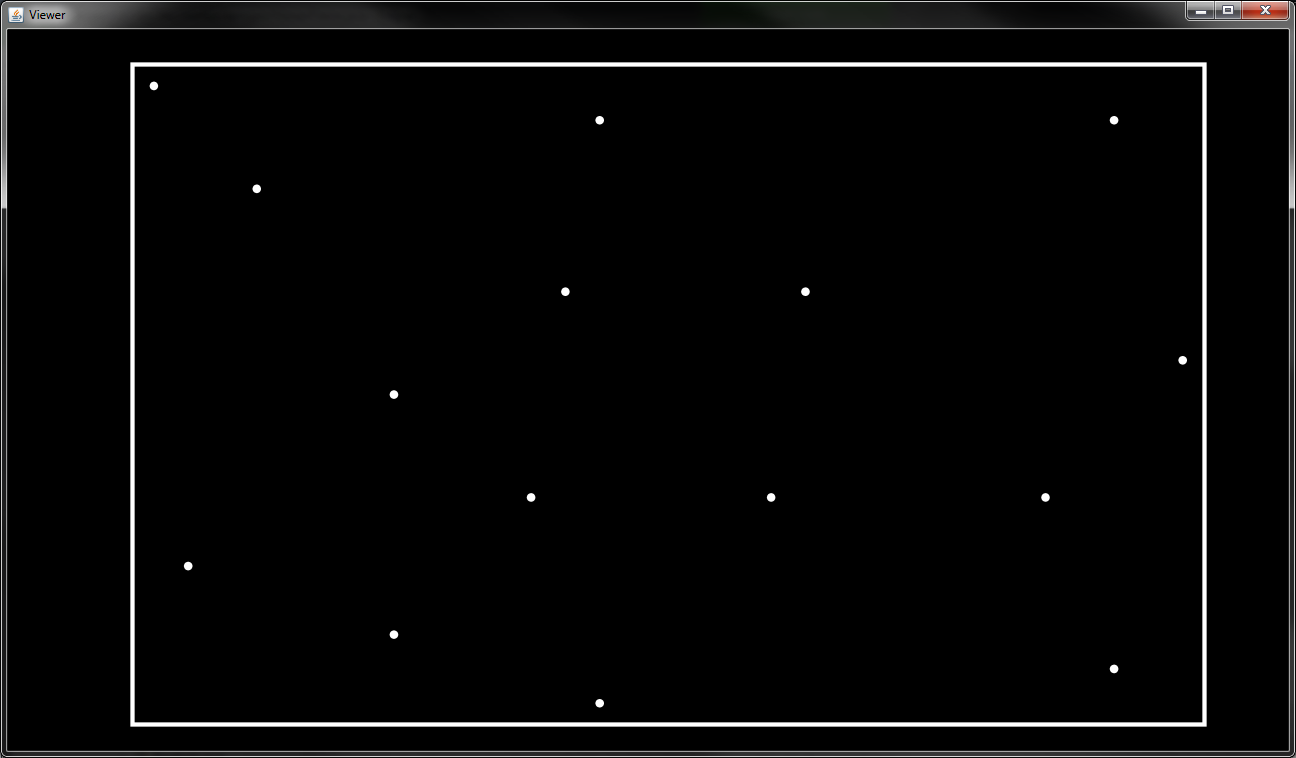
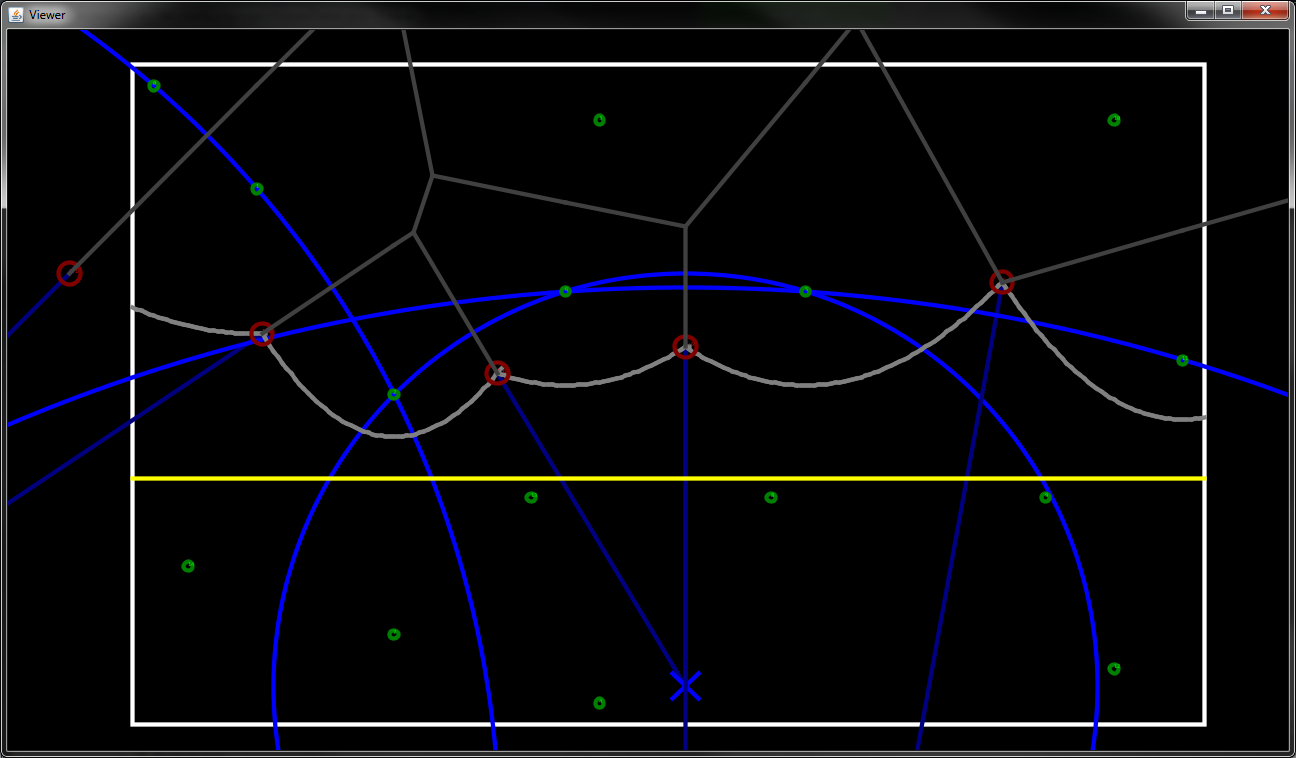
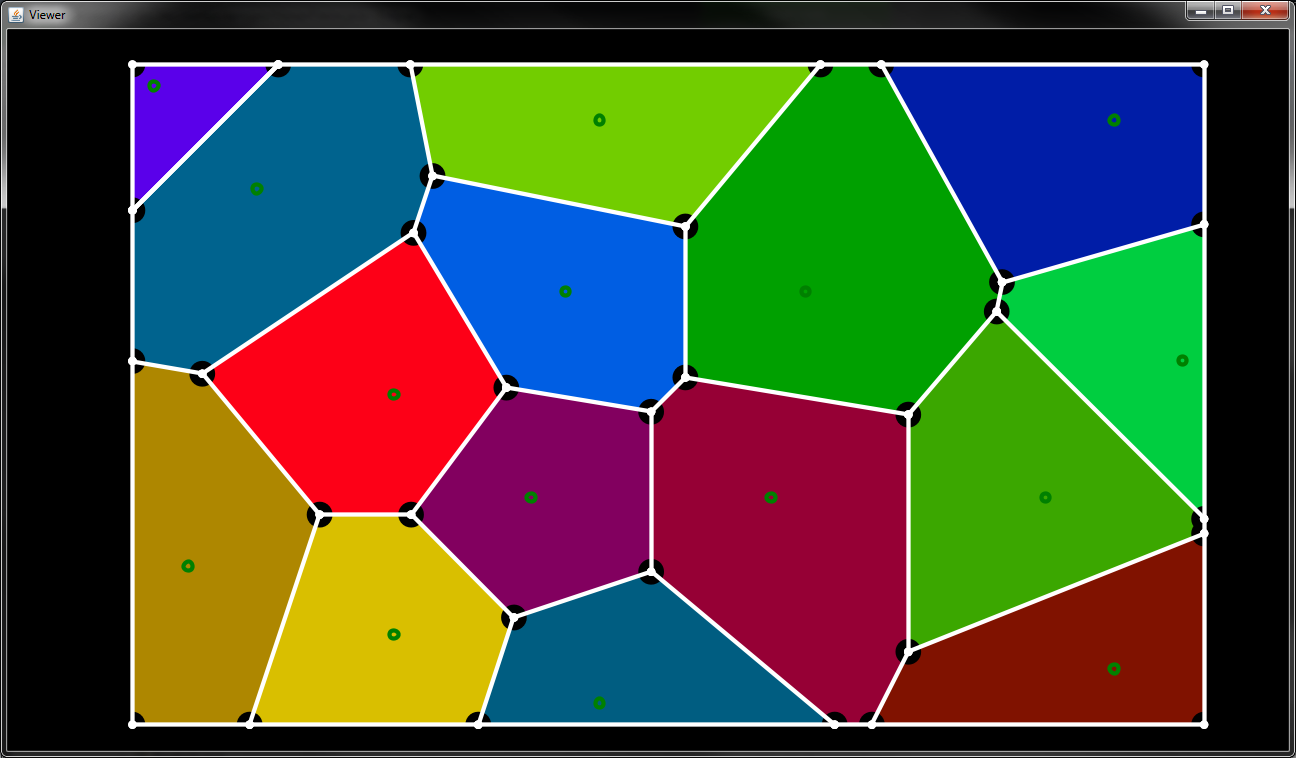
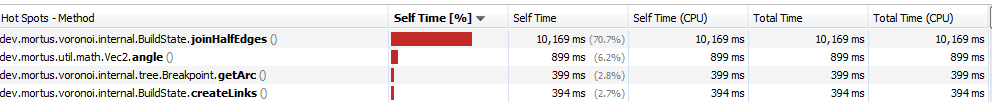
**

Time scaling before modifications

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 checking contains on for every edge as well as issuing two remove calls per edge. ArrayList has linear search time for the .contains() and .remove() methods. Here’s proof:

For each list size, an ArrayList was filled with dummy objects and then searched for every one of them using .contains(). The average time per item was then computed. The time scaling is very convincingly linear, as expected.

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

1. Use a data structure with better lookup 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());

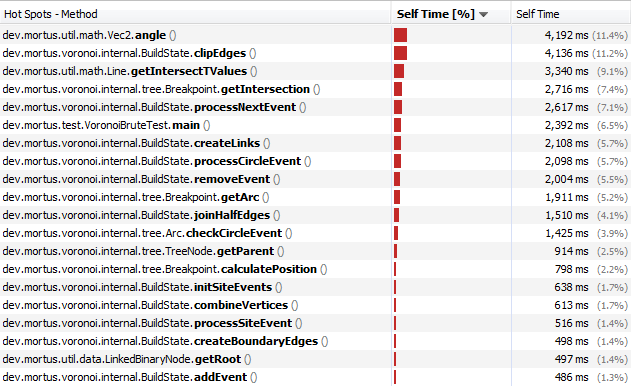
}

edges = joined;

}

The new code removes every single lookup and also avoids the use of “**instanceof**” which I had an unverified suspicion might be slower than an overridden method. 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 many of the top methods being called very frequently. As a matter of fact, Vec2.angle() is already a “cached” 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;

}