Project Report: Street Art Bot

Group 1
Jonathan Caines, Phoebe Howard,
Jake Pierrepont, Joel Shinu, Hitesh Verma

***Abstract*—Robots are now replacing activities that humans were once tasked with performing with a more reliable and accurate output as our society progresses towards automation. This paper shows a ROS (Robot Operating System) based street art robot that can map an unknown environment while locating itself within it. The user inputs an image, and the robot finds an appropriate place within the environment to paint the image on. The selection of ROS was justified as it provides the tools, libraries and capabilities needed to develop robotics applications whilst consuming resources minimally. The algorithms used to approach the task are shown in this study, together with experimental data that evaluate the system's efficiency and completion of the objective. The experimental data includes multiple versions of the robot which are extensively tested.**

The final version of the code can be seen on the Git link provided: https://git.cs.bham.ac.uk/jcp889/street-art-bot

I. Introduction

FOR our project we have chosen to program a robot to navigate an unknown environment, find a location to paint in and paint images provided by the user on the floor. It should be capable of mapping its environment using SLAM, aligning itself to be able to print in an area of space that is adequate to the image size and painting the image without skewing it, missing pixels, or painting in the wrong place.

This project will have several contributions made by us. Once the robot has mapped out its environment, we will design an algorithm to find a large enough area to paint in, or return an error if no such area exists. Once the robot has decided on an area to paint in, it must find a way back to that area of the map. This will be done with an implementation of the A* path-finding algorithm. This algorithm will also be used to move the robot back to the position it needs to paint next if it localizes somewhere else. After the robot reaches that area, we will use a localization algorithm based on Monte Carlo Localization, but with an additional sensor reading, to increase accuracy: colour. This will be useful while the robot is painting to constantly check if it is where it needs to be to apply the next pixel of paint.

This project will also use an existing implementation of SLAM, as the robot will need to map the environment out so it can then find an area to paint in.

II. Related Work

T. Lindemeier, L. Scalera and many others, have all produced robots able to paint an image to a canvas [1], [2], however their work was more focused on colour matching and the physical hardware of the paints and brushstrokes. Our work will be an abstraction of the real world, and as such the hardware is out of scope of this project. We will however build on this work in other ways as all of them used a canvas of fixed size and shape, known to the robot in advance, and the same resolution as the target image. Our canvas will be the floor of any given space, and as such there is much more work to be done before painting can begin, and more noise to deal with during the process.

Unlike the work produced by P. V. Arman, we are not trying to produce originality within our artwork [3]. One possible use case of our robot would be to create some digital art, and have the robot physically create it, possibly in multiple locations, for the purposes of advertising or simply creativity. However, it is not the focus of this project to have the robot be the artist, rather the robot is the tool used by the artist.

We will utilise an implementation of SLAM built by S. Kohlbrecher, J. Meyer, O. von Stryk and U. Klingauf, called Hector SLAM [4]. As the authors state in their paper, it is an efficient implementation of SLAM that requires low computational resources. It is also kindly included as an open source package for ROS.

III. Approach

For our approach, we will initially have the robot start at a specific position on a given map and paint an image on the floor at that location. The robot will have to stay localized while painting in order to create the image within that space. Once this is working we will have the robot choose its own position to paint at, and navigate to it on the map, in addition to navigation and image creation the robot will need to utilise an algorithm we will develop to find a good paint location. Finally the robot will not be given any data and will map out an unknown location using SLAM and find an area of appropriate size on the created map to paint in.

The painting will be done by drawing pixels on the map, which would be translated into the real world as some sort of spray can. These pixels should be placed next to each other to create a coherent image. To do this the robot will need to stay localised in relation to the image it has already painted. The robot will have the ability to view the paint that it has placed, and we believe that this could be used to assist in

localisation, providing a distinct marker to localise from as well as greatly helping with the kidnapped robot problem if the robot was kidnapped mid painting.

Image handling and manipulation will be handled by the python library called Pillow. It will be used to load the image that the robot will paint, as well as generating an image of the robot's finished painting and an image of a "perfect" painting. As it is a simulated environment the image generation step will provide an appropriate view of the map and the location of the finished painting an appropriate comparison point with the "perfect image".

IV. Algorithms

Our program will use several algorithms, some of which rely on existing implementations or methods, however two of which are our own original contributions, and are detailed below.

The first algorithm takes an image and a map as input, and finds a place to draw the image on the map such that it does not overlap any walls.

Algorithm 1 Finding a location large enough to paint in

```
I ← image of pixel width x and pixel height y
M ←occupancy map
for all grid-points g in M do
    if g = 0 then // 0 is a valid grid-point
        for all lines l in range(0, x) do
            for all pixels p in range(0, y) do
                G ← grid(g,l,p,M) // the pixel's grid-point
                if G = 0 then
                    continue
                else
                    break
        return Grid-point(g)
return Error - No Valid Location(e)
```

The algorithm goes through each valid location in the occupancy map M (a valid location is a location in which there are known to be no walls) and for each one checks if the image could be drawn with its top left corner there. To do this, it checks every pixel can be placed in the correct place relative to this position, and breaks if it ever finds a wall in the way. If it ever finds that the image can be drawn at a specific grid-point, it returns that grid-point, otherwise if it reaches the end of the occupancy map it returns an error. The algorithm makes use of the helper function "grid()" which takes the coordinates of a pixel on the image and returns the grid-point on the occupancy map that pixel would occupy if the image were drawn from point g on map M. The algorithm can be run twice for each image, if the image has a different height and width it is possible that it may fit if rotated 90 degrees, so if the first attempt fails the program should rotate the image and try again.

The algorithm below is our modified particle filter algorithm that uses a combination of a colour sensor and laser measurements to perform the observation step each time it tries to paint a new pixel.

Algorithm 2 Monte Carlo Localization using colours

```
cloud ← weighted particle cloud
do
    weightsum ← 0
    for all particle p in cloud do
        p.weight ←getWeight()
        weightsum ← weightsum + p.weight
    for all particle p in cloud do
        p.weight ← p.weight/weightsum
    U ← 0
    I ← 0
    C ← cloud[0].weight/weightsum
    newCloud ← weighted particle cloud
    for all particle p in cloud do
        if C > U then
            append cloud[I] to newCloud
            U ← U + 1/cloud.size
        else
            I ← I + 1
            C ← C + cloud[i].weight/weightsum
    cloud ← newCloud
    add gaussian noise to cloud
    N ← cloud.size
    M ←mode(all particle.position in cloud)
    E ← particle
    for all particle p in cloud do
        if p.position in area M±THRESHOLD then
            E.position ← E.position + p.position
        else
            N ← N - 1
    E.position ← E.position/N
while count(particles in area E±THRESHOLD1)<THRESHOLD2
return Position Estimate(E)
```

This algorithm waits for a good estimate of the robot's position before returning it. It does this by first assigning weights to all the particles from a new set of laser and colour readings via the helper function "getWeight()". After that the weights are summed and normalised. The next part of the algorithm is an implementation of the stochastic universal sampling algorithm in order to resample the cloud.

This checks the cumulative weight (C) against 1/number of particles (U). If it is larger, we add the current particle to the new cloud, otherwise move on to the next particle to inspect and increment the cumulative weight. This can often lead to a the particles converging on one point so to combat this a small amount of gaussian noise is added. By adding this noise the robot is better able to correct a bad estimate.

The next step is to actually get our position estimate. This

is achieved by taking the most common area of the particle cloud (using helper function "mode()") and then taking the average of the positions of all particles that fall within a certain distance threshold of the mode. Finally there is a check to see how many particles are near the estimate to see if the estimate is good. If it is deemed good then it is returned, otherwise the algorithm starts over.

V. Experimental Evaluation

A. Testing Methodology

Testing of the code is done by evaluating the robot on its ability to complete the objectives set out at the start of this project. Namely, its ability to:

- Paint a recognisable rendering of the image provided on the ground
- Choose an area to paint that is large enough for the image
- Paint the image within that area (not offset or skewed)
- Return to the image it is painting if moved during the process

We evaluate the robot based on the tests set out in table 1. For each test we adjust several variables (map, image, noise, external movement of the robot).

TABLE I
Testing Plan

Requirement	Test	Desired Result
Robot can paint a good approximation of an image onto the ground	Visual comparison of the image given to the robot, and its drawing, possibly using test users feedback to avoid bias	Visually similar images, can tell what the image is of using the robot drawing, no major differences that distract from the image.
Robot chooses an area to paint in that has space for the whole image.	Give the robot an area to paint in with many walls. Does the robot hit a wall when trying to draw out the image? Is it able to finish the painting? What about if the robot is given an image too large to fit on the map anywhere?	The robot maps out an area and ensures it is large enough before it starts painting. If the image is too large, the robot should give an error message, not simply crash or try to paint the image anyway.
Robot paints the image in the area it selected/was told to use.	Give the robot an area to paint in and an image. Is there paint outside the selected area? Does the image warp or shrink in some way?	No paint outside the selected area and no warping or shrinking of the image.
Robot can deal with being moved during the painting process.	Move robot to different locations, both on and off the image. Does the robot reset itself if moved to a different point on the image or just carry on? Does the robot find the image again if placed in a different location on the map?	Robot should be able to localise to both the image and a specific point on that image to carry on painting.

B. Testing Results

The first version of the code we tested fully was our prototype, this had all the functionality we intended, but had not been tuned or tested for accuracy yet, and as such performed rather poorly. In around 2 and a half hours, this first prototype was able to paint a 32 by 32 pixel image with many errors, and offset from the intended paint location. The full results are below:

TABLE II
First Prototype

Requirement	Test	Result
Robot can paint a good approximation of an image onto the ground	Visual comparison of the image given to the robot, and its drawing	Similar colours used in the general vicinity of their intended location, but no clear image formed.
Robot chooses an area to paint in that has space for the whole image.	Give the robot an area to paint in with many walls. Does the robot hit a wall when trying to draw out the image? Is it able to finish the painting? What about if the robot is given an image too large to fit on the map anywhere?	The robot frequently hits walls when trying to paint, as the first available paint location is often adjacent to a wall. If given a very large image, the code returns None and then crashes.
Robot paints the image in the area it selected/was told to use.	Give the robot an area to paint in and an image. Is there paint outside the selected area? Does the image warp or shrink in some way?	The image appears shrunken and displaced from the intended paint location.
Robot can deal with being moved during the painting process.	Move robot to different locations, both on and off the image. Does the robot reset itself if moved to a different point on the image or just carry on? Does the robot find the image again if placed in a different location on the map?	As the pathing and painting code are separate, if the robot is moved during painting it tries to use "simple movement" to get back to its next paint location and often tries to go through walls.

There were clearly major areas for improvement following this initial testing, so we made the following improvements:

- Give the robot a buffer around the image when calculating its paint location, reducing the probability of hitting a wall
- Do further tests of the localisation code, to discover why the image is being painted in the wrong place.
- Increase the accuracy of the painting code by decreasing the range in which a pixel is considered to be in the correct position
- Combine the code for painting the image and pathing, leaving the robot free to go back to A* pathing if it gets sufficiently displaced from the image

As mentioned above, we did some additional testing of the localisation code, as it appeared to be performing worse than our assignment 1 localisation code. We tested the code to paint a simple 8 by 8 pixel image, and found that although the offset between the estimated and actual pose were similar, the code struggled to localise correctly when using the added weight from the colour sensor, which was why the image was painted in the wrong place, and was also contributing to the painting process taking longer to run. We believe this is due to the colour sensor not having any hits early on in

the painting process, and as such adding a near constant to all of the weights, which makes smaller weight particles more likely to produce new particles in the next generation of the particle cloud, while not affecting the larger ones. We therefore came to the conclusion that our initial hypothesis that adding colour sensor data to the particle cloud would make for a more accurate localisation was in fact incorrect. We decided to complete the project leaving both modes of operation intact, such that we had both a better working version and one that matched our original project goals.

We also discovered that the check to ensure sufficient particles were in an area around the estimated pose was not helpful. This was because the amount of particles around the estimated pose was mostly defined by the amount of gaussian noise we added to the cloud. We attempted to change this to a "confidence" value that would take into account multiple other factors such as the weights of the particles. This however proved to be very volatile. Using either of these checks in those forms would have not improved the localisation and slowed it down, so this was scrapped.

With the other changes made and the colour localisation turned off, we then had a second prototype, which we performed further testing on. This version was still far from perfect, but it had pretty much dealt with all the localisation issues we were having in the first version, whilst also increasing the speed at which it painted. It was now capable of painting a 64 pixel image in around half an hour. The full results for this prototype are bellow:

This is as far as we were able to get with this project, leaving us with a version of the code that is able to perform all the functionality we intended, though with very limited success in some areas.

TABLE III
SECOND PROTOTYPE

Requirement	Test	Result
Robot can paint a good approximation of an image onto the ground	Visual comparison of the image given to the robot, and its drawing	The image was painted, but large parts of it did overlap and were out by about 1-2 pixels, this was likely because of the offset between the estimated and actual pose, which we had no easy way of reducing.
Robot chooses an area to paint in that has space for the whole image.	Give the robot an area to paint in with many walls. Does the robot hit a wall when trying to draw out the image? Is it able to finish the painting? What about if the robot is given an image too large to fit on the map anywhere?	The robot never hit a wall while painting. If given a very large image, the code returns None and then crashes.
Robot paints the image in the area it selected/was told to use.	Give the robot an area to paint in and an image. Is there paint outside the selected area? Does the image warp or shrink in some way?	The image is slightly less shrunken and displaced, but there is still paint outside its intended paint area.
Robot can deal with being moved during the painting process.	Move robot to different locations, both on and off the image. Does the robot reset itself if moved to a different point on the image or just carry on? Does the robot find the image again if placed in a different location on the map?	The robot deals with the kidnapped robot problem seamlessly, returning to its painting task from anywhere on the map and at any point.

VI. CONCLUSIONS

For our project, we proposed a ROS (Robot Operating System) based street art robot that can navigate to a location suitable for painting an image in. The robot uses the image dimensions to find an appropriate location for it to paint in, within the known map. We did not get to test in the situation where the robot uses a SLAM implementation to first map out the area, however in an already known map if there was an appropriate area, it would find it. An implementation of the "A*" path-finding algorithm leads the robot to the painting location to then start the painting algorithm, moving to the next position to paint the pixel of the image until none remains, all the while using our estimated position from the localisation implementation to check its position before painting.

The colour aspect of the localisation proved difficult to implement and would be the first thing to improve in a future implementation as using the in progress painting as a landmark/orientation point would improve the accuracy of the image painting. The attempt to use the confidence in the estimate to improve localisation failed, this could potentially help to get a more accurate localisation in future, so warrants further investigation. A modification to the painting algorithm would involve optimising the distance between the current and next pixel to paint, improving the speed of the overall painting process and potentially allowing for larger images to be painted in the same amount of time.

VII. REFERENCES

REFERENCES

[1] Thomas Lindemeier et al, *Hardware-Based Non-Photorealistic Rendering Using a Painting Robot*. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1111/cgf.12562

[2] Lorenzo Scalera et al, *Non-Photorealistic Rendering Techniques for Artistic Robotic Painting*. [Online]. Available: https://www.mdpi.com/2218-6581/8/1/10/html

[3] P. V. Arman, *Pindar Van Armen's Cloud Painter*. [Online]. Available: https://www.cloudpainter.com/

[4] S. Kohlbrecher, J. Meyer, O. von Stryk and U. Klingauf, *A Flexible and Scalable SLAM System with Full 3D Motion Estimation*. [Online]. Available: https://www.sim.tu-darmstadt.de/publ/download/2011_SSRR_KohlbrecherMeyerStryk Klingauf_Flexible_SLAM_System.pdf