MATH0075: Advanced Computer Graphics.

Russell Bradford
(based on notes by Phil Willis)

1 Introduction

This course has the title "Advanced Computer Graphics". Three words, of which a lot of people miss the first two.

This is an *advanced* course, meaning that I expect you to go away and read books to fill in details. As a final year course, I shall be indicating the general areas of interest, but further reading is up to you.

This course uses *computers*. There is programming involved: 25% of the course marks goes on an assignment. The choice of programming language will be up to you (within reason), but the programming itself should be reasonably straightforward. The thought behind the program will be quite challenging, though.

So what is this course about? At the simplest level, graphics is about drawing pretty pictures. But, of course, there is really a lot more to it than that.

Science and Engineering

- Computer Aided Design (CAD), e.g., architecture. Model it before you build it!
- User interfaces. You can tell a good interface by the fact that you don't notice it. A bad UI interferes with your work. Requires psychology to do properly.
- Visualisation. Chemists like to see molecules. Medical applications.
- Simulation. Military models battlefields. Aviation training.

This leads into virtual reality.

Entertainment

- Games. Big money. Difference between models that look OK and are fast, and things that are "physically correct".
- Films. Almost every big film these days use SFX, often to patch over errors. Realistic (Titanic) vs. non-realistic (Toy Story).

Animation vs. still.

The eye is a high bandwidth connection to the brain. Most graphics systems require huge amounts of data to be shifted about very quickly. There is a high performance requirement: (a) real time generation of images, and (b) non-real time (e.g., frames in a film), but time is money.

Thus need to think about clever software to do this. Alternatively, and becoming a popular choice, is to use clever hardware (e.g., graphics cards in PCs).

Also it turns out that a little knowledge of human perception helps a great deal in designing these things. Facts about how the eye and brain work (human physiology), optical illusions, colours, shades.

Much of the above requires extensive models, in particular virtual reality.

- Objects. E.g., spheres, cubes, chairs, tables
- Properties. Colour, reflectivity, scattering (fog)
- Motion. Newtonian physics. internal motion, e.g., jelly, cloth.

In principle we can use Schrödinger's equation to model the world: in practice we approximate!

1.1 Chapter Headings

Order may vary in practice!

1. Output Hardware. Workstations. Displays.
2. Surface modelling. Wire Frames. 3D transformations (matrices).
3. Surface rendering. Hidden surface removal. Lighting. Projections 3D to 2D.
4. Solid Modelling.
5. Ray tracing.
6. Radiosity.
7. Colour. The eye.
8. Graphics Libraries (if time). OpenGL.

1.2 Book

"Introduction to Computer Graphics," by Foley, van Dam, Feiner, Hughes, Phillips. Addison Wesley.

Alternatively, many other books on graphics are OK. Not much to choose between them.

1.3 Problems Class

This will be *student driven*. Probably more useful when we get to the assignment. This will be midway through the semester, and will be ray tracing (unless I get a better idea).

2 Output Hardware

These are the things that you sit and stare at: there are many kinds of output device, but we shall be concentrating on the most popular: the colour graphics workstation.

There are many other types of device

- plotters
- laser, ink jet, thermal etc.
- bus destination indicators (blobs of colour)
- radar screens (long persistence phosphor)
- vector graphics (Battlezone)
- wave LEDs about

2.1 Colour Graphics workstation

The current PC with colour monitor has developed from the workstation concept: an area of memory holds the information to b displayed, and some piece of hardware converts the bits and bytes into the electrical signals needed to power the monitor's electron guns.

The screen is constantly refreshed from the memory. The hardware is dedicated these days: the ZX80 used the main CPU to drive the screen, with the consequence that the machine could be processing, or displaying, but not both!

The picture on the screen is updated simply by changing the contents of the memory. Modern systems use *graphics coprocessors*: a dedicated CPU with its own memory that can do many of the operations required for sophisticated graphics in hardware (e.g., drawing polygons, shading colours or textures). This relieves the main CPU to do other processing.

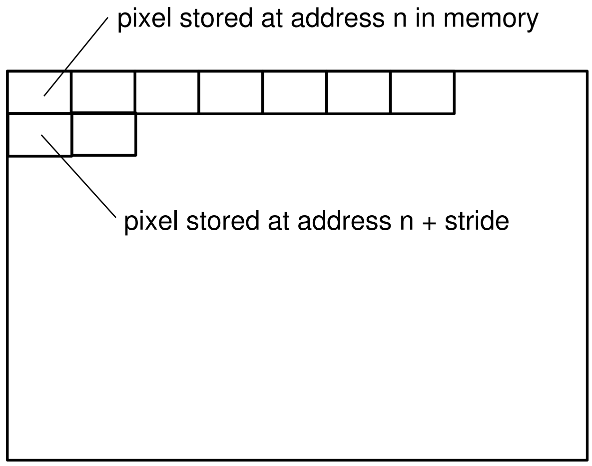
Screens can be black and white, colour, or *greyscale* (popular for CAD). We shall concentrate on colour, but think also of black and white.

2.2 Display Organisation

Each element of colour on the screen is called a *pixel* (or *pel* or *pictel*), and each requires a chunk of memory. This can be one bit per pixel, or up to 32 bits per pixel.

bits/pixel	no. colours	
1	2	black and white
8	256	popular choice for PCs
24	16 million	workstations
32	16 million	high-end workstations: uses alpha channels

We think of screen memory in 2D, but it is really a 1D line: the translation between the 2D coordinates of a pixel and its corresponding memory address can be fiddly: recall a pixel can be 1 bit, or 1 byte, or 3 bytes, or whatever. Sometimes we have bytes per pixel, sometimes pixels per byte (many pixels packed into a single byte).

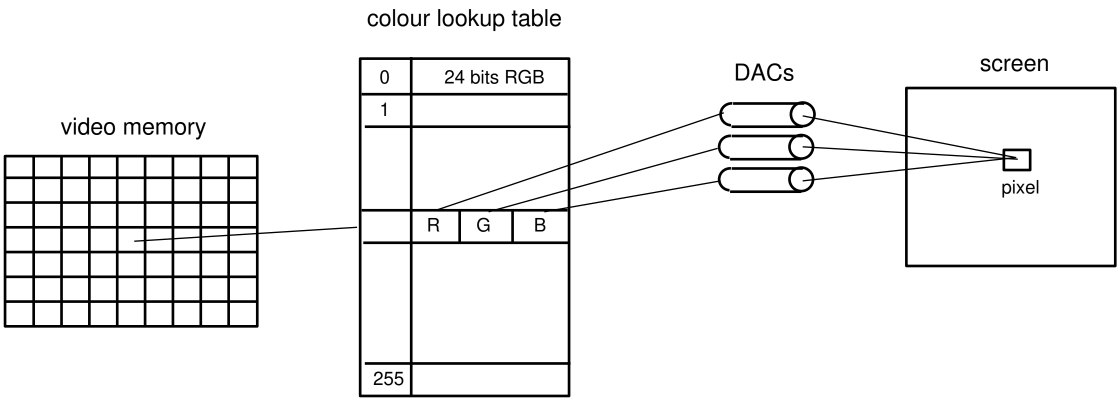


A *stride* is the difference in addresses of a pixel in contiguous lines of the display.

16 million colours is generally considered to be enough: it's about what the eye can distinguish. If we have only 8 bits/pixel, we can only display 256 colours: but which 256? Clearly, any particular choice of 256 is not going to be suitable for all the pictures we might want to display.

The solution is to use a *colour look up table*, or CLUT (or CLU or LUT). This is a table of 256 entries of 24 bits (3 bytes). When we wish to draw a pixel on screen we consult memory to determine what the value is (0 to 255). This is used as an index into the CLUT, where we get a full 24 bit colour. This is the colour we draw.

The good point is that we can change the values in the CLUT to suit our particular picture. At any one point in time we can show 256 colours, but those 256 can be chosen from 16 million. To display a different picture, we fill the CLUT with another set of 256 suitable values.

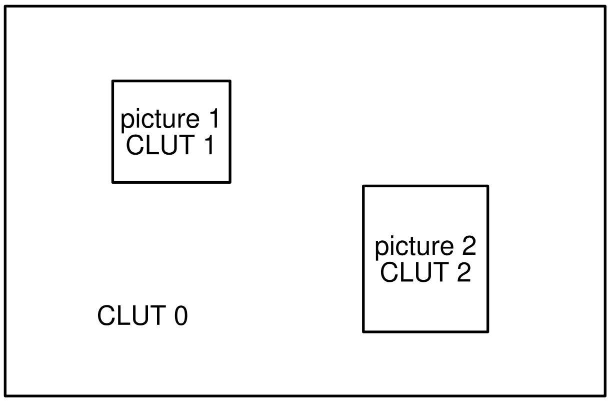


There are some drawbacks.

- We still have to choose 256 colours: natural pictures tend to have a lot more than this. This is not so bad, though, as we can generally get away with something that looks reasonable.
- We can't properly compare two different pictures side-by-side at the same time. If they contain different sets of colours, which CLUT do we choose? We could (a) switch between CLUTs as we move the mouse

from one picture to the other, but then the non-focus picture is displayed with wrong colours. Colours flash and change as we swap colour maps. Or (b) compromise with a shared CLUT that picks colours as best as possible from both pictures. But then both pictures have degraded colours.

Intermediate to getting proper 24 bit screens is to use multiple colour maps: each area of screen is given its own colour map, and as the screen is displayed, the hardware selects the right CLUT for the current area.



The display hardware has to switch in and out of CLUTs as the screen raster moves across the various areas. This is a fair solution if your hardware can support enough CLUTs (12 or so can be found).

2.3 More features

Monitors are refreshed at 60 Hz or more: the faster the better. Slower refresh produces screen flicker, particularly under artificial light.

An engineering solution to avoid high refresh rates is to use an *interlaced* display, developed by early TV people. Instead of drawing a full screen of lines from top to bottom (*non-interlaced*), we first draw the odd-numbered lines, and then go back and fill in the even-numbered lines. Each pass is called a *field*. For natural images (e.g., TV pictures) the local coherence, the relatively slow decay of the screen phosphor and the persistence of the eye fills in the gaps, and we get approximately as much flicker as if we were refreshing at twice the rate. Thus field rate is twice the frame rate.

For non-natural (computer generated) images, this is not so good. Better to use high refresh rate, non-interlaced displays.

In the UK, TVs have 625 lines, and are refreshed at 50 Hz. In the USA, they have 525 lines at 60 Hz.

On a good computer monitor, a single pixel is about 0.2mm square, and you have typically 1280 by 1024 pixels, maybe at 100 dots per inch. A good printer might have 2400 dpi.

eye	11000 x 11000
TV	400? x 625
Kodak disc film	1400 by 2200
35mm slide	4500 by 4500

Approximate numbers.

Note that higher resolution screens require more memory, thus the amount of memory that need to be processed for a high resolution picture is larger: this means that the perceived speed of a high resolution screen can be lower than a low resolution screen.

Note that there is quite a few differences between a computer monitor and a TV screen:

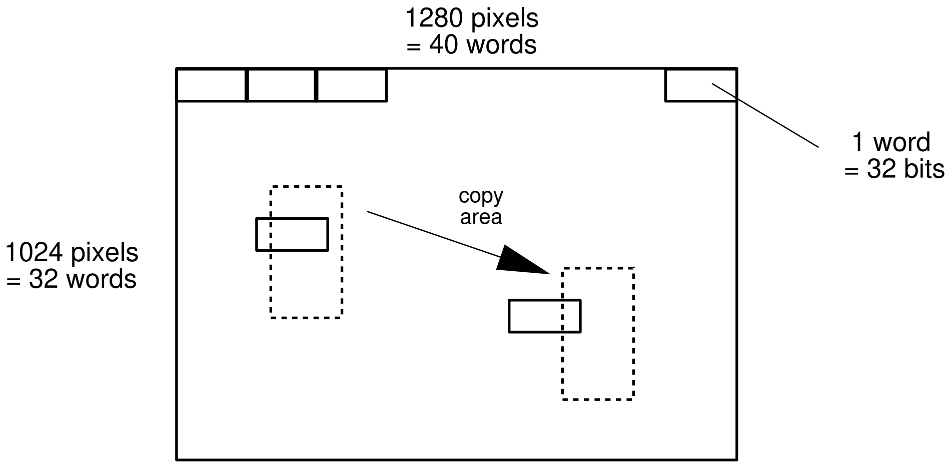
	TV	monitor
resolution	low	high
phosphor	slow	fast
refresh	25Hz	75Hz or more
brightness	bright	dim

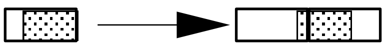
It's not really a good idea to try to use a TV as a monitor.

Consider the amount of data that a single frame contains: even a low resolution TV screen takes about 1.2M bytes. At 25 frames per second this is about 32G bytes per second. A high resolution computer monitor has correspondinly higher data rates. It was realised quite early that hardware support is required to move large numbers of pixels rapidly. Thus the *bitblt* (or *rasterop*) was developed.

2.4 Bitblt

It's easier to start thinking of a 1 bit (black and white) screen. Recall we actually are working with a 1D array of bytes.





Suppose we have a 1280 by 1024 screen. Each line requires 1280/8 = 160 bytes (the stride), total 160K bytes. If we wand to copy a rectangular area of pixels. The Blt hardware has to

- identify all the bytes involved, and separate out partial bytes at the edges
- copy the bits to the destination, with the same edge problems, and making sure it doesn't overwrite any other bits at the destination.

- do this very quickly

Copy is only one of the operations that are useful. There are 16 ways of combining a source bit and a destination bit to give a result bit.

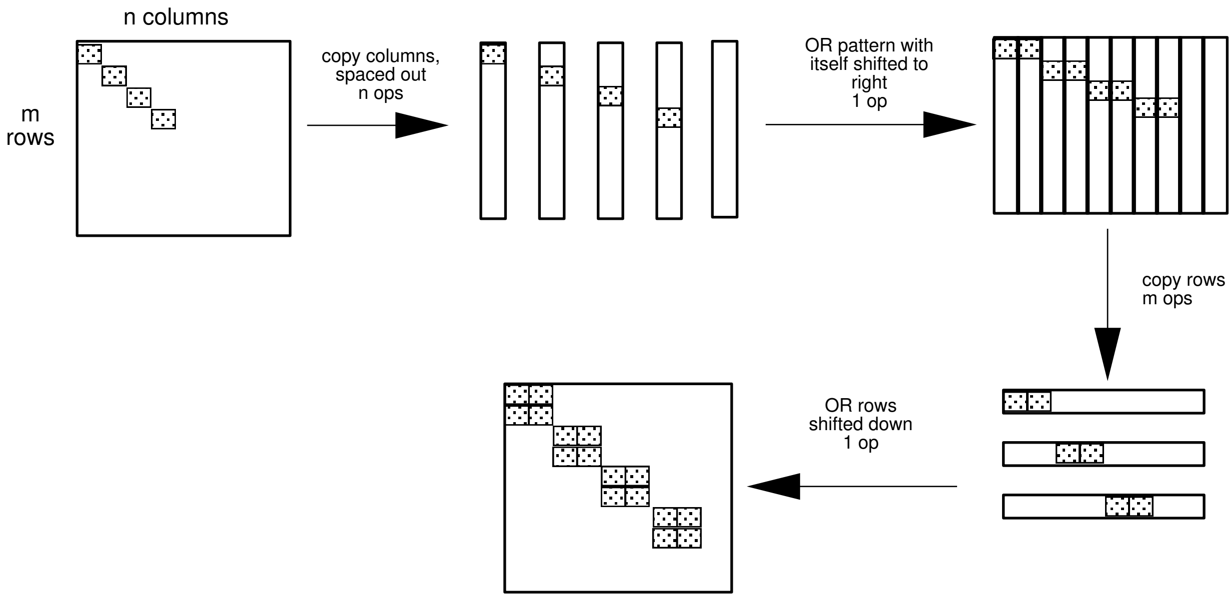
input	00	01	10	11	operation
	0	0	0	0	false/clear
	0	0	0	1	and
	0	0	1	0	and not
	0	0	1	1	source/copy
	0	1	0	0	not implies
	0	1	0	1	dest/nop
output	0	1	1	0	xor
bits	0	1	1	1	or
	1	0	0	0	nor
	1	0	0	1	xnor
	1	0	1	0	not dest
	1	0	1	1	implies
	1	1	0	0	not source
	1	1	0	1	nand not
	1	1	1	0	nand
	1	1	1	1	true/set

Read this as: each column is headed by xy. x is the source bit, y the destination bit. The values in the columns are the result bits.

If the source is the same as the destination, i.e., we Blt to ourself, we get

input	0	1	operation
	0	0	false, and not, not implies, xor
output	0	1	and, source, dest, or
	1	0	nor, not dest, not source, nand
	1	1	xnor, implies, nand not, true

Using BitBlt we can get some nice effects. Here is how to magnify an $n \times m$ image by 2 times, using $n+m+2$ BitBlt operations.



The familiar "drag and drop" effect is done using BitBlt: as an image is dragged around, the original background is copied to somewhere off-screen, and the image is copied in its place. Small images can be moved smoothly like this: as the areas get bigger, we get more flicker as we see the BitBlt hardware in action.

2.5 ByteBlt

BitBlt is used for single bit (black and white images). We can extend the idea to tonal and colour images: this is called ByteBlt. We can directly apply all of the above, but now there are extra complications. If we are using CLUTs, then the mapping of byte values to colours is arbitrary, so we can not quite know what colour to expect, unless, of course, we were very clever in choosing the values in the CLUT.

We can also get other "layering" effects. Suppose we want to copy an area that has a red design on a blue background, but we want only to copy the design, and not the background. We are considering the red as "foreground," and the blue as "background." A typical use of this is copying text from one place to another, where we want to copy the text only, and only to obscure the pixels the letters themselves cover at the destination.

We can generalise this to impose various conditions on source and destination foreground and background. Conditions include

$$\begin{array}{ll} \text{source} = \text{colour} & \text{destination} = \text{colour} \\ \text{source} \neq \text{colour} & \text{destination} \neq \text{colour} \\ \text{source} \geq \text{colour} & \text{destination} \geq \text{colour} \\ \text{source} \leq \text{colour} & \text{destination} \leq \text{colour} \end{array}$$

So we copy those source pixels that are equal (or not equal, or ...) to a given colour; and we update only those destination pixels that that are equal (or not equal, ...) to another given colour. This kind of test fits well with using CLUTs, as we can pick and choose the right values for the effect we want.

2.6 Limited Tonal Resolution

What do we do if we have an image that has more bits per pixel than our display? In particular, we might have a greyscale image but only a black and white monitor. How can we display a decent approximation on our limited hardware?

The simplest approach we might take is to truncate our data. E.g., we have 32 bit grey, to display on an 8 bit screen, just chop off the lowest 24 bits. However, this produces very poor pictures, particularly if most of the picture information is in subtle variations: a smooth grey gradient (hex) 4000, 4008, 4010, 4018 truncates to a flat 4, 4, 4, 4. A similar slightly darker gradient 4FF0, 4FF8, 5000, 5004 becomes a sharp transition 4, 4, 5, 5.

Incidentally, about 64 levels of grey seems to be the minimum required for a smooth-looking picture.

So we need better methods. There are several, and they rely on *spatial integration* of the eye. That is, the eye tends to average out the tones in an image over an area: a dark area near a bright area tends to be seen as a grey area from a large enough distance.

We can trade *spatial resolution* (how sharp a picture is) for *tonal resolution* (how many greys a picture has). This idea is not new: it was discovered in the Art world by the pointilists in the early part of this century, and again by the pop artists of the 1960s.

2.6.1 Half Tones

For the first method, consider a 1 bit screen. We can display various greys by using patterns of dots. Look at a black and white newspaper picture: it may seem to be greyscale, but if you look closely, you see it is comprised of variously sized black blobs. The more black in an area, the darker the perceived image when seen from a distance.

A newspaper uses more-or-less continuously sized blobs, while on a screen we a limited to pixels. A typical approach is to use a cell of 2 by 2 pixels. For white, we set all 4 pixels to white. For black, we set all 4 pixels to black. For grey levels, we can set one, two or three pixels to black with the rest white.

bb bb	bw bb	wb wb	bw ww	ww ww
cell 0	cell 1	cell 2	cell 3	cell 4

Cell 0 is black, 4 is white, and cells 1, 2, and 3 appear as three levels of intermediate grey.

There is a problem with this, though. The eye is very good at picking out edges in images. If we have a large area of grey from cell 2, the edge detector kicks in, and you see vertical stripes, not the grey we want. We need to build the mid-greys in an irregular fashion so the eye cannot pick out patterns.

Another problem occurs when we overlay patterns: we don't get a nice additive effect of greys as black plus black is black, while black plus white is also black.

To overcome this we make cell n differ from cell $n-1$ in just one pixel:

bb	bb	bw	bw	ww
bb	wb	wb	ww	ww
cell 0	cell 1	cell 2	cell 3	cell 4
0–0.2	0.2–0.4	0.4–0.6	0.6–0.8	0.8–1

Now large areas of cell 2 look reasonable.

We can represent this progression of grey cells by a single matrix that tells us at which level a pixel first becomes white.

$$\begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix}$$

This matrix is called the *spot function* by PostScript.

So now we have an image with 5 levels of grey: what have we lost? Well, now the picture is twice as large as we are using 4 pixels to display the original one pixel. The spatial resolution has dropped.

We can get more levels of grey by using large spot function matrixes. For example, a 3 by 3 matrix gives us 10 levels of grey, at the price of scaling the picture up 3 times. Here's a suitable matrix

$$\begin{pmatrix} 7 & 9 & 5 \\ 2 & 1 & 4 \\ 6 & 3 & 8 \end{pmatrix}$$

This technique also applies to colour, using pixels in the array set to various colours to approximate the desired colour. This is much harder, though, as it requires a good knowledge of how the eye perceives colours. It also applies to reducing to non-bilevel images. If we have a 2 bit display, each pixel can be one of 4 values, meaning a 2 by 2 cell can display $4 \times 3 + 1 = 13$ levels (4 pixels $\times$ 3 levels other than black + black).

2.6.2 Dithering

What if doubling (or more) the size is unacceptable, and we want to display the image at the original size? We can use an alternate technique called *dithering*, or *linear diffusion*.

This, too, is mainly used for tonal images, but can be used for colour. Like halftoning, it has the property that you can output the picture pixel by pixel in scan order (compare with error diffusion, below). This means that we don't need to scan the image multiple times, or use extra memory to buffer partial results.

The basis of dithering is this: as we scan along a row we form a running total of all the pixel values so far (i.e., those values that we would like to have output), outputting black pixels as we go. Whenever the sum exceeds a given threshold value, we output a white pixel, and reduce the running sum by the threshold amount.

The sum can be thought of as the local error between the total intensity (white) displayed so far, and the actual accumulated intensity from the source image. When the error gets too large, we reduce it by outputting a white pixel. Grey areas therefore tend to contain the appropriate proportion of black and white pixels.

We get a perceived greyscale, with the picture the same size as the original.

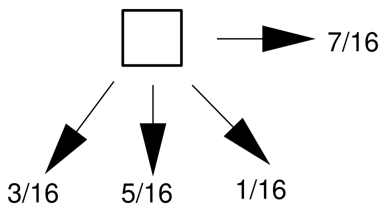
Two problems arise.

- Areas of uniform grey produce stripes, just as for the bad halftone spot function. This happens due to the regular build up and reduction of error. One way to fix this is to introduce a small element of random *noise* from a distribution with mean zero. This will break up the regularity, while the zero mean ensures that the overall intensity of the picture is unchanged. Notice that we are making a picture look better by adding noise! This is another way we lose spatial resolution.
- How to choose the threshold? If we choose mid-grey, then predominately dark pictures will become almost entirely black, while white ones will become almost entirely white. Ideally the threshold should be in the middle of the tonal range of the picture, which requires us to make two passes through the data.

2.6.3 Error Diffusion

This is an extension of the dithering in that we try to spread error locally, rather than just pushing it to the right until it becomes visible. They argument is that error appears close to where it should lie, and is not visible as an artifact in the picture.

Here is error diffusion, as described by Floyd and Steinberg. The error term at any given pixel (i.e., the difference between the actual pixel value and the rounded or truncated displayed value) is passed forwards *and downwards* to the neighbouring pixels (which have not yet been visited by the scan). Some proportion of the error is added into the unvisited pixels. Floyd and Steinberg pass the error to four other pixels: 7/16 of the error goes t the right; 3/16 to below left; 5/16 to directly below; 1/16 to the right.



The error is therefore kept spatially close to the pixel (unlike dithering where the error can appear much later down the scan line). It tends to keep local average of tone (or colour) the same in the displayed image as in the source image.

Notice that this method requires a little lookahead to access the neighbour pixels. Generally this is not a problem.

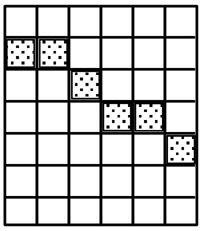
A slight improvement is to alternate the directions of scanning across row: left then right, then left, etc., as this reduces the bias in the direction of the error.

This generalises to displays of any depth: we display the higher order bits of the source image, and the error is the lower order bits. It is particularly good at reducing colour images with little visible degradation (e.g., 24 bits to 8 bits, as is common on many PCs).

2.7 Effects of Limited Spatial Resolution

We now turn to a different problem: what happens if our screen has too few pixels to display our picture adequately? For example, we have more pixels in our picture than on the screen?

Again, simple sampling of the image is not good, as we get *aliasing* (also called *sampling defects*). Technically this is the appearance of high frequency components in the original as low frequency components in the sampled image (see below). This is because the sampling rate is not high enough to capture the detail of the object we are trying to draw. This is most obvious for simple line drawings, where sloping lines appear with *jaggies*, a staircase effect.



Another manifestation is in time, rather than space, in animations, or any other moving picture. The classical example is the backwards rotating wheel on a car or cowboy waggon, where the sampling rate (the frame rate) is not fast enough to capture the rotation of the wheel.

This is a general feature of trying to digitising any kind of continuous data, including sound: aliasing can appear in sound, too, and is the source of much debate of the quality of CDs vs. records.

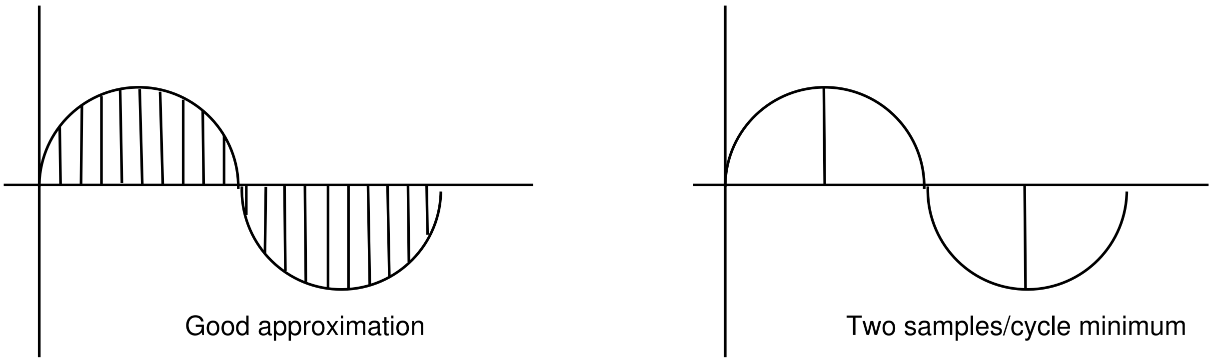
Correcting for this kind of error is called *anti-aliasing*.

2.7.1 Sampling and the Nyquist limit

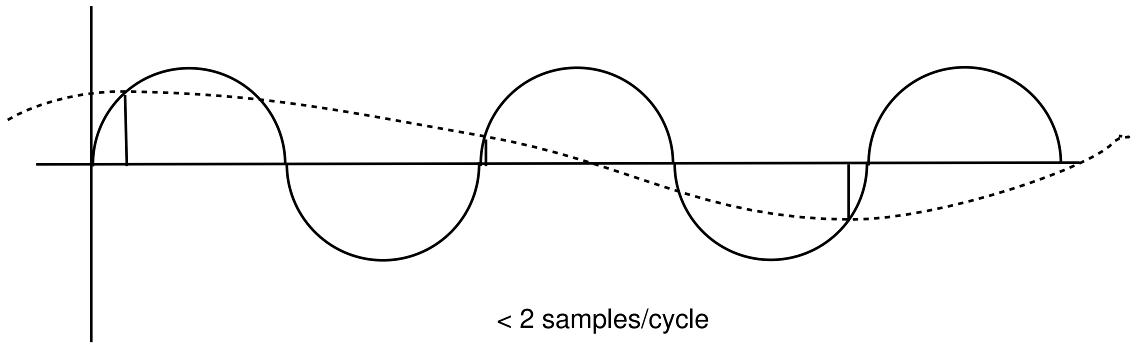
Sampling theory explains what is happening, be it space (staircasing) or time (backwards rotating wheel), or sound or whatever. A picture of a line samples "the true line" at regular intervals (the pixels). An animation samples "the true image" at regular intervals (the frames). A basic theorem says:

> you cannot reconstruct the original signal from a sampled version of it unless the sample rate is at least double the highest Fourier component of the original.

The is the *Nyquist Sampling Theorem*. The sampling limit is the *Nyquist limit*. To put it another way, the highest frequency you can safely reconstruct from samples at rate n is $n/2$. So for an animation, you cannot safely view an object that moves faster than half the frame rate. Or for lines you cannot properly draw lines with slopes that exceed half the pixel size. Or for CDs, which are sampled at 44kHz, you cannot reconstruct a frequency higher than 22kHz (which should be safe, as this is the limit of human hearing).



If there are higher frequencies than the sampling rate, aliasing can occur.



Here the high frequency solid line is sampled at too low a rate, and is reconstructed as the dotted line. Thus a high frequency appears under the alias of the low frequency.

The solution is to remove all such high frequencies (whenever possible) before sampling. Thus CD music is filtered to remove all sound above 44kHz before being digitised. Such filtering is not always possible on images, though.

Digitising almost any natural picture has this problem. The samples (the pixels) try to represent the original source image.

- fine detail that falls between samples disappears
- straight edges become jagged
- regular patterns close to the pixel spacing break up (we get sections that are sufficiently on the pixel, and so appear, followed by sections that are sufficiently off the pixel, and this disappear)

Thus we need to treat each pixel as a *area* sample of the "real" picture (whether the "real" picture is a real picture, or a higher resolution digital image).

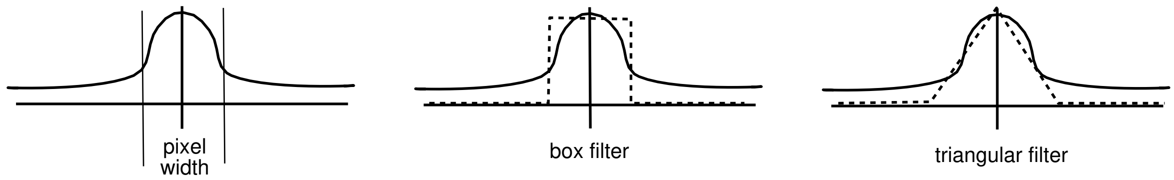
To see how to do this, we need to consider how images are displayed.

2.7.2 Raster Monitors

We again concentrate on one particular kind of display device, the raster monitor (standard TV screen). Other kinds of display have the same kinds of problems.

In the vertical direction there is a discrete structure afforded by the scan lines. In the horizontal direction there is an almost continuous trace, but this tends to be discretised by the blobbiness of the phosphor on the screen, and the fact that pixels are held discretely in memory.

Getting in more closely, we find that the electron gun actually illuminates a finite area at any particular instant, and it does so non-uniformly, with more electrons at the point of focus than the surrounding area. Generally, there is a normal distribution of electrons about the point of focus. Thus the spot will be circular, and brightest in the middle. Furthermore, neighbouring pixels will be illuminated (by the tails of the normal bell). Better VDU have screens between the electron gun and the phosphor to try to minimise this effect.



So to get an accurate illumination, we would have to store at a pixel location a value that is the weighted sum of the central pixel and its neighbours. Ideally we would use a Gaussian weighting, but this is bothersome to compute, so we approximate.

A simple and popular approximation is to isolate each pixel, and not consider the contribution from neighbours. Further, we assume a constant illumination across the pixel. This is a box filter. A better, and nearly as easy approximation is to use a triangular filter.

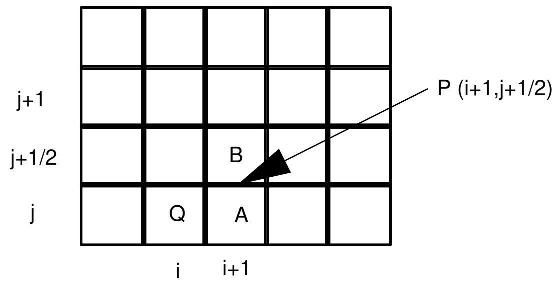
2.7.3 Straight Lines

We will look at other ways of antialiasing later, but for now we look at the simple case of drawing an antialiased straight line. And to do that, we will have to remember how to draw an *un-antialiased* line!

Here is the derivation of Bresenham's algorithm.

The problem: given the equation of a line, draw it as fast as possible on the screen. This means we want to use integer arithmetic, as floating point will be too cumbersome (and difficult to implement in hardware). Our main assumption is that we will illuminate those pixels that have centres closest to the actual line.

We simplify by assuming the slope of the line is from 0 to 1, that is the line lies within the first octant. Every other case follows by symmetry.



Suppose we have just illuminated pixel centred on Q. Since $0 \leq$ slope ≤ 1, the next pixel on the line can only be one of A or B. We require a simple mechanism to make the choice.

So we have: the equation of the line is $y = mx$ (assuming the line passes through the origin), with $0 \leq m \leq 1$. Let the value $g = mx - y$. Then

- $g < 0$ for points (x, y) that are above the line
- $g \geq 0$ for points (x, y) that are below or on the line

Consider the point P, half way between A and B, at $(i + 1, j + 1/2)$. So

- $g_P < 0$ illuminate A
- $g_P \geq 0$ illuminate B

(Think why it is this way about.)

We now show how to compute the sign of g_P quickly. At P,

$$g_P = m(i+1) - (j+1/2) = (i+1)dy/dx - (j+1/2).$$

Rearrange

$$2g\,dx = 2dy(i+1) - dx(2j+1),$$

so define $d_P = 2dy(i+1) - dx(2j+1)$, where $dx = x_2 - x_1$, $dy = y_2 - y_1$ and (x_1, y_1) and (x_2, y_2) are any two points on the line.

Notice we can compute d_P with no divisions, and that it has the same sign as g_P. Thus

- $d_P < 0$ illuminate A
- $d_P \geq 0$ illuminate B

This is good, but we can make it better. The formula for d_P has several multiplies in it: we can get rid of them by using an *iterative* formulation. Suppose that Q at (i, j) is the $(k-1)$th point n the line we have computed, viz., $x_{k-1} = i$,$y_{k-1} = j$. Substitute in the equation for $d_P = d_k$ above

$$d_k = 2dy(x_{k-1} + 1) - dx(2y_{k-1} + 1),$$

or,

$$d_{k+1} = 2dy(x_k + 1) - dx(2y_k + 1).$$

Subtract these two equations

$$d_{k+1} = d_k + 2dy(x_k - x_{k-1}) - 2dx(y_k - y_{k-1}).$$

But $x_k - x_{k-1} = 1$, so

$$d_{k+1} = d_k + 2dy - 2dx(y_k - y_{k-1}).$$

And more: since $y_k - y_{k-1} = 0$ or 1:

if $d_k \geq 0$, then we select B, and $y_k - y_{k-1} = 1$, so

$$d_{k+1} = d_k + 2(dy - dx),$$

if $d_k < 0$, then we select A, and $y_k - y_{k-1} = 0$, so

$$d_{k+1} = d_k + 2dy.$$

Thus, given d_k it is easy to determine (a) which pixel to move to next, and (b) the value of d_{k+1}. We need a way to start: let $(x_1, y_1) = (0, 0)$. Putting this into the equation we get

$$d_1 = 2dy - dx.$$

In summary

$$d_{k+1} = \begin{cases} 2dy - dx & \text{if } k = 0 \\ d_k + 2(dy - dx) & \text{if } d_k \geq 0 \\ d_k + 2dy & \text{if } d_k < 0 \end{cases} .$$

This all operates using integer arithmetic, and $2dy$ and $2(dy - dx)$ can be precomputed just once for each line. So the per pixel cost is roughly the decision plus a single addition.

An example. Draw a line from $(0, 0)$ to $(10, 6)$. We have $m = dy/dx = 0.6$. Also, $dy = 6$, $dx = 10$, $2dy = 12$, and $2(dy - dx) = -8$.

k	d_k	a	
1	2	0.5	
2	-6	0.1 (dark)	
3	6	0.7 (light)	
4	-2	0.3	
5	10		If positive, subtract 8; negative, add 12.
6	2		
7	-6		
8	6		
9	-2		
10	10		

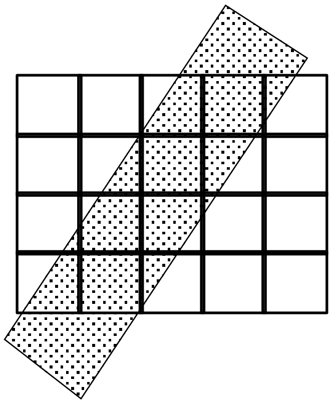
For $d_k < 0$ go across; for $d_k \geq 0$ go up:

```
            x
          xx
        xx
       x
     xx
   xx
  x
(0,0)
```

2.7.4 Drawing an Antialiased Line

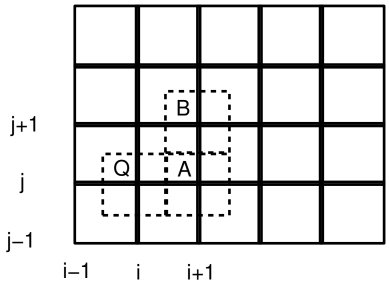
We can now tackle the problem of antialiasing the line. Clearly the line as produced by Bresenham has pronounced staircasing. If we have a greyscale display we can reduce this visual effect to display a line that appears smooth.

Consider a "real" line as it passes through the pixels:



The line has edges, and is black on the inside, white on the outside. Each pixel on the screen covers a finite area which intersects the line. Thus we need to ask how much of a pixel is covered by the line, and set the tone of the pixel to a grey that is proportional to the amount of cover.

Let us consider a single edge of the line, and here is the diagram again (except we have shifted the squares slightly to make some computations easier):



What fraction of the Q pixel is covered by the line? The white area within the pixel is $a(i, j) = mi - (j - 1/2)$. In Bresenham's algorithm we looked at the value g

$$g(i + 1, j + 1/2) = m(i + 1) - (j + 1/2) = a(i, j) - 1 + m,$$

or

$$a(i, j) = g(i + 1, j + 1/2) + (1 - m).$$

We actually had $d_k = 2dx\, g$. Thus

$$a(i, j) = d_k/(2dx) + (1 - m)$$

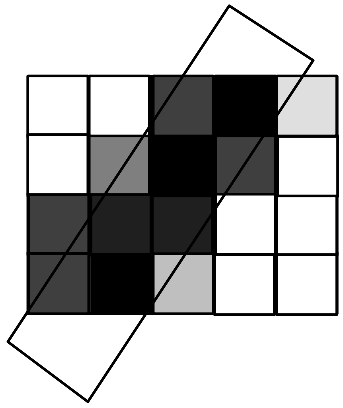
is the area we want.

We can precompute $2dx$ and $1 - m$ for a whole line, but now to compute a we need to do a floating point division (and add) for each pixel.

Example, continued.

$a(i, j) = d_k/(2dx) + (1 - m)$. Now $1 - m = 0.4$, $2dx = 20$, so to compute a we divide d_k by 20, then add 0.4.

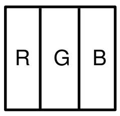
The kind of effect we get is:



At a suitable resolution or distance the eye sees this as a simple line with smooth edges.

This method is called *unweighted area sampling*. More sophisticated antialiasing uses *weighted area sampling*, where we might want to take into account which part of the pixel is covered (e.g., a pixel with its centre covered should be darker than a pixel with an edge covered by the same area); or pixels that don't intersect the line (pixels at a small distance from the line can usefully add to the antialiasing). Such methods are more expensive to compute, of course, but produce better results.

A new (according to Microsoft, used in their ClearType font system) or old (according to Apple) variant on antialiasing is possible on screens that allow *sub-pixel* addressing. On a LCD screen a single pixel is actually composed of three coloured (red, green, blue) LEDs in a row, and each can be illuminated individually by choosing the colour of the pixel appropriately.



Single pixel composed
of three LEDs

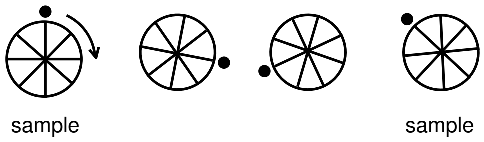
Thus, if we arrange things carefully, we can triple the effective horizontal resolution of a screen. The problem is to ensure that we illuminate the correct proportion of each colour LED so that we don't change the overall perceived colour of a region: for example, if we want a white region we need to illuminate an equal proportion of red, green and blue pixels. If we are drawing, say, black text on a white background this tuns out to be fairly easy, and we get a massive improvement in text legibility.

2.8 Motion Blur

Producing an animation is not simply taking a set of still pictures and running them together. If we do this we get another kind of aliasing.

2.8.1 Temporal Aliasing

An animated sequence is a discrete sampling of continuous action. Thus we can get *temporal aliasing* is the sample rate is not high enough. This causes things like the backwards rotating waggon wheel in cowboy films.



As the wheel moves quickly around clockwise, it reaches a position at the next sample that appears to be just backwards from the start. The eye interprets this as anticlockwise motion.

Antialiasing appears as motion blur in the still frames of the animation. The blur is in the direction of motion of the object in the scene. For a waggon wheel, the blur would be enough to lead the eye in the direction of clockwise motion.

If you look at a single frame of, say, Toy Story, you will see it is not a sharp picture as you might expect to be generated by a computer. Instead objects are blurred, and the faster the object is moving, the more blurred it is. A similar effect happens with some of the new digital cameras: viewing people running past we see a sequence of frames rather than continuous motion as we get from analogue cameras.

Getting an absolutely correct solution to motion blur is very difficult. Consider the problem of deciding what colour to make one pixel in one frame. The frame lasts for a fraction of a second. In that time several object may move across the pixel, each contributing some portion to the overall colour.

Moreover, object that cover the pixel early in the frame should contribute less than objects that cover it for the same amount of time late in the frame due to phosphor decay. Even better would be to extend the sample time region to either side of the frame's time interval to allow for the persistence of vision.

These calculations tend to be expensive, so motion blur algorithms start by determining which objects within a frame are stationary, and which are moving, and ignoring the stationary areas.

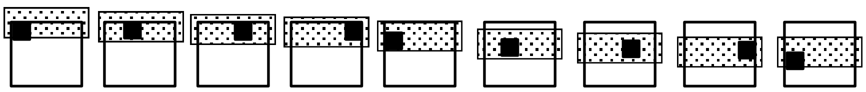
2.8.2 Cook's Method

This method was described by Cook in 1986. This is a statistical approach that adds noise to a picture. It works because the eye tolerates noise better than aliasing.

We start by *supersampling* each pixel. This means we split a single pixel up into (say) 4×4 subpixels.

7	11	3	14
4	15	13	9
16	1	8	12
6	10	5	2

We will sample the image 16 times, once through each subpixel. We do these samples at different points in time, namely every 1/16th of a second. As usual, we avoid supersampling in a regular order, as this causes artifacts. For example, if we had numbers left to right, top to bottom, then the upper half of the pixel would be biased towards the earlier part of the time interval, the lower half towards the later part. An object moving from top to bottom at just the right rate could be hit by every sample, even though it actually covers the pixel only a fraction of the interval.



An object moving down at 1/4 pixel per second, but only covering the pixel for half the time interval.

So instead we pick the subpixel from the grid above. And more, we add random *jitter* to the sample time of up to $\pm 1/32$ of a frame. The jitter will be different for each frame of the animation, unlike the supersampling pattern. The jitter is essential, for otherwise we have regularly timed subsamples, and any regularly occurring event would be sampled identically. Here we are trading regular samples for a noise element that the jittering introduces.

We do this for each of th 16 subpixels, and average the result: this is the final colour of the pixel.

We can reduce the noise by increasing the supersampling rate, e.g., using a 8×8 array. We can do this dynamically, too, if we know something about the speed of the objects in the animation. We might use 8×8 for "fast" objects, but 4×4 for slow objects. Cook suggests that objects that move more than 8 pixel in x of y in a time frame are "fast".

2.9 Intensities and Gamma Correction

There is one further property of monitors that we want to consider, namely the selection of tones or colours to display so that the viewed picture is as close to the true image as possible.

For example, on an 8 bit screen we have 256 levels of grey to choose. If we have an image with pixel values ranging from 0 to 255, it may appear that there is no problem: simply map pixel value 0 to grey 0, pixel value 1 to grey 1, etc.

This assumes that the screen has a linear response: that the intensity (energy output) is linearly proportional to the value. More importantly, it assumes that the eye has a linear response, too. This is not so, as the eye has a roughly logarithmic response: the perceived change in brightness of a pixel is proportional to the *ratios* of intensity.

For example, changing the intensity of a pixel from 0.11 to 0.1 appears to the eye to be the same size step as changing from 0.55 to 0.5.

So to get a perceived linear response from a screen we need a multiplicative scale, not an additive one. We measure intensity from 0 to 1. Let I_0 be the minimum brightness of a pixel (this is never quite 0, as there is always ambient light reflecting off a screen, and the colour of a screen is never true black), and let $I_{255} = 1.0$ be the maximum intensity.

To get a perceived linear scale we require the ratios I_{k+1}/I_k to be constant, r say.

$$I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2 I_0, \ldots I_{255} = r^{255} I_0 = 1.$$

So $r = (1/I_0)^{(1/255)}$, $I_k = I_0^{(255-k)/255}$.

Typical values for I_0 are 0.005 to 0.025, measured by a photometer on a black and white screen in a darkened room.

The *dynamic range* of a screen is the ratio of maximum and minimum intensities $I_{255}/I_0 = 1/I_0$. With an I_0 of 0.02 we have a dynamic range of 50, $r = 1.015$, and the intensity values I_k are

k	0	1	2		254	255
I_k	0.02	0.0203	0.0206	...	0.9845	1.000

If the ratio r is 1.01 or less, neighbouring intensities are indistinguishable to the eye.

Next is the problem of non-linear response of the screen. You may find that a picture that displays well on one screen looks quite dark or light on another screen. This is because the actual intensity of a pixel is not linear in the intensity value, but is (roughly) proportional to v^γ, where v is the intensity value (or voltage on the electron gun). The value γ tends to be a small number, e.g., 2.2, and should be determined for each individual screen. Rather than recompute for each pixel, appropriate values are stored in a table, and looked up when needed. This is called *gamma correction*. A given picture should look the same on any screen that is gamma corrected. For an uncorrected screen, a picture will look darker on a screen with a higher gamma.

Other kinds of display require different kinds of gamma correction. For example, a liquid crystal display has more of an "S" shaped response curve. The same remedy applies: precompute the corrections and store in a table.

3 Solid Models and Raytracing

To draw a picture, such as a frame from an animation, or a architectural plan of a building, we must first start with some model within the computer. This model will contain all the information we require to make the picture, such as what objects are in the picture; what colours they are; what textures, reflectivity, and so on.

We need to consider how to store this kind of data in such a way that it is convenient for the rendering process. We shall look in detail at two popular ways, surface modelling and solid modelling. The kind of model strongly affects the rendering process in that certain kinds of models tend to be rendered in certain ways, but this connection is not absolute. For example, solid models fit well with ray tracing.

3.1 Solid Models and Ray Tracing

We shall first consider solid models.

A solid model explicitly represents the entire body of an object, rather that just its surface. It can be thought of being closer to reality, and is popular with engineers, who need to know how much metal there is in a bridge. This sort of information is difficult to get from surface models.

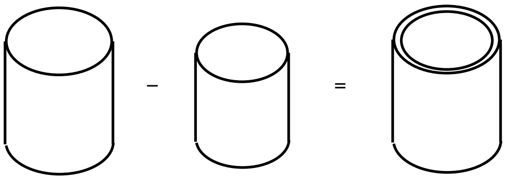
One of the big advantages of ray tracing is that we can simply model many phenomena that real light exhibits, such as reflection and refraction.

3.2 Constructive Solid Geometry

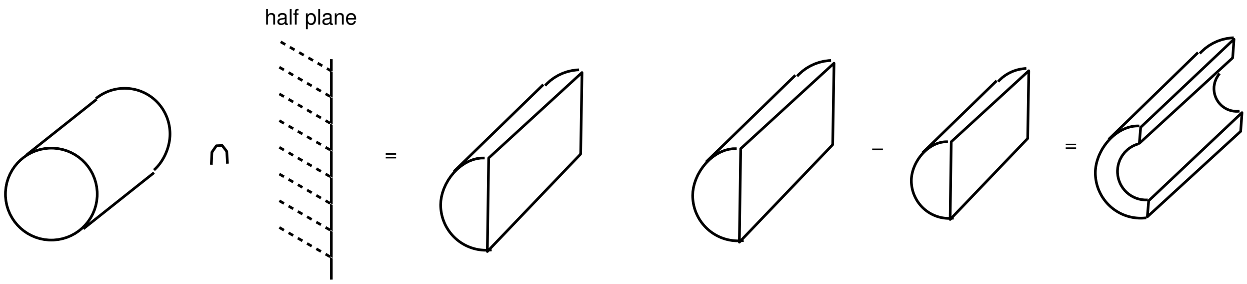
We must start with some representation of the world we wish to draw.

CSG models build on basic mathematical solid primitives, such as the sphere, cylinder, cone and cube. These are combined in various ways to make more interesting shapes. CSG uses a binary tree of combinations: an internal node is a boolean operation on transformations (scale, rotate, etc.) of its branches, while a leaf node is a primitive object. Boolean operations include union, intersection, difference, etc.

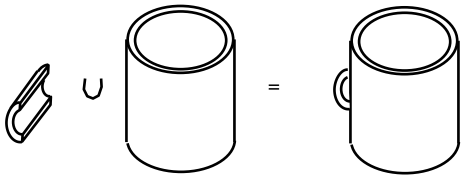
Suppose we wish to model a cup. We can think of this as a solid cylinder with a slightly smaller cylinder cut from it



Then we can add a handle that is composed of the difference of two half cylinders. A half cylinder can be made by subtracting a half plane from a cylinder.



We then glue (union) the handle onto the cup.



In tree form, this looks like

```
BODY =   -                 op
       /   \
     big  small            transforms
     /       \
    CYL      CYL           primitives

HALF =  INT                 op
       /   \
      ?     ?              transforms
     /       \
   CYL    HALFPLANE        primitives

HANDLE = -                 op
       /   \
      ?     ?              transforms
     /       \
   HALF     HALF

CUP =    U                 op
```

```
        /   \
       ?     ?               transforms
      /       \
    BODY     HANDLE
```

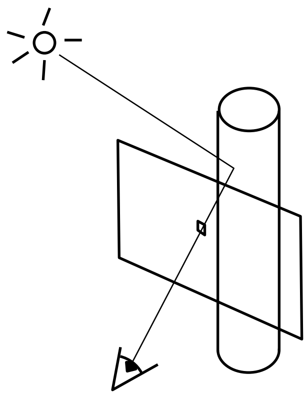
In this way we can build up complex objects. The only limitations are the original primitives.

The primitives can be represented by mathematical equations, e.g., a sphere is $(x-c_x)^2+(y-c_y)^2+(z-c_z)^2 = r^2$. Then we can use polynomial equations to determine intersections. this may be done algebraically or numerically.

3.3 Ray Tracing

This technique, also known as *ray casting*, is well suited to solid models as it requires knowledge of where lines intersect the objects in the model.

The basic mechanism is based on the way light moves in straight lines in the real world. A line from the eye is traced through the current pixel until it hits an object. We can then trace back from the object to the light source.



This allows us to determine what colour the pixel should be. We repeat for all pixels on the screen. This is the essence of ray tracing: the details are very much more complicated.

3.4 An Example of Ray Tracing

Here is a simple example of the kind of computations involved in ray tracing.

A ray is defined by a start point $\mathbf{s}$, and a direction $\mathbf{d}$, usually a unit vector:

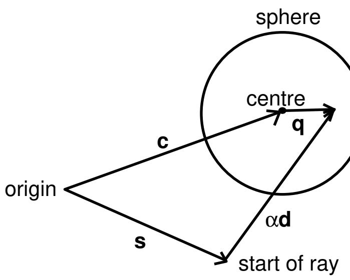
$$\mathbf{p} = \mathbf{s} + \alpha\mathbf{d},$$

with α the distance along the ray from $\mathbf{s}$. When we intersect this line with an object, the value of α gives us the distance to the object.

When a ray is reflected in a mirror, say, or refracted when passing through glass, say, we can restart with new values of $\mathbf{s}$ and $\mathbf{d}$.

3.4.1 Intersection with a Sphere

We will intersect out line with a sphere, centre $\mathbf{c}$, radius r.



Suppose the intersection is at $\mathbf{q}$ on the surface of the sphere. Then

$$\mathbf{c} + \mathbf{q} = \mathbf{s} + \alpha\mathbf{d}.$$

Known are $\mathbf{c}$, r, $\mathbf{s}$ and $\mathbf{d}$. We want to find α and maybe $\mathbf{q}$.

Thus, by rearranging and taking the dot product of each side with itself,

$$\mathbf{q} \circ \mathbf{q} = (\mathbf{s} - \mathbf{c}) \circ (\mathbf{s} - \mathbf{c}) + 2\alpha\mathbf{d} \circ (\mathbf{s} - \mathbf{c}) + \alpha^2,$$

(since $|\mathbf{d}| = 1$), and so

$$\alpha^2 + 2\mathbf{d} \circ (\mathbf{s} - \mathbf{c})\alpha + (\mathbf{s} - \mathbf{c}) \circ (\mathbf{s} - \mathbf{c}) - r^2 = 0,$$

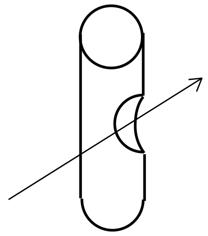
since $|\mathbf{q}| = r$.

This is a scalar equation that we can solve for α. We get two (complex) roots α_1 and α_2. If both are real, it corresponds to the ray entering and leaving the sphere, with the smaller root being the entry, and the larger root the exit. Both complex means the ray misses the sphere. A single root means the ray is tangent to the sphere. Notice that a quick test to determine if there is an intersection is just to compute the discriminant of the quadratic equation.

The normal to the surface at the point of intersection is $\mathbf{q}/|\mathbf{q}|$, where $\mathbf{q}$ is given by $\mathbf{q} = \mathbf{s} + \alpha\mathbf{d} - \mathbf{c}$.

3.5 Tracing the CSG Tree

In the general case of a complex model, how do we determine what part of the model the ray hit? For a single object in the world (e.g., a sphere), is is easy to find if the ray hits it or not. For a more realistic world model, many primitives will be combined using CSG and it is much harder to determine if a ray intersects. For example, we must take into account chunks of the model that are removed using a subtraction operator.

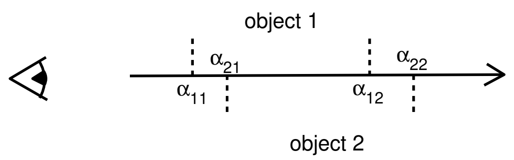


A ray misses a cylinder with a chunk subtracted.

Consider the CSG tree. We traverse this depth first, intersecting the ray with each leaf primitive (in practice only those primitives that lie with the octree voxel or bounding volume, see below). These intersections give α distances along the ray.

Between each α pair from an object there is solid; outside of them there is space. When solids overlap along the ray we use the boolean operations in the CSG to decide what the result should be (solid or space).

For example, suppose we have two intersecting objects, the first has intercepts with the ray at α_{11} and α_{12}, while the second has α_{21} and α_{22}.



If the combining operation is

- union, then there is solid from α_{11} to α_{22}
- subtraction, there is solid from α_{11} to α_{21}
- intersection, there is solid from α_{21} to α_{12}
- symmetric difference, there is solid from α_{11} to α_{21} and α_{12} to α_{22}.

We move up the tree, modifying these spans as we go. When we reach the top, we take the α of the nearest solid. This is the object we display.

Thus the intersection of the line and object can be computed by moving down the CSG binary tree taking note of the boolean operations. Eventually we come to a primitive, and it is fairly easy to find the intersection of a line with a sphere, or cone, or whatever. Then it is straightforward to find the surface normal at the point of intersection. This, together with the angle of the viewpoint and the angle of the light source are enough to apply *lighting models*.

3.6 Lighting

Light is essential to viewing a model: with no light we can't see anything! The techniques to light a model vary from simple ambient diffuse light to more complex ideas such as spotlights or strip lights.

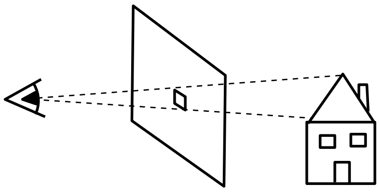
We require a model for the light. This will incorporate

- the position, intensity and colour of the light source or sources
- how incident and reflected specular light are related
- how incident and reflected diffuse light are related
- the contribution from ambient diffuse light

We also need to know the position and viewing direction of the viewer to complete the calculations.

We shall simplify and assume a single point light source. If there are more than one, the contribution from each can be computed and added. Similarly, each light will have a colour composed from red, green, and blue: we can compute each component individually and add the results.

We apply the computations for each pixel on the screen.



3.6.1 Surface Light Effects

There are several surface effects that can be individually computed, and then summed to form the result.

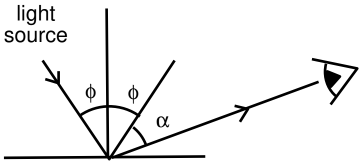
Specular Reflection

This is the cause of highlights on polished or shiny materials like glass and metal. It is

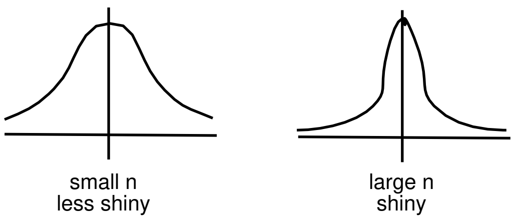
- highly directional
- the same colour as the incident light

You may like to think of it as being caused by the polish or varnish on the surface of an object, and not the object itself. For example, think of a black snooker ball: you can see highlights even though the ball is black. The perfect example of specular reflection is a mirror: you do not see the surface at all!

A simple approximation to the directionality is to use $\cos^n \alpha$, where α is the (positive) angle between view direction and ϕ the direction of perfect reflection.



The larger the n, the more rapidly the reflected light drops off as we move away from the angle ϕ, and the smaller the highlight patch. Thus large n correspond to highly shiny surfaces. Typical values are $n = 1$ for a slight sheen to $n = 20$ for polished metal.



Hence:

$$I_s = I_p K_s(\phi) \cos^n \alpha$$

is the *specular component* of the reflected light. Here I_p is the intensity of the source light, and K_s is the specular reflection coefficient: this is a function of ϕ to enable us to model material whose shininess varies with direction. Typically, $K_s = 1$ for $\phi = 90°$, and $K_s < 1$ otherwise. It is often set to a constant, values for which can be looked up in books for various actual materials.

There is just one specular coefficient (not one for each of red, green and blue) as all colours of light are reflected in the same way (otherwise specular colour would be affected by the colour of the surface).

Diffuse Reflection

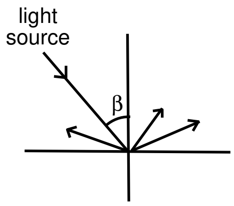
Matt surfaces scatter light evenly in all directions. Thus the amount of light seen is independent of the viewer's direction. Its colour is that of the surface: indeed this is what we mean by the (perceived) colour of an object.

- direction independent
- colour that of the surface

A surface perpendicular to the lighting direction will scatter most light; one parallel will scatter no light. The is captured in *Lambert's cosine law*:

$$I_d = (I_p - I_s) K_d \cos \beta$$

is the *diffuse component* of reflected light. The values of I_p and I_s are as above, while K_d is a constant dependent on the material ($0 < K_d < 1$), and β is the angle between the surface normal and the incident light.



For a shiny surface $I_p - I_s$ is the amount of light that is not specularly reflected. It is as if the surface has a shiny upper layer and a matt lower layer, so that the non-specularly-reflected fraction of light is scattered by the matt layer. For a purely matt surface, $I_s = 0$.

We repeat this for each of the red, green and blue components of the light, with possibly different K_d for each.

Ambient Lighting

The basic illumination described here produces very strong contrasts since all surfaces facing away from the light source will be completely dark. It is common practice to assume that the environment scatters some light uniformly everywhere over all objects, and we model this by adding a fixed amount of diffuse (non-directional) ambient light $K_a I_a$, where I_a is the incident ambient light, and K_a the reflection coefficient for this light. Again there are red, green, and blue components.

Autophanous Facets

Some parts of the model can be self-luminous, e.g., street lights. These are not subject to the usual lighting computations, which are designed for light scatterers, not emitters. Usually, we just note them in the model, and render them assuming constant output.

Furthermore, we do not consider their effect on the rest of the environment (at this stage), as that can be quite complex.

Total Lighting

We should assume an inverse square law for light sources. In practice it is common that the light sources are well out of the scene, and effectively at infinity, so that the actual light distribution is uniform. Otherwise we would have to factor in the inverse square effect, which is quite costly.

A typical lighting model combines diffuse reflection (ambient and scattered) with some fraction of the specular component

$$I = K_a I_a + I_d + k I_s$$

The extra parameter gives us more flexibility in choosing the appearance of the object.

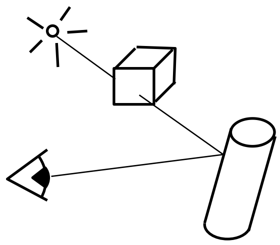
3.7 Optical Effects

The basic ray tracing idea can be extended to give a variety of optical effects that are quite difficult to get from a surface renderer.

3.7.1 Shadows

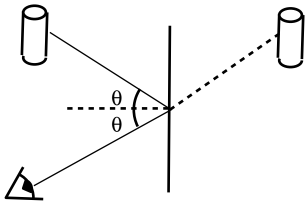
Shadows are easy with ray tracing. Once we have the intercept of the ray with the object, we can cast a ray from from the intercept to the light source. If there is an object in the way, we are in shadow, and we can darken the relevant pixel.

This also works for multiple light sources: cast a ray back to each source, and sum the results.



3.7.2 Reflection

Mirrors are easy to simulate. Simply reflect the ray, and continue from the new start point and direction.



Angle of reflection is the angle of incidence.

Generally a mirror is not 100% reflective, so we tend to reduce the intensity of the ray a little on each reflection. This also fixes the problem of infinitely reflecting mirrors by choosing a cutoff point where we don't bother any more. Another fix is to count the number of reflections, and stop when they exceed a threshold.

A mirror can affect the colour of the ray, too, by reflecting different colours to different intensities, or adding a colour component to the light.

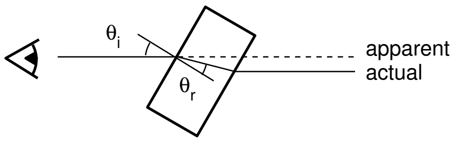
3.7.3 Transparency

Such as through glass. The simplest model is to ignore refraction, and just modify the colour components according to the material. This is OK for very thin pieces of glass (or other transparent material). There are various models of how colour might be affected: we can regard the material as a filter that selectively transmits different colours to different extents (thus modifying the red, green and blue components of the intensity of the ray).

3.7.4 Refractive Transparency

This is a more physically correct model of transparency, and looks good when the glass is more than vanishingly thin.

For refraction we use Snell's law to compute the new direction of the ray.

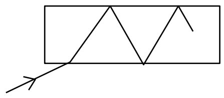


Snell's law: if θ_i is the angle to the normal of the incident ray, and θ_r is the angle of the refracted ray, then

$$\frac{\sin\theta_i}{\sin\theta_r} = \frac{\eta_r}{\eta_i},$$

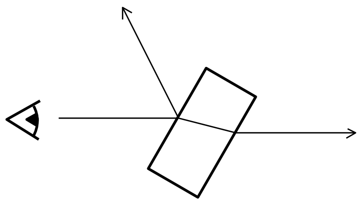
where the η are the *refractive indices* air and glass. The refractive index can vary with wavelength (colour), so the angle of refraction can vary with wavelength. In vacuum $\eta = 1$, for other materials $\eta > 1$ (so light is always bent towards the normal on moving into glass, and away when moving out).

What happens if the ray inside the glass hits the boundary at a larg angle, that is, it is close to the plane of the glass? Then we might have $\sin\theta_r = \eta_i \sin\theta_i/\eta_r > 1$, as $\sin\theta_i$ is close to 1. This is not possible for a real angle θ_r, so Snell's law breaks down. Instead we get *total internal reflection*, where the ray is actually reflected back into the glass.



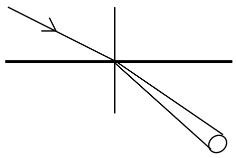
This is the effect that allows optical fibre to work. We treat the reflections just as for mirrors.

We can include the slight reflective effect of the glass: this generates a new ray that can be traced off to its destination. Potentially, this splits again as it hits another piece of glass, etc., so we may get a large number of rays bouncing about that were spawned from the single original! However, we make the intensity of each subsequent ray get smaller and smaller, so we can cut off as before.



3.7.5 Translucency

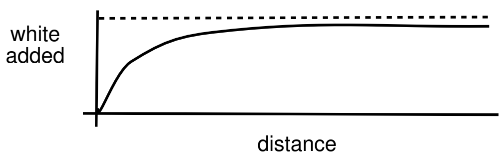
This is diffuse transmission, such as through frosted glass. Objects seem through such materials are blurred. For this, calculate the reflective lighting contribution (possibly including the colour of the material), and add it to the result of letting (a fraction of the) the ray continue.



The blurring can be modelled by *specular refraction*: the ray splits into a cone of light as it refracts. We ray trace this by using a sample set of rays lying in the cone.

3.7.6 Atmospherics

For outdoor scenes we may wish to allow for atmospheric scattering to give a sense of depth. This might be mist or smoke. This is done by exponentially desaturating (adding white) the colour with distance. At some point the colour becomes indistinguishable from white (the colour of the mist), and the object can no longer be seen. The rate of the exponential governs the depth of the mist.

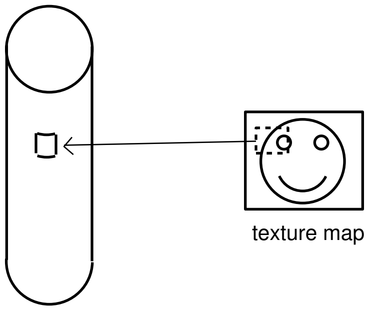


3.7.7 Surface Detail: Texture Maps

Real surfaces are not completely smooth and lacking in detail: they have a texture, which can range from being quite subtle to very strong. There are three common methods for producing a textured surface, *texture maps*, *bump maps*, and *displacement maps*.

A texture map regards surface texture as a modification of the surface reflection coefficient. The surface itself is still quite flat, but we can get the illusion of both texture *on* the surface (such as woodgrain), and texturing *of* the surface (such as the tread of a tyre).

A texture is defined by providing a 2D array of colour values. This can be synthetic or a scanned real image, such as an area of a wood table top. We also define a regular mapping from this pattern onto the surface being textured. An easy mapping is a regular square tessellation.



In practice, for each pixel we determine what area of the texture map it corresponds to, average the colour within that area, and use that to modify the surface colour. Then we apply lighting as usual.

This produces quite a nice effect if done properly.

3.7.8 Bump Maps

This is a variant on texture maps that use the texture pattern to hold normal offsets, rather than colours. These are added to the local normal on the surface before the lighting model is applied. This gives the appearance of a non-flat surface, such as a tyre tread. This is a bit of a fudge, as we are not actually changing the flatness of the surface, merely drawing it as if it were non-flat. As long as the viewer is not too close, and you cannot see the surface in silhouette the effect is good.

3.7.9 Displacement Maps

This is a technique much better suited to surface modelling, but we mention it here for completeness. A more costly approach to adding surface detail is to really displace the surface: use the texture map to actually move points on the surface. This is quite tricky to do for a solid model defined by equations. The result is a fully correct texture, but at considerable cost.

3.7.10 All the Above

We can combine all the above effects in a single picture, and the results can be quite impressive. The cost is in the multiple rays we have to trace (remember this is for just one pixel), and we can get into *recursive ray tracing*, where one ray generates several as it splits and reflects.

3.8 Coordinate Systems

So far we have assumed that there is some sort of ambient coordinate system that we have used to fix our models. In practice it turns out to be too restrictive to have a single coordinate system: suppose we have a model containing a chair, and we want to draw two identical chairs in the picture. One way would be to incorporate each chair individually, but this is a waste of effort, duplicating all the details. Similarly we might want a big chair next to a small chair, or an upside down chair, and so on.

What is useful is to have a collection of coordinate systems all linked together in a way that is convenient. Each stage of the modelling and rendering process can use a coordinate system that is convenient for that particular stage: for example, the modeller can build their world in a coordinate system of their choice, not one that is foist upon them by the graphics package they choose to render the picture.

All systems will be Cartesian.

(a) Object Coordinates (3D). Each object is defined in is own coordinate space. E.g., a cube, a sphere, etc.

(b) Model Coordinates (3D). We transform and combine those objects to produce the complete model we want to view.

(c) World Coordinates (3D). To generate a picture we may want to use a graphics library, such as OpenGL. The library routines will work in their own prefered coordinates, possibly definable by the user.

(d) Normalised Device Coordinates (2D). The output from the package will probably be a bunch of floating point numbers in the range $[0.0, 0.1]$ for each (x, y), corresponding to some hypothetical output device.

(e) Device Coordinates (2D). These numbers will be converted to other numbers (often integers, e.g., pixel locations) to address a specific output device, be it a screen or a printer or whatever.

A modeller might choose a system where coordinates vary from 0–100, while a graphics renderer might prefer a value range of 0.0–1.0, or -1.0–1.0, say. Each can use their preferred system, and it just takes a simple transformation (e.g., divide by 10) to move between the systems. Furthermore, a modeller can create their model even if they don't know which renderer will be used.

As part of (c) we usually define a *window* (in 3D world coordinate), which is a rectangle through which we look. This limits what we can see. We can also have a *viewpoint* and a *view direction* to determine from where we see and in what direction we look.

Similarly in (d) we can define a *viewport* on the screen (normalised device coordinates) to indicate where on the screen the picture will appear.

Notes.

- These systems may or may not be distinct, e.g., model coordinates might (just by chance or by design) be the same as world coordinates.

- This is just one way of organising coordinate systems: other people might do things differently, or name the systems differently.
- (a) and (b) are part of the modelling process, while (c) to (e) are part of the viewing process. These are quite separate and are easily confused. Modelling is often application specific, and can take the larger part of the programming effort. The rest can be handled with standard graphics packages like OpenGL, GKS, and others.
- It can help to think of the window as the position of the "camera", and the viewport as where on the page we glue the resultant picture.

3.9 Speeding up Ray Tracing

Ray tracing is inherently very slow. It requires much floating point computation for each pixel, and a traversal of the CSG tree for each pixel. However, because each pixel is computed independently, there is scope for ray tracing to be done in parallel: in practice the large amounts of data that needs to be interrogated makes parallelism less than 100% successful. We shall now consider some more basic ways of improving speed.

3.9.1 Using Object Coordinates

A scene will typically have many instances of a particular primitive. Each will have been separately transformed to fit into the model. This means the defining equation of the primitive will be quite complicated when expressed in model coordinates. Rather than try to compute intersections in model space, it is simpler to transform the ray back into object space, and solve the intersection there. For example, a general sphere might be given by $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$ in model space, but represented as $x^2 + y^2 + z^2 = 1$ in object space. Solving an intersection of a straight line with the latter is numerically much simpler.

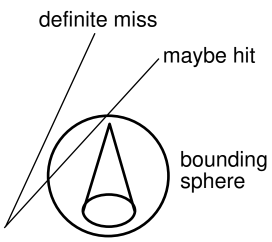
Once we have found the intersection in object coordinates, we transform it back to model coordinates.

3.9.2 Volumes of Interest

This is much like clipping in surface models. We want to find a way of rapidly rejecting primitives which are well away from the current ray, e.g., objects that are behind the point of view. We would then avoid computing putative intersections with surfaces that cannot possibly be involved.

A simple way of doing this is to enclose each primitive with a imaginary sphere (the *bounding* sphere), or box (the bounding box). We first test if the ray intersects the sphere. This is a simple and fast test. If the ray misses the sphere we know it cannot hit the object inside, no matter how complicated it is. This is a fast *culling* operation.

If the ray hits the sphere, we proceed as before, but we hope this happens infrequently.

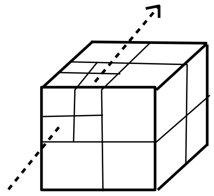


3.9.3 Octrees

This is a recursive subdivision method of producing volumes of interest. We imagine that the entire model is held in an enclosing cube. We recursively subdivide the cube into eight smaller cubes (*voxels*), resulting in an *octree* structure. We keep subdividing until the contents of each voxel is “simple”, e.g., entirely solid, or entirely air, or we hit some limit on voxel size. In particular this allow us to identify empty space, and home in on the surface of the solid elements.

Ray tracing now consists of walking a ray through the octree voxels until we hit a non-empty voxel. We then solve for the intersection for the object(s) in this voxel. In this way we avoid solving for intersections with objects that lie along the ray but will not be seen.

The cost is the storage of the octree, and the tracing of a ray through it.



4 Surface Modelling

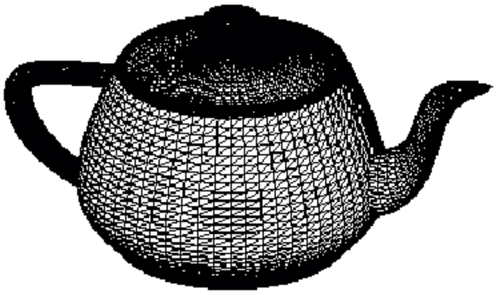
To draw a picture, such as a frame from an animation, or a architectural plan of a building, we must first start with some model within the computer. This model will contain all the information we require to make the picture, such as what objects are in the picture; what colours they are; what textures, reflectivity, and so on.

We need to consider how to store this kind of data in such a way that it is convenient for the rendering process. We shall look in detail at two popular ways, surface modelling and solid modelling. The kind of model strongly affects the rendering process in that certain kinds of models tend to be rendered in certain ways, but this connection is not absolute.

4.1 Surface Models

A object is stored in terms of its *surface*. That is, a sphere is represented by its surface; a chair is represented by a collection of planes or whatever.

Usually we use a *mesh* representation: a mesh consists of a (large) number of flat polygons, usually triangles, that map out the surface in question. Note that each polygon is flat, so this will be an approximation to a curved surface, but it is easy to make an approximation as good as we require simply by making the polygons smaller.



Newell's Teapot (the Utah Teapot)

4.2 Data Structures

All good computing courses require an investigation into datastructures. In graphics, it is essential to have good datastructures in order to have good performance. We will look at several ways to store a mesh model, and discuss how well they are suited to rendering.

A mesh is composed of many facets, each a polygon. We shall restrict ourselves to triangular facets. The advantages of triangles are

1. any 3 (non collinear) points define a plane, even in the presence of rounding error: with four or more points the facets we describe using floating point numbers may not be quite flat, and edges may not line up properly
2. triangles are convex: non convex polygons introduce extra complication later in hidden surface removal

We would like a datastructure that is easily modified, as the typically we want to manipulate dynamically a line drawing of the mesh, grabbing and moving edges and vertices interactively. We must allow for such alterations.

4.2.1 Simple Vertex List

Define all vertices v_i by a list of their 3D coordinates. Define all edges by a list of (v_i, v_j) pairs.

This is enough to draw our image, as we can run along the edge list, drawing a line between the vertices.

It is useless for interaction: we cannot relate edges to polygons, or vertices to edges. Thus, if a user picks a vertex and want to move it, we would have to search the edge list to find the edges the vertex belongs to. In a model containing 10000 edges this is woefully slow.

Being unable to determine polygons means that we are unable to produce a solid shaded picture, or determine which parts of the object are obscured by other parts.

4.2.2 Polygon Vertex List

Identify each polygon in the model as a list of vertices.

$$P = [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots (x_n, y_n, z_n)].$$

Edges are implied by the order of the vertices in the list. So now we can determine polygons easily. However there are problems.

1. Each edge is doubly represented, being in the lists for two polygons. If we want to move an edge, we have to make sure we update both copies consistently.
2. Vertices are similarly duplicated.
3. We can only find edges and vertices by searching all the data. So "find the vertex closest to these coordinates" will be very slow.

4.2.3 Polygon Pointer List

List all vertices once. Define polygons as pointers into this list. E.g., polygon P_2 might be vertices $[2, 5, 7]$.

Now vertices can be moved easily, but we still have difficulties with edges.

4.2.4 Edge List

We put all the above together:

Polygon list	Edge list		Vertex list
edges	polygons	vertices	coordinates
1: 1 2 3	1: 1	1 4	1: x_1 y_1 z_1
2: 3 4 5	2: 1	1 2	2: x_2 y_2 z_2
	3: 1 2	2 4	3: x_3 y_3 z_3
	4: 2	2 3	4: x_4 y_4 z_4
	5: 2	3 4	5: x_5 y_5 z_5

Polygons are pointers into the edge list. Pointers are held in (say) clockwise order as seen from the front, so we can easily determine the front and back of a facet.

Edges are pointers to the containing polygons, and to their vertices.

Vertices are kept just once.

This allows us to add, delete and change vertices, edges and polygons quite flexibly.

4.3 Transformations

Suppose we have a collection of primitive objects (each in their own object coordinates), such as a cylinder and a torus. How can we combine these to model a bicycle wheel? We can think of a wheel as a large thin torus, with many long thin cylinders within it for the spokes.

First we *scale* the objects to the correct size, *rotate* to the correct (3D) orientation, then *translate* to the final destination. In principle we can do these transformations in any order, but it is far easier to use this order (translate last, certainly). This is because scale and rotate are origin-centred transformations, and doing them first reduces the computation significantly.

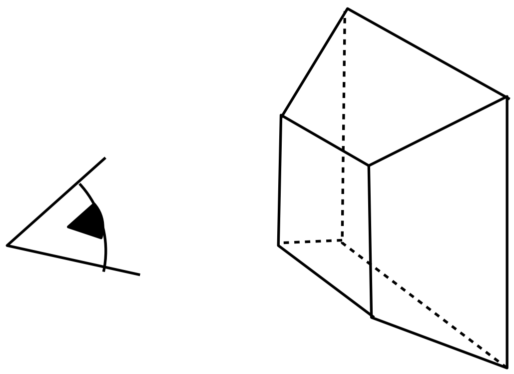
Also it is convenient to scale first: if we want to scale in different amount in the x and y directions, it is better to do that before we play about with rotating the direction of these axes.

One more transformation is needed: *clipping*. This means throwing away those parts of the model that cannot be seen because they are outside the window.

4.3.1 Clipping in 3D

Clipping is very useful for large models: if we can discard (i.e., not bother to consider) half the model because we know it cannot be seen from this window, then we have halved the amount of computation necessary. Often clipping saves a huge chunk of time.

But it's not quite so simple in 3D:



The volume that the eye can see is a truncated pyramid. The near plane is the screen, the far plane at some convenient distance.

Clipping is usually done by a fast subdivision method: for each edge in the model we check against a clipping plane and find whether both ends are

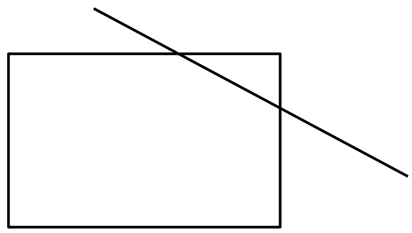
(a) both outside the viewing volume: throw it away

(b) both inside the viewing volume: (potientially) draw it

(c) one inside one outside: bisect the line and recurse on each half

Testing for inside or outside is a matter of plugging the coordinates into the equation for the clip plane, and seeing if the result is positive or negative.

We have to test against each clip plane individually and combine the results. It could be that the ends of an edge are outside the viewing volume, but the middle of the edge intersects



An alternative to bisection is to compute the intersection of the edge with the clip plane, and then each sub-edge will fall into category (a) of (b).

4.3.2 Translation in 3D

Rotation and scaling have convenient matrix representations: we can devise matrices that rotate of scale coordinates in a multiplicative way. However, translation is different from rotation and scaling in that it is an additive process. It would be nice to have a single transformation mechanism that combines all three in a uniform manner, so that we can compose any of the three in arbitrary combinations.

The solution is to use *homogeneous coordinates*. Instead of using the simple triple (x, y, z) we use (wx, wy, wz, w), where w is any non-zero number. From the homogeneous form (x, y, z, w) we can retrieve the original as $(x/w, y/w, z/w)$ since $w \neq 0$. Often $w = 1$ is chosen, so we have a direct correspondence

$$(x, y, z) \leftrightarrow (x, y, z, 1).$$

The advantage of this complexification is that we can use 4D matrices to represent 3D translation, scaling and rotation in a uniform way (N.B. all are relative to the origin).

Suppose we want to translate by (t_x, t_y, t_z). We can represent this in matrix form as

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

so that a point (x, y, z) transforms as

$$(x\ y\ z\ 1)T = (x + t_x\ \ y + t_y\ \ z + t_z\ \ 1).$$

The inverse of T is simple

$$T^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{pmatrix}.$$

4.3.3 Rotation in 3D

Rotation in 3D is a lot more complicated than rotation in 2D. Before we start we have to determine some conventions.

In 2D, a positive rotation of an angle θ is given by a matrix

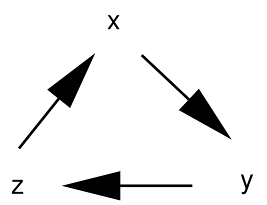
$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix},$$

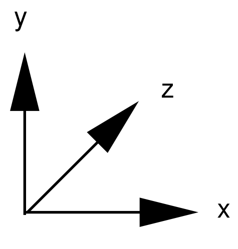
where, by convention, positive is anticlockwise. Notice that rotation is about a *point* in 2D, but about an *axis* in 3D.

Similarly, in 3D, we need to determine which way is positive, and we need to do it for all 3 axes.

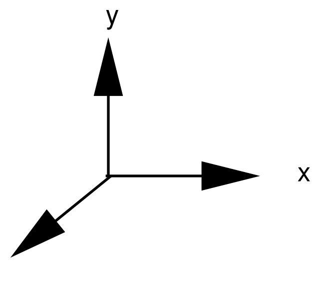
rotation axis	direction of positive rotation
x	from y towards z
y	from z towards x
z	from x towards y

This is easy to remember as x, y, and z are cyclic.





Left handed



Right handed

In right-handed systems these are anticlockwise rotations. In left-handed systems they are clockwise. Being clock- or anticlockwise is determined by looking along the axis, from a positive value towards the origin.

This definition gives us "positive" regardless of handedness of the coordinates: thus matrices are also independent of handedness.

Here are the matrices for rotations about the three major axes.

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

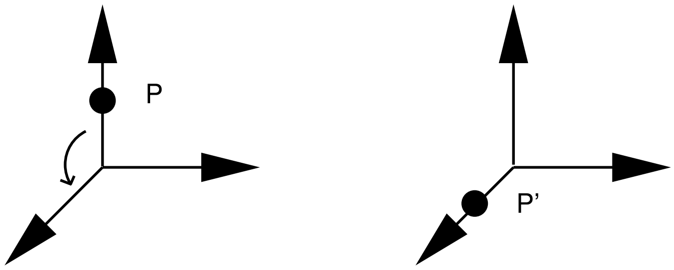
$$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The inverses of these matrices are their transposes: equivalently replace θ by $-\theta$ everywhere.

Notice the similarity between 2D and 3D rotations: this is because the familiar 2D rotation is really a rotation about the z axis.

Example. Let $P = (0, 1, 0)$ in a right-handed system. Apply a rotation about the x-axis of $+90°$.



The matrix is

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus, in homogeneous form

$$PM = (0\ 1\ 0\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (0\ 0\ 1\ 1),$$

as expected.

4.3.4 Scaling in 3D

This one is easy. Here is the homogeneous matrix for scaling s_x in direction x, etc.

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This has inverse

$$S = \begin{pmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

as long as all of the ss are non-zero. A zero s corresponds to flattening an object into a lower dimension.

4.3.5 Combining Transformations

We can compose translations, scalings and rotations as we please. We will end up with a matrix of the form

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}.$$

The rs represent scaling and rotation, while the ts represent a translation.

For modelling, we scale, then rotate, then translate. For *viewing* we use the same transformations, but in the opposite order, namely translate, then rotate, then scale. This is because we are moving the *viewer*, not the object: moving the viewer in one direction is as if we are moving the object in the opposite direction.

For example, suppose we have define an object, and we wish to look at it from different directions. This way viewing and tumbling an object is quite popular. Suppose we have some axis that we wish to rotate about: recall that transformations are relative to the origin. So we have to translate some point P on the axis to the origin, rotate, then translate P back:

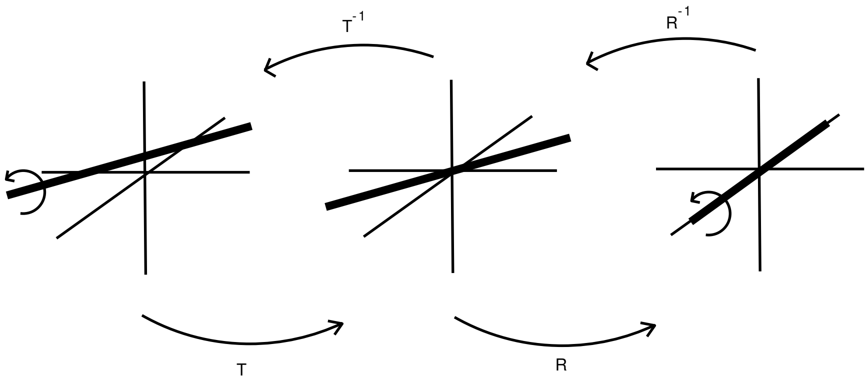
1. translate by $-P$
2. rotate about desired axis
3. translate by P

Thus far, so good. But we only know how to rotate about the x, y, and z axes: how do we rotate about another axis?

4.3.6 Rotation about Arbitrary Axes

The solution is to reduce the problem to one we have solved already, viz., transform the model until our desired axis of rotation lies along (say) the z axis, rotate about z, then transform back.

1. translate so the arbitrary axis passes through the origin
2. rotate the model about one or two axes until the arbitrary axis lies along the z axis
3. rotate about z the desired amount
4. invert step 2
5. invert step 1



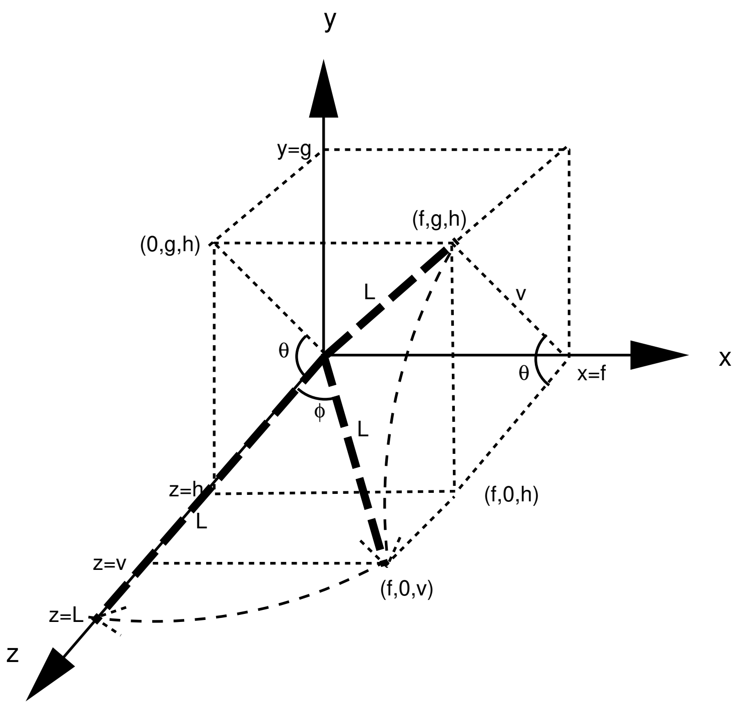
So we need to determine the matrices to do this. Suppose the axis we want to rotate about lies on (x_1, y_1, z_1) in the direction of (f, g, h). So the line has parametric equation

$$x = x_1 + ft, \; y = y_1 + gt, \; z = z_1 + ht.$$

Thus the translation we need is $(-x_1, -y_1, -z_1)$ with matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 0 \end{pmatrix}.$$

The picture now looks like:



We now rotate about the x axis by an angle θ until the axis lies in the xz plane. Now we do not actually have to determine θ, as the rotation matrix only requires $\sin\theta$ and $\cos\theta$, and these are easy. The diagonal $v = \sqrt{g^2 + h^2}$, so

$$\sin\theta = g/v, \;\; \cos\theta = h/v.$$

This gives us the rotation matrix (rotating about x)

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & h/v & g/v & 0 \\ 0 & -g/v & h/v & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This is particularly nice as we don't have to compute any expensive trigonometric functions.

So the axis now lines in the xz plane. Next we are going to rotate about the y axis to bring it in line with the z axis. The diagonal of the triangle is $L = \sqrt{f^2 + g^2 + h^2}$. Rotate about y by ϕ (negative)

$$\sin\phi = f/L, \;\; \cos\phi = v/L,$$

giving the matrix

$$R_y = \begin{pmatrix} v/L & 0 & f/L & 0 \\ 0 & 1 & 0 & 0 \\ -f/l & 0 & v/L & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can now rotate about z by the requested amount (say ψ)

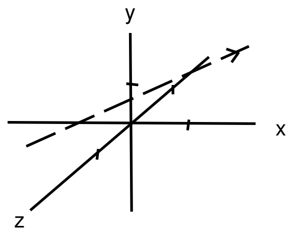
$$R_z = \begin{pmatrix} \cos\psi & \sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now we restore everything back to is starting place: apply R_y^{-1}, then R_x^{-1}, and then T^{-1}.

Combining all these together we get

$$R = TR_xR_yR_zR_y^{-1}R_x^{-1}T^{-1}.$$

Example. Rotate about the axis through $(1, 1, 1)$ in direction $(1, 1, -1)$.



The line has parametric form $(1 + t, 1 + t, 1 - t)$, and passes though $(0, 0, 2)$, $(1, 1, 1)$, $(2, 2, 0)$, etc. In homogeneous form this is $(1 + t, 1 + t, 1 - t, 1)$ First translate

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{pmatrix}.$$

The line transforms to $(1 + t, 1 + t, 1 - t, 1)T = (t, t, -t, 1)$, which passes through the origin. Next rotate about x. We get $v = \sqrt{2}$, so

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & -1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The line becomes $(t, t, -t, 1)R_x = (t, 0, \sqrt{2}t, 1)$, which line in the xz plane. Rotate about y. We get $L = \sqrt{3}$, then

$$R_y = \begin{pmatrix} \sqrt{2/3} & 0 & 1/\sqrt{3} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{3} & 0 & \sqrt{2/3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The line becomes $(t, 0, \sqrt{2}t, 1)R_y = (0, 0, \sqrt{3}t, 1)$, which runs along the z axis.

Finally, rotate about z by ψ, R_z is as above. Combining all of these and their inverses,

$$\begin{aligned} R &= TR_xR_yR_zR_y^{-1}R_x^{-1}T^{-1} \\ &= \frac{1}{3}\begin{pmatrix} 2\cos\psi & -\sqrt{3}\sin\psi - \cos\psi + 1 & -\sqrt{3}\sin\psi + \cos\psi - 1 & 0 \\ -\cos\psi + \sqrt{3}\sin\psi + 1 & 2\cos\psi + 1 & \sqrt{3}\sin\psi + \cos\psi - 1 & 0 \\ \sqrt{3}\sin\psi + \cos\psi - 1 & -\sqrt{3}\sin\psi + \cos\psi - 1 & 2\cos\psi + 1 & 0 \\ -2\cos\psi - 2\sqrt{3}\sin\psi + 2 & 2 + 2\sqrt{3}\sin\psi - 2\cos\psi & 4 - 4\cos\psi & 3 \end{pmatrix}. \end{aligned}$$

5 Surface Rendering

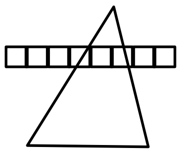
We now have the preliminaries settled: how to build a model, and how to transform it. Next we have to discuss how to produce an actual picture.

We start from a mesh model, generally formed from triangular facets. We shall look at producing a shaded colour picture, which will involve

- lighting calculations: the use of light direction and intensity
- surface reflection calculations: different surfaces have different reflective properties
- applying lighting model to surface patches
- projecting from 3D to 2D: projective transformations
- hidden surface removal

5.0.7 Surface Reflection

We now look at the way we apply lighting models (as described for ray tracing) to the patches that form the model. We assume that the renderer works on a horizontal scan line, producing the pixels for that line before moving on to the next line. At any point, therefore, the renderer only has access to a small slice of the model.

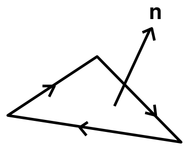


We can work out the correct intensity for every pixel as we come to it, but the result is not very good, and the facets are still visible.

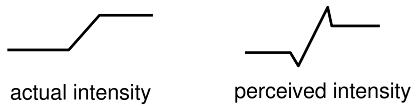
Constant Shading

This is the first and simplest method. Assign each facet a single intensity value. Thus we find the surface normal of the facet, use the diffuse computation, and colour the whole triangle with the resulting colour.

The surface normal is given by the cross product of two edges of the triangle. We arrange the model so that every facet has an orientation, so we can determine what is its front, and what is its back.



This is computationally cheap, and is good for a quick look at a model. One problem is that the individual triangles of the mesh are all visible, and something worse. There is an effect, the *Mach band effect* that the eye perceives sharp intensity changes as dark and bright bands. This makes the edges look much worse.



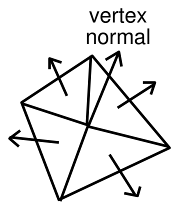
This shading does at least reveal the real structure of the model: it really does consist of triangles.

Method Calculate the surface normal, then the cosine of the angle between it and the lighting direction (from the dot product of the directions: we don't ever need to know the actual angle itself). Shade proportional to this.

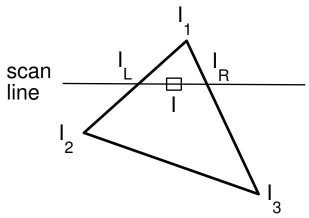
Intensity Interpolation

Also known as *Gouraud shading*. This avoids the intensity discontinuity on edges (but the intensity derivative is not continuous). It gives the impression of curvature, and is widely used to help display curved objects that are really composed of flat facets.

Method Calculate the surface normal for each facet, then vertex normals by averaging the surface normals for each facet meeting the vertex.



Calculate intensities for the vertices with your chosen shading model. For each intersection of the triangle edge with the scan line linearly interpolate the intensity between the vertices at the end of the edge. Interpolate across the scan line between these edge intensities.



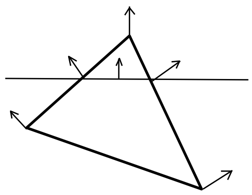
Thus, if we have computed intensities for the three vertices of this triangle as I_1, I_2 and I_3, we can compute I_L as the interpolation between I_1 and I_2, while I_R is interpolated between I_1 and I_3.

Then the intensity I for a pixel is interpolated between I_L and I_R.

- Gouraud smoothes across edges, which is generally the thing you want. If you want an edge to be visible, you need to mark this in the datastructure, and you can calculate two vertex normals, one for each side of the edge.
- The shading is continuous, but the derivative is not continuous. This can sometimes be seen as Mach bands, particularly during motion.

Normal-Vector Interpolation

Also known as *Phong shading*. This is similar to Gouraud, but instead of computing and interpolating intensities, you interpolate the normal vectors, and compute the intensity for each pixel.



This is particularly good in conjunction with specular reflection, models with highly curved surfaces (many small patches), and it gives quite good moving pictures. Mach banding is much reduced, but the cost is high.

Note that there are incremental versions of Gouraud and Phong that are more efficient than the above. Also, these days you can get hardware support for Gouraud and Phong shading: they run at hundreds of thousands or millions of fully shaded triangles per second.

Texture maps are increasingly supported in hardware, too. This is why fast moving action games use surface rendering in preference to ray tracing: they can get speedups of ten or hundreds times by using purpose-built hardware.

5.1 The Projection Calculation

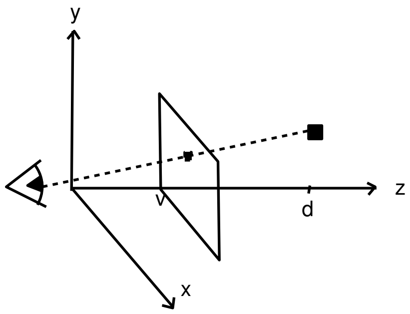
Here we describe how the 3D model is reduced to a 2D picture on the screen. Later we give a proper mathematical treatment of projection, but for the moment we give a simple overview.

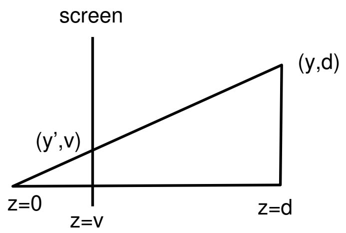
There are many ways of projecting into 2D, we shall look at the two most popular.

5.1.1 Perspective Projection

This is the kind of projection that we get from a conventional camera (with a normal lens). Objects further away from the viewpoint are smaller as we might expect, but such a projection does not leave angles or parallel lines invariant. That is lines that are parallel in reality may appear to converge in the projected picture.

This gives a good sense of depth to the picture, and is close to our everyday experience.





We want to find (x', y'), the screen coordinates of an object at (x, y, d) in the model. The derivation of x' and y' is identical, so we need only consider one, y.

We have a viewer at the origin, a screen at $z = v$, and a point in the model at (y, d). We want to find y'. By similar triangles,

$$\frac{y'}{v} = \frac{y}{d},$$

or

$$y' = \frac{yv}{d} = \frac{y}{d/v}.$$

Thus we are dividing by depth, v acts as a scale factor. So objects at a greater distance (larger d) appear smaller, and the rate at which objects diminish is size with distance is governed by the viewing distance v.

If v is small, the eye is close to the screen and we have a wide angle of view (like a wide-angle camera lens), the perspective is acute, so close objects appear unnaturally large by comparison with further objects. This is also true of normal vision by the eye, except the brain convinces us that things look normal. When we look at a photograph taken this way we can see the apparent distortion.

Note that zooming in to get a close-up view of a distant object (moving the screen forwards) is not the same as moving the viewpoint forwards.

The matrix for the perspective transformation:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/v \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Thus a point (x, y, z) becomes

$$(x\ y\ z\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/v \\ 0 & 0 & 0 & 0 \end{pmatrix} = (x\ y\ z\ z/v),$$

which corresponds to the real point $(xv/z, yv/z, v)$ when we normalise.

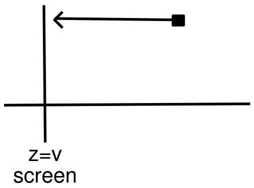
Note that

- the object is flattened to 2D, and lives on the $z = v$ plane

- we must clip (discard) points which are very close to the viewer, else we risk getting overflow when dividing by small numbers
- we must clip points with $z < v$ as these will also project on to the screen, including objects behind the viewer!

5.1.2 Parallel Projection

This kind of projection is popular with engineers and architects: it preserves angles and parallel lines, and so is good for when you want to make things. It is a limiting form of perspective projection as we take v to infinity.



This is a very easy transform, we simply drop the z coordinate.

$$\begin{aligned} x' &= x \\ y' &= y \\ z' &= v \end{aligned}$$

We still get a bit of foreshortening, but this is not a function of distance. Objects remain the same size on the screen regardless of their distance in the model.

The matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & v & 1 \end{pmatrix}.$$

5.2 Hidden Surface Calculation

In a 3D model, some objects may be behind others and thus will be partly or wholly obscured from a given viewpoint. When producing a realistic picture we need to determine those parts of the model that can't be seen, and not draw them.

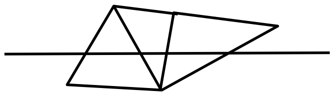
There are many ways to calculate what can be seen, but they really all reduce to a problem of sorting.

Our assumption here is that our model contains no self-intersecting surfaces: if there are, we can break it down into smaller triangles that don't intersect.

5.2.1 Scanline Algorithm

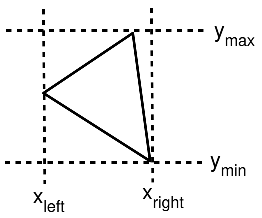
This is due to Romney, Watkins and Evans (1969), and displays a model composed of polygonal (triangular) facets on a raster display.

Consider a faceted model, and think of the way a scan line traces across it.

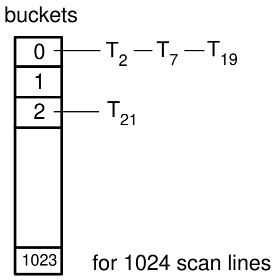


Notice that the picture remains the same (in terms of the visibility of facets) except when the scanline crosses an edge. This observation leads to the following algorithm.

1 Remove all backwards facing facets (recall that facets have an orientation given by the direction of their edges). Such triangles cannot be seen from the viewpoint. Incidentally, we regard facets as having zero thickness, so to model concave objects like a cup we have meshes for both the inner and outer surfaces.



2 Now compute the y_{min} and y_{max} for every triangle, and sort into *buckets* with $y_{\mathrm{min}} = 0$, $y_{\mathrm{min}} = 1$, etc. Within a bucket, sort the triangles left to right using x_{left}.



3 Create an *active list* initialised to bucket list zero. The active list will be the list of triangles that intercept the current scan line. We scan line 0, then 1, etc.

4 Draw the current scan line (see below).

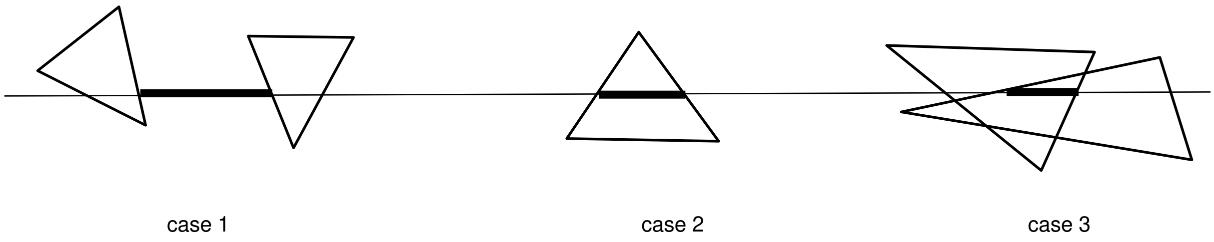
5 Move to the next scan line: update the active list by

(a) removing all polygons whose y_{max} is now less than the current y

(b) merging in x_{left} order those triangles in the next bucket (whose y_{min}s say they are now on the current scan line).

6 Repeat until all scan lines done.

We just need to show how to draw a scan line. We work along the active list, considering the intercepts of the scan line with the edges of the triangles. This produces a list of line segments over each of which the visible surface does not change. There are three cases.



1. The line segment lies outside all triangles. Output a span of the background colour.
2. The line segment lies within exactly one triangle. Output the colour of the triangle.
3. The line segment lies within two or more triangles. For each of these triangles compute its z depth at $(x_{\text{left}}, y_{\text{scan}})$, and output the colour of the closest triangle. At this point we rely on the non-intersection of triangles.

Notes.

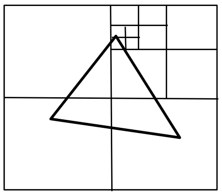
- The computation of the z depths can be done incrementally across scan lines.
- Case 3 is the most costly, and we try to avoid the computation is incurs. Thus, if the active list remains unchanged across scan lines we know that the hidden surfaces cannot have changed. Further, there tends to be only small changes in the active list as we proceed, so large parts of it can be dealt with as in the proceeding case.
- If we are using constant shading, the natural output of this algorithm is a (colour, length of span) pair, which is a compact representation of a picture (run length encoding).

5.2.2 Warnock's Algorithm

Here is an outline of an alternate hidden surface removal algorithm to indicate a completely different approach.

This method relies on the *coherence* of a picture drawn from a polygonal mesh. This means that locally the picture does not change very much: neighbouring pixels tend to be from the same facet.

Method. Recursively quarter the screen until either (a) each quarter no edges, or (b) the quarter covers just one pixel.



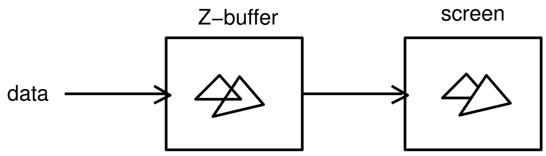
In either case, compute those triangles that cover the quarter, find the closest, and draw its colour over the entire quarter. The time taken for this algorithm is proportional to the screen resolution.

5.2.3 The Z-buffer Method

As with many graphics problems, there is a hardware solution to hidden surface removal, too. This is often implemented in graphics workstations.

This involves extra hardware, the *Z-buffer*. This is an area of memory with a location for each pixel on the screen. In this memory we store the z depth values of each pixel as we draw it. When is pixel is rendered, we check in the Z-buffer: if the depth of the pixel is greater than the value stored, we do not draw the pixel to the screen (the surface is obscured by something closer). If the new pixel is closer, we draw it and store its z value in the Z-buffer.

We initialise the Z-buffer to contain $-\infty$ for every pixel. Thus the final picture that appears on the screen show exactly those pixels that are closest to the viewer.



Shadows do not automatically appear when using surface lighting computations: either something is lit or it is dark. We can adapt the hidden surface algorithm to determine what can be seen from the point of view of the light source, i.e., what is illuminated and what is not.

One approach is to do this as the model is created, and incorporate shadows as objects in the model. This can be rendered quickly as shadow patches in the mesh are no different from any other.

5.3 Conclusion

Here is a summary of the process for surface models and scan line rendering.

We need a model; a viewpoint and viewing direction; lighting specifications. We process by

- viewing transforms
- breaking up intersecting surfaces
- back face removal
- hidden surface removal
- shading
- output to display

One more point: we have assumed in the above that the depth is measured along the z axis. Or course in general this is not true, so we will need to transform the model so that the viewpoint is at the origin and the viewing axis is along the z axis. This is a translation followed by a rotation.

Of course, much hardware in modern computers is dedicated to rendering surface models (e.g., hardware support for shading and texturing), so there are techniques to produce surfaces meshes from solid models, thus getting the best of both worlds.

6 Radiosity

Radiosity is a third way of rendering pictures. It is by far the most expensive, but produces arguably the most realistic effects being based on the physics of light.

6.1 Definition

We define *radiosity* as the energy per unit area leaving a surface patch in unit time.

Simple rendering and lighting techniques do not allow for interchange of light energy between surfaces in a scene. Thus an object next to another bright white object will be at least partly illuminated by the white object. And vice versa.

A radiosity computation for a given scene starts from an assumption of energy equilibrium (light levels are not changing), it calculates (usually approximately) how much energy is reaching each surface due to the light it is receiving from every other surface.

This allows for softer lighting effects, like colour bleeding. This is where light coming from (say) a red cube on a white table makes an area of the table look red. it also allows non-point sources of light (e.g., strip lighting). There are two important features of this approach:

- light sources are treated no differently from any other surface, except that they have a positive emission
- the equations pertaining to radiosity are solved once before the scene is rendered (e.g., using Gouraud), and the scene can be viewed from anywhere without recomputation. Thus there is an initial long computation, but after that multiple views can be produced quickly.

Radiosity is usually associated with surface rendering.

6.2 Basic Approach

Imagine a closed 3D world, perhaps the interior of a room with furniture and other objects, illuminated by several light panels. The room is is a state of energy equilibrium in the sense that for any surface (other than the light sources), the amount of energy reaching it is equal to the amount absorbed plus the amount re-radiated. This re-radiated light becomes a fraction of the light reaching the other surfaces in the scene. The total amount of light energy in the room must eventually be scattered towards the eye or absorbed.

We regard this interchange of energy as a (large) set of simultaneous energy equations to be solved to get the individual illuminations for each surface facet. Doing this allows us to model all the soft-edged lighting effects we get in a real room.

In practice, constructing and solving such a set of equations is too hard as there are far too many surface facets in a typical model. So we approximate the model to a collection of patches, each assumed to be energetically uniform. We use large patches for areas that have (near) uniform illumination, and smaller patches for more rapidly varying areas. We set up the system of equations by determining all the possible interchanges of light between each patch and every other patch, solve them, and finally render the scene using Gouraud (to blend across the edges of the patches).

One advantage of this method is we can have lights of any shape built from patches, not just point sources.

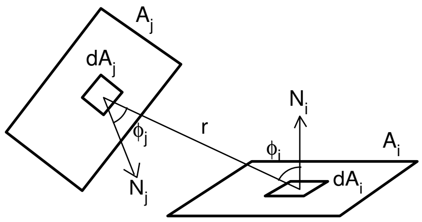
6.3 Theory

The radiosity of a patch is the sum of its emitted energy (if it is a source), and the reflected energy coming from other patches. At equilibrium

$$B_i dA_i = E_i dA_i + R_i \int F_{ji} B_j dA_j$$

where

- B_i the radiosity of patch i
- E_i is the emitted energy per unit area per unit time (light source)
- R_i is the reflection coefficient, the fraction of the incident light that is reflected
- F_{ji} is the *form factor*, the fraction of the energy that leaves patch j that reaches patch i (see later)
- A_i is the patch of interest



So $B_j dA_j$ is the radiosity (energy) leaving dA_j, while F_{ji} is the that part of it that reaches (can be seen at) patch i. We approximate this integral by a more tractable summation

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ji}$$

since, at equilibrium, $F_{ij} A_i = F_{ji} A_j$.

We can rewrite this summation as a set of simultaneous linear equations

$$\begin{pmatrix} 1 - R_1F_{11} & -R_1F_{12} & \cdots & -R_1F_{1n} \\ -R_2F_{21} & 1 - R_2F_{22} & \cdots & -R_2F_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & \cdots & 1 - R_nF_{nn} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}.$$

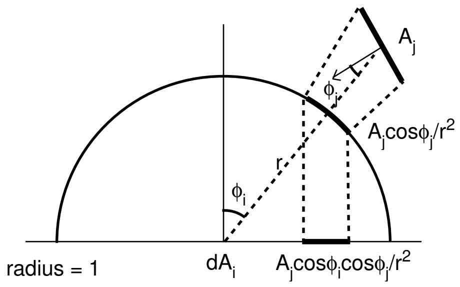
Actually, this is just one set of three: we need such a set of equations for each of red, green, and blue.

The E_i are zero except for the light-emitting patches. These represent the energy entering the system. The R_i are known from the surface characteristics. The F_{ij} must be calculated from the geometry of the scene. As patches are flat, no energy leaving a patch hits itself, so $F_{ii} = 0$. Computing the other F_{ij} is very time intensive: we need to determine what fraction of a patch is visible from another patch.

6.4 Form Factors

A numerical method to compute form factors was published in 1985 by Cohen and Greenberg, called the *hemicube method*.

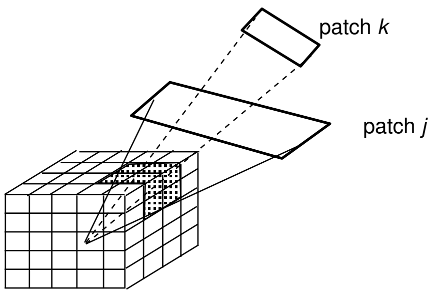
This starts by imagining a elemental area on patch i which has a hemispherical view of the world about it. We project the other patches onto this hemisphere, and use the projections to compute the form factors.



So consider a small area dA_i on patch i, and another patch A_j. We can approximate

$$F_{ij} \approx \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j.$$

Even this integral is too hard to compute, so we approximate further by replacing it with a hemicube.



Projecting onto planes is much easier. We also divide the hemicube into small cells: the more the better the accuracy, usually many hundreds. We project the other patches onto the hemicube: some patches project onto one face of the cube, others more faces. This is treated by taking one face of the hemicube at a time and clipping the patches against that face before projecting.

Also we can use hidden surface computations (z-buffering) to determine which is the nearest patch when several project onto the same area.

To fix some of the symmetry lost when we moved from the hemisphere to the hemicube, each cell on the surface is given a different weighting. A cell in the centre of a face can be seen directly at the centre of patch i, while one near an edge of the hemicube is seen obliquely. Thus extra factors are incorporated to reflect this. These can be precomputed when generating the hemicube.

Once all of this is done, we can finally compute the radiosity values. We choose a viewpoint, and Gouraud shade to blend the patches. Moving to another viewpoint is simple, we don't have to recompute the radiosity.

The expense of radiosity tends to lead people not to use it for cases other than when accuracy is more important than time. In fact most of its good features can be emulated by clever ray tracing or surface rendering techniques.

7 Colour

We now turn to look at colour and its implementation. So far we have talked about red green and blue components (as in colour TVs), but the real story is very much more complicated.

7.1 Introduction

It is not commonly realised that there are many ways to represent colour. The RGB (red green blue) model is widely understood due to its use in (a) the eye, and (b) televisions, but there are at least two dozen other models in common use, largely depending on the application area. Many of these are industry standards.

This variety of models and standards is not because people can't agree on the best one to use. It's because there is no best: each device has very different needs. Thus, a colour printer (based on inks absorbing light) needs a different model from a VDU (based on phosphor emitting light).

However, there is more more fundamental problem that we can't standardise, namely human perception. The difficulty is that different humans respond to the same colour in different ways. This can be physiologically based (someone can be red-green colour blind; colours you have seen recently can affect the way you view another colour), but also culturally based. For example, in the West, white is associated with marriage, black with funerals. In other cultures the opposite is true. We associate red with danger, but this is only a convention (though probably derived from the colour of blood). Similarly, green is for "go."

Language is also important. Before the 16th century there was no word for "pink" or "orange," both were considered variants on red. In *Macbeth* the word "guilded" is used for the application of blood: gold, too, was thought to be a shade of red. Today we wouldn't think of relating gold and red, but this is merely an effect of language. This consideration becomes important when we ask when one colour is "close" to another: we would think orange and red are dissimilar while Macbeth might consider them quite close.

The seven colour rainbow spectrum was invented by Newton, who thought seven was a nice number, to match with the seven notes in an octave. Looking at a spectrum it is difficult to count seven distinct colours.

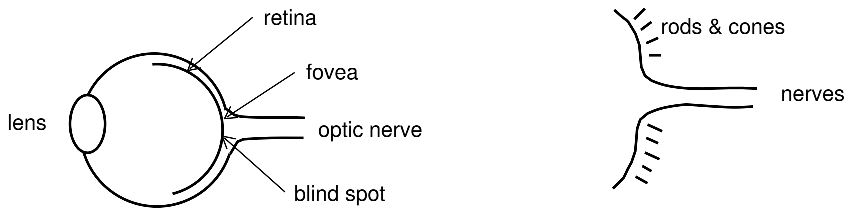
Colour is thus a mix of psychology (the mind), physiology (the eye), language and culture.

We shall start by addressing the physiology of the eye.

7.2 The Eye

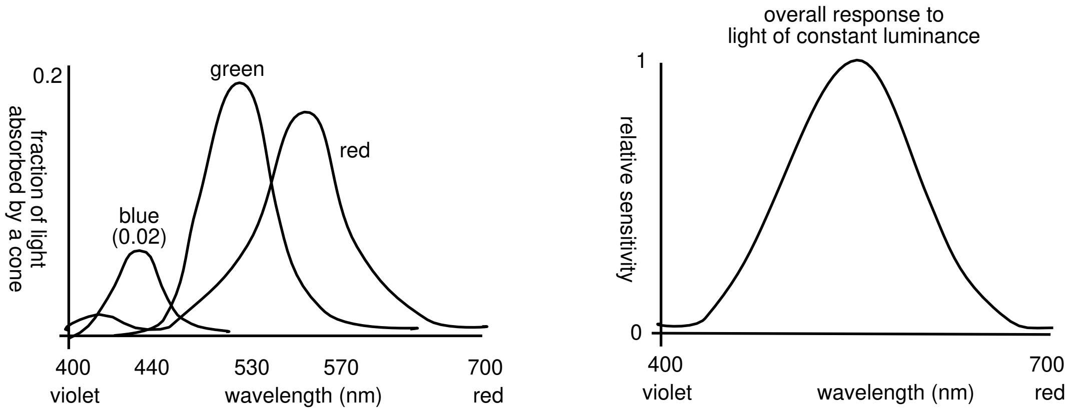
It is well established that the eye uses a combination of *rods* and *cones*. Near the centre of the eye there are many cones, which detect colour. Generally spread about are the rods, which detect brightness (shades of grey).

Cones only work well in bright light. In low light only the rods are contributing to what you see: this is why night time scenes are grey. Though the brain does try to fill in with colours it thinks you ought to be seeing.



The optic nerve enters the eye from behind creating a *blind spot* where there is no visual sensitivity. The nerves then continue into the eye over the *top* of the rods and cones. This is a strange arrangement, but the brain edits what we think we are seeing, eliding away both the effect of the nerves and the blind spot. Even at this level the brain has an effect on the optics of the eye.

Cones come in three flavours, with peak sensitivities normally in the red, green and blue parts of the spectrum.



The responses are quite broad, with considerable overlap. Green cones have the strongest response, red is close but slightly weaker, while blue is 20 times lower, i.e., the eye's response to blue is much lower than red and green (there are no very aggressive blue things in the natural world?). Blue peaks at about 440nm, green about 545nm, and red about 580nm. Interestingly, the blue and green peaks are within the yellow part of the spectrum. Furthermore, red cones have a secondary response in the blue part of the spectrum. The limits are about 400nm for violet and 700nm for red.

Thus, when we see, say, yellow light, what is really happening is that the red and green cones are being excited to certain relative levels. If we were to take red light and green light of the right intensities and mix them, the relative excitement of the cones would be the same, and we would perceive yellow. This, of course, is the principle of TVs.

Overall, the eye's response to light of constant luminance has a peak at about 550nm, roughly the colour of sunlight.

Even though we have a completely non-linear response in the eye, we will start with a simple *tri-stimulus* system. This means we shall represent a colour by a mixture of standard red, green and blue sources, mixed in some proportion. These are our *primary colours*.

We get *secondary colours* by mixing the primary ones in equal measure:

$$\text{yellow} = \text{red} + \text{green}; \text{ cyan} = \text{blue} + \text{green}; \text{ magenta} = \text{red} + \text{blue}.$$

We get white by adding all three primaries in equal proportion:

$$\text{white} = \text{red} + \text{green} + \text{blue}.$$

7.3 Tri-Stimulus Experiments

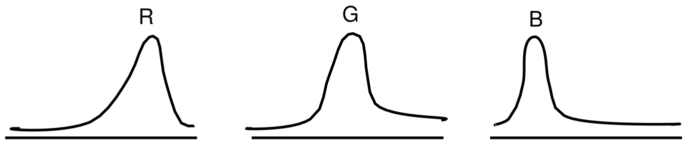
We can test the usefulness of this model. We can show a large of colours to a sample of (normal vision) people under some standardised conditions. We then get them to match the colours by mixing red, green and blue lights. If they can do this, then we may *define* the proportions to be the tri-stimulus values for the colours.

But there is a snag. It's not possible to get every perceived colour by mixing red, green and blue lights. The strange thing is that we *can* get every colour if we allow *negative* proportions of light.

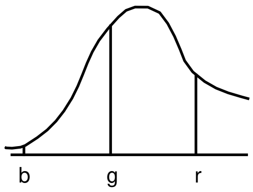
To explain this, we need to explain the experiments in more detail. The set-up presents the subject with two panels side-by-side. Once has the colour to be matched, the other is a neutral panel illuminated by the three sources. The subject can control the intensities of the three sources. Given a match, the three intensities define the tri-stimulus values for the colour.

If the subject cannot get a match, we add a colour *bias* to the original by adding, say, a little red to it. If the subject can now achieve a match, then the tri-stimulus value for the red component has to be reduced by the bias, producing a negative number.

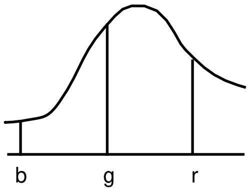
To see what is happening, recall the responses of the cones are roughly



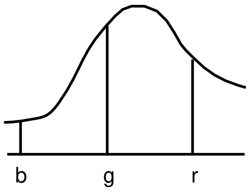
Suppose we wish to match a colour that should have this response in the eye



Mixing red and green we get too much appearance of blue due to the tails of the red and green responses.



So adding in a little blue to the test colour we get



which matches what we get from the tri-stimulus. Thus we would deem this colour to have a negative blue value.

Thus a colour TV or VDU *cannot* display every colour that the eye can see! However, the loss is not great, as they *can* display a huge number of colours, and any colour it can't display is only a smidgen away from one it can.

It seems that the eye can distinguish about 1/3 of a million colours if we judge them side by side, and the sensitivity of the eye is dependent on wavelength. For colours of equal brightness, the eye can distinguish colours when they are separated by

- about 10nm at the ends of the spectrum
- about 2nm near blue (480nm) and yellow (580nm)
- generally 4nm elsewhere.

Thus the eye is fairly unresponsive to blue, but can distinguish many shades.

7.4 The CIE Chromaticity Diagram

This was devised by the Commission Internationale de L'Eclairage (International Lighting Committee) in 1931, though it has been updated many times since. It is also known as the XYZ model.

This addresses the problem of having negative weights for colours: we would prefer only to have positive values. It solves it at the cost of having the three primary colours not physically realisable!

It starts with three *ideal* primary colours **X**, **Y** and **Z**, which are offset from red, green and blue in such a way we need only positive sums of **X**, **Y** and **Z** to produce our desired set of colours. Note that primary red, green and blue would require negative weights, and so are not representable in this system.

Suppose our colour **C** has weights X, Y, and Z

$$\mathbf{C} = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}.$$

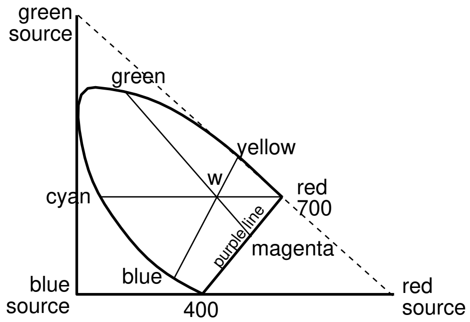
We call (X, Y, Z) the *chromaticity coordinates* of the colour, and these numbers are determined by experimentation with observers. We further define *chromaticity* values, or *normalised chromaticity coordinates* of the colour as

$$x = \frac{X}{X+Y+Z},\ y = \frac{Y}{X+Y+Z},\ z = \frac{Z}{X+Y+Z}.$$

This is dividing out the total energy of the colour, so the result is intensity independent. Now note that $x + y + z = 1$, so we can specify the colour by just x and y. We have lost intensity information, so we have dropped a dimension. To recover X, Y, and Z we need an extra piece of information, say Y, when

$$X = \frac{x}{y}Y,\ Y = Y,\ Z = \frac{1-x-y}{y}Y.$$

We can plot colours in the (x, y) space:

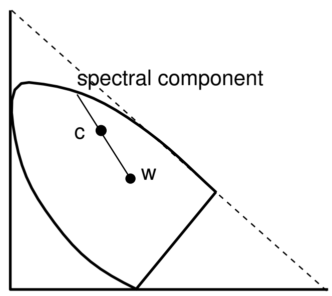


We have

- colours of the spectrum fall on the curved part of the boundary
- white is in the middle, somewhere close to $x = y = z = 1/3$
- the straight edge consists of *non-spectral purples* these have no single free colour equivalent
- as we move towards w, colours are more desaturated (contain more white): these are pastel colours
- all colours are of unit intensity due to the normalisation, so there are no intensity related colours, such as brown

An extra trick: we can choose **Y** to match the eye's overall response to luminance, i.e., the Y value is the luminance of the colour.

We can read off the spectral components and saturation of a colour from the diagram



The saturation is the distance from the colour to the white point, and the spectral component is the intercept with the curved edge.

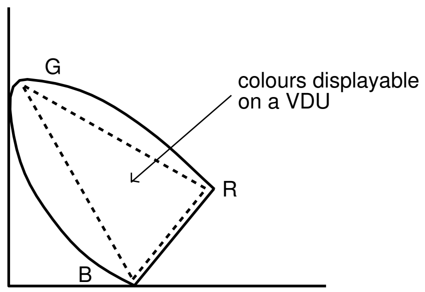
The position of white in this diagram is somewhat complex. People's opinion of what is true white has changed over the years, and has drifted somewhat towards a more bluish white. So the "reference white" has been defined with reference to the black body radiation of 6774 Kelvin. This is close to natural daylight.

7.5 RGB on the CIE Diagram

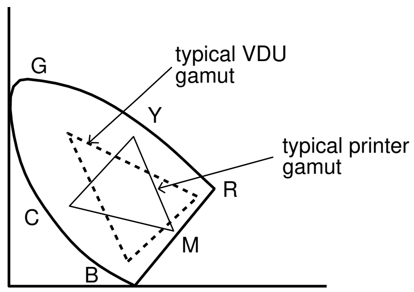
A VDU has three colours of phosphor red, green, and blue. Typical CIE values for them are

	x	y
R	0.628	0.330
G	0.285	0.590
B	0.1507	0.060

This defines a triangle in the CIE diagram, and all the colours that the VDU can display lie within this triangle.



We can never get all the CIE diagram within a triangle, so (again) we cannot display all colours on a VDU. In fact the typical triangle of a monitor is much smaller than we have pictured. We call the range of colours a device can display the *colour gamut* of the device. We can use the CIE diagram to discover the colour gamut of, say, a printer.



The problem of mapping a picture on the screen onto the printer is a difficult one. What should we do with those colours that the printer cannot print? There are several approaches.

- Perceptual mapping. Compress the colourspace so that colours outside the target gamut are mapped to inside. This will alter colours inside the gamut, but tries to retain relationships between colours.

- Relative Colorimetric. Map a colour outside the target gamut to the closest within, and leave colours within the gamut unchanged. Two colors which appear different in the source color space might be mapped to the same colour in the target gamut. Also known as *clipping*.
- Saturation. A mapping that keeps the original image's colours saturations. Used where bright colours are more important than accurate colours (e.g., business graphics).

A *colour management system* (CMS) tries to manage the mappings between devices while maintaining colour accuracy. CIE is often used as an intermediate colour model.

7.6 Colour Television

The next colour model we consider is that used by colour televisions. Or rather, that used in the transmission of colour television. This differs from the RGB that is used is the actual display. The reason is backwards compatibility with black and white television.

The PAL system (and other systems: SECAM in France, NTSC in North America) was designed to interoperate with the existing monochrome television services. Black and white deals with *luminance* (brightness), so PAL splits the colour information into a luminance and a *chroma* (colour) part. Old black and white TVs could use the luminance signal and ignore the chroma.

There was only a little unused bandwidth available for the chroma signal, and so it is squeezed into the high spatial frequencies of the monochrome signal. Thus we have a system with a fairly high definition luminance signal, with a smear of low resolution information over it. This is not so bad as it corresponds to the relative sensitivity of the eye to these two things: by exploiting known characteristics of the eye we can deliver a good engineering solution.

Gamma correction is applied at the studio before transmission as this is cheaper than requiring every TV set to have correction circuitry. (These days this would not be a problem, of course.)

The problem is when there are high spatial frequencies in the original scene, such as fine stripes on clothing. This produces high frequency components in the luminance signal, which are decoded as colour. Look at a test card to see the way fine lined areas shimmer in colour.

The luminance signal, named Y, is constructed as

$$Y = 0.30R + 0.59G + 0.11B,$$

for a colour with components R, G and B. These scale factors reflect the relative response of the eye: the blue component is much less important than red and green. This Y value is what is displayed on a black and white TV.

We now need to send the chroma values. We already sent Y, so we send values $R - Y$, and $B - Y$. These values are smaller than sending R and B directly, and so take less bandwidth. From this we can reconstruct immediately R, B, and then G from the equation for Y. We send differences for red and blue rather than green, as the difference for green would be larger, taking more bandwidth.

The YUV system, as used by PAL, has

$$Y = 0.30R + 0.59G + 0.11B,\ U = 0.49(B - Y),\ V = 0.88(R - Y),$$

being a collection of coefficients that fitted well with the old monochrome system.

The American NTSC is similar to PAL, but with different coefficients, called the YIQ system. The colour signal has less bandwidth than PAL, so NTSC colours tend to be worse, and we get more interference with the luminance signal.

7.7 Subtractive Primaries

We move away from light-emitting displays for a moment to consider light absorbing devices, in particular printers.

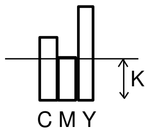
Primaries for printers are the secondaries for VDUs

$$\begin{aligned} \text{cyan} &= 1- \text{ red} \\ \text{magenta} &= 1- \text{ green} \\ \text{yellow} &= 1- \text{ blue.} \end{aligned}$$

This is the CMY colour model.

Thus yellow ink absorbs only blue; cyan absorbs only red; magenta absorbs only green. In practice, a magenta ink will absorb a bit of blue, sometimes even 50% of the blue light. Thus the simple "one minus RGB" formula can produce picture that are yellower than they should be.

Adding these three inks in equal proportion should produce black, but experience in play school teaches us we only get a muddy brown. It is difficult to add the three colours in precisely the correct amounts. Futhermore, there is a further complicating factor: the quality (and colour) of the paper you print on. There tends to be a transmissive element *through* the paper which introduces an additive element to the equation (higher quality papers try to reduce the transmitted light). To combat this printers tend to use a *four* colour system, with cyan, magenta, yellow and black, the CMYK colour model. The K value (the *key*) is the minimum of C, M and Y: it is the amount of black in the colour. The C, M and Y values are the remaining amount of colour after we have subtracted the black:



$$\begin{aligned} K &= \min\{C, M, Y\} \\ C &= C - K \\ M &= M - K \\ Y &= Y - K. \end{aligned}$$

one of C, M or Y is always zero. This produces a much better control over colours.

There are additional advantages to CMYK, too:

- coloured inks are typically more expensive than simple black ink
- if we have a black area composed of overlaid C, M, and Y mechanical tolerances will produce edges with coloured fringes (*registration errors*). The eye is most demanding at black and white edges.
- printing a single black rather than three inks will dry faster, and the print can run faster.

Some printers use the *hexacolour* model, which takes CMYK and adds light cyan and light magenta, which produces better colour control.

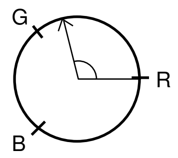
7.8 User-Oriented Colour

The CIE and RGB models are quite technical, and do not correspond to how normal people think of colours. Most people think of colours in terms of hue, rather than primary components. The *Hue*, *Saturation* and *Value* (HSV) model tries to accommodate this. Also known as HSB, with B for brightness.

We define HSV in terms of RGB. Suppose we have an RGB triple (all three values from 0.0 to 1.0). The amount of white in the colour is its *saturation*. If white $W = \min(R, G, B)$, then saturation

$$S = 1 - W = 1 - \min\{R, G, B\}.$$

Subtracting the white from the colour, we get the basic hue. In RGB, the hue is $(R-W, G-W, B-W)$. But one of these values is zero (just as in CMYK), so we have only two numbers to represent. Conventionally this is done as an angle around a circle.



Put R at $0°$, G at $120°$, and B at $240°$. A triple $(0.1, 0.4, 0.0)$ is $0.4/(0.4+0.1) = 4/5$ of the way around the arc between R and G. This is given either as an angle: $120° \times 4/5 = 96°$, or as a proportion of the whole circle: $96/120 = 0.27$.

Finally, the third component is *value*. This is

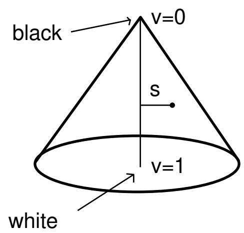
$$V = \max\{R, G, B\},$$

and is a measure of the brightness of the colour. We use maximum rather than the sum $R + G + B$ (which is a better value for the brightness), so that we control brightness uniformly across the spectrum: otherwise pure yellow (= red + green) would have a V value twice that of pure red.

So if we have a RGB colour $(0.3, 0.6, 0.2)$ this corresponds to $W = 0.2$, so saturation $S = 0.8$. Next, value $V = 0.6$, and finally hue: subtracting W from the RGB value we get $(0.1, 0.4, 0.0)$, which is $96°$. Thus RBG $(0.3, 0.6, 0.2)$ is HSV $(96°, 0.8, 0.6)$, a half-bright fairly solid slightly reddish green.

Tools for choosing colours using the HSV model can be easier to use than tools that use RGB, say. We directly pick a colour's hue, its saturation to determine how pure we want it, and its brightness. No need to try to remember that yellow is red plus green.

We can get an analogue of the RGB colour cube. This adds a third dimension to the hue circle:

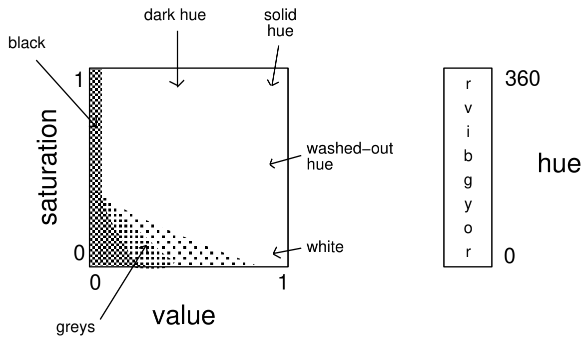


The vertical axis measures value, while radial distance measures saturation. Colours on the edge of the cone are fully saturated.

Problems with HSV include:

- Non-uniformity as we vary the HSV parameters. For example as we vary H the perceived rate of change of hues is not uniform (slow rate of change near the primaries, faster elsewhere).
- Non-independent parameters. As above, the value of V, say, can have an effect on the values S can take.

There are several variants on the basic HSV model. For example, where saturation is measured as $S = (\max\{R,G,B\} - \min\{R,G,B\})/\max\{R,G,B\} = 1 - W/V$, assuming $V \neq 0$. This adjusts saturation in proportion to the maximum amount of white available for a given colour (a colour with a high V can have a larger amount of white in it). In the basic model we must have $S \leq 1 - V$, while the adjusted model allows S the full range of values.



Colour Picker in Gimp

7.9 Other Models

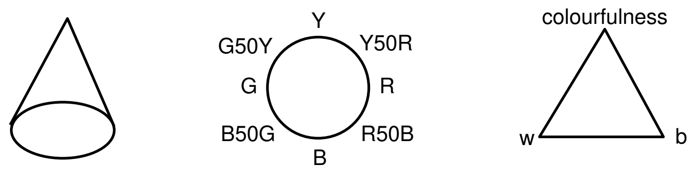
There are many other models, each for (generally) good reasons.

- CIELUV, CIELAB: these are variants on CIE. We find that *just noticeable differences* vary across the CIE diagram meaning that small movements in one part of the diagram may lead to a larger perceptual change of colour than equally sized steps in another part of the diagram. Near the top of the diagram you have to move further to just a just noticeable difference than near the bottom (even after the 1976 adjustment). The CIELUV and CIELAB models distort CIE so that equal changes on the diagram lead to equal perceptual changes. These models are better for colour interpolation between close colours, not so good for relating distant colours. CIELAB is used for subtractive, and CIELUV for additive displays.
- RGBA: the basic RGB plus an extra parameter called an *alpha* channel. This governs the transparency of a colour. For example, covering a blue with a green we get just the green in basic RGB. Using alpha channels we can declare the green to be 50% transparent. Then covering the blue again we get a mixture of 50% red with 50% green. This produces much more natural mixing of colours, e.g., green leaves of a tree let through a little of the blue sky behind.

  Alpha values are only supported poorly in the popular graphics file formats: JPEG doesn't, while GIF allows a limited form (a pixel can have transparency of 0% or 100% only, i.e., fully opaque or fully transparent). The PNG file format does alpha values properly.

- Bath Colour Model: this takes colour mixing further. it is a 7 parameter model. A colour is regarded as consisting of pigment (reflective, 3 parameters), plus a coloured medium (transparent, 3 parameters), plus a beta channel (proportion of pigment to medium, one parameter). This is good for mixing colours, and painting over other colours.

- NCS: the Natural Colour System is used by architects, interior decorators, and paint manufacturers. Another user-oriented system, fairly close to HSV. It is based on the relationship of a colour to six *elementary* colours: red, green, blue, yellow, black and white. Colours form a 3D cone, horizontal sections of which are *colour circles*:

  The colours are arranged in spectrum order about the circle, uniformly between red, yellow, green and blue.

  A vertical section gives a *colour triangle*, providing a scale between *whiteness* and *blackness* (value); and also *colourfulness* (saturation). Colours are described as Y70Rs70c20, meaning a dark brown orange color with 70% red; $s = 70$ means it has 70% blackness and $c = 20$ that it has 20% colourfulness.

- And so on.

Interestingly, tests on colour matching using various models seem to indicate that RGB is not so bad after all. Using RGB users made matches the fastest, but the results were not so accurate. Using HSV they were slower to make matches, but the results were more accurate.

7.10 Colour Interpolation

This is a very difficult subject in general. We use colour interpolation in many places. For example, Gouraud shading, antialiasing and blending images.

We can regard the interpolation as a straight line between two points in colour space: but which colour space? RGB, HSV, CIE, etc.? This is an important question as each gives different results for the interpolation.

For example, interpolate red to green. Using RGB, this is moving from point $(1, 0, 0)$ to $(0, 1, 0)$. Midway is $(0.5, 0.5, 0)$.

Using HSV we move from $(0°, 1, 1)$ to $(120°, 1, 1)$, and midway is $(60°, 1, 1)$.

Converting the RGB midway point to HSV we get $(60°, 1, 0.5)$, which is not so bright as the HSV midway point.

It is generally found that it is better to use RGB to interpolate between distant colours, e.g., in antialiasing, and to use HSV interpolate between close colours, e.g., in Gouraud shading. Other colour models may be even better, or more appropriate for other situations.

8 Projective Geometry

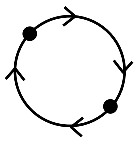
We now go back and look at the question of projection from 3D to 2D, and in particular homogeneous coordinates and projective geometry.

8.1 Geometry

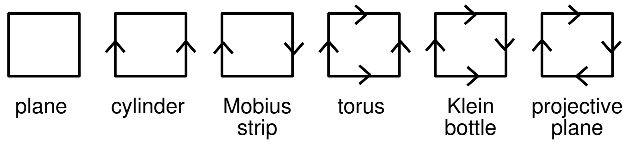
Euclidean geometry is familiar to everyone, being the study of properties of lines, points, circles, and so on. One property we take for granted is this: any two (straight) lines meet at a single point, *except* for parallel lines. This seemingly small exception was a great problem in the development of geometry, and eventually led to the birth of non-Euclidean geometry (as used by Einstein in the General Theory of Relativity).

What happens if we remove this exception? So *every* pair of lines meets at a point. We can make this happen by taking the ordinary Euclidean plane an adding extra *points at infinity*, these being the points of intersection of the parallel lines. If we imagine the Euclidean plane as a disc with us at the centre, the extended plane is the original disc plus a new circle of points around the edge. These are sometimes called *ideal points*, but points are abstract concepts anyway. These points form a line, the *line at infinity.* The whole thing is called the *projective plane*, and is the basis of projective geometry.

There is a problem with this simple image, though. Lines will intersect at ideal points on both sides of the disc: does this mean parallel lines meet at *two* points? The fix is to *identify* points that are diametrically opposite on the disc, i.e., the ideal point on the left *is the same point* as the ideal point on the right.

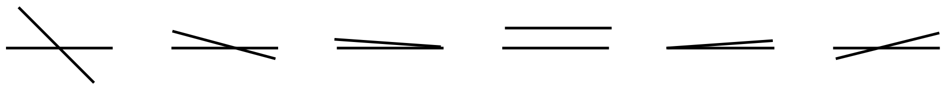


You can make models from paper:



Glue the edges of the piece of paper in such a way that the arrows point in the same direction. You will need to twist the paper for the Möbius strip, Klein bottle and the projective plane. Except that the Klein bottle and projective plane don't work well in 3D space! The projective plane is a bit like a sphere, except that the edge has a twist in it before we glue it together.

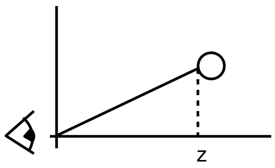
The twist is essential to ensure that intersections of lines behave as we would expect. Consider a pair of lines that intersect, and slowly rotate one line about:



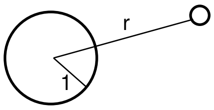
The point of intersection moves off the the right, until the lines are parallel, when they intersect at a point at infinity. Continuing the rotation, the intersection moves in from the *left*, thus indicating the point at infinity is both to the left and the right of the real plane. This is why we identify points at opposite sides of the disc.

8.2 Perspective Projection

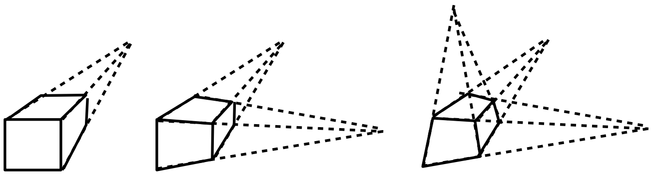
Projection is "divide by distance," so an object at twice the distance is half the size. Typically graphics books describe how to divide by the z coordinate ("divide by depth") to get this effect. But this is not quite correct, as distance is not the z depth.



The z value is not the distance to the object, but merely an approximation that is not so bad when the object is close to the z axis. Perhaps the Euclidean distance $\sqrt{x^2 + y^2 + z^2}$ is what we need? This gives proper perspective, but for a projection on to a unit sphere about the eye, not for a projection on to a flat screen.



If you talk to someone who does technical drawing about perspective, you will find that they mention one-point, two-point and three-point perspectives.



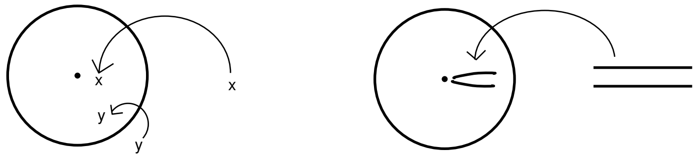
Artists, too, have multiple point perspectives, maybe 10 or 20 as the picture seems to require them. So how does this fit in with our notions of perspective?

8.3 The Riemann Sphere

The essence of perspective is that coordinates at Euclidean distance r are transformed to be at $1/r$. Thus we are mapping $r \mapsto 1/r$, which has a stationary point at $r = 1$.

Suppose we have a sphere of radius 1, centred on the origin in the "real" world. For every point outside the sphere at distance r there will be a corresponding point within the sphere at distance $1/r$. We can construct a model of the entire outside world within the sphere.

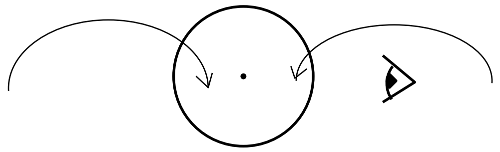
Points just outside will map to points just inside. Points at a great distance will map to points close to the centre, and points further and further away in the real world will bunch together progressively in the perspective world. Thus parallel lines in the real world will converge in the sphere. Points on the surface map to themselves.



Now there is no preferred direction in this sphere: every direction is equal. Looking into the sphere we see the world in perspective. Note that this means that we actually have a perspective model packed into the sphere, it's not just that is *looks* like it's in perspective.

The perspective model is certainly affected by the choice of origin. Is is the perspective that you would see from standing at that point. So to have a correctly distorted model of, say, a cube, we need to place our Riemann sphere at the point from where we want to view the cube.

Moving towards the centre of the sphere takes us out towards infinity in the real world. Looking towards the centre is like looking out to infinity. All outward directions in the real world meet at the centre of the projected world. But there's more to the sphere than that. A section of the sphere corresponds to the part of the real world *behind* you.



Things in the sphere beyond the centre correspond to real world objects beyond the sphere.

We have squashed most of the real world into a sphere of unit radius. What happened to the part of the real world that the sphere itself occupies? Well, that is mapped to *outside* the sphere. In fact, it is mapped to fill the entire rest of the universe. The point that was at the centre of the sphere is mapped off to infinity in all directions: it becomes an infinitely distant spherical shell. Thus what we have is not just a mapping on the sphere, it is a mapping of the entire universe to itself such that points on the sphere of radius 1 map to themselves. In the mapped universe perspective is "built in."

There is another important property of infinity. If we travel to plus infinity and keep going, we re-emerge from minus infinity. This total space, including the "sphere at infinity" is called the *Riemann sphere.*

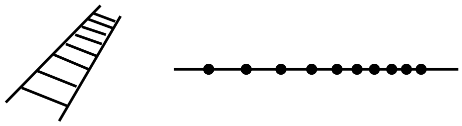
The *steepness* of the perspective is built-in, too. This is the measure of how fast parallel lines converge in the perspective world. With a steep perspective parallel lines meet faster, i.e., at a wider angle. The steepness of the perspective map can be controlled by using p^2/r instead of $1/r$, where p controls the steepness. This is clearly still a perspective map, but the stationary points are now at $r = p$, so we have a sphere of radius p. Changing this *radius of inversion* thus changes the steepness of the perspective.

It turns out that this idea of rebuilding the universe in perspective is a very useful one. We relate this to computer graphics by considering the simplest case, the one dimensional case.

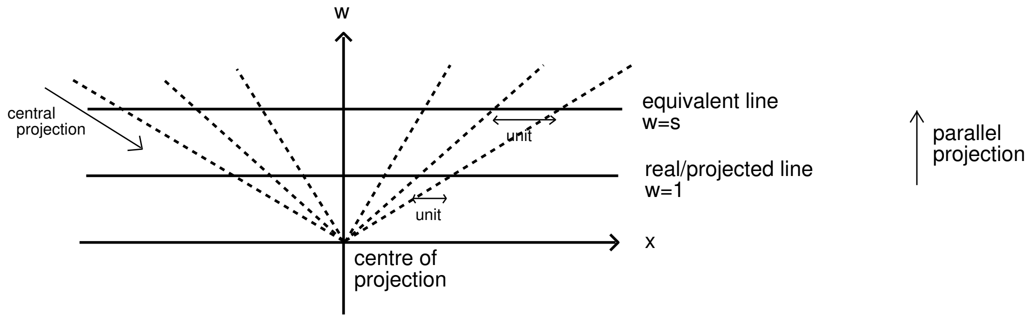
8.4 One Dimension

It is not immediately clear how to project in a 1D space: this is just a line. how can we tell if the points on the line are in perspective or not?

We can tell by looking at how the spacing of individual points of the line changes. We take a line in the real world and watch points that are a unit distance apart. Projecting this to a perspective line we expect that the points are no longer uniformly spaced, but cluster together as we look into the distance. Of course, this is true of all points on the line, but it helps just to think of the integer points.



When we talk of projection, we imply that we are projecting from one thing to another, in this case from one line to another.



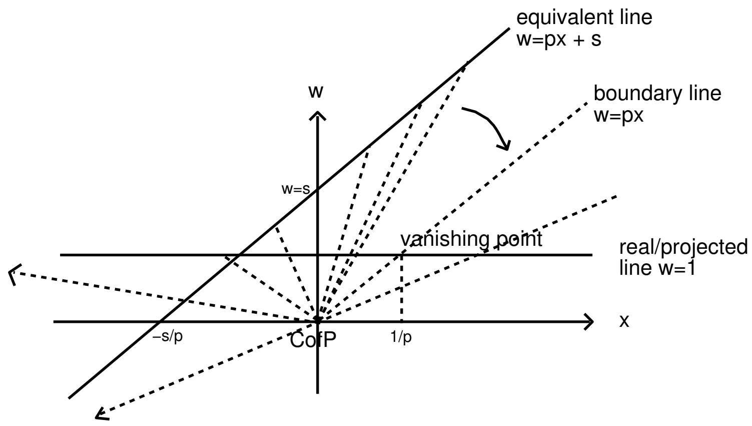
In this picture we have added a dimension—w—that is not one of the normal xyz dimensions in order to separate the original line and the projected line.

We start with the real line which lives in this diagram at $w = 1$. Parallel to it a little way above is the *equivalent line*, which is just a copy of the real line at $w = s$. Thus we *parallel project* the real line to the equivalent line.

Now we project the equivalent line back to the real world, but this time use a *central projection*. In particular, we can imagine how the integral points on the line map, as indicated by the dotted lines. This fan of lines is just another manifestation of the Riemann sphere. Indeed, the fan should go down as well as up to cover negative values of w.

The result is that we end up with a scaled version of our original line. If we move the equivalent line, i.e., change s, the scale changes. In fact the scale factor is $1/s$. We can choose $s < 1$ for a magnification, or $s < 0$ for an inversion of the line, where plus and minus infinity are swapped.

We have not yet produced any perspective effect. To do this, we skew the equivalent line so that is is not parallel to the real line.



We do the same for this diagram, namely take the real line, project on to the equivalent line, and centrally project back. Now the scale factor of the central projection varies linearly along the projection.

This is where homogeneous coordinates arise: the w allows us to perform perspective projections in higher dimensional spaces.

We can choose any non-zero value of w to place our real line. The only change is the amount of scaling going on, and this is independent of the projective scaling. Thus we usually choose $w = 1$ so the projected line has the same scale as the original. The value $w = 0$ corresponds to the real line passing through the centre of projection, and we cannot get a sensible projection.

The number p gives the steepness of the perspective projection.

Each radial line in wx space corresponds to a single point in our original x space. Put another way, a line in projective space is equivalent to a point in Euclidean space.

In the above diagram, there is a line called the *boundary line*. Imagine how a radial line behaves as it moves to the right along the equivalent line. The real points it corresponds to cluster closer and closer together until the radial line meets the slope of the equivalent line. The point at infinity of the equivalent line projects back to the *vanishing point* on the real line at $x = 1/p$.

If, now, we continue to move the radial line to the right, the real points now move along the real line from minus infinity towards zero. Again we find plus and minus infinity are the same place. Finite parts of the equivalent line correspond to all those points of the real line, excepting the vanishing point.

we will generalise this idea to 2D, and then 3D. Summarising, we

- start with a physical line
- embed in the higher dimensional xw space at $w = 1$
- create the equivalent line at an angle appropriate for the required steepness of perspective, and parallel project the real line on to it
- we centrally project the equivalent line back on to the real line
- we drop back down into x space by dropping the w coordinate.

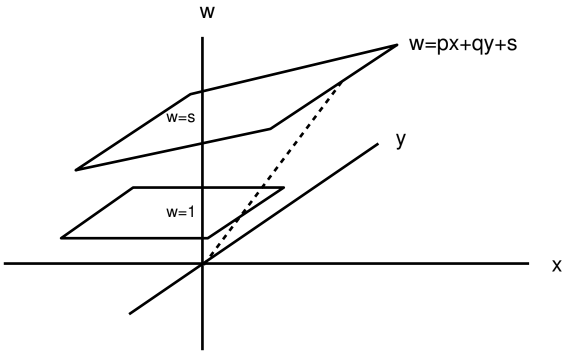
This process can be written in matrix form.

$$(x\ 1)\begin{pmatrix} 1 & p \\ 0 & s \end{pmatrix} = (x\ px+s) \stackrel{\text{normalise}}{\longrightarrow} (x/(px+s)\ 1).$$

Thus the line at $w = 1$ maps to the lie $w = px + s$. We can choose $s = 1$ to avoid an overall scaling. If we now centrally project back, this is just normalising w, giving the desired perspective.

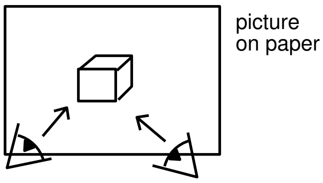
8.5 Two Dimensions

We follow precisely the same route as for the 1D case. Now we start with an xy space and add a dimension to get xyw space. The real plane is $w = 1$, which is parallel projected on to the equivalent plane $w = px+qy$. Notice that we now have two parameters to control steepness of perspective, one for each of the x and y directions. So we will also get two vanishing points, one for x and one for y. This is the *two point* perspective. If we happen to choose, say, $q = 1$, we only get perspective in the x direction, and only one vanishing point: this is a *one point* perspective.



The "divide by distance" process works even though there is no z distance to divide by: it is the central projection that does the work.

There is nothing here to fix the point of view of the viewer, though the centre of projection is the only place where the projection will appear correct. This is quite typical: we often draw perspective pictures of 3D objects on 2D paper, and these pictures will only look technically correct when the viewer is directly in front of the paper.



The matrix form is a simple extension of the 1D case.

$$(x\ y\ 1)\begin{pmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & s \end{pmatrix} = (x\ y\ px+qy+s) \stackrel{\text{normalise}}{\longrightarrow} (x/(px+qy+s)\ y/(px+qy+s)\ 1).$$

As before, choose $s = 1$ to avoid overall scaling. The division to normalise is the central projection.

8.6 Three Dimensions

We can't draw a picture for this case, but we can proceed with the algebra. The projection plane is $w = px + qy + rz + s$, and we can have up to three vanishing points. As before, we have to choose a viewpoint (the centre of projection) to ensure that the projected object we see is "correct."

Think of what is happening here when we draw a picture of the result: a 3D model is projected into a 4D perspective space. This is projected back to a 3D "perspectivised" version. This is then parallel projected down to a 2D representation to view on the screen. This can produce quite a bit of distortion of the original model!

The matrix form is direct:

$$(x\ y\ z\ 1)\begin{pmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & s \end{pmatrix} = (x\ y\ z\ px + qy + rz + s)$$

$$\xrightarrow{\text{normalise}}$$

$$(x/(px + qy + rz + s)\ y/(px + qy + rz + s)\ z/(px + qy + rz + s)\ 1).$$

Now p, q and r determine the steepness of the perspective in their respective directions, while s controls the overall scaling.