Quick Tip: Use Quadtrees to Detect Likely Collisions in 2D Space

Steven Lambert
Sep 3, 2012 • 8 min read

Coding Fundamentals | Game Development

Many games require the use of collision detection algorithms to determine when two objects have collided, but these algorithms are often expensive operations and can greatly slow down a game. In this article we'll learn about quadtrees, and how we can use them to speed up collision detection by skipping pairs of objects that are too far apart to collide.

Note: *Although this tutorial is written using Java, you should be able to use the same techniques and concepts in almost any game development environment.*

Introduction

Collision detection is an essential part of most video games. Both in 2D and 3D games, detecting when two objects have collided is important as poor collision detection can lead to some very interesting results:

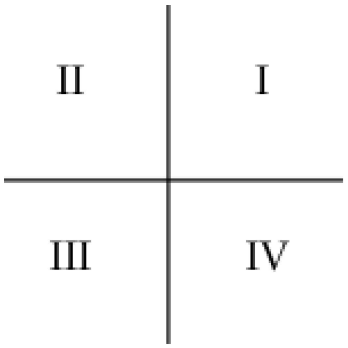
However, collision detection is also a very expensive operation. Let's say there are 100 objects that need to be checked for collision. Comparing each pair of objects requires 10,000 operations - that's a lot of checks!

One way to speed things up is to reduce the number of checks that have to be made. Two objects that are at opposite ends of the screen can not possibly collide, so there is no need to check for a collision between them. This is where a quadtree comes into play.

What Is a Quadtree?

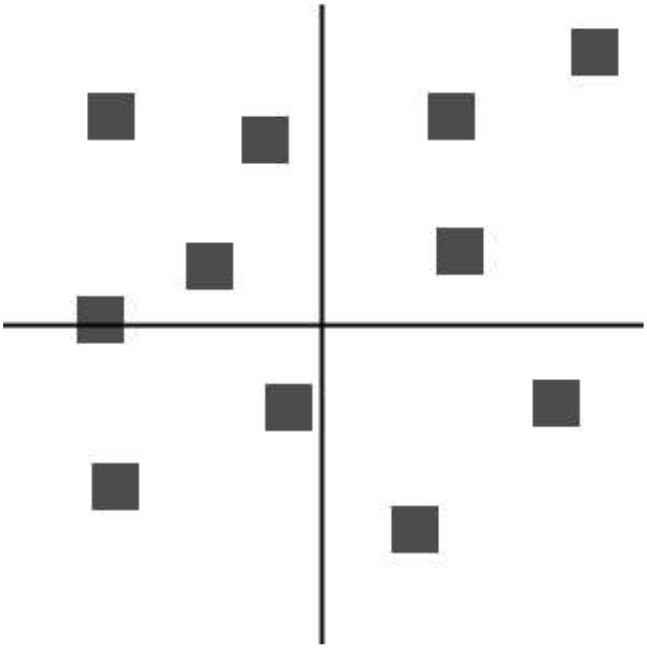
A quadtree is a data structure used to divide a 2D region into more manageable parts. It's an extended binary tree, but instead of two child nodes it has four.

In the images below, each image is a visual representation of the 2D space and the red squares represent objects. For the purposes of this article, subnodes will be labelled counter-clockwise as follows:

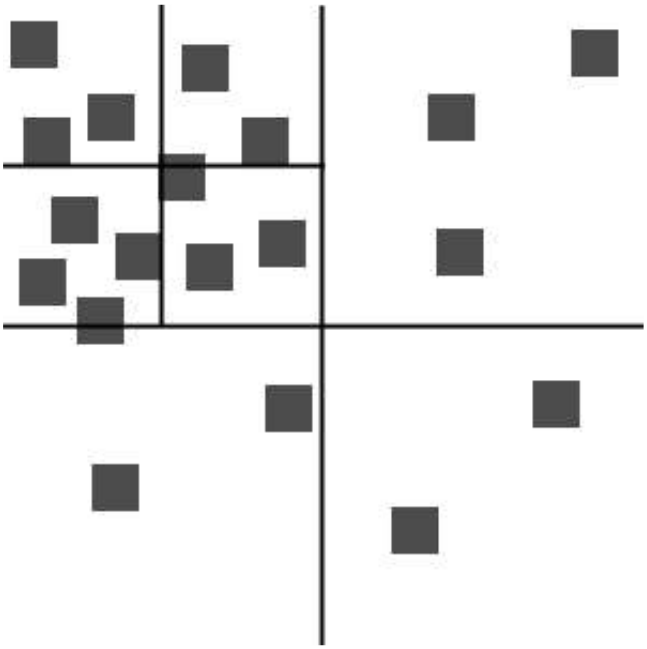


A quadtree starts as a single node. Objects added to the quadtree are added to the single node.

When more objects are added to the quadtree, it will eventually split into four subnodes. Each object will then be put into one of these subnodes according to where it lies in the 2D space. Any object that cannot fully fit inside a node's boundary will be placed in the parent node.



Each subnode can continue subdividing as more objects are added.



As you can see, each node only contains a few objects. We know then that, for instance, the objects in the top-left node cannot be colliding with the objects in the bottom-right node, so we don't need to run an expensive collision detection algorithm between such pairs.

Take a look at this JavaScript example to see a quadtree in action.

Implementing a Quadtree

Implementing a quadtree is fairly simple. The following code is written in Java, but the same techniques can be used for most other programming languages. I'll comment after each code snippet.

We'll start off by creating the main Quadtree class. Below is the code for `Quadtree.java`.

```
public class Quadtree {

  private int MAX_OBJECTS = 10;
  private int MAX_LEVELS = 5;

  private int level;
  private List objects;
  private Rectangle bounds;
  private Quadtree[] nodes;

 /*
* Constructor
*/
  public Quadtree(int pLevel, Rectangle pBounds) {
   level = pLevel;
   objects = new ArrayList();
   bounds = pBounds;
   nodes = new Quadtree[4];
  }
}
```

The `Quadtree` class is straightforward. `MAX_OBJECTS` defines how many objects a node can hold before it splits and `MAX_LEVELS` defines the deepest level subnode. `Level` is the current node level (0 being the topmost node), `bounds` represents the 2D space that the node occupies, and `nodes` are the four subnodes.

In this example, the objects the quadtree will hold are `Rectangles`, but for your own quadtree it can be whatever you want.

Next, we'll implement the five methods of a quadtree: `clear`, `split`, `getIndex`, `insert`, and `retrieve`.

```
/*
```

```
 * Clears the quadtree
 */
  public void clear() {
    objects.clear();

    for (int i = 0; i < nodes.length; i++) {
      if (nodes[i] != null) {
        nodes[i].clear();
        nodes[i] = null;
      }
    }
  }
```

The `clear` method clears the quadtree by recursively clearing all objects from all nodes.

```
/*
 * Splits the node into 4 subnodes
 */
  private void split() {
    int subWidth = (int)(bounds.getWidth() / 2);
    int subHeight = (int)(bounds.getHeight() / 2);
    int x = (int)bounds.getX();
    int y = (int)bounds.getY();

    nodes[0] = new Quadtree(level+1, new Rectangle(x + subWidth, y, subWidth, subHei
    nodes[1] = new Quadtree(level+1, new Rectangle(x, y, subWidth, subHeight));
    nodes[2] = new Quadtree(level+1, new Rectangle(x, y + subHeight, subWidth, subHe
    nodes[3] = new Quadtree(level+1, new Rectangle(x + subWidth, y + subHeight, subW
  }
```

The `split` method splits the node into four subnodes by dividing the node into four equal parts and initializing the four subnodes with the new bounds.

```
/*
 * Determine which node the object belongs to. -1 means
 * object cannot completely fit within a child node and is part
 * of the parent node
 */
  private int getIndex(Rectangle pRect) {
    int index = -1;
    double verticalMidpoint = bounds.getX() + (bounds.getWidth() / 2);
    double horizontalMidpoint = bounds.getY() + (bounds.getHeight() / 2);

    // Object can completely fit within the top quadrants
    boolean topQuadrant = (pRect.getY() < horizontalMidpoint && pRect.getY() + pRect
    // Object can completely fit within the bottom quadrants
    boolean bottomQuadrant = (pRect.getY() > horizontalMidpoint);

    // Object can completely fit within the left quadrants
```

```
    if (pRect.getX() < verticalMidpoint && pRect.getX() + pRect.getWidth() < vertica
       if (topQuadrant) {
         index = 1;
       }
       else if (bottomQuadrant) {
         index = 2;
       }
     }
     // Object can completely fit within the right quadrants
     else if (pRect.getX() > verticalMidpoint) {
      if (topQuadrant) {
        index = 0;
      }
      else if (bottomQuadrant) {
        index = 3;
      }
    }

    return index;
  }
```

The `getIndex` method is a helper function of the quadtree. It determines where an object belongs in the quadtree by determining which node the object can fit into.

```
/*
* Insert the object into the quadtree. If the node
* exceeds the capacity, it will split and add all
* objects to their corresponding nodes.
*/
 public void insert(Rectangle pRect) {
   if (nodes[0] != null) {
     int index = getIndex(pRect);

     if (index != -1) {
       nodes[index].insert(pRect);

       return;
     }
   }

   objects.add(pRect);

   if (objects.size() > MAX_OBJECTS && level < MAX_LEVELS) {
      if (nodes[0] == null) {
         split();
      }

     int i = 0;
     while (i < objects.size()) {
       int index = getIndex(objects.get(i));
       if (index != -1) {
         nodes[index].insert(objects.remove(i));
       }
       else {
```

```
            i++;
          }
        }
      }
    }
```

The `insert` method is where everything comes together. The method first determines whether the node has any child nodes and tries to add the object there. If there are no child nodes or the object doesn't fit in a child node, it adds the object to the parent node.

Once the object is added, it determines whether the node needs to split by checking if the current number of objects exceeds the max allowed objects. Splitting will cause the node to insert any object that can fit in a child node to be added to the child node; otherwise the object will stay in the parent node.

```
/*
* Return all objects that could collide with the given object
*/
 public List retrieve(List returnObjects, Rectangle pRect) {
   int index = getIndex(pRect);
   if (index != -1 && nodes[0] != null) {
     nodes[index].retrieve(returnObjects, pRect);
   }

   returnObjects.addAll(objects);

   return returnObjects;
 }
```

The final method of the quadtree is the `retrieve` method. It returns all objects in all nodes that the given object could potentially collide with. This method is what helps to reduce the number of pairs to check collision against.

Using This for 2D Collision Detection

Now that we have a fully functional quadtree, it's time to use it to help reduce the checks needed for collision detection.

In a typical game, you'll start by creating the quadtree and passing the bounds of the screen.

```
Quadtree quad = new Quadtree(0, new Rectangle(0,0,600,600));
```

At every frame, you'll insert all objects into the quadtree by first clearing the quadtree then using the `insert` method for every object.

```
quad.clear();
for (int i = 0; i < allObjects.size(); i++) {
  quad.insert(allObjects.get(i));
}
```

Once all objects have been inserted, you'll go through each object and retrieve a list of objects it could possibly collide with. You'll then check for collisions between each object in the list and the initial object, using a collision detection algorithm.

```
List returnObjects = new ArrayList();
for (int i = 0; i < allObjects.size(); i++) {
  returnObjects.clear();
  quad.retrieve(returnObjects, objects.get(i));

  for (int x = 0; x < returnObjects.size(); x++) {
    // Run collision detection algorithm between objects
  }
}
```

Note: Collision detection algorithms are beyond the scope of this tutorial. See this article for an example.

Conclusion

Collision detection can be an expensive operation and can slow down the performance of your game. Quadtrees are one way you can help speed up collision detection and keep your game running at top speeds.

Related Posts

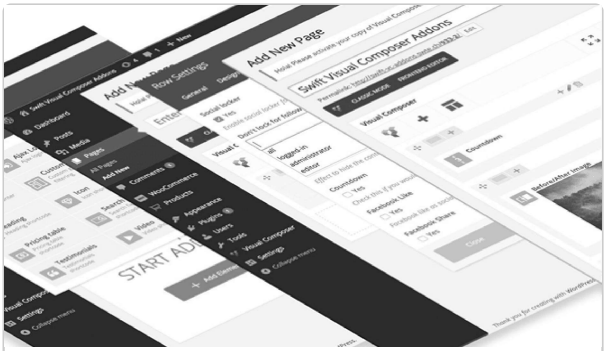
- Make Your Game Pop With Particle Effects and Quadtrees

Sign up

Steven Lambert

Steven Lambert is a Computer Science graduate, a freelance web and
applications developer, avid gamer, and hobbyist game developer. He often writes about HTML5 game development on his blog and you can follow Steven on Twitter to stay up-to-date with his latest activities.

StevenKLambert



WordPress Plugins

From $5

PHP Scripts

From $5

Unlimited Downloads
From $16.50/month

Get access to over one million creative assets on Envato Elements.

Over 9 Million Digital Assets

Everything you need for your next creative project.

QUICK LINKS - Explore popular categories

ENVATO TUTS+

About Envato Tuts+
Terms of Use
Advertise

HELP

FAQ
Help Center

tuts+

30,388 Tutorials

553 Courses

42,531 Translations

Certified B Corporation
