Introduction to Image Processing and DRC Vehicle Control

Wyatt Newman

February, 2013

These notes introduce example code for performing image processing from Atlas image topics and using this information for real-time control—steering of the DRC simulated vehicle.

A simple snapshot program with OpenCV: The accompanying package, "snapshot", contains ROS code that illustrates how to use OpenCV with ROS. OpenCV is a long-standing, open-source computer-vision library that performs useful image-processing operations. A good on-line reference can be found here:
http://www.amazon.com/gp/product/1849513244/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1849513244&linkCode=as2&tag=opencv00-20

The example package illustrates how ROS interfaces to OpenCV, and it performs the useful function of saving images (from a ROS topic) to disk as *.BMP files. The example program subscribes to a ROS image topic, `//multisense_sl/camera/right/image_rect_color` by default, but optionally an alternative image topic by use of a command-line argument when executing the program. "Snapshot" also re-publishes the image to the new topic "snapshot_out." The user is prompted to enter "1" (take a snapshot) or "0" (quit the program).

The corresponding CMakeLists.txt and manifest.xml documents include the necessary dependencies for compilation. Using the provided package, compile the code (as usual) with:

```
rosmake snapshot
```

Functionally, this program is similar to using the ROS program:

```
rosrun image_view image_view  image:=
//multisense_sl/camera/left/image_rect_color
```

(for which a right-click on the displayed image window will save a snapshot).

Since "snapshot" (by default) expects an Atlas image topic, launch the simulator, e.g. with:

```
roslaunch atlas_utils atlas_drc_vehicle_fire_hose.launch)
```

(so Atlas has something interesting to look at).

Before running the program, one should change directories to where it is desired to save the snapshots. From this directory, do: `rosrun snapshot snapshot`. The program will subscribe to the default topic (right eye of Atlas's head) and prompt the user. Each time "1" is entered, the most recent image will be saved to disk as a *.BMP file, called: snapshotN.bmp (where "N" is the current count of images saved).

To specify an alternative camera topic, include a command-line argument.   E.g., to get rectified color images from the left eye, do:

```
rosrun snapshot snapshot in:=//multisense_sl/camera/left/image_rect_color
```

You can also view the saved images by subscribing to the topic "snapshot_out". To do so, in another terminal, run:

```
  rosrun image_view image_view image:=/snapshot_out
```

The snapshot code: The program "snapshot.cpp" is included. Explanation of some key lines follows. In the class constructor, the callback function is set to subscribe (by default) to the topic of the Atlas head's right camera, using:

```
image_sub_ = it_.subscribe
("//multisense_sl/camera/right/image_rect_color", 1, &ImageConverter::imageCb, this);
```

To integrate openCV with ROS, it is necessary to translate between the ROS message type for images and the image format required by openCV. This is done with a "bridge" operation as follows:

```
cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
```

At this point, the cv_ptr is a pointer to an image type compatible with openCV.

The line:

```
imwrite(savename, cv_ptr->image);
```

uses the openCV function "imwrite()" to save the image to disk.

The openCV image is converted back to ROS message type and published with the line:

```
image_pub_.publish(cv_ptr->toImageMsg());
```

This code shows the necessary steps for translating between ROS and openCV formats, and use of a simple openCV function (imwrite). However, the power of openCV has not yet been illustrated. The sample code follows:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <cv.h>
#include <highgui.h> // need this for imwrite()
#include <iostream>
using namespace cv;
namespace enc = sensor_msgs::image_encodings;
int trigger;
int savenum;
bool default_input;
```

```
class ImageConverter
{
// member variables
ros::NodeHandle nh_, priv_nh_;
image_transport::ImageTransport it_;
image_transport::Subscriber image_sub_;
image_transport::Publisher image_pub_;
int image_num;
public:
ImageConverter()
: nh_(),
priv_nh_("~"),
it_(nh_)
{
// We'll publish the resulting image on "out" and subscribe to "in"
image_pub_ = it_.advertise("snapshot_out", 1);
//default to topic: //multisense_sl/camera/right/image_rect_color
if (default_input) {
image_sub_ = it_.subscribe
("//multisense_sl/camera/right/image_rect_color", 1, &ImageConverter::imageCb, this);
ROS_INFO("using default input of //multisense_sl/camera/right/image_rect_color");
ROS_INFO("to use an altenative input topic, run as: rosrun snapshot snapshot in:=//topic_name");
}
else
image_sub_ = it_.subscribe("in", 1, &ImageConverter::imageCb, this);
}
~ImageConverter()
{
}

void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
char head[] = "snapshot";
char tail[]=".bmp";
char savename[256];
char snapnum[128];
strcpy(savename,head);
strcat(savename,tail);
cv_bridge::CvImagePtr cv_ptr;
try
{
// Create a copy of the image in msg, requesting that the image be converted to BGR8 if it isn't already
// We make a copy so that we can operate in place on it... if we wanted a readonly
copy, we could use toCvShare to avoid extra memcpy ops
cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
}
catch (cv_bridge::Exception& e)
{
ROS_ERROR("cv_bridge exception: %s", e.what());
return;
}
savenum++;
sprintf(savename,"%s%d%s",head,savenum,tail);
ROS_INFO("saving snapshot to %s", savename); //snapshot.bmp");
//imwrite("snapshot.bmp", cv_ptr->image); //wsn
imwrite(savename, cv_ptr->image); //wsn
// Publish the modified image
image_pub_.publish(cv_ptr->toImageMsg());
trigger= 0;
}
};

int main(int argc, char** argv)
{
default_input=true; //use default input, unless command-line arg is provided
if (argc==2)
default_input=false;
ros::init(argc, argv, "snapshot");
savenum=0;
ROS_INFO("wsn snapshot pgm: ");
ROS_INFO("do: rosrun image_view image_view image:=/snapshot_out to see the snapshots");
trigger=0; // by default, don't save image
ImageConverter ic;
while (true)
{
ROS_INFO("enter 1 to save image, 0 to quit:");
std::cin>>trigger;
if (trigger==0)
return 0;
ros::spinOnce();
}
return 0;
}
```

Identifying key colors: The saved images can be examined with another program. E.g., you can start "gimp" (if installed, will be under Applications->Graphics->GIMP Image Editor). In the GIMP main window, do `File->Open...` and navigate to an image of interest and open it.

In the GIMP Toolbox panel, you can select the Color Picker Tool (looks like an eye dropper),
In the Color Picker panel,
  click "sample average",
  increase the radius to some desired value for sampling
  click "Pick only"
  click "Use info window"

In the loaded/displayed image, your mouse will look like an eye dropper. Click on some region, and the Color Picker Information window will show the color information (e.g., clicking on the sky will yield something like Red: 144, Green: 161, Blue: 212).

This color information can be used to design a program to recognize regions by distinctive coloring.

A sample openCV image-processing program: The package "opencv_sample" extends the "snapshot" example. This code subscribes to a ROS image topic (again, Atlas' right camera, by default) and converts received ROS messages into openCV-compatible images. These images are processed pixel-by-pixel to test if they are sufficiently close in color to some target (target_color[], hard-coded in "main()"). If a pixel is close enough to the target color (as measured by a "Manhattan distance" in color space, relative to a specified threshold value), then in the output image, that pixel is converted to pure white (255,255,255). If the pixel fails the test, it is converted to pure black (0,0,0).

The user is prompted to enter to enter a value for the threshold for the color-matching test. If the selected value is 0, the program halts.

Using the provided package, compile the code (as usual) with:

```
  rosmake opencv_sample
```

Run the code with:

```
  rosrun opencv_sample opencv_sample
```

The resulting output can be viewed on the topic "opencv_sample_out" by running (from another terminal):

```
  rosrun  image_view image_view image:=/opencv_sample_out
```

The By changing the threshold value interactively, one can choose an appropriate tolerance for color matching. To change the target color, this program would have to be edited to change the assigned values of `target_color`, and recompiled.

Within the code, the line:

```
dist = abs(pixel_color[0]-target_color[0])+ abs(pixel_color[1]-target_color[1])+ abs(pixel_color[2]-
target_color[2]);
```

computes the intensity difference of the blue, green and red channels, and adds them together (thus the "Manhattan" distance). This distance is compared to the threshold to decide if pixels should be turned white or black, with the following code:

```
if (dist>thresh) {
cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 0;
cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 0;
cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 0;
}
else {
cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 255;
cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 255;
cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 255;
}
```

Should where "i" is the column index and "j" is the row index. Every pixel (i,j) is evaluated and changed with the above logic.

The resulting B/W image is published to the (new) ROS topic "opencv_sample_out" with the line:

```
image_pub_.publish(cv_ptr->toImageMsg());
```

Note that in the code body, the following line is commented out:

```
// image dilation example:
cv::dilate(cv_ptr->image, cv_ptr->image, cv::Mat(), cv::Point(-1,-1), 10);
```

This line of code performs the image-processing operation of "dilation." It may be uncommented (and the color-matching block may be commented out) to see how this works. The image-dilation example is more representative of openCV processing. In fact, the color-matching code example provided could be significantly faster using "iterators", and the code could be better organized using classes. (See the referenced openCV2 manual for more information).

The *.cpp file for this example follows:

```
// example program using opencv
// by default, grabs image data from ROS topic //multisense_sl/camera/right/image_rect_color, though
// the source is a command-line argument option
// This example single-steps with user input for a threshold
// A hard-coded target color is defined in this code
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
using namespace cv;
namespace enc = sensor_msgs::image_encodings;
int thresh=30;
bool default_input;
cv::Vec3d target_color; // FILL IN THE VALUES OF THE TARGET COLOR IN main
(); HARD CODED IN THIS EXAMPLE
class ImageConverter
{
// member variables
```

```
ros::NodeHandle nh_, priv_nh_;
image_transport::ImageTransport it_;
image_transport::Subscriber image_sub_;
image_transport::Publisher image_pub_;
int image_num;
public:
ImageConverter()
: nh_(),
priv_nh_("~"),
it_(nh_)
{
// We'll publish the resulting image on "out" and subscribe to "in"
image_pub_ = it_.advertise("opencv_sample_out", 1);
//default to topic: //multisense_sl/camera/right/image_rect_color
if (default_input) {
image_sub_ = it_.subscribe
("//multisense_sl/camera/right/image_rect_color", 1, &ImageConverter::imageCb, this);
ROS_INFO("using default input of //multisense_sl/camera/right/image_rect_color");
ROS_INFO
("to use an altenative input topic, run as: rosrun opencv_sample opencv_sample
in:=//topic_name");
}
else
image_sub_ = it_.subscribe("in", 1, &ImageConverter::imageCb, this);
}
~ImageConverter()
{
}


void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
cv_bridge::CvImagePtr cv_ptr; //use cv_ptr->image in place of "image" in OpenCV manual
int dist; //color distance
cv::Vec3d pixel_color; //variable to hold a pixel color for comparison
try
{
// Create a copy of the image in msg, requesting that the image be converted to BGR8 if it isn't already
// We make a copy so that we can operate in place on it... if we wanted a readonly
copy, we could use toCvShare to avoid extra memcpy ops
cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
}
catch (cv_bridge::Exception& e)
{
ROS_ERROR("cv_bridge exception: %s", e.what());
return;
}
//do interesting work here...
// image dilation example:
// cv::dilate(cv_ptr->image, cv_ptr->image, cv::Mat(), cv::Point(-1,-1), 10);
// for each pixel
int nl= cv_ptr->image.rows; // number of lines
int nc= cv_ptr->image.cols;
```

```
ROS_INFO("number of rows: %d; number of columns: %d",nl, nc);
//NOTE: could/should wrap this up in a class and run faster using iterators
// also, since this example only outputs black or white, could have used a gray-scale or binary image
output
// process each pixel ---------------------
for (int i=0;i<nc;i++)
for (int j=0;j<nl;j++) {
// compute distance from target color
//if (getDistance(*it)<minDist) {
pixel_color[0]=cv_ptr->image.at<cv::Vec3b>(j,i)[0];
pixel_color[1]=cv_ptr->image.at<cv::Vec3b>(j,i)[1];
pixel_color[2]=cv_ptr->image.at<cv::Vec3b>(j,i)[2];
dist = abs(pixel_color[0]-target_color[0])+ abs(pixel_color[1]-target_color[1])+ abs(pixel_color[2]-
target_color
[2]);
cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 255;
cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 0;
cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 0;
if (dist>thresh) {
cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 0;
cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 0;
cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 0;
}
else {
cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 255;
cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 255;
cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 255;
}
} //done with all pixels
// Publish the modified image
image_pub_.publish(cv_ptr->toImageMsg());
}
};


int main(int argc, char** argv)
{
// SET THE TARGET COLOR HERE--keep pixels that are close to this color
target_color[0]=30;
target_color[1]=30;
target_color[2]=30;
default_input=true; //use default input, unless command-line arg is provided
if (argc==2)
default_input=false;
ros::init(argc, argv, "opencv_sample");
ROS_INFO("wsn opencv_sample pgm: ");
ROS_INFO
("do: rosrun image_view image_view image:=/opencv_sample_out to see the output of this
program");
ImageConverter ic;
while (true)
{
ROS_INFO("enter threshold distance: 0 to quit:");
std::cin>>thresh;
if (thresh==0)
return 0;
ros::spinOnce();
}
return 0;
}
```

Vision-Based Vehicle Control: An example of using real-time image processing for control is provided in the package “vehicle_controller.” This is a starting point for vision-based driving. The code is illustrative, NOT truly functional nor efficient. It uses elements of the example "opencv_sample" package, and includes use of the "cheat" topics to control the DRC vehicle brake, accelerator and steering.

The theory of operation is as follows. The Atlas robot is pre-positioned in the driver’s seat of the DRC vehicle. Looking forward, Atlas has a view of the ground. The “road” is recognized based on color thresholding. (The reference color may be found as described above, using Gimp to examine colors of regions of interest, and what makes road colors distinct from other regions in the environment). Using the previous openCV example code, input images from Atlas are analyzed for color matching, and the resulting images are coded as white (color match) or black (no match) for each pixel.

Simplistically, a single horizontal line of the view is analyzed to find the centroid in the “i” direction (left/right). This value is compared to the center column to evaluate if the road appears to be shifted left or right of the robot’s center of visual field. If the estimated center of the road is to the left, then the control program commands the vehicle to steer right (CW), and vice-versa.

This package should be compiled, as usual, with: `rosmake vehicle_control`

However, before the program can be of use, the simulator has to be prepared with the proper initial conditions and environment.

The code assumes that a suitable "track" has been loaded into the simulator. To do so, get "besttrackever" (a copy of which is in this package in the team repository). Copy the necessary files (besttrackever.sdf and model.config) into your directory:

```
  /home/user/.gazebo/models
```

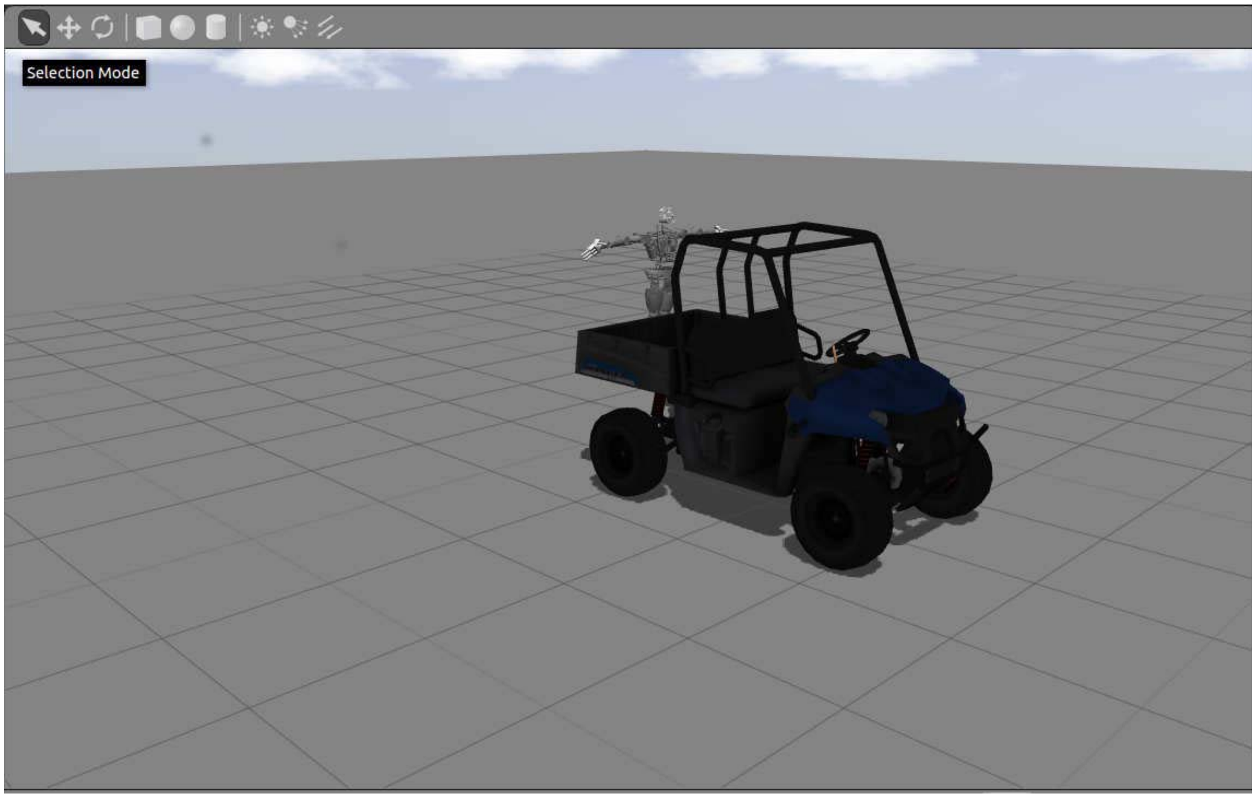
(on the team machines, put it in `/home/public/.gazebo/models`)

If you are browsing your directory structure using the "files" application, you will need to hit control-h to see the hidden files (files starting with ".").

With this model installed, launch the simulator with the DRC vehicle using:

```
roslaunch atlas_utils atlas_drc_vehicle_fire_hose.launch
```

The Gazebo simulator window should look like the following:



Next, get Atlas in the driver's seat. This will be a significant challenge to do with sensor-based control of Atlas' joints, but for development purposes, there is a short-cut. In a terminal window, run the following:

```
  rostopic pub --once /drc_world/robot_enter_car geometry_msgs/Pose '{}'
```

You should see Atlas seated in the vehicle, as shown below.

Now, insert the track. To do so, in the Gazebo window, choose the "insert" tab on the left. Youd should see your newly installed "Best Track Ever" option on the list if items. Click on it to insert the scene.

In the simulation-display window, you can move the ground view around with your mouse. Move it until the vehicle is on the track, then click to place the ground there. The result should look like the figure below:

Optionally, you can visualize Atlas's view with rviz. In a terminal, enter:

```
  rosrun rviz rviz
```

In the left panel, .Global Options, under "fixed frame", choose "pelvis."

In the same panel, at the bottom, click "Add". From the pop-up window, choose "camera."

A "camera" item will come up in the "displays" panel on the left side of rviz. Click to the right of "Image Topic" and choose:

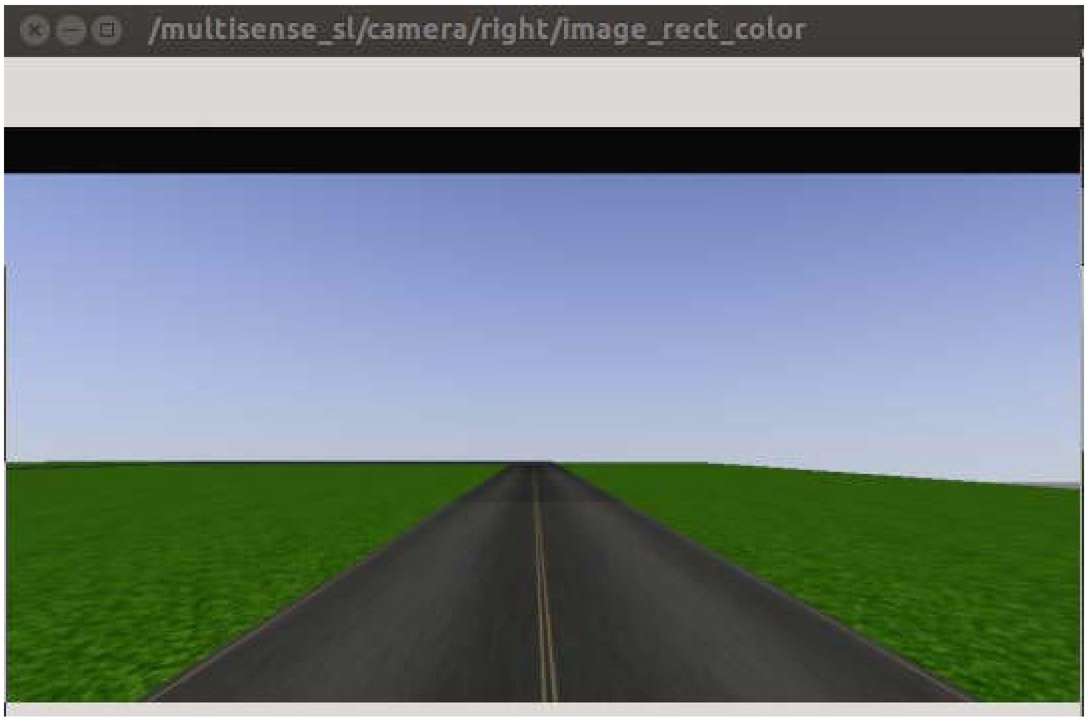
```
    multisense_sl->camera->right->image_rect_color
```

Rviz should now display a panel that shows the road in front of the vehicle with grass on either side.

Alternatively, use “image_view” to visualize Atlas’s view. Do this (from a separate terminal) with:

```
rosrun image_view image_view image:=
//multisense_sl/camera/left/image_rect_color
```

Atlas’s view should look something like the following:



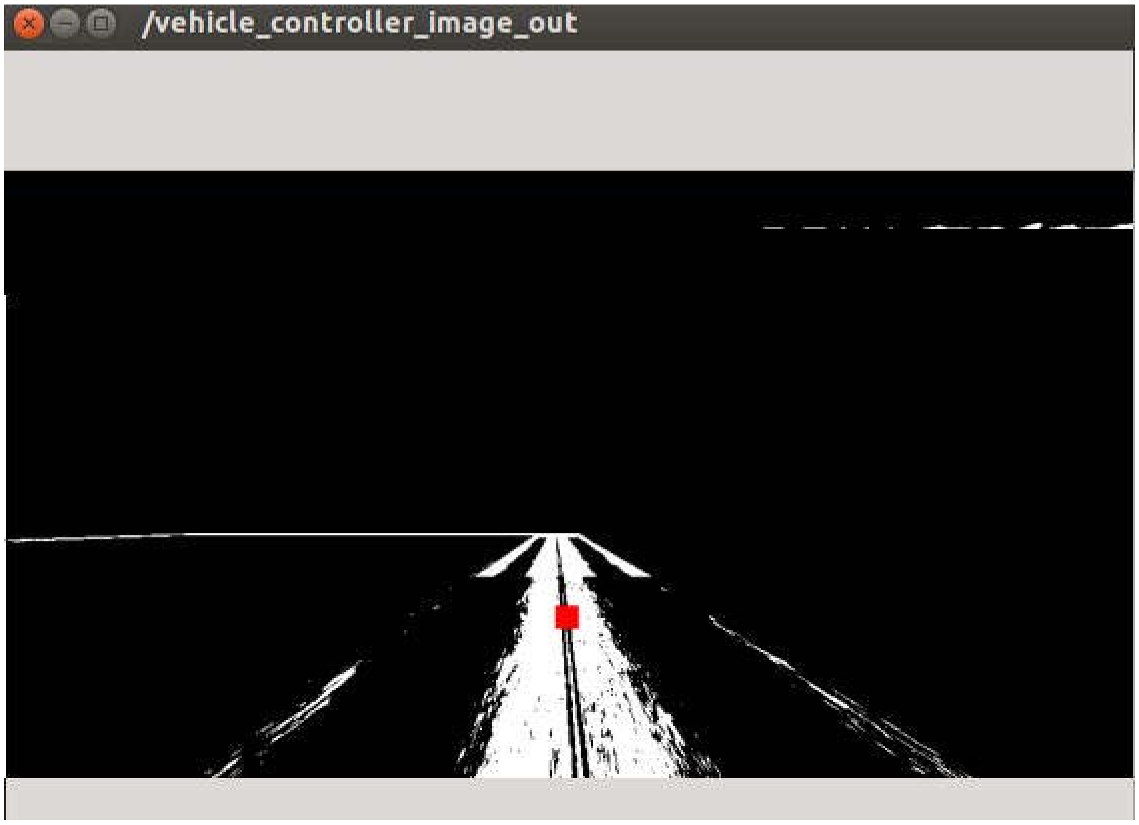
Finally, run the vehicle controller demo. In a separate terminal, run:

```
  rosrun vehicle_control vehicle_control
```

To view the image-processing results, in a separate terminal, run:

```
  rosrun image_view image_view image:=/vehicle_controller_image_out
```

The image-view window should display something like this:



The white pixels have been judged to be sufficiently “gray” that they are believed to be part of the pavement. Obviously, some of the true pavement has been neglected, and some pixels near the horizon fit the color description.

For the example controller, only a single, horizontal line of this image is considered. The average column-index value along this row of pixels is computed. The result is displayed, superimposed on the processed image, as a red square. From this display, you can see the horizontal line that was chosen to be analyzed (about 1/3 of the way up from the bottom) and the interpretation of the center of the road (which looks quite good in this example).

Controlling the vehicle requires publishing to three topics: the hand brake (set to 0.0 to release the brake), the gas pedal (which seems to respond on/off, rather than expected range of speed control), and the wheel angle (which can range from -7rad to +7rad). A positive wheel-angle command results in the vehicle driving in a curve to the left (CCW).

The example control algorithm (very crude!) simple commands a left turn if the perceived center of the road is to the left, and a right turn if the perceived center of the road is to the right. No attempt has been made to optimize this control. Nonetheless, it almost works—on straightaways. (The vehicle controller is confused by the expanse of asphalt at the ends of the tracks).

The vehicle_controller code follows (with some typesetting difficulties).

```
// example program  using opencv to drive the DRC vehicle
// based on opencv_sample

// by default, grabs image data from ROS topic //multisense_sl/camera/right/image_rect_color, though
//  the source is a command-line argument option
//  A hard-coded target color is defined in this code

#include<std_msgs/Float64.h>
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>

using namespace cv;

namespace enc = sensor_msgs::image_encodings;
int thresh=30;
bool default_input;
int g_nc; //number of columns in image
int col_centroid; // centroid of columns along chosen horizontal line; pass this to "main" for steering
int num_pix_on;  // similarly,  make number of "on" pixels available to "Main"

cv::Vec3d target_color; // FILL IN THE VALUES OF THE TARGET COLOR IN main
();  HARD CODED IN THIS EXAMPLE

class ImageConverter
{

  // member variables
  ros::NodeHandle nh_, priv_nh_;
 // pub_joint_commands_ = nh_jntPub_.advertise<osrf_msgs::JointCommands>(
  //                      "/atlas/joint_commands",  1,
  true); image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
  int image_num;

  public:
  ImageConver
  ter()
    : nh_(),
    priv_nh_("
    ~"),
    it_(nh_)

  {
    // We'll publish the resulting image on "out" and subscribe to "in"
    image_pub_ = it_.advertise("vehicle_controller_image_out", 1);
    //default to topic: //multisense_sl/camera/right/image_rect_color
    if (default_input) {
       image_sub_ = it_.subscribe
```

```
      ("//multisense_sl/camera/right/image_rect_color", 1, &ImageConverter::imageCb,
      this); ROS_INFO("using default input of
      //multisense_sl/camera/right/image_rect_color"); ROS_INFO
 ("to use an altenative input topic, run as: rosrun opencv_sample opencv_sample in:=//topic_name");
    }
    else
      image_sub_ = it_.subscribe("in", 1, &ImageConverter::imageCb, this);
   }

   ~ImageConverter()
   {
   }

void imageCb(const sensor_msgs::ImageConstPtr& msg)

   {

     cv_bridge::CvImagePtr cv_ptr; //use cv_ptr->image in place of "image" in OpenCV manual
    int dist; //color distance
    cv::Vec3d pixel_color; //variable to hold a pixel color for comparison

     try
     {

      // Create a copy of the image in msg, requesting that the image be converted to BGR8 if it isn't already
      // We make a copy so that we can operate in place on it... if we wanted a read-
  only copy, we could use toCvShare to avoid extra memcpy ops
      cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);

     }
     catch (cv_bridge::Exception& e)
     {

      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
     }

    //do image processing work here...

    // for each pixel
    int nl= cv_ptr->image.rows; // number of lines
    g_nc= cv_ptr->image.cols;  //number of columnes--a global variable passed to "main"
    //ROS_INFO("number of rows: %d; number of columns: %d",nl, g_nc);

   //NOTE: could/should wrap this up in a class and run faster using iterators
     // process each pixel ---------------------
   for (int i=0;i<g_nc;i++) //column index
    for (int j=0;j<nl;j++) //row index
     {
     // compute distance from target color
     // unnecessary step, but for clarity, copy the current pixel's color values into a local vector, pixel_color
        pixel_color[0]=cv_ptr->image.at<cv::Vec3b>(j,i)[0];
        pixel_color[1]=cv_ptr->image.at<cv::Vec3b>(j,i)[1];
        pixel_color[2]=cv_ptr->image.at<cv::Vec3b>(j,i)[2];

     // compute the manhattan distance from target_color to pixel_color in color space:
        dist = abs(pixel_color[0]-target_color[0])+ abs(pixel_color[1]-target_color[1])+ abs(pixel_color[2]-
        target_color[2]);

      if (dist>thresh) {  //here if not a good enough match;  may need to adjust target pix color and thresh tolerance
     cv_ptr->image.at<cv::Vec3b>(j,i)[0]= 0; //not a match, so turn this pixel black
     cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 0;
     cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 0;
      }
      else {
     cv_ptr->image.at<cv::Vec3b>(j,i)[0]=  255; //got a match, so turn this pixel white
     cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 255;
     cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 255;
      }
    }  //done with all pixels

// now, compute the centroid  of the surviving  pixels for one row:

    // note: this could  have been done in above loop--and  would  not be necessary to do the entire scene!
    int mid_line = 400; //nl/2; //pick a horizontal line somewhere up ahead of the robot..magic number to be tuned
    num_pix_on = 1; //initialize with one pixel in the middle column--avoid  divide by zero
    int sum_col_indices=g_nc/2; // default centroid in center

    //ROS_INFO("mid_line = %d",mid_line);

    for (int i=0;i<g_nc;i++) { // for all values of column index, along  chosen row:
      if (cv_ptr->image.at<cv::Vec3b>(mid_line,i)[0]==255) {  // if pixel passed color test, count it
        //ROS_INFO("i=%d pix is on",i);
        num_pix_on++;  // another pixel passed the test
        sum_col_indices+=i;  // add in its column index

      }
    }

    col_centroid = sum_col_indices/num_pix_on;  //compute centroid, in column coordinates, along chosen row of
    image

    //ROS_INFO("num_pix_on = %d; col_centroid = %d",num_pix_on,col_centroid);

    //color  centroid  pixel  red...and  some of its neighbors--just for display/debug purposes
    if ((col_centroid>10) &&  (col_centroid< g_nc-10))
      for (int i=col_centroid-
      10;i<col_centroid+10;i++) for (int
      j=mid_line-10;j<mid_line+10;j++) { cv_ptr-
      >image.at<cv::Vec3b>(j,i)[0]= 0;
      cv_ptr->image.at<cv::Vec3b>(j,i)[1]= 0;
      cv_ptr->image.at<cv::Vec3b>(j,i)[2]= 255;
      }

      // Publish  the modified  image; can view this, e.g., in image_view
      image_pub_.publish(cv_ptr->toImageMsg());
   }
 }
 ;
```

```
int main(int argc, char** argv)
{

  ros::init(argc, argv, "vehicle_controller");
  ros::NodeHandle nhp_gas; //nodehandle for publisher of gas
  pedal ros::NodeHandle nhp_steer; //nodehandle for publisher of
  steering ros::NodeHandle nhp_brake; //nodehandle for publisher
  of hand brake

  ros::Publisher pub_gas_pedal =
  nhp_gas.advertise<std_msgs::Float64>("/drc_vehicle/gas_pedal/cmd",1); ros::Publisher pub_steering
  = nhp_steer.advertise<std_msgs::Float64>("/drc_vehicle/hand_wheel/cmd",1); ros::Publisher
  pub_brake = nhp_brake.advertise<std_msgs::Float64>("/drc_vehicle/hand_brake/cmd",1);

  std_msgs::Float64
  wheel_ang;
  std_msgs::Float64
  gas_pedal;
  std_msgs::Float64
  hand_brake;

  ros::Rate looprate(10.0);  //loop at
  10Hz wheel_ang.data=3.0;
  gas_pedal.data=0.1;
  hand_brake.data=0.0; //1.0; //1.0 for brake on, 0.0 for brake off

  pub_steering.publish(wheel_ang);


pub_gas_pedal.publish(gas_pedal);
// SET THE TARGET COLOR HERE--keep pixels that are close to this color
target_color[0]=30;
target_color[1]=30;
target_color[2]=30;

  thresh=60; // manually set color dist
  threshold

 default_input=true; //use default input, unless command-line arg is
 provided
```

```
  if (argc==2) {
  default_input=false;
  ROS_INFO("wsn vehicle_controller
  pgm: ");
  ROS_INFO("do:  rosrun
  image_view image_view
  image:=/vehicle_controller_image_
  out to see the output of this
  program");
ImageConverter ic;
int col_err; // compute the error of column centroid relative to center
while (ros::ok()) {  // main loop for steering
    col_err= col_centroid-g_nc/2;
// positive column error-> road is to the right, -> need to turn right -> exert negative steering
//harsh, discretized (jerky) controller
if (col_err>100)
   wheel_ang.data= -7.0;
else if (col_err>50)
   wheel_ang.data= -3.0;
else if (col_err> -50)
   wheel_ang.data= 0.0;
else if (col_err> -100)
   wheel_ang.data= 3.0;
else
   wheel_ang.data= 7.0;
if (num_pix_on < 10) {
   wheel_ang.data=3.0; // turn left if lost road
   ROS_INFO("lost the road; turning  left");

}
      if (num_pix_on > 300) {
         wheel_ang.data=3.0; // turn left if in "end-zone"
         ROS_INFO("too much asphalt!  turrning  hard left");

       }
       ROS_INFO("col_centroid = %d;   col_err= %d;  num_pix= %d;  steer ang= %
f",col_centroid, col_err,num_pix_on,wheel_ang.data);


      pub_steering.publish(wh
      eel_ang);
      pub_gas_pedal.publish(g
      as_pedal);
      pub_brake.publish(hand_
      brake);
      looprate.sleep(); // this will cause the loop to sleep for balance of time of desired (100ms) period
      ros::spinOnce(); // need this to receive callbacks when interleaving subscribe and publish actions
  }
return 0;
}
```