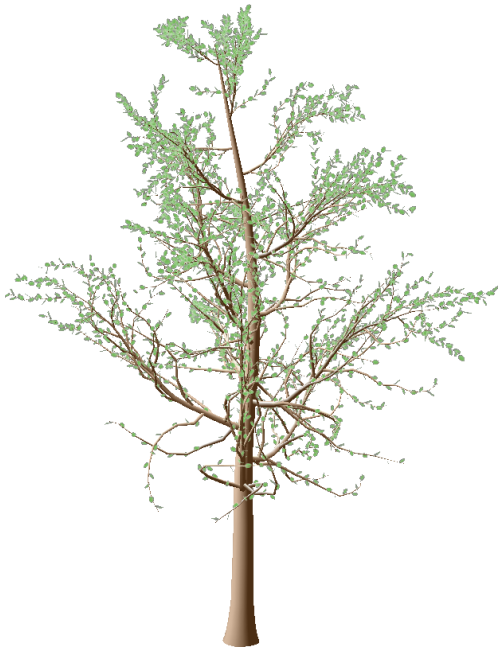
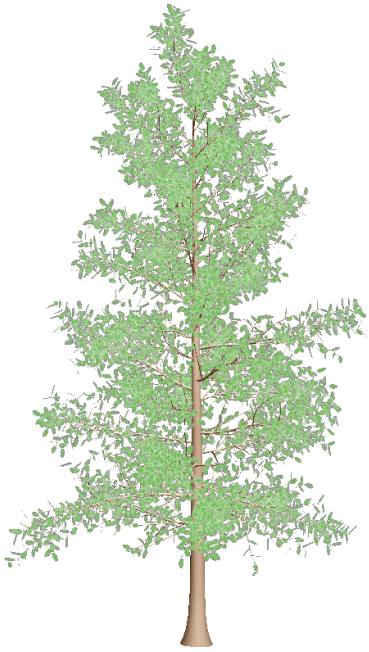
Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Travail de bachelor 2008

Filière Informatique de gestion

Génération procédurale de végétation pour des applications temps réel.





Etudiant : Damien Bagnoud

Professeur : Nicole Glassey

www.hevs.ch

Tables des matières

Chapitre 1

Introduction

1.1 Résumé

Grâce aux avancées récentes dans les domaines du rendu temps réel et à l'augmentation de la puissance des ordinateurs, les jeux récent s'approchent de plus en plus du photoréalisme. Le besoin en qualité et complexité des objets présents sur une scène est donc croissant.

La végétation est particulièrement difficile à représenter, les arbres sont des objets complexes et détaillés, il n'est pas facile de leur donner un aspect réaliste. Jusqu'à récemment, dans la plupart des jeux, leur rendu n'était pas toujours convaincant. Cependant, depuis peu, commencent à apparaitre des solutions permettant de générer de la végétation avec un aspect réaliste.

Le travail présenté dans ce dossier prend cette direction. La méthode utilisée pour la création d'arbres est celle présentée par Jason Weber et Joseph Penn dans le papier « *Creation and Rendering of Realistic Tree* ». Cette algorithme présente l'avantage de permettre de générer un arbre ayant un aspect réaliste tout en n'obligeant pas à l'utilisateur a avoir des notions de botanique ou de mathématique poussées. Une méthode permettant de simplifier la géométrie de la plante avec la distance est aussi abordée dans ce dossier.

Ce travail a porté sur le développement d'une libraire permettant de générer un arbre, d'un éditeur visuel pour aider à la conception et d'un exemple d'implémentation dans un moteur de rendu.

1.2 Aperçu

La représentation de la végétation, de par sa complexité, pose plusieurs problèmes dans les domaines liés au jeu vidéo et aux applications faisant appel au rendu en temps réel (réalité virtuelle, etc...) :

- La demande croissante en réalisme dans le rendu des environnements extérieurs. Jusqu'à récemment la création de la végétation était confiée à des artistes qui la réalisaient à l'aide d'outils de CAO[1] classique, mais il est difficile d'approcher la complexité qu'offre un arbre ou une plante de cette façon. De plus, la qualité n'est pas non plus toujours optimale (feuillage, complexité des branchages, etc…). Le résultat

[1] CAO : Conception Assisté par Ordinateur – CAD en anglais

n'est pas non plus toujours adapté à toutes les situations (simulation du vent ou de différents effets physiques) à moins d'un investissement en temps important.

- Des contraintes de performances de l'application elle-même. Une forêt peux être composée de plusieurs centaines, voir de milliers d'arbre. Il s'agit à ce moment par exemple de dégrader la qualité de la végétation en fonction de la distance à la caméra, de ne charger en mémoire que ce qui est nécessaire ou d'adapter la complexité des arbres en fonction de l'ordinateur sur laquelle l'application est exécutée.

L'application réalisée dans le cadre de ce travail répond principalement aux problèmes liés à la création d'un arbre ayant un aspect réaliste ainsi que de la simplification des détails de cet arbre avec la distance. Outre apporter une solution à ces deux points, ce travail essaie également de répondre à un dernier problème : l'absence de solutions open-source ou à faible coût répondant à ces besoins pour les développeurs amateurs ou ayant un budget limité.

De nombreuses ébauches existent, mais aucune solution aboutie n'est disponible à l'heure actuelle. La plupart des solutions complètes existantes allant dans cette direction sont commerciales, soit indépendant d'un quelconque moteur de rendu (*SpeeTree RT*[2]), soit liées à un moteur de jeu spécifique (*CryEngine*) ou soit internes à un développeur (*Linda* chez Bohemia Interactive Studio). La solution la plus répandue actuellement est *SpeedTree RT*, utilisé par de nombreux jeux à succès (*The Elder Scrolls IV : Oblivion*, etc.) et disponible aussi pour d'autre usages que le jeu vidéo.

L'application développée ici n'est pas en soit un concurrent direct des produits cités précédemment, mais se présente plus comme étant une solution simple à prendre en main et à utiliser dans le cadre de projet de petite taille.

Ce travail a été réalisé en s'appuyant sur un modèle développé par Jason Weber et Joseph Penn, décrit dans « *Creation and Rendering of Realistic Tree* » et présenté lors du *Siggraph '95*[3]. Ce modèle présente plusieurs avantages. Aucune connaissance poussée en botanique n'est nécessaire pour implémenter l'algorithme. L'utilisateur final n'a pas non plus besoin d'avoir de connaissance en botanique ou en mathématique pour utiliser une application conçue sur ce modèle et la création d'un arbre repose sur plusieurs paramètres basiques permettant de le décrire.

Le produit fini est composé de deux éléments, une librairie de génération qui à partir d'une liste de paramètres est capable de générer un arbre et un éditeur visuel permettant de visualiser les modifications apportées à un arbre.

Le présent document décrit dans un premier temps l'organisation de base de l'application développée, puis présente plusieurs termes de vocabulaire spécifique au rendu temps réel. Le chapitre suivant présente les règles de génération utilisées de façon détaillée ainsi que le fonctionnement logique de la méthode de dégradation à distance utilisée. Le quatrième chapitre comprend un guide d'utilisation décrivant l'implémentation de la librairie ainsi que le détail du processus de création d'un arbre. Un guide d'utilisation de l'éditeur visuel est aussi

[2] http://www.speedtree.com

[3] Disponible sur : http://portal.acm.org/citation.cfm?id=218427 et ainsi qu'en annexe.

présenté. Pour terminer, les problèmes connus de la librairie sont présentés ainsi que les évolutions futures planifiées. En annexe sont disponibles les ressources utilisées pour développer ce projet, des exemples de code source présentant une implémentation de base du travail ainsi qu'une copie du document de référence de Weber et Penn.

1.3 Description de l'application

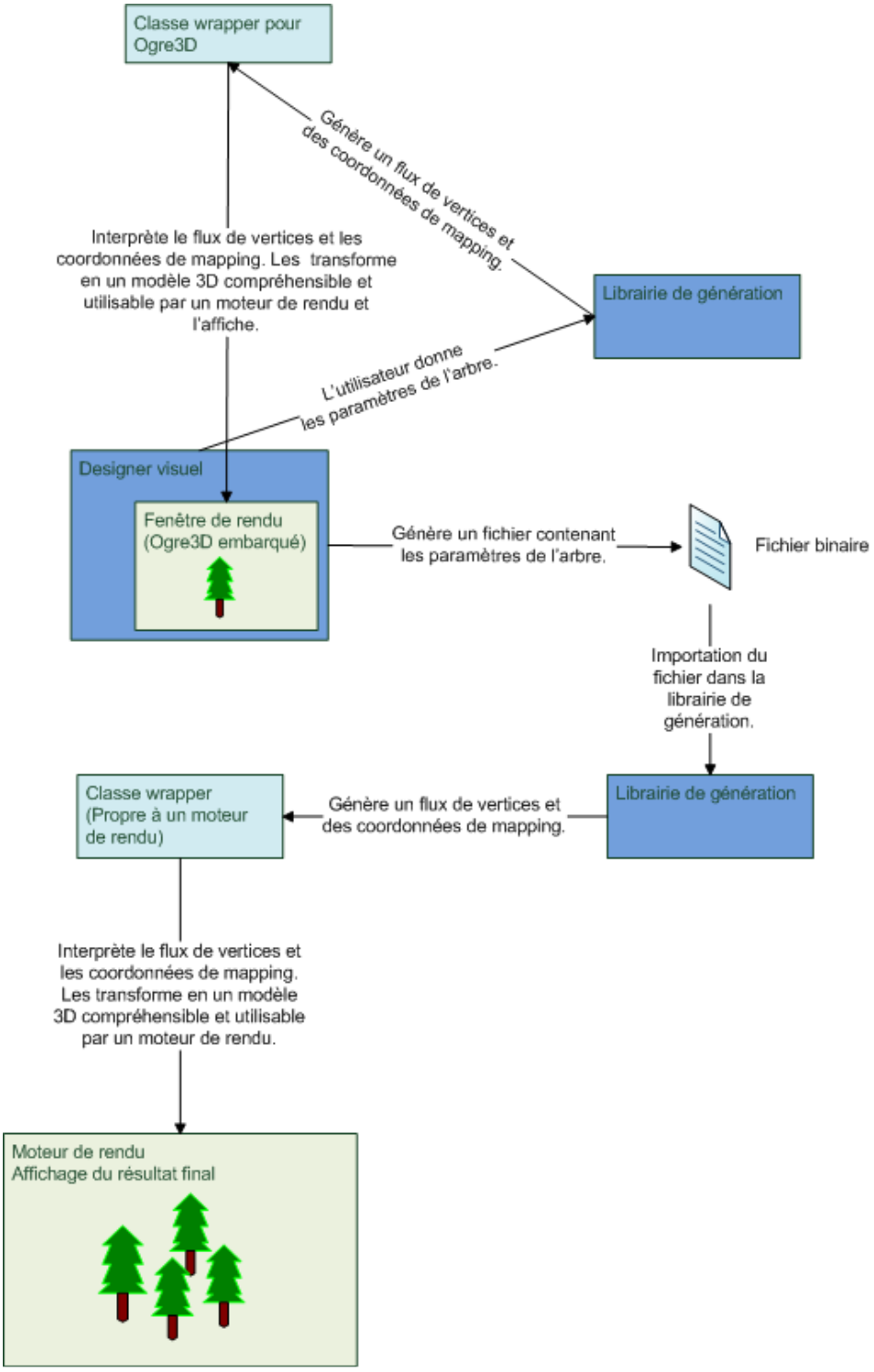


Figure 1.1 Description du flux de données entre les composants du projet.

Comme présenté précédemment, ce projet est composé de deux parties principales. La première partie est une librairie de génération d'arbres. En fonction de paramètres donnés cette libraire permet de générer un flux de données pouvant être interprété par un moteur de rendu pour afficher un arbre. Cette libraire n'est pas être dépendante directement d'un moteur de rendu en particulier et peut facilement s'adapter à la plupart des moteurs, moyennant l'écriture d'une classe intermédiaire (« wrapper »). Cette classe wrapper implémente le code permettant de faire le lien entre la librairie et le moteur. Elle contient par exemple le code permettant d'interpréter le flux de données généré pour créer un objet utilisable par le moteur. Dans le cadre du travail, le moteur de rendu utilisé est Ogre3D.

La librairie est accompagnée d'un éditeur visuel (*Flora Studio*). Cette application s'appuie sur wxWidgets pour l'interface et Ogre3D pour le rendu. Elle permet de visualiser en temps réel les modifications apportées aux paramètres d'un arbre ainsi que de pré-visualiser la dégradation à distance de l'arbre. Il permet aussi de lire et sauvegarder les fichiers contenant la description d'un arbre. *Flora Studio* s'appuie sur la librairie présentée ci-dessus pour générer un arbre.

Les fichiers binaires contiennent les paramètres de description d'un arbre et peuvent être chargés et lus par la librairie de génération elle-même.

A coté de ces deux composants, le travail porte aussi sur le développement d'une petite application de démonstration basée sur Ogre3D.

1.4 Choix techniques

Le projet est développé sur Windows avec comme cible principale cette plateforme. Le développement a été effectué avec Visual Studio 2005. Windows est actuellement la plateforme la plus utilisée dans le jeu vidéo pour le développement.

Le choix du C++ comme langage de développement a principalement été motivé par le fait qu'il s'agit d'un langage encore très répandu dans le monde du jeu vidéo et du rendu temps réel et ce, pour des raisons de performances.

Ogre3D a été choisi pour sa maturité. Ce moteur de rendu, qui existe depuis plus de 5 ans, est bien reconnu, dispose d'une importante communauté et de très nombreuses ressources et tutoriaux en tout genre. De plus dans le cadre de ce travail, il offre de nombreuses facilités entre autre pour la création manuelle d'objets. Ces avantages laissent ainsi plus de temps pour se concentrer sur le développement même de l'application sans avoir à se focaliser sur le développement de la partie de rendu. Par ailleurs, connaissant déjà un peu Ogre3D avant de débuter ce travail, il m'a été plus facile de le prendre en main.

wxWidgets a été en partie choisis pour sa robustesse, sa portabilité, les nombreuses ressources disponibles sur internet, et surtout pour l'existence de plusieurs wrappers pour Ogre3D. L'existence de ces wrappers facilite grandement l'intégration du moteur de rendu. Dans le cadre de ce travail, wxOgre a été utilisé pour faire le lien.

L’avantage supplémentaire que confère l’utilisation du C++, d’Ogre3D et de wxWidgets est que tant le langage de développement que les librairies sont parfaitement multiplateformes, ce qui facilitera le portage du projet par la suite.

Chapitre 2

Bases

2.1 Vocabulaire technique

Ce chapitre à pour but de détailler plusieurs notions de vocabulaire liées au domaine du rendu temps réel pour qu'un lecteur n'étant pas familier avec ces sujets puisse plus facilement comprendre la suite du travail.

Les termes sont listés par ordre alphabétique.

2.1.1 Billboard

Un objet composé de deux triangles formant un quadrilatère toujours orienté en direction de la caméra, utilisé pour les particules et certains effets spéciaux entre autres.

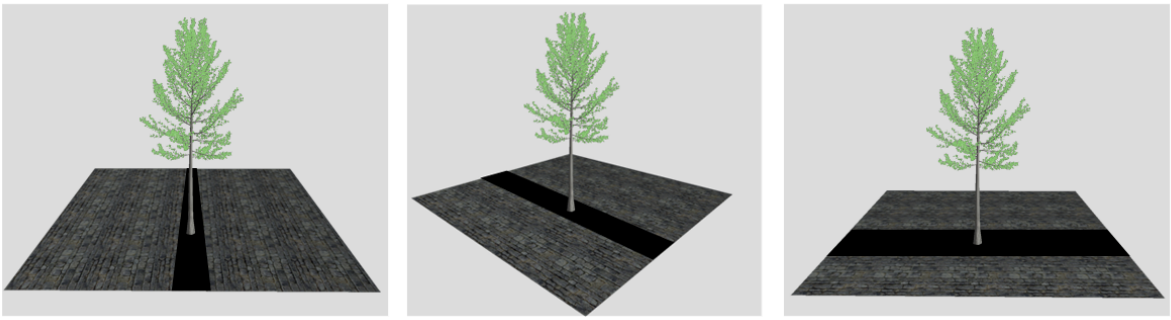


Figure 2.1 Exemple de *billboard*, l'arbre (ici une image appliqué sur un polygone) est en permanence orienté vers la caméra, quelque soit l'angle avec lequel la scène est observée.

2.1.2 Bounding box

Volume contenant entièrement un objet. Les *bounding boxes* ne servent pas à effectuer le rendu d'un objet, mais servent lors de certains calculs pour accélérer les opérations – par exemple la visibilité d'un objet sur une scène : au lieu de tester pour chaque vertex de l'objet s'il est visible, seul le volume dans lequel l'objet est contenu est testé.

2.1.3 DirectX / OpenGL

DirectX est l'API de Microsoft orientée vers le développement de jeu vidéo (rendu 3D). OpenGL est une API open-source multiplateforme pour le rendu 3D en général.

2.1.4 Impostor

Un *impostor e*st une simplification d'un objet 3D complexe, implémenté comme étant un *billboard* dont la texture est un rendu de l'objet 3D en question. Le but des *impostors* est de réduire le temps nécessaire à rendre une scène 3D. Cela fonctionne en mettant en cache des images de cet objet 3D et en utilisant ces images à la place du véritable objet (voir Figure 2.1).

2.1.5 LOD

Technique permettant d'adapter la qualité d'un objet 3D aux besoins de l'utilisateur selon la distance avec la caméra. En effet un objet situé en arrière plan n'a pas besoin d'être autant détaillé qu'un objet situé au premier plan.

Figure 2.2 Exemple de LODs pour un fusil tiré d'un jeu. A gauche le modèle avec différent niveau de détail. A droite le modèle tel qu'il apparaîtrait dans son contexte d'utilisation. L'emploie d'un modèle composé de peu de triangles ne se remarque pas à grande distance. Modèle © *Bohemia Interactive Studio*

2.1.6 Mapping

Méthode permettant d'ajouter des détails (texture ou couleur) sur un modèle 3D ou du contenu généré.

Les coordonnées de texture (*texture coordinates*) correspondent aux coordonnées attribuées aux vertices d'un *mesh* et qui sont utilisées pour appliquer une texture (coordonnées U et V).

Une texture carrée a des cotés ayant toujours une longueur de 1, quelque soit sa taille en pixel (Figure 2.3). Dans le cas d'une texture rectangulaire, son plus grand coté aura une longueur de 1 tandis que l'autre coté aura une longueur fractionnelle proportionnelle.

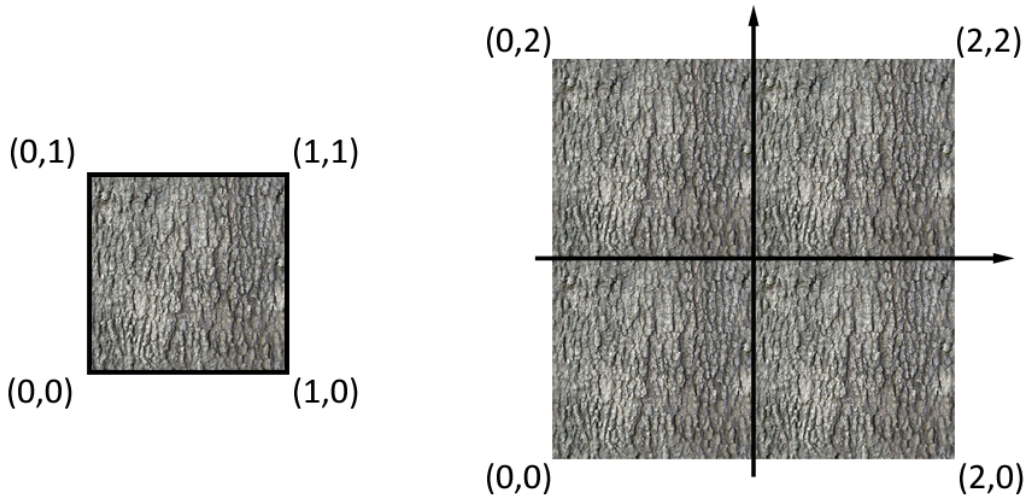


Figure 2.3 Un carré (à droite) composé des vertices (-1, -1), (1, -1), (1, 1) et (-1, 1) est dessiné. On lui applique une texture (à gauche) en utilisant les coordonnées de texture suivante pour chaque vertex du carré : (0, 0), (2, 0), (2, 2) et (0, 2). La texture est dans ce cas répétée en U et V.

2.1.7 Material

Un *material* détermine la façon dont une surface est rendue – couleur, texture, transparence, ainsi que d'autres effets avancés (Bump mapping, etc.).

2.1.8 Mesh

Géométrie d'un objet visible dans une application 3D. Un *mesh* est composé de triangles, eux-mêmes composé de vertices.

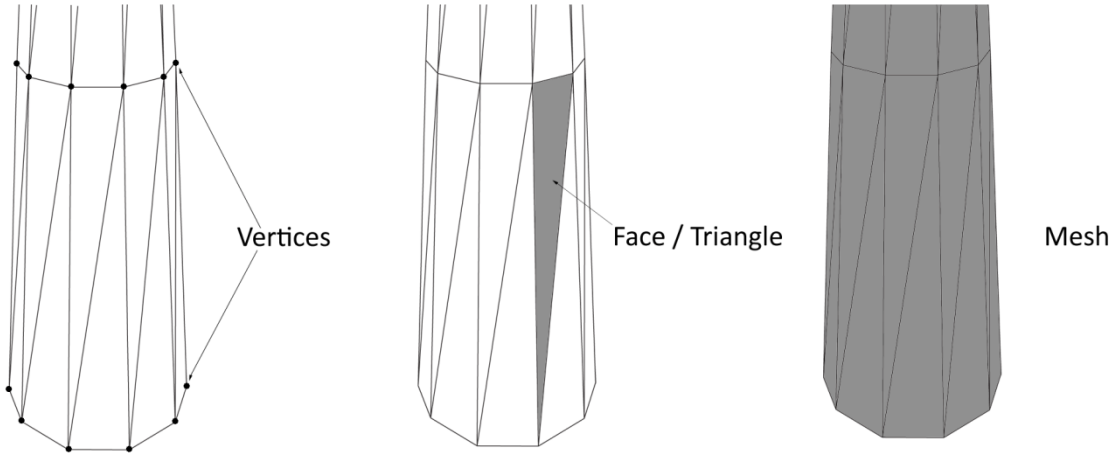


Figure 2.4 Un *mesh* cylindrique sur lequel sont présenté les vertices et les triangles.

2.1.9 Normale

Une normale permet de déterminer la façon dont la lumière interagit avec une surface, la normale est représentée par un vecteur perpendiculaire à une surface quand il s'agit de la normale d'un triangle. Pour la normale d'un vertex il s'agit de la moyenne des normales des triangles adjacents.

2.1.10 Ogre3D

Moteur de rendu 3D open-source et multiplateforme écrit en C++. Conçu pour faciliter le travail des développeurs désirant créer des applications 3D en offrant une couche d'abstraction au dessus de DirectX et OpenGL. Cela permet aux développeurs de travailler sans avoir à tenir compte des spécificités de ces deux APIs.

Site internet : http://www.ogre3d.org

2.1.11 Quaternion

Objet mathématique, les quaternions sont une extension des nombres complexes. Ils sont principalement utilisés dans le rendu temps réel pour effectuer des rotations et représenter des directions. Les quaternions sont plus simples d'usage que les matrices et offrent une plus grande stabilité lors de nombreuses opérations.

L'algèbre des quaternions n'est pas commutative, mais partiellement anticommutative, il est donc important de respecter l'ordre dans lequel sont effectuées les opérations.

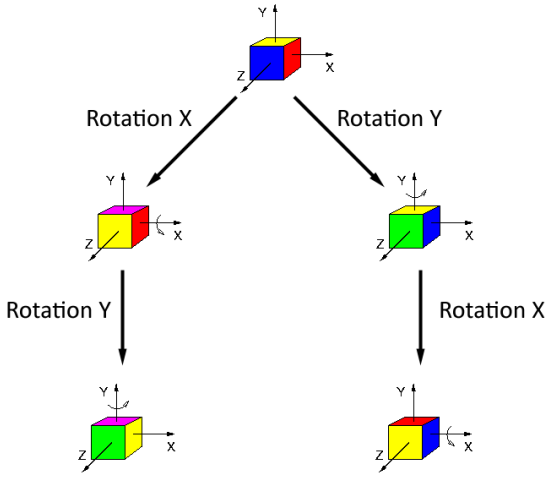


Figure 2.5 Un cube subit les mêmes opérations de rotation, mais dans un ordre différent. Ce schéma montre la non-commutativité des rotations dans l'espace. © fr.wikipedia.org (article « Quaternion »)

Dans le cadre de ce travail, les quaternions sont utilisés pour représenter des directions et pour orienter des vecteurs dans ces directions.

2.1.12 Règle de la main droite

Moyen permettant de se rappeler aisément comment sont liées des directions. Selon la règle de la main droite, lorsque les doigts sont placé perpendiculairement les uns aux autres, le pouce correspond à l'axe X, l'index à l'axe Y et le majeur à l'axe Z. Ce système de coordonnées est utilisé dans le cadre de ce travail.

2.1.13 Render To Texture

Technique de rendu dans laquelle les pixels calculés sont appliqués sur une texture au lieu d'être affichés à l'écran.

2.1.14 Texture

Une image bitmap appliquée sur une surface.

2.1.15 Triangle

Groupe de trois vertices (voir Figure 2.4).

2.1.16 Vertex

Vertices au pluriel.

Correspond à un point unique d'un *mesh* 3D. Les vertices définissent des triangles qui définissent un *mesh*. Un vertex correspond généralement à un ensemble de trois coordonnées X, Y et Z dans l'espace (voir Figure 2.4).

Dans notre cas, lorsqu'un vertex est défini, une normale, une couleur et des coordonnées de mapping lui sont attribuées en plus de ses coordonnées dans l'espace.

2.1.17 Vertex Colour

Couleur donnée à un vertex.

2.1.18 Vertex et Index Buffers

Aussi appelé Vertex / Index Arrays (OpenGL)

Les Vertex et Index Buffers représentent la manière la plus efficace de fournir des informations d'un modèle en 3D à une carte graphique récente. L'idée du vertex buffer et de stocker les données du modèle dans des emplacements contigus de mémoire. L'intérêt réside dans le fait que les données n'ont pas à être copiées de l'application au driver de la carte graphique, l'application ne faisant que transmettre un pointeur vers l'emplacement mémoire au driver qui y accède directement.

Un vertex buffer contient donc les informations sur des vertices similaires, il est donc nécessaire de définir le format de vertex utilisé dans le vertex buffer.

Il existe différente façon de traiter les informations présentes dans un *vertex buffer* :

- Une liste de points individuels
- Une liste de lignes – des paires de vertices.

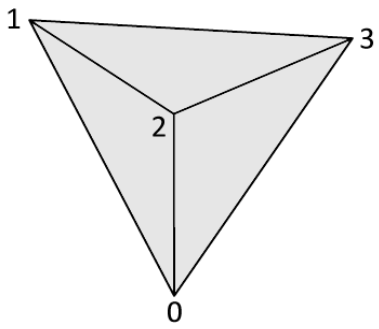
- Une polyligne – deux vertices pour la première ligne, puis un vertex supplémentaire par ligne.
- Une *Triangle List* – défini par trois vertices par triangle.
- Un *Triangle Fan* – groupe de triangles ayant tous un vertex commun, défini par trois vertices pour le premier triangle puis un par triangle supplémentaire.
- Un *Triangle Strip* – liste de triangles définis par trois vertices pour le premier triangle, puis un vertex par triangle suivant.

Dans le cadre de ce travail, le *vertex buffer* n'est pas utilisé de cette façon, nous faisons appel à un *index buffer* (index au singulier, indices au pluriel).

L'*index buffer* est la manière la plus rapide d'accéder à un *vertex buffer*. Chaque triangle y est décrit par trois indices la plupart du temps. De la sorte, l'ordre dans lequel les vertices sont stockés dans le *vertex buffer* à peu d'importance. Chaque index à une valeur comprise entre 0 et n-1, où n représente le nombre de vertices et désigne ainsi vers un vertex du *vertex buffer*.

De cette façon nous pouvons définir un *triangle mesh*, qui représente la manière la plus efficace de partager des données de vertices.

Par exemple, le *mesh* ci-contre est composé de trois triangles et quatre vertices dont leurs positions vont de p_0 à p_3 et de leurs normal n_0 à n_3. Selon la méthode utilisée, le vertex buffer et l'index buffer auront un aspect différent. Utilisé comme dans ce travail, ils auront l'aspect suivant :



Vertex Buffer : $p_0n_0\ p_1n_1\ p_2n_2\ p_3n_3$

Index Buffer : 012 132 302

Le *vertex buffer* contient chaque vertex (position et normale) une seule fois et l'*index buffer* contient la liste des triangles en donnant chaque fois l'identifiant du vertex correspondant dans le vertex buffer.

2.1.19 wxWidgets

Toolkit Open Source et multiplateforme écrit en C++ permettant la création de GUI exportable facilement sur plusieurs plateforme (Linux, MacOS, Windows).

Site internet : http://www.wxwidgets.org

Chapitre 3

Règles de génération

3.1 Introduction

Les règles utilisées pour la génération d'un arbre, et plus précisément du flux de données (vertices et indices) sont en grande partie basées sur le papier « *Creation and Rendering of Realistic Trees* » écrit par Jason Weber et Joseph Penn et présenté lors du *Siggraph '95*.

Un arbre est présenté comme étant une structure composée de branches et de feuilles. Les branches représentent le tronc et n'importe quel autre niveau de branches. Les feuilles sont placées sur le dernier niveau de branche. Ces deux concepts sont détaillés aux sections 3.2 et 3.3.

Dans ce document, les paramètres permettant de décrire un arbre sont écrits en gras italique et doivent être interprétés de la manière suivante. Certains paramètres sont uniques et ne s'appliquent qu'à une partie précise de l'arbre, par exemple le paramètre ***Shape*** qui décrit la forme générale de l'arbre. D'autre paramètres sont plus généraux et s'appliquent à chaque niveau de récursion de l'arbre avec des valeurs différentes. Ces derniers sont présentés sous la forme ***nLength***, où le préfixe ***n*** représente le niveau de récursion sur lequel le paramètre est appliqué et ***Length*** le nom du paramètre. D'autres paramètres sont accompagnés d'un suffixe ***V***, comme ***nLengthV***, ce suffixe signale la présence d'une variation et la valeur donne l'amplitude de la variation appliquée au paramètre décrit. Les paramètres sont généralement des nombres positifs, cependant il peut parfois être nécessaire pour certains arbres d'avoir un comportement particulier, dans quel cas l'emploi d'une valeur négative active un mode spécial. Les valeurs d'angles sont toutes exprimées en degrés.

Dans ce chapitre, le processus de génération décrit permet d'avoir un aperçu du processus et ne se concentre que sur les parties implémentées. Pour plus de détails concernant les formules utilisées pour la génération, il est conseillé de se référer directement au document d'origine[4].

Des images d'arbres générés à l'aide de cet algorithme sont présentes au chapitre 10.6.

3.2 Branches

Une branche est une structure presque conique dont l'axe Z relatif coïncide avec son axe principal. Chaque branche a son système de coordonnées propre (coordonnée locale). Pour une branche principale dont l'axe Z est perpendiculaire au tronc, son axe Y sera parallèle à

[4] Disponible sur : http://portal.acm.org/citation.cfm?id=218427 et ainsi qu'en annexe.

l'axe Z du tronc. Un système de coordonnées dit « de la main droite » dont l'axe Z correspond à l'axe vertical est utilisé ici.

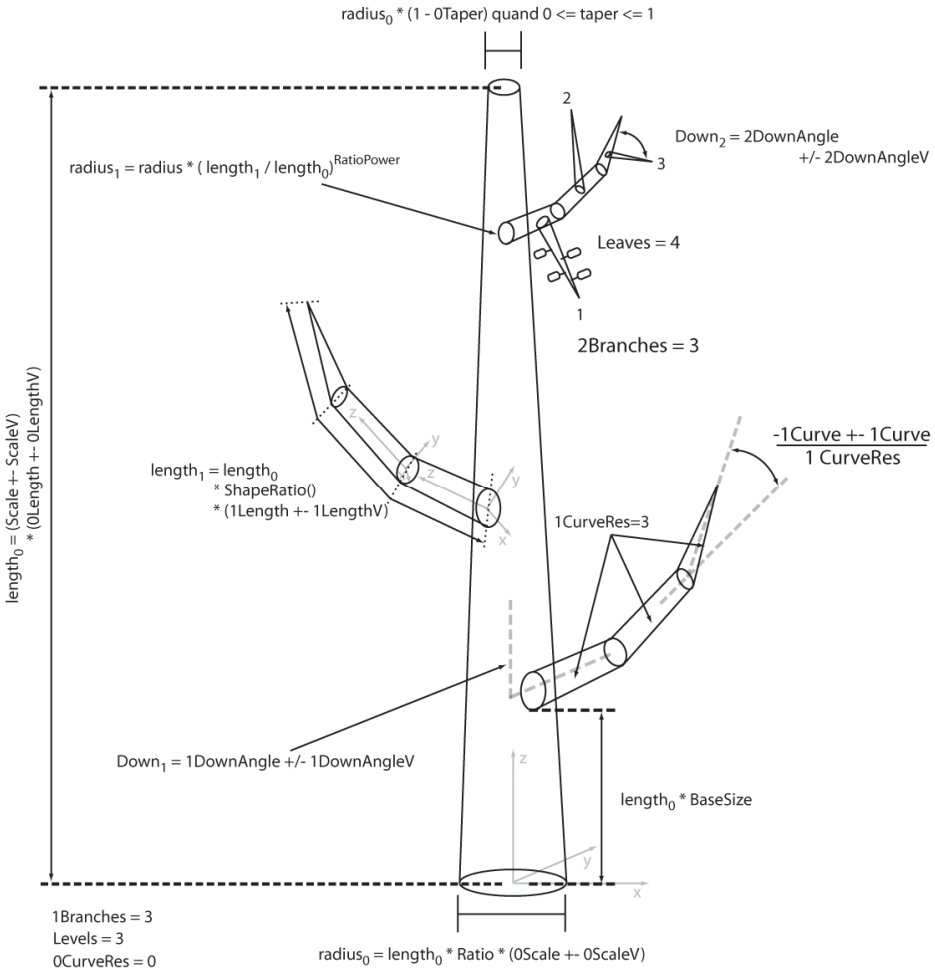


Figure 3.1 Description de plusieurs paramètres sur un arbre simple.

Généralement, trois niveaux de branches permettent de représenter la plupart des arbres, un quatrième niveau est cependant parfois utile. Le niveau 0 représente le tronc, le niveau 1, les branches principales, le niveau 2, les sous-branches et le niveau 3, les sous-sous-branches de l'arbre.

Une branche est composée de plusieurs segments presque cylindriques dont le nombre est définit par ***nCurveRes***. Ces segments sont stockés sous la forme d'une succession de sections presque circulaires, reliées entre elles pour dessiner des triangles et former les segments. Dans le cas où ***nCurveBack*** est égal à zéro, l'axe Z de chaque section est incliné de ***(nCurve / nCurveRes)*** degrés

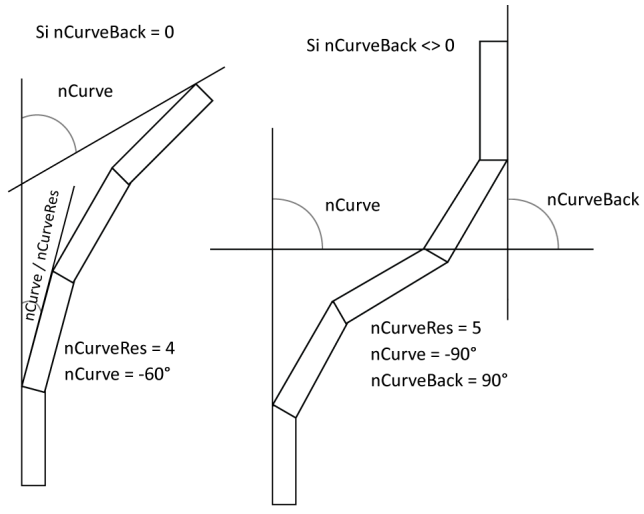


Figure 3.2 Influence de *nCurve*, *nCurveRes* et *nCurveBack* sur l'aspect d'une branche.

par rapport a la section précédente ce qui permet de donner un effet de courbe à la branche. Dans le cas où ***nCurveBack*** est différent de zéro, la première moitié des sections de la branche sont inclinés de ***(nCurve / (nCurveRes / 2))*** degrés, tandis que la seconde moitié l'est de ***(nCurveBack / (nCurveRes / 2))*** degrés. Cela permet d'obtenir des branches ayant une forme en « S » (voir Figure 3.2).

Une branche (parent) a aucune, une ou plusieurs branches filles (enfants). Le nombre de branches d'un certain niveau est représenté par le paramètre ***nBranches***, qui représente le nombre maximum de branches enfants possible. Ce nombre peut varier en fonction de la longueur de la branche parente ainsi que la position de cette dernière le long du tronc ou de sa propre branche parente. Le tronc à un maximum de branches filles défini par ***1Branches***, cependant ce nombre défini la densité de branche sur la longueur total du tronc. Le nombre réel de branches est influencé par le paramètre ***BaseSize***. Ce paramètre en plus de déterminer la hauteur à laquelle la première branche se situe, retire un certain pourcentage de branches de la base du tronc. Par exemple dans le cas d'un ***BaseSize*** de 0.2 et d'un ***1Branches*** de 50, 20% des branches ne seront pas affichées, ce qui correspond à un peu plus de 10 branches. Ce dernier point n'est pas présenté dans le document de référence, mais précisé sur le site internet de l'auteur[5].

La longueur maximale d'une branche à un certain niveau est définie par ***nLength +/- nLengthV***. Cette longueur maximale est définie comme une fraction de la longueur de la branche parente. Par exemple une branche fille ayant une longueur maximale de 0.3 et un parent de 10 mètre de long aura une longueur maximale de 3 mètres. Par la suite la longueur réelle de la branche dépendra de la forme de l'arbre ***Shape*** et de la position de la branche le long de sa branche parente.

Shape permet de définir neuf formes globales d'arbre différentes :

0. Conique
1. Sphérique
2. Hémisphérique
3. Cylindrique
4. Cylindre effilé
5. Flamme
6. Conique inverse
7. Tendance vers une flamme
8. Enveloppe (Non implémenté, voir 8.1.2)

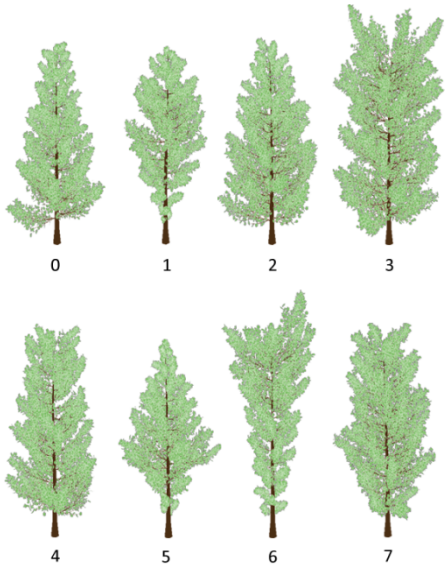


Figure 3.3 Les différents aspects que l'on peut obtenir pour un arbre à l'aide du paramètre *Shape*, à l'exception de l'enveloppe.

[5] http://www.imonk.com/baboon/trees/faq.html

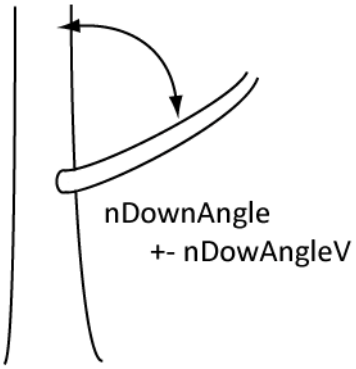


Figure 3.4 Effet de *nDownAngle*.

Par ailleurs, la position d'une branche sur son parent dépend aussi de son ***nDownAngle***. Ce paramètre définit l'angle avec lequel la branche s'incline sur l'axe X de la branche parente en s'éloignant de son axe Z. Dans le cas où ***nDownAngleV*** est négatif, l'inclinaison est répartie sur la longueur de la branche parente – les branches à la base seront plus inclinées vers le bas, tandis que les branches au sommet auront une inclinaison plus faible et pointeront vers la cime de l'arbre. La position dépend aussi de ***nRotate*** qui effectue une rotation de l'axe Z de la branche fille autour de l'axe Z de la branche parente relativement à l'angle de rotation de la branche fille précédente. Dans le cas où ***nRotate*** est négatif, cela active un mode spécial répartissant les branches de façon presque coplanaire en alternant les cotés de la branche. La rotation s'effectue relativement à l'axe Y de la branche parente.

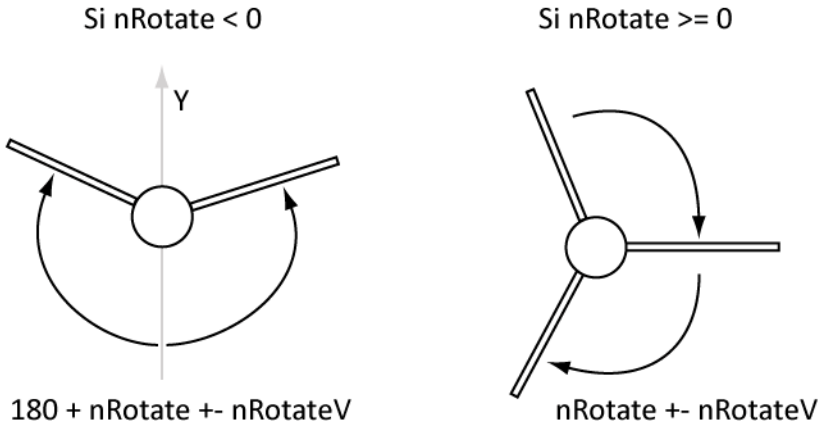


Figure 3.5 Effet de *nRotate* sur les branches filles autour de leur branche parente.

Une branche a un rayon de base qui est, pour le tronc, proportionnel à l'échelle du tronc (défini par ***Ratio*** et ***0Scale***). Le rayon des branches, dépend de sa longueur réelle et de la longueur de son parent, tout en étant bien sûr toujours plus faible que le rayon du parent à la position à laquelle la branche fille se trouve. Pour calculer le rayon d'une branche a un point de sa longueur, le paramètre ***nTaper*** est utilisé.

nTaper définit l'aspect global des branches selon la table suivante (voir Figure 3.6 aussi) :

nTaper	**Effet**
0	Cylindre ne se rétrécissant pas
1	Cylindre se rétrécissant en un point (cône)
2	Cylindre avec une extrémité sphérique.
3	Rétrécissement périodique (Sphères concaténées)

nTaper peut avoir des valeurs intermédiaires ce qui résultera dans des branches ayant un aspect entre ceux définit par l'entier supérieur et l'entier inférieur (Figure 3.6). En utilisant une valeur fractionnelle (entre 0 et 1) représentant une position de long de la branche, il est possible de déterminer exactement le rayon de la branche à ce niveau à l'aide de type de ***nTaper*** utilisé pour la branche.

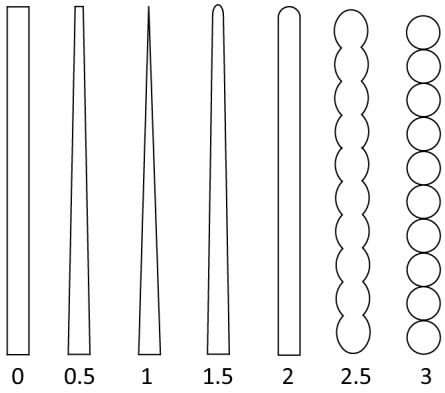


Figure 3.6 Effet des différentes valeurs de *nTaper* sur une branche.

En plus de ***nTaper*** qui a une influence sur le rayon du tronc et des branches, il existe deux autres paramètres supplémentaires ne s'appliquant qu'au tronc et permettant de faire varier son rayon. Il s'agit de ***Flare*** qui détermine une expansion exponentielle à la base du tronc (Figure 3.7). ***Lobes*** et ***LobeDepth*** quant à eux permettent de créer une variation sinusoïdale pour simuler une section transversale non-circulaire, le premier paramètre déterminant le nombre de *pointes* de cette variation et le second la magnitude de celles-ci. Il est déconseillé d'utiliser des nombres pairs pour les ***Lobes***, cela donne une impression de symétrie marqué (Figure 3.8).

Finalement, l'orientation des branches d'un niveau supérieur à 1 est aussi influencée par un paramètre d'attraction verticale ***AttractionUp*** simulant l'attraction des arbres pour la lumière. Une valeur positive fera pointer les branches vers le haut, tandis qu'une valeur négative les fera pointer vers le sol.

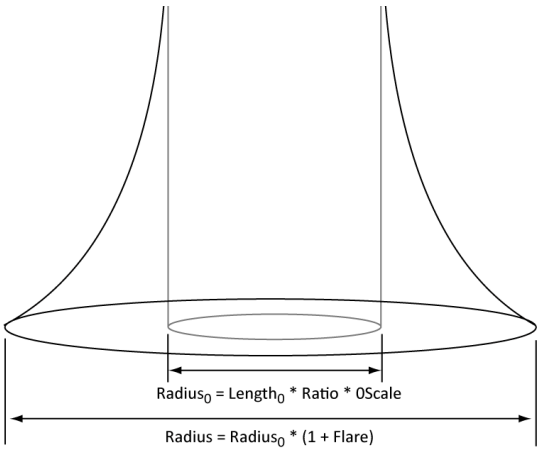


Figure 3.7 Effet du paramètre *Flare* sur le tronc.

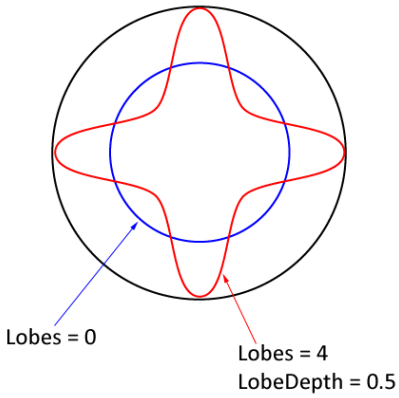


Figure 3.8 Effet des paramètres *Lobes* et *LobesDepth* sur le tronc.

3.3 Feuilles

Les feuilles représentent le dernier niveau de récursion d'un arbre. Si le nombre de feuilles est différent de 0, les feuilles utilisent les paramètres du dernier niveau de récursion (niveau trois si trois niveaux de branches sont utilisés, si quatre niveaux de branches sont utilisés, les feuilles utilisent les paramètres du troisième niveau de branches).

Un nombre positif de feuilles répartit les feuilles sur la longueur de la branche parente. Les paramètres ***nRotate*** et ***nDownAngle*** utilisés pour les feuilles ont la même influence que pour les branches (Figure 3.11).Dans le cas où le nombre de feuille est négatif, les feuilles sont placées en éventail à l'extrémité de la branche, l'angle ***nRotate*** définissant à ce moment non plus l'angle de rotation autour de la branche, mais l'angle sur lequel l'éventail de feuilles est ouvert (Figure 3.9).

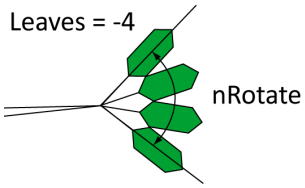


Figure 3.9 Effet d'un nombre négatif de feuilles.

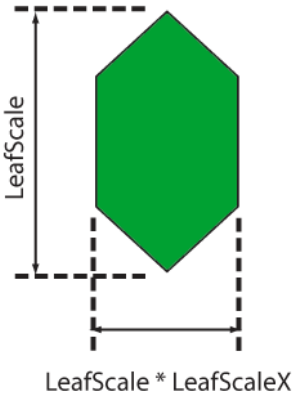


Figure 3.10 Paramètres décrivant la taille d'une feuille.

Le nombre de feuilles sur une branche dépend principalement de la longueur de cette dernière, de sa position le long de la branche parente et du nombre maximum de feuilles possibles sur une branche (***Leaves***), de plus un paramètre ***Quality*** supplémentaire permet de faire varier uniformément la couverture de feuilles sur l'arbre. ***LeafScale*** et ***LeafScaleX*** détermine la longueur et la largeur de la feuille.

LeafShape définit la forme de la feuille. Dans le cas de ce travail, les valeurs 0 (feuilles carrés) ou 1 (feuilles ovales) peuvent être utilisées. La feuille carrée est préférable car étant composée d'un nombre plus faible de triangles et de plus il est possible de lui appliquer une texture avec un effet de transparence afin de représenter aisément n'importe quelle forme de feuille.

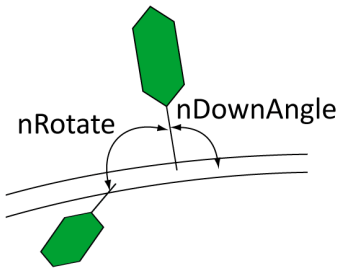


Figure 3.11 Influence de *nRotate* et *nDownAngle*

3.4 Dégradation à distance

Un arbre généré suivant cet algorithme est facilement composé de plusieurs milliers de triangles. Ce qui est idéal pour un rendu à courte distance. Cependant une telle précision n'est d'aucune utilité pour un arbre situé à plusieurs dizaines ou centaines de mètres de la caméra. Certaines branches ne sont plus discernables ou il n'est plus nécessaire d'afficher toutes les feuilles.

Le document de référence présente un algorithme de simplification basé sur un remplacement progressif des branches par des lignes et des feuilles par des points. Une ligne étant un objet faisant toujours 1 pixel de large à l'écran quelque soit sa distance à la caméra et reliant deux points. Un point est un objet de 1 pixel de coté à l'écran, quelque soit sa distance à la caméra. L'idée de base est de remplacer progressivement les feuilles par des points et les branches par

des lignes, tout en réduisant avec la distance le nombre de feuilles, branches, points et lignes affichés pour au final n'en afficher que le minimum. Cet algorithme est idéal d'un point de vue visuel car il présente une dégradation progressive et douce de l'arbre.

Après quelques essais dans cette direction, il s'est avéré deux choses. D'une part, l'usage de lignes et de points a un impact sur les performances trop important par rapport à l'emploi des polygones seuls. D'autre part, pour pouvoir implémenter cette méthode, il est nécessaire que chaque feuille et chaque branche soient des éléments indépendants de façon à pouvoir être masqué ou affiché à volonté. Le fait de diviser un *mesh* en un trop grand nombre de *sous-meshes* indépendants tue tout simplement les performances de l'application.

Il a donc été nécessaire de s'orienter vers d'autres méthodes moins pénalisantes en termes de performance. La méthode utilisée est basée sur l'usage de *billboards* et d'*impostors*.

L'arbre final est composé de plusieurs *sous-meshes* - un par niveau de branches et les feuilles, soit un maximum de cinq. Il est possible de masquer ou d'afficher à volonté ces *sous-meshes*. De cette manière, dès que la distance entre la caméra et l'arbre devienne suffisamment grande pour que masquer un niveau de branche se remarque peu, il soit possible de le faire.

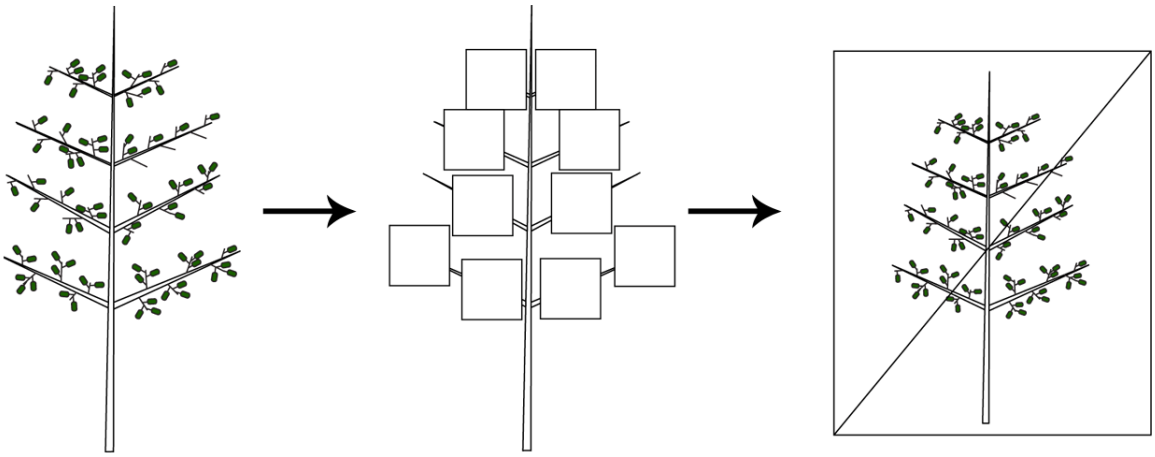


Figure 3.12 Les différentes étapes de la dégradation à distance. A gauche, l'arbre complet avec tous ses détails. Au centre le deuxième niveau de branches et les feuilles sont masqués et des *billboards* sont placés sur le premier niveau de branches pour remplacer les feuilles. A droite, un *impostor* de l'arbre est utilisé pour remplacer le tout, la géométrie d'origine de l'arbre n'est plus visible.

Par ailleurs, pour remplacer les feuilles d'origine, un ensemble de « feuilles » qui sont en fait des *billboards* représentant des groupes de feuilles sont générés. L'intérêt du *billboard* est ici évident, du fait qu'il soit toujours orienté en direction de la caméra. L'utilisateur a ainsi l'impression d'avoir un arbre dont la couverture en feuilles est homogène sur toute sa surface et ce quelque soit la direction dans laquelle il le regarde. De plus, comme les *billboards* ne sont composés que de deux triangles et ne sont placés que sur le premier niveau de branches, l'économie en terme de triangles est importante, les feuilles représentant la plus grande partie des triangles de l'arbre d'origine.

Passé une certaine distance, l'arbre est lui-même remplacé par un *impostor*, ce qui dérange peu. L'arbre est à ce moment situé suffisamment loin pour que la différence entre l'arbre d'origine et l'*impostor* soit faible (Voir aussi Figure 10.4 en page 55).

Toutes ces informations sont créées en même temps que l'arbre lui-même est créé.

3.5 Liste des paramètres

Ci-dessous, une liste complète de paramètres présentés par Weber et Penn dans leur article ainsi que de leur description.

Paramètre	**Description**
Shape	Forme général de l'arbre
BaseSize	Zone fractionnelle sans branche à la base de l'arbre
Scale, ScaleV, ZScale, ZScaleV	Taille et échelle de l'arbre
Levels	Niveaux de récursion
Ratio, RatioPower	Ratio Rayon/Longueur, réduction
Lobes, LobeDepth	Variation sinusoïdale de la section transversale
Flare	Expansion exponentielle à la base du tronc
0Scale, 0ScaleV	Echelle du tronc et variation
0Length, 0LengthV, 0Taper	Longueur relative, échelle de la section transversale
0BaseSplits	Division des branches à la base du tronc
0SegSplits, 0SplitAngle, 0SplitAngleV	Division de la branche et angle par segments
0CurveRes, 0Curve, 0CurveBack, 0CurveV	Résolution de la courbure et angles
1DownAngle, 1DownAngleV	Branche principale : angle à partir du parent
1Rotate, 1RotateV, 1Branches	Angle de rotation, Nombre de branche
1Length, 1LengthV, 1Taper	Longueur relative, échelle de la section transversale
1SegSplits, 1SplitAngle, 1SplitAngleV	Division de la branche par segment
1CurveRes, 1Curve, 1CurveBack, 1CurveV	Résolution de la courbure et angles
2DownAngle, 2DownAngleV	Seconde branche : angle à partir du parent
2Rotate, 2RotateV, 2Branches	Angle de rotation, Nombre de branche
2Length, 2LengthV, 2Taper	Longueur relative, échelle de la section transversale
2SegSplits, 2SplitAngle, 2SplitAngleV	Division de la branche par segments
2CurveRes, Curve, 2CurveBack, 2CurveV	Résolution de la courbure et angles
3DownAngle, 3DownAangleV	Troisième branche : angle à partir du parent
3Rotate, 3RotateV, 3Branches	Angle de rotation, Nombre de branche
3Length, 3LengthV, 3Taper	Longueur relative, échelle de la section transversale
3SegSplits, 3SplitAngle, 3SplitAngleV	Division de la branche par segment
3CurveRes, 3Curve, 3CurveBack, 3CurveV	Résolution de la courbure et angles
Leaves, LeafShape	Nombre de feuille par parent, ID de la forme
LeafScale, LeafScaleX	Longueur des feuilles, échelle relative de l'axe X
AttractionUp	Tendance de croissance vers le haut
PruneRatio	Effet fractionnaire du *Pruning*
PruneWidth, PruneWidthPeak	Largeur, Position de la cime de l'enveloppe
PrunePowerLow, PrunePowerHigh	Courbe de l'enveloppe

En plus de ces paramètres, il a été nécessaire d'en rajouter plusieurs pour affiner au mieux la génération de l'arbre en lui-même. Ces paramètres supplémentaires sont les suivants :

Paramètre	**Description**
MeshQuality	Qualité du *mesh* généré (utilisé pour la création du tronc).
Quality	Qualité du feuillage, utilisé par Weber et Penn, mais non listé.
StemColour	Vertex Colour utilisé pour les branches.
LeafColour	Vertex Colour utilisé pour les feuilles.
StemMat	*Material* utilisé pour les branches.
LeafMat	*Material* utilisé par les feuilles.
LowResLeafMat	*Material* utilisé par les feuilles basses qualités.
0MaxVertices	Nombre maximum de vertices utilisé pour la création d'une section du tronc.
1MaxVertices	Nombre maximum de vertices utilisé pour la création d'une section du premier niveau de branches.
2MaxVertices	Nombre maximum de vertices utilisé pour la création d'une section du second niveau de branches.
3MaxVertices	Nombre maximum de vertices utilisé pour la création d'une section du troisième niveau de branches.
Level0MaxDistance	Distance maximale à laquelle le tronc est visible.
Level1MaxDistance	Distance maximale à laquelle le premier niveau de branches est visible.
Level2MaxDistance	Distance maximale à laquelle le second niveau de branches est visible.
Level3MaxDistance	Distance maximale à laquelle le troisième niveau de branches est visible.
LeafMaxDistance	Distance maximale à laquelle les feuilles sont visibles avant d'être remplacé par les feuilles de plus basse qualité.
LowResLeaves	Nombre de feuilles par parent.
LowResLeafScale	Longueur des feuilles.
LowResLeafScaleX	Echelle relative de l'axe X.
Bend	Orientation des feuilles (valeur entre 0 et 1).

Chapitre 4

Librairie de génération

4.1 Introduction

La librairie de génération est un libraire DLL servant à générer un flux de données (*Index* et *Vertex Buffers*) qui peut être interprété par un moteur de rendu. L'utilisateur doit cependant implémenter l'interprétation de ce flux de données pour le moteur de rendu dans lequel il souhaite utiliser cette librairie.

4.2 Guide d'utilisation

Ce guide d'utilisation décrit la façon d'utiliser la librairie de génération dans le cadre d'un projet personnel.

4.2.1 Génération du *mesh* d'arbre

Dans cette section, seule la logique générale de la création d'un *mesh* est décrite, cette démarche est cependant fortement appuyée sur Ogre3D. Pour un exemple détaillé d'une implémentation avec Ogre3D, il est préférable de se référer aux annexes qui contiennent le code exemple commenté.

Pour utiliser le namespace de la librairie :

```
using namespace Flora;
```

La première étape consiste à créer un objet contenant les paramètres et à le remplir, soit manuellement, soit en chargent un fichier binaire contenant les informations préalablement crées via *Flora Studio* (voir Chapitre 5).

```
Parameter* pParam = new Parameter();
pParam->LoadFromFile("C:\quakingAspen.fsf");
```

L'étape suivante consiste en créer un nouvel arbre et à le générer.

```
Tree* pTree = new Tree(pParam, 1);
```

Il est nécessaire de passer au constructeur de l'arbre l'objet contenant les paramètres de génération ainsi qu'une « graine » (*Seed*). Cette graine est utilisée pour la génération de nombre aléatoire afin de donner un aspect légèrement différent à chaque arbre que nous créons (voir 4.3 Technique). La génération se fait automatiquement lors de l'appel du constructeur.

A partir de ce moment, l'objet pointé par `pTree` contient toutes les informations nécessaires à la création d'un *mesh*, il faut maintenant récupérer ces informations, les interpréter et les afficher dans le moteur de rendu.

La première étape consiste à créer un pointeur vers un *mesh* créé manuellement (Ogre 3D).

L'étape suivante consiste en définir le format des vertices qui sera utilisé. La libraire de génération fournit les informations suivantes pour chaque vertex :

- Coordonnées dans l'espace (Position) – X, Y, Z (trois float)
- Normal – X, Y, Z (trois float)
- Coordonnée de mapping – U, V (deux float)
- Couleur – RGB (un unsigned int)

La taille du vertex buffer est connu en appelant la méthode `getVerticesCount()` de `pTree` qui retourne le nombre de vertices créés.

Le vertex buffer et colour buffer doivent être créés séparemment. Les coordonnées dans l'espace, les normales et les coordonées de mapping peuvent être représentés par des float, tandis que les couleurs sont représentées par un unsigned int.

Désormais les flux de vertices et de couleurs sont prêts à être récupérés. Il existe deux solutions pour y parvenir. La première, qui est aussi la plus simple consiste a appeler directement la méthode de `pTree` :

```
void createMesh(float **pVertexArray, unsigned int **pColourArray,
      unsigned long **pIndexArray)
```

qui demande de passer l'index buffer avec le vertex et le colour buffer. Cette méthode permet de récupérer un arbre entier composé d'un seul *mesh*. Cependant ce n'est pas exactement ce qui est désiré puisque l'arbre doit être divisé en *sous-meshes*, par niveau de récursion, afin de pouvoir adapter selon la distance la complexité de ce qui est affiché.

Dans ce but, le vertex buffer doit être rempli par niveau de l'arbre. Cette façon de faire est moins efficace et légèrement plus gourmande en mémoire, mais elle permet de construire le vertex buffer de manière à pouvoir l'utiliser ultérieurement avec l'index buffer :

```
void createVerticesStream(float **pVertexArray, unsigned int
      **pColourArray)
```

Après avoir récupéré le contenu du vertex buffer, il faut donc créer un *sous-mesh* par niveau de l'arbre en appellant la méthode permettant de récupérer le flux d'indices pour chaque niveau de l'arbre (le nombre total de niveaux de l'arbre est obtenu avec `pParam->m_Levels`)

```
void createIndicesStream(unsigned long **pIndexArray, int level)
```

Pour construire l'index buffer, sa taille doit être donnée. La méthode `countFaces(int level)` compte le nombre de faces (triangles) dont est composées un niveau de l'arbre (ou

de tout l'arbre en utilisant le paramètre « -1 » comme niveau). Cependant vu que trois vertices sont utilisés pour définir un triangle, il est nécessaire de multiplier par trois le nombre de faces obtenues pour créer le buffer. En dernière étape, un *material* est attribué au *sous-mesh*, soit en donnant sa référence manuellement soit en utilisant `pParam->m_StemMat` si une valeur lui a été attribuée.

Sous certaines conditions, la méthode `countFaces(int level)` retourne une valeur nulle. Par exemple si le nombre de niveaux de l'arbre est de trois, mais qu'il n'y a aucune branche sur les niveaux un et deux. Dans ce cas il est utile de tester le nombre de faces au préalable, d'interrompre la création des *sous-meshes* et de ne pas créer le *sous-mesh* contenant les feuilles – les vertices des feuilles n'ayant pas été créées, celles-ci étant placées sur le dernier niveau défini par `pParam->m_Levels`. Pour un exemple complet, il est conseillé de se référer au code en annexe.

L'index buffer des feuilles est créé en appellant la méthode :

```
void createLeavesIndicesStream(unsigned long **pIndexArray)
```

Le compte des faces des feuilles s'effectue à l'aide la méthode :

```
unsigned long countLeavesFaces()
```

Le reste est identique aux branches, si ce n'est que nous ne ne créons qu'un seul *sous-mesh* et que ce dernier n'est construit que si l'arbre dispose de feuilles (`pParam->m_Leaves != 0`, un nombre négatif de feuilles pouvant être utilisé comme paramètre).

Viens ensuite la création des *billboards* servant aux feuilles de basse qualité. La méthode :

```
void createLowResMesh(float **pVertexArray, unsigned int
      **pColourArray, bool recursive)
```

permet de recupérer toutes les informations nécessaires à la création des *billboards*, soit :

- Coordonnées dans l'espace – X, Y, Z (trois float)
- Dimensions – X, Y (deux float)

La méthode `unsigned long countLowResLeaves()` compte le nombre de feuilles de basse qualité (*billboards*) présente sur l'arbre.

Dernière étape de la génération du *mesh*, il s'agit de définir la *bounding box* du mesh. Les méthodes :

`float getMinX(),float getMinY(),float getMinZ(),float getMaxX(),float getMaxY()` et `float getMaxZ()`

permettent d'obtenir les valeurs des points extrêmes de l'arbre qui sont utilisées dans ce but.

Notons finalement, que l'arbre a été généré en utilisant l'axe Z comme étant l'axe vertical. Il sera peut-être nécessaire de lui faire faire une rotation si l'axe vertical du moteur de rendu est un autre axe.

4.2.2 Dégradation à distance

La manière la plus simple d'utiliser la dégradation à distance se fait à l'aide de l'*observer pattern*.

L'*observer pattern* est un *design pattern* observant l'état d'un sujet et en cas de modification de ce sujet, invoque une mise à jour de tous les éléments enregistrés auprès de celui-ci. Dans le cas d'Ogre3D ce sujet est un *FrameListener*, une interface recevant des notifications sur les événements à chaque image (*frame*). En cas d'un événement lors d'un *frame*, par exemple la caméra est déplacée par un mouvement de souris ou par l'usage du clavier, le sujet invoque un update sur tous les objets qui sont enregistrés (dans notre cas, les arbres).

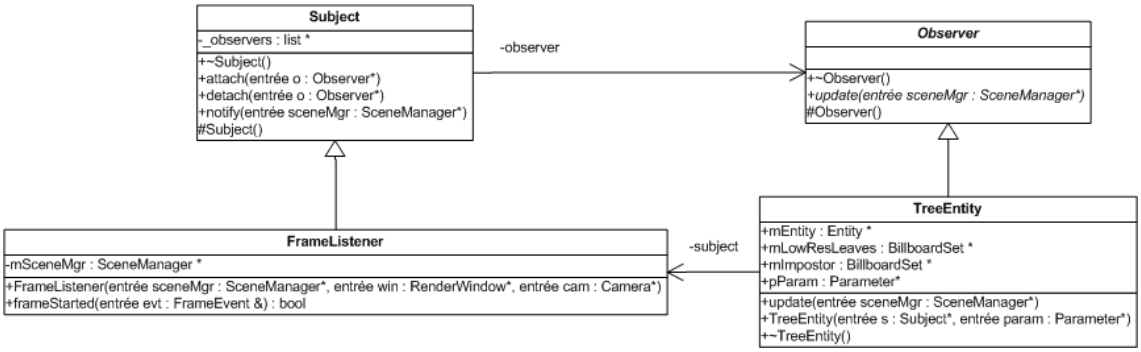


Figure 4.1 Diagramme de classe de l'implémentation de l'*observer pattern* dans l'exemple sur Ogre3D.

Il est évident que ce n'est pas une technique particulièrement efficace dans le cas où si un grand nombre d'arbres sont présents sur une scène. La mise à jour s'effectue sur tous les arbres, y compris ceux qui ne sont pas visibles, des opérations inutiles sont donc effectuées. Il serait utile ici d'y ajouter une notion de pagination (voir section 8.1.4) ne permettant de travailler que sur les arbres visibles par l'utilisateur à ce moment.

Une fois la mise à jour invoquée, l'application compare pour chaque arbre sa distance à la caméra et modifie son apparence en fonction de celle-ci en affichant ou masquant les différents *sous-meshes* qui ont été construis durant la génération du *mesh* de l'arbre.

Par ailleurs, et pour assurer de bonne performances, il est préférable à partir d'une certaine distance de remplacer le *mesh* de l'arbre par un *impostor*. Les dimensions du *billboard* servant d'*impostor* peuvent être déterminés à l'aide de la méthode `getMaxZ()` si l'on part du principe que l'arbre est plus haut que large. L'*impostor* est fait à partir d'un *Render To Texture* standard de l'arbre.

Un exemple simple d'une implémentation sur Ogre3D est disponible dans les annexes.

4.3 Technique

Cette partie décrit la façon dont la librairie de génération a été implémentée.

4.3.1 Diagramme de classe

Le diagramme de classe présenté ci-contre est une version simplifié sans le détail de chaque classe, afin d'avoir une vue d'ensemble de la librairie.

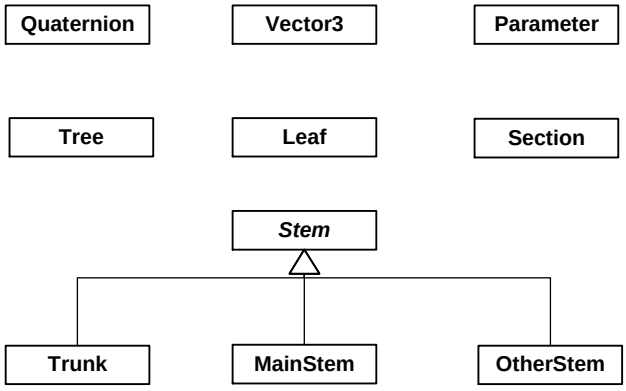


Figure 4.2 Diagramme de classe.

Quaternion : Classe implémentant les fonctions nécessaires à l'usage des quaternions.

Vector3 : Classe implémentant un vecteur de trois float et toutes les opérations liées nécessaires.

Parameter : Classe contenant la liste des paramètres d'un arbre, ainsi que les méthodes permettant de lire et d'écrire des fichiers de paramètres.

Tree : Classe contenant l'implémentation de base d'un arbre et servant à lancer sa génération.

Leaf : Classe contenant l'implémentation des feuilles et les méthodes nécessaires à leur génération.

Section : Classe contenant l'implémentation des sections d'une branche et les méthodes nécessaires à leur génération.

Stem : Classe abstraite contenant l'implémentation des branches et de leur génération complète.

Trunk : Classe héritant de *Stem* et contenant l'implémentation spécifique de certaines fonctions liées au tronc.

MainStem : Classe héritant de *Stem* et contenant l'implémentation spécifique de certaines fonctions liées aux branches principales.

OtherStem : Classe héritant de *Stem* et contenant l'implémentation spécifique de certaines fonctions liées aux autres branches de l'arbre.

4.3.2 Fonctionnement logique de l'application

La génération de la structure de l'arbre s'effectue en deux parties. Dans un premier temps la structure de l'arbre est créée avec toutes ses branches et toutes ses feuilles. Cette première phase permet aussi de calculer un certain nombre de paramètres qui seront nécessaire pour la suite de la génération. La deuxième phase consiste à créer la structure de chaque composant de l'arbre et à les positionner.

Création de la structure de l'arbre

Comme il l'a été présenté dans le guide d'utilisation, un des éléments passé au constructeur de l'arbre est une graine (*Seed*). Cette graine permet d'initialiser un générateur de nombre aléatoire. Ce procédé permet de créer une grande quantité de variation d'un même arbre en se basant sur l'amplitude donnée pour certains les paramètres. Pour chaque paramètre ayant une variation, un nombre aléatoire entre 0 et la valeur de l'amplitude est tiré. Le signe avec lequel l'amplitude sera appliquée est lui aussi choisi aléatoirement. En réutilisant la même graine et les mêmes paramètres, exactement le même arbre est généré à chaque fois, une même graine générant à chaque utilisation une même suite de nombre aléatoires.

Lors de l'instanciation d'un objet « Tree », un tronc est créé. Le tronc est le point de départ de l'arbre qui est ensuite généré récursivement en profondeur. Durant cette phase, pour chaque branche, sont calculés :

- Sa longueur maximale – permet de déterminer sa longueur réelle.
- Sa distance par rapport à la base de la branche parente en mètre (offset de la branche) – utilisée pour calculer sa longueur réelle.
- Sa longueur réelle, en fonction sa longueur maximale, de la longueur de sa branche parente et de sa position le long de sa branche parente.
- Son rayon de base.
- Le nombre de branches filles.
- Son inclinaison par rapport à sa branche parente.
- Le nombre de feuilles que la branche aura.
- Le nombre de *billboards* (feuille de basse qualité) sur cette même branche.

La plupart de ces paramètres se calculent différemment selon si la branche est le tronc, une branche principale ou une autre branche. Une fois ces paramètres définis, les branches filles, les feuilles et les feuilles de basse qualité sont crées récursivement. A ce stade, il ne s'agit que de créer l'objet. Pour les feuilles, sa position relative le long de la branche parente est calculée.

Création de l'arbre

Pour la seconde phase, le point de départ est aussi le tronc, l'arbre est ensuite construit récursivement.

La première étape consiste à créées les sections des branches.

Toutes les branches ont leur origine au point (0.0, 0.0, 0.0) et sont créées à partir de ce point. Dans le cas où la branche est le tronc, il faut tenir compte du paramètre ***Flare***. La présence d'un ***Flare*** va subdiviser le premier segment du tronc en autant de sous-segments qu'il y en a sur le reste de sa longueur. Par exemple si le tronc a un ***0CurveRes*** = 4, le premier segment sera composé de quatre sous-segments lui aussi. Vu que ***Flare*** décrit une croissance

exponentielle, les changements seront plus importants vers la base du tronc. Pour cette raison, le premier segment du tronc n'est pas subdivisé en quatre sous-segments de longueur équivalente. La première section après le point (0.0, 0.0, 0.0) sera placé à une hauteur de ($longueur_{section}$ / $2^{(nCurveRes - 1)}$), pour chaque section supplémentaire, la distance par rapport au point d'origine est doublé. Par exemple, si la hauteur du segment du tronc est égale à 1, les sous-segments seront placés aux hauteurs : 0.0, 0.125, 0.25 et 0.5 (Figure 4.3). Le calcule de ***Flare*** donne une valeur supérieur ou égale à 1 se multipliant au rayon de base du tronc. Si ***Flare*** donne une valeur inférieure à 1, celle-ci est arrondie à 1 de sorte à ne pas avoir de segments de bases ayant un rayon inférieur aux segments suivants. Les sous-segments placés ici sont à la verticale de l'origine et ne sont pas influencés par la courbure du tronc si celui-ci en a une.

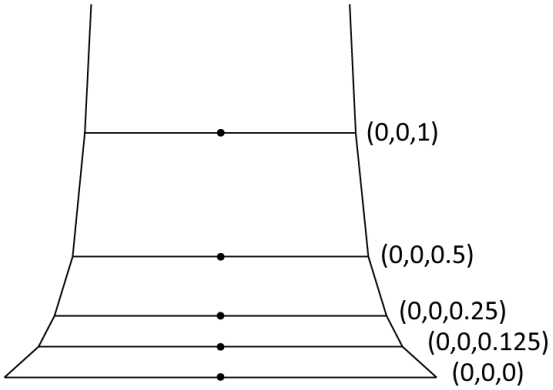
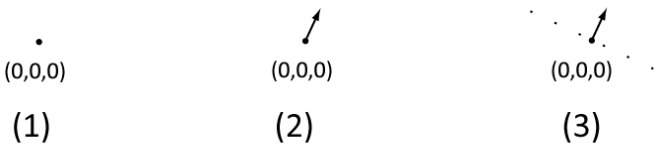


Figure 4.3 Tronc sur lequel *Flare* est appliqué, *0CurveRes* vaut 4.

Dans le cas d'une autre branche que le tronc ou si la valeur de ***Flare*** est nulle, le paramètre ***Flare*** n'a aucune influence. Dans ce cas la première section est placée au point (0.0, 0.0, 0.0) (1). Pour placer la seconde section de la branche, il faut connaître d'une part la longueur d'un segment du tronc, d'autre part de l'orientation de la section précédente. L'orientation d'une section est définie par un quaternion. Par défaut le quaternion de direction est un quaternion identité pour le tronc (ne correspond à aucune direction). Pour les branches suivantes, il dépendra du quaternion défini lorsque la branche est positionnée le long de sa branche parente (voir plus bas) (2). Après quoi la section est créée (3).

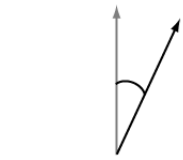


Dans un premier temps un vecteur (0.0, 0.0, Z) est définit (4). Dans ce vecteur, Z représente la longueur d'un segment de la branche ($longueur_{branche}$ / ***nCurveRes***). Ce vecteur est ensuite multiplié par le quaternion de direction de la section précédente. Cette multiplication retourne un vecteur orienté dans la direction donnée par le quaternion (5). La valeur de ce nouveau

vecteur est additionnée aux coordonnées du point précédent (6). De la sorte le point d'origine du segment suivant est placé.

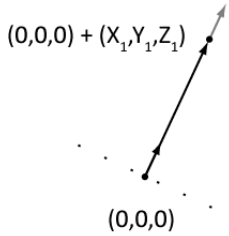
Vecteur (0,0,Z)

(4)



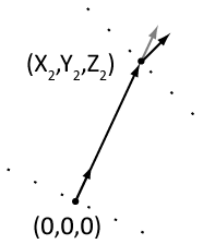
Vecteur orienté (X_1,Y_1,Z_1)

(5)



(6)

Le calcul du quaternion d'orientation permettant de placer les segments de la branche s'effectue à partir du quaternion d'orientation globale de la branche (voir plus bas), de la courbe de la branche définie par ***nCurve*** et ***nCurveBack*** et pour les sous-sous-branches aussi à partir du paramètre ***AttractionUp***. Pour chaque section, le quaternion d'orientation de la section précédente est multiplié au quaternion nouvellement créé, ce qui donne l'orientation de la section courante (7). La première section utilise le quaternion d'orientation globale de la branche comme quaternion d'orientation de base. Le premier segment n'est pas non plus influencé par la courbure de la branche.



(7)

Multiplier deux quaternions revient à additionner leurs effets. Les quaternions ont aussi leurs avantages ici. Il est possible de créer un quaternion à partir d'un angle (exprimé en radian) et d'un axe (vecteur), ce qui revient à effectuer une rotation d'un certain angle autour d'un axe. Par contre il est important de tenir compte de l'ordre dans lesquelles les multiplications sont effectuées, l'algèbre des quaternions n'étant pas commutative.

Ces opérations sont effectuées pour chaque section de la branche.

La création d'une section est une procédure relativement simple dans laquelle les vertices sont placés autour de l'origine de la section. Dans le cas du tronc, il faut aussi tenir compte du paramètre ***Lobes*** qui a un impact sur le rayon. L'impact de ce paramètre est de plus différent pour chaque vertex composant une section du tronc. Une fois le vertex créé, celui-ci est encore multiplié avec le quaternion de direction de la section afin d'orienter la section dans la bonne direction.

Une fois les sous-sections de la branche créées, les branches filles sont positionnées le long de leur branche parente. La première étape consiste à déterminer sur quel segment de la branche parente, la branche fille se trouve. Ce calcul est effectué en se basant sur l'offset de la branche fille. Ici, il est nécessaire de faire attention, dans le cas du tronc, de la présence d'un paramètre ***Flare*** qui ajoute des segments supplémentaires. Ensuite deux quaternions d'orientations sont déterminés. Il s'agit de quaternions basé sur ***nRotate*** et ***nDownAngle***. Ceux-ci sont multipliés entre eux pour déterminer l'orientation finale de la branche autour de sa branche parente. La position exacte le long de la branche parente est déterminée à l'aide du quaternion d'orientation de la section précédant le segment sur lequel la branche fille est placée et de la distance de cette section à la branche.

Une fois les branches placées, la même opération est effectuée pour les feuilles. Le positionnement d'une feuille est similaire à peu de chose près au positionnement d'une branche. La création de ses vertices se fait selon les deux types de feuilles prédéfinis disponibles, soit une feuille carrée (quatre vertices), soit une feuille ovale (six vertices). Ces deux types de feuilles sont en dur dans le code. Le choix d'un type de feuille remplit le tableau de vertices de la feuille avec les vertices du type demandé (voir aussi Figure 4.4). Durant la création des vertices de la feuille, un vecteur perpendiculaire à celle-ci est crée. Il servira de base pour la normale des vertices de la feuille par la suite. Les vertices et le vecteur normal sont ensuite orientés selon le quaternion d'orientation de la feuille.

Le positionnement des *billboards* est plus simple que le positionnement des feuilles vu qu'il suffit de placer l'origine du *billboard* le long de sa branche parente, ils ne dépendent pas de ***nDownAngle*** et ***nRotate***. La procédure utilisée reste cependant la même.

A partir de ce point, l'arbre est complet et il ne reste plus qu'à générer les vertex, colour et index buffers.

Génération du flux de vertices et d'indices

La première étape consiste à générer le flux de vertices des branches. Pour cela, chaque branche de l'arbre est parcourue. Sur chaque branche, le vecteur contenant les sections est parcouru lui aussi. Pour chaque section, son vecteur de vertices est à son tour parcouru.

Tout d'abord la normale du vertex est calculée, ensuite sa position globale en additionnant la position locale (par rapport à l'origine de sa branche) à la position de l'origine de la branche.

Une fois ces calculs effectués, les coordonnées sont ajoutées dans le tableau de vertices passé en argument à la méthode, suivi des normales et enfin des coordonnées de mapping. La couleur du vertex est elle aussi ajoutée à son tableau de couleur.

Après avoir créé le vertex, les coordonnées dans l'espace du point sont testées afin de déterminer les extrêmes de l'arbre. Ces extrêmes seront nécessaires plus tard au calcul de la *bounding box* et aux dimensions de l'*impostor*.

La génération des vertices des feuilles est similaire à celle des branches. La principale différence se situe dans la manière de calculer la normale, celle-ci étant calculée lors de la création de la feuille, et dans la manière de calculer les coordonnées de mapping. Les

coordonnées de mapping sont calculées à partir des coordonnées de base des vertices d'une feuille avant leur orientation.

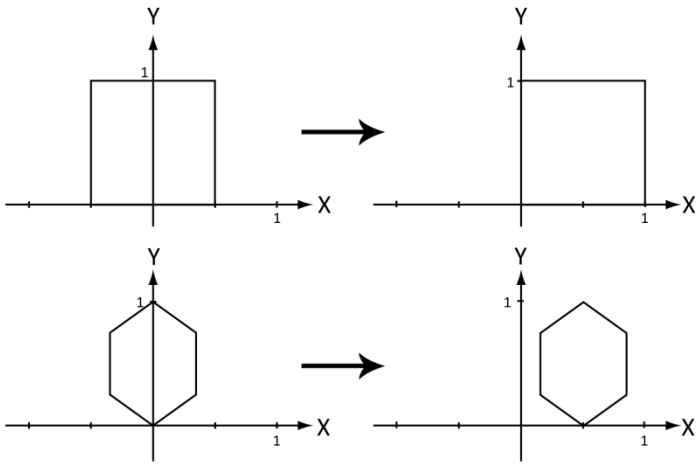


Figure 4.4 Définition des coordonnées de mapping des feuilles. A gauche, les coordonnées de bases des vertices de chaque type de feuille. A droite, après un décalage de +0.5 sur l'axe X, on obtient les coordonnées de mapping.

De plus, vu qu'une feuille dispose de deux côtés, il est nécessaire de recalculer une nouvelle série de vertices pour la face inférieure, en prenant l'inverse de la normale de la face supérieure afin d'avoir une meilleure gestion de la lumière.

Pour la génération du flux d'indices, les triangles des branches sont créés deux à deux. De plus pour générer le flux d'indices nous avons besoin d'une valeur d'offset. L'offset permet de « sauter » les vertices des sections qui ont déjà placées. Le nombre de faces généré correspond au nombre de vertices d'une section d'une branche multiplié par deux. Le nombre de vertices utilisé pour créer une section est donc utilisé comme base.

```
for(unsigned long j = 0 ; j < verticesCount ; j++)
{
	//First Tri
	*(*pIndexArray)++ = offset + (j + 1) % verticesCount;
	*(*pIndexArray)++ = offset + j;
	*(*pIndexArray)++ = offset + j + verticesCount;

	//Second Tri
	*(*pIndexArray)++ = offset + (j + 1) % verticesCount;
	*(*pIndexArray)++ = offset + j + verticesCount;
	*(*pIndexArray)++ = offset + (j + 1) % verticesCount +
verticesCount;
}
```

`offset` correspond à l'offset utilisé pour la branche précédente (`pOffset`), additionné aux nombres de vertices déjà placé sur la branche courante :

```
offset = *pOffset + segment * verticesCount;
```

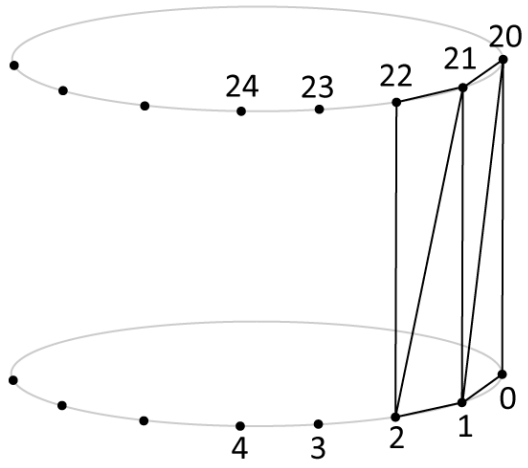


Figure 4.5 Création de triangles sur une branche. Les triangles affichés sont créés dans l'ordre suivant : [1 – 0 – 20 / 1 – 20 – 21] [2 – 1 – 21 / 2 – 21 – 22]

Une fois les indices créés pour la branche, `pOffset` est incrémenté de nombre de vertices de la branche :

```
*pOffset += verticesCount * (curveRes + 1);
```

Pour la génération des faces des feuilles, les faces inférieures et supérieures des feuilles sont générées en deux temps. L'orientation de la face dépend du sens dans lequel sont construites les faces.

```
//verticesCount - 2 = nombre de faces
for(unsigned short j = 0 ; j < verticesCount - 2 ; j++)
{
	//Downside
	*(*pIndexArray)++ = *pOffset;
	*(*pIndexArray)++ = *pOffset + j + 1;
	*(*pIndexArray)++ = *pOffset + j + 2;
}

*pOffset += verticesCount;

//verticesCount - 2 = nombre de faces
for(unsigned short j = 0 ; j < verticesCount - 2 ; j++)
{
	//Upside
	*(*pIndexArray)++ = *pOffset;
	*(*pIndexArray)++ = *pOffset + verticesCount - j - 1;
	*(*pIndexArray)++ = *pOffset + verticesCount - j - 2;
}

*pOffset += verticesCount;
```

La création du flux de données pour les *billboards* est plus simple vu qu'il s'agit uniquement de retourner dans un tableau les coordonnées de feuilles, suivi de deux dimensions pour le *billboard* (calculées à partir de ***LowResLeafScale*** et ***LowResLeafScaleX***). La couleur utilisée est la même que celle utilisée pour les feuilles.

Chapitre 5

Editeur visuel (*Flora Studio*)

5.1 Introduction

L'éditeur visuel est une application Win32 basée sur wxWidgets embarquant une fenêtre de rendu Ogre3D pour pré-visualiser les modifications apportées à l'arbre.

La fenêtre est découpée en deux parties, sur la droite la fenêtre de rendu et sur la gauche le menu permettant de gérer les différents paramètres de l'arbre ainsi que de pré visualiser les LODs.

Il est possible de sauvegarder les paramètres utilisées pour créer l'arbre ainsi que de charger un fichier de paramètres.

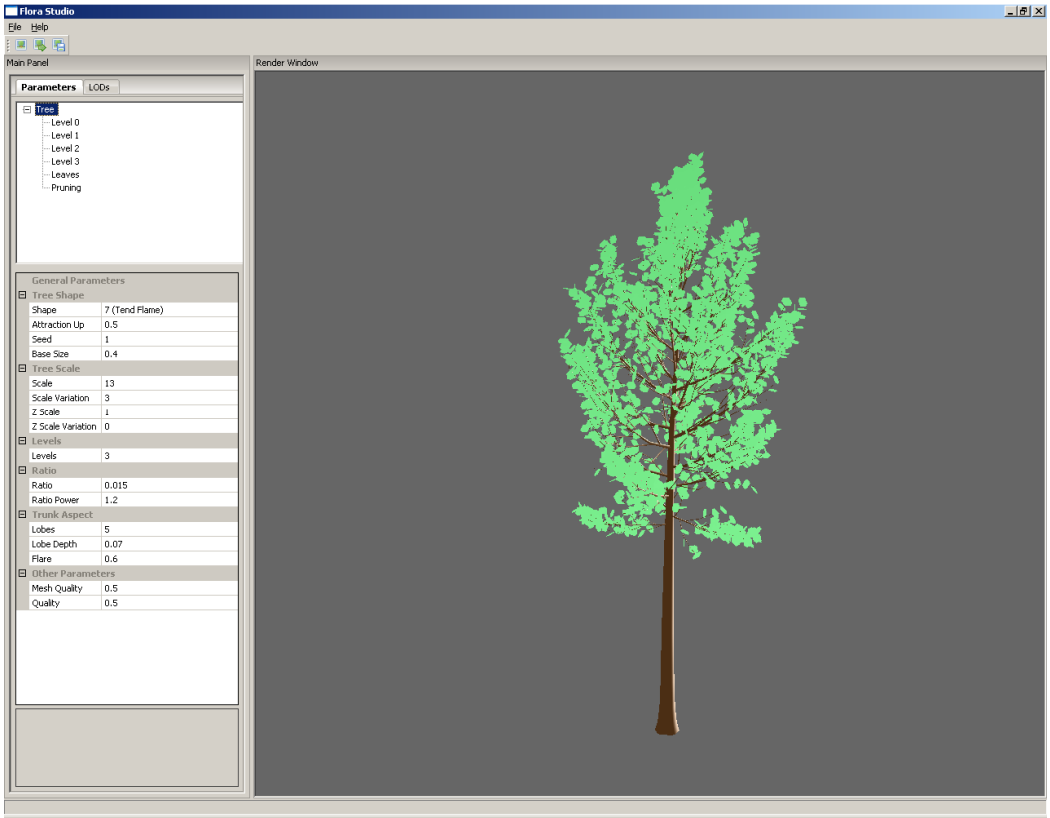
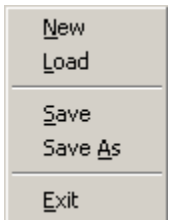


Figure 5.1 Interface de *Flora Studio.*

5.2 Guide d'utilisation

5.2.1 Description des menus

Le menu « Fichier » permet de créer un nouvel arbre (réinitialise aux paramètres par défaut), de charger un fichier de paramètres, de sauvegarder un arbre en cours d'édition et de quitter l'application.

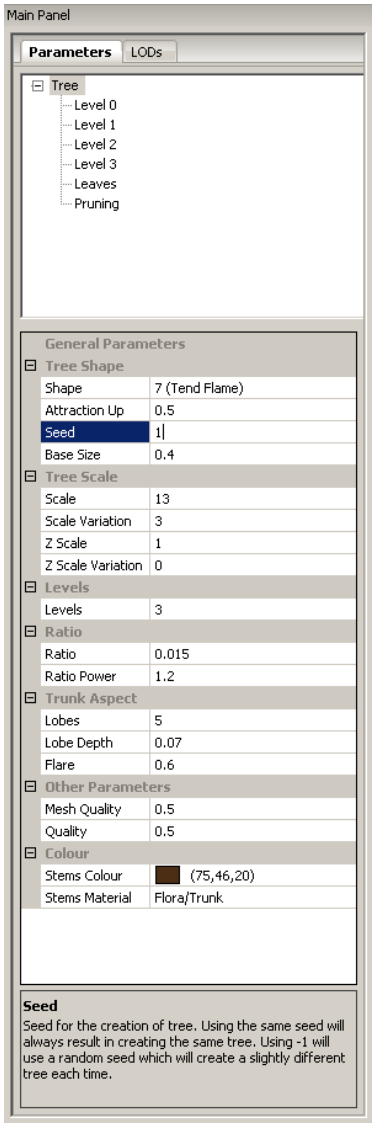


Un fichier est sauvegardé dans un fichier portant l'extension *.fsf (Flora Studio File). Ces fichiers ne sont pas lisibles directement et il est conseillé de passer par « Flora Studio » pour les modifier.

5.2.2 Barre d'outils

La barre d'outils permet de créer un nouvel arbre, charger un fichier ou enregistrer le travail en cours.

5.2.3 Panneau « Parameters »

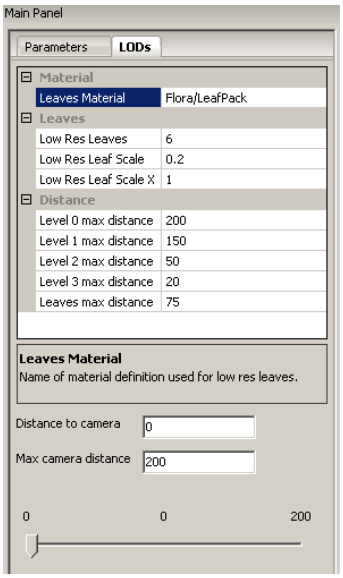


Le panneau principal est composé de deux onglets, l'onglet « *Parameters* » permet d'accéder à tout ce qui est en rapport avec les paramètres de l'arbre. La partie du haut contient un arbre avec les différents niveaux sur lesquels il est possible d'agir (Arbre en général, tronc, premier niveau, etc.).

A la modification de chaque paramètre, l'arbre est modifié en temps réel pour refléter le changement apporté. De plus un descriptif de l'effet des paramètres sur l'arbre est présent au fond du panneau.

Une description complète des paramètres est disponible au chapitre 3.

5.2.4 Panneau « LODs »



L'onglet « *LODs* » permet d'accéder à un panneau de configuration de la gestion de la dégradation à distance. La moitié supérieure du panneau contient les paramètres liés au *material* utilisé pour les feuilles de basse résolution, au nombre de feuilles affichées et à leurs dimensions, ainsi que les distances maximales à laquelles les différents niveaux de branches sont visible. Les distances sont exprimées en unité utilisée par le moteur de rendu, généralement une unité

correspond à un mètre.

La moitié inférieure du panneau contient un champ « *Distance to camera »* permettant de régler la distance « virtuelle » de la caméra de la fenêtre de rendu et d'afficher l'arbre tel qu'il serait visible depuis cette distance. Le champ « *Max camera distance* » permet de modifier la distance maximale sur laquelle on peut utiliser le slider. Le slider a le même effet que le champ « *Distance to camera* », mais permet de modifier la distance de manière plus fluide.

5.2.5 Fenêtre de rendu

La fenêtre de rendu offre un moyen de visualiser l'arbre. Il est possible à l'aide de la souris et du clavier de le déplacer pour le voir sous différents angles.

Les différents mouvements possibles sont :

- *Clique-droit maintenu et mouvement horizontal de la souris* : Permet de faire tourner l'arbre autour de son axe vertical.
- *Clique-droit maintenu et mouvement vertical de la souris* : Permet de faire tourner l'arbre autour d'un axe parallèle à la largeur de l'écran.
- *Clique droit + gauche simultané et mouvement de la souris* : Permet de faire glisser la camera vers le haut ou le bas.
- *Touche W* : Permet d'avancer la caméra en direction de l'arbre.
- *Touche S* : Permet de faire reculer la caméra.
- *Touche A* : Fait glisser la camera sur la gauche.
- *Touche D* : Fait glisser la caméra sur la droite.
- *Touche Page Up* : Fait bouger la caméra verticalement vers le haut.
- *Touche Page Down* : Fait bouger la caméra verticalement vers le bas.

L'usage de la souris pour les déplacements de la caméra est parfois chaotique et demande un temps d'adaptation.

Chapitre 6

Problèmes connus

6.1 Usage de la librairie de génération

L'emploi de la librairie de génération n'est pas autant « fluide » qu'il aurait dû l'être idéalement. L'utilisateur final doit encore effectuer de nombreux contrôles et écrire de nombreuses lignes de code concernant la dégradation à distance.

6.2 Attraction up

Le problème vient l'angle de l'attraction verticale qui est additionné à l'angle de ***nCurve***, ce qui ne devrait pas être le cas. L'angle de l'attraction verticale devrait être un quaternion à lui tout seul et orienter progressivement chaque segment de la sous-branche vers le haut. Malheureusement je n'ai pas réussi à implémenter ce comportement comme il l'aurait fallu. L'origine du problème semble être lié au fait que l'attraction verticale se fasse par rapport au système d'axe locale de la sous-branche – Celui-ci tourne autour de l'axe de la branche parente lorsque ***nRotate*** est appliqué – et non au système d'axe global de l'arbre. L'implémentation actuelle, bien que fausse est celle qui donne le résultat le plus proche ce qui devrait être fait.

6.3 Leaf orientation

L'origine du problème est similaire à celle de l'attraction verticale et est principalement lié à des problèmes de système d'axe. Les feuilles sont pour la plupart réorientées comme il le faudrait, mais certaines prennent des positions particulières.

6.4 Stabilité de l'éditeur visuel

L'éditeur visuel crash à sa fermeture lorsqu'un fichier contenant des noms de *material* a été chargé auparavant.

Chapitre 7

Problèmes rencontrés

7.1 Mauvais choix

Dans les premières semaines du travail, je m'étais intéressé à un *framework* complet pour le développement d'application wxWidgets (wxOgreMVC) intégrant Ogre3D. La solution semblait bien conçue et l'application de démonstration fournie avec wxOgreMVC était convaincante. Après quelques jours à travailler avec ce framework, il s'est avéré que d'une part, il était un peu ardu à prendre en main (documentation limitée) et qu'il était particulièrement buggé depuis des mises à jour plus récentes d'Ogre3D.

7.2 Connaissances en mathématiques

Mes lacunes en mathématiques ont été un des plus gros problèmes pour ce travail. Ce travail était complexe au niveau mathématique, avec des notions que nous n'avions soit jamais vu (quaternion), soit qui étaient partiellement oubliées (trigonométrie). Il est aussi parfois difficile de réellement comprendre pourquoi certaines opérations sont utilisées à certains moments et quel sont leurs effets sur ce qui va être généré – particulièrement avec l'attraction verticale et l'orientation des feuilles.

7.3 Connaissance pratique en rendu temps réel

L'absence d'expérience pratique dans le domaine de la synthèse d'image (*Computer Graphics*) s'est avérée parfois pénalisante sur la façon dont utiliser certaines fonctionnalités d'un moteur de rendu, ou sur la façon dont faire certaines choses. Par exemple, l'utilisation du *render to texture* a demandé un temps d'apprentissage.

Plus d'expérience aurait pu permette d'éviter de prendre de mauvaises directions ou de mieux implémenter des fonctionnalités (attraction verticale, orientation des feuilles, etc.).

Un exemple de mauvaise direction prise dans durant le développement concerne la dégradation à distance présentée dans le document de référence. Cet algorithme est peu adapté à une utilisation pratique. Il a été nécessaire de développer un nouvel algorithme plus adapté basé sur des *billboards* et des *impostors*.

7.4 Langage de développement

Le C++ a par moment posé quelques problèmes. Ce langage offre une gestion de la mémoire différente de ce que nous avions l'habitude d'utiliser ce qui n'était pas sans poser parfois des problèmes (fuites mémoires, mauvaise utilisations de pointeurs, heap corruption, etc.).

Chapitre 8

Evolutions futures

8.1 Ajouts de fonctionnalités

8.1.1 Splits

L'implémentation actuelle de l'algorithme n'est malheureusement pas complète. Certains éléments ont dû être abandonnés en cours de développement, comme les *splits* (séparations) de branches.

Pour résumer le fonctionnement des *splits*, une branche a une chance de se séparer en clones sur sa longueur. Cette séparation en clones est définie par le paramètre ***nSegSplits***, qui détermine le nombre de séparations à chaque segment d'une branche. Les valeurs généralement utilisées vont de 0 à 1, 1 signifiant une séparation en deux à chaque segment, une valeur de 2 impliquerai une séparation ternaire.

Par exemple, une valeur de 1.2 amène à la création d'un clone sur 80% des segments et de deux clones sur les 20% restant. L'usage de valeur aléatoire n'étant pas conseillé pour déterminer sur quel segment les séparations ont lieu, une technique similaire à la diffusion d'erreur de Floyd-Steinberg est utilisée.

Une version de base des *splits* de branches a été commencé, mais ne fonctionne cependant que pour le tronc et n'est pas incluse dans la version finale de la librairie de génération car elle demanderait de retravailler une partie trop importante du code afin d'être pleinement fonctionnelle.

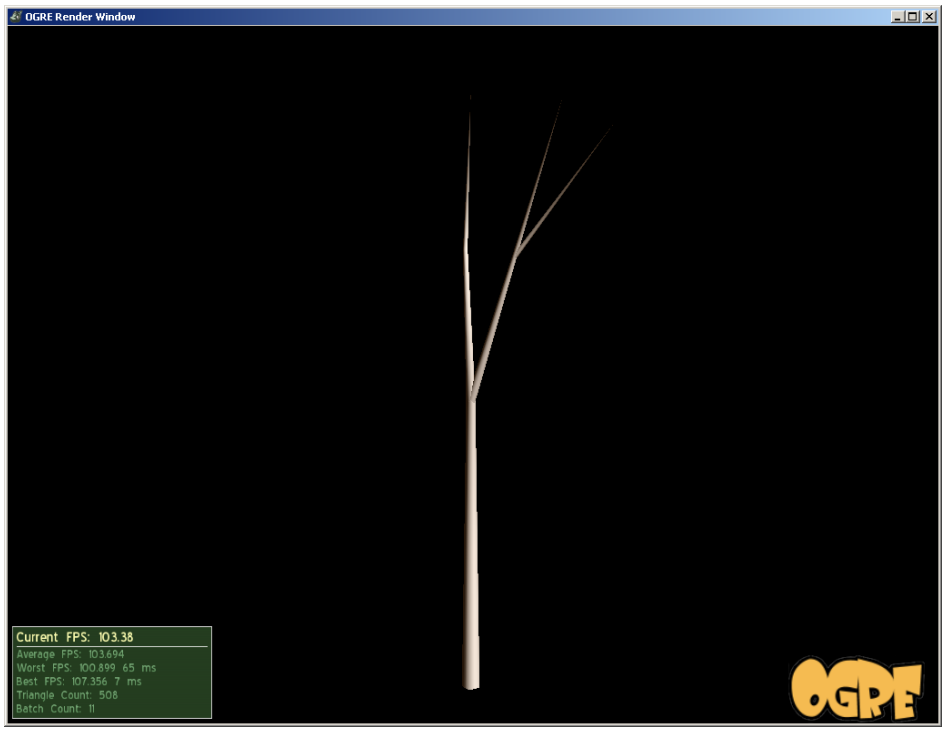


Figure 8.1 Version basique des séparations de branche sur le tronc.

BaseSplits quand à lui permet de créer une séparation en un certain nombre de clones au niveau du premier segment du tronc afin de simuler un arbre ayant deux ou plusieurs troncs mais n'ayant plus de séparation pas la suite.

8.1.2 Pruning

Le *pruning*, tel que décrit par Weber et Penn dans leur papier, est utilisé pour forcer à arbre à tenir à l'intérieur d'une enveloppe spécifique. Cela permet de donner un aspect que l'arbre ne pourrait pas avoir en se limitant aux formes de bases (***Shape***) décrites dans le document de référence.

8.1.3 Amélioration de la dégradation à distance

La méthode de dégradation à distance utilisée pour le moment est satisfaisante, mais il serait bien sûr possible de faire mieux et plus efficace en affinant la création de modèles intermédiaires par exemple.

8.1.4 Ombres

Le calcul des ombres pour des objets complexes comme un arbre est un processus long et coûteux en ressources, surtout s'il doit être effectué à chaque image dans le cadre d'une application en temps réel.

Une possibilité serait de générer une *lightmap* – une structure de données qui contient la clarté d'une surface pour en simuler l'ombre pour des objets statiques – en fonction des paramètres de créations de l'arbre et de sa densité de feuilles. Le but est de l'utiliser comme une ombre projetée sur le sol et de l'adapter en fonction de la position de la source lumineuse, sans avoir à la recalculer à chaque image.

Sans être une solution parfaite, cela peut-être une solution satisfaisante pour donner une impression de sol ombragé sous un arbre par exemple.

8.1.5 Vents

L'ajout de la simulation d'effets externes comme le vent peut indéniablement rajouter au réalisme d'une scène contenant des arbres. Il reste cependant à mettre au point une méthode permettant d'animer un arbre.

8.1.6 Pagination

En allant plus loin avec cette librairie, il pourrait devenir intéressant d'inclure une gestion de la pagination pour les arbres crées. La pagination consiste à ne charger en mémoire et ne travailler qu'avec les arbres présents sur les « *pages* » visible à un moment donné par l'utilisateur de l'application.

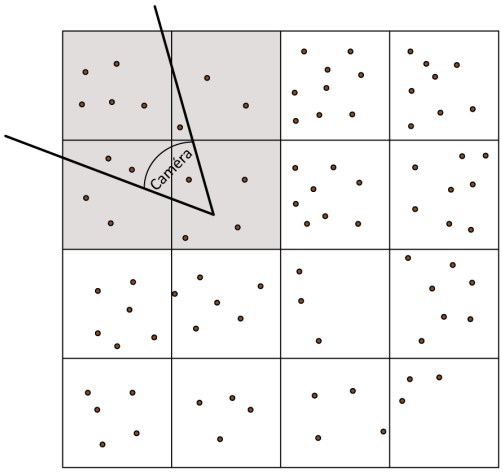


Figure 8.2 Exemple d'une scène recouverte d'arbres et découpée en 16 *pages*. Depuis le point de vu de la caméra, seule les *pages* grisées sont visible et donc uniquement les arbres présents sur ces *pages* sont calculés.

8.1.7 Density maps

De manière générale une « *map* » est un fichier image pouvant contenir différents types d'informations. Une « *density map* » est une *map* contenant des informations sur la densité d'arbre pour chaque lieu d'une scène. Il est ensuite possible de combiner plusieurs *map* pour plusieurs types d'arbre différents.

L'avantage offert ici est qu'une personne créant une scène n'aurait plus à placer manuellement chaque arbre, mais plus qu'à définir des zones et des densités de présence de tel type d'arbre dans cette zone. L'application finale générerait et placerait les arbres selon les informations données.

8.2 Ouverture à la communauté Open-Source

L'intérêt même de cette application serait son ouverture à la communauté open-source pour en faire une option viable pour tout développeur amateur désirant utiliser des arbres réaliste

dans un projet. Ce serait aussi l'opportunité d'en accélérer le développement et d'y faire prendre part des personnes ayant plus d'expérience dans la synthèse d'image.

8.3 Multiplateforme

De par les choix technologique effectué au cours du développement, il serait utile pour tout un chacun de porter l'ensemble sur Linux et MacOS. wxWidgets a été conçu pour permettre la création de GUI multiplateforme sans changement important dans le code écrit et Ogre3D est entièrement multiplateforme.

La librairie de génération ne fait appel qu'à peu de code lié à Windows. Son portage pourrait donc être effectué moyennant de légères modifications. Le code propre à Windows concerne surtout la compilation en DLL.

Chapitre 9

Conclusion

9.1 Conclusion générale

Au travers de ce travail, une méthode permettant la création d'arbres ayant un aspect réaliste a été présentée. Cette méthode présente plusieurs avantages : la création d'un arbre est rapide, les arbres sont détaillés, leur aspect plutôt convaincant et il est facile d'en générer des centaines avec de très légères variations. De plus la création d'une grande variété d'arbres pouvant être très différents les uns des autres est aisée. La dégradation à distance implémentée, bien que limitée, remplit son rôle. L'utilisation finale demande une intervention du développeur souhaitant utiliser la librairie de génération, mais reste cependant relativement simple et ouverte à la plupart des moteurs de rendu.

L'éditeur visuel, de son côté, remplit son rôle. Il permet de concevoir et de pré-visualiser des arbres. Il faut cependant passer du temps à maîtriser le rôle de chaque paramètre impliqué dans la génération de l'arbre final. Tous ne sont pas clairs à comprendre de prime abord et il y a en une quantité importante. Mais un utilisateur habitué devrait arriver à reproduire un arbre en se basant sur des documents photographiques et les exemples disponibles.

Malgré tout, plusieurs problèmes persistent encore. Toutes les fonctionnalités présentées dans le document de référence ne sont pas implémentées, d'autres encore présentent de légers défauts. L'éditeur visuel est encore loin d'avoir atteint sa maturité. Le projet est encore jeune et demandera un travail supplémentaire pour devenir facilement utilisable dans un contexte pratique.

Du point de vue de l'utilisateur final, un ordinateur standard actuel permettant de jouer peut afficher aisément des arbres composés de plusieurs dizaines de milliers de triangles de manière fluide. A l'aide d'un bon algorithme de dégradation à distance, une grande quantité d'arbres peut facilement être affichée à l'écran avec un rendu convaincant et des performances acceptable. De plus, la puissance des processeurs et des cartes graphiques augmente rapidement et le matériel performant pour ce type d'application devient abordable. Il est donc de plus en plus facile de présenter des environnements très détaillés remplis d'objets eux aussi détaillés.

On ne peut au final que se réjouir de la présence de ce type d'outil pour les développeurs de jeux et d'autres produits multimédia faisant appel à du rendu 3D et on ne peut qu'espérer leur utilisation par le plus grand nombre. La représentation de la végétation est un domaine vaste et fascinant dans lequel il reste encore beaucoup à faire.

9.2 Conclusion personnelle

Ce travail a été pour moi l'occasion d'aborder une matière qui me tient à cœur depuis quelques années. Le champ de la génération procédurale de végétation est un sujet qui m'attire tout particulièrement dans le domaine de la synthèse d'image. Ce travail a aussi été l'occasion de mener pour la première fois un projet de bonne taille en C++, ce langage n'ayant jamais été abordé en cours. Ce fut également l'occasion d'utiliser des technologies liées de près ou de loin au jeu vidéo, domaine qui m'attire fortement et vers lequel je souhaite m'orienter après mes études. Ce domaine cependant correspond peu à l'orientation de l'école.

Le fait de partir sur des bases ainsi limitées a ses désavantages. On passe plus de temps à se former, à chercher et à essayer de comprendre comment faire certaines choses. On doit de plus acquérir de nombreuses notions de bases dans les domaines du travail. Celui-ci n'avance pas toujours autant vite qu'on le souhaiterait et souvent, les directions prisent ne mènent à rien. Mais en dehors de ces handicaps, il est très gratifiant d'arriver à un résultat, de voir un arbre apparaître à l'écran et de le voir se modifier de la façon dont on le souhaite en fonction des paramètres donnés.

Au final, ce travail m'aura beaucoup appris : sur la façon de gérer un tel projet, sur la pratique même de la programmation, sur l'usage de wxWidgets et Ogre3D que je n'avais que très peu utilisés auparavant. Je suis sûr que l'expérience amassée lors de ce travail s'avérera utile pour la suite de mon parcours, à commencer par la rentrée prochaine avec une formation de développeur orientée vers le jeu vidéo, que je suivrai à l'université de Lyon.

Avec le recul cependant, j'aborderai certaines choses autrement. J'essaierai d'organiser mon code différemment et de prendre plus de temps en début de projet afin de réfléchir à une meilleure façon de construire l'application et de mener le développement.

Enfin, puisque que ce projet a pour moi une certaine importance certaine, j'espère pouvoir le poursuivre. Je souhaiterais le mener à maturité, pouvoir l'utiliser dans mes futurs projets et pourquoi pas, en faire profiter d'autres personnes.

Chapitre 10

Annexes

10.1 Attestation

Je déclare, par ce document, que j'ai effectué le travail de diplôme ci-annexé seul, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de diplôme, y compris au partenaire de recherche appliquée avec lequel j'ai collaboré, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail et que je cite ci-après :

10.2 Bibliographie / Ressources

Livres

TOMAS AKENINE-MÖLLER et ERIC HAINES, *Real-Time Rendering,* Seconde Edition, Editions A K Peters, 2002

ERIC LANGYEL, *Mathematics for 3D Games Programming & Computer* Graphics, Seconde Edition, Editions Charles River Media, 2004

DAVID H. EBERLY, *3D Game Engine Architecture*, Editions Morgan Kaufmann, 2005

Sites Internet

Tutoriaux divers sur Ogre3D
http://www.ogre3d.org/wiki

Création manuel d'une sphère dans Ogre3D avec Vertex et Index Buffers
http://www.ogre3d.org/wiki/index.php/ManualSphereMeshes

Introduction à l'emploie des quaternions
http://www.ogre3d.org/wiki/index.php/Quaternion_and_Rotation_Primer

Manuel d'Ogre3D - Materials
http://www.ogre3d.org/docs/manual/manual_14.html

Manuel d'Ogre3D – Hardware Buffers
http://www.ogre3d.org/docs/manual/manual_46.html

Documentation de l'API d'Ogre3D
http://www.ogre3d.org/docs/api/html/

Forum officiel d'Ogre3D
http://www.ogre3d.org/phpBB2/

Documentation de l'API de wxWidgets
http://docs.wxwidgets.org/

Tutoriaux divers sur wxWdigets
http://wiki.wxwidgets.org/

Site de Jason Weber, FAQ sur son article utilisé dans ce travail
http://www.imonk.com/baboon/trees/faq.html

Ressources / outils utilisés

Code source Ogre3D : Moteur de rendu
http://www.ogre3d.org

Code source Wild Magic 4.7 : Moteur de rendu
http://www.geometrictools.com

wxFormBuilder : Editeur WYSIWYG pour wxWidgets
http://www.wxformbuilder.org

wxOgre : *Wrapper* wxWidgets pour Ogre3D
http://www.ogre3d.org/phpBB2/viewtopic.php?t=20652&start=175

wxPropertyGrid : Add-on wxWidgets pour des grilles de propriétés (utilisées dans l'éditeur visuel)
http://wxpropgrid.sourceforge.net

Arbaro : Implémentation Java de l'article de Weber et Penn
http://arbaro.sourceforge.net

MeshTree : Portage en C++ de TReal (http://members.chello.nl/~l.vandenheuvel2/TReal/)
http://www.ogre3d.org/phpBB2/viewtopic.php?t=6384

PagedGeometry : Add-on Ogre3D
http://www.ogre3d.org/wiki/index.php/PagedGeometry_Engine

DirectX SDK Mars 2008
http://msdn.microsoft.com/en-us/directx/aa937788.aspx

10.3 Rapport des heures

Ci-dessous, le rapport des heures par jour et par semaine. Pour un rapport des heures détaillé ainsi que du travail réalisé chaque jour, merci de se référer au fichier Excel contenu sur le CD joint à ce dossier.

Semaine	19.05.2008	26.05.2008	02.06.2008	09.06.2008	16.06.2008
Lundi	9h	0h	5h	6h	8h30
Mardi	8h	9h30	0h	3h30	8h
Mercredi	0h	0h	0h	0h	9h
Jeudi	0h	3h30	0h	7h	6h
Vendredi	0h	6h	0h	7h30	5h
Samedi	0h	0h	0h	0h	4h
Dimanche	0h	2h	2h	0h	0h
Total semaine	17h	21h	7h	24h	40h30
Total cumulé	**17h**	**38h**	**45h**	**69h**	**109h30**

Semaine	23.06.2008	30.06.2008	07.07.2008	17.07.2008	21.07.2008
Lundi	2h	8h	6h	7h	9h
Mardi	9h	8h	6h	7h	9h30
Mercredi	5h	9h	0h	0h	8h
Jeudi	9h30	9h30	0h	9h	9h30
Vendredi	7h	6h30	7h	8h30	9h
Samedi	7h	2h	0h	8h	9h30
Dimanche	5h	0h	0h	9h	6h
Total semaine	44h30	43h	19h	48h30	60h30
Total cumulé	**154h**	**197h**	**216h**	**264h5**	**325h**

Semaine	**28.07.2008**
Lundi	7h
Mardi	7h
Mercredi	8h
Jeudi	9h
Vendredi	8h30
Samedi	9h30
Dimanche	9h
Total semaine	58h
Total cumulé	**383h**

10.4 Implémentation exemple dans Ogre3D

Ci-dessous, quelques extraits de code d'une implémentation dans Ogre3D. Le code source complet est disponible sur le CD joint à ce dossier.

```
// Créationde la scène
virtual void createScene(void)
{
	initRTT();

	// Configure la lumière ambiante
	mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));

	// Crée une source de lumière
	Light* l = mSceneMgr->createLight("MainLight");
	l->setPosition(150,150,150);

	m_pParam = createParameterList();

	createTree(Ogre::String("Tree0"), m_pParam, -1);
	createTree(Ogre::String("Tree1"), m_pParam, -1);
	createTree(Ogre::String("Tree2"), m_pParam, -1);
	createTree(Ogre::String("Tree3"), m_pParam, -1);
	createTree(Ogre::String("Tree4"), m_pParam, -1);

	Entity* entity;
	SceneNode* sceneNode;
	BillboardSet* impostorSet;
	Billboard* bill;
	TreeEntity* treeEntity;

	// Crée et place les 5 arbres défini plus haut
	for(int i = 0 ; i < 5 ; i++)
	{
		std::stringstream out;
		out << i;

		Ogre::String num(out.str());

		entity = mSceneMgr->createEntity("TreeEntity" + num, "Tree" + num);
		sceneNode = mSceneMgr->getRootSceneNode()->createChildSceneNode();
		sceneNode->pitch(Degree(-90));
		sceneNode->setPosition(5 * i, 0, 5 * i);
		sceneNode->attachObject(entity);

		createTreeImpostor(entity);

		Ogre::String strName("Flora/Tree" + num);

		MaterialPtr material = MaterialManager::getSingleton().create(
		strName, ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

		material->getTechnique(0)->getPass(0)->createTextureUnitState(texture->
                                                        getName());
		material->getTechnique(0)->getPass(0)->
                        setAlphaRejectSettings(CMPF_GREATER_EQUAL, 128);
		material->getTechnique(0)->getPass(0)->
                        setSceneBlending(SBT_TRANSPARENT_ALPHA);

		impostorSet = mSceneMgr->createBillboardSet("ImpostorSet_Tree" + num);
		bill = impostorSet->createBillboard(0, 0, 0);
		bill->setDimensions(entity->getBoundingBox().getMaximum().z, entity->
                        getBoundingBox().getMaximum().z);
		impostorSet->setBillboardOrigin(BillboardOrigin::BBO_BOTTOM_CENTER);
		impostorSet->setMaterialName(strName);
		impostorSet->setBillboardType(BillboardType::BBT_ORIENTED_COMMON);

		sceneNode->attachObject(impostorSet);

		//TreeEntity utilisé pour l'observer pattern (voir chapitre 4.2.2)
		treeEntity = new TreeEntity(mFrameListener, m_pParam);
		treeEntity->mEntity = entity;
		treeEntity->mLowResLeaves = mSceneMgr->getBillboardSet(
                                                "Leaves_Tree" + num);
```

```
                sceneNode->attachObject(treeEntity->mLowResLeaves);
                treeEntity->mImpostor = impostorSet;
        }
}

//Génération du mesh d'un arbre
void createTree(Ogre::String& name, Parameter* pParam, int seed)
{
        //Creating Tree
        Tree* pTreeGen = new Tree(pParam, seed);
        //End Tree's creation

        MeshPtr pTree = MeshManager::getSingleton().createManual(name,
                ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

        //VertexData
        pTree->sharedVertexData = new VertexData();
        VertexData* vertexData = pTree->sharedVertexData;

        //VertexDeclaration
        VertexDeclaration* vertexDecl = vertexData->vertexDeclaration;
        size_t currentOffset = 0;

        //Position
        vertexDecl->addElement(0, currentOffset, VET_FLOAT3, VES_POSITION);
        currentOffset += VertexElement::getTypeSize(VET_FLOAT3);
        //Normal
        vertexDecl->addElement(0, currentOffset, VET_FLOAT3, VES_NORMAL);
        currentOffset += VertexElement::getTypeSize(VET_FLOAT3);
        //Mapping
        vertexDecl->addElement(0, currentOffset, VET_FLOAT2, VES_TEXTURE_COORDINATES);
        currentOffset += VertexElement::getTypeSize(VET_FLOAT2);

        //Color
        currentOffset = 0;
        vertexDecl->addElement(1, currentOffset, VET_COLOUR, VES_DIFFUSE);
        currentOffset += VertexElement::getTypeSize(VET_COLOUR);

        vertexData->vertexCount = pTreeGen->getVerticesCount();

        //Allocate vertex buffer
        HardwareVertexBufferSharedPtr vBuf = HardwareBufferManager::getSingleton().
                createVertexBuffer(vertexDecl->getVertexSize(0),
                vertexData->vertexCount,
                HardwareBuffer::HBU_STATIC_WRITE_ONLY,
                false);

        //binding
        VertexBufferBinding* binding = vertexData->vertexBufferBinding;
        binding->setBinding(0, vBuf);

        float* pVertexArray = static_cast<float*>(vBuf->
                lock(HardwareBuffer::HBL_DISCARD));

        //Allocate vertex color buffer
        HardwareVertexBufferSharedPtr vColBuf = HardwareBufferManager::getSingleton().
                createVertexBuffer(vertexDecl->getVertexSize(1),
                vertexData->vertexCount,
                HardwareBuffer::HBU_STATIC_WRITE_ONLY,
                false);

        //binding
        binding->setBinding(1, vColBuf);

        RGBA* pVertexColour = static_cast<RGBA*>(vColBuf->
                lock(HardwareBuffer::HBL_DISCARD));

        pTreeGen->createVerticesStream(&pVertexArray, &pVertexColour);

        vBuf->unlock();
        vColBuf->unlock();
```

```
//
// Faces
//

int indexCount;
bool flag = true;

for(int i = 0 ; i < pParam->m_Levels ; i++)
{
	indexCount = 3 * pTreeGen->getTrunk()->countFaces(i);

	if(indexCount == 0)
	{
		flag = false;
		break;
	}

	SubMesh* pStemSubMesh = pTree->createSubMesh();

	pStemSubMesh->useSharedVertices = true;

	pStemSubMesh->indexData->indexCount = indexCount;

	pStemSubMesh->indexData->indexBuffer =
		HardwareBufferManager::getSingleton().
			createIndexBuffer(HardwareIndexBuffer::IT_32BIT,
			pStemSubMesh->indexData->indexCount,
			HardwareBuffer::HBU_STATIC_WRITE_ONLY,
			false);

	HardwareIndexBufferSharedPtr iBuf = pStemSubMesh->indexData->indexBuffer;

	unsigned long* pFacesIndices = static_cast<unsigned long*>(iBuf->
		lock(HardwareBuffer::HBL_DISCARD));

	pTreeGen->createIndicesStream(&pFacesIndices, i);

	iBuf->unlock();

	pStemSubMesh->setMaterialName("Flora/Trunk");
}

indexCount = 3 * pTreeGen->countLeavesFaces();

if(indexCount != 0 && flag)
{
	SubMesh* pStemSubMesh = pTree->createSubMesh();

	pStemSubMesh->useSharedVertices = true;

	pStemSubMesh->indexData->indexCount = 3 * pTreeGen->countLeavesFaces();

	pStemSubMesh->indexData->indexBuffer =
		HardwareBufferManager::getSingleton().
			createIndexBuffer(HardwareIndexBuffer::IT_32BIT,
			pStemSubMesh->indexData->indexCount,
			HardwareBuffer::HBU_STATIC_WRITE_ONLY,
			false);

	HardwareIndexBufferSharedPtr iBuf = pStemSubMesh->indexData->indexBuffer;

	unsigned long* pFacesIndices = static_cast<unsigned long*>(iBuf->
		lock(HardwareBuffer::HBL_DISCARD));

	pTreeGen->createLeavesIndicesStream(&pFacesIndices);

	iBuf->unlock();

	pStemSubMesh->setMaterialName("Flora/Leaf");

	createBillboardLeaves(pTreeGen, name);
}
```

```
        Ogre::Vector3 vb1, vb2;
        vb1 = Ogre::Vector3(pTreeGen->getMinX(), pTreeGen->getMinY(),
                                    pTreeGen->getMinZ()) ;
        vb2 = Ogre::Vector3(pTreeGen->getMaxX(), pTreeGen->getMaxY(),
                                    pTreeGen->getMaxZ()) ;
        pTree->_setBounds(AxisAlignedBox(vb1, vb2));
        pTree->_setBoundingSphereRadius((vb1 + vb2).length()/2);

        pTree->load();
}

//Génération des billboards des feuilles
void createBillboardLeaves(Flora::Tree* pTree, Ogre::String& name)
{
        unsigned long nbLeaves = pTree->countLowResLeaves();

        BillboardSet* billSet = mSceneMgr->createBillboardSet(
                                            "Leaves_" + name, nbLeaves);

        float* pVertexArray = new float[nbLeaves*5];
        unsigned int* pColourArray = new unsigned int[nbLeaves];

        float* pVertexArrayTmp = pVertexArray;
        unsigned int* pColourArrayTmp = pColourArray;

        pTree->createLowResMesh(&pVertexArrayTmp, &pColourArrayTmp, true);

        Billboard* leaf;

        for(int i = 0 ; i < nbLeaves * 5 ; i = i+5)
        {
                leaf = billSet->createBillboard(pVertexArray[i], pVertexArray[i+1],
                                                        pVertexArray[i+2]);
                leaf->setDimensions(pVertexArray[i+3], pVertexArray[i+4]);
        }

        billSet->setMaterialName("Flora/LeafPack");
        billSet->setVisible(false);
}

//Génération de l'imposteur
void createTreeImpostor(Entity* pTree)
{
        texture = TextureManager::getSingleton().createManual(
                pTree->getName() + "_RttTex",
                ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, TEX_TYPE_2D,
                256, 256, 0, PF_R8G8B8A8, TU_RENDERTARGET );

        rttText = texture->getBuffer()->getRenderTarget();
        {
                rttText->setAutoUpdated(false);
                texture->setNumMipmaps(MIP_UNLIMITED);

                SceneNode* oldSceneNode = pTree->getParentSceneNode();

                if(oldSceneNode != 0)
                        oldSceneNode->detachObject(pTree);

                renderNode->attachObject(pTree);
                renderNode->setPosition(0,0,0);

                renderCamera->setNearClipDistance(mCamera->getNearClipDistance());
                renderCamera->setFarClipDistance(mCamera->getFarClipDistance());
                renderCamera->setAspectRatio(1.0f);
                renderCamera->setFOVy(mCamera->getFOVy());
                renderCamera->setOrientation(Ogre::Quaternion::IDENTITY);

                //Disable mipmapping (without this, masked textures look bad)
                MaterialManager *mm = MaterialManager::getSingletonPtr();
                FilterOptions oldMinFilter = mm->getDefaultTextureFiltering(FT_MIN);
                FilterOptions oldMagFilter = mm->getDefaultTextureFiltering(FT_MAG);
                FilterOptions oldMipFilter = mm->getDefaultTextureFiltering(FT_MIP);
                mm->setDefaultTextureFiltering(FO_POINT, FO_LINEAR, FO_NONE);

                Real entityRadius = pTree->getBoundingBox().getMaximum().x / 2;
```

```
                Real entityDiameter = pTree->getBoundingBox().getMaximum().x;

                renderCamera->setPosition(0,pTree->getBoundingBox().getMaximum().z / 2,
                        (pTree->getBoundingBox().getMaximum().z / 2) /
                        Math::Sin(renderCamera->getFOVy() / 2));

                //Render only the entity
                Ogre::SceneManager::SpecialCaseRenderQueueMode
                       OldSpecialCaseRenderQueueMode = mSceneMgr->
                                                    getSpecialCaseRenderQueueMode();

                mSceneMgr->
                       setSpecialCaseRenderQueueMode(Ogre::SceneManager::SCRQM_INCLUDE);
                mSceneMgr->addSpecialCaseRenderQueue(RENDER_QUEUE_6 + 1);

                uint8 oldRenderQueueGroup = pTree->getRenderQueueGroup();
                pTree->setRenderQueueGroup(RENDER_QUEUE_6 + 1);
                bool oldVisible = pTree->getVisible();
                pTree->setVisible(true);
                float oldMaxDistance = pTree->getRenderingDistance();
                pTree->setRenderingDistance(0);
                //~Render only the entity

                v = rttText->addViewport(renderCamera);
                v->setClearEveryFrame(true);
                v->setBackgroundColour(ColourValue(0.0, 0.0, 1.0, 0.0));
                v->setOverlaysEnabled(false);

                rttText->update();

                //Restor previous state
                pTree->setVisible(oldVisible);
                pTree->setRenderQueueGroup(oldRenderQueueGroup);
                pTree->setRenderingDistance(oldMaxDistance);
                mSceneMgr->removeSpecialCaseRenderQueue(RENDER_QUEUE_6 + 1);

                mSceneMgr->setSpecialCaseRenderQueueMode(OldSpecialCaseRenderQueueMode);
                //~Restor previous state

                mm->setDefaultTextureFiltering(oldMinFilter, oldMagFilter, oldMipFilter);

                renderNode->detachObject(pTree);
                oldSceneNode->attachObject(pTree);
        }
}
```

10.5 Liste de paramètres

Ci-dessous, trois listes de paramètres pour trois arbres. Seuls les paramètres définis par Weber et Penn sont listés.

Paramètre	**Tremble**	**Tupelo Noir**	**Sassafras**
Shape	7	4	2
BaseSize	0.4	0.2	0.2
Scale, ScaleV, ZScale, ZScaleV	13, 3, 1, 0	23, 5, 1, 0	23, 7, 1, 0
Levels	3	4	4
Ratio, RatioPower	0.015, 1.2	0.015, 1.3	0.02, 1.3
Lobes, LobeDepth	5, 0.07	3, 0.1	3, 0.05
Flare	0.6	1	0.5
0Scale, 0ScaleV	1, 0	1, 0	1, 0
0Length, 0LengthV, 0Taper	1, 0, 1	1, 0, 1.1	1, 0, 1.05
0BaseSplits	0	0	0
0SegSplits, 0SplitAngle, 0SplitAngleV	0, 0, 0	0, 0, 0	0, 0, 0
0CurveRes, 0Curve, 0CurveBack, 0CurveV	3, 0, 0, 20	10, 0, 0, 40	16, 0, 0, 60
1DownAngle, 1DownAngleV	60, -50	60, -40	90, -10
1Rotate, 1RotateV, 1Branches	140, 0, 50	140, 0, 50	140, 0, 20
1Length, 1LengthV, 1Taper	0.3, 0, 1	0.3, 0.05, 1	0.4, 0, 1
1SegSplits, 1SplitAngle, 1SplitAngleV	0, 0, 0	0, 0, 0	0, 0, 0
1CurveRes, 1Curve, 1CurveBack, 1CurveV	5, -40, 0, 50	10, 0, 0, 90	15, -60, 30, 200
2DownAngle, 2DownAngleV	45, 10	30, 10	50, 10
2Rotate, 2RotateV, 2Branches	140, 0, 30	140, 0, 25	140, 0, 20
2Length, 2LengthV, 2Taper	0.6, 0, 1	0.6, 0.1, 1	0.7, 0, 1
2SegSplits, 2SplitAngle, 2SplitAngleV	0, 0, 0	0, 0, 0	0.05, 20, 0
2CurveRes, Curve, 2CurveBack, 2CurveV	3, -40, 0, 75	10, -10, 0, 150	8, -40, 0, 300
3DownAngle, 3DownAangleV	45, 10	45, 10	45, 10
3Rotate, 3RotateV, 3Branches	77, 0, 10	140, 0, 12	140, 0, 30
3Length, 3LengthV, 3Taper	0, 0, 1	0.4, 0, 1	0.4, 0, 1
3SegSplits, 3SplitAngle, 3SplitAngleV	0, 0, 0	0, 0, 0	0, 0, 0
3CurveRes, 3Curve, 3CurveBack, 3CurveV	1, 0, 0, 0	1, 0, 0, 0	3, 0, 0, 200
Leaves, LeafShape	25, 0	6, 0	20, 0
LeafScale, LeafScaleX	0.17, 1	0.3, 0.5	0.25, 0.7
AttractionUp	0.5	0.5	0.5
PruneRatio	0	0	0
PruneWidth, PruneWidthPeak	0.5, 0.5	0.5, 0.5	0.5, 0.5
PrunePowerLow, PrunePowerHigh	0.5, 0.5	0.5, 0.5	0.5, 0.5

10.6 Arbres

Dans cette section, quelques images d'arbres créés à l'aide de la suite d'outils développé dans le cadre de ce travail.

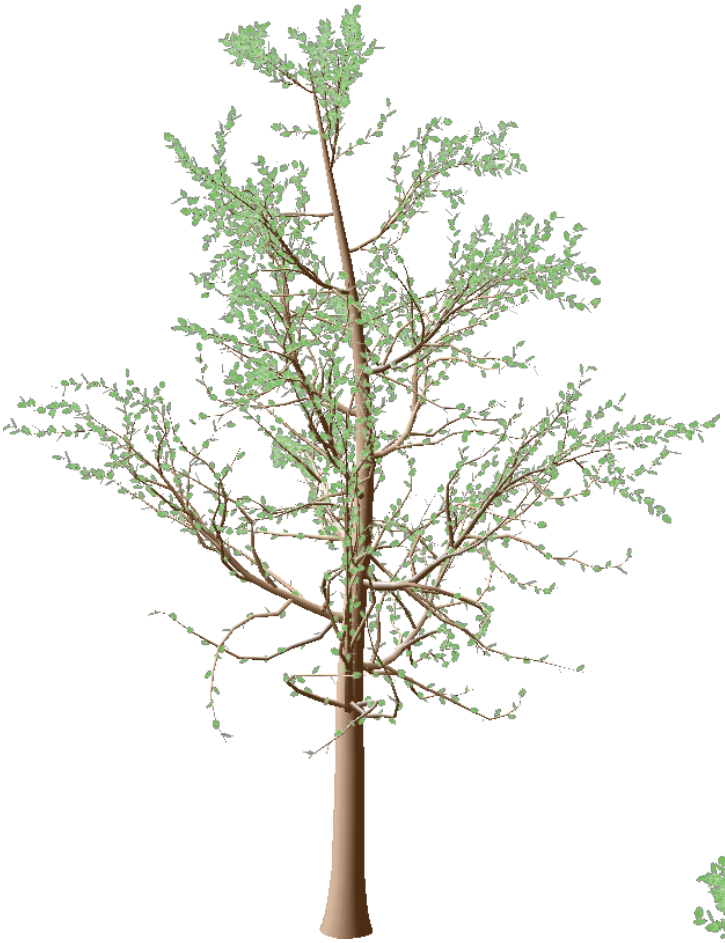


Figure 10.1 Sassafras

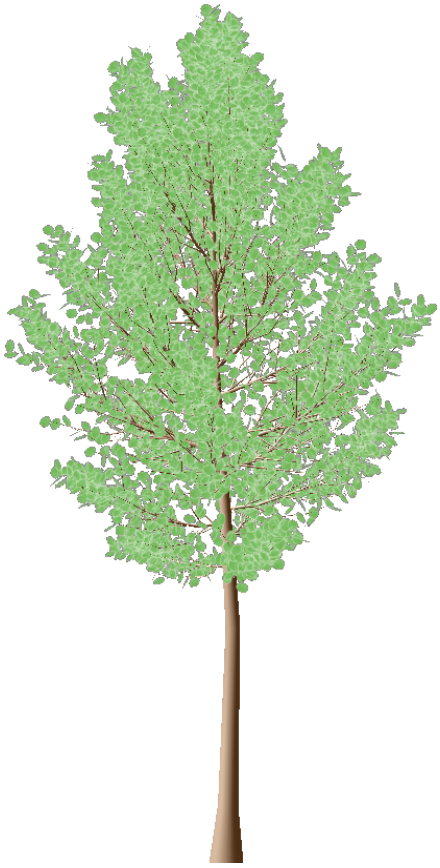


Figure 10.2 Tremble

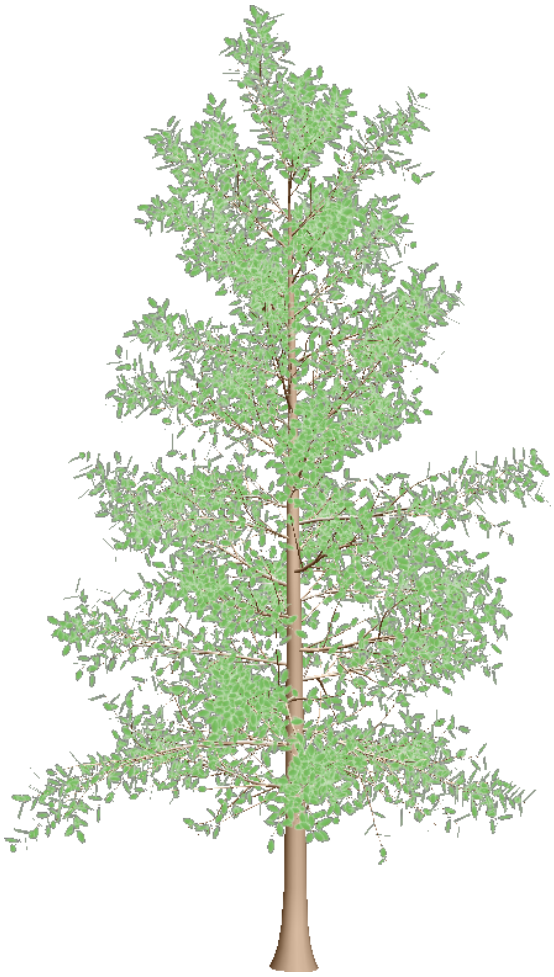


Figure 10.3 Tupelo Noir

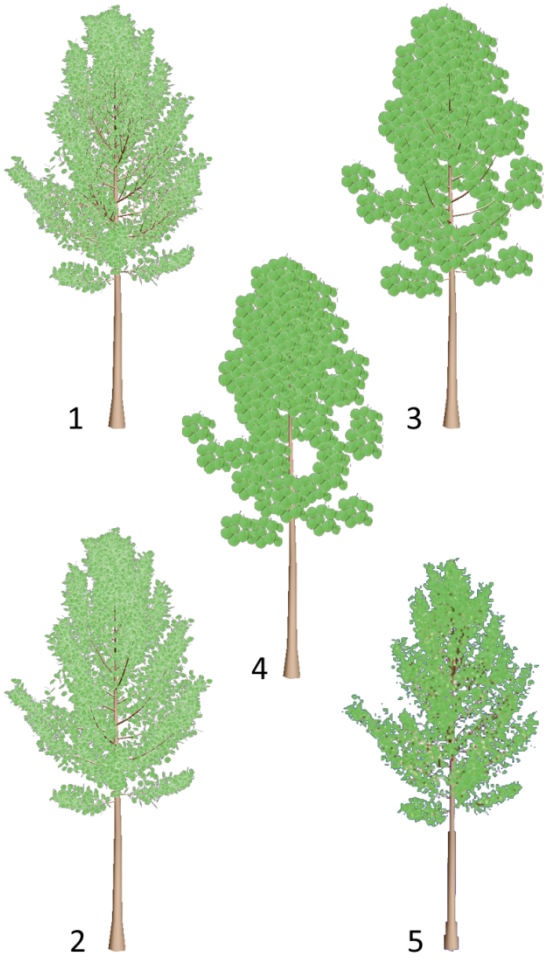


Figure 10.4 Les différentes étapes de la dégradation à distance. (1) L'arbre avec tous ses détails. A chaque étape, des détails sont retirés jusqu'à (5) où l'arbre est remplacé par un *impostor*.

10.7 Creation and Rendering of Realistic Trees

Une copie de l'article de référence est annexée à partir de la page suivante.

Cet article est aussi disponible sur le site de l'ACM à l'adresse :
http://portal.acm.org/citation.cfm?id=218427