REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider – BISKRA
Faculté des sciences exactes et des sciences de la nature et de la vie
Département d'informatique

Mémoire en vue de l'obtention du diplôme de

Magister en Informatique

Option : Synthèse d'Image et Vie Artificielle

Intitulé

Illumination et ombres pour les forêts pour un rendu temps-réel

Par

Henni Nadir

2013

Devant le jury :

Président	**Djedi Noureddine**	**Professeur**	**Université de Biskra**
Rapporteur	**Babahenini Mouhamed Chaouki**	**Maître de Conférences A**	**Université de Biskra**
Examinateur	**Foudil Cherif**	**Maître de Conférences A**	**Université de Biskra**
Examinateur	**Melkmi Kamal Eddine**	**Maître de Conférences A**	**Université de Biskra**

Remerciements

Premièrement et avant tout je veux remercier ALLAH le tout puissant.

Je veux remercier infiniment mon encadreur Monsieur Babahenini Mohamed Chaouki pour ces conseils très précieux, le temps et la patience qu'il a consacré pour mener à bout ce travail, aussi un grand merci pour tous les professeurs du département de l'informatique de l'université de Mohamed Khider de Biskra qui m'ont enseigné pendant l'année théorique pour les efforts très appréciés qu'ils ont fourni.

Je remercie chaleureusement Monsieur le Professeur Noureddine Djedi pour m'avoir donné le grand honneur de présider ce jury.

J'exprime mes remerciements à Messieurs les Docteurs Foudil Cherif et Melkmi Kamal Eddine pour le temps et l'effort qu'ils ont fournis pour lire et évaluer ce travail.

Je remercie ma famille et spécialement mes parents pour leur soutienet leur encouragement qu'ils m'ont apporté pendant mes études, et pendant la période durant laquelle je travaillais sur mon mémoire.

Je remercie enfin mes collègues de l'année théorique (SIVA) pour les temps agréables qu'on a passé ensemble et je cite spécialement mes amis Farid Zerari, Khaldi Belkacemet Ali Beddiaf.

Henni Nadir

Résumé

La représentation des scènes naturelles et leurs rendu a toujours représenté un défit pour la synthèse d'image, car ils sont d'un côté indispensable pour beaucoup d'applications comme les simulateurs d'écosystèmes, la cartographie 3D, les simulateurs de voles, jeux de vidéo …, et d'un autre coté vu leur complexité géométrique, due au très grand nombre de petits détails qu'ils contiennent et qui sont très difficile à modéliser, très couteux en espace mémoire, et en temps de calcul.

Dans ces dernières années le matériel graphique a connu un développement énorme, avec l'introduction des GPU (Graphical Processing Units), dont l'évolution a été adaptée au calcul parallèle intensif. Le développement des GPUs a visé la flexibilité de programmation en introduisant les shaders qui sont des programmes exécutables directement sur ces unités, et la capacité de générer localement de nouveaux sommets, qui permet d'enrichir les représentations géométriques des objets de la scène sans avoir à transférer des données supplémentaires entre CPU et GPU, et exploiter les unités de calcule du GPU pour le faire.

Le développement énorme des capacités de calcul des GPU, et les nouvelles possibilités offrent plus d'implication de la géométrie dans le rendu temps réel des végétaux.

Pour pouvoir donner une méthode de rendu des végétaux qui maximise l'exploitation des GPUs en gardant la possibilité d'utiliser la représentation géométrique du végétale, à la demande, on doit d'abord déterminer une méthode de rendu qui soit adaptée au rendu de très grandes quantités de géométrie en tirant un profit maximal des performances du GPU, puis définir une représentation géométrique qui est la mieux adaptée a cette méthode de rendu, et enfin optimiser tout le procédé de rendu en exploitant les propriétés spécifique à cette représentation lors de rendu (similarités et pré-connaissance sur la géométrie). Enfin la méthode que nous avons proposée prend en compte les niveaux de détail dans le calcul du rendu, ce qui a permis d'optimiser les performances.

ملخص:

لطالما شكل تمثيل المشاهد الطبيعية تحديا دائما لرسومات الحاسوب، وذلك بسبب ضرورتها لكثير من التطبيقات مثل نظم المحاكاة الإيكولوجية، ورسم الخرائط ثلاثيه الأبعاد و نظم محاكاة الطيران، وألعاب الفيديو ... من جهة، وبسبب تعقيداتها الهندسية، الناجمة عن العدد الكبير جدا من التفاصيل الصغيرة التي تحتوي عليها من جهة أخرى، هذه التفاصيل صعبة جدا للبناء بواسطة نموذج هندسي، وتمثيلها مكلف جدا في الذاكرة، كما أن رسمها يتطلب الكثير من وقت المعالجة.

شهدت أجهزة معالجة الرسوم في السنوات الأخيرةتطورا كبيرا، مع ظهور وحدات GPU (وحدة معالجة الرسوم)، وهي معالجات متخصصة لإنشاء الرسومات عرفت نموا كبيرا جدا، تكيفت بشكل كبير خلال تطورها مع المعالجة المتوازية المكثفة، كما هدف تطور هذه الوحدات أيضا إلى زيادة مرونة البرمجة، عن طريق إدخال برامج قابلة للتنفيذ مباشرة على GPU (sredahs)، القدرة على إنتاج رؤوس (stemmos)جديدة محليا ، معطية إمكانية إثراء تمثيل الأجسام الهندسية في المشهد دون الحاجة إلى نقل المزيد من البيانات بين CPU و GPU، واستخدام إمكانيات وحدة GPU الحسابية للقيام بذلك.
هذا التطور الهائل لوحدات GPU أعطى فرصا جديدة توفر إمكانية لمزيد من إشراك الهندسة في إنشاء النباتات للتطبيقات التفاعلية، إذا ما استغلت بشكل صحيح.

لتوفير طريقة لإنشاء صور الأشجار بأحسن استغلال لوحدات معالجة الرسومات مع الإبقاء على إمكانية استخدام التمثيل الهندسي لإنشاء صورة النبات حسب الطلب، يجب علينا أولا تحديد طريقة تتكيف جيدا مع معالجة كميات كبيرة جدا من الهندسة عن طريق سحب أقصى قدر من أداء GPU ثم تحديد التمثيل الهندسي الأنسب لهذه الطريقة ، ثم تحسين عملية إنشاء الصور من خلال استغلال كامل لخصائص هذا التمثيل (التشابه و المعرفة المسبقة بالخصائص الهندسية).

وهناك جانب مهم جدا لإنشاء النباتات في الوقت الحقيقي وهو القدرة على إنتاج مستويات متعددة من التفاصيل، لأن استخدام تمثيل هندسي مكلف للغاية، وليس مناسبا لإنشاء الأشجار البعيدة عن المراقب، ففي هذه الحالة النظام سوف يفقد الكثير من الوقت في إنشاء تفاصيل غير مرئية للمستخدم، لهذا يجب أن يكون النظام قادرا على إنتاج مستويات من التفصيل أقل تعقيدا لهذه الحالات.

Abstract

The representation of natural scenes and their rendering has always been a challenge for computer graphics, because of their indispensability to many applications such as ecosystems simulators, 3D mapping, flying simulators, video games ..., and because of their geometric complexity, due to very large number of small details they contain which are very difficult to model, and their representation is very expensive in memory, also their rendering requires a lot of computing time.

In recent years, graphics hardware has seen a huge development, with the introduction of GPU (Graphical Processing Units), their evolution has been adapted to intensive parallel computing, the development these units aimed to increase programming flexibility, by introducing shaders that are programs executable directly on the GPU, and the ability to locally generate new vertices, which allows enriching the geometric representations of objects in the scene without having to transfer more data between a CPU and GPU, and uses the GPU compute units to do so.

This huge development of GPU and these new opportunities may offer the ability for more involvement of geometry in real time rendering plants, if they are properly exploited.

To provide a method of plant rendering that maximizes the benefit of using recent GPUs retaining the possibility of using the geometrical representation of the plant on demand, we must first determine a rendering method that is adapted to the rendering very large amounts of geometry by pulling a maximum profit of GPU performances and then define a geometric representation that is best suited to this method of rendering, and then optimize the entire rendering process by exploiting the specific properties of this representation (similarities and pre-knowledge of the geometry).

A very important aspect as the rendering plants in real time is the ability of the application to produce multiple levels of detail, because using the geometrical representation is very expensive, and not suitable for rendering trees that are far from the observer, in this case the application will lose a lot of time rendering details that are not visible to the user, then the render method must be able to produce coarser levels of detail for this case.

Introduction générale………………………………………………… 1

Chapitre1: Etat de l'art sur la représentation de paysages naturels

1. Modélisation et représentation des terrains ……………………………….. 4
1.1. Génération des terrains ……………………………………………………… 4
1.2. Représentation des terrains ………………………………………………… 7
1.2.1 Les cartes d'élévation ……………………………………………………... 7
1.2.2 Les réseaux de polygones ………………………………………………….. 7
2. Représentation et modélisation des végétaux ………………. 8
2.1 Modèles structurels…………………………………………………….. 8
2.1.1 Structure ramifiée d'un végétale……………………………………… 8
2.1.2 Modèles à caractère géométrique ……………………………………... 9
2.1.3 Modèles fractales………………………………………………………... 10
2.1.4. Les modèles à base de paramètres botaniques……………………….. 12
2.1.5 Modèles à base de grammaires………………………………………… 14
2.1.6. Modèles combinatoires………………………………………………… 17
2.1.7. Travaux de Weber et Penn…………………………………………… 18
2.2. Modèles impressionnistes…………………………………………….. 20
2.2.1 Modèles à base de texture ……………………………………………… 20
2.2.2 Modèles dans un espace voxels………………………………………… 22
2.2.3. Modèles à base de systèmes de particules…………………………….. 23
2.2.4. Modèles a base de points……………………………………………….. 26
4. Modèles d'illumination à base d'image adaptés pour le rendu des arbres 26
4.1. Tranches d'images de profondeurs (Layer Depth Images (LDI)). 26
4.2 Rendu à base de textures volumiques ……………………………... 27
4.3 Rendu à base de couches d'images ……………………………… 28
Conclusion ……………………………………………………………………….. 29

Chapitre 2 : Les processeurs graphiques (Graphical Processing Units (GPU))

1. Introduction………………………………………………… 31
2. Pipeline graphique……………………………………….... 32
3. Evolution de la programmabilité des GPUs……………….... 34
4. GPUs Programmables……………………………………... 36
5. Architecture interne et Parallélisme des GPUs…………….... 37
5.1. Architecture général du GPU …………………………………… 38
5.1.1. Quelque exemples d'architecture des GPUs ………………………. 39
5.2. GPGPU ……………………………………………………………… 41
5.3. Architecture Tesla…………………………………………………… 42
5.3.1. Jeu d'instructions………………………………………….............. 41

5.3.2 .Organisation générale……………………………………………………. 42
5.3.3. Interface……………………………………………………………………… 44
5.3.4. Répartition du travail……………………………………………………… 44
5.3.5. Front-end…………………………………………………………………. 45
5.3.6. Hiérarchie de mémoire……………………………………………….. 46
6. Programmation GPU par shaders…………………………… 47
6.1. Vertex shader……………………………………………………………. 47
6.2. Geometry shader …………………………………………………... 48
6.3. Fragment shader ………………………………………………….. 48
6.4. Langages de programmation des GPU…………………………... 49
6.5. Optimisation de programmation des shaders……………………. 50
Conclusion ……………………………………………………………………. 50

Chapitre 3 : Instanciation de géométrie par GPU

Introduction……………………………………………………… 52
1. Concepts de l'instanciation de géométrie……………………. 52
2. Implémentations de l'instanciation de géométrie…………... 53
2.1 Statique batching…………………………………………………... 53
2.2 Dynamique batching……………………………………………….. 53
2.3 Instanciation par vertex constants………………….. ………. 54
2.4 batching en utilisant l'instanciation par API …………………….. 55
2.5 Pseudo instantiation…………………………………………………... 57
3. Analyse de performance……………………………………… 57
3.1. Taille des baches…………………………………………………… 57
3.2 Implication du CPU dans la tache de rendu……………………… 58
4. Conclusions ………………………………………………………… 59
4.1. Utilisation du géométrie shader ………………………………….. 59
4.2. Adaptation au rendu des arbres………………………………….. 61

Chapitre 4 : Rendu des arbres avec instanciation de Géométrie par GPU

I. Interprétation géométrique adaptée GPU des modèles de représentation d'arbres ………………………………………. 62
1. génération des primitives géométriques…………………….. 62
1.1Représentation des modèles……………………………………... 64
1.2 Représentation de branches…………………………………….. 64
1.3 Représentation des feuilles……………………………………… 64
2. Implémentation………………………………………………….. 65

2.1. Generation de geometrie……………………………………… 65
2.2. Implémentation des méthodes d'interprétation………………… 67
3. Sauvegarde des modèles pour le rendu……………………… 70
3.1 Les branches……………………………………………………... 70
3.2 Les feuilles……………………………………………………….... 71
II. Rendu d'arbres en utilisant l'instanciation de géométrie par GPU ………………………………………………………… 72
1. l'instanciation de géométrie……………………………………. 72
1.1. Gestion des bâches (lots de géométrie) …..………………………… 72
1.2. Les données d'instances…………………………………………….. 73
2. Procédé de rendu………………………………………………….. 73
2.1. Initialisations……………………………………………………………. 73
2.1.1. Attribut Id………………………………………………………………… 74
2.1.2. Next position et next normal ………………………………………… 75
3. Mécanisme du dessin………………………………………………. 76
3.1. Production de surfaces par géométrie shader ………………… 77
3.2. Aces aux données d'instances……………………………………… 79
3.3. Détermination de niveau de détails…………………………….... 80
3.4. Implémentation par géométrie shader………………………….. 81
4. Clping d'instances…………………………………………………….. 81
4.1. Application du cliping………………………………………………… 82
5. Traitement par modèle…………………………………………….. 83
5.1. Application du traitement par modèle ………………………. 84
5.1.1. Illumination approximative per-vertex……………………………. 84
5.1.2. Ombres générés par les feuilles sur les autres feuilles………………. 84
5.2. Animation d'arbres ………………………………………………. 85
6. Texturage……………………………………………………………… 85
7. Rendu de scènes avec un très grand nombre d'arbres…… 86
8. Terrain et ombres portées sur le terrain……………………… 86
8.1 Construction du terrain ……………………………………………. 86
8.2. Gestion des positions sur le terrain ……………………………… 87
8.3. Ombres …………………………………………………………………... 87
9. Algorithme de rendu ………………………………………………… 87
Conclusion ……………………………………………………………… 88

Chapitre 5 : Résultats et conclusion

1. Exemples de résultats de rendu…………………………………... 90
2. Performances…………………………………………………………. 91
2.1 teste de fps (image par seconde)………………………………… 91
2.2. Traitement par modèle …………………………………………….. 92

3. Conclusion ……………………………………………………………… 93
3.1. Limitations de portabilité ………………………………………… 93
3.2 : Perspectives pour l'interprétation en géométrie………………… 93
3.3 : Plus d'implication de techniques issues des modèles impressionnistes ………………………………………………… 94

Conclusion générale ………………………………………………….…... 95

Figure	Page
Fig1 : Reliefs produits par Gforge.	5
Fig2 : Application du principe de subdivision sur une carte d'élévation [3]	5
Fig3 : terrains générés par fractales [10]	6
Fig4 : trains générés par la méthode d'érosion [7]	6
Fig5 : contrôle local de la dimension des fractals pour l'érosion [7]	7
Fig6 : types de ramification possible pour un végétal [11]	8
Fig7 : Tendances de croissances [11]	9
Fig8 : Modes de croissance par bourgeons [11]	9
Fig9 : Génération d'une branche fille en utilisant un modèle géométrique [12]	10
Fig10 : exemples d'arbres générés par le modèle deAono [12]	10
Fig11 : Topologie sellent Oppenheimer [5]	11
Fig12 : Image produite par le modèle des IFS [5]	12
Fig13 : Structure ramifiée et bourgeons [4]	13
Fig14 : exemples de croissance a base de paramètres botaniques [42]	13
Fig15 : Résultats du rendu d'arbres générés a base de modèles botaniques [42]	14
Fig16 : Codage d'un arbre généré par L-system [5]	14
Fig17 : Application d'une règle de production a l'arrête s d'un arbre axial [5]	15
Fig18 : Un arbre généré par un L-system [5]	15
Fig19 : Exemples de simulation de croissance suivent les L-Systems [15]	15
Fig20 : Exemple de croissance avec contrainte de limitation de taille [11]	16
Fig21 : processus de génération L-System avec prise en charge des facteurs externes [15]	16
Fig22 : Un champ de tournesol produit par les L-Systems stochastiques [15]	17
Fig23 : Système de modélisation de plantes de Lintermann [16]	17
Fig24 : Ordre des nœuds dans un arbre [17]	18
Fig25 : Matrice de ramification des arbres parfaits et un arbre généré à partir d'elle [17]	18
Fig26 : Arbre binaire aléatoire croissant généré par matrice limite des arbres binaires aléatoires croissants. [17]	18
Fig27: génération d'arbres à base de modèle géométrique de Weber et Penn [18]	19
Fig28 : Exemples d'arbres générés par le modèle de Weber et Penn [18]	19
Fig29 : Image multi-échelle d'un arbre pour le rendu à 30, 60, 120, 240,600, et 1200 m [18]	20
Fig30 : modèle à base de texture 2D [19]	20
Fig31 : Ellipsoïdes Fractales [11]	21
Fig32 : utilisation des ellipsoïdes fractales pour la représentation d'arbres [20]	22
Fig33 : Croissance dans un espace voxel avec prise en charge de contraintes d'environnement extérieur [21]	23
Fig34 : Modélisation avec systèmes de particules [22]	24
Fig35 : Paramètres géométriques qui contrôlent la génération de particules [5]	24
Fig36 : Rendu d'arbres générés avec des systèmes de particules [23]	25
Fig37 : Résultats de représentation d'arbres à base de points [26]	26
Fig38 : Application de la technique LDI [37]	27
Fig39: Rendu à base de texture volumique [39]	28
Fig40 : Rendu à base de couches d'images [41]	29
Fig41 : Evolution des cartes graphiques [43]	31
Fig42 : Etapes du pipeline graphique [44]	32
Fig43 : Passage du repère local vers le repère global [44]	32
Fig44 : Transformation du repère global au coordonnés du repère de la camera [44]	33

Fig45 : Passage aux coordonnées normalisées [44]	33
Fig46 : Projection dans l'espace écran pour produire l'image 2D a affiché [44]	33
Fig47 : Découpage des primitives en fragments [44]	34
Fig48 : Les nouvelles entrées du GPU [43]	35
Fig49 : Intégration de Vertex shader et Fragment shader dans le pipeline graphique [43]	35
Fig50 : Localisation du géométrie shader dans le pipeline graphique [48]	36
Fig51: Différence architecturale entre CPU et GPU [51]	38
Fig52 : Comparaison des performances CPU et GPU [51]	38
Fig53 : Caractéristiques de quelques cartes graphiques [50]	38
Fig54 : Architecture générale du GPU[51]	38
Fig55 : Architecture du G70	39
Fig56 : Architecture du R 580 [51]	40
Fig57 : Architecture unifiée [51]	40
Fig58 : Architecture du G80 [51]	40
Fig59 : GPU pour les applications graphiques [51]	41
Fig60 : GPU pour GPGPU [51]	41
Fig61 : Révision des architectures NVDIA [50]	42
Fig62 : Vue générale de l'architecture Tesla [50]	44
Fig63 : Vue d'ensemble du pipeline d'exécution d'un GPU Tesla [50]	45
Fig64 : Environnement d'exécution du vertex shader [51]	47
Fig65 : Environnement d'exécution du Géométrie Shader [51]	48
Fig66 : Environnement d'exécution d'un fragment Shader [51]	49
Fig67 : Streaming pour l'instanciation en utilisant les vertex constants [54]	54
Fig68 : Algorithme d'instanciation de géométrie en utilisant les constantes des sommets [54]	55
Fig69 : Vertex buffers pour l'instanciation de géométrie par API [54]	56
Fig70 : Algorithme d'instanciation de géométrie par multi-streaming [54]	56
Fig71 : Influence de l'implication du CPU dans la tâche de rendu par instanciation de géométrie [55]	58
Fig72 : Comparaison de l'évolution de performances GPU et CPU [55]	58
Fig73 : Construction de cylindres	66
Fig74 : Construction d'ellipsoïde	67
Fig75 : Extraction de quatre sommets de l'ellipsoïde à partir de chaque point calculé	68
Fig76 : Résultats de la méthode déform	69
Fig77 :Résultats de plusieurs invocations de la méthode grew	70
Fig78 : Fusionnement des surfaces	75
Fig79 : Problème de saut de sommets pour l'élimination de surfaces.	76
Fig80: Utilisation des attributs next position et next normal pour le fusionnement des surfaces	77
Fig81 : Ordonnancement des sommets dans le vertex buffer	78
Fig82 : Mécanisme de dessin	79
Fig83 : Génération de coordonnées de texture	86
Fig84: Rendu de scène avec 220 arbres	90
Fig85 : Les ombres portées sur le terrain	90
Fig86 : Changement de position et intensité de la lumière	91
Fig87 : Performances en fps en fonction du nombre d'arbres	91
Fig88 : Impact de l'animation sur les performances	92

Introduction générale

La représentation des scènes naturelles et leurs rendu a toujours représenté un défi pour la synthèse d'image, à cause d'un côté de leur indispensabilité a beaucoup d'applications comme les simulateurs d'écosystèmes, la cartographie 3D, les simulateurs de vols, jeux de vidéo …, et d'un autre côté de leur complexités géométrique, due au nombre très grand de petits détails qu'ils contiennent qui sont très difficiles a modéliser et a représenter, et très couteux pour le stockage en mémoire, et aussi leur rendu qui nécessite un temps de calcule très élevé.

Plusieurs méthodes ont étés présentées pour traiter cette complexité, en abordant deux aspects : la modélisation et le rendu.

Les techniques qui traitent l'aspect de modélisation des végétaux forme un ensemble de modèles appelé les modèles structurels, ces modèles ont un point commun qui est l'existence d'une représentation géométrique des végétaux.

Les études sur la croissance et l'évolution des végétaux a fourni une très grande base d'informations et une source d'inspiration pour les techniques de modélisation des végétaux pour la synthèse d'image, permettant de les rendre de plus en plus réaliste, mais la quantité de géométrie générée par ces méthodes reste très couteuse en mémoire et en temps de calcul pour les scènes naturelles qui contiennent de grande quantité de végétation, et inadéquate pour le rendu temps-réel.

Les techniques qui traitent l'aspect de rendu, sont appelées les modèles impressionnistes, ces modèles n'utilisent pas une représentation géométrique des végétaux, mais se concentrent sur l'aspect visuel des résultats, en produisant des images qui donnent à l'observateur l'impression d'existence des végétaux. Ces modèles donnent une solution très convenable pour le rendu temps réel par ce qu'ils ne traitent pas la complexité géométrique des végétaux, qui peuvent même ne pas être construits dans l'espace 3D.

Les modèles impressionnistes offrent des images très convaincantes dans les cas des scènes ou l'observateur ne se rapproche pas des végétaux (par exemple le cas de simulateurs de vol), mais ils sont incapables de traiter les scènes qui offrent la possibilité de navigation rapprochée des végétaux (par exemple la navigation à l'intérieur des arbres d'une forêt).

Une grande portion de la complexité des scènes naturelles provient des arbres, qui nécessitent un très grand nombre de primitives pour la représentation de leurs feuilles, et un temps de calcule énorme pour leur rendu en utilisant les méthodes de rendu et de représentation classique. Dans ce travaille nous allons nous intéresser au rendu des arbres, en essayant d'offrir une solution pour la navigation rapprochée des scènes naturelles.

Les modèles structurelles sont les modèles les plus adéquats pour la navigation rapprochée des scènes comportant des arbres, par ce qu'ils offrent la structure géométrique complète pour le rendu proche des arbres, mais pour les utiliser pour un rendu temps réel il faut résoudre les problèmes engendrés par la complexité géométrique qu'ils traitent.

Le choix de traiter les structures géométriques malgré leurs coût en traitement et en mémoire est motivé par le développement énorme des GPUs récents qui deviennent de plus en plus puissants en capacité de calcul et en degré de parallélisme qu'ils offrent, et de plus en plus flexibles en programmabilité.

Dans ce travail nous allons présenter une solution qui permet l'utilisation des modèle structurelle pour le rendu temps réel de scènes peuplées d'un grand nombre d'arbres, en exploitant les capacités des GPUs récents, nous avons travaillé sur les aspects suivants :

- La représentation des végétaux avec de la géométrie souffre d'un grand problème de cout en mémoire, nous avons adopter une technique d'instanciation de géométrie qui permet de réduire énormément la quantité de mémoire nécessaire pour stocker les arbres, en ne stockant que des modèles d'arbres qui seront dupliqués plusieurs fois, au lieu de stocker pour chaque arbre dans la scène sa propre structure géométrique.
- Les capacités énorme des GPUs peuvent être mal exploités pour des raisons de dépendance aux capacités du système sur lequel ils sont installés a les alimentés en géométrie, nous proposons une technique d'instanciation de géométrie qui diminue au maximum la dépendance du GPU au reste de système en diminuant a une quantité très petite les informations transférés entre le système et le GPU, et en minimisant l'implication du CPU dans la tâche de rendu.

Ce problème a été toujours présent avec les techniques d'instanciation de géométrie, par ce que la génération des nouveaux sommets se faisait par API sur le CPU, et puis ils sont transmis au GPU, en modifiant un attributs de tous les sommets du modèle géométrique pour pointer sur de nouvelle données d'instance par CPU et transférer les nouveaux attribut au GPU, ce qui résulte en un temps de transfert très grand dans le cas des modèles de grande tailles.

La méthode d'instanciation de géométrie que nous proposons exploite la capacité des GPUs récents de produire de nouveaux sommets par géométrie shader pour éliminer tout le transfert relatifà l'instanciation des sommets.

- Les modèles structurels fournissent des moyens suffisants pour représenter tout espèce d'arbre avec un réalisme très convaincant, mais leurs interprétations en géométrie produisent des représentations très complexes, et difficiles à gérer, dans ce travail nous proposons un procédé

permettant d'interpréter en géométrie n'importe quel modèle, dont les résultats sont adaptés à l'instanciation de géométrie, et à maximiser l'exploitation du GPU.

- le rendu des arbres par instanciation de géométrie offre aussi un concept très bénéfique en temps de calcul qui est le traitement par modèle, ce concept donne la possibilité d'appliquer des traitements par vertex shader sur les modèles et après appliquer les résultats directement aux instances de chaque modèle.
- Plusieurs travaux ont proposé des modèles d'illumination pour les arbres, comme le modèle probabiliste de Chaudy [11], et de Boulanger [56], qui peuvent être appliqué per-vertex, et avec le traitement par modèle ils peuvent bénéficier d'un gain très important en temps de calcul.
- Quel que soit les capacités du GPU, il y a toujours une limitation au nombre de primitives qu'il peut traiter en temps réel, mais d'un autre coté quel que soit le nombre d'arbres dans une scène, dans le cas de navigation rapprochée le nombre d'arbres à l'intérieur de la pyramide de vision est limité, alors nous proposons une technique d'illumination d'instances qui ne sont pas visible qui permet de dépasser cette limitation.

Pour répondre à cette problématique, nous avons organisé notre mémoire en cinq chapitres, dans le premier chapitre, nous présentons un état de l'art sur la représentation de paysages naturels, où nous décrirons la représentation et le rendu des terrains et des végétaux et leurs interactions à la lumière ainsi que la manière dont ils interagissent entre eux. Dans le deuxième chapitre, nous présentons les GPUs, leur évolution technologique, le pipeline graphique 3D et surtout l'aspect programmabilité. Le chapitre trois est dédié à l'instanciation de la géométrie qui permet de dupliquer un modèle géométrique (un mesh, un ensemble de sommets...) plusieurs fois à partir des attributs des instances. Dans les deux chapitres suivants nous présentons l'essentiel de nos contributions, ainsi dans le quatrième chapitre nous abordons le rendu des arbres avec instanciation de Géométrie par GPU et le chapitre cinq présente les résultats, l'évaluation de la technique proposée et quelques perspectives.

Chapitre 1:

Etat de l'art sur la représentation de paysages naturels

Chapitre 1

Etat de l'art sur la représentation de paysages naturels

Le traitement des scènes naturelles en synthèse d'image traite principalement la représentation et le rendu des terrains et des végétaux et leurs interactions à la lumière ainsi que la manière dont ils interagissent entre eux.

1. Modélisation et représentation des terrains :

La modélisation des terrains est un aspect très important dans plusieurs applications, la géographie, les jeux vidéo, les simulateurs de vols, l'aménagement du territoire etc.

La complexité de modélisation et représentation des terrains provient de deux aspects essentiels, le premier aspect concerne les modèles de la génération et l'amplification des données qui définissent le terrain, et le deuxième est les techniques adéquates pour le rendu de ses modèles.

1.1. Génération des terrains :

Les techniques de génération des terrains peuvent être classées par l'ordre chronologique de leurs apparitions en 3 catégories : [5]

- Techniques à base de plaquage de texture : ces techniques mettent en évidence la technique de rendu et de texturage plutôt que la technique de génération, et elles sont utilisées jusqu'à aujourd'hui pour rendre des paysages générés à base de fractales ou de prises de mesures.
- Méthodes d'érosion : utilisées pour améliorer le réalisme des paysages synthétisés, et utilisées dans des parties de relief où on veut faire apparaitre le phénomène d'érosion, leur utilisation peut être combinée avec l'utilisation des fractals ou les systèmes de particules.
- Méthodes fractales : utilisent les fonctions fractales pour la génération des terrains.

Selon la nature du terrain artificiel ou réel, on distingue deux familles de techniques [6]:
- Méthodes extrayant des données réelles issues de masure de reliefs (méthodes numériques) :Ces méthodes se basent sur l'utilisation des Modèles numérique du terrain (MTN) issu des masures réels de reliefs.

Ces méthodes souffrent d'un problème de prise de mesures et stockage de la très grande quantité d'information.

- Méthodes de génération de relief artificiel :

Méthode basée sur les fractals, développée par Mandelbrot. [1], qui a remarqué la similitude entre la forme d'une crête de montagne et la courbe produite par un mouvement Brownien fractionnaire (*fBm*).
Un mouvement Brownien est une suite de petits déplacements indépendants et isotropes (toutes les directions ont la même probabilité).
Un mouvement Brownien fractionnaire est la généralisation du mouvement Brownien correspondant à une dimension non entière (fractale). Mandelbrot propose Trois méthodes pour calculer une approximation discrète du fBm en plusieurs dimensions.

- Les champs de Poisson fractionnaires (fractional Poisson eld).
- les approximations d'un bruit Gaussien discret (discrete fractional Gaussian noise)
- la transformation de Fourier d'un bruit blanc. [7]

J. Beale : propose dans son logiciel Gforge [8] une implémentation de ces techniques, Gforge utilise les techniques de synthèse spectrale par transformée de Fourier inverse décrites par Voss [9].

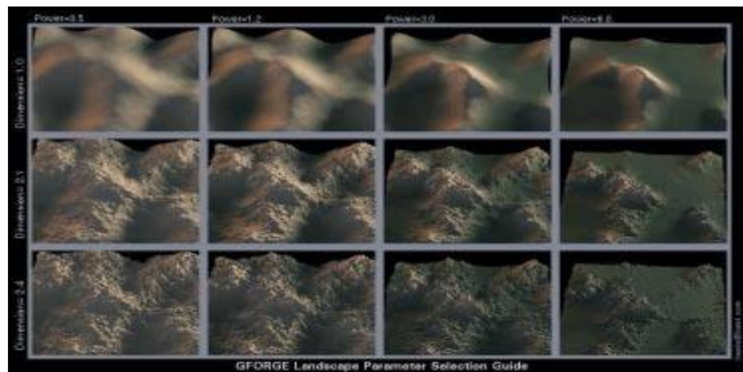


Fig1 : Reliefs produits par Gforge.

Fourier propose une méthode algorithmique pour le calcul approché du « fBm »baséesur une subdivision récursive du modèle et sur l'introduction d'un facteur aléatoire : pour chaque intervalle non subdivisé, on calcule le point milieu auquel on applique une perturbation aléatoire dont l'amplitude est proportionnelle au niveau de récursivité.[3]

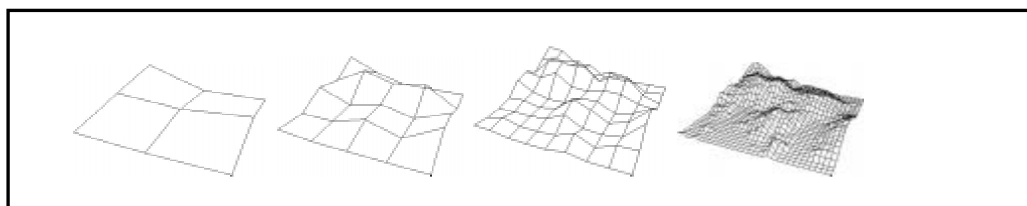
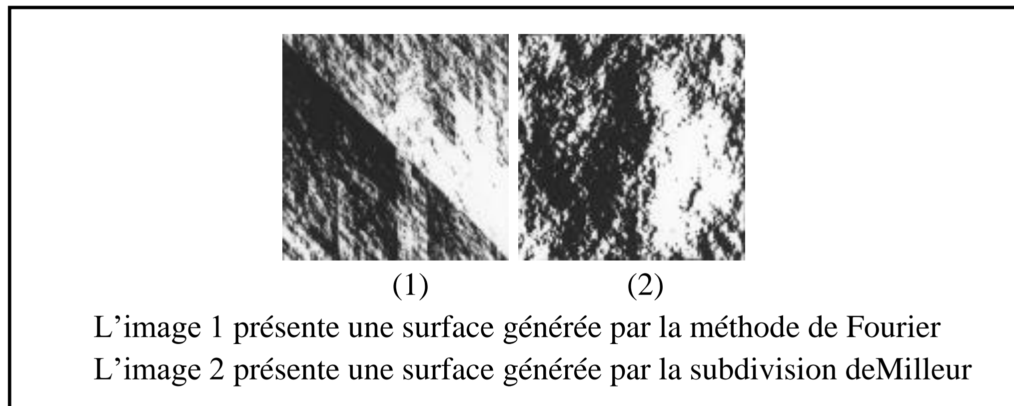


Fig2 : Application du principe de subdivision sur une carte d'élévation [3]

Mandelbrot souligne que cette méthode n'est qu'une approximation et ne possède pas les caractéristiques d'isotropisme ou d'autosimilarité des fBm, ce qui provoque certains défauts sur les surfaces générées.

Miller [10] propose une autre technique de subdivision pour résoudre ces problèmes et générer des résultats plus réalistes, cette technique prend en compte pour la subdivision un grand nombre d'échantillons utilisé pour une subdivision pondérée.

Les inconvénients majeurs de cette méthode sont de deux ordres : la surface générée ne passe plus par les points de contrôle et les grilles successives sont de tailles différentes.


(1) (2)

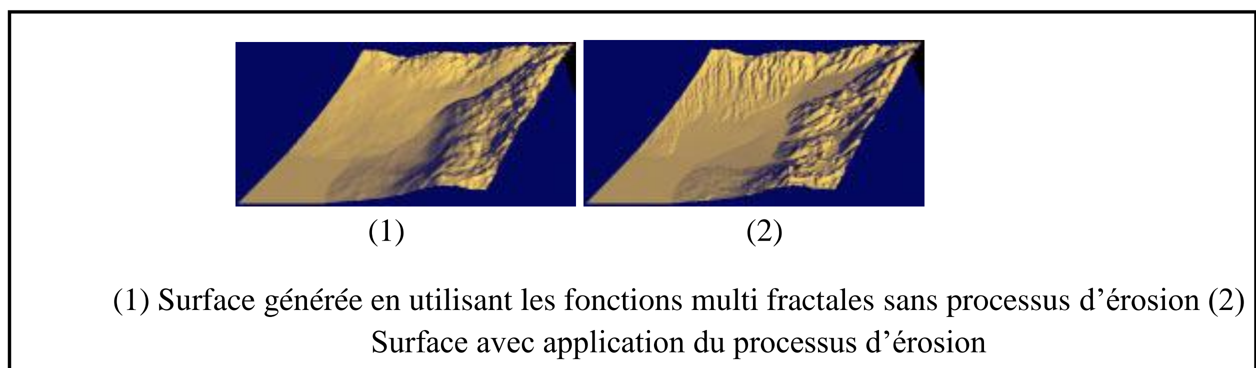
L'image 1 présente une surface générée par la méthode de Fourier
L'image 2 présente une surface générée par la subdivision deMilleur

Fig3 : terrains générés par fractales [10]

Les méthodes à base de subdivisions récursives donnent des résultats qui ne paraissent pas toujours très réalistes dans la génération de relief. Il apparait difficile de faire varier la rugosité de la surface ou d'effectuer un contrôle local sur les caractéristiques du terrain. [11]

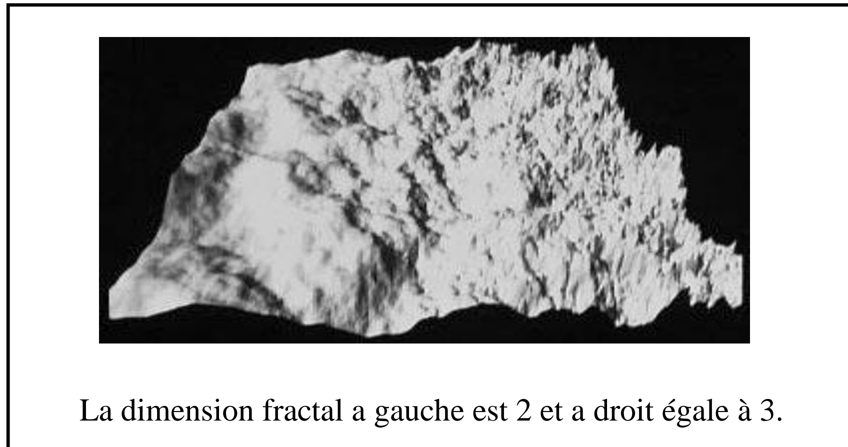
Musgrave [7] apporte deux améliorations pour réduire l'uniformité dans les formes produites par les méthodes à base de subdivision récursives, il propose un processus de génération à deux étapes :

– l'utilisation des fonctions multi-fractales, qui permettent de prendre en compte les variations hétérogènes d'un même terrain, pour la génération d'un terrain fractal.
La mise en œuvre des fonctions multifractales est réalisée par la somme ou le produit de fonctions monofractales comme les fbms, ce qui permet d'utiliser de faibles dimensions fractals pour les vallées, et d'autres dimensions pour les montagnes.
– l'ajout d'un processus d'érosion basé sur un modèle hydraulique : l'eau ruisselle sur le relief en déposant des sédiments ou en érodant la matière.
Cette méthode étant basée sur le calcule de somme de produit des fbm, chaque point est indépendant et peut être évalué sans contraintes, il est possible de contrôler localement la dimension fractal et le facteur d'échelle du terrain (le rapport entre la largeur et la hauteur du terrain).


(1) (2)

(1) Surface générée en utilisant les fonctions multi fractales sans processus d'érosion (2) Surface avec application du processus d'érosion

Fig4 : trains générés par la méthode d'érosion [7]



La dimension fractal a gauche est 2 et a droit égale à 3.

Fig5 : contrôle local de la dimension des fractals pour l'érosion [7]

Cette technique soufre du problème des temps de calcul très long mais les résultats sont très réalistes.[11]

1.2 Représentation des terrains :

Deux structures de données sont essentiellement utilisées pour la représentation des terrains, les cartes d'élévation et les réseaux de polygones. [11]

1.2.1 Les cartes d'élévation :

C'est la structure de données la plus utilisée dans la représentation des terrains, une carte d'élévation est une grille à deux dimensions qui associe à chaque couple (u, v) de coordonnées dans le plan, une altitude z (u, v) et on peut aussi associer une couleur à chaque case, obtenant ainsi une texture pour le terrain.

Cette structure présente l'avantage de la simplicité et la facilité d'utilisation, qui provient de la possibilité de considérer ces cartes comme des images 2D dont la manipulation est très intuitive et la génération de niveaux de détailles est facile. [6]

1.2.2 Les réseaux de polygones :

Dans cette structure les hauteurs d'une carte d'élévation sont représentées par un maillage de triangles, les maillages de triangles sont plus compacts que les grilles de cartes d'élévation, et il existe beaucoup de moteurs de rendu spécialisés dans le rendu rapide des polygones.
La difficulté de cette structure est la construction du maillage triangulaire (triangulated irregular network ou TIN), il est important de trouver un compromis entre la précision du maillage (le niveau de détail sur le terrain) et le nombre de triangles.
Garland [6]propose une classification pour les techniques de construction de maillage :

- Les grilles uniformes : c'est un échantillonnage régulier des altitudes selon deux axes.
- Les subdivisions hiérarchiques : application des quadtrees ou des k-d trees.
- La recherche des singularités : les caractéristiques importantes du terrain (sommet, falaise, vallée ...) sont les nœuds du maillage.
- Le raffinement du maillage : à partir d'une approximation minimale, de nouveaux points sont créés jusqu'au maillage final.

-La simplification du maillage : à partir du maillage exhaustif de toutes les latitudes, le réseau est simplifié.

-Les autres méthodes basées sur les techniques d'optimisations.

2. Représentation et modélisation des végétaux :

Le traitement de la représentation et la génération des végétaux est un aspect très important dans la modélisation de paysages, du faite de leur présence dans la grande majorité des paysages, sous forme d'herbe, de buissons, d'arbustes ou bien d'arbres.

On distingue essentiellement deux grandes catégories de modèles pour la synthèse d'image des végétaux : [11]

- Modèles structurels : ces modèles proposent des méthodes pour la construction de la structure de l'objet, et puis l'interprétation de ces structures en descriptions géométriques 3D, en fin le rendu pour produire des images finale est appliquer sur ces description en tenant compte des paramètres de l'entérinement (position de l'observateur, sources lumineuse …etc.)
- Modèles impressionnistes : dans ces modèles les processus de modélisation et rendu sont confondus, ici on prend en compte l'aspect visuel des objets au moment de leur construction, qui diminue la complexité structurelle qu'on doit traiter.

2.1 Modèles structurels :

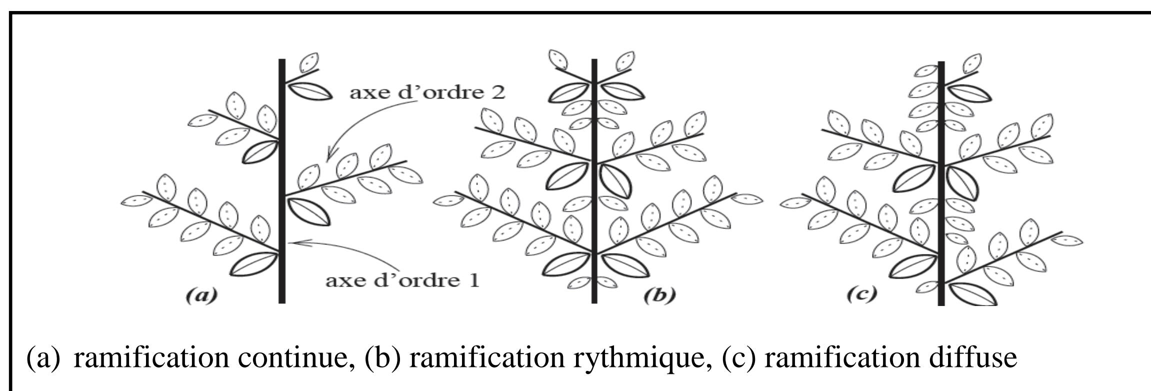
Ces modèles utilisent des méthodes mathématiques ou algorithmiques pour la génération de la structure ramifiée du végétale, cette structure dépend de l'espèce du végétale, puis interpréter cette structure en représentation géométrique, et enfin le rendu topologique pour obtenir les images voulus.

2.1.1 Structure ramifiée d'un végétale :

La structure d'un végétal est le résultat de la croissance de ses axes, les bourgeons qui apparaissent sur ces axes peuvent, soit continuer la croissance de l'axe soit donner naissance à une feuille, ou bien créer un nouvel axe, ce processus et appelé ramification.

La ramification peut prendre essentiellement trois directions : [11]

- la ramification continue : dans ce cas, chaque bourgeon donne naissance à un nouvel axe latéral.
- la ramification rythmique : seuls certains bourgeons engendrent régulièrement de nouveaux axes.
- la ramification diffuse : seuls certains bourgeons aléatoirement repartis donnent naissance à un axe

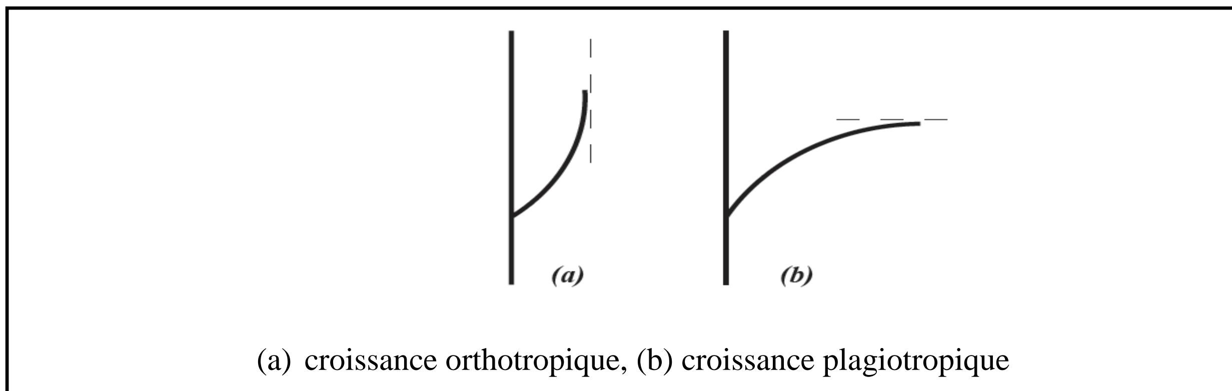


(a) ramification continue, (b) ramification rythmique, (c) ramification diffuse

Fig6 : Types de ramification possible pour un végétal [11]

Le type de ramification est fonction du niveau de l'axe, et diffère d'une espèce de végétale à une autre et suivent l'espèce l'axe aussi peut se développé de deux façons :

- Croissance orthotropique : la tendance de développement de l'axe est verticale.
- Croissance plagiotropique : la tendance de développement de l'axe est horizontale

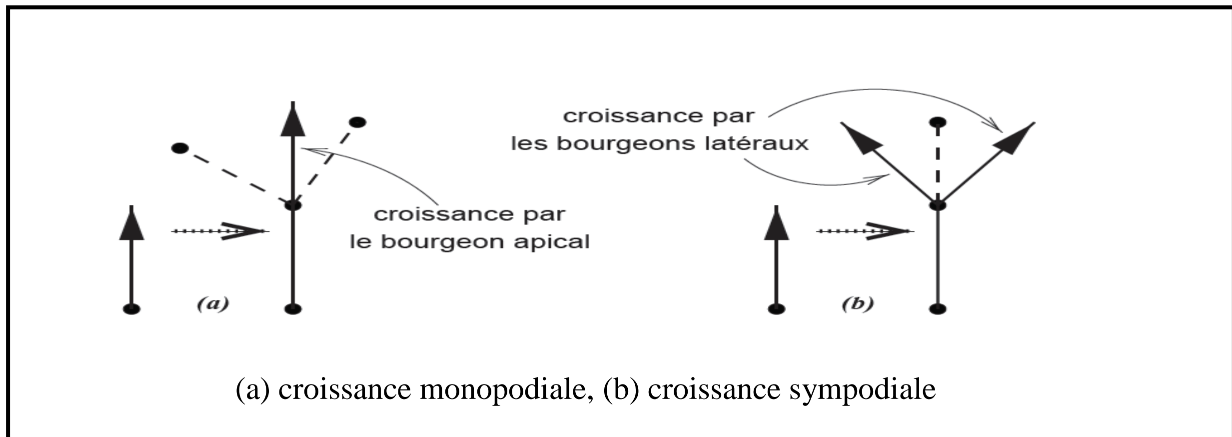


(a) croissance orthotropique, (b) croissance plagiotropique

Fig7 : Tendances de croissances [11]

On peut aussi remarquer deux modes de croissance possible en chaque bourgeon :

- Croissance monopodiale : le résultat de la croissance du bourgeon et la continuation verticale de l'axe sur lequel il est situé (croissance par bourgeon apical).
- Croissance sympodiale : le résultat de la croissance du bourgeon est deux nouveaux axes (croissance par bourgeons latéraux)



(a) croissance monopodiale, (b) croissance sympodiale

Fig8 : Modes de croissance par bourgeons [11]

2.1.2 Modèles à caractère géométrique :

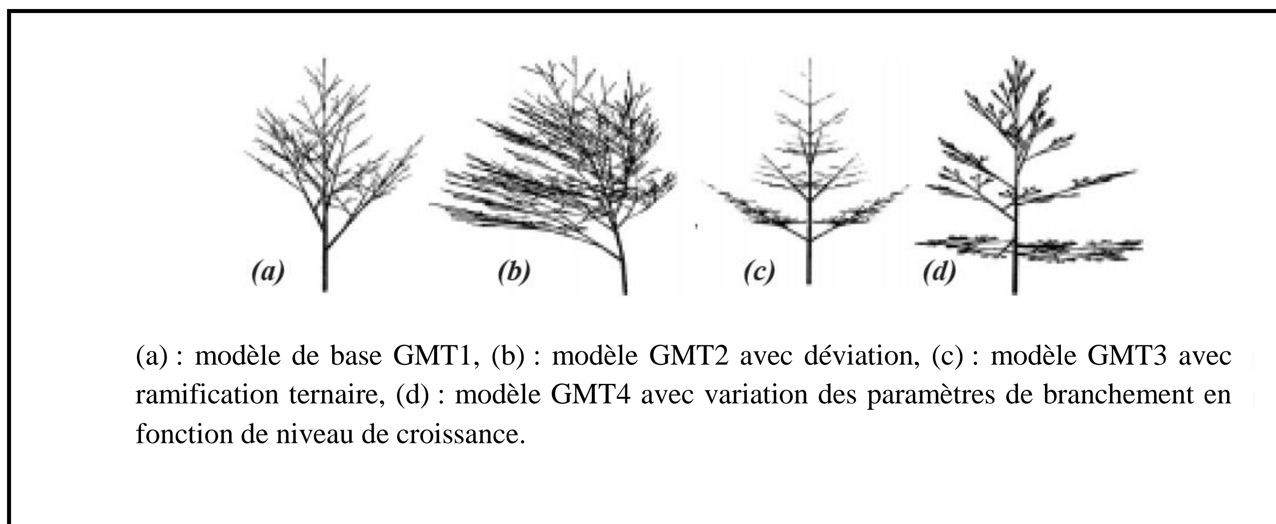
Dans ces modèles les caractéristiques métriques de l'arborescence sont décrites par un certain nombre de paramètres géométriques comme les angles d'embranchement ou les rapports des longueurs des branches [11].

Aono et Kunii [12] proposent de modèles géométriques qui calculent la direction et les caractéristiques des ramifications à partir des équations mathématiques dont les paramètres sont déduits directement de l'observation de la morphologie d'arbres réels.

Ces modèles se sont étendus pour prendre en compte l'effet d'influences externes comme le vent, le soleil et la gravite. De plus ils permettent des ramifications de type binaire et ternaire (à chaque ramification, deux ou trois branches filles sont créées) et des variations complexes des angles de branchement (en fonction de la distance au sol par exemple).

Pi=P+Ri*F(hi)
Avec :
Pi : position de la i eme branche fille
P : la position de la branche mére
Ri :le facteur de contraction de la branche fille par rapport à la branche mére
hi : les angles de branchement

Fig9 : Génération d'une branche fille en utilisant un modèle géométrique [12]



(a) : modèle de base GMT1, (b) : modèle GMT2 avec déviation, (c) : modèle GMT3 avec ramification ternaire, (d) : modèle GMT4 avec variation des paramètres de branchement en fonction de niveau de croissance.

Fig10 : exemples d'arbres générés par le modèle de Aono [12]

2.1.3 Modèles fractales :

Oppenheimer [13] propose un modèle fractal pour générer des structures de plantes et de structures non-organiques comme les deltas des fleuves ou les flocons de neige. Il implémente ce modèle par un programme récursif qui, à chaque itération, associe au nœud courant un certain nombre de sous-arbres transformés par une application linéaire (une simple matrice 3x3).

Cette récursivité est arrêté à un certain niveau prédéfinis et une feuille est produite.

Les paramètres géométriques de transformation des branches filles sont la rotation et le changement d'échelle, en plus, une information sur la forme des branches entre les nœuds de ramification (trajectoire rectiligne, spirale ou hélicoïdale).

Pour permettre la diversité et diminuer l'autosimilarité de ce modèle, Oppenheimer spécifie pour chacun des paramètres, une valeur moyenne et un écart type. Pour cet auteur, ces facteurs contrôlant la topologie et la géométrie sont semblables à des paramètres génétiques qui définissent la forme et l'évolution des espèces générées.

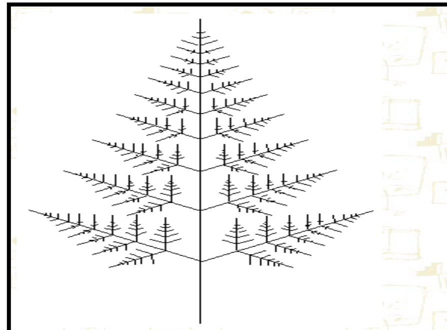


Fig11 : Topologie selon Oppenheimer [5]

La génération suivent le modèle d'Oppenheimer est réalisé partir de quatre primitives géométriques simples : cylindre, spirale, hélice et tortillon, de règles de réduction et de collage faisant intervenir les paramètres suivants : [5].

- l'angle entre un segment principal et les arêtes qui s'y connectent.
- le rapport de longueur entre les arêtes principales d'un branchement et les branches qui s'y connectent.
- le rapport de décroissance des arêtes sur un segment.
- la valeur de la rotation hélicoïdale sur les branches.

Oppenheimer [11]utilise pour l'interprétation géométrique et le rendu des prismes polygonaux avec une texture d'écorce pour les branches les plus importantes. La texture est créée par bumpmapping (perturbation de la normale en chaque point de la surface) à l'aide d'une fonction de bruit fractal.

Les systèmes de fonctions itérées (IFS) :

Un modèle développé par Démo, Hognes et Taylor, basé sur une technique purement mathématique fondée sur des théorèmes ergodiques.

IFS : A tout ensemble fini de fonctions affines strictement contractantes {wn, 1=n=N} du plan, appelé système de fonctions itérées (IFS pour Iterated Function System), on associe la transformation W qui à B compact du plan associe :

$$W(B)=.\bigcup_{n=1}^{N} Wn(B).$$

Les images obtenues représentent souvent des objets à forte caractéristique auto-similaire (feuilles d'érable, fougères). Dans les exemples donnés (images de la côte de Monterey, de la forêt noire, d'un champ de tournesols…, le nombre d'applications affines des IFS est de 160 à 180. Ces nombres sont réduits à environ 100 par l'utilisation de la notion de hiérarchie d'IFS avec ensemble de condensation, correspondant à la hiérarchie des éléments végétaux.

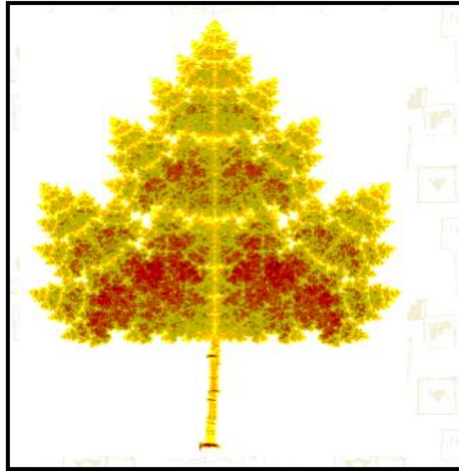


Fig12 :Image produite par le modèle des IFS [5]

Il apparait difficile avec l'outil mathématique (les courbes fractales), de générer un ensemble de modèles précis de végétaux. La diversité des espèces s'accommode mal avec l'autosimilarité stricte des fonctions fractales et la structure d'un végétal ne peut pas être facilement décrite par le faible nombre de paramètres de ces outils. [11].

2.1.4. Les modèles à base de paramètres botaniques :

Ces modèles sont des modèles physiques basés sur la simulation du développement des plantes dans une échelle du temps discrète.

Le modèle botanique de Reffye, Edelin, Françon, Jaeger, Puech [4], simule la croissance de végétaux au moyen de règles purement botaniques extraites d'analyses empiriques réalisées sur le terrain.

L'élément déterminant dans la structure d'un arbre est l'axe, dont la croissance de pend d'un bourgeon situé à son extrémité (bourgeon apical), cette axe se constitue d'une série d'inter-nœuds, créés les uns après les autres, Entre deux inter-nœuds on trouve un nœud qui donne naissance à des feuilles et à des bourgeons latéraux. Les bourgerons latéraux donnent naissance à de nouveaux axes (ramification).

Ainsi l'arbre est généré par des successions du processus de ramification, cette ramification peut être continue, rythmique ou diffuse.

Un instant donne de la simulation du développement de l'arbre un bourgeon peut évoluer vers l'un des quatre états suivants :[11]

- le bourgeon se transforme en fleur et meurt ;
- le bourgeon suspend son activité pendant une unité d'horloge ;
- le bourgeon devient un inter-nœud composé de feuilles, de nouveaux bourgeons latéraux et d'un bourgeon apical ;
- le bourgeon meurt.

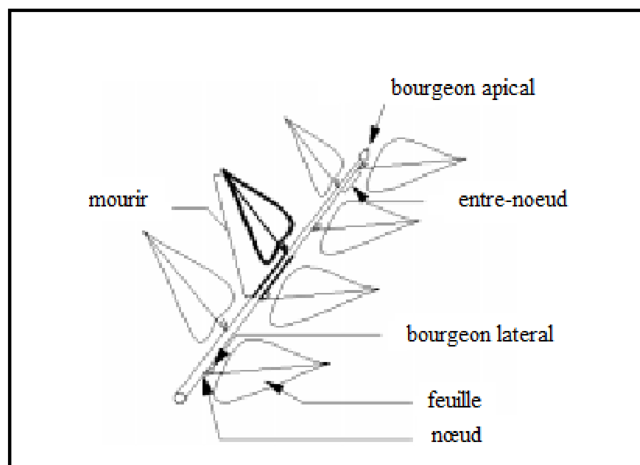
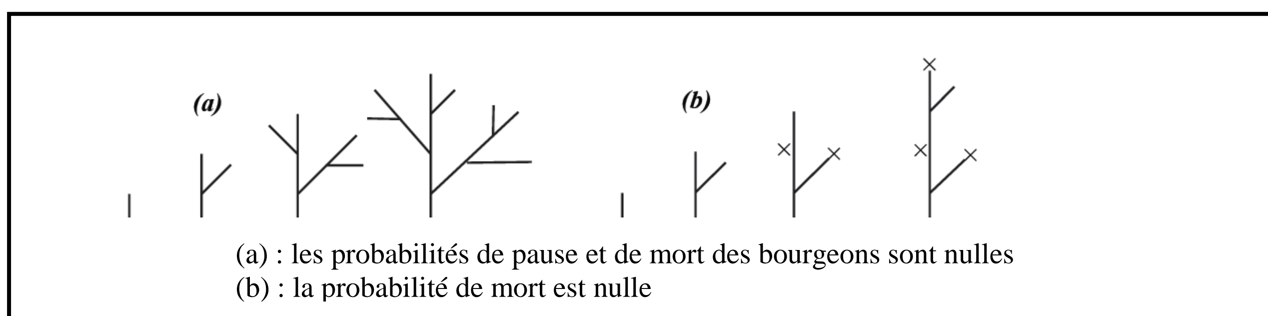


Fig13 : Structure ramifiée et bourgeons [4]

Des probabilités sont associées à chaque état futur du bourgeon, en fonction du niveau du développement de l’arbre, et de la position du bourgeon, ce modèle nécessite aussi d’autres paramètres issus d’études réels sur les différentes espèces, comme la vitesse de croissance et le nombre de bourgeons dans chaque nœud.

Des facteurs externes peuvent être facilement intégrés, comme par la coupe des branches (tous les bourgeons d'une branche meurent) ou l'effet de la gravite sur les branches (en utilisant un paramètre d'élasticité dépendant de la nature du bois).



(a) : les probabilités de pause et de mort des bourgeons sont nulles
(b) : la probabilité de mort est nulle

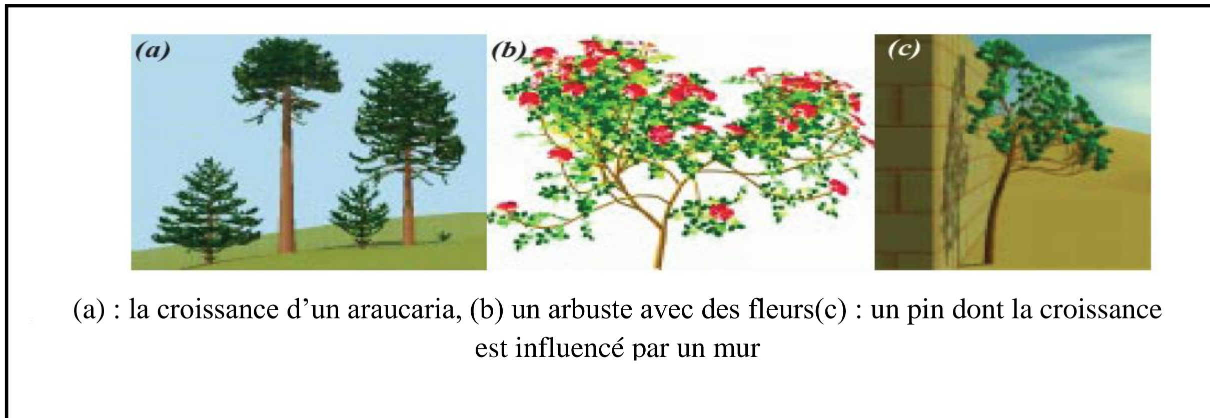
Fig14 : exemples de croissance à base de paramètres botaniques [42]

La description résultante de la simulation de croissance est ensuite interprétée en un ensemble de primitives géométriques simples, par exemple tronc de cône pour les branches, et des polygones pour les feuilles, qui seront traitées par un système de rendu adéquat.

Dans un premier temps on peut utiliser le Z-buffer pour réaliser l'élimination des parties cachées et des méthodes simples d'illumination (éclairage Lambertien), cependant n'importe quel système de rendu peut être utilise.

Ce modèle très complet et très réaliste de simulation et de visualisation de la croissance des végétaux a abouti à un système de simulation commercialisé par le CIRAD (Centre de coopération Internationale en Recherche Agronomique pour le Développement) sous l'appellation AMAP. Il

fonctionne sur stations de travail Silicon Graphics et l'interface avec des logiciels de synthèse d'images. [11]



(a) : la croissance d'un araucaria, (b) un arbuste avec des fleurs(c) : un pin dont la croissance est influencé par un mur

Fig15 : Résultats du rendu d'arbres générés à base de modèles botaniques [42]

2.1.5 Modèles à base de grammaires :

Linden mayer [14] a proposé en 1968 un modèle de développement de structures biologique basé sur un ensemble de règles de réécriture appelé L-system, chaque règle se composant de deux parties un prédécesseur et un successeur, le processus de génération est réalisé successivement par le remplacement parallèle des prédécesseurs par les successeurs correspondant aux règles applicables dans chaque itération. Cette succession d'itérations représente la structure ramifiée du végétal.

Cette modélisation topologique est fondée sur la notion d'arbre axial : [5]

- A chaque nœud on distingue au plus une arête fille appelée arête principale, les autres arêtes issues de ce nœud étant appelées arêtes latérales.
- Un *axe* est une sous-suite maximale d'arêtes commençant par l'arête issue de la racine ou bien par une arête latérale, et continuant par une sous-suite maximale d'arêtes principales.
- Le sommet terminal de l'axe issu de la racine est appelé sommet de l'arbre.
- Une branche est l'ensemble constitué d'un axe et de tous ses axes descendants dans l'arbre, une branche est elle-même un arbre axial.

Des valeurs numériques sont attribués aux axes et aux arêtes appelés ordres :

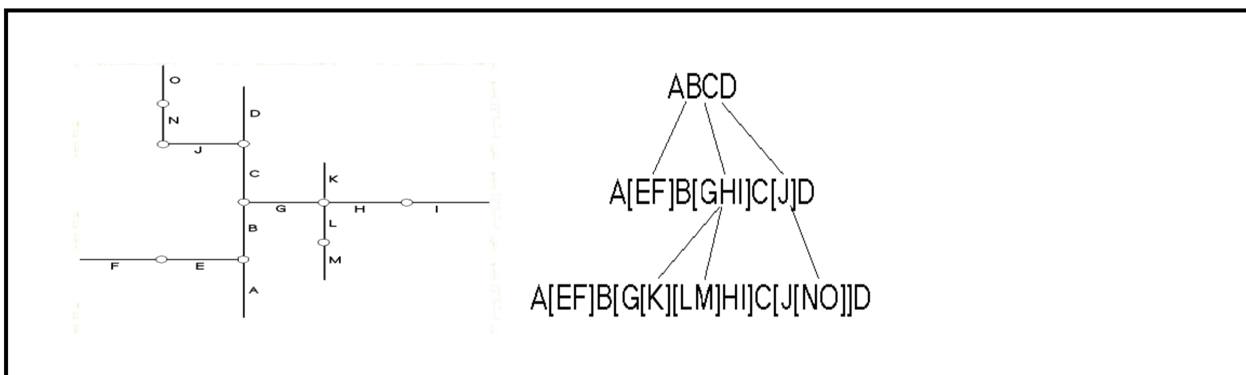


Fig16 : Codage d'un arbre généré par L-system [5]

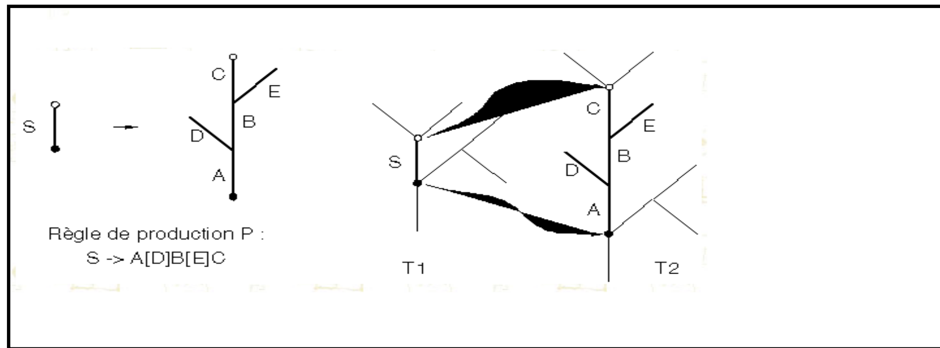


Fig17 : Application d'une règle de production a l'arrête s d'un arbre axial [5]

- L'axe dont l'arête racine est la racine de l'arbre a l'ordre 0.
- Les axes d'ordre i commencent par une première arête latérale généré a praire d'un axe d'ordre i-1.

Alors un L-système peut être définit comme un triplet (V,w,P) où :

- V est un alphabet,
- w est un mot de V appelé axiome représentant un arbre axial,
- P est un ensemble de règles de production associant à chaque lettre de l'alphabet V un arbre axial.

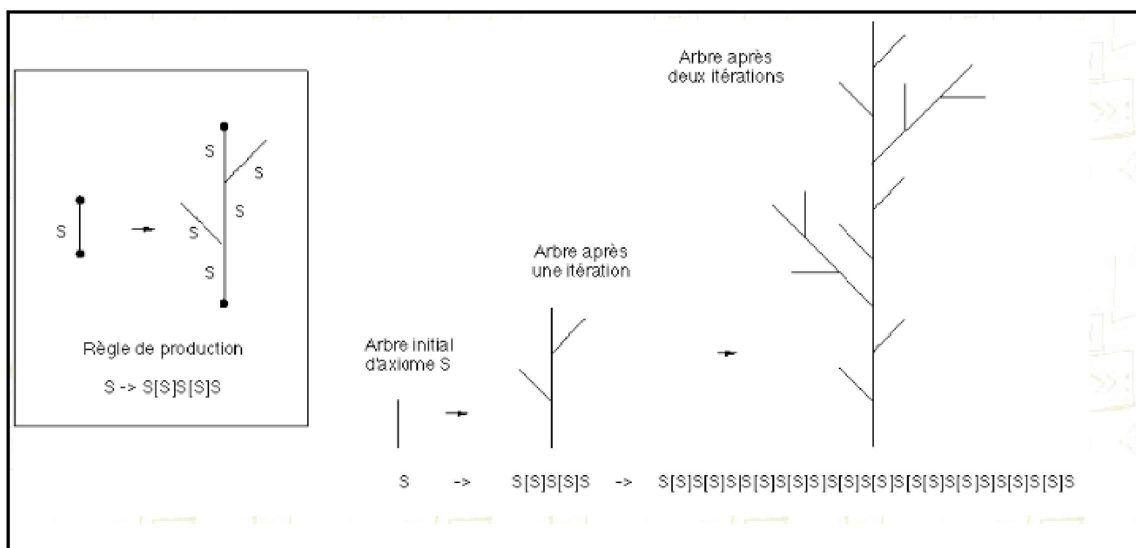


Fig18 : Un arbre généré par un L-system [5]

Des simulations très complexes, pour divers espèces de végétaux ont été possible grâce a la possibilité de propagation de signaux, et l'héritage fournit par le L-Systems.

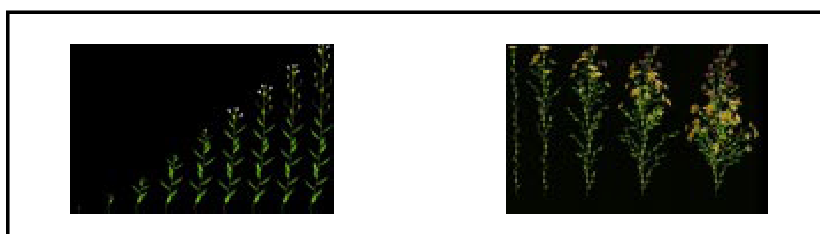


Fig19 : Exemples de simulation de croissance suivent les L-Systems [15]

Influence de l'environnement :

Il existe deux types de facteurs qui influencent sur la croissance d'un végétal : [11]

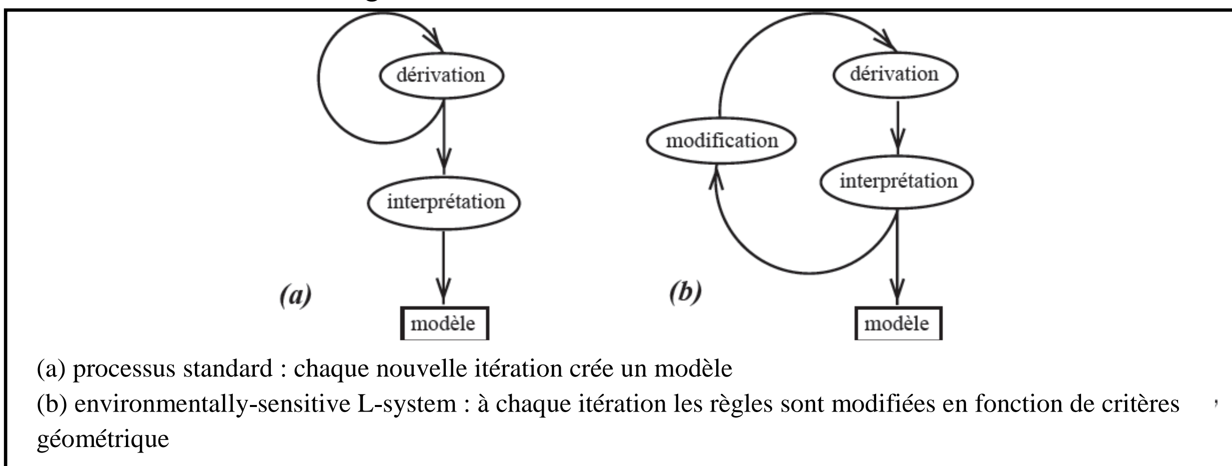
- les mécanismes endogènes : ils représentent l'action sur la croissance de paramètres internes à la plante, comme la propagation des informations entre les cellules.
- les mécanismes exogènes sont les influences externes à la plante, comme le voisinage ou la recherche de la lumière.

Les L-Systèmes prennent en charge le premier facteur, grâce aux possibilités de transfert de signaux et l'héritage.

Prusinkiewicz[15] propose une extension qui permet la prise en charge du deuxième type de facteurs. L'impossibilité de prise en charge des facteurs externes influencent la croissance d'une plante par les L-Systèmes est due à la séparation de la phase de génération de la structure de l'interprétation géométrique, alors dans la phase de génération aucune information concernant l'environnement est disponible, Prusinkiewicz propose d'effectuer une interprétation géométrique à chaque étape de la génération, donnant ainsi la possibilité de l'intégration des paramètres géométriques dans les règles de réécriture. Ce nouveau modèle est appelé « environmentally-sensitive L-systèmes ».

Fig20 : Exemple de croissance avec contrainte de limitation de taille [11]

L'interprétation graphique des structures générées à base de grammaires se fait en utilisant un système de tortue graphique, où chaque symbole va être interprété par, soit un mouvement de la tortue, soit par un changement de son orientation, ou bien une ampliation ou dépilation de l'état courent du contexte graphique, Prusinkiewicz propose pour enrichir cette interprétation, d'intégrer dans l'alphabet des grammaires des symboles qui contrôlent l'interprétation graphique, par exemple l'épaisseur, la couleur, ou la longueur des inter-nœud.



(a) processus standard : chaque nouvelle itération crée un modèle
(b) environmentally-sensitive L-system : à chaque itération les règles sont modifiées en fonction de critères géométrique

Fig21 : processus de génération L-System avec prise en charge des facteurs externes [15]

Pour représenter des variations structurelles dans une même espèce, Prusinkiewicz, propose l'introduction d'un facteur aléatoire en ajoutant des probabilités associer à chaque règle,

Cette version des L-Systèmes est appelés les L-Systèmes stochastiques.

Fig22 : Un champ de tournesol produit par les L-Systèmes stochastiques [15]

Lintermann [16]propose un système interactif pour la modélisation de plantes à base des L-Systèmes, qui permet la génération de structures par la sélection dans un ensemble d'objet et actions prédéfinies.

Fig23 : Système de modélisation de plantes de Lintermann [16]

2.1.6. Modèles combinatoires :

Les modèles combinatoires ont été développés par Viennot, Eyrolles, Janey et Arqués [17], qui utilise une matrice stochastique appelée matrice de ramification pour la génération des arbres.
La matrice de ramification est une matrice stochastique qui donne pour un nœud les probabilités de type de ramification qu'il suivra à chaque niveau de la génération.

- **Ordre d'un nœud :**
Afin d'établir un ordre sur les nœuds d'un arbre on utilise une fonction récursive qui permet d'avoir un ordre pour chaque nœud de l'arbre, on remarque que les nœuds terminaux auront tous l'ordre 1, et plus on s'approfondit dans l'arborescence, plus l'ordre devient plus grand jusqu'à la racine ou on aura le plus grand ordre, appelé nombre de Strahler de l'arbre.
Un segment d'ordre k est une succession maximale d'arrêtes (liaison entre deux nœuds) d'ordre k.

- **Bi-ordre d'un nœud :**
Considérons un nœud interne n d'ordre k ayant deux nœuds fils d'ordres i et j :
Si j>i alors j = k et le bi-ordre du nœud n est définit comme la paire (k,i).
Si j = i (et donc = k-1), le bi-ordre du nœud n est (k-1, k-1).

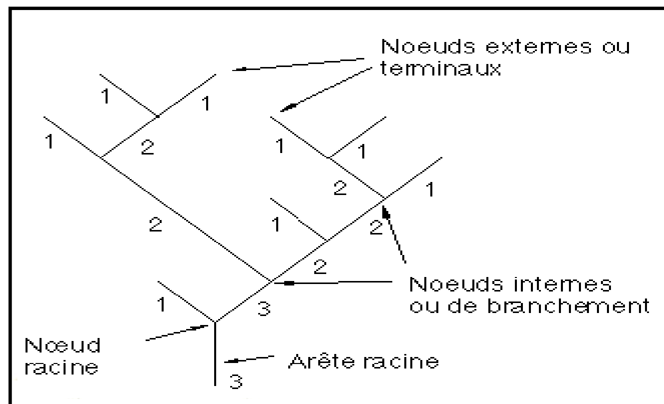


Fig24 : Ordre des nœuds dans un arbre [17]

Alors la matrice de ramification donne, pour chaque nœud d'ordre k, les probabilités de radication vers un couple de nœuds d'ordre (j, k) avec j < k ou d'ordre (k – 1, k-1), ainsi la matrice de ramification va contrôler l’apparence de l’arbre.

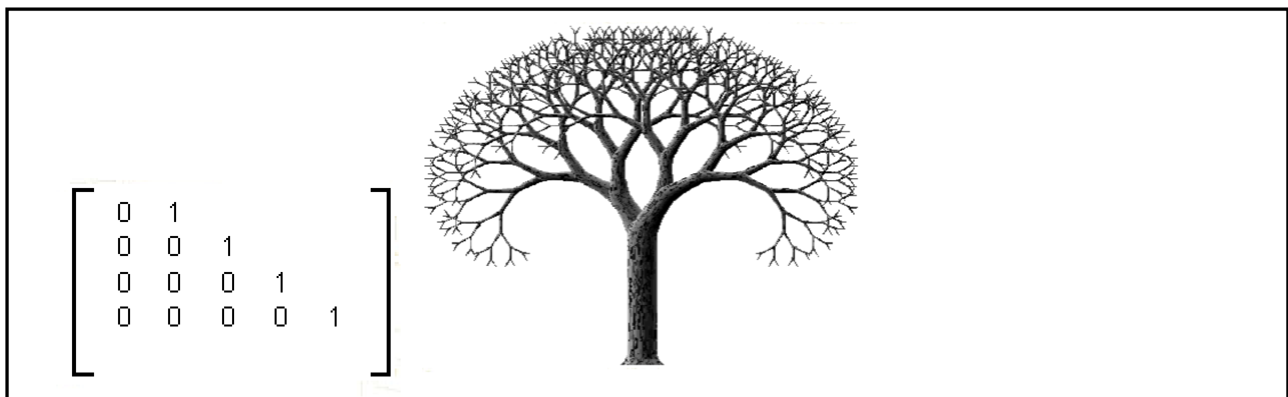
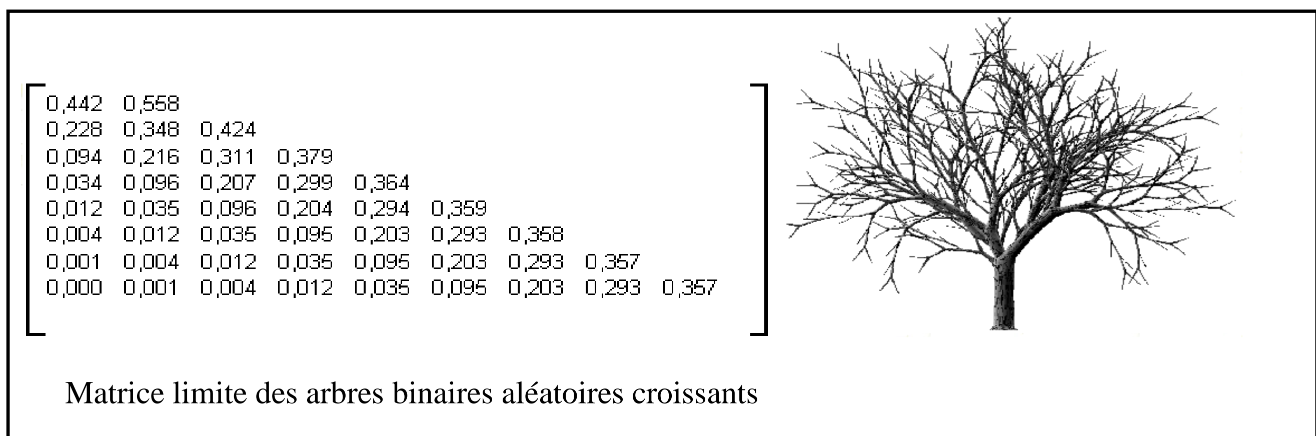


Fig25 : Matrice de ramification des arbres parfaits et un arbre généré a partir d’elle [17]



Matrice limite des arbres binaires aléatoires croissants

Fig26 : Arbre binaire aléatoire croissant généré par matrice limite des arbres binaires aléatoires croissants. [17]

2.1.7. Travaux de Weber et Penn :

Weber et Penn [18]ont proposé un modèle purement géométrique, ils ont remarqués que la structure ramifiée d’un arbre dépasse rarement quatre niveaux de récurions, alors ils définissent pour chaque niveaux de récurions un ensemble de paramètres géométriques généraux, et associent à chaque paramètre une valeur de tolérance, qui définit son intervalle de valeurs possibles.

Les paramètres principaux pour chaque niveau de récurions sont :

- CurveRes et Curve : déniassent le nombre et la rotation des troncs de cône qui composent une branche.
- SegSplits : donne le nombre de clones créés sur une branche, Ces clones ont le même niveau de récursion que la branche et héritent des mêmes paramètres géométriques.
- SplitAngle : définit l'angle de ramification des clones.
- Branches : définit le nombre maximum de sous-branches que peut porter une branche.
- Length : c'est le rapport de la longueur de la sous-branche par rapport à la longueur de la branche.
- DownAngle et Rotate : définissent les angles de branchement d'une sous-branche.

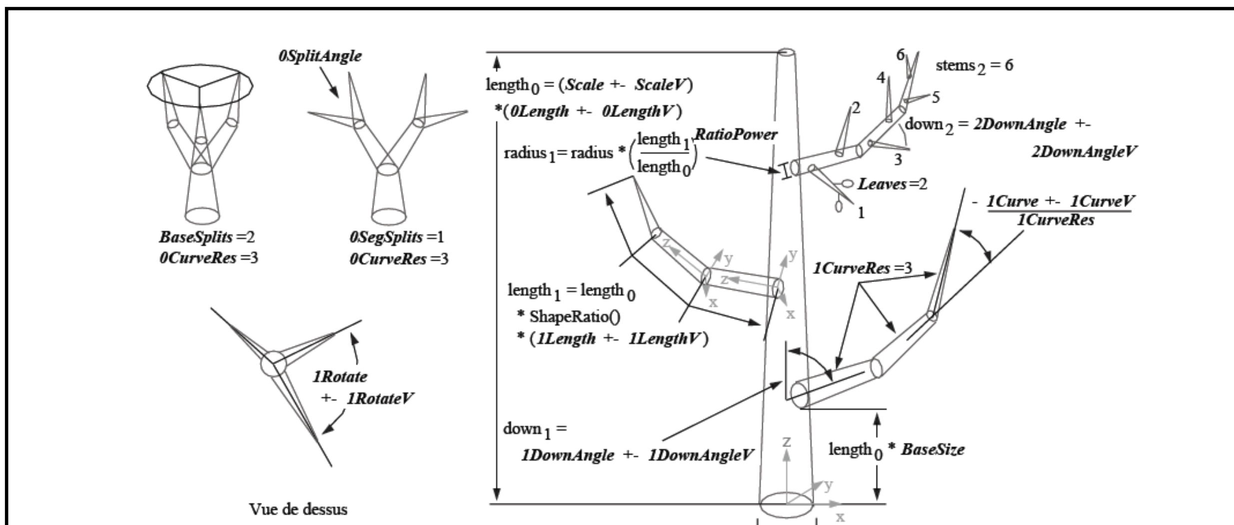


Fig27: génération d'arbres à base de modèle géométrique de Weber et Penn [18]

Avec cet ensemble de paramètres principaux ce modèle utilise un ensemble d'autres valeurs qui permettent plus de diversité comme le paramètre Shape qui est une valeur choisie parmi huit valeurs prédéfinis et permet de spécifier la forme de l'arbre, en spécifiant la longueur et les angles de branchement des sous branches. Taper, flare, est lobe contrôlent le rayon des branchements. Leave détermine la densité de feuillage et leaves-hape la forme des feuilles.
Ce modèle supporte une résolution multi-échelle, ou le niveau de détaille dépend de la distance para port à l'observateur, par une interprétation géométrique adaptative, les branches sont progressivement interprétés par des lignes et les feuilles par des points, à une certaine distance certain branches et feuilles ne sont plus interprétés.

Fig28 : Exemples d'arbres générés par le modèle de Weber et Penn [18]

Fig29 : Image multi-échelle d'un arbre pour le rendu à 30, 60, 120, 240,600, et 1200 m[18]

Ce modèle a atteint les objectifs de ses auteurs, avec une très grande variété d'arbres pouvant être représentés, aucune connaissance en botanique est nécessaire pour la génération de nouvelles espèces de végétaux, mais il représente l'inconvénient d'avoir un très grand nombre de préemptives générées, ce qui nécessite une grande place mémoire pour les sauvegarder.[11]

2.2. Modèles impressionnistes :

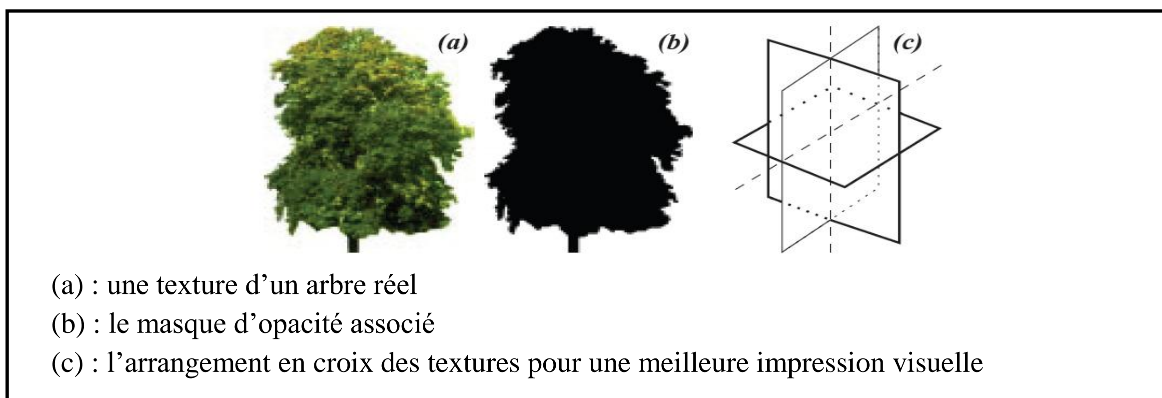
Dans cette catégorie de modèles [19], on se concentre sur l'aspect visuel de l'image résultat, plutôt que l'aspect structurel du végétale, ici l'objectif est de produire une image 2D convaincante a l'observateur, sans s'occuper du réalisme de la structure.

2.2.1 Modèles à base de texture :

1. Texture 2D :

Cette technique utilise une texture 2D représentant un arbre plaqué sur un polygone, et veiller à ce qu'il soit toujours orienté vers l'observateur. Cette technique permet un rendu très rapide d'un très grand nombre d'arbres en temps réel, mais elle a beaucoup d'inconvénient:

- A chaque arbre présenté dans la scène doit être associés une texture, alors le nombre d'arbres qu'on peut représenter est limité par la mémoire disponible pour les textures dans le système de rendu.
- La difficulté d'intégration des effets convaincants d'illumination et d'ombrage.
- L'observateur en se délassant dans la scène peut remarquer que l'image de l'arbre est toujours la même et aussi que l'arbre est parfaitement plat.



(a) : une texture d'un arbre réel
(b) : le masque d'opacité associé
(c) : l'arrangement en croix des textures pour une meilleure impression visuelle

Fig30 : modèle à base de texture 2D [19]

2. Texture 3D sur quadriques :

Cette technique a été proposée par Gardner pour le rendu temps réel des arbres et des nuages, en utilisant les ellipsoïdes fractales.

Un ellipsoïde fractale est une élision dont la couleur et l'opacité est modulée par une texture 3D.
La géométrie d'un ellipsoïde est définie par l'équation :

$$(X - X_0)^2/R_x^2 + (Y - Y_0)^2/R_y^2 + (Z - Z_0)^2/R_z^2 = 1.$$

Une texture fractale Fr(X, Y, Z) est calculée à partir du produit de séries de Fourier :

$$S_X = \sum_{i=1}^{N} C_i(cos(\omega_{X_i}(X + \Phi_{G_X}) + \phi_{X_i}) + 1)$$

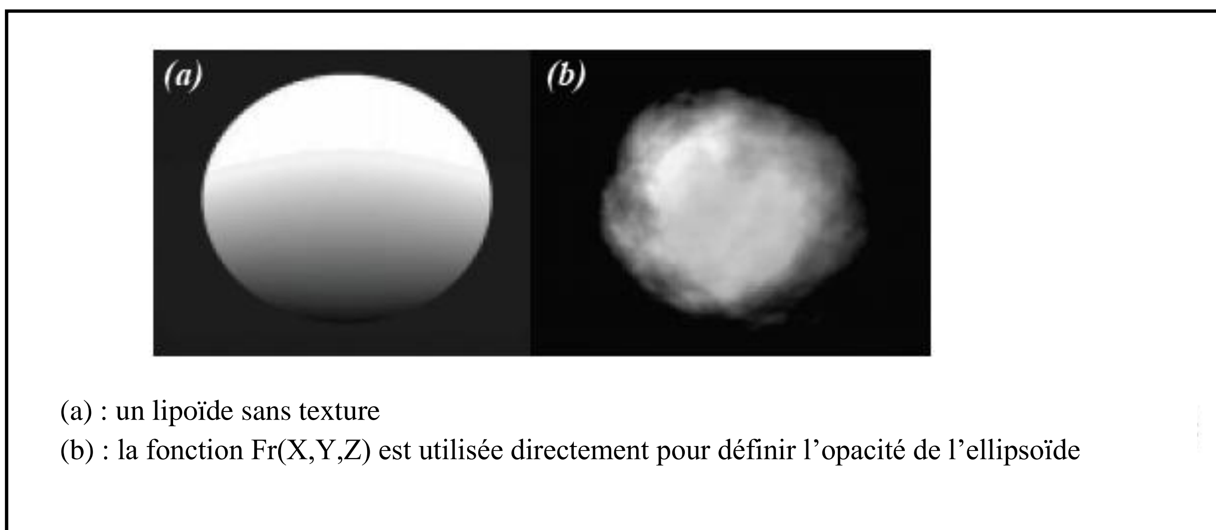
$$S_Y = \sum_{i=1}^{N} C_i(cos(\omega_{Y_i}(Y + \Phi_{G_Y}) + \phi_{Y_i}) + 1)$$

$$S_Z = \sum_{i=1}^{N} C_i(cos(\omega_{Z_i}(Z + \Phi_{G_Z}) + \phi_{Z_i}) + 1)$$

$$Fr(X, Y, Z) = S_X S_Y S_Z$$

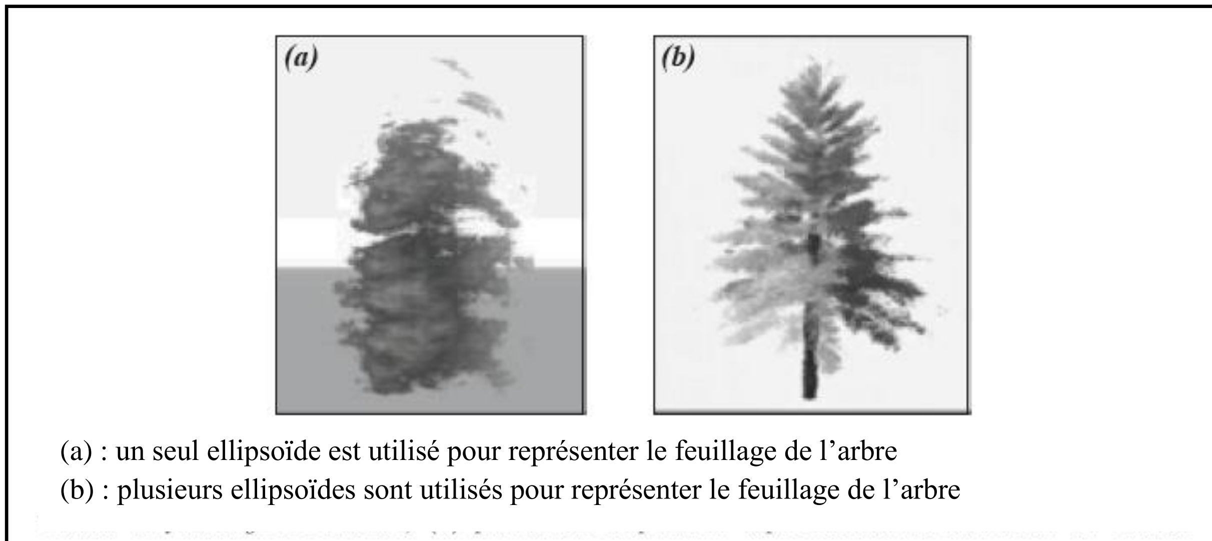
Les paramètres Ci, wXi, wYi, wZi définissent le spectre des fréquences de la texture, ØXi, ØYi, ØZi représentent les phases locales et sont utilisées pour introduire des variations aléatoires, tandis que ɸXi , ɸYi , ɸZi sont les phases globales et permettent d'appliquer une translation a la texture.

Le nombre d'ellipsoïdes utilisés pour la représentation des arbres dépend du niveau de précession requis, on peut utiliser un seul ellipsoïde pour représenter un arbre pour précision faible, ou un ellipsoïde par branche et un pour le tronc pour haute pression.[20]



(a) : un lipoïde sans texture
(b) : la fonction Fr(X,Y,Z) est utilisée directement pour définir l'opacité de l'ellipsoïde

Fig31 : Ellipsoïdes Fractals [11]



(a) : un seul ellipsoïde est utilisé pour représenter le feuillage de l'arbre
(b) : plusieurs ellipsoïdes sont utilisés pour représenter le feuillage de l'arbre

Fig32 : utilisation des ellipsoïdes fractaels pour la représentation d'arbres [20]

2.2.2 Modèles dans un espace voxels :

Modèle développé par Ned Green[21], qui propose une technique de génération de plantes à travers un processus de croissance stochastique dans un espace discret (divisé en voxels), ce qui permet à ce modèle de bénéficier des simplifications de certaines opérations offertes par les voxels.

La représentation dans un espace voxels offre :

- chaque objet étant représenté par les voxels qu'il occupe, un simple test suffit pour déterminer si un espace est occupé ou non.
- On peut déterminer rapidement les objets les plus proches d'une position dans l'espace, en parcourant les voxels voisins.

Ces deux propriétés des espaces voxels permettent la perception de l'entérinement d'une plante pendant sa croissance, ce qui offre la possibilité de prendre en charge les facteurs externes influençant le développement de la plante (éviter les obstacles, recherche de lumière …).

Processus de modélisation :

Le processus de croissance d'une plante dans les espace voxels et basé sur la recherche de trajectoires satisfaisant certain régleuses, à chaque itération de nouvelles positions pour les extrémités de la plante sont calculées selon une méthode de Monte-Carlo (on échantillonne un ensemble de positions possibles et on sélectionne les plus satisfaisantes au règles).

Les règles qui contrôlent la croissance sont des contraintes sur la structure ramifié de la plante ou des contraintes sur sa réaction à l'environnement (par exemple éviter les obstacles, rester à proximité d'un objet, chercher un maximum d'ensoleillement …).

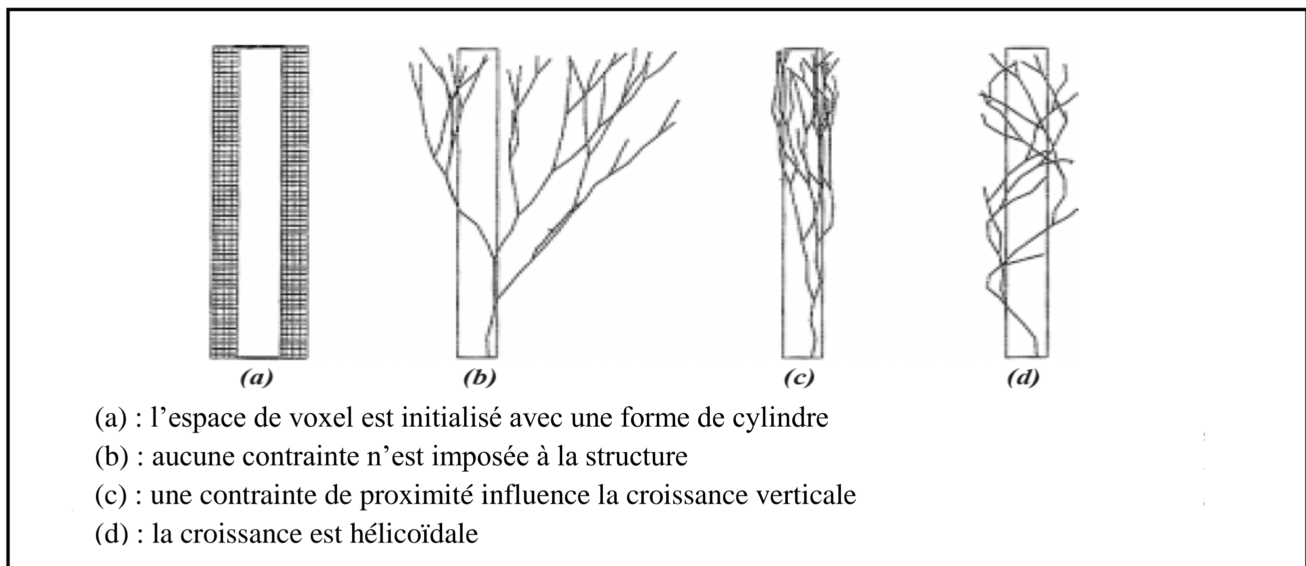


Fig33 : Croissance dans un espace voxel avec prise en charge de contraintes d’environnement extérieur [21]

2.2.3. Modèles à base de systèmes de particules :

Reeves a proposé en 1983[22] l’utilisation des systèmes de particules pour la représentation des éléments naturels, il a utilisé les systèmes de particules pour représenter et animer des phénomène naturels dont les caractéristiques changent au cour du temps qui sont très complexes pour les représenter à l’aide de primitives géométriques classiques, comme le feu, les nuages, la brume et l’eau.

L’aidé et d’utiliser des primitives très simples, qui peuvent grâce a leurs très grand nombre représenter de formes et des animation très complexes, une particule a les attributs suivent :

- La position dans l’espace.
- Le vecteur vitesse.
- La taille de la particule
- La couleur et la transparence.
- La forme de la particule.
- L’âge de la particule.

Pour chaque particule ces attributs sont initialisés avec certaines valeurs puis elle évolue en subissant les influences externes qui changent les valeurs de ses attributs (gravité par exemple).

Pour le rendu chaque particule ajoute sa contribution élémentaire aux pixels qu’elle couvre.

La représentation en utilisant les systèmes de particules est intéressante pour plusieurs raisons :

- La particule ponctuelle étant la plus simple des primitives graphiques. Alors un très grand nombre pourra en être dessiné dans le même laps de temps, des objets avec plus de détails pourront donc être obtenus.
- Les systèmes de particules sont procéduraux et stochastiques : on peut minimiser le temps passé à la modélisation car un faible nombre de paramètres initiaux permettent d'obtenir des images très fouillées, on dit qu'il y a un grand taux d'amplification des données.
- Ils permettent de modéliser des scènes dynamiques (feu, vent dans de l'herbe).

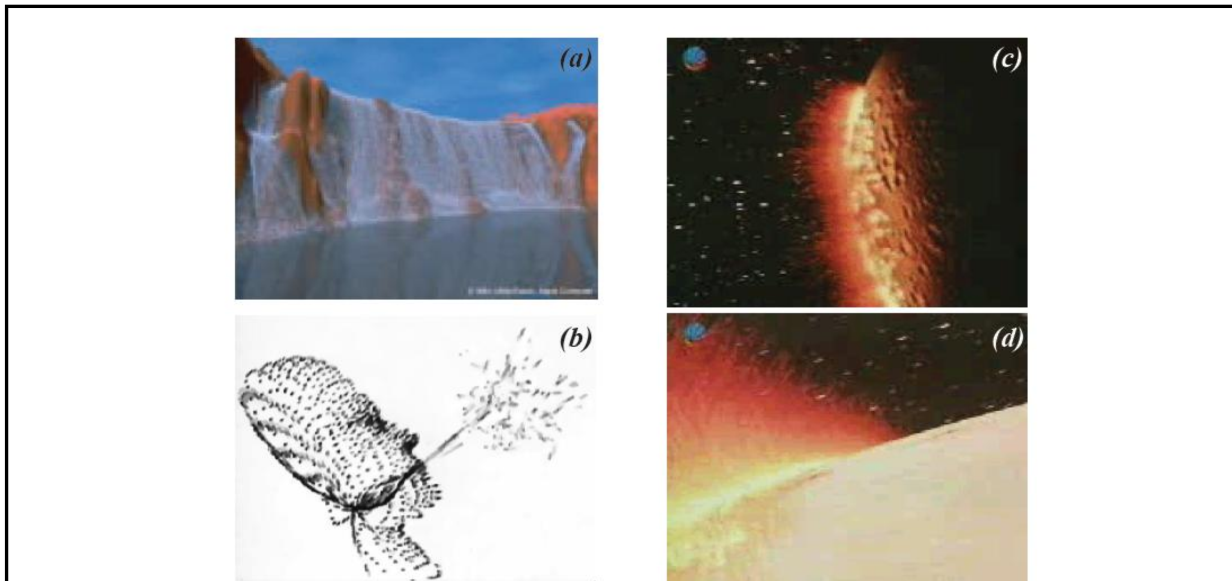


Fig34 : modélisation avec systèmes de particules [22]

En 1985 Reeves et Blau[23], présentent une variante de systèmes de particules adapter à la représentation des végétaux au lieu d'utiliser des particules strictement indépendantes, dans cette variante ils proposent d'utiliser des relations entre les particules ce qui produit un système plus contraint et plus adapter à la représentation d'objets structurés comme les arbres.

Reeves et Blau proposent un algorithme pour la génération des particules représentant l'arbre, en tenant compte de son aspect général, cet algorithme est paramétré par des contraintes géométriques, comme la longueur moyenne de l'arbre, l'épaisseur du tronc, la longueur et l'épaisseur moyenne des branches, les angles de branchement… .

A chacun de ces paramètres une moyenne et un écart type est affecté, ainsi qu'une probabilité de ramification pour les branches, et une densité, position, couleur, et orientation pour les feuilles, à partir desquelles l'algorithme génère récursivement un grand nombre de particules représentées par des segments pour les branches et le tronc, et des disques pour les feuilles,

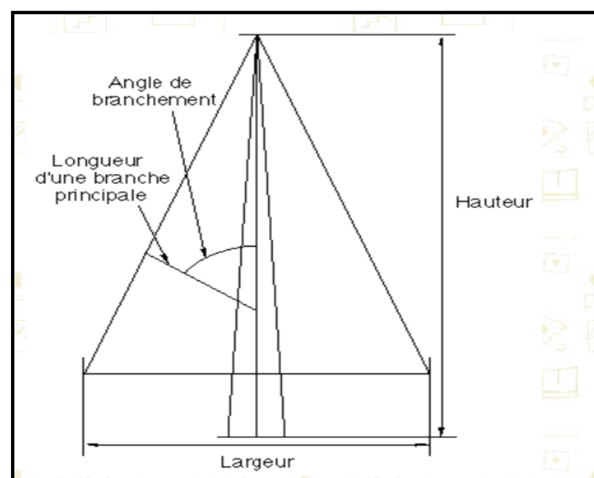
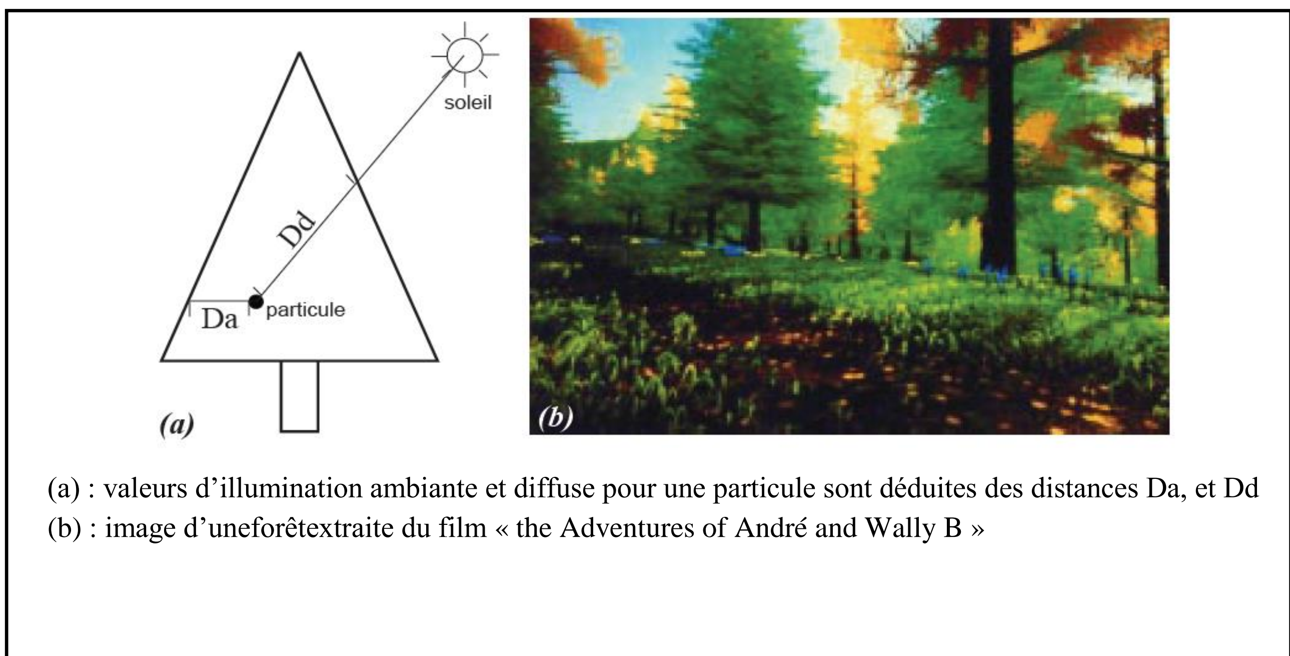


Fig35 : Paramètres géométriques qui contrôlent la génération de particules[5]

Cet algorithme génère plusieurs milliers de particules par arbre, ce qui pose un problème de rendu de ce nombre très important de primitive géométriques, la solution proposé par Reeves et Blau est une méthode de rendu localement approximative, pour chaque particule une approximation d'illumination est calculé à partir de sa position dans l'arbre, la forme globale de l'arbre, et la position du soleil.

Même si localement le rendu est approximatif, le très grand nombre de particules formant un arbre permet une excellente impression globale de l'arbre.

Pour le rendu d'une scène contenant plusieurs arbres, les particules constituants un arbre sont classées grossièrement sellant leurs distances para port a l'observateur de la plus loin a la plus proche, puis le shading et le tracer de chaque arbre est effectué indépendamment du plus loin au plus proche de l'observateur.



(a) : valeurs d'illumination ambiante et diffuse pour une particule sont déduites des distances Da, et Dd
(b) : image d'uneforêtextraite du film « the Adventures of André and Wally B »

Fig36 : rendu d'arbres générés avec des systèmes de particules[23]

- une composante D de lumière diffuse exponentiellement décroissante de la distance dd dans l'arbre à la source lumineuse (D = e-add),
- une composante A de lumière ambiante exponentiellement croissante de la distance horizontale da de la particule au bord extérieur de l'arbre et minorée par une valeur minimale d'éclairage ambiant Amin (A = max(e-bda,Amin)),
- une composante S de lumière spéculaire pour les particules de distance dd faible et dont la branche associée est perpendiculaire à la direction d'incidence des rayons lumineux.

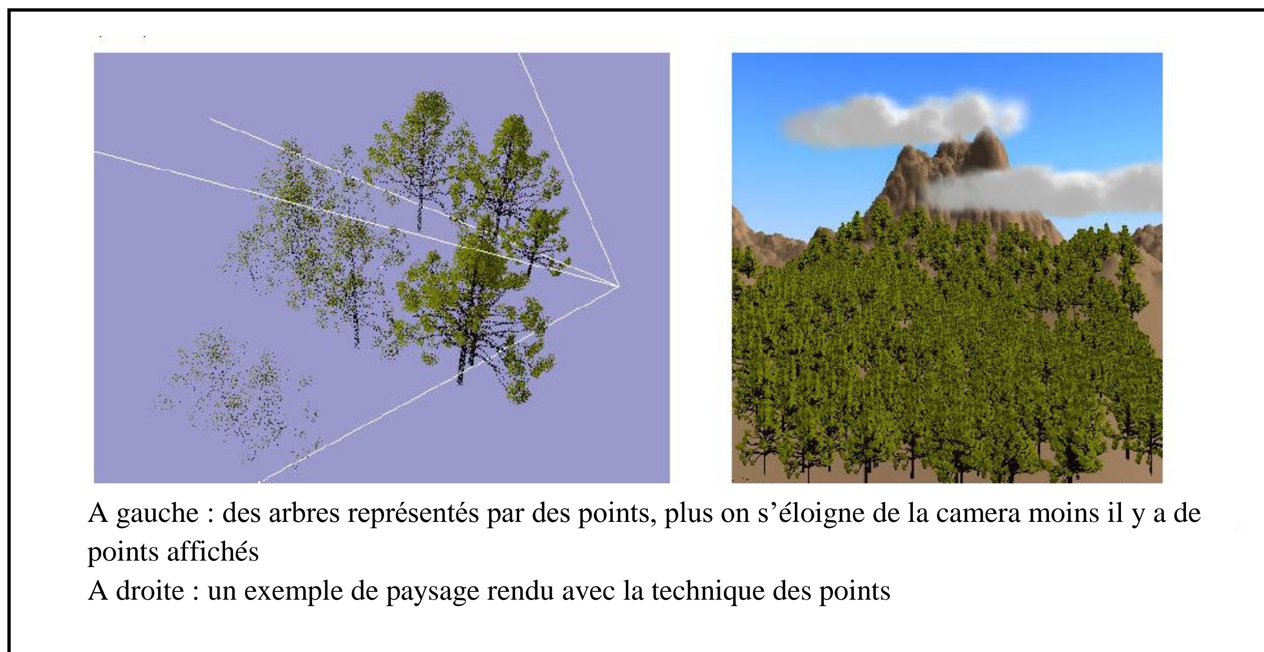
Ce modèle donne des résultats visuels excellents, mais il est très limité en matière de réalisme biologique (le nombre limité de paramètres géométriques ne permet pas la représentation de la structure ramifiée spécifique d'une espèce précise), en plus la structure de l'arbre n'est pas très convaincante sans les feuilles, les résultats de ce modèles peuvent être décevants en cas d'arbres sans feuilles (en automne par exemple).

2.2.4. Modèles à base de points :

En 2000 Pfister [24] et Rusinkiewicz [25] proposent une représentation à base de points en utilisant le disque comme primitive de modélisation, qui est très facilement géré par une carte graphique, est une primitive bien adaptée à la représentation de nombreux phénomènes ou d'objets complexes.

À la place des polygones, Pfister utilise des points appelés surfels (surface elements), ces points ne sont pas connexes, ce qui permet l'adaptation de leurs nombre en fonction de la taille de l'objet à l'écran, alors le coût d'affichage d'un objet est proportionnel à sa taille à l'écran et non à sa complexité intrinsèque, ce qui permet de conserver un bon taux de rafraîchissement de l'image, même avec beaucoup d'objets affichés.

Stamminger [26] propose d'échantillonner à la volée des objets procéduraux, ce qui offre une plus grande souplesse dans le choix de la précision. Il montre aussi que cette technique est bien adaptée au rendu de phénomènes naturels variés comme une montagne, le mouvement de l'eau ou le rendu d'arbres.



A gauche : des arbres représentés par des points, plus on s'éloigne de la camera moins il y a de points affichés
A droite : un exemple de paysage rendu avec la technique des points

Fig37 : Résultats de représentation d'arbres à base de points [26]

4. Modèles d'illumination à base d'image adaptés pour le rendu des arbres :

4.1. Tranches d'images de profondeurs (Layer Depth Images (LDI)) :

Méthode proposée par Shade en 1998 [36], en utilisant plusieurs images et leur tampon de profondeur associé, Cependant cette technique, tout comme les lumigraphs, ne permet pas le temps réel sur des centaines d'objets, ni le changement d'éclairage, et les objets semi-opaques ne sont pas représentables, parce que le tampon de profondeur ne stocke qu'une unique valeur de profondeur par pixel.

Une adaptation aux arbres [37] est l'utilisation d'un tampon de profondeur par pixel qui contient la couleur, la profondeur ainsi qu'une unique normale. La représentation est hiérarchique, c'est-à-dire que ce modèle part des images d'une feuille (une feuille réelle scannée), avec lesquelles il construit une petite branche dont il tire une image et son tampon de profondeur, lesquels sont utilisés pour

construire une branche entière, etc. jusqu'à arriver au niveau de l'arbre. Il précalcule une vingtaine de points de vue pour chaque objet, qu'il sélectionne au moment du rendu en fonction de la position de l'œil. Le rendu est effectué avec l'aide du matériel graphique, l'illumination est calculée en post-traitement à l'aide de la normale et de la couleur stockée dans chaque pixel de l'image. L'antialiassage est obtenu par sur-échantillonnage.

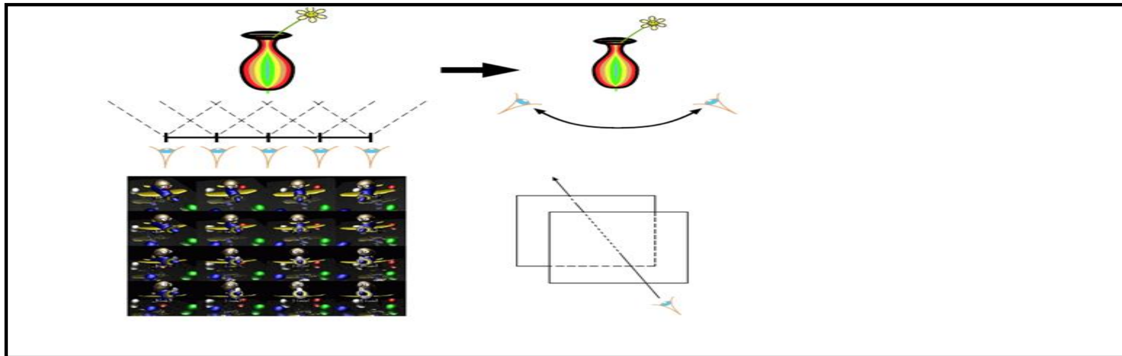


Fig38 : Application de la technique LDI [37]

4.2. Rendu à base de textures volumiques :

Une approche orientée texture pour représenter des géométries répétitives complexes recouvrant une surface à la manière d'une peau épaisse, comme par exemple, la fourrure d'un animal, une forêt sur une colline, etc. Un volume cubique contient un échantillon de référence de cette géométrie encodée sous forme d'un volume de voxels. Les instances de ce volume plaquées sur une surface sont appelées « texels » pour texture élément. Ce volume de référence est déformé de façon à être plaqué sur la surface, à la manière d'une texture 2D classique.
Le volume cubique de référence est constitué de voxels. Chaque voxel contient formellement trois informations :

- une densité (i.e. une présence)
- un ensemble de 3 vecteurs donnant l'orientation locale de la surface (Normale, Tangente, BiNormale)
- une fonction indiquant comment la lumière se réfléchit (réflectance).

L'orientation locale de la surface et la réflectance peuvent être regroupées en une unique donnée.
Un texel contient donc une information spatiale (densité) et une information sur le comportement local vis à vis de la lumière (réflectance).

Le volume de référence est destiné à être plaqué de manière répétitive sur toute la surface comme pour une texture surfacique classique. Pour qu'il y ait continuité entre texels voisins il faut que le volume soit déformé.

Le rendu utilise un algorithme de lancer de rayons qui devient un peu particulier lorsque celui-ci traverse un texel. Lorsque le rayon traverse un texel on se reporte dans le volume de référence où on applique une technique de rendu volumique : le rayon parcourt les voxels tout en accumulant la transparence et l'illumination locale, qui est évaluée en appliquant la fonction de réflectance, pondérée par l'ombrage obtenu en lançant un rayon entre le voxel et la source de lumière.[38]

Noma [39] propose une adaptation du principe des textures volumiques de pour le rendu spécifique d'arbres. En chaque voxel, il stocke l'opacité de manière discrète en échantillonnant la géométrie depuis une série de directions autour de la sphère, lors du rendu il interpole les valeurs. Pour la fonction d'illumination, il calcule en chaque voxel la moyenne des normales des feuilles et la

moyenne des cosinus des angles entre ces normales et les trois axes X, Y, Z. Lors du rendu, il interpole ces valeurs en fonction des angles entre la lumière et les axes X, Y, Z. Il utilise de la géométrie pour les arbres proches qu'il mélange aux texels pour les arbres éloignés.

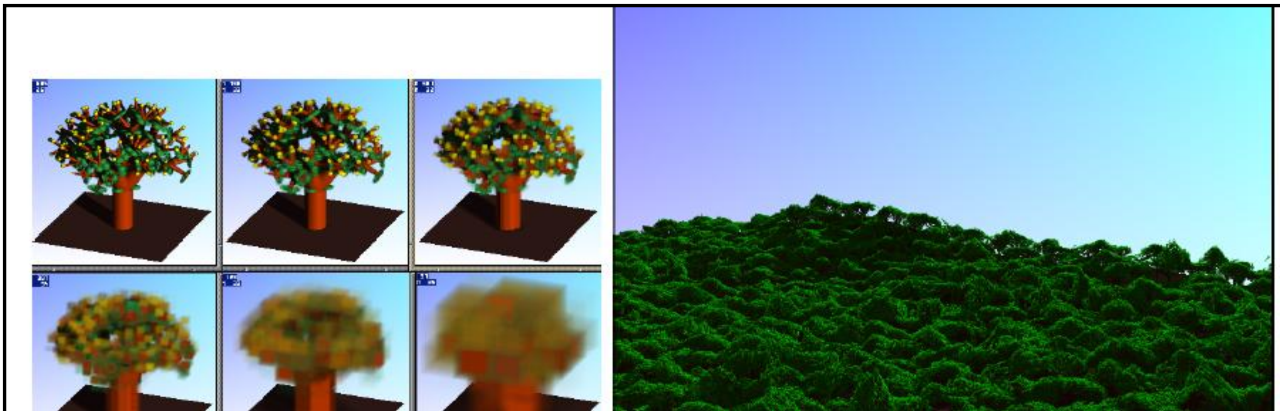


Fig39 : Rendu a base de texture volumique[39]

Les textures volumiques peuvent être vues de deux manières :

- une représentation pour les scènes complexes, facile à contrôler par l'utilisateur ;
- une représentation permettant un rendu très efficace en temps et en qualité, car générique et multi-échelle.

Cette représentation alternative propose de remplacer une géométrie complexe par un octree où chaque voxel est représenté par une fonction analytique d'illumination. Neyret propose une fonction générique ellipsoïdale. Cette fonction de réflectance générique peut s'avérer inadaptée à certaines géométries : les arêtes vives, ou toutes géométries où la distribution de normales comporte des discontinuités. Pour remédier à ceci l'idée serait de calculer une série de fonctions d'illumination pour des classes d'objets spécifiques. [6]

4.3 Rendu à base de couches d'images :

Les textures volumiques donnent des résultats visuels réalistes, et rapidement pour une technique utilisant le lancer de rayons. Cependant, pour des applications exigeant le temps réel, l'obtention d'un gain supplémentaire parait difficilement envisageable en conservant cette technique telle quelle. Des approches adaptées au matériel graphique ont donc été développées. Les moteurs de rendu des cartes graphiques traitant efficacement des polygones, nous présentons ici des techniques basées sur le rendu par couches, où chaque couche est représentée par un polygone texturé. Cette idée était à l'origine destinée au rendu volumique puis elle s'est généralisée à divers domaines du rendu.

Le rendu volumique se calculait usuellement en lançant des rayons à travers l'espace voxelisé, ce qui se traduisait par des temps de calcul très longs. Pour l'accélérer Lacroute et Levoy[40] introduisent le rendu par couches d'images.

Ils interprètent leurs données volumiques comme des couches, avec l'idée de factoriser les voxels en une texture et de traiter en parallèle ces données. Pour le rendu ils proposent de projeter et composer successivement les couches sur le plan image et ainsi obtenir l'image finale. Chaque couche projetée est combinée avec le résultat précédent en tenant compte de la densité, on peut donc interpréter une couche comme une texture transparente. La projection d'une couche est plus rapide que les calculs de projection pour chaque voxel le long d'un rayon : cette factorisation permet donc un gain de temps important.

Neyret et Mayer [41] ont proposés une technique temps réel des textures volumiques. en tirant profit de la capacité des cartes graphiques à traiter rapidement des polygones texturés. Le principe est de représenter le texel par une superposition de polygones texturés et transparents.

Comme les tranches ne font pas face à l'observateur et n'ont pas d'épaisseur, on risque de voir entre elles. On définit donc trois directions de tranches. Le point de vue de l'observateur détermine laquelle de ces directions va être utilisée. La génération des texels peut se faire de deux manières, soit en convertissant une représentation polygonale, soit à partir d'une texture de hauteur.

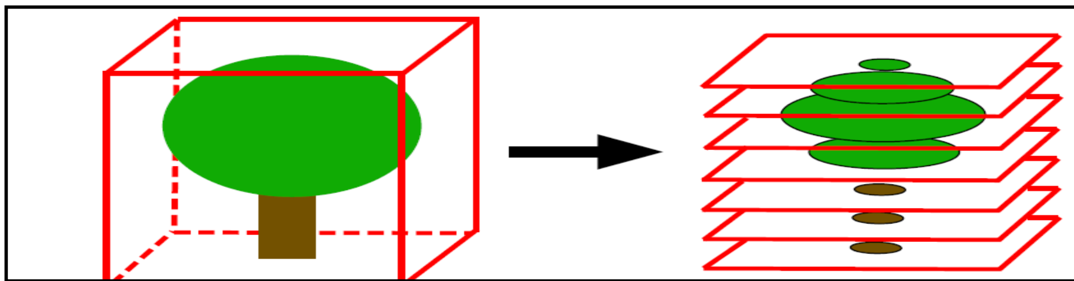


Fig40 : Rendu à base de couches d'images [41]

Conclusion :

Les deux types de modèles de représentation des végétaux sont chacun adapté à un type d'utilisation qui exploit ses avenages.

Les modèle structurelles représente l'avantage de la présence de la géométrie du végétal qui permet d'avoir un calcul exact des positions de ces composants et leur éclairage, donnant ainsi des résultats très cohérents pour les vues rapprochés, et une interaction réelle avec les autres objets présents dans la scène, mais ils souffrent de la quantité énorme de géométrie nécessaire à la représentation, qui cause plusieurs problèmes :

- Un temps de calcul énorme pour le rendu des scènes naturelles qui les rend non envisageable pour les applications temps réelle.
- Le coût de rendu et de stockage augmente très rapidement en fonction de la complexité de structure du végétale ce qui impose une limitation sur cette dernière.

Les modèles impressionnistes n'utilisent pas une représentation géométrique des végétaux, ce qui résout les problèmes dont souffrent les modèles structurels :

- Un temps de calcul significativement réduit par ce qu'ils traitent une représentation beaucoup plus simple que la géométrie(image, plans texturées…).
- La complexité de la structure du végétale affecte beaucoup moins le coût de rendu et de stockage, permettant ainsi plus de liberté de représentation des espèces plus complexe.

Ces avantages rendent les modèles impressionnistes les plus adaptés aux applications temps réel, mais l'absence de la structure géométrique cause aussi des problèmes du réalisme pour les vues rapprochées, et pour la cohérence de l'interaction des végétaux avec le reste de la scène.

La croissance qu'a connue le matériel graphique ces dernières années permet d'envisager la gestion de plus de complexité géométrique en temps réel, donnant ainsi la possibilité de considérer des modèles structurels pour le rendu temps réel, mais la relation entre le cout de rendu et la complexité géométrique qui caractérise les modèles structurels reste toujours une limitation imposée sur les applications temps réel qui utilisent purement ces modèles.

L'effet de cette limitation peut être réduit en combinant avec la représentation de structure géométrique des techniques inspirée des modèles impressionnistes, qui seront appliqué pour les vues lointaines des végétaux, et qui sont déduits de la représentation géométrique elle-même pour produire des résultats cohérents.

Chapitre 2 :
Les processeurs graphiques (Graphical Processing Units)

Chapitre 2

Les processeurs graphiques (Graphical Processing Units (GPU))

1. Introduction :

Quand les premières cartes graphiques sont apparues leur tâche était très simple : prendre l'image finale produite par le CPU et l'afficher sur l'écran, vers la moitié des années 1990 une révolution a pris place avec l'avènement des cartes graphiques équipées de processeurs spécialisés pour le rendu 3D, appelés GPU (Graphics Processing Unit), libérant ainsi partiellement le CPU de la tâche du rendu.

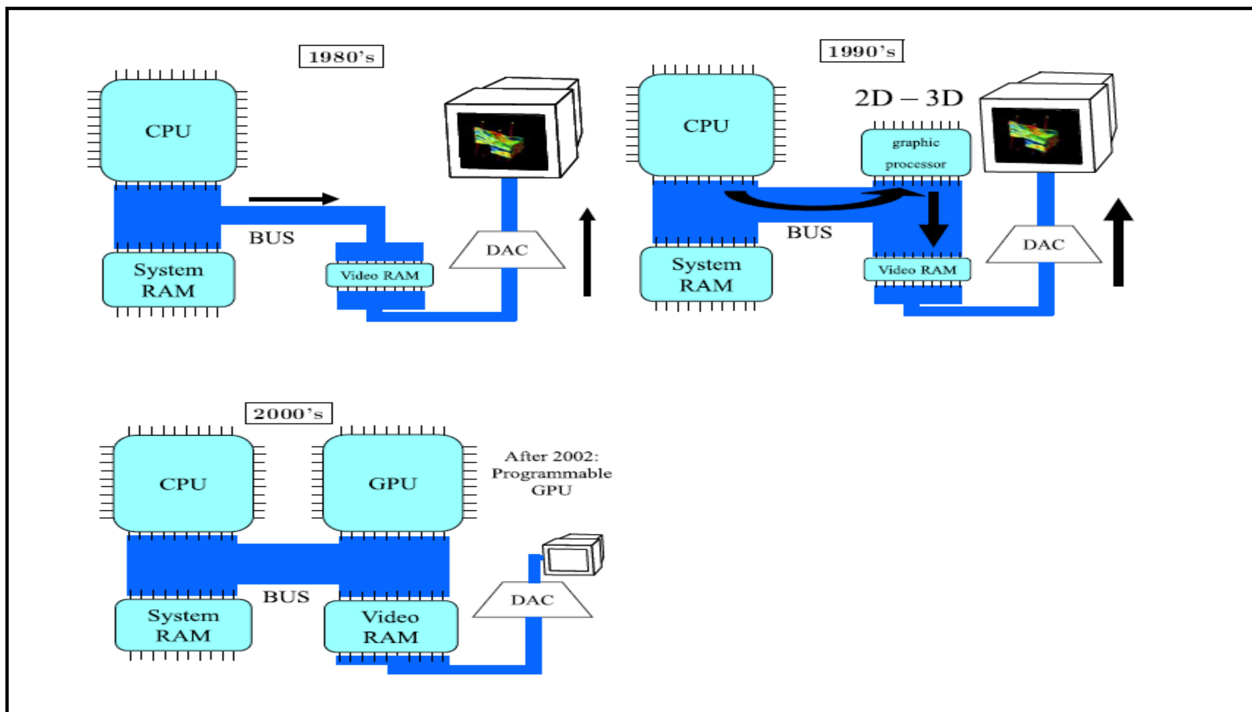


Fig41 : Evolution des cartes graphiques [43]

La technologie des GPU est principalement dédiée au rendu des triangles 3D, essentiellement par ce que leur algorithme de rastérisation est câblé, en plus de quelques raisons géométriques qui ont conduit au choix des triangles comme la primitive géométrique à utilisée :

- Les sommets d'un triangle appartiennent toujours au même plan.
- N'importe quel objet peut être approximé par un mesh triangulaire.
- N'importe quel polygone peut être subdivisé en triangles.
- Les triangles sont toujours convexes, et les coordonnées barycentriques peuvent être utilisées sans ambigüité pour interpoler les sommets dans le domaine des triangles. Pour

traiter ces triangles le GPU implémente un pipeline graphique 3D, qui admet en entrée le scenario 3D (incluant les meshs), et produit l'image 2D à afficher sur l'écran en sortie. [44]

2. Pipeline graphique :

Le pipeline graphique consiste en plusieurs étapes de traitement que subis les informations du scénario 3D (les sommets et les informations qui constituent la scène), pour produire l'image 2D à afficher sur l'écran.

Le scénario 3D est constitué de:[44]

- Un modèle géométrique : objets, surfaces sources de lumières … .
- Un modèle d'illumination : les données nécessaires pour le calcul des interactions lumineuses, les normales, les propriétés de matériaux, l'intensité et la couleur de la lumière, les textures … .
- Camera : position, ouverture (frustum).
- Viewport (fenêtre) : une grille de pixels sur laquelle on plaque l'image résultat du rendu.

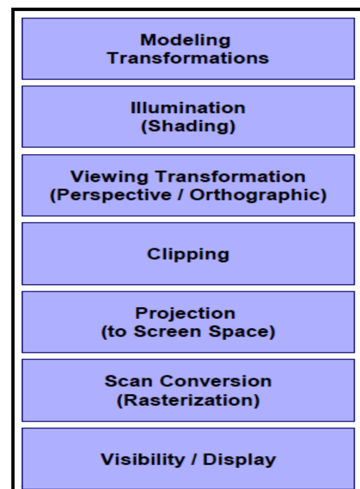


Fig42 : Etapes du pipeline graphique [44]

Chaque primitive passe successivement par toutes les étapes, qui sont implémentées de diverses manières dans le hardware ou le software.

- Transformation : cette étape fait le passage du système de coordonnées local de chaque objet 3D (object space) vers un repère global (world space), et les translations et les rotations.

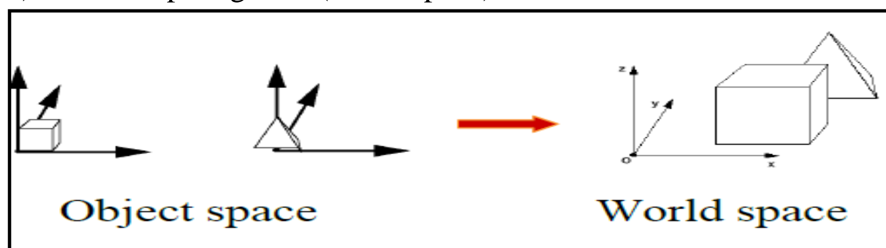


Fig43 : Passage du repère local vers le repère global [44]

- Illumination (Shading) : Les primitives sont éclairées selon leur matériau, le type de surface et les sources de lumière, et les normales des sommets, les modèles d'illumination sont locaux (pas d'ombres) car le calcul est effectué par primitive.
- Transformation perspective (viewing transformation) : Passe des coordonnées du monde à celles du point de vue (repère caméra ou eyespace). En général le repère est aligné selon z.

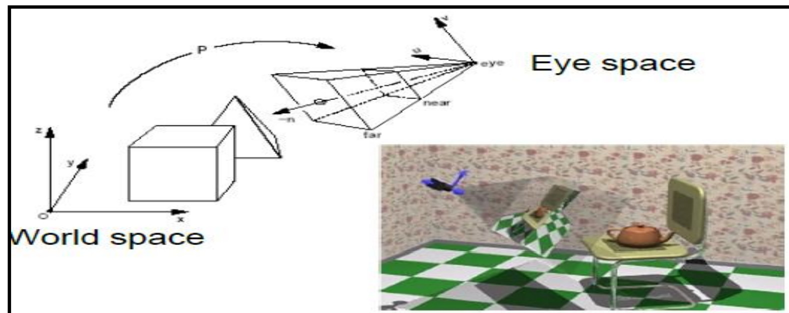


Fig44 : Transformation du repère global au coordonnés du repère de la caméra [44]

- Clipping : dans cette étape les portions en dehors du volume de vue (frustum) sont coupées, pour réaliser cette tâche, on utilise les coordonnées normalisées (NDC :normalizeddevicecoordinates)

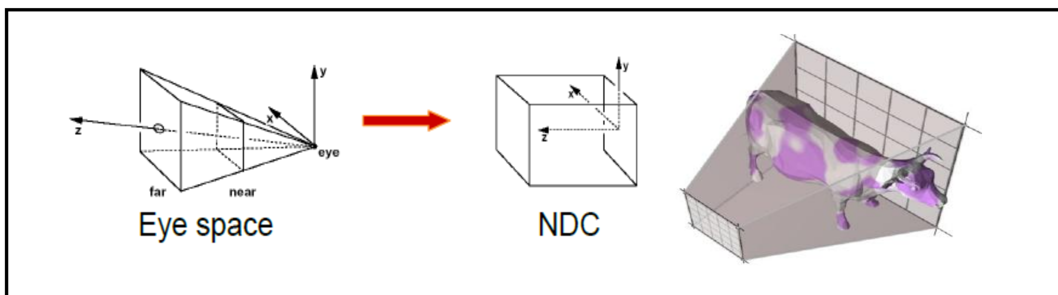


Fig45 :Passage aux coordonnées normalisées et clipping[44]

- Projection sur l'écran (screenspace projection): dans cette étape les primitives 3D sont projetées sur l'image 2D

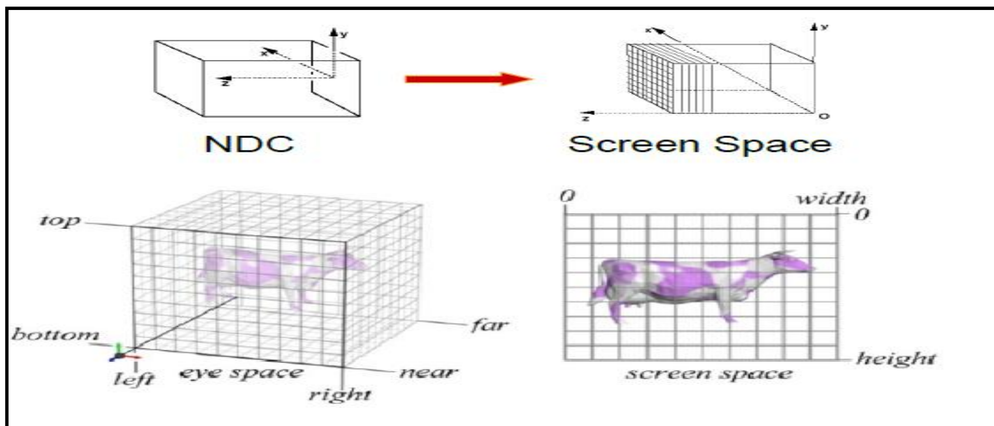


Fig46 :Projection dans l'espace écran pour produire l'image 2D a affiché [44]

- Rastérisation : cette étape génère les fragments, en découpant les primitives en petites parties dont chacune couvre un pixel de l'image, et calcule pour chaque fragment les attributs (couleur, normale, coordonnées texture …), en interpolant les attributs des sommets.

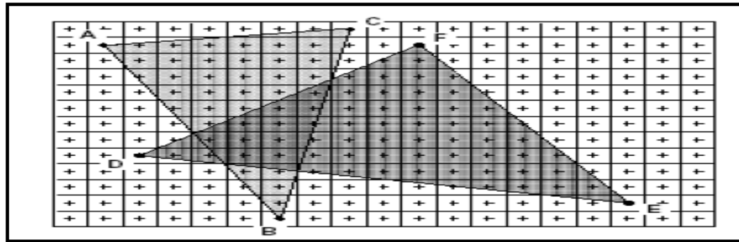


Fig47 : Découpage des primitives en fragments [44]

- Affichage (Display) : cette étape est responsable de remplissage de frame buffer conformément aux spécifications API, et aux propriétés des fragments (couleur, visibilité …).

Dans les premiers temps ce pipeline a était implémenté d'une façon fixe dans les cartes graphiques, avec des paramètres accessibles qui permettent de contrôler la procédure de rendu, mais on ne pouvait pas modifier l'algorithme à l'intérieur.

Ce type de cartes graphiques a connue plusieurs améliorations comme le multi-texturage, et les textures 3D, avant l'introduction de la programmabilité dans certaines étapes du pipeline. [44]

3. Evolution de la programmabilité des GPUs :

Depuis leur apparition, les GPU sont connus un développement très rapide non seulement dans leur capacité de calcul (par le double chaque 12 mois, ce qui dépasse la loi de Moor pour les CPUs : le double chaque 18 mois), mais aussi dans les fonctionnalités qu'ils offrent.

Au début ils étaient capables de rendre des triangles en calculant leurs illuminations et en leurs appliquant une textures 2D, de nouvelles fonctionnalités sont apparues avec chaque nouvelle génération de carte graphique, comme le multi-texturage, le blending, le multi-passes, les requêtes d'occlusion, et les textures 3D, mais ils étaient supposés fonctionner comme une boite noire, avec comme entée la scène 3D (objets texturés + lumière + position de la camera), et comme sortie l'image 2D, à l'intérieur du processeur graphique plusieurs étapes s'exécutent (pipeline graphique), sans possibilité de changer son comportement, qu'à travers les paramètres qu'on peut ajouter au entrées.

En 2003 un nouveau modèle de GPUs est apparu, avec la possibilité de changer complètement son comportement par défaut, à travers des programmes écrits à l'aide des instructions GPU qui peuvent être passé au pipeline graphique, et constitués le nouveau comportement de l'une de ses étapes.[43]

La possibilité de la programmation, concerne les deux étapes les plus essentielles du pipeline graphique : la transformation (géométrie, illumination, projection), et la rastérisation.

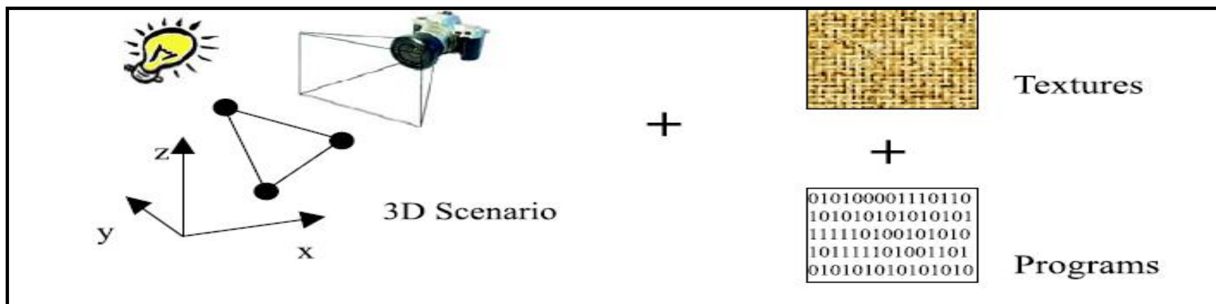


Fig48 : Les nouvelles entrées du GPU[43]

Dans cette génération de cartes graphiques, les GPUs donnent la possibilité de substituer le traitement standard d'illumination et projection par un programme spécifique appelé « vertex shader », les entrées et les sorties du vertex shader sont des sommets et leurs paramètres, plus loin dans le pipeline on peut implémenter un « fragment shader » qui exécute des manipulations complexes sur la couleur résultat de rastérisation pour chaque pixel. [43]

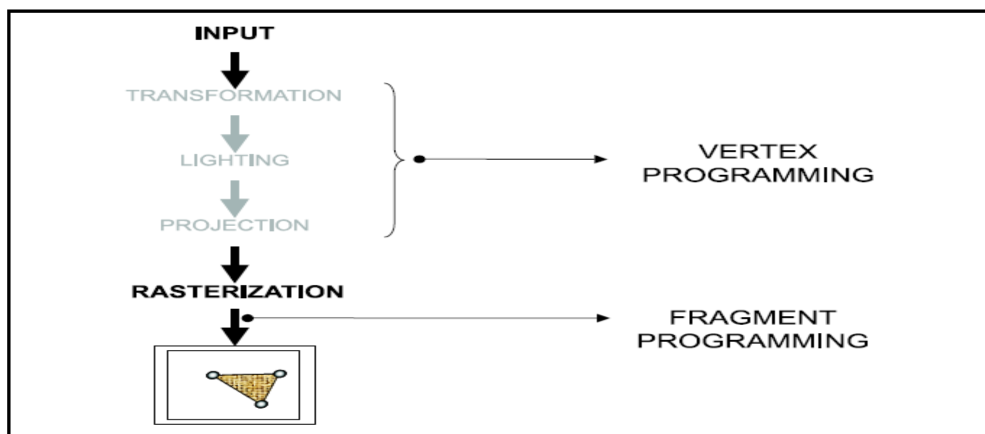


Fig49 :Intégration de Vertex shader et Fragment shader dans le pipeline graphique [43]

Les shaders ont étés développés pour encourager les programmeurs à produire des applications temps réelle avec une meilleur qualité. Le pixel shader peut exécuter des techniques très puissantes avant de produire la couleur finale du pixel, comme le Bump-mapping, Phong-shading, et environnement-mapping, du même le vertex shader peut calculer la position finale du sommet sur l'écran en se basant sur le code du programmeur.

Avec la possibilité de créer des programmes exécutables sur GPU, ces derniers sont devenus capables d'effectuer des tâches qui ne sont pas spécifiques au rendu, ce qui a permis l'apparition d'une nouvelle branche de recherche appelés GPGPU (General Purpose GPU), qui a commencée à inclure des domaines comme l'algèbre linaire, le traitement d'image, et les solveurs multi-grides. Même dans le domaine de la visualisation il existe deux approches qui sont considérés comme du calcul générique sur GPU, parce qu'elles n'utilisent pas le pipeline de rendu des triangles standards, l'implémentation sur GPU de l'illumination globale [45 et 46]et de lancer de rayon [47].

En 2006 NVIDIA à lancer un nouveau niveau de programmabilité au niveau de ses cartes graphiques, appelé géométrie shader. Un géométrie shader commence avec une seule

primitive (point, line, triangle), il peut lire tous les attributs des sommets constituant cette primitive, et les utilisé pour générer de nouvelles primitives. Les primitives générées par le géométrie shader sont par la suite clippées et traitées comme les primitives spécifiées par l'application. [48]

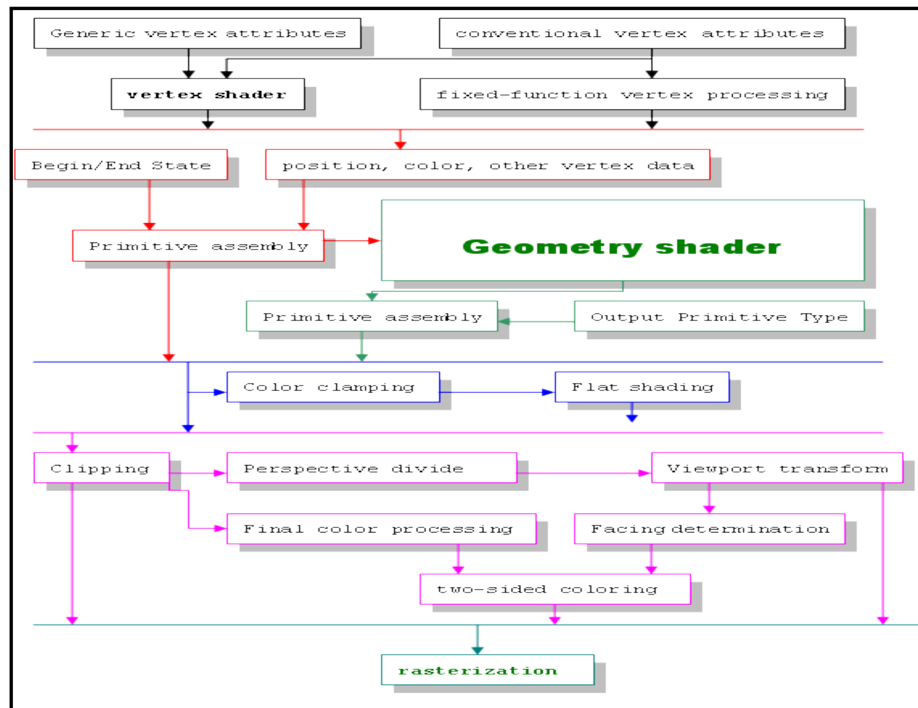


Fig50 : Localisation du géométrie shader dans le pipeline graphique [48]

4.GPUs Programmables :

Les cartes graphiques récentes comportent des processeurs spécialisées pour chaque étape du pipeline graphique, ils peuvent tous fonctionner en parallèle en passant les données à travers les étapes du pipeline (Streaming) et dans chaque étape en fonction du nombre des processeurs disponibles pour chaque étape (plusieurs streaming). [49]

1. Processeur des vertex : une unité programmable qui opère sur les données associées au vertex en entrée, les unités de programme écrits pour s'exécutés sur ces processeurs sont appelés « vertex shaders ». un vertex shader ne peut accéder qu'aux données associées à un seul vertex, et ne peut pas faire des opérations qui nécessitent la connaissance de plusieurs sommets.

2. Processeur de contrôle tessellation : une unité programmable qui opère sur des primitives constitués d'ensembles de sommets, et produit en sortie de nouvelles primitives (ensembles de sommets),les unités de programmes écrits pour s'exécuter sur ces processeurs sont appelées « tessellation control shaders ». le shader de contrôle tessellation est invoqué pour chaque pixel dans les patches de sorties, chaque shader est capable de lire n'importe quelle données des vertex dans les primitives d'entrées ou de sorties, à la fin de toutes les invocations du shader de contrôle tessellation, les vertex et les informations prés-primitive sont utilisées pour produire les primitives a utilisées dans l'étape suivante du pipeline. Les shaders de contrôle tessellation

s'exécute en parallèle, alors la lecture par un shader d'une information de pixel ou primitive écrite par un autre peut avoir des résultats indéterminés, la fonction « barrier() » peut être utilisée pour ordonner l'exécution des shaders, par la synchronisation de leurs invocations, ce qui divise le procédé d'exécution en plusieurs étapes.

3. Processeur d'évaluation de la tessellation : unité programmable qui évalue les positions et d'autres attributs des sommets, en utilisant des patches de vertex générés par le shader de contrôle de tessellation, les unités de programmes exécutables sur ce type de processeur sont « les shaders d'évaluation du maillage ».
Les shaders d'évaluation du maillage sont exécutés pour un seul sommet résultant du générateur du maillage à la fois, ils peuvent lire les attributs de n'importe quel sommet dans le patch d'entrée, plus ses coordonnées du maillage, et peut écrire les coordonnées du sommet et ses attributs.

4. Processeur de géométrie : opère sur les sommets qui ont été assemblés dans des primitives (primitives d'entrées)après avoir été traité par le processeur des sommets, et généré des primitives de sorties, les unités de programmes qui s'exécutent sur ce type de processeurs sont « les Géométrie shaders ».
Une invocation du géométrie shader opère sur une primitive déclarée à l'avance comme primitive d'entrée, avec un nombre limité de sommets, et produit un nombre variant de sommets en sortie qui forment la primitive de sortie déclaré à l'avance aussi, qui seront transmis au reste du pipeline graphique.

5. Processeur de fragment : unité programmable qui opère sur les valeurs et les informations associés aux fragments, les unités de programmes qui s'exécutent sur ce type de processeurs sont appelés « fragment shaders ».
Un fragment shader ne peut pas changer les coordonnées (x,y) d'un fragment, ni accéder aux données des fragment adjacents, il calcule des valeurs qui sont utiliser pour mettre à jour la mémoire frame buffer ou la mémoire texture, en dépendance de l'état courant de l'API, et de la commande API utilisée pour écrire dans le framebuffer .

5. Architecture interne et Parallélisme des GPUs :

Durant leur développement l'architecture des processeurs graphiques s'est adaptée de plus en plus pour être plus performent pour la programmation graphique loin des contraintes imposées sur les processeurs généraux. Cette trajectoire d'évolution a conduit à une architecture hautement parallélisable très performante pour le traitement des flux de données indépendantes, avec certaines restrictions sur les types de ces données, contrairement aux architectures des processeurs généraux, qui traitent des données quelconques souvent dépendantes ce qui implique beaucoup de transitions dédiées au contrôle des flux de données, les processeurs graphiques ne nécessitent pas une unité de contrôle de flux sophistiquée en plus, alors que la latence mémoire est masquée au niveau de processeurs généraux par l'utilisation de la mémoire cache, les processeurs graphiques peuvent masquer cette latence par l'intensité du calcul parallèle.[51]

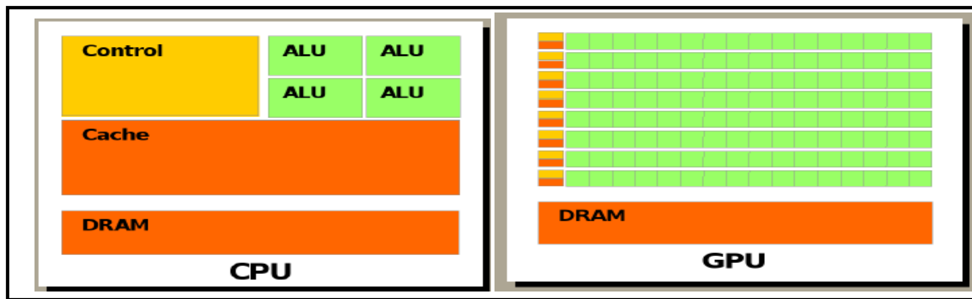


Fig51: Différance architecturale entre CPU et GPU [51]

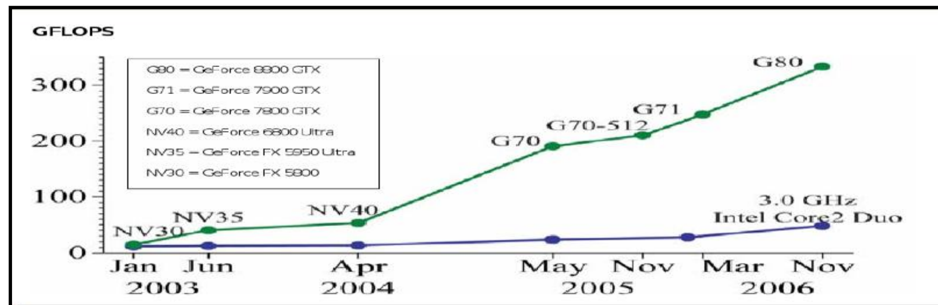


Fig52 : Comparaison des performances CPU et GPU [51]

Les processeurs graphiques forment une nouvelle architecture parallèle. A l'inverse des architectures parallèles des années 1990, leur conception doit répondre à des enjeux économiques importants, et des contraintes imposées par leur utilisation pour des tâches autre que le rendu graphique (GPGPU).[50]

Nom commercial	GPU	Fréquence cœur (MHz)	Fréquence calcul (MHz)	Fréquence mémoire (MHz)
Tesla C870	G80	575	1350	800
GeForce 8500 GT	G86	459	918	400
GeForce 9800 GX2	G92	600	1512	1000
Prototype T10P	GT200	540	1080	900

Fig53 : Caractéristiques de quelque carte graphique [50]

5.1 Architecture général du GPU :

Les GPUs sont conçus selon une architecture pipelinée en étages (plus de deux cents étages), qui suit le pipeline graphique des API, avec différentes unités de calcul.

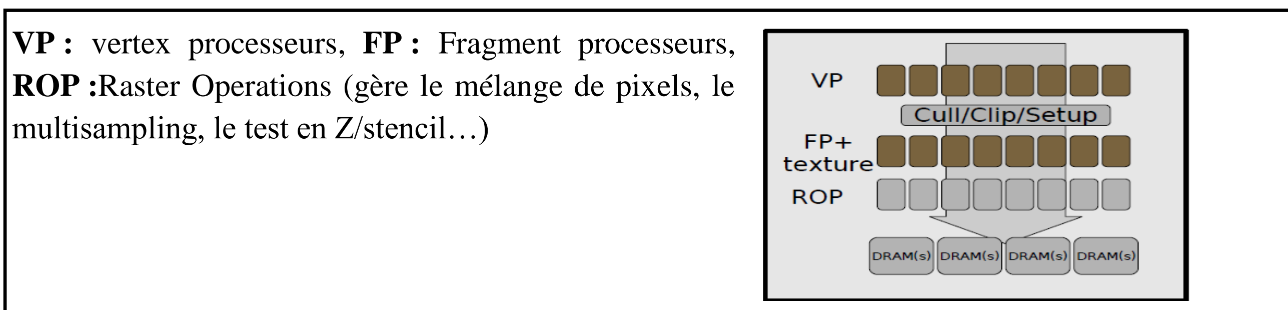

VP : vertex processeurs, **FP :** Fragment processeurs, **ROP :**Raster Operations (gère le mélange de pixels, le multisampling, le test en Z/stencil…)

Fig54 : Architecture générale du GPU [51]

5.1.1. Quelque exemples d'architecture des GPUs :

- **NVDIA G70 (juin 2005) :** son architecture est dérivée du NV40, avec 32 millions de transistors (0.11µ) cadencés a 430 MHZ. Il contient :
  - 8 vertex processors
  - 24 fragment processors (groupés par 4)
  - 24 unités de texture (1 par fragment processor)
  - 16 ROP
  - Contrôleur mémoire 256 bits (4 canaux 64 bits)

Fig55 : Architecture du G70

- **ATI R520/580 (janvier 2006) :**le processeur R580 de ATI (évolution du R520) intégrant 64 millions de transistors (0.09µ) cadencé à 650 MHZ, a introduit ledécouplagedes fragments processeurs des unités de textures, il contient :
  - 8 vertex processors
  - 48 fragment processors(groupés par 4)
  - 16 unités de textures
  - 16 ROP
  - Contrôleur mémoire 256-bits (8 canaux 32-bits

- **NVDIA G80 (novembre 2007) :** dans ce GPU NVIDIA a introduit l'architecture unifiée (pas de séparation entre vertex processeurs et fragment processeurs), il intègre 681 millions de transistors (0.09µ), avec plusieurs domaines d'horloge : 575 /900 /1350 Mhz GPU/DRAM/ALU, il contient :

  - 128 ALU scalaires (groupées par 16)
  - 32 unités de textures (4 par groupe de 16 ALU)
  - 24 ROP
  - Contrôleur mémoire 384-bits (6 canaux 64-bits)

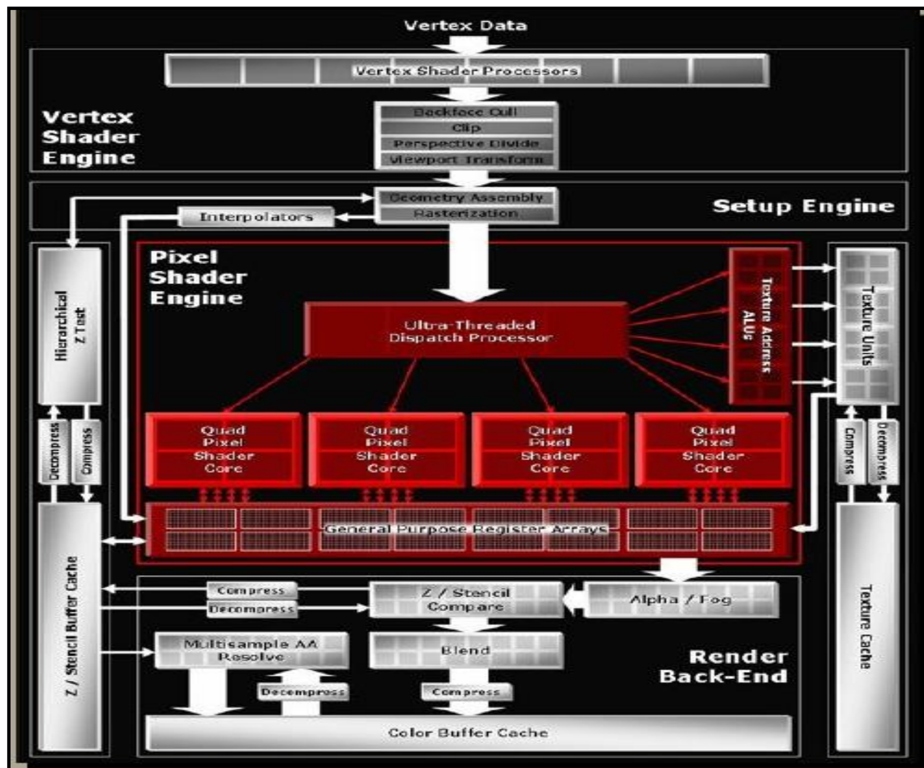


Fig56 : Architecture du R 580 [51]

L'architecture unifiée permet au GPU de faire un équilibrage automatique de la charge, ce qui permet une meilleur distribution des unités de calculs disponibles entre les shaders, et par conséquent une meilleur exploitation des ressources du GPU et une meilleure performance.

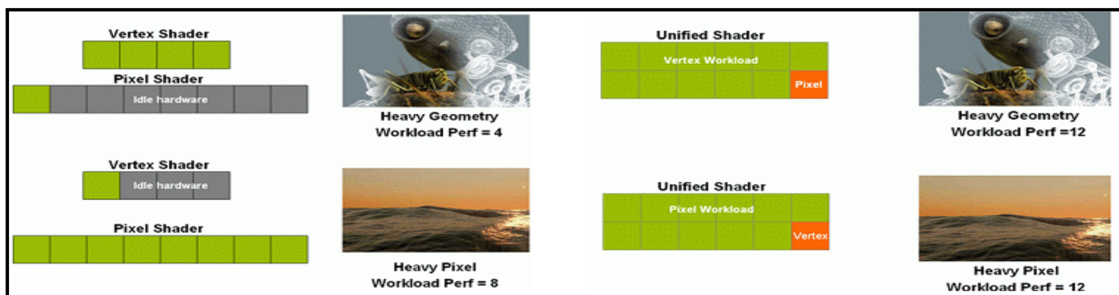


Fig57 : Architecture unifiée [51]

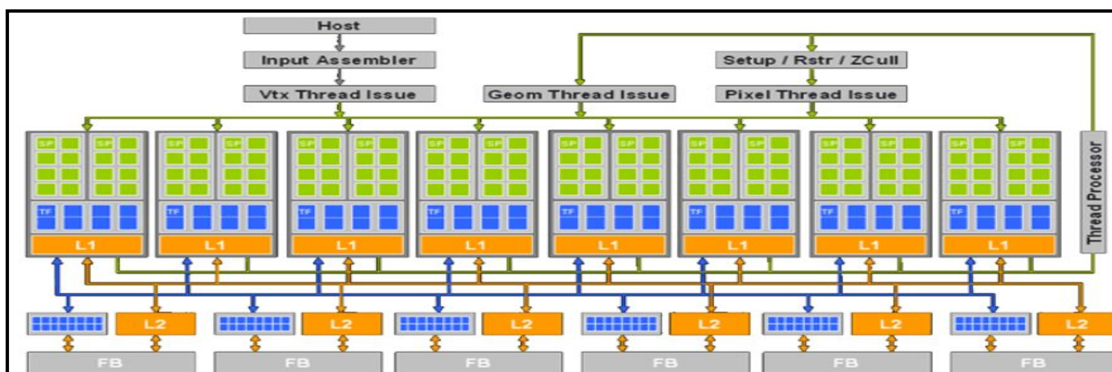


Fig58 : Architecture du G80 [51]

Le G80 a aussi introduit plusieurs améliorations :

- Un nouvel "étage" programmable : Geometry processor
- Buffers de constantes
- Support du format entier
- Opérations bit à bit
- Textures entières
- Shaders unifiés ie : même interface de programmation pour tous types de shaders (avec quelques variantes suivant le type de shader)

5.2. GPGPU :

General Purpose computation on GPU, signifie l'exploitation des GPUspour des applications non graphique. L'apparition de cette nouvelle tendance de programmation a été motivée par les capacités de parallélisme massif des GPU, et les nouvelles architectures GPUs qui se sont éloignées du pipeline des APIs graphiques (architecture unifiée), ce qui est très adapté au applications nécessitant la manipulation de très grandes quantités de données indépendantes, et hautement prallélisables.[53]

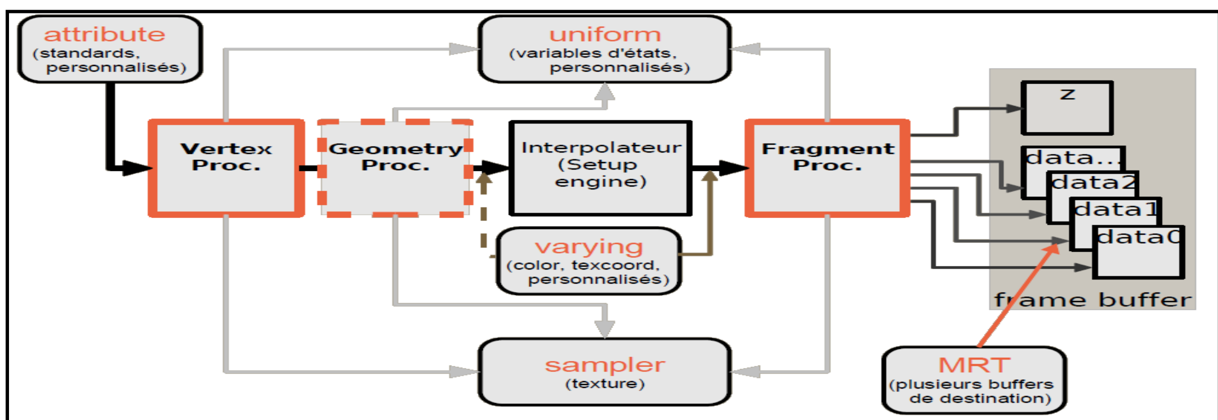


Fig59: GPU pour les applications graphiques [51]

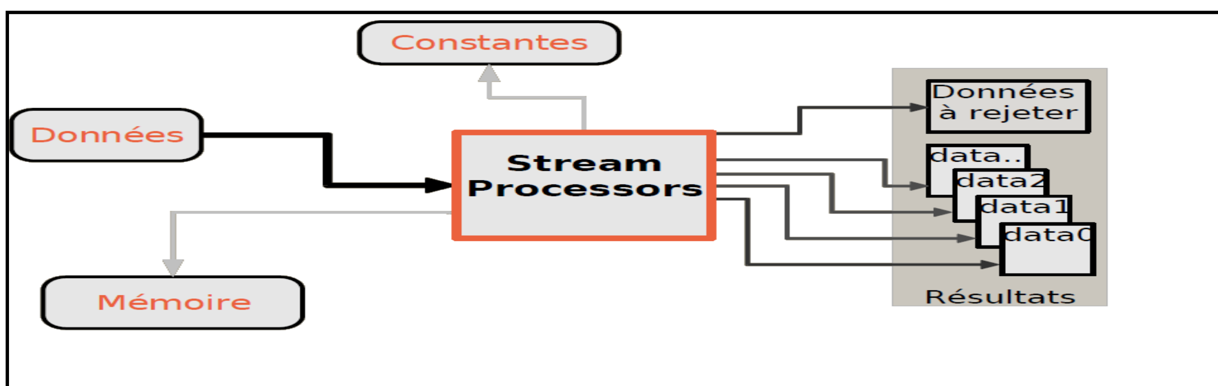


Fig60 : GPU pour GPGPU [51]

5.3. Architecture Tesla :

Tesla est la base architecturale commune aux GPUs Nvidia supportant le GPGPU débutée avec le G80 (GeForce 8800 GTX). Cette appellation correspond à un nom de code associé à l'architecture, et n'a pas de relation directe avec le nom commercial Tesla désignant la gamme destinée au calcul scientifique du même constructeur.[52]

En tant que partie intégrante de la plate-forme CUDA [53], cette architecture a eu une influence majeure sur le développement du GPGPU, au point de devenir un standard. En novembre 2009, NVIDIA recensait ainsi près de 700 travaux académiques et industriels basés sur CUDA et donc Tesla [53].

5.3.1. Jeu d'instructions :

Au niveau architectural, les unités de calcul de tous les GPU Tesla reconnaissent le même jeu d'instruction de base, mais des extensions sont introduites ou retirées en fonction des révisions de l'architecture ou de la gamme de produit.

Par exemple pour les GPUs NVIDIA ces révisions de l'architecture sont désignées par le nom de « Compute Model » que NVIDIA représente sous forme d'un numéro de version.
Le premier chiffre indique l'architecture, et le second le numéro de révision de l'architecture. Chaque révision englobe les fonctionnalités apportées par l'ensemble des révisions antérieures. [50]

Compute model	Fonctionnalités
1.0	Architecture de base Tesla
1.1	Instructions atomiques en mémoire globale
1.2	Atomiques en mémoire partagée, instructions de vote
1.3	Double précision

Fig61 : Révision des architectures NVDIA [50]

Ce jeu d'instruction partage de nombreuses similitudes avec les langages intermédiaires issus de la compilation des shaders de Direct3D (assembleur HLSL5), OpenGL (ARB Fragment /Vertex/ . . . Program6) ou CUDA (tesla PTX [17]). En particulier, il abstrait presque complètement la largeur SIMD architecturale.

Les instructions opèrent toutes sur des vecteurs, y compris les instructions de lecture/écriture mémoire, qui deviennent alors des instructions gather/scatter, et les instructions de comparaison et de branchement, qui reproduisent de manière transparente une exécution proche d'un modèle MIMD. Il n'offre aucune instruction de calcul scalaire, ni de registre scalaire, ni même d'opération de lecture ou écriture mémoire ; le jeu d'instruction est entièrement vectoriel. Du point de vue d'une voie SIMD, les calculs effectués sont indépendants des autres voies du vecteur SIMD, et peuvent se décrire comme si le mode d'exécution était scalaire, ou MIMD.
Ce modèle étant très proche de celui qui est exposé au programmeur par les langages de développement sur GPU, le processus de compilation reste aisé et efficace.

Au niveau architectural, l'espace mémoire de Tesla est hétérogène, divisé en espaces distincts adressés explicitement. Chaque espace est accessible au moyen d'instructions distinctes. La

décision de placement des données ne peut donc être faite qu'au moment de la compilation. En particulier, le compilateur doit être capable d'inférer statiquement l'espace mémoire désigné par chaque pointeur. [50]

Ces espaces logiques comprennent :

- la mémoire de constantes,
- la mémoire de texture,
- la mémoire globale,
- la mémoire locale,
- la mémoire partagée.

Les mémoires de textures et de constantes sont dédiées aux langages de shaders pour la programmation graphique, qui y font référence explicitement. Effectuer la distinction entre ces espaces mémoire au niveau du jeu d'instruction est une décision naturelle permettant une mise en œuvre efficace de l'architecture et une traduction simplifiée pour le compilateur. En revanche, la séparation entre les mémoires partagée, globale et locale destinées principalement au calcul généraliste nécessite un effort supplémentaire de la part du compilateur (en C pour CUDA) ou du programmeur (en OpenCL).

5.3.2. Organisation générale :

La partie calcul est composée d'une hiérarchie de cœurs (tesla)[12, 10]. Au niveau le plus élevé, le GPU contient jusqu'`a 10 TPC (texture / processor cluster) connectés aux autres composants par réseau d'interconnexion en croix. Chaque TPC contient une unité d'accès a la mémoire, disposant de son propre cache de premier niveau et de ses unités de calcul d'adresse et filtrage de texture, et plusieurs cœurs de calcul ou SM (streaming multiprocessors) ou encore multiprocesseurs. Chacun de ces SM est un processeur autonome SIMD multithread.

Domaines d'horloge : Le GPU est divisé entre plusieurs domaines d'horloges en fonctions des compromis entre latence, débit, surface et consommation souhaités pour chaque composant matériel. Les unités de calcul (SP) fonctionnent a la fréquence la plus haute. Le reste des SM est animé par une horloge de fréquence divisée par deux. Les unités non dédiées au calcul généraliste tel que le réseau d'interconnexion et l'ordonnanceur global fonctionnent a une fréquence indépendante, généralement inférieure. Enfin, les contrôleurs mémoires suivent l'horloge de la mémoire externe, en général de type GDDR3.

Souplesse et passage l'échelle : Les différentes microarchitectures de l'architecture Tesla ont suivi des évolutions progressives. La stratégie généralement suivie consiste à introduire une nouvelle révision majeure dans le segment haut-de-gamme, pour ensuite réutiliser sa microarchitecture pour les segments milieu et entrée de gamme en réduisant progressivement le nombre de TPC et de partitions mémoire.

Ces contraintes commerciales nécessitent que l'architecture soit capable de passer à l'échelle vers le bas : les dérivés doivent rester compétitifs pour un cout très réduit, et les unités et partitions mémoires doivent pouvoir être désactivées et le réseau d'interconnexion reconfiguré de manière transparente.

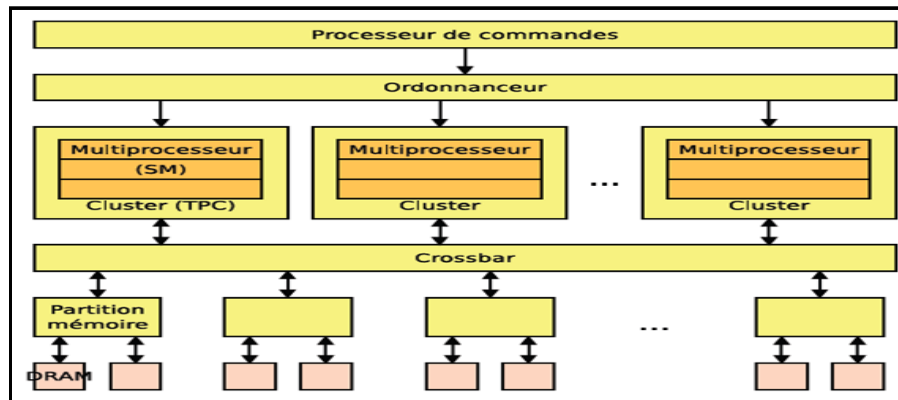


Fig62 : Vue générale de l'architecture Tesla [50]

5.3.3. Interface :

Le Tesla étant un coprocesseur spécialisé, il n'est pas capable de fonctionner de manière autonome. Il est dirigé par un pilote s'exécutant sur le CPU. Celui-ci contrôle l'état (registres de configuration) du GPU, lui envoie des commandes, et peut recevoir des interruptions indiquant la complétion d'une tache.

Processeur de commandes : Pour fonctionner de manière asynchrone par rapport aux CPU, le GPU est contrôlé au travers d'une file de commandes selon un schéma producteur-consommateur classique. La file est typiquement placée en mémoire centrale. Le pilote graphique ou GPGPU se charge d'ajouter des commandes dans la file, tandis que le processeur de commandes du GPU retire les commandes à l'autre extrémité de la file.

Les commandes peuvent être des commandes de configuration, des copies mémoires, des lancements de kernels. Pour permettre à plusieurs contextes d'exécution de coexister, et pour exploiter du parallélisme en recouvrant copies mémoire et exécution sur le GPU, plusieurs files de commandes virtuelles peuvent être utilisées. Le basculement du GPU d'un contexte d'exécution à l'autre se fait par exécution de microcode.

5.3.4. Répartition du travail :

L'ordonnanceur est l'unité du GPU chargée de répartir les blocs (groupes de threads garantis de s'exécuter sur le même multiprocesseur) a exécuter sur les multiprocesseurs (tesla) [16]. Avant d'envoyer un signal de début d'exécution aux multiprocesseurs, il procède à l'initialisation des registres et des mémoires partagées.

En particulier, il doit initialiser :

– les arguments du kernel,
– les coordonnées du bloc,
– la taille des blocs et de la grille,
– les coordonnées du thread à l'intérieur du bloc.

Les arguments et les coordonnées du bloc sont communs à tous les threads d'un bloc. Ils peuvent donc être placés en mémoire partagée. Ainsi, les 16 premiers octets de la mémoire partagée sont initialisés par 8 entiers 16 bits contenant les informations de dimension. Les arguments sont

stockés aux adresses immédiatement supérieures. Il est à noter que certaines de ces données sont constantes, et en tant que telles, pourraient être passées par la mémoire de constantes. Le choix qui a été fait s'explique probablement par le coût d'une initialisation de la mémoire de constantes.

5.3.5.Front-end

La capacité de masquer les latences des diverses opérations tout en conservant un débit soutenu élevé repose sur l'exploitation par les multiprocesseurs de multithreading simultané (SMT) massif, par exemple le G80 et le GT200 deNVDIA maintiennent respectivement 24 warps (fil d'exécution d'instructions SIMD) et 32 warps en vol simultanément.

A la différence des supercalculateurs vectoriels des années 90 ou une même instruction est exécutée sur des vecteurs significativement plus longs que la largeur des unités de calcul, cette approche permet de désynchroniser l'exécution des sous-vecteurs (warps), offrant un fonctionnement de type MIMD au sein d'un cœur de calcul.

Il existe des processeurs super scalaires conventionnels qui gèrent le SMT, mais se limitent a deux voire quatre threads. Au-delà, le gain de performance additionnel justifie difficilement le cout en matériel et la complexité de la conception, et du multithreading a basculement sur événement est préféré.

Tesla n'utilise pas le multithreading a basculement sur événement est préféré, alors Pour atteindre efficacement un tel niveau de parallélisme, l'architecture des multiprocesseurs est conçue d´es l'origine pour le SMT. En contrepartie, cette architecture est inefficace pour les charges de type monothread.

Ainsi, lorsqu'un seul warp est en cours d'exécution, le multiprocesseur n'est capable d'exécuter qu'une instruction tous les 8 cycles SP, contre un CPI de 2 lorsque le multiprocesseur est saturé. De plus, Tesla se limite à l'exécution de code SPMD, qui offre plus de régularité et de localité que l'exécution simultanée de threads arbitraires.
Plutôt que d'ajouter de la complexité au système, le SMT apporte ici à l'inverse de la régularité et du parallélisme permettant de simplifier l'architecture.(tesla)

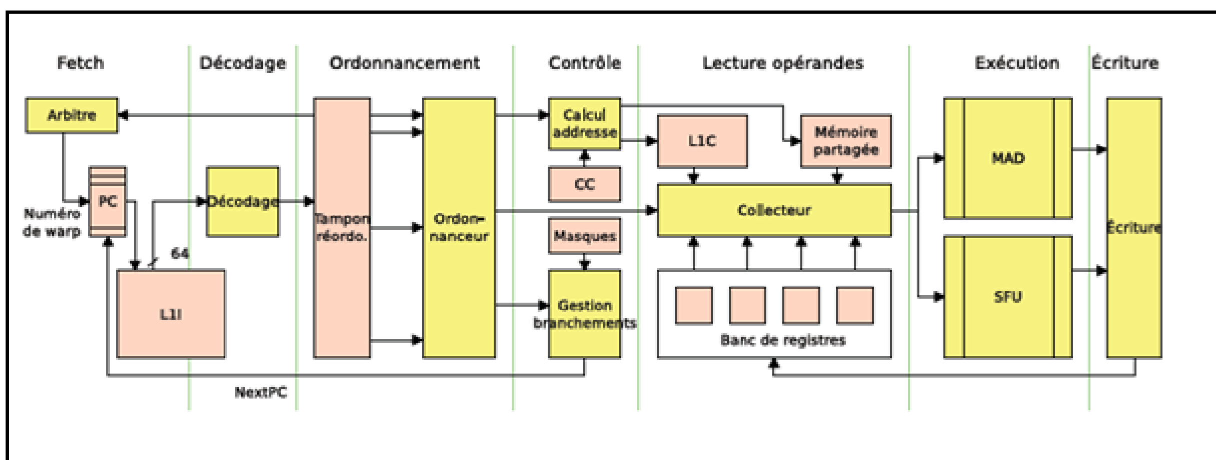


Fig63 : Vue d'ensemble du pipeline d'exécution d'un GPU Tesla [50]

5.3.6.Hiérarchie de mémoire :

Il existe des chemins d'accès distincts et autant de hiérarchies mémoires pour la mémoire de textures, la mémoire de constantes, la mémoire d'instructions et la mémoire partagée. Cependant, pour des raisons de réutilisation de matériel, les espaces mémoires globaux et locaux partagent la majeure partie du chemin d'accès avec la mémoire de textures.

Mémoires internes :

Chaque SM dispose :

– d'un cache de constantes de premier niveau
– d'un cache d'instruction de premier niveau,
– d'une mémoire partagée.

Les constantes et les instructions étant des données accessibles en lecture uniquement, elles peuvent être dupliquées a plusieurs emplacements parmi le sous-système mémoire sans nécessiter de protocole de maintien de la cohérence.

De plus, ces données peuvent être considérées comme scalaires. Une seule instruction SIMD est nécessaire pour effectuer une opération vectorielle, par définition. Le débit effectif que le cache d'instructions doit fournir est donc réduit à une instruction par cycle. De même, les données lues en mémoire de constantes sont typiquement des scalaires, qui sont « étalés » dans un registre vectoriel lors de la lecture. Il est donc possible au cache de constantes de se contenter d'un unique port de 32 bits.[Tesla)

En revanche, la mémoire partagée doit permettre de stocker temporairement et accéder rapidement `a des données vectorielles. C'est également au travers de cette mémoire que sont effectuées les opérations d'échanges entre voies SIMD (swizzle), plutôt qu'au moyen d'instructions dédiées comme dans d'autres architectures SIMD (tesla) [8].

Pour être capable de nourrir les unités de calcul sans devenir un goulot d'étranglement, la mémoire partagée doit offrir l'équivalent de 16 ports de 32 bits en lecture-écriture.

Une telle mémoire n'étant pas raisonnablement réalisable, la mémoire partagée est composée de 16 bancs indépendants. Chacun de ces bancs ne possède qu'un unique port de lecture-écriture. Un arbitre se charge de simuler la présence de 16 ports capable d'identifier les requêtes concurrentes a une cellule identique et de les fusionner, dans la limite d'une fusion par cycle.

En dehors du risque de conflits de bancs, les accès avec indirection ne sont pas plus couteux que les accès avec adressage absolu, grâce à l'utilisation de registres d'adresse dédiés dont la valeur est disponible plus tôt dans le pipeline que celle des registres généraux.

Mémoires niveau cluster :

Le TPC regroupe le cache de constantes de deuxième niveau et les unités d'accès mémoire, incluant les caches de texture de premier niveau et les unités de filtrage de texture.

La mémoire de texture sert typiquement à effectuer des lectures à un emplacement calculé à partir de coordonnées bidimensionnelles, suivi d'un filtrage éventuel. Ce type d'accès offre une forte localité spatiale. Un cache de texture agit comme un filtre pour fusionner les lectures similaires et adjacentes. Les accès redondants sont éliminés, et la granularité des accès augmentée, ce qui permet une meilleure exploitation de la bande passante mémoire.

Toutes les lectures de texture sont des opérations de type gather sur 16 données distinctes voire plus, et l'exécution des instructions SIMD dépendant du résultat ne peut se poursuivre tant que l'ensemble des résultats ne sont pas disponibles. Ce fait diminue significativement le bénéfice qu'un cache peut avoir sur la latence des accès.

Pour obtenir un gain de latence, il faut que toutes les lectures composant l'opération gather soient des succès dans le cache, événement dont la probabilité est faible. De fait, le cache de texture est conçu en priorité comme un moyen d'augmenter le débit effectif plutôt que de diminuer la latence.

Dans l'architecture Tesla, le cache de texture intégré a chaque TPC est partagé statiquement entre chaque SM, et se comporte donc comme deux ou trois caches indépendants.[50]

6. Programmation GPU pars haders :

La programmation des GPUs est faite par des petits programmes, qui s'exécutent dans certaines étapes du pipeline de la carte graphique, l'étape de transformation, illumination, et la rastérisation, ces programmes sont :

6.1.Vertex shader :

Un programme qui s'exécute pour chaque sommet transmit par l'application (qui s'exécute sur le CPU), en parallèle en dépendance du nombre de vertex processeurs présents sur la carte. [49]

Le vertex shader peut effectuer les taches : [51]

- Les transformations de modélisation et projection du sommet attribué.
- Calcul d'éclairage par sommet.
- Génération et transformation des coordonnées de texture
- Calcul de la taille de la primitive « point ».
- Calcul des plans de clipping supplémentaires.

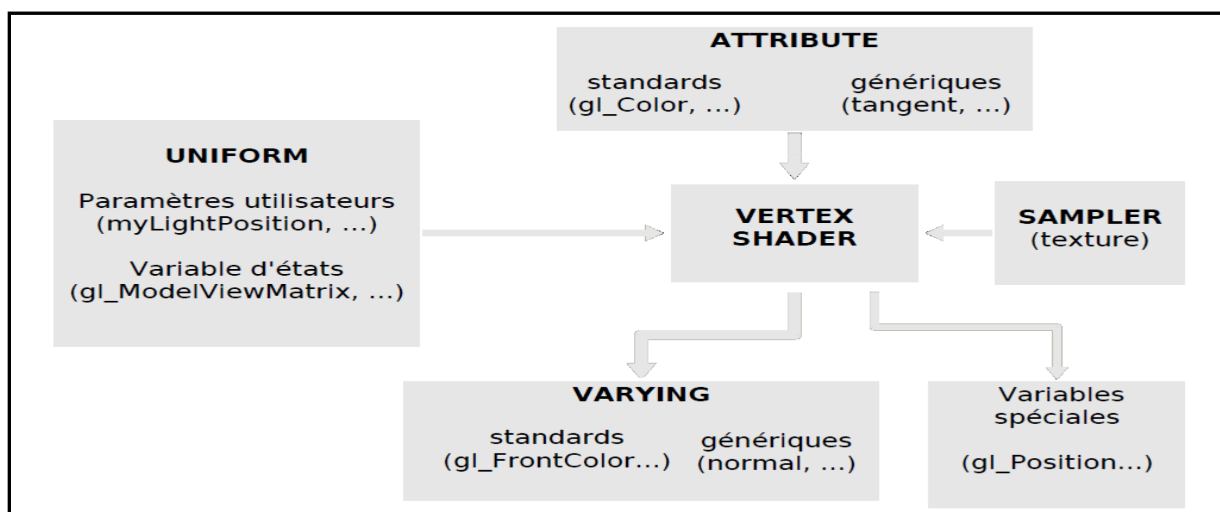


Fig64 :Environnement d'exécution du vertex shader [51]

6.2.Geometry shader :

Sa présence est optionnelle. S'il est présent il s'exécute après le vertex shader sur chaque primitive dont tous les sommets ont étés traités par ce dernier, ces primitives doivent être du même type. Le degré du parallélisme ici dépend du nombre de géométrie processeurs présents sur la carte graphique, et le nombre de primitives prés (tous leurs sommet ont étés traités par le vertex shader) a un instant donné. Le programme a accès à tous les attributs de tous les sommets formant une primitive.

La capacité de génération de nouveaux sommets du géométrie shader est limité et doit être indiqué lors de la compilation du shader, ce nombre dépend du modèle de la carte graphique et la taille des sommets à produire (le nombre d'attributs des sommets), le minimum de sommets que peut un géométrie shader produire (sur les cartes qui supporte le géométrie shader) est 255 sommets avec deux attributs vecteurs (un vecteur position et un vecteur normal par exemple), si d'autres attributs sont ajoutés le nombre maximal de sommets qui peuvent êtres générés diminue.[49]

Un géométrie shader peut effectuer les tâches suivantes :[51]

- La transformation des primitives.
- Accès aux attributs des sommets de la primitive attribuée.
- La création d'un certain nombre de nouvelles primitives.
- Rendu par couches, par rendu dans des textures 3D, Cube Map, ou Texture Array.

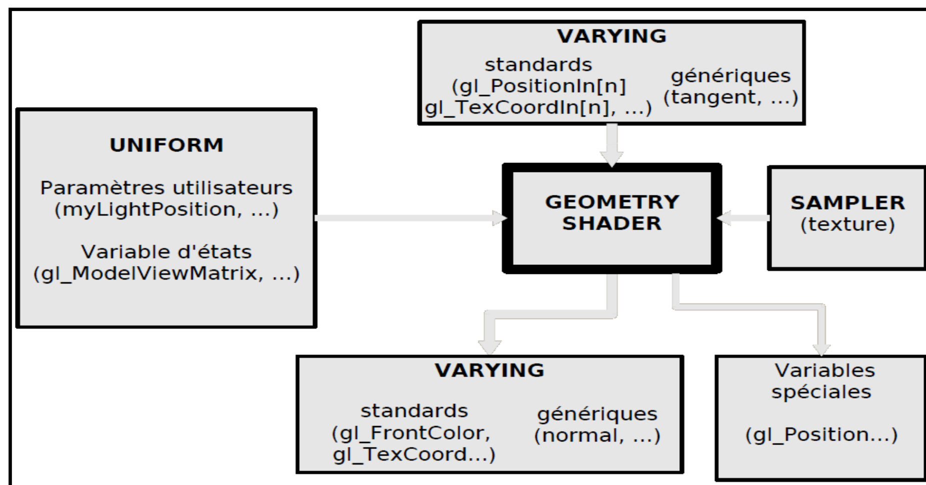


Fig65 : Environnement d'exécution du Géométrie Shader [51]

6.3. Fragment shader :

Ce programme s'exécute une fois pour chaque fragment résultant de la rastérisation, si on considère les pixels, un fragment shader peut s'exécuter plusieurs fois par pixel (plusieurs fragments peuvent être combinés pour produire la couleur d'un pixel), le degré de parallélisme d'exécution du fragment shader dépend du nombre de fragment processeurs présents sur la carte. [49]

Un fragment shader peut effectuer les tâches : [51]

- Accès et application de textures.
- Adition de couleurs et application du brouillard
- N'importe quelle opération sur les valeurs interpolées.
- Supprimer un fragment.

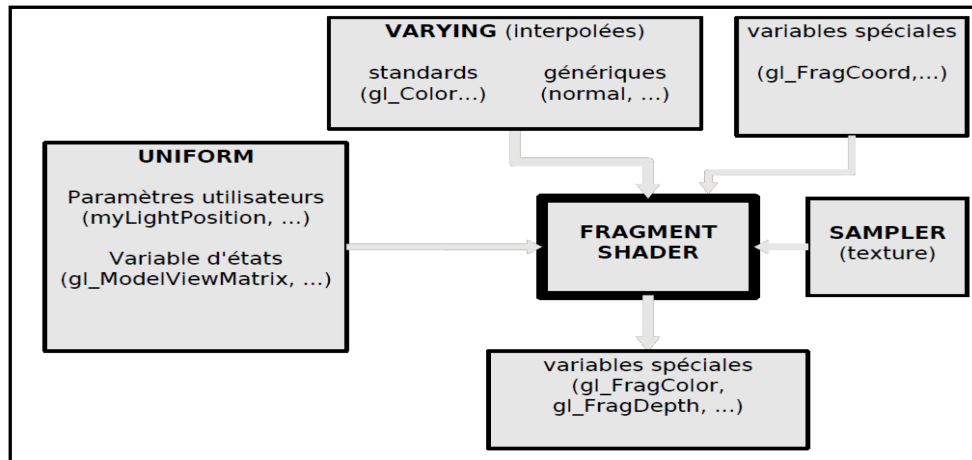


Fig66 : Environnement d'exécution d'un fragment Shader [51]

6.4. Langages de programmation des GPU :

Les langages de programmation des GPUs peuvent être classés dans deux catégories : les langages d'assemblage, el les langage haut niveau, ces langages sont très profondément inspiré du langage Renderman, qui est un langage de programmation de shaders mais pour les application non-temps réelle.

Le groupe ARB[57] a proposé deux langages pour les deux types de shaders : ARBVP pour le vertex shader et ARBFP pour le fragment shader, l'avenage de cette standardisation est que toutes les producteurs doivent implémenter les instructions définies, et aussi ils peuvent offrir leurs propres langages d'assemblage, par exemple NVIDIA a développé son propre langage d'assemblage NVVP et NVFP.

Les langages haut niveau de programmation de GPU ont une syntaxe similaire à celle du langage C, les plus connus sont : CG de NVidia, GLSL par ARB groupe, et Microsoft HLSL, CG et HLSL ont un compilateur qui peut produire plusieurs langages d'assemblages, incluant ARBVP et ARBFP, GLSL a un concept différant, la compilation est effectué par le driver de la carte graphique, cette approche a deux avantages : le driver peut vérifier n'importe quelle extension ou restriction du GPU, et quand le driver est mis à jour, les amélioration du driveur peuvent être exploité ; d'un autre coté quand la compilation est faite par CG vers un langage d'assemblage, il est possible de produire le code du shader une fois pour tout, permettant ainsi une meilleur optimisation.

6.5. Optimisation de programmation des shaders :

Pour une utilisation effaçasse des capacités de cette architecture il faut pondre en considération ses caractéristique lors de la programmation, et penser d'une façon différant de celle utilisée pour les CPU classique :

- Les GPUs sont très effaçasses pour l'exécution de plusieurs threads lèges, mais ils fournissent un très faible rendement pour un seul thread lourd.
- L'architecture des GPUs est conçues pour opérer sur des vecteurs, alors l'utilisation des vecteurs permet un exploit maximale de leurs capacités de calcule. Quand des opérations sont appliquées sur des scalaires, ces scalaires sont transformés en vecteurs de taille un et traités comme des vecteurs alors il vaut mieux penser dès le début à utiliser des vecteurs pour maximiser les résultats par opération.
- les GPU sont, destinés à et spécialisés pour des calculs intensifs. Ceci leur permet de réserver plus de transistors au traitement des données, au lieu de les utiliser pour le cache ou bien pour la gestion des flux d'entrée ou de sortie, alors il est recommandé de minimiser au maximum les opérations d'entrées sorties.

Conclusion :

Les GPUs récents offrent essentiellement deux aspects très important :

- Une énorme capacité de calcule parallèle : mais leur architecture spécifique (extrêmement parallèle) impose des contraintes de spécification et programmation des algorithmes qui doivent être respectées afin de pouvoir exploiter ces capacités.

Les algorithmes dédiés a l'implémentation sur GPU doivent être aussi parallèle que possible, et les taches granulaires formant les bloques séquentielles qui seront exécutés sur les ALUs du GPU doivent être aussi réduites que possible, pour pouvoir occuper un maximum d'unités de calcule pendant l'exécution.

Une contrainte importante aussi est la gestion de stockage, comme on a dit précédemment les unités de calcules présentes sur les GPUs ne possèdent pas des registres de calcules pour les variables locaux au instances des shaders, a la place il existe un espace de stockage commun (Memory Pool) dont les branchements sont reconfigurables, qui est repartie selon les besoins des shaders en cours d'exécution, alors si le nombre de variables locaux déclarées dans les instances des shaders excède la capacité de la mémoire commune, le nombre d'unités de calcules actives au niveau du GPU va être réduit en fonction de la mémoire disponible, pour éviter ce cas le nombre des variables locaux des shaders doit être réduit au minimum possible.

Les GPUs étant organisés sous forme de clusters de bloques d'ALUs possédant chacun un seul ordonnanceur (compteur d'instructions), ne sont pas efficaces pour l'exécution des branchement conditionnels, cars ces dernier s'exécutent en mettant en attente les ALUs pour les quelles la condition n'est pas vérifiée jusqu'au retour du branchement (ou la fin de la

boucle), alors si un shader exécute des branchement conditionnels on doit s'assurer que les lots de données envoyés aux GPU sont regroupés de sorte a ce que l'évaluation des conditions de branchements change le moins souvent possible.

- Flexibilité de programmation : cette capacité doit être exploitée pour adapter le fonctionnement du GPU aux besoins spécifiques de l'application, cette spécification permet une meilleure exploitation du GPU, en écartant tout fonctionnalités qui n'est pas nécessaire a l'application, et offre la possibilité d'implémenter directement sur le GPU n'importe quelle technique de rendu, en modifiant des étapes du pipeline graphique.

Chapitre 3:
Instanciation de géométrie par GPU

Chapitre 3

Instanciation de géométrie par GPU

Introduction

L'instanciation de la géométrie est une méthode qui permet de dupliquer un modèle géométrique (un mesh, un ensemble de sommets...) plusieurs fois à partir des attributs des instances. Les attributs des instances fournissent une description des modifications que doit subir le modèle pour générer une instance spécifique (matrice de transformation, couleur, texture…).

Cette méthode est utilisée pour enrichir l'expérience de l'utilisateur en ajoutant de très grands nombres de petits objets, similaires mais avec des différences qui n'infectent pas la forme (la position, l'orientation, la couleur, la taille…). [54]

Les APIs, comme OpenGl et Direct3D, ne sont pas conçus pour supporter le rendu d'un nombre restreint de polygones de façon efficace, un très grand nombre de fois, alors l'utilisation de cette technique doit exploitée la flexibilité offerte par les GPU récents [55].

1. Méthode et implémentation:

GPU Gems 2 [54] donne une définition les concepts de l'instanciation de la géométrie :

a) **Pack de géométrie :** Les API graphiques (comme OpenGL et Direct 3D) sont optimisés pour le transfère directe de sommets et des indexes qui seront dessinés, les attributs per-vertex des sommets au vertex stream du GPU, où chaque sommet est décrit par ces propres attributs comme sa position, coordonnés de texture, normale, couleurs, plus les attributs optionnels comme l'espace tangent, des informations bones data pour le skining, en terme des indices.
Le pack de géométrie en cas de l'instanciation de géométrie est un paquet d'information qui décrit un modèle à partir du quel vont être crées les instances, avec des sommets définis dans le repère locale du modèle sans informations explicites concernant le contexte dans lequel ils vont être rendus, comme le nombre de sommets, leurs positions dans le repère du modèle … .
b) **Attributs d'instances :** Typiquement ce sont les informations nécessaires pour le passage du repère local du modèle au repère globale (world space) qui sont généralement regroupées dans une matrice de transformation, plus la couleur.
c) **Instance :** Une instance est l'association d'un paquet de géométrie a une instance spécifique des attributs d'instance, donnant ainsi une description presque complet d'un objet a dessiné, et qui sera transmis au GPU pour le rendu.
d) **Contexte de rendu et de textures :** Comporte les états de rendu du GPU, en terme des textures actives, les tests comme lez-buffer, alpha blinding …, qui sont

sauvegardés et appliqués pour l'instanciation de toutes les copies du modèle géométrique lors de rendu.

e) **Batchs de géométrie :** Un batch est une collection d'un ensemble d'instances, un modèle et un contexte de rendu, qui sont rassemblé pour être transférés au GPU par un seul appel à la fonction de dessin API.

2. Implémentation de l'instanciation de géométrie :

GPU gems présente plusieurs implémentations possibles pour l'instanciation de la géométrie, qui diffèrent dans le type d'informations transmises au GPU et la méthode de transfère. [54]

2.1 Statique batching :

Dans cette implémentation toutes les instances sont transformées une seule fois, puis transférés au GPU dans un vertex buffer statique réservé au niveau de la mémoire graphique, en exploitant les possibilités offertes par l'API pour indiquer au driver de la carte graphique d'utiliser la mémoire la plus rapide, puisque il y aura pas de mis ajoure du contenu des buffers, la tache unique qui sera exécutée per-frame est un appel à la fonction du dessin de l'API permutant le rendu du contenu du buffer.

Le statique batching est l'implémentation la plus performante de l'instanciation de la géométrie, par ce qu'elle ne comporte aucun transfert des données, ni de calcule per-frame pour transformer les instances au repère globale (world space), mais elle souffre de :

- Une grande consommation de mémoire : en fonction de la taille des modèles géométriques et le nombre d'instances, la quantité de mémoire nécessaire pour cette implémentation peut être très grande et peut dépasser la capacité de mémoire graphique, si c'est le cas l'utilisation de la mémoire système devient impérative, ce qui diminue largement la performance.
- Ne supporte pas différents niveaux de détails : par ce que toutes les instances sont transformées et passées au GPU une seule fois. L'utilisation du même niveau de détails pour toutes les instances conduit au traitement de beaucoup de géométrie, ce qui est très coûteux en temps de calcul.

Une solution peut être de calculer pour chaque instance plusieurs niveaux de détails, et les mettre tous dans le vertex buffer puis passer les indices de sommets a dessinés pour chaque frame, mais cette solution induit une utilisation très excessive de la mémoire ainsi que le transfert d'une quantité d'information qui peut être très grande en fonction du nombre d'instances et la taille du modèle géométrique a instancier, diminuant ainsi la performance de l'implémentation.

2.2 Dynamique batching :

Dynamique batching résout les problèmes du statique batching au coût de la performance, dans cette implémentation des vertex buffers moins performants sont utilisés mais qui peuvent être dynamiquement mis à jour, en fonction de la taille des modèles et le nombre d'instances à rendre on distingue deux approches pour appliquer cette implémentation :

- si la mémoire graphique est suffisante pour contenir la géométrie de toutes les instances, alors la phase de mise à jour des buffers va être complètement séparée de la phase de rendu, et fera la transformation de toutes les instances. La phase de rendu exécutera juste un appel de la fonction de l'API, pour rendre le contenu des buffers.

- la deuxième approche nécessite l'alternance entre le streaming de la géométrie (mise à jour des buffers) et le rendu, les buffers sont remplis de géométrie des instances et quand un buffer est plein, son contenu est transmis pour le rendu, puis il est effacé et prêt pour recevoir d'autres instances et ainsi de suite jusqu'à ce que toutes les instances sont rendues. Cette approche est plus lente que la première mais elle permet de rendre une quantité de géométrie plus grande que la mémoire graphique disponible dans chaque frame.

2.3 Instanciation par vertex constants :

Les instances sont générées à partir d'un vertex buffer constant qui contient plusieurs copies de la même géométrie (une copie par instance), ou à chaque sommet est ajouté un index qui détermine à quelle instance appartient ce sommet, cet index localise les attributs per-instance qui seront utilisés pour compléter ses attributs.

Le vertex buffer contient typiquement les positions des sommets dans un repère locale (model space), qui sont transformées pendant le rendu en utilisant les informations stockées dans la mémoire des constantes des sommets, qui contient d'autres per-instant data comme la couleur de l'instance par exemple.

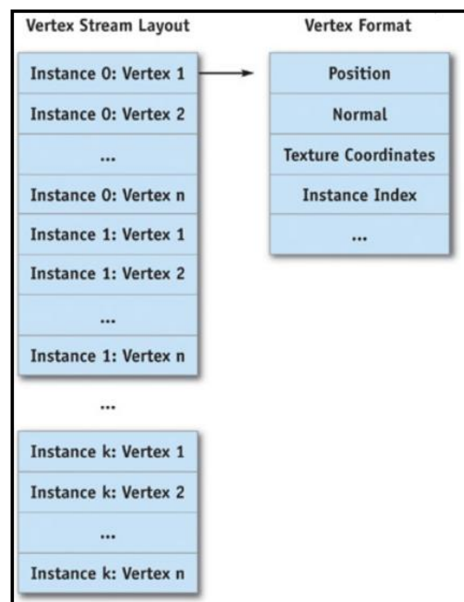


Fig67 : Streaming pour l'instanciation en utilisant les vertex constants [54]

La transformation des sommets transférés au GPU est faite par un vertex shader qui combine le contenu des constantes de sommets appropriées pour chaque instance avec le contenu du vertex buffer pour traités les sommets qui seront passées au reste du pipeline graphique.

Cette implémentation est très performante en termes de vitesse de rendu, mais elle souffre de la limitation du nombre d'instances par batch est limité par la taille de la mémoire réservée au vertex constants, qui est habituellement entre cinquante et sent valeurs. Malgré cette limitation l'utilisation de cette technique permet une réduction significative de la charge du CPU, de l'exécution excessive des appels de dessin de l'API.

```
structvsInput
{
float4 position : POSITION;
float3 normal   : NORMAL;
// other vertex data
int4instance_index :BLENDINDICES;
};
vsOutputVertexConstantsInstancingVS(invsInput input)
{
// get the instance index; the index is premultipliedby 5 to take account of the number of constants
// used by each instance
intinstanceIndex = ((int[4]) (input.instance_index))[0];
// access each row of the instance model matrix
float4 m0 = InstanceData[instanceIndex + 0];
float4 m1 = InstanceData[instanceIndex + 1];
float4 m2 = InstanceData[instanceIndex + 2];
float4 m3 = InstanceData[instanceIndex + 3];
// construct the model matrix
float4x4modelMatrix = { m0, m1, m2, m3 };
// get the instance color
float4instanceColor = InstanceData[instanceIndex + 4];
// transform input position and normal to world space withthe instance model matrix
float4worldPosition = mul(input.position, modelMatrix);
float3worldNormal = mul(input.normal, modelMatrix);
// output position, normal, and color
output.position = mul(worldPosition, ViewProjectionMatrix);
output.normal = mul(worldNormal, ViewProjectionMatrix);
output.color = instanceColor;
// output other vertex data
}
```

Fig68. Algorithme d’instanciation de géométrie en utilisant les constantes des sommets [54]

2.4 Batching en utilisant l’instanciation par API :

Cette technique exploite la structure des APIs qui permettent d’utiliser plusieurs vertex streams implémentés par Directx 9 et OpenGL 3.0, et totalement implémentée par hardware sur les GPU de série six de Nvidia, elle résolve les limitations imposées par l’instanciation par vertex constants.

L’instanciation par API utilise deux vertex buffer, un buffer statique pour stocker le paquet de géométrie, qui va être dupliqué plusieurs fois, et un autre dynamique pour y mettre les attributs d’instances.

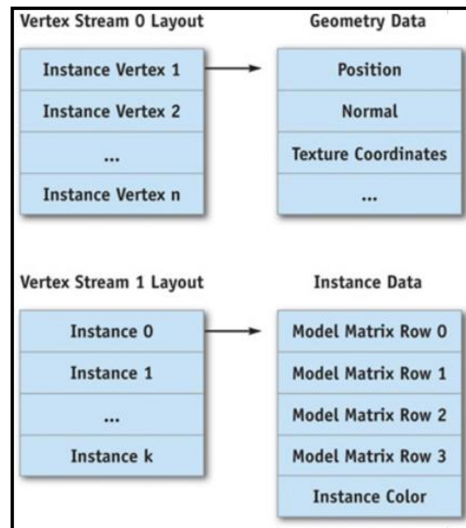


Fig69. Vertex buffers pour l'instanciation de géométrie par API [54]

Le GPU s'occupe de la duplication virtuelle des sommets du premier buffer, en les combinant avec les informations des instances dans le deuxième buffer par un vertex shader. Cette technique, est optimisée pour une implication minimale du CPU dans la tache de rendu, et pour une utilisation de mémoire raisonnable, mais elle est limité par un seul modèle de géométrie possible, il est impossible d'utiliser plus d'un seul paquet de géométrie.

```
structvsInput
{
// stream 0
float4 position : POSITION;
float3 normal   : NORMAL;
// stream 1
float4 model_matrix0 :TEXCOORD0;
float4 model_matrix1 :TEXCOORD1;
float4 model_matrix2 :TEXCOORD2;
float4 model_matrix3 :TEXCOORD3;
float4instance_color :D3DCOLOR;
};
vsOutputGeometryInstancingVS(
invsInput input)
{
// construct the model matrix
float4x4modelMatrix =
  {
    input.model_matrix0,
    input.model_matrix1,
    input.model_matrix2,
    input.model_matrix3
  };
// transform input position and normal to world space with the instance model matrix
float4worldPosition = mul(input.position, modelMatrix);
float3worldNormal = mul(input.normal, modelMatrix);
// output position, normal, and color
output.position = mul(worldPosition, ViewProjectionMatrix);
output.normal = mul(worldNormal, ViewProjectionMatrix);
output.color = input.instance_color;
// output other vertex data
```

Fig70: Algorithme d'instanciation de géométrie par multi-streaming [54]

2.5 Pseudo-instanciation:

L'objectif de cette technique [56]est d'augmenter la performance en minimisant l'implication du CPU dans la tâche de rendu, et le transfert des données per-frame au GPU, en utilisant les attributs des sommets pour transporter les per-instance data, dont le transfert est optimisé par l'API.

Le rendu d'un grand nombre d'instances en utilisant la pile de matrices de vision pour les ModelViewProjectionMatrix des instances qui sont utilisées dans le programme de rendu (vertex shader ou géométrie shader), implique une charge de calcule sur le CPU pour calculer les matrices de transformation et les transmettre vers le GPU, alors l'application devient limitée par les capacités du CPU.

Une méthode alternative est de passer une fois pour toutes les instances une matrice ViewMatrix, et pour chaque instance une matrice de transformation au repère globale (world space) en utilisant une variable uniforme. Cette méthode nécessite plus de calcules au niveau du GPU (par ce que chaque vertex sera transformé en utilisant trois matrices au lieu d'une seule, mais même avec la charge supplémentaire pour le GPU, elle reste moins couteuse que le calcule par CPU des matrices de transformation et leurs transfère, rendant ainsi l'application moins dépendante du CPU.

Du même l'approche de pseudo-instanciation exploit l'efficacité de transfère et de traitement software et hardware des attributs persistants des sommets, comme la position la couleur, les coordonnées de textures… pour transférer les per-instance data au GPU.

Cette approche s'applique très bien aux instances avec un petit nombre d'attributs d'instance, par ce que les attributs des sommets doivent être suffisants pour représenter les sommets du modèle et au même temps les propriétés d'instances, en plus malgré que le transfert des données au GPU par attributs est plus efficace que l'utilisation des variables uniformes ou des matrices qui doivent être calculées pour chaque instance par CPU, le CPU reste impliqué dans la tache de rendu, et doit exécuter des appels API pour transférer le modèle plus les attributs de chaque instance.

3. Analyse de performance :

Les performances de l'implémentation de l'instanciation de géométrie dépendent de trois principaux facteurs, la taille de batch, la capacité du CPU d'alimenter le GPU en données, et la vitesse de transfert CPU, GPU [54].

3.1Taille des batchs :

Pour maximiser le profit en performance de l'instanciation de la géométrie il est recommandé que les batchs doivent vérifier :

- transmettre un nombre maximal d'instances par batch, alors toutes les instances qui ont le même contexte de rendu et le même modèle de géométrie doivent être regroupés une seul fois dans le même batch et transmit au GPU par un seul appel à l'API.

- chaque batch doit correspondreà un seul contexte de rendu GPU, alors le changement du contexte de rendu la création d'un nouveau batch. [55].

3.2 Implication du CPU dans la tache de rendu :

Le diagramme dans la figure 85[55] représente les résultats de tests effectués sur plusieurs combinaisons GPU, CPU, et qui confirment qu'en utilisant l'instanciation de la géométrie, les limites de la performance sont définis par la quantité d'information que le CPU peut traiter et envoyer au GPU, et le temps de transfère CPU, GPU, et non pas par la vitesse du GPU.

Sachant que depuis 2003(figure 86) l'évolution des GPU a été beaucoup plus rapide que les CPU, et cette différence de performance doit être maintenant beaucoup plus grande, nous pouvons déduire que pour exploiter les capacités du GPU, nous devons réduire au maximum l'implication du CPU dans le rendu.

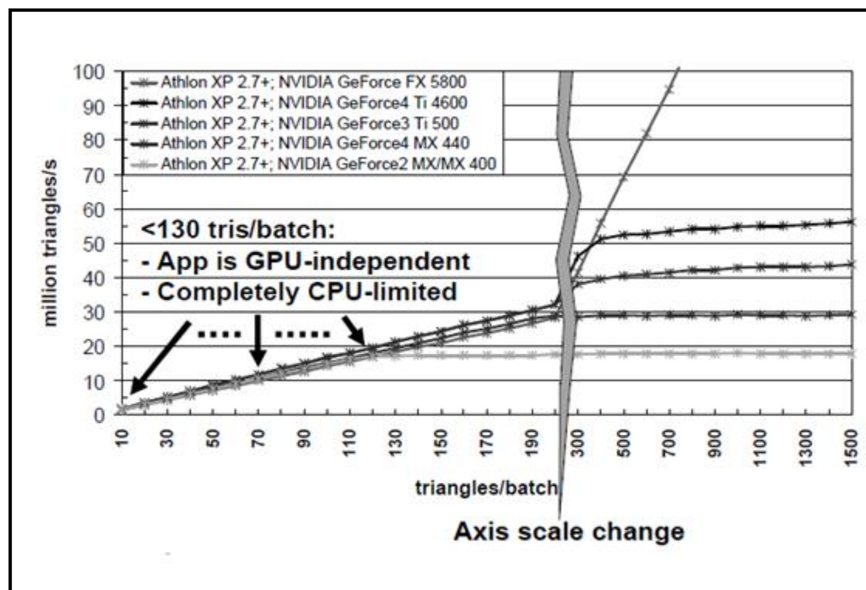


Fig71 : influence de l'implication du CPU dans la tache de rendu par instanciation de géométrie[55]

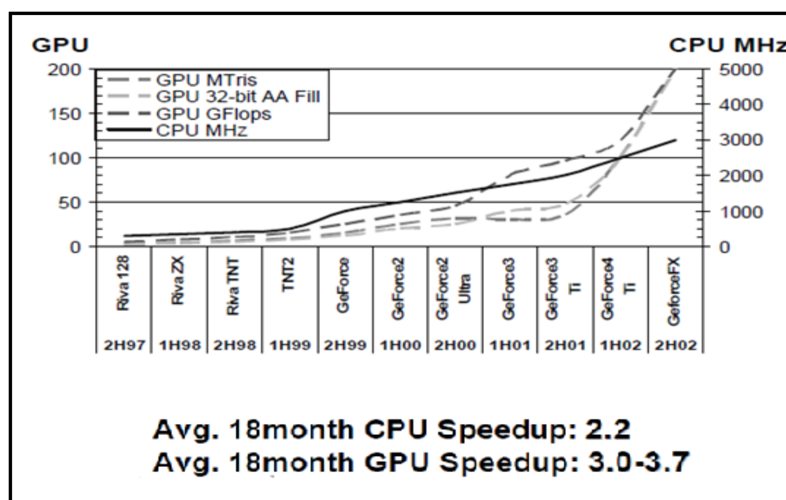


Fig72 : Comparaison de l'évolution de performances GPU et CPU [55]

4. Conclusion :

On peut conclure que les limitations de performances de l'instanciation de la géométrie sont au batching, qui est une tache bornée par les capacités du CPU, et la latence du transfert des données entre CPU et GPU, alors pour améliorer les performances de cette technique nous devons minimiser au maximum l'implication du CPU et la quantité d'information transférée vers le GPU lors de rendu, le cas idéal sera que le CPU n'exécute qu'un seul appelle de la fonction du dessin de l'API.

Le transfert d'information entre CPU et GPU, avec des quantités minimale d'informations peut être bénéfique, dans les cas ou des informations sont les mêmes par frame, leurs calcule par GPU impose une très grande charge sur ce dernier par ce qu'il les recalcule pour chaque sommets. Dans ce cas le calcule par CPU une seule fois et le transfert d'une valeur devient beaucoup plus efficace que son calcule par GPU.

4.1Utilisation du géométrie shader :

Avec l'introduction de la nouvelle étape programmable dans le pipeline graphique des cartes graphiques récentes il est devenu possible de générer des nouveaux sommets par le GPU, ce qui donne la possibilité d'implémenter des fonctions qui génère de la géométrie et contrôle son instanciation.

Nous proposons deux méthodes pour l'exploitation du géométrie shader pour l'instanciation de géométrie :

- L'utilisation d'un géométrie shader pour instancié un modèle géométrique, peut diminuer d'une façon très significative le nombre de bâches, per-frame, avec la possibilité de générer pour chaque sommet entrant au moins 256 nouveaux sommets (et plus en fonction du modèle de la carte graphique, et la taille des sommets générés).

Le géométrie shader offre aussi la possibilité de transférer jusqu'à 256 fois moins de batchs, si en utilise les sommets pour transférer les attributs d'une instance, avec un modèle de géométrie sauvegardé par des variables uniformes.

L'implémentation de la génération de géométrie par GPU permet de contrôler lors de rendu la complexité des primitives générées et instanciées permettant ainsi l'implémentation d'une génération dynamique par GPU de niveaux de détails.

Cette méthode s'adapte très bien aux modèles géométriques de taille réduite(le nombre de sommets du modèle de géométrie a instancié est petit), par ce que le modèle est totalement sauvegardé dans la mémoire graphique sous forme de variable uniforme, et les attributs des instances peuvent être aussi sauvegardés dans un VBO alloué dans la mémoire vidéo.

En cas ou les instances ne bougent pas, il n'y aura aucun transfert des données relatives à la géométrie entre CPU et GPU, tout le modèle géométrique et les attributs des instances sont sauvegardés au niveau de la mémoire graphique.

En cas ou les instances bougent les buffers doivent être mis ajoure, dans ce cas cette technique devient moins performante que le premier cas (utilisation des VBOs dynamiques) mais profit du transfère le plus effaçasse entre CPU GPU qui est le transfert des attributs persistants des sommets (Streaming).

Le nombre de batches dépend du nombre d'instances a dessinées et de la taille de la mémoire vidéo disponible pour les VBO, si la taille des données per-instance de tout les instances est inferieure a la taille de mémoire maximale offerte par la carte graphique aux VBOs, alors tout ces données peuvent être regroupées dans un seul batch offrant ainsi la meilleure performance en cas ou les instances ne bougent pas. Si la taille des données per-instance dépasse la capacité de la carte graphique alors l'utilisation de plusieurs batches devient impérative, alors il faut faire des misa jours des VBOs, diminuant ainsi la performance mais offrant un nombre d'instances illimités.

Cette méthode permet une exploitation maximale des capacités du GPU, en minimisant (même éliminant) le temps d'attente du CPU et transfère des données au GPU, mais elle est limité au modèles géométriques de petites tailles qui peuvent être sauvegardé au niveau de la mémoire disponible pour les variables uniformes.

- La deuxième méthode est d'utiliser la mémoire réservée au variable uniformes pour sauvegarder les attributs des instances et les VBOs pour les modèles de géométrie, par exemple on peut sauvegarder les données de transformation au repère world-space de chaque instance sous forme de variables uniformes qui seront utilisées par le géométrie shader pour générer plusieurs instances de chaque sommets de chaque primitive dans le VBO, produisant ainsi à la fin plusieurs instances de la géométrie contenu dans ce dernier.

Cette technique offre la meilleure performance possible dans le cas ou le nombre d'instances est leurs attributs suffisamment petits pour être contenu dans la mémoire disponible pour les variables uniformes dans la carte graphique (cette mémoire est plus de plus en plus grande avec le développement des cartes graphiques).

Le nombre d'instances possible est limité par la quantité d'information d'instances que peut contenir la mémoire réservée aux variables uniformes, par ce qu'il est impossible de mettre à jouer les variables uniformes pour le même frame (les variables uniformes peuvent être mis a jour seulement par frame).

L'avenage de cette méthode est la possibilité d'instancier des modèles géométriques très grands, mais a nombre de copies limité.

Une solution à la limitation de cette méthode est d'utiliser une texture look-up pour sauvegarder les données des instances, moins performante que l'utilisation des variables uniformes mais permet un nombre illimité d'instances, le batching ici devient une opération de mise à jour du contenu de la texture.

Cette solution atteint son maximum de performance dans les cas où la texture est suffisante pour contenir toutes les attributs de toutes les instances et que les instances ne bougent pas (pas de mise à jour de textures entre les frames), si le nombre d'instances est très grand on

peut exploiter le Multi-texturage, pour l'utilisation de plusieurs diminuant ainsi le temps de mise à jour inter-frame.

4.2.Adaptation au rendu des arbres :

Dans une forêt nous pouvons remarquer une très grande similarité entre les arbres, les branches, et les feuilles, cette similarité nous offre la possibilité de donner à l'observateur l'impression d'être entrain de regarder une forêt en dupliquant quelque modèles d'arbres, plusieurs fois, ces modèles offrent une impression de différence entre les arbres (dupliquer un seule modèle peut être remarqué par l'utilisateur), tendis les copies de ces modèles permettent de peupler la scène d'arbres.

Le rendu de toute les arbres formant une forêt peut implique la manipulation d'une énorme quantité de géométrie, et son transfert pour chaque frame vers le GPU est très couteaux, en plus le CPU va être occupé à ne rien faire que transférer des polygones au GPU, et tout ça serait limité par la capacité du CPU (qui aura très probablement d'autres taches à faire que l'alimentation du GPU en géométrie).

L'instanciation de la géométrie apparait alors comme une solution très convenable pour ces problèmes. Puisque les arbres sont des instances fixes et ne bougent pas d'un frame a l'autre, ils permettent une exploitation maximale des méthodes d'instanciation par géométrie shader, que nous avons présentée précédemment, en plus la propriété de similarité des constituants de l'arbre (entre branches et entre feuilles, et entres les arbres), que nous pouvons exploiter pour générer une description de sa géométrie qui sera la mieux adaptée à l'instanciation.

Chapitre 4:
Rendu des arbres avec instanciation de Géométrie par GPU

Chapitre 4

Rendu des arbres avec instanciation de Géométrie par GPU

I. interprétation géométrique adaptée GPU des modèles de représentation d'arbres :

Il existe plusieurs méthodes pour la génération d'arbres et de plantes, qui sont suffisantes pour créer des modèles très convaincants, réalistes et qui couvrent une très grande variété d'espèces, comme les modèles stochastiques, les fractales,… .

Dans ce travail nous allons nous concentrer sur l'interprétation de ces modèles en géométrie (description + primitives géométriques), qui soit le mieux adaptée au traitement par GPU. Nous allons prendre comme un exemple de modèle l'algorithme de Chaudy [11] (décrit précédemment), sur lequel nous allons appliquer notre interprétation. Nous avons choisi l'algorithme de Chaudy pour sa séparation très claire entre la génération du modèle (l'algorithme de ramification) et son interprétation en géométrie (les appels aux fonctions de création, destruction, et délassement des particules), on introduit la nouvelle interprétation en remplaçant les appels de gestion des particules par les méthodes de génération de géométrie qu'on expliquera par la suite.

Nous allons utiliser une fonction ramification qui va définir la manière dont la plante (l'arbre dans notre cas) va évoluer, le résultat retourné par cette fonction va être utilisé pour sélectionner et générer la primitive géométrique correspondante. Dans ce qui suit la fonction de ramification implémente l'algorithme de Chaudy, pour utiliser un autre modèle de génération de plantes, il faut modifier la fonction ramification, pour qu'elle implémente la méthode de génération utilisée par ce modèle.

Dans l'interprétation que nous proposant un arbre est représenté par un ensemble de cylindres et d'ellipsoïdes, obtenu par l'automate de Chaudy, mais au lieu de produire des trajectoires et des cercles finaux en produit des cylindres pour chaque trajectoire (un cylindre à chaque changement de direction et à chaque production de nouvelles branches, et en produit des ellipsoïdes à chaque nœud finale représentant une feuille.

1. Génération des primitives géométriques :

Le choix des primitives géométriques qui vont être utilisées comme les unités de bases pour construire les modèles d'arbres doit être guidé par la maximisation de l'exploitation des capacités du GPU, et vu que les GPU sont optimisés pour le traitement des triangles, ces primitives doivent être simples a construits à base de triangles, et elle doivent être aussi capable de produire les formes nécessaires pour la représentation d'arbres avec un nombre de primitives minimal et par conséquent un nombre de triangles optimal.

La construction des arbres à partir d'un nombre limité de variations primitives géométriques permet de maitriser la complexité de leur géométrie lors de génération des modèles, et offre aussi la possibilité de traitement identique des instances présentent dans la scène lors de rendu, ce traitement identique permet d'éviter l'utilisation d'algorithmes complexes qui doivent manipuler beaucoup de cas lors de rendu, ce qui est très bien approprié au shaders exécutables sur GPU (les shaders ne sont pas optimisés pour l'exécution de tâches très complexes).

Les primitives géométriques que nous allons utiliser sont les cylindres pour les branches, et les ellipsoïdes pour les feuilles, les résultats de l'interprétation seront sauvegardés pour être transmis, au GPU lors de rendu.

Ces résultats représentent des modèles d'arbres, qui seront chacun dupliqué par le GPU plusieurs fois, par instanciation de la géométrie. La représentation d'un modèle est séparée en deux parties, sommets et description,

En plus des donnés sauvegardé par l'étape de la génération de géométrie, pendant le rendu il est possible d'utiliser les caractéristiques des ellipsoïdes et des cylindres pour simplifier certain tâches, comme la génération des sommets par géométrie shader, et la sélection des indexes des sommets a représentés correspondant à un niveau de détails (simplification du maillage).

L'automate de Chaudy [11] commence par une particule principale et génères sa trajectoire puis il génère d'autres particules correspondants à des ramifications de l'arbre, pour chaque nouvelle particule il refait itérativement le même processus.

Pour utiliser cette automate pour la génération de primitives géométriques en utilise un cylindre comme point de départ, qui suit les changements de trajectoire de la particule principale de l'automate en fait les opérations suivantes :

- si la particule continue son déplacement dans la même direction, on ajoute le déplacement à la hauteur du cylindre actuel.
- si la particule change de direction, on génère un cylindre dont la variation de l'axe par rapport à l'axe du cylindre actuel correspond au changement de la direction et la hauteur est une valeur minimale (en peut utiliser même 0, et après on ajoute de la hauteur dès que la particule commence à se déplacer).
- si l'automate donne naissance à une nouvelle particule, on génère un nouveau cylindre avec l'axe correspondant à la direction dans laquelle la particule a été créée et de hauteur minimale.
- si une particule meurt, on ferme le cylindre correspondant (génère une cape).
- si l'automate produit une feuille (une particule finale), on crée un ellipsoïde avec une taille et un axe aléatoire (suivant des probabilités correspondantes à l'espèce).

Pour améliorer cette représentation, on ajoute au niveau des branchements un mécanisme de transition souple. A chaque branchement, on ne produit pas directement le cylindre correspondant à la trajectoire, on produit plusieurs cylindres transitoires qui représente un changement moins brusque d'axe avec une hauteur réduite jusqu'à l'obtention d'un cylindre dont l'axe correspond à la trajectoire produite par l'automate.

En plus, si le changement de l'axe est petit, on peut utiliser une méthode qui donne une transition souple sans ajouter de nouveaux cylindres, en utilisant les coordonnées des sommets du cercle supérieur du cylindre précédent comme coordonnées du cercle inferieurs du nouveau cylindre.

1.1Représentation des modèles :

Un modèle d'arbre est séparé en deux ensembles de primitives géométriques : les branches (un ensemble de cylindres), et les feuilles (un ensemble d'ellipsoïdes),dont chacun possède deux types de données : les données communes qui représentent des propriétés commune à tous les éléments de l'ensemble, et les données spécifiques à chaque élément qui comme les positions des centres des primitives, les dimensions…,et les données des sommets constituants ces éléments qui sont les coordonnées dans le repère local du modèle et les normales.

1.2 Représentation de branches :

Une branche est représentée par un ou plusieurs cylindres (plusieurs dans le cas ou on utilise plusieurs cylindres pour la transition aux niveaux des branchements entre les branches), un cylindre est décrit par:

Données communes :

- le nombre de sommets par cylindres : ce nombre est le même pour tous les cylindre dans le modèle.
- le nombre de sommets à éliminer pour la transition entre niveaux de détails, qui définit le nombre de sommets à « sauter » pour trouver le prochain sommet en cas de simplification de maillage (le mécanisme sera expliqué dans la partie rendu).

Données spécifiques à chaque cylindre :

- Le centre : les coordonnées x, y, et z d'un point qui contrôle la position à laquelle le cylindre sera positionnée permet de déterminer l'axe du cylindre avec la propriété axe.
- Le grand rayon et le petit rayon : on va utiliser pour la construction de chaque cylindre deux cercles, un cercle inferieur qui aura comme rayon le grand rayon et un cercle inférieur qui aura la valeur de petit rayon comme rayon, Ce qui permet de créer un effet de diminution du rayon des branches.
- La hauteur : c'est la hauteur du cylindre.
- Axe : un vecteur qui représente la direction de la branche para port à l'origine du repère local du modèle.
- Nombre de feuilles : le nombre d'ellipsoïdes générés pour une certaine direction, si on génère des nouveaux cylindres sans changements de directions, on leur affecte la même valeur du nombre de feuilles (la valeur finale obtenue au prochain changement de direction). Ce paramètre est incrémenté de un à chaque génération d'une feuille, et sera utilisé dans le rendu pour produire l'information d'intensité qui sera utilisé pour la génération du niveau de détail le plus grossier, et pour la génération du masque d'ombrage qui va être appliqué lors de rendu pour la production des ombres (son utilisation sera expliqué dans la partie rendu).

1.3 Représentation des feuilles :

Chaque feuille sera représentée par un ellipsoïde décrit par les paramètres suivants:
Données communes :

- Nombre de sommets par feuille : le nombre de sommets qui forment une ellipsoïde (représentant une feuille).
- Nombre de sommets a éliminé pour la simplification du maillage

Données spécifique pour chaque feuille :

- Centre : vecteur de trois éléments qui contient les coordonnées x, y, et z de la position du centre de l'ellipsoïde dans le modèle, contrôlant ainsi la position de la feuille dans le modèle. Les instances de feuilles (ellipsoïdes) seront générés pendant le rendu en application une translation des coordonnées des sommets communs par x, y, et z.
- Paramètre de rotation : un vecteur de valeurs dont chacune indique la rotation selon un axe (ox et oz). Dans l'implémentation nous avons remplacé ce paramètre par un vecteur de quatre éléments qui contient le sinus et cosinus des angles de rotations selon chaque axe, qui seront utilisés pendant le rendu pour construire une matrice de rotation, qui serre a calculé les coordonnées des sommets de chaque ellipsoïde, à partir des coordonnées des sommets communs, cette méthode permet de gagner du temps de calcul dans le géométrie shader (pré-calcul de cosinus et sinus des angles de rotation), et d'optimiser le stockage en sauvegardant un vecteur de quatre éléments, pour chaque instance au lieu de sauvegarder toute une matrice de rotation dans la méthode classique d'instanciation de géométrie.
- La normale : le vecteur normal de chaque instance d'ellipsoïde calculé lors de l'interprétation du modèle et sauvegardé pour être utilisé lors du rendu.

2. Implémentation :

Pour l'interprétation en géométrie des modèles de génération d'arbre, nous avons implémenté un ensemble de fonctions qui génèrent pour chaque type de ramification un ensemble correspondant de primitives géométriques ou les modifications correspondantes sur des primitives déjà existantes.

Dans ces fonctions on manipule deux classes qui encapsule les informations et les opérations applicables sur ces deux unités de base de construction de nos modèles et qui représentent les primitives géométriques formants consécutivement les branches et les feuilles d'arbres.
Plus une classe modèle, qui comporte un ensemble de cylindres, qui représente les branches du modèle d'arbre, et un ensemble d'ellipsoïde qui représente les feuilles, ainsi que les données communes aux cylindres et ellipsoïdes du modèle. Cette classe comporte aussi la méthode de ramification qui calcule le type de la prochaine ramification, et la méthode grew ( ), qui exécute à chaque fois invoqué un appel à la fonction ramification, et utilise le résultat pour déterminer la méthode d'interprétation à invoquer.

La ramification va se passer au niveau des bourgeons qui sont représentés par un ensemble d'indexes des branches, qui contiennent des bourgeons encore vivants, et qui seront les points qu'infectera la prochaine ramification.

Cette ensemble est initialisé avec le premier cylindre (le début du tronc de l'arbre), et sera géré au cours de l'interprétation par deux méthodes :

1. addbrg(cylindre c) : cette méthode ajoute un nouveau bourgeon a la liste de bourgeons actifs, si le nombre maximal des bourgeons actifs n'est pas encore atteint.
2. dellbrg(index) : permet de supprimer un bourgon de la liste des bourgeons actifs, cette methode est invoquée après qu'un bourgeon subit une ramifiction (croissance ou mort).

2.1. Génération de la géometrie :

Pour la génération de la géometrie nous avons implementé les méthodes :

1.**Génération de primitives :** au niveau de chaque classe ellipsoide et cylindre nous avons implementer des méthodes de génération de coordonnées des sommets qui forment la primitive de base au point (0,0,0), qui seront manipulés ensuite pour les adapté au résultats de la ramification, ainsi que la normale et les coordonnées de texture.

-Cylindres :
Pour générer les coordonnées des sommets V1, V2... on utilise l'équation paramétrique suivante avec le paramètre Theta : un angle qui varie entre 0 et 2PI :

X=R*COS(Theta)
Y=Y
Z=R*SIN(Theta)

La variation de l'angle theta est contrôlée par un incrément inc qui dépend du nombre d'itérations qu'on veut avoir, dont chacune produit deux vertex, alors on a :

inc= 360/n avec n : le nombre d'itérations ciblé (nombre de vertex ciblé /2).
Pour chaque itérations on calcul les coordonnés de deux vertex:

X=R*COS(Theta) Y=0 Z=R*SIN(Theta) et	X=R*COS(Theta) Y=0 Z=R*SIN(Theta)

Avec H: hauteur du cylindre et R: rayon du cylindre.

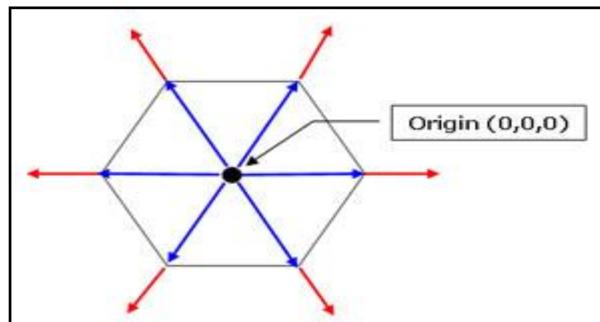


Fig73 : Construction de cylindres

On remarque que les coordonnées des normales sont les mêmes que celle des vertex sauf pour la composante z qui est =0, alors : pour les deux vertex générées dans chaque itération la normale est : n= normalize (x,0,z).

- Ellipsoides :
Un ellipsoide est toujours generé au point c (cx, cy, cz), dans le plan xoy, avec un petit rayon p« Pr » pris sur l'axe oy, et grand rayon « Gr » pris sur l'axe ox, puis il est translaté pour être ramené au point où il sera connecté à un cylindre (branche), ainsi que deux rotations selon les deux axes ox, et oz, par un angle déterminé par la méthode de génération (dans notre cas nous avons appliqué des propabilités de rotation que nous avons ajouté à l'algorithme de Chaudy).

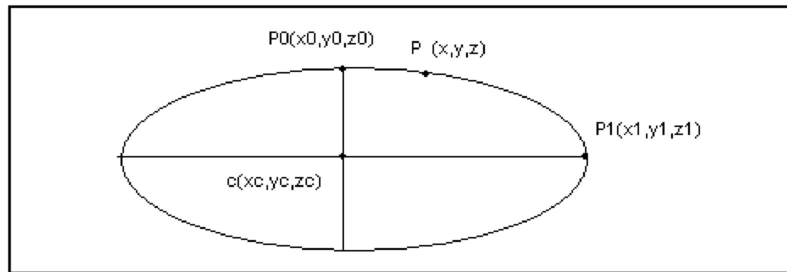


Fig74 : Construction d'ellipsoïde

Pour calculer les coordonnées du point p, on utilise les coordonnées du point P0 et P1 comme des point de contrôles, alors les coordonnées du point p seront influencés par les coordonnés de P1 et P0 selon les distances respectives par rapport à ces deux points :

Avec :

x0=xc		x1=xc+Gr		x=x0+dx
y0=yc+Pr		y0=yc		y=y0+dy
z0=zc	et	z0=zc	en parton de P0 en a pour chaque point p(x,y,z):	z=z0+dz

Pour calculer en chaque point p les valeurs dx, dy, dz, on définit n le nombre de points qu'on veut générer, alors le déplacement d'un point vers l'autre selon l'axe ox est dx= x1-x0/n.
Selon oz dz=0, par ce que nous allons dessiner la surface sur le plan xoy.

Pour l'axe oy, l'utilisation d'un déplacement uniforme dy, va résulter en une forme rectangulaire et non pas ellipsoïdale, alors on doit utiliser des déplacements différents, pour chaque ensemble de point, le nombre de ces ensemble doit être au minimum égale à trois (de un point ou plus chacun) pour donner la forme ellipsoïdale, avec dy1 pour le premier ensemble, dy2, est dy3, pour le deuxième et le troisième ensemble consécutivement, vérifiant

dy1<dy2<dy3 et
dy1*n1+dy2*n2+dy3*n 3=Pr avec ni le nombre de points dans l'ensemble i.

Pour l'implémentation nous avons utilisé trois ensembles de trois points chacun (un totale de neuf points par un quart d'ellipsoïde), alors on utilise trois valeurs pour dy :

dy1= 0.2*Pr/3
dy2=0.3*Pr/3
dy3=0.5*Pr/3

Si on veut générer tous les points pi entre P0 et P1, on procède itérativement d'i=0 jusqu'à i=n, en incrémentant à chaque fois deux paramètres incx, incy, initialisés à 0 de dx, et dyi correspondant à l'ensemble actuel du point en cours.

Cette méthode permet de calculer neuf points sur un quart d'ellipsoïde, pour générer les trois autres quart, on déduit à partir de chacun les coordonnées des trois autres points sur les trois autres cas pour chaque itération par :

P2 : x2=x et y2=-y
P3 : x3=-x et y3=-y
P4 : x4=-x et y4=y

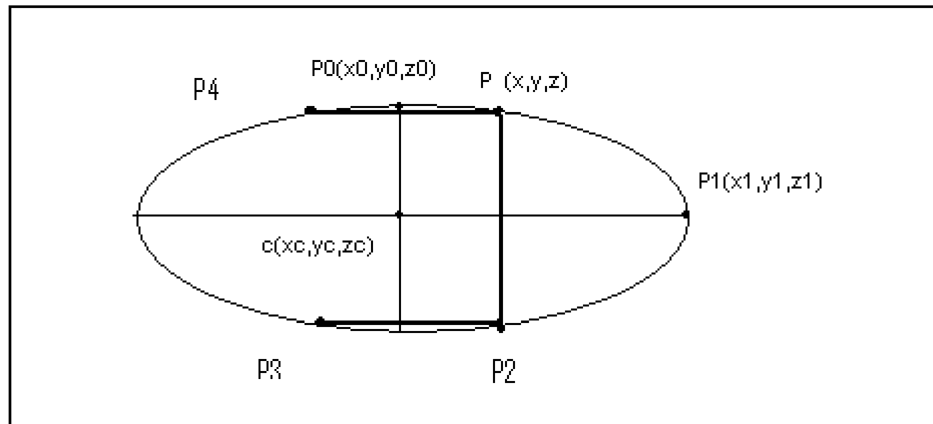


Fig75 : Extraction de quatre sommets de l'ellipsoïde à partir de chaque point calculé

Par ce qu'on a généré la surface sur le plan xoy grâce la normale et un vecteur unitaire sur l'axe oz, alors la normale initiale est n (0, 0,-1), qui sera modifiée aussi par les mêmes transformations de rotation que subiront les sommets de la surface.

Il faut s'assuré au moment de l'interprétation que les angles rotation selon ox ne sont pas supérieur à pi (pour ne pas avoir de feuilles d'arbres qui sont orienté vers la terre).

2.2. Implémentation des méthodes d'interprétation :

Pour manipuler les primitives géométriques nous avons implémenté les méthodes :

- translate (dx, dy, dz) : une méthode membre de chaque classe ellipsoïde et cylindre qui permet de translater la primitive des distances dx, dy et dz, en translatant, tous les sommets.
- rotate (ang, a) : implémentée aussi dans les deux classes, et permet de faire une rotation d'un angle ang selon un axe a (à un vecteur de trois valeurs x, y, et z), cette méthode affecte aussi la normal.
- connecte : cette méthode est implémentée de façon différente pour chaque primitive.

1.Ellipsoïde : connect (x,y,z) pour la classe ellipsoïde la méthode connect fait simplement une translation de tous les sommets de telle sorte que le point de connexion de coordonnées xcon,ycon et zcon coïncide avec les valeurs x,y, et z.

2.Cylindre : avec la classe cylindre la méthode est un peu plus complexe par ce qu'en plus de la translation au point de connexion de l'autre cylindre, elle modifie aussi la forme des sommets pour donner une transition plus souple d'un cylindre a l'autre (entre branches), cette modification consiste à remplacer les sommets formant le cercle inferieur du cylindre a connecté par les sommets du cercle supérieur du cylindre sur lequel il se connecte.

- stritch (dh,dr, min) : méthode implémentée dans la class cylindre, qui permet d'ajouter à la hauteur du cylindre la valeur dh, avec un facteur de diminution du rayon dr, avec une valeur minimale du rayon égale a min, si la valeur du rayon est inférieur ou égale à min alors la diminution du rayon ne s'appliquera pas.

Cette méthode est invoquée pour appliquer le type de ramification croissance d'une branche, en translatant les sommets du cercle supérieur d'un cylindre de dx, dy et dz dans la direction qui préserve sa forme.

Pour préserver la forme du cylindre les relations des déplacements dx, dy et dz dans le déplacement

dh totale, doivent être les mêmes que les relations de participation de xb-xc, yb-yc, et zb-zc dans la distance entre le point du centre c et le point du branchement b du cylindre alors :

dx/dh= (xb-xc)/h
dy/dh=(yb-yc)/h
dz/dh=(zb-zc)/halors :

dx=(xb-xc)*dh/h
dy=(yb-yc)*dh/h
dz=(zb-zc)*dh/h

Où h est l'ancienne hauteur du cylindre
Pour appliquer la diminution du rayon, on déplace les sommets du cercle supérieur du cylindre vers le centre de façon à ce que le rayon du cercle soit diminué de dr, en préservant les rapports des contributions des anciennes distances entre les sommets et le centre, en s'approchant du centre avec les valeurs drx, dry et drz alors, on pour chaque sommet:

drx=(x-xc)*dr/pr
dry= (y-yc)*dr/pr
drz=(z-zc)*dr/pr

Avec pr l'ancien rayon du cercle supérieur du cylindre, x, y, et z les coordonnées de l'ancien sommet.
Les nouvelles coordonnées de chaque nouveau sommet sont :
x=x+drx
y=y+dry
z=z+drz

- addfeuille (ellipsoide) : membre de la classe cylindre qui permet d'ajouter une feuille à une branche, en connectant un ellipsoïde sur un cylindre.

-deforme (index, ang, dh, a) : membre de la classe modèle, invoquée pour exécuter une ramification de type déformation de branche sur le bourgeon ayant l'index « index », en créant trois nouveaux cylindres (nous avons choisie trois comme un meilleur compromis entre le nombre de cylindre et la qualité visuelle de la zone déformée)sur une distance dh dont l'angle entre le cylindre courant et le dernier est égale a ang, sur l'axe a, et les deux autres sont des cylindres intermédiaires.

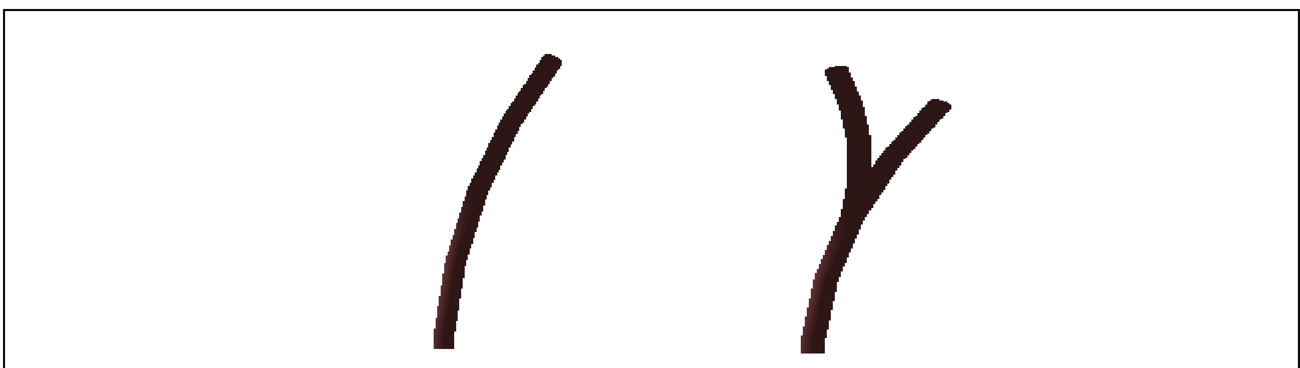


Fig76 : Résultats de la méthode déforme

- grew () : méthode membre de la classe modèle, quand elle est invoquée, elle évalue les ramifications de tous les bourgeons actifs, et applique les générations et modification de géométrie correspondantes, puis elle mit à jour l'ensemble des bourgeons actifs (supprime les bourgeons morts, ou qui ont finis leur développement, et ajoute les nouveaux bourgeons).

La création d'un arbre se fait à travers plusieurs invocations consécutives à cette méthode, jusqu'à

ce que tous les bourgeons soient mort, ou un maximum de ramification est atteint, ou est interrompu par l'utilisateur.

3. Sauvegarde des modèles pour le rendu :

Après l'interprétation des modèles en géométrie, nous devons sauvegarder les données nécessaires pour les utiliser lors du rendu, pour la sauvegarde nous avons séparé les branches et les feuilles, ceci est due aux différences entre les données qu'on sauvegarde pour chaque ensemble.

Comme on a expliqué préalablement, pour décrire un arbre, on utilise deux types de données, données communes à toutes les primitives géométriques, et des données spécifiques à chaque cylindre, cette séparation sera exploitée lors de rendu par GPU, ou les données communes seront passé au programme de rendu comme uniformes et les données spécifiques seront passées comme des attributs de sommets.

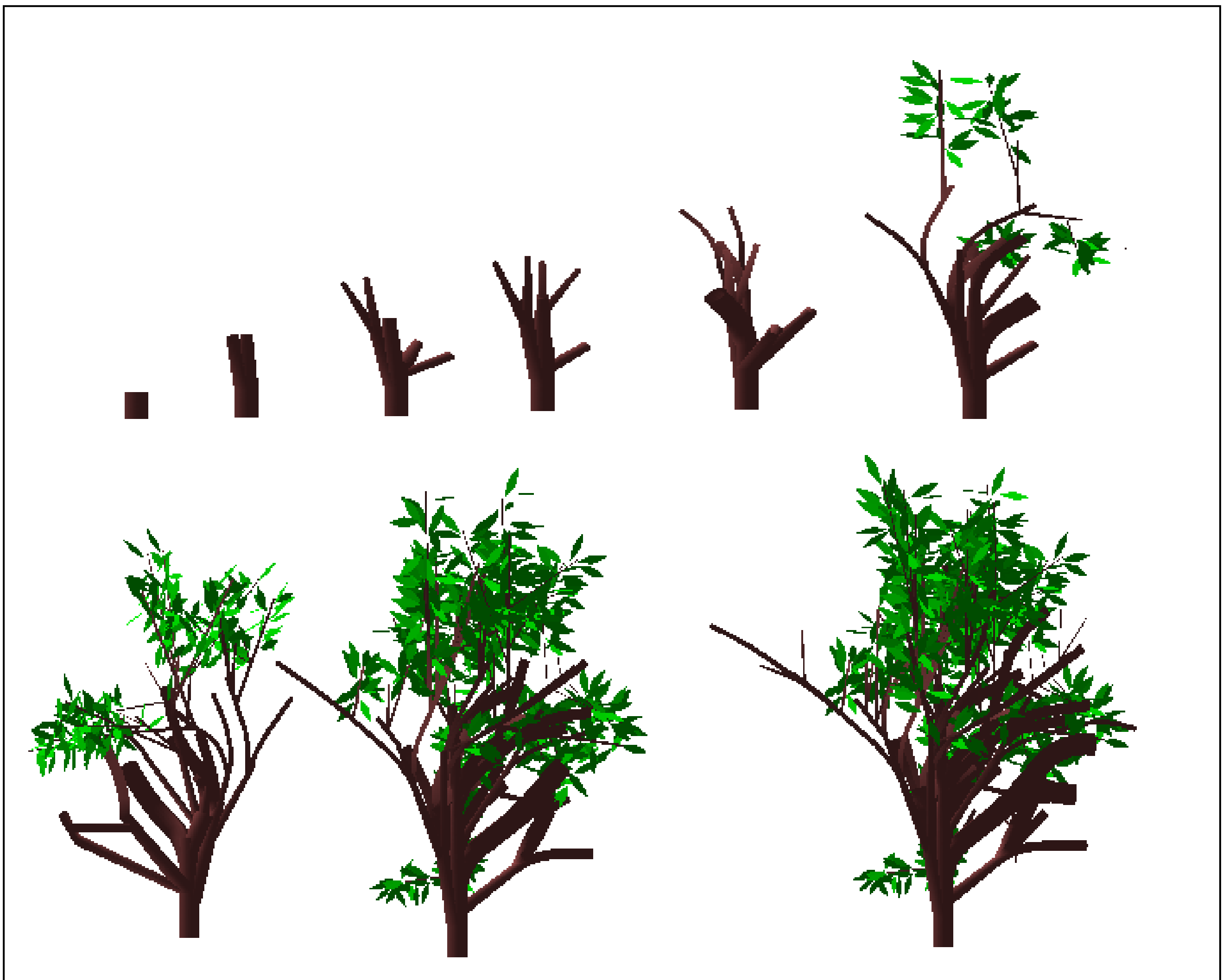


Fig77 : Résultats de plusieurs invocations de la méthode grew

3.1 Les branches :

On sauvegarde comme données communes, le nombre de sommets par cylindre, et les coordonnées de texture, dans notre implémentation, nous avons sauvegardés toutes les coordonnées des sommets et les normales pré-calculées lors de l'interprétation en géométrie du modèle, mais au lieu de sauvegarder tous les sommets, formant un cylindre, on sauvegarde pour chaque deux cylindres les

sommets de trois cercles, le cercle inferieur du premier cylindre avec les normales des surfaces du premier cylindre, les sommets du cercle supérieur du premier cylindre avec les normales situées entre les normales aux surfaces du premier cylindre et le deuxième, et le troisième cercle est le cercle supérieur du deuxième cylindre avec les normales aux surfaces du deuxième cylindre.

Cette technique permet d'optimiser l'espace de sauvegarde, et le temps de calcul par une diminution importante de nombre de sommets, et améliorer aussi la qualité visuelle par l'interpolation des normales.

3.2 Les feuilles :

Pour les feuilles deux méthodes de représentation au niveau du GPU sont possibles :
-les ellipsoïdes qui représentants les feuilles sont complètement indépendants, alors il suffit de sauvegarder uniquement les coordonnées des sommets formant un modèle générique d'ellipsoïde, qui sera instancié au niveau du GPU par géométrie shader, et manipulé pour chaque instance pour correspondre à une feuille spécifique, le nombre de sommets par ellipsoïde est unique pour toutes les instances, la normale peut être sauvegardé une fois et puis modifié pour chaque instance (plus de calcul et moins de mémoire) ou bien sauvegardée pour chaque instance (plus de mémoire et moins de calcul).

Cette méthode dépend des capacités du GPU à exécuter le géométrie shader et du nombre de sommets qu'il peut générer (il doit générer un nombre d'ellipsoïdes égale aux nombres de feuilles du modèle multiplié par le nombre de ses instances).

Le nombre de sommets que le GPU peut créer est limité alors le dessin de toute les feuilles nécessite plusieurs appels API du dessin sur le VBO contenant les feuilles du modèle, en les dupliquant à chaque fois le maximum possible par le géométrie shader, Le nombre d'appels API va dépendre du nombre maximale de sommets que peut générer le géométrie shader, alors les performances augmentent avec l'augmentation du nombre de sommets possible à créer au niveau du géométrie shader .

La génération par géométrie shader des ellipsoïdes étant complètement parallèle (chaque ellipsoïde est indépendant des autres), la vitesse du traitement va dépendre aussi de la capacité de traitement et du nombre d'unités de calcul présentes sur le GPU (on ne parle pas de géométrie processeurs ici en considérant l'architecture unifié, qui est implémentée sur tous les GPU récents.).

Les GPUs les plus récents exécutent la géométrie shaders avec plus d'efficacité, et ont un nombre de sommets possible à générer plus grand, alors ils sont mieux adaptés pour exécuter cette méthode, et la tendance de développement des GPUs favorise l'utilisation de ce type de méthodes qui génèrent des éléments de géométrie d'une façon parallèle avec des traitements minimaux par élément (la géométrie shaders sont de plus en plus efficaces).

- la deuxième approche consiste à la même méthode utilisée pour les branches, en sauvegardant les coordonnées et les normales de tous les sommets (dans des VBOs) et puis utiliser le géométrie shader pour les dupliqués pour chaque instance du modèle.
Pour chaque instance on sauvegarde le centre qui indique l'emplacement de l'ellipsoïde dans l'arbre.

II. Rendu d'arbres en utilisant l'instanciation de géométrie par GPU :

Nous avons utilisé pour le rendu un programme composé de shaders qui comprend un géométrie shader capable de dupliquer les modèles résultants de l'étape de l'interprétation en géométrie, en utilisant des données d'instances sauvegardées au niveau de la mémoire des constantes d'attributs (variables uniformes), ceci est faisable par ce que les cartes graphiques récentes comportent une quantité de mémoire importante qui peut contenir ces données, et aussi la quantité de ces données n'est pas trop grande en considérant les instances comme des arbres, les données des instances d'arbres à rendre à un instant donné ne vont pas dépasser la capacité de la mémoire.

1. L'instanciation de géométrie :

Comme nous avons expliqués préalablement (chapitre instanciation de géométrie), les anciennes méthodes d'instanciation de géométrie qui n'utilisaient pas la capacité de génération des sommets par GPU dépendaient des capacités du CPU, ce qui résulte une mauvaise exploitation du GPU, nous proposons dans ce qui suit une méthode qui exploite cette capacité de trois façons :
La génération de nouvelles instances sans intervention du CPU, éliminant ainsi le problème d'attente de GPU et exploitant ces capacités de calcul parallèle.
La possibilité de ne pas générer des sommets de la géométrie shader, en appliquant une génération d'instances conditionnelle en se basant sur un mécanisme d'élimination d'instances hors de la pyramide de vision.
La possibilité de choisir la manière dont les sommets vont être générés, en fonction du niveau de détails des instances, en appliquant aussi un procédé conditionnel de génération d'instances.
L'utilisation d'une description identique pour les primitives géométriques de construction des branches et feuilles (cylindre et ellipsoïdes), qui sont décrites par un petit rayon, un grand rayon, un centre, un axe, et un point de branchement, et un ordre de sommets permettant de les construire en prenant ces sommets du premier au dernier (la méthode de construction sera détaillée après), permet de gérer la géométrie sur GPU avec un seul programme qui n'a pas besoin d'être complexe pour gérer les deux primitives de manières différentes.

1.1. Gestion des bâches (lots de géométrie) :

Le nombre de sommets que le GPU peut générer par appel à l'API du dessin est limité, ce qui impose l'utilisation d'un système de batching, pour dessiner toutes les instances qu'on veut produire.
Un ensemble d'arbres, est représenté par un ensemble de coordonnées (x, y, z) des positions d'arbres, ces coordonnées sont passées au GPU sous forme d'un tableau uniforme de float. Le modèle à partir duquel est instancié un arbre correspondant à une position est déduit de la position de ces coordonnées dans le tableau de positions d'arbres, par exemple les n premières positions correspondent aux instances du premier modèle et les positions de n+1 jusqu'à 2*n correspondent au deuxième et ainsi de suite.
L'incorporation de la déduction du modèle a instancié pour chaque position permet d'économiser de la mémoire uniforme sur la carte graphique, et simplifie le procédé de détermination du modèle approprié pour chaque instance.
Le nombre d'instances par modèle n doit être choisi de façon a exploité au maximum les capacités du GPU, en fonction du nombre de nouveau sommets possibles à générer par le géométrie shader, alors pour chaque modèle on doit générer le maximum d'instances (arbres) possibles que le nombre maximal de sommets permet.
La méthode qu'on a utilisé pour construire les batchs consiste à faire correspondre trois batchs à

chaque modèle, chacun est appliqué par un appel API du dessin, le premier sur les parties des VBOs contenant les attributs des sommets du premier modèle, la gestion des indexes des sommets ne sera pas couteuse par ce qu'on n'utilise pas un VBO qui doit être mis à jour (buffer dynamique moins performant) pour la simplification du maillage, à la place, on utilise un procédé de saut de sommets, qui peut être exécuté par le changement du Strid au niveau du deuxième appel API, enfin pour le niveau de détails le plus grossier, on utilise un appel API de dessin sur les VBOs contenant la description du modèle.
Cette méthode n'impose aucune mise à jour des VBOs (transfert des données CPU, GPU) par ce que tous les VBOs sont remplis une fois pour tous au démarrage de l'application, et on n'a pas besoin de VBOs d'indexes (le stride sera suffisant), ce qui permet l'exploitation la plus efficace des VBOs (STATIC DRAW), le seul transfert de données dont on aura besoin est la mise à jour des informations d'instances (variables uniformes), dues au opération du cliping d'instances et détermination du niveau de détails. Cette mise à jour est très rapide (le nombre d'arbres est négligeable par rapport au nombre des sommets).
Dans le cas de notre méthode d'instanciation de géométrie les tests montrent qu'on peut générer 12 arbres par géométrie shader avec un géométrie shader capable de générer un maximum de 255 sommets (minimum sur les cartes graphiques supportant le géométrie shader), ce nombre est le résultat de la requête envoyée au GPU pour récupérer le maximum de sommets possible à générer par géométrie shader, mais en réalité ce nombre ne peut être générer que pour des sommets avec un minimum d'attributs (deux vecteurs de quatre composantes), ce nombre baisse en fonction du nombres d'attributs des sommets.

1.2. Les données d'instances :

Les données d'instances sont transmises au GPU sous forme de variables uniformes, elles seront alors déterminés une fois pour toute et transmîtes au GPU par frame, (les variables uniformes ne peuvent pas être mise à jour pour la même image).
Les données d'instance dans notre implémentation sont les positions des arbres qui seront transmises au GPU sous forme d'un tableau uniforme de coordonnées x, y, et z comme on a indiqué avant, en plus des données relatives aux opérations de clipping et niveau de détails qui seront transférées au GPU dans un autre tableau uniforme qui va contenir une description de chaque instance (position).
Le géométrie shader va prendre chaque position dans le tableau des positions et décider si l'instance va être dessinée ou non, et la façon avec laquelle elle être dessiné, en fonction des valeurs dans le deuxième tableau (le tableau de description).

2. Procédé de rendu :

L'application du rendu récupère les positions des sommets dans leurs repères locaux, les normales et la définition des modèles à partir des fichiers sauvegardés dans l'étape de l'interprétation, et le charge dans des VBOs sur la mémoire vidéo, et regroupe les positions et les descriptions des instances dans des tableaux, et pour les sommets de chaque modèle elle ajoute des attributs permettant de gérer l'instanciation et l'illumination des modèles.
L'API indique au programme de rendu que les sommets sont manipulés sous forme de TRIANGL_STRIPS, et les entrées et sorties du géométrie shader sont de TRIANG_STRIPS.

2.1. Initialisations :

En plus des attributs position et normal obtenus à partir des fichiers de chaque modèle l'application génère les attributs suivants :

2.1.1. Attribut Id :
Un vecteur de trois valeurs qui permettent de localiser le sommet par rapport a la primitive, le niveau de détails auquel il appartient, et le type de primitive (une branche ou un feuille) ; ces trois composantes sont utilisée comme suit :

- La première composante est un numéro qui peut être un ou zéro, il est généré alternativement, pour les sommets de chaque cylindre et ellipsoïde, par exemple pour les sommets du premier cylindre il est égale à zéro, et pour les sommets du deuxième il est égal à un, pour le troisième zéro et ainsi de suite.

  L'utilisation de TRIANGL_STRIPS comme primitive d'entré impose l'utilisation d'un appelle API par primitive de construction (cylindre ou ellipsoïde), qui seront traités chacun comme un TRIANGLE_STRIP séparé, alors on retombe dans le problème d'occupation du CPU à ne rien faire que alimenter le GPU en géométrie, et tout l'application sera cadencé à la capacité du CPU à alimenter le GPU, qui aura beaucoup de temps de chômage (problème discuté au chapitre instanciation de la géométrie).
  Le numéro de la primitive permet d'éviter ce problème, en fournissant un moyen de contrôle de dessin des sommets au niveau de la géométrie shader.
  Du côté de l'API tout le modèle sera considéré comme un seul TRIANG_STRIP, alors un seul appel API est suffisant pour le dessiner, et au niveau du géométrie shader, on peut détecter la fin de chaque primitive en comparant les numéros des sommets en entrés, si ils ne sont pas le mêmes alors en ne dessine pas le triangle qui les relient. Ce mécanisme nécessite un test supplémentaire au niveau du géométrie shader qui est justifié par rapport à l'élimination de la dépendance du CPU, et la quantité immense d'appels APIs (plus de miles appelles par modèle), ou la manipulation des indexes très couteuse en mémoire.
- La deuxième composante est utilisé pour la simplification de maillage, elle peut avoir plusieurs valeurs relativement au nombre de niveaux de détails par simplification de maillage voulu.

Simplification du maillage avec l'illimitation de surfaces :
Cette simplification se fait par diminution du nombre de sommets constituant une primitive géométrique, pour chaque instance en fonction de leurs distances par rapport au point de vue.
La figure Fig 92 représente la construction des faces d'un cylindre à partir de sommets indexés, ou chaque surface est construite à partir de quatre sommets, dont les deux premiers sont partagés avec la face précédente et les deux autres sont partagés avec la face suivante.
La fusion d'un ensemble de faces en une seule, se fait par élimination des sommets qui génèrent les faces situées entre la première et la dernière, et ne considérer que les premiers sommets de la première face et les derniers sommets de la dernière face.
La représentation initiale du modèle d'arbres, est considérée comme le niveau de détail le plus fin, pour générer d'autre niveau moins fin, on applique la fusion des surfaces d'une façon régulière (le même nombre de surfaces est fusionné) au niveau des cylindres constituant le modèle d'arbre, pour conserver la forme générale de ces derniers.
Pour une fusion de face régulière sur un cylindre, on utilise la méthode de sélection des sommets suivante :

- supposons que le nombre de faces qu'on veut fusionner est n, alors pour chaque n face, les sommets qui ne seront pas éliminés sont les deux premiers sommets de la face au début et les deux derniers de la dernière face de chaque groupe de n faces, dont les indexes vérifient la condition : le reste de la division de l'index du sommet est égal à un ou zéro, alors pour toute primitive un sommet est éliminé si le reste de division de son index par le nombre de faces à fusionner est différent de un ou de zéro.
- pour conserver la forme de la primitive nous devons assurer deux conditions :

- Les sommets de début doivent être les mêmes que ceux de la fin : cette condition est assurée dans notre représentation géométrique, mais il faut s'assurer que les indexes des sommets de début de la primitive et de la fin passeront le teste d'élimination de sommets.
- Dans le cas d'un cylindre, il faut s'assurer d'avoir un nombre de faces correspondant à un cylindre, ou au moins quatre faces pour un cube pour le niveau de détails le plus grossier.

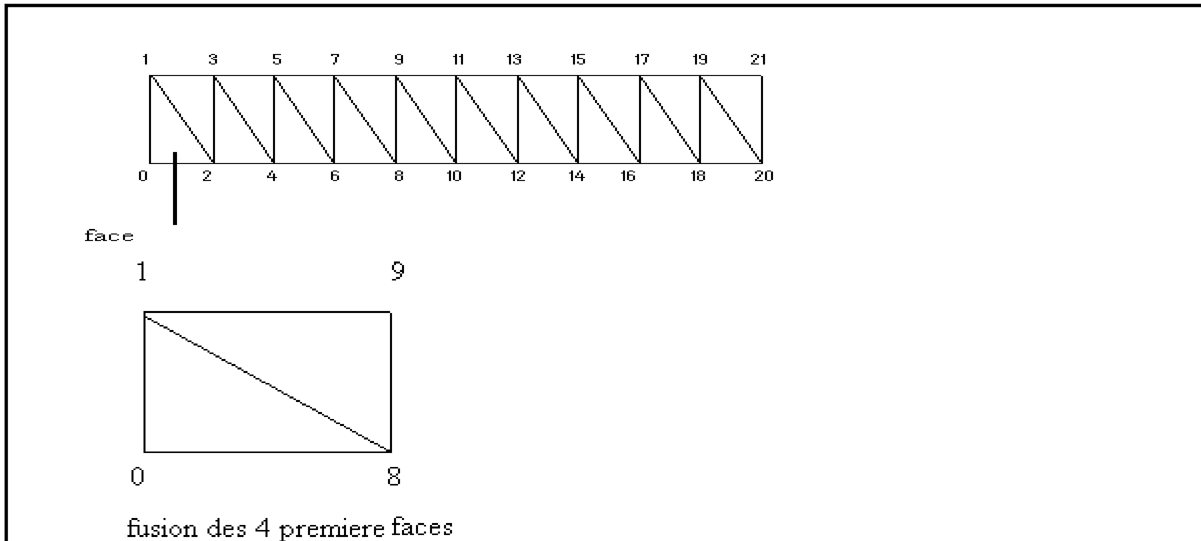


Fig78 : Fusion des surfaces

Au niveau de l'application une simplification du maillage correspond à un deuxième appel API du dessin sur les sommets d'un modèle en changent de strid (ne provoque pas la mise à jour des VBOs mais uniquement le changement du comportement de leurs pointeurs).
Au niveau de la géométrie shader la valeur de niveau de détails dans l'attribut Id, est comparée avec la valeur niveau de détails dans la description de l'instance s'ils sont différant alors l'instance ne sera pas dessinée, pour éviter qu'une instance soit dessinée avec plusieurs niveau de détails.

- La troisième composante est le type de la primitive, qui détermine si le sommet appartient à une branche ou une feuille, elle permet de définir la façon dont sera traité le sommet par le vertex shader et le fragment shader.

Au niveau de vertex shader, cet attribut permet selon la méthode de rendu de :
Affecter à chaque sommet les coordonnées de texture en cas d'utilisation du texturage, qui sera appliqué par cylindre ou ellipsoïde.
Faire un calcul d'illumination différent pour les branches et les feuilles, en utilisant des structures uniformes qui contiennent des propriétés de matériaux différentes pour chaque type (propriété diffuse, spéculaire, brillance …), et même des méthodes de calcul différentes (par exemple on peut calculer une composante de lumière spéculaire pour les feuilles et ne pas la calculée pour les branches).
Appliquer des méthodes différentes du Bump-maping, ou perturbation de normales ou autre méthodes pour d'améliorer les modèles, de manières différentes pour les branches ou ellipsoïdes.
Au niveau du fragment shader :
Faire le calcul d'illumination en cas d'illumination par fragment, en fonction de type de primitive.
Déterminer la texture à utiliser parmi deux textures chargées dans des simplers 2d.

2.1.2. Next position et next normal :
Deux vecteurs qui contiennent consécutivement les coordonnées du prochain sommet, et sa normale. En plus des trois sommets du triangle en entrée du géométrie shader, ces deux attributs

sont utilisés pour déduire pour chaque sommet le prochain sommet avec lequel il forme le début d'une surface.
Pour pouvoir implémenter la simplification du dessin par le changement de déplacement de pointeur de sommets dans l'appel API du dessin il faut pouvoir avec n'importe quel niveau de maillage être capable d'accéder aux sommets de début et de fin des surfaces même si certains surfaces de la primitive sont éliminées par le déplacement du pointeur de sommets, c'est ce que ce nouveau attribut assure avec un mécanisme de dessin approprié implémenté au niveau du géométrie shader.

3. Mécanisme du dessin :

L'utilisation de la méthode traditionnelle de dessin des triangle-strips ne permet pas la simplification du maillage par élimination de surface, par ce que le changement des déplacements du pointeur de sommets va conduire à la perte de la forme.
Dans la figure Fig79 on peut voir que le saut du sommet 1 après le sommet 0 rend la construction d'une surface à partir du sommet 0 impossible

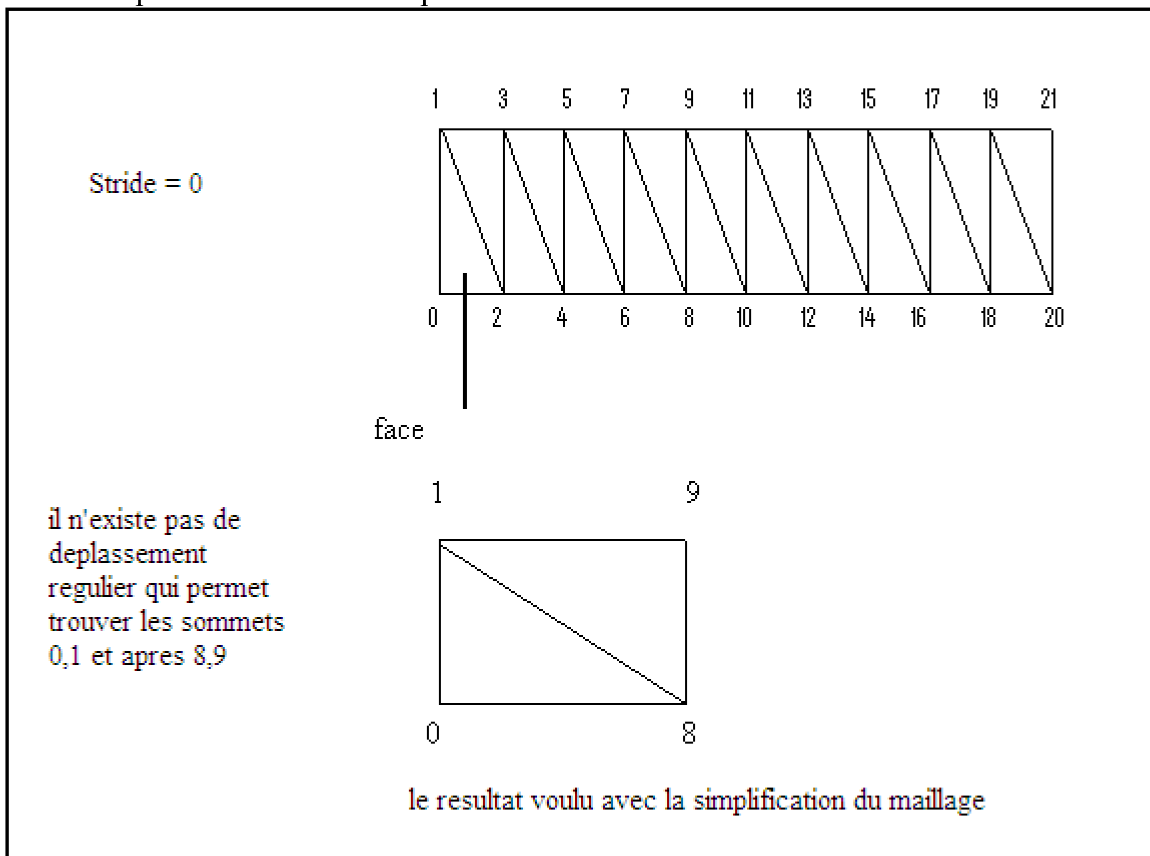


Fig79 : Problème de saut de sommets pour l'élimination de surfaces.

Avec l'utilisation des indexes on peut reconfigurer les indexes des sommets pour fusionner les surfaces, mais ceci engendre l'utilisation d'un buffer d'indexes dynamique qui sera mis à jour pour plusieurs fois pour la même image, impliquant :

- Un cout très élevé en temps de calcul pour la reconstruction du buffer qui contient un très grand nombre d'indexes (égale au nombre de sommets de toute les modèles).
- La réécriture du buffer d'indexes impose l'utilisation d'un buffer dynamique, moins performants que les buffers statiques.
- Le coût de transfert énorme des nouveaux buffers d'indexes.

Pour remédier à ce problème nous avons utilisé un mécanisme de dessin qui utilise les attributs next position et next normal, pour dessiner nous modèle avec la possibilité de l'illimitation de surfaces.
Les sommets de chaque modèle sont organisés de façon à ce que chacun avec son suivant forme une limite de surface, alors en prenant chaque next pos et next normal d'un sommet comme le sommet qui le suit dans le buffer, on aura pour chaque sommet la possibilité de construire une bordure d'une surface.
Dans ce qui suit on appellera next d'un sommet, ses deux attributs next normal et next position pour la simplification.
Dans la figure FIG.80.On peut voir comment le next de chaque sommet est choisie, pour le sommet 0 le next sera 1, pour le 2 le next sera 3 et ainsi de suite, alors dans l'application de rendu la définition de ces attributs revient à devenir de nouveaux pointeur sur le VBO contenant les sommets, en prenant comme début la valeur du pointeur au deuxième sommet avec le même stride que le pointeur de sommets.
Avec cette méthode il ne faut pas passer tous les sommets au GPU, mais seulement la moitié, l'autre moitié va être accessible via le next de chaque sommet, alors les strids de pointeurs des sommets et next vont être multiplié par deux.
Pour fusionner chaque n surfaces en une seul, il faut prendre les strids des sommets et des nexts égale à n*2 taille d'un sommet sur le VBO, par exemple pour la première surface si on veut dessiner les quatre surfaces, on passe au GPU les sommets 0,2,4, et 8 avec les attributs next 1,3,5,7, et 9, pour fusionner chaque quatre surfaces en un seul, on prend le strid égal à 6*taille du sommet (3*2*taille d'un sommet), alors on aura dans le GPU les deux sommets 0 et 8 avec les attributs next 1 et 9 qui seront suffisants pour dessiner la nouvelle surface fusionnée.

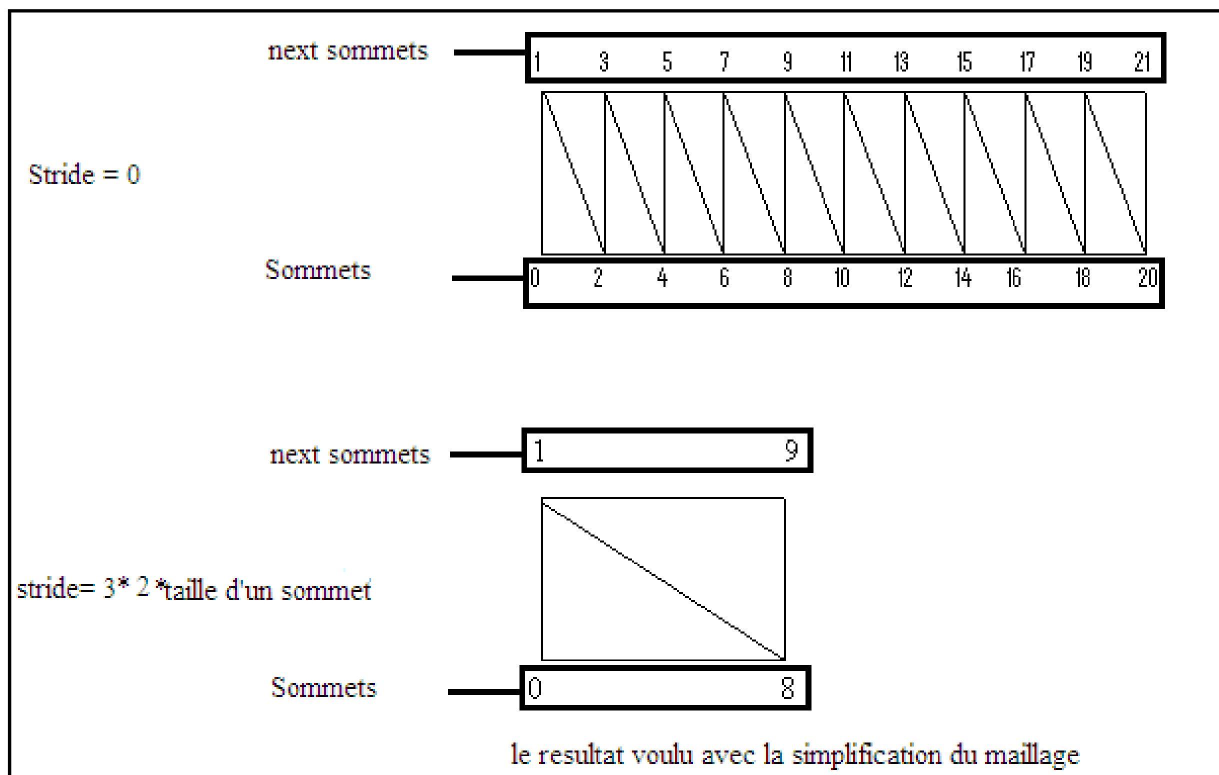


Fig80 : Utilisation des attributs next position et next normal pour la fusion des surfaces

3.1. Production de surfaces par géométrie shader :

Dans le buffer des sommets des modèles d'arbre les sommets sont placés dans un ordre qui permet la construction de surfaces consécutivement pour produire la primitive.

La géométrie shader reçoit comme entrée trois sommets paires (Fig. 81) qui ont l'attribut next les sommets impaires, par ce qu'on a désigné comme entrés du géométrie shader les triangles, et pour chaque exécution il produit en sortie deux triangles qui forment une surface.

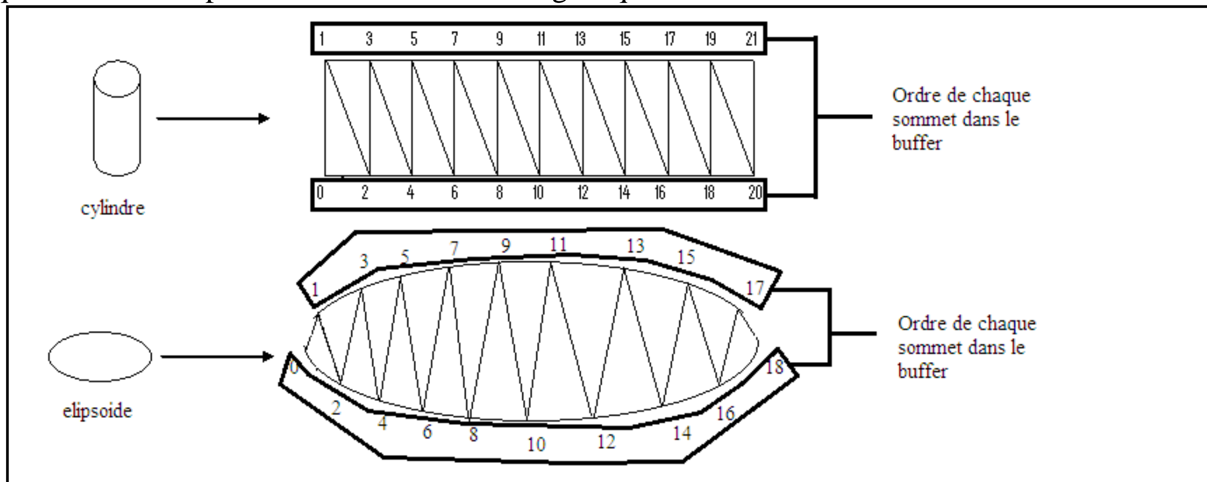


Fig81 : Ordonnancement des sommets dans le vertex buffer

La construction des triangles de sortie se fait avec des appels EmitVertex() en combinant les sommets et les nexts comme suit :
On notraces sommets en entrée : sommet0, sommet2, et sommet4.
Les nexts seront notés comme : sommet1, sommet3, et sommet5.
Alors l'algorithme de dessin sera :
(On utilise un pseudo code pour la simplification le code complet en GLSL du géométrie shader sera présenté par la suite dans la partie implémentation)

```
glPosition= sommet0 ; glNormal=normal0
glPosition=  next_position0 ;/*sommet1*/ glNormal=next_normal0 ;/*normal au sommet1*/
glPosition= sommet2 ; glNormal=normal2
EmitVertex() ;
glPosition=  next_position0 ;/*sommet1*/ glNormal=next_normal0 ;/*normal au sommet1*/
glPosition= sommet2 ; glNormal=normal2
glPosition=  next_position2 ;/*sommet3*/ glNormal=next_normal2 ;/*normal au sommet3*/
EmitVertex() ;
End primitive () ;
```

L'API en cas d'utilisation triangle_strips sans indexation (tous les sommets passent le pipeline dans l'ordre de leurs positions dans le vertex buffer) alimente le géométrie shader en sommets sous forme de tableau de trois sommets à la fois, à chaque fois un sommet de ce tableau est remplacé par un nouveau.

Le géométrie shader est capable de manipuler les sommets dans l'ordre de leurs arrivé de la manière décrit dans la figure (fig82).

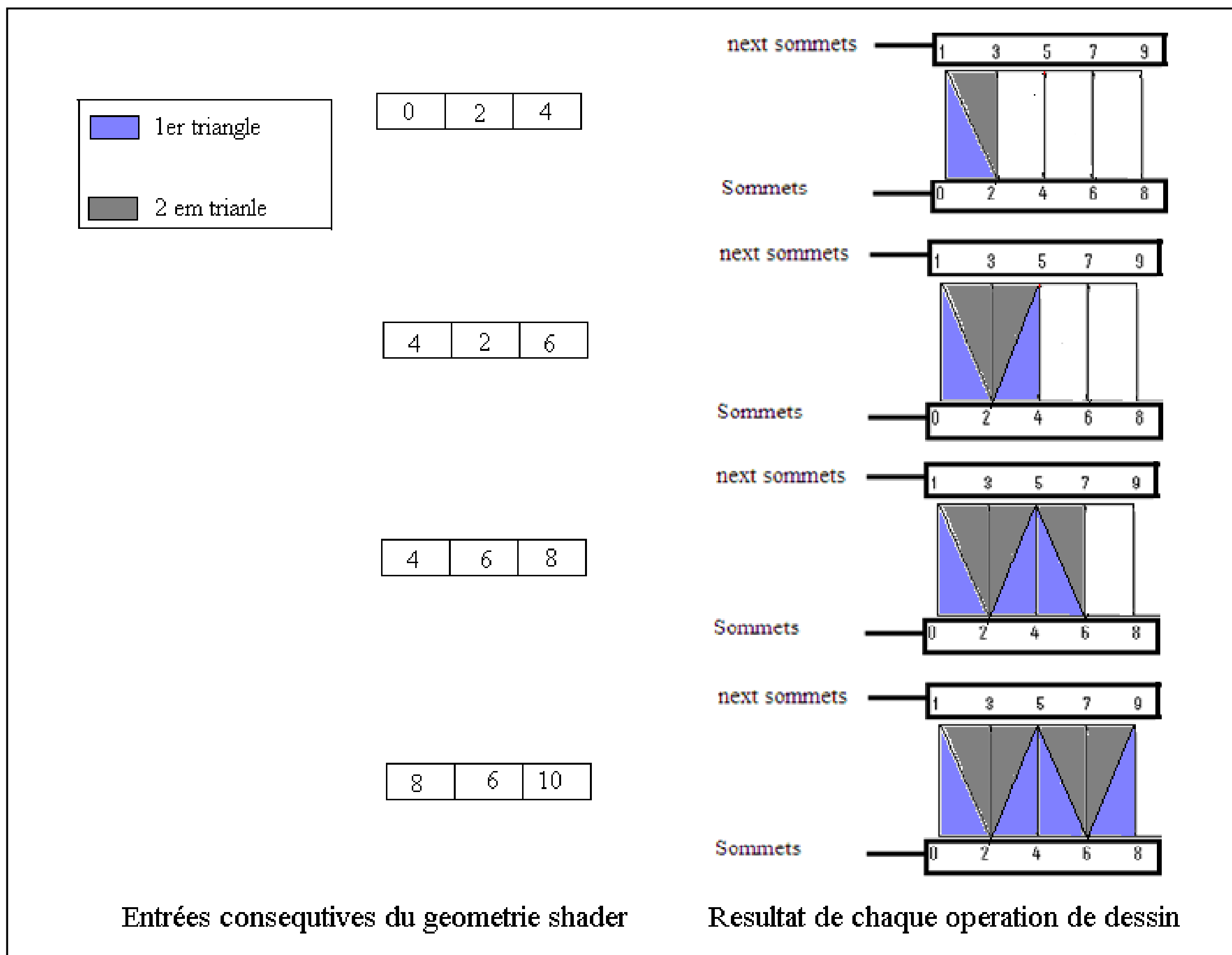


Fig82 : Mécanisme de dessin

3.2. Accès aux données d'instances :

Pour accéder aux données de chaque instance, on utilise l'attribut « debut » qui est une valeur qui indique l'index où commence les coordonnées des positions des arbres instanciés à partir du modèle auquel appartient le sommet, on a introduit cet attribut pour remédier au problème de l'incapacité des GPUs à boucler dynamiquement (dynamic looping), au niveau des shaders le nombre de fois qu'une boucle est exécuté doit être déterminé lors de la compilation du shader, et puisque les positions des instances qu'on utilise sont passées au programme de rendu dans des tableaux uniforme il est impossible faire des mises à jour ou changer les position à chaque fois qu'on change de modèle pour la même frame.

Le bouclage sur tout le tableau avec une comparaison d'un attribut « modèle » qui indique pour chaque sommet et chaque position (instance) le modèle auquel il appartient est possible mais présente deux inconvénients :

- Le coût en temps de calcul pour boucler sur tout le tableau qui va être multiplié pour chaque sommet par le nombre de modèles utilisés, si on utilise n instances pour chaque modèle avec m modèles, au lieu d'avoir pour les sommets de chaque modèle une boucle qui s'exécute n

fois, on aura n exécutions du traitement et n*m tests inutiles car le nombre n est le même pour tous les modèles.
n*m tests est un coût très élevé, si on considère le nombre très important de sommets de chaque modèle.

- Une des raisons pour lesquelles le nombre des boucles doit être déterminé lors de la compilation du géométrie shader est que le driver d'affichage doit être capable de déterminer s'il y a dépassement de la capacité de génération des sommets par GPU, si on boucle sur toutes les instances avec la possibilité de génération de sommets pour toutes ces instances, le driver va considérer dans chaque possibilité de génération lors de calcul des sommets que le programme de rendu va générer, et indiquera l'erreur de déplacement des capacité hardware, même si en réalité on les a pas dépassé, résultant ainsi en nombre de sommets possibles à générer par GPU divisé par le nombres de modèles qu'on utilise.

La solution que nous avons utilisée consiste à utiliser un attribut « debut », pour déterminer le premier index des coordonnées d'instances de chaque modèle, la fin est intuitivement implémentée dans le code du géométrie shader et qui est égale à trois fois le nombre d'instances de chaque modèle.
Pour le bouclage au niveau de la géométrie shader, on utilise un index de boucle initialisé à zéro jusqu'à trois fois le nombre d'instances des modèles et qui est incrémenté de trois à chaque itération, auquel on ajoute pour chaque modèle l'attribut debut, debut+1, et debut+2 pour accéder au coordonnées de ces instances.
L'attribut debut est utilisé de la même façon pour indexer le tableau de description des instances, même si nous n'avons pas besoin de trois valeurs pour la description d'une instance on réserve trois places pour chaque instance dans le tableau, pour pouvoir réutiliser le même attribut debut, dans la même boucle pour accéder a la description d'instances sans calcul supplémentaire.

```
Uniform floatpos[n*3]// tableau des positions avec n instances
Uniform intdesc[n*3] // tableau de description pour n instances
Varying in floatdebut// attribut debut du sommet, dans un géométrie shader cet attribut   //est un
tableau qui contient une valeur pour chaque sommet en entrée
Int i=0 ,k=0 ;// index de bouclage
Floatdx,dy,dz ; // coordonnées de la position de l'instance
While( i/n) { // n le nombre d'instances
k=i+ debut // calcul de l'index pour la i'eme instance du  modèle courant
if (desc [k]==0) // accès a la description de l'instance pour test de visibilité
 dx=pos[k] ;dy=pos[k+1] ;dz= pos[k+2];
// Exécution du reste de la boucle
} //fin if
I+=3 ;
} fin de la boucle
```

3.3. Détermination de niveau de détails :

Avec la représentation de géométrie que nous avons adoptée il est possible de générer plusieurs niveaux de détails, par deux façons ;
La première est par simplification de maillage, en appliquant la fusion des surfaces expliqué précédemment. L'exploitation de cette approche est faite en utilisant l'information « saut » sauvegardée lors de l'interprétation des modèles en géométrie, qui indique le nombre de sommets à sauter entre niveaux de détails.
Cette valeur est choisie d'une façon qui conserve la forme des cylindres (ou produit une forme

cubique pour approximer la forme des cylindre), en assurant que le sommet du début coïncide avec celui de la fin (avec leurs nexts) et que le nombre des surfaces après fusionnement ne soit pas inférieur à quatre (on peut même aller a trois ce qui donne au branche une forme …….).
Avec les feuilles les contraintes sur cette valeur sont plus faibles, il suffit de produire une surface de n'importe quelle forme, avec la conservation des dimensions de la feuille qui est assurée par les coordonnés des sommets, par ce que de loin la forme des feuilles devient peut percevable. On peut même utiliser une seule surface (formée de deux triangles) par feuille.
Lors de choix de la valeur « saut » le plus important est de conserver la forme des branches et s'assurer qu'elles sont présentées avec des formes fermées.
La deuxième consiste à partir de la description des branches du modèle, et produit une approximation très grossière de l'arbre, en créant à partir des informations : centre, axe, hauteur, et rayon une surface tournée vers l'observateur avec quatre sommets dont deux pointent vers le haut et les deux autres pointent vers le bas pour avoir une illumination proche de celle d'un cylindre(avec l'interpolation des normales entre géométrie shader et fragment shader), les feuilles sont présentées par deux autres surfaces parallèles générées à partir de l'information « intensité » qui est calculée en fonction de nombre de feuilles sur la branche.
Cette méthode est appliquée au niveau du géométrie shader pour les instances très lointaine, avec un appel API du dessin séparé sur le VBO contenant la description des branches du modèle, ces instances sont repérées par une valeur spécifique du niveau de détails dans le tableau de description des instances.
La détermination du niveau de détails par GPU est possible par ce que tous les sommets sont générée par géométrie shader sur GPU, mais le cout va être énorme.
Si on applique la détermination du niveau de détails par géométrie shader alors le calcul des distances et les tests vont être répétés pour chaque sommet, ce qui est très couteux même sur les GPUs récentes, en plus ces calculs vont avoir presque les mêmes résultats par instance.
Une alternative plus raisonnable est de déterminer le niveau de détails par CPU pour chaque instance, et le passé au GPU dans le tableau de description de l'instance.
L'implémentation par CPU produit le calcul du niveau de détails une fois par instance d'arbre, ce qui est un nombre négligeable comparé au calcul par GPU, mais produit un transfert des résultats entre CPU et GPU, ce transfert est acceptable par ce que la quantité d'information à transférer est trop petite (un entier par instance d'arbre).

3.4. Implémentation par géométrie shader :

Au niveau du géométrie shader, la simplification du maillage est détecté en comparant la deuxième composante du vecteur id avec le niveau de détails indiqué dans le tableau de description des instances, si cette valeur est égale à la valeur donnée par la description, l'instance est dessinée si non elle n'est pas dessiné (une instance n'est pas dessiné si le géométrie shader n'exécute aucun EmitVertex( ) pour l'itération courante de la boucle sur les instances d'un modèle).
Cette comparaison évite de dessiner toutes les instances avec tous les niveaux de détails, malgré qu'avec la simplification du maillage tous les sommets passés au GPU appartiennent au nouveau maillage (ils sont sélectionnés par le changement du stride), cette comparaison est importante parce que si elle n'est pas faite, toutes les instances seront dessinées par le premier appel API du dessin avec le stride correspondant au niveau du maillage le plus fin, et redessinées avec le deuxième appel avec le stride de simplification de maillage, et ainsi de suite. Enfin toutes les instances seront dessinées avec tous les appels de dessin des niveaux de simplification du maillage.

4. Clipping d'instances :

L'élimination des instances qui sont hors de la pyramide de vision est très intéressante dans la représentation d'un très grand nombre d'arbres, car le temps gaspillé en cas où cette opération

n'élimine pas un très grand nombre d'arbres, est récupéré par la diminution du niveau de détails.
Pour que la pyramide de vision contient un grand nombre d'arbre ou tous les arbres, le point de vue doit être situé très loin par rapport aux arbres alors ils sont presque tous dessiné avec un niveau de détail moyen ou grossier, dans le cas où le nombres d'arbres à l'intérieur de la pyramide de vision est petit, le point de vision doit être situé près d'un nombre important d'arbres, qui seront dessinés avec un niveau de détails élevé, ce qui est très couteux en temps de calcul, alors l'application du clipping pour les arbres hors de la pyramide de vision devient très bénéfique.
Pour les même raisons de redondance de calcul que celle de détermination du niveau de détails l'implémentation du clipping est très couteuse sur GPU, alors nous l'avons implémenté sur CPU est plus convenable, et le résultat est une valeur entière égale à zéro indiquant que l'instance est éliminée ou un indiquant que l'instance est à l'intérieur de la pyramide de vision, ces résultats sont transmis au GPU dans le tableau de description des instances.
L'approche de détermination de l'appartenance à la pyramide de vision doit être le plus rapide possible, par ce qu'elle est implémentée au niveau du CPU et les résultats doivent être transmis dans un tableau uniforme alors toutes les valeurs doivent être calculées par frame une fois pour toute.
Nous proposons une méthode approximative basée sur des cubes englobant des modèles d'arbres calculés lors de l'interprétation en géométrie, si l'un des sommets du cube est à l'intérieur de la pyramide de vision alors l'instance est considéré visible.

4.1. Application du clipping :

Le clipping des instances est appliqué sur la copie de tableau de position existante au niveau de l'application du rendu. Pour chaque instance on déplace le cube englobant du modèle correspondant vers la position de l'instance, et puis en compare les sommets du cube déplacé aux limites de la pyramide de vision, pour déterminer si l'instance est à l'intérieur ou non.
En fonction des résultats, on mit à jour la copie du tableau de description des instances qui va être transmit au GPU pour rendre la prochaine image.
Le test de visibilité est la première étape que le géométrie shader effectue pour chaque instance, pour chaque itération de la boucle sur les instances d'un modèle, il vérifie la valeur dans le tableau de description d'instances (avec l'index de l'itération + debut), si l'instance n'est pas marquée comme visible il passe à la suivante.
L'élimination des instances est très efficace en cas de navigation à l'intérieur des arbres, et apporte une augmentation très signifiante au débit des images temps que l'observateur navigue dans la scène à l'intérieur des arbres.
Le gain en temps de calcul relatif à l'illimitation des instances est très grand, et permet le rendu en temps réel des scènes comportant de très grand nombres d'arbres, en gardant le nombre d'instances visibles dans les limites des performances de l'application de rendu en jouant sur le plan far de la pyramide de vision, et les probabilités de couverture entre instances.
En utilisant un plan far relativement proche, les instances sont éliminées si elles sont situées plus loin que ce plan et elles réapparaissent avec le rapprochement de l'observateur, l'apparition des nouveaux arbres peut passer inaperçu par l'observateur en fonction du niveau de la couvertures des instances entre elles, alors la valeur de la distance du plan far doit être choisie en fonction de la densité des feuilles des modèles d'arbres, si les modèles sont très dance il aura de forte probabilités de couverture des instance a une petite distance, si non la probabilité de couverture des instances a certaine distance diminue avec la diminution de la densité des modèles et la distance du plan far doit prendre des valeurs plus grandes.
Un plan far plus loin résulte en plus d'instances visibles, mais ces instances ont une densité plus petite, alors leurs rendu est moins coûteux, ce rapport entre densité et nombre d'instances permet de maintenir les performances de l'application même avec des modèles lourd avec une intensité de feuilles importante.

Le cas le plus pire avec l'application de cette méthode se produit quand toutes les instances (ou la plupart d'entre elles) sont visibles, lorsque le point de vue est situé à une position très lointaine et plus haute que les arbres, dans ce cas l'élimination des instances devient inutile, elles seront tous visible.
Un tel cas n'est pas probable avec la navigation à l'intérieur des arbres, mais la perte en temps de calcul sera récupérée par la diminution du niveau de détails résultant de la grande distance entre l'observateur et les instances d'arbres.

5. Traitement par modèle :

L'instanciation de géométrie présente un avantage pour le rendu, qui est la possibilité d'appliquer des opérations sur les sommets des modèles puis propager les résultats sur toutes ces instances, produisant ainsi les résultats de ces opérations avec un temps de calcule réduit un nombre de fois égale au nombre d'instances de chaque modèle.
Les opérations applicables par sommets comme l'illumination per-vertex, peuvent avec l'instanciation de géométrie par géométrie shader être appliquées une fois par modèle, dans le vertex shader, pour modifier des attributs des sommets du modèle existants ou calculé de nouveaux attributs, et réutilisés en suite pour toutes ces instances lors de la génération des sommets des instances.
Si on veut appliquer un traitement par sommets sur des instances d'un modèle il suffit de le faire une seule fois pour le modèle au niveau du vertex shader pour modifier les attributs du modèle, et quand le géométrie shader instancie ces sommets il les duplique avec les nouveaux attributs, alors le temps de calcule totale nécessaire pour faire ce type de traitement surtout les instances du modèle présents dans la scène est divisé par le nombre d'instances.
Cette possibilité est très bénéfique dans le cas de rendu probabiliste qui est souvent utilisé pour le rendu des arbres en temps réel, avec le rendu probabiliste au lieu d'appliquer des modèles empirique très couteux pour le calcul exacte de l'illumination, il se base sur des probabilités d'illumination calculée à partir de la distribution géométrique des feuilles et la distance à partir de la frontière du feuillage, comme le modèle de Chaudy [11], ou de Boulanger [57] qui propose un modèle d'illumination qui se base sur la distribution spatial des feuilles, pour leurs affectées des propriétés probabilistiques correspondantes à leurs réactions à trois composantes de lumières possibles : une composante environnementale qui simule la lumière qui provient de la réflexion de la lumière par ciel, une composante directe qui définit la réaction à la lumière directe qui provient du soleil et une composante indirecte qui simule toutes les réflexions de lumière provenant des autres feuilles.
Avec une architecture GPU unifiée, il n'existe pas des unités de calculs figées pour le vertex shader, géométrie shader, et fragment shader. La distribution des unités de calculs disponibles sur le GPU est automatiquement reconfigurée relativement à la complexité de chacun, cette redistribution est appliquée avec l'optimisation du programme après le premier appel du dessin.
L'optimisation du programme fait essentiellement deux tâches [49] :
La réallocation de mémoire aux variables du programme de rendu en fonction de leurs fréquence de lecture et de modification, en mettant les variables les plus fréquemment utilisées en lecture dans la mémoire la plus rapide en accès en lecture, et les variables qui changent constamment les tampons les mieux adaptés, et la redistribution des unités de calcul selon le temps d'exécution de chaque étape du programme de rendu.
Le nombre d'exécutions du vertex shader avec l'instanciation de géométrie est beaucoup plus petit qu'avec le dessin directe de toute la géométrie (il n'est exécuté que pour les modèles), mais le nombre d'exécution du fragment shader est le même, alors l'exécution des calculs redondants sur toutes les instances avec fragment shader ne causera pas seulement la perte du temps de calcul pour trouver les mêmes résultats mais aussi conduira à une mauvaise allocation des unités de calcul, et à

la perte d'un nombre important de ces derniers pour recalculer les mêmes résultats au lieu de les exploitées pour accélérer le vertex shader ou le fragment shader.

5.1. Application du traitement par modèle :

5.1.1. Illumination approximative per-vertex :

Une application très bénéfique en temps de calcul du traitement par modèle est le calcul de l'illumination per-vertex empirique ou probabiliste.
Avec l'instanciation de géométrie il n'est pas possible de calculer une illumination exacte pour tous les arbres de la scène dans le vertex shader, par ce qu'il n'est pas exécuté surtout les arbres mais seulement sur les modèles qui seront instanciés par le géométrie shader.
Les arbres sont en général rendus à l'extérieur ou la source de lumière est le soleil et tous les arbres sont illuminés de la manière. Alors si on calcule l'illumination exacte pour chaque instance individuellement, le changement des résultats relatif à la source de lumière ne sera pas très grand entre une instance et une autre.
Avec un modèle de Phong, la composante qui sera affectée le plus le changement de position (le modèle par rapport à ces instances), est la composante spéculaire.
Dans le cas des arbres, les branches sont de nature matte, les feuilles peuvent avoir une interaction spéculaire, mais à cause de leurs taille réduite la différence entre un calcul approximatif (par modèle) et exacte (pour chaque instance) est difficilement remarquable par l'observateur de la scène, à condition d'assurer la consistance de la sécularité au changement de la position de la source lumineuse.
Dans ce qui suit on site deux façons d'exploiter le gain en temps de calcul de l'illumination engendré par le traitement par modèle, la première est d'appliquer un modèle empirique (de Phong par exemple) sur les modèles et la deuxième est l'application des méthodes probabilistes présentés par Chaudy [11] et Boulanger [56].

- **Illumination par modèle empirique :**

Une bonne approximation de l'illumination des arbres de la scène est d'exécuter un modèle empirique d'illumination sur les modèles et d'appliquer après les résultats directement sur toutes les instances, cette méthode donne des résultats très proche que l'illumination exacte (per-fragment), et consistants aux changements de la position de la source lumineuse.
La consistance de l'illumination vient du fait qu'elle est calculée avec des valeurs exactes per-vertex pour chaque modèle, alors l'effet du changement de la position de la lumière est calculé exactement.

- **Illumination par modèles probabiliste :**

Une autre façon de tirer profit du traitement par modèle est l'illumination probabiliste, qui peut être avec notre modèle étendu même pour supporter des simplifications de calcules en fonction du niveau de détails courent.
Au niveau du vertex shader le changement du niveau de détails est perceptible de la même façon qu'au niveau du géométrie shader (en testant la deuxième composante de l'attribut id des sommets), par exemple si on applique le modèle avec les composantes environnementale directe, et indirect, on peut par exemple éliminer la composante indirecte pour certains niveaux de détails.

5.1.2. Ombres générés par les feuilles sur les autres feuilles :

Le calcul exacte des ombres générés par les feuilles entre elles sont très couteux, à cause de leurs très grand nombre, et les résultats exactes ne sont pas aussi important sur l'image rendue, par ce que l'observateur ne perçoit que leurs présence générale dans la scène à cause de leurs grand nombre est leurs petites tailles.

Une alternative très efficace pour le rendu temps réel est un calcul approximatif de ces ombres, Chaudy [11] propose dans son modèle d'illumination une méthode probabiliste, où la possibilité que la feuille soit à l'ombre dépend de sa position dans le feuillage (distance par rapport à la limite du feuillage), et la direction de la lumière.
Boulanger [56] propose le calcul d'un masque d'ombrage de chaque feuille, représentant un environnement d'occlusion des feuilles adjacentes, qui représente les résultats de la visibilité de la feuille à partir du soleil en fonction de la direction d'illumination, une telle technique va être très couteuse en mémoire alors il propose l'utilisation d'un seul masque générique pour toutes les feuilles générées à partir des propriétés de la distribution spatiale des feuilles (comme l'intensité), qui n'est pas exacte mais qui va être assez convaincante, les valeurs obtenues à partir de ce masque seront perturbés en fonction de la distance de la feuille par rapport à la limite du feuillage pour ne pas avoir une occlusion uniforme sur tout le feuillage, et augmenter le réalisme des résultats.
Avec l'instanciation de géométrie ces méthodes ne seront appliqués qu'au modèles au lieu d'être appliquées sur toute la géométrie, qui est très bénéfique en temps de calcule, et diminue leurs cout en mémoire, en ne sauvegardant les informations nécessaire pour leurs application que par modèle. Les résultats obtenus seront les même par ce que ces méthodes ne dépendent que de la direction de la lumière et la distribution spatiale de feuilles.
De la même façon que pour l'illumination probabiliste ces méthodes peuvent être simplifiées en fonction du niveau de détails.

5.2. Animation d'arbres :

Puisque tous les arbres dans la scène réagissent de la même manière au vent, nous pouvons implémenter une animation qui simule la réaction au vent au niveau du vertex shader, et tirer profit du gain en temps de calcul qu'offre le traitement par modèle.
La réaction des arbres au vent se traduit par un déplacement des sommets qui leurs forme en fonction de leurs distances par rapport au tronc de l'arbre et en fonction du temps, pour implémenter cette réaction nous avons utilisé une valeur réelle « temp » qui est transférée au vertex shader sous forme de uniforme, pour chaque image.
En considérant que le vent bouge selon l'axe $\overrightarrow{ox}$ (pour simplifier) le mouvement que vont faire les sommets des branches et feuilles est un mouvement d'aller et de retour selon la composante x de leurs position, et afin de pouvoir simuler ce mouvement, on utilise une fonction de cosinus, qui génère les valeurs qui sont ajoutées à la composante x des coordonnées des positions, en fonction du temps, qui sont multipliées par la puissance du vent alors :

$$x = x + y^2 * p * \cos(temp)$$ Avec p : la puissance du vent.

Nous avons introduit l'effet de la hauteur en multipliant la puissance du vent par y mais pour avoir plus d'effet nous avons utilisé la carré de y (donne de meilleurs résultats dans les tests).

6. Texturage :

Nous proposons une méthode de calcul de coordonnées de textures dans le vertex shader, par primitive (cylindre ou ellipsoïde), on utilise des textures carrées dont les coordonnées sont affectées aux sommets à partir d'une variable uniforme pour chacune des deux types de primitives. Ces coordonnées sont prè-calculées par l'application avant de commencer la boucle de rendu de la façon suivante :
La coordonnée u reçoit les valeurs commençants à zéro et incrémentée pour chaque deux sommets consécutifs d'une valeur égale à (1/nombre de sommets)*2 de la primitive.
La deuxième coordonnée v est alternativement assignée par la valeur 0 et 1.

Par exemple si le nombre de sommet est égal à 10 sommets, on a :

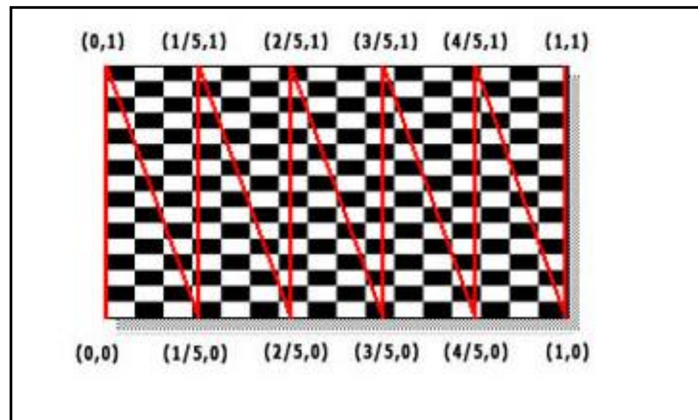


Fig83 : Génération de coordonnées de texture

7. Rendu de scènes avec un très grand nombre d'arbres :

Le programme de rendu est limité à un nombre maximal d'arbres qui peut être traité en temps réel, mais en utilisant l'illimitation d'instances on peut augmenter le nombre d'arbres qui peuvent être présents dans la scène.
Puisque l'élimination des instances hors de la pyramide de vision se fait par l'application (CPU), il est possible de garantir à tout instant des performances temps réel, en garantissant de ne passer au programme de rendu (sur GPU) que des instances qu'il peut gérer en temps réel.
L' application doit reconstruire à chaque fois les tableaux de position est de description des arbres selon la position de la caméra, dans une scène qui contient un nombre supérieur d'arbres que le programme de rendu peut traiter en temps réel, l'application va sélectionner les arbres visibles (à l'intérieur de la pyramide de vision), les trier par modèles, de façon compatible à l'instanciation par géométrie shader, construire le tableaux de description et puis transmettre les nouvelles positions et descriptions d'instances au programme de rendu.
Dans ce cas il peut avoir pour un modèle un nombre d'instances supérieur au nombre possible par le programme de rendu, alors on aura besoin de plusieurs appels de dessin sur le modèle avec des positions d'instances différentes, dans ce cas le problème qui se pose est la modification des attributs début des sommets du modèle entre les appels du dessin qui seront utilisés pour dessiner ses instances.
La mise à jour des VBOs peut résoudre le problème mais elle engendre la diminution des performances à cause du temps de transfert.
Une autre solution peut être d'utiliser un VBO séparé pour l'attribut « debut » qui contient toutes les valeurs possible du début et manipuler les pointeur dans ce VBO pour chaque appelle, ca qui impose que la taille du VBO soit égale au nombre de sommets du modèle qui a le plus de sommets multiplié par le nombre de modèles, et les pointeurs des débuts de l'attributs « debuts » pour le modèles seront pris régulièrement dans le VBO, alors si on veut redessiner un modèle sur une différente zone du tableaux des position en effectue un deuxième appelle du dessin avec le pointeur correspondant dans le VBO.

8. Terrain et ombres portées sur le terrain:

8.1 Construction du terrain :

Dans notre implémentation nous avons utilisé un terrain construit à partir d'une carte des hauteurs (hightsmap), sous forme d'image bitmap a niveaux de gris, qui donne pour chaque point au coordonnées x,z, une valeur de hauteur du terrain y, calculée à partir des valeurs RGB par «h= R+G+B/3/255 », pour obtenir une valeurs entre zéro et un, puis on multiplie la hauteur par la différence entre les coordonnées y de la position maximale et minimale entre lequel nous allons dessiner le terrain pour obtenir la hauteur réelle réduite à l'intervalle de hauteurs dans lequel en veut

dessiner le terrain:
« h=h*(maxy-miny) », alors la coordonnée y pour un sommets du terrain va être « y=ymin+h » avec h la hauteur obtenu en fonction des coordonnées x et z a partie de la carte des hauteurs.
Apres l'extraction des hauteurs de la carte, à chaque coordonnée x on ajoute la plus petite valeur de x à partir de laquelle on veut dessiner le terrain, et de même pour z.
Les normales sont obtenues par le produit vectoriel entre les vecteurs générés par les deux voisins sur l'axe $\overrightarrow{ox}$et $\overrightarrow{oz}$ à partir du sommet au quel on veut calculer la normale.
Pour avoir un terrain plus souple en recalcule, chaque normale par une moyenne, en utilisant les normales des quatre sommets voisins en utilisant l'équation :
Soit xn, yn, zn les composantes de la normale au sommet considéré, et xni,yni,zni les composantes de la normale du voisin i alors :
$xn = (xn + \sum_{i=0}^{4} xni)/5$, $yn = (yn + \sum_{i=0}^{4} yni)/5$, $zn = (xn + \sum_{i=0}^{4} zni)/5$
Cette formule donne les résultats attendus parce que les distances entre les coordonnées x et z sont uniforme sur tout le terrain, en fin la normale va être le résultat de la normalisation du vecteur obtenu.

8.2. Gestion des positions sur le terrain :

Avec les coordonnées des sommets formant le terrain, nous devons réaliser un mécanisme permettant d'obtenir des positions cohérentes sur le terrain pour dessiner les arbres, pour cela on a utilisé une structure de sauvegarde qui permet d'accéder aux coordonnées des sommets du terrain, en utilisant deux indexes dont la valeur est entre zéro et un, cette indexation sur l'espace du terrain permet d'obtenir des position (x,y,z), à partir des index s et t avec des hauteurs cohérentes sur le terrain.
Les coordonnées x, y, z retournées pour deux valeurs s et t sont calculées par :
x= xmin+s*xmax-xmin et z=zmin+t*zmax-zmin avec :
s et t les indexes d'accès dans l'espace terrain
xmin, xmax les valeurs maximale et minimale de coordonnées x du terrain
zmin, zmax les valeurs maximale et minimale de coordonnées z du terrain
Pour y, on cherche x0, z0, x1, et z1 : deux valeurs existantes dans le tableau des coordonnées du terrain et qui vérifient : x0<=x<=x1 et z0<=z<= z1, et les plus proches a x et z 'la plus petite différence), alors on calcule y par :
y= (hauteur(x0,z0)*((x-x0+z-z0)/4) +(hauteur(x0,z1)*((x-x0+z1-z)/4) +
(hauteur(x1,z0)*((x1-x+z-z0)/4)+ (hauteur(x1,z1)*((x1-x+z1-z)/4).
Avec : hauteur (x,y) : permet d'obtenir la coordonnée y (hauteur) de chaque couple (x,z).

8.3. Ombres :

Pour produire les ombres sur le terrain, nous avons utilisé une carte d'ombre, calculé par GPU, en positionnant la caméra dans la position de la source lumineuse, et en faisant le rendu dans un Frame Buffer Object 2D, qu'on attache au résultat du rendu. Nous spécifions le type du stockage par « GL_DEPTH_COMPONENT24 », que nous l'attachons au buffer de profondeur par la fonction « glFramebufferRenderbuffer », puis nous définissons la texture sur laquelle le résultat va être sauvegardé, par la fonction « glFramebufferTexture2D », enfin on désactive le rendu pour les buffers autre que le depth buffer par un appel « glDrawBuffer(GL_NONE) », et on fait un premier rendu de la scène avec l'illumination désactivée.

Avant d'effectuer le premier rendu dans le FBO, on sauvegarde les matrice Modelview et Projection, relative au rendu à partir de la source de lumière qui seront utilisées lors du deuxième rendu pour déduire les coordonnées de textures pour chaque fragment avec lesquelles on accède à la

texture de profondeur, et pour reproduire sa valeur de profondeur (la coordonnée z dans le repère de la source lumineuse) qui va être comparée avec la valeur stockée dans la texture de profondeurs afin de déterminer si il est illuminé directement par la source de lumière ou non.

9. Algorithme de rendu :

L'application de rendu va effectuer les tâches :

- Chargement des données des modèles,
- Chargement des VBOs et génération des nouveaux attributs : Id, debuts
- Optimisation des shaders : à travers des requêtes au GPU, l'application configure les constantes des shaders en fonction des capacités du GPU, la taille des tableaux de description et de position d'instances, la valeur de fin de la boucle d'instanciation sont initialisée en fonction du nombre maximale des sommets que le GPU peut générer.
- Sélection et modification des shaders à utiliser.
- Compilation et édition de lien des shaders, et chargement des textures
- Pour chaque reshape :
  - Détermination des instances visibles et mis à jour des tableaux de position et de description des instances.
  - Détermination du niveau de détails pour chaque instance
  - Calcul des différents pointeurs et strides sur les VBOs qui vont être utilisés
  - Faire les appels de dessin correspondant à tous les modèles pour chaque niveau de détails.

Conclusion :

Dans la méthode d'instanciation de géométrie avec géométrie shader que nous avons présentée a pour but de maximiser l'exploitation du GPU, mais il existe encore plusieurs problèmes concernant l'implémentation de ce genre de solutions :

- La difficulté de trouver la valeur optimale pour le nombre d'instances : cette difficulté est due l'hétérogénéité des architectures des GPUs, et les différences entre leurs gammes (entre high profile, medium profile, et low profile). Le nombre maximal de sommets que peut générer le GPU peut être récupéré en envoyant une requête à ce dernier, mais ce maximum n'est pas toujours l'optimum, car l'implémentation d'une boucle qui génère les sommets augmente les performances jusqu'à a un certain nombre d'itérations ou la charge de bouclage au niveau d'un seule cluster devient assez grande pur engendrer un temps considérable au niveau des autres clusters qui affecte significativement la performance front-end du GPU, dans ce cas l'exécution de plusieurs appels API du dessin devient plus bénéfique que la génération des sommet localement au niveau du GPU.

- Quand les géométries shaders sont apparus comme une nouvelle étape programmable du pipeline graphique, ils n'étaient pas adaptés à la production d'un grand nombre de sommets, en exécutant des boucles avec un très grand nombre d'itérations, et aussi à cause du problème de « dynamique looping », dans la méthode que nous avons présentée nous avons proposé une solution au problème du dynamique looping en introduisant l'attribut « début » aux sommets des modèles, et nous avons essayé de déterminer un optimum pour le nombre d'itérations sur GPU d'une façon expérimentale.

Sur les GPUs les plus récents, les constructeurs ont donné accès au contrôle des invocations du géométrie shader, avec OpenGL 4.0, GPU shader 5, et GLSL v400 et plus [57], permettant ainsi de déterminer le nombre de fois que le géométrie shader sera invoqué sur chaque sommet, et éliminant la nécessité d'utiliser les boucles pour générer les sommets par géométrie shader, ce qui résout ces problèmes et augmente encore plus les performances.

Le souci avec cette nouvelle implémentation est qu'elle n'est disponible que les GPUs très récents (à partir de 2010), et sur les hautes gammes, mais ce nouveau contrôle sur l'invocation du géométrie shader, et l'introduction du tessellation évaluation shader, et tessellation contrôle shader qui peuvent eux aussi générer de nouveaux sommets, encourage la stratégie d'aller de plus en plus vers la génération des sommets au niveau du GPU plutôt que le transfert de la géométrie à partir du CPU.

Chapitre 5:
Résultats, bilans et conclusions

Chapitre 5

Résultats, bilans et conclusions

1. Exemples de résultats de rendu :

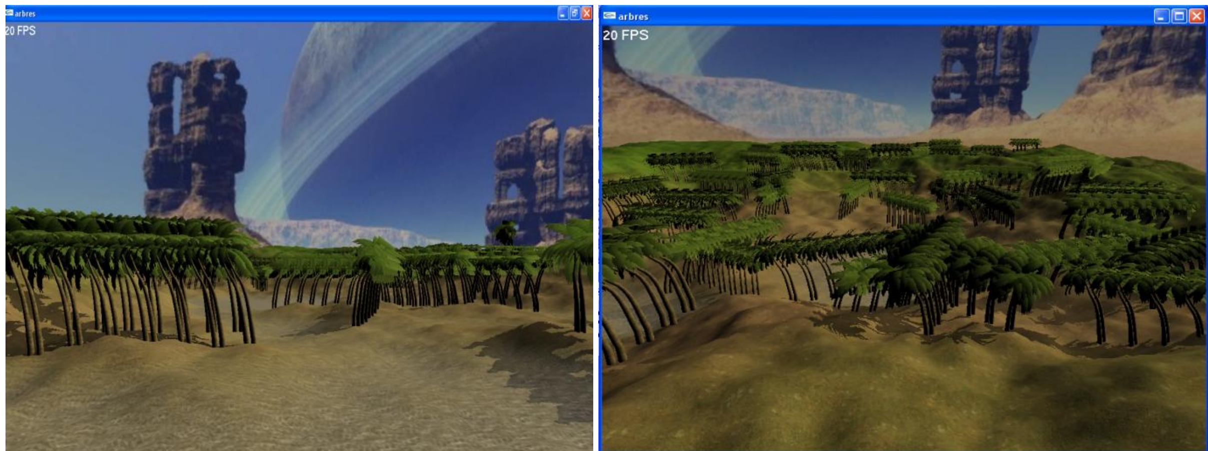


Fig84: Rendu de scène avec 220 arbres

Fig85 :Les ombres portés sur le terrain

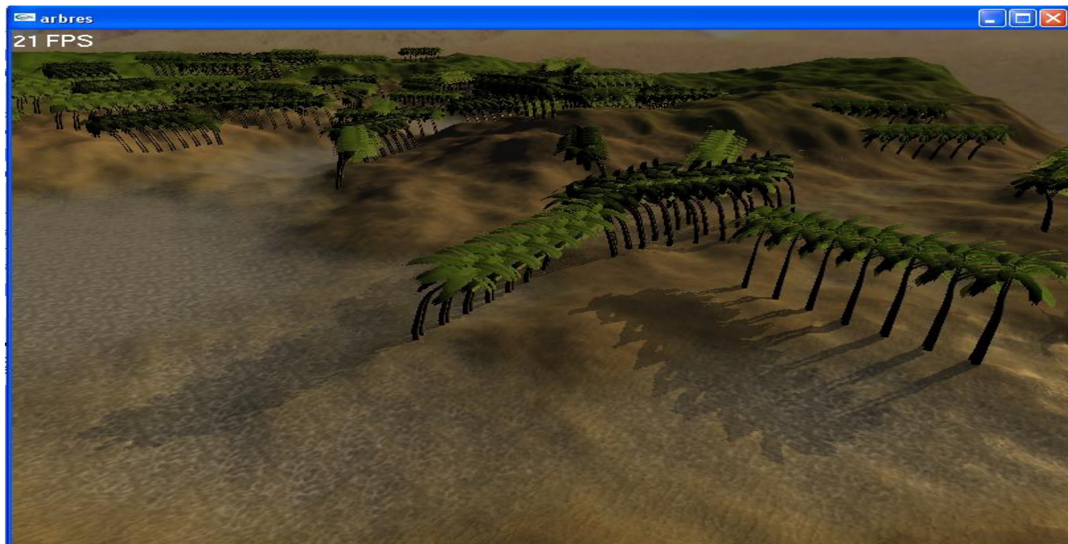


Fig86 : Changement de position et intensité de la lumière

2. Performances :

2.1 Test en fps (image par seconde)

Pour tester et évaluer les résultats obtenus nous avons comparé les performances de l'application qui utilise l'instanciation de géométrie avec une application qui dessine tous les arbres sans instanciation de géométrie.

L'évaluation est calculée sur trois aspects, la consommation de mémoire, les performances en frame per second (fps), et l'occupation du CPU, en variant la complexité des arbres (calculé en nombre de sommets), et le nombre d'arbres à rendre.

Ces tests ont étés effectués sur un system équipé d'une carte graphique Nvidia 220GT, et un processeur Dual-Core CPU 2.8 GHZ.

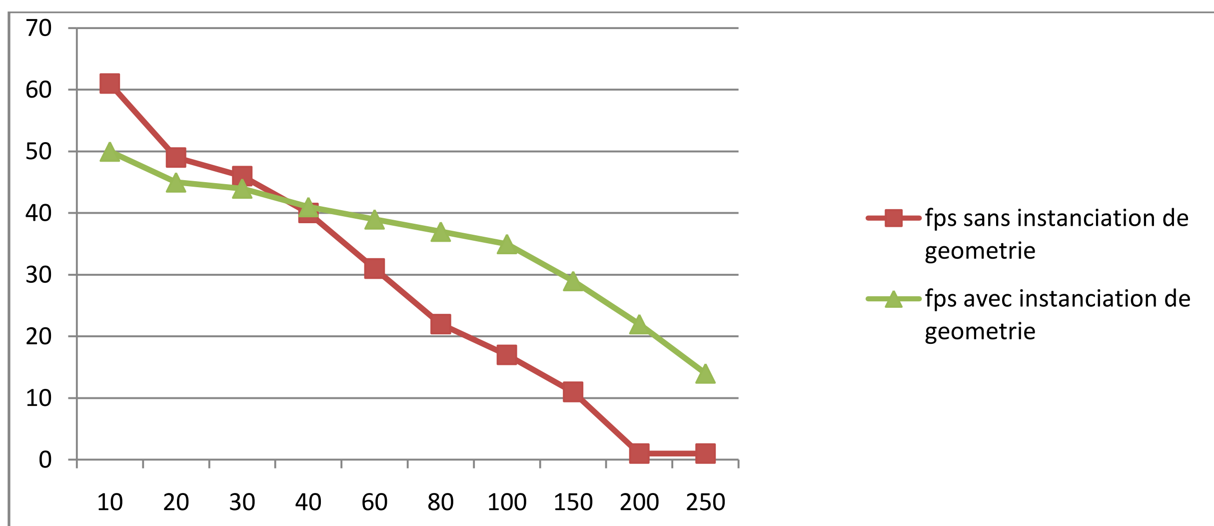


Fig87 : Performances en fps en fonction du nombre d'arbres

On peut constater dans ce graphe que la méthode d'instanciation de géométrie est moins performante quand le nombre d'arbres est petit, mais elle est moins sensible à l'augmentation du nombre d'arbres, et devient la plus efficace à partir d'un certaine nombre d'arbres.

L'utilisation de géométrie shader pour dupliquer les arbres dans la scène n'est pas très efficace pour rendre un petit nombre d'arbres par ce qu'on exécutant une boucle au niveau de le géométrie shader, on diminue la granularité de la tâche de rendu, et le degré de parallélisme affectant ainsi les performances front-end du GPU.

En augmentant le nombre d'arbres, le rendu va nécessiter plus de travail avec un rendu directe (sans instanciation), en utilisant le géométrie shader pour dupliquer les résultats de l'illumination et des transformations des sommets, cette charge du traitement va être diminuée à ne pas être exécutée qu'une seule fois par modèle, alors les performances sont moins affectées par l'augmentation de nombre d'arbres.

2.2. Traitement par modèle :

Pour tester l'efficacité du traitement par modèle, nous avons utilisée l'animation des arbres (effet du vent), et testé le dessin direct et l'instanciation de géométrie sans et avec animation.

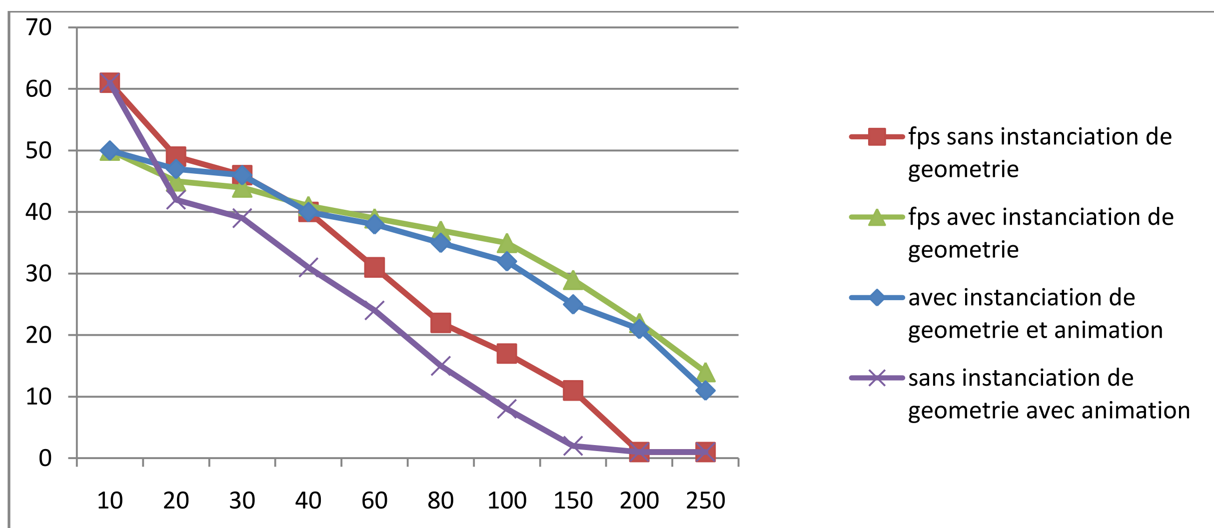


Fig88 : Impact de l'animation sur les performances

3. Conclusion :

La méthode d'instanciation de géométrie par géométrie shader n'est pas très efficace pour rendre un petit nombre d'arbres, mais elle est la moins sensible à l'augmentation du nombre d'arbres, et à l'augmentation de complexité des vertex shaders (comme l'animation, masque d'ombrage…), ce qui la rend très utile pour le rendu des arbres dans les scènes naturelles.

Elle représente aussi un avantage très important qui est sa stabilité en consommation du temps CPU et avec des valeurs réduites (n'utilise le CPU que pour les appels API du dessin), permettant ainsi l'exploitation de toutes les capacités du GPU.

3.1 Limitations de portabilité :

Etant fortement liée au GPU, la technique que nous avons présentée, ne fonctionne pas avec les même performances sur tout les GPUs, même avec des GPUs de la même gamme (même profile), les différances architecturales peuvent causer des traitements de compatibilités implémentés au niveau des drivers des cartes graphiques, qui peuvent dans le pire scenario arriver jusqu'à exécuter sur CPU certains taches non supporté par le GPU ou qui s'appliquent de manières différents sur ce dernier, puis transférer les résultats au GPU, se cas a un effet catastrophique sur les performances de l'application, car dans ce cas le traitement sur GPU va être suspendu, et les données sur lesquelles le traitement de compatibilité sera appliqué seront transmit vers le CPU, ou le pilote exécute les taches correspondantes, puis il retransmit les résultats vers le GPU, ce procédé va résulter en une perte du temps énorme et en chômage du GPU.

Une solution est d'envisager des reconfigurations au niveau de l'application en fonction des résultats des requêtes d'identification du GPU, et de ses capacités, mais une telle solution sera tres difficile a implémenter, par exemple dans le cas ou en veut optimiser les itérations en niveaux du géométrie shader, la requête GPU, peut déterminer le nombre maximal de sommets que peut produire le géométrie shader, mais ce résultats est n'est pas l'optimum, alors il faut déduire le nombre optimal en fonction de la complexité de la tache de production des sommets et le nombre de clusters de calcul présent sur le GPU, ce qui est tres difficile à automatisé, en plus le résultat de cette requête n'est pas toujours exacte par ce qu'il dépend de la taille des attributs des sommets et du cash présent pour les attributs des sommets, et si en essaye de produire plus de sommets que perméable par le GPU l'application crachera totalement.

3.2 : Perspectives pour l'interprétation en géométrie :

Dans ce travail nous avons essayer de présenter une représentation géométrique adapté au rendu par GPU, et compatible avec les techniques de rendu et de construction des végétaux qu'on a jugées les mieux adaptée a l'exploitation des GPUs, nous avons aussi présenter un mécanisme capable de générer cette représentation a partir des modèles de génération d'arbres existant, mais le résultats reste toujours moins satisfaisant que d'autres représentations qui ont gagnées beaucoup plus de réalisme et d'optimisation au cours du temps (par exemple Arbaro pour blinder, la bibliothèque Forest d'Ogre3d).

Cette représentation est impérative pour l'application de certains taches qu'on a présentées, mais il est possible d'envisager des prés traitements permettant la compatibilité avec d'autre représentations, comme le prés traitement sur des fichiers OBJ, il est possible de réarranger facilement les sommets pour obtenir l'ordonnancement des sommets qui est utile pour la simplification de maillage par élimination de surfaces, mais qui va nécessiter beaucoup de travail pour pouvoir déduire une description de l'arbre a partir de ces fichiers.

Ce problème de compatibilité ne se pose pas pour d'autres techniques comme le traitement par modèle et l'élimination des instances, qui peuvent être appliqués sur n'importe quelle représentation (géométrique ou impressionniste) avec géométrie shader adapté, mais les

performances ne vont pas être aussi importantes qu'avec notre représentation, alors cette pratique devient intéressante avec un grand nombre d'arbres et un traitement couteaux et redondant par arbre (un rendu probabiliste ou animation d'arbres par exemple), dans ce cas l'élimination des instances et la propagation des résultats de calcul sur un modèle sur ses instances vont êtres beaucoup plus adéquats que le recalcule sur chaque arbre.

3.3 : Plus d'implication de techniques issues des modèles impressionnistes :

La flexibilité offerte par les GPUs ressent peut permettre la combinaison a moindre cout de plusieurs méthodes rendu, nous avons essayer d'exploiter ce concept en générant a partir de la description du modèle des rectangles tournés face a l'observateur en gardant les normales initiaux au lieu des cylindres pour les branches et des groupes de feuilles ce qui n'est pas très convainquant en thermes de qualité visuelle, une alternative peut être l'application de textures avec des canaux alpha sur des plans générés pour remplacer les branches, ces textures seront compatibles avec les branches (ou l'ors de la génération de géométrie on respecte la cohérence avec les textures disponible), alors lors de l'application du niveau de détails correspondant a la génération de ces « imposteurs », le géométrie shader génère les plans et choisie la texture adéquate pour chaque branche.

Conclusion générale

Le rendu des scènes contenants des végétaux en temps réel a était toujours l'un des plus grand défis de la synthèse d'image, à cause du coût énorme en temps de calcul nécessaire pour traiter la grande quantité de géométrie exigée pour sa représentation.

Plusieurs techniques ont été présentées pour contourner ce problème et offrir des mécanismes permettant de maitriser le coût du traitement et générer des scènes avec des végétaux en temps réel dans les limites de performances matériel disponible, mais étant à base d'image ou plaquage de textures ces techniques souffrent de l'absence d'une représentation géométrique du végétal.

L'introduction des GPUs offrent plus de capacités du traitement et de flexibilité de programmation, permettant le traitement de plus de géométrie en temps réel, donnant ainsi la possibilité pour plus d'implication de géométrie dans les techniques de rendu des végétaux.

Dans ce travail nous avons essayé de présenter un procédé de rendu et une méthode de représentation qui permet une exploitation maximale des capacités des nouveaux GPU, et de leurs flexibilité, afin de pouvoir produire des scènes contenant une grande quantité d'arbres en temps réel avec la possibilité d'utilisation de la représentation géométrique complète si nécessaire, pour les vues rapprochées.

Cette technique de rendu est très adaptée au rendu de très grande quantité de géométrie qui forme les arbres, et a donnée dans les tests que nous avons effectué la plus basse sensibilité à l'augmentation du nombre d'arbres dans la scène, par rapport à plusieurs autres méthodes adapté au rendu de grandes quantités de géométrie, en exploitant les GPU, ces comparaisons ont étés effectués en utilisant de la géométrie pure avec le plus haut niveau de détails (sans niveaux de détails, sans clipping de géométrie, et sans niveaux de détails).

Pour maximiser l'exploitation du GPU, nous avons suivi une stratégie de minimisation de l'implication du CPU, dans un procéder d'instanciation de géométrie, cette stratégie a été déduite du phénomène de limitation des capacités des applications graphiques qui utilisent l'instanciation de géométrie non pas aux capacités du GPU mais a la capacité du CPU à fournir de la géométrie au GPU, alors plus le travail affecté au CPU durant le rendu et minimale plus l'exploitation du GPU est meilleure.

Cette stratégie est aussi très bénéfique en regardant la direction et l'ampleur du développement des GPUs qui deviennent de plus en plus performants, (leurs capacités de calculs deviennent beaucoup plus grandes que celle des transferts et d'alimentation en données disponibles sur les systèmes sur lesquels ils sont montés), et aussi leurs capacités de génération de géométrie qui sont de plus en plus importantes sur les GPUs les plus récents, cette dernière qui constitue un concept clé dans l'application de cette stratégie, en permettant

de compenser le transfert de la géométrie par la génération de nouveaux sommets localement sur le GPU.

Durant ces tests notre technique c'est avérée moins performante par rapport aux autres techniques qu'on a testé pour le rendu d'un nombre restreint d'arbres (spécialement dans le cas d'un seul arbre), ceci est due à la charge supplémentaire des attributs dédiés à la tâche de duplication des sommets et à la perte du temps sur les tests, mais avec l'augmentation du nombre d'arbres dans la scène, on peut constater les performances (FPS) augmentent offrant une très bonne stabilité pour le rendu de grandes quantités d'arbres.

Nous avons aussi présenté des mécanismes pour améliorer les performances de cette technique, qui sont : l'illimitation des arbres hors de la pyramide de vision avec un procédé très simple qui exploite la capacité du géométries hader a ne pas envoyer certains sommets au reste du pipeline, la simplification de maillage sans avoir à transférer ni de des données de sommets supplémentaires ni se nouveaux indexes, et la génération dynamique par géométrie shader des représentations approximatives(imposteurs) pour les arbres situés loin de la camera en appliquant une méthode très économique en temps de calcul est sans transfert de données supplémentaire pendant le rendu (inter frame data), ce qui donne des améliorations très significatives en termes de temps de calcul sans alterner la qualité visuelle de la scène, et d'une méthode de texturage basée sur le texturage séparé des cylindres et ellipsoïde formant les arbres.

Cette technique introduit le concept de traitement par modèle qui offre la possibilité d'appliquer directement des résultats d'un traitement effectué sur un seul modèle d'arbres sur toutes ces instances dans la scène permettant un gain très important en temps de calcule.

Le modèle de représentation et rendu que nous avons présenté sont complémentaires, le procédé de rendu exploit des pré-connaissances sur les unités de construction de la géométrie pour simplifier le contrôle de la génération et duplication des sommets, et la gestion des niveaux de détails, ce qui pose un problème de compatibilité avec les résultats des autres software de génération d'arbres, qui seront très intéressantes a utilisé pour des raisons de qualités des modèles, une solution possible à ce problème est de traiter les meshs qu'ils génèrent pour produire une description compatible avec notre technique.

Bibliographie

[1] B. Mandelbrot: *The fractal geometry of nature*.W.H.Freeman and company editor. 1982.

[3] A. Fournier, D. Fussell, et L. Carpenter : *Computer rendering of stochastic models*.Communications of the ACM.1982.

[4] P. Reffye, C. Edelin, J.Françon, M. Jaeger, et C.Puech: *Plant Models Faithful to Botanical Structure and Development*. Actes de SIGGRAPH '88, Atlanta, Août 1988.Computer Graphics, Vol. 22, n°4, 1988.

[5] N.Janey : *Modélisation et synthèse d'image d'arbres et bassins fluviaux combinant méthode combinatoires et plongement automatique d'arbres et cartes planaires.* thèse de doctorat à l'université de FRANCHE-COMTE .1992

[6] A. MEYER : *Représentations d'arbres réalistes et efficaces pour la synthèse d'images de paysages.* thèse de doctorat Université Grenoble 1 - Joseph Fourier. 1992

[7] F. KentonMusgrave, Craig E. Kolb, et Robert S. Mace : *The synthesis and rendering of eroded fractal terrains*. In SIGGRAPH 1989, Computer Graphics Proceedings.

[8] John P. Beale, Université de Stanford, Gforge landscape generator.

[9] R. P. Voss: *Fractal forgeries*. In R. A. Earnshaw: Fundamental Algorithms for Computer Graphics.Springer-Verlag, 1985.

[10] G. P. Miller: *The definition and rendering of terrain maps*. In David C. Evans and Russell J. Athay, editors, Computer Graphics (SIGGRAPH '86 Proceedings), volume 20, August 1986.

[11] C. Chaudy : *Modélisation et rendu d'images réalistes de paysages naturels*, thèse de doctorat université Joseph Fourier - Grenoble 1. 1997

[12] M. Aono et T. L. Kunii : *Botanical tree image generation*. IEEE Computer Graphics and Applications, 1984.

[13] P. E. Oppenheimer: *Real time design and animation of fractal plants and trees*. In David C. Evans and Russell J. Athay, editors, Computer Graphics(SIGGRAPH '86 Proceedings), volume 20, 1986.

[14] P. Prusinkiewicz et A. Lindenmayer: *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[15] P.Prusinkiewicz, M. James, and R. Mech: *Synthetic topiary*. In Andrew Glassner, editor, Proceedings of SIGGRAPH '94 (Orlando, Florida,1994), Computer Graphics Proceedings, Annual Conference Series. ACM SIGGRAPH, ACM Press, July 1994.

[16] B.Lintermann and O. Deussen: *Interactive modelling and animation of branching botanical structures*. Eurographics, 1996.

[17] X. Gérard Viennot, G. Eyrolles, N. Janey, and D. Arqués: *Combinatorial analysis of ramified patterns and computer imagery of trees*. Jeffrey Lane, editor, Computer Graphics (SIGGRAPH '89 Proceedings), volume 23, 1989.

[18] J. Weber and J. Penn: *Creation and rendering of realistic trees*. In Robert Cook, editor, SIGGRAPH 95 Conference Proceedings, Annual Conference Series, ACM SIGGRAPH, Addison Wesley, 1995.

[19] G.Y. Gardner: *Simulation of natural scenes using textured quadric surfaces*. In Computer Graphics (SIGGRAPH '84 Proceedings), volume 18, 1984.

[20] X. Gérard Viennot, G. Eyrolles, N. Janey, and D. Arqués: *Combinatorial analysis of ramified patterns and computer imagery of trees*. Jeffrey Lane, editor, Computer Graphics (SIGGRAPH '89 Proceedings), volume 23, 1989.

[21] N. Greene: *Voxel space automata*: *Modeling with stochastic growth processes in voxel space*, Computer Graphics (SIGGRAPH '89 Proceedings), volume 23,1989.

[22] W. T. Reeves: *Particle systems, a technique for modeling a class of fuzzy objects*. ACM Trans. Graphics, 1983.

[23] W. T. Reeves and R. Blau: *Approximate and probabilistic algorithms for shading and rendering structured particle systems*. In B. A. Barsky, Computer Graphics (SIGGRAPH '85 Proceedings), volume 19, 1985.

[24] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross: Surfels *:Surface elements asrendering primitives*. In SIGGRAPH 2000, Computer Graphics Proceedings, 2000.

[25] S. Rusinkiewicz and M. Levoy: *QSplat : A multiresolution point rendering system for large meshes.* In SIGGRAPH 2000, Computer Graphics Proceedings, 2000.

[26] M. Stamminger and G. Drettakis: *Interactive sampling and rendering for complex and procedural geometry.* In Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering 01).Eurographics, 2001.

[27] T. Nishita, Y.Dobashi, and E.Nakamae: *Display of clouds taking into account multiple anisotropic scattering and sky light.* In Computer Graphics (SIGGRAPH '96 Proceedings), 1996.

[28] G. Y. Gardner: *Simulation of natural scenes using textured quadric surfaces*. In Hank Christiansen, editor, Computer Graphics (SIGGRAPH '84Proceedings), volume 18, 1984.

[29] T. Nishita, Y. Dobashi, K.Kaneda, and H.Yamashita: *Display method of the sky color taking into account multiple scattering*. In Pacific Graphics '96, pages 1996.

[30] J. F. Blinn. Models of light reflection for computer synthesized pictures. In SIGGRAPH 1977, Computer Graphics Proceedings, pages 192–198, 1977.

[31] L.Williams: *Pyramidal parametrics*. In SIGGRAPH 1983, Computer Graphics Proceedings, July 1983.

[32] P. Poulin and A. Fournier: *A model for anisotropic reflection*. In SIGGRAPH 1990,Computer Graphics Proceedings, August 1990.

[33] K.J. Dana, B. van Ginneken, S.K. Nayar, and J.J. Koenderink: *Reflectance andtexture of real-world surfaces*. ACM Transactions on Graphics, 1999.

[34] L. McMillan and G. Bishop: *Plenopticmodeling : An image-based rendering system*. In SIGGRAPH 1995, Computer Graphics Proceedings, 1995.

[35] M. Levoy and P. Hanrahan: *Light field rendering*. In SIGGRAPH 1996, ComputerGraphics Proceedings, August 1996.

[36] J. Shade, S.J. Gortler, L. He, and R. Szeliski: *Layered depth images*. In SIGGRAPH1998, Computer Graphics Proceedings, July 1998.

[37] N. Max, O. Deussen, and B. Keating: *Hierarchical image-based rendering using texture mapping hardware*. In Eurographics Workshop on Rendering, 1999.

[38] J. T. Kajiya and T. L. Kay: *Rendering fur with three dimensional textures*. In SIGGRAPH1989, Computer Graphics Proceedings, July 1989.

[39] T. Noma: *Bridging between surface rendering and volume rendering for multiresolution display*. In Eurographics Workshop on Rendering 1995, June 1995.

[40] P. Lacroute and M. Levoy: *Fast volume rendering using a shear-warp factorization of the viewing transformation*. In SIGGRAPH 1994, Computer Graphics Proceedings, July 1994.

[41] A. Meyer and F. Neyret. Interactive volumetric textures.In Eurographics Workshop on Rendering 1998, June 1998.

[42] P. de Reffye, C. Edelin, J. Francon, M. Jaeger, and C. Puech : *Plant models faithful to botanical structure and development*. In JohnDill, editor, Computer Graphics (SIGGRAPH '88 Proceedings), volume 22 , August 1988.

[43] R. Toledo : *Visualisation Interactive de Modèles Complexes avec les Cartes Graphiques Programmables*, thèse de doctorat de l'université Henri Poincaré – Nancy1,2007

[44] B. Jobard : *Evolution des cartes pipeline graphiques* ,2005.

[45] G. Coombe, M. J. Harris, and A. Lastra: *Radiosity on graphics hardware.* In Graphics Interface (to appeared). CIPS, Canadian Human-Computer Commnication Society, 2004.

[46] T. J. Purcell, C. Donner, M. Cammarano, H.Wann Jensen, and P. Hanrahan: *Photon mapping on programmable graphics hardware*. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Eurographics Association, 2003.

[47] J. Krueger and R. Westermann: *Acceleration techniques for gpu-based volume rendering*. In Proceedings IEEE Visualization 2003.

[49]J. Kessenich, D. Baldwin, R. Rost: *The OpenGL® Shading LanguageVersion: 4.00,Document Revision: 8-10 Mar 2010*, Editor: John Kessenich, Intel, Version 1.1, 2010.

[50] Sylvain Collange, DALI, ELIAUS, « Analyse de l'architecture GPU Tesla», Université de Perpignan. janvier 2010.

[51] V. Forest : *Introduction à la programmation des GPU*, séminaire GPU Vortex 2007

[52] J. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym: *NVIDIATesla : A unified graphics and computing architecture*. IEEE Micro, 28(2) :39–55, 2008.

[54] GPU Gems 2: *Programming Techniques for High-Performance Graphics and General-Purpose Computation*,Chapter 3. Inside Geometry Instancing.

[55].W., Matthias. 2003: *Batch, Batch, Batch: What Does It Really Mean.* Presentation at Game Developers Conference 2003.

[56] NVDIA Technical Report :*GLSL Pseudo-Instancing*, **2004**

[57] J. Kessenich, D. Baldwin, R. Rost *: The OpenGL® Shading Language Language Version: 4.00*, Document Revision: 8, 10-Mar-2010, Intel, Version 1.1

Sites webs :

[2] www.syti.net/Fractals.html

[48] www. developeur.nvidia.com

[53] www.nvidia.com/cuda