My project was inspired by the DARPA Grand Challenge. This is a nationwide competition for autonomous land vehicles, robots racing across the desert with the winning robot’s creator receiving a two million dollar prize and the possibility of government contracts. Now entering the second year of the competition, no vehicles have yet managed to complete the course. Having stumbled across their website, I was intrigued by the idea that people were actually only being partially successful in creating autonomous land vehicles. I knew that this wasn't a trivial problem, but I was still amazed that even such renowned groups such as Carnegie-Mellon were having such difficulty designing an effective vehicle.

Wondering if this would be something that my friends and I could tackle, even on a smaller scale, I simultaneously contacted them and began working on what seemed to me like the first obvious necessity: a computer-controlled navigational system. Since I didn't have an in-depth knowledge of GPS systems, I decided to start out with a simulation using "pixel-like" values as my coordinate system. Interestingly enough, using this type of coordinate system turned out to translate quite easily into an on-screen display of the course and the position of the vehicle. I also decided to use Java as my programming language of choice, both because that is what most of my experience has been with, and also because it would allow us to use the code on whatever kind of machine we could come up with, if we made it to the hardware stage of the project.

Since there are so many people and teams involved in the Grand Challenge and similar endeavors, I am sure that there has been extensive work done on similar systems. Most of these other systems, however, are proprietary and unavailable to the general public. Part of this has to
do with the fact that since they are under development in order to compete in the Grand Challenge, the developers of these systems don't wish to give away the 2 million dollar cash prize and chance at a lucrative government contract in the name of "open source."

Currently, the system I have developed is capable of navigating a variety of courses with a high degree of accuracy. It is limited to using simple simulated values for the GPS coordinates, which correspond directly to pixel coordinates on the screen. The default range setting is for an 800x600 pixel map. Also included is a simple program for generating random maps, primarily for testing a wide variety of situations without having to write each map out by hand. The maps are created in the format of the standard DARPA Route Definition Data File (rddf), this format is described in the Grand Challenge Rules on page 22. The map generation program is set to create routes within certain parameters defined by the simulated vehicle’s on-screen performance, so that the units are arbitrary and at this point don’t really have much to do with real world values.

The classes I ended up coming up with for this project included several that were intended to simulate various portions of the project that would later have to interact with hardware (or, more likely, hardware drivers). The project required a class to simulate the "black box" control module handed out by DARPA, and a class to simulate interactions with the various mechanisms on the vehicle itself (acceleration/braking, turning, and "GPS" coordinate location of the vehicle). The most code went into the main (AutoDrive) class, because of all of the GUI design involved, but the most work by far went into the Pilot class, since that's where all of the real work gets done in this program. A lot of trouble was caused by using a calculation of difference between the "actual" and "desired" headings of the vehicle, since when the difference would cross from (for example) 1 to 359, the vehicle would reverse it's turn. The problem
appears to have been solved primarily by converting the value of this angle from a range of 0 to 360 degrees to a range of -180 to 180, since that (along with some other calculations) prevents the "rollover" effect. Another problem encountered even before much of the coding was begun was distance-finding: how to calculate whether the vehicle was within a certain distance from the imaginary line drawn from "point A" to "point B." This was, of course, solvable using just trigonometry, but it was a very long and ugly looking function that I felt compelled to check several times for accuracy.

Everything regarding the navigation system was completed by myself. I did talk out some of my problems with the math controlling the vehicle's heading with my fellow student Dan Goudy, but the problems were eventually solved by myself. Beta testing was performed by my friend Joshua Pearson, a student at Shasta Bible College in Redding. Many of the ideas for more easily understandable informational displays (such as the in/near/out boundary percentage counters) were developed after suggestions from him. Friend and cohort Jasen Murphy, a student at CSU, Sacramento is working on implementing much of the vision/object detection system for the second phase of this project.

A lot of this program involved putting into practice techniques and knowledge that I had learned in various classes at Chico State. There was a lot of exploration and learning that took place in trying to develop the algorithms to do the math for my coordinate system, though. Now that I'm working on making the coordinate system relate to actual GPS coordinates, I'm starting to learn about the lovely mathematics required to calculate straight-line distances over a sphere, when one part of the coordinate system varies in distance relative to the other part of the coordinate system. Latitude and longitude look a lot less inviting the more I learn about them.
There are still some minor issues with the program, mainly in corner handling and dealing with too-high speed settings, but all major flaws (that have been found) have been ironed out. Attached are a basic UML diagram of the project, some of my sketches from the early development of the distance-finding equations, the “readme” file from the project website, a few screenshots of the program in operation, as well as a printout of the entire Pilot class, since almost every part of that class was non-trivial to develop.
References

DARPA Grand Challenge website:
http://www.darpa.mil/grandchallenge

DARPA Grand Challenge Rules (includes rddf formatting):

Carnegie-Mellon University “Robot Racing”:
http://www.redteamracing.org/
/**
 * Pilot
 * Takes care of basic steering/navigational functions
 * @author Gabriel Dawson
 * @version 2004.11.29
 */

public class Pilot extends Thread{
    private double nextLat, nextLon, maxV, lat, lon, heading, lastTurn;
    private double corrFactor, r;
    private int index = 0;
    private MapModel mModel;
    private int WAIT = 100;

    public DoDSysm blackBox;
    public double desired = 0;
    public int boundary = 0;

    Pilot(MapModel m, double c){
        blackBox = new DoDSysm();
        mModel = m;
        corrFactor = c;
        maxV = mModel.thisPoint.speed;
        nextLat = mModel.nextPoint.lat;
        nextLon = mModel.nextPoint.lon;
        checkGPS();
    } // </Pilot()>

    public void setDelay(int d){
        WAIT = d;
    }

    public void checkGPS(){
        lat = mModel.toyYoda.getLat();
        lon = mModel.toyYoda.getLon();
        heading = mModel.toyYoda.getHeading();
        if ((Math.abs(lat - nextLat) < Definitions.PCLOSE) &&
            (Math.abs(lon - nextLon) < Definitions.PCLOSE))
            getNextPoint();
    } // </checkGPS>

    public void getNextPoint(){
        index++;
        mModel.incrementPoint();
        maxV = mModel.thisPoint.speed;
        nextLat = mModel.nextPoint.lat;
        nextLon = mModel.nextPoint.lon;
public int nearBounds(){

double opp = (mModel.getOpp(index)==0?.000000001:mModel.getOpp(index));
double adj = (mModel.getAdj(index)==0?.000000001:mModel.getAdj(index));
double phi = Math.atan(opp / adj);

r = (Math.sin(phi) * (lat - mModel.thisPoint.lat - ((adj / opp) *
((lon - mModel.thisPoint.lon))));
double d = mModel.thisPoint.bound - Math.abs(r);
// return "out of bounds"
if (d < 0){
    boundary = 2;
    return boundary;
}
// return "nearing boundary"
if (d <= Definitions.BCLOSE){
    boundary = 1;
    return boundary;
} // </if>
// return "not near boundary"
boundary = 0;
return boundary;
} // </nearBounds>

public double whichWay(double h, boolean correct){

    // be sure to change correction algorithm for use with real GPS rather than
    // on-screen coordinates!! (flip y-axis values==shift quadrants)

double x = nextLat - lat;
double y = nextLon - lon;
// shift target point if truck needs to come back in-bounds
if (boundary == 2 && correct){
    // if already out-of-bounds
    double dx = mModel.nextPoint.lat - mModel.thisPoint.lat;
double dy = mModel.nextPoint.lon - mModel.thisPoint.lon;

    if ((dx > 0 && dy > 0 && r < 0) ||
        (dx < 0 && dy < 0 && r > 0) ||
        (dx < 0 && dy > 0 && r < 0) ||
        (dx > 0 && dy < 0 && r > 0)){
        y *= .25;
    } else{
        x *= .25;
    }
} // </if out-of-bounds>
if (boundary == 1 && correct){
    // if nearing boundary
    double dx = mModel.nextPoint.lat - mModel.thisPoint.lat;
double dy = mModel.nextPoint.lon - mModel.thisPoint.lon;

    if ((dx > 0 && dy > 0 && r < 0) ||
        (dx < 0 && dy < 0 && r > 0) ||
        (dx < 0 && dy > 0 && r < 0) ||
        (dx > 0 && dy < 0 && r > 0)){
        y *= .25;
    } else{
        x *= .25;
    }
} // </if nearing boundary>
if ((dx > 0 && dy > 0 && r < 0) ||
(dx < 0 && dy < 0 && r > 0) ||
(dx < 0 && dy > 0 && r < 0) ||
(dx > 0 && dy < 0 && r > 0)) {
  y *= .7;
} else{
  x *= .7;
}
} // </if near-bounds>

// calculate angle to the next waypoint
desired = Math.abs(Math.toDegrees(Math.atan(y / x)));  

// adjust for using coordinate system: (-90 < atan < 90)
if (x < 0 && y > 0)
  desired = 180 - desired;
if (x < 0 && y < 0)
  desired += 180;
if (x > 0 && y < 0)
  desired = 360 - desired;

// switch from 0/360 system to -180/180 system
if (desired > 180) desired -= 360;
if (h > 180) h -= 360;

// calculate difference between actual angle and angle to next waypoint
double offset = desired - h;
// correct for "wraparound" values
if (offset > 180)
  offset -= 360;
if (offset < -180)
  offset += 360;

// provide correction factor, if desired (prevent overcorrection)
if (correct){
  if (Math.abs(offset) < Math.abs(lastTurn * corrFactor)){
    double l = offset;
    offset -= lastTurn;
    lastTurn = l;
  } // </if offset < lastTurn>
  else
    lastTurn = offset;
} // </if correct>

double dir = offset;

if (Math.abs(offset) > 50)
  return (Definitions.HARD * (dir < 0?Definitions.LEFT:Definitions.RIGHT));
else if (Math.abs(offset) > 30)
  return (Definitions.TURN * (dir < 0?Definitions.LEFT:Definitions.RIGHT));
else if (Math.abs(offset) > 6)
return (Definitions.VEER * (dir < 0?Definitions.LEFT:Definitions.RIGHT));
else return dir < 0?Definitions.LEFT:Definitions.RIGHT;
} // </whichWay>

public void run(){
  checkGPS();
  while (blackBox.state != blackBox.STOP){
    while(blackBox.state == blackBox.PAUSE){
      try{
        sleep(100);
      } catch(Exception e){
        System.out.println("Thread interruption error " + e);
      } // </try-catch>
    } // </while PAUSE>
    // turn steering wheel
    mModel.toyYoda.turn(whichWay(mModel.toyYoda.getHeading(), true));
    double w = Math.abs(whichWay(mModel.toyYoda.getHeading(), false));
    double topSpeed = maxV;
    // drop topSpeed if not heading right direction
    if (w > Definitions.VEER)
      topSpeed /= 3;
    // drop topSpeed if coming up on a possible corner
    if (Math.sqrt((lat - nextLat) * (lat - nextLat) + (lon - nextLon) * (lon - nextLon)) <= (Definitions.PCLOSE * 4)){
      topSpeed /= 3;
    }
    int n = nearBounds();
    if (n > 0){
      if (w >= Definitions.TURN)
        // slow down if "near" bounds and not heading right direction
        mModel.toyYoda.correctSpeed(0 - (maxV / 3));
      else if (n == 2){
        if (w <= Definitions.VEER)
          // speed up if out of bounds and heading right direction
          mModel.toyYoda.correctSpeed((topSpeed / 4) - mModel.toyYoda.getSpeed());
        else
          // slow down if out of bounds and not heading right direction
          mModel.toyYoda.correctSpeed(0 - (maxV / 4));
      }
    else
      // speed up if "near" bounds and heading right direction
      mModel.toyYoda.correctSpeed((topSpeed / 2) - mModel.toyYoda.getSpeed());
} // </if not "in" bounds>
else
  // speed up if "in" bounds
  mModel.toyYoda.correctSpeed(topSpeed - mModel.toyYoda.getSpeed());
  mModel.toyYoda.run();
  checkGPS();
  try{
    sleep(WAIT);
  }
} catch(Exception e){
    System.out.println("Thread interruption error " + e);
} // </try-catch>

} // </while>

} // </run>

} // </Pilot>