Manuel d'utilisation

BENTROUDI Galaad - BLERON Alexandre
MEYRON Jocelyn - TRAN QUANG Son

19 juin 2014

1 Introduction

Dans tout ce qui suit, on appelle « prédécesseur » la partie gauche d'une règle et les successeurs seront en partie droite.

N'importe quelle règle peut s'écrire de la manière suivante : `Pred -> Operation Succ1 ... SuccN`. Les opérations possibles sont décrites plus bas.

2 Description de la grammaire

Les règles disponibles sont les suivantes :

2.1 Construction des fondations

- `Polygon2D`$((x_1, y_1), \ldots, (x_N, y_N))$: construction des fondations d'une forteresse sous la forme d'un polygone dont les coordonnées sont passées en arguments. Les coordonnées sont données dans le repère du terrain.
- `stronghold_seed`$((x, y), n, r)$: création d'une graine de forteresse dont l'origine est (x, y) qui est initialement un polygone régulier à n côtés et de rayon r.

Ces deux règles possèdent un seul successeur : les fondations de la forteresse.

2.2 Construction d'abjects

- `citywalls`(w, h) : s'applique sur un polygone et crée des murs de hauteur h et de largeur w de la forteresse en s'adaptant au terrain environnant. Elle possède 4 successeurs : les murs, les coins, l'intérieur et le mur sur lequel on doit mettre la porte.
- `build_door` : s'applique sur les murs de la forteresse et crée la porte de celle-ci en choisissant le mur où le terrain est le plus plat. Elle possède 4 successeurs : les 4 murs autour de la porte dans l'ordre suivant gauche, droit, haut et bas.
- `build_cylinder`$(r_{min}$ - r_{max}, h_{min} - $h_{max})$: s'applique sur un prisme (coin de la forteresse) et construit un cylindre dont le rayon sera choisi aléatoirement entre r_{min} et r_{max} et dont la hauteur sera choisie aléatoirement entre h_{min} et h_{max}. Elle possède un successeur : le cylindre crée.
- `build_rectangle`(w, h, d) : s'appliquer sur les murs de la forteresse et construit une tour rectangulaire de hauteur h, de largeur w et de profondeur d. Elle possède un successeur : le parallélépipède rectangle crée.
- `make_arrow_slit` : s'applique sur les murs de la forteresse et construit les meurtrières. Elle possède un successeur : les « nouveaux » murs de la forteresse avec les meurtrières dessus.

— `build_roof` : s'applique sur un objet (cube, cylindre) et crée un toit adapté à la forme initiale. Elle possède deux successeurs : le toit et la base du mur.
— `build_houses`(n) : s'applique sur l'intérieur de la forteresse et crée n maisons placées aléatoirement dans celle-ci. Elle possède un successeur : les maisons créées.
— `model`(s, "model.xxx", "texture.xxx") : s'applique sur n'importe quel objet et charge le modèle 3D "model.xxx" (beaucoup de formats sont supportés) associé à la texture "texture.xxx" (beaucoup de formats sont également supportés). C'est un terminal : il n'a pas de successeur.
— `make_village` : s'applique sur l'intérieur de la forteresse et crée un village dans celle-ci c'est-à-dire place le donjon, crée les maisons et les chemins à l'intérieur de celle-ci. Elle possède deux successeurs : la position du donjon et les maisons créées.
— `build_dungeon`(r, h) : s'applique sur la position du donjon précédemment déterminée et crée un donjon qui est une grande tour cylindrique de rayon r et de hauteur h. Elle possède un successeur : le donjon crée.

2.3 Opérations sur des volumes

— `select_faces` : ne s'applique qu'aux cuboïdes. Permet de désigner les faces du cuboïde. Elle possède 6 successeurs : les 6 faces du cuboïd : devant, derrière, en haut, en bas, à gauche et à droite.
— `select_edges`(w) : ne s'applique qu'aux cuboïdes. Permet de créer des rectangles en 2D à partir de chacune des arêtes du dessus avec une certaine épaisseur passée en argument. Elle possède 5 successeurs : le cuboïde, le rectangle à gauche, à droite, en haut et en bas.
— `extrude`(h) : ne s'applique qu'aux polygones 2D et aux rectangles 2D. Permet d'extruder avec une certaine hauteur passée en argument le rectangle 2D ou chaque côté du polygone 2D. Elle possède 1 successeur : le volume 3D de la surface 2D extrudée.
— `splitEqually` (dir, n) : ne s'applique qu'aux cuboïdes. Permet de diviser n fois un cuboïde selon la direction dir. Elle possède un successeur : le vecteur de n cuboïde issu du split.
— `splitRepeat` (dir, $vec < float >$, n, $alignement$) : ne s'applique qu'aux cuboïdes. Permet de répéter un motif de split passé dans vec n fois (ou infiniment lorsque n = -1). L'argument alignement prend 3 valeurs : `SPLIT_BEGIN, SPLIT_END, SPLIT_MIDDLE`. Il permet de savoir où placer l'espace vide lorsqu'on ne peut plus répéter le motif (resp. au début, à la fin et center le motif). Elle possède un successeur : le vecteur de n cuboïde issu du split.
— `splitProportions` (dir, $vec < float >$) : ne s'applique qu'aux cuboïdes. Permet de diviser un cuboïde selon les ratios passés dans vec (automatiquement normalisés). Elle possède un successeur : le vecteur de cuboïde issu du split.

2.4 Autres

— `show` : s'applique à n'importe quel élément et effectue son rendu à l'écran.
— `translation`(x, z) : s'applique aux fondations de la forteresse et la déplace suivant le vecteur de coordonnées (x, z). Elle possède un seul successeur qui est la forteresse translatée.
— `epsilon` : s'applique à n'importe quel élément et ne fait rien. Elle ne possède aucun successeur.

2.5 Contraintes et poids

En plus des règles précédemment décrites, on peut ajouter également des contraintes et des poids sur les règles. Il faut savoir qu'à chaque élément est associé un index qui permet de se répérer. Les contraintes disponibles sont les suivantes :

— odd : choisi les éléments impairs passés en arguments.
— even : choisi les éléments pairs passés en arguments.
— index = i : choisi le i-ème élément de l'argument.

Syntaxiquement, une contrainte s'ajoute de la manière suivante : `Pred -> Op Succ : {contrainte}`.

On peut également rajouter des poids aux règles ayant le même prédécesseur qui permettent de favoriser ou pas certaines règles. Syntaxiquement, un poids s'ajoute de la manière suivante : `Pred -> Op Succ : {contrainte} : {poids}`.

2.6 Exemples

Voici quelques exemples de grammaires :

2.6.1 Exemple simple

Dans cet exemple, on crée une forteresse : ses fondations, une muraille simple et des tours rectangulaires sans décorations :

```
1 : S -> Polygon2D((110, 120), (100, 150), (150, 200), (200, 200), (100, 100)) Stronghold;
2 : Stronghold -> citywalls(10, 30) MiddleRampart Corners Interior DoorWall;
3 : Corners -> build_rectangle(20, 100, 20) Towers;
4 : Towers -> show;
```

2.6.2 Exemple plus complexe

Dans cet exemple, on crée deux forteresses, leurs murailles avec des créneaux, des tours avec des décorations, une porte ainsi qu'un donjon et des bâtiments intérieurs :

```
1 : S -> Polygon2D((110, 120), (100, 150), (150, 200), (200, 200), (100, 100)) Stronghold;
2 : Stronghold -> citywalls(10, 30) MiddleRampart Corners Interior DoorWall;

3 : DoorWall -> build_door DoorLeft DoorRight DoorUp DoorDown;

311 : DoorLeft -> show;
312 : DoorRight -> show;
313 : DoorUp -> show;
314 : DoorDown -> epsilon;

4 : Corners -> build_rectangle (20, 100, 20) Towers : {tower};
41 : Towers -> select_edges (2) Base EdgeLeft EdgeRight EdgeTop EdgeBottom;
411 : Base -> show;

412 : EdgeLeft -> extrude(5) EdgeLeftExtruded;
4121 : EdgeLeftExtruded -> splitEqually(Z, 5) EdgeLeftCrenels;
4122 : EdgeLeftCrenels -> show : {even};

413 : EdgeRight -> extrude(5) EdgeRightExtruded;
```

```
4131 : EdgeRightExtruded -> splitEqually(Z, 5) EdgeRightCrenels;
4132 : EdgeRightCrenels -> show : {even};

414 : EdgeTop -> extrude(5) EdgeTopExtruded;
4141 : EdgeTopExtruded -> splitEqually(X, 5) EdgeTopCrenels;
4142 : EdgeTopCrenels -> show : {even};

415 : EdgeBottom -> extrude(5) EdgeBottomExtruded;
4151 : EdgeBottomExtruded -> splitEqually(X, 5) EdgeBottomCrenels;
4152 : EdgeBottomCrenels -> show : {even};

5 : Interior -> make_village DungeonPos Houses;

6 : DungeonPos -> build_dungeon (20, 100) Dungeon;
61 : Dungeon -> build_roof(15) DungeonRoof DungeonBase;
611 : DungeonBase -> split_cylinder (4) DungeonCylinders;
62 : DungeonCylinders -> show;
63 : DungeonRoof -> show;

7 : MiddleRampart -> make_arrow_slit NewWalls;
71 : NewWalls -> select_faces
    WallsTopSide
    WallsTopSide
    WallsTopSideUp
    WallsTopSide
    WallsTopSide
    WallsTopSide;

711 : WallsTopSide -> show;
712 : WallsTopSideUp -> extrude(2) CrenelTopVolume;

7121 : CrenelTopVolume -> splitEqually(Z, 3) CrenelTopVolume2;
7122 : CrenelTopVolume2 -> Crenels : { even };
7123 : CrenelTopVolume2 -> epsilon : { odd };
7124 : Crenels -> splitRepeat(X, (2.0,1.0), -1, Begin) Crenels0;
7125 : Crenels0 ->  show : { even };
7126 : Crenels0 -> epsilon : { odd };

72 : CrenelTopVolumeFinal -> show;

8 : Houses -> build_roof(3) HousesRoof HousesBase;
81 : HousesRoof -> show;
82 : HousesBase -> show;
```

2.6.3 Fichiers d'exemples fournis

3 fichiers d'exemples sont fournis :
— `minimal.in`
— `exemple1.in`
— `exemple3.in`

2.7 Compilation et lancement de l'application

L'application fonctionne uniquement sur les PCs des salles E103 et E301 du fait de la dépendance d'OpenGL 3. Elle peut marcher également sur d'autres machines sous OS X, Windows ou d'autres distributions Linux si les librairies suivantes sont installées :
— `glfw3`
— `glew`
— `assimp`
L'application dépend aussi de `cmake`.

2.7.1 Compilation

Dans le dossier `code`, les commandes suivantes permettent de compiler l'application

```
mkdir build
cd build
cmake ..
make
```

Afin de lancer l'application, il faut tout d'abord créer un fichier de règles, on supposera pour l'exemple qu'il s'appelle `rules.in` et se situe dans le dossier `code/grammars`. On lance alors l'application via les commandes suivantes :

```
build/main grammars/rules.in
```