

# Using RRTs to Navigate Gibson Environments

Paul McKerley  
George Mason University  
pmckerle@masonlive.gmu.edu

## Abstract

*Produce efficient maps to navigate two-dimensional floorplans extracted from three-dimensional Gibson Environment buildings.*

## 1. Introduction

The goal of this project was to produce road maps for Gibson Environment buildings that would allow robots to navigate from one arbitrary point in the map to another. This navigation should find reasonable straight-line paths. My approach was to use Rapidly-Exploring Random Trees, which provided maps in a matter of seconds. A\* search can traverse these maps, and the results paths refined to be straighter. Many samples paths can be found and refined, with the resulting paths added to the map. These refined paths give highly efficient paths between rooms.

## 2. Approach

There are several steps requirement to produce a roadmap of a Gibson Environment structure.

### 2.1. Extract Two-Dimensional Floorplan

The Gibson buildings come as three-dimensional graphs. To extract two-dimensional floor plans I used a Python module called meshcut.py (kindly provided by Yimeng Li.) Figure 1 shows a 3D representation of the Allensville apartment. When a 2D images is extracted at a height of 0.5 meters the result is found in 2. This picture is an image saved from the Python module matplotlib. But it can be converted to an OpenCV-compatible numpy array and manipulated in memory or saved to file.

### 2.2. Preparing Floorplan for Geometry Checks

The size of the buildings are not large compared to the expected size of robot we would expect to operate in them. There also need to be many checks of lines for collisions with solid objects (walls and other items in the rooms.)

Figure 1. 3D Image of Allensville

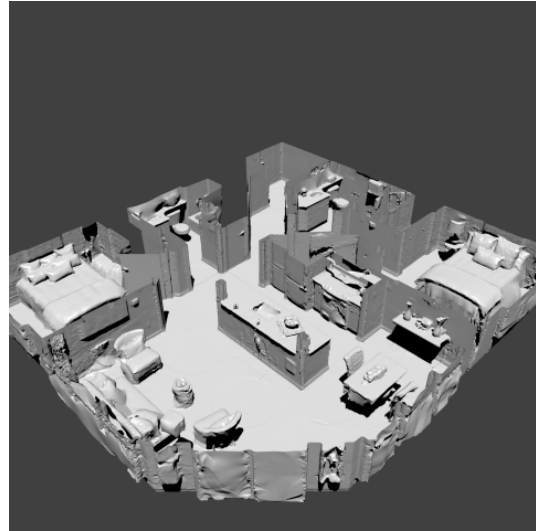
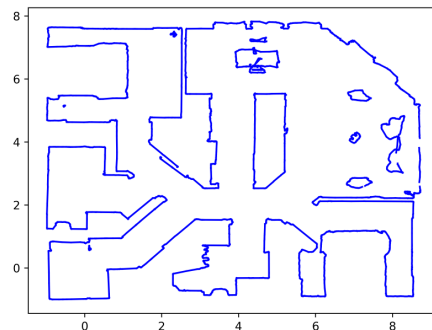


Figure 2. 2D Image of Allensville at 0.5 m



An efficient way to do these checks is by rasterizing the maps and using Bresenham's line-drawing algorithm to draw on the image. Solid areas are represented as black pixels, and free areas as white pixels. The way to do this is to use `opencv.floodFill()` starting at a point in the free space to fill the freespace with white pixels.

Even though the dataset documentation says most of the Gibson maps have had the holes filled, it is still the case

that the 2D has visible holes in the walls. This makes the `floodFill()` operation fail. These gaps can be filled “manually”, but this is tedious. So a function finds the end-points to all lines, and connects them to the nearest endpoint of another line. This can be done efficiently with numpy representations of the line arrays. Not all maps work, but enough to have a decent set to work with.

Finally, There are still small gaps between objects that are evidently too small for a robot to fit through, but which the RRT lines would traverse. To get rid of these, `opencv.erode` is applied to the image. This also has the effect of providing some space from the walls that represents the thickness of a robot. Frankly, I used only a few pixels of erosion, so the spacing effect is not realistically large. However, more erosion could certainly be used to get the correct effect.

Figure 3 shows a rasterized free space map for Allensville after gaps have been filled and erosion applied.

Figure 3. Allensville free space image.



In summary, the algorithm to convert a set of numpy arrays, each representing a line in the map, to the rasterized free space image, is:

1. Find minimum and maximum x and y positions of lines.
2. Pad lines out with configurable quantity (typically 0.5 m in examples.)
3. Draw lines with `opencv.polylines()` on free image initialized to black.
4. Calculate lines to fill gaps and draw free image.
5. Floodfill image with white starting at fixed point in image (this should be made configurable.)
6. Erode image with a 5x5 mask a configurable number of times (5 by default.)

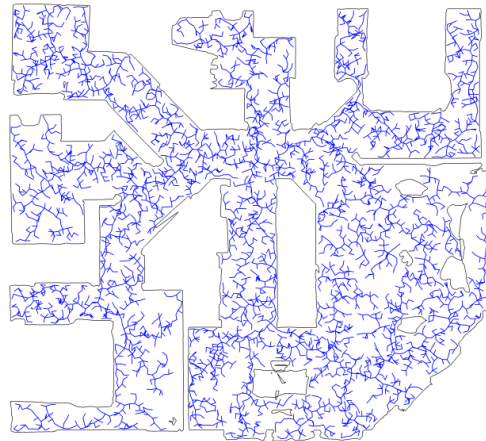
### 2.3. Creating the RRT

The algorithm to produce the RRT is fairly simple:

1. Find random starting point in free space.
2. Perform  $N$  times.
  - (a) Find random point  $s$  in anywhere the image.
  - (b) Find existing point  $t$  in tree closest to  $s$ .
  - (c) If line of length  $d$  can be drawn from  $s$  to  $t$  without hitting an obstruction (or, optionally, another line), then draw it, and add end of line as a new point in the RRT.

The check for an obstruction is made by using Bresenham line-drawing to trace the line on the free-space image. If it hits a black pixel then the line is not used. If the option not to cross tree lines is specified, then a copy of the free space image is made, and each successful new edge is drawn in black on it. This prevents future edges from crossing it. A sample RRT is shown in 4. The number of nodes to draw in the tree is configurable. For Allensville 5000 is sufficient to fill the graph. Larger spaces require more.

Figure 4. Allensville RRT



Two programs can be used to produce an RRT: `make_rrt.py`, and `rrtgui.py`. Both use the library file `rrt.py`. `make_rrt.py` creates RRTs with several command line options and saves the floormap image, with the RRT, the free image, and the parameters to files. `rrtgui.py` is a demonstration program that displays an animation of the RRT as `rrt.py` creates it. I did not get to the point of saving the products of `rrtgui.py` to file.

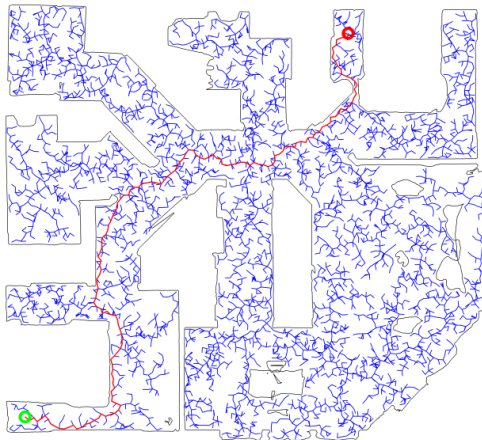
Before evaluating the usefulness of the RRT, we’ll examine the method to find paths between arbitrary points.

## 2.4. Finding a Path Using A\* Search

Given a starting point, and an ending point, both in real world coordinate measured in meters, A\* is used to find a path on the tree between them. This algorithm is basically a direct implementation of the one found in Russell and Norvig. The only difficulty was finding a heapq with replacement. I didn't find one, but subclassed the standard Python module as AStarHeap to work with special objects that have a deleted flag to support deletion. The AStarHeap has a dictionary of nodes to track when a node value already exists. This allows it to implement a method to see if a nodes already exists with a worse cost.

So the algorithm uses the free image to find the closest node to each of the start and end points, and then find the path. This is very fast. A sample path is shown in 5. The path start at the green circle and ends at the red one. This path looks reasonable, but because it has to follow the meandering RRT path, it doesn't really look as good as it could be.

Figure 5. Allensville Sample Path 1



If we look carefully at the circled area in 6 we see that the leaves of the RRT do not meet up behind the object (a sofa.) This is because an RRT really is a tree, not a graph. This leads to a path from one side of the sofa to the other that goes around in front of it, as shown in 7.

## 2.5. Refining the Path

In order to make the path more efficient we do the following:

1. Create a new graph with just the nodes and edges of the path.
2. Repeatedly add random edges between pairs of unconnected nodes.

Figure 6. Allensville RRT Gap

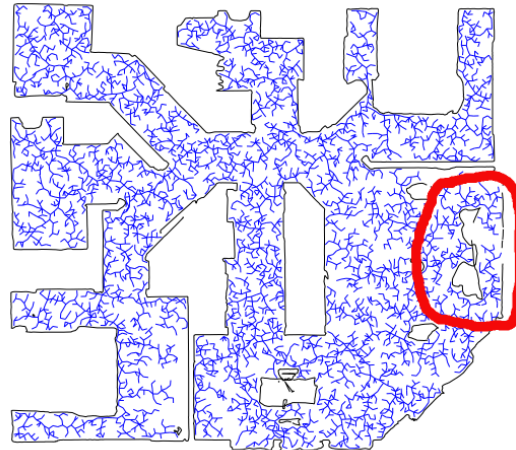
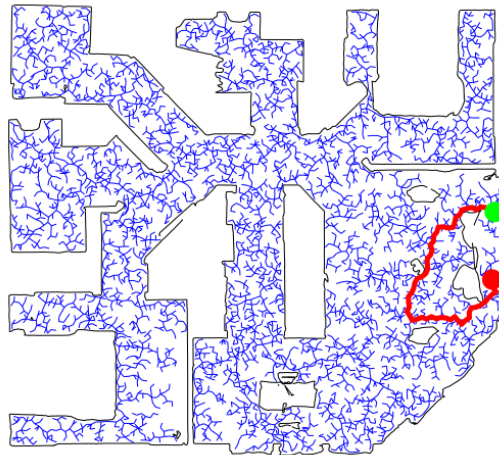


Figure 7. Allensville Sample Path 2



3. Check cost of A\* search for graph (since it is no longer a tree.)
4. When the cost appears to converge (no more improvements are being made to the cost of A\*) then return the new path.

We can see the effect in 8 of refining the path from 7. The path now goes directly behind the sofa, instead of going around the front. But rather than refine the path every time, we'd rather be able to have a static map and calculate the path one time.

## 2.6. Refining the Map

To make a fixed map that is more efficient, we perform the the steps in the previous section a configurable number of times on random start and end points. After each iteration, we add the new path the the map graph. Once this is

Figure 8. Refined Path

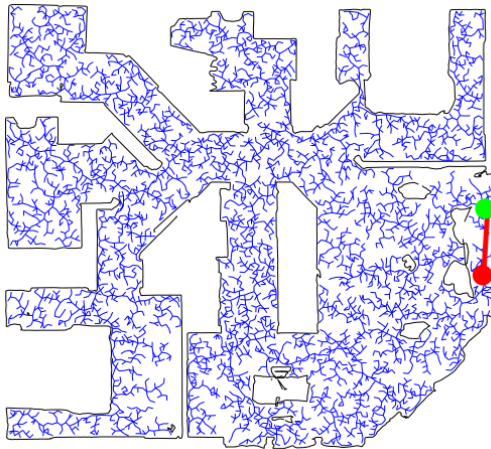
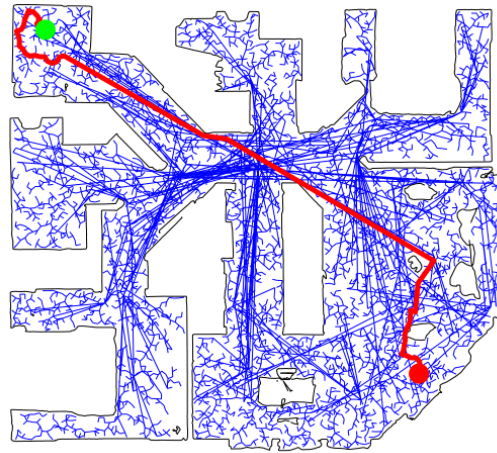


Figure 10. Refined Path 2



done we have a graph where the most common paths between major areas of the building have more direct straight-line routes.

Figure 9 shows a map with 100 added paths. Figures 10 and 11 show paths created using the refined map.

Figure 9. Refined Map

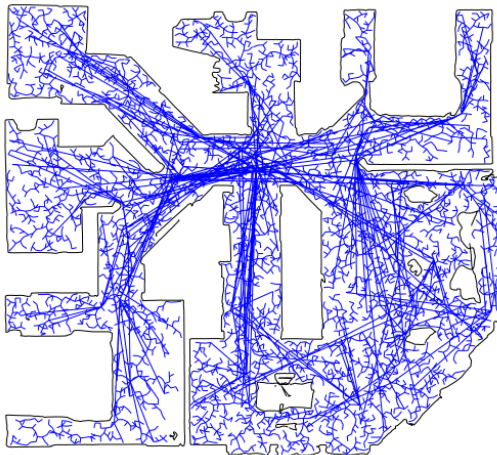
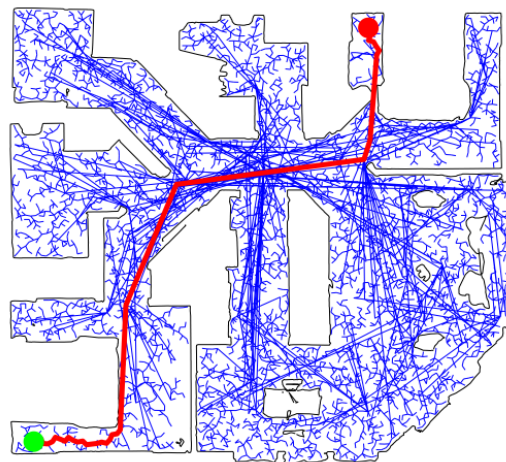


Figure 11. Refined Path 3



### 3. Results

Several results have already been shown. To demonstrate the fairly general applicability of this approach, here are some refined maps of other Gibson environments.

#### 3.1. References

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [?]. Where appropriate, include the name(s) of editors of referenced books.

#### 3.2. Illustrations, graphs, and photographs

All graphics should be centered. Please ensure that any point you wish to make is resolvable in a printed copy of the paper. Resize fonts in figures to match the font in the body text, and choose line widths which render effectively in print. Many readers (and reviewers), even of an electronic copy, will choose to print your paper in order to read it. You cannot insist that they do otherwise, and therefore must not assume that they can zoom in to see tiny details on a graphic.

When placing figures in  $\LaTeX$ , it's almost always best to use `\includegraphics`, and to specify the figure width as a multiple of the line width as in the example below

```
\usepackage[dvips]{graphicx} ...
\includegraphics[width=0.8\linewidth]
{myfile.eps}
```

### **3.3. Color**

Color is valuable, and will be visible to readers of the electronic copy. However ensure that, when printed on a monochrome printer, no important information is lost by the conversion to grayscale.

## **4. Final copy**

You must include your signed IEEE copyright release form when you submit your finished paper. We **MUST** have this form before your paper can be published in the proceedings.

Please direct any questions to the production editor in charge of these proceedings at the IEEE Computer Society Press: Phone (714) 821-8380, or Fax (714) 761-1784.