Michael Damert and Don Lavelle

Advisor: Prof. Eugenio Schuster
**Project Overview:**

The previous work done on this project included a video capture system and a particle filter implementation. The video capture system was based on Carnegie-Mellon University camera driver using Unibrain Fire-i IEEE-1394 cameras. The code provided an initialization function for the camera, and was also capable of capturing images from the camera and holding them in a variety of formats with minor modification. The capture formats used for this project were YUV 4:2:2 and RGB. The initialization function was modified and used in the current vision system to capture at 320x240 resolution at 15 fps.

The particle filter, which will only be discussed briefly, was based on tracking template images loaded from image bitmap files. Essentially, the particle filter would randomly search a given number of locations, randomly distributed, for an image match on one of the templates. This algorithm was timed to run at 5 frames per second. Additionally, there were bugs and other errors in the functionality of the particle filter code. It was the decision of the incoming group to change to a method that uses color segmentation and blob location instead of a particle filter; the group then met with better results.

A basic architecture was then developed in order to give organization to the various project components. (One problem with the previous work was that it was disorganized; a piecemeal approach had been used instead of a structured approach.) The architecture designed follows the modular programming paradigm. (See Figure 1.)
Going in order of the program flow, the camera input is taken from the CMU 1394 driver and is stored in BGR format. It is then converted to RGB format. The system then passes the image to the image processing algorithm. This consists of two components, a blob locator and a pattern recognizer. The blob locator locates patches based on the segmentation done by the OpenCV library. The pattern locator uses the blob locator to determine the positions of the patches and then the positions and orientations of the patterns.
The OpenCV library is used primarily for its color segmentation software and its data storage containers. The vision system, given an image to process, begins by segmenting the image over the team patch colors. In our testing, yellow was used as the team patch color. The segmented image consists of a white pixel in each location that corresponds to a pixel in the original frame that falls within a given color range.

![Figure 2](image)

**Figure 2. Camera input using a single pattern before and after segmentation.**

Once the pattern is segmented, the blob detection algorithm is run. The algorithm implemented is based roughly on the CMVision library. The algorithm works by grouping rows of pixels, sorting the rows for connectedness, and finding the centroid of the runs. **Figure 3** (shown below) represents a section of a segmented image. The pink (positive) pixels represent an in-range pixel in the original image; white pixels represent a pixel out of range.
When the blob detection algorithm executes, it iterates over every row in the segmented image. If there are no positive pixels in the row, it continues to the next row. Once a positive pixel is detected, the location of and length of the run of positive pixels is stored. (Thresholding is possible; in fact, the algorithm used ignores runs that are two pixels or shorter.)

Once all of the runs are stored, the algorithm checks to see for a given run if any runs overlap horizontally in adjacent rows. Runs (that are of sufficient length) that are adjacent are processed in a depth-first traversal, summing the coordinates of each pixel and then dividing by the number of pixels in the run to calculate the centroid of the blob. This is the point that is used as the location of the blob.
This blob detection algorithm was written by hand using the OpenCV library as a basis.

The pattern location algorithm directly uses the blob detection algorithm. It begins by running the blob location algorithm to locate the team patches. It then takes a region of interest around each team patch and processes that region. The pattern detection component then takes the four nearest neighbor patches and treats those as the patch pattern for the instant robot. The algorithm designed then takes the two farthest patches and the two nearest patches.
compression algorithm used for simulated input for testing, which created a checkerboard pattern. Simulated input is used because of physical constraints in the lab; background noise, e.g. person in the lab, desks, furniture, etc, interfere the segmentation algorithm. The patch patterns used in sample input were constructed of thin styrofoam wrapped in black cotton cloth; the patches were attached using velcro to the black base.

Currently, the lab is outfitted with a single Fire-i camera mounted approximately six feet above a green outdoor carpet. The carpet is 12"x9". The F18 league regulations require camera to be mounted on a mounting bar placed four meters above the ground with a field 4900 mm by 3400 mm. The smaller field was chosen due to the dimensional limitations of the lab; the camera placement modification was due to the nature of the ceilings in the lab and the size of the room. By selecting a camera height that was approximately one half of regulation, we were able to process one quarter of the expected frame size (320x240 as opposed to 640x480) to simulate the number of pixels per color patch.

Since the robots were not in working, playable condition over the course of the semester and one of our goals was to implement an AI system for the robots, it was decided that the best way to create an AI system for this project would be to use a program to simulate the conditions of the robots playing soccer.

Several RoboCup papers¹ were found that discussed a RoboCup soccer simulator program which was freely available online. This simulator’s webpage is http://sserver.sourceforge.net/. The

¹ Papers were found in the source: RoboCup-97: Robot Soccer World Cup I. Hiroaki Kitano (Editor). 1998. Springer (publisher). New York.
The simulator itself was written on and for a Mac computer, and although the software claimed that it was cross-platform, the files offered very little documentation on how to port the code to a Windows computer, and the files of this software were not created to be easily compatible with a Windows-based software compiler. The group working on this project spent roughly an hour attempting to get the code to compile and run on a Windows system but was unable to and the software created many, many errors. Given the time constraints for this project (by then there was little more than a month before the project was due to be finished), it was decided that it would be safest to design an original simulator program for the testing of the AI system.

The AI system is designed using the SDL 2D graphics library. This library was chosen for its extreme ease-of-use and because the group working on the project is familiar with the library. The decision to use SDL was a fruitful one: the simulator’s graphics were finished in only about a week with the help of an online, free library\(^2\) and a tutorial\(^3\).

\[\text{Figure 7. RoboCup simulator program.}\]

The approach used was that all AI is restricted to the files “AI.h” and “AI.cpp.” AI is modularized as much as possible to ensure that when the robots’ AI subsystem is interfaced with the actual real-world field and camera, as little work as possible will be necessary to interface the AI with the other robot subsystems. In order to allow for real-world

\(^3\) “Graphics with SDL” tutorial series, Lesson 2. Marius Andra (http://cone3d.gamedev.net/).
image processing, the AI is calculated on a frame-by-frame basis. Although this method is limited by the fact that robots can only easily make linear motions, calculating images on a frame-by-frame basis will make it much easier to transition to a real-world camera-based system – in the real-world system, robots are imperfect; they may collide with other robots; the ball may bounce in unpredictable ways; the signals sent to robots may take more time than the simulated 0.0 seconds. In addition, frame-by-frame processing was a valuable choice given that the time frame for this project was relatively short, and frame-by-frame processing is comparatively easier to implement than internal processing.

The goal set out for work this semester was to implement a basic AI system which would offer a starting point for future work on the RoboCup project. The AI system was built based on the papers “The Spirit of Bolivia: Complex Behavior Through Minimal Control,” by Barry Brian Werger and “Andhill-98: A RoboCup Team which Reinforces Positioning with Observation,” by Tomohito Andou.

The simulation program was designed such that it can be easily changed to accommodate real-world values. The program was designed in C++ on a Windows computer. The SDL library is cross-platform, and the only Windows-specific function call the program makes is `GetTickCount()`, used to get the current time in milliseconds, which is defined in the header file “windows.h,” however since the Windows header files are only included when the program is running on a Windows computer, the `GetTickCount()` could easily be re-defined for another operating system.
The numbers used by the simulator are handled in the file “constants.h.” The design decision to make only one central set of constants was made in order to most easily facilitate the porting of the code from simulator to real-world applications. Therefore, when the AI is put into a real-world situation, the only data that need to be changed are the values in “constants.h” of field width, field height, ball radius, etc.

The Field Coordinate system requires that the coordinate (0, 0) refer to the upper, left-hand corner of the field. X is positive in the right-hand direction; Y is positive in the downward direction. Coordinates are measured with floating-point (decimal) numbers so that positions are not fixed to a grid.

Robots are drawn (arbitrarily) as blue cylinders (circles) with a red line segment pointing in the direction of their kicker mechanism. The ball is drawn as a small, orange sphere (circle). The markings on the field are minimal because the field markings have no effect on our simulation. The field is drawn to scale as specified in the RoboCup F18 league regulations. The robots are drawn as being the maximum size the rules allow.

The simulated field is organized such that the goal for our team (alias “My Goal” or “MG”) is on the left of the screen and our opponent’s goal (alias “Opponent’s Goal” or “OG”) is on the right of the screen. Robots’ positions are defined in field coordinates. Our particular implementation uses a field size of 800x600. This size was chosen arbitrarily since it will fit easily on most computer screens. Robot orientation is the direction that the robot’s kicker is currently facing. Orientation is described in radians, where 0 radians describes the condition when the robot is
directly facing OG and (π/2) radians describes when the robot’s kicker is facing directly up (see Figure 8).

Figure 8. Field coordinates, robot orientation, and MG and OG.

Robots do not use their dribbler mechanism because no use of the dribbler was explained in the papers this AI system is modeled after. Also, the kicker mechanism the simulator uses is only a guess as to how hard the robot hits the ball and how much friction the ball encounters while in motion. These values were not tested because the real-world kicker mechanism had not been finalized at the time the simulator was made. The real-world kicker may (or may not) also be able to kick the ball at variable speeds, though the kicker in the simulator only kicks the ball at one speed. Given the uncertainty about of how the kicker mechanism will work, it was decided to allow future groups to choose how best to simulate or implement the kicker.
The simulator’s physics are implemented ideally, meaning all collisions are perfectly elastic and the ball will make no unexpected turns, will not jump in the air, will not go off of the field. This does not affect our team’s AI. Several shortcuts were taken in implementing the physics system, for example, robots move according to their velocity alone rather than a combination of their velocity and acceleration. Again, the AI is affected very, very little by this decision.

Robot deceleration is assumed to be instantaneous. In practice, there is no way of knowing how quickly robots decelerate, because the robots were not available for testing while the simulator was being written.

The simulator was timed as running at between 35 and 40 frames per second on a Pentium 4, 2.0 GHz computer. The simulator’s running speed is primarily limited by the graphics processing and the complexity of the physics simulation.

At the beginning of the game, the robots are given both starting positions and a default position. The default position is a defensive position which the robot will return to when the ball is too far away from the robot for the robot to reasonably chase after the ball.

Robots’ AI plans are selected based on two rubrics:

- The robots’ position from the ball
- Whether the ball is (a) between the robot and MG; or (b) between the robot and OG
Figure 9. The blue circle represents distance NEAR, the yellow circle is distance MID, and all other space is distance FAR.

The overall plan for the AI, as described in Werger, is to attempt to kick the ball into the opponent’s goal and to always be in a position horizontally between MG and the ball. This particular implementation of Werger’s paper uses a three-distance system as pictured in Figure 9. The distances for this system are chosen arbitrarily but seem to work well in practice. NEAR has a radius of roughly four times the radius of the bot; MID has a radius of about the width of half the field.

The particular implementation of the AI used calls for the following: when the bot’s distance from the ball is NEAR, the bot will go to the position behind the ball, properly align itself so that there is a line from OG through the ball to the bot, and then kick the ball. When the bot’s distance from the ball is MID, the bot’s plan is very similar: to get into a position where it can kick the ball into OG. When the bot’s distance from the bot is FAR, the bot will take a defensive
stance by moving into its default position. Default positions are defined at the beginning of the game but are dynamically changed throughout the game, as described in Andou.

Andou’s paper outlined a system of choosing positions based on observational learning. The simulator does the same by recording the ball’s position on every frame the ball has moved and recording a log of the number of times the bot has appeared in each position. If a position is recorded a large number of times in the log and that position is not offsides, that position will become the new default position of one of the robots.

Robot motion is handled by calling a function `MoveToward()`, which is a very general function that will easily be ported to the real-world application. In the simulator, this function calls for a bot to move linearly toward a point.

Time is only handled in the simulator – no time is managed in the actual AI files, which means that the AI will run completely independent of any framerate at which the camera can capture images.

The results for the AI are that a considerable amount of the time spent working on the simulation program was spent creating a believable physics system for the simulator program. This is an unfortunate side-effect of the decision to create an original simulator program. However, the tradeoff was that no new code had to be learned and compiling the code was quick and easy, which made for extremely quick visual results that the simulator program was being successfully implemented.
One of the particular issues which the AI runs into is that since the AI is implemented entirely
independent of any framerate or position system, the bots do not have any way of knowing when
to slow down.

The bots’ AI could easily be extended to break the team into offensive and defensive positions.
The framework for this distinction has been created – bots have a variable which describes one
item in the enumeration BotType. At present, BotType only refers to whether the bot is on
My Team or the Opponent’s Team, but this could easily be extended to break bots into goalie,
offensive, and defensive positions, and AI could be described in terms of these cases.

Of the different modules outlined in the architecture, only the steering model was not touched
upon. This was due to the fact that there were no functioning robots to test with.

Finally, a serial output module was created. This serial port module can be broken down into
two classes. The inner class, SerialPort, abstracts the serial port. This code is platform
specific and was designed to run on Microsoft Windows XP. The code was based on sample
code from the MSDN reference. The outer class, RobotMotion, provides an interface which
allows the steering model to be interfaced with the serial port. Specifically, RobotMotion takes
as input wheel speeds and converts them to a bit pattern, which is then transmitted via a wireless
serial port bridge to the robots. This bit pattern was specified by the electrical engineering
group.