Robot Vision: Follow the Leader

Shawn Weisfeld

University College: Orlando Campus

Florida Institute of Technology
2420 Lakemont Avenue

Suite 190

Orlando FL 32814

shawn@shawnweisfeld.com

http://www.shawnweisfeld.com

Abstract—**The focus of this study is to examine an implementation of "follow the leader". This process will use a vision processing algorithm to acquire a target then tracking logic to determine the best way to move the robot to follow that target.**

I. Introduction

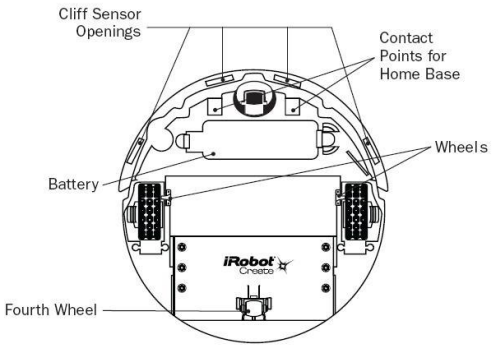
Robots are becoming more and more prevalent in the world today, from toys such as the Sony Aibo, to working robots similar to the iRobot Roomba, and research robots like the Honda Asimo. Robots have also become a major focus of military planners for use in hostile environments for tasks like bomb disposal, intelligence gathering, and military supply missions. It has been mandated by Congress that by 2015 one third of all military ground vehicles shall be unmanned. Technology developed to accomplish this could also play a large role in civilian motorized transportation.

The goal of this project was to develop a vision and drive system for robots so that they would be able to identify specified objects and perform a game of "follow the leader".

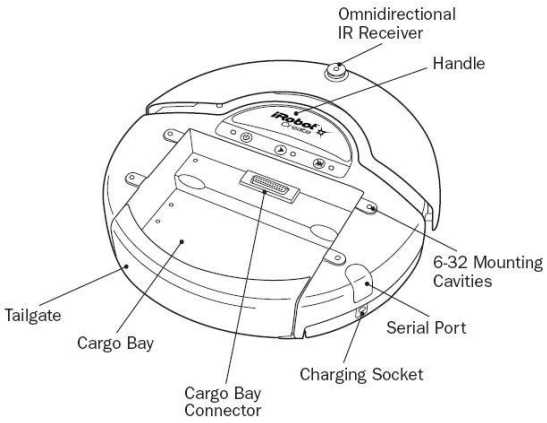
II. The Robot

Since the main goal of this project was to develop the vision processing aspects of the "follow the leader" problem, an existing robot kit was selected and modified for this research. The iRobot Create kit (http://www.irobot.com) was purchased and assembled to form the base of the robot. The kit came with the robot chassis, including sensors, and a battery. However the kit did not include the camera necessary for the robot vision. To that end the Panasonic BL-C30A was purchased and mounted to the robot.

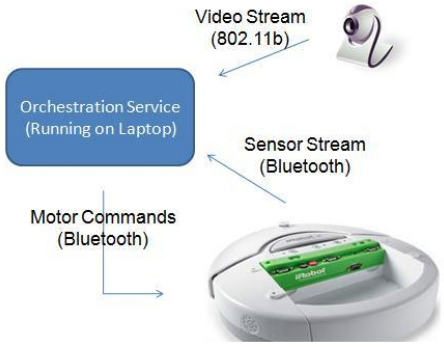
Bottom View



Top View



The iRobot Create platform provided a robust foundation for this project. The robot uses a differential drive system. A differential drive system consists of two wheels. To move forward or backward both wheels are turned on to the same power level. To turn right or left only one of the wheels is moved causing the robot to pivot on the other. The robot is also equipped with a suite of sensors and a command module that are not being used for this project.



Since the robot

does not have enough onboard processing power and is too small to have an onboard laptop the robot was designed to perform in a wireless tethered fashion. This means that all processing could take place on a powerful laptop computer and the instructions are sent wirelessly to the robot. To that end a RooTooth, a Bluetooth wireless connector, was added to the robot. To capture images the Panasonic camera was used. This camera is able to stream live motion video to the laptop via an 802.11b connection.

III. The Software

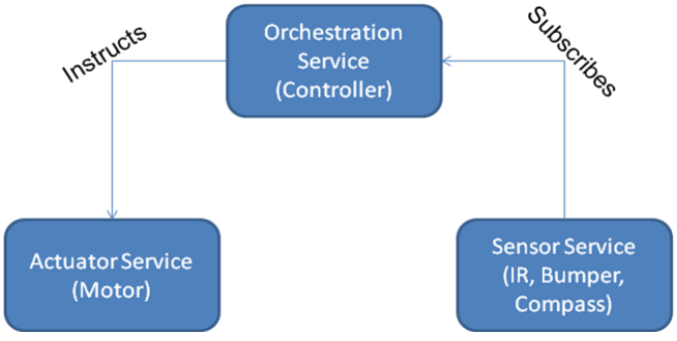
The software was written and implemented utilizing C#.NET, part of the Microsoft .NET (pronounced dot net) Framework and the Microsoft Robotics Studio.

A. Microsoft Robotics Studio

Microsoft Robotics Studio (MSRS) is a framework for robotic development that sits on top of the Microsoft .NET Framework v2.0. While the .NET Framework provides generalized Application Programming Interfaces (APIs) for common programming needs, MSRS provides a specific set of APIs and a runtime especially designed to meet the needs of developers working with robotics. The MSRS runtime consists of two major runtime components. The Concurrency and Coordination Runtime (CCR) and the Decentralized System Services (DSS). MSRS also contains a suite of development tools simplifying the development of applications for robots.

The CCR allows for the coordination of messages between the different components of the robot without the need for manual threading, locks, or semaphores. The DSS is a service hosting environment and a set of basic services facilitating tasks such as debugging, logging, monitoring, security, discovery, and data persistence.

With MSRS the developer writes a service for each part of the robot. For example a drive service was written to control the motion of the wheels and services were written for each type of sensor. This service oriented design means that adding an additional sensor of the same type to a robot does not require writing any additional code. The user just needs to configure an additional instance of the sensor's service. Each service communicates with the others by sending messages to the DSS.

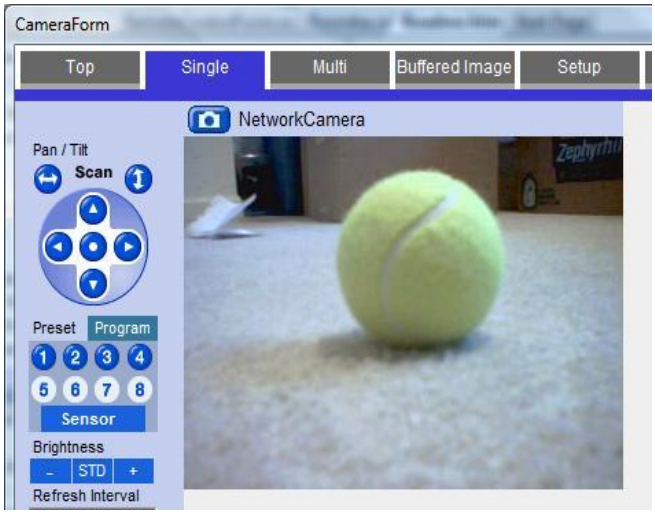


The DSS decides what service should get the message and passes it along. Since so many messages are being sent and received at the same time the CCR enables this process to thread out across the machine utilizing the full processing power of the computer.

A simple robot consists of three services that have been loaded into the DSS. Each suite of services must have an orchestration service. This service acts as the controller. The DSS knows to route messages from the input services (sensors) to the orchestration service through a process called subscription. Conversely through the process of instruction the orchestration service sends messages to actuator services. As the environment becomes more complex, the DSS can route messages from many different sensors to the orchestration service then instructions can be sent to many different actuators.

B. Image Capture

The Panasonic camera transmits a live video stream via an internal web server that is accessible over 802.11b and the built-in wireless card. The web page with the video utilizes an ActiveX control to display the image. The webpage is accessed via a browser window embedded into the orchestration service. In a tight loop a still image is taken of the live picture and the image is analyzed. This process utilizing a quick laptop, multithreading, and the CCR, can capture and process about 20 frames per second.



C. Target Acquisition

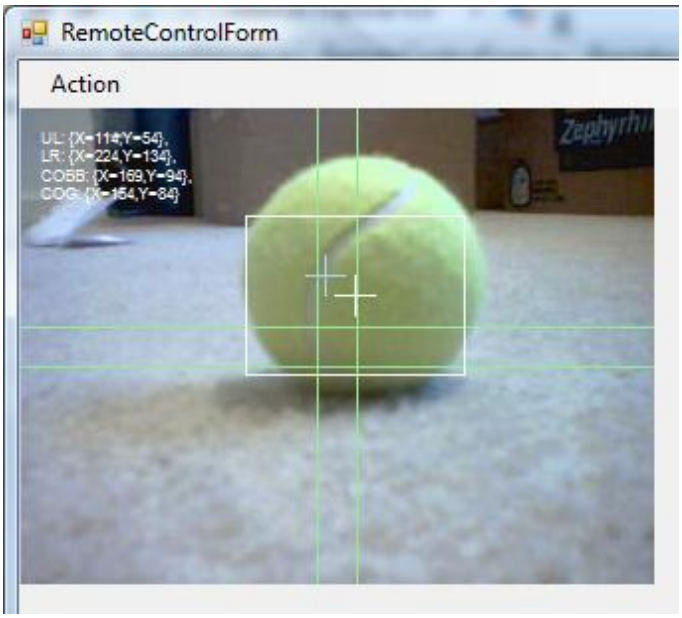
The first step in target acquisition is to determine what the user wants the robot to track. This is done by requesting the user double click on the object to track. When the user selects the object the software goes through a process to remove the background. Once the background is removed the center of mass of the target is found. Finally, a bounding box is calculated to

find the size of the target. All of these processes take advantage of an image processing library that was written for this project.

The managed code for image handling in the .NET framework eliminates the need for pointers and the like, but it does not have the performance that an unmanaged library has due to the management overhead. This overhead is very expensive when processing the quantities of pixels that need to be processed. To that end an unmanaged library was written in C#. This library unlocks the bitmap then takes the raw bits and forces them into a programming structure, known as a struct in C#, to determine the pixel value at any point in the image.

1) *Background Removal*: Background removal is done by iterating through all the pixels in the image. Each pixel is examined for its current value. If that RGB value is not within a given threshold of that selected by the user the pixel color is set to black. Black is used to simplify the process of finding the center of mass, the next step in the process. This is actually a process of color segmentation where all the colors close to the target color are kept and all others are removed.

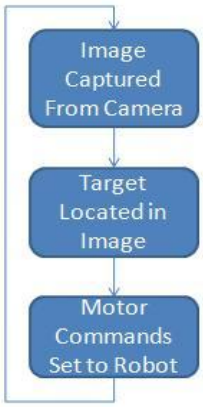
2) Center of Mass: The center of mass or center of gravity is calculated by averaging the luminance value of each of the pixels in the image.



3) *Bounding Box*: After identifying the center of the target we need to find a bounding box approximating the size of the target as illustrated in figure 5. To do this, starting at the center of mass, the program samples the pixel values up, left, right, and down from the center of mass to see if they still have some luminance to determine that they are part of the target. This process is repeated until the pixel with the smallest x and y values (upper left corner) and the pixel with the largest x and y value (lower right corner) are found that still represent part of the target. Since the robot knows the size of the target and now know how big the target appears to be in the image it can calculate the distance the robot is away from the target.

IV. Tracking Logic

Since the robot knows the distance the robot is away from the target it can now send instructions to the wheels to move the robot forward or backward to keep the target the desired distance away. To determine if the robot should turn, the center of the bounding box is found. If that center point is in the left one third of the picture the robot turns left. Conversely if the center point is in the right one third of the picture the robot turns right. A priority is given to the turning commands since it is important that the robot keeps the target centered in the camera because if the target falls out of the frame the robot will not know where to go.



V. Tests

The robot was tested in a real world environment tracking a few different targets. In the first test a tennis ball was used in a room with tan color carpet. The robot was able to follow the target until it was moved into a sunny area of the room where the color of the tennis ball was washed out and the robot could not differentiate between it and the carpet. A similar problem occurred when a racket ball was used and it was passed next to another object of a similar blue color. Both problems can be traced to the idea of only using color segmentation to do the target acquisition. On the other hand the robot was able to follow the target with little problem when there was a good contrast between the color of the target and the background.

VI. Conclusions

This study demonstrates that with image processing code and basic tracking logic a robot can be programmed to play follow the leader. This is the first step in what has practical implications in both the civilian and defense industries. Today most passenger cars are equipped with cruise control allowing the driver to keep the car's speed at a constant, but this does nothing for drivers in stop and go traffic. This vision system could be used to allow a driver to set the distance to be maintained from the car ahead so that if the car ahead slows, the following programmed vehicle slows, and if the lead car goes around a turn, the robot-controlled car would also go around the turn. This would also be very useful for military applications. For example, the military could have just one manned vehicle leading a convoy and have all the other vehicles equipped to

follow the lead truck thus reducing the number of solders placed in danger's way when bringing in the fuel, bullets, and beans.

Some possible enhancements to this would be making the vision system more robust and not relying on color segmentation to find the target. Additionally the robot could be enhanced to actually capture the target.

Acknowledgment

I would like to thank Holger Findling and the staff at the FIT Orlando Campus for their countless hours working with me on this project. Without their guidance this project would not have been possible.

About the Author

Shawn is a Sr. Developer at an Orlando, FL based Fortune 100 company. There he specializes in intranet and smart client development for business applications. Besides his day job Shawn also is an Adjunct Professor at The Florida Institute of Technology and does software development work for local small businesses. In his free time he volunteers as the President of the Orlando .NET users group and at other local charities. Shawn started his career at his family business in Port St. Lucie, FL while working on his undergraduate degree in Business Administration at the University of Central Florida and after a year off Shawn moved back to Orlando to pursue a Masters degree in Management Information Systems at The University of Central Florida. Currently Shawn is working on a second Masters degree in Computer Science at FIT. Shawn is also the INETA NorAm Membership Manager in Florida and a Microsoft C# MVP.

References

[1] Gunnerson, Eric: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp11152001.asp
[2] iRobot: http://www.irobot.com
[3] Microsoft Robotics Studio: http://www.microsoft.com/robotics
[4] Shapiro, Linda; Stockman, George. Computer Vision. 2001, Prentice Hall

Appendix
Application Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Configuration;

namespace Microsoft.Robotics.Services.IRobot.Roomba
{
    /// <summary>
    /// The directions the robot can move
    /// </summary>
    public enum MoveDirection
    {
        /// <summary>
        /// up
        /// </summary>
        Up,
        /// <summary>
        /// back
        /// </summary>
        Back,
        /// <summary>
        /// left
        /// </summary>
        Left,
        /// <summary>
        /// right
        /// </summary>
        Right,
        /// <summary>
        /// stop
        /// </summary>
        Stop
    }

    /// <summary>
    /// a form to display the image tracking data
    /// </summary>
    public partial class RemoteControlForm : Form
    {
        RoombaService _svc = null;
        private Color _trackingColor = Color.Blue;
```

```
CameraForm _camera = null;

/// <summary>
/// constructor
/// </summary>
public RemoteControlForm()
{
    InitializeComponent();
}

/// <summary>
/// constructor
/// </summary>
/// <param name="svc"></param>
public RemoteControlForm(RoombaService svc) :this ()
{
    _svc = svc;
}

/// <summary>
/// allows for driving of the robot using the keys on the keyboard
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RemoteControlForm_KeyUp(object sender, KeyEventArgs e)
{
    switch (e.KeyCode)
    {
        case Keys.Up:
            MoveRobot(MoveDirection.Up);
            break;

        case Keys.Down:
            MoveRobot(MoveDirection.Back);
            break;

        case Keys.Left:
            MoveRobot(MoveDirection.Left);
            break;

        case Keys.Right:
            MoveRobot(MoveDirection.Right);
            break;

        case Keys.Space:
            MoveRobot(MoveDirection.Stop);
            break;
    }
}

/// <summary>
/// togle the tracking on and off
```

```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void trackTargetToolStripMenuItem_Click(object sender, EventArgs e)
{
    trackTargetToolStripMenuItem.Checked = !trackTargetToolStripMenuItem.Checked;
}

/// <summary>
/// start image capturing
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void startImageCaptureToolStripMenuItem_Click(object sender, EventArgs e)
{
    bwImageProcessor.RunWorkerAsync(_camera.GetSnapshot());
}

/// <summary>
/// Move the robot
/// </summary>
/// <param name="m"></param>
private void MoveRobot(MoveDirection m)
{
    Console.WriteLine(m);
    switch (m)
    {
        case MoveDirection.Up:
            _svc.SendCommand(new CmdDrive(100, 32768), _svc._mainPort);
            break;
        case MoveDirection.Left:
            _svc.SendCommand(new CmdDrive(100, 100), _svc._mainPort);
            break;
        case MoveDirection.Right:
            _svc.SendCommand(new CmdDrive(-100, 100), _svc._mainPort);
            break;
        case MoveDirection.Back:
            _svc.SendCommand(new CmdDrive(-100, 32768), _svc._mainPort);
            break;
        case MoveDirection.Stop:
            _svc.SendCommand(new CmdDrive(0, 32768), _svc._mainPort);
            break;
    }
}

/// <summary>
/// Process the image (this is on a different thread
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void bwImageProcessor_DoWork(object sender, DoWorkEventArgs e)
{
```

```
            DateTime start = DateTime.Now;
            Point upperLeft = new Point(0, 0);
            Point lowerRight = new Point(0, 0);
            Point cog = new Point(0, 0);
            Image img = e.Argument as Image;
            Font f = new Font("Arial", 6);
            Point coi = new Point(img.Width / 2, img.Height / 2);

            try
            {
                //Find the target
                ImagerBitmap.FindTheBall(img as Bitmap,
                    _trackingColor, 60,
                    ref cog, ref upperLeft, ref lowerRight);

                //Find the center of the bounding box
                Point cobb = new Point(
                    ((lowerRight.X - upperLeft.X) / 2) + upperLeft.X,
                    ((lowerRight.Y - upperLeft.Y) / 2) + upperLeft.Y);
                int offset = 10;

                //Draw the targeting information on the image
                using (Graphics g = Graphics.FromImage(img))
                {
                    string status = string.Format("UL: {0}, \r\nLR: {1}, \r\nCOBB: {2}, \r\nCOG:
{3}",
                        upperLeft, lowerRight, cobb, cog);
                    g.DrawString(status, f, Brushes.White, 10, 10);

                    g.DrawLine(Pens.LightBlue, cog.X - offset, cog.Y, cog.X + offset, cog.Y);
                    g.DrawLine(Pens.LightBlue, cog.X, cog.Y - offset, cog.X, cog.Y + offset);

                    g.DrawLine(Pens.LightGreen, 0, coi.Y - offset, img.Width, coi.Y - offset);
                    g.DrawLine(Pens.LightGreen, 0, coi.Y + offset, img.Width, coi.Y + offset);
                    g.DrawLine(Pens.LightGreen, coi.X - offset, 0, coi.X - offset, img.Height);
                    g.DrawLine(Pens.LightGreen, coi.X + offset, 0, coi.X + offset, img.Height);

                    g.DrawLine(Pens.White, cobb.X - offset, cobb.Y, cobb.X + offset, cobb.Y);
                    g.DrawLine(Pens.White, cobb.X, cobb.Y - offset, cobb.X, cobb.Y + offset);
                    g.DrawRectangle(Pens.White, upperLeft.X, upperLeft.Y, lowerRight.X -
upperLeft.X, lowerRight.Y - upperLeft.Y);
                }

                //move the robot
                if (trackTargetToolStripMenuItem.Checked)
                {
                    int third = img.Width / 3;
                    if (cobb.X < third)
                    {
                        MoveRobot(MoveDirection.Left);
                    }
                    else if (cobb.X > third * 2)
```

```
                {
                    MoveRobot(MoveDirection.Right);
                }
                else if (lowerRight.X - upperLeft.X > 60)
                {
                    MoveRobot(MoveDirection.Back);
                }
                else if (lowerRight.X - upperLeft.X < 50)
                {
                    MoveRobot(MoveDirection.Up);
                }
                else
                {
                    MoveRobot(MoveDirection.Stop);
                }
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }

        e.Result = img;
        TimeSpan span = DateTime.Now.Subtract(start);
        Console.WriteLine("Image Processed in {0} seconds.", span.TotalSeconds);
    }

    /// <summary>
    /// When we are done processing the image, display it, then process another
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void bwImageProcessor_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
    {
        if (e.Result != null)
        {
            pbMain.Image = e.Result as Image;
        }

        bwImageProcessor.RunWorkerAsync(_camera.GetSnapshot());
    }

    /// <summary>
    /// Get the target color from the pixel the user clicked on
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void pbMain_MouseDoubleClick(object sender, MouseEventArgs e)
    {
        Bitmap b = pbMain.Image as Bitmap;
```

```
            _trackingColor = b.GetPixel(e.X, e.Y);
        }

        /// <summary>
        /// Load the form
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void RemoteControlForm_Load(object sender, EventArgs e)
        {
            _camera = new CameraForm();
            _camera.Show();
        }


    }
}
```

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Drawing2D;
//using ChartDirector;
using System.IO;
using System.Collections;
using System.Collections.Generic;

namespace Microsoft.Robotics.Services.IRobot.Roomba
{
    /// <summary>
    /// This class provides methods to Resize and Apply Filters to bitmap images.
    ///
    /// ideas from
    /// http://msdn.microsoft.com/library/default.asp?url=/library/en-
us/dncscol/html/csharp11152001.asp
    ///
http://www.msnewsgroups.net/group/microsoft.public.dotnet.languages.csharp/topic9351.aspx
    ///
    /// </summary>
    public unsafe class ImagerBitmap
    {
        /// <summary>
        /// This struct is used to hold the RGB values when we find a pixel using the pointer
        /// </summary>
        private struct PixelData
        {
            public byte Blue;
            public byte Green;
            public byte Red;
        }

        /// <summary>
        /// This object contains pointer addressable information about the bitmap
        /// </summary>
        private BitmapData _bitmapData = null;

        /// <summary>
        /// The pointer to the upper left corner of the bitmap
        /// </summary>
        private byte* pBase = null;

        private Bitmap _bitmap = null;

        /// <summary>
        /// Get the Image we are working with
        /// </summary>
        public Bitmap Bitmap
        {
            get
            {
```

```
        return _bitmap;
    }
}

/// <summary>
/// Constructor
/// </summary>
/// <param name="b">Bitmap to use as the basis of the Imager</param>
private ImagerBitmap(Bitmap b)
{
    if (b.PixelFormat != PixelFormat.Format24bppRgb)
    {
        //Convert the image into Format24bppRgb since our unmanaged code
        //can walk that image type
        Bitmap b2 = new Bitmap(b.Size.Width, b.Size.Height, PixelFormat.Format24bppRgb);
        Graphics g = Graphics.FromImage(b2);
        g.DrawImage(b, new Point(0, 0));
        _bitmap = b2;
        g.Dispose();
    }
    else
    {
        _bitmap = b;
    }
    LockBitmap();
}

/// <summary>
/// Load a pixel with color
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <param name="c">Color</param>
private void SetPixel(int x, int y, Color c)
{
    PixelData* p = PixelAt(x, y);
    p->Red = c.R;
    p->Green = c.G;
    p->Blue = c.B;
}

/// <summary>
/// Get a pixel
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <returns>Color</returns>
private Color GetPixel(int x, int y)
{
    PixelData* p = PixelAt(x, y);
    return Color.FromArgb((int)p->Red, (int)p->Green, (int)p->Blue);
}
```

```
        /// <summary>
        /// Get the Grey value for a give pixel
        /// </summary>
        /// <param name="column">Column</param>
        /// <param name="row">Row</param>
        /// <returns>Grey value for the pixel</returns>
        private int GetGreyPixel(int column, int row)
        {
            return (int)((GetPixel(column, row).R * 0.3) + (GetPixel(column, row).G * 0.59)
                + (GetPixel(column, row).B * 0.11));
        }

        /// <summary>
        /// Use the 2 most signifigant bits from each of the values in the RGB to get an in
value
        /// for the histogram
        /// </summary>
        /// <param name="column">Column</param>
        /// <param name="row">Row</param>
        /// <returns>Histogram Value</returns>
        public int GetRGBHistogramValue(int column, int row)
        {
            Color c = GetPixel(column, row);
            int val = 0;
            int tmp = 0;
            tmp = c.R;
            if (tmp - 128 > 0)
            {
                tmp -= 128;
                val += 32;
            }
            if (tmp - 64 > 0)
            {
                val += 16;
            }
            tmp = c.G;
            if (tmp - 128 > 0)
            {
                tmp -= 128;
                val += 8;
            }
            if (tmp - 64 > 0)
            {
                val += 4;
            }
            tmp = c.B;
            if (tmp - 128 > 0)
            {
                tmp -= 128;
                val += 2;
            }
```

```
    if (tmp - 64 > 0)
    {
        val += 1;
    }
    return val;
}

/// <summary>
/// Lock the bitmap so we can use the pointers to access it
/// </summary>
private void LockBitmap()
{
    _bitmapData = _bitmap.LockBits(new Rectangle(0, 0, _bitmap.Width, _bitmap.Height),
        ImageLockMode.ReadWrite, _bitmap.PixelFormat);
    pBase = (Byte*)_bitmapData.Scan0.ToPointer();
}

/// <summary>
/// Get a pointer to a pixel
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <returns></returns>
private unsafe PixelData* PixelAt(int x, int y)
{
    return (PixelData*)(pBase + y * _bitmapData.Stride + x * sizeof(PixelData));
}

/// <summary>
/// Unlock the bitmap to end the pointer access session
/// </summary>
private void UnlockBitmap()
{
    _bitmap.UnlockBits(_bitmapData);
    _bitmapData = null;
    pBase = null;
}

/// <summary>
/// Get a 3x3 matrix of the pixles around the center
/// </summary>
/// <param name="row">Center Row</param>
/// <param name="column">Center Column</param>
/// <returns>3x3 matrix of the pixles around the center</returns>
private Color[,] Get3x3(int row, int column)
{
    Color[,] c = new Color[3, 3];
    c[0, 0] = this.GetPixel(column - 1, row - 1);
    c[0, 1] = this.GetPixel(column - 1, row);
    c[0, 2] = this.GetPixel(column - 1, row + 1);
    c[1, 0] = this.GetPixel(column, row - 1);
    c[1, 1] = this.GetPixel(column, row);
```

```
    c[1, 2] = this.GetPixel(column, row + 1);
    c[2, 0] = this.GetPixel(column + 1, row - 1);
    c[2, 1] = this.GetPixel(column + 1, row);
    c[2, 2] = this.GetPixel(column + 1, row + 1);
    return c;
}

private int[,] GetGrey3x3(int row, int column)
{
    int[,] c = new int[3, 3];
    c[0, 0] = this.GetGreyPixel(column - 1, row - 1);
    c[0, 1] = this.GetGreyPixel(column - 1, row);
    c[0, 2] = this.GetGreyPixel(column - 1, row + 1);
    c[1, 0] = this.GetGreyPixel(column, row - 1);
    c[1, 1] = this.GetGreyPixel(column, row);
    c[1, 2] = this.GetGreyPixel(column, row + 1);
    c[2, 0] = this.GetGreyPixel(column + 1, row - 1);
    c[2, 1] = this.GetGreyPixel(column + 1, row);
    c[2, 2] = this.GetGreyPixel(column + 1, row + 1);
    return c;
}

/// <summary>
/// Get at 5x5 matrix of the pixels around the center
/// </summary>
/// <param name="row">Center Row</param>
/// <param name="column">Center Column</param>
/// <returns>5x5 matrix of pixles around center</returns>
private Color[,] Get5x5(int row, int column)
{
    Color[,] c = new Color[5, 5];
    c[0, 0] = this.GetPixel(column - 2, row - 2);
    c[0, 1] = this.GetPixel(column - 2, row - 1);
    c[0, 2] = this.GetPixel(column - 2, row);
    c[0, 3] = this.GetPixel(column - 2, row + 1);
    c[0, 4] = this.GetPixel(column - 2, row + 2);
    c[1, 0] = this.GetPixel(column - 1, row - 2);
    c[1, 1] = this.GetPixel(column - 1, row - 1);
    c[1, 2] = this.GetPixel(column - 1, row);
    c[1, 3] = this.GetPixel(column - 1, row + 1);
    c[1, 4] = this.GetPixel(column - 1, row + 2);
    c[2, 0] = this.GetPixel(column, row - 2);
    c[2, 1] = this.GetPixel(column, row - 1);
    c[2, 2] = this.GetPixel(column, row);
    c[2, 3] = this.GetPixel(column, row + 1);
    c[2, 4] = this.GetPixel(column, row + 2);
    c[3, 0] = this.GetPixel(column + 1, row - 2);
    c[3, 1] = this.GetPixel(column + 1, row - 1);
    c[3, 2] = this.GetPixel(column + 1, row);
    c[3, 3] = this.GetPixel(column + 1, row + 1);
    c[3, 4] = this.GetPixel(column + 1, row + 2);
    c[4, 0] = this.GetPixel(column + 2, row - 2);
```

```
            c[4, 1] = this.GetPixel(column + 2, row - 1);
            c[4, 2] = this.GetPixel(column + 2, row);
            c[4, 3] = this.GetPixel(column + 2, row + 1);
            c[4, 4] = this.GetPixel(column + 2, row + 2);
            return c;
        }

        /// <summary>
        /// Perform laplace edge detection on the image
        /// </summary>
        /// <param name="b">Source Image</param>
        /// <returns>Edges</returns>
        public static Bitmap Laplace(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 1; column < i.Bitmap.Width - 1; column++)
            {
                for (int row = 1; row < i.Bitmap.Height - 1; row++)
                {
                    Color[,] c = i.Get3x3(row, column);

                    int red = (((c[0, 0].R + c[0, 1].R + c[0, 2].R + c[1, 0].R + c[1, 2].R +
c[2, 0].R
                        + c[2, 1].R + c[2, 2].R) * -1) + (c[1, 1].R * 8)) + 128;

                    int green = (((c[0, 0].G + c[0, 1].G + c[0, 2].G + c[1, 0].G + c[1, 2].G
                        + c[2, 0].G + c[2, 1].G + c[2, 2].G) * -1) + (c[1, 1].G * 8)) + 128;

                    int blue = (((c[0, 0].B + c[0, 1].B + c[0, 2].B + c[1, 0].B + c[1, 2].B +
c[2, 0].B
                        + c[2, 1].B + c[2, 2].B) * -1) + (c[1, 1].B * 8)) + 128;

                    if (red >= 128)
                        red = 0;
                    else
                        red = 255;

                    if (green >= 128)
                        green = 0;
                    else
                        green = 255;

                    if (blue >= 128)
                        blue = 0;
                    else
                        blue = 255;

                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
            }
            i.UnlockBitmap();
```

```
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="b"></param>
        /// <returns></returns>
        public static Bitmap LaplaceGreyscale(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 1; column < i.Bitmap.Width - 1; column++)
            {
                for (int row = 1; row < i.Bitmap.Height - 1; row++)
                {
                    int[,] c = i.GetGrey3x3(row, column);
                    int val = (((c[0, 0] + c[0, 1] + c[0, 2] + c[1, 0] + c[1, 2] + c[2, 0] +
c[2, 1]
                        + c[2, 2]) * -1) + (c[1, 1] * 8)) + 128;

                    if (val >= 128)
                        val = 0;
                    else
                        val = 255;

                    i2.SetPixel(column, row, Color.FromArgb(val, val, val));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Subtract b2 from b1 and normalize the image
        /// </summary>
        /// <param name="b1">Image</param>
        /// <param name="b2">Image</param>
        /// <returns>Normalized Image</returns>
        public static Bitmap Subtract(Bitmap b1, Bitmap b2)
        {
            if (b1.Width != b2.Width || b1.Height != b2.Height)
                throw new Exception("Images not the same size cannot subtract");

            ImagerBitmap i = new ImagerBitmap(b1.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b2.Clone() as Bitmap);
            int[,] red = new int[i.Bitmap.Width, i.Bitmap.Height];
            int[,] blue = new int[i.Bitmap.Width, i.Bitmap.Height];
            int[,] green = new int[i.Bitmap.Width, i.Bitmap.Height];
            int redMax = 0;
```

```
            int redMin = 0;
            int redRange = 0;
            int blueMax = 0;
            int blueMin = 0;
            int blueRange = 0;
            int greenMax = 0;
            int greenMin = 0;
            int greenRange = 0;

            //fill the arrays with the subtracted values
            //Keep track of the min and max values for later
            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    Color c1 = i.GetPixel(column, row);
                    Color c2 = i2.GetPixel(column, row);
                    red[column, row] = c2.R - c1.R;
                    blue[column, row] = c2.B - c1.B;
                    green[column, row] = c2.G - c1.G;
                    if (red[column, row] > redMax) redMax = red[column, row];
                    if (red[column, row] < redMin) redMin = red[column, row];
                    if (blue[column, row] > blueMax) blueMax = blue[column, row];
                    if (blue[column, row] < blueMin) blueMin = blue[column, row];
                    if (green[column, row] > greenMax) greenMax = green[column, row];
                    if (green[column, row] < greenMin) greenMin = green[column, row];
                }
            }
            //find the range of the min an max
            redRange = Math.Abs(redMax - redMin);
            blueRange = Math.Abs(blueMax - blueMin);
            greenRange = Math.Abs(greenRange - greenMin);
            //Normalize the values in the arrays and load the result image
            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    if (redRange != 0)
                        red[column, row] = 255 - (((redMax - red[column, row]) * 255) /
redRange);
                    if (blueRange != 0)
                        blue[column, row] = 255 - (((blueMax - blue[column, row]) * 255) /
blueRange);
                    if (greenRange != 0)
                        green[column, row] = 255 - (((greenMax - green[column, row]) * 255) /
greenRange);
                    if (red[column, row] < 0)
                        red[column, row] = 0;
                    if (blue[column, row] < 0)
                        blue[column, row] = 0;
                    if (green[column, row] < 0)
                        green[column, row] = 0;
```

```
                    i2.SetPixel(column, row, Color.FromArgb(red[column, row], green[column,
row],
                        blue[column, row]));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Get the red pixels from the bitmap
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap GetRedBitmap(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    i.SetPixel(column, row, Color.FromArgb(i.GetPixel(column, row).R, 0, 0));
                }
            }
            i.UnlockBitmap();
            return i.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Get the Blue Pixes from the bitmap
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap GetBlueBitmap(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    i.SetPixel(column, row, Color.FromArgb(0, 0, i.GetPixel(column, row).B));
                }
            }
            i.UnlockBitmap();
            return i.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Get the green pixels from the image
        /// </summary>
        /// <param name="b">Image to Process</param>
```

```
/// <returns>Filtered Image</returns>
public static Bitmap GetGreenBitmap(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            i.SetPixel(column, row, Color.FromArgb(0, i.GetPixel(column, row).G, 0));
        }
    }
    i.UnlockBitmap();
    return i.Bitmap.Clone() as Bitmap;
}

/// <summary>
/// Make the image Grey scale
/// </summary>
/// <param name="b">Image to Process</param>
/// <returns>Filtered Image</returns>
public static Bitmap GetGreyScaleBitmap(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            int val = i.GetGreyPixel(column, row);
            i.SetPixel(column, row, Color.FromArgb(val, val, val));
        }
    }
    i.UnlockBitmap();
    return i.Bitmap.Clone() as Bitmap;
}

/// <summary>
/// Make the image black and white
/// </summary>
/// <param name="b">Image to Process</param>
/// <returns>Filtered Image</returns>
public static Bitmap GetBlackAndWhiteBitmap(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    double[] histogram = new double[256];
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            histogram[i.GetGreyPixel(column, row)]++;
        }
    }
```

```
    //find the position of the max value on the left
    int leftK = 0;
    for (int k = 0; k < 128; k++)
    {
        if (histogram[k] > histogram[leftK]) leftK = k;
    }

    //find the position of the max value on the right
    int rightK = 0;
    for (int k = 128; k < 256; k++)
    {
        if (histogram[k] > histogram[rightK]) rightK = k;
    }

    //find the min value between the 2 local maxes
    int localMin = rightK;
    for (int k = leftK; k < rightK; k++)
    {
        if (histogram[k] < histogram[localMin]) localMin = k;
    }

    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            int val = (i.GetPixel(column, row).R + i.GetPixel(column, row).G
                + i.GetPixel(column, row).B) / 3;
            if (val > localMin)
                val = 255;
            else
                val = 0;
            i.SetPixel(column, row, Color.FromArgb(val, val, val));
        }
    }
    i.UnlockBitmap();
    return i.Bitmap.Clone() as Bitmap;
}

/// <summary>
/// Removes any pixes that are not within our threshold
/// </summary>
/// <param name="b">Bitmap to use</param>
/// <param name="c">Color to look for</param>
/// <param name="threshold">Value to use as the threshold</param>
/// <returns>Image without the background</returns>
public static Bitmap RemoveBackground(Bitmap b, Color c, int threshold)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    int maxGreen = c.G + threshold;
    int minGreen = c.G - threshold;
    int maxRed = c.R + threshold;
    int minRed = c.R - threshold;
```

```
    int maxBlue = c.B + threshold;
    int minBlue = c.B - threshold;

    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            Color px = i.GetPixel(column, row);

            if (px.G > minGreen && px.G < maxGreen
                && px.R > minRed && px.R < maxRed
                && px.B > minBlue && px.B < maxBlue)
            {
                //i.SetPixel(column, row, Color.FromArgb(255, 255, 255));
            }
            else
            {
                i.SetPixel(column, row, Color.Black);
            }
        }
    }
    i.UnlockBitmap();
    return i.Bitmap.Clone() as Bitmap;
}

/// <summary>
/// Find the Center of mass in an image
/// </summary>
/// <param name="b"></param>
/// <returns></returns>
public static Point CenterOfMass(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    int cogX = 0;
    int cogY = 0;
    int cogTotal = 0;

    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            int px = i.GetGreyPixel(column, row);
            cogX += (px * column);
            cogY += (px * row);
            cogTotal += px;
        }
    }
    i.UnlockBitmap();
    return new Point(cogX / cogTotal, cogY / cogTotal);
}

/// <summary>
```

```
        /// Create a bounding box around a part of the picture
        /// uses the upperLeft corner as the starting position of the search
        /// </summary>
        /// <param name="b">Bitmap to use</param>
        /// <param name="upperLeft">Upperleft corner of the box</param>
        /// <param name="lowerRight">LowerRight corner of the box</param>
        public static void BoundingBox(Bitmap b, ref Point upperLeft, ref Point lowerRight)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            Queue<Point> q = new Queue<Point>();
            q.Enqueue(upperLeft);

            int jump = 10;

            while (q.Count > 0)
            {
                Point p = q.Dequeue();
                AddToStack(q, b, i, new Point(p.X + jump, p.Y), ref upperLeft, ref lowerRight);
                AddToStack(q, b, i, new Point(p.X - jump, p.Y), ref upperLeft, ref lowerRight);
                AddToStack(q, b, i, new Point(p.X, p.Y + jump), ref upperLeft, ref lowerRight);
                AddToStack(q, b, i, new Point(p.X, p.Y - jump), ref upperLeft, ref lowerRight);
            }
            i.UnlockBitmap();
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="b"></param>
        /// <param name="c"></param>
        /// <param name="threshold"></param>
        /// <param name="cog"></param>
        /// <param name="upperLeft"></param>
        /// <param name="lowerRight"></param>
        public static void FindTheBall(Bitmap b, Color c, int threshold, ref Point cog, ref
Point upperLeft, ref Point lowerRight)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            int maxGreen = c.G + threshold;
            int minGreen = c.G - threshold;
            int maxRed = c.R + threshold;
            int minRed = c.R - threshold;
            int maxBlue = c.B + threshold;
            int minBlue = c.B - threshold;
            int cogX = 0;
            int cogY = 0;
            int cogTotal = 0;

            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
```

```
            Color px = i.GetPixel(column, row);

            if (px.G > minGreen && px.G < maxGreen
                && px.R > minRed && px.R < maxRed
                && px.B > minBlue && px.B < maxBlue)
            {
                int px2 = i.GetGreyPixel(column, row);
                cogX += (px2 * column);
                cogY += (px2 * row);
                cogTotal += px2;
            }
            else
            {
                i.SetPixel(column, row, Color.Black);
            }
        }
    }

    if (cogTotal > 0)
    {
        cog = new Point(cogX / cogTotal, cogY / cogTotal);
    }
    else
    {
        cog = new Point(0, 0);
    }

    upperLeft = new Point(cog.X, cog.Y);
    lowerRight = new Point(cog.X, cog.Y);

    Queue<Point> q = new Queue<Point>();
    q.Enqueue(upperLeft);

    int jump = 10;

    while (q.Count > 0)
    {
        Point p = q.Dequeue();
        AddToStack(q, b, i, new Point(p.X + jump, p.Y), ref upperLeft, ref lowerRight);
        AddToStack(q, b, i, new Point(p.X - jump, p.Y), ref upperLeft, ref lowerRight);
        AddToStack(q, b, i, new Point(p.X, p.Y + jump), ref upperLeft, ref lowerRight);
        AddToStack(q, b, i, new Point(p.X, p.Y - jump), ref upperLeft, ref lowerRight);
    }

    i.UnlockBitmap();
}

/// <summary>
/// Helper Method for the Bounding Box
/// if a point is in the image and it has some color
/// then resize the bounding box and add the pixel to the queue to search
/// </summary>
```

```
        /// <param name="q"></param>
        /// <param name="b"></param>
        /// <param name="i"></param>
        /// <param name="p"></param>
        /// <param name="upperLeft"></param>
        /// <param name="lowerRight"></param>
        private static void AddToStack(Queue<Point> q, Bitmap b, ImagerBitmap i, Point p, ref
Point upperLeft, ref Point lowerRight)
        {
            if (p.X < b.Width
                && p.Y < b.Height
                && i.GetGreyPixel(p.X, p.Y) != 0)
            {
                if (upperLeft.X > p.X)
                {
                    upperLeft.X = p.X;
                    q.Enqueue(p);
                }
                if (upperLeft.Y > p.Y)
                {
                    upperLeft.Y = p.Y;
                    q.Enqueue(p);
                }
                if (lowerRight.X < p.X)
                {
                    lowerRight.X = p.X;
                    q.Enqueue(p);
                }
                if (lowerRight.Y < p.Y)
                {
                    lowerRight.Y = p.Y;
                    q.Enqueue(p);
                }
            }
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="b"></param>
        /// <returns></returns>
        public static Bitmap GetBlackAndWhiteBitmap2(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            double[] histogram = new double[256];
            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    histogram[i.GetGreyPixel(column, row)]++;
                }
            }
```

```
            double k = b.Width * b.Height;
            double half = k / 2;
            int middle = 0;
            while (k > half)
            {
                k = k - histogram[middle];
                middle++;
            }

            for (int column = 0; column < i.Bitmap.Width; column++)
            {
                for (int row = 0; row < i.Bitmap.Height; row++)
                {
                    int val = (i.GetPixel(column, row).R + i.GetPixel(column, row).G
                        + i.GetPixel(column, row).B) / 3;
                    if (val > middle)
                        val = 255;
                    else
                        val = 0;
                    i.SetPixel(column, row, Color.FromArgb(val, val, val));
                }
            }
            i.UnlockBitmap();
            return i.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Box filter using a 3x3 Mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Box3x3(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 1; column < i.Bitmap.Width - 1; column++)
            {
                for (int row = 1; row < i.Bitmap.Height - 1; row++)
                {
                    Color[,] c = i.Get3x3(row, column);
                    int red = (c[0, 0].R + c[0, 1].R + c[0, 2].R + c[1, 0].R + c[1, 2].R + c[2,
0].R
                        + c[2, 1].R + c[2, 2].R + c[1, 1].R) / 9;
                    int green = (c[0, 0].G + c[0, 1].G + c[0, 2].G + c[1, 0].G + c[1, 2].G +
c[2, 0].G
                        + c[2, 1].G + c[2, 2].G + c[1, 1].G) / 9;
                    int blue = (c[0, 0].B + c[0, 1].B + c[0, 2].B + c[1, 0].B + c[1, 2].B + c[2,
0].B
                        + c[2, 1].B + c[2, 2].B + c[1, 1].B) / 9;
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
```

```
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Bionomial filter using a 3x3 Mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Binomial3x3(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 1; column < i.Bitmap.Width - 1; column++)
            {
                for (int row = 1; row < i.Bitmap.Height - 1; row++)
                {
                    Color[,] c = i.Get3x3(row, column);
                    int red = ((c[0, 0].R * 1) + (c[0, 1].R * 2) + (c[0, 2].R * 1) + (c[1, 0].R
* 2)
                        + (c[1, 1].R * 4) + (c[1, 2].R * 2) + (c[2, 0].R * 1) + (c[2, 1].R * 2)
                        + (c[2, 2].R * 1)) / 16;
                    int green = ((c[0, 0].G * 1) + (c[0, 1].G * 2) + (c[0, 2].G * 1) + (c[1,
0].G * 2)
                        + (c[1, 1].G * 4) + (c[1, 2].G * 2) + (c[2, 0].G * 1) + (c[2, 1].G * 2)
                        + (c[2, 2].G * 1)) / 16;
                    int blue = ((c[0, 0].B * 1) + (c[0, 1].B * 2) + (c[0, 2].B * 1) + (c[1, 0].B
* 2)
                        + (c[1, 1].B * 4) + (c[1, 2].B * 2) + (c[2, 0].B * 1) + (c[2, 1].B * 2)
                        + (c[2, 2].B * 1)) / 16;
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Median filter using a 3x3 mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Median3x3(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 1; column < i.Bitmap.Width - 1; column++)
            {
                for (int row = 1; row < i.Bitmap.Height - 1; row++)
```

```
                {
                    Color[,] c = i.Get3x3(row, column);
                    int red = Median(c[0, 0].R, c[0, 1].R, c[0, 2].R, c[1, 0].R, c[1, 1].R, c[1,
2].R,
                        c[2, 0].R, c[2, 1].R, c[2, 2].R);
                    int green = Median(c[0, 0].G, c[0, 1].G, c[0, 2].G, c[1, 0].G, c[1, 1].G,
                        c[1, 2].G, c[2, 0].G, c[2, 1].G, c[2, 2].G);
                    int blue = Median(c[0, 0].B, c[0, 1].B, c[0, 2].B, c[1, 0].B, c[1, 1].B,
c[1, 2].B,
                        c[2, 0].B, c[2, 1].B, c[2, 2].B);
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Box filter using a 5x5 Mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Box5x5(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 2; column < i.Bitmap.Width - 2; column++)
            {
                for (int row = 2; row < i.Bitmap.Height - 2; row++)
                {
                    Color[,] c = i.Get5x5(row, column);
                    int red = (c[0, 0].R + c[0, 1].R + c[0, 2].R + c[0, 3].R + c[0, 4].R + c[1,
0].R
                        + c[1, 1].R + c[1, 2].R + c[1, 3].R + c[1, 4].R + c[2, 0].R + c[2, 1].R
                        + c[2, 2].R + c[2, 3].R + c[2, 4].R + c[3, 0].R + c[3, 1].R + c[3, 2].R
                        + c[3, 3].R + c[3, 4].R + c[4, 0].R + c[4, 1].R + c[4, 2].R + c[4, 3].R
                        + c[4, 4].R) / 25;
                    int green = (c[0, 0].G + c[0, 1].G + c[0, 2].G + c[0, 3].G + c[0, 4].G +
c[1, 0].G
                        + c[1, 1].G + c[1, 2].G + c[1, 3].G + c[1, 4].G + c[2, 0].G + c[2, 1].G
                        + c[2, 2].G + c[2, 3].G + c[2, 4].G + c[3, 0].G + c[3, 1].G + c[3, 2].G
                        + c[3, 3].G + c[3, 4].G + c[4, 0].G + c[4, 1].G + c[4, 2].G + c[4, 3].G
                        + c[4, 4].G) / 25;
                    int blue = (c[0, 0].B + c[0, 1].B + c[0, 2].B + c[0, 3].B + c[0, 4].B + c[1,
0].B
                        + c[1, 1].B + c[1, 2].B + c[1, 3].B + c[1, 4].B + c[2, 0].B + c[2, 1].B
                        + c[2, 2].B + c[2, 3].B + c[2, 4].B + c[3, 0].B + c[3, 1].B + c[3, 2].B
                        + c[3, 3].B + c[3, 4].B + c[4, 0].B + c[4, 1].B + c[4, 2].B + c[4, 3].B
                        + c[4, 4].B) / 25;
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
```

```
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Binomial filter using a 5x5 Mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Binomial5x5(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 2; column < i.Bitmap.Width - 2; column++)
            {
                for (int row = 2; row < i.Bitmap.Height - 2; row++)
                {
                    Color[,] c = i.Get5x5(row, column);
                    int red = ((c[0, 0].R * 1) + (c[0, 1].R * 4) + (c[0, 2].R * 6) + (c[0, 3].R
* 4)
                        + (c[0, 4].R * 1) + (c[1, 0].R * 4) + (c[1, 1].R * 16) + (c[1, 2].R *
24)
                        + (c[1, 3].R * 16) + (c[1, 4].R * 4) + (c[2, 0].R * 6) + (c[2, 1].R *
24)
                        + (c[2, 2].R * 36) + (c[2, 3].R * 24) + (c[2, 4].R * 6) + (c[3, 0].R *
4)
                        + (c[3, 1].R * 16) + (c[3, 2].R * 24) + (c[3, 3].R * 16) + (c[3, 4].R *
4)
                        + (c[4, 0].R * 1) + (c[4, 1].R * 4) + (c[4, 2].R * 6) + (c[4, 3].R * 4)
                        + (c[4, 4].R * 1)) / 256;
                    int green = ((c[0, 0].G * 1) + (c[0, 1].G * 4) + (c[0, 2].G * 6) + (c[0,
3].G * 4)
                        + (c[0, 4].G * 1) + (c[1, 0].G * 4) + (c[1, 1].G * 16) + (c[1, 2].G *
24)
                        + (c[1, 3].G * 16) + (c[1, 4].G * 4) + (c[2, 0].G * 6) + (c[2, 1].G *
24)
                        + (c[2, 2].G * 36) + (c[2, 3].G * 24) + (c[2, 4].G * 6) + (c[3, 0].G *
4)
                        + (c[3, 1].G * 16) + (c[3, 2].G * 24) + (c[3, 3].G * 16) + (c[3, 4].G *
4)
                        + (c[4, 0].G * 1) + (c[4, 1].G * 4) + (c[4, 2].G * 6) + (c[4, 3].G * 4)
                        + (c[4, 4].G * 1)) / 256;
                    int blue = ((c[0, 0].B * 1) + (c[0, 1].B * 4) + (c[0, 2].B * 6) + (c[0, 3].B
* 4)
                        + (c[0, 4].B * 1) + (c[1, 0].B * 4) + (c[1, 1].B * 16) + (c[1, 2].B *
24)
                        + (c[1, 3].B * 16) + (c[1, 4].B * 4) + (c[2, 0].B * 6) + (c[2, 1].B *
24)
                        + (c[2, 2].B * 36) + (c[2, 3].B * 24) + (c[2, 4].B * 6) + (c[3, 0].B *
4)
```

```
                        + (c[3, 1].B * 16) + (c[3, 2].B * 24) + (c[3, 3].B * 16) + (c[3, 4].B *
4)
                        + (c[4, 0].B * 1) + (c[4, 1].B * 4) + (c[4, 2].B * 6) + (c[4, 3].B * 4)
                        + (c[4, 4].B * 1)) / 256;
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
            return i2.Bitmap.Clone() as Bitmap;
        }

        /// <summary>
        /// Perform a Median fiter using a 5x5 Mask
        /// </summary>
        /// <param name="b">Image to Process</param>
        /// <returns>Filtered Image</returns>
        public static Bitmap Median5x5(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            ImagerBitmap i2 = new ImagerBitmap(b.Clone() as Bitmap);
            for (int column = 2; column < i.Bitmap.Width - 2; column++)
            {
                for (int row = 2; row < i.Bitmap.Height - 2; row++)
                {
                    Color[,] c = i.Get5x5(row, column);
                    int red = Median(c[0, 0].R, c[0, 1].R, c[0, 2].R, c[0, 3].R, c[0, 4].R, c[1,
0].R,
                        c[1, 1].R, c[1, 2].R, c[1, 3].R, c[1, 4].R, c[2, 0].R, c[2, 1].R, c[2,
2].R,
                        c[2, 3].R, c[2, 4].R, c[3, 0].R, c[3, 1].R, c[3, 2].R, c[3, 3].R, c[3,
4].R,
                        c[4, 0].R, c[4, 1].R, c[4, 2].R, c[4, 3].R, c[4, 4].R);
                    int green = Median(c[0, 0].G, c[0, 1].G, c[0, 2].G, c[0, 3].G, c[0, 4].G,
c[1, 0].G,
                        c[1, 1].G, c[1, 2].G, c[1, 3].G, c[1, 4].G, c[2, 0].G, c[2, 1].G, c[2,
2].G,
                        c[2, 3].G, c[2, 4].G, c[3, 0].G, c[3, 1].G, c[3, 2].G, c[3, 3].G, c[3,
4].G,
                        c[4, 0].G, c[4, 1].G, c[4, 2].G, c[4, 3].G, c[4, 4].G);
                    int blue = Median(c[0, 0].B, c[0, 1].B, c[0, 2].B, c[0, 3].B, c[0, 4].B,
c[1, 0].B,
                        c[1, 1].B, c[1, 2].B, c[1, 3].B, c[1, 4].B, c[2, 0].B, c[2, 1].B, c[2,
2].B,
                        c[2, 3].B, c[2, 4].B, c[3, 0].B, c[3, 1].B, c[3, 2].B, c[3, 3].B, c[3,
4].B,
                        c[4, 0].B, c[4, 1].B, c[4, 2].B, c[4, 3].B, c[4, 4].B);
                    i2.SetPixel(column, row, Color.FromArgb(red, green, blue));
                }
            }
            i.UnlockBitmap();
            i2.UnlockBitmap();
```

```
    return i2.Bitmap.Clone() as Bitmap;
}

/// <summary>
/// Find the Median value in an array of int's
/// </summary>
/// <param name="values">Array of values</param>
/// <returns>Median value</returns>
private static int Median(params int[] values)
{
    Array.Sort(values);
    return values[((values.Length - 1) / 2) + 1];
}

/// <summary>
/// Resize an image
/// </summary>
/// <param name="b">Image to resize</param>
/// <param name="width">Width in pixels</param>
/// <param name="height">Height in pixels</param>
/// <returns>New Image</returns>
public static Bitmap Resize(Bitmap b, int width, int height)
{
    Bitmap b2 = new Bitmap(width, height, PixelFormat.Format24bppRgb);
    Graphics g = Graphics.FromImage(b2);
    g.DrawImage(b, new Rectangle(0, 0, width, height), 0, 0, b.Width, b.Height,
        GraphicsUnit.Pixel);
    g.Dispose();
    return b2;
}

/// <summary>
/// Generate a greyscale histogram chart from the bitmap
/// </summary>        /// <param name="b">Image to use</param>
/// <returns>Histogram chart</returns>
public static Bitmap GenerateGreyscaleHistogram(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    double[] histogram = new double[256];
    string[] lbls = new string[256];
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            histogram[i.GetGreyPixel(column, row)]++;
        }
    }
    i.UnlockBitmap();
    for (int j = 0; j < 256; j++)
    {
        if ((j % 30) == 0)
        {
```

```
            lbls[j] = j.ToString();
        }
        else
        {
            lbls[j] = string.Empty;
        }
    }

    //XYChart c = new XYChart(b.Width + 60, b.Height + 60);
    //c.setPlotArea(40, 40, b.Width - 20, b.Height - 20);
    //c.addTitle("Greyscale Historgram", "Arial Bold", 10).setBackground(
    //    Chart.metalColor(0x9999ff), -1, 1);
    //c.addBarLayer(histogram);
    //c.xAxis().setLabels(lbls);
    //return c.makeImage() as Bitmap;
    return null;
}

/// <summary>
/// Generate a histogram chart from the bitmap
/// </summary>
/// <param name="b">Image to use</param>
/// <returns>Histogram chart</returns>
public static Bitmap GenerateHistogram(Bitmap b)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    double[] histogram = new double[64];
    string[] lbls = new string[64];
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        for (int row = 0; row < i.Bitmap.Height; row++)
        {
            histogram[i.GetRGBHistogramValue(column, row)]++;
        }
    }
    i.UnlockBitmap();
    for (int j = 0; j < 64; j++)
    {
        if ((j % 10) == 0)
        {
            lbls[j] = j.ToString();
        }
        else
        {
            lbls[j] = string.Empty;
        }
    }
    //XYChart c = new XYChart(b.Width + 60, b.Height + 60);
    //c.setPlotArea(40, 40, b.Width - 20, b.Height - 20);
    //c.addTitle("Color Historgram", "Arial Bold", 10).setBackground(
    //    Chart.metalColor(0x9999ff), -1, 1);
    //c.addBarLayer(histogram);
```

```
    //c.xAxis().setLabels(lbls);
    //return c.makeImage() as Bitmap;
    return null;
}

/// <summary>
/// Generate a Chart of a Scan line
/// </summary>
/// <param name="b">Image to use</param>
/// <param name="row">Row to use</param>
/// <returns>Scan Line Chart</returns>
public static Bitmap GenerateScanLineChart(Bitmap b, int row)
{
    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
    double[] luminance = new double[b.Width];
    string[] lbls = new string[b.Width];
    double max = double.MinValue;
    double min = double.MaxValue;
    for (int column = 0; column < i.Bitmap.Width; column++)
    {
        luminance[column] = i.GetGreyPixel(column, row);
        if (luminance[column] > max) max = luminance[column];
        if (luminance[column] < min) min = luminance[column];
        if ((column % 40) == 0)
        {
            lbls[column] = column.ToString();
        }
        else
        {
            lbls[column] = string.Empty;
        }
    }
    i.UnlockBitmap();
    //XYChart c = new XYChart(b.Width + 60, b.Height + 60);
    //c.setPlotArea(40, 40, b.Width - 20, b.Height - 20);
    //c.addTitle(string.Format("Pixel Luminance for Scan Line {0}, STF {1:0.00}",
    //    row.ToString(),
    //    (max - min) / (max + min)), "Arial Bold", 10).setBackground(
    //    Chart.metalColor(0x9999ff), -1, 1);
    //c.addLineLayer(luminance);
    //c.xAxis().setLabels(lbls);
    //return c.makeImage() as Bitmap;
    return null;
}

///// <summary>
///// Generate a Chart of a Vertical Scan line
///// </summary>
///// <param name="b">Image to use</param>
///// <param name="column">Column to use</param>
///// <returns>Scan Line Chart</returns>
//public static Bitmap GenerateVerticalScanLineChart(Bitmap b, int col)
```

```
        //{
        //    ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
        //    double[] luminance = new double[b.Height];
        //    string[] lbls = new string[b.Height];
        //    for (int row = 0; row < i.Bitmap.Height; row++)
        //    {
        //        luminance[row] = i.GetGreyPixel(col, row);
        //        if ((row % 40) == 0)
        //        {
        //            lbls[row] = row.ToString();
        //        }
        //        else
        //        {
        //            lbls[row] = string.Empty;
        //        }
        //    }
        //    i.UnlockBitmap();
        //    //XYChart c = new XYChart(b.Width + 60, b.Height + 60);
        //    //c.setPlotArea(40, 40, b.Width - 20, b.Height - 20);
        //    //c.addTitle("Pixel Luminance for Vertical Scan Line " + col.ToString(),
        //    //    "Arial Bold", 10).setBackground(
        //    //    Chart.metalColor(0x9999ff), -1, 1);
        //    //c.addLineLayer(luminance);
        //    //c.xAxis().setLabels(lbls);
        //    //return c.makeImage() as Bitmap;
        //    return null;
        //}

        /// <summary>
        ///
        /// </summary>
        /// <param name="b"></param>
        public static void GenerateExcelFile(Bitmap b)
        {
            ImagerBitmap i = new ImagerBitmap(b.Clone() as Bitmap);
            string fileName = "output.txt";
            int f = 0;
            using (StreamWriter sw = new StreamWriter(fileName, false))
            {
                sw.WriteLine("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}", "Index", "Row", "Column",
"Red",
                    "Green", "Blue", "Gray");
                for (int column = 0; column < i.Bitmap.Width; column++)
                {
                    for (int row = 0; row < i.Bitmap.Height; row++)
                    {
                        f++;
                        Color c = i.GetPixel(column, row);
                        int g = i.GetGreyPixel(column, row);
                        sw.WriteLine("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}", f,
                            row, column, c.R, c.G, c.B, g);
                    }
```

```
                }
            }
            i.UnlockBitmap();
            System.Diagnostics.Process.Start(fileName);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Microsoft.Robotics.Services.IRobot.Roomba
{
    /// <summary>
    ///
    /// </summary>
    public partial class CameraForm : Form
    {
        /// <summary>
        /// constructor
        /// </summary>
        public CameraForm()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Get a still from the video stream
        /// </summary>
        /// <returns></returns>
        public Image GetSnapshot()
        {
            Bitmap bmp = new Bitmap(wbMain.Bounds.Width, wbMain.Bounds.Height);
            wbMain.BringToFront();
            wbMain.DrawToBitmap(bmp, wbMain.Bounds);
            Bitmap bmp2 = new Bitmap(320, 240,
System.Drawing.Imaging.PixelFormat.Format24bppRgb);
            using (Graphics g = Graphics.FromImage(bmp2))
            {
                g.DrawImage(bmp,
                    new Rectangle(0, 0, 320, 240),
                    new Rectangle(122, 62, 320, 240),
                    GraphicsUnit.Pixel);
            }
            return bmp2;
        }

        /// <summary>
        /// setup the form
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void CameraForm_Load(object sender, EventArgs e)
        {
            wbMain.Url = new Uri("http://192.168.1.253/CgiStart?page=Single&amp;Language=0");
```

```
        }
    }
}
```