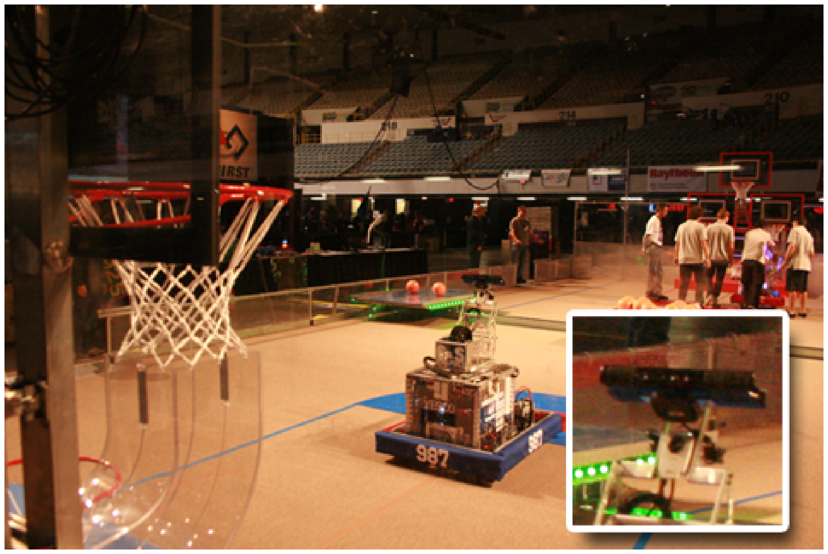
How 987 used the Kinect

Team 987

Introduction

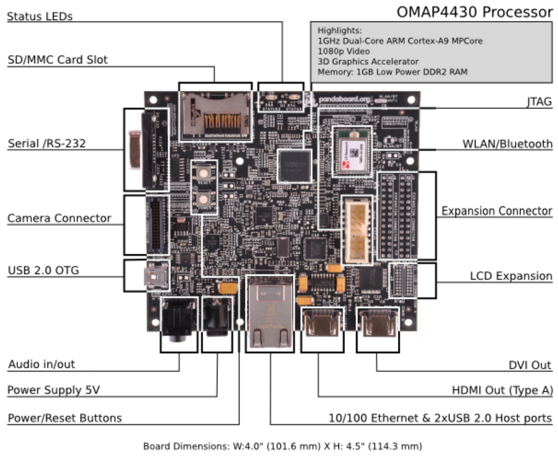
For the 2012 FRC season, a Microsoft Kinect was included in the kit of parts. The Kinect was meant to be used during the Hybrid period of “Rebound Rumble”. Our team decided to try to utilize the Kinect as a sensor on our robot. In addition to color images, the Kinect generates “depth maps”. A depth map is an image where the value stored at each pixel location is the distance from the sensor to whatever it sees. The Kinect generates depth maps at a resolution of 640x480 with 11-bit depth values. We found the depth values to be accurate down to inches up to a range of about 12 feet; perfect for our shooting range in Rebound Rumble.



In this paper we’ll describe how we integrated the Kinect into our robot. The FRC build season is only 6 weeks and this project required us to work with many unfamiliar technologies such as the kinect, sockets programming, linux, and single-board computers.

You need a second computer onboard.

The Kinect transfers data using a standard USB cable. Unfortunately the cRio cannot interface with USB. This means you need a second computer onboard. One advantage of this is you can have that computer process the data and then transmit just the targeting results to the cRio (more on that later). Near the beginning of the 2012 season, someone on Chief Delphi mentioned that you “could use a Pandaboard” for this purpose so we did some google searches on Pandaboards. The Pandaboard is a single board computer which has no moving parts and can run off of a 5V 2A power supply. We found several web-pages showing how other people have used a Pandaboard to interface with the Kinect so it seemed likely to work for us.



You can get the Pandaboard or Pandaboard ES from digikey here:

http://www.digikey.com/

You’ll also probably want to read everything you can on this website:

http://pandaboard.org

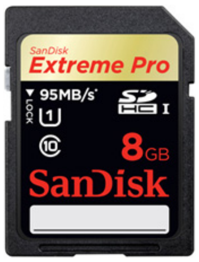
We have both a Pandaboard and a Pandaboard ES. On the actual competition robot, we used the Pandaboard just because that’s what we got first and we never changed it out. The ES version is slightly newer and has a faster processor (1.2Ghz vs 1Ghz). Both of them worked fine in our tests.

Ubuntu

The Pandaboard can run Ubuntu which is a flavor of the Linux operating system. We did not have a lot of experience working with Linux but it turned out to be reasonably easy to use. It has an especially nice system for finding and installing free software packages. Getting Ubuntu up and running on our Pandaboard did take a while though. First of all, you have to find an Ubuntu image that is compiled for the Pandaboard CPU. We used the Ubuntu 11.10 image found here:

http://omappedia.org/wiki/Ubuntu_Pre-Built_Binaries

Once you’ve downloaded the image, you copy it onto an SD card using a tool called “Win32DiskImager”. Then you plug the SD card into the pandaboard and power it up. What is supposed to happen is the Pandaboard will boot up and then “expand” Ubuntu onto the SD card. Unfortunately we had to try this process many times before we got it to work. We tried 4 different types of SD cards and only one type would work for us. Typically we would get halfway through the process and some cryptic error would pop up and we'd have to start over in an endless loop. I won't go over all of the details of installing Ubuntu but here is the type of SD card we used that actually worked (we have two of these and both worked for us):



We think the key here is the speed rating, signified by the '1' inside of a 'U' shape. These are the fastest currently available SD cards.

Pandaboard Stability

Once you have Ubuntu running on the Pandaboard, you really need to update its drivers to the ones that Texas Instruments provides. Until we did this, the Pandaboard would randomly hard lock, in many cases causing us to lose data and even corrupt our SD card. At first we thought the board was overheating because the cpu does get a bit hot to the touch but this was not the problem. We even built a case with a fan in it but that did not fix the problem. We found the real solution after talking to a TI engineer in a Pandaboard IRC channel. After updating to TI's drivers, we have not experienced a single hard lock even if we run our Kinect code for days at a time. Overheating has not been (never was) a problem and we stopped using our fan. To get the TI drivers, you can follow the instructions here:

http://omappedia.org/wiki/PandaBoard_Ubuntu_PPA

Kinect Programming

We wrote our Kinect targeting code in C++ using CodeBlocks and the OpenKinect library. The following two commands should install them on your Ubuntu system:

```
sudo apt-get install codeblocks
sudo apt-get install freenect
```

We chose the smallest simplest sample program that came with OpenKinect as a starting point for our targeting code. The sample we started with is called **glpclview.c** and you can find it on the OpenKinect website. It uses a simple synchronous API to read color and depth images from the Kinect and then display the data in 3D using OpenGL. As a bonus, the entire program is only 230 lines of C code! You may be able to get better performance by using the more advanced asynchronous APIs in OpenKinect but we have not yet tried.

Working with 3D Point Cloud Data

The depth image data you get from the kinect comes in the form of a bitmap of 11bit values. Here is the line of the code from the sample program mentioned above that reads the depth map from the Kinect:

```
freenect_sync_get_depth((void**)&depth, &ts, 0, FREENECT_DEPTH_11BIT)
```

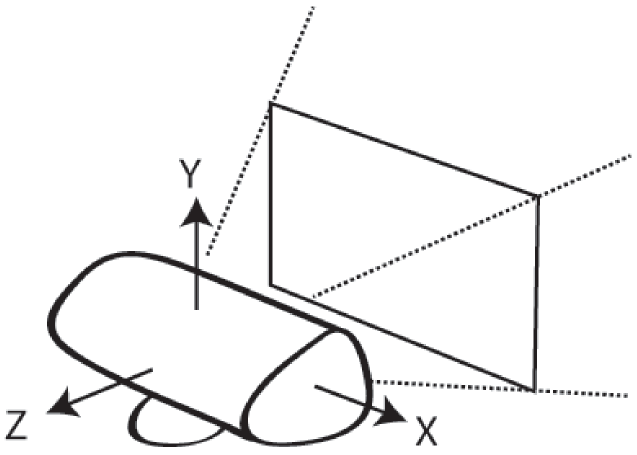
The 'freenect_sync_get_depth' function sets the pointer named 'depth' to point to an array of 640*480 16bit values. For performance reasons, we created a loop which can skip every 'n' points in X and Y. We control the resolution by setting the 'g_SampleDivisor' variable. If this variable is set to '2', then we only use every other point in x and y (1/4 of the total amount of points). We found that to get a good framerate on the Pandaboard, we had to set this variable to '5' (thus we used only 1/25th of the points). Below is the structure of the loop:

```
    int i,j;
    for (j = 0; j < 480/g_SampleDivisor; j++) {
        for (i = 0; i < 640/g_SampleDivisor; i++) {

            int sample_i = i*g_SampleDivisor;
            int sample_j = j*g_SampleDivisor;
            short d = depth[sample_i+640*sample_j];

            // Compute 3d point in 'kinect space'
            float x,y,z;
            Kinect_Data_To_XYZ(sample_i,sample_j,d,&x,&y,&z);

            // Transform point into world space
            float wx,wy,wz;
            Kinect_Point_Transform_To_World(x,y,z,&wx,&wy,&wz);

            // NOW DO SOMETHING WITH wx,wy,wz...
        }
    }
```

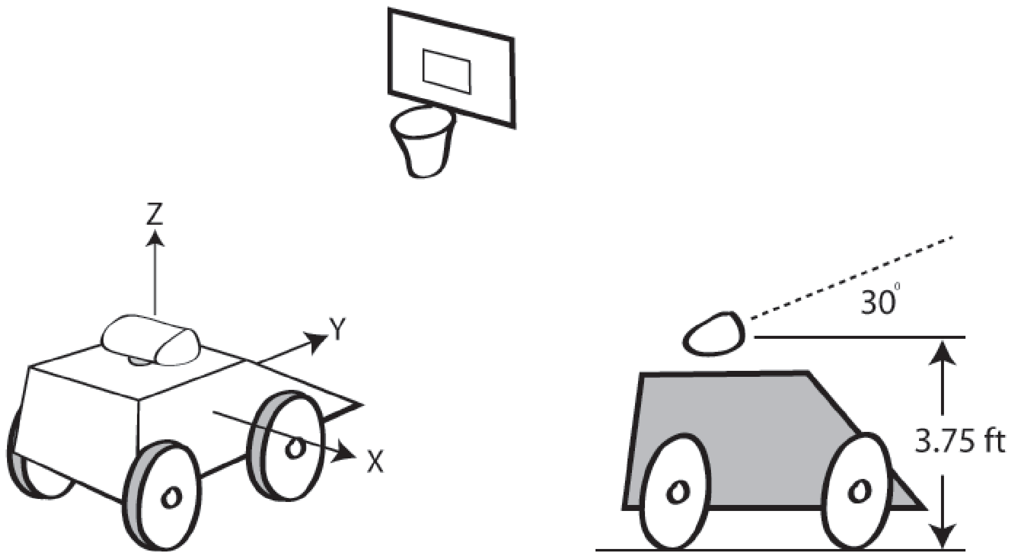
Next we show the code for actually computing the x,y,z position for a point given the pixel coordinate and depth. The formulas in this function are derived from the sample code's matrix for projecting the kinect data to 3D. We just pulled the numbers out of the matrix that the sample code uses so we could directly compute the 3D points in our code. The sample code uses OpenGL to do all of the calculations on the GPU which is not convenient for doing targeting logic.

```
void Kinect_Data_To_XYZ(int px,int py,int d,float * x_out,float * y_out, float * z_out)
{
    // Kinect projection formula
    float fx = 594.21f;
    float fy = 591.04f;
    float a = -0.0030711f;
    float b = 3.3309495f;
    float cx = 339.5f;
    float cy = 242.7f;

    float x = (1.0f/fx) * (float)px - (cx/fx);
    float y = (-1.0f/fy) * (float)py + (cy/fy);
    float z = -1.0f;
    float w = (float)depth * a + b;
    float oow = 1.0f / w;

    *x_out = x*oow;
    *y_out = y*oow;
    *z_out = z*oow;
    const float METERS_TO_FEET = 39.0f/12.0f;   // 39 in/m * 1/12 ft/in
    *x_out *= METERS_TO_FEET;
    *y_out *= METERS_TO_FEET;
    *z_out *= METERS_TO_FEET;
}
```

Using this function, you can calculate 3D points relative to the Kinect. The X axis represents left and right, the Y axis represents up and down, and the negative Z axis represents near and far. The fact that negative Z represents depth is a long-standing convention from computer graphics. The point (0,0,0) is the location of the Kinect. Our "kinect coordinate system" is displayed in the diagram below:



We next use the known mounting position and orientation of our Kinect to calculate the actual world positions of the points. For example, our Kinect is mounted on our robot about 45 inches (3.75ft) off the ground and it is tilted at 30 degrees. See the next diagram for our desired "robot coordinate system".



If you have a 3D math library, you can use a matrix transformation to do this calculation. For the purposes of this project, we'll just do the math by hand. If you carefully look at the figure with the robot coordinate system above, and compare it with the kinect coordinate system, you can see that we need to rotate the points about the X axis by 120 degrees (90 + 30) and then move the points up in Z by 3.75ft. The rotation formula below comes from linear algebra and can be found in many books on computer graphics. Keep in mind that these are just our conventions, once you understand the math there are other ways you can implement this.

```
void Kinect_Point_Transform_To_World
(
    float x,float y,float z,float *out_x,float *out_y,float *out_z
)
{
    const float KINECT_ANGLE = 30.0f;
    const float KINECT_HEIGHT = 3.75f;

    // Rotate the points about the X axis:
    float rads = (90.0f + KINECT_ANGLE) * 3.1415f / 180.0f;
    float c = cosf(rads);
    float s = sinf(rads);

    *out_x = x;
    *out_y = c*y - s*z;
    *out_z = s*y + c*z;

    // Translate the point up in Z:
    *out_z += KINECT_HEIGHT;
}
```

Using the above functions, you should be able to calculate an array of points in space relative to the robot. These points are on the surfaces of the objects that the Kinect could see. We found that you can use extremely simple algorithms once you have correctly calculated these points.

Targeting Version 1

Our first version of the Kinect targeting code was very simple:

- Calculate the point cloud
- Pick the closest point that is near 8ft off the ground

That's it! In the Rebound Rumble field, the closest point that is around 8 ft high, is the front of the rim. This algorithm doesn't even get confused by robots that are in front of you because all of their points are rejected due to their Z coordinate being below the rim height. For brief moments the kinect might see a basketball flying through the air or you might be very close to a wall but other than that, the closest point at 8 ft high is the rim. We experimentally determined the ideal RPM for our shooter wheel for various distances and entered those values into a table in our code. The cRio would read the Kinect distance, and interpolate an RPM value from that table. Our first competition in 2012 was at the Los Angeles regional. Most of our shots in this competition were controlled by this algorithm and we found that we could make shots from random places on the court..

One problem with this algorithm was that it is not well suited to aiming the turret. The "closest" point tends to jump around on the front edge of the rim by about 6-12 inches due to the noise in the distance readings.

It can’t be that easy! Targeting Version 2

In Las Vegas, our first algorithm did not work at all. For some reason, our Kinect data was much cleaner in LA. You could clearly see the entire rim of the basket in the point cloud in LA and in our shop.   However in Las Vegas and later in St. Louis, we could not see the rim at all. Our theory is that the lighting was different or maybe we had damaged our Kinect to some degree (we killed two of them during the season, more on that later). Anyway, in Las Vegas we developed a better algorithm which tracks the backboard rather than the rim.
The next algorithm worked like this:

- Calculate the point cloud
- Keep the points along the top of the backboard (height is near 10 ft)
- Average all of these points together to find the center of the backboard

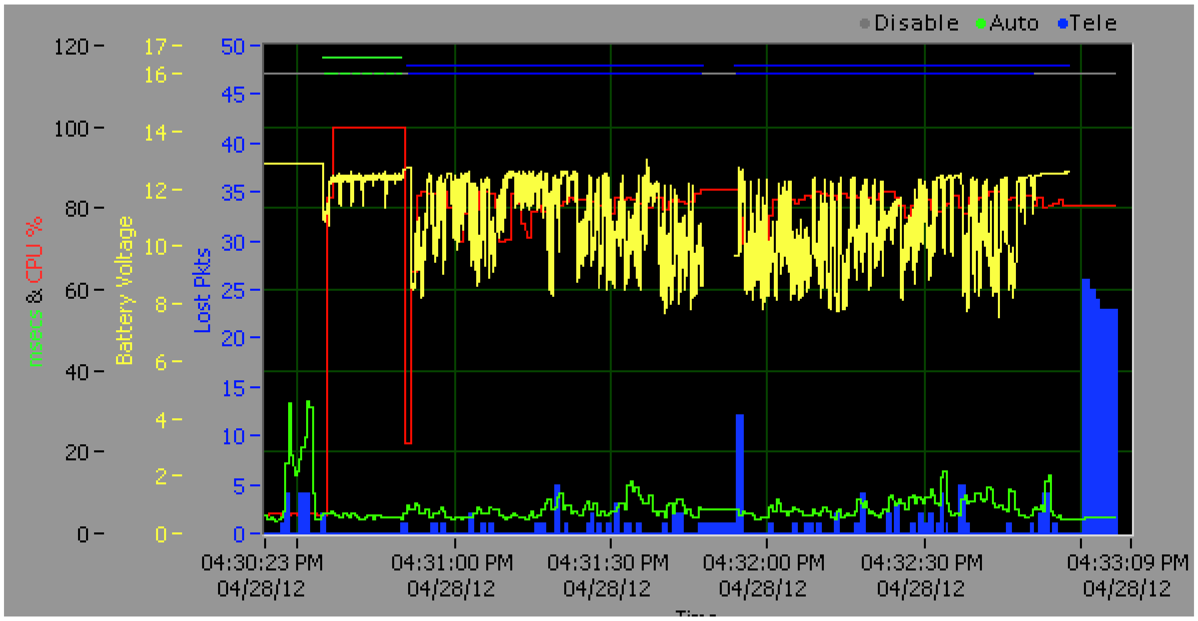
Since we’re incorporating data from a large number of points, this generates a very stable point at the top center of the backboard. With this algorithm we also implemented an automatic aiming function for the robot. In autonomous, the robot can aim and during tele-op the driver has a button he can hold to auto-aim.

Targeting Version 3

The final version of our targeting code added one more step. First we find the center of the top of the backboard. Then we divide the points into two groups, the set of points to the left of the center and the set that is to the right of the center. We find the center (average) of these two groups and now we have 3 points that form a line in 3 dimensions across the backboard. Using these points we can calculate the “bank” angle between the robot and the backboard.

Powering the Kinect and the Pandaboard

For this system to be integrated into an FRC robot, you need to provide clean stable power to the Kinect and to the Pandaboard. All FRC robots are powered by a 12V DC battery. First we’ll consider the Kinect. A Kinect requires a clean 12V, 1A power supply. Unfortunately you cannot simply connect it up to the power distribution board on your robot. The problem is that the battery voltage on an FRC robot sags significantly when you’re running all of those motors. For example, it is not uncommon to see the battery voltage drop close to 7V when the robot is running a lot of motors or has motors that are stalling. You can look at your driver station logs to see a graph of the battery voltage. It typically spikes up and down wildly between 7-12V. See the yellow plot in the diagram below for an example of the battery voltage during one of 987’s matches:



You really don't want to feed this into the Kinect (we tried!). To provide power to the Kinect, we used a DC-DC boost converter to boost the voltage up to 24V and then connected a 12V regulator to its output.

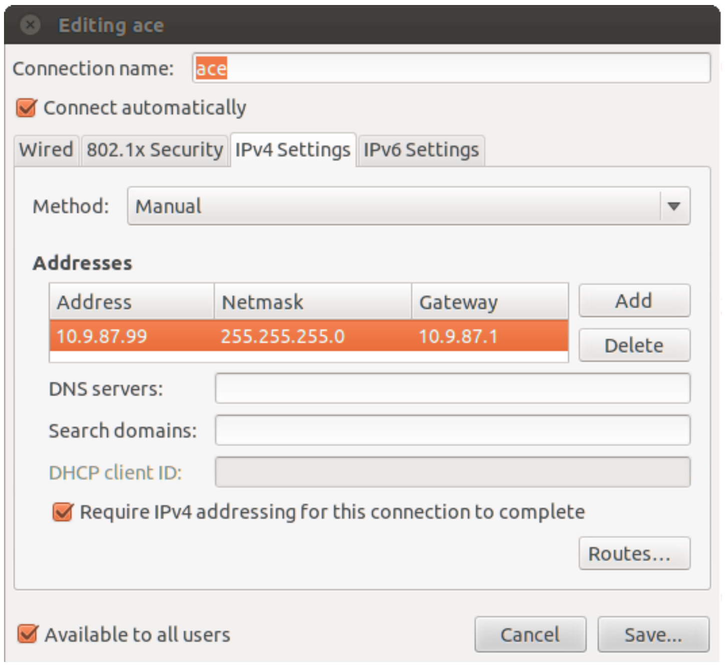
http://sumaoutlet.com/150w-car-dcdc-1032v-to-1235v-converter-boost-charger-p-2734.html

http://sumaoutlet.com/5pcs-4560v-lm2596hv-dc-voltage-regulator-power-converter-p-3474.html

The Pandaboard requires a 5V 2A power supply. This is a lot easier to solve because you can just connect it to the regulated 5V supply on the power distribution board. You just have to build a cable that can plug into the Pandaboard and has bare wires that can be connected into your power distribution board.

Connecting the Pandaboard into the Robot's Network

All FRC robots have a wifi router onboard which is used to connect the cRio and the camera to the driver station. You can simply connect the Pandboard into the same network using a short Ethernet cable. You will want to configure your Pandaboard's networking options to work on the robot's network. For example, we set our Pandaboard to use the 10.9.87.99 IP address. You need to set the netmask and gateway as well.



Connection name: ace

☑ Connect automatically

Method: Manual

Addresses

Address	Netmask	Gateway
10.9.87.99	255.255.255.0	10.9.87.1

DNS servers:

Search domains:

DHCP client ID:

☑ Require IPv4 addressing for this connection to complete

☑ Available to all users

Communication between the Pandaboard and cRio

To communicate data between the Pandaboard and cRio we chose to use network sockets programming. Network sockets are available on almost any platform and are the foundation of almost all network communication. We followed the information on this site:

http://beej.us/guide/bgnet/

We chose to use TCP/IP packets. Our Pandaboard program opens up a listening socket which accepts connections. Whenever another program connects to it, a simple protocol is used to request the latest targeting data from the Kinect. Essentially, the Pandaboard acts as the targeting server and the cRio connects to it as a client. These two programs send packets to each other in the form of very simple strings that are parsed using the C standard library function sscanf.

Starting the Targeting Server on Power-Up

We needed to make our targeting program program run right away when the Pandaboard is powered up. The reason for this is the Pandaboard is inside the robot with no screen, keyboard or mouse attached. Also, you don't want to have to manually launch the Kinect program every time you turn the robot on. To accomplish this we added a line to the .Profile file in our user directory that launches our program. For example:

```
xterm -e ./KinectVisionServer/bin/Debug/KinectVisionServer &
```

At this point, whenever we log in, the targeting program will automatically run. Next we set up Ubuntu to automatically log in by going into the system settings, then into “User Accounts”, and enabled the “Automatic Login” option on our user account.

Shutting Down the Pandaboard

The next puzzle to solve is how to safely shut down the Pandaboard. If you just cut power to the Pandaboard, it is possible to damage the filesystem on your SD card. We tried a “Mod” that we found on the internet which showed how to add a power button to the Pandaboard but in our experiments, it didn’t seem to do anything better than cutting power.

Next we tried using a process that involves logging in remotely using SSH. You can install SSH on your Pandaboard using the following command:

```
sudo apt-get install ssh
```

SSH will let you log into the Pandaboard remotely and then you will be able to use a command prompt interface on your Pandaboard. You can issue the ‘Shutdown’ command:

```
shutdown –h now
```

To use SSH, you can run a program on your driver station such as Putty. In Putty, we saved a configuration which uses SSH to log into our Pandaboard. The process to shut down was to open up Putty, log in to the Pandaboard, then type in the shutdown command.

By the end of the season, we took the shutdown process one step further. We added a network packet command to our kinect server program which tells it to shut down the computer. It then runs shutdown itself. This way you can put a button on your driverstation that the cRio code watches for and then sends the shutdown packet to the pandaboard.

Pitfalls to Avoid

- Get a really good SD card. We tried many that just would not work with the Pandaboard.
- As mentioned above, our Pandaboard was hard-locking frequently until we got the updated drivers from TI.
- Make sure you supply clean power to your Kinect and your Pandaboard.
- When the Kinect sees nothing at all (which happens frequently in an FRC match), it can sometimes take up to a couple of seconds to start sending valid data again. This can also happen if something is extremely close to the Kinect. We had to make our bridge autonomous take this into account; we can’t trust the targeting data until we get a couple of good updates from the Kinect after we’re back in range of the backboard
- Be careful not to constantly cut power to your Pandaboard by turning the robot off, you can corrupt the file system on the SD card.
- It would be nice to have more computing power. That way you could use more of the

points and you could start using more advanced point-cloud techniques.

- The extra complexity of a second computer in an FRC robot should be carefully considered. Many teams had great success this year using vision tracking without exposing themselves to this level of complexity.