µMouse

- CSUSB IEEE Student Chapter
- 2013

- alexkrauseCSUSB@gmail.com

consider the starting point as (0, 0) on a cartesian plane

Robot direction: use memory or look into compass sensor

microcontroller low memory
unsigned short 8 bits has data value [0-255]
Bits in hex 0x00 - 0xff
(0 - 255)
16 * 16 maze is 255, so this fits our needs

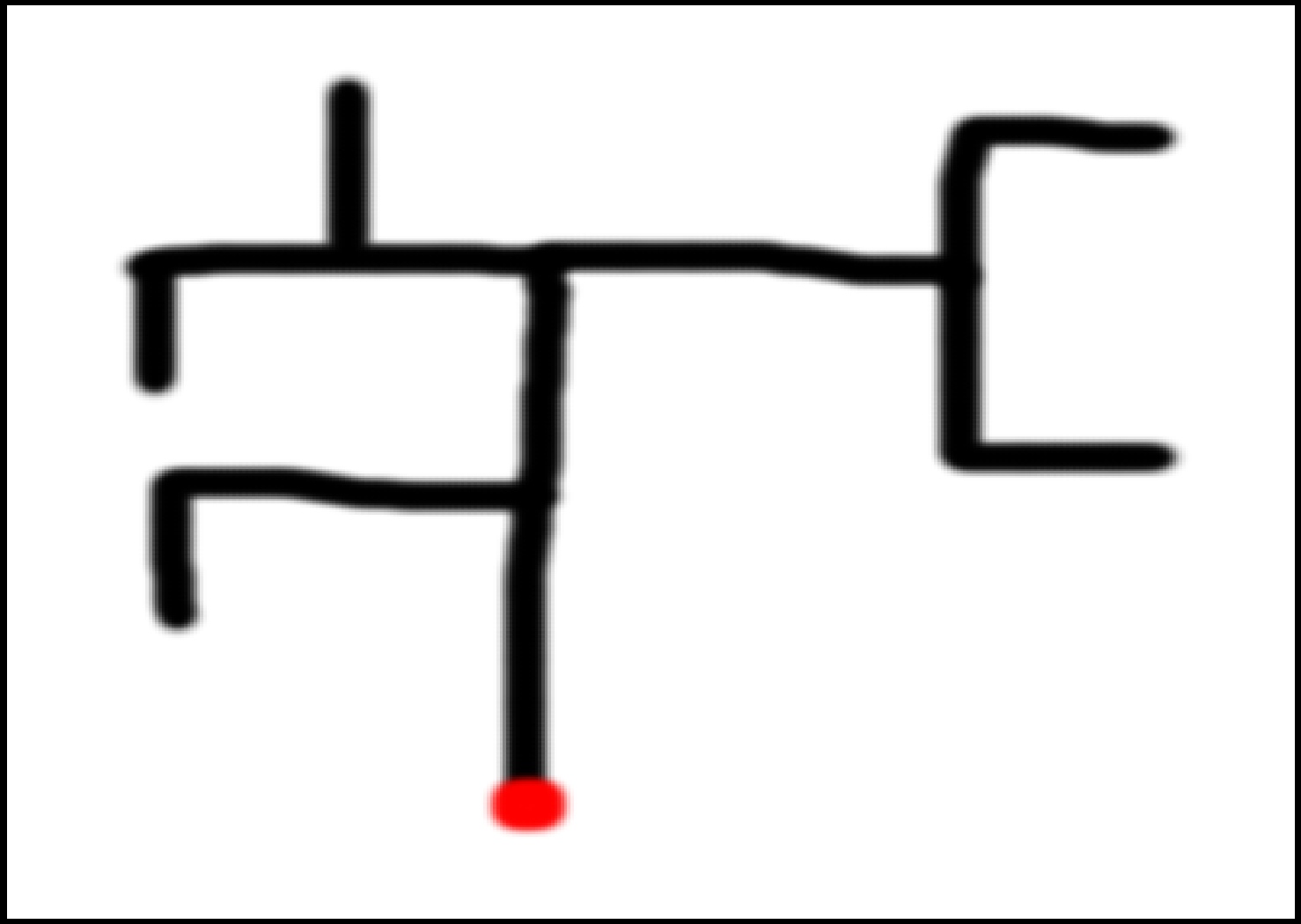
Dealing with forks

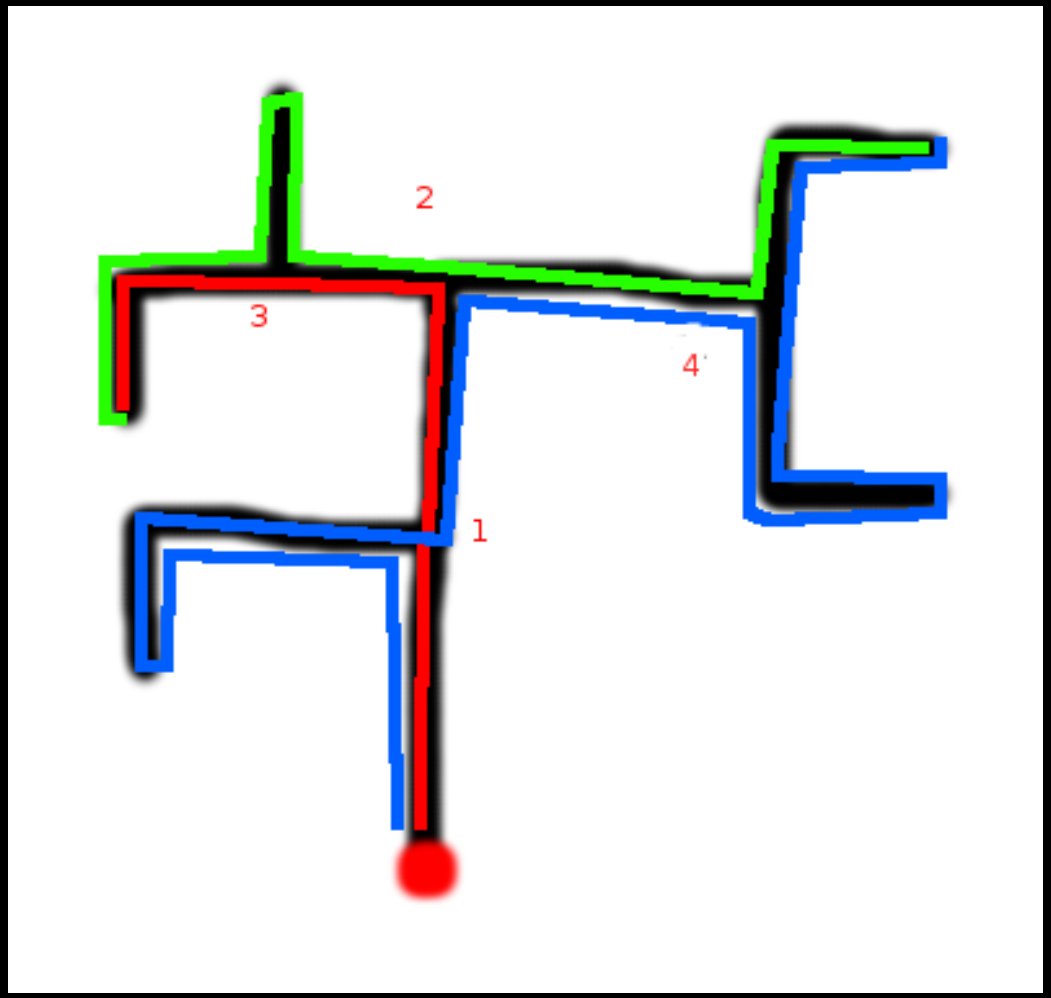
- When we get to a dead end we need a data type to keep track of the previous highest grid value which had multiple paths
- also need to have some data type in memory to remember paths where there was a fork
- if we get to a fork with two paths that both have equal grid value pick the path direction with priority (forwards, left, right)

Using a stack

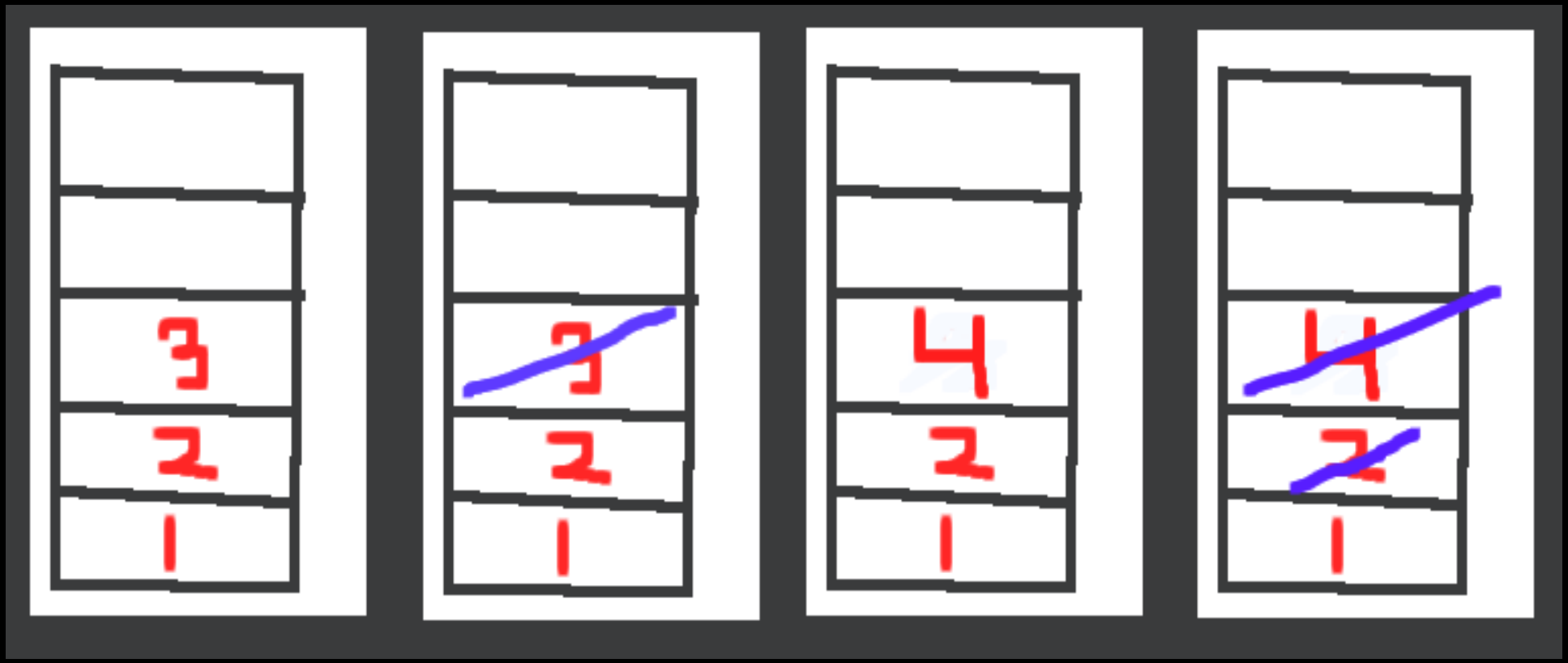
- possible stack implementation
- On movement put the direction on a stack to keep position when we get to a fork
- As we progress each fork, consider them subdivisions of the branched path
- need our code to have an overarching system which
- divides a fork into sub-sections search each sub-division and return to the main fork if no alternative is presented

Example of forking priority




2
3
4
1

Implementing the stack



- As the sub-paths are exhausted their respective forks are popped off the stack

upon reaching a fork add to a stack with sub-section being further forks

along a route come across 3 forks (maintaining normal route priorities)

each time a fork is accessed

int forkValue = 0;

std::stack<int> forkStack;

if ((Robot.pathForward() && Robot.pathLeft()) || (Robot.pathLeft() && Robot.pathRight()) || ...) { // two paths are open is some direction (forward, left, right)

// we're at a fork, let's add the fork to the stack

forkStack.push (forkValue);

forkValue++;

}

so we've gone through and now we have 3 forks on our stack

0 3

1 2

2 1

exhaust all the possible sub-sections of 3

now that 3 has been plotted and has no paths out we can take it off of our stack and backtrack to 2

forkStack.pop();

0 2

1 1

we search 2 and find no paths so we remove it from the stack

forkStack.pop();

0 1

we searched the top route of 1's sub-section now we try alternate paths

we find a path to the right and as we explore it we find another fork, so we add it to the stack

0 4

1 1

we should keep in mind all this function should do is be a method to keep track of sub-sections visited and give us a method to back-track to previous forks

we need to keep the priority of the algorithm to maintain the most efficient method

when going from 1 fork to another we should keep something in memory that keeps track of the path to the previous fork for quick recovery

when searching every fork of a sub-section when a dead-end is found with no forks in the route we should go immediately to the root of the fork

When the fork is being pushed onto the stack we have a few options on how we want to approach what we push.

We can make a forkClass which holds the parameters of a fork to hold position of the fork, some memory or function to return to it if subsequent forks are exhausted, and friend class of robot to access any necessary attributes.

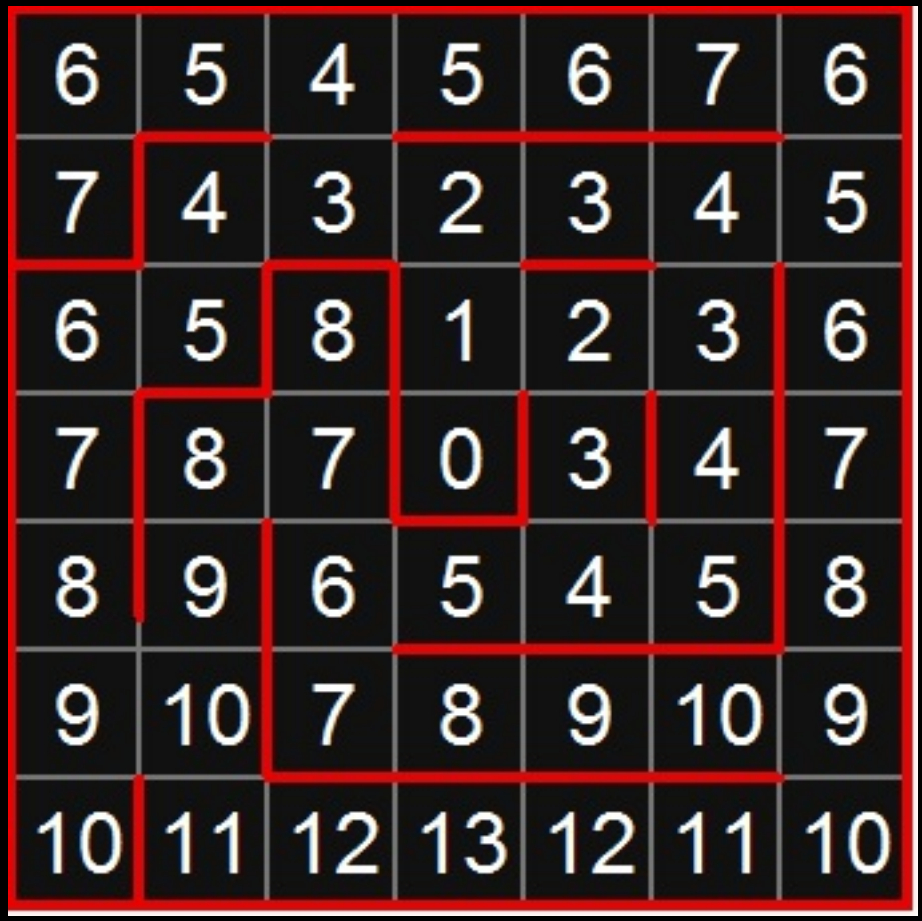
We could just push an integer and use that corresponding index value to associate to some other value such as an array, vector or map.

We could push the double position of the fork in order to maintain the corresponding positions provided we have a method somewhere else to manage returning to previous forks.

Priority

Given equal values and openings: forward, left and right take the forward route, left second and right last

- If any route has a higher grid value than the others it takes priority
- Notice in the image at the top row as the robot gets to the middle the values increase because it is getting overall further from the center
- Need a method to continually update our grid values when walls and dead ends detected



Bit manipulation

unsigned int var = 0x93; // 147 base 10, 1001 0011 base 2

- if we want to access the first four bits of var we use a mask
- unsigned int mask = 0xf0; // 240 base 10, 1111 0000 base 2
- use a bitwise and (&) to use find binary and arithmetic
-
- 1001 0011
- & 1111 0000
- ---------
- 1001 0000
-
- by using a mask comparison to see the top four bits only it doesn't matter what the second four bits are
-
- get these results use unsigned int var_masked = var & mask; // 1001 0000 base 2
-
- it shouldn't be too difficult to use basic arithmetic, temp variables and bit masking to work with double indices's for grids (#.#)

Using a double for Robot Position

- we start at position (0,0)
- one possible way to keep track of our index simply would be to use a
- double to minimize space and still keep track of our position
- moving our robot forwards or backwards we would alter the value of the
- double +1 or -1 respectively
- moving our robot left or right we would alter the value of the double
- -.1 or +.1 respectively
- // we shouldn't run into this problem if we implement properly but
- since we're subtracting need to keep error checking for bounds in mind
- don't want negatives or outside bounds

example: start at 0.0

- const double left = -.1; // declare constants
- const unsigned double right = .1;
- const double backwards = -1;
- const unsigned double forwards = 1;
-
- double initGrid = 0.0; // initialize grid
- // move right
- initGrid += right; // 0.1
- //move forward 3 spaces
- for (int i = 0; i < 3; i++)
  - initGrid += forwards; // 1.1, 2.1, 3.1
- // move left
- initGrid += left; // 3.0
- // move backwards 2 times
- for (int i = 0; i < 2; i++)
  - initGrid += backwards; // 2.0, 1.0 , ends at position (1, 0)
- so let's take a random position 4.6 (position (4,6)), we can easily
- separate this value by using temp variables or bit masking to modify
- or access our position

- class Robot {
- public:
  - Robot (double, short, short);
  - double getPosition () { return position };
  - void setPosition ();
  - short getGridValue () { return gridValue };
  - int getDirection ();
  - void setDirection (int d);
- private:
  - unsigned double position;
  - unsigned short gridValue;
  - unsigned short direction;
  - stack <forkClass> memoryStack; // ignored forks
  - int gArray [16][16]; // multi-dimensional array to map the grid
- };

```
class forkClass {
public:
    friend class Robot;
    // methods for accessing robot position
    // function for memory to remember path to previous fork
    // private attribute to increment to tell the robot which path
takes priority
    /* the first time the robot hits a path the value visitedValue
== 0, if for example the robot can go either forward or left it will
go forward if it has a higher grid value. We increment
visitedValue++ and based off of its value we can tell the robot
not to go forward again and take the second priority path
If we have gone all available paths the sub-system has been
exhausted and will be popped off the stack and the robot will
go to the previous fork */
}
```

```
Robot::Robot ( double pos, short gValue, short dir ) {
    // initialize values to corner of grid
    pos = 0.0;
    dir = 0; // forward
    gValue = 0;
}
```

when turning we want our robot to update the direction it's moving
use current position + turning position

```
int Robot::getDirection ()  {
    return direction; // private int class attribute
}

void Robot::setDirection ( int direction ) {
    // direction = (member function to get current direction) + altered
    direction
    direction = getDirection() + ( this → direction );
}
```

use modulus values (0 - 3) or come up with alternative to enumerate or keep track of direction
0 forward
1 right
2 left
3 backward

need to keep in mind combinations will alter direction (turn right, turn right will now face backwards) so if this approach is desired will need to come up with a more complete implementation of setDirection() to account for all possible turns

possible implementation:
Robot r1;
r1.getDirection (); // moving forward
r1.setDirection (2); // robot now has turned left from current direction
r1.getDirection (); // moving left

or come up with an alternative approach to keep track of direction (compass sensor?)

on each square we get update our current position, set a value to the grid square we are on, get current direction

```
void Robot::setPosition ( int direction, double prevPos ) {
    static_cast < double > ( direction ); // cast direction as a double
    so it can be used in arithmetic with doubles

    // if direction is forward
    // curPos is private class attribute

    if ( direction == 0 ) // forwards was the last direction, update
    position
        curPos = prevPos + 1; // assign class attribute curPos to
        previous position + direction

    If ( direction == 1 )
        ...
}
```

each time the robot makes 'x' wheel rotations we know that it has traversed a grid and needs to update it's information
need to reset wheel rotation counter each time the robot turns or gets to a new grid

```
const  gridSize = #;
wheelCounter = 0;
int newGrid = gridSize; // some arbitrary number to represent rotations to get to a new grid
// some movement to modify the counter happens
while ( wheelCounter <= gridSize ) {
      // more movement
      if ( wheelCounter == gridSize ) {
      // update grid value, set direction, set position, reset the counter
      wheelCounter = 0;
      }
// if direction changes
If ( Robot.left() || Robot.right() )
      wheelCounter = 0;
}
```

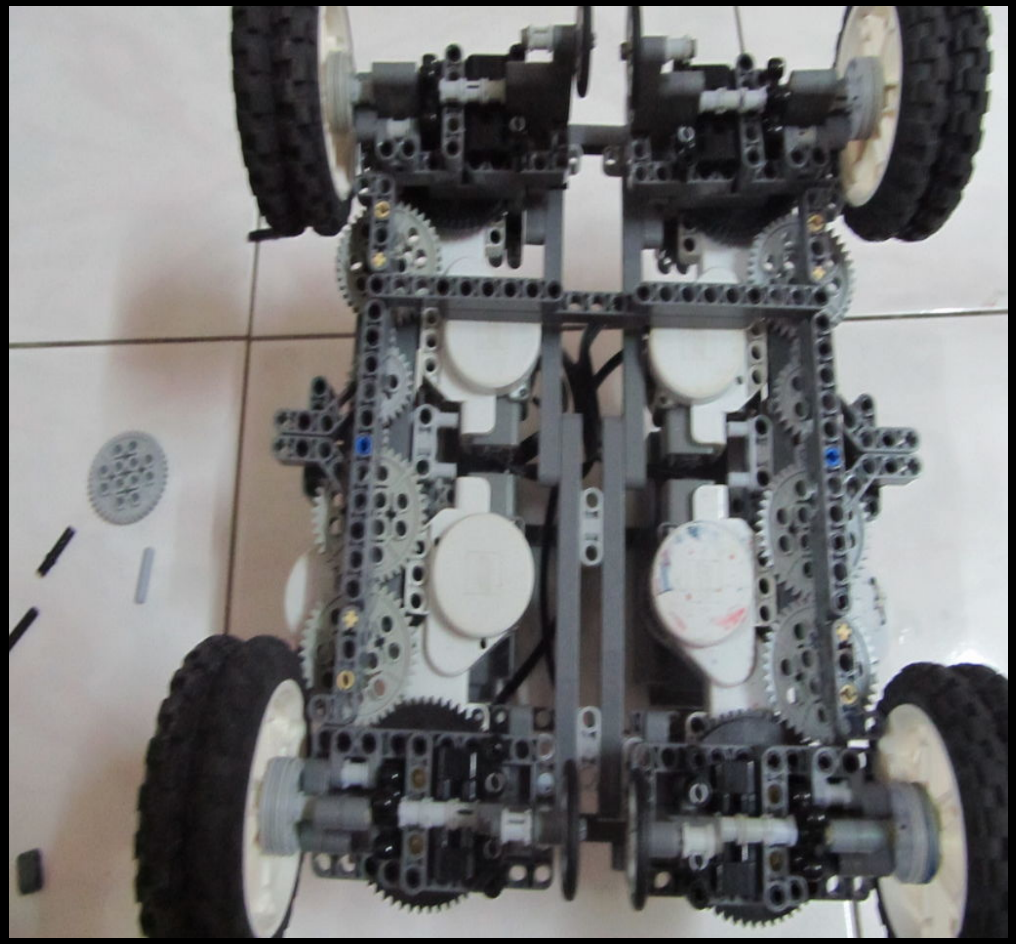
Swivel Sensors

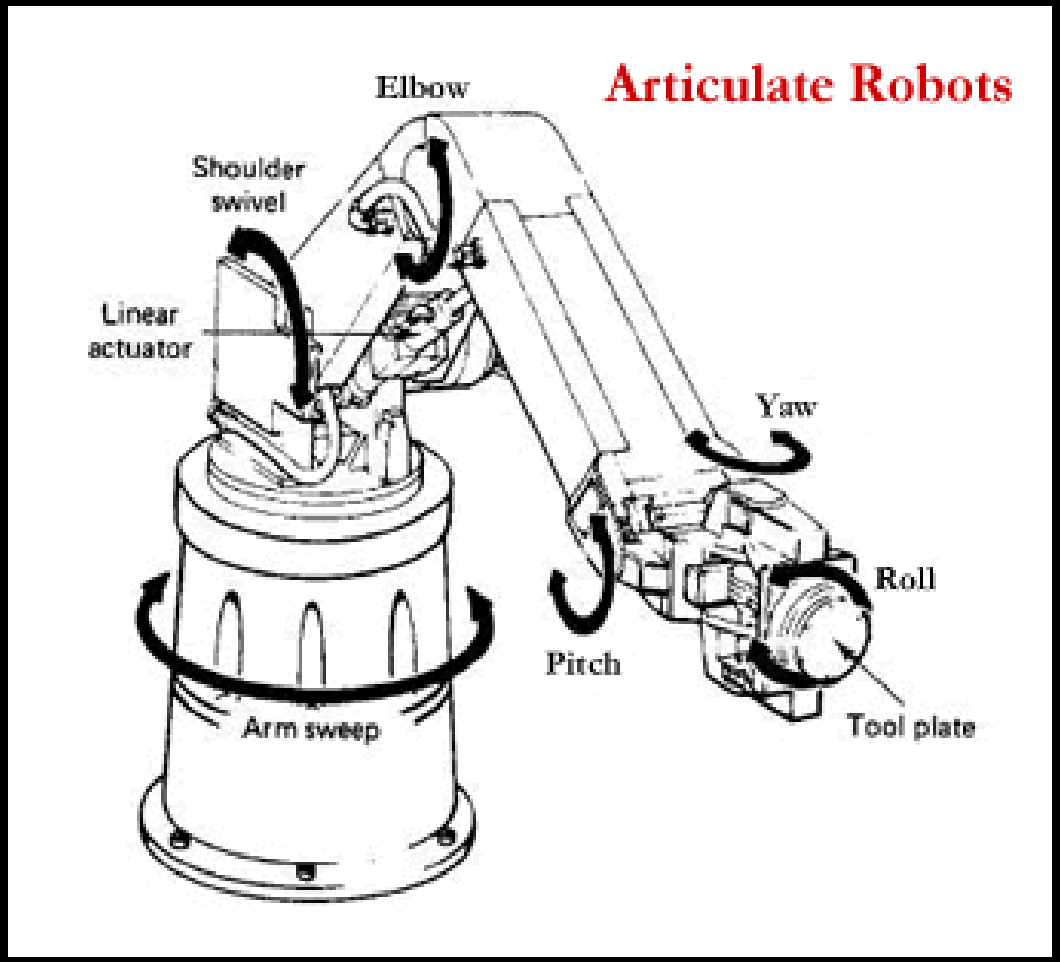
Consider four sensors detecting walls at 90 degree sections; if we could signal to the robot which direction faced is north we could assign the sensors priority live without having to turn around and having no movable top.

Alternatively, consider a swivel top where the top section which houses the sensors can rotate independently of the chassis; when the robot needs to move backwards instead of having to move 180 degrees we could move straight backwards and rotate the top independent of wheels reducing time in transition from turning.

http://www.instructables.com/id/Crab-Bot-sidestepping-robot/?ALLSTEPS

Neat NXT robot that has swivel implementations with video, worth looking at.

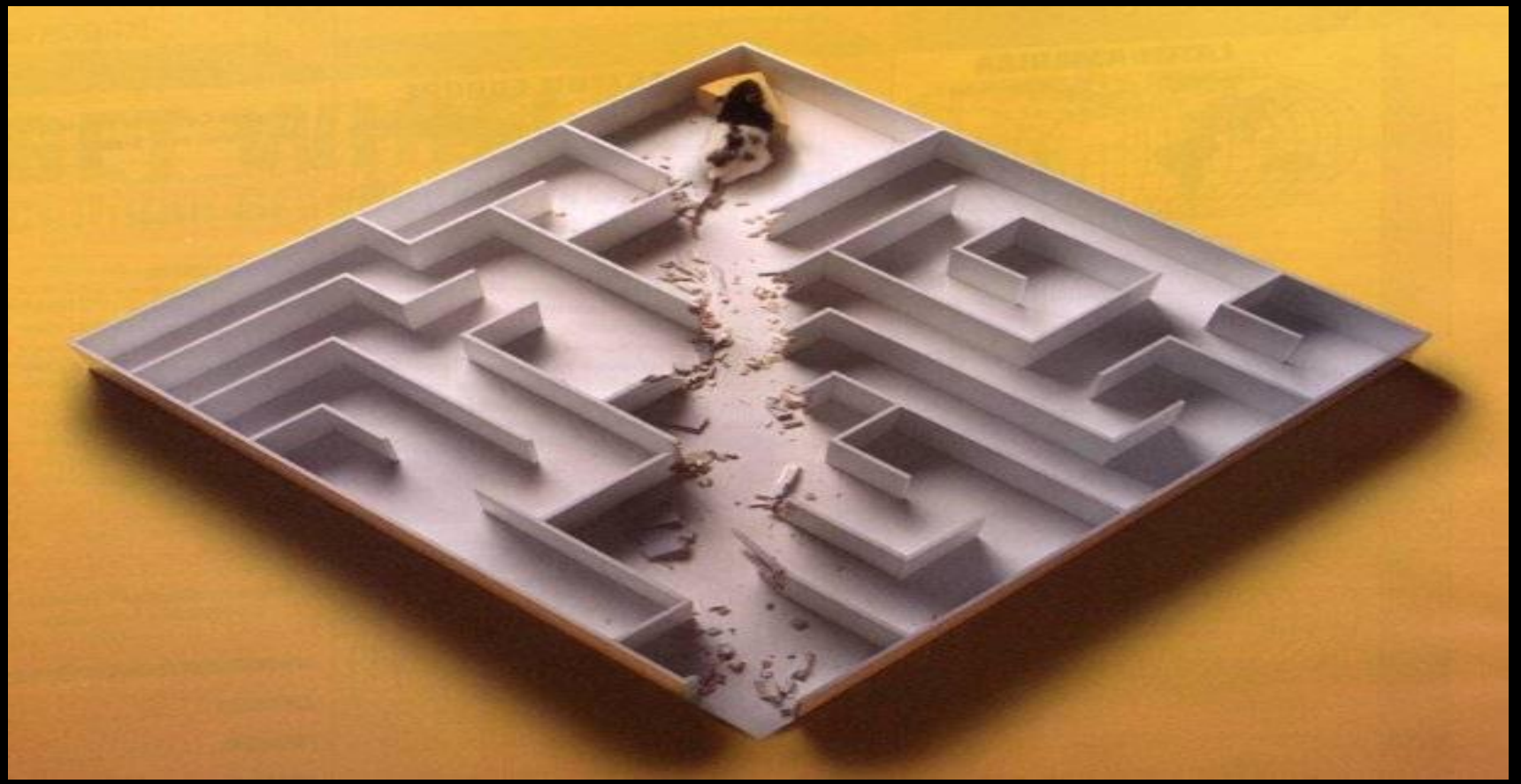




•NXT simulator
•Most of the time we don't have access to the robots
•Online there are a few simulators for the NXT, I've tried most of them and found that the best for our needs is the NXC editor and simulator. It allows simulation of the NXT and has the ability to import custom backgrounds such as mazes, and we can add sensors to the simulated robots. It is released under the GNU GPL V3 license
•NXC has a slightly different syntax than what we are using but would be a good tool to test portions of our algorithms.
•It runs off of OpenSUSE and has a disk image you can live boot off of so no installation or prior knowledge of OpenSUSE is necessary.
•It will also run off of Windows and Linux Distributions although installation of
- Java: J2SE 1.4.2 SDK or better
- Additionally for Linux: Java3D 1.3.1 or better

•Are necessary and the Linux version requires a few extra installations of “gambas“, “sox” and “ImageMagick”
•The sourceforge page is located at
•http://nxceditor.sourceforge.net/
•The disk image files are
•32 bit: https://sourceforge.net/projects/nxceditor/files/nxcEditor/nxcLinux.i686-0.1.0.iso
•64 bit:https://sourceforge.net/projects/nxceditor/files/nxcEditor/nxcLinux_64.x86_64-0.1.0.iso
•Root password for both is “toor”
•Added note this simulator has Compass Sensor (hitechnic)
•Good documentation for NXC: http://bricxcc.sourceforge.net/nbc/nxcdoc/

µMouse



- Written for IEEE CSUSB Student Chapter 2013 for the Micromouse project
- Made in LibreOffice Impress
- Alex Krause: alexkrauseCSUSB@gmail.com