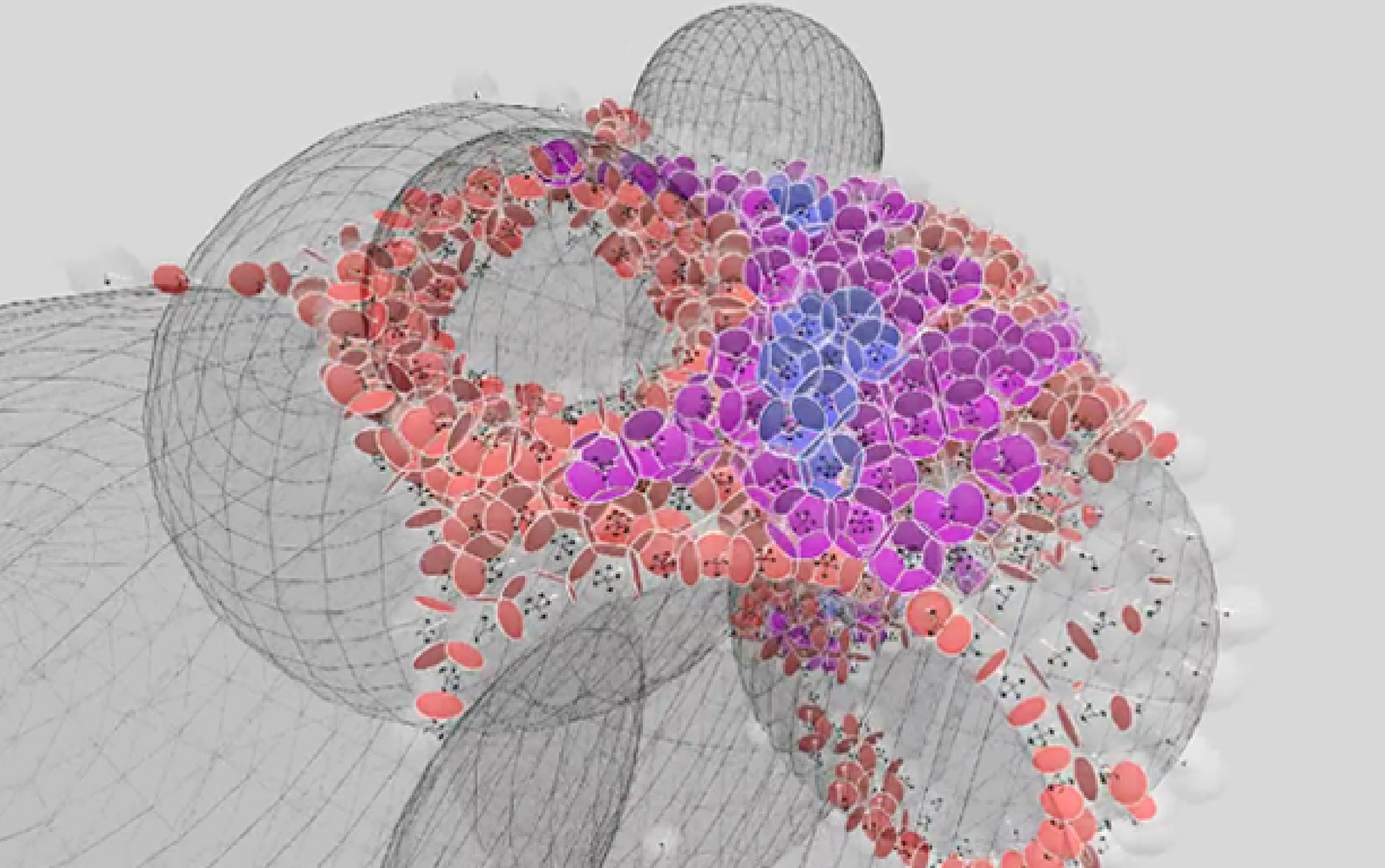
Computer Graphics Algorithms and Image Processing

Rozella Hedrick
Gregory Collett



First Edition, 2012

ISBN 978-81-323-1268-0

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Alpha Compositing and Bresenham's Line Algorithm

Alpha compositing

Example image demonstrating hue and alpha channels

In computer graphics, **alpha compositing** is the process of combining an image with a background to create the appearance of partial transparency. It is often useful to render image elements in separate passes, and then combine the resulting multiple 2D images into a single, final image in a process called compositing. For example, compositing is used extensively when combining computer rendered image elements with live footage.

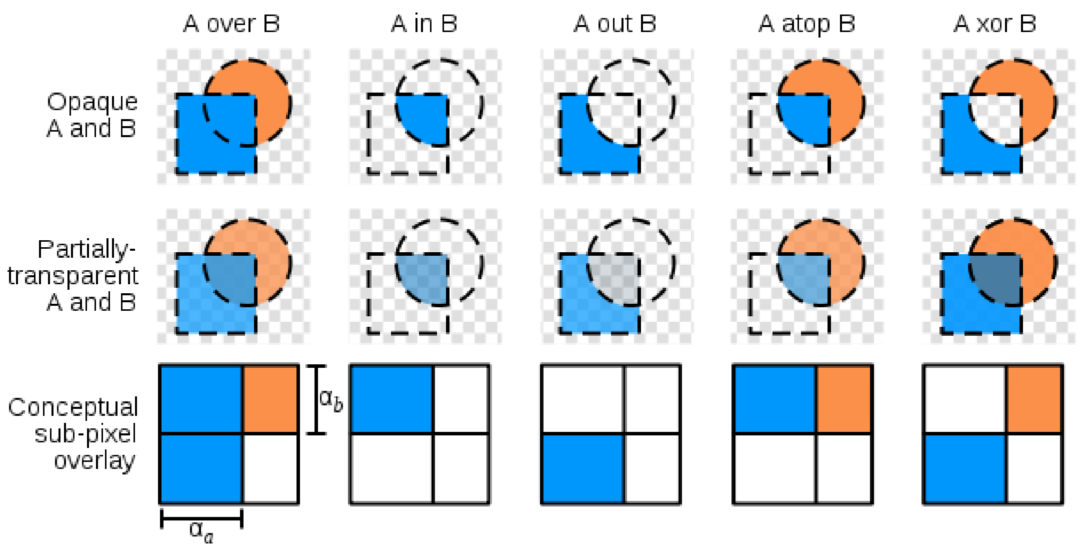
In order to combine these image elements correctly, it is necessary to keep an associated *matte* for each element. This matte contains the coverage information—the shape of the geometry being drawn—making it possible to distinguish between parts of the image where the geometry was actually drawn and other parts of the image which are empty.

Description

To store matte information, the concept of an **alpha channel** was introduced by Alvy Ray Smith in the late 1970s, and fully developed in a 1984 paper by Thomas Porter and Tom Duff. In a 2D image element which stores a color for each pixel, additional data is stored in the alpha channel with a value between 0 and 1. A value of 0 means that the pixel does not have any coverage information and is transparent; i.e. there was no color contribution from any geometry because the geometry did not overlap this pixel. A value of 1 means that the pixel is opaque because the geometry completely overlapped the pixel.

If an alpha channel is used in an image, it is common to also multiply the color by the alpha value, to save on additional multiplications during compositing. This is usually referred to as *premultiplied alpha*. Thus, assuming that the pixel color is expressed using RGBA tuples, a pixel value of (0.0, 0.5, 0.0, 0.5) implies a pixel which is half green and has 50% coverage. (Explanation: The RGB values are the first three values, (0, 0.5, 0) and the alpha value is the fourth, 0.5. If the color were fully green, its RGB would be (0, 1, 0). Since this pixel is using a premultiplied alpha, all of the RGB values in the ordered triplet (0, 1, 0) are multiplied by 0.5 and then the alpha is added to the end to yield (0, 0.5, 0, 0.5).) Premultiplied alpha also has some advantages over normal alpha blending because premultiplied alpha blending is associative and linear interpolation gives better results.

With the existence of an alpha channel, it is possible to express compositing image operations, using a *compositing algebra*. For example, given two image elements A and B, the most common compositing operation is to combine the images such that A appears in the foreground and B appears in the background. This can be expressed as A **over** B. In addition to **over**, Porter and Duff defined the compositing operators **in**, **held out by** (usually abbreviated **out**), **atop**, and **xor** (and the reverse operators **rover**, **rin**, **rout**, and **ratop**) from a consideration of choices in blending the colors of two pixels when their coverage is, conceptually, overlaid orthogonally:



The **over** operator is, in effect, the normal painting operation. The **in** operator is the alpha compositing equivalent of clipping.

As an example, the **over** operator can be accomplished by applying the following formula to each pixel value:

$$C_o = C_a \alpha_a + C_b \alpha_b (1 - \alpha_a)$$

where C_o is the result of the operation, C_a is the color of the pixel in element A, C_b is the color of the pixel in element B, and α_a and α_b are the alpha of the pixels in elements A and B respectively. If it is assumed that all color values are premultiplied by their alpha values ($c_i = \alpha_i C_i$), we can rewrite this as:

$$C_o = c_a + (1 - \alpha_a) c_b$$

and

$$\alpha_o = c_o / C_o = \alpha_a + \alpha_b (1 - \alpha_a)$$

However, this operation may not be appropriate for all applications, since it is not associative. The associative version of this operation is very similar; simply take the newly computed color value and divide it by its new alpha value, as follows:

$$C_o = \frac{C_a \alpha_a + C_b \alpha_b (1 - \alpha_a)}{\alpha_o}$$

Image editing applications that allow reordering of layers generally prefer this second approach.

Analytical derivation of the over operator

Porter and Duff gave a geometric interpretation of the alpha compositing formula by studying orthogonal coverages. Another derivation of the formula, based on a physical reflectance/transmittance model, can be found in a 1981 paper by Bruce A. Wallace.

A third approach is found by starting out with two very simple assumptions. For simplicity, we shall here use the shorthand notation $a \odot b$for representing the **over** operator.

The first assumption is that in the case where the background is opaque (i.e. $\alpha_b = 1$), the over operator represents the convex combination of *a* and *b*:

$$C_o = \alpha_a C_a + (1 - \alpha_a) C_b$$

The second assumption is that the operator must respect the associative rule:

$$(a \odot b) \odot c = a \odot (b \odot c)$$

Now, let us assume that *a* and *b* have variable transparencies, whereas *c* is opaque. We're interested in finding

$$o = a \odot b.$$

We know from the associative rule that the following must be true:

$$o \odot c = a \odot (b \odot c)$$

We know that *c* is opaque and thus follows that $b \odot c$is opaque, so in the above equation, each $\odot$operator can be written as a convex combination:

$$\begin{aligned}\alpha_o C_o + (1 - \alpha_o) C_c &= \alpha_a C_a + (1 - \alpha_a)(\alpha_b C_b + (1 - \alpha_b) C_c) \\ &= \alpha_a C_a + (1 - \alpha_a)\alpha_b C_b + (1 - \alpha_a)(1 - \alpha_b) C_c\end{aligned}$$

Hence we see that this represents an equation of the form $X_0 + Y_0 C_c = X_1 + Y_1 C_c$. By setting $X_0 = X_1$ and $Y_0 = Y_1$ we get

$$\begin{aligned}\alpha_o &= 1 - (1 - \alpha_a)(1 - \alpha_b), \\ C_o &= \frac{\alpha_a C_a + (1 - \alpha_a)\alpha_b C_b}{\alpha_o},\end{aligned}$$

which means that we have analytically derived a formula for the output alpha and the output color of $a \odot b$.

An even more compact representation is given by noticing that $(1 - \alpha_a)\alpha_b = \alpha_o - \alpha_a$:

$$C_o = \frac{\alpha_a}{\alpha_o} C_a + \left(1 - \frac{\alpha_a}{\alpha_o}\right) C_b$$

It is also interesting to note that the $\odot$operator fulfills all the requirements of a non-commutative monoid, where the identity element e is chosen such that $e \odot a = a \odot e = a$(i.e. the identity element can be any tuple $\langle C, \alpha \rangle$with $\alpha = 0$.)

Alpha blending

Alpha blending is a convex combination of two colors allowing for transparency effects in computer graphics. The value of `alpha` in the color code ranges from 0.0 to 1.0, where 0.0 represents a fully transparent color, and 1.0 represents a fully opaque color.

The value of the resulting color when color `Value1` with an alpha value of α is drawn over an opaque background of color `Value0` is given by:

$$\text{Value} = (1 - \alpha)\,\text{Value}_0 + \alpha\,\text{Value}_1$$

The alpha component may be used to blend to red, green and blue components equally, as in 32-bit RGBA, or, alternatively, there may be three alpha values specified corresponding to each of the primary colors for spectral color filtering.

Other transparency methods

Although used for similar purposes, transparent colors and image masks do not permit the smooth blending of the superimposed image pixels with those of the background (only whole image pixels or whole background pixels allowed).

A similar effect can be achieved with a 1-bit alpha channel, as found in the 16-bit RGBA Highcolor mode of the Truevision TGA image file format and related TARGA and AT-Vista/NU-Vista display adapters' Highcolor graphic mode. This mode devotes 5 bits for every primary RGB color (15-bit RGB) plus a remaining bit as the "alpha channel".

For some applications, a single alpha channel is not sufficient: a stained-glass window, for instance, requires a separate transparency channel for each RGB channel to model the red transparency, green transparency, and blue transparency. More alpha channels can be added for accurate spectral color filtration applications.

Applications

Alpha blending is used in a variety of applications. It is natively supported by these operating systems/GUIs for drawing windows (where applicable) or widgets:

- AmigaOS 4.1
- Web OS
- Android (Operating System)
- BeOS, Zeta and Haiku
- Inferno
- iOS
- Mac OS X
- MorphOS
- Plan 9
- QNX Neutrino
- RISC OS Adjust
- Syllable
- Windows 2000, XP, Server 2003, Windows CE, Windows Mobile, Vista and Windows 7
- The XRender extension to the X Window System (this includes modern Linux systems)

Other software may use alpha blended transparent elements in the GUI independently of OS provided API's by precomposing elements in an off-screen memory buffer before displaying them. (Such as when displaying partially transparent composited elements in an embedded system which provides only a simple frame buffer.) Compositing software is used to combine images, and makes extensive use of alpha compositing techniques.

Bresenham's line algorithm

The **Bresenham line algorithm** is an algorithm which determines which points in an n-dimensional raster should be plotted in order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is one of the earliest algorithms developed in the field of computer graphics. A minor extension to the original algorithm also deals with drawing circles.

While algorithms such as Wu's algorithm are also frequently used in modern computer graphics because they can support antialiasing, the speed and simplicity of Bresenham's line algorithm mean that it is still important. The algorithm is used in hardware such as plotters and in the graphics chips of modern graphics cards. It can also be found in many software graphics libraries. Because the algorithm is very simple, it is often implemented in either the firmware or the hardware of modern graphics cards.

The label "Bresenham" is used today for a whole family of algorithms extending or modifying Bresenham's original algorithm.

The algorithm

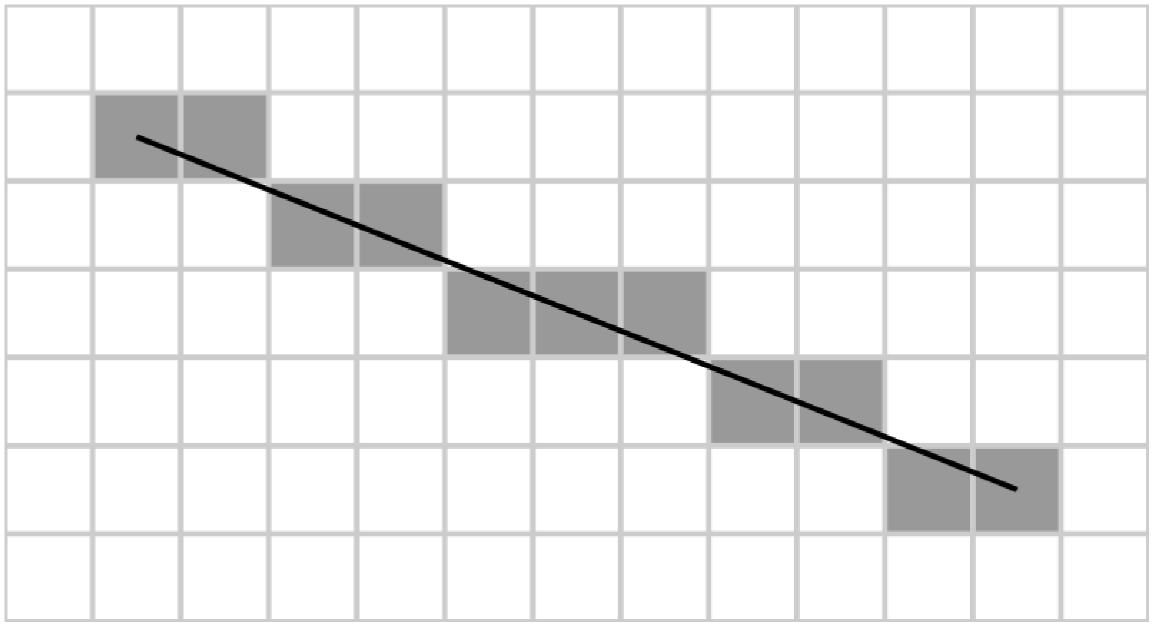


Illustration of the result of Bresenham's line algorithm. (0,0) is at the top left corner.

The common conventions that pixel coordinates increase in the down and right directions (e.g. that the pixel at (1,1) is directly above the pixel at (1,2)) and that the pixel centers that have integer coordinates will be used. The endpoints of the line are the pixels at (x_0, y_0) and (x_1, y_1), where the first coordinate of the pair is the column and the second is the row.

The algorithm will be initially presented only for the octant in which the segment goes down and to the right ($x_0 \leq x_1$ and $y_0 \leq y_1$), and its horizontal projection $x_1 - x_0$ is longer than the vertical projection $y_1 - y_0$ (the line has a slope whose absolute value is less than 1 and greater than 0.) In this octant, for each column x between x_0 and x_1, there is exactly one row y (computed by the algorithm) containing a pixel of the line, while each row between y_0 and y_1 may contain multiple rasterized pixels.

Bresenham's algorithm chooses the integer y corresponding to the pixel center that is closest to the ideal (fractional) y for the same x; on successive columns y can remain the same or increase by 1. The general equation of the line through the endpoints is given by:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}.$$

Since we know the column, x, the pixel's row, y, is given by rounding this quantity to the nearest integer:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0.$$

The slope $(y_1 - y_0) / (x_1 - x_0)$ depends on the endpoint coordinates only and can be precomputed, and the ideal y for successive integer values of x can be computed starting from y_0 and repeatedly adding the slope.

In practice, the algorithm can track, instead of possibly large y values, a small *error value* between −0.5 and 0.5: the vertical distance between the rounded and the exact y values for the current x. Each time x is increased, the error is increased by the slope; if it exceeds 0.5, the rasterization y is increased by 1 (the line continues on the next lower row of the raster) and the error is decremented by 1.0.

In the following pseudocode sample `plot(x,y)` plots a point and `abs` returns absolute value:

```
 function line(x0, x1, y0, y1)
     int deltax := x1 - x0
     int deltay := y1 - y0
     real error := 0
     real deltaerr := deltay / deltax    // Assume deltax != 0 (line is
not vertical),
           // note that this division needs to be done in a way that
preserves the fractional part
     int y := y0
     for x from x0 to x1
         plot(x,y)
         error := error + deltaerr
         if abs(error) ≥ 0.5 then
             y := y + 1
             error := error - 1.0
```

Generalization

The version above only handles lines that descend to the right. We would of course like to be able to draw all lines. The first case is allowing us to draw lines that still slope downwards but head in the opposite direction. This is a simple matter of swapping the initial points if `x0 > x1`. Trickier is determining how to draw lines that go up. To do this, we check if $y_0 \geq y_1$; if so, we step y by -1 instead of 1. Lastly, we still need to generalize the algorithm to drawing lines in *all* directions. Up until now we have only been able to draw lines with a slope less than one. To be able to draw lines with a steeper slope, we take advantage of the fact that a steep line can be reflected across the line $y=x$ to obtain a line with a small slope. The effect is to switch the x and y variables throughout, including switching the parameters to *plot*. The code looks like this:

```
 function line(x0, x1, y0, y1)
     boolean steep := abs(y1 - y0) > abs(x1 - x0)
     if steep then
         swap(x0, y0)
         swap(x1, y1)
     if x0 > x1 then
         swap(x0, x1)
         swap(y0, y1)
     int deltax := x1 - x0
     int deltay := abs(y1 - y0)
     real error := 0
     real deltaerr := deltay / deltax
     int ystep
     int y := y0
     if y0 < y1 then ystep := 1 else ystep := -1
     for x from x0 to x1
         if steep then plot(y,x) else plot(x,y)
         error := error + deltaerr
         if error ≥ 0.5 then
             y := y + ystep
             error := error - 1.0
```

The function now handles all lines and implements the complete Bresenham's algorithm.

Optimization

The problem with this approach is that computers operate relatively slowly on fractional numbers like `error` and `deltaerr`; moreover, errors can accumulate over many floating-point additions. Working with integers will be both faster and more accurate. The trick we use is to multiply all the fractional numbers above by `deltax`, which enables us to express them as integers. The only problem remaining is the constant 0.5—to deal with this, we change the initialization of the variable `error`, and invert it for an additional small optimization. The new program looks like this:

```
 function line(x0, x1, y0, y1)
     boolean steep := abs(y1 - y0) > abs(x1 - x0)
     if steep then
         swap(x0, y0)
         swap(x1, y1)
     if x0 > x1 then
         swap(x0, x1)
         swap(y0, y1)
     int deltax := x1 - x0
     int deltay := abs(y1 - y0)
     int error := deltax / 2
     int ystep
     int y := y0
     if y0 < y1 then ystep := 1 else ystep := -1
     for x from x0 to x1
         if steep then plot(y,x) else plot(x,y)
         error := error - deltay
         if error < 0 then
             y := y + ystep
             error := error + deltax
```

Remark: If you need to control the points **in order of appearance** (for example to print several consecutive dashed lines) you will have to simplify this code by skipping the 2nd swap:

```
function line(x0, x1, y0, y1)
     boolean steep := abs(y1 - y0) > abs(x1 - x0)
     if steep then
         swap(x0, y0)
         swap(x1, y1)
       if x0 > x1 then
           swap(x0, x1)
           swap(y0, y1)
     int deltax := abs(x1 - x0)
     int deltay := abs(y1 - y0)
     int error := deltax / 2
     int ystep
     int y := y0

     int inc REM added
     if x0 < x1 then inc := 1 else inc := -1 REM added

     if y0 < y1 then ystep := 1 else ystep := -1
     for x from x0 to x1 with increment inc REM changed
         if steep then plot(y,x) else plot(x,y)
         REM increment here a variable to control the progress of the
line drawing
         error := error - deltay
         if error < 0 then
             y := y + ystep
             error := error + deltax
```

Simplification

It is further possible to eliminate the `swap`s in the initialisation by considering the error calculation for both directions simultaneously:

```
 function line(x0, y0, x1, y1)
   dx := abs(x1-x0)
   dy := abs(y1-y0)
   if x0 < x1 then sx := 1 else sx := -1
   if y0 < y1 then sy := 1 else sy := -1
   if dx > dy then err := dx/2 else err := -dy/2

   loop
     setPixel(x0,y0)
     if x0 = x1 and y0 = y1 exit loop
     e2 := err
     if e2 > -dx then
       err := err - dy
       x0 := x0 + sx
     if e2 <  dy then
       err := err + dx
       y0 := y0 + sy
   end loop
```

History

The algorithm was developed by Jack E. Bresenham in 1962 at IBM. In 2001 Bresenham wrote:

I was working in the computation lab at IBM's San Jose development lab. A Calcomp plotter had been attached to an IBM 1401 via the 1407 typewriter console. [The algorithm] was in production use by summer 1962, possibly a month or so earlier. Programs in those days were freely exchanged among corporations so Calcomp (Jim Newland and Calvin Hefte) had copies. When I returned to Stanford in Fall 1962, I put a copy in the Stanford comp center library. A description of the line drawing routine was accepted for presentation at the 1963 ACM national convention in Denver, Colorado. It was a year in which no proceedings were published, only the agenda of speakers and topics in an issue of Communications of the ACM. A person from the IBM Systems Journal asked me after I made my presentation if they could publish the paper. I happily agreed, and they printed it in 1965.

Bresenham's algorithm was later modified to produce circles, the resulting algorithm being sometimes known as either "Bresenham's circle algorithm" or midpoint circle algorithm.

Similar algorithms

The Bresenham algorithm can be interpreted as slightly modified DDA (using 0.5 as error threshold instead of 0, which is required for non-overlapping polygon rasterizing).

The principle of using an incremental error in place of division operations has other applications in graphics. It is possible to use this technique to calculate the U,V co-ordinates during raster scan of texture mapped polygons. The voxel heightmap software-rendering engines seen in some PC games also used this principle.

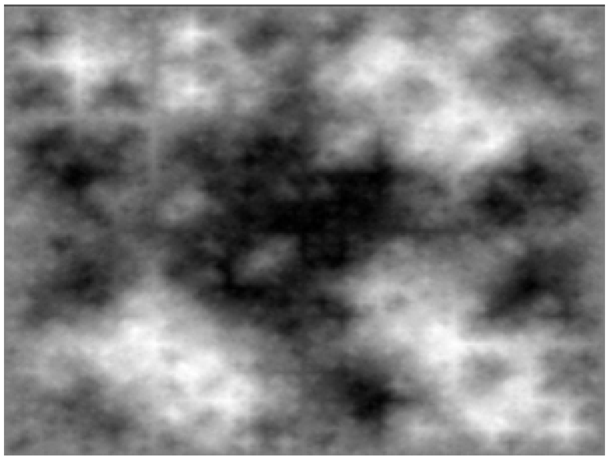
Bresenham also published a Run-Slice (as opposed to the Run-Length) computational algorithm.

An extension to the algorithm that handles thick lines was created by Alan Murphy at IBM

Chapter 2

Diamond-Square Algorithm and Flood Fill

Diamond-square algorithm



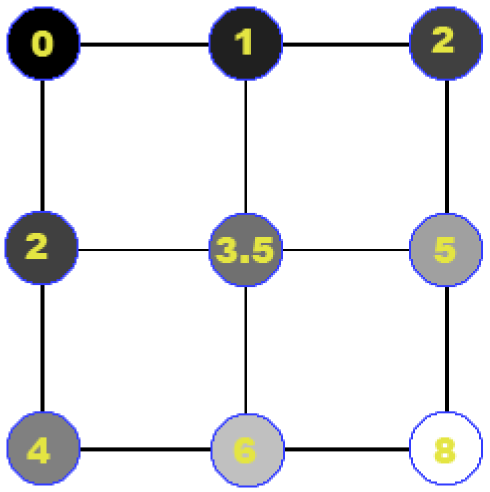
Plasma fractal

The **diamond-square algorithm** is a method for generating highly realistic heightmaps for computer graphics. It is a slightly better algorithm than the three-dimensional implementation of the midpoint displacement algorithm which produces two-dimensional landscapes. It is also known as the **random midpoint displacement fractal**, the **cloud fractal** or the **plasma fractal**, because of the plasma effect produced when applied.

The idea was first introduced by Fournier, Fussell and Carpenter at SIGGRAPH 1982. It was later analyzed by Gavin S. P. Miller in SIGGRAPH 1986 who described it as flawed.

The algorithm starts with a 2D grid then randomly generates terrain height from four seed values arranged in a grid of points so that the entire plane is covered in squares.

Midpoint displacement algorithm



Example on first iteration

- Assign a height value to each corner of the rectangle (image).
- Divide the rectangle into 4 subrectangles, and let their height values be the mean values of the corners of the parent rectangle.

  For example, the upper left sub-rectangle in
  $\begin{bmatrix} 0 & 2 \\ 4 & 8 \end{bmatrix}$will have the height values
  $\begin{bmatrix} 0 & (0+2)/2 \\ (0+4)/2 & (0+2+4+8)/4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3.5 \end{bmatrix}$
  But when computing the middle height, one should add a small error that depends on the size of the rectangle (the standard is to let the error be proportional to the size of the rectangle and some constant. The constant controls the "roughness" of the fractal; a bigger constant results in more valleys and mountains).

- Iterate and subdivide each rectangle into smaller ones. Eventually, they will be too small to produce a noticeable difference. When this occurs, stop the iteration, and render the pixel with the mean of the height values.

Diamond-square algorithm

The difference from the above algorithm is an intermediate step that regards diamond-shaped squares as well. This reduces the squared-shaped artifacts in the landscape, since the diamonds are rotated 45 degrees relative to the squares.

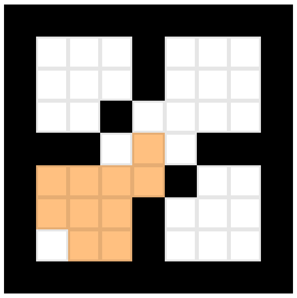
Applications

This algorithm can be used to generate realistic-looking landscapes, and different implementations are used in computer graphics software such as Terragen.

Papers

[Random Midpoint Displacement Method] Fournier,A., Fussel,D., and Carpenter, L. 1982. Computer Rendering of Stochastic Models. Communications of the ACM, 25: 371-384.

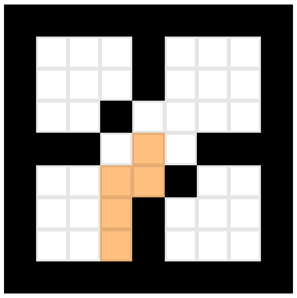
Flood fill



recursive flood-fill with 4 directions

Flood fill, also called ***seed fill,*** is an algorithm that determines the area connected to a given node in a multi-dimensional array. It is used in the "bucket" fill tool of paint programs to determine which parts of a bitmap to fill with color, and in games such as Go, Minesweeper, Puyo Puyo, Lumines, Samegame and Magical Drop for determining which pieces are cleared. When applied on an image to fill a particular bounded area with color, it is also known as ***boundary fill***.

The algorithm



recursive flood-fill with 8 directions

The flood fill algorithm takes three parameters: a start node, a target color, and a replacement color. The algorithm looks for all nodes in the array which are connected to the start node by a path of the target color, and changes them to the replacement color. There are many ways in which the flood-fill algorithm can be structured, but they all make use of a queue or stack data structure, explicitly or implicitly. One implicitly stack-based (recursive) flood-fill implementation (for a two-dimensional array) goes as follows:

```
Flood-fill (node, target-color, replacement-color):
 1. If the color of node is not equal to target-color, return.
 2. Set the color of node to replacement-color.
 3. Perform Flood-fill (one step to the west of node, target-color,
replacement-color).
    Perform Flood-fill (one step to the east of node, target-color,
replacement-color).
    Perform Flood-fill (one step to the north of node, target-color,
replacement-color).
    Perform Flood-fill (one step to the south of node, target-color,
replacement-color).
 4. Return.
```

Alternative implementations

Though easy to understand, the implementation of the algorithm used above is impractical in languages and environments where stack space is severely constrained (e.g. Java applets).

An explicitly queue-based implementation is shown in the pseudo-code below. This implementation is not very efficient, but can be coded quickly, does not use a stack, and it is easy to debug:

```
Flood-fill (node, target-color, replacement-color):
 1. Set Q to the empty queue.
 2. If the color of node is not equal to target-color, return.
 3. Add node to the end of Q.
 4. While Q is not empty:
 5.     Set n equal to the first element of Q
 6.     If the color of n is equal to target-color, set the color of n
to replacement-color.
 7.     Remove first element from Q
 8.     If the color of the node to the west of n is target-color:
 9.         Set the color of that node to replacement-color
10.         Add that node to the end of Q
11.     If the color of the node to the east of n is target-color:
12.         Set the color of that node to replacement-color
13.         Add that node to the end of Q
14.     If the color of the node to the north of n is target-color:
15.         Set the color of that node to replacement-color
16.         Add that node to the end of Q
17.     If the color of the node to the south of n is target-color:
18.         Set the color of that node to replacement-color
19.         Add that node to the end of Q
20. Return.
```

Most practical implementations use a loop for the west and east directions as an optimization to avoid the overhead of stack or queue management:

```
Flood-fill (node, target-color, replacement-color):
 1. Set Q to the empty queue.
 2. If the color of node is not equal to target-color, return.
 3. Add node to Q.
 4. For each element n of Q:
 5.  If the color of n is equal to target-color:
 6.   Set w and e equal to n.
 7.   Move w to the west until the color of the node to the west of w
no longer matches target-color.
 8.   Move e to the east until the color of the node to the east of e
no longer matches target-color.
 9.   Set the color of nodes between w and e to replacement-color.
10.   For each node n between w and e:
11.    If the color of the node to the north of n is target-color, add
that node to Q.
       If the color of the node to the south of n is target-color, add
that node to Q.
12. Continue looping until Q is exhausted.
13. Return.
```

Adapting the algorithm to use an additional array to store the shape of the region allows generalization to cover "fuzzy" flood filling, where an element can differ by up to a specified threshold from the source symbol. Using this additional array as an alpha channel allows the edges of the filled region to blend somewhat smoothly with the not-filled region.

Fixed memory method (right-hand fill method)

A method exists that uses essentially no memory for four-connected regions by pretending to be a painter trying to paint the region without painting themselves into a corner. This is also a method for solving mazes. The four pixels making the primary boundary are examined to see what action should be taken. The painter could find themselves in one of several conditions:

1. All four boundary pixels are filled.
2. Three of the boundary pixels are filled.
3. Two of the boundary pixels are filled.
4. One boundary pixel is filled.
5. Zero boundary pixels are filled.

Where a path or boundary is to be followed, the right-hand rule is used. The painter follows the region by placing their right-hand on the wall (the boundary of the region) and progressing around the edge of the region without removing their hand.

For case #1, the painter paints (fills) the pixel the painter is standing upon and stops the algorithm.

For case #2, a path leading out of the area exists. Paint the pixel the painter is standing upon and move in the direction of the open path.

For case #3, the two boundary pixels define a path which, if we painted the current pixel, may block us from ever getting back to the other side of the path. We need a "mark" to define where we are and which direction we are heading to see if we ever get back to exactly the same pixel. If we already created such a "mark", then we preserve our previous mark and move to the next pixel following the right-hand rule.

A mark is used for the first 2-pixel boundary that is encountered to remember where the passage started and in what direction the painter was moving. If the mark is encountered again and the painter is traveling in the same direction, then the painter knows that it is safe to paint the square with the mark and to continue in the same direction. This is because (through some unknown path) the pixels on the other side of the mark can be reached and painted in the future. The mark is removed for future use.

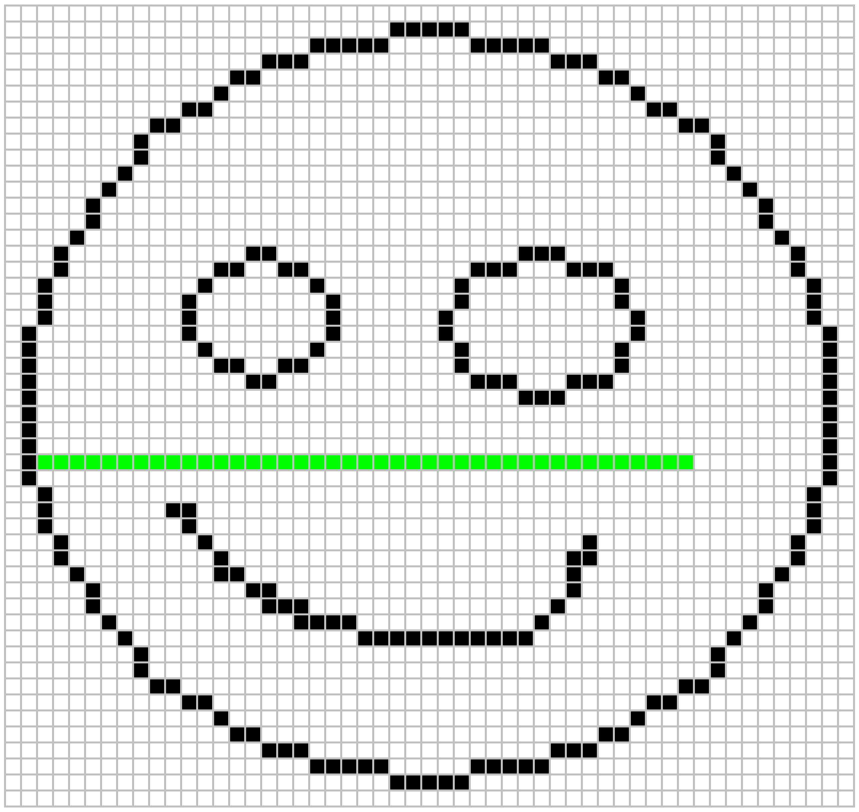
If the painter encounters the mark but is going in a different direction, then some sort of loop has occurred which caused the painter to return to the mark. This loop must be eliminated. The mark is picked up and the painter then proceeds in the direction indicated previously by the mark using a left-hand rule for the boundary (similar to the right-hand rule but using the painter's left hand). This continues until an intersection is found (with three or more open boundary pixels). Still using the left-hand rule the painter now searches for a simple passage (made by two boundary pixels). Upon finding this two-pixel boundary path, that pixel is painted. This breaks the loop and allows the algorithm to continue.

For case #4, we need to check the opposite 8-connected corners to see if they are filled or not. If either or both are filled, then this creates a many-path intersection and cannot be filled. If both are empty, then the current pixel can be painted and the painter can move following the right-hand rule.

The algorithm trades time for memory. For simple shapes it is very efficient. However, if the shape is complex with many features, the algorithm spends a large amount of time tracing the edges of the region trying to ensure that all can be painted.

This algorithm was first available commercially in 1981 on a Vicom Image Processing system manufactured by Vicom Systems, Inc. The classic recursive flood fill algorithm was available on this system as well.

Scanline fill



Scanline fill

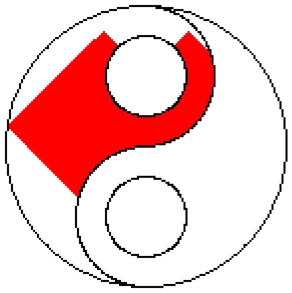
The algorithm can be sped up by filling lines. Instead of pushing each potential future pixel coordinate on the stack, it inspects the neighbour lines (previous and next) to find adjacent segments that may be filled in a future pass; the coordinates (either the start or the end) of the line segment are pushed on the stack. In most cases this scanline algorithm is at least an order of magnitude faster than the per-pixel one.

Efficiency : each pixel is checked once.

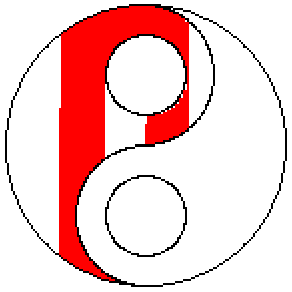
Vector implementations

Version 0.46 of Inkscape includes a bucket fill tool, giving output similar to ordinary bitmap operations and indeed using one: the canvas is rendered, a flood fill operation is performed on the selected area and the result is then traced back to a path. It uses the concept of a boundary condition.

Large scale behaviour



Four-way floodfill using a queue for storage



Four-way floodfill using a stack for storage

The primary technique used to control a flood fill will either be data-centric or process-centric. A data-centric approach can use either a stack or a queue to keep track of seed pixels that need to be checked. A process-centric algorithm must necessarily use a stack.

A 4-way floodfill algorithm that uses the adjacency technique and a queue as its seed pixel store yields an expanding lozenge-shaped fill.

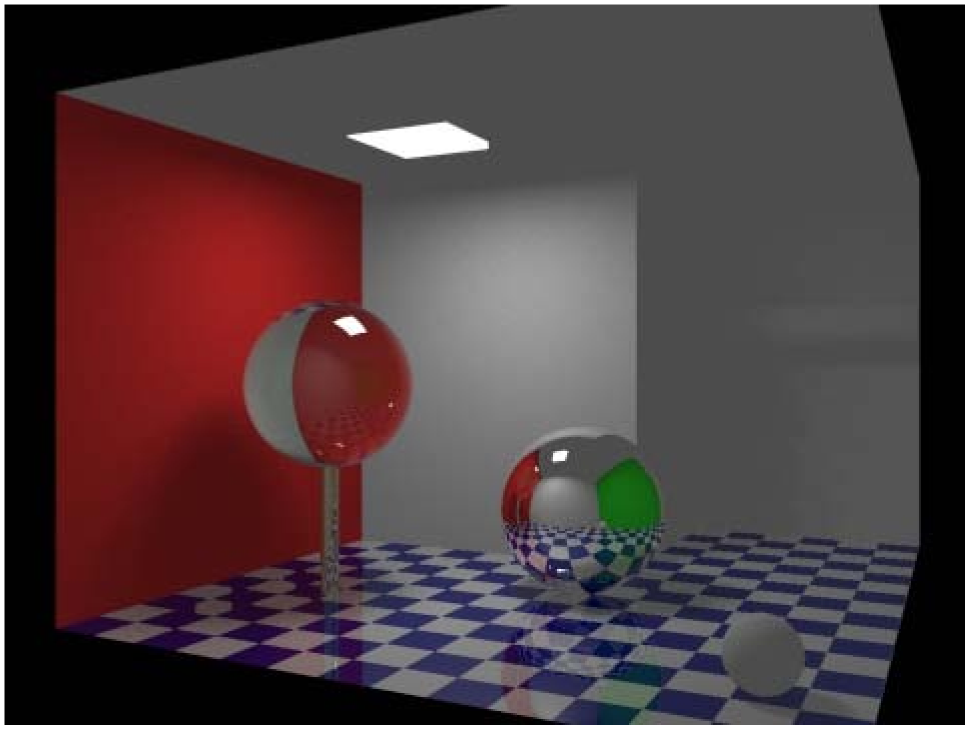
Efficiency : 4 pixels checked for each pixel filled (8 for an 8-way fill).

A 4-way floodfill algorithm that use the adjacency technique and a stack as its seed pixel store exhibits a characteristic "leave gaps and then return to fill them later" behaviour. This approach can be particularly seen in older 8-bit computer games.

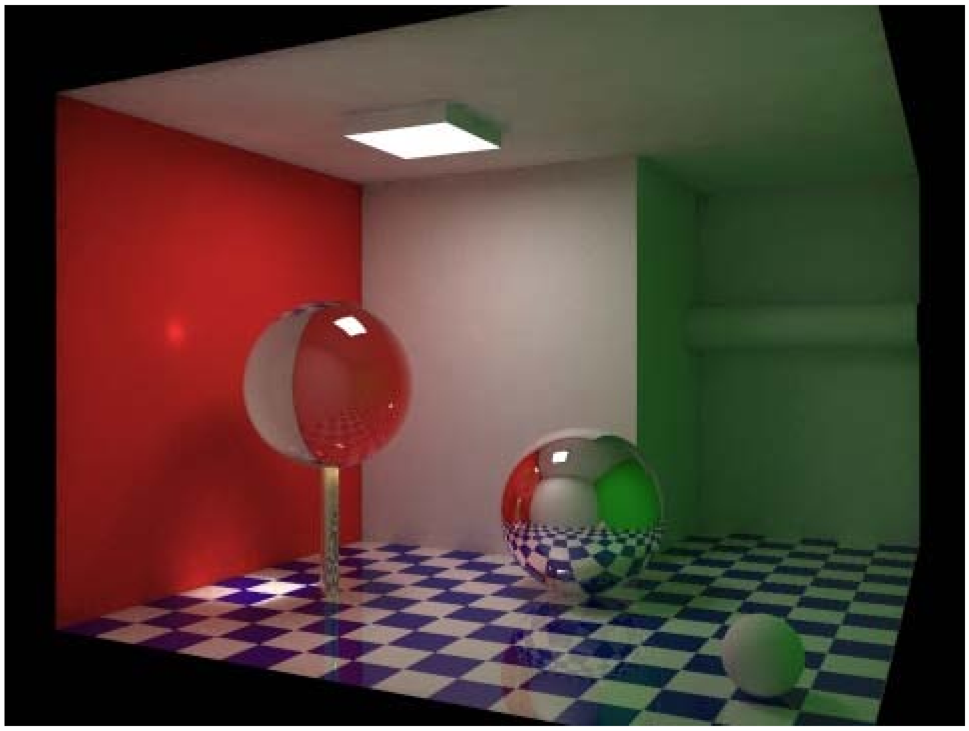
Chapter 3

Global Illumination and Graftal

Global illumination



Rendering without global illumination. Areas that lie outside of the ceiling lamp's direct light lack definition. For example, the lamp's housing appears completely uniform. Without the ambient light added into the render, it would appear uniformly black.



Rendering with global illumination. Light is reflected by surfaces, and colored light transfers from one surface to another. Notice how color from the red wall and green wall (not visible) reflects onto other surfaces in the scene. Also notable is the caustic projected onto the red wall from light passing through the glass sphere.

Global illumination is a general name for a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light which comes directly from a light source (*direct illumination*), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or non (*indirect illumination*).

Theoretically reflections, refractions, and shadows are all examples of global illumination, because when simulating them, one object affects the rendering of another object (as opposed to an object being affected only by a direct light). In practice, however, only the simulation of diffuse inter-reflection or caustics is called global illumination.

Images rendered using global illumination algorithms often appear more photorealistic than images rendered using only direct illumination algorithms. However, such images are computationally more expensive and consequently much slower to generate. One common approach is to compute the global illumination of a scene and store that information with the geometry, i.e., radiosity. That stored data can then be used to generate images from different viewpoints for generating walkthroughs of a scene without having to go through expensive lighting calculations repeatedly.

Radiosity, ray tracing, beam tracing, cone tracing, path tracing, Metropolis light transport, ambient occlusion, photon mapping, and image based lighting are examples of algorithms used in global illumination, some of which may be used together to yield results that are fast, but accurate.

These algorithms model diffuse inter-reflection which is a very important part of global illumination; however most of these (excluding radiosity) also model specular reflection, which makes them more accurate algorithms to solve the lighting equation and provide a more realistically illuminated scene.

The algorithms used to calculate the distribution of light energy between surfaces of a scene are closely related to heat transfer simulations performed using finite-element methods in engineering design.

In real-time 3D graphics, the diffuse inter-reflection component of global illumination is sometimes approximated by an "ambient" term in the lighting equation, which is also called "ambient lighting" or "ambient color" in 3D software packages. Though this method of approximation (also known as a "cheat" because it's not really a global illumination method) is easy to perform computationally, when used alone it does not provide an adequately realistic effect. Ambient lighting is known to "flatten" shadows in 3D scenes, making the overall visual effect more bland. However, used properly, ambient lighting can be an efficient way to make up for a lack of processing power.

Procedure

For the simulation of global illumination are used in 3D programs, more and more specialized algorithms that can effectively simulate the global illumination. These are, for example, path tracing or photon mapping, under certain conditions, including radiosity. These are always methods to try to solve the rendering equation.

The following approaches can be distinguished here:

- Inversion: $L = (1 - T)^{-1} L^e$
    - is not applied in practice
- Expansion: $L = \sum_{i=0}^{\infty} T^i L^e$
    - bi-directional approach: Photon Mapping + Distributed ray tracing, Bi-directional path tracing, Metropolis light transport
- Iteration: $L_n t l_e + = L^{(n-1)}$
    - Radiosity

In Light path notation global lighting the paths of the type L (D | S) corresponds * E.

Image-Based Lighting

Another way to simulate real global illumination, is the use of High dynamic range images (HDRIs), also known as environment maps, which encircle the scene, and they illuminate. This process is known as image-based lighting.

Graftal

A **graftal** or L-system is a formal grammar used in computer graphics to recursively define branching tree and plant shapes in a compact format. The shape is defined by a string of symbols constructed by a graftal grammar. A graftal grammar consists of an alphabet of symbols that can be used in the strings, a set of production rules which translate each symbol into a non-empty string of symbols, and an axiom from which to begin construction.

Example

- axiom:
  - 0
- rules:
  - 1 → 11
  - 0 → 1[0]0
  - [→ [
  -] →]

The graftal is built by recursively feeding the axiom through the production rules. Each character of the input string is checked against the rule list to determine which character or string to replace it with in the output string. In this example, a '1' in the input string becomes '11' in the output string, while '[' remains the same. Applying this to the axiom of '0', we get:

axiom:	0
1st recursion:	1[0]0
2nd recursion:	11[1[0]0]1[0]0
3rd recursion:	1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0

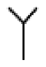
…

We can see that this string quickly grows in size and complexity. This string can be drawn as an image by using turtle graphics, where each symbol is assigned a graphical operation for the turtle to perform. For example, in the sample above, the turtle may be given the following instructions:

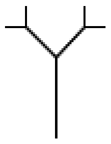
- 0: draw a line segment ending in a leaf
- 1: draw a line segment
- [: push position and angle, turn left 45 degrees
-]: pop position and angle, turn right 45 degrees

The push and pop refer to a LIFO stack (more technical grammar would have separate symbols for "push position" and "turn left"). When the turtle interpretation encounters a '[', the current position and angle are saved, and are then restored when the interpretation encounters a ']'. If multiple values have been "pushed," then a "pop" restores the most recently saved values. Applying the graphical rules listed above to our earlier recursion, we get:

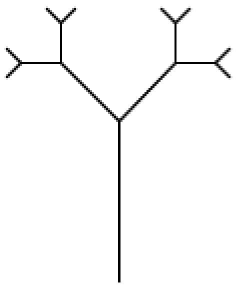
Axiom



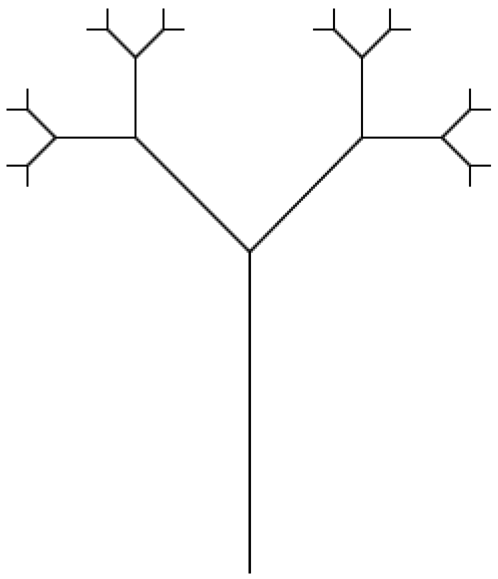
First recursion



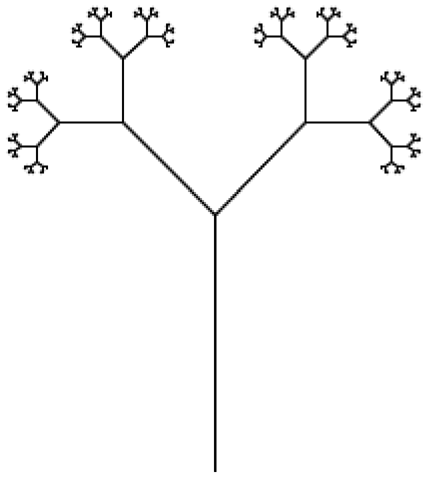
Second recursion



Third recursion



Fourth recursion



Seventh recursion, scaled down ten times

Variations

A number of elaborations on this basic graftal technique have been developed which can be used in conjunction with each other. Among these are stochastic, context sensitive, and parametric grammars.

Stochastic grammars

The grammar model we have discussed thus far has been deterministic—that is, given any symbol in the grammar's alphabet, there has been exactly one production rule, which is always chosen, and always performs the same conversion. One alternative is to specify more than one production rule for a symbol, giving each a probability of occurring. For example, in the above grammar, we could change the rule for rewriting "0" from:

$0 \rightarrow 1[0]0$

To a probabilistic rule:

$0\ (0.5) \rightarrow 1[0]0$
$0\ (0.5) \rightarrow 0$

Under this production, whenever a "0" is encountered during string rewriting, there would be a 50% chance it would behave as previously described, and a 50% chance it would not change during production. When a stochastic grammar is used in an evolutionary context, it is advisable to incorporate a random seed into the genotype, so that the stochastic properties of the image remain constant between generations.

Context sensitive grammars

A context sensitive production rule looks not only at the symbol it is modifying, but the symbols on the string appearing before and after it. For instance, the production rule:

$b < a > c \rightarrow aa$

transforms "a" to "aa", but only If the a occurs between a "b" and a "c" in the input string:

…bac…

As with stochastic productions, there are multiple productions to handle symbols in different contexts. If no production rule can be found for a given context, the identity production is assumed, and the symbol does not change on transformation. If context-sensitive and context-free productions both exist within the same grammar, the context-sensitive production is assumed to take precedence when it is applicable.

Parametric grammars

In a parametric grammar, each symbol in the alphabet has a parameter list associated with it. A symbol coupled with its parameter list is called a module, and a string in a parametric grammar is a series of modules. An example string might be:

a(0,1)[b(0,0)]a(1,2)

The parameters can be used by the functions drawing the graftal, and also by the production rules. The production rules can use the parameters in two ways: first, in a conditional statement determining whether the rule will apply, and second, the production rule can modify the actual parameters. For example, look at:

$$a(x,y) : x = 0 \rightarrow a(1, y+1)b(2,3)$$

The module a(x,y) undergoes transformation under this production rule if the conditional x=0 is met. For example, a(0,2) would undergo transformation, and a(1,2) would not.

In the transformation portion of the production rule, the parameters as well as entire modules can be affected. In the above example, the module b(x,y) is added to the string, with initial parameters (2,3). Also, the parameters of the already existing module are transformed. Under the above production rule,

$$a(0,2)$$

Becomes

$$a(1,3)b(2,3)$$

as the "x" parameter of a(x,y) is explicitly transformed to a "1" and the "y" parameter of a is incremented by one.

Parametric grammars allow line lengths and branching angles to be determined by the graftal grammar, rather than the turtle interpretation methods. Also, if age is given as a parameter for a module, rules can change depending on the age of a plant segment, allowing animations of the entire life-cycle of the tree to be created.

Chapter 4

Hidden Surface Determination and Marching Cubes

Hidden surface determination

In 3D computer graphics, **hidden surface determination** (also known as **hidden surface removal** (**HSR**), **occlusion culling** (**OC**) or **visible surface determination** (**VSD**) is the process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint. A hidden surface determination algorithm is a solution to the visibility problem, which was one of the first major problems in the field of 3D computer graphics. The process of hidden surface determination is sometimes called **hiding**, and such an algorithm is sometimes called a **hider**. The analogue for line rendering is hidden line removal. Hidden surface determination is necessary to render an image correctly, so that one cannot look through walls in virtual reality.

There are many techniques for hidden surface determination. They are fundamentally an exercise in sorting, and usually vary in the order in which the sort is performed and how the problem is subdivided. Sorting large quantities of graphics primitives is usually done by divide and conquer.

Hidden surface removal algorithms

Considering the rendering pipeline, the projection , the clipping, and the rasterization steps are handled differently by the following algorithms:

- Z-buffering During rasterization the depth/Z value of each pixel (or *sample* in the case of anti-aliasing, but without loss of generality we use the term *pixel*) is checked against an existing depth value. If the current pixel is behind the pixel in the Z-buffer, the pixel is rejected, otherwise it is shaded and its depth value replaces the one in the Z-buffer. Z-buffering supports dynamic scenes easily, and is currently implemented efficiently in graphics hardware. This is the current standard. The cost of using Z-buffering is that it uses up to 4 bytes per pixel, and that the rasterization algorithm needs to check each rasterized sample against the z-buffer. The z-buffer can also suffer from artifacts due to precision errors (also known as z-fighting), although this is far less common now that commodity hardware supports 24-bit and higher precision buffers.

- Coverage buffers (C-Buffer) and Surface buffer (S-Buffer): faster than z-buffers and commonly used in games in the Quake I era. Instead of storing the Z value per pixel, they store list of already displayed segments per line of the screen. New polygons are then cut against already displayed segments that would hide them. A S-Buffer can display unsorted polygons, while a C-Buffer require polygons to be displayed from the nearest to the furthest. C-buffer having no overdrawn, they will make the rendering a bit faster. They were commonly used with BSP trees which would give the polygon sorting.
- Sorted Active Edge List: used in Quake 1, this was storing a list of the edges of already displayed polygons. Polygons are displayed from the nearest to the furthest. New polygons are clipped against already displayed polygons' edges, creating new polygons to display then storing the additional edges. It's much harder to implement than S/C/Z buffers, but it will scale much better with the increase in resolution.
- Painter's algorithm sorts polygons by their barycenter and draws them back to front. This produces few artifacts when applied to scenes with polygons of similar size forming smooth meshes and backface culling turned on. The cost here is the sorting step and the fact that visual artifacts can occur.
- Binary space partitioning (BSP) divides a scene along planes corresponding to polygon boundaries. The subdivision is constructed in such a way as to provide an unambiguous depth ordering from any point in the scene when the BSP tree is traversed. The disadvantage here is that the BSP tree is created with an expensive pre-process. This means that it is less suitable for scenes consisting of dynamic geometry. The advantage is that the data is pre-sorted and error free, ready for the previously mentioned algorithms. Note that the BSP is not a solution to HSR, only a help.
- Ray tracing attempts to model the path of light rays to a viewpoint by tracing rays from the viewpoint into the scene. Although not a hidden surface removal algorithm as such, it implicitly solves the hidden surface removal problem by finding the nearest surface along each view-ray. Effectively this is equivalent to sorting all the geometry on a per pixel basis.
- The Warnock algorithm divides the screen into smaller areas and sorts triangles within these. If there is ambiguity (i.e., polygons overlap in depth extent within these areas), then further subdivision occurs. At the limit, subdivision may occur down to the pixel level.

Culling and VSD

A related area to VSD is *culling*, which usually happens before VSD in a rendering pipeline. Primitives or batches of primitives can be rejected in their entirety, which *usually* reduces the load on a well-designed system.

The advantage of culling early on the pipeline is that entire objects that are invisible do not have to be fetched, transformed, rasterized or shaded. Here are some types of culling algorithms:

Viewing frustum culling

The viewing frustum is a geometric representation of the volume visible to the virtual camera. Naturally, objects outside this volume will not be visible in the final image, so they are discarded. Often, objects lie on the boundary of the viewing frustum. These objects are cut into pieces along this boundary in a process called clipping, and the pieces that lie outside the frustum are discarded as there is no place to draw them.

Backface culling

Since meshes are hollow shells, not solid objects, the back side of some faces, or polygons, in the mesh will never face the camera. Typically, there is no reason to draw such faces. This is responsible for the effect often seen in computer and video games in which, if the camera happens to be inside a mesh, rather than seeing the "inside" surfaces of the mesh, it mostly disappears. (Some game engines continue to render any forward-facing or double-sided polygons, resulting in stray shapes appearing without the rest of the penetrated mesh.)

Contribution culling

Often, objects are so far away that they do not contribute significantly to the final image. These objects are thrown away if their screen projection is too small.

Occlusion culling

Objects that are entirely behind other opaque objects may be culled. This is a very popular mechanism to speed up the rendering of large scenes that have a moderate to high depth complexity. There are several types of occlusion culling approaches:

- Potentially visible set or *PVS* rendering, divides a scene into regions and pre-computes visibility for them. These visibility sets are then indexed at run-time to obtain high quality visibility sets (accounting for complex occluder interactions) quickly.
- Portal rendering divides a scene into cells/sectors (rooms) and portals (doors), and computes which sectors are visible by clipping them against portals.

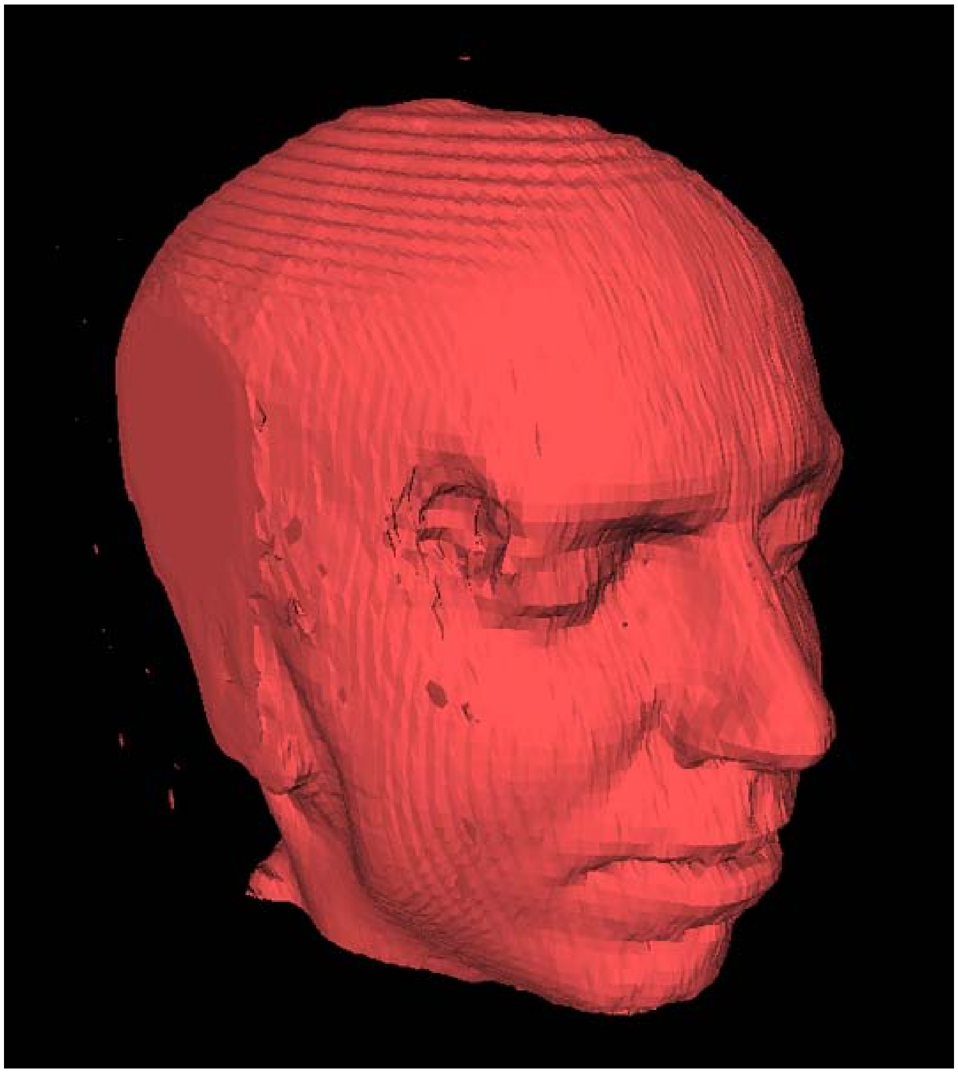
Hansong Zhang's dissertation "Effective Occlusion Culling for the Interactive Display of Arbitrary Models" describes an occlusion culling approach.

Divide and conquer

A popular theme in the VSD literature is divide and conquer. The Warnock algorithm pioneered dividing the screen. Beam tracing is a ray-tracing approach which divides the visible volumes into beams. Various screen-space subdivision approaches reducing the number of primitives considered per region, e.g. tiling, or screen-space BSP clipping. Tiling may be used as a preprocess to other techniques. ZBuffer hardware may typically include a coarse 'hi-Z' against which primitives can be early-rejected without rasterization, this is a form of occlusion culling.

Bounding volume hierarchies (BVHs) are often used to subdivide the scene's space (examples are the BSP tree, the octree and the kd-tree). This allows visibility determination to be performed hierarchically: effectively, if a node in the tree is considered to be *invisible* then all of its child nodes are also invisible, and no further processing is necessary (they can all be rejected by the renderer). If a node is considered *visible*, then each of its children need to be evaluated. This traversal is effectively a tree walk where invisibility/occlusion or reaching a leaf node determines whether to stop or whether to recurse respectively.

Marching cubes



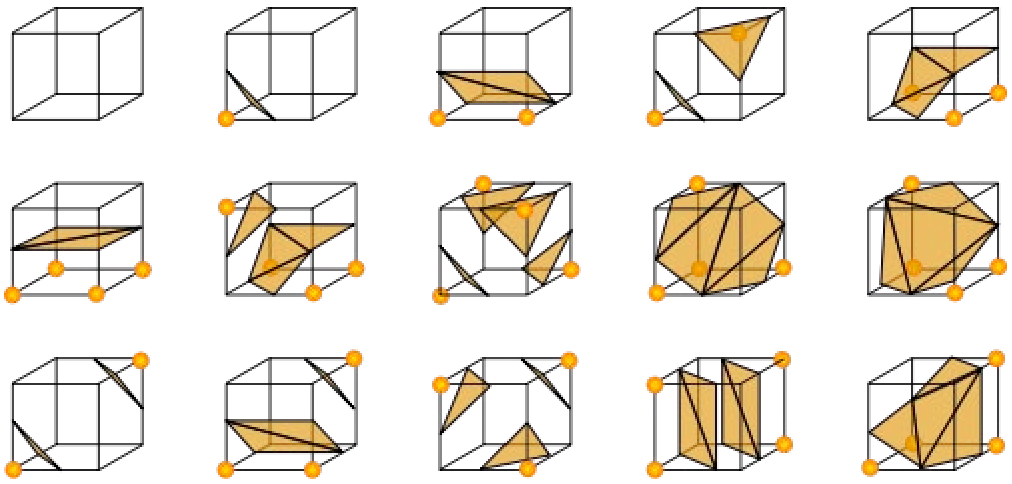
Head and cerebral structures (hidden) extracted from 150 MRI slices using marching-cubes (about 150,000 triangles)

Marching cubes is a computer graphics algorithm, published in the 1987 SIGGRAPH proceedings by Lorensen and Cline, for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field (sometimes called voxels). An equivalent two-dimensional method is called the marching squares algorithm.

The algorithm proceeds through the scalar field, taking eight neighbor locations at a time (thus forming an imaginary cube), then determining the polygon(s) needed to represent the part of the isosurface that passes through this cube. The individual polygons are then fused into the desired surface.

This is done by creating an index to a precalculated array of 256 possible polygon configurations ($2^8 = 256$) within the cube, by treating each of the 8 scalar values as a bit in an 8-bit integer. If the scalar's value is higher than the iso-value (i.e., it is inside the surface) then the appropriate bit is set to one, while if it is lower (outside), it is set to zero. The final value after all 8 scalars are checked, is the actual index to the polygon configuration array.

Finally each vertex of the generated polygons is placed on the appropriate position along the cube's edge by linearly interpolating the two scalar values that are connected by that edge.



15 unique cube configurations

The precalculated array of 256 cube configurations can be obtained by reflections and symmetrical rotations of 14 unique cases. However, using just 14 unique base cases produces an isosurface that is not water-tight. The base cases that can produce a water-tight isosurface are shown in the Marching Cubes survey article external link below (and in other papers).

The gradient of the scalar field at each grid point is also the normal vector of a hypothetical isosurface passing from that point. Therefore, we may interpolate these normals along the edges of each cube to find the normals of the generated vertices which are essential for shading the resulting mesh with some illumination model.

The applications of this algorithm are mainly concerned with medical visualizations such as CT and MRI scan data images, and special effects or 3-D modelling with what is usually called metaballs or other metasurfaces.

Patent issues

The Marching Cubes algorithm is claimed by anti-software patent advocates as a prime example in the graphics field of the woes of patenting software. An implementation was patented despite being a relatively obvious solution to the surface-generation problem, they claim. Another similar algorithm was developed, called Marching Tetrahedrons, in order to circumvent the patent as well as solve a minor ambiguity problem of marching cubes with some cube configurations. This patent expired in 2005, and it is now legal for the graphics community to use it without royalties since more than 17 years have passed from its issue date (December 1, 1987).

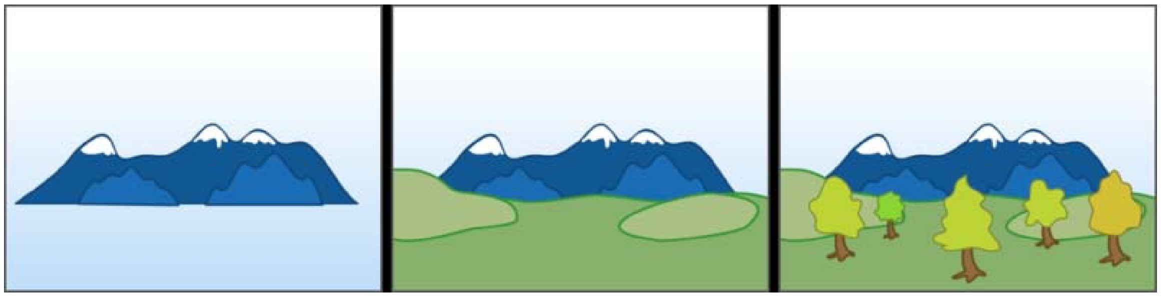
Chapter 5

Painter's Algorithm and Phong Shading

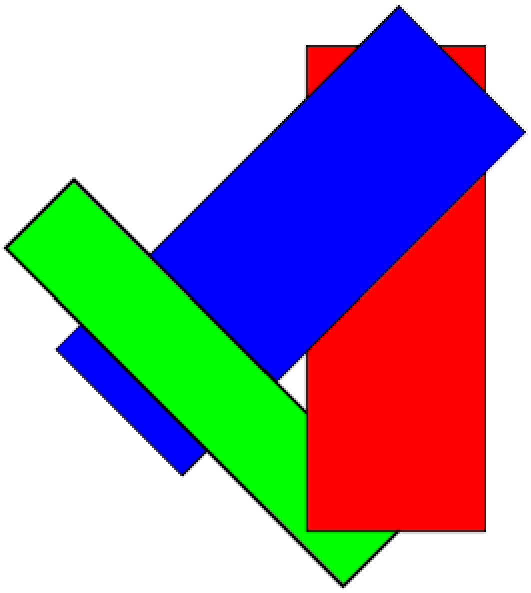
Painter's algorithm

The **painter's algorithm**, also known as a **priority fill**, is one of the simplest solutions to the visibility problem in 3D computer graphics. When projecting a 3D scene onto a 2D plane, it is necessary at some point to decide which polygons are visible, and which are hidden.

The name "painter's algorithm" refers to the technique employed by many painters of painting distant parts of a scene before parts which are nearer thereby covering some areas of distant parts. The painter's algorithm sorts all the polygons in a scene by their depth and then paints them in this order, farthest to closest. It will paint over the parts that are normally not visible — thus solving the visibility problem — at the cost of having painted redundant areas of distant objects.



The distant mountains are painted first, followed by the closer meadows; finally, the closest objects in this scene, the trees, are painted.



Overlapping polygons can cause the algorithm to fail

The algorithm can fail in some cases, including cyclic overlap or piercing polygons. In the case of cyclic overlap, as shown in the figure to the right, Polygons A, B, and C overlap each other in such a way that it is impossible to determine which polygon is above the others. In this case, the offending polygons must be cut to allow sorting. Newell's algorithm, proposed in 1972, provides a method for cutting such polygons. Numerous methods have also been proposed in the field of computational geometry.

The case of piercing polygons arises when one polygon intersects another. As with cyclic overlap, this problem may be resolved by cutting the offending polygons.

In basic implementations, the painter's algorithm can be inefficient. It forces the system to render each point on every polygon in the visible set, even if that polygon is occluded in the finished scene. This means that, for detailed scenes, the painter's algorithm can overly tax the computer hardware.

A **reverse painter's algorithm** is sometimes used, in which objects nearest to the viewer are painted first — with the rule that paint must never be applied to parts of the image that are already painted. In a computer graphic system, this can be very efficient, since it is not necessary to calculate the colors (using lighting, texturing and such) for parts of the more distant scene that are hidden by nearby objects. However, the reverse algorithm suffers from many of the same problems as the standard version.

These and other flaws with the algorithm led to the development of Z-buffer techniques, which can be viewed as a development of the painter's algorithm, by resolving depth conflicts on a pixel-by-pixel basis, reducing the need for a depth-based rendering order. Even in such systems, a variant of the painter's algorithm is sometimes employed. As Z-buffer implementations generally rely on fixed-precision depth-buffer registers implemented in hardware, there is scope for visibility problems due to rounding error. These are overlaps or gaps at joins between polygons. To avoid this, some graphics engine implementations "overrender", drawing the affected edges of both polygons in the order given by painter's algorithm. This means that some pixels are actually drawn twice (as in the full painters algorithm) but this happens on only small parts of the image and has a negligible performance effect.

Phong shading

Phong shading refers to a set of techniques in 3D computer graphics. Phong shading includes a model for the reflection of light from surfaces and a compatible method of estimating pixel colors by interpolating surface normals across rasterized polygons.

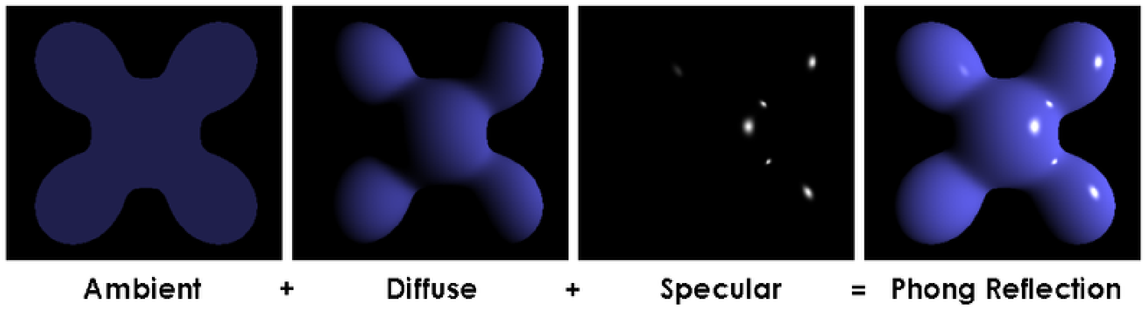
The model of reflection may also be referred to as the **Phong reflection model**, **Phong illumination** or **Phong lighting**. It may be called Phong shading in the context of pixel shaders or other places where a lighting calculation can be referred to as "shading". The interpolation method may also be called **Phong interpolation**, which is usually referred to by "per-pixel lighting". Typically it is called "shading" when contrasted with other interpolation methods such as Gouraud shading or flat shading. The Phong reflection model may be used in conjunction with any of these interpolation methods.

History

These methods were developed by Bui Tuong Phong at the University of Utah, who published them in his 1973 Ph.D. dissertation. Phong's shading methods were considered radical at the time of their introduction, but have evolved into a baseline shading method for many rendering applications. Phong's methods have proven popular due to their generally parsimonious use of CPU time per rendered pixel.

Phong reflection model

Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Bui Tuong Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The reflection model also includes an *ambient* term to account for the small amount of light that is scattered about the entire scene.



Visual illustration of the Phong equation: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting a small part of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction).

For each light source in the scene, we define the components i_s and i_d as the intensities (often as RGB values) of the specular and diffuse components of the light sources respectively. A single term i_a controls the ambient lighting; it is sometimes computed as a sum of contributions from all light sources.

For each *material* in the scene, we define:

> k_s: specular reflection constant, the ratio of reflection of the specular term of incoming light
> k_d: diffuse reflection constant, the ratio of reflection of the diffuse term of incoming light (Lambertian reflectance)
> k_a: ambient reflection constant, the ratio of reflection of the ambient term present in all points in the scene rendered
> α: is a *shininess* constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.

We further define lights as the set of all light sources, L as the direction vector from the point on the surface toward each light source, N as the normal at this point on the surface, R as the direction that a perfectly reflected ray of light would take from this point on the surface, and V as the direction pointing towards the viewer (such as a virtual camera).

Then the *Phong reflection model* provides an equation for computing the shading value of each surface point I_p:

$$I_p = k_a i_a + \sum_{\mathrm{m} \in \mathrm{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^{\alpha} i_s).$$

The diffuse term is not affected by the viewer direction (V). The specular term is large only when the viewer direction (V) is aligned with the reflection direction R. Their alignment is measured by the α power of the cosine of the angle between them. The cosine of the angle between the normalized vectors *R* and *V* is equal to their dot product. When α is large, in the case of a nearly mirror-like reflection, the specular highlight will be small, because any viewpoint not aligned with the reflection will have a cosine less than one which rapidly approaches zero when raised to a high power.

When we have color representations as RGB values, this equation will typically be calculated separately for R, G and B intensities.

Although the above formulation is the common way of presenting the Phong model, a particular term in the sum should only be included if it is positive, i.e. the equation is formally incorrect.

Inverse Phong reflection model

The Phong shading reflection model is an approximation of shading of objects in real life. This means that the Phong equation can relate the shading seen in a photograph with the surface normals of the visible object. Inverse refers to the wish to estimate the surface normals given a rendered image, natural or computer-made.

The Phong reflection model contains many parameters, such as the surface diffuse reflection parameter (albedo) which may vary within the object. Thus the normals of an object in a photograph can only be determined, by introducing additive information such as the number of lights, light directions and reflection parameters.

For example we have a cylindrical object for instance a finger and like to calculate the normal $N = [N_x,N_z]$ on a line on the object. We assume only one light, no specular reflection, and uniform known (approximated) reflection parameters. We can then simplify the Phong equation to:

$$I_p(x) = C_a + C_d(L(x) \cdot N(x))$$

With C_a a constant equal to the ambient light and C_d a constant equal to the diffusion reflection. We can re-write the equation to:

$$(I_p(x) - C_a)/C_d = L(x) \cdot N(x)$$

Which can be rewritten for a line through the cylindrical object as:

$$(I_p - C_a) / C_d = L_xN_x + L_zN_z$$

For instance if the light direction is 45 degrees above the object $L = [0.71,0.71]$ we get two equations with two unknowns.

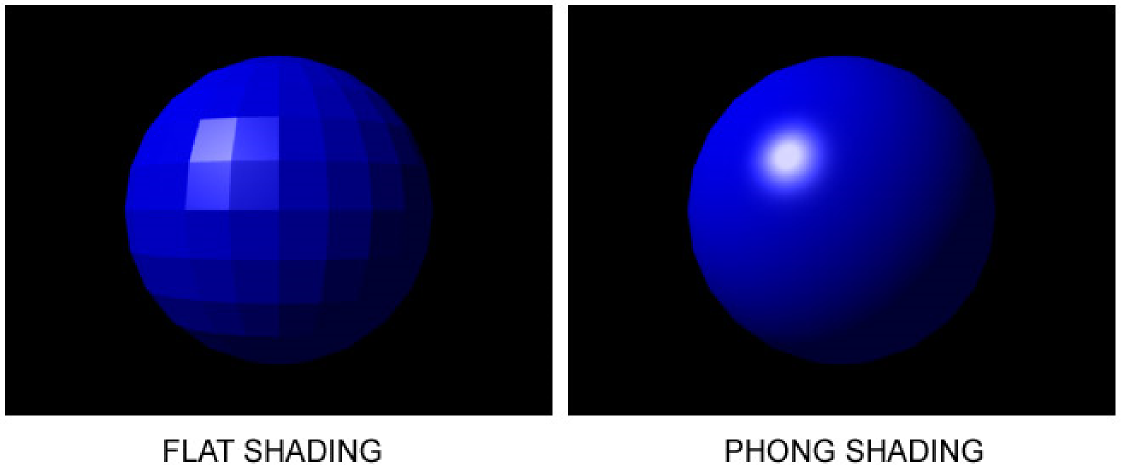
$$(I_p - C_a) / C_d = 0.71N_x + 0.71N_z$$

$$1 = \sqrt{(N_x^2 + N_z^2)}$$

Because of the powers of two in the equation there are two possible solutions for the normal direction. Thus some prior information of the geometry is needed to define the correct normal direction. The normals are directly related to angles of inclination of the line on the object surface. Thus the normals allow the calculation of the relative surface heights of the line on the object using a line integral, if we assume a continuous surface.

If the object is not cylindrical, we have three unknown normal values $N = [N_x, N_y, N_z]$. Then the two equations still allow the normal to rotate around the view vector, thus additional constraints are needed from prior geometric information. For instance in face recognition those geometric constraints can be obtained using principal component analysis (PCA) on a database of depth-maps of faces, allowing only surface normals solutions which are found in a normal population .

Phong interpolation



Phong shading interpolation example

Phong shading improves upon Gouraud shading and provides a better approximation of the shading of a smooth surface. Phong shading assumes a smoothly varying surface normal vector. The Phong interpolation method works better than Gouraud shading when applied to a reflection model that has small specular highlights such as the Phong reflection model.

The most serious problem with Gouraud shading occurs when specular highlights are found in the middle of a large polygon. Since these specular highlights are absent from the polygon's vertices and Gouraud shading interpolates based on the vertex colors, the specular highlight will be missing from the polygon's interior. This problem is fixed by Phong shading.

Unlike Gouraud shading, which interpolates colors across polygons, in Phong shading we linearly interpolate a normal vector across the surface of the polygon from the polygon's vertex normals. The surface normal is interpolated and normalized at each pixel and then used in the Phong reflection model to obtain the final pixel color. Phong shading is more computationally expensive than Gouraud shading since the reflection model must be computed at each pixel instead of at each vertex.

In some modern hardware, variants of this algorithm are implemented using pixel or fragment shaders. This can be accomplished by coding normal vectors as secondary colors for each polygon, have the rasterizer use Gouraud shading to interpolate them and interpret them appropriately in the pixel or fragment shader to calculate the light for each pixel based on this normal information.

Chapter 6

Ramer–Douglas–Peucker Algorithm and Ray Tracing (Graphics)

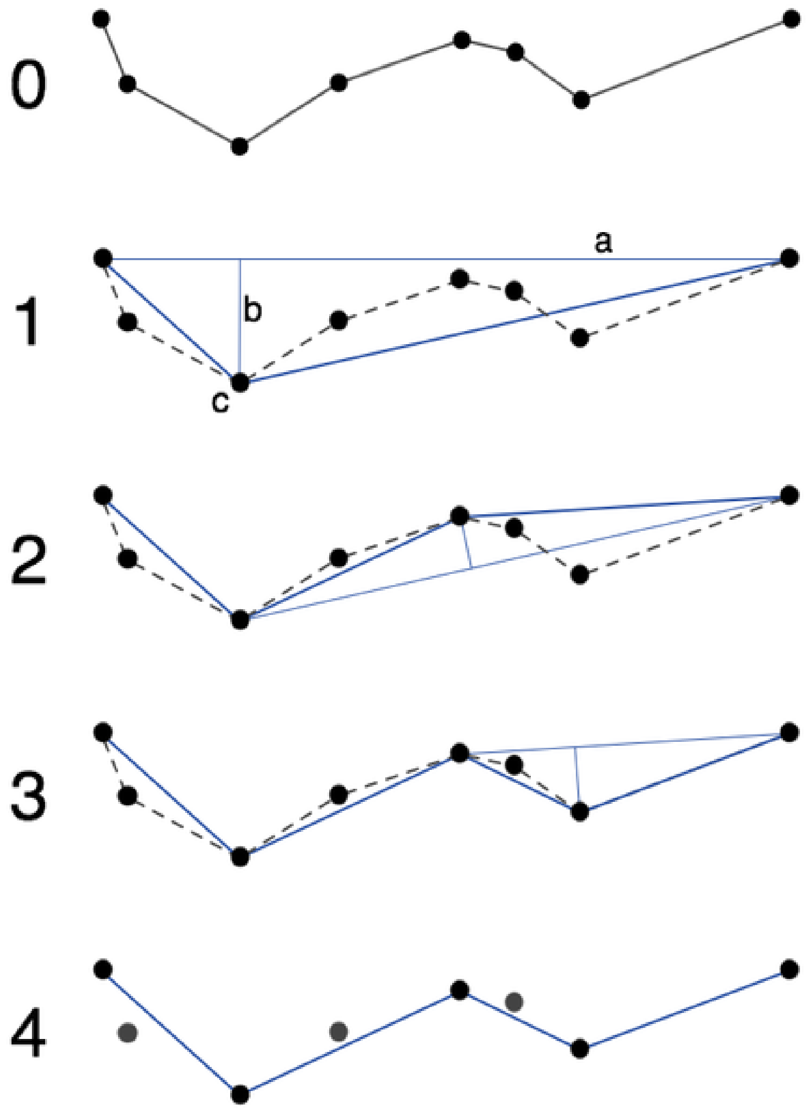
Ramer–Douglas–Peucker algorithm

The **Douglas–Peucker algorithm** is an algorithm for reducing the number of points in a curve that is approximated by a series of points. The initial form of the algorithm was independently suggested in 1972 by Urs Ramer and 1973 by David Douglas and Thomas Peucker. This algorithm is also known under the following names: the Ramer–Douglas–Peucker algorithm, the iterative end-point fit algorithm or the split-and-merge algorithm.

Idea

The purpose of the algorithm is, given a 'curve' composed of line segments, to find a similar curve with fewer points. The algorithm defines 'dissimilar' based on the maximum distance between the original curve and the simplified curve. The simplified curve consists of a subset of the points that defined the original curve.

Algorithm



Smoothing a piecewise linear curve with the Douglas–Peucker algorithm

The starting curve is an ordered set of points or lines and the distance dimension $\varepsilon > 0$. The original (unsmoothed) curve is shown in 0 and the final output curve is shown in blue on row 4.

The algorithm recursively divides the line. Initially it is given all the points between the first and last point. It automatically marks the first and last point to be kept. It then finds the point that is furthest from the line segment with the first and last points as end points (this point is obviously furthest on the curve from the approximating line segment between the end points). If the point is closer than ε to the line segment then any points not currently marked to keep can be discarded without the smoothed curve being worse than ε.

If the point furthest from the line segment is greater than ε from the approximation then that point must be kept. The algorithm recursively calls itself with the first point and the worst point and then with the worst point and the last point (which includes marking the worst point being marked as kept).

When the recursion is completed a new output curve can be generated consisting of all (and only) those points that have been marked as kept.

Pseudocode

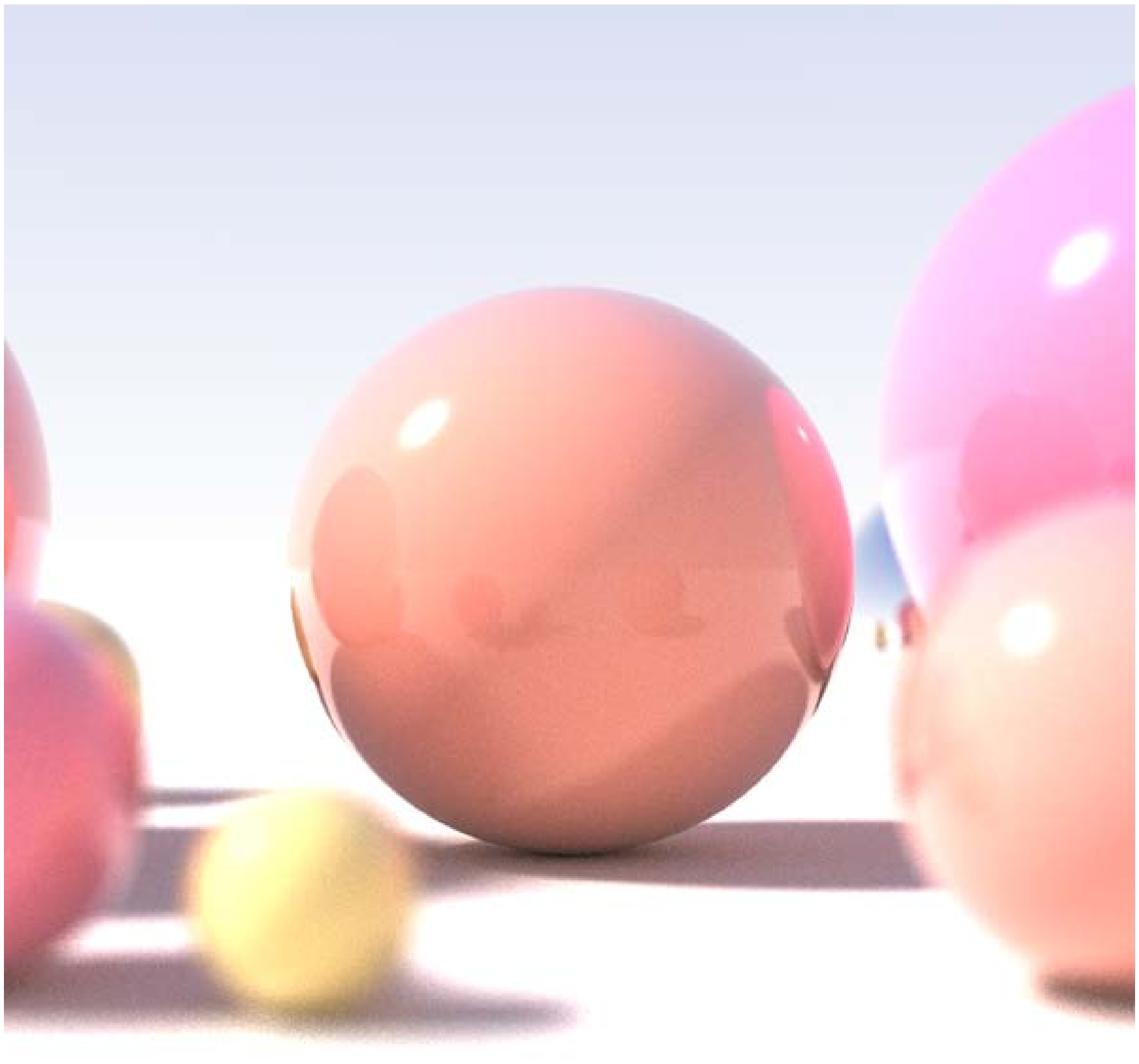
```
function DouglasPeucker(PointList[], epsilon)
 //Find the point with the maximum distance
 dmax = 0
 index = 0
 for i = 2 to (length(PointList) - 1)
  d = OrthogonalDistance(PointList[i], Line(PointList, PointList[end]))
  if d > dmax
   index = i
   dmax = d
  end
 end

 //If max distance is greater than epsilon, recursively simplify
 if dmax >= epsilon
  //Recursive call
  recResults1[] = DouglasPeucker(PointList[1...index], epsilon)
  recResults2[] = DouglasPeucker(PointList[index...end], epsilon)

  // Build the result list
  ResultList[] = {recResults1[1...end-1] recResults2[1...end]}
 else
  ResultList[] = {PointList, PointList[end]}
 end

 //Return the result
 return ResultList[]
end
```

Application

The algorithm is used for the processing of vector graphics and cartographic generalization.

The algorithm is widely used in robotics to perform simplification and denoising of range data acquired by a rotating range scanner, in this field it is called the split-and-merge algorithm and is attributed to Duda and Hart.

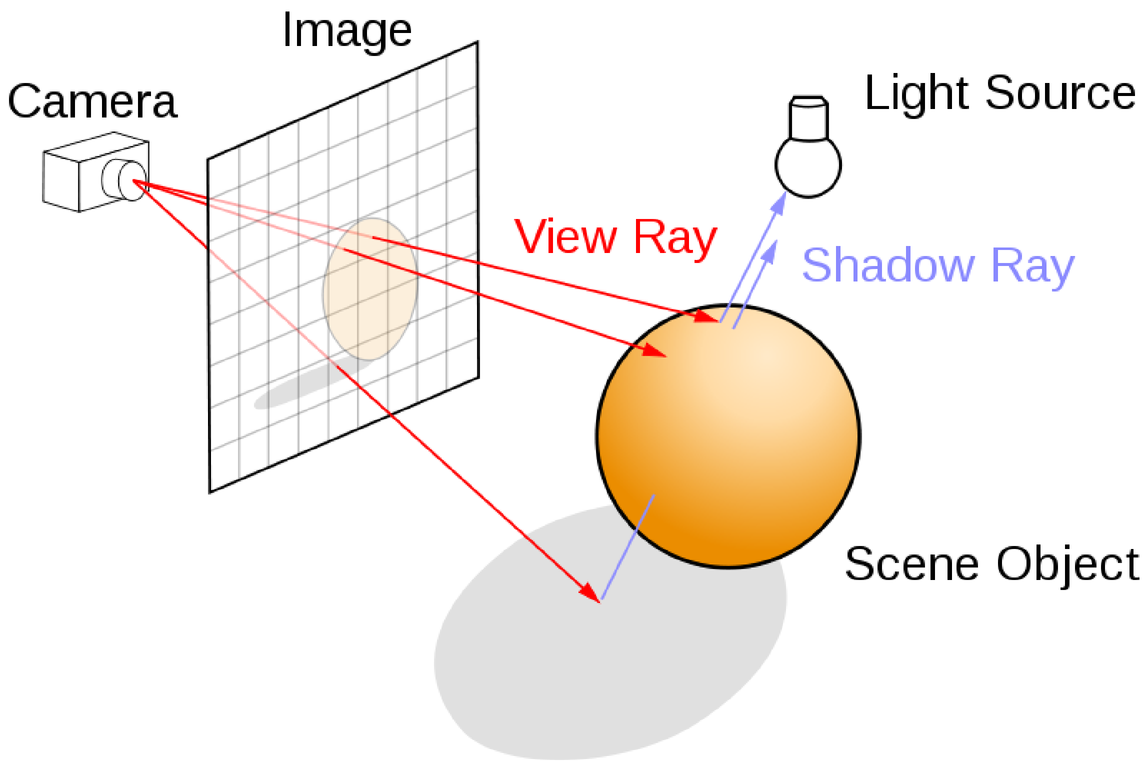
Ray tracing (graphics)



This recursive ray tracing of a sphere demonstrates the effects of shallow depth of field, area light sources, diffuse interreflection, ambient occlusion and fresnel reflection.

In computer graphics, **ray tracing** is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television special effects, and more poorly suited for real-time applications like computer games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and chromatic aberration.

Algorithm overview



The ray tracing algorithm builds an image by extending rays into a scene

Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than either ray casting or scanline rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it.

Scenes in raytracing are described mathematically by a programmer or by a visual artist (typically using intermediary tools). Scenes may also incorporate data from images and models captured by means such as digital photography.

Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene.

It may at first seem counterintuitive or "backwards" to send rays *away* from the camera, rather than *into* it (as actual light does in reality), but doing so is many orders of magnitude more efficient. Since the overwhelming majority of light rays from a given light source do not make it directly into the viewer's eye, a "forward" simulation could potentially waste a tremendous amount of computation on light paths that are never recorded. A computer simulation that starts by casting rays from the light source is called Photon mapping, and it takes much longer than a comparable ray trace.

Therefore, the shortcut taken in raytracing is to presuppose that a given ray intersects the view frame. After either a maximum number of reflections or a ray traveling a certain distance without intersection, the ray ceases to travel and the pixel's value is updated. The light intensity of this pixel is computed using a number of algorithms, which may include the classic rendering algorithm and may also incorporate techniques such as radiosity.

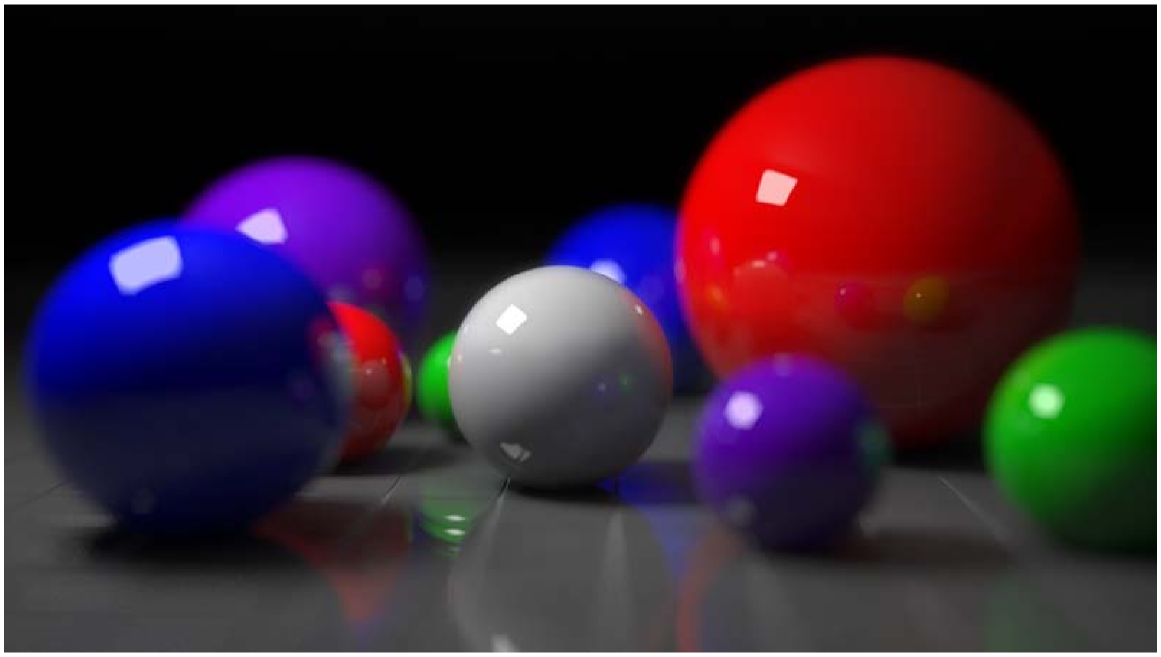
Detailed description of ray tracing computer algorithm and its genesis

What happens in nature

Ray tracing can achieve a very high degree of visual realism

In nature, a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this "ray" as a stream of photons traveling along the same path. In a perfect vacuum this ray will be a straight line (ignoring relativistic effects). In reality, any combination of four things might happen with this light ray: absorption, reflection, refraction and fluorescence. A surface may reflect all or part of the light ray, in one or more directions. It might also absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light. If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the color). Less commonly, a surface may absorb some portion of the light and fluorescently re-emit the light at a longer wavelength colour in a random direction, though this is rare enough that it can be discounted from most rendering applications. Between absorption, reflection, refraction and fluorescence, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract 50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, reflective and fluorescent properties again affect the progress of the incoming rays. Some of these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image.

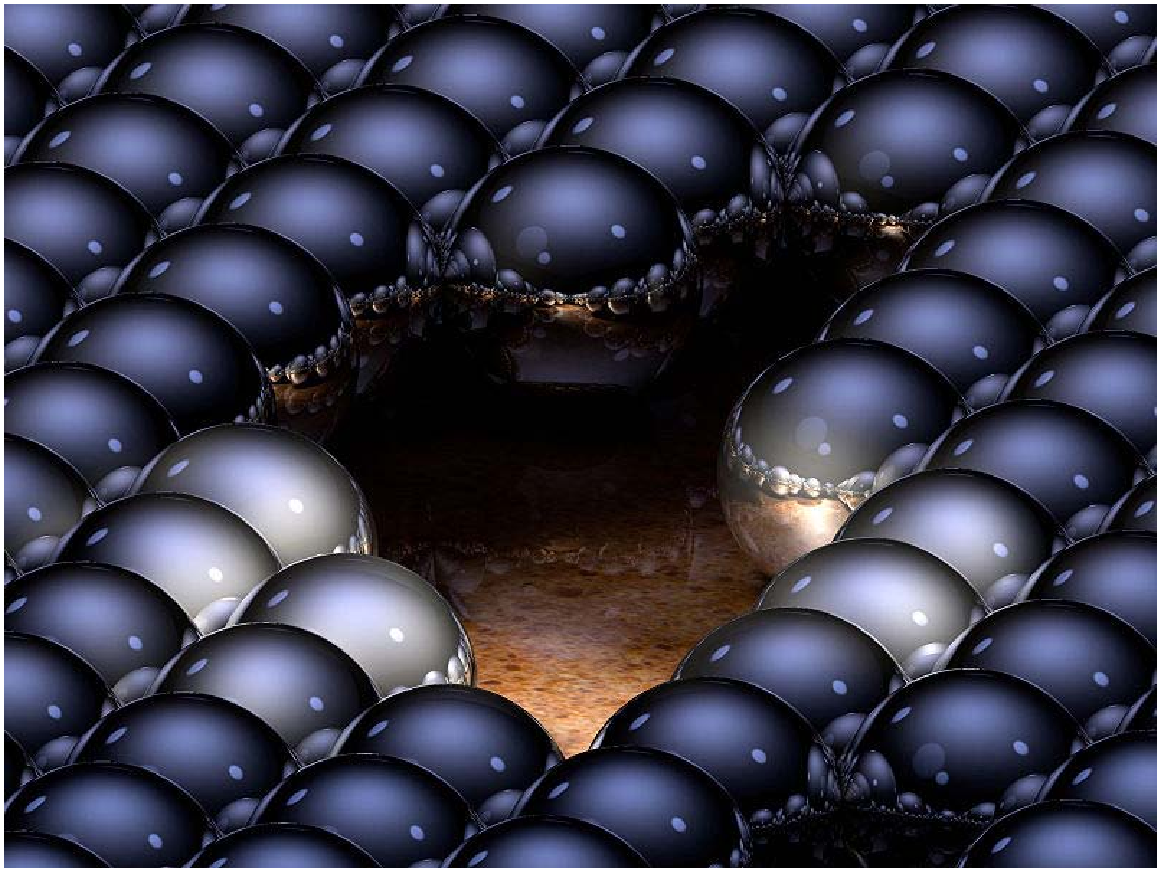
Ray casting algorithm



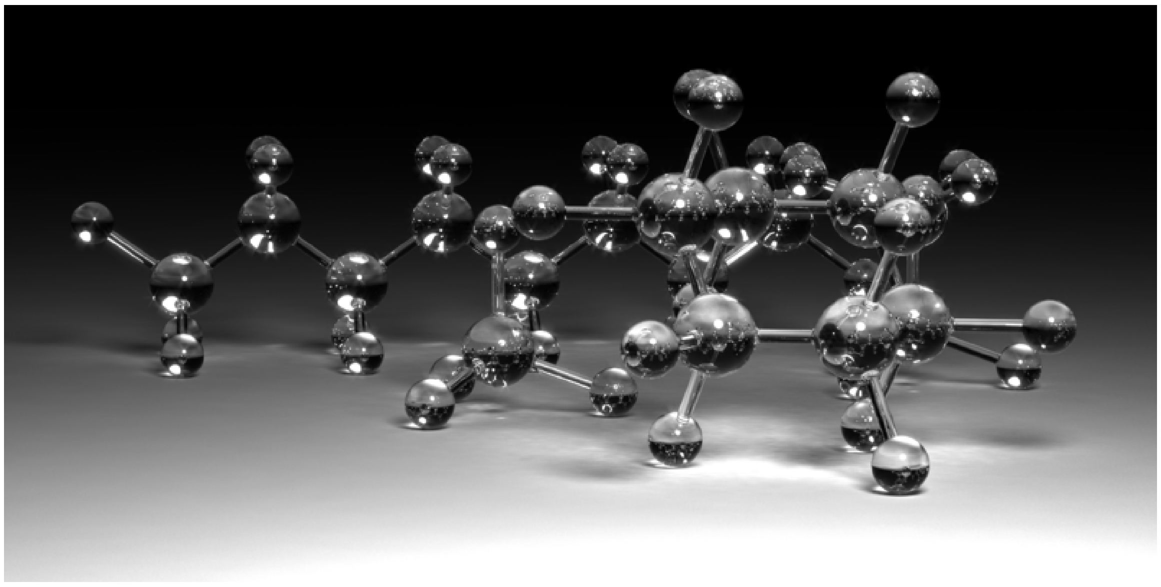
In addition to the high degree of realism, ray tracing can simulate the effects of a camera due to depth of field and aperture shape (in this case a hexagon).

The first ray casting (versus ray tracing) algorithm used for rendering was presented by Arthur Appel in 1968. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray – think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye normally sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow. The shading of the surface is computed using traditional 3D computer graphics shading models. One important advantage ray casting offered over older scanline algorithms is its ability to easily deal with non-planar surfaces and solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using solid modeling techniques and easily rendered.

Ray tracing algorithm



The number of reflections a "ray" can take and how it is affected each time it encounters a surface is all controlled via software settings during ray tracing. Here, each ray was allowed to reflect up to 16 times. Multiple "reflections of reflections" can thus be seen. *Created with Cobalt*



The number of refractions a “ray” can take and how it is affected each time it encounters a surface is all controlled via software settings during ray tracing. Here, each ray was allowed to refract and reflect up to 9 times. Fresnel reflections were used. Also note the caustics. *Created with Vray*

The next important research breakthrough came from Turner Whitted in 1979. Previous algorithms cast rays from the eye into the scene, but the rays were traced no further. Whitted continued the process. When a ray hits a surface, it could generate up to three new types of rays: reflection, refraction, and shadow. A reflected ray continues on in the mirror-reflection direction from a shiny surface. It is then intersected with objects in the scene; the closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. To further avoid tracing all rays in a scene, a shadow ray is used to test if a surface is visible to a light. A ray hits a surface at some point. If the surface at this point faces a light, a ray (to the computer, a line segment) is traced between this intersection point and the light. If any opaque object is found in between the surface and the light, the surface is in shadow and so the light does not contribute to its shade. This new layer of ray calculation added more realism to ray traced images.

Advantages over other rendering methods

Ray tracing's popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. Relatively simple to implement yet yielding impressive visual results, ray tracing often represents a first foray into graphics programming. The computational independence of each ray makes ray tracing amenable to parallelization.

Disadvantages

A serious disadvantage of ray tracing is performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform anti-aliasing and improve image quality where needed. Although it does handle interreflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required. The realism of all rendering methods, then, must be evaluated as an approximation to the equation, and in the case of ray tracing, it is not necessarily the most realistic. Other methods, including photon mapping, are based upon ray tracing for certain parts of the algorithm, yet give far better results.

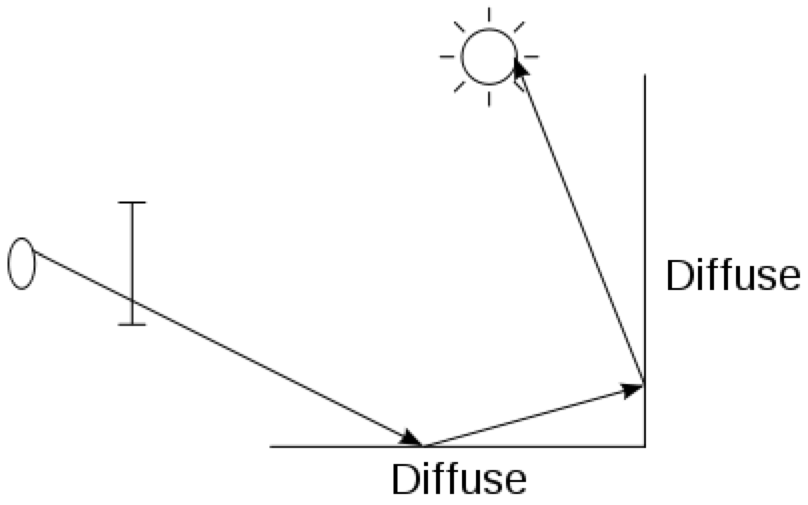
Reversed direction of traversal of scene by the rays

The process of shooting rays from the eye to the light source to render an image is sometimes called *backwards ray tracing*, since it is the opposite direction photons actually travel. However, there is confusion with this terminology. Early ray tracing was always done from the eye, and early researchers such as James Arvo used the term *backwards ray tracing* to mean shooting rays from the lights and gathering the results. Therefore it is clearer to distinguish *eye-based* versus *light-based* ray tracing.

While the direct illumination is generally best sampled using eye-based ray tracing, certain indirect effects can benefit from rays generated from the lights. Caustics are bright patterns caused by the focusing of light off a wide reflective region onto a narrow area of (near-)diffuse surface. An algorithm that casts rays directly from lights onto reflective objects, tracing their paths to the eye, will better sample this phenomenon. This integration of eye-based and light-based rays is often expressed as bidirectional path tracing, in which paths are traced from both the eye and lights, and the paths subsequently joined by a connecting ray after some length.

Photon mapping is another method that uses both light-based and eye-based ray tracing; in an initial pass, energetic photons are traced along rays from the light source so as to compute an estimate of radiant flux as a function of 3-dimensional space (the eponymous photon map itself). In a subsequent pass, rays are traced from the eye into the scene to determine the visible surfaces, and the photon map is used to estimate the illumination at the visible surface points. The advantage of photon mapping versus bidirectional path tracing is the ability to achieve significant reuse of photons, reducing computation, at the cost of statistical bias.

An additional problem occurs when light must pass through a very narrow aperture to illuminate the scene (consider a darkened room, with a door slightly ajar leading to a brightly-lit room), or a scene in which most points do not have direct line-of-sight to any light source (such as with ceiling-directed light fixtures or torchieres). In such cases, only a very small subset of paths will transport energy; Metropolis light transport is a method which begins with a random search of the path space, and when energetic paths are found, reuses this information by exploring the nearby space of rays.



To the right is an image showing a simple example of a path of rays recursively generated from the camera (or eye) to the light source using the above algorithm. A diffuse surface reflects light in all directions.

First, a ray is created at an eyepoint and traced through a pixel and into the scene, where it hits a diffuse surface. From that surface the algorithm recursively generates a reflection ray, which is traced through the scene, where it hits another diffuse surface. Finally, another reflection ray is generated and traced through the scene, where it hits the light source and is absorbed. The color of the pixel now depends on the colors of the first and second diffuse surface and the color of the light emitted from the light source. For example if the light source emitted white light and the two diffuse surfaces were blue, then the resulting color of the pixel is blue.

In real time

The first implementation of a "real-time" ray-tracer was credited at the 2005 SIGGRAPH computer graphics conference as the REMRT/RT tools developed in 1986 by Mike Muuss for the BRL-CAD solid modeling system. Initially published in 1987 at USENIX, the BRL-CAD ray-tracer is the first known implementation of a parallel network distributed ray-tracing system that achieved several frames per second in rendering performance. This performance was attained by means of the highly-optimized yet platform independent LIBRT ray-tracing engine in BRL-CAD and by using solid implicit CSG geometry on several shared memory parallel machines over a commodity network. BRL-CAD's ray-tracer, including REMRT/RT tools, continue to be available and developed today as Open source software.

Since then, there have been considerable efforts and research towards implementing ray tracing in real time speeds for a variety of purposes on stand-alone desktop configurations. These purposes include interactive 3D graphics applications such as demoscene productions, computer and video games, and image rendering. Some real-time software 3D engines based on ray tracing have been developed by hobbyist demo programmers since the late 1990s.

The OpenRT project includes a highly-optimized software core for ray tracing along with an OpenGL-like API in order to offer an alternative to the current rasterisation based approach for interactive 3D graphics. Ray tracing hardware, such as the experimental Ray Processing Unit developed at the Saarland University, has been designed to accelerate some of the computationally intensive operations of ray tracing. On March 16, 2007, the University of Saarland revealed an implementation of a high-performance ray tracing engine that allowed computer games to be rendered via ray tracing without intensive resource usage.

On June 12, 2008 Intel demonstrated a special version of Enemy Territory: Quake Wars, titled Quake Wars: Ray Traced, using ray tracing for rendering, running in basic HD (720p) resolution. ETQW operated at 14-29 frames per second. The demonstration ran on a 16-core (4 socket, 4 core) Tigerton system running at 2.93 GHz.

At SIGGRAPH 2009, Nvidia announced OptiX, an API for real-time ray tracing on Nvidia GPUs. The API exposes seven programmable entry points within the ray tracing pipeline, allowing for custom cameras, ray-primitive intersections, shaders, shadowing, etc.

Example

As a demonstration of the principles involved in raytracing, let us consider how one would find the intersection between a ray and a sphere. In vector notation, the equation of a sphere with center $\mathbf{c}$and radius $\mathbf{r}$is

$$\left\|\mathbf{x}-\mathbf{c}\right\|^2=r^2.$$

Any point on a ray starting from point $\mathbf{s}$with direction $\mathbf{d}$(here $\mathbf{d}$is a unit vector) can be written as

$$\mathbf{x}=\mathbf{s}+t\mathbf{d},$$

where *t* is its distance between $\mathbf{x}$and $\mathbf{s}$. In our problem, we know $\mathbf{c}$, $\mathbf{r}$, $\mathbf{s}$(e.g. the position of a light source) and $\mathbf{d}$, and we need to find *t*. Therefore, we substitute for $\mathbf{x}$:

$$\left\|\mathbf{s}+t\mathbf{d}-\mathbf{c}\right\|^2=r^2.$$

Let $\mathbf{v}\ \overset{\mathrm{def}}{=}\ \mathbf{s}-\mathbf{c}$for simplicity; then

$$\|\mathbf{v} + t\mathbf{d}\|^2 = r^2$$
$$\mathbf{v}^2 + t^2\mathbf{d}^2 + 2\mathbf{v} \cdot t\mathbf{d} = r^2$$
$$(\mathbf{d}^2)t^2 + (2\mathbf{v} \cdot \mathbf{d})t + (\mathbf{v}^2 - r^2) = 0.$$

Knowing that d is a unit vector allows us this minor simplification:

$$t^2 + (2\mathbf{v} \cdot \mathbf{d})t + (\mathbf{v}^2 - r^2) = 0.$$

This quadratic equation has solutions

$$t = \frac{-(2\mathbf{v} \cdot \mathbf{d}) \pm \sqrt{(2\mathbf{v} \cdot \mathbf{d})^2 - 4(\mathbf{v}^2 - r^2)}}{2} = -(\mathbf{v} \cdot \mathbf{d}) \pm \sqrt{(\mathbf{v} \cdot \mathbf{d})^2 - (\mathbf{v}^2 - r^2)}.$$

The two values of *t* found by solving this equation are the two ones such that $\mathbf{s} + t\mathbf{d}$are the points where the ray intersects the sphere.

If one (or both) of them are negative, then the intersections do not lie on the ray but in the opposite half-line (i.e. the one starting from $\mathbf{s}$with opposite direction).

If the quantity under the square root (the discriminant) is negative, then the ray does not intersect the sphere.

Let us suppose now that there is at least a positive solution, and let *t* be the minimal one. In addition, let us suppose that the sphere is the nearest object on our scene intersecting our ray, and that it is made of a reflective material. We need to find in which direction the light ray is reflected. The laws of reflection state that the angle of reflection is equal and opposite to the angle of incidence between the incident ray and the normal to the sphere.

The normal to the sphere is simply

$$\mathbf{n} = \frac{\mathbf{y} - \mathbf{c}}{\|\mathbf{y} - \mathbf{c}\|},$$

where $\mathbf{y} = \mathbf{s} + t\mathbf{d}$is the intersection point found before. The reflection direction can be found by a reflection of $\mathbf{d}$with respect to $\mathbf{n}$, that is

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{n} \cdot \mathbf{d})\mathbf{n}.$$

Thus the reflected ray has equation

$$\mathbf{x} = \mathbf{y} + u\mathbf{r}.$$

Now we only need to compute the intersection of the latter ray with our field of view, to get the pixel which our reflected light ray will hit. Lastly, this pixel is set to an appropriate color, taking into account how the color of the original light source and the one of the sphere are combined by the reflection.

This is merely the math behind the line–sphere intersection and the subsequent determination of the colour of the pixel being calculated. There is, of course, far more to the general process of raytracing, but this demonstrates an example of the algorithms used.

Chapter 7

Scanline Rendering and Slerp

Scanline rendering

Scanline rendering is an algorithm for visible surface determination, in 3D computer graphics, that works on a row-by-row basis rather than a polygon-by-polygon or pixel-by-pixel basis. All of the polygons to be rendered are first sorted by the top y coordinate at which they first appear, then each row or scan line of the image is computed using the intersection of a scan line with the polygons on the front of the sorted list, while the sorted list is updated to discard no-longer-visible polygons as the active scan line is advanced down the picture.

The asset of this method is that it is not necessary to translate the coordinates of all vertices from the main memory into the working memory—only vertices defining edges that intersect the current scan line need to be in active memory, and each vertex is read in only once. The main memory is often very slow compared to the link between the central processing unit and cache memory, and thus avoiding re-accessing vertices in main memory can provide a substantial speedup.

This kind of algorithm can be easily integrated with the Phong reflection model, the Z-buffer algorithm, and many other graphics techniques.

Algorithm

The usual method starts with edges of projected polygons inserted into buckets, one per scanline; the rasterizer maintains an active edge table(*AET*). Entries maintain sort links, X coordinates, gradients, and references to the polygons they bound. To rasterize the next scanline, the edges no longer relevant are removed; new edges from the current scanlines' Y-bucket are added, inserted sorted by X coordinate. The active edge table entries have X and other parameter information incremented. Active edge table entries are maintained in an X-sorted list by bubble-sort, effecting a change when 2 edges cross. After updating edges, the active edge table is traversed in X order to emit only the visible spans, maintaining a Z-sorted active Span table, inserting and deleting the surfaces when edges are crossed.

Variants

A hybrid between this and Z-buffering does away with the active edge table sorting, and instead rasterizes one scanline at a time into a Z-buffer, maintaining active polygon spans from one scanline to the next.

In another variant, an ID buffer is rasterized in an intermediate step, allowing deferred shading of the resulting visible pixels.

History

The first publication of the scanline rendering technique was probably by Wylie, Romney, Evans, and Erdahl in 1967.

Other early developments of the scanline rendering method were by Bouknight in 1969, and Newell, Newell, and Sancha in 1972. Much of the early work on these methods was done in Ivan Sutherland's graphics group at the University of Utah, and at the Evans & Sutherland company in Salt Lake City, Utah.

Use in realtime rendering

The early Evans & Sutherland ESIG line of image-generators (IGs) employed the technique in hardware 'on the fly', to generate images one raster-line at a time without a framebuffer, saving the need for then costly memory. Later variants used a hybrid approach.

The Nintendo DS is the latest hardware to render 3D scenes in this manner, with the option of caching the rasterized images into VRAM.

The sprite hardware prevalent in 1980s games machines can be considered a simple 2D form of scanline rendering.

The technique was used in the first Quake engine for software rendering of environments (but moving objects were Z-buffered over the top). Static scenery used BSP-derived sorting for priority. It proved better than Z-buffer/painter's type algorithms at handling scenes of high depth complexity with costly pixel operations (i.e. perspective-correct texture mapping without hardware assist). This use preceded the widespread adoption of Z-buffer-based GPUs now common in PCs.

Sony experimented with software scanline renderers on a second Cell processor during the development of the PlayStation 3, before settling on a conventional CPU/GPU arrangement.

Similar techniques

A similar principle is employed in tiled rendering (most famously the PowerVR 3D chip); that is, primitives are sorted into screen space, then rendered in fast on-chip memory, one tile at a time. The Dreamcast provided a mode for rasterizing one row of tiles at a time for direct raster scanout, saving the need for a complete framebuffer, somewhat in the spirit of hardware scanline rendering.

Some software rasterizers use 'span buffering' (or 'coverage buffering'), in which a list of sorted, clipped spans are stored in scanline buckets. Primitives would be successively added to this datastructure, before rasterizing only the visible pixels in a final stage.

Comparison with Z-buffer algorithm

The main advantage of scanline rendering over Z-buffering is that visible pixels are only ever processed once—a benefit for the case of high resolution or expensive shading computations.

In modern Z-buffer systems, similar benefits can be gained through rough front-to-back sorting (approaching the 'reverse painters algorithm'), early Z-reject (in conjunction with hierarchical Z), and less common deferred rendering techniques possible on programmable GPUs.

Scanline techniques working on the raster have the drawback that overload is not handled gracefully.

The technique is not considered to scale well as the number of primitives increases. This is because of the size of the intermediate datastructures required during rendering—which can exceed the size of a Z-buffer for a complex scene.

Consequently, in contemporary interactive graphics applications, the Z-buffer has become ubiquitous. The Z-buffer allows larger volumes of primitives to be traversed linearly, in parallel, in a manner friendly to modern hardware. Transformed coordinates, attribute gradients, etc., need never leave the graphics chip; only the visible pixels and depth values are stored.

Slerp

In computer graphics, **Slerp** is shorthand for **spherical linear interpolation**, introduced by Ken Shoemake in the context of quaternion interpolation for the purpose of animating 3D rotation. It refers to constant speed motion along a unit radius great circle arc, given the ends and an interpolation parameter between 0 and 1.

Geometric Slerp

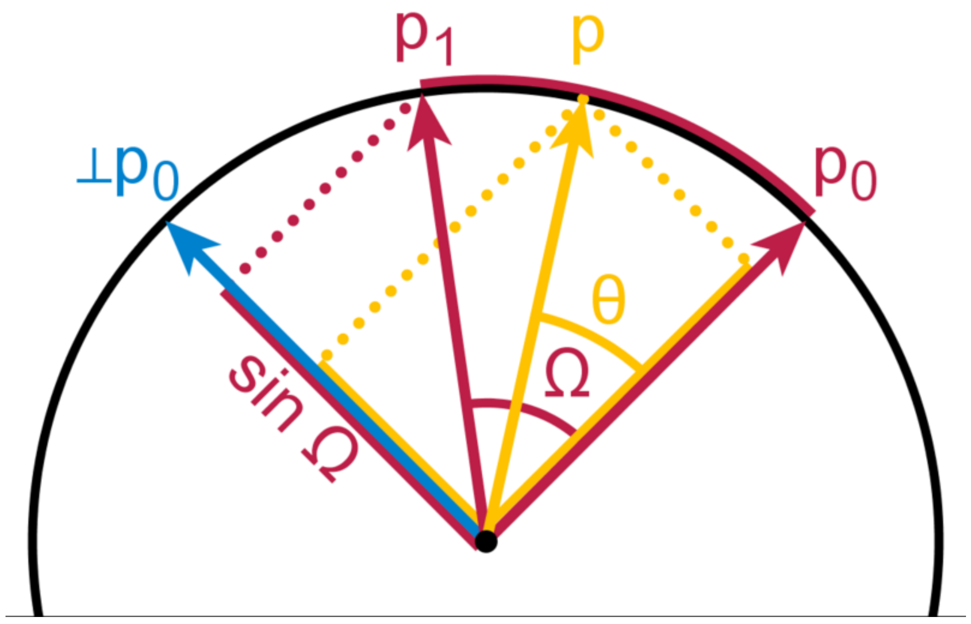
Slerp has a geometric formula independent of quaternions, and independent of the dimension of the space in which the arc is embedded. This formula, a symmetric weighted sum credited to Glenn Davis, is based on the fact that any point on the curve must be a linear combination of the ends. Let p_0 and p_1 be the first and last points of the arc, and let t be the parameter, $0 \leq t \leq 1$. Compute Ω as the angle subtended by the arc, so that $\cos \Omega = p_0 \cdot p_1$, the n-dimensional dot product of the unit vectors from the origin to the ends. The geometric formula is then

$$\mathrm{Slerp}(p_0, p_1; t) = \frac{\sin\,[(1-t)\Omega]}{\sin \Omega} p_0 + \frac{\sin[t\Omega]}{\sin \Omega} p_1.$$

The symmetry can be seen in the fact that $\mathrm{Slerp}(p_0,p_1;t) = \mathrm{Slerp}(p_1,p_0;1-t)$. In the limit $\Omega \rightarrow 0$, this formula reduces to the corresponding symmetric formula for linear interpolation,

$$\mathrm{lerp}(p_0, p_1; t) = (1-t)p_0 + tp_1.$$

A Slerp path is, in fact, the spherical geometry equivalent of a path along a line segment in the plane; a great circle is a spherical geodesic.



Oblique vector rectifies to Slerp factor

More familiar than the general Slerp formula is the case when the end vectors are perpendicular, in which case the formula is $p_0 \cos \theta + p_1 \sin \theta$. Letting $\theta = t\,\pi/2$, and applying the trigonometric identity cos θ = sin (π/2−θ), this becomes the Slerp formula. The factor of 1/sin Ω in the general formula is a normalization, since a vector p_1 at an angle of Ω to p_0 projects onto the perpendicular $\perp p_0$ with a length of only sin Ω.

Some special cases of Slerp admit more efficient calculation. When a circular arc is to be drawn into a raster image, the preferred method is some variation of Bresenham's circle algorithm. Evaluation at the special parameter values 0 and 1 trivially yields p_0 and p_1, respectively; and bisection, evaluation at ½, simplifies to $(p_0+p_1)/2$, normalized. Another special case, common in animation, is evaluation with fixed ends and equal parametric steps. If p_{k-1} and p_k are two consecutive values, and if c is twice their dot product (constant for all steps), then the next value, p_{k+1}, is the reflection $p_{k+1} = c'\mathbf{p_k} - \mathbf{p_{k-1}}$.

Quaternion Slerp

When Slerp is applied to unit quaternions, the quaternion path maps to a path through 3D rotations in a standard way. The effect is a rotation with uniform angular velocity around a fixed rotation axis. When the initial end point is the identity quaternion, Slerp gives a segment of a one-parameter subgroup of both the Lie group of 3D rotations, SO(3), and its universal covering group of unit quaternions, S^3. Slerp gives a straightest and shortest path between its quaternion end points, and maps to a rotation through an angle of 2Ω. However, because the covering is double (q and $-q$ map to the same rotation), the rotation path may turn either the "short way" (less than 180°) or the "long way" (more than 180°). Long paths can be prevented by negating one end if the dot product, cos Ω, is negative, thus ensuring that −90° ≤ Ω ≤ 90°.

Slerp also has expressions in terms of quaternion algebra, all using exponentiation. For a quaternion q and a real number t, the exponential q^t is defined in terms of the exponential e^q, itself given by the power series familiar from complex analysis,

$$e^q = 1 + q + \frac{q^2}{2} + \frac{q^3}{6} + \cdots + \frac{q^n}{n!} + \cdots.$$

Writing a unit quaternion q in versor form, cos Ω + **v** sin Ω, with **v** a unit 3-vector, and noting that the quaternion square $\mathbf{v}^2$ equals −1 (implying a quaternion version of Euler's formula), we have $e^{\mathrm{v}\Omega} = q$, and $q^t = \cos t\Omega + \mathbf{v} \sin t\Omega$. The identification of interest is $q = q_1 q_0^{-1}$, so that the real part of q is cos Ω, the same as the geometric dot product used above. Here are four equivalent quaternion expressions for Slerp.

$$\begin{aligned}\mathrm{Slerp}(q_0, q_1; t) &= q_0(q_0^{-1}q_1)^t \\ &= q_1(q_1^{-1}q_0)^{1-t} \\ &= (q_0 q_1^{-1})^{1-t} q_1 \\ &= (q_1 q_0^{-1})^t q_0\end{aligned}$$

The derivative of Slerp(q_0,q_1;t) with respect to t, assuming the ends are fixed, is log($q_1 q_0^{-1}$) times the function value, where the quaternion natural logarithm in this case yields half the 3D angular velocity vector. The initial tangent vector is parallel transported to each tangent along the curve; thus the curve is, indeed, a geodesic.

In the tangent space at any point on a quaternion Slerp curve, the inverse of the exponential map transforms the curve into a line segment. Slerp curves not extending through a point fail to transform into lines in that point's tangent space.

Quaternion Slerps are commonly used to construct smooth animation curves by mimicking affine constructions like the de Casteljau algorithm for Bézier curves. Since the sphere is not an affine space, familiar properties of affine constructions may fail, though the constructed curves may otherwise be entirely satisfactory. For example, the de Casteljau algorithm may be used to split a curve in affine space; this does not work on a sphere.

The two-valued Slerp can be extended to interpolate among many unit quaternions, but the extension loses the fixed execution-time of the Slerp algorithm.

Chapter 8

Cohen–Sutherland and Line Clipping

Cohen–Sutherland

In computer graphics, the **Cohen–Sutherland** algorithm is a line clipping algorithm. The algorithm divides a 2D space into 9 regions, of which only the middle part (viewport) is visible.

In 1967, flight simulation work by Danny Cohen lead to the development of the Cohen–Sutherland computer graphics two and three dimensional line clipping algorithms, created with Ivan Sutherland..

The algorithm

The algorithm includes, excludes or partially includes the line based on where:

- Both endpoints are in the viewport region (bitwise OR of endpoints == 0): trivial accept.
- Both endpoints are on the same non-visible region (bitwise AND of endpoints != 0): trivial reject.
- Both endpoints are in different regions: In case of this non trivial situation the algorithm finds one of the two points that are outside the viewport region (there will be at least one point outside). The intersection of the outpoint and extended viewport border is then calculated (i.e. with the parametric equation for the line) and this new point replaces the outpoint. The algorithm repeats until a trivial accept or reject occurs.

The numbers in the figure below are called outcodes. An outcode is computed for each of the two points in the line. The first bit is set to 1 if the point is above the viewport. The bits in the outcode represent: Top, Bottom, Right, Left. For example the outcode 1010 represents a point that is top-right of the viewport. Note that the outcodes for endpoints **must** be recalculated on each iteration after the clipping occurs.

1001	1000	1010
0001	0000	0010

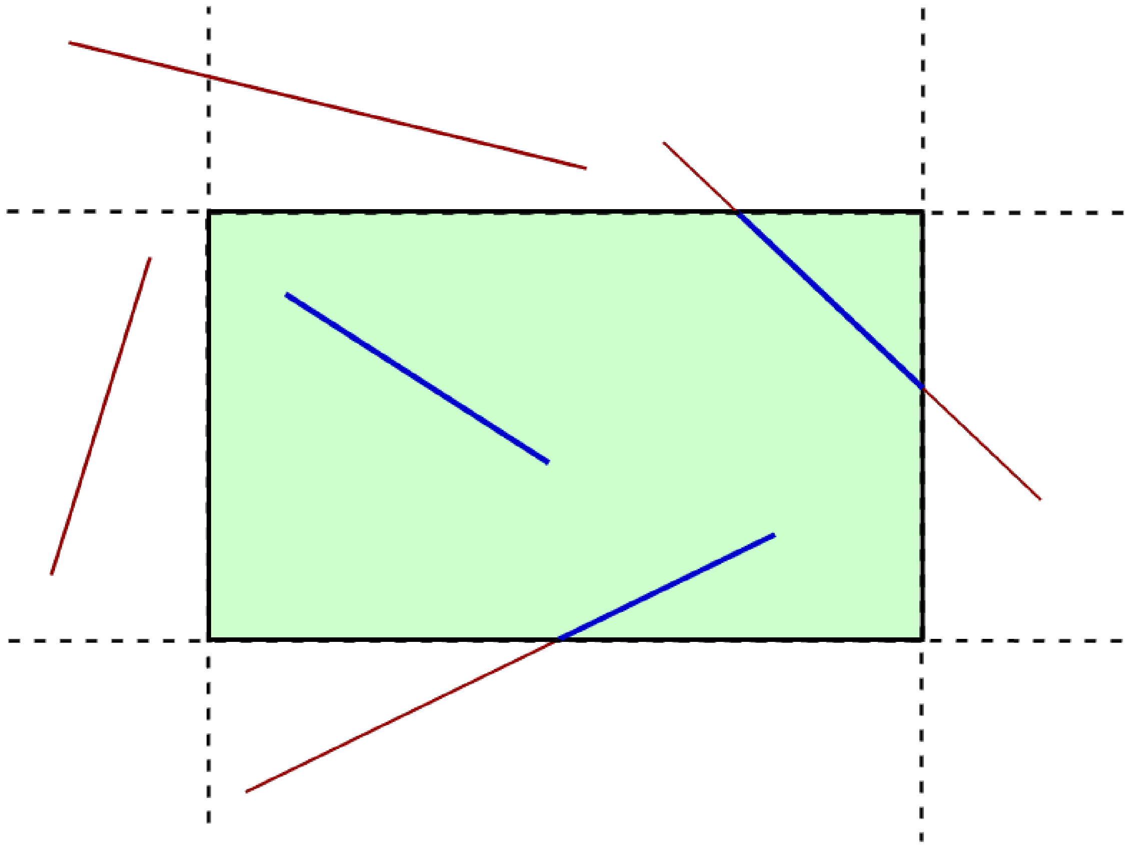
0101	0100	0110

Example C/C++ implementation

```
#define OutCode int

const int INSIDE = 0; // 0000
const int LEFT = 1;   // 0001
const int RIGHT = 2;  // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8;    // 1000

// Compute the bit code for a point (x, y) using the clip rectangle
// bounded diagonally by (xmin, ymin), and (xmax, ymax)
OutCode ComputeOutCode(double x, double y)
{
        OutCode code;

        code = INSIDE;          // initialised as being inside of clip
window

        if (x < xmin)           // to the left of clip window
                code |= LEFT;
        else if (x > xmax)      // to the right of clip window
                code |= RIGHT;
        if (y < ymin)           // below the clip window
                code |= BOTTOM;
        else if (y > ymax)      // above the clip window
                code |= TOP;

        return code;
}

// Cohen-Sutherland clipping algorithm clips a line from
// P0 = (x0, y0) to P1 = (x1, y1) against a rectangle with
// diagonal from (xmin, ymin) to (xmax, ymax).
void CohenSutherlandLineClipAndDraw(double x0, double y0, double x1,
double y1)
{
        // compute outcodes for P0, P1, and whatever point lies outside
the clip rectangle
        OutCode outcode0 = ComputeOutCode(x0, y0);
        OutCode outcode1 = ComputeOutCode(x1, y1);
        bool accept = false;

        while (true) {
                if (!(outcode0 | outcode1)) {     //logical or is 0.
Trivially accept and get out of loop
                        accept = true;
                        break;
                } else if (outcode0 & outcode1) { //logical and is not
0. Trivially reject and get out of loop
                        break;
                 } else {
                        // failed both tests, so calculate the line
segment to clip
```

```
                        // from an outside point to an intersection with
clip edge
                        double x, y;

                        // At least one endpoint is outside the clip
rectangle; pick it.
                        OutCode outcodeOut = outcode0? outcode0 :
outcode1;

                        // Now find the intersection point;
                        // use formulas y = y0 + slope * (x - x0), x = x0
+ (1 / slope) * (y - y0)
                        if (outcodeOut & TOP) {           // point is
above the clip rectangle
                                x = x0 + (x1 - x0) * (ymax - y0) / (y1 -
y0);
                                y = ymax;
                        } else if (outcodeOut & BOTTOM) { // point is
below the clip rectangle
                                x = x0 + (x1 - x0) * (ymin - y0) / (y1 -
y0);
                                y = ymin;
                        } else if (outcodeOut & RIGHT) {  // point is to
the right of clip rectangle
                                y = y0 + (y1 - y0) * (xmax - x0) / (x1 -
x0);
                                x = xmax;
                        } else if (outcodeOut & LEFT) {   // point is to
the left of clip rectangle
                                y = y0 + (y1 - y0) * (xmin - x0) / (x1 -
x0);
                                x = xmin;
                        }
                        // Now we move outside point to intersection
point to clip
                        // and get ready for next pass.
                        if (outcodeOut == outcode0) {
                                x0 = x;
                                y0 = y;
                                outcode0 = ComputeOutCode(x0, y0);
                        } else {
                                x1 = x;
                                y1 = y;
                                outcode1 = ComputeOutCode(x1, y1);
                        }
                }
        }
        if (accept) {
                // Following functions are left for implementation by
user based on his platform(OpenGL/graphics.h etc.)
                DrawRectangle(xmin, ymin, xmax, ymax);
                LineSegment(x0, y0, x1, y1);
        }
}
```

Line clipping



In computer graphics, **line clipping** is the process of removing lines or portions of lines outside of an area of interest. Typically, any line or part thereof which is outside of the viewing area is removed.

There are two common algorithms for line clipping: Cohen–Sutherland and Liang–Barsky.

Liang–Barsky

In computer graphics, the **Liang–Barsky** algorithm is a line clipping algorithm. The Liang–Barsky algorithm uses the parametric equation of a line and inequalities describing the range of the clipping box to determine the intersections between the line and the clipping box. With these intersections it knows which portion of the line should be drawn. This algorithm is significantly more efficient than Cohen–Sutherland.

The idea of the Liang-Barsky clipping algorithm is to do as much testing as possible before computing line intersections. Consider first the usual parametric form of a straight line:

$$x = x_0 + u(x_1 - x_0) = x_0 + u\Delta x$$
$$y = y_0 + u(y_1 - y_0) = y_0 + u\Delta y$$

A point is in the clip window, if

$$x_{\min} \le x_0 + u\Delta x \le x_{\max}$$

and

$$y_{\min} \le y_0 + u\Delta y \le y_{\max},$$

which can be expressed as the 4 inequalities

$$up_k \le q_k, \quad k = 1, 2, 3, 4,$$

where

$$p_1 = -\Delta x, q_1 = x_0 - x_{\min} \text{(left)}$$
$$p_2 = \Delta x, q_2 = x_{\max} - x_0 \text{(right)}$$
$$p_3 = -\Delta y, q_3 = y_0 - y_{\min} \text{(bottom)}$$
$$p_4 = \Delta y, q_4 = y_{\max} - y_0 \text{(top)}$$

To compute the final line segment:

1. A line parallel to a clipping window edge has $p_k = 0$ for that boundary.
2. If for that k, $q_k < 0$, the line is completely outside and can be eliminated.
3. When $p_k < 0$ the line proceeds outside to inside the clip window and when $p_k > 0$, the line proceeds inside to outside.
4. For nonzero p_k, $u = \frac{q_k}{p_k}$ gives the intersection point.
5. For each line, calculate u_1 and u_2. For u_1, look at boundaries for which $p_k < 0$ (outside -> in). Take u_1 to be the largest among $\left(0, \frac{q_k}{p_k}\right)$. For u_2, look at boundaries for which $p_k > 0$ (inside -> out). Take u_2 to be the minimum of $\left(1, \frac{q_k}{p_k}\right)$. If $u_1 > u_2$, the line is outside and therefore rejected.

Cyrus–Beck algorithm

The **Cyrus–Beck algorithm** is a line clipping algorithm. It was designed to be more efficient than the Sutherland–Cohen algorithm which uses repetitive clipping . Cyrus–

Beck is a general algorithm and can be used with a convex polygon clipping window unlike Sutherland-Cohen that can be used only on a rectangular clipping area

Here the parametric equation of a line in the view plane is:

$$P(t) = P(0) + t(P_1 - P_0)$$

where $t = 0$ at $P(1)$ $t = o$ at t_0

Now to find intersection point with the clipping window we calculate value of dot product

n(Pt-f)
{if > 0 vector pointed toward interior
if = 0 vector pointed parallel to plane containing *f*
if < 0 vector pointed away from interior

here *n* stands for normal.

By this we select the point of intersection of line and clipping window where (dot product = 0) and hence clip the line

Nicholl–Lee–Nicholl

The Nicholl-Lee-Nicholl(NLN) is a fast clipping algorithim that reduces the chances of multiple clipping of a single line segment as may happen in the Cohen-Sutherland algorithm. Here the area around the clipping window is divided into a number of different areas depending on the position of the initial point of the line to be clipped. This initial point should be in the three predetermined areas, thus the line may have to be translated and/or rotated to bring it into the desired region. The line segment may then be re-translated and/or re-rotated to bring it to the original position.After that, straight line segments are drawn from the line end point, passing through the corners of the clipping window. These areas are then designated as L,LT,LB,TR, depending on the location of the initial point. Then the other end point of the line is checked against these areas. If , say, a line starts in the L area and finishes in the LT area then the algorithim concludes that the line should be clipped at xw(max). Thus the no. of clipping points is reduced to one only compared to other algorithms that may require two or more clipping points.

Fast clipping

This algorithm has similarities with Cohen-Sutherland. The start and end positions are classified by which portion of the 9 area grid they occupy. A large switch statement jumps to a specialized handler for that case. In contrast, Cohen-Sutherland may have to iterate several times to handle the same case.

Chapter 9

Anisotropic Diffusion

In image processing and computer vision, **anisotropic diffusion**, also called **Perona–Malik diffusion**, is a technique aiming at reducing image noise without removing significant parts of the image content, typically edges, lines or other details that are important for the interpretation of the image. Anisotropic diffusion resembles the process that creates a scale-space, where an image generates a parameterized family of successively more and more blurred images based on a diffusion process. Each of the resulting images in this family are given as a convolution between the image and a 2D isotropic Gaussian filter, where the width of the filter increases with the parameter. This diffusion process is a *linear* and *space-invariant* transformation of the original image. Anisotropic diffusion is a generalization of this diffusion process: it produces a family of parameterized images, but each resulting image is a combination between the original image and a filter that depends on the local content of the original image. As a consequence, anisotropic diffusion is a *non-linear* and *space-variant* transformation of the original image.

In its original formulation, presented by Perona and Malik in 1987, the space-variant filter is in fact isotropic but depends on the image content such that it approximates an impulse function close to edges and other structures that should be preserved in the image over the different levels of the resulting scale-space. This formulation was referred to as *anisotropic diffusion* by Perona and Malik even though the locally adapted filter is isotropic, but it has also been referred to as *inhomogeneous and nonlinear diffusion*, or *Perona-Malik diffusion* by other authors. A more general formulation allows the locally adapted filter to be truly anisotropic close to linear structures such as edges or lines: it has an orientation given by the structure such that it is elongated along the structure and narrow across. As a consequence, the resulting images preserve linear structures while at the same time smoothing is made along these structures. Both these cases can be described by a generalization of the usual diffusion equation where the diffusion coefficient, instead of being a constant scalar, is a function of image position and assumes a matrix (or tensor) value.

Although the resulting family of images can be described as a combination between the original image and space-variant filters, the locally adapted filter and its combination with the image do not have to be realized in practice. Anisotropic diffusion is normally implemented by means of an approximation of the generalized diffusion equation: each new image in the family is computed by applying this equation to the previous image.

Consequently, anisotropic diffusion is an iterative process where a relatively simple set of computation are used to compute each successive image in the family and this process is continued until a sufficient degree of smoothing is obtained.

Formal definition

Formally, let $\Omega \subset \mathbb{R}^2$denote a subset of the plane and $I(\cdot, t) : \Omega \rightarrow \mathbb{R}$be a family of gray scale images, then anisotropic diffusion is defined as

$$\frac{\partial I}{\partial t} = \operatorname{div}\left(c(x,y,t)\nabla I\right) = \nabla c \cdot \nabla I + c(x,y,t)\Delta I$$

where Δ denotes the Laplacian, ∇denotes the gradient, $\operatorname{div}(\cdot)$is the divergence operator and *c*(*x*,*y*,*t*) is the diffusion coefficient. *c*(*x*,*y*,*t*) controls the rate of diffusion and is usually chosen as a function of the image gradient so as to preserve edges in the image. Pietro Perona and Jitendra Malik pioneered the idea of anisotropic diffusion in 1990 and proposed two functions for the diffusion coefficient:

$$c\left(||\nabla I||\right) = e^{-(||\nabla I||/K)^2}$$

and

$$c\left(||\nabla I||\right) = \frac{1}{1 + \left(\frac{||\nabla I||}{K}\right)^2}$$

the constant K controls the sensitivity to edges and is usually chosen experimentally or as a function of the noise in the image.

Motivation

Let *M* denote the manifold of smooth images, then the diffusion equations presented above can be interpreted as the gradient descent equations for the minimization of the energy functional $E : M \rightarrow \mathbb{R}$defined by

$$E[I] = \frac{1}{2}\int_{\Omega} g\left(||\nabla I(x)||^2\right)\, dx$$

where $g : \mathbb{R} \rightarrow \mathbb{R}$is a real-valued function which we will see is intimately related to the diffusion coefficient. Then for any compactly supported infinitely differentiable test function *h*, we have

$$\frac{d}{dt}\bigg|_{t=0} E[I+th] = \frac{d}{dt}\Big|_{t=0}\frac{1}{2}\int_{\Omega} g\left(||\nabla(I+th)(x)||^2\right)\,dx$$
$$= \int_{\Omega} g'\left(||\nabla I(x)||^2\right)\nabla I\cdot\nabla h\,dx$$
$$= -\int_{\Omega} \mathrm{div}(g'\left(||\nabla I(x)||^2\right)\nabla I)h\,dx$$

where the last line follow from multidimensional integration by parts. Letting ∇E_Idenote the gradient of E with respect to the $L^2(\Omega,\mathbb{R})$inner product evaluated at I, this gives

$$\nabla E_I = -\mathrm{div}(g'\left(||\nabla I(x)||^2\right)\nabla I)$$

Therefore, the gradient descent equations on the functional *E* are given by

$$\frac{\partial I}{\partial t} = -\nabla E_I = \mathrm{div}(g'\left(||\nabla I(x)||^2\right)\nabla I)$$

Thus by letting *c* = *g*' we obtain the anisotropic diffusion equations.

Applications

Anisotropic diffusion can be used to remove noise from digital images without blurring edges. With a constant diffusion coefficient, the anisotropic diffusion equations reduce to the heat equation which is equivalent to Gaussian blurring. This is ideal for removing noise but also indiscriminately blurs edges too. When the diffusion coefficient is chosen as an edge seeking function, such as in Perona and Malik, the resulting equations encourage diffusion (hence smoothing) within regions and prohibit it across strong edges. Hence the edges can be preserved while removing noise from the image.

Along the same lines as noise removal, anisotropic diffusion can be used in edge detection algorithms. By running the diffusion with an edge seeking diffusion coefficient for a certain number of iterations, the image can be evolved towards a piecewise constant image with the boundaries between the constant components being detected as edges.

Chapter 10

Camera Interface and Canny Edge Detector

Camera interface

The **CAMIF**, also the Camera Interface block is the hardware block that interfaces with different image sensor interfaces and provides a standard output that can be used for subsequent image processing.

A typical **Camera Interface** would support at least a parallel interface although these days many camera interfaces are beginning to support the MIPI CSI interface.

The camera interface's parallel interface consists of the following lines :-

8 to 12 bits parallel data line

```
  These are parallel data lines that carry pixel data. The data
transmitted on these lines change with every Pixel Clock (PCLK).
```

Horizontal Sync (HSYNC)

```
  This is a special signal that goes from the camera sensor or ISP to
the camera interface. An HSYNC indicates that one line of the frame is
transmitted.
```

Vertical Sync (VSYNC)

```
  This signal is transmitted after the entire frame is transferred.
This signal is often a way to indicate that one entire frame is
transmitted.
```

Pixel Clock (PCLK)

```
  This is the pixel clock and it would change on every pixel.
```

NOTE: The above lines are all treated as input lines to the Camera Interface hardware.

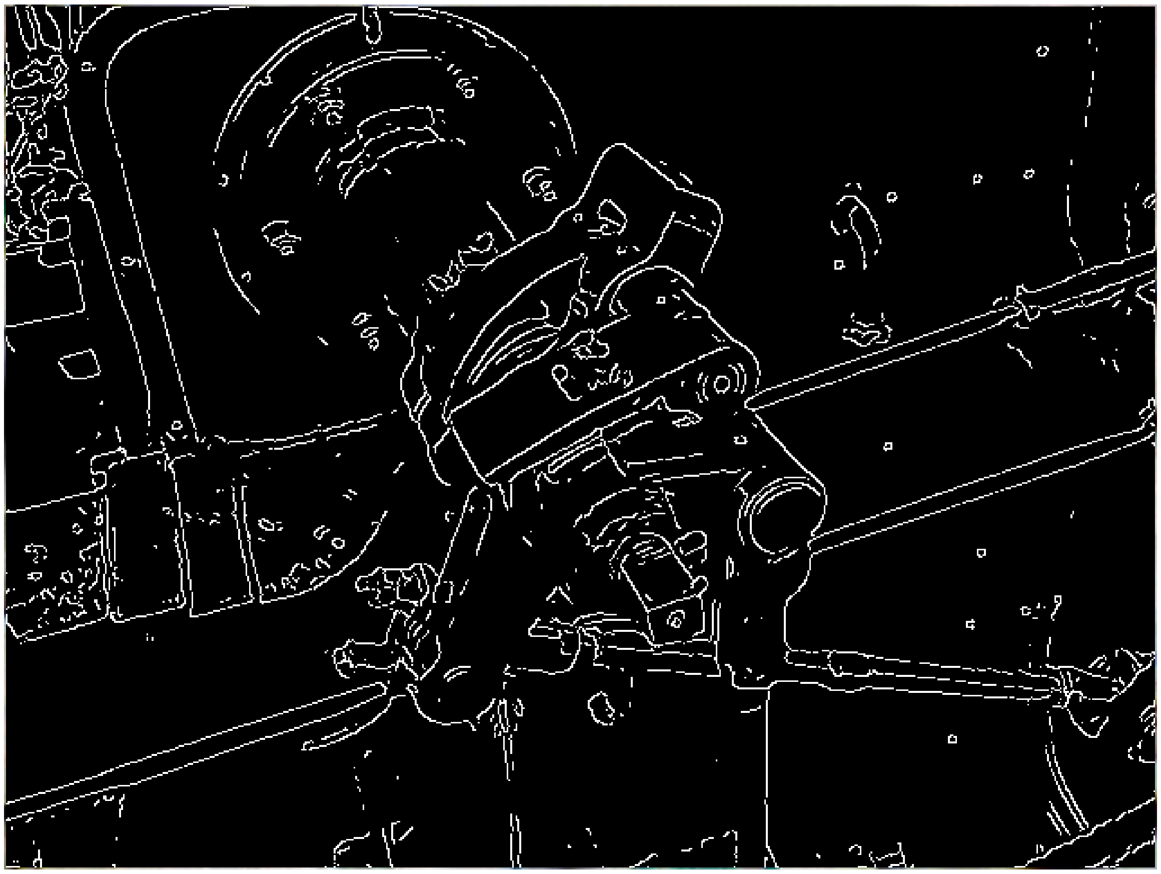
Example

Let us suppose that a sensor is transmitting a VGA frame 640x480. The video frame is of a format RGB888. Let's assume that we have a camera sensor transmitting 8 Bit's per Pixel clock (PCLK). This means to transfer one pixel of data, 3 PCLK's would be required. The HSYNC would be fired by the sensor after every 640 x 3, 1920 PCLK's. A VSYNC would be fired by the sensor after the entire frame is transmitted i.e. after 1920x480, 921600 PCLK's.

The camera interface's hardware block (that could be a part of any SOC) would constantly monitor the above lines to see of the sensor has transmitted anything. A typical camera interface would come with some internal buffering and would also have an associated DMA to transfer the image to the destination memory. The buffer would capture the incoming pixels to temporarily buffer them, and using the DMA the pixels would be transferred (probably line by line) through multiple burst DMA transfers to a destination address in the memory (pre programmed by the camera interface driver programmer). The camera interface's programmer interface might also give a facility of issuing hardware interrupts upon the receipt of the HSYNC, VSYNC signals to the host micro-controller. This could serve as a useful trigger for DMA reprogramming if required.

Canny edge detector

The **Canny edge detection** operator was developed by John F. Canny in 1986 and uses a multi-stage algorithm to detect a wide range of edges in images. Most importantly, Canny also produced a *computational theory of edge detection* explaining why the technique works.



The Canny edge detector applied to a colour photograph of a steam engine

Development of the Canny algorithm

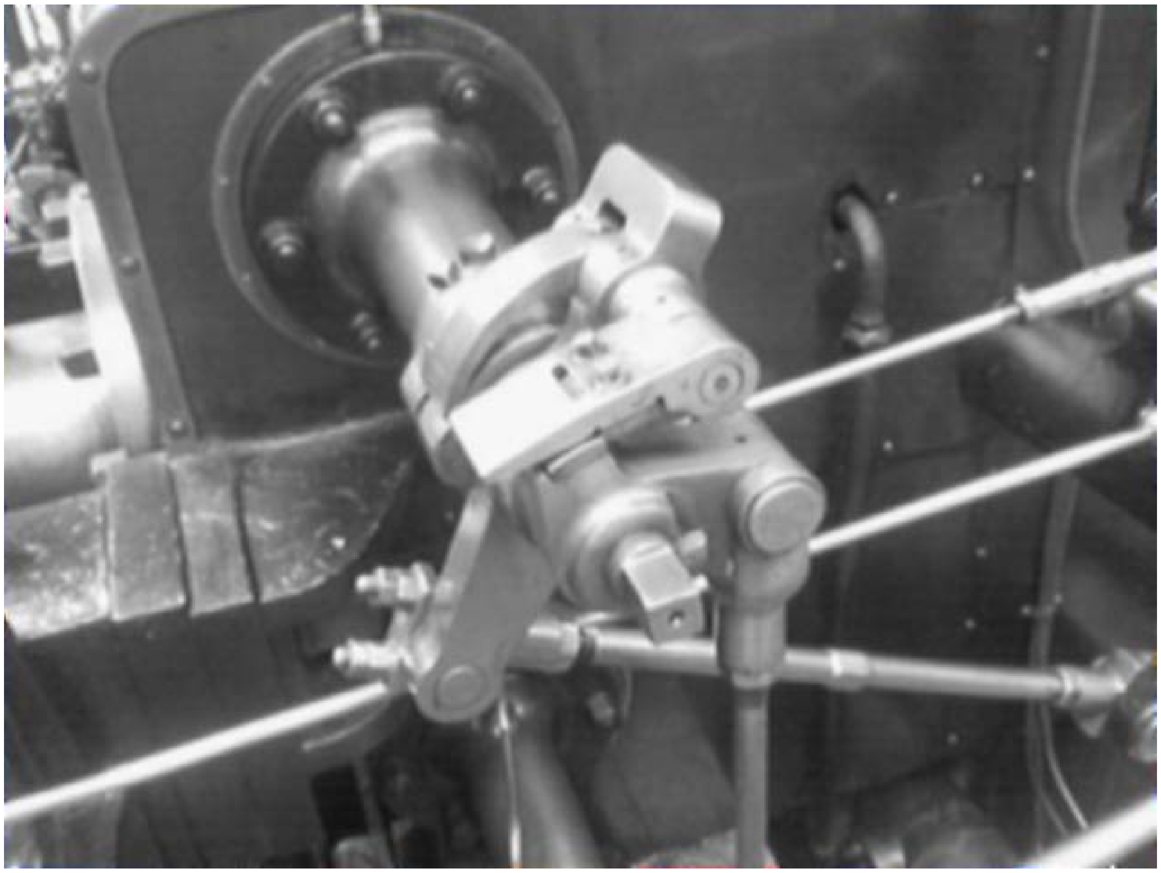
Canny's aim was to discover the optimal edge detection algorithm. In this situation, an "optimal" edge detector means:

- *good detection* – the algorithm should mark as many real edges in the image as possible.
- *good localization* – edges marked should be as close as possible to the edge in the real image.
- *minimal response* – a given edge in the image should only be marked once, and where possible, image noise should not create false edges.

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but can be approximated by the first derivative of a Gaussian.

Stages of the Canny algorithm

Noise reduction



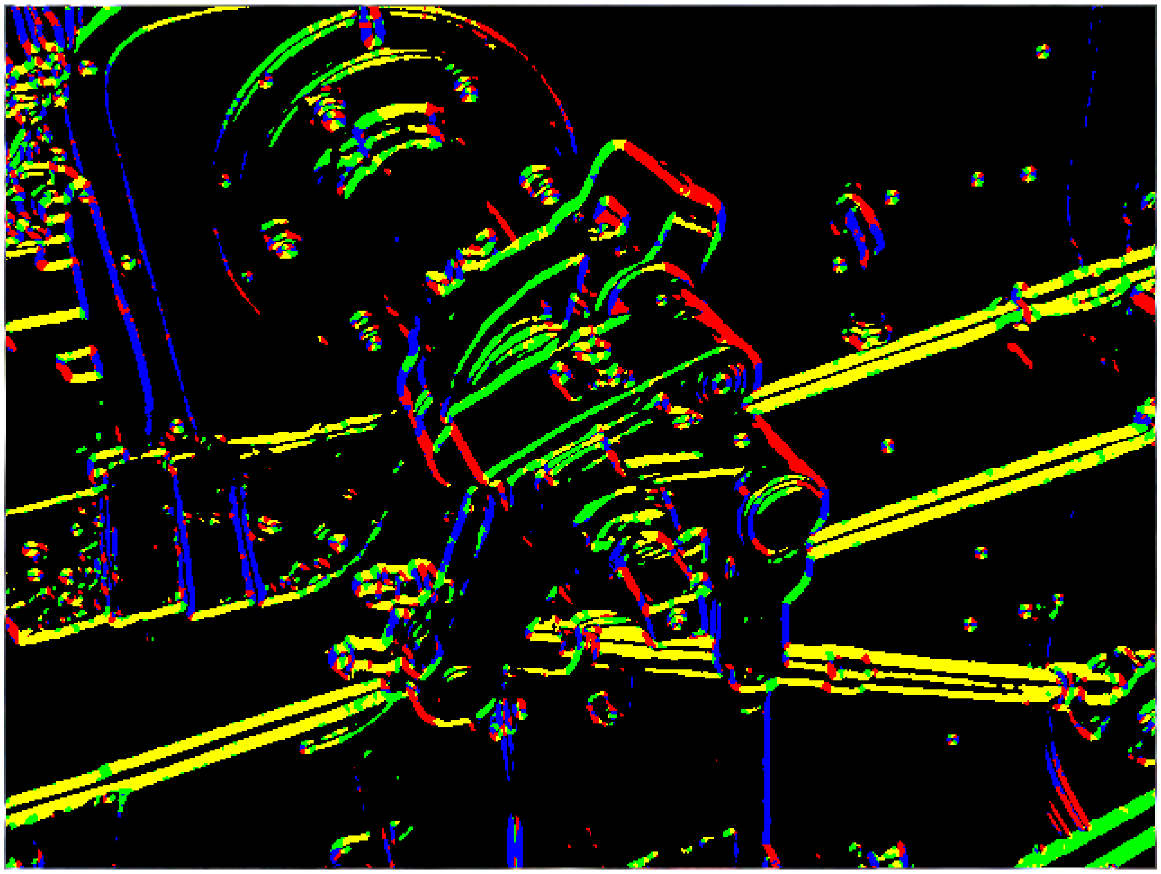
The image after a 5x5 Gaussian mask has been passed across each pixel

The Canny edge detector uses a filter based on the first derivative of a Gaussian, because it is susceptible to noise present on raw unprocessed image data, so to begin with, the raw image is convolved with a Gaussian filter. The result is a slightly blurred version of the original which is not affected by a single noisy pixel to any significant degree.

Here is an example of a 5x5 Gaussian filter, used to create the image to the right, with σ = 1.4:

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}.$$

Finding the intensity gradient of the image



A binary edge map, derived from the Sobel operator, with a threshold of 80. The edges are coloured to indicate the edge direction: yellow for zero degrees, green for 45 degrees, blue for 90 degrees and red for 135 degrees.

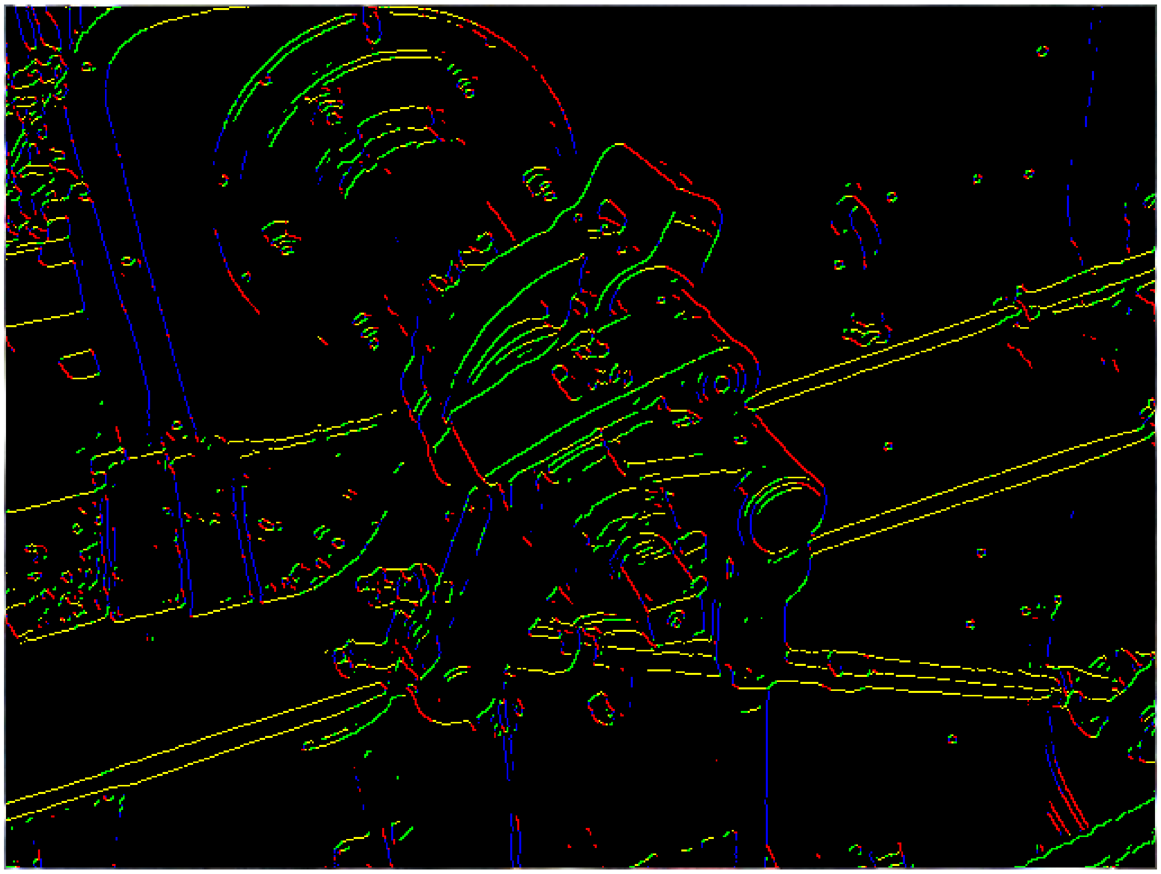
An edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator (Roberts, Prewitt, Sobel for example) returns a value for the first derivative in the horizontal direction (Gy) and the vertical direction (Gx). From this the edge gradient and direction can be determined:

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

$$\mathbf{\Theta} = \operatorname{arctan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right).$$

The edge direction angle is rounded to one of four angles representing vertical, horizontal and the two diagonals (0, 45, 90 and 135 degrees for example).

Non-maximum suppression



The same binary map shown on the left after non-maxima suppression. The edges are still coloured to indicate direction.

Given estimates of the image gradients, a search is then carried out to determine if the gradient magnitude assumes a local maximum in the gradient direction. So, for example,

- if the rounded angle is zero degrees the point will be considered to be on the edge if its intensity is greater than the intensities in the **west and east** directions,
- if the rounded angle is 90 degrees the point will be considered to be on the edge if its intensity is greater than the intensities in the **north and south** directions,
- if the rounded angle is 135 degrees the point will be considered to be on the edge if its intensity is greater than the intensities in the **north west and south east** directions,
- if the rounded angle is 45 degrees the point will be considered to be on the edge if its intensity is greater than the intensities in the **north east and south west** directions.

From this stage referred to as non-maximum suppression, a set of edge points, in the form of a binary image, is obtained. These are sometimes referred to as "thin edges".

Tracing edges through the image and hysteresis thresholding

Intensity gradients which are large are more likely to correspond to edges than if they are small. It is in most cases impossible to specify a threshold at which a given intensity gradient switches from corresponding to an edge into not doing so. Therefore Canny uses thresholding with hysteresis.

Thresholding with hysteresis requires two thresholds – high and low. Making the assumption that important edges should be along continuous curves in the image allows us to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. Therefore we begin by applying a high threshold. This marks out the edges we can be fairly sure are genuine. Starting from these, using the directional information derived earlier, edges can be traced through the image. While tracing an edge, we apply the lower threshold, allowing us to trace faint sections of edges as long as we find a starting point.

Once this process is complete we have a binary image where each pixel is marked as either an edge pixel or a non-edge pixel. From complementary output from the edge tracing step, the binary edge map obtained in this way can also be treated as a set of edge curves, which after further processing can be represented as polygons in the image domain.

Differential geometric formulation of the Canny edge detector

A more refined approach to obtain edges with sub-pixel accuracy is by using the approach of differential edge detection, where the requirement of non-maximum suppression is formulated in terms of second- and third-order derivatives computed from a scale-space representation (Lindeberg 1998).

Variational-geometric formulation of the Haralick-Canny edge detector

A variational explanation for the main ingredient of the Canny edge detector, that is, finding the zero crossings of the 2nd derivative along the gradient direction, was shown to be the result of minimizing a Kronrod-Minkowski functional while maximizing the integral over the alignment of the edge with the gradient field (Kimmel and Bruckstein 2003).

Parameters

The Canny algorithm contains a number of adjustable parameters, which can affect the computation time and effectiveness of the algorithm.

- The size of the Gaussian filter: the smoothing filter used in the first stage directly affects the results of the Canny algorithm. Smaller filters cause less blurring, and allow detection of small, sharp lines. A larger filter causes more blurring, smearing out the value of a given pixel over a larger area of the image. Larger blurring radii are more useful for detecting larger, smoother edges – for instance, the edge of a rainbow.

- Thresholds: the use of two thresholds with hysteresis allows more flexibility than in a single-threshold approach, but general problems of thresholding approaches still apply. A threshold set too high can miss important information. On the other hand, a threshold set too low will falsely identify irrelevant information (such as noise) as important. It is difficult to give a generic threshold that works well on all images. No tried and tested approach to this problem yet exists.

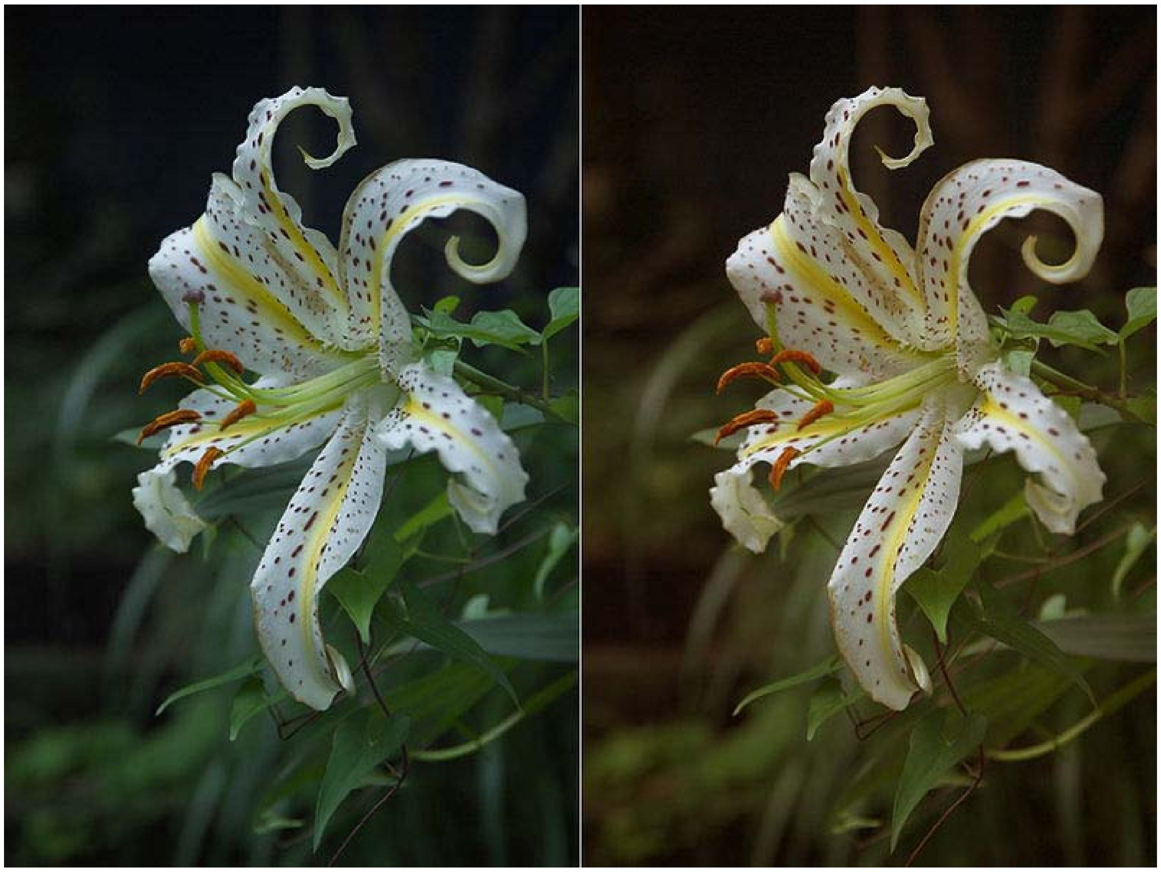
Conclusion

The Canny algorithm is adaptable to various environments. Its parameters allow it to be tailored to recognition of edges of differing characteristics depending on the particular requirements of a given implementation. In Canny's original paper, the derivation of the optimal filter led to a Finite Impulse Response filter, which can be slow to compute in the spatial domain if the amount of smoothing required is important (the filter will have a large spatial support in that case). For this reason, it is often suggested to use Rachid Deriche's Infinite Impulse Response form of Canny's filter (the Canny-Deriche detector), which is recursive, and which can be computed in a short, fixed amount of time for any desired amount of smoothing. The second form is suitable for real time implementations in FPGAs or DSPs, or very fast embedded PCs. In this context, however, the regular recursive implementation of the Canny operator does not give a good approximation of rotational symmetry and therefore gives a bias towards horizontal and vertical edges.

Chapter 11

Color Balance and Segmentation (Image Processing)

Color balance



The left half shows the photo as it came from the digital camera. The right half shows the photo adjusted to make a gray surface neutral in the same light.

A seascape photograph at Clifton Beach, South Arm, Tasmania, Australia. The white balance has been adjusted towards the warm side for creative effect.

Photograph of a ColorChecker as a reference shot for color balance adjustments

In photography and image processing, **color balance** is the global adjustment of the intensities of the colors (typically red, green, and blue primary colors). An important goal of this adjustment is to render specific colors – particularly neutral colors – correctly; hence, the general method is sometimes called **gray balance**, **neutral balance**, or **white balance**. Color balance changes the overall mixture of colors in an image and is used for color correction; generalized versions of color balance are used to get colors other than neutrals to also appear correct or pleasing.

Image data acquired by sensors – either film or electronic image sensors – must be transformed from the acquired values to new values that are appropriate for color reproduction or display. Several aspects of the acquisition and display process make such color correction essential – including the fact that the acquisition sensors do not match the sensors in the human eye, that the properties of the display medium must be accounted for, and that the ambient viewing conditions of the acquisition differ from the display viewing conditions.

The color balance operations in popular image editing applications usually operate directly on the red, green, and blue channel pixel values, without respect to any color sensing or reproduction model. In shooting film, color balance is typically achieved by using color correction filters over the lights or on the camera lens.

Generalized color balance

Sometimes the adjustment to keep neutrals neutral is called *white balance*, and the phrase *color balance* refers to the adjustment that in addition makes other colors in a displayed image appear to have the same general appearance as the colors in an original scene. It is particularly important that neutral (gray, achromatic, white) colors in a scene appear neutral in the reproduction. Hence, the special case of balancing the neutral colors (sometimes *gray balance*, *neutral balance*, or *white balance*) is a particularly important – perhaps dominant – element of color balancing.

Normally, one would not use the phrase *color balance* to describe the adjustments needed to account for differences between the sensors and the human eye, or the details of the display primaries. *Color balance* is normally reserved to refer to correction for differences in the ambient illumination conditions. However, the algorithms for transforming the data do not always clearly separate out the different elements of the correction. Hence, it can be difficult to assign color balance to a specific step in the color correction process. Moreover, there can be significant differences in the color balancing goal. Some applications are created to produce an accurate rendering – as suggested above. In other applications, the goal of color balancing is to produce a pleasing rendering. This difference also creates difficulty in defining the color balancing processing operations.

Illuminant estimation and adaptation

Most digital cameras have a means to select a color correction based on the type of scene illumination, using either manual illuminant selection, or automatic white balance (AWB), or custom white balance. The algorithm that performs this analysis performs generalized color balancing, known as illuminant adaptation or chromatic adaptation.

Many methods are used to achieve color balancing. Setting a button on a camera is a way for the user to indicate to the processor the nature of the scene lighting. Another option on some cameras is a button which one may press when the camera is pointed at a gray card or other neutral object. This "custom white balance" step captures an image of the ambient light, and this information is helpful in controlling color balance.

There is a large literature on how one might estimate the ambient illumination from the camera data and then use this information to transform the image data. A variety of algorithms have been proposed, and the quality of these have been debated. A few examples and examination of the references therein will lead the reader to many others. Examples are Retinex, an artificial neural network or a Bayesian method.

Color balance and chromatic colors

Color balancing an image affects not only the neutrals, but other colors as well. An image that is not color balanced is said to have a color cast, as everything in the image appears to have been shifted towards one color or another. Color balancing may be thought in terms of removing this color cast.

Color balance is also related to color constancy. Algorithms and techniques used to attain color constancy are frequently used for color balancing, as well. Color constancy is, in turn, related to chromatic adaptation. Conceptually, color balancing consists of two steps: first, determining the illuminant under which an image was captured; and second, scaling the components (e.g., R, G, and B) of the image or otherwise transforming the components so they conform to the viewing illuminant.

Viggiano found that white balancing in the camera's native RGB tended to produce less color inconstancy (i.e., less distortion of the colors) than in monitor RGB for over 4000 hypothetical sets of camera sensitivities. This difference typically amounted to a factor of more than two in favor of camera RGB. This means that it is advantageous to get color balance right at the time an image is captured, rather than edit later on a monitor. If one must color balance later, balancing the raw image data will tend to produce less distortion of chromatic colors than balancing in monitor RGB.

Mathematics of color balance

Color balancing is sometimes performed on a three-component image (e.g., RGB) using a 3x3 matrix. This type of transformation is appropriate if the image were captured using the wrong white balance setting on a digital camera, or through a color filter.

Scaling monitor R, G, and B

In principle, one wants to scale all relative luminances in an image so that objects which are believed to be neutral appear so. If, say, a surface with $R = 240$ was believed to be a white object, and if 255 is the count which corresponds to white, one could multiply all red values by 255/240. Doing analogously for green and blue would result, at least in theory, in a color balanced image. In this type of transformation the 3x3 matrix is a diagonal matrix.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 255/R'_w & 0 & 0 \\ 0 & 255/G'_w & 0 \\ 0 & 0 & 255/B'_w \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

where R, G, and B are the color balanced red, green, and blue components of a pixel in the image; R', G', and B' are the red, green, and blue components of the image before color balancing, and R'_w, G'_w, and B'_w are the red, green, and blue components of a pixel which is believed to be a white surface in the image before color balancing. This is a simple scaling of the red, green, and blue channels, and is why color balance tools in Photoshop and the GIMP have a white eyedropper tool. It has been demonstrated that performing the white balancing in the phosphor set assumed by sRGB tends to produce large errors in chromatic colors, even though it can render the neutral surfaces perfectly neutral.

Scaling X, Y, Z

If the image may be transformed into CIE XYZ tristimulus values, the color balancing may be performed there. This has been termed a “wrong von Kries” transformation. Although it has been demonstrated to offer usually poorer results than balancing in monitor RGB, it is mentioned here as a bridge to other things. Mathematically, one computes:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_w/X'_w & 0 & 0 \\ 0 & Y_w/Y'_w & 0 \\ 0 & 0 & Z_w/Z'_w \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}$$

where X, Y, and Z are the color-balanced tristimulus values; X_w, Y_w, and Z_w are the tristimulus values of the viewing illuminant (the white point to which the image is being transformed to conform to); X'_w, Y'_w, and Z'_w are the tristimulus values of an object believed to be white in the un-color-balanced image, and X', Y', and Z' are the tristimulus values of a pixel in the un-color-balanced image. If the tristimulus values of the monitor primaries are in a matrix $\mathbf{P}$ so that:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{P} \begin{bmatrix} L_R \\ L_G \\ L_B \end{bmatrix}$$

where L_R, L_G, and L_B are the un-gamma corrected monitor RGB, one may use:

$$\begin{bmatrix} L_R \\ L_G \\ L_B \end{bmatrix} = \mathbf{P}^{-1} \begin{bmatrix} X_w/X'_w & 0 & 0 \\ 0 & Y_w/Y'_w & 0 \\ 0 & 0 & Z_w/Z'_w \end{bmatrix} \mathbf{P} \begin{bmatrix} L_{R'} \\ L_{G'} \\ L_{B'} \end{bmatrix}$$

Von Kries's method

Johannes von Kries, whose theory of rods and three different color-sensitive cone types in the retina has survived as the dominant explanation of color sensation for over 100 years, motivated the method of converting color to the LMS color space, representing the effective stimuli for the Long-, Medium-, and Short-wavelength cone types that are modeled as adapting independently. A 3x3 matrix converts RGB or XYZ to LMS, and then the three LMS primary values are scaled to balance the neutral; the color can then be converted back to the desired final color space:

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 1/L'_w & 0 & 0 \\ 0 & 1/M'_w & 0 \\ 0 & 0 & 1/S'_w \end{bmatrix} \begin{bmatrix} L' \\ M' \\ S' \end{bmatrix}$$

where L, M, and S are the color-balanced LMS cone tristimulus values; L'_w, M'_w, and S'_w are the tristimulus values of an object believed to be white in the un-color-balanced image, and L', M', and S' are the tristimulus values of a pixel in the un-color-balanced image.

Matrices to convert to LMS space were not specified by von Kries, but can be derived from CIE color matching functions and LMS color matching functions when the latter are specified; matrices can also be found in reference books.

Scaling camera RGB

By Viggiano's measure, and using his model of gaussian camera spectral sensitivities, most camera RGB spaces performed better than either monitor RGB or XYZ. If the camera's raw RGB values are known, one may use the 3x3 diagonal matrix:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 255/R'_w & 0 & 0 \\ 0 & 255/G'_w & 0 \\ 0 & 0 & 255/B'_w \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

and then convert to a working RGB space such as sRGB or Adobe RGB after balancing.

Preferred chromatic adaptation spaces

Comparisons of images balanced by diagonal transforms in a number of different RGB spaces have identified several such spaces that work better than others, and better than camera or monitor spaces, for chromatic adaptation, as measured by several color appearance models; the systems that performed statistically as well as the best on the majority of the image test sets used were the "Sharp", "Bradford", "CMCCAT", and "ROMM" spaces.

General illuminant adaptation

The best color matrix for adapting to a change in illuminant is not necessarily a diagonal matrix in a fixed color space. It has long been known that if the space of illuminants can be described as a linear model with *N* basis terms, the proper color transformation will be the weighted sum of *N* fixed linear transformations, not necessarily consistently diagonalizable.

Segmentation (image processing)

In computer vision, **segmentation** refers to the process of partitioning a digital image into multiple segments (sets of pixels, also known as superpixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain visual characteristics.

The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image. Each of the pixels in a region are similar with respect to some characteristic or computed property, such as color, intensity, or texture. Adjacent regions are significantly different with respect to the same characteristic(s).

Applications

Some of the practical applications of image segmentation are:

- Medical Imaging
    - Locate tumors and other pathologies
    - Measure tissue volumes

    - Computer-guided surgery
    - Diagnosis
    - Treatment planning
    - Study of anatomical structure
- Locate objects in satellite images (roads, forests, etc.)
- Face recognition
- Fingerprint recognition
- Traffic control systems
- Brake light detection
- Machine vision

Several general-purpose algorithms and techniques have been developed for image segmentation. Since there is no general solution to the image segmentation problem, these techniques often have to be combined with domain knowledge in order to effectively solve an image segmentation problem for a problem domain.

Clustering methods

The K-means algorithm is an iterative technique that is used to partition an image into *K* clusters. The basic algorithm is:

1. Pick *K* cluster centers, either randomly or based on some heuristic
2. Assign each pixel in the image to the cluster that minimizes the distance between the pixel and the cluster center
3. Re-compute the cluster centers by averaging all of the pixels in the cluster
4. Repeat steps 2 and 3 until convergence is attained (e.g. no pixels change clusters)

In this case, distance is the squared or absolute difference between a pixel and a cluster center. The difference is typically based on pixel color, intensity, texture, and location, or a weighted combination of these factors. *K* can be selected manually, randomly, or by a heuristic.

This algorithm is guaranteed to converge, but it may not return the optimal solution. The quality of the solution depends on the initial set of clusters and the value of *K*.

In statistics and machine learning, the k-means algorithm is clustering algorithm to partition n objects into k clusters, where $k < n$. It is similar to the expectation-maximization algorithm for mixtures of Gaussians in that they both attempt to find the centers of natural clusters in the data. The model requires that the object attributes correspond to elements of a vector space. The objective it tries to achieve is to minimize total intra-cluster variance, or, the squared error function. The k-means clustering was invented in 1956. The most common form of the algorithm uses an iterative refinement heuristic known as Lloyd's algorithm. Lloyd's algorithm starts by partitioning the input points into k initial sets, either at random or using some heuristic data. It then calculates the mean point, or centroid, of each set. It constructs a new partition by associating each point with the closest centroid. Then the centroids are recalculated for the new clusters, and algorithm repeated by alternate application of these two steps until convergence, which is obtained when the points no longer switch clusters (or alternatively centroids are no longer changed). Lloyd's algorithm and k-means are often used synonymously, but in reality Lloyd's algorithm is a heuristic for solving the k-means problem, as with certain combinations of starting points and centroids, Lloyd's algorithm can in fact converge to the wrong answer. Other variations exist, but Lloyd's algorithm has remained popular, because it converges extremely quickly in practice. In terms of performance the algorithm is not guaranteed to return a global optimum. The quality of the final solution depends largely on the initial set of clusters, and may, in practice, be much poorer than the global optimum. Since the algorithm is extremely fast, a common method is to run the algorithm several times and return the best clustering found. A drawback of the k-means algorithm is that the number of clusters k is an input parameter. An inappropriate choice of k may yield poor results. The algorithm also assumes that the variance is an appropriate measure of cluster scatter.

Compression-based methods

Compression based methods postulate that the optimal segmentation is the one that minimizes, over all possible segmentations, the coding length of the data . The connection between these two concepts is that segmentation tries to find patterns in an image and any regularity in the image can be used to compress it. The method describes each segment by its texture and boundary shape. Each of these components is modeled by a probability distribution function and its coding length is computed as follows:

1. The boundary encoding leverages the fact that regions in natural images tend to have a smooth contour. This prior is used by huffman coding to encode the difference chain code of the contours in an image. Thus, the smoother a boundary is, the shorter coding length it attains.
2. Texture is encoded by lossy compression in a way similar to minimum description length (MDL) principle, but here the length of the data given the model is approximated by the number of samples times the entropy of the model. The texture in each region is modeled by a multivariate normal distribution whose entropy has closed form expression. An interesting property of this model is that the estimated entropy bounds the true entropy of the data from above. This is because among all distributions with a given mean and covariance, normal distribution has the largest entropy. Thus, the true coding length cannot be more than what the algorithm tries to minimize.

For any given segmentation of an image, this scheme yields the number of bits required to encode that image based on the given segmentation. Thus, among all possible segmentations of an image, the goal is to find the segmentation which produces the shortest coding length. This can be achieved by a simple agglomerative clustering method. The distortion in the lossy compression determines the coarseness of the segmentation and its optimal value may differ for each image. This parameter can be estimated heuristically from the contrast of textures in an image. For example, when the textures in an image are similar, such as in camouflage images, stronger sensitivity and thus lower quantization is required.

Histogram-based methods

Histogram-based methods are very efficient when compared to other image segmentation methods because they typically require only one pass through the pixels. In this technique, a histogram is computed from all of the pixels in the image, and the peaks and valleys in the histogram are used to locate the clusters in the image. Color or intensity can be used as the measure.

A refinement of this technique is to recursively apply the histogram-seeking method to clusters in the image in order to divide them into smaller clusters. This is repeated with smaller and smaller clusters until no more clusters are formed.

One disadvantage of the histogram-seeking method is that it may be difficult to identify significant peaks and valleys in the image. In this technique of image classification distance metric and integrated region matching are familiar.

Histogram-based approaches can also be quickly adapted to occur over multiple frames, while maintaining their single pass efficiency. The histogram can be done in multiple fashions when multiple frames are considered. The same approach that is taken with one frame can be applied to multiple, and after the results are merged, peaks and valleys that were previously difficult to identify are more likely to be distinguishable. The histogram can also be applied on a per pixel basis where the information result are used to determine the most frequent color for the pixel location. This approach segments based on active objects and a static environment, resulting in a different type of segmentation useful in Video tracking.

Edge detection

Edge detection is a well-developed field on its own within image processing. Region boundaries and edges are closely related, since there is often a sharp adjustment in intensity at the region boundaries. Edge detection techniques have therefore been used as the base of another segmentation technique.

The edges identified by edge detection are often disconnected. To segment an object from an image however, one needs closed region boundaries.

Region growing methods

The first region growing method was the seeded region growing method. This method takes a set of seeds as input along with the image. The seeds mark each of the objects to be segmented. The regions are iteratively grown by comparing all unallocated neighbouring pixels to the regions. The difference between a pixel's intensity value and the region's mean, δ, is used as a measure of similarity. The pixel with the smallest difference measured this way is allocated to the respective region. This process continues until all pixels are allocated to a region.

Seeded region growing requires seeds as additional input. The segmentation results are dependent on the choice of seeds. Noise in the image can cause the seeds to be poorly placed. Unseeded region growing is a modified algorithm that doesn't require explicit seeds. It starts off with a single region A_1 – the pixel chosen here does not significantly influence final segmentation. At each iteration it considers the neighbouring pixels in the same way as seeded region growing. It differs from seeded region growing in that if the minimum δ is less than a predefined threshold T then it is added to the respective region A_j. If not, then the pixel is considered significantly different from all current regions A_i and a new region A_{n+1} is created with this pixel.

One variant of this technique, proposed by Haralick and Shapiro (1985), is based on pixel intensities. The mean and scatter of the region and the intensity of the candidate pixel is used to compute a test statistic. If the test statistic is sufficiently small, the pixel is added to the region, and the region's mean and scatter are recomputed. Otherwise, the pixel is rejected, and is used to form a new region.

Partial Differential Equation based methods

Using a Partial Differential Equation (PDE) based method and solving the PDE equation by a numerical scheme, one can segment the image.

Level Set methods

Curve propagation is a popular technique in image analysis for object extraction, object tracking, stereo reconstruction, etc. The central idea behind such an approach is to evolve a curve towards the lowest potential of a cost function, where its definition reflects the task to be addressed and imposes certain smoothness constraints. Lagrangian techniques are based on parameterizing the contour according to some sampling strategy and then evolve each element according to image and internal terms. While such a technique can be very efficient, it suffers from various limitations like deciding on the sampling strategy, estimating the internal geometric properties of the curve, changing its topology, addressing problems in higher dimensions, etc. In each case, a partial differential equation (PDE) called the level set equation is solved by finite differences.

The level set method was initially proposed to track moving interfaces by Osher and Sethian in 1988 and has spread across various imaging domains in the late nineties. It can be used to efficiently address the problem of curve/surface/etc. propagation in an implicit manner. The central idea is to represent the evolving contour using a signed function, where its zero level corresponds to the actual contour. Then, according to the motion equation of the contour, one can easily derive a similar flow for the implicit surface that when applied to the zero-level will reflect the propagation of the contour. The level set method encodes numerous advantages: it is implicit, parameter free, provides a direct way to estimate the geometric properties of the evolving structure, can change the topology and is intrinsic. Furthermore, they can be used to define an optimization framework as proposed by Zhao, Merriman and Osher in 1996. Therefore, one can conclude that it is a very convenient framework to address numerous applications of computer vision and medical image analysis. Furthermore, research into various level set data structures has led to very efficient implementations of this method.

Graph partitioning methods

Graph partitioning methods can effectively be used for image segmentation. In these methods, the image is modeled as a weighted, undirected graph. Usually a pixel or a group of pixels are associated with nodes and edge weights define the (dis)similarity between the neighborhood pixels. The graph (image) is then partitioned according to a criterion designed to model "good" clusters. Each partition of the nodes (pixels) output from these algorithms are considered an object segment in the image. Some popular algorithms of this category are normalized cuts, random walker, minimum cut, isoperimetric partitioning and minimum spanning tree-based segmentation .

Watershed transformation

The watershed transformation considers the gradient magnitude of an image as a topographic surface. Pixels having the highest gradient magnitude intensities (GMIs) correspond to watershed lines, which represent the region boundaries. Water placed on any pixel enclosed by a common watershed line flows downhill to a common local intensity minimum (LIM). Pixels draining to a common minimum form a catch basin, which represents a segment.

Model based segmentation

The central assumption of such an approach is that structures of interest/organs have a repetitive form of geometry. Therefore, one can seek for a probabilistic model towards explaining the variation of the shape of the organ and then when segmenting an image impose constraints using this model as prior. Such a task involves (i) registration of the training examples to a common pose, (ii) probabilistic representation of the variation of the registered samples, and (iii) statistical inference between the model and the image. State of the art methods in the literature for knowledge-based segmentation involve active shape and appearance models, active contours and deformable templates and level-set based methods.

Multi-scale segmentation

Image segmentations are computed at multiple scales in scale-space and sometimes propagated from coarse to fine scales.

Segmentation criteria can be arbitrarily complex and may take into account global as well as local criteria. A common requirement is that each region must be connected in some sense.

One-dimensional hierarchical signal segmentation

Witkin's seminal work in scale space included the notion that a one-dimensional signal could be unambiguously segmented into regions, with one scale parameter controlling the scale of segmentation.

A key observation is that the zero-crossings of the second derivatives (minima and maxima of the first derivative or slope) of multi-scale-smoothed versions of a signal form a nesting tree, which defines hierarchical relations between segments at different scales. Specifically, slope extrema at coarse scales can be traced back to corresponding features at fine scales. When a slope maximum and slope minimum annihilate each other at a larger scale, the three segments that they separated merge into one segment, thus defining the hierarchy of segments.

Image segmentation and primal sketch

There have been numerous research works in this area, out of which a few have now reached a state where they can be applied either with interactive manual intervention (usually with application to medical imaging) or fully automatically. The following is a brief overview of some of the main research ideas that current approaches are based upon.

The nesting structure that Within described is, however, specific for one-dimensional signals and does not trivially transfer to higher-dimensional images. Nevertheless, this general idea has inspired several other authors to investigate coarse-to-fine schemes for image segmentation. Koenderink proposed to study how iso-intensity contours evolve over scales and this approach was investigated in more detail by Lifshitz and Pizer. Unfortunately, however, the intensity of image features changes over scales, which implies that it is hard to trace coarse-scale image features to finer scales using iso-intensity information.

Lindeberg studied the problem of linking local extrema and saddle points over scales, and proposed an image representation called the scale-space primal sketch which makes explicit the relations between structures at different scales, and also makes explicit which image features are stable over large ranges of scale including locally appropriate scales for those. Bergholm proposed to detect edges at coarse scales in scale-space and then trace them back to finer scales with manual choice of both the coarse detection scale and the fine localization scale.

Gauch and Pizer studied the complementary problem of ridges and valleys at multiple scales and developed a tool for interactive image segmentation based on multi-scale watersheds. The use of multi-scale watershed with application to the gradient map has also been investigated by Olsen and Nielsen and been carried over to clinical use by Dam Vincken et al. proposed a hyperstack for defining probabilistic relations between image structures at different scales. The use of stable image structures over scales has been furthered by Ahuja and his co-workers into a fully automated system.

More recently, these ideas for multi-scale image segmentation by linking image structures over scales have been picked up by Florack and Kuijper. Bijaoui and Rué associate structures detected in scale-space above a minimum noise threshold into an object tree which spans multiple scales and corresponds to a kind of feature in the original signal. Extracted features are accurately reconstructed using an iterative conjugate gradient matrix method.

Semi-automatic segmentation

In this kind of segmentation, the user outlines the region of interest with the mouse clicks and algorithms are applied so that the path that best fits the edge of the image is shown.

Techniques like Siox, Livewire, Intelligent Scissors or IT-SNAPS are used in this kind of segmentation.

Neural networks segmentation

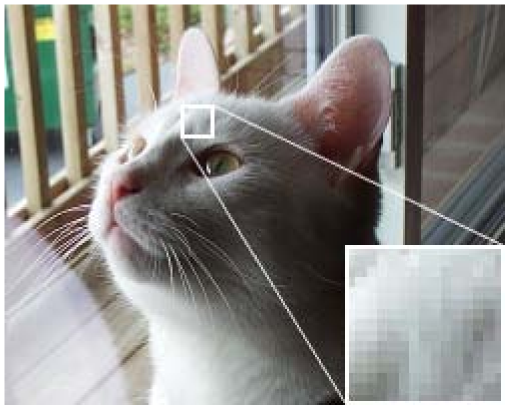
Neural Network segmentation relies on processing small areas of an image using an artificial neural network or a set of neural networks. After such processing the decision-making mechanism marks the areas of an image accordingly to the category recognized by the neural network. A type of network designed especially for this is the Kohonen map.

Pulse-Coupled Neural Networks (PCNNs) are neural models proposed by modeling a cat's visual cortex and developed for high-performance biomimetic image processing. In 1989, Eckhorn introduced a neural model to emulate the mechanism of cat's visual cortex. The Eckhorn model provided a simple and effective tool for studying small mammal's visual cortex, and was soon recognized as having significant application potential in image processing. In 1994, the Eckhorn model was adapted to be an image processing algorithm by Johnson, who termed this algorithm Pulse-Coupled Neural Network. Over the past decade, PCNNs have been utilized for a variety of image processing applications, including: image segmentation, feature generation, face extraction, motion detection, region growing, noise reduction, and so on. A PCNN is a two-dimensional neural network. Each neuron in the network corresponds to one pixel in an input image, receiving its corresponding pixel's color information (e.g. intensity) as an external stimulus. Each neuron also connects with its neighboring neurons, receiving local stimuli from them. The external and local stimuli are combined in an internal activation system, which accumulates the stimuli until it exceeds a dynamic threshold, resulting in a pulse output. Through iterative computation, PCNN neurons produce temporal series of pulse outputs. The temporal series of pulse outputs contain information of input images and can be utilized for various image processing applications, such as image segmentation and feature generation. Compared with conventional image processing means, PCNNs have several significant merits, including robustness against noise, independence of geometric variations in input patterns, capability of bridging minor intensity variations in input patterns, etc.

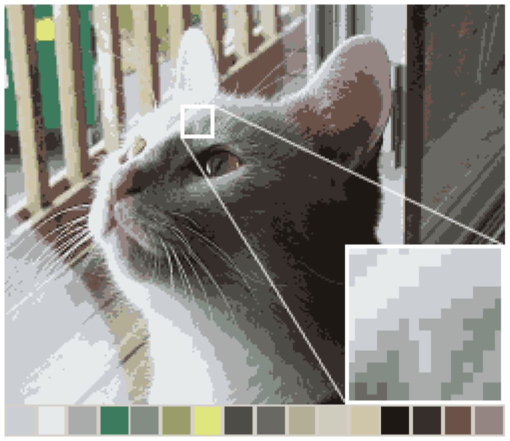
Chapter 12

Color Quantization and Foveated Imaging

Color quantization



An example image in 24-bit RGB color



The same image reduced to a palette of 16 colors specifically chosen to best represent the image; the selected palette is shown by the squares above

In computer graphics, **color quantization** or **color image quantization** is a process that reduces the number of distinct colors used in an image, usually with the intention that the new image should be as visually similar as possible to the original image. Computer algorithms to perform color quantization on bitmaps have been studied since the 1970s. Color quantization is critical for displaying images with many colors on devices that can only display a limited number of colors, usually due to memory limitations, and enables efficient compression of certain types of images.

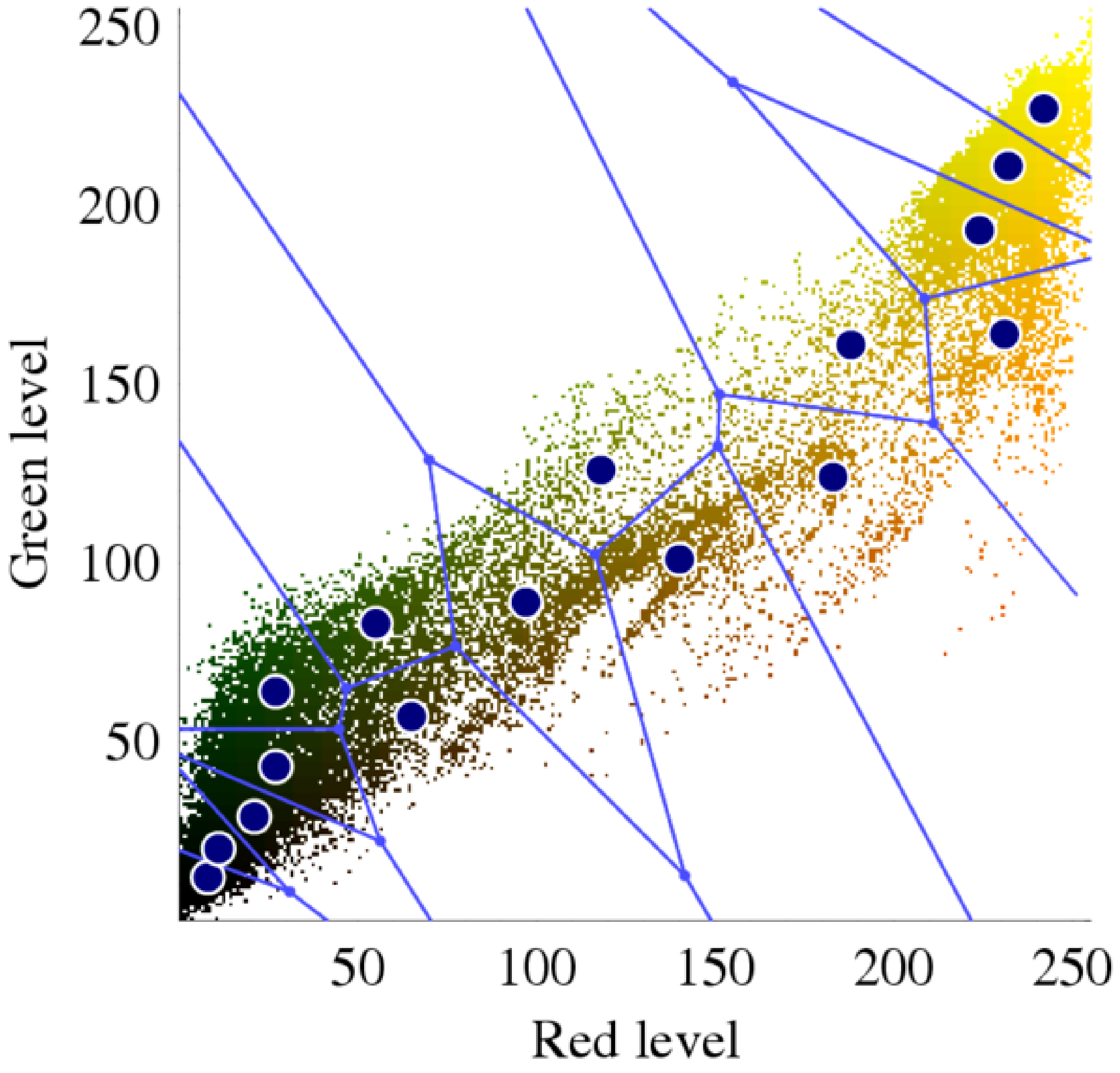
The name "color quantization" is primarily used in computer graphics research literature; in applications, terms such as *optimized palette generation*, *optimal palette generation*, or *decreasing color depth* are used. Some of these are misleading, as the palettes generated by standard algorithms are not necessarily the best possible.

Algorithms

Most standard techniques treat color quantization as a problem of clustering points in three-dimensional space, where the points represent colors found in the original image and the three axes represent the three color channels. Almost any three-dimensional clustering algorithm can be applied to color quantization, and vice versa. After the clusters are located, typically the points in each cluster are averaged to obtain the representative color that all colors in that cluster are mapped to. The three color channels are usually red, green, and blue, but another popular choice is the Lab color space, in which Euclidean distance is more consistent with perceptual difference.

The most popular algorithm by far for color quantization, invented by Paul Heckbert in 1980, is the median cut algorithm. Many variations on this scheme are in use. Before this time, most color quantization was done using the *population algorithm* or *population method*, which essentially constructs a histogram of equal-sized ranges and assigns colors to the ranges containing the most points. A more modern popular method is clustering using octrees, first conceived by Gervautz and Purgathofer and improved by Xerox PARC researcher Dan Bloomberg.

A small photograph that has had its blue channel removed. This means all of its pixel colors lie in a two-dimensional plane in the color cube.



The color space of the photograph to the left, along with a 16-color optimized palette produced by Photoshop. The Voronoi regions of each palette entry are shown.

If the palette is fixed, as is often the case in real-time color quantization systems such as those used in operating systems, color quantization is usually done using the "straight-line distance" or "nearest color" algorithm, which simply takes each color in the original image and finds the closest palette entry, where distance is determined by the distance between the two corresponding points in three-dimensional space. In other words, if the colors are (r_1,g_1,b_1) and (r_2,g_2,b_2), we want to minimize the Euclidean distance:

$$\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}.$$

This effectively decomposes the color cube into a Voronoi diagram, where the palette entries are the points and a cell contains all colors mapping to a single palette entry.

There are efficient algorithms from computational geometry for computing Voronoi diagrams and determining which region a given point falls in; in practice, indexed palettes are so small that these are usually overkill.

Color quantization is frequently combined with dithering, which can eliminate unpleasant artifacts such as banding that appear when quantizing smooth gradients and give the appearance of a larger number of colors. Some modern schemes for color quantization attempt to combine palette selection with dithering in one stage, rather than perform them independently.

A number of other much less frequently used methods have been invented that use entirely different approaches. The Local K-means algorithm, conceived by Oleg Verevka in 1995, is designed for use in windowing systems where a core set of "reserved colors" is fixed for use by the system and many images with different color schemes might be displayed simultaneously. It is a post-clustering scheme that makes an initial guess at the palette and then iteratively refines it.

The high quality but slow *NeuQuant* algorithm reduces images to 256 colors by training a Kohonen neural network "which self-organises through learning to match the distribution of colours in an input image. Taking the position in RGB-space of each neuron gives a high-quality colour map in which adjacent colours are similar." It is particularly advantageous for images with gradients.

Finally, one of the most promising new methods is *spatial color quantization*, conceived by Puzicha, Held, Ketterer, Buhmann, and Fellner of the University of Bonn, which combines dithering with palette generation and a simplified model of human perception to produce visually impressive results even for very small numbers of colors. It does not treat palette selection strictly as a clustering problem, in that the colors of nearby pixels in the original image also affect the color of a pixel.

History and applications

In the early days of PCs, it was common for video adapters to support only 2, 4, 16, or (eventually) 256 colors due to video memory limitations; they preferred to dedicate the video memory to having more pixels (higher resolution) rather than more colors. Color quantization helped to justify this tradeoff by making it possible to display many high color images in 16- and 256-color modes with limited visual degradation. The Windows operating system and many other operating systems automatically perform quantization and dithering when viewing high color images in a 256 color video mode, which was important when video devices limited to 256 color modes were dominant. Modern computers can now display millions of colors at once, far more than can be distinguished by the human eye, limiting this application primarily to mobile devices and legacy hardware.

Nowadays, color quantization is mainly used in GIF and PNG images. GIF, for a long time the most popular lossless and animated bitmap format on the World Wide Web, only supports up to 256 colors, necessitating quantization for many images. Some early web browsers constrained images to use a specific palette known as the web colors, leading to severe degradation in quality compared to optimized palettes. PNG images support 24-bit color, but can often be made much smaller in filesize without much visual degradation by application of color quantization, since PNG files use fewer bits per pixel for palettized images.

The infinite number of colors available through the lens of a camera is impossible to display on a computer screen; thus converting any photograph to a digital representation necessarily involves some quantization. In practice, 24-bit color is sufficiently rich to represent almost all colors perceivable by humans with sufficiently small error as to be visually identical (if presented faithfully.)

With the few colors available on early computers, different quantization algorithms produced very different-looking output images. As a result, a lot of time was spent on writing sophisticated algorithms to be more lifelike.

In his PhD about color quantization, Andreas Schrader from the University of Siegen, Germany developed an evolutionary algorithm using natural selection of an artificial population consisting of color tables. This algorithm outperformed all known algorithms at the time of writing in 1998.

Editor support

Many bitmap graphics editors contain built-in support for color quantization, and will automatically perform it when converting an image with many colors to an image format with fewer colors. Most of these implementations allow the user to set exactly the number of desired colors. Examples of such support include:

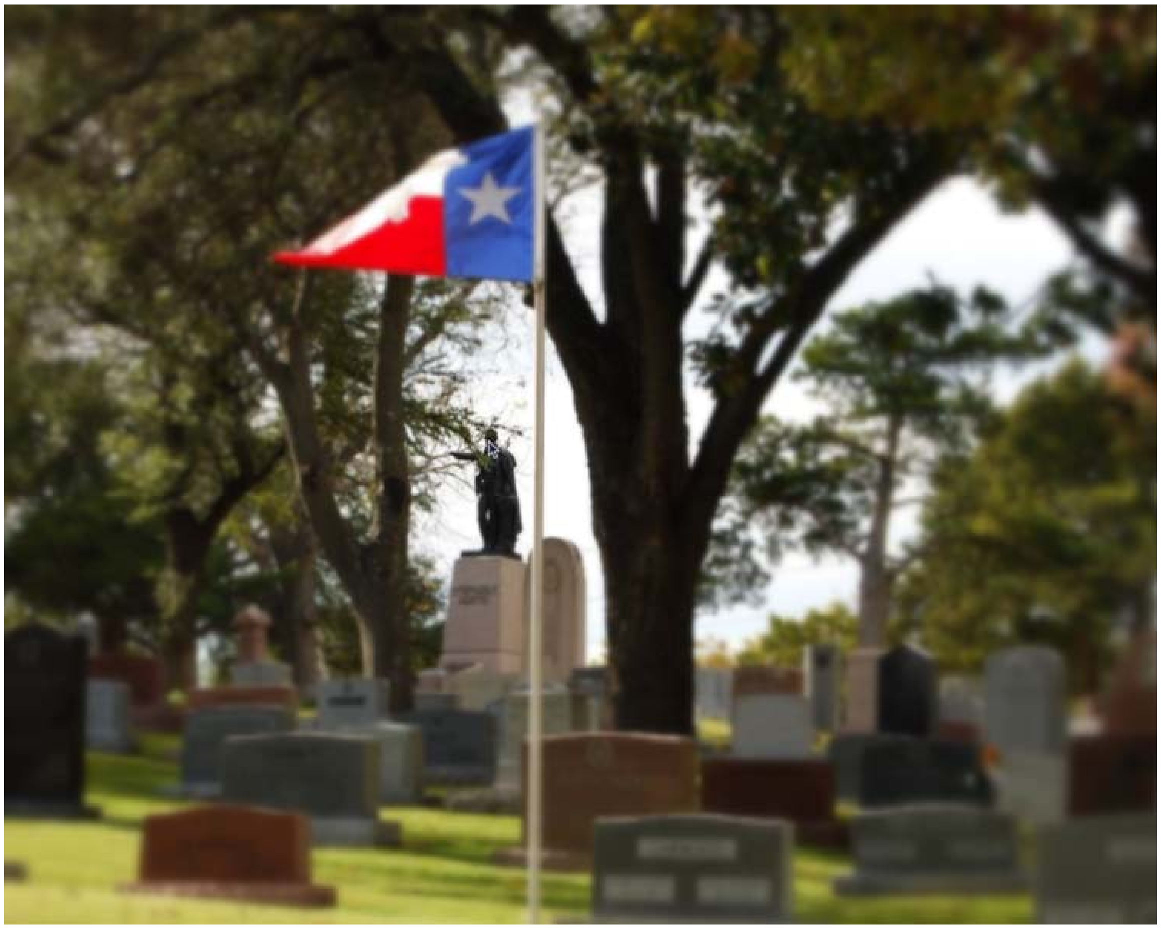
- Photoshop's *Mode→Indexed Color* function, supplies a number of quantization algorithms ranging from the fixed Windows system and Web palettes to the proprietary Local and Global algorithms for generating palettes suited to a particular image or images.
- Paint Shop Pro, in its *Colors→Decrease Color Depth* dialog, supplies three standard color quantization algorithms: median cut, octree, and the fixed standard "web safe" palette.
- The GIMP's *Generate Optimal Palette with 256 Colours* option, known to use the median cut algorithm. There has been some discussion in the developer community of adding support for spatial color quantization.

Color quantization is also used to create posterization effects, although posterization has the slightly different goal of minimizing the number of colors used within the same color space.

Some vector graphics editors also utilize color quantization, especially for raster-to-vector techniques that create tracings of bitmap images with the help of edge detection.

- *Inkscape's Path→Trace Bitmap: Multiple Scans: Color* function uses octree quantization to create color traces.

Foveated imaging



16:1 compression. Foveated image with fixation point at Stephen F. Austin statue.

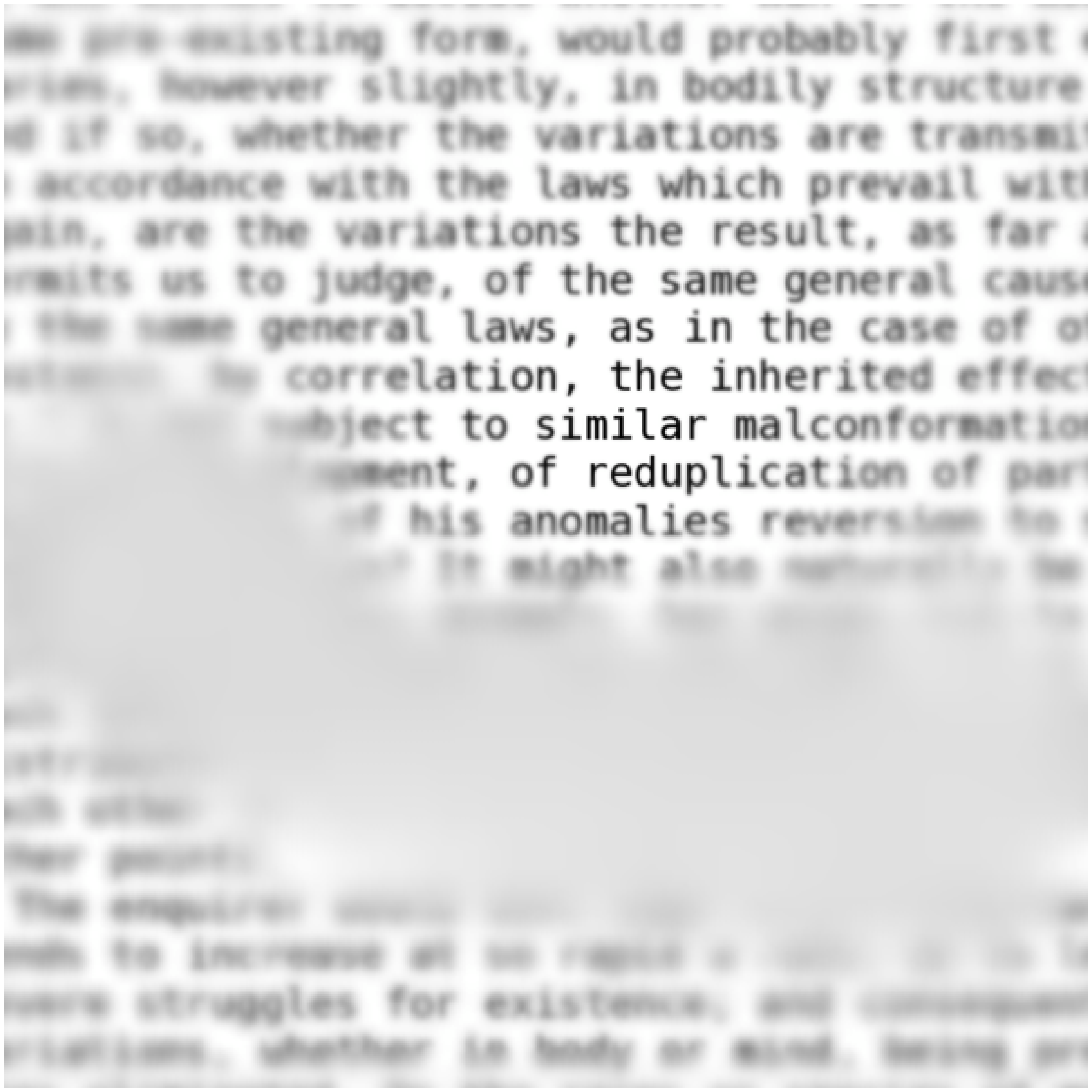
Foveated imaging is a digital image processing technique in which the image resolution, or amount of detail, varies across the image according to one or more "fixation points." A fixation point indicates the highest resolution region of the image and corresponds to the center of the eye's retina, the fovea.

The location of a fixation point may be specified in many ways. For example, when viewing an image on a computer monitor, one may specify a fixation using a pointing device, like a computer mouse. Eye trackers which precisely measure the eye's position and movement are also commonly used to determine fixation points in perception experiments. When the display is manipulated with the use of an eye tracker, this is known as a gaze contingent display. Fixations may also be determined automatically using computer algorithms.

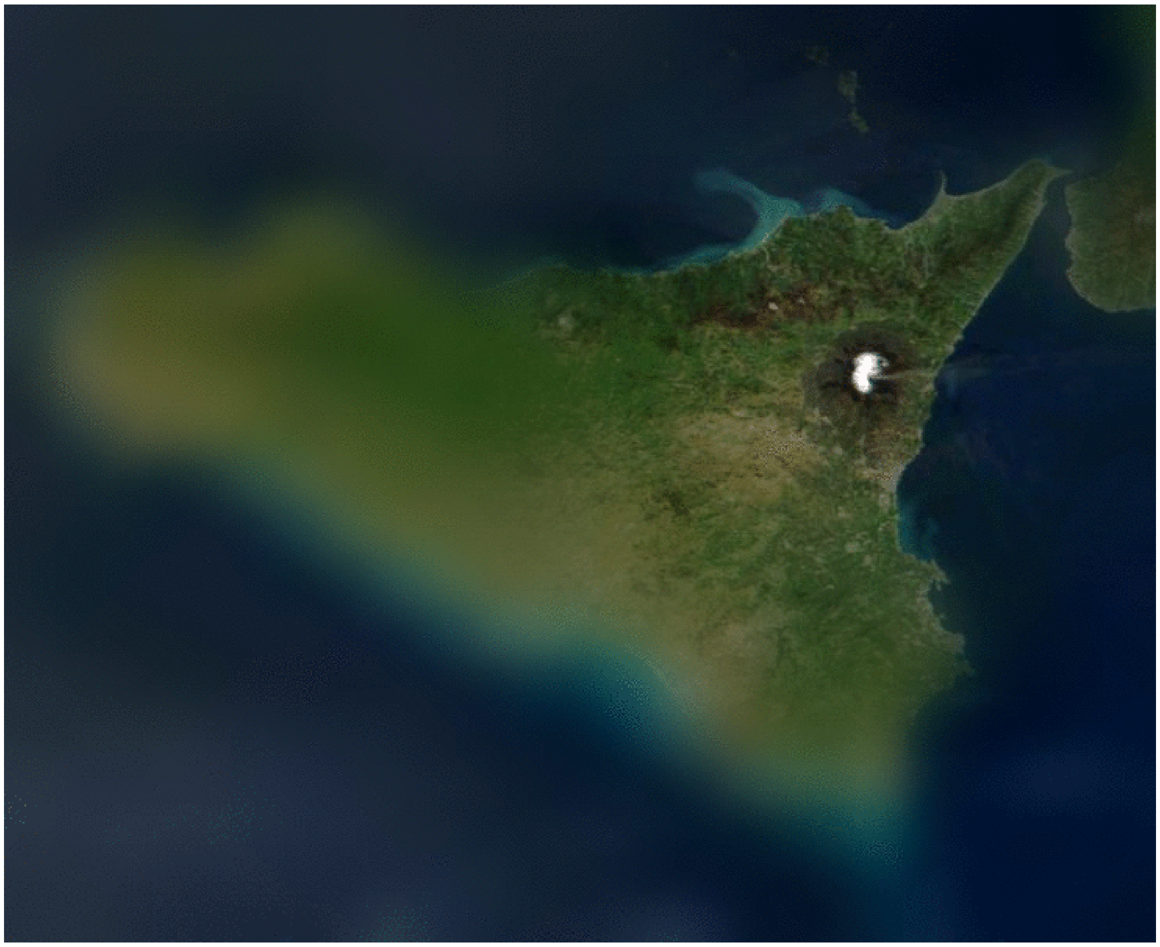
Some common applications of foveated imaging include imaging sensor hardware and image compression.

Foveated imaging is also commonly referred to as space variant imaging or gaze contingent imaging.

Applications


me pre-existing form, would probably first
ries, however slightly, in bodily structure
d if so, whether the variations are transmi
accordance with the laws which prevail wit
ain, are the variations the result, as far
rmits us to judge, of the same general caus
the same general laws, as in the case of o
[illegible] correlation, the inherited effec
[illegible] bject to similar malconformatio
[illegible] ment, of reduplication of par
[illegible] f his anomalies reversion to
[illegible] It might also naturally be
[illegible]
ch other [illegible]
her points [illegible]
The enquirer [illegible]
nds to increase at so rapid a [illegible]
vere struggles for existence, and consequen
riations, whether in body or mind, being pr

Reading task with simulated scotoma



Foveated imaging for progressive transmission

Compression

Contrast sensitivity falls off dramatically as one moves from the center of the retina to the periphery. One may take advantage of this fact in order to compactly code images. If one knows the viewer's approximate point of gaze, one may reduce the amount of information contained in the image as the distance from the point of gaze increases. Because the fall-off in the eye's resolution is dramatic, the potential reduction in display information can be substantial. Also, foveation encoding may be applied to the image before other types of image compression are applied and therefore can result in a multiplicative reduction.

Foveated sensors

Foveated sensors are multiresolution hardware devices that allow image data to be collected with higher resolution concentrated at a fixation point. An advantage to using foveated sensor hardware is that the image collection and encoding can occur much faster than in a system that post-processes a high resolution image in software.

Simulation

Foveated imaging has been used to simulate visual fields with arbitrary spatial resolution. For example, one may present video containing a blurred region representing a scotoma. By utilizing an eye-tracker and holding the blurred region fixed relative to the viewer's gaze, the viewer will have a visual experience similar to that of a person with an actual scotoma. The figure on the right shows a frame from a simulation of a glaucoma patient with the eye fixated on the word "similar."

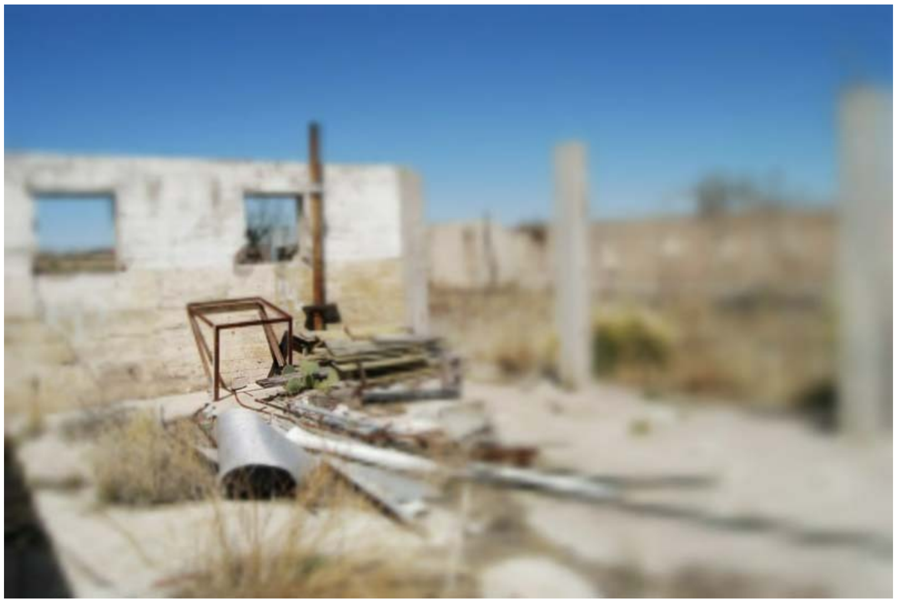
Quality assessment

Foveated imaging may be useful in providing a subjective image quality measure. Traditional image quality measures, such as peak signal-to-noise ratio, are typically performed on fixed resolution images and do not take into account some aspects of the human visual system, like the change in spatial resolution across the retina. A foveated quality index may therefore more accurately determine image quality as perceived by humans.

Image database retrieval

In databases that contain very high resolution images, such as a satellite image database, it may be desirable to interactively retrieve images in order to reduce retrieval time. Foveated imaging allows one to scan low resolution images and retrieve only high resolution portions as they are needed. This is sometimes called progressive transmission.

Example images



Foveated image with fixation selected using an entropy minimization algorithm

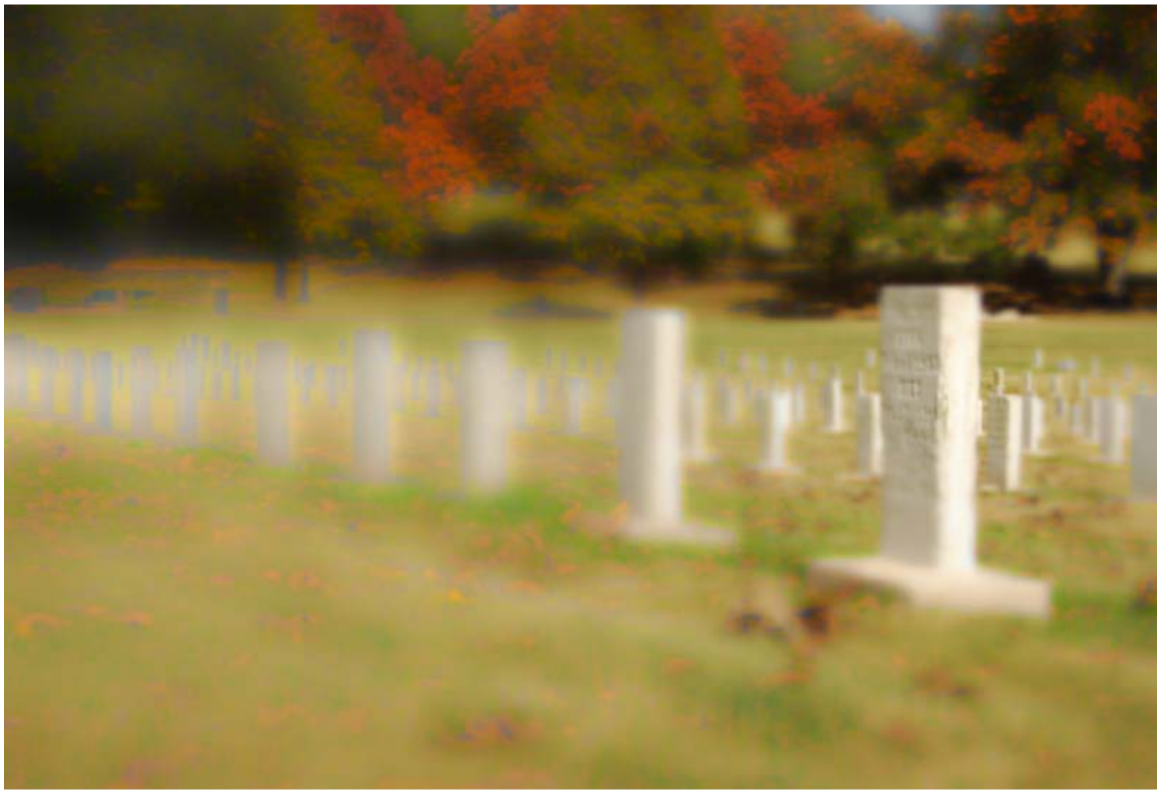
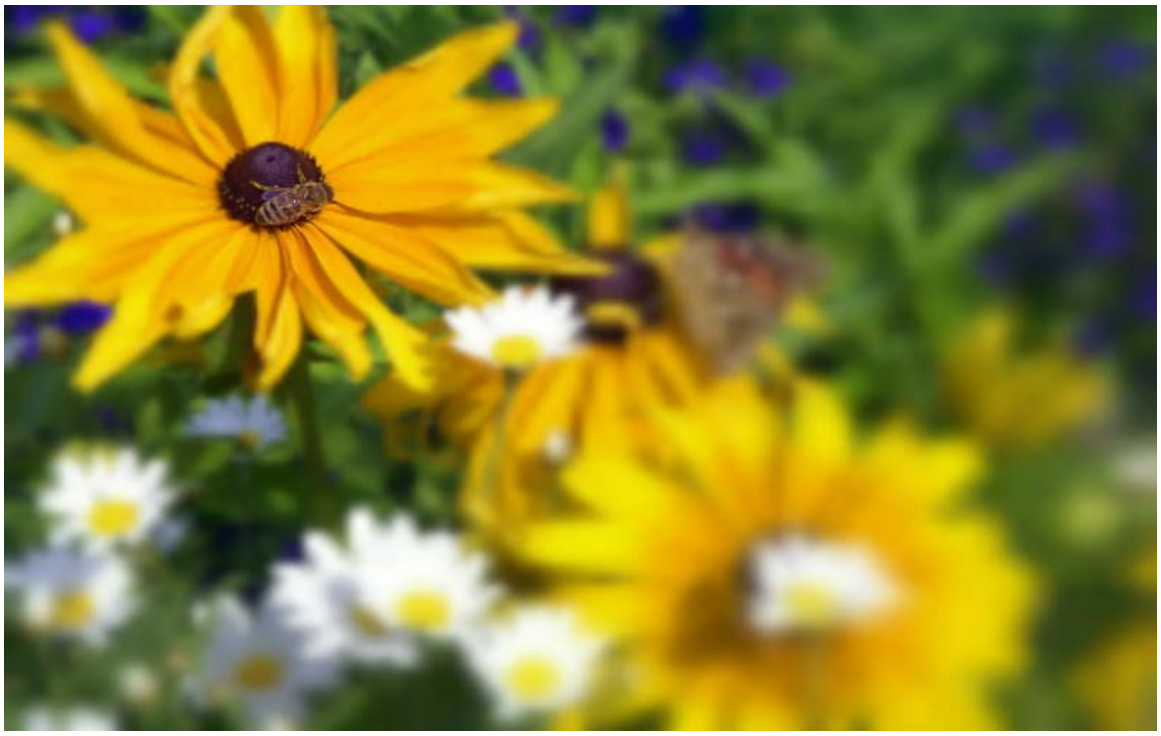
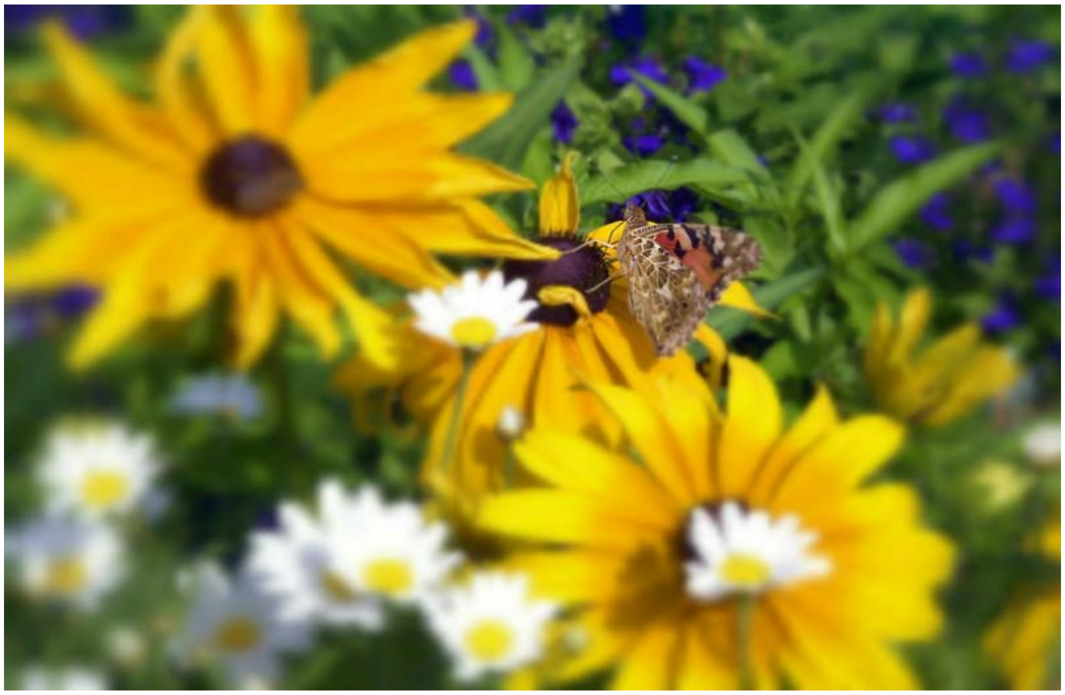


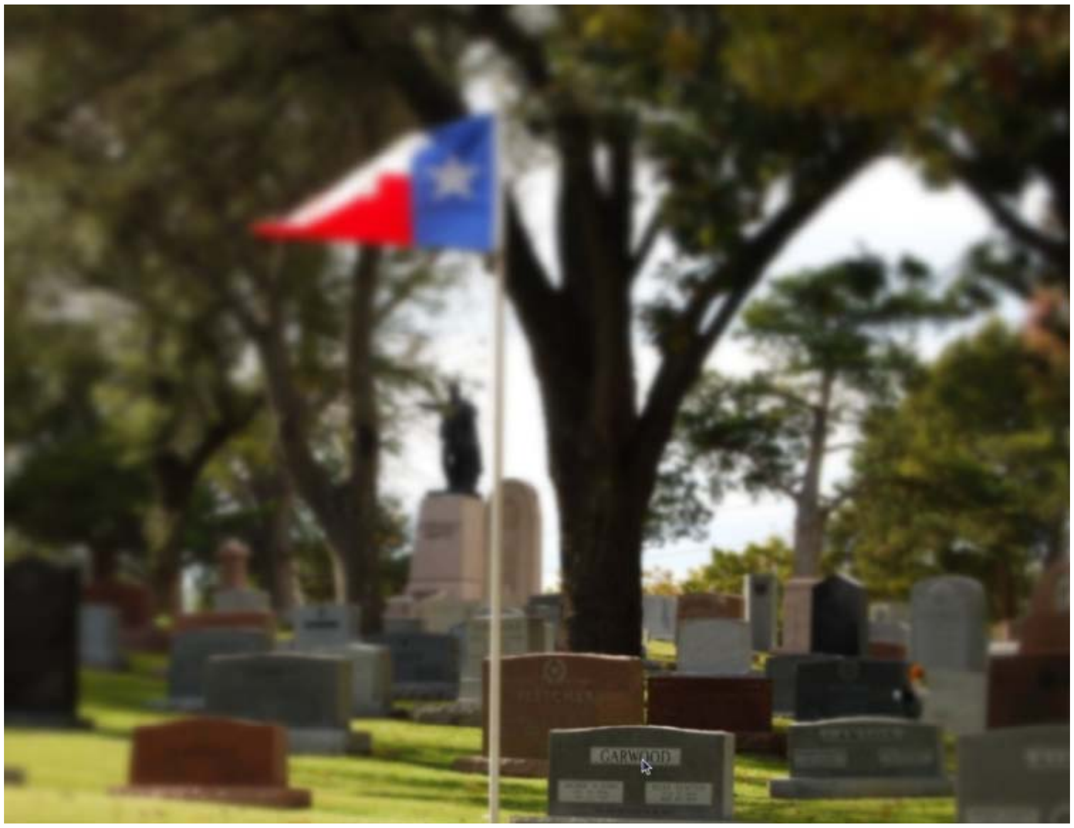
Image with foveated luminance channel



Foveated image with fixation on bee



Same foveated image with fixation on butterfly



18:1 compression. Foveated image with fixation point on tombstone

Chapter 13

Digital Image Processing, Drizzle and EasyHDR Pro

Digital image processing

Digital image processing is the use of computer algorithms to perform image processing on digital images. As a subcategory or field of digital signal processing, digital image processing has many advantages over analog image processing. It allows a much wider range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and signal distortion during processing. Since images are defined over two dimensions (perhaps more) digital image processing may be modeled in the form of Multidimensional Systems.

History

Many of the techniques of digital image processing, or digital picture processing as it often was called, were developed in the 1960s at the Jet Propulsion Laboratory, Massachusetts Institute of Technology, Bell Laboratories, University of Maryland, and a few other research facilities, with application to satellite imagery, wire-photo standards conversion, medical imaging, videophone, character recognition, and photograph enhancement. The cost of processing was fairly high, however, with the computing equipment of that era. That changed in the 1970s, when digital image processing proliferated as cheaper computers and dedicated hardware became available. Images then could be processed in real time, for some dedicated problems such as television standards conversion. As general-purpose computers became faster, they started to take over the role of dedicated hardware for all but the most specialized and computer-intensive operations.

With the fast computers and signal processors available in the 2000s, digital image processing has become the most common form of image processing and generally, is used because it is not only the most versatile method, but also the cheapest.

Digital image processing technology for medical applications was inducted into the Space Foundation Space Technology Hall of Fame in 1994.

Tasks

Digital image processing allows the use of much more complex algorithms for image processing, and hence, can offer both more sophisticated performance at simple tasks, and the implementation of methods which would be impossible by analog means.

In particular, digital image processing is the only practical technology for:

- Classification
- Feature extraction
- Pattern recognition
- Projection
- Multi-scale signal analysis

Some techniques which are used in digital image processing include:

- Pixelization
- Linear filtering
- Principal components analysis
- Independent component analysis
- Hidden Markov models
- Partial differential equations
- Self-organizing maps
- Neural networks
- Wavelets

Applications

Digital camera images

Digital cameras generally include dedicated digital image processing chips to convert the raw data from the image sensor into a color-corrected image in a standard image file format. Images from digital cameras often receive further processing to improve their quality, a distinct advantage that digital cameras have over film cameras. The digital image processing typically is executed by special software programs that can manipulate the images in many ways.

Many digital cameras also enable viewing of histograms of images, as an aid for the photographer to understand the rendered brightness range of each shot more readily.

Film

Westworld (1973) was the first feature film to use digital image processing to pixellate photography to simulate an android's point of view.

Intelligent Transportation Systems

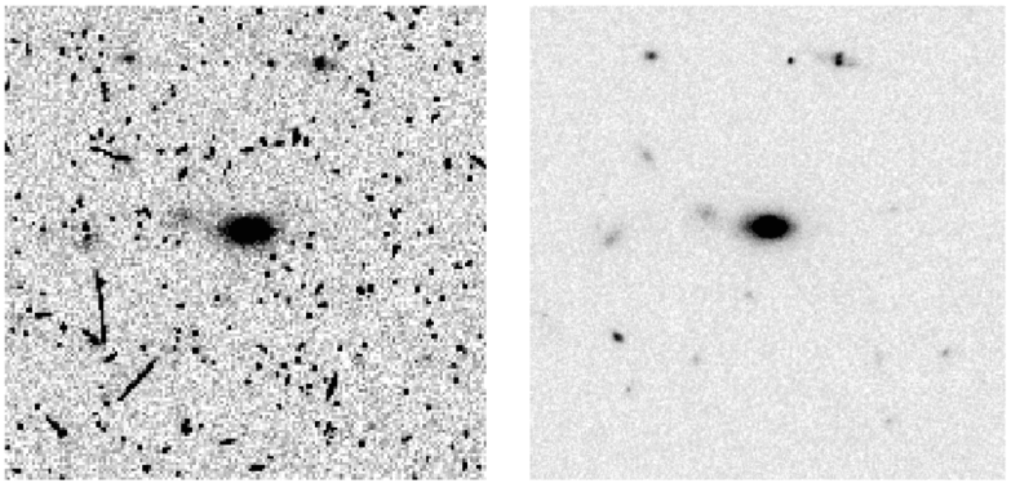
Digital image processing has a wide applications in intelligent transportation systems, such as Automatic number plate recognition and Traffic sign recognition.

Drizzle (image processing)

Drizzle (or **DRIZZLE**) is a digital image processing method for the linear reconstruction of undersampled images. It is normally used for the combination of astronomical images and was originally developed for the Hubble Deep Field observations made by the Hubble Space Telescope. The algorithm, known as Variable-Pixel Linear Reconstruction, or informally as "Drizzle," preserves photometry and resolution, can weight input images according to the statistical significance of each pixel, and removes the effects of geometric distortion on both image shape and photometry. In addition, it is possible to use drizzling to combine dithered images in the presence of cosmic rays.

According to astrophotographer David Ratledge, "Results using the DRIZZLE command can be spectacular with amateur instruments."

Overview



On the left a single 2400s F814W WF2 image taken from the HST archive. On the right, the drizzled combination of twelve such images, each taken at a different dither position.

Camera optics generally introduce geometric distortion of images. Undersampled images are, for example, common in astronomy because instrument designers are frequently forced to choose between properly sampling a small field of view and undersampling a larger field. This is a particular problem for the Hubble Space Telescope (HST), where the corrected optics may provide superb resolution, but the detectors are only able to take full advantage of the full resolving power of the telescope over a limited field of view. Fortunately, much of the information lost to undersampling can be restored. The most commonly used of these techniques are shift-and-add and interlacing.

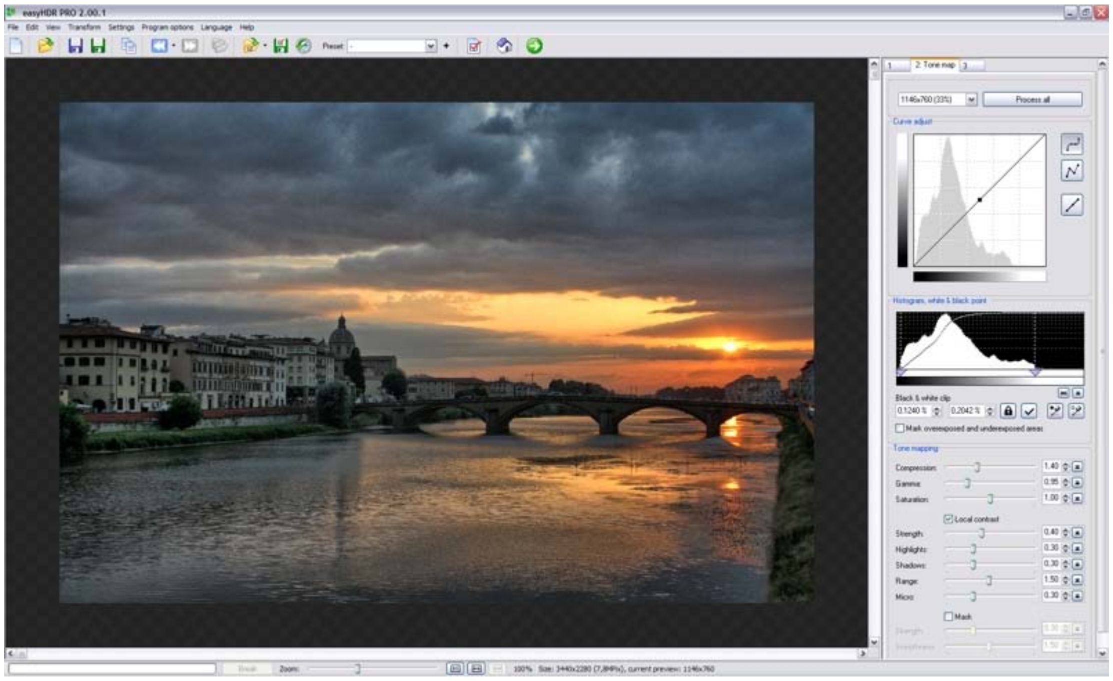
Drizzle was originally developed to combine the dithered images of the Hubble Deep Field North and has since been widely used for the combination of dithered images from both HST's cameras and those on other telescopes. Drizzle has the versatility of shift-and-add yet largely maintains the resolution and independent noise statistics of interlacing. Drizzle has the advantage of being able to handle images with essentially arbitrary shifts, rotations, and geometric distortion and, when given input images with proper associated weight maps, creates an optimal statistically summed image. Drizzle also naturally handles images with "missing" data, due, for instance, to corruption by cosmic rays or detector defects.

Drizzle is freely available as an IRAF task as part of the Space Telescope Science Data Analysis System (STSDAS) package and can be retrieved from the Space Telescope Science Institute (STScI) web site. In addition to Drizzle, a number of ancillary tasks that assist in the combination of Hubble Space Telescope imaging data are available as part of the "dither" package in STSDAS.

Drizzle was developed as a collaboration between the Space Telescope Science Institute and the Space Telescope European Coordinating Facility.

EasyHDR Pro

easyHDR PRO


easyHDR PRO 2.00.1
File Edit View Transform Settings Program options Language Help
Preset
1 2: Tone map 3
Process all
Curve adjust
Histogram, white & black point
Tone mapping
Zoom:

1146x760 (33%)

Black & white clip: 0.1240 % 0.2042 %

☐ Mark overexposed and underexposed areas

Compression: 1.40
Gamma: 0.95
Saturation: 1.00

☑ Local contrast

Strength: 0.40
Highlights: 0.30
Shadows: 0.30
Range: 1.50
Micro: 0.30

☐ Mask

100% Size: 3440x2280 (7,8MPx), current preview: 1146x760

easyHDR PRO 2 screenshot

Developer(s)	SIMPARTEK (Bartlomiej Okonek)
Stable release	2.01.1 / Aug 2010
Operating system	Microsoft Windows
Type	High dynamic range imaging
License	Proprietary

EasyHDR PRO is an image processing software that produces and tone maps High Dynamic Range (HDR) images out of photo sequences taken with a digital camera. It is also capable of processing just single photos in order to enhance contrasts and colors. The program can import a number of image file types: JPEG, 24/48/96-bit TIFF and many RAW image formats, thanks to using DCRAW decoder..

The very first version of easyHDR was made available on February 3, 2006 and since then it was constantly under development. On 8 May 2010 the new major version was released - easyHDR PRO 2.00.1, which introduced a completely redesigned tone mapping algorithm.

Generating HDR image

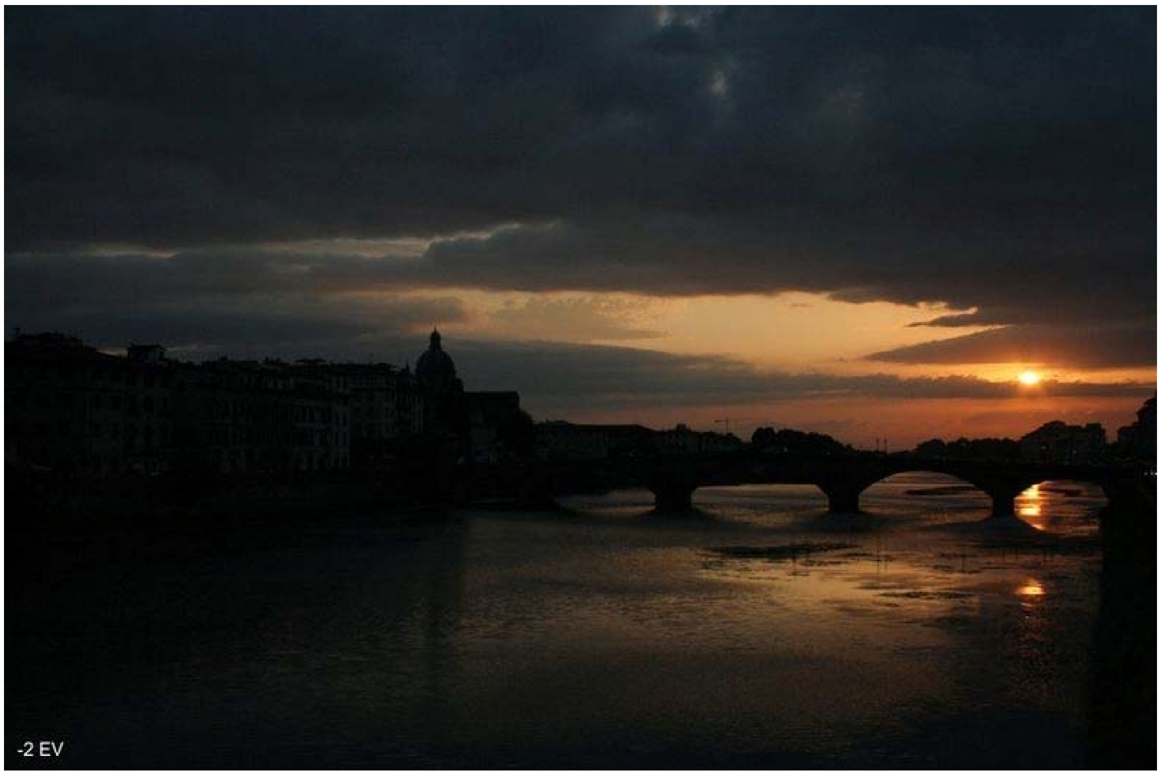
Before a High Dynamic Range image is generated it is possible, with easyHDR PRO, to apply manual, or automatic alignment. It may be necessary in case of processing photos that were taken without a tripod. The program features also a chromatic aberration correction tool.

When the sequence of photos is aligned and optionally also the chromatic aberration is corrected, the HDR image can be generated. There are 3 methods available:

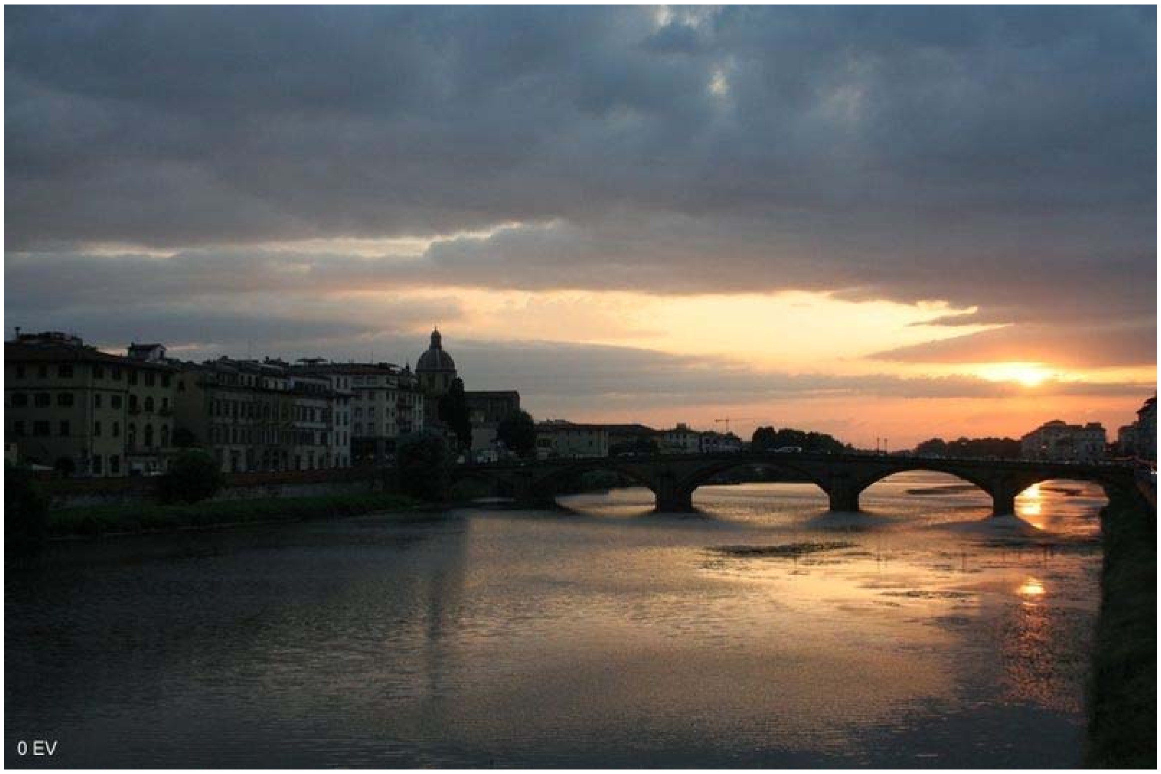
- True HDR - a scientifically correct method of generating HDR radiance maps. Ensures that the result photo looks realistically.
- Smart Merge - merges the photographs, by taking the best exposed areas of each photo. This is a "pseudo-HDR" technique.
- Stacking - create an average of photos - useful when merging photos taken with the same exposure settings in order to reduce noise level.

Tone mapping

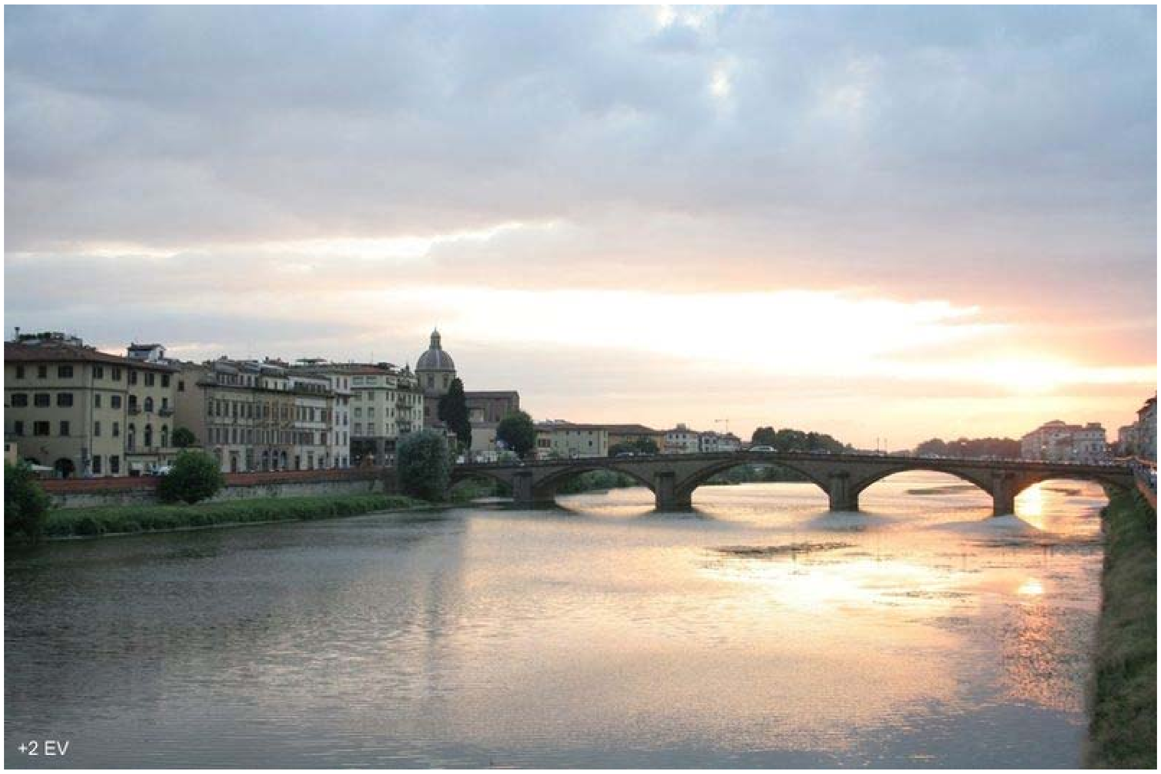
The generated HDR image has so wide dynamic range that it cannot be properly printed or displayed on a computer screen. The dynamic range has to be compressed, but the compression has to be done on local level so that the local contrasts are preserved. Local tone mapping operators take each pixel's neighborhood into consideration while calculating the spatially variant parameters for contrast and brightness modification . EasyHDR PRO 2 features two local tone mapping operators: "Local Contrast" and "Mask". Each of them can be enabled/disabled independently and each has its own set of parameters. The user modifies the behavior of the operators with several sliders that control the overall strength of the effect, range of the neighborhood, smoothness, as well as selective strength in shadows and highlights.



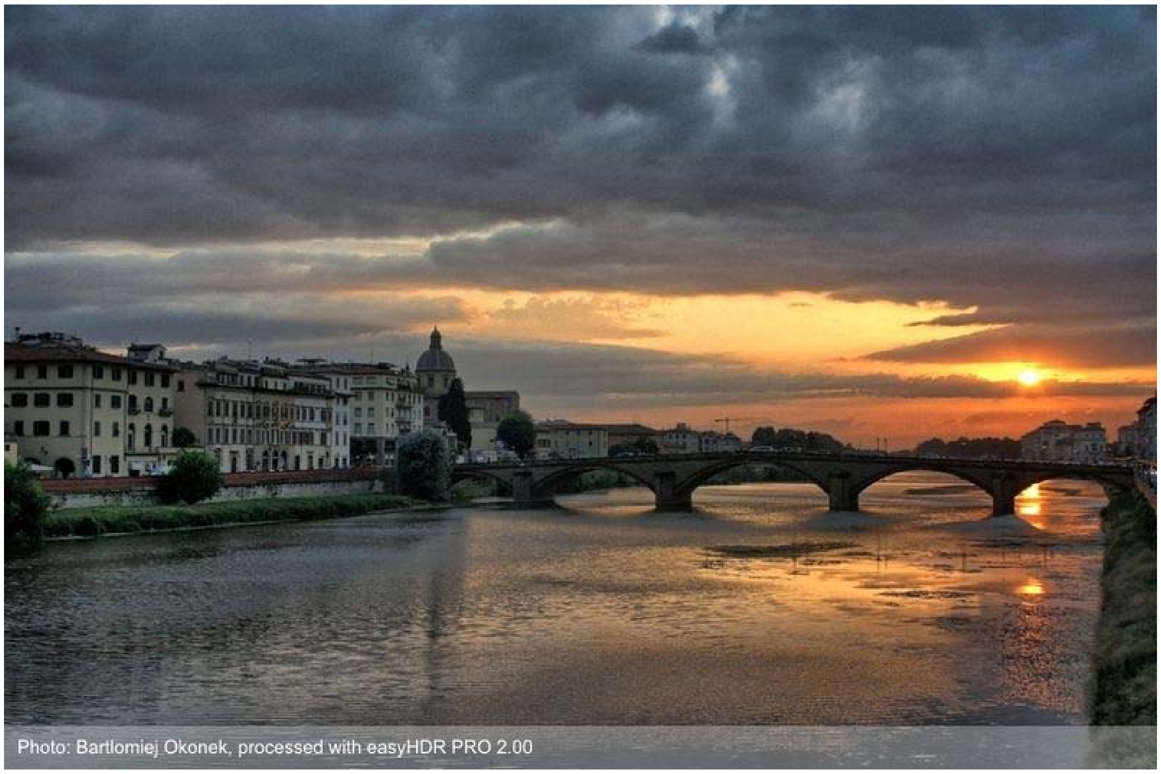
-2 stops



0 stops



+2 stops



Result

Input photo

Result

Post processing

EasyHDR PRO allows some further processing of the tone mapped photos. There are several filters and transformations available:

- Gaussian blur,
- Sharpening,
- Median filter - for automatic filtering out salt and pepper like noise,
- Bilateral filter - for de-noising smoothing that preserves edges,
- White balance,
- Sample/target balance - for advanced color modification,

- Color tone,
- Rotate and mirror,
- Crop.

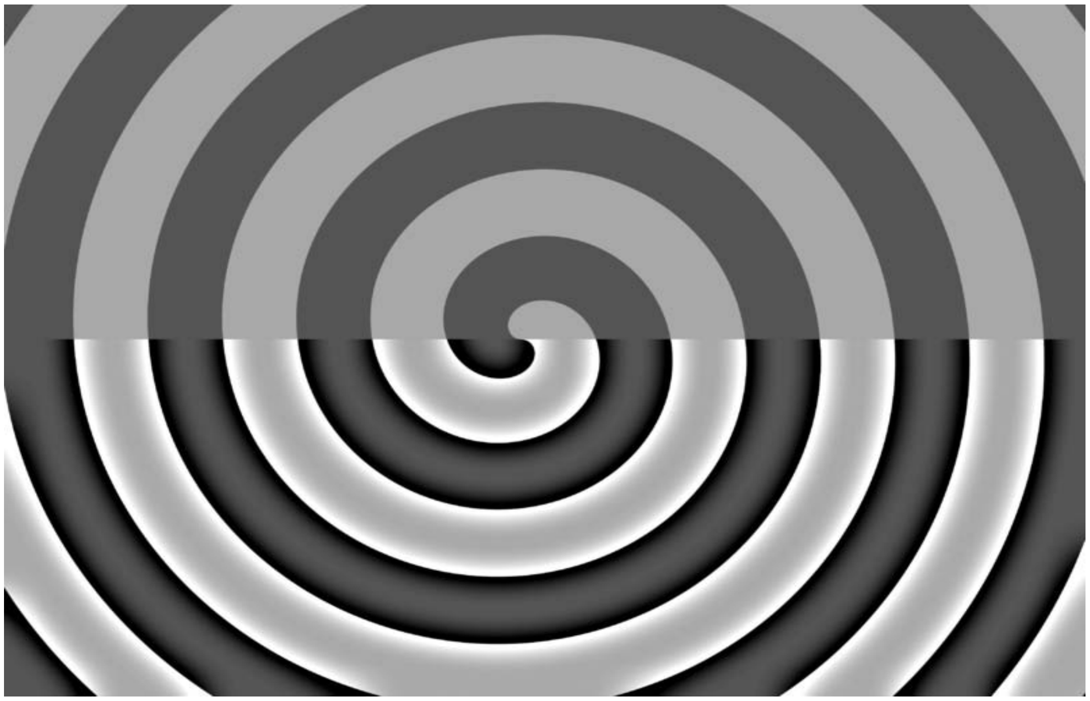
Color management

When the color management support is enabled (option in the main menu), the program is aware of the input photo color space and it uses the provided monitor color profile to properly transform the image colors. EasyHDR PRO uses LittleCMS as the color management engine.

Chapter 14

Edge Enhancement and Edge Detection

Edge enhancement



Unsharp masking has been applied to lower part of image, creating overshoot and undershoot and increasing acutance.

Edge enhancement is an image processing filter that enhances the edge contrast of an image or video in an attempt to improve its acutance (apparent sharpness).

The filter works by identifying sharp edge boundaries in the image, such as the edge between a subject and a background of a contrasting color, and increasing the image contrast in the area immediately around the edge. This has the effect of creating subtle bright and dark highlights on either side of any edges in the image, called overshoot and undershoot, leading the edge to look more defined when viewed from a typical viewing distance.

The process is prevalent in the video field, appearing to some degree in the majority of TV broadcasts and DVDs. A modern television set's "sharpness" control is an example of edge enhancement. It is also widely used in computer printers especially for font or/and graphics to get a better printing quality. Most digital cameras also perform some edge enhancement, which in some cases cannot be adjusted.

Edge enhancement can be either an analog or a digital process. Analog edge enhancement may be used, for example, in all-analog video equipment such as modern CRT televisions.

Properties

Edge enhancement applied to an image can vary according to a number of properties; the most common algorithm is unsharp masking, which has the following parameters:

- *Amount*. This controls the extent to which contrast in the edge detected area is enhanced.
- *Radius* or *aperture*. This affects the size of the edges to be detected or enhanced, and the size of the area surrounding the edge that will be altered by the enhancement. A smaller radius will result in enhancement being applied only to sharper, finer edges, and the enhancement being confined to a smaller area around the edge.
- *Threshold*. Where available, this adjusts the sensitivity of the edge detection mechanism. A lower threshold results in more subtle boundaries of colour being identified as edges. A threshold that is too low may result in some small parts of surface textures, film grain or noise being incorrectly identified as being an edge.

In some cases, edge enhancement can be applied in the horizontal or vertical direction only, or to both directions in different amounts. This may be useful, for example, when applying edge enhancement to images that were originally sourced from analog video.

Effects of edge enhancement

Unlike some forms of image sharpening, edge enhancement does not enhance subtle detail which may appear in more uniform areas of the image, such as texture or grain which appears in flat or smooth areas of the image. The benefit to this is that imperfections in the image reproduction, such as grain or noise, or imperfections in the subject, such as natural imperfections on a person's skin, are not made more obvious by the process. A drawback to this is that the image may begin to look less natural, because the apparent sharpness of the overall image has increased but the level of detail in flat, smooth areas has not.

As with other forms of image sharpening, edge enhancement is only capable of improving the *perceived* sharpness or acutance of an image. The enhancement is not completely reversible, and as such some detail in the image is lost as a result of filtering. Further sharpening operations on the resulting image compound the loss of detail, leading to artifacts such as ringing. An example of this can be seen when an image that has already had edge enhancement applied, such as the picture on a DVD video, has further edge enhancement applied by the DVD player it is played on, and possibly also by the television it is displayed on. Essentially, the first edge enhancement filter creates new edges on either side of the existing edges, which are then further enhanced.

Viewing conditions

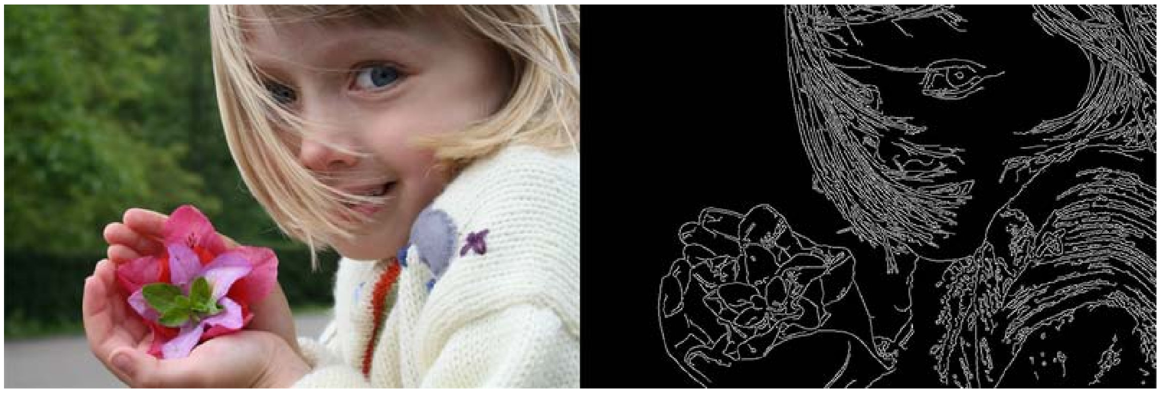
The ideal amount of edge enhancement that is required to produce a pleasant and sharp-looking image, without losing too much detail, varies according to several factors. An image that is to be viewed from a nearer distance, at a larger display size, on a medium that is inherently more "sharp" or by a person with excellent eyesight will typically demand a finer or lesser amount of edge enhancement than an image that is to be shown at a smaller display size, further viewing distance, on a medium that is inherently softer or by a person with poorer eyesight.

For this reason, home theatre enthusiasts who invest in larger, higher quality screens often complain about the amount of edge enhancement present in commercially produced DVD videos, claiming that such edge enhancement is optimized for playback on smaller, poorer quality television screens, but the loss of detail as a result of the edge enhancement is much more noticeable in their viewing conditions.

Edge detection

Edge detection is a fundamental tool in image processing and computer vision, particularly in the areas of feature detection and feature extraction, which aim at identifying points in a digital image at which the image brightness changes sharply or more formally has discontinuities.

Motivations



Canny edge detection applied to a photograph

The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. It can be shown that under rather general assumptions for an image formation model, discontinuities in image brightness are likely to correspond to:

- discontinuities in depth,
- discontinuities in surface orientation,
- changes in material properties and
- variations in scene illumination.

In the ideal case, the result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings as well as curves that correspond to discontinuities in surface orientation. Thus, applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image. If the edge detection step is successful, the subsequent task of interpreting the information contents in the original image may therefore be substantially simplified. However, it is not always possible to obtain such ideal edges from real life images of moderate complexity. Edges extracted from non-trivial images are often hampered by *fragmentation*, meaning that the edge curves are not connected, missing edge segments as well as *false edges* not corresponding to interesting phenomena in the image – thus complicating the subsequent task of interpreting the image data.

Edge detection is one of the fundamental steps in image processing, image analysis, image pattern recognition, and computer vision techniques. During recent years, however, substantial (and successful) research has also been made on computer vision methods that do not explicitly rely on edge detection as a pre-processing step.

Edge properties

The edges extracted from a two-dimensional image of a three-dimensional scene can be classified as either viewpoint dependent or viewpoint independent. A *viewpoint independent edge* typically reflects inherent properties of the three-dimensional objects, such as surface markings and surface shape. A *viewpoint dependent edge* may change as the viewpoint changes, and typically reflects the geometry of the scene, such as objects occluding one another.

A typical edge might for instance be the border between a block of red color and a block of yellow. In contrast a **line** (as can be extracted by a ridge detector) can be a small number of pixels of a different color on an otherwise unchanging background. For a line, there may therefore usually be one edge on each side of the line.

A simple edge model

Although certain literature has considered the detection of ideal step edges, the edges obtained from natural images are usually not at all ideal step edges. Instead they are normally affected by one or several of the following effects:

- focal blur caused by a finite depth-of-field and finite point spread function.
- penumbral blur caused by shadows created by light sources of non-zero radius.
- shading at a smooth object

A number of researchers have used a Gaussian smoothed step edge (an error function) as the simplest extension of the ideal step edge model for modeling the effects of edge blur in practical applications. Thus, a one-dimensional image *f* which has exactly one edge placed at *x* = 0 may be modeled as:

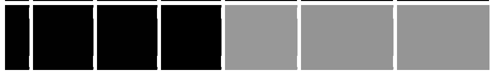
$$f(x) = \frac{I_r - I_l}{2}\left(\operatorname{erf}\left(\frac{x}{\sqrt{2}\sigma}\right) + 1\right) + I_l.$$

At the left side of the edge, the intensity is $I_l = \lim_{x\to-\infty} f(x)$, and right of the edge it is $I_r = \lim_{x\to\infty} f(x)$. The scale parameter σ is called the blur scale of the edge.

Why edge detection is a non-trivial task

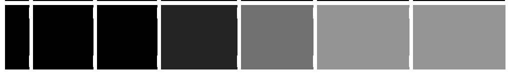
To illustrate why edge detection is not a trivial task, let us consider the problem of detecting edges in the following one-dimensional signal. Here, we may intuitively say that there should be an edge between the 4th and 5th pixels.

5	7	6	4	152	148	149



If the intensity difference were smaller between the 4th and the 5th pixels and if the intensity differences between the adjacent neighboring pixels were higher, it would not be as easy to say that there should be an edge in the corresponding region. Moreover, one could argue that this case is one in which there are several edges.

5	7	6	41	113	148	149



Hence, to firmly state a specific threshold on how large the intensity change between two neighbouring pixels must be for us to say that there should be an edge between these pixels is not always simple. Indeed, this is one of the reasons why edge detection may be a non-trivial problem unless the objects in the scene are particularly simple and the illumination conditions can be well controlled (see for example, the edges extracted from the image with the girl above).

Approaches to edge detection

There are many methods for edge detection, but most of them can be grouped into two categories, search-based and zero-crossing based. The search-based methods detect edges by first computing a measure of edge strength, usually a first-order derivative expression such as the gradient magnitude, and then searching for local directional maxima of the gradient magnitude using a computed estimate of the local orientation of the edge, usually the gradient direction. The zero-crossing based methods search for zero crossings in a second-order derivative expression computed from the image in order to find edges, usually the zero-crossings of the Laplacian or the zero-crossings of a non-linear differential expression. As a pre-processing step to edge detection, a smoothing stage, typically Gaussian smoothing, is almost always applied.

The edge detection methods that have been published mainly differ in the types of smoothing filters that are applied and the way the measures of edge strength are computed. As many edge detection methods rely on the computation of image gradients, they also differ in the types of filters used for computing gradient estimates in the x- and y-directions.

A survey of a number of different edge detection methods can be found in (Ziou and Tabbone 1998).

Canny edge detection

John Canny considered the mathematical problem of deriving an optimal smoothing filter given the criteria of detection, localization and minimizing multiple responses to a single edge. He showed that the optimal filter given these assumptions is a sum of four exponential terms. He also showed that this filter can be well approximated by first-order derivatives of Gaussians. Canny also introduced the notion of non-maximum suppression, which means that given the presmoothing filters, edge points are defined as points where the gradient magnitude assumes a local maximum in the gradient direction. Looking for the zero crossing of the 2nd derivative along the gradient direction was first proposed by Haralick . It took less than two decades to find a modern geometric variational meaning for that operator that links it to the Marr-Hildreth (zero crossing of the Laplacian) edge detector. That observation was presented by Ron Kimmel and Alfred Bruckstein.

Although his work was done in the early days of computer vision, the Canny edge detector (including its variations) is still a state-of-the-art edge detector. Unless the preconditions are particularly suitable, it is hard to find an edge detector that performs significantly better than the Canny edge detector.

The Canny-Deriche detector was derived from similar mathematical criteria as the Canny edge detector, although starting from a discrete viewpoint and then leading to a set of recursive filters for image smoothing instead of exponential filters or Gaussian filters.

The differential edge detector described below can be seen as a reformulation of Canny's method from the viewpoint of differential invariants computed from a scale-space representation leading to a number of advantages in terms of both theoretical analysis and sub-pixel implementation.

Other first-order methods

For estimating image gradients from the input image or a smoothed version of it, different gradient operators can be applied. The simplest approach is to use central differences:

$$\begin{aligned} L_x(x,y) &= -1/2 \cdot L(x-1,y) + 0 \cdot L(x,y) + 1/2 \cdot L(x+1,y) \\ L_y(x,y) &= -1/2 \cdot L(x,y-1) + 0 \cdot L(x,y) + 1/2 \cdot L(x,y+1), \end{aligned}$$

corresponding to the application of the following filter masks to the image data:

$$L_x = \begin{bmatrix} -1/2 & 0 & 1/2 \end{bmatrix} * L \quad \text{and} \quad L_y = \begin{bmatrix} +1/2 \\ 0 \\ -1/2 \end{bmatrix} * L.$$

The well-known and earlier Sobel operator is based on the following filters:

$$L_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * L \quad \text{and} \quad L_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * L.$$

Given such estimates of first- order derivatives, the gradient magnitude is then computed as:

$$|\nabla L| = \sqrt{L_x^2 + L_y^2}$$

while the gradient orientation can be estimated as

$$\theta = \operatorname{atan2}(L_y, L_x).$$

Other first-order difference operators for estimating image gradient have been proposed in the Prewitt operator and Roberts cross.

Thresholding and linking

Once we have computed a measure of edge strength (typically the gradient magnitude), the next stage is to apply a threshold, to decide whether edges are present or not at an image point. The lower the threshold, the more edges will be detected, and the result will be increasingly susceptible to noise and detecting edges of irrelevant features in the image. Conversely a high threshold may miss subtle edges, or result in fragmented edges.

If the edge thresholding is applied to just the gradient magnitude image, the resulting edges will in general be thick and some type of edge thinning post-processing is necessary. For edges detected with non-maximum suppression however, the edge curves are thin by definition and the edge pixels can be linked into edge polygon by an edge linking (edge tracking) procedure. On a discrete grid, the non-maximum suppression stage can be implemented by estimating the gradient direction using first-order derivatives, then rounding off the gradient direction to multiples of 45 degrees, and finally comparing the values of the gradient magnitude in the estimated gradient direction.

A commonly used approach to handle the problem of appropriate thresholds for thresholding is by using thresholding with hysteresis. This method uses multiple thresholds to find edges. We begin by using the upper threshold to find the start of an edge. Once we have a start point, we then trace the path of the edge through the image pixel by pixel, marking an edge whenever we are above the lower threshold. We stop marking our edge only when the value falls below our lower threshold. This approach makes the assumption that edges are likely to be in continuous curves, and allows us to follow a faint section of an edge we have previously seen, without meaning that every noisy pixel in the image is marked down as an edge. Still, however, we have the problem of choosing appropriate thresholding parameters, and suitable thresholding values may vary over the image.

Edge Thinning

Edge thinning is a technique used to remove the unwanted spurious points on the edge of an image. This technique is employed after the image has been filtered for noise (using median, Gaussian filter etc.), the edge operator has been applied (like the ones described above) to detect the edges and after the edges have been smoothed using an appropriate threshold value. This removes all the unwanted points and if applied carefully, results in one pixel thick edge elements.

Advantages: 1) Sharp and thin edges lead to greater efficiency in object recognition. 2) If you are using Hough transforms to detect lines and ellipses then thinning could give much better results. 3) If the edge happens to be boundary of a region then, thinning could easily give the image parameters like perimeter without much algebra.

There are many popular algorithms used to do this, one such is described below:

1) Choose a type of connectivity, like 8, 6 or 4.

2) 8 connectivity is preferred, where all the immediate pixels surrounding a particular pixel are considered.

3) Remove points from North, south, east and west.

4) Do this in multiple passes, i.e. after the north pass, use the same semi processed image in the other passes and so on.

5) Remove a point if:

```
    The point has no neighbors in the North (if you are in the north
pass, and
         respective directions for other passes.)
    The point is not the end of a line.
    The point is isolated.
    Removing the points will not cause to disconnect its neighbors in
any way.
```

6) Else keep the point. The number of passes across direction should be chosen according to the level of accuracy desired.

Second-order approaches to edge detection

Some edge-detection operators are instead based upon second-order derivatives of the intensity. This essentially captures the rate of change in the intensity gradient. Thus, in the ideal continuous case, detection of zero-crossings in the second derivative captures local maxima in the gradient.

The early Marr-Hildreth operator is based on the detection of zero-crossings of the Laplacian operator applied to a Gaussian-smoothed image. It can be shown, however, that this operator will also return false edges corresponding to local minima of the gradient magnitude. Moreover, this operator will give poor localization at curved edges. Hence, this operator is today mainly of historical interest.

Differential edge detection

A more refined second-order edge detection approach which automatically detects edges with sub-pixel accuracy, uses the following *differential approach* of detecting zero-crossings of the second-order directional derivative in the gradient direction:

Following the differential geometric way of expressing the requirement of non-maximum suppression proposed by Lindeberg, let us introduce at every image point a local coordinate system (u,v), with the v-direction parallel to the gradient direction. Assuming that the image has been presmoothed by Gaussian smoothing and a scale-space representation $L(x,y;t)$ at scale t has been computed, we can require that the gradient magnitude of the scale-space representation, which is equal to the first-order directional derivative in the *v*-direction L_v, should have its first order directional derivative in the *v*-direction equal to zero

$$\partial_v(L_v) = 0$$

while the second-order directional derivative in the *v*-direction of L_v should be negative, i.e.,

$$\partial_{vv}(L_v) \leq 0.$$

Written out as an explicit expression in terms of local partial derivatives L_x, L_y ... L_{yyy}, this edge definition can be expressed as the zero-crossing curves of the differential invariant

$$L_v^2 L_{vv} = L_x^2\, L_{xx} + 2\, L_x\, L_y\, L_{xy} + L_y^2\, L_{yy} = 0,$$

that satisfy a sign-condition on the following differential invariant

$$L_v^3 L_{vvv} = L_x^3\, L_{xxx} + 3\, L_x^2\, L_y\, L_{xxy} + 3\, L_x\, L_y^2\, L_{xyy} + L_y^3\, L_{yyy} \leq 0$$

where L_x, L_y ... L_{yyy} denote partial derivatives computed from a scale-space representation *L* obtained by smoothing the original image with a Gaussian kernel. In this way, the edges will be automatically obtained as continuous curves with subpixel accuracy. Hysteresis thresholding can also be applied to these differential and subpixel edge segments.

In practice, first-order derivative approximations can be computed by central differences as described above, while second-order derivatives can be computed from the scale-space representation *L* according to:

$$L_{xx}(x,y) = L(x-1,y) - 2L(x,y) + L(x+1,y).$$
$$L_{xy}(x,y) = (L(x-1,y-1) - L(x-1,y+1) - L(x+1,y-1) + L(x+1,y+1))/4,$$
$$L_{yy}(x,y) = L(x,y-1) - 2L(x,y) + L(x,y+1).$$

corresponding to the following filter masks:

$$L_{xx} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} * L \quad \text{and} \quad L_{xy} = \begin{bmatrix} -1/4 & 0 & 1/4 \\ 0 & 0 & 0 \\ 1/4 & 0 & -1/4 \end{bmatrix} * L \quad \text{and} \quad L_{yy} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} * L.$$

Higher-order derivatives for the third-order sign condition can be obtained in an analogous fashion.

Phase congruency based edge detection

A recent development in edge detection techniques takes a frequency domain approach to finding edge locations. Phase congruency (also known as phase coherence) methods attempt to find locations in an image where all sinusoids in the frequency domain are in phase. These locations will generally correspond to the location of a perceived edge, regardless of whether the edge is represented by a large change in intensity in the spatial domain. A key benefit of this technique is that it responds strongly to Mach bands, and avoids false positives typically found around roof edges. A roof edge, is a discontinuity in the first order derivative of a grey-level profile.