

Dynamic Graph : un outil générique pour la modélisation multi-échelle

Frank Perbet

Laboratoire GRAVIR / Equipe EVASION

frank.perbet@imag.fr

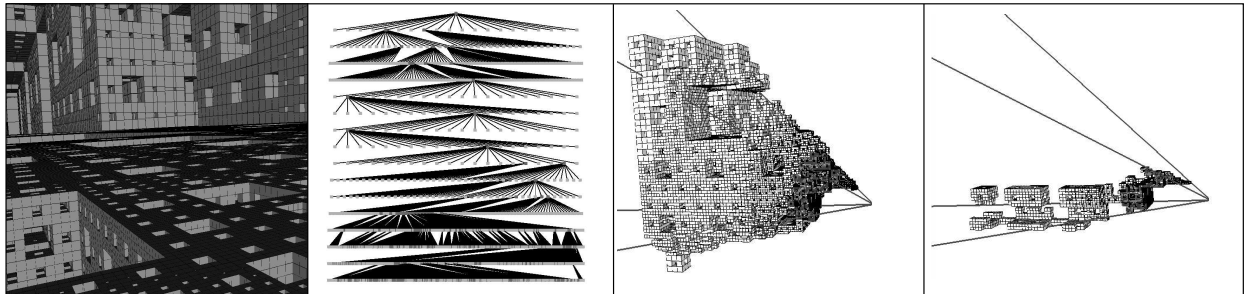


FIG. 1 – De gauche à droite : le cube de Sierpinski à 2 images par seconde sur un PC à 2.4 Ghz, le graphe d'évaluation de la première image (14 subdivisions), le rendu sans et avec l'algorithme de détection d'occlusion.

Résumé : *Modéliser et afficher des scènes animées avec de grandes variations d'échelle est une tâche très coûteuse et aujourd'hui impossible à réaliser en temps-réel. Nous montrons que seule la modélisation procédurale multi-échelle est apte à résoudre un tel problème. Une analyse des quelques travaux existants permet d'une part de constater l'efficacité de cette approche, et d'autre part de mettre en évidence l'un de ses plus gros défauts : la quantité de code à écrire pour créer chaque nouveau modèle. C'est là notre principale contribution : nous présentons Dynamic Graph (DG), le premier outil générique pour la modélisation procédurale multi-échelle. Autour d'un noyau basé sur des graphes acycliques dynamiques, de nombreuses fonctionnalités allègent le processus de création sans pour autant nuire à son expressivité.*

Mots-clés : Modélisation, multi-échelle, méthode procédurale, animation, temps-réel, graphes acycliques dynamiques, cohérence temporelle, précision, visibilité, navigation, fractale

1 Introduction

Lorsqu'une forme 3D est mal échantillonnée, de sérieux problèmes détériorent son affichage. Sous-échantillonnée, elle est trop grossière et presque méconnaissable. Sur-échantillonnée, les coûts mémoires et le temps de calcul des algorithmes de rendu augmentent tandis que le fameux phénomène d'aliasing "parasite" inexorablement l'affichage. La lutte pour un échantillonnage correct est présente tout le long du processus de création (fig. 2). Les méthodes d'*anti-aliasing* agissent en fin de cycle, au niveau du rendu (fig. 2). L'*animation multi-échelle* traite les problèmes d'échantillonnage temporels. Le contexte de ce travail est la *modélisation multi-échelle*. Cette approche se propose de résoudre les problèmes d'échantillonnage très tôt dans le processus de création, dans la phase de création de l'objet. Dans cette section, nous précisons le contexte scientifique de la modélisation multi-échelle puis nous présentons les contributions et la structure de cet article.

1.1 Contexte scientifique

Le terme *échelle* se réfère à l'ordre de grandeur d'un signal. Le terme *multi-échelle* implique des variations de cet ordre de grandeur. Plus ces variations sont grandes, plus il est délicat d'empêcher les problèmes d'échantillonnage de nuire au traitement de ce signal. La motivation de notre travail est de pouvoir observer des formes tridimensionnelles de près ou de loin sans que cela n'influence sur le temps de calcul ou la qualité du rendu. Une particularité de notre approche consiste à ne poser a priori *aucune limitation sur le zoom*, ce qui implique des variations d'échelles potentiellement infinies.

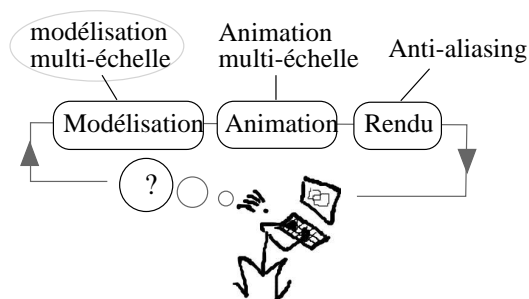


FIG. 2 – Le processus de création est un cycle : le créateur imagine un objet, le modélise avec des fonctions qui vont reproduire sa forme et son mouvement. Il se rapproche itérativement de son idée en la comparant au résultat et modifiant le modèle en conséquent.

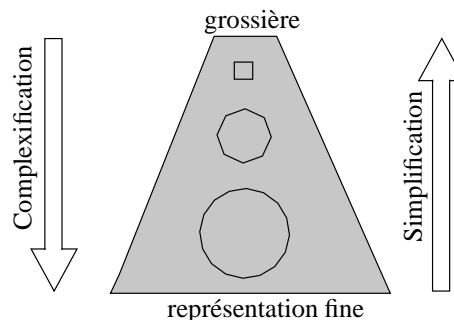


FIG. 3 – A l’inverse de la simplification, la complexification part d’une forme grossière et la raffine. Elle ne peut être réalisée qu’avec une modélisation procédurale.

La *modélisation multi-échelle* (fig. 2) se propose de remédier aux problèmes d’échantillonnage dès la phase de modélisation. La méthode généralement utilisée consiste à partir d’un objet à sa précision la plus grande pour le *simplifier* ultérieurement. Plus rarement, la précision est prise en compte dès la création de l’objet. Ce dernier est alors considéré comme une forme simple à laquelle est appliquée une série de fonctions qui vont la *complexifier*. L’algorithme présenté ici appartient à cette catégorie : les fonctions qui enrichissent la forme durant la phase de modélisation sont stockées et triées de la plus grossière à la plus fine puis rejouées lors de l’affichage jusqu’à ce que la précision soit suffisante.

Tous les grands modélisateurs sont munis de structures définissant l’ordre d’évaluation des fonctions appliquées durant la phase de modélisation. L’objet est alors décrit *procéduralement*[EMP⁺98] comme une fonction compliquée qu’il est possible d’évaluer pour engendrer une géométrie. Inversement, les descriptions les plus souvent utilisées à l’extérieur de ces modélisateurs sont simples et massivement discrétisés (comme la fameuse soupe de polygones). Ici, deux définitions s’imposent :

Description discrétisée : description d’un signal par un ensemble de valeur pré-évaluées.

Description procédurale : description d’un signal par un langage complexe dont l’interprétation est plus coûteuse mais plus flexible qu’une description discrétisée.

1.2 Contribution et plan

Après l’état de l’art, nous présentons DG, un nouvel outil de modélisation procédurale multi-échelle par complexification (fig. 3). Son but est d’assister la génération de scènes complexes et animées pouvant supporter de grande variation d’échelle. Il utilise un nouveau langage procédurale permet à l’utilisateur de décrire la scène comme un ensemble de fonction pouvant s’appeler entre elles selon un ordre déterminé. Une description de ce nouveau langage se trouve dans la section 3.

Pour finir, dans la section 4, les principales fonctionnalités de DG facilitant la modélisation sont rapidement décrites et quelques résultats sont montrés. Nous terminerons par une discussion générale sur l’outil présenté.

2 Etat de l’art

Les travaux de modélisation multi-échelle peuvent être classés en deux approches opposés : la simplification et la complexification. nous montrons pourquoi les méthodes de simplification ne permettent pas la modélisation avec de grande variation d’échelle. En revanche, les fractales, exemples typiques de complexification, s’y prêtent beaucoup mieux. Une vue d’ensemble des travaux traitant de modèles particuliers termine cette section.

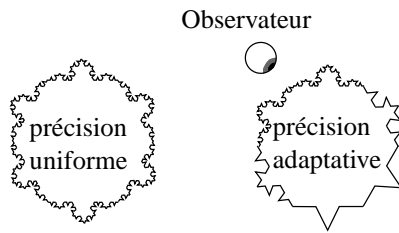


FIG. 4 – Critères d’arrêt uniforme et adaptatif dans le calcul récursif d’une fractale.

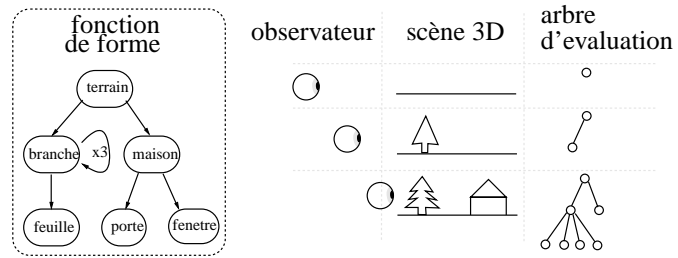


FIG. 5 – La *fonction de forme* est une représentation procédurale de la scène 3D. Elle est constituée de sous-fonctions pouvant s’appeler entre elles et dont l’ordre d’évaluation est déterminé. Lorsque d’une observation, ces fonctions sont calculées (et le graphe d’évaluation grandi) de façon à être suffisamment précise.

2.1 Simplification

De nombreux travaux prennent en entrée un modèle précis et produisent une hiérarchie de modèles simplifiés appelés aussi *niveaux de détail*. Lors de l’affichage, le niveau adéquat est choisi.

Le maillage est la représentation la plus communément utilisée. Les travaux sur la simplification de maillage sont recensés dans [LRC⁺02]. La plupart des algorithmes produisent automatiquement des représentations avec plus ou moins de polygones. Quelquefois, des transitions continues sont possibles, ce qui supprime les désagréables discontinuités de l’affichage. Dans certains cas, la structure est suffisamment souple pour offrir différentes précisions au sein du même objet (view-dependant LOD). C’est typiquement le cas des terrains à cause du mode d’observation particulier (angle de vue rasant). L’une des principales limitations de cette approche est son automatisme : aucune méthode de simplification de maillage universelle n’existe. Chaque algorithme fonctionne sur un type d’objet particulier. De plus les modèles d’illumination sont fréquemment ignorés. Par conséquent, un maillage très complexe (tel qu’une forêt par exemple) est impossible à simplifier puisque cela nécessiterait le re-codage de la fonction d’illumination à un niveau plus grossier. D’ailleurs, la simplification de maillage échoue systématiquement lors de variation d’échelle trop importante.

Certaines représentations se basent sur un échantillon régulier de l’information visuelle. Usuellement nommé *texture*, en 2D ou 3D, ce type de codage se prête bien aux méthodes de simplification [EWWL98]. Une information plus précise qu’une simple couleur est souvent utilisée ce qui permet, à l’instar des maillages, de réaliser des simplifications extrêmes [Rat97]. Notamment, la transparence est généralement bien mieux supportée. Les cartes graphiques offrent de plus en plus de possibilités : en 2D, un MIP-mapping anisotrope permet des simplifications de bonne qualité. Les textures volumiques sont un peu à la traîne mais certains travaux les simplifient par des tranches 2D [MN98, DDS03]. Malheureusement, les représentations volumiques ont actuellement un coût mémoire trop important pour une utilisation massive.

Entre le maillage et le volume, de nombreuses représentations hybrides se sont développées. Les méthodes de simplification d’objet représenté par des surfels produisent des résultats satisfaisants simplement en enlevant ou en supprimant des points [DVS03, RL00]. Les imposteurs proposent d’approximer des objets par des images dépendant du point de vue. Ce type de simplification redoutablement efficace pose néanmoins des contraintes très fortes sur l’observation : soit l’angle solide de validité de l’imposteur est petit, soit on retombe sur les coûts mémoires exorbitants d’un codage volumique.

D’autres méthodes de simplification originales existent et il est laborieux de toutes les recenser. Remarquons pourtant que ces méthodes ont une caractéristique commune : elles postulent l’existence d’un modèle décrit à son niveau le plus fin. Ce postulat implicite est souvent valable car les descriptions géométriques couramment utilisées supportent mal de gros modèles. Néanmoins, rien n’empêche de définir des modèles infiniment complexes (les fractales en sont un bel exemple). De plus, à cause du coût important des pré-calculs, l’animation ne passe que très rarement le cap de la simplification. En bref, dans le cadre d’une modélisation à grande variation d’échelle, la taille et le statisme des structures inhérentes à toutes méthodes de simplification limitent sérieusement leur utilité.

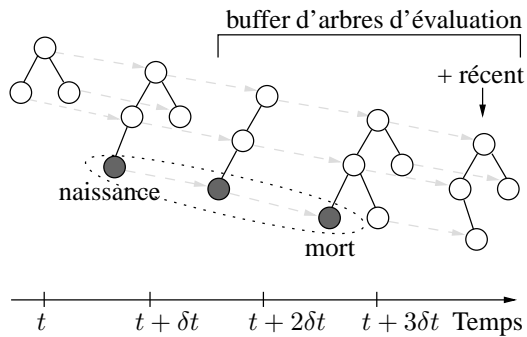


FIG. 6 – La connaissance de ces ancêtres permet d'utiliser la cohérence temporelle. Par exemple, une *vertex array* peut être allouée à la naissance d'un noeud, gardée sur la carte graphique tout au long de sa vie et détruite à sa mort.

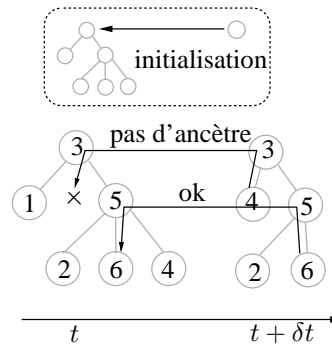


FIG. 7 – Avant l'évaluation d'un nouvel arbre, sa racine est connectée à la racine de l'arbre précédent. Ensuite, un parent, lorsqu'il instancie le noeud fils dont l'identifiant local est i , vérifie l'existence de l'ancêtre en demandant à son ancêtre s'il a un fils dont l'identifiant est i .

2.2 Fractales

Les plus beaux exemples d'algorithme de complexification sont sans aucun doute les générateurs de fractales [Bar93]. L'une des définitions fréquemment utilisées est : "une fractale est un objet infiniment complexe et auto-similaire à différentes échelles". Dans le cadre d'une modélisation à grande variation d'échelle, l'aspect infiniment complexe est définitivement une bonne propriété. Les fractales sont fréquemment codées par des systèmes récursifs tels que les L-systems [PHM99] ou les IFS [DHN85]. Ces méthodes définissent des fonctions de base appliquées récursivement un grand nombre de fois. Ainsi, quelques symboles suffisent à engendrer des objets qui nous *semblent* infiniment complexe. Cette disproportion entre la taille de la définition et la complexité de l'objet généré caractérise bien le charme des fractales.

Mais cette disproportion est aussi la cause de leurs plus grandes faiblesses : leur imprévisibilité et leur manque d'interactivité. Les L-systems paramétrables [Mec98] permettent de retrouver un minimum de contrôle. De plus, l'aspect auto-similaire des fractales n'est pas toujours désirable. En effet, on peut vouloir créer des mondes virtuels aussi complexes que *variés*. Les sub-L-systems [Mec98] permettent de constituer un modèle très varié constitué de plusieurs sous-modèles "emboîtés". Le langage L+C [KP03] se propose d'augmenter l'expressivité des L-systems en se rapprochant du langage C++. Ces deux travaux ont été notre principale source d'inspiration. Enfin, certains algorithmes de visualisation de fractales permettent un rendu temps-réel [Hub]. Ceci est réalisé grâce à une précision adaptative 4 et à l'utilisation intensive de la cohérence temporelle.

L'incontrôlabilité, la lenteur et la répétitivité des fractales sont donc des problèmes individuellement surmontés. DG rassemble ses solutions au sein d'un même outils qui peut être vu comme un générateur rapide et souple de fractales non-répétitives tridimensionnelles.

2.3 Cas particuliers

De récents travaux appliquent des méthodes de modélisation procédurale multi-échelle à des cas particuliers. Des mèches adaptatives permettent une animation et un rendu de cheveux efficace [BKCN03]. Des imposteurs semi-transparents à trois niveaux de détails sont utilisés pour modéliser des prairies [GPR⁺03]. Un océan modélisé par un maillage issu de la projection de la grille de l'écran est animé par des fonctions adaptatives [HNC02]. Des textures bi-directionnelles hiérarchiques sont utilisées pour rendre des arbres [MNP01]. Des surfels dont le nombre varie en fonction de la distance représentent des arbres [DCSD02]. La même représentation est utilisée pour des terrains grâce à un échantillonnage intelligent des lignes de fuite [SD01] et pour la foule [WS02] grâce à un ré-échantillonnage dynamique du maillage. Certains outils se concentrent sur les arbres et les terrains [LCV03, blu, spe, Des]. D'autres modélisent l'univers "vu de loin" [cel, gam, Win03, pla], d'autre des villes [PM03]. [vte] recense un grand nombre de travaux sur modélisation de terrain et de végétation.

Tous ces modèles dédiés sont innovants dans la mesure où ils ont pour la première fois permis d'afficher des

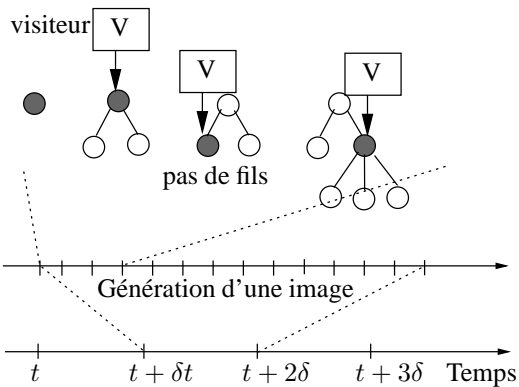


FIG. 8 – Le générateur, si la précision d'un noeud n'est pas suffisante, lui ordonne de s'enrichir en engendrant des noeuds fils.

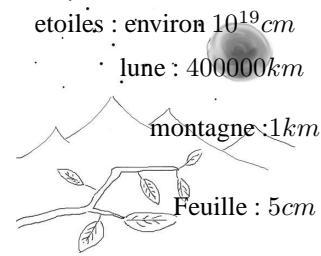


FIG. 9 – La précision du Z-buffer est parfois insuffisante...

phénomènes complexes et animés en temps-réel. En revanche, ils ont chacun fait l'objet d'un programme spécifique. C'est une tâche laborieuse qu'il serait souhaitable de simplifier. En effet, les environnements de développement actuels ne sont pas des lieux propices à la créativité. D'autre part, ces modèles ne sont valides que dans un certain intervalle d'observation. En proposant un environnement commun, nous permettons de coller plusieurs modèles bout à bout afin d'en faire un nouveau modèle au champ d'observation plus large.

Certains travaux proposent une interface graphique pour la modélisation multi-échelle dans des cadres bien particulier [KN02, BPF⁺02]. Des retouches de plus en plus précises sont réalisées afin de se rapprocher itérativement de la forme désirée. Nous adhérons complètement à la philosophie de ses travaux très axés sur l'interface utilisateur. Ces modeleurs sont complémentaires à notre représentation fonctionnelle et à nos outils de rendu multi-échelles.

3 Représentation procédurale

La scène est définie par une représentation procédurale appelée *fonction de forme* évaluée à chaque pas de temps (fig. 5). Cette fonction est évaluée dans un arbre d'évaluation qui permet, grâce à un système d'identifiant unique, l'utilisation de la cohérence temporelle (fig. 6).

3.1 La fonction de forme

La matière visible est représentée procéduralement par une *fonction de forme*. Elle est composée de *sous-fonctions* capables de s'appeler les unes les autres (fig. 5). Chacune de ces fonctions enrichie successivement la forme déjà calculée. L'évaluation de la fonction de forme est réalisée grâce à un *arbre d'évaluation* dont chaque noeud est une instance de l'une des sous-fonctions. Remarquez bien la différence entre la description procédurale de la scène (la fonction de forme) et son interprétation, c'est à dire son calcul à chaque pas de temps (l'arbre d'évaluation "dynamique").

Les sous-fonctions de la fonction de forme sont écrits dans un langage spécifique basé sur un C++ légèrement modifié (à la manière des Q_OBJECT de Qt). Une sous-fonction est en fait une classe particulière héritant d'une classe de base. Cette classe de base impose certaines propriétés commune à toutes les sous-fonctions et nécessaire pour le bon déroulement de l'évaluation :

- Une sous-fonction sait évaluer sa *visibilité*. La visibilité définit l'état de la matière créée par le noeud relativement à une l'observation. La matière peut être plus ou moins précise, plus ou moins occultée et hors-champ ou non.
- Elle sait dessiner la forme qu'elle a créée et sait l'entourer d'une boîte englobante.
- Elle connaît sa place dans l'arbre d'évaluation (distance à la racine, distance aux feuilles, père, fils...).

Certaines caractéristiques ont un comportement par défaut, mais la plupart doivent être explicitement codées par l'utilisateur.

L'évaluation de la fonction de forme commence par l'instance d'une sous-fonction axiome donnée par l'utilisateur. A partir de cette instance est évalué tout le restant de l'*arbre d'évaluation* (dont l'instance axiome est

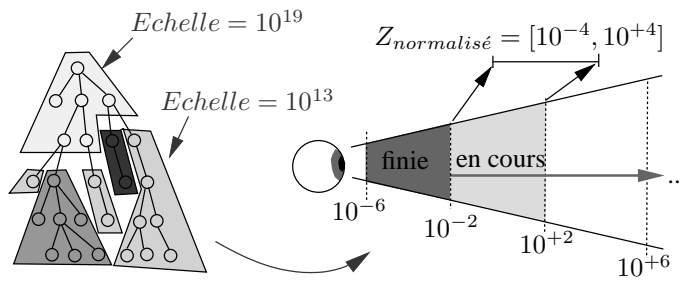


FIG. 10 – Afin de garder des réels toujours bien normalisés, des zones de différentes échelles sont créées au sein du graphe. Elles se répercutent indirectement sur les zones de profondeur du Z-buffer.

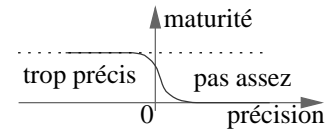


FIG. 11 – La précision est une fonction qui renvoie un réel en fonction d'un noeud-fonction. La maturité est une normalisation commode de cette valeur.

la racine). Cette évaluation est assurée par un visiteur [SLL02] particulier : le *générateur* (fig. 8). D'autres visiteurs existent : le destructeur détruit un graphe d'évaluation. Le visualiseur donne une représentation formelle de l'arbre d'évaluation. Le re-dessinateur permet de dessiner une fonction sous un autre angle de vue que celui de la génération (fig. 1). Au besoin, de nouveaux visiteurs peuvent être ajoutés par l'utilisateur.

A chaque observation correspond un arbre d'évaluation. Dans tout ce qui suit, on considère que la caméra bouge à chaque pas de temps. En conséquence, un nouvel arbre d'évaluation est entièrement généré à chaque nouvelle image. Chaque évaluation est dite *paresseuse* . Cela signifie la précision ne sera ajoutée (et calculée) que là où nécessaire (cf. John C. Hart dans [EMP⁺98]) : lors de la visite du générateur, les sous-fonctions filles ne sont évaluées que si leur parent n'est pas assez précis.

3.2 Noeuds fils et identifiant

La plupart des modèles procéduraux sont basés sur un appel récursif de fonction. Dans ce cas, l'arbre d'évaluation est en fait la pile d'appels du processeur. Cette pile étant détruite après chaque évaluation, l'état du modèle aux pas de temps précédents est inconnu. Pourtant, cette connaissance est essentielle pour deux raisons :

- Si la caméra bouge continûment (comme c'est souvent le cas), l'évaluation d'un modèle procédural est très similaire sur des pas de temps voisins. Cette *cohérence temporelle* permet d'optimiser de nombreux calculs et de gagner un temps précieux.
- Pour rendre un univers interactif, il faut l'animer de façon *réactive* : les paramètres d'un modèle à un instant t sont une modification de ceux au temps $t - \delta t$.

Ainsi, DG assure que chaque arbre d'évaluation connaît son ancêtre et son successeur (si ceux-ci existent). Un tampon garde en mémoire les n derniers arbres d'évaluation (fig. 6). Lorsqu'un nouvel arbre est évalué il crée des relations d'ancêtre avec l'arbre précédent et le dernier arbre est détruit.

Pour que chaque instance de sous-fonction, au sein d'un l'arbre d'évaluation, puisse se souvenir de son ancêtre (si celui-ci existe), un *identifiant global* leur est attribuée. Cet identifiant, pour une évaluation, est unique. Ainsi, lorsqu'un noeud est instancié, il vérifie si un noeud du même identifiant global existe dans l'arbre d'évaluation précédent. Un identifiant global est une suite d'*identifiant local* indiquant le chemin de la racine au noeud en question. Chaque identifiant local est unique parmi ces frères (fils d'un même parent), ce qui assure un chemin unique. De fait, les comparaisons d'identifiant global ont un coût proportionnel à la hauteur du graphe. Pour revenir à un coût constant, on se ramène toujours à des comparaisons locales (fig. 7).

Les fils d'un noeud sont stockés dans un conteneur. La connaissance de l'arbre d'évaluation précédent repose sur une utilisation intensive de ce conteneur et l'efficacité de ce dernier influe donc énormément sur les performances globales. Malheureusement, il est impossible de trouver un conteneur universellement efficace car la répartition et le nombre de fils peuvent varier complètement d'un type de sous-fonction à l'autre. Par exemple, un tableau à taille fixe, d'habitude très efficace, est inutilisable dans le cas d'une prairie car il faudrait allouer un tableau de pointeur dont la taille est le nombre de brin d'herbe total. En conséquent, l'implémentation du conteneur est laissée au soin de l'utilisateur. Ceci complexifie sensiblement la modélisation. Néanmoins, ce choix est aussi motivé par le tri spatial effectué sur les fils d'un noeud à chaque évaluation (cf. section 3.3). En effet, une bonne connaissance de la répartition spatiale des fils d'un noeud permet souvent de réduire le coût du tri (c'est typiquement le cas d'une grille régulière).

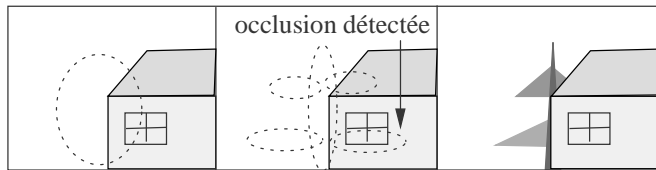


FIG. 12 – Les boîtes englobantes dont aucun fragment ne passe le test de profondeur de la carte graphique sont ignorés. Ce test de détection d’occlusion est surtout avantageux pour des noeuds fonctions susceptible d’engendrer beaucoup de fils.

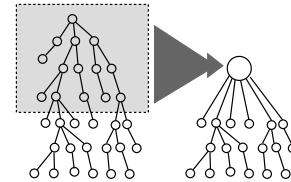


FIG. 13 – Travaux futurs : si on est sûr de ne jamais ”dé-zoomer”, on peut geler une partie du graphe et simplifier sa géométrie définitivement.

3.3 Affichage de l’arbre d’évaluation

L’affichage est effectué en même temps que l’évaluation, c’est une condition essentielle au bon déroulement de l’algorithme de visibilité. Les cartes graphiques permettent désormais de récupérer le nombre de fragments qui passe le test de profondeur (`ARB_occlusion_query`). Une requête de visibilité permet d’évaluer le nombre de pixels affichés par un objet 3D. Ainsi, si aucun fragment de la boîte englobante d’un objet n’atteint le tampon image, celui-ci est donc complètement occulté. Il est alors possible de confronter la boîte englobante d’un noeud à l’état courant du Z-buffer afin de déterminer s’il doit être affiché ou enrichi, ou bien simplement ignoré (fig. 12). La méthode présentée ici est une version procédural des algorithmes présentés dans [HSLM02, YSM03]. L’idée de base est la même, mais nous tirons parti de la souplesse d’une description procédurale pour éviter un pré-calcul de cluster. En conséquence, l’animation est parfaitement supportée.

La précision flottante est normalisée tout le long de la génération. Des *zones d’échelle* sont créées au sein de l’arbre d’évaluation : lorsqu’un noeud devient trop petit, on change de zone de façon à retrouver une taille raisonnablement grosse. L’utilisation de ces zones signifie qu’un noeud ne peut accéder directement aux caractéristiques spatiales d’un autre noeud. Afin de lever partiellement cette limitation, il est possible de contraindre l’algorithme pour assurer que des objets ayant particulièrement besoin d’échanger des informations fassent partie de la même zone.

La précision du Z-buffer est traitée indépendamment. Quelque soit la distance d’un objet par rapport à la caméra, c’est toujours le même intervalle normalisé qui est utilisé. La profondeur est découpée en *zones de profondeur* qui sont affichées de la plus proche à la plus lointaine (fig. 10). D’habitude, l’affichage est réalisé du plus lointain au plus près, mais cela est incompatible avec l’algorithme de détection d’occlusion. Après le rendu de chaque zone de profondeur, un bit du stencil-buffer est mis à jour de la façon suivante : pour chaque pixel de l’écran, si la profondeur n’est pas maximum, alors la valeur du bit est mise à 1. Le test de profondeur devient alors : `(bit == 0) && (zfragment < ztarget)`. Cette méthode simple permet de donner au tampon de profondeur une précision infinie au prix d’une mise à jour du stencil-buffer avant le rendu de chaque zone.

L’utilisation de l’algorithme de détection d’occlusion et de précision infinie entraîne de nombreuses contraintes. Il est impossible de dessiner l’objet d’une zone de profondeur dont ce n’est pas encore le tour. La synchronisation entre le processeur central et la carte graphique doit être soigneusement ajustée afin d’éviter toute attente de l’un ou de l’autre. Le diagramme 14 représente très succinctement la façon dont elle sont traitées.

4 Conclusion

Les résultats d’une utilisation de DG par deux personnes durant deux mois sont décrites. S’ensuit une discussion générale.

4.1 Utilisation

Avant de décrire les modèles codés avec DG, décrivons rapidement quelques fonctionnalités jusque-là passées sous silence :

- Lorsque le code d’un modèle est recompilé (avec un compilateur C++ classique), le programme de visualisation recharge le code à la volée. Modulo le temps de compilation (quelques secondes en mode non-optimisé), on peut

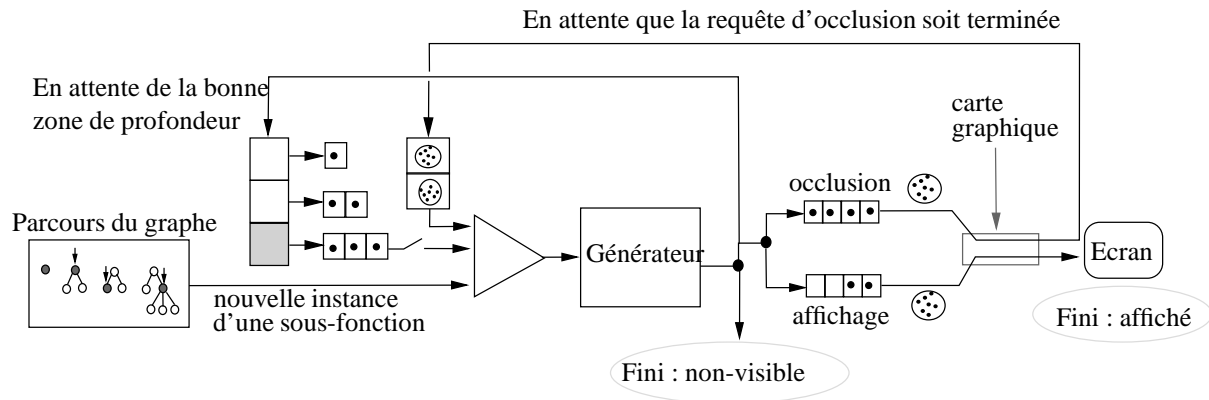


FIG. 14 – L’algorithme de génération est une sorte de prolongation software de la carte graphique.

donc voir instantanément les répercussions des modifications effectuées.

- Afin de contrôler l’énorme masse d’information générée, l’arbre d’évaluation peut être visualisé 1. Les instances de noeud peuvent être sélectionnées sur ce graphe ou directement sur la forme générée. Ils ont chacun une mini-interface OpenGL qui permet d’afficher et de modifier leurs valeurs.
- Un mode pas-à-pas permet de dérouler doucement l’arbre d’évaluation afin de trouver plus efficacement un bug.
- Des mesures d’occupation mémoire et de coût de calcul sont constamment disponibles sous forme de courbes 2D et permettent de profiler aisément le code du modèle.
- La précision de chaque noeud du graphe est évaluée par défaut et propose un coefficient de *maturité* 11. Celui-ci permet aisément d’interpoler continûment des formes d’un niveau de détail à l’autre.

Durant le développement de modèle sous DG, ces fonctionnalités ont définitivement prouvées leur utilité.

En dehors de son concepteur, deux personnes ont utilisé DG pendant deux mois. Ces deux utilisateurs ”test” avaient tous deux de bonnes connaissances en C++ et aucune connaissance en OpenGL. En trois jours, ils ont chacun réalisé une fractale (le tétraèdre et le cube de Sierpinski). Ce temps comprenant la prise en main de DG, cela nous semble satisfaisant. Bien sur, ce genre de modèle peut être codé à partir de rien en une ou deux heures par un expert. Mais il ne bénéficiera pas d’un affichage interactif après 14 subdivisions (fig. 1). Le reste du temps, ils se sont concentrés sur la réalisation d’arbres (fig. 15). Les résultats sont esthétiquement peu probants mais très encourageants. La structure développée est robuste et efficace et permet de modéliser différents types d’arbres aisément. Grâce à l’utilisation de la maturité 11, l’arbre adapte *continûment* sa précision à l’observation. Il est animé (sans aucun surcoût) par des bourrasques de vent.

4.2 Discussion

L’outil que nous avons développé propose un cadre de travail pour la modélisation multi-échelle. Avec des fonctions simples, un affichage interactif est réalisé. Lorsque la scène est réellement complexe, dans le meilleur des cas, la fréquence d’affichage est d’environ cinq images par seconde (avec un PC à 2.4 Ghz et un carte graphique NVidia TI4000). Le nombre de polygones affichés, souvent utilisé pour quantifier les performances, n’est pas un critère valide puisque la scène est infiniment précise. Compte tenu de la conception très dynamique de l’algorithme, l’animation du modèle est permise et n’entraîne comme surcoût que son propre calcul. D’ailleurs, remarquons qu’il est impossible d’assurer une certaine fréquence d’affichage. En effet, le coût de l’algorithme dépend du coût des fonctions d’évaluation de chaque noeud-fonction et celles-ci sont entrées par l’utilisateur. En pratique, ces fonctions déterminent quasiment à elle seule le coût de l’évaluation de la fonction de forme.

Pourtant, l’algorithme ”à vide” a évidemment un coût. En effet, si les noeuds-fonctions sont négligeables, le facteur limitant devient la génération du graphe. Cette limitation soulève un point très délicat de la modélisation par complexification : la forme est régénérée à chaque pas de temps à partir de sa représentation la plus grossière. Notre algorithme ne permet pas de *simplification procédurale*. Ceci semble se rapprocher du *problème inverse* des méthodes de simplifications fractales. Si l’approximation de fonction 1D ou 2D fonctionne dans certain cas, il n’est actuellement pas envisageable de l’étendre à n’importe qu’elle forme tridimensionnelle. Peut-être est-il possible de ”geler” la partie imprécise du graphe pour se concentrer sur la partie précise. Mais la encore, le ”dégel” s’apparente



FIG. 15 – Les arbres sont générés par deux type de sous-fonctions : les branches (trois niveaux de récursion) et les feuilles.

au *problème inverse* et pose de sérieux problèmes. Remarquons qu’une application dont le zoom serait un ”aller simple” (un shoot-them-up multi-échelle ?) permettrait simplement d’oublier toute la partie haute du graphe (fig. 13).

Pour assister la modélisation multi-échelle, l’interface que nous fournissons n’est malheureusement pas graphique. Il s’agit ici d’interface au sens API (application program interface). Cela signifie que les modèles procéduraux développés avec notre algorithme sont écrits en C++. Cela est évidemment un facteur limitant, mais aussi un passage obligé. En effet, Une interface graphique est en quelque sorte une API à son plus haut niveau d’abstraction et pour l’atteindre, il faut bien passer par les niveaux intermédiaires. Ceci dit, nous pensons que la réalisation d’interfaces graphiques pour la modélisation procédurale est réellement prometteuse. Cette activité revient plus ou moins à imaginer une interface de développement purement graphique. Nous pensons qu’elle compte parmi les défis scientifiques majeurs et qu’elle sera la motivation d’un pont essentiel entre la synthèse d’image et l’interaction homme-machine.

Références

- [Bar93] Michael F. Barnsley. Fractal modelling of real world images. In *Fractals Everywhere*. pub-AP, 1993.
- [BKCNO3] Florence Bertails, Tae-Yong Kim, Marie-Paule Cani, and Ulrich Neumann. Adaptive wisp tree - a multiresolution control structure for simulating dynamic clustering in hair motion. *Symposium on Computer Animation ’03*, July 2003.
- [blu] Blueberry3d. <http://www.blueberry3d.com/>.
- [BPF⁺02] Frederic Boudon, Przemyslaw Prusinkiewicz, Pavol Federl, Christophe Godin, and Radoslaw Karwowski. Interactive design of bonsai tree models. In *Eurographics Rendering Workshop 2002*, 2002.
- [cel] Celestia. <http://www.shatters.net/celestia>.
- [DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the IEEE Visualization Conference*. IEEE, October 2002.
- [DDSD03] Xavier Décoret, Frédo Durand, François Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *Proceedings of the ACM Siggraph*. ACM Press, 2003.
- [Des] Descensor. <http://www.binaryworlds.com/index.html>.
- [DHN85] Stephen Demko, Laurie Hodges, and Bruce Naylor. Construction of fractal objects with iterated function systems. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 271–278. ACM Press, 1985.
- [DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics (TOG)*, 22(3) :657–662, 2003.
- [EMP⁺98] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling, A Procedural Approach*. AP Professional, third edition, 1998.
- [EWWL98] Jon P. Ewins, Marcus D. Waller, Martin White, and Paul F. Lister. Mip-map level selection for texture mapping. In *IEEE Transactions on Visualization and Computer Graphics*, 1998.

- [gam] Celestia. http://www.gamasutra.com/features/20010302/oneil_01.htm.
- [GPR⁺03] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure, and Marie-Paule Cani. A procedural approach to animate interactive natural sceneries. In *CASA03*, 2003.
- [HNC02] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *Symposium on Computer Animation*, july 2002.
- [HSLM02] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha. Fast and simple occlusion culling using hardware-based depth. Technical report, University of North Carolina, Chapell Hill, 2002.
- [Hub] Jan Hubicka. Real-time fractal zoomer. <http://www.gnu.org/software/xaos/xaos.html>.
- [KN02] Tae-Yong Kim and Ulrich Neumann. Interactive multiresolution hair modeling and editing. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 620–629. ACM Press, 2002.
- [KP03] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. Design and implementation of the l+c modeling language. In Jean-Louis Giavitto and Pierre-Etienne Moreau, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
- [LCV03] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. In *Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 31–38. ACM Press, 2003.
- [LRC⁺02] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann / Elsevier Science, 2002.
- [Mec98] Radomir Mech. *CPFG Version 3.4 User's Manual*, 1998.
- [MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, Jul 1998. Eurographics, Springer Wein. ISBN.
- [MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, Jul 2001.
- [PHM99] Przemyslaw Prusinkiewicz, Jim Hanan, and Radomir Mech. An l-system-based plant modeling language. In *Proceedings of the International workshop AGTIVE'99*, 1999.
- [pla] Planet engine. <http://drtypo.free.fr/index.html>.
- [PM03] Yoah Parish and Pascal Müller. Procedural modeling of cities. *ACM Transactions on Graphics (TOG)*, 2003.
- [Rat97] Karim Ratib. Texture volumique multi-echelle pour l'affichage de scenes complexes. M.sc. thesis, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 1997.
- [RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat : A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
- [SD01] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In K. Myskowski and S. Gortler, editors, *Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering 01)*, 12th Eurographics workshop on Rendering. Eurographics, Springer Verlag, 2001.
- [SLL02] Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library : User Guide and Reference Manual*. Addison-Wesley, 2002.
- [spe] Speedtree. http://www.idvinc.com/html/product_browser.htm.
- [vte] Virtual terrain project. <http://www.vterrain.org/>.
- [Win03] Jochen Winzen. Interactive visualisation of a planetary system. Master's thesis, university of Karlsruhe, Germany, 2003.
- [WS02] M. Wand and W. StraBer. Multi-resolution rendering of complex animated scenes, 2002.
- [YSM03] Sung-Eui Yoon, Brian Salomon, and Dinesh Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *IEEE Visualization, 2003*, 2003.