Smart Lot

Landon Anderton, Alex Freshman,
Kameron Sheffield and Sunny Trinh

smartlot@googlegroups.com
www.kamsheffield.com/school/smartlot

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

SmartLot is the result of an obsession to get the closest parking space. Have you ever settled for a parking space in the back, only to walk by a closer one on your way in? Or have you ever risked it all for the sweet spot up front, only to head back in shame, when you found all the spots were taken? SmartLot is the solution to this everyday annoyance. Upon entering a parking-lot, a driver is directed to the closest available spot by following traffic signals adjacent to each parking row. If the traffic signal shines red, then there are no available spots on that row. If the light shines green, then there is at least one available spot, and the driver can safely venture down the row without the risk of shame.

The obvious problem at hand is how to detect empty parking spots from the occupied. An easy and reliable solution would be to install a sensor in each parking space, whether it is an induction sensor, distance sensor, laser, or anything else that can sense a ton of steel. The problem with installing sensors in every parking space is that it would be impractical in most situations. The cost to install such a system in an already existing parking-lot would be too great, not to mention the cost of each individual sensor. The SmartLot solution is to use a camera with the ability to monitor the occupancy of many parking spaces. Image processing is then used to determine the occupancy of the parking spaces. This solution is ideal, because it is easily installed into existing parking-lots and requires less hardware at a cheaper cost.

In an ideal situation, the SmartLot system would incorporate many low definition cameras attached to light poles that monitor a subset of the overall parking-lot. The system would take into account all of the different camera angles to determine the occupancy of a given row or area. Also in an ideal situation, drivers could be updated on the closest spot through any number of interfaces, whether it be though traffics lights as described above, text message, or a screen that shows a list of exact available spaces upon entry to the lot.

For the purposes of this project, a prototype SmartLot is being developed to demonstrate the functionality of the image processing algorithms and one possible interface to drivers – in this case the traffic lights at the edges of rows. For the prototype, instead of multiple cameras attached to light poles, one high-definition camera is placed high above the parking-lot and is used to monitor as many parking spaces as possible.

2. SYSTEM OVERVIEW

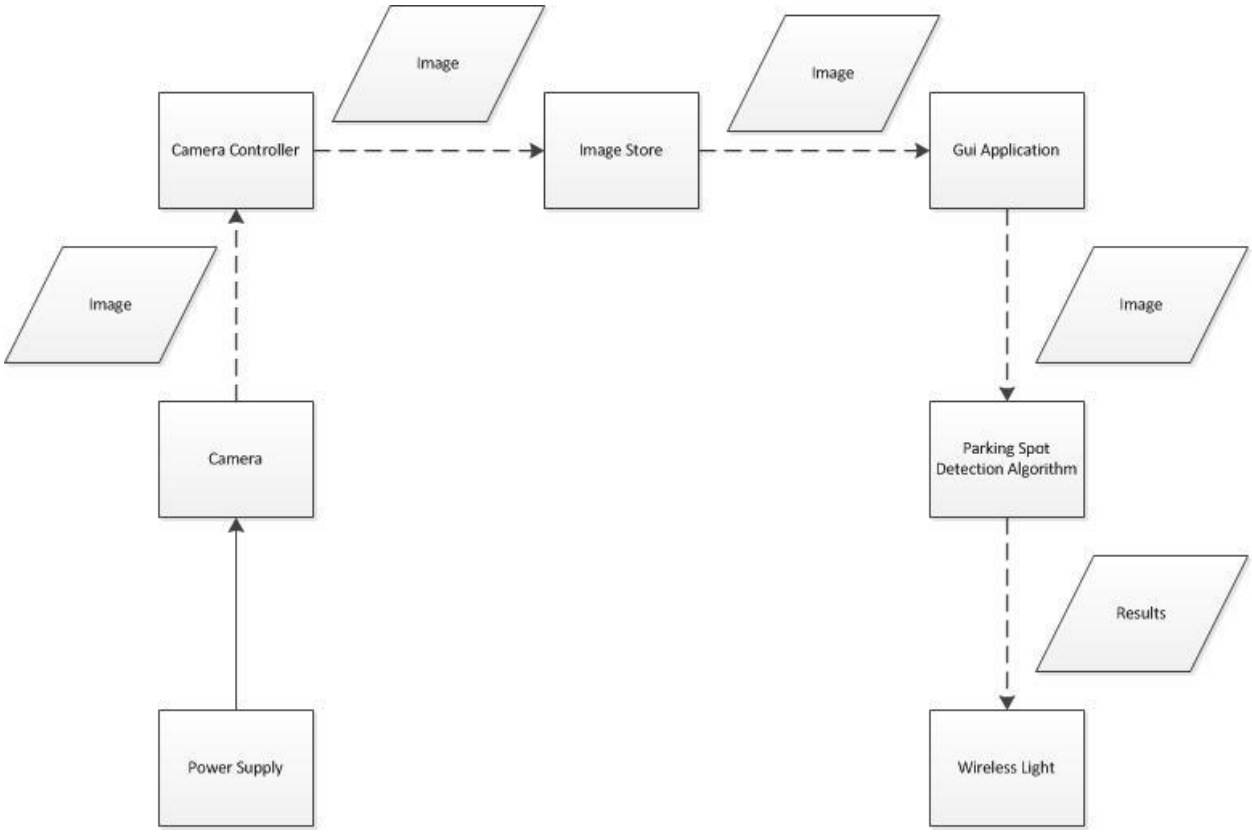

Figure 1 - SmartLot Flow Chart

SmartLot can be divided into three basic parts: the camera and camera controller, the software application and the wireless-light. The purpose of the camera controller is to tell the digital camera when to take a picture. Once the picture has been acquired from the camera it needs to be sent somewhere so that it can be processed by the software application. The software application is responsible for acquiring the images from the image store, running the parking spot detection algorithm on those images, and sending the results to the wireless-light so that the appropriate lights can be signaled. The software application represents the bulk of the work in SmartLot; it consists of two main parts, a calibration tab and a demo tab. The calibration tab is used to provide the parking spot detection algorithm with all of the information it needs in order to process the image. It also provides a laboratory for developing the parking spot detection algorithm and testing it. The demo tab is used to actually run SmartLot, once started; the application will download images, process them, display the results and send the results to the wireless-light.

For this project a Nikon Coolpix digital camera, powered by a custom power supply and connected to a PC, is stationed on the 13th floor of the behavioral science building at the University of Utah. The head of behavioral sciences department was nice enough to loan us an office that looked out on the adjacent parking-lot. An image of the subject parking-lot is shown in Figure 2. The PC connected to the camera uploads the parking-lot images to a server where they can be downloaded and processed on the other end.

Figure 2 - Subject Parking-lot

3. DESIGN

Subsequent headings illustrate the design of each aspect of the SmartLot flow (shown in Figure 1). A detailed design of each item in the flow chart is described below.

3.1. DESIGN OF CAMERA POWER SUPPLY

The most important criterion used to select a camera for the project was the requirement that the camera had to work with the gPhoto2 Linux library; the Nikon Coolpix was selected for this reason. The camera needs to run continuously, which means the battery can't be relied upon for power. There are very few cameras out there that can be powered through the USB cable or AC adapter, and the Nikon Coolpix was no exception. This required us to use our ingenuity to come up with a work around for power.

The battery provided with the camera provides between 3.7 V to 4.2 V at a guaranteed 0.4 A. We replaced this battery with the power supply shown in Figure 3.

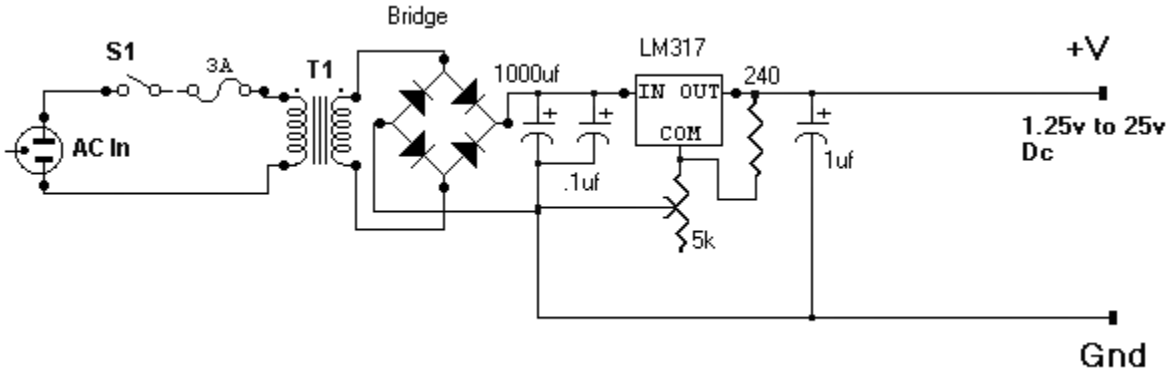

Figure 3 - Power Supply Schematic (1)

The circuit shown takes a 120V AC waveform and turns it into a stable DC supply ranging from 1.25V to 25V. The output voltage is controlled by a 5 kΩ potentiometer. A variable voltage supply was chosen so the output could easily be calibrated to match that of the battery. The circuit works in a series of steps

that slowly turn the sinusoidal waveform into a flat DC output. The first step is the transformer, labeled T1. The transformer lowers the 120V AC input to around 40V AC with minimal loss of power. The second step is the bridge, which changes the AC signal so that the entire waveform is positive. This is shown in Figure 4.

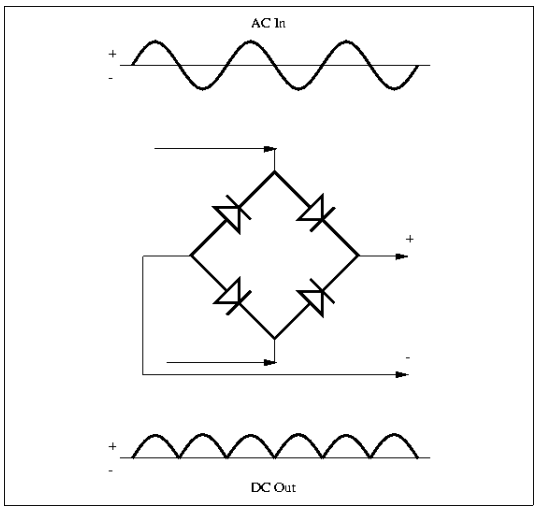

Figure 4 - Waveform after Bridge (2)

The waveform on top of the figure is the input, and the output on bottom shows the positive version. This is accomplished with diodes that only allow the current to flow in one direction. The transformer we use is a center-tapped transformer, so the bridge was constructed with only two diodes. This is possible because the two waveforms the transformer provides have a 90° phase shift. The third step is performed by capacitors after the bridge. These capacitors charge and discharge slowly which causes the changes in the signal to happen more slowly as well. Essentially they take the bottom signal shown in Figure 4 and turn it into the dark line shown in Figure 5.

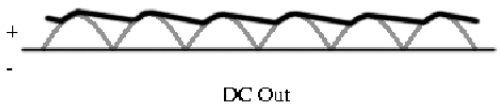

Figure 5 - Waveform after Capacitors (2)

The final step is performed by the LM317 integrated circuit. This IC takes the signal, stabilizes it, and turns it into an output voltage dependent on the resistors on its output and com ports. The potentiometer was set by connecting the output to a volt meter. The potentiometer was then adjusted until a value of 4.1 V was achieved. The output was also hooked to a 10Ω resistor to verify that the maximum current draw of 0.4 A could be achieved. Finally the power supply was allowed to run for several days to make sure no over-heating would occur. We still needed a way to hook the power supply terminals to where the battery connects to the camera. A mock battery was fabricated on a Sherline mill out of Matt Wax carving wax. The center of the wax was hollowed out so wires could easily be run from one end to the other. Contacts were cut from a brass rod and counter sunk into the wax. The wires were soldered to the contacts and allowed to stick out from the bottom of the mock battery. Finally these wires were soldered to the power supply. A picture of the mock battery shown next to the original can be seen in Figure 6.

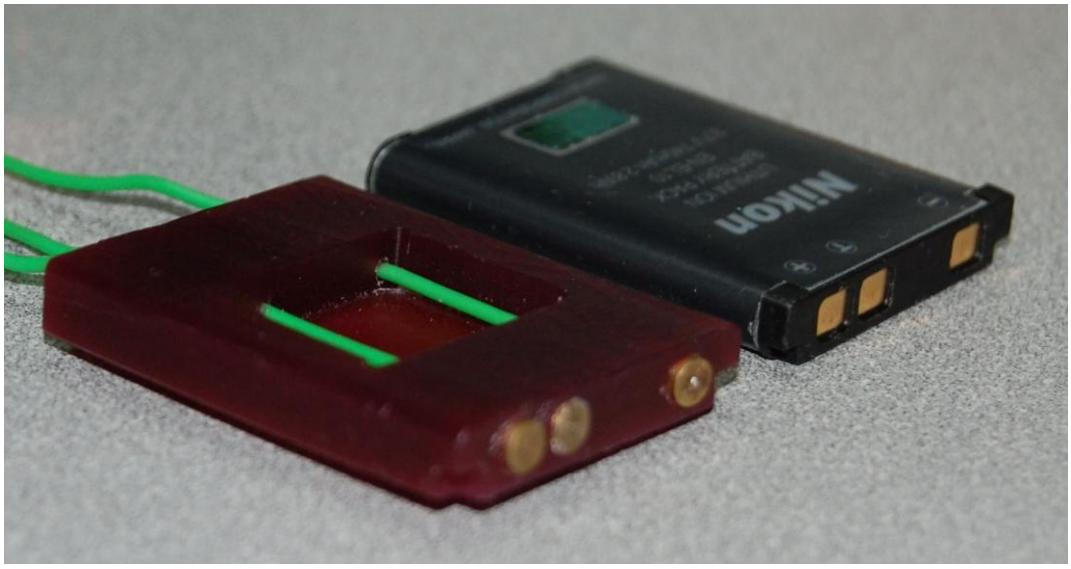

Figure 6 - Mock Camera Battery

3.2. DESIGN OF CAMERA CONTROLLER

The system is designed to utilize one high resolution camera to capture an image of the entire parking-lot from one location, on a regular interval. In order to achieve this in a timely manner, without having to do a large amount of hardware design an off the shelf digital camera is used with a desktop computer to control it. The computer runs a small C++ program which utilizes the gPhoto library to control the camera. In addition to these features the software also uses the cURL library to upload the images to the image store via FTP.

The design of the software is straightforward. First, because both cURL and gPhoto are C libraries, small C++ wrapper classes are used to make the needed functionality more easily used. The core functionality of the application is contained within a single loop. The loop begins after accepting arguments that control where to upload the image files and how often to take them. The application then takes a picture, uploads it, and then sleeps the designated amount of time before starting over again. A link to the complete code is found in 7.APPENDIX B:.

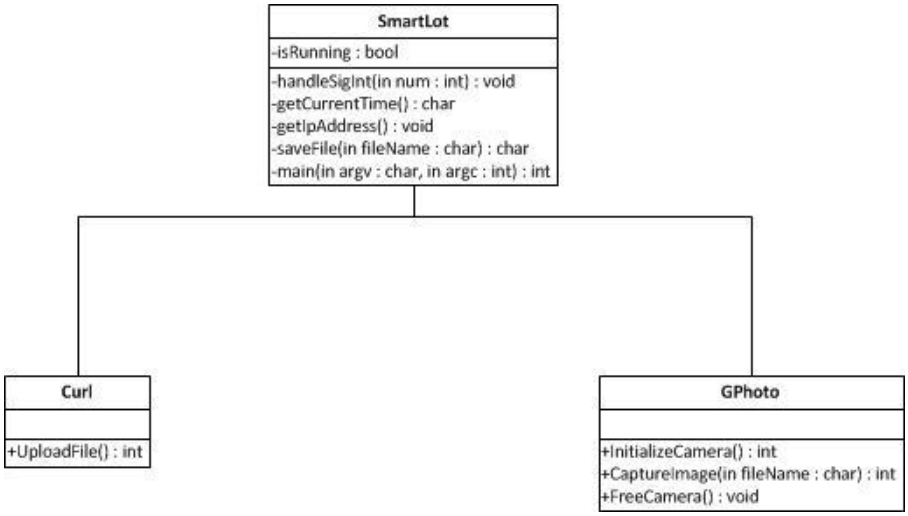

Figure 7 - Camera Controller Class Diagram

3.3. SMARTLOT APPLICATION DESIGN

The SmartLot application has three purposes. The first is to calibrate the parking-lot, or in other words lay down the patch lines. Each parking space needs one line to show where a parking space lies and a second line to compare with. The second purpose of the application is for testing different occupancy-detection algorithms. Once the parking-lot is laid out, different images can be opened and different algorithms can be performed on them. The third purpose of the application is to demonstrate a live parking-lot. In this mode the latest images are automatically opened and the results automatically update the wireless-lights. A screen shot of the calibration tab is shown in Figure 8. A screen shot of the live application tab is shown in Figure 9.

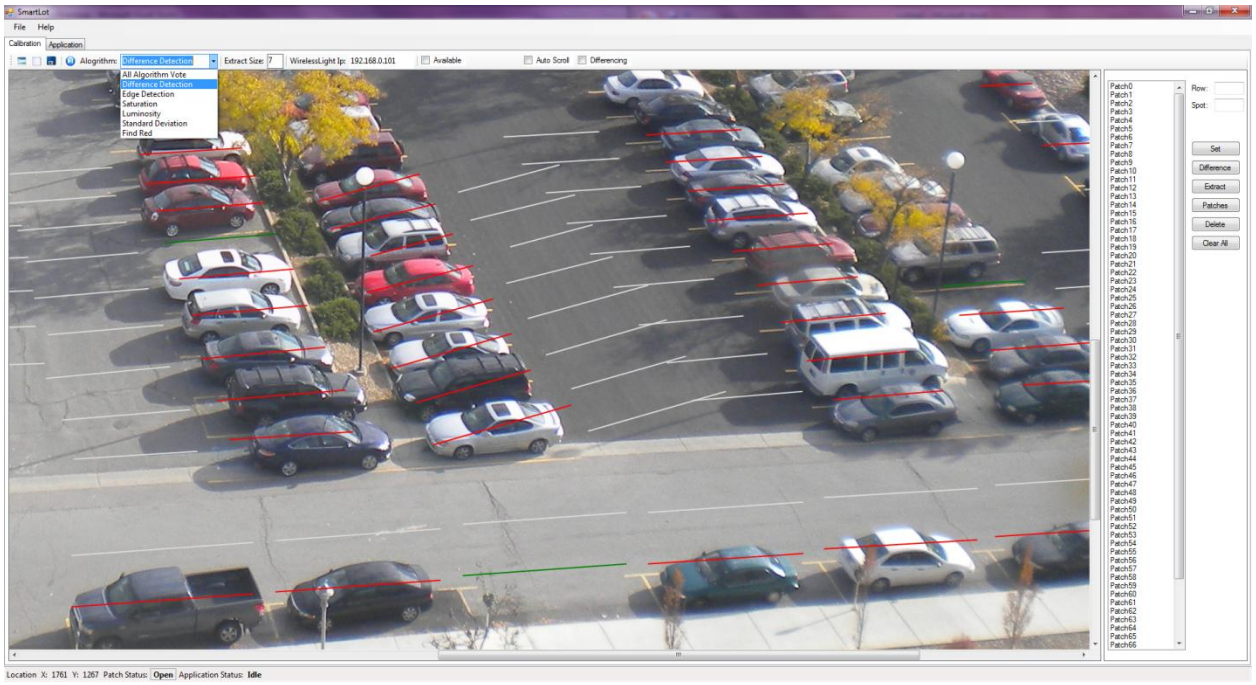

Figure 8- SmartLot Application Calibration Tab

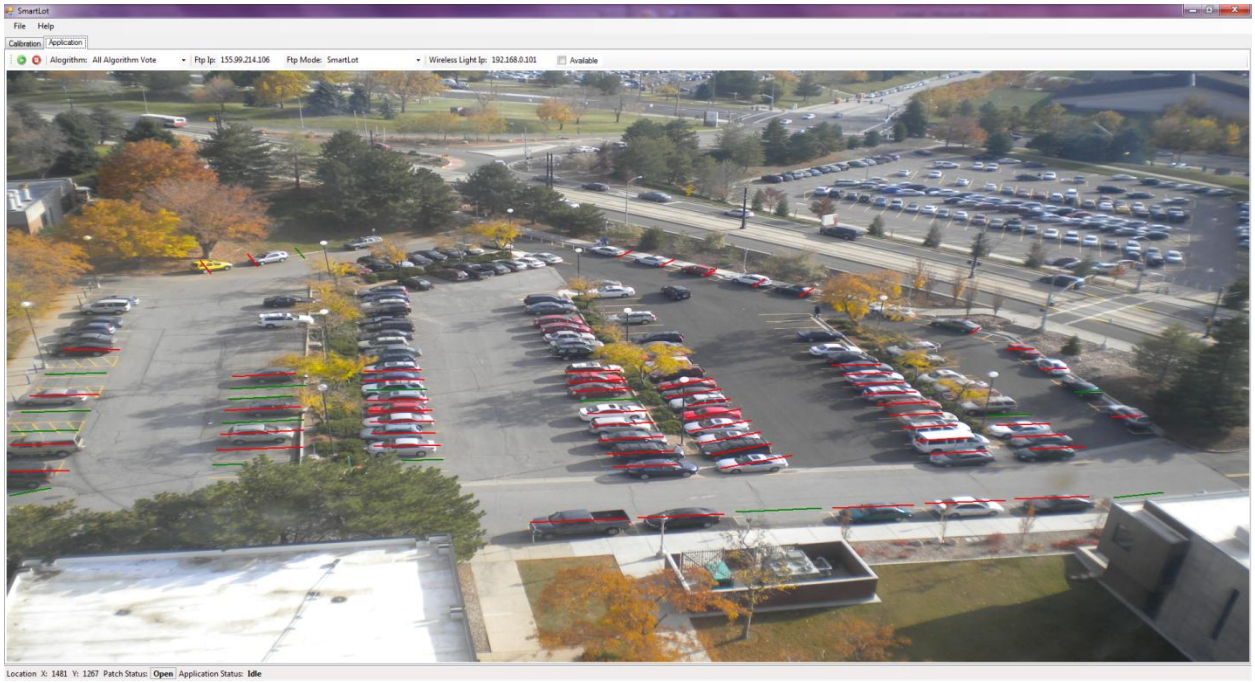

Figure 9 - SmartLot Application Tab

A brief description of each class that makes up the SmartLot Application is given in Table 1. A link to the complete code is given in 7.APPENDIX A.1.

Table 1- SmartLot Application Class Descriptions

Class Name	Purpose
About.cs	This is an about pop-up window that provides our contact information.
DifferencePatch.cs	Inherits from Patch (see Patch.cs). This is a duplicate patch to the one marking the parking space. This patch is shifted to a location that marks blank asphalt. The patch covering the parking spot and the difference patch are compared to determine if the spot is occupied.
ExtractedImageForm.cs	This is a pop-up window that shows a selected line patch extracted from the image.
Form1.cs	This is the main window of the application. It contains a file menu and two main tabs – Calibration and Application. The calibration tab provides all of the tools to calibrate a parking-lot map. The application tab allows for continuous running of a live parking-lot.
FtpClient.cs	This class handles the downloading of the image from the server. It checks regularly for a new image. When a new image is uploaded to the server from the camera, it downloads the latest and passes it to the RunRoutine.
ParkingSpotPatch.cs	Inherits from Patch (see Patch.cs). This is a line that marks a spot in a parking space. The pixels beneath this line are processed to determine if the spot is occupied.
Patch.cs	The parent of DifferencePatch and ParkingSpotPatch. Contains information as to where the line patch is located in the image, as well as the function for extracting the pixels beneath the line.
Program.cs	Contains the Main function that creates a new SmartLot application.
RunRoutine.cs	The heart of the application. This contains the routines for running selected

	occupancy-detection algorithms on the patches. An image is passed in, patches are extracted, occupancy-detection algorithms are preformed, information is combined, and finally the wireless-lights are updated.
ShowPatchesForm.cs	This is a pop-up window that shows the extracted ParkingSpotPatch and DifferencePatch, as well as the applied difference filter.
Win32.cs	This is a console that prints out debug information.
WirelessLightCtrl.cs	This is a client to the Wireless-Light. The results from the RunRoutine are passed in. An HTTP request is made to the wireless-light server and then a post is made to update the lights.

3.3.1. Instruction Manual on How to Set Up a Parking-Lot Map

Follow the steps below to configure a Parking-lot Map.

1) Start the SmartLot Application.
2) Press the image icon to open a saved image of a parking-lot, preferably open an image where the parking-lot is mostly empty.
3) Draw patch-lines on parking spaces by left-clicking and then left-clicking again. Place the lines in a location that will not overlap with another spot. See Figure 8 for a good example. Note: Patches can be deleted by pressing the Delete button.
4) If needed, shift a drawn patch line by using the 'A', 'S', 'D', and 'W' keys.
5) Select a patch or several patches at a time, and press the Difference button to make a duplicate-difference patch. Select patches by clicking on the item in the list. Hold down Ctrl and use the 'A', 'S', 'D', and 'W' keys to move the difference patch to a spot that will continuously show blank ground. Note: Entire rows can be done at a time.
6) Select any number of patches at a time and assign a row and spot number to the patch.
7) If needed, select a patch from the list and press the Extract button to see what the patch looks like extracted from the image. An example of this pop-up window is shown in Figure 10. This is helpful to verify that the patch isn't directly over a yellow line. A small amount of yellow line is acceptable, but if the patch is completely yellow, it will mess up the difference algorithm. The width of the line can be set by changing the value in the Extract Size field.

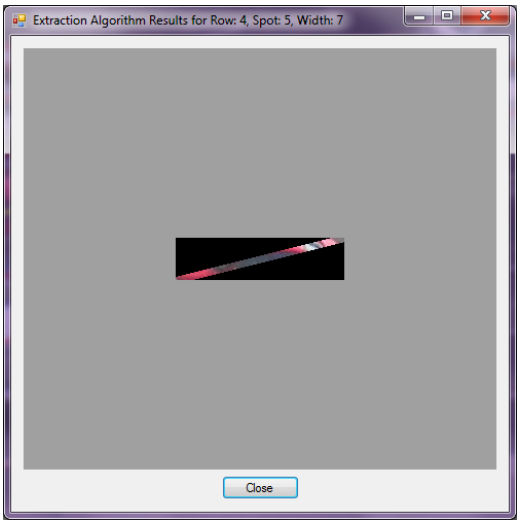

Figure 10 - Extracted Patch Window

8) The Patches button can be pressed to see both the parking spot patch and the difference patch. The pop up also shows the applied difference filter, as well as, how different the two images are from each other. An example is shown in Figure 11.

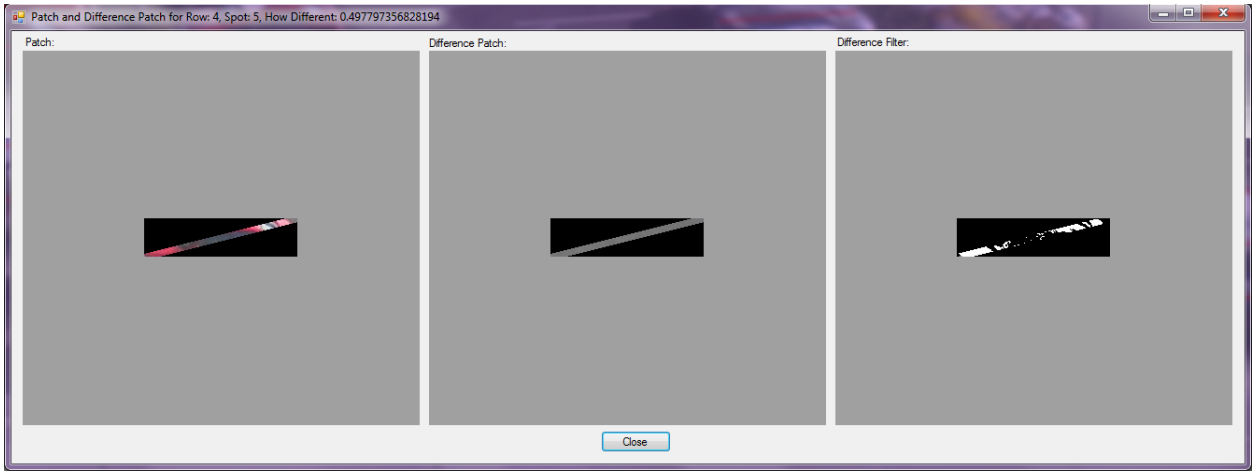

Figure 11 - Extracted Patches and Filter Window

9) Save the parking-lot configuration, by pressing the save icon and choosing a file name and location. A text file will be created containing the start and end point of each patch and difference patch. This saved parking-lot calibration file can be loaded at a later time by pressing the open icon and selecting this file.
10) Once the parking-lot is entirely laid out, choose an algorithm to test by using the drop down.
11) If the Wireless-Light is available, click the available check-box and enter the IP of the Wireless-Light Server.
12) To run a test, press the blue play button. Occupied spaces are shown red, vacant spaces are shown green.
13) To run a live application, switch to the Application tab.
14) Enter the IP of the FTP Server.
15) Enter the IP of the Wireless-Light Server and click available.
16) Press the green play button to run the application. When a new image arrives the data will be updated.
17) Stop the application by pressing the stop button.

3.4. DESIGN OF OCCUPANCY-DETECTION ALGORITHM

A number of occupancy-detection algorithms were experimented with throughout the development of the SmartLot system. Each algorithm was vigorously tested on many different parking spots and at many different times of day. As the result of testing, each algorithm was given a score from one to ten. In the end, the Difference Detection Algorithm came out on top. A description of each algorithm is given in Table 2. Subsequent headings highlight some of the more interesting algorithms for finding occupied parking spaces.

Table 2 - Occupancy-Detection Algorithm Descriptions

Algorithm	Description	Rank (1-10) for Occupied	Rank (1-10) for Vacant
All Algorithm Vote	This algorithm runs all of the algorithms listed below. Each algorithm has a vote for occupied or vacant. The voting points that each algorithm has for occupied or vacant are the	8	8

	same as the ranking points shown in this table. Unfortunately, the Difference Detection worked more reliably than all of the algorithms combined, so this algorithm was not used.		
Difference Detection	This algorithm compares two patches of the same size. One patch is the line covering a parking spot; the second is a line covering blank asphalt. A difference filter is applied to compare the two patches. After the difference filter is applied, brighter colors show bigger differences. Once the difference filter is applied, a threshold is then applied to change the image to entirely black and white. Then if the image reaches a set limit of white pixels, the spot is considered occupied. This algorithm works reasonably well, except for when the car is the same color of gray as the asphalt.	9	9
Edge Detection	An edge filter is applied to the overall image. When the edge filter is applied, sharp changes in color are highlighted in white. Each patch is then analyzed to see how many white edges appear in the patch. If a set limit is reached, the spot is considered occupied. Unfortunately this algorithm is only reliable when there are no painted lines, shadows, or oil spots in a vacant parking spot. Also, in some cases the patch can be put in a spot that is over a vehicle, but never crosses an edge – like the top of a long van.	2	9
Saturation	This algorithm looks for highly saturated colors, or bright colors, in the patch. Brighter colors are considered cars, and dull grays are considered vacant. This algorithm works well when the sun is directly shining on a patch, but if the patch is in a shadowy area or if the car is gray, this algorithm is highly unreliable. Also, if the patch covers part of a freshly painted yellow line, it will assume it is a vehicle.	3	5
Luminosity	This algorithm looks for bright reflections on the surface of vehicles. Like saturation, this algorithm is unreliable when the sun is not shining on the patch, or if the car has an old dull color. Also, at high-noon, the sun can reflect off the asphalt making this highly unreliable.	3	7
Standard Deviation	This algorithm creates a histogram of all pixel colors in a patch. There are many different colors over the surface of a vehicle, so if the standard deviation of the histogram is large, the spot is considered occupied; if the standard deviation is small the spot is considered vacant.	8	9

	This works well if a vacant spot is completely gray, but if a vacant spot has any variation in color, it throws off the standard deviation.		
Find Red/Blue	This algorithm looks to see if there exist any red hued or blue hued pixels in a patch. Red and Blue are unique to cars, but yellow, green, and other colors can be seen in painted lines, leaves, and other things existing in parking-lots. This algorithm works well for finding red and blue cars, but if it doesn't find red or blue it doesn't mean there isn't a vehicle in the spot.	9	1

3.4.1. Difference Detection Algorithm

The difference-detection algorithm proved to be the most reliable of any algorithm, even more so than the All Algorithm Vote. It is also one of the more simple solutions to the problem – compare a patch to a vacant patch, if they are the same the spot is empty, otherwise the spot is occupied. One question that may arise is "Why does the difference patch need to be taken from the current image?" Why not just compare the patch covering the parking spot with gray pixels? The idea is that the tone of gray changes over the course of the day. The ground is almost black at night, darker in the morning and evening, and lighter during the afternoon. Extracting the current color of the ground from each image was the best solution found.

This algorithm showed to be about 90% accurate. The other 10% of failures were gray vehicles matching the current color of the ground. An example of such a failure is shown in Figure 12. The patch on the left is a gray car, the patch in the middle is the current ground color, and the patch on the right is the image after the difference-detection filter. The window title shows that the two images are 20% different – the limit for an occupied space is 25% different. Why not just change the percentage to 20%? The limit was set at 25% to allow for shadows that lay differently across the two patches. Changing this value just made the solution fail in other ways.

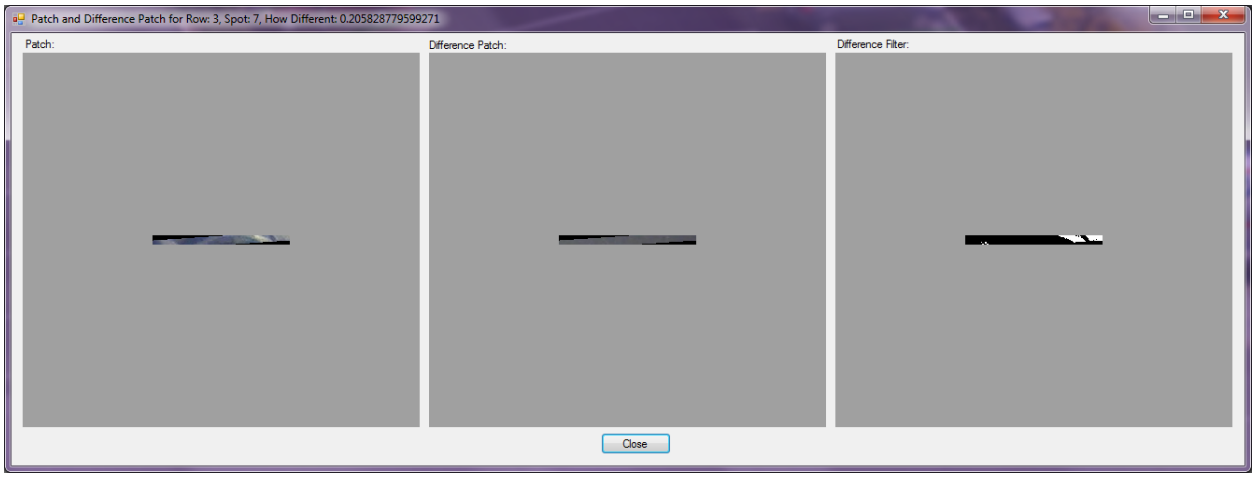

Figure 12 - Example of Difference Detection Failure

3.4.2. Edged Detection Algorithm

An example of an image with an edge filter applied is shown in Figure 13. The problem with the Edge Detection Algorithm is that painted lines, tree branches, and shadows also show up in the extracted patches.

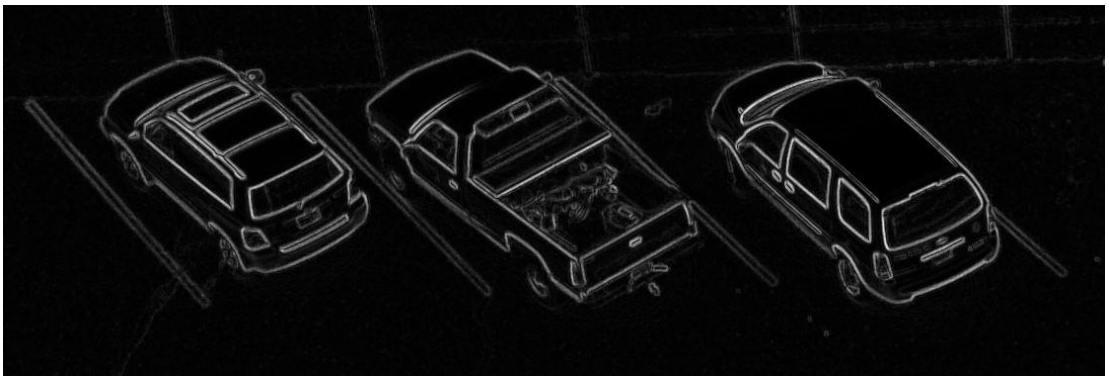

Figure 13 - Edge Filter

3.4.3. Standard Deviation Algorithm

Three ideal situations for the Standard Deviation Algorithm are shown in Figure 14. The histogram on the left shows an ideal situation for an occupied spot. There is a large variation in colors. The histogram in the middle shows an ideal situation for an empty parking spot. The pixels are basically all the same color, and thus the standard deviation is small. The histogram on the right shows an ideal situation for a vacant spot with a partial shadow. The graph has two spikes; one spike for the lighter gray and one spike for the shadow. The algorithm could handle these three ideal situations, but failed on situations that were not ideal.

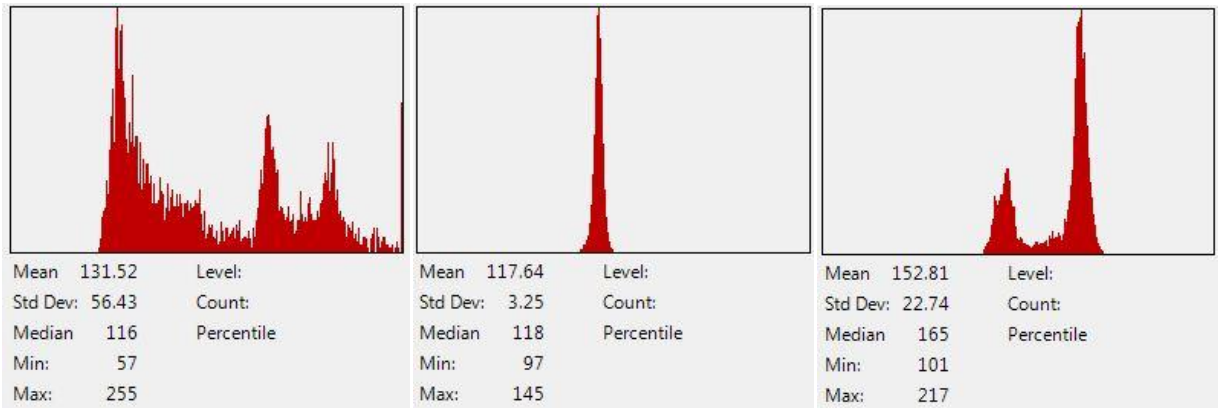

Figure 14 - Histograms

3.5. DESIGN OF WIRELESS-LIGHT CONTROLLER

The Wireless-Light Controller is the mechanism that controls the switching of the traffic lights out in the parking-lot. The device is designed to be wireless so that data lines won't need to be laid in an already existing parking-lot. In an ideal situation, traffic lights would be installed onto light poles, where they could easily access power. Each light would have its own wireless-microcontroller which would communicate with a server and control the switching from red to green. For the purposes of this project, a small-scale version of the behavioral science parking-lot is being used. LEDs are used here, rather than actual traffic lights, so instead of one microcontroller per traffic light, we have one microcontroller that

controls all of the traffic lights on our small-scale lot. A picture of a small-scale model of the behavioral science parking-lot is show in Figure 15. The behavioral science parking-lot is broken up into eight rows. Each row has a red and green LED associated with it. There is also an added blue LED on the far left to show if there are available handicap spots.

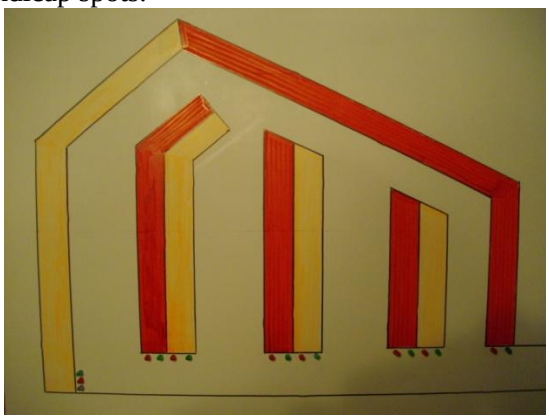

Figure 15 - Small Scale Parking-lot

A picture of the Arduino microcontroller attached to a wireless-internet shield is shown in Figure 16. The Arduino acts as a small web server. A client can make a POST request to the server to change the lights. The request is received through a WiFly GSX card and passed to the Arduino. The Arduino processes the request and sends eight bits of serial data out to the breadboard. The breadboard parallelizes the serial data out to the LEDs. A seven-segment-display is also added to show the signal strength of the server. Multiplying the number on the display by ten gives the signal strength percentage. A complete schematic of the Wireless-Light is shown in Figure 17 and a link to the schematic of the WiFly shield is given in 7.APPENDIX C.3.

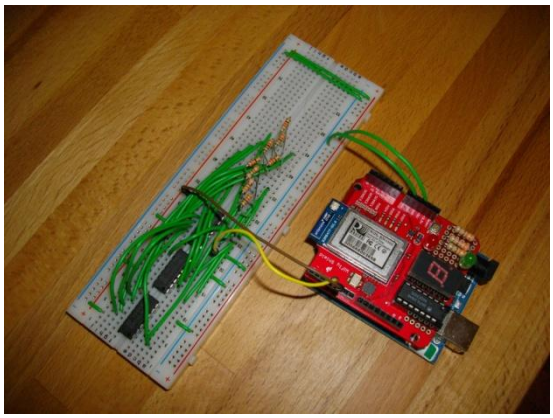

Figure 16 - Wireless-light Server

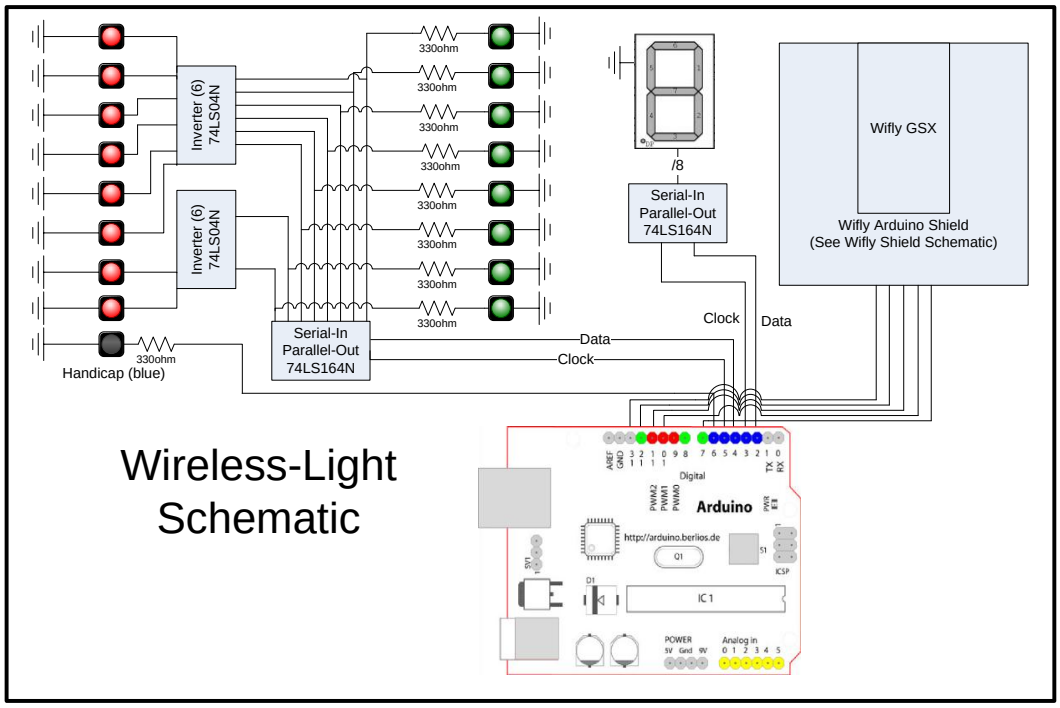

Figure 17 - Schematic of Wireless-Light

The WiFly GSX has a simple UART interface, but the Arduino doesn't actually interface with the shield through UART, it interfaces through a SPI-UART bridge that allows for a higher baud rate. In order to send commands to the WiFly, the WiFly first needs to be in command mode. Sending "$$$" puts the device in command mode. Once in command mode, commands can be sent to the WiFly to join a password protected network, send data, received data, get the signal strength, and much more. A complete list of commands can be found in the WiFly User Guide, found on the Sparkfun website. Sparkfun also provides an experimental Arduino library for their WiFly shield that abstracts much of these lower level commands to the WiFly GSX (see 7.APPENDIX C.2). However, the library does not include a command to get the signal strength, so this was added, because we were interested in the range of our server.

Once the device was up and talking to a wireless router, a simple web server was written for the Arduino. The Arduino web server code can be found in 7.APPENDIX C.1. This code functions as follows: At startup, the sever joins a specified password-protected wireless-network and is assigned an IP. It then waits for a client to connect. A client can make a request to the server by using the IP address assigned to the server. Once a client is connected, the server responds by sending some HTML code. The client can then make a POST to the server by sending a string of ones and zeros – "01010101" for example. This string is then parsed out of the POST and used to update the LEDs. Ones will turn a row green, and zeros will turn a row red.

4. LESSONS LEARNED

Throughout the project, we encountered a few unexpected problems, such as how long it took us to get some test images. This was mainly because we didn't have the camera setup early enough and it took us longer than planned. Another problem that was also unexpected was the power supply for the camera. This was easily resolved by implementing one. Overall, we only had a few small unforeseen issues that came up. The most important lesson we feel we learned from this project was the result of the final implementation of each individual component and how each component influenced the others. From previous classes or projects, when we finished the assignment, it was turned in and checked off. The

project was never used as part of another project. So it only mattered if it worked at that moment of check-off.

5. CONCLUSION

As a result of this project we have a working prototype of the SmartLot system. The system performed well; not to our wildest expectations, but pretty well for a first attempt. We learned a lot and established a good foundation for future improvements. We now have working tools for calibrating parking-lots, a large set of test images, and much more experience in the image processing realm. Where might SmartLot go in the future? Well, maybe to SmartLot2.0 or even GeniousLot. The biggest obstacle right now is getting the occupancy-detection algorithms to work reliably in all situations – including different times of day, different weather conditions, and different camera angles. As of now the system is only about 90% accurate. Sure some of these bugs would be flushed out if there were more cameras, each responsible for only 10 to 20 parking stalls each, but the algorithm might still get confused on gray cars. A cure all for many of the bugs would be to make the system dynamic. As of now the system is fairly static. For one, the parking-lot is calibrated at a specific angle and if the camera is bumped or changes position, everything must be re-calibrated. Another static characteristic is using patches to define where the cars are expected to park. A better and more dynamic approach would be to track the vehicles using motion detection. Once a vehicle stops in a parking spot, it will be determined to still be there until it again moves. A dynamic approach could help move the accuracy closer to 100%. After all, the only thing worse than missing out on a closer parking space, is being tricked out of a closer parking space by a machine.

6. ACKNOWLEDGEMENTS

- AForge.NET : http://www.aforge.net

AFoge.NET provides an image processing library and image processing lab free of charge. The image processing lab was used to prototype different potential algorithms for parking spot vacancy detection. Once an algorithm was chosen it was implemented in C# using the image processing library.

- cURL : http://curl.haxx.se

cURL provides a C library that contains the functionality required to perform FTP transactions. This library was used on the camera controller to upload recently taken photos to the image store.

- gPhoto : http://www.gphoto.org

gPhoto provides an open source library free of charge that contains the functionality required to communicate with many off the shelf digital cameras over PTP. This library was used in the camera controller to control the digital camera and acquire images from it.

- SparkFun Electronics : http://www.sparkfun.com

SparkFun provides many different devices for Arduino based projects including the WiFly shield that was used in SmartLot to implement the wireless-light. They also have many tutorials on how to use their products which were used to aid in implementation of the wireless-light.

- Sharon Benivides

Sharon is the head of the Behavioral Science department at the University of Utah and allowed us to use an office in the Behavioral Science building to set up our camera.

7. BIBLIOGRAPHY

(1) Power Supply Schematic: http://www.elecircuit.com/
(2) Bridge Diagram: http://electronicsandyou.com/

APPENDIX A: SMARTLOT APPLICATION

APPENDIX A.1 SmartLot Application Code

The complete SmartLot application code can be found on the SmartLot webpage:
http://www.kamsheffield.com/school/smartlot/src/smartlot/

APPENDIX A.2 SmartLot Application Class Diagram

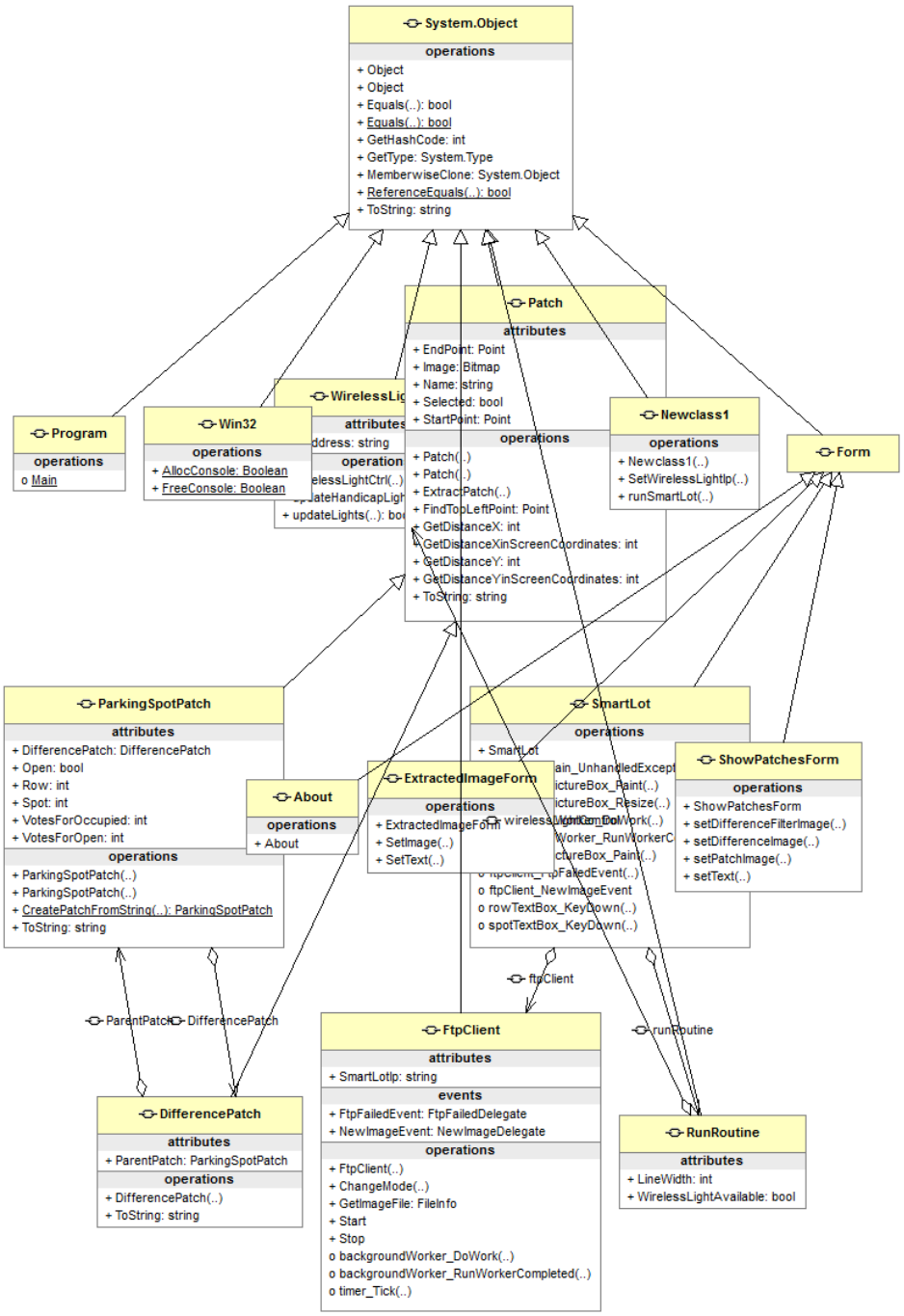

Figure 18 - SmartLot Application Class Diagram

APPENDIX B: CAMERA CONTROLLER CODE

The code for the camera controller can be found on the SmartLot webpage:
http://www.kamsheffield.com/school/smartlot/src/camera/

APPENDIX C: WIRELESS-LIGHT

APPENDIX C.1 Wireless-Light Server Code

```
/*
 * Wireless-light Web Server
 *
 * A simple web server that serves up some controls for a wireless parking-lot.'
 * The parking-lot has 8 red-green lights and one blue handicap light
 */

#include "WiFly.h"
#include "Credentials.h"

#define rows_data 4
#define rows_clock 5
#define seven_seg_data 2
#define seven_seg_clock 3
#define ledBlue 6
#define USE_SIGNAL_STRENGTH 1

// These numbers are for the signal strength 7 seg display.
byte zero  = B11111100;
byte one   = B01100000;
byte two   = B11011010;
byte three = B11110010;
byte four  = B01100110;
byte five  = B10110110;
byte six   = B10111110;
byte seven = B11100000;
byte eight = B11111110;
byte nine  = B11110110;
byte nada  = B00000000;

// These numbers are for the 8 rows of parking spaces.
byte rows[8]  =
{
  B10000000,
  B01000000,
  B00100000,
  B00010000,
  B00001000,
  B00000100,
  B00000010,
  B00000001
};

int timer     = 0;
int signal    = 0;

char buffer[700];
boolean new_rows_string = false;
boolean handicap_on = true;
int location_in_string = 0;

Server server(80);

// This runs once at the begining.
void setup() {
  pinMode(ledBlue, OUTPUT);
  pinMode(seven_seg_clock, OUTPUT);
  pinMode(seven_seg_data, OUTPUT);
  pinMode(rows_data, OUTPUT);
  pinMode(rows_clock, OUTPUT);
```

```
  // Initialize the signal strength to 0.
  shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, zero);
  // Initialize the parking rows to all Green (meaning open).
  shiftOut(rows_data, rows_clock, LSBFIRST, nada);
  // Initialize the handicap spots to blue (meaning open).
  digitalWrite(ledBlue, HIGH);

  Serial.begin(9600);
  Serial.println("Begining...");

  WiFly.begin();
  Serial.println("Joining...");

  while(!WiFly.join(ssid, passphrase))
  {
   delay(10000);
  }

  // Print the IP address of the wireless-light server
  Serial.print("IP: ");
  Serial.println(WiFly.ip());

  server.begin();
}

// This is the main loop of the Wireless-light server.
void loop() {

  Client client = server.available();
  if (client) {
    Serial.println("got client");
    // an http request ends with a blank line
    boolean current_line_is_blank = true;
    boolean ignore_the_rest_of_line = false;

    int index = 0;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();
        buffer[index] = c;
        if(c == -1)
        {
          // Nothing to read.
          Serial.println("Nothing to Read");
          break;
        }
        // if we've gotten to the end of the line (received a newline
        // character) and the line is blank, the http request has ended,
        // so we can send a reply
        if (c == '\n' && current_line_is_blank) {
          client.print("<html>");
          client.print("<title>Psychology Lot</title>");
          client.print("<form name=\"input1\" action=\"@H\" method=\"get\">");
          client.print("<input type=\"submit\" name=\"button1\" value=\"handicap\">");
          client.print("</form>");
          client.print("Type an eight digit binary number:");
          client.print("<form name=\"input\" action=\"$R\" method=\"get\">");
          client.print("<input type=\"text\" name=\"%\" size=\"100\">");
          client.print("<input type=\"submit\" value=\"Submit\">");
          client.print("</form>");
          client.print("</html>");
          ignore_the_rest_of_line = false;
          break;
        }
        if (c == '\n') {
          // we're starting a new line
          current_line_is_blank = true;
        }
        else if(c == '%' && !ignore_the_rest_of_line)
```

```
      {
        // We received a string of 8 binary numbers, meaning change the lights.
        location_in_string = index;
        new_rows_string = true;
        ignore_the_rest_of_line = true;
      }
      else if (c == '@' && !ignore_the_rest_of_line)
      {
        // We received a command to toggle the handicap light
        if(handicap_on)
        {
          digitalWrite(ledBlue, LOW);
          handicap_on = false;
        }
        else
        {
          digitalWrite(ledBlue, HIGH);
          handicap_on = true;
        }
        ignore_the_rest_of_line = true;
      }
      else if (c != '\r') {
        // we've gotten a character on the current line
        current_line_is_blank = false;
        //Serial.print(c);
      }

      index++;
     }
    }
    // give the web browser time to receive the data
    delay(100);
    client.stop();
  }

  if(new_rows_string)
  {
    light_up_the_lights(location_in_string);
  }

  if(USE_SIGNAL_STRENGTH)
  {
    timer++;
    // At the timer, update the signal strength.
    if(timer > 32000)
    {
      update_signal_strength();
    }
  }
}

// Converts an 8 bit string to an 8 bit number to be shifted out to the LEDs.
void light_up_the_lights(int index)
{
  byte output = B00000000;
  char binary_num[8];
  strncpy(binary_num, &buffer[index+4], 8);
  new_rows_string = false;
  location_in_string = 0;

  for(int i = 0; i < 8; i++)
  {
    if(binary_num[i] == '1')
    {
      output = output | rows[i];
    }
  }
  shiftOut(rows_data, rows_clock, LSBFIRST, output);
}
```

```
void update_signal_strength()
{
   timer = 0;
   signal = atoi(WiFly.getSignalStrength());
   signal = signal + 100;
   if(signal > 90)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, nine);
   else if(signal > 80)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, eight);
   else if(signal > 70)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, seven);
   else if(signal > 60)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, six);
   else if(signal > 50)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, five);
   else if(signal > 40)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, four);
   else if(signal > 30)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, three);
   else if(signal > 20)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, two);
   else if(signal > 10)
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, one);
   else
     shiftOut(seven_seg_data, seven_seg_clock, LSBFIRST, zero);
}
```

APPENDIX C.2 WiFly Arduino Library

http://sparkfun.com/Code/WiFly/WiFly-20 ... lpha-2.zip

APPENDIX C.3 WiFly-Shield Schematic

The WiFly-Shield Schematic can be found on the Sparkfun Electronics website:
http://www.sparkfun.com/datasheets/DevTools/Arduino/WiFly_Shield-v14.pdf