Table des matières

1. Introduction

Le présent document est une capitalisation des recherches entreprises depuis 1990 dans le domaine de la simulation de l'architecture des plantes au laboratoire AMAP du Cirad. Les connaissances Botaniques incluses dans l'architecture végétale proviennent quasiment dans leur totalité des travaux de l'ex institut botanique de Montpellier sous la direction de Francis Hallé. Celui-ci a défini la notion de modèle architectural (1978) qui permet de classifier le développement de l'architecture des arbres en moins de trente types caractéristiques. Cette classification s'appuie directement sur le fonctionnement des méristèmes et non sur l'allure ou le port de l'arbre. Un même port (en boule, fastigié, pleureur) peut cacher des fonctionnements méristématiques très différents. Un méristème peut être ainsi caractérisé par un fonctionnement indéterminé avec une floraison latérale (comme pour un séquoia) ou déterminé donnant naissance à des modules à floraison terminale (comme pour un frangipanier). Les modes de ramification engendrés par ce type de fonctionnement sont nommés respectivement monopodial ou sympodial. La croissance des axes peut être d'autre part rythmique ce qui donne naissance à des unités de croissance (exemple Hévéa) ou bien continue en donnant une ramification diffuse (exemple eucalyptus). Ajoutons à cela qu'il existe deux types de rameaux : dressés et dénommés orthotropes ou bien étalés et dénommés plagiotropes. Ce sont les combinaisons de ces divers caractères qui engendrent la diversité des modèles architecturaux des arbres. Au niveau des herbacées on retrouve les mêmes phénomènes en plus simple dans la structure des inflorescences, véritables petits arbres où un arrangement géométrique d'empilement de modules à fleurs peut s'observer. Alors que ces particularités étaient connues depuis longtemps par les botanistes (Weberling 1989) chez les fleurs, les aspects combinatoires de l'architecture des arbres, beaucoup moins apparents, sont restés ignorés jusqu'à ce que F. Hallé les mette en évidence.

Dans le fouillis apparemment inextricable de l'architecture des arbres on peut observer un même type de rameau émergeant ça et là de nombreuses fois. C'est le cas notamment des arbres fruitiers abondamment taillés. Ainsi les axes courts porteurs de la fructification (appelés lambourdes, dards, chiffones selon l'espèce), se trouvent-ils répartis dans la totalité de la structure végétale. Les différents types d'axes en petit nombre (au plus une demi douzaine) que l'on observe sur un arbre peuvent être ordonnés selon un gradient morphologique que l'on appelle âge physiologique (Rivals 1965, Rouane, 1977) et qui dépend de l'état de différentiation des méristèmes.

Toutes les valeurs numériques des fonctionnalités décrites ci-dessus pour caractériser le fonctionnement d'un méristème dépendent ainsi de l'âge physiologique qui devient un index commode pour classifier les modes de croissance d'un axe. Ainsi les gradients de différentiation des feuilles observés notamment chez les herbacées depuis la base de la tige jusqu'à l'inflorescence terminale et appelés « métamorphose » selon le terme de Goethe (1790), peuvent être contrôlés par cette notion d'âge physiologique. Cette notion d'âge physiologique comme descripteur de mode de croissance et ce principe de métamorphose comme règle d'évolution de l'âge physiologique ont été utilisés pour formaliser la description de la croissance du cyprès par Rouanne (1977).

Il est d'usage de parler de développement lorsqu'on évoque la mise en place des nouveaux organes par les méristèmes et de croissance lorsqu'on parle de l'expansion de ceux-ci. Ainsi le développement architectural sera lié ici au fonctionnement des bourgeons c'est à dire à l'organogénèse et la croissance végétale désignera l'expansion des organes comme le résultat de l'allocation de la biomasse produite par la photosynthèse. Naturellement développement et croissance vont de pair, mais cette distinction a son importance car elle est à l'origine des deux grands types de modèles : les arbres géométriques de l'informatique pour le premier et les modèles fonctionnels de l'agronomie pour la deuxième.

1.1 Modèles de croissance de plante

1.1.1 Modèles de développement des plantes informatiques (Growth Models)

Les arbres informatiques couplent un algorithme d'organogénèse avec des règles géométriques de mise en place d'organes pré dimensionnés dans une structure tridimensionnelle. Les organes ne jouent ici qu'un rôle figuratif et le seul fonctionnement pris en compte est celui de l'organogénèse.

Il n'est point besoin d'avoir des connaissances en botanique pour obtenir des maquettes 3D d'arbres réalistes. Il faut laisser suffisamment de points de contrôle sur les algorithmes du développement et de la géométrie du végétal à un « designer » pour qu'il puisse modeler une forme complexe avec une bonne esthétique. Des logiciels comme Onyx tree ou Xfrog en sont une belle démonstration. De toute façon, généralement la perception de l'homme ordinaire de ce qu'est un végétal est très en dessous de la réalité botanique et il est inutile de raffiner au delà … Le plus souvent c'est à la forme des feuilles ou des fleurs qu'une plante est reconnue et non pas à son architecture.

Dans le domaine de l'image de synthèse « what is good is what looks good » selon O. Deussen le développeur d'XFrog. On est loin des exigences scientifiques de la réalité botanique. Par contre sur le plan informatique on peut parler de prouesses dans les améliorations du rendu des scènes végétales : des algorithmes ingénieux pour simuler le développement végétal, (Grammaires, Automates, Fractals, lancé de particules) et de rendu (textures, radiosité, lancé de rayons) contribuent à fournir des scènes de plus en plus réalistes pour l'image de synthèse et le paysagisme. Nous aborderons cet aspect au paragraphe 1.2.2. Les plantes virtuelles sont devenues les auxiliaires indispensables des projets d'aménagements autour de la simulation du bâti.

Deux exceptions notables des arbres informatiques sont le cas des L-system et le cas d'AMAP qui ont visé la qualité botanique des architectures de plantes simulées comme préalable à leur visualisation. On peut parler d'une sous classe botanique des arbres informatiques. En particulier l'organogénèse est simulée le plus exactement possible afin d'engendrer la structure topologique correspondante. Par contre l'habillage géométrique de cette structure utilise les mêmes algorithmes. Il s'agit ici de modéliser le fonctionnement des méristèmes et de simuler la production d'organes en relation par exemple avec le rendement. Les L-system se sont davantage focalisés sur la structure des inflorescences et AMAP sur la structure des arbres.

Les grammaires ou les automates stochastiques peuvent être calibrés à partir des données architecturales prélevées sur les plantes réelles, on obtient ainsi un modèle de développement entièrement dédié à la plante étudiée.

1.1.2 Modèles de croissance de plantes de l'agronomie.(Process Based Models)

En agronomie ce n'est pas la représentation géométrique de la plante qui compte, mais la production végétale de celle-ci au niveau du m². Le mètre carré de plante est ainsi divisé en compartiments qui regroupent les différents types d'organes (feuilles, tiges, fruits, racines) provenant du mélange de plantes considérées. On parle de source en ce qui concerne les feuilles productrices de biomasse et de puits pour les autres organes attracteurs de biomasse (y compris les feuilles). La croissance peut être abordée comme un problème de relation source-puits c'est-à-dire de production et d'allocation de biomasse dans les différents compartiments.

Les paramètres environnementaux principaux (en supposant la plante non stressée hydriquement) sont la température et la lumière. Tant qu'elle reste dans la plage de réponse linéaire, la température agit essentiellement sur la vitesse de développement des plantes (Bonhomme, 2000).

La lumière agit de deux façons :

- L'intensité utile perçue par la plante (PAR). Seule une partie de la radiation solaire est utilisée par la photosynthèse.
- La quantité de lumière interceptée par m². Les feuilles sont des capteurs étalés au dessus de la surface du sol. L'indice foliaire (LAI) est la somme des surfaces de feuilles par mètre carré de sol. La production de biomasse /m² sature rapidement à partir du moment où l'interception de lumière par les feuilles atteint un maximum, ce qui arrive pratiquement à partir de LAI=4. Il ne sert plus à rien à partir de ce seuil d'augmenter la densité de plantation (Jones 1992).

Le taux de production/m² est donné par l'équation :

$$Q = LUE \cdot PAR \cdot (1 - e^{-k*LAI})$$ (Monsi, 1953)

Le paramètre LUE est l'efficience de la lumière pour convertir l'énergie en biomasse, le paramètre k est lié à l'orientation des feuilles et en général est voisin de 0.8 (deReffye, 2008).

En été le PAR peut avoisiner 8 MJ/j/m² et la LUE 30 g Mf/m²/j. Ce qui fait que l'on peut produire un peu plus de 200 g de matière fraiche/m²/jour. Le formalisme robuste de l'effet du LAI sur l'interception de la lumière est celui de la loi dite de Beer-Lambert (Monsi, 1953).

Les modèles de croissance sont très utilisés. Leur qualité principale est la robustesse de la relation biomasse-énergie reçue par le couvert qui repose sur le fait que l'on peut considérer la photosynthèse nette (coût de respiration déduit) et qui repose également sur une notion de pool commun des réserves qui alimentent les compartiments selon une règle de

puits proportionnels. La simplicité de leur formulation rend leur calibration possible. L'inconvénient est qu'ils ne suivent pas la phénologie de la plante, c'est-à-dire les

changements progressifs du fonctionnement dus entre autres à des modifications de la morphogénèse. La qualité de prédiction de ces modèles diminue quand les données de l'environnement changent. Par exemple si on change la densité de plantation la production au m² ne changera pas à partir d'un certain niveau de LAI, mais cela recouvrira des architectures de plantes très différentes avec des changements phénologiques importants qui modifient les proportions allouées aux compartiments.

La principale cause du manque de qualité de prédiction en environnement variable est reconnue comme l'absence de prise en compte de l'architecture de la plante et des feed-back entre l'organogénèse et la photosynthèse (de Reffye 2003).

1.1.3 Modèles structure-fonction (Functional Structural Process Models)

Pour pallier aux inconvénients des PBM, les agronomes et les informaticiens ont créé les modèles structures fonctions qui consistent à donner aux organes mis en place par les méristèmes un rôle fonctionnel. On se place dans la classe botanique des arbres informatiques. Pour autant les modèles structure-fonction n'ont pas essayé de rivaliser avec les PBMs dans le domaine de la production végétale, mais plutôt se sont concentrés sur le fonctionnement physiologique détaillé de l'individu plante. La structure topologique issue du développement permet d'interconnecter les organes entre eux et la structure géométrique par le biais des maillages, permet de calculer l'interception de la lumière par chaque feuille et de faire circuler les sèves brute et élaborée dans le réseau ramifié de l'architecture de la plante. Toutes les simplifications validées et utilisées dans les PBMs sont ignorées dans les FSPMs ou l'objectif est le recours à tout simuler dans le détail en concentrant un maximum de connaissances (Marcelis 1998) :

1. L'interception de la lumière sera simulée par des logiciels de transferts radiatifs opérant sur la maquette 3D maillée par des polygones.
2. La photosynthèse sera simulée au niveau de chaque feuille compte tenu de son microclimat.
3. un modèle transport-résistance diffuse selon un pas de temps et un gradient physiologique, la matière produite par les organes sources aux organes puits en utilisant le réseau des branches maillé en éléments finis.
4. les coûts de la respiration (transport, entretien) sont évalués, ils diminuent d'autant l'assimilation par la structure de la matière produite par les feuilles.

Toutes ces opérations sont coûteuses en temps et en mémoire dès que l'on monte à l'échelle d'un arbre qui produit des milliers voire des centaines de milliers d'éléments. Le temps de calcul du développement de la plante est proportionnel au nombre d'éléments à mettre en place auquel il faut ajouter le temps de calcul de la croissance. Se pose alors le problème de la fiabilité de la simulation elle-même, qu'il est difficile voire impossible de calibrer sur des fonctions mathématiques. Ces simulations sont sans preuves au sens informatique du terme.

La dérivation du code est généralement impossible, ce qui empêche la résolution des problèmes inverses, l'optimisation et le contrôle de s'appuyer sur des algorithmes mathématiques efficaces et il faut recourir aux méthodes heuristiques qui obligent à lancer des milliers de fois le calcul de la plante.

Les codes des FSPMs apparaissent davantage donc comme des outils de capitalisation des connaissances et sont utiles en tant que tels à l'enseignement par exemple, plutôt que comme des nouveaux outils utilisables en Agronomie.

Pour finir les FSPMs concernent le fonctionnement de la plante individuelle. Un autre problème est celui du passage de la plante au peuplement que nous n'aborderons pas ici.

Au travers de ce panorama des différentes classes de modèles de croissance de plante on se rend compte que la manière dont un modèle est conçu dépend étroitement de la question que l'on se pose au sujet de l'objet que l'on cherche à modéliser et qui définit sa problématique :

> *« To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A" (Minsky, 1965).*

1.2 Simulation

Quand le modèle a été établi on peut le programmer dans un ordinateur afin de le simuler. On peut ainsi obtenir rapidement des sorties du modèle en fonction des valeurs de ses paramètres d'entrée. Cela permet d'étudier le comportement du modèle, sa robustesse et sa sensibilité afin d'éprouver notre propre connaissance (Jorgensen, 1994).

1.2.1. Considérations générales

1.2.1.1. Connaissances informatiques requises

Dans la mesure ou l'objectif est de simuler en utilisant un ordinateur, il est nécessaire d'écrire un programme qui va être déroulé par le ou les processeurs du calculateur. Selon la spécificité ou la généricité du modèle à simuler, on peut envisager d'utiliser des environnements plus ou moins dédiés et des méthodes de programmation de différentes natures. Nous pensons de manière de plus en plus abstraite au choix du matériel qui va héberger le simulateur, au langage et à l'environnement dans lequel il va être développé et enfin les techniques de simulation qui vont être utilisées.

Il va falloir effectuer un choix pour le langage, la technique et l'environnement de programmation (Zanella 1999). Ces considérations peuvent paraitre futiles mais influent grandement sur les performances et la pertinence du simulateur qui va être produit.

Pour ce qui concerne le langage, on dispose globalement de trois niveaux d'approche.

Les langages génériques de programmation tels que C++, Java ou Python permettent une totale souplesse dans les choix algorithmiques et offrent tous les avantages des langages objets (héritage, réutilisation, factorisation…) et proposent des librairies éprouvées permettant de résoudre des problèmes connus (manipulation de tableaux, de fichiers, interfaces graphiques…). Par contre le travail dédié à la programmation prend une part importante de la charge de réalisation d'un simulateur en regard de la partie conception.

Une tendance actuelle alternative à la programmation consiste à utiliser **des environnements dédiés dits de « haut niveau »** qui proposent des objets intégrés offrant des fonctionnalités dédiées à des types de modélisation particuliers. Simula (Dahl et Nygaard, 1966) est le parent de cette approche et propose depuis l'aube de l'informatique le paradigme orienté objet. Grâce à une librairie de classes offrant un support concret à la simulation discrète, il définit les notions de base de l'approche objet (classe, attributs, méthodes, héritage, encapsulation, polymorphisme, instance…). Ensuite, nous pensons en particulier à la modélisation par grammaire à réécriture de type L-system (L-studio (Karwowski and Prusinkiewicz, 2004), GroImp (Kniemayer, 2004)…) ou bien à l'approche multi-agents (MadKit (Gutknecht, 2000), CORMAS(Bousquet, 1998)…). Dans ces deux cas un ensemble d'outils adaptés au choix de modélisation permettent à l'utilisateur de s'affranchir d'une partie des tâches de programmation liées à ces choix. On peut citer des interpréteurs de règles, des ordonnanceurs de tâches, des interfaces de communication ou bien des syntaxes adaptées à la description des objets à manipuler. L'utilisation de ces environnements pour des applications sophistiquées n'affranchira malheureusement pas l'utilisateur de la fastidieuse tâche de développement des fonctions spécifiques à intégrer dans l'environnement hôte. On peut cependant noter que l'approche objet constitue non seulement un outil de programmation mais également une approche de conception et donc de modélisation (Coquillard et Hill, 1996).

Enfin, le stade ultime de l'outil d'aide au développement de simulateur de modèle est actuellement **la plateforme** (L-Studio, ModelMaker (Cherwell Scientific, Oxford, UK), OpenAlea (Pradal, 2006)). On peut considérer que ces objets possèdent un noyau de type « haut niveau » auquel on a ajouté une interface graphique qui permet à l'utilisateur de bâtir son application par l'usage exclusif de la souris. Evidemment le trait est extrêmement forcé mais l'idée consiste tout de même à abstraire au maximum l'utilisateur de la programmation de base et de lui offrir des interfaces conviviales qui permettent de construire un simulateur à partir de la mise en relations et du paramétrage d'objets préprogrammés. Dans la mesure où ils s'appuient fortement sur la composante graphique, ces environnements proposent souvent des outils de visualisation et d'analyse du résultat des simulateurs qu'ils hébergent. Il n'en reste pas moins que la tâche qui consiste à développer les objets préprogrammés reste incontournable et que l'utilisateur qui ne trouve pas son bonheur dans l'offre par défaut d'une plateforme devra à son tour revenir vers le monde merveilleux de la programmation ! Par contre chaque objet développé devient immédiatement disponible pour la communauté des utilisateurs de la plateforme, on atteint pleinement l'objectif de mutualisation et de factorisation des outils.

En résumé on peut dire que le niveau technique de connaissance en développement informatique requis pour développer un simulateur est proportionnel au niveau d'exigence et de spécificité imposé par le modèle à simuler. Les choix à effectuer pour choisir le bon environnement sont systématiquement des compromis entre la facilité, la souplesse et la performance.

Selon les applications auxquelles sont destinées les simulations, plusieurs techniques de développement existent parmi lesquelles on peut citer :

- **le calcul numérique** qui permet de résoudre des classes de problèmes mathématiques classiques et d'écrire des applications simples basées sur des algorithmes standards (manipulation de matrice, résolution de systèmes d'équations différentielles, calculs aux éléments finis, linéarisation-développement…). Des environnements dédiés (SciLab (http://www.scilab.org),

Abaqus (http://www.abaqus.com), R (http://www.r-project.org)) offrent un cadre adapté à ce genre de traitement.

- **L'image de synthèse** qui propose des techniques pour plonger dans un espace 2D ou 3D un objet spatialisé. On se sert de l'image de synthèse pour visualiser des scènes virtuelles ou pour représenter des phénomènes physiques. Des librairies (OpenGL, DirectX) basées sur des standards de description offrent les outils de base propre à cette classe de traitement.
- **L'automatisme** dont un des objectifs est de traiter la simulation de systèmes en représentant leurs changements d'état et la dépendance entre ces états. Ces représentations sont effectuées selon des modèles discrets tant en temps qu'en évènements. L'automatisme traite également des problèmes de régulation en temps continu (équations différentielles).
- **Les bases de données et les systèmes experts** qui permettent d'exploiter de grandes quantités d'information à partir d'une base de règles et d'en extraire des faits que l'on ne sait pas obtenir de manière analytique. Dans ce domaine on s'intéresse particulièrement à l'abstraction, la reformulation et la fouille de données (Zucker, 1991).

Un même simulateur peut utiliser plusieurs de ces techniques d'applications selon le type de problème qu'il cherche à résoudre, on parlera alors de système hybride (Bertrand, 2004. Gordon, 1978).

Pour finir ce très rapide tour d'horizon de l'environnement informatique qui va héberger les simulateurs, il convient de mentionner que le matériel qui va effectuer les opérations logiques du programme de simulation possède également son influence et peut même contraindre les choix d'architecture logicielle.

Dans la gamme des matériels disponibles viennent en premier lieu les PC. Ces machines sont très largement utilisées et deviennent de plus en plus performantes autant en puissance de calcul, qu'en mémoire, qu'en capacités graphiques. Tant que le simulateur ne nécessite pas une puissance de calcul exceptionnelle, ces matériels constituent un bon compromis entre le prix et la performance disponible. Aujourd'hui la plupart des PC fonctionnent grâce à un système d'adressage calculé sur 32 bits ce qui limite la quantité de segments de mémoire utilisable, cette limite est entrain d'être repoussée par l'émergence des processeurs 64 bits.

Lorsque la simulation requiert des capacités que ne proposent pas les PC, il convient de se tourner vers des calculateurs adaptés. Dans le cas des calculs lourds pour lesquels on attend un temps de réponse court (type modèle météo ou bien financiers), un recours aux super calculateurs est envisageable. Pour ce qui concerne le calcul et l'affichage en temps réel, les stations graphiques offrent les performances requises. Ces deux types de machines constituent des investissements financiers importants que ne peuvent pas se permettre n'importe quel utilisateur, limitant d'autant leur champ d'implantation.

Une technique émergente et alternative aux super calculateurs consiste à mettre en réseau un grand nombre de machines afin de mettre en commun leur puissance de calcul pour atteindre un même objectif (grille informatique). On constitue ainsi une machine virtuelle dont la puissance de calcul est d'autant plus grande que le nombre de contributeurs l'est également. Pour pouvoir s'adapter à un calcul en grille, un modèle et son simulateur doivent accepter un découpage de type parallèle ce qui contraint la manière d'envisager la simulation.

Le bus logique permet également de partager les ressources de calcul entre plusieurs machines. Dans cette configuration, les machines connectées au bus proposent leurs services pour traiter une liste de fonctions ou d'objets. Quand le simulateur appelle une fonction, cette fonction peut s'exécuter sur n'importe quelle machine connectée au bus logique et qui propose les services adéquats.

Nous ne nous intéresserons pas à la partie matériel dans les choix d'implémentation de simulateurs.

1.2.1.2. Classes de simulateurs

Nous présentons ici différentes approches qui permettent de simuler un modèle.

La simulation quasi continue, où le système se présente sous la forme d'un système d'équations (souvent différentielles) à résoudre. Elle permet de suppléer à la résolution analytique quand celle-ci est impossible. Effectuée au départ sur des calculateurs analogiques, elle s'est effectuée aussi sur des ordinateurs ainsi que des machines hybrides, et un troisième type de calculateurs qui n'a pas eu de lendemain, les calculateurs stochastiques.

La simulation à évènements discrets (Leroudier, 1980). Dans ce formalisme, le système est représenté par un jeu de variables dont le nombre de valeurs possibles est fini. Le changement de valeur de ces variables se produit lors de moments précis. L'ordonnancement de ces évènements nécessite un échéancier piloté par un contrôleur central. Une combinaison de valeurs des variables représente un état du système. Un ensemble d'états sont définis pour un système donné. Les transitions entre états sont les évènements, ils sont datés. Dans ce type de simulation, le système est soumis à une succession d'évènements qui le modifient. Ces simulations ont vocation à appliquer des principes simples à des systèmes de grande taille. La simulation discrète se divise en deux grandes catégories :

Dirigée par horloge : dans cette catégorie on simule à chaque fois le passage d'une unité de temps (ou pas de temps) sur tout le système qui est réévalué globalement, on parle également de gestion du temps dirigée par horloge. Dans ce genre de simulation le choix de la valeur du pas de temps de réévaluation est critique et s'adapte au traitement d'un système dont les changements d'états interviennent globalement de manière régulière et diffuse sur l'ensemble du système (Coquillard, 1996). Les grammaires constituent un exemple classique d'application de simulation dirigée par horloge (Lindenmayer, 1990).

Une grammaire formelle (ou, simplement, grammaire) est constituée de quatre éléments :

-Un ensemble fini de symboles, appelés *symboles terminaux* (qui sont les « lettres » du langage), notés conventionnellement par des minuscules.

-Un ensemble fini de symboles, appelés *non-terminaux,* notés conventionnellement par des majuscules.

-Un élément de l'ensemble des non-terminaux, appelé *axiome,* noté conventionnellement S.

-Un ensemble de *règles de production*, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux ; par exemple, A → ABa.

Appliquer une règle de production consiste à remplacer dans un mot une occurrence du membre de gauche de cette règle par son membre de droite ; l'application successive de règles de productions s'appelle une dérivation. Le langage défini par une grammaire est l'ensemble des mots formés uniquement de symboles terminaux qui peuvent être atteints par dérivation à partir de l'axiome.

Dirigée par évènements : dans cette catégorie on calcule l'arrivée du prochain événement, et on ne simule qu'événement par événement, ce qui permet souvent des simulations rapides, bien qu'un peu plus complexes à programmer. Les automates constituent un exemple classique de simulation dirigée par évènements.

La simulation à évènement discrets présente plusieurs avantages. Tout d'abord elle permet de se focaliser sur les événements par ordre d'importance et permet de découper les traitements de la simulation par blocs fonctionnels imbriqués ou tout du moins hiérarchisés. On dispose ainsi d'une approche d'aide à la modélisation souple. Elle est bien adaptée aux traitements par ordinateur et propose naturellement un traitement quasi-parallèle. Par contre ce type de simulation est couteux en ressources informatiques tant pour son traitement que pour assurer la convergence de ses estimateurs. Le résultat n'est bien souvent pas prédictible autrement qu'en effectuant la simulation. En particulier dans le cas des processus stochastique toutes les étapes doivent être simulées.

La simulation par agents : la simulation est segmentée en différentes entités qui interagissent. Elle est surtout utilisée dans les simulations économiques et sociales, où chaque agent représente un individu ou un groupe d'individus. Par nature, son fonctionnement est asynchrone.

1.2.2. Simulateurs de plante

Nous donnons ici un panorama général des différentes classes de simulateurs de plante qui proposent en sortie une apparence géométrique de la simulation et donc une description individuelle des organes qui la composent. Nous mettons volontairement de coté les simulateurs de type purement agronomique (appelés en anglais Process Based Model) qui, s'ils peuvent fournir des résultats très précis en termes de production de biomasse, occultent l'aspect architectural de l'organisation végétale.

Nous avons classé ces simulateurs en trois catégories qui s'approchent de plus en plus de représentations botaniquement correctes et qui intègrent de plus en plus de connaissance écophysiologique. Dans chacune de ces classes nous avons choisi de décrire quelques simulateurs représentatifs.

1.2.2.1. Simulateurs géométriques

OnyxTree (Bosanac et Zanchi, 2002)

Ce logiciel est orienté pour des applications de type image de synthèse. Il propose la modélisation très intuitive par catégories de plante (feuillus, conifères, bambous, fleurs) et pour chacune d'elle permet de régler l'apparence de la plante par ordre de ramification. L'approche en est essentiellement géométrique et fait peu de place à l'organisation. Les notions de pousse, d'acrotonie ou de mélange n'y sont pas abordées, restreignant d'autant le champ d'application du simulateur. Pour finir, la modélisation est effectuée pour un âge

donné, empêchant ainsi toute idée de croissance. Le logiciel est commercialisé avec une base de données de quelques centaines de plantes dont l'apparence satisfait les utilisateurs qui ne sont pas attachés au détail botanique.

Figure 1. Exemples d'arbres produits par OnyxTree.

XFrog (Deussen 2003)

Basé sur le formalisme L-System, il permet de modéliser quasiment n'importe quelle espèce de plante mais malheureusement se base essentiellement sur la géométrie et propose peu d'outils basés sur la connaissance biologique des plantes. L'outil, d'après ses créateurs, demande un fort niveau d'utilisation pour produire une plante même très simple.

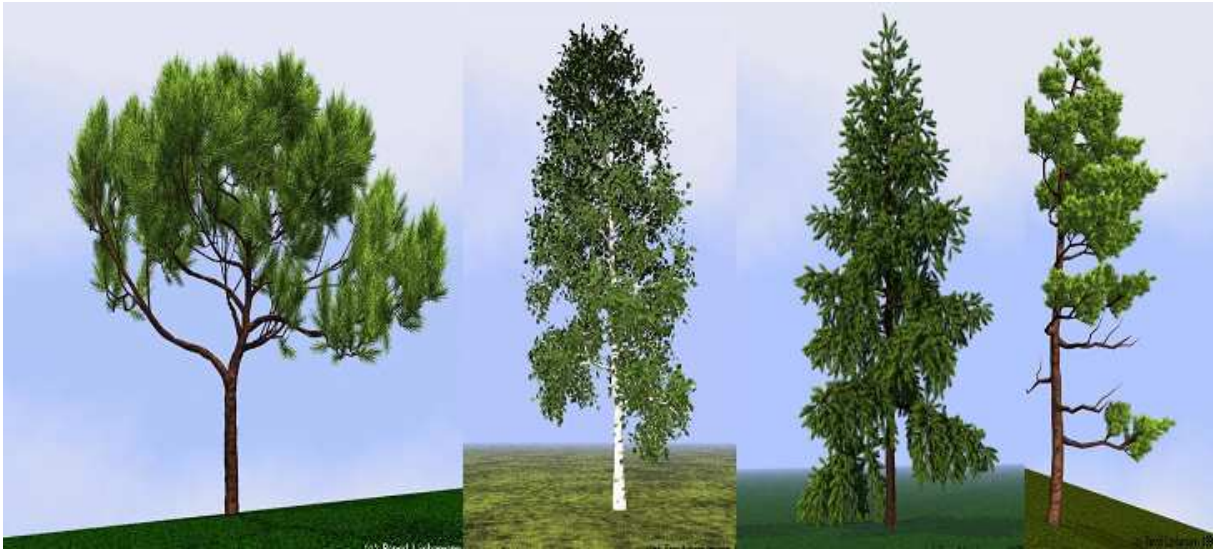


Figure 2. Exemples d'arbres produits par XFrog.

1.2.2.2. Simulateurs botaniques

Amap (de Reffye, 1988)

Issu du laboratoire éponyme du Cirad, ce modèle repose sur une approche relativement similaire à celle proposée par OnyxTree. Par contre, la composante botanique y est plus présente et la notion de croissance y est abordée en tant que paramètre du simulateur. Les sorties de ce modèle sont validées d'un point de vue botanique. Deux versions du modèle de croissance ont été créées. La première a été développée par M. Jaeger et implémente stricto sensu la connaissance botanique proposée par Hallé, 1970 et améliorée par de Reffye, 1979. Ses applications résident essentiellement dans l'image de synthèse, une suite de logiciels permettent de mettre en scène les maquettes produites par la simulation et proposent un outil dédié au paysagisme. L'ensemble de ces outils à été externalisé dans une société à but commercial nommée Bionatics. Une seconde version du modèle a été développée dans le cadre de la thèse de F. Blaise (1991). Cette version possède la particularité de simuler un pseudo-parallélisme du fonctionnement des méristèmes de la plante (ce qui n'était pas le cas de la première version pour des considérations de performance). Cet aménagement a permis de mettre en place les premiers essais d'interaction entre la géométrie et la topologie de la plante ou entre la plante et son environnement.

Figure 3. Exemples d'arbres produits par Amap.

GroImp (Kniemayer, 2004)

Issu du laboratoire de foresterie de Göttingen, ce simulateur étend le modèle de croissance de plantes proposé par Prusinkiewicz. Le moteur de croissance est basé sur un formalisme de type L-system qui a été progressivement enrichi de fonctions adaptées à la simulation de la croissance des plantes. En particulier les notions de Relational Growth Grammar et Sensitive Growth Grammar offrent un cadre formel très puissant pour tester dynamiquement des hypothèses concernant la simulation de la croissance des plantes tant au niveau topologique que géométrique ou bien fonctionnel. En particulier, cet environnement propose des fonctionnalités dédiées à la description en mode « objet » des blocs fonctionnels d'un modèle. Nous avons ici affaire à un environnement qui relève plus du domaine des langages de programmation que de la simulation de la croissance des plantes en tant que telle. Par défaut aucun modèle de plante n'existe dans cet environnement. Le modélisateur doit tout d'abord écrire les règles du modèle de croissance qu'il veut tester pour ensuite le paramétrer avant de le simuler.

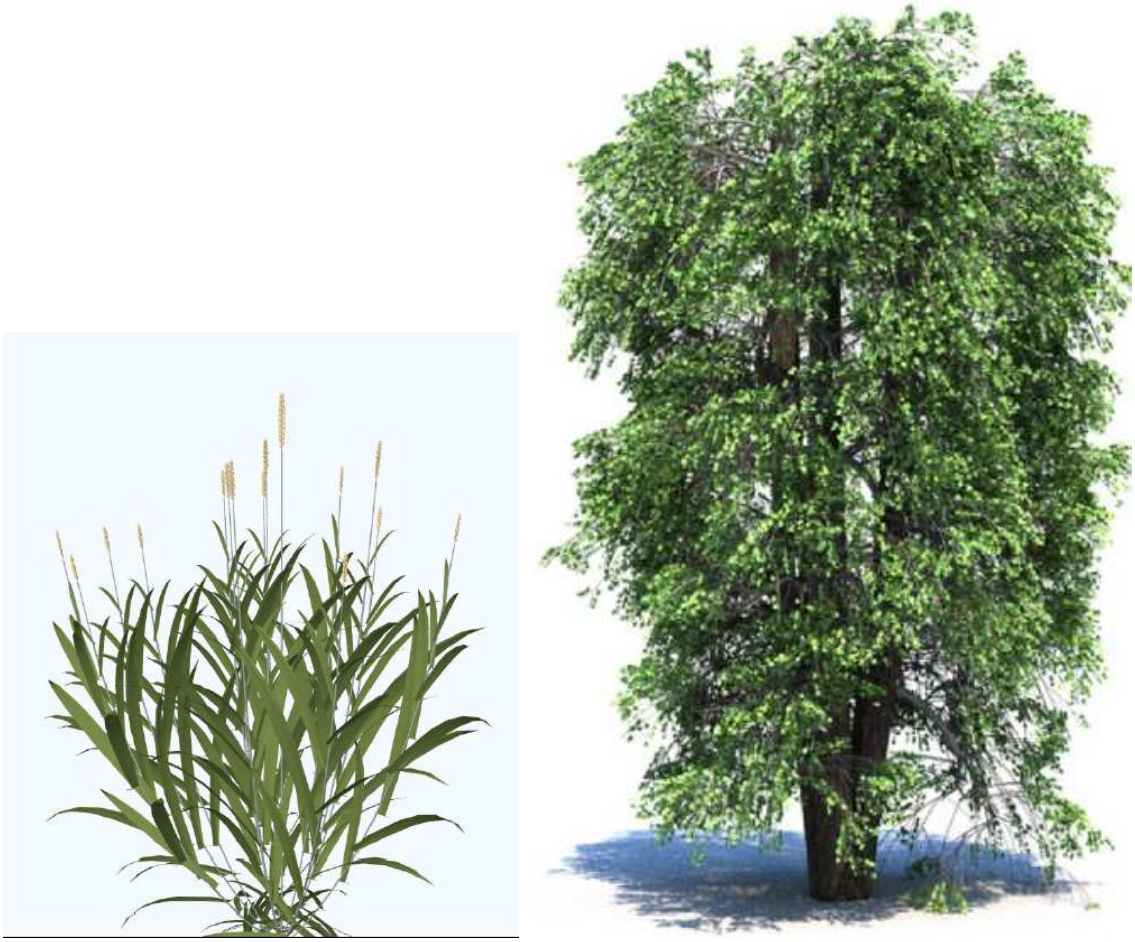


Figure 4. Exemples d'arbres produits par GroImp.

1.2.2.3. Simulateurs écophysiologiques

La production végétale est le résultat final du processus de croissance de la plante, et l'optimisation de sa valeur économique est le but de la recherche agronomique. La récolte utile concernera différents organes selon l'espèce considérée : feuilles, fruits, racines, tubercules, tiges, troncs qui sont autant de récoltes partielles et spécialisées, prélevées sur l'architecture de la plante. Le but des itinéraires culturaux et de la sélection est d'optimiser sur une espèce donnée la production de l'organe végétal utile, en quantité, mais également en qualité. Un même poids de récolte peut correspondre à des nombres d'organes variables associés à des calibres différents qui donneront autant de valeurs économiques. Si l'on disposait d'un modèle de croissance et d'architecture fiable, où toute la structure géométrique serait décrite avec tous les organes selon leurs volumes et leurs poids, la production serait exactement connue. Tout le problème consiste donc à savoir si un tel modèle est nécessaire et s'il est possible de l'obtenir théoriquement.

Dans un premier temps les agronomes ont, à l'instar des informaticiens, préféré contourner la difficulté en créant des modèles empiriques, qui par calibration sur des observations globales de la plante (nombre d'organes, poids, surface foliaire,…) donnent une cinétique de la croissance en fonction des paramètres de l'environnement (lumière,

température, …). Ici les organes ne jouent aucun rôle individuellement mais sont rassemblés en compartiments qui deviennent sources (surface foliaire) et puits (feuilles, tiges, fruits, racines). L'organogenèse et la photosynthèse ne sont pas couplées bien qu'elles progressent ensemble (figure 5).

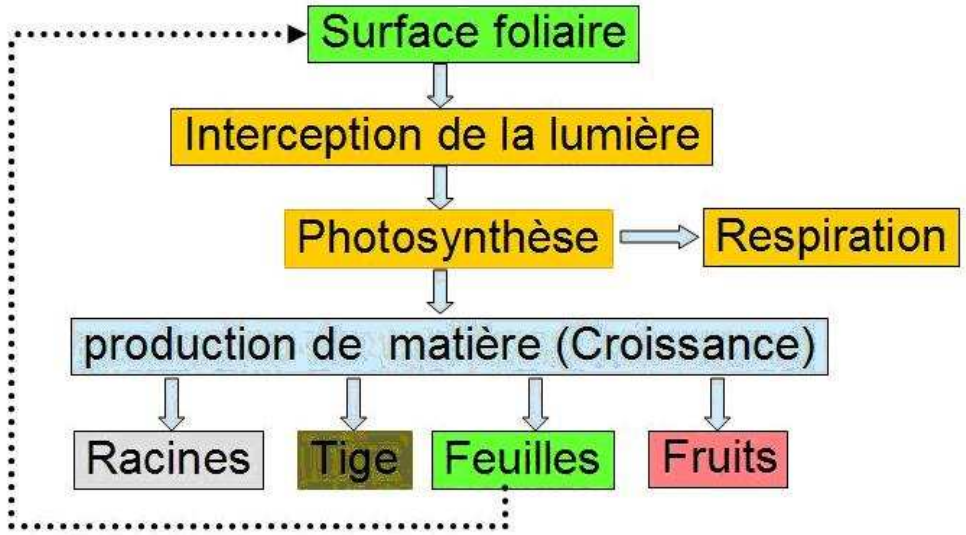


Figure 5. Principes de la simulation de la production végétale chez les modèles agronomiques (Heuvelink, 2004)

En conditions expérimentales contrôlées ces modèles peuvent donner de bons résultats dans le suivi de la production, grâce à la connaissance physiologique du fonctionnement végétal qu'ils intègrent, bien que l'architecture comme support du fonctionnement soit ignorée. Le pas de temps pour intégrer les paramètres de l'environnement est souvent court, de l'ordre de l'heure, ce qui peut donner lieu à de nombreuses itérations sur un cycle cultural. La photosynthèse est calculée par une estimation de l'interception de la lumière par la surface foliaire. La matière produite, respiration végétale déduite, est alors distribuée dans les compartiments qui rassemblent les organes produits par catégories. Ici, c'est la connaissance physiologique qui prime avec la mise en œuvre des lois simples et empiriques directement mesurées par des régressions statistiques sur les données expérimentales. Les programmes informatiques qui simulent la production végétale, sont donc de facture simple sans formalisme particulier et à la portée des chercheurs non mathématiciens ou informaticiens.
De Ceres pour le maïs (1986), jusqu'à Tomsim (2000) pour la tomate, de nombreux logiciels spécifiques basés sur ce principe ont ainsi vu le jour. Mais leur limitation est vite apparue par leur incapacité de suivre convenablement la phénologie de la plante et sa plasticité : c'est-à-dire l'évolution du fonctionnement architectural qui change progressivement dans son organogenèse et ses relations sources puits, avec des modifications importantes de la forme et de la dimension des organes. C'est pourquoi les agronomes après avoir exploré les limites de l'approche empirique des modèles de production se tournent de plus en plus vers des modèles qui intègrent l'architecture en tant que support du fonctionnement.

Aujourd'hui, les recherches s'orientent principalement vers des modèles qui prennent en compte le couplage de l'organogenèse et de la photosynthèse, en faisant fonctionner l'architecture et en donnant aux organes (bourgeons, feuilles, entre-nœuds, fruits, racines)

leurs véritables rôles fonctionnels. De tels modèles qui sont une réunion des modèles architecturaux de l'informatique et des modèles fonctionnels de l'agronomie ont reçu le qualificatif de « structure-fonction » (Figure 6) (Sievanen, 2000).

Mais il apparaît des écueils importants dès que l'on considère des architectures complexes comme celles des arbres :

- La fusion et la synthèse des connaissances nécessitent une coopération pluridisciplinaire car aucune des sciences en présence (Botanique, Agronomie, Informatique, Mathématiques) ne possède l'ensemble des compétences nécessaires pour la réaliser toute seule. Comment adapter de façon pertinente les acquis des différentes disciplines pour les faire cohabiter dans un même modèle ?
- Comment vont se comporter les modèles structure-fonction si à la lourdeur des calculs liés au fonctionnement de l'organogenèse, on ajoute celle des relations sources-puits qui créent et distribuent la matière dans l'architecture à chaque pas de temps ?
- Comment parvenir à des simplifications pertinentes de façon à obtenir un modèle général avec un nombre réduit de paramètres, et ainsi pouvoir calibrer celui-ci sur les données expérimentales ?
- Finalement, quel formalisme adapter pour décrire de tels systèmes complexes et surtout pour les analyser ?

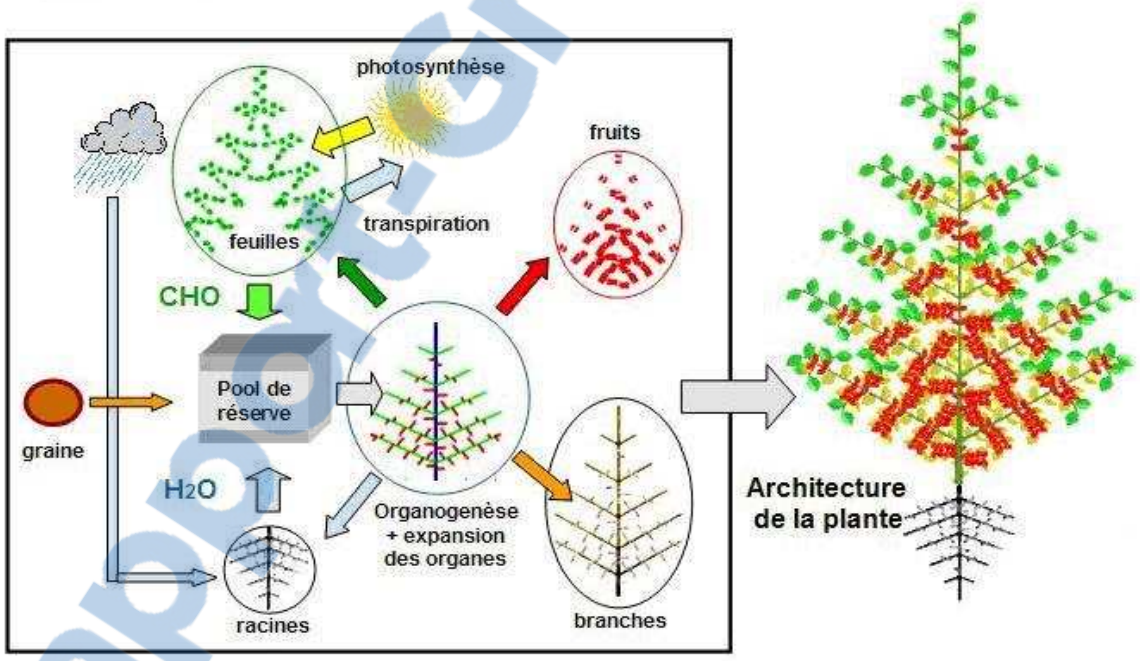


Figure 6. Principes de la simulation de la morphogénèse des plantes chez les modèles structure-fonction (deReffye, 2008)

Vica (Wernecke 2000)

Ce simulateur est très fortement orienté vers une programmation de type objet. Il propose une structure d'objets principaux qui permettent une description structurée de la plante. Chacun de ces objets possède des attributs qui lui sont propres et qui permettent d'une part de décrire leur géométrie et d'autre part leur capacité à produire de la matière et à l'échanger avec les autres objets décrivant la plante.

Les attributs décrivant la géométrie sont des triangles positionnés dans l'espace selon des règles empiriques. La cinématique d'apparition des éléments peut être contrôlée par le fonctionnement physiologique de la plante et leur âge compté en somme de température par unité de temps. Chaque objet de plante possède de plus des propriétés optiques qui permettront le calcul de la photosynthèse.

Les attributs décrivant les propriétés physiologiques concernent d'une part la capacité de l'élément à produire de la matière (photosynthèse) et d'autre part à séquestrer, consommer ou bien transmettre cette matière aux autres éléments de la plante. La matière détenue par chaque élément peut être séparée en plusieurs catégories qui auront chacune leurs propriétés de production, séquestration et transmission. On peut ainsi séparer les cycles du carbone, de l'azote et de l'eau.

La photosynthèse est basée sur le principe d'efficience de la lumière, des algorithmes de transfert radiatifs sont proposés pour calculer la quantité de lumière interceptée par chaque triangle de la plante (Ross and Marshak 1991 ; Soler *et al.* 2003 ; Andrieu *et al.* 1997 ; Jacquemoud *et al*. 1996).

Ce simulateur a été essentiellement utilisé pour des applications sur des graminées. Un orge virtuel a été programmé dans cet environnement pour tester sa capacité de production en fonction de différentes conditions climatiques de température et d'apport en azote. La plante est divisée en trois classes d'objets (tige, feuille, racine). La géométrie des éléments de la plante est fixée selon des mesures expérimentales pour les feuilles (on découple donc la dimension des feuilles de la quantité de matière qu'elles reçoivent !) et selon une loi dépendante de l'âge pour la tige. Le système racinaire ne possède pas de géométrie. La vitesse d'apparition des organes est contrôlée par le temps thermique, le nombre d'épis est couplé à la masse en carbone des organes. Les paramètres physiologiques de production, consommation, transmission sont les mêmes pour tous les objets d'une même classe. Les lois d'échange sont dédiées entre les différents types d'organes. Ce modèle simplifié a pu être calibré sur des données de terrain par méthode d'inversion.

Le modèle Vica semble très souple pour ce qui concerne la définition des modalités de production et de partage des différents flux à l'intérieur de la plante. On peut cependant imaginer que la généricité du modèle Vica de base place la responsabilité de la définition de la géométrie et de la calibration des paramètres retenus sous la responsabilité du modélisateur.

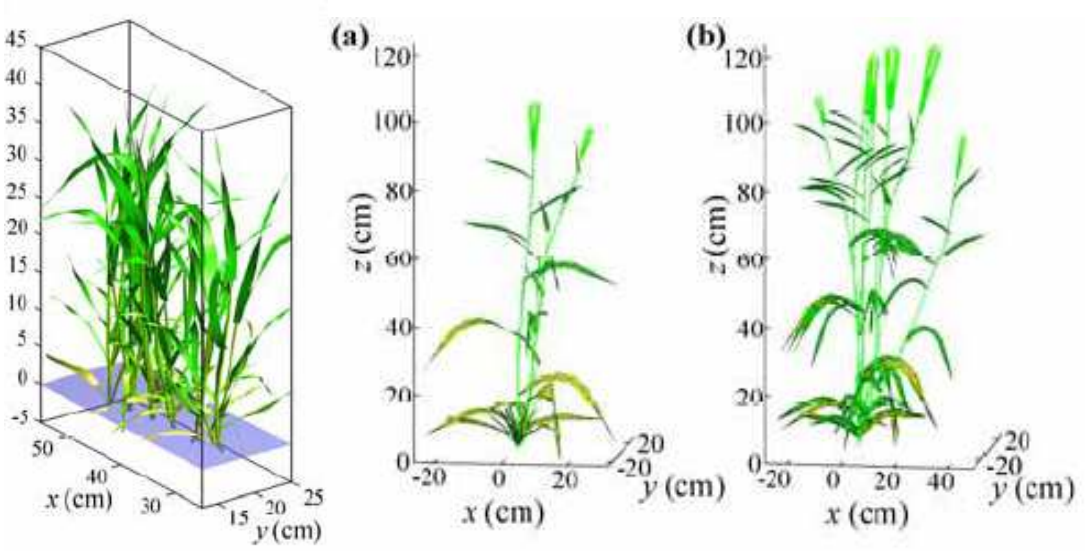


Figure 7. Exemples de plantes produites par VICA.

DigiPlante (Cournède *et al.*2006)

Ce simulateur est basé sur le modèle GreenLab (Yan 2004) et propose une approche générale qui inclut tant l'aspect architectural qu'écophysiologique. Il propose de plus un outil de calibration des paramètres qui contrôlent la production et la répartition de la matière. Le modèle mélange d'une part un formalisme de représentation du développement de la plante par un automate dont le changement d'état est règlé par une distribution discrète de phases type (Neuts, 1975) et d'autre part un modèle de photosynthèse/allocation qui permet de calculer la dimension des organes et éventuellement interagir avec leur nombre.

GreenLab décrit l'organogenèse à l'aide d'un système d'automates à double échelle, déterministe ou stochastique. Celui-ci contrôle, à partir des bourgeons, la fabrication des unités de croissance et des phytomères qui les composent. Les états de cet automate qui représentent l'âge physiologique caractérisent les UC (macro-état) et les phytomères (micro-états) comme décrit dans Zhao 2001. Chaque état contient tous les renseignements de types fonctionnels ou géométriques pour créer l'élément botanique correspondant. Un bourgeon peut rester dans un même état selon une loi d'occupation ou changer d'état (métamorphose) selon une loi de transition. Il met en place des phytomères avec leurs bourgeons axillaires d'âges physiologiques différents qui correspondent à autant de transitions selon les règles botaniques observées. Le formalisme des relations entre états dans GreenLab est typiquement celui des chaînes de « Markov ». Dans un but de simplicité, certains phénomènes botaniques tels que le polycyclisme ou bien la dynamique de démarrage des axes ne sont pas pris en compte (figure 8).

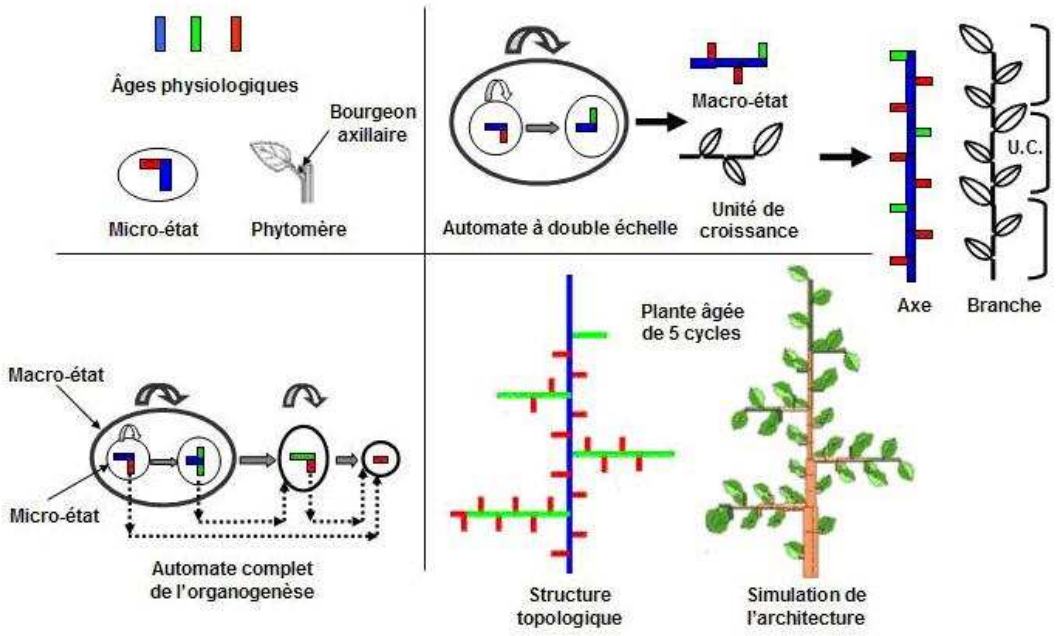


Figure 8. Modélisation de l'organogénèse dans GreenLab : automate et âge physiologique.(Zhao, 2001)

GreenLab propose un modèle de photosynthèse basé sur le principe d'efficience de la lumière et simule l'interception du rayonnement atténué dans une canopée par analogie avec la loi de Beer Lambert (Monsi, 1953). Le partage de la matière fabriquée est effectué à partir d'un pool commun selon une règle de proportionnalité entre les différentes valeurs de puits affectées aux organes de la plante. Ceux-ci sont de quatre natures : feuilles, bois, racines et bourgeons. Les feuilles produisent de la matière, leur surface est proportionnelle à leur masse. Le bois peut accumuler de la matière dans son cœur puis dans ses cernes, le rapport diamètre/longueur des éléments de branche est déterminé selon une loi empirique. Les racines sont présentes uniquement en tant que puits. Les bourgeons captent de la matière mais ne peuvent produire une pousse que s'ils en ont accumulé suffisamment selon un seuil. Le nombre d'organes produits dépendra également de la quantité de matière accumulée (Matthieu 2006). L'ensemble de la matière produite est entièrement consommée entre chaque cycle, le modèle ne considère ni la respiration d'entretien ni la mobilisation/relâchement de matière par le bois de la plante. On peut cependant reproduire une croissance rythmique ou bien l'apparition de réitération uniquement par réglage des paramètres écophysiologiques.

Quand l'automate a une production sûre, c'est-à-dire quand ses probabilités de réalisation valent 1 ou 0, on peut calculer le nombre d'organes présent dans la plante et leur masse respective à tout moment sans le simuler. Si l'automate n'est pas à réalisation sure, on peut alors prédire la moyenne et la variance de sa production tant en nombre d'organes qu'en masse de ceux-ci. Cette capacité de calcul analytique provient de « bonnes » propriétés de la formulation du modèle qui permettent de l'exprimer sous forme d'équations récurrentes.

Contrairement aux autres modèles implémentés par les simulateurs de la classe FSPM, GreenLab reprend à son compte les choix des Process Based Model pour exprimer le

fonctionnement (pool commun, interception de la lumière utilisant la loi de Beer…). La construction géométrique de la plante n'est pas nécessaire dans ce simulateur.

Digiplante propose une interface conviviale de saisie de ses paramètres et propose également des outils pour leur calibration. Cependant le système est totalement fermé et contraint l'utilisateur à globalement accepter ou bien refuser les composantes du modèle qu'il implémente.

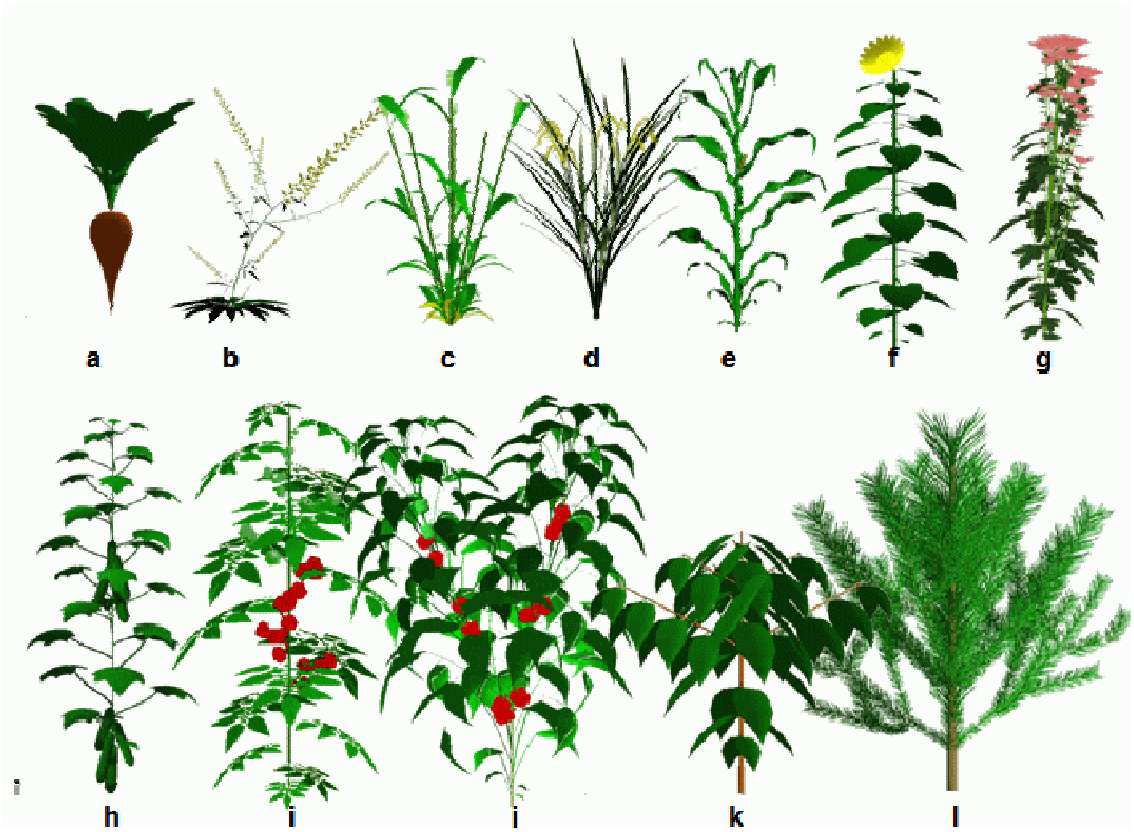


Figure 9. Exemples de plantes produites par DigiPlante.

Lignum (Perttunen, 2001)

Ce simulateur a été développé pour des applications aux espèces forestières cultivées dans les forêts du nord de l'Europe. Son architecture a été adaptée et paramétrée pour différentes espèces telles que le pin sylvestre, l'érable à sucre ou le pin noir. Un modèle de photosynthèse basé sur l'interception lumineuse permet de calculer la production de biomasse au pas annuel. Une balance du carbone est calculée qui inclut la production, la respiration, l'accroissement et la sénescence des feuilles, éléments de branche et racines. La répartition de matière est effectuée selon le modèle de Makela, 1997 qui propose un pool commun de production des feuilles diminué de leur respiration d'entretien. Ce pool commun est partagé entre tous les éléments de la plante selon une proportion de puits. Chaque élément possède également une perte de masse par mortalité cellulaire. Le rapport diamètre/longueur des éléments de branche est ajusté selon le modèle de Shinozaki (Shinozaki et al 1964). Le moteur de croissance est basé sur un formalisme à base de L-system qui permet d'empiler des éléments de plantes selon les règles fournies à l'interpréteur. Chaque élément de plante est composé d'un segment portant éventuellement une feuille et des bourgeons axillaires. Ces éléments sont placés dans l'espace selon des règles géométriques empiriques. La description

de l'arbre est très structurée et permet grâce au formalisme L-system d'inclure des traitements spécifiques pour décrire les interactions entre son développement et sa physiologie ou sa géométrie.

L'environnement de simulation propose donc une description structurée d'arbre sur laquelle un ensemble de traitements de simulation de la photosynthèse et de l'allocation est prévu. Le développement de la plante est effectué grâce à l'interprétation d'un jeu de règles de type L-system qui mettent en place des éléments de plante tant au niveau topologique que géométrique. Ce formalisme permet d'étendre un noyau de simulation à des applications spécifiques qui permettent d'affiner les modalités de croissance. Le défaut de cette approche est que l'utilisateur doit écrire les règles puis donner les valeurs à leurs paramètres pour décrire une espèce particulière. De ce fait, à notre connaissance, un nombre restreint d'espèces à pu être décrit.

Figure 10. Simulation du pin sylvestre par Lignum.

L-peach (Allen, 2004)

Ce simulateur est entièrement dédié à la simulation de l'architecture et de la production de fruits du pécher. Il s'appuie sur un formalisme Open L-system (Mech, 1996) et décrit une architecture qui ressemble géométriquement au pécher mais qui manque de finesse quant aux détails botaniques de cette espèce (absence de polycyclisme, pas de développement anticipé, nombre de feuilles grossièrement approximé…). Il propose un modèle de photosynthèse, de respiration d'entretien et de partage de la production d'assimilats. La production de biomasse est asservie d'une part à l'interception de la lumière et d'autre part à la disponibilité en eau (au travers d'une masse de racines). Le partage des ressources est effectué selon un équilibre source-puits-résistances. Le modèle prévoit la réaction du développement aux balances source-puits. En particulier, il permet l'abscission quand l'offre est trop faible, la préparation du stock des entre-nœuds préformés et la fabrication des entre-nœuds néoformés quand l'offre est suffisante. De plus le bois est capable de stocker de la biomasse qu'il libère à de6 moments choisis par l'utilisateur. On a donc affaire à un réel simulateur FSPM qui mêle les interactions entre croissance et fonctionnement. Cependant il

est entièrement dédié au pécher et la calibration de ses paramètres est effectuée manuellement.

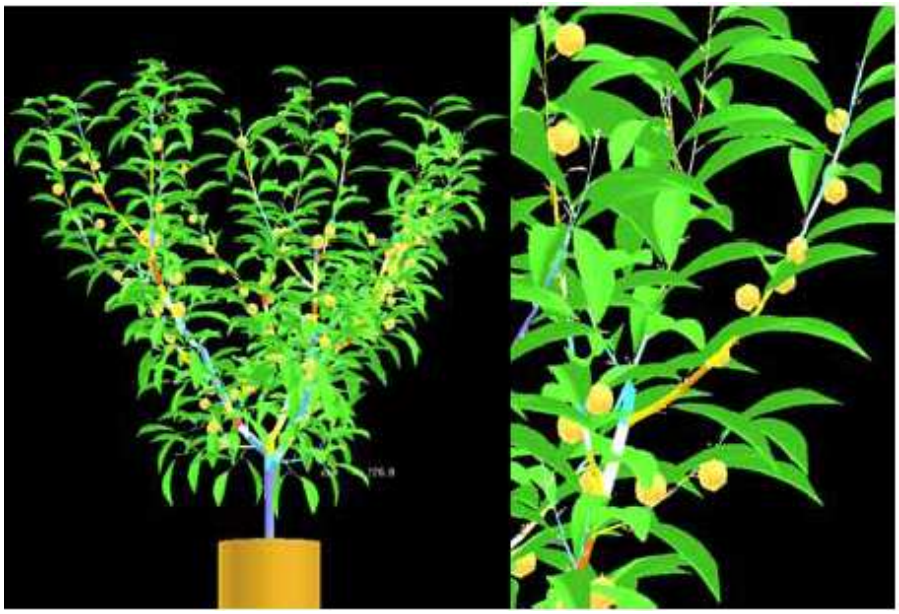


Figure 11. Simulations du pêcher par L_peach.

Cotons (Jallas 2003)

Ce simulateur est basé sur le modèle GOSSYM (Baker 1989). Il est entièrement dédié à la simulation de la culture du cotonnier en plantation. Il prend en compte tant des aspects architecturaux qu'écophysiologiques qu'agronomiques que pédologiques ou que phytosanitaires. Il intègre une connaissance très pointue de cette espèce et est utilisé dans le cadre d'applications agronomiques. Plusieurs modèles simulant l'organogénèse, la photosynthèse et la géométrie sont interconnectés. Nous avons affaire ici à un exemple type de simulateur qui intègre de la connaissance dans un but précis et dont on peut difficilement imaginer de l'appliquer facilement à d'autres espèces.

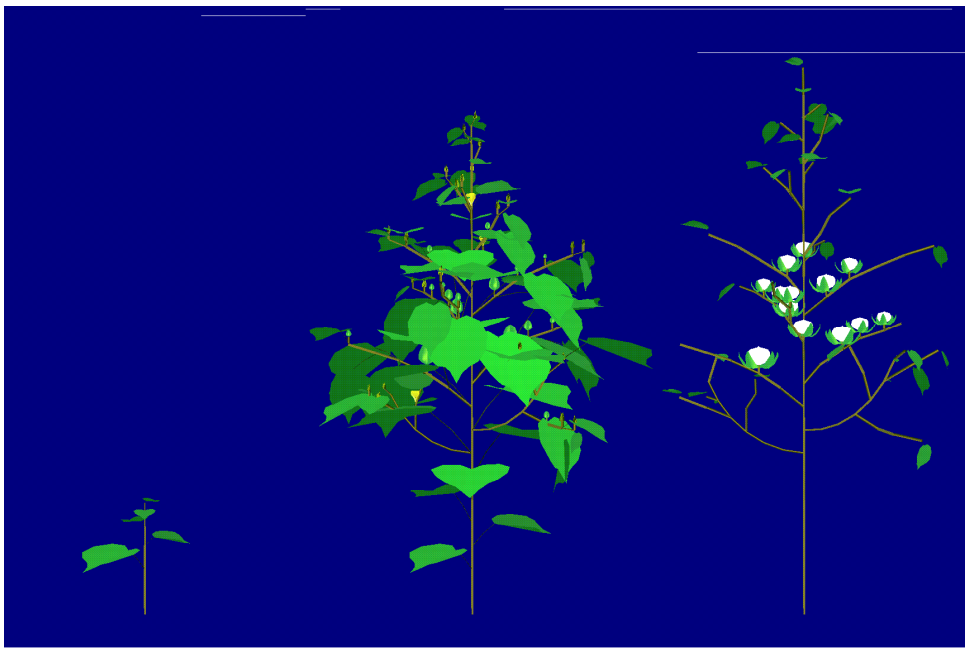


Figure 12. Croisssance de cotonnier par Cotons.

1.2.2.4 Conclusion sur les simulateurs de plante

Ce tour d'horizon nous montre différentes tendances dans le domaine de la simulation de l'architecture des plantes. Chacune des approches présentées offre des avantages et des inconvénients.

Les simulateurs purement géométriques permettent d'obtenir des formes végétales dont le réalisme est obtenu au travers des outils de l'image de synthèse (textures, shading…) mais dont l'organisation au sein de l'architecture n'est pas forcément conforme aux lois de la botanique. Certains présentent une interface de modelage très ergonomique et intuitive, d'autres offre la possibilité d'inclure de la connaissance supplémentaire et tous proposent plusieurs formats de sortie propres à assurer l'interface avec des logiciels d'image de synthèse. En aucun cas leurs sorties ne sont prévues pour être utilisées dans un cadre d'application agronomique.

Les simulateurs qui proposent une approche botanique permettent de produire des maquettes de plantes possédant une caution scientifique et donc peuvent non seulement produire des maquettes à l'usage de la visualisation mais également s'inclure dans des applications qui requièrent des plantes virtuelles réalistes. Leur programmation est destinée à des spécialistes connaissant les plantes et nécessite éventuellement une bonne connaissance de la programmation informatique dans le cas des simulateurs basés sur les L-system. On a affaire à des systèmes fermés de type Amap ou bien des architectures ouvertes de type GroImp.

Les simulateurs qui incluent de la connaissance écophysiologique mêlent les aspects développement avec le fonctionnement, ils permettent de calculer la biomasse produite par la photosynthèse et de la répartir dans l'architecture simulée. Certains permettent de simuler une rétroaction de la photosynthèse sur le développement de la plante. D'une manière générale leur calibration se révèle difficile voire impossible. Certains sont spécialisés dans la simulation d'une espèce particulière, d'autres proposent une approche plus générale mais sont organisés en architecture fermée et les autres ne proposent pas de formalisme préformaté pour la description générique d'une architecture végétale. De plus, les choix qui sont effectués au niveau des modèles de production et de partage des assimilats diffèrent d'un simulateur à l'autre sans qu'on puisse à priori retenir l'un plutôt que l'autre.

1.3 Problématique

Il semble qu'il n'existe pas d'architecture logicielle proposant à la fois une implémentation de modèle générique d'architecture de plante et un environnement de développement de son simulateur qui soit ouvert à des extensions dynamiques.

La présente thèse essaie de combler cette carence en proposant la combinaison d'un modèle de description d'architecture végétale, d'un environnement propre à accueillir des simulateurs de plantes et d'une implémentation de ce modèle dans cet environnement.

Dans un premier chapitre, nous présenterons le modèle AmapSim qui propose une description de l'architecture des plantes essentiellement basée sur l'utilisation de processus

stochastiques pour représenter l'organisation topologique des plantes (position relatives de branches, décomposition des branches en pousses annuelles, cycles, zones et entre-nœuds). La topologie ainsi engendrée est ensuite habillée grâce à des fonctions géométriques simples qui permettent de placer les composants de la plante dans un espace tridimensionnel. Ce modèle s'attache à une description purement structurelle des plantes sans chercher à proposer des mécanismes expliquant le fonctionnement des parties actives de la plante et qui engendrent sa structure. Les maquettes que peut produire ce modèle possèdent une finesse botanique exceptionnelle et prend en compte des phénomènes détaillés tels que préformation-néoformation, polycyclisme, développement immédiat ou retardé…

Dans un second chapitre, nous proposerons une architecture logicielle (Vitis) propre à accueillir des simulateurs de croissance d'architecture végétale. Cet environnement permet la gestion d'une structure branchée, le séquencement de traitements selon un principe à évènements discrets, l'adjonction dynamique et la connexion de modules fonctionnels entre eux, la gestion de jeux de paramètres d'entrée de modèle et la sortie de résultats de simulation.

Dans un troisième chapitre nous exposerons l'implémentation de AmapSim dans Vitis et exposerons les étapes de traduction du modèle en termes de simulation. Nous exposerons en particulier l'utilisation des mécanismes d'abstraction homomorphe que propose Vitis pour ce qui concerne la représentation de la plante. Nous donnerons ensuite un panorama des possibilités de ce simulateur et des applications qu'on en a tirées.

Dans un quatrième chapitre nous montrerons comment le développement informatique d'un simulateur de plante dans Vitis (en l'occurrence AmapSim) permet de tester divers modules fonctionnels autour de règles de mise en place de l'architecture. On décrira comment les interfaces logicielles fournies par Vitis permettent de connecter des modules additionnels à AmapSim pour enrichir ses fonctionnalités et en particulier la description du fonctionnement des plantes. Si le développement de la plante (mise en place des organes) se fait selon un schéma botanique couramment admis, il n'en va pas de même par exemple pour les fonctionnalités physiologiques qui concernent la photosynthèse c'est-à-dire l'interception de la lumière, la production et la répartition des assimilats. Il est important d'ouvrir un champ, pour tester et mettre en œuvre diverses hypothèses. On s'appesantit ici sur les méthodes de description du développement de la plante tout en prévoyant l'adjonction d'éventuels outils de simulation du fonctionnement. Nous exposerons plusieurs cas d'utilisation de l'architecture logicielle bâtie autour de la mise en place de l'architecture végétale et montrerons comment elle peut accueillir des modules externes qui simulent différents modèles interagissant avec l'édification de la plante.

2. Le Modèle AmapSim

Ici nous décrivons les notions botaniques nécessaires pour comprendre ce qu'est une architecture végétale vue sous l'angle dynamique de sa mise en place puis nous exposons les outils mathématiques que nous allons utiliser pour bâtir un modèle représentant cette connaissance botanique. Ce modèle reste déconnecté de toute notion de simulation informatique bien qu'il propose un cadre propice à son implémentation dans un programme.

2.1 Choix du niveau d'observation

On peut distinguer plusieurs échelles d'observation de la plante qui ont donné naissance à autant de modèles :

- La cellule : le fonctionnement des méristèmes (divisions, expansions des cellules), le rôle fonctionnels des cellules est concerné.
- L'organe : on observe les caractéristiques des organes de la plante (feuilles, fleurs) pour tirer une classification du monde végétal.
- La plante : l'architecture de la plante en tant qu'assemblage d'organes est décrite selon des structures topologiques et géométriques. On travaille sur des individus.
- Le peuplement : les organes sont regroupés en compartiments par unité de surface de sol. On s'intéresse à la production par unité de surface et non la plante individuelle.

Seul le deuxième niveau, « la plante » sera pris en compte ici et le peuplement sera compris comme un assemblage de plantes et non pas comme un tout avec des propriétés émergentes à ce niveau. De plus ce qui est propre au simulateur AmapSim, qui fait l'objet de cette thèse ne concerne que la Botanique, c'est-à-dire le développement de la plante et non son fonctionnement. La partie développement qui contrôle la mise en place l'architecture sera détaillée ici, elle correspond à un important travail de recherche toujours d'actualité sur la simulation des plantes. Certains modules externes comme GreenLab ont été développés spécifiquement pour simuler le fonctionnement végétal. Ils seront détaillés à leur place dans le document.

2.2. Connaissances biologiques requises

Outils de classification, Age physiologique, catégories d'axes, mode de reproduction...

L'âge physiologique qualifie l'état de différenciation d'un méristème au travers des structures qu'il a engendrées. En croissance libre, on va considérer que l'âge physiologique évolue normalement le long d'un axe d'un stade végétatif initial vers un stade floral terminal. On parle de "vieillissement" ou de "métamorphose" (Goethe, 1790) dans le fonctionnement du méristème. La plupart du temps, il y a une discontinuité entre l'âge physiologique du méristème principal et des méristèmes axillaires. Ces derniers sont plus "vieux", ils ont un âge physiologique plus avancé.

Les pousses ou Unités de Croissance (correspondant aux portions d'axe fabriquées par le méristème terminal entre deux pauses de fonctionnement) se succèdent le long d'un même axe et sont soumises à un processus de différenciation graduel appelé "dérive". Celle-ci n'est autre que l'augmentation de l'âge physiologique du méristème de l'axe végétatif. Ainsi le méristème édificateur du tronc passe généralement par une phase d'établissement initiale (effet de base), puis par une phase de fonctionnement constant (phase linéaire de croissance) enfin par un vieillissement progressif (dérive) qui aboutit à un stade physiologique terminal (U.C. courte souvent florifère). Les axes d'ordre supérieur partent d'un certain degré de différenciation pour aboutir au même âge physiologique ultime. Tout se passe, en première approximation, comme si les bourgeons axillaires naissaient plus vieux que le bourgeon principal. Le cas particulier de la réitération s'explique par une duplication de l'âge physiologique courant du méristème qui édifie le tronc. D'un point de vue architectural, on peut imaginer un "axe de référence" (deReffye 1991), sur lequel toutes les étapes de différenciation possibles de l'U.C. d'un arbre sont décrites en séquence. L'âge physiologique se déroule en étapes successives le long de cet axe. A tout bourgeon axillaire on associe une étape initiale plus viei1le où égale à celle du méristème porteur et une étape finale qui peut être la fin de l'axe. Ainsi apparaît la notion de "saut" en âge physiologique lors du processus de ramification et de "dérive" lors du processus de croissance du méristème principal. On peut imaginer aussi des bouclages sur une étape et des progressions plus ou moins rapides le long de l'axe de référence. Ce principe, qui est une matérialisation de l'âge physiologique, rend bien compte, en première approximation, des principales notions botaniques architecturales (ordre de ramification, réitération, acrotonie, intercalation...) (Barthélémy, 2007).

Description de l'architecture végétale (modèles descriptifs) développement, port d’une plante.

Hallé et Oldeman (1970) ont proposé une classification des arbres tropicaux en "modèles architecturaux" qui repose sur le fonctionnement des méristèmes. Pour le spécialiste, le modèle de RAUH, le modèle de TROLL évoquent des images d'arbres bien définies. Cependant cette classification n'est pas d'un grand secours pour la conception d'un logiciel de simulation de la croissance des végétaux (ou encore moteur de croissance d'arbre).

Des changements progressifs du fonctionnement des méristèmes font passer insensiblement d'un modèle à l'autre. Ainsi les modèles de RAUH et ATTIMS diffèrent par une ramification rythmique chez l'un (ex. hévéa), diffuse chez l'autre (ex. eucalyptus). Du point de vue de la simulation, on passera de l'un à l'autre en changeant seulement la valeur numérique d'un paramètre de ramification. Cela ne justifie pas d'en faire deux classes de modèles au sens numérique. De même le modèle de RAUH et le modèle de MASSART diffèrent par des branches orthotropes chez le premier, plagiotropes chez le deuxième. Du point de vue de la simulation, on modifiera les valeurs numériques des paramètres géométriques (angles de branchements et de phyllotaxie) et mécaniques (module d'élasticité du bois). On peut donc constater qu'un seul et même programme peut simuler toute une variété de modèles, sans qu'il y ait besoin de faire des traitements informatiques particuliers. Le point de vue des modèles de croissance ne se situe pas au même niveau que celui des modèles architecturaux. De plus ces derniers concernent des architectures tropicales, qui sont souvent soumises à des règles de croissance particulières. Les conditions du milieu sont le plus souvent favorables et la plante exprime un rythme de développement endogène. En climat tempéré, l'hiver puis le printemps ont un effet de resynchronisation forcée sur le fonctionnement des méristèmes.

Hallé et Oldeman (1970) ont opéré, de plus, des simplifications légitimes pour les besoins de leur classification, mais en contrepartie ils ont laissé dans l'ombre des éléments essentiels pour renseigner la simulation. Ainsi les modèles architecturaux reposent essentiellement sur des types d'axes décrits comme des entités, alors que la simulation nécessite un fonctionnement feuille à feuille. Certes l'unité de croissance (U.C.) y est intégrée comme un élément essentiel constitutif de l'axe végétatif et lui confère une rythmicité, mais ses spécificités ne sont pas décrites précisément. Or lorsque l'on aborde l'architecture des arbres en détail, on s'aperçoit qu'il existe des particularités incontournables. Ainsi il faudra préférer la notion de "type d'axe", ou si l'on veut d'âge physiologique, à la notion d'ordre de ramification, pour décrire l'architecture. Il faudra introduire la notion de préformation et de néoformation dans la constitution de l'unité de croissance. Enfin, il faudra prendre en compte le polycyclisme et les rameaux à développement immédiat (Barthélémy, 2007). On peut déjà imaginer sans entrer dans les détails une classification obligée des modèles de fonctionnement, basée sur des paramètres qui auront une incidence importante sur la simulation.

Fonctionnement des méristèmes : croissance, mort, ramification

Le processus de croissance d'un axe végétatif est caractérisé par l'ajout de nouveaux entre-nœuds à son extrémité. Pendant la période de croissance (Unité.de Temps et, par extension, l'ensemble des U.C. qui sont mises en place pendant cette unité de temps) il peut y avoir des variations du rythme d'émission des nouveaux entre-nœuds sans que cela change le résultat final. On parlera de dilatation de l'échelle des temps. Le temps qui sépare l'apparition de deux entre-nœuds successifs, ou temps de renouvellement, peut avoir un caractère régulier comme chez le caféier (un nouvel entre-nœud environ tous les quinze jours). La distribution du temps de renouvellement ressemble alors à une loi normale. Il peut être irrégulier et varier entre quelques heures et plusieurs jours comme chez l'abricotier. Dans ce cas la distribution du temps de renouvellement suit une loi exponentielle. On se trouve dans les conditions d'un processus de Poisson. A tout moment une nouvelle feuille peut apparaître avec une égale probabilité.

Les méristèmes cessent de fonctionner pour deux types de raison : soit ils entrent en pause de longue durée après une période de fonctionnement, soit ils meurent ou se transforment. Une pousse végétative qui peut être une U.C. est alors réalisée. Les méristèmes édificateurs produisent à chaque nœud des méristèmes axillaires de diverses natures. Ainsi, classiquement, on trouvera un méristème à l'aisselle d'une feuille pouvant donner un rameau axillaire ou une inflorescence. Le long d'un axe végétatif, on peut trouver, successivement, des séries de nœuds ramifiés et des séries de nœuds non ramifiés (ex. abricotier). Des séries de rameaux alternant avec des séries d'inflorescences (ex. bégonia) ... Ces séries ont très souvent des distributions d'allure géométrique si elles ne sont pas tronquées par la fin de l'U.C. porteuse.

2.3. Connaissances mathématiques requises

Le cadre mathématique des processus stochastiques est particulièrement bien adapté à la description de la production des bourgeons et des méristèmes comptée en nombres de composants sans avoir à en expliciter le fonctionnement interne. On dispose d'outils pour engendrer des distributions statistiques (nombre de branches, nombre de pousses, nombre d'organes) calibrées sur les observations effectuées sur le terrain.

La mise en place des nouveaux éléments en bout d'axe se fait séquentiellement dans le temps selon des rythmes plus ou moins réguliers qui engendrent d'un axe à l'autre des distributions statistiques du nombre d'éléments produits. On peut caractériser ce fonctionnement selon la théorie « du renouvellement » qui définit un temps de renouvellement comme une variable aléatoire. Le nombre d'éléments fabriqués par les différents bourgeons qui fonctionnent en parallèle suit une « loi de comptage » dont la moyenne et la variance ne dépendent que du temps de fonctionnement et de la moyenne et la variance du temps de renouvellement. Les distributions tendent rapidement vers des lois normales qui sont approximées par des lois binomiales. Le processus de renouvellement est alors modélisé d'une façon simplifiée par un « processus de Bernoulli » qui caractérise des successions de périodes d'activité et de pause selon une probabilité.

Une conséquence de ce modèle est que le temps continu du fonctionnement est discrétisé et à chaque pas de temps on peut associer à un méristème des probabilités de croissance, de mort et de mise en place d'éléments axillaires (fleurs, branches). Ce système a été établi par de Reffye (1979).

Le modèle AmapSim est ainsi logiquement relié aux théories du renouvellement, de la fiabilité et aux chaînes de Markov. L'aspect markovien des séquences ramifiées des axes végétatifs a été mis en évidence sur le caféier (de Reffye 1979) et perfectionné par la suite sur les arbres fruitiers (Guédon 2001). Les travaux récents exploitent ce modèle pour leurs donner un formalisme basé sur les grammaires dont on peut déduire les fonctions génératrices, les moments et les distributions engendrées (Cournède 2006, Kang 2008). Ces aspects mathématiques ne seront pas détaillés ici.

2.4. Le modèle AmapSim

Le modèle et son implémentation sont basés sur le travail décrit dans Barczi et al (1997). Des améliorations y ont été apportées pour mieux décrire les phénomènes de métamorphose et le processus de ramification. De plus, le simulateur a été réarchitecturé afin de séparer les fonctions de base de gestion des données du noyau de simulation lui-même. Ces améliorations permettent de proposer une interface fonctionnelle pour développer des modules additionnels (Barczi *et al.*, 2007). AmapSim est un modèle de croissance de plantes purement structurel dont les principes s'appuient sur des considérations botaniques éprouvées et dont les paramètres permettent de reconstruire des maquettes de plantes tridimensionnelles dont les caractéristiques s'approchent le plus possible des mesures effectuées sur le terrain sur de vraies plantes. Dans la mesure où les individus d'une même espèce peuvent présenter une certaine variabilité les distributions de branches, pousses, entre-nœuds, etc., seront simulés par des processus stochastiques.

Le modèle s'appuie sur les concepts suivants :

1. les éléments constituants de la plante (les organes),
2. l'indexation de ces composants en fonction de l'état de différentiation de son méristème édificateur (l'axe de référence),
3. un graphe des connections entre les composants de la plante (la topologie),
4. le rythme d'apparition de ces composants par unité de temps (le développement),
5. la position et l'apparence des composants de la plante dans l'espace 3D (la géométrie),

2.4.1. Eléments constituants de la plante.

AmapSim décrit la structure de la plante selon une décomposition hiérarchisée qui correspond aux observations. Les différents niveaux de décomposition sont :

- l'axe
- la pousse annuelle
- l'unité de croissance
- la zone
- le métamère

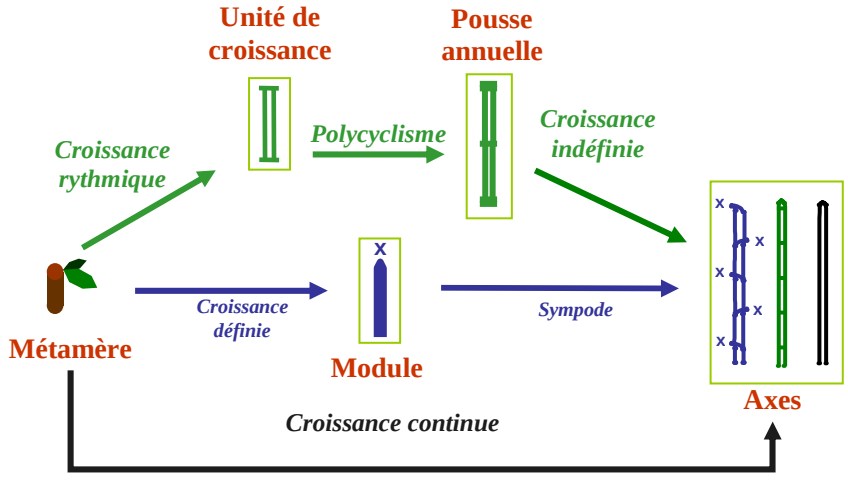


Fig 13 : Shéma de décomposition d'un axe (Barthélémy, 2007)

La partie active du modèle est le bourgeon qui correspond au méristème apical ou latéral. Il n'a pas de réalité physique mais engendre des entre-nœuds qui constituent un axe ou des bourgeons latéraux. Il est à noter que dans ce modèle toutes les productions latérales ont le même statut qu'elles soient des feuilles, des fleurs ou des axes. Leur apparition et leur modalité de développement sont régies par le même algorithme.

2.4.2. Axe de référence

L'axe de référence traduit les caractéristiques botaniques et leur évolution au cours de la vie des méristèmes au travers de l'évolution de l'âge physiologique tel qu'il a été proposé par Rouane (1977) dans le cas des cyprès. Cet axe de référence est une séquence ordonnée par valeur croissante de tous les âges physiologiques possibles pour un genre donné. D'un point de vue informatique on simule l'axe de référence et son parcours par les méristèmes de la plante par un automate cellulaire dont la variable d'état est l'âge physiologique. Afin de mimer au plus près les observations botaniques qui montrent que le vieillissement des méristèmes peut suivre des parcours différents à des vitesses différentes selon leur environnement, la loi de progression dans l'automate a été choisie comme un processus semi-markovien. Le nombre d'étapes occupées dans un même état est simulé comme une loi binomiale éventuellement négative, la probabilité de transition définit les changements d'état. Chaque méristème de la plante possède une variable qui contient la valeur de son étape courante (l'âge physiologique). Le fonctionnement normal correspond à la progression continue le long de l'axe de référence, la simulation de la floraison peut être effectuée par un saut en fin d'axe de référence (état de différentiation ultime). Toutes les valeurs des paramètres du modèle sont indexées sur l'âge physiologique ce qui traduit l'évolution de la production des méristèmes au cours de leur vie (cf figure 14).

2.4.3. Topologie

On simule le développement d'une pousse végétative (U.C.) par la succession de N tests d'accroissement avec pour chacun une probabilité 'p' d'engendrer un nouvel entre-nœud. On simule ainsi des distributions de type binomial (éventuellement négative) au sein de la pousse. Une pousse annuelle peut consister en plusieurs pousses végétatives dont le nombre est simulé par un tirage dans une loi de probabilités cumulative, on parle alors de polycyclisme, l'enchainement de la caractéristique « polycyclisme » entre les pousses annuelles est simulé par un processus markovien. L'ensemble des valeurs des paramètres de ces lois peut varier d'une pousse annuelle à l'autre. Par combinaisons de ces paramètres ainsi que par des tirages aléatoires, on engendre des axes qui comprennent des nombres variables de pousses végétatives comprenant elle-même des nombres variables d'entre-nœuds. La correspondance entre le pas de temps de calcul et le temps réel observé au cours de la croissance de la plante dépend du point de vue du modélisateur et de l'espèce de plante modélisée. Dans le cas des plantes à croissance continue et non rythmique, une unité de temps peut correspondre à un test d'élongation. A l'opposé, dans le cas du *Zelkova serrata* (deReffye 1991), une unité de temps permet de mettre en place une pousse annuelle (U.T.) constituée de plusieurs pousses végétatives comprenant chacune des dizaines d'entre-nœuds.

Chaque pousse végétative (Unité de Croissance) peut consister en une partie préformée et éventuellement d'une partie néoformée. Chacune de ces parties est composée

d'un nombre d'entre-nœuds issus de la simulation d'une loi binomiale décalée éventuellement négative. Le choix de l'apparition de la néoformation est décidé selon une simple probabilité.

2.4.4. Simulation de l'organogenèse par un automate gauche droite

D'un point de vue mathématique, on peut associer un *automate stochastique « gauche-droite »* au fonctionnement d'un méristème. Cet automate peut être dans différents états qui correspondent aux âges physiologiques. Le passage d'un état à l'autre se fait par cycles successifs selon les lois de transition d'un état à l'autre. A chaque cycle correspond un état, même si ce dernier peut être répliqué plusieurs fois. Le nombre de cycles de fonctionnement de l'automate est limité, et l'état qui correspond à l'âge physiologique le plus vieux marque dans tous les cas l'arrêt définitif de la croissance de l'axe. L'automate enchaîne des U.C. qui portent elles-mêmes des bourgeons axillaires qui renvoient à des âges physiologiques égaux ou plus vieux que ceux des U.C. porteuses selon la règle botanique qui interdit le rajeunissement sauf dans de rares cas.

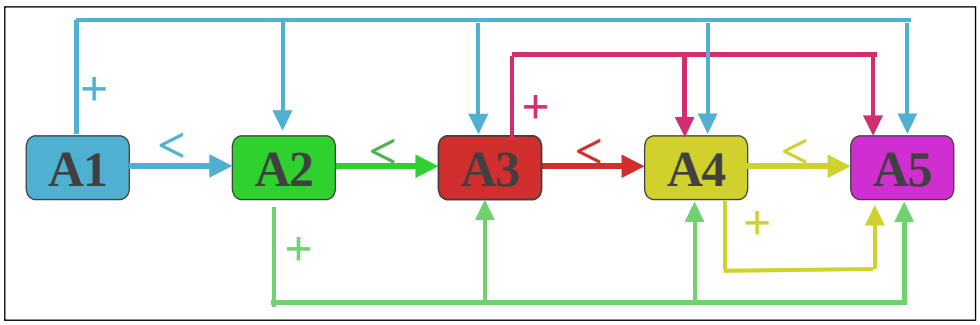
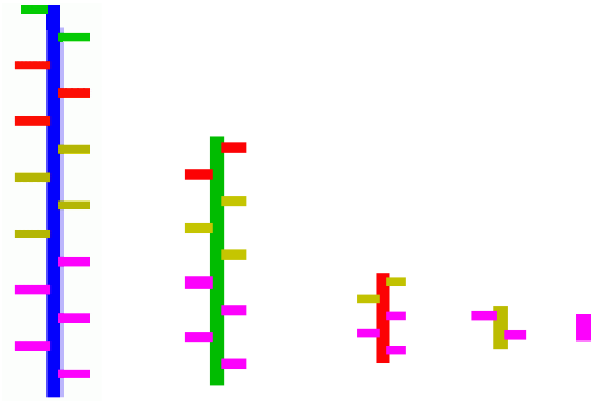


Fig 14. Exemple d'automate gauche droite à 5 états pilotant la croissance de la structure de la figure 3 : Symbole de prolongation **:<** ; de ramification : **+**



Etats : 1 2 3 4 5

Figure 15 : Description des U.C. des cinq âges physiologiques et états de l'automate correspondant à la figure 1.

Le nombre d'entre-nœuds d'une U.C. suit une distribution statistique caractéristique de l'état courant de l'automate. Les séries axillaires sont disposées le long de l'U.C. selon des

chaînes de Markov caractéristiques également de ce fonctionnement. Entre-nœuds et bourgeons axillaires ne sont donc pas créés en même temps dans ce modèle.

La figure 14 montre l'organisation d'un tel automate qui met en place à chaque cycle une nouvelle U.C. et des U.C. axillaires associées à partir de l'U.C. précédente. Le méristème principal ne subit qu'une étape de transition à la fois alors que les méristèmes axillaires peuvent effectuer des sauts d'un nombre quelconque d'étapes. Pour chaque état de l'automate, la structure topologique des U.C. doit être définie en nombre d'entre-nœuds et en production axillaire (figure 15).

Le fonctionnement de l'automate assure le développement de la structure cycle par cycle selon les lois de transition programmées pour les états. La trace du fonctionnement peut être représentée par un « ***axe de référence*** » qui schématise l'organisation botanique de la structure (figure 16)

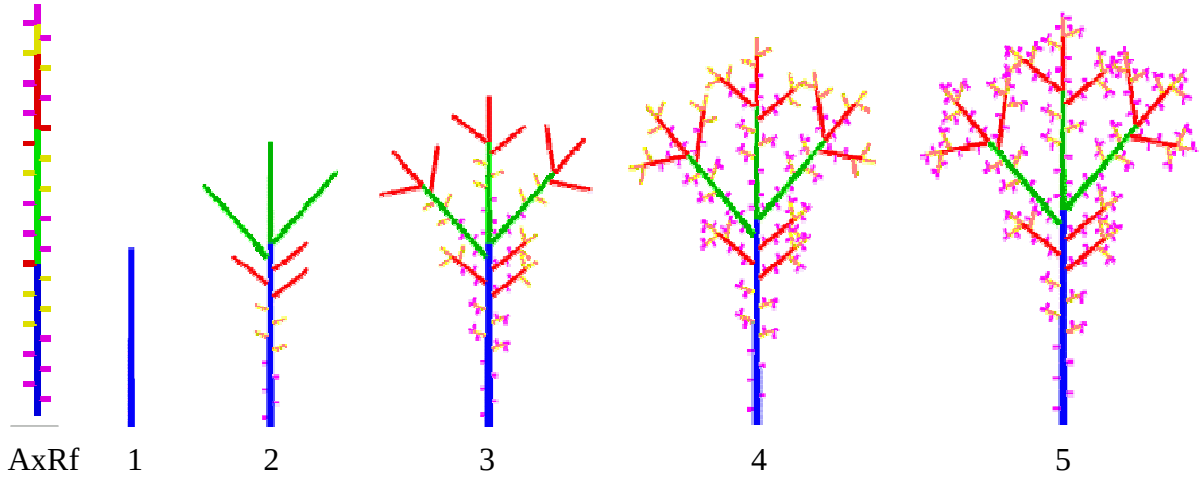


Figure 16. Croissance d'une structure en 5 cycles construite par l'automate de la figure 14 avec les U.C. de la figure 15 et Axe de Référence (AxRf) correspondant.

La réplication d'un même état provoque la construction d'un axe à structure rythmique qui permet d'édifier par exemple un tronc ou des branches dominantes (figure 17).

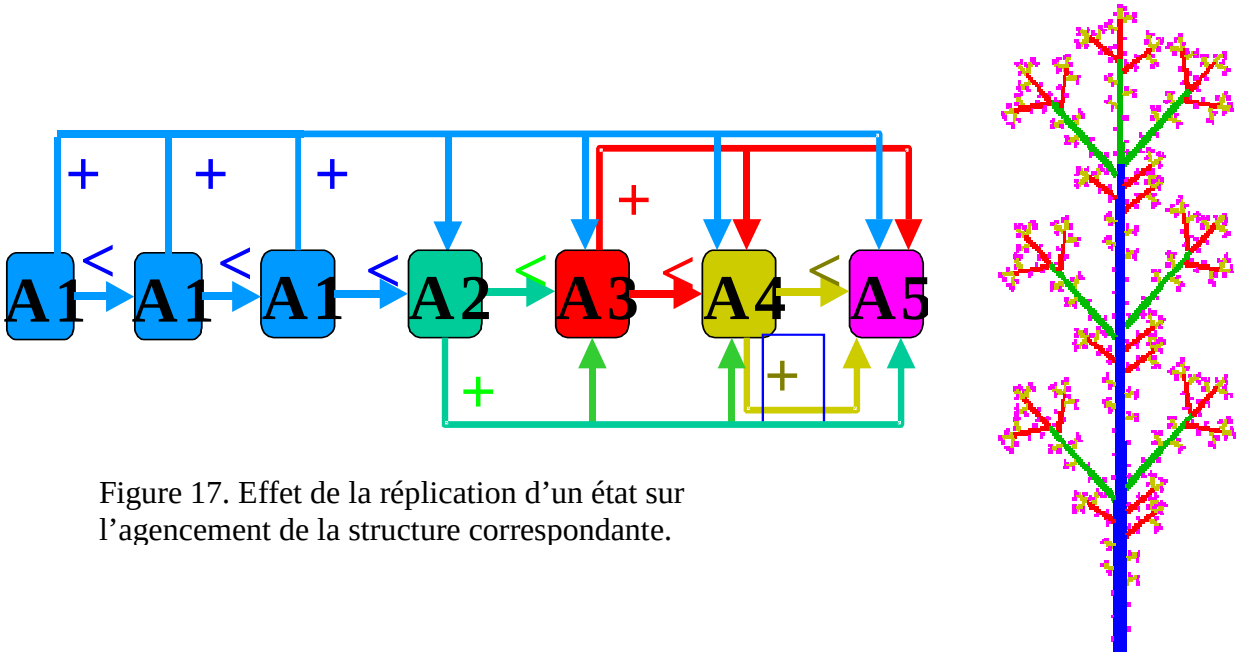
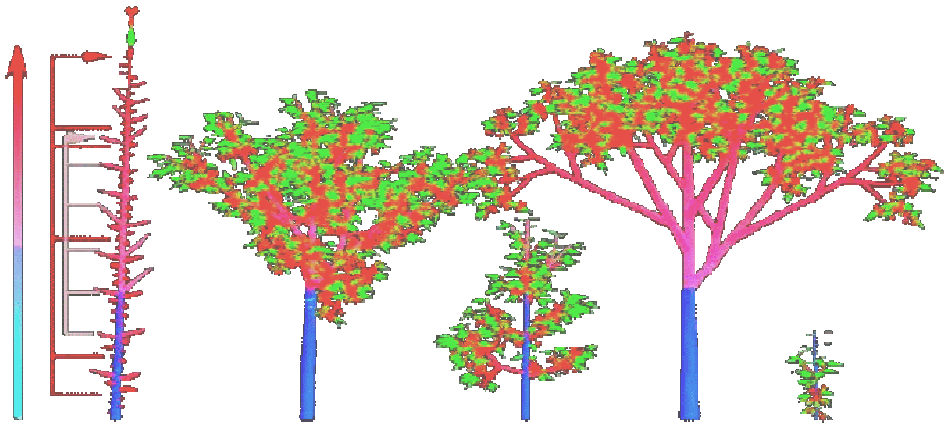


Figure 17. Effet de la réplication d'un état sur l'agencement de la structure correspondante.

L’ « *Axe de Référence* » permet de suivre pas à pas le changement de fonctionnement des bourgeons d’un cycle à l’autre (figure 18).



Axe de Référence (3) (2) (4) (1)

Figure 18. Construction d’un arbre selon la méthode de l’axe de référence : (1) stade pépinière; (2) jeune arbre avec tronc dominant ; (3) apparition des réitérations ; (4) Arbre adulte. La couleur qui va du bleu au rouge symbolise l’évolution de l’âge physiologique des bourgeons.

Le port des arbres change suivant leur niveau de développement. Dans les stades juvéniles les modèles architecturaux sont bien visibles car les catégories d’axes sont bien organisées en fonction de l’ordre de ramification. Les axes porteurs dominent les axes portés. Au fur et à mesure du vieillissement les bourgeons édificateurs « rattrapent » l’âge physiologiques des axillaires qu’ils mettent en place et la dominance apicale disparaît. La périphérie d’un arbre âgé semble un empilement désorganisé d’axes équivalents et vieux physiologiquement. Ces axes ont des U.C. avec des entre-nœuds courts, florifères et peu nombreux. Le nombre d’étapes d’un axe de référence est théoriquement illimité. Certains arbres (eucalyptus) nécessitent plusieurs centaines d’âges physiologiques pour être simulés.

La théorie du renouvellement permet bien de modéliser le phénomène d'apparition de nouveaux entre-nœuds le long d'un axe. Le nombre d'événements qui se produisent pendant une période fixée suit une loi de comptage. Celle-ci dépend complètement de la distribution du temps de renouvellement. On démontre que lorsque la loi de comptage est calée sur un temps suffisamment long elle prend une allure de loi normale (théorème central limite) (Feller, 1968). Soit m la moyenne du temps de renouvellement et s l'écart type. Soit T la période d'observation. M la moyenne de la loi de comptage, V sa variance. On démontre les relations suivantes :

$$M = \frac{T}{m}$$

$$V = T\frac{s^2}{m^3}$$

Sous cette forme on s'aperçoit que si la moyenne du temps de renouvellement augmente comme l'écart type, on aura une simple dilatation de l'échelle des temps pour l'observation de la loi de comptage. On obtiendra exactement la même loi de comptage au

bout d'un temps d'observation augmenté dans une même proportion que celle de la moyenne du temps de renouvellement. Cela montre l'invariance d'une architecture finale par rapport à la vitesse de construction. Une autre caractéristique est que la variance de la loi de comptage reste proportionnelle à la moyenne, quelle que soit la variation de la moyenne du temps de renouvellement si celui-ci reste proportionnel à son écart type. Le parti pris dans cette version de la simulation est d'abandonner le temps réel, pour se caler sur un temps fictif, en choisissant un renouvellement sommaire lorsque cela est possible. On peut le faire lorsque le nombre d'entre-nœuds fabriqués par des méristèmes semblables au cours d'une même période suit une distribution proche d'une loi binomiale, ce qui est généralement vérifié. En effet on peut écrire en égalisant les moyennes et les variances de la loi binomiale avec celle de la loi de comptage :

$$N \cdot b = \frac{T}{m} \text{ et } N \cdot b \cdot (1-b) = T \cdot \frac{s^2}{m^3}$$

Cela donne :

$$b = 1 - \frac{s^2}{m^2}$$

Ainsi tout se passe comme si les méristèmes avaient subi N tests d'accroissements avec une probabilité b de faire un entre-nœud à chaque test. N joue ici le rôle du temps. On l'appelle âge du méristème. Pour chaque N on peut associer un stade de développement et donc une architecture, mais pas une durée de croissance précise. Il est évident qu'il faut recourir à la théorie plus générale du renouvellement si l'on veut se préoccuper des suivis de croissance dynamiques de mise en place d'une structure. Toutefois lorsqu'on peut lier le développement de la plante à une somme de températures (par exemple), il est alors possible d'indexer l'âge N du méristème sur ce paramètre. On obtiendra alors un modèle hybride de croissance stochastique proche du renouvellement. L'exemple suivant nous montre la bonne convergence de la loi de comptage vers la loi binomiale pour un processus de renouvellement donné. Choisissons deux processus de renouvellement qui ont même moyenne (m = 6) et même variance (s = 2) : le premier a un temps de renouvellement distribué suivant la loi binomiale de paramètres N = 9 et b = 2/3.le deuxième a un temps de renouvellement distribué selon une loi géométrique de paramètres q = 0.5 et de décalage d = 5.

On vérifie pour la loi binomiale que $m = N \cdot b = 6$ et $s^2 = N \cdot b \cdot (1-b) = 2$, et pour la

Loi géométrique décalée que $m = d + \frac{1-q}{q} = 6$ et $\frac{1-q}{q^2} = 2$

Ces deux lois convergent, pour le temps de comptage T, vers la loi binomiale de paramètres :

$$b = 1 - \frac{s^2}{m^2} \text{ et } N = \frac{T}{m \cdot (1 - \frac{s^2}{m^2})} \text{ (figure 19)}$$

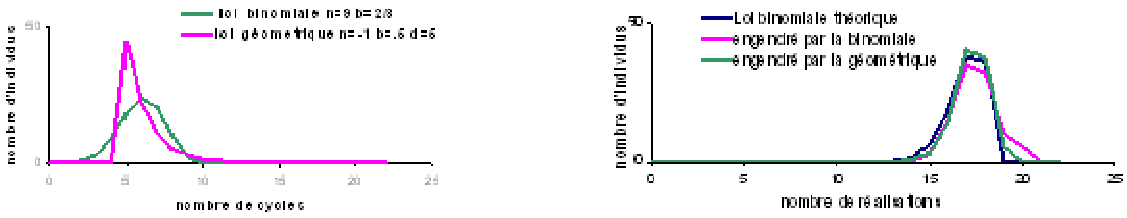


Figure 19. Similarité des lois de comptage (à droite) engendrées par deux lois de renouvellement de mêmes moments (6,2) (à gauche).

Le processus d'arrêt et le processus de croissance se combinent en une loi composée qui détermine la distribution du nombre d'entre-nœuds par pousse végétative. La grande majorité des distributions rencontrées s'ajustent correctement à des lois binomiales positives ou négatives décalées ou non (en passant par la loi de Poisson) (deReffye *et al.*, 1991*).* D'un point de vue mathématique on peut montrer qu'il y a continuité entre ces lois. En effet soit $G(Z) = (1 - c + c.Z)^N$ la fonction génératrice de la loi d'arrêt. Selon que N est positif ou négatif on aura une binomiale de même caractéristique. Si c tend vers 0 et N tend vers l'infini, avec N*c constant, on obtiendra une loi de Poisson. Si l'on combine cette loi d'arrêt et un processus de croissance binomiale avec une probabilité b de faire un nouvel entre-nœud à chaque test d'accroissement. On peut directement écrire la fonction génératrice de la distribution résultante selon les règles des lois composées :

$$G(Z) = (1 - c + c(1 - b + bZ))^N$$

soit après simplification :

$$G(Z) = (1 - bc + bcZ)^N$$

On remarque que la composition de la croissance et de l'arrêt ne change pas la nature de la loi originale de l'arrêt. Celle-ci se tasse simplement. Ce résultat est d'un grand intérêt car il permet de savoir grossièrement comment fonctionne la pousse végétative sans avoir observé sa croissance. En effet supposons que nous trouvions une loi binomiale dans la distribution du nombre d'entre-nœuds d'une U.C. Cela veut dire que la distribution du moment de l'arrêt est groupée en un pic binomial (ex. néoformation du merisier). Ces méristèmes ont donc une période de croissance bien déterminée. .Supposons que le nombre d'entre-nœuds se distribue selon une loi géométrique. Cela veut dire que la probabilité d'arrêt du méristème est constante à chaque test (ex. néoformation de l'orme du Japon). Ces méristèmes ont donc une période de croissance indéterminée. Notons que seul le produit bc est estimable après coup. Pour connaître b et c séparément. Il faut avoir fait des mesures sur la population de rameaux en cours de croissance.

La figure 20 montre la distribution du nombre d'entre-nœuds fabriqués par la composition d'une loi de croissance et d'une loi d'arrêt. La loi de croissance est une binomiale (N,b) et la loi de mortalité une loi géométrique de paramètre c. La partie gauche de la figure concerne les axes dont les bourgeons sont arrêtés et la partie droite l'effectif des axes regroupés en un pic qui demeurent en croissance à l'étape N de l'observation. Lorsque le temps croît indéfiniment, tous les bourgeons finissent par s'arrêter et le pic de croissance disparaît. La distribution du nombre d'entre-nœuds fabriqués tend alors vers la loi géométrique de paramètre b * c.

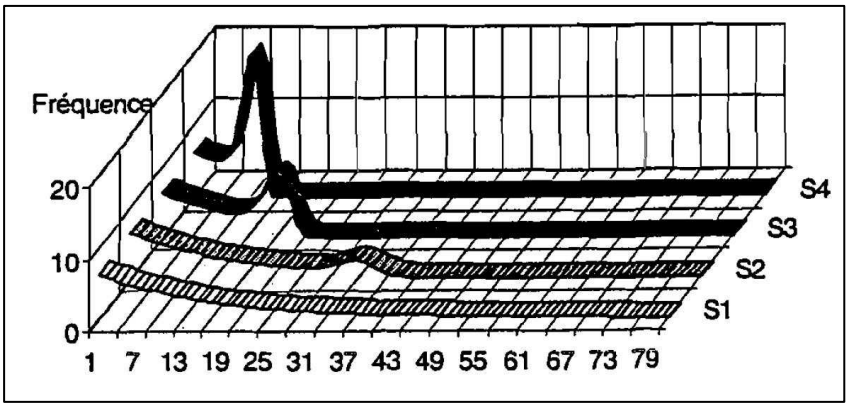


Figure 20. Composition de la loi de croissance binomiale avec la loi d'arrêt géométrique pour différentes périodes de croissance

Chez certaines plantes tropicales à croissance rythmique (Eperua Falcata) ou bien chez des plantes tempérées (Pinus Alepensis), on observe que les méristèmes d'une même plante peuvent émettre des pousses à des fréquences différentes. On définit ainsi un rapport de rythme *r* qui correspond à la vitesse de mise en place des pousses proportionnellement à une base virtuelle choisie par le modélisateur et qui correspond à l'année dans le cas des arbres poussant en zone tempérée. Une pousse annuelle consommera un pas de temps 1/r.

La mortalité ou la mutation des méristèmes doit être également prise en compte pour décrire l'établissement d'une architecture végétale. Celle-ci peut avoir plusieurs origines. Si elle est accidentelle elle sera contrôlée par une probabilité de survie à chaque cycle de croissance. Dans ce cas les nombres d'entre-nœuds des pousses engendrées se distribueront selon des lois géométriques si la probabilité de survie est constante. Si elle est le résultat d'une différentiation programmée alors les nombres d'entre-nœuds des pousses engendrées se distribueront selon des lois normales. Le méristème peut dans ce cas avorter, se métamorphoser en parenchyme, vrille, fleurs etc…

Les méristèmes édificateurs des axes produisent également à chaque nœud des méristèmes axillaires de diverses natures. Ainsi, classiquement, on trouvera un méristème à l'aisselle d'une feuille pouvant donner un rameau axillaire ou une inflorescence. Le long d'un axe végétatif, on peut trouver, successivement, des séries de nœuds ramifiés et des séries de nœuds non ramifiés (ex. abricotier), des séries de rameaux alternant avec des séries d'inflorescences. Ces séries peuvent être décrites par des chaînes de Markov ou Semi- Markov (ex. bégonia) (Guédon *et al*. 2001).

2.4.5. Géométrie de la maquette

Une fois la structure topologique de la plante définie, il faut encore les disposer dans l'espace afin d'obtenir une maquette tridimensionnelle aussi fidèle que possible. Nous rappellerons brièvement les procédures à suivre qui ont été exposées par Jaeger (1987). A chaque étape de l'axe de référence, c'est-à-dire à un âge physiologique donné, est associé un organe dessiné grâce à un modeleur 3D. Ainsi l'entre-nœud sera un cylindre à facettes, la feuille une surface divisée en polygones... Ces organes seront placés par le simulateur d'AmapSim suivant des règles géométriques précises. L'axe se construit par l'empilement des entre-nœuds successifs auxquels on donne des règles géométriques de rotation (flexion ou redressement). Des paramètres permettent de contrôler cette flexion. Le redressement caractéristique des fins d'axes végétatifs orthotropes est aussi paramétré. La direction d'un axe végétatif dépend du positionnement de son premier entre-nœud. Celui-ci est placé par la phyllotaxie et l'angle de branchement. Chaque organe possède une taille à sa création. Il peut se dilater pendant un certain nombre de tests de croissance. C'est le cas notamment du diamètre des entre-nœuds. Pour l'instant leur dilatation est une fonction linéaire empirique du temps et de la position dans l'unité de croissance choisie.

Tous les paramètres géométriques peuvent varier le long de l'axe de référence. Cette procédure permet de contrôler la géométrie de la plante en fonction de l'âge physiologique des organes fabriqués. Pour l'instant AmapSim ne simule pas les interactions de type géométrie/topologie (comme l'épitonie) bien qu'elles soient actuellement à l'étude.

3. La librairie Vitis

Lorsqu'un modèle a été défini, on peut envisager son implémentation dans un simulateur. En l'occurrence, plusieurs versions de simulateur d'AmapSim ont été développées, chacune d'entre elles correspondait à un niveau de technicité et à des attentes différents. Dans un premier temps et dans l'urgence de vérifier le bien-fondé des hypothèses du modèle, un premier prototype a vu le jour qui reproduisait stricto sensu les principes d'édification de plante selon AmapSim (Barczi, 1997). Malheureusement, si les sorties étaient acceptables, un calcul préfixé de la structure interdisait toute autre application que celles liées à l'image de synthèse. Le calcul préfixé (chaque branche est simulée intégralement avant de passer à la suivante) avait été choisi pour obtenir un temps de calcul et une consommation mémoire compatibles avec les performances du matériel de cette lointaine époque. Une seconde version a été développée qui assurait la compatibilité ascendante des paramètres avec la version précédente, fournissait des sorties de qualité équivalente mais dont l'algorithme reproduisait une croissance globale homogène (quasi parallèle) selon la méthode proposée dans la thèse de Frédéric Blaise (1991). Cette nouvelle version ouvrait l'accès à l'application liée aux problématiques agronomiques, malheureusement le temps de calcul et la quantité de mémoire requis restreignaient les applications à de petites plantes. Ces restrictions nous ont amenés à concevoir un principe d'abstraction des données de représentation de la plante par simplification topologique. A ce niveau de réalisation, le simulateur d'AmapSim a permis beaucoup d'applications dans le domaine de l'agronomie, malheureusement (rien n'est parfait) chacune d'entre elles nécessitait de petites améliorations de détail dans le modèle pour s'adapter correctement à leurs contraintes spécifiques, engendrant autant de prototypes dédiés et par là même une lourdeur dans la maintenance de leur multiples codes. Le développeur étant feignant par nature, il a alors fait part de ses doléances au concepteur qui a revu l'organisation du simulateur de manière à rendre son utilisation plus souple et surtout plus générique. La tâche consistait essentiellement en deux points : d'une part séparer clairement la partie concernant la simulation du modèle lui-même des services dont il peut avoir besoin et d'autre part en offrant la possibilité d'adapter le comportement du modèle sur des points de détails sans avoir à réécrire son simulateur à chaque fois.

Ce bref historique des différentes versions successives du simulateur d'AmapSim expose la manière peu orthodoxe (au sens de la conception) qui nous a menés à l'architecture logicielle que nous présentons dans le cadre de cette thèse. Il existe aujourd'hui des outils de conception qui permettent (en se posant malgré tout de bonnes questions quant aux problématiques auxquelles on souhaite répondre (Hill, 1996)) de concevoir rapidement des applications robustes et flexibles. On peut citer UML (Booch, 2000) qui permet de décrire de manière synthétique et incrémentale une application informatique, ou bien les Design Pattern (Gamma et al, 1999) qui proposent des solutions éprouvées à des problèmes classiques de la programmation ou bien Devs (Zeigler, 1996) qui propose un cadre conceptuel solide pour la conception et la réalisation de simulateurs de modèles discrets.

Si l'on considère les méthodes actuelles de construction d'environnements de développement, on voit apparaître une contrainte qui permet un progrès majeur bien connu dans le génie logiciel : la nécessité de **capitaliser** les données et les traitements que la plupart des modèles de simulation de plante sont amenés à utiliser.

Cette considération nous a amenés à choisir de définir une librairie d'outils constituant un noyau à l'usage des développeurs de simulateurs de plantes. Le modèle de simulation proprement dit est intégré dans cet environnement et utilise les fonctionnalités offertes par ce noyau.

Tout d'abord l'objectif étant de simuler la croissance de plantes, il semble nécessaire de se pencher sur la question de la représentation de la plante en mémoire et des outils pour manipuler cette donnée, l'objectif étant de proposer une solution qui soit extensible autant en termes de finesse de description que de contenu de représentation.

Ensuite, dans la mesure où nous nous intéressons à un automate à états finis dont une grande quantité d'instances gèrent le fonctionnement de plusieurs bourgeons, la question de la gestion du déclenchement de traitements ordonnés dans le temps doit être traitée. L'approche évènements discrets semble adaptée au traitement du développement de l'architecture des plantes.

Par ailleurs il semble nécessaire d'offrir la possibilité de construire un simulateur à partir de plusieurs briques logicielles (dont une partie constitue le modèle de base) qu'il faut alors synchroniser et faire éventuellement communiquer.

Enfin les mécanismes de sauvegarde des résultats de simulation se doivent d'être indépendants du simulateur qui les engendre mais plutôt d'être liés aux données présentes en mémoire. On mutualise ainsi la tâche fastidieuse d'écriture de fonctions de sortie et on propose naturellement une solution de comparaison des résultats de simulations qui seraient issues de différents simulateurs hébergés dans le même environnement.

Dans cette optique, la librairie fournit des blocs fonctionnels remplissant différentes tâches de base :

- Représentation de Plante, outils de mise en place et de parcours de cette représentation, simplification topologique/géométrique, enrichissement dynamique de la description de la représentation de la plante.
- Gestion d'événements discrets, échéancier, abonnement, priorité, déclenchement de traitements.
- Gestion de paramètres de modèle de simulation.
- Communication et déclenchement fonctionnel par signaux, interface fonctionnelle par défaut (au moment de la mise en place des éléments de plante, en début et fin de la construction géométrique, au moment de l'accès aux paramètres du modèle)
- Gestion des sorties sous différents formats.
- Mécanisme de persistance mémoire pour permettre de stopper la simulation et de la reprendre ultérieurement.

L'implémentation de la librairie a été réalisée en C++. La description détaillée des objets qui la constituent peut être consultée à l'adresse *http://umramap.cirad.fr/amap2/logiciels_amap/index.php?page=documentation-4/docVitis.doc*. Le choix du langage a été décidé plus pour des raisons historiques que technique sachant que C++ semble fournir l'ensemble des outils requis. N'importe quel autre langage objet aurait pu faire l'affaire. Il ressort tout de même la contrainte de l'apprentissage de ce langage pour quiconque souhaiterait utiliser cette librairie.

On peut considérer les outils qui vont être décrits dans les paragraphes suivants comme une Application Programing Interface (Hines 1996). Si un travail de conception d'une interface graphique est effectué sur la base de cette API, on pourra atteindre peut-être le Graal de la plateforme.

3.1 Description et gestion de la plante

La librairie fournit des classes de base permettant de construire et parcourir la plante en cours de simulation.

Cette description est proposée selon le point de vue topologique ou bien géométrique.

D'un point de vue topologique la plante est vue comme une collection de branches qui s'organisent selon une hiérarchie de porteurs et de portés. Chaque branche peut se décomposer à différents niveaux de précision selon les besoins du modèle à implémenter. Selon le type de plante considérée, selon qu'elle présente une croissance continue ou pas, qu'elle est capable d'exprimer du polycyclisme ou de la néoformation, le modèle peut avoir besoin de niveaux de finesse de représentation variable. Dans le cas du modèle AmapSim, toute branche se décompose en pousses annuelles qui se décomposent en cycles qui se décomposent en zones qui se décomposent en entre-nœuds. Dans le cas d'un modèle spécifique pour simuler des systèmes racinaires ou bien d'une plante qui ne présente aucune rythmicité on n'aurait besoin que d'axes qui se décomposent en entre-nœuds.

C'est donc le modèle qui définit les niveaux de décomposition qu'il va utiliser et qui spécialise éventuellement chacun de ces niveaux en fonction de ses besoins. Le niveau le plus haut (correspondant aux axes) est spécialisé pour permettre de gérer l'organisation des branches entre elles.

La relation porteur/porté entre les différentes branches de la plante est stockée dans un tableau (que l'on appelle « hiérarchie ») où chaque enregistrement possède une référence à la branche topologique qui lui correspond, une référence à l'enregistrement qui décrit sa branche porteuse, la position d'insertion de cette branche (au niveau de décomposition le plus bas), la liste des enregistrements qui décrivent les branches portées et leur position d'insertion au niveau de décomposition le plus bas. Il est important de noter que le nombre d'enregistrements dans la hiérarchie correspond exactement au nombre de branches dans la plante alors que pour des considérations de simplification topologique, il peut y avoir en mémoire un nombre plus restreint de branches topologiques (voir le paragraphe 4.2 pour une illustration de cette fonctionnalité dans AmapSim).

Chaque fois que le modèle demande la mise en place d'une nouvelle branche, il adresse une requête au *TopoManager* le noyau crée un enregistrement dans la hiérarchie. Pour cela il faut évidemment fournir la référence du porteur ainsi que la position sur ce porteur. Chaque fois que le modèle demande la mise en place d'un composant à un niveau de décomposition inférieur, celui-ci est inséré à la position courante du niveau immédiatement supérieur. Si celui-ci n'existe pas, il est créé au préalable.

La librairie fournit un jeu de fonctions qui permettent de sélectionner la branche courante par parcours de la hiérarchie, de sélectionner l'élément courant dans chaque branche et de parcourir une branche à tout niveau de décomposition.

Considérant qu'une plante en général et les arbres en particulier présentent une grande quantité d'auto similitude et afin de permettre d'alléger la représentation en mémoire de la topologie d'une plante, le noyau fournit une classe qui gère l'association de plusieurs enregistrements de la hiérarchie à la même branche topologique. Un modèle peut grâce à cet outil constituer des classes de branches qui sont identifiées par une clé alphanumérique. Chaque classe peut contenir un nombre variable de représentants au choix du modèle. Lorsque le modèle de croissance décide de créer une nouvelle branche il peut interroger le simplificateur de topologie pour obtenir une instance de branche qui correspond au critère qu'il a calculé pour cette branche. Si le simplificateur lui rend une instance de branche existante alors il crée simplement un enregistrement dans la table de hiérarchie et lui associe cette branche. Si le simplificateur ne rend pas d'instance de branche alors le modèle crée et enregistre la branche dans la liste en lui associant sa clé alphanumérique puis crée un enregistrement dans la table de hiérarchie et lui associe cette branche.

D'un point de vue géométrique la plante est vue comme une collection de branches qui se décomposent en une séquence d'éléments géométriques de base. Le lien avec la topologie est réalisé pour chaque branche par une référence à l'enregistrement correspondant dans la hiérarchie. Chaque élément géométrique de base est lié au composant topologique de niveau de décomposition le plus bas qui lui correspond. Les éléments géométriques de base possèdent en attribut une variable qui contient la longueur du constituant qu'il représente dans la plante et une matrice de transformation qui permet de positionner cet élément dans l'espace.

La librairie fournit un mécanisme de simplification de la géométrie indexé sur un paramètre numérique. Si ce paramètre a une valeur comprise entre 1 et 3 alors les éléments successifs d'une même branche sont susceptibles d'être concaténés s'ils ont des directions principales proches. Si la valeur est supérieure à 4 alors le plan vertical est divisé en autant de secteurs angulaires. Les branches qui font partie de la même classe topologique et qui ont leur angle d'insertion dans le même secteur angulaire d'élévation partageront la même représentation géométrique.

Tant l'algorithme de simplification topologique que de simplification géométrique ont pour but d'alléger le temps de calcul du modèle qui sera hébergé que de diminuer la quantité de mémoire utilisée pour représenter la plante en cours de simulation. Il va sans dire que ces gains s'accompagnent d'un déficit en précision ou variabilité de la représentation. C'est à l'utilisateur de choisir le meilleur compromis en fonction de l'application qu'il envisage et de la puissance de calcul dont il dispose (voir le paragraphe 4.2 pour une illustration de cette fonctionnalité dans AmapSim).

Dans sa version de base, la librairie fournit une description de la plante qui est suffisante pour gérer sa topologie et sa géométrie. Il est cependant prévisible que les modèles de croissance qui seront hébergés souhaiteront attacher à cette description des informations qui leurs sont propres. Par exemple, un modèle qui met en jeu des mécanismes physiologiques pourra souhaiter mémoriser la surface des feuilles, la résistance hydraulique des composants de la plante ou bien la biomasse qu'ils ont accumulée. Un modèle de type dendrométrique voudra peut-être calculer et attacher un volume de houppier au tronc ou bien une masse feuillée portée par chaque rameau. Un modèle de type mécanique souhaitera attacher à chacun des composants de la plante les contraintes qu'il supporte. Pour chacun de ces exemples, une solution d'implémentation consisterait à dériver les classes fournies par le noyau et à ajouter

les attributs souhaités ainsi que les traitements qui leurs sont associés. On fabrique alors des applications dédiées à chaque type de modèle choisi. Mais on peut alors se demander de quelle manière il sera possible de créer une application qui mélange par exemple les composantes physiologiques et mécanique. Encore une fois on peut imaginer de bâtir ce simulateur en dérivant les classes de base de la librairie en leur rajoutant les composants et les traitements associés mais il parait évident que toutes les combinaisons des différentes applications mènent à une multitude de sources qui vont être difficilement maintenable. Nous verrons dans la suite qu'on peut envisager une solution alternative en bâtissant un modèle à partir de l'agrégation dynamique de plusieurs blocs fonctionnels qui remplissent chacun une tâche précise. Dans ce cas il est souhaitable que l'ensemble de ces blocs partage une seule représentation de la plante en cours de simulation. Ceci permet d'une part d'alléger la charge d'utilisation de la mémoire du simulateur mais cela permet également d'éventuellement partager des informations entre modules. Pour ce type d'implémentation la librairie propose un attribut particulier à tous les objets qu'elle fournit. Cet objet se nomme « AdditionalData ». Chaque modèle hébergé par la librairie a accès à cet objet en lecture, écriture et destruction et peut, grâce à lui, attacher ses propres données à la description de la plante en cours de simulation. Chaque objet de type AdditionalData possède un champ de données et une clé alphanumérique qui permet de retrouver les informations qu'il détient. Grâce à ce mécanisme, différentes parties fonctionnelles du simulateur peuvent partager des données.

Les principales classes de bases sont :

DecompAxeLevel : objet décrivant une partie de branche selon une règle de décomposition choisie par le modèle de croissance.
Branc : objet contenant la description du résultat de la simulation d'une branche.
Hierarc : objet gérant la position topologique d'une instance de branche dans la plante.
TopoManager : objet permettant de gérer la hiérarchie des branches qui constituent la plante.
SimplifTopo : objet permettant d'associer à un élément de la hiérarchie une instance de branche selon un critère de discrimination.
GeomBranc : objet contenant la description géométrique d'une instance de branche.
GeomElem : objet contenant la description géométrique d'une partie homogène de branche.
GeomManager : objet permettant de gérer la mise en place des composantes géométriques de la plante et leur calcul.
SimplifGeom : objet permettant de gérer la classification de composants géométriques en fonction de leurs caractéristiques géométriques.
PlantBuilder : objet permettant la création d'instances de parties de la plante.
AdditionalData : objet attribut de tout objet de Vitis. Les modèles accueillis peuvent associer leurs propres données à tous niveaux.
Plant : C'est l'objet principal de la librairie. Il détient une instance de *PlantBuilder*, de *TopoManager*, de *GeomManager* et de l'échéancier auquel elle réfère (voir plus bas).

3.2 Gestion du temps et des événements

D'une manière générale un simulateur peut être construit à partir de plusieurs blocs fonctionnels dont les traitements doivent être synchronisés. On peut également penser à une simulation de type multi-agents où bien tout simplement de type multi-tâche qui nécessitent également de synchroniser leurs déclenchements. Dans le cas qui nous intéresse, bien que ce

ne soit pas une obligation et pour répondre à des contraintes de simulation particulières, la croissance de la plante peut être gérée comme une séquence d'événements discrets à exécuter selon un ordre dans le temps et de niveau de priorité.

Pour ces différentes raisons, la librairie fournit un échéancier qui permet la création, l'enregistrement, l'ordonnancement et le déclenchement des traitements attachés à des actions. Chaque action possède donc trois attributs :

- Le processus qui va être déclenché par cette action ainsi que les données qui lui seront fournies.
- la date de l'action.
- la priorité de l'action.

L'échéancier fournit à tous les objets du simulateur les méthodes nécessaires pour enregistrer une nouvelle action, parcourir et modifier l'ensemble des actions couramment enregistrées.

L'échéancier (scheduler) est principalement constitué d'une boucle qui parcourt une pile d'actions ordonnées en fonction du temps puis de la priorité. L'échéancier traite les actions enregistrées tant que la pile n'est pas vide. Quand une action a été traitée, elle est enlevée de la pile. Chaque action possède une nature qui peut prendre trois valeurs : traitement, initialisation ou bien arrêt. Si l'action a une nature « traitement » alors l'échéancier appelle la fonction « process_event » du processus de cette action et fournit à cette fonction les données qui ont été associées à l'action au moment de son enregistrement. Si l'action à une nature « initialisation » alors l'échéancier appelle la fonction « init » du processus de cette action. Si l'action a une nature « arrêt » alors aucun traitement n'est lancé, l'action est simplement ôtée de la pile des actions.

Les principales classes de base sont :

VProcess : objet susceptible de déclencher un traitement par événement. Il doit fournir la méthode *process_event*(Data*) et la méthode *init*(Data*).
Action : objet contenant la description d'un événement, il détient une référence du *VProcess* correspondant; la date et la priorité de l'événement ; les données liées a cet événement.
Scheduler : objet permettant la gestion de la liste des *VProcess*, des événements associes et de leur déclenchement.

3.3 Gestion des sorties de simulation

Une sortie de résultats peut être nécessaire à des instants particuliers de la simulation. La librairie fournit une classe mère qui propose des méthodes de séquencement des sorties, de préparation des données à produire et d'écriture dans un fichier. Il est important de noter que cette manière de procéder attache la fabrication de données de sorties à la représentation de la plante et non au modèle de simulation. Cela signifie concrètement qu'un format de sortie est utilisable par n'importe quel simulateur pour peu que celui-ci s'appuie sur la représentation de plante fournie par la librairie. On offre ainsi une méthode simple pour comparer naturellement les résultats de simulations issues de différents modèles pour peu que leur simulateur utilise les outils de la librairie.

Par défaut, la librairie fournit des sorties aux formats ligne élastique (un format propriétaire Amap), MTG (Godin 1998) et GTDS (voir en annexe).

Les principales classes de base sont :

OutPutManager : objet qui permet de charger un objet de sortie à partir d'une librairie partagée, de l'initialiser et de l'enregistrer dans le scheduler.
Output : objet qui fournit la méthode *process_event* de traitement de la sortie et la méthode *computeDataToOutput* de préparation des données à sortir.
OutputFormat : objet qui fournit les méthodes d'écritures dans des fichiers à des formats spécifiques. Pour l'instant la librairie fournit le format classique ligne élastique, le format MTG et le nouveau format GTDS.

3.4 Simulation au cours du temps, persistance mémoire

Il peut être utile de pouvoir stopper le traitement d'une simulation et de sauvegarder les données nécessaires à son redémarrage ultérieur. Cette fonctionnalité permet de tester plusieurs scénarios contextuels (variation environnementale ou bien intervention extérieure sur la plante) sans avoir à redémarrer la simulation à partir de son état initial ou bien de reprendre une simulation à un âge supérieur à partir d'un état sauvegardé. On peut gagner ainsi du temps de simulation et libérer de la mémoire pour d'autres traitements. L'objet *Plant* de la librairie possède la capacité de se sérialiser, c'est-à-dire qu'il propose une fonction qui permet de créer ou bien de lire dans un fichier une image de la mémoire du simulateur de telle manière que la simulation puisse reprendre immédiatement (en cas de lecture) ou bien ultérieurement (en cas d'écriture). Cela suppose de sauvegarder au moins une image fidèle de la structure de la plante et de l'ensemble des actions à traiter. Le modèle en cours de simulation devra ajouter la sauvegarde de ses propres données en dérivant les objets à sauvegarder de la classe *ArchivableObject* et en renseignant leur méthode *serialize*.

Les classes de base sont :

Serializer : objet qui hérite de la classe VProcess. Il possède un attribut qui pointe sur la plante en cours de simulation.
ArchivableObject : objet qui fournit la méthode virtuelle serialize() effectuant la sauvegarde ou le chargement de l'objet.
Archive : objet recevant les requêtes de sérialisation et possédant un attribut indiquant son mode de traitement (input ou output). Une classe *FileArchive* est fournie pour effectuer les entrées-sorties vers un fichier.

3.5 Gestion de paramètres

Tout modèle possède des paramètres de fonctionnement qui permettent d'adapter le comportement du modèle à une situation ou un objectif souhaité. La plupart du temps ces paramètres sont de nature numérique mais peuvent également prendre des valeurs alphanumériques. Nous ne traiterons pas ce dernier cas. La manière d'interpréter les valeurs des paramètres dépend du modèle qui les utilise. En général, lors d'une simulation, les valeurs

initiales des paramètres du modèle à simuler sont obtenues par lecture dans un ou plusieurs fichiers.

Il nous a semblé intéressant de déléguer la tâche qui consiste à lire/sauvegarder des valeurs de paramètres dans des fichiers et de fournir les valeurs de ces paramètres aux modèles à un ensemble d'objets spécialisés dans ce but.

Dans la mesure où ni le nombre ni la teneur des paramètres à utiliser ne peuvent être fixés à l'avance et afin de faciliter leur manipulation, chacun d'entre eux est associé à une chaine alphanumérique qui contient son nom. Cette valeur alphanumérique permettra de retrouver le paramètre.

Un ensemble d'objets paramètres de type paramétrique est proposé. Ils consistent soit en une valeur unique, soit en un tableau de valeurs indexées selon une abscisse, soit en un tableau de valeurs indexées selon une abscisse et une ordonnée. Pour l'instant, deux modes d'interpolation sont proposés entre les points de contrôle : constant ou linéaire.

Les classes de base sont :

ParamManager : objet qui détient une liste de paramètres associés à leur nom et une liste de fichiers de paramètres. Il propose une méthode qui retourne un paramètre en fonction de son nom.
ParamFile : objet qui permet de lire et écrire une liste de paramètres dans un fichier.
Parameter : objet contenant une valeur numérique et fournissant une méthode *val* qui retourne cette valeur.
ParametricCurveParameter : objet dérivé de *Parameter* contenant un tableau de couples de valeurs numériques décrivant une courbe paramétrique.
ParametricSurfaceParameter : objet dérivé de *ParametricCurveParameter* contenant un tableau de triplets de valeurs numériques décrivant une surface paramétrique.

3.6 Chargement et communication entre modules additionnels

Un modèle de simulation de croissance peut être bâti à partir de blocs fonctionnels dont certains sont soit optionnels soit interchangeables. Pour cela la librairie fournit un mécanisme de chargement dynamique des modules et de définition d'interface fonctionnelle entre des objets offrant des points de connexion et d'autres objets susceptibles d'attacher leur traitement à ces interfaces et d'échanger des données entre eux.

Le principe retenu pour assurer le chargement dynamique de blocs fonctionnels (ou modules additionnels ou bien plugins) se base sur la notion de librairie dynamique (DLL sous Windows, Shared Objects sous Linux/Unix). Un programme peut lors de son exécution ouvrir ces fichiers particuliers et construire des objets par exécution de fonctions déclarées dans ces fichiers. Vitis propose un objet *CPlugin* qui permet d'ouvrir un fichier de librairie dynamique (méthode *Load*), de rechercher une fonction dont le nom est « *startPlugin* » et d'exécuter cette fonction qui retourne une instance de l'objet contenant le bloc fonctionnel proposé le module (méthode *Start*). Ce mécanisme permet, à partir d'un programme de base, de charger dynamiquement des composants qui viennent agrémenter les fonctionnalités du noyau sans avoir à recompiler l'ensemble. On peut ainsi bâtir à volonté des applications différentes en assemblant des blocs fonctionnels pour peu que ceux-ci dérivent de la classe *CPlugin* de Vitis.

Le principe retenu pour assurer la communication entre modules fait appel aux notions de *notifier* et de *subscriber*. Le bloc fonctionnel qui souhaite offrir une interface déclare le type de message qu'il souhaite envoyer puis un *notifie*r attaché à ce type de message qui envoie le message grâce à la méthode *notify*. Le bloc fonctionnel qui désire s'abonner à cette interface déclare un simplement un *subscriber* du type de message qui l'intéresse et renseigne sa méthode *on_notify*. Chaque fois que le message sera envoyé par l'interface, la méthode *on_notify* du *subscriber* sera appelée.

Par défaut la librairie offre une interface qui informe les clients éventuels de son activité. Un message spécifique est envoyé à chaque fois qu'une plante, une branche ou un élément de branche est créé. Un message est également envoyé en début ou fin du calcul de la géométrie de la plante ou d'une branche et à chaque fois qu'une composante de la géométrie (position, longueur, diamètre) d'un élément de branche est calculée. Enfin un message est envoyé à chaque fois qu'une valeur de paramètres est demandée au gestionnaire de paramètres. Le contenu de ces messages permet au client d'éventuellement modifier la valeur qui vient d'être calculée. Cela signifie concrètement qu'il est possible de modifier le comportement d'un simulateur hébergé par Vitis soit en intervenant sur la valeur des paramètres qui lui sont retournées ou bien en transformant la géométrie qui est calculée par défaut. Nous verrons des applications de l'utilisation de cette interface dans le chapitre qui décrit l'implémentation d'AmapSim dans Vitis.

Pour avoir des informations détaillées sur le contenu de cette interface, voir la documentation du fichier VitisCoreSignalInterface fournie à l'adresse *http://umramap.cirad.fr/amap2/logiciels_amap/index.php?page=documentation-4/docVitis.doc*.

Les deux classes fournies sont :

Notifier : objet fournissant un point de connexion sous la forme d'une méthode *notify* (*status*, *data*). *Status* permet de définir la nature du point de connexion, *data* contient les données échangées.
Subscriber : objet établissant une connexion avec une interface. Il fournit la méthode subscribe*()* qui permet d'établir la connexion avec l'interface et la méthode *on_notify (status, data)* qui sera déclenchée quand l'interface appelle *notify ().*

3.7 Organisation générale

L'ensemble des classes de Vitis fournit un jeu de données et les managers qui permettent de les gérer. Elles proposent une description de plante, un ensemble d'actions et un jeu de paramètres. A priori, un simulateur développé dans Vitis accédera à ces informations au travers des managers. De la même manière, les services de sortie et de sérialisation sont disponibles de manière native puisqu'ils s'adressent à la représentation interne de la plante et aux objets fournis par Vitis ou bien dérivés d'eux. On obtient une organisation de la forme :

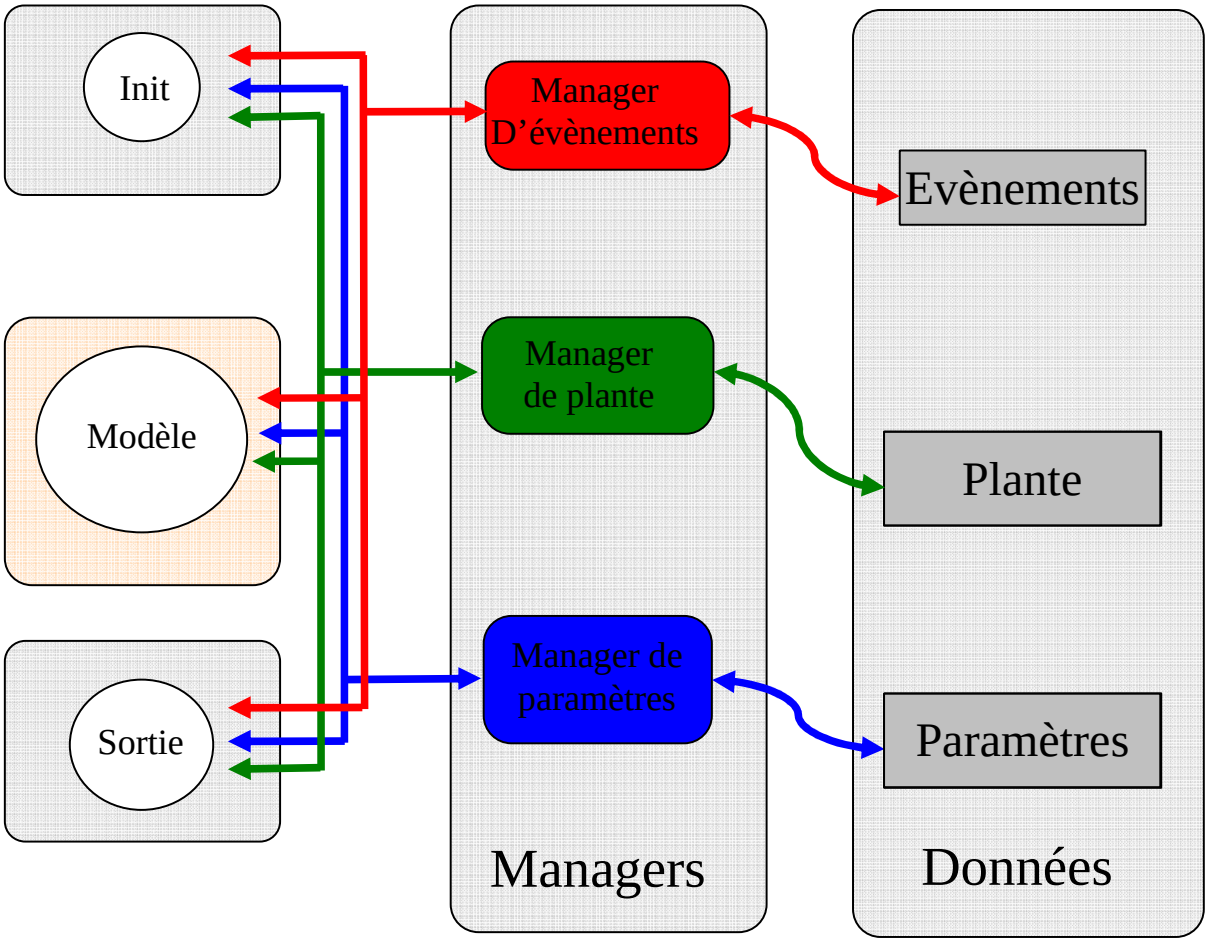


Figure 21.Organisation de la librairie Vitis. Le modèle de croissance est client des services offerts par la librairie.

On remarque que le modèle, la partie active du simulateur, est client des fonctionnalités offertes par les managers. Il édifie et parcourt la plante grâce au manager de plante, accède a ses paramètres au travers du manager de paramètres. Il est important de noter qu'il n'accède pas directement aux données, il passe par une interface logicielle. De plus les fonctionnalités offertes par les managers peuvent également être disponibles pour d'autres blocs fonctionnels. On peut par exemple imaginer que la plante en cours de simulation soit édifiée par plusieurs modèles dédiés à différentes de ses parties (partie caulinaire et partie racinaire par exemple) ou bien qu'un module indépendant accède à la structure de plante à des instants précis (module de taille par exemple).

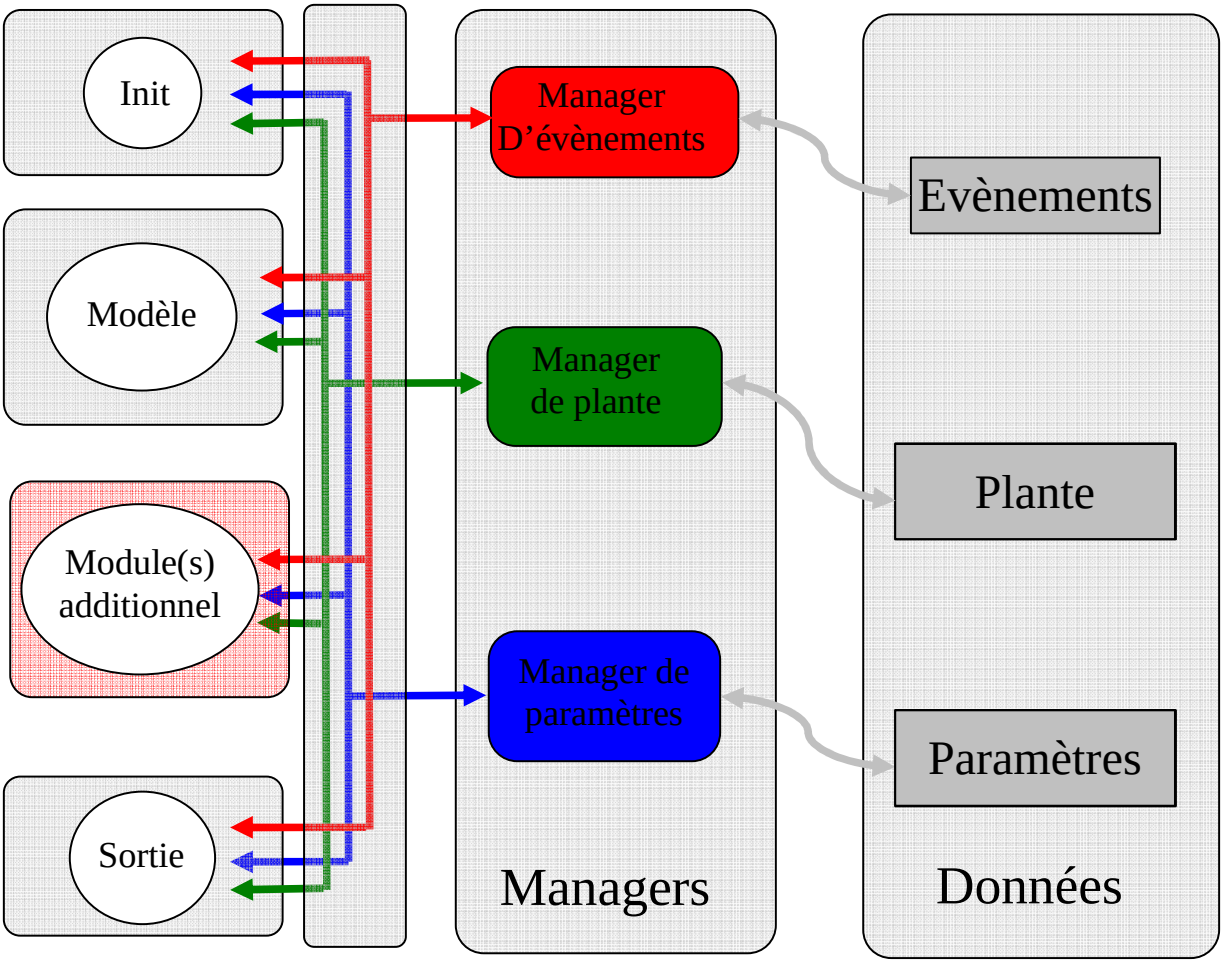


Figure 22. Organisation de la librairie Vitis. Les services de la librairie peuvent être partagés par plusieurs modules..

Dans le cas où le simulateur se compose de plusieurs blocs fonctionnels se posent éventuellement deux problèmes : celui de la synchronisation et celui de la communication entre ces modules.

3.7.1 Synchronisation des modules

Le manager d'évènements propose un mécanisme de synchronisation temporel. Pour cela il faut que les modules fonctionnent sur la même base de temps et que leurs traitements soient découpés de manière à permettre d'être simulés par évènements discrets. Dans ce cas chacun des modules fournit au manager d'évènements les actions qu'il souhaite exécuter aux instants souhaités, cet abonnement pouvant être effectué en cours de simulation.

L'interface fonctionnelle proposée par défaut par la librairie fournit un moyen de déclencher des traitements de modules sur occurrence d'actions particulières requises au niveau des managers. On a affaire ici à une synchronisation fonctionnelle. Nous rappelons que la librairie envoie un message dans différentes circonstances :

- A chaque fois qu'une plante, une branche ou un élément de branche est créé. Si un module s'abonne à un de ces messages, sa méthode correspondante *on_notify* est déclenchée.
- A chaque fois qu'un calcul de la géométrie de la plante va débuter ou vient de se terminer.
- A chaque fois que la dimension d'un composant de plante est calculée.
- A chaque fois que la position ou l'orientation dans l'espace d'un composant de plante est calculé.
- A chaque fois que la valeur d'un paramètre va être retournée.

Nous verrons dans le paragraphe suivant que ce mécanisme permet également une transmission de données entre modules.

Le mécanisme *Notifier/Subscriber* (paragraphe suivant) permet à chaque module de déclarer sa propre interface fonctionnelle ou bien de se connecter à l'interface d'un autre module qu'elle connait.

3.7.2 Echange de données entre modules

La communication entre modules peut avoir lieu selon deux modalités.

La première est prévue au travers de l'attachement de données privées à la structure de description de la plante (*AdditionalData*). N'importe quel module peut associer des informations qui lui sont propres aux éléments décrivant la plante. Un autre module peut parcourir la plante à son tour et accéder à ces informations complémentaires. On dispose ainsi d'un moyen de communication passif, un module dépose des informations que n'importe quel module peut consulter ou modifier.

La seconde est prévue grâce au mécanisme *notifier/subscriber*. Un message est émis par un *notifier* grâce à la méthode notify qui possède dans sa signature un objet de type *ParamMsg* prévu pour contenir des informations transmises par le message. L'appel de la méthode *notify* déclenche la méthode *on_notify* de tous les *subscribers* qui se sont abonnés à ce type de message et qui reçoit l'objet *ParamMsg* fourni à la méthode *notify*.

3.8 Conclusion sur la librairie Vitis.

Nous proposons donc un ensemble d'outils destinés à aider au développement de modèles de simulation de croissance de plante en capitalisant des traitements qui nous semblent génériques. En particulier la librairie offre des outils de construction et de parcours d'une structure de plante organisée en branches se décomposant au choix du simulateur, de synchronisation temporelle pour une simulation à évènements discrets, d'outils de lecture et d'accès à des paramètres numériques et enfin de sorties de résultats.

Cet environnement permet d'accueillir des simulateurs de modèles de croissance de plante. Il offre de plus la possibilité d'agrémenter dynamiquement ces simulateurs de modules additionnels qui peuvent se synchroniser ou interagir avec le modèle de base. Ces modules additionnels disposent de toutes les fonctionnalités offertes par la librairie, en particulier ils

ont accès à la structure de la plante en cours de simulation. La librairie propose une interface fonctionnelle qui permet aux modules additionnels de suivre les étapes de croissance et d'avoir accès aux paramètres du modèle.

4 Le simulateur de AmapSim

4.1 Architecture logicielle

Le logiciel de simulation du modèle implémente fidèlement les fonctionnalités botaniques et agronomiques proposées dans le modèle. Plusieurs versions ont été développées mais nous ne nous intéresserons qu'à la version qui utilise les fonctionnalités de Vitis.

La plante se met progressivement en place au travers de la production d'une collection de bourgeons virtuels (v_buds). Chacun d'entre eux déroule une même boucle d'organogénèse dont les modalités et les paramètres sont basés sur la définition du modèle botanique et sur le principe d'axe de référence. L'enchainement des traitements de ces différents bourgeons est synchronisé dans le temps grâce au *Scheduler*. Le pas de temps élémentaire d'exécution de la boucle de simulation de croissance est la production de nouveaux entre-nœuds (plastochrone). Ceci signifie que chaque v_bud prend la main à son tour pour mettre en place un nouvel entre-nœud puis rend la main au S*cheduler* qui déclenche le v_bud suivant dans l'ordre du temps. De cette manière la plante simulée se met en place de manière quasi parallèle, la simulation peut être stoppée à n'importe quel moment tout en proposant un état homogène de la plante simulée. Il est important de noter que tous les v_buds tournent le même algorithme avec des âges physiologiques et chronologiques éventuellement différents. Cela signifie qu'à un instant donné de la simulation les valeurs des paramètres qui alimentent l'algorithme d'organogénèse peuvent être différents d'un v_bud à l'autre. Ceci permet d'engendrer des productions et des comportements différents selon l'âge physiologique de chaque v_bud. Pour un même âge physiologique, deux v_buds pourront éventuellement avoir des productions différentes étant donnés les tirages différents pour les processus stochastiques (lois du nombre d'entre-nœuds des pousses, ramification…). La boucle élémentaire de simulation (plastochrone) déroulée par chaque v_bud consiste en trois étapes principales :

- **Test de la production d'un nouvel entre-nœud**. Chaque v_bud prend la main à son tour pour tester l'apparition d'un entre-nœud en bout de l'axe qu'il édifie. Le temps consommé avant d'effectuer un nouveau test est calculé comme l'unité de temps du v_bud divisé par un nombre maximum de tests. Une nouvelle unité de croissance est commencée à chaque unité de temps. Celle-ci présente ou non un caractère polycyclique selon une loi markovienne. En cas de polycyclisme, le nombre de pousses est déterminé selon une simple loi cumulative. Chaque cycle comprend une première zone éventuellement suivie d'une seconde (selon une probabilité). Chaque zone a son nombre d'entre-nœuds calculé par tirage dans une distribution binomiale éventuellement négative. Les temps de latence éventuelle entre deux pousses successives ne sont pas comptabilisés.
- **Test de la ramification.** A chaque fois qu'un nouvel entre-nœud est produit, sa production de bourgeons latéraux est testée. Tous les types de production latérale possibles sont parcourus. Pour chacun d'entre eux quatre modes de ramification sont proposés : immédiate, retardée d'un cycle, retardée d'une unité de temps et traumatique. La ramification immédiate est testée systématiquement. La ramification retardée d'un cycle est testée quand le dernier entre-nœud d'un cycle est mis en place et pour tous les entre-nœuds faisant partie de ce cycle. La

ramification retardée d'une unité de temps est testée quand le dernier entre-nœud d'une unité de temps est mis en place et pour tous les entre-nœuds faisant partie de cette unité de temps. Pour chaque test réussi, un bourgeon latéral est créé, son âge physiologique initial est déterminé selon le type de ramification, l'âge physiologique et la position sur le porteur. Pour un type de production latérale particulière, on a donc une priorité entre les différents types de dynamique de ramification (immédiate, retardée sur cycle, retardée, traumatique) ainsi qu'une exclusion mutuelle. Lorsqu'un entre-nœud porte un bourgeon latéral d'un type de démarrage donné, il est considéré comme ramifié mais ne pourra pas porter de bourgeon latéral d'un autre type de démarrage. Lorsqu'un entre-nœud porte des bourgeons latéraux de types de démarrage différents, il faudra utiliser deux productions latérales indépendantes. Par exemple, chez le caféier, il peut y avoir deux types de bourgeons qui coexistent à l'aisselle d'une même feuille.

- **Test de changement d'âge physiologique et d'arrêt**. Après chaque test les v_bud testent leur mort selon une probabilité. En cas de mort, l'élagage de la branche portant ce v_bud est programmé avec un certain retard par l'envoi au scheduler d'une action spécifique à l'instant voulu. Si le bourgeon ne meurt pas, il s'abonne à une nouvelle boucle de croissance en envoyant une action pour lui-même au scheduler. Si l'unité de temps de ce v_bud est échue, son âge physiologique est recalculé selon une loi semi-markovienne à loi d'occupation binomiale. Normalement la valeur de l'âge physiologique ne peut qu'augmenter ce qui mime le vieillissement du bourgeon. Au cas où le nouvel âge physiologique est inférieur à la valeur courante, il est alors ajusté sur elle. Les cas (rares) de rajeunissement sont prévus et symbolisés par une transition vers un âge physiologique négatif. Dans ce cas le nouvel âge est pris comme la valeur absolue en dehors de toute considération de vieillissement.

Toutes les valeurs des paramètres de ces différents processus sont indexées sur l'âge physiologique courant du v_bud qui est en train de dérouler sa boucle de simulation.

La croissance apicale engendre une succession d'entre-nœuds qui forment des branches. Conformément au modèle, ces branches se découpent en pousses (ce qui est engendré durant une unité de temps) qui elle-même se décomposent en unités de croissance qui à leur tour peuvent comprendre deux zones de croissance (préformation, néoformation) qui se décomposent en entre-nœuds. Ceci conduit le choix du niveau de découpage en *DecompAxeLevel* proposé par la librairie Vitis. Pour s'adapter au modèle le simulateur dérive la classe *PlantBuilder* en *AmapSimModele* afin de produire une décomposition des branches adaptée. La méthode *createInstanceDecompAxe* est surchargée, elle crée des éléments décomposés selon cinq niveau hiérachiques : GROWTHUNIT (niveau 1), GROWTHCYCLE (niveau 2), ZONE (niveau 3), TESTUNIT (niveau 4) et INTERNODE (niveau 5). Le niveau de base *Branc* est dérivé en *BrancAMAP*, il est créé par la méthode *createInstanceBranc* de *AmapSimModele.* Les niveaux GROWTHUNIT, GROWTHCYCLE et INTERNODE sont dérivés de *DecompAxeLevel* en *GrowthUnitAMAP*, *CycleAMAP* et *InterNodeAMAP*. Chacun d'entre eux ainsi que *BrancAMAP* comporte des attributs spécifiques au modèle AmapSim et en particulier leur âge physiologique.

Pour ce qui concerne le calcul de la géométrie de la plante, AmapSim dérive les classes de base fournies par Vitis de manière à implémenter les fonctionnalités particulières définies par le modèle. Il est à noter que le déclenchement du calcul de la géométrie de la plante est laissé à l'appréciation de l'utilisateur. Il est tout à fait possible de mener une

simulation complète en ne demandant pas de calcul géométrique. On ne crée alors qu'une structure topologique de description de la plante. La classe géométrique héritant de Vitis est *GeomBrancAMAP*, elle dérive de *GeomBrancCone* et fournit les méthodes :

- *computeBranc* pour démarrer le calcul de la géométrie d'une branche selon une suite d'éléments coniques.
- *computeTopDiam* pour calculer le diamètre au sommet d'un élément.
- *computeLength* pour calculer la longueur d'un élément.
- *computeInsertAnglePhy* pour calculer l'angle de phyllotaxie à l'insertion d'une branche.
- *computeInsertAngle* pour calculer l'angle d'insertion d'une branche par rapport à son porteur.
- computeBending pour calculer la flexion globale d'une branche.
- *flexionAtNode* pour calculer l'angle de flexion d'un élément par rapport à son précédent.

4.2 Utilisation de la simplification, performances

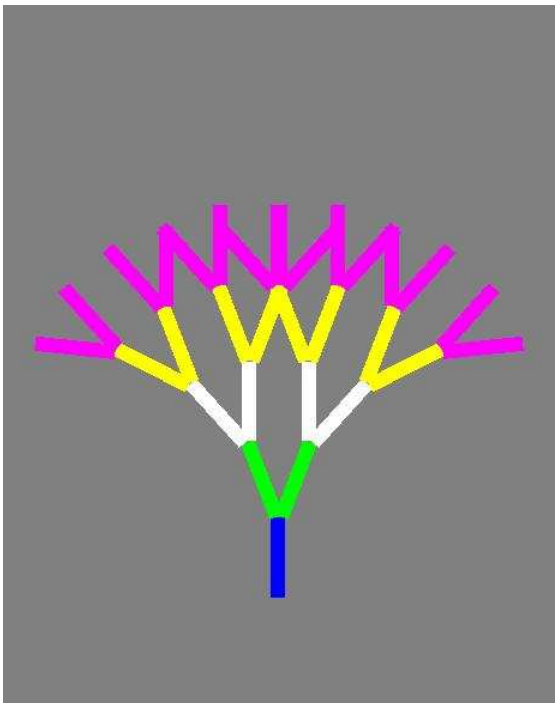
Dans le cas du modèle AmapSim, on peut considérer que deux branches qui démarrent sensiblement au même moment avec le même âge physiologique auront des nombres de pousses et d'entre-nœuds issus de la même distribution. On peut donc constituer des classes de branches dont l'index est construit sur l'âge physiologique initial du bourgeon et sa date de démarrage à un delta près. Le fait de ne pas simuler toutes les branches d'une même classe permet d'économiser de la mémoire et du temps de calcul mais présente l'inconvénient de diminuer la variabilité à l'intérieur de la plante.

AmapSim implémente son propre algorithme de simplification topologique et utilise la simplification géométrique proposée par défaut par Vitis. Parmi les arguments d'appel d'une simulation d'AmapSim figurent le niveau de simplification topologique choisi, la tolérance en décalage de temps et le niveau de simplification géométrique. Au niveau topologique les classes de branches seront définies par leur âge physiologique initial, leur instant de démarrage et le delta de tolérance sur l'instant de démarrage.

La stratégie adoptée a consisté à définir quatre niveaux de simplification topologique. Au niveau zéro aucune simplification n'est appliquée. Au niveau 1, seuls les organes simples immédiats (feuilles, fleurs ou fruits en général) font l'objet d'une simplification. Au niveau 2, chaque classe comporte deux individus (pour préserver une variabilité minimale). Au niveau 3, chaque classe est représentée par un seul individu, c'est-à-dire que toutes les branches de cette classe seront représentées par cet individu. Cette stratégie est arbitraire, on aurait pu laisser le choix du nombre d'instances par classe à l'utilisateur.

La qualité des résultats risque d'être inversement proportionnelle au gain en temps de calcul. En effet plus le niveau de simplification topologique augmente et plus la factorisation (simplification) devient efficace. De plus cet algorithme se révèlera d'autant plus adapté que la plante simulée possède une architecture auto-similaire. Si on prend l'exemple théorique d'une plante dont chaque bourgeon met en place une seule pousse qui porte en son sommet deux bourgeons latéraux similaires, on voit qu'en appliquant aucune simplification on va avoir un temps de calcul proportionnel au carré de l'âge de la plante alors qu'en simplification maximum le temps de calcul devient proportionnel à l'âge de la plante. A l'opposé, si on prend une autre plante théorique dont chaque bourgeon met en place des pousses successives

qui portent chacune une réitération à son sommet, l'algorithme de simplification sera parfaitement neutre tant sur le temps de calcul que sur la qualité du résultat.



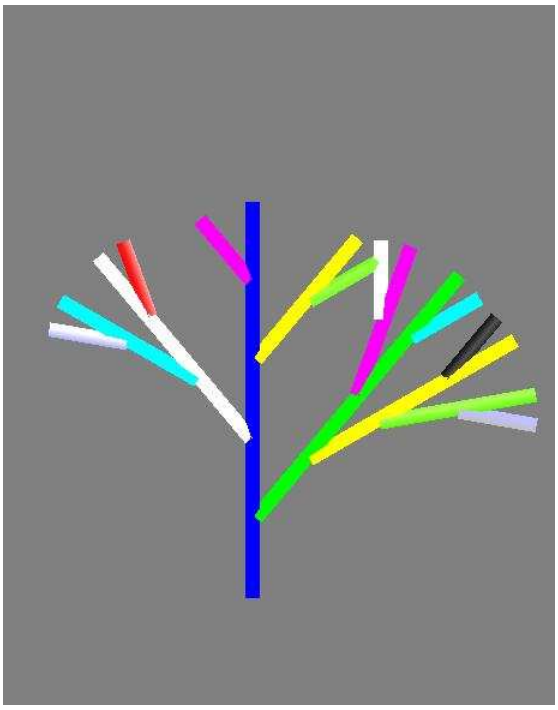
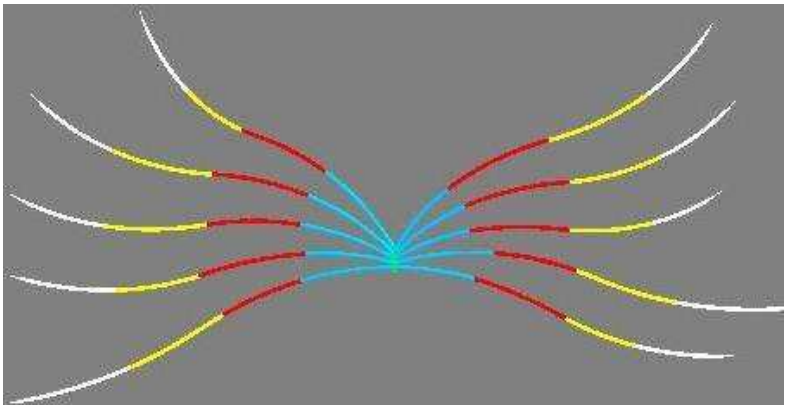


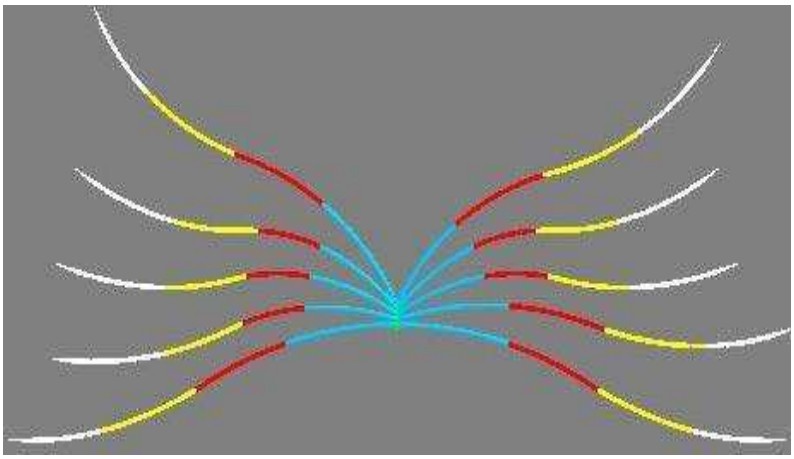
Figure 23. Plante théorique simplifiable Plante théorique non simplifiable

Nous rappelons que le niveau de simplification géométrique est représenté par une valeur numérique entière (cf. page 48). 0 pour aucune simplification, 1 à 3 pour une concaténation des segments de branche selon une tolérance en alignement et positionnement, 5 et plus pour un regroupement des branches d'une même classe topologique selon des secteurs verticaux de direction de démarrage.

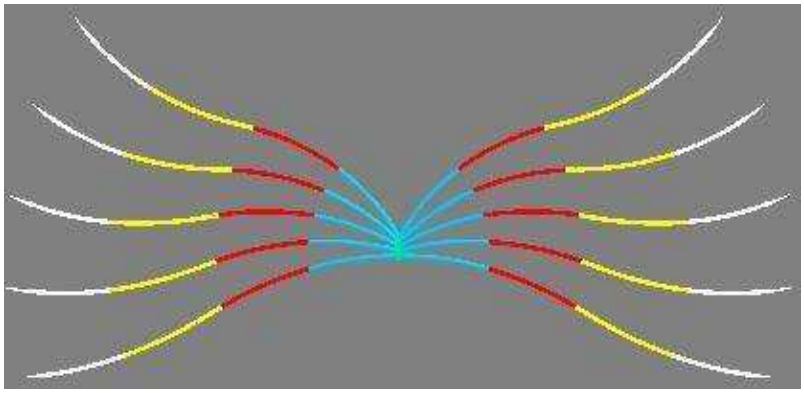
Voici l'exemple d'une autre plante théorique qui permettra d'illustrer l'effet des différents niveaux de simplification. Cette plante (on pourrait plutôt parler de système ramifié) est constituée d'un tronc qui ne comporte qu'une pousse de 10 entre-nœuds, chacun d'entre eux portant une branche. Toutes les branches débutent leur croissance en même temps et avec le même âge physiologique. Chaque pousse est marquée par une couleur spécifique. La production de toutes les pousses est tirée dans une distribution normale de moyenne 40 et de probabilité 0.5.



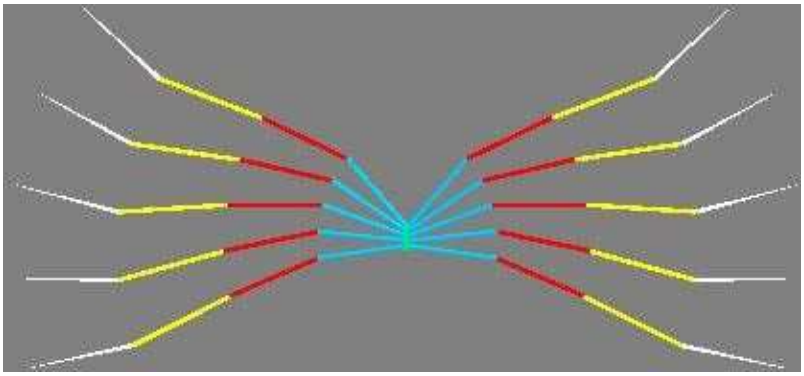
Voici une réalisation de cette plante calculée à 5 ans. On remarque que toutes les branches présentent la même succession de couleurs mais que, du fait des tirages aléatoires, les pousses n'ont pas toutes la même longueur.



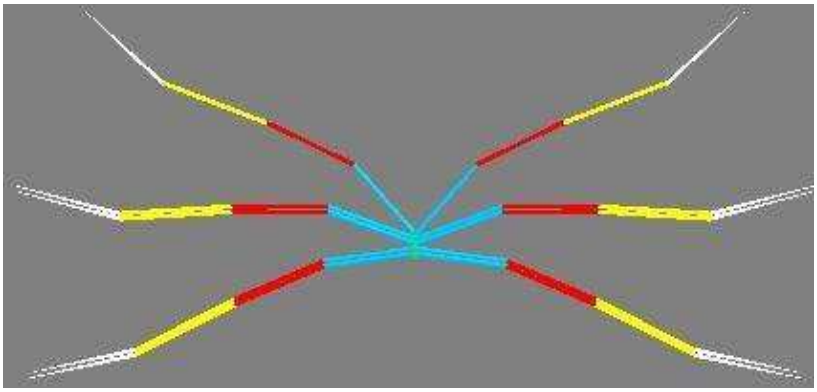
Voici la même plante calculée au même âge mais en lui appliquant une simplification topologique de niveau 2. En regardant attentivement on remarque qu'il n'y a que deux sortes de branches. L'œil conserve une impression de variabilité alors que le statisticien constate une nette dégradation de celle-ci. Le temps de calcul a été divisé par 5.



Voici toujours la même plante que l'on a calculée toujours au même âge mais avec un niveau de simplification topologique 3. Tant l'observateur que le statisticien remarquent la régularité des branches. Le temps de calcul a été divisé par 10.



On applique maintenant une simplification topologique maximum et une simplification géométrique de niveau 1. Les entre-nœuds sont concaténés et l'apparence se dégrade (mais le nombre de polygones dessinés est divisé par 20).



Voici pour finir, la même plante simplifiée au maximum topologique et géométrique. Les branches sont regroupées selon 5 secteurs de direction initiale. L'apparence a radicalement changé mais le temps de calcul de la géométrie a été divisé par 3.

Globalement entre la première image et la dernière, le temps de calcul de la topologie a été divisé par 10, le temps de calcul de la géométrie a été divisé par 3 et le temps d'affichage par 20. Il a fallu pour cela diminuer la variabilité à l'intérieur des branches et accepter un appauvrissement de la qualité géométrique de la sortie.

Cet exemple peut suggérer que les processus de simplification proposés ne sont pas très performants au regard des concessions à accepter pour obtenir un gain peu significatif en performances. Le but était juste d'illustrer le principe de simplification. On peut obtenir des résultats significativement meilleurs sur des plantes qui ramifient plus et pour lesquelles la variabilité est petite. Par exemple, le cèdre qui apparait sur la figure 24 est très autosimilaire et ramifie à l'ordre 4, de plus il ne présente aucune variabilité (toutes ses probabilités de réalisation sont bloquées à 1). Nous avons affaire ici à l'exemple caricatural inverse d'efficacité de la simplification. Les temps de calcul sont divisés par 5000 à 15 ans et il n'y a aucune perte d'information.

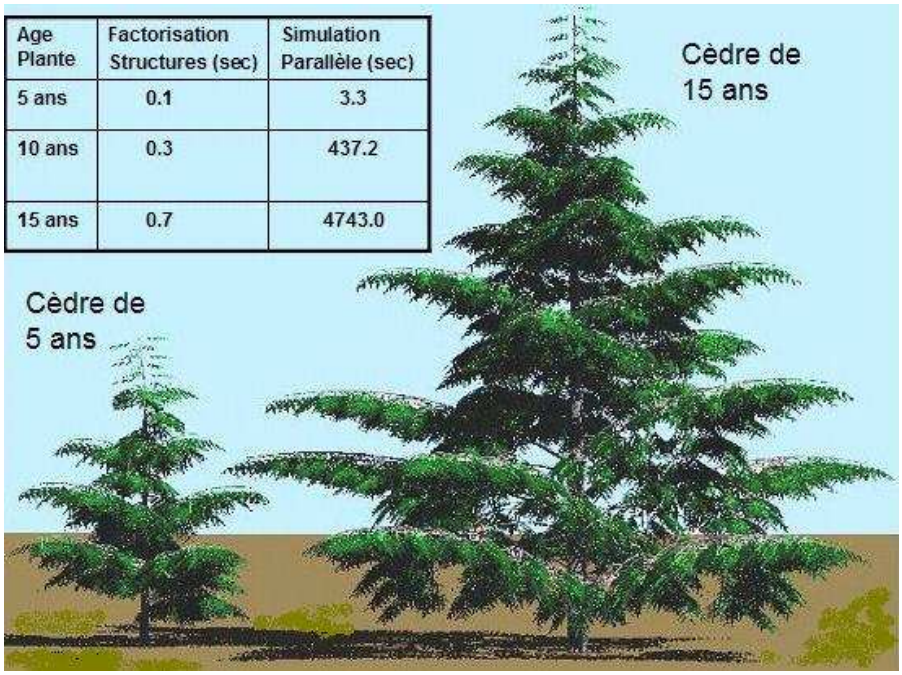


Figure 24. Modèle de cèdre particulièrement adapté à la simplification topologique

Si nous réalisons la paramétrisation d'une espèce de plante plus « raisonnable » et que nous appliquons le même processus de simplification progressif nous obtenons les résultats suivants :

Ce Pin noir est simulé à 25 ans sans aucune simplification. Il est calculé en 7,2s. Cette valeur est une référence pour base de comparaison avec les simulations suivantes. Pour information, elle a été obtenue en faisant tourner le programme de simulation sur un processeur Intel P4 cadencé à 1Ghz avec 2Go de mémoire RAM.

Voici le même pin noir simulé avec un niveau de simplification topologique1 (on ne simplifie que les feuilles). La forme des branches est transformée du fait de la variabilité intraspécifique introduite par le modélisateur. Du fait de la simplification, une même branche ne sera pas simulée avec les mêmes nombres aléatoires pour deux niveaux de simplifications différents. Le temps de calcul passe à : 1,3s.

Voici le même pin simulé en simplification topologique 2 (deux instances de branche sont simulées par classe). Visuellement, la forme architecturale du pin noir est conservée. Le temps de calcul est passé à : 0; 16.

Voici le même pin calculé en simplification topologique 3 (une instance de branche est simulée par classe). Visuellement, la forme architecturale du pin noir est conservée. Le temps de calcul est passé à : 0,08s (la moitié du temps nécessaire en simplification 2).

La même simulation en simplification topologique maximale mais en appliquant une simplification géométrique de concaténation le long des axes. Pour ce point de vue global, on ne voit pas de différence significative avec un arbre sans simplification. Pourtant le nombre de polygone a été divisé par 2 (de 260000 à 120000) et le temps d'affichage d'autant.

Voici le même pin calculé en simplification topologique 3 et simplification géométrique maximale (on ne calcule la géométrie que d'une branche par secteur vertical de 18 degrés et par classe topologique). L'apparence globale reste acceptable. Le temps de calcul de la géométrie a été divisé par 3. Le temps global de calcul par rapport à la simulation sans simplification a été divisé par 6. Ce temps global comprend le calcul topologique de la croissance (on passe de 7,2s à 0,08s), le temps de calcul de la géométrie (de 6,6 à 2s) et le temps de la sauvegarde de la géométrie en fichier (2s)

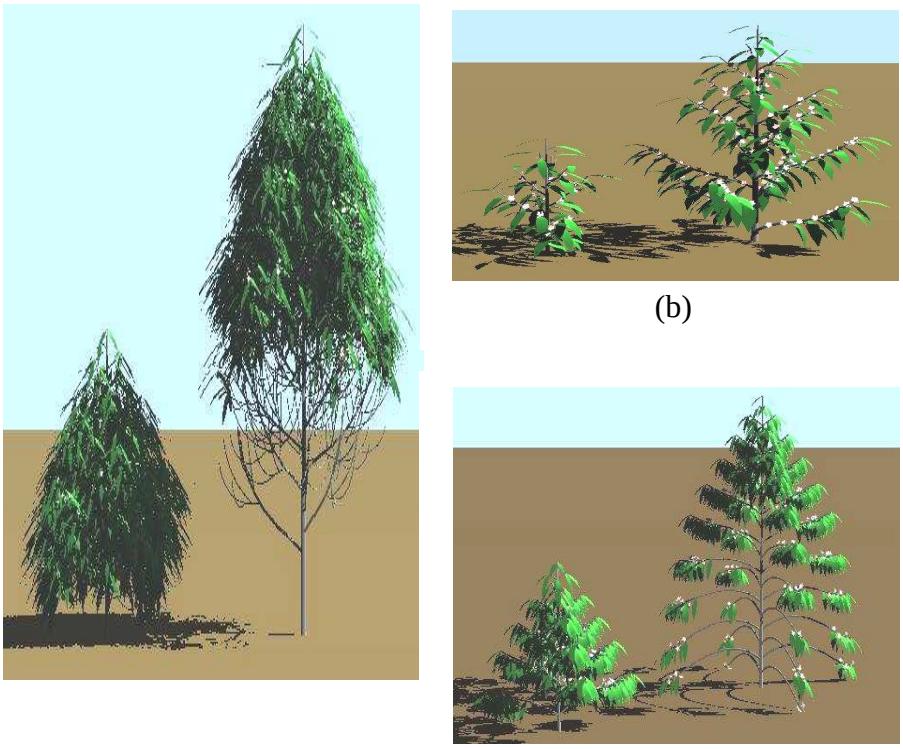
Il va sans dire que le niveau de simplification appliqué à la simulation dépend du compromis que l'utilisateur choisi d'appliquer entre d'une part la reproduction la plus fidèle de la variabilité intra plante et de sa géométrie et d'autre part le temps de calcul et d'affichage. Nous n'avons pas encore trouvé de méthode de simplification de calcul qui offre concomitamment une qualité de sortie maximale et une temps de calcul minimal, mais nous y travaillons ardemment.

4.3 Simulation des modèles architecturaux de base

AmapSim permet de simuler en 3 dimensions avec une assez bonne précision les lois de l'organogenèse des plantes. On peut ainsi reproduire correctement les empilements des axes d'âges physiologiques différents correspondant aux modèles architecturaux de base décrits par Hallé et Oldeman (1974). Voici quelques exemples paramétrés par P. de Reffye.

Modèles avec des unités de croissance simples constituées d'un seul entre-nœud.

Ces modèles correspondent aux modèles d'Attims à branchaison diffuse, de Petit à branchaison sympodiale et Roux à branchaison plagiotrope (figure 25).



(a) (b) (c)

Figure 25. Modèles architecturaux à U.C. simples : (a) Attims (type eucalyptus), (b) Petit (type cotonnier), (c) Roux (type caféier).

Modèles à unités de croissance composée, constituées de plusieurs entre-nœuds

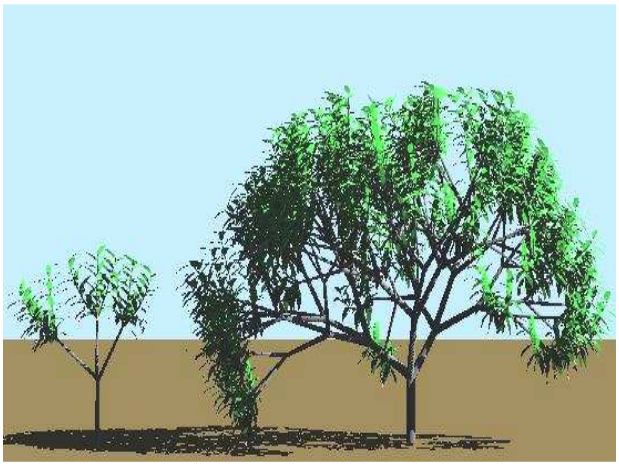


Figure 26. Modèle de Leeuwenberg (Frangipanier)

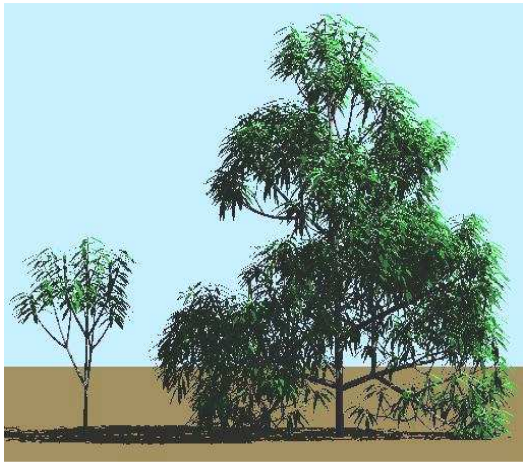


Figure 27. Modèle de Scaronne (Manguier)

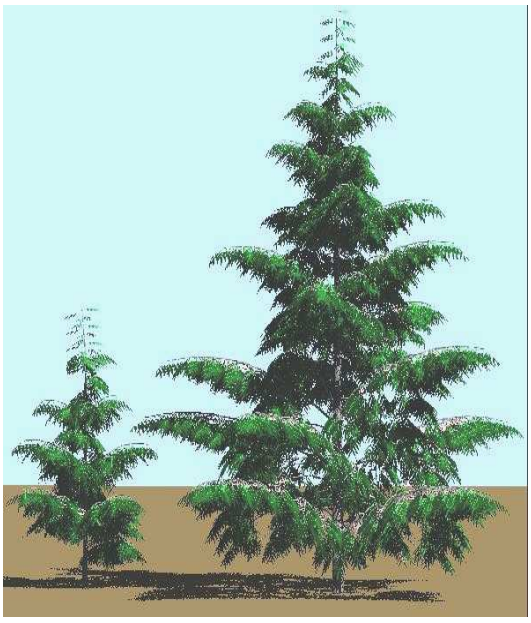


Figure 28. Modèle de Massart (Cèdre)

Figure 29. Modèle de Troll (Albizia)

Figure 30. Modèle de Mangenot

Figure 31. Modèle d'Aubreville
(terminalia)

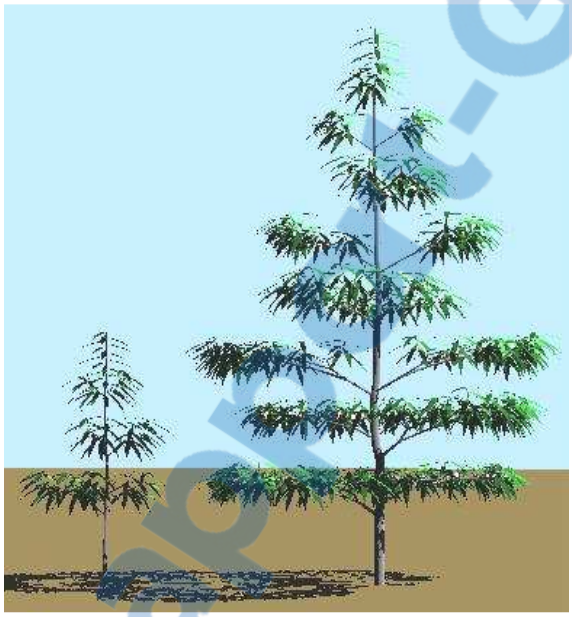


Figure 32. Modèle de Prévot

Figure 33. Modèle de Rauh
(Peuplier)

4.4 Utilisation des maquettes végétales

La modélisation et la simulation de l'organogenèse ne sont pas suffisantes pour obtenir des maquettes végétales. On obtient la production du nombre d'organes en fonction du temps. Il faut y ajouter des mesures géométriques (longueurs, angles) associées aux âges physiologiques sur la dimension des organes. Ici cette géométrie est plaquée et non le résultat de la photosynthèse et de la distribution des hydrates de carbone. L'obtention d'une maquette végétale nécessite un travail de relevé des mesures sur les plantes qui peut se révéler coûteux en temps et en personnel. Certains paramètres sont cachés et doivent faire l 'objet d'un calcul pour leur estimation. C'est le cas de tous les paramètres des lois stochastiques qui sont calculés à partir des distributions observées. D'autres comme les longueurs et les angles sont mesurés directement sur la plante.

Le simulateur de AmapSim a pu fabriquer des maquettes 3D suffisamment précises pour pouvoir être utilisé dans différents domaines où l'obtention d'une architecture précise est nécessaire :

4.4.1 Utilisation en Génétique

Les paramètres architecturaux qui contrôlent l'architecture peuvent être utilisés pour étudier la variabilité génétique dans une famille. Ainsi une expérience a été réalisée sur des tabacs ornementaux avec la Seita.

Figure 34. Simulations de 3 tabacs ornementaux (collaboration Cirad-Seita) (Hervé Rey)

4.4.2 Utilisation en Agronomie

La variabilité architecturale individuelle a pour conséquence une variance non négligeable du rendement d'une plante à l'autre. La simulation des aléas par des processus stochastiques mesurés sur l'architecture des plantes permet de rendre compte de cette composante et de l'intégrer dans l'expression de la production. Deux espèces ont été particulièrement étudiées au Cirad : le cotonnier et le caféier.

Figure 35. Simulation de deux cotonniers d'une même variété. (P deReffye)

Figure 36.Simulation d'un caféier robusta cultivé sur 3 tiges. (P deReffye)

Une autre espèce a été étudiée à la fois pour son système aérien et pour son système racinaire : le palmier à huile.

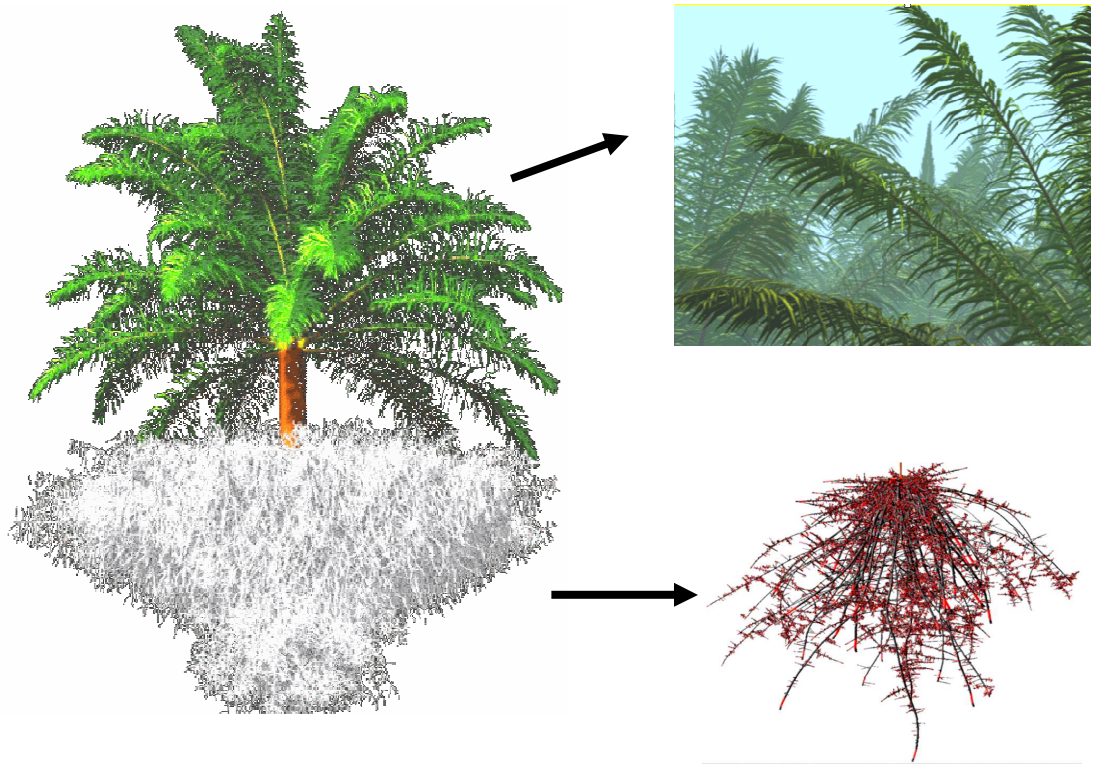


Figure 37. Simulation des architectures aériennes et souterraines du palmier à huile (H. Rey et Ch. Jourdan)

4.4.3 Utilisation en écophysiologie

L'écophysiologie s'efforce d'intégrer ses connaissances au sein de l'architecture de la plante afin de mieux simuler le fonctionnement de celle-ci. Les maquettes tridimensionnelles fournissent les polygones nécessaires au calcul de l'interception de la lumière ou des maillages pour calculer par des méthodes d'éléments finis la transpiration végétale ou la mécanique de la plante sur pieds.

Les études peuvent être faites non seulement sur une plante individuelle mais aussi sur une plantation reconstituée dans laquelle on simulera par exemple le transfert radiatif. Le but à long terme de ce genre de calcul est de tendre vers des expériences virtuelles dans lesquelles on pourra calibrer les paramètres qui optimisent le fonctionnement d'une plantation. L'amélioration des modèles mathématiques, l'intégration des connaissances pluridisciplinaires et l'augmentation de la puissance des ordinateurs devraient permettre d'atteindre ce but.

Le Pois et le Tournesol font l'objet de développement de modèles structure-fonction en collaboration Cirad-Inra.

Figure 38. Maquettes de pois et de tournesol (Hervé Rey)

L'utilisation de maquettes pour reconstituer une plantation permet d'étudier par exemple le transfert radiatif avec des logiciels de lancer de rayons ou de radiosité. On peut ainsi dans le cas de cultures associées optimiser les densités de plantation afin que les plantes reçoivent suffisamment de lumière autant dans la canopée que sous le couvert.

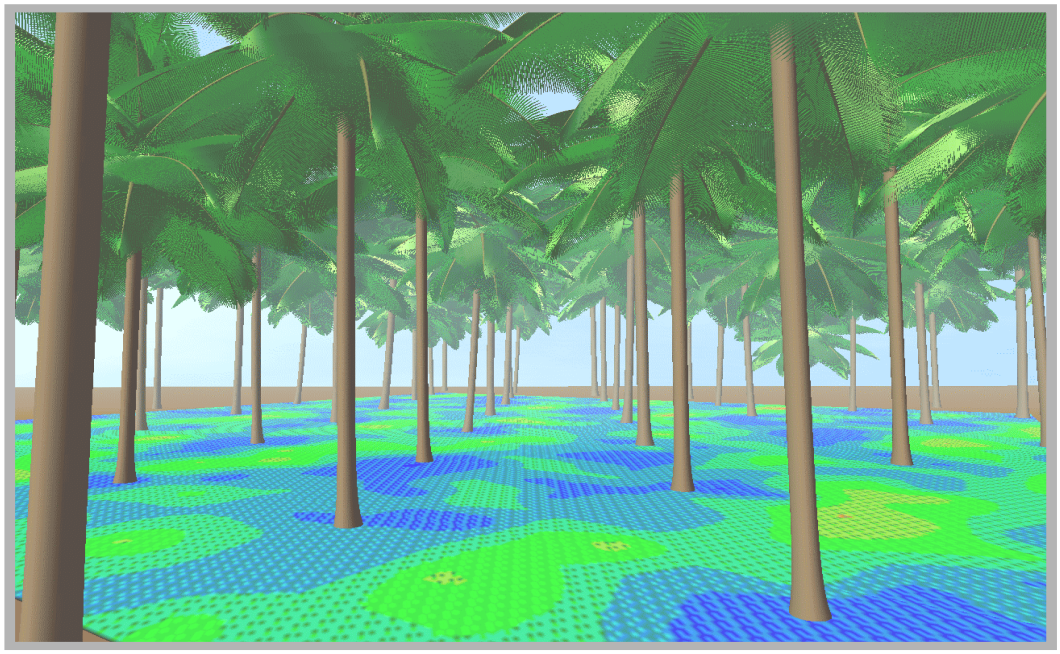


Figure 39. Calcul de l'interception de la lumière dans une cocoteraie simulée (Jean Dauzat)

4.4.4 Utilisation en télédétection

Les modèles d'analyse d'images en télédétection ont besoin d'être calibrés sur des données de terrain pour pouvoir estimer par exemple la biomasse d'une plantation. On sait simuler l'écho radar sur un objet tridimensionnel dont la forme géométrique ainsi que le coefficient de rétrodiffusion de son matériau sont connus. Ce principe appliqué à une maquette végétale permet d'étudier le signal de réponse d'une plante virtuelle et par comparaison de calibrer correctement celui d'une plante réelle. Les images radar de satellites permettent ainsi d'estimer la biomasse d'une forêt (pour l'instant monospécifique) de pins noirs à divers stades de développement. Une telle étude a été réalisée dans le cadre du projet Européen.

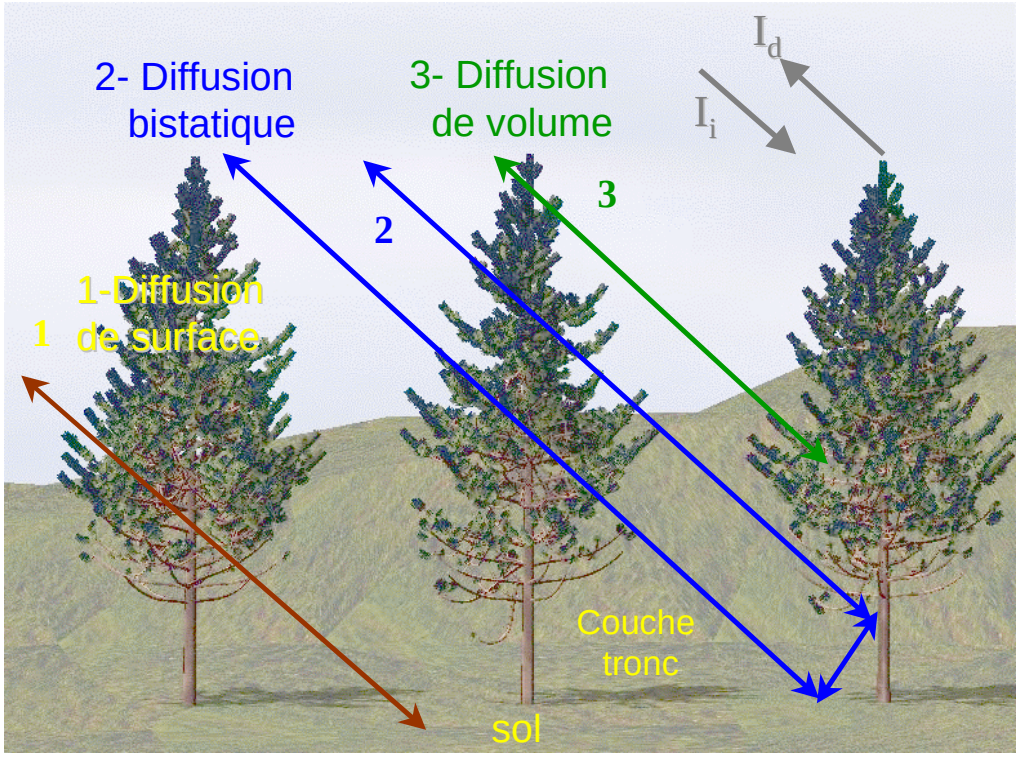


Figure 40. Simulations de l'écho radar sur maquettes de Pins noirs.
(Thierry Castel, Yves Carraglio)

4.5 Conclusion sur la méthodologie de « l'Axe de Référence »

L'utilisation d'une représentation topologique de la notion d'âge physiologique sous la forme d'un axe orienté (Axe de référence) qui intègre tous les états possibles de la différentiation des organes de la graine à la fleur s'avère très pratique. Ce concept intuitif est à la fois descriptif et proche de la Botanique. L'Axe de référence permet de rendre compte de toutes les définitions botaniques liées à l'architecture végétale : (type d'axes, acrotonie, intercalation, réitération…). Les modèles architecturaux se simulent également correctement avec un Axe de référence. Il est relativement aisé pour un utilisateur botaniste expérimenté de coder une plante réelle selon cette technique, d'après les mesures de terrain et en y ajoutant une couche stochastique. Le logiciel AmapSim permet d'obtenir des maquettes 3D suffisamment sophistiquées pour sembler photoréalistes.

Cependant si l'ensemble des paramètres proposés par AmapSim offrent une grande souplesse pour permettre de modéliser n'importe quelle espèce du règne végétal, ils contraignent l'utilisateur à renseigner des valeurs qui ne l'intéressent pas forcément. Par exemple les notions de polycylisme, d'UT ou de néoformation ne concernent pas du tout le paramétrage d'un arbre comme l'eucalyptus qui pousse entre-nœud par entre-nœud. De la même manière, dans le cas des arbres la longueur des entre-nœuds est fixée quasiment à leur apparition au regard de leur durée de vie alors que cette longueur possède une variation dont l'importance devient déterminante si on veut simuler finement la croissance d'une graminée. Il conviendra donc de soigner l'ergonomie des éditeurs de ces paramètres de manière à ne pas noyer l'utilisateur sous une masse d'informations qui ne le concernent pas. Un éditeur spécifique des paramètres d'AmapSim a été développé dans ce sens. Il regroupe les paramètres par catégories (topologie des axes, ramification, géométrie…) dans des pages organisées en onglets. Dans chacune de ces pages, une partie des paramètres peut être cachée ou bien affichée selon le besoin de l'utilisateur comme illustré dans les figures 41 et 42.

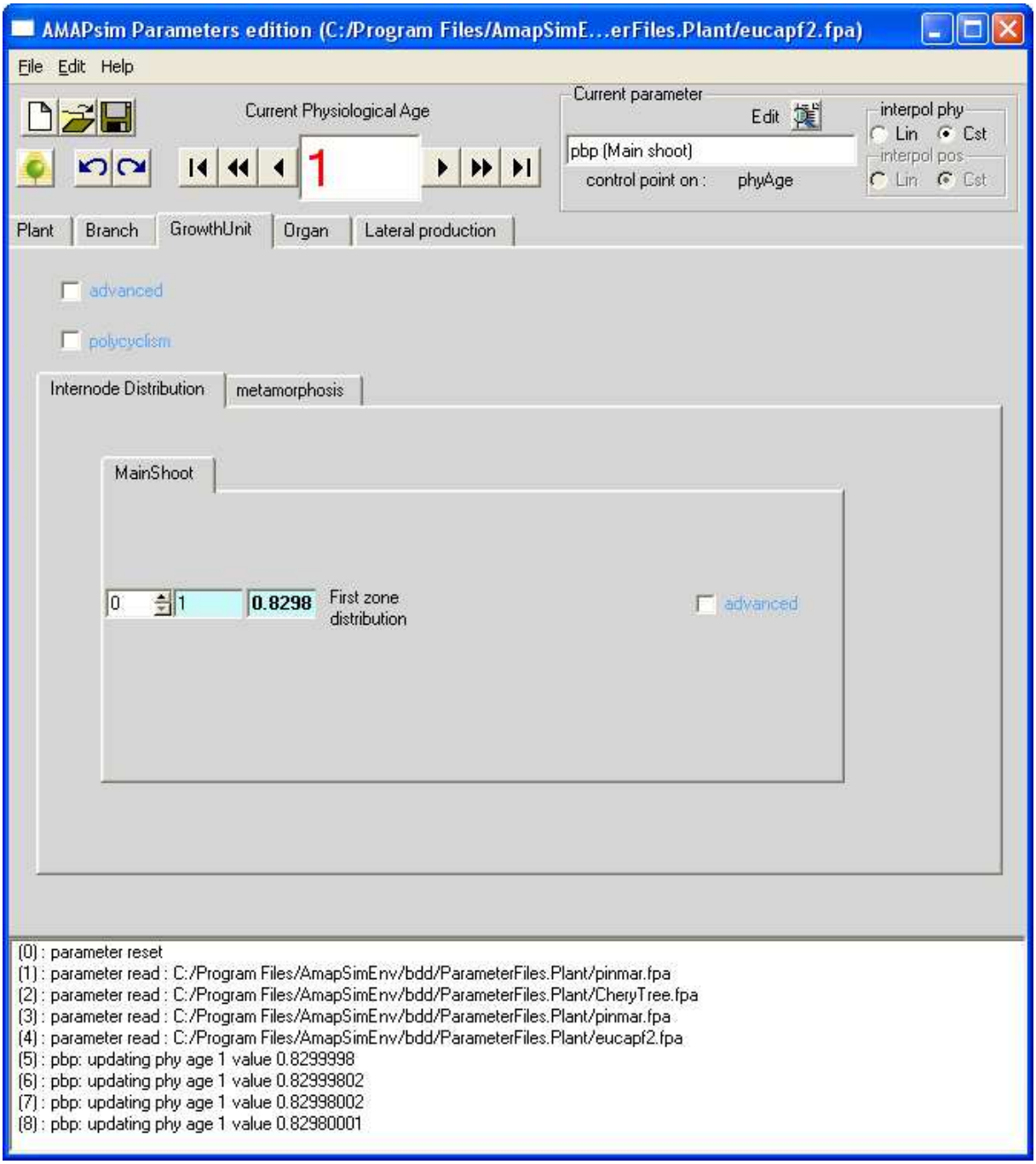


Figure 41. Page d'édition des caractéristiques d'une pousse annuelle d'eucalyptus. Seuls les paramètres de la préformation de l'unique cycle sont affichés.

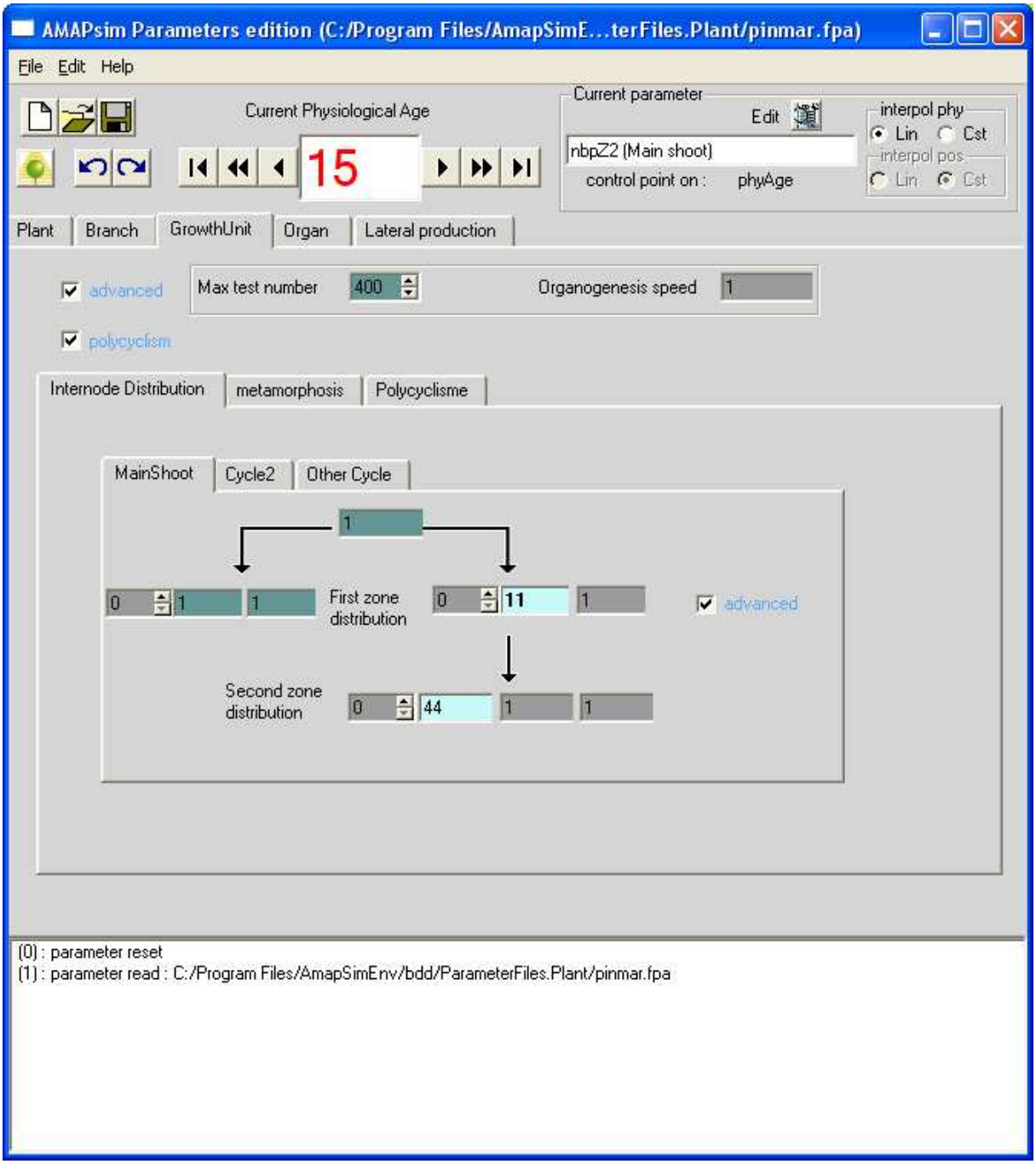


Figure 42. Page d'édition des caractéristiques d'une pousse annuelle de pin maritime. L'utilisateur a accès aux caractéristiques des cycles possibles et, pour chacuns d'eux, aux paramètres des distributions d'entre-nœuds préformés et neoformés..

Pour ce qui concerne la géométrie, les règles retenues permettent de placer les composants de la plante dans l'espace et de contrôler leur dimension de manière globalement satisfaisante mais il arrive souvent que ces règles soient insuffisamment précises pour le cas d'applications spécifiques. Nous verrons dans le paragraphe suivant comment les modules externes permettent de corriger ponctuellement cette faiblesse.

5 Utilisation de l'interface fonctionnelle de Vitis pour AmapSim

Les applications de la simulation de croissance de plantes que nous avons montrées dans les paragraphes précédents utilisent le modèle AmapSim stricto sensu. Dans ces cas, le modèle et son simulateur suffisent à rendre les services attendus. Une fois les valeurs des paramètres d'AmapSim fixées (ce qui est une tâche non triviale car elle nécessite conjointement la connaissance des plantes et la manière de la traduire dans le formalisme Axe de référence), la simulation devient extrêmement simple puisqu'il suffit de choisir le temps de simulation (l'âge de la plante à simuler) et le format de sortie des résultats puis de lancer la simulation.

Cependant, dans beaucoup de cas, si la majorité des sous-modèles proposés par AmapSim satisfont l'utilisateur, un ou plusieurs d'entre eux ne lui conviennent pas. Ceci est lié à une connaissance spécifique d'un mode de fonctionnement de la plante dont AmapSim ne peut pas rendre compte. Pour tenir compte des ces spécificités, se présentent alors trois solutions à l'utilisateur obstiné :

- Chercher un environnement de développement de simulateur mieux adapté à son besoin et y coder le simulateur qui correspond exactement à son idée.
- Coder le simulateur qui correspond exactement à son idée dans Vitis.
- Adapter le comportement d'AmapSim grâce à un ou plusieurs modules additionnels déjà existants ou bien qu'il devra coder.

Nous excluons bien évidemment la première solution qui placerait les idées proposées dans ce document au rebut en même temps que notre fierté. Par contre le choix entre les deux autres solutions dépendra de la quantité d'adaptations à effectuer. S'il a été possible jusqu'à présent, d'adapter le comportement d'AmapSim à tous les cas qui se sont présentés, il est arrivé quelques fois que l'exercice relève de l'acrobatie. Nous pensons en particulier à la modélisation des systèmes racinaires qui s'adapte très mal à la décomposition des axes en entre-nœuds et pour laquelle un modèle dédié serait probablement souhaitable. Ceci étant posé, dans la majorité des cas que nous avons rencontrés l'essentiel du contenu d'AmapSim convenait à la simulation envisagée et uniquement une petite partie devait être adaptée pour remplir le cahier des charges. Il se trouve que cette approche a permis de valider la méthode qui consiste à proposer un modèle de simulation par défaut et d'agir sur celui-ci grâce à l'adjonction dynamique de connaissance propre à adapter la simulation.

Les paragraphes suivants décrivent des applications qui se basent sur l'implémentation d'AmapSim dans Vitis et pour lesquelles un ou plusieurs modules externes ont été développés et connectés afin de produire un simulateur adapté à un besoin spécifique. A chaque fois nous expliquerons brièvement les objectifs en fournissant des références pour un éventuel complément de détails, nous nous étendrons un peu plus sur les solutions logicielles retenues pour atteindre ces objectifs.

5.1. Effet géométrique du vent.

Une première application très simple consiste à agir sur la géométrie de la plante simulée en fonction de contraintes mécaniques liées à la pression horizontale appliquée sur ses branches par un vent virtuel. Cette application rend compte de mesures effectuées sur des *Pinus pinea* et desquelles a été tiré un modèle empirique. Une partie ces pins était soumise à des conditions de vents faibles sans effet sur la croissance des arbres et l'autre partie à des conditions stressantes (plus de 40km/h en moyenne tout au long de l'année). Les mesures montrent que les individus « ventés » présentent une taille globale plus faible, un tronc vertical et des branches déviées dans un plan horizontal d'autant plus qu'elle sont perpendiculaires à la direction du vent et déviées dans un plan vertical d'autant plus qu'elles sont face au vent (figure 43). Les coefficients à appliquer à la dimension et aux déviations doivent être calibrés sur des simulations.

Figure 43. Effet du vent sur la forme de *Pinus pinea*. A gauche, en situation peu ventée. A droite en situation fortement ventée.

AmapSim propose par défaut un calcul de la flexion des branches qui ne tient compte que d'une déviation dans un plan vertical et qui simule l'effet de la gravité selon une analogie à une poutre conique chargée ponctuellement. Il s'agit donc d'ajouter des fonctions propres à prendre en compte l'effet du vent sur la taille de l'arbre et sur la déviation due à sa pression sur les branches.

Pour cette application le modèle additionnel de déviation est extrêmement simple et fonctionne de manière purement géométrique. Nous verrons que les résultats obtenus sont malgré tout d'une qualité visuelle très acceptable. La déviation horizontale est calculée pour chaque élément de branche comme une fraction de la force du vent multipliée par la valeur absolue de la norme du produit vectoriel de la direction du vent et de la direction horizontale de l'élément. La déviation verticale est calculée pour chaque élément de branche dont la direction est dans le secteur du vent comme une fraction de la force du vent multipliée par le produit scalaire de la direction du vent et de la direction de l'élément.

Un module additionnel a été développé pour implémenter ce modèle. Sa fonction *StartPlugin* lit un fichier contenant trois valeurs numériques représentant la force du vent et les coefficients à appliquer aux déviations horizontale et verticale.

Le module s'abonne à un message envoyé par Vitis : *messageVitisElemPosition*, il déclare donc un *Subscriber* de ce type de message. Chaque fois qu'AmapSim demandera le

positionnement d'un élément au manager de plante, celui-ci invoquera la méthode on_notify du *Subscriber* du module. La méthode *on_notify* reçoit en paramètre un objet de type *paramElemPosition* qui contient l'adresse de l'élément concerné et la position calculée par AmapSim. La direction de l'élément est recalculée selon le modèle décrit ci-dessus et vient modifier celle fournie par AmapSim. La figure 44 montre un exemple de simulation de la même plante par AmapSim avec et sans ce module additionnel.

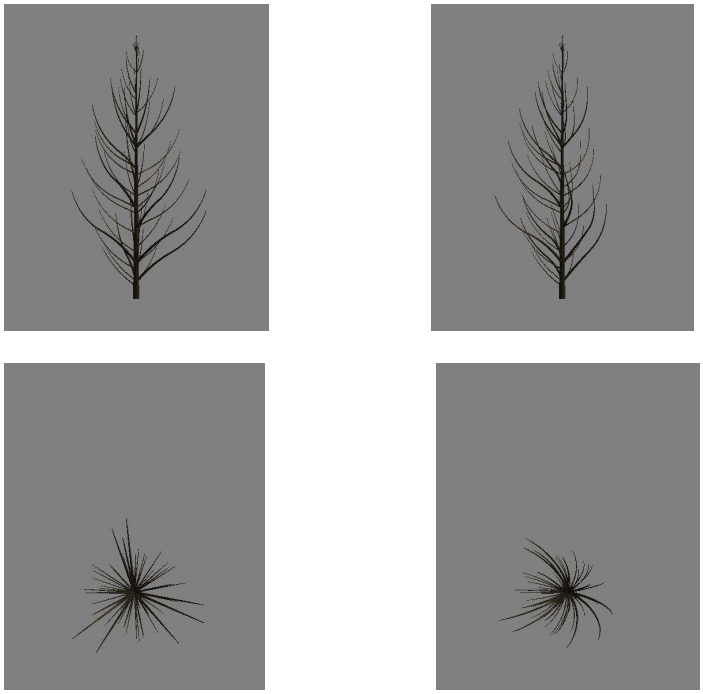


Figure 44. Simulation de l'effet du vent sur une forme de pin simplifiée. A gauche, en situation peu ventée. A droite en situation fortement ventée. En Haut une vue de côté, en bas une vue de dessus.

5.2. Flexion des feuilles de céréales.

Cette application est destinée à la calibration de la réponse d'un capteur de télédétection dans le but de qualifier l'occupation de parcelles occupées par des céréales. Pour ce genre de plante, les feuilles constituent l'essentiel du contenu du signal mesuré et leur géométrie est donc probablement déterminante. La contrainte consiste à contrôler finement la courbure et la torsion des feuilles de différentes variétés de céréales modélisées dans AmapSim.

La torsion des feuilles doit être contrôlée par trois paramètres :

- La position du point de torsion maximale exprimé en pourcentage de la longueur totale.
- La plage dans laquelle cette torsion s'applique exprimée en pourcentage de la longueur totale.
- La valeur totale de cette torsion qui est une fonction de l'âge de la feuille.

La valeur de la torsion à l'intérieur de la plage d'application est une fonction linéaire croissante du début jusqu'au maximum puis linéaire décroissante du maximum jusqu'à la fin.

La flexion des feuilles doit être contrôlée par quatre paramètres.

- La position du point de flexion centrale exprimée en pourcentage de la longueur totale.
- Une plage exprimée en pourcentage de longueur totale définissant la longueur de la plage centrale.
- La valeur totale de la flexion qui est une fonction de l'âge de la feuille.
- Un taux de répartition de la flexion totale entre la plage centrale et la fin de la feuille.

Avant la première zone de flexion, la feuille reste rectiligne. Dans la plage centrale, on applique une flexion en arc de cercle de la valeur affectée à la plage centrale. Le reste de la flexion est appliquée en arc de cercle entre la fin de la zone centrale et l'extrémité de la feuille.

Pour réaliser cette fonctionnalité un plugin a été écrit qui va remplacer le calcul de positionnement fourni par défaut par AmapSim par les fonctions décrites ci-dessus. La fonction *StartPlugin* du module externe lit un fichier contenant les valeurs de ces six paramètres.

Le module additionnel s'abonne à l'inspection du calcul de positionnement dans l'espace d'AmapSim par le message de type *messageVitisElemPosition*. Le fonctionnement général est le même que dans l'application précédente. Dans le cas de ce module elle ne traite que les éléments constituant les feuilles. Pour chaque élément, en fonction de sa position le long de la feuille, un angle de flexion et de torsion est calculé selon le modèle spécifique. On applique les rotations de ces deux angles et on recalcule la matrice de positionnement attachée à cet élément.

Ce module est essentiellement constitué de deux fonctions : *ComputeTorsion* et *ComputeBending* qui sont utilisées pour recalculer l'orientation d'un élément dans l'espace. En l'occurrence nous avons fidèlement implémenté dans ces fonctions le modèle décrit pour les céréales. Il aurait été aussi simple d'un point de vue algorithmique d'implémenter n'importe quel autre modèle de flexion ou de torsion sans changer la structure générale de ce module.

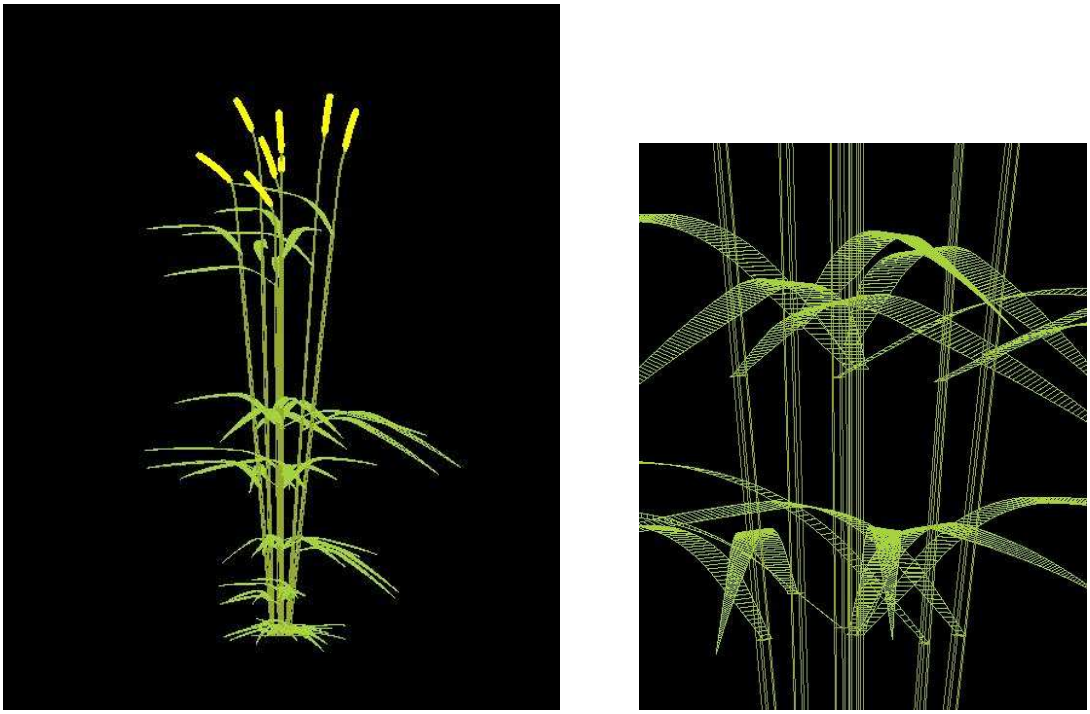


Figure 45. Simulation du seigle. Forme par défaut. Vue globale et détail. (Daniel Spengler)

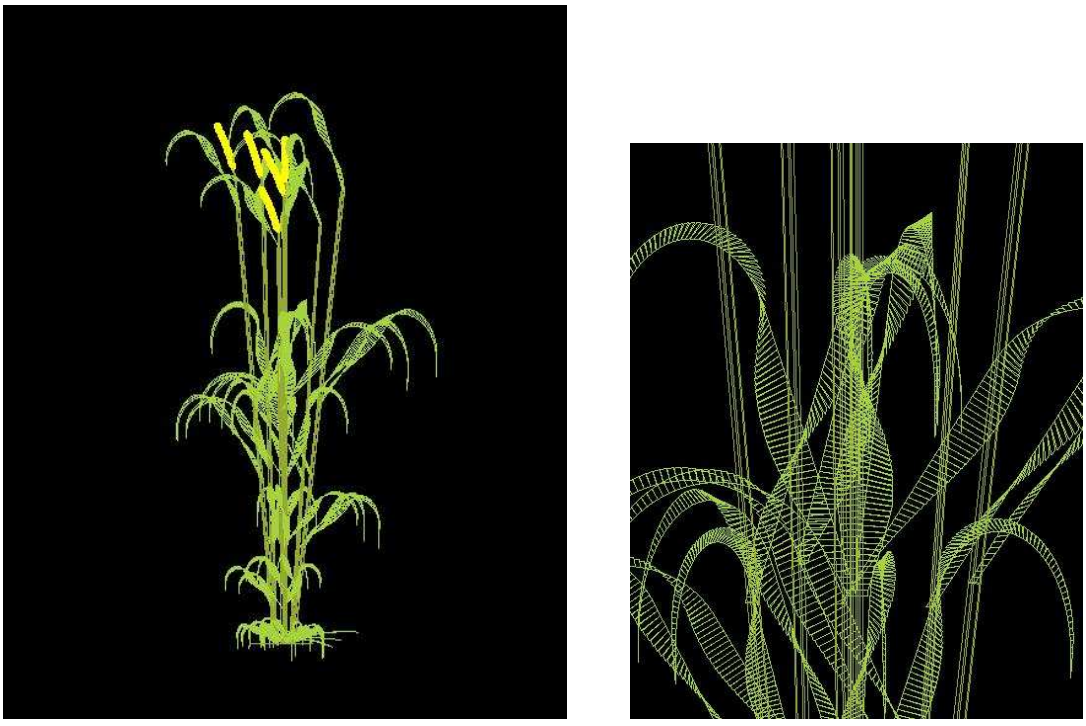


Figure 46. Simulation du seigle. Forme des feuilles avec le module de flexion paramétré de manière « exagérée ». la torsion totale vaut 570° appliquée sur la totalité de la feuille et la flexion totale vaut 180° dont 20% sont appliqués sur une zone de 10% autour du centre en le reste en fin de feuille. Vue globale et détail.(Daniel Spengler)

5.3. Diffusion de la matière selon la loi de Shinozaki (Shinozaki et al 1964).

Par défaut AmapSim propose un mode de calcul du diamètre des branches extrêmement simple et purement empirique : on donne aux composants des axes un diamètre initial qui atteindra une valeur finale selon une progression constante en un temps donné (les diamètres initiaux, finaux et le temps d'accroissement sont indexés sur l'âge physiologique du bourgeon qui édifie l'axe). Ce modèle simpliste permet de rendre compte des observations effectuées sur le terrain mais ne simule en aucun cas les mécanismes physiologiques de production et d'allocation de matière qui permettent d'engendrer ce diamètre. Il y a beaucoup de modèles qui permettent de simuler la photosynthèse et la fixation de matière dans les différents compartiments d'une plante, certains sont extrêmement détaillés et prennent en compte le fonctionnement de la plante avec beaucoup de précision, d'autres sont beaucoup plus grossiers et se trouvent souvent dans le domaine de la foresterie où les arbres sont représentés par leur diamètre à 1,3m, leur hauteur et parfois la forme de leur houppier. Le modèle de Shinozaki (ou *Pipe Model*) fait partie des modèles simple qui permettent d'expliquer en tout point le diamètre des axes d'un arbre en fonction de la masse feuillée qui se trouve en amont (en direction des extrémités des branches) du point de calcul.

Il s'agit ici de calculer à pas de temps régulier (l'année pour les arbres poussant en zone tempérée) la production des feuilles puis d'affecter cette production aux entre-nœuds des branches. Le diamètre des branches sera calculé en fonction de leur longueur et du volume de matière captée par chaque entre-nœud depuis leur apparition. Pour calculer ce volume, on affecte une quantité de matière à chaque feuille de la plante et on répartit cette matière de manière uniforme de l'extrémité de la branche qui porte la feuille jusqu'au collet.

Pour réaliser cette fonctionnalité, un plugin spécifique a été écrit qui s'abonne à quatre messages envoyés par Vitis : *messageVitisDecompAxeLevel*, *messageVitisBeginGeomCompute*, *messageVitisElemLength* et *messageVitisElemDiameter*.

Le message *messageVitisDecompAxeLevel* est envoyé à chaque fois qu'un nouvel élément topologique est créé. La méthode *on_notify* du *Subscriber* reçoit en paramètre un objet de type *paramDecompAxeLevel* qui contient une référence de l'élément qui vient d'être créé. A chaque fois que ce message est envoyé pour un élément au niveau de décomposition le plus bas, la méthode lui attache quatre *AdditionnalData* dont les noms et les attributs sont les suivants.

- « length » qui est associé à une valeur flottante et qui est destiné à contenir la longueur de l'entre-nœud.
- « volume » qui est associé à une valeur flottante et qui est destiné à contenir le volume total de matière reçue par l'entre-nœud.
- « ring number » qui est associé à une valeur entière et qui contient le nombre de cernes accumulés par l'entre-nœud.
- « ring volume » qui est associé à un tableau de flottants et qui contient le volume de chaque cerne. Cet attribut permettra de constituer des profils de tronc en sortie du module.

Le message *messageVitisBeginGeomCompute* est envoyé avant le commencement d'un calcul de la géométrie de la plante. La méthode *on_notify* du *Subscriber* reçoit en paramètre un objet de type *paramGeomCompute* qui contient une référence au *GeomManager* de la plante. Ce *GeomManager* détient la liste de toutes les branches qui composent la plante. La méthode *on_notify* peut ainsi parcourir la structure de la plante et calculer la longueur de chaque entre-nœud (par défaut on utilise le calcul fourni par AmapSim mais il est possible de calculer cette longueur selon une autre méthode), affecter cette longueur au champ additionnel « length », incrémenter le nombre de cernes « ring number » et augmenter la taille du tableau « ring volume » d'un enregistrement. Elle parcourt à nouveau la liste des branches, recherche celles qui correspondent à des feuilles et pour chacune d'elles, calcule la longueur L du chemin qui mène de l'extrémité de la branche porteuse jusqu'au collet et diffuse la matière M qu'elle a produit le long de ce chemin. Chaque entre-nœud de longueur l reçoit ainsi : $m = M \cdot l / L$. Cette masse est ajoutée à l'attribut « volume » et au dernier enregistrement de l'attribut « ring volume ». A la fin de l'exécution de la méthode *on_notify*, tous les entre-nœuds connaissent leur longueur et le volume de matière qu'ils ont accumulé.

Le message *messageVitisElemLength* est envoyé avant d'affecter une valeur à la longueur d'un élément. La méthode *on_notify* du *Subscriber* reçoit en paramètre un objet de type *paramElemGeom* qui contient une référence à l'élément dont on a calculé la longueur et la valeur calculée. La méthode remplace simplement la valeur calculée par celle attachée à l'attribut « length ».

Le message *messageVitisElemDiameter* est envoyé avant d'affecter une valeur au diamètre d'un élément. La méthode *on_notify* du *Subscriber* reçoit en paramètre un objet de type *paramElemGeom* qui contient une référence à l'élément dont on a calculé le diamètre et la valeur calculée. La méthode remplace simplement la valeur calculée par celle correspondant à un cylindre de longueur « length » et de volume « volume ».

5.4. Contrôle de la phénologie du tournesol (Casadebaig 2004)

La température joue un rôle important dans la vitesse d'apparition et la sénescence des organes des plantes. On observe même une relation linéaire entre la somme de températures accumulée et le nombre d'organes mis en place le long des axes des plantes (pour peu qu'on reste dans des gammes de valeurs raisonnables !). Le simulateur d'AmapSim fonctionne à priori selon une base de temps constante qui, étant donnée la considération précédente, ressemble plus à du temps thermique qu'à du temps chronologique. Pour peu que la température moyenne du milieu dans lequel pousse la plante évolue au cours du temps, on observera une production décorélée du vieillissement de la plante.

L'idée consiste ici à asservir la vitesse de développement du tournesol et l'abscission de ses feuilles à la variation de température au cours de la croissance qui sera lue dans un fichier météorologique indexé sur des dates calendaires. Ce travail a été effectué dans le cadre du DEA (autrement dit : master 2) de Pierre Casadebaig.

Pour ce faire la fonction *StartPlugin* du module externe lit un fichier météorologique contenant les températures moyennes mesurées par tranches horaires, les seuils de température accumulée qui déclenchent l'abscission des feuilles et le coefficient directeur Δ de la droite qui relie la somme de température au nombre d'organes de la plante. Il faut

donc accumuler une somme de température $T°C = 1/\Delta$ pour voir un nouvel entre-nœud (et donc une nouvelle feuille) en bout de la tige principale.

Le module externe s'abonne à la demande de valeur des paramètres d'AmapSim par le message de type *messageVitisParam,* il déclare donc un *Subscriber* de ce type de message. Chaque fois qu'AmapSim demandera la valeur d'un de ses paramètres au manager de paramètres, celui-ci invoquera la méthode on_notify du *Subscriber* du module. La méthode *on_notify* reçoit en paramètre le nom du paramètre, sa valeur courante et l'adresse de la branche pour laquelle on vient de calculer cette valeur. Dans le cas de ce module elle ne traite que les paramètres « rythm » et « selfPruningLag ».

Le paramètre « rythm » correspond au rythme d'émission d'une nouvelle pousse. Dans le cas du tournesol il a été choisi que la pousse corresponde à un entre-nœud de la tige principale et que l'unité de temps corresponde à la journée. Si la demande de paramètre concerne la tige principale, il faut, par inspection du fichier météo, calculer l'incrément de temps qui va permettre d'accumuler $T°C$. On retourne l'inverse de ce temps qui vient remplacer la valeur par défaut qu'on trouve dans le fichier paramètre d'AmapSim. Une autre solution aurait pu consister à directement changer la date d'apparition du nouvel entre-nœud dans l'évènement associé au bourgeon correspondant.

Le paramètre « selfPruningLag » correspond au temps à appliquer avant l'élagage d'une branche après son arrêt de développement. Nous rappelons que dans le cas d'AmapSim, toute production latérale d'un entre-nœud est considérée comme un axe. Dans le cas des feuilles le développement est immédiat, c'est le phénomène de croissance qui prend immédiatement le relais et qui dure plusieurs jours. Le module connait le seuil de temps thermique avant élagage pour chaque feuille, il suffit donc, par recherche dans le fichier météo, de calculer le temps nécessaire pour accumuler suffisamment de température et de renvoyer cette valeur à AmapSim.

On voit donc par cet exemple qu'on peut, en s'abonnant au message *messageVitisParam,* modifier le comportement par défaut d'AmapSim pour ce qui concerne les rythmes d'émission et d'abscission de feuille du tournesol et piloter ces phénomènes en fonction de données de température (figure 47).

Figure 47. Abscission de feuilles de *helianthus Melody* contrôlée par l'accumulation de température (Casadebaig, 2004).

5.5. Gestion de la forme des palmes du palmier (Julia 2007)

La production de matière des plantes dépend entre autre de la quantité d'énergie qu'elle capte au travers de la lumière qu'interceptent ses organes photosynthétiques. Quand on cherche à simuler l'interception de la lumière par une plante aussi complexe qu'un arbre, on est confronté au problème de la lourdeur des calculs à effectuer pour s'approcher au mieux de la réalité. On peut essayer de diminuer cette charge en simplifiant les modèles d'interception lumineuse ou en simplifiant l'apparence géométrique des objets qui simulent les plantes virtuelles.

Pour ce qui concerne la simulation de l'interception de la lumière, on peut citer par ordre de complexité décroissante :

La radiosité (Cohen, 1985)
Le lancer de rayon (Spencer, 1962)
La Loi de Beer-Lambert (Monsi, 1953)

Pour ce qui concerne l'apparence des objets géométriques qui simulent les plantes on peut appliquer des simplifications progressives qui vont d'une description détaillée de tous les polygones des organes qui les constituent jusqu'à une agglomération en une « soupe » de

polygones censée approximer le milieu turbide que constitue la masse feuillée de la plante. Tous les niveaux de simplification géométriques intermédiaires sont envisageables.

Etant donné un outil de mesure de l'interception de la lumière, il est important d'évaluer la perte d'information que constitue la simplification géométrique de l'apparence des plantes. Cet exercice a été effectué dans le cas du palmier pour lequel une description fine de ses palmes était nécessaire et qu'AmapSim ne sait pas engendrer.

Les défauts à corriger concernent :
- le profil et la direction du tronc (les spécialistes disent « stipe »). On veut pouvoir ajouter de la variabilité dans la longueur de ses entre-nœuds.
- la longueur des rachis (les béotiens disent « palmes ») ainsi que le nombre de folioles et épines qu'elles portent. Ce qui va amener à remplacer les valeurs des paramètres du calcul du nombre d'entre-nœuds des palmes et de développer un algorithme spécifique de calcul de la distance inter folioles.
- la flexion, la torsion et la déviation des rachis. Pour cela une fonction spécifique permettra de contrôler la forme des palmes.
- la transition le long des rachis entre la présence d'épines puis de folioles. On réalise cette tâche en affectant une valeur spécifique d'âge physiologique aux épines.
- l'orientation en nappe organisée des folioles et des épines le long des rachis. Une fonction spécifique prendra cette question en compte en plus de l'ajustement du paramètre de raideur des folioles en fonction de leur direction initiale.
- la phyllotaxie et l'angle d'insertion des palmes sur le tronc auxquels il faut rajouter de la variabilité.

L'ensemble de ces ajustements est pris en compte par un module additionnel qui possède un jeu de paramètres privés conséquent (72 paramètres) chargés au moment de l'exécution de la méthode *StartPlugin* et qui s'abonne aux messages suivants :

messageVitisParam *:* pour modifier la valeur de cinq paramètres d'AmapSim.

messageVitisDecompAxeLevel *:* Pour calculer et attacher à la description des palmes des données additionnelles qui permettront d'obtenir les bons nombres de folioles et les bonnes distances inter-folioles.

messageVitisBeginGeomCompute *:* pour réinitialiser des compteurs internes du module.

messageVitisEndGeomCompute *:* pour reconstituer les rachis qui sont calculés par moitié gauche et droite indépendantes dans un premier temps.

messageVitisBeginGeomBrancCone *:* pour établir les liens topologiques et géométriques entre les moitiés droite et gauche des demi-rachis. Pour positionner les épines et les folioles du demi-rachis gauche sur le demi-rachis droit correspondant.

messageVitisElemLength *:* pour recalculer la longueur bruitée des entre-nœud du tronc. Pour recalculer la longueur des éléments de palme. Pour recalculer la longueur des épines et des folioles.

messageVitisElemDiameter : pour recalculer le diamètre bruité des entre-nœud du tronc. Pour recalculer le diamètre des éléments de palme. Pour recalculer le diamètre des épines et des folioles.

messageVitisElemPosition : pour recalculer l'orientation des éléments de palme en tenant compte de sa flexion spécifique, de sa torsion et de sa déviation. Pour calculer l'orientation des épines et des folioles en nappe le long des palmes, on dispose de trois orientations des base autour du plan médian de la palme : au dessus, en dessous et dans le plan (voir figure 49).

Nous ne rentrerons pas dans les détails de cette application qui sont décrits dans le document cité en référence. Il nous semble plus important de noter l'ensemble des influences qu'un module additionnel peut avoir sur le comportement par défaut d'AmapSim.

A notre avis ce module additionnel atteint un degré de complexité et un taux de remplacement des fonctionnalités d'AmapSim tels qu'il devient légitime d'envisager d'écrire un modèle entièrement dédié à la simulation de la croissance fine du palmier. Il a malgré tout été intéressant de pousser l'exercice pour voir jusqu'où les fonctions proposées par l'interface fonctionnelle de Vitis permettent à un plugin d'interagir avec un modèle de base. Il n'en reste pas moins que les résultats des simulations sont de bonne qualité (figures 48 et 49)

Figure 48. Simulation d'un palmier et vue d'un réel individu (Julia, 2007)

Figure 49. Détail d'une palme. On contrôle finement la longueur et la courbure de la palme, le nombre, l'espacement et la courbure des folioles en fonction de trois directions privilégiées autour du plan médian de la palme. (Julia, 2007).

5.6. Implémentation de GreenLab

La partie fonctionnelle physiologique du modèle GreenLab a été décrite à la page 23 et a fait également l'objet de l'écriture d'un module additionnel d'AmapSim. Une version simplifiée du modèle a été développée. Elle implémente fidèlement le modèle où la photosynthèse influe uniquement sur la croissance des organes et reste indépendante du développement de la plante (GLII (Yan, 2004)). La physiologie contrôle donc la dimension des organes de la plante mais n'exerce pas de rétroaction sur leur nombre qui est entièrement calculé par les fonctions d'AmapSim. Un essai de faisabilité a été tenté pour implémenter GLIV (Mathieu, 2006). Il s'agit d'introduire la notion de bourgeon comme puits supplémentaire. Ces bourgeons vont contrôler leur développement apical et la création de bourgeons latéraux selon des seuils de matière accumulée. Cet essai donne de bons résultats mais ne reproduit pas fidèlement la partie rétroaction proposée par GLIV. Nous le décrivons malgré tout sachant que l'architecture logicielle du module ne sera pas affectée par l'introduction fidèle des fonctionnalités de GLIV.

La fonction *StartPlugin* du module additionnel lit un fichier contenant les valeurs numériques représentant la force de puits des organes et leur allométrie d'une part, les paramètres de la photosynthèse d'autre part et pour finir la valeur du climat.

Le module s'abonne également aux messages suivants :
messageVitisPlant : ce message est envoyé à la création d'une plante. La méthode *on_notify* du *subscriber* est appelée avec en paramètre un objet detype *paramPlant*. Cet objet contient une référence de l'objet Plant qui vient d'être créé et qui est sauvegardée par le module. La méthode *on_notify* effectue une autre tâche très importante et qui consiste à demander la création d'une action pour la date 1 avec une priorité 5 (élevée) au *Scheduler*. Le *Scheduler* appelera la méthode *on_notify* du module à cette date (voir plus bas la description de la méthode *process_event*). On enclenche ainsi la boucle de calcul régulier de la photosynthèse et de l'allocation de matière de GreenLab.
messageVitisDecompAxeLevel : à la création de chaque branche (élément de niveau de décomposition 0) un objet *AdditionnalData* lui est attaché. Il porte le nom « BudVolume » associé à une valeur flottante. A la création d'un élément de niveau de décomposition maximal (INTERNODE), un objet *AdditionnalData* lui est attaché. Il porte le nom « woodVolume », « leafVolume » ou « fruitVolume » selon le type d'élément, associé à un tableau de flottants dans le cas de « woodVolume », à un tableau de deux flottants dans le cas de « leafVolume » (une pour le pétiole et une pour le limbe) et à une valeur flottante pour « fruitVolume ».
messageVitisParam : C'est grâce à ce message que le lien est réalisé pour faire rétroagir la photosynthèse sur l'organogénèse. En l'occurrence, les paramètres « preformProba », neoformProba » qui contrôlent la probabilité de réalisation des entre-nœuds préformés et néoformés, puis les paramètres « initialBranchingProba », « remainingBranchedProba » et « remainingNonBranchedProba » qui contrôlent le processus de ramification. Lorsqu'un bourgeon dispose d'une quantité de matière M inférieure à un seuil S non nul donné en paramètre, les probabilités sont ajustées par produit par M/S (ou 1-M/S pour le paramètre « remainingBranchedProba »). Encore

une fois, ceci n'est pas le modèle proposé dans GLIV mais simplement un essai de mise en place de rétroaction de la photosynthèse sur l'organogénèse. Il suffit de placer le seuil S à 0 et les puits des bourgeons à 0 pour retourner à une pure version de GLII.

messageVitisElemLength : Le calcul de la longueur depend du type d'element qu'on traite. Dans le cas d'un élément de branche, la longueur calculée par AmapSim est remplacée par une valeur calculée selon une allométrie basée sur le volume de matière affecté au premier calcul de répartition de matière. Les feuilles sont assimilées à un cylindre dont l'épaisseur est un paramètre du modèle, leur longueur correspond au diamètre du cylindre et est déduite du volume du limbe. Les fruits sont assimilés à des sphères, leur longueur correspond au diamètre de la sphère et est déduite du volume du fruit.

messageVitisElemDiameter : Le calcul du diametre depend du type d'element qu'on traite. Dans le cas d'un élément de branche, le diamètre calculé par AmapSim est remplacée par une valeur calculée en supposant que cet élément est un cylindre dont on connait la longueur et le volume total. Les feuilles sont assimilées à un cylindre dont l'épaisseur est un paramètre du modèle, leur diamètre correspond au diamètre du cylindre et est déduite du volume du limbe. Les fruits sont assimilés à des sphères, leur diamètre correspond au diamètre de la sphère et est déduite du volume du fruit.

Finalement le module fournit la méthode *process_event* qui sera appelée par le *Scheduler* aux instants choisis par le module (en l'occurrence, à l'initialisation de la plante puis à pas de temps régulier d'une unité). Cette méthode assure deux tâches. Tout d'abord elle exécute le calcul de photosynthèse et de partage de la matière entre les différents organes de la plante, cette matière est affectée aux champs additionnels « woodVolume », « leafVolume » ou « fruitVolume » selon le type de l'organe ou bien « budVolume » pour les bourgeon en bout de branche, ensuite elle s'enregistre au *Scheduler* pour le pas de temps suivant (t+1). On synchronise ainsi le traitement de la photosynthèse avec celui de l'organogénèse en enchainant les cycles développement-croissance. A chaque demande de calcul de géométrie, la dimension des organes est obtenue par l'interprétation de la quantité de matière qu'ils ont captée.

Dans le cas où les puits et les seuils de stress des bourgeons sont nuls, cette application propose une méthode alternative pour calculer le diamètre des branches d'une plante. La figure 50 montre le résultat de la simulation d'un arbre théorique qui utilise les fonctions proposées par défaut par AmapSim, puis le modèle Shinozaki, puis GLII. Dans chaque cas le nombre d'organes est exactement le même, seule leur dimension est affectée. Cependant on peut dire que le mode AmapSim pur est bien adapté à une simulation où on dispose de peu d'information sur l'explication de la dimension des axes, le mode Shinozaki séduira les forestiers (on s'y intéresse essentiellement au diamètre des branches) alors que le mode GLII sera plus adapté aux applications de type agronomique (tous les organes ont leurs dimensions expliquées par le modèle physiologique).

Figure 50. Simulation du même modèle architectural en changeant le mode de calcul des dimensions des organes. De gauche à droite : AmapSim, Shinozaki, GLII.

5.7. Influence de la densité de plantation sur la croissance de la tomate (Dong, 2005)

On sait que les plantes sont capables de développer des stratégies de croissance propre à optimiser leur objectif de survie en fonction de l'environnement dans lequel elles se développent. La lumière joue un rôle déterminant sur la photosynthèse et donc sur la croissance. En cas de stress lié à un manque de lumière, la plupart des plantes privilégient leur croissance en hauteur en y investissant une bonne part de leurs ressources. Dans le domaine de l'horticulture ce phénomène à une influence sur la taille des fruits (voire sur leur abscission) et donc sur le rendement de production. L'agronome est confronté au problème qui consiste à trouver la densité de plantation optimale car une plantation espacée produira peu de fruits mais de masse importante alors qu'une plantation dense produira beaucoup de fruits mais petits voire pas de fruits si les plantes sont trop serrées.

L'idée consiste ici à essayer de comparer différentes situation de plantation (dense ou peu dense) en terme d'allométrie et d'allocation de matière dans les organes produits par des plants de tomate cultivés en serre. On espère ainsi voir émerger par simulation les stratégies de croissance des plantes de tomate en fonction de leur densité de plantation. Dans chacune des situations, on va calculer la quantité de lumière interceptée par chaque feuille au cours de la croissance et simuler leur production de matière. Un algorithme heuristique ajustera les paramètres du modèle de production sur les mesures effectuées dans les serres. Cette application mêle la simulation du développement du plant de tomate paramétré dans AmapSim et agrémenté du module additionnel simulant GreenLab décrit plus haut et d'un autre module contenant un simulateur de radiosité (Soler *et al*., 2003).

Ce second module est abonné au message *messageVitisPlant* et possède la fonction *process_event*.

La fonction *StartPlugin* du module additionnel lit un fichier contenant les valeurs numériques représentant une scène. On y trouve la position des plantes, les propriétés optiques de leurs classes de composants ainsi que la position et les propriétés des sources lumineuses. Ces valeurs ont été calibrées selon des mesures effectuées dans les serres.

Sur réception du message *messageVitisPlant* , la méthode *on_notify* du *subscriber* est appelée avec en paramètre un objet detype *paramPlant*. Cet objet contient une référence de l'objet Plant qui vient d'être créé et qui est sauvegardée par le module. La méthode *on_notify* effectue une autre tâche très importante et qui consiste à demander la création d'une action pour la date 1 avec une priorité 10 (peu prioritaire) au *Scheduler*. Le *Scheduler* appelera la méthode *on_notify* du module à cette date (voir plus bas la description de la méthode *process_event*). On enclenche ainsi la boucle de calcul régulier de la captation de la lumière par les feuilles des plantes.

Le module fournit la méthode *process_event* qui sera appelée par le *Scheduler* aux instants choisis par le module (en l'occurrence, à l'initialisation de la plante puis à pas de temps régulier d'une unité). Cette méthode s'enregistre au *Scheduler* pour le pas de temps suivant (t+1) puis elle reconstitue la scène en y introduisant la plante courante reproduite autant de fois que nécessaire. Elle calcule ensuite la quantité de lumière interceptée par chaque feuille et leur attache cette valeur grâce à un objet de type *additionnalData*.

Pour cette application, le module GreenLab a été adapté pour calculer la photosynthèse de chaque feuille en fonction de la quantité de lumière qu'elle reçoit plutôt que d'effectuer un calcul global amorti par la loi de Beer.

5.8. Conclusion sur l'utilisation de modules additionnels pour AmapSim

Grâce aux sept exemples décrits ci-dessus, nous avons montré différentes manières d'utiliser les fonctionnalités proposées par Vitis pour adapter le fonctionnement d'AmapSim à des besoins spécifiques. Les modules additionnels ne sont pas forcément dédiés à AmapSim, ils s'appuient essentiellement sur les fonctionnalités de Vitis, on peut donc envisager leur utilisation en connexion à d'autres modèles.

Nous avons vu comment interagir avec le simulateur en utilisant les messages envoyés par les *notifier* de Vitis dans le cas de modification de valeur de paramètre, dans le cas d'ajout dynamique de données de description d'élément de plante et dans le cas de modification de la géométrie des composants de la plante. Nous avons également illustré la possibilité de synchroniser les traitements sur une base temporelle et d'enrichir la description de la plante en cours de simulation avec des données propres aux modules qui sont connectés à AmapSim.

Il est important de noter que, si l'on exclut l'application palmier (à la page 89), l'utilisation des modules externes modifie le comportement d'AmapSim sur des points de détails (qui peuvent se révéler importants pour des applications particulières) et permet de se reposer en grande partie sur le fonctionnement par défaut d'AmapSim. On évite ainsi la

redondance des codes en capitalisant les traitements et en permettant à une application spécifique de ne pas devoir être développée à partir de zéro mais plutôt de s'appuyer sur des outils éprouvés.

Il serait malgré tout prétentieux d'affirmer que l'architecture de Vitis accueillant le simulateur d'AmapSim va permettre de simuler facilement n'importe quelle application de simulation de croissance de plante. Chaque nouvelle situation de simulation devra être analysée pour savoir si la quantité de travail à fournir pour adapter AmapSim au besoin est raisonnable en comparaison du développement d'un simulateur spécifique à partir des fonctionnalités de Vitis.

6 Conclusion

La revue du monde de la simulation de plantes tant en croissance que statique nous a montré l'existence de deux approches diamétralement opposées. D'une part la communauté des modélisateurs spécialistes de la plante qui construisent des prototypes capables de rendre compte de leurs connaissances sans se soucier particulièrement de la qualité de construction de ces prototypes ou bien de leur capacité à évoluer. D'autre part la communauté des « informaticiens » qui utilisent le champ d'étude des plantes pour appliquer les techniques émergentes du génie logiciel ou de l'image de synthèse sans vraiment chercher à fournir des outils opérationnels aux utilisateurs potentiels dans le domaine de l'agronomie.

Nous avons essayé de placer notre étude à l'interface de ces deux mondes en proposant une architecture logicielle adaptée à l'hébergement de simulateur de croissance de plante (Vitis) et en y plongeant un modèle de description d'architecture végétale basée sur des connaissances botaniques (AmapSim). La combinaison de l'approche « architecture logicielle » et du concept « architecture végétale » nous semble amener une solution de base adaptée pour des applications ayant pour préoccupation la description du monde végétal tant en structure qu'en fonctionnement.

Le modèle AmapSim est basé sur la notion d'axe de référence qui classifie par ordre d'âges physiologiques la capacité de production des méristèmes qui constituent la partie active du développement des axes de la plante. Le point de vue adopté est purement dédié à la description de la structure de la plante à l'exclusion de toute notion de fonctionnement. Pour cela des processus stochastiques dont les valeurs des paramètres sont issues de mesures sur des individus réels permettent de simuler l'organisation des axes qui constituent la plante et la structuration de ces axes. L'utilisateur contrôle la ramification au travers de processus semi-markoviens appliqués à un nombre non limité de productions latérales dont la dynamique de démarrage (immédiate, retardée, traumatique…) est également prise compte. Les axes se développent selon un schéma de décomposition communément accepté par la communauté des botanistes. Pour cela le modèle propose un développement des axes en pousses (correspondant à un pas de temps arbitraire) qui peuvent comprendre plusieurs cycles de croissances constitués d'entre-nœuds séparés en deux zones (préformée possiblement suivie d'une zone néoformée). La simulation du nombre de chacun de ces contenants est réglée par des processus stochastiques simples essentiellement basés sur des lois binomiales. A tout moment de la simulation la structure topologique ainsi engendrée peut se voir agrémentée d'une géométrie qui permet de conférer à chaque constituant de la plante une forme, des dimensions et une position dans l'espace.

La simulation de ce modèle rend compte des mesures effectuées sur des plantes réelles avec une grande finesse pour ce qui concerne l'aspect botanique et permet de reproduire le développement et la croissance d'une plante.

Toutefois le concept trouve des limitations car il est lié aux notions d'entre-nœud et d'axe végétatif, toutes notions qui ne sont pas forcément pertinentes pour décrire finement certaines parties de la plante. De plus les méthodes utilisées pour engendrer la géométrie de la plante trouvent rapidement leurs limites pour peu qu'on veuille obtenir dans le détail une forme photoréaliste. Pour finir, la stratégie qui consiste à simuler les mesures mêle sans distinction la capacité de production de la plante liée à son bagage génétique (endogène) et la

contrainte de développement et croissance que constitue l'environnement dans lequel elle pousse (exogène). Nous abordons ici la question de la simulation du fonctionnement de la plante en lien avec son environnement que le modèle AmapSim met délibérément de côté en se concentrant sur l'aspect structure de la description. Il est néanmoins nécessaire d'envisager le couplage d'AmapSim avec des modules fonctionnels qui permettront d'expliquer la valeur de certains de ses paramètres en fonction de connaissances spécifiques sur les modes d'expression de la plante.

Ces limitations nous ont amenés à définir l'architecture logicielle de Vitis de telle manière qu'elle soit propre à accueillir des simulateurs de croissance de plante (dont AmapSim constitue un exemple) et offre la possibilité d'étendre dynamiquement les modalités de production de ces simulateurs tant en termes de formats de sortie que d'interaction avec les simulateurs. Les fonctionnalités que nous avons identifiées concernent trois domaines :

- *La gestion de la représentation de la plante* en cours de simulation. Cette représentation s'appuie sur une hiérarchie d'axes qui peuvent se décomposer au choix du modélisateur. Chaque composant de plante a la possibilité d'être enrichi dynamiquement au cours de la simulation par des données complémentaires.
- *Le chargement dynamique, la communication et la synchronisation de plusieurs blocs fonctionnels*. Cette synchronisation est effectuée soit grâce à l'abonnement à une liste d'actions dont les traitements sont ordonnés dans le temps soit par l'envoi de messages au travers d'interfaces fonctionnelles. Le chargement dynamique de modules s'appuie sur la notion de librairie partagée et d'exécution de fonctions tirées de ces librairies. La communication est réalisée soit au travers de la description de plante soit par l'échange de données contenues dans les messages et dans les actions.
- *La gestion de jeux de paramètres* alimentant des blocs fonctionnels de simulation.

Tout bloc fonctionnel peut proposer sa propre interface logicielle. Vitis propose la sienne par défaut qui informe les clients éventuels de son activité :

- Création de composant de plante.
- Calcul de valeur de paramètre.
- Calcul de caractéristique géométrique de composants (dimension, position).

Nous obtenons finalement une structure générale centrée sur un noyau contenant une boite à outils dédiés à la simulation de la croissance de plante. Cet environnement accueille un ou plusieurs modèles de base propres à assurer la simulation et s'appuyant sur les outils fournis par le noyau. Les applications constituées par ces simulateurs ont la possibilité de charger dynamiquement des modules additionnels qui enrichissent les modèles de manière générique (par exemple pour effectuer des sorties) ou bien de manière spécifique.

Cet outil a été implémenté sous la forme d'une librairie de classes écrites en C++. Nous avons utilisé Vitis pour développer un simulateur d'AmapSim et nous avons proposé des modules additionnels qui permettent d'interagir avec les algorithmes fournis par défaut selon le modèle de base. Nous avons par ailleurs proposé des modules spécifiques pour fournir des sorties selon trois formats particuliers.

Il reste maintenant à expérimenter cette architecture sur la base d'autres modèles. Nous pensons en particulier à un modèle dédié à la description de la croissance du système racinaire des plantes qui se prête mal au formalisme AmapSim. Quand un tel travail sera

réalisé, on pourra naturellement proposer une application qui fera pousser une plante dont la partie aérienne est édifiée par AmapSim et la partie caulinaire par ce modèle « racine » et donc envisager de décrire le fonctionnement de la plante entière.

Pour l'instant, au niveau Vitis, le simulateur de modèle de croissance est vu comme un bloc fonctionnel (qu'on peut éventuellement enrichir par des modules additionnels). D'un point de vue purement technique, en utilisant les outils de synchronisation offerts par Vitis, il doit être possible de construire des simulateurs par agrégation. Par exemple on peut imaginer un module qui prenne en charge la croissance apicale qui se synchronise avec un autre module qui prenne en charge la ramification. Pour aller dans ce sens, un travail de modularisation de simulateur de plante pourrait être effectué et permettre de proposer un ensemble de classes de briques fonctionnelles interchangeables prêtes à être insérées dans un schéma générique de simulation de croissance de plante. On définirait ainsi des « patrons » de simulateurs proposant un squelette dont il suffirait de renseigner les nœuds.

Les algorithmes implantés par défaut dans Vitis pour calculer la géométrie des composants de la plante ont été mis au point en 1987. Ils rendent de grands services mais montrent rapidement leurs limites lorsqu'il s'agit de répondre à des contraintes sophistiquées concernant la trajectoire des branches ou bien la forme d'organes complexes tels que les inflorescences ou les feuilles composées. Nous pensons qu'il y a lieu d'améliorer significativement ces algorithmes afin de proposer des outils plus performants tant en résultat qu'en ergonomie.

Reste la sempiternelle question du choix du principe de simulation et les polémiques qui en découlent. Multi-agents ou bien L-system ? Evènements discrets ou bien simulation continue ? Nous serions plus attirés pour notre part par des systèmes multi-agents à évènements discrets. On retrouve d'ailleurs ces préférences dans les solutions qui ont été adoptées dans Vitis. Cependant il convient de noter qu'un interpréteur de règles à réécriture peut très bien être implémenté à partir de Vitis et qu'AmapSim peut également faire l'objet d'une réécriture (au sens logiciel) selon cette méthode, probablement sans que le reste des outils proposés n'aient à changer. En tant que fournisseur de modèles et d'outils de modélisation, il nous semble important de ne pas faire de choix à priori et définitif sur une méthode plutôt qu'une autre. Il nous semble plus judicieux d'adopter les solutions ouvertes et adaptées tant au niveau de la granularité (on ne choisit pas les même solutions pour simuler un système à 5 composants qu'un système à 10^9 composants) qu'au niveau de la localité (une branche ne se simule pas comme une feuille) qu'au niveau de la temporalité (les systèmes à réécriture sont adaptés à la simulation de phénomènes répétitifs et simultanés tandis que les systèmes à évènements discrets permettent de synchroniser des phénomènes qui ne fonctionnent pas sur les même bases de temps). On peut ainsi envisager de bâtir des systèmes hybrides qui tirent avantage des solutions retenues en fonction des problèmes posés. Nous espérons que Vitis constitue une bonne base de départ pour offrir ce genre de possibilité.

7 Bibliographie

Allen MT, Prusinkiewicz P, DeJong T. 2004. Using L-systems for modeling source-sink interactions, architecture and physiology of growing trees: the L-PEACH model. *New Phytologist* 166(3): 869-880.

Andrieu B, Baret F, Jacquemoud S, et al. 1997. Evaluation of an improved version of SAIL model for simulating bidirectional reflectance of sugar beet canopies. *Remote Sensing of Environment*, 60 (3), 247-257.

Baker, D.N., J.R. Lambert and J.M. McKinion (1989) GOSSYM: A Simulator of Cotton Crop Growth and Yield. *S.C. Agr. Exp. Sta. Tech. Bul*. 1089. 134 pages.

Barczi JF, de Reffye P, Caraglio Y, 1997. Essai sur l'identification et la mise en œuvre des paramètres nécessaires a la simulation d'une architecture végétale : le logiciel AmapSim. In : *Modélisation et simulation de l'architecture des* végétaux. Bouchon J, de Reffye P, Barthélémy D, eds *INRA Editions*, Paris: 205-254.

Barczi JF, Rey H, Caraglio Y, de Reffye P, Barthélémy D, Dong QX, Fourcaud T. 2008. AmapSim: a structural whole plant simulator based on botanical knowledge and designed to host external functional models. *Annals of Botany* 101(8) : 1125-1138.

Barthélémy D, Caraglio Y. 2007. Plant architecture: a dynamic, multilevel and comprehensive approach of plant form, structure and ontogeny. *Annals of Botany* 99: 375-407.

Bertrand M., Lung C., Zaytoon J. 2004. Systèmes hybrides: modélisation et simulation. *Technique de l'ingénieur*, S7 105, 1-14.

Blaise F. 1991. Thèse de l'Université Louis Pasteur de Strasbourg. Simulation du parallélisme dans la croissance des plantes et applications. 186 pages.

Bonhomme R. 2000. Review : bases and limits to using « degree.day » units. *European journal of agronomy* 13 : 1-10.

Booch G., Rumbaugh J., Jacobson I. 2000. Le guide de l'utilisateur UML. Eyrolles. 552 pages.

Bosanac B, Zanchi P. 2002. Onyx Tree Conifer User's Manual. Version 5.1. Onyx Computing Inc., 10 Avon Street, Cambridge, MA .

Bouguer P. 1729. Essai d'Optique sur la Gradation de la Lumière. Claude Jombert, Paris.

Bousquet F, Bakam I, Proton H, Le Page C. 1998. Cormas: common-pool resources and multi-agent Systems. Lecture Notes in *Artificial Intelligence* 1416: 826-838.

Casadebaig P; 2004, DEA de l'Ecole Doctorale Biologie Intégrative de Montpellier, Analyse de la variabilité génétique de l'architecture du tournesol :Conséquences sur l'interception du rayonnement de la floraison à la maturité, 36pages.

Cohen M, Greenberg D. 1985. The Hemi-Cube: a Radiosity Solution for Complex Environments, *SIGGRAPH'85, Computer Graphics*. vol. 19(3) 31-40.

Coquillard P., Hill D.R.C. 1996. Modélisation et simulation d'écosystèmes. Masson.

Cournède PH, Kang MZ, Mathieu A, Barczi JF, Yan HP, Hu BG, de Reffye P. 2006., Structural Factorization of Plants to Compute their functional and architectural growth**.** *Simulation- Transaction of the Society for modeling and simulation international* , 82(7): pages 427-438, 2006.

Dahl O.J., Nygaard K. 1966. Simula – An algol based simulation langage. *Communication of the ACM*. Vol 9. 671-678.

Deussen O, 2003,Computergenerierte Pflanzen,Technik und Design digitaler Pflanzenwelten, Heidelberg (Springer Verlag). ISBN 3-540-43606-5.

De Reffye P, 1979, Thèse de l'université Paris sud Orsay, Modélisation de l'architecture des arbres par des processus stochastiques. Simulation spatiale des modèles tropicaux sou l'effet de la pesanteur. Application au *Coffea robusta*. 195 pages.

De Reffye P., Edelin C., Françon J., Jaeger M., Puech C.. *1988.* Plants models faithful to botanical structure and development. *Computer Graphics, Siggraph'88* : 22 (4):, 151-158

De Reffye P, Dinouard P, Barthélémy D. 1991. Modélisation et simulation de l'architecture de l'Orme du Japon *Zelkova serrata* (Thunb.) Makino (Ulmaceae): la notion d'axe de référence. In: Edelin C, eds. *L'arbre : Biologie et Développement*. Montpellier : Naturalia Monspeliensia, n° h.s., 251-266.

De Reffye P, Hu BG. 2003. "Relevant qualitative and quantitative choices for building an efficient dynamic plant growth model: GreenLab Case. *Plant Growth Modeling and Applications: Proceedings - PMA03*, Hu B.-G. and Jaeger M., (eds.), Tsinghua University Press and Springer, Beijing, China, pp. 87-107

De Reffye P., Heuvelink E., Barthélémy D., Cournède P.H. 2008. Modeling plant growth and architecture. *Encyclopedia of Ecology*, Ed. Jorgensen, S., Publ. Elsevier.

Dong QX. 2005. Thèse de la Chinese Agronomy University. Study on structural-functional model of tomato growth based on accurate radiation transfer. 101 pages.

Feller W. 1968**.** An introduction to probability theory and its applications, volume I, 3e édition John WILEY New-York, 509 pages

Gamma E., Helm R., Johnson R., Vlissides J. (trad. Jean-Marie Lasvergères). 1999. Design Patterns - Catalogue de modèles de conceptions réutilisables. Vuibert. 490 pages.

Godin C, Caraglio Y. 1998. A multiscale model of plant topological structures. *Journal of theoretical biology*. 191 : 1-46.

Gordon G. 1978. System simulation. Prentice Hall-Inc, Engelwood Cliffs, New Jersey, USA.

Guédon Y,Barthélémy D, Caraglio Y, Costes E. 2001. Pattern analysis in branching and axillary flowering sequences. *Journal of Theoretical Biology* 212: 481–520.

Gutknecht O. 2000. Madkit, A generic multi-agent platform, *AGENTS'00 : 4th International. Conference on Autonomous Agents* , 78-79

Hallé F, Oldeman RAA. 1970. Essai sur l'architecture et la dynamique de croissance des arbres tropicaux. Paris : Masson.

Heuvelink E.; Tijskens L.M.M.; Kang, M. 2004. Modelling product quality in horticulture: an overview. *Acta Horticulturae* 654. 19-30.

Hines JR. 1996. "Software Engineering." *IEEE Spectrum* : 60-64.

Jacquemoud S, Ustin S L, Verdebout J, et al., 1996. Estimating leaf biochemistry using the PROSPECT leaf optical properties model. *Remote Sensing of Environment,* 56 (3), 194-202.

Jallas E, Martin P, Turner, Crétenet M, McKinion J, Mohtar RH**.** 2003**.** Plant population vs plant average : COTONS the new paradigm of crop simulation modelling**.** *Resource,* 10 (5).

Jaeger M. 1987. Thèse de l'Université Louis Pasteur de Strasbourg. Représentation et simulation de la croissance des végétaux. 156 pages.

Jones HG. 1992. Plants and microclimate. A quantitative approach to environmental plant physiology. *Cambridge University Press.*

Jorgensen S.E. 1994. Fundamentals of ecological modeling. Elsevier, Amsterdam, The Netherlands.

Julia C, 2007, Mémoire de fin d'études PV 26859, ENITA de Clermont-Ferrand, Modélisation et simulation de l'architecture du palmiers à huile afin de simuler l'interception du rayonnement au sein de ce couvert (Indonésie), 40 pages.

Karwowski R., and Prusinkiewicz P. 2004. The L-system-based plant-modeling environment L-studion 4.0. In *Proceedings of the 4th International Workshop on Functional-Structural Plant Models*. 403-405.

Kniemeyer O. 2004. Rule-based modelling with the XL/GroIMP software. In: Harald Schaub, Frank Detje, Ulrike Brüggemann (eds.), *The Logic of Artificial Life. Proceedings of 6th GWAL, Bamberg April 14-16, 2004*, AKA Akademische Verlagsgesellschaft Berlin, 56-65.

Kang M., Cournède PH, De Reffye P, Auclair D, Hu BG**.** 2008. Analytical study of a stochastic plant growth model : Application to the GreenLab model. *Mathematics and computers in simulation*, 78 (1) : 57-75.

Leroudier J. 1980. La simulation à évènements discrets. Hommes et techniques.

Lindenmayer A., Prusinkievicz P. 1990. The algorithmic beauty of plants. Springer verlag. 240 pages.

Louarn G, Dong QX, Wang Y, Barczi JF, de Reffye P. 2006. Parameter stability of the structural-functional model GreenLab-tomato as affected by plant density and biomass dada acquisition. IEEE *Computer Graphics & Applications* PMA06 proceedings :142-148.

Mäkelä A, Vanninen P, Ikonen VP. 1997. An application of process-based modelling to the development of branchiness in Scots pine. *Silva Fennica* 31(3): 369–380

Marcelis LFM, Heuvelink E, Goudriaan j. 1998. Modeling biomass production and yield of orthicultural crops ; a review. *Scientia horticulturae.* 74 (1/2), 83-111.

Mathieu A, Cournède PH, P. de Reffye P. 2006. A Dynamical Model of Plant Growth with Full Retroaction between Organogenesis and Photosynthesis. *ARIMA Journal*, 4.

Mech R., Prusinkiewicz P. 1996. Visual models of plants interacting with their environment. In *Proceedings of SIGGRAPH 96*. 1996 August 4–9; New Orleans. Edited by Rushmeier H. New York: ACM SIGGRAPH; 1996:397-410.

Minsky M.L. 1965. Matter, minds and models, *International Federation of Information Processing Congress*, Vol. 1, 45-49.

Monsi M, and Saeki T. 1953. Über den lichtfaktor in den Pflanzengesellschaften und seine Bedeutung für die Stoffproduktion. *Jpn. J. Botany* 14, 22–52.

Neuts M.F. 1975. Probability distributions of phase type. *Liber Amicorum Prof. Emeritus H. Florin*, 176–206. University of Louvain, Belgium.

Perttunen J, Nikinmaa E, Lechowicz M, Sievänen R, Messier C. 2001. Application of the functional-structural tree model LIGNUM to sugar maple saplings growing in forest gaps. *Annals of Botany* 88: 471-481.

Pradal C, Boudon F, Dones N, Durand JB, Barbier De Reuille P, Fournier C, Sinoquet H, Godin C. 2006. OpenAlea - A platform for plant modelling, analysis and simulation, *Europython conference*.

Rivals, P. 1965. Essai sur la croissance des arbres et sur leurs systèmes de floraison. *Journée d'Agriculture Tropicale et de Botanique appliquée*. Vol.XII(12), 655-686, Vol.XII(1-2-3), 91-122, Vol.XIV, 67-102.

Ross J, Marshak A. 1991. Monte Carlo methods. *In:* Myneni, R.B. and Ross, J. eds. *Photonvegetation interactions: applications in optical remote sensing and plant ecology*. Springer, Berlin, 443-467.

Rouane P., 1977. Un modèle de la ramification et de la croissance végétale en tant qu'image de la différenciation cellulaire. /*Comptes Rendus de l'Académie des Sciences*, Paris/, 258 : 657-660.

Shinozaki, K., Yoda, K., Hozumi, K. and Kira, T., 1964. A quantitative analysis of plant form. Pipe model theory. I. Basic analysis. *Jpn. J. Ecol.* 14. 97–105.

Soler C, Sillion F, Blaise F, de Reffye P. 2003. An efficient instantiation algorithm for simulating radiant energy transfer in plant models. *ACM Transactions On Graphics* 22 (2): 204-233.

Spencer GH, Murty MVRK. (1962). General ray tracing Procedure. *J. Opt. Soc. Am.* 52 (6): 672–678.

Weberling S. 1989. Morphology of flowers and inflorescences. *Cambridge University Press*

Wernecke P, Buck-Sorlin G, Diepenbrock W. 2000. Combining process- with architectural models: the simulation tool VICA. *Systems Analysis Modelling Simulation,* 39 (2), 235-277.

Zhao X, de Reffye P, Xiong FL. 2001. Dual-scale automaton model for virtual plant development. *Journal of Computer*, 24(6): 524-529

Yan HP, Kang MZ, de Reffye P, Dingkuhn M. 2004. A dynamic, architectural plant model simulating resource-dependent growth. *Annals of Botany* 93(5): 591-602.

Zanella P, Ligier Yves. 1999. Architecture et technologie des ordinateurs. 3ème édition, Dunod, Paris. 495 pages.

Zeigler B.P., Moon Y., Kim J.G. 1996. DEVS-C++: A hight performance modelling and simulation environment, *29th Hawaï International Conference on System Science.*

Zucker J.D. 1991. HDR de l'Université Pierre et Marie Curie. Changement de representation, abstraction et apprentissage. 135 pages.

8 Glossaire (On notera les homonymies)

A l'usage des non-botanistes

Architecture : méthode botanique permettant l'identification des espèces végétales en s'appuyant sur la description de l'organisation de leurs axes.

Bourgeon : excroissance donnant naissance aux organes de la plante. Il comprend au minimum un méristème primaire.

Classe : troisième niveau de la classification classique (c'est-à-dire n'utilisant pas la notion de distance génétique) des espèces vivantes.

Développement : phase de création de nouveaux organes.

Développement immédiat : axe dont le développement démarre immédiatement après la mise en place de son méristème édificateur.

D**éveloppement retardé** : axe dont le développement démarre après une pause suivant la mise en place de son méristème édificateur. L'hiver peut constituer le temps de pause chez les plantes poussant en zone tempérée.

Hybride : croisement de deux individus de deux variétés, sous-espèces (croisement intraspécifique), espèces (croisement interspécifique) ou genres (croisement intergénérique) différents.

Méristème : amas cellulaire indifférencié formant une zone de croissance où ont lieu les divisions cellulaires. On distingue les méristèmes primaires qui assurent le développement de la plante, et les méristèmes secondaires responsables de la croissance en diamètre des organes de certaines plantes.

Néoformation : débourrement spontané d'un nouvel entrenœud sans stockage préalable.

Polycyclisme : qualificatif d'un fonctionnement ou le méristème exprime plusieurs pousses au cours d'une année.

Pousse : expression d'un méristème entre deux pauses d'accroissement.

Pousse annuelle : réalisation d'un méristème au cours d'une année.

Préformation : débourrement simultané d'un stock d'entre-nœuds préparés par le méristème et stockés dans son bourgeon.

Réitération : réplication par un méristème axillaire de la production du méristème apical.

Synchrone : mode de fonctionnement où tous les méristèmes d'une plante observe des périodes de pauses et d'activités simultanées.

A l'usage des non-informaticiens

Architecture : organisation matérielle ou bien logicielle d'un ordinateur. Agencement et interaction de composants logiciels.

Attribut : entité qui définit les propriétés d'une classe.

Classe : une classe déclare des propriétés communes à un ensemble d'objets. La classe déclare des attributs représentant l'état des objets et des méthodes représentant leur comportement.

Développement : action d'écrire un programme dans un langage propre à être éventuellement compilé puis exécuté par le ou les processeurs d'un ordinateur.

Héritage, polymorphisme : transmission des attributs d'une (héritage) ou plusieurs (polymorphisme) classe(s) « mère » à une classe « fille ».

Hybride : se dit d'un modèle combinant plusieurs modes de fonctionnement différents (continu et discret par exemple).

Instance : anglicisme désignant la réalisation d'une classe (objet !).

Méthode : fonction faisant partie de l'interface d'une classe.

Objet : instanciation d'une classe

Paradigme : représentation cohérente d'un objet réel qui repose sur une base définie.

Plateforme :

Phyllotaxie : angle de spirale formé par les branches le long de leur porteur.

Synchrone : un système informatique fonctionne de manière synchrone quand tous ses composants sont réévalués simultanément à pas de temps réguliers.

9 Annexe

Generic Tree Data Structure User's guide

Introduction

What is the Generic Tree Data Structure (GTDS) ?

The GTDS contains two parts :

- A file format specification, that can be used to store any kind of structured data, especially (but not only) data that will be used to represent plants.
- A C++ library that can be used to parse such files and navigate through the internal data representations.

What is the purpose of GTDS ?

By now, many different applications describe plants. All of these applications use their own file formats to store the plant description on disk. What we want is a file format that could be used by all of them. Since each application use its own informations about the plants, our format should be able to contain any kind of informations about it, in a way that any application could retrieve the informations it wants among the totality of it.

The solution for that is to use meta-informations, which means that into the file, not only do we store the data, but also the meaning of it. This concept is very simple indeed : if for example we want to store the size of a branch (of value 10), we can do it this way in a file :

```
10
```

Then, when we read this, we're supposed to know that we are reading a branch size. Now with meta-informations it looks like that :

```
<branch><size>10</size></branch>
```

Here is it specified that the value 10 corresponds to the attribute “size” of an object of type branch (then we still have to suppose that the reader knows what is called “branch” and “size”). In this example, we used xml style, we'll see later that GTDS also uses xml style.

With a file containing meta-informations, the applications just have to filter the data to get just the ones that corresponds to the informations they want.

Who can use GTDS ?

Every one that wants his application to work with data that may be partially used by another plant application. Then it is up to every one to see if he needs it or not. A good solution for an application is to use its own file format, and to have the ability to export and import into gtds format.

GTDS file format description

The green lab data structure file format is compliant with the xml specification. That means that gtds files are xml files. We won't enter into a complete description of xml (which can be found at http://www.w3.org). As a beginning, we can just think of an xml file as a tag structured document. For example :

```
<?xml encoding="UTF-8">
<data>
  <plant id="plant1">
    <branch>
      <size>10</size>
    </branch>
    <branch>
      <size>13</size>
    </branch>
  </plant>
</data>
```

Represents a plant named “plant1” containing two branches containing sizes containing values (10 and 13).

Root node name

The root element of a gtds file must be called “gtds”. That means that every document must have this structure :

```
<?xml encoding="UTF-8">
<gtds>
        /// the elements ///
</gtds>
```

Elements

Each element of a gtds file can contain any other element, of any name.

Vector

A vector of numbers can be represented by a list of numbers separated by blank spaces, for example :

```
<my_vect> 10 20 30 </my_vect>
```

Matrix

A Matrix of numbers can be represented by a list of numbers separated by blank spaces and carriage returns, for example :

```
<my_matrix> 10 20 30
            40 50 60
            70 80 100
</my_matrix>
```

Element id

Every element can contain an “id” attribute, that is a unique identifier of the element. Example of a gtds file containing elements with ids

```
<?xml encoding="UTF-8">
<gtds>
  <branch id="branch1">
     <size>10</size>
  </branch>
  <branch id="branch2">
    <size>20</size>
  </branch>
</gtds>
```

Base element

Each element can inherit properties from an other element, called the base element, to specify that an element has a base element, we can use the attributes “baseid” or “basexpath”. With “baseid” we give the identifier of the base element. With “basexpath”, we give the xpath of the base element. Example :

```
<?xml encoding="UTF-8">
<gtds>
  <branch id="branch1">
     <size>10</size>
  </branch>

  <branch>
    <size>20</size>
  </branch>

  <plant>
    <branches>
      <branch baseid="branch1" />
      <!-- this branch has a size of 10 -->
      <branch basexpath="//branch[2]" />
      <!-- this branch has a size of 20 -->
    </branches>
  </plant>
</gtds>
```

When this file will be parsed, it will be as if the plant element had been written like that :

```
<plant>
  <branches>

    <branch>
     <size>10</size>
    </branch>

    <branch>
      <size>20</size>
    </branch>

  </branches>
</plant>
```

Substructure tree representation

This format is an extension of gtds that includes substructure instantiation feature. Every previous features are kept. A special tag put at the beginning of a file will show that the plant may be built out of substructure instances.

The word substructure means “part of a tree that may be instantiated at different positions across it”.

This format mainly deals with topology and geometry but it may be improved with other data depending on the need of final user's and the capability of simulator to provide it.

new tags

topo_substructure

Tag to show that there will be a substructure description of the tree.

sstruct

Tag to encapsulate a substructure description. A substructure is a set of element that describes the geometry of the main branch. Each element may bear other substructures.

It has an attribute 'ref' associated to a unique index within the file. This will be the reference index to this substructure.

For instance :

```
<sstruct ref=”3”> ... </sstruct>
```

clone

Tag to encapsulate the instantiation of a substructure. It is slightly different from *baseid* tag since it is a real clone with no own name.

It has an attribute that contains the reference index to the target substructure.

For instance :

```
            <clone refto=”3”> ... </clone>
```

Every instance is associated to a transformation matrix relative to the bearer element transformation.

For instance :

```
      <clone refto="2">
            <mat>0 0 1 0.04
                  0 1 0 0
                  1 0 0 2 </mat>
      </clone>
```

topo_xxx

Tag to encapsulate a branch element topological description inside a substructure. Elements are described along the main branch of substructure from insertion position to end position.

It has one attribute 'index' associated to the position of this element inside its mother container.

Xxx stands for the topological component class to which this element belongs.

Topology is the main skeleton of a plant description. It consists in a collection of axes that may split in subparts. The borne axes of a parent axe are positioned on the corresponding subpart at the lowest decomposition level.

geom_xxx

Tag to encapsulate a branch element geometrical description inside a substructure. Elements are described along the main branch of substructure from insertion position to end position.

It has two attributes 'id' associated to a unique index for that element inside the plant and 'pa' for elements physiological age.

Xxx stands for the geometrical component class to which this element belongs.

For instance :

```
            <geom_elm id=”8” pa=”2”> ... </geom_elm>

      Each element will contain its geometrical description :
            <nsb> : symbol number for that element.
            <dtl> : detail level to which that element belongs.
            <nat> : element's kind (branch section or organ)
            <prt> : position of that element within the list of
branch elements.
            <mat> : transformation matrix relative to the current
substructure reference transformation (see at the end for matrix
description).
            <dUp> : upper diameter.
            <dDwn> : bottom diameter.
```

For each substructure that are borne by this element, a reference will be given :

```
            <clone refto=”xxx”> : reference of the substructure to
be pasted to this element
```

(see below).

For instance, an element geometrical descritption may look like :

```
      <geom_elm id="0" pa="0">
```

```
            <nsb>1</nsb>
            <dtl>1</dtl>
            <nat>0</nat>
            <prt>0</prt>
            <mat> 0 0 1 0
                  0 1 0 0
                  10 0 0 0 </mat>
            <dUp>0.1</dUp>
            <dDwn>0.1</dDwn>
            <clone refto="1">
                  <mat>0 0 1 -0.04
                        0 1 0 0
                        1 0 0 1 </mat>
            </clone>
            <clone refto="2">
                  <mat>0 0 1 0.04
                        0 1 0 0
                        1 0 0 2 </mat>
            </clone>
      </geom_elm>
```

Vectors , matrices and geometrical coordinates.

The assumption for 3D geometrical coordinates will be :

- x : right viewing direction
- y : rear viewing direction
- z : top viewing direction

A vector is a 3D coordinates set.
A matrix is a 4D vectors set. First line of the matrix deals with x coordinates of the vectors, second deals with y and third with z. the three first vectors describe a 3D rotation, the last vector describe a translation. The rotation vector's norm describe the scale along transformation axis. First vector gives x transformed direction inside current transformed coordinates, second vector gives y, third vector gives z.

Example of a simple plant

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <gtds>
  <headerlig>noHeader</headerlig>
- <Plant>
- <topo_axe>
- <topo_GU index="0">
  <topo_entn index="0" />
  <topo_entn index="1" />
  <topo_entn index="2" />
- <topo_entn index="3">
      - <topo_axe>
      - <topo_GU index="0">
      - <topo_entn index="0"/>
      - <topo_entn index="1"/>
      - <topo_entn index="2"/>
      - <topo_entn index="3">
            - <topo_axe>
            - <topo_entn index="0">
            - <geom_gelm id="3" pa="79">
             <nsb>2</nsb>
```

```
            <dtl>1</dtl>
             <nat>-1</nat>
             <prt>0</prt>
             <mat>-0.707052 -6.25412e-006 0.707091 2.8407 -6.25439e-
006  1  2.58994e-006  1.02588e-005  0.707162  2.5912e-006  0.707123
6.85646</mat>
             <dUp>1</dUp>
             <dDwn>1</dDwn>
             </geom_gelm>
       </topo_entn>
      </topo_axe>
      <geom_gelm id="17" pa="0">
      <nsb>1</nsb>
      <dtl>1</dtl>
      <nat>0</nat>
      <prt>1</prt>
      <mat>-1.74533e-009 0 1 0 0 1 0 0 10 0 1.74533e-010 0</mat>
      <dUp>0.1</dUp>
      <dDwn>0.1</dDwn>
      </geom_gelm>
  </topo_entn>
  </topo_GU>
  </topo_axe>
  </Plant>
  </gtds>
```

Example of a substructured plant

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
<headerlig>noHeader</headerlig>
<topo_substructure>
      <sstruct ref="0">
            <topo_entn index="0">
                  <clone refto="1">
                        <mat> 0 0 1 -0.04
                              0 1 0 0
                              1 0 0 1 </mat>
                  </clone>
                  <clone refto="1">
                        <mat> 0 -8.74228e-008 -1 0.04
                              0 -1 8.74228e-008 3.49671e-009
                              1 0 0 1 </mat>
                  </clone>
            </topo_entn>
            <topo_entn index="1">
                  <clone refto="3">
                        <mat> 0 0 1 0.0401
                              0 1 0 5.11806e-008
                              1 0 0 2 </mat>
                  </clone>
                  <clone refto="3">
                        <mat> 0 3.25841e-007 -1 -0.0399
                              0 -1 -3.25841e-007 -5.72205e-008
                              1 0 0 2 </mat>
                  </clone>
```

```
            </topo_entn>
            <topo_entn index="2">
                  <geom_elm id="0" pa="0">
                        <nsb>1</nsb>
                        <dtl>1</dtl>
                        <nat>0</nat>
                        <prt>0</prt>
                        <mat> -5.23599e-010 0 1 0
                              0 1 0 0
                              3 0 1.74533e-010 0 </mat>
                        <dUp>0.1</dUp>
                        <dDwn>0.1</dDwn>
                  </geom_elm>
            </topo_entn>
      </sstruct>
      <sstruct ref="1">
            <topo_entn index="0"/>
            <topo_entn index="1">
                  <geom_elm id="1" pa="1">
                        <nsb>1</nsb>
                        <dtl>1</dtl>
                        <nat>1</nat>
                        <prt>0</prt>
                        <mat> -1.73205 0 0.5 -0.04
                              0 1 0 0
                              0.999999 0 0.866026 1 </mat>
                        <dUp>0.1</dUp>
                        <dDwn>0.1</dDwn>
                  </geom_elm>
            </topo_entn>
      </sstruct>
      <sstruct ref="3">
            <topo_entn index="0">
                  <geom_elm id="2" pa="2">
                        <nsb>1</nsb>
                        <dtl>1</dtl>
                        <nat>3</nat>
                        <prt>0</prt>
                        <mat> 0.866076 1.27952e-006 -0.499826 0.0401
                              1.10809e-006 -1 -6.39647e-007 5.11806e-
008
                              0.499913   -1.27952e-010   0.866126   2
</mat>
                        <dUp>0.1</dUp>
                        <dDwn>0.1</dDwn>
                  </geom_elm>
            </topo_entn>
      </sstruct>
</topo_substructure>
</gtds>
```

Tutorial

Creating a new empty document

In libgtds, all the data are stored into what is called documents. A document can be seen as a tree representation of the data.

Here is the code of a very simple -and useless- program that creates a new document, and then save it into a file :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;
int main()
{
  cout << "creating a new empty gtds document" << endl;
  Document doc;
  cout << "saving it into \"doc.gtds\"" << endl;
  doc.writeToFile("doc.gtds");
  return 0;
}
```

If we run this program, then the following file doc.gtds will be created :

```
<?xml version="1.0" encoding="UTF-8"?>
```

This file contains no data, just a header.

Adding element data to the document

As we said before, a gtds document can be seen as a tree representation of the data. From the document you can access the root node of the data, with the method “getRootNode”, that returns an object of type Node. When you create a new document, it has no root node at the begining, you have to create it with the method “createRootNode”. Since every node is identified by a name, you have to give the name of the root node as a parameter of the method. In our last example, we could have added this line :

```
Node rootNode = doc.createRootNode("gtds");
```

Then the file doc.gtds would have been :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds></gtds>
```

Now, if we want to add a child node to our root node, we can use the Node method “addChildren”, just like this :

```
Node child = rootNode.addChild("plant");
```

Then the file doc.gtds became :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
  <plant></plant>
</gtds>
```

Here is a full example of a program that create a new document with a root node called “root” and several children elements :

```
#include <iostream>
#include <libgtds/libgtds>

using namespace std;
using namespace gtds;

int main()
{
  // creating the document
  Document doc;
  // creating the root node, called "gtds"
  Node rootNode = doc.createRootNode("gtds");
  // adding a sub element node called "plant"
  Node plant1Node = rootNode.addChild("plant");
  // adding a sub element to this one, called "internode"
  plant1Node.addChild("internode");
  // adding a second element node to the root node, also called plant
  rootNode.addChild("plant");

  // writing the document to a file
  doc.saveToFile("doc.gtds");

  return 0;
}
```

adding value to elements

Text node

We now know how to create a document and add elements to it. A node can contain other nodes (that's what we did with the method addChild), these kind of nodes are called element. Now they are other kinds of nodes that contain text or numerical values. Let's see a first example with a text node :

```
#include <iostream>
#include <libgtds/libgtds.h>

using namespace std;
using namespace gtds;

int main()
{
  // creating the document
  Document doc;
  // creating the root node, called "gtds"
  Node rootNode = doc.createRootNode("gtds");
  // adding a sub text node called "text"
  Text textNode = rootNode.addChild("text");
  // set the value of the text node
  textNode.setValue("hello");

  // writing the document to a file
  doc.writeToFile("doc.gtds");

  return 0;
}
```

The code requires a little explanation, the class “gtds::Text” (here we just wrote “Text” for we used the namespace gtds) herits from the class Node. So a Text is a special Node that can have a text value.

When we call rootNode.addChild(“text”), we get a Node, but since the object textNode is of type Text, a conversion is made and then we get a Text.

In the following line we assign the value to the Text Node, we could also have written :

```
textNode = "hello"
```

which is a convenient way to assign a value to a gtds::Text.

The doc.gtds then looks like that :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
  <text>hello</text>
</gtds>
```

Number Node

To assign a number to a node, you can do the same way. But instead of using a gtds::Text class you use a gtds::Number<T> class, where T can be any numerical type (int, float, double, etc.) In the last example, we can had the following line :

```
Number<int> intNode = rootNode.addChild("number");
intNode.setValue(10);
// intNode = 10 works as well
```

Vector Node

Gtds offers a convenient way to store vectors of numbers. To do so you need to use the class gtds::Vector<T>, where T can be any numerical type. Here is an example of the use of the class Vector<T> :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // creating the document
  Document doc;
  // creating the root node, called "gtds"
  Node rootNode = doc.createRootNode("gtds");
  // adding a sub vector node called "vect"
  Vector<int> vectorNode = rootNode.addChild("vect");

  // we can fill the vectorNode from a std::vector
  vector<int> vect(3);
  vect[0] = 1;
  vect[1] = 2;
  vect[2] = 3;
  vectorNode.setValue(vect);

  // we can also fill the vectorNode from a pointor to an array of numbers :

  // adding another sub vector node called "vect2"
  Vector<int> vectorNode2 = rootNode.addChild("vect2");
  int array[4] = {10, 20, 30, 40};
  vectorNode2.setValue(&array[0], 4);

  // writing the document to a file
  doc.writeToFile("doc.gtds");
  return 0;
}
```

Then the file doc.gtds will looks like that :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
  <vect>1 2 3 </vect>
  <vect2>10 20 30 40 </vect2>
</gtds>
```

Matrix Node

In the same way that for the vector, you can store matrix of number :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // creating the document
  Document doc;
  // creating the root node, called "gtds"
  Node rootNode = doc.createRootNode("gtds");

  // adding a sub matrix node called "mat"
  Matrix<int> matrixNode = rootNode.addChild("mat");

  // we can fill the matrixNode from a std::vector<std::vector>
  // that is a vector of vectors
  vector<vector<int> > mat(2);
  mat[0].resize(2);
  mat[0][0] = 1;
  mat[0][1] = 2;
  mat[1].resize(2);
  mat[1][0] = 3;
  mat[1][1] = 4;
  matrixNode.setValue(mat);

  // we can also fill the matrixNode from a pointor
  // adding another sub vector node called "vect2"
  Matrix<int> matrixNode2 = rootNode.addChild("mat2");
  int array[2][2] = {{10,20},{30,40}};
  matrixNode2.setValue(&array[0][0], 2, 2);

  // writing the document to a file
  doc.writeToFile("doc.gtds");
  return 0;
}
```

With this code the file doc.gtds looks like that :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
  <mat>1 2
3 4
</mat>
  <mat2>10 20
30 40
</mat2>
</gtds>
```

Reading a gtds document

In this section we will see how to read a document previously saved into a file with the gtds::Document writeToFile method.

First of all you must parse the file in order to get a valid Document. To do so we use the class gtds::Parser. Example of parsing a gtds file :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");

  // now we have a new document
  Document* doc = parser.getDocument();

  return 0;
}
```

This is all you need to do in order to read a new document, you can then use it as if it has been created from scratch, except that you have a pointer to a document instead of a real instance of a document.

Acceding the data

To access the data you first have to get the root node of the document, with the method getRootNode. Then there are several methods of the class Node that permit you to find the data you want. These methods are : getChild, getChildren, find, getFirstDescendants.

getChild

The Node::getChild method returns the first child Node of the node with the specified name.

Example : we want to read a gtds document that looks like that :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
  <plant>
    <size>10</size>
  </plant>
</gtds>
```

When parsed, the document will contain a root node called gtds, containing a node called plant, containing a number node called size. Here is a code that get the size of the plant :

```
#include <iostream>
#include <libgtds/libgtds.h>

using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");
  // now we have a new document
  Document* doc = parser.getDocument();

  // get the root node
  Node rootNode = doc->getRootNode();
  // get the plant node
  Node plantNode = rootNode.getChild("plant");
  // get the size node
  Number<int> sizeNode = plantNode.getChild("size");

  // print the size
  cout << "size: " << sizeNode << endl;

  return 0;
}
```

Here we could have done everything in a single line :

```
Number<int> sizeNode = doc->getRootNode().getChild("plant").getChild("size");
```

getChildren

GetChildren returns a vector of all the children nodes that have a specified name. If no name is specified then it returns all the children of the node.

Example :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");
  // now we have a new document
  Document* doc = parser.getDocument();

  // get the root node
  Node rootNode = doc->getRootNode();

  // get all the plant node
  NodeList plantNodes = rootNode.getChildren("plant");

  // print the number of plant nodes
  cout << plantNodes.size() << endl;

  if (plantNodes.size() <= 0) return 0;

  // get the first Node of the list
  Node plantNode = plantNodes.front();

  return 0;
}
```

find

"Find" is a very powerful method that permits to access data from a string in xpath format. To learn more about xpath, visit the official site http://www.w3.org/TR/xpath. The "find" method returns the list of the nodes that fit the xpath expression. Example of using :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");
  // now we have a new document
  Document* doc = parser.getDocument();
  // get the root node
  Node rootNode = doc->getRootNode();

  // we find directly the size of the first plant
  NodeList sizeNodes = rootNode.find("plant[1]/size");

  if (sizeNodes.size() != 1) return 1;

  Number<int> sizeNode = sizeNodes.front();
  cout << sizeNode << endl;

  return 0;
}
```

getFirstDecendants

This is a useful convenient method to find the first descendants of a node that match a given name. For example if you have this gtds file :

```
<?xml version="1.0" encoding="UTF-8"?>
<gtds>
 <plant>
  <branch>
    <branch> <size>20</size> </branch>
    <node>
      <branch>
       <size>10</size>
       <node>
         <branch>
         </branch>
       </node>
      </branch>
    </node>
  </branch>
  <size>10</size>
 </plant>
</gtds>
```

Here we can see that the first "branch" node contains two branches, one is a direct child, the other is owned by a "node" node. So if you want to get these two "branch" nodes in a single instruction, you have to use the getFirstDescendants method. Note that the third "branch" node won't be returned by the method, since it is owned by a branch already included in the list of descendants.

Here the code could be :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");
  // now we have a new document
  Document* doc = parser.getDocument();
  // get the root node
  Node rootNode = doc->getRootNode();
  // we get the first branch
  Node branchNode = rootNode.find("plant/branch").front();

  // now we get all the first branch descendants of the branch
  NodeList branchNodes = branchNode.getFirstDescendants("branch");

  cout << branchNodes.size() << endl;
  // it will print "2"
  return 0;
}
```

Tutorial 2

In this tutorial, we will see more advanced concepts of the gtds c++ library.

Attributes

Since gtds is based on the xml standard, it is possible to add attributes to the nodes. An attribute can be seen the same way that an child Node, except that you can not have children of attribute, nor have two attributes of a node with same names. In xml, we represent attribut into the opening tag of the element :

```
<document>
  <element fr="bonjour" ch="ni hao"> </element>
</document>
```

In this example we represented an element with two attributes called “fr” and “ch” with the value “bonjour” and “ni hao”.

In libgtds, you can add an attribute to a gtds::Node with the method setAttribute. you can get the value of an attribute with the method getAttribute. Example :

```
#include <iostream>
#include <libgtds/libgtds.h>
using namespace std;
using namespace gtds;

int main()
{
  // create the parser
  Parser parser;
  // parse the file
  parser.parseFile("doc.gtds");
  // now we have a new document
  Document* doc = parser.getDocument();
  // get the plant node
  Node plantNode = doc->getRootNode().find("//plant").front();

  // add an attribute called name with the value "plant1"
  Attribute attribute = plantNode.setAttribute("name", "plant1");

  // print the doc to the standard input
  cout << *doc << endl;

  // get the name of the plant
  Attribute name = plantNode.getAttribute("name");
  cout << "name = " << name << endl;

  return 0;
}
```

Base Node

The base node concept has been explained in the gtds file format specification. Now we will see how to set base node. There are 3 methods that can be used :

- setBaseNode, that takes the base node as argument.
- setBaseNodeId, that takes the base node id as argument.
- setBaseNodeXpath, that takes the base node xpath as argument.

Example of use :

```
#include <libgtds/libgtds.h>
#include <iostream>
#include <fstream>
#include <exception>
using namespace std;
using namespace gtds;
int main()
{
  // we create an empty document
  Document doc;
  // we add a branch
  Node root = doc.createRootNode("root");
  Node branch = root.addChild("branch");
  // we set the branch id
  branch.setId("br");
  branch.setAttribute("size", 10);

  // now a second branch that herits from the first
  Node branch2 = root.addChild("branch");
  // we specify the base node by its id
  branch2.setBaseNodeId("br");

  // and a third one that herits from the second
  Node branch3 = root.addChild("branch");
  // we give directly the base node
  branch3.setBaseNode(branch2);

  // we print the doc
  cout << doc << endl << endl;

  // now to see if it works, we try to get the size attribute
  // from the branches 2 and 3
  Attribute att = branch2.getAttribute("size");
  if (!att) return -1;
  cout << att << endl;

  att = branch3.getAttribute("size");
  if (!att) return -1;
  cout << att << endl;

  return 0;
}
```

The output of this program is, as expected :

```
hello

<?xml version="1.0" encoding="UTF-8"?>
<root>
  <branch id="br" size="10"/>
  <branch baseid="br"/>
  <branch basexpath="/root/branch[2]"/>
</root>

10
10
```

Compressed file

You have the possibility to save your gtds files, not directly as xml, but in a binary format (which is actually just the xml file zipped). To do so you just need to use the methods Parser::parseFileCompressed and Document::writeToFileCompressed insted of Parser::parseFile and Document::writeToFileCompressed.

Example :

```
#include <libgtds/libgtds.h>
#include <iostream>
#include <fstream>
#include <exception>
using namespace std;
using namespace gtds;
int main()
{
  try {
    cout << "test i/o compressed gtds files" << endl << endl;

    Parser parser;
    cout << "reading plant.xml" << endl;
    parser.parseFile("plant.xml");
    Document* doc = parser.getDocument();

    cout << "writing plant.xml.gz" << endl;
    doc->writeToFileCompressed("plant.xml.gz");
    cout << "reading plant.xml.gz" << endl;
    parser.parseFileCompressed("plant.xml.gz");
    cout << "output :" << endl << *(parser.getDocument());

  }
  catch (const exception& exc) {
    cerr << "exception : " << exc.what() << endl;
    return -1;
  }
  return 0;
}
```

TsltXML2LIG C++ class , XML to linetree format translator

Interface

```
#ifndef TSLTXML2LIG_H
#define TSLTXML2LIG_H
#include <string>
#include <iostream>
using namespace std;

#include "Matrix.h"

#include "bfstream.h"
using namespace Vitis;

#include "libgtds.h"
using namespace gtds;

class TreeElemRecord
{
        int l_nsb;
        int l_detail;
        int l_age;
        int l_prt;
        float l_haut;
        float l_bas;
        long l_index;
        int l_phyage;
        Matrix4 transformation;
        TreeElemRecord * bearer;
        std::vector<TreeElemRecord *> children;

       public:

              TreeElemRecord() ;
              ~TreeElemRecord();

              //reading Accessor
              inline int getSymbole () const {return l_nsb;}
              inline int getDetail ()const {return l_detail;}
              inline int getPosition ()const {return l_prt;}
```

```
                inline int getAge ()const {return l_age;}
                inline int getPhyAge () const {return l_phyage;}
                inline float getDiamUp () const{return l_haut;}
                inline float getDiamDown ()const {return l_bas;}
                inline long getIndex ()const {return l_index;}
                inline int getChildSize ()const {return children.size();}
                inline Matrix4 & getTransformation () {return transformation;}
                inline const Matrix4 & getTransformation () const {return transformation;}
                inline const TreeElemRecord * getBearer () const {return bearer;}
                inline std::vector<TreeElemRecord *> & getChildren () {return children;}
                inline  const  std::vector<TreeElemRecord  *>  &  getChildren  ()  const  {return
children;}
                inline const TreeElemRecord * getChildrenAt (int i) const {return children[i];}
                inline TreeElemRecord * getChildrenAt (int i) {return children[i];}

                //Writing Accessor
                inline void setSymbole (int s) {l_nsb=s;}
                inline void setDetail (int d) {l_detail=d;}
                inline void setPosition (int p) {l_prt=p;}
                inline void setAge (int a) {l_age=a;}
                inline void setPhyAge (int a) {l_phyage=a;}
                inline void setDiamUp (float d) {l_haut=d;}
                inline void setDiamDown (float d) {l_bas=d;}
                inline void setIndex (long i) {l_index=i;}
                inline void setBearer (TreeElemRecord *e) {bearer=e;}
                inline void setTransformation (const Matrix4 & m) {transformation=m;}
                inline void addChildren (TreeElemRecord * e) {children.push_back(e);}
                void removeChildren(int i);
};
```

```
#ifndef TSLTXML2LIG_H
#define TSLTXML2LIG_H
#include <string>
#include <iostream>
using namespace std;

#include "Matrix.h"

#include "bfstream.h"
using namespace Vitis;

#include "libgtds.h"
using namespace gtds;

class TreeElemRecord
{
        int l_nsb;
        int l_detail;
        int l_age;
        int l_prt;
        float l_haut;
        float l_bas;
        long l_index;
        int l_phyage;
        Matrix4 transformation;
        TreeElemRecord * bearer;
        std::vector<TreeElemRecord *> children;

       public:

               TreeElemRecord() ;
               ~TreeElemRecord();

               //reading Accessor
               inline int getSymbole () const {return l_nsb;}
               inline int getDetail ()const {return l_detail;}
               inline int getPosition ()const {return l_prt;}
               inline int getAge ()const {return l_age;}
               inline int getPhyAge () const {return l_phyage;}
               inline float getDiamUp () const{return l_haut;}
               inline float getDiamDown ()const {return l_bas;}
               inline long getIndex ()const {return l_index;}
               inline int getChildSize ()const {return children.size();}
               inline Matrix4 & getTransformation () {return transformation;}
               inline const Matrix4 & getTransformation () const {return transformation;}
               inline const TreeElemRecord * getBearer () const {return bearer;}
```

```
                inline std::vector<TreeElemRecord *> & getChildren () {return children;}
                inline  const  std::vector<TreeElemRecord  *>  &  getChildren  ()  const  {return
children;}
                inline const TreeElemRecord * getChildrenAt (int i) const {return children[i];}
                inline TreeElemRecord * getChildrenAt (int i) {return children[i];}

                //Writing Accessor
                inline void setSymbole (int s) {l_nsb=s;}
                inline void setDetail (int d) {l_detail=d;}
                inline void setPosition (int p) {l_prt=p;}
                inline void setAge (int a) {l_age=a;}
                inline void setPhyAge (int a) {l_phyage=a;}
                inline void setDiamUp (float d) {l_haut=d;}
                inline void setDiamDown (float d) {l_bas=d;}
                inline void setIndex (long i) {l_index=i;}
                inline void setBearer (TreeElemRecord *e) {bearer=e;}
                inline void setTransformation (const Matrix4 & m) {transformation=m;}
                inline void addChildren (TreeElemRecord * e) {children.push_back(e);}
                void removeChildren(int i);
};
```

```
class TranslatorLIGXML
{
        std::string filename;
        beofstream __ligstream;
        beofstream __arcstream;
        Parser parser;
        TreeElemRecord * rootNode;
        int indexCpt;
        NodeList ligs;

public:
        TranslatorLIGXML(const std::string & filename);
        ~TranslatorLIGXML(void);
        void writeTree ();
        void writeTreeFrom (const TreeElemRecord * trec);
        void writeHeader ();
        void writeRecord (const TreeElemRecord * trec);
        int buildTree();
        void readTreeNode(const Node & rootnod, TreeElemRecord * t, Matrix4 * transf);
        Node findCloneElem(Node & n);
        inline TreeElemRecord * getRootNode () {return rootNode;}
};


#endif
```

Implementation

```
#include "SimulationManager.h"
#include "tsltxml2lig.h"
#include "VitisObject.h"

TreeElemRecord::TreeElemRecord()
{
         l_nsb=l_detail=l_age=l_phyage=l_prt=0;
         l_haut=l_bas=0;
         l_index=0;
         bearer=NULL;
}

TreeElemRecord::~TreeElemRecord()
{
        if(!children.empty())
        {
                std::for_each(children.begin(), children.end(), Delete());
                children.clear();
        }
}

void TreeElemRecord::removeChildren(int i)
```

```
{
	children.erase(children.begin()+i);
}


TranslatorLIGXML::TranslatorLIGXML(const std::string & filename):__ligstream
(filename+".lig"),__arcstream(filename+".arc")
{
	this->filename=filename+".xml";
	rootNode=NULL;
	indexCpt=0;
}

TranslatorLIGXML::~TranslatorLIGXML(void)
{
}

void TranslatorLIGXML::writeTree ()
{
	writeTreeFrom(rootNode);
}

void TranslatorLIGXML::writeTreeFrom (const TreeElemRecord * trec)
{
	writeRecord(trec);
	for(int i=0; i<trec->getChildSize(); i++)
	{
		writeTreeFrom(trec->getChildrenAt(i));
	}
}


void TranslatorLIGXML::writeHeader ()
{
	gtds::Text headerLig = parser.getDocument()->getRootNode().find("/gtds/headerlig").at(0);
	__ligstream<<headerLig.getValue();
	__arcstream<<indexCpt;
}

void TranslatorLIGXML::writeRecord (const TreeElemRecord * trec)
{
		Vector3 dp, dssb ,ds, t;

		dssb=trec->getTransformation().getNormalVector();
		ds=trec->getTransformation().getSecondaryVector();
		dp=trec->getTransformation().getMainVector();
		t=trec->getTransformation().getTranslation();

		//LIG RECORD
		__ligstream << long(trec->getSymbole()) << long(trec->getDetail());
		__ligstream << long(trec->getAge()) <<long(trec->getPosition());
		__ligstream << (float) dp[0]<< (float) dssb[0]<< (float) ds[0]<< (float) t[0];
		__ligstream << (float) dp[1]<< (float) dssb[1]<< (float) ds[1]<< (float) t[1];
		__ligstream << (float) dp[2]<< (float) dssb[2]<< (float) ds[2]<< (float) t[2];
		__ligstream << float(trec->getDiamDown())<< float(trec->getDiamUp());
		__ligstream <<int(trec->getIndex());

		//on recupère tous les elements qui suivent
		__arcstream << int(trec->getIndex()) << int(trec->getBearer()->getIndex());
		__arcstream << short(trec->getChildSize());
		__arcstream << float(t[0]) << float(t[1]) << float(t[2]);
		__arcstream << (float)trec->getPosition()<<float(trec->getDiamDown());
		__arcstream << float(trec->getDiamUp())<<short(trec->getSymbole());
		__arcstream << int(trec->getAge())<<long(trec->getPosition());

		//on ajoute les id des premiers elements des noeuds fils "axe"
		for(int i=0; i<trec->getChildSize(); i++)
		{
			__arcstream<<trec->getChildrenAt(i)->getIndex();
		}
}


int TranslatorLIGXML::buildTree()
{
	// we open and read the xml file
	parser.parseFile(filename);
```

```
	if (!parser.getDocument()) {
				perror(filename.c_str());
				return 0;
	}

	try
	{
		ligs = parser.getDocument()->getRootNode().find("/gtds/Plant/axe");

		if(ligs.empty())
		{  cerr<<" error : I can't find root !"<<endl;
			//try substruct
			ligs = parser.getDocument()-
>getRootNode().find("/gtds/substructure/sstruct");
			if(ligs.empty())
				return -1;
		}
		readTreeNode(ligs[0],NULL,NULL);
	}
	catch (const exception& exc) {
	 cerr << "error : " << exc.what() << endl;
	 return -1;
	}

	return 1;
}


void TranslatorLIGXML::readTreeNode (const Node & nodaxe,TreeElemRecord * trec, Matrix4 * transf)
{
	  float mat[3][4];
	  int l_nsb,l_detail,l_age,l_prt;
	  float l_haut, l_bas;
	  long l_index, l_indexb;
	  TreeElemRecord * current;
	  TreeElemRecord * bearer = trec;
	  Matrix4 currentTrans;
	  Matrix4  cloneTrans;
	  Vector3 main (0,0,0);
	  NodeList nodLst=nodaxe.getChildren("elm");

	  //Pour chaque noeud elem de l'axe courant
	  for(int numNod=0; numNod<nodLst.size(); numNod++)
	  {
		//On recupere données du noeud xml
		const gtds::Node & nod=nodLst[numNod];
		NodeList nodElem=nod.getChildren();
		gtds::Matrix<float> gtdsMat = nodElem[4];
		gtdsMat.fill(&mat[0][0]);
		currentTrans.setMainVector(mat[0][0],mat[1][0] ,mat[2][0]);
		currentTrans.setNormalVector(mat[0][1],mat[1][1],mat[2][1]);
		currentTrans.setSecondaryVector(mat[0][2],mat[1][2],mat[2][2]);
		currentTrans.setTranslation(mat[0][3],mat[1][3],mat[2][3]);

		if(transf)
		{
			Vector3 translat=currentTrans.getTranslation();
			//warning : on deplace ac main
			if(numNod==0)
				currentTrans= (*transf)*Matrix4::translation(-
translat)*currentTrans;
			else
				currentTrans=
Matrix4::translation(main)*(*transf)*Matrix4::translation(-translat)*currentTrans;
		}

		main+=currentTrans.getMainVector();
		l_nsb =gtds::Number<int>(nodElem[0]);
		l_detail = gtds::Number<int>(nodElem[1]);
		l_age = gtds::Number<int>(nodElem[2]);
		l_prt = gtds::Number<int>(nodElem[3]);
		l_haut = gtds::Number<float> (nodElem[5]);
		l_bas = gtds::Number<float> (nodElem[6]);

		 //on crée et init un noeud mémoire
		current=new TreeElemRecord();
		if(bearer != NULL)
```

```
		{
			bearer->addChildren(current);
			current->setBearer(bearer);
		}
		else
		{	rootNode=current;
			current->setBearer(current);
		}
		current->setAge(l_age);
		current->setDetail(l_detail);
		current->setDiamDown(l_bas);
		current->setDiamUp(l_haut);
		current->setPosition(l_prt);
		current->setSymbole(l_nsb);
		current->setIndex(indexCpt);
		current->setTransformation(currentTrans);
		current->setPhyAge(atoi(((Attribute)nod.getAttribute("pa")).getValue().c_str()));

		indexCpt++;

		//on lance la tradcution pour chaque axe porté
		for(int i=7; i<nodElem.size(); i++)
		{
			if(nodElem[i].getName()=="clone")
			{
				Node clone=findCloneElem(nodElem[i]);
				NodeList childtmp=nodElem[i].getChildren("mat");
				gtdsMat = childtmp.at(0);
				gtdsMat.fill(&mat[0][0]);

				cloneTrans.setMainVector(mat[0][0],mat[1][0] ,mat[2][0]);
				cloneTrans.setNormalVector(mat[0][1],mat[1][1],mat[2][1]);
				cloneTrans.setSecondaryVector(mat[0][2],mat[1][2],mat[2][2]);
				cloneTrans.setTranslation(mat[0][3],mat[1][3],mat[2][3]);

				if (transf)
					cloneTrans = (*transf)*cloneTrans;

				readTreeNode(clone, current, &cloneTrans);
			}
			else
			{
				readTreeNode(nodElem[i], current, NULL);
			}
		}
		bearer=current;
	  }//end for
}


Node TranslatorLIGXML::findCloneElem(Node & n)
{
	Int ref=atoi(((Attribute)n.getAttribute("refto")).getValue().c_str());

	for (NodeList::iterator iter = ligs.begin(); iter != ligs.end(); ++iter)
	{
		if(atoi(((Attribute)iter->getAttribute("ref")).getValue().c_str())==ref)
			return *iter;
	}

	return Node();
}
```