CleanAR

Authors: Nathalie Jeans, Harshul Singh, Erin Gao
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of mapping a floor using Augmented Reality to create an overlay. This mapping will keep track of remaining uncovered floor space, erasing parts of the overlay that have been covered by the vacuum. In addition, there will be a camera behind the vacuum to detect whether the dirt has been removed or not, highlighting the area where dirt is still present. Overall, this augmented-reality and object detection based system improves baseline vacuuming coverage and cleanliness.**

***Index Terms*—Augmented Reality, Computer Vision, Design, Detection, Localization, Mapping, Sensor, Swift, XCode**

1 INTRODUCTION

There are two key problems at hand that we have solved. When you clean a floor surface, there's no quantitative way to determine whether or not that area has been cleaned just by moving the vacuum over it; there's no objective metric for cleanliness beyond inspection with the human eye. The second problem at hand is that there's no way to guarantee complete coverage using human recollection of the area of the room you have traversed.

To address these two problems we are developing an AR assisted vacuum cleaning app with an accompanying sensor module to provide the user with real time feedback on surface cleanliness and net coverage of a room's surface area. This solution has the added benefit of addressing the ABET areas of physical and psychological health due to the improved coverage + cleanliness feedback reducing the quantity of allergens in a room along with the AR app creating a gamification of a positive habit leading to a cleaner space that is closely correlated with improved mental health benefits. The ECE areas that our solution will involve are primarily software systems with the development of our dirt detection CV module and our AR app with the secondary areas of signals for the CV and the Bluetooth connection from the Jetson Camera to the iPhone. We leveraged Apple's ARKit infrastructure that already contains end to end SLAM functions for map generation that are tightly integrated with its texture mapping for AR features.

2 USE-CASE REQUIREMENTS

Our system requirements together define the functioning product. In order to have an effective product, we have defined three main aspects: (1) coverage requirements relating to the augmented reality plane mapping, (2) cleanliness for our dirt detection algorithm, and (3) battery life for the Jetson to ensure that it will work for multiple vacuuming instances and over the course of our demo. We designed the system to be a vacuum mount in order to increase accessibility instead of creating a whole new apparatus. We developed our use case requirements from a desire to improve the vacuuming experience and quantitatively ensure its effectiveness. Ultimately, this system has positive effects on the public health of society by encouraging cleaner spaces and relieving the pain points of vacuuming, specifically the mental overhead required to track covered areas. By increasing the efficiency of this process, we also conserve vacuum battery life and reduce time to perform the task while still being thorough.

2.1 Coverage

We have divided coverage requirements into 2 sections: mapping and tracking coverage. Coverage refers to our augmented reality floor mapping overlay. Specifically, mapping pertains to our ability to use ARKit to initially detect the boundaries of a floor surface and how accurate the mapping is compared to the actual surface. The tracking coverage metric relates to our ability to erase the augmented reality floor overlay as it is passed over by our vacuuming device. This use case metric changed from the design document because we decided to forego the object removal part of the plane detection. This decision was made due to time constraints to focus on more core tracking and detection functionality, while also allowing us to use our system in more diverse, object-filled, environments. Since our patternless white floor is a constraint on the system, we did not want to limit the scope even more to only allow for certain types of objects.

2.1.1 Mapping Coverage

Mapping coverage involves boundary detection and plane classification. The AR app identifies and classifies a single floor surface with a rectangular bounding box and area meshing. Based on ARKit's plane inspection features and our initial proof-of-concept analysis where we mapped 10 different rooms, the use case requirement for floor mapping accuracy is measured by the distance between the furthest meshing point to the ground truth wall. This should be within ± 10 cm, and originates from ARKit's inherent accuracy in detecting planes and mapping surfaces. By experimenting with different devices (specifically the iPhone 14 Pro Max, the iPhone 13 Pro and the iPhone X which does not have the LiDAR scanner), we found there to be discrepancies between mapping abilities for the same surface.

2.1.2 Tracking Coverage

In addition to the initial floor mapping, our floor overlay has a memory component. More specifically, it is able to detect which parts of the surface have been covered by the vacuum, and erase parts of the overlay. The accuracy of erasing should be $\pm 10\%$ of the width height of the vacuum in order to accommodate the side areas of the vacuum head (5cm) that are edges and hence are not part of the suction. In order to be effective, the augmented reality overlay remembers the mappings that it created on a surface regardless of whether or not the area is in iPhone AR app camera view. That way, the tracking ability is completely independent of iPhone movements and the area that is in its field of vision. Upon return to a previously covered surface, the AR app will reflect the user's prior actions. This is relevant in the case where a user points the camera to a part of the room, partially vacuums that section of the room, then proceeds to change the camera's field of vision, and then returns to its original angle.

2.2 Cleanliness

Our cleanliness metric establishes the standard for surfaces that are considered clean and dirty. Once the vacuum has passed over a surface area and is in the field of view of our camera, the computer vision object detection algorithm will classify the surface as either clean or dirty. Our use case requirement defines a dirty surface to be one where dirt particles greater than 1mm are present on our white surface, as these would be detectable to the human eye from a five foot distance. The object detection algorithm should mark at least 90% of dirty surfaces as such.

We define an additional requirement stating that the object detection algorithm should propagate the data from the Jetson device to the iPhone within 6 seconds. Based on our initial tests, we believed that messages would be able to be sent at at around five second increments. This is also a reasonable latency for the system, as a five second delay is still manageable for a smooth user experience. Our initial product release was limited to sending messages over BLE every 4.88 seconds, which satisfied our use-case requirements, but was not ideal. We were able to obtain a large speedup, and now we are able to serialize messages every 102ms, on average, further detailed in Section 7.

2.3 Battery Life

In order for this system to enhance the user experience and effectiveness of vacuuming, we placed constraints on the battery life of our system, excluding the iPhone and the vacuum itself. The Jetson needs to be able to run for an extended amount of time for the sake of practical use and the demo, and so the Jetson's power source's battery life is important because it is being used for the dirt detection portion of our system. Hence, it must last for at least the duration of one vacuuming instance for the dirt detection to be impactful. In practical terms, the battery life of the Jetson's power source should be able to last multiple vacuuming instances, up to 4 hours without recharging. To satisfy this use case requirement, we have obtained a 20k+ maH wireless power source to power the Jetson.

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

In order to solve the problem outlined there are three key tasks that we worked on in parallel and then integrated together:

1. Mapping
2. Dirt Detection
3. Movement Tracking

3.1 ABET Considerations

By developing this using an iPhone rather than more costly solutions like the Apple Vision Pro and the Meta Quest which have entry costs that could be prohibitive to some users prioritising **accessibility and scalability**. Secondly we want to ensure that the technology is **human-centered** to empower people to take an active role and ownership in keeping their spaces clean.

3.2 System Decomposition

Our overall solution consists of 4 key subsystems. The first system is an active illumination module that will work to illuminate dirt particles to enhance the visibility and augment the computer vision component of the detection unit. This detection unit comprises a microprocessor to act as onboard compute on the camera input coupled with a Bluetooth module to facilitate communication with the iPhone running the AR app.

The AR app is the primary user facing subsystem: this app is be the user's window the the AR representation of their room and their source of cleaning feedback. The iPhone is mounted on the vacuum so that the camera view captures the view and the app draws coverage and classification segments on the screen. Our AR app is also running image detection in order to orient the coordinates of the camera's view in space. The image of the airplane is mounted on the front of the vacuum, and the user must keep it in the field of view at all times. The components for the AR system are primarily software based beyond the base hardware of the iPhone.

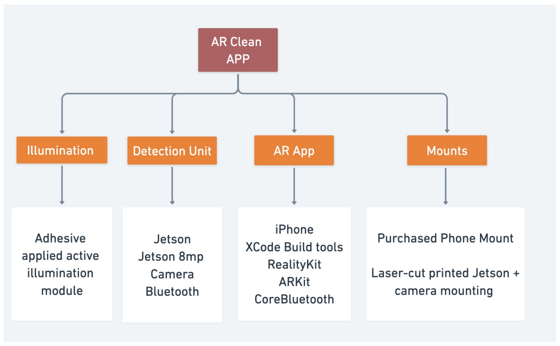


Figure 1: Overall system decomposition system-subsystem-components

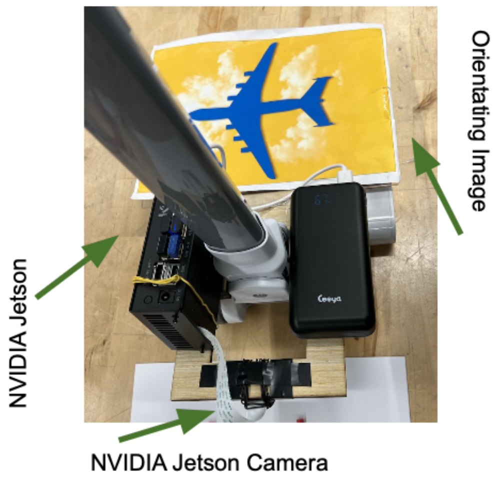


Figure 2: Mounted system configured for testing

The final subsystem is the mechanical component that involves mounting the phone, Jetson, and camera to the vacuum. We created a mount by laser cutting wood and acrylic. The mount that holds the Jetson and its power bank is a 2 inch rectangular border with a hollow inside. The inside of the rectangle is hollow and allows for vacuum handle movements by 60º backwards. This mounting system is modelled in Figure 2. Additionally, an acrylic plaque holds the reference image flat and above ground, in addition to active illumination fastened underneath.

3.3 Block Diagram

Fig. 12 outlines the high level operation of the overall system. There are two key sources of data from the real world that the system ingests: LiDAR data from the iPhone and image data from the Jetson camera. The image data frames are processed by OpenCV to detect dirt particles, and the result of this image processing is serialized into a message which is transmitted to the iPhone via Bluetooth, elaborated more in Section 6. On the iPhone's end, LiDAR, in conjunction with Depth and Disparity data from the cameras are being packaged into an AR Frame which updates the overall world map. In the app, we run plane detection, and then we overlay the cleanliness information we receive from the Jetson to update the texture that we project onto the AR World Map.

3.4 Hardware I/O

This architecture outlines the general I/O of the Jetson in order for it to perform its tasks defined by the broader block diagram and use-case requirements. It is supplied by a 20,000 mWah battery pack, and has a WiFi and Bluetooth dongle to facilitate development over ssh and allow for communication with the iPhone app. Input from the camera will be consumed by OpenCV to detect dirt particles.

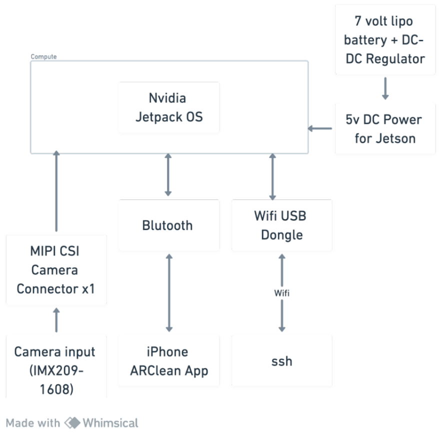


Figure 3: Hardware I/O Diagram of dirt sensing module

4 DESIGN REQUIREMENTS

Our design requirements originate from the chosen implementation technologies as detailed in Section 3. Based on our architecture and experimental testing, we were able to define bounds that account for error with our technical tools and human error while preserving the functionality and effectiveness of our system design. The below sections follow the structure of our use case requirements in Section 2, but elaborate on the technical limitations that governed the quantitative metrics.

4.1 Coverage

4.1.1 Mapping Coverage

The limiting factor on mapping accuracy is the accuracy of ARKit as a plane detection tool. We used our iPhones (13 Pro and 14 Pro Max) to quantify the error in ARKit's ability to detect a plane with ARKit's developer code for 'Tracking and Visualizing Plans', resulting in an accuracy error of ± 5cm [9]. We found that different models of iPhones have different versions of LiDAR sensor technology and camera qualities that affect the accuracy of the plane detection. Similarly, human error (angle and mapping strategy) is a factor in the quality of the output, adding an additional ±100% factor to this ARKit error. Ultimately, the initial mapping boundaries should be within ±10cm of the real floor plane, which is ±5cm error ±100%.

4.1.2 Tracking Coverage

The tracking coverage design requirement comes from the restrictions of our physical vacuum: there is a 5cm edge on either side of the suction, accounting for areas that may not actually be vacuumed and by extension, marked as either clean or dirty. The tracking algorithm response with (at most) a 0.5 second latency, ensuring a smooth up-to-date user experience. This latency arises from the movement tracking of the vacuum head, with the camera image being the input, processing this update in the backend, and then reflecting that movement in the augmented reality floor overlay that is created by ARKit. Details of the connections exist in Section 3, the system architecture block diagram.

4.2 Cleanliness

In the case of measuring cleanliness, we have defined a dirty surface as one which includes particles greater than one millimeter in diameter which are also visible by the human eye from a five foot distance. A passing state requires that at least 90% of these such surfaces are classified as dirty surfaces by our dirt detection algorithm. We discuss 5 how we decided to tune the dirt detection model to be more sensitive to small dirt particles by sharpening the image.

Our initial experiments showed that our algorithm was very susceptible to picking up noise. As we worked to tune that, we found that image preprocessing would be a valuable asset. Our design requirements were also heavily informed by the specifications of the camera we have chosen to use. We chose to use a Jetson 8 Megapixel Camera. This is because it was compatible with the Jetson Nano, as it came with a CSI cable, allowing for easy integration with the system we had, and it was incredibly small and lightweight, which would simplify the mounting process. The resolution of the Jetson's camera has allowed us to require the camera to detect 95% of dirt particles which were visible within a ten centimeter range of the camera's field of view. The 5% comes from the false negative error rate stated earlier.

4.3 Battery Life

We want to satisfy the non-functional requirement of being able to use our system for multiple vacuuming instances. The Jetson computing source can exist in two modes: a 5W mode and a 10W mode. On its idle mode, it consumes one watt of power. For a vacuuming instance, we disregard the idle mode, as it is not a limiting factor in our technical design requirements. This is because the Jetson should always be active whether or not the vacuum is running unless it is explicitly turned off. The 10W mode is the limiting factor, requiring approximately 16,000 maH versus the 5W mode requires approximately 8,000 maH, based on the specs from Jetson benchmarks. Since the Jetson operates using a 5W source, that is the mode which we will focus on. [14]

5 DESIGN TRADE STUDIES

Over the course of our development process, there were multiple engineering and design trade-offs that we faced. To name a few, we balanced image size and tracking functionality with user experience, opted for commercially standard mount over a custom-built one for practicality and integration speed, and enhanced dirt detection image processing at the risk of increased noise. Each of these choices impacted not only the technical performance but also the scope, timeline, and resource allocation of our project, as reflected in the Gantt chart.

As mentioned earlier, we made several key decisions when designing our system:

1. Image Size for Tracking Detection
2. Mount Design: Development Priorities
3. Dirt Detection Algorithm Tuning
4. Communication Protocol

5.1 Image Size for Tracking Detection

In our project, the trade-off in choosing a larger image size was essential to maintain the tracking functionality necessary for our application. We utilized image detection, which operates optimally with images of a minimum size of 8.5"x11". Smaller image sizes like 4"x6", were tested but presented similar challenges in consistent tracking. This is mainly due to the reduction in distinct asymmetrical features and the small percentage area that it consumes of the screen. The choice of image size was also influenced by the need for adequate color histogram compatibility (see Section 7. Minimally designed and more intuitive images could potentially compromise the ARKit's ability to distinguish the images from the background, especially in cases lacking sufficient color disparity or having mostly uniform shapes.

5.2 Mount Design: Development Priorities

For the mounting mechanism of our device, we faced a choice between designing the mount ourselves or finding a commercial phone mount that more easily integrates with our system. We considered using 3D printing and CAD modeling, which would have offered a more customized and potentially cost-effective solution at $0.50/gram. However, we ultimately decided to purchase a commercial iPhone mount in order to prioritize other core technical developments, like the augmented reality app tracking functionality and the Bluetooth connection. Since we wanted to test and integrate those immediately, we chose to purchase the mount. This trade-off, while sacrificing some customizability and potential elegance of integration, allowed us to focus on other critical aspects of our project by leveraging a ready-made solution that was inexpensive and reliable.

5.3 Dirt Detection Algorithm Tuning

To improve the tracking accuracy, we enhanced the input images from the Jetson CSI camera by sharpening them. In this section, false positives refers to non-dirty surfaces being classified as dirty and then false negatives refers to dirty surfaces being classified as clean. This process introduced some additional noise, which occasionally resulted in false positives in tracking scenarios. The sharpening was necessary because the original camera input did not provide enough detail during motion, primarily blurring the images, which would typically result in false negatives. The choice to enhance the images was decided to balance the accuracy of the tracking algorithm against the potential increase in noise. We agreed that it would be better for this use case to have a false positive than a false negative.

5.4 Communication Protocol

The two main communication protocols we were examining to transmit data from the Jetson to the iPhone were Bluetooth vs Websockets over WiFi. We opted for Bluetooth because of four main reasons:

1. **Proximity** the Jetson will be in close proximity to the phone which is within the range of Bluetooth's operating range [7]
2. **Bandwidth** Our data that we are transmitting is not very large in terms of file size since it is serialized text which is in line with the 1mbps speed of Bluetooth [12]
3. **Power efficiency** Bluetooth is more power efficient than WiFi which reduces strain on the battery requirement [4]
4. **Ease of Setup** Bluetooth is designed for direct to device transfer, it has fewer layers of the network stack to go through in comparison to WiFi [13]

However, Bluetooth did not work because of the incompatibility between Apple and non-Apple products. Instead, we used Bluetooth Low Energy (BLE) which is not subject to the same requirement, but has an additional constraint that each packet size is only 24 bytes. Our serialized data messages were designed to be under 24 bytes each, containing the most minimal amount of information possible: the UNIX time stamp and a binary classification of dirty or clean.

6 SYSTEM IMPLEMENTATION

There are two main subsystems in our larger system, (1) the augmented reality floor mapping and (2) the dirt detection.

6.1 Software Stack

Figure 5 outlines our development stack consisting of Swift for the AR application, Python for the OpenCV with both languages leveraging Bluetooth APIs.

Figure 6 outlines the tasks during operation of our system. There's a continual Bluetooth connection established between iPhone and Jetson with the Jetson continually processing the video-stream and sending its cleanliness updates to the iPhone app which is running the concurrent tasks to build and update an AR World Map. [9] [3]

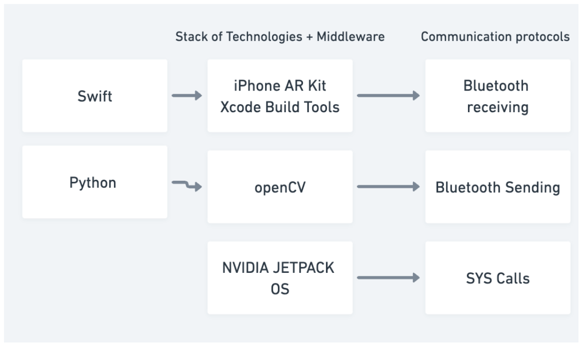


Figure 4: Overall software stack and technologies

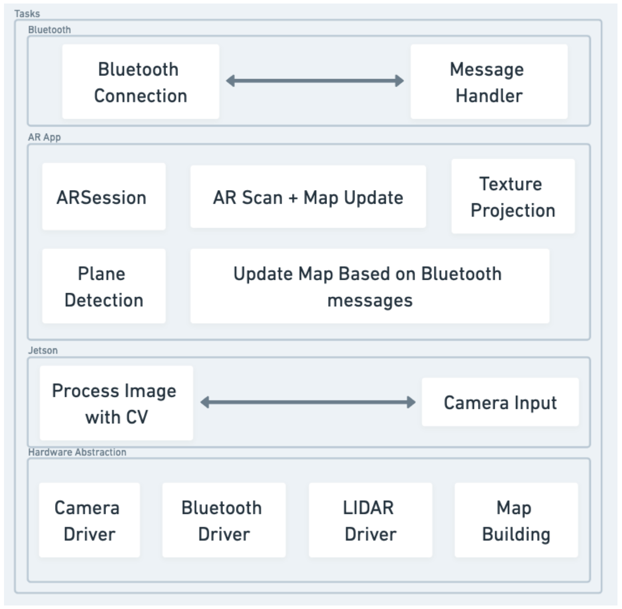


Figure 5: Task groupings for parallelized distribution of work

6.2 SceneKit

SceneKit was a new learning experience for our team as we had never worked extensively with 3D graphics libraries before let alone one in AR. We used these throughout the development process and got more familiar with them over the course of the semester. SceneKit follows a tree node structure for objects in the world with all nodes being relatives of a root node that's instantiated upon starting the app.

SceneKit followed the Model-View-Controller pipeline and allowed us to render what we wanted by providing renderers with different schemas:

- `override func viewDidLoad()`: function that was called once the app was started and the view was created allowing us to initialize our objects and specify materials for the lines we were drawing (these materials had characteristics such as color and transparency along with exposing attributes to let us control their rendering order.
- `func renderer SCNSceneRenderer didAdd SCNNode for ARAnchor`: this renderer ran every time a new object was added to the scene tied to a specific anchor in the world, for example a plane anchor or an image anchor. This renderer was where we placed our image detection logic and where we would identify and extract information about the floor plane upon it being added to the scene.
- `func renderer SCNSceneRenderer didUpdate SCNNode for ARAnchor`: this renderer was closely linked with our plane logic this renderer fired every time an existing node tied to an anchor (i.e. our plane) was updated and this is where we controlled the logic that allows the user to freeze the plane upon it being mapped to their satisfaction to define the area they are going to be cleaning over.
- `func renderer SCNSceneRenderer updateAtTime TimeInterval`: This renderer fired every frame of the view. This was where our logic for our boundary check and drawing the tracking line of our vacuum moving in the world.

Figure 13 (in the Appendix) is an image of the SceneView hierarchy. Every node we add to the scene is a descendant of the root node.The 4 key nodes for our application are outlined in the tree with the floor plane and ARImage anchor representing the nodes for the floor plane and tracking image respectively. SphereNode and OffsetNode (all nodes in SceneKit do this) encode transformation matrices and explicit offsets defined with respect to the tracking image, these offsets and transformations enabled us to draw tracking lines accurately behind the vacuum head and map the view of the Jetson camera into its corresponding location in the AR world scene.

6.3 Augmented Reality Floor Overlay: Plane Detection

ARKit allows for robust and continuous detection of horizontal planes by utilizing the device's camera feed and built-in motion sensors to analyze the scene presented, leveraging LiDAR sensors and SLAM [3]. Once a horizontal surface is detected, an anchor was created to that surface, was then manipulated within SceneKit. We configured ARKit to optimize its detection for larger horizontal planes to better suit our application's focus on floor detection, adjusting the 'planeDetection' configuration to 'horizontal'.

With the detected planes anchored, we used SceneKit to visually represent these areas within the AR environment, enabling us to create overlays (a SCNPlane) on the detected floor. This visual representation is crucial for user interaction, allowing users to see exactly which areas of the real world the system has mapped.

We added features like the ability to freeze and unfreeze a mapped floor, which essentially means we are constraining when new child nodes can be added to the SCNPlane at the tap of a button.

6.4 Augmented Reality Tracking

With ARKit, we are able to track world coordinates of the iPhone's camera. We used image detection (pictured below as a blue airplane) in order to orient ourselves around a point that has a fixed distance to the rear-mounted Jetson camera. We couldn't just use the iPhone coordinates because it is mounted on a moving handle. We tracked the head of the vacuum with the airplane picture, discussed further in Section 7.

Figure 6 shows a real-time data for the blue coverage line that is being drawn through a SCNLine module [10], which allowed us to render this path dynamically within the AR space. The red segments are indication of the passed-over surface being marked as dirty. The green dot represents the center of the vacuum, while the back red dot represents the camera's current view. Figure 7 depicts a moment before the iPhone has received the dirt classification of the current camera view, but after it classified the previous segments as dirty.

The color of the path segments drawn by SCNLine was dictated by the input received from the vacuum's dirt detection system and processed by the Jetson script. When the dirt detection system identifies the existence of debris, BLE sends a new classification to create a new red path segment, indicating areas needing further cleaning. Conversely, when no dirt was detected, the path segments remained green, signaling that the area was clean. This color-coding provided an intuitive and immediate visual cue about cleaning performance and areas requiring additional attention.

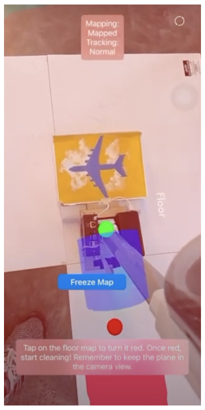

Mapping:
Mapped
Tracking:
Normal
Floor
Freeze Map
Tap on the floor map to turn it red. Once red, start cleaning! Remember to keep the plane in the camera view.

Figure 6: Vacuum tracking lines: blue is for coverage and red is for dirt classification

6.5 Dirt Detection

6.5.1 Dirt Detection Implementation

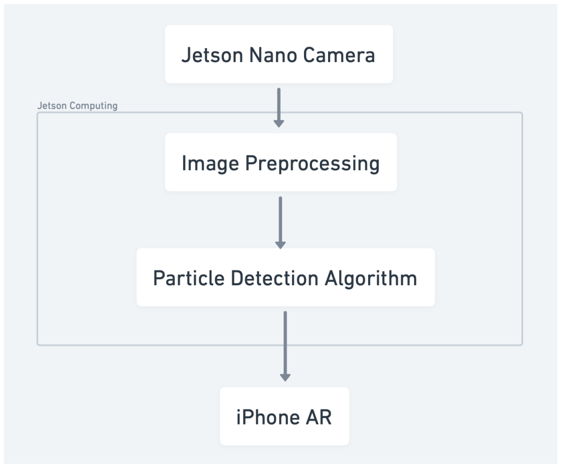


Figure 7: Dirt detection components

Dirt detection is the main component driving our cleanliness use-case requirement, as specified in Section 2.2. This consists of a camera component, a computer vision algorithm, and a means of serializing the data from the Jetson module to the iPhone in a format which can then be processed for the AR application.

The camera input and the Jetson module are easily integratable because the camera that we are using was manufactured for the Jetson Nano. As such, we simply connect the two devices via the given cable (or an extender, if we opt for that option), and are then able to receive image data which will be processed by Python code.

The CV component of this system follows the following format:

```
frame ← input_image
if frame = None then
    error()
else
    frame ← crop_image(frame)
    frame ← grayscale(frame)
    frame ← sharpen(frame)
    edge_detect = canny_edge_detection(frame)
    number_contours ← find_contours(edge_detect)
    return number_contours
end if
```

We drew heavy inspiration from an OpenCV blog post [11] in creating our initial particle detection (find_particles) algorithm. We started with a series of hand-tuned preprocessing steps which then piped the image into a series of OpenCV's existing functions. We discovered that in practice, the system behavior deviated greatly from what we initially planned our test cases out to be. In the primary iterations of our dirt detection algorithm, we were blurring the received images prior to running a contour detection algorithm. While the performance of this older algorithm was fine on static images taken by an iPhone camera, the results produced in our use-case setting were less than satisfactory.

We have also integrated an active illumination device to aid our dirt detection process. The purpose of adding this component to our subsystem was to enhance the performance of our dirt detection script. A small dirt particle might seem incredibly insignificant when initially run through the computer vision algorithm, and there is a possibility that our system will not be able to catch that dirt particle. However, with the addition of our active illumination, the light provided by our LED causes the small dirt particle to cast a shadow behind it. As a result, the dirt detection algorithm now sees a much larger object within the camera view, allowing us to more easily detect unclean particulates.

At the time of creating the initial pass of our algorithm, we failed to consider the fact that the Jetson's CSI camera would be capturing data frames as the entire system was in motion. The quality of our mounted camera solution was not only significantly worse than the output from an iPhone's rear view camera—the CSI camera also had a fisheye-like warp on the images. Due to these changes, we decided to crop each input photo to account for significant warp on the edges of the image frame, as well as to limit the camera view to only fall within the scope of the flooring which the vacuum has already cleaned. Beyond this, we also chose to lightly sharpen the camera image. This choice was due to the fact that the input images were at times slightly blurry, since the system is in constant motion when vacuuming. We tuned the sharpening threshold values to enhance the images just enough to amplify dirt particles which were already there, but not too much to the point where it starts introducing noise which we could have avoided.

Below is an example of the thresholding we have performed on a sample image.

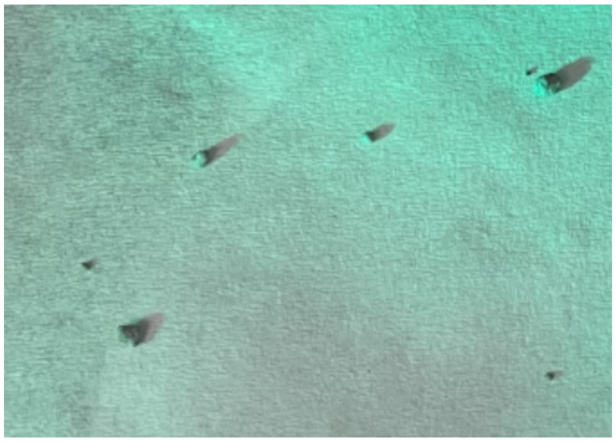


Figure 8: Test Dirt Image (before)

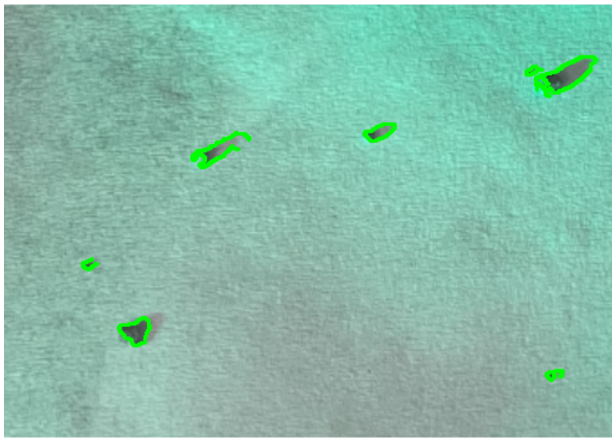


Figure 9: Test Dirt Image (after)

6.5.2 Bluetooth Data Transfer Implementation

As for our data transfer and communication process, we wanted to serialize the outputs of our Python scripts and send them to the iPhone over a Bluetooth connection. Unfortunately, due to security constraints on the iPhone placed by Apple's system, we were unable to get traditional Bluetooth pairing and transmission to work with our system. This caused us to pivot to BLE, or Bluetooth Low Energy as our communication protocol.

Our system still requires that a Bluetooth dongle must be connected to the Jetson device. However, this Bluetooth adapter must also be compatible with Bluetooth Low Energy. We opted for Kinivo's BTD400 Receiver, which satisfied our system requirements.

Once the dirt detection script finishes running, we will have two values to send to the iPhone: a UNIX timestamp, as well as an integer value, denoting the number of particles that our algorithm was able to detect. After serializing this as a string, we need to send a message from the Jetson Nano to the iPhone.

Our script uses the Python package `bless`. We took heavy inspiration from an online source [8] which set up a Python script for a device to act as a peripheral. We set up our Jetson as a peripheral (server) device and our iPhone as a central (client) device in order for data transfer to work. The server side script was hosted on the Jetson using Python, and the client side logic was pulled from Apple's native CoreBluetoothLE example [1]. When the Python script is running, the Jetson is able to broadcast itself to all nearby devices which support BLE. Once the iPhone detects the peripheral, the Jetson can send a pairing request to the iPhone. If the user clicks accept, the BLE connection is established, and we are then able to send messages wirelessly from the Jetson to the iPhone. In fact, we simply send our timestamp and integer value from the Jetson to the iPhone as a string. The reason why we chose to send just one integer value was due to a BLE constraint; we are only able to send 24B per message. While our initial design involved serializing the entire image and running more robust algorithms on the iPhone side, we discovered that this would not be possible given the limitations on BLE. In fact, we had just enough data to be able to send a timestamp and the contour count value to from the Jetson to the iPhone.

One significant hurdle that we had to overcome was the fact that our dirt detection script and our BLE transmission script ran on different Python versions. We were unable to resolve certain package dependencies, and so we needed to run two versions of Python concurrently. This caused issues within our system because the dirt detection algorithm relied on Python3.6, while we were using Python3.8 for BLE, but the two processes needed to happen concurrently, and we needed to query results from the dirt detection script to use for BLE transmission. Our solution was to use the Python `subprocess` module.

In order to run both version of Python simultaneously, we ran the dirt detection script as a subprocess from within the BLE connection script. We recognize that this is not the best approach if we were to productionize our product, but in a pinch, this was the cleanest and simplest solution we were able to come up with, as we were unable to circumvent the blocking dependency issues.

Within the dirt detection script, we write the output values to a standard output stream. The BLE script then queries the resulting output from the dirt detection script, and communicates those values to the iPhone. Our initial implementation involved relaunching the dirt detection script every time we wanted to query a new value. Unfortunately, this process was incredibly slow. After some benchmarking, we found that the bottleneck of our script was the delay caused by waiting for the Jetson's camera to open and be stored as an object within the dirt detection script. This incurred around a two second delay, which was sufficient to pass our use-case requirement for latency, but highly suboptimal. To navigate around this issue, we decided to use the `Popen` interface from the `subprocess` module. This allowed us to keep the dirt detection script running continuously in the background while simultaneously running the BLE script. Now, to query a result from the dirt detection script, the BLE script simply sends a character using the PIPE functionality. This improved our message transmission rate to send once every 102ms, on average, giving us a huge speedup from an average 2.1s delay.

6.5.3 Dirt Detection & AR Integration

On the iPhone's side, we created a queue which maps timestamps to 3D coordinates. This data structure, which we call "timeQueue", logs the position of the Jetson's camera at a 100 millisecond interval, which is well under our outlined requirement of a six second latency. Once a message from the Jetson is received by the iPhone, we check to see if the integer value received was within the threshold value we set for an image to classified as "dirty". This truth value is stored as a boolean value, "isDirty", and we use this value to determine what color block we will render on the AR application. We iterate through our timeQueue and we look for a timestamp which matches the time we received from the Jetson. If we find a match, a red or green block is drawn on the AR app. The color is dependent on the boolean value we receive. In this case, green denotes "clean", and red denotes "dirty".

The reason why we choose to use the timestamp (in particular, the UNIX timestamp) is because the rear-facing Jetson camera has no information about its position in a 3D context. In contrast, that is information that we have and require on the AR side. However, we know that the Jetson camera is always a fixed distance away from the vacuum head, which we are tracking (as specified by section 6.4). The most elegant solution we came up with was to create an offset between the location where we find the vacuum head and where the camera view is. We measured this by creating a bounding box for the corners of the camera view and measuring accordingly. This offset value is what is being stored in our timeQueue, along with the time at which the position was logged. This way, we are able to match the information we receive from the Jetson into something understandable in a 3D space.

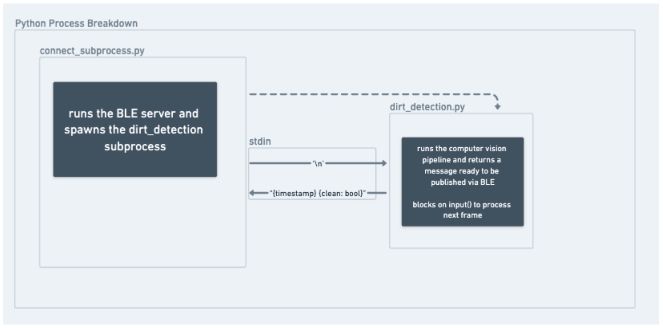

Python Process Breakdown
connect_subprocess.py
runs the BLE server and spawns the dirt_detection subprocess
dirt_detection.py
stdin
'\n'
"{timestamp} {clean: bool}"
runs the computer vision pipeline and returns a message ready to be published via BLE
blocks on input() to process next frame

Figure 10: Python Process Breakdown

This diagram outlines the subprocess architecture we had to employ in order to get both our computer vision component and our Bluetooth functionality to run at the same time. This subprocess approach was necessary due to version dependencies. The computer vision component ran on python3.6 on a custom installation built on the Jetson that contained GStreamer which enabled connectivity to the camera. However, for the Bluetooth Low Energy package it depended on `asyncio` which is a Python builtin only available at version 3.8 and above. Initially we ran a subprocessing script that opened a connection to the camera, returned the timestamped cleanliness message and then close the camera, but upon profiling its performance we noticed that opening and closing the camera was a very time consuming process so we opted to pivot our subprocess design t allow for the child computer vision process to be persistent. We achieved this by piping stdout from the child node into the parent node this connection allowed for the computer vision node to receive input from the connect_subprocess script. Thus by blocking in the child process until receiving a new line from the parent we could query the output of the computer vision pipeline without needing to explicitly establish a gstreamer connection every single time. This led to a 33x speedup in our latency.

7 TEST & VALIDATION

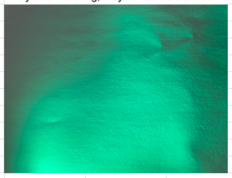
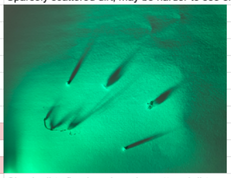
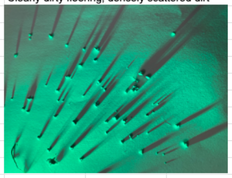
This section covers some of the tests we are running to ensure our system meets our use-case requirements, as specified in Section 2.

7.1 Dirt Detection

Since it was be incredibly difficult to quantify the performance of our dirt detection algorithm in real life, we have opted to use simulation environments to calibrate the level of accuracy our algorithm is achieving. We have chosen to create three different configurations of dirt on white, pattern-less flooring, which we will test our algorithm against (with the active illumination light on). The "dirt" that we use for our testing configurations aligns with the constraints mentioned in our use-case requirements—each speck will be at least one millimeter in diameter and should be visible to the human eye from a five-foot distance. The input for this test are the images that the Jetson camera receives when pointed at the test dirt configurations. These images are then sent to the actual Jetson module, which runs our dirt detection Python script. The output image would be thresholded into a numeric output, representing the number of irregular contours the CV algorithm picked up on.

A passing test output should successfully classify at least 97% of our given dirt particles as to be dirt. We are not looking at the entire area of the dirty particles to be classified; rather, we are just looking for the existence of a cluster which is identified to be "dirty". Beyond this, the detection algorithm should only have a 5% false positive rate. This is measured by the number of pixels the algorithm determines to be part of a dirt particle which do not match the ground truth. For this all to be set up, we must first pre-process the inputs from the Jetson camera. We must know where the dirt is, and how much of the overall pixel space the dirt occupies to be able to accurately retrieve the metrics from these test evaluations.

Below is an example of the tests that we ran, with an example image of each category of dirt.

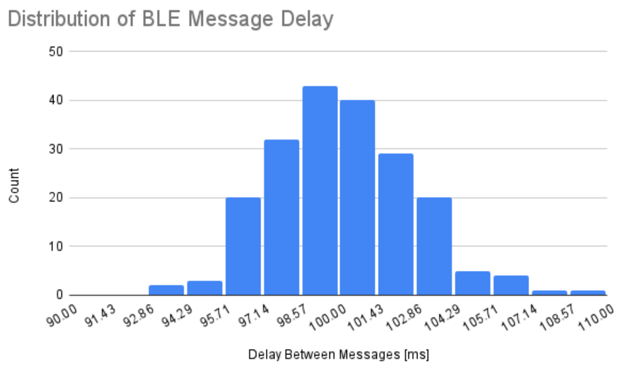
Image	Algorithm 1	Algorithm 2	
Clean			Truly clean flooring, may have a little shadow because of variance in lighting
clean_1	TRUE	FALSE	
clean_2	TRUE	FALSE	
clean_3	FALSE	FALSE	
clean_4	TRUE	FALSE	
clean_5	TRUE	FALSE	
clean_6	TRUE	FALSE	
clean_7	TRUE	FALSE	
clean_8	TRUE	FALSE	
clean_9	FALSE	FALSE	
clean_10	TRUE	FALSE	
clean_11	TRUE	FALSE	
Dirty (1)			Sparsely scattered dirt, may be harder to see on the camera
dirty-1_1	TRUE	TRUE	
dirty-1_2	TRUE	TRUE	
dirty-1_3	TRUE	TRUE	
dirty-1_4	TRUE	TRUE	
dirty-1_5	TRUE	TRUE	
dirty-1_6	TRUE	TRUE	
dirty-1_7	TRUE	TRUE	
dirty-1_8	TRUE	FALSE	
dirty-1_9	TRUE	TRUE	
dirty-1_10	TRUE	FALSE	
Dirty (2)			Clearly dirty flooring, densely scattered dirt
dirty-2_1	TRUE	TRUE	
dirty-2_2	TRUE	TRUE	
dirty-2_3	TRUE	TRUE	
dirty-2_4	TRUE	TRUE	
dirty-2_5	TRUE	TRUE	
dirty-2_6	TRUE	TRUE	
dirty-2_7	TRUE	TRUE	
dirty-2_8	TRUE	TRUE	
dirty-2_9	TRUE	TRUE	
dirty-2_10	TRUE	TRUE	
dirty-2_11	TRUE	TRUE	
FPV	81.82%	0.00%	
FNV	0.00%	9.52%	
ACC	71.88%	93.75%	

7.2 Data Processing & Propagation Delay

We want to ensure that the user is receiving their data within some reasonable time frame. As such, this test aims to threshold the amount of time it takes for dirty flooring to be displayed on the AR platform (in this case, the iPhone). To test the amount of time this data processing and serialization would take, we marked the timestamps when the image is captured from the Jetson camera. The Jetson runs the dirt detection algorithm and then sends the outputs, along with the initial timestamps, in a serialized format to the iPhone. We then record the timestamp when the iPhone receives the data from the Jetson. The frames denoting "dirty" or "clean" spots do not have to be rendered; rather, they simply need to be ready to be displayed on the iPhone screen. At this point, we can simply subtract the final timestamp (on the iPhone side) from the initial timestamp (sent from the Jetson) to find the overall delay due to processing and serialization. We note that since there is no perfect synchronization of the clocks between the Jetson and then iPhone. We are relying on the UNIX timestamp, and this is data which is queried off the internet using WiFi. As such, we may have introduced a small amount of error when performing this calculation. Another source of error is the latency between the Jetson camera and the actual computing source, as well as its corresponding serialization delay. We are not accounting for this here and assume it to be negligible for the sake of this test metric.

We verify the success of this test by looking at the difference between the two aforementioned timestamps. For clarification, we used the UNIX timestamp (`time.time()` from the Python `time` module, `timeIntervalSince1970` in Swift) as our time metric. Any time difference less than six seconds was considered a successful output. The results of this test help us maintain our use-case requirements for tracking coverage, as we stated that we would like for all the dirt to be accurately detected and interpreted by the iPhone within a six second time frame.

While our use-case requirements stated that we needed all messages to be received within a six second time frame, we were able to reduce this delay to approximately 102ms, on average. To obtain this value, we ran the test (as described above, subtracting timestamps) over two hundred times, collecting a vast number of data samples. We noticed a small amount of spread in the data, but this could be attributed to the processing time of the dirt detection algorithm. In addition, this variance in delay time is negligible, compared to both the value of the delay and our use-case requirement thresholds. The following chart shows the distribution of values we got for the message delay:



7.3 Jetson Camera Mount

Our design includes a rear-facing camera whose primary purpose is to capture the state of the floor after the vacuum passes over it. The images from this camera will be used to determine the cleanliness of the floor and will be the input that we use for our dirt detection algorithm. To test where we want to mount the camera, we set up a series of testing configurations and compared the results. We tested three different camera heights, and for each height configuration, we will examine how the camera performs at eight different angles (between 0° and 70°, inclusive, at 10° increments).

To quantify how well each configuration performs, we included three different testing configurations of dirt, all utilizing the active illumination lights. We set up the following configurations in our test:

- Clean: No dirt
- Sparse: Fewer than five dirt particles spaced at least one centimeter apart
- Dense: At least twenty dirt particles scattered within the camera frame

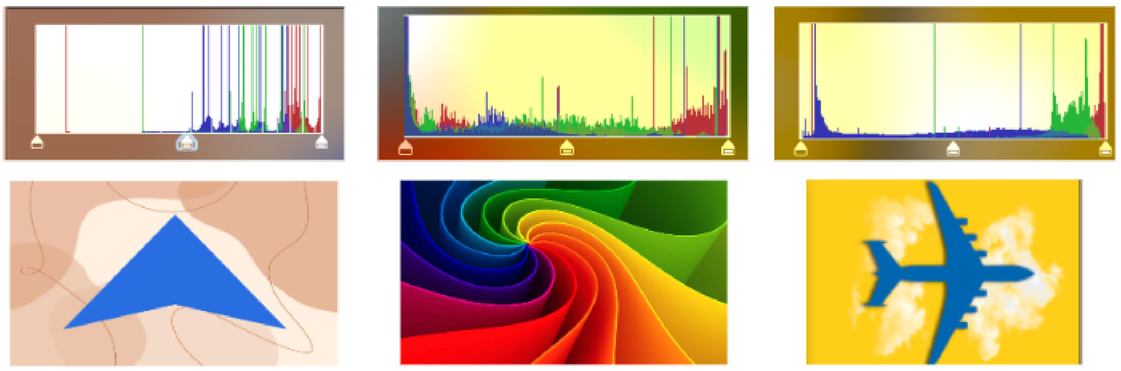
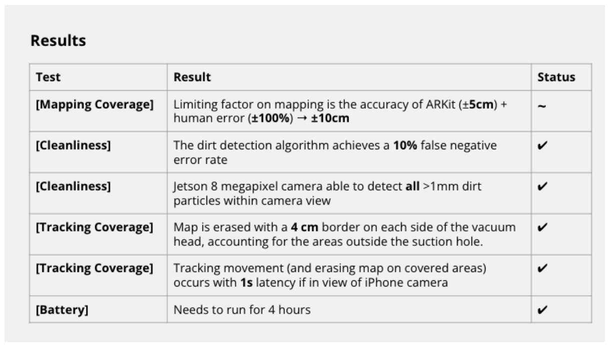


Figure 11: Color histograms of different detection images

We ran the tests in the daytime, as with all of the rest of our dirt detection tests. We used either man-made illumination methods (lights), or we made use of natural lighting. This did not appear to change the outcome of our testing results. We also did not change the configuration of our active illumination LEDs, and they were constant throughout all tests. In addition, the dirt detection algorithm was also held constant across all testing configurations. For each of the twenty-four camera angles, we took pictures of both configurations of dirt using the Jetson camera and then ran them through our dirt detection algorithm.



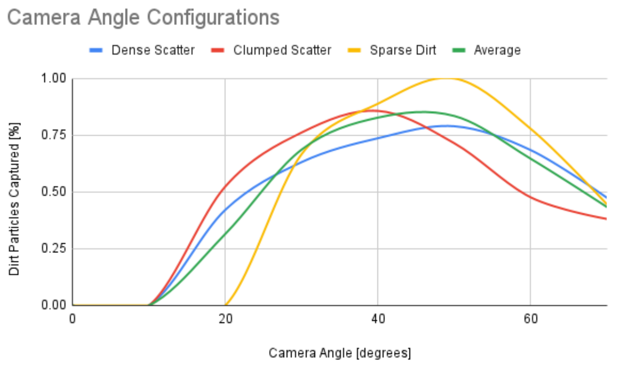
Results

Test	Result	Status
[Mapping Coverage]	Limiting factor on mapping is the accuracy of ARKit (**±5cm**) + human error (**±100%**) → **±10cm**	~
[Cleanliness]	The dirt detection algorithm achieves a **10%** false negative error rate	✔
[Cleanliness]	Jetson 8 megapixel camera able to detect **all** >1mm dirt particles within camera view	✔
[Tracking Coverage]	Map is erased with a **4 cm** border on each side of the vacuum head, accounting for the areas outside the suction hole.	✔
[Tracking Coverage]	Tracking movement (and erasing map on covered areas) occurs with **1s** latency if in view of iPhone camera	✔
[Battery]	Needs to run for 4 hours	✔

Results from each of our tests, showing passing results, with the maping coverage test marked as a tilde because it is less performant on reflective surfaces, like the TechSpark flooring. However, our use case is patternless white flooring so this is less of a concern.

Firstly, we considered how much area the camera angle is able to cover. The rear facing camera should be able to capture the area which the vacuum has just passed over, and thus the resulting image should contain all the dirt particles which we placed in our test environment. We further quantified the performance of our camera angles by the same metrics which were described in our dirt detection test. In this test, we compared the results relatively; we checked to see how many of the dirt particles were detected and we checked the false positive rate. The outputs of this test follow a ranking format. This helps contribute to the cleanliness use-case requirement (Section 2.2) as it is another method we are using to improve our dirt detection results.

The following chart is shows the results that we received based on our test suite. Based on these results, we ended up choosing to mount the camera at a 45° angle.

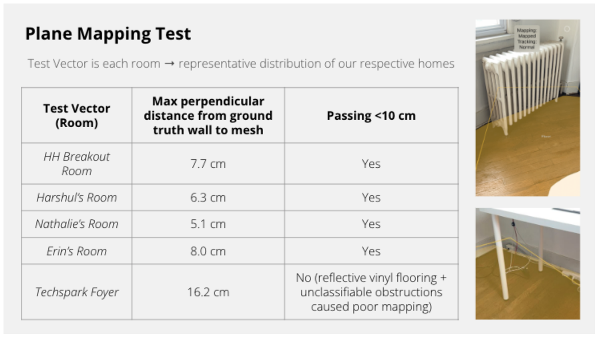


7.4 Image Tracking

For the image tracking functionality in our project, we conducted tests using a variety of images that featured different color histograms to determine how well the system could detect and track these images from a fixed mounting distance of 4 feet, as seen in Figure 11. The criterion for passing these tests was the system's ability to consistently recognize and follow the image despite the variations in color and pattern. During our tests, each image vector was processed through our AR application to assess detectability and tracking stability. The results showed that while the system managed to track most images effectively, compromises had to be made between the clarity and specificity of the images and the robustness of tracking. This was evident as some images with unbalanced color histograms were not accepted by the tracking algorithm or demonstrated poor performance.

7.5 Accuracy of Mapping Overlay

In the testing of mapping overlay functionality, we tested from a 4 ft height for the iPhone to examine how well our system could overlay mapped data onto the physical environment. The test focused on determining the accuracy of meshing AR overlay with the actual room layout. For a test to pass, the mapped overlay needed to align closely with the real-world boundaries of the room, measured against a ground truth wall distance within 10 cm at its furthest point. The system performed better in environments with matte floors and fewer reflective surfaces (which is why we had chosen patternless white floor as our constraint).

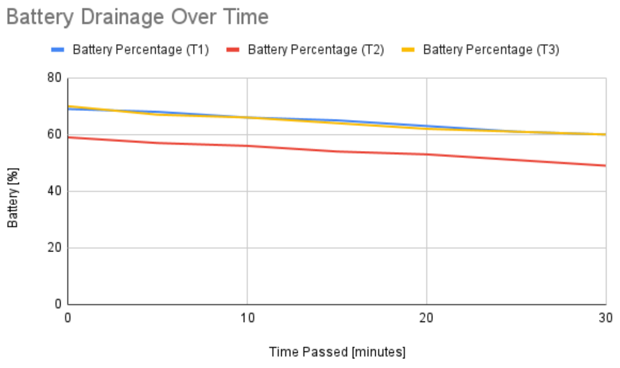

Plane Mapping Test

Test Vector is each room → representative distribution of our respective homes

Test Vector (Room)	Max perpendicular distance from ground truth wall to mesh	Passing <10 cm
HH Breakout Room	7.7 cm	Yes
Harshul's Room	6.3 cm	Yes
Nathalie's Room	5.1 cm	Yes
Erin's Room	8.0 cm	Yes
Techspark Foyer	16.2 cm	No (reflective vinyl flooring + unclassifiable obstructions caused poor mapping)

7.6 Latency of Mapping Update

Our testing for the mapping update functionality involved assessing the system's responsiveness and accuracy in updating the mapped area in under 6 seconds from detection to appearance for the classification subsystem. The other update latency was for the tracking of the vacuum head, which had a requirement of being under 1 second. Both these tests were passed by our system latency of less than 1s for tracking coverage and less than 6s for dirt updates. We had to make changes to our initial implementation to pass these tests: our initial idea of using a hit test to draw lines revealed latency issues, prompting us to explore options, where we learnt about SCNLine module in ARKit [10]. After tuning our dirt detection scripts and resolving how we deal with incompatibilities between Python versions, we achieved consistent latency reductions in Bluetooth communication classification updates. This kept tracking coverage consistently below one second, far below our threshold. With these tests and requirements, we ensured that all mapping updates were reflected in the app quickly, enhancing the user experience.



7.7 Battery Life

To test the battery life performance of the system, we logged the charge status of the Jetson's power source at the beginning of each test, and we then ran the entire end-to-end system for ten minutes. After the ten minutes were up, we checked the charge on the power bank. A charge depletion of less than 4.15% indicated a successful output. This was thresholded by the runtime of the vacuum we are using. By design, it has a runtime of up to 44 minutes. We are assuming four separate vacuuming cycles and we are rounding up the time of one individual vacuuming instance to one hour. We are not considering the battery performance of the iPhone as a bottlenecking factor, as an iPhone in good battery health should easily run for at least an hour, even with heavy use. This test relates directly to the use-case requirement specified in Section 2.3.

We ran this test four times, but not back-to-back. This helped us reduce the variance in our data a little bit, thus giving us a little more confidence about the values we were receiving. Our tests confirmed that the power supply we chose was more than capable of holding up to our use-case requirements. The following diagram shows the results from our tests.

8 PROJECT MANAGEMENT

8.1 Schedule

Our schedule is shown in Fig. 14. There were no updates from the design report until now, so the allocation and timelines remain the same.

8.2 Team Member Responsibilities

The distribution of responsibilities (with respect to who is the owner of the task) is color coded in Fig. 14. Specifically, Erin worked closest with the dirt detection, BLE/Bluetooth connection, and associating the Jetson's output to a logical value in the AR space. Nathalie focused on plane mapping, SCNLine tracking and coloration alongside Harshul, who defined the bounding box, projected world coordinate vectors to orient ourselves in time and space, as well as ran point on the image detection implementation.

9 RISK MANAGEMENT

We encountered several unanticipated challenges when working on our project, which arose particularly when we began integrating the system together.

9.1 Risks and Mitigation

9.1.1 Bluetooth Communication Alternative

Through the development of our project we worked on developing a Bluetooth Low Energy central/peripheral using the cross-platform Python library bless. While getting this mock-up to work was relatively unhindered by hardware when testing Bluetooth from a Mac to iPhone, we ran into numerous issues getting this same connection to be established on the Jetson due to Linux kernel and driver dependencies.

To hedge against this risk as the time allocated to this task drew to a close on our Gantt chart, we set an internal deadline ahead of the scheduled deadline upon which we would pivot to using an Arduino with a BLE shield. The Arduino's specialized hardware and simplified operating system architecture would circumvent the OS issues we were running into with the Jetson's kernel. We had also confirmed online that people have successfully implemented a BLE connection between an Arduino and an iPhone, this was not the case for the Jetson. If we were unable to get the Jetson's BLE working, we would purchase the Arduino and its accompanying BLE shield. Then our contingency plan was to control and connect to it from the Jetson via serial communication over USB (an interface for serial communication exists in Python with the package pyserial). This would minimize the design changes required on the software side and prevent us being blocked for too long troubleshooting JetpackOS.

9.1.2 Modularity and Adaptability

We had delayed mounting everything in order to prioritize working on our subsystems, but we pivoted to focus on integration due to the high potential that things would go awry. Since not all subsystems were complete, this involved careful consideration of materials and techniques to ensure stability and functionality. We made sure to discuss the components that we may want to adjust or remove later, differentiating between permanent fixation with epoxy and the components suitable for temporary attachment using hot glue and duct tape. For example, we wanted the portable charger to be removable in case we need to charge or replace it, but used epoxy for the laser cut wood pieces to solidify them together. By selecting appropriate materials and techniques for each component, we have a solid mounting of components that meets the demands of our application.

9.1.3 High Latency System

The first time that we sent and received messages, it took between 30 and 50 seconds for a message to propagate from the Jetson to the iPhone. This is far too long to actually make anything practical or usable, and we thought it was the Bluetooth speeding everything down. While working on trying to speed up both the dirt detection and the speed of the Bluetooth communication, we thought of ways that we could refactor code to lower the latency. This includes running background processes so that there could be multiple processes running simultaneously. However this was a sub optimal option, as it would involve a significant refactor. We also checked whether our Jetson Nano was actually making use of the CUDA architecture that was on board, but it seemed like the hardware acceleration was not being integrated. The next step we considered taking would be to rebuild with CUDA acceleration, but we ended up finding a solution without needing to complete this step.

10 ETHICAL ISSUES

Our augmented reality vacuuming system generated a substantial amount of waste product. This includes the plastic used to make up the vacuum, which often has significant environmental costs, especially at mass production. One way to prevent this (at scale) would be to partner with vacuum production companies that sell vacuums made out of recycled plastic. Our project aims to help public health by providing a cleaning tool, but users must always be environmentally conscious as vacuuming does not provide a sterile environment; it simply removes dirt particles.

The physical design of our augmented reality vacuuming system also brings up important safety considerations, especially regarding children. Since there are sharp corners on the Jetson computing source and the attachments that we incorporated include certain modular hardware components, we must be considerate of what would happen if a system component got dislodged. For example, the camera cap of the Jetson could be a choking hazard as well as the attached cords since they can be removed.

Using the iPhone camera technology in an AR vacuuming system can raise potential privacy concerns. Our device uses iPhone and Jetson cameras as well as LiDAR sensors to navigate and map the home, which could be dangerous if unauthorized data is collected and leaked. To make the system more robust, there could be added security in the Jetson message transmission as well as the visual mapping that our AR app remembers of the room in question.

The method that we used to run both the dirt detection script and the BLE script is highly insecure. We opted to use Python subprocesses, and although this worked for our situation, our code is not scalable or productionizable. If we were to push this product to market, we would definitely need to find a cleaner way to resolve the dependencies which were creating blockers.

11 RELATED WORK

We were inspired by the capabilities of the recent release of the Apple Vision Pro. There are video examples of a space vacuuming experiment [5], but importantly rely on access to an Apple Vision Pro, which costs $3,500 pre-tax [2]. Developed by Daniel Beauchamp, a Principal AR/VR Engineer at Shopify, mixed reality provides an interesting avenue for gamification of mundane tasks, addressing normal pain points of every day [6]. We wanted to create a version of this that is financially accessible by adding supplemental components to an existing vacuum. In addition, we wanted to add a cleanliness measure to address public health concerns. Our final product is different than existing prototypes by creating a solution that addresses both coverage and cleanliness.

11.1 Future Work

11.1.1 Communication Over Bluetooth

To scale this project up in the future with more time and resources we would want to offload all of the computer vision processing onto the iPhone device. This would enable us to leverage CoreVision and CoreML libraries to perform a learning based approach that could not just detect objects, but classify them enabling the user's legs/shoe to avoid being detected as dirt as well as minimizing the form factor of the sensing module.

11.1.2 Conversion to Virtual Reality Headset

We did not approach using a virtual reality headset for a multitude of reasons but this could be pursued in a future work. The main software component of the system relies on anchor reference images in the field of view, and the system is modularly designed that the core logic could theoretically be repurposed in a an Apple Vision Pro. We do not have experience with visionOS but could learn if given more time and financial resources.

11.1.3 Gamification

We could add more gamified metrics like percentage of floor covered, and statistics to help motivate cleaning. In addition, this system could have an interactive web component where users compete with their friends, trying to beat one another's scores. Another gamification possibility is putting tokens or candies to collect on the floor, potentially making cleaning more fun for children.

11.2 Lessons Learned

11.2.1 Start Integration Early

We learned the importance of starting to integrate early in the development process. Initially, we focused heavily on working in parallel with different individual components, assuming integration could be handled once all parts were ready. However, we quickly realized that issues often arise when combining elements. Early integration testing allowed us to identify and address these issues sooner. There were a lot of unanticipated challenges; for example, we ran into a plethora of issues with Bluetooth. Troubleshooting these problems resulted in us needed to re-flash the OS on our Jetson multiple times, and the abysmal speed of our disk caused us to need to replace the SD card entirely. Throughout this project, we were constantly reminded that hardware is unpredictable. This approach helped us keep on schedule and ultimately led to a more reliable product.

11.2.2 Frequent Team Communication

Another key lesson was the importance of team communication. We found that regular check-ins via messages made sure that everyone was on the same page with tasks, expectations, and development work. This was really difficult amongst our challenging classes because it was hard to find times to sync, but we got into a rhythm of knowing one another's schedules after awhile.

11.2.3 Setting Meeting Objectives

For each work session, we found that we were more productive when we established what our goals were for the end of the meeting. While these were not always satisfied, it was a good checkpoint to know whether we were on track or needed help. We also assigned specific tasks to team members, and prioritized tasks that had to be done together instead of the tasks that could be done asynchronously. This structure helped make us more efficient and also ensured that every team member had a clear understanding of their responsibilities and deadlines.

12 SUMMARY

This project aims to reduce the headache that can be caused by missing spots on the floor when vacuuming. Existing vacuum products may include active illumination, but there is no metric to keep track of what part of the flooring has been cleaned or alert the user of any problem areas. Our product helps users achieve a better, deeper clean of their space by integrating AR into the vacuuming experience. Our product highlights any locations which have not yet been cleaned, and will alert the user to any spots they may need to clean again.

Glossary of Acronyms

- AR – Augmented Reality
- BLE - Bluetooth Low Energy
- CUDA – Compute Unified Device Architecture (NVIDIA's proprietary GPU platform)
- CV – Computer Vision
- LiDAR – Light Detection and Ranging
- RPi – Raspberry Pi
- SLAM – Simultaneous Localization and Mapping

References

[1] Apple. URL: https://developer.apple.com/documentation/corebluetooth/transferring-data-between-bluetooth-low-energy-devices.

[2] Apple. *Buy Apple Vision Pro*. URL: https://www.apple.com/shop/buy-vision/apple-vision-pro.

[3] Apple. *Saving and Loading World Data*. URL: https://developer.apple.com/documentation/arkit/arkit_in_ios/data_management/saving_and_loading_world_data.

[4] bluetooth beacon. *Bluetooth VS WiFi comparison*. URL: https://www.netguru.com/blog/bluetooth-vs-wifi-comparison-for-the-iot-solutions#:~:text=In%20most%20cases%2C%20Bluetooth%20devices,electric%20power%20than%20BLE%20devices..

[5] Daniel Beauchamp. *Spatial Vacuuming*. URL: https://twitter.com/pushmatrix/status/1749797146961006716.

[6] Loz Blain. *AR vacuuming app makes sure you don't miss a spot*. URL: https://newatlas.com/vr/ar-vacuum-app/.

[7] bluetooth. *Bluetooth Key Attributes*. URL: https://www.bluetooth.com/learn-about-bluetooth/key-attributes/range/.

[8] Kevin Davis. URL: https://github.com/kevincar/bless/blob/master/examples/server.py.

[9] Apple Developer. *Tracking and Visualizing Planes*. URL: https://developer.apple.com/documentation/arkit/arkit_in_ios/content_anchors/tracking_and_visualizing_planes.

[10] Max Frazer. *SceneKit-SCNLine*. URL: https://github.com/maxxfrazer/SceneKit-SCNLine.

[11] Shawn Halayka. *Particle analyze in binary image*. URL: https://answers.opencv.org/question/180900/particle-analyze-in-binary-image/.

[12] iamtecksay. *How fast can bluetooth transfer data*. URL: https://medium.com/@iamtecksay/how-fast-can-bluetooth-transfer-data-c9611fd67a6d.

[13] RF Wireless World. *Bluetooth Protocol Stack*. URL: https://www.rfwireless-world.com/Tutorials/Bluetooth-protocol-stack.html.

[14] Ximea. *Jetson Nano with Embedded vision cameras – Benchmarks*. URL: https://www.ximea.com/support/wiki/apis/Jetson_Nano_Benchmarks#:~:text=The%20critical%20point%20is%20that,the%20J28%20Micro%2DUSB%20connector..

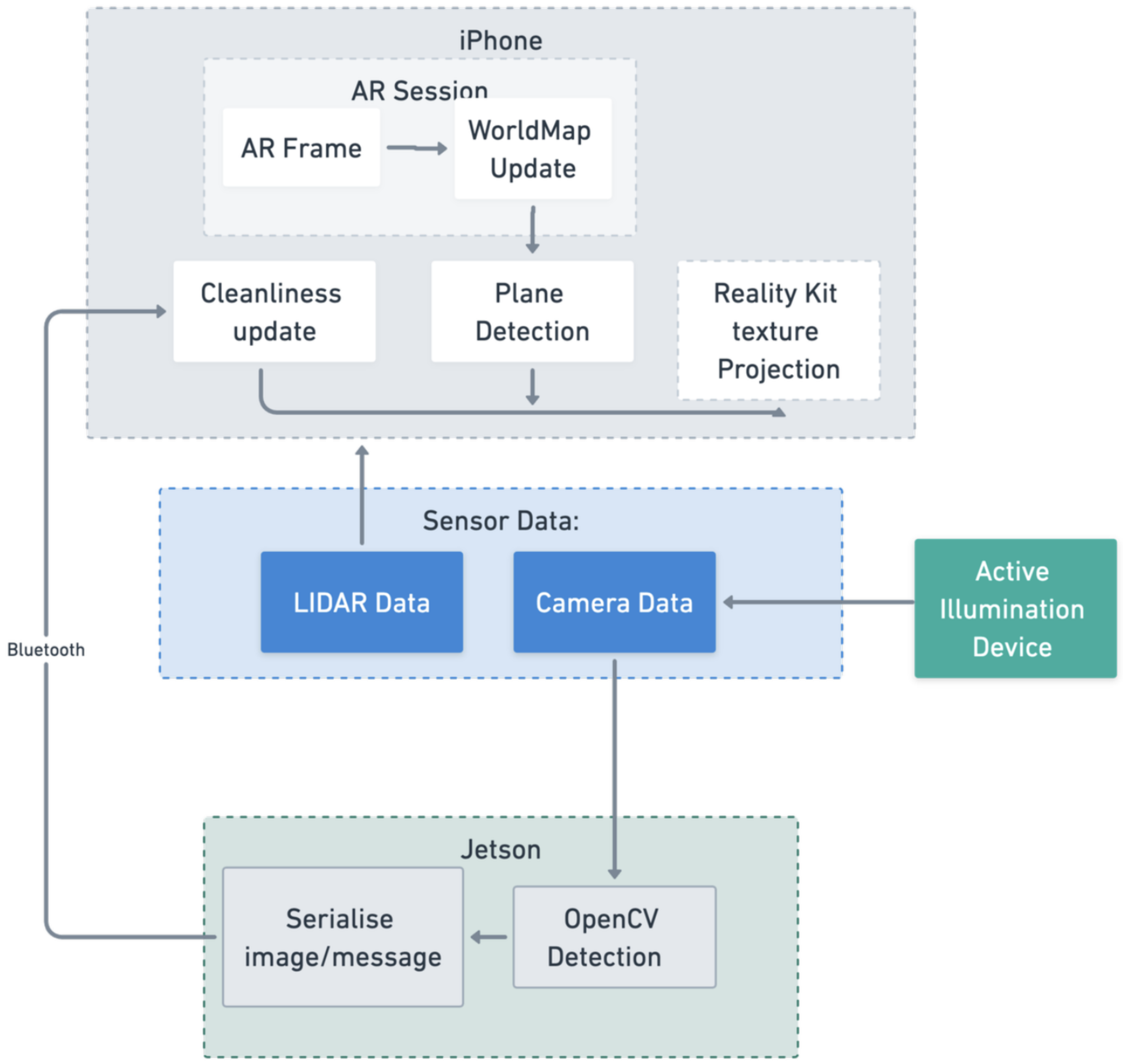


Figure 12: Block Diagram for the system

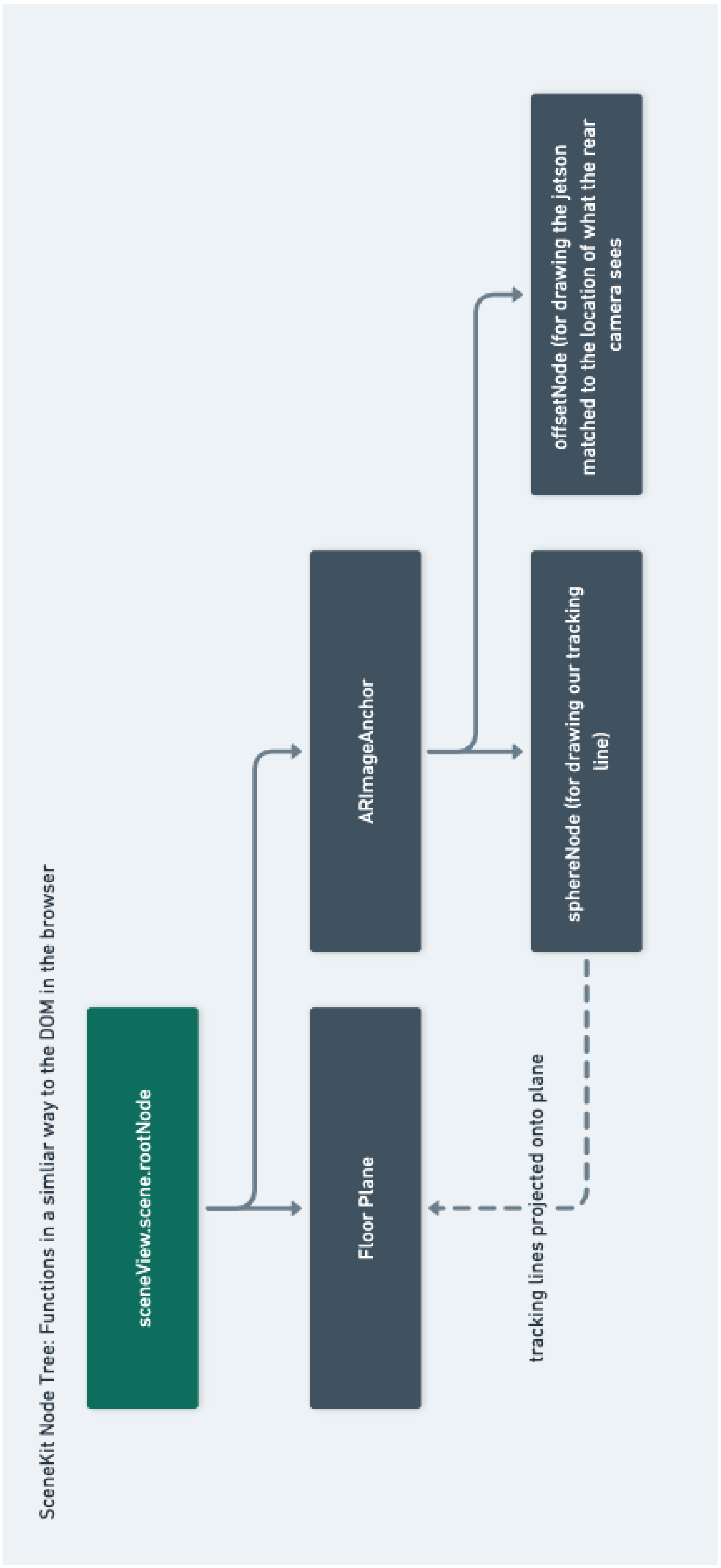

SceneKit Node Tree: Functions in a simliar way to the DOM in the browser
sceneView.scene.rootNode
Floor Plane
ARImageAnchor
sphereNode (for drawing our tracking line)
offsetNode (for drawing the jetson matched to the location of what the rear camera sees
tracking lines projected onto plane

Figure 13: SceneView Hierarchy

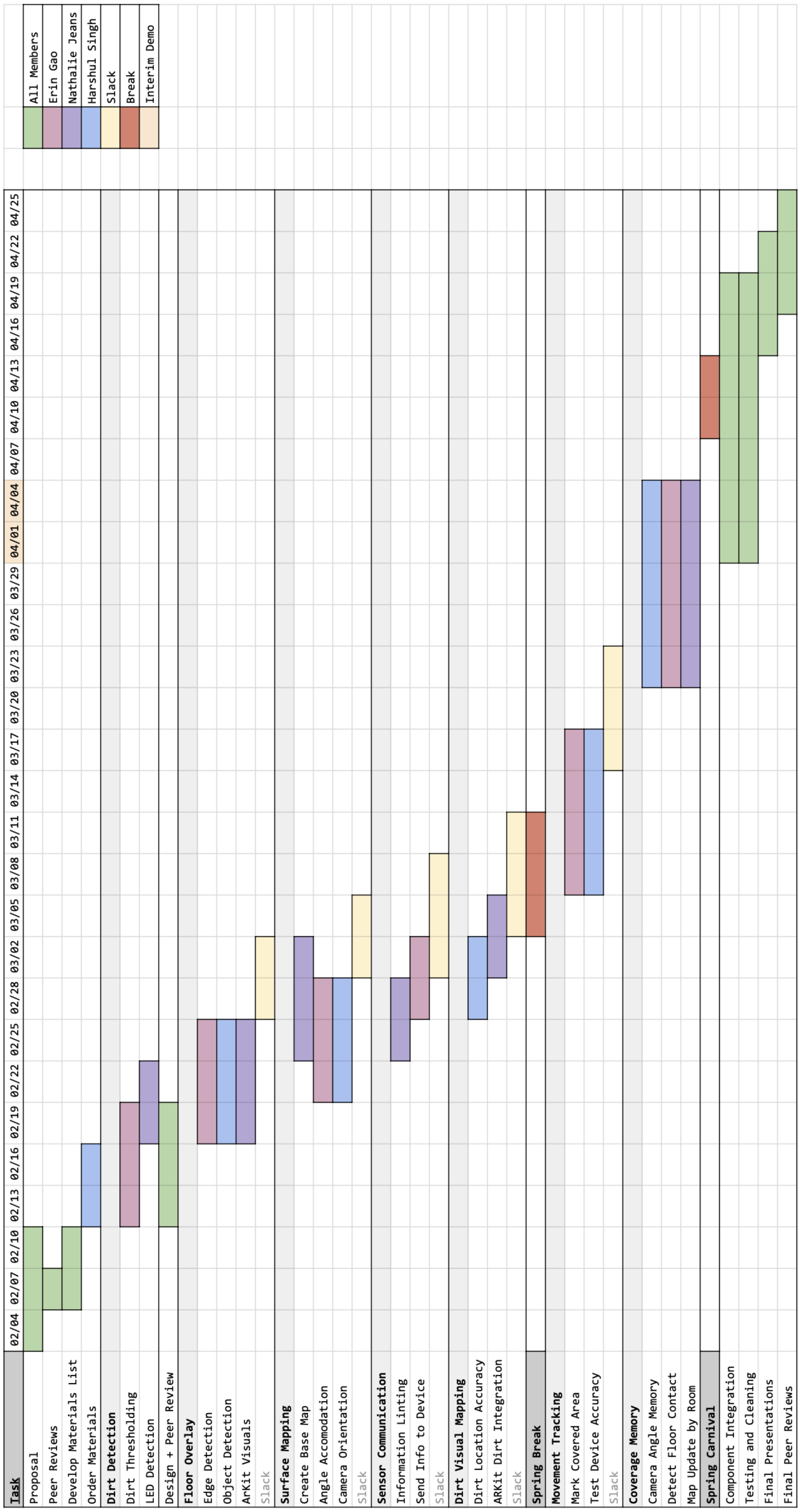


Figure 14: Gantt Chart