Image and Video Processing

Image processing is the name given to any procedure that takes as its input a picture, manipulates it in some way and creates another picture from it. It can range from the simple action of turning an image upside down to the application of sophisticated noise reduction algorithms such as those that enhance photographs from satellites, security cameras and space probes.

Video processing can be thought of as an extension of image processing because a video signal is simply a collection of images. For most processes each video frame is processed one at a time in exactly the same way that a single image is. However a few processes such as motion blur are simulated by considering a sequence of consecutive pictures. Another common video processing action is to mix two or more video streams together, a simple wipe from one to the other is an example, as is a page turn effect.

In the context of 3D graphics, the output from a renderer is just an image, and therefore it can be processed as such. However, a renderer has the potential to provide additonal information. Indeed, quite a few 3D effects are more efficiently implemented as an image processing stage executed after the rendering phase is complete. Particle models for explosions and fireworks are potential candidates for implementation in this way. An image processor working on the output of a 3D renderer has a decided advantage over one working on an image alone because it can have access to the Z buffer and other information like the position of the observer, models in the scene and the coordinate frame of reference used to describe the scene.

With the exception of those effects that use information recorded in a Z buffer all of the topics discussed in this chapter could be applied to any image or set of images not just those generated by rendering a 3D scene.

A number of the algorithms given here are fairly simple to understand and therefore will be mainly illustrated with computer code using **C** and **C++** compatible constructs. The code for the processes discussed here is included with the book.

0.1 A data structure for image processing

An image is made up from pixels, each pixel is a sample of the color in a *small* rectangular area of the image. The color of a pixel is recorded as three values, one for each of the primary colors red, green and blue (RGB). A combination of primary colors is sufficient to give the illusion that a full rainbow color spectrum is reproducible on a computer monitor. To record an excellent approximation to a **full** spectrum the standard method is to allocate one byte for each of the red, green and blue (RGB) components. Computers naturally work in powers of two and it has proved very useful for image processing and 3D applications to allocate a fourth byte to each pixel for the support of an alpha channel (A).

An alpha channel is used to hold a compositing mask that covers part of an image. A 3D computer animation program typically uses an alpha channel to facilitate the overlay of computer generated images onto video signals or background images from other sources. A specific example is the use of a live video signal as a background for a computer generated model. If the model

Figure 1: Using an alpha channel to compose a computer generated image with a photograph.

contains some "glass" the alpha channel allows the background to show through the glass with an appropriate level of attenuation. Figure 1.1 illustrates the use of an alpha channel to composit a computer generated image onto a photograph.

As stated above, pixels in an image are conveniently represented by a four byte structure with members *R,G,B* and *A*. An image, with width w and height h, is represented by a two-dimensional array of such structures, we will represent this as $I(x, y)$. The array is usually recorded in an application as a one dimensional block of memory ordered row by row, within a row pixels are stored left to right. Such a storage scheme makes it easy for a program to; process the whole image by indexing a pointer, and to dynamically allocate memory so that images of differing resolution can be processed. A one dimensional array fits naturally with the idea of an output raster and the scanline rendering algorithm.

In the **C** or **C++** languages a structure representing a pixel in the frame buffer may be defined as:

```
typedef unsigned char BYTE;

typedef tagPIXEL{
BYTE Red,
     Green,
     Blue,
     Alpha;
} PIXEL, *lpPIXEL;
```

Note that because the color and alpha entries are represented by unsigned 8 bit quantities they can only take values in the range $[0 - 255]$.

A framebuffer suitable to record an image n pixels wide and m lines high is established by:

```
 .
 lpPIXEL Image=(lpPIXEL)malloc(n*m*sizeof(PIXEL));
 .
```

If the whole image is to be processed, each pixel can be accessed by pointer indexation. For example to attenuate the brightness of an image by 50% use the following code fragment:

```
 .
 lpPIXEL p=Image;
```

```
.
for(i=0;i<n*m;i++,p++)
{
  p->Red   /=2;
  p->Green /=2;
  p->Blue  /=2;
}
.
```

We might note that the above code fragment could equally well be written as:

```
.
for(i=0;i<n*m;i++)
{
  Image[i].Red   /=2;
  Image[i].Green /=2;
  Image[i].Blue  /=2;
}
.
```

where array addressing (with $[i]$) replaces the pointer indexation. Or indeed it could even be written as:

```
.
for(i=0;i<n*m;i++)
{
  (Image+i)->Red   /=2;
  (Image+i)->Green /=2;
  (Image+i)->Blue  /=2;
}
.
```

Since these three methods of accessing the pixels in an image are equivalent the description of the algorithms in this chapter will use whichever one of them allows the functions to be written in the most comprehensible form. The first method is usually translated by a compiler into the fastest possible code. The second method has the most in common with the way a mathematical expression of an algorithm is written.

Whenever a Z buffer (see Chapter **??**) is available it records a floating point number for each pixel. Entries in the Z buffer are accessed in the same way as entries in the frame buffer. For example suppose we want to copy the contents of one frame buffer to another whenever the Z depth is greater than the specified value d. We might use the following function:

```
void CopyFrameBufferOnDepth(
       float d,                  // depth threshold
       int height,               // image height
       int width,                // image width
       lpPIXEL in_buffer,        // input buffer
       lpPIXEL out_buffer,       // output buffer
       float *Z_buffer)          // Z depth buffer
{
 long i;
 for(i=0;i<height*width;i++)
 {
   if(*Z_buffer++ > d)
   {
     out_buffer->Red   = in_buffer->Red;
     out_buffer->Green = in_buffer->Green;
     out_buffer->Blue  = in_buffer->Blue;
     out_buffer->Alpha = in_buffer->Alpha;
     //
     // an alternative is to use the memcpy function
     // memcpy(out_buffer,in_buffer,sizeof(PIXEL));
   }
   in_buffer++;
```

```
  out_buffer++;
}
```

Many image processing algorithms require access to a particular pixel at location (i, j) in the two-dimensional array (w, h) of w columns and h rows which hold the image data. To address pixel (i, j) when the framebuffer is held in a dynamically allocated linear array a calculation using either of the following methods is required:

```
.
//
// For example to extract the value of the Red component
// of pixel (i,j).
//
// Either use an array address
//
Red_pixel_i_j = Image_buffer[i+j*w].Red;
//
// Or calculate the pointer address
//
Red_pixel_i_j = (Image_buffer + i + (j*w))->Red;
.
```

0.2 Basic functions for drawing in a framebuffer

Some image processing functions draw simple geometric constructs into a framebuffer. The two most commonly used are those which render lines and points. To avoid the problems of aliasing that arise in all pixelated displays these functions should incorporate anti-aliasing techniques.

Two functions whose specifications are prototyped below are included with the code for the book. The functions require a parameter that points to the framebuffer in which the drawing is to occur and parameters to specify the color of the line or point as an RGB triple. The R,G and B values are specified as a floating point number in the range $[0, 1]$.

Basic frame buffer functions

1. `DrawAntiAliasedPoint( )`
   The algorithm for this function was described in section **??**.

   The parameters are as follows:

   ```
   void DrawAntiAliasedPoint(
    float x,       // horizontal position of point to set
    float y,       // vertical position of point.  Row 0 is at top.
    float w,       // width and height of point.  w=1.0 => 1 screen
                   // pixel if w < 1.0 the brightness is attenuated to
                   // simulate smaller points
    int Brightness,// brightness [0 - 255] of the pixel to draw
    float Red,     // Red, Green, and Blue define the color of the pixel
    float Green,   // by giving the proportion of that primary color.
    float Blue,    // Each takes values in the range [0-1]
    long width,    // with of image stored in "buffer"
    long height,   // height of image stored in "buffer"

    lpPIXEL buffer // Pointer to the framebuffer to be drawn in.
   )
   ```

2. `DrawAntiAliasedLine( )`
   This function draws a line of width w, it uses the Bresenham algorithm described in section **??** to draw the line by setting pixels. The pixels are set by calling *DrawAntiAliased-Point()*. It is *not* optimized for speed because it is assumed that it is *not* being used in real time systems, for most *off line* image processing it will be fast enough.

The parameters are as follows:

```
void DrawAntiAliasedLine(
 long i1,       // horizontal position (column) of start of line
 long j1,       // vertical position (row) of start of line
 long i2,       // horizontal position (column) of end of line
 long j2,       // vertical position (row) of end of line
 float w,       // width and height of pixel.  w=1.0 => 1 screen
                // pixel if w < 1.0 the brightness is attenuated to
                // simulate smaller Pixels
 int Brightness,// Brightness of the line
 float Red,     // Red, Green, and Blue define the color of the line
 float Green,   // by giving the proportion of that primary color.
 float Blue,    // Each takes values in the range [0-1]
 long width,    // with of image stored in "buffer"
 long height,   // height of image stored in "buffer"
 lpPIXEL buffer // Pointer to the framebuffer to be drawn in.
)
```

Note; these functions draw their shapes by blending them with the current contents of the framebuffer. Thus for example it is not possible to draw a *black* line because whatever the contents of the framebuffer, mixing zero intensity with them will leave the value unchanged.

It's a simple *one line* change to modify the functions so that they *overwrite* the contents of the framebuffer rather than blending them with the incoming line or point. For most of the processes discussed in this chapter blending is essential to achieve the desired result.

0.3 Filtering images

A very broad range of processes can be classified as one of applying a *filter* to an image. In these processes, one image is taken as the input, it is passed through a function that considers each of its pixels and generates a second image as output. The term filtering follows from the fact that to the computer an image is just a *digital signal* and therefore like any other digital signal it can be passed through a filter to alter its characteristics. Filtering digital signals in one aspect of signal processing, and digital signal processing in particular is a major focus for R&D in the electronics industry today. Like any other filter a digital image filter modifies an image so that its spectral content is changed in some way. For example, in the context of image processing a *low pass* filter which removes high frequency components is equivalent to making the image appear*blurred*.

There is a wealth of theory on digital filtering and all of it is applicable to the filtering of signals representing images. However the practical application of a filter to an image is accomplished by a very simple algorithm which considers each pixels individually and calculates a new value based on a weighted combination of one or more pixels in the image according to the expression:

$$p_i = \sum_{\text{all pixels } j} w_j p_j$$

For most image filters it is usual for the weights w_j to be non-zero only in pixels adjacent to pixels i. This process called *convolution* is illustrated in Figure 1.2. The family of filters which follow the approach illustrated in Figure 1.2 have nine non zero weights, at pixel i and its eight nearest neighbors. The array of weights used during a convolution process is commonly referred to as the *kernel*. The elements of the 3×3 array of weights governs the action of the filter. It is thus possible to write code for a general procedure to implement the convolution function and achieve the action of several different filters. Such a function which uses the data structures described in section 1.1 is given below. In it there are a number of points to note:

1. The framebuffer uses a $0 - 255$ integer range with which to record the color intensities. Some of the filters have negative weights and thus negative numbers can appear in the

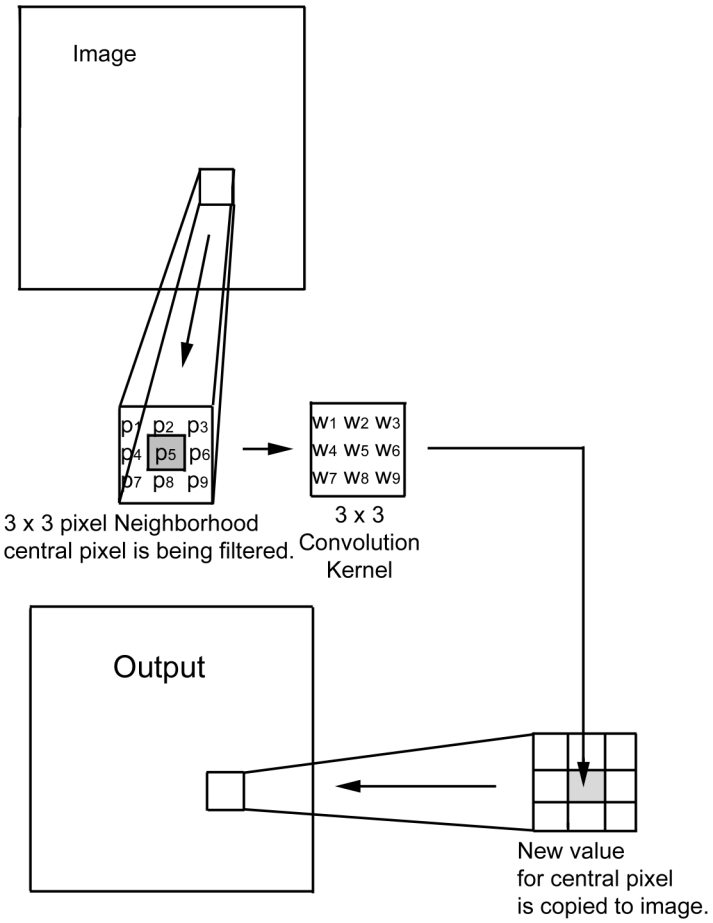

Figure 2: Convolution to implement an image filter. The value of pixel 5 becomes $p'_5 = p_1w_1 + p_2w_2 + p_3w_3 + p_4w_4 + p_5w_5 + p_6w_6 + p_7w_7 + p_8w_8 + p_9w_9$.

calculations. The `bias` parameter allows the *zero* level to be set anywhere in the range $0 - 255$.

2. The pixel value determined from some of the filters will exceed the $0 - 255$ integer range. In these cases the result can be clamped to 255 or scaled down, the parameter `scale` attenuates the weighted pixel average.

3. Linear arrays are used to record the two-dimensional image and the convolution weights. Thus, when forming the convolution sum an appropriate address calculation must be performed.

4. Care must be taken near the edges of the image because at those points there are no neighboring pixels on some sides. For example a pixel on the left edge of the image has no neighbor to its left.

5. Although the majority of convolution filters have a 3×3 array of weights the function can be used with an array of arbitrary dimension, say $m \times n$ but, n and m must both be odd.

The convolution function

```
void Convolve(
 long Xres,      // Width of image
 long Yres,      // Height of image
 lpPIXEL Si,     // Pointer to location where image is recorded
 lpPIXEL So,     // Pointer to location for result
 float scale,    // Scale value to apply to result
 long bias,      // Bias added to all pixel values
 long filterX,   // Width of the filter matrix
 long filterY,   // Height of filter matrix
 float *filter)  // Pointer to filter matrix
{
 lpPIXEL S;
 long i,j,k,l,c,r;
 float ar,ag,ab,weight;
 //
 // To prevent addressing errors at near the edges of the image
 // only work with pixels that are located at a distance of
 // >= half the width or height of the filter matrix from the
 // edge of the image.
 //
 r=filterY/2;
 c=filterX/2;
 for(i=r;i<Yres-r;i++)   // Do all valid rows in the image
 {
   for(j=c;j<Xres-c;j++) // Do all valid columns in the image
   {
     //
     // reset the accumulated weighted average for each color
     // of pixel (j,i)
     //
     ar=ag=ab=0.0;
     //
     // Apply the filter to pixel (j,i)
     //
     for(l=0;l<filterY;l++)for(k=0;k<filterX;k++)
     {
       //
       // Calculate the address in the linear input array
       //
       S=(Si+(Xres*(i+(l-r))+(j+(k-c))));
       //
       // Determine the weight for filter matrix element (k,l)
       //
       weight = *(filter+filterX*l+k);
       //
       // Accumulate the weighted sum
```

```
      //
      ar+=(double)S->Red*weight;
      ag+=(double)S->Green*weight;
      ab+=(double)S->Blue*weight;
    }
    //
    // Determine the address in the output array
    //
    S=(So+(Xres*i)+j);
    //
    // Write output values applying the "bias" and "scale". Note
    // use of "max" and "min" functions to limit the output range
    // to [0 - 255]. If not provided by the compiler "max" and "min"
    // functions can be written as the following macros:
    //#ifndef min
    //#define min(a,b)  ( ((a) < (b)) ? (a) : (b) )
    //#endif
    //#ifndef max
    //#define max(a,b)  ( ((a) > (b)) ? (a) : (b) )
    //#endif
    //
    S->Red   = (unsigned char)max(0.0,(min(255.0,bias+ar*scale)));
    S->Green = (unsigned char)max(0.0,(min(255.0,bias+ag*scale)));
    S->Blue  = (unsigned char)max(0.0,(min(255.0,bias+ab*scale)));
  }
 }
}
```

Details of the analysis that provides values to use for the weights for each type of filter may be found in any one of a number of theoretical texts, for example [2] or [3]. The following subsections discuss some specific examples.

0.3.1 Low-pass filters

By removing high frequency components from an image a Low-Pass filter increases the *blurriness* and can be useful when it is necessary to reduce the appearance of 'specs' of noise.

A 3×3 array of weights is sufficient to describe a Low-Pass filter. The array is passes to function `Convolve` which then carries out the filtering process. There are three common settings for the value of the weights:

$$\begin{array}{ccc} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{array} \qquad \begin{array}{ccc} \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{5} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \end{array} \qquad \begin{array}{ccc} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{5} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{array}$$

The different values of the weights causes slightly different contributions from neighboring pixels to be added to the central one. Note that when all the weights are added the sum is unity. This is important because it implies that there is **no** overall boosting of the brightness of the image and therefore no scaling is required.

Figure 1.3 shows the result of applying the first filter to an image of resolution 540×420.

0.3.2 High-pass filters

A High-Pass filter is the opposite of a Low-Pass filter, it accentuates parts of an image that change rapidly and thus can give the illusion of clarifying or *sharpening* the picture. High-Pass/Low-Pass filters are used in video recorders which have a "Soft/Sharp" control.

A 3×3 array of weights is sufficient to describe a High-Pass filter. The array is passed to function `Convolve` which then carries out the filtering process. The following sets of weight values are in

Figure 3: An example of applying a Low-Pass filter to the image on the left gives the result shown on the right.

common use:

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{matrix} \qquad \begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix} \qquad \begin{matrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{matrix}$$

Note that some of the weights are negative and therefore the resulting image should be biased to the center of the $0 - 255$ range.

Figure 1.4 demonstrates the result of applying the first filter to an image of resolution 540×420.

0.3.3 Blurring

Again the convolution function can be used to blur an image. This time its effect is simply to replace the value of every pixel with the average of a square section of the image centered on that pixel. The bigger the area chosen the more blurred the image becomes. A typical array of convolution weights is the 5×5 matrix:

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

Note that the sum of all components is 25 and therefore a scale factor of $\frac{1}{25}$ must be used. Of course this could be accomplished by using an array of weights in which each entry is $\frac{1}{25}$. Figure 1.5 shows the effect of applying successively larger *blurring* areas to the original image. This simple blurring is really nothing more than a low pass filter. However, if the weights are changed to emphasis regions of the convolution array other filter shapes such as a Gaussian can be obtained and often these give better results.

Figure 4: An example of applying a High-Pass filter to the image on the left gives the result shown on the right.

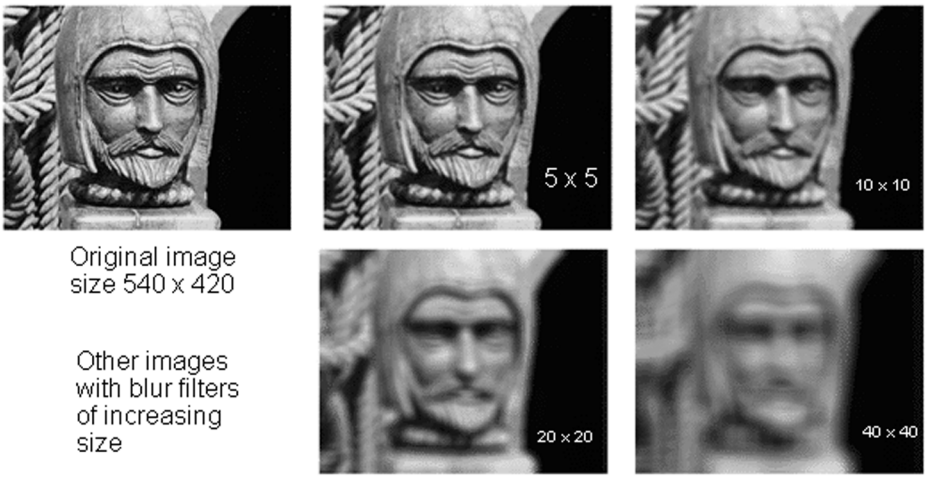

Figure 5: Blurring an image by averaging blocks of pixels centered on pixel (i, j). The "blurring" area is increased from 0 on the left to 40×40 at the bottom the right for an image of resolution 540×420.

0.3.4 Sharpening

It was stated above in section 1.3.2 that a High-pass filter accentuated the fine detail in an image. To the eye this has the illusion of making the image look clearing and sharper.

A High-pass filter on its own is usually unsatisfactory but if the output from such a filter is mixed with the original image good results are obtained. For example adding $\frac{1}{5}$ of the output from a High-Pass filter to $\frac{4}{5}$ of the original will give an image with a quite noticeable increase in *sharpness*. The following code fragment implements this:

```
.
.
{
 //
 // High-Pass Spatial filter convolution kernel
 //
 float filter[]={-1.0,-1.0,-1.0,  -1.0,9.0,-1.0,  -1.0,-1.0,-1.0};
 //
 // Implement the filter. Note the scaling factor of 1.0 and the
 // bias of 128
 //
 Convolve(Xres,Yres,Si,So,1.00,128,3,3,filter);
 //
 // mix in the high frequency component. Note how the bias
 // is removed before adding in the high frequency component.
 //
 for(i=0;i<Yres;i++)for(j=0;j<Xres;j++)
 {
   S=(So+(Xres*i)+j);
   s=(Si+(Xres*i)+j);
   S->Red   = (unsigned char)max(0.0,min(255.0,0.80*(double)s->Red +
   0.40*((double)S->Red - 128)));
   S->Green = (unsigned char)max(0.0,min(255.0,0.80*(double)s->Green +
   0.40*((double)S->Green - 128)));
   S->Blue  = (unsigned char)max(0.0,min(255.0,0.80*(double)s->Blue +
   0.40*((double)S->Blue - 128)));
 }
}
.
.
```

The result of boosting the high frequency components is illustrated in Figure 1.6.

0.3.5 Edge detection

Edge detection is a significant topic in itself because it has implications for pattern recognition, robotic vision and collision avoidance. From our point of view the most probable use of edge detection is in sharpening an image by mixing the output of a High-Pass filter with the original image in those areas where an edge has been detected.

There are several convolution kernels that can be used to detect specific types of edge, horizontal, vertical, diagonal etc. Lindley [1] gives examples of eight types.

Perhaps the best know edge detection algorithm is Sobel's. It is non-linear and in one of its forms it can be coded quite simply and is therefore fast. Sobel's algorithm has the advantage that a threshold can be set which allows it to detect only *prominent* edges, it is also omni-directional, i.e. it does not favor horizontal or vertical lines. Figure 1.7 illustrates the detection of the most significant edges.

Figure 6: Giving the illusion of increased sharpness in an image by amplifying the high frequency components. The image on the right is a mixture of 80% original (left) and 20 % high pass filter.

Figure 7: Sobel's algorithm for edge detection. With the threshold set to detect significant edges.

0.4 Effects using the Z buffer

A number of appealing visual effects that enhance the appearance of rendered images may be created in an 'postprocessing' stage if it has access to the renderer's Z buffer.

0.4.1 Depth of field

A photographer using a real camera usually tries to make sure that the scene in the viewfinder is in focus. For real cameras that have a finite aperture size it can prove difficult to ensure that all object in the scene are in focus at the same time. When objects near to the camera are in focus, object far away from the camera are out of focus, and vice-versa. With the perfect 'pin hole' model for a camera used by a 3D rendering algorithm all elements in a scene are always perfectly in focus. To simulate a real camera the finite diameter of its lens must be accounted for and this is typically done by rendering and mixing slightly different views. Is it only within a certain *depth of field*, centered at a fixed distance from the camera, that objects will appear to be in focus, elsewhere they will blurred and fuzzy. As objects move away from the center of the depth of field, their fuzziness will increase.

Having to render the same scene several times, or use multiple rays for each pixel in the case of a ray tracer adds considerably to the time it takes to generate an image. If however, the values in the Z buffer are available it is possible to set up a fairly good approximation to the *out of focus* effect by using a simple blurring filter in which the degree of blurring is proportional to the difference in Z distance between any point in the scene and the focal distance assigned to the camera.

Put another way, a blurring filter is applied to pixels in which the size of the filter's $(n \times n)$ matrix is made proportional to $|Z_i - d|$. Z_i is the depth recorded in the Z buffer for pixel i and d is the distance from the camera to a point that is to remain perfectly in focus. Typically, the filter kernel would increase from 1×1 when $Z_i = d$ to *say* 15×15 when $Z_i = 0$ or $Z_i = Z_{max}$.

In the case where the focal depth is d and Δd is the depth range within which the filter kernel reduces from a maximum n_{max} down to unity and back to n_{max} the size of the $n \times n$ filter matrix is given by:

$$n = \begin{cases} n_{max} & : Z_i \geq d + \dfrac{\Delta d}{2} \\ 1 + \dfrac{2(n_{max} - 1)}{\Delta d}(Z_i - d) & : d \leq Z_i < d + \dfrac{\Delta d}{2} \\ 1 + \dfrac{2(n_{max} - 1)}{\Delta d}(d - Z_i) & : d - \dfrac{\Delta d}{2} < Z_i < d \\ n_{max} & : Z_i \leq d - \dfrac{\Delta d}{2} \end{cases}$$

Here Z_i is the depth recorded in the Z buffer for pixel i. Figure 1.8 illustrates a before and after application of this simulation of photographic depth of field.

0.4.2 Lens flares

A lens flare is an artifact that appears in the lens of a camera when it is pointed in the direction of a source of light. Ideally it should never occur and in the perfect world of computer graphics it doesn't. However photographers sometimes use this effect for added atmosphere in their work and as a result may commercial graphics applications provide it as an option.

To simulate exactly the optical properties of lenses and how light interacts with the glass material to produce a *flare* would be an extremely complex process. It would probably add far too much to the rendering times or not be realistic enough to be of practical value. A much simpler

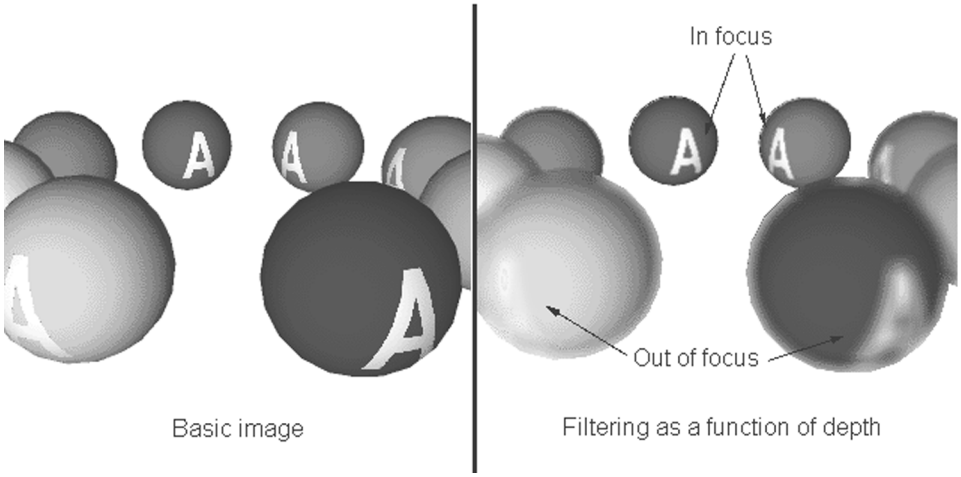

Figure 8: Simulating the photographic depth of field. The image on the right show blurring on objects close to the camera while those at the back remaining in focus.

approach, one that actually gives the 3D programmer greater control, is to "*paint*" the lens flare into the framebuffer using simple drawing functions.

There are effectively three elements in a lens flare:

1. A glow, which may be multi-colored, disks or rings or a combination of both.

2. A series of streaks leading away from the source of the flare, some random, some regular. Indeed photographers sometimes add *star* filters to the lens which enhances the effect, these produce fairly regular streaks.

3. In a multi lens camera inter-lens reflections occur that show up as rings and colored blotches in the image.

The most significant observation that makes it possible to *draw* a lens flare is that the first two effects are centered on the position in the picture where the source of the flare (usually a light but it also might be a bright reflection) is located. The third one lies along a line joining the location of the light to the center of the picture.

Figures 1.9, 1.11 and 1.10 illustrate the appearance of several types of lens flares. Given this basic idea it is possible to *draw* the flare to suit artistic taste.

In the context of 3D computer animation it is also desirable to make the flare interact with the scene in which the light sources occur. This can be accomplished by having the flare fade as the light approaches and eventually goes behind objects. The flare will disappear completely if the light is behind an object although it might leave a "glow" round the object. The flare should also fade as the light moves out of the field of view. These actions which fool the viewer into believing the light sources are *in the scene* are accomplished by knowing the Z depth of the light source and having access to the Z buffer.

An example lens flare

Consider the flare shown in Figure 1.12 which is built up from 25 elements. Here a light source is located in the lower right quarter of the picture.

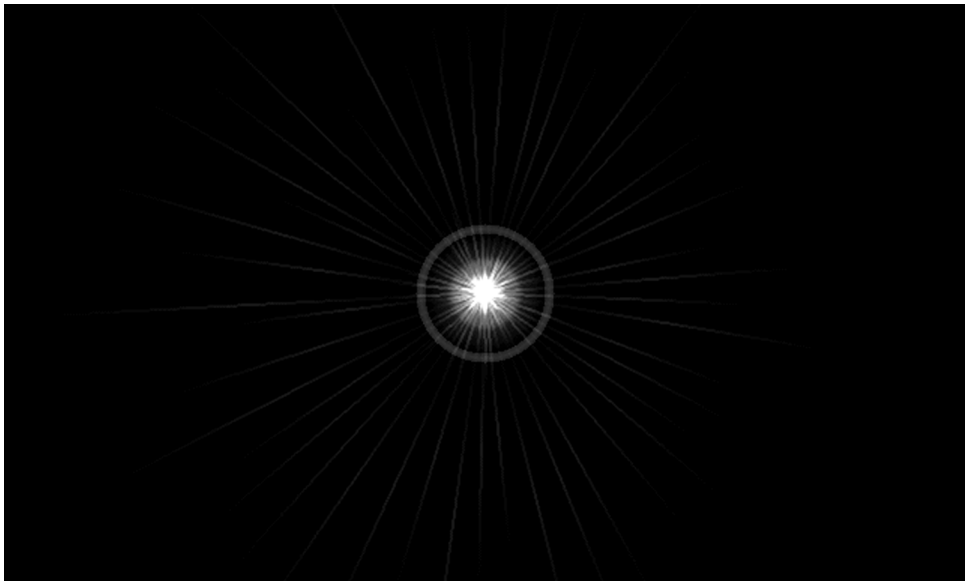

Figure 9: Basic lens flare with halo centered on light.

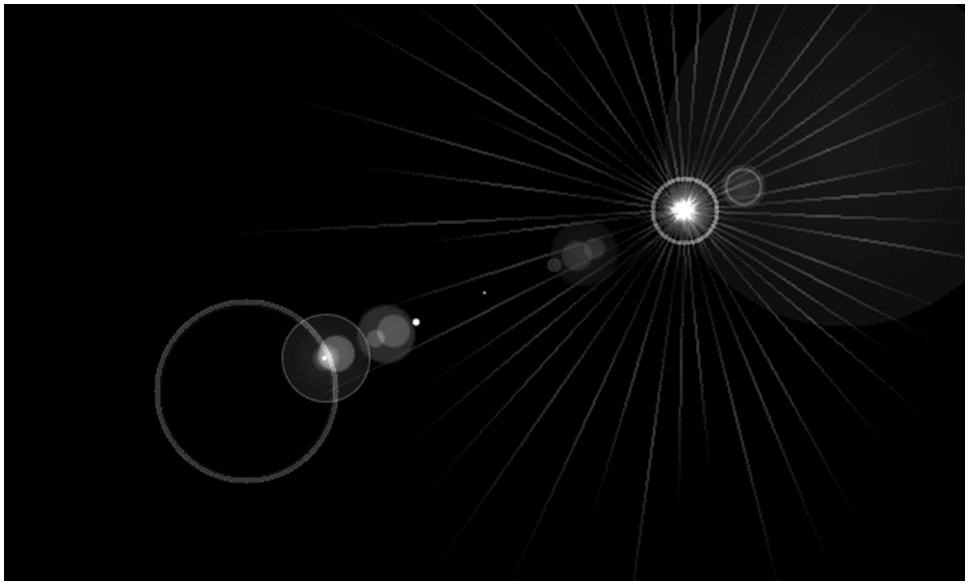

Figure 10: Lens flare with inter-lens reflections.

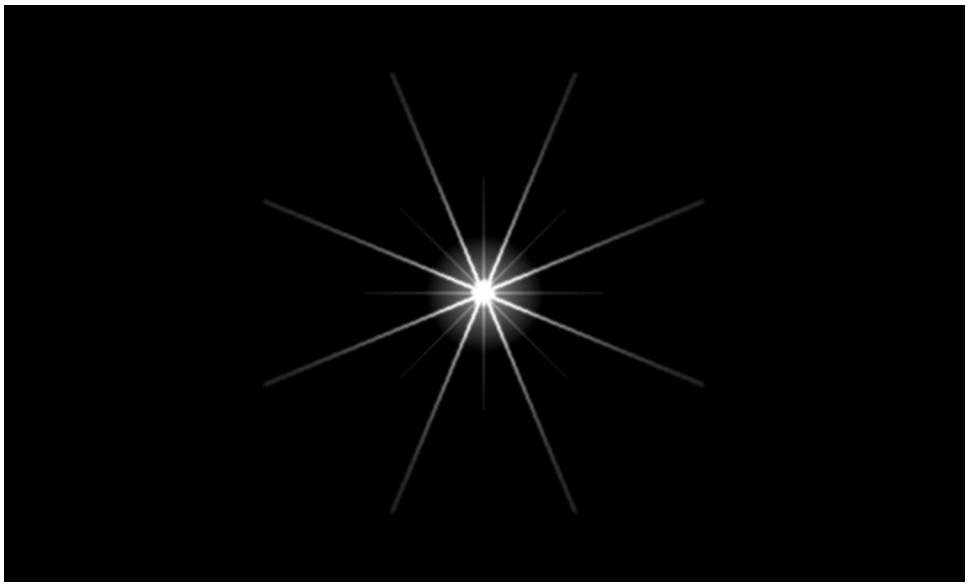

Figure 11: Lens flaring when a star filter is in use on the camera lens.

The 25 elements are used in three groups: (a) the halo round the light, (b) the fine spines radiating from the light and (c) the inter-lens reflections spaced along a line passing through the center of the screen and the point where the light is visible.

To specify the position of the items in group (c) we define l as the distance from the center of the image at $\mathbf{P}_c$ to the position of the light at $\mathbf{P}_l$, and let:

$$\mathbf{\Delta L} = \frac{\mathbf{P}_l - \mathbf{P}_c}{|\mathbf{P}_l - \mathbf{P}_c|}$$

which is a unit vector that points from the center of the image towards the position of the light. The feature size of the items in group (c) all of which have circular symmetry is specified as multiples of a basic radius r. Using this notation, the 25 components in the lens flare of Figure 1.12 are:

1. A set of 45 spines drawn as anti-aliased lines radiating from the light. Each spine increases in width towards its mid-point and then narrows and fades as the end is neared. The length of a spine is chosen randomly from a uniform distribution in a given interval. The direction (the angle of the spine) lies in the interval $[0 - 360°]$ with the addition of a small random perturbation.

2. Another three sets of 45 spines with lengths that are proportional to the length of the first set in the ratios of $\frac{1}{5}$, $\frac{1}{10}$ and $\frac{1}{20}$.

   Where the two set of spines overlap an irregular bright white region occurs.

3. A bright disk, radius $3r$, centered on the light which fades out linearly.

4. Another disk, radius $20r$, around the light but reddish in color and $\frac{1}{3}$ as bright.

5. A medium brightness red ring centered on the light with inner radius $20r$ and outer radius $23r$.

6. A bright dot at the center of the image of radius r.

7. A white disk with faded edge centered on the point $\mathbf{P}_c - 0.35l\mathbf{\Delta L}$ and radius $3r$.

8. A large brown faint disk centered at $\mathbf{P}_c + 1.8l\mathbf{\Delta L}$ and with radius $120r$. (Remember that the light is positioned at $\mathbf{P}_c + 1.0l\mathbf{\Delta L}$.)
9. A yellow disk at $\mathbf{P}_c + 1.3l\mathbf{\Delta L}$, radius r.
10. The center of the yellow disk above.
11. Faint blue disk at $\mathbf{P}_c + 0.5l\mathbf{\Delta L}$, inner radius $20r$ outer radius $25r$.
12. Second blue disk offset from the first away from the light.
13. Third blue disk offset from the first but towards the light.
14. Faint brown disk. At $\mathbf{P}_c - 0.5l\mathbf{\Delta L}$, inner radius $18r$ outer radius $21r$.
15. Second brown disk offset from the first but further away from the light.
16. Third brown disk offset from the first but nearer to the light.
17. A single faint brown disk lying between the screen center and light, at $\mathbf{P}_c + 0.35l\mathbf{\Delta L}$, radius $5r$.
18. A fading blue disk, at $\mathbf{P}_c - 0.8l\mathbf{\Delta L}$, and with radius $10r$.
19. A white spot offset from the blue disk.
20. A faded green disk offset from the fading blue disk.
21. A faint green disk at $\mathbf{P}_c - 0.8l\mathbf{\Delta L}$, radius $30r$.
22. An outline for the green disk.
23. The red section of a rainbow ring centered on $\mathbf{P}_c - 1.2l\mathbf{\Delta L}$.
24. The green section of the rainbow ring.
25. The blue section of the rainbow ring.

Code which draws a number of types of lens flares including that shown in Figure 1.12 is included with the book.

0.4.3 Motion blur

When fast moving objects are photographed they appear blurred because they have moved during the short interval of time in which the camera shutter is open. Technically this is a failing in the system. Computer animation programs don't make such mistakes but because we *expect* moving objects to look blurred in a photograph when they don't (especially movies) we feel they look *less realistic*.

To be precise, motion blur occurs as a result of a type of anti-aliasing called *temporal anti-aliasing*. In the context of computer animation it should be done by averaging the images from several pictures with each one rendered during the time interval between successive frames. (Spatial anti-aliasing, section **??**, is accomplished by the equivalent of averaging several images made from slightly different camera positions.)

Strictly speaking, to accurately simulate motion blur will requires a modification to the basic algorithm for computer animation. However if that is not possible a reasonable approximation can be achieved by appropriate manipulation of the image in the framebuffer. This works particularly well if all the objects are moving in the same direction or the camera is panning

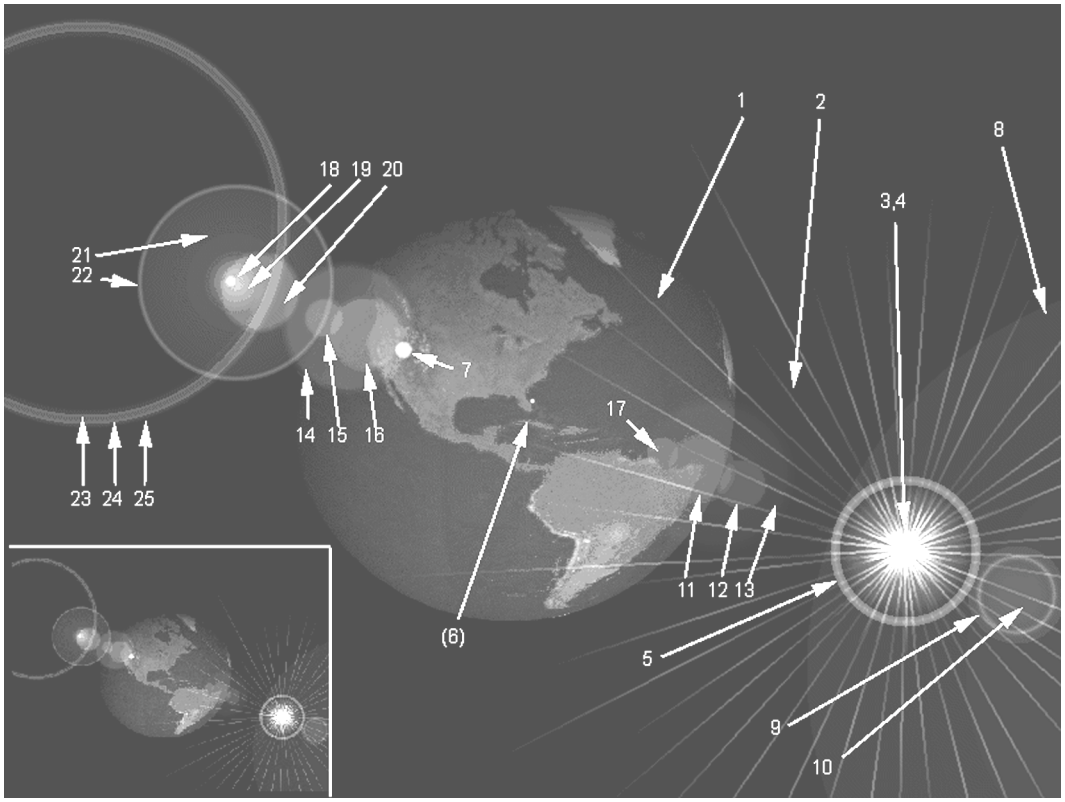

Figure 12: Building a lens flare by blending 25 elements. These are labeled and discussed in the text. (This figure is reproduced in color plate **??**.)

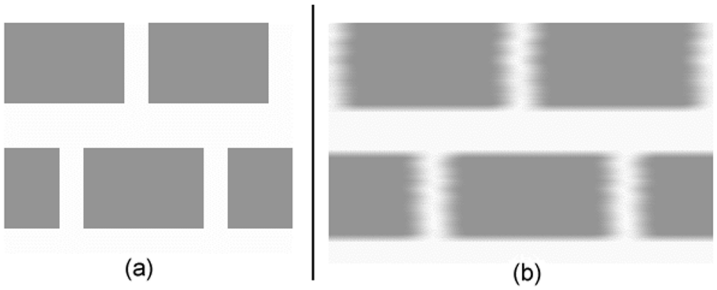

Figure 13: Directional blurring to simulate motion, (a) original image and (b) distorted image.

across a scene. In such cases the basic blurring kernel is modified to accommodate a *directional* bias. For example the 5×5 kernels become:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix} \quad \text{or} \quad \begin{matrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

Using these give the illusion of movement in horizontal and diagonal directions respectively. The illusion of vertical or horizontal motion can be further accentuated by adding a small random displacement to the rows or columns of pixels in the framebuffer. Figure 1.13 illustrates the illusion of rapid horizontal motion produced with this technique.

0.4.4 Atmospheric effects (fog)

Fog is simulated by using the value from the Z buffer to determine the proportion of a *fog* color to be blended with the image recorded in the framebuffer. A fog effect is independent of the absolute properties of a scene and depends solely on the distance from the point of observation, i.e. the very values recorded in a Z buffer. To build a scene with a rolling fog a volume texture will be required. Modern graphics hardware often includes the capability of rendering fog effects.

There are a number of possible models that simulate different fog conditions, such as a haze or mist. If Z_i is the depth of the point visible within pixel i then three models for fog give rise to a fraction f:

1. Exponential (Haze):
$$f = e^{-density \cdot Z_i}$$
2. Double Exponential (Dense fog):
$$f = e^{-density \cdot {Z_i}^2}$$
3. Pseudo fog, gives greater control and simulates a fog curtain:
$$f = \frac{d_{end} - Z_i}{d_{end} - dstart}$$

The fraction f is used to mix values in the framebuffer with the fog color as follows:

```
.
Image[i].Red   = f*Image[i].Red   + (1.0 - f)*FogColor_Red;
Image[i].Green = f*Image[i].Green + (1.0 - f)*FogColor_Green;
Image[i].Blue  = f*Image[i].Blue  + (1.0 - f)*FogColor_Blue;
.
```

0.5 Video effects

Video effects are image processes that modify a sequence of frames. The simplest video effect is to fade from one image or sequence of images into another. Sophisticated hardware devices are available that allow a huge variety of mixes and other special effects to be applied to a video signal in real time, i.e. 25-30 frames per second. These effects are all implemented on digital data streams using custom built processors which are directed by algorithms that would execute equally well on any processor (if not quite in real time).

It is the aim of this section to present some algorithms that can be used to perform a range of effects from simple cross fading up to a non-linear *swirl* action that gives the illusion of an image being stirred round its center.

Each algorithm operates through a standard interface which takes two pointers to framebuffers holding the input images and a parameter τ indicating the proportion of the effect to implement. It is usual to put the result into a separate output framebuffer of similar size. Some effects require a single input buffer, when more than two input streams are needed they can be accommodated by mixing them two at a time. The parameter τ is deemed to lie in the range $[0, 1]$ and therefore if the effect is to occur between say frames a and b the processing for frame $i : a \leq i < b$ will be determined using τ given by:

$$\tau = \frac{i - a}{a - b}$$

Using the structures and notation established in section 1.1 we write a function prototype for each video effect as follows:

```
void VideoEffect_NAME(
     lpPIXEL input1,  // Pointer to the first input framebuffer
     lpPIXEL input2,  // Pointer to the second input framebuffer
     lpPIXEL output,  // Pointer to the output framebuffer
     long    Width,   // Width of image
     long    Height,  // Height of image
     float   tau);    // parameter to specify the progress of effect
```

Function `VideoEffect_NAME()` calculates each pixel (i, j) in the output buffer sequentially by obtaining an appropriate combination of pixels from the input buffers. It is this combination that makes one effect different from another. The annotated source code listings that accompany the book contain functions for the following actions.

0.5.1 Mixes, wipes and 2D effects

The effects in this section are straightforward and can be followed from the source code listings that accompany the book. In the code and specifications below I1 refers to the first video channel and I2 to the second.

1. `BasicMix( )`
   The simplest effect possible. Mix images from I1 to I2 as the parameter changes from 0 to 1. The effect is illustrated in Figure 1.14.

2. `BasicLeftToRightWipe( )`
   Image I2 appears from the left side of the screen as the parameter changes $0 \rightarrow 1$ The function is easily modified for right to left mix The effect is illustrated in Figure 1.15.

3. `LeftToRightWipeWithSoftEdge( )`
   Image I2 appears from the left side of the screen as the parameter changes $0 \rightarrow 1$ the edge between the two images appears soft as the two images blend together over a small fraction of the width of the image.

4. `RightToLeftWipeWithSoftEdge( )`
   Image I2 appears from the right side of the screen as the parameter changes $0 \rightarrow 1$ the edge between the two images appears soft as the two images blend together over a small fraction of the width of the image.

5. `HorizontalWipeTowardsCenterWithSoftEdge( )`
   Image I2 appears from the left and right edges as the parameter changes $0 \rightarrow 1$ Edges between the two images appear soft since the two images blend together over a small fraction of the width of the image.

6. `HorizontalWipeAwayFromCenterWithSoftEdge( )`
   Image I2 appears in the center of screen and moves towards the left and right edges as the parameter changes $0 \rightarrow 1$ Edges between the two images appear soft as the two images blend together over a small fraction of the width of the image. The effect is illustrated in Figure 1.16.

7. `BasicTopToBottomWipe ( )`
   Image I2 appears from the top of the screen as the parameter changes $0 \rightarrow 1$ The function is easily modified for BottomToTopWipe

8. `VerticalWipeTowardsCenterWithSoftEdge( )`
   Image I2 appears from the top and bottom edges as the parameter changes $0 \rightarrow 1$ Edges between the two images appear soft as the two images blend together over a small fraction of the width of the image.

9. `VerticalWipeAwayFromCenterWithSoftEdge( )`
   Image I2 appears in a horizontal band at the center of the image. It expands towards the top and bottom of framebuffer as the parameter changes $0 \rightarrow 1$ Edges between the two images appears soft as the two images blend together over a small fraction of the width of the image.

10. `BasicSquareWipeFromCenter( )`
    Image I2 emerges from the center of the screen.

11. `CircularWipeToCentre( )`
    Image I1 fades away towards the center of the screen leaving I2 behind. Edges between the two images appears soft as the two images blend together over a small fraction of the width of the image.

12. `CircularWipeFromCentre( )`
    Image I2 appears from the center of the screen in a soft edged disk. The effect is illustrated in Figure 1.17.

13. `WipeWithRectangles()`
    Wipe the image by exposing the second image in an increasing number of rectangles that spread in lines down from the top left corner.

14. `MirrorImage( )`
    Make the output the mirror image of input I1.

15. `UpsideDown( )`
    Turn the image in I1 upside down.

Figure 14: Basic fade between the two images shown on the left.

Figure 15: Basic horizontal wipe between the two images shown on the left.

16. `SliceHorizontally(`
    Slice the image from I1 into strips siding the strips apart to reveal the image from I2

17. `RandomBlockMix( )`
    Mixes two images together by creating an increasing number of "squares" of I2 in I1 until eventually I2 replaces I1. The squares occur at random over the screen and can be any size from 1 pixel up. The effect is illustrated in Figure 1.18.

0.5.2 3D transformations: twists and spins

In a three-dimensional effect the image appears to be picked up off the screen, moved, twisted or rotated, and then set back down onto the screen. To execute the process two components are required:

1. A transformation matrix $[T]$ which describes the twists and spins. A suitable $[T]$ can be obtained by a combination (section **??**) of the transformations given in sections **??** and **??**.

Figure 16: Horizontal wipe from both sided of the image with a soft edge at the join.

Figure 17: A circular wipe between the two images on the left with a soft edge as the fade progresses.

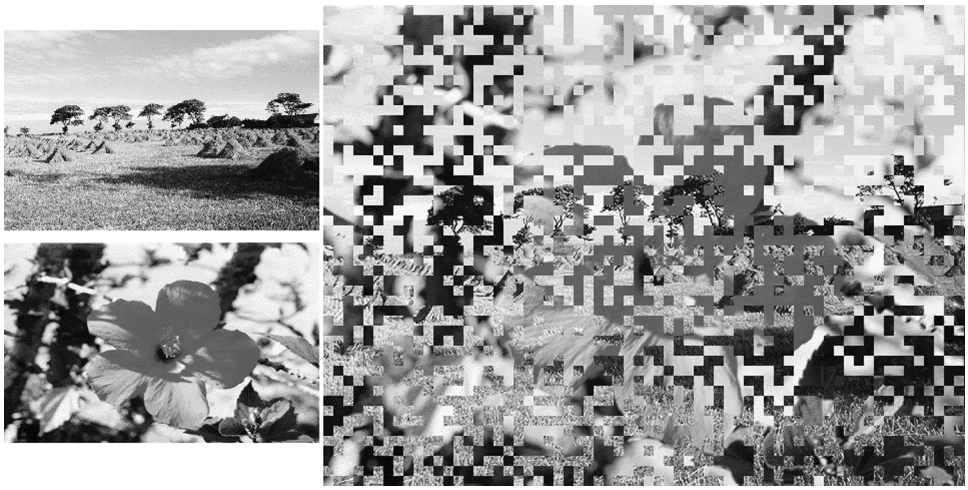

Figure 18: Wipe image in a number of rectangles placed randomly over the image.

2. A mapping function which, when given a coordinate (i, j) in the output framebuffer, uses $[T]$ to map this to a pixel in the original image whose RGB value is recorded in the output framebuffer at (i, j).

The theory that underlies these mapping functions is the same as that required for the planar image mapping discussed in Chapters **??** and **??**. Section **??** gave a description on how mapping coordinates which cover a plane are used to determine an RGB value at a specific location. Section **??** described how mapping coordinates are assigned to a plane lying in an arbitrary orientation. For the specific case of the 3D video transitions under discussion in this section these two stages in mapping can be combined into a single step without the need for a mesh or mapping coordinates.

To understand the action performed by the mapping function consider the setup shown in Figure 1.19. In (a), a representation of the input framebuffer is shown. The top left corner is designated as point $\mathbf{P}_p$, the top right corner as $\mathbf{P}_x$ and bottom left as $\mathbf{P}_y$. Vectors $\mathbf{\Delta x} = \mathbf{P}_x - \mathbf{P}_p$ and $\mathbf{\Delta y} = \mathbf{P}_y - \mathbf{P}_p$ lie along the top and left edges. Matrix $[T]$ is determined so that the points $\mathbf{P}_p$ etc. are moved to $\mathbf{P'}_p$ etc. as shown in (b). (For the specific case illustrated, the transformation is a combination of a rotation about an axis normal to the page and a translation).

Assuming that the input image is to appear inside the rectangle formed by the points $\mathbf{P'}_p$, $\mathbf{P'}_x$ and $\mathbf{P'}_y$ it is the job of the mapping function to:

1. Determine the transform $[T]$
2. Find its inverse $[T]^{-1}$
3. For each pixel (i, j) in the output buffer use $[T]^{-1}$ to determine the corresponding pixel in the original image (i', j') and write its value into the output buffer. Since some of the pixels (i, j) lie outside the rectangle formed by $\mathbf{P}_p$ etc. they have no equivalent coordinate in the original image. In these cases the function returns with an error code.

Important note: Mapping integer coordinates (i, j) with $[T]^{-1}$ to another set of integers is not satisfactory due to aliasing effects. In practice coordinates (i, j) are promoted to real numbers (x, y) which are them mapped to real numbers (x', y'). It is unlikely that these will be exact integers. Therefore, rather than simply choosing the image value from the nearest integer coordinate (i', j'), we can use bilinear interpolation to a obtain a value from the pixels at (i', j') and its nearest neighbors, this should result in a decrease in the unpleasant effects of aliasing.

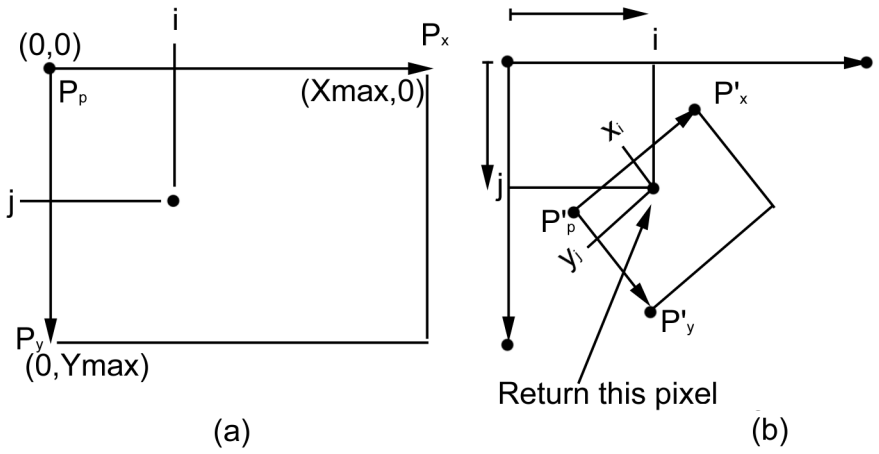

Figure 19: Mapping pseudo three-dimensional transformations.

Code for a version of the mapping function that can be used with the 3D processes described in this section accompanies the book.

Codes to execute examples of 3D video mixes and wipes are included with the book. Each function establishes vectors that lie in the plane of the screen and then transforms them with a matrix corresponding to rotations, translations and changes of scale. Once transformed the mapping function is used to determine what value is written into the output framebuffer. In the descriptions below, I1 refers to the first video channel, and I2 to the second. The effects in this class are:

1. `BasicPictureInPicture( )`
   This effect creates a small scaled down version of I1 inside the image I2. A small black border round I1 is added and the effect progressively reduces the size of the rectangle over time.
2. `ZoomOutFromImage( )`
   This effect creates the illusion of the image receding away into the distance. The effect is generated by gradually scaling down the size of the image and moving it towards the center of the screen.
3. `RotatePictureRoundZaxis( )`
   The image is rotated about an axis perpendicular to the screen. The effect is illustrated in Figure 1.20.
4. `Tumble( )`
   The image is tumbled by simultaneously rotating it about several axes and optionally scaling it down so that it appears to fly off into the distance. The effect is illustrated in Figure 1.21.

Many other combinations are possible in this class of video effects, for example, rotating round an axis running down the left hand side can appear to *partially turn the page* leaving space on the right to overlay and scroll text or other images.

0.5.3 Miscellaneous effects

A few video effects that are neither mixes, nor use a pseudo three-dimensional transformation are also included with the book, they are:

Figure 20: Rotating a picture around an axis perpendicular to the screen.

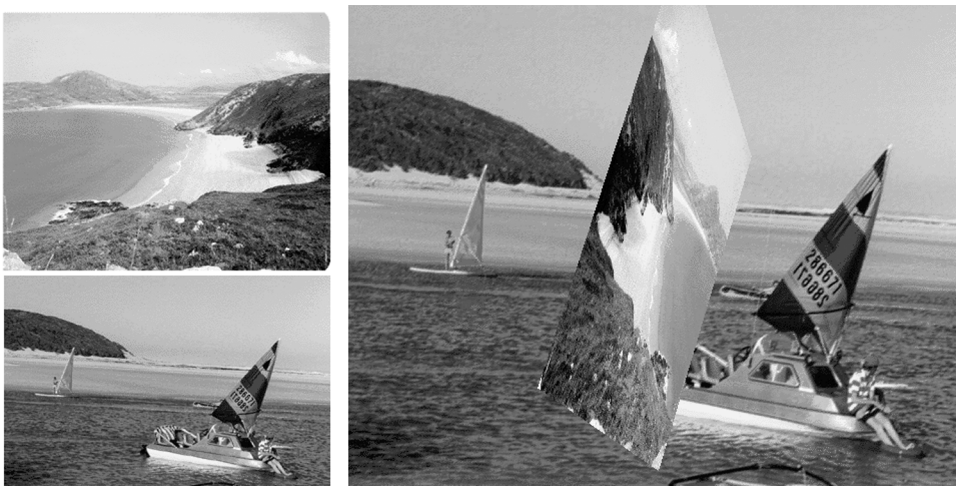

Figure 21: Tumbling an image by rotating around several axes simultaneously.

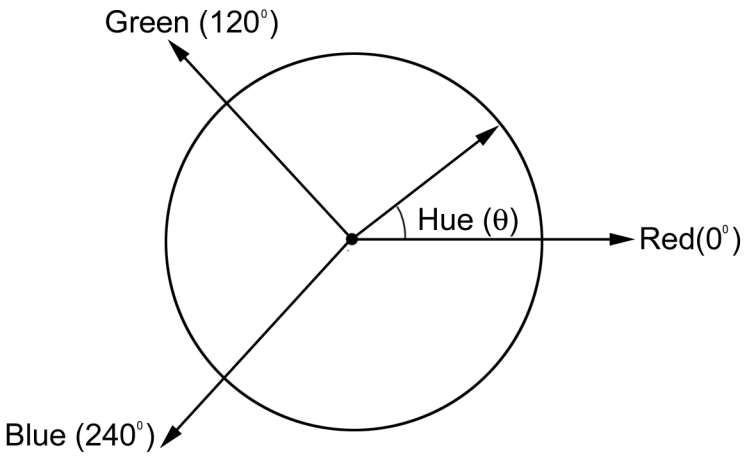

Figure 22: Describing color with a Hue, a unit vector that points towards a unit circle with a continuous spectrum of color drawn round it.

1. `FadeToMonochrome( )`
   Gradually remove the color from the input.

2. `FadeToNegative( )`
   Gradually make a photographic negative from the input.

3. `QuantizeColor( )`
   Fix the brightness and saturation of an image into certain bands. This function mimics the *colorizing* effect of digital video mixers. It also requires the use of the (H)ue, (S)aturation and (V)alue (HSV) method of specifying a color. HSV is a more natural way to describe a color than can be achieved by choosing RGB values. The *Value* is analogous to the brightness of a color e.g. bright red, dark red. The *Saturation* is a measure of how strong is the color, Red and Pink are both really a red color but Pink is less strongly red (the red being mixed with some white). The *Hue* is the parameter governing the color itself, Red, Green, Blue etc. Hue is generally thought of as a vector with the angle it makes to a base axis giving a value for a color measured in degrees. This is depicted in Figure 1.22.

4. `RotateHue( )`
   By incrementing the Hue the illusion of color cycling an image can be simulated. For example: If a Red raster is give as I1 the output can be made to change color through yellow, green, blue, purple and back to red over a time interval.

5. `Pixelate( )`
   Make the input image look like it is made up of large pixels. The color value for each large pixel is made by averaging all the actual pixels that lie within it. The effect is illustrated in Figure 1.23.

6. `RandomNoise( )`
   Make an output image that consists of nothing more than random noise. This effect is similar to the picture of a TV when no aerial is connected, or no transmitting station is selected.

7. `ScrollImage( )`
   Scroll the image towards the right, parts of the image that disappear off the right edge reappear at the left hand side. The effect is illustrated in Figure 1.24.

8. `RollImage( )`
   Roll the image down the display. Simulates a TV that has suffered a loss of vertical hold.

Figure 23: A pixelated image.

9. `StirImage( )`
   Transform the image as if it was being stirred round the center of the screen. This is a similar effect to what would happen to a drawing on the surface of a viscous liquid as it is gently stirred round. The effect is illustrated in Figure 1.25.

0.5.4 Artistic effects

Throughout this book we have looked at techniques which generate images of 3D models that try to approximate photographs of the object being modeled. However, in the real world despite the fact that the science of photography has been around for more than 100 years artists continue to flourish. Portrait, landscape and abstract pictures are as popular as ever, the interpretation by the artist adds an extra dimension that the perfect reproduction a photograph offers seems to sometimes miss. Artists will often use such features as; brush strokes, *quantized colors* and canvas textures etc. to add that personal touch. In this section we will look at the implementation of two such techniques, Pointillization and Crystallization. More details on these and other ideas, such as adding the illusion of brush strokes can be found in [4]. Many examples of work embellished with these techniques can be found in the gallery of the Computer Graphics World Magazine, and its associated web site [5].

Figures 1.27 and 1.28 illustrate the action of two effects which are called filters because they take an input image and process it point by point in much the same way as any other image processing algorithm. In fact both Crystallization and Pointillization are manifestations of the same algorithm with slightly changed parameters.

The idea behind both effects, is to sample the original image at random locations (i, j) and replace the color values at all pixels inside a circle of fixed radius centered on (i, j) with the value at pixel (i, j). If two or more circles overlap then an edge equidistant from their centers is found and it forms the boundary between the two regions. Where multiple circles overlap a series of irregular crystal grains begins to form. This process is illustrated in Figure 1.29.

Implementing this concept for an image stored in a framebuffer is quite straightforward if you imagine introducing a third dimension, perpendicular to the plane of the original image, and that the overlapping circles form the bases of a set of cones rising out of the plane of the image. Viewed *from above* these cones overlap and the lines of intersection (which are *straight* lines) resemble the crystal edges we wish to obtain, see Figure 1.30.

Figure 24: Scrolling the image. An image that matches along the left and right sides can use this effect to generated a scrolling background. Combine it with a left right blur filter to enhance the motion.

Figure 25: Swirling an image round an axis at right angles to and running through the center of the image.

Figure 26: A scanned photographic image.

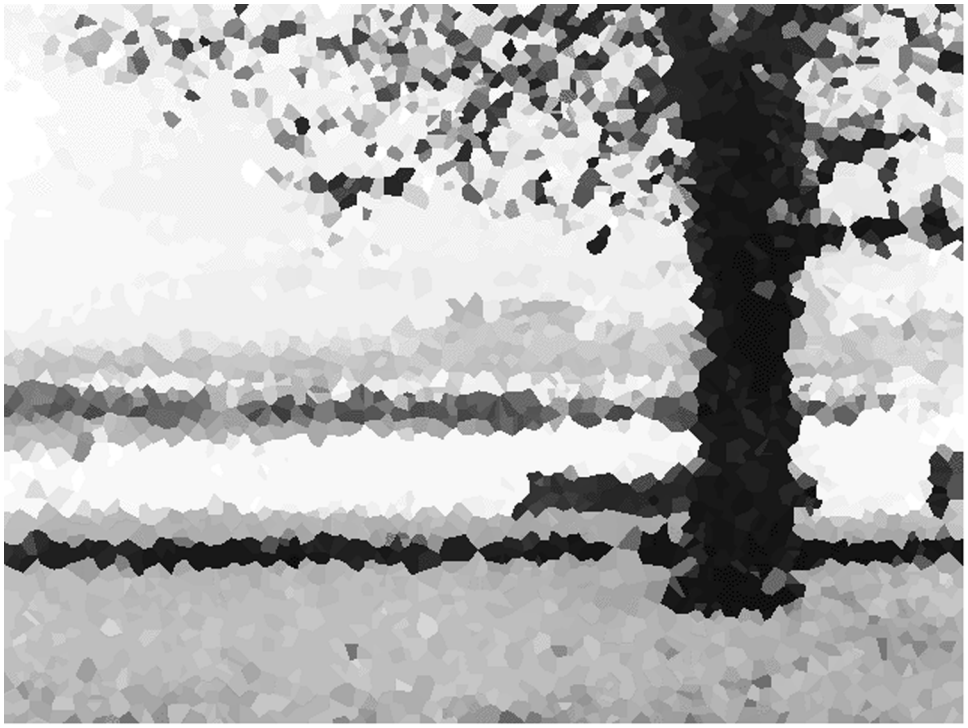

Figure 27: The picture from Figure 1.26 after processing with the Crystallization filter.

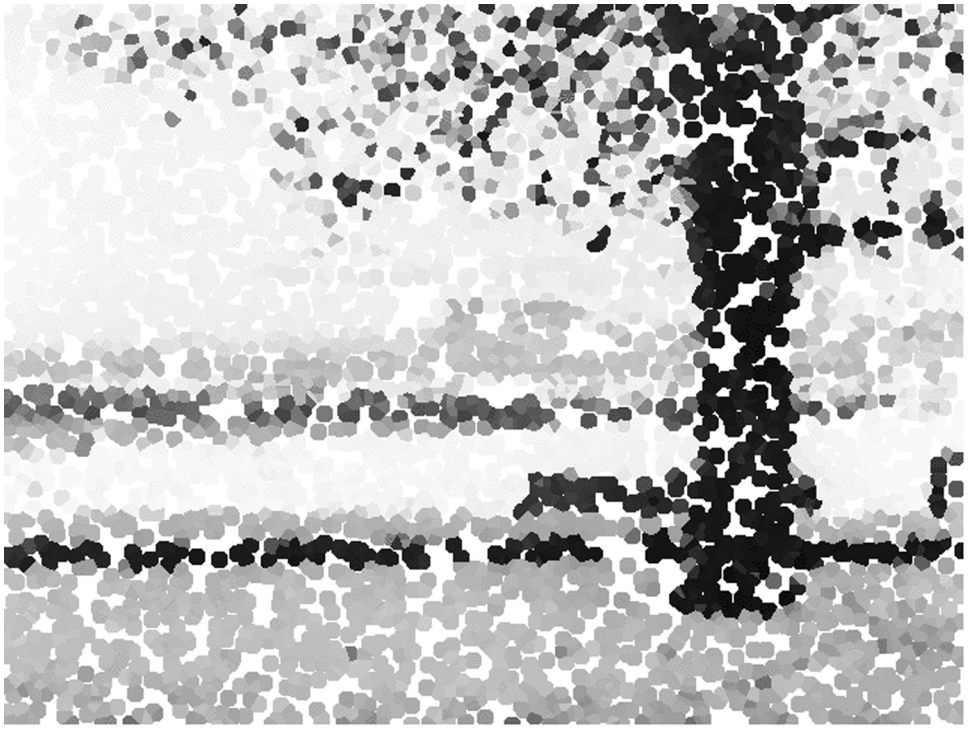

Figure 28: The picture from Figure 1.26 after processing with the Pointillization filter.

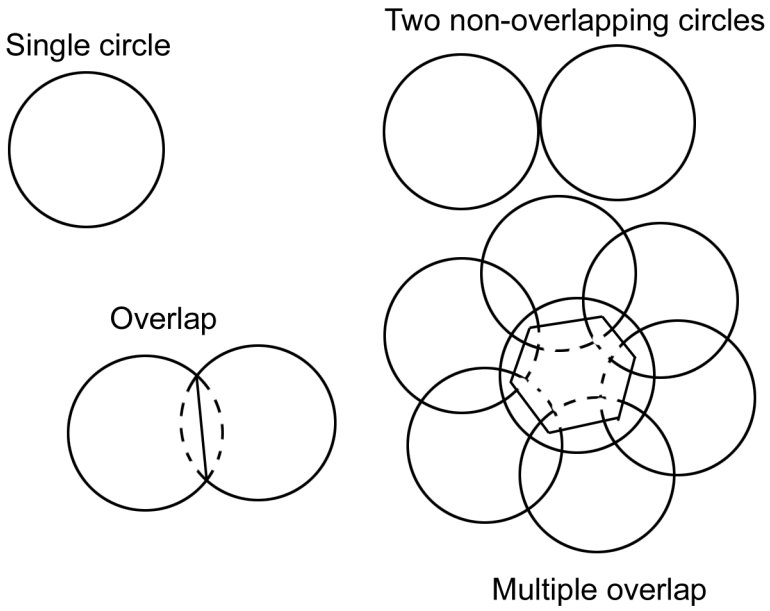

Figure 29: Forming crystal boundaries by overlapping circles.

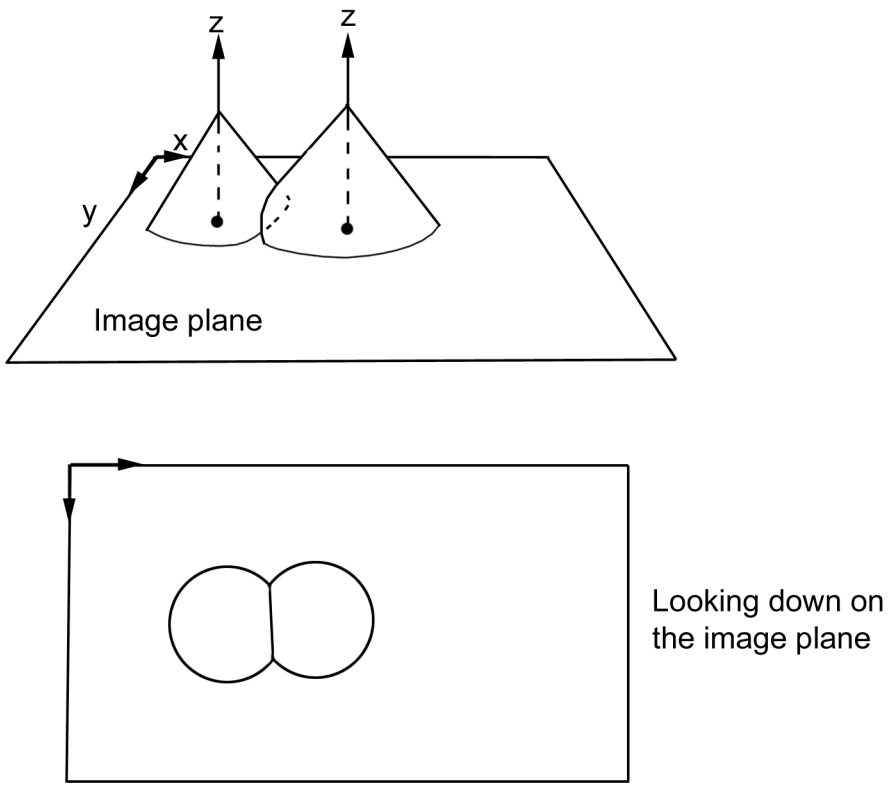

Figure 30: Overlapping cones in 3D form the necessary 3D pattern when viewed from above.

Let the n cones base radius be r and the
maximum cone height also be r. Then for each
pixel (i, j) repeat the following algorithm:

Let each cone k have axis at pixel (x_k, y_k)

Set $z = 0$ and cone $id = -1$
repeat the following for $k = 0$ until $k = n$ {
    calculate d, the distance between pixel (i, j)
            and pixel (x_k, y_k)
    if $d < r$ then {
        if $r - d > z$ then {
            set $z = r - d$
            set cone $id = k$
        }
    }
}
if cone $id \geq 0$ then {
    pixel (i, j) is set to color value for
    cone id i.e. The color value at pixel (x_{id}, y_{id})
}

Figure 31: Use a Z buffer to determine which one of n cones is visible at pixel coordinate (i, j)

The implementation of the effect is achieved using a Z buffer: For any pixel (i, j), we use the Z buffer to determine which cone is visible at coordinate (i, j) when looking from above because at that point it is the cone that has attained the greatest height which will be visible. Figure 1.31 summarizes the steps necessary to determine which, of n possible cones, is the visible one.

If the number of cones is high then it might be appropriate to assign a Z buffer covering the whole image and work by looking at each cone in turn to see what pixels it covers rather than looking at each pixel and seeing which cones cover it.

The difference between a Pointillized and Crystallized filter lies in the number of cones placed over the image. The radius of the base of the cone determines how large the crystals or points are. For the crystallization effect there must be enough cones to cover every pixel and they must have sufficient overlap so that every cone is overlapped to some degree all around it. In the case of a Pointillized appearance some gaps should be left between cones so that the background color is visible and some of the curved bases of the cones remain evident.

For both effects a large cone base radius will imply a large crystal size and much less of a resemblance to the original picture. In the limit, a cone size of less than one pixel for its base radius will show no filter effect at all. Pointillized pictures usually benefit from adding a small random variation to the average color used to fill each region covered by a cone. This results in a slight color variation even within regions where samples taken at two different pixels have the same values.

There are an almost infinite variety of other filters that could be built into this framework, the only limitation is your imagination.

References

[1] C. A. Lindley, *Practical Image Processing in C.* John Wiley and Sons, New York, 1991, Page 369.

[2] G. A. Baxes, *Digital Image Processing, A Practical Primer.* Prentice Hall, Englewood Cliffs, NJ, 1984.

[3] R. K. Castleman, *Digital Image Processing.* Prentice Hall, Englewood Cliffs, NJ, 1979.

[4] P. Haeberli, *Paint By Numbers: Abstract Image Representations.* Computer Graphics, Vol. 24, No. 4, Aug. 1990.

[5] *Computer Graphics World* published by PenWell Press, Nashua NH, *http://www.cgw.com*