

Projet IG : Doom-like

Loïc Loriné, Julien Nauroy

12 mai 2005

Table des matières

1	Présentation	3
2	Architecture du projet	3
3	Modèles Géométriques	5
3.1	Terrains	5
3.2	Construction de l'avatar	6
3.2.1	Présentation	6
3.2.2	Controle	6
3.2.3	Objets 3D	10
3.2.4	Animation	10
4	Extrait de code commenté	11
5	Notice d'utilisation	12
6	Approfondissement OpenGL	13

1 Présentation

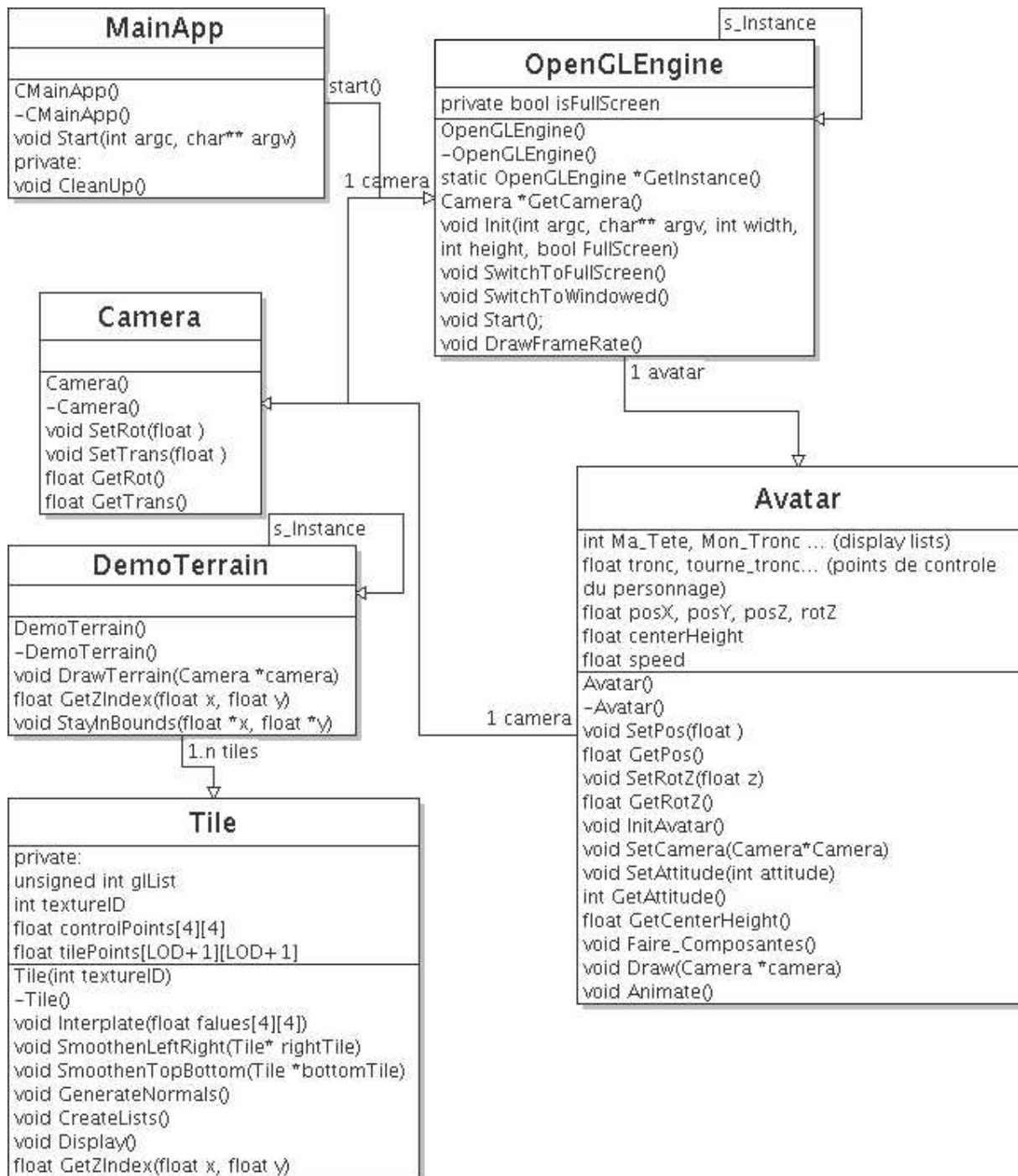
Ce projet est basé sur le sujet de Doom-Like, mais pour lequel nous avons voulu mettre l'accent sur les lignes de vue, les hauteurs et avantages qu'ils procurent, ainsi que le jeu sur les parties visibles et cachées. Nous sommes partis du principe des jeux 2D en tour par tour qui sont très orientés sur des règles de réussite basées sur la difficulté d'effectuer une action et les compétences du personnage pour réaliser cette action, et avons tenté de faire évoluer certaines de ces règles qui peuvent être calculées précisément informatiquement : par exemple connaître le taux de couverture d'un personnage par rapport à un autre en tenant compte du décors, ou bien encore évaluer si un personnage tirant sur un autre le touche, non pas en lançant virtuellement des dés mais en simulant la trajectoire d'une balle tirée avec un certain aléa par rapport à la trajectoire voulue.

Un autre objectif était de proposer des environnements modulables, que l'on pourrait facilement changer pour donner de la diversité au jeu. Ainsi nous nous devions de proposer un moyen simple de changer le relief d'un terrain, ses textures ou encore les objets qui en font partie.

Enfin, nous avons voulu mettre un personnage sur ce terrain, et gérer son déplacement par rapport au terrain, et les mouvements de son corps.

Les sections suivantes présentent les résultats que nous avons obtenus lors de la réalisation de ce projet.

2 Architecture du projet



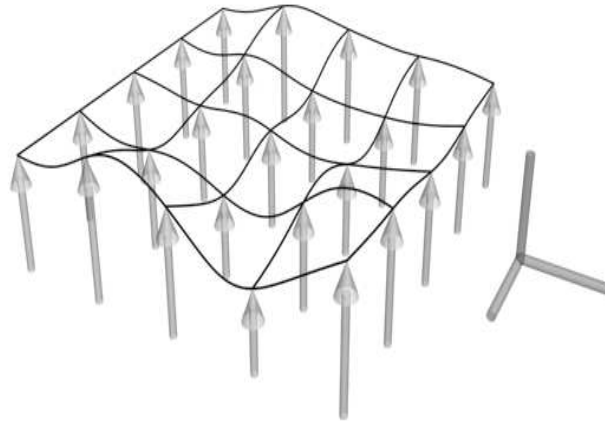
Ce diagramme UML ne contient, par choix, ni toutes les classes, ni toutes les propriétés des classes montrées. Le diagramme se concentre sur la partie intéressante du projet, et occulte certaines parties

comme les exceptions (peu utilisées), les classes dérivées (DemoTerrain dérive en fait de BaseTerrain, classe faite dans l'optique de proposer plusieurs terrain, et DemoTerrain classe créée à l'origine uniquement pour nos tests), et autres propriétés inutiles dans un tel diagramme.

3 Modèles Géométriques

3.1 Terrains

Les terrains sont constitués d'un ensemble de tuiles (Tiles en anglais), chacune étant en fait un patch de Bézier. L'ensemble ressemble donc au schéma suivant :



Les tiles ne sont pas générés directement par l'algorithme de Bézier mais par un algorithme similaire : au lieu d'avoir 4 points fixes

et 12 points intérieurs à ce carré, constituant les points de contrôle, notre algorithme utilise des points de contrôle extérieurs au tile.

Le tile et ses points de contrôle ressemblent donc à ceci :

c	c	c	c
c	p	p	c
c	p	p	c
c	c	c	c

Les c représentent ici les points de contrôle et les p représentent les points du tile. L'algorithme conserve les propriétés des surfaces de Bézier, en particulier d'être de classe G1 (continuité aux bords lorsque les points de contrôle correspondent).

Le patch continu est ensuite discrétisé en un certain nombre de parties prédéterminé, choisi suivant le niveau de détail désiré. Dans notre cas nous nous sommes limité à découper la tuile en 4x4 ou 8x8 sections, ce qui est largement suffisant pour avoir un très bon niveau de détail. Dans toutes les captures écran le niveau de détail est de 4x4. Chaque sous-partie carrée du tile est ensuite subdivisée simplement en deux triangles. On a donc seulement 32 triangles par tile !

3.2 Construction de l'avatar

3.2.1 Présentation

Pour notre avatar nous voulions mettre l'accent sur l'animation. Nous voulions un avatar capable de prendre toutes les postures possibles, aussi nous avons définis un ensemble minimal de points de "mouvement". Ces points sont les parties du corps sur lesquels notre corps s'articule : coudes, genoux , ect...

3.2.2 Controle

Voici la totalité des points de contrôle définis pour le mouvement de l'avatar, ces points de contrôle représentent des angles en degrés :

1. partie superieure

- (a) tronc : $25 > \text{tronc} > -15$ selon Ox, permet de faire se pencher le sujet (sert pour les positions penchés que prennent les militaires pour éviter les balles).(non utilisé actuellement)
- (b) tourne_tronc : $-35 > \text{tourne_tronc} > 35$ selon Oz, la rotation du tronc sert pour la marche et la course, c'est la rotation naturelle du corps due à l'inertie des bras.
- (c) cou : $45 > \text{cou} > -5$ selon Ox permet de faire pencher la tete vers l'avant.(non utilisè actuellement)
- (d) tête : définie par trois composantes, malheureusement la nature sphérique de la tete actuelle annule totalement l'intérêt de ces possibilités de mouvement.
 - i. inclin_tete_avant :
aquiecer $80 > \text{inclin_tete_avant} > -45$ selon Ox.
 - ii. inclin_tete_cote :
hocher la tête $45 > \text{inclin_tete_cote} > -45$ selon Oy.
 - iii. tourne_tete :
nier $80 > \text{tourne_tete} > -80$ selon Oz.
- (e) bras (symétriques) : Les bras n'ont pas été animés pour l'instant, ils devaient porter une arme qui n'as pas été implémentée, et donc ne pas bouger par rapport au tronc. Ils ont simplement été orientés par rapport au tronc pour essayer de figurer le port de l'arme.
 - i. avance_bras_droit :
 $180 > \text{avance_bras_droit}(\text{ ou gauche }) > -45$ selon Ox, permet de faire avancer le bras devant le sujet (parallèlement au tronc).
 - ii. leve_bras_droit :
 $180 > \text{leve_bras_droit}(\text{ ou gauche }) > 0$ selon Oy permet de lever le bras en l'air (perpendiculairement au tronc).

iii. tourne_bras_droit :

$-60 < \text{tourne_bras_droit} \text{ (ou gauche) } < 90$ selon Oz
permet de tourner le bras vers l'intérieur du corps.

iv. avance_avbras_droit :

$160 > \text{avance_avbras_droit} \text{ (ou gauche) } > 0$ selon Ox
même mouvement que celui du bras

v. poignet_droit :

$60 > \text{poignet_droit} \text{ (ou gauche) } > -80$ selon Oy

(f) partie inferieure

i. jambes (symetriques) : les mouvements sont les mêmes
que ceux des bras.

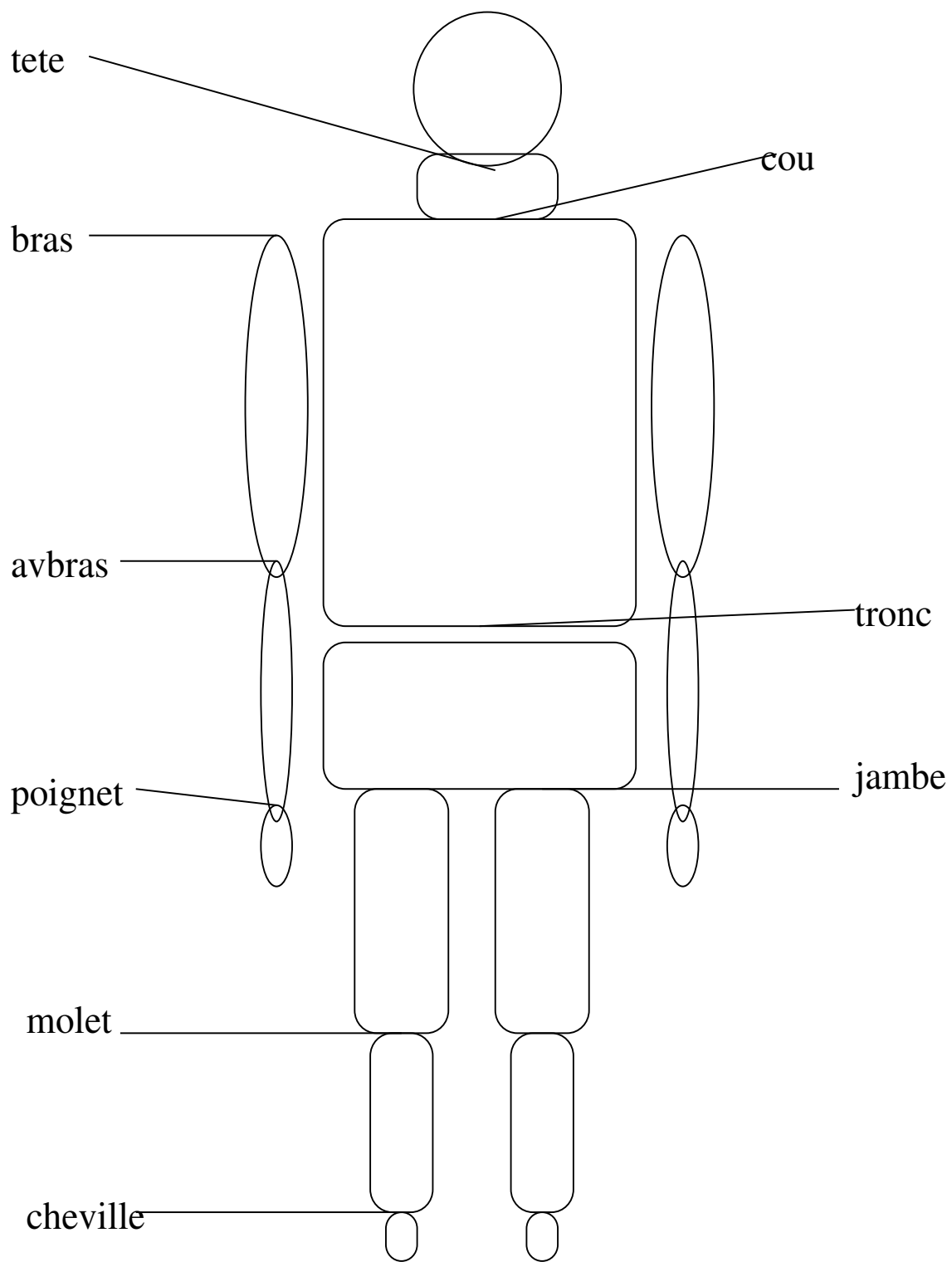
A. avance_jambe_droit : $110 > \text{avance_jambe_droit} \text{ (ou gauche) } > -35$ selon Ox

B. leve_jambe_droit : $90 > \text{leve_jambe_droit} \text{ (ou gauche) } > 0$ selon Oy

C. avance_mollet_droit : $160 > \text{avance_mollet_droit} \text{ (ou gauche) } > 0$ selon Ox

D. cheville_droit : $90 > \text{poignet_droit} \text{ (ou gauche) } > 0$
selon Ox

Les emplacements des points de contrôle sont rapelés sur la figure
suivante :



Les jambes et bras gauches sont définis de la même manière.

Toutes les différentes parties du corps sont ensuite définies dans des display lists, comme nous l'avons vu dans le TD5. Pour que les membres soient le plus jolis possibles, nous avons utilisé pour les créer une rotation de courbes de Bézier telle qu'elle est mise en œuvre dans le TD8 (surfaces de révolution).

3.2.3 Objets 3D

Dessiner des membres humains est très difficile, en effet modéliser correctement une main peut être très long. De la même manière, autant modéliser l'approximation d'un bras est simple puisque un bras est circulaire, autant un tronc ne peut pas être facilement modélisé par une surface de révolution. Si notre sujet avait uniquement consisté en l'animation d'un avatar nous serions certainement allés plus loin, mais nous avons décidé de simplifier le problème.

Les membres supérieurs et inférieurs ainsi que le cou ont donc été modélisés par une révolution de surfaces de Bézier, alors que les autres parties du corps sont des cylindres et la tête une sphère.

3.2.4 Animation

Nous avons une variable d'état, qui à chaque instant nous dit si l'avatar marche, recule, cours, est accroupi ou immobile. Selon son état l'avatar s'anime différemment. L'animation est gérée de la même manière que dans le TD5, la fonction `gl_timer` appelle `Animate` cent fois par seconde. À chaque appel, une variable d'animation est modifiée (comprise entre 0 et 1), et chaque point de contrôle est modifié selon sa valeur. Certaines animations auraient pu être rajoutées, nous voulions pouvoir faire s'accroupir l'avatar, et après réflexion il aurait été plus agréable qu'il bouge en tournant.

4 Extrait de code commenté

Pour cette section, nous avons choisi de présenter la création des tiles du terrain. Cette création est en fait assez directe, mais présente quelques subtilités.

```
// chargement du fichier jpeg qui va servir a creer la heightmap.
// notez que seule la composante rouge est utilisee
tImageJPG *image = LoadJPG("GameData/Maps/Demomap/map.jpg");
if (image->sizeY != MAPSIZE+3 || image->sizeX != MAPSIZE+3)
    throw new BadTerrainLoadedException(__FILE__, __LINE__, "Map size d

// chargement de la carte des textures correspondant aux tiles. Actue
// d'herbe mais avons donne la possibilite d'en utiliser d'autres
tImageJPG *textures = LoadJPG("GameData/Maps/Demomap/textures.jpg");
if (textures->sizeY != MAPSIZE || textures->sizeX != MAPSIZE)
    throw new BadTerrainLoadedException(__FILE__, __LINE__, "Texture ma

// Initialisation separee de chaque tile a partir d'une matrice 4x4 d
for(int j = 1; j < MAPSIZE+1; j++)
    for(int i = 1; i < MAPSIZE+1; i++)
    {
tiles[j-1][i-1] = new Tile(textures->data[(j-1)*image->rowSpan+3*(i-1)]
float neighborhood[4][4] = (... construction du voisinage ici...);
tiles[j-1][i-1]->Interpolate(neighborhood);
    }

// demande calcul des normales
for(int j = 0; j < MAPSIZE; j++)
    for(int i = 0; i < MAPSIZE; i++)
        tiles[j][i]->GenerateNormals();
// Adoucissement des normales aux bornes des tiles
// deux tiles jointifs ont des points identiques mais a normales diff
```

```

// les lignes suivantes vont demander a faire la moyenne des normales
for(int j = 0; j < MAPSIZE-1; j++)
    for(int i = 0; i < MAPSIZE; i++)
        tiles[j][i]->SmootherTopBottom(tiles[j+1][i]);
for(int i = 0; i < MAPSIZE-1; i++)
    for(int j = 0; j < MAPSIZE; j++)
        tiles[j][i]->SmootherLeftRight(tiles[j][i+1]);

// sauvegarde des listes d'affichage dans la carte video
// ce peut etre des display list classiques, ou si la carte video le
// des interleaved array stockees dans la carte video (extension ARB
for(int j = 0; j < MAPSIZE; j++)
    for(int i = 0; i < MAPSIZE; i++)
        tiles[j][i]->CreateLists();

// Ajout des objets statiques sur la carte
BaseItem::GetItemList()->push_back((BaseItem*)(new PalmTree(55.0f, 10
    GetZIndex(55.0f, 10.0f))) );
// (...etc...)
}

```

5 Notice d'utilisation

Le fonctionnement du programme est le suivant :

1. avancer la caméra : touche haut du clavier
2. reculer la caméra : touche bas du clavier
3. tourner la caméra à gauche : touche gauche du clavier (non recommandé)
4. tourner la caméra à droite : touche droite du clavier (non recommandé)
5. avancer l'avatar : touche w sur clavier qwerty ou z sur clavier azerty

6. reculer l'avatar : touche s du clavier
7. tourner l'avatar à gauche : touche a sur clavier qwerty ou q sur clavier azerty
8. tourner l'avatar à droite : touche d du clavier
9. faire courrir l'avatar : touche r du clavier
10. tourner avec la souris : maintenez clic gauche ou droit appuyé puis bougez la souris (recommandé)
11. afficher les cotés des triangles au lieu de les remplir (mode wi-reframe) : espace
12. afficher la position courante de la caméra : p
13. passer en mode plein ecran (fullscreen) : f
14. prendre / perdre de l'altitude (caméra) : molette de la souris

6 Approfondissement OpenGL

Dans ce projet, nous avons essayé de mettre en oeuvre certaines techniques plus ou moins avancées d'OpenGL :

1. Le Vertex Buffer Object, qui est la technique consistant à envoyer dans la carte vidéo les tableaux d'affichage (sommets, normales, coordonnees de texture), au lieu de soit les garder en mémoire, soit les envoyer en tant que display list. L'intérêt au niveau performance n'est pas aussi flagrant que ce qui est donné sur les sites traitant de la technique (l'écart entre vertex array dans la mémoire centrale et la mémoire vidéo est flagrant, mais pas celui entre les display list et le VBO), mais il nous a permis d'expérimenter les extensions préconisées par l'ARB (Architecture Review Board), et de rendre notre programme compatible avec les cartes vidéo supportant ces extensions ou non.
2. Le mip-mapping, permettant à une texture d'être toujours bien représentée à l'écran. Lorsque celui-ci est désactivé, l'un des deux problèmes suivants apparaît :

- (a) si la texture est trop petite, les objets les plus proches de la caméra apparaîtront très flous
 - (b) si la texture est trop grande, les objets loin de la caméra (souvent même ceux près de la camera) auront une texture "scintillante" : suivant les variations de la camera, la coordonnée de la texture retenue pour colorier un pixel précis d'un triangle peut beaucoup varier, entraînant un scintillement.
3. Le culling, permettant de cacher les triangles en face inverse. Il a pour cela fallu faire très attention au sens des triangles que nous dessinions, et faire en sorte que les objets 2D (par exemple feuilles des palmiers) soient double face.
 4. la détection de parties cachées de la scène, afin de ne pas demander son affichage pour rien. Pour cette partie nous avons utilisé un algorithme très simple qui cache en moyenne 50