Zebravision 4.0 Tracking

Alon Greyber

A Zebravision Labs White Paper:

Brought to you by Team 900:

http://team900.org

Introduction

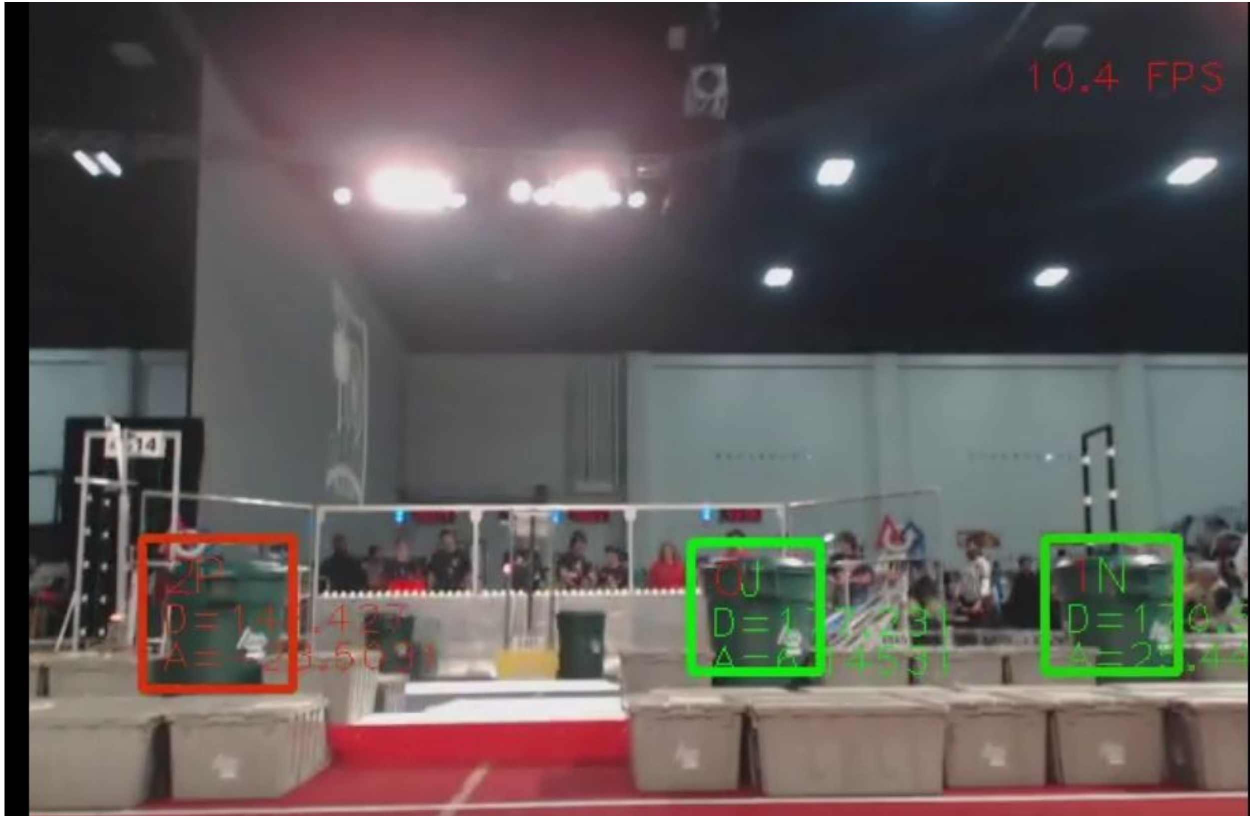
Zebravision 4.0 is Team 900s vision system for the 2016 season; FIRST Stronghold. Our work was focused around recognizing the vision goals using shape and color based matching, recognizing the boulders using a neural network, and integrating the detection systems into a tracking system using the StereoLabs ZED stereo camera. This paper describes our tracking system, or how we get useful information that is persistent across frames from our detections.

Previous Tracking System

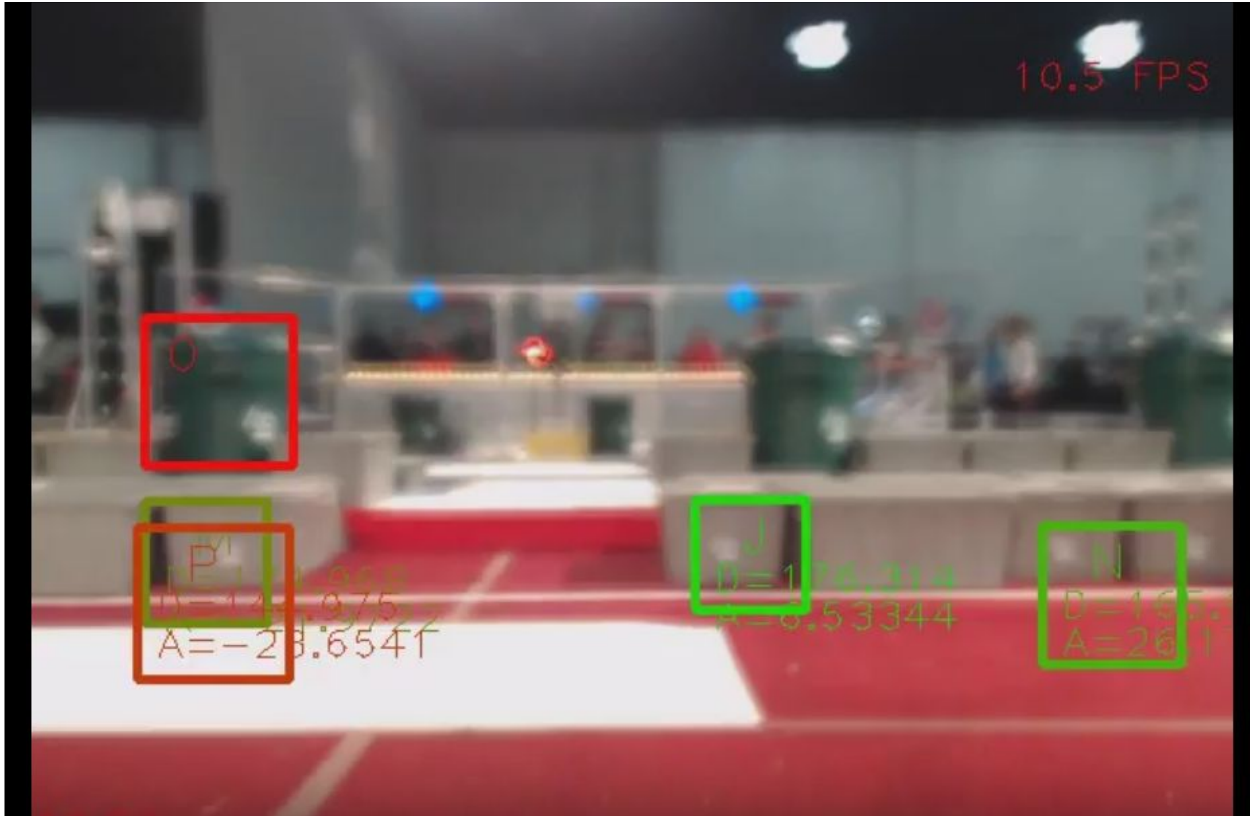
Zebravision 3.0 was used during the Recycle Rush game for detection and tracking of bins on the field. The tracking system was relatively simple, it used the bounding rectangles of the detected objects on the screen and kept track of them across multiple frames using a list. Each detection that was seen over multiple frames would gain in confidence and would subsequently be harder to remove.

This system had many drawbacks:

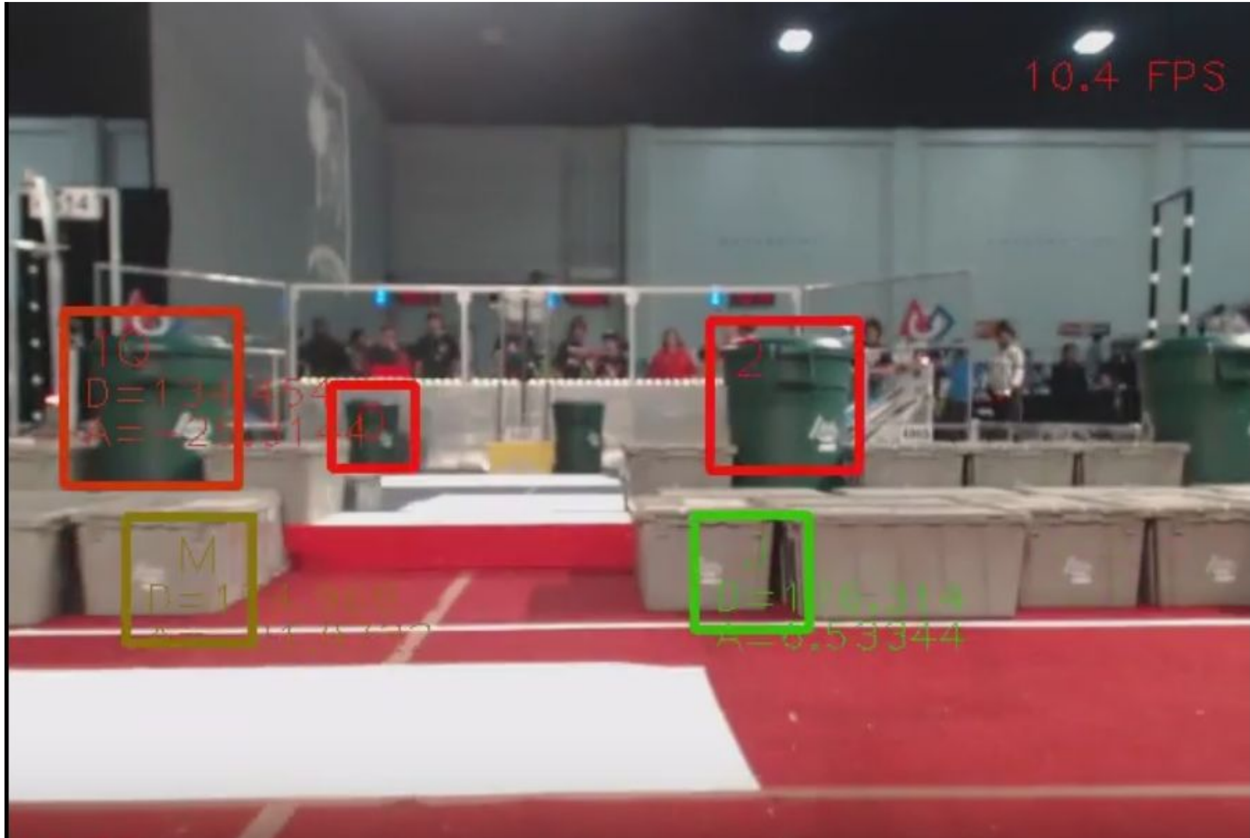
- To assign new detections to old objects we searched in a radius around the new detection for a tracked object to assign it to. This failed if there were two detections for a single object.
- If the robot moved too quickly the system would consider the new detections as new objects because everything shifted. See the picture below.
- Each detection was searched individually for an object to be assigned to. This caused problems when multiple detections were close together, as one would be assigned to the object and one to be assigned to the next closest which was often a different detection.



This sequences of images show the problem our previous code had with abrupt camera motion. The first frame shows that objects are being tracked correctly. The rectangles are centred over the targets correctly indicating that the code can accurate match up detections in this frame with objects tracked from previous frames.



The second frame shows the robot just after it came off the step which jerked the camera down a large amount. The tracks remain in the same place on the screen, but since the camera moved they're no longer correctly centered on the tracked objects. It has also failed to assign either of them to the detected target on the left (the box with 0 in it). The overlapping boxes in the lower left corner also indicate the code has mistakenly created a second tracked object there, probably due to an intermediate detection which happened as the camera was pitching down.



In the third frame, the camera has stabilized and more targets are detected. But instead of correctly assigning those new detections to the old tracks, the code can't match them up and creates new tracks instead.

Some of these issues could be solved by increasing the detection radius - that is, allow a wider area to be searched to find a matching old track. Given how quickly robots can move between frames of video this would require enlarging the search radius so much that it would match a large percentage of the frame.

How it works

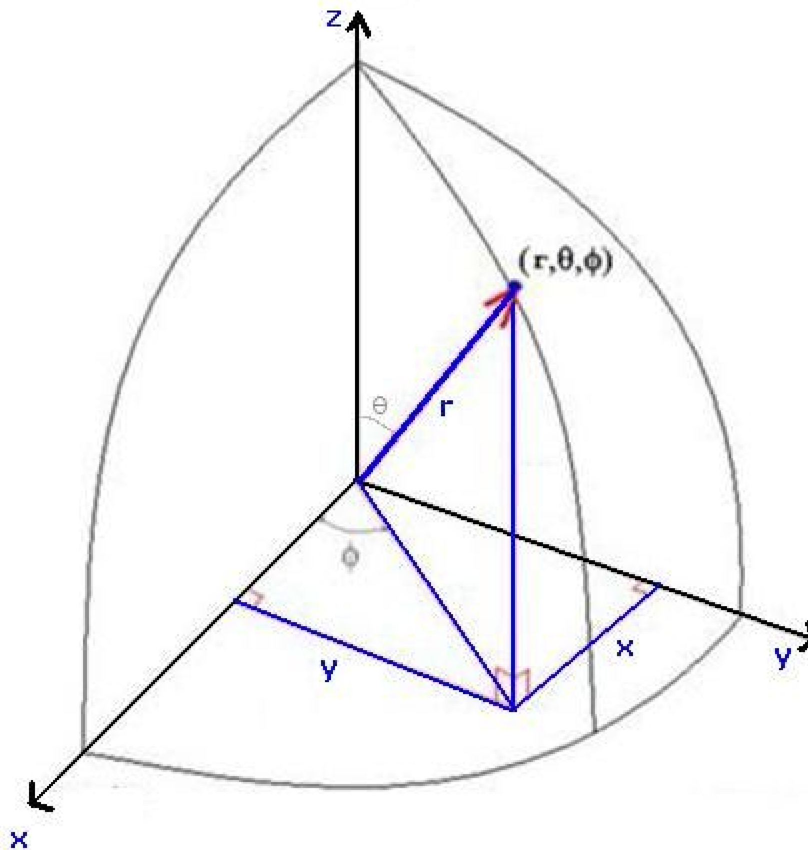
Conversion to 3D Points

Zebravision 4.0 represents detected objects in 3D space rather than as bounding rectangles on the screen. This is made possible from the depth data that we get by using the StereoLabs ZED camera. To determine the x, y, and z coordinates of an object detected we use the position of the object on the screen in addition to the distance that the object is away from the camera.

To find the position of the object on the screen we use the center of the bounding rectangle of the object. The ZED camera gives a depth in real world units for each pixel on the screen. We find the minimum depth on the object we are looking at. This finds the point on the object closest to the camera which works well in most cases. This also helps eliminate a common

problem which is when detection finds part of the background, as the background has a very high depth value therefore it won't be included in the calculation of the minimum.

The math for computation of x,y, and z coordinates of objects given the position on the screen and a distance precisely mirrors the conversion from a spherical coordinate system to a cartesian one. This is best illustrated with a diagram:



Spherical coordinates are given by two angles, named azimuth (φ) and inclination (θ)[1] and a radius (r). These are very similar to polar coordinates but in 3 dimensions rather than 2. The formulas for conversion to cartesian are relatively simple and look quite similar to polar coordinates:

$$x = r \sin\theta \cos\varphi$$

[1] Physicists decided they wanted to be different and switched azimuth to be the vertical angle and inclination to be the horizontal so you may see it written switched in some contexts.

$$x = r \sin\theta \sin\varphi$$

$$x = r \cos\theta$$

We already have r, it's just the distance that we get from the depth information. To find the angles we need to use the field of view of the camera. The field of view is the number of degrees taken up by the entire frame. The concept for computing the angles is this:

distance_to_center_x = detection_bounding_rect_center.x - (frame_width / 2)

distance_to_center_y = detection_bounding_rect_center.y - (frame_height / 2)

percent_frame_x = distance_to_center_x / frame_width

percent_frame_y = distance_to_center_y / frame_height

*azimuth = percent_frame_x * fov_x*

*inclination = percent_frame_y * fov_y*

This conversion is based on a simple ratio. The idea is that $\frac{size\ of\ object\ (px)}{image\ size\ (px)} = \frac{size\ of\ object\ (rad)}{fov\ size\ (rad)}$. This lets us convert the position of the object easily to two angles. Once we have the two angles we convert to cartesian coordinates using the formulas given above

In our code we also implement a function that does the reverse of this; determining the bounding box of an object based on its 3d position. This reversal operation requires properties of the object such as width and height as we do not have depth. The formulas use the same ratio technique but in reverse. In this case we know the size of the object in radians and we are trying to determine the size of the object in pixels.

Assignment

In any camera sensor there is noise inherent in measurements. Sometimes a single object is detected multiple times. Sometimes we lose track of objects for a frame or multiple frames and then pick it back up later. This creates the need for a system that keeps track of objects over multiple frames and uses all the information it has to produce useful data.

We use a simple array to keep track of objects we have detected. The first challenge of this is how to assign new detections into the array. For this we use a method called hungarian assignment. Hungarian assignment works to minimize the total distance between each detection and its

assigned track. This produces better solutions than going through each detection because it works with all detections rather than one at a time. Since we are working with 3d points the distance is given by $\sqrt{(x_{new}-x_{old})^2 + (y_{new}-y_{old})^2 + (z_{new}-z_{old})^2}$.

Without the hungarian method the way to find the optimal assignment would be to compare every possible arrangement of new detections to tracked objects, a method that quickly gets out of hand with anything over a few detections. The hungarian method completes this task with a very computationally efficient strategy. For an overview of how the hungarian assignment works view this presentation. We used an implementation provided by Smorodov on GitHub. Our results with this have been that it is incredibly accurate and resistant to large movements in the camera as well as multiple detections very close together. If there are extra detections that are unassigned after the process, i.e. there is no corresponding object for a detection we create a new object.

Tracking

Once we have a list of assignments we need a way to update the position of the object in the array based on the position of the new detection. The easiest way to do this would be to replace the position in the array with the new object. However detections always have some inaccuracies so it makes sense to do averaging in this step.

The compromise of averaging and direct assignment is a Kalman filter. Kalman filters track a history of measurements of an object. Since they keep track of both past measurements and the error in them it can make accurate predictions of the future state of the object.

For each new frame the Kalman filter generates the predicted state of each tracked object. Internally the Kf tracks not only position but velocity, acceleration, etc. so that using a known time step it can estimate the future positions of objects.

Next, each detection is matched to a predicted object position. The prediction is then corrected with the new measurement. For objects which don't match up to a any prediction, a new track with its own Kalman filter is created. For the opposite case - predicted objects which aren't detected this frame - the code uses the the predicted location for this frame and tries to match it in subsequent frames. Tracked objects which aren't seen for a number of frames are deleted.

For an in depth look at how Kalman filters work and the theory behind them you can visit this paper here.

Position Adjustment

A problem with our tracking system arises when the camera moves, moving the objects in our image. Although the Hungarian Algorithm does a good job assigning objects even if they have shifted positions it's much easier when they don't move or if we have an estimate of how far they have moved and their new positions. For this we use optical flow. Optical flow is a method of estimating the average movement of an object in an image based on two successive frames.

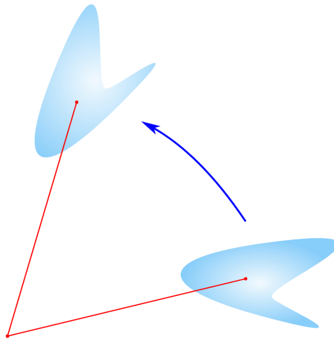
[*Image from imaco.pl*](https://imaco.pl)

In this example optical flow is used to determine which objects in the frame are moving and in which direction. Each dot represents a spot in the image that has been detected to have movement by the optical flow.

In our code we use optical flow to determine how much the entire frame has moved. As mentioned earlier, the main source of this movement is movement of the camera. OpenCV provides a function to calculate optical flow. This provides us with a 2d transformation matrix that represents the average movement of a pixel in the image.

To apply this to our objects we convert each object into a pixel on the screen using the reverse process of the conversion from screen to world coordinates. Then we apply the inverse of the transformation matrix to the object and finally convert back to world coordinates.

It's important to note that during the step of converting back to world coordinates we don't have the distance to the object because the object's new location is just a prediction. We assume that the object's r value is the same as during the conversion to screen coordinates. This works well for rotation because rotation does not change the distance the object is from the camera. Also this doesn't have to be very accurate as the error does not accumulate because the position is corrected every camera frame.



R does not change for rotation

Looking Forward

This system worked extremely well for tracking balls in the 2016 game using our neural network detection system. However it does have a few issues:

- As the framerate of the detection decreases down below 10 frames per second the performance of the Kalman filters deteriorates rapidly. This is because Kalman filters work best when they can make good predictions of velocity and if the object is only seen twice as it crosses through the screen the Kalman filter can't make a good estimate.
- Optical flow adjustment only works in two dimensions and this is very obviously a problem with the necessity to assume that r remains constant.
- Finding the minimum depth on an object for tracking tracks the very front point of the object. For an object like a ball this remains consistent but for something with a forward facing flat side this may vary.
- Finding and tracking the minimum depth point on the object tracks the front face. If a robot has a mechanism it would be better to add half the thickness of the object to that depth so we are tracking the center.
- Our conversion to 3d coordinates using the ratio is an approximation. The actual formula is nonlinear and slightly different from our linear approximation. More about that here.

One solution we have explored to the optical flow correction problem is switching to a system that can estimate the change in both rotation and position of the camera in 3d. This process is called visual odometry. The library we explored for visual odometry is called fovis. This library estimates the change in position and rotation of the camera and keeps track of the change to form a pose. We eventually abandon this library early in the season because we could not get frame rates high enough to have good accuracy. However now that we have multithreaded code and faster ZED code in SDK version 0.9.3 we may in the future be returning to fovis as a possible solution.

Another possible solution to the correction problem is an IMU (Inertial Measurement Unit) on the robot that can estimate the movement of the camera. It also might be possible to combine the strengths of both fovis and an IMU and come up with a more accurate estimate.

Kalman filters estimate the future state of the system based on a prediction model and a time difference between the current and previous frame. Our time differences are estimated to be constant and this may be inciting inaccuracies as our frame rate varies. In the future we will try to use timestamps on each frame provided to us by the ZED camera to establish an actual time difference between frames.

Thanks

Special thanks to Stereolabs for providing support and assistance as we tested and integrated the ZED into our own work. We could not do this without their support!

Thanks to our vision team mentors Eric Blau and Kevin Jaget! They were instrumental to bringing the ideas we came up with into a functioning vision system for competition.

Also a huge thank you to Nvidia whose sponsorship and continued support for our team's work in vision has been incredibly helpful. They've been great at giving us support for their mobile computing solutions, the Jetson TK1 and TX1, which we've been using on our robot for the past two years.