Exercise – Specialty Trees

Exercises:

Implement your own Quadtree in the project provided.

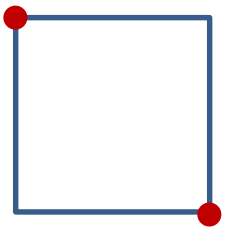
The Project contains a level with dozens of little triangular enemies randomly scattered across it. As the mouse moves near the enemies they turn red, as it moves away they turn back to orange. Currently the `UpdateEnemies()` function in Game1.cpp is doing this by linearly iterating through all the enemies in an array and performing a distance check against each one.

This distance check is quite slow. In this project we have artificially slowed it down even more to emphasize the point and represent what might happen if we had thousands of enemies in the scene instead of a few dozen.

A Quadtree would allow us to roughly and more efficiently find all the nearby enemies and then we could perform our normal slow distance check against just those enemies to confirm they really are within range (the Quadtree is inaccurate and will give us false positives). This way we only need to check the distance of a small number of enemies instead of inefficiently checking every single one.

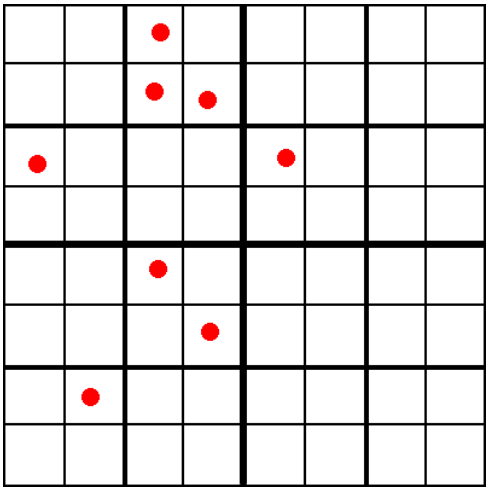
- Use the provided project.
- The code you need to replace is in the UpdateEnemies() function in Game1.cpp.
- The included Vector2 class provides a good way to store positions.
- The Included HelperFunctions.h file provides a function to detect collision between circles and squares (the circular radius around the mouse against the square quads in the quadtree). It also contains a function to help render a quad so you can visualize your quadtree.

1. Implement your own Quadtree. This should consist of:
    a. A Node class containing:
        i. The top left and bottom right corners of the node's quad. This is all the positional data you need to store a square.
        ii. The ability to store pointers to one or more enemies that are within the quad.
        iii. Pointers to the node's 4 child nodes.
    b. A Quadtree class containing:
        i. The root node.
        ii. A function to add enemies to the Quadtree.
        iii. A function to search the tree for all enemies in an area.
    c. Your Add and Search functions could be coded entirely in the Quadtree or could recursively call functions in the Nodes. The latter is probably the better option. *(e.g. instead of the Quadtree searching through all nodes, it could call a search function on the root which checks itself and then recursively calls the same search function on its 4 children.)*

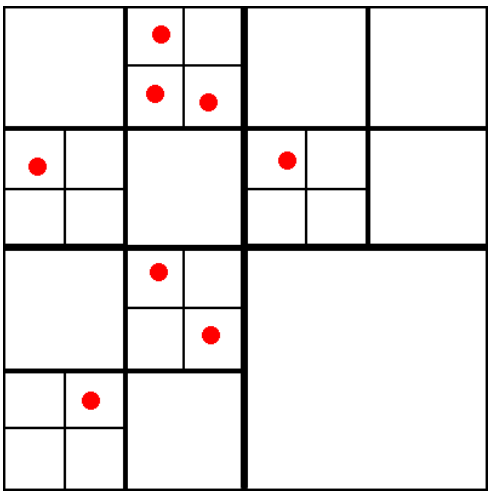


2. Add your Quadtree to the provided project:
    a. Instantiate the Quadtree within Game1.cpp.
    b. When the enemies are created in Game1's constructor, add them to the Quadtree at the same time.
    c. At the bottom of the UpdateEnemies() function, replace the linear search with your Quadtree search.
    d. Optionally you could use the DrawSquare() function in HelperFunctions.h to draw your Quadtree so you can check it's working correctly.

3. CHALLENGE: Make your Quadtree Adaptive. This means that rather than subdividing your level equally over and over, you only divide when needed. This methods works as follows:
    a. Start with just a root node with 4 null child nodes.
    b. When an enemy is added, add it to the root node.
    c. If an enemy is added and the root node is already full (it's up to you how many it can hold, but 1 is probably easiest):
        i. Subdivide the root into 4.
        ii. Add these 4 new nodes as children of the root.
        iii. Move the root's existing enemies into the appropriate child.
        iv. Add the new enemy that caused this split into the appropriate child node.
    d. Now that the root node has 4 children, it should never store its own list of enemies. Instead any new enemy added should be added to the appropriate child.
    e. If the child node is full, subdivide it again as described above and move its enemies down to its new children.
    f. In this way each node should have:
        i. Either a list of enemies as leaves in the tree or 4 quad nodes, never both.
        ii. Any enemies should always be leaves.



Quadtree



Adaptive Quadtree