Stereogram FAQ

Stereogram Creation

Subject: [21] How can I write my own programs?

There are several approaches to take to write a SIRDS program (we'll start with SIRDS and move on to SIS in the next section).

We have some facts that will help us write the program:

- We need two objects (pixels) for stereo vision (ie. 2 eyes)
- Eye convergence (where we look) informs us of it's 3D depth

To make a SIRDS we have to make sure (for each 3D point in the object) we have two pixels the same colour (say either black or white) at a particular distance apart, so that when we "look through" each of the pixels, we will see the corresponding pixel in 3D.

To calculate the relationship between the pixels is the *only* complicated stage. We use an array called 'same[]' which simply points to a pixel (in the same scan line) that has the same value.

The second "for x" loop does this. At each position in the object, calculate the dot separation, calculate where the left and right line of sight will intersect the image, and shuffle the array so there is a one to one link.

After we have this 'same[]' array we simply iterate over the array, picking a colour and propagating it's colour across the bitmap. And then the process is finished, the result: a Single Image Random Dot Stereogram.

```
#define round(X) (int)((X)+0.5)
#define DPI 72
#define E round(2.5*DPI)
#define mu (1/3.0)
#define separation(Z) round((1-mu*Z)*E/(2-mu*Z))
#define far separation(0)
#define maxX 256
#define maxY 256

void DrawAutoStereogram(float Z[][])
{
  int x, y;
  for( y = 0; y < maxY; y++ ) {
    int pix[maxX];
    int same[maxX];
    int s;
    int left, right;
    /* initialise the links */
    for( x = 0; x < maxX; x++ )
      same[x] = x;

    /* calculate the links for the Z[][] object */
    for( x = 0; x < maxX; x++ ) {
      s = separation(Z[x][y]);
      left = x - (s/2);
      right = left + s;
      if( 0 <= left && right < maxX ){
        { int k;
          for(k=same[left]; k!=left && k!=right; k=same[left])
            if( k < right )
              left = k;
            else {
              left = right;
              right = k;
            }
          same[left] = right;
        }
      }
    }

    /* assign the colors */
    for( x = maxX-1; x >= 0; x-- ) {
      if( same[x] == x ) pix[x] = random()&1;
      else pix[x] = pix[same[x]];
      Set_Pixel(x, y, pix[x]);
    }
  }
}
```

Subject: [22] Creation of SIS

kindly written by Pascal Massimino (massimin@clipper.ens.fr)

(As opposed to Subject21, where the creation of a SIRDS was based on a bitmap, here we have a ray-tracing approach. ftp the RaySIS program)

The first step in the generation of a SIS (Single Image Stereogram) is to transform the scene you want to render into a *depth field*. One interesting method is to scan your screen line by line and intersect objects with one ray (say using a ray-tracing assimilated method). But you can also *slice* your scene if it appears more convenient. A proper rescaling of your depth may also be useful when objects extend to far from (or to close to) the eyes, for this could make your SIS hard to be seen when finished.

Once you've got your depth field, this 3D information requires been encoded in the SIS using a repetitive pattern. You will need to set proper pixels to the same color, this color being taken from an initial pattern. The following sketch shows the pixels (marked with 'o') on the screen that will need be allocated with the same color. The initial ray is the one (passing right in the middle of your eyes) that was used to determine h, the depth related to the scanned pixel (*). Then, from the point of intersection, two rays have been drawn in the direction of the eyes. They determine position of the linked pixels 'o', separated by a distance dx.

```
                           initial
                             ray
                              |
       Eyes:        Left      |      Right
                      +<-------ES------>+            ES=eye separation
```

```
                        \       |       /
                         \   dx |      /
                          \ <---|---> /
Screen --------------------o----*----o-------------------
                         ^  \   |   /     ^
                         |   \  |  /      |
                        h|    \ | /       |
                         |     \|/        |H
                         -    ****        |
                          ***********     -
-------------------------*** object **--------- average plane
                        *************           in your scene
                      *****************
```

In your scene you must have a virtual *average plane*: every point laying on this plane will produce two pixels separated by a distance X on the screen, with X being the width of the initial pattern. This method is non-linear: dx/X*(ES-X)/(ES-dx)=h/H. One can nevertheless approximate this relation to the linear one: dx/X=h/H without your brain getting injured...

This operation needs been repeated for each pixel of the scan line to produce a field of distances dx. The hard part still remains intact: deform this pattern to match the correlations inherent in the formation of the 3D image.

Propagation/deformation:

The initial pattern is drawn, say, on the left of the screen. Then, every pixel of this pattern is redrawn at distance dx, on the right, and re-use the new pattern it produces as initial pattern, etc...

```
   initial    new pattern   ....     ......
   pattern    after 1st
              deformation
              (larger)
            2            2'        2''
  1          1'           1''
  +---------*+-----------*+--------*--------------- ...
  |         /           /
  |  dx    /|    dx    /
  +-->----/ +---->>---/
```

Point 1 goes to 1', which himself is mapped to point 1'', etc...

Problem:

The field dx may present discrepancies, discontinuities, due to objects edges, sides, etc... In the point where this occur are actually points that, in real vision, are only seen with ONE eye (eg. if your directive eye is looking just in the center of a small box, one side of this box will be seen by the other eye, only). They produce gaps or overlappings in the pattern deformation/ propagation. But you can ignore this overlapping or fill the gaps with what you want (the initial pattern for instance),for this points does not take part of the 3D-effect. As a drawback, this can cause *ghost-objects* to appear when you are not focusing on the right distance (that is: the angle between your eyes' sight direction is *nearly* good, but your lens did not catch the right focal distance).

Note:

Because dx is not an integer, but a real number,interpolation of colors is required to avoid pixel-level slices of the objects to be generated. Scene will then appear *smooth*.

You can also start the deformation/propagation from the right or the middle of your screen...

Animation:

Once you've produced stereograms (SIRDS, SIS, or SIRTS), you may create an animation out of the them. But some problems arise:

The pattern of the background is *not* fixed, because it's content *heavily* depends on the position of the objects in your scene. Each new frame will produce different background. There are some methods to damp this: let a part of stereograms untouched by deformations, free from objects, so your eyes have a *stable* part to catch in the animation. This work rather well with SIS if your using a deformation of pattern that started, for instance, from the left: this part of the stereograms will remain the same along the animation.

A more *biological* problem: the brain is not used to see objects moving without the textures, that *seem* tied to the object, moving with it. Especially with SIS, the objects rather appear to be moving under a colored piece of sheet than in front of you, but this is just a matter of acclimatization. Do you remember the first time you saw a stereogram ?

There still remains a mean to temper this effect: in fact, to gain the third dimension in your image, you dropped one degree of freedom (colors). But there still remains some latitude in the choice of the pattern you use. You can *choose* any colors you want in a pre-definite vertical strip of your stereogram. So, why don't you choose a 'pattern' which is, for instance, a classicaly ray-traced image of your object, whose horizontal position can be adjust to superimpose and match your object when the 3D-effect will take place ? The only restriction is that your object does not extend to much beyond the strip, for only a part of width less than X can be color-controlled by this mean.

Subject: [23] Multiple stereograms

Is it possible to generate a stereogram such that the image is dependent on the viewing rotation?

The short answer is YES! In a "normal" stereogram the constraints are only in the horizontal direction, but by assigning constraints in 2-dimensions instead of linearly across the image, it is possible. I believe the first time I saw this was an image by Tyler [to be referenced].

--*comment by John Olsen to Andrew Steer(follows)*
>Also I think it should be possible to create a stereogram which gives
>TWO images: one when viewed landscape and another when looked at
>portrait. It would however only be possible for certain patterns
>and NOT in general (your average real image or logo).

Typically, you can only do a small image, entirely contained in the first copy of the random buffer (50 pixels wide in your case). The "vertical" image is repeated, but it gets more and more distorted as you go across the page.

There are, as you say, limited things you can do which cover greater areas, but the limitations are rather severe. The quality of the results depends on how much error you're willing to put up with, as "fog" and uncertainty in the resulting image if you want both vertical and horizontal to be full page images.

Can you "tile" or "wallpaper" stereograms?

--*from the net*
Some people say YES!, others say NO!

What do I mean? Assume we have an image that looks like

```
+----+
|    |
| X  |
|    |
|    |
+----+
```

can the colours be assigned such that copies of the image can be placed adjacent to the original image like this:

```
+----+----+----+----+----+
|    |    |    |    |    |
| .  | .  | .  | .  | .  |
|    |    |    |    |    |
|    |    |    |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
| .  | .  | X  | .  |etc.|
|    |    |    |    |    |
|    |    |    |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
| .  | .  | .  |etc.|etc.|
|    |    |    |    |    |
|    |    |    |    |    |
+----+----+----+----+----+
```

so that there appears to be a **continuous** 3D surface?

Is it possible to see two *completely* different images by alternating between the "wall-eyed" and "cross-eyed" techniques?

Most definately! The problem that is encountered is if we want two different images to be seen, each pixel on the stereogram corresponds to *two* different positions, this is a form of 3D aliasing which people refer to as "fog" -- or more plainly "hard to see". Using a method that creates links between corresponding pixels in the image (such as the one in Subject 21) the links simply need to be updated for each 3D object.

People have tried a simple method to ameliorate this; when generating the stereogram alternate using a pixel for the wall-eyed or cross-eyed approaches, this will at least half the horizontal resolution. [Has anyone tried this alternating technique?]

Subject: [24] Losing the color

By using complementary colors for the left and right eye, is it possible to create a stereogram in which the 3D image "loses" it's color and appears in greyscale?

Yes! It can be done. Would anyone like to elaborate on this matter? :-)

Subject: [25] C code for windows

Version I

From: zcapl31@ucl.ac.uk (William Andrew Steer)

Newsgroups: alt.3d
Subject: Constructing SIRDS, Windows source code MK1
Summary: Most basic program to draw SIRDS, written in C++ for Windows
Date: Tue, 31 May 1994 11:06:20 GMT

This is about the simplest Windows program for drawing SIRDS. It is only bare-bones, you'll have to modify the program for alternative depth sources, and the SIRDS is reconstructed from scratch after every WM_PAINT message ie whenever the window is resized or uncovered. Use CTRL+ALT+DEL to exit while it's drawing.

If you don't program in C, just look at the TMyWindow::Paint function. You should be aware that the random(arg) function returns an integer between 0 and arg-1.

If you have Turbo C++ then make a copy of one of the example project files in the /tcwin/owl/examples subdirectory, and copy the program below to your /examples subdirectory. Open Turbo C++, load the new project, and change it's contents to include just the program below and OWL.DEF. It should then run ok.

[-- *later comments by Andrew Steer*
I would like to stress that it uses the 'lookback' algorithm, which has some limitations, namely:
- it assumes that the right eye looks perpendicular to the screen while the left eye looks slightly sideways (so the rays converge), when in reality both eyes should look inwards. This causes asymmetry in the image (which according to some sources makes it more difficult for some people to see) and results in near objects appearing marginally further right than far ones.]

```
// ObjectWindows SIRDS Program  (C) W.A. Steer 1994

// Simplest routine possible


// Picture not stored
// - is completely redrawn for each WM_PAINT

#include <owl.h>
#include <math.h>

const pattwidth=96;  // the basic repeat distance.
// On a 14" monitor and 640x512 display, 96 pixels
// represents about half the distance between the eyes.

const NumColors=4;


// Define the colors to use in form 0xbbggrrL
//  0x  signifies hex notation
//  bb  blue value, gg  green value, rr  red value
//  L tells the compiler the constant is Long ie 32bit

COLORREF cols[NumColors]=
{
 0x000000L,
 0x800000L,
 0xFF0000L,
 0x000080L
};



// ---------------  TMyWindow  ---------------

class TMyWindow : public TWindow
```

```
{
public:
 TMyWindow( PTWindowsObject AParent, LPSTR ATitle);

 virtual void Paint( HDC PaintDC, PAINTSTRUCT& PaintInfo );
};


TMyWindow::TMyWindow( PTWindowsObject AParent, LPSTR ATitle) :
              TWindow(AParent, ATitle)
{
 Attr.W=620;  // Set the default window size to 620x340
 Attr.H=330;
}


void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& )
{
 int pixels[700];

 int x,y;
 int h;    // height of 'features' above the background
 int l,pl; // lookback and previous lookback distances

 long r,s; // temporary storage for constructing sphere


 for (y=0; y < 300; y++)
 {
  for (x=0; x < pattwidth; x++)
  {
   pixels[x]=random(NumColors);
  }

  pl=pattwidth;

  for (x=pattwidth; x < 612; x++)
  {
   h=0; // by default the image is flush with the background

   // Calculate the height of a point on the sphere
   if ((y >= 36) && (y <= 164))
   {
    r=64*64-(y-100L)*(y-100L);
    if (r > 0)
    {
     s=r-(x-256L)*(x-256L);
     if (s > 0) h=sqrt(s)+64;
    }
   }

   // Calculate the lookback distance
   l=(int)(pattwidth-h/8.0+0.5);

   // if image has got deeper (new lookback is greater
   //  than old lookback distance) generate a new pixel,
   // otherwise repeat an old one
   if (l > pl)
    pixels[x]=random(NumColors);
   else
    pixels[x]=pixels[x-l];

   pl=l;
  }

  // Copy the image to screen
  for (x=0; x < 612; x++)
  {
   // use the colors defined at the top in cols[]
   SetPixel(PaintDC,x,y,cols[pixels[x]]);
  }
 }
}



// ----------------  TMyApp  ----------------

class TMyApp : public TApplication
{
public:
 TMyApp(LPSTR AName, HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
   : TApplication(AName, hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};

 virtual void InitMainWindow();
};


void TMyApp::InitMainWindow()
{
 MainWindow = new TMyWindow(NULL, Name);
}


int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
 TMyApp MyApp("Original SIRDS by W.A.Steer", hInstance, hPrevInstance,
              lpCmdLine, nCmdShow);
 MyApp.Run();
 return MyApp.Status;
}
```

Version II

From: zcapl31@ucl.ac.uk (William Andrew Steer)
Newsgroups: alt.3d
Subject: Windows/C++ SIRDS source code Mk.2
Summary: Minimal code to generate high quality SIRDS in Windows
Date: Thu, 9 Jun 1994 11:06:19 GMT

Windows SIRDS source code MK2 (C) W.A.Steer 1994

Getting the program running

For Borland C++ / Turbo C++ Windows users

Unless you have an complete knowledge of the whereabouts of the various include & system files on your hard disk and other essential parameters I suggest you do the following:

- save my program in your owl\examples\ subdirectory as 'sirds.cpp'
- make a copy of one of the project files in your owl\examples\ directory under the name 'sirds.prj' in the same directory
- load up C++, and open the new 'sirds.prj' file

- remove from it all the files other than 'owl.def'
- select project|add item and add my program, 'sirds.cpp'
- Try to run the program!!!

YOU MUST BE RUNNING WINDOWS IN AT LEAST 256 COLORS otherwise the oversampling won't work properly, and you may only get three color output.

Although the program is not short, it is still the very minimum required to do what it does within the Windows environment. (One day the .EXE file for an all-singing all-dancing user friendly masterpiece *may* appear somewhere deep in cyberspace!)

As supplied, the user interface is non-existent - the program itself must be changed to alter key parameters.

The pictured is redrawn from scratch on every WM_PAINT message - which takes some time... don't be afraid to use CTRL+ALT+DEL to abort a redraw, you'll get a blue background and the message 'SIRDS.EXE This program has stopped responding to the system...' press enter to accept, and the program will be terminated.

The object is defined mathematically within the program - currently a sphere surrounded by a ring, 'Saturn-like' and a test pattern at the top and linear depth scale - slope and large staircase at the bottom.

You can try changing the code which sets the depth for a given point for other objects using 2D or 3D math's and/or conditions (could be quite horrendous depending on the shape), or adapting it to import depth information from some 3D modeling program, suitable scientific data, or fractal code. I have created 3D Mandelbrots, a SIRDS Scanning Tunneling Microscope (STM) picture and have seen Atomic Force Microscope (AFM) images.

As it stands the program does not have features for saving or printing the output. You'll have to use the print-screen key to copy to clipboard and save from there, or import to some other package.

Conversion for other languages / operating systems

If you want to convert the program to run on something other than Windows, concern yourself primarily with the TMyWindow::Paint procedure as this contains the guts of the program; the rest is largely Windows housekeeping. (Note that some of the arrays are defined outside the Paint procedure (otherwise there is a tendency to run out of stack space), the main parameters are at the top of the program, and you will need to program a color palette).

```
// ObjectWindows SIRDS Program Mk2  (C) W.A. Steer 1994
//                              email: w.steer@ucl.ac.uk

// Picture not stored
// - is completely redrawn for each WM_PAINT

// Has saturn & rings

// Switch 'dohiddenrem' to TRUE to enable (slow) hidden surface removal

#include <owl.h>
#include <math.h>
#include <alloc.h>

int bkdepth=-800;       // depth of the background in pixels
long E=192;             // typical eye separation in pixels
int o=700;              // observer-screen distance in pixels
```

```
const oversam=6;        // oversampling ratio - set to 1,2,4, or 6
                        //    1 implies no oversampling
BOOL dohiddenrem=FALSE; // enable/disable SLOW hidden point removal

const picwidth=620;     // width of the picture in pixels
const picheight=350;    // height of picture in pixels
const NumColors=64;


// ----------------  TMyWindow  ----------------

class TMyWindow : public TWindow
{
private:
 int pixels[picwidth*oversam];
 int link[picwidth*oversam];
 int z[picwidth];
 HPALETTE hpal;

public:
 TMyWindow( PTWindowsObject AParent, LPSTR ATitle);
 ~TMyWindow();

 virtual void Paint( HDC PaintDC, PAINTSTRUCT& PaintInfo );
};


TMyWindow::TMyWindow( PTWindowsObject AParent, LPSTR ATitle) :
              TWindow(AParent, ATitle)
{
 Attr.W=picwidth+8;   // Set the default window size
 Attr.H=picheight+26;

 // Create and initialise color palette with 64 shades of blue/green
 LPLOGPALETTE pal;

 pal=(LPLOGPALETTE) farmalloc(sizeof(LOGPALETTE)
                          + sizeof(PALETTEENTRY) * NumColors );
 pal->palVersion = 0x300;
 pal->palNumEntries = NumColors;

 for(int n=0; n < NumColors; n++)
 {
  pal->palPalEntry[n].peRed   = 0;
  pal->palPalEntry[n].peGreen = n*2;
  pal->palPalEntry[n].peBlue  = n*4;
  pal->palPalEntry[n].peFlags = PC_RESERVED;
 }

 hpal = CreatePalette(pal);
 farfree(pal);
}


void TMyWindow::~TMyWindow()
{
 DeleteObject(hpal);   // delete the palette
}


void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& )
{
 int x,y;
 int h;    // height of 'features'
 int u,dx,c,xx;
 int highest;
 int separation,left,right;
 int pp;
 long xs=260,ys=150,zs=-580;
 float v;
 BOOL visible;

 long r,s; // temporary storage for constructing sphere

 HPALETTE oldPalette;

 oldPalette=SelectPalette(PaintDC,hpal,FALSE);
 UnrealizeObject(hpal);
 RealizePalette(PaintDC);


 for (y=0; y < picheight; y++)
 {
  for (x=0; x < picwidth*oversam; x++)
  {
   link[x]=x;
  }

  highest=bkdepth;

  for (x=0; x < picwidth; x++)
  {
   h=bkdepth; // by default, image is flush with the background

   // start of scene-generating code
   if ((y >= ys-64) && (y <= ys+64))
   {
    r=64*64-(y-ys)*(y-ys);
    if (r>0)
    {
     s=r-(x-xs)*(x-xs);
     if (s > 0) h=sqrt(s)+zs;
    }
   }
   s=(3*xs-5*ys+4*zs-3*x+5*y)/4;
   xx=sqrt((x-xs)*(x-xs)+(y-ys)*(y-ys)+(s-zs)*(s-zs));
   if ((xx > 80) && (xx < 120) && (s > h)) h=s;

   if ((y >= 8) && (y < 32)) h=((x/32)%2)*32+bkdepth;
   if ((y >= 256) && (y < 280)) h=(x/32)*16+bkdepth;
   if ((y >= 296) && (y < 320)) h=x/2+bkdepth;
   // end of scene-generating code

   z[x]=h;   // store the height in the array

   if (h > highest) highest=h;
  }


  for (x=0; x < picwidth*oversam; x++)
  {
   separation=(E*oversam*z[x/oversam])/(z[x/oversam]-o);

   left=x-separation/2;
   right=left+separation;

   if ((left >= 0) && (right < picwidth*oversam))
   {
    visible=TRUE;

    if (dohiddenrem)
    {
     v=2.0*(o-z[x/oversam])/E;

     dx=1;
     do
     {
      u=z[x/oversam]+dx*v;
      if ((z[(x+dx)/oversam]>=u) || (z[(x-dx)/oversam]>=u)) visible=FALSE;
      dx++;
     }
     while ((u <= highest) && (visible==TRUE));
    }

    if (visible) link[right]=left;
   }
  }

  pp=0;
  for (x=0; x < picwidth*oversam; x++)
  {
   if (link[x]==x)
   {
    // ensures basic pattern does not change much on a scale
      // of less than one pixel when oversampling is used
    if ((pp%oversam)==0) c=random(NumColors);
    pixels[x]=c;
    pp++;
   }
   else
    pixels[x]=pixels[link[x]];
  }

  for (x=0; x < picwidth; x++)
  {
   xx=x*oversam;


   switch (oversam)  // use different 'filters' depending
                     // on oversampling ratio
   {
    case 1:
     c=pixels[xx];
    break;

    case 2:
     c=(pixels[xx]*42+(pixels[xx-1]+pixels[xx+1])*24
       +(pixels[xx-2]+pixels[xx+2])*5)/100;
    break;

    case 4:
     c=(pixels[xx]*26+(pixels[xx-1]+pixels[xx+1])*18
       +(pixels[xx-2]+pixels[xx+2])*12
       +(pixels[xx-3]+pixels[xx+3])*7)/100;
    break;

    case 6:
     c=(pixels[xx]*14+(pixels[xx-1]+pixels[xx+1])*14
       +(pixels[xx-2]+pixels[xx+2])*11
       +(pixels[xx-3]+pixels[xx+3])*8
       +(pixels[xx-4]+pixels[xx+4])*5
       +(pixels[xx-5]+pixels[xx+5])*3
       +(pixels[xx-6]+pixels[xx+6])*2)/100;
    break;
   }

   SetPixel(PaintDC,x,y,PALETTEINDEX(c));
  }
```

```
 }
 SelectPalette(PaintDC,oldPalette,FALSE);
}



// ----------------  TMyApp  ----------------

class TMyApp : public TApplication
{
public:
 TMyApp(LPSTR AName, HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
   : TApplication(AName, hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};

 virtual void InitMainWindow();
};


void TMyApp::InitMainWindow()
{
 MainWindow = new TMyWindow(NULL, Name);
}


int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
 TMyApp MyApp("Original SIRDS by W.A.Steer", hInstance, hPrevInstance,
               lpCmdLine, nCmdShow);
 MyApp.Run();
 return MyApp.Status;
}
```

HOW IT WORKS

Principles of 3D Imagery

```
---------------------------
    xxx           xxxx
      x    xxxxxxxx    object
      xxxx*x
          |
         | |                            BASIC PRINCIPLE for
      L  | |  R                         3D Imagery
........*...*.............. image plane
        |   |
       |     |
       |     |
      |       |
      |       |
     |         |
     o         o
   L             R
```

All single-image 3D systems (eg red-green glasses 3D) work on the principle that the left and right eyes see different features on the image plane which the brain interprets as a 3D object (see diagram above). The glasses ensure that each eye sees only one of the two images (with red/green specs, the eye with the RED filter only sees the GREEN image). Other technologies for the same effect include polarized images/glasses (used for a few films), and flashing left/right dark/clear LCD specs with corresponding alternate left and right images on a computer screen.

BUT with autostereograms any point for the right eye is ALSO seen by the left eye as shown below. An interpretation must then be made for that, so an extra point X is introduced, as a corresponding point for the right eye. This dependence must continue and be repeated across the entire display.

```
---------------------------
    xxx           xxxx
      x    xxxxxx*x    object
      xxxx*x    /|
          |    / |
         | |  /  |                      IDEAL / REAL LIFE
      L  | | /R  | X                    geometry
........*...*....*......... image plane
        |  /|   |
       |  /  |  |
       | /   |  |
      | /     | |
      |/      | |
     |/        |
     o         o

   L             R
```

For the purpose of generating SIRDS it is usual to assume the geometry below, where the eyes 'move' along the image.

```
---------------------------
    xxx           xxxx
      x    xxx*xxxx    object
      xxxx*x  |
          |   ||
         | | |  |
         | | |  |                       SIMPLIFIED geometry
........*...*....*......... image plane
        |   |    |
       |   | |    |
       |   | |    |
      |   |   |    |
      |   |   |    |
     |   |     |    |
     o   O     o    O

    L   L       R    R
```

This Program

Simplifications:

- We assume that the viewer looks STRAIGHT AT all parts of the image (looks along the perpendicular to the screen at all points on the object) as shown above.

  reasons: much simpler math's, no preference for a particular viewing point.

  adverse effect: parallax error: features towards the sides get pulled inwards slightly.

- In this program, only one value of depth is allowed for given values of x & y.

  reasons: smaller/simpler storage requirements for the object, generally simpler and faster to code.

  adverse effects: imperfect representation of objects behind narrow objects.

```
            plan views:
```

```
background  xxxxxxxxxxxxxx    xxxxx      xxxxx   xxxxx!!!!!!xxxxx
                                                      !!!!!!
                                                      !!!!!!
                                                      !!!!!!
                 xxxx                                 !!!!!!
 pencil         xxxxxx             x    x             x!!!!x
                 xxxx               xxxx               xxxx

              real scene         as stored           program's
                                                  interpretation
```

Clearly the data offers no information about what goes on behind any point defined on the scene. The only sensible assumption to make is that the object extends from the given point back to infinity (or the background).

A scene where the viewer looks through the bars of a prison cell for example, might warrant a fuller depth description.

- The program is not capable of producing a perspective image, given the above limitations, although there is no reason why more distant parts of the image could be defined smaller.

I have adopted the following coordinate system as it seems logical and avoids the use of floating point math which is slow. (On a 486sx without math co-processor about 100 integer multiplications can be performed in the time taken to do ONE similar floating point operation (about 30 and 3000 clock cycles respectively). When considering speed, it should be borne in mind that merely plotting several hundred thousand pixels on the screen takes an appreciable amount of time!

```
------------------------------   background
                             ^
                             |
                  object     |
             xxxxxxxx        |
         ^      /\           | background depth  bkdepth
         |     |  |          |
       d |     |  |          |
         v     |  |          v
 .............*....*...........   image plane
separation -->|    |<--      ^
              |    |         |
              |    |         |
             |      |        | observer distance  o
             |      |        |
             |      |        |
       eyes  o      o        v

             <------>
                E
```

Similar triangles:

separation/d = E/(d+o)

separation = d*E/(d+o)

Now let us introduce an (x,y,z) coordinate system:

x - distance across the screen, measured from the left
y - distance down the screen, measured from the top (unconventional but Windows and older IBM graphics systems go that way)
z - distance from the screen; negative behind the screen, positive in front.

(mathematicians would call this an anticlockwise (unconventional) coordinate system. If necessary we could swap the y-direction by making the program plot the right way up (+ve upwards) but unless the data warrants it it just adds an unnecessary complication)

separation = z[x][y]*E/(o-z[x][y])

Almost invariably we need a continuous background for the scene; it is usually chosen to be the same distance behind the screen as the observer is in front, enabling the observer to look at his reflection in order to aid the correct convergence of his eyes.

In general, it is best not to allow a range of depths which causes the separation to vary by a factor of two or more since the image can be optically misinterpreted - and difficult to see properly. With *caution*, (basically not allowing a direct boundary between very near and far objects, and including several slopes to guide the eyes) you can get away with deeper pictures.

For scientific images or fractals, it may be convenient to set the z[] values as bkdepth+h where h is the height of the data.

It should be noted that as the observer moves further away the depth effect becomes stronger and vice-versa. The 'correct' depth will only be seen when he is at the distance the image was designed for, o - if the image is reproduced at its original size.

Hidden point removal

It is technically incorrect to plot a stereo pair of dots corresponding to a point on the object which is visible to only one eye - to do so would cause an ambiguity near a change in depth.

```
--------------------------------   background

                object
       xxxxxxxxxxxx ______
        ^      / \x      |
        |     /   xxxxx  | Dz
      d |    /  Dx \     v
        |   /   |-->\ -----
        v  /         \   ^ u
..........*...........*..|......   image plane
         /             \
        /               \
       /                 \
      /                   \
     o        eyes         o
```

If any part of the ray to either eye goes behind a point defined as being on the surface of the object then the ray is deemed to be intercepted, since we defined the object to be continuous in the z-dimension.

The depth, u(x), of any point of the ray can be found by similar triangles.

$2*Dx/Dz = E/(d+o)$

$Dz = (2*(d+o)/E)*Dx$

$u(x+Dx) = d-(2*(d+o)/E)*Dx$

Amending for the coordinate system where depths into the screen are negative (and hence u() is also -ve)

$u(x+Dx) = z[x]+(2*(o-z[x])/E)*Dx$

Then if

$z[x+Dx] >= u(x+Dx)$

is true for any value Dx up to where u() meets the image plane the ray is intercepted - and the point is not visible to both eyes.

For speed, we only need to do the test up to u(x+Dx) = height of most prominent point on the current scan line.

Algorithm

This version of my SIRDS program uses a symmetric algorithm based on information given in:

> "Displaying 3D Images: Algorithms for Single Image Random Dot Stereograms", a paper by H.W. Thimbleby, S. Inglis and I.H. Witten (available from ftp://ftp.cs.waikato.ac.nz/pub/SIRDS)

although I have adopted a different coordinate system.

In summary:

```
for each line (y-coordinate)
{
 for each x
 {
  link[x]=x        // link each point with itself
 }

 for each x-coordinate of the object
 {
  find the stereo separation corresponding to the depth of the
    object at this value of x & y, as given in the math's previously

  left=x-separation/2
  right=left+separation   // to reduce effects of rounding errors

  if the point is visible to both eyes
   link[right]=left   // link these two points
 }

 for each x-coordinate
 {
  if (link[x]=x)
   generate a random colored dot
  else
   print a dot in the color of the dot at link[x]
 }
}
```

N.B. There is no geometric reason to cause a dot already linked to be linked again, although rounding errors could create two links to two adjacent points - in this case the latter link wins!

One last problem:

On an ordinary computer monitor (around 70dpi), curved or sloped surfaces in stereograms as described appear broken into distinct planes parallel to the image plane.

Examination of the geometry reveals that for usual depths, the z-resolution is around 7 times worse than the x-resolution of the display device.

(Sheer high-definition alone won't solve the problem either: if you were to draw for a 600dpi laser, the dots may turn out too small to see easily)

Need to introduce Z-RESOLUTION ENHANCEMENT

If the stereogram is calculated at higher x-resolution - say 4 times the display resolution (I call it oversampling), and then properly reduced for display we can lose those distracting 'staircases'.

Basically each screen point is assigned a color by means of a weighted average of several of the calculated points.

eg for 2* oversampling:

```
calc pts        x    x    x    x    x    x    x

weighting           .05  .24  0.42 .24  .05

 mix together          \    \  |  /    /

screen point                  X
```

The weightings must add up to one, and a bell-shaped distribution works quite well.

The figures given were derived from a Normal (Gaussian) distribution:

```
            1           -(dx^2)/(2*S^2)
   w = --------------  e
       S * sqrt(2*PI)
```

dx is the distance from the centre of the distribution
S is the standard deviation (try S=oversam/2)
w is the (fractional) weighting factor

The distribution extends to +/- infinity but the weighting factors tend to zero, so we only use the first few.

In practice, it is noticeably faster to make the weightings integer on a scale from 0 to 100, then divide the sum by 100 (remember the speed advantage of integer math).

To accurately reproduce the averaged color a display with more than 16 colors is needed. For a simple, with a linear color series (eg black through to blue, or red to green) in a palette it is easy to find the in-between color reference. With more complicated programming and/or a 16.7million color display, in-betweens for ANY color combinations could be found.

(Actually you could use fewer colors, even ordinary black and white, by using probabilities to paint 'in-between' colors - providing there is linear resolution to spare.)

It is important that the bulk of the calculated stereogram pattern does not contain detail smaller than one pixel as this would get lost as the resolution is reduced for display. Hence for 4* oversampling the colors in the basic pattern should not change more often than every 4th point.

Conclusion

Stereograms are a rapidly expanding business and there are very good posters by NVision and others. Unfortunately there is also an increasing amount of rubbish (especially on the Internet).

The program offered is a basis for creating stereograms of a high technical quality, but a good deal of artistic ability is needed to produce aesthetically pleasing masterpieces.

send all enquiries to:

Andrew Steer (w.steer@ucl.ac.uk))

Subject: [26] Use POV-RAY to build depth images?

From: jolsen@nyx10.cs.du.edu (John Olsen)
Newsgroups: alt.3d
Subject: Re: Using POV-RAY to generate data for SIRDS? (Yes! Source included.)
Date: 29 Jun 1994 21:40:13 -060
joel@wam.umd.edu (Joel M. Hoffman) writes:
[Use POV-RAY to build depth images?]

This comes up once eery month or so. Here's how to do it. (I just happen to be reading news on the system containing the modified source for a change. Stuart or Todd: Can this go in the FAQ?)

You need to change render.c, and should not need to hit any other files. Insie the Trace() function, you need to replace where it looks up colors with the already available depth information. The full diff ("diff render.c.new render.c" assuming POV2.0) contains a bit of other tweaking:

```
----------------------------------------------------
382c382
<   /* Make_Colour (Colour, 0.0, 0.0, 0.0); */
---
>   Make_Colour (Colour, 0.0, 0.0, 0.0);
408,414c408
<     {
<       Make_Colour ( Colour,
<                  1-((int)(Best_Intersection.Depth) % 255 ) / 255.0,
<                  1-((in)(Best_Intersection.Depth) % 255 ) / 255.0,
<                  1-((int)(Best_Intersection.Depth) % 255 ) / 255.0);
<       /* Determine_Apparent_Colour (&Best_Intersection, Colour, Ray); */
<     }
---
>     Determine_Apparent_Colour (&Best_Intersection, olour, Ray);
416,422c410,413
<     {
<       /* if (Frame.Fog_Distance > 0.0)
<        *Colour = Frame.Fog_Colour;
<         else
<         *Colour = Frame.Background_Colour; */
<        Make_Col, 0.0, 0.0, 0.0 );
<      }
---
>      if (Frame.Fog_Distance > 00)
>        *Colour = Frame.Fog_Colour;
>      else
>        *Colour = Frame.Background_Colour;
-----------------------------------------------------
```

Subject: [27] Perspective correct generation

Peter Change has written a short article on autostereograms with a perspective corrected for in their generation.

The article is available here: http://www.ccc.nottingham.ac.uk/~etzpc/ss/ss.html

Subject: [28] (Almost) Correct Color SIRDS

(from Esa Kuru's home page - 5th Jan 1995)

Let it be stated here that **perfect Correct Color SIS** are impossible to render. In these images the only random thing is the offset of horizontal position of random-dot-pattern-buffer. So these are really SIS images - not SIRDS.

- A new improved SIS in color now under construction... available soon at this page only. ... Source code fully rewritten from scratch.
- doublepyramid.jpg is one pyramid towards You and another away from You on top of each other - planes with different colors.
- colorplane8.jpg shows how saturation can be changed without too much suffering of lack of 3d effect.
- pyramid.gif of green dots for beginners. This is cross-eyed version though.
- world.jpg is the most sophisticated version using "dolby stereo Pro Logic"-like dynamics. This is the story **how** it was done. This was rendered before Infix Technologies got theirs out and personally I prefer mine because of it's colors.
- earthsird.jpg is the first attempt to render a correct color on multicolored surface. The idea is to compute average of two colors seen by both eyes and lay it on random-dot-buffer presenting the depth of that part of the image. Earth.sird.readme at nic.funet.fi. This is slightly more colorful than the world.jpg above.
- 3D frames of an animation presenting a tour round a world in true colors. I have put a Macintosh QuickTime movie out of these frames and it seems to work ok. Unfortunately it is removed from publicly accessible disks due to it's huge size. At the time it was create d, no-one really knew what it really was for. Nowadays these SIRDS-things are g etting into day light among netsurfers.
- WaveSIRDham.lha is a animation with antialiasing for amiga. It is viewable also on X-windows.