Ruby-esque Pseudo-Code for
Convex Hull Divide-and-Conquer Algorithm

```
def solve(points)
    points = recursive_solve(points)
    (points.size / 2).times do
        points = points.convexify
    end
    return points
end



def recursive_solve(points)
    if points.size <= 2
        return points
    else
        left, right = points.divide_in_half
        return combine(recursive_solve(left), recursive_solve(right))
    end
end



def combine(left, right)
    merged = left.merge(right)
    clockwise = merged.sort_clockwise
    return clockwise.convexify
end



def convexify(points)
    if points.size > 3
        centroid = points.get_centroid
        # pseudo code:
        # 1. loop through each point in 'points'; let each be called 'current'
        # 2. form a triangle from the centroid to
        #    the point before and the point after 'current'
        # 3. if 'current' falls inside the triangle, remove it from 'points'
    end
    return points
end
```

Analysis of Computational Efficiency

1. convexify and sort_clockwise are situated in the inner-most loop.
    convexify: order *n* operation
    sort_clockwise: order *n log n* operation (merge sort)

2. combine is the next outer loop.
    Merge: order *n* operation
    + time to sort_clockwise
    + time to convexify

3. recursive_solve is the next outer loop
    This is the divide-and-conquer portion of the algorithm
    Divide: order *n* operation
    + time to combine

4. solve adds precision to the recursively created solution by iteratively removing unnecessary inner points (using convexify).
    + time to recursive_solve
    + convexify * (m / 2) where m is the number of points in the outer rim after applying recursive_solve

$$T(n) = 2T(n/2) + T_{combine}(n) \qquad T_{combine}(n) = 2n + n \cdot \log(n)$$

$$T(n) = 2T(n/2) + 2n + n \cdot \log(n)$$

$$T(n) = 2T(n/2) + O(n^2)^*$$

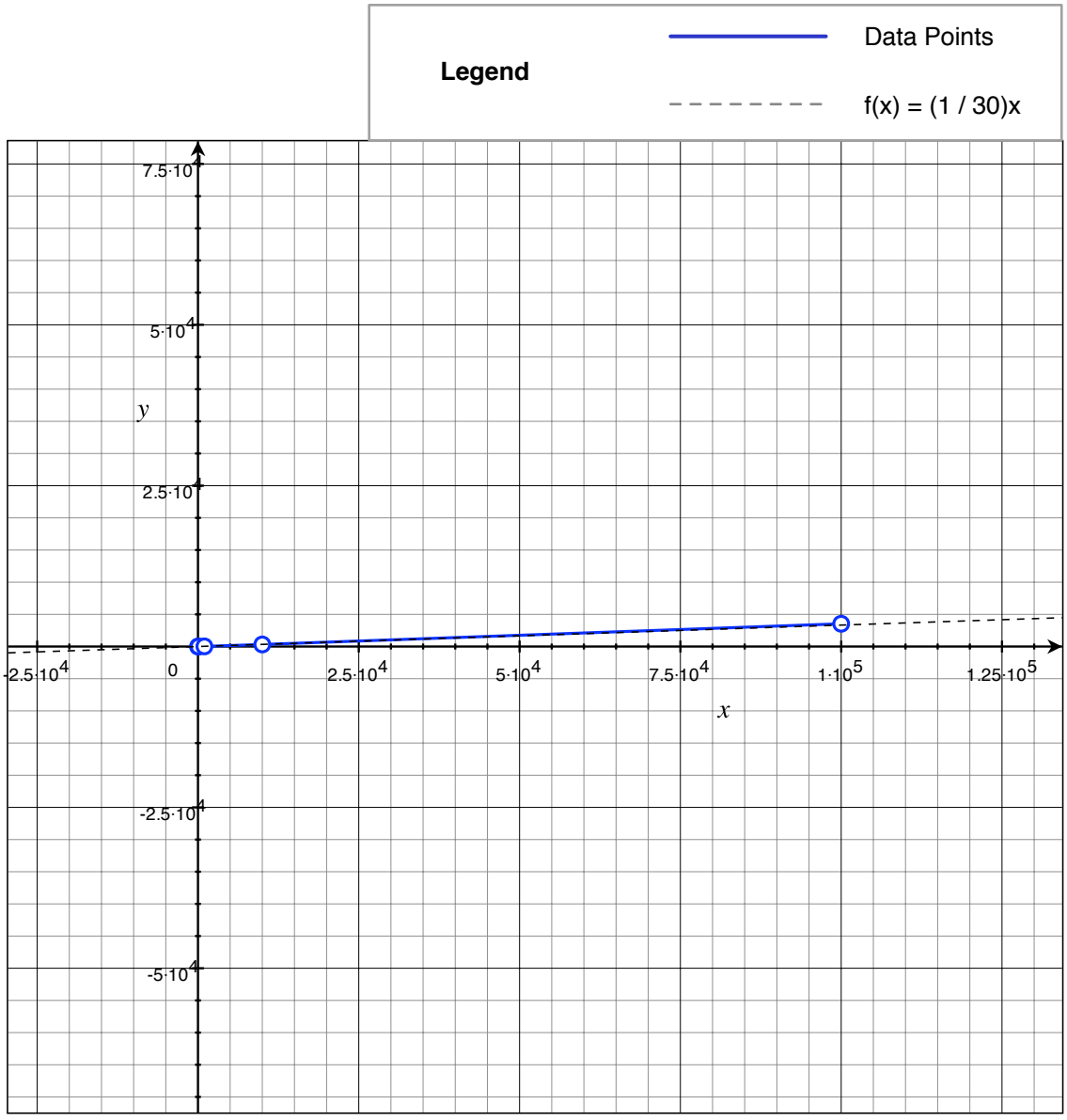
So, by the master theorem:

$$T(n) = O(n^2)$$

* Actually, we can do better than that, but the master theorem doesn't work well without a power in the 2nd function (**n** raised to some power of **d**)... so T(n) may actually be bound by O(n log(n)), or something in between.

Emperical Analysis

n	Time (ms)	Time (ms)	Time (ms)
10	0.1	0.2	0.2
100	1.8	1.9	1.8
1,000	21.1	29.5	21.2
10,000	320.2	728.6	368.3
100,000	3,537.1	3,752.2	3,235.8
1,000,000	bad data*	bad data*	bad data*



* Test was completed on a machine with only 256 MB RAM. Disk swapping (virtual memory) became a significant time factor at 1M points.

Comparison of Theoretical and Emperical Analyses

My emperical analysis seems to suggest this is a linear-order growth problem; however, my theoretical analysis places it at O(n^2), or possibly somewhere between O(n log(n)) and O (n^2). So there is a discrepancy between the theory and the practical tests.

It may be that if I could increase the number of points in my test beyond 100,000 I would start to see an increase matching one of the possibilities above. In fact, it seems very unlikely that this is an order n problem, so if I were a betting man, I would place my chips in the "at least n log(n)" bin. But alas, this is an analysis and betting is not allowed. So I will point out that the theoretical big-O bound of n^2 does indeed bound the seemingly linear order output.

I used a graphing program ("Grapher" on Mac OS X) to find the constant of proportionality, i.e. the fraction 1/30, as noted on the previous page.

Guide to the Source Code

This page (the one you are reading) is PDF **page 4**.

All of the helper classes such as ColoredPoint and PointList (a list of ColoredPoints) are contained in PDF **pages 5-11**.

The ConvexHullSolver class starts on PDF **page 12** of the source code output and is the most relevant portion of this paper.

I made several modifications to the source code package that was given to us, as I wanted a more hands-on GUI that would let me experiment with groups of Hull objects etc. Therefore, I've included the source code output of FormMain starting on PDF **page 13**.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Text;

namespace ConvexHull
{
    class ColoredPoint
    {
        bool highlighted;
        Pen pen;
        Size size;
        float width;
        public float X, Y;

        public ColoredPoint(ColoredPoint cp)
        {
            init(cp.X, cp.Y, cp.getPen(), cp.getSize(), cp.getWidth(), cp.isHighlighted
    ());
        }

        public ColoredPoint(float x, float y)
        {
            init(x, y, Color.Black);
        }

        public ColoredPoint(float x, float y, Color color)
        {
            init(x, y, color);
        }

        private void init(float x, float y, Color color)
        {
            init(x, y, new Pen(color, width), new Size(4, 4), 1.0F, false);
        }

        private void init(float x, float y, Pen p, Size s, float w, bool high)
        {
            width = w;
            X = x; Y = y;
            pen = p;
            size = s;
            highlighted = high;
        }

        public bool contains(Point p)
        {
            return getSurroundingRect().Contains(p);
        }

        public Rectangle getRect()
        {
            return getRectWithPadding(0);
        }

        public Rectangle getSurroundingRect()
        {
            return getRectWithPadding(4);
        }

        public Rectangle getRectWithPadding(int padding)
        {
            Size s = new Size(size.Width, size.Height);
            s.Width += padding * 2;
            s.Height += padding * 2;
            PointF p = getPointF();
```

```
            p.X -= s.Width / 2;
            p.Y -= s.Height / 2;
            return new Rectangle(Point.Round(p), s);
        }

        public Point getPoint()
        {
            return new Point((int)X, (int)Y);
        }

        public PointF getPointF()
        {
            return new PointF(X, Y);
        }

        public Pen getPen()
        {
            return pen;
        }

        public Color getColor()
        {
            return pen.Color;
        }

        public Size getSize()
        {
            return size;
        }

        public float getWidth()
        {
            return width;
        }

        public bool isHighlighted()
        {
            return highlighted;
        }

        public void highlight()
        {
            highlighted = true;
            width = 2.0F;
            pen = new Pen(Color.Red, width);
        }

        public void unhighlight()
        {
            highlighted = false;
            width = 1.0F;
            pen = new Pen(Color.Black, width);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;

namespace ConvexHull
{
    class DividedList
    {
        public PointList left;
        public PointList right;

        public DividedList(PointList l, PointList r)
        {
            left = l; right = r;
        }
    }

    class PointList : List<ColoredPoint>
    {
        static PointF compareCenter = new PointF();

        public PointList() : base() {}

        public PointList(PointList p) : base(p) {}

        public PointList(List<ColoredPoint> p) : base(p) { }

        public PointList copy()
        {
            PointList pl = new PointList();
            foreach (ColoredPoint pc in this) pl.Add(new ColoredPoint(pc));
            return pl;
        }

        public int topPointIndex()
        {
            int topIndex = -1;
            for (int i = 0; i < Count; i++)
                if (topIndex == -1 || this[i].Y < this[topIndex].Y) topIndex = i;
            return topIndex;
        }
        public ColoredPoint topPoint() { return this[topPointIndex()]; }

        public int bottomPointIndex()
        {
            int bottomIndex = -1;
            for (int i = 0; i < Count; i++)
                if (bottomIndex == -1 || this[i].Y > this[bottomIndex].Y) bottomIndex = i↙
    ;
            return bottomIndex;
        }
        public ColoredPoint bottomPoint() { return this[bottomPointIndex()]; }

        public int leftPointIndex()
        {
            int leftIndex = -1;
            for (int i = 0; i < Count; i++)
                if (leftIndex == -1 || this[i].X < this[leftIndex].X) leftIndex = i;
            return leftIndex;
        }
        public ColoredPoint leftPoint() { return this[leftPointIndex()]; }

        public int rightPointIndex()
        {
            int rightIndex = -1;
```

```
            for (int i = 0; i < Count; i++)
                if (rightIndex == -1 || this[i].X > this[rightIndex].X) rightIndex = i;
            return rightIndex;
        }
        public ColoredPoint rightPoint() { return this[rightPointIndex()]; }

        public DividedList divide()
        {
            PointF p = getCentroid();
            Predicate<ColoredPoint> left = delegate(ColoredPoint cp) { return cp.X < p.X; ↙
 };
            Predicate<ColoredPoint> right = delegate(ColoredPoint cp) { return cp.X >= p. ↙
X; };
            return new DividedList(
                new PointList(FindAll(left)),
                new PointList(FindAll(right)));
        }

        public PointList highlight()
        {
            foreach (ColoredPoint cp in this) cp.highlight();
            return this;
        }

        public PointList unhighlight()
        {
            foreach (ColoredPoint cp in this) cp.unhighlight();
            return this;
        }

        public PointF getCentroid()
        {
            float x = 0.0F, y = 0.0F;
            foreach (ColoredPoint cp in this)
            {
                x += cp.X;
                y += cp.Y;
            }
            return new PointF(x / Count, y / Count);
        }

        public PointList highlightedList()
        {
            Predicate<ColoredPoint> highlighted = delegate(ColoredPoint cp) { return cp. ↙
isHighlighted(); };
            return new PointList(FindAll(highlighted));
        }

        public PointList clockWiseList()
        {
            PointList orderedList = new PointList(this);
            compareCenter = orderedList.getCentroid();
            orderedList.Sort(compareClockWise);
            return orderedList;
        }

        public PointList counterClockWiseList()
        {
            PointList orderedList = new PointList(this);
            compareCenter = orderedList.getCentroid();
            orderedList.Sort(compareCounterClockWise);
            return orderedList;
        }

        public PointList selectTopHalf()
        {
```

```
            PointList topHalf = new PointList();
            int leftMost = leftPointIndex(), rightMost = rightPointIndex();
            float bottom = Math.Max(this[leftMost].Y, this[rightMost].Y);
            foreach (ColoredPoint cp in this)
                if (cp.Y <= bottom) topHalf.Add(cp);
            return topHalf;
        }

        public PointList selectBotHalf()
        {
            PointList bottomHalf = new PointList();
            int leftMost = leftPointIndex(), rightMost = rightPointIndex();
            float top = Math.Min(this[leftMost].Y, this[rightMost].Y);
            foreach (ColoredPoint cp in this)
                if (cp.Y >= top) bottomHalf.Add(cp);
            return bottomHalf;
        }

        private static int compareClockWise(ColoredPoint p1, ColoredPoint p2)
        {
            double x1 = (double)(p1.X - compareCenter.X);
            double y1 = (double)(p1.Y - compareCenter.Y);
            double r1 = Math.Atan2(y1, x1);

            double x2 = (double)(p2.X - compareCenter.X);
            double y2 = (double)(p2.Y - compareCenter.Y);
            double r2 = Math.Atan2(y2, x2);

            if (r1 == r2) return 0;
            if (r1 > r2) return 1;
            return -1;
        }

        private static int compareCounterClockWise(ColoredPoint p1, ColoredPoint p2)
        {
            return compareClockWise(p2, p1);
        }

        public void drawPoints(Graphics g)
        {
            foreach (ColoredPoint cp in this)
                g.DrawRectangle(cp.getPen(), cp.getRect());
        }

        public void drawConnections(Graphics g, bool primaryConnections)
        {
            PointList orderedList = clockWiseList();
            ColoredPoint prev = null;
            Pen pen = primaryConnections ? new Pen(Color.Gray, 2F) : new Pen(Color.Blue, ↙
2F);
            foreach (ColoredPoint cp in orderedList)
            {
                if (prev != null)
                {
                    g.DrawLine(pen, prev.getPoint(), cp.getPoint());
                }
                prev = cp;
            }
            // Connect head to tail
            if (orderedList.Count > 0)
                g.DrawLine(pen,
                    orderedList[0].getPoint(),
                    orderedList[orderedList.Count - 1].getPoint());
        }

        public void drawHighlightedConnections(Graphics g, bool primaryConnections)
```

```
        {
            highlightedList().drawConnections(g, primaryConnections);
        }

        private bool pointAboveLine(ColoredPoint p1, ColoredPoint n1, ColoredPoint n2)
        {
            if (n2.Y - n1.Y == 0F)
                return p1.Y <= n1.Y;
            else
                return p1.Y <= ((n2.X - n1.X) * (p1.X - n1.X)) / (n2.Y - n1.Y) + n1.Y;
        }

        private bool pointBelowLine(ColoredPoint p1, ColoredPoint n1, ColoredPoint n2)
        {
            if (n2.Y - n1.Y == 0F)
                return p1.Y >= n1.Y;
            else
                return p1.Y >= ((n2.X - n1.X) * (p1.X - n1.X)) / (n2.Y - n1.Y) + n1.Y;
        }

        public bool hasPointAboveLine(ColoredPoint n1, ColoredPoint n2)
        {
            foreach (ColoredPoint cp in this)
                if (pointAboveLine(cp, n1, n2) && n1 != cp && n2 != cp) return true;
            return false;
        }

        public bool hasPointBelowLine(ColoredPoint n1, ColoredPoint n2)
        {
            foreach (ColoredPoint cp in this)
                if (pointBelowLine(cp, n1, n2) && n1 != cp && n2 != cp) return true;
            return false;
        }

        public void RemoveBetween(int start, int end)
        {
            if (start == end) return;
            if (++start >= Count) start = 0;
            if (--end < 0) end = Count - 1;

            if (start < end)
            {
                RemoveRange(start, end - start);
            }
            else
            {
                RemoveRange(start, Count - start);
                RemoveRange(0, end);
            }
        }

        public PointList Convexify()
        {
            if (Count <= 3) return new PointList(this);

            PointList convex = new PointList();
            PointF centroidF = getCentroid();
            ColoredPoint centroid = new ColoredPoint(centroidF.X, centroidF.Y);

            for (int i = 0; i < Count - 2; i++)
            {
                ColoredPoint p1 = this[i], p2 = this[i + 1], p3 = this[i + 2];
                if (!PointList.insideTriangle(p2, centroid, p1, p3))
                    convex.Add(p2);
            }
```

```
            // Add the end points
            if (!PointList.insideTriangle(this[0], centroid, this[Count - 1], this[1]))
                convex.Add(this[0]);

            if (!PointList.insideTriangle(this[Count - 1], centroid, this[0], this[Count - 2]))
                convex.Add(this[Count - 1]);

            return convex;
        }

        public static PointList Merge(PointList p1, PointList p2)
        {
            PointList newList = new PointList();
            foreach (ColoredPoint cp in p1) newList.Add(cp);
            foreach (ColoredPoint cp in p2) newList.Add(cp);
            return newList;
        }

        public static bool insideTriangle(ColoredPoint p, ColoredPoint a, ColoredPoint b, ColoredPoint c)
        {
            float m = (a.X - p.X) * (b.Y - p.Y) - (b.X - p.X) * (a.Y - p.Y);
            float n = (b.X - p.X) * (c.Y - p.Y) - (c.X - p.X) * (b.Y - p.Y);
            float o = (c.X - p.X) * (a.Y - p.Y) - (a.X - p.X) * (c.Y - p.Y);
            return (Math.Sign(m) == Math.Sign(n) && Math.Sign(n) == Math.Sign(o));
        }
    }
}
```

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

namespace ConvexHull
{
    class ConvexHullSolver
    {
        PictureBox picture;
        Graphics graphics;

        public ConvexHullSolver(PictureBox p)
        {
            picture = p;
            graphics = Graphics.FromImage(picture.Image);
        }

        public PointList Solve(PointList points)
        {
            //return RecursiveSolve(points).Convexify().Convexify().Convexify().Convexify
    ().Convexify().Convexify();
            PointList solution = RecursiveSolve(points);
            for (int i = 0; i < solution.Count / 2; i++)
                solution = solution.Convexify();
            return solution;
        }

        public PointList RecursiveSolve(PointList points)
        {
            if (points.Count <= 2)
            {
                return points.copy();
            }
            else
            {
                DividedList dl  = points.divide();
                PointList left  = Solve(dl.left);
                PointList right = Solve(dl.right);
                return Combine(left, right);
            }
        }

        public PointList Combine(PointList left, PointList right)
        {
            if (left.Count == 0) return right;
            if (right.Count == 0) return left;

            return PointList.Merge(left, right).clockWiseList().Convexify();
        }

        public PointList CombineHex(PointList left, PointList right)
        {
            if (left.Count == 0) return right;
            if (right.Count == 0) return left;

            left = left.clockWiseList();
            right = right.counterClockWiseList();

            PointList combined = new PointList();
            ColoredPoint edgePoint;

            edgePoint = left.topPoint();
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);
            edgePoint = left.leftPoint();
```

```
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);
            edgePoint = left.bottomPoint();
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);

            edgePoint = right.topPoint();
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);
            edgePoint = right.rightPoint();
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);
            edgePoint = right.bottomPoint();
            if (!combined.Contains(edgePoint)) combined.Add(edgePoint);

            return combined;
        }

    }
}
```

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

namespace ConvexHull
{
    public partial class FormMain : Form
    {

        abstract class PointGenerator
        {
            /// <summary>
            /// Generates a random point in a rectangle defined with the given width x height.
            /// Sub classes should implement a specific algorithm to determine point distribution.
            /// </summary>
            /// <param name="width">Width of the bounding rectangle</param>
            /// <param name="height">Height of the bounding rectangle</param>
            /// <returns>The generated random point</returns>
            public abstract ColoredPoint generatePointIn(int width, int height);
        }

        class GaussianPointGenerator : PointGenerator {

            // Uses this to generate the Guassian distribution
            // from http://msdn.microsoft.com/msdnmag/issues/06/09/TestRun/default.aspx
            // (http://msdn.microsoft.com/msdnmag/issues/06/09/TestRun/default.aspx?loc=&fig=true#fig9)
            class Gaussian
            {
                private Random r = new Random(0);
                private bool use_last_result = false; // flag for NextGaussian3()
                private double y2 = 0.0;  // secondary result for NextGaussian3()

                public double NextGaussian(double mean, double sd) // most efficient
                {
                    double x1, x2, w, y1 = 0.0;

                    if (use_last_result) // use answer from previous call?
                    {
                        y1 = y2;
                        use_last_result = false;
                    }
                    else
                    {
                        do
                        {
                            x1 = 2.0 * r.NextDouble() - 1.0;
                            x2 = 2.0 * r.NextDouble() - 1.0;
                            w = (x1 * x1) + (x2 * x2);
                        }
                        while (w >= 1.0); // are x1 and x2 inside unit circle?

                        w = Math.Sqrt((-2.0 * Math.Log(w)) / w);
                        y1 = x1 * w;
                        y2 = x2 * w;
                        use_last_result = true;
                    }
```

```
                    return mean + y1 * sd;
                }
            }
            Gaussian m_rand;

            public GaussianPointGenerator() {
                m_rand = new Gaussian();
            }

            public override ColoredPoint generatePointIn(int width, int height) {
                // mdj 1/8/07 generate random points with floats instead of ints to avoid having so many
                // duplicates as per Dr. Ringger's comments from Fall 06.  Doubles would be better, but there's
                // no built-in point data structure for doubles.
                return new ColoredPoint(width / 2 + (float)m_rand.NextGaussian(0, width / 6),
                                        height / 2 + (float)m_rand.NextGaussian(0, height / 6));
            }
        }

        class UniformPointGenerator : PointGenerator
        {
            private Random rand = new Random(0);

            /// <summary>
            /// Generates points that are uniformly distributed inside of the oval
            /// defined by the bounding rectangle passed in.
            /// </summary>
            /// <param name="width">Width of the bounding rectangle</param>
            /// <param name="height">Height of the bounding rectangle</param>
            /// <returns>Random point in an oval bound by the rectangle</returns>
            public override ColoredPoint generatePointIn(int width, int height) {
                double r, x, y;

                do {
                    //First generate points inside a circle
                    x = 2.0 * rand.NextDouble() - 1.0;
                    y = 2.0 * rand.NextDouble() - 1.0;

                    //Check radius
                    r = Math.Sqrt(x * x + y * y);
                } while( r > 1.0 );

                //Now convert to possibly-oval, larger bounds
                x *= width / 2 - 10;    //giving 5px border on each side
                y *= width / 2 - 10;
                //Translate to screen coords
                x += width / 2;
                y += height / 2;

                //Using float gives fewer duplicates than using int.
                //Double would be better but there is no Point-Double class.
                return new ColoredPoint((float)x, (float)y);
            }
        }

        PointGenerator pointGenerator;
        PointList m_primaryPointList, m_originalPointList;
        List<PointList> m_pointLists;
        ColoredPoint m_mouseDragPoint;
        ColoredPoint m_contextPoint;
        private Hashtable UniquePoints;
        //bool m_imageScaled;

        public FormMain()
```

```
        {
            InitializeComponent();
            pictureBoxView.Image = new Bitmap(pictureBoxView.Width, pictureBoxView.
    Height);
            pointGenerator = new UniformPointGenerator();
            radioUniform.Checked = true;    //start with this as the default
            UniquePoints = new Hashtable();
            m_primaryPointList = new PointList();
            m_originalPointList = new PointList();
            m_pointLists = new List<PointList>();
            this.ContextMenuStrip = cmUnhighlightedPoint;

        }

        private ColoredPoint getRandomPoint()
        {
            //eam, 1/17/08 -- changed to use a Strategy Pattern for generating pts w/
    different distributions
            return pointGenerator.generatePointIn(pictureBoxView.Width, pictureBoxView.
    Height);
        }

        private void resetPoints()
        {
            m_primaryPointList.Clear();
            m_pointLists.Clear();
            buttonCombineHulls.Enabled = false;
            buttonRemoveHulls.Enabled = false;
        }

        private void generatePoints()
        {
            // create point list
            int numPoints = int.Parse(textBoxNumPoints.Text);
            resetPoints();
            UniquePoints.Clear();
            ColoredPoint NewlyCreatedPoint;
            pbProgress.Value = pbProgress.Minimum;
            pbProgress.Maximum = 100;

            // make sure X value are unique.  Y values may contain duplicates by the way.

            while (UniquePoints.Count < numPoints)
            {
                pbProgress.Value = (int) (100f * ((float) UniquePoints.Count / ((float)
    numPoints)));
                NewlyCreatedPoint = getRandomPoint();   //get the next point to add
                if (!UniquePoints.Contains(NewlyCreatedPoint.X))
                    UniquePoints.Add(NewlyCreatedPoint.X,NewlyCreatedPoint);
            };

            // more convenient from here on out to use list.
            foreach (ColoredPoint point in UniquePoints.Values)
            {
                m_primaryPointList.Add(point);
            }

            // find the max and min
            float maxX = pictureBoxView.Image.Width, maxY = pictureBoxView.Image.Height;
            float minX = 0, minY = 0;
            foreach (ColoredPoint point in m_primaryPointList)
            {
                if(maxX < point.X) maxX = point.X;
                if(maxY < point.Y) maxY = point.Y;
                if(minX > point.X) minX = point.X;
                if(minY > point.Y) minY = point.Y;
```

```
            }

            // find translation factors
            float transX = (minX < 0F) ? -minX : 0F;
            float transY = (minY < 0F) ? -minY : 0F;

            // find the point range
            float rangeX = transX + (maxX - minX);
            float rangeY = transY + (maxY - minY);

            // find scaling factors
            const int padding = 20;
            float scaleX = (pictureBoxView.Image.Width - padding) / rangeX;
            float scaleY = (pictureBoxView.Image.Height - padding) / rangeY;

            // only shrink points , not enlarge them
            if (scaleX > 1.0) scaleX = 1F;
            if (scaleY > 1.0) scaleY = 1F;

            // scale points
            for(int i = 0; i < m_primaryPointList.Count; ++i)
            {
                ColoredPoint p = m_primaryPointList[i];
                p.X = transX + (p.X * scaleX);
                p.Y = transY + (p.Y * scaleY);
                m_primaryPointList[i] = p;
            }

            /*if (scaleX != 1.0 || scaleY != 1.0)
                m_imageScaled = true;*/
            m_originalPointList = m_primaryPointList.copy();
            pictureBoxView.Invalidate();
            statusLabel.Text = "" + numPoints + " points Generated. Scale factor: " +
                 ((scaleX >= scaleY) ? scaleX : scaleY);
        }

        #region GUI Control
        private void buttonGenerate_Click(object sender, EventArgs e)
        {
            generatePoints();
        }

        private void textBoxNumPoints_Validating(object sender, CancelEventArgs e)
        {
            int result;
            if(!(int.TryParse(textBoxNumPoints.Text, out result)))
            {
                e.Cancel = true;
            }
        }

        private void buttonSolve_Click(object sender, EventArgs e)
        {
            // Clear our list of lists, except for the first one

            // Start solving
            Stopwatch timer = new Stopwatch();
            timer.Start();
            ConvexHullSolver convexHullSolver = new ConvexHullSolver(pictureBoxView);
            //m_pointLists = convexHullSolver.Solve2(m_primaryPointList);
            m_pointLists.Add(convexHullSolver.Solve(m_primaryPointList).highlight());
            timer.Stop();
            m_primaryPointList.unhighlight();
            buttonRemoveHulls.Enabled = true;
            pictureBoxView.Invalidate();
            statusLabel.Text = "Done.  Time taken: " + timer.Elapsed;
```

```
        }

        private void buttonClearToPoints_Click(object sender, EventArgs e)
        {
            resetPoints();
            m_primaryPointList = m_originalPointList.copy();
            pictureBoxView.Invalidate();
            statusLabel.Text = "Cleared to the original points.";
        }
        #endregion

        private void radioUniform_CheckedChanged(object sender, EventArgs e) {
            if( !(pointGenerator is UniformPointGenerator) ) {
                pointGenerator = new UniformPointGenerator();
            }
        }

        private void radioGaussian_CheckedChanged(object sender, EventArgs e) {
            if( !(pointGenerator is GaussianPointGenerator) ) {
                pointGenerator = new GaussianPointGenerator();
            }
        }

        private void pictureBoxView_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            // Clear the picture box view
            g.Clear(Color.White);
            // Draw dots and connections
            foreach (PointList pointList in m_pointLists)
            {
                pointList.drawPoints(g);
                pointList.drawHighlightedConnections(g, false);
            }
            // Draw the official source list as well
            m_primaryPointList.drawPoints(g);
            m_primaryPointList.drawHighlightedConnections(g, true);
        }

        private void buttonHighlightEverything_Click(object sender, EventArgs e)
        {
            m_primaryPointList.highlight();
            pictureBoxView.Invalidate();
        }

        private void buttonHighlightNothing_Click(object sender, EventArgs e)
        {
            m_primaryPointList.unhighlight();
            pictureBoxView.Invalidate();
        }

        private void btnTop_Click(object sender, EventArgs e)
        {
            PointList top = m_primaryPointList.selectTopHalf();
            if (top[0].isHighlighted())
                m_primaryPointList.unhighlight();
            else
                top.highlight();
            pictureBoxView.Invalidate();
        }

        private void btnBottom_Click(object sender, EventArgs e)
        {
            PointList bottom = m_primaryPointList.selectBotHalf();
            if (bottom[0].isHighlighted())
```

```
            m_primaryPointList.unhighlight();
        else
            bottom.highlight();
        pictureBoxView.Invalidate();
    }

    private void pictureBoxView_MouseDown(object sender, MouseEventArgs e)
    {
        Point mouse = new Point(e.X, e.Y);
        if (e.Button == MouseButtons.Left)
        {
            foreach (ColoredPoint cp in m_primaryPointList)
            {
                if (cp.contains(mouse))
                {
                    m_mouseDragPoint = cp;
                }
            }
        }
    }

    private void pictureBoxView_MouseMove(object sender, MouseEventArgs e)
    {
        if (e.Button == MouseButtons.Left && m_mouseDragPoint != null)
        {
            m_mouseDragPoint.X = e.X;
            m_mouseDragPoint.Y = e.Y;
            pictureBoxView.Invalidate();
        }


        // Context Menus
        Point mouse = new Point(e.X, e.Y);
        foreach (ColoredPoint cp in m_primaryPointList)
        {
            if (cp.contains(mouse))
            {
                // Store for later context menu use
                m_contextPoint = cp;

                if (cp.isHighlighted())
                {
                    this.ContextMenuStrip = cmHighlightedPoint;
                    break;
                }
                else
                {
                    this.ContextMenuStrip = cmUnhighlightedPoint;
                    break;
                }
            }
            else
            {
                this.ContextMenuStrip = null;
            }
        }

    }

    private void pictureBoxView_MouseUp(object sender, MouseEventArgs e)
    {
        if (e.Button == MouseButtons.Left)
        {
            m_mouseDragPoint = null;
        }
    }
```

```
        private void pictureBoxView_MouseClick(object sender, MouseEventArgs e)
        {

        }

        private void pictureBoxView_MouseDoubleClick(object sender, MouseEventArgs e)
        {
            Point mouse = new Point(e.X, e.Y);
            if (e.Button == MouseButtons.Left)
            {
                foreach (ColoredPoint cp in m_primaryPointList)
                {
                    if (cp.contains(mouse))
                    {
                        if (cp.isHighlighted())
                            cp.unhighlight();
                        else
                            cp.highlight();
                        pictureBoxView.Invalidate();
                        return;
                    }
                }

                // Did not click on a point, so create one
                ColoredPoint newPoint = new ColoredPoint(e.X, e.Y);
                newPoint.highlight();
                m_primaryPointList.Add(newPoint);
                pictureBoxView.Invalidate();
            }
        }

        private void highlightMenuItem_Click(object sender, EventArgs e)
        {
            m_contextPoint.highlight();
            pictureBoxView.Invalidate();
        }

        private void unhighlightMenuItem_Click(object sender, EventArgs e)
        {
            m_contextPoint.unhighlight();
            pictureBoxView.Invalidate();
        }

        private void removeUnhighlightedMenuItem_Click(object sender, EventArgs e)
        {
            m_primaryPointList.Remove(m_contextPoint);
            pictureBoxView.Invalidate();
        }

        private void removeHighlightedMenuItem_Click(object sender, EventArgs e)
        {
            m_primaryPointList.Remove(m_contextPoint);
            pictureBoxView.Invalidate();
        }

        private void buttonCombineHulls_Click(object sender, EventArgs e)
        {
            ConvexHullSolver convexHullSolver = new ConvexHullSolver(pictureBoxView);
            if (m_pointLists.Count < 2) throw new Exception("Must have 2 point lists to combine.");
            PointList combined = convexHullSolver.Combine(m_pointLists[0], m_pointLists[1]);

            m_primaryPointList.unhighlight();
            m_pointLists.Clear();
```

```
            buttonCombineHulls.Enabled = false;
            buttonRemoveHulls.Enabled = false;

            m_pointLists.Add(combined);
            pictureBoxView.Invalidate();
        }

        private void addAsHullToolStripMenuItem_Click(object sender, EventArgs e)
        {
            m_pointLists.Add(m_primaryPointList.highlightedList().copy());
            if (m_pointLists.Count >= 1) buttonRemoveHulls.Enabled = true;
            if (m_pointLists.Count >= 2) buttonCombineHulls.Enabled = true;
            m_primaryPointList.unhighlight();
            pictureBoxView.Invalidate();
        }

        private void buttonRemoveHulls_Click(object sender, EventArgs e)
        {
            m_pointLists.Clear();
            buttonCombineHulls.Enabled = false;
            buttonRemoveHulls.Enabled = false;
            pictureBoxView.Invalidate();
        }

    }
}
```