Simulated Navigational Assistance Device for Visually Impaired Persons

Katie Foss and Advait Rane

University of Southern California, Los Angeles CA 90007, USA
katiefos@usc.edu
advaitran@usc.edu

Abstract. A visually impaired person or (VIP) can use a walking cane or even a smart cane to help them complete navigational tasks in their day to day lives. However, current devices on the market only tell a user an obstacle is ahead, not how to avoid the obstacle. In this paper, we present a simulated wearable device that communicates to a user what direction to walk in to avoid an obstacle. We evaluate the wearable device against STL safety requirements.

Keywords: autonomous cyber-physical systems, object detection, navi,, robustness

1 Introduction

A visually impaired person or (VIP) has multiple options for obstacle avoidance. These options range from walking canes, apps like *Be my Eyes*, and smart walking canes like *WeWalk*. Recently a new line of products that take advantage of sonar and lidar have hit the market. These systems alert a user of a potential obstacle but don't provide a suggested navigational path around the obstacle. In this paper, we present a simulated wearable device that communicates to a user what direction to walk to avoid an obstacle. The device contains an RGB-D camera and a controller to make a path planning decision. The camera feed is sent to the controller, and the controller outputs are sent to the device wearer. After a series of simulation runs, we evaluate the wearable device against 3 STL safety requirements.

2 The Approach

The experiment is split into two parts, the simulation and the safety evaluation. Because the environment is simulated, the controller can be tested in a series of episodes. Episodes start with the user walking straight on a sidewalk with obstacles randomly placed on the sidewalk. As each episode runs, data about the episodes are written to disk as traces. After completing all episodes, the traces are used to calculate the robustness of the signal temporal logic (STL) requirements.

2.1 Simulation

Figure 1 displays a summary of the simulation architecture. A wearable device with an RGB-D camera is worn on the chest of a visually impaired person (user). A controller on the wearable then processes the camera footage and outputs instructions to the user to go straight, left, right or stop. In the simulation, the direction instructions are sent directly to the human controller; however, these instructions could be audio or haptic feedback from the wearable device to the user in real life.

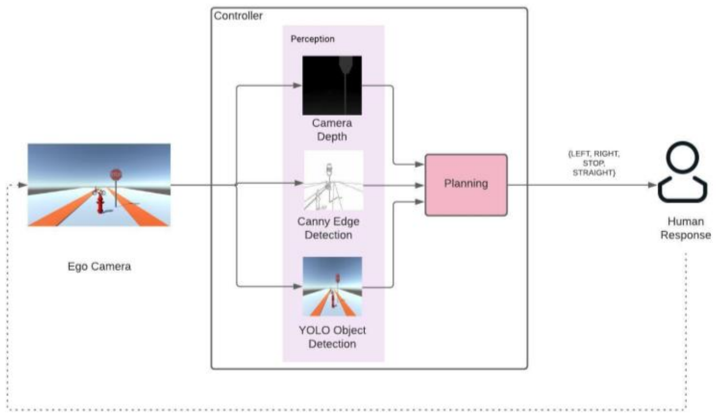

Fig. 1. Simulation Architecture Diagram

The controller itself is split into two parts, perception and planning. The perception portion uses the camera input to locate obstacles, determine their depth, and find the sidewalk's edge. Objects are detected using a trained neural network, and bounding boxes are visually drawn in the simulation. Canny edge detection is used to determine the left and right edges of the sidewalk.

The planning portion of the controller analyzes the perceived environment and outputs instructions to the user. To determine the next instruction to give the user, we implemented a similar technique as presented by Bor-Shing Lin et al. Bor-Shing Lin et al. proposed splitting a pedestrian's path into three lanes: left, center, and right [1]. If an obstacle is detected in a lane, another lane is chosen as the next path forward. Fig. 2 shows the sidewalk split into three lanes.

Fig. 2 also shows a detected bicycle (far away) blocking an otherwise free lane. To determine when an obstacle should be considered, our simulated wearable device relies on depth-camera reading. A threshold is used to ignore obstacles that are too far away. Without a depth threshold, the user would not be able to move forward in Fig. 2.

To keep the scope of the experiment manageable, the simulation was designed with the following constraints:

- The user will walk only on a straight sidewalk
- No cracks or lines appear on the sidewalk
- Obstacles will be a fire hydrant, stop sign, or bicycle
- Only the listed three obstacles will be on the path

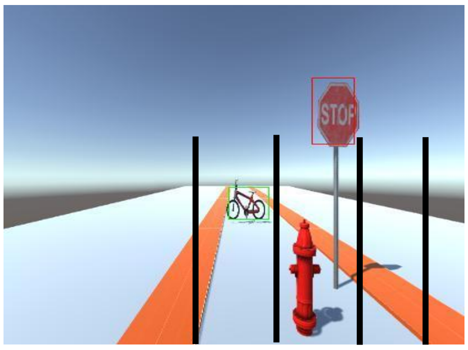

Fig. 2. Bicycle blocks lane

During a run of the experiment, the simulated user will try to walk the length of the sidewalk. Once the simulation is done running, the traces are saved to disk, and the safety portion of the experiment begins.

2.2 Safety

Safety considerations are imperative in this system since it directly interacts with a human and directs them in an environment where accidents could occur. We evaluate the safety of our system using three requirements framed as STL formulas:

1. Stay on the path: the user should stay within the bounds of the sidewalk. This can roughly be stated as "$G((x_{user}>x_{leftSidewalk})\&(x_{user}< x_{rightSidewalk}))$"
2. Avoid obstacles: the user should not collide with an obstacle by maintaining a safe distance from all obstacles. This can be roughly stated as "$G((d_{fireHydrant}>d_{safe})\&(d_{stopSign}>d_{safe})\&(d_{bicycle}>d_{safe}))$"
3. Reach the end: the user reaches the end of the sidewalk. This is stated as "$F(z_{user} \geq z_{sidewalk})$"

The robustness values are used as a metric to evaluate the satisfaction of the formulas. Since all of our requirements are either "always" or "eventually" formulas, we can use the robustness value at the first time step to evaluate satisfaction over the entire trace.

3 Simulation Results

We ran episodes with randomized obstacle positions to evaluate our controller under different circumstances. We fixed each obstacle to a lane and randomized the distance of the obstacle from the user. We then compared the performance of our controller on shorter sidewalks as opposed to longer ones. We ran 20 episodes at two different sidewalk lengths and the results are shown in Fig. 3.

We found the controller performed well in trials where the obstacles were spaced out such that at least one lane was free at all times. We observed the best performance when two obstacles were at a similar distance and the third was further away, allowing us to avoid the first two obstacles by finding the free lane and then avoiding the third obstacle further down. As a result, our controller had more success on longer sidewalks than smaller sidewalks because obstacles were further spread apart.

We observed poor performance in trials where the obstacles were placed closer together, in a cluster. In these cases, the controller continuously tries to switch lanes until it is too close to detect obstacles and crashes. Furthermore, the controller struggled to stop the user from side stepping into other obstacles. The controller would direct a user to switch lanes to avoid the first obstacle, see a second obstacle in the new lane and immediately direct the user to switch back into the lane with the first obstacle. This causes the user to sidestep into the first obstacle.

For the trials in which the user successfully navigated the sidewalk, the robustness values of all three requirements were positive in the first timestep. For trials in which the user crashed into an obstacle, the robustness values for the second and third requirements (reaching the end of the sidewalk and avoiding a collision) were negative. The third requirement had a negative robustness value when we stopped to avoid obstacles. The first requirement always had a positive robustness value as the user never left the sidewalk in our runs..

Description	# of Trials	Meet all 3 requirements	1. Stay on Path	2. Avoid Collisions	3. Reach end of Path
Longer Sidewalk	20	5	20	7	5
Shorter Sidewalk	20	10	20	13	10

Fig. 3. Perception Summary

4 Implementation Details

The simulation is built using Unity3D. Unity is a game engine that can be used to build complex simulations. The code to produce our simulation is written in C# and automatically generates scenes to randomly place obstacles in the path of the pedestrian. The simulation consists of 4 physical components: a camera, human, obstacles, and sidewalk. Table 2 summarizes the Unity assets and packages used to produce the simulation.

4.1 Perception

The camera is a Unity component and provides the visual input to our controller. The camera's output is written to a Render Texture. The Render Texture is either used directly or written to a Texture2D to do canny edge detection, object detection, and depth processing.

Canny Edge Detection. *Paper Plan Tools* published the OpenCV+Unity asset. This asset allowed us to apply Canny edge detection to the Texture2D of our camera feed. The black and white image in Fig.3 labeled *Canny Edge Detection* shows the edges detected. The controller finds the edges of the sidewalk by searching for the black pixels in the Texture2D.

Object Detection. Unity provides a "*lightweight and cross-platform Neural Net inference library*" called Barracuda [8]. We chose the pre-trained Convolutional Neural

Network (CNN) YOLO v4 (or *you only look once)* trained on 80 object labels, including those in the simulation. We also experimented with TinyYOLOv2 and Faster-RCNN models, but YOLOv4 performed the best. The models were accessed as onnx files from [7]. Barracuda can be run on CPU or GPU; this was particularly useful as the object detection is the slowest part of the controller.

Depth Processing. The unity camera has a depth channel that provides very accurate pixel depth metrics. Unfortunately, these metrics are only available to the GPU and not the CPU. To get the depth of a pixel location in the image, a shader must be written to render the depth using the GPU. The depth is then rendered to a Render Texture, and the greyscale value of the render texture can be used to compare against the depth threshold. In the simulation an exact depth is not used, instead a value between 0 and 1 (white and black) is used, 0 representing a location close to the camera and 1 representing far away from the camera. To better mimic a depth camera in a real-world environment, we add a small random noise to the calculated values.

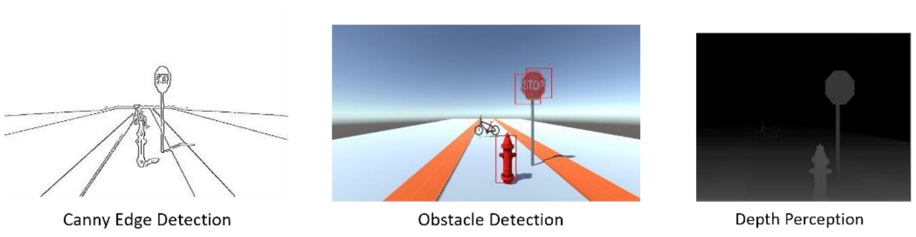

Fig. 3. Perception Summary

4.2 Planning

The controller logic is explained in the following pseudocode. These steps are repeated at each timestep step.

```
1.  Convert camera output to a texture
2.  Use object detection to get bounding boxes from the texture
3.  Calculate the depth of each pixel in the texture
4.  Use Canny Edge Detection to get the edges from the texture
5.  Estimate the edges of the sidewalk from the edges estimated in step 4
6.  Estimate the current lane of the user with respect to the sidewalk edges in step
    5
7.  If the current lane is free, move straight. A lane is free if none of the bounding
    boxes within a certain depth threshold intersect with the lane, and moving into
    the lane would not make the user reach a sidewalk edge.
8.  Else check if a lane to the right is free and if it is, move to the right
9.  Else check if a lane to the left is free and if it is, move to the left
10. Else stop
```

Some additional considerations are:

1. When moving to the right or the left, we switch entirely into that lane and move a little forward. Not moving forward could cause the user to get stuck in a loop where they kept changing lanes without moving forward at all.
2. To calculate the closeness of a bounding box, we used the maximum value of the depth of the pixels in that box. We could also use the mean value.
3. We only divide the image between the sidewalk edges into three lanes. We divide it into three lanes simply using vertical lines.
4. The evaluation of whether we hit the sidewalk or whether a box is in a lane is done at the pixel level in the image. We add fixed buffer values while evaluating these inequalities since our estimates may be noisy, which could lead to undesirable decisions when evaluated at the pixel level.

4.3 Safety

We evaluate the three safety requirements programmatically using the rtamt Python package. We log the location of all objects at each time step in trace files for all trials and use offline dense time STL specifications. The formula evaluation produces the robustness value at each time step when there is a change in the value. We only refer to the robustness value of the first time step for each trial.

Table 2. Unity Packages and Assets Summary

Name	Use	Description	Author/Publisher	Type
Unity Barracuda	Object Detection	Detect obstacles with YOLO model	Unity	Package
OpenCV+Unity	Edge Detection	Canny Edge Detection for sidewalk	Paper Plan Tools	Asset
GAZ Street Props	Obstacle	Fire hydrant and bench obstacle model	GAZ Game	Asset
FREE Bicycle Pack	Obstacle	Bicycle obstacle model	Undertone	Asset
Old Road Signs PBR	Obstacle	Stop sign obstacle model	Aleks Barnes	Asset
3D Character Dummy	Human Model	Human model in simulation	Kevin Iglesias	Asset
Free Basic Motion	Human Animation	Animate the human to walk are stop	Kevin Iglesias	Asset

5 Conclusion

In this paper, we outlined a system that assists visually impaired people in navigating obstacles on a sidewalk. We created a simulation of a sidewalk in Unity and designed a controller that directs the user to move straight, left, right, or stop to circumvent obstacles and reach the end. We use Object Detection, Canny Edge Detection and Depth Sensing for perception, and a simple planning algorithm that divides the input image into three lanes to find the free lane based on the perceived obstacles. Our algorithm performs well in several trials on our simulation. However, in a safety-critical system like this, a system must perform better to prevent accidents, as is observed by the non-satisfaction of the STL safety requirements we prescribe.

5.1 **Future Consideration**

Several future improvements could be made to the system to boost performance or make the algorithm more efficient. These are listed below-

1. We may be able to avoid using a RGB-D camera and only an RGB camera by calculating the depth of obstacles by using camera intrinsics.
2. Our algorithm will direct a user to walk into previously detected obstacles to avoid a newly detected obstacle. This could be avoided by storing the information about detected obstacles in a memory buffer and checking each obstacle for collision before taking a step. Additionally sonar could be used to detect obstacles outside of camera view.
3. Our control algorithm does not plan into the future. We take steps to avoid nearby obstacles without planning an entire trajectory at the start. In the future planning algorithms should be considered to improve controller performance.

5.2 **Related Work**

There are different approaches for navigation assistance for VIPs in literature. Our approach is most similar to Lin et. al., but we build on their depth estimation technique by using an RGB-D camera (or potentially monocular depth estimation) instead [1]. Aladren et. al. also use an RGB-D camera to create an obstacle-free path segmentation using RANSAC, polygonal segmentation, and watershed segmentation [2]. Bulat & Glowacz provides navigation assistance using simpler image processing techniques from OpenCV and a stereoscopy video system [3]. Edge detection and Local Depth Hypothesis is used on RGB images in Praveen & Paily [4]. We use a CNN and Canny edge detection as opposed to these image processing techniques.

Navigation assistance on a larger scale as Wayfinding has also been studied in Bai et. al. [5] and Heuten et. al. [6] using SLAM and GPS to construct longer routes to destinations.

References

1. Lin, B.-S., Lee, C.-C., & Chiang, P.-Y. (2017). Simple Smartphone-Based Guiding System for Visually Impaired People. In Sensors (Vol. 17, Issue 6, p. 1371). MDPI AG. https://doi.org/10.3390/s17061371
2. Aladren, A., Lopez-Nicolas, G., Puig, L., & Guerrero, J. J. (2016). Navigation Assistance for the Visually Impaired Using RGB-D Sensor With Range Expansion. In IEEE Systems Journal (Vol. 10, Issue 3, pp. 922–932). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/jsyst.2014.2320639
3. Bulat, J., & Glowacz, A. (2016). Vision-based navigation assistance for visually impaired individuals using general purpose mobile devices. In 2016 International Conference on Signals and Electronic Systems (ICSES). 2016 International Conference on Signals and Electronic Systems (ICSES). IEEE. https://doi.org/10.1109/icses.2016.7593849
4. Praveen, R. G., & Paily, R. P. (2013). Blind Navigation Assistance for Visually Impaired based on Local Depth Hypothesis from a Single Image. In Procedia Engineering (Vol. 64, pp. 351–360). Elsevier BV. https://doi.org/10.1016/j.proeng.2013.09.107
5. Bai, J., Lian, S., Liu, Z., Wang, K., & Liu, D. (2018). Virtual-Blind-Road Following-Based Wearable Navigation Device for Blind People. In IEEE Transactions on Consumer

Electronics (Vol. 64, Issue 1, pp. 136–143). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/tce.2018.2812498

6. Heuten, W., Henze, N., Boll, S., & Pielot, M. (2008). Tactile wayfinder. In Proceedings of the 5th Nordic conference on Human-computer interaction building bridges - NordiCHI '08. the 5th Nordic conference. ACM Press. https://doi.org/10.1145/1463160.1463179
7. Onnx. (n.d.). Onnx/models: A collection of pre-trained, state-of-the-art models in the ONNX format. GitHub. Retrieved November 12, 2021, from https://github.com/onnx/models#object_detection.
8. Unity Technologies. (n.d.). Unity Barracuda: Barracuda: 0.7.1-preview. Barracuda | 0.7.1-preview. Retrieved November 10, 2021, from https://docs.unity3d.com/Packages/com.unity.barracuda@0.7/manual/index.html.