File: /home/unit3/School/cs405/cs405/gfx.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3
 4  namespace cs405
 5  {
 6    // this file holds all our graphics operations and algorithms
 7
 8    /***** an 8bpp greyscale image buffer ******/
 9    public class buffer
10    {
11      // we're working in an 8-bit greyscale colourspace
12      public static byte WHITE = 255;
13      public static byte BLACK = 0;
14      // image width and height
15      internal int ROWS = 512;
16      internal int COLS = 512;
17
18      /* Global Variables */
19      // C#'s "jagged" array declaration is weird, we need to init in the constructor
20      internal byte[][] image;  /* image buffer */
21      internal byte cur_color = BLACK;    /* the current drawing color */
22
23      /* Illumination Variables */
24      internal float amb_light;   /* ambient light intensity */
25      internal float dir_light;   /* directional light intensity */
26      internal float[] dir_dir = new float[3];    /* the direction of directional light
    */
27
28      /* Transformation Variables */
29      internal float[] VRP = new float[3] {0.0f, 0.0f, 0.0f};   /* View Reference Point
    */
30      internal float[] VPN = new float[3] {0.0f, 0.0f, 1.0f};   /* View Plane Normal */
31      internal float[] VUP = new float[3] {0.0f, 1.0f, 0.0f};   /* View Up Direction */
32      internal float focal = 0.05f;     /* Focal length of the virtual camera */
33      // "film" size of virtual camera, default to 35mm
34      internal float xmin = -0.0175f; internal float xmax = 0.0175f;
35      internal float ymin = -0.0175f; internal float ymax = 0.0175f;
36
37      // we don't need jagged arrays for matrixes, so just use standard multidimensional
38      internal float[,] M3dview = new float[4,4];   /* 3D Viewing Transformation Matrix
    */
39      internal float[,] Mper = new float[4,4]; /* Perspective Projection Matrix */
40      internal float[,] Muv = new float[3,3];   /* View Port Mapping Matrix */
41
42      public buffer(int width, int height) {
43        ROWS = height;
44        COLS = width;
45        // initialize jagged image array
46        image = new byte[ROWS][];
47        for (int y = 0; y < ROWS; y++) {
48          image[y] = new byte[COLS];
49        }
50        // init_global()
51        fill(WHITE);
```

```
 52        cur_color = BLACK;
 53        // init Muv, it is a fixed transform matrix
 54        float Sx = 511.0f / (xmax - xmin);
 55        float Sy = 511.0f / (ymax - ymin);
 56        Muv[0,0] = -Sx; Muv[2,0] = (-xmin)*Sx;
 57        Muv[1,1] = Sy; Muv[2,1] = (-ymin)*Sy;
 58        Muv[2,2] = 1.0f;
 59      }
 60
 61      // small method to determine if a given vertex is inside our window
 62      public bool vertex_inside(int x, int y) {
 63        if ((x < 0) || (y < 0) || (x >= COLS) || (y >= ROWS)) {
 64          return false;
 65        } else {
 66          return true;
 67        }
 68      }
 69      public bool vertex_inside(point p) {
 70        return vertex_inside(p.x, p.y);
 71      }
 72      // test for just one screen edge, for polygon drawing
 73      public bool vertex_inside(point p, screen_edge e) {
 74        switch (e) {
 75        case screen_edge.EAST:
 76          if (p.x >= COLS) { return false; }
 77          else { return true; }
 78        case screen_edge.SOUTH:
 79          if (p.y >= ROWS) { return false; }
 80          else { return true; }
 81        case screen_edge.WEST:
 82          if (p.x < 0) { return false; }
 83          else { return true; }
 84        case screen_edge.NORTH:
 85          if (p.y < 0) { return false; }
 86          else { return true; }
 87        default:
 88          return false;
 89        }
 90      }
 91
 92      // very simple algorithm to fill image with a color
 93      public void fill(byte color) {
 94        int x,y;
 95        /* Initialize the image buffer */
 96        for (y=0; y<ROWS; y++) {
 97          for (x=0; x<COLS; x++) {
 98            image[y][x] = color;
 99          }
100        }
101      }
102      // convert our image buffer to a serial data stream for output to file /
     texture / whatever
103      public byte[] serialize() {
104        int i = 0; // store the current index in our output array.
```

```
105         // init array to return
106         byte[] serial = new byte[ROWS*COLS];
107         // now convert from jagged array to normal array
108         for (int y = 0; y < ROWS; y++) {
109           // jagged arrays make this easy to iterrate through and write each line of
    data
110           Array.Copy(image[y], 0, serial, i, COLS);
111           i += COLS;
112         }
113         return serial;
114       }
115 
116       // helper to multiply 4,4 matricies
117       internal float[,] mult4(float[,] A, float[,] B) {
118         // check dimensions of matricies
119         if ((A.GetLength(0) != 4) || (A.GetLength(1) != 4) || (A.GetLength(0) != 4) ||
    (A.GetLength(1) != 4)) {
120           throw new Exception("Incorrect matrix dimensions in gfx.mult4");
121         }
122         float[,] m = new float[4,4];
123         float val = 0.0f; // temp to hold addition while doing matrix multiplication
124         for (int row = 0; row < 4; row++) {
125           for (int col = 0; col < 4; col++) {
126             val = 0.0f;
127             for (int i = 0; i < 4; i++) {
128               val += A[i,row]*B[col,i];
129             }
130             m[col,row] = val;
131           }
132         }
133         return m;
134       }
135       // debugging helper to print a 4,4 matrix to console
136       internal void printMatrix(string name, float[,] m) {
137         System.Console.WriteLine("matrix " + name);
138         for (int row = 0; row < 4; row++) {
139           System.Console.Write("( ");
140           for (int col = 0; col < 4; col++) {
141             System.Console.Write(m[col,row] + " ");
142           }
143           System.Console.WriteLine(")");
144         }
145       }
146       // construct our 3d viewing transforms for rendering 3d polys
147       public void buildM3dView() {
148         float[] P0 = new float[4];
149         float[] P1 = new float[4];
150         float[] P2 = new float[4];
151         // and some temporary matrices to store values in
152         float[,] tm1 = new float[4,4];
153         float[,] tm2 = new float[4,4];
154         float[,] tm3 = new float[4,4];
155         float[,] tm4 = new float[4,4];
156 
```

```
157         /* Covert VRP, VPN, and VUP to P0, P1, P2 */
158         for (int i=0; i<3; i++) {
159           P0[i] = VRP[i];
160           P1[i] = VRP[i] + VPN[i];
161           P2[i] = VRP[i] + VUP[i];
162         }
163         P0[3] = P1[3] = P2[3] = 1.0f;
164 
165         /* Step 1: Get T(-P0).
166         - construct the translation matrix, and store it in tm1. */
167         tm1[0,0] = 1; tm1[1,1] = 1; tm1[2,2] = 1; tm1[3,3] = 1;
168         tm1[3,0] = -(P0[0]);
169         tm1[3,1] = -(P0[1]);
170         tm1[3,2] = -(P0[2]);
171 
172         /* Step 2: Get Ry(a).
173         - compute D1
174         - compute sin(a), cos(a)
175         - construct the rotation matrix, and store it in tm2. */
176         // the xz projection length of p0'p1'
177         float D1 = (float)Math.Sqrt(Math.Pow(P1[2]-P0[2], 2)+Math.Pow(P1[0]-P0[0], 2));
178         float cos = (P1[2] - P0[2]) / D1;
179         float sin = -(P1[0] - P0[0]) / D1;
180         // construct our Ry matrix
181         tm2[0,0] = cos; tm2[2,0] = sin;
182         tm2[1,1] = 1;
183         tm2[0, 2] = -1 * (sin); tm2[2,2] = cos;
184         tm2[3,3] = 1;
185 
186         /* Step 3: get Rx(b).
187         - compute D2
188         - compute sin(b), cos(b)
189         - construct the rotation matrix, and store it in tm3. */
190         // the yz projection length of p1"p2"
191         float D2 = (float)Math.Sqrt(Math.Pow(P1[0]-P0[0], 2) + Math.Pow(P1[1]-P0[1], 2)
    + Math.Pow(P1[2]-P0[2], 2));
192         cos = D1 / D2;
193         sin = (P1[1] - P0[1]) / D2;
194         // construct our Rx matrix
195         tm3[0,0] = 1;
196         tm3[1,1] = cos; tm3[2,1] = -sin;
197         tm3[1,2] = sin; tm3[2,2] = cos;
198         tm3[3,3] = 1;
199 
200         /* Step 4: Get Rz(r).
201         - first concatenate tm3*tm2*tm1 = tm1;
202         - apply the new tm1 to point P2 to get P2'''
203         - compute D3
204         - compute sin(r), and cos(r)
205         - construct Rz(r) and store it in tm2. */
206         // concatenate previous operations
207         tm4 = mult4(tm2, tm1);
208         tm1 = mult4(tm3, tm4);
209         // find P2'''
```

```
210         float[] P2ppp = new float[4];
211         P2ppp[0] = (tm1[0,0]*P2[0]) + (tm1[1,0]*P2[1]) + (tm1[2,0]*P2[2]) + (tm1[3,0]*P2
    [3]);
212         P2ppp[1] = (tm1[0,1]*P2[0]) + (tm1[1,1]*P2[1]) + (tm1[2,1]*P2[2]) + (tm1[3,1]*P2
    [3]);
213         P2ppp[2] = (tm1[0,2]*P2[0]) + (tm1[1,2]*P2[1]) + (tm1[2,2]*P2[2]) + (tm1[3,2]*P2
    [3]);
214         P2ppp[3] = (tm1[0,3]*P2[0]) + (tm1[1,3]*P2[1]) + (tm1[2,3]*P2[2]) + (tm1[3,3]*P2
    [3]);
215         // D3 = projection length
216         float D3 = (float)Math.Sqrt(Math.Pow(P2ppp[0], 2.0) + Math.Pow(P2ppp[1], 2.0));
217         cos = P2ppp[1] / D3;
218         sin = P2ppp[0] / D3;
219         // construct Rz matrix
220         tm2[0,0] = cos; tm2[1,0] = -sin; tm2[2,0] = 0.0f; tm2[3,0] = 0.0f;
221         tm2[0,1] = sin; tm2[1,1] = cos; tm2[2,1] = 0.0f; tm2[3,1] = 0.0f;
222         tm2[0,2] = 0.0f; tm2[1,2] = 0.0f; tm2[2,2] = 1.0f; tm2[3,2] = 0.0f;
223         tm2[0,3] = 0.0f; tm2[1,3] = 0.0f; tm2[2,3] = 0.0f; tm2[3,3] = 1.0f;
224
225         /* Step 5: Get M3dview.
226         - the final 3D viewing transformation matrix is obtained by
227         concatenating tm2*tm1 = M3dview. */
228         M3dview = mult4(tm2, tm1);
229       }
230       // method to build our perspective transform matrix
231       public void buildMper() {
232         Mper[0,0] = 1; Mper[1,1] = 1; Mper[2,2] = 1;
233         Mper[2,3] = 1.0f / focal;
234       }
235       // method to transform 3d world coordinates into 2d screen coordinates
236       public point transform(point3d p) {
237         point o = new point(); // this will be our transformed point eventually
238         float[] tmp = new float[4]; // put transforms in here as we go
239         // create size 4 matrix
240         tmp[0] = p.x;
241         tmp[1] = p.y;
242         tmp[2] = p.z;
243         tmp[3] = 1.0f;
244         // now, apply 3d viewing transform
245         tmp[0] = (M3dview[0,0]*tmp[0]) + (M3dview[1,0]*tmp[1]) + (M3dview[2,0]*tmp[2])
    + (M3dview[3,0]*tmp[3]);
246         tmp[1] = (M3dview[0,1]*tmp[0]) + (M3dview[1,1]*tmp[1]) + (M3dview[2,1]*tmp[2])
    + (M3dview[3,1]*tmp[3]);
247         tmp[2] = (M3dview[0,2]*tmp[0]) + (M3dview[1,2]*tmp[1]) + (M3dview[2,2]*tmp[2])
    + (M3dview[3,2]*tmp[3]);
248         tmp[3] = (M3dview[0,3]*tmp[0]) + (M3dview[1,3]*tmp[1]) + (M3dview[2,3]*tmp[2])
    + (M3dview[3,3]*tmp[3]);
249         // now, apply perspective projection
250         tmp[0] = (Mper[0,0]*tmp[0]) + (Mper[1,0]*tmp[1]) + (Mper[2,0]*tmp[2]) + (Mper
    [3,0]*tmp[3]);
251         tmp[1] = (Mper[0,1]*tmp[0]) + (Mper[1,1]*tmp[1]) + (Mper[2,1]*tmp[2]) + (Mper
    [3,1]*tmp[3]);
252         tmp[2] = (Mper[0,2]*tmp[0]) + (Mper[1,2]*tmp[1]) + (Mper[2,2]*tmp[2]) + (Mper
    [3,2]*tmp[3]);
```

```
253         tmp[3] = (Mper[0,3]*tmp[0]) + (Mper[1,3]*tmp[1]) + (Mper[2,3]*tmp[2]) + (Mper
    [3,3]*tmp[3]);
254         // normalize 2d components of tmp
255         tmp[0] = tmp[0] / (tmp[2] / focal);
256         tmp[1] = tmp[1] / (tmp[2] / focal);
257         tmp[2] = 1.0f;
258         o.x = (int)((Muv[0,0]*tmp[0]) + (Muv[1,0]*tmp[1]) + (Muv[2,0]*tmp[2]));
259         o.y = (int)((Muv[0,1]*tmp[0]) + (Muv[1,1]*tmp[1]) + (Muv[2,1]*tmp[2]));
260
261         return o;
262       }
263     }
264
265     /***** a class to do line drawing *****/
266     public class line {
267       public int x0, y0, x1, y1; // our start and endpoints
268       protected int dx, dy, incE, incNE, d, x, y; // for our main algorithm
269       protected float slope; // only used to quickly test transform cases
270       protected buffer img; // reference to the buffer to draw into
271       protected draw_pixel draw; // our instantiated pixel drawing function
272
273       public line(int startX, int startY, int endX, int endY, buffer buff) {
274         img = buff;
275         x0 = startX;
276         y0 = startY;
277         x1 = endX;
278         y1 = endY;
279         // make sure line doesn't go outside of our buffer's bounds
280         if (!(img.vertex_inside(x0, y0)) || !(img.vertex_inside(x1, y1))) {
281           throw new Exception("Line values out of bounds (" + x0 + "," + y0 + ")->(" +
    x1 + "," + y1 + ")");
282         }
283         dx = x1 - x0;
284         dy = y1 - y0;
285         slope = (float)dy / (float)dx;
286         // we're going to create a pixel drawing function called draw() to use,
287         // and give it the midpoint line base case by default (0 < slope <= 1)
288         draw = new draw_pixel(draw_pos);
289       }
290
291       // helper method to switch which point is point 0 and which is point 1
292       // used to help optimize line drawing algorithm
293       public void swap() {
294         int tmp; // hold values to be swapped
295         tmp = x0;
296         x0 = x1;
297         x1 = tmp;
298         tmp = y0;
299         y0 = y1;
300         y1 = tmp;
301         // recalculate slope values
302         dx = x1 - x0;
303         dy = y1 - y0;
304         slope = (float)dy / (float)dx;
```

```
305       }
306
307       // have a delegate method to handle different drawing transforms efficiently
308       // first, we declare a generic method
309       // Note: x0,y0 is needed so we can reverse pre-algorithm transformations
310       protected delegate void draw_pixel(int x, int y);
311       // now functions to instantiate the delegate depending on case in line()
312       protected void draw_pos(int x, int y) { // base case
313         img.image[y][x] = img.cur_color;
314       }
315       protected void draw_gt1(int x, int y) { // slope > 1
316         img.image[x][y] = img.cur_color;
317       }
318       protected void draw_neg(int x, int y) { // -1 <= slope < 0
319         int tmp = (y - y0); // relative dy
320         img.image[y0 - tmp][x] = img.cur_color;
321       }
322       protected void draw_lt1(int x, int y) { // slope < -1
323         int tmp = (x - x0); // relative dy
324         img.image[x0 - tmp][y] = img.cur_color;
325       }
326
327       /***** midpoint line algorithm *****/
328       public void draw_line() {
329         int tmp; // temp variable to hold values being swapped around
330         // handle non-midpoint algorithm cases
331         if (x0 == x1) { // vertical line
332           if (y1 < y0) {
333             swap();
334           }
335           for (y = y0; y <= y1; y++) {
336             img.image[y][x0] = img.cur_color;
337           }
338         } else if (y0 == y1) { // horizontal line
339           if (x1 < x0) {
340             swap();
341           }
342           for (x = x0; x <= x1; x++) {
343             img.image[y0][x] = img.cur_color;
344           }
345         } else { // midpoint algorithm
346           // if points are reversed horizontally, swap them.
347           if (x1 < x0) {
348             swap();
349           }
350
351           // now determine transforms to handle different types of lines
352           // we transform x0,y0 x1,y1 here, and then count on our drawing function
353           // to reverse the transform when drawing the pixel
354           if ((slope <= 1) && (slope > 0)) {
355             draw = new draw_pixel(draw_pos);
356           } else if (slope > 1) {
357             // symmetry around 45 degrees
358             tmp = x0; x0 = y0; y0 = tmp; tmp = x1; x1 = y1; y1 = tmp; tmp = 0;
```

```
359             // here is where we set which version of the drawing function to use
360             draw = new draw_pixel(draw_gt1);
361           } else if ((slope < 0) && (slope >= -1)) {
362             // vertical symmetry around horizontal line extending from x0
363             dy = y1 - y0;
364             y1 = y0 - dy;
365             draw = new draw_pixel(draw_neg);
366           } else if (slope < -1) {
367             // flip around localized horizontal, and then around 45 degrees;
368             dy = y1 - y0;
369             y1 = y0 - dy;
370             tmp = x0; x0 = y0; y0 = tmp; tmp = x1; x1 = y1; y1 = tmp; tmp = 0;
371             draw = new draw_pixel(draw_lt1);
372           }
373 
374           // base case, handles 0 < slope <= 1
375           dx = x1 - x0;
376           dy = y1 - y0;
377           d = 2*dy - dx;          /* initial value of d */
378           incE = 2*dy;            /* increment for move to E */
379           incNE = 2*(dy - dx);    /* increment for move to NE */
380           x = x0;
381           y = y0;
382           while (x <= x1) {
383             draw(x, y);
384             if (d <= 0) {       /* M is above the line */
385               d += incE;
386               x++;              /* Move to E */
387             } else {            /* M is on or below the line */
388               d += incNE;
389               x++; y++;         /* Move to NE */
390             }
391           }
392         }
393       }
394     }
395 
396     /***** class implementing midpoint circle algorithm *****/
397     public class circle {
398       int x, y, x0, y0, h; // algorithm variables
399       buffer img; // framebuffer to draw to
400 
401       // midpoint circle algorithm
402       public circle(int X, int Y, int R, buffer buff) {
403         x0 = X;
404         y0 = Y;
405         x = 0;
406         y = R;
407         h = 1 - R; // use h method so no floats involved
408         img = buff;
409       }
410 
411       // the bulk of the midpoint circle algorithm is done here
412       public void draw() {
```

```
413         draw_point(x, y, x0, y0);
414         while (y >= x) {
415           if (h < 0) {  /* select E */
416             h += 2*x + 3;
417             x++;
418           } else {       /* select SE */
419             h += 2*(x-y) + 5;
420             x++; y--;
421           }
422           draw_point(x, y, x0, y0);
423         }
424       }
425
426       // method to do symmetry drawing for circle
427       private void draw_point(int x, int y, int x0, int y0) { // slope < -1
428         line l;
429         // to do filled circle, draw lines from relative y axis (x0) to each symmetry
    point
430         l = new line(x0, y0 + y, x0 + x, y0 + y, img);
431         l.draw_line();
432         l = new line(x0, y0 + x, x0 + y, y0 + x, img);
433         l.draw_line();
434         l = new line(x0, y0 - y, x0 - x, y0 - y, img);
435         l.draw_line();
436         l = new line(x0, y0 - x, x0 - y, y0 - x, img);
437         l.draw_line();
438         l = new line(x0, y0 - y, x0 + x, y0 - y, img);
439         l.draw_line();
440         l = new line(x0, y0 - x, x0 + y, y0 - x, img);
441         l.draw_line();
442         l = new line(x0, y0 + y, x0 - x, y0 + y, img);
443         l.draw_line();
444         l = new line(x0, y0 + x, x0 - y, y0 + x, img);
445         l.draw_line();
446       }
447     }
448
449     /***** class implementing scan-converting polygon drawing *****/
450     public class polygon {
451       // list to parse points from when doing initial clipping
452       // List generic also lets us dynamically add points, which is useful for clipping
453       private List<point> points;
454       internal protected edge_node ET; // edge node linked lists, generated from points
455       protected buffer img; // frame buffer to draw to
456
457       // polygon expects a list of alternating x,y values that form edge points
458       // in newpoints
459       public polygon(point[] newpoints, buffer buff) {
460         points = new List<point>();
461         points.AddRange(newpoints);
462         img = buff;
463         edges_scan();
464       }
465
```

```
466       // scan through our points array and enter points into our edge table
467       protected void edges_scan() {
468         // first, add start point to end of list, to create a closed poly
469         points.Add(points[0]);
470         // now, clip points to viewing area
471         clip();
472         // now create edge table ET from clipped points
473         for (int i = 0; i < points.Count - 1; i++) {
474           // only insert non horizontal edges
475           if (points[i].y != points[i+1].y) {
476             edges_insert(points[i], points[i+1]);
477           }
478         }
479         // close polygon if edge clipping has reopened it
480         if (points[points.Count-1] != points[0]) {
481           edges_insert(points[points.Count-1], points[0]);
482         }
483       }
484
485       // calculates intersection point between line segment and screen edges
486       protected point intersect(point p0, point p1, screen_edge e) {
487         point i = new point(); // our intersection point to return
488         // calculate slope of line segment, to find intersection point
489         float slope = (float)(p1.y - p0.y) / (float)(p1.x - p0.x);
490         switch (e) {
491         case screen_edge.EAST:
492           i.x = img.COLS-1; // EAST value
493           // calculate y position using slope and relative origin of p0
494           // y = mx + b (the local y intercept)
495           i.y = (int)((i.x - p0.x) * slope) + p0.y;
496           break;
497         case screen_edge.SOUTH:
498           i.y = img.ROWS; // SOUTH value, not -1 because of drawing algorithm
499           // calculate x position using slope and relative origin of p0
500           // x = (y / m) + c (the local x intercept)
501           i.x = (int)((i.y - p0.y) / slope) + p0.x;
502           break;
503         case screen_edge.WEST:
504           i.x = 0; // WEST value
505           i.y = (int)((i.x - p0.x) * slope) + p0.y;
506           break;
507         case screen_edge.NORTH:
508           i.y = 0; // NORTH value
509           // calculate x position using slope and relative origin of p0
510           i.x = (int)((i.y - p0.y) / slope) + p0.x;
511           break;
512         }
513         return i;
514       }
515
516       // this method clips our points list to the screen boundaries
517       // using modified Sutherland_Hodgmen algorithm.
518       protected void clip() {
519         poly_clip(screen_edge.EAST);
```

```
520         poly_clip(screen_edge.NORTH);
521         poly_clip(screen_edge.WEST);
522         poly_clip(screen_edge.SOUTH);
523       }
524       // clip to a specified edge only
525       protected void poly_clip(screen_edge e) {
526         // we'll assign points to this after clipping is done
527         List<point> newpoints = new List<point>();
528         bool S, E; // our flags for determining case
529         point sect; // our intersection point
530         // kickstart point checking
531         S = img.vertex_inside(points[0], e);
532         // now check all points and insert into new list
533         for (int pos = 0; pos < points.Count - 1; pos++) {
534           E = img.vertex_inside(points[pos+1], e);
535           if (S && E) { // edge is completely inside window
536             // add starting point if it's the first in the list
537             if (pos == 0) {
538               newpoints.Add(points[0]);
539             }
540             // add second point
541             newpoints.Add(points[pos+1]);
542           } else if (!(S || E)) { // edge is completely outside window
543             // add nothing to the output list, this edge is clipped
544           } else if (S && !E) { // endpoint exits window
545             // add starting point if it's the first in the list
546             if (pos == 0) {
547               newpoints.Add(points[0]);
548             }
549             sect = intersect(points[pos], points[pos+1], e);
550             // add intersection point
551             newpoints.Add(sect);
552           } else if (!S && E) { // start point exits window
553             sect = intersect(points[pos], points[pos+1], e);
554             // add intersection point and end point
555             newpoints.Add(sect);
556             newpoints.Add(points[pos+1]);
557           }
558           S = E;
559         }
560         points = newpoints;
561       }
562
563       // Do an insertion sort, which finds the right place for a node
564       // when it's added into the linked list
565       protected void edges_insert(point p0, point p1) {
566         // create new edge node
567         edge_node e = new edge_node(p0, p1);
568         // if there's no starting edge, then this is it
569         if (ET == null) {
570           ET = e;
571         } else {
572           edge_node curr = ET;
573           edge_node prev = ET;
```

```
574           while (curr != null) {
575             // we sort by ymin and then ymax, so even if ymin is the same,
576             // put in front if ymax is less
577             if ((e.ymin < curr.ymin) || ((e.ymin == curr.ymin) && (e.ymax <
      curr.ymax))) {
578               // special case, if prev == start of list, make e start of list
579               if (prev == ET) {
580                 e.link = prev;
581                 ET = e;
582               } else {
583                 prev.link = e;
584                 e.link = curr;
585               }
586               // we're done, don't need to loop anymore
587               break;
588             }
589             // move on in list
590             prev = curr;
591             curr = curr.link;
592             // special case, e > everything in list, add to end of list
593             if (curr == null) {
594               prev.link = e;
595             }
596           }
597         }
598       }
599
600       // scan converting polygon drawing
601       public void draw() {
602         // check to see if we have any nodes
603         if (ET == null) {
604           throw new Exception("No nodes to scan convert, polygon drawing failed.");
605         }
606         int y = ET.ymin;
607         // keep track of current and previous nodes while going through linked list
608         edge_node curr, prev;
609         // we'll use C#'s class to sort our x points on each scanline
610         List<int> Xs;
611         while (y < img.ROWS) {
612           // clear out any segments we're done with
613           curr = ET;
614           prev = ET;
615           while (curr != null) {
616             if (curr.ymax <= y) {
617               // remove curr from our active list!
618               if (curr == ET) {
619                 ET = curr.link;
620               } else {
621                 prev.link = curr.link;
622               }
623             }
624             prev = curr;
625             curr = curr.link;
626           }
```

```
627           // if we're out of nodes, we're done.
628           if (ET == null) {
629             break;
630           }
631           // otherwise, find all x values for this line
632           // new list of x values every line
633           Xs = new List<int>();
634           curr = ET;
635           while ((curr != null) && (curr.ymin <= y)) {
636             Xs.Add((int)curr.x);
637             curr = curr.link;
638           }
639           Xs.Sort();
640           Xs.TrimExcess();
641           if (Xs.Count % 2 != 0) {
642             Console.WriteLine("Got "+Xs.Count+" Xs on line "+y+":");
643             foreach (float x in Xs) {
644               Console.Write(x+", ");
645             }
646             Console.WriteLine();
647           }
648
649           // now draw between each pair of x values
650           for (int i=0; i<Xs.Count; i+=2) {
651             line l = new line(Xs[i], y, Xs[i+1], y, img);
652             l.draw_line();
653           }
654           // and finally increment all active values
655           y++;
656           curr = ET;
657           while ((curr != null) && (curr.ymin <= y)) {
658             curr.inc();
659             curr = curr.link;
660           }
661         }
662       }
663     }
664   }
```