Methods for recognizing dimensions of rectangular panels and laying panels on planar surfaces

1 Introduction

The problem originated from the task of generating rectangular tile layout for KSA Pavilion. It involves two parts: 1. For convenience of users and recycling process, an algorithm is needed to recognize the dimensions of given tiles (Figure 1). 2. Given surfaces and numeric desciption of panel options, fill surfaces with panels in a random way with optional gradient effect without leaving gaps. (Figure 2).

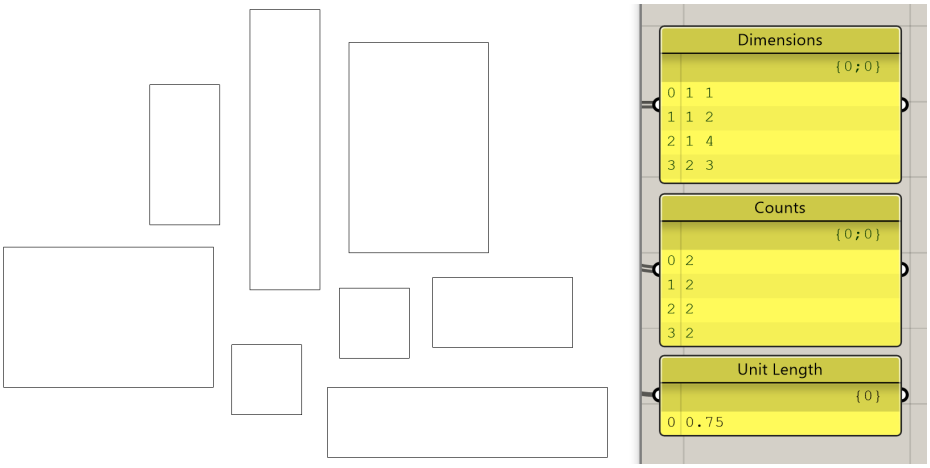

Figure 1: Example result of dimension recognition

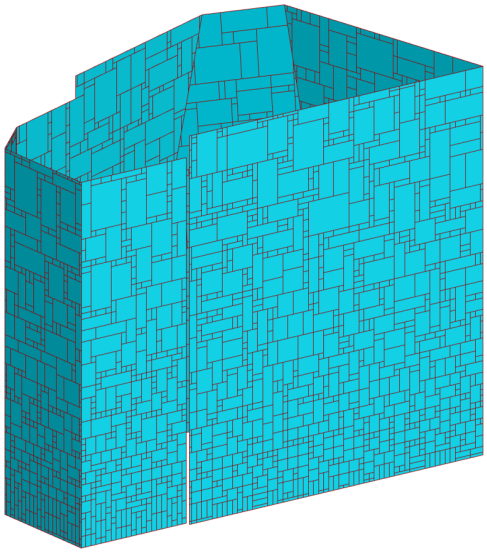

Figure 2: Example of gradient "Orthogonal" tiling of KSA Pavilion

The actual implementation of the algorithms assumes all input and output to be on XY plane aligning with axes in convenience of future application in different scenarios. Please see the diagram showing how they are involved in KSA Pavilion (Figure 3).

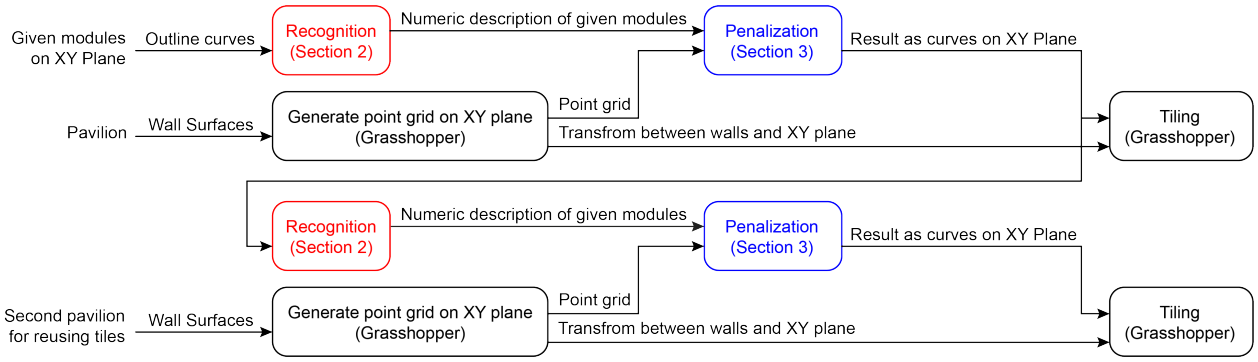

Figure 3: Diagram of components used in KSA Pavilion project

2 Recognition

The unit length of the tiles is the greatest common divisor (GCD) of their widths and heights. The only extra part to pay attention is to deal with non-integer dimensions.

2.1 Steps

Denoting the collection of bounding boxes of each panel as B.

1. Extract the set of lengths from bounding boxes' width and length.
2. Scale all numbers by 10^n if they are decimal numbers accurate to n decimal places.
3. Calculate GCD of all numbers. Pseudo code:

```
result = arr[0]
for i in range(len(arr)):
    result = gcd(result, arr[i])
```

Note: GCD of two numbers can be calculate with Euclidean algorithm. Pseudo code:

```
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

4. Divide all dimensions by GCD to get normalized dimensions.
5. Scale GCD by $1/10^n$ to get the unit length.

2.2 Note

Since dimensions are extracted by `BoundingBox.Max - BoundingBox.Min`, There's chance for this to fail due to accuracy of floating point representation, for example getting 7499 instead of 7500 for scaled dimensions.

3 Penalization

The main idea of this algorithm is to use 2D array for placing panels, instead of checking whether the panel fits a place through geometrically testing intersections. It is convenient whether the user would like to feed in Rhino curves or values to describe the panel options.

A queue of positions in a pregenerated random order is used instead of randomly picking positions ensures all positions are filled. If a panel doesn't fit a position, it will never fit later. Therefore, for each tile option, iterating through all available positions once is sufficient, and positions not fitting can be put to the end of the queue for smaller options.

3.1 Steps

Preprocess: Convert given surfaces to point grid aligning with XY axes with same unit length as panels. This is achieved by Grasshopper. Besides, add infinite number of 1x1 tiles to ensure all gaps to be filled.

1. Convert given point grid to 2D array, where boolean value `map[i][j]` denotes whether position i, j has been occupied.

2. Create a queue and enqueue all available positions in a random order.

Note 1: A random permutation of a sequence can be generated in $\mathcal{O}(n)$. Pseudo code:

```
for i in range(n - 2):
    j = i + random_integer(n - i)
    swap(arr[i], arr[j])
```

where `random_integer(n)` randomly pick an integer $0 \leq x < n$.

Note 2: In KSA Pavilion project, the extra step to achieve a gradient effect is to generate the random permutation non-uniformly with a custom `random_integer` method. Pseudo code:

```
def random_integer(n, grad)
    pick = random() * n ** grad / (grad + 1)
    return int(n - (pick * (grad + 1)) ** (1 / (grad + 1)))
```

where parameter `grad` controls how much gradient effect and `random()` randomly pick a floating number between 0 and 1. Note that using integer for `pick` will result in unsmooth distribution.

3. From largest to smallest, attempt to lay all panels to the 2D space. Pseudo code:

```
for option in options:
    i = 0
    position_count = size(queue)
    while i < position_count and option.remaining > 0:
        position = queue.Dequeue()
        if map[position.x][position.y] and fit(option, position):
            option.remaining--
            construct outline curve
            update map
        else:
            queue.Enqueue(position)
        i++
```