Final Project

# Brief Introduction to the Project

## Description

Our iRobot will intelligently calculate an optimal path to push a box to a destination while avoiding obstacles (sensed by camera). This project is interesting because we combined multiple FPGA hardwares (iRobot, camera) and AI algorithm into a working design.

The purpose is to help automate the process of moving objects from one place to another in a room. For example, our project can help warehouse managers to improve their work efficiency.

In this project, we used multiple FPGA hardware and algorithm: iRobot control, camera control, object/color recognition, and AI-based path generator.

## Real World Constraints

Robot can not move up/downstairs, poor calibration, cannot push heavy objects, poor camera quality, and grid-based path

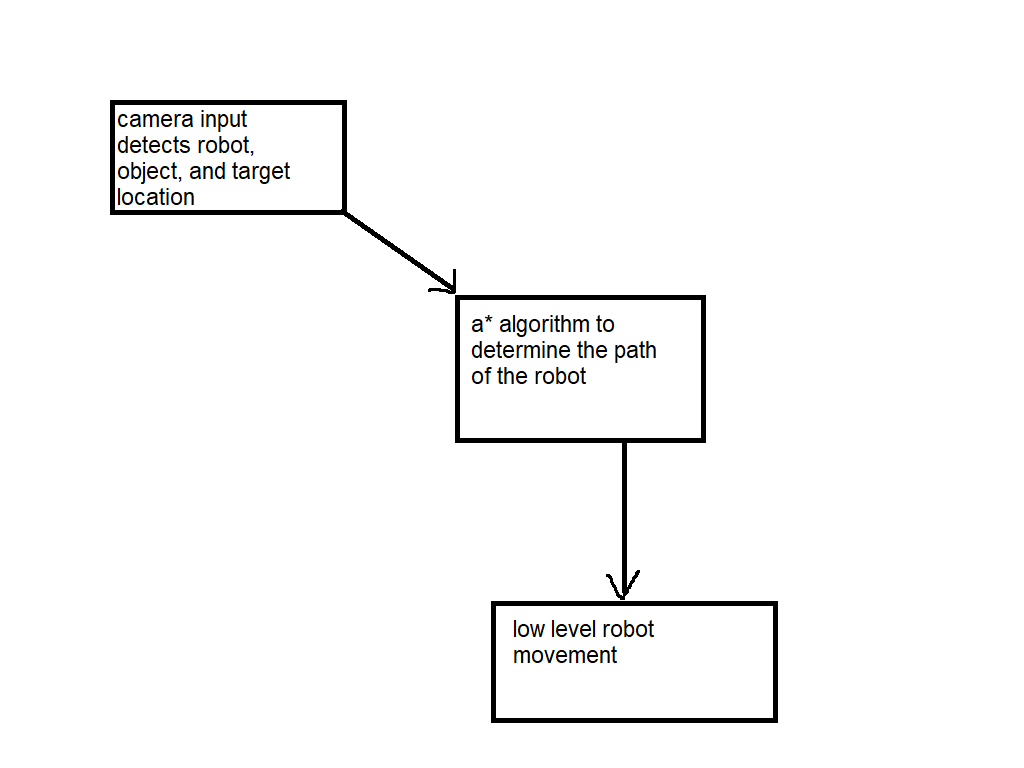
Cost: iRobot, FPGA board, camera, and electricity cost.

Should be easy to manufacture. Just need board, camera, robot, and code.

Safety constraint: should not bump into human, need a safety switch.

Real world example. For example we want to use the robot in a warehouse to help moving boxes around. First we need to make sure the robot does not run into humans or restricted zones. Also, we need to make sure even if it does, the harm that is caused by the robot is minimal.

## Brief Design overview



First, we use camera to determine the locations of the robot, obstacles (walls), box to push, and destination

The FPGA receives these information and generates a path for the robot (using A\* algorithm)

The FPGA then converts the paths into low level individual commands then sends them to the Robot

## Video Demo Links

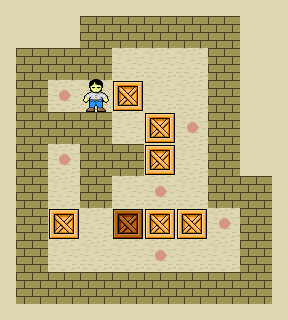
Visit the two links below for a video of our running project:

<https://drive.google.com/file/d/1jb_n-vmK--PtI9ChC39C4J7icH0t5Do2/view>

<https://drive.google.com/file/d/1yNWeFa_LRhkmNOYlrPj3Nwmye-18NmgM/view?usp=sharing>

# Sokoban AI Algorithm

Sokoban is a well-known puzzle video game that was first introduced in Japan in 1980. The word Sokoban means “warehouse keeper” in Japanese which refers to the main character of the game. In this game, the player controls the warehouse keeper (sokoban) who is placed in a closed warehouse with a number of boxes that need to be put in some predefined goal locations. The keeper can walk around the warehouse and push boxes around in order to get them into goal positions. The goal of this game is to put all boxes into goal positions in the fewest number of moves. A state is of this game is shown in below figure



# Sokoban as a search problem

1. State representation  
   Each state of the game will be represented by a 4x4 matrix. The content of each square in the game will be represented by an integer. We will assume that each state contains exactly one keeper, at least one box, and that the number of goals is greater than or equal to the number of boxes in the state.  
   The table shows a mapping between square contents and integers that we will use to represent them.

|  |  |
| --- | --- |
| Content | Integer |
| Blank | 0 |
| Wall | 1 |
| Box | 2 |
| Keeper | 3 |
| Goal | 4 |
| Box + goal | 5 |
| Keeper + goal | 6 |

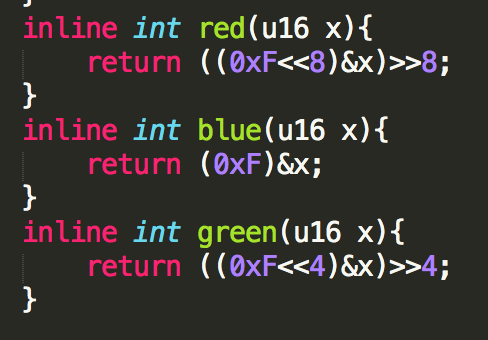
1. Successor function  
   A successor function indicates which states can be reached from a given state in one step. To make things simple, we only allow the keeper to move in four directions (up, down, left, right). Given a state of the game, each next state can be generated by moving the keeper in one of the valid directions. According to this formulation, each state can have at most four children.
2. A goal test  
   There is no specific goal that a box has to be in. Boxes can be placed in goals in any order. A box can also pushed out of a goal if needed. The game ends as soon as every box is in a goal position.
3. A cost function  
   We want to minimize the number of moves of keeper.
4. Algorithm  
   We use the most widely known form of greedy best-first search so called A\* search. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:  
   f(n) = g(n) + h(n)  
   h(n) is the estimated cost of the cheapest path from n to the goal, we have  
   f(n) = estimated cost of the cheapest solution through n  
   Therefore, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + f(n)
5. Pseudocode  
   FUNCTION A-star ( initial-state)  
   Node <- initial-state, path-cost = 0  
   Frontier <- a priority queue ordered by path-cost, with node as the only element  
   Explored <- an empty set  
   Loop do  
    If EMPTY?(frontier) then return failure  
    Node <- POP(frontier) // pop the node with lowest-cost node in frontier  
    If GOAL?(node) then return SOLUTION(node)  
    Add node to explored  
    For each action applied to node do // up, down, left, right  
    Child <- child-node(node, action)  
    If child is not in explored or frontier then  
    Frontier <- INSERT(frontier, node)  
    Else if child is in frontier with higher path-cost then  
    Replace that frontier node with child

# Image recognition Part

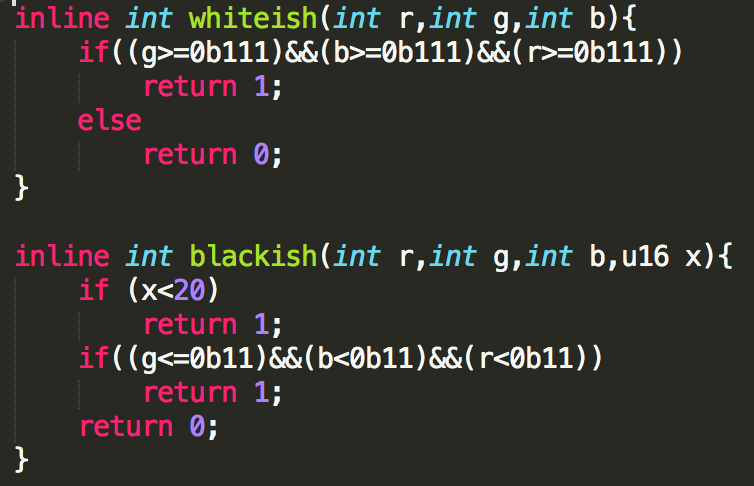
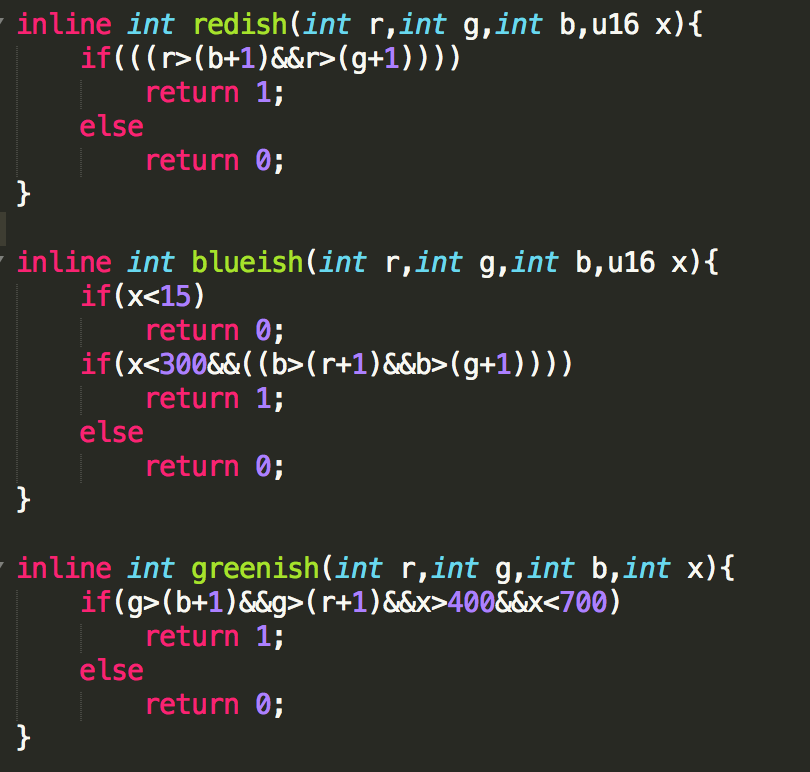
The project was intended to use Camera as the input. By recognizing the image, we are able to generate a map of 4x4 2D array containing Wall, Block, Keeper, and Goal.

## Determine the color of a single pixel

First, extract R G B value from the integer.

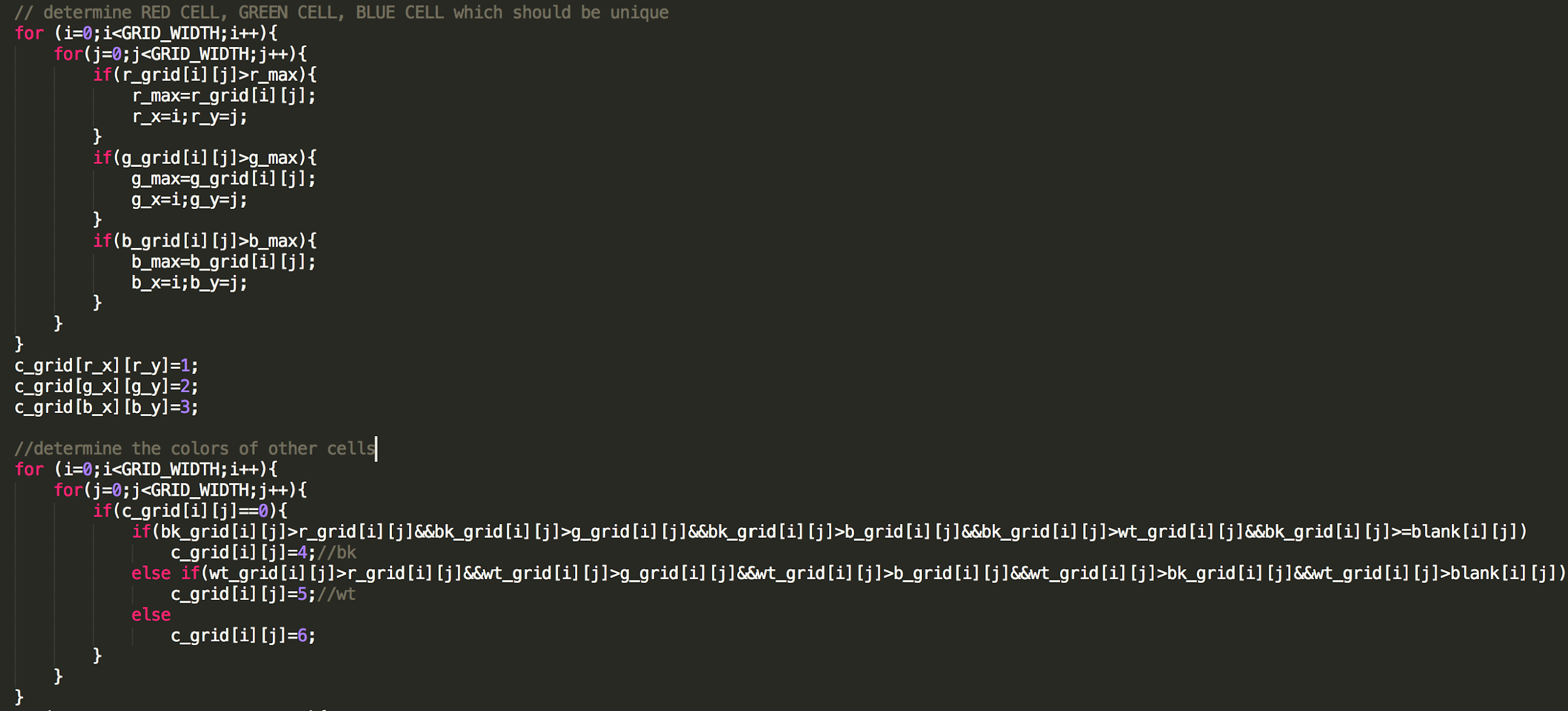


Then, we can get threshold values by observing the data read from the camera. Determine the color of a pixel by using these values.

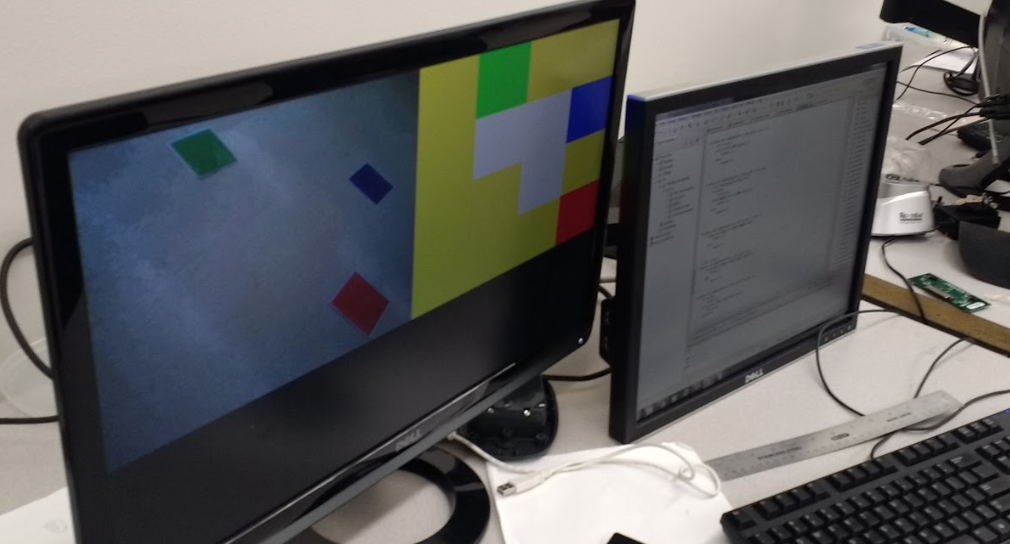


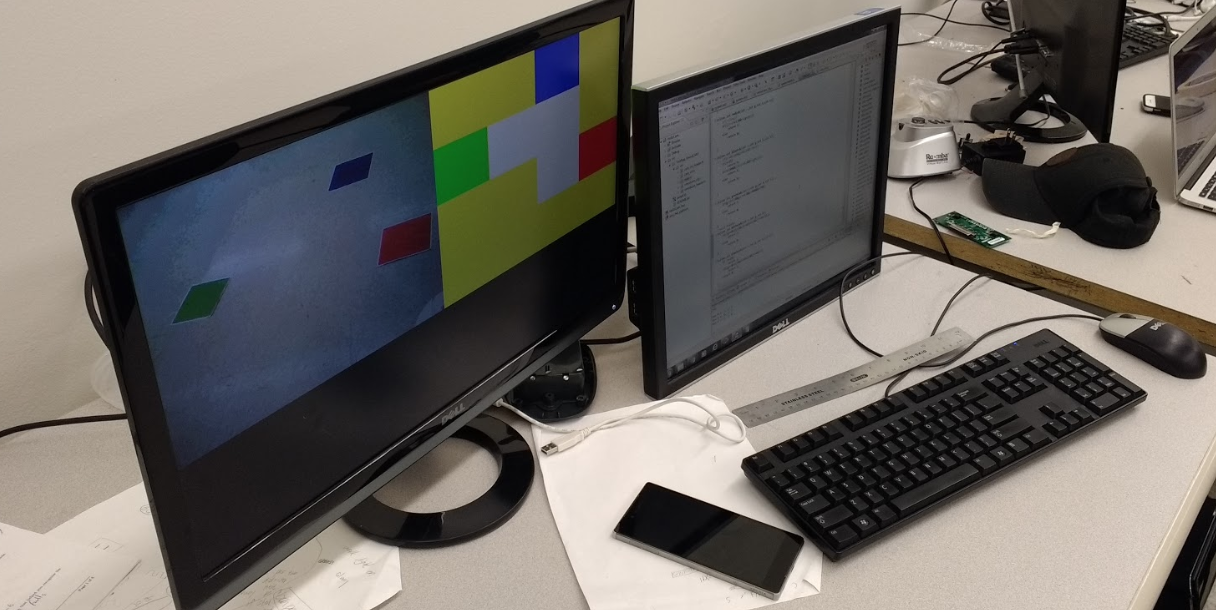
After identifying the color of each pixel, we split the whole screen into 4 by 4 cells, and determine the color of the cell by selecting the most frequent color within that cell.

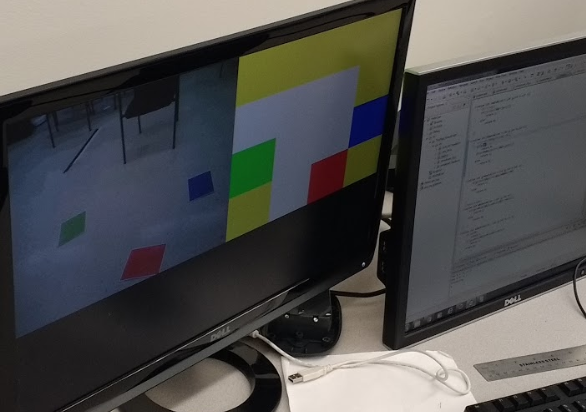




At the end, c\_grid will contain the map information.







# iRobot Control

## Brief Introduction

Using iRobot Create, we can create a moving Robot without worrying about the low level mechanical assembly. iRobot is controlled using iRobot Create’s Open Interface (OI).

The particular useful commands that we used are drive, script, and wait.

## Controlling the iRobot

The opcodes can be found in iRobot’s user manual.

To access the robot, first, we need to set baud rate and set the length of a datagram to 1 byte.

|  |
| --- |
| XUartNs550\_SetBaud(XPAR\_RS232\_UART\_0\_BASEADDR, XPAR\_XUARTNS550\_CLOCK\_HZ, **57600**); XUartNs550\_SetLineControlReg(XPAR\_RS232\_UART\_0\_BASEADDR, XUN\_LCR\_8\_DATA\_BITS); |

Now, we start the iRobot and put it into full mode so we gain full control of the robot.

|  |
| --- |
| XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR,**0x80**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR,**0x84**); |

A key point to notice here is that some commands cannot be sent immediately after another; usually a certain amount of wait time is required (about 2 seconds). By doing so, we can make sure that the robot will not ignore and discards newly inputted commands. We do so by create a function called wait2sec() which call nop to instruct the FPGA board to perform a “no operation”.

|  |
| --- |
| **void** **wait2sec**() {  **int** i, j;  **for** (j = **0**; j < **2**; j++)  **for** (i = **0**; i < **25000000**; i++)  **asm**("nop"); } |

One of the most important part of our project is to be able to instruct our robot to turn ±90, ±180, or ±270 degrees. To do so, we created a function called rotateTo(). This function takes a custom defined enum FACING. FACING can be either RIGHT, DOWN, LEFT, or UP. Since this function is quite long, below I will just present in the report a simplified and stripped down version of rotateTo() which is only able to rotate 90 degrees. The actual original rotateTo() that is in our project is capable of turning ±90, ±180, or ±270 degrees.

First, we put the robot into script mode which allows scripting. Second, we tell the robot that the script will take 13 instruction. Third, we put the robot in drive mode. In the drive mode, we need to specify the velocity and the radius. Since we just want the robot to robot instead of moving, we can just send 100 for velocity and 1 for turning counterclockwise.

At the end of the function, we need to tell the robot to stop moving by putting the robot into drive mode and give 0 and 0 for the velocity and radius.

Notice that there is a variable called counterClockwiseAngleModifier. We will talk about this variable in detail later.

|  |
| --- |
| XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x98**); // put the robot into script mode XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x0D**); // script length is 13 = 0x0D XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x89**); // put it in drive mode XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x64**); // velocity: 100mm/s XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x01**); // turn in place counterclockwise XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x9D**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x5A** - counterClockwiseAngleModifier); currentFacing = facing; // the next 3 lines stop the robot XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x89**); // put it in drive mode XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); // no velocity XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x00**); XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, **0x99**); // Play script |

Controlling the robot to move forward is done in a similar fashion. The function is called moveForward40cm().

|  |
| --- |
| void moveForward40cm() {  // This drives 40cm and stop  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x98);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x0D);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x89);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x01);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x2C);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x80);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x00);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x9C);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x01);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x90);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x89);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x00);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x00);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x00);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x00);  XUartNs550\_SendByte(XPAR\_RS232\_UART\_0\_BASEADDR, 0x99); // Play script  wait2sec();  return;  } |

Lastly, we call actFromAI(act) to retrieve the path generated by the AI and move the robot.

## Calibration

We notice that the robot cannot accurately rotate to the desired angle. For example, say we instruct the robot the rotate 90 degrees and stop. What actually happens is the robot will rotate about 95 degrees. This may seem to be a minor issue. However, since we need to move the robot a lot, these small errors will accumulate, and eventually the robot will deviate from the original intended path.

To remedy this problem, we add two variable called counterClockwiseAngleModifier and ClockwiseAngleModifier. These two variables basically adds/subtracts a certain number of degrees while rotating. After extensive experiments with the iRobot, we found out that a value of 5 for both variables will minimize the error in rotation.

# Encountered Difficulties

## Calibration Issues of the Robot

As mentioned in the previous section, the iRobot cannot accurately to a certain angle. This is due to the way how iRobot senses angle rotation (using the wheels instead of actual sensors).

This is fixed by adding some angles in addition to the desired angle. The added angle is determined using trial and errors.

## Processing Power not Enough

For the AI algorithm, we noticed that the processor on FPGA can only allow us to produce a limited number of steps (about 7). If the actual path is above that, the program will basically hang.

To fix this issue, we limit our grid to 4x4, and add some special constraints to the algorithm.

## Camera Quality Issue

The project requires us to distinguish five different colors for each object. However the camera quality is not good enough for color recognition based on our algorithm. The performance isn’t consistent overtime. Sometimes the ambient colors may tamper with our intended colors. Due to time and this problem, we had to leave the camera out in our final design. Although we do have a almost finished camera program in our project submission, it can run but just not connected to the main module.