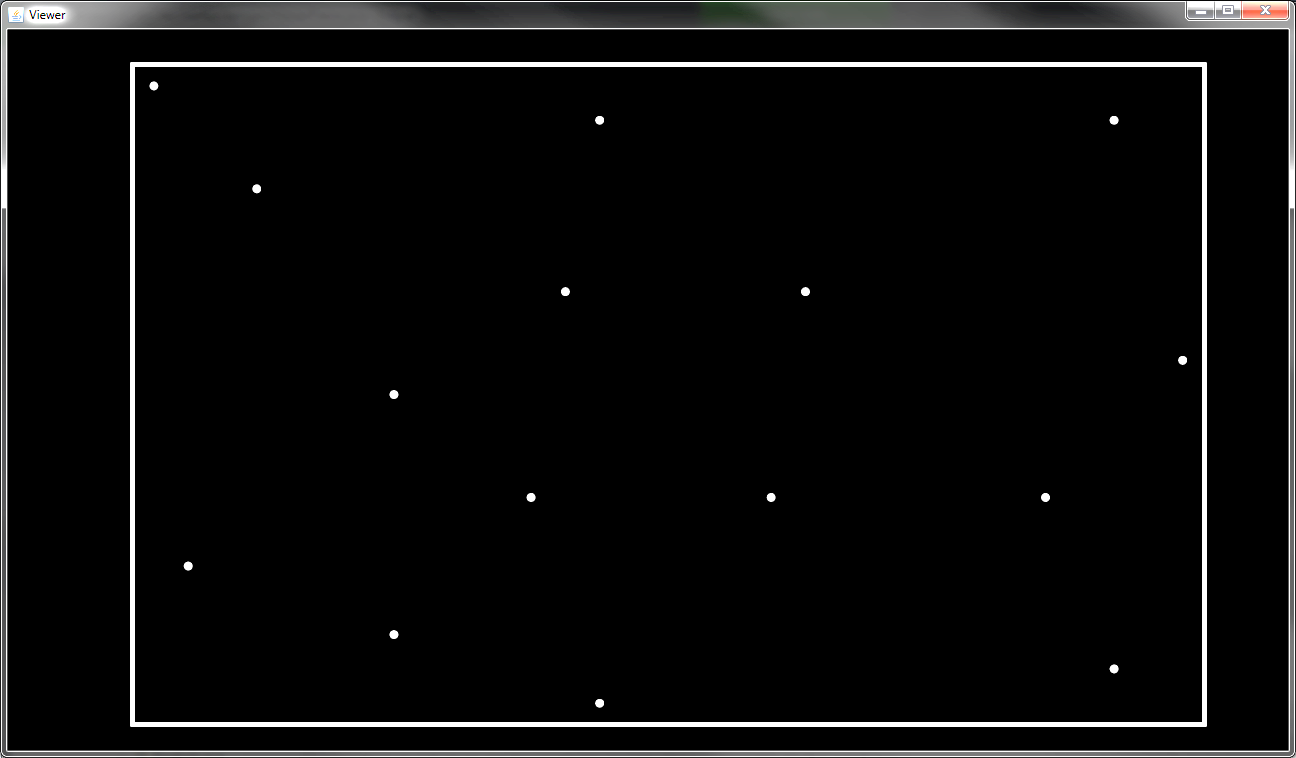
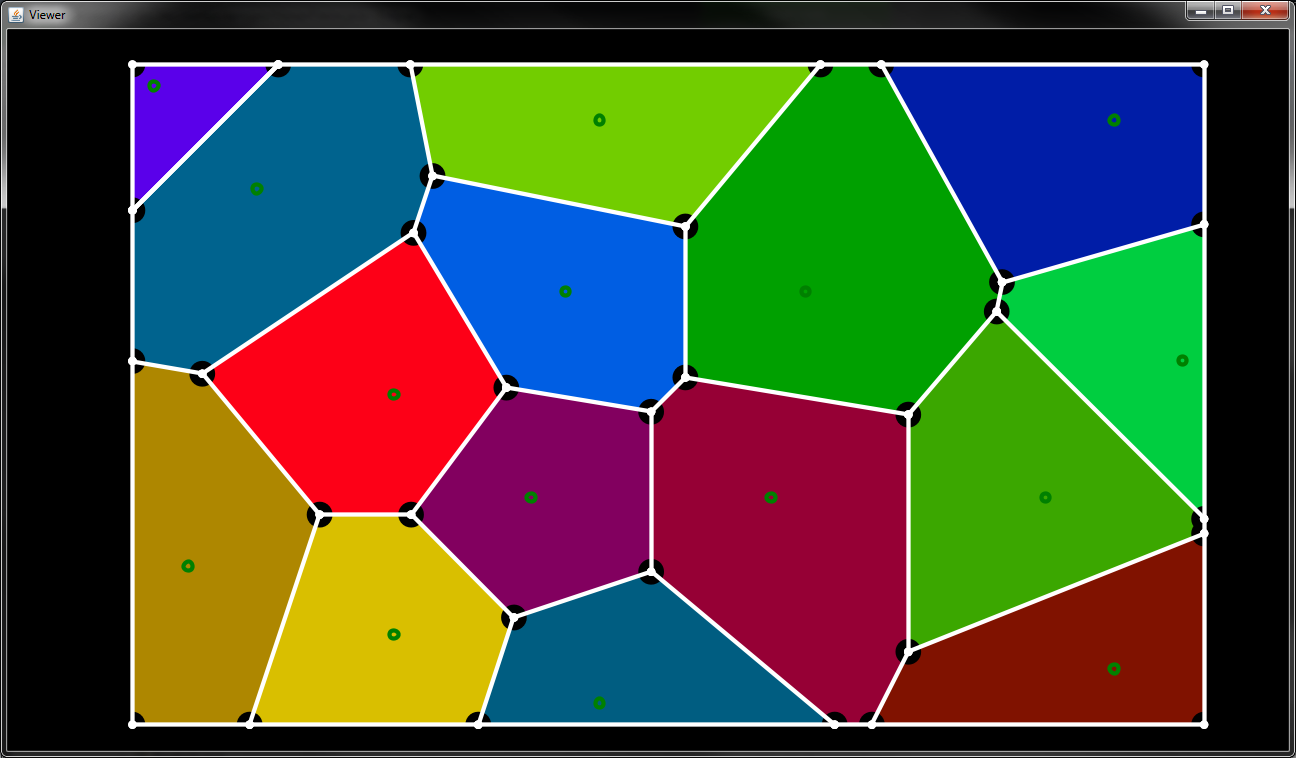
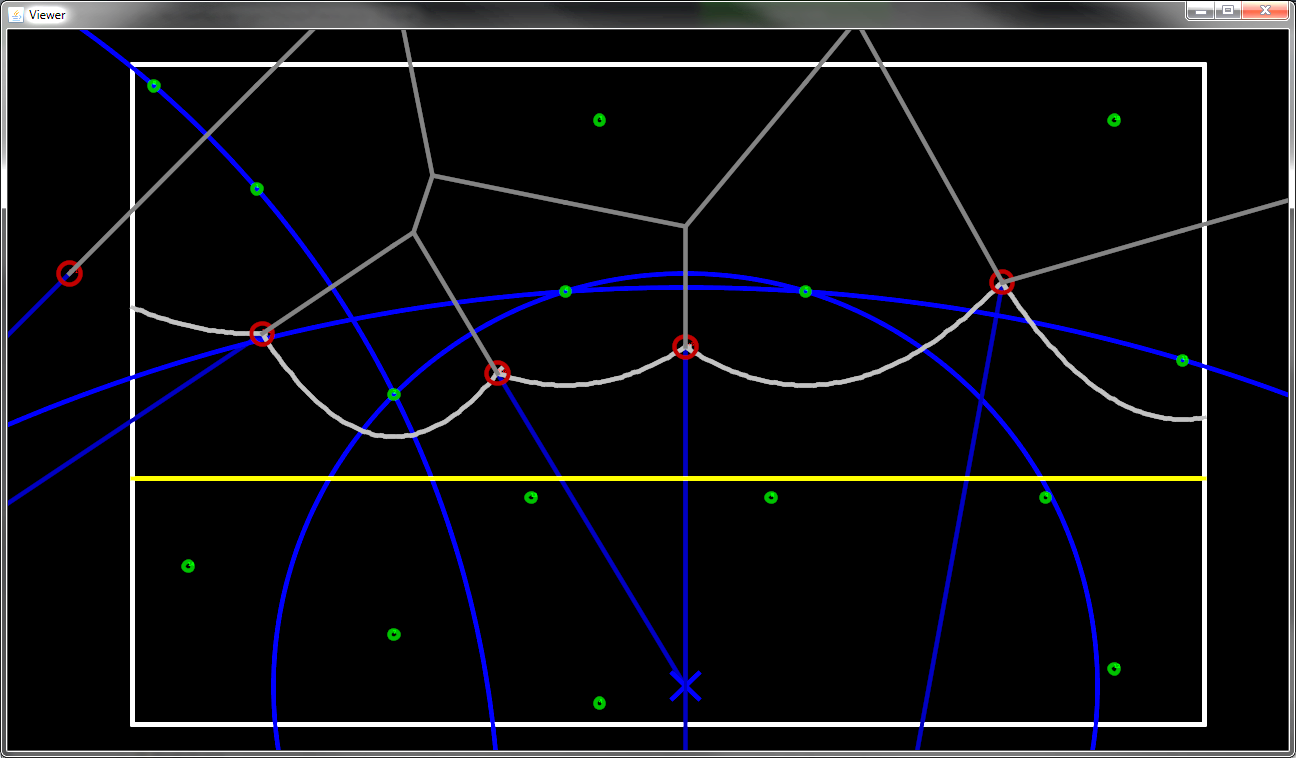
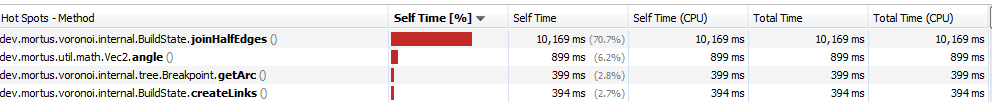
Optimizing the Voronoi Diagram Generator

**Early Results:**

**

**Optimizations:**

This code feels like it could be faster. We’re seeing 30,000 sites taking a full 3.5 seconds. Optimization may not be necessary, but I can’t help but take a look at what a simple profiler has to say:



This is a very unbalanced distribution of time, especially since joinHalfEdges() is a method that only gets called ONCE! Let’s see what makes this method so greedy.

The original method code:

**private** **void** joinHalfEdges() {

List<MutableEdge> remove = **new** ArrayList<MutableEdge>();

List<MutableEdge> add = **new** ArrayList<MutableEdge>();

**for** (MutableEdge e : edges) {

**if** (remove.contains(e)) **continue**;

**if** (!(e **instanceof** HalfEdge)) **continue**;

HalfEdge edge = (HalfEdge) e;

HalfEdge twin = edge.getTwin();

vertices.remove(edge.start());

remove.add(edge);

remove.add(twin);

add.add(edge.joinHalves());

}

**for** (MutableEdge edge : remove) removeEdge(edge);

**for** (MutableEdge edge : add) addEdge(edge);

}

At first, this method seems an odd candidate for biggest time hog. However, considering the way in which most edges are formed, the majority of edges are actually half edges. Why is this important? Because ArrayList is being used heavily in a bad way. This method is calling .contains() for every edge as well as issuing two remove calls per edge. ArrayList has linear search time for the .contains() and .remove() methods.

So, to solve this method’s overwhelming greediness, there are three possible solutions:

1. Use a data structure with better lookup and remove times
2. Remove as many lookups as possible
3. Both

I went with the #2 for simplicity. It is perfectly true that another data structure could be better here. But in order to get faster results I wanted to avoid finding or implementing multiple other structures to determine which is best. #2 is a solid optimization regardless of data structure. As we will see, it is enough. Here is the modified code:

**private** **void** joinHalfEdges() {

List<MutableEdge> joined = **new** ArrayList<MutableEdge>();

**for** (MutableEdge e : edges) {

**if** (!e.isHalf()) {

joined.add(e);

**continue**;

}

HalfEdge edge = (HalfEdge) e;

HalfEdge twin = edge.getTwin();

**if** (edge.hashCode() > twin.hashCode()) **continue**;

joined.add(edge.joinHalves());

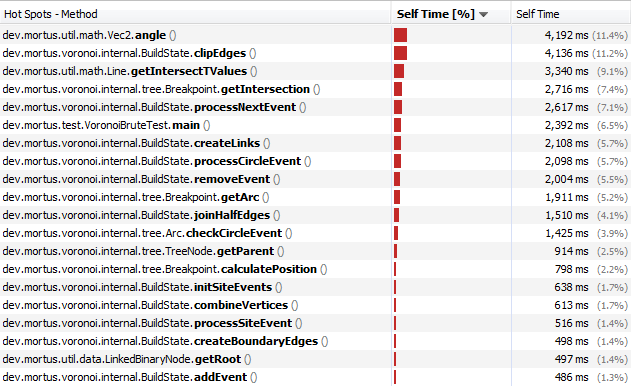
}

edges = joined;

}

The new code removes every single .contains() and .remove() call. Removing the starting vertex of the half edges has been made unnecessary by simply not adding it to the list in the first place. We also avoid the extra two lists I was previously using to get around concurrent modification. Now all items that “pass” are placed into a new list. Finally, to ensure that only one combined edge is created per pair, the hash code values of the twin half edges are now used to select the lower of the pair. These improvements make the code much faster and easier to read.

Here are the new profiling results:



This profiling chart looks much more balanced, with most of the most time consuming methods being methods that are actually doing a lot of work.

Interestingly, Vec2.angle() is already a memoized result because I knew it would be expensive and frequently used. With these results, further optimization looks unnecessary.

**public** **double** angle() {

**if** (angle == Double.***MAX\_VALUE***) {

angle = Math.*atan2*(y, x);

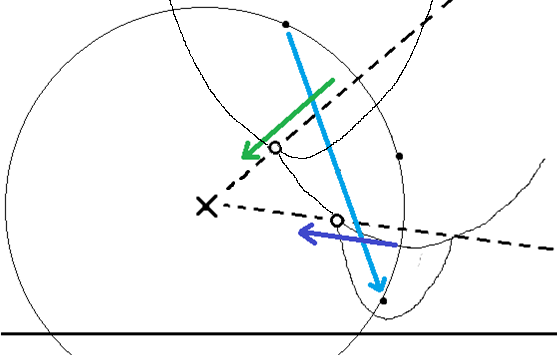
}

**return** angle;

}

**1, 2, Skip a few…**

In the end, a lot of bugs existed. Documenting all of them as I worked on debugging would have been a huge waste of time. But the most important improvement I remember making was a change to how circle events are detected.

Checking where breakpoint collisions occur used to be done with Ray vs. Ray intersection tests and a number of edge cases related to rounding errors (which didn’t work).   
  
Solution: using the center of a circumcircle including the 3 site points gives the intersection point of the 2 breakpoints. Divergence is now tested with dot products instead of an intersection test:

The diagram shows two breakpoints and the fact that their intersection is the center of a circumcircle. It also shows the 3 colored vectors that are used to determine divergence:   
  
if (Green.dot(Blue) > Purple.dot(Blue)) {  
 // intersection  
}

Green is the direction of motion of the left breakpoint, Purple is the direction of motion of the right breakpoint, and Blue is the vector from the left arc’s site to the right arc’s site. After making this improvement, a lot of glitches and bugs vanished, presumably due to rays that *should* intersect missing each other.

I also put in the effort to switch away from ArrayLists and designed my own storage classes that would provide much better insert/delete speed.

**Final Results**

The improvements made to the Voronoi code since its first working versions are significant. While I do believe further improvements are possible, I am more than satisfied with these results. Now the code can handle a very large number of sites without error and is much faster than it used to be (more than twice as fast). Here is the graph:

The red set is the generated from the old code, before optimizations. After about 170,000 sites, the old code could not reliably compute a proper Voronoi diagram, instead the somewhat rare errors become far too common. The new code can likely handle an unlimited number of sites, however 1 million seemed a good stopping point.