

UNIVERSITE DE TECHNOLOGIE DE BELFORT-MONTBELIARD

AUBRY Baptiste  
DESPoulain Thibaut  
DIDIERjean Lauranne  
MAURANYAPIN Adrien

# Génération et rendu 3D de forêt en OpenGL

---

IN55

Printemps 2012 - Département Informatique



# Sommaire

Introduction .....	3
1. Présentation de l'application.....	4
1.1. Interface .....	4
1.2. Technologies et outils utilisés.....	5
2. Implémentation.....	6
2.1. Moteur 3D .....	6
Librairie mathématique.....	6
Scenegraph et structures de donnée .....	6
Chargement de fichier .obj.....	8
2.2. Génération de forêt.....	9
Génération de terrain (Height Mapping) .....	9
Gestion d'ambiance sonore.....	9
Création de modèles 3D de végétation.....	9
Création de la scène et génération d'instances aléatoires d'objets 3D .....	10
Génération de chemin.....	11
2.3. Rendu.....	12
Pipeline de rendu .....	12
Illumination et textures.....	13
Correction gamma.....	13
Ombres .....	14
Lissage PCF des ombres.....	15
Ombres en cascade .....	16
Skybox et sky mapping .....	18
2.4. Post-traitement .....	19
FXAA (Fast Approximation Anti Aliasing).....	19
Downsampling.....	20
Bloom HDRA (High Dynamic Range Approximation).....	21
Godrays (Volumetric Light Scattering Approximation) .....	23
Lens flare (parasite et dispersion de lumière).....	25
Flou d'objectif.....	27
3. Performances .....	28
3.1. Notes .....	28
Conclusion.....	30
Continuation .....	30

## **Introduction**

Le but de notre projet, réalisé en trois mois, est de générer puis d'afficher en 3 dimensions une forêt en OpenGL avec C++ et Qt.

Nous avons opté pour un rendu de type réaliste, basé sur un certain nombre d'effets plus ou moins avancés, notamment :

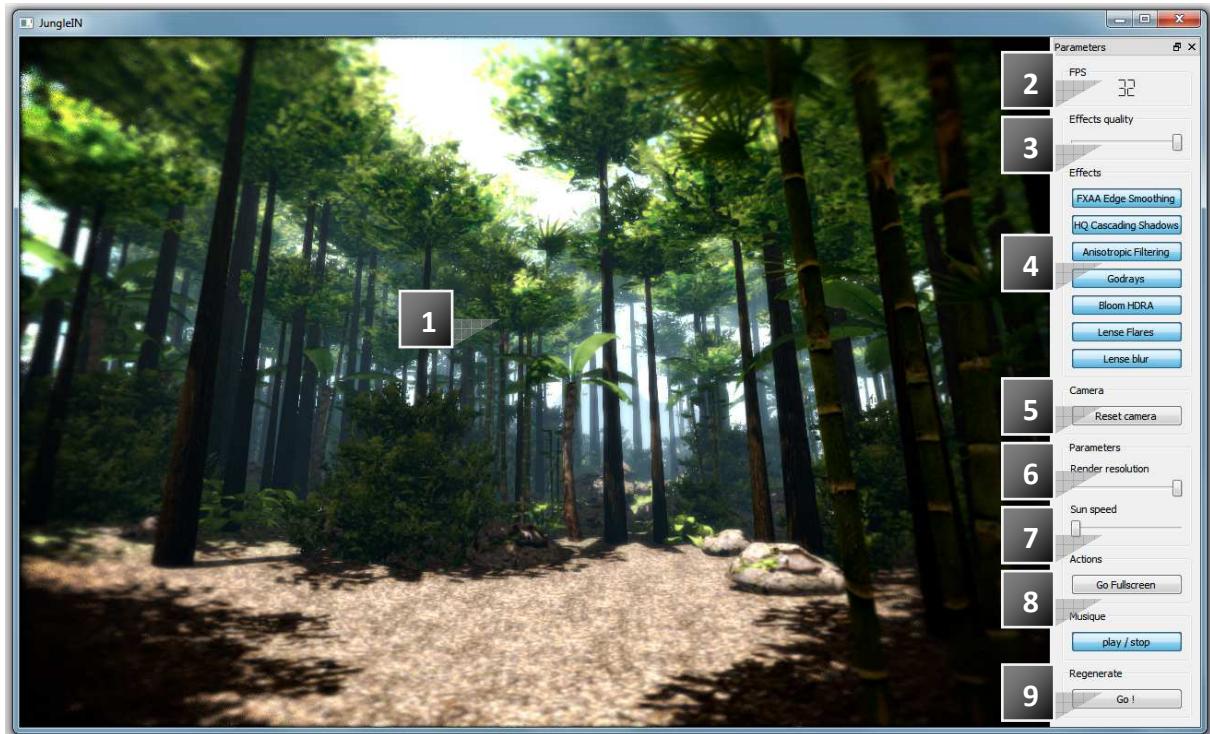
- Du texturing.
- De l'illumination au fragment shader.
- De l'anti-aliasing en post-processing (FXAA).
- Des ombres portées en cascade filtrées via PCF.
- Du filtrage anisotropique.
- Du mip-mapping.
- Des godrays.
- Du bloom.
- De la simulation d'objectif.
- De la correction Gamma.

L'ensemble de ces effets sera discuté plus en détails dans la partie "Rendu" et "Post-processing" un peu plus loin dans ce rapport.

# 1. Présentation de l'application

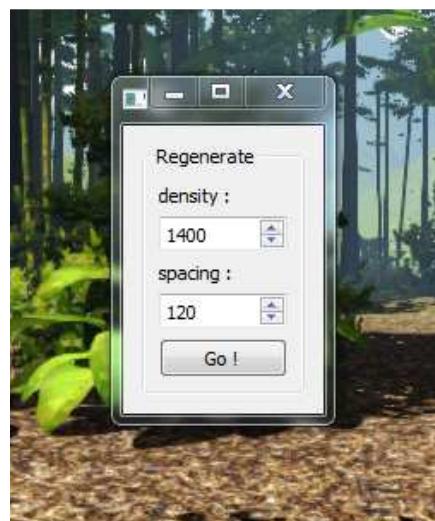
## 1.1. Interface

Le contrôle de la caméra se fait à la souris et au clavier avec les touches directionnelles.



1. La vue 3D principale.
2. Compteur d'images par secondes.
3. Slider influant sur la taille des buffers de downsampling (voir partie "Downsampling"). Ce slider influe sur la qualité globale des effets.
4. Panel d'effets permettant d'activer et de désactiver indépendamment les différents effets.
5. Bouton replaçant la camera à son origine.
6. Slider de changement de résolution de rendu à la volée. Ce slider permet de modifier en temps réel la résolution des buffers de rendu principaux tout en conservant la résolution de la fenêtre de rendu. Cela permet notamment de rendre la scène en résolution moindre afin de gagner en performances sur des machines peu puissantes.
7. Slider contrôlant la vitesse du passage du temps (et donc la vitesse du soleil).
8. Boutons permettant de passer en plein écran et d'arrêter/reprendre la musique d'ambiance.

9. Bouton affichant la fenêtre de paramétrage de la génération de la forêt.
  - a. *Density* contrôle le nombre d'instances d'objets (arbres, végétation, rochers)
  - b. *Spacing* contrôle la taille de la surface occupée par la forêt.



## 1.2. Technologies et outils utilisés

Pour réaliser ce projet, nous avons utilisé les librairies suivantes :

- Qt 4.8
- Glew 1.5.4
- OpenGL 4.0
- GLSL 4.0

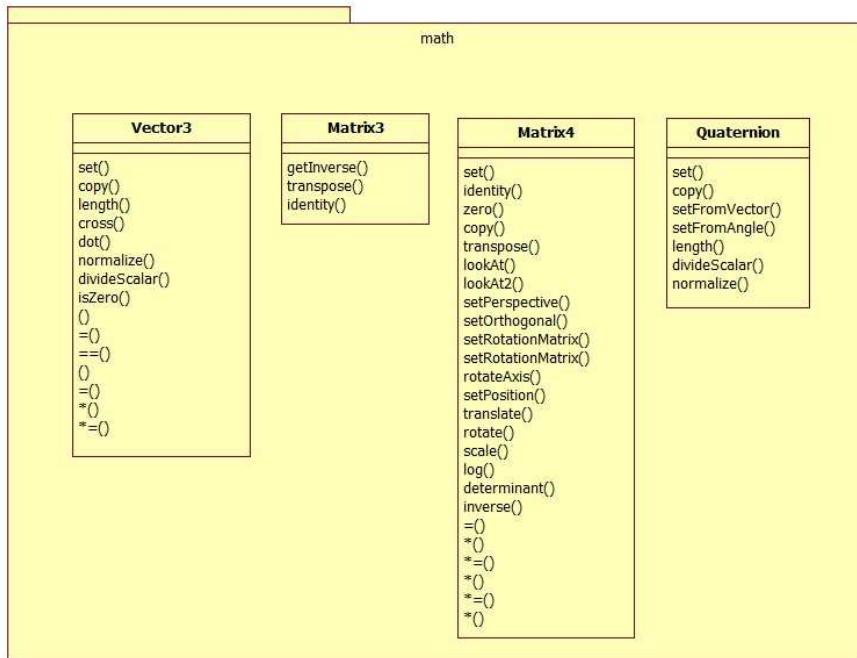
Nous avons utilisé les logiciels suivants :

- QtCreator
- Sublime-Text
- Tortoise SVN
- AMD GPU ShaderAnalyser
- 3ds Max
- Photoshop
- StarUML

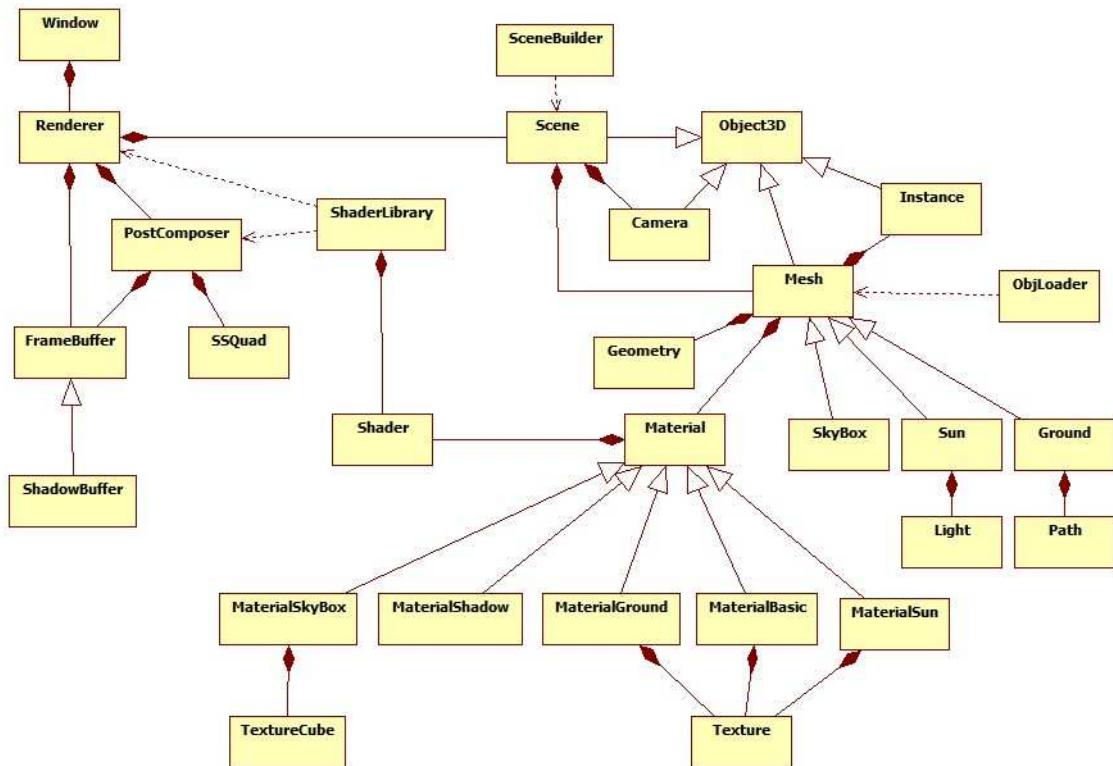
## 2. Implémentation

### 2.1. Moteur 3D

#### Librairie mathématique



#### Scenegraph et structures de donnée



Object3D
matrix: Matrix4 rotation: Vector3 position: Vector3 scale: Vector3 needUpdate: bool visible: bool  setRotation() setPosition() setScale() updateMatrix() rotate() rotateX() rotateY() rotateZ() translate() translateX() translateY() translateZ() translateRel() translateRelX() translateRelY() translateRelZ() getMatrix() getPosition()

**Object3D** : c'est notre classe de base pour la représentation des objets de notre monde.

On voit ici l'utilité de nos classes mathématiques. En effet on utilise une matrice 4\*4 pour la matrice modèle de notre objet. On utilise aussi des vecteurs à 3 composantes pour représenter la rotation, la position et la mise à l'échelle sur les 3 axes. La classe possède toutes les méthodes permettant de mettre à jour les paramètres qui viennent d'être cités.

**Mesh** : cette classe nous permet de représenter un maillage. Elle hérite évidemment de tous les attributs et méthodes de l'Object3d. Elle intègre en plus de cela une Geometry et un Material. On y trouve aussi les méthodes d'initialisation des VertexBufferObject et de dessin du maillage.

**Geometry** : Cette classe permet essentiellement de stocker un tableau de vertices, un tableau d'indice, un de coordonnées de textures ainsi qu'un tableau de normales.

**Material** : Un material contient un shader et des paramètres (qui peuvent être des textures, couleurs, intensités lumineuses selon le type de material).

**Light** : Cette classe nous permet de représenter une lumière et contient la matrice de vue du point de vue de la lumière qui nous est utile dans le calcul des ombres (cf. paragraphe ombres).

**Camera** : c'est une classe de caméra à la première personne. Elle contient en plus de sa matrice modèle une matrice de projection pour voir la scène du point de vue de la caméra.

Une méthode de mise à jour<sup>1</sup> permet à partir des événements souris/clavier de mettre à jour la matrice modèle de la caméra.

---

<sup>1</sup> <http://www.lloydgoodall.com/tutorials/first-person-camera-control-with-lwjgl/>

## Chargement de fichier .obj

```
...
v -0.5075 4.2105 0.2094
v 2.5525 3.1523 -0.5333
v -0.9381 3.5581 -2.0822
v 5.7135 1.6989 -0.4622
v 3.3769 2.8764 -3.3269
v -0.8277 2.3857 -4.1446
v -4.7515 4.6560 -0.3554
v -3.5332 2.9260 -5.0265
v -6.2962 2.4949 -3.2370
v -3.0680 4.6560 1.4735
# 10 vertices

vn 0.1661 0.9493 -0.2671
vn 0.3762 0.9263 -0.0217
vn 0.1507 0.9289 -0.3383
vn 0.7611 0.6466 -0.0512
vn 0.4183 0.7541 -0.5063
vn 0.2237 0.7421 -0.6318
vn -0.3092 0.9487 -0.0656
vn -0.1657 0.6171 -0.7692
vn -0.6648 0.4667 -0.5832
vn -0.1342 0.9414 0.3095
# 10 vertex normals

vt 0.4474 0.5510 0.0000
vt 0.2861 0.6828 0.0000
vt 0.5299 0.6953 0.0000
vt 0.0985 0.7500 0.0000
vt 0.3074 0.9109 0.0000
vt 0.5749 0.8298 0.0000
vt 0.7109 0.4674 0.0000
vt 0.5299 0.2575 0.0000
vt 0.5749 0.4440 0.0000
vt 0.7560 0.3753 0.0000
vt 0.7560 0.8201 0.0000
vt 0.8737 0.5921 0.0000
vt 0.5663 0.3848 0.0000
vt 0.8099 0.2627 0.0000
vt 0.6805 0.0459 0.0000
vt 0.3389 0.4370 0.0000
vt 0.4267 0.1206 0.0000
vt 0.2517 0.2781 0.0000
# 18 texture coords

g Example01
f 1/1/1 2/2/2 3/3/3
f 2/2/2 4/4/4 5/5/5
f 2/2/2 5/5/5 3/3/3
f 3/3/3 5/5/5 6/6/6
f 1/1/1 3/3/3 7/7/7
f 3/8/3 6/9/6 8/10/8
f 3/3/3 8/11/8 7/7/7
f 7/7/7 8/11/8 9/12/9
f 1/1/1 7/7/7 10/13/10
f 7/7/7 9/12/9 11/14/11
f 7/7/7 11/14/11 10/13/10
f 10/13/10 8/14/8 10/15/10
# 12faces
```

La classe statique OBJLoader nous permet à partir d'un fichier .obj (obtenu par exemple à partir d'un modèle 3D créé sous 3dsMax) de recréer un maillage défini comme dans notre structure de données.

Voici un exemple (fictif) d'un fichier .obj. On remarque qu'il est composé de différents blocs représentant chacun des données différentes. On peut alors lire le fichier ligne par ligne de cette manière<sup>2</sup> :

- Une ligne commençant par « v » correspond à un vertex. Les 3 nombres suivants représentent donc les coordonnées du vertex.
- Une ligne débutant par « vn » correspond à une normale de vertex. Les nombres suivants sont ses coordonnées.
- Une ligne commençant par « vt » correspond à des coordonnées de textures.
- Enfin chaque ligne débutant par « f » représente une facette. Pour les 3 sommets de la facette, on donne l'indice du vertice/ de la coordonnée de texture/ de la normale. (Il est à noter que les indices commencent à 1 et correspondent à l'ordre dans lequel les v/vt/vn sont déclarés dans le .obj).

A partir de ces informations, il est donc possible de reconstituer une Geometry et un Mesh.

On peut voir qu'il est possible qu'il y ait plus de coordonnées de texture que de vertices. Nous nous sommes rendu compte que dans certains de nos modèles 3D, plusieurs coordonnées de texture étaient attribuées à un même vertex. Il a donc fallu traiter ce cas en dupliquant les normales et coordonnées des vertex concernés.

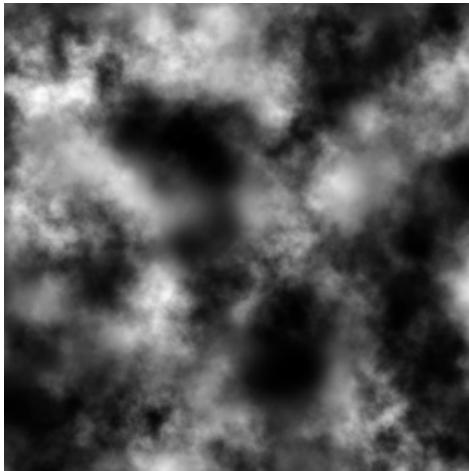
Enfin, nous avons décidé de ne pas gérer les smoothing group. D'une part, cela aurait demandé de gérer plusieurs Geometry par Mesh, et d'autre part, nous avons estimé que le rendu était satisfaisant sans cette technique.

<sup>2</sup> <http://www.limegarden.net/2010/03/02/wavefront-obj-mesh-loader/>

## 2.2. Génération de forêt

### Génération de terrain (Height Mapping)

En ce qui concerne la génération du terrain nous utilisons une height map. Celle-ci consiste en une image en niveau de gris. Le chargement de l'image dans le programme est réalisé grâce à la classe QImage mise à disposition par Qt.



*La height map utilisée dans notre application*

Une fois l'image chargée dans le programme, il nous suffit de la parcourir pixel par pixel pour construire le maillage du terrain. En effet, chaque pixel de l'image représente un vertex du maillage. Les coordonnées x et z correspondent à la position du pixel dans l'image, multipliées par un coefficient définissant la taille du terrain voulu. Nous faisons également en sorte que le terrain soit centré sur l'origine du repère.

Quant à la coordonnée y, celle-ci est obtenue grâce à la valeur de la couleur du pixel multipliée par un coefficient déterminant la hauteur maximale du terrain. Nous avons aussi implémenté un modificateur permettant, en augmentant le range des hauteurs générées avec la distance au centre de la map, de créer des montagnes aux abords du terrain.

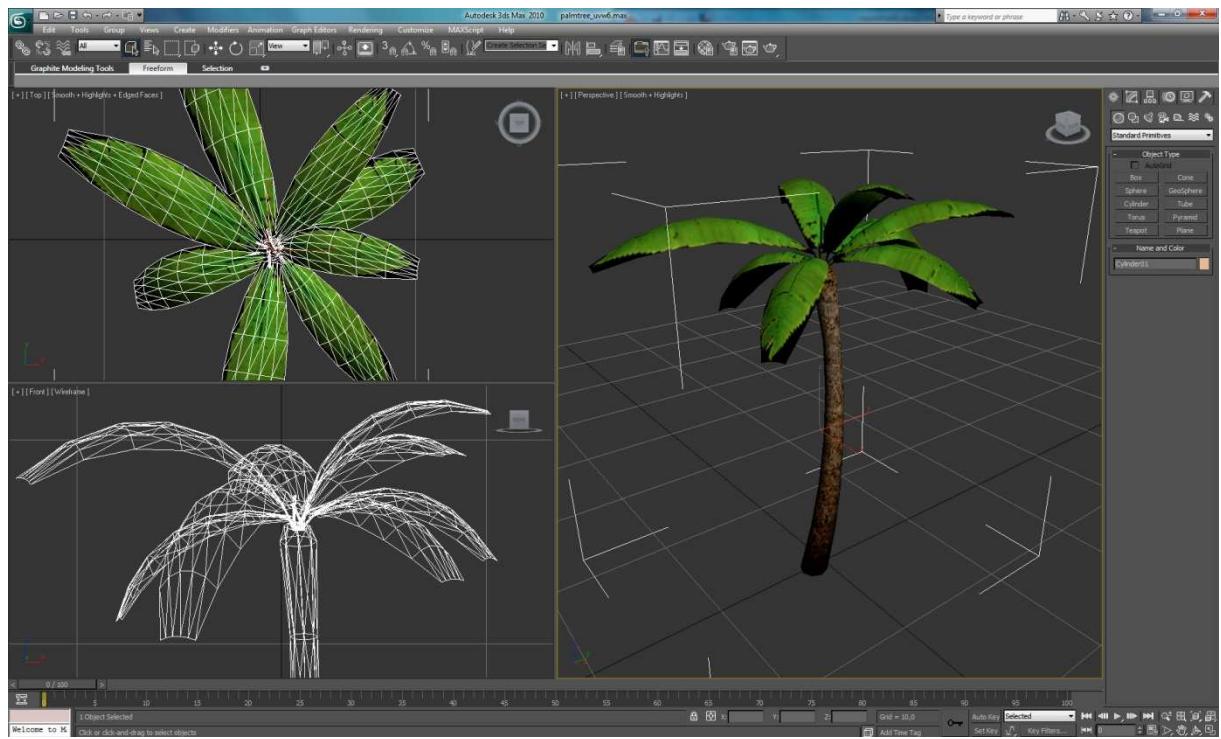
Lors de la création des vertices du maillage, nous effectuons également le calcul des normales à chaque plan. Ensuite, lors d'un second passage nous calculons les normales à chaque point. Ces normales vont par la suite nous servir pour le calcul d'illumination.

### Gestion d'ambiance sonore

Cette fonctionnalité est assurée par la classe QSound disponible sous Qt. La musique est lancée au démarrage de l'application. L'utilisateur a ensuite le choix de stopper ou réécouter la musique en cliquant sur le bouton « play/stop » disponible dans la fenêtre de paramétrage.

### Création de modèles 3D de végétation

Dans le cadre de notre projet, nous avons dû réaliser différents modèles 3D de végétation et de roche. Nous avons pour cela utilisé le logiciel 3ds Max, et nous nous sommes servis de textures provenant de [cgtextures.com](http://cgtextures.com).



La difficulté fut ici de créer des modèles 3D réalistes mais au nombre de polygones restreint afin de pouvoir générer le plus d'instances possible sans perdre trop de performances.

La plupart de nos modèles sont donc en dessous de 400 polygones.

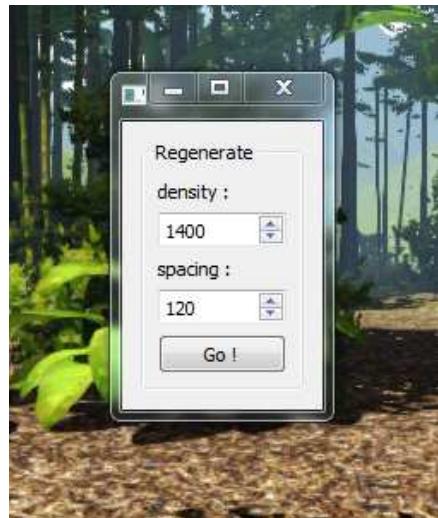
### Création de la scène et génération d'instances aléatoires d'objets 3D

La classe SceneBuilder nous permet de créer notre scène et d'y placer tous les objets. C'est dans cette classe qu'on initialise le soleil, le ciel et le terrain et la végétation.

Scene
meshes: QList<Mesh*>
sun: Sun
cameras: QList<Camera*>
sky: SkyBox
ground: Ground
currentCamera: Camera
renderable: bool
update()
init()
[...]

Lors de la génération des objets la taille des instances, leur orientation et leur disposition sur le terrain sont définies aléatoirement.

Cette méthode est ensuite appelée à chaque fois que l'utilisateur souhaite re-générer une forêt. Cette action lui permet de choisir la densité (le nombre d'arbres) et l'espacement entre les arbres. Toutes les instances sont alors supprimées et ensuite recréées avec les nouveaux paramètres.



Une bounding box sphérique est associée aux instances de type « rocher » afin d'éviter que les arbres ou arbustes poussent au travers de la pierre. Pour cela, la position et le rayon de chaque instance de rocher sont stockés dans une liste. Nous vérifions ensuite lors de la génération des autres instances, que leur position n'est pas en conflit avec les bounding box des rochers.

### Génération de chemin

La classe Path permet la création d'un chemin au travers de la forêt.

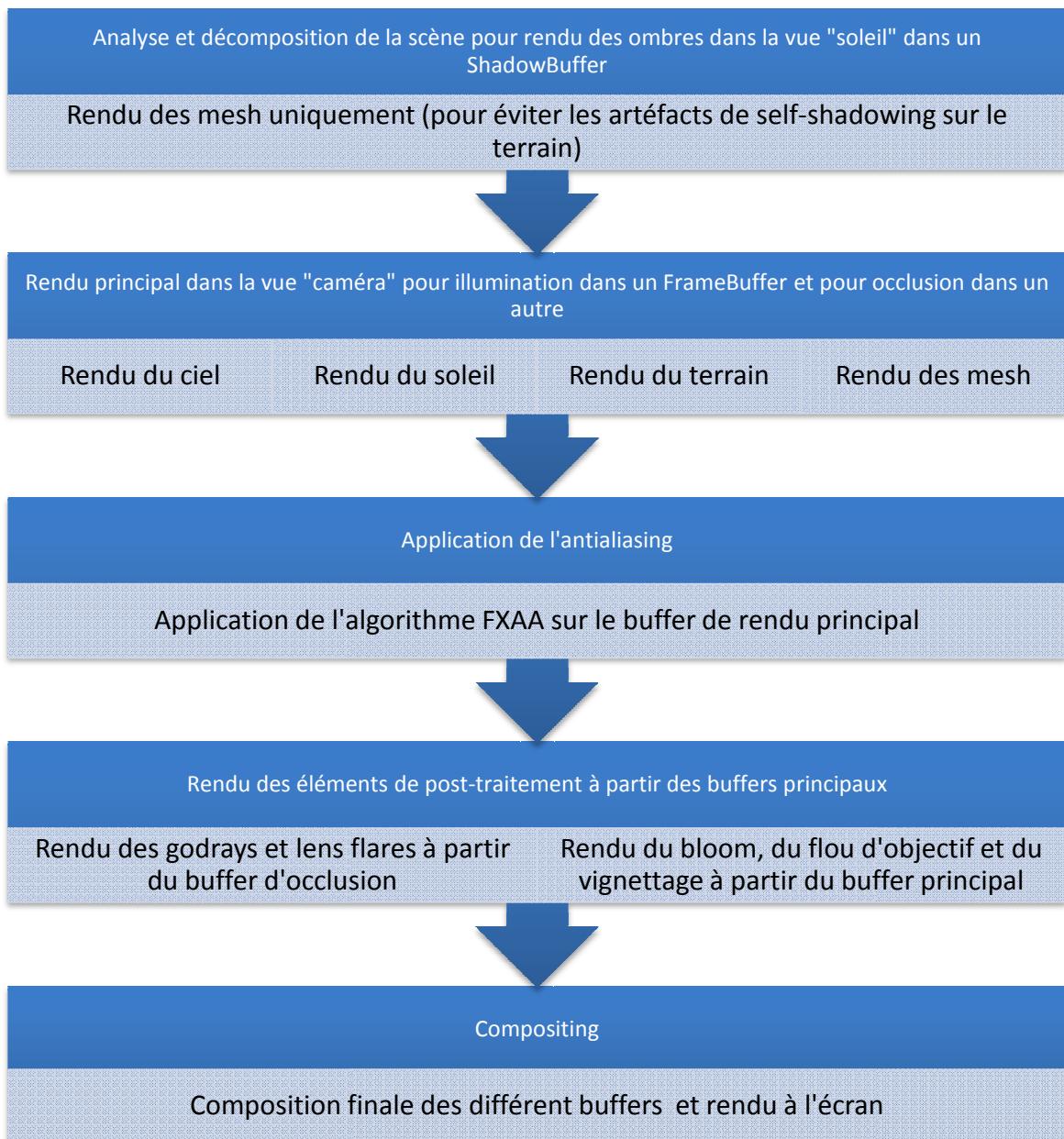
Le chemin est généré à l'aide d'une image noir et blanc. La partie blanche de l'image désigne le chemin et la noire les emplacements disponibles pour les instances.



Lors du calcul des positions de chaque instance, on vérifie si elles sont comprises ou non dans le chemin. Pour cela, on convertit la position de notre instance dans le monde en coordonnées dans l'image, on récupère ensuite la couleur du pixel et on vérifie que cette couleur n'est pas blanche.

## 2.3. Rendu

### Pipeline de rendu



## Illumination et textures

### **Illumination**

Pour notre rendu, nous étions à l'origine partis sur le modèle de Blinn-Phong, cependant, les spéculaires résultantes donnaient un aspect très "plastique" à notre rendu, cela dénaturait notre scène et la rendait peu réaliste.

Pour pallier à cela nous avons envisagé et implémenté la technique de normal mapping permettant d'influer sur la normale d'une facette grâce à une texture, permettant ainsi d'avoir des ombres et des spéculaires plus réalistes. Cependant, nous n'avons pas retenu ce modèle car malgré un certain nombre d'essais, il nous a été impossible de créer des textures normales de qualité suffisante. En effet, nous avons tenté de réaliser ces textures grâce à un outil développé par Nvidia nommé Nmapper. Cependant, avec ces textures auto-générées, le rendu n'était pas idéal et loin d'être réaliste, nous conservions toujours cet effet "plastique" même après avoir paramétré notre rendu Phong.

Nous avons finalement décidé de ne conserver qu'un modèle d'illumination de Lambert, sans spéculaires. C'est selon nous le modèle qui donnait les résultats les plus réalistes étant donné nos difficultés à créer de bonnes normal maps.

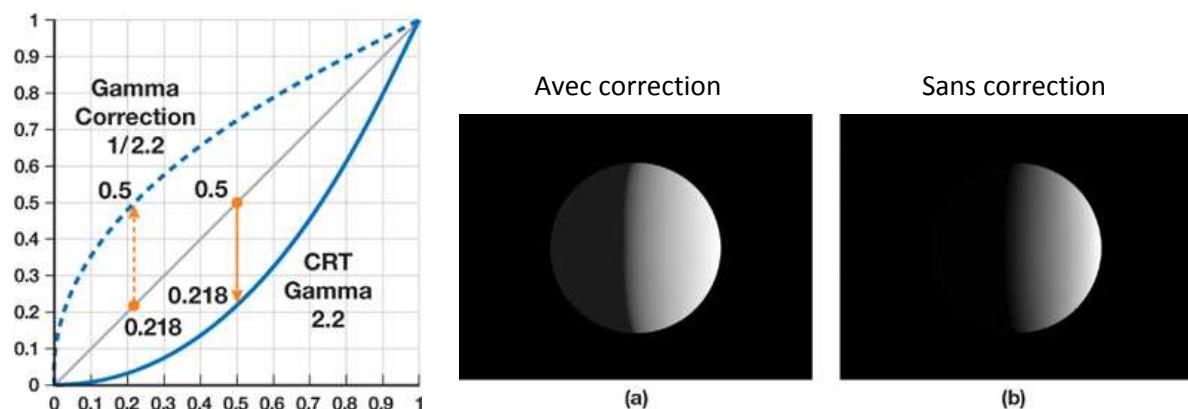
### **Textures**

Pour ce qui est des textures, nous tenons à noter l'utilisation d'un filtre d'anisotropie, permettant d'obtenir des textures de meilleure qualité lorsque celles-ci ne font pas face à la caméra grâce à un multi-échantillonnage directionnel.

De plus, nous avons utilisé la technique du mip-mapping, stockant des versions redimensionnées de nos textures dans la mémoire graphique, apportant performance et qualité aux textures en fonction de leur aspect et distance.

### **Correction gamma**

Enfin, dernier point important de notre illumination et texturing, nous appliquons une correction gamma à nos textures en entrée puis en sortie lors du rendu principal afin de passer en espace couleur linéaire le temps de l'illumination, pour un rendu encore plus réaliste en se basant sur un article publié dans le volume 3 des GPU Gems<sup>3</sup> édité par Nvidia.



<sup>3</sup> [http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch24.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html)

## Ombres

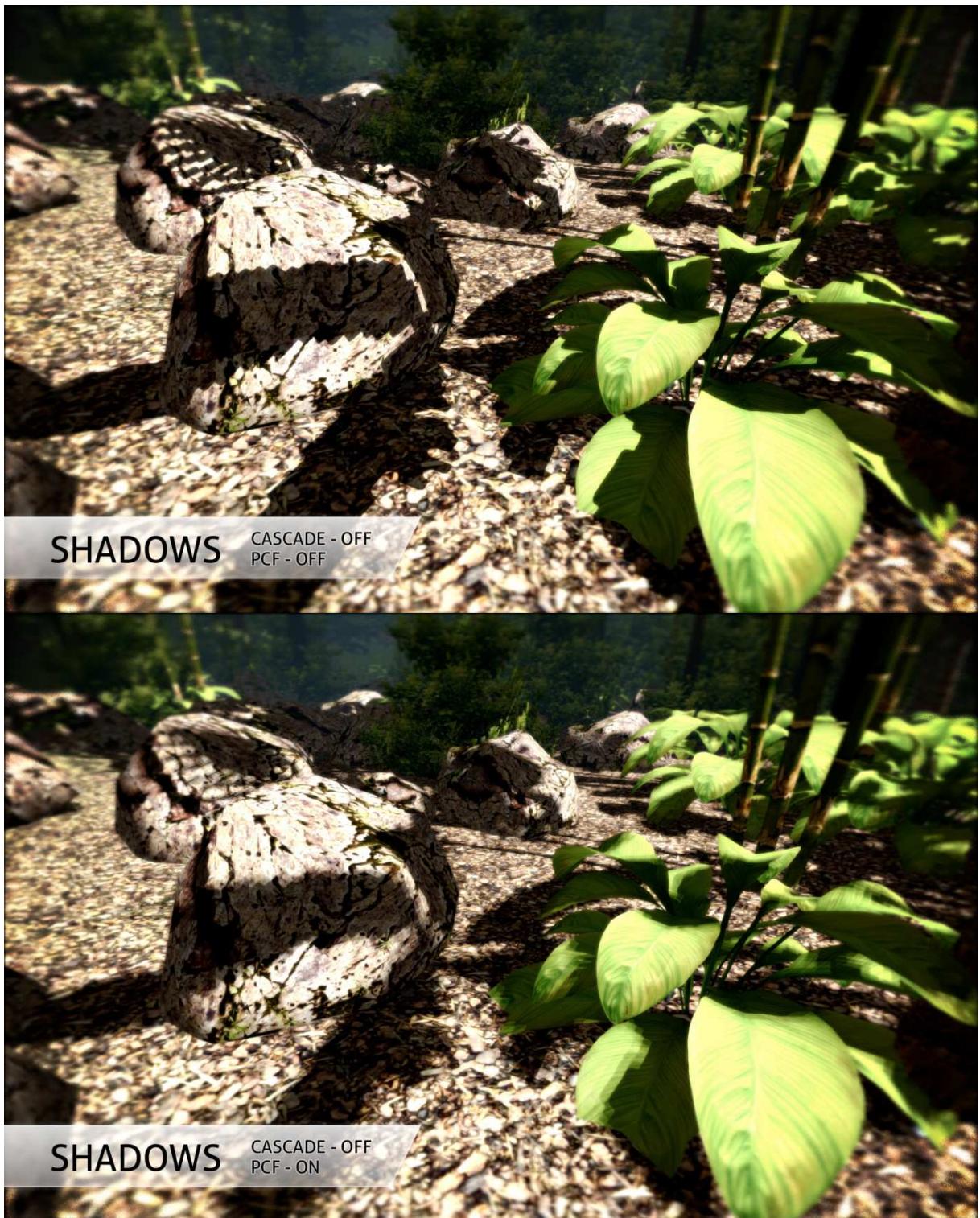


Notre moteur graphique dispose du support des ombres portées. Cette technique se déroule en deux étapes principales, la première étant le rendu de la géométrie du point de vue du soleil, en projection orthogonale (car c'est une lumière directionnelle) le tout dans un ShadowBuffer encodant la profondeur des pixels sur 24 bits. Comme dit précédemment, le terrain n'est pas rendu pour cette passe afin d'éviter les artéfacts dits de "self-shadowing".



La deuxième étape consiste, lors du rendu principal, à récupérer la profondeur du pixel courant dans la vue du soleil et à la comparer à sa profondeur réelle afin de déterminer si le pixel courant est dans l'ombre ou pas.

## Lissage PCF des ombres



Ci-dessus, deux captures d'écran exposant notre Percentage-Close Filter (PCF) pour nos ombres.

Ce filtre permet, en évaluant les pixels alentours au pixel courant, de lisser les ombres portées. Ainsi, au lieu d'évaluer la profondeur sur un seul pixel, nous l'évaluons sur le pixel courant ainsi que sur ses 8 voisins avant de mélanger le résultat pour obtenir notre intensité d'ombre finale.

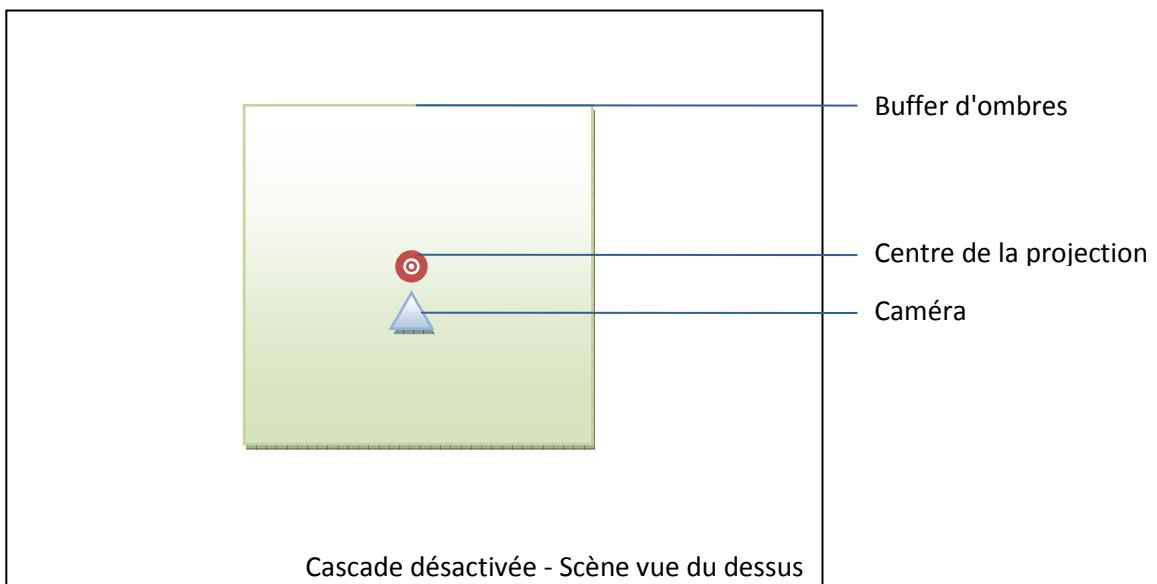
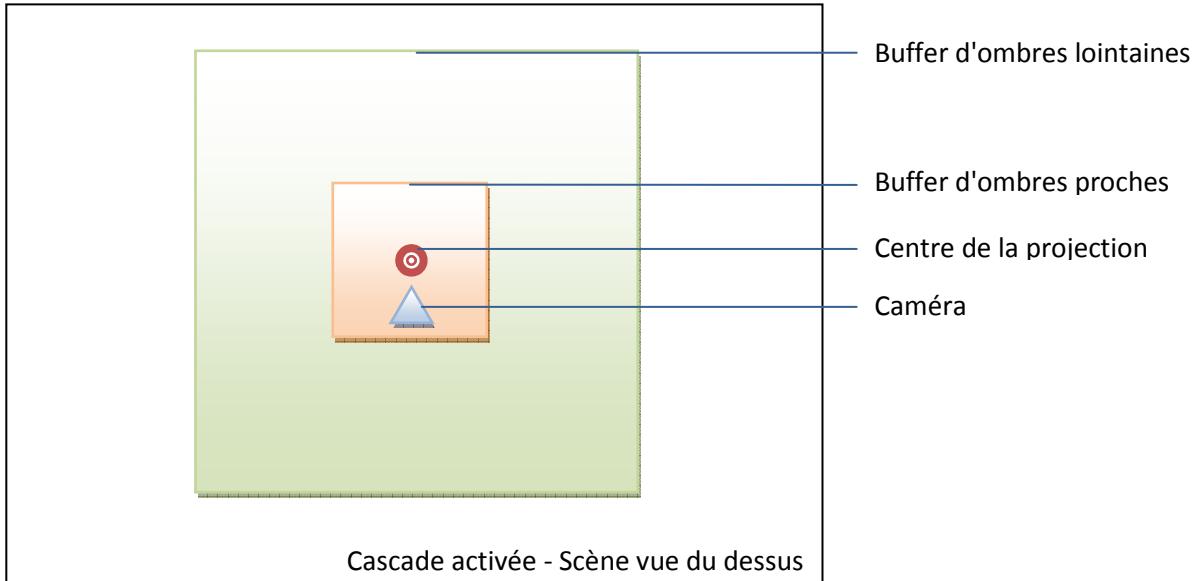
## Ombres en cascade



Ci-dessus, deux captures d'écran exposant notre algorithme d'ombres en cascade.

Cet algorithme permet, en utilisant plusieurs buffers d'ombres, d'augmenter de façon significative la qualité et la portée des ombres. Ainsi, un buffer va recevoir les ombres proches en pleine résolution, alors qu'un deuxième buffer recevra les ombres éloignées de résolution moindre.

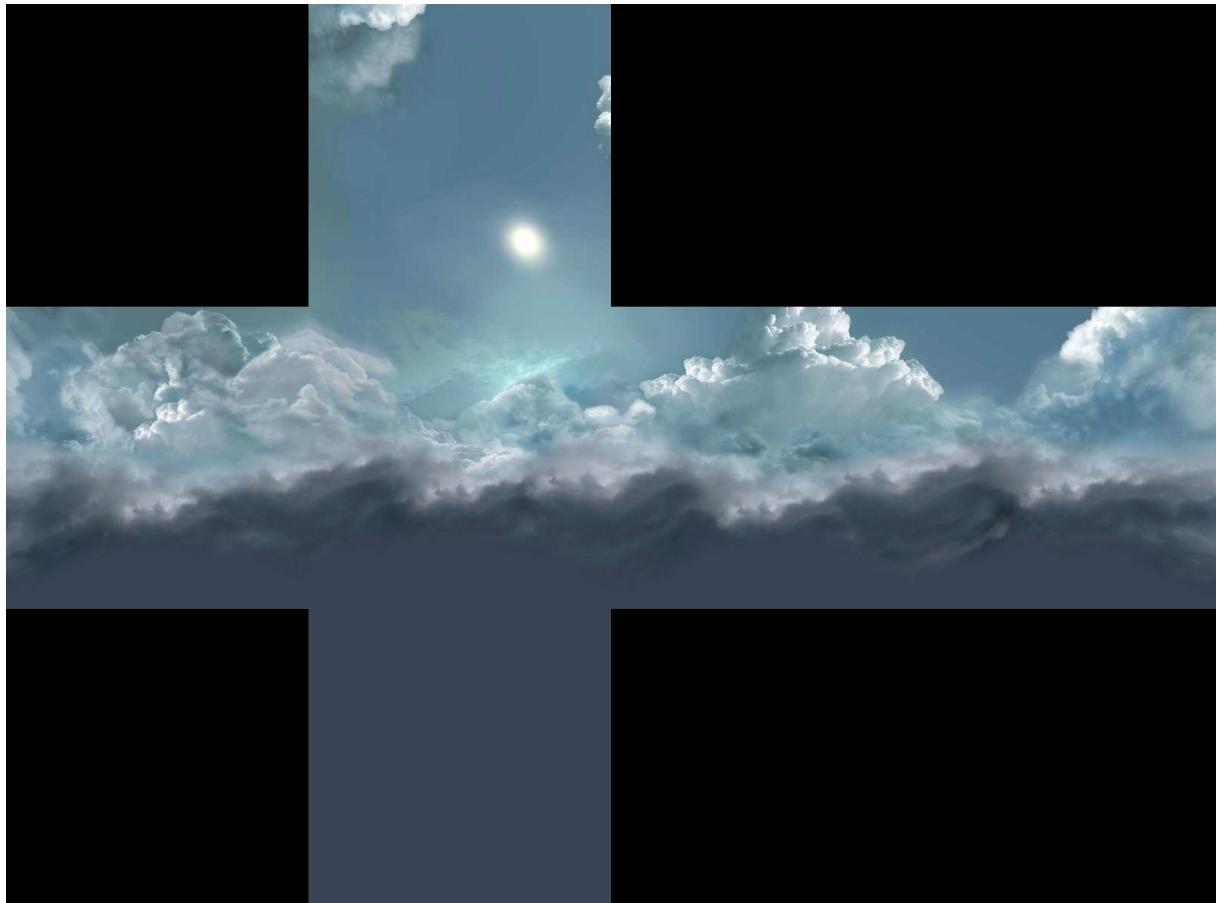
De plus pour optimiser la zone des ombres proches, la projection est légèrement décalée dans la direction de la caméra afin d'avoir une zone d'ombres détaillées plus importante devant que derrière la caméra.



## Skybox et sky mapping

Afin de réaliser le rendu du ciel, nous avons utilisé la technique de cube mapping, qui consiste à appliquer une texture sur chacune des faces d'un cube. En plaçant notre caméra à l'intérieur de ce cube, cela donne l'illusion d'avoir un ciel tout autour de nous.

Nous avons ici utilisé une texture libre de droits créée par une personne répondant au pseudo "Hipshot" :



## 2.4. Post-traitement

### FXAA (Fast Approximation Anti Aliasing)



Ci-dessus, deux captures d'écran exposant la mise en place d'un algorithme d'antialiasing en post-processing appelé FXAA pour Fast Approximation Anti Aliasing.

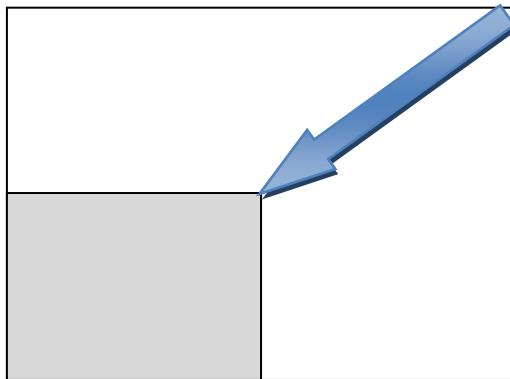
Cet algorithme nous vient de Timothy Lottes<sup>4</sup>, développeur chez Nvidia. C'est un algorithme de lissage de contour ne nécessitant qu'une seule passe de fragment shader, et le seul buffer principal du rendu de la scène comme entrée.

Nous avons utilisé ici la version 3.8 en se basant sur le port GLSL de GLGE<sup>5</sup>.

Cet algorithme utilise une détection de contours basée sur les écarts de luminance, et permet de lisser les contours d'une scène suffisamment contrastée et saturée de façon beaucoup plus rapide qu'un antialiasing matériel par sous-échantillonnage.

L'un des inconvénients de cette technique reste le léger flou de l'image finale, notamment au niveau des détails de textures car le lissage s'applique après le rendu de la scène et ne tient pas compte de la géométrie. Ce flou pourrait être réduit par un unsharp-mask, mais pour des raisons de performances et de goûts, nous avons décidé de ne pas appliquer de filtre de netteté.

## Downsampling



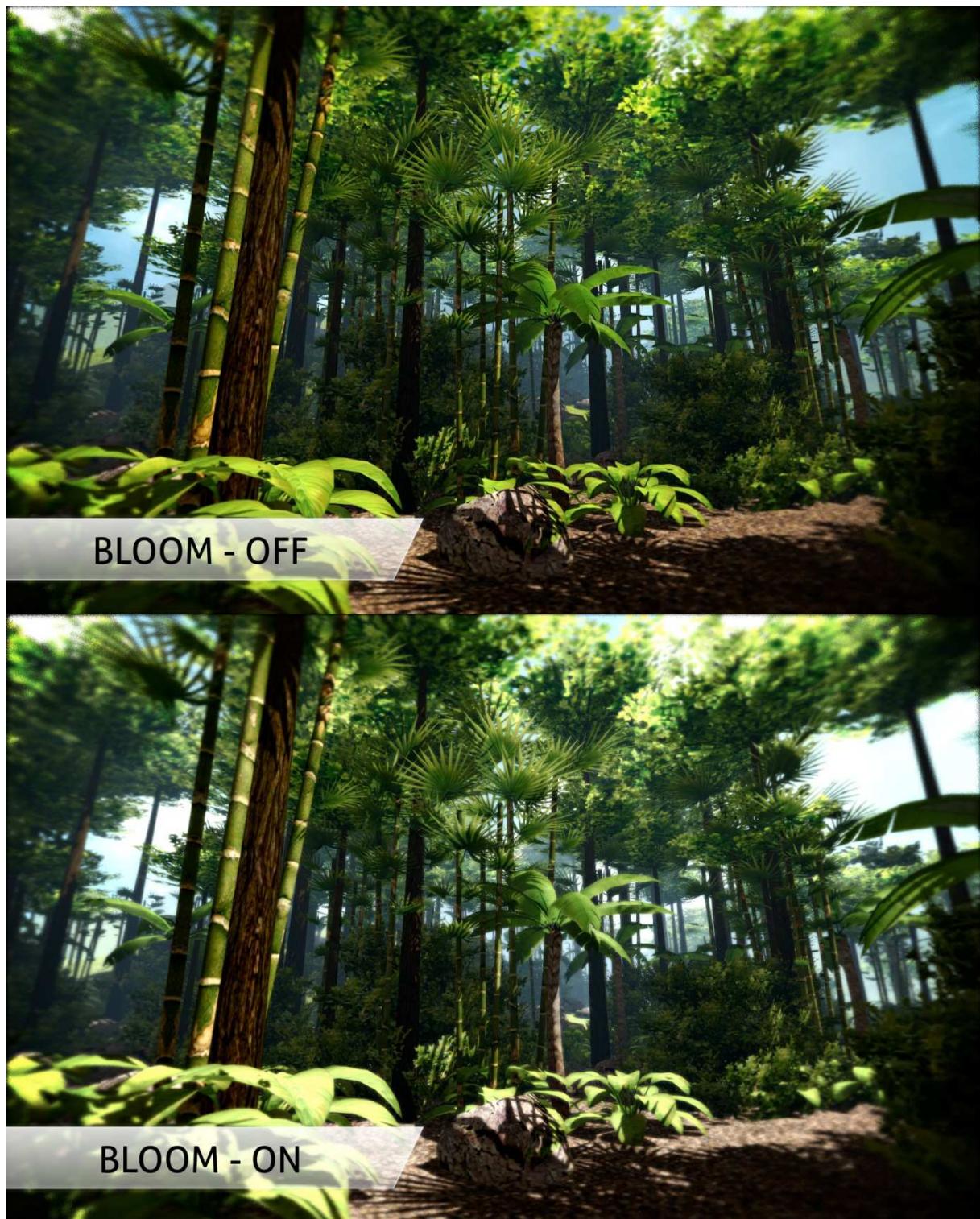
Par la suite, nous allons voir différentes implémentations d'effets de post-traitement qui sont en eux même très coûteux au niveau des performances.

Afin de pallier en partie à ce problème, la plupart de ces algorithmes sont appliqués sur des tiers voir des quarts de buffers. C'est-à-dire que les framebuffers utilisés pour ces rendus sont de tailles inférieures à la fenêtre de rendu final. C'est ce qui s'appelle le "downsampling". Cela permet d'avoir un nombre de pixels beaucoup moins important à traiter par les fragment shaders et donc, de gagner en performance.

<sup>4</sup> <http://timothylottes.blogspot.fr/2011/03/nvidia-fxaa.html>

<sup>5</sup> <http://www.glge.org/demos/fxaa/>

## Bloom HDRA (High Dynamic Range Approximation)



Ci-dessus, deux captures d'écran exposant la mise en place d'un effet de post-processing appelé "bloom".

Cette technique permet d'étendre artificiellement la plage de dynamique (HDRA) en superposant au rendu de la scène une version dont les niveaux ont été modifiés de manière à conserver uniquement les hautes lumières. Afin de simuler le "halo" visible sur les faces éclairées en hautes lumières, un filtre de flou par approximation gaussienne<sup>6</sup> est appliqué avant la superposition.

De plus, pour donner un effet désaturé aux hautes lumières, nous avons simplement passé celles-ci en niveaux de gris en maximisant les composantes RVB avant la mise à niveau. Cela donne un rendu "film" assez intéressant.



<sup>6</sup> <http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>

## Godrays (Volumetric Light Scattering Approximation)



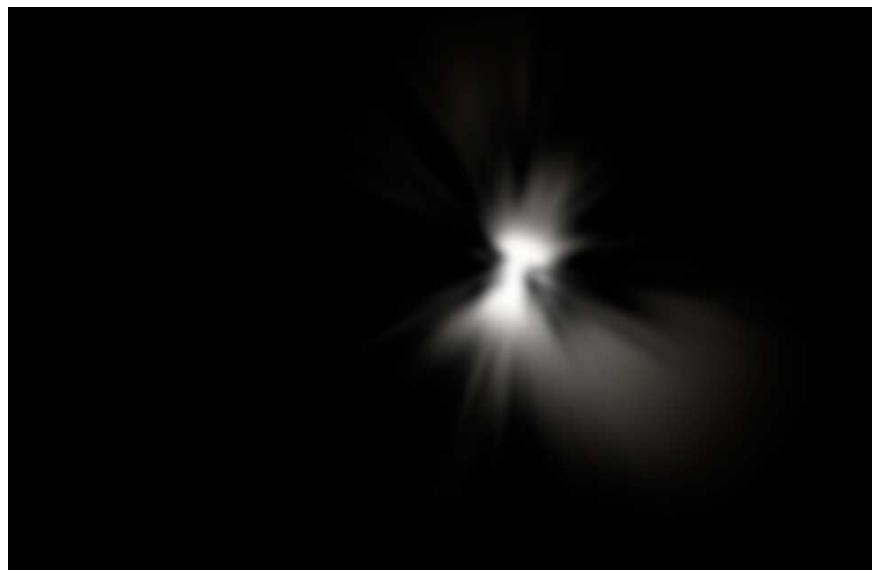
Ci-dessus, deux captures d'écran exposant la mise en place d'un effet approximant la diffusion volumétrique de la lumière, plus communément appelé "godrays".

Basée sur un article<sup>7</sup> écrit par Kenny Mitchell d'Electronic Arts, parut dans le volume 3 des *GPU Gems* de Nvidia, cette technique approxime l'effet de rais de lumière lorsque une source lumineuse est occulte par des objets opaques.

Pour réaliser cet effet, il nous fallait rendre dans un premier temps la scène de manière à obtenir un buffer d'occlusion, c'est-à-dire, un buffer où seules les sources lumineuses sont représentées en blanc, et le reste des objets occultant sont représentés en noir.



Après cette première passe, nous appliquons un filtre de flou radial centré sur la position du soleil, suivit d'un flou rapide, identique à celui utilisé pour le bloom, afin de lisser le résultat (car nous utilisons des buffers de moindre résolution).



Le résultat est finalement superposé au rendu standard afin d'obtenir ce que nous avons en page précédente.

---

<sup>7</sup> [http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch13.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html)

## Lens flare (parasite et dispersion de lumière)

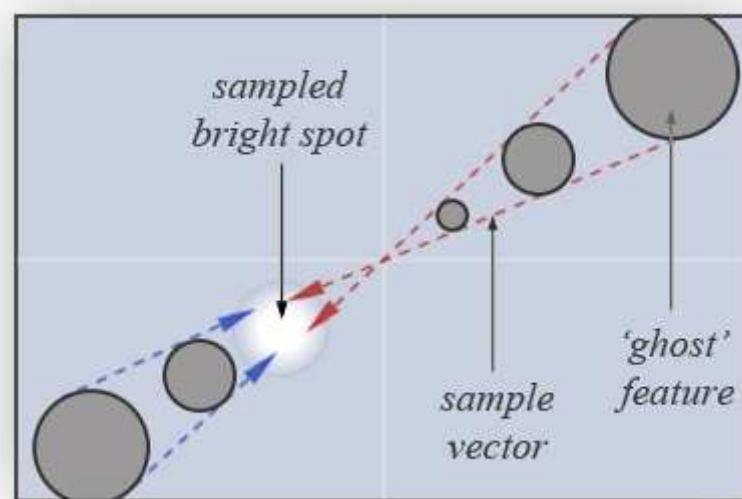


Ci-dessus, deux captures d'écran exposant la mise en place d'un effet simulant les parasites et la dispersion de la lumière au travers de l'objectif, plus communément appelé "lens flare".

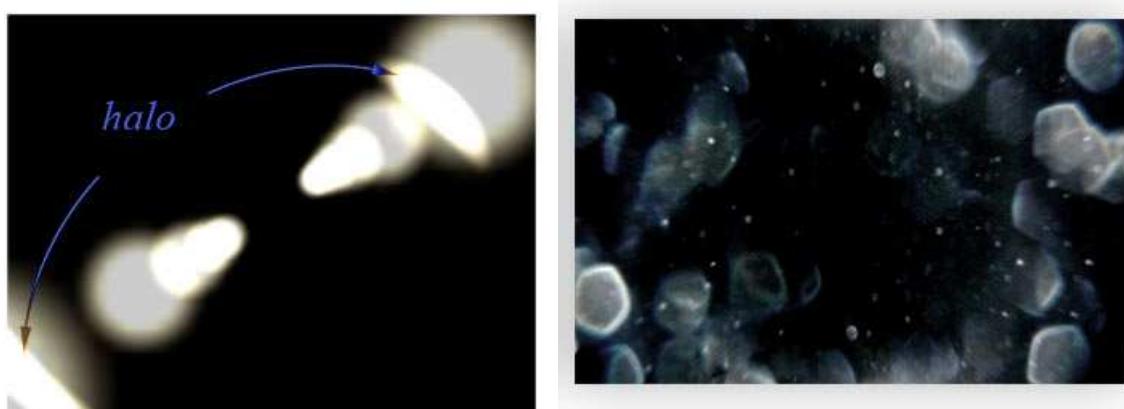
Basée sur un tutorial<sup>8</sup> écrit par John Chapman, cette technique permet de simuler de manière artistique et non physique les effets de lumières notables sur un objectif de caméra.

La première passe de rendu de cet effet consiste à obtenir une version inversée par symétrie centrale du buffer d'occlusion.

La seconde passe génère les différentes formes lumineuses de la dispersion en échantillonnant les pixels en direction de la source lumineuse de manière radiale.



Enfin, lors de la passe de composition avec le buffer de rendu principal, on mélange ce buffer de dispersion avec une texture de "salissure de lentille" en pleine résolution pour obtenir le résultat vu en page précédente.



<sup>8</sup> <http://www.john-chapman.net/content.php?id=18>

## Flou d'objectif



Cette technique utilise un simple noise blur<sup>9</sup>, modulé en fonction de la distance au centre, en une seule passe afin de conserver des performances suffisantes car cet effet est appliqué en pleine résolution.

---

<sup>9</sup> <http://devlog-martinsh.blogspot.fr/2011/10/glsl-cubic-lens-distortion.html>

### 3. Performances

#### 3.1. Notes

Le nombre d'images par seconde maximal est fixé à 32 fps.

**Trois facteurs sont variables durant les tests :**

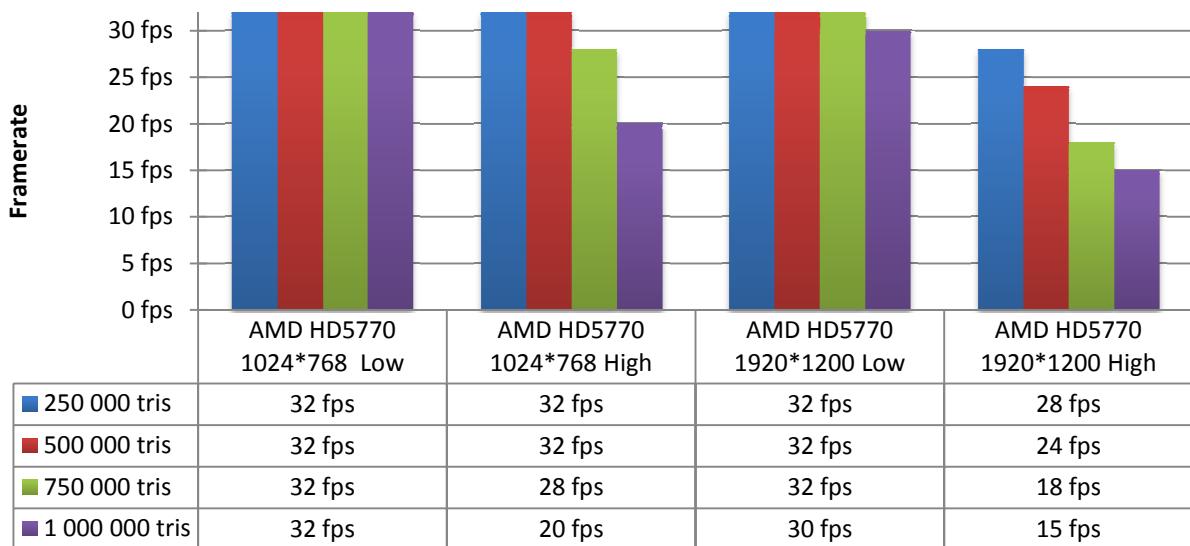
- La résolution :
  - 1024\*768px
  - 1920\*1200px
- Les effets :
  - **Low** : La plupart des effets sont désactivés.  
Détails :
    - Ombres de base
    - Filtrage bilinéaire classique
    - Pas d'anti-aliasing
    - Downsampling sur quarts de buffers.
  - **High** : Les effets par défaut sont activés  
Détails :
    - Ombres en cascade avec filtre PCF
    - Filtrage anisotropique
    - Anti-aliasing FXAA
    - Godrays
    - Bloom
    - Vignettage
    - Downsampling sur tiers de buffers
- La taille de la forêt :

Nombre d'instances	Nombre de polygones (approximatifs)
1000	250 000 tris
2000	500 000 tris
3000	750 000 tris
4000	1 000 000 tris

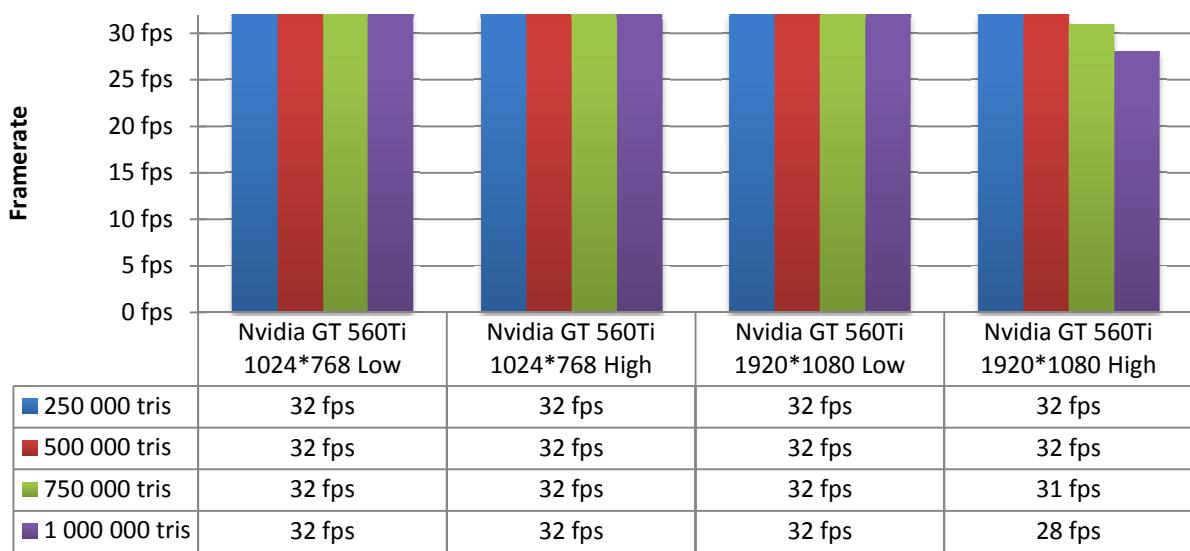
**Nos benchmarks ont été réalisés sur deux cartes graphiques :**

- AMD HD5770
- Nvidia GT 560 Ti

## AMD HD5770 - Benchmark



## Nvidia GT 560 Ti - Benchmark



## **Conclusion**

Ce projet fut pour nous un véritable challenge, tant sur le plan organisationnel que technique. Il nous a fallu acquérir un certain nombre de connaissances par nos propres moyens, ainsi qu'utiliser ces mêmes connaissances en un temps relativement court (trois mois).

Il nous a donc fallu faire preuve d'adaptabilité et de dynamisme tout au long de la réalisation de ce projet afin d'arriver à un produit suffisamment abouti au bout de ces trois mois.

Sur un plan plus technique, ce projet nous a permis d'approfondir de manière conséquente une parties des connaissances vues en cours, et nous a permis de comprendre l'enjeu de la réalisation d'un moteur 3D de A à Z.

## **Continuation**

Pour la postérité, voici une liste non-exhaustive des nombreuses améliorations envisageables à notre projet :

- Réalisation de normal maps dignes de ce nom.
- Optimisations, notamment avec la mise en place d'un système de LOD et de frustum culling.
- Ajouts de plus de modèles.
- Implémentation du SSAO.
- Implémentation d'une caméra pour réaliser des cut-scenes.
- Ajout de la déformation par vertex pour simuler le vent dans les feuillages.
- Implémentation d'un mode "nuit".
- Amélioration des performances.