Démonstrateur de L-Systèmes

Rapport de projet de réalité virtuelle

Réalisé par :

Rémi DUCCESCHI et Thomas NOGUER

Proposé et encadré par :

Sébastien AUPETIT

Polytech Tours 2012-2013

INTRODUCTION

Ce projet s'inscrit dans le cadre de la formation ingénieur de Polytech Tours. Il a été réalisé par deux étudiants de dernière année Rémi Ducceschi et Thomas Noguer. Il s'agit d'un projet d'option de réalité virtuelle proposé et encadré par Sébastien Aupetit. Le projet a pour but principal de mettre en place une application évolutive de démonstrateurs L-Système. L'application doit pouvoir interpréter tous les types de grammaires L-Système et proposer un affichage 3D du résultat. La nature de l'affichage peut être très variée selon les interprétations de la tortue choisies. Le projet aura donc pour but de proposer des interprétations de la tortue de base et de permettre un ajout le plus simple possible de nouvelles interprétations.

TABLE DES MATIERES

I. DESCRIPTION L-SYSTEME

Notre projet repose sur la visualisation d'interprétations de L-Systèmes. Dans cette partie, nous allons décrire ce qu'est une grammaire L-Système.

DESCRIPTION GENERALE

Les grammaires L-Systèmes sont des spécialisations des grammaires formelles. Elles ont été inventées par le biologiste hongrois Aristid Lindenmayer (et sont parfois appelées Lindenmayer systèmes). Ces grammaires ont la particularité d'utiliser un développement parallèle lors de l'application de règles sur un mot : lors de l'utilisation en production, on utilise à chaque étape l'ensemble des règles applicables sur le mot alors que dans les grammaires de Chomsky, on n'applique qu'une règle par étape. Cela permet de simuler des comportements complexes que l'on peut retrouver dans la nature, comme la division cellulaire (voir ci-dessous) ou la pousse d'arbres.

Si l'on veut simuler la division cellulaire à l'aide d'une telle grammaire, on pourrait partir d'un axiome "O" (qui représente une cellule) et utiliser une règle qui change tous les "O" en "OO" (modélisant ainsi la division d'une cellule). On obtiendrait alors :

- Étape 1 : O
- Étape 2 : OO
- Étape 3 : OOOO
- Étape 4 : OOOOOOOO
- …

Cette modélisation se comprend facilement juste en lisant la production de la grammaire. Cela n'est pas le cas de la simulation de pousse d'arbre qui nécessite une interprétation spécifique de la grammaire (appelée interprétation de la tortue) qui permet de représenter les symboles produits dans une visualisation compréhensible par l'homme. C'est le but de notre projet.

De par leurs spécificités, les grammaires L-Systèmes sont généralement utilisées en production, à l'inverse des grammaires de Chomsky qui sont plutôt utilisées en vérification. L'application des L-Systèmes dans la réalité virtuelle permet de générer des environnements réalistes et détaillés en insérant des arbres générés à l'aide d'une grammaire. L'intérêt étant ici de disposer d'une grande quantité d'arbres différents sans pour autant avoir à modéliser tous les arbres directement. Ceux-ci seront automatiquement générés.

Afin de pouvoir utiliser pleinement une grammaire L-Système, il faut pouvoir la définir. Une telle grammaire est composée de :

- Un ensemble de symboles utilisables (appelé "V") qui seront tous les symboles pouvant apparaître dans les mots produits par la grammaire. Cet ensemble est généralement coupé en deux sous-parties :
  - L'ensemble des symboles constants qui ne peuvent pas être modifiés par une règle de réécriture
  - L'ensemble des symboles modifiables qui ont au moins une règle de réécriture qui leur est associée.
- Un axiome qui comporte un unique symbole faisant partie des symboles utilisables (généralement constant)

- Une liste de règles de réécriture qui permettent de définir l'évolution des mots lors de l'application de ces règles sur l'axiome.
- Éventuellement un angle qui définit l'angle entre les branches lors de l'interprétation de la tortue.

Une règle de réécriture est composée de deux parties :

- La partie gauche contient le symbole qui sera modifié par l'application de la règle ;
- La partie droite contient l'ensemble des symboles qui viendront remplacer celui de la partie gauche.

En plus de cela, le résultat de la production d'une grammaire dépend beaucoup du nombre d'itérations que l'on souhaite appliquer. Selon le nombre d'étapes, le mot final pourra être plus ou moins long et complexe, et donc l'arbre plus ou moins détaillé.

Une règle ne peut modifier qu'un seul symbole. La partie gauche ne doit donc contenir qu'un seul symbole à modifier. Il est possible de supprimer un symbole en mettant en partie droite le symbole "ε".

Il existe plusieurs manières d'appliquer une règle à un symbole, et selon la manière, on peut modifier le type de la grammaire. Il existe deux catégories de grammaires : le déterminisme et la dépendance du contexte. Tous les L-Systèmes apparaissent dans ces deux catégories. Ainsi, une grammaire peut être déterministe et indépendante du contexte, ou déterministe et dépendante du contexte… On identifie ainsi 4 types différents de grammaires :

- DOL : déterministe et indépendante du contexte ;
- SOL : stochastique et indépendante du contexte ;
- DIL : déterministe et dépendante du contexte ;
- SIL : stochastique et dépendante du contexte.

DETERMINISME

Le déterminisme permet de créer des grammaires qui seront constantes dans la production à partir d'un axiome donné, ou au contraire qui pourront générer différents résultats avec la même configuration de base. Les grammaires stochastiques permettent ainsi de générer des arbres différents à chaque fois, sans pour autant changer de grammaires. Cela est particulièrement pratique dans un jeu vidéo par exemple, ou un grand nombre d'arbres peuvent être générés lors de la création d'un niveau.

Dans les grammaires déterministes, il ne peut y avoir au maximum qu'une seule règle de réécriture pour un symbole donné. Ainsi, chaque fois que le symbole est rencontré, il est toujours remplacé par la même séquence. Une grammaire déterministe est notée "DL-System".

Au contraire, les grammaires stochastiques peuvent avoir un nombre quelconque de réécritures pour un même symbole. Lorsque le symbole est rencontré lors de la génération, un tirage au sort est fait entre les règles applicables et la règle choisie est appliquée. Les règles éligibles peuvent avoir une pondération définie par l'utilisateur, lui donnant alors plus ou moins de chance que les autres d'être sélectionnée. C'est ainsi que pour une même grammaire (même axiome, mêmes ensembles de règles et de symboles utilisables) et le même nombre d'itérations, le résultat peut être complètement différent. Ces grammaires sont notées "SL-Systems".

Si l'on prend un ensemble de symboles `V = { a; b }` et un axiome a et qu'on y ajoute une règle changeant "a" en "b" et une autre "b" en "a", on obtient une grammaire déterministe. Si l'on rajoute la règle

transformant "a" en "ab", la grammaire devient stochastique. Il est alors possible de donner une probabilité pour les règles stochastiques (ici, celles qui réécrivent le symbole "a"). On aura alors les règles suivantes :

- `a → b : 0.2`
- `a → ab : 0.8`
- `b → a`

La première règle aura une probabilité de 0.2 alors que la deuxième sera choisie dans 80 % des cas lors de la rencontre du symbole "a". La dernière règle est déterministe : il n'existe pas d'autres règles redéfinissant le symbole "b".

DEPENDANCE DU CONTEXTE

Les grammaires indépendantes du contexte, notées "OL-Systems" (pour 0 context sensitive) sont des systèmes dont l'application d'une règle ne dépend que du caractère courant, aucunement du reste du mot à modifier. Ces grammaires sont très simples et la génération est très rapide. L'écriture de leurs règles est aussi très simple : `a → b`. Toutes les grammaires vues jusqu'à présent étaient indépendantes du contexte.

Les grammaires dépendantes du contexte (notées "IL-Systems") doivent, lors de la génération, regarder le symbole courant pour déterminer une règle à appliquer, mais aussi l'ensemble du mot pour savoir quelles sont les règles applicables. Les règles dépendantes du contexte peuvent être dépendantes à gauche, à droite ou les deux.

Prenons par exemple la règle suivante (avec le même `V` que précédemment) : `a` **`< a`** `→ b`. On retrouve ici la règle de "a" qui se transforme en "b", mais on a ajouté la partie en gras qui signifie "si un "a" a déjà été rencontré", le symbole "<" indiquant une précédence. La règle `a` **`> a`** `→ b` est équivalente sauf qu'un autre "a" doit suivre dans le mot au lieu d'être avant. Le premier symbole d'une règle est toujours le symbole à modifier. Les notions de contextes viennent après. Il est enfin possible de dire qu'un "a" ne doit se transformer en "b" uniquement si un "a" a déjà été rencontré, et qu'un autre suit : `a` **`< a > a`** `→ b`. Une telle règle sur le mot "`aaaaaaa`" aura pour résultat après une itération : "`abbbbba`". Il est important de noter que la notion de contexte s'applique au mot entier, et non juste aux symboles directement à côté du symbole étudié.

Dans les paramètres de contexte, il est possible de mettre non pas un simple symbole, mais un ensemble de symboles. La règle `a > abb → bba` signifie "a suivie de a, b et b donne bba". Attention toutefois, "abb" n'est pas ici une séquence. Si les caractères "a", "b" et "b" existent dans cet ordre après le symbole "a" courant, mais pas forcément consécutivement, la règle reste applicable.

Bien que complexes, ces grammaires permettent des résultats très proches de la réalité lors de la création d'arbres.

RECAPITULATION ET EXEMPLE CONCRET

Finalement, une grammaire peut mélanger déterminisme et dépendance au contexte pour générer des mots compliqués représentant fidèlement des arbres que l'on pourrait trouver dans la nature.

Il est important de noter que dans le cas de grammaires SIL, il est possible pour un même symbole d'écrire des règles de réécritures dépendantes du contexte, et d'autres indépendantes. Dans ce cas, les règles dépendantes du contexte sont forcément prioritaires sur les autres. Lors de la génération, à chaque symbole trouvé, une liste de règles éligibles est créée. Les règles dépendantes du contexte seront toutes testées pour

voir si elles sont applicables sur le symbole courant. Si après cette étude la liste est vide, les règles indépendantes du contexte seront essayées, sinon, aucune de ces règles ne sera ajoutée à la liste. Une fois la liste créée, une règle est choisie au hasard pour être réellement appliquée.

En annexe B, vous pouvez trouver un fichier de configuration commenté présentant les 4 types de grammaires avec des exemples de règles et les générations que produisent les grammaires. L'exemple est écrit dans le langage du programme que nous avons produit, il est donc recommandé de lire la section suivante avant de l'étudier pour en comprendre facilement la signification.

II. PARSER ET GENERATEUR

Afin de visualiser des grammaires dans le programme, nous avons choisi de laisser l'utilisateur créer des fichiers de grammaires. Ces fichiers doivent suivre un squelette précis que nous allons détailler dans cette section. Il n'est pas possible pour le moment de créer une grammaire directement dans le programme.

LES FICHIERS DE REGLES

En annexe A, vous trouverez le fichier README du projet qui contient le détail de la syntaxe d'un fichier de configuration. La description d'un L-System pour le programme reprend les éléments de base d'une telle grammaire :

- L'ensemble des symboles utilisables `V` (appelé `SYMBOLS` dans le fichier). On n'y fait pas la distinction entre les éléments constants et les non constants.
- L'axiome. Dans le programme, un axiome peut être un symbole unique (défini alors avec `AXIOM`), ou une séquence (définie avec `PHRASE`).
- La liste de règles (définie avec `RULES`). Les règles ont la syntaxe précédemment utilisée dans les exemples. Celle-ci est détaillée dans le README.
- Un angle optionnel définissant l'angle à utiliser dans les interprétations de la tortue (définie à l'aide d'`ANGLE`).

Nous ajoutons à cela un nom pour identifier la grammaire. L'utilisateur doit aussi préciser le type de la grammaire (`DOL`, `SOL`, `DIL` ou `SIL`). Il est de plus possible d'ajouter une interprétation pour chaque symbole qui sera utilisé lors de l'interprétation des mots générés. L'ordre des sections dans la description d'une grammaire est important.

Il est important d'écrire les grammaires à importer dans le programme dans des fichiers encodés en UTF-8 et portant l'extension ".lsys". Chaque symbole ne doit contenir qu'un seul caractère. La liste des caractères et des noms interdits est disponible dans le README.

Nous avons utilisé JavaCC pour parser le fichier de grammaires. Ainsi, si une erreur existe dans le fichier, un message d'erreur compréhensible apparaît pour l'utilisateur.

LA GENERATION DES SYMBOLES

Pour les grammaires de Chomsky, il existe de nombreux parsers ou générateurs. Nous avons par exemple défini une grammaire pour les fichiers ".lsys" gr^^ace à JavaCC. Cependant, il n'existe pas de tels outils pour les grammaires L-Systems. Ces dernières n'étant que peu souvent utilisées.

Nous avons dû écrire un générateur qui, à partir d'une grammaire L-System, peut générer des mots en appliquant successivement les règles de réécriture à partir de l'axiome. Ce générateur est capable de gérer les 4 types de grammaires et de générer des mots très longs.

Pour cela, il travaille sur des Symboles, des objets contenant une représentation (un caractère) et une interprétation. Le générateur travaille uniquement sur les représentations pour générer de nouveaux mots, alors que les tortues (les interpréteurs) ne travaillent qu'avec les interprétations des symboles.

La génération de DOL-Systems est très simple puisqu'elle ne nécessite que d'appliquer la règle correspondant au symbole courant s'il y en a une. Les DIL-Systems sont un peu plus compliquées et plus longues puisque pour chaque symbole, il faut parcourir l'ensemble de la chaîne pour savoir si le contexte permet ou non de valider la règle.

C'est pour les grammaires stochastiques que le problème se complexe. Nous avons vu que les règles dépendantes du contexte étaient prioritaires sur les indépendantes du contexte, et qu'il était possible de spécifier une probabilité pour chaque règle.

La difficulté commence avec le générateur de nombres aléatoires de Java. En effet, `Math.random()` ne permet pas de générer de vrais nombres aléatoires. Si l'on répète la même suite de demande de génération à cet objet, nous obtiendrons la même séquence. Afin d'obtenir du vrai hasard avec Java, il faut utiliser la classe `SecureRandom`.

Il faut ensuite gérer les probabilités données par l'utilisateur. Imaginons la liste de règles suivantes :

- `a < b -> b : 0.3 # "a" donne "b" si un "b" existe avant le "a" courant ; proba de 30 %#`
- `a > b -> 0.3 #"a" donne "b" si un "b" existe après le "a" courant ; proba de 30 % #`
- `a < ba > ba -> b : 0.4 # "a" donne "b" si il existe "b" et "a" avant et après le "a"" courant ; proba de 40 % #`

Et la séquence : `baba`.

Pour le premier "a", seules les deux premières règles sont applicables, chacune ayant une probabilité de 30 %. La probabilité finale n'est donc pas de 100 %. Il peut arriver qu'elle soit plus grande ou plus faible.

En réalité, ce que va faire le générateur, c'est de rapporter la somme des probabilités à 100 % et en modifier ainsi les probabilités de chacune des règles. Ici, les deux premières règles deviennent équiprobables. Chacune a une chance sur deux d'être choisie.

Il peut arriver que les chaines produites soient très longues (de l'ordre de plusieurs milliers de symboles). La structure utilisée (une liste d'objets Symboles) devient très lourde à gérer (plusieurs mébioctets) et à afficher. Le temps de traitement peut être assez long.

III. TORTUE

GENERALITES

Les interprétations de la tortue peuvent être très variées et différentes les unes des autres. Il est possible de mettre en place une interprétation de la tortue pour dessiner de simples lignes ou bien une interprétation avec un personnage qui effectue différents pas de danse selon les symboles à interpréter. Il est possible de tout faire.

Cependant, toutes les interprétations de la tortue ont des points communs. En effet, elles sont toutes des interprétations et cherchent à représenter d'une manière ou d'une autre une chaîne de symboles. C'est pour cette raison et d'autres qu'une tortue mère a été mise en place. Chaque interprétation de la tortue existante et future se doit d'hériter de cette classe mère qui regroupe toutes les caractéristiques communes des interprétations de la tortue. En plus de la liste des symboles et des mécanismes qui tournent autour, les tortues mises en place ont une construction commune du graphe de scène. Après la création d'une entité dans la scène, on rajoute celle-ci dans le graphe de scène afin de permettre au moteur 3D de l'afficher à l'écran. Pour les deux interprétations de la tortue existantes (Tube, Tree) le graphe de scène est construit de cette façon : lorsque l'on ajoute un tube ou une branche dans le graphe de scène, on crée un nouveau nœud qui se situe au bout de l'entité ajoutée. De cette façon, le prochain objet qui sera ajouté le sera au bout de l'objet précédent. Ce système permet aussi de mettre en place le système de sauvegarde de position et de restauration. Lorsque l'on souhaite sauvegarder la position courante, on sauvegarde le nœud courant dans une pile. Pour restaurer la position, il suffit de définir le nœud courant comme étant le premier nœud de la pile de sauvegarde.

Néanmoins, cette architecture du graphe de scène, bien que commune aux deux interprétations existantes, peut ne pas convenir à toutes les interprétations possibles. Il faudra modifier ce système et en ajouter une autre le cas échéant.

TUBE

TubeTurle est l'interprétation de la tortue de base. Elle est capable de représenter toutes les listes de symboles qui ne contiennent que les symboles de base. Elle est très simple et permet un rapide coup d'œil du résultat de la grammaire. Cette interprétation n'est capable de dessiner que des cylindres lorsqu'elle rencontre un symbole FORWARD, tous les autres symboles ne font que modifier la position ou la direction de dessin des prochains cylindres. Par conséquent, si une chaîne ne contient aucun symbole FORWARD, il n'y aura aucun résultat visible à l'écran.

Cette interprétation est historiquement la première et aussi la plus simple des deux. Elle a été conçue pour permettre une représentation la plus simple possible. Les objets dessinés sont donc des lignes 3D : des cylindres ou tubes. La conception de cette interprétation a permis de mettre en avant l'architecture du graphe de scène expliqué plus haut. L'interprétation de la tortue est capable de réaliser les actions suivantes selon les symboles à interpréter :

- Dessiner un tube à la position courante
- Tourner d'un angle fixe selon un des 3 axes
- Sauvegarder la position courante

- Restaurer la dernière position sauvée

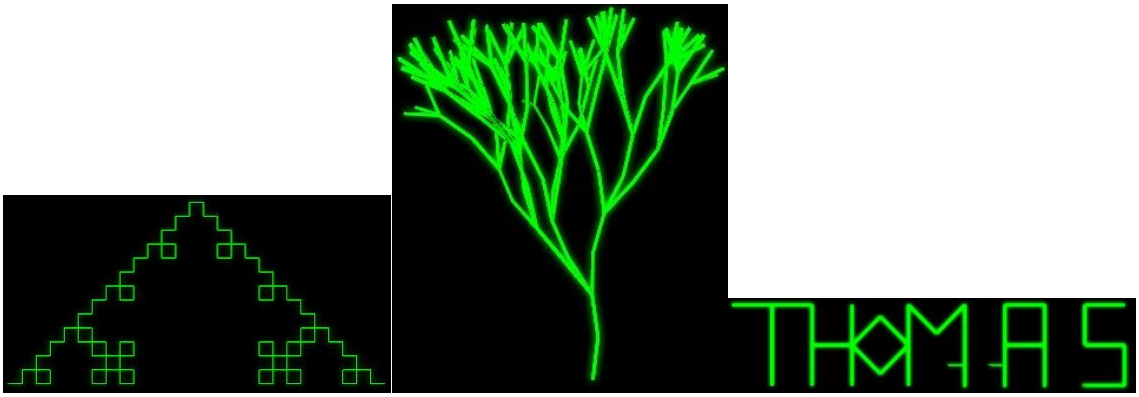
Pour réaliser l'action de dessin d'un tube, il suffit de créer un nouvel objet et de l'attacher au nœud courant. Il faut aussi créer un nouveau nœud fils et l'attacher au nœud courant en prenant soin de le déplacer de la taille du tube. De cette façon, le nœud fils est positionné au bout du tube dessiné. Le nœud fils devient le nouveau nœud courant.

Pour réaliser l'action de rotation selon un axe il suffit d'effectuer une rotation du nœud courant selon l'axe souhaité.

La sauvegarde la position courante utilise une pile de nœuds. À chaque appel de l'action de sauvegarde, on ajoute le nœud courant dans la pile de sauvegarde.

L'action de restauration de la position utilise la même pile de nœuds que l'action de sauvegarde. On dépile le dernier nœud ajouté dans la pile et on le considère comme nouveau nœud courant.

Ces simples actions permettent d'obtenir des rendus variés :



TREE

L'interprétation de la tortue Tree Turtle utilise les mêmes mécanismes que décrit précédemment pour Tube Turtle. Elle rajoute de nouvelles spécificités liées à la représentation d'arbres. Le tronc d'un arbre est plus mince et long que ses branches. Afin de simuler ce comportement, deux paramètres ont été rajoutés :

- Width reduction
- Length reduction

Ces deux paramètres sont respectivement les pourcentages de réductions de la longueur et de l'épaisseur d'un tube fils par rapport à son père. Un tube est fils de son père lorsqu'il est situé après celui-ci.

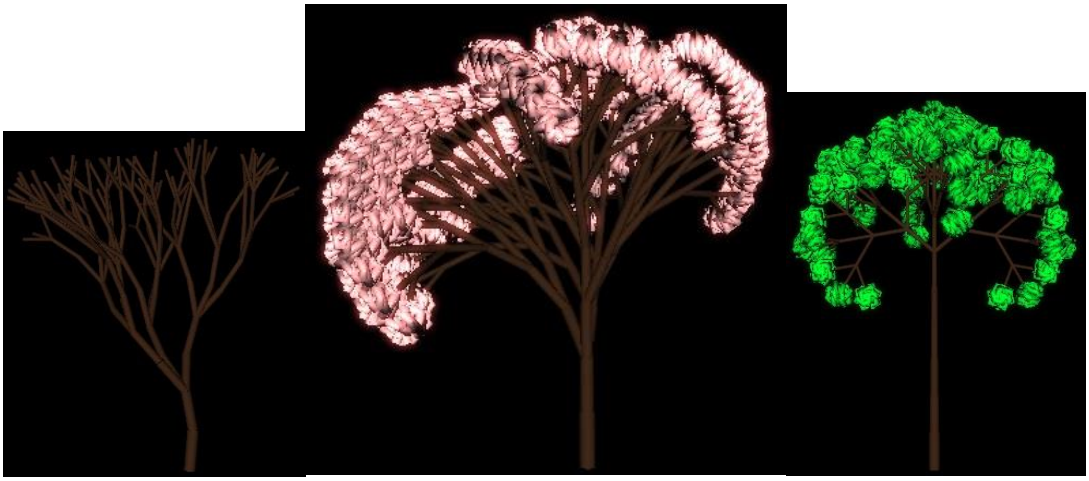
Sur l'image ci-dessus on peut voir une branche père (à gauche) avec sa branche fille (à droite). La branche fille est moins large et longue que sa branche père.

Cette interprétation est aussi capable de représenter le symbole feuille (interprétation 21). Les feuilles sont des représentés avec des tords afin de simuler un amas de feuilles.

Contrairement aux tubes, lorsqu'une feuille est ajoutée dans le graphe de scène on choisit de ne pas déplacer le nœud fils. En effet, un arbre possède des feuilles sur des branches uniquement et ne peut pas avoir de branches qui poussent sur des feuilles.

Ci-dessous le même screenshot que précédemment avec une branche père et une branche fils avec en plus une feuille.

Cette nouvelle interprétation permet d'obtenir des résultats d'arbres intéressants :



AVANCEES

Une évolution importante et particulièrement intéressante de l'application actuelle serait de pouvoir utiliser des meshs à la place des objets de base qui sont utilisés actuellement. Deux possibilités sont envisageables. La première serait de permettre à l'utilisateur d'utiliser ses propres meshs en les plaçant dans un dossier par exemple. Cette solution est la plus compliquée à mettre en place, mais permettrait d'avoir un rendu très varié et de pouvoir, par exemple, représenter tout type d'arbre avec l'interprétation Tree Turtle.

La deuxième possibilité serait de proposer un échantillon de meshs différents à l'utilisateur. Cette solution serait plus simple à mettre en place que la première sous réserve de trouver un bon échantillon de meshs.

Enfin, l'idéale serait d'avoir les deux. Un échantillon de meshs à proposer à l'utilisateur et la possibilité d'ajouter ses propres meshs dans la liste.

Une autre évolution de l'application serait d'un niveau plus technique. Lorsque l'on choisit d'afficher une chaîne très longue (dans le cas d'un nombre d'itérations important ou d'une grammaire très verbeuse), il est possible de faire exploser la pile de l'application.

En effet, JMonkey parcourt le graphe de scène de manière récursive. Lors de la création d'un grand tube sans aucune branche, le graphe devient longiligne est peut contenir plusieurs dizaines de milliers d'objets. Les appels récursifs font exploser la pile et l'application ne peut plus fonctionner correctement.

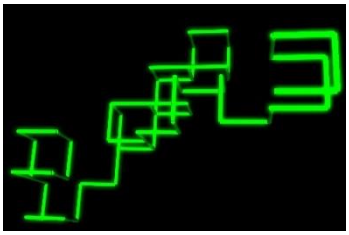
Il faudrait donc imposer une limite maximale d'éléments consécutifs dans une même branche, après quoi on force la création d'une branche qui reprendrait directement à la fin de la première. Nous n'avons malheureusement pas eu le temps de corriger ce problème.

IV. CONFIGURATION DE MOTEUR 3D

LUMIERE

La lumière est une composante indispensable d'une scène 3D, elle permet de mettre en avant les reliefs et de voir les objets tout simplement.

Le cas qui nous intéresse est assez spécifique. Nous ne connaissons pas à l'avance la disposition des objets et leur nombre. Il faut positionner un système de lumière qui permette d'éclairer la scène comme nécessaire dans toutes les situations. C'est avec ces contraintes à l'esprit que nous avons décidé de positionner une lumière directionnelle dans la direction de la caméra. Cette direction est mise à jour à chaque fois que la caméra se déplace. De cette façon la scène est toujours éclairée selon la position de la caméra et peu importe sa position. Même si la caméra est très éloignée des objets de la scène, ils seront éclairés par la lumière si la caméra regarde ces objets.



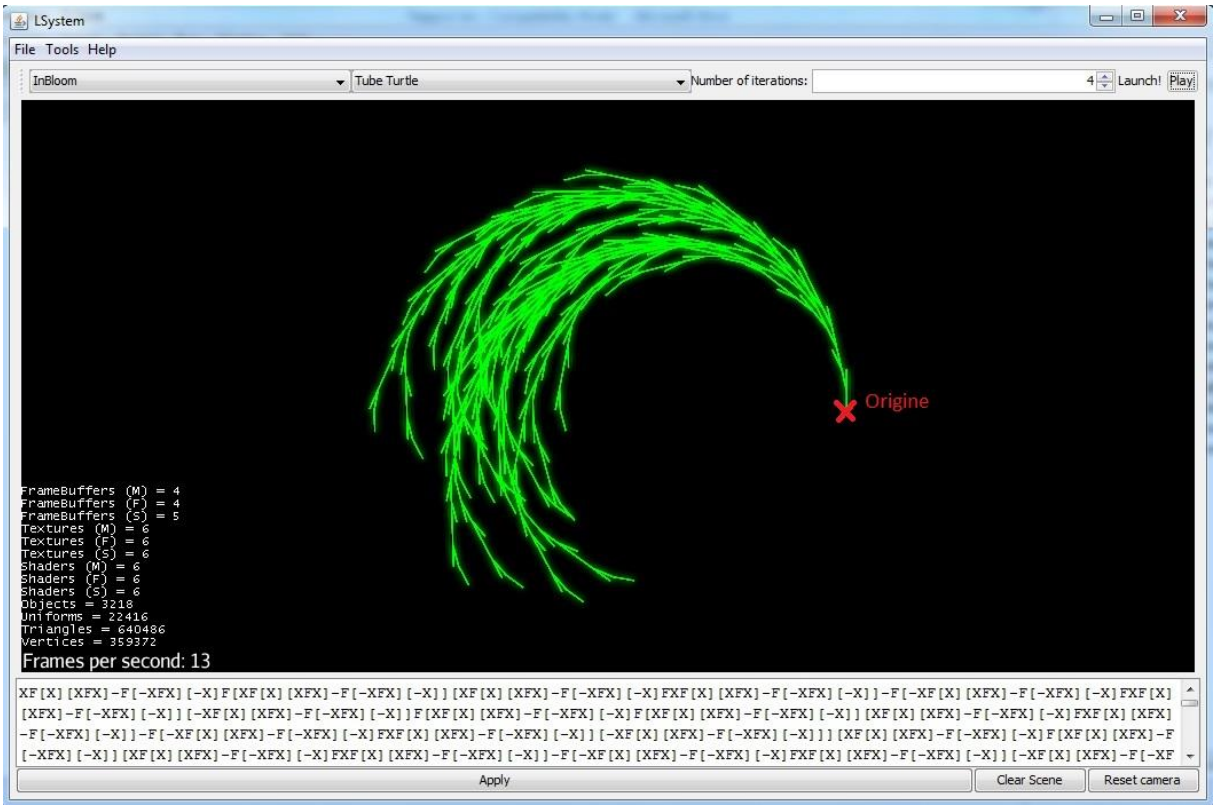
CAMERA

Le problème de la position initiale de la caméra est aussi lié aux mêmes contraintes que celles décrites pour la lumière. On ne connait pas le nombre d'objets et leur positionnement avant qu'ils soient affichés à l'écran.

La solution adoptée ici est différente. Lorsque l'on dessine un objet à l'écran, on regarde si cet objet dépasse le maximum ou le minimum de chaque axe de notre repère. De cette façon, à la fin de la création de la scène nous disposons des coordonnées maximum et minimum sur chaque axe des objets de la scène. Il est très simple avec ces données de calculer le point central de la scène et de centrer la caméra sur ce point.

Il reste néanmoins encore un problème, il faut reculer suffisamment la caméra par rapport à ce point central afin de voir toute la scène à l'écran. Ce problème a été résolu de façon presque empirique, nous prenons le maximum entre la longueur et la largeur de l'écran et on recule la caméra en multipliant cette valeur par un facteur déterminé par expérimentations. Cette solution n'est pas parfaite, mais fonctionne avec l'application actuelle.

Ci-dessous une capture-écran montrant le positionnement initial de la caméra avec une forme qui prend son origine sur la croix rouge. La caméra était initialement centrée sur ce point.



INPUTS

Afin de permettre à l'utilisateur de visualiser correctement le résultat produit par une interprétation, nous avons permis différents moyens de naviguer dans le monde 3D contenant l'interprétation, grâce au clavier et à la souris :

- Avancer : Z ou Flèche Haut
- Reculer : S ou Flèche Bas
- Se déplacer sur la gauche : Q ou Flèche Gauche
- Se déplacer sur la droite : D ou Flèche Droite
- Se déplacer verticalement vers le haut : Espace
- Se déplacer verticalement vers le bas : MAJ Gauche
- Accélérer tous les mouvements de déplacement : CTRL Gauche

- Tourner la caméra : bouger la Souris avec Clic Gauche enfoncé
- Tourner l'interprétation : bouger la Souris avec Clic Droit enfoncé
- Zoomer / Dézoomer : Molette de la Souris
- Réinitialiser l'orientation de l'interprétation : Clic sur le Molette de la Souris.

V. ARCHI/ÉVOLUTIVITE

MODELE MVC

Comme nous avons trouvé le projet très intéressant et qu'il n'existe pas réellement d'équivalent libre à ce programme, nous avons décidé d'en faire un logiciel libre qui pourra être continué par des tierces personnes. Nous avons donc choisi de porter un intérêt particulier à l'architecture du logiciel, notamment en implémentant un modèle MVC pour faciliter l'évolutivité du logiciel.

M – GRAMMAIRES

Le modèle est constitué des grammaires dont il faut se servir pour générer des listes de symboles qui seront alors interprétées et affichées. Pour cela, une classe `Grammaire` existe, qui contient tous les paramètres donnés dans le fichier la décrivant. Cette classe surcharge la méthode `toString()` qui génère une chaîne de caractères contenant la grammaire telle qu'elle doit être définie dans un fichier. L'exportation d'une grammaire est donc très facilitée (mais pas implémentée).

C'est le générateur qui utilise le plus les objets `Grammaires` afin de générer des `ListSymbols` contenant la liste des symboles générés. Le contrôleur accède aussi aux grammaires lors de l'édition de la signification des symboles. Le générateur agit comme une passerelle entre la grammaire et l'interprétation qui en est faite dans le logiciel.

V – GUI – JME

La vue est essentiellement composée de l'interface graphique. Celle-ci est composée d'une fenêtre principale et de 3 fenêtres de sélection / configuration. Toutes les actions faites dans ces fenêtres sont gérées par le contrôleur. La vue se contente d'être un écran manipulable par l'utilisateur.

La fenêtre principale contient aussi le canevas JMonkey dans lequel sont affichées les interprétations générées par les différentes Tortues. Celles-ci génèrent l'interprétation de la liste de symboles et l'affiche dans le canevas JMonkey à l'aide d'une fonction `Callable`. En effet, lors de l'utilisation de JMonkey dans un canevas, celui-ci se créée un thread spécial dans lequel il s'exécute et il n'est pas possible d'y accéder directement depuis un thread externe.

C – CONTROLLER

Le contrôleur est l'élément le plus important de l'application : c'est lui qui gère la manipulation des données et la génération des symboles et des interprétations.

C'est le contrôleur qui va permettre de charger un fichier de grammaires et de le transformer en une liste d'objets `Grammaires`. C'est aussi lui qui demande au générateur de créer une liste de symboles, puis l'envoie à l'interprétation pour qu'elle dessine cette liste dans JMonkey.

Seul le canevas JMonkey n'est pas réellement géré directement par le contrôleur. Celui-ci ne s'occupe que de sa création. Il est ensuite manipulé par les interprétations.

NOUVELLES INTERPRETATIONS

La création de nouvelles interprétations de la tortue est relativement simple à faire comme dit précédemment. Il y a quelques points importants cependant à ne pas omettre :

- Il faut rajouter un attribut `static int` dans la classe `Turtle` pour chaque nouvelle interprétation de la tortue ajoutée. Il suffit de copier une ligne déjà présente pour une autre interprétation et de prendre un entier non utilisé.
- La classe de la nouvelle interprétation doit hériter de la classe `Turtle` et redéfinir les fonctions `drawScene`, `checkSymbols`, `initParameters` et `setParameters`.
- Afin de permettre à la caméra de se positionner correctement il est important d'appeler la fonction `updateBoundsCoordinates` après l'ajout d'un objet dans la scène dans la fonction `drawScene` avec les coordonnées de cet objet.

GESTION DES PARAMETRES

Les paramètres sont une composante importante d'une interprétation de la tortue. On veut, par exemple, pouvoir changer la couleur des tubes, leur longueur, leur largeur pour l'interprétation Tube. Dans une optique d'évolutivité de l'application des paramètres génériques ont été mis en place. Ainsi si une nouvelle interprétation de la tortue est amenée à être mise en place, l'intégration de ses paramètres sera le plus simple possible. Afin d'ajouter un paramètre, il suffit d'ajouter deux lignes dans les fonctions `initParameters` et `setParameters` de la classe de l'interprétation de la tortue.

Par exemple si l'on veut ajouter un paramètre de type entier pour l'angle il suffit d'ajouter ces lignes :

```
parameters.add(new Parameter("Angle", ParameterType.TYPE_INTEGER, new Integer((int) angle)));
angle = ((Integer) parameters.get(0).getValue()).floatValue();
```

La première ligne permet de créer un nouveau paramètre ayant pour nom Angle, de type entier et dont la valeur initiale est celle de la variable angle.

La deuxième ligne permet de récupérer la valeur du paramètre et de l'affecter à la variable angle. Ainsi la valeur sera disponible dans la classe.

Une fenêtre de paramètres est créée automatiquement. Dans notre cas la fenêtre donne le rendu suivant :

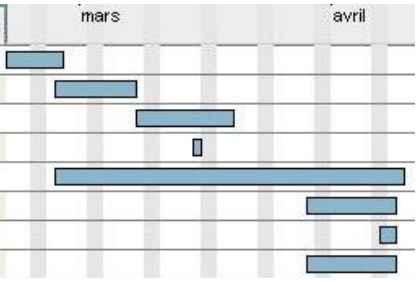
VI. GESTION DE PROJET

GIT

Afin de gérer nos sources, nous avons naturellement choisi d'utiliser Git. Il est simple d'utilisation et nous est familier. De plus, l'utilisation de Git permet également d'utiliser le dépôt GitHub qui est très répandu et permet une distribution de l'application gratuitement et rapidement. Notre application est ainsi disponible et libre d'accès à tous.

GANTT

Nom	Date de début	Date de fin
Configuration de l'environnement de développement	20/02/13	26/02/13
Parser & Generateur	26/02/13	07/03/13
Drawer & TubeTurtle	08/03/13	19/03/13
Importation des fichiers	15/03/13	15/03/13
Interface utilisateur	26/02/13	09/04/13
Gestion de la caméra	29/03/13	08/04/13
TreeTurtle	07/04/13	08/04/13
Gestions des Inputs	29/03/13	08/04/13



Ce diagramme de Gantt présente les fonctionnalités principales qui ont été réalisées au cours de ce projet. Il s'agit d'un diagramme construit avec les dates des commits du dépôt Git.

CONCLUSION

Ce projet fut particulièrement intéressant à réaliser pour plusieurs raisons. Premièrement, les L-Systèmes sont des types de grammaires spécifiques mais qui permettent de mettre en application des cours théoriques qu'il nous a été amené à suivre comme Théorie des Langages. Cette application est très visuelle grâce à la modélisation 3D et permet d'avoir une satisfaction certaine lorsque notre travail donne le résultat souhaité et permet une application directe des cours d'options de réalité virtuelle.

En plus de l'intérêt scolaire du projet et de la mise en application des connaissances théoriques acquises, le projet présente un réel intérêt puisqu'il existe peu de logiciel disponibles équivalents. C'est dans cette optique que nous avons choisi de mettre le résultat de ce projet sous licence GNU GPL v.3 afin de permettre sa distribution et sa réutilisation future par d'autres personnes intéressés par les L-Systèmes. La diversité des interprétations de la tortue possibles présente une réelle possibilité d'évolution du projet. Il sera intéressant et amusant de faire une interprétation de la tortue différente de celles qui génèrent des arbres ou différentes formes 3D. L'idée du personnage dansant déjà énoncée dans ce rapport en est un exemple. Plutôt que d'avoir une interprétation qui s'étends dans l'espace il s'agirait d'une interprétation qui s'étend dans le temps. Un modèle 3D fixe qui réalise une série d'animations.

Enfin, il y a des applications artistiques dans la génération de plantes par des L-Systèmes. Mais aussi dans tous les mondes 3D qui veulent donner une impression réaliste de forêt, plutôt que d'utiliser quelques modèles identiques d'arbres, la génération stochastique de L-Système permet de se rapprocher de la pousse réelle des plantes et arbres. Le lien suivant présente quelques exemples de représentation possible avec les L-Systèmes : http://philipgalanter.com/generative_art/wiki/index.php5?title=L-systems_II

BIBLIOGRAPHIE

Nous nous sommes tout d'abord documenté grâce aux sites internet suivants :

https://fr.wikipedia.org/wiki/L-System

https://en.wikipedia.org/wiki/L-System

http://www.geekyblogger.com/2008/04/tree-and-l-system.html

http://www.mizuno.org/applet/branching/

http://www.techno-science.net/?onglet=glossaire&definition=11374

Puis nous nous sommes aidés des logiciels suivants :

http://sourceforge.net/projects/magicgarden/ (logiciel libre multi-plateforme)

http://www.oocities.org/tperz/L4Download.htm

Enfin, nous avons largement utilisé la documentation Java et JMonkey :

http://docs.oracle.com/javase/7/docs/

http://jmonkeyengine.org/javadoc/

http://jmonkeyengine.org/wiki/doku.php

ANNEXES

Vous trouverez à la suite le fichier README du projet qui contient, en anglais, la description du programme et la manière de l'utiliser. Suit le fichier de grammaire que nous avons produit durant ce projet. Ce fichier est commenté et peut servir de base pour en générer des nouveaux. Vous trouverez des grammaires plus complexes et réelles dans le dossier "doc" du projet.

A. FICHIER README

L-SYSTEMS

This file describes the L-System project. You can export this file in HTML with Markdown by executing the file "exportREADME2html.sh" in a Linux based system, with Markdown installed.

I. INTRODUCTION

This project is realized by Thomas NOGUER and Rémi DUCCESCHI, students in 5th year at Polytech'Tours. It is supervised by Dr Sébastien AUPETIT.

The goal of this project is to interpret L-Systems grammar, generating words; and use JMonkey to create an animation based on the generated words (turtle interpretation or other).

THIS PROJECT IS ACTUALLY IN DEVELOPPEMENT

II. PRESENTATION

This project uses L-System grammars: it generates words from a given grammar and interprets these words to create a 2D / 3D scene (turtle interpretation or other).

1. L-SYSTEMS

A L-System is a specific grammar that differs from Chomsky's ones on how the rules are applied. In fact, in a L-System, all rules that can be apply on a word are applied in one go, whereas a Chomsky grammar will take X steps for X rules on the same word. See Wikipedia for more details.

A L-System is defined by:

- a list of symbols (aka V)
- an axiome (composed of one symbol)
- a list of rules
- eventually an angle to set-up the turtle interpretation.

NOTE: In a rule, the left-part, what will be changed, must be a single symbol.

There are 2 principle types of L-Systems that can be mixed-up whether they are context-free or not, determinist or not.

DETERMINISM

A L-System is determinist if a symbol, appear only once in the left-side of the rules. Otherwise, the system is stochastic. Here is a little example:

Consider the following rules (V = {a, b})

- a -> b
- b -> a

This grammar is determinist.

Contrary, this grammar is stochastic (same V):

- a -> b
- a -> ab

When we find "a" symbol, we don't know which rule to apply. So, we take one randomly (default is to have the same chance to take the first or the second one).

CONTEXT

A context-free L-System (noted OL) is a grammar where the choice of a rule depends only on the current character. There are much easier than the context-dependent L-Systems. The writting of their rules is also simpler:

- a -> bbb

In a dependent context L-System (noted IL), we have to specify a context that can be before or after the current symbol:

- b < a -> bbb : to apply this rule, we must have met "a" symbol previously, and actually be on "b".
- a > b -> aaa : to apply this rule, we must be on a "a" symbol, and a "b" must follow somewhere in the word.

we can give a context before AND after (this syntax is not very userfriendly, but it was to have a grammar LL(1) easily parsable in the file):

- b < bb > aa -> "baba"

Here, we transform the sequence "b" in "baba" only if we previously met the sequence "bb" and the sequence "aa" follows somewhere after.

NOTE: A context-dependent system may mix context-dependent rules with context-free rules. In this case, the context-dependent rules are tested first (priority for the context-dependent):

- a -> b (1)
- a > a -> b (2)

(1) will be tested only if (2) is not applicable.

NOTATIONS

We have seen that we can mix-up the 2 types of grammar. Here is how is identified a L-System:

- DOL is a determinist context-free L-System
- SOL is a stochastic context-free L-System
- DIL is a determinist context-dependent L-System
- SIL is a stochastic context-dependent L-System

2. INTERPRETATIONS

TUBE TURTLE

This is the basic turtle interpretatation. It is able to draw all the basic interpretations. It uses basic cylinders with parametrable width, length and color. Ths only interpretation that actually draw something is FORWARD. If you do not have any FORWARD symbol in your list of symbol to draw you will not be able to see any result on the screen.

TREE TURTLE

This is a turtle for drawing tree type grammar. It uses cylinders like the tube turtle but has more parameters such as width reduction and length reduction. These allow to have a trunk bigger than its branches. This interpretation can also draw the leaves with its specific symbol interpretation 21. The leaves' size and color can also be parametrable.

III. HOWTO (HOW TO MAKE THE PROGRAM WORK???)

Here is described how to interact with the program and how does it work (there will be in the future a real manual if we have the time (so there won't)).

1. LOAD A L-SYSTEM IN THE PROGRAM

The first thing to do is to give to the program a grammar and a word to work on in order to generate a sequence applying the rules of the given grammar on the word.

This sequence will be interpreted by the system to generate a beautiful animation.

1. WITH FILES

You have to write a config file where you describe the grammar you want to use. We have seen that a L-System is described by a type (DOL, SOL, DIL or SIL), a list of symbols, a list of rules, an axiom and a optional angle. In the program, we also give a name to the grammars.

Each of these will have to be written in the file. Let's see how it works.

NOTE: the file MUST be written in UTF-8.

NOTE: an example file can be studied in the doc/ folder: grammars-example.txt. A example is also available at the end of this section.

NOTE: You can add comments in a conf file by putting the comment between 2 characters "#". Comments MUST begin a new line.

NOTE: The following sections are in the order where they should appear in a conf file. For instance, the optional ANGLE definition must be after the AXIOME definition and before the RULES.

1. The name and the type

To declare a grammar, you have to write the name and the type of the grammar. The grammar will be written between "{}":

```
NAME
TYPE
{
    # ...
}
```

2. Symbols

You have to write all the symbols the grammar will use, even if they don't appear in the rules. To do so, you need to declare a bloc "{}" called "SYMBOLS".

A symbol is a simple character. Each symbol must be declared on a new line.

You can add an interpretation to a symbol by adding ": INTERPRETATION" after the character. INTERPRETATION must be one of the followings (basic order to the turtle interpretation):

- FORWARD (make the turtle go forward)
- TURNLEFT (turn the orientation of the turtle on the left)
- TURNRIGHT (turn the orientation of the turtle on the left)
- TURNUP (turn the orientation of the turtle up)
- TURNDOWN (turn the orientation of the turtle down)
- ROLLLEFT (make the turtle roll on the left)
- ROLLRIGHT (make the turtle roll on the right)
- ABOUTTURN (turn the orientation of the turtle by 180°)
- SAVEPOSITION (save the current position and orientation of the turtle)
- RESTOREPOSITION (restore the last saved position and orientation of the turtle)
- Any numbers strictly greater than 20 (for personalized interpretations depending on the turtle)

Here how to write it:

```
SYMBOLS
{
    a[: INTERPRETATION]
    # ...
}
```

3. The axiom

In the program, the axiom can be a simple symbol, or a word made with the symbols previously described. If it is a simple symbol, the declaration must be preceded by "AXIOM:", otherwise you should use "PHRASE:".

```
AXIOM: a | PHRASE: abab
```

4. The Angle

An angle can be given that will be used in the turtle interpretation for the symbols TURNXXX. You just have to put:

```
ANGLE: X
```

Where X ∈ ⟦0;360⟧ (thx UTF-8!!!)

If no angle is precised, the default value is 90°.

5. The rules

The rules are separated in 2: the left side (what may be replaced) and the right side (substitute), separated by "->".

```
a -> bb
```

For the stochastic systems, you may add a probability that indicates the chance a rule have to be selected when several can be. If nothing is written, the rules have all the same probability to be selected.

```
a -> bb[: X]
```

Where X ∈ [0;1]

The deletion is symbolized by a rule where the substitute (the right side) only contains the symbol "ε" (you can copy / paste it from here to put it in your file if needed):

```
a -> ε[: X] # the probability can be given, like for the other rules
```

Finally:

```
RULES
{
    X [< S] [> T] -> Y[: Z]
    # ...
}
```

6. Forbidden characters and words

For your symbols or the name of the grammar, you can use any symbols you want except the followings:

- Any numbers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- "{"
- "}"
- "ε"
- "#"
- "<"
- ">"
- ":"
- "\n", "\r", "\r\n" (end of line character)
- "\t" (tabulation)
- " " (space)

For the name of the grammar, you can not use the following reserved words:

- ANGLE
- AXIOM
- PHRASE
- RULES
- SYMBOLS
- DOL
- SOL
- DIL
- SIL
- FORWARD
- TURNLEFT
- TURNRIGHT
- TURNUP
- TURNDOWN
- ROLLLEFT
- ROLLRIGHT
- ABOUTTURN
- SAVEPOSITION
- RESTOREPOSITION

7. Complete file

Finally, this is what we get at the end:

```
NAME
TYPE
{
    SYMBOLS
    {
        a[: INTERPRETATION]
        # ...
    }

    AXIOM: a | PHRASE: abab

    [ANGLE: X]

    RULES
    {
        X [< S] [> T] -> Y[: Z]
        # ...
    }
```

}

You can successively write several grammars in the same file. The behavior of the program is not yet determined.

2. In the program

NOT IMPLEMENTED YET

2. Choose an interpretation

The tube turtle is the interpretable that you can choose to draw any grammar that uses only basic symbol interpretation. If you have any non-default symbol interpretation you can not use the tube turtle interpretation.

For tree-type grammar the tree turtle is the best fitted. It implements width and length reduction. It can also draw leaves with the symbol interpretation 21.

3. Launch a grammar

Here we describe all the steps you have to do to display a grammar in the program.

Import grammar

First, you have to import a grammar in the program. You can do it openning a file (File -> Open). You will be able to choose the grammar you want in the first combo box (top-left of the window), only if your program contains more than one gramar.

Choose an interpretation

Then, you can choose the interpretation you want in the second combo box. This one contains all the interpretations that can draw your grammar. If an interpretation you want does not appear here, it is because you use in your symbols definition a symbol that this interpretation cannot display. You should refer to the documentation of this interpretation.

You can edit your current grammar (the interpretation of each symbols) in "Tools -> Edit current grammar". Thus, you'll be able to remove the interpretations specific to an interpretation and the new interpretations may appear in the second combo box.

Choose a number of iterations

This is the number of steps the generator must do with the grammar. 0 display only the axiom. If you put to much here, the program may crash or take very long time to display the result.

Launch!

Clic the button "Launch!" to display the demanded iteration. If you clic on "Play", all the iterations from 0 to what you put in the spin box will be displayed sequentially. With this button, you can see the trees grow!

You'll see the result of the generation in the text area at the bottom of the window. It contains the word generated while the main part display the interpretation of this word. It is possible to edit this word and apply your changes to see what it does in the interpretation.

A bug exists in Windows for the change of the text. It is because of JMonkey with Swing...

EDIT THE TURTLE

Finally, you can edit the turtle (interpretation)'s configuration in "Tools -> Edit current Turtle". There, you can change the colors, the width... Refer to the documentation of the turtle for more.

IV. ABOUT TURTLES

It is possible to create your own turtle interpretation. For doing so it is advised to take example on the existing turtles.

V. LICENSE

This project is under GNU GPL v. 3.

L-Systems is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

B. FICHIER D'EXEMPLE : SIMPLE-GRAMMARS-EXAMPLE

FIRST_EXAMPLE

```
# Example of a conf file to load grammars in L-System program #

# first simple example of a grammar with 2 symbols: "a" and "b" #
# it is the same as the one discribed there: #
```

```
# https://fr.wikipedia.org/wiki/L-System Exemple 1: l'algue de Lindenmayer #
FIRST_EXAMPLE
DOL
{
	SYMBOLS
	{
		F : FORWARD
		a : TURNLEFT
		b : TURNRIGHT
	}

	AXIOM: a

	RULES
	{
		a -> aFbF
		b -> a
	}
}
```

SECOND_EXAMPLE

```
# Another simple example with interpretations #
# from: https://fr.wikipedia.org/wiki/L-System Exemple de la courbe de Koch #
SECOND_EXAMPLE
DOL
{
	SYMBOLS
	{
		# we precise the interpretation #
		F: FORWARD
		+: TURNRIGHT
		-: TURNLEFT
	}

	AXIOM: F
	# angle = 90°, same as default, we don't have to specify it #
	ANGLE: 90

	RULES
	{
		F -> F+F-F-F+F
	}
}
```

SECOND_EXAMPLE_2

```
# Based on the last one but in 3D #
SECOND_EXAMPLE_2
DOL
{
	SYMBOLS
	{
		# we precise the interpretation #
		F: FORWARD
		+: TURNRIGHT
		-: TURNLEFT
		^: TURNUP
		v: TURNDOWN
		A
		B
```

```
		C
		D
	}

	AXIOM: A

	RULES
	{
		À -> B-F+CFC+F-DvF^D-F+vvCFC+F+B
		B -> AvF^CFB^F^D^^-F-D^--F^B--FC^F^A
		C -> --D^--F^B-F+C^F^AvvFAvF^C+F+B^+^F^D
		D -> --CFB-F+B--FAvF^AvvFB-F+B--FC
	}
}
```

SOL_EXAMPLE

```
# a SOL example from http://www.techno-science.net/?onglet=glossaire&definition=11374 #
SOL_Example
SOL
{
	SYMBOLS
	{
		F : FORWARD
		X
		+ : TURNLEFT
		- : TURNRIGHT
		[ : SAVEPOSITION
		] : RESTOREPOSITION
	}
	AXIOM: X
	ANGLE: 20
	RULES
	{
		X -> F[++X]F[-X]+X: 0.2
		X -> F[+X]F[-X]+X: 0.8
		F -> FF
	}
}
```

DIL_EXAMPLE

```
# DIL example inspired by http://www.techno-science.net/?onglet=glossaire&definition=11374 #
DIL_EXAMPLE
DIL
{
	SYMBOLS
	{
		À : FORWARD
		B : FORWARD
		C : FORWARD
		+ : TURNRIGHT
		- : TURNLEFT
		[ : SAVEPOSITION
		] : RESTOREPOSITION
	}
	PHRASE: B[+C]A[-A]A[+C]A
	RULES
	{
		À < B -> B
		C -> B
		C < A -> A
```

```
                # this grammar is determinist: rule(2) is applied only if rule(3) can't be #
        }
}
# the run of this grammar is: #
# 0. B[+C]A[-A]A[+C]A #
# 1. B[+B]B[-B]B[+A]B #
# 2. B[+B]B[-B]B[+B]B and it stay stable on this state #
```

SIL_EXAMPLE

```
# last example much more complicated #
# we use a stochastic non context-free L-System #
SCIOUSC
SIL
{
        # we have 3 symbols: a, b and c #
        SYMBOLS
        {
                a
                # we don't precise an interpretation for b, the program will ask us for one #
                b: FORWARD
                c: ABOUTTURN
        }

        # here, we start with a sequence of symbols, so we don't use "AXIOM" #
        PHRASE: aabacc

        ANGLE: 20

        RULES
        {
                a -> b
                # the following rule has more priority than the first rule #
                a < c -> c
                # when "a" and "a", this rule has 50% probability to be applied, just like the
previous one if there also were a "c" #
                a < a > b -> bbb
                c < aa -> acb: 0.8
                # when we see "c", this rule has 20% to be applied, while the previous one has
80% #
                c > aa -> ccc: 0.2
                # when "b" and "bbb", we delete the "b" #
                b > bbb -> ε
        }
}
# an example of a run of this grammar is: #
# 0. aabacc #
# 1. bbbbbbacbacb #
# 2. bbcbbbbacbb #
# 3. cbcbb and it stay stable on this state #
```