UNCC\_ASTR1 Spring 2017

Design: Navigation – Pathfinding Prototype

William Karnavas

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Staff** | **Notes** |
| Original | January 16, 2017 | WK | First draft |
| Rev A | January 27, 2017 | WK | More implementation details |
| Rev B | February 9, 2017 | WK | Continued adding implementation details |
| Rev C | Feb 16, 2017 | WK | More details for controls |
| Rev D | Mar 16, 2017 | WK | Details of algorithm improvements |

# Summary:

Create an app to allow fast, broad tweaking and tuning of robot pathfinding algorithms before implementation in LabVIEW and slower, physical testing. Visualizations and quick tuning will be emphasized, as well as swift development.

Final deliverables will include a set of initial parameters to tune our LabVIEW pathfinding and movement modelling code with, and diagrams and animations illustrating robot navigation.

Contents

[Summary: 1](#_Toc477441140)

[Critical Requirements: 2](#_Toc477441141)

[Project Steps: 2](#_Toc477441142)

[Getting Results: 2](#_Toc477441143)

[Initial Requirements: 3](#_Toc477441144)

[Notes: 4](#_Toc477441145)

[Notes on Movement Modelling: 4](#_Toc477441146)

[Tool Selections 5](#_Toc477441147)

[Communication Between Components 5](#_Toc477441148)

[Control Components – Initial List 6](#_Toc477441149)

# Critical Requirements:

* Build Time: <= 2 weeks
* Audience: Just me, WK.
* Produce illustrations suitable for posters & documentation
  + Animations?
* Allow quick, gui- or console-based tweaking of path algorithm & parameters at runtime
* Simulate:
  + Pathfinding algorithm
  + Movement modelling
  + Movement error, maybe, if needed

# Project Steps:

1. Choose tools (Lua + LOVE for visuals, over sockets with Java or VEE for controls, as judged swiftest)
   1. Learn tools
      1. LOVE tutorial
      2. Lua or LOVE networking
2. Create detailed initial requirements
   1. Controls & Parameters needed
   2. Visualizations wanted
3. Start coding project (all simultaneously)
   1. Interface, controls, and visualizations
   2. Pathfinding algorithm
   3. Movement modelling
4. Iterate project code

# Getting Results:

1. Iterate sets of parameters
2. Decide on factors to consider when choosing parameters
3. Continue iterating parameters, optimizing to chosen parameters
   1. Try scenarios
      1. From trough to mining area
      2. From mining areas to trough
      3. Lateral movement (as when correcting moving and missing a target point)
      4. Spinning (as when correcting bad alignment when arriving at trough)
4. With near-final parameters, create visualizations
   1. Low res for documents
   2. High res for poster and slides

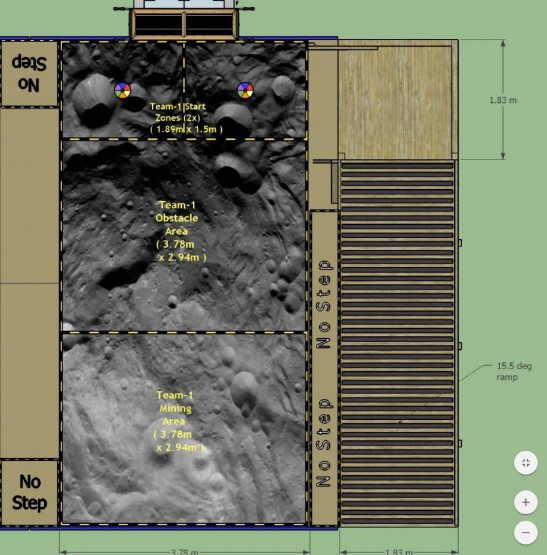
# Initial Requirements:

Below is a list of capabilities MAP-Tune could include.

* Controls & Parameters
  + Situation
    - Start Position, Angle
    - End Position, Angle
      * Pos & Angle Tolerances
    - Whether to choose best mining location as a destination
    - Previously mined locations
  + Environment
    - Path position density (x, y)
    - Dead zone at margin(s?) of arena
  + Path Scoring
    - Weighting constants for all factors, including
      * Straightness per segment
      * Length per segment
      * # of Movements
      * Destination desirability
  + Movement Modelling (if using iterative solution)
    - Movement per iteration (i.e. resolution)
  + Pathfinding
    - Max # of Movements
  + Mining Destination Selection
    - Score gradient (start, end values)
* Readouts & Stats
  + % of paths that reach destination
  + # of paths considered
  + Top N paths:
    - Steps, should show on a map w/modelled curves
    - Scores, total and for each step
* Algorithms
* Visualizations
  + [All arena visualizations feature…]
    - Faint points
      * & Dead zone
    - Already mined areas
    - Start location marker?
  + Arena Visualizations (drawn on a picture of the arena):
    - Points on arena
      * Numbers & Labels
      * Mining destination score labels
    - Movement model
      * Possible end orientations & “can reach” for each point in arena (sea of arrows)
    - “Track” of a single or multiple paths across the arena
      * Red lines for motion
      * Animated movement?
    - Best destinations heatmap
      * In mining area
    - Some animation of the pathfinding process
  + Path tree visualization
    - Showing “shape” of destination tree, color coding on nodes for “valid destination”
      * Hopefully patterns show up, and we can design a heuristic
  + Save screenshots of visualizations
  + ------------------------------------LIST INCOMPLETE----------------------------------

# Notes:

Arena Size: 3.78m x 7.38m



# Notes on Movement Modelling:

Movement modelling will be performed using circular curves, to which an iterative solution was found.

The state of the curve solution is as follows:

* Given information: initial heading, initial position, end position, radius of curve.
* Expected result: whether curve is possible without exceeding curvature limit, end heading.

After informal evaluation of solutions, there appears to be no single possible curve with these characteristics.

An iterative solution involves inching the “robot” forward from its initial position, changing its heading by a constant maximum amount towards the destination position, as needed. Iteration continues until the “robot” is within tolerance of the destination, or the Euclidian distance of the robot away from the destination begins to increase again (passing destination) or remain the exact same (circling destination point).

# Tool Selections

Visualizations:

* Lua + LOVE (game library)
* Create a generic arena renderer, then specific functions to render different information on top.
* All visualizations just take set of pathfinding results and display them.
* Additionally, a socket server will communicate with another program displaying widgets for, e.g., choosing which of a list of paths are rendered to the screen.

Pathfinding:

* Same process as visualizations, and also in Lua
* Outputs a set of resulting paths as an object to Visualizer

Pathfinding Controls:

* Agilent VEE
* Communicates over socket to Pathfinding

Visualizer Controls:

* Java – Swing (for widgets)
  + Needed for nice lists, like the big list of paths to select between
* Communicates over separate socket to Visualizer

# Communication Between Components

* Pathfinding Controls SEND pathfinding parameters VIA socket TO Pathfinding
* Pathfinding SENDS pathfinding results VIA object reference (same program) TO Visualizer
* Visualizer SENDS list of paths (or other gui information) VIA socket TO Visualizer Controls
* Visualizer SENDS clicks on the map TO Pathfinding Controls
* Visualizer Controls SENDS live control prompts VIA socket back TO visualizer

# Control Components – Initial List

* Controls & Parameters
  + Situation
    - Mode: Destination point OR best mining destination
      * SWITCH
  + Environment
    - Path intermediate positions
      * Dead zone/margin width
        + SLIDER
      * Distance between positions
        + SLIDER
  + Path Scoring
    - Weighting constants for all factors, including
      * SLIDERS:
      * Straightness cost per segment
      * Length cost per segment
      * # of Movements
      * //Destination desirability
  + Movement Modelling
    - SLIDER
    - Segment Length
    - Turn Radius
    - Destination Tolerance
  + Pathfinding
    - SPINNER
    - Max # of Movements
  + Mining Destination Selection
    - Score gradient (start, end values)
* Stuff to Add To Visualizer
  + Previously mined locations
  + Best mining location mode
* Readouts & Stats
  + % of paths that reach destination
  + # of paths considered
  + Top N paths:
    - Steps, should show on a map w/modelled curves
    - Scores, total and for each step

# Details of Improvements to Pathfinding Algorithm

After some time observing the behavior of the pathfinding algorithm, the following needs were identified:

* Paths “shuffled” in place to turn mid-way
* Performance for “simple” two+ arc paths was terrible, worse than difficult paths
* Sometimes, no path could be found out of starting area

In response, the following algorithm improvements were implemented:

* A penalty for short back-and-forth arcs
* Three regions of the arena, each with a different intermediate position density
* Desperation, or adaptive intermediate position densities
* Minor: cache an atan calculation used in calculating robot corners, for performance

The penalty for back and forth arcs will add a fixed amount to the cost of an arc if it both is a reversal in direction, and is shorter than the length of the robot or a meter, whatever is found most effective in experimentations.

The three regions of the arena will be represented as two x-coordinates, where the rectangle where intermediate positions are placed will be trisected, and a different resolution, or distance between each point, will be used. Additionally, desperation will change this resolution as well depending on pathfinding results. Desperation will start with a much less dense grid of locations, particularly in the middle of the arena, move to the “standard” density that has been used up until now, and finally, if no paths can still be found, attempt a search with an even denser grid than has been used up to now. This improvement may speed up pathfinding, particularly for simple cases, while also making the chance of not finding a path at all much lower than currently.

Finally, an atan calculation in the prototype, which is used to calculate the positions of each of the corners of the robot, will be cached at the start of the program as opposed to recalculated on the inner loop of the movement simulator, which itself is in the inner loop of the pathfinding algorithm. Performance improvement should resemble that of removing an atan previously from the inner loop of the movement simulator itself.

TO DO: IMPLEMENT HOLE COLLISION DETECTION