Multidimensional vector index

By Orlando José Luque Moraira. May 8 of 2019

Overview

The kind of problem I'm trying to tackle with this algorithm is the following: Imagine we have a map, with many items as shops, points of interest, etc. We have to calculate which ones of those items are up to 5 kilometers from our position. This is a classical computational problem, as the higher the items count, the slower we will obtain the result.

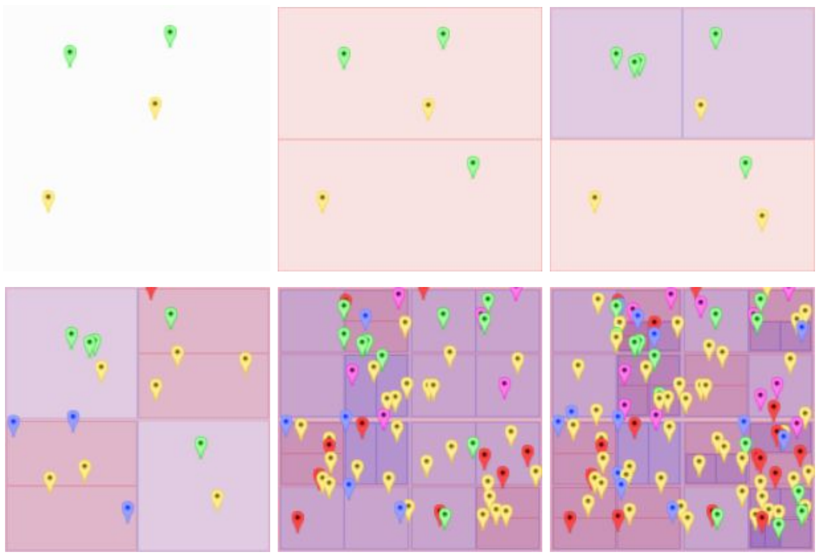
There are already many, worse or better, ways to try to solve this. The method presented in this paper is focused in speed. As a trade-off, it uses additional memory in order to achieve a greater speed.

While this document is focused in a 2D map and items which are 'points' (they have no area), you can use this idea with any number of dimensions. You could have a three-dimensional space and process items handled as areas or volumes if needed.

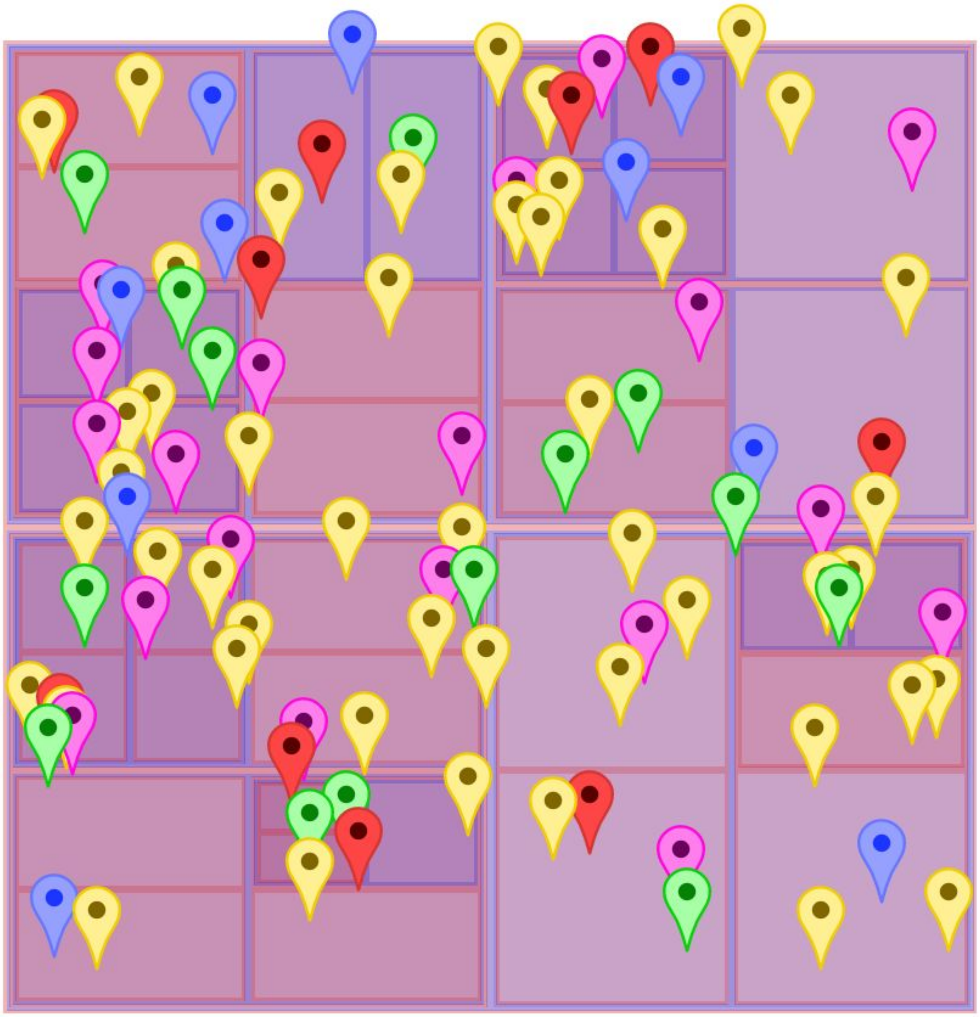
Strategy

To put it shortly the strategy I am following in this document is:

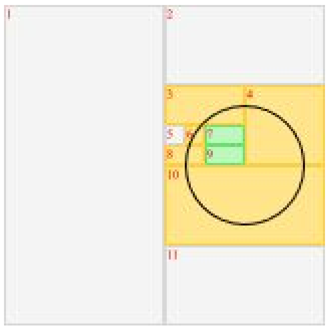
1. We prepare in advance a set of templates and precalculated data. I will describe this later.
2. We have a empty map,which is a grid with a single cell, and we add the items into it:
    a. The items are progressively added into the map cell
    b. When there are too many items in the map we divide it in half, creating smaller cells which occupies the same space the original did and splitting up the items between the new cells, based on the spatial location of each one
    c. Now we have two cells, each one with a shorter item list than the original one
    d. We continue adding new items, each one in the corresponding cell based in its spatial location
    e. When there are too many items in one of those cells we divide it further again
    f. And so on!



3. After adding all the items, the map has been divided into many cells, but none of them contains many items



4. We want to know the items closer than 5Km away of our position on the map, so we use the templates and pre-calculations to classify the cells generating the next two sets:



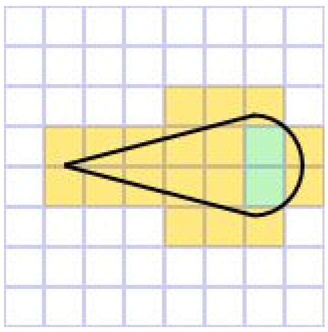
    a. cells completely inside the 5 Km radius area (green in picture)
    b. cells partially inside the 5 Km radius area (yellow in picture) whose items could be inside or outside the area
5. We disregard the cells completely outside the 5 Km radius area (gray in picture).
6. Finally we just use the two list described above (4.a and 4.b) to generate the result set. We must follow the next logic:

   a. For each of the cells in the 4.a list (completely inside) we add all their items to the result set.
   b. For each of the cells in the 4.b list (partially inside) we check if their items are inside the 5 Km radius area before adding it to the result set.
7. We are done! (until we need to add more items!)

Templates

The precalculated data and templates are specially tailored in order to meet your needs.

Most applications will always use the same shapes (e.g. circles, lines, etc) with a fixed set of cell sizes, so you will prepare the templates for those shapes and cell sizes and configurations and no others.



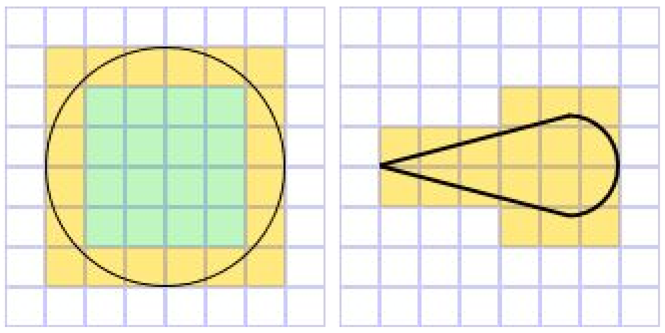
The templates are cell grids indicating cells inside, outside or partially inside a shape. Shapes as circles, lines, etc, and if you are working in a 3d space, lines, spheres and so on. Templates can be prepared for irregular shapes and volumes as well.

Make sure that, for every cell size and configuration, from the starting cell, you have to generate a template for each starting point of the shape in the cell. (See example below). Those are a big number of templates, but many of them will be exactly the same and there are computational tricks to reduce the memory consumption.

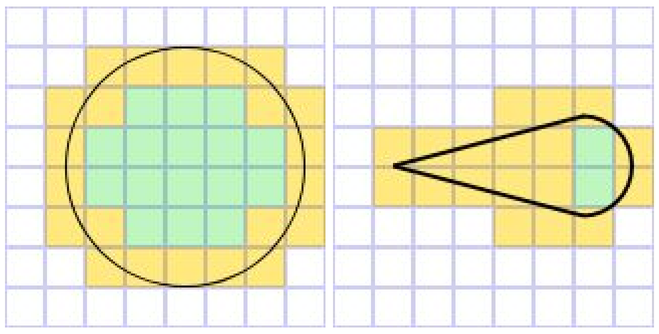
The cells of the next examples are coloured with the following logic:

- Green cells are completely inside the shape so the items contained are handled automatically as positive results. If those cells had been divided themselves their children will be handled the same way.
- Yellow cells are partially inside the shape, so their items have to be checked individually before handling them as positive or negative results. A yellow cell could be divided itself and some of their children could be white or green cells instead of yellow ones.
- White cells are completely outside the shape so their items are negative results so they are discarded.

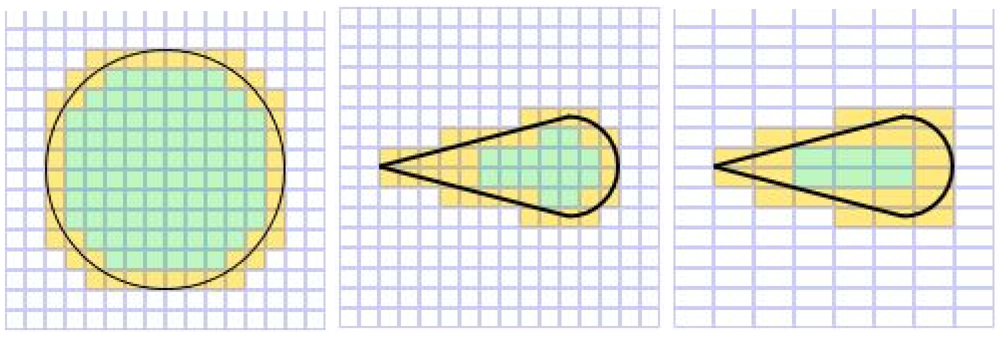
Si no son te vi en la tienda
So, as described we will generate templates for different shapes:



Different spatial displacements:



And different cell sizes and shapes:



Basically, we start having a cell grid consisting in a unique cell. Each time a cell is divided it will be replaced by a new grid, so we end with a grid containing grids as needed to compensate the population of items in each zone of the area or volume.

Of course, the initial map could be a grid with more cells (2 x 2, 4 x 4, ...) and instead of dividing the cells in half, you can divide it in cells of 2 x 2 or 4 x 4, etc.

Culling

From now onwards we will call 'culling' the already described process of obtaining the items inside the desired shape.

Ending

Do not be concerned about memory usage, in this document I have already detailed a nice list of procedures to reduce the memory usage. I.E. the 5 Km circle template for a determined cell size and shape can be used for a 10Km circle with double size map cells.

Implementation details

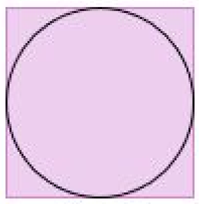
Following we have some pseudocode for a basic implementation. Feel free to play with different values and thresholds or modifications to some of the strategies. Later on I will describe some cases with an higher complexity.

The described pseudocode will be based on the next additional principles:

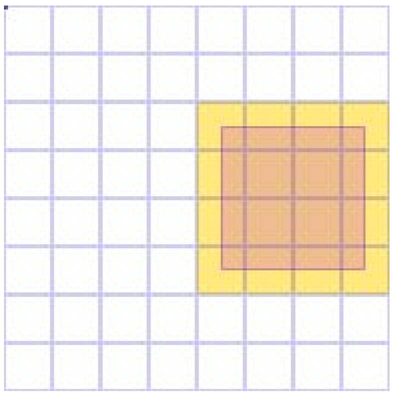
- Items: as this is a basic implementation, the items will be handled as points, so an item will be contained in a unique cell. Later on we will detail how to handle items as areas or volumes.
- Nodes:
    - A node is created to hold a cell.
    - If the holded cell is divided, a new node will be created to hold each one of the new parts while this node becomes it's parent, holding no cells itself thus keeping the tree structure.
    - Each node will hold the next data:
        - Items: when the node holds a cell, we keep here the cell's items.
        - Children: when the node does not hold a cell, we insert here the children nodes
        - Parent: the parent of every no root node.
        - Holding a cell or not each node represent an space, so:
            - X
            - Y
            - Height
            - Width
- Cells:
    - As detailed earlier, cells with too many items are divided into two smaller cells in horizontal or vertical way. To limit the number of cell shapes and so limit the number of required templates for this document I decided to,
        - if the cell to be divided is rectangular we divide it so the new cells are square,
        - else if the cell is square select the best division type so the items are are as evenly distributed throughout the new cells as possible.
    - Each cell is a node on a data tree. When a cell is divided the node is not removed from the tree but instead it is the parent of the two nodes of its new children.
    - No every data tree node have a related cell nor items as when a cell is divided into two smaller cells the original cell is replace with the new smaller

ones while its node is keeped as the parent of the new cells nodes, and the items are moved to the children as well.

- Bounding box:
    - It is the minimal box needed to surround the shape. It can be an square or a rectangle.

    - During the culling process, if we need to apply a template and there is no template available for such a cell proportions, shape and density configuration, we will use the bounding box instead. Basically every cell “touched” or inside the bounding box is handled as a cell of the ‘possible' or ‘partially covered’ type.

- Templates:
    - When we end calculating a template, if it entirely consists in cells of the partial type, we will discard such template and the culling algorithm will use the bounding box in its place.
- Constants:
    - ITEM_COUNT_LIMIT_PER_NODE: the maximum number of items inside a node before it is “divided”.
- Shape vectorial data: have everything needed to know about the shape-to-apply type and placement
    - Bounding box (with its own X, Y, width and height)
    - Shape type: as circle, “flame template”, square, ...
    - Center, radius / x1, y1, x2, y2 / ... / or any other specific data we need to use the current shape, including orientation if applicable
    - Fast as possible function to know if a “possible” item is actually inside the shape or not. It will be used on the items of cells “partially inside the shape”. The function parameters are shapeVectorialData, node and the item).

Cell division

As the cells are filled with too many items, they will divide so there is no cell with too many items inside. The way I am describing at the next pseudocode is just a nice way to do it, but not necessarily the best performing one, so we obtain more squared cells than rectangular ones. Feel free to do it in a different way or dividing into more cells, or replicate quadTrees, etc.

```
function divideCell(currentNode)
  1.  determine best division mode calling
      1.1.  if currentNode is wider than taller
            1.1.1.  the division type will be vertical (one child above the other)
      1.2.  else if currentNode is taller than wider
            1.2.1.  the division type will be vertical (one child above the other)
      1.3.  else (the cell is an square)
  2.  following the selected division strategy create 2 new nodes as children of
      currentNode and distribute the items of currentNode (property currentNode.items)
      between the two new nodes, so currentNode.items is now empty.
  3.  check if the items count of  any of the new nodes is greater than
      ITEM_COUNT_LIMIT_PER_NODE. It this case repeat this method calling
      divideCell(child)
```

Insert an item into the map

```
isOk = insertItemIntoArea(item, tree.root)

function insertItemIntoArea(newItem, currentNode)
  4.  if currentNode have not children
      4.1.  if newItem collide with any other of the items of currentNode.items
            4.1.1.  return false
      4.2.  add newItem to currentNode.items
      4.3.  if count (currentNode.items) > ITEM_COUNT_LIMIT_PER_NODE we have to
            divide the cell into smaller cells so we have to call divideCell(currentNode).
      4.4.  return true
  5.  else
      5.1.  determine which one of the children should contain the newItem comparing
            the newItem coordinates with the children vectorial data. That child is
            determinedNode.
      5.2.  return insertItemIntoArea(item, determinedNode)
```

Obtain the node containing a certain item

nodeContainingTheItem = locateItem(item, tree.root)

```
function locateItemNode(item, currentNode)
  1.    if currentNode have children
        1.1.    determine which one of the children should contain the item comparing the
                item coordinates with the children vectorial data. That child is
                determinedNode
        1.2.    return locateItemNode(item, determinedNode)
  2.    else
        2.1.    return currentNode;
```

Removing an item from the tree

```
function removeItemFromTheTree(item, treeRoot)
  1.    node = locateItemNode(item, treeRoot);
  2.    remove item from node.items
  3.    As long as the node is not the treeRoot,  and therefore it has a parent ,  if the sum of
        all the childrens item count is below ITEM_COUNT_LIMIT_PER_NODE
        3.1.    We move all those items to the parent
        3.2.    We remove those parent's children from the tree
        3.3.    node = node.parent
        3.4.    Go to 3
```

Move an item

```
function moveItem(item, newItemPosition, treeRoot)
  1.    //removeItemFromTheTree(item, treeRoot);
  2.    currentNode = locateItemNode(item, treeRoot);
  3.    if newItemPosition is inside node
        3.1.    return true
  4.    Else
        4.1.    remove item from currentNode.items
        4.2.    return insertItemIntoArea(item, treeRoot)
```

xxxxxxxxxx This coding is the simplest one. There is a variation consisting on following the tree while both current and target places share the same nodes.

Items culling

To cull items is to, using a shape positioned in space, determine the items inside it.
The method nature is recursive. The method parameter 'fullyInside' is just a boolean (true or false) to directly add the items it contains to the final result.

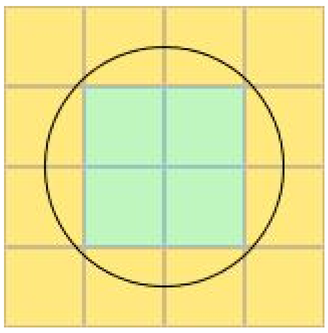
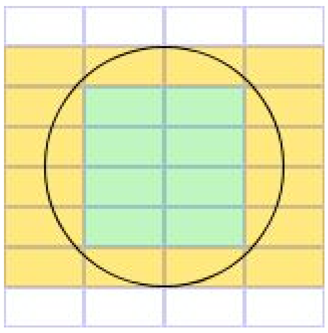
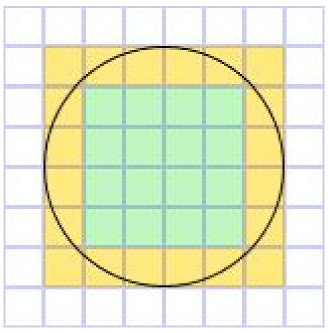
```
culledItems = getItemsInsideShape(shapeType, shapeVectorialData, treeRoot, false)

function getItemsInShape(shapeType, shapeVectorialData, currentNode, fullyInside)
  1.  if fullyInside
      1.1.  if currentNode have children
            1.1.1.  for each children we obtain the result of getItemsInShape(shapeType,
                    shapeVectorialData, child, true)
            1.1.2.  return the union of all the obtained results at 1.1.1
      1.2.  else (no children)
            1.2.1.  return currentNode.items
  2.  else if currentNode have children
      2.1.  if we have not an matching template
            2.1.1.  for each children in contact or inside the bounding box we obtain the
                    result of getItemsInShape(shapeType, shapeVectorialData, child,
                    false)
            2.1.2.  return union of all the obtained results at 2.1.1
      2.2.  else (children and template)
            2.2.1.  foreach children we check it against the template
                    2.2.1.1.  the related template cell is of the partial type we obtain the
                              result of getItemsInShape(shapeType, shapeVectorialData,
                              child, false)
                    2.2.1.2.  else if the related template cell is of the complete type we
                              obtain the result of getItemsInShape(shapeType,
                              shapeVectorialData, child, true)
            2.2.2.  return union of all the results obtained at 1.2.1.1 and 2.2.1.1
  3.  else (currentNode without children)
      3.1.  we initialize an empty list as the result list.
      3.2.  foreach item we call shapeVectorialData.checkFunction(shapeVectorialData,
            currentNode, item) so we know if the item is inside the shape or not. If it is, we
            add the item to the result list.
      3.3.  we return the list.
```

Culling example

Introduction

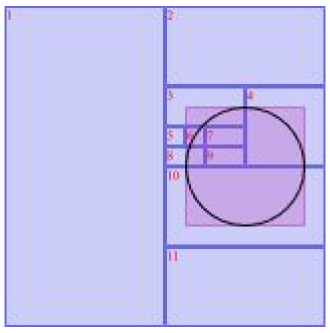
For the next example we will not shown any item on the grid and we will use the next templates:

Color legend:

- Green: fully inside so we take every item as part of the result
- Yellow: partially inside so we have to check every item inside before adding it to the result or not.
- White: we can discard every item inside.

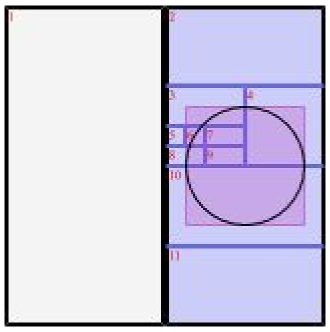
We have the grid, a circle as the shape already in place, and its bounding box as that reddish square. The blue cells have not yet been processed. A cell being processed and its child will be bordered in black.



- When a cell is discarded, it will be turned gray.
- Yellow cells are those cells whose items will be checked individually using the function we called 'shapeVectorialData.checkFunction' in the pseudocode.
- Green colored cells are the ones whose items are automatically included as part of the culling result.

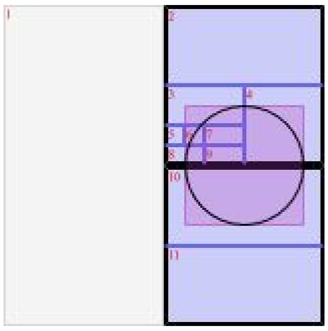
How the culling works, step by step

Step 1: We start with 'fullyInside' = false. We grab the first node of the three, the tree root. It has 2 children.

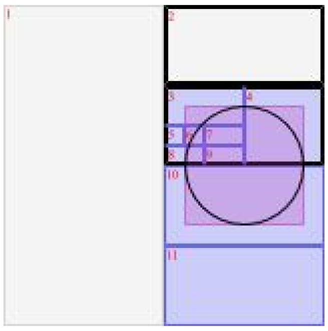


For this cell combination we have no template available, so we have to check the children versus the bounding box. As left node is not touching the bounding box, we discard it. Then we process the right with 'fullyInside' = false.

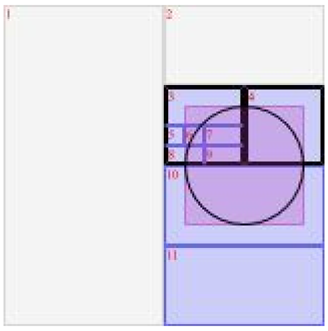
Step 2: Right side of step 1 node. We have not any corresponding template and both children are touching the bounding box, so we proceed to process both of them with ‘fullyInside’ = false.



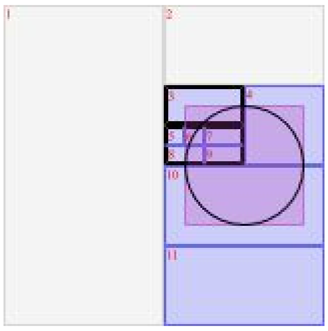
Step 3:Top side of step 2 node. No template. We discard top child as it is outside the bounding box. Now process bottom child with ‘fullyInside’ = false.



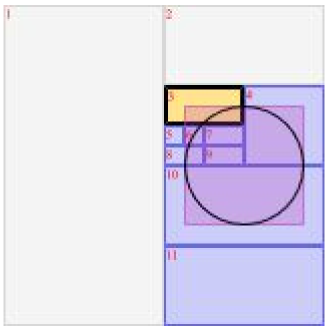
Step 4: Bottom side of step 2 node. No template and both children collide with the bounding box, so we process both of them with ‘fullyInside’ = false.



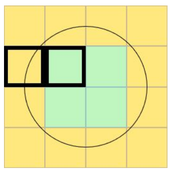
Step 5: Left side of step 4 node. No template and both children collide with the bounding box, so we process both of them with ‘fullyInside’ = false.

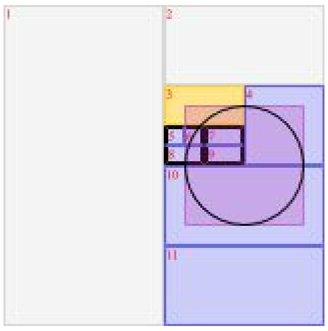


Step 6: Top side of step 5 node. No template and no children. An ‘fullyInside’ = false and the node has no children this is a ‘possible’ node, so we have to check each one of its items with ‘shapeVectorialData.checkFunction(...)’, create a list of the items confirmed to be inside the shape and return it as part of the result.

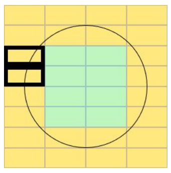


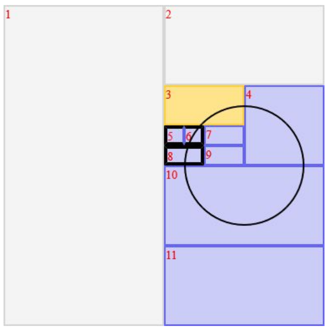
Step 7: Bottom side of step 5 node. This time we have our first template match. We observe that the left child is ‘partial’ and the right one a ‘complete’, so we process the left one with ‘fullyInside’ = false, and the right one with ‘fullyInside’ = true.



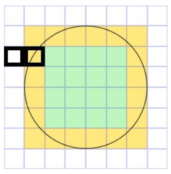


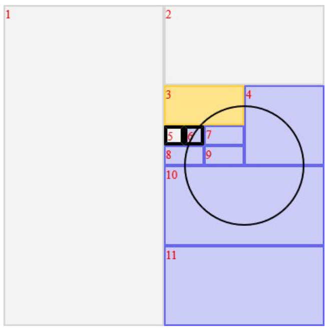
Step 8: Left side of step 7 node. Template match. Checking the template both children are ‘partial’ so we proceed to process them with ‘fullyInside’ = false.



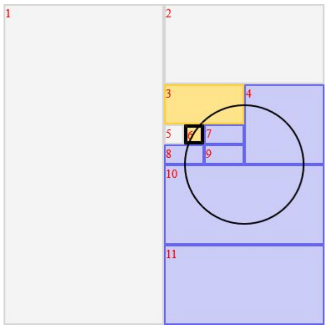


Step 9: Top side of step 8 node. Template match. Checking the template we discard the left child and we process the right one with ‘fullyInside’ = false.

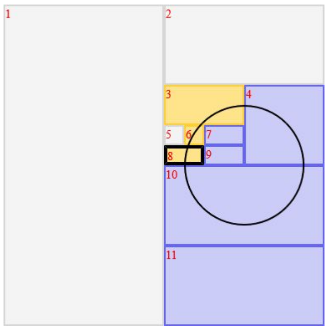




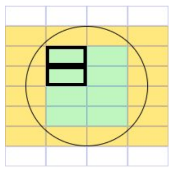
Step 10: Right side of step 9 node. This node have no children so there is no need of template matching. As there are no children and ‘fullyInside’ = false we have to check each one of its items with ‘shapeVectorialData.checkFunction(...)’, create a list of the items confirmed to be inside the shape and return it as part of the result.

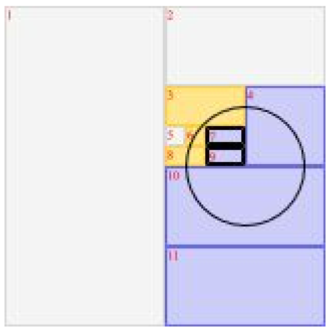


Step 11: Bottom side of step 8 node. As there are no children and ‘fullyInside’ = false we check each one of its items with ‘shapeVectorialData.checkFunction(...)’, create a list of the items confirmed to be inside the shape and return it as part of the result.

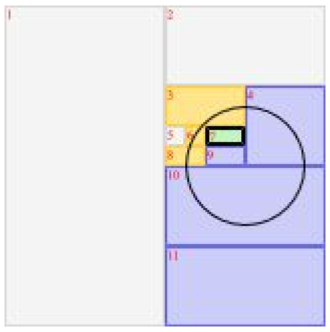


Step 12: Right side of step 7 node. Template match. Checking the template both children are 'complete' so we proceed to process them with 'fullyInside' = true.

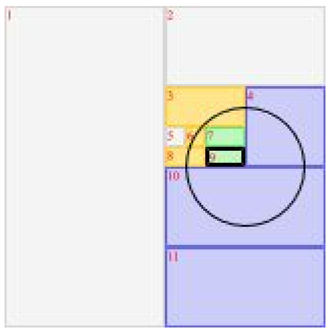




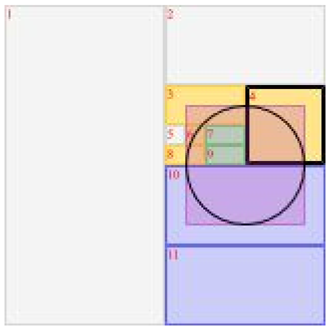
Step 13: Top side of step 12 node. As 'fullyInside' = true, we return the node item list as part of the result.



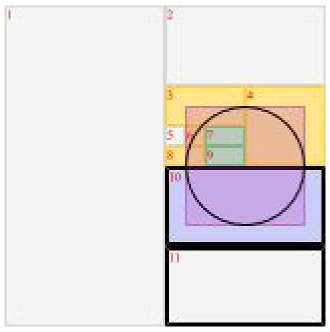
Step 14: Bottom side of step 12 node. As 'fullyInside' = true, we return the node item list as part of the result.



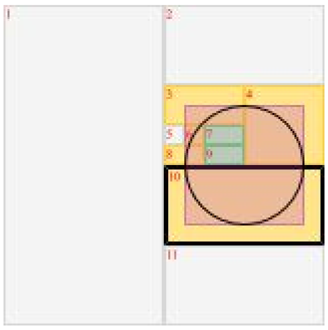
Step 15: Right side of step 4 node. No children and 'fullyInside' = false so we have to check each one of its items with 'shapeVectorialData.checkFunction(...)', create a list of the items confirmed to be inside the shape and return it as part of the result.



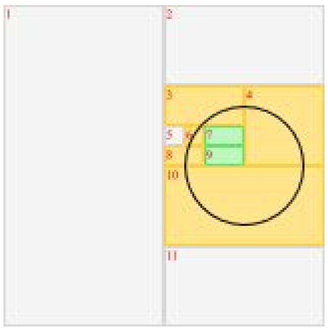
Step 16: Bottom side of step 2 node. No template, so we discard the out of the bounding box bottom child and we process the top child with 'fullyInside' = false.



Step 17: Top side of step 16 node. No children and 'fullyInside' = false so we have to check each one of its items with 'shapeVectorialData.checkFunction(...)', create a list of the items confirmed to be inside the shape and return it as part of the result.



Final step: return back recursively every result, joining together all results' lists and returning it as the final result of the culling.



For a shape with the size of 11% of the whole map we have:

- discarded the items of 75% of the map
- automatically collect as results an 1'5%
- only had to check the items of 23% of the map.

With smaller shapes the checked % will be lower.

Other considerations

Memory optimization

- Reusing templates as:
    - Some combinations of parameters and cell size have the same results. By example, for a certain shape and cell size, you can use the template of a shape half the size over cells half the size.
    - A lot of offsets combinations share the same results.
    - Symmetrics. With one template for a certain shapes, offsets and orientation (as a droplet shape template), you can reuse the template with every 90 degrees rotation of the case. The same can be applied to rectangled cell templates.

As a general procedure two or more cases with the same result should reference the same result on memory thus saving memory consumption.

- Combine templates for simpler shapes to cull bigger shapes. By example you can use a quarter circle set of templates and apply it 4 times to cull a circle. You can as well use the templates both as a whitelist (the default one) as a blacklist (so the items inside green cells are excluded) so you could use a bigger whitelist circle template with an inside smaller blacklist one so you cull a ring shape.
- Better control of the cell division shapes, as new shapes requires their own set of templates. That is the reason I recommended trying to get squares and half an square shaped rectangles.

- For certain cases, in place of templates, you can use other strategies. By example, create a circle template where in each cell you store the radius value ranges to consider the cell as “probably” or “completely inside”.

Additional possibilities

- The cell division could be made in different ways. The octree way is one possibility.
- As already described, an item can be processed as an area or volume instead of as a point. Some of the pseudocode detailed above have to be modified. The main parts needed to be modified are:
    - Item insertion, so the item is added to one or more cells as needed.
    - Culling, so a item is taken as part of the result only one time.
- Controlling a cell neighbourhood we probably (pending to be tested) can develop a nice “server meshing” or load balancing system) system as you can distribute the cells into different computers dynamically. It is relatively easy as we always have the references/pointers to every cell involved. The next step is to use this type of index, with a cell item limit as big as the limit of items/players in a server and use as items the binding box of each server assigned space/cells.
- Can be used in databases for searching values in a range.

Similar or related algorithms

- Hierarchical view frustum culling. https://www.halolinux.us/3d-graphics/hierarchical-view-frustum-culling.html
- Portal culling (Luebke 1995)
- Hierarchical Z-buffering (HZB) http://www.cs.princeton.edu/courses/archive/spr01/cs598b/papers/greene93.pdf
- https://en.wikipedia.org/wiki/Back-face_culling
- Quad trees, Octree, etc.
- https://en.wikipedia.org/wiki/Vector_space
- Open Geospatial Consortium is using some type of algorithm. I have seen they uses as well the binding boxes. www.opengeospatial.org
- https://webpages.uncc.edu/krs/courses/5010/ged/lectures/cull_lod2.pdf