KinectBot

John P Mayer, Cong Liao, and Jun Guo
Department of Electrical and System Engineering
University of Pennsylvania
{jpmayer, liaocong, junguo}@seas.upenn.edu

Abstract - This paper describes our team's approach to design an autonomous pathfinding robot. An iRobot Roomba with an on-board BeagleBoard computer uses a Kinect Sensor to avoid a series of obstacles to a target object and return to the starting position. Implementation details, design decision, and results are dicussed.

I. Introduction

Path finding robots have existed using many different techniques for obstacle perception, position tracking, and avoidance intelligence. As relevant technologies continue to increase in complexity and decrease in cost, it becomes quite attainable for hackers to quickly assemble a system from consumer components and a bit of glue software.

This system was designed and implemented in the context of a competition; two teams using similar base hardware would independently develop vision, control, and intelligence algorithms to navigate a fixed course, retrieve an object at a colored checkpoint, and return to the starting position.

The motivation for this competition was twofold. First, there exists a desire to explore the limitations of the consumer devices used in the system. All three major hardware components, the Kinect, Roomba, and BeagleBoard, have been used to great extent by the embedded hacker community in a variety of independent use cases. The opportunity to contribute to this growing collection is in itself desirable.

However, there exist many practical applications of autonomous path finding units. The state of the art in this field includes automated distribution centers, autonomous passenger vehicles, and even space-faring mobile vehicles. While KinectBot doesn't even begin to reach the complexity of these examples, the experience sheds light on some of the challenges and engineering tradeoffs that inevitably penetrate the design process of these advanced systems.

II. Implementation

Kinectbot consists of a handful of distinct hardware and software subsystems. This section will explain in detail the functionality and design choices that define each component. At a high level, KinectBot combines stock and custom hardware with a collection of vision, control, and intelligence algorithms to complete the pathfinding task.

A. Stock Hardware Components

An iRobot Roomba Create (TM) was used as the chassis of the device. The Roomba is capable of forward and backward movement, in-place turning, and curved movement. It has cliff detectors on its underside that are used to prevent falling over high edges, and has a single bumper-detector in the front which is used by the cleaning system to detect obstacles. Most important is the serial communication interface, which can be used to effect direct control of the Roomba's drive motors and query different status values of the chassis. The serial communication interface will be discussed in greater detail in section II.E.

The Microsoft Kinect is the primary input device of the system. Originally designed for the xBox, it provides a video camera and an integrated depth sensor, as well as the (unused in this project) microphone and positioning motor. While an official SDK is available for the Microsoft Windows platform, the OpenKinect driver was developed by open source and embedded systems enthusiasts to use the kinect on a wider variety of systems. It provides the C/C++ libfreenect shared library on linux, and its use will be discussed in section II.D.

Finally, the BeagleBoard xM is a small ARM-powered embedded computer. It provides a number of hardware interfaces; of particular use were the USB ports, as well as the Ethernet port for code deployment. An 8GB micro SD card was used to hold the operating system and application code between powered sessions.

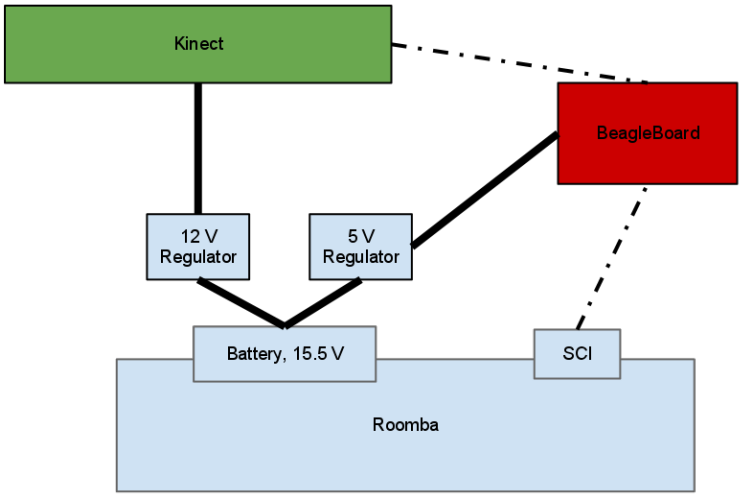

Figure 1. Wiring Diagram

B. Custom Hardware Components

All three components require independent power sources to operate, but only the Roomba was designed explicitly for battery-powered operation. Luckily, the Roomba Create exposes multiple current-regulated access points to the battery at 15.5V. It was decided to use this port on the

back of the Roomba, so a custom power unit was designed to bridge between the battery and the Kinect and BeagleBoard. A breadboard was used to hold the components responsible for this bridging. For the beagleboard, a stock 5V voltage regulator was used, and for the Kinect, a 12V voltage regulator. The power cables for the Kinect and Beagleboard were then modified to be directly attached to the newly available voltage sources. The wiring diagram of the entire system can be seen in Figure 1.

C. Operating System and Requisite Libraries

Ubuntu Linux 11.10 was chosen as the operating system for two main reasons. First, Ubuntu has a reputation of terrific hardware support, which was confirmed in our implementation. An OMAP image was downloaded from Canonical, and was written to the microSD card using the unix *dd* utility. Upon first boot, Ubuntu was configured to support networking, which enabled code deployment via a remote repository. The second reason Ubuntu was chosen was the extensive library support, as Ubuntu is a derivative of the Debian Linux distribution. Using the apt-get utility, it was extremely easy to install packages required by libfreenect on-the-fly. Although the process of downloading, configuring, and installing all of the packages did take on the order of hours, it was only required once.

A few libraries required by KinectBot are worth noting. The first, libusb, is an open source USB interface that is used by libferenect to access the Kinect. Build tools like cmake and gcc were essential for setting up the libfreenect library as well, since it was downloaded from github as source and built on the device. The distributed version control system, git, was instrumental in keeping track of changes made on-the-fly on the board, as well as providing an easy way to deploy changes made on development machines. Finally, minicom was used extensively during experimentation and demonstration when KinectBot needed to be disconnected from any networks.

D. Vision Techniques

The kinect sensor was responsible for two independent tasks: obstacle detection and object recognition. At the library level, a synchronous C library was provided by the OpenKinect community for direct access to the video and depth buffers on the device. Each depth buffer is a 640x480-pixel array; for video these are 3x8-bit RGB structs, and for depth these are 11-bit depth values. Before being used, video data was converted from the RGB to HSV color space. Finally, both buffers took advantage of region partitioning; the array was broken into 40x40-pixel regions, where target metrics could be summed or averaged for use by the AI and control subsystems.

Obstacle detection used the region-partitioned depth buffer to determine if an obstacle was present in front of the robot. For each region, the depth values of each pixel was averaged and compared to a threshold value. This way, the control system had access to a 16x12 binary matrix, with truth values referring to the presence or absence of an obstacle in that region.

Object recognition, on the other hand, needed to check if the field of view in front of the robot

contained red objects. To determine if a pixel was red, the first approach was done in RGB space, where the angle between the test pixel and pure red was calculated. This proved to be extremely sensitive to light levels, and thus too unreliable for different experiment settings. The second and current approach defines a region of HSV space to be red, which had a much higher accuracy in different lighting environments. Using this approach, the number of red pixels in a region could be determined, which could be compared to a threshold value.

The red-detection functionality needed to be used in two places. First, when an obstacle was detected, it was important to treat it differently if the obstacle was red, signifying a checkpoint. In this manner, the matrix of obstacle presence was used as a mask to define in which regions the red-detection should be applied. In the second case, the entire field of view should be checked for red regions, but with a different threshold. The code was generalized to take both a bit-mask and a threshold value, with the field-of-view instance using the all-true bit mask and a lower threshold.

E. Base Roomba Control Library

The Roomba Serial Command Interface (SCI) enables us to control Roomba through serial protocol. Commands to control Roomba actuators (motors, LEDs and speaker) as well as querying sensor data are included in SCI. We implement the actuation by sending specific data to Roomba through serial port.

In a Linux system, the serial port operation is just like file operation. For example, if we want to start Roomba, we can use this piece of code:

```
char buf[] = {128};
int ret = write(fd, buf, 1);
```

Here, we need to initialize serial port at first and “fd” is the returned file description when open serial port as a file. And according to the Roomba SCI manual, the command op-code to start Roomba is 128 with no data bytes. So, Roomba will start after receive the data “128”. Detailed instructions about SCI command can be found in the Roomba SCI manual.

We use the existing Roomba library written by Tod E. Kurt in our system. Source code of this library can be found in website: hackingroomba.com. This library is written in C language and provides us some basic functions so that we can directly use this library in Ubuntu on our beagleboard. The rest of this subsection details the parts of the roomba control library used by our system.

```
Roomba* roomba_init( const char* portpath )
```

By passing the path of serial port, this function will initialize Roomba and return a pointer to

initialized object. The pointer will be used during the whole process.

```
void roomba_forward( Roomba* roomba )
```

This function makes Roomba move forward until stop command is sent.

```
void roomba_spinleft( Roomba* roomba )
```

This function makes Roomba spin left until stop command is sent.

```
int roomba_read_sensors( Roomba* roomba )
```

This function enables us read sensor data of Roomba including button status, traveled distance since last query in millimeters, turned angle since last request and so on.

```
void roomba_delay( int millisecs )
```

There should be a minimum interval between two consecutive commands, otherwise the second command will be ignored by Roomba. So we need to call roomba_delay to satisfy this interval requirement.

```
void roomba_stop( Roomba* roomba )
```

This function can stop Roomba.

```
void roomba_free( Roomba* roomba )
```

We should never forget to call this function to free the memory which is dynamically allocated when initialize Roomba.

F. Roomba Control Abstractions

In order to improve the performance of our system, we synthesize some high level control logics based on the functions Roomba library provided, for example, turning to specified absolute degree, moving desired distance along x axis or y axis. By using these control logics we can achieve simple obstacle avoidance algorithm, e.g. when detected an obstacle, turn right or turn left through 90 degree and move forward for desired distance to avoid this obstacle, then turn back to original direction. Detailed AI algorithm will be introduced in Section II.G.

In this section, I will first of all describe how our coordinate system keeps track of the absolute position of Roomba. Then the detailed implementation of high level logics will be introduced.

i. Positioning

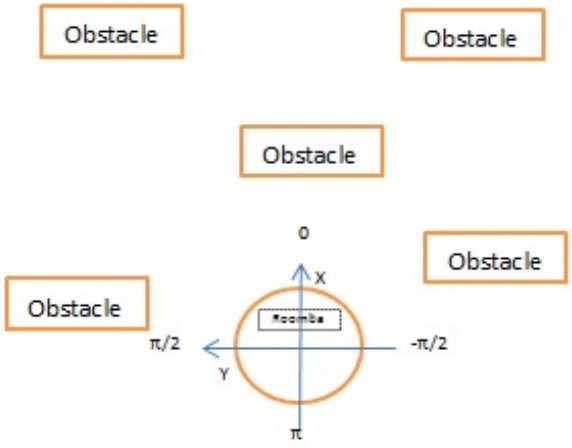

Figure 2. Coordinate System

We use three metrics to record the absolute position of Roomba, namely, *posX*, *posY*, *posT*. Respectively they are x position, y position, radian between current direction and x axis. When the system starts, Roomba will automatically set its position as origin (0, 0, 0). Every time Roomba moves, it will update the position information. The update codes are shown below:

```
roomba_delay(100);
roomba_read_sensors(_roomba);
int8_t* sb = ((Roomba*)_roomba)->sensor_bytes;
int16_t dist_i  = (sb[12]<<8) | sb[13];
int16_t angle_i = (sb[14]<<8) | sb[15]; // in degrees
printf("delta dist:%d angle:%d\n", dist_i, angle_i);
double dist = dist_i;
double angle = angle_i * (2.0 * PI) / 360;
posT += angle;
posT = normalize_angle(posT);
posX += dist * cos(posT);
posY += dist * sin(posT);
```

Once Roomba need to update its position information, it first reads the sensor data to see the relative distance it moves and relative angle it turns through. The position related information is stored in 12th, 13th, 14th, 15th bytes of the data packets. Then we calculate the absolute position. It is worth noticing that we need to normalize the value of post so that it is within the range of (-PI, PI).

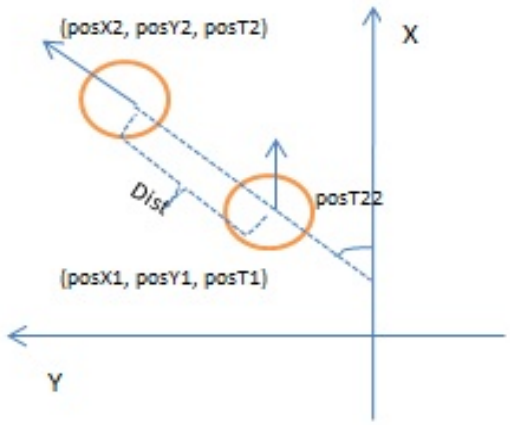

Figure 3. Position Update

From Figure 3, we can see that when Roomba moves to the second position, the absolute angle between Roomba's direction and x axis is posT2. Therefore, the calculation should be as follows:

```
posX2 = posX1 + Dist * cos(posT2)
posY2 = posY1 + Dist * sin(posT2)
```

ii. Scripted Movement

```
void waitRoombaPlayBtnPush(Roomba* _roomba)
```

In this function, we will constantly read the sensor data of Roomba to check whether desired button is pushed. Once the button is pushed, we will receive the data with specified byte set to 1 and thus break the loop.

```
void orientToAngle(Roomba* roomba, double targetAngle)
```

The most important thing to make Roomba turn through desired angle is that we must always update current position of Roomba. If current angle is not in tolerable deviation range, then keep Roomba spinning.

```
void moveMillimetersY(Roomba* roomba, double yDist)
{
   if (yDist == 0) return;
   if (yDist > 0)
```

```
      orientToAngle(roomba, PI/2);
   else
      orientToAngle(roomba, -PI/2);
   double targetYPos = posY + yDist;
   while ( ( yDist > 0 && posY < targetYPos ) ||
           ( yDist < 0 && posY > targetYPos ) )
   {
    roomba_forward_at(roomba, FOW_SPEED);
    exc_one(roomba, 'q');
   }
}
```

In case that Roomba moves out of the area, if *posY* is greater than *CENTER_COLUMN_R*, namely on the left of the area, we will let Roomba move to right, otherwise move to left. We also have two different avoid distance which is the distance that Roomba need to move in order to avoid an obstacle. First of all, we will check whether the obstacle is right ahead of Roomba, if so, Roomba need to walk a relatively long distance to avoid it, otherwise just make Roomba move a small distance towards suitable direction. For example, if Roomba detects that the obstacle is on the right, it just turns left and make a small move.

G. AI

The task required for the Roomba to complete is to reach a target position by moving through a harsh environment with obstacles and return back to the starting point safely and autonomously. So what AI needs to accomplish can be essentially classified into two sub-tasks: one task is controlling the robot to navigate along a path free of obstacles in the round trip, the other one is enabling the robot to tell whether or not it has reached the target position or returned back to the starting location indicated by red objects.

The goals of obstacle avoidance and target detection are achieved primarily based on the image data gathered from the RGB and depth camera of kinect sensor as well as position data from the Roomba itself. The status of the robot during the whole process is basically classified into four modes: 1) MODE_SEEK; 2) MODE_UTURN; 3) MODE_RETURN; 4) MODE_FINISH. Obviously, the AI is required to perform obstacle avoidance when in MODE_SEEK, seeking a safe route to the destination and in MODE_RETURN, returning back to the starting position without any collision. Once the robot confirms that it has arrives at the target location or returned to the starting point by red detection, it will be in MODE_UTURN or MODE_FINISH respectively. The methods of obstacle avoidance and red detection will be elaborated in details as follows.

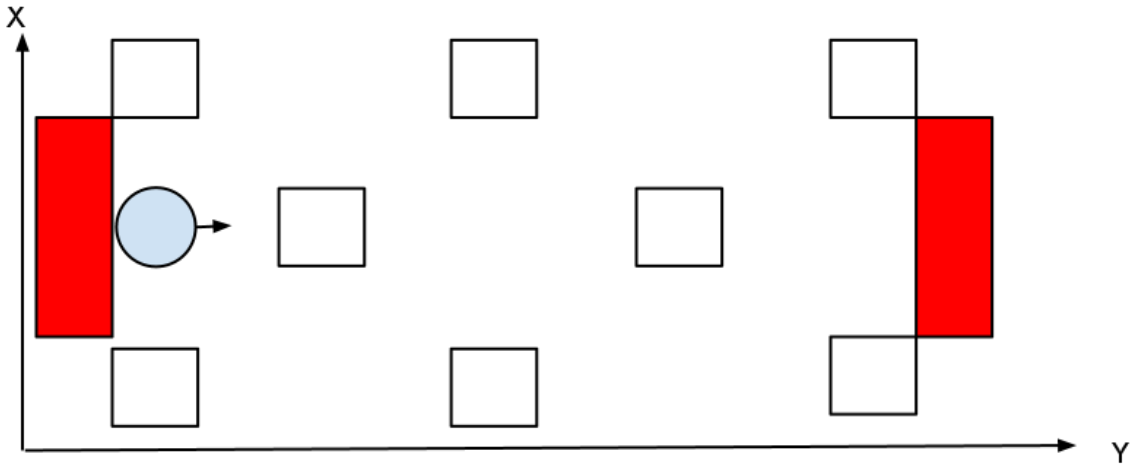

Figure 4. Obstacle Arena

i. Obstacle Avoidance

The information of the environment that the Roomba relies on for obstacle avoidance is perceived by the kinect sensor, which retrieves a 640x480 depth image, displaying what the robot actually "sees" in front of it. The process of decision making is mainly based on the depth image.

a. Approach I

Intuitively, the robot will not bump into any obstacle if there is no obstacle in its way at all as it moves. Therefore, our preliminary thought of navigation is to make the robot move in the direction that is safe and clear of obstacles. First of all, the depth image is processed to show only the closest obstacle in the images. Then it is divided into 16x12 regions and finally converted into a 16x12 binary array, containing 0 and 1 after a serial of image data processing. 0 means the corresponding region is clear and safe, while 1 means the robot "sees" that the particular region is occupied by an obstacle. Each column of the binary array can be deemed as a direction, towards which the robot can choose to move, on its path to the target. To avoid a detected obstacle, the simplest way is to move to the right/left when the majority part of the obstacle is on your left/right side. So the basic idea is to check every column (index from 0~15) of this 16x12 binary array to see if it contains occupied regions. If it does, then the column or direction is considered unsafe and labeled 1. We use a variable *weight* to determine which side (left or right) the majority part of obstacle is located, and then control the robot to turn towards the other direction. If the weight is greater than 7.5, then the robot should turn left, while the robot should turn right if weight is less than 7.5.

$$weight = \frac{Sum\,of\,index\,of\,columns\,with\,label\,1}{Total\,number\,of\,columns\,with\,label\,1}$$

The biggest problem we have with this approach is that the robot cannot detect obstacles if it

enters the blind zone when changing its route. Therefore, there are potential risks that the robot might not be able to avoid the obstacles completely and bump into partial parts of obstacles because we don't take the shape and size of obstacles and Roomba itself into accounts.

b. Approach II

In order to solve the problems we encountered before and make improvement on our obstacle avoidance process, we come up with another method that greatly increases the chances of successful obstacle avoidance. The way to do obstacle detection is slightly different from the previously mentioned method. We don't use the 0/1 binary array to represent a region but an array with average pixel value of every region. So we decide whether or not a region is considered a occupied region in terms of the average pixel value of that region. If the average value (*region_avg*) is higher than the threshold (*D_THRESH*), it is considered as a occupied region and its corresponding column is labeled 1 as unsafe direction. As for obstacle avoidance, the major difference between the two approaches is that we adopt a more safe and robust strategy to avoid the obstacles. We use a variable *centerofmass* to denote the relative position of detected obstacle with respect to the robot. The way we calculate *centerofmass* is the same as we do with weight. To avoid a obstacle, we make the robot turn left/right first and move a certain distance *D*, then let the robot turn back to original moving direction and keep going forward. We sets two different distance D, based on the size of the obstacles and field. One is *Y_AVOID_S* with a small value and the other is *Y_AVOID_L* with a relatively larger value. The choice of distance *D* only depends on where the *centerofmass* locates with respect to the robot. As mentioned before, each column of the 16x12 array can represents a direction in front. It can also be regarded as one section of the front-view area. If the *centerofmass* is within the leftmost / rightmost sections, the robot will just move a relatively small amount of distance (*Y_AVOID_S*) to the right / left, otherwise, it will travel a longer distance (*Y_AVOID_L*). For the second scenario, we record the position (*posX*, *posY*) of the robot in global coordinate to determine which direction it should turn to. If the *posY* is negative / positive, it should move towards left / right. Each time the robot turns and moves a certain distance *D*, it will orient back to its initial orientation.

ii. Red Detection

The red detection is executed after obstacle detection. It is primarily based on the RGB image provided by the kinect sensor. It is actually a region-based approach, just the same as how we deal with the depth image in obstacle avoidance. The sensor data read from the kinect sensor is interpreted into a 16x12 array with information about the color of every region by some image processing techniques covered in section II.D. We loop through every element of the array and compute the total number of red regions in the image. If the number is above the threshold (*R_COUNT_THRESH*), then the detected obstacle is likely to be a red object. However, since a red object can also be seen as an obstacle by the robot, we therefore define another criteria (*RedObstacleRatio*) to determine whether or not the current detected obstacle can be considered as a real obstacle or a red object that defines a target location or starting point. If the ratio is higher than the threshold (*RED_RATIO_THRESH*), then the robot is positively sure that it finds the red object indicating it has arrived at destination or return to starting location. Finally, the

robot will be in MODE_UTURN if its previous mode is MODE_SEEK after red detection, otherwise, it switches to MODE_FINISH if it is in MODE_RETURN before red detection.

III. Results

KinectBot is able to navigate the obstacle course with high reliability and precision. The final implementation outperformed previous iterations by a wide margin, where obstacle detection failures were common, and return implementation was incomplete.

One of the more interesting experimental results involves the maximum speeds of the robot where operation is correct and accurate. As noted in section II.H on the Roomba SCI, the speed at which the Roomba chassis moves forward or turns can be parametrized, with a maximum value of 500mm/s. We added command-line flags to experimentally determine the fastest forward speed and turn speed that did not affect the operation of the pathfinding.

For forward movement, we were able to achieve a maximum speed of 300mm/s. At higher speeds, KinectBot would drive through obstacles. The best explanation for this behavior is that the frame rate of the kinect sensor was lower than the time between getting in range of the obstacle and getting too close to the obstacle where it wouldn't be registered. Possible techniques to improve the maximum forward speed could include optimizing the control algorithm, or using the asynchronous kinect API to remove I/O latency in the outer loop.

Spin speed, on the other hand, was capped at a much lower 50mm/s. At higher spin speeds, the kinect bot would not be able to stop at a specified angle, and would turn back and forth, always overshooting the target angle and the tolerated error threshold. The major cause for this is the serial port latency of the Roomba SCI - while writing to the port was very fast, the time between receiving the command and changing the behavior of the motors was disproportionately large. Two techniques could have been used here. First, direct control of the motors could offer a faster response time; however, the amount of hardware hacking would have been significant. A second approach would have used a variable turn speed - turning quickly when the difference between the target and current angle was large, but slowly when high precision was required for the last few degrees.

IV. Conclusion

Overall, KinectBot is able to accomplish the task that it was designed for. However, the system was very much designed to fit the specific task and relies heavily on tight specifications and guarantees. As a result, this section first will discuss the limitations of KinectBot and the changes that would be required to accommodate a more general purpose. In addition, there will be short discussion on the challenges faced during the design process.

As previously stated, KinectBot's goal is quite narrow. In the strictest definition, the kinectbot does not *search* for the red checkpoint objects. The kinectbot expects that these checkpoints will be in the center of the arena, and that its original position is directly along this center line.

Obstacles are expected to be of fixed size, as the avoidance algorithm uses hard-coded distances to ‘side-step’ anything in front of it. Moreover, there must exist a path through the arena that is monotonically increasing in the x direction - the robot is incapable of detecting a solid wall and subsequently realizing that backtracking may be necessary to complete the course.

A few additions could be made to the intelligence algorithm to accommodate this more advanced behavior. A history of objects could be recorded and used in a algorithm similar to depth-first-search in graph traversal. Either objects could be remembered with spatial coordinates relative to the position where they were discovered, or a graph consisting of vertices representing safe coordinates and edges representing safe paths could be maintained and used to accomplish backtracking.

Another limitation of the implementation is the complexity of checkpoint objects. A small modification to the could could enable the detection of other solid colors by altering the hue in HSV color space, but a significant addition to the visual component would be required to instead use a target shape or logo. This would also have a non-trivial impact on the latency of the system.

The single-threaded implementation of the system, while practical and simple to reason about, could be seen as a major architectural shortcoming. The latency of blocking calls to both the kinect and the Roomba could be replaced with interrupt driven tasks that update mutually exclusive state objects, with a periodic task devoted to control and AI. We would first expect this alternative implementation to improve the latency of obstacle detection to actuation, allowing an increase in drive speed. In general it would simply be good practice to break up the outer loop into its elementary functions, giving us control at a finer granularity to prioritize different tasks.

Finally, the infamous ‘blind-spot’ of the kinect sensor was a source of major headaches and quite a few workarounds. Because of the proprietary hardware implementation of the depth sensor, there was little that could be done to mitigate the fact that objects closer than about 20cm from the device yielded no depth data, but more importantly, were indistinguishable from a background of infinite distance. Replacing the kinect sensor with another vision system that could support the nearsighted requirements for obstacle detection could result in major improvements in accuracy.

V. Acknowledgement

We would first like to thank the OpenKinect community for their contributions to kinect hackers everywhere; our project would not be possible for their hard work. Similarly, we’d like to thank the Kinecthesia team for their help finding hardware as well as their useful tutorials for setting up our environment, and the members of the mLab for their support and resources.

Finally, we truly appreciate the criticisms, guidance, and motivation from our TA Madhur Behl and and our instructor Rahul Mangharam.

VI. References

1. http://www.kinecthesia.com/
2. https://github.com/OpenKinect/libfreenect
3. http://openkinect.org/wiki/C_Sync_Wrapper
4. http://roombahacking.com/roombahacks/roombacmd/
5. http://en.literateprograms.org/RGB_to_HSV_color_space_conversion_(C)
6. http://www.irobot.com/images/consumer/hacker/Roomba_SCI_Spec_Manual.pdf
7. http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf