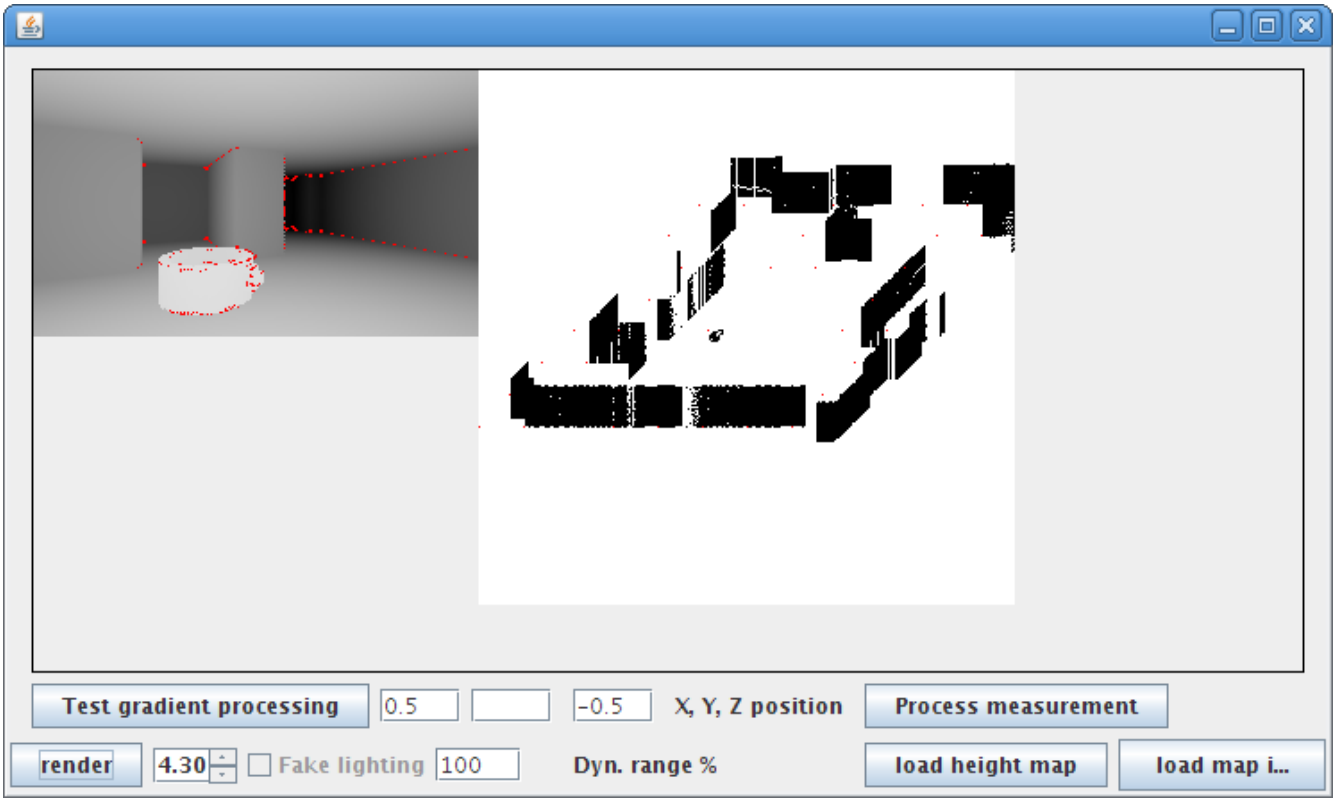
Simulator for the Ceilbot location and mapping software
Final report

Juhana Leiwo

2010

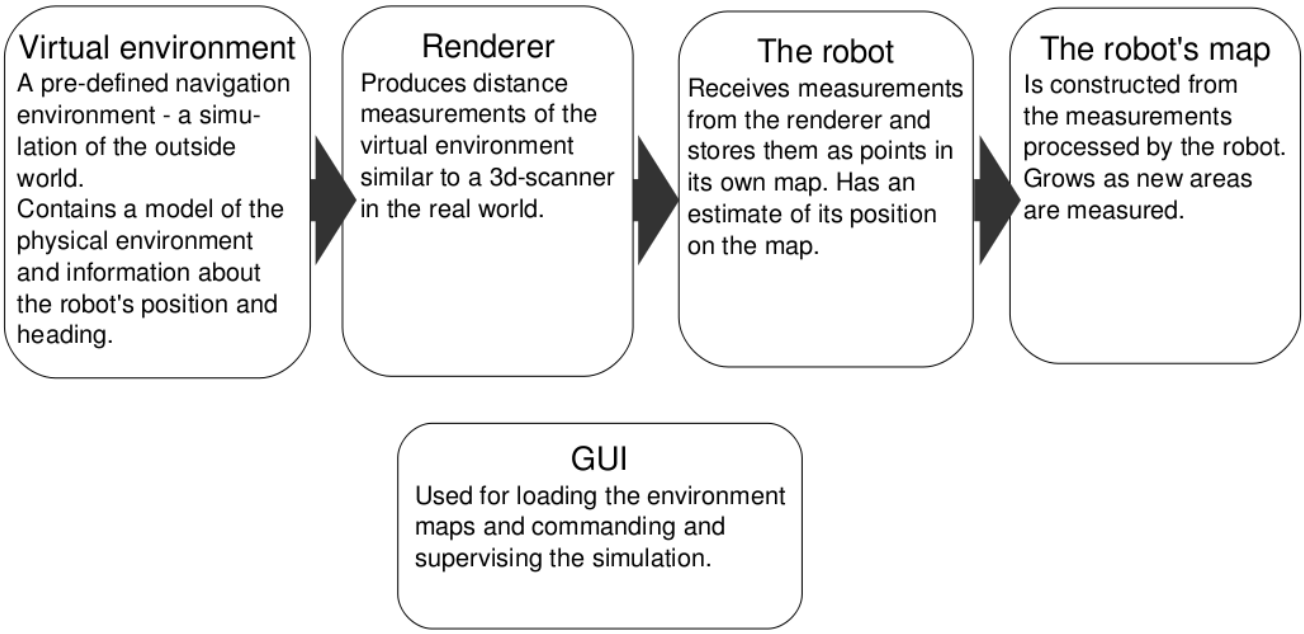


My goal in the Ceilbot project for the spring term of 2010 was to develop a simulator program for the testing and development of a simultaneous location and mapping system to be used in the Ceilbot. The plan was to create a program which would simulate a robot which takes spatial measurements from its environment and then uses these measurements to create a map of its surroundings and pinpoint itself on this map. The concepts developed in the simulator could then be tested in the real world with already a fair idea of how the algorithm should behave.
I started development on the simulator with the expectation that it would not be fully finished by the end of the term. As it was, I managed to create a working draft version with enough functionality to be a useful platform for further development. The environment and measurement simulation section is fully operable and the mapping section is finished to a point where a map can be created when the pose information is known to be correct. What's still missing is the ability to correct errors in pose between measurements or re-position a “lost” robot on the map based on new measurements. Moving the robot around the environment must currently be done manually. An autonomous navigation system is also missing. Development of these features is under way.

Structure of the simulator program

The program can be divided in sections based on functionality. The virtual environment section defines the “outside world”, the renderer creates distance measurements from the virtual environment like a range camera, the robot section receives these

measurements and processes them to be stored in the map section. User interaction and supervision is done through a graphic user interface (GUI).

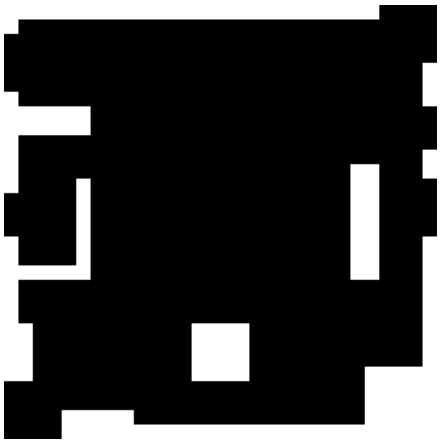


The construction of the program follows the situation in the real world so that the robot section can only probe the outside world through measurements from the renderer. The position and orientation of the robot is stored in the virtual environment and the renderer makes a measurement based on this information. The robot has information of its own movements relative to its earlier position, but it can't access the absolute position data stored in the virtual environment. The robot's relative position information is not tied to the absolute position variable. Error can be introduced between these two when testing location correction algorithms. In the current state of development the robot's relative position information is however kept correct to allow testing of the mapping algorithm without location correction.

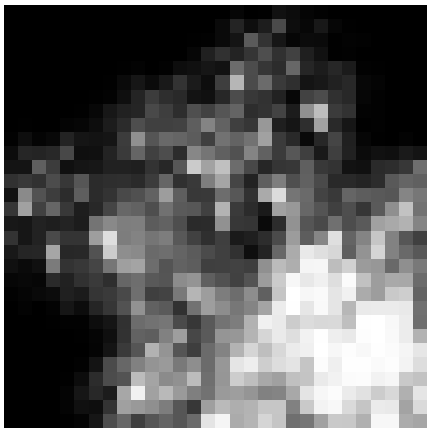
The virtual environment

The virtual environment is simply a map which tells whether there is empty space at a given coordinate or not. Because the simulator concerns a robot that is intended to operate inside the rooms of a house, the natural way to define the environment map is to define a floor, a ceiling and the walls. For defining such a room in terms of having empty space or not, a simple way is to use a boolean-valued function of a three-dimensional vector variable which gives “true” if the coordinates are full and “false” if the coordinates are empty.
In the simulator program the function is defined by marking the heights of the floor and ceiling with number variables and using a two-dimensional boolean array to represent the wall positions. A value of “true” in the array indicates a wall at that position and a value of “false” indicates empty space. A scale parameter defines the dimensions of one element of the array, and coordinates outside the array's range are considered “true”, or filled with wall.

The values of the wall array are parsed from a black-and-white bitmap image. Black is interpreted as empty space and white as wall. The dimensions of the image determine the size of the map array. An image map can also be loaded as a height map for the floor. In this case the floor height variable is replaced by a local height value defined by the height map's pixel value at that coordinate. In other words, a light color means a hump in the floor and a dark color means a pit.



Example map image



Example height map image

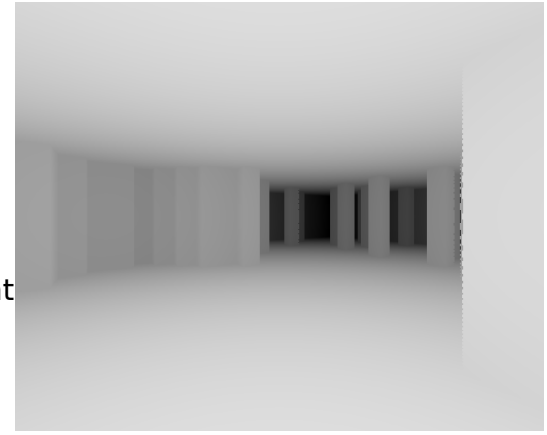
In addition to the floor, ceiling and walls, I also tested modeling objects by parsing a three-dimensional boolean array slice by slice from bitmap images. This worked, but without a dedicated voxel editor the process of drawing and loading the image slices to form an object was so tedious that I only ever created one teacup and left it at that.

Slices for a teacup object

The renderer

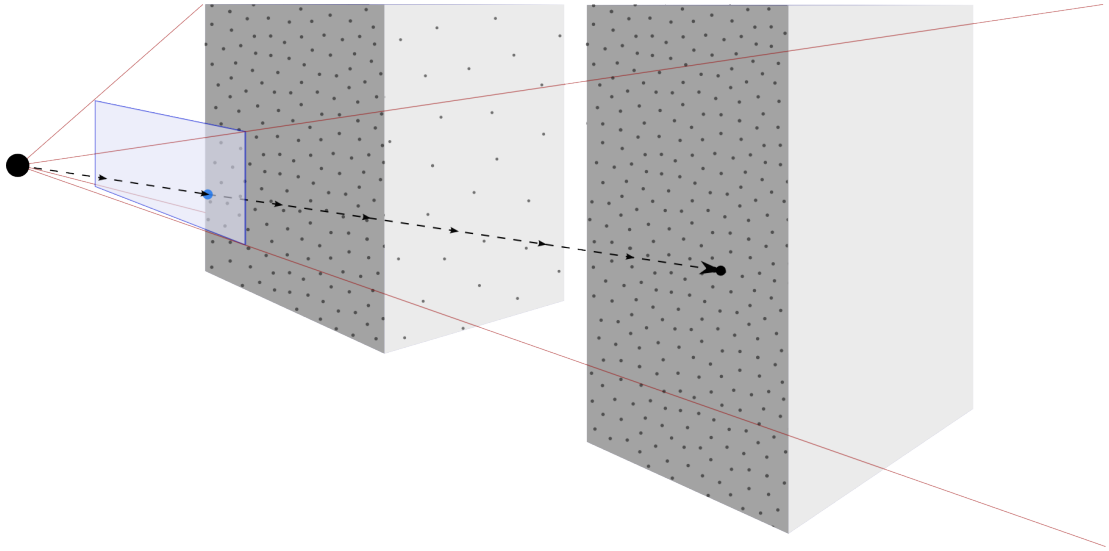
The renderer is the simulation equivalent of a 3d laser scanner. From a certain viewpoint at a certain focal angle it produces an array of values representing the distance from the viewing point to the first non-empty position of the virtual environment at that direction. The array of distance values is in effect a grayscale image of the environment with levels of gray representing different distances.

The renderer casts a ray from the viewing point through each pixel of the distance image, which can be thought of as hovering in front of the viewing point at a distance defined by a focal length vector. The ray is essentially a vector pointing into the direction of the pixel. The length of this vector is increased in small increments, and for each



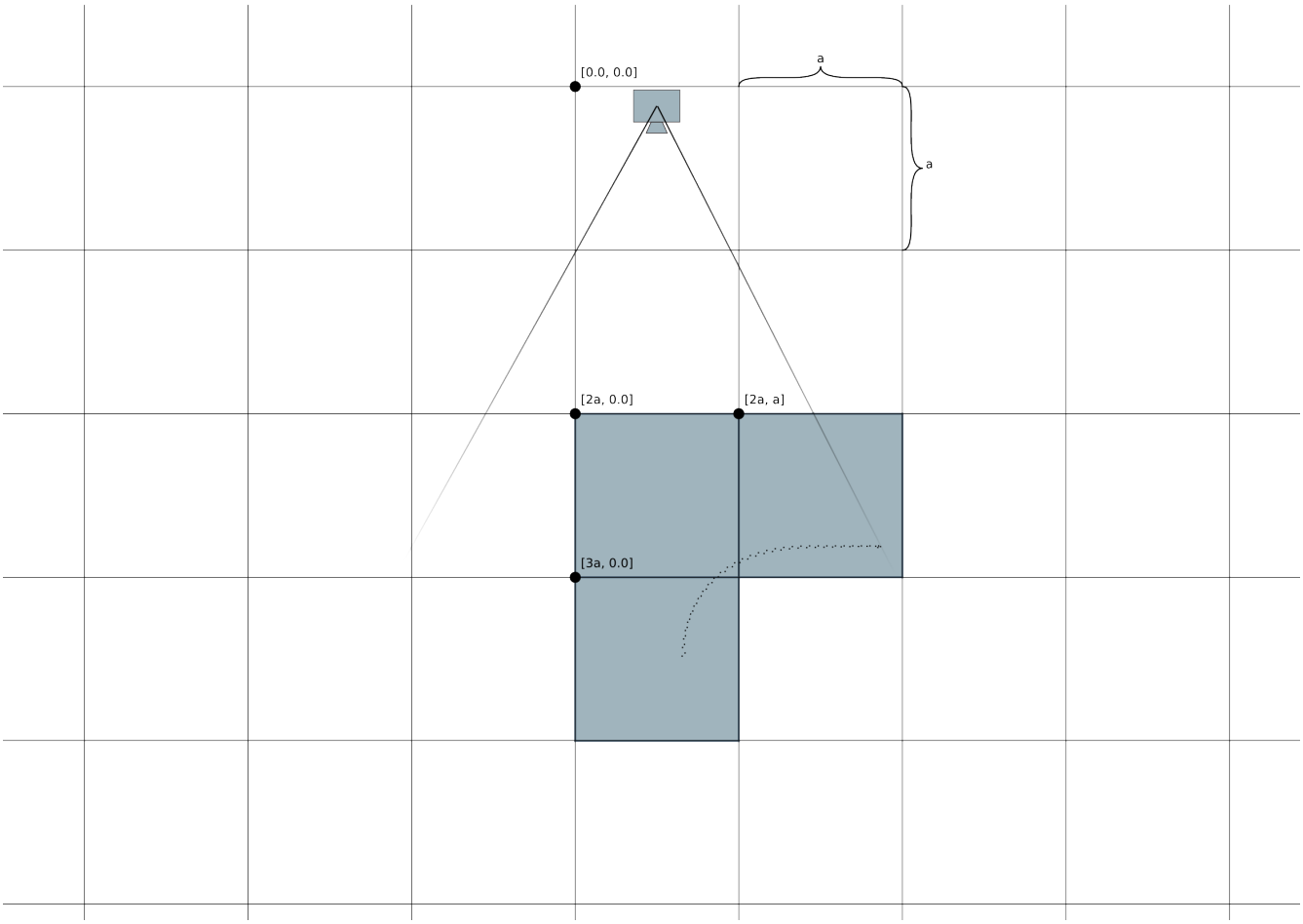
A rendered distance array shown as a grayscale image (the distances are processed as floating point numbers, not RGB values in the program).

increment the coordinates where the vector points at are checked to see whether they contain empty space or not. When the ray hits a non-empty region of space, the length of the vector is measured and stored into the measurement array.
The angular resolution of the renderer depends on the size of the measurement array and the focal length, and the distance resolution depends on the size of the ray increment. With smaller increments the distances are traced more accurately, at the cost of longer rendering times. The ray increment length is chosen to be similar to the distance resolution of a real laser time-of-flight 3d-scanner.
A slight problem in the renderer that is not present in real scanners is that the rays will not detect sharp corners, but rather pass through them. This causes the edges of columns appear jagged in the renderings. The performance is however good enough for this application, especially as the method is so simple.



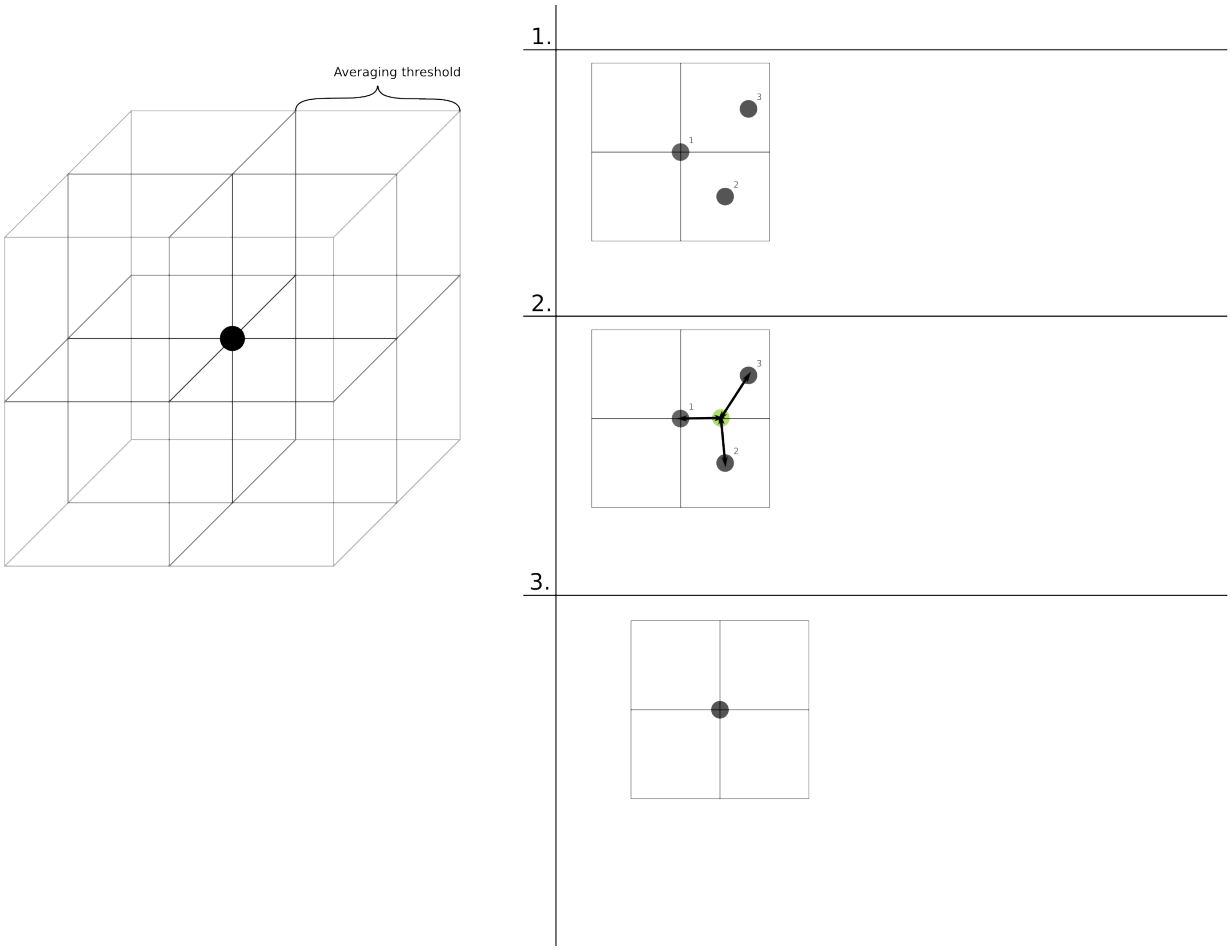
The robot's map

The measured distances are processed by the “robot” section of the program to form points with three-dimensional coordinates. The robot knows the focal length of the rendered array, and based on that and its estimate (or knowledge, in the current version) of its position and orientation, the distances are projected back into space as points. Those points are stored to form a map of the environment.
The map consists of so-called map cells, which subdivide the map space into regular-sized regions. The cells are arranged in a grid with fixed side length. Each cell covers an unique range of coordinates. When new points are added, they go to the cell which has the point's coordinates inside its range. The number of cells is not set in advance, new cells are created when points are added that fit no existing cells.



The figure shows how the mapping space is divided into a cell grid. The blue rectangles are existing cells, the white rectangles are empty. Cell side length is marked with a.

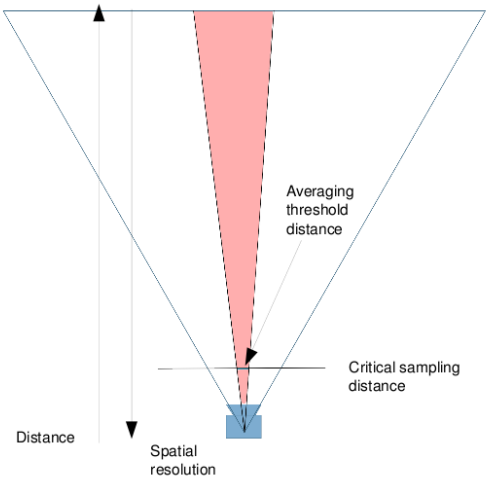
When new points are added into the map, they either get a new cell or are added to an existing cell. A point added to an existing cell is averaged with the other points in the cell. Each point already in the cell is tested to see if it lies within a threshold region of the new point. All points within the threshold region are collected, and the average coordinates of them and the new point are calculated. The collected points are then replaced with one new point in the average coordinates. This way a cell will not accumulate more points than the threshold region's size allows even when measurements from the same position are repeatedly added.



Points within the threshold region are averaged into one.

Optimization of adding points

The lateral distance in space between two elements of the measurement array decreases as distance to the viewing point decreases. In practise this means that points on a plane closer to the viewing point than a specific critical distance will always be closer together than the averaging threshold. This in turn means that adding a measurement of a close by wall will result in most of the measured points being averaged with their neighbours without contributing much to the map, especially when the same location has been measured before. A simple improvement is to calculate the critical distance and discard all points closer than that before adding to the map. The program currently does this. If, on the other hand, the location has not been measured before, throwing away the points wastes valuable information. In future development this will be taken to account, either by checking if the area has been measured before or pre-averaging the points before adding.



Critical sampling distance

$d_c = \tan a_{view} * d_{threshold}$

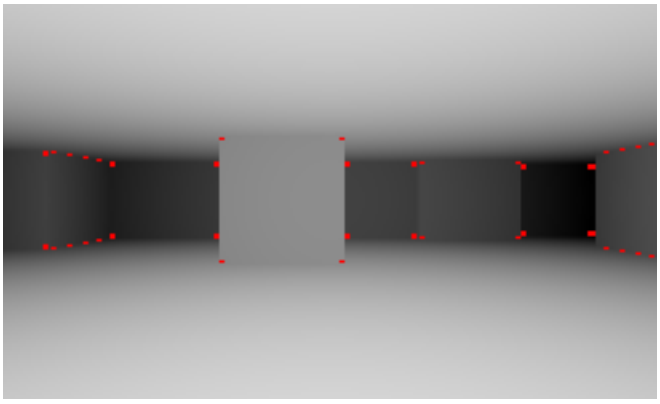
where a_{view} is the viewing angle and $d_{threshold}$ is the point averaging threshold

Location

To work in a real robot navigating in a real environment, the mapping algorithm must take into account that the robot cannot be sure whether it has moved and turned exactly the way it tried to. The robot must be able to align a new set of measured points with the ones already in the map and thus correct its location and orientation. This is a non-trivial task, because among other things even the measurement itself is by no means a perfect representation of the actual environment.
A relatively simple way to align the measurements is to use the Iterative Closest Point algorithm, which compares the points of two overlapping measurements and tries to find the closest pairs. There are shortcomings in this technique, however. For example, the algorithm cannot match scans that are rotated relative to each other, and although the result approaches the correct solution when more iterations are performed, it can't be said that after any specific amount of iterations the result is even close to correct.
Another way to attempt alignment is to extract special points of interest, that is, landmarks from the collected data and try to keep track of the positions of these landmarks relative to the robot. This method will be included in the future development of the simulator program.

Landmark extraction

The last feature I had time to include in the simulator program before the end of the term is an experimental method of finding landmarks from the measured array of points. The method looks for convex and concave places by comparing each point with its left, right, upper and lower neighbours. A vector is drawn to each of these points and normalized. Then the cross products of the vertical and horizontal pairs of unit vectors are taken and their magnitude compared to a (more or less arbitrarily chosen) threshold value. If the cross products both cross the threshold, that is, the vectors span a large enough area, the region is convex/concave enough and can be considered a corner - a good candidate for a landmark. The values of the cross products can be used to characterize the landmark.
The algorithm does find the corners in the virtual environment, but unfortunately it finds many false positives as well. When the vectors are normalized to make the cross products comparable, the scale information is lost. If the vector in one direction is a lot shorter than in the other direction, the actual shape of the area is often misinterpreted and the algorithm gives a false positive. Even with the true positives this might harm the characterization of the landmark - it might look very different from another point of view. I have thought about a few ways to improve this algorithm, for example blurring the measurement array and using a step larger than one when choosing the neighbouring points, or comparing the landmarks extracted from two different measurements a small distance apart and checking which ones stay still. I did not have time to make these improvements on this term, so I will continue on them in the summer and report when the project continues in the fall.



Found landmarks shown in red. True positives in corners, false positives along wall edges.