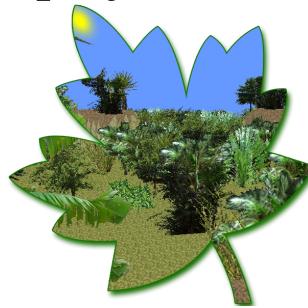


M1 Informatique - Module Synthèse d'images

## Rapport du projet : Balade en forêt



MARECHAL Anthony et Benoit

Encadrant : D. Michelucci et M. Neveu

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Gestion d'une caméra contrôlable par l'utilisateur</b>	<b>3</b>
2.1	Principe de fonctionnement d'une caméra Freefly . . . . .	3
2.2	Représentation d'une caméra en trois dimensions . . . . .	5
2.3	Gestion du regard à la souris . . . . .	6
2.4	Gestion des déplacements au clavier . . . . .	7
2.5	Implémentation de la caméra . . . . .	8
<b>3</b>	<b>Génération du terrain</b>	<b>8</b>
3.1	Génération du maillage . . . . .	8
3.2	Algorithme de placement des sommets . . . . .	10
3.3	Algorithme permettant de relier les sommets en triangles . . . . .	10
3.4	Elévation du terrain . . . . .	12
<b>4</b>	<b>Le mode marche</b>	<b>14</b>
4.1	Détermination des trois sommets de la face située en dessous de la caméra . . . . .	15
4.2	Equation du plan de la face à partir des trois sommets . . . . .	17
4.2.1	Calcul de la normale du plan de la face . . . . .	17
4.2.2	Equation de la droite D . . . . .	18
4.2.3	Intersection de la droite D avec le plan P . . . . .	18
4.2.4	Mise à jour de la hauteur de la caméra . . . . .	19
<b>5</b>	<b>Création des arbres et textures</b>	<b>19</b>
5.1	Création des textures . . . . .	19
5.1.1	Chargeur d'image OpenGL . . . . .	19
5.1.2	Création des textures au format targa . . . . .	20
5.1.3	Directives générales pour l'application des textures . . . . .	21
5.2	Génération des arbres et mappage du sol . . . . .	22
5.3	Placement des arbres sur le terrain . . . . .	22
5.4	Fonctionnalités supplémentaires . . . . .	22
<b>6</b>	<b>Conclusion</b>	<b>23</b>
<b>7</b>	<b>Bibliographie</b>	<b>24</b>

## Table des figures

1	Gestion de la caméra : l'orientation de gauche à droite . . . . .	4
2	Gestion de la caméra : l'orientation de bas en haut . . . . .	4
3	Gestion de la caméra : le mouvement avancer / reculer . . . . .	5
4	Gestion de la caméra : le mouvement gauche / droite . . . . .	5
5	Vecteurs représentant une caméra . . . . .	6
6	Coordonnées sphériques. Le vecteur forward représente l'orientation d'une caméra dans l'espace. . . . .	6
7	Rappels trigonométrique. . . . .	7
8	Schéma de la première version du maillage du terrain . . . . .	9
9	Schéma du maillage retenu pour notre terrain . . . . .	9
10	Schéma montrant les deux faces à relier lorsque l'algorithme est sur le sommet 0 . . . . .	10
11	Schéma du maillage sans le relief . . . . .	12
12	Fonctionnement de l'élévation d'un maillage à partir d'une heightmap . . . . .	13
13	Schéma représentant ce qu'on doit déterminer pour trouver le point d'intersection M . . . . .	14
14	Schéma de la première version du maillage du terrain . . . . .	15
15	Schéma indiquant la face dans laquelle ce situe la caméra, en fonction de l'équation de la diagonale . . . . .	15
16	Aperçus des textures TGA des arbres et arbustes de la forêt. . . . .	20
17	Textures d'arbres appliquées sur deux faces croisées perpendiculairement. . . . .	21
18	Les différentes textures du terrain. . . . .	22
19	Screenshot de la forêt. . . . .	23

# 1 Introduction

Dans le cadre de notre module Synthèse d’images<sup>1</sup> nous avons eu à réaliser le projet : *Balade en forêt*. Le but principal de ce projet étant d’apprendre les bases de la programmation avec OpenGL.

Ce projet a été mis en place avec le langage C++ et la librairie *Glut* dont nous avons appris à nous servir en TP. Pour créer cette balade en forêt nous avons ainsi dû générer un terrain, celui-ci accueille ensuite une végétation et est englobé dans un décor (soleil, nuage, etc.). Enfin, pour mettre en avant l’aspect ‘balade’, et être le plus réaliste possible, nous avons créé une caméra en mode ‘marcheur’ qui suit le terrain permettant ainsi de se promener.

Lors de la réalisation de ce travail nous nous sommes répartis les tâches de manière à ce qu’une personne mette en place le terrain et les caméras (mode vol libre, génération du terrain et mode marcheur) et l’autre personne crée tous ce qui touche aux décors (la végétation, l’animation des nuages, les textures du sol, etc.).

Le projet est composé de quatre parties. Tout d’abord nous étudierons la mise en place de la caméra Freefly (vol libre). Puis nous expliquerons la génération d’un terrain 3D en OpenGL. Ensuite nous verrons comment nous avons créé une caméra en mode ‘marcheur’ et enfin nous traiterons de l’application de la végétation sur le terrain, des textures et de l’animation de certaines d’entre-elles afin de mettre en place le décor du projet.

## 2 Gestion d’une caméra contrôlable par l’utilisateur

Le premier défi que nous avons eu à relever est l’élaboration d’une caméra entièrement contrôlable par l’utilisateur. Il existe différentes manières de manipuler une caméra dans un environnement trois dimensions, les plus connus sont les caméras TrackBall, comme celle utilisées dans Google Earth et les modeleurs 3D, ce type de caméra est utile pour tourner autour d’un objet, et les caméras Freefly, utilisées essentiellement dans les jeux vidéos pour se déplacer en volant librement dans une scène.

Comme vous pourrez le constater ce type de caméra n’est pas trivial à implémenter en OpenGL, pour notre part nous nous sommes aidés d’un tutorial [10]<sup>2</sup> dont les images réalisées dans cette partie en sont inspirées.

### 2.1 Principe de fonctionnement d’une caméra Freefly

Avec une caméra Freefly on gère le regard avec la souris de la manière suivante :

Un mouvement horizontal de la souris fait tourner la caméra horizontalement autour de la verticale du monde. Ce qui permet de regarder à gauche et à droite, comme indiqué sur la figure [1] page [4].

---

<sup>1</sup>Unité d’enseignement du semestre 1 du Master 1 d’informatique de l’Université de Bourgogne.

<sup>2</sup>Décris la réalisation d’une caméra FreeFly et donne une implémentation en OpenGL / SDL.

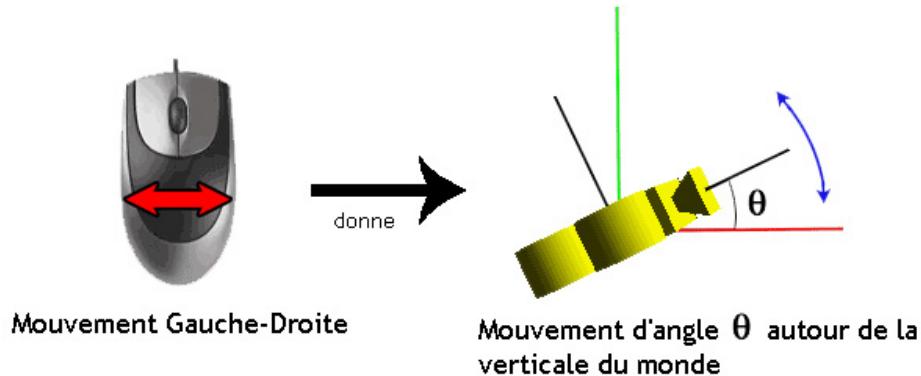


FIG. 1 – Gestion de la caméra : l'orientation de gauche à droite

Un mouvement vertical de la souris fait tourner la caméra verticalement. Ce qui permet de regarder en haut et en bas, comme indiqué sur la figure [2] page [4].

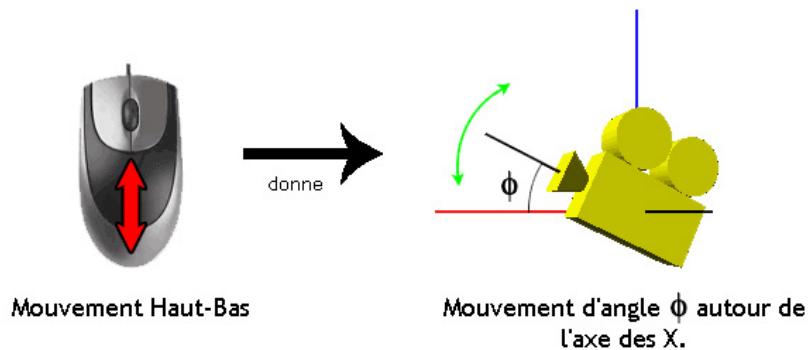


FIG. 2 – Gestion de la caméra : l'orientation de bas en haut

Les déplacements sont quant à eux gérés au clavier de la façon suivante :  
 Deux touches permettent de faire avancer/reculer la caméra, comme indiqué sur la figure [3] page [5].

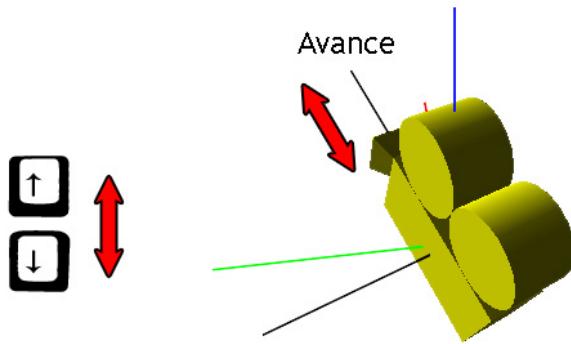


FIG. 3 – Gestion de la caméra : le mouvement avancer / reculer

Deux touches permettent de déplacer latéralement (strafer) la caméra, comme indiqué sur la figure [4] page [5].

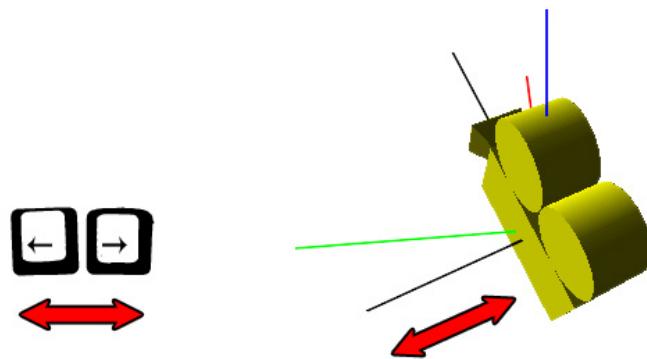


FIG. 4 – Gestion de la caméra : le mouvement gauche / droite

Voyons ensemble la méthode retenue pour représenter ce type de caméra dans un espace en trois dimensions.

## 2.2 Représentation d'une caméra en trois dimensions

OpenGL demande l'utilisation de la fonction *gluLookAt()* pour positionner et orienter la caméra. Les deux informations essentielles à connaître sont donc la position et la cible (*target*) de la caméra. Pour cela, nous maintenons deux vecteurs que nous avons nommés *position* et *target*. Nous avons choisi de les représenter par des vecteurs afin de faciliter leur calculs, mais il est évident que nous aurions pu utiliser tout aussi bien de simples points.

Afin de connaître la direction à suivre lorsqu'on avance et celle lorsqu'on veut se déplacer latéralement nous avons également besoin de deux autres vecteurs que nous avons appelé respectivement *forward* (avancer) et *left* (gauche), voir figure [5] page [6].

Avant de poursuivre les explications, notez que nous avons choisi d'utiliser l'axe des Z pour représenter la verticale du monde, et l'axe des Y pour la profondeur.

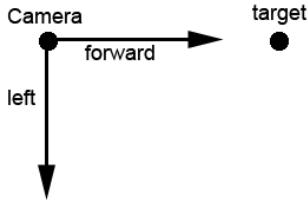


FIG. 5 – Vecteurs représentant une caméra

### 2.3 Gestion du regard à la souris

Pour gérer le regard nous avons associé à un déplacement de la souris à l'écran, deux angles de rotation de la caméra. Le premier angle, appelé  $\theta$ , représente la rotation autour de la verticale et est obtenu en prenant la valeur courante de  $\theta$  moins le déplacement en abscisse de la souris, ce qui s'écrit :

$$\theta = \theta - (x - xold)$$

Le second angle  $\phi$  représente quant à lui la rotation autour de l'axe des abscisses et est obtenu selon le même principe que l'angle  $\theta$  mais avec les ordonnées, d'où :

$$\phi = \phi - (y - yold)$$

La figure [6] page [4] représente l'orientation de la caméra en coordonnées sphériques.

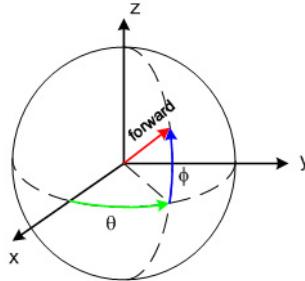


FIG. 6 – Coordonnées sphériques. Le vecteur forward représente l'orientation d'une caméra dans l'espace.

Cependant OpenGL, comme nous l'avons vu, ne gère pas des coordonnées sphériques pour représenter une caméra, mais des coordonnées cartésiennes. Afin d'effectuer la conversion nous avons utilisé nos connaissances en trigonométrie, rappelées brièvement par le schéma suivant :

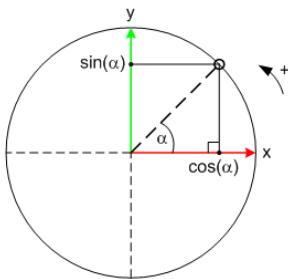


FIG. 7 – Rappels trigonométrique.

Grâce à la trigonométrie nous pouvons alors facilement calculer les nouvelles coordonnées x, y et z de notre vecteur forward :

$$forward.x = \cos(\theta)$$

$$forward.y = \sin(\theta)$$

$$forward.z = \sin(\phi)$$

Remarque : Les fonctions mathématiques d’OpenGL travaillant avec des radians nécessite de multiplier les angles  $\theta$  et  $\phi$  par  $\frac{\pi}{180}$  avant d’effectuer les calculs ci-dessus.

Nous savons désormais comment calculer le vecteur *forward*, recherchons maintenant à déterminer le vecteur des déplacements latéraux : le vecteur *left*. Ce vecteur étant orthogonal au vecteur *forward* et à la verticale du monde (axe des Z), il nous suffit de calculer le produit vectoriel de ces deux vecteurs pour l’obtenir.

La cible (*target*) regardée par la caméra est quant à elle facile à calculer puisqu’il suffit d’ajouter le vecteur *position* au vecteur *forward*.

A ce stade la gestion du regard est fonctionnel nous verrons un peu plus loin le détail des méthodes qui l’implémente. Il reste maintenant à gérer les déplacements de la caméra.

## 2.4 Gestion des déplacements au clavier

La gestion des déplacements est quand à elle plus simple puisqu’il suffit d’ajouter ou de soustraire le vecteur approprié en fonction de la flèche directionnelle enfoncee. Ainsi pour avancer ou reculer la caméra, on additionne ou on soustrait le vecteur *forward* au vecteur *position* et pour se déplacer latéralement à gauche ou à droite on additionne ou on soustrait le vecteur *left* au vecteur *position*.

Remarque : Après avoir modifié la position il est nécessaire de mettre à jour la cible regardée comme indiqué dans la partie précédente.

## 2.5 Implémentation de la caméra

Nous avons implémenté la caméra en utilisant une classe que nous avons nommée *cameraFreeFly*. Cette classe fait appel à une autre classe nommée *vector3D*<sup>3</sup> réalisant des opérations élémentaires sur les vecteurs comme l'addition, la multiplication par des scalaires ou encore la réalisation de produits vectoriels.

La classe *cameraFreeFly* contient les variables suivantes :

- *position* : *vector3D* représentant la position de la caméra.
- *target* : *vector3D*, désignant la cible regardée.
- *forward* : *vector3D*, définissant l'orientation de la caméra. Utilisé pour les déplacements Avant / Arrière.
- *left* : *vector3D*, définissant le déplacement à réaliser lorsqu'on veut se déplacer latéralement.
- *theta* et *phi* : Définissent l'angle de rotation de la caméra selon la verticale et l'abscisse respectivement.

*cameraFreeFly* comporte aussi les méthodes suivantes :

- *VectorFromAngle()* : Cette méthode utilisée pour la gestion du regard, met à jour les vecteurs de la caméra (*forward*, *left* et *target*) en fonction de *theta* et *phi*. Voir les calculs trigonométriques réalisés dans la partie précédente pour plus de détails.
- *OnMouseMotion(int x, int y)* : Méthode désignée à glut pour intercepter et effectuer les opérations nécessaire lors d'un déplacement de la souris. C'est elle qui calcule les angles *theta* et *phi* et qui demande une mise à jour de la caméra avec la méthode *VectorFromAngle()*.
- *OnKeyBoard(int touche, int x, int y)* : Selon la flèche enfoncee, ajoute ou soustrait le vecteur *left* ou le vecteur *forward* au vecteur *position*.
- *look()* : Méthode réalisant l'appel à la fonction *gluLookAt* pour positionner la caméra en fonction des vecteurs *position* et *target*.

## 3 Génération du terrain

Il existe différentes façons de générer un terrain en trois dimensions. Après lecture de plusieurs tutoriaux traitant de ce sujet, nous avons choisi d'implémenter notre propre système de génération de terrain. Pour cela nous avons procédé en deux étapes principales :

1. Réalisation d'une grille de sommets reliés sous forme de maillage de triangles dans le plan du sol (xOy).
2. Elévation du terrain de manière réaliste. Pour cela nous avons retenu le système de heightmap.

### 3.1 Génération du maillage

Dans notre repère nous avons le plan xOy pour le sol. Pour réaliser le maillage nous avons, dans un premier temps, relié les points selon le schéma suivant :

---

<sup>3</sup>La classe *vector3D* a été récupérée sur Internet voir la référence [10]

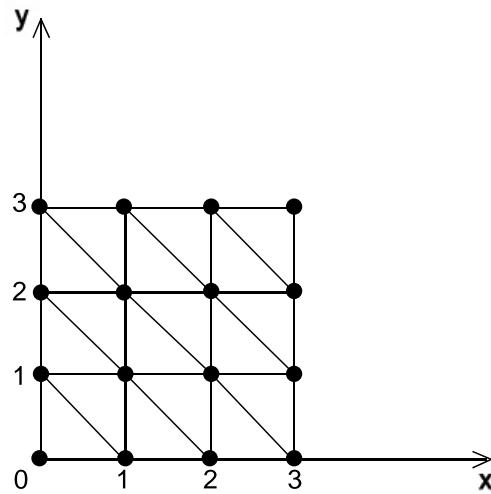


FIG. 8 – Schéma de la première version du maillage du terrain

Cependant l'utilisation de triangles rectangles pour représenter un terrain n'est ni pratique ni esthétique. Afin d'avoir des triangles isocèles nous avons alors translaté de 0,5 en abscisse les sommets des lignes impairs. Ce qui donne le maillage suivant :

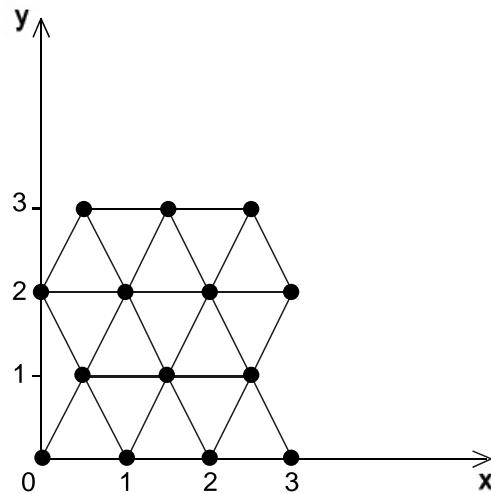


FIG. 9 – Schéma du maillage retenu pour notre terrain

Ce maillage est beaucoup plus réaliste pour représenter un terrain naturel car il est moins symétrique que le maillage précédent.

Pour représenter en trois dimensions en tel maillage, nous avons eu besoin de deux algorithmes :

1. Un algorithme de placement des sommets.
2. Un algorithme qui permet de relier les sommets en triangles.

### 3.2 Algorithme de placement des sommets

Pour un terrain de largeur  $l$  et de longueur  $L$ . On utilise un tableau de *point* - structure ou classe contenant trois variables représentant les coordonnées  $x$ ,  $y$  et  $z$  - de taille  $L*l$ , puisqu'il y aura  $L*l$  sommets dans le maillage pour stocker les sommets.

Cet algorithme consiste à parcourir toute les lignes et toutes les colonnes de la grille et à définir les coordonnées de chaque sommets :

Listing 1 – Pseudo-code de la génération de la grille de sommets

```

1 Pour i de 0 à L
2   Pour j de 0 à L
3     indice = (i*l) + j
4
5     Si on est sur une ligne paire
6       sommets[indice].x = j
7     Sinon
8       sommets[indice].x = j+0,5
9
10    Fin Si
11
12    sommets[indice].y = i
13    sommets[indice].z = 0
14
15  Fin Pour
16 Fin Pour

```

Cet algorithme est assez simple, on notera juste la subtilité consistant à translater de 0,5 en X les sommets des lignes impaires afin d'avoir des triangles isocèles.

### 3.3 Algorithme permettant de relier les sommets en triangles

Afin de générer chaque facette de notre terrain nous avons besoin de définir un nouvel algorithme. Ce dernier va parcourir chaque sommet et va créer deux facettes comme indiquer sur le schéma suivant :

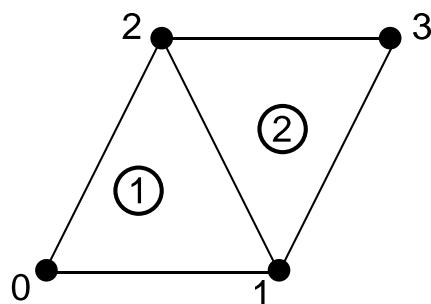


FIG. 10 – Schéma montrant les deux faces à relier lorsque l'algorithme est sur le sommet 0

Le principe général de l'algorithme nécessite, pour chaque sommet  $i$  allant de 0 à tous les sommets ( $L \times l$ ) du tableau de sommets, à créer une face contenant les sommets  $i, i+1$  et  $i+l$  et une face contenant les sommets  $i, i+l, i+l-1$ .

Cependant l'algorithme doit également respecter les contraintes suivantes :

- On ne doit pas parcourir les sommets situés sur la dernière ligne du maillage. Donc la boucle doit s'arrêter à  $L \times (l - 1)$ . On aura donc tracé  $L \times (l - 1) \times 2$  faces au total.
- On ne doit pas essayer de créer de facettes lorsqu'on est sur le dernier sommet d'une ligne. Celui-ci étant désigné par l'expression  $(i + 1) \bmod l = 0$ .
- Lorsque la ligne du sommet  $i$  est impaire les facettes à relier sont les suivantes :  $i, i+l, i+l+1$  et  $i, i+1, i+l+1$

On utilise un tableau à deux dimensions pour stocker les faces désignées par un numéro et trois sommets, ainsi qu'un compteur pour les identifier.

L'algorithme est donc :

Listing 2 – Pseudo-code de l'algorithme reliant les sommets en face

```

1 Pour i de 0 à L*(l-1)
2   Si i/l est paire Faire
3     Si (i+1) modulo 1 différent de 0 Faire
4       face[compteur][0] = i
5       face[compteur][1] = i+1
6       face[compteur][2] = i+l
7
8       compteur = compteur + 1
9     Fin Si
10
11    Si i modulo 1 n'est pas paire Faire
12      face[compteur][0] = i
13      face[compteur][1] = i+1
14      face[compteur][0] = i+l-1
15
16  compteur = compteur + 1
17
18  Fin Si
19
20 Sinon
21   Si (i+1) modulo 1 différent de 0 Faire
22     face[compteur][0] = i
23     face[compteur][1] = i+l+1
24     face[compteur][0] = i+l
25
26  compteur = compteur + 1
27
28
29   face[compteur][0] = i
30   face[compteur][1] = i+l
31   face[compteur][0] = i+l+1
32
33  compteur = compteur + 1
34  Fin Si
35

```

A ce stade nous avons le maillage suivant :

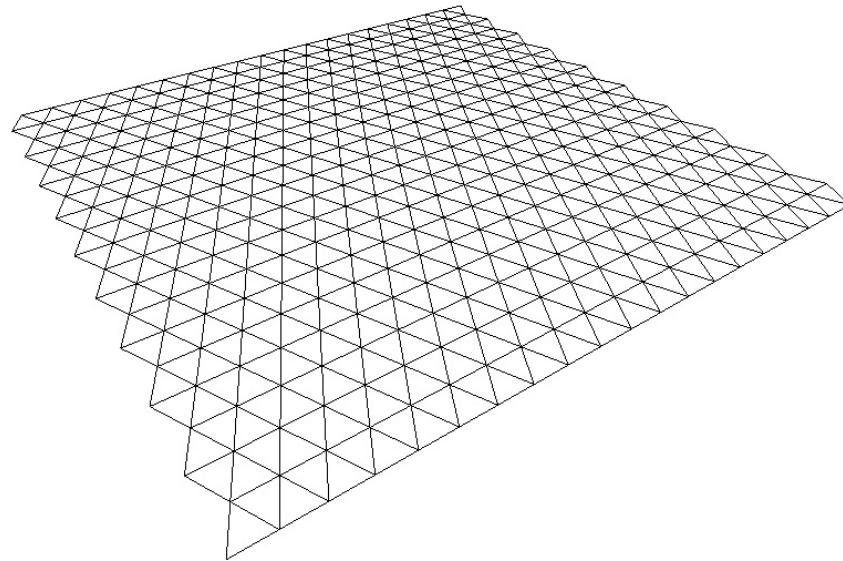


FIG. 11 – Schéma du maillage sans le relief

Comme on peut le remarquer ce terrain manque de relief, passons maintenant à l’élévation des sommets.

### 3.4 Elévation du terrain

Pour mettre en relief notre maillage nous avons choisi d’associer l’intensité en niveau de gris des pixels d’une image, à la hauteur de chaque sommet du maillage. Ce genre d’image est appelé une *heightmap*. Ainsi, un pixel blanc sur l’image indiquera qu’il n’y a pas d’élévation à réaliser, alors qu’un pixel de couleur noir indiquera l’élévation maximale.

Voir la figure [12] page [13] pour avoir un exemple d’utilisation de *heightmap*.

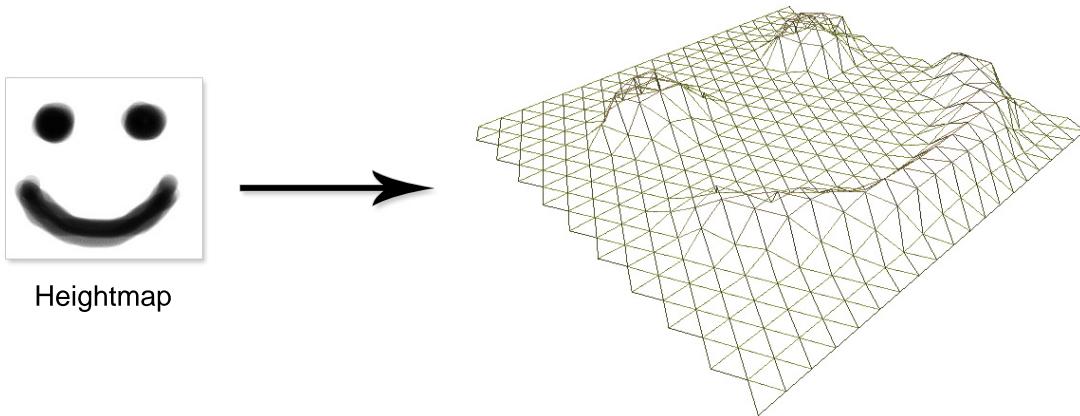


FIG. 12 – Fonctionnement de l’élévation d’un maillage à partir d’une heightmap

Pour effectuer cette opération il est nécessaire de lire pixel à pixel l’image. Pour cela, au lieu de se servir d’une bibliothèque externe, nous avons choisi d’utiliser le format pgm ASCII que nous avons étudié lors du module Image pour le multimédia en Licence 3.

L’avantage de ce format est d’écrire des images en niveau de gris au format ASCII, c’est-à-dire lisible comme une chaîne de caractère.

Exemple de fichier PGM ASCII :

Listing 3 – Exemple de fichier PGM ASCII

```

1 P2
2 3 2
3 255
4 0
5 10
6 67
7 67
8 67
9 0

```

Les trois premières lignes indiquent respectivement le format du fichier (P2 pour PGM ASCII), les dimensions de l’image (ici 3\*2), et le nombre de niveau de gris (255). Ces trois lignes sont uniquement informatives, notre algorithme les ignore, puis pour chaque sommet du maillage, il va convertir le niveau de gris lu dans le fichier en une hauteur. Pour cela, il associe 255 (blanc) à la hauteur 0 et la couleur 0 à la hauteur maximal que nous avons appelé *hauteurMax* grâce à la formule :

$$\text{hauteur} = \text{hauteurMax} - (\text{niveau\_de\_gris}/255) \times \text{hauteurMax}$$

Remarque : Il est nécessaire d'utiliser une image dont les dimensions correspondent aux dimensions du terrain à générer pour avoir un résultat cohérent.

Nous avons désormais un terrain et une caméra qui permet de voler au dessus. Nous aimerais maintenant que la caméra suive le terrain comme le ferait un marcheur.

## 4 Le mode marche

Afin de rendre plus réaliste la balade en forêt nous avons souhaité réaliser un mode dans lequel la caméra suit à hauteur constante le terrain tel un marcheur, nous l'avons appelé : le *mode marche*.

Pour rendre possible cette fonctionnalité il est nécessaire de connaître, pour une position de la caméra donnée, la hauteur du terrain situé juste en dessous. Autrement dit, nous devons trouver le point d'intersection M de la droite D parallèle à la verticale du mode et passant par la position de la caméra, avec le plan P défini par la face située en dessous de la caméra. Ces informations sont rassemblées dans le schéma figure [13] page [14].

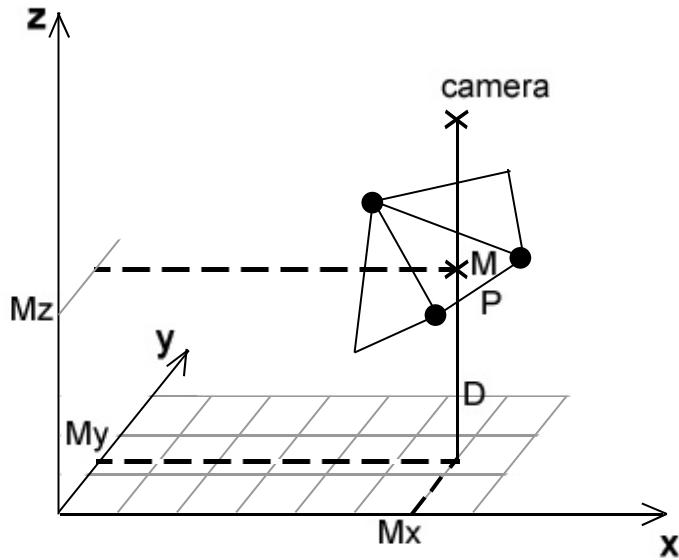


FIG. 13 – Schéma représentant ce qu'on doit déterminer pour trouver le point d'intersection M

Pour mettre en place le mode marcheur nous avons besoin d'effectuer, dans l'ordre, les cinq étapes suivantes :

1. Déterminer la face située en dessous de la caméra.
2. Déterminer l'équation du plan P appartenant à la face trouvée à l'étape précédente.
3. Déterminer l'équation de la droite D verticale au monde et passant par la caméra.

4. Trouver la hauteur du point d'intersection M de la droite D et du plan P.
5. Positionner la caméra à une constante K au dessus du point d'intersection M.

#### 4.1 Détermination des trois sommets de la face située en dessous de la caméra

L'objectif de cette étape consiste à retrouver les trois sommets de la face située sous la caméra. Comme nous connaissons l'algorithme de génération du terrain nous avons décidé de retrouver par un nouvel algorithme ces trois sommets. Pour cela nous travaillerons dans le plan du sol ( $xOy$ ) en projetant le maillage du terrain et la position de la caméra selon la verticale du monde. On déterminera alors les coordonnées X et Y des sommets de la face, puis à l'aide d'un tableau contenant pour un X et un Y la hauteur Z associée on retrouvera la coordonnée Z de chacune des faces.

Afin de faciliter la recherche de la face nous considérerons dans un premier temps que le maillage du terrain est dans la première forme que nous avions montré.

Rappel du premier maillage :

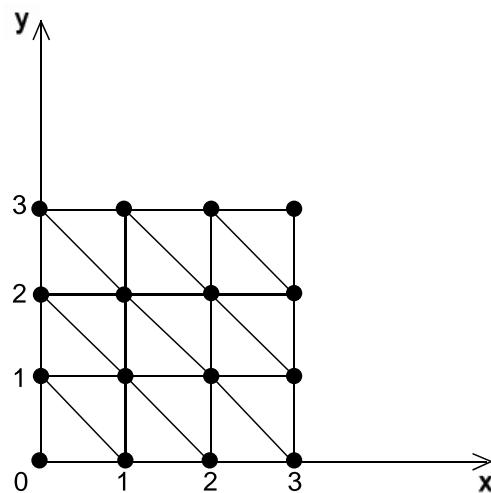


FIG. 14 – Schéma de la première version du maillage du terrain

Dans ce type de maillage, si la caméra est positionné en X et en Y entre 0 et 1 nous avons les données suivantes :

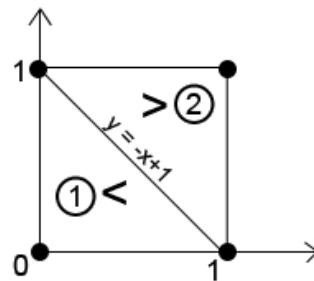


FIG. 15 – Schéma indiquant la face dans laquelle se situe la caméra, en fonction de l'équation de la diagonale

Afin de connaitre la face se situant sous la caméra, il est nécessaire de vérifier la valeur de  $y$  pour  $-x + 1$ , avec  $X$  et  $Y$  les coordonnées de la caméra. Si on a  $Y < -X + 1$  alors la face en dessous de la camera est la face 1 avec les sommets  $(0,0)$ ,  $(1,0)$  et  $(0,1)$  sinon se sont les sommets de la face 2 à savoir  $(1,0)$ ,  $(1,1)$  et  $(0,1)$ .

Cependant la caméra n'a que rarement une position en  $x$  et en  $y$  comprise entre 0 et 1, mais cela n'est pas un problème puisqu'on peut généraliser la propriété ci-dessus à toutes positions de la caméra, en prenant les parties décimales de  $X$  et  $Y$  dans l'équation  $Y < -X + 1$ , et en ajoutant les parties entières aux coordonnées des trois sommets trouvés.

Pour adapter les propriétés précédentes à notre maillage en triangles isocèles nous devons rajouter encore deux propriétés :

1. Les sommets des lignes impaires sont translatés de 0,5 en  $X$ .
2. Les lignes impaires ont des coordonnées pour les faces qui diffèrent des lignes paires.

Avec  $CX$  et  $CY$  les coordonnées en  $X$  et en  $Y$  de la caméra, nous avons alors l'algorithme :

Listing 4 – Pseudo-code de la génération d'un grille tonique.

```

1 EX = Partie entière de CX
2 EY = Partie entière de CY
3 DX = Partie décimal de CX
4 DY = Partie décimal de CY
5
6 Si DY < -DX +1 Faire
7   Si EY est paire Faire
8     retourner les sommets (EX,EY) , (EX+1, EY) et (EX+0,5 , EY +1)
9
10  Sinon Faire
11    retourner les sommets (EX+0,5 , EY) , (EX+1, EY+1) , (EX, EY+1)
12
13  Fin Si
14
15 Sinon Faire
16   Si EY est paire Faire
17     retourner les sommets (EX+1, EY) , (EX+0,5+1 , EY + 1) et (EX
18     +0,5 , EY + 1)
19
20   Sinon Faire
21     retourner les sommets (EX+0,5 , EY) , (EX+1, EY+1) et (EX, EY
22     +1)
23 Fin Si

```

Il existe d'autres méthodes pour récupérer les sommets de la face en dessous de la caméra, celle implémentée ici n'est pas forcément la plus simple, mais elle fonctionne très bien (même lorsque l'utilisateur sort du terrain généré).

## 4.2 Equation du plan de la face à partir des trois sommets

Pour déterminer l'équation du plan de la forme :

$$ax + by + cz + d = 0$$

Nous devons tout d'abord calculer la normale au plan de la face obtenue en réalisant le produit vectoriel de deux vecteurs issus de la face, ce qui nous donnera les paramètres  $a$ ,  $b$  et  $c$ . Il restera alors à résoudre l'équation donnée plus haut pour trouver le paramètre  $d$ .

### 4.2.1 Calcul de la normale du plan de la face

Il nous faut dans un premier temps deux vecteurs du plan, comme nous connaissons trois sommets de ce plan nous pouvons facilement obtenir ces deux vecteurs que nous appellerons par la suite  $\vec{AB}$  et  $\vec{AC}$ , grâce à la formule :

$$\vec{AB} = \begin{pmatrix} Bx - Ax \\ By - Ay \\ Bz - Az \end{pmatrix}$$

Notons  $S1$  ( $S1x, S1y, S1z$ ),  $S2$  ( $S2x, S2y, S2z$ ) et  $S3$  ( $S3x, S3y, S3z$ ) les trois sommets du plan.  
On a alors,

$$\vec{AB} = \begin{pmatrix} ABx \\ ABy \\ ABz \end{pmatrix} = \begin{pmatrix} S2x - S1x \\ S2y - S1y \\ S2z - S1z \end{pmatrix}$$

et

$$\vec{AC} = \begin{pmatrix} ACx \\ ACy \\ ACz \end{pmatrix} = \begin{pmatrix} S3x - S1x \\ S3y - S1y \\ S3z - S1z \end{pmatrix}$$

On peut désormais calculer la normal du plan en faisant le produit vectoriel de  $\vec{AB}$  et  $\vec{AC}$  :

$$\begin{aligned} \vec{n} &= \begin{pmatrix} a \\ b \\ c \end{pmatrix} = AB \otimes AC \\ \vec{n} &= \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} AB_y \times AC_z - AC_y \times AB_z \\ AB_z \times AC_x - AC_z \times AB_x \\ AB_x \times AC_y - AC_x \times AB_y \end{pmatrix} \end{aligned}$$

On a alors l'équation du plan  $P$  :  $ax + by + cz + d = 0$  où  $a$ ,  $b$  et  $c$  sont connus.

Seul le paramètre  $d$  reste à déterminer, pour cela on prend les coordonnées d'un sommet du plan que l'on remplace dans l'équation d'où :

$$d = -a \times S1x - b \times S1y - c \times S1z$$

Tous les paramètres de l'équation du plan sont trouvés, passons à la recherche de l'équation de la droite  $D$  verticale au monde et passant par la caméra.

#### 4.2.2 Equation de la droite D

L'équation d'une droite D est défini par le théorème [8] suivant : Soit un point A de coordonnées  $(a_1, b_1, c_1)$ ,  $\vec{u}$  un vecteur non nul de coordonnées  $(u_1, u_2, u_3)$  et D la droite passant par A et de vecteur directeur  $\vec{u}$ . Un point M de coordonnées  $(M_x, M_y, M_z)$  appartient à la droite D si et seulement s'il existe un réel t tel que :

$$D = \begin{cases} M_x = a_1 + t \times u_1 \\ M_y = b_1 + t \times u_2 \\ M_z = c_1 + t \times u_3 \end{cases} \quad \forall t \in R$$

Nous avons donc besoin de calculer le vecteur directeur  $\vec{u}$ , qui peut être déterminer à partir de deux points issus de la droite D. On désigne alors un premier point A1  $(a_1, b_1, c_1)$  correspondant aux coordonnées de la caméra, et un second point A2  $(a_2, b_2, c_2)$  également issus de cette droite. Sachant que A2 est sur D qui est verticale au monde on sait que  $a_1 = a_2$  et  $b_1 = b_2$  d'où : A2  $(a_2, b_2, c_2) = (a_1, b_1, c_1)$

Le vecteur directeur  $\vec{u}$  peut désormais s'exprimer de la manière suivante :

$$\vec{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} a_2 - a_1 \\ b_2 - b_1 \\ c_2 - c_1 \end{pmatrix} = \begin{pmatrix} a_1 - a_1 \\ b_1 - b_1 \\ c_2 - c_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ c_2 \end{pmatrix}$$

D'où l'équation de la droite D :

$$D = \begin{cases} M_x = a_1 + a_1 + t \times (0) \\ M_y = b_1 + t \times (0) \\ M_z = c_1 + t \times (c_2 - c_1) \end{cases}$$

$$D = \begin{cases} M_x = a_1 \\ M_y = b_1 \\ M_z = c_1 + t \times (c_2 - c_1) \end{cases}$$

Maintenant que nous avons l'équation de la droite et l'équation du plan, calculons l'intersection.

#### 4.2.3 Intersection de la droite D avec le plan P

Pour déterminer le point d'intersection M, il suffit de résoudre le système comprenant l'équation du plan P et de la droite D.

$$\begin{cases} ax + by + cz + d = 0 \\ M_x = a_1 \\ M_y = b_1 \\ M_z = c_1 + t \times (c_2 - c_1) \end{cases}$$

On remplace l'équation de la droite dans le plan P :

$$\begin{cases} a \times (a1) + b \times (b1) + c \times ((c1) + t(c2 - c1)) + d = 0 \\ Mx = a1 \\ My = b1 \\ Mz = c1 + t \times (c2 - c1) \end{cases}$$

Il s'agit ici d'une équation à une inconnue (la variable t) et de degré 1, on a donc :

$$t = \frac{-a \times a1 - b \times b1 - c \times c1 - d}{c \times c2 - c \times c1}$$

Maintenant que nous avons  $t$  il suffit de le remplacer dans l'expression de la hauteur du point d'intersection M soit :

$$Mz = c1 + t \times (c2 - c1)$$

#### 4.2.4 Mise à jour de la hauteur de la caméra

A la hauteur  $Mz$  du point d'intersection trouvé à l'étape précédente, on ajoute à la coordonnée Cz de la caméra, une constante K simulant la taille du marcheur :

$$Cz = Mz + K$$

Le mode marche est désormais opérationnel, nous pouvons donc nous balader sur notre terrain. Essayons maintenant de planter quelques arbres.

## 5 Crédit des arbres et textures

La création des arbres de notre forêt a été réalisée par l'application d'une texture sur deux carrés croisés perpendiculairement dans l'espace. Nous allons voir comment les textures ont été créées puis appliquées sur ces carrés afin de donner naissance à différents arbres et arbustes.

### 5.1 Crédit des textures

#### 5.1.1 Chargeur d'image OpenGL

Pour utiliser les textures en OpenGL nous avons deux possibilités : soit nous utilisons des textures procédurales (codé en C), soit nous utilisons des textures issues de fichiers image tel que les fichiers .jpeg, .png, .gif, .bmp ou .tga. Les arbres étant des objets complexes et donc difficiles à créer de manière procédurale : la deuxième solution a été retenue.

La première étape consiste à charger un fichier image représentant un arbre vu de face afin de le mettre en mémoire. L'étape suivante consistant à appliquer la texture chargée en mémoire sur un objet (plus précisément, sur une facette d'un objet).

Le chargeur d'image que nous avons choisi est le chargeur de fichier targa (fichier .tga) car les pixels de ces images sont codées sur 4 octets, 3 octets représentants chacun des composantes couleurs et le dernier octet représentant la valeur de transparence du pixel (canal alpha). Cette transparence nous permet de plaquer un arbre sur un carré transparent et non pas sur un carré blanc par exemple.

Le chargeur de fichier targa a été récupéré sur le site [3]. Voici un aperçu des principales méthodes de ce chargeur :

*void GetTextureInfo (tga\_header\_t \*, gl\_texture\_t \*)* : rassemble les informations sur le fichier targa notamment celles contenues dans son entête (largeur, hauteur, profondeur de l'image, etc.).

*void ReadTGAXXbitsYYY (FILE \*fp, GLubyte \*, gl\_texture\_t \*)* : les méthodes permettant de lire le fichier selon son type, XX correspond à la profondeur des pixels du fichier (compris entre 8 et 32 bits) et YYY spécifie si le fichier lu a été compressé avec la méthode RLE ou non.

*GLuint loadTGATexture (const char \*)* : méthode prenant en paramètre le chemin d'accès au fichier à lire et ce chargeant d'appeler la méthode de lecture adéquat selon la profondeur et l'encodage de l'image.

Maintenant que nous avons notre chargeur de fichier il va falloir créer les textures pour mettre en place la forêt.

### 5.1.2 Crédit des textures au format targa

Pour trouver des textures d'arbres nous avons parcourut internet et s'est le site [4] qui s'est avéré le plus utile. Cependant le défaut de ce site est qu'il fait payer les textures détournées, mais qu'il donne librement les images originales où ont été prisent les arbres. Nous avons alors effectué le détourage nous même avec le logiciel The Gimp. Pour le format d'enregistrement, nous avec sauvegardé les textures targa 32bits et sans compression RLE. Voici, figure [16] page [20], un aperçu des textures réalisées.



FIG. 16 – Aperçus des textures TGA des arbres et arbustes de la forêt.

Une fois ces textures réalisées, il est nécessaire de les appliquer sur des faces en OpenGL, nous les appliquerons en l'occurrence sur deux faces croisées perpendiculairement. Nous pouvons voir, figure [17] page [21], un exemple d'arbre ainsi créé.



FIG. 17 – Textures d'arbres appliquées sur deux faces croisées perpendiculairement.

Ce procédé bien connus en 3D permet de donner l'illusion d'un arbre représenté dans les trois dimensions.

### 5.1.3 Directives générales pour l'application des textures

OpenGL propose plusieurs directives permettant de spécifier le comportement des textures lors du mappage de celle-ci. Nous allons voir quelles ont été ces directives (qui sont à ajoutées dans la fonction *main*).

Listing 5 – Extrait

```

1 //La première directives a pour but d'activer la transparence dans les
2 //textures
3 glEnable(GL_ALPHA_TEST);
4 //La seconde permet de spécifier un seuil pour lequel le canal alpha de la
5 //texture
6 //n'est pas pris en compte ce qui permet d'affiner les parasites comme
7 //des lignes blanches sur les bords des textures
8 glAlphaFunc(GL_GREATER, 0.5);
9
10 // Spécifie la valeur du pixel lorsqu'il est transparent (dépend du format de
11 //fichier , soit alpha=1, soit alpha=0)
12 glBindFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
13 //On spécifie ensuite le nombre de texture et le tableau
14 //contenant les identifiants de texture
15 glGenTextures (9, IdTex);
```

Apres la déclaration des directives vient le chargement des textures avec la fonction *loadTGA-Texture* qui prend en paramètre le chemin d'accès du fichier TGA à lire et l'identifiant de texture qui lui sera associé.

## 5.2 Génération des arbres et mappage du sol

Pour générer les arbres sur toute la surface du terrain nous avons plusieurs variables à mettre en place, en l'occurrence, un nombre total d'arbre à créer, un tableau de position aléatoire sur X, et un sur Y ainsi qu'un tableau choisissant le type d'arbres à "planter".

Le mappage du terrain, quant à lui, a été réalisé à partir des textures figure [18] page [22]. Ces trois textures changent avec la hauteur du terrain, plus celui-ci est haut, moins il y a d'herbe :



FIG. 18 – Les différentes textures du terrain.

Le mappage du sol est réalisé lors de la création des triangles, en fonction de sa hauteur, et où l'on spécifie la texture adéquat avec :

```
glBindTexture (GL_TEXTURE_2D, IdTex[<numéro texture sol>])
```

## 5.3 Placement des arbres sur le terrain

Afin de placer les arbres sur le terrain nous avons utilisé trois tableaux définissant respectivement la position en X, en Y et la texture (parmi 8) de chaque arbre.

Chacun des tableaux comporte  $L \times l$  cases, donc il peut y avoir au maximum  $L \times l$  arbres sur le terrain (un arbre par sommet du terrain).

Puisqu'à l'aide d'une méthode appelé *initArbres()* nous définissons une valeur aléatoire pour chacun de ces trois paramètres et cela pour un nombre d'arbres données (variable global appelée *nbArbres*).

Il suffit alors de parcourir les trois tableaux pour connaître, pour chaque arbre à afficher, sa position et sa texture.

## 5.4 Fonctionnalités supplémentaires

Nous avons également défini plusieurs autres fonctionnalités permettant l'augmentation du réalisme et de l'interaction avec l'utilisateur comme des nuages qui bougent dans le ciel grâce à la fonction d'animation *idle()*. Un menu contextuel apparaissant lors d'un clic bouton droit permet notamment de passer du mode vol libre au mode marcheur et de charger différents terrains (choix parmi un grand nombre de terrains correspondant à des *heightmap* différentes). Enfin il est possible, à l'aide des touches du clavier, de régler en temps réel la hauteur du terrain, la densité du brouillard, la taille du marcheur et le nombre d'arbres.

## 6 Conclusion

Tout au long du projet plusieurs problèmes se sont présentés à nous, mais nous avons réussi à tous les surmonter. Nous nous sommes, par exemple, surpris de voir la facilité que nous avons eu à résoudre les problèmes de géométrie spatiale, nous devons certainement cela à tous les modules de mathématiques suivis dans notre parcours, et notamment le module de Synthèse d'image. Cependant un des problèmes qui nous a pris le plus de temps à résoudre concernait la transparence des textures. Les fichiers targa n'étant pas courant, le traitement de tel fichier dans The Gimp n'a pas été totalement aisés et l'incorporation en OpenGL laissait à désirer, avec notamment des effets de pixelisation tout au tour des textures résolus avec la fonction *glAlphaFunc*.

Le projet d'une balade en forêt s'est avéré très intéressant sur plusieurs points. En effet, nous avons pu réaliser ici un véritable terrain en trois dimensions ainsi que le texturage de celui-ci et la génération d'arbres afin de créer, au final, un monde virtuel tel que nous pouvons les voir dans les jeux vidéos d'il y a cinq - dix années passées. Grâce à la création d'une caméra en mode "marcheur" il ne nous resterait maintenant plus qu'à gérer un système de tire avec une arme pour prétendre avoir conçu un moteur basique de FPS (first person shooter). Enfin, ce projet nous a réellement enthousiasmé et la mise en place de toutes les fonctionnalités pour la création de cette forêt à souvent été un challenge dont nous étions heureux de relever.



FIG. 19 – Screenshot de la forêt.

## 7 Bibliographie

Ce rapport a été réalisé à partir de l'ouvrage [1], des sites Internet [2, 3, 4, 5, 6, 8, 9, 10] et de nos cours de synthèse d'images [7].

## Références

- [1] Jackie Neider, Mason Woo, Tom Davis, Dave Shreiner, Véronique Campillo. Open GL 2.0 : Guide officiel . ISBN : 978-2744020865
- [2] Mandragor section OpenGL. [http://docs.mandragor.org/files/Misc/GLFM/lm32/OpenGL\\_Texture\\_end.html](http://docs.mandragor.org/files/Misc/GLFM/lm32/OpenGL_Texture_end.html)
- [3] Developpez.com section 2D-3D. <http://www.developpez.com/>
- [4] Got3D. <http://www.got3d.com>
- [5] Cours d'openGL de l'académie de Reims. <http://helios.univ-reims.fr/Labos/LERI/membre/bittar/03OpenGL/index.html>
- [6] Evasion. <http://www-evasion.imag.fr/Membres/Antoine.Bouthors/>
- [7] Cours de synthèse d'images de l'université. <http://ufrsciencestech.u-bourgogne.fr/Master1/mil-tcl>
- [8] Site de vulgarisation des mathématiques. <http://homeomath.imingo.net/index3.htm>
- [9] Portail comportant un lexique de mathématique. <http://euler.ac-versailles.fr/>
- [10] Tutorial de mise en place d'une caméra Freefly en SDL. <http://www.siteduzero.com/tuto-3-9448-1-controle-avance-de-la-camera-partie-2-2.html>