



Universiteit  
Leiden

Modern Game AI Algorithms 2023, Assignment 1

## Procedural Content Generation

**Tom Stein**

February-March 2023

# Abstract

## 1 Introduction

The following report is structured into independent sections, containing the method and results for the respective component. First, a method for the building placement is described, results are shown and limitations are discussed. The next section focuses on the building generation using the wave function collapse (WFC) algorithm. Third, a method to add interior decoration is shown. Fourth, results for the combination of building placement, building generation and interior decoration are shown. Finally, the report concludes with a discussion of the results, the limitations of this approach and directions for improvement.

## 2 Building Placement

## 3 Building Generation

This section describes the generation of buildings using the wave function collapse (WFC) algorithm.

### 3.1 Structure Building Blocks

In order to use the WFC algorithm a set of small structures, sometimes called tiles or prefabs, is required to combine them to a larger structure, i.e. a building. These structures can in general be of any granularity from single blocks to larger groups such as walls, rooms or even whole corridors. However, there is a trade-off between variation and believability. There are more combinations of smaller structures in a fixed area than there are combinations of larger structures but many of these combinations of smaller structure combinations result in unrealistic buildings, e.g. five doors placed next to each other. Therefore, building structures with a size of  $11 \times 6 \times 11$  and  $11 \times 9 \times 11$  blocks were used, where each represents a single room of a building. This size gives enough possible variations in the typical size of a Minecraft house while believability, accessibility and architectural style. A selection of the designed structures can be seen in [Figure 1](#).

In order to duplicate these structures, and apply translation and rotation onto them, they need to be constructed using a computer program. It was chosen to follow a scanner-builder approach over manually translating these structures into code that constructs them using simple geometry like cuboids because the scanner-builder approach is easier to implement and allows for faster iterations. The scanner<sup>1</sup> works by serializing the user defined build area to a Python pickle file. Afterwards, the builder<sup>2</sup> can be used to duplicate the scanned structure with arbitrary rotation and translation. Every structure was scanned from the bottom left corner in positive XZ direction, by standing on the golden blocks shown in [Figure 1a](#).

In total 14 building structures were designed, which are shown in figure [Figure 2](#). They are split between first floor ( $11 \times 6 \times 11$ ) and roof house ( $11 \times 9 \times 11$ ) elements. There are simple corner elements and middle wall elements to allow building  $n \times 2$  rectangular shaped buildings. Additionally, center elements like center and

---

<sup>1</sup>[structure\\_scanner.py](#)

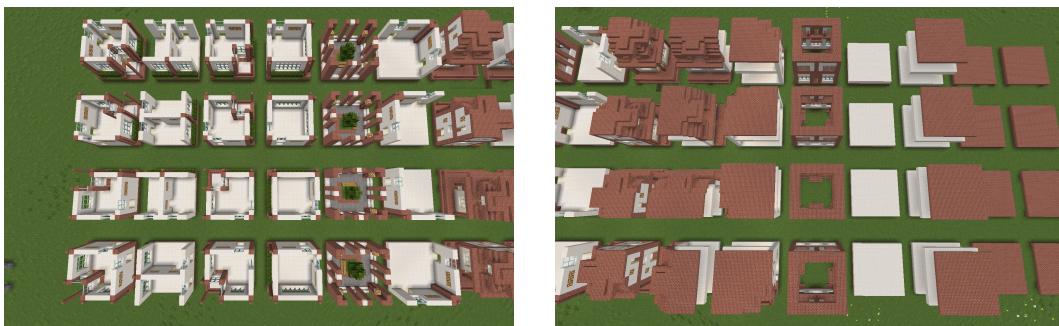
<sup>2</sup>[structure\\_builder.py](#)



(a) From left to right: building entrance corner, balcony and big window corner.  
(b) Doors between each building structure.

Figure 1: Some manually designed building structures which can be combined by placing them next to each other. Doors between them are aligned to ensure accessibility of the whole building.

courtyard were added to support  $n \times m$  shapes. Later, the concept of an inner corner was added to allow for more complex shapes like  $L$ ,  $O$  and  $+$ .



(a) From left to right: building entrance corner, middle wall, balcony corner, big window corner, courtyard and inner corner.  
(b) From left to right: corner roof house, middle wall roof house, inner corner roof house, courtyard roof house, center, center roof house, corner flat roof and big window flat roof.

Figure 2: All 14 designed building structures in all rotations. Each column represents a single structure and each of the four rows represent a rotation of that structure. The lowermost row is the baseline rotation 0.

With the introduced building structures one can already build whole houses by deterministically placing them next to each other, while making sure that all doors align as they are supposed to. Two examples are shown in Figure 3.

### 3.2 Wave Function Collapse

With the previously introduced set of structures it was only possible to generate deterministic buildings so far. Using the wave function collapse (WFC) algorithm it will be possible to generate randomized buildings which are still believable because they are only built out of valid combinations of structure building blocks.



(a) A  $3 \times 3$  house with courtyard and a second floor (roof house).

(b) A  $2 \times 2$  house with a flat roof.

Figure 3: Two deterministically built buildings.

### 3.2.1 Rule Set

To achieve this, it is necessary to define the adjacency rules for each structure. These define how structures can be placed next to each other, whereby rotations of the structures must also be taken into account. For the 14 structures with six sides and four rotations (rotation around the Y-axis only), this would mean that  $14 \cdot 6 \cdot 4 = 336$  rotated sides would have to be considered. For each of them one would have to define whether they may be placed next to each other, i.e.  $\frac{336(336-1)}{2} = 56280$  rules. Obviously, this is not very practical to define so many rules manually. Therefore, one can make use of the symmetries of the structures and define the adjacency rules only for one rotation of the structure. The others can be derived from this. In addition, one can only define the rules that are allowed, and assume for all rules that are not defined that they are not allowed.

The total of 320 rules<sup>3</sup>, that were necessary for the 14 structures, were realized directly in Python. A small excerpt from this is shown in Listing 1. Direct relationships between the objects were specified directly, i.e. if A is allowed next to B, there must also be a rule that B is allowed to be next to A. This explicit symmetry makes it easier to create the rules, as errors can be avoided in this way. Additionally, a consistency check `check_symmetry` for the rule set was implemented along with some other unit tests because the process of defining the rule set and the implementation to drive the implicit rules are both nontrivial.

## 4 Interior Design

## 5 Overall Results

## 6 Discussion

---

<sup>3</sup>`structure_adjacency.py`

```

1  structure_adjecencies = {
2      brickhouse_middle: StructureAdjacency(
3          structure_name=brickhouse_middle,
4          x_plus=[
5              StructureRotation(brickhouse_middle, 0),
6          ],
7          x_minus=[
8              StructureRotation(brickhouse_middle, 0),
9          ],
10         z_plus=[
11             StructureRotation(brickhouse_middle, 2),
12             *all_rotations(brickhouse_center),
13         ],
14     ),
15     brickhouse_center: StructureAdjacency(
16         structure_name=brickhouse_center,
17         x_plus=[
18             *all_rotations(brickhouse_center),
19             StructureRotation(brickhouse_middle, 1),
20         ],
21         x_minus=[
22             *all_rotations(brickhouse_center),
23             StructureRotation(brickhouse_middle, 3),
24         ],
25         z_plus=[
26             *all_rotations(brickhouse_center),
27             StructureRotation(brickhouse_middle, 2),
28         ],
29         z_minus=[
30             *all_rotations(brickhouse_center),
31             StructureRotation(brickhouse_middle, 0),
32         ],
33     ),
34 }

```

Listing 1: A (very) short excerpt from the adjacency rules limited to the middle wall and center structure. All rules are defined from the respective element's perspective in rotation 0 (lowermost row in Figure 2).

## References