

# **A 2D Drawing Program for Controlling Plotters, Engraving Machines, and Roving Robots**

**Yan Tran**

**Joseph Giardina**

University of Massachusetts Lowell

## **Abstract**

**This paper deals with the creation of a simplified software user interfaces that allow users to control mechanical devices. The software is based on the “What-You-See-Is-What-You-Get” (WYSIWYG) paradigm where the user manipulates a graphical model that the output of the mechanical devices will ultimately map to. CAD programs are a form of WYSIWYG software. Output from CAD programs are commonly used for outputting to engraving machines and pen plotters. If one considers that the command interpretation from the software to the mechanical device controls largely the movements of the engraving or plotting head, the same idea could be used to control a roving robot. The controller plots out the movements of the robot all before execution occurs. This sort of control is not difficult to understand and can be designed for use by students. In fact, a simple drawing program can be designed for use as a CAD program for an engraver and pen plotter and as a robot controller that can be used by elementary school students.**

## **Introduction**

The goal is two-fold. The first goal is to let a person use the simple CAD interface to draw an image and then convert this image into its physical representation using the Roland EGX-300 Desktop Engraver. The second goal is to let the person use the simple CAD interface to draw contiguous line segments, then have these “series of movements” downloaded and carried out by the RCX-bot. These ideas come from an NSF grant proposal by Fred Martin [Martin 2004].

Children may have difficulty in understanding programs such as Eagle, which outlines a user-designed circuit board by having steps such as drawing the schematic, creating the circuit board, etc. A child would not have any desire to use an engraver when presented with Eagle because the interface is beyond their understanding. Programs such as AutoCAD which can output to pen plotters also present a challenge to novices as it has many functions geared towards mechanical and architectural professionals. A simplified CAD program created in OpenGL allows the child to click once to start to draw a line and click again to end a line. There is also a feature that allows a child use the mouse and draw whatever they desire by holding down the mouse. These features make it fun for the child to use the program not only because the graphical program is easy to use, but because the child gets to create drawings using large tools and gets to show their drawings

off. The layout and structure of the program is based on examples done by Edward Angel [Angel, 2002].

The child, using the robot and the CAD program, can also map out a course for the robot to follow. This allows a child to visualize how the robot is going to behave without knowing the details of the code involved in creating these movements. It also promotes planning. This is because a child usually plays in real-time with a remote controlled robot without the care of where it is going to go.

Figure1 is a diagram of the the CAD program and the types of devices it is capable of interfacing with.

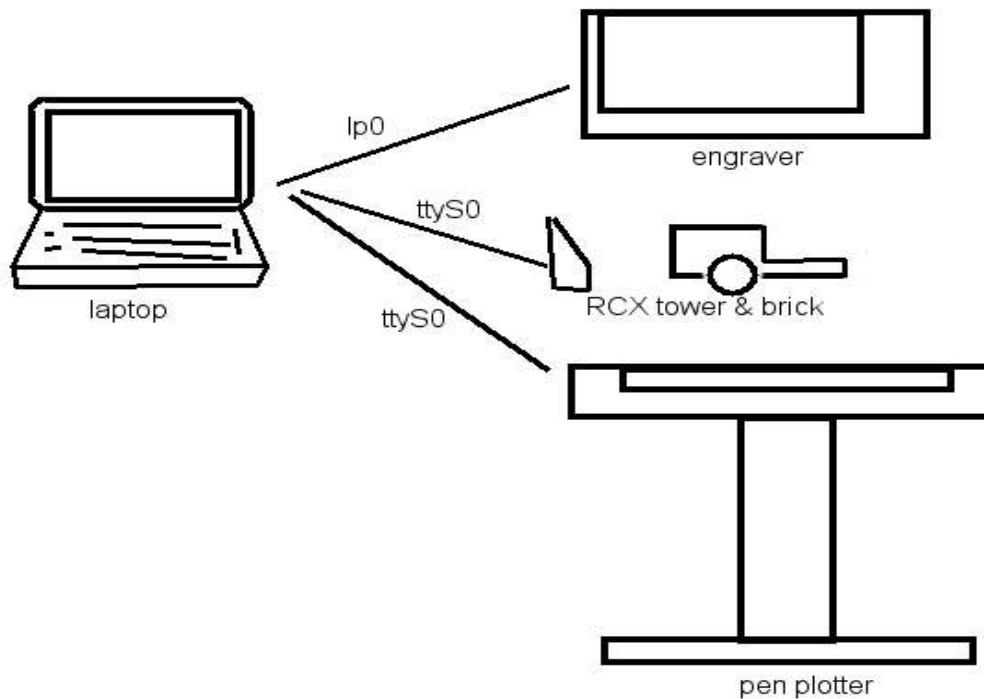


Figure 1.

## User Control

The program for drawing a picture was designed with a basic interface. This interface is familiar to nearly everyone who has used a Windows or Mac OS: drop down menus with user options. Pointing, clicking, and dragging a mouse are also an integral part of the

program and are used to create the picture after the drawing options have been set. The drawing options are as follows:

1. Create line: When this option is selected, the user clicks the mouse once to start the line then drags the mouse to where the line should end. The user then clicks again to set the line's endpoint.
2. Create Square: When this option is selected, the user clicks the mouse once to indicate where the first corner of the square should be. The mouse is dragged to another point and then clicked again to indicate where the square's diagonal should be.
3. Free Draw: When this option is selected, the user clicks and holds down the mouse to start a free draw sequence and drags the mouse around to create their picture. When the user is done with a sequence of free draw, they release the mouse.

Figures 2 and 3 show students using the CAD program the New England Botfest, a non-competitive robotics exhibition, held on March 27, 2004 [BotFest, 2004].

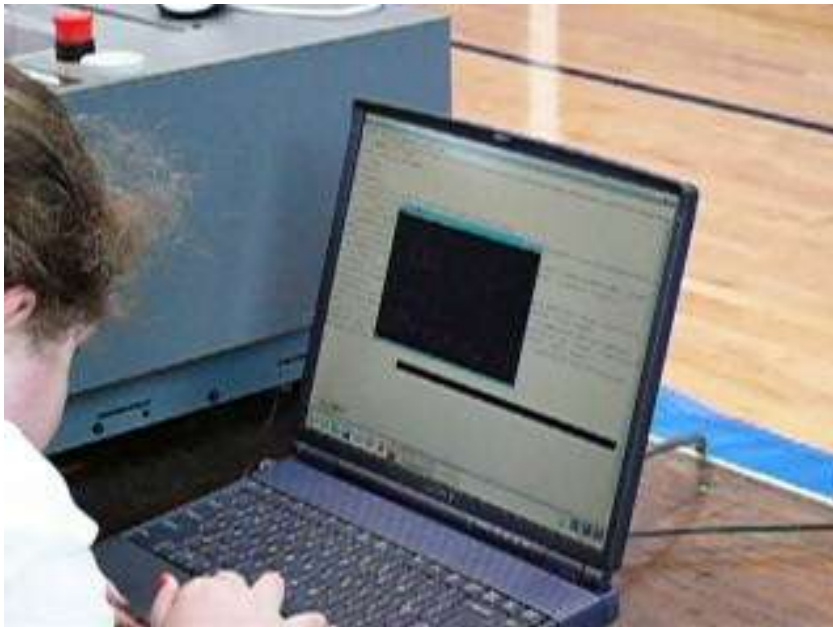


Figure 2. Students were presented with a laptop running the CAD program in a KDE window environment and drew vectors with a touchpad.



Figure3. The Roland engraving machine with white, foam-core poster board as the medium for output.

### **Basic Algorithm of RCX bot interfaced with CAD program**

The RCX bot has two motors on it, a left and a right motor.

The lines created by the user on the CAD program had to be converted into motor movements. This was done using some basic geometric calculations based on sets of points. Two sets of points  $X_1, Y_1$  and  $X_2, Y_2$  were used at a time. These two sets of points made a straight line, which could then be made into a right triangle as follows:

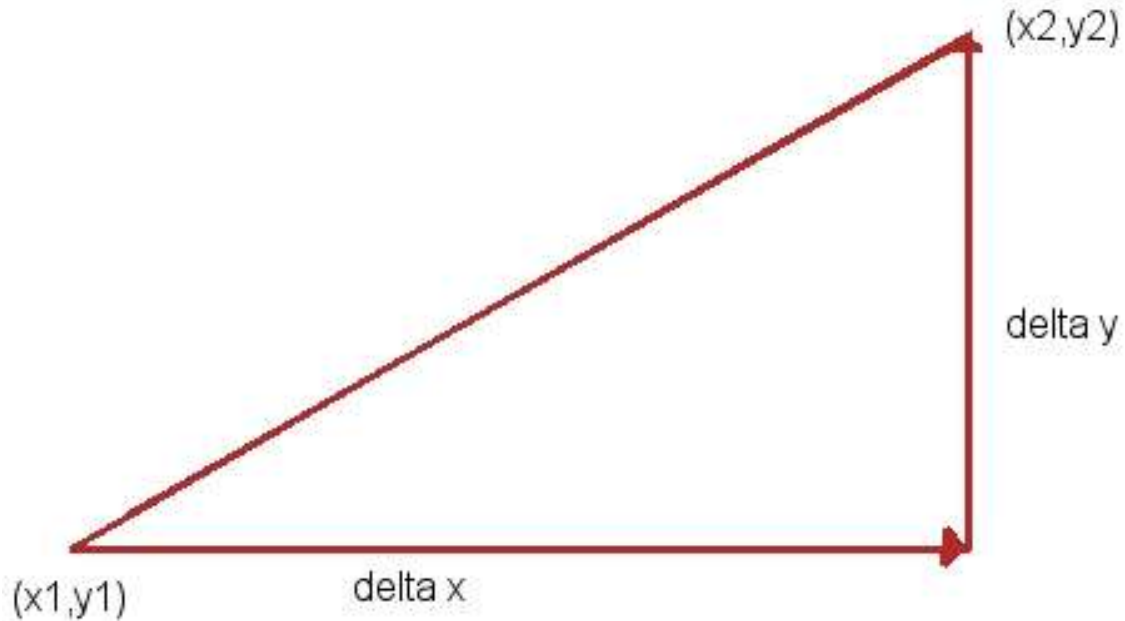


Figure 4.

The following calculations were made to determine the angle the robot had to move next using this pseudocode, where + represents positive and – represents negative:

If:	Then:
+ DELTA X and + DELTA Y	ANGLE = inverse $\tan( y / x )$
- DELTA X and + DELTA Y	ANGLE = 180 - inverse $\tan( y / x )$
- DELTA X and - DELTA Y	ANGLE = 180 + inverse $\tan( y / x )$
+ DELTA X and - DELTA Y	ANGLE = 360 - inverse $\tan( y / x )$

The calculated angle was then compared with the current angle that the robot was facing and was turned accordingly (For counter clockwise movement, when the calculated angle was positive, the right motor moved forward while the left motor moved back and for the clockwise case the opposite was true). The robot then had to move in a straight line after orienting itself in the right direction. This was just simply:

$$\text{Square root}((\text{delta\_x})^2 + (\text{delta\_y})^2)$$

## Implementation Details of RCX Roving Robot

The mathematic calculations of the above algorithm had to be translated into code that the RCX board could interpret. These were a series of motor commands for turning and

moving forward. A turn consisted of code for moving one motor forward and moving the other motor backward. Moving forward consisted of code that moved both motors forward. The CAD program generated C code that was compiled and uploaded to the robot which ran brickOS which is freely available over the Internet [brickOS, 2004]. The following is a sample of the code that was generated for both types of motor commands:

```
if(triplet_count==0){
    rotational_direction = current_value;
    if(rotational_direction == 'E'){
        break;
    }
}

if(triplet_count == 1){
    if(rotational_direction == 'S'){
        motor_a_dir(1);
        motor_c_dir(2);
        motor_a_speed(128);
        motor_c_speed(128);
        msleep(6*current_value);

    }else if(rotational_direction == 'T'){
        motor_a_dir(2);
        motor_c_dir(1);
        motor_a_speed(128);
        motor_c_speed(128);
        msleep(6*current_value);
    }
    motor_a_dir(0);
    motor_c_dir(0);
    motor_a_speed(0);
    motor_c_speed(0);
}else if(triplet_count== 2){
    motor_a_dir(1);
    motor_c_dir(1);
    motor_a_speed(128);
    motor_c_speed(128);
    msleep(current_value*10);
}
}
```

The commands were broken down into a series of “triplets”. The first command in the series of triplets or “triplet\_count” as the code indicates above, told the RCX board whether to turn clockwise or counter-clockwise. The second command in the triplet count used the first command to coordinate the motor movement, clockwise or counter-clockwise, and had its own pre-calculated or “current\_value”, for the code, to turn for. The third command in the triplet used the “current\_value” that told the RCX board how long it should move both motors forward. The files that generate the RCX code are available through the internet at <http://www.cs.uml.edu/~ytran/robotics/lab4>. The files are footer.txt, header.txt, and cad4.cpp.

## Basic Algorithm of CAD program interfaced with Engraver and Pen Plotter

There were three basic steps to make an HPGL file that could be directly exported to the engraver from the CAD program. They were:

1. Having a linked list that kept track of the endpoints of a line and the unique points created from a free-draw sequence that were generated from the OpenGL program.
2. Creating the HPGL file itself from the stored points.
3. Using a system call to export the file to the printer via parallel port.

OpenGL works by taking points and manipulating them depending on what the user wants to do with them. In the case of this program we are only interested in one basic thing: that is, creating lines. For the creation of an image in freedraw mode, all the points that the user dragged on by using the mouse are needed. All these points are connected to give the illusion of a contiguous drag. For the creation of a line though, a set of only two points is needed. One point is needed for where the line started and one for where the line ended. OpenGL itself has no way of keeping track of these points, so the linked list was implemented to keep track of each point. A node on the linked list in OpenGL has the following structure:

```
/**basic geometry data structure***/
class geomlist{
public:
    int type;          //line or rectangle or freedraw
    int freedraw_sequence;
    //1st point
    int x1a;
    int y1a;
    //2nd point
    int x2a;
    int y2a;
    //shape color
    float r, g, b;
    int fill;
    float width;
    int lstyle; //solid or stippled
    geomlist * previous;
    geomlist * next;
    geomlist();
    ~geomlist();
};
```

These nodes are then converted into an HPGL file that is just a sequence of “pen up” and “pen down” commands. The code to do that is as follows:

```
void listholder::engrave(){
    FILE *fd = NULL;
    fd = fopen("temp.hppl", "w");
    char output;
    if (fd == NULL){
        perror("open_port: Unable to open temp file - ");
        return;
    }
```

```

    }
    geomlist * temp = list;
    int count = 0;
    fprintf(fd, "IN;SP1;PA;");//initialize
    while(temp){

        if(temp->type==line || temp->type==freedraw){
fprintf(fd, "PU%d,%d;PD%d,%d;", temp->x1a*10, temp->y1a *10, temp->x2a
*10, temp->y2a *10);

            }else if(temp->type == rectangle){

fprintf(fd, "PU%d,%d;PD%d,%d;", temp->x1a*10,
temp->y1a *10, temp->x1a *10, temp->y2a *10);
                fprintf(fd, "PD%d,%d;", temp->x2a*10, temp->y2a *10);
                fprintf(fd, "PD%d,%d;", temp->x2a*10, temp->y1a*10);
                fprintf(fd, "PD%d,%d;", temp->x1a*10, temp->y1a *10);
            }
            count++;
            temp = temp->next;
        }
        fprintf(fd, "PU0,0;");
        fclose(fd);
    }
    ...

```

The following code is the 3rd step to export the file to the parallel device(the engraver).

```

    ...
        system("cp temp.hpgl /dev/lp0");
    }

```

Originally, the serial port was going to be used as the device for transferring files, but using the command set for doing so provided by the Roland EGX-300 Manual yielded nothing.

## Other aspects of the application

The drawing program that was created also imports and exports files so that the people who use the program can save their work for later use. A standard file format had to be created for this to be possible. Since OpenGL is a vector based API, each point had to be saved and the state of what type of geometry was being drawn also had to be carried along with these points. So, each line in the file format looks like the following:

geometry type, x1, y1, x2, y2

The geometry type could be a line or a point for example. For a point,  $x1 = x2$  and  $y1 = y2$ . An example of this file format is given at the web address

<http://www.cs.uml.edu/~fredm/courses/91.548-spr04/student/ytran/lab4/temp.yjp>.

Another aspect of this application was that it created a “snapshot” that made the OpenGL preview picture of the path of the RCX bot or the engraver visible via world wide web. All the user would then have to do is to cut and copy the html code generated and put it on their website. An example of a preview picture is shown in Figure 5.

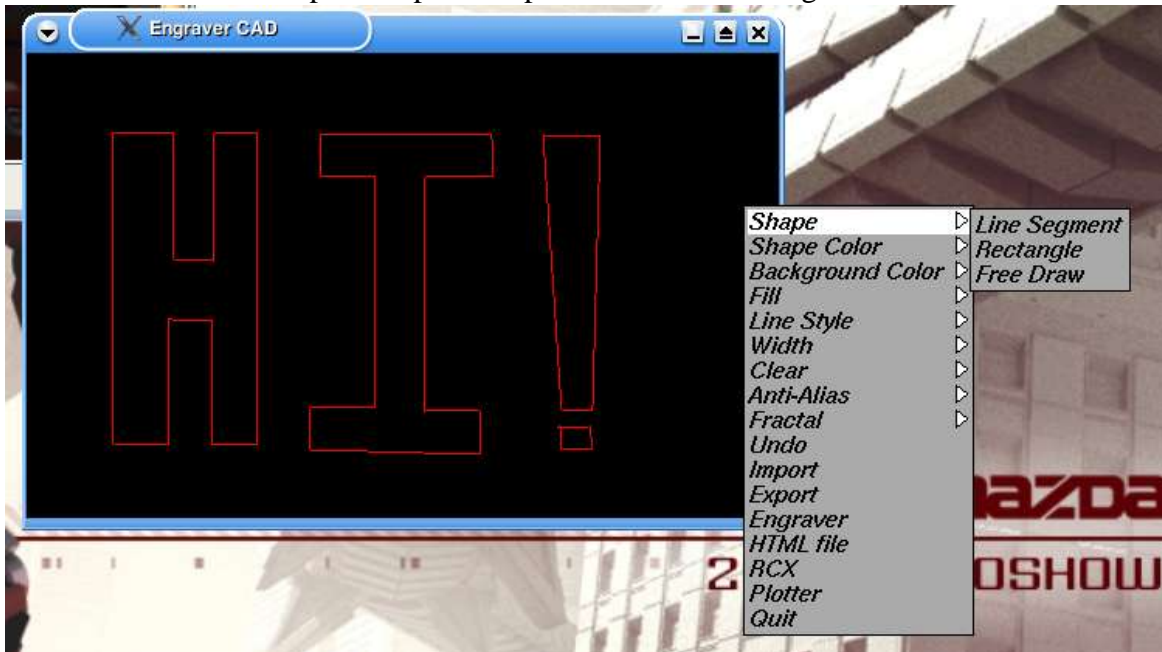


Figure 5.

## Conclusion

The purpose of this interface, although not realized at the time we started with the project, was its extensibility to interact with other hardware. For example, if we had time to continue this project, we would have built an RCX-Bot that dynamically created a map. The RCX-Bot would have been placed in a maze structure and would have traversed the maze, while at the same time, creating a map based on the robotic sensors to be displayed on a graphic program such as OpenGL. This could be useful in the area of search and rescue using relative positioning and using the robot for compiling data such as temperature and displaying it back as hot(red), warm(pink), or cool(blue), for example. Other data could be compiled and use. Our project was just a beginning point.

## References

- Botfest 2004. "<http://www.cs.uml.edu/botfest>" Description of the New England Botfest at UMass Lowell.
- Martin, Fred (2004). "iCricket/iDesign: Embedded Computing and Shared CAD." Proposal submitted to the National Science Foundation, January, 2004.
- Angel, Edward (2002). "Interactive Computer Graphics, A top-down approach with OpenGL (third edition)."

brickOS (2004). “<http://brickos.sourceforge.net>” Website that contains the source code and documentation of brickOS.