Christine Talbot

ITCS 6150 / 8150

Project Report

Due: 12/2/2010

PROBLEM DESCRIPTION & OBJECTIVE

Finding a path through an environment while avoiding obstacles can be a challenging task for a robot. Completing this optimally is even more difficult. This application attempts to provide a method for achieving such a task. The user is asked to setup a random environment, place the robot in the environment, and tell it where its goal is within the environment. The robot is allowed to achieve offline planning prior to moving about in the environment and therefore has full knowledge of the environment.

The objective is to utilize planning with an A* algorithm to allow the robot to plan out its moves in order to find an optimal path through this environment without hitting or going through obstacles. Ideally, I had hoped to also expand this to encompass online searching through the environment with LRTA* algorithm usage as well in order to compare those two methods. However, due to time constraints, only the offline planning methods were utilized.

APPROACH & MODEL

I approached the project by providing a GUI interface that allows the user to create obstacles (polygons) within an environment, choose a starting point and a goal point, and output the optimal path to the screen.

The environment is translated for the robot into sets of vertices that are within sight of each other, and therefore are locations that can be visited by the robot. These vertices are formed into a graph where each vertex of an obstacle (or the start and end points) are a node and each pair of nodes that are within sight of each other are an edge.

The edges of the graph provide the capability of providing the successor function that is required by the A* algorithm. To avoid excessive node generation while running A*, duplicate nodes in the prior path or that are in the queue to be expanded are eliminated.

PROGRAM EXECUTION & EXAMPLES

The program was written to run on Linux with the FLTK package for graphics. It can be built by running "make" on a Linux box, followed by executing the "robot" executable that gets created by the compile. Upon completing this, a window will appear that looks like Figure 1.

Once the application is started, you have a few options that you can take. The first is to manually draw the environment you wish to work with. To do this, you select the "Choose Polygon Points" menu item from the "Draw Polygons" menu, as can be seen in Figure 2. Once the menu is selected, you can click anywhere on the screen to choose the points for obstacles in the environment. The points you choose will be highlighted by yellow dots on the screen (Figure 3) until you choose to draw finish the polygon. When all the points have been chosen, you can either right-click to draw the polygon (for a shortcut), or choose the "Draw Polygon" from the "Draw Polygons" menu (Figure 4).

Figure 1

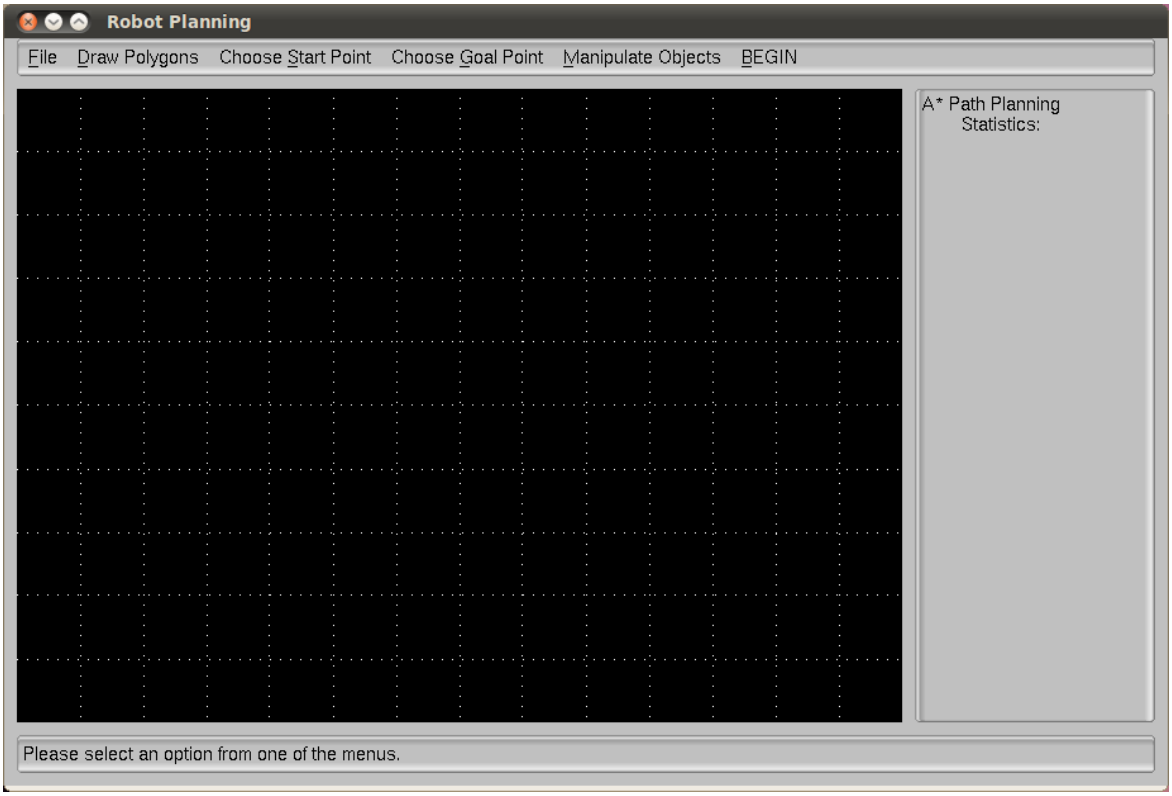


Figure 2

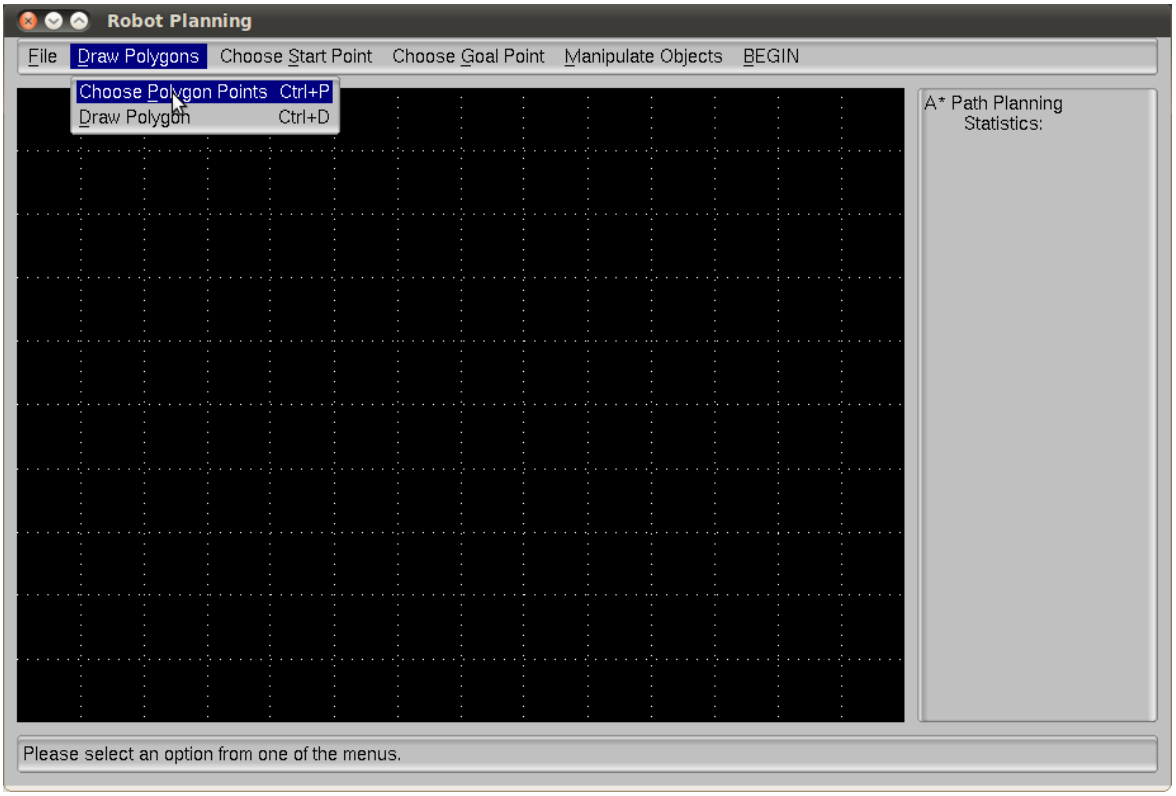


Figure 3

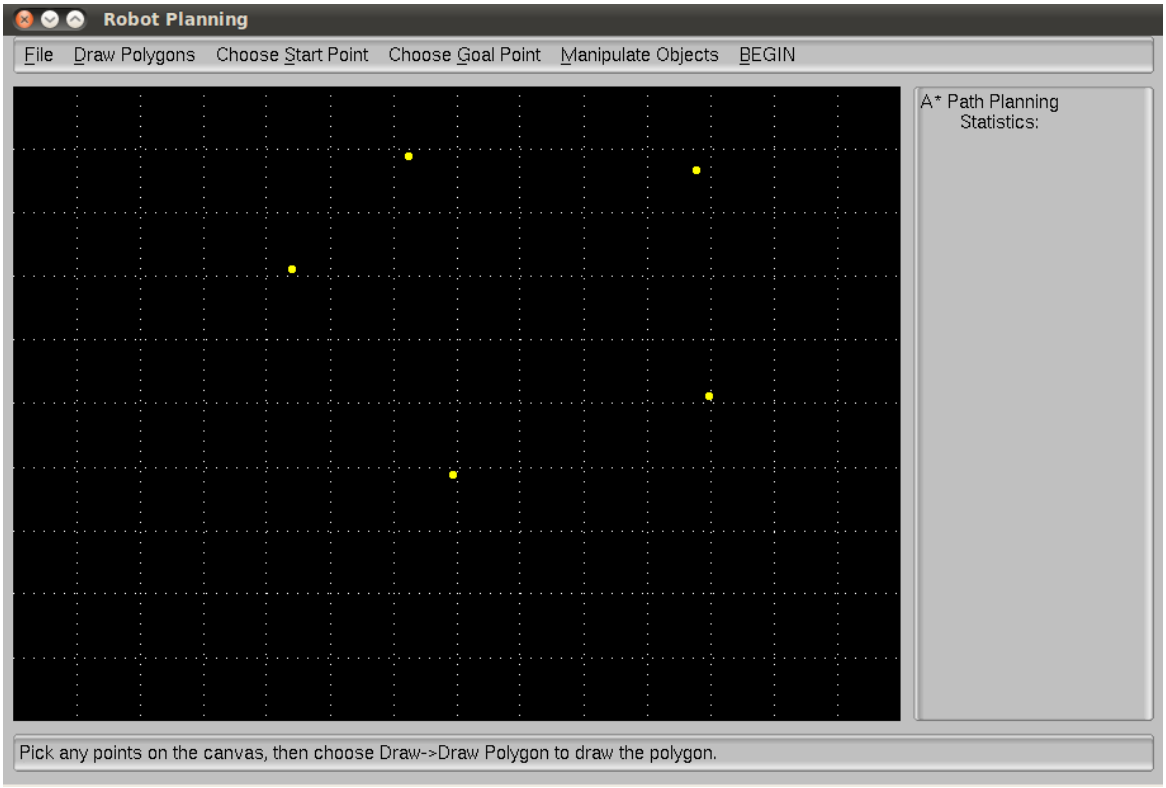
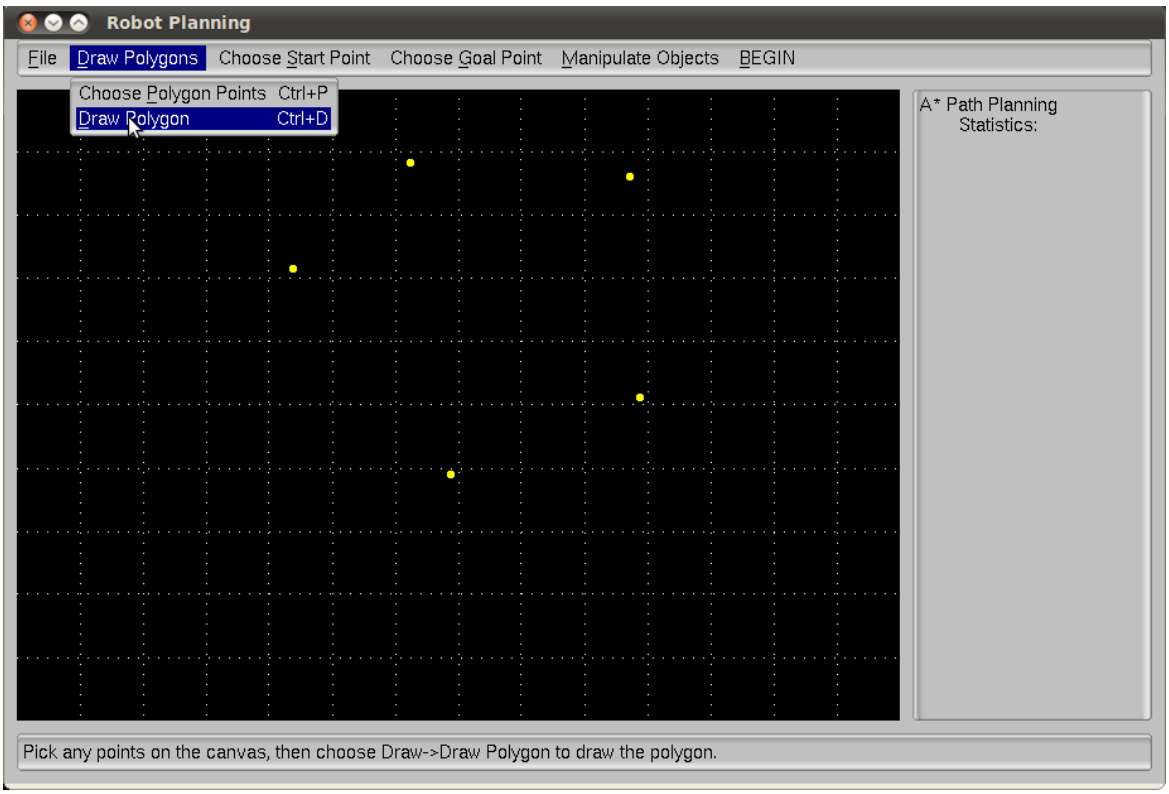


Figure 4



This will create an obstacle in the environment, as can be seen in Figure 5. You can repeat this, as needed, to create as many obstacles as you desire within the environment. Once all of your obstacles are created, you can choose where you want the start and end point to be within the environment. To choose the start point, select the "Choose Start Point" item from the menus and click anywhere on the screen (Figure 6). If you click multiple times, it will move the start point to where you last clicked. For the goal / end point, choose the "Choose Goal Point" item from the menus and click anywhere on the screen (Figure 7). The start point is always in red and the goal point is always in green.

You can save the current environment by choosing "Save File" from the File menu (Figure 8) and giving the file a name (Figure 9). The name defaults to "save.txt" in the current directory where the program is running. You can save the file before or after running the path finder portion of the application.

Figure 5

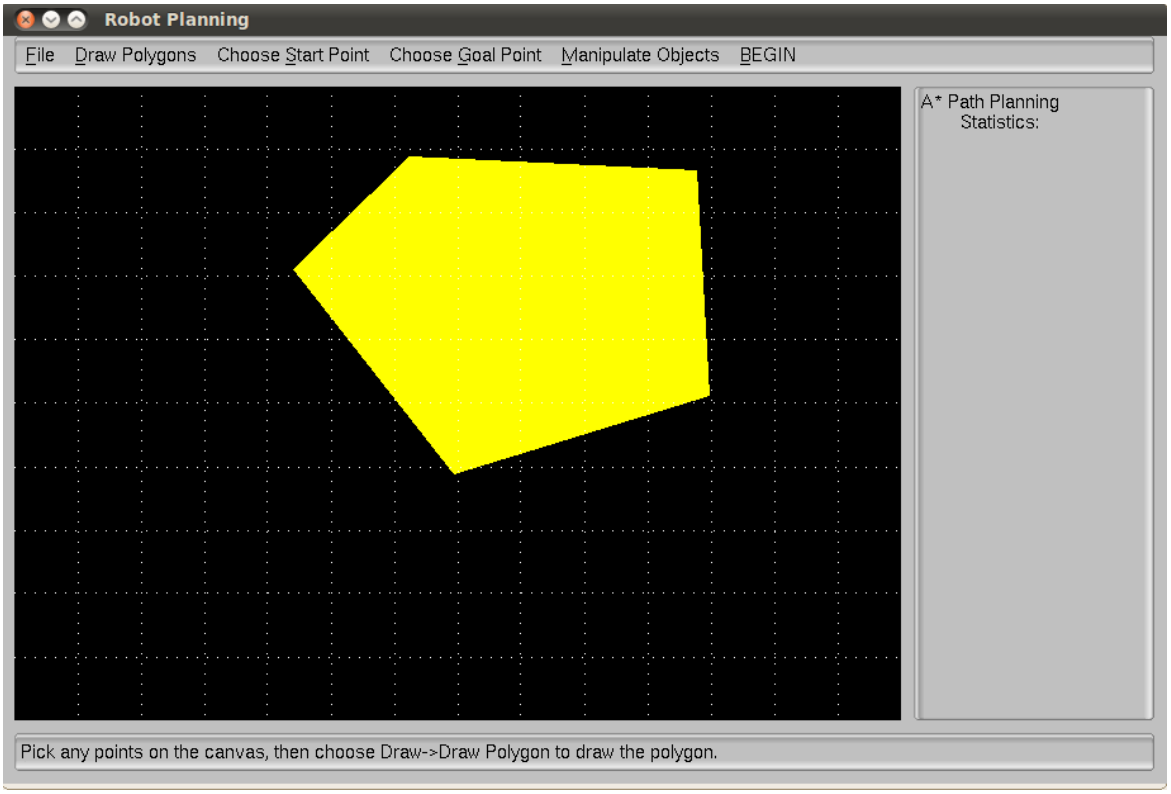


Figure 6

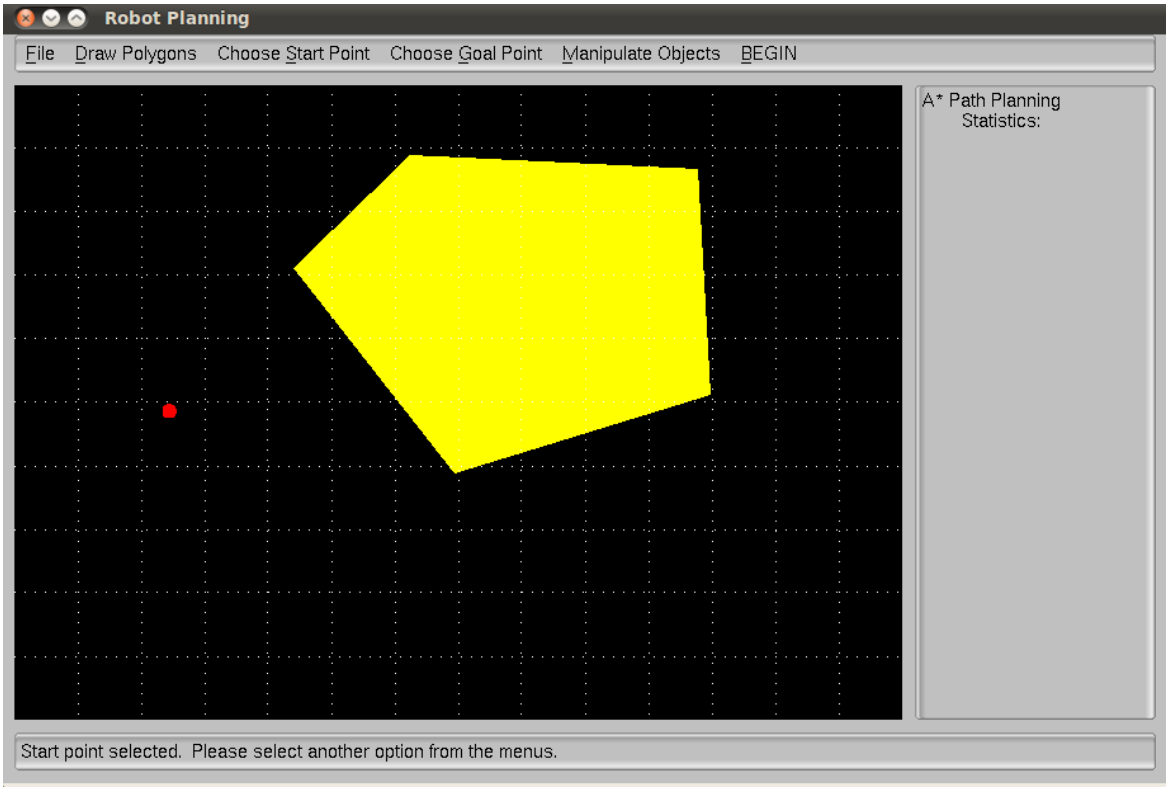


Figure 7

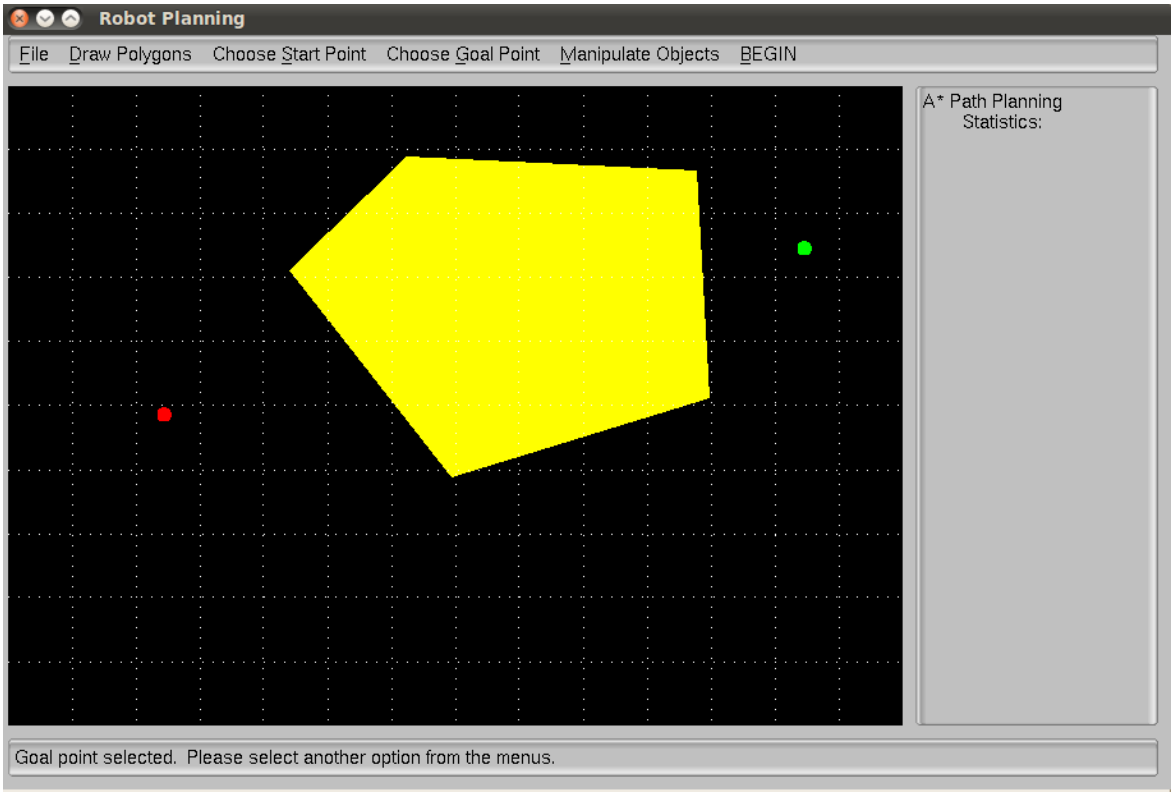


Figure 8

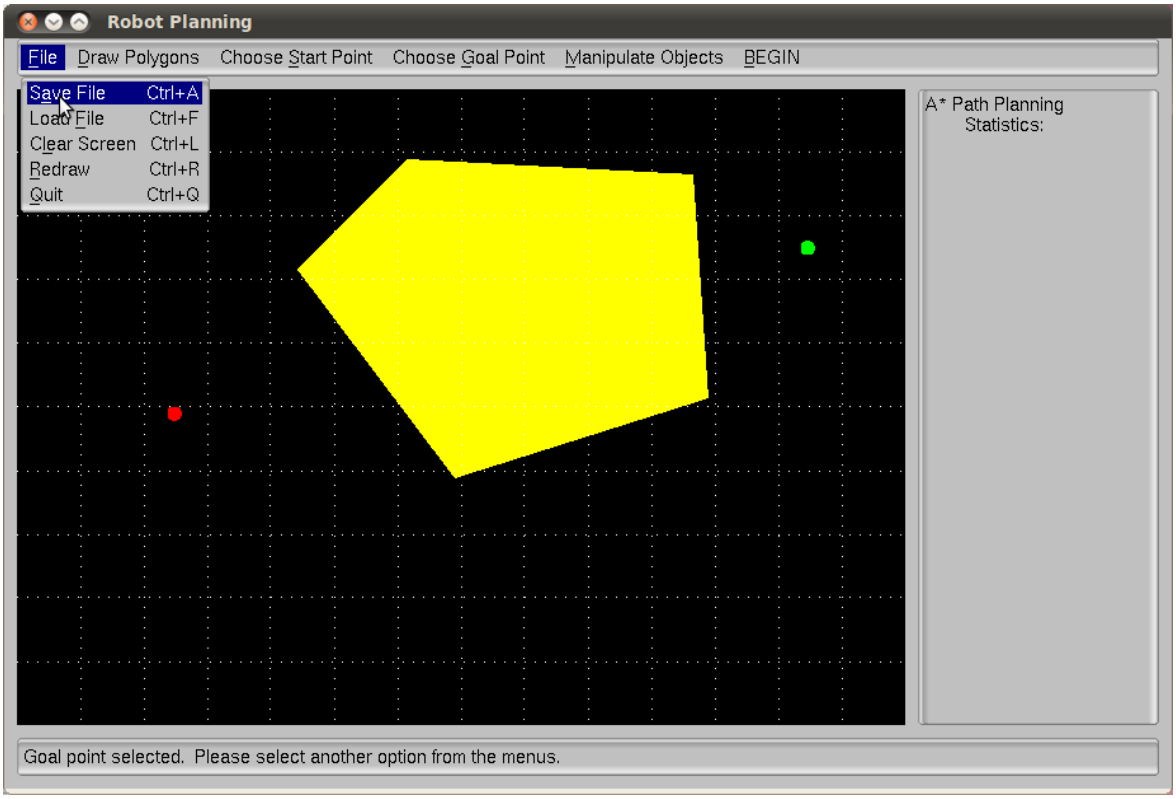
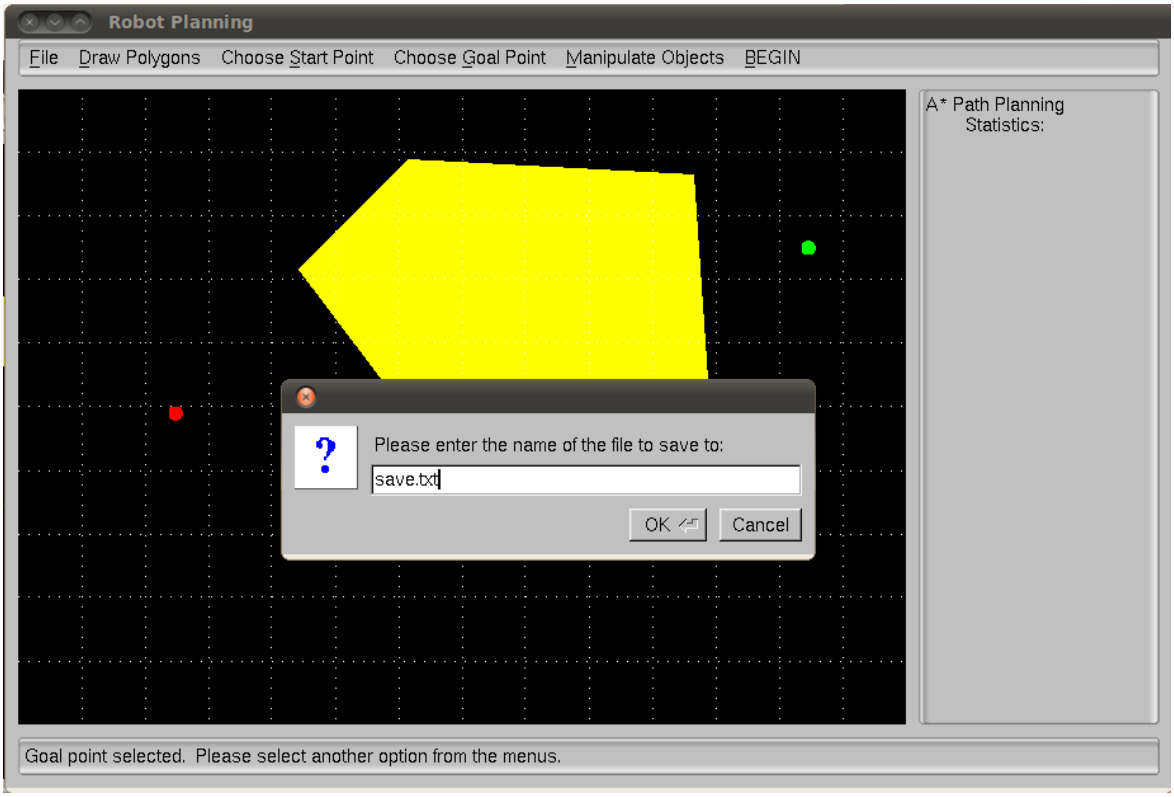


Figure 9



You can also load this file (or any previously saved file) to be the current environment by choosing the "Load File" from the "File" menu (Figure 10). This will bring up an explorer that will allow you to browse for the *.txt file you want to load (Figure 11). Four examples have been provided as part of this project (save0.txt, save1.txt, save2.txt, and save3.txt) and will be shown later in this paper. Once you choose a file, it will load to the screen (Figure 12) with all the saved information (just the obstacles and start and end points if saved prior to running the path finder). If the file was saved after running the path finder, you will see the solution that was found as well.

Even though you load an environment from a file, you can still modify it for a subsequent run. This can be done by choosing the "Choose Object" menu item from the "Manipulate Objects" menu (Figure 13). Once that item is activated, you can click anywhere on the screen to select an object. Based on where you click, it will choose the object with a point closest to your click and highlight it orange (Figure 14). It also allows you to choose the start or end point in the same manner and will turn that point orange (Figure 15). From there, you can choose the "Delete Object" menu item from the "Manipulate Objects" menu (Figure 16) to delete the selected object (Figure 17). You can also continue to create additional objects by repeating the polygon creation steps mentioned earlier.

Figure 10

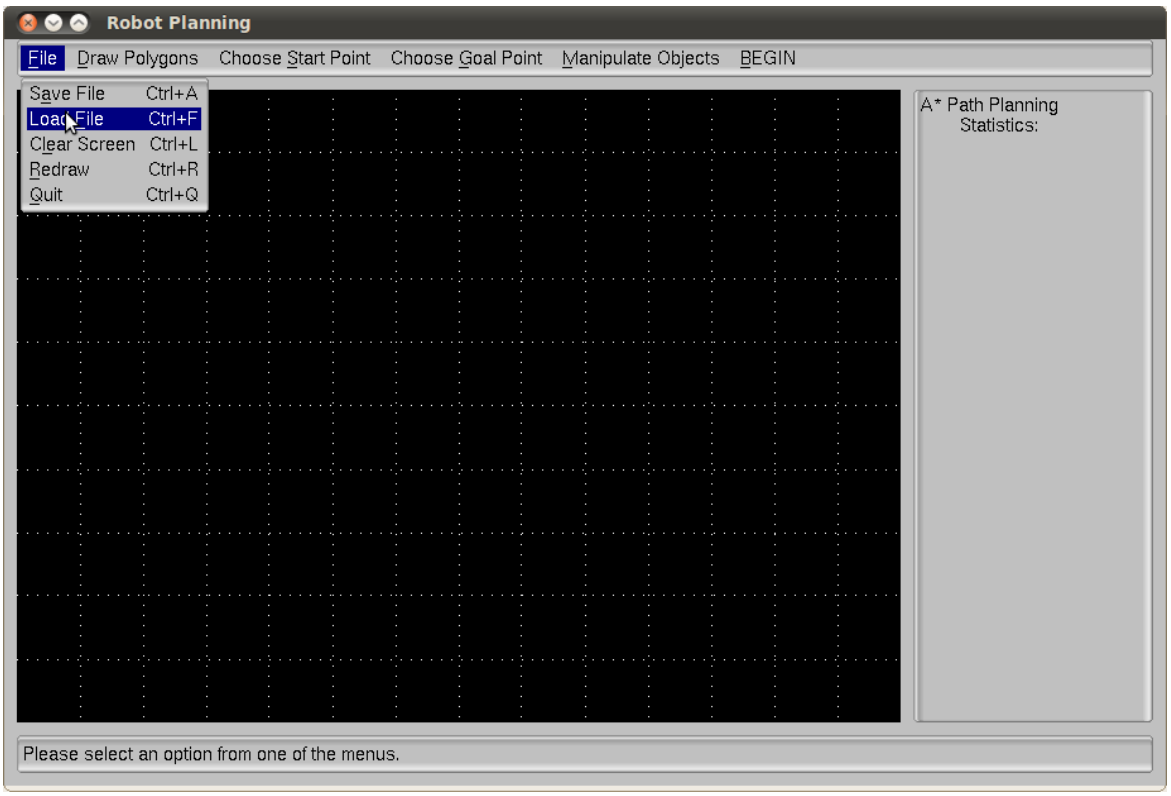


Figure 11

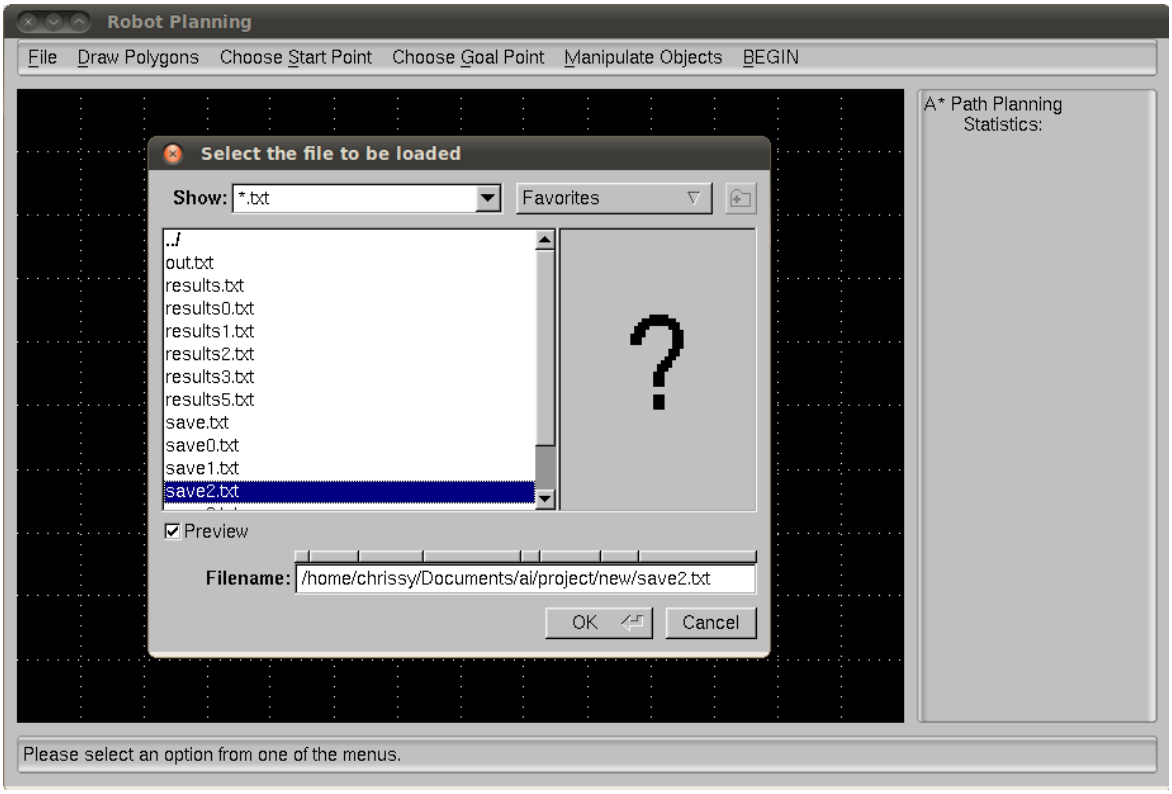


Figure 12

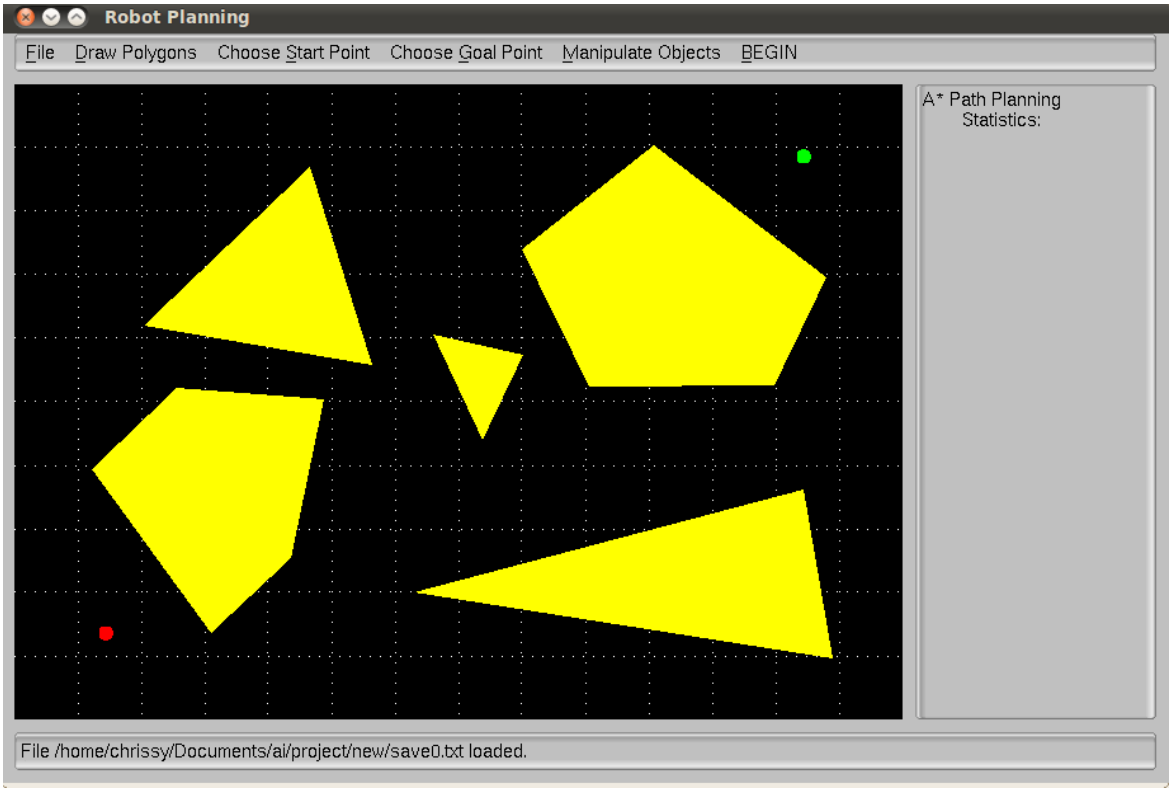


Figure 13

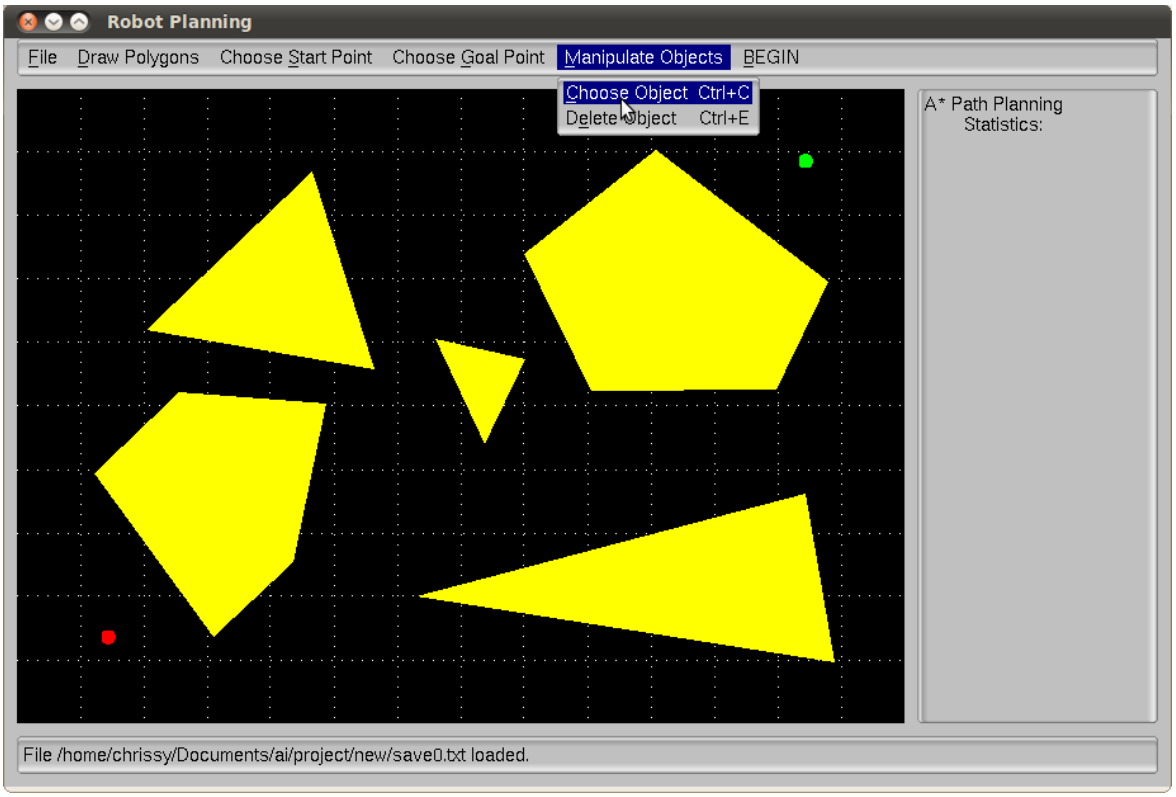


Figure 14

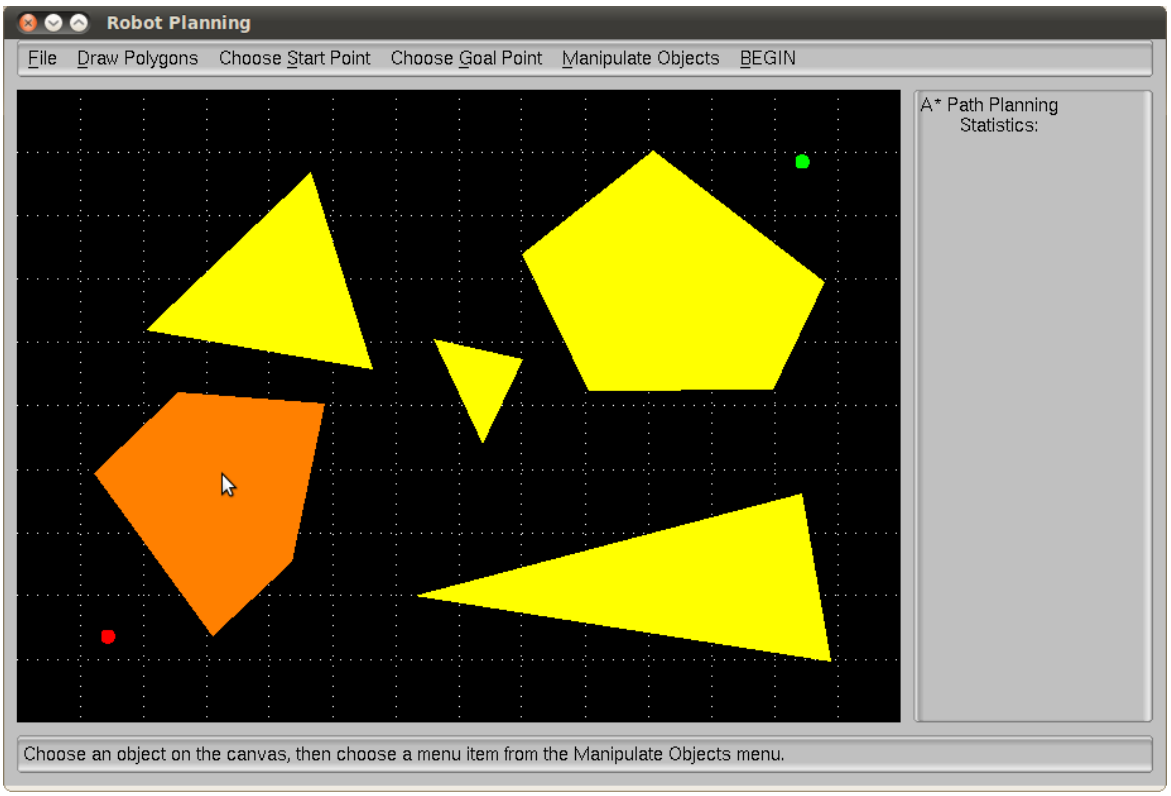


Figure 15

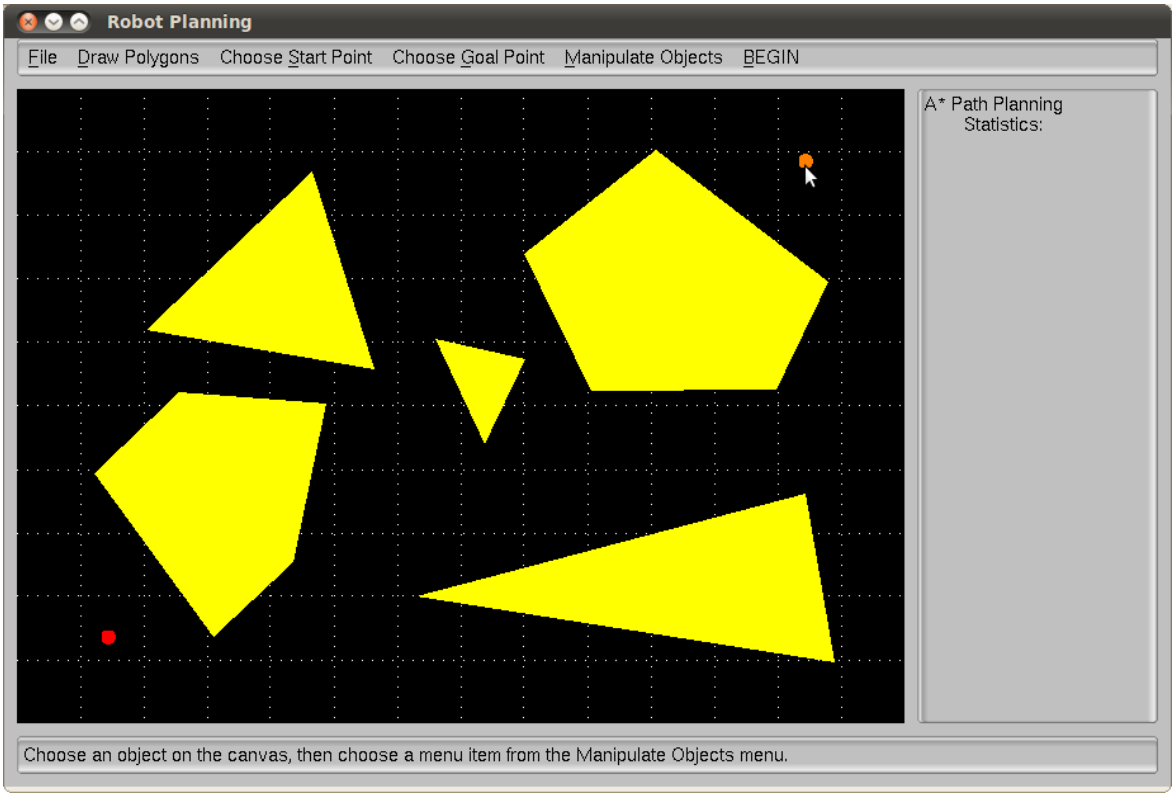


Figure 16

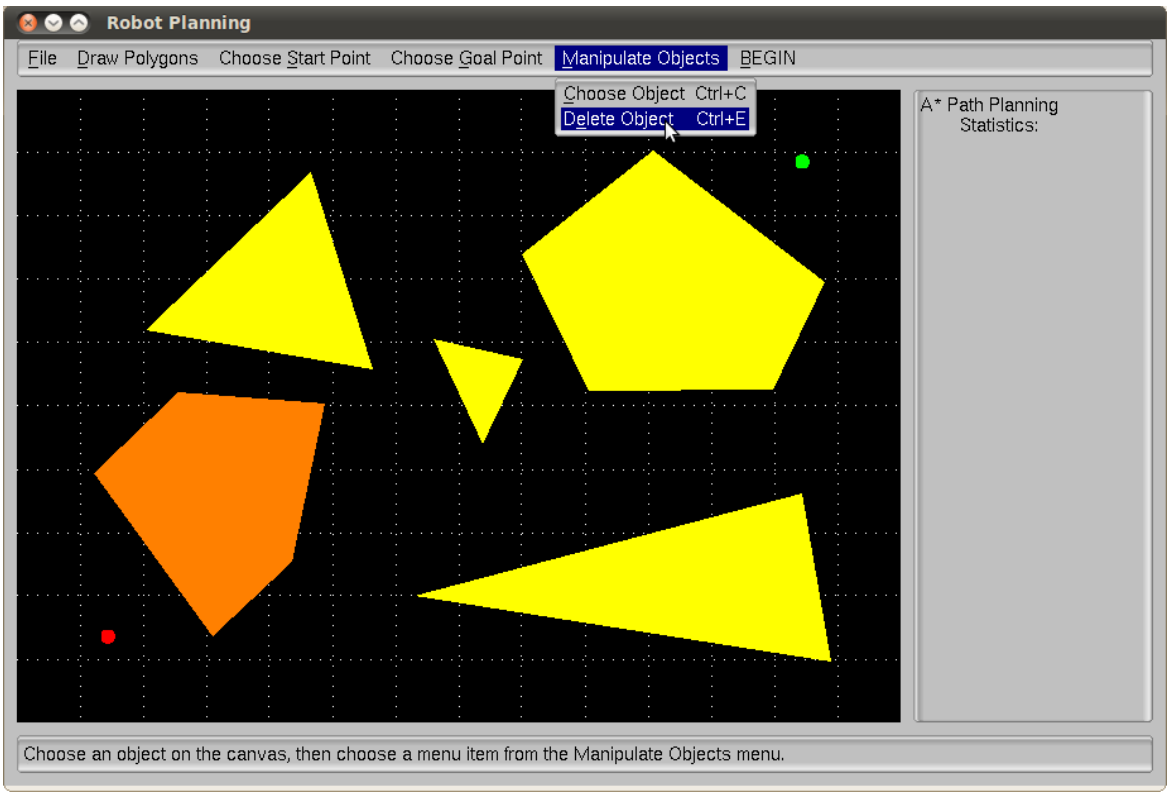
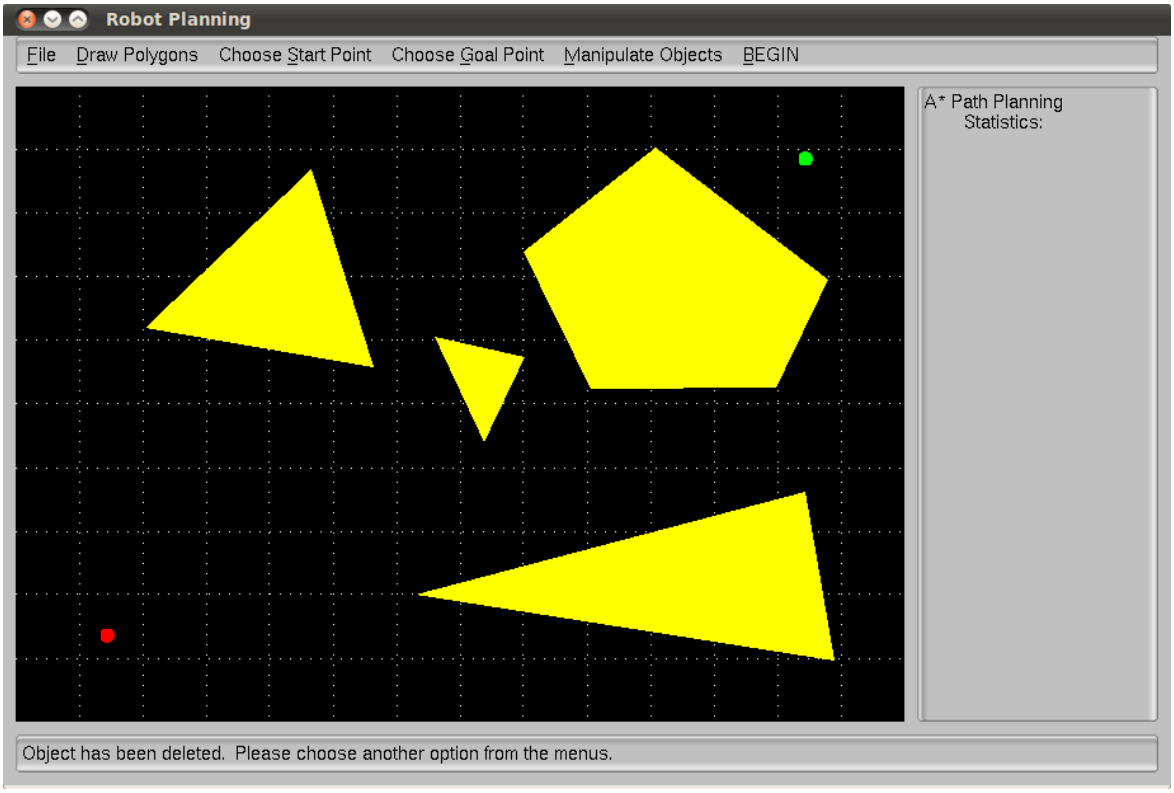


Figure 17



Once the environment is setup the way you want it, you can run the program to find the shortest path from the start point (red dot) to the goal point (green dot) within that environment. To do this, you click on the "BEGIN" menu item.

If you have not selected at least a start and goal point in the environment, you will get an error to the status bar which can be seen in Figure 18 and Figure 19.

Otherwise, this will convert the environment into a graph of all the vertexes and the start and end points which will be used for navigating through the environment. Then, the program will utilize the A* algorithm, starting at the start point and expanding each node in the graph to obtain its successors.

Only nodes that are not in the path already and not in the queue with a lower g(n) or current cost will be added to the queue. Successor nodes will only be expanded if they have the lowest estimated total cost for reaching the goal (g(n) + h(n)). G(n) is calculated by the linear distance between the node and the expanded node geographically within the environment. H(n) is calculated by the linear (direct) distance to the goal point. This assumes that there are no obstacles between the node and the goal, and therefore is always the same or less than the real cost to get to the goal.

Once a node is removed from the queue for expansion, it is checked for whether it is the goal. If it is, then the program will stop and save this node as the "answer" to the solution. By backtracking this solution's parents, we obtain the optimal path from the start point to the goal point, avoiding any obstacles in the environment.

This solution is then printed to the screen, along with the timing and total distance required for that solution. You are then prompted to save these results as a file, for future use (Figure 20). By default, this is "results.txt", but can be any filename you desire. The file will be saved in the same directory as where the program is being executed.

Once completed, you can close the application by clicking the X on the screen or by choosing "Quit" from the "File" menu. Also available on the "File" menu are the ability to clear the screen (to restart drawing of the environment from scratch), and the redraw options – as can be seen in Figure 21.

Figure 18

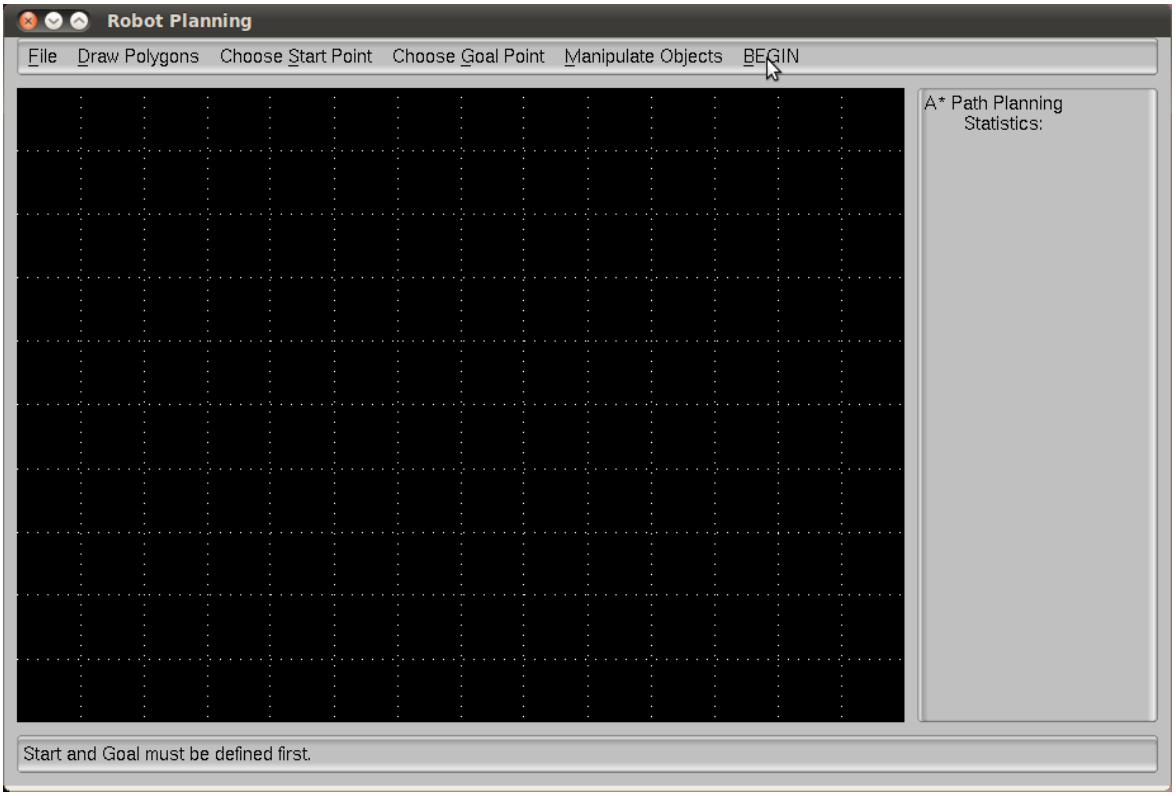


Figure 19

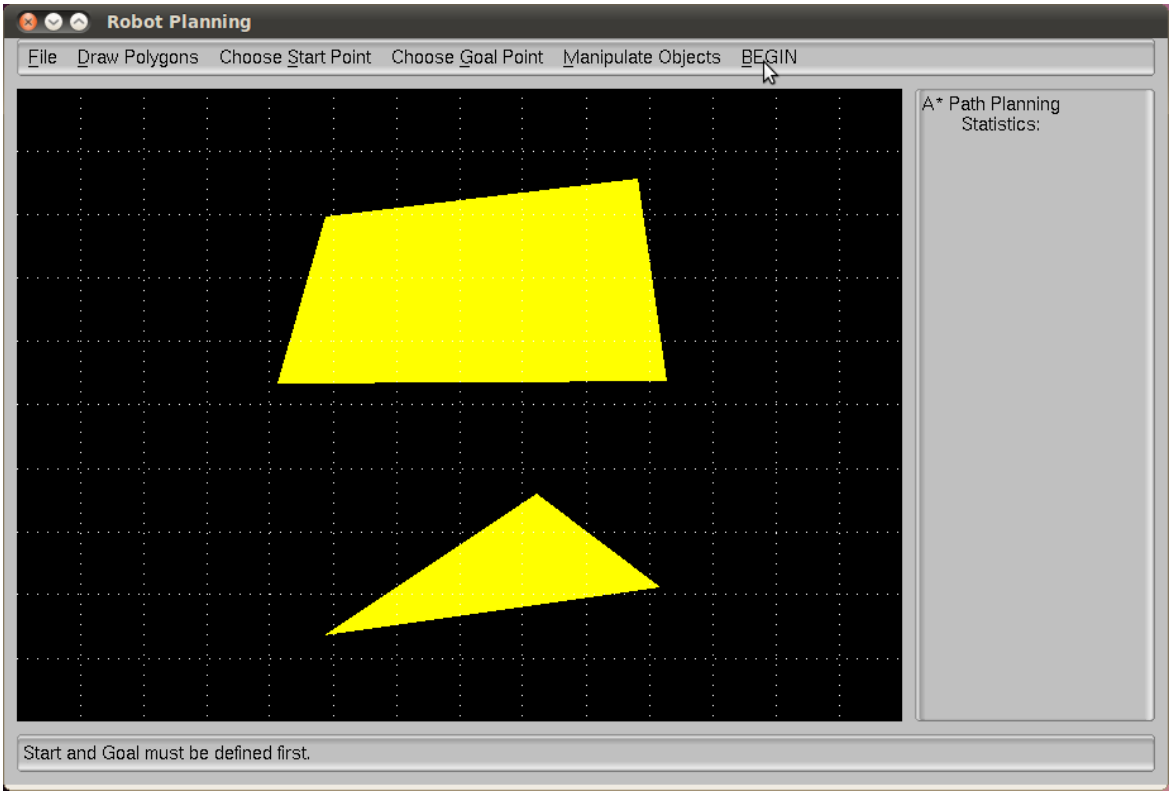


Figure 20

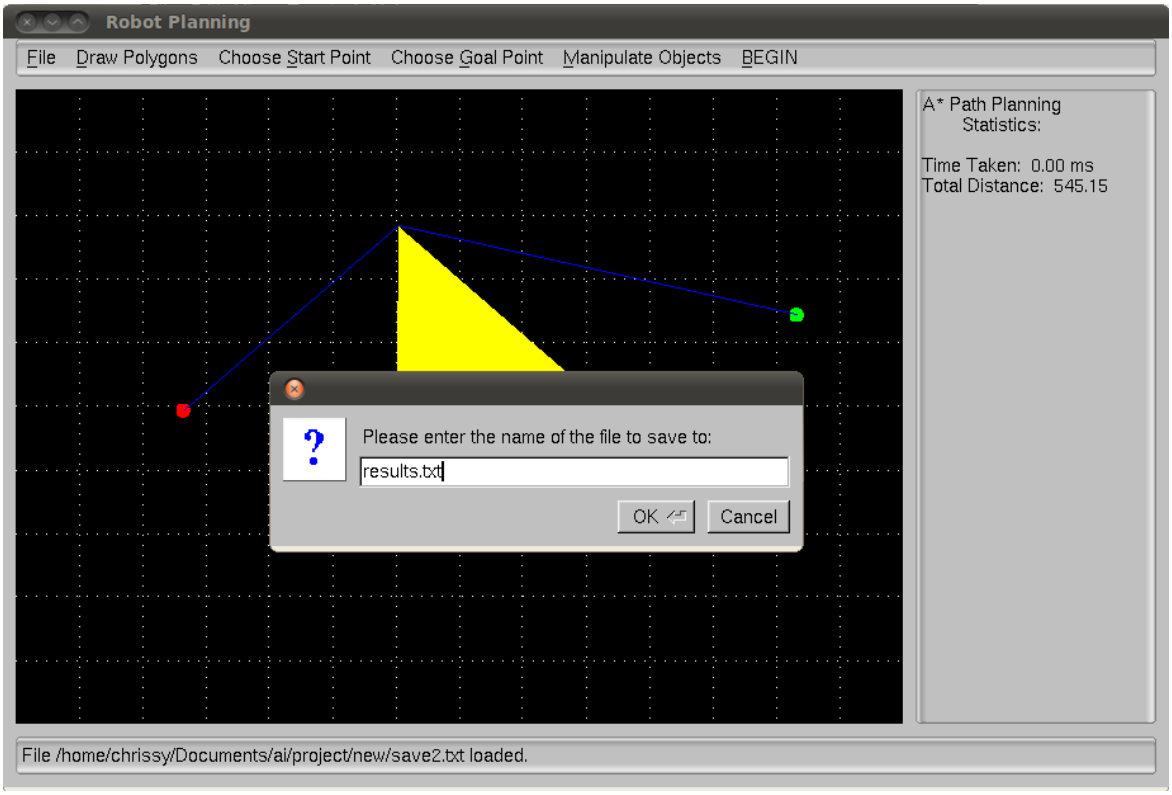
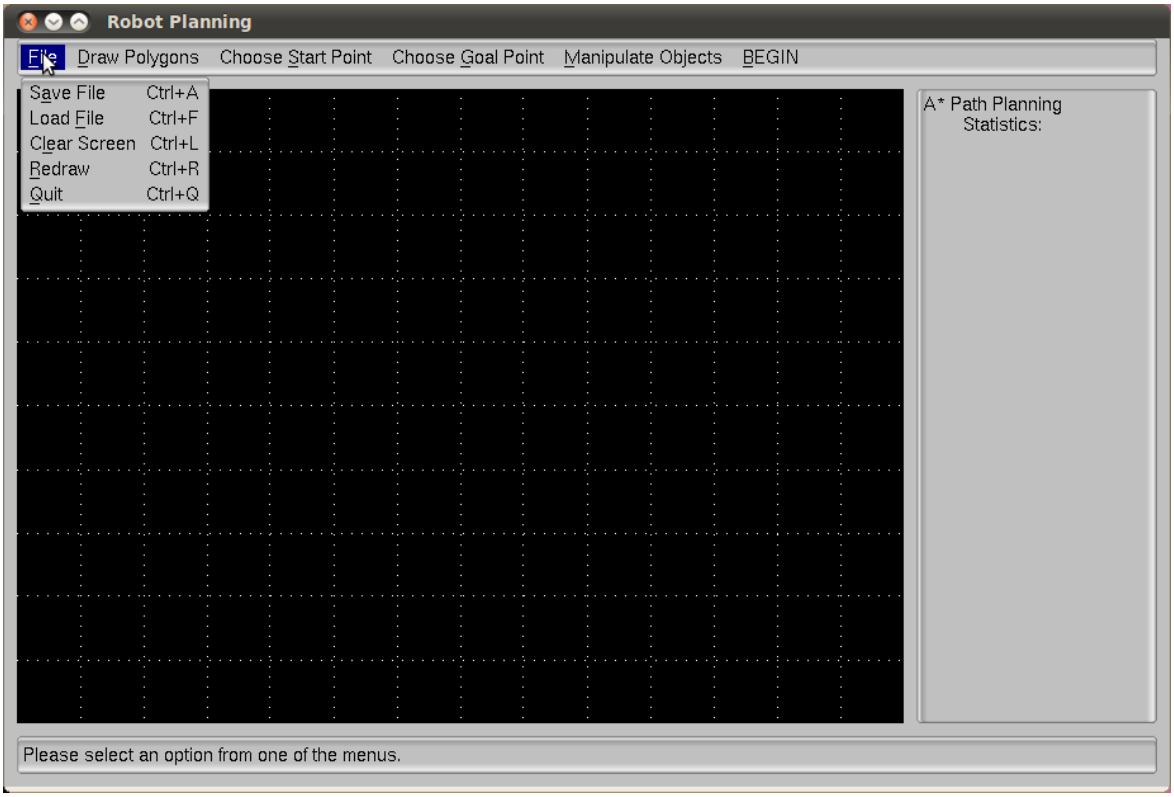


Figure 21



General Comments & Results

While running the program, I noticed that the more obstacles there were in the environment, the more complex the problem gets. Each vertex of the polygons ends up being a node in the graph and has to be compared to every other node in the graph to determine if it is within sight of the node or not. This becomes an $O(n^2)$ just to loop through all the possible in-sight points. It doesn't take into consideration all the calculations that are required to determine whether the point is in-sight of another which requires you to then compare those two points with every polygon in the environment to determine if there is an intersection. Therefore, this problem easily gets out-of-hand very quickly.

I found that generating the graph itself took more time than the A* algorithm did because of this issue. Overall, the A* algorithm was relatively efficient in finding solutions for any environment thrown at it. Some of the examples can be seen in Figures 22-31 below. They are arranged in pairs where the even numbered figure is the before shot and the odd numbered figure is the results shot for the previous environment.

It was also interesting that although the paths found were the optimal paths for hugging the obstacles, they weren't necessarily optimal for the environment. This is due to the fact that the robot can only visit the points of the obstacles and not any other "free" space in the environment. This can be seen in Figures 24 and 25 where the robot could've gotten a shorter path by navigating halfway between the two large triangles instead of travelling along one edge of the first triangle.

Figure 22

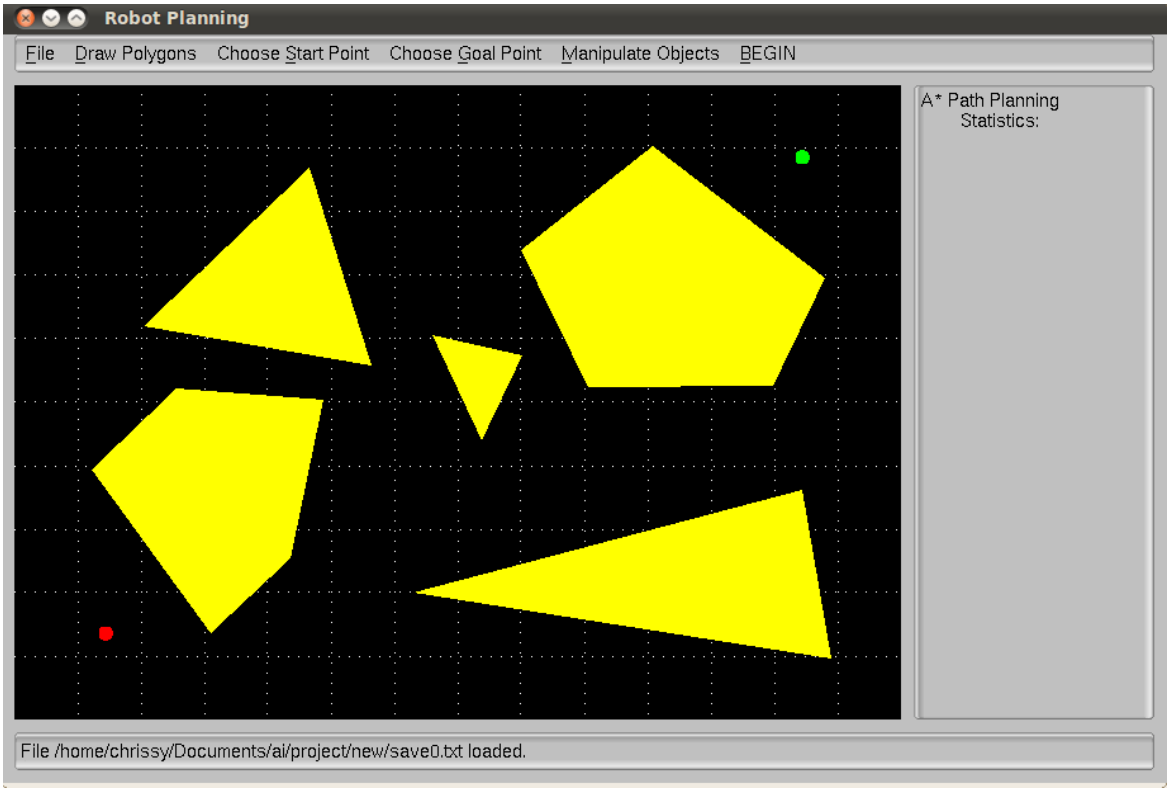


Figure 23

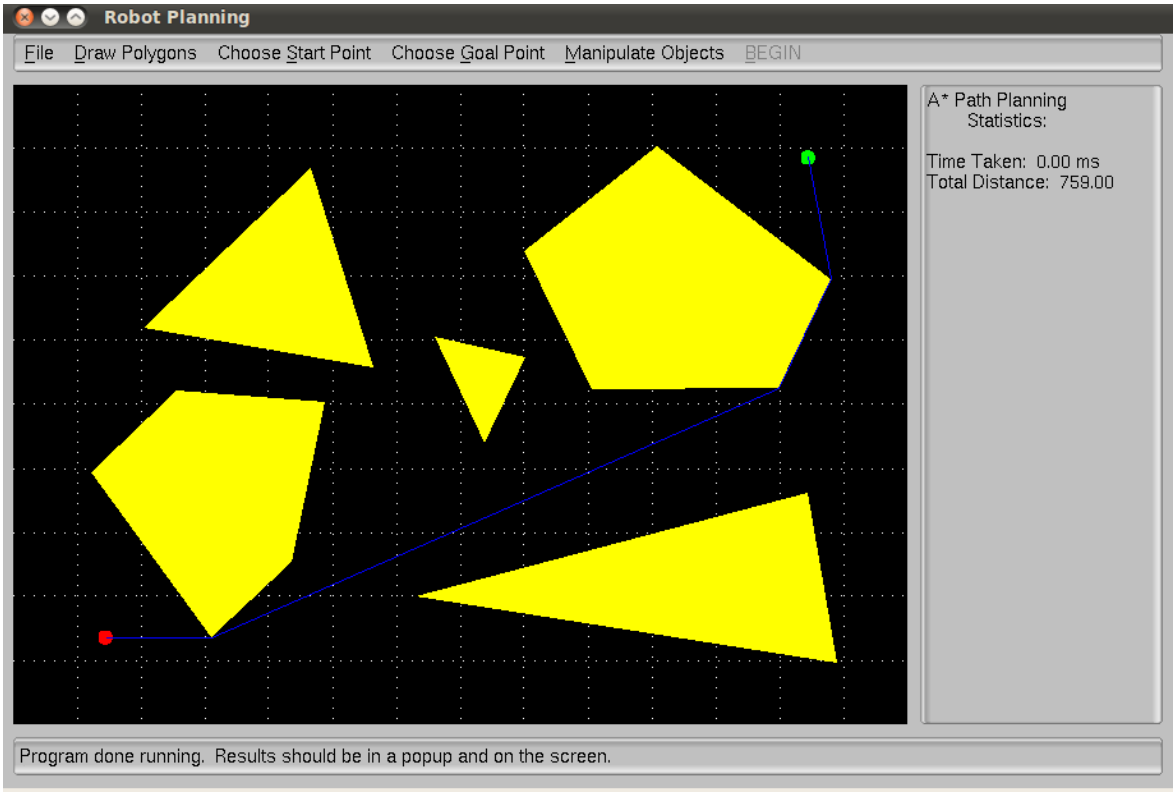


Figure 24

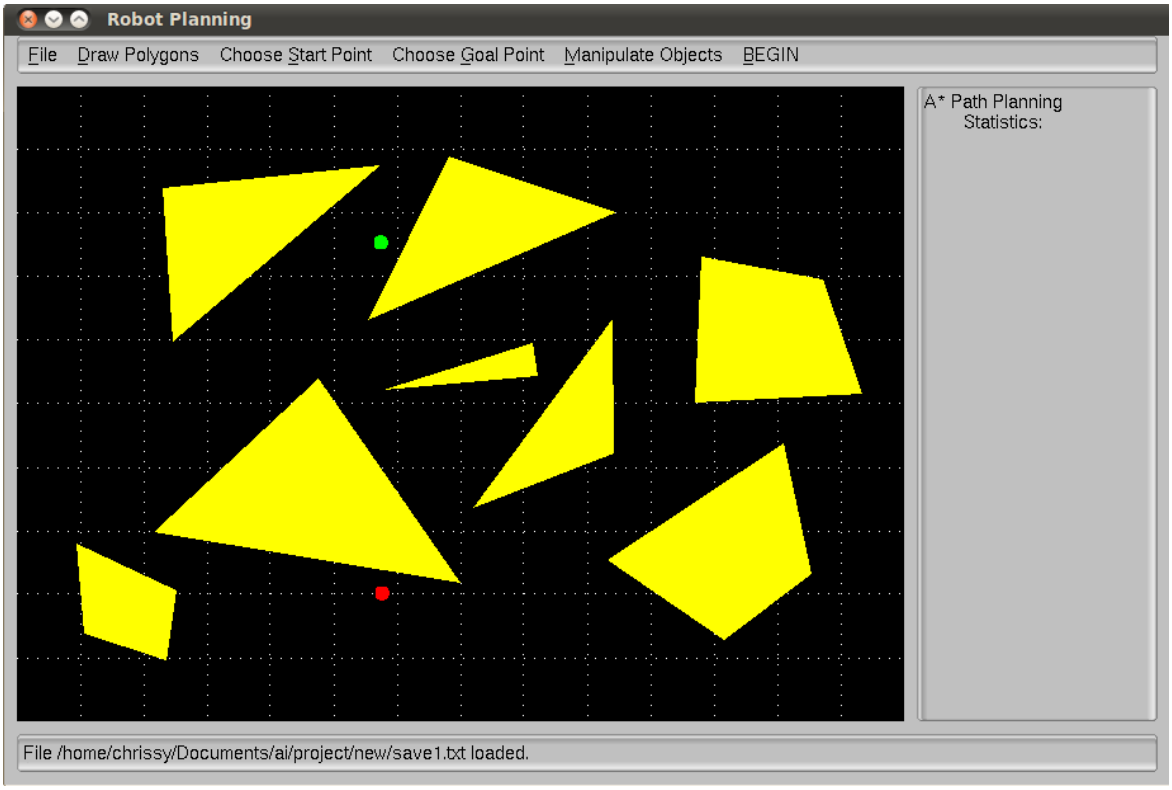


Figure 25

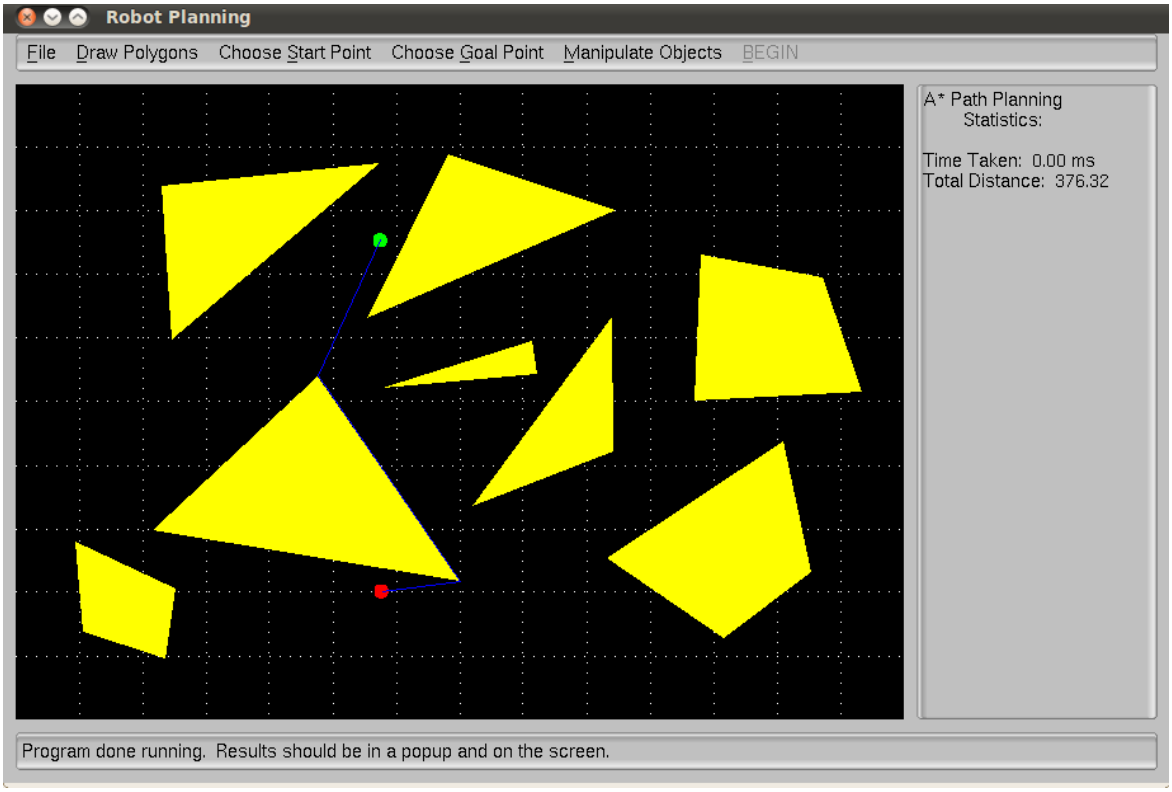


Figure 26

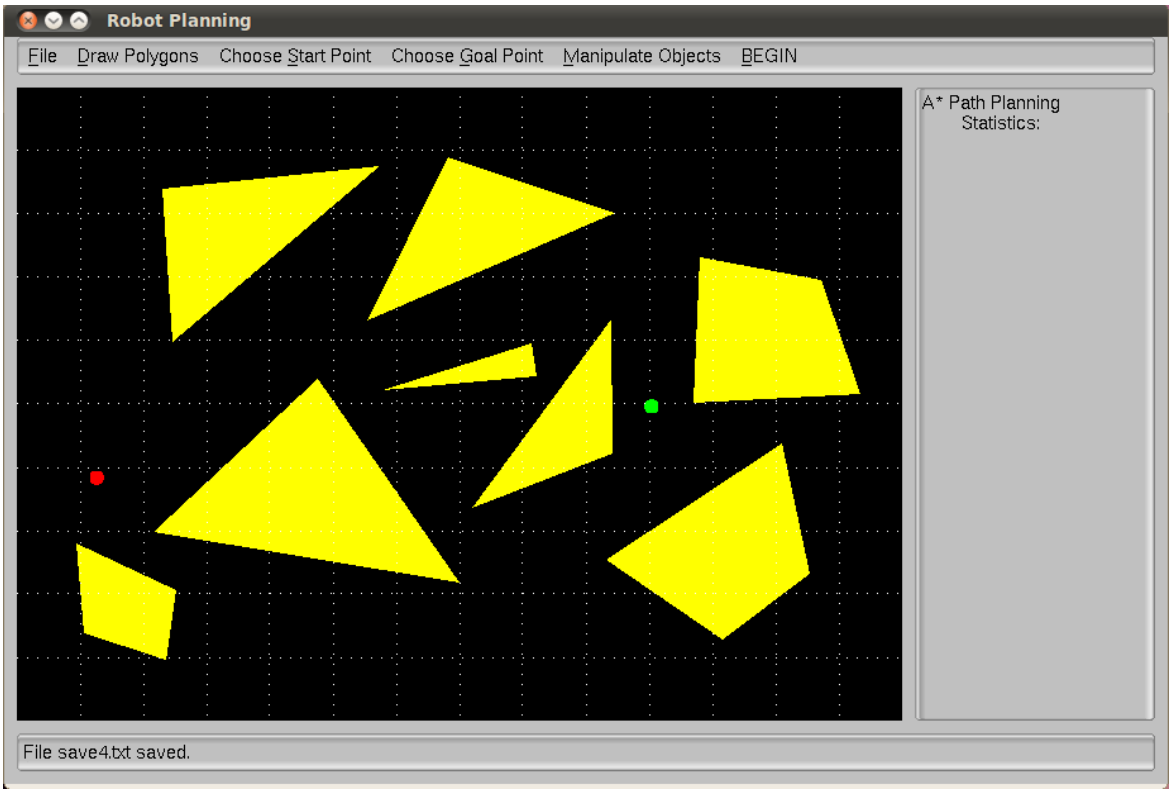


Figure 27

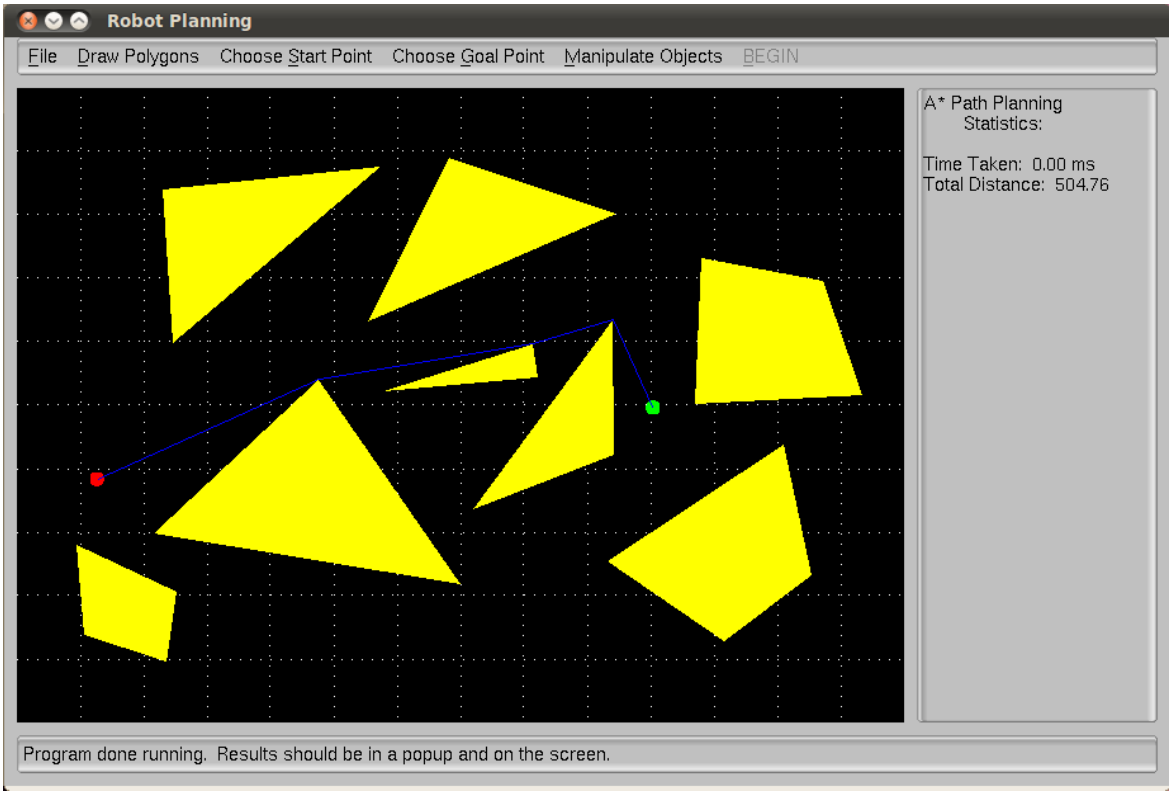


Figure 28

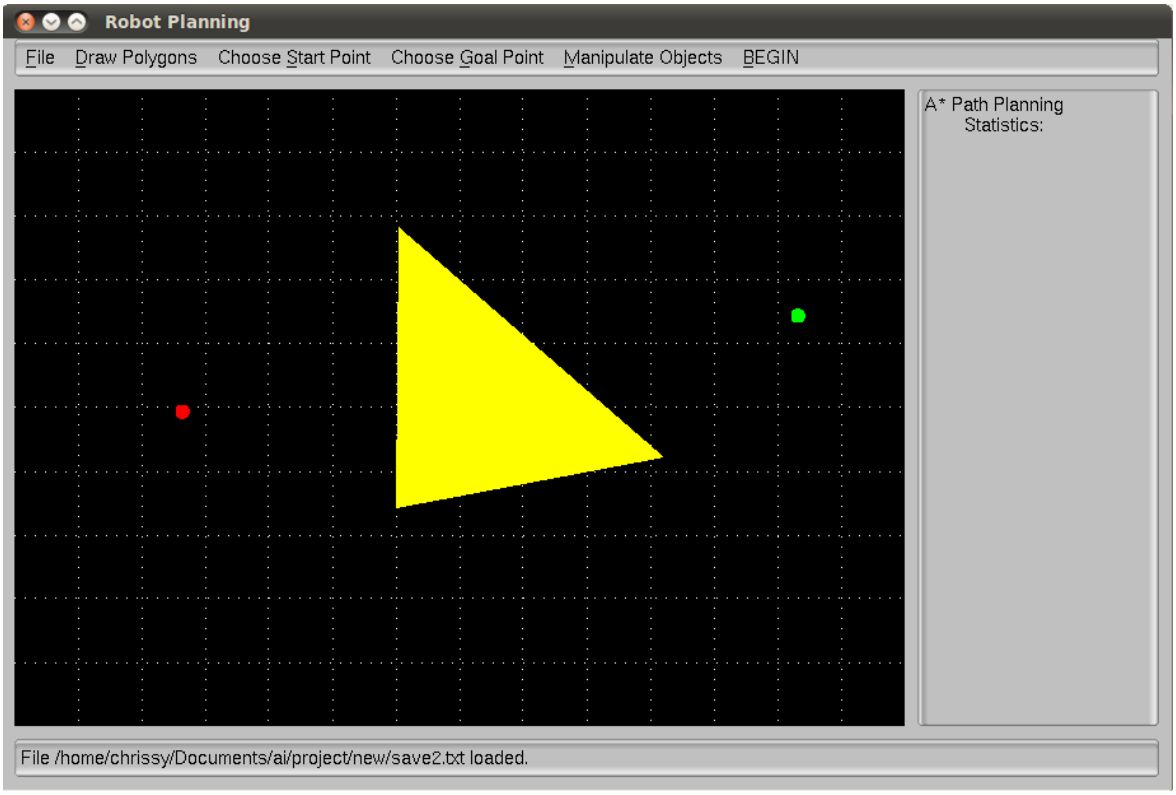


Figure 29

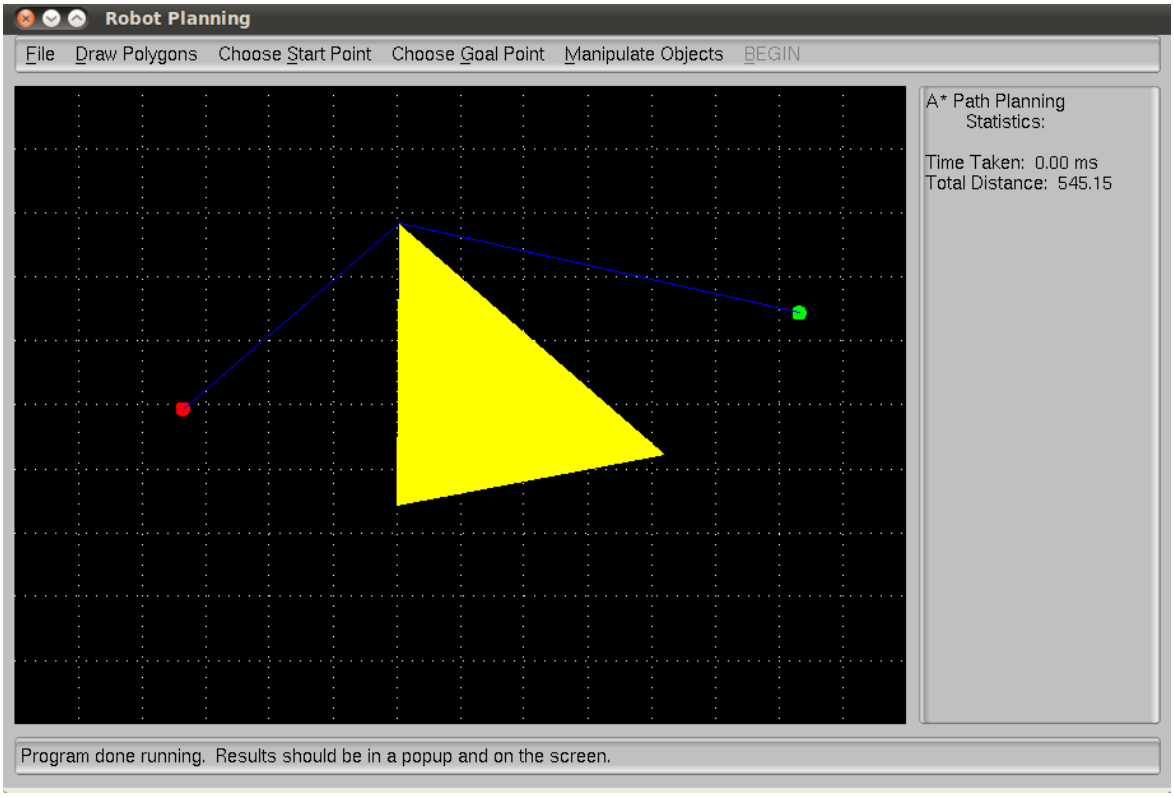


Figure 30

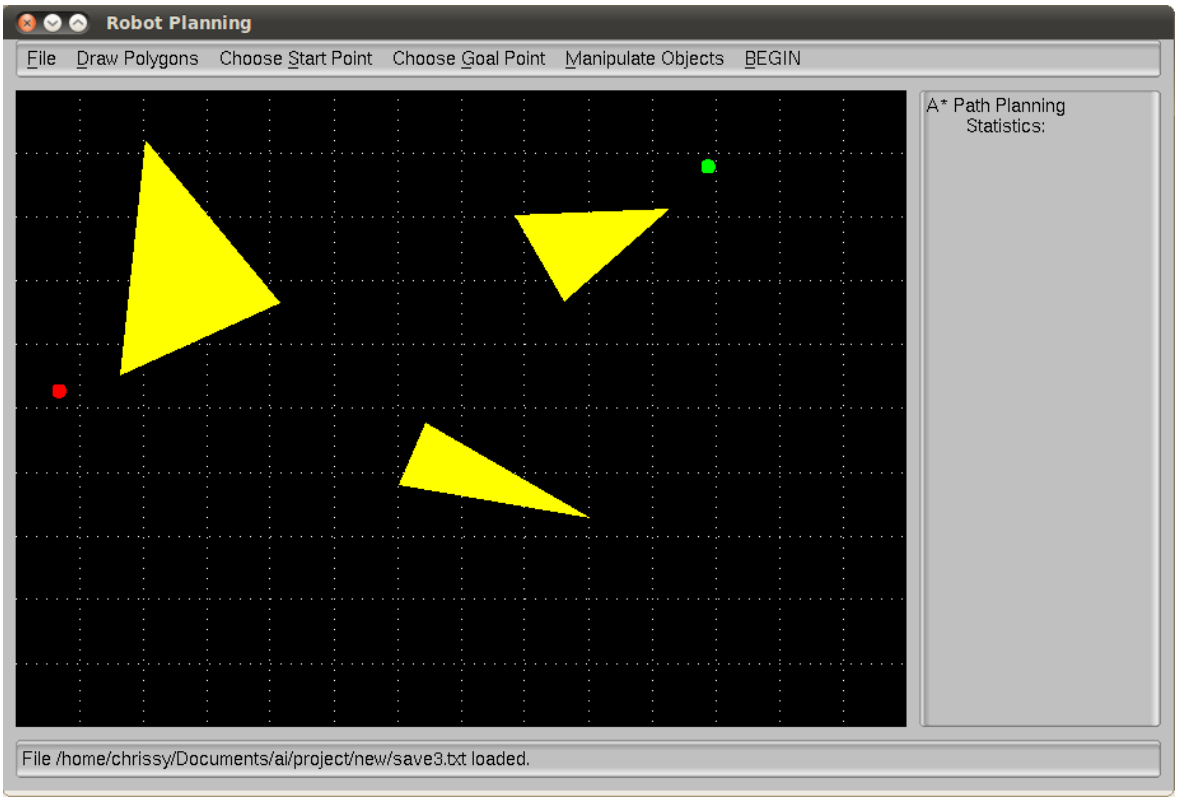
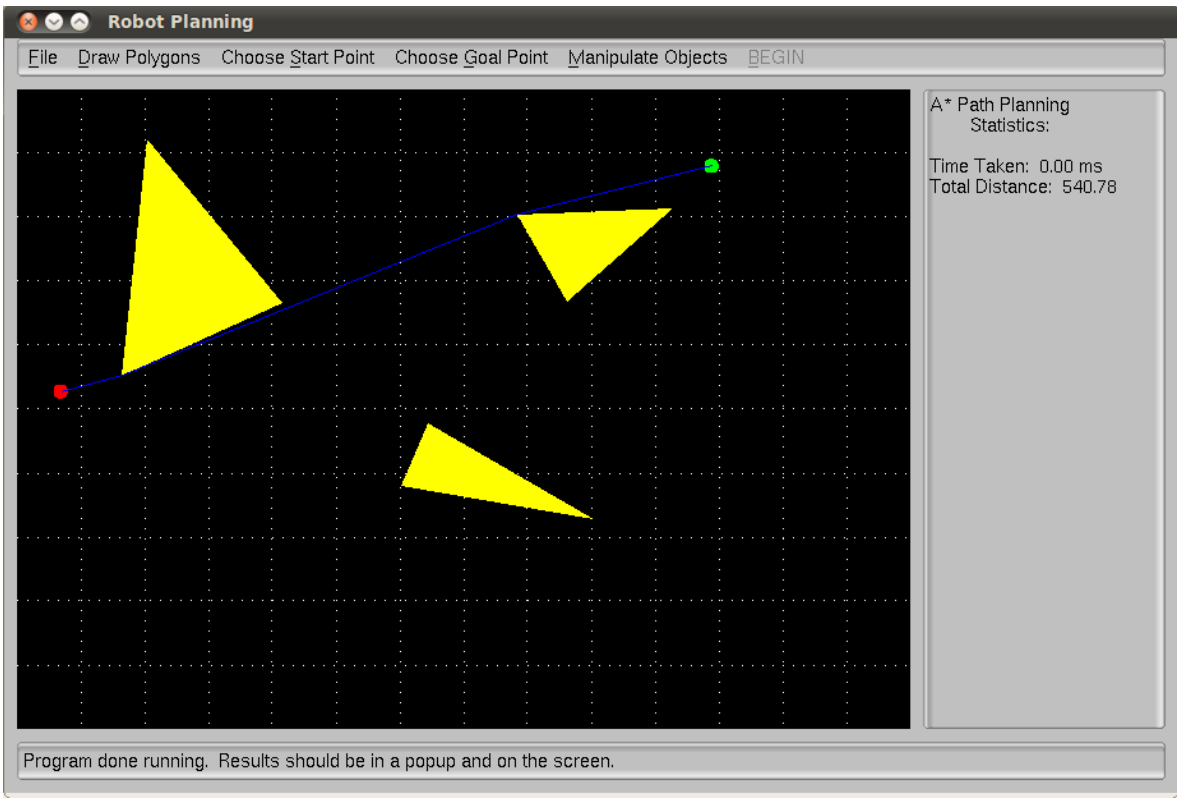


Figure 31


Robot Planning
File
Draw Polygons
Choose Start Point
Choose Goal Point
Manipulate Objects
BEGIN
A* Path Planning
Statistics:
Time Taken: 0.00 ms
Total Distance: 540.78
Program done running. Results should be in a popup and on the screen.

Future Extensions

I had started implementing the LRTA* algorithm utilizing this GUI in order to do a comparison of the two algorithms and an offline vs online solution. Unfortunately, due to time, those components were disabled in this project.

I had also hoped to expand it further to utilize the grid lines in the environment for a more ad-hoc navigation through the environment instead of only utilizing the vertexes of the obstacles. This would help alleviate the issue with navigating between obstacles requiring going from a point to another point only.

And finally, I would like to expand this to enable the robot to be more "blind" to the environment and learn how to find the goal without knowing where the goal is in the environment.

Outline of Work

For this project, I created a graphical interface which is highly interactive and has the ability to load and save different environments for reuse. I created from scratch my own graph, node, edge, state, and priority queue classes to support the A* algorithm. I also completed a fair amount of the LRTA* algorithm hooks, but had to disable them due to time constraints.

Documentation of the program, functions, etc can be found below:

Program Name:
robot

Description:
This includes the main function which drives the application

Method Summary:
int main (int argc, char **argv)
Creates the window, then handles events

Class Name(s):
MainWindow

Description:
Implementation of MainWindow class which provides the window, menu, and status bars

Method Summary:
void quit_cb(Fl_Widget*, void*)
Exits the application
void changeMode_cb(Fl_Widget*, int newmode)
Changes the mode of the application via the menus
MainWindow()
Constructor that setups the window, menu, and status bars
~MainWindow()
Destructor for the class
void show(int argc, char **argv)
Passes show command on to the window
void changeMode(int newmode)
Invokes the change mode method & updates the status bar
void setLabel(string newlabel)
Allows us to set the status bar for errors from the glWidget class

Class Name(s):
GLWidget

Description:
Implementation of GLWidget class which is inherited from the Fl_Gl_Window class for viewing & drawing in the application

Method Summary:

- GLWidget(int x, int y, int w, int h, const char *l):
- Fl_Gl_Window(x, y, w, h, l)
  - Constructor that creates a window
- void init()
  - Initialization method to setup the window & object coordinates and related clipwindows
- void setcolor (DrawColor col_val)
  - Changes the drawing color for objects
- float distance(float x0, float y0, float x1, float y1)
  - Calculates the distance between two points
- int handle(int event)
  - Handles mouse events in the application
- void setWindowExtents()
  - Sets up the window & object coordinate systems
- Point toNDC(Point p)
  - Converts the window coordinate system to the Normalized device coordinates from (-1,-1) to (1,1)
- void changeMode(DrawMode newmode)
  - Handles the menu choices within the application
- void draw()
  - Handles all drawing for the application
- void drawviewports(Polygon p, Polygon *win)
  - Is called by the draw() function and does all coordinate translations & drawing of objects within the clip window win into the corresponding viewport
- void init_clip()
  - Initializes and resets the clipping windows
- void drawpolys()
  - Is called by the draw() function and loops through all saved polygons & draws them
- void drawlines()
  - Is called by the draw() function and loops through all saved lines & draws them
- void drawwinobjs()
  - Is called by the draw() function and draws the two clip windows
- void drawviewports(Line l, Polygon *win)
  - Checks if any part of the line is in the clipwindow & if so, draws that piece of the line
- void drawviewports(Polygon p, Polygon *win)
  - Checks if any part of the polygon is in the clipwindow & if so, draws those pieces of the outline of the polygon
- Point getCenter(Line l)
  - Calculates the midpoint of the Line & returns that point
- Point getCenter(Polygon p)
  - Calculates the center of the Polygon & returns that point
- Line lineClip(Point winMin, Point winMax, Point p1, Point p2)
  - Returns the clipped part of the line that resides within the clip window & colors the line ORANGE (for reference only) if any part of the line is within the clip window
- int cliptest(float p, float q, float * u1, float * u2)
  - Returns the clipped part of the line that resides within the clip

```
Class Name(s):
      AI

Description:
      Implementation of AI algorithms and supporting methods such as A*

Method Summary:
      AI();
            dummy constructor
      ~AI();
            dummy destructor
      void createGraph();
            reads all the polygons, start point, and end point and creates a
            graph connecting all points that are within sight of each other
      void init(GLWidget::Point a, GLWidget::Point b,
      vector<GLWidget::Polygon> c);
            initializes the globals for this class, such as the graph & other
            vectors
      void runAstar(GLWidget::Point a, GLWidget::Point b,
      vector<GLWidget::Polygon> c);
            runs the A* algorithm to find the shortest path using the graph
            that was generated
      void runLRTAstar(GLWidget::Point a, GLWidget::Point b,
      vector<GLWidget::Polygon> c);
            not used currently
      void converttolines();
            takes every point of the polygons & start/goal points and
            connects them
            to determine which points are within direct site of each other
      bool checkintersections(GLWidget::Line a);
            checks to see if this line between two random points in the
            environment
            are intersected by another polygon
      float min(float a, float b);
            returns whichever one is less
      float max(float a, float b);
            returns whichever one is greater
      bool intersect(GLWidget::Point p1, GLWidget::Point p2, GLWidget::Point
      p3, GLWidget::Point p4);
            determines if the lines from p1 to p2 and p3 to p4 intersect each
            other or not
      int Astar();
            A* algorithm that uses the graph to find the shortest path from
            the start to the goal point
      int LRTAstar();
            not used currently
      void generateSuccessors(int s);
            uses the generated graph to determine points that are within site
            of the current point
      void checkfordupes(int anc, int newpt );
            checks to see if any of the successors are already in the queue
            or in the ancestors
      float diffclock(clock_t clock1,clock_t clock2)
            calculates duration times in milliseconds
```

```
Class Name(s):
      State

Description:
      Implementation of State class which provides the point, current
      cost, and estimated future cost for expanded & generated nodes
      during A* algorithm

Method Summary:
      State(float x, float y, int p, int c);
            constructor
      State();
            constructor
      ~State();
            destructor
      float getfn();
            calculates the sum of g(n) + h(n) for total estimated cost
      float getgn();
            returns the current cost to get here
      float gethn();
            returns the estimated cost to get from here to the goal
      string toString();
            prints a formatted version of this state
      bool isgoal();
            returns whether this state is the same as the goal state or not
      State& operator=(const State& s);
            overloads the = operator to copy the state
      float getx();
            gets the x coordinate
      float gety();
            gets the y coordinate
      int getparent();
            gets the parent for this node (which point we came from)
      Node getn();
            returns the node / point for this state
      bool operator==(const State &s1) const;
            overloads the == operator to allow comparisons for equality
      string printparents();
            for debugging - prints the parent hierarchy for this state
```

```
Class Name(s):
      PQ

Description:
      Implementation of Priority Queue class which provides the min
      extract as well as deletions

Method Summary:
      PQ();
            constructor
      ~PQ();
            destructor
      void push(int s);
            adds s to the priority queue in the correct priority order
      int peek();
            lets you see which item is next in the queue without removing it
      void erase(int s, float val);
            removes the item s if it has a greater value than val
      void erase(int s);
            removes the item s
      void clear();
            cleans up the priority queue
      int size();
            returns how big the priority queue is
      string toString();
            formats the priority queue for printing
```

```
Class Name(s):
      Graph

Description:
      Implementation of Graph class which provides the graph to help
      identify successors for the A* algorithm

Method Summary:
      Graph();
            constructor
      virtual ~Graph();
            destructor
      void addNode(float x, float y);
            adds a new node to the existing graph
      void addEdge(float x1, float y1, float x2, float y2);
            finds the starting and ending nodes & adds an edge to both to
            represent they are successors of each other
      Node getMinSuccessor(float x, float y);
            finds the min f(n) for all successors of the node at this point
      NodeEdges getNode(float x, float y);
            returns the node with this point
      string toString();
            prints the graph in a pretty format
```

```
Class Name(s):
      Node

Description:
      Implementation of Node class which provides the "points" in the graph

Method Summary:
      Node();
            constructor
      Node(float x, float y);
            constructor
      Node(const Node &n);
            constructor
      void setx(float a);
            sets the x coordinate to a
      void sety(float a);
            sets the y coordinate to a
      virtual ~Node();
            destructor
      string toString();
            formats the node for printing
      bool operator==(const Node &n1) const;
            overloads the == operator for comparing two nodes
      Node & operator=(const Node &n);
            overloads the = operator for copying / assigning a node to this
            one
      float getx();
            returns the x coordinate
      float gety();
            returns the y coordinate
```

```
Class Name(s):
      Edge

Description:
      Implementation of Edge class which provides connections between
      the points in the graph which are in-sight of each other

Method Summary:
      Edge();
            default constructor - not used
      Edge(float x1, float y1, float x2, float y2);
            constructor that connects the point at (x1,y1) to (x2,y2)
      virtual ~Edge();
            destructor
      float getweight();
            returns the distance from the point (x1,y1) to (x2,y2) for this
            edge
      Node getstart();
            returns the starting node for the edge (x1,y1)
      Node getend();
            returns the endingn node for the edge (x2,y2)
      Edge& operator=(const Edge &e);
            overloads the = operator to allow assigning of Edges to new Edges
      string toString();
            for debugging - prints out the edge information
```