A procedural building generator
PRIM Project - Report

Paul Vallet
<luap.vallet@gmail.com>

March 14, 2016

TELECOM ParisTech

Contents

1 Introduction

In this report, we will present the implementation of a procedural building generator using a shape grammar to produce 3D files. The basic grammar language presented in the report is inspired by the grammar language CGA++ [3].

These kinds of grammars are used by private software such as City Engine to help the user generate buildings that he will be able to integrate to his cities. The main goal in this project is to be able to generate buildings, but as the final objective would be to integrate them to a whole city, we will present a way to generate it. Indeed, the main interest of this kind of generation is to be able to produce a huge number of buildings with a low modelling cost. As the buildings differ slightly while keeping some similar features, the city will keep some visual consistency while remaining diversified. This is further discussed in Section 4.

A shape grammar is like a context-free grammar, but instead of being tokens, the terminal symbols are 3D modelling primitives. Their combined actions produce the 3D buildings. The primitives used in this project are listed in Section 3.

To be able to create a shape grammar, I have created a simple language to express the derivation rules, and further options. It will be referred throughout the report as CCGA (custom CGA). The software takes a file written in this language, computes an associated derivation randomly and outputs the resulting building as a `.obj` or `.off` file. This process is detailed in Section 2.

I've used CGAL to store geometry and to compute each geometric algorithm that will be presented in this report.

Figure 1: Example of buildings produced by the software

Paul Vallet `<luap.vallet@gmail.com>`

See Page 9

2 Global structure of the software

2.1 Code structure

```
initFromFile "plane.off"

%%

Parcel --> split("x") {~1: extrude(15) Tower | ~1: GreenSpace | ~1: extrude(12) Tower} ;;

Tower --> split("y") { 2: Base | {~0.4: Floor}* } ;;

Floor --> translate(0.2, 0, 0) Floor2 ;;
Floor --> translate(-0.2, 0, 0) Floor2 ;;

Floor2 1 --> translate(0, 0, 0.2) ;;
Floor2 1 --> translate(0, 0, -0.2) ;;
```

Above is presented a simple CCGA code. It reproduces an example presented in the original paper about CGA++ by Michael Schwarz and Pascal Müller [3]. It produces the two-towers building shown in Figure 2.

The source code is seperated in two distinct sections by the token `%%`. The first section allows the user to set global options. In this example, the only option used indicates that the axiom used as a blueprint is the shape contained in the file `plane.off` (which is a flat rectangle).

In the second section we define the rules of the grammar. They have a name and a series of actions to execute, given after the arrow. The terminal symbols are set in blue while the non-terminals are in green. As the grammar is context-free, there can be only a non terminal symbol on the left hand of the rule.

Here the code splits the rectangle axiom in 3 equal parts, extrudes two towers with fixed floors up to the height 2, and then produces as much floors of height 0.4 as it can. The rules `Floor` and `Floor2` will then shift randomly the floors.

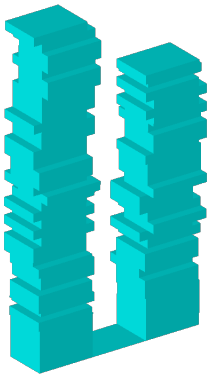

Figure 2: Simple generated building

2.2 Derivation process

Several rule versions can derive from the same non-terminal symbol. In this case, the rule version to be applied is chosen randomly among the different rule versions with the same non-terminal on the left-hand side. We can define weights as it is done in `Floor2` to influence the derivation process, even though in this case they are dispensable (the weight is 1 by default). This is at this point the only way to introduce randomness in the process, which is the main interest of a grammar, along with the ability to simulate function calls (even when the the grammar is deterministic, it is very useful to define rules to be able to call them several times, especially in a split pattern.

When the code is parsed, all the rules are given to a shape tree. The first one is added to a FIFO queue, and each call to another rule will add the named rule to the queue. If the non-terminal is not referred to in any rule, it will be considered as a terminal empty symbol (ϵ in classic grammars).

The derivation ends when the queue is empty.

However, when a succession of rule calls create a cycle (e.g. when a rules calls itself), the derivation could become infinite. The user could add a version of a rule that goes to an end with some probability. However, this is not very handy, so I've implemented a way to control the derivation.

In the second part of a CCGA code, the user can define an option to set n the maximum number of calls to a rule (infinity by default). At the n^{th} call, the rule will be replaced by a fallback one that can be defined by the user (empty rule by default). The annex presents a code that uses this function.

2.3 Geometry storage

Each time a primitive is called during the derivation, the program updates a shape tree. A rule contains all the nodes that will be affected by the right-hand side of the rule. The nodes contain the intermediate states of the geometry, and the final 3D building will be the sum of the shapes contained in the leaf nodes.

The way primitives affect the shape tree is greatly inspired from the one presented in the paper about the language CGA++ [3]. It is shown in Figure 3. The current node is in blue.

All the shapes contained in nodes are cuboids aligned along the x, y and z axis. There can be multiple cuboids or just one, and they can all have some missing faces. However, there will not be more complex shapes. I made this choice so that the manipulation of the shapes could be easier.

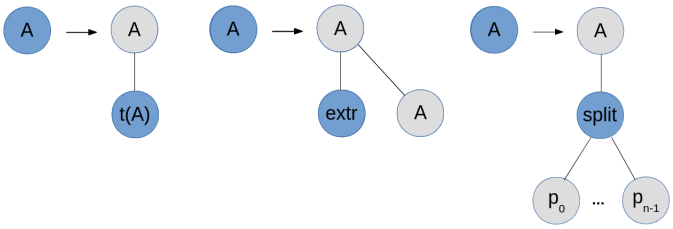

Figure 3: Examples of modifications of the shape tree. From left to right: translation, extrusion, split

3 Language specification

In this section we present the different terminal symbols implemented in the software.

`translate(x,y,z)` Creates a new node with shifted coordinates by the given offset.

`selectFaces(pattern)` Selects faces according to the pattern. The accepted patterns are described by the following regular expression `(all|(x|y|z)(neg|pos)?)`. If the pattern does not match this expression, all the faces will be unselected. The pattern refers to the normal of the faces to be selected.

I found this way of handling faces easier to handle than the `comp` function of the papers [2] [3].

`removeFaces()` Removes selected faces in a new child node.

`extrude(height)` Extrudes the selected faces of the current shape by the given amount and puts all the generated geometry in a child node. If more than one face is selected, the generated node will contain several cuboids.

This function also copies the shape of the current node to another child, as we remind that only the geometry of the leaves is taken into account. This is illustrated by Figure 3

`split(axis) pattern` Splits the current scope along the axis ("x", "y" or "z") according to the pattern `pattern`. This pattern consists in a list of actions associated with sizes. They can be absolute or relative. The lists can be nested, and repeated, with the symbol *. Figure 4 gives an example for scopes of different sizes.

The relative sizes fill the space between the parts with absolute size. When there is a repetition, several adapted relative sizes can be chosen (corresponding to the chosen number of repetitions). We keep the one that will be the closest to the input relative size. In the example of Figure 4, `A` must be the closest possible to 1.

To do this optimisation, we first put all the stars to zero, and then we optimise each one in order of appearance: we take all the possible values, starting from 0. As the distance function is affine, we keep the first value to be

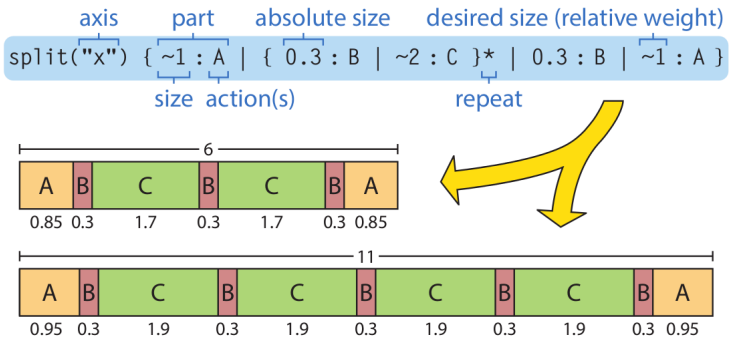

Figure 4: Example of pattern for the `split()` operation, instances for input sizes 6 and 11

closer to 0 than its successor. The local optimum however does not guarantee a global one, but it is not what we seek anyways: these kind of split patterns are not meant to be used with multiple stars.

Indeed, the result is quite difficult to anticipate as the optimisation of the first star could already yield an optimal result, keeping all the others to zero. However, it is still possible to do so (optimise on multiple stars), for example to optimise on big relative sizes, and then to refine with smaller ones.

Note though that it can be done with several splits, which offers more control. This should be the way to use split patterns.

`textureFaces(textureName)` Applies the texture referenced by `textureName` to the selected faces of the current node. This function does not create a child node.

The texturing process is exposed in subsection 3.1.

`createRoof(angle,offset)` The current node will contain an independant roof created from the subnodes that will call the function `roof()`. `angle` gives the gradient of the roof. `offset` indicates how much the roof exceeds the ceiling. See Subsection 3.2 for more details.

Calling this function will also discard all the roofs created in parent nodes, ignoring them for roof generation. By default, the program calls `createRoof(20,0)` on the root node, so that the use of function `roof()` is always defined.

`roof()` Adds the top faces of the current node (with an union) to the ceiling of the closest parent to have been called with `createRoof()`.

3.1 Texture handling

To store the geometry of each node of the shape tree, I've used CGAL framework `Surface_mesh`. However, CGAL does not support the use of texture coordinates. And even if it has a simple way to output geometry to `.off` files, it does not support `.obj` ones. Therefore, I implemented these features myself.

The process to use textures follows different steps :

- The user specifies in the options section of the CCGA code the path to the texture file. There is only one file for the whole building (plus eventually one for the roof).
- Still in the same section, he has to define sub rectangles of the texture, and name them.
- He can then apply these textures with the function `setTexture`.

The following code gives an example of a CCGA code using texturing. It builds the student's residence of my school, Télécom ParisTech. The result building is presented in Figure 5

```
initFromFile "plane.off"
setOutputFilename "examples/maisel.obj"
setTextureFile "../res/maisel.png"
addTextureRect "withoutbalc" 0. 0.625 1. 1.
addTextureRect "withbalc" 0. 0.25 1. 0.625
addTextureRect "longbalc" 0. 0. 0.488 0.21875
addTextureRect "shortbalc" 0.4746 0. 0.6543 0.21875

%%

GlobalShape --> extrude(4.8) split("x") {~1: ColumnA | ~1: ColumnB }* ;;

ColumnA --> split("y") {~0.4: WithBalc | ~0.4: WithoutBalc }* ;;
ColumnB --> split("y") {~0.4: WithoutBalc | ~0.4: WithBalc }* ;;

WithoutBalc --> selectFaces("z") setTexture("withoutbalc") ;;
WithBalc --> selectFaces("z") setTexture("withbalc")
             split("x") {~56:Trash | ~200:Balcony | ~200:Balcony | ~56:Trash} ;;

Balcony --> split("y") {~90: selectFaces("z") extrude(0.15) TextureBalc | ~100:Trash} ;;

TextureBalc --> selectFaces("z") setTexture("longbalc") selectFaces("")
                selectFaces("x") setTexture("shortbalc") selectFaces("")
                selectFaces("ypos") removeFaces() ;;
```

The texture rectangles are stored as a map from faces to indices of an array containing the rectangle coordinates. The names give access to these indices. Therefore, each vertex can have several texture coordinates that depend on the considered face. Figure 5 presents the building generated with the code above.

The real difficulty arrives when we have to split an already textured face. In this case, we want to keep the texture in all the generated children. We then have to split the texture rectangle accordingly, add these new textures to the texture coordinates array. They cannot be referenced by text as it could interfere with user's definitions. There is therefore another mapping from splitting patterns to subtextures, which avoid generating too many new subtextures.

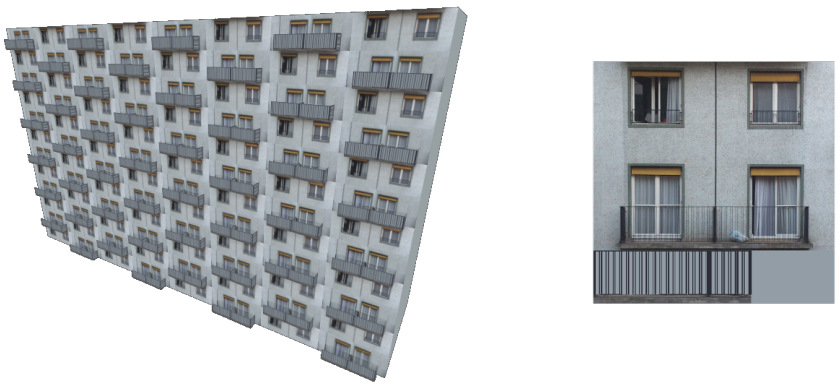

Figure 5: The Maisel: a building using textures and splits, and its texture file

This way of texturing offers a lot of control, but is not very handy when having to texture complex facades. CGA and CGA++ languages [2] [3] feature automatic texture generation for facades. In CCGA, we have to split the shape and texture each face. Moreover, as it only supports one texture, we cannot easily tile textures with texture coordinates out of $[0, 1]$. This is why the roof texture comes separately: this easy form of tiling is necessary.

3.2 Roof generation

For each node upon which we have called `createRoof()`, we store the ceilings as a map of heights to lists of disjoint 2D polygons. As said in `roof` specification, the top faces of nodes are added to the adequate list of polygons according to the height of the face. An union is then performed to update the list of disjoint polygons.

Each ceiling layer is also copied to all the others underneath, to avoid to have ceilings such as the one shown in Figure 6: the lower part of the roof should have an edge going to the upper one, to avoid rain water to go straight to the wall.

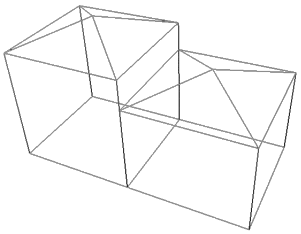

Figure 6: Generated roof without copying ceiling geometry throughout the levels

At the end of the derivation, there will be no more calls to add polygons to the ceilings. The program will then compute the roofs according to the method shown below.

3.2.1 Generating roofs from polygons

In order to compute the roofs of each node upon which we have called `createRoof()`, we use an intuitive method derived from the straight skeleton of a polygon presented in the article [1], on each disjoint ceiling 2D polygon.

The straight skeleton of a polygon is the result of a continuous shrinking process. The edges are moved inwards at a constant speed in the direction given by their normal. The edges of the skeleton are defined by the trajectories of the vertices. It is shown in blue lines in figure 7.

Then, we can compute the roof by setting to the contour edges the height of the ceiling and adding to the inner vertices $d \cdot \tan angle$, where d is the distance of the vertex to its generating contour edge, that is the edge that has been shrinked to length 0 to generate the vertex.

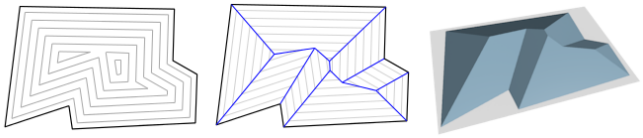

Figure 7: A straight skeleton and its associated roof

3.2.2 Offsetting roofs

In this implementation, the straight skeleton has another use. It allows to compute an offset polygon from the original one: during the shrinking process, the offset polygon is seen (if the offset is within the right bounds).

However, for the roofs, what is interesting would be to have a negative offset, in order to enlarge the polygon. This can be through a derived way, presented in Figure 8.

This method consists basically in shrinking the complementary polygon. We have to create a frame: an enclosing rectangle big enough so that the straight skeleton around the polygon we want to enlarge is not affected. We can then use the offset of polygons and remove the frame. All the holes of the polygon must be processed separately.

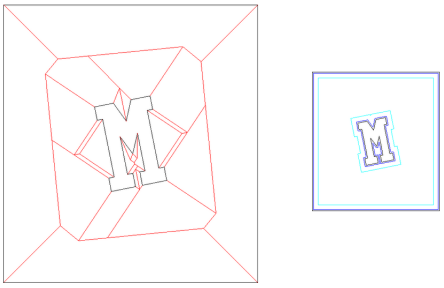

Figure 8: Enlarging polygons with the straight skeleton using a frame

See Page 9

4 Town generation

The primary objective of this project is to generate variations of buildings that look similar but have differences. However, what is interesting in procedural modeling is to be able to generate large amounts of buildings and integrate them in an environment.

Let's start from the variations of a house shown in Figure 9. These are houses of the style of houses from the south of France. The positions of the entrance and the garage doors are enforced, and floors can rise as modules. The CCGA code for these houses is presented in the Annex.

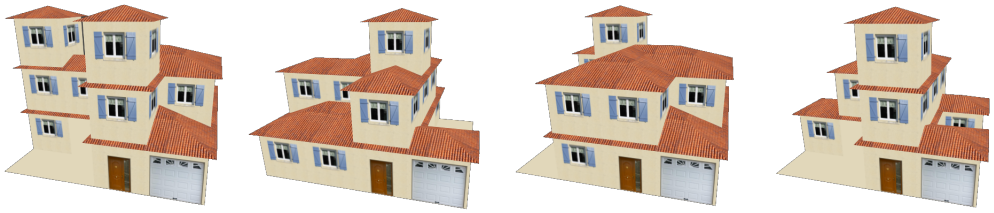

Figure 9: Variations of houses from the south of France

The user could then use a script to generate lots of variations and integrate them in his city. He can also directly conceive a grammar from it, and only add a few lines of code as shown below.

```
initFromRect 130 130
...
addTextureRect "road" 0.82 0.75. 0.82001 0.75001

%%

Village --> split("x") {~10:VillageX | ~3:Road}* ;;
VillageX --> split("z") {~10:House | ~3:Road}* ;;

Road --> selectFaces("ypos") setTexture("road") ;;

House --> Road ;;
House --> ...

...
```

The initialisation is made from a much bigger rectangle, and the area is splitted in roads and houses, the code of the house being pasted underneath.

The result can be seen in Figure 10.

Figure 10: Town generated with multiple uses of the rule `House`

At first, the storage of roof geometry was in the shape tree, common for all nodes. With this first solution, we

were already able to generate big cities, but when having several roof levels, the computation would take a lot of time, and eventually fail.

This is why the solution of the local roofs with `createRoofs()` was implemented. The computation time is now much reduced. However, we still cannot generate bigger cities as at some point the CGAL function generating offset polygons loops infinitely. There is also another bug where CGAL skeleton generation fails on simple polygons, on some derivations. Others work well.

We can still generate very big cities when we don't use roofs. Figure 11 shows a big city generated in less than a minute. The `.obj` weighs more than 9 Mo, that's why it is not on the git repository.

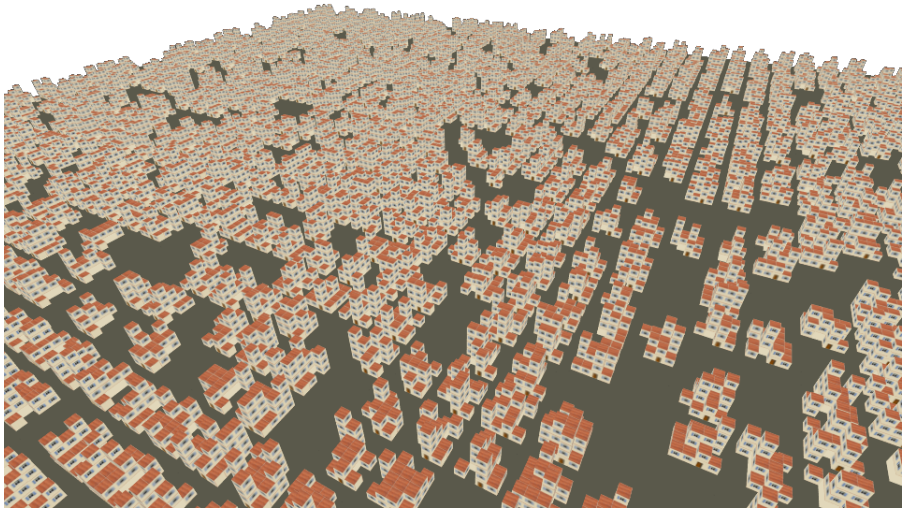

Figure 11: A bigger town generated using a simple texture for roofs (500x500 houses)

5 Conclusion

This project allowed me to successfully implement some interesting framework around shape grammars. Even if the results are simple, the main interest of the project was to build the generator from scratch. It was a great opportunity for me to face the difficulties of using CGAL, which is not user friendly, and to discover Flex and Bison, which are quite central to the project.

To go further in the project, we could add some features such as rotations, or other types of roofs, for instance the ones presented in the paper [1] (I find the gable one particularly interesting). But the main improvement would be a script to simplify the output geometry. Even though it is quite optimal for our texturing strategy, we would need to generate them automatically and reduce the number of polygons, so that the buildings could be efficiently used in other environments.

References

[1] R.G. Laycock and A. M. Day. Automatically generating roof models from building footprints. 2003.

[2] P. Müller, P. Wonka, S. Haegler, A. Ulmer, , and L. Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006.

[3] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4 (Proceedings of SIGGRAPH 2015)):107:1–107:12, August 2015.

6 Annex

```
initFromRect 10 10
setOutputFilename "examples/house_south.obj"
setRoofZoom 2
setRoofTexture "../res/roof_south.jpg"
setTextureFile "../res/house_south.png"
addTextureRect "garage" 0. 0. 0.5 0.45
addTextureRect "wall" 0.5 0. 1. 0.5
addTextureRect "window" 0. 0.5 0.5 1.
addTextureRect "door" 0.5 0.5. 1. 1.

%%

Foundations --> createRoof(20,0.3) split("x") {~1: Garage | ~2: Center1 | ~1: EdgeZ } ;;

Garage --> split("z") {2.5: GrowLevel roof() TextureGarage | ~1: EdgeZ} ;;

Center1 --> split("z") {~1:Front | ~2: GrowLevel Center | ~1: EdgeX} ;;

Front --> split("x") {~1: GrowLevel roof() TextureEntrance Room | ~1: Edge} ;;

EdgeX --> split("x") {~2.5: Edge}* ;;
EdgeZ --> split("z") {~2.5: Edge}* ;;
RoomX --> split("x") {~2.5: Room}* ;;

Edge --> Nothing ;;
Edge --> Room ;;

Room --> GrowLevel roof() TextureWall ;;
Room --> GrowLevel TextureWall Room ;;

Center --> split("z") {~2.5:RoomX}* ;;

TextureGarage --> selectFaces("zneg") setTexture("garage")
                  selectFaces("") selectFaces("zpos") selectFaces("x") setTexture("wall")
                      ;;

TextureEntrance --> TextureWall selectFaces("") selectFaces("zneg") setTexture("door") ;;

TextureWall --> selectFaces("all") setTexture("wall") PaintWindow ;;
PaintWindow --> selectFaces("z") setTexture("window") ;;
PaintWindow --> selectFaces("x") setTexture("window") ;;
PaintWindow 2 --> selectFaces("all") setTexture("window") ;;

GrowLevel --> selectFaces("") selectFaces("ypos") extrude(2.5) ;;

setRecDepth GrowLevel 7
fallback GrowLevel --> roof() ;;
```