AR.Drone Libraries for the CBC Botball Controller (Part 2)
Jeremy Rand, Marty Rand, Kevin Cotrone, Ali Hajimirza, Josh Maddux
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy.rand@ou.edu, marsedge95@gmail.com, kevincotrone@gmail.com, a92hm@ou.edu, jdmaddux@gmail.com

AR.Drone Libraries for the CBC Botball Controller (Part 2)

11 Welcome to Part 2!

Welcome back. And now, on with Part 2!

12 Reading Video Frames

Pongsak's library is able to read video frames from the Drone's cameras. However, the framerate was extremely low when we tested it (less than 0.25 frame/s). We used the GNU Profiler [10] to figure out where the bottleneck was, and determined that libcommonc++'s multithreading features are defective. Specifically, locking mutexes takes way longer than it should (we believe that if Thread A attempts to lock a mutex already locked by Thread B, and Thread B unlocks and immediately locks again the mutex, Thread A will not always receive the mutex). To fix this, we simply eliminated the mutex and instead used two buffers for the video frame (so that only one buffer can be receiving data at a time; the other one is safe to read). This boosted framerates to well over 1 frame/s (in some cases over 4 frame/s). The modifications are on our GitHub [11]. Approximately 2 seconds of latency are present, which is annoying but manageable.

Once our patches are applied to the video receiver, the following code will receive a video frame and write it to a flash drive as "`DroneRGB24.rgb`":

```
VideoDecoder::Image * videoData = new VideoDecoder::Image();
long timestamp;

printf("Image created\n");

myDrone->videoDataReceiver().copyDataTo(*videoData, timestamp);

printf("Copied data\n");

printf("Dimensions: %d by %d\n", videoData->width, videoData->height);

FILE * rgbOut = fopen("/mnt/browser/usb/DroneRGB24.rgb", "w");

fwrite (videoData->data , 1 , sizeof(videoData->data) , rgbOut );

fclose(rgbOut);
```

```
system("sync");
```

The resulting file is RGB24, and can be viewed in GIMP as a RAW file. Below is the first image we dumped from the Drone's front camera, taken in a conference room at the OU Engineering Practice Facility:

Instead of writing the image to a file, processing can be done directly on the data; the pointer to the RGB data is in `videoData->data` .

To switch between the front and downward cameras, the following commands can be used:

```
myDrone->controller().switchToFrontCamera();
myDrone->controller().switchToDownCamera();
```

Keep in mind that the camera change is delayed just like the video itself. The dimensions of the image will tell you when it has successfully switched (the front camera is 320x240, while the bottom camera is 176x144):

```
if(videoData->width >= 320)
{
      // Front camera is active
}
else
{
      // Down camera is active
}
```

13 Vision with `mb_vision`

Raw video data is a great first step, but we wanted blob tracking. One option was to feed the video into the CBC firmware's blob tracker. Unfortunately, we didn't have the ability to recompile recent CBC firmwares, so we considered what our other options were. OpenCV [4] was considered, but we decided that OpenCV was overkill, and too difficult to compile.

We instead opted to use mb_vision, a vision library written by Matthew Thompson for Nease Robotics Club in 2010 (which propelled them near the top of the rankings and earned them a Judges' Choice Award). After GCER 2010, when his entire team graduated from the Botball scene, Matthew released mb_vision on GitHub [12]. Matthew had originally used mb_vision with CBCLua [13], but luckily he had made sure that the C++ vision code was neatly segregated from the Lua bindings. We copied the C++ code verbatim and started trying to integrate it with the Drone.

As it turned out, Matthew had done an excellent job; we were able to make his code work with almost no modifications. All we did was remove the CBC camera code, and reverse the order of the subpixels (the CBC camera is BGR, while Pongsak's library returns RGB). After doing this, we placed an mb_vision folder in the root of our project directory and added the .cpp files to our project in Code::Blocks.

After the following #includes:

```
#include <vector>
#include "../mb_vision/BlobImageProcessor.h"
#include "../mb_vision/RGB24Image.h"
```

We were able to use the following code to look for green objects with the front camera:

```
long timestamp = 0;

vector<Blob> blobs_found;

VideoDecoder::Image* visionImage = new VideoDecoder::Image();
RGB24Image* visionImageWrapper;

visionImageWrapper = new RGB24Image(320, 240);

// These arguments are reasonable defaults; see mb_vision docs
BlobImageProcessor* blobTracker = new BlobImageProcessor(10, 8, 8, false);
ColorModel *model;

model = new ColorModel(90, 180, 64, 64); // Hmin, Hmax, Smin, Vmin
blobTracker->addColorModel(model);

myDrone->videoDataReceiver().copyDataTo(*visionImage, timestamp);

visionImageWrapper->setBuffer(visionImage->data);

blobTracker->processImage(*visionImageWrapper);

blobs_found = blobTracker->getBlobs();

printf("%d blobs!\n", blobs_found.size());

delete blobTracker;
delete model;
delete visionImageWrapper;
delete visionImage;
```

If blobs were found, the bounding box of each blob can be read using the x, y, w, and h properties of each blob. For example, the width of the third blob can be read by:

```
blobs_found[2].w
```

Remember that the blobs returned are a C++ vector (dynamically sized array). **This means that if you access a blob that is higher than the number of blobs found, your program will immediately segfault and terminate.** As such, always check `blobs_found.size()` before accessing blobs.

You'll notice that the bounding box is the only data we receive from the blob tracker (no centroid, size, confidence, or elliptical fit). In practice, this is not a problem in Botball situations (the bounding box is sufficient).

14 Color Models

If you carefully read the above code, you'll notice that we manually set the HSV parameters of the color model. In some cases, it is desirable to import them from the CBC's vision system, which reduces trial and error. We attempted this, but found that KIPR's API for retrieving color models from the CBC vision system is defective (it returned random integers which weren't even in the correct range). We first noticed this problem at around midnight the evening of open practice, so without time to fix it, we reverted to manual color models. Hypothetically, we could have developed a UI to change color models in real-time, but it wasn't worth it. (The targets in question were easy to see.)

15 AAV Contest Aftermath

After the AAV Contest, KIPR contracted us to port these libraries to a more user-friendly system which could be distributed to teams in future years (including schools without hacking knowledge). He requested that flight and position tracking be integrated into libcbc (the official CBC library), and that the vision system be integrated into cbcui (the official CBC vision system). We will now discuss how this integration was done.

16 libcommonc++ Integration

The compile scripts were modified to link all C programs with libcommonc++. This was difficult, because the gcc linker cannot be used to link C code with C++ code (the g++ linker is needed). However, C code cannot be compiled with g++. The solution was to compile with gcc, output a .o file, and then link with g++. This is the standard method used to link C with C++, but many Botballers may be unfamiliar with it since both Interactive C and KISS compile and link at the same time.

Initially, it was attempted to use the statically linked version of libcommonc++, since KIPR's firmware doesn't play well with shared libraries. Unfortunately, it appears that some features of libcommonc++ (unused in Pongsak's library) do not build on ARM architectures such as the CBC. Shared libraries worked because those features are only tested during compile if static linking is used.

To implement shared libraries on the CBC, the compile script was modified to produce a robot.bin file containing the program, and a shell script (called "robot", the same name as a typical compiled CBC program) which setup the shared library and booted robot.bin. This is basically what we did with the bootloader.c program previously, but in a somewhat more user-friendly form (multiple programs can use the same library, and the program name isn't hardcoded). The downside of this method is that the processes running will be ash (the Linux shell) and robot.bin, not robot. At this time the long-term stability of this system due to the use of different processes is untested, but we have not run into any issues thus far.

17 libcbc Integration

A simple C wrapper library was constructed to insulate the user from the complexities of Pongsak's C++ library. In particular, the watchdog and XYZ tracking are now handled automatically, and no knowledge of the underlying structs is necessary to use the library.

18 cbcui Integration

The final problem was integrating the Drone cameras with the cbcui vision system instead of using mb_vision. Major modifications to cbcui were undesirable, because cbcui has a habit of crashing when too much stuff is modified (we're not really sure why). The absolute minimum modification necessary was to make cbcui's Microdia Camera source check if a buffer file in /tmp/ was present, which would contain camera data. If this buffer file was present, cbcui would ignore the USB camera and instead wait for a "ready" flag to be set in /tmp/. When the flag was set, indicating that a full frame had been written by the Drone library, cbcui would read the buffer file and process it as though it were from the USB camera. It would then delete the "ready" flag and wait for a new frame.

The Drone library would, in turn, read images from the Drone, convert them to 160x120 BGR, and write them to the buffer file. After writing an image, it would set the "ready" flag, and wait for it to be deleted by cbcui, indicating that it was ready for a new frame.

The advantage of this approach is that cbcui can be switched between the USB camera and the Drone camera by simply creating or deleting a temporary file. No memory mapping or anything else complex is required. Color models and blob tracking work exactly as they do with the USB camera; the CBC vision extension library by Adam Farabaugh and Evan Wilson [14] would probably work with no modification whatsoever (we haven't tried this).

Upon initial testing, things seemed to work except for a slow memory leak in cbcui. This caused immense confusion since the added code doesn't allocate memory, and it was far too slow (taking several hours before a crash) to be from the video data being transferred.

The “facepalm” moment occurred when Jeremy realized that the “memory leak” was the KISS-C printf calls with debug data eating up the cbcui print buffer. No problem after all.

19 Packaging

Attempts to package up the C wrapper into a firmware image failed. Until this is resolved, the C wrapper and associated files are packaged into a C-based flash drive installer (much like the Norman/Nease CBC Mod Installer from 2009 [15]). The fun trick is how we replace cbcui (without killing our installer at the same time): we launch a shell script which *then* kills cbcui and copies files. Our C program is killed with cbcui, but the shell script survives and continues installation. After the files are copied and the filesystem is synced, the new cbcui is booted. Installation is much faster than a full firmware install, but must be redone after a new firmware is flashed.

20 The Current State of Things

Newbie users will probably use the C wrapper with the flash drive installer, and never touch libcommonc++, CodeBlocks, or any C++ code. Power users (i.e. hackers) will probably use the C++ libraries, but they won’t have to mess with cross-compiling or most of the things in this paper (unless they want to). However, using the C++ libraries will probably not be supported by KIPR (and it will require a modification to the C++ compile script, since only the C compile script links with libcommonc++ by default). The hackers who need this will probably know how to do it themselves.

21 Conclusion

With any luck, this paper has explained how the Drone libraries for the CBC were developed, to the point where other teams could do something of similar scope if necessary. We hope you enjoy the libraries.

Many things were not covered by this paper, including the new AR.Drone 2.0, and the possibility of doing some of these operations on-board the Drone. We’re very interested in these subjects, so if you’d like to collaborate on these subjects, or anything else regarding the AR.Drone, please don’t hesitate to contact Jeremy on the Botball Community Forums. [1]

Happy Hovering!

22 References

[1] Botball Youth Advisory Council. Botball Community. http://community.botball.org , June 2012.
[4] OpenCV. http://opencv.willowgarage.com/wiki/ , May 2012.
[10] Martyn Honeyford. Speed your code with the GNU profiler. https://www.ibm.com/developerworks/library/l-gnuprof.html , April 2006.
[11] Jeremy Rand. cbc. https://github.com/JeremyRand/cbc/tree/drone/userlib/libcbc/src , May

2012.
[12] Matthew Thompson. Botball-2010. https://github.com/matthewbot/Botball-2010/tree/master/mb/vision , August 2010.
[13] Matthew Thompson. CBCLua: Bringing Lua Scripting to Competitive Robotics (Parts 1 and 2). Proceedings of the 2009 Global Conference on Educational Robotics, July 2009.
[14] Adam Farabaugh and Evan Wilson. The Camera: Botball's Most Underrated Sensor. Proceedings of the 2011 Global Conference on Educational Robotics, July 2011.
[15] Jeremy Rand, Matt Thompson, Braden McDorman. Hacking the CBC Botball Controller: Because It Wouldn't Be a Botball Controller if It Couldn't Be Hacked. Proceedings of the 2009 Global Conference on Educational Robotics, July 2009.