Université de Strasbourg

UFR de mathématique et d'informatique

Master 1 Image et 3D, 2024

Géométrie Numérique
Génération d'arbre procédurale en 3D

Professeurs:

VIVILLE Paul, RAVAGLIA Joris

Projet de:

TEISSANDIER Alban, BONNEAU Audric, GOETZ Arnaud,
FRANZ Axel, JOUFFROY Pierre

Date de remise: 06/01/2025

Contents

1 Introduction

Les modèles 3D d'arbre peuvent être très lourds, en particulier dans les jeux vidéo ou les simulations.

Afin de concilier réalisme visuel et contraintes de performance, il faut donc essayer de trouver une méthode qui nous permettent d'avoir un rendu réaliste d'arbre personnalisable et reproductible.

Ce rapport fait l'état de notre application proposée pour résoudre ce problème. Notre solution devra répondre à deux contraintes :

1. Méthodes rapides. Temps d'exécution temps réel.
2. Procédurales. À chaque lancement du programme avec différent paramètre, l'arbre généré devra être unique.
3. Le maillage généré devra avoir une topologie correcte, manifold.

2 Description technique

Cette partie fait l'état des choix d'implémentation et des méthodes utilisées pour répondre à la problématique de construction de l'arbre et des contraintes posées. Nous verrons en premier temps comment nous avons générer un nuage de point à l'aide du *super-formule* 2.1. Dans un second temps, nous verrons le choix de rajouter une structure accélératrice au paquet, le tri par paquet 2.2. Ensuite nous verrons l'algorithme d'exploration de colonisation d'espace 2.3 puis la transformation en maillage manifold 2.4. Pour terminer, nous verrons l'affichage et le rendu de l'application 2.5.

2.1 Nuage de point

Il a été décidé de rependre l'algorithme de colonisation de l'espace (Ratul, 2019 [2]). Cet algorithme est une généralisation des *super-ellipses* proposée par Johan Gielis (voir [1]) qui permet de modéliser une énorme variété de formes (fleurs, étoiles, sphères, etc.).

On peut utiliser l'équation connue comme *Superformula* pour générer pour générer des formes courbes lisses en deux dimensions

$$r(\phi) = \left(\left| \frac{\cos\left(\frac{m\phi}{4}\right)}{a} \right|^{n_2} + \left| \frac{\sin\left(\frac{m\phi}{4}\right)}{b} \right|^{n_3} \right)^{-\frac{1}{n_1}}$$

Cette équation peut être étendue à la 3D en faisant

$$\begin{cases} x = r_1(\theta)\cos(\theta) \cdot r_2(\phi)\cos(\phi) \\ y = r_1(\theta)\sin(\theta) \cdot r_2(\phi)\cos(\phi) \\ z = r_2(\phi)\sin(\phi) \end{cases}$$

L'équation contient plusieurs paramètres :

- a - Correspond à l'échelle horizontale, permet étirer ou comprimer la forme sur l'axe X
- b - Correspond à l'échelle verticale, permet étirer ou comprimer la forme sur l'axe Y
- m - Nombre de lobes ou symétries, plus m est grand, plus il y a de répétitions
- n1 - Contrôle général de la forme, permet de modifier la rondeur global
- n2 - Exposant pour cos() dans la formule
- n3 - Exposant pour sin() dans la formule

L'algorithme a été un peu changé pour essayer d'avoir une distribution plus normale (aléatoire) des points :

```
1    std::normal_distribution<float> dist_phi(mV, spread);
2    float phi = dist_phi(generator);
```

- mV est la moyenne (centre de la distribution)

- spread est l'écart type de la distribution normale (controle à quel point ce disperse la distribution)

Et en faisant maintenant

$$\begin{cases} x = r_1(\theta)\cos(\theta) \cdot r_2(\phi)\cos(phi) \\ y = r_1(\theta)\sin(\theta) \cdot r_2(\phi)\cos(phi) \\ z = r_2(\phi)\sin(phi) \end{cases}$$

Finalement, la superformule générant uniquement des points sur les bords, nous multiplions tous les points par une valeur aléatoire entre 0 et 1 pour avoir aussi des points à l'intérieur (et donc un volume de point et non juste une surface).

2.2 Tri par paquets

L'exploration du nuage de point (nous le verrons dans la partie suivante 2.3) a besoin d'une structure permettant de trouver les points le plus proche d'un autre, selon une distance de recherche fixe. Pour respecter la condition d'exécution en temps rapide 1, nous avons implémenté un *bucket sort* ou tri par paquets en structure accélératrice. Un tri par paquet consiste à subdiviser l'espace en paquet, donc en 3 dimension des cubes ou des rectangles si la subdivision n'est pas de même taille sur chaque axe. La taille de la subdivision dépend de :

- La taille du nuage de point en entrée pour la taille de la grille.
- Le rayon de recherche pour la taille des paquets.

Une classe *BucketSort* est chargé de contenir l'ensemble de la définition et implémentation des méthodes nécessaires. A l'initiation, un tri obligatoire est fait pour ranger chaque point du nuage de point dans un paquet. L'accélération offerte par la structure consiste en la recherche de points voisins. En effet, si je suis un point dans le paquet i, je sais que le voisin le plus proche sera forcément soit dans le même paquet ou dans les paquets alentour à distance 1 (donc i-1 à i+1 pour chaque axe,

ce qui fait 27 paquets candidats en 3D). Cela permet donc d'éviter de parcourir l'ensemble des points et de calculer leur distance au point testé et de se cantonner à un ensemble de points candidats.

Notre implémentation permet aussi de :

- Rajouter un point dans un paquet en cours de route.
- Récupérer l'ensemble des points qui sont le plus proche d'un point, c'est à dire l'ensemble des points contenus dans un paquet et ceux environnants à distance 1.
- Récupérer le point le plus proche sans se restreindre à la couronne de paquet environnant de 1 et de s'étendre à la couronne i+1 si rien n'a été trouvé dans les paquets précédents.

2.3 Algorithme d'exploration

Pour créer notre arbre, nous sommes partis sur l'algorithme d'exploration de l'espace [2]. Cet algorithme nécessite une enveloppe et des points qu'on appelle points d'attractions. Ce sont des points qui indiquent que l'espace autour de ce point n'est pas encore exploré. Nous utilisons le nuage de points créé par la *super-formule* dans la section 2.1. Ce nuage de points constitue donc l'enveloppe de notre parcours. Les points d'attractions sont donc situés soit sur l'enveloppe soit à l'intérieur. Ces points ont deux paramètres :

- Une distance d'élimination : Si la distance entre un noeud et le point est inférieure à ce paramètre, le point d'attraction est supprimé. Cela signifie que l'espace a bien été occupé par une branche.
- Une distance d'influence : Si la distance entre un nœud et le point est inférieure à ce paramètre, le nœud est attiré par ce point.

L'idée est de parcourir notre espace défini par nos points d'attractions et créer une structure d'arbre composé de pères et de fils. L'arbre est construit itérativement en ajoutant un nœud à un

père à une distance connue (distance de croissance).

On défini les nœuds actifs comme étant les nœuds pouvant avoir des fils dans l'arbre et les nœuds passifs comme étant des nœuds à ne plus parcourir car ils n'auront plus d'impact dans la création de l'arbre.Toutes les recherches de points d'attraction et de nœuds utilisées sont accélérées avec tri par paquets 2.2.

La première étape permet de constituer le tronc. On part d'un nœud de départ, la racine de notre arbre. Tant qu'un nœud n'a pas trouvé de points d'attraction, c'est à dire qu'il ne se trouve pas dans une distance d'influence, on ajoute un nouveau nœud à une distance *growDistance* en Z.

On réalise les étapes suivantes :

- L'algorithme de départ proposé [2] choisit de parcourir tous les points d'attractions et de trouver son nœud le plus proche dans sa distance d'influence. Ainsi, si nous avons 1000 points d'attractions, le temps de calcul diminuera au cours du temps au fur et à mesure que ces points sont supprimés mais cela reste coûteux en terme de parcours. Nous avons trouvé une solution plus rapide avec la notion de nœuds actifs détaillés plus tôt. Nous parcourons tout les nœuds actifs et bien que notre temps de parcours augmente en fonction du temps, comme nous supprimons les nœuds passifs du parcours, nous parcourons moins de points.

  Pour chaque nœud, on récupère ses points d'attraction. Si il n'y en a pas, nous le marquons comme passif, il n'aura plus de fils. Sinon, on sélectionne uniquement les points d'attraction dont le nœud en cours est le plus proche. On récupère le vecteur d'attraction moyen et on crée un nouveau nœud d'une distance *growDistance* dans la direction du vecteur. Ce nœud est ajouté à l'arbre à la fin de l'itération en cours. Si ce prochain nouveau nœud est à une distance d'élimination d'un point d'attraction, alors le marquer comme à supprimer.

- Mettre à jour l'arbre avec les nouveaux nœuds

- Supprimer les points d'attractions marqués

- Supprimer les nœuds passifs des nœuds à parcourir

Recommencer soit un certain nombre de fois, soit jusqu'à ce que l'arbre n'ait pas été modifié lors d'une itération.

2.4 Transformation volumique

La construction du maillage prend en entrée les points de constructions générés par l'algorithme de colonisation d'espace 2.3. La construction se fait en deux parties :

1. Construction d'un cylindre au niveau de chaque branche.

2. Construction de la boîte englobante des bases des cylindres autour de l'intersection entre deux ou plusieurs branches.

Ces deux parties reposent sur une structure de nœuds de constructions possédant une référence vers son nœud père et ses nœuds fils (structure d'arbre). Chaque nœud père possède une référence vers les sommets des cylindres qu'il crée à X% (pourcentage en paramètre) du vecteur entre lui et ses fils. Ainsi, un nœud père voulant créer l'enveloppe convexe va pouvoir accéder aux sommets des cylindres qu'il possède. Pour la création des cylindres d'un nœud père vers un de ses fils, le père doit récupérer les sommets de constructions du cylindre vers son fils qu'il possède, et accéder à son fils pour récupérer les sommets de constructions du cylindre vers lui-même, que possède son fils.

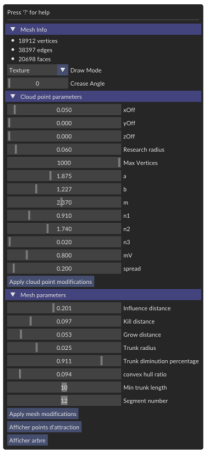
L'algorithme de construction de l'enveloppe convexe est géré par la librairie *CGAL* et la structure point est gérée par la librairie *PMP*. Le reste dont la structure père fils et des cylindres sont faits à partir de zéro.

2.5 Affichage et rendu (UI)

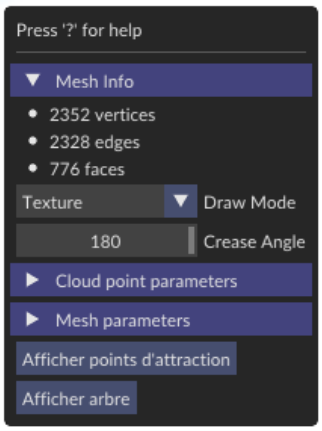
Nous avons décidé de partir sur la base *ImGUI* présente dans les viewers de *PMP*. Cette base a ensuite été enrichie par deux menus déroulants permettant de modifier les paramètres des algorithmes décrits

ultérieurement.

- Le premier menu déroulant permet de modifier les paramètres de génération du nuage de points.
- Le deuxième menu déroulant permet de modifier les paramètres de génération du maillage découlant du nuage de points.



(a) Menu ouvert



(b) Menu fermé

Figure 1: Menu ouvert vs fermé

Chaque menu déroulant est indépendant et changer les paramètres du maillage ne va pas influer sur le nuage de point mais changer le nuage de point va automatiquement recalculer le maillage.

Il y a aussi 2 modes de visualisation possibles grâce à des boutons permettant d'afficher le nuage de points d'attraction ou le maillage.

3 Organisation et choix d'implémentation

3.1 Organisation des fichiers

L'organisation des fichiers est la suivante :

- **point_cloud.hpp, point_cloud.cpp** : Fonctions de la *super-formule* et de la création du nuage de point. Dépend de **vertex.hpp** définissant la structure de point commune avec le tri par paquets.
- **BucketSort.hpp, BucketSort.cpp** : Classe du tri par paquets. Dépend de **vertex.hpp** définissant la structure de point commune avec le nuage de point.
- **exploration.hpp, exploration.cpp** : Fonctions de l'exploration par colonisation d'espace et maillage à partir de points de constructions.
- **Interface.hpp, Interface.cpp** : Le visualiseur *ImGUI*.
- **main.cpp** : L'appel vers les différentes fonctions et méthodes pour la chaîne de construction de l'algorithme complet.

3.2 Documentation

La documentation est générée par Doxygen à l'aide du fichier shell *doc.sh*.

3.3 Choix d'implémentation

3.3.1 Tri par paquets

Nous avons débattu sur l'utilisation de *unordered set* ou ensemble non ordonnée. Ce conteneur c++ permet d'utiliser une table de hachage qui pourrait être en théorie plus rapide qu'un tri par paquets. Seulement, comme montre cette article, il est nécessaire de trouver une bonne fonction de hashage ce qui peut être couteux. Donc pour éviter ce côut en plus, nous avons décidé de garder l'idée d'un tri par paquets.

3.3.2 Exploration

Nous avons choisi l'algorithme d'exploration car les résultats présentées dans les papiers donnaient deux avantages par rapport aux L-systèmes. Premièrement, les arbres générés sont plus "réalistes" si on choisit nos paramètres avec précaution. Ensuite, ce "coût" de réalisme ne se répercute pas de façon trop importante sur les performances d'autant plus qu'on souhaite faire de la génération en temps réel.

Un choix d'implémentation, lié à la partie tri par paquets 3.3.1, était la forme de la recherche de point d'attraction. Dans l'état final de l'algorithme, la recherche se fait en forme de sphère de la taille du rayon de recherche. Pendant le développement, il a été testé d'implémenté une recherche en forme de cône de 60°. L'idée était de projeter le vecteur du point p au point potentiel d'attraction sur le vecteur entre le point p et le sommet du cône. En sachant que p est le point dont nous recherchons ses points d'attractions et donc est aussi le sommet du cône. Il suffisait ensuite de calculer les angles pour vérifier si on était dans le cône. Cette implémentation permettait de supprimer les demi tours de branches. Cependant les temps mesurés en moyenne sur 10 itérations était de 21 secondes ce qui est beaucoup plus long que le temps d'exécution avec une recherche en forme de sphère. Nous n'avons donc pas continué sur une piste avec une recherche en forme de cône et gardé celle avec une forme de sphère.

3.3.3 Maillage

Nous avons choisi de générer le maillage de cette manière (cylindres + boîtes englobantes) car les articles soit n'explicitent pas leur approche, soit elle est trop complexe pour être mise en place dans le cadre de ce projet. Après discussion avec l'enseignant de TP, nous avons conclu de cette approche. Pour la mise en place, lorsque l'on tente de tout relier en une seule composante connexe, cela crée des configurations non manifold dans le maillage et mène à des faces manquantes. Nous avons donc choisi de séparer les cylindres et les boîtes englobantes sur des sommets distincts afin qu'il n'y ait pas de problèmes de connexité. Cela permet un beau rendu visuel mais cache le fait que le maillage

est donc composé de pleins de composantes connexes différentes assemblées en un seul maillage.

Figure 2: Arbre avec uniquement les cylindres

Figure 3: Arbre avec uniquement boîtes englobantes

4 Résultats

Nous constatons que notre algorithme de colonisation de l'espace fonctionne bien, et que la structure de l'arbre est correcte. Tout est modifiable en temps réel grâce au viewer intégrée. Pour la génération du maillage, cela pose plus de problèmes pour avoir quelque chose de propre et manifold. Nous arrivons à générer des arbres soit avec des faces manquantes, soit avec plusieurs composantes connexes, ... Visuellement, cela fonctionne bien mais le maillage n'est pas exploitable si le besoin est qu'il soit manifold. La solution proposée ne permet donc pas dans l'état de répondre à la contrainte manifold 3.

Figure 4: Comparaison des trois arbres générés

Concernant le temps d'exécution, on peut se référer au tableau 1 qui démontre que la solution est exécutable rapidement sur un nombre d'exécution avec des paramètres standards. On peut retenir en effet que le temps d'exécution moyen sur 10 itérations est de **685 microsecondes** soit 0.7 secondes si on arrondit. Donc la contrainte 1 est respectée.

Essai n°	Temps d'exécution (en ms)
1	680
2	680
3	691
4	674
5	680
6	684
7	697
8	625
9	705
10	735
Moyenne pondérée	685

Table 1: Mesure de temps d'exécution en microsecondes (ms)

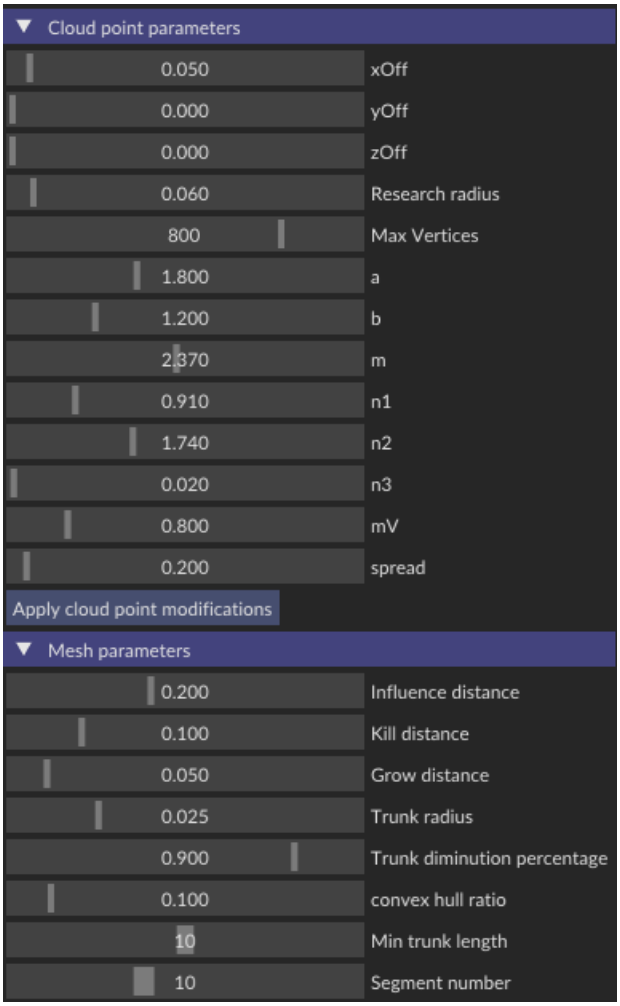


Figure 5: Les paramètres pour le calcul de temps d'exécution

Concernant la contrainte d'arbre unique à chaque itération 2, cela est respectée du à la génération aléatoire des points du nuage (selon une graine d'aléatoire). La contrainte est donc respectée.

5 Discussion

5.1 Avantages et inconvénients des algorithmes

	Colonisation de l'espace	maillage 3D
Avantages	Rendu plus réaliste que d'autres approches comme les L-systems / Paramétrable et rapide	Manifold en théorie, rapide
Inconvénients	Difficile de trouver les bons paramètres pour la superformule et pour les grow/kill/influence distances	Difficile d'avoir un maillage manifold en pratique

Table 2: Tableau des avantages/inconvénients

5.2 Ressenti personnel

5.2.1 Alban

J'ai trouvé que ce projet est assez dur à prendre en main du fait de notre inexpérience en C++ et de la complexité de la tâche. En tout, beaucoup des difficultés ont été en rapport avec certaines fonctionnalités du C++, ou de CMake, mais pas tant que ça sur l'algorithme de colonisation de l'espace par exemple. La génération du maillage est très intéressante, mais il est difficile de trouver par nous-mêmes des façons de faire ou d'optimiser ce que nous avons.

5.2.2 Arnaud

Le projet était à réaliser en parallèle avec les séances de TP et donc a mis du temps à se mettre en place. Implémenter concrètement "à notre sauce" un algorithme décrit dans un papier scientifique était très intéressant d'autant plus qu'on a essayé de le modifier et l'adapter à nos structures de données.

5.2.3 Audric

Le projet était intéressant et le fait de laisser la liberté d'implémentation était une bonne chose, mais possiblement à double tranchant (dans tous les cas, l'idée derrière est pertinente). Commencer le projet et comprendre ce qu'on devait faire concrètement a été difficile car *PMP* était assez obscur en première approche. Cela a causé un retard sur le fait d'implémenter notre solution et donc des soucis sont apparus tard dans le processus de développement.

5.2.4 Axel

J'ai trouvé ce projet très déroutant au début. Très peu d'indications étaient données quant à la marche à suivre à part quelques articles de recherche avec des algorithmes que l'on pourrait implémenter. J'ai passé beaucoup de temps à régler des erreurs de compilation car *CMake* et par extension le *C++* a une très mauvaise gestion des librairies externes, c'était un assez gros frein. Je trouve aussi avoir moins travaillé que les autres sur la parite code pure car tout ce que j'ai fait après la compilation (l'UI et modification visuelle des fonctions) se basait sur leur travail et vu la haute qualité de celui-ci, j'avais très peu de débug et de réflexion à faire (en dehors des heures passer à lire des messages d'erreur *CMake*).

5.2.5 Pierre

Je pense que les algorithmes choisis étaient cohérent en terme de rapidité, une des difficultés a été de commencer le projet en début d'année (lire les articles, trouver les bonnes ressources (librairies), utiliser la librairie *PMP* à plusieurs, ...) et de se bien répartir les tâches parmi les membres du groupe (qui fait quoi). Une fois que c'était fait, le projet est devenu plus fluide et avançait avec régularité. De plus, plus nous avancions, plus voir les résultats devenait ludiques et amusant.

6 Conclusion

Nous avons donc réussi une application permettant de générer un maillage d'arbre entièrement procéduralement, personnalisable et généré rapidement en utilisant un nuage de points et un algorithme d'exploration par colonisation d'espace ainsi que notre méthode pour le maillage. L'arbre générée est visuellement *propre* mais n'est pas manifold. Il est donc dans certains cas d'utilisations comme de la visualisation suffisant.

Pour aller plus loin, il serait donc crucial de se pencher sur un algorithme de maillage manifold et aussi de trouver une manière d'éviter les demi-tours des branches lors de l'exploration du nuage de points par l'algorithme de colonisation d'espace, plus rapide que la recherche en forme de cône proposée, qui n'a pas été concluante.

Ce projet nous a permis d'en apprendre plus sur la génération procédurale, les algorithmes utilisés pour les maillages 3D et le travail sous forme de projet agile.

7 Bibliographie

References

[1] Johan Gielis. A generic geometric transformation that unifies a wide range of natural and abstract shapes. *American Journal of Botany*, 90(3):333–338, 2003.

[2] Rafsan Ratul, Shaheena Sultana, Jarin Tasnim, and Arifinur Rahman. Applicability of space colonization algorithm for real time tree generation. In *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, pages 1–6. IEEE, 2019.