Stereography and Autostereograms

Peter Chang
peterc@a3.ph.man.ac.uk
or etzpc@thmech.nott.ac.uk

May 10, 1995

Abstract

A short article on autostereograms with a perspectively correct method of generation.

1 Preface

2 Introduction

Throughout pedestrian high streets, poster shops and book shops, up and down the country — not just in one country but sweeping all over the world — people are standing around staring at what, at first sight, looks like images of modern art. They come in several varieties: random dots in black and white or in assorted colours; seemingly repetitive splotches of colour; and tiled patterns. But they are not works by Jackson Pollock aspirants.

They are a relatively new type of stereogram, and proliferate in various forms from postcards to books and albums to posters. Even some advertisers use them as eye-catching novelties or gimmicks. They come in many names (Holusion, Stare-E-O grams, SIRDS, etc) but are generically know as single-image stereograms or autostereograms.

3 Survey of Stereography

Stereography stems from the need to reproduce realistic images on a flat surface, that is, images which depict objects with a feeling of depth and solidity. In other words, flat images look three dimensional.

Man has used other forms of representation for the purpose of modelling his situation (imaginary or otherwise), such as sculptures and carvings. But drawing on two-dimensional surfaces is his main form of expression whether it be sketching, painting or even writing.

Most of the information here has been culled from the following references, [1, 2, 3, 4].

3.1 History

Through the ages, artists have experimented to find ways of producing more lifelike pictures. Perspective and other visual cues were introduced from the early 15th century on to the present day where computer generated images are beginning to approach photographs in realism. [Historical note: Brunelleschi was the originator of the first demonstration of scientific (geometric?) perspective and Alberti devised a perspective construction for painters (in his treatise "On Painting," 1436).]

Sir Charles Wheatstone invented the stereoscope [1838] and thus began a wave of interest in stereography. Though the ancient Greeks had some knowledge of stereography, the Victorian renewal of interest was further sparked by the invention of photography [1839]. [Euclid did examine the nature of seeing with two eyes and perceiving three dimensions in his treatise "Optics".] This enabled Wheatstone's invention to be combined with photographs (instead of hand-drawn images) resulting in a stunning capture of the real world in fine detail and great depth. Such was the rage that stereo-viewers were produced commercially in the 1850s and the London Stereoscopic Society was founded in 1893. Even today there is remnant interest in 3D photography. (Viewmaster stuff.)

More recently, anaglyphs were introduced where the viewer is required to wear a pair of cardboard glasses which had a red filter over one eye and a blue or green filter over the other eye. Films were made so that audiences with these glasses could experience a 3D effect.

Holography was the next development though its progress was hampered by the technical difficulties of the process. Polarization has been used to deliver different images to each eye, either using polarized light and filters orientated in perpendicular directions or using liquid crystal filters which alternately block the eyes of the viewer in time with the showing of left and right images.

Last year, the Pulfrich phenomenon was used in some TV programmes for 3D effects. This effect requires one eye to be covered with a fairly dark filter and relies on the psychological fact that the brain appears to process dim images at a slower pace than brighter ones. Thus with a moving series of pictures (such as a video sequence on TV) earlier frames from the filtered eye are fused with the present frames from the naked eye.

3.2 RDS and SIRDS

But the invention of the autostereogram is of most interest here. The history of this began at Bell Labs with the research of Bela Julesz into the psychological problem of vision and perception. He discovered [1959] the human brain could perceive depth information from a pair of random dot stereograms (RDS).

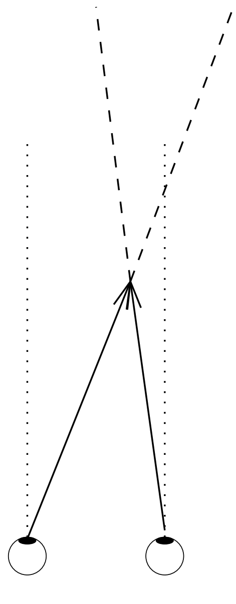

Figure 1: Rays from eyes converge onto object

The importance of his result was that without any apparent visual clues, the human brain could reconstruct the 3D information. In someone with normal binocular vision, each eye has a slightly different view of the world around because they are separated by about 6–7cm. When both eyes look at a distant object, two processes cooperate to bring two images into the brain.

The first process directs the eyes to the object (so that the images are on the centres of the retinae) and the second focuses the individual images. For an object directly in front of the face, the first process turns the eyes inwards towards it (if it is assumed they start off pointing in a perpendicular direction from the face) and the process is known as convergence. See figure 1.

The brain must then fuse the different images together to produce the perception of depth; this fusion is known as stereopsis. Julesz with Miller showed that the sense of depth could arise solely from stereopsis, using patterns of random dots.

The next step came with the discovery that a separate pair of images is not needed for stereopsis. Tyler and Clarke [1979] made the first single-image random dot stereograms (SIRDS). They relied on the "wallpaper effect," discovered by Sir David Brewster [1844] who noticed a 3D effect from staring cross-eyed at wallpaper, see figure 2. This effect is due the brain matching up different units of the repeating block pattern in the wallpaper. The 'mismatch' translated into perception of the plane of the wallpaper at a closer distance than what it really was.

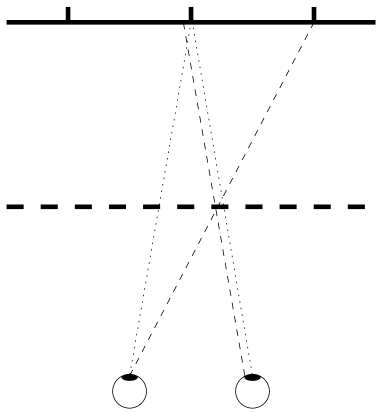

Figure 2: The wallpaper effect: the horizontal lines represent the wallpaper (solid line with ticks to show the edges of each strip) and apparent wallpaper (dashed line); the thin lines emerging from each eye show the direction in normal viewing mode (dotted) and cross-eyed mode (dashed).

The important detail was that the apparent depth was controlled and determined by the period of repetition of the wallpaper. Tyler and Clarke using an Apple II computer wrote a BASIC program to generate a repeating image in which the 3D information was embedded by changing the repetition interval in parts of the image. The result was a mess of random dots which only yielded its secret when viewed in the correct manner (cross-eyed).

The same technique was later applied by other people to coloured patterns and images thus creating single-image stereograms (SIS) or autostereograms.

3.3 Viewing stereograms and autostereograms

Whereas with stereo pairs, be they photographs or random dots or drawings, the two images needed to be brought to the eyes separately using a stereoscope, anaglyph glasses, LCD glasses or any other equipment, in contrast autostereograms can be seen with the naked eyes.

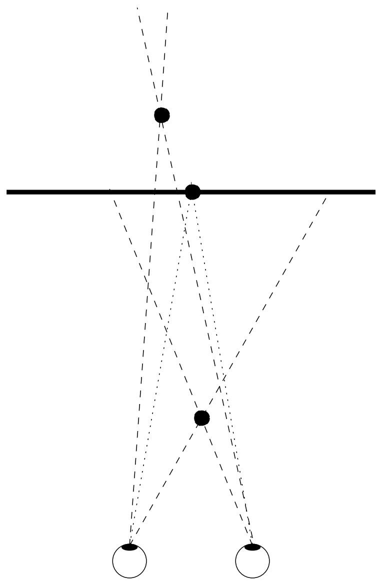

Figure 3: Wide-eyed and cross-eyed viewing

The only disadvantage is that the eyes must lock on to different parts of the repeating pattern. There are two techniques available: wide-eyed (deconverged) and cross-eyed viewing.

Both required the decoupling of the two processes mentioned above that man utilizes to send useful images to the brain. The convergence mechanism must be separated from the focussing which remains fixed on the flat image.

The two viewing methods differ only in the position of the point at which hypothetical lines drawn from each eye, in the direction of their targets, intersect (i.e. the point of convergence). For wide-eyed viewing, the convergence point is behind the autostereogram and for cross-eyed viewing, it is in front; see figure 3. The decoupling can take some practice to achieve because it is so against what is the unconscious norm.

3.4 Examples and applications of stereopsis

Here are just a few uses of stereopsis:

- Everyday life(!), especially in judging distance and size.
- Data extraction, eg. aerial photography with stereo camera to measure heights and find contours of landscapes.
- Object detection (spot the difference), eg. the technique Carolyn Shoemaker uses to spot comets. Using two similar images as stereo disparity, anomalies will appear 'greyed out' or on a different depth level from the constant background.
- Virtual reality (constructing the imaginary), eg. VR headsets and autostereograms.

4 SIRDS and SIS

Here is an explanation of how autostereograms work and are made. It is assumed that the wide-eyed technique is used for viewing. The paper by Thimbleby et al. [2] explains past and present algorithms but the perspectively correct method described here is new.

4.1 Basic geometry

As is the usual practice in three-dimensional computer graphics, one starts with a coordinate system. In the present case, it has a right-handed set of axes and its origin lies at a distance OS behind the centre of the autostereogram. Let the x direction point right, the y direction up and z towards the viewer. The viewer, whose eyes are ES apart, stands a distance VD away. A plan view of the defined geometry is shown in figure 4.

For a point with coordinates (x, y, z), rays are drawn directly to the left and right eyes. These rays intersect the plane where the autostereogram lies with x coordinates l and r. So to make an autostereogram, the two points of

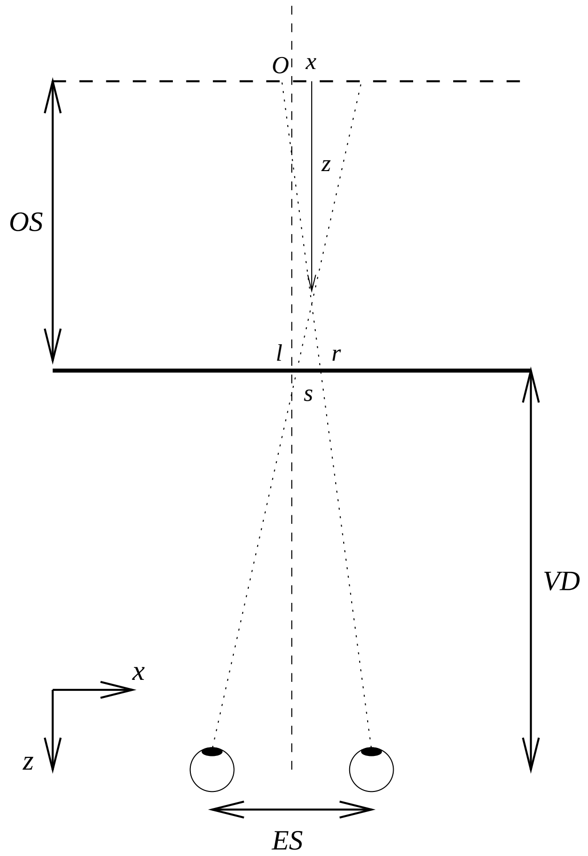

Figure 4: Plan view of the geometrical configuration of an autostereogram

intersection must hold the same visual information so that both eyes receive identical data, i.e. the points have possess the same colour.

Let s be the point separation so that $r = l + s$. By the law of similar triangles [a mathematical law which states that for two triangles which have the same interior angles, the sides opposite each angle are related by a fixed ratio], the following relation can be derived

$$\frac{s}{OS - z} = \frac{ES}{VD + OS - z}\,. \tag{1}$$

This is rewritten as

$$s = ES\,(1 - z') \tag{2}$$

where the new variable z' is defined by

$$z' = \frac{VD}{VD + OS - z}\,. \tag{3}$$

Notice that as the point approaches the viewer (as z increases) the separation s decreases.

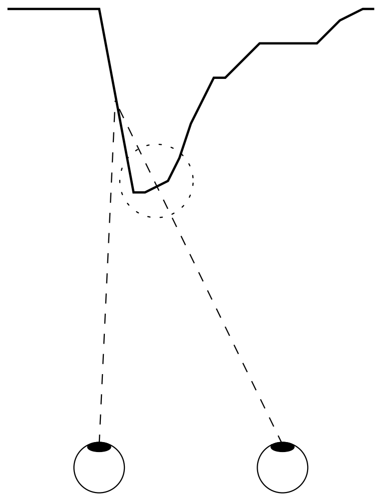

Figure 5: Example of a hidden surface

To make an autostereogram, all of the pixels [picture elements: computer jargon for the small dots that make up a picture on either a monitor screen or some physical media such as paper] (of the final image) must be matched to their stereo partner. The procedure to take is to scan across each line of the image and work out the matching pairs using some depth information. There are several ways to do this; here, only the z-buffer method is explained.

4.2 z-buffer

In computer graphics, one way of describing three-dimensional objects is to approximate its surface with a net of planar polygons. There are many ways to depict the object including drawing the outline of each polygon and filling the interior of each polygon with constant or varying colour tones. The problems involved in such ways include determining which polygon lies in front of another from a certain viewpoint.

The z-buffer method uses an array to store and check the z values of points within a polygon before applying colour. In making autostereograms, only the depth information is required.

Assuming the buffer is filled with appropriate information — in this case, the z-buffer is a 2D array with each slot containing the highest z value for a particular (x, y) point on the screen — it is repeatedly scanned in the x direction, for every y value, and matching points are marked out. Once a horizontal line is traversed, colours are assigned to each pixel in the line subject to the constraints of the matching-pairs list and the line of pixels is drawn on to the screen or written to a file for later use.

There are quite a few methods for implementing this scheme. The simplest way is to ignore the actual geometry and assume that each point lies centrally between your eyes. Then the matching points are given by $x \pm s/2$. The advantage of this method is the speed of calculation due to its simplicity, but its disadvantage is twofold. Apart from the obvious distortion, the possible circumstance of a surface that is only visible to one eye is ignored. For example, see figure 5. That is, points in a hidden surface would be erroneously included in the matching list.

Nevertheless, for pedagogical reasons, some C code examples of this scheme are shown. See the next section for more details.

4.3 Geometrical correctness

A more complicated scheme involves using geometrical perspective. The algorithms given in [2] are conceded to be not truly free from geometrical distortions. Here a new algorithm is explained.

Basically, to incorporate perspective every point in space is projected on to the plane where the image lies. Again using the law of similar triangles, a point (x, y, z) is projected to a point (x', y') on the screen where these new coordinates are related by

$$x' = z'x \qquad \text{and} \qquad y' = z'y\,, \tag{4}$$

and the variable z' was defined above, in Eq. (3).

One way to check for hidden surfaces is to trace the rays back from the eyes and see if they converge to the same point. The disadvantage with this method is that it is slow. A much quicker method stems from the way a z-buffer is used in computer graphics.

With polygonal data for an object, the z-buffer can be filled with z values of all the points interior to a polygon using linear interpolation (from the vertex points supplied) because the polygon is planar and obeys the following equation for a plane

$$z = \alpha x + \beta y + \gamma\,, \tag{5}$$

where α, β & γ are constant for a given polygon.

There is a snag that this does not work for perspective projection, ie. using (x', y', z). In this case, another function of z is used to fill the 2D array. A bit of algebra shows that if z' is used and if Eq. (5) holds then the following is true

$$z' = \alpha' x' + \beta' y' + \gamma' . \tag{6}$$

This is known as the inverse z-buffer method because z' is a rational function of z, Eq. (3).

The inspirational idea to find hidden surfaces is due to Luke Hutchison in private communication with the author. He hit upon the neat idea of using two (instead of one) z-buffers, one for each eye.

His method is to fill each buffer with z values corresponding to the view from each eye then, using the separation function $s(z)$, work out which slot in the right array matches a slot in the left array and compare the z values contained in those slots. If they are the same numerical value (within a given tolerance) then the point represented by the slots is visible to both eyes. As shown in figure 5, a partially hidden point would be seen to possess different z values; the right eye perceives the point having a higher z value than the the left eye does because it is actually seeing a different point.

With the refinement of perspective, this method is much faster than the hidden surface checking mentioned in the Thimbleby paper [2]. The disadvantages include computer memory constraints due to the fact that two large buffers are used, but this can be circumvented by using smaller buffers and processing the image in horizontal strips.

Implementing this method is simple. The only difference is that the projected x values of each point are different because the eyes are separated by a horizontal gap. The projected x values are

$$l = x' - s/2 \qquad \text{and} \qquad r = x' + s/2 . \tag{7}$$

Note that since s is a linear function of z', see Eq. (2), then equation (6) still holds when l or r is substituted for x'. To summarize, for each point (x, y, z) the buffers are filled with the projected points (l, y', z') and (r, y', z').

Some C code shows the scheme in action.

4.4 Software Generators

Apart from the z-buffer method, the only way to make stereograms is to modify ray-tracers to output depth information (instead of colour).

Ray-tracing is a well-known method of generating realistic computer graphics where, as the name implies, rays are traced from the eye(s) towards the scene, reflecting off objects therein and back to the sources of light. The computer program can be easily changed to trace the rays back to the first object encountered and return with the depth information.

This method is also free from geometrical distortions. Commercial stereograms are generally done with modified ray-tracers.

Most stereogram software available rely on the z-buffer method. In these, the user is required to compose a scene in a variety of methods. Most let the user draw the depth information directly with standard graphics-package tools such as polygonal, elliptical and curved outlines and boundary fills. The depth information is represented as colour, with white as the closest level, black the furthest and greys for intermediate levels. Usually the fills are done with a constant colour though there are exceptions to this that allow gradated fills.

Though this direct approach allows easy manipulation of the scene by the user, the disadvantage is the difficulty of obtaining a realistic 3D scene. Other programs allow the user to handle objects derived from three dimensional data. The problem with this is the limited availability of this data, unless the user has access to a CAD package [Computer Aided Design: used by architects, product designers and engineers to draw 2D and 3D objects and scenes] and/or a large database of pre-designed objects.

Obviously, the ideal program would allow the user to do both these things.

4.5 Limitations of SIRDS

Apart from the initial difficulties in seeing the first stereogram, there is an inherent limitation to using single-image stereograms as representations of three-dimensional scenes.

The fact that a computer-generated image will in general consist of pixels implies that the separation of matching points will be discretized to units of pixels. This imposes a considerable constraint on the realism of the depth seen in a stereogram because discrete steps in the separation means discrete steps in the perceived depth. That is, a layering effect is seen in the stereogram.

Using the above equations, the number of layers available is

$$layers = s(z_{\min}) - s(z_{\max}) = \frac{2z_{\max} ES\ VD}{(VD + OS)^2 - z_{\max}^2} \tag{8}$$

where $z_{\min} = -z_{\max}$.

Typical values of the parameters are $ES = 65 * \text{pmm}$, $OS = 150 * \text{pmm}$, $VD = 300 * \text{pmm}$, where pmm (pixels per millimetre) is the pixel resolution of the output device — a monitor has around 100dpi (dots per inch), which means pmm ≈ 4.0.

For a object to fit within the image and appear with the right proportions, its diameter must be less than the height or width of the stereogram, whichever is smaller. So for a stereogram of size 1024x768 (in pixels), $z_{\max} = 384$. Therefore, $layers = 77$. This is a low figure though with a greater size (hence a higher resolution — to make the stereogram retain a reasonable physical size) it could be much bigger.

So on a computer screen, stereograms looks noticeably layered. With the higher resolution provided by a laser printer, the printed image is much smoother.

There is a way to reduce the layering effect. This is similar to techniques used to make straight lines in computer graphics appear less jagged.

Briefly, in that situation intermediate colours are used to smooth out the steps. So in coloured stereograms, instead of matching the colour of pixels, the fractional part of the separation value $s(z)$ is used to interpolate the colour between adjacent pixels. This technique works with the side-effect of leaving the resultant image looking a little fuzzy, indicating an effective loss of resolution.

5 Example code

Two C programs are listed below. Figure 6 shows some sample output from ex1.c.

5.1 ex1.c

```
/*
 * Example C code 1, ex1.c for simple SIRDS scheme
 *
 * (C) Copyright 1994 By Peter Chang <peterc@v2.ph.man.ac.uk>
 *
 * The output is a SIRTS (single image random text stereogram)
 * of a sphere, though the aspect ratio of text characters
 * will make this look like an ellipsoid.
 *
 * Due to the limited resolution of using text characters, the result
```

```
 * will be very layered.
 * It looks like a pile of discs of decreasing size.
 *
 * To compile under UNIX, use something like this:
 *   cc -ansi -O ex1.c -lm
 *
 * Things to be done by you, the reader, include putting in
 * the graphics stuff, some error checking and a user interface --
 * whether it be a straightforward command line interface or
 * a fancy GUI.
 *
 * It is assumed that the graphics library uses the following
 * coordinate system: upper left is the origin (0,0) and x,y increase
 * in the rightward and downward direction, respectively.
 *
 * Possible problems include the restrictive memory architecture
 * of personal computers and their operating systems.
 * For example, the IBM PC and its clones running {MS,DR,PC}-DOS
 * only allow a program to access 640K of memory directly.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VD  (300.0*pmm) /* Viewing distance from screen (in pixels) */
#define ES  (65.0*pmm)  /* Eye separation */
#define OS  (150.0*pmm) /* Offset into screen */

#define MYLOW -32000
#define MYHIGH 32000

/*
 * This define statement controls the number of different text
 * characters used by the program to make the SIRTS
 * If you want to use just two characters ('!' and '"')
 * then comment it out.
 */
```

```
#define MANYCHARS

typedef struct {
   int t;
   int f;
} Llist; /* Link list structure, contains 'to' and 'from' indices */

int width, height; /* dimensions of final picture, in pixels */

/*
 * this is set for ASCII text output with a fixed width of 6 pixels
 * (ie. an xterm with 'fixed' font) on a monitor with 100dpi resolution
 */
float pmm = 0.656;   /* resolution in pixels per mm */

unsigned int *dtos = NULL;
                     /* pointer to lookup table for depth to separation */
int density = 127; /* density of dots, out of 255 */


int main()
{
   int **ztop = NULL; /* z-buffer */

   void zfillall(int **zbuf);
   void drawsirds(int **zbuf);
   int **zmema(void);
   void zmemf(int **zbuf);

   width = 80; /* set these two values to whatever you want */
   height = 30;
   ztop = zmema();
   zfillall(ztop);
   drawsirds(ztop);
   zmemf(ztop);
   exit(EXIT_SUCCESS);
}


/*
```

```
 * Defines z-values for all objects using discretised height fields
 */
void zfillall(int **zbuf)
{
   int x, y;
   int *zll;
   int bv, iv;

   void drawobj(int **zbuf);

   iv = MYLOW;

   for (y=0; y<height; y++) { /* set background to const initially */
      zll = zbuf[y];
      for (x=0; x<width; x++) {
         *zll++ = iv;
      }
   }

   drawobj(zbuf);

   bv = 0;
   for (y=0; y<height; y++) { /* reset background to 0 */
      zll = zbuf[y];
      for (x=0; x<width; x++, zll++) {
         if (*zll == iv) *zll = bv;
      }
   }
}


/*
 * as an example, let's fill in the z-values of hemisphere of radius
 * height/2
 */
void drawobj(int **zbuf)
{
   int ox, x, y, z, t;
   register int oy, tx, ty;
   int *zline;
```

```
   int hemisphere(register int tx, register int ty, register int oy);

   ox = width/2;
   oy = height/2;

   ty = oy;
   for (y=0; y<height; y++, ty--) {
      zline = zbuf[y];
      tx = -ox;
      for (x=0; x<width; x++, tx++, zline++) {
         z = *zline;
         t = hemisphere(tx, ty, oy);
/* this line is all important, it's the essence of a z buffer */
         if (t > z) *zline = t;
      }
   }
}


/*
 * a hemisphere function,
 * you can try one of your own instead
 */
int hemisphere(register int tx, register int ty, register int oy)
{
   float tt;

   tt = oy*oy - tx*tx - ty*ty;
   return (tt > 0.0) ? (int) sqrt(tt) : 0;
}


/*
 * generate and output SIRDS
 */
void drawsirds(int **zbuf)
{
   int x, y, pixpos;
   int *zline;
```

```
   Llist *same; /* doubly-linked list of matching points */
   unsigned char *pixarray; /* holds the colours of a row */
   unsigned int *nowdtos = NULL; /* pointer of somewhere in lookup table */
   unsigned int colour;

   unsigned int *lookup (void );
   void sirdssimp(int *zline, Llist *same, unsigned int *nowdtos);
   /* generic graphics routine */
   void DrawPoint(int x, int y, unsigned int colour);

   srand(time(NULL));
   same = (Llist *) malloc(width*sizeof(Llist));
   pixarray = (unsigned char *) malloc(width*sizeof(char));

   nowdtos = lookup();

   /* this is to show the repetition of the background */
   y = *nowdtos;
   printf("Repetition value %d\n", y);
   for (x=0; x<width; x++) {
      printf(((x%y) == 0) ? "x" : " ");
   }
   printf("\n");

   for (y=0; y<height; y++) {
      zline = zbuf[y];
      for (x=0; x<width; x++) { /* initialise matching list */
         same[x].t = x;
         same[x].f = x;
      }
      sirdssimp(zline, same, nowdtos);
      for (x=0; x<width; x++) {
         pixpos = same[x].f;
         colour = (pixpos != x) ? pixarray[pixpos] :
#ifdef MANYCHARS
         /* get a random value from 0 to 255 */
            (rand() & 0xff);
#else
         /* use this for two colours, at a set density */
            ((rand() & 0xff) > density);
```

```
#endif
         pixarray[x] = colour;
         DrawPoint(x, y, colour);
      }
   }
   free(same);
   free(pixarray);
   free(dtos);
}


/*
 * a lookup table for calculating the depth to shift info
 */
unsigned int *lookup (void )
{
   int i, range;
   unsigned int *look;
   float qfac, oss, vd;

   oss = 2.0*OS;
   qfac = ES;
   range = (int) (oss+0.5);
   vd = VD + oss;
   if (dtos != NULL) free(dtos);
   dtos = (unsigned int *) malloc (range*sizeof(int));

   look = dtos;
   for (i=0; i<range; i++, look++) {
      *look = (unsigned int) (qfac*(oss-i)/(vd-i)+0.5);
   }
   return dtos+(range/2); /* return pointer to lookup table */
}


/*
 * simple SIRDS scheme, no hidden surface removal
 *
 * the algorithm derives from the Thimbleby paper
 */
```

```
void sirdssimp(int *zline, Llist *same, unsigned int *nowdtos)
{
   register int x, z0, s;
   register short left, right;
   register short st, sl, sr;

   for (x=0; x<width; x++) {
      z0 = *zline++;
      /* works out separation from depth using lookup table */
      s = *(nowdtos+z0);
      left = x - s/2;
      right = left + s;
      if (left >= 0 && right < width) {
         /* now we put in to & from pointers for matching points */
         sl = left;
         sr = right;
         for (st=same[sr].f; st!=sl && st!=sr; st=same[sr].f) {
            if (st > sl) {
               sr = st;
            } else {
               same[sl].t = sr;
               same[sr].f = sl;
               sr = sl;
               sl = st;
            }
         }
         same[sl].t = sr;
         same[sr].f = sl;
      }
   }
}


/* memory (de)allocation for zbuffer */
int **zmema(void)
{
   int i;
   int **zbuf;

   zbuf = (int **) malloc(height*sizeof(int *));
```

```
   for (i=0; i<height; i++)
      zbuf[i] = (int *) malloc(width*sizeof(int));
   return zbuf;
}


void zmemf(int **zbuf)
{
   int i;

   if (zbuf != NULL) {
      for(i=0; i<height; i++) free(zbuf[i]);
      free(zbuf);
   }
}

/*
 * Draw a point of a given colour
 *
 * This is just ASCII output at the moment, ie. a SIRTS output
 */
void DrawPoint(int x, int y, unsigned int colour)
{
   int c;

   /* there are 94 printable ASCII characters from '!' to '~' inclusive */
   c = (colour % 94) + '!';
   printf("%c", c);
   if (x==width-1) printf("\n"); /* fudge to go to next line */
}
```

```
Repetition value 14
x             x             x             x             x             x
D<_!6qDI5N]<`]D<_!6qDI5N]<`]D<_!6qDI5N]<`]D<_!6qDI5N]<`]D<_!6qDI5N]<`]D<_!6qDI5N
zxSP0'|`oX)"CLzxSP0'|`oX)"CLzxSP0'|`oX)"CLzxSP0'|`oX)"CLzxSP0'|`oX)"CLzxSP0'|`oX
7zW|Hz.'X6AZ!%7zW|Hz.'X6AZ!%7zW|Hz.'X6AZ!%7zW|Hz.'X6AZ!%7zW|Hz.'X6AZ!%7zW|Hz.'X6
?QzM"Cj-$@fH[??QzM"Cj-$@fH[??QzM"Cj-$@fH[??zM"Cj-$@ffH[??zM"Cj-$@ffH[??zM"Cj-$@f
qn')ts]7Apj8\Qqn')ts]7Apj8\Qqn')ts]7Apj8\qn')ts]7Apj8\\qn')ts]7Apj8\\qn')ts]7Apj
}vVeLVKGc59lJN}vVeLVKGc59lJN}vVeLVKGc59lN}vVeLVKGc59lN}}vVeLVKGc59lN}}vVeLVKGc59
Wk>;]4{`'a4mGLWk>;]4{`'a4mGLWk>;]4{`'a4GLWk>;]4{`'a4GLWkk>;]4{`'a4GLWkk>;]4{`'a4
XkbWg7F"'Ji>NUXkbWg7F"'Ji>NUXkbWg7F"'J>NUXkbWg7F"'J>NUXkbbWg7F"'J>NUXkbbWg7F"'J>
M=wOC(n@H`s5B{M=wOC(n@H`s5B{M=wOC(n@Hs5B{M=wOC(n@Hs5B{M=wOOC(n@Hs5B{M=wOOC(n@Hs5
1Y:J_{\|;g2W{_1Y:J_{\|;g2W{_1Y:J_{\|g2W{_1Y:J_{\|g2W{_1Y:J__{\|g2W{_1Y:J__{\|g2W
P2c+F!>Z|V%E`/P2c+F!>Z|V%E`/P2c+F!>ZV%E`/P2c+F!>ZV%E`/P2c+FF!>ZV%E`/P2c+FF!>ZV%E
'P]"whL"e!i"jj'P]"whL"e!i"jj'P]"whLe!i"jj'P]"whLe!i"jj'P]"whhLe!i"jj'P]"whhLe!i"
dZINl[q^|F@CdGdZINl[q^|F@CdGdZINl[q|F@CdGdZINl[q|F@CdGdZINl[[q|F@CdGdZINl[[q|F@C
4j^=,7Pau^3+[v4j^=,7Pau^3+[v4j^=,7Pu^3+[v4j^=,7Pu^3+[v4j^=,77Pu^3+[v4j^=,77Pu^3+
:~Z8`'f0bL;wJI:~Z8`'f0bL;wJI:~Z8`'fbL;wJI:~Z8`'fbL;wJI:~Z8`''fbL;wJI:~Z8`''fbL;w
*$3AvF$[!|T:#H*$3AvF$[!|T:#H*$3AvF$!|T:#H*$3AvF$!|T:#H*$3AvFF$!|T:#H*$3AvFF$!|T:
0^ZN/%6#</J{B!0^ZN/%6#</J{B!0^ZN/%6</J{B!0^ZN/%6</J{B!0^ZN/%%6</J{B!0^ZN/%%6</J{
2|?<YWUN)H@gV52|?<YWUN)H@gV52|?<YWU)H@gV52|?<YWU)H@gV52|?<YWWU)H@gV52|?<YWWU)H@g
][`I009cW1:flR][`I009cW1:flR][`I009W1:flR][`I009W1:flR][`I0009W1:flR][`I0009W1:f
n,)*M};E+SoH_2n,)*M};E+SoH_2n,)*M};+SoH_2n,)*M};+SoH_2n,)*M}};+SoH_2n,)*M}};+SoH
P3Ca+h>oq~U*.8P3Ca+h>oq~U*.8P3Ca+h>o~U*.8P3Ca+h>o~U*.8P3Ca++h>o~U*.8P3Ca++h>o~U*
_:JaIwNjkUoMM0_:JaIwNjkUoMM0_:JaIwNjUoMM0_:JaIwNjUoMM0_:JaIIwNjUoMM0_:JaIIwNjUoM
3/Sqj@+1TTV9h^3/Sqj@+1TTV9h^3/Sqj@+1TV9h^3/Sqj@+1TV9h^3/Sqqj@+1TV9h^3/Sqqj@+1TV9
ADJoXS0:TqIt?YADJoXS0:TqIt?YADJoXS0:Tqt?YADJoXS0:Tqt?YADJJoXS0:Tqt?YADJJoXS0:Tqt
y4`cyUujFxv`*Ty4`cyUujFxv`*Ty4`cyUujFxv*Ty4`cyUujFxv*Ty44`cyUujFxv*Ty44`cyUujFxv
B?U@tZ'6H,NQdyB?U@tZ'6H,NQdyB?U@tZ'6H,NQyB?U@tZ'6H,NQyBB?U@tZ'6H,NQyBB?U@tZ'6H,N
jc}~oSUI{4,->Djc}~oSUI{4,->Djc}~oSUI{4,->jc}~oSUI{4,->>jc}~oSUI{4,->>jc}~oSUI{4,
idm@s3$JLR[!@`idm@s3$JLR[!@`idm@s3$JLR[!@`im@s3$JLR[[!@`im@s3$JLR[[!@`im@s3$JLR[
([6GYEb)gx+WoD([6GYEb)gx+WoD([6GYEb)gx+WoD([6GYEb)gx+WoD([6GYEb)gx+WoD([6GYEb)gx
`Q#.8;o4N4sK\B`Q#.8;o4N4sK\B`Q#.8;o4N4sK\B`Q#.8;o4N4sK\B`Q#.8;o4N4sK\B`Q#.8;o4N4
```

Figure 6: Sample from ex1.c

5.2 ex2.c

```
/*
 * Example C code 2, ex2.c for correct geometry SIRDS scheme
 *
 * (C) Copyright 1994 By Peter Chang <peterc@v2.ph.man.ac.uk>
 *
 * This is incomplete code showing a sketch of the ideas
 * introduced in the text.
 *
```

```
 * Only the elements that differ from ex1.c are given.
 * More importantly, filltriangle() is also missing.
 *
 * The main deviation from the text is that the two buffers store
 * values of s, which decrease with increasing z.
 * So the buffer fills done by filltriangle() must update each slot
 * only if the new s is less than the old value.
 */

/* sets the tolerance for the comparison of s values */
#define TOLERANCE 1

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VD  (300.0*pmm) /* Viewing distance from screen (in pixels) */
#define ES  (65.0*pmm)  /* Eye separation */
#define OS  (150.0*pmm) /* Offset into screen */

#define MYLOW -32000
#define MYHIGH 32000

#define MANYCHARS

typedef struct {
   int t;
   int f;
} Llist; /* Link list structure, contains 'to' and 'from' indices */

int width, height; /* dimensions of final picture, in pixels */
float pmm = 0.656;   /* resolution in pixels per mm */
int density = 127; /* density of dots, out of 255 */


int main()
{
   int **ztopl = NULL, **ztopr = NULL; /* s-buffers */
```

```
   void sfillall(int **zbufl, int **zbufr);
   void drawsirds(int **zbufl, int **zbufr);
   int **zmema(void);
   void zmemf(int **zbuf);

   width = 80; /* set these two values to whatever you want */
   height = 50;
   ztopl = zmema();
   ztopr = zmema();
   sfillall(ztopl, ztopr);
   drawsirds(ztopl, ztopr);
   zmemf(ztopl);
   zmemf(ztopr);
   exit(EXIT_SUCCESS);
}

/*
 * fills s-values for all objects
 */
void sfillall(int **zbufl, int **zbufr)
{
   int x, y;
   int *zll, *zlr;
   int bv, iv;

   void sdrawobj(int **zbufl, int **zbufr);

   iv = MYHIGH;

   for (y=0; y<height; y++) { /* set background to const initially */
      zll = zbufl[y];
      zlr = zbufr[y];
      for (x=0; x<width; x++) {
         *zll++ = iv;
         *zlr++ = iv;
      }
   }

   sdrawobj(zbufl, zbufr);
```

```
    /* z-buffer now contains values of s */
    bv = (int) ES*(OS+height/2.0)/(VD+OS+height/2.0); /* z = 0 */

    for (y=0; y<height; y++) { /* reset background */
        zll = zbufl[y];
        zlr = zbufr[y];
        for (x=0; x<width; x++, zll++, zlr++) {
            if (*zll == iv) *zll = bv;
            if (*zlr == iv) *zlr = bv;
        }
    }
}


/*
 * this time let's do a triangle
 * notice that we have two buffers
 */
void sdrawobj(int **zbufl, int **zbufr)
{
    int xa, ya, xb, yb, xc, yc;
    float sa, sb, sc, za, zb, zc;
    float es, oy, vd, vdos;
    float ax, ay, az;
    float bx, by, bz;
    float cx, cy, cz;

    /*
     * The interpolated z-buffer fill is done by this function.
     * Its inputs are the coordinates of the vertices of a triangle
     * and the z-buffer.
     */
    int filltriangle(int x1, int y1, float z1, int x2, int y2, float z2,
                int x3, int y3, float z3, int **zbuffer);

    es = ES;
    vd = VD;
    vdos = vd + OS;
    oy = height/2.0;
```

```
   /* let's do an isoceles triangle */
   ax = 0.8*oy; ay = 0.0*oy; az = 0.0*oy;
   bx = -0.7*oy; by = 0.6*oy; bz = 0.3*oy;
   cx = -0.7*oy; cy = -0.6*oy; cz = -0.3*oy;

   za = vd/(vdos - az); /* z' */
   xa = za*ax; /* x' */
   ya = za*ay; /* y' */
   sa = es*(1 - za); /* s */
   xa -= sa/2; /* xL */

   zb = vd/(vdos - bz);
   xb = zb*bx;
   yb = zb*by;
   sb = es*(1 - zb);
   xb -= sb/2;

   zc = vd/(vdos - cz);
   xc = zc*cx;
   yc = zc*cy;
   sc = es*(1 - zc);
   xc -= sc/2;

   /* fill left buffer then right */
   filltriangle(xa, ya, sa, xb, yb, sb, xc, yc, sc, zbufl);
   filltriangle((xa+sa), ya, sa, (xb+sb), yb, sb, (xc+sc), yc, sc, zbufr);
}


/*
 * generate and output SIRDS
 */
void drawsirds(int **zbufl, int **zbufr)
{
   int x, y, pixpos;
   int *zll, *zlr;
   Llist *same; /* doubly-linked list of matching points */
   unsigned char *pixarray; /* holds the colours of a row */
   unsigned int colour;
```

```
   void sirdsnew(int *zll, int *zlr, Llist *same);
   /* generic graphics routine */
   void DrawPoint(int x, int y, unsigned int colour);

   srand(time(NULL));
   same = (Llist *) malloc(width*sizeof(Llist));
   pixarray = (unsigned char *) malloc(width*sizeof(char));

   /* no need for lookup table */

   for (y=0; y<height; y++) {
      zll = zbufl[y];
      zlr = zbufr[y];
      for (x=0; x<width; x++) { /* initialise matching list */
         same[x].t = x;
         same[x].f = x;
      }
      sirdsnew(zll, zlr, same);
      for (x=0; x<width; x++) {
         pixpos = same[x].f;
         colour = (pixpos != x) ? pixarray[pixpos] :
#ifdef MANYCHARS
         /* get a random value from 0 to 255 */
            (rand() & 0xff);
#else
         /* use this for two colours, at a set density */
            ((rand() & 0xff) > density);
#endif
         pixarray[x] = colour;
         DrawPoint(x, y, colour);
      }
   }
   free(same);
   free(pixarray);
}


/*
 * new SIRDS scheme, with hidden surface removal
 *
```

```
 * the new algorithm discussed in the paper
 */
void sirdsnew(int *zll, int *zlr, Llist *same)
{
   register int s;
   register short left, right;
   register short st, sl, sr;

   for (left=0; left<width; left++) {
      s = *zll++;
      right = left + s;
      if (right < width) {
         s -= zlr[right];
         /* check that both eyes can see the point */
         if (s <= TOLERANCE && s >= -TOLERANCE) {
            /* now we put in to & from pointers for matching points */
            sl = left;
            sr = right;
            for (st=same[sr].f; st!=sl && st!=sr; st=same[sr].f) {
               if (st > sl) {
                  sr = st;
               } else {
                  same[sl].t = sr;
                  same[sr].f = sl;
                  sr = sl;
                  sl = st;
               }
            }
            same[sl].t = sr;
            same[sr].f = sl;
         }
      }
   }
}
```

6 Notes and History of xpgs

The author has recently completed his doctorate in theoretical physics and is looking for a job.

I first came across SIRDS in the spring of 1991 when a friend's friend in the computer department showed us them. Though I couldn't see them then, I wrote some simple programs in C using the GKS graphics library from the outline code given to me.

My interest in SIRDS was rewaken by a friend, Gareth Richards, who found some on the net and printed them out in May 1993. From there, we took it upon ourselves to write a SIRDS program using proper 3D datasets in C with Xlib graphics.

By 15th July 1993, we released a free and simple SIRDS animation package called "xps". Soon after getting hold of the Thimbleby paper [2], we modified our program to implement his algorithms and released "xpgs," version 1.5 (the change of name was due to the existence of another free X package with the same name) on 26th October 1993.

Gareth also reworked the code to work on a PC and called the subsequent program "SIRDSAni". This is a shareware package, available from Platonic Software.

Both these programs and this article can be obtained from us. If you have net access and a WWW client then have a look at the pages given by the URL `http://h2.ph.man.ac.uk/gareth/sirds.html`. I can be contacted by email at `peterc@a3.ph.man.ac.uk` or `etzpc@thmech.nott.ac.uk`.

The "Stereogram-FAQ" [4] is a list of frequently asked questions (with answers) that are asked in the USENET newsgroup `alt.3d`.

References

[1] *Stereogram*, published in the UK by Boxtree Ltd with ISBN 0 7522 0924 8. ©1994 by Cadence Books, USA.

[2] H.W. Thimbleby, S. Inglis & I.H. Witten, "Displaying 3D Images: Algorithms for Single Image Random Dot Stereograms," Department of Computer Science preprint at University of Waikato, Hamilton, NZ. To be published in *IEEE Computer*.

[3] H.W. Thimbleby & C. Neesham, "How to play tricks with dots," *New Scientist* **140 #1894**, 26 (9 October 1993).

[4] "Stereogram-FAQ," compiled by S. Inglis. It can be found at the URL `http://www.cs.waikato.ac.nz/~singlis/sirds.html`. Stuart Inglis can be contacted by email, send your messages to `singlis@cs.waikato.ac.nz`.