Guide to Stacking Blocks
Aaron Zhao
Dead Robot Society

Guide to Stacking Blocks

Overall Strategy

As part of our overall strategy, our team decided that one of our robots should stack the blocks, because the blocks are worth a lot of points. When we brainstormed ideas for how to stack the blocks in the correct color order, we came across a problem: there are three blocks in three locations creating six different possibilities for the arrangement of the three blocks. This is really difficult because the colors have to be recognized, the blocks have to be picked up and then stacked in the Marine Protected Area (MPA). We came up with a creative solution of using areas near the blocks as checkpoints. We enjoy finding patterns. If there is a pattern, then there only needs to be one code segment and the pattern is done; simple in concept, but not so simple to implement in a robot.

1. Checkpoints – Guaranteed location

One major part of the code, which is vital to the success and consistency of block stacking, is the need for the robot to know its own location on the game board. We settled on a checkpoint location such as is highlighted as #1 in Figure 1. This is the starting location; we chose this location because it is central to all three blocks. We use checkpoints because there are many different cases where the blocks could be, but many of the motions are repeated.

We ensured that the robot was capable of moving from one checkpoint to another. The thing that is so great about checkpoints is how consistent they are. In robotics, a lot of things could happen that you didn't predict, but with the checkpoint system, the robot knows where it is at every checkpoint. The robot will hit the checkpoint in the same fashion every time. This increases the consistency of all the code, and then it allows for coordinating synchronization with the other robot.

There is one major flow point, which is after picking up the first two blocks, the robot moves to the checkpoint at the downspout. The downspout is a location that needs to be reached no matter the location of the first two blocks. This is the strength of checkpoints because, after the robot reaches the downspout, Figure 1 highlight #2, the approach is exactly the same.

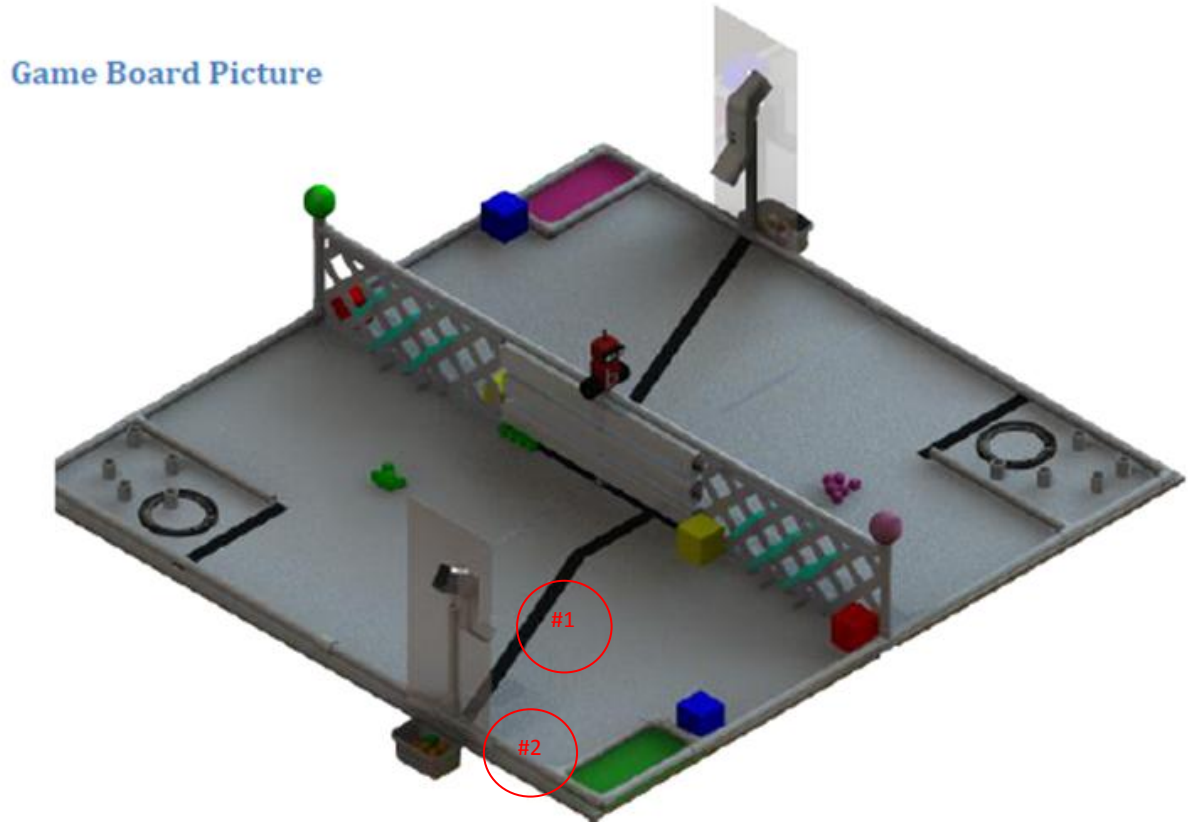

Figure 1: Game board, displaying the checkpoint locations.

2. Camera and Infrared Range (IR) Sensor – Understand your location

In the java camera class, there is a call to take a picture. What we learned is that the camera needs a 0.3 second sleep to allow the camera to stabilize for higher quality. The way that the java camera class approached picking up the blocks on the picture was called a blob. The camera class returns parameters about each blob such as blob location and confidence that helped us determine that the specified blob was read correctly. The blob parameters we used the most were the x and y coordinates defining the center of the blob for a block. The coordinates were calculated using the right–left boundaries and top–bottom boundaries. We did this for green target and all three block colors – red, yellow and blue. We used three global variables named by location. Once we learned where each block was located, the robot did not forget the locations. That is really useful, especially in the logic statements for checkpoints.

Once these basic blob parameters were acquired, the numbers were analyzed. The first check was that the center of the color blob had a green blob with basically the same center. The second check was to see if the blob was on the table: we did an if-statement analyzing the y coordinate of the center point. This eliminates a lot of the noise in the background especially when we can't control the audience's clothing. Another check we did was on the blob size: we did not want to look at anything under a certain size. The reason for this is, again, to eliminate as much noise as we could. Then the positioning is consistent and the block location is known.

Once the camera knows the color, it is easy to find the block again. For consistency's sake, we square up on every block so that it makes it easier on the camera. The camera is only taking pictures in 2D, which changes the perspective. If the camera is not square with the block, the area of the block will change. After we line up on the block of interest, we move to a certain fixed distance from the block. We can do this because we know we are straight on the block, so, as we move forward, we are approaching the block. This is where we use the IR sensor to measure the distance from the block. This is *how* we establish checkpoints. We know we are straight on the block and x distance away. This is the perfect location for logic statements because a well-chosen checkpoint will be capable of going into any branch.

Because the camera and the IR sensor will never be 100% accurate, there have to be backup methods to compensate for the errors. So, in both our square-up method using the camera and the method to get to a known distance, a margin is involved so that the robot is not just sitting in place waiting for a perfect situation. If the camera is processing but does not see any color, the camera waits for 0.3 seconds for the camera to stabilize and then tries again. The IR sensor reads a distance. For java, the number comes back in cm. This is really convenient as the desired distance could just be measured and input into our code. The program makes a quick check of the number; if the number is over 60 cm, the robot realizes that it is no longer facing the block. The code after this is crucial because once you know the problem, it must be addressed. The robot recalibrates with the camera so that it will adjust itself so that it is once more facing the blocks, and then tries it again with the IR sensor. But there are some things that cannot be corrected. For example, owing to the differences in the left-right direction at the regional competition, our robot followed a judge's blue pants thinking it was a blue block. This completely sabotaged that run.

Another problem was that the IR read the block and not also the lattice wall behind it. We ran into this problem because our robot's lift arm does not raise straight up. The robot, after grabbing the blob, is too close to the lattice and the block would rub up against the lattice wall. So our solution was just hardcoded because the lattice wall could not be depended upon to return a consistent result. Figure 2 is an example of our robot with the block. It had to back up about 5 cm off the wall before it could lift up all the way.

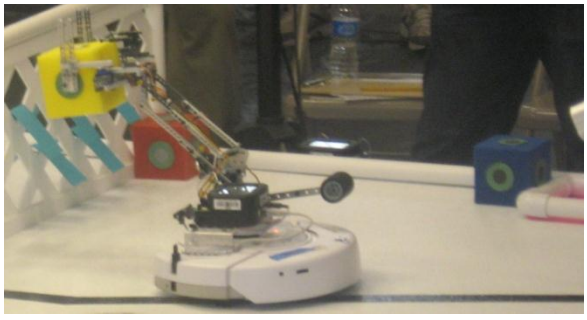

Figure 2: Robot lifting the first block

3. For the Future – Problems and lessons learned

Another problem we ran into was the unintentional consequences of code changes. Because some code segments were called in multiple places, when there was one change, it might affect other places. So whenever we made a change to a large method, such picking up the blocks, we had to test out all six cases to ensure that the robot did not behave unexpectedly. This approach saves a lot of code writing, but the coder must be very aware of the whole structure of the code. This made the code not as readable for people who hadn't spent significant time studying it. In the future, we probably should take the simpler approach of writing code for six paths so that it allowed other things in the code such as squaring up, picking up, etc., to be easily read. Also, had we written code for six paths, it would have been more code, but the synchronization would be more exact. Because we coded with checkpoints, we had to time the robot's routine according to the slowest time where sometimes the robot may have been capable of scoring more points. While our approach to coding was fun and new, I would definitely suggest doing the simplest thing that could possibly work. The code will naturally become more complicated, so starting simple is crucial to the readability of the code. Had we known how complicated just the task of stacking blocks would be, we might have chosen to code the six paths as opposed to this more complicated solution.