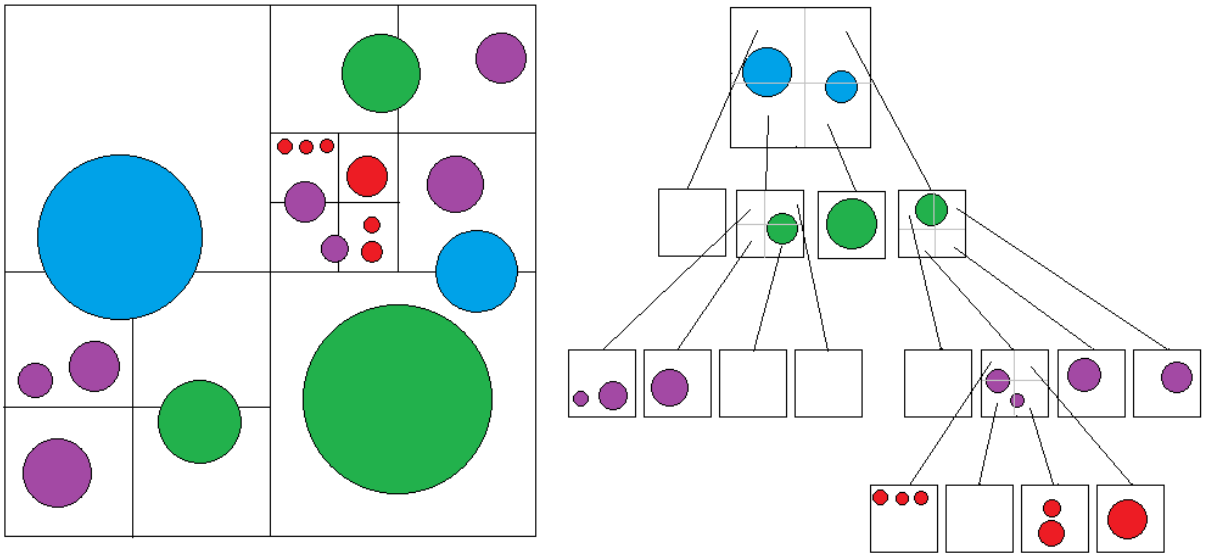
Homework 3: Quad Trees

Due: Sunday, March 8th by 11:59PM

Quads Trees are used to recursively partition a two dimensional space. Each quad can be subdivided into 4 smaller, equal sized regions. Non-moving objects are added to the tree at different levels depending on their size and location. This allows large areas of a game map to be quickly ignored when searching for objects, increasing performance. Feel free to read up on Quad Trees on Wikipedia.



In the above graphic, a Quad Tree containing circle objects is shown in two ways. On the left is the actual layout of the objects divided up by the quad tree. Each color represents objects that are stored at *different levels* of the tree (not just leaf nodes). The circle objects are stored in the smallest rectangle that completely contains them. For instance, the blue circles are contained in the root node of the tree since that is the smallest rectangle that completely contains them.

On the right are the same divisions laid out in a tree structure, with the circles in their corresponding quads at each level. The blue circles are stored in the root, the green ones are in the next level and so on.

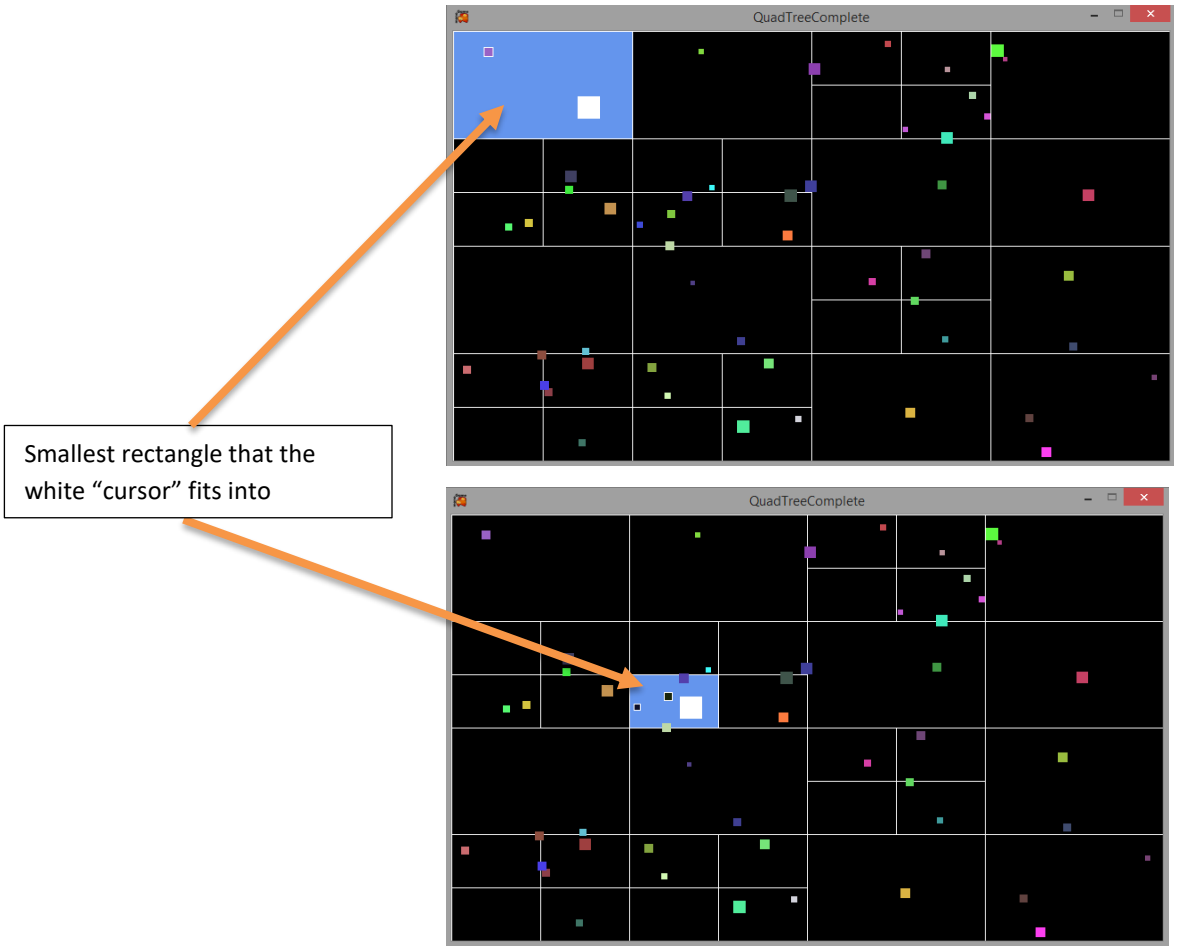
Basic Information & Examples

For an example of what you're making, run **QuadTreeCOMPLETE.exe**. In this example, your mouse cursor is a small white box. The white outlines around the screen are the different Quad Tree subdivisions, also known as "quads". The program will highlight the smallest quad

(rectangle) that your “cursor” currently fits into. The program will also outline and blink any colored boxes that are in the same quad as the cursor.

Each time you run the program, a different configuration of colored boxes will be randomly generated. This, in turn, will cause the program to create a different quad tree, as each rectangle will sub-divide when there are more than 3 boxes inside of it. After sub-dividing, a quad may actually end up with more than 3 boxes inside of it if none of those boxes completely fit into any of the sub-divisions.

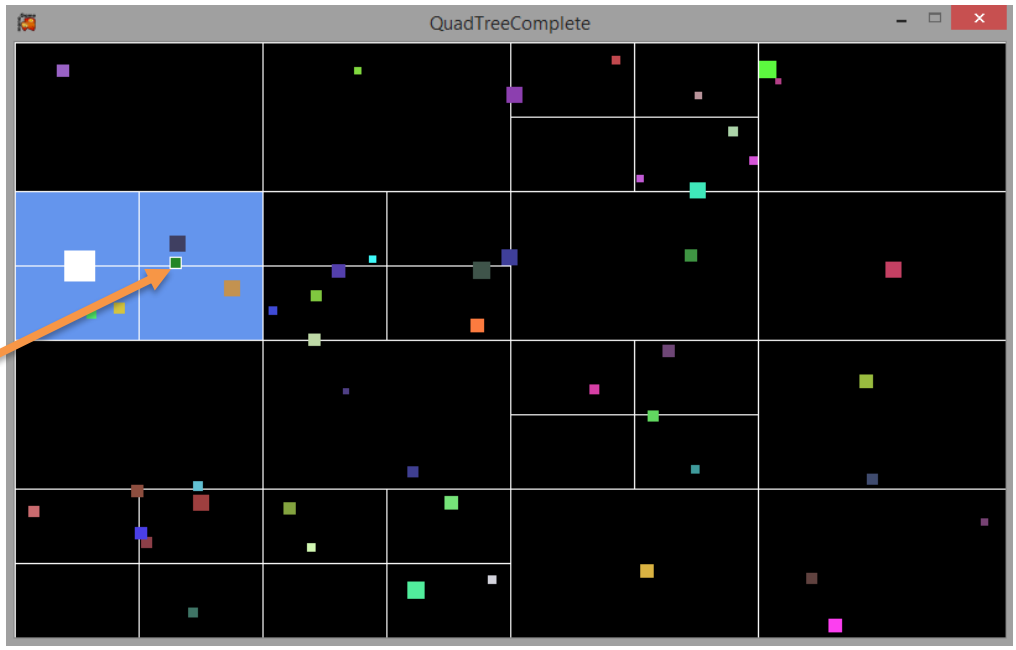
An important note: objects can be stored at *any* level of this particular tree, *not* only at leaf nodes. This particular implementation works this way because a game object (the color boxes) might not fit completely inside any particular leaf node. They will, at the very least, *always* fit inside the root node.



More Examples

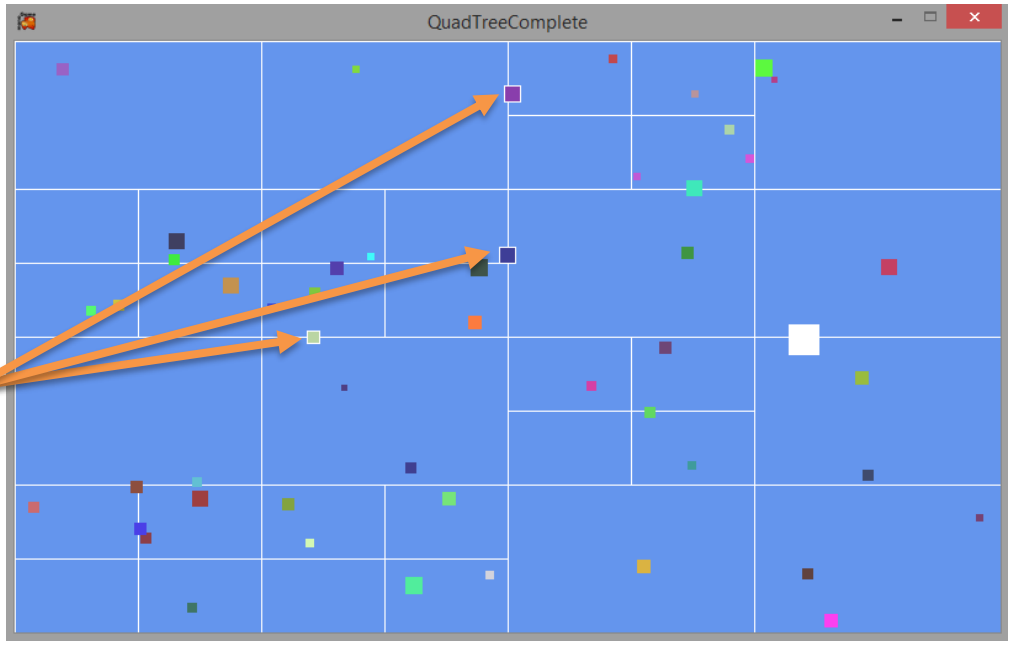
In the example to the right, the cursor does not fit inside of a leaf node. It is instead fully contained in a node which also has subdivisions.

Notice too that there is another object contained at the same level, since it does not fit into any smaller subdivisions.



In this example, the cursor only completely fits into the root node, so the entire window is highlighted.

Notice there are three other objects contained at this level, since the root node is the only quad (rectangle) in which they completely fit.



Starter Code

If you run the starter project without adding any code, you'll see different sized and colored box objects on the screen. Each time you run the program, the boxes will be different. Moving the mouse around will move a white box around the screen. The goal is to get all of the colored box objects added to the quad tree, and have it subdivide as many times as necessary. Once this is done, moving the mouse around should highlight the smallest quad that contains the white box, as well as any other colored objects in that quad.

The code to draw and highlight the quads, as well as flash the objects, is already written for you. As you complete the required methods described below, some of them will be called for you.

Activity #1: Adding & Subdividing

Start with the starter project from the .zip file you got from MyCourses. The **QuadTreeNode** class represents a single node of your quad tree.

For this activity, you will need to complete the **AddObject()** and **Divide()** methods in the **QuadTreeNode** class (found in **QuadTree.cs**). The details of each method are outlined below. Once these are complete, the randomly generated cubes will be added to the quad tree and each quad should subdivide as necessary.

In general, a quad shouldn't divide until its list of objects grows larger than the constant **MAX_OBJECTS_BEFORE_SUBDIVIDE**. When a divide occurs, any eligible objects currently in the quad being divided should be *moved* into the new smaller quads (if they fit). Objects which do not completely fit into one of the newly created subdivisions should *remain* in the current quad. You may end up subdividing and not actually moving a single object, if none of them actually fit into the subdivisions.

The **QuadTreeNode** objects have a **Rectangle** property that represents the area they occupy, as well as a list of game objects in that quad. Each **QuadTreeNode** also has an array of 4 more **QuadTreeNode** objects, which are the subdivisions. Remember: An object can be anywhere in the tree, not just leaf nodes.

`void AddObject(GameObject gameObj)`

This method takes one parameter: the new game object to add to the tree. The first thing to check is whether or not the object's rectangle even fits into this quad. Use the **Rectangle** class's **Contains()** method to see if one rectangle is completely contained within another.

If the game object doesn't fit inside this quad, it simply shouldn't be added and you're done. If it does fit, you will need to determine where it should be added. The object could be added to this quad's game object list, or it could be added to one of the subdivisions' lists (if there are any subdivisions yet). Make this decision based on which of these quads contains the object being added. If the object could fit inside a subdivision, recursively call AddObject on that quad tree node.

You may need to divide at this point. You should think about what conditions would warrant a division. You only want to subdivide (by calling the **Divide()** method) when those conditions are met, and you should only ever do it once per quad node. Once a division occurs, game objects can still be added to this quad's list since the object might not fit into any of the subdivisions.

For debugging purposes, it might be useful to print out some information inside the **AddObject()** method to determine where (and if) the object is being added.

`void Divide()`

The **Divide()** method should create 4 new **QuadTreeNode** objects and store them in the **divisions** array of the quad being divided. Only do this if the quad hasn't been divided yet. Each of these new quads will take up ¼ of the area of its parent. The constructor takes an x, y, width and height for the new **QuadTreeNode**. Each division should be the same size but have a different position.

Once the divisions have been created, the next step is to move any eligible game objects from this quad to whichever child quad (one of the 4 new divisions) they fit into. If an object doesn't fit inside any of the new divisions, don't actually move it. Remember: any quad can hold game objects, not just new divisions (leaf nodes).

Activity #2: Extracting Information

A quad tree would be useless if we couldn't get information back out of it. For this simple homework, you're going to get two pieces of information from the tree: a list of every rectangle in the tree (for debugging purposes), and the single quad that contains a given rectangle. You will need to complete the following methods found in the **QuadTreeNode** class.

`List<Rectangle> GetAllRectangles()`

The **GetAllRectangles** method has one purpose: to return a list with all of the *quad node rectangles* in the tree, including any subdivisions. Since each quad already contains a rectangle object for you, add that to the list. This will be a recursive method, so call **GetAllRectangles** on any subdivisions and add the elements from those method calls to this list as well. You may find the List class's "AddRange" method useful for this, as it adds all elements from one list to another. Lastly, return the list.

Once this method is working, you should see all of the subdivision rectangles on the screen.

`QuadTreeNode GetContainingQuad(Rectangle rect)`

This method takes a rectangle parameter and returns the smallest quad that contains that rectangle. There are three possible cases that could arise. The easiest is that this quad's rectangle doesn't contain the given rectangle at all. In that case, simply return null. If the quad does contain the rectangle, then one of the other cases will be valid: Either the rectangle is only contained by this quad, or it is also contained by one of the subdivisions. Since we're looking for the smallest quad, the subdivisions would take precedence over the parent. Once you find the correct quad, return that whole object (which will be a **QuadTreeNode** object).

Once this method is working, moving the mouse around the screen will highlight the smallest quad that contains the moving white box. It will also highlight any other objects in that quad by flashing them red.

All done?

Once finished, your program should match the functionality of the provided sample .exe file.