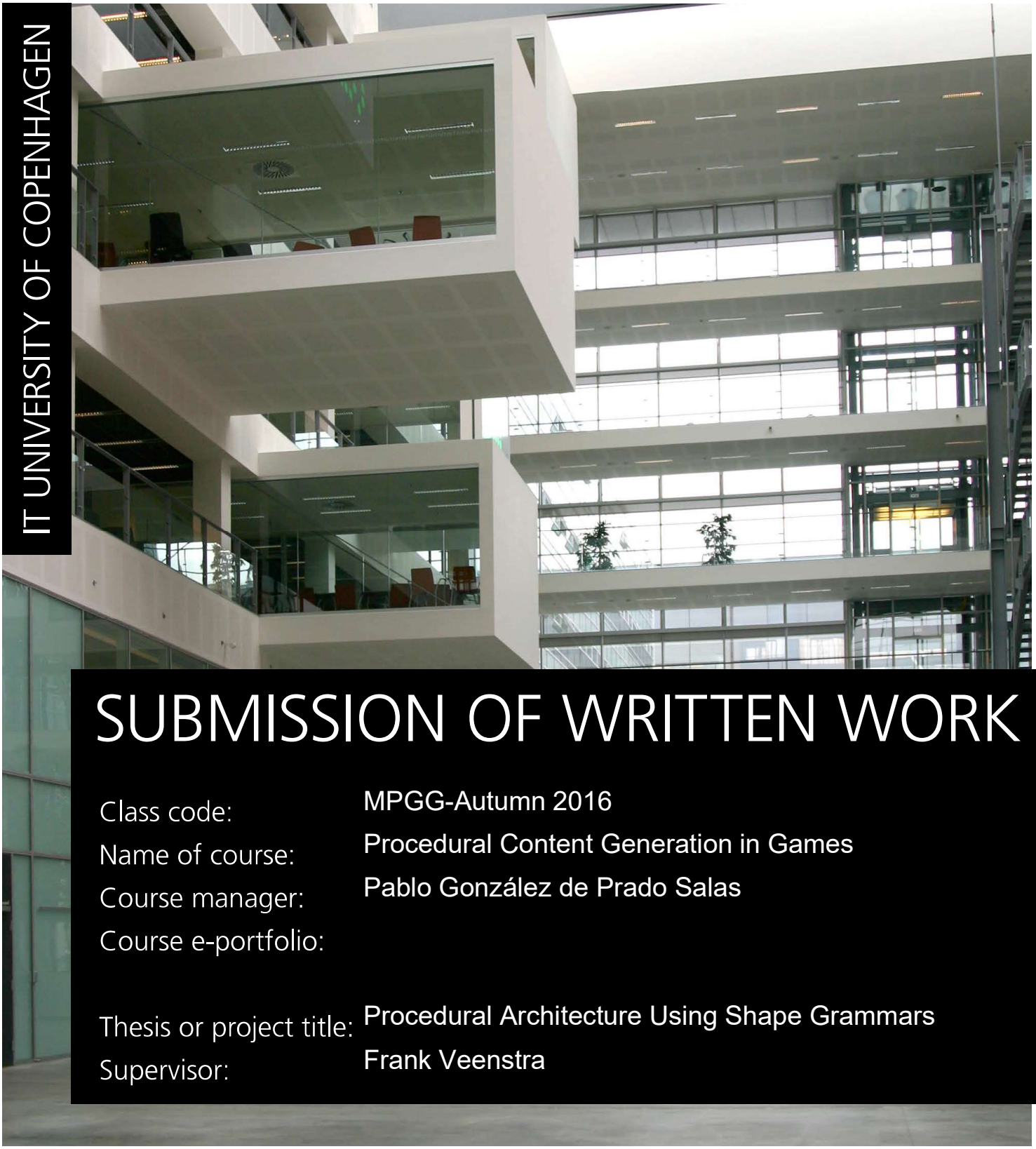
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code: MPGG-Autumn 2016

Name of course: Procedural Content Generation in Games

Course manager: Pablo González de Prado Salas

Course e-portfolio:

Thesis or project title: Procedural Architecture Using Shape Grammars

Supervisor: Frank Veenstra

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Alberto Álvarez	30/11-1992	aalv @itu.dk
2.	Óscar Manuel Losada Suárez	30/05-1992	oslo @itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Procedural Architecture Using Shape Grammars

Alberto Álvarez
Óscar Manuel Losada Suárez

December 15, 2016

Abstract

This paper presents the work done towards the creation of a framework that can be used to procedurally generate buildings. The framework is based on the use of stochastic shape grammars and provides a syntax to define advanced rules with greater expressive power than "vanilla" grammar rules have. The framework is shown to be usable to generate structurally interesting buildings.

1 Introduction

The usefulness of procedural content generation (PCG) in games has been argued for in the past and there are many commercial games where it has been used to great effect, see for example Minecraft (Mojang, 2011), Spelunky (Mossmouth, 2008), Diablo 3 (Blizzard Entertainment, 2012), Path of Exile (Grinding Gear Games, 2013) and Borderlands (Gearbox Software, 2009). As discussed in (Shaker, Togelius, & Nelson, 2016), PCG in general can, among other things, reduce the production cost of games by removing the need to spend human work hours to create content and can augment the creativity of individual human creators. Attaining these virtues in a procedural building generation tool is the goal of this work.

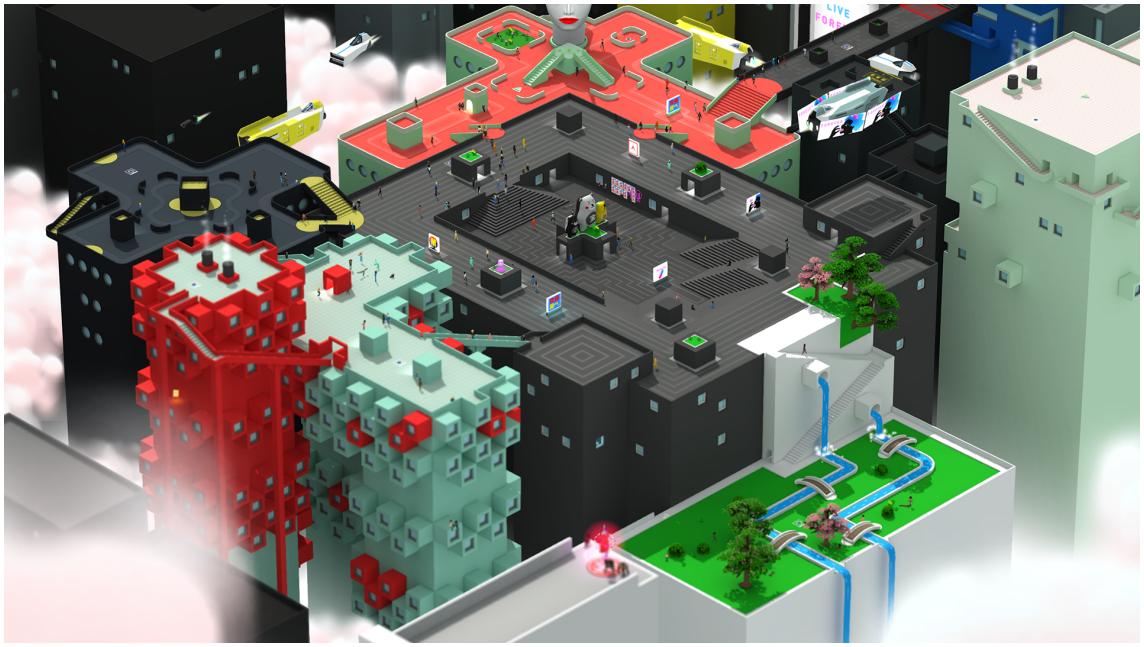


Figure 1: This image from Tokyo 42 shows the style we aimed to recreate: minimalistic, emphasizing the aesthetics of structure and geometry.

As far as the type of content that this tool can be used to create, namely buildings, it seems indisputable that it could be used in games, since urban settings are fairly common in games. In this context for example, modeling a large and varied 3D urban environment can take a large amount of work, which could be reduced by instead defining a fairly compact rule set for this tool

and generating the buildings with it. Another advantage of the use of tools such as this one is that their stochastic nature can provide games with a degree of unpredictability for the player that can increase the replayability of the game.

In order to set a frame of reference, Figure 1 shows a scene from the upcoming game Tokyo 42 (SMAC Games, 2017), where as far as we know, the buildings are handcrafted. This is the style of buildings we set out to create and it might help understand the level of detail and feature that we have aimed to be able to express with our system.

2 Background

As indicated in (Ruiz-Montiel et al., 2014), shape grammars have been used for various generation tasks because of "their power to capture and recreate heterogeneous design styles", they are compact - few rules "can yield [...] complex and unexpected shapes" and can be used to automate design.

The work presented here is heavily based on that of (Müller, Wonka, Haegler, Ulmer, & Van Gool, 2006), who describe system for procedural modeling of buildings that draws from shape grammars ((Stiny, 1980), (Togelius, Shaker, & Dormans, 2016), (Veenstra, 2016)) and attributed grammars (Knuth, 1968), where functions are associated with production rule in the grammar to define and modify attributes for the symbols in the grammar.

Other approaches that were contemplated were based on the use of Cellular Automata (CA) to generate buildings (Krawczyk, 2002), however the obtained results seemed less structured, varied and controllable. The more irregular patterns shown by Krawczyk did suggest to us that there might be potential in combining CA with grammars to produce buildings with irregular structures (cf. Section 5.2).

3 Methods

3.1 Grammar system description

The framework defines a syntax for the creation of rewriting rules that will form a stochastic shape grammar and provides the parsing-compiling system to load the rules from files, the rewriting system to apply the rules in parallel (L-Systems (Togelius et al., 2016)) a certain number of times and a simple example interpreter that transforms the structures generated into actual meshes. The framework is implemented in C# and uses Unity (Unity Technologies, 2005) to render the results.

Rules have the following general structure:

$$\begin{aligned} &Predecessor -> Successor_1 : Probability_1 | ... | Successor_N : Probability_N \\ &where \sum_{i=1}^{N} Probability_i = 1 \end{aligned} \tag{1}$$

Every time one of these rules is applied, one of the successors is chosen at random with the probability specified in the rule for each of them.

The system uses a tree structure to store the results of the rewriting process in *Nodes*. These *Nodes* are comprised of a *Symbol*, which identifies them for the purpose of applying rewriting rules, a *Scope*, which defines their position, rotation and scale in space, and a *Shape*, which holds information about the geometry. The leaves of the tree represent the current configuration. Storing information in this way makes the rewriting easier by allowing us to maintain the previously mentioned information, which would be hard to keep track of with a traditional string representation. This approach also has the advantage of maintaining information about the structure of the generated object which could be used with certain types of rules in future expansions of the system.

What follows is a description of the rule types that are supported by the system currently.

3.1.1 Transform and Replace

Transform and Replace rules are used to replace a *Node* with other nodes that inherit that *Node*'s scope and allow us to apply translation, rotation and scale to the new *Nodes*. This type of rule looks like this:

$$A- > Transform(a = 1, b = Random(2, 5)) \\ \{[S(a, 2, a)I(CUBE, cube, outer)]T(a, 0, a)I(CYLINDER, cylinder, outer)\} \tag{2}$$

Where after the $Transform$ keyword we can declare local variables for that rule, we can apply rotations with $R(x, y, z)$, scales with $S(x, y, z)$ and translations with $T(x, y, z)$, we can push and pop the current scope defined by these transformations with [and] respectively and we can create new *Nodes* with $I(symbol, geometryPrimitive, visibleFaceSide)$.

3.1.2 Subdivide

Subdivide rules split the predecessor *Node* into new *Nodes* along one of the local axis of the predecessor. See an example:

$$B- > Subdiv(x, 1r, 2r, 1r) \\ I(BX, cube, OUTER), I(BX, cube, OUTER), I(BX, cube, OUTER)outer)\} \tag{3}$$

The new *Nodes* inherit the parent's *Scope*, except for the scale in the splitting axis, which is determined by the parameters after the axis. The suffix r marks these parameters to be interpreted as relative the other relative values and the scale of the parent in the split axis or absolute, which simply serves a robust way of aligning new *Nodes* with their parent. Note that absolute values allow the subdivision children to go beyond the bounds of the parent which can be used for alignment purposes and stacking *Shapes*. In the last part, the new Nodes are specified with the same syntax as in the Transform and Replace rules.

3.1.3 Repeat

Repeat rules tile the predecessor along one of it's local axis into a pattern of new Nodes. These rules have the following structure:

$$B- > Repeat(x, strict, 0.5, 1) \\ I(C, cube, outer), I(D, cube, outer)\} \tag{4}$$

After the *Repeat* keyword we define the repeat axis (x, y or z), whether the pattern can be broken if it doesn't fit perfectly or not (!*strict* or *strict* respectively) and then the scale of the children along the repeat axis, which otherwise inherit the parent's *Scope*. In the last part, the new Nodes are again specified with the same syntax as in the Transform and Replace rules.

3.1.4 Component Split

Component Split rules create new *Nodes* from the faces, edges or vertices of the predecessor or subsets of the same (e.g. side faces, top face, bottom face, etc). Here is the an example:

$$B- > Comp(rename, faces, edges) \\ FACE, EDGE\} \tag{5}$$

The first parameter ($rename$ or !$rename$) specifies whether the *Symbol* of the new *Nodes* should have a number suffix appended the the *Symbol* specified in the last part of the rule (e.g. in this case, the new *Nodes* corresponding to the faces would have symbols $FACE1$, $FACE2$... which allows us to distinguish them after the operation is performed). The rest of the parameters select the components that the predecessor will be split into and are mapped to the *Symbols* that are specified.

3.2 Taxonomy

The taxonomy suggested in (Togelius, Yannakakis, Stanley, & Browne, 2010) can be used to classify this framework as follows:

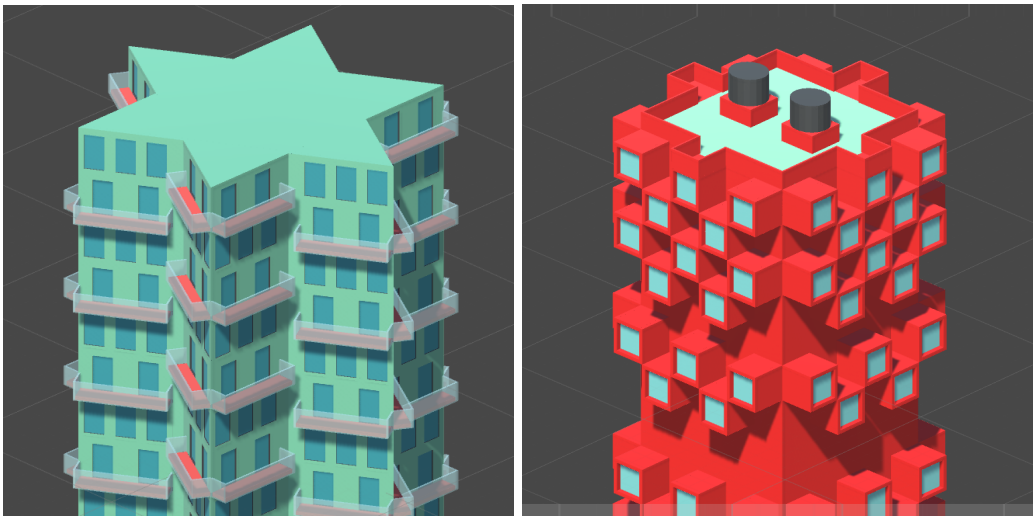


Figure 2: Buildings generated with the framework. The colors were added manually for better visualization. The left one was inspired by a real building in Copenhagen, the one on the right is a partial replica of a building from Tokyo 42.

- At this point, generating large amounts of buildings like the ones shown in Figure 1 is not fast enough to be done in real time, so this framework is mostly on the **offline** side.
- Generative grammars afford a high degree of control over the content they produce, so the framework can be used to produce **necessary** content i.e. of the kind that needs to be correct in some sense.
- Since the user of the framework has to define the specific grammatical rules that produce the content and their priority, the amount of **control** over the type of results given to the user is very high. In terms of reliability.
- The system as it is currently is **generic**, since it doesn't support adaptive generation, although there is no reason why another system couldn't be used in combination with this one to allow this.
- The grammar uses **stochastic** rules and parameters, although minimal changes are required to also allow recreation of content by saving the random generator seed. It is important to note that the use of the stochastic features of the system is entirely optional, so this system can be used to in a completely **deterministic** way as well.
- Grammars on their own are **constructive** by default and our system does not change that.
- The framework itself could be seen as a **mixed authorship** system since it doesn't generate content on it's own and instead provides a syntax for rule creation that the user is expected to use to create a rule set. A specific rule set in this framework would, however constitute an automatic generation system together with the framework.

4 Results

Figures 2 and 3 show buildings generated using the framework. They generated with tailor-made rule sets that can only produce those buildings and are meant as a demonstration of the expressive power of the system.

On the other hand, to demonstrate that the system allows the creation of single grammars that produce big languages (used in the sense explained in (Stiny, 1980)) i.e. that generate a large amount of different buildings, we present the rule set shown in Listing 1, which can generate over 16 million different proto-buildings, to which details could be added in a second stage with a

[1]This estimation is done counting each instance of random values as producing only 2 versions. The high numbers are the result of the stochastic rules and the ways they interact with each other.

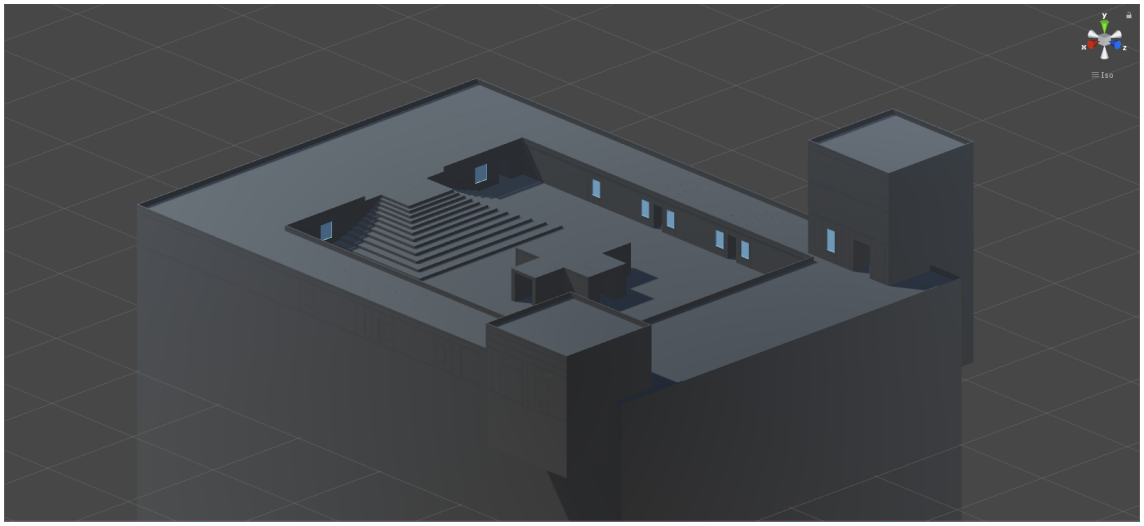


Figure 3: Another building generated with the framework. The colors were added manually for better visualization. It is a partial replica of a building from Tokyo 42.

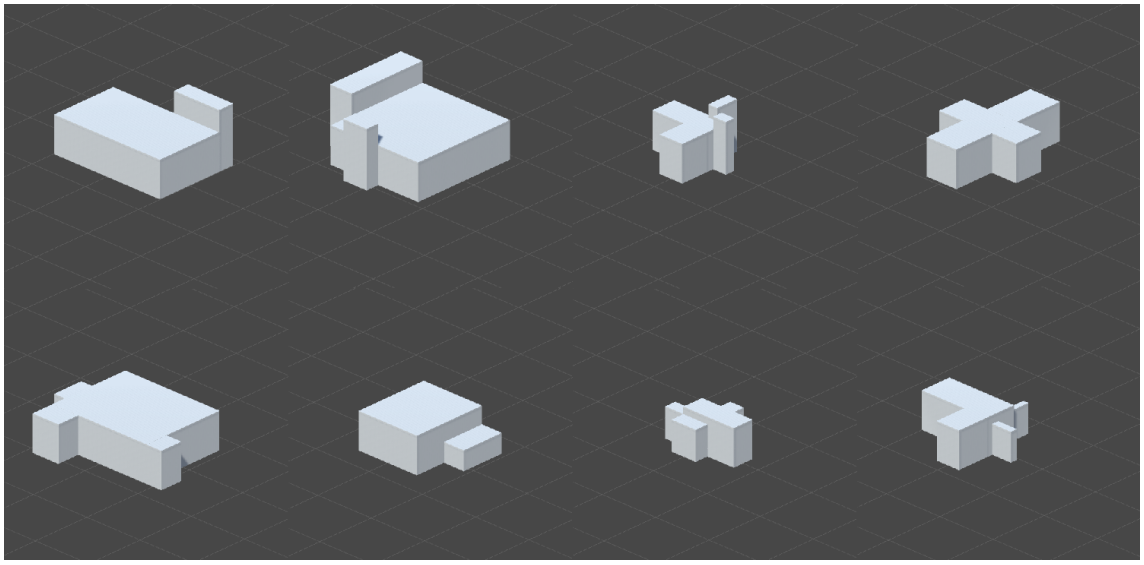


Figure 4: 8 proto-buildings or building skeletons generated with a grammar for the framework.

different grammar using the same symbols or by expanding the original rules. Figure 4 shows some of the building skeletons that this grammar can generate.

Some performance tests were conducted to measure how long the examples shown here take to be generated with the grammars we created for our system. The tests were conducted on an MSI GE62 (Intel i7 6700HQ CPU @2.60GHz, 8GB RAM and GeForce GTX 965M), a mid to high end gaming laptop. These results can be seen in Table 1.

```
1 AXIOM -> Transform()S(3,12,3)I(MASS, cube, outer) : 0.1 | Transform()S(Random(3,6)
      ,12,Random(3,6))I(MASS, cube, outer)I(PROTO_MASS_EXTRACTION, cube, outer) : 0.3
       | Transform(Scale=Random(3,6))S(Scale,12,Scale)I(MASS, cube, outer)I(
      PROTO_MASS_EXTRACTION, cube, outer) : 0.3 | Transform()S(Random(10, 20),12,
      Random(10, 20))I(MASS, cube, outer)I(BIG_BUILDING_EXTRACTION, cube, outer) :
      0.3
2 MASS -> Subdiv(y, 1r,1r){I(empty_space, empty, outer), I(mass, cube, outer)}
3 PROTO_MASS_EXTRACTION -> Subdiv(y, 1r,1r){I(empty_space, empty, outer), I(
      MASS_EXTRACTION, cube, outer)}
4 BIG_BUILDING_EXTRACTION -> Subdiv(y, 1r,1r){I(empty_space, empty, outer), I(
      BIG_EXTRACTION, cube, outer)}
5 BIG_EXTRACTION -> Comp(!rename, side_faces){BIG_EXTRACTED_F}
6 BIG_EXTRACTED_F -> Transform()I(extracted_face, quad, outer) : 0.5 | Subdiv(y,
      Random(3.0, 10.0)){I(PROTO_OFFSET_MASS, cube, outer)} : 0.5
```

```
7 MASS_EXTRACTION -> Comp(!rename, side_faces){EXTRACTED_F}
8 EXTRACTED_F -> Transform()S(1.0,1.0,Random(6.0, 18.0))I(PROTO_NEXT_MASS, cube,
      outer) : 0.6 | Transform()I(extracted_face, quad, outer) : 0.1 | Subdiv(y,
      Random(3.0, 10.0)){I(PROTO_OFFSET, cube, outer)} : 0.3
9 PROTO_NEXT_MASS -> Subdiv(z, 1r,1r){I(NEXT_MASS, cube, outer),I(empty_space, empty,
       outer)}
10 PROTO_OFFSET_MASS -> Transform()S(1.0,1.0,Random(3.0,9.0))I(INNER_OFFSET_MASS, cube
      , outer)
11 PROTO_OFFSET -> Transform()S(1.0,1.0,Random(3.0,9.0))I(OFFSET_MASS, cube, outer) :
      0.5 | Transform()S(1.0,1.0,Random(3.0,9.0))I(INNER_OFFSET_MASS, cube, outer) :
      0.5
12 OFFSET_MASS -> Subdiv(z, 1r,1r){I(mass, cube, outer),I(empty_space, empty, outer)}
      : 0.7 | Subdiv(z, Random(0.2,0.7)r,1r,1r){I(empty_space, empty, outer),I(mass,
      cube, outer),I(empty_space, empty, outer)} : 0.3
13 INNER_OFFSET_MASS -> Subdiv(x, Random(0.0, 1)r,Random(0.3, 1)r){I(empty_space,
      empty, outer),I(OFFSET_MASS, cube, outer)} : 0.33 | Subdiv(x, Random(0.3, 1)r,
      Random(0.0, 1)r){I(OFFSET_MASS, cube, outer),I(empty_space, empty, outer)} :
      0.34 | Subdiv(x, Random(0.0, 1.0)r,Random(0.3, 1.0)r,Random(0.3, 1.0)r){I(
      empty_space, empty, outer),I(OFFSET_MASS, cube, outer),I(empty_space, empty,
      outer)} : 0.33
14 NEXT_MASS -> Transform()I(mass, cube, outer)
```

Listing 1: These rules can generate 16 million different proto-buildings like the ones in Figure 4.

	Grammar Loading	Rewriting	Interpreting
Figure 2 left	13 ms	2561 ms	176 ms
Figure 2 right	54 ms	888 ms	44 ms
Figure 3	138 ms	285 ms	16 ms
Figure 4	<15 ms	<50 ms	<1 ms

Table 1: The time it takes to generate each of the examples.

5 Discussion

5.1 Properties

A list of desirable properties of PCG solutions can be found in (Shaker et al., 2016). We will use these to analyze the strengths and shortcomings of our tool.

5.1.1 Speed

The execution times shown in Table 1 mean that it is certainly fast enough for any offline generation (e.g. allows fast iteration when developing a rule set for it), but it might be too slow for some online generation cases.

5.1.2 Reliability

Given the nature of the tool and of grammars, it is mostly up to the user to define a set of rules that guarantees that the produced results meet any requirements. There is a caveat, of course: high control comes at the cost of either reducing the variety of content that a rule set produces or increasing the difficulty of creating the rules.

5.1.3 Controllability

Similarly to reliability, because the user defines the rules that ultimately determine the features of the content that is produced, there control of the user is potentially absolute with this approach, albeit limited to the existing rule types.

The bigger disadvantage with this framework as a PCG tool is that although it empowers the user, it relies heavily on human input to produce satisfactory results. To add insult to injury, creating rules for the system is not trivial and requires some learning of the syntax and the intricacies of each rule type.

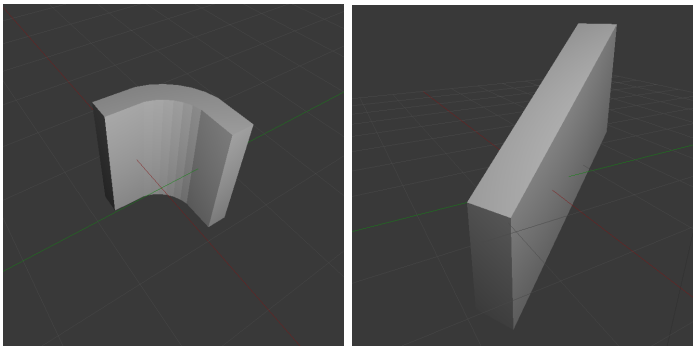


Figure 5: These shapes are almost impossible to create in at this stage.

5.1.4 Expressivity and Diversity

We understand this label to refer to two qualities: expressive power (i.e. what can you encode with the system?) and variety (i.e. how many different buildings can you produce with a single - and reasonably compact - rule set?, what is the content space size?). Figures 2 and 3 showcase the expressive power of the framework, while the proto-building rule set shown in Listing 1 demonstrates how many different buildings you can produce with one set of rules.

The framework in its current state does have some limitations:

- It only defines geometry, in other words, shading and textures can't be defined with any of the current rule types.
- In general, creating highly irregular structures is hard, particularly when a high degree of control is needed.
- Shapes like the ones in Figure 5 are practically impossible to create.
- The best way to control number of steps in a staircase for example is to change the number of rewriting passes, which is certainly not ideal.

5.1.5 Creativity and Believability

Once again, the level of control afforded to the rule creator means that it will be mostly up to her to make the results look human-made. The system allows emergence through the interaction of rules, although this, of course, requires a very careful design to maintain control.

5.2 Future Work

There are a lot of options available to improve this tool, some of the most interesting from our point of view are listed below.

- An extension of existing rule types or an addition of a new rule type that allows the user to define shading and textures for the different parts of the buildings.
- The system described in (Müller et al., 2006) supported conditional rules; rules that allow the specification of conditions that restrict when the rule can be applied. This could facilitate the construction of structures like staircases (they require rules that work in a recursive way and this could allow the definition of stopping conditions) and in general would improve the controllability and reliability of the tool. In the same vein.
- The creation of irregular structures could be facilitated by using 3D cellular automata or 3D noise functions such as fractals could be used as probability distributions to determine whether a rule should be applied at a certain position in space, instead of the space independent uniform distribution that is currently used. This would be a significant deviation from (Müller et al., 2006) and could expand the expressiveness of the system greatly. This was one of the ideas for the project from the beginning, but it has not been implemented yet.

- Adding rule types that can deform shapes by manipulating vertices, edges or faces would make Figure 5 and virtually every other shape attainable.
- Optimization of the framework could make it more viable for online PCG applications.

References

Blizzard Entertainment. (2012). *Diablo 3.* Retrieved from http://us.battle.net/d3/en/

Gearbox Software. (2009). *Borderlands.* 2K Games. Retrieved from https://borderlandsthegame.com/

Grinding Gear Games. (2013). *Path of Exile.* Grinding Gear Games. Retrieved from https://www.pathofexile.com/

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*(2), 127–145. doi: 10.1007/BF01692511

Krawczyk, R. J. (2002). Architectural Interpretation of Cellular Automata. *Generative Art.*

Mojang. (2011). *Minecraft.* Microsoft. Retrieved from https://minecraft.net/en/

Mossmouth, L. (2008). *Spelunky.* Author. Retrieved from http://www.spelunkyworld.com/

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural Modeling of Buildings. In *Proceedings of siggraph.*

Ruiz-Montiel, M., Belmonte, M. V., Boned, J., Mandow, L., Millán, E., Badillo, A. R., & Pérez-De-La-Cruz, J. L. (2014). Layered shape grammars. *CAD Computer Aided Design*. doi: 10.1016/j.cad.2014.06.012

Shaker, N., Togelius, J., & Nelson, M. J. (2016). PCG Book - Introduction. In *Procedural content generation in games: A textbook and an overview of current research* (pp. 1–15). Springer.

SMAC Games. (2017). *Tokyo 42.* Mode 7. Retrieved from http://store.steampowered.com/app/490450/

Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B*, *7*, 343–351. Retrieved from http://home.fa.ulisboa.pt/{~{}}miarq4p5/2011-12/Course{_}Programs/0{_}Projecto{%}20-{%}20Arch{%}20Design{%}20IV/1{_}Turma{%}20A/Shape{_}Grammars/1{_}Fundamentals{%}20of{%}20shape{%}20grammars/Stiny{_}IntroShapeAndShapeGrammars,.pdf

Togelius, J., Shaker, N., & Dormans, J. (2016). Grammars and L-systems with applications to vegetation and levels. In *Procedural content generation in games: A textbook and an overview of current research* (pp. 73–98). Springer.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2010). Search-based Procedural Content Generation. *Applications of Evolutionary Computation*, 141–150.

Unity Technologies. (2005). *Unity.* Retrieved from https://unity3d.com/

Veenstra, F. (2016). *L-Systems and Grammars.* Copenhagen: IT University.