USB Camera + Kangaroo + GRIP + RoboRIO (+ JAVA)

Scott Taylor
s-taylor@att.net
Mentor, FRC 1735 (Green Reapers)
Mar1, 2016

It seems that many teams are having difficulties getting a USB camera to connect to a Kangaroo co-processor running GRIP, and then sending usable data to the RoboRIO for vision processing.

Since we seem to have found a recipe that works, we're capturing it for others to use.

Table of Contents

Camera setup

This write-up used a Microsoft Lifecam 3000. We used it straight out of the box, without any modifications or software changes to it. We have not tested using any other camera type given the build season time pressures.

Lighting setup

We used the standard KOP green ring light (superbrightleds.com? Insert link here), but found it to be under-powered to overcome bright fluorescent lighting in our test area. We added a larger concentric light, and that seems to have provided enough unique reflected light to make the HSL (hue/saturation/luminence) filter work more consistently.

GRIP setup

We decided to use GRIP for several reasons:

1) Originally we wanted to experiment with vision processing directly on the RoboRIO to see if it actually had enough horsepower
2) RoboRealms only runs on Windows, so was not an option
3) We like supporting WPI and their efforts

Useful links:

- GRIP wiki homepage: https://github.com/WPIRoboticsProjects/GRIP/wiki
- GRIP releases: https://github.com/WPIRoboticsProjects/GRIP/releases
- GRIP streaming video extension for SmartDashboard: https://github.com/WPIRoboticsProjects/GRIP-SmartDashboard/releases

We downloaded the software from the releases page (We've used both the 32- and 64-bit versions successfully). This write-up assumes a minimum version of 1.3rc1, which is the first release to fully support all the requirements.

We started our GRIP usage on an available laptop for most initial development (more convenient screen/keyboard, primarily) but have done all the same steps on the Kangaroo with no changes.

Grip pretty much works out of the box; we followed the wiki tutorials to set up a basic pipeline of Image -> resize -> HSL Threshold -> Find Contours -> Filter Contours -> Publish Contours.
Specific settings at this time are:

- HSL lots of experimentation here; the second ring light really helped us filter down to "mostly" the target image. Hue 36-66; Sat 56-77; Lum 235-255
- filterContours: We played around with these a lot too when trying to eliminate non-target 'hits'. This was working for us but after adding the additional ring light, we might be able to back off on the max size settings. These settings were empirical for positions starting at the courtyard/outer works boundary up to the closest tower shooting distance we could manage (about 8' from tower face).

    - Min area 100; min perimeter 10; min width 20; max width 100; min height 25; max height 100. Surprisingly “Solidity” had a larger effect than we expected. We ran at 0-58.
- Publish Contours: We set the name to “StrongholdContours”. The robot code’s table lookup refers to this name. We left all items selected

Side note: We originally intended to test this on the RoboRIO, but found that the program crashed after only a minute or so. It turns out Grip was running out of memory (alloc from heap) even though it looked like there was plenty of free memory on the RIO (by logging in and examining the running machine and processes) and its image size was completely reasonable. We were fortunate to be on the WPI campus that night w/ the robot, and the developers dropped by the Robot Pits to help debug. GRIP was running just fine as long as the robot was booted but disabled. When enabled, enough objects were created by the robot code that the robot code memory was at 74% and initial debug indicated that GRIP was being limited in some way by the Java environment robot code that spawned it. Further debug has gone on since then, but an initial theory was that Java wasn’t accounting for the native libraries (like CV) that were eating memory.
That being said, if the memory issue is resolved, processing directly on the RIO may become a reasonable option again.

For the “settings” menu, the defaults that are created once you set the team name are just right—the NetworkTables address and the deploy address for us was roborio-1735-frc.local.
We saved the resulting GRIP file in the top level of our robot project at the same level as the build/src/bin directories. (note: We use RobotBuilder and Java programming). This allowed us to version control the file and to transfer changes between development systems and the kangaroo using our GitHub repository and GitHub desktop app.

Kangaroo Setup

Software and Config

This was some of the most difficult trial-and-error parts of the entire saga!

- Initial installation and SOFTWARE setup
    - We went through the original O/S setup screens, setting up an account but not a password (this enables the system to boot completely without user intervention
    - We hooked up to the local wireless network for software setup
    - We downloaded the FRC 2016 Update suite from NI in order to get mDNS and the USB device driver
    - We downloaded GRIP 1.3rc1 and installed it
    - We installed GitHub and then pulled our repository (including the GRIP configuration file)
    - We installed RealVNC (to take advantage of the Kangaroo’s nifty “local direct wireless” capability)
        - This is great for logging in to access the headless unit on the robot, but turning on that feature (by sliding the accessory switch) disconnects the Kangaroo from the USB and Ethernet networks, so we could never use VNC to debug the running system! We added a

        USB Hub and a wireless mouse/keyboard dongle, and hooked up to an external monitor for the bulk of the debug.
        - It might be possible to VNC in from another computer already on the robot radio wireless network, but we never got around to trying that. Hmmm…
- Network setup: We determined that things don't work properly if the kangaroo thinks the robot radio network is PUBLIC. Setting it to HOME allowed some critical messages to get through the system
- We had to turn off Windows Firewall completely. It might be possible to turn this back on once we figured out the HOME network trick, but didn't get to try it.
- To configure GRIP to run at startup:
    - Navigate to c:\Users\<user>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
    - In another file browser, navigate to the GRIP program.
    - Right click on the GRIP and drag it to the Startup folder.
    - Select "Create a SHORTCUT here"
    - Right click on the resulting shortcut, and edit the command line to specify two additional (space-separated) arguments of --headless (note: two dashes) and the path to the GRIP config file
    - We didn't add the 'headless' part until the very end, so that we could see it running on the external monitor while doing live debugging.
- The power button, by default, puts the Kangaroo to sleep. In order to turn it off without having a mouse/keyboard/monitor, you need to go into the power settings and change the switch to do a Shutdown instead!
    - Right click on the power icon in the system tray and select "power options"
    - Click "Choose what the power button does" on the left sidebar
    - Select "shutdown" for both "on battery" and "plugged in"

Kangaroo <-> Robot wiring

That gets us to the basic installation. For hooking to the robot:

- We originally used the USB connection so that we could use the extra Ethernet port for our Driver Station when in the pits.
    - However, we have found that "something" is killing the USB network connection after a short time. GRIP is still running, but an 'ipconfig' from the dos CMD prompt shows our USB connection is no longer present. We don't know what crashes it, but adding a USB-to-ethernet dongle and hooking to the radio port (the one farther away from the power plug; there are online posts that mention the RIO needs to plug into the port closer to the power plug!) allows the network connection to run stably.
- So, we have a USB connecton to the camera, a USB dongle for wireless mouse/keyboard, a USB connection for the Ethernet dongle, and occasionally a USB connection directly to the RIO when debugging that network connection. Yup, we had to add a USB hub to the robot to support all this. I bought a USB2.0 hub at Staples that worked.
    - Side note: since we ultimately had to use the radio's Ethernet port (see above) for the Kangaroo, we may need to add an Ethernet hub to the robot as well, since we prefer to hook the

    Driver Station to the robot via Ethernet while in the pits. We haven't gone any farther with that yet.

- We opted to plug the camera into the blue USB3.0 port and everything else into the USB2.0 port. Note that the documentation with the Kangaroo shows the 2.0 and 3.0 ports reversed in the diagram… the "ss" labelled port should be the USB3.0 from what we can tell.
- We tested using a USB connection from the RoboRIO to the microUSB of the Kangaroo, which is supposed to be only for trickle-charging the battery. It does look like the RIO was charging the Kangaroo, but it's not enough to hold the battery level when the Kangaroo is actually on (so is of dubious value since the robot isn't generally going to be powered up without having the Kangaroo also on!)

GRIP + Kangaroo bringup/debug

Standalone (Self-contained to Kangaroo)

At this point, you can test the Kangaroo + GRIP system:

- Run GRIP in GUI mode (may need to kill the startup headless version of GRIP if that is already running)
- Tune your pipeline until you see the contour you want, and it is (fairly) stable as the only contour.
- We found it useful when testing to get things noisy enough that waving things around in front of the camera changed the published contour or had some other visible effect on the NetworkTables output so that we could see if the connection was indeed alive (rather than just statically showing the last value received, which happens sometimes). If the values are all 100% unchanging, you may not actually be getting data!
- Save the GRIP file
- You can test the LOCAL loop by going into GRIP settings and setting the networkTables server address to "localhost" (without the quotes).
    - Then, assuming you downloaded enough robot software to have the wpilib stuff locally, you can navigate to c:\Users\<user>\wpilib\tools and run OutlineViewer. Use an address of "localhost" and run a SERVER (to substitute for the RIO—not the client, which is what you will normally use and point to the RIO). It should see the values published by GRIP. This is also helpful for making sure your path to the table is correct (NOTE CASE SENSITIVITY!!!). In our installation, the path was Root -> GRIP -> StrongholdContours.
    - With GRIP running on the kangaroo, the values should be changing constantly. If you have the value lines (Area, width, height, etc) but nothing for a value (e.g. denoted by empty square brackets [] and a type of "Number[0]" for each line), you have likely lost or failed to make your NetworkTable connection… or possibly are not detecting a contour anymore—go look at the GUI and see if FilterContours is indicating a detected contour. If not… you won't get anything published!
    - Once you have this part of the loop worked out, we can add the RIO into the loop

Kangaroo + RIO bringup

- Adding the RIO

- Make sure the robot is booted and you are connected via enet or usb (Again, USB didn't work for us at the time of writing)
- First, verify network connectivity: on the Kangaroo, bring up a command prompt (search for "Cmd" and run it)
- Ipconfig should give you a valid IPV4 address.
    - For us, if connected to the USB, it was 172.11.22.1
    - If connected to the radio Ethernet, it was 10.17.35.56 (note that 17.35 is our team number)
    - If .56 doesn't work for you, you may need to log into the radio to find out what address was assigned. Send a web browser to 10.te.am.1 and enter the user/password for the radio (both are "root" if you didn't change the defaults). Scroll down to see the list of assigned DHCP leases and choose the address for the machine you are seeking
- Change the GRIP NetworkTable settings to go back to the RIO (Set the server address to roborio-<team number>-frc.local
- From the cmd prompt, ping the RIO ("ping roborio-<teamnumber>-frc.local") and see valid responses.
- Kill the Outline Viewer app running on the RIO
- Restart the app, but select the server as roborio-<teamnumber>-frc.local, and start the CLIENT instead of the server
- You should see the same GRIP data being published, but now it is also accessible to robot code!

Kangaroo + RIO + Driver station bringup

- Next, check to see if the data is visible from the DriverStation computer or some other computer connected to the robot network (Wirelessly or wired): Just run OutlineViewer on that machine (Point to roborio-<team>-frc.local, run the CLIENT), and you should see the same data that you were seeing on the Kangaroo. This should prove that all the networking is running properly everywhere
    - Important note: This is where we had the most debugging to do—getting the tables to show up everywhere. Sometimes we could see things from the Kangaroo but not from other systems. We found that you had to set the appropriate computer's connection to be HOME rather than PUBLIC on any system that was trying to read the networktables. (thanks to whoever posted that tidbit to Chief Delphi!!). As mentioned before, the Windows Firewall also played a part in blocking traffic and we currently have it disabled.

Adding the Robot Code

Finally we are now at a point where we can get the robot to access the vision data.

We program in Java, using RobotBuilder to create the framework, and then filling in the user code in eclipse. There's no reason to think other methods/languages won't work, but we haven't tried testing them!

Robot.java

- Make sure you aren't running CameraServer—there are ChiefDelphi posts indicating it won't work if CameraServer and GRIP are accessing the same camera device. This is probably a holdover in our code from when we were trying to run GRIP on the Rio. If we hooked up a separate camera directly to the RIO, our research indicates that we could safely connect the CameraServer to it, and things would work—key is to not have two apps access the same camera

- In **public class** Robot **extends** IterativeRobot {
  - Add these lines (Yes, you need to add a constructor function!). Also note that the "StrongholdContours" name must match what you put in the Publish widget in the GRIP pipeline, and that the string is case sensitive.

```
// Get the network table for vision processing (via GRIP)
NetworkTable table;
public Robot () {
// Grip processes image data and produces specific informaton in the NetworkTables object
table = NetworkTable.getTable("GRIP/StrongholdContours");
}
```

- We put our processing code in a RobotBuilder-created subsystem called "Vision" so the remaining code will reflect that choice.
- *IF* you wanted to run GRIP directly on the RoboRIO, there are instructions on the GRIP wiki for doing this. You have to invoke GRIP from the robot program (Which may be the cause of the memory issue as both GRIP and the robot may be running with the same heap? Not sure), but for completeness, you would add this code to the end of your `public void robotInit()` function:

```
// Vision Processing:
/* Run GRIP in a new process */
try {
new ProcessBuilder("/home/lvuser/grip").inheritIO().start();
} catch (IOException e) {
e.printStackTrace();
}
```

- If you are only running GRIP during autonomous, you may wish to avail yourself of the undocumented "Pause" feature to stop processing after autonomous (Also found in Chief Delphi post at http://www.chiefdelphi.com/forums/showthread.php?t=144070&highlight=grip+smartdashboard). We haven't coded/tested this yet, but you would put this code into the appropriate places to stop/start… perhaps just adding the stop to your teleopInit() function:
  - Add a second table variable in Robot class (or maybe you could just change the initial one to start at GRIP, and add variables elsewhere to specify the StrongholdContours path?)

```
// Get an additional network table top-level for GRIP control
NetworkTable gripTopTable;
```

  - And in teleopInit(), add this code:

```
gripTopTable.putBoolean("run", false); // Stop GRIP without killing the process
```

  - Optionally in other code, you can re-enable GRIP like this:

```
Robot.gripTopTable.putBoolean("run", true); // Start GRIP again
```

Vision.java

- In Vision.java, we process the remaining information. For completeness, it's easiest just to copy the entire file contents here.

- Reviewing the code, we used a local table variable rather than the one in Robot. That should probably be changed so that we don't have two pointers lying around, but certainly doesn't hurt anything.
- We really did hit cases where consecutive calls to getNumberArray() returned different sized arrays—indicating the NetworkTables asynchronously updated in between the consecutive lines of JAVA code. This CRASHES the robot code if you are accessing an index that doesn't exist in all variables, so we added some brain-dead code to check and punt if necessary. This isn't exactly the most desirable or elegant solution, but it got us past the crashes and working on the robot again. Cleanup will be a post-competition task for the students!!
  - It also doesn't cover the case where the table changes but stays the same size—you could end up with area and X position referring to two different objects! We don't currently attempt to account for this, and are burying our heads in the sand hoping that one iteration of the PID loop won't go insane for this corner case. Adding an atomic "get" of the entire table would solve this, but not sure if that exists in the current implementation.

- Full code for vision.java:

```
// RobotBuilder Version: 2.0
//
// This file was generated by RobotBuilder. It contains sections of
// code that are automatically generated and assigned by robotbuilder.
// These sections will be updated in the future when you export to
// Java from RobotBuilder. Do not put any code or make any change in
// the blocks indicating autogenerated code or it will be lost on an
// update. Deleting the comments indicating the section will prevent
// it from being updated in the future.


package org.usfirst.frc1735.Stronghold2016.subsystems;

import org.usfirst.frc1735.Stronghold2016.RobotMap;
import org.usfirst.frc1735.Stronghold2016.commands.*;

import edu.wpi.first.wpilibj.command.Subsystem;
import edu.wpi.first.wpilibj.networktables.NetworkTable;


/**
 *
 */
public class Vision extends Subsystem {
	NetworkTable table;
	public static final double xRes = 320; // this is the maximum resolution in pixels for the x
(horizontal) direction

   // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

   // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
   // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

   // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

           // create a new constructor
           public Vision() {
   // Get a pointer to the networkTable.  "StrongholdContours" is the name we entered into the publish
box in GRIP
   table = NetworkTable.getTable("GRIP/StrongholdContours"); // TODO:  Grab the pointer we created in
Robot.java?
           }



   // Put methods for controlling this subsystem
   // here. Call these from Commands.

   public void initDefaultCommand() {
      // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND


   // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND

      // Set the default command for a subsystem here.
      // setDefaultCommand(new MySpecialCommand());
   }

   public double getRawTargetXpos() {
           double xPos; // return value

           // 1) Get the current list of targets found.  There might be more than one visible at a time
           // Firststep: get the vision system data for the target
           double[] defaultValue = new double[0]; // set up a default value in case the table isn't published
yet

           // Get all needed table items at the same time, in case the table updates between reads.
           // (could end up with different array sizes)
           double[] targetX = table.getNumberArray("centerX", defaultValue);
                   double[] areas = table.getNumberArray("area", defaultValue);
                   if (targetX.length != areas.length) {
                           // here the table updated in the middle; we'll have to punt
                           System.out.println("NetworkTable udpated in the middle of getRawTargetXpos;
returning first valid entry");
                   if (targetX.length==0)
                   {
                           // we didn't find ANY object.  Return a perfectly centered answer so that the
system doesn't
```

```
                        // try to adapt
                        xPos = xRes/2;
                }
                else xPos = targetX[0];
                return xPos;
                }

        // For initial debug, just print out the table so we can see what's going on
/*
        System.out.print("centerX: ");
        for (double xval : targetX) { // for each target found,
                System.out.print(xval + " ");
        }
        System.out.println();
*/

        // 2) Choose the one that has the largest area.  This is PROBABLY the closest target (and most in-
line)
        //    Don't want to choose the one closest to the center because that might actually be the target
        //    for a different face that's very oblique to our robot position.
                int largestIdx = 0;
        if (targetX.length > 1) {
                double largest = 0;
                for (int c = 0; c < areas.length; c++) {
                        if (areas[c] > largest){
                                largest = areas[c];
                                largestIdx = c;
                        }
                }
        }

        if (targetX.length==0)
        {
                // we didn't find ANY object.  Return a perfectly centered answer so that the system
doesn't
                // try to adapt
                xPos = xRes/2;
        }
        else xPos = targetX[largestIdx];

        return xPos;
  }

  public double getScaledTargetXpos() {
        // get the raw position
        double raw = getRawTargetXpos();
        // Scale the resolution
        // Find out how far (magnitude and direction) off-center that target is (the "error")
        //      positions in the NetworkTable are in pixels relative to the camera resolution, with (0,0)
being the upper left corner
        //      and (resolutionx,resolutiony) being the bottom right.  We would want to convert to a
different scale
        //      where 0,0 is center, and the extents are -1 and +1 (like a joystick input!)
        //       The equation for this (see "identifying and processing the targets" under Vision on the
Screensteps pages)
        //              Aim = (Pixel - resolution/2)/(resolution/2) for each of x and y direction
        double scaled = (raw-xRes/2)/(xRes/2);
        return scaled;
  }
}
```

Further processing of Vision data

At this point, you should be able to see/print/use the vision data produced by the system. We developed a PID drivetrain subsystem that used a ScaledXPos==0 as the setpoint, and drove the motors to get to that setpoint (P,I,D are still/always being fine-tuned, of course!).

Summary

It *is* possible. It’s a bit painful to set up and debug, but we had the robot successfully tracking a student holding the target and walking around the room. And if the robot was not aimed at the target (but it was within the camera view), enabling the drivetrain PID would swing the robot to center on the target.

If you have any further questions, feel free to contact me (the Electrical/Programming Mentor, Scott Taylor) at the email address at the top of this document.

Good luck!