Acknowledgements

Thank you to the 2009 rUNSWift team: Aaron, Adrian, Amy, Andrew, Bryan, Hung, Jayen, & Mitch; and to the academics who supervised and guided us throughout the project: Morri, Claude, and Bernhard; and to Brad, for always supplying pizza, making sure we all got on the right flights, and taking care of our every need whilst at the competition in Austria. Without you all we would not have had this fantastic opportunity to learn so much and participate in this world-class tournament. We have all had a fantastic experience, and hope that our small contribution will be helpful in future years.
## Contents

1 Overview 1
   1.1 Background ..................................................... 1
   1.2 Document Overview ............................................ 1

2 Code Architecture 1
   2.1 Blackboard Shared Memory Interface ............................ 2
      2.1.1 Motivation ................................................. 2
      2.1.2 Operations and Types ...................................... 3
      2.1.3 Concurrency Considerations ................................. 4
   2.2 Thread Manager .................................................. 5
      2.2.1 Perception Thread ......................................... 5
   2.3 Communication ................................................... 6
      2.3.1 Blackboard Server .......................................... 6
      2.3.2 Remote Control Protocol ................................. 6
      2.3.3 Localisation Streamer ..................................... 6
      2.3.4 Inter-Robot Communications .............................. 6
      2.3.5 Game Controller Receiver ................................. 7
      2.3.6 OffVision ................................................. 7
   2.4 Vision ............................................................ 7
      2.4.1 Background ............................................... 7
      2.4.2 Overview of the 2008 Vision Module ...................... 8
      2.4.3 Overview of the 2009 Vision Module ...................... 8
      2.4.4 Colour model ............................................. 8
      2.4.5 Colour Classification ...................................... 8
      2.4.6 Natural Lighting .......................................... 10
      2.4.7 Line Segment Detection .................................. 10
      2.4.8 Zero Detection ........................................... 11
      2.4.9 Horizon .................................................... 13
      2.4.10 Goal Detection ........................................... 13
      2.4.11 Ball Detection ............................................ 14
      2.4.12 Sanity Checking .......................................... 14
      2.4.13 Camera Settings .......................................... 15
      2.4.14 Camera Interface ......................................... 15
1 Overview

1.1 Background

Robocup is an international robotics tournament/competition that is held annually throughout the world, it aims to promote the development of Artificial Intelligence and Robotics through the medium of soccer. The competition consists of different leagues in which many different universities compete, the competition mirrors that of the FIFA World Cup in which teams play group stages and then top teams are selected into knock out rounds. rUNSWift has been participating in this competition since its inception in 1999 and remains the most successful team in the 4 legged league (which has been superseded by the Standard Platform League (SPL) as of 2008) with three world championships (2000, 2001, 2003), three second placings (1999, 2002, 2006) and a third placing (2005).

The SPL league focuses on software development where all hardware within the league is standardised, this year signifies the second year in which the Alderbaran Nao has been used as the standardised robotics platform within this league. Prior years utilised the Sony Aibo as the standardised platform, this represents a shift from an inherently stable quadruped robot to an unstable bipedal humanoid robot. This year’s competition utilised the third generation of the Alderbaran Nao, this was a much more reliable and durable platform compared to last year’s second generation Nao. In this year’s competition, held in Graz, Austria, despite a strong showing during the initial matches rUNSWift was unfortunately knocked out of the first rounds of the competition.

1.2 Document Overview

The primary audience of this report is the rUNSWift 2010 team, in a hope they can learn from our experience in the 2009 Robocup Competition. This report will outline the overall code architecture, and specific implementation details of the rUNSWift 2009 team’s code, as run in Graz. We hope to explain the design decisions that were made; then, in hindsight, discuss what worked and what did not, so that future teams may learn from our experience.

The first section of this report describes the overall architecture of the rUNSWift 2009 code base, and how it interacted with the Nao Humanoid robot, manufactured by Aldebaran. The later sections describe the design and implementation of each of the subsystems in this design, with a particular emphasis on the vision, communication and behaviour modules. The final section describes our experience at the competition in Graz, in particular the ‘challenge’ competition which aims to accelerate the rate of innovation in Robocup by providing specific technical challenges.

2 Code Architecture

Figure 1 describes the overall design of the rUNSWift software architecture, which is based on the same design used for Robocup in the Aibo Standard Platform League by rUNSWift teams of previous years.

The image scanning and odometry modules are the key inputs to the system, supplemented by supporting information transmitted over wireless by other robots. These inputs are processed into simple observations that the localisation module can filter and provide to the behaviours, which in turn direct the locomotion module to actuate the robot. The vision module is described in detail in Section 2.4, and the behaviour section is described in detail in Section 2.5.
2.1 Blackboard Shared Memory Interface

2.1.1 Motivation

In order to implement the aforementioned system architecture, it was necessary to develop a protocol for the various modules to communicate. Our solution to this was the ‘Blackboard’, a custom-built global namespace manager compiled into the rUNSWift broker.

NaoQi, the hardware abstraction provided by Aldebaran, contains a module named ALMemory, designed specifically for this purpose. In fact it is required that this module is used to communicate with other parts of NaoQi, such as when reading sensors and writing to hardware devices such as the LEDs.

ALMemory supports several basic operations, including:

- Writing single data items into ALMemory
- Writing a list of data items into ALMemory
- Reading single data items from ALMemory
- Reading a list of data items from ALMemory
- Busy-waiting for a change on a value in ALMemory
- Subscribing to changes of a particular value in ALMemory via a callback function
All reads and writes to ALMemory are thread-safe and guaranteed to be atomic, and considering the necessity of using this interface to talk to other NaoQi modules anyway, it sounds as if this would have been a wise choice for sharing data between various modules of the rUNSWift code base.

The primary reason for developing this customised ‘Blackboard’ interface in lieu of ALMemory was performance. The rUNSWift broker is a shared-object library that is dynamically loaded at runtime by NaoQi, and communications between the broker and any NaoQi modules take place through an ‘ALProxy’ which is a language-independent interface for making callbacks to NaoQi, by specifying the NaoQi module and method names as strings in calls to ALProxy. The overhead of these conversions greatly contributes to extremely poor run-time performance. See Figure 2.

The conversion process involved in making a call from the rUNSWift broker into a NaoQi module

![Diagram of conversion process](image)

Figure 2: The conversion process involved in making a call from the rUNSWift broker into a NaoQi module

The poor performance of ALMemory created a need for a high-speed data sharing interface within the rUNSWift broker itself.

### 2.1.2 Operations and Types

The Blackboard was designed with the specific operations needed for communication between the various modules and threads of the rUNSWift broker. The supported operations are:

- Write a component to the Blackboard
- Write n bytes from a file descriptor to the Blackboard
- Write an item to a queue on the Blackboard
- Read a component on the Blackboard
- Pop an item from a queue on the Blackboard
- Connect PRE/POST READ/WRITE hooks to a Blackboard component

To reduce code verbosity, each of these operations was implemented with a macro that wraps the various methods of the Blackboard class. Some sample use cases of the Blackboard macros follow:
Sets the 'ballDistance' component to the value 42

valueToBlackboard(ballDistance, 42);

Dereferences the const pointer returned from readFromBlackboard and stores result in a local variable

```
int ballDistance = *readFromBlackboard(ballDistance);
```

Call the function calculateBallAngle(ballDistance) after the ballDistance component is written to the Blackboard

```
addHookToBlackboard(ballDistance, calculateBallAngle, WRITEHOOK, POSTHOOK);
```

For more details on the blackboard interface see Appendix A

Since the readFromBlackboard always returns a const pointer to the component type, any code written involving the Blackboard is guaranteed to be type-safe.

Note that the inclusion of hooks would allow the Blackboard to be used as a framework for writing event-driven code, however this year it was only used as a global namespace for reading/writing data that other modules may be interested in.

### 2.1.3 Concurrency Considerations

One significant distinction to be made between the rUNSWift Blackboard implementation and the ALMemory module, is that the Blackboard is inherently not thread-safe. Safety is guaranteed by convention that no module writes any Blackboard component that is not its own.

One caveat of using the blackboard is that reads of structures greater than 32 bits are not guaranteed to be atomic, and as a result, in rare circumstances garbage data may be read. Furthermore, reads of multiple interdependent variables may result in mismatched data if a sequence of reads is pre-empted by a write in another thread. All possible mismatched reads in the current version of the software are considered insignificant, within acceptable limits, but this issue may need to be addressed in the future.

The queue structure, however, is atomic. It uses the boost::mutex primitive from the Boost Synchronization library to ensure atomicity of the enqueue and dequeue operations.

If other, more complex data types are to be stored on the Blackboard, concurrency primitives will need to be utilised, perhaps by having atomic and non-atomic versions of all read and write operations. These atomic reads and writes would have slower performance than their non-atomic counterparts, and as such, should be used sparingly.
2.2 Thread Manager

The rUNSWift 2009 code base contains several concurrent threads:

- Actuation
- Speech
- Perception
- Communication
- Vision
- Sensory

Though as discussed in Section 2.2.1, there was additionally a separate thread for behaviour and inter-robot communication as well.

The top-level thread, where the rUNSWift system is initialised, creates and schedules the other threads, registering them with a Thread Watcher class. The Thread Watcher also imposes limits on how frequently each thread can be run in terms of number of complete cycles per second, this allowed fine-grained distribution of the available CPU time.

The Thread Watcher also catches any exceptions or signals that would otherwise cause the rUNSWift broker to crash with a fatal runtime error, and instead restarts the offending thread immediately. This action is logged for later analysis. This provided greater robustness when the robots were in play on the field, avoiding the potential time-wasting of having to completely restart NaoQi on a robot if a fatal error occurs, whilst maintaining a complete record of the incident so that bugs do not go unnoticed in later iteration of the code.

2.2.1 Perception Thread

In the earlier versions of the rUNSWift 2009 code, the Localisation and Behaviour modules were found in their own independently scheduled thread.

However, as updates from the Localisation modules were thought to be the primary cause of change of state in Behaviour, it was decided that these two modules should be joined into a single ‘Perception’ thread.

In reality the Behaviour module’s decision state can be affected by any of Localisation, Sensory, Vision or Communication; so this approach had the drawback that changes in an individual sensor’s input value would take a full cycle of the Perception thread before Behaviour was impacted. This slowed down responsiveness to events such as: the robot falling or being knocked over, the Game Controller (referee) signalling a penalty, or the robots manual interface (chest and foot push buttons) being used.

A potential way to overcome this defect would be to run Behaviour in a separate thread, and create Blackboard hooks (see Section 2.1), that set the Behaviour thread’s state as ‘runnable’ whenever a relevant input has been updated, thus allowing Behaviour to function independent to Localisation or other parts of the Perception thread, and update its decision-tree state more quickly.
2.3 Communication

There are six different communication modules used throughout the rUNSWift 2009 code base:

- Blackboard Server
- Remote Control Protocol
- Localisation Streamer
- Inter-Robot Communications
- Game Controller Receiver
- OffVision

Each was developed to fulfill a specific need, and each could be switched on or off at compile-time. It may be advisable to move towards a more generic communications interface for future versions of the rUNSWift broker.

2.3.1 Blackboard Server

The Blackboard Server is a generic interface for reading and writing to a Nao’s Blackboard over the network at runtime. It was implemented using the TCP/IP protocol, and was primarily used to stream images and detected object information from the Nao’s camera for use in debugging the Vision module.

This module could potentially be used for all communication between the robots and the various debugging clients and front-ends to the Nao’s perception that we would like to use.

2.3.2 Remote Control Protocol

Prior to the development of the Blackboard Server, a simple text-based protocol was developed to remotely send locomotion commands to the robot. The client connects over TCP/IP to the robot and the user enters strings corresponding to a command interface specified in the source code (not otherwise documented). It has been our desire to port this to use the Blackboard Server protocol, however there was not time this year to complete these modifications.

2.3.3 Localisation Streamer

This module was developed towards the end of the project to provide information to a remote viewer that displays the perceived location of the robot at runtime, hoping to help with debugging the Localisation module. The existing Blackboard Server interface could have been used for this purpose, however a lack of intra-team communication led to this redundant module being developed.

2.3.4 Inter-Robot Communications

This module broadcasts a structure containing select items from the robot’s Blackboard to all other robots on the network using the UDP protocol. This module was tested with success in laboratory
conditions; however network congestion prevented it from working at game-time. This module could potentially be used to process observations from other robots on the field, to contribute to the World View localisation filter.

### 2.3.5 Game Controller Receiver

This module is carried over from the code used in previous years rUNSWift implementations to decode information provided at game-time by the electronic referee. The Game Controller broadcasts information such as the current score, clock, state, and penalty status of each robot in the game. The Receiver places this information on the Blackboard each time an update is received from the GameController.

### 2.3.6 OffVision

OffVision is a debugging suite built and maintained by the previous rUNSWift teams. The application contained vision, localisation and behaviour debugging utilities, as well as a colour calibration tool. This tool used the UDP protocol to send/receive data to/from the robot.

Attempts were made to port the OffVision utilities to interact with the Nao and the more recent rUNSWift code, however these attempts were unsuccessful. OffVision was still used as a colour calibration tool, accomplished by running the rUNSWift 2006 code on the robot. Other debugging was achieved by building debugging tools for each specific module, using the Blackboard Server to communicate with the robot.

In future iterations of the rUNSWift system, it would be beneficial if OffVision could be re-written to integrate all the Nao debugging utilities, utilising a unified communications protocol, such as the Blackboard Server.

### 2.4 Vision

#### 2.4.1 Background

The information extracted by the vision module provides the basis for higher level modules to operate, so naturally it is desirable to have an accurate vision system. However, computer vision is not a precise science, and is therefore one of the major challenges posed by Robocup. Even so, Robocup has progressed, and as these challenges are met, the competition evolves, moving closer and closer towards a more realistic human soccer competition. This year was the first year that required the robots to function under natural lighting (around 500 lumen), as opposed to 1400 lumen in previous years. Additionally, all localisation beacons have now been removed, and due to time constraints the rUNSWift localisation module was never ported to the Nao. This severely hampered our ability to sanity check against a world model. Finally, the move to the Nao has also complicated the calculation of the horizon due to the height of the camera and greater error in the estimated robot pose.

These challenges have tested the robustness and accuracy of the previous rUNSWift vision model with cases it was never designed to handle.
2.4.2 Overview of the 2008 Vision Module

At the 2008 competition, the rUNSWift team relied on a slightly modified version of the AIBO vision module dating back to 2005. The vision model introduced by Alex North [4] made use of scanlines to minimise the aggregate CPU usage required to process the image by analysing interesting regions of the image only. These interesting regions were isolated in two ways. The first, known as SubSampling, assumes that points of interest appear in clusters throughout the image. Scanlines drawn by the vision module are relatively sparse, however when a single interesting point is detected the adjacent area is scanned densely for other related points of interest. Secondly, the AIBOs also made use of a horizon calculated based on the robot’s pose. The horizon was used to estimate the density of interesting points throughout the image. The closer to the horizon a point in the image is, the farther it is from the camera. Regions in the image at a distance are scanned finely to detect objects that appear smaller due to perspective, whilst regions closer to the camera can be scanned less densely. These two factors allowed the vision module to generate a very effective scan grid optimised for the input image. Once enough points of interest have been found, they are then passed to the SubObject object detection algorithms.

2.4.3 Overview of the 2009 Vision Module

The 2009 Vision was also originally ported from the AIBO code, but by competition time almost everything had been replaced. The underlying model however remained similar. The major difference is instead of using points of interest, line segments are used to construct the object models. Other than that the vision module remained a single pass pipeline, which passed data from the image to the feature detector, to the object modeller, to the sanity checker, and ultimately to localisation.

2.4.4 Colour model

The native format of the images received from the camera is YUV422. This colour model has a brightness channel, Y, which is a grey scale representation of the image. The U and V channels define the ”blueness” and ”redness” of the image respectively. The YUV422 format compresses the size of the image by sharing a single U and V value across two horizontally adjacent pixels. A side effect of this is that the Y channel compensates for the constrained UV values, causing a wavy pattern in the Y channel that can interfere with edge detection.

2.4.5 Colour Classification

The colour space is divided up into several symbolic colour regions using a nearest neighbour colour classification approach. These symbolic colours included

- Ball Orange
- Goal Blue
- Goal Yellow
- Field Green
- White
Figure 3: Top: Zoomed in image of a ball demonstrating the effects of a shared YUV pixel. Note the highlighted ‘light dark light’ pattern in the Y channel. Bottom: Plot of classified ball and goal colours in YUV space.
• Robot Blue
• Robot Red

The separation of these symbolic colours are somewhat indistinct, as shown by the lack of separation of colours in Figure 3.

2.4.6 Natural Lighting

The 2005 code relied heavily on colour transitions to detect points of interest. Colour transitions, rather than individual pixel colours, were chosen due to their stability varying lighting conditions. Alex North [4] provided a simple formula for defining a ball point in terms of the derivatives of each colour channel histogram along a scanline. From experimentation, it can be seen that this formula only sets loose bounds on what is and is not a ball point, and further colour analysis of surrounding pixels is performed to determine whether or not the point is actually a ball point.

In natural lighting conditions however, this method fares far worse for a number of reasons. Firstly, one of the less obvious advantages provided by artificial lighting was a separation between a brightly illuminated field and a lesser illuminated background. This meant colour changes on the field were far more drastic than in the background, and it was easier to place bounds on what defined a ball point in terms of the image derivatives without including cases commonly occurring in the background. Secondly, in a darker image contrast is lost between colours. Parts of the red robot uniforms and the bottom half of the ball have almost identical colour. There are many transitions to and from robot red that fall well inside the formula for detecting ball points, and as a result the detection of false balls inside of red robot uniform parts became a serious issue. Similar problems occurred due to the lack of contrast between robot blue and goal blue, which caused blue goals to be falsely detected. Under the right conditions, even goal yellow and ball orange are similar enough that goals will be detected inside of a ball. In order to dampen the impact of reduced lighting, the camera exposure was raised. This led to a third issue; the images captured by the robot under competition lighting were extremely blurry. This distorted all notion of colour transition, and the blurry images would often produce false positives. The lighting in the UNSW lab is actually even darker than 500 lumen, which explains why extreme blurriness was not observed under lab conditions. In future, set the camera exposure to its lowest possible setting.

2.4.7 Line Segment Detection

Instead of trying to process individual points along a scan line, many of which were falsely detected anyway, we moved to trying to detect line segments of an object instead. The scanlines were broken up using hand written MMX assembly edge detection routines. The edge detection algorithm worked by detecting zero crossings of the second derivative across each colour channel. These gave an accurate approximation of where the points of inflection across each channel lay, and these were taken to be the edge points. The scanline segments delimited by these points are then processed, and any two adjacent segments with the same symbolic colour are joined.

Line segments provide useful information that points of interest do not, such as width, average colour, and a relationship between two endpoints that are points of interest themselves. Such information is very useful for both object modeling and sanity checking.

The major disadvantage is that, like the previous method of detecting points of interest, crisp images are a necessity. At competition, the blurry images forced all edge detection thresholds to
be set very low, and line segments were predominately built by combining many smaller segments of similar colour. Shadows can also create an interesting ‘gotcha’ which is worth being aware of. Before completion, the vision module was used under AIBO lighting levels to test its robustness. One goal post was poorly illuminated from one side and was left in shadow, and the edge detection algorithm split the goal in two, and only one side was classified. This had the effect of making the goal twice as thin and appearing 4 times further away. Shadow detection may also be usable for obstacle detection, however this requires further experimentation.

2.4.8 Zero Detection

On board the Nao has a x86 processor with a 3D Now! and MMX module attached. MMX instructions are SIMD (single instruction multiple data), which allow for multiple identical operations to be performed in parallel. As the zero detection algorithm performed on each colour channel is identical, the calculation time can be greatly reduced by manipulating all three channels at once. Furthermore, as only 3 bytes are required to represent a pixel, and the MMX registers are 8 bytes wide, two scanlines can be processed in parallel. When combined some other optimisations, the result is an 8 times speedup over equivalent C code.
Figure 5: Across: First derivative, Second Derivative, Zero Detection. Downward: Y, U, V channels. Corresponding values calculated across an image. Note the image is half width because shared pixels have been averaged together into a single pixel. This eliminated the ‘wave’ problem, but was not used at competition
The exact algorithm used to detect zeros first requires the first and second derivatives across the scanline to be calculated. The second derivative is then integrated over the last three pixels to smooth out irregularities from the noisy image data. Next, the derivative of the integral are calculated, which is what will eventually be searched for zeros. First however, a simple filter is applied to remove any positive - negative - positive or negative - positive - negative patterns in the integral derivative. Finally, zero detection is done to detect any point where the integral derivative that passes through zero. All these stages are implemented in such a way that only a single pass is required, and other than looping along the scanline no other branching is required. This also provides a significant reduction to computation time.

2.4.9 Horizon

Unfortunately, this year we were unable to accurately calculate the horizon. This was a result of two problems introduced by the Nao. The first is that the camera is now a significant height (450 millimeters) above the ground. As such, the horizon can no longer simply be considered the horizontal line passing through the point at infinity, directly in front of camera. Instead, the Nao looks downward towards the field, and calculating a useful horizon requires a localisation that can provide a rough estimate of where the robot is in relation to the edge of the field.

Horizon calculation code was developed that calculated the horizon in the image given a distance between the robot and the edge of the field, but this was not used due to the lack of a reliable localisation module.

Early on, the horizon was fixed at 3 metres in front of the robot, and later fixed at 7 metres, however neither of these values were able to consistently improve the performance of the vision module.

Joint angle error causes a further complication when calculating the horizon. Unlike the AIBO which has four support legs providing a relatively stable base for the camera, the Nao has a tendency to rock back and forth as it walks. As a result, there is considerable error in the calculation of the angle between the camera and the ground plane, a vital parameter used when calculating the horizon.

Instead of using a horizon to cull falsely detected points of interest, the relationships between adjacent line segments were used to validate or invalidate individual line segments. For example, an orange ball segment must lie between two green line segments. This worked well, however the logic quickly became unwieldy and such a solution is simply unscalable. A better approach would be to bound the field, and cull all points outside the field.

The lack of a horizon also meant that we were unable to distribute scanlines in an intelligent manner. With the optimised edge detection routines however, we were able to process one horizontal scanline every 5 pixels, and this proved to be sufficient for detecting the ball at full length of the field. Most of the processing power was wasted though, and should be reallocated to other modules such as localisation.

2.4.10 Goal Detection

The goals used in the competition have changed from a solid coloured rectangle to a more realistic coloured miniature soccer goal. As such, new goal detection routines were introduced. The basic methodology used to detect goals was to build stacks of goal coloured line segments with similar widths. If a line segment fit on top of an existing stack, it was added to the stack and the stack’s
properties, such as center, median width, and line count, were adjusted to reflect the added segment. If a single segment fit onto multiple stacks, it is added to each of them. However if the segment did not fit any stacks, a new stack was formed with the rejected segment as its base. After all line segments have been processed, overlapping stacks with similar properties are merged together, and finally the best two stacks are taken to be goal posts.

Later on, linear regression was used to describe a line passing through the midpoints of all segments added to a stack. In order to be accepted into a stack, a new line segment had to also fit line. This extra constraint reduced false positives, but at the same time caused many more stacks to be created. This was handled later by the merging like stacks together if they either overlapped horizontally or had similar regressed lines.

2.4.11 Ball Detection

The ball detection algorithm used was exactly the same as the one used in 2005. Points of interest were taken from either side of a ball coloured line segment, and fed into the algorithm. (See [4] for exact implementation details). The one downfall of this algorithm is that it copes poorly when there is more than one ball like object present, such as a robot hand and a ball. In future, reliance on colour must be avoided at all costs, and any object detection algorithm should be able to deal with multiple hypotheses.

2.4.12 Sanity Checking

The original plan was to allow vision to produce multiple hypotheses of varying degrees of confidence and then allow localisation to apply each observation to its world model. However, the vision capable of producing multiple hypotheses was scrapped due to time constraints. The final localisation used was also somewhat primitive, and suffered badly if passed any false positives. Therefore, vision had to be extremely robust against any false positives, and this was achieved through stringent sanity checking. There are several approaches that were taken towards sanity checking and these are listed as follows:

**Ball Size** Measurements were taken to determine the maximum ball radius that could be expected to be seen in a frame. If a ball was detected with a larger radius, it was discarded.

**Goal Width** In order for a pair of goals to be accepted, their widths had to be within a certain tolerance. If one thin goal was detected alongside a wide goal, at least one is likely to be a false positive.

**Goal Line Segment Count** For an individual goal post to be detected, its stack had to contain a minimum number of line segments. This worked well because at a distance it was likely that both goals would be seen, but each goal would have a low line count. In this case the goals could be checked by width against each other. At a lesser distance, goal posts would contain many more line segments, but seeing both goals at once is unlikely. A large line count is therefore required to confirm the existence of an individual goal.

**Colour Composition** After an object had been detected, the pixels in the area inside and around it are sampled for their colour. In order for an object to be accepted, it had to both contain a great enough proportion of related colours, and not contain a proportion of other colours. For example, a ball had to contain at least 70% orange pixels, and less than 20% of either robot red or goal yellow. (See Appendix B)
Goals in Balls  To stop yellow goals appearing in the sides of a blurry ball, any goals which were small than the ball radius and within a distance of two radii to the ball are discarded.

2.4.13 Camera Settings

The most important lesson from this year in regards to vision is that the exposure must be set to its absolute minimum. This will also require fixing the lighting in the lab to bring it up to competition levels. With a dark image, it was also advantageous to set the contrast as high as possible, as this had the effect of separating like colours such as orange and robot red. The disadvantage is that like colours varying across the image due to camera imperfections are also separated. This can be seen by comparing the field green in the corners of the camera against the centre of the image. The corners appear to be almost black, whilst the centre of the image retains a more natural green colour. The camera also has a reddish tinge, but the image blue balance can be raised to counter this.

2.4.14 Camera Interface

Unlike most teams, the rUNSWift used v4l (video for Linux) to directly communicate with the camera, rather than via proxy through the Naoqi video module. v4l provides three methods of receiving input from the camera.

File Descriptor  As with everything in *NIX, the camera device can be read by using open to obtain a file descriptor and then using read. This is the simplest method to implement, however it also requires a redundant copy of the image to be made in kernel space before being copied into user space.

mmap  mmap allows for a user space application to map protected kernel space memory buffers into its own address space, eliminating the need for a redundant kernel space copy. This is the method used in this year’s code.

User Pointers  In theory, this method is safer than mmap because the user space application allocates the buffers itself, and only passes a pointer to the kernel which writes to the buffers. We were unable to get this method to work with the Nao’s camera.

This year the Nao gained a second camera allowing it to look downwards at the ground. Many of the 2009 teams played very well using only the bottom camera, so it may be worth reconsidering whether or not the top camera is needed. There are a two drawbacks to using both cameras. First, only one camera may be used at a time, as they share the same video bus. Secondly, the switching process takes about 200 ms, so the cost of switching cameras is approximately 6 frames.

To switch cameras, an on camera register must be written over an i2c bus. (See Appendices C & D for full implementation details.)

/**
 * These addresses, commands and flags are given to us by aldeberan. They enable talking to and controlling the camera controller.
 * I2CDEVICE  -- the i2c bus that the camera controller resides on
 * I2C_SLAVE  -- the flag to indicate slave mode on the i2c bus. allows writing to the camera controller
 */
2.4.15 Debugging

To debug vision, a series of macros are defined that draw various primitives onto a separate debug image. The debug image can then be copied onto a streaming client via the blackboard server. The macros can all be disabled using a compile time flag such that there is no overhead caused by the debugging code. When enabled however, the debug image caused a five times slowdown of vision, causing difficulties when testing other modules.

The blackboard server can also send the standard image, along with the coordinates of the ball and goals. Streaming via this method had no noticeable effect on the performance of vision and as can be used whilst testing other modules.

2.4.16 Shoulder Cropping

If the Nao’s head is turned too far left or right, its bottom camera will look directly at its shoulder uniform pieces. These are coloured either robot red or robot blue, and will be detected as either balls or goals. To fix this problem, the shoulders are cropped out of the image before processing.

2.4.17 Gotchas

There are two issues with the camera which we can not explain. Sometimes the camera will start up and the entire image will have either a strong green or purple tinge. Restarting the camera will fix this. Also, sometimes two bytes will be missing when reading the image from the camera. This has the effect of inverting the U and V channels of the image. Luckily, there is also a line along the bottom of the camera image which always has a bluish tinge due to some property of the camera. If two bytes are missing, then this line will appear orangey. This can be detected and used to correct the image.

2.4.18 Moving Forward

After experimenting over the course of the year, I am convinced that scanlines are not the solution to the robocup vision problem, or vision in general. Compared to previous methods which required processing the entire image, scanline processing was relatively efficient. However, to make the process efficient a reliable horizon is needed, and it will be difficult to calculate such a horizon based purely from robot pose now that the competition has moved to humanoid robots.

Instead of searching for individual objects, the robots must begin to process the world model as a whole. From my balcony there are several white struts, with green trees behind it. If parts of
this image were processed using the current vision model, the white struts between the green leaves
would be detected as field line segments. As humans, we analyse the balcony as a whole, and see
white struts, not field lines. The processing of the relationships between adjacent line segments is
a primitive precursor of world model processing. Orange line segments are no longer ball segments
simply because they are orange, but they must fall between other line segments which reaffirm that
the ball segment segment fits within the ‘world’ model.

Processing the world as a whole will also require a highly accurate world model. From conversations
with team members of Nao Devils and BHuman, I am led to believe that the Nao Devils’ world
model of all robots and ball is accurate to 5cm in ideal conditions, and BHuman’s is accurate to
within 10cm. These are ambitious targets to compete with, however if rUNSWift is going to remain
competitive such a localisation is a must. Localisation data should be based primarily on field lines,
the Any Ball challenge (See Section 3.1) proved the field can be divided up easily into green and
non green sections based purely on symbolic colour classification. From here, non green regions
can be processed to detect field lines and other objects, instead of searching the entire image as a
whole. It is my guess that about 50% of the image processing done in the 2009 vision module was
wasted drawing scanlines across green field.

Once the robot has a solid localisation, it should also be possible to calculate the shape of the
goals by projecting them onto the image plane where they would be expected to appear. Look
for shape, not colour. Colour calibration is finicky, and with such little contrast between colours
serious problems arise. It is also very likely that in the future the ball and goals will lose their
artificial colour, and it will be a huge advantage if rUNSWift can already detect goals based on
shape rather than colour.

A solid world model should also allow the vision module to be predictive in the regions of the image
it scans. The first priority should always be matching the visible field lines to the world model to
provide a reference point for the current image frame. Objects can then be searched for based on
their previous locations in the world model.

Finally, the camera should be switched to use a 640x480 resolution. There was no team at the
2009 competition that I am aware of using this resolution as there is simply not enough processing
power to process the entire image. However, by processing only intelligently selected regions of the
image, it should be possible to extract all relevant data in real time. To highlight the necessity for
a higher resolution, from 6 metres across the field, a goal post is 6 pixels wide, and a ball is either
6 or 4 pixels wide depending on where it falls on the shared UV pixel component boundaries.

2.5 Behaviour

2.5.1 Introduction

This chapter aims to provide a detailed analysis of the Behaviour module that was used at the 2009
Robocup World Championships. However it will also provide as brief overview of the evolution of
the behaviour module to date, beginning with the 2008 World Championship code base.

2.5.2 Behaviour Module 2008

The Behaviour that was utilised in the 2008 World Championships was the third iteration of the
Behaviour module that was developed for the 2008 competition, the first two having been developed
in scripting languages (specifically: Python and Ruby). Many issues arose from the implementation
of this module in a scripting language and eventually, since the rest of the code base was already in C++, it was decided that the behaviour module should be implemented in C++.

The Behaviour module was run together with the locomotion thread to assure that they would both run synchronously.

The 2008 World Championship module was divided into 4 layers Players, Roles, Brains and Skills. Behaviour would initially create a Player, of this Player one of three possible roles was attributed to it (Striker, Defender, or Goalie).

Every Role then contains a Brain, this Brain plans the movements and actions of the robot in accordance with information it is receiving and available Skills. Every Player shares the same Skills set, however the Brain dictates how these Skills are used.

Skills are then further classified into two separate categories Head Skills or Body Skills. Head Skills deal with the tracking and localising of the robot, whilst Body skills deal with the movement, kicking and stability of the robot.

This Behaviour architecture represents a very primitive architecture, whilst it worked well for its purpose it had a number of issues and was fundamentally flawed. Due to the nature of the competition, the structure of the code also became unstructured. This combined with the lack of scalability and extensibility led to the module being re-written.

### 2.5.3 Finite-State Machine Behaviour

Learning from the problems of our Behaviour architecture from last year’s competition, we started the design of the new Behaviour module with high level analysis of the requirements for the robot within a game of soccer. This analysis revealed that the robot is constantly in two states of actions, one state attributed towards the head and the other towards the body. Further analysis also showed us that the states of actions in which the robot could be in is finite and can be predetermined, hence the idea of utilising a Finite State Machine (FSM) was born. The following are the FSMs developed for the robot:

There is only one instance of the Behaviour module running on each robot, this module generates the player class which generates the head and body FSMs attributed to the robot. States are linked to each other through the player class allowing for the transitions between states, initialisation of state is also achieved through the player class. Each state contains multiple skills. These skills do not have to be unique to the one state. Using this FSM architecture, behaviours can be easily expanded and altered by adding new skills or adding state transitions.

The Behaviour FSM consisted of an initial state in which all player classes began in, a default transition was also embedded into the player, this serves as safe guard in case no criteria were ever met for the transition between one state and another. The robot’s FSMs function as two individual FSMs, however communication between the two FSMs is facilitated. Inside the blackboard there are two struts which control the movement of the body and head of the robot respectively. This is then read by the locomotion class which generates real world movements.

The formation of the Behaviour module was designed on the basis of generic and plug and playability, with the implementation of our modular blackboard base software architecture, the Behaviour module was designed in a similar way. Player classes were easily changed, altered and swapped for different roles, the transitions between states and skills were easily amended or improved upon allowing for the rapid development of player behaviours. The modular design of Behaviour into separate classes or players, states and skills also meant the integrity of the module was kept in
Figure 6: Overview of the behaviour structure used in 2008 [1]

Figure 7: Overview of the body FSM [3]
Figure 8: Overview of the head FSM [3]

Figure 9: Overview of whole FSM behaviour [3]
place making it easier to amend and extend the Behaviour module.

2.5.4 Decision Tree Behaviour

This FSM architecture was utilised at the China Open of 2008 and with early friendly matches against Newcastle University. Whilst the architecture provided an easily extendible and scalable solution, we discovered fatal flaws which eventually meant that our Behaviour module was re-designed. The two separate FSMs caused many issues with our architecture, whilst there was communication from the head FSM to the body FSM, there was no communication vice versa. This made it very difficult to switch to required states for the head FSM, and also resulted in wrong state transitions for the head FSM. The code base in which the Behaviour module was developed became too cluttered and complex; there was too much logic attributed to the player level. This made it very difficult to map the exact transitions between states in the FSM, this meant that there were a lot of issues with the robot’s state transitions resulting in incorrect states being run.

From this code base debugging became a complex task, there was simply too much logic and code attributed to individual layers (especially the player level). As the number of skills increased it became clear that this current architecture and design of the Behaviour module needed to be re-written.

Due to time constraints we decided to adopt a new architecture that would utilise all existing skills with the FSM Behaviour code base, this new architecture would also need to mimic the FSM’s code base in terms of extensibility, scalability and utility, a quick plug and play ideology.

From the number of issues encountered utilising the FSM Behaviour Module (especially state transitions), it was decided we would scrap the FSM and adopt an architecture based on a decision tree instead.

The decision tree would be split into multiple layers, with each layer sharing the amount of complexity for the skills the robot is trying to perform, this made sure that the code base remained easy to read, step through and debug. Transitions between skills should be easier to map.

All players had the same skill base which was adopted from the skill set in the FSM, it was then dependent on the role of the player the layout of the skills within it’s decision tree. This allowed for the rapid prototyping of new players as required, this was especially important at the Robocup competition as it became clearer what skill level our opponents had.

The decision tree Behaviour module is split into the following 4 layers:

**Behaviour** This is the class that initialises all the players with the Behaviour module, it also deals with what role the robot should start on based on the game controller data it is receiving.

**Players** The player layer assigns the role to the robot and it also sets up the composite skill layer which creates the decision tree with allows the robot to function according to its role. The connections between the composite layer is assigned in this layer, learning from the previous architecture a lot of the complexity of deciding of what the robot is required to do is shared between this layer and the composite skill making it easier to maintain, expand and debug.

**Composite Skills** A composite skill combines skills from the head and body and utilise them to achieve the tasks set within its domain. It creates a decision tree within it own class and from there the robot is able to choose what skills it requires to achieve it purpose. Composite skills can be reused by other players hence reducing the amount of code that is required to
be rewritten, this layer also provides for a means of communication between the body and head improving on our previous architecture.

**Head/Body Skills** This is the lowest level of the Behaviour module the skills are split between head and body skills. These skills are reused from the old behaviour architecture; they are the only layer of the behaviour module that actually writes to the blackboard. Skills may be used by multiple composite skills allowing for less code to be written, this also improves on expandability and scalability.

![Diagram of new decision tree behaviour](image)

*Figure 10: Overview of new decision tree behaviour*

Utilising the new decision tree Behaviour two main players were created namely a Goalie and a Striker. Others players were created too and these will be detailed further in this report. Whilst the Goalie and Striker had the shared skill set, their composite skills were formed differently leading to each robot having a different role. The following details the structure of both players from a high level composite skill perspective.

### 2.5.5 Goalie

After analysing the requirements of a goalie keeping in the Robocup domain, it became apparent that a goal keeper has three tasks in which it must accomplish. These tasks are to Guard the goal, to relocate itself in front of the goal, and to stand in front of goal, from these three simple tasks we developed the composite skills of the Goalie.
Goalie Guard  This composite skill makes sure that the Goalie is in the optimal position to protect or chase the ball out of the goal mouth when required. The Goalie will always try to line itself behind the ball as long as the ball is in front of the goal mouth, if the ball is within a certain distance it will decided to chase the ball out and kick it to clear any possible danger. Whilst this composite skill sounds quite simplistic and easy to implement we had a lot of issues trying to implement it. Two key issues we encountered were the lack of a reliable ball distance measure and the lack of an accurate localisation, without these two functionalities it became very hard to build a strong and effective Goalie Guard.

Goalie Centre  After the Goalie guards the ball and eliminates the threat of a goal it will be out of it optimal position, in the middle of the goal, this composite skill was designed to allow the goalie to relocalse itself and move to the best position to protect the goal. This composite skill relies heavily on localisation. Accurate and reliable data must be obtained from localisation so that the Goalie will know its heading (which direction it is facing) and location on the field. From this data the optimal path to return to the centre of the goal can then be calculated, our main approach to this composite skill was to utilise our own goal posts as a key landmark to localise off. Since our own goal post will be within close proximity of our goalie the data obtain should be much more reliable than localisation off other landmarks. This approach combined with a field line detection system should yield a fairly accurate result. Due to time constraints and issues within the localisation module neither of these requirements could be met, making localisation of the goalie very difficult. We decided to utilise odometer data instead, and whilst this yielded some results it was too inaccurate to be readily used.
Goalie Stand The most simple of the composite skills attributed to the goalie once the goalie was centred on goal and if the ball was not within sight. The goalie should stand in front of the goal whilst using its head to constantly stand for the ball. As we progressed into the penalty rounds of the Robocup competition we added an extra goalie crouch move. The move involved the goalie having a very wide low stance that would cover a large area of the goal; the ball if blocked by the goalie would be trapped between the goalie’s legs. From simply getting up from this crouch position the goalie pushed the ball out of the goal mouth, this proved to be a very simplistic but effective strategy that was utilised by many other teams too.

2.5.6 Striker

The design of the striker player was formed with the important concept of simplicity in mind, and the lessons learned from the FSM Behaviour model. However analysing the tasks attributed towards a striker player was no simple task, a striker player has a many possibilities in each and every scenario. We reduced our striker was model down to three composite skills:

![Decision tree of the goalie player](image)

Search Field A critical skill of the striker player is the ability to locate the ball regardless of its location on the field. It must also be able to localise itself so that it will know its location, heading and optimal path to reach its destination. In order to do this an efficient scanning and searching mechanism was developed for the searching the field.

Developing an optimal head scanning routine for the robot was not a simple task. The Nao has two cameras available to it, one with a long and another with a short distance range. We
encompassed both cameras into our scanning routine that allowed the robot to scan the field more throughly, whilst also enabling it to better track the ball once found. Another problem faced was the latency between the time commands that are sent to the head and the rate at which such commands can be processed. If commands are sent too rapidly, the head cannot complete each command resulting in a very jerky head scan. This caused the images taken from the camera to be blurred and impacted drastically on the robot’s ability to see the ball and other landmarks. To overcome this, a PID controller is used to smoothly couple head commands with the actual head movement routines.

Our optimal head scanning technique involved using the short distance camera first, to see if the ball was within close proximity. The head would scan from left to right and then right to left covering field of vision that spanned 120 degrees. If the ball was not seen then the head pitch would be raised to 25 degree and vision would be signaled to utilise the top camera. The switching of cameras had the ability to generate a blind spot, however this was overcome by utilising our optimal head pitch angle that would make sure the two images seen by the two different camera interlaced. The head would then scan from left to right in two sweeps covering a field of vision of 120 degrees.

With an optimal head scanning routine an efficient field roaming routine was required in order to scan the field in the most efficient manner. This routine had to be able to cover the largest area of field within the shortest possible amount of time whilst still being able detect the ball, as speeding up the walk compromised sharpness of an image taken from the camera. The best results from the head scanning routine came when the robot’s body was stationary; hence from this idea we developed a efficient body scanning routine to compliment the head scanning routine. The robot would rotate on the spot stopping every 90 degrees to allow the head to scan the part of the field, while our head scanning routine had a field of vision of 120 degrees which meant that we could turn on the spot every 120 degrees instead of 90 degrees, it was discovered that scanning towards the extreme left or right of the robot (yaw of 60 or -60 degrees) the vision was impeded by the robot’s shoulder. Hence we built in a bit of redundancy to assure the whole field is scanned.

We combined these optimal scanning routines and used them to scan the whole field efficiently and quickly. Despite vast improvements to our vision module our robot was still not able to see the ball if it were close to a field length distance away, this meant that there were cases where despite scanning the whole field the robot will still not see the ball, we created a skill called wander field specifically for this worst case scenario.

Wander field attempts to return to the centre of the field and scan 360 degrees to pick up the ball, as our vision module was able to pick up the ball reliably from a distance of half a field length. However due to the lack of a reliable localisation module and self world model for the robot, this skill never worked very well.

**Approach Ball** This composite skill is responsible for getting the robot within a close range of the ball and facing the right direction towards goal. When the ball was seen the robot would immediately jump to this composite skill, this meant that this composite skill dealt with both long range and close range methods of approaching the ball. The head would constantly track the ball, if the ball was far away the top camera would be used and if close it would switch to the bottom camera and vice versa.

This composite skill had different functionalities depending on the distance of the ball. If the ball was far away its main goal was to approach the ball as quickly as possible, this was achieved through setting the gait engine to it’s maximum speed and only using turn skills to align the robot to the ball. The location and heading of the robot is ignored.
As soon as we are within proximity of the ball the robot begins to localise in preparation to score a goal. The top camera is used to pans for a goal and obtains a rough heading and location of the robot. The robot then attempts to align itself, the ball, and the opponent’s goal. The worst case scenario occurs when the robot is facing its own goal, in which case it must promptly rotate around the ball, getting behind it and facing the opponent goal.

**Attack Ball** Once aligned, the robot can enter an attacking composite skill. The purpose of this composite skill is to keep the robot aligned during the final approach to the ball and then either kick or push the ball into the goal. To achieve this a we utilised a player that would switch from kicking and pushing based on the distance from the opponent goal. During our friendly matches against Newcastle University, we discovered for the Robocup world Championships just kicking the ball in the right direction yielded better results. Kicking the ball in the right direction however requires very accurate ball distance and heading, as a slight deviation will kick the ball off at an angle. The lack of accurate measurements caused many issues with our all distance calculation function, but through experimentation it was discovered that taking raw data straight from the vision module, rather than through localisation, yielded excellent and accurate results when within close proximity to the ball. Hence once the robot was close enough to kick the ball, it would switch from reliance on localisation data to raw vision data. In order for the robot to effectively kick a ball one of its two feet must be aligned (within +12 degrees relative to the robot’s body) with the ball in kicking distance. Speed was another critical factor, the ability to align quickly and kick was very important. However, if we increased the speed of the robot’s gait as it was trying to align to the ball, the robot would on many occasions overshoot its target and walk into the ball, tapping it forwards. The robot would then have to realign itself to the ball once more, reducing the effectiveness of the skill. Through experimentation we discovered that the most effective speed to align behind the ball was 20 percent of its maximum speed (16cm/s).

**2.5.7 Conclusion**

This new Behaviour architecture was a huge improvement on the old Behaviour architecture. Complexity was well distributed through the different layers of the decision tree, which led to more structured code that was easier to debug. Whilst this behaviour was able to function and meet its requirements, there were several flaws within its architecture that can be further improved.

One of the key issues faced was the lack of a scheduler. Skills would run, but it was impossible to explicitly set one skill to follow the next. A prime example of this is when the robot kicks for goal. We want the robot to look up to confirm that it can see the opponent’s goal and then kick the ball. However once we look up for the goal the decision tree will think that we have lost the ball and enter a search for ball routine. In this situation we should be able to force the decision tree to commit to the kick skill. To solve this, a local blackboard within behaviour was created which allowed state to be shared amongst the different tiers of the decision tree, however this wasn’t a very elegant solution and reintroduced the problems inherent of the FSM.

Communication between the head and body skills was vastly improved as the composite layer facilitated the sharing of data between the two skills. Despite this the behaviour module also suffered greatly due to shortfalls of other modules, particularly localisation.

A lack of an accurate ball distance function led to a lot of issues within the behaviour module. Skills were based on thresholds that were derived through trial and error, rather than accurate distance readings. The kicking skill was also very difficult to develop utilising the provided ball distance. It
was not until we switched to raw vision data that this issue was solved. This year’s competition was played with a basic goal heading value and no other reliable data from localisation; this meant that the robot had no world model or reliable sense of self-position. The sharing of world models between robots and the development of team strategies with complex skills was not possible due to these shortcomings. However, considering our skill set the behaviour performed very well in the competition despite its flaws.

3 Competition In Graz

The 2009 Robocup Symposium took place in Graz, Austria. We were among 25 teams from around the world competing in the Standard Platform League, the league which uses the Nao robot from Aldebaran with no hardware modifications.

We were placed into Pool D which consists of three teams in a round robin:

- rUNSWift (UNSW, Australia)
- ZaDeAt (Knowledge-Based Systems Group, RWTH Aachen University, Germany; Institute for Software Technology, Graz University of Technology, Austria, Agents and Robotics Research Lab, University of Cape Town, South Africa)
- Mousqetaires (Laboratory of System Engineering, Versailles University, France)

The following is adapted from an email sent by Mr Brad Hall to the CSE community when we had finished competing:

Our first game was against ZaDeAt, which was problematic due to a colour calibration problem in the first half. In the second half we were also still having calibration problems and kicked the ball out a lot. However, we kept possession most of the game and so although we were unable to score, we kept it to a 0-0 draw.

In the second game against Les 3 Mousquetaire we also had possession most of the game. At one kick-off we scored immediately, however, this is not allowed (the ball must leave the circle before a goal can be scored by the attackers). We had a few more close goals before we fumbled the ball and left it near our goal without a defender, and Les 3 Mousquetaire were able to score. However, with just 5 seconds remaining on the clock, we were able to get a goal in and equalized the game. There are no penalty shoot-outs in the first pool games, so it was a 1-1 draw.

Later, Les 3 Mousquetaire won a game against ZaDeAt 1-0, so we came second in our pool.

Second and third teams then proceed to an Intermediate Round, where we came up against TJ Ark from the School of Electronics and Information, Tongji University, China. This game ended in a 0-0 draw, but in this round, draws lead to a penalty shoot-out. There were 5 attempts at goal for each team, and unfortunately, the Chinese scored after the third, and we were not able to equalise, so we were knocked out of the competition.

Following our being knocked out of the soccer competition, we spent our remaining time in Austria developing code for some of the ‘challenge’ competitions. Each year at Robocup, a number of
challenges are put forward, with the aim of rapidly advancing the level of innovation found in Robocup software.

rUNSWift 2009 entered in the Any Ball Challenge, the Passing Challenge and the Localisation Challenge. We will describe our entry into the former two in detail:

3.1 Any Ball Challenge

So far in the history of Robocup, ball detection has relied heavily on the precise specifications of the ball, in particular colour information, but also the size, weight and texture. The Any Ball Challenge aimed to move towards the development of ball detection software that did not rely on these strict specifications.

The challenge required participants to locate a number of previously unseen balls on the field, and kick as many of them as possible into the goals. Points were awarded for making contact with the ball, and further points were added if a goal was scored.

To participate in this challenge, the rUNSWift 2009 team developed custom image recognition routines and behaviours.

3.1.1 Vision

From a vision point of view, the any ball challenge was interesting because it proved that useful information could be extracted from an image by considering only two symbolic colours, green and non-green. The process used relied on the opencv circular hough transform for detecting circles. The ‘any ball’ image is first converted to black and white, with all green pixels being black, and all other pixels being white. A large Gaussian blur is then applied to smooth over any noise caused by the rough edges. This is highly CPU intensive, and if this step can be removed, a hough transform should be able to be performed in real time. Once blurred, cvHoughCircles was used to find potential balls in the image. The sanity checks were then run on each candidate ball to weed out false positives, caused by other curves on the field such as the centre circle. (See Appendix E)

3.1.2 Behaviour

The downfall of our performance in the Any Ball Challenge was our inability to develop suitable behaviour code in time for the contest. The Any Ball Challenge exposed an inherent weakness in the design of many parts of our behavioural system.

Many of the simple operations behaviour would perform, such as walking to a particular point, performing a kick, or looking around, were conducted as follows:

1. Set \textit{counter} to 0, and issue \textit{command} to locomotion

2. Each run of behaviour, increment \textit{counter} until it reaches \textit{threshold}

3. Once \textit{threshold} is reached, assume \textit{command} is complete

Due to the use of these thresholds, it was necessary to tweak the behavioural constants every time a performance enhancement was made elsewhere.
Figure 13: The any ball algorithm applied in two stages, blurring and then circular hough transform
This became particularly prevalent, when the more advanced, but slower vision module was used for the Any Ball Challenge, resulting in all behaviours not functioning correctly due to having thresholds that are too large.

Our attempt at solving this problem was to perform some simple arithmetic: Since the Any Ball Vision was 8 times slower than regular vision, all constants in Behaviour were divided by 8. This did not seem to work correctly, since there is a non-linear relationship between each of these constants, and as a result we were unable to perform in the Any Ball Challenge.

This problem could easily be solved if the locomotion module provided information on the Blackboard about when these sorts of events have finished, then Behaviour can subscribe to the locomotion state using the Blackboard Hooks (Section 2.1), and not need to rely on fiddly timing constants.

3.2 Passing Challenge

3.2.1 Background

This year’s Robocup challenge signified a big skill jump from last year’s league, many teams who completed were capable of scoring multiple goals and having quite complex strategies. This year’s competition also saw reinstatement of Robocup challenges within the SPL league, the purpose of these challenges is to push the envelope of the league, robotics and AI. This chapter will focus on the passing challenge of the 2009 Robocup World Championships SPL league.

3.2.2 Requirements

This year’s passing challenge was adapted from last year’s humanoid league; it involves two robots on either side of the field having to pass the ball between them. The ball was not allowed to leave the field and a buffer zone was set in the middle of the field where the robots could not enter. The purpose of this challenge was to develop a team passing strategy, and investigate the idea of actual team play. As within the competition passing was actually quite rare and a very difficult task to achieve, this challenge presented a small subset of this problem. The robots had 2 minutes to perform 3 successive passes between them.

3.2.3 Limitations!

Our main focus throughout our lead up to the competition was the actually soccer competition itself, so we had not given the challenges much thought at all. However unfortunately we were eliminated quite early in the soccer competition hence we chose then to focus our energy on the challenges.

The design of the new Behaviour module made it very easy to create a new player class based on pre-existing skills, lining up and kicking the ball skills had already been implemented. The current existing kick on the robot was too powerful though it would always result in the ball being kicked out of the field, hence we had to tune down the power of the kick whilst still keeping the robot stable.

Localisation was another key factor that was required for this challenge; with the localisation that was available at the time we had no reliable world model or self positioning on the field hence it
became very difficult to find a solution to the passing challenge, however lucky we had a somewhat reliable opponent goal heading.

3.2.4 Our Approach

The ideal solution to this problem would be to have an accurate localisation, team communication and field line detection. Utilising the localisation data one robot could then communicate to the other robot as to which role it is (passer or receiver) and which direction/heading the ball would be going towards. Using field lines detection would guarantee the ball would not leave the field and the robots would not stray into the buffer zone.

However with our current codebase this ideal solution was not possible as we lacked an accurate localisation and field line detection. We had a robot communication module but without an accurate localisation there was not much sense in communicating between the two robots. Analysing our limitation we came upon a nave solution to solve the passing challenge, factoring in the time in which we had to develop a solution was also critical, namely 1 day.

Both robots would be assigned the pass or receiver role, rather than having this communicated between the two robots we decided that this should be based on the return value of the ball distance function when the ball was in sight. Taking into account the field size, goal box and the buffer zone we set this ball distance threshold to 1.5m. If the robot was greater than 1.5m from the ball it would assume the receiver role, if the robot was less than 1.5m it would be passer.

The passer’s role was to try to pass the ball, utilising the our tuned down kick, towards the centre of its opponent goal where the receiver will be standing and tracking the ball. Once the ball is outside of the 1.5m ball distance threshold of the passer it will switch to the receiver role and vice
versa for the receiving robot. The passer utilised skills that we developed for the soccer competition namely approaching the ball, aligning itself to kick in the right goal, and aligning itself to kick. This was a huge advantage that our new behaviour architecture allowed, we were able to rapidly develop and test this role within one day.

The receiver’s role is much more simplistic, it simply needs to track the ball and be ready to receive and switch roles to a passer as the ball comes closer to it. However it needed to allow itself to be in the best field position to receive, the first approach that was utilised was to get the receiver to return to the centre of goal after it has passed the ball and switched to the receiver role. But due to a lack of a reliable localisation we weren’t able to return to the centre of goal, with this setback another method was devised. This time utilising the crouch of the goalie, this stance allowed for a wide coverage of area which increased our chances of catching the ball and then quickly returning it to the other robot. This method had one flaw though the goalie crouch took a very long time to stand up from, with a 2 minute time constraint this was no a possible solution. Due to time constraints we chose to go with no way of optimising the receiver’s position.

In order to avoid the robots entering the buffer zone in the centre of the field, due to a lack of field line detection we simply made sure that our kick was strong enough to away go beyond the buffer zone but not outside of the field.

3.2.5 Results

During initial testing of our solution to the passing challenge we were able to get the robot to pass between themselves at least 2 times, whilst this solution was somewhat simplistic and naive for the time constraints and the demands of the competition it had achieved its purpose. However as we are all students of Murphy’s Law during the actual run of the passing challenge our robot had fallen over which heavily skewed its localisation goal heading data resulting in the robot kicking the ball outside of the field.

Another run was done after this official run and the robot were able to pass between themselves thrice; the irony of the situation is that was the best performance achieved within the league. Overall no teams were able to achieve passing it their official runs; hence there was no winner in this challenge.

3.3 Lessons Learned

Our entry at the 2009 Robocup Symposium highlighted a few aspects of writing software for the Nao platform that require particular attention if we are to be more competitive in future years:

Robust Vision Our vision system needs to become less dependent on colour segmentation, and be able to deal with greater variance in lighting conditions.

Accurate Localisation Not having an accurate world model had a crippling effect on our ability to play soccer. This module needs to be addressed as a priority.

High Level Behaviours It must be possible to represent behaviours in a high-level format, to allow for more rapid prototyping. The output of the behaviour module should not depend on the runtime performance of other modules.

Unit Testing & Integration Testing A higher emphasis has to be given to writing tests for every class developed for the rUNSWift broker. Last-minute code changes with no regression
test suite caused some disastrous issues when playing games. Furthermore, all modules must be tested together, on the robot, highly frequently. With a full-time team, it should be a requirement that the nightly ‘master’ build be tested on the robots at the start of each day of work. Then fixing bugs should be giving priority over any new development.

**Debugging Tools**  rUNSWift needs an integrated suite of debugging utilities that allow problems with vision, localisation and behaviours to be detected and diagnosed rapidly. These tools need to have the ability to log all data processed by the robot so that errors can be reproduced off-line.
Bibliography

References


A Blackboard.hpp

#ifndef _BLACKBOARD_HPP_
#define _BLACKBOARD_HPP_

using namespace std;

#ifndef BLACKBOARD_HOOKS
enum OperationHook {READ_HOOK, WRITE_HOOK, NUM_OPERATION_HOOK_TYPES};
enum TimingHook {PRE_HOOK, POST_HOOK, NUM_TIMING_HOOK_TYPES};
#endif /* BLACKBOARD_HOOKS */

/** *
 * Macro wrapping the push of <code>value</code> to <code>queue</code> on the blackboard. *
 * @param queue the queue of the blackboard being written to *
 * @param value the value to be pushed *
 */
#define pushToBlackboardQueue(queue, value) \blackboard.write(&blackboard.queue, value)

/** *
 * Macro wrapping the write of <code>value</code> to <code>component</code> on the blackboard. *
 * @param component the component of the blackboard being written to *
 * @param value the value to be written *
 */
#define valueToBlackboard(component, value) \blackboard.write(&blackboard.component, value)

/** *
 * Macro wrapping the write of <code>count</code> bytes from <code>fd</code> to <code>component</code> on the blackboard. *
 * @param component the component of the blackboard being written to *
 * @param fd file descriptor to be written from *
 * @param count number of bytes to be written *
 * @return number of bytes actually written *
 */
#define fDToBlackboard(component, fd, count) \blackboard.write(&blackboard.component, fd, count)

/** *
 * Macro wrapping a read of <code>component</code> off the blackboard. *
 * @param component the component of the blackboard being read from *
 * @return the value of <code>component</code> *
 */
#define readFromBlackboard(component) \blackboard.read (&blackboard.component)

/** *
 * Macro wrapping a pop of <code>queue</code> off the blackboard. *
 * @param queue the queue of the blackboard being pop from *
 * @return the value pop from <code>queue</code> *
 */
#define popFromBlackboardQueue(queue)  
    blackboard.read(&blackboard.queue)

#ifndef BLACKBOARD_HOOKS  
/**  
 * Macro wrapping the adding of a hook <code>function</code> to  
 * <code>component</code>.  
 * @param component the component of the blackboard being hooked  
 * @param function the hook being added. must be of type void (*)(void *),  
 * where the function parameter will be a void * to  
 * <code>component</code>  
 * @param operationHook operation of hook (READ_HOOK or WRITE_HOOK)  
 * @param timingHook timing of hook (PRE_HOOK or POST_HOOK)  
 * @return the value of <code>component</code>  
 */
#define addHookToBlackboard(component, function, operationHook, timingHook)  
    blackboard.addHook(&blackboard.component, function, operationHook,  
    timingHook);
#endif  

/**  
 * Stores all shared data.  
 * Also provides hook functionality  
 */  
class Blackboard{  
#ifndef BLACKBOARD_HOOKS  
private:  
/**  
 * The hash table that stores the hooks for any of the data types on the  
 * blackboard.  
 * A 2x2 array of hash tables mapping from components to a set of functions.  
 */  
    map<pointer, vector<void(*)(void *)>> hooks[NUM_OPERATION_HOOK_TYPES][NUM_TIMING_HOOK_TYPES];
#endif  
public:

/**  
 * Blackboard Constructor. Responsible for initialising ALL variables.  
 * Update this constructor as items are added to the blackboard.  
 * The variables should be initialised to something meaningful to the  
 * module retrieving the variables data  
 * NOTE: Any objects stored on the blackboard are assumed to have their  
 * contents initialised via their default constructor  
 */  
    Blackboard();

/**  
 * Cleans up memory allocated by the constructor  
 */  
    ~Blackboard();

/**  
 * Pop <code>component</code> from the blackboard <code>queue</code>.  
 * Calls pre and post read hooks, if set  
 * @param queue the queue to be popped  
 * @return reference of type <code>component</code> to the data from  
 * <code>queue</code>

36
template <class T>
const T read(ConcurrentQueue<T> *queue);

/**
 * Read <code>component</code> from the blackboard.
 * Calls pre and post read hooks, if set
 * @param component the data to be read
 * @return pointer of type <code>component</code> to the data in
 * <code>component</code>
 */
template <class T>
const T *read(const T *component);

/**
 * Write <code>count</code> bytes from <code>fd</code> to <code>component</code> in the blackboard.
 * Calls pre and post write hooks, if set
 * @param component location to be written to
 * @param fd file descriptor being written from
 * @param count number of bytes to be written
 * @return number of bytes actually written
 */

ssizet write(T component, int fd, size_t count);

/**
 * Push <code>value</code> to <code>component</code> in the blackboard.
 * Calls pre and post write hooks, if set
 * @param queue queue to be written to
 * @param value value to be written
 */

template <class T>
void write(ConcurrentQueue<T> *queue, const T &value);

/**
 * Write <code>value</code> to <code>component</code> in the blackboard.
 * Calls pre and post write hooks, if set
 * @param component location to be written to
 * @param value value to be written
 */

template <class T>
void write(T *component, const T &value);

#ifdef BLACKBOARD_HOOKS
/**
 * Call all hooks on <code>key</code>.
 * @see doHook
 * @param key pointer to the data location being hooked
 * @param operationHook flag operation for hook (READ_HOOK, WRITE_HOOK)
 * @param timingHook flag timing of hook (PRE_HOOK or POST_HOOK)
 * @return <code>true</code> if successful,
 * <code>false</code> otherwise (e.g. if the <code>key</code> is invalid)
 */
#endif
// template <class T>
void callHooks(void *key, OperationHook operationHook,
    TimingHook timingHook);

/**
 * Add a hook on <code>key</code>.
 *
 * @param key pointer to the data location being hooked
 * @param function a pointer to the hook function, which takes as its
 *     single parameter the <code>component</code> it is
 *     hooking. It is responsible for its own self-contained
 *     error reporting
 * @param operationHook flag operation for hook (READ_HOOK, WRITE_HOOK)
 * @param timingHook flag timing of hook (PRE_HOOK or POST_HOOK)
 * @return <code>true</code> if successful,
 *     <code>false</code> otherwise (e.g. if the <code>key</code> is
 *     invalid)
 */
// template <class T>
bool addHook(void *key,
    void (*function)(void *),
    OperationHook operationHook,
    TimingHook timingHook);
#endif

/**
 * Lock, increment a counter, signal and release lock on a condition
 * variable. This is used for thread synchronisation. Whenever this
 * function is called, counter increments by 1 and whenever the waiting
 * thread processes the data the counter is reset to 0.
 * @param conditionVariable the condition variable struct to be
 *     operated on.
 * @see waitOnConditionVariable
 */
void signalThread(ConditionVariable& conditionVariable);

/**
 * macro to make it look like everything else in blackboard
 */
#define signalWaitingThread(component) \
    blackboard.signalThread(blackboard.component)

/**
 * Lock, wait, reset counter and unlock on a condition variable. This
 * is used by a waiting thread such as Behaviour
 */
void waitOnVariable(ConditionVariable& conditionVariable);

#define waitOnConditionVariable(component) \
    blackboard.waitOnVariable(blackboard.component)

/**
 * Unlock the mutex. This should only be used in the destructor of
 * classes to ensure that it is not deadlocking the system.
 * @param conditionVariable the condition variable
 */
void unlock(ConditionVariable& conditionVariable);

#define tryUnlock(component) \
    blackboard.unlock(blackboard.component)
/*
 * From this point on, definitions for variables on the Blackboard are
 * declared. Omitted here for brevity.
 */

};

#include "Blackboard.tcc"

extern Blackboard blackboard;

#endif
B SanityChecks.cpp

#include <stdio.h>
#include <blackboard/Blackboard.hpp>
#include "SanityChecks.hpp"
#include "SanityConstant.hpp"
#include "VisionDef.hpp"
#include "DebugDraw.hpp"
#include "LeastSquaresLine.hpp"

#undef MAX
#undef MIN

#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

static Insanity ballSizeTest (const VisualObject &ball);
static Insanity ballOrangeTest (const VisualObject &ball);
static Insanity sanitizeGoalPost (const GoalObject &goal, Color color);
static Insanity postColourTest (const GoalObject &goal, Color color);

/**
 * The ball colour tester will check pixels (radius / BALL_COLOUR_DENSITY) pixels apart
 */
#define BALL_COLOUR_DENSITY 5

/**
 * Percent of ball radius to skip as edge
 */
#define BALL_COLOUR_EDGE_SKIP_PERCENT 20

/**
 * Maximum edge size to be skipped
 */
#define BALL_COLOUR_EDGE_SKIP_MAX 5

/**
 * Percent of orange pixels which must be contained within the ball for it not to be invalidated
 */
#define BALL_COLOUR_ORANGE_PERCENT 70

/**
 * Percent of robot pixels which must be contained within the ball for it to be invalidated
 */
#define BALL_COLOUR_ROBOT_RED_PERCENT 20

/**
 * Percent of yellow goal pixels which must be contained within the ball for it to be invalidated
 */
#define BALL_COLOUR_YELLOW_GOAL_PERCENT 20

/**
 * The goal colour tester will check pixels
 */
* (goalWidth / BALL_COLOUR_DENSITY / 2)
* pixels apart
*/
#define GOAL_COLOUR_DENSITY 10
/
/**
* Percent of goal width to skip as edge
*/
#define GOAL_COLOUR_EDGE_SKIP_PERCENT 20
/
/**
* Maximum edge size to be skipped
*/
#define GOAL_COLOUR_EDGE_SKIP_MAX 5
/
/**
* Percent of goal pixels which must be contained
* within the ball for it not to be invalidated
*/
#define GOAL_COLOUR_COLOUR_PERCENT 70
/
/**
* Maximum diameter of ball when using the bottom camera
*/
#define BALL_SIZE_BOTTOM_MAX
/
/**
* Maximum diameter of ball when using the top camera
*/
#define BALL_SIZE_TOP_MAX
#define STRONG_GOAL_MIN_WIDTH 10
#define STRONG_GOAL_MIN_HEIGHT 50
#define STRONG_GOAL_MIN_SCANLINES 6
#define STRONG_GOAL_AUTO_SCANLINES 10
#define STRONG_GOAL_PAIR_WIDTH_TOL_PERC 30

Insanity sanitizeBlueGoalPost (const GoalObject &post)
{
    return sanitizeGoalPost (post, cBEACON_BLUE);
}

Insanity sanitizeYellowGoalPost (const GoalObject &post)
{
    return sanitizeGoalPost (post, cBEACON_YELLOW);
}

Insanity sanitizeGoalPost (const GoalObject &post, Color c)
{
    Insanity ret;
    if ((ret = postColourTest (post, c)) != SANE) {
        return ret;
    }
    return SANE;
}

Insanity postColourTest (const GoalObject &post, Color color)
if the goal angled more than 45 degrees its insane

const LeastSquaresLine &l = post.mid;
if (! l.isAbsSlopeGT1()) {
    return GOAL_TOO_SLANTED;
}

int x, y;
int xstart, ystart, xend, yend;
int goalPixels = 0, tested = 0;

int edgeSkip = MIN(GOAL_COLOUR_EDGE_SKIP_MAX, (GOAL_COLOUR_EDGE_SKIP_PERCENT * post.aveWidth) / 100 + 1);

int testWidth2 = (post.aveWidth - edgeSkip) / 2;

int step = (testWidth2 / GOAL_COLOUR_DENSITY) + 2;

ystart = MAX(IMAGE_EDGE, post.bb.y);
yend = MIN(BASIC_HEIGHT - IMAGE_EDGE, post.bb.endy);
for (y = ystart; y < yend; y += step * 3) {
    int cx = l.predictX(y);
    xstart = MAX(IMAGE_EDGE, cx - testWidth2);
    xend = MIN(BASIC_WIDTH - IMAGE_EDGE, cx + testWidth2);
    for (x = xstart; x < xend; x += step) {
        int c = CorrectedImage::classify(x, y);
        if (c == color) {
            DEBUG_DRAW_PIXEL(x, y, 105, 78, 64);
            goalPixels ++;
        }
    }
    tested ++;
}

// printf("goal %d %d %d %d\n", testWidth2, edgeSkip, step, goalPixels, tested);
if (tested == 0) {
    return SANE;
}
if (((goalPixels * 100 / tested) < GOAL_COLOUR_COLOUR_PERCENT) {
    // printf("Post invalidated: not coloured enough\n");
    return GOAL_NOT_MUCH_COLOUR;
} else {
    return SANE;
}

Insanity sanitizeBall (const VisualObject &ball)
{
    Insanity ret;
    if ((ret = ballSizeTest (ball)) != SANE) {
        return ret;
    } else if ((ret = ballOrangeTest (ball)) != SANE) {
        return ret;
    }
    return SANE;
}
Insanity ballSizeTest (const VisualObject &ball)
{
    return SANE;
}

Insanity ballOrangeTest (const VisualObject &ball)
{
    int x, y;
    int xstart, ystart, xend, yend;
    int orange = 0, robotRed = 0, yellow = 0, tested = 0;

    int edgeSkip = MIN (BALL_COLOUR_EDGE_SKIP_MAX, (BALL_COLOUR_EDGE_SKIP_PERCENT * ball.radius) / 100 + 1);

    int testRad = ball.radius - edgeSkip;
    int step = (testRad / BALL_COLOUR_DENSITY) + 1;

    if (testRad <= 0) {
        return SANE;
    }

    int testRad2 = testRad * testRad;

    xstart = MAX (IMAGE_EDGE, ball.cx - testRad);
    ystart = MAX (IMAGE_EDGE, ball.cy - testRad);
    xend = MIN (BASIC_WIDTH - IMAGE_EDGE, ball.cx + testRad);
    yend = MIN (BASIC_HEIGHT - IMAGE_EDGE, ball.cy + testRad);

    for (x = xstart; x < xend; x += step) {
        for (y = ystart; y < yend; y += step) {
            /**
             * Check that the point is within the ball
             * /
            int rx = (x - ball.cx);
            int ry = (y - ball.cy);
            if (rx * rx + ry * ry <= testRad2) {
                DEBUG_DRAW_PIXEL (x, y, 82, 91, 240);
                Color c = CorrectedImage::classify (x, y);
                if (c != eWHITE) {
                    if (c == eBALL) {
                        DEBUG_DRAW_PIXEL (x, y, 105, 78, 64);
                        orange ++;
                    }
                    if (c == eBEACON_YELLOW) {
                        yellow ++;
                    }
                    if (c == eROBOT_RED) {
                        robotRed ++;
                    }
                }
                tested ++;
            }
        }
    }

    // printf ("ball %d %d %d %d %d %d\n",
    // testRad, edgeSkip, step, orange, robotRed, yellow, tested);
    if (tested == 0) {
        return SANE;
    }
if ((orange * 100 / tested) < BALL_COLOUR_ORANGE_PERCENT) {
    // printf("Ball invalidated: not orange enough\n");
    return BALL_NOT_MUCH_ORANGE;
} else if ((robotRed * 100 / tested) > BALL_COLOUR_ROBOT_RED_PERCENT) {
    // printf("Ball invalidated: too reddish\n");
    return BALL_TOO MUCH ROBOT RED;
} else if ((yellow * 100 / tested) > BALL_COLOUR_YELLOW_GOAL_PERCENT) {
    // printf("Ball invalidated: too yellowish\n");
    return BALL_TOO MUCH GOAL YELLOW;
} else {
    return SANE;
}

bool isStrongSinglePost (const GoalObject &post) {
    if (post.lineCount >= STRONG_GOAL_AUTO_SCANLINES) {
        return true;
    }
    if (post.aveWidth < STRONG_GOAL_MIN_WIDTH) {
        return false;
    }
    int height = post.bb_endy - post.bb_y;
    if (height < STRONG_GOAL_MIN_HEIGHT) {
        return false;
    }
    if (post.lineCount < STRONG_GOAL_MIN_SCANLINES) {
        return false;
    }
    return true;
}

bool isStrongPostPair (const GoalObject &post1, const GoalObject &post2) {
    int widthDiffPerc = abs((post1.aveWidth - post2.aveWidth) * 100 /
        ((post1.aveWidth + post2.aveWidth) / 2));
    if (widthDiffPerc < STRONG_GOAL_PAIR_WIDTH_TOL_PERC) {
        return false;
    }
    return true;
}

void sanitizeGoalInBall (FrameInfo &frameInfo) {
    VisualObject &ball = frameInfo.vob[vobBall];
    VisualObject &yellow1 = frameInfo.vob[vobYellowGoalPost1];
    VisualObject &yellow2 = frameInfo.vob[vobYellowGoalPost2];
    /* allow extra radius for goal height */
    if (yellow1.height < ball.radius * 3) {
        int dist = hypot(ball.cx - yellow1.cx, ball.cy - yellow2.cy);
        /* allow for two radii between goal */
        if (dist < ball.radius * 2) {
            printf("VISION: killed goal in ball\n");
            yellow1.reset();
        }
    }
/* allow extra radius for goal height */
if (yellow2.height < ball.radius * 3) {
    int dist = hypot(ball.cx - yellow2.cx, ball.cy - yellow2.cy);
    /* allow for two radii between goal */
    if (dist < ball.radius * 2) {
        printf("VISION: killed goal in ball\n");
        yellow2.reset();
    }
}

}
```cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ioctl.h>
#include <unistd.h>
#include <linux/videodev2.h>

#define DEV_VIDEO "/dev/video0"

typedef enum
{
  IO_METHOD_READ, 
  IO_METHOD_MMAP, 
  IO_METHOD_USERPTR, 
  IO_METHOD_FILE
} io_method;

#define DEFAULT_IO_METHOD IO_METHOD_MMAP

#define NUM_FRAME_BUFFERS 4

#define USE_V4L

#define USE_V4L

#ifndef USE_V4L
#endif

#ifndef USE_V4L
#endif

enum WhichCamera
{
  whichCameraError = -1,
  topCamera = 0,
  bottomCamera = 1,
  numCameras = 2
};

class NaoCamera
{
public:
  /* Constructor
   * opens the device, calibrates it, and sets it up for streaming
   */
  NaoCamera(io_method method = DEFAULT_IO_METHOD);

  /* Destructor
   * closes the device
   */
  virtual ~NaoCamera();

  /* tells the called the correct image dimensions
   * @param width will be set to the image’s width
   * @param height will be set to the image’s height
   * @return if a correct format has been set
   */
  virtual bool imgDimensions(int& width, int& height);
};
```
virtual bool getYUV422();

bool setCamera(WhichCamera whichCamera);

WhichCamera getCamera();

void process_image(const uint8_t*p);

bool read_frame();

void setCameraSettings(WhichCamera camera);

virtual bool init_camera();

void init_buffers(void);

void start_capturing(void);
/**
 * turns off the data stream
 */
void stop_capturing(void);

/**
 * cleans up the buffers
 */
void uninitBuffers(void);

/**
 * closes the device
 */
void close_device(void);

/**
 * set mmap to write to the image buffer
 */
void init_mmap(void);

/**
 * set mmap to write to the image buffer
 */
void init_userp(void);

/**
 * just clears the image buffer
 */
void init_read(void);

/**
 * device file handle for camera. −1 indicates uninitialised
 */
int fd;

/**
 * the location and length for each saved frame
 */
struct Buffer {
    /**
     * the location for each frame. in kernel space in mmap mode
     */
    uint8_t *start;

    /**
     * the length for each frame
     */
    size_t length;
};

/**
 * stores the location and length for each possible saved frame
 */
Buffer buffers[NUM_FRAME_BUFFERS];

/**
 * the actual number of frames we’re saving. device dependent
 */
unsigned int n_buffers;
/**
 * the io method we are using right now
 */
io_method io;

/**
 * device handle for i2c bus. used for switching cameras.
 */
int i2cBus;

unsigned int imageSize;

int format;
int colourSpace; // REVIEW: american spelling fixed
int fps;

int frame;

// Keep track of the current camera. if unknown should be set
// to whichCameraError
WhichCamera currentCamera;

struct v4l2_queryctrl queryctrl;
struct v4l2_querymenu querymenu;

/**
 * When dealing with mmap'ed buffers, we should
 * 1. Dequeue a buffer and write its pointer to the blackboard
 * 2. Allow perception to process the image
 * 3. Enqueue the buffer again so it can be reused
 * 4. Before we were not allowing perception to process the current frame
 *    before enqueuing the buffer again, which could have potentially have
 *    lead to split images
 */
struct v4l2_buffer lastDequeued;
};

#endif
D NaoCamera.cpp

#include <ctime>
#include <iostream>
#include <malloc.h>
#include <limits.h>
#include <stdio.h>
#include <sys/mman.h>
#ifndef OFFLINE
#include <alvisionimage.h>
#include <alvisiondefinitions.h>
#endif
#include "blackboard/Blackboard.hpp"
#include "act/speech/Speech.hpp"
#include "config.h"
#include "sense/vision/NaoCamera.hpp"
#include "sense/vision/VisionModule.hpp"
#include "share/Common.hpp"
#include "share/Exception.hpp"
#include "share/Timer.hpp"
#include "utils/i2c-dev.h"
#include "utils/log.hpp"

#define TEST_IMAGE "dump.yuv"
// #define TEST_IMAGE "frame0.yuv"
#define NUM_CONTROLS 8

/**
 * Controls:
 * Brightness MAX = 100
 */

s32 controlIds[NUM_CONTROLS] =

s32 controlValues_dark[numCameras][NUM_CONTROLS] =
{ {100, 75, 128, 198, 128, 5, true, true},
  {100, 75, 128, 128, 128, 5, false, false} };

s32 controlValues_hotel[numCameras][NUM_CONTROLS] =
{ {100, 75, 128, 128, 128, 5, true, true},
  {100, 75, 128, 128, 128, 5, false, false} };

s32 controlValues_lights[numCameras][NUM_CONTROLS] =
{ {100, 75, 128, 128, 128, 5, true, true},
  {100, 75, 128, 128, 128, 5, false, false} };

#ifdef LIGHTS_ON
  s32 (*controlValues)[NUM_CONTROLS] = controlValues_lights;
#else
  #if 0
    s32 (*controlValues)[NUM_CONTROLS] = controlValues_hotel;
  #else
    s32 (*controlValues)[NUM_CONTROLS] = controlValues_dark;
  #endif
#endif

/**
 * reads the system error and writes it to the log, then throws an exception
 */
inline void errno_throw(const char *s)
{
    printf(ERROR, "%s error %d, %s\n", s, errno, strerror(errno));
    throw Exception(\_FILE\_, strerror(errno));
}

#define CLEAR(x) memset(&(x), 0, sizeof(x))

/*
* These addresses, commands and flags are given to us by aldeberan. They enable
* talking to and controlling the camera controller
* I2C_DEVICE – the i2c bus that the camera controller resides on
* I2C_SLAVE – the flag to indicate slave mode on the i2c bus. allows
* writing to the camera controller
* DSPIC_I2C_ADDR – the address of the camera controller on the i2c bus
* DSPIC_SWITCH_REG – register of the active camera. can be read to determine
* camera or set to change camera.
* currently 0x01 is top camera and 0x02 is bottom.
*/
#define I2C_DEVICE "/dev/i2c-0"
#define I2C_SLAVE 0x0703
#define DSPIC_I2C_ADDR 0x8
#define DSPIC_SWITCH_REG 220

using namespace AL;

NaoCamera::NaoCamera(io_method method): currentCamera(whichCameraError){
    io = method;
    if (io == IO_METHOD_FILE) {
        /* set so that imgDimensions returns correctly **/
        format = kQVGA;
        fd = -1;

        int width = 0;
        int height = 0;
        imgDimensions(width, height);
        imageSize = width * height * 2; /* bytes per yuv pixel */

        frame = 0;

        // open your camera and set the parameters
        fd = open(TEST_IMAGE, O_RDONLY);
        if (fd < 0) {
            log(ERROR) << "failed to open "TEST_IMAGE << endl;
            throw Exception(\_FILE\_, "inside NaoCamera.cpp, failed to open "TEST_IMAGE);
        }
        currentCamera = topCamera;

        init_buffers();
    }
return;
}

// pick 1: kVGA (640*480), kQVGA (320*240), kQQVGA (160*120).
format = kQVGA;

// pick 1:
// kYuvColorSpace, kyUvColorSpace, kyuVColorSpace,
// kYUVColorSpace, kYUV422InterlacedColorSpace, kRGBColorSpace.
// (definitions contained in alvisiondefinitions.h)
colourSpace = kYUVColorSpace;

// pick 1: 5, 10, 15, and 30 fps.
fps = 5;
fds = -1;
imageSize = 0;
frame = 0;

// open your camera and set the parameters
fd = open( DEV_VIDEO, O_RDWR);
if (fd < 0) {
    llog(ERROR) << " failed to open /dev/video0" << endl;
    throw Exception(,,FILE,, "inside NaoCamera.cpp, failed to open /dev/video0");
}
i2cBus = open(I2C DEVICE, O_RDWR);
if (i2cBus < 0) {
    llog(ERROR) << " : error opening I2C for connection to dsPIC" << endl;
    throw Exception(,,FILE,, "error opening I2C for connection to dsPIC");
}
initBuffers();

// init() the first camera
if (!initCamera()){
    lprintf(FATAL, "Error initializing camera!
");
    throw Exception(,,FILE,, "Error initializing camera");
}

// force getCamera to check the hardware
currentCamera = getCamera();

start_capturing();
// swap to and init() the second camera
// note that there is no swap back
setCamera((currentCamera==topCamera)?bottomCamera:topCamera);

// because set camera starts capturing
stop_capturing();

if (!initCamera()){
    lprintf(FATAL, "Error initializing second camera!
");
    throw Exception(,,FILE,, "Error initializing second camera");
}
start_capturing();
setCamera(*readFromBlackboard(useBottomCamera)?bottomCamera:topCamera);
SAY("Cameras initialized");
}
void NaoCamera::setCameraSettings(WhichCamera camera) {
    if (io == IO_METHOD_FILE) {
        /* not applicable for file io */
        return;
    }
    // struct v4l2_queryctrl queryctrl;
    struct v4l2_control control;

    CLEAR(control);

    /**
     * macro to set a control on the camera. reports errors but does not stop.
     *
     * @param controlId the id of the control
     * @param controlValue the value to set controlId to
     */
    #define setControl(controlId, controlValue)
        { 
        control.id = controlId;
        control.value = controlValue;
        if(-1 == ioctl(fd, VIDIOC_S_CTRL, &control) && errno != ERANGE){ 
            lprintf(ERROR, "%s error %d, %s", "VIDIOC_S_CTRL", errno, strerror(errno)); 
        } 
    }

    // setControl(V4L2_CID_SATURATION, 128);
    setControl(V4L2_CID_HUE, 0); // Auto Exposure
    setControl(V4L2_CID_AUDIO_MUTE, false); // Auto Exposure
    setControl(V4L2_CID_AUTO_WHITE_BALANCE, false);
    // setControl(V4L2_CID_EXPOSURE, 504);
    setControl(V4L2_CID_AUTOGAIN, false);
    for(unsigned int controlIndex = 0; controlIndex < NUM_CONTROLS; ++controlIndex)
        setControl(controlIds[controlIndex], controlValues[camera][controlIndex]);
}

NaoCamera::~NaoCamera() {
    stopCapturing();
    unInitBuffers();
    closeDevice();
    if(0 > i2cBus && 0 > close(i2cBus)) {
        llog(ERROR) << ___PRETTY_FUNCTION___ << " : error closing I2C connection to dsPIC" 
            << endl;
    }
}

bool NaoCamera::imgDimensions(int& width, int& height) {
    switch (format) {
    case kVGA:
        width = 640;
        height = 480;
        break;
    case kQVGA:
        width = 320;
        height = 240;
        break;
    case kQQVGA:
        width = 160;
        height = 120;
        break;
    }
break;
default:
    return false;
}
return true;
}

void NaoCamera::process_image(const uint8 *p) {
    valueToBlackboard(cameraImage, p);
}

bool NaoCamera::read_frame(void) {
    unsigned int i;

    switch (io) {
    /* reading from file and video device are exactly the same */
    case IO_METHOD_FILE:
        getchar();
        if (-1 == read(fd, buffers[0].start, buffers[0].length)) {
            /* Try resetting the stream to the beginning */
            errno = 0;
            while (lseek(fd, 0, SEEK_SET) == -1 && errno == EAGAIN);
            if (errno != 0 && errno != EAGAIN) {
                errno_throw("read");
            }
            if (-1 == read(fd, buffers[0].start, buffers[0].length)) {
                errno_throw("read");

            }
            process_image(buffers[0].start);
            break;
    }
    case IO_METHOD_READ:
        if (-1 == read(fd, buffers[0].start, buffers[0].length)) {

            switch (errno) {
            case EAGAIN:
                return false;

            case EIO:
                /* Could ignore EIO, see spec. */
                /* fall through */
            default:
                errno_throw("read");
            }
            process_image(buffers[0].start);
            break;
    }
    case IO_METHOD_MMAP:
        if (lastDequeued.index != UINT_MAX) {
            if (-1 == ioctl(fd, VIDIOC_QBUF, &lastDequeued)) {
                errno_throw("VIDEOC_BUF");
            }
        }
    }

    54
CLEAR (lastDequeued);

lastDequeued.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
lastDequeued.memory = V4L2_MEMORY_MMAP;

if (-1 == ioctl(fd, VIDIOC_DQBUF, &lastDequeued)) {
    switch (errno) {
    case EAGAIN:
        return false;
        
    case EIO:
        /* Could ignore EIO, see spec. */
        /* fall through */
        
    default:
        errno_throw ("VIDIOC_DQBUF");
    }
}

assert (lastDequeued.index < n_buffers);

process_image (buffers[lastDequeued.index].start);
break;

case IO_METHOD_USERPTR:
    // TODO verify that lastDequeued fix works for user pointers
    if (lastDequeued.index != UINT_MAX) {
        if (-1 == ioctl(fd, VIDIOC_QBUF, &lastDequeued)) {
            errno_throw ("VIDEOC_BUF");
        }
    }
}

CLEAR (lastDequeued);

lastDequeued.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
lastDequeued.memory = V4L2_MEMORY_USERPTR;

if (-1 == ioctl(fd, VIDIOC_DQBUF, &lastDequeued)) {
    switch (errno) {
    case EAGAIN:
        return false;
        
    case EIO:
        /* Could ignore EIO, see spec. */
        /* fall through */
        
    default:
        errno_throw ("VIDIOC_DQBUF");
    }
}

for (i = 0; i < n_buffers; ++i)
    if (lastDequeued.m.userptr == (unsigned long) buffers[i].start
        && lastDequeued.length == buffers[i].length)
        break;
assert (i < n_buffers);

process_image ( (uint8 *) lastDequeued.m.userptr );

break;
}

return true;
}

bool NaoCamera::getYUV422() {
    fd_set fds;
    struct timeval tv;
    int r;
    FD_ZERO (&fds);
    FD_SET (fd, &fds);
    
    /* Timeout. */
    tv.tv_sec = 2;
    tv.tv_usec = 0;

    if (io == IO_METHOD_MMAP) {
        r = select (fd + 1, &fds, NULL, NULL, &tv);

        if (-1 == r) {
            if (EINTR == errno)
                return false; // just skip this frame because we were interrupted

        errno_throw ("select");
        }

        if (0 == r) {
            lprintf(ERROR, "select timeout\n");
            SAY("Camera took more than 2 seconds to respond."
                "I'm probably overheated");
        }
    }

    if (!read_frame ())
        lprintf(WARNING, "was asked to read frame again\n");
    return true;
}

bool NaoCamera::init_camera () {
    // GET video device information
    v4l2_std_id esid0;
    int test;
    test = ioctl (fd, V4IOC_G_STD, &esid0);
    if (test != 0) {
        perror ("ioctl 1");
        llog (ERROR) << "failed ioctl with error code " << test << endl;
        throw Exception (.FILE,, "inside NaoCamera.cpp, failed ioctl");
    }

    // set video device standard
    switch (format) {
        // aldebaran is invalidly using this field
        // The 32 most significant bits are reserved for custom (driver defined)
        // video standards.
// There isn't supposed to be a way to specify QQVGA with this field
// REVIEW: can these not be defined earlier?

switch (format) {
case kVGA:
    esid0 = (v4l2_std_id)0x08000000; /* VGA */
    break;
case kQVGA:
    esid0 = (v4l2_std_id)0x04000000; /* QQVGA */
    break;
default:
    throw Exception("FILE", "inside NaoCamera.cpp, unknown format");
}
test = ioctl(fd, VIDIOC_S_STD, &esid0);
if (test != 0) {
    perror("ioctl 2");
    llog(ERROR) << "failed ioctl 2 error " << test << endl;
    #ifdef BROKER
    throw Exception("FILE", "inside NaoCamera.cpp, failed ioctl 2");
    #endif
}

// set video device format
struct v4l2_format fmt0;
memset(&fmt0, 0, sizeof(fmt0));
fmt0.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt0.fmt.pix.field = V4L2_FIELD_NONE;

switch (format) {
case kVGA:
    fmt0.fmt.pix.width = 640;
    fmt0.fmt.pix.height = 480;
    break;
case kQVGA:
    fmt0.fmt.pix.width = 320;
    fmt0.fmt.pix.height = 240;
    break;
default:
    throw Exception("FILE", "inside NaoCamera.cpp, unknown format");
}

//fmt0.fmt.pix.pixelformat = 0x56595559;
fmt0.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
test = ioctl(fd, VIDIOC_S_FMT, &fmt0);
if (test != 0) {
    perror("ioctl 3");
    llog(ERROR) << "failed ioctl 3 error " << test << endl;
    throw Exception("FILE", "NaoCamera.cpp, failed ioctl 3");
}

// set fps
struct v4l2_streamparm parm;
int overflow = 0x7FFFFFFF;

parm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
test = ioctl(fd, VIDIOC_G_PARM, &parm);
printf("overflow = %d\n", overflow);
if (test != 0) {
    perror("ioctl 4");
    llog(ERROR) << "failed ioctl 4 " << endl;
    throw Exception("FILE", "NaoCamera.cpp, failed ioctl 4");
}
parm.parm.capture.timeperframe.numerator = 1;
parm.parm.capture.timeperframe.denominator = fps;
parm.parm.capture.capability = V4L2_CAP_TIMEPERFRAME;
ioctl(fd, VIDIOC_S_PARM, &parm);

imageSize = fmt0.fmt.pix.sizeimage;
setCameraSettings(getCamera());

return true;

} // NaoCamera: : stop_capturing

} // NaoCamera: : close_device
errno_throw("close");

fd = -1;
}

void NaoCamera::init_mmap(void)
{
    struct v4l2_requestbuffers req;

    CLEAR(req);

    req.count = NUM_FRAME_BUFFERS;
    req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    req.memory = V4L2_MEMORY_MMAP;

    if (-1 == ioctl(fd, VIDIOC_REQBUFS, &req)) {
        if (EINVAL == errno) {
            printf(ERROR, "%s does not support
" "memory mapping\n", DEV_VIDEO);
            exit(EXIT_FAILURE);
        } else {
            errno_throw("VIDIOC_REQBUFS");
        }
    }

    if (req.count < 1) {
        printf(ERROR, "Insufficient buffer memory on %s\n", DEV_VIDEO);
        exit(EXIT_FAILURE);
    }

    CLEAR(buffers);

    if (!buffers) {
        printf(ERROR, "Out of memory\n");
        exit(EXIT_FAILURE);
    }

    for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
        struct v4l2_buffer buf;

        CLEAR(buf);

        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = n_buffers;

        if (-1 == ioctl(fd, VIDIOC_QUERYBUF, &buf))
            errno_throw("VIDIOC_QUERYBUF");

        buffers[n_buffers].length = buf.length;
        buffers[n_buffers].start = (uint8*)
            mmap(NULL /* start anywhere */,
                 buf.length,
                 PROT_READ | PROT_WRITE /* required */,
                 MAP_SHARED /* recommended */,
                 fd, buf.m.offset);

        if (MAP_FAILED == buffers[n_buffers].start)
            errno_throw("mmap");

        59
void NaoCamera::init_buffers(void) {
    if (io == IO_METHOD_FILE) {
        /* This is all that is needed if we are not going to use the camera */
        init_read();
        return;
    }

    struct v4l2_capability cap;
    struct v4l2_cropcap cropcap;
    struct v4l2_crop crop;
    struct v4l2_format fmt;
    v4l2_std_id esid0;
    unsigned int min;

    if (-1 == ioctl(fd, VIDIOC_QUERYCAP, &cap)) {
        if (EINVAL == errno) {
            lprintf(ERROR, "%s is no V4L2 device\n", DEV_VIDEO);
            exit(EXIT_FAILURE);
        } else {
            errno.throw("VIDIOC_QUERYCAP");
        }
    }

    if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
        lprintf(ERROR, "%s is no video capture device\n", DEV_VIDEO);
        exit(EXIT_FAILURE);
    }

    switch (io) {
    case IO_METHOD_FILE:
        break;
    case IO_METHOD_READ:
        if (!(cap.capabilities & V4L2_CAP_READWRITE)) {
            lprintf(ERROR, "%s does not support read i/o\n", DEV_VIDEO);
            exit(EXIT_FAILURE);
        }
        break;
    case IO_METHOD_MMAP:
    case IO_METHOD_USERPTR:
        if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
            lprintf(ERROR, "%s does not support streaming i/o\n", DEV_VIDEO);
            exit(EXIT_FAILURE);
        }
        break;
    }

    /* Select video input, video standard and tune here. */
}
CLEAR (cropcap);

cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (0 == ioctl(fd, VIDIOC_CROPCAP, &cropcap)) {
  crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
  crop.c = cropcap.default; /* reset to default */
}

if (-1 == ioctl(fd, VIDIOC_S_CROP, &crop)) {
  switch (errno) {
    case EINVAL:
      /* Cropping not supported. */
      break;
    default:
      /* Errors ignored. */
      break;
  }
} else {
  /* Errors ignored. */
}

switch (format) {
// REVIEW: this is done somewhere else — maybe we can refactor it out
// aldebaran is invalidly using this field
// The 32 most significant bits are reserved for custom (driver defined)
// video standards.
// There isn’t supposed to be a way to specify QQVGA with this field
  case kVGA:
    esid0 = 0x08000000UL; /* VGA*/
    break;
  case kQVGA:
    esid0 = 0x04000000UL; /* QQVGA*/
    break;
}

if (-1 == ioctl(fd, VIDIOC_S_STD, &esid0))
  errno_throw("VIDIOC_S_STD");

CLEAR (fmt);

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

switch (format){
  case kVGA:
    fmt.fmt.pix.width = 640;
    fmt.fmt.pix.height = 480;
    break;
  case kQVGA:
    fmt.fmt.pix.width = 320;
    fmt.fmt.pix.height = 240;
    break;
}

fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
fmt.fmt.pix.field = V4L2_FIELD_ANY;

if (-1 == ioctl(fd, VIDIOC_S_FMT, &fmt))
  errno_throw("VIDIOC_S_FMT");

/* Note VIDIOC_S_FMT may change width and height. */
/* Buggy driver paranoia. */
min = fmt.fmt.pix.width * 2;
if (fmt.fmt.pix.bytesperline < min)
    fmt.fmt.pix.bytesperline = min;
min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
if (fmt.fmt.pix.sizeimage < min)
    fmt.fmt.pix.sizeimage = min;

imageSize = fmt.fmt.pix.sizeimage;

switch (io) {
    case IO_METHOD_FILE:
        break;
    case IO_METHOD_READ:
        init_read();
        break;
    case IO_METHOD_MMAP:
        init_mmap();
        break;
    case IO_METHOD_USERPTR:
        init_userp();
        break;
}

void NaoCamera::init_userp() {
    struct v4l2_requestbuffers req;
    unsigned int page_size;

    page_size = getpagesize();
    imageSize = (imageSize + page_size - 1) & ~(page_size - 1);

    CLEAR(req);
    req.count = NUM_FRAME_BUFFERS;
    req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    req.memory = V4L2_MEMORY_USERPTR;

    if (-1 == ioctl(fd, VIDIOC_REQBUFS, &req)) {
        if (EINVAL == errno) {
            lprintf(ERROR, "%s does not support "
                    "user pointer i/o\n", DEV_VIDEO);
            exit(EXIT_FAILURE);
        } else {
            errno_throw("VIDIOC_REQBUFS");
        }
    }

    CLEAR(buffers);

    if (!buffers) {
        lprintf(ERROR, "Out of memory\n");
        exit(EXIT_FAILURE);
    }

    for (n_buffers = 0; n_buffers < NUM_FRAME_BUFFERS; ++n_buffers) {
        buffers[n_buffers].length = imageSize;
        buffers[n_buffers].start = (uint8*) memalign(*/ boundary */ page_size,
void NaoCamera::start_capturing(void) {
    // assign an impossible index to lastDequeued so we know not to
    // re-enqueue if there have been no previous buffers
    lastDequeued.index = UINT_MAX;

    unsigned int i;
    enum v4l2_buf_type type;

    switch (io) {
        case IO_METHOD_FILE:
            case IO_METHOD_READ:
                /* Nothing to do. */
                break;
        case IO_METHOD_MMAP:
            for (i = 0; i < n_buffers; ++i) {
                struct v4l2_buffer buf;

                CLEAR (buf);

                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                buf.index = i;

                if (-1 == ioctl(fd, VIDIOC_QBUF, &buf))
                    errno_throw("VIDIOC_QBUF");
            }
            type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

            if (-1 == ioctl(fd, VIDIOC_STREAMON, &type))
                errno_throw("VIDIOC_STREAMON");
        break;
    case IO_METHOD_USERPTR:
        for (i = 0; i < n_buffers; ++i) {
                struct v4l2_buffer buf;

                CLEAR (buf);

                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_USERPTR;
                buf.index = i;
                buf.m.userptr = (unsigned long) buffers[i].start;
                buf.length = buffers[i].length;

                if (-1 == ioctl(fd, VIDIOC_QBUF, &buf))
                    errno_throw("VIDIOC_QBUF");
        }
    }
}
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl(fd, VIDIOC_STREAMON, &type))
    errno_throw("VIDIOC_STREAMON");

break;
}
}

void NaoCamera::init_read(void)
{
    CLEAR(buffers);

    if (!buffers) {
        lprintf(ERROR, "Out of memory\n");
        exit(EXIT_FAILURE);
    }

    buffers[0].length = imageSize;
    buffers[0].start = (uint8*) malloc(imageSize);

    if (!buffers[0].start) {
        lprintf(ERROR, "Out of memory\n");
        exit(EXIT_FAILURE);
    }
}

bool NaoCamera::setCamera(WhichCamera whichCamera){
    if (io == IO_METHOD_FILE) {
        //** not applicable for file io **/
        return true;
    }

    if (whichCamera == getCamera())
        return true;

    Timer timer;
    // cmd will be written to the camera controllers 'switch camera' register
    // we set it to the appropriate value.
    //REVIEW: perhaps we should define 0x01 and 0x02 earlier in case it changes...
    static unsigned char cmd[2] = {0, 0}; // aldeberan uses 2 bytes so we do too!
    switch (whichCamera) {
    case topCamera:
        cmd[0] = 0x01;
        break;
    case bottomCamera:
        cmd[0] = 0x02;
        break;
    default:
        return false;
    }

    // Connect to dsPIC through I2C, then set cameras
    if (ioctl(i2cBus, I2C_SLAVE, DSPIC_I2C_ADDR)) {
        llog(ERROR) << " : Can’t connect I2C to dsPIC" << endl;
        if (0 > close(i2cBus))
            llog(ERROR) << _PRETTY_FUNCTION_ << " : error closing I2C" << endl;
        i2cBus = -1;
        return false;
    }
// need to stop capture at switch time, so as not to switch in mid-frame
stop_capturing();
// notice only 1 byte is written
int size = i2c_smbus_write_block_data(i2cBus, DSPIC_SWITCH_REG, 1, cmd);
start_capturing();

if (size == -1) {
    log(ERROR) << "PRETTY_FUNCTION_ << " : error switching to bottom camera"
        << endl;
    if(0 > close(i2cBus))
        log(ERROR) << "PRETTY_FUNCTION_ << " : error closing I2C connection to dsPIC" << endl;
    i2cBus = -1;
    return false;
}
currentCamera = whichCamera;
lprintf(INFO, "setCamera took %ld useconds\n", timer.elapsed_us());
return true;
}

WhichCamera NaoCamera::getCamera() {
    if(currentCamera != whichCameraError) {
        return currentCamera;
    }

    // Connect to dsPIC through I2C
    if( iocctl(i2cBus, I2C_SLAVE, DSPIC_I2C_ADDR) ) {
        log(ERROR) << " : Can't connect I2C to dsPIC" << endl;
        if(0 > close(i2cBus))
            log(ERROR) << "PRETTY_FUNCTION_ << " : error closing I2C" << endl;
        i2cBus = -1;
        return whichCameraError;
    }

    // read the camera controller register which contains the current camera
    int val = i2c_smbus_read_byte_data(i2cBus, DSPIC_SWITCH_REG);
    if (val == -1) {
        log(ERROR) << "PRETTY_FUNCTION_ 
        " : error asking which camera is active to dsPIC" << endl;
        if(0 > close(i2cBus))
            log(ERROR) << "PRETTY_FUNCTION_ 
        " : error closing I2C connection to dsPIC" << endl;
        i2cBus = -1;
        return whichCameraError;
    }
    lprintf(INFO, ((val==0x1)?"Top Camera\n" : "Bottom Camera\n"));

currentCamera = ((val==0x1)?topCamera:bottomCamera);
return currentCamera;
}
E AnyBall.cpp

```cpp
#include "AnyBall.hpp"
#include "CorrectedImage.hpp"
#include "VisionDef.hpp"
#include "DebugDraw.hpp"
#include "VisualObject.hpp"
#include <blackboard/Blackboard.hpp>
#include <stdint.h>
#include <opencv/cv.hpp>

void AnyBall::findMyBalls (WhichCamera cam)
{
    IplImage* gray = cvCreateImage (cvSize (BASIC_WIDTH, BASIC_HEIGHT), 8, 1);
    CvMemStorage* storage = cvCreateMemStorage (0);
    Color c;

    const uint8_t *yuv = CorrectedImage::getYUV ();
    int32_t i = 0;
    while (i < BASIC_WIDTH * BASIC_HEIGHT) {
        uint8_t y1 = yuv[0];
        uint8_t u = yuv[1];
        uint8_t y2 = yuv[2];
        uint8_t v = yuv[3];

        c = CorrectedImage::classify (y1, u, v);
        if (c == cFIELD_GREEN) {
            gray->imageData[i] = 0;
        } else if (c == cWHITE) {
            gray->imageData[i] = 100;
        } else {
            gray->imageData[i] = 255;
        }
        i ++;

        c = CorrectedImage::classify (y2, u, v);
        if (c == cFIELD_GREEN) {
            gray->imageData[i] = 0;
        } else if (c == cWHITE) {
            gray->imageData[i] = 100;
        } else {
            gray->imageData[i] = 255;
        }
        i ++;

        yuv += 4;
    }

    // smooth it, otherwise a lot of false circles may be detected

    int32_t minBallRad, maxBallRad;

    if (cam == bottomCamera) {
        minBallRad = 20;
        maxBallRad = 100;
    } else {
        minBallRad = 0;
    }
```
maxBallRad = 30;
}
minBallRad = 0;
maxBallRad = 100;
cvSmooth (gray, gray, CV_GAUSSIAN, 21, 21);
CvSeq* circles = cvHoughCircles (gray, storage, CV_HOUGH_GRADIENT, 4,
gray->height/8, 100,
50,
minBallRad, maxBallRad);

uint8_t *data = (uint8_t *)gray->imageData;
for (int y = 0; y < BASIC_HEIGHT; y++) {
    for (int x = 0; x < BASIC_WIDTH; x++) {
        DEBUG_DRAW_PIXEL (x, y, *data, 128, 128);
        data++;
    }
}

int32_t biggestBall = 0;
int32_t biggestBallx = 0;
int32_t biggestBally = 0;
for (i = 0; i < circles->total; i++) {

    float *p = (float *)cvGetSeqElem (circles, i);
    int32_t x = cvRound (p[0]);
    int32_t y = cvRound (p[1]);
    int32_t r = cvRound (p[2]);

    if (rateMyBalls (cvRound (p[0]),
                    cvRound (p[1]),
                    cvRound (p[2]),
                    cam))
    {
        if (r > biggestBall) {
            biggestBall = r;
            biggestBallx = x;
            biggestBally = y;
            DEBUG_DRAW_CIRCLE (cvRound (p[0]),
                                cvRound (p[1]),
                                cvRound (p[2]),
                                82, 91, 240);
        }
    }
}

cvReleaseImage (&gray);
cvReleaseMemStorage (&storage);
VisualObject tmp;
tmp.reset ();
tmp.nFeatures = 0;
tmp.medianWidth = 0;
if (biggestBall != 0) {
    printf("VISION writing ball to blackboard\n");
    myBalls.cf = 500;
    myBalls.radius = biggestBall;
    myBalls.cx = biggestBallx;
    myBalls.cy = biggestBally;
    myBalls.nFeatures = 10;
} else {
    myBalls.cf = 0;
    myBalls.nFeatures = 10;
}

void AnyBall::grabMyBalls (VisualObject &ball) {
    ball = myBalls;
}

int32_t AnyBall::rateMyBalls (int32_t cx, int32_t cy, int32_t rad, WhichCamera cam) {
    int32_t x, y;
    int32_t xstart, ystart, xend, yend;
    int32_t outgreen = 0, ingreen = 0,
        outwhite = 0, inwhite = 0,
        outtested = 0, intested = 0;

    int32_t step = (rad / 5) + 1;
    int32_t inRad = rad - 3;
    int32_t outRad = rad + 3;
    int32_t testRad = outRad + step * 3;
    int32_t testRad2 = testRad * testRad;
    int32_t inRad2 = inRad * inRad;
    int32_t outRad2 = outRad * outRad;

    xstart = MAX (IMAGEEDGE, cx - testRad);
    ystart = MAX (IMAGEEDGE, cy - testRad);
    xend = MIN (BASIC_WIDTH - IMAGEEDGE, cx + testRad);
    yend = MIN (BASIC_HEIGHT - IMAGEEDGE, cy + testRad);

    for (x = xstart; x < xend; x += step) {
        for (y = ystart; y < yend; y += step) {
            /* Check that the point is within the ball */
            int32_t rx = (x - cx);
            int32_t ry = (y - cy);
            int32_t dist2 = rx * rx + ry * ry;

            Color c = CorrectedImage::classify (x, y);
            if (dist2 <= inRad2) {
                DEBUG_DRAW_PIXEL (x, y, 105, 78, 64);
                if (c == cWHITE) {
                    outgreen++;
            }
```c
int32_t maxInGreen, minOutGreen, maxOutWhite;
if (cam == bottomCamera) {
    maxInGreen = 20;
    minOutGreen = 50;
    maxOutWhite = 50;
} else {
    maxInGreen = 20;
    minOutGreen = 70;
    maxOutWhite = 30;
}

if (ingreen * 100) / intested > maxInGreen) {
    printf("Any Ball invalidated: too much inner green\n");
    return 0;
} else if ((outgreen * 100 / outtested) < 50) {
    printf("Any Ball invalidated: not enough outer green\n");
    return 0;
} else if (((outgreen + outwhite) * 100 / outtested) < minOutGreen) {
    printf("Any Ball invalidated: not enough outer green\n");
    return 0;
} else if (outwhite * 100 / outtested > maxOutWhite) {
    printf("Any Ball invalidated: too much outer white\n");
    return 0;
} else {
    return 1;
}
```
#ifndef __LINE_FAST_MMX_HPP
#define __LINE_FAST_MMX_HPP

#include "YUVPixel.h"
#include "PixelData.h"

/**
 * Fast asm routine used by Line to crunch raw YUVPixels into useful information
 * @param src Array of raw YUVPixels
 * @param dest Pre-allocated buffer for writing processed information
 * @param length length of line
 */
#ifdef __cplusplus
extern "C"
{
#endif

    void fastLineMMX1 ( YUVPixel * src , PixelData * dest , int length )
    __attribute__((cdecl));

    void fastLineMMX2 ( YUVPixel * src1 , PixelData * dest1 ,
                        YUVPixel * src2 , PixelData * dest2 ,
                        int length )
    __attribute__((cdecl));

#endif

#endif
G  MMX Routine Implementations

/**
 * This file is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include "config.h"

/**
 * struct YUVPixel
 * a YUVPixel contains 3 byte values for YUV
 * note it will be stored as a long
 */

.equ YUV_PIXEL_Y, 0
.equ YUV_PIXEL_U, 1
.equ YUV_PIXEL_V, 2

/* UNUSED */
.equ YUV_PIXEL_SIZE, 4

/**
 * struct PixelData
 * Note the alignment an 4 byte boundaries
 * so each YUV group can be read as a long
 */

.equ PIXEL_DATA_Y, 0
.equ PIXEL_DATA_U, 1
.equ PIXEL_DATA_V, 2

/* UNUSED */

.equ PIXEL_DATA_DY, 4
.equ PIXEL_DATA_DU, 5
.equ PIXEL_DATA_DV, 6

/* UNUSED */

.equ PIXEL_DATA_JY, 8
.equ PIXEL_DATA_JU, 9
.equ PIXEL_DATA_JV, 10

/* UNUSED */

.equ PIXEL_DATA_DJY, 12
.equ PIXEL_DATA_DJU, 13
.equ PIXEL_DATA_DJV, 14

/* UNUSED */

.equ PIXEL_DATA_Y_ZERO, 16
.equ PIXEL_DATA_U_ZERO, 17
.equ PIXEL_DATA_V_ZERO, 18
/* UNUSED */
.equ PIXEL_DATA_SIZE, 20

/**
 * dYUV thresholds. a zero is only a zero if its corresponding dl value is
 * greater than or equal to its threshold
 */

#ifndef LIGHTS_ON
.equ IY_THRESHOLD, 8
.equ IUV_THRESHOLD, 4
#else
.equ IY_THRESHOLD, 6
.equ IUV_THRESHOLD, 3
#endif

.sect .data
divide2_constant:
.long 0x7F7F7F7F
.long 0x7F7F7F7F
sign_constant:
.long 0x80808080
.long 0x80808080
.sect .text

/* note we are really storing a qword in little endian form */
YUV_threshold_constant:
.byte IY_THRESHOLD - 1
.byte IUV_THRESHOLD - 1
.byte IUV_THRESHOLD - 1
.byte 0
.byte IY_THRESHOLD - 1
.byte IUV_THRESHOLD - 1
.byte IUV_THRESHOLD - 1
.byte 0

YUV_neg_threshold_constant:
.byte -(IY_THRESHOLD - 1)
.byte -(IUV_THRESHOLD - 1)
.byte -(IUV_THRESHOLD - 1)
.byte 0
.byte -(IY_THRESHOLD - 1)
.byte -(IUV_THRESHOLD - 1)
.byte -(IUV_THRESHOLD - 1)
.byte 0

**
* fastLineMMX1 Omitted due to similarity to fastLineMMX2
* fastLineMMX1 only processes a single array, whilst
* fastLineMMX2 processes two arrays in parallel
*/

/**
 * Does the number crunching to convert two
 * arrays of YUV PIXEL to two arrays of YUV DATA
 * arrays must be of equal length
 *
* @YUV_PIXELS1 pointer to source YUV_PIXEL array
* @YUV_DATA1 pointer to dest YUV_DATA array
* @YUV_PIXELS2 pointer to source YUV_PIXEL array
* @YUV_DATA2 pointer to dest YUV_DATA array
* @LENGTH length of arrays
* @return undefined
* @see Line.cpp
*/

.equ fastLineMMX2_YUV_PIXELS1, 8
.equ fastLineMMX2_PIXEL_DATA1, 12
.equ fastLineMMX2_YUV_PIXELS2, 16
.equ fastLineMMX2_PIXEL_DATA2, 20
.equ fastLineMMX2_LENGTH, 24

/* local vars */
.equ fastLineMMX2_TAIL0, 8
.equ fastLineMMX2_TAIL1, 16
.equ fastLineMMX2_TAIL2, 24
.equ fastLineMMX2_TMP_YUV, 32

.global fastLineMMX2

.type fastLineMMX2, @function

fastLineMMX2:
pushl %ebp
movl %esp, %ebp

; we need 32 bytes locals */
subl $32, %esp

push %esi
push %edi

; length of the array in ecx for looping
; yuv pixels in esi (source index)
; yuv data in the edi (dest index)

/*
* if some dibshit gave us a line of length 0, then exit quick fast
*/
test %ecx, %ecx
jz fastLineMMX2_quick_exit

movl fastLineMMX2_LENGTH(%ebp), %ecx

movl fastLineMMX2_YUV_PIXELS1(%ebp), %esi
movl fastLineMMX2_YUV_PIXELS2(%ebp), %eax
movl fastLineMMX2_PIXEL_DATA1(%ebp), %edi
movl fastLineMMX2_PIXEL_DATA2(%ebp), %edx

; Before entering the loop, calculate the first 2 values */

; read in the YUV values from YUV_PIXELS1
movd (%esi), %mm0
punpckldq (%eax), %mm0
addl $YUV_PIXEL_SIZE, %esi
addl $YUV_PIXEL_SIZE, %eax

73
/* save the raw YUV values back to the PIXEL_DATA */
movd %mm0, PIXEL_DATA_Y(%edi)
movq %mm0, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_Y(%edx)

/∗ divide by two ∗/
movq (divide2_constant), %mm7
psrlq $1, %mm0
pand %mm7, %mm0

/∗ the first TAIL value is 0 ∗/
/∗ we also initialize TAIL2 to 0 as it is not yet relevant ∗/
pxor %mm3, %mm3
movq %mm3, fastLineMMX2_TAIL1(%ebp)
movq %mm3, fastLineMMX2_TAIL2(%ebp)
movd %mm3, PIXEL_DATA_DY(%edi)
movd %mm3, PIXEL_DATA_DY(%edx)

/∗ the first I (integral) value is also 0 ∗/
pxor %mm4, %mm4
movd %mm4, PIXEL_DATA_DY(%edi)
movd %mm4, PIXEL_DATA_DY(%edx)

/∗ as is dl ∗/
movd %mm4, PIXEL_DATA_DIY(%edi)
movd %mm4, PIXEL_DATA_DIY(%edx)

/∗ the first pixel of a line is always a zero ∗/
pcmpeqd %mm4, %mm4
movd %mm4, PIXEL_DATA_YZERO(%edi)
movd %mm4, PIXEL_DATA_YZERO(%edx)

/∗ advance the edi by PIXEL_DATA_SIZE ∗/
addl $PIXEL_DATA_SIZE, %edi
addl $PIXEL_DATA_SIZE, %edx

/∗ if the line was only one long, exit ∗/
decl %ecx
jz fastLineMMX2_exit

/∗ set the second values, note for a slight speed increase we save mm0 ∗/
movd (%esi), %mm1
punpckldq (%eax), %mm1
addl $YUV_PIXEL_SIZE, %esi
addl $YUV_PIXEL_SIZE, %eax
movd %mm1, PIXEL_DATA_Y(%edi)
movq %mm1, %mm6
psrlq $32, %mm6
movd %mm6, PIXEL_DATA_Y(%edx)

/∗ mm7 still = divide2_constant ∗/
psrlq $1, %mm1
pand %mm7, %mm1

/∗ save previous yuv/2 in mm6 ∗/
movq %mm1, %mm6

74
/* set Y.ZERO to 0 */
movd %mm3, PIXEL_DATA_Y_ZERO(%edi)
movd %mm3, PIXEL_DATA_Y_ZERO(%edx)

/* the TAIL[N % 3] = PIXEL_YUV[N] - PIXEL_YUV[N - 1] */
psubb %mm0, %mm1
movq %mm1, fastLineMMX2_TAIL0(%ebp)
movd %mm1, PIXEL_DATA_Y(%edi)
movq %mm1, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_Y(%edx)

/**
 * in this case we know that TAIL2 == 0 and I[N - 1] == 0 so I[N] == TAIL[N]
 */
movd %mm1, PIXEL_DATA_IY(%edi)
movq %mm1, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_IY(%edx)

/* dI[N - 1] = 0 -> dI[N] = I[N] */
movd %mm1, PIXEL_DATA_DIY(%edi)
movq %mm1, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_DIY(%edx)

/* advance the edi by PIXEL_DATA_SIZE */
addl $PIXEL_DATA_SIZE, %edi
addl $PIXEL_DATA_SIZE, %edx

/* if the line was only two long, exit */
decl %ecx
jz fastLineMMX2_exit

/**
 * This loop assumes two things
 * 1. The previous YUV_PIXEL is stored in mm6
 * 2. The TAIL is set as follows
 *   TAIL0 = TAIL[N]
 *   TAIL1 = TAIL[N - 1]
 *   TAIL2 = TAIL[N - 2]
 *   ie TAIL0 is the most recent value
 */
fastLineMMX2_loop_begin:

/* load next pixel */
movd (%esi), %mm1
punpckldq (%eax), %mm1

movd %mm4, PIXEL_DATA_Y(%edi)
movq %mm4, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_Y(%edx)

/* divide by 2 */
movq (divide2_constant), %mm7
psrlq $1, %mm4

75
pand %mm7, %mm4
movq %mm4, fastLineMMX2_TMP_YUV(%ebp)

/* load tail2. Its put this here for scheduling purposes*/
movq fastLineMMX2_TAIL2(%ebp), %mm3

/* the TAIL[N] = PIXEL_YUV[N] - PIXEL_YUV[N-1] */
psubb %mm6, %mm4

movd %mm4, PIXEL_DATA_DY(%edi)
movq %mm4, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_DY(%edx)

movd PIXEL_DATA_JV-PIXEL_DATA_SIZE(%edi), %mm2
punpckldq PIXEL_DATA_JV-PIXEL_DATA_SIZE(%edx), %mm2

/* copy to mm1 for use in the filter */
movq %mm2, %mm1

/* calculate new I */
psubb %mm3, %mm2
paddb %mm4, %mm2

movd %mm2, PIXEL_DATA_JV(%edi)
movq %mm2, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_JV(%edx)

/**
* move the TAIL down.
*/
movq fastLineMMX2_TAIL0(%ebp), %mm0
movq fastLineMMX2_TAIL1(%ebp), %mm5
movq %mm4, fastLineMMX2_TAIL0(%ebp)
movq %mm0, fastLineMMX2_TAIL1(%ebp)
movq %mm5, fastLineMMX2_TAIL2(%ebp)

/* calculate dl */
psubb %mm1, %mm2

movd %mm2, PIXEL_DATA_DY(%edi)
movq %mm2, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_DY(%edx)

/* load the other two dl's */
movd PIXEL_DATA_DY-2*PIXEL_DATA_SIZE(%edi), %mm0
punpckldq PIXEL_DATA_DY-2*PIXEL_DATA_SIZE(%edx), %mm0

movd PIXEL_DATA_DY-1*PIXEL_DATA_SIZE(%edi), %mm1
punpckldq PIXEL_DATA_DY-1*PIXEL_DATA_SIZE(%edx), %mm1

/**
* test for dl filter with mmA = mm0, mmB = mm1 mmC = mm2
*/
pcmpeqd %mm6, %mm6
pxor %mm7, %mm7
```c
/* check for pattern - + -
 * if (!(mmA > -1)) & (mmC > -1)) == if (mmA < 0 & mmC < 0)
 */
    movq  %mm0, %mm3
    pcmptgtb %mm6, %mm3
    pxor  %mm6, %mm3
    movq  %mm2, %mm4
    pcmptgtb %mm6, %mm4
    pxor  %mm6, %mm4
    pand  %mm4, %mm3

/* if (mmB > -1) == (mmB < 0) */
    movq  %mm1, %mm4
    pcmptgtb %mm6, %mm4
    pand  %mm4, %mm3

/* check for pattern + - +
 * if (mmA > 0 & mmC > 0)
 */
    movq  %mm0, %mm4
    pcmptgtb %mm7, %mm4
    movq  %mm2, %mm5
    pcmptgtb %mm7, %mm5
    pand  %mm5, %mm4

/* if (mmB > 0) */
    movq  %mm1, %mm5
    pcmptgtb %mm7, %mm5
    pxor  %mm6, %mm5
    pand  %mm5, %mm4

/*** finally or the two together
 * por %mm4, %mm3
 */

/*** filter requires
 * ed points to current PIXEL_DATA
 * mm3 = 1111... if filter is required, else 0000...
 * mm6 = 1111...
 * the end result is
 * if (filter is needed)
 * mm1 = (mm0 + mm2) / 2;
 * mm2 = undefined
 * else
 * mm1 = mm1
 * mm2 = undefined
 */

/* calculate filtered result
 * pavgb %mm0, %mm2
 */

/*** Now to calculate mm1 we do
 */
/*** mm1 & !condition + average & condition */
```
```assembly
pxor %mm3, %mm6       /* mm6 != condition */
pand %mm6, %mm1
pand %mm3, %mm2
paddb %mm2, %mm1

/**
* End Filter
* /

/** write the result from the filter */
movd %mm1, PIXEL_DATA_DIY−PIXEL_DATA_SIZE(%edi)
movq %mm1, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_DIY−PIXEL_DATA_SIZE(%edx)

/**
* Detect zeros with mmA: dI[n − 2] = mm0, mmB: dI[n − 1] = mm1
**
* Preload a couple values
*/
pcmpeqb %mm2, %mm2
pxor %mm3, %mm3

/** calculate abs (mmA) and abs (mmB) */
movq %mm0, %mm4
pcmpeqt %mm3, %mm4
movq %mm1, %mm5
pcmpeqt %mm3, %mm5

/** calculate sign (a) != sign (b) and store */
movq %mm4, %mm6
pcmpeqt %mm5, %mm6
pxor %mm2, %mm6       /* invert */
movq %mm6, %mm7

/** continuing with abs */
pxor %mm2, %mm4
pxor %mm4, %mm0
psubb %mm4, %mm0
pxor %mm2, %mm5
pxor %mm5, %mm1
psubb %mm5, %mm1

/** if abs (a) > abs (b) then the 0 occurs at b. Else the zero occurs at a */
movq %mm0, %mm4
pcmpeqt %mm1, %mm4

/** and result with sign (a) != sign (b) */
pand %mm4, %mm7
pxor %mm2, %mm4
pand %mm4, %mm6

/** calculate a == 0, b == 0 */
pcmpeqb %mm3, %mm0
pcmpeqb %mm3, %mm1
```

78
/* or with previous result */

por %mm6, %mm0
por %mm7, %mm1

/* determine whether or not abs(I) values are above thresholds */
/* by calculating -threshold > I || I < threshold */
movq (YUV_neg_threshold_constant), %mm4

movd %mm4, PIXEL_DATA_Y-2*PIXEL_DATA_SIZE(%edi), %mm5
punpckldq %mm5, PIXEL_DATA_Y-2*PIXEL_DATA_SIZE(%edx), %mm5

/** change this in MMX1 */

pcmpgtb %mm5, %mm4
pcmpgtb (YUV_threshold_constant), %mm5

por %mm5, %mm4
pand %mm4, %mm0

/* repeat for I[n-1] */
movq (YUV_neg_threshold_constant), %mm4

movd %mm4, PIXEL_DATA_Y-1*PIXEL_DATA_SIZE(%edi), %mm5
punpckldq %mm5, PIXEL_DATA_Y-1*PIXEL_DATA_SIZE(%edx), %mm5

pcmpgtb %mm5, %mm4
pcmpgtb (YUV_threshold_constant), %mm5

por %mm5, %mm4
pand %mm4, %mm1

/* or with zeros [n-1] so we dont overwrite it */
movd %mm5, PIXEL_DATA_Y_ZERO-2*PIXEL_DATA_SIZE(%edi), %mm4
punpckldq %mm4, PIXEL_DATA_Y_ZERO-2*PIXEL_DATA_SIZE(%edi), %mm4

por %mm4, %mm0

/* write zeros */
movd %mm0, PIXEL_DATA_Y_ZERO-2*PIXEL_DATA_SIZE(%edi)
movq %mm0, %mm7
psrlq $32, %mm7
movd %mm7, PIXEL_DATA_Y_ZERO-2*PIXEL_DATA_SIZE(%edx)
movd %mm7, PIXEL_DATA_Y_ZERO-1*PIXEL_DATA_SIZE(%edi)
movq %mm1, %mm7
psrlq $32, %mm7
movd %mm1, PIXEL_DATA_Y_ZERO-1*PIXEL_DATA_SIZE(%edi)

psrlq %mm7, PIXEL_DATA_Y_ZERO-1*PIXEL_DATA_SIZE(%edx)

/**
* End zero detection */

/* restore current YUV values to %mm6 */
movq fastLineMMX2_TMP_YUV(%ebp), %mm6

addl $YUV_PIXEL_SIZE, %esi
addl $YUV_PIXEL_SIZE, %eax
addl $PIXEL_DATA_SIZE, %edi
addl $PIXEL_DATA_SIZE, %edx

dcl %ecx
jnz fastLineMMX2_loop_begin
/**
 * Detect final zeros with mmA: dl[n-1] = mm0, mmB: dl[n-0] = mm1
 */
movd    PIXEL_DATA_DIY-2+PIXEL_DATA_SIZE(%edi) , %mm0
punpckldq    PIXEL_DATA_DIY-2+PIXEL_DATA_SIZE(%edx) , %mm0

movd    PIXEL_DATA_DIY-1+PIXEL_DATA_SIZE(%edi) , %mm1
punpckldq    PIXEL_DATA_DIY-1+PIXEL_DATA_SIZE(%edx) , %mm1

/*************************************************************************/
/* Preload a couple values */
pcmpeqb %mm2, %mm2
pxor   %mm3, %mm3

/*************************************************************************/
/* calculate abs (mmA) and abs (mmB) */
movq   %mm0, %mm1
pcmpgtb %mm3, %mm1
movq   %mm1, %mm5
pcmpgtb %mm3, %mm5

/*************************************************************************/
/* calculate sign (a) != sign (b) and store */
movq   %mm4, %mm6
pcmpeq %mm5, %mm6
pxor  %mm2, %mm6  # invert
movq   %mm6, %mm7

/*************************************************************************/
/* continuing with abs */
pxor   %mm2, %mm4
pxor   %mm4, %mm0
psubb  %mm4, %mm0
pxor  %mm5, %mm6
pxor  %mm6, %mm1
psubb  %mm6, %mm1

/*************************************************************************/
/* if abs (a) > abs (b) then the 0 occurs at b. Else the zero occurs at a */
movq   %mm0, %mm4
pcmpgtb %mm1, %mm4

/*************************************************************************/
/* and result with sign (a) != sign (b) */
pand  %mm4, %mm7
pxor   %mm2, %mm4
pand  %mm4, %mm6

/*************************************************************************/
/* calculate a == 0, b == 0 */
pcmpeq %mm3, %mm0
pcmpeq %mm3, %mm1

/*************************************************************************/
/* or with previous result */
por   %mm6, %mm0
por   %mm7, %mm1

/*************************************************************************/
/* determine whether or not abs (I) values are above thresholds */
/* by calculating -threshold > I || I < threshold */
movq   (YUV_neg_threshold_constant) , %mm4

movd    PIXEL_DATA_IY-2+PIXEL_DATA_SIZE(%edi) , %mm5
punpckldq    PIXEL_DATA_IY-2+PIXEL_DATA_SIZE(%edx) , %mm5

80
/** change this in MMX */

pcmpgtb %mm5, %mm4

pcmpgtb (YUV_threshold_constant), %mm5

por  %mm5, %mm4

pand  %mm4, %mm0

/ * repeat for I[n − 1] */

movq  (YUV_neg_threshold_constant), %mm4

movd  PIXEL_DATA_IY−1+PIXEL_DATA_SIZE(%edi), %mm5

punpckldq  PIXEL_DATA_IY−1+PIXEL_DATA_SIZE(%edx), %mm5

pcmpgtb  %mm5, %mm4

pcmpgtb (YUV_threshold_constant), %mm5

por  %mm5, %mm4

pand  %mm4, %mm1

/ * or with zeros[n − 1] so we don’t overwrite it */

movd  PIXEL_DATA_YZERO−2+PIXEL_DATA_SIZE(%edi), %mm4

punpckldq  PIXEL_DATA_YZERO−2+PIXEL_DATA_SIZE(%edi), %mm4

por  %mm4, %mm0

/ * write zeros */

movd  %mm0, PIXEL_DATA_YZERO−2+PIXEL_DATA_SIZE(%edi)

movd  %mm0, %mm7

psrlq  $32, %mm7

movd  %mm7, PIXEL_DATA_YZERO−2+PIXEL_DATA_SIZE(%edx)

movd  %mm1, PIXEL_DATA_YZERO−1+PIXEL_DATA_SIZE(%edi)

movq  %mm1, %mm7

psrlq  $32, %mm7

movd  %mm7, PIXEL_DATA_YZERO−1+PIXEL_DATA_SIZE(%edx)

/ **

* End final zero detection */

fastLineMMX2_exit:

/ * free the MMX registers */

emms

fastLineMMX2_quick_exit:

pop  %edi

pop  %esi

movl  %ebp, %esp

popl  %ebp

ret

81