Main Functions:

- **void loadImage(cv::Mat& image_out, const std::string& config_folder);**
  This function can be used to replace the simulator camera and test the developed pipeline on a set of custom image
  - param[out]: image_out: The loaded raw image
  - param[in]: config_folder: A custom string from config file.
- **void genericImageListener(const cv::Mat& img_in, std::string topic, const std::string& config_folder);**
  Generic listener used from the image listener node to save an image.
  - param[in]: image_in: Input image to store
  - param[in]: topic: Topic from where the image is taken
  - param[in]: config_folder: A custom string from config file.
- **bool extrinsicCalib(const cv::Mat& img_in, std::vector<cv::Point3f> object_points, const cv::Mat& camera_matrix, cv::Mat& rvec, cv::Mat& tvec, const std::string& config_folder);**
  Finds arena pose from 3D(object_points)-2D(image_in) point correspondences.
  - param[in]: image_in: Input image to store
  - param[in]: object_points: 3D position of the 4 corners of the arena, following a counterclockwise order starting from the one near the red line.
  - param[in]: camera_matrix: 3x3 floating-point camera matrix
  - param[out]: rvec: Rotation vectors estimated linking the camera and the arena
  - param[out]: tvec: Translation vectors estimated for the arena
  - param[in]: config_folder: A custom string from config file.
- **void imageUndistort(const cv::Mat& img_in, cv::Mat& img_out, const cv::Mat& cam_matrix, const cv::Mat& dist_coeffs, const std::string& config_folder);**
  Transform an image to compensate for the lens distortion.
  - param[in]: image_in: distorted image
  - param[out]: image_out : undistorted image
  - param[in]: camera_matrix: 3x3 floating-point camera matrix
  - param[out]: dist_coeffs: distortion coefficients [k1,k2,p1,p2,k3]
  - param[in]: config_folder: A custom string from config file.
- **void findPlaneTransform(const cv::Mat& cam_matrix, const cv::Mat& rvec, const cv::Mat& tvec, const std::vector<cv::Point3f>& object_points_plane, const std::vector<cv::Point2f>& dest_image_points_plane, cv::Mat& plane_transf, const std::string& config_folder);**
  Calculates a perspective transform from four pairs of the corresponding points.
  - param[in]: camera_matrix: 3x3 floating-point camera matrix
  - param[in]: rvec: Rotation vectors estimated linking the camera and the arena
  - param[in]: tvec: Translation vectors estimated for the arena
  - param[in]: object_points_plane: 3D position of the 4 corners of the arena, following a counterclockwise order starting from the one near the red line.
  - param[in]: dest_image_points:_plane destinatino point in px of the object_points_plane
  - param[out] plane_transf: plane perspective trasform (3x3 matrix)
  - param[in] config_folder: A custom string from config file.
- **void unwarp(const cv::Mat& img_in, cv::Mat& img_out, const cv::Mat& transf, const std::string& config_folder);**

Applies a perspective transformation to an image.

- param[in]: image_in: input image
- param[out]: image_out: unwarped image
- param[in]: transf: plane perspective trasform (3x3 matrix)
- param[in]: config_folder: A custom string from config file.

The output of these functions gives the result of an undistorted and unwarped in the respective plane as per figure below:

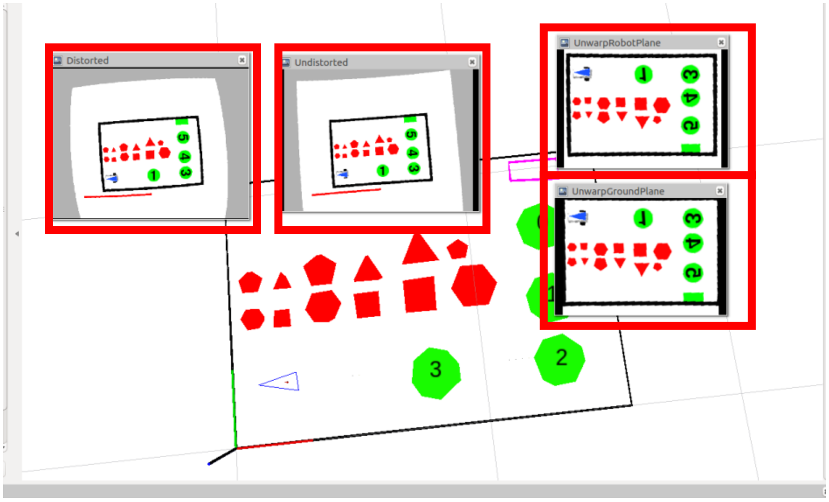

Figure 1: Undistorted and unwarped result in the respective plane

- **bool processMap(const cv::Mat& img_in, const double scale, std::vector<Polygon>& obstacle_list, std::vector<std::pair<int,Polygon>>& victim_list, Polygon& gate, const std::string& config_folder);**

  Process the image to detect victims, obtacles and the gate

  - param[in]: image_in: input image
  - param[in]: scale: 1px/scale = X meters
  - param[out]: obstacle_list: list of obstacle polygon (vertex in meters)
  - param[out]: victim_list: list of pair victim_id and polygon (vertex in meters)
  - param[out]: gate: polygon representing the gate (vertex in meters)
  - param[in]: config_folder: A custom string from config file.

  Obstacles

  First the image is converted to HSV format. Then a filter (erosion and dilation) and mask is applied to detect the obstacles in red colour. Finally the resulting shapes are detected and saved as obstacle. The test of all the steps on real camera image is given below:

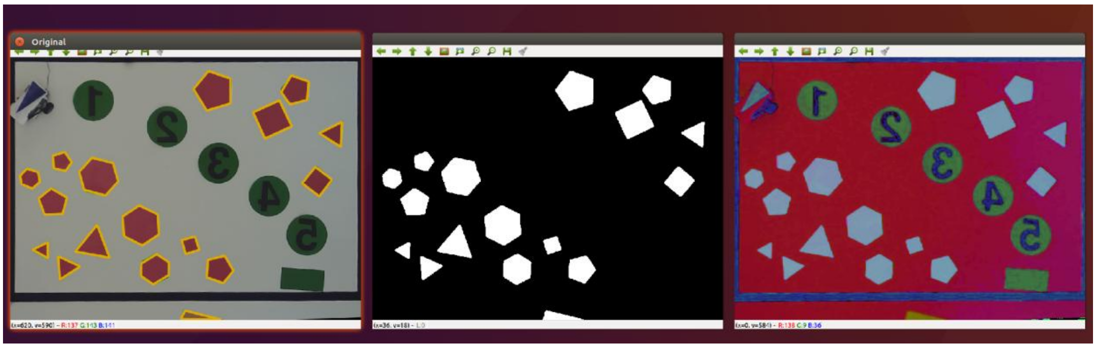

Figure 3: HSV conversion, colour filtering/masking and obstacle detection in arena

Gate/Victims

First the image is converted to HSV format. Then a filter (erosion and dilation) and mask is applied to detect the victims in green colour. Finally the resulting shapes are detected and saved as obstacle. The test of all the steps on real camera image is given below:

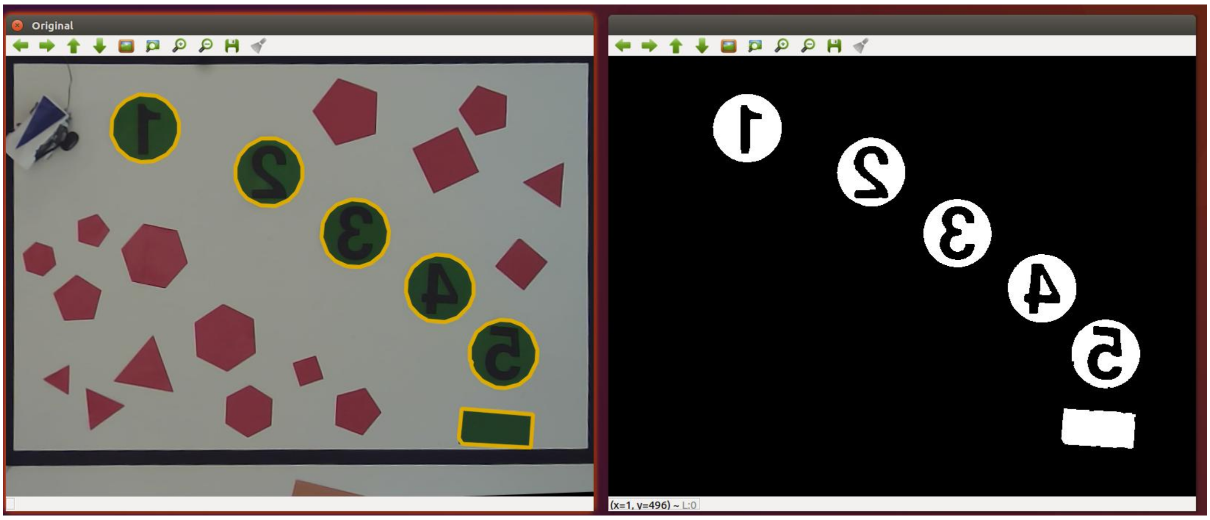

Figure 3: HSV conversion, colour filtering/masking and victim/gate detection in arena

- **bool findRobot(const cv::Mat& img_in, const double scale, Polygon& triangle, double& x, double& y, double& theta, const std::string& config_folder);**
  Process the image to detect the robot pose
- param[in]: image_in: input image
- param[in]: scale: 1px/scale = X meters
- param[out]: x: x position of the robot in the arena reference system
- param[out]: y: y position of the robot in the arena reference system
- param[out]: theta: yaw of the robot in the arena reference system
- param[in]: config_folder: A custom string from config file.

First the image is converted to HSV format. Then a blue filter and mask is applied to detect the triangle of robot in blue colour. Finally the resulting triangle's center and vertex are joined to find the robot's position. The image of all the steps is given below:

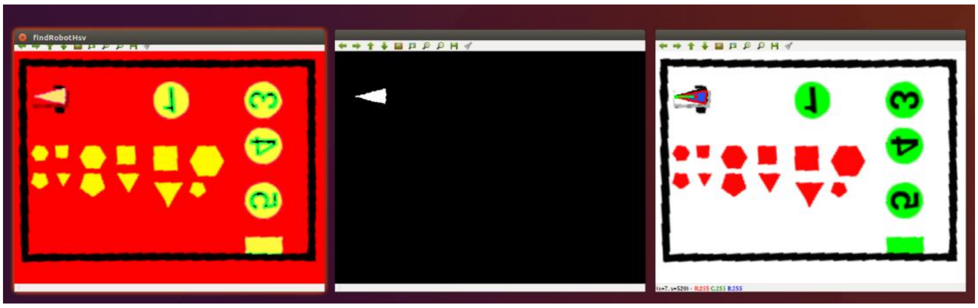

Figure 3: HSV conversion, colour filtering/masking and robot localization in arena

Project Backlog

Need to rotate the template numbers for matching so that if there is some tilt in actual arena, the algorithm can still recognize the numbers.