**Texture Packing Algorithm**

This algorithm solves the following problem:

* We have an ordered list of N rectangles each having its own size.
* We have the rectangle that is the destination of the packing. This rectangle has maximum bounds.
* How we can put these N rectangles inside this big rectangle using the minimum space available?

**Definitions**

* *rectList*: This is the rectangles to be packed. This should be a linked list.
* *maxAtlasSize*: This is the maximum size of the atlas.

**Table**

We’re going to need a data structure that works like a table. The following properties applies:

* The table does not need to be square: have equal number of rows and columns.
* Each cell in the same row has equal height.
* Each cell in the same column has equal width.

The above properties are invariant. We need a data structure that has the following additional performance properties.

* We can reach adjacent cells easily.
* We can easily add new rows and columns.
* We don’t need random access to a cell in this table.

We’re going to use a data structure were each cell has four pointers:

* The pointer to its left adjacent cell.
* The pointer to its right adjacent cell.
* The pointer to its upper adjacent cell.
* The pointer to its lower adjacent cell.

The table does not need to wrap. We can take advantage of the fact that the table always grows, so no cell will ever be deleted. Each cell has the following additional properties:

* Width of the cell.
* Height of the cell.
* Used flag: If the cell is empty or not.
* Texture: If the cell is used, which texture it has. A texture can be spanned among multiple cells.

The following categorizations applies:

* Neighbor: include all eight possible neighbors to a cell.
* Boundary cell: A cell whose at least one of the four pointers is null.
* Front cell: Cell whose one of its neighbors is used but cell itself isn’t.
* Corner Front cell: A front cell that has its right and down cell free. Its left cell and upper cell must be filled or the cell must be a boundary in those directions.

**Front Cell List**

The algorithm maintains a list of front cells and corner front cells.

**Operations on the table**

We can do one of two possible operations on a table:

* Subdivide row: A row gets split in two parts.
* Subdivide column: A column gets split in two parts.

Wen we subdivide, we must preserve the following properties:

* Front cells continue to be front cells.
* Front corner cells continue to be front corner cells.
* Boundary cells continue to be boundary cells in all the directions they were before.

Note that a subdivision may introduce new boundary cells, but never add/remove front cells/front corner cells. But we don’t care about boundary cells.

Operations to subdivide:

1. Let’s say we want to subdivide a row or column. We start with a cell that is in the start of the row or collection and iterates by walking down or right (depends if we’re subdividing a row or column).
2. If the cell is a front cell, front corner cell, boundary cell:
   1. We preserve the cell and add one to the left or down.
3. If the cell is a filled cell:
   1. Just split the texture between the two cells and set them as filled.
4. For all cells:
   1. Just split the cell.

**Important Property: Upper Left packing**

This property means that if a cell is free, its right and down neighbor will always be free or a boundary. Its important to always preserve this table property.

**Padding**

Padding is a used cell but not associated with any texture. Its used to preserve upper left packing.

**Filling/Unfilling a cell**

Subdividing a cell never introduces new front cells but filling a cell introduces/removes front cells. These rules apply:

* We only fill empty cells and the root filled cell (the upper leftmost cell to be filled) must be a front corner cell.

We must check if the area is available. The test is simple:

* From the root cell, keep iterating to the right until the sum of the width is equal or more than the width of the rectangle.
* From the cell you stopped, iterate down until the sum of the height is equal or more that the height of the rectangle.

After you have found your area, do the following:

* Subdivide the last row and column of the area so the area becomes an exact match.
* Start filling the rectangles with the texture. For each cell filled, do the following:
  + If the cell is a front corner cell:
    - Remove the cell as front corner cell and mark the adjacent cells as front corner cell or front cell if that applies (right and bottom).
    - You must always fill a front corner cell.
    - You must fill from left to right, from up to bottom.
    - If at some point we can’t fill anymore because there is no more front cell, or we have reached a boundary, we can do the following:
      * Unfill the filled cells in reverse order. For each unfilled cell:
        + Mark the cell as front corner cell. Unmark possible adjacent (right and bottom) front corner cells (because front corner cells are never adjacent to front corner cells).
        + We should also check if left cells and upper cells are padding when we unfill. Because if they are, we must uncheck all the left and upper cells as padding and revert them to unfilled state.
      * Mark the cell we can’t fill as front corner cell and mark all the left or upper cells adjacent to that as filled (until we find a filled cell) and also mark as not having texture (padding).
* If you fail to fill the entire rectangle, you may need to merge columns and cells:
  + The leftmost or topmost cell is preserved [which may be the front corner cell].
  + The merges must be done in the reverse order they were made.

**Running the algorithm**

1. Sort the list of rectangles to be packed from largest to tiniest. Sort by area.
2. Start trying to pack rectangles one by one. You may try to pack each rectangle in multiple passes.
3. In the first pass you try to pack the rectangle passing a maximum padding of 1% of max width/height for width or height. That means the padding rectangle may have at maximum 1% of max width/height at its width or height.
4. If the rectangle can’t be packet, its rotate 90 degree and tried again.
5. If the rectangle can’t still be packed (its unrotated and) it goes to the end of the list and the pass on it is increased.
6. The pass may be cleared when a new rectangle is packed.
7. If you try to pack a rectangle with a pass of one, you increase the padding to be of any size and try it normally or 90 degrees rotated.
8. If it can’t be rotated, its pass is increased and it goes to the end of the list again.
9. The algorithm stops when the unpacked list is equal (in size) to the list of triangles to be packed. Remember that this list is cleared every time a new triangle is packed.
10. If the unpacked list is cleared, then all rectangles were packed successfully.