Greater DC/Virginia Region

Period 2 Code Review Document

Introduction:

The following code was designed for our create, named Big Roomba. Big Roomba's purpose is to sort two color poms (red and pink) into the furrows. The code for Big Roomba was written by Devin Frost and reviewed by Jacob Barats. The review was performed on March 4th, 2017.

Best Practices Checklist:

✔ Code uses functions in order to organize itself
x Code has comments that document purpose of its functions
✔ Code includes comments in general
✔ Variable names are descriptive and represent their in code use !
x Code avoids un-named numeric constants other than 0,1, or 2
✔ Code is formatted to show the flow of control !
✔ Comments that are no longer in use have been removed !

Many of our functions purposes aren't documented in comments because their names are very self explanatory. While we have un-named numeric constants other than 0, 1, or 2, we are planning on naming them in the next code revision.

Reliability:

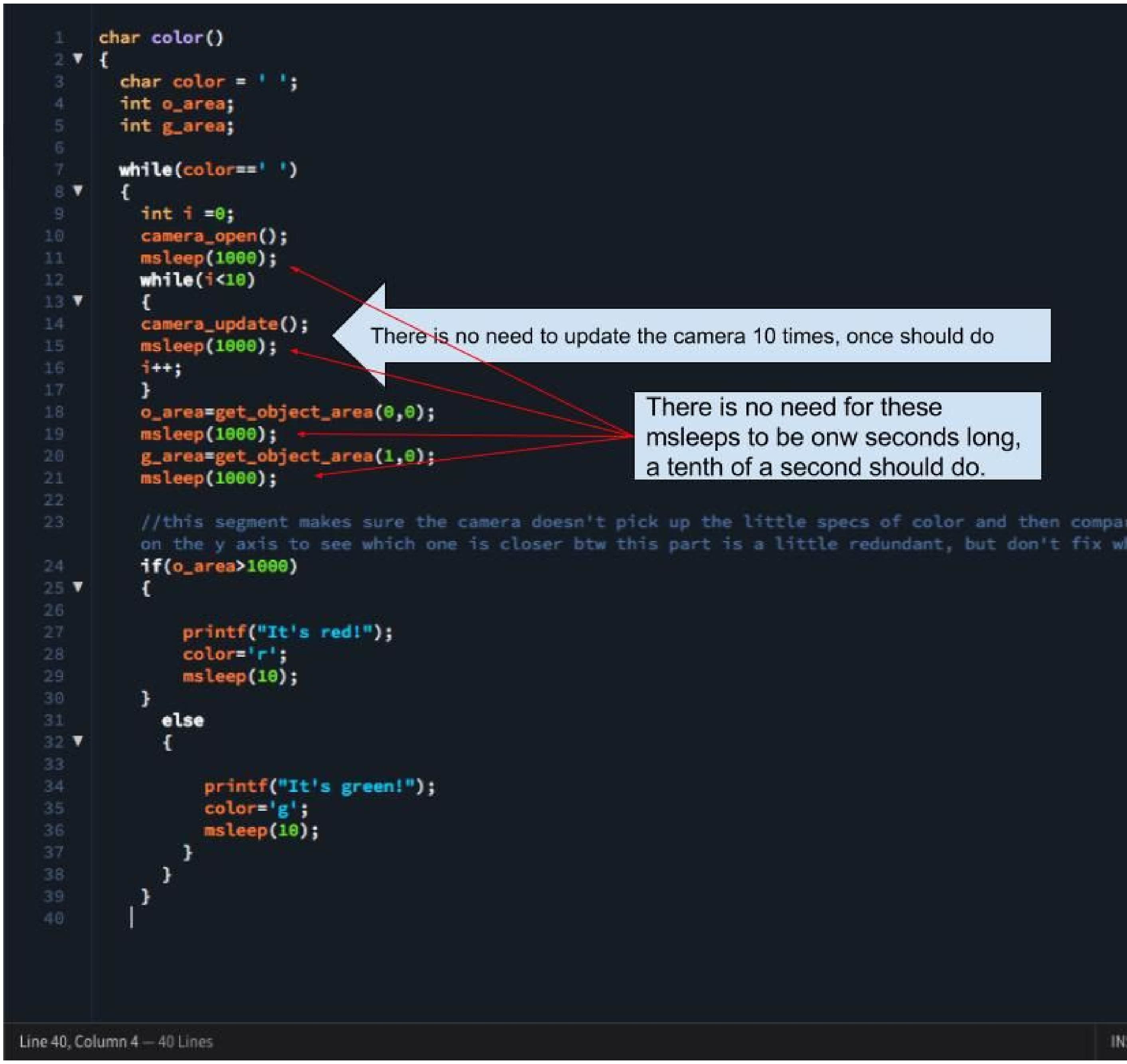
We use error detection and recovery code in our moving straight functions. To make sure that the robot goes straight, we use an if else if else statement paired with the gmpc(), checking which wheel has passed more ticks. If the left wheel has more ticks, it means that the robot has begun to drift left, so we add power to the right wheel. If the right wheel has more ticks, it means that the robot has begun to drift right, so we add power to the left wheel. If both ticks are actually the same we have both of the motors continue at the same power. It's important to keep in mind that one of the wheels goes faster than the other. To improve the reliability of the straight function, we can place more power on the wheel that naturally goes slower when the number of ticks passed are the same, so that we can continue going perfectly straight for even longer.

Maintainability:

Overall, our code can be a bit tricky to understand at times. While we use many functions, it can be easy to lose track of where the robot is when that function is being called. For example, take the turn() function. We may see it in the code, but we have to read through all of the code to see where this turn is taking place. On the other hand, our code is very modifiable. Because of our large emphasis in functions, if we discover that say the black line following is slightly off, we can just change the code for it in one place as opposed to changing it in every place where the robot line follows. Because many of the functions utilize the create's built in sensors instead of external ones, the code is highly reusable on any create. To make our code easier to understand, we could use more comments in the main() function. This would take out the tedious task of looking through all of the code to figure out where the robot would be on the board when the function is called. Additionally, each create's distance counters are slightly off. In order to make the code more reusable, we could have have more inputs in our functions, accounting for these variations in different creates.

Effectiveness:

The create robot is meant to collect the orange poms from the hopper and then obtain the pink poms from the other robot, sorting these two colors into the furrows. The create code is highly effective at collecting the poms. It uses precise movement functions, implementing the built in sensors (e.g. bumpers and degree counters) to navigate the board and collect the poms. The code is not very effective at sorting the poms once obtained. Because this is our first time as a team using camera code, we have many lines which are redundant and often use timers which are unnecessarily long, slowing the entire sorting process down. The effectiveness of the sorting code could be easily improved by removing unnecessary camera_update() functions and cutting down on the lengths of msleep() functions.



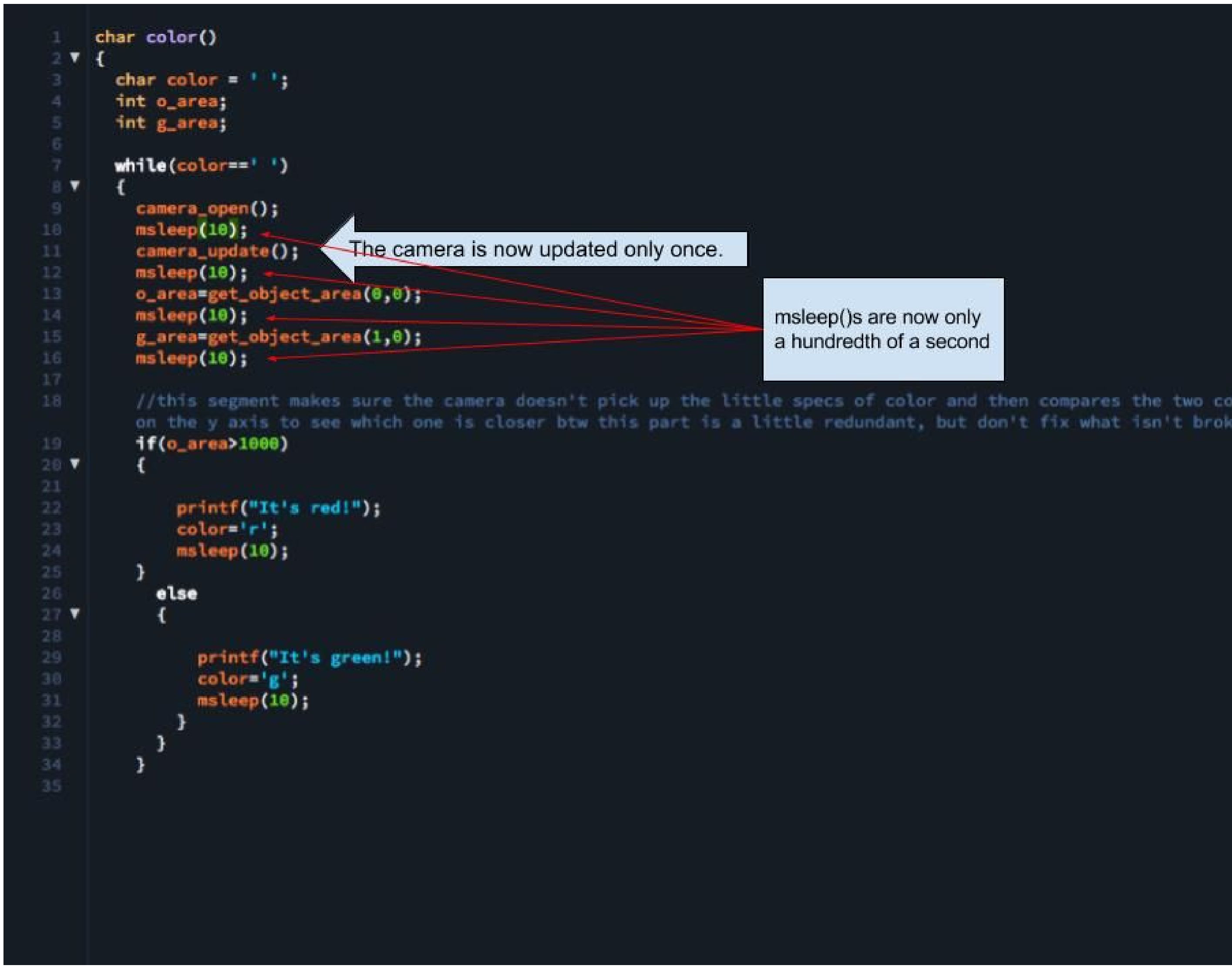
```
char color()
{
  char color = ' ';
  int o_area;
  int g_area;

  while(color==' ')
  {
    int i =0;
    camera_open();
    msleep(1000);
    while(i<10)
    {
    camera_update();
    msleep(1000);
    i++;
    }
    o_area=get_object_area(0,0);
    msleep(1000);
    g_area=get_object_area(1,0);
    msleep(1000);

    //this segment makes sure the camera doesn't pick up the little specs of color and then compa
    on the y axis to see which one is closer btw this part is a little redundant, but don't fix w
    if(o_area>1000)
    {

        printf("It's red!");
        color='r';
        msleep(10);
    }
      else
      {

          printf("It's green!");
          color='g';
          msleep(10);
        }
      }
    }
   
```

There is no need to update the camera 10 times, once should do

There is no need for these msleeps to be onw seconds long, a tenth of a second should do.

Line 40, Column 4 — 40 Lines

Above is the inefficient code



```c
char color()
{
  char color = ' ';
  int o_area;
  int g_area;

  while(color==' ')
  {
    camera_open();
    msleep(10);
    camera_update();
    msleep(10);
    o_area=get_object_area(0,0);
    msleep(10);
    g_area=get_object_area(1,0);
    msleep(10);

    //this segment makes sure the camera doesn't pick up the little specs of color and then compares the two co
    on the y axis to see which one is closer btw this part is a little redundant, but don't fix what isn't brok
    if(o_area>1000)
    {

        printf("It's red!");
        color='r';
        msleep(10);
    }
      else
      {

          printf("It's green!");
          color='g';
          msleep(10);
        }
      }
    }

```

The camera is now updated only once.

msleep()s are now only a hundredth of a second

Above is the efficient code