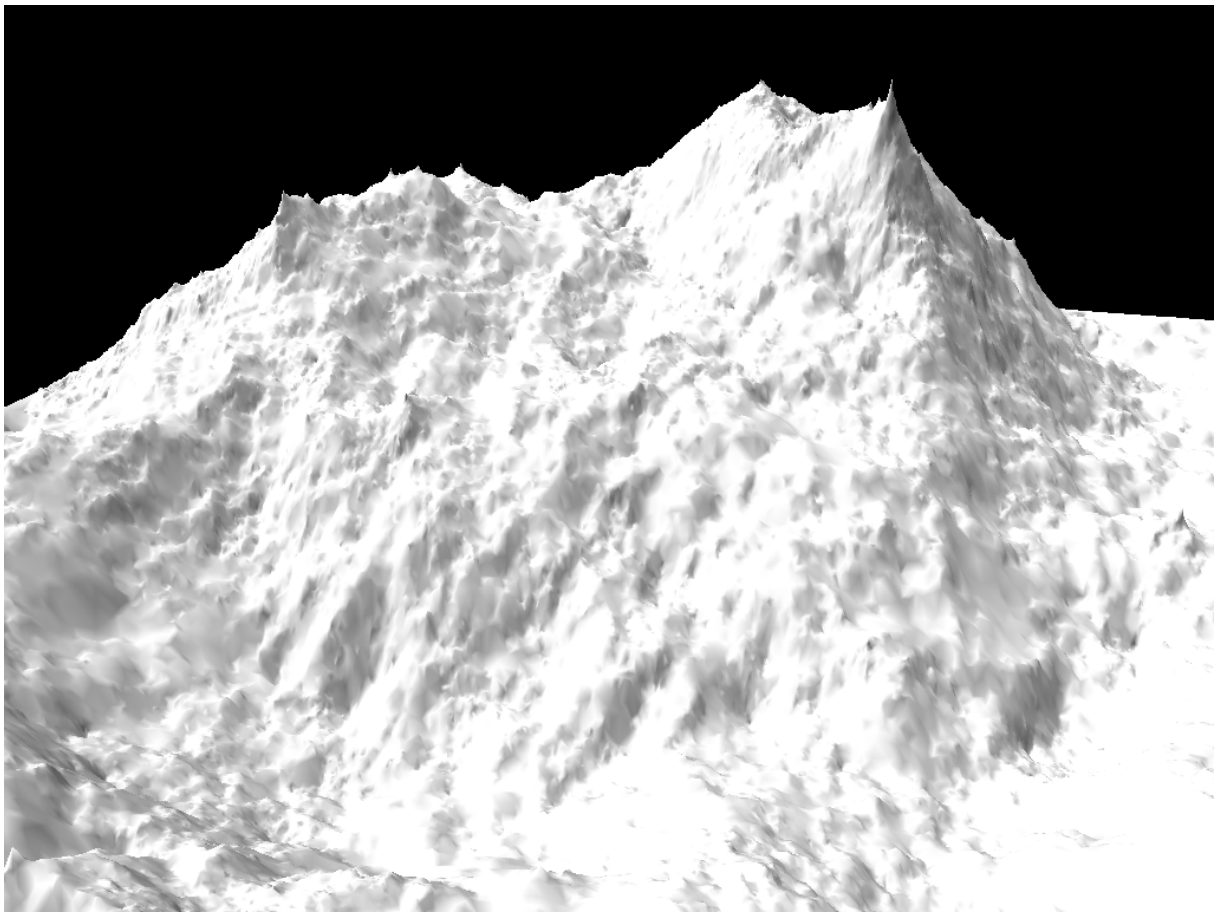


Figure 5 : un paysage enneigé avec plus de 350 000 triangles

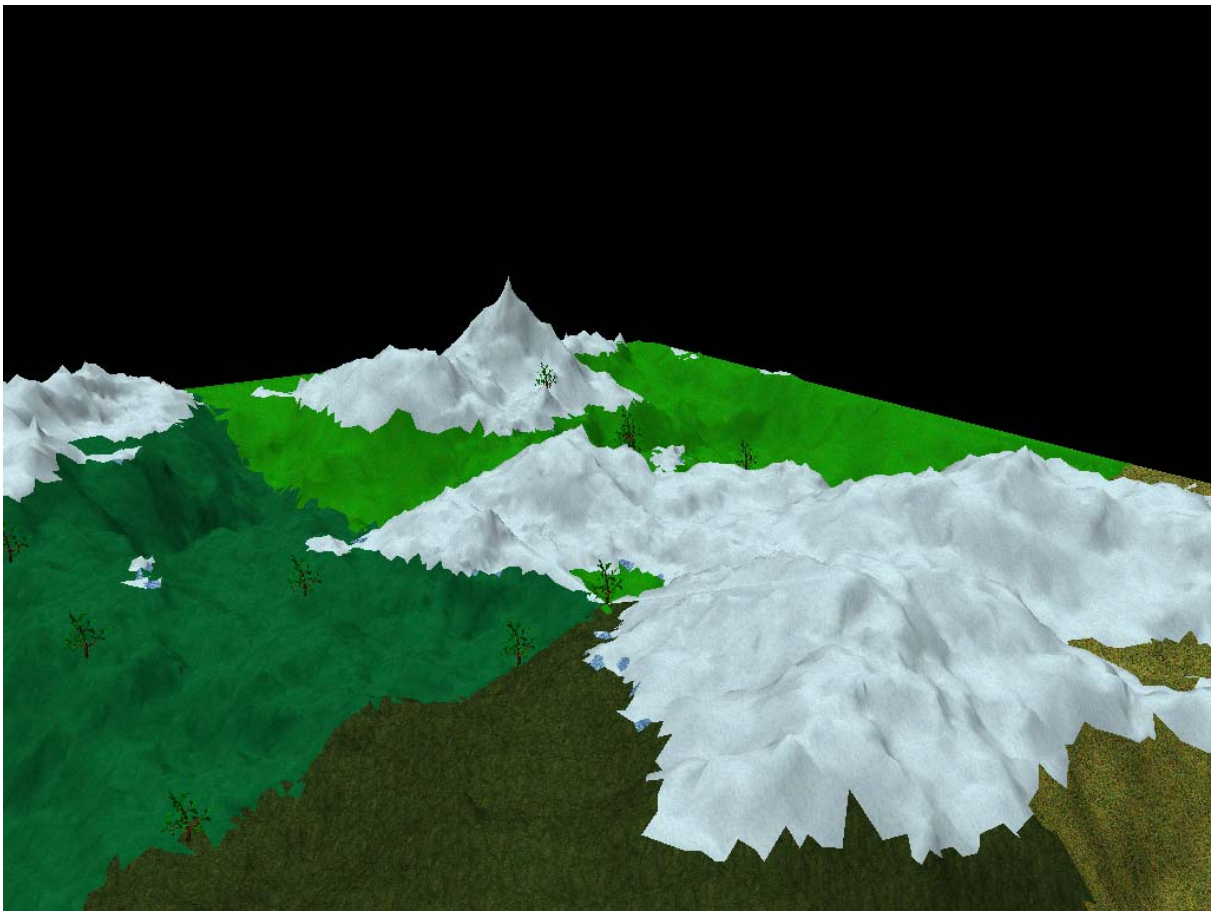


Figure 6 : un autre paysage avec quelques arbres et de la neige mais sans skieurs…

ANNEXE B :
L'INTERFACE

Dans cette annexe, nous allons montrer les différents boutons de l'interface graphique.

I. Le menu

Figure 7 : la barre de menu

I. 1. Le menu Fichier

Figure 8 : le menu Fichier

Dans le menu fichier, il existe 2 options :

- Nouveau : permet de vider l'écran et de recréer un paysage
- Quitter : permet de quitter l'application

I. 2. Le menu Edition

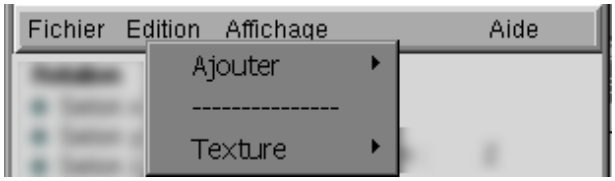


Figure 9 : le menu Edition

Dans le menu Edition, il y a également 2 options.

I. 2. 1. Le sous-menu Ajouter

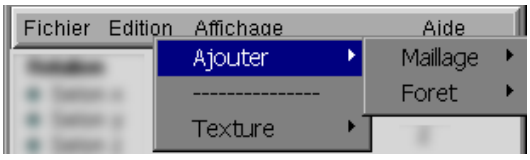


Figure 10 : le sous-menu ajouter

Le menu Ajouter permet d'ajouter des forêts et des maillages. Les maillages sont la base des terrains. Il y a donc 2 options dans ce sous-menu :

- l'option « Maillage » permet de lire différentes sortes de fichiers de paysage (.pay, .dem, .vistapro et autres types de fichiers) ;
- l'option « Foret » permet de lire des fichiers de forêts en choisissant à partir de quel algorithme les arbres vont être créés. Dans le sous-menu L-System du sous-menu Foret, il y a 2 options qui sont Deterministe et Aleatoire. La première indique que tous les arbres de cette forêt auront le même patron. La seconde va permettre la création d'arbres différents pour une même forêt.

I. 2. 2. Le sous-menu Texture

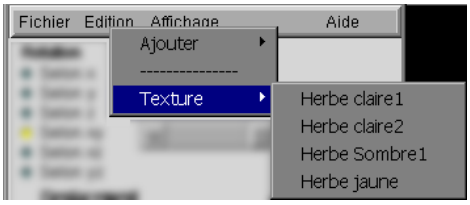


Figure 11 : le sous-menu Texture

Ce sous-menu permet de choisir la texture qui va être appliquée au terrain.

I. 3. Le menu Affichage

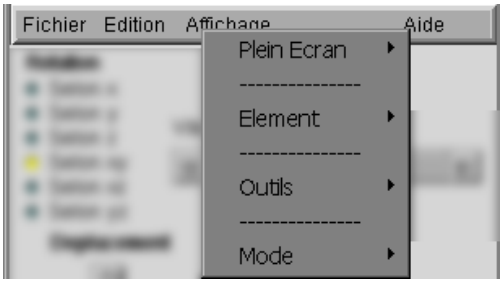


Figure 12 : le menu Affichage

Le menu Affichage va permettre de se mettre en plein écran ou bien d'ajouter ou enlever les textures. Ce menu est accessible en mode plein écran en appuyant sur le bouton droit de la souris. La plupart des options de ce menu sont accessibles directement grâce aux boutons de la barre d'outils.

I. 3. 1. Le sous-menu Plein Ecran

Le sous-menu Plein Ecran possède trois options :

- l'option « avec Barre d'Outils » permet de se mettre en plein écran tout en gardant la barre d'outils sur la gauche ;
- l'option « sans Barre d'Outils » permet de se mettre en plein écran sans avoir la barre d'outils. Cette option est aussi accessible en appuyant sur la touche 'f' lorsque la fenêtre de visualisation est active ;
- l'option « normal » revient à l'affichage de départ.

I. 3. 2. Le sous-menu Element

Le sous-menu Element possède 4 options :

- l'option « Maillage » permet d'afficher ou non le terrain. Un clic sur le bouton Maillage de la barre d'outils produit le même effet ;
- l'option « Foret » permet d'afficher ou non les forêts. Un clic sur le bouton Foret de la barre d'outils produit le même effet ;
- l'option « Repere » permet d'afficher ou non le repère. Un clic sur le bouton Repere de la barre d'outils ou l'appui sur la touche 'r' lorsque la fenêtre de visualisation est active produit le même effet ;
- l'option « Lampe » permet d'afficher l'emplacement de la lumière. Un clic sur le bouton Lampe de la barre d'outils ou l'appui sur la touche 'L' lorsque la fenêtre de visualisation est active produit le même effet.

I. 3. 3. Le sous-menu Outils

Le sous-menu Outils possède 2 options :

- l'option « Lumiere » permet d'activer ou de désactiver l'éclairage de la scène. Un clic sur le bouton Lumiere de la barre d'outils ou l'appui sur la touche 'u' lorsque la fenêtre de visualisation est active produit le même effet ;
- l'option « Texture » permet d'activer ou de désactiver l'éclairage de la scène. Un clic sur le bouton Texture de la barre d'outils produit le même effet.

I. 3. 4. Le sous-menu Mode

Le sous-menu Mode possède 3 options

- l'option « Point » permet l'affichage en mode point. L'appui sur la touche 'P' lorsque la fenêtre de visualisation est activée produit le même effet ;
- l'option « Filaire » permet l'affichage en mode fil de fer. L'appui sur la touche 't' lorsque la fenêtre de visualisation est activée produit le même effet ;
- l'option « Plein » permet l'affichage en mode plein. L'appui sur la touche 'T' lorsque la fenêtre de visualisation est activée produit le même effet.

Le sous-menu aide n'est pas encore implémenté.

II. La barre d'outils

La barre d'outils regroupe les différentes options de déplacement au sein de la fenêtre de visualisation.

Il y a 6 champs dans cette barre d'outils qui sont : « Rotation », « Deplacement », « zoom », « Affichage », « Actions » et « Operations sur le paysage ».

II. 1. Le champ Rotation

Il y a 2 parties bien distinctes dans ce champ : une partie qui permet d'indiquer autour de quel(s) axe(s) va(vont) se produire les rotations pour le paysage et une partie qui va jouer sur la sensibilité de la souris pour les rotations.

Figure 13 : le champ Rotation

II. 2. Le champ Deplacement

Le champ Deplacement possède 4 boutons qui permettent de déplacer le paysage. Les 4 flèches du clavier ont le même effet sur le paysage lorsque la fenêtre de visualisation est active.

Figure 14 : le champ Deplacement

II. 3. Le champ zoom

Le champ zoom permet de zoomer et de dézoomer sur le terrain. Les touches page suivante et page précédente ont le même effet lorsque la fenêtre de visualisation est active.

Figure 15 : le champ zoom

II. 4. Le champ Affichage

Le champ Affichage permet de choisir les options d'affichage pour le paysage. Ce sont les mêmes options que celle du menu Affichage (cf I.3)

Figure 16 : le champ Affichage

II. 5. Le champ Actions

Le champ Actions va permettre de saisir les différents noms de fichiers demandés. C'est aussi dans ce champ que seront affichés les différents messages d'erreur.

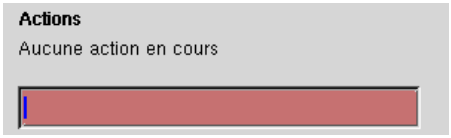


Figure 17 : le champ Actions

II. 6. Le champ Operations sur le paysage

Ce champ a 2 fonctions principales :

- permettre de réaliser les itérations fractales, l'harmonisation des triangles et le réglage l'altitude de la neige ;
- renseigner sur le nombre de triangles.

Pour les itérations fractales, l'harmonisation des triangles et l'altitude de la neige, les différentes opérations s'effectuent grâce à des boutons. Une itération fractale peut être effectuée grâce à la touche 'i' si la fenêtre de visualisation est active. L'altitude de la neige peut être réglée avec les touches '+' et '-'.

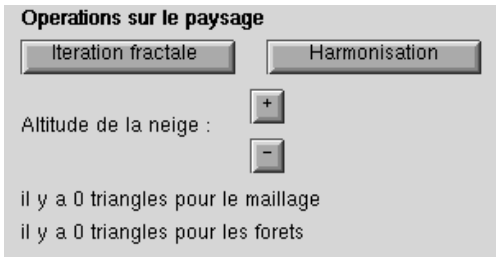


Figure 18 : le champ Operations sur le paysage

ANNEXE C
IMAGES D'ARBRES PAR CROISEMENT D'IMAGES

Figure 19 : une première image tirée de GTAIII™

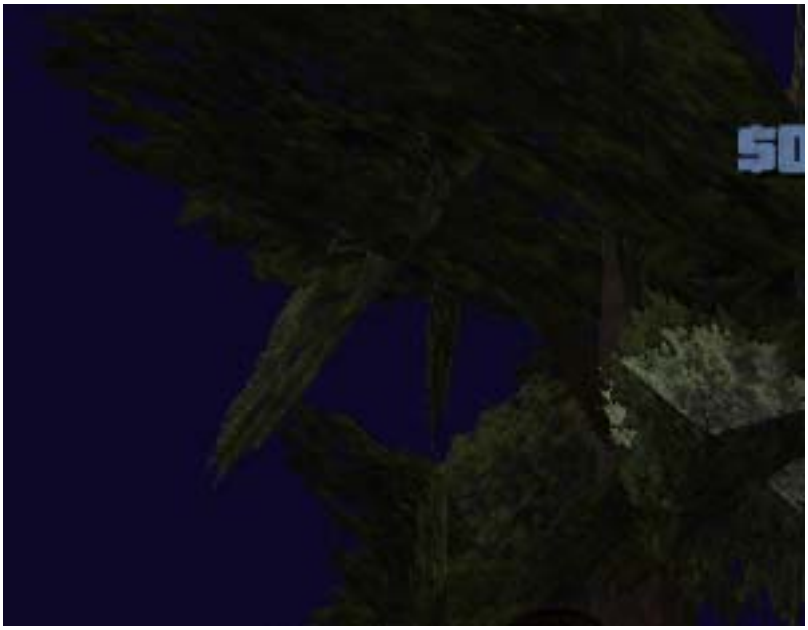


Figure 20 : une seconde image tirée de GTAIII™