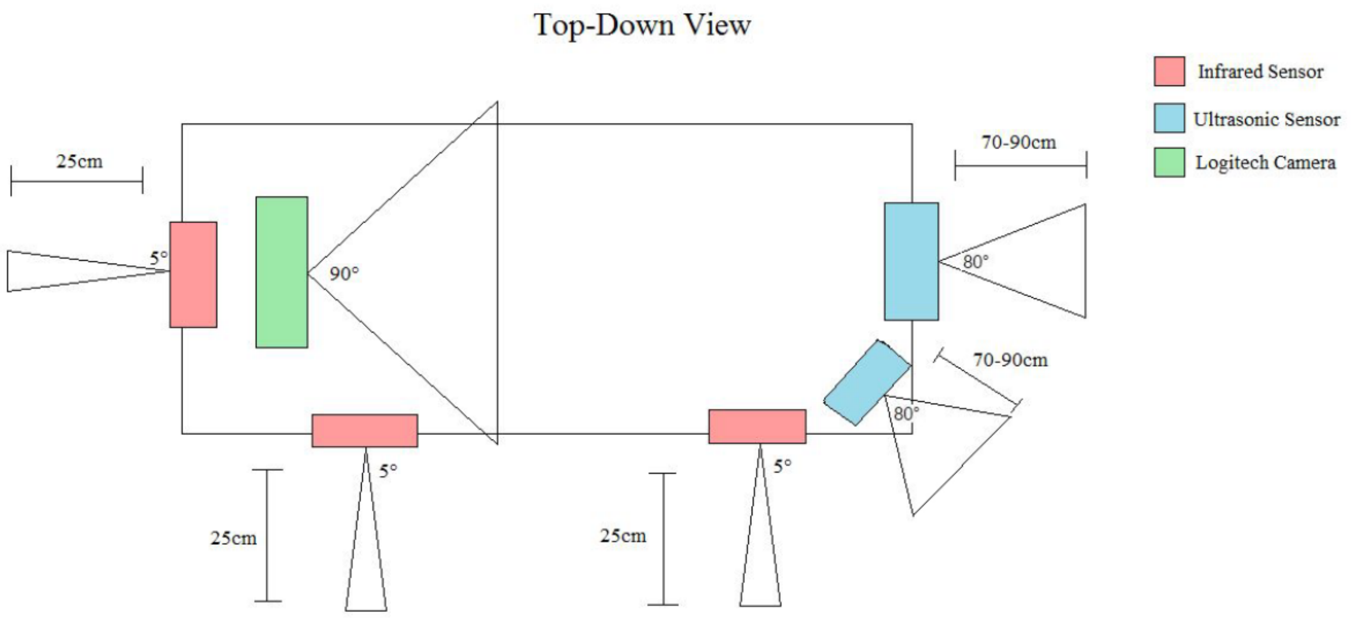
Smart Car Revised Architecture

Kai Salmon, Tobias Lindell, Martina Freiholtz, Kristiyan Dimitrov,
Johannes Flood, Junjie Cheng, Danielle Serafim

March 2016

1 System Architecture

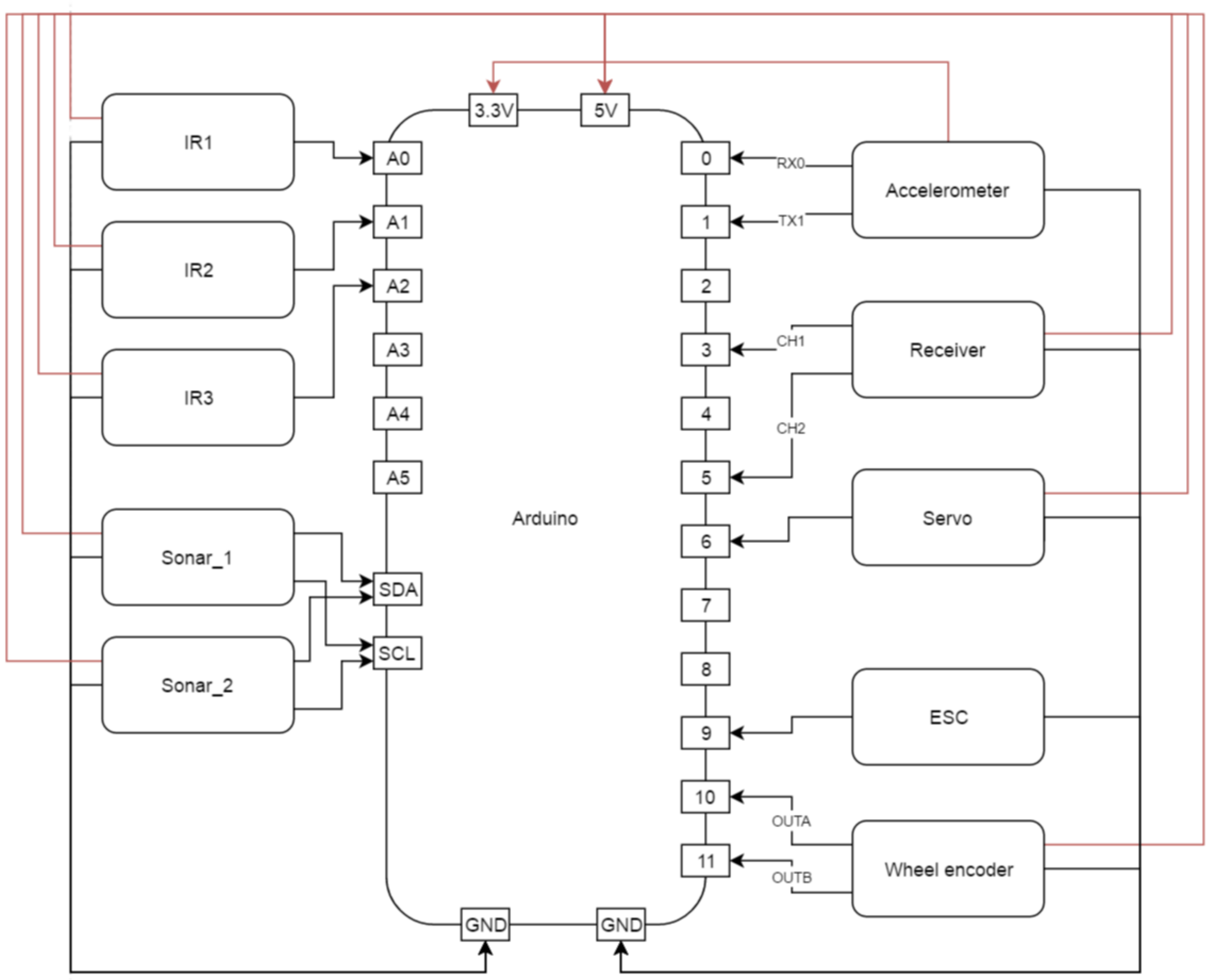
1.1 Bird's eye view sensor layout



There are two ultrasonic sensors in the front of the car, two infrared sensors on the right side of the car, and one on the back. Furthermore there is a camera looking forward giving a large view in front of the car. The infrared sensor on the back will primarily be used for the parking behaviour. It will keep track of when the car should stop when backing into the parking spot. The camera will be used primarily for the lane following process. The images recorded by the camera will be run through filters and analyzed to determine how the car should behave. The rest of the sensors will be used for both the overtaking and the parking behaviours. The side sensors will be used to determine when the car is finished overtaking as well as determining when there is an open parking spot.

1.2 Sensor and actor components

1.2.1 Block Diagram



Block diagram describing power, ground, and signal connection on Arduino.

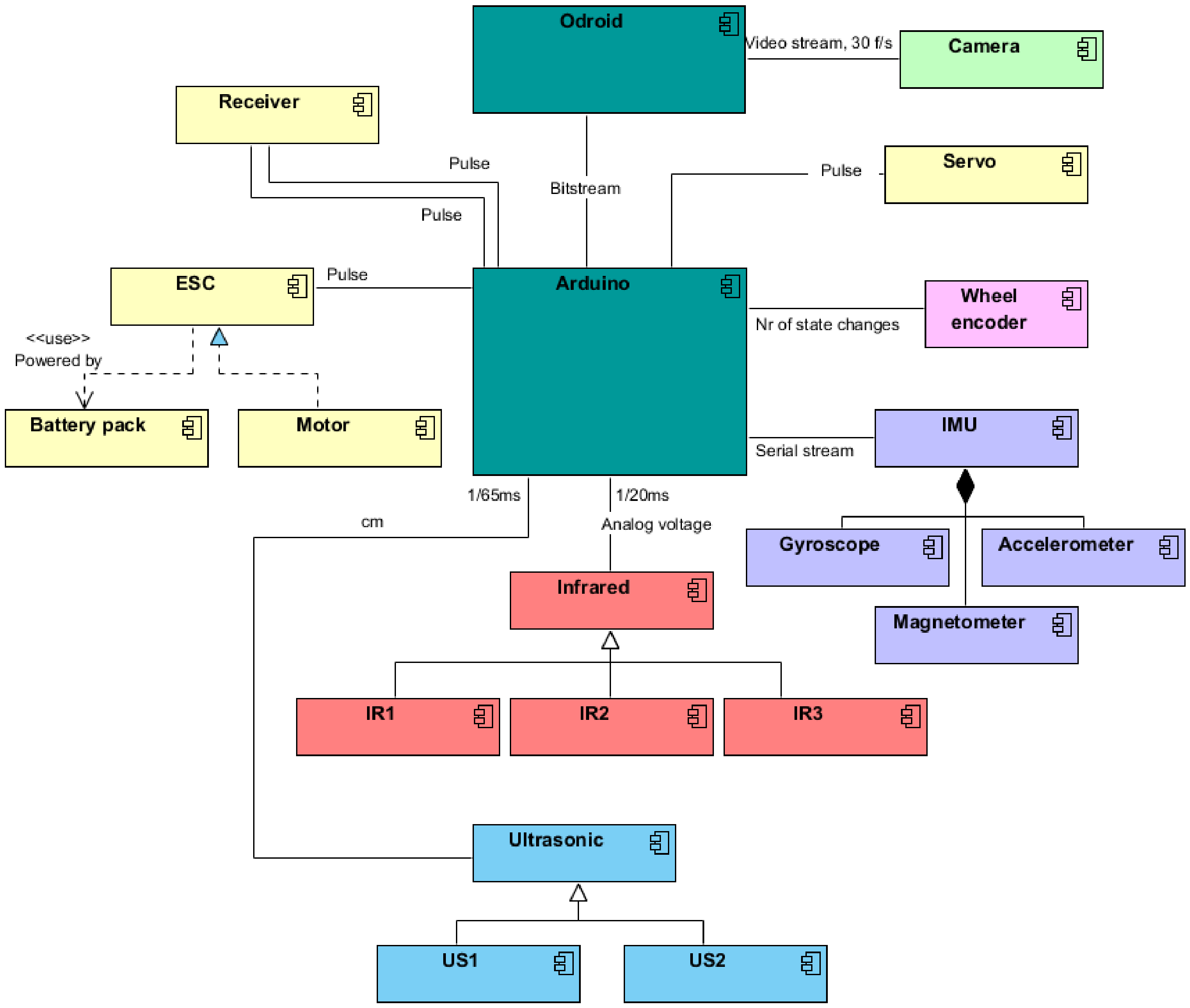
Most of the components are powered with 5 volts through the Arduino. Exceptions to this are the IMU (powered with 3.3 volts) and the ESC, which is powered by a battery pack.

For the ultrasonic sensors, the special data line (SDA) and clock line (SCL) pins are used. The IR sensors outputs analogue values.

We use two channels for the receiver, where channel 1 is connected to an interrupt pin (3). When a remote control is switched on, the change in input from channel 1 causes an interrupt which allows us to control the car manually.

The RX (receiver) of the IMU is hooked up to an TX (transmitter) pin on the Arduino (1) and vice versa (the IMU's TX to Arduino's RX, pin 0).

1.2.2 Sensors and Actors



The camera, which is the only sensor that is connected directly to the Odroid board, sends a 1080p video stream. It is capable of sending 30 frames per second.

The servo as well as the ESC are controlled by sending pulsing signals, adjusting the duration in accordance to the wished speed and direction. The ESC is driven by the battery pack, and controls the motor. If a remote controller is switched on, the signals from the receiver are directed (through the Arduino board) to the ESC and motor, respectively.

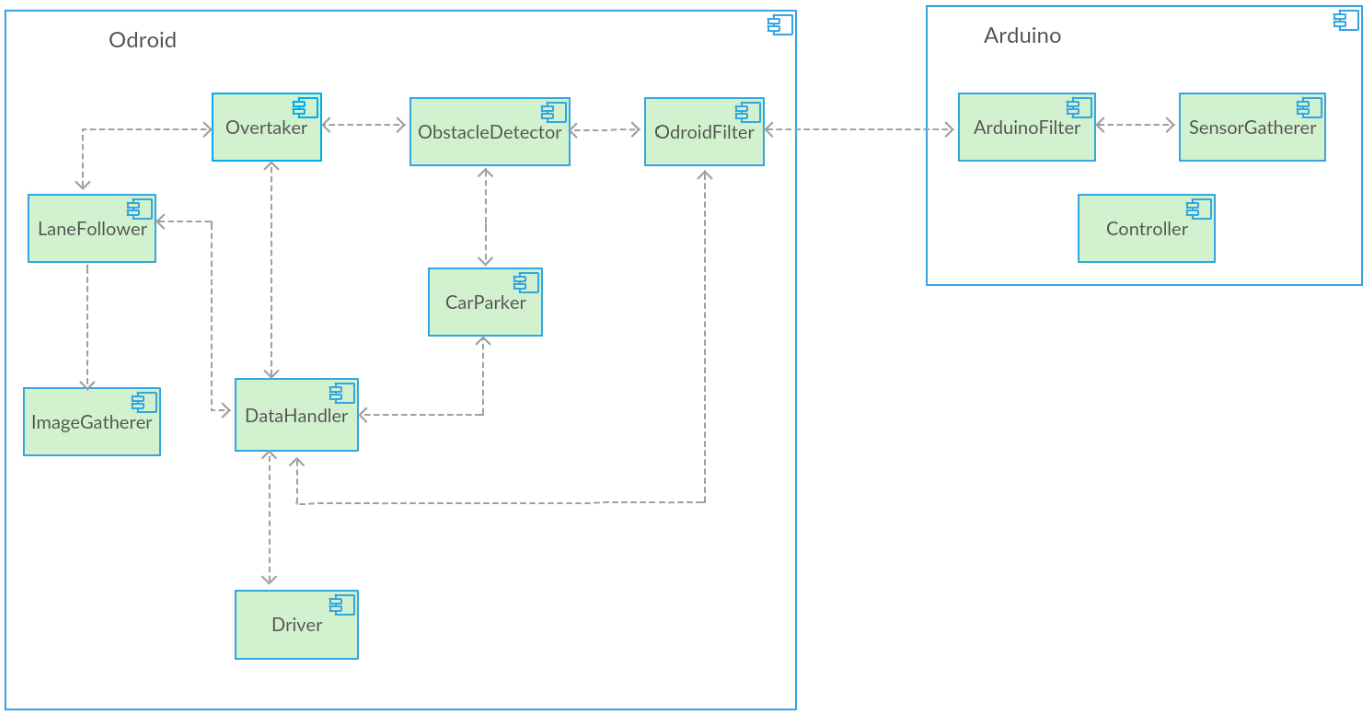
The infrared sensors outputs analogue voltage (indicating the distance to the closest detected object) once every 20 milliseconds. The ultrasonic sensors are programmed to output the distance in cm to the nearest detected object (if an object is detected, 0 otherwise), and does this once every 65 milliseconds in a best case scenario.

The IMU, which contains three separate sensors, outputs a serial stream with the combined data from the gyroscope, accelerometer and the magnetometer. The wheel encoder outputs the number of state changes recorded (or 'counts').

The Arduino and Odroid boards are connected via USB, and a bitstream with data packets containing sensor data is sent to the Odroid board when prompted. These data packets are separated by delimiters and contains a control bit to check for missing and/or corrupt data. The stream can be interrupted to send data in the other direction, as to change the output to the ESC and servo when the car enters and executes a behaviour.

2 Revised System Architecture

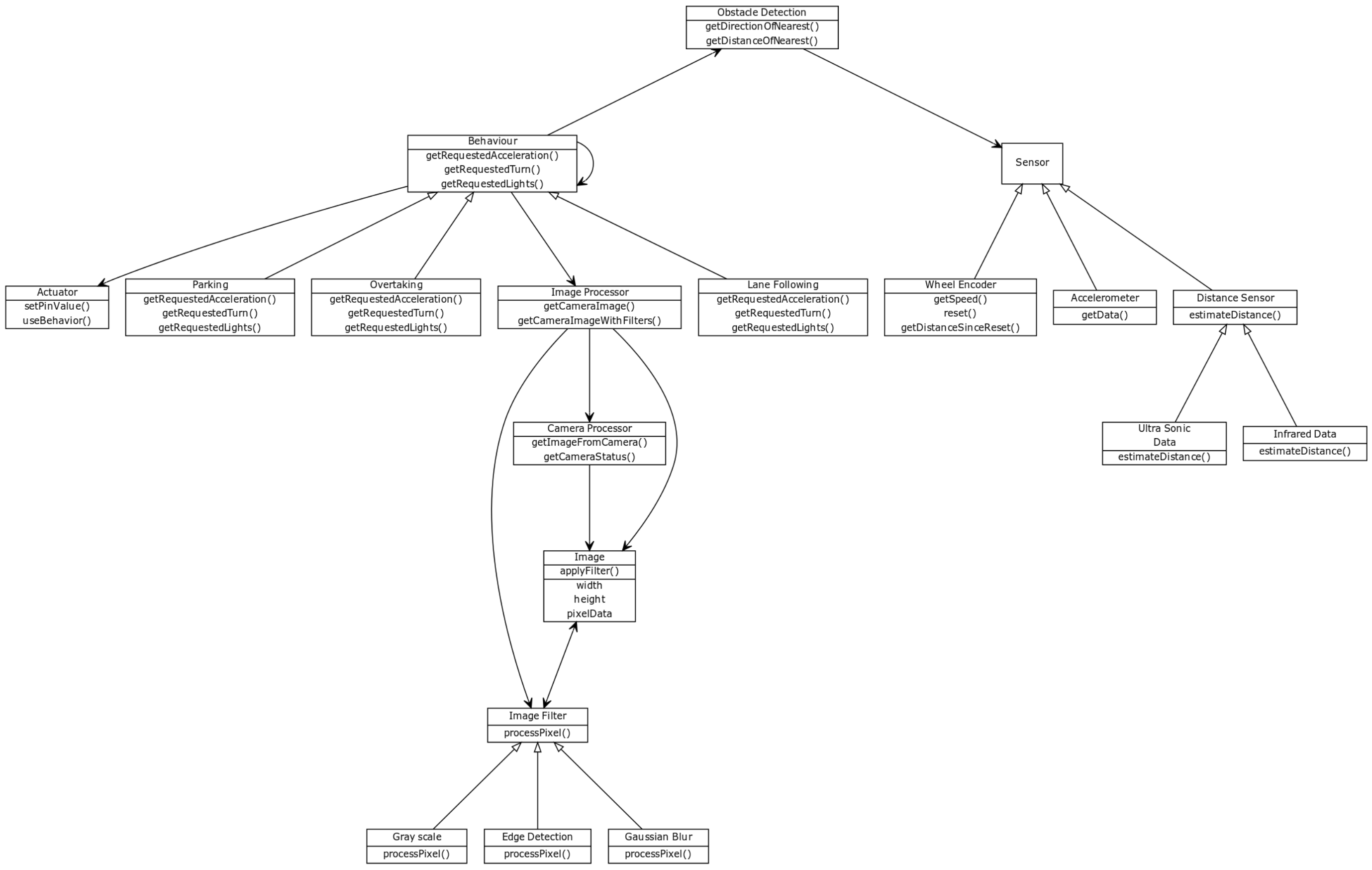
2.1 Component Diagram



OdroidFilter: This component is responsible for gathering and unpacking the data from the Arduino side. It is also responsible for packing and sending data to the Arduino.

ObstacleDetector: This component is responsible for calculating the distance from the obstacles.
Overtaker: This component is responsible for estimating the necessary turn in order to overtake a car. It uses the obstacles distance/direction for that.
LaneFollower: It uses the live image from the camera to calculate the lane images with all edges highlighted in white.
ImageGatherer: This component contains the classes responsible for processing and filtering the image from the camera.
CarParker: It is responsible for calculating the parking space and parking maneuver.
DataHandler: The data handler is responsible for collecting and filtering the data we want according to the Driver state. **Driver:** The driver requests the data as it changes behaviour such as following lane, overtaking and parking.
ArduinoFilter: This component is responsible for unpacking and gathering the data from the Odroid side. It is also responsible for packing and sending data to the Odroid.
SensorGatherer: It gathers all data from the sensors.
Controller: The controller uses output pins to control the motor and servo hardware.

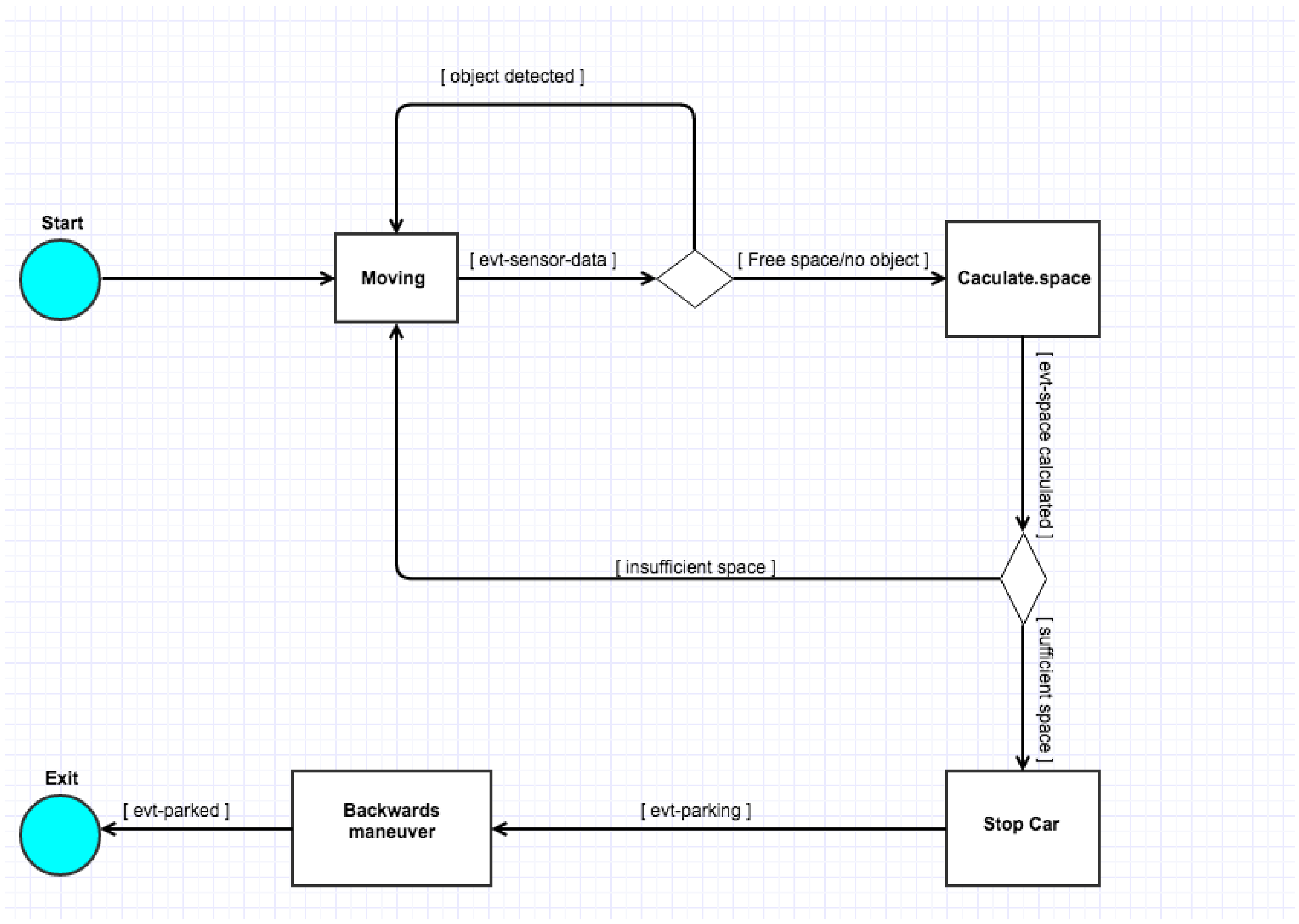
2.2 Class Diagram



In this current version of the class diagram, the team had a much clearer idea of what software entities would be interacting with each other. For example, the behaviour class will make it much easier to get the required action that the car should take depending on which behaviour is active. Additionally, the new Actuator class will be the only one interfacing directly with the Arduino board, and will drive the car through the pins after the data has been analyzed through the behaviours.

3 Parking Behaviour

3.1 Planned State Machine



The start point of the car is it’s idle, off state. Once the car is started it changes into it’s moving state. The car receives sensor data from the sensors on a time-period basis and changes state or remains in moving state depending on the data received. If an object is detected the car remains in a solely moving state, on the other hand if free space is detected, it will change into the calculate space state. Once the calculation is complete, it’ll decide if the space is sufficient or insufficient, depending on the outcome it will either return in the moving state (insufficient space found) or in the stop car state (sufficient space found). The stop car state is not immediately stopping the car as the name suggests, instead it moves the car forward until there is sufficient space for the car to perform a backwards maneuver and then stops the car. Afterwards the car enters the

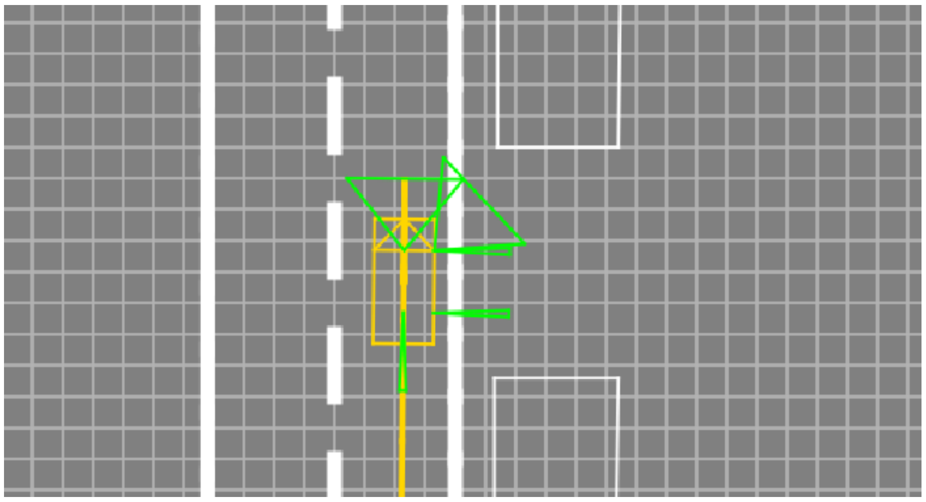


Figure 1: Parking with Sufficient Space

event of parking which will change the the state into the backwards maneuver state. The backwards maneuver state does a backwards maneuver which parks the car in the free space between the two objects, stops it complete, and sends an event that the car is parked which terminates the process.

3.2 Overview of planned and modeled test scenarios

The first test case we want to evaluate is when normal circumstances are met. This happens when the sensors detect an empty space to the right of the car which is large enough to park in. In order to pass this test, the car should go through the states without looping back into any previous states. The sensors will calculate distance travelled from when the space is detected, to when a new obstacle is found. When the space is deemed to be enough for a parking, the car will stop and perform a predetermined backwards maneuver to park in the spot.

The second test case is when the car detects a potential parking space, but the space is not large enough for parking. To pass the test, the car will initialize the parking behaviour like normal. Once it finishes measuring the empty spot and it identifies it as being too short, the car will revert into the "moving" state and continue searching for a parking spot.

The final test case we need is when the space for parking is large enough, but there is an obstacle inside the space which inhibits the car from parking. In order to pass this test, the car should determine that the change in sensor data does not mean that there is a potential parking space. The car should instead continue searching for an open space like normal. The behaviour needs to be calibrated in a way so that the car does not react to small changes in sensor data.

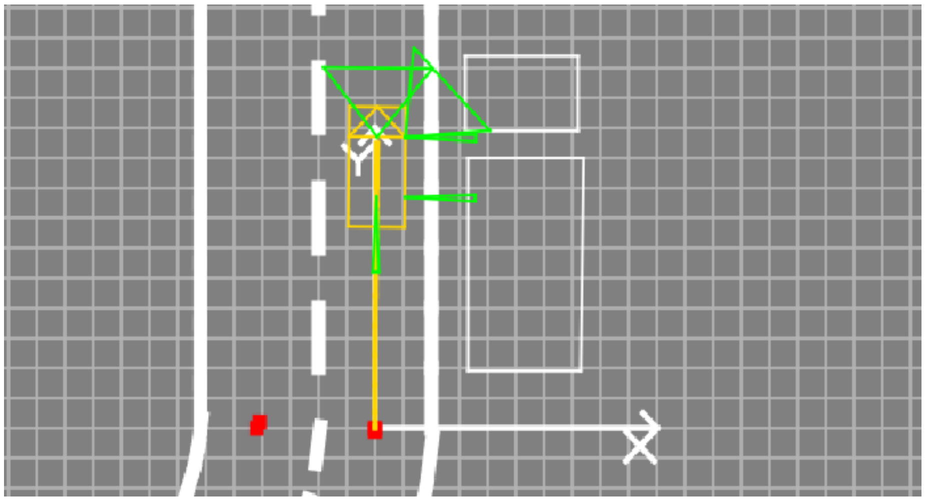


Figure 2: Parking with Insufficient Space

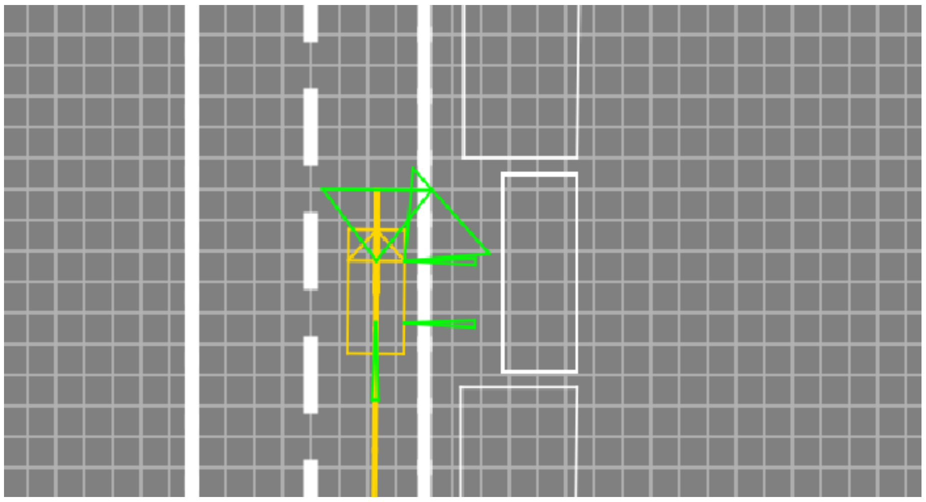
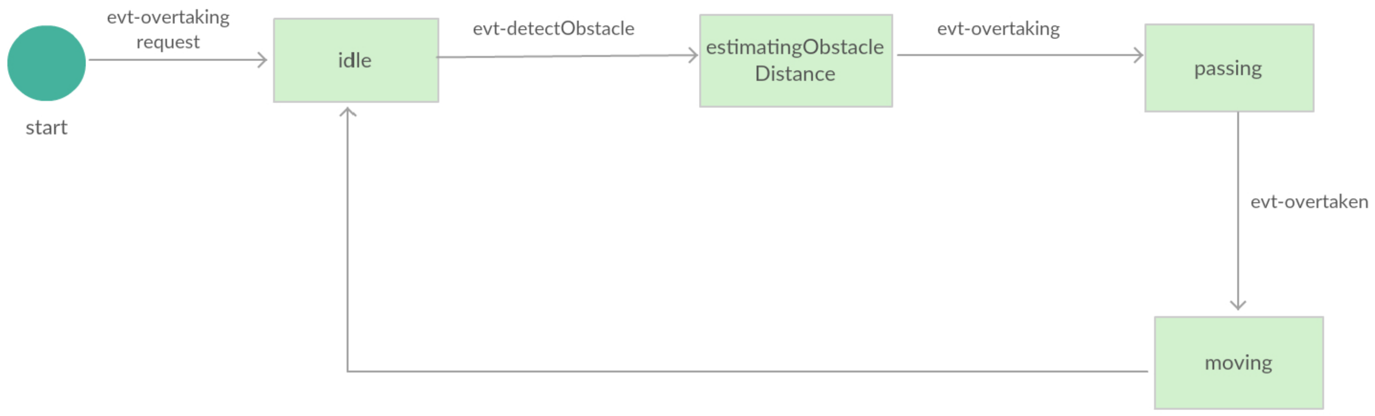


Figure 3: Parking with an Obstacle in Space

4 Overtaking Behaviour

4.1 Planned State Machine



Unlike many state diagrams, the car is not actually inactive during the idle state. Until the overtaking request is sent, the overtaking behaviour is active and listening for a request just like the other behaviours. Normally, the car will be engaged in the lane following process at this time. The overtaking process is first activated when the sensors detect an obstacle ahead of the car in the lane. Once the distance to that obstacle is measured, and the car as approached it to a certain distance, the car will start overtaking the obstacle. Once the sensors have detected that the obstacle is no longer in the way on the right hand side of the vehicle (overtaken), the car will turn back into its lane and return to the original position and continue lane following. Because the car returns to the lane following process after the overtaking is completed, the overtaking process can also start from the beginning and wait for a new obstacle, making it an infinite loop.

4.2 Overview of planned and modeled test scenarios

The first test case we want to evaluate is when the normal overtaking circumstances are met. This situation arises when the overtaking behaviour is called with only one obstacle ahead with a clear lane to the left of the obstacle. In order to pass the test, the car needs enter the following states without any of the sensors reading obstacles closer than 2 centimeters (which we are interpreting as a collision) or stopping: idle, estimatingObstacleDistance, passing, and moving. The simulation will be using a straight road, with a single obstacle in the same lane as the car with similar size to the car. To be in the idle state, the car should be in lane following mode. It will change states when the front sensor finds an obstacle. After that, it needs to turn into the left lane and continue following the lane normally while reading the values of the right IR sensors. When the back right IR sensor sees no more obstacles, the car will turn back into the right lane and continue using the lane following process.

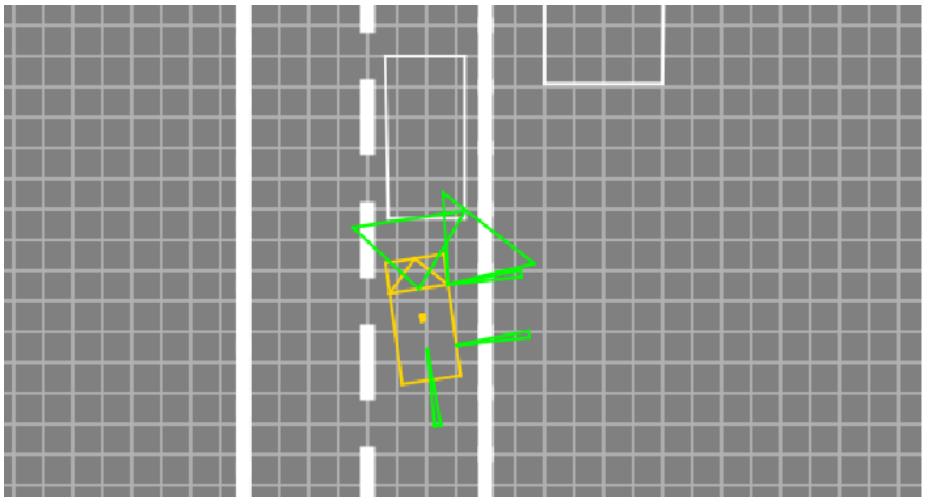


Figure 4: Overtaking on a straight road

The second test case is for when there is an obstacle in the way in front of the initial obstacle. The test is passed if the car manages to go into lane following mode after passing the secondary obstacle without colliding with either obstacle. The simulation will use a straight road with two separate obstacles in same lane as the car. The second obstacle will be smaller than the first. The second obstacle might not be picked up at all by the IR sensors at first, or might have different readings from the main obstacle. The car needs to behave normally when overtaking while the right facing IR sensors read a short distance. (Meaning there is still an obstacle there.) The car will only behave differently once the initial obstacle has been passed. If the secondary obstacle is not seen by the sensors at all, the car will move into the passed state, then instantly go back into passing when it senses the obstacle. When the sensors pick up the secondary obstacle, it should continue going straight along the lane until the second obstacle is passed as well. If the second obstacle is detected right away after the first one is passed, the car should not exit its passing state until the second one is passed as well.

Finally, we need a test case for when there is an obstacle in a curve. The test is passed if the car stays in the idle state for the entirety of the curve and doesn't collide with any obstacles. The car then continues with the other states after the road continues in a straight line. The simulation will contain a curved road followed by straight road. The road will turn 90 degrees with a 50 centimeter radius.

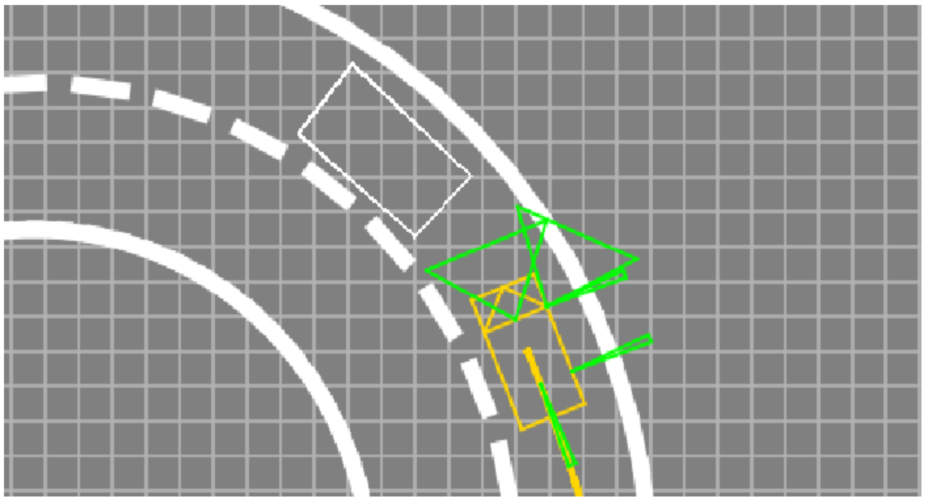


Figure 5: Overtaking on a curve

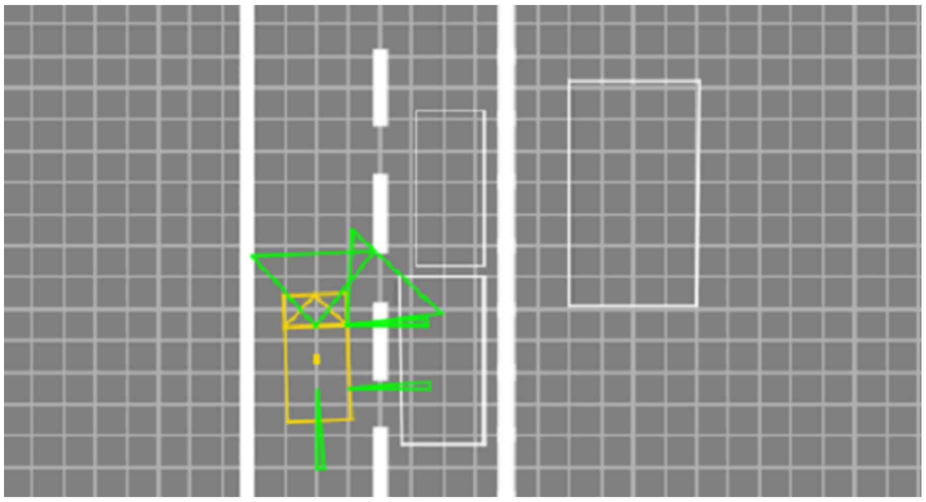


Figure 6: Overtaking two obstacles

5 Lane Following Behaviour

5.1 General Concept

The Lane-Following behaviour of the SmartCar will have two major components. First, we will take the image from the live camera and pass it through a number of Image Filters. These filters, in order, are 'Gaussian Blur', 'Edge Detection', and 'Canny Edge Detection'.

The Gaussian Blur filter blurs the pixel data in the image to reduce noise. For each pixel, the filter will take an weighted-average of all pixels within in some distance from the current pixel with weighting such that pixels further from the current pixel have lower effect on the resulting pixel.

The Edge Detection filter considers each pixel of an image, and outputs a value proportional to the absolute difference between the current pixel and each of it's neighbours. This means that if a given pixel in the input image is identical to it's neighbours (and therefore not part of an edge), the corresponding pixel in the output image will have the value zero. Conversely if a pixel is surrounded by pixels with a very different value, it will output a pixel of very high value.

Finally, the Canny Edge Detection takes the output of the previous filter. For each pixel in the image it's "Intensity Gradient" is calculated. This can be visualized by taking the image as a height-map (That is, a 3D surface were each pixel is a point, and the brighter the point the higher that point is on the surface). For each pixel, its "Intensity Gradient" will be angle at which the surface "slopes" downwards. For each pixel, the "positive direction" is the direction in that angle, and the "negative direction" is the opposite direction. For each pixel, the adjacent pixel in both the current and negative directions are considered. If it is the current pixel is greatest of the three pixels, the output is white. Otherwise the output is black.

The result of these filters should be an image with all edges highlighted in white, and with a sufficiently low resolution image, this calculation should be possible to achieve in real time. The simplest way to use this image to support the lane following behaviour is to take a pixel in the centre of the image, and find the distance from that pixel to the first white pixel in both the positive X and negative X directions. If the positive X direction has a lower distance, than we should turn left, and otherwise should turn right. This approach can be improved by taking multiple samples down the centre line of the image, and take an average.

5.2 Overview of planned and modeled test scenarios

The first test case we want to evaluate is the base case of driving on a straight road, with no obstacles. The road layout will be a single straight road. To pass the test the car should not steer in any direction, and move forward. It should not touch the lane markings on either side of the lane, or cross into the next lane.

The second test will be to evaluate the case of driving on a curve. The road

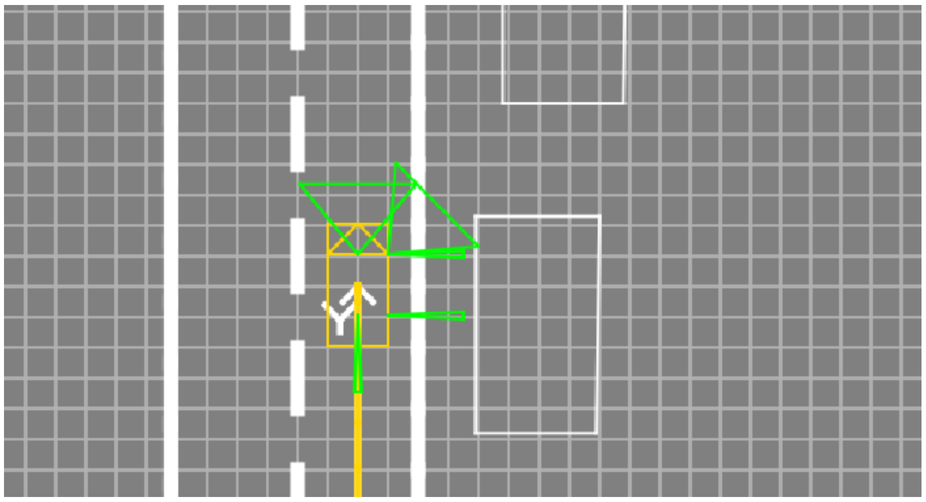


Figure 7: Lane Following on a straight road

layout will be one straight road; followed by a 90 degree curve to right, with a radius of 50cm; followed by another straight road. In order to pass the test the car should not touch the lane markings on either side of the lane, or cross into the next lane, and should progress down the straight, around the corner, and down the second straight.

The third and final test will be evaluate the case of driving across and intersection. The road layout will have a two large straights crossing at an appropriately marked intersection point. In order to pass the test the car should drive through the intersection and continue to the other side of the straight. As above, the car should not touch the lane markings on either side of the lane or cross into the next lane.

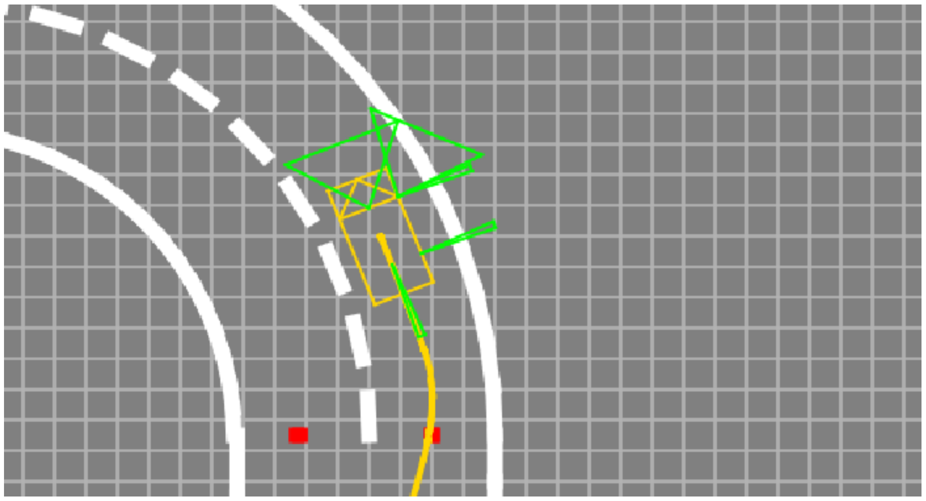


Figure 8: Lane Following on a curve

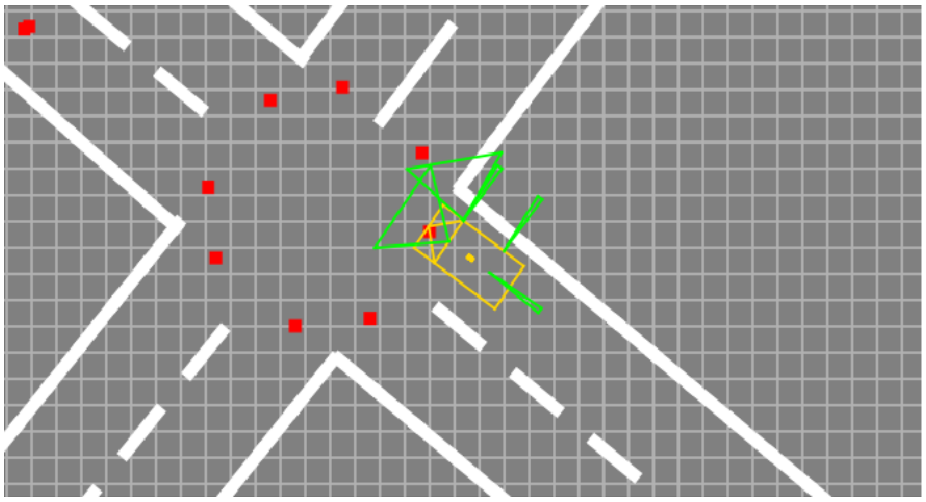


Figure 9: Lane Following through an intersection