Graphics Notes
http://www.matthew.ath.cx/notes/

Matthew Johnson
Trinity Hall
mjj29-notes@srcf.ucam.org

May 27, 2003

1 Why Computer Graphics and Image Processing?

All visual computer output depends on Computer Graphics - whether its printed out, on a monitor, or otherwise.

2 Lecture1 : Background

2.1 What are CG & IP used for?

2D computer graphics are used for designing: logos, posters, maps - even cereal packets and other packaging. All results of Computer Graphics. 2d also for rendering graphical interfaces, typesetting etc.

Image processing is used for photo retouching, collaging satellite imagery, etc. New forms of artwork are also possible.

3D graphics have many applications, from scientific and medical uses, to CAD, and a lot of use in the entertainment industry - special effects, games, movies etc.

c.f.: alien song! 42 seconds of animation, 442 hours of work, and a few hours of rendering.

Most of the graphics problems are solved - it is now an issue of making sure designers have to tools to work quickly and efficiently.

2.2 What is a digital image?

We need to know about human vision, to determine the limits of digital images.

An image is a 2 dimensional function, the value at any point is an intensity or colour. This is a continuous function in X and Y. *digital* images are in a sense a contradiction in terms. If you can see it, its not digital - digital is just a collection of numbers. It is a quantised and sampled version of the image - normally a 2-d array of discrete colour values. A variety of devices can be used to capture the images. Commonly a CCD is used, a line of them in flatbed scanners, or an area CCD in cameras. Expensive drum scanners have a single (expensive) spot detector, that the image is moved around to be measured.

During sampling, the detector looks at a small area of the picture, and averages the colours together, to normally produce red green and blue phases - each normally in the range 0-255. This gives you a rectangular array of numbers - how do you display it. The image must be reconstructed on a CRT, LCD etc.

This reconstruction produces artifacts from the digitisation, and that depends on the output device. CRTs have a Gaussian distribution on each dot, producing a blurry image, where as the LCD is more pixel lated.

2.3 Sampling

A digital image is a rectangular array of intensity values. Each value is called a pixel (abbr: "picture element"). Sampling resolution is measured in ppi or dpi ({pixels,dots}-per-inch). Computer monitors are $\approx$ 100dpi, and laser printers are between 300 and 1200dpi. After a point, the difference between the resolutions is negligible, compared to the sampling resolution of your eye, at the distance you are expected to look at it from.

2.4 Quantisation

Each intensity value is a number, and you have to quantise this for digital storage. This limits how many different intensity values we can store, and importantly limit the brightest intensity we can represent. The quantisation level as well as the resolution are factors in the resulting quality and size of an image.

2.5 ref: the human eye

2.5.1 The Retina

The retina has $\approx$ 150 million light receptors, concentrated in the Fovea. There are only $\approx$ 1 million nerve fibres in the optic nerve. Hence, the retina performs preprocessing on the signals to reduce 150M signals to about 1M. This includes averaging of inputs, edge detection, and processing the colour signal. The distribution of rods and cones also depends on distance from the Fovea, with colour detection concentrated in the Fovea.

2.5.2 Processing in the eye

The eye has significant processing ability.

- discrimination - The eye can discriminate between colours and intensities.
- adaptation - changes in illumination and colour are adapted to by adjusting the iris, but within that we can distinguish about a 1:100 ratio of contrast. With adaptations, we can see over a range of $\approx 10^{1}0$, but not simultaneously. The 1:100 range is a sliding window that can cover the whole $10^{1}0$ range. The exact limits depend on the person. The detection threshold of rods an cones are also different - meaning that you can see things out of the corner of your eye, that are invisible straight on. People have also looked at what the threshold of intensity to be able to distinguish different colours is, and toward the white & black ends the distinguishing threshold is higher.
- persistence - the eye integrates light over a period of about a 30th of a second.
- edge detection - better in the fovea than away from it.

2.5.3 Resolution

The fovea has about 150,000 cones / square millimetre. Elsewhere the resolution drops off, as does the colour detection, but is still enough for most purposes.

The non-linear behaviour of the processing makes it extremely hard to model or predict.

3 Lecture 2 : Background ctd.

ref: random snake video.

3.1 Requirements for vision

What do we need to be able to see things? We've talked about the eyes - the detection methods, but we also need illumination, and something to reflect the light.

3.1.1 Illumination

Light is electro-magnetic radiation, but only a tiny part of the spectrum. Visible light is in the range 700nm (red) to 400nm (violet) in wavelength. We can detect other parts of the spectrum, but in the case of infrared, we detect it as heat, rather than light. Light sources are rarely a single wavelengths, but generally emit a wide spectrum of light, and is a composite of those wavelengths. (see spectrum graph). Sunlight has a much wider (and more blue / violet) spectrum than normal lighting, which is mostly towards the red end. Our eyes can adapt to this though, so we don't notice. Objects, on the other had, also reflect a composite of wavelengths, and the interaction between this, and the illumination they are under, and completely change the apparent colour.

3.2 Classifying Colours

Munsell classified colours using three axes - hue, lightness and saturation. The hue tells you the dominant colour, the lightness contrasts between dark & light, whereas the saturation corresponds to vividness of colours. (see colour supplement 3D colour wheel). The Munsell system should have the (perceptual) difference between adjacent colours a constant over all of the colours. There seems to be some correlation between saturation & lightness.

3.2.1 How the eye responds to colours

There are 3 types of cone. Each responds to a different spectrum, roughly long, medium & short wavelengths. However, they aren't spaced on the red / green / blue peaks evenly, there are two clustered near the green/yellow point (and are on the X chromosome, hence when men are more likely to be colour blind than women). The 3rd (blue) set of cones is 120 times less frequent than the others, and hence you can't determine fine detail in blue.

The retina does some preprocessing too:

- long + medium + short, to get luminance
- long - medium, to get the red/green distinguishing (and loosing one of these two produces red/green colour blindness.)
- long + medium - short, to get yellow/blue distinction.

Since we only have 3 input signals, then using 3 numbers to represent colours is sensible. The values of these can be calculated as ... (see sheet)

It is also possible to induce the the same response in our cones with several different input signals.

3.3 Mixing Colours

Generating the other colours we can see is done by mixing red, green and blue. However, not every wavelength we can see can be represented as a mix of red, green and blue. Since defining coloured light as a mixture of three primaries is desirable, we define a different set of 3 primaries. In 1931 CIE defined 3 primaries X, Y & Z. Y was defined as a function such that the human eye's response to it is constant at each wavelength. The other 2 appear to be defined randomly, and don't correspond to colours, but make it possible to construct all possible visible colours. The function is not a bijection though - many combinations of X, Y & Z don't map to visible colours (see XYZ graph). Once luminance is extracted (divide by $X + Y + Z$), it can be reduced to a 2-D diagram of colour. Pure colours lie on the outer bound of this graph, and mixed colours are in the centre.

3.3.1 RGB

The horseshoe of XYZ-space is a superset of that representable with RGB. The colours Red, Green and Blue are points on the edges of XYZ space, and they form the points of a triangle, all within the horseshoe, and a subset of it. Anything within that triangle is representable using RGB, and outside that you can't represent using RGB.

This is true using any three colours (pure, or mixed) as the base to represent colours, by mixing. Colours within a triangle defined by the 3 primaries used are representable, and ones outside are not. (since light is only additive, not subtractive).

3.4 CompSci - Representing Colour

You can use XYZ or Yxy to represent colour, but the pragmatic approach is to store colour in the same format that the hardware uses. For display, we use Red, Green & Blue. For printing, however, are *subtractive*, not additive, hence their primary colours are Cyan, Magenta & Yellow. These colours each *absorb* one of the light primaries, and reflect the other two colours. CMYK is a convenient extra to add, since trying to get black by missing all the colours tends to not work well, and waste ink.

Many people use the Munsell colour space - HSV, or HLS. This is because its quite difficult to estimate the Red, Green and Blue required to form a given colour. Both have Hue (colour), Saturation (intensity of colour) and either Value or Lightness to give the intensity. In the case of Lightness, you have the most vividness at half the maximum lightness, whereas value has the maximum saturation at the maximum value.

Uniform colour space is based on having equal steps in any direction producing an equal difference in perception. There are two versions of this, L*a*b* and L*u*v*. These use different coordinate systems, but the standards body ratified both of them, because they couldn't work out which was best. L*a*b* is quite close to the Munsell model, whereas L*u*v* is closer to the XYZ model.

4 Lecture 3: Background ctd

4.1 Implications of vision on graphics

4.1.1 resolution

In theory you can see about 600dpi, 30cm from your eye. Actually, according to opticians the ability to see a white gap between 2 black lines, the limit is about 1 minute. This corresponds to =300dpi at 30cm. This value is not a constant, it is affected by several factors. The distinguishable resolution drops as the contrast decreases (if things are very similar in contrast, then you can't resolve them as closely). Colour resolution is much worse than intensity resolution, due to the preprocessing they eye does on the colour channels. This all means that at a meter away from a 100dpi monitor you can't distinguish pixels, but you can closer.

The eye also does some other tricks, which makes it very good at distinguishing vernier accuracy. You can resolve small discontinuities down to 4 seconds of arc, which is about 4500dpi. You can fool the eye with colour grading though.

4.1.2 quantisation

Humans can distinguish about a 2% change in intensity, and on CRT screens the difference between the minimum and maximum intensity is about a factor of 25. Looking at it logarithmically, then $\frac{\log 25}{\log 1.02} \approx 163$ levels of quantisation is OK. For linear, then you need $n = \frac{25-1}{0.02} \approx 1200 \approx 2^{10}$, so 10 bits is OK.

For TV, then, we need 10 bits of information - although 8 bits is usually sufficient, and for movie film (with 10 times the contrast of TV) use 14 bits (usually 16) of intensity information.

4.2 Storage of images

We've said that we need 8 bits of intensity information per pixel. So, we encode 8 bits as 1 bye, and use a $W * H$ array of bytes to store the image.

For colour images, we use 1 byte per Red, Green, Blue pixel, and hence get 24 bit colour. You could store each pixel as 3 bytes in a $W * H * 3$ array, but its more common to use separate planes for each colour. This can be extended to alpha planes, z-buffers etc.

Most graphics systems have an area of dedicated RAM used as a buffer for the screen. Just writing to it directly can produce odd visual effects due to concurrency with the scan rate. Hence, we employ double buffering to ensure that the buffer being read by the screen is not being written by the system at the same time. You then switch which buffer does what between the two, at the frame rate you are expecting.

4.3 Display Devices

There are many types of display technologies, but only a handful cover 99% of those in use.

4.3.1 Liquid Crystal Displays

These work by twisting the polarisation of light. The front has a polarising filter, and the individual pixels either match, or mask at right angles, to the filter. (see diagram from notes). When you pass current across the pixel you can switch the polarisation of the light from the back-light. When the pixels are 'off' they don't line up the polarisation of the light, and no light gets through. When they are 'off' light comes through, and hits a coloured filter. Variation in intensity is achieved by only making the liquid crystal partly re-polarise the light.

The LCD doesn't take much power, but the back-light does.

4.3.2 Cathode Ray Tubes

Invented by J.J. Thompson. These work with an electron gun firing through a vacuum onto a phosphor screen. The phosphor screen then lights up. There are two sets of magnetic coils. One that focuses the electron gun, and one that deflects the electron gun to the appropriate part of the screen. This then scan linearly across all the pixels once per refresh cycle. The intensity is varied by changing the intensity of the electron gun. A scan frequency of higher than 20Hz can look like continuous motion, however, humans can be sensitive to flicker at higher frequencies.

The desired effect with the scan line is an instantaneous flick back to the next scan line, however, you have a small about of time to reset to the left hand end. (about 1/4 of the time to scan horizontally). Ditto with the vertical scan, which is a slow even drift down, continuously while the horizontal scan happens, then quickly back to the top. This time required is due to resetting the electronics.

The refresh rate is the speed we can update the entire screen. The default for PAL TVs is 50Hz. At this refresh rate many people can see it flickering. NTSC use 60Hz, however at a lower resolution to fit it in the same bandwidth. Modern TVs and CRTs display each signal twice and get 80-90Hz displays. 99% of people can't see flicker at this rate.

4.3.3 Colour in CRTs

Electrons have no colour. To get this small enough that you can't notice, the spots have to be smaller than the electron beam. This means that you can't focus it on individual colour spots. The solution is to have 3 electron guns and a shadow mask, which is a metal plate with small holes in it. You align the mask and the guns, so that the electrons from the 'blue' gun can only hit blue phosphor etc. This can get the dot-pitch down to 0.25mm. Trinitron (or flat-tension) shadow masks can improve this to 0.15mm. This has vertical wires & strips, rather than holes in a metal plate.

4.3.4 Printers

There are several main types of printer.

- Ink Jet - sprays ink onto paper. up to 1200x2400dpi
- Dot Matrix - pushes pins against ribbons
- Laser - uses a laser to lay down a charge pattern, that toner sticks to. up to about 1200dpi
- Photo Typesetter - can get up to 3000dpi.

All of the above either have a pixel set, or don't. How do we get gray-scale? This is done by half-toning, which is the technique of printing different sizes of small dots to represent lighter shades of grey.

4.3.5 Laser Printers

Built from a drum that rotates over the paper. This drum is selenium coated. This is good at holding charge, but releases it under light. You charge the whole surface, then scan a laser across the drum to remove parts of the charge. You then apply negatively-charged toner onto the positively charged drum, and it sticks where the charge is left on the drum. This is then pressed against the paper, and then heated to infuse the ink. This method can only be black or white, and grey-scale is done via half-toning.

Half-toning groups the pixels on the printer into cells, that correspond to 1 pixel in the computer. You then set a selection of the sub-pixels black to correspond to the greys.

4.3.6 Dye Sublimation Printers

These can do real grey-scale. You have a dye sheet that you place over the paper, and pixel-sized heaters sublime the dye off the printer onto the paper. The amount of dye that gets put on the paper is proportional to the heat applied. You can also apply several different coloured dye sheets, and the heat will mix them together.

4.3.7 Printing in Colour

Generally we use a mixture of CMYK inks. Inks absorb colour, so CMY is the inverse of RGB. K (Key, or Black) is necessary because inks are not perfect absorber, and C + M + Y doesn't quite produce black. Also, we print a lot of black text, so it is more efficient, and makes it unnecessary to align them quite as close. Again, half-toning is used to give different intensity values. Unlike with monochrome, you have to carefully align the the spots in the half-toning so that the colours don't overlap. Each colour half-toning is generated as a separate 'screen', which is then combined in the above 'moire' patterns.

5 Lecture 4: 2D Computer Graphics.

5.1 Toy Story

Introduced the idea of "CPU Century" - it took 800,000 of CPU time on 117 dual & quad processor SPARC Station 20s - for 79 minutes of animation.

5.2 Lines & Curves

How do we draw straight lines? Normally we would just draw a line along a ruler, however, we have only got pixels we can turn on or off.

5.2.1 Straight lines

A straight line can be defined by $y = mx + c$. Mathematically, a line is "length without breadth", however, we can't do that either. There are 2 sensible methods for doing this with pixels. Firstly, we colour every pixel the line passes through, or we colour the closest pixel in each row or column. The second one is what we normally use, although we have to know whether to colour by-column or by-row.

Locations of pixels, which have integer sizes and locations, we define by either the centre, or the corner of the pixel. We will define the centre at real co-ordinates (x, y) and extend from $(x - 1/2, y - 1/2)$ to $(x + 1/2, y + 1/2)$.

5.2.2 Bresenham's line drawing algorithm 1

```
d = (y1 - y0) / (x1 - x0)
x = x0
yi = y0
y = y0
DRAW(x,y)

WHILE x < x1 DO
   x = x + 1
   yi = yi + d
   y = ROUND(yi)
   DRAW(x,y)
END WHILE
```

Here, `d` is the gradient of the line. Note that this algorithm assumes the end points are integer. Also note that you have to consider the end conditions carefully, as the boundaries can be difficult to get right.

5.2.3 Bresenham's line drawing algorithm 2

```
d = (y1 - y0) / (x1 - x0)
x = x0
yf = 0
y = y0
DRAW(x,y)

WHILE X < X1 DO#
   x = x + 1
   yf = yf + d
   IF (yf > 0.5) THEN
      Y = y + 1
      yf = yf -1
   FI
   DRAW(x,y)
END WHILE
```

5.2.4 Bresenham's line drawing algorithm 3

```
dy = (y1 - y0)
dx = (x1 - x0)
x = x0
yf = 0
y = y0
DRAW(x,y)

WHILE X < X1 DO#
   x = x + 1
   yf = yf + 2dy
   IF (yf > dx) THEN
      Y = y + 1
      yf = yf - 2dx
   FI
   DRAW(x,y)
END WHILE
```

This last algorithm has removed all the floating point calculations and made them into integer calculations, which when this algorithm was written made it a lot faster on the processors that were around. In fact the loop can be reduced to 5 machine instructions.

Depending on which octant (eighth of a circle) the line is in, we need 4 different algorithms (depending whether we are colouring per-row or per-column, etc.). Creating the other 4 octants is merely a case of swapping the end points and using one of the first four.

5.3 More line drawing algorithms

Bresenham's algorithm can't easily be generalised to curves and circles, however, other algorithms can.

5.3.1 Midpoint line drawing algorithm

This works by dividing the area into three regions, $> k$, $< k$ and $= k$ where $= k$ is on the line. We evaluate this *decision variable* at each point to find out which region we are in, to decide whether we should move right or up.

In a 1st octant line, we have the choice of the next pixel being to the east or the north-east. We evaluate the line at the midpoint of the next pixel, and depending if the line is to the left or the right of this point define whether we are going east or north-east.

The decision variable $d = a(x+1) + b(y+1/2) + c$ for successive ds collapses to $d' = d + a$, $d'' = d + a + b$.... etc.

As before we have to similarly extend the algorithm into lines in the other octants, but this algorithm can be extended to draw arbitrary circles & ellipses - whereas Bresenham can only be used for circles with integer radii.

6 Lecture 5: Curve Drawing

6.1 Curves

- Circles & Ellipses
- Bézier cubics - developed by Pierre Bézier, and widely used in Graphic Design.
- Overhauser cubics - another version
- Non-Uniform Rational B-Splines (Part II course)

6.1.1 Midpoint Circle Algorithm

The Equation of a circle is $x^2 + y^2 = r^2$, centred at the origin. We take the decision variable to be $d = x^2 + y^2 - r^2$ with d being inside, outside the circle or on the circle.

The circle has to be divided into 8 octants as before. Looking at the second octant, we use the decision variable to decide whether to go E or SE.

6.1.2 Extensions

The above algorithm only draws circles at the origin, but can easily be altered to draw them at arbitrary points. Ovals can also be drawn, however, you have to be careful about octants - you divide up on the 45 degree lines. You can't, however, with this algorithm allow you to draw ovals which aren't aligned with the axis.

6.2 Are circles & Ellipses enough?

For almost any real applications, we need free form curves. These are based on cubic equations. Lower orders (linear or quadratics) are not flexible enough - they are only straight lines of hyperbola. The number of degrees of freedom is important, since linear we only get 2, and quadratics 3. Cubics can match 4 degrees of freedom ("wiggles"). Above cubics, we have more degrees of freedom, but this generally "wiggles" more than we want, and takes longer to compute.

6.2.1 Hermite Cubics

Hermite cubics are defined by their end points, and the tangent vectors at those points. The standard cubic formula is $y = at^3 + bt^2 + ct + d$. In terms of the Hermite cubic definition, we get $a = b = c = d = ...$

You can join up Hermite Cubics by matching position & tangent at the end points of the cubics.

6.2.2 Bézier Cubics

Bézier decided to define the 4 variables as the end points, and 2 other "control" points, that are used to calculate the tangents. This produces the formula: $P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$, which is a sum of Bernstein polynomials. This is equivalent to the Hermite Cubics, but easier for people to conceptualise.

Another property is that a Bézier curve lies within the convex hull of its control points.

Again, we can join endpoints. We get a smooth join (C_1) if the control points are equal distances and directions from the join point, C_2 continuity only needs the end points to be in the same place. C_n means continuous in that derivative.

6.2.3 Continuity

The two types of continuity we consider are C_0 and C_1 continuity. These are the types of join we have between 2 types of Bézier cubics. C_n continuity is the case where the cubics are continuous in all derivatives up to n. C_0 therefore, only has the positions (zeroth derivative)

6.3 Drawing Bézier Cubics

6.3.1 Naive Method

We cannot have a nice algorithm like we had for circles, so we approximate them by taking the parameter t, and for each step, drawing the closest straight line to match that part of the Bézier curve. Unfortunately, you cannot fix the stepping of the parameter, since the appropriate value is different for all Béziers. Picking large numbers arbitrarily will slow down the program, and low ones often gives bad approximations. Also, distance in real space is not linearly related to parameter space.

6.3.2 A more sensible method

We can use a divide and conquer technique to draw Béziers. First, you check if a straight line between the end points is good enough, and if so you draw it. If not, you split the line in two, and recurse the algorithm in each half. We define "good enough" to be when the distance between the curve, and the approximating straight line, is under half a pixel.

This gives ha an easy algorithm:

```
procedure DrawCurve(Bezier curve)

   var
      Bezier Left, Right;
begin
   if Flat(curve) then
      DrawLine (curve);
   else
      SubdivideCurve(curve,left,right);
      DrawCurve (left);
      DrawCurve (right);
   end
end
```

We know how to draw lines, so the main thing above we don't know how to do is define whether a curve is flat. From earlier we know that the curve must lie within th convex hull of its control points. Hence, we can just use the control points (who's position we know) and check whether *they* are within half a pixel of the straight line.

The first thing we do is to find the approximation between P_0 and P_3. We then need to find the perpendicular distance of the control points from the approximation. This can be done by the dot-product, since the two lines are perpendicular. If that distance is less than some ϵ (1/2 a pixel). The special case that must be checked for is when the curve is outside the end points, but within the control points, since although it might approximate to a straight line, the line should be *longer* than the distance between P_0 and P_3

6.3.3 Subdividing Bézier Cubics

In the above algorithm, we needed to split the cubic. See the diagram in the notes, but you basically take the midpoints of all the lines, and the end points, and the left and right halves go to each curve.

6.4 Overhauser's Cubic

In the case where we don't have any tangent vectors, or control points, merely a series of points we want to go through. In this case for each of the smaller curves, we use the two endpoints of that section, and the points either side of that section. As you move along drawing sections of the curve, you will shift along one point at a time. You then take the vectors from the endpoints to the opposite extra point as the tangent for drawing the curve.

This is the basis of Overhauser's Cubic. The best way to do this, is to calculate the Bézier of Hermite values and draw them. The difference between the Overhauser cubic and Bézier, is that when you move a point in a Bézier cubic, it only alters the lines directly joining that point. Overhauser's method also changes the properties of the curves outside of those. This can be useful in animation, but usually not for CAD.

7 Lecture 6: Line Chains & Clipping

7.1 Simplifying Line Chains

If you have a really high resolution series of lines, which you are drawing at a low resolution, you need to simplify it, otherwise you could try and draw several hundred lines, all of which occur in the same pixel. The solution for this is *Douglas & Pücker's line chain simplification algorithm*. This is used a lot (and was originally developed for) cartography. You can also apply it top high resolution Bézier cubics.

The method, is to scan along the chain of lines, to find the point that is furthest away from the line between the end points. If that distance is further than some tolerance from the baseline then you split into two halves, using that midpoint.

The problems are firstly similar to the above case with the Bézier cubic method, and also when you have a closed curve. This can be solved by just splitting the line straight away.

7.2 Clipping

It is very important that we know where the bounds we are allowed to draw to are. Both to avoid writing off the screen, and also when we are scrolling within a window.

7.2.1 Points

You can check whether a point is outside a rectangle by checking four inequalities. Iff all four are true, its in the area.

7.2.2 Lines

If both of the end points are inside the bounds, this is easy, since the whole line is inside. One in one out is obvious - part of the line must be drawn, however, you can't just discard a line which is all outside the bounds, since it may pass completely through the area, and out the other side, and bits of the line should be drawn.

To clip against an edge, you parameterise the line, and then calculate the value of the parameter when the (infinite) parameterised line crosses the edge. The value of this parameter tells you whether you cross a boundary. You can calculate this four times, and use the resultant values to test the type of line to draw.

BUT: each one of these clips takes 2 subtractions and a division, times 4, per line. This is expensive. You also have to watch for divide-by-zeros on vertical lines.

7.2.3 Cohen-Sutherland

This algorithm assigns a 4-bit code to say whether a point is inside of outside a line. Each bit position corresponds to one side of the region you are drawing in, and if the point is outside that bound, then that bit is set. If both ends have a code of zero ($Q_1 = Q_2 = 0$) then it is completely within the bounds and can safely be drawn. If there is a common bit set in the codes ($Q_1 \wedge Q_2 \neq 0$) then it is completely off to one side, and can be ignored, The other case you intersect with one of the lines and start again, but which bit is set tells you which edge to clip against. The last case does require the previous divisions - but this is also the rarer case. If several bits are set, then you have multiple edge crosses, and you have to calculate both clips, and recalculate to find which you should clip against.

8 Lecture 7: Clipping Béziers & Polygons

8.1 Clipping Béziers

We've looked at clipping straight lines, now we need an algorithm for clipping Bézier curves. The simple way to do it is to convert the Bézier to its straight-line approximation to draw it, and then clip those before you draw it. If a Bézier converts to 100 lines, and the entire Bézier is outside the clipping bounds then this is very inefficient.

We can, however, do the same thing as with the straight line clipping. We know a Bézier lies within the convex hull of its control points, so we calculate the clip on the convex hull. If they all lie inside or all outside the clip bounds, then you can discard or draw the entire thing with no other clipping. You can use Cohen-Sutherland codes for 4 points this time, and if $Q_0 = Q_1 = Q_2 = Q_3 = 0$ then it can just be drawn, as above, and similarly $Q_0 \wedge Q_1 \wedge Q_2 \wedge Q_3 \neq 0$ is a complete reject. The other options you have to subdivide the cubic, and clip again.

We can improve it still further, however, but considering the *bounding box*. The bounding box is the smallest axis-aligned rectangle that encloses the control points (and hence the convex hull, and then the curve). We use this to calculate min & max X & Y of the bounding box. You can then use the equations $BB_L > x_R \vee BB_R < x_L \vee BB_B > x_T \vee BB_T \Rightarrow Reject$, and similar for accept. Other cases you have to recurse down at a finer level of detail. This uses the same number of comparisons, however is more generalisable.

8.2 Polygon Filling

Which pixels do we need to turn on inside a polygon? For the moment we assume that we turn on any pixel who's centre lies within the polygon.

8.2.1 Scan-line polygon fill algorithm

This algorithm works by scanning along each horizontal line of pixels, filling pixels as appropriate. You start by making a list of the edges of the polygon, and sort them in order of increasing Y value. The first scan line to choose is the one just above the first Y value in the edge list (the lowest one). All edges that intersect that line are moved into an active edge list, sorted by X coordinate on that scan line. You then fill every point on the line that is between the pairs of active edges. You then move up a scan line and repeat, deleting edges from the active list when they are below the current scan line, and finishing when you run out of active edges.

There are a large number of optimisations you can perform on this algorithm. Firstly, calculating the X intersect of each line with the scan line each time is slow, however, you can use the line drawing algorithm to calculate the next X position of a line, when you increment the scan line. Another one is that the sort-by-x value almost always maintains the sort in the active edge list, so a bubble sort with complete very quickly.

8.2.2 Special cases

Intersection points of lines and scan lines can be difficult. The correct way to handle this is to make one end point inclusive of the line, and the other one exclusive - i.e. if the end points are on the line they are discarded, and they are included if the start point is on the line.

The other problem is with horizontal lines. You get divide by zero errors when calculating the X intersect. In fact, horizontal lines add nothing to the shape, you can discard them

8.3 Clipping Polygons

We want to be able to clip arbitrary polygons against other arbitrary polygons. In fact this algorithm will only allow you to clip against convex polygons. Because clipping against polygons is not obvious, we reduce it to clipping against N infinitely long edges, aligned with our desired polygon's edges.

Clipping a polygon against a line is easy. We start with one vertex of the polygon, and move round it until we get back to the start, noting when we cross the clipping line. You output a new set of points such that: if an edge is entirely within the clip bounds, we output the end point, if we cross the line going out of the bounds we output the intersection point with the edge, if we cross going into the clip area we output both the intersection point, and the end point, and if the edge is completely outside the clip area, then we output nothing. This gives us another set of points that describes exactly that bit of the polygon that is on the "in"side of the line. Repeat this for all edges of your clipping polygon, and you have a shape which is entirely within the bounds, and can safely be drawn with no further checks.

8.4 2D transformations

It is extremely useful to be able to transform predefined objects to arbitrary locations, orientations and sizes. Basic 2D transformations are:

scale	about origin by factor m	$x' = mx$ $y' = my$
rotate	about origin by angle θ	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$
translate	by vector (x_0, y_0)	$x' = x + x_0$ $y' = y + y_0$
shear	parallel to x axis, by factor a	$x' = x + ay$ $y' = y$

We calculate these using matrices. We perform 2D matrix multiplications to perform all of the above transformations, except for translation. To do translations we use *homogeneous* coordinates (x, y, w) where $(x, y, w) \equiv (\frac{x}{w}, \frac{y}{w})$.

9 Lecture 8: Transformations, Bounding Boxes

9.1 Homogeneous Coordinates

For most of the transformations we can use normal 2D matrices, but since we need Homogeneous coordinates for translation, we use them for all the transformation. You move the transformation matrix to a homogeneous matrix by adding a RH column of 0 0 1, and a bottom row of 0 0 1.

9.2 Concatenating Transformations

Because we often want to apply several transformations to an object, and since applying a transformation is an expensive operation, since things tend to have a lot of Bézier points (e.g. fonts), we want to add the transformations together first, and then only apply the sum once. We can combine them by multiplying the matrices together - however, we must multiply them in reverse order to the order we want to apply them.

9.2.1 Rotation / Scaling etc about an arbitrary point

The rotation and scaling transforms are only about the origin. To perform them about arbitrary points, we must translate them to the origin, then rotate or scale, then translate them back. We can calculate this into one matrix as above.

9.3 Bounding Boxes

As with clipping, we can use bounding boxes to speed up many operations. With clipping we use bounding boxes to quickly determine 3 states. Definite accept / reject, or unsure. We only need to apply detailed clipping algorithms to the unsure objects. We can also define objects as a hierarchy, starting at the whole world, and then moving slightly finer each time you have a 'unsure' result on bounding box clipping, down to Bézier curves, which you can then apply normal clipping to - this way you can often discard large areas in one go.

9.4 Bit block transfer (Bitblt)

It is sometimes preferable to pre-draw sprites, and just copy it to the correct place on the screen when its required. Copying an image from place to place is essentially just a memory copy.

9.5 XOR drawing

Generally we just overwrite what is on the screen when we draw something. Sometimes, however, we want a temporary object, and to restore the original afterwards (e.g. in drag areas). You can do this by XORing in the line you want to draw, and then repeating to restore the original. This is great in monochrome, because you switch black / white. With grey-scale, or colour, whoever, it sometimes doesn't work as well (its hard to get nice colours).

9.5.1 Applications: 1 - UI

User interfaces tend to use objects which are quick to draw (lines, filled rectangles), and for anything complicated, just sending bitmaps (text, icons etc)

9.5.2 Applications: 2 - typography

Nowadays we can actually render Bézier fonts, however for a long time (and still in places) it is done with pre-drawn bitmaps.

9.5.3 Applications: 3 - Postscript

Postscript is an industry standard rendering language for printers. It is actually a stack based interpreted programming language. It has lots of basic operations, such as Béziers, filling objects, all the 2D transformations, and half-toning etc.

10 3D Computer Graphics

- 3D $\Rightarrow$ 2D projection
- 3D versions of 2D operations (clipping, transforms, curves & surfaces)
- 3D scan conversion (depth-sort, BSP tree, z-Buffer ...)
- sampling
- lighting
- ray tracing
- texture mapping

10.1 3D $\Rightarrow$ 2D projection

to make a picture you have to project a 3D world onto a 2D one (like in photography). There are two types of projection. *Parallel projection* merely ignores the z coordinate - but this looks unrealistic, and is mainly used for CAD applications. In *Perspective projection* you have to take the z axis into account. The simplest of these is to say $(x', y') = (\frac{x}{z}, \frac{y}{z})$.

10.1.1 Viewing Volume

When projecting we are looking in a certain direction, and projecting onto a plane some distance away. A point a distance z away, can be projected onto the screen by converting x & y, using similar triangles. This works nicely if the camera is positioned at the origin. If there are arbitrary camera positions, we can either work out equations for projecting arbitrary points onto arbitrary planes, or we need to transform all the objects in the world into the coordinate system of the camera.

10.2 3D transforms

These can all be done similarly to 2D with matrices. Again we have to use Homogeneous matrices, so we have vectors (x, y, z, w), and conversion from 3D to homogeneous is again similar to 2D, and only the translations use the extra column.

Homogeneous Matrices can be broken into several sections. A 3x3 affine transform matrix, a 3x1 translation matrix, the extra w corner is a global scale factor.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ p_1 & p_2 & p_3 & s \end{bmatrix}$$

Here, a is a 3x3 affine transform matrix, which can do scales, rotation etc. t is a translation vector, s is a global scale factor, and p is a perspective transformation

With these transformation matrices, we can pre-multiply the matrices before applying them to the world coordinates. It turns out we can also include projection in matrix transforms, so it can all be done in 1 operations.

10.2.1 Standard Transforms

Scale by s:

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2D Clockwise rotate by θ:

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In a 3D rotate you're rotating about 1 axis, that coordinate is the identity, the other 2 have the sin/cosine. E.g. rotate about y axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation by (x_0, y_0, z_0):

$$\begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

11 Lecture 9: Viewing Transforms

We need 3 sets of numbers. The location of the eye, a point we are looking at, and an up vector. First we translate the eye point to the origin with a translation matrix. Secondly, we have to scale so that the look-point distance is the distance onto the screen. Next we have to align the system correctly. This is more difficult. We need to align $\bar{el}$ with our new z axis. First we transform e and l into our new coordinate system, then rotate $\bar{e'l'}$ into the yz plane, by rotating about y-axis (to set x to 0). We then need to zero the y value, by rotating about the x axis, similarly to above. This aligns the look-vector with the z-axis, but we could still have the up vector pointing in the wrong direction. This is a single rotate about the z-axis. This leaves us with 3 rotation matrices, a scale, and a translation.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} \times R_3 \times R_2 \times R_1 \times S \times T \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

you can use the resulting matrix to transform every point into the appropriate coordinate system for projection.

We in fact usually have more transformation than this. To go from the *object* coordinates to *world* coordinates you have a modelling transform, then from world to *viewing* coordinates we have the transform above. We can combine the modelling transform and the viewing transform into 1 matrix.

11.1 Modelling Transforms

For example you could represent all cylinders as centred on the origin and of unit radius and height in object coordinates. To place it in your world, however, you may wish to change the size, orientation and position, etc. This is done via a similar set of transformations as above, but you are doing to reverse. In the viewing transform, you are taking arbitrary vectors and mapping them onto the axis, in this case we are mapping a standard vector on the axis onto an arbitrary vector. This easiest way to do this is to calculate the transform as before, and then apply it in reverse. (reverse the order of the rotations, and negate the angles you rotate by). Translation and scaling is obvious, and you multiply all the matrices together, as before - scaling first, then rotating (since these operate about the origin) before translating away from the origin.

You can create a single object $\rightarrow$ viewing matrix by multiplying the modelling and viewing transform matrices together, but that product is a per-object transformation.

11.2 Projection

The perspective vector in homogeneous 3D matrices can be used for projection. As noted before, we want $x' = x\frac{d}{z}$ and $y' = x\frac{d}{z}$. We can do this with homogeneous matrices using a matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

After converting back to 'real' coordinates from homogeneous coordinates, we find that all the z coordinates are d, and the x & y coordinates have been appropriately scaled for perspective.

11.3 Do we need to know all this?

Generally, if you are using a graphics library, all the matrix details will be hidden from you - you'll probably just have to specify the three initial vectors. You will have primitives such as `translate(tx,ty,tz);`. This tends to build a *current transformation matrix*, into which you can plug (x, y, z) values, to get a transformed set of (x', y', z'). It is hence useful to know how the system works, even if we aren't directly implementing it ourselves. For example, drawing the cylinder, appropriately transformed might look like:

```
initCTM();
scale(2,1.5,2);
rotateZ(-19.45);
rotateX(-45);
translate(1,3,2);
cylinder();
```

12 Lecture 10: 3D-2D equivalents

12.1 Clipping

In 3D we are clipping a 3D object against a *volume*. This is relevant for projection, since you need to clip within the pyramid made with the screen at the base, and your eye at the point. You also need to make sure that you don't consider things with negative z, wrt the eye. This is obvious, however, you also need to consider the objects very close and very distant to you. Usually you discard objects immediately in front of the eye, and those in the very distance, by defining a front and rear clipping plane, and clipping only within them. This defines a *frustrom*.

This gives us 6 planes to clip against, where two of them are simple z clips. You can project the others to 2D, without considering z, and then reduce the equations to simple 2D clipping. This may require converting lots of points to 2D, so in some circumstances doing the 3D clip is more efficient, although harder to program.

12.2 Bounding boxes

We can also define bounding cubes or spheres, which behave exactly like bounding boxes in 2D.

12.3 Curves in 3D

Curves in 3D have an extra coordinate per point, but that is it - the equations remain the same. When drawing them, we divide them up as before, then just project the lines onto the viewing plane and draw them.

12.4 Polygons

With 3-vertices polygons they are guaranteed to be in a plane, and can just be projected and drawn. With more then 3 vertices, then they may be non-planar. Any planar polygon can be drawn trivially, however, non-planar polygons need special treatment.

12.4.1 Triangles

As we said, triangles are easy to draw, since they are always planar. They are also a frequent basic unit used by graphics hardware for drawing. They are also useful for making optimisations to the polygon drawing algorithm. Drawing triangles is simple, and easy to implement in hardware. For once there are no special cases.

You can draw flat non-planar polygons easily, by simply breaking them up into triangles, and drawing those.

12.5 Surfaces in 3D

The polygons we drew had flat surfaces, but what about arbitrary curved ones? We use a *Bézier Patch*, which uses 4 Bézier curves, one along each edge, and another 4 internal control points to define the surface. It therefore has 16 control points $P_{0,0}...P_{3,3}$. The formula, which is parameterised in 2 directions, is

$$P(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(s) B_j(t) P_{i,j}$$

12.5.1 Joining surfaces

We have the same continuity definitions as with curves. Surface C_0 continuity implies continuous in position, so the four control points for the edge Bézier must match. C_1 continuity needs the tangent vectors to match, so the next 4 control points away from the edge must be *co-linear* with their opposites, and also *equidistant* from the curve.

12.5.2 Drawing surfaces

As with Bézier curves, we need to divide up surfaces before drawing them. We divide Bézier Patches up into planar polygons to draw them. The easy way to do this is to define a resolution you wish to draw at, and just splitting the surface into that many polygons.

As before, we can produce a better approximation by recursively testing approximations, and either drawing it if its OK, or splitting the patch, and trying again with a better resolution of approximation. Once the quadrilaterals are good approximations of the curve you need to convert them to planar triangles to draw. However, the approximations may leave gaps in the surface, where some of the surface is drawn at one resolution, and some is drawn at another. You need to alter the convert-to-triangle algorithm to match up the all the new points on the surface.

12.6 3D scan-line conversion

How do we actually draw things in 3D?

12.6.1 lines

Given a list of 3D lines we draw them by projection their end points onto 2D, and then drawing them with the 2D line algorithm. This produced a wire-frame model. However, often we don't want to draw lines behind other lines. This used to be the topic of lots of work, but now we have dedicated polygon-drawing hardware which will do this for us, so we just draw polygons.

12.6.2 Polygons

When drawing polygons, we can project them onto 2D, and then use the 2D algorithms. If we retain the z information, we can simply sort them by z coordinate, and then draw them back-to-front, so that things behind get drawn over. There is a faster method called Binary Space-Partitioning Tree (used in quake).

However, there is a better method, however, that you can just draw them in any order, called z-buffer.

12.7 Depth Sort Algorithm

This method is quite simple.

- transform all polygon vertices into 2D, keeping z informations
- calculate the polygon depth, based on the most distance z coordinate
- resolve ambiguities of overlapping-z by splitting them
- draw the polygons in the order back-to-from in z, so that later polygons are drawn on top of earlier ones

Steps 1 & 2 are simple, step 4 is just the polygon scan line algorithm. 3 is more difficult.

12.7.1 Resolving z ambiguities

We perform several tests, in order of complexity, and if any of them passes you can just draw them. First we check if they overlap in any of x, y or z using bounding boxes, and if they don't in any one of them, you can just draw them in the order you had. Next we check whether they are entirely on appropriate sides of each other's planes wrt the view-port. You may have to switch the order of the shapes you are comparing, in which case draw them in the other order. If all of these fail, you have to split P or Q, and try again with the smaller polygons. We want to do this intelligently, so we split it along the intersection with the other shape.

Once you've checked them all, simply draw from the rear-most polygon.

13 Lecture 11: Depth algorithms

13.1 Depth Sort Comments

Depth sort algorithm is reasonably cheap for small numbers of polygons, and is an $O(N \log N)$ algorithm. However, with large numbers of polygons it is in-efficient. The ordering is also only valid from 1 viewpoint.

13.1.1 Back Face Culling

We can improve the algorithm by removing any polygons whose normal points away from the camera (assuming it is part of a larger polyhedron) since they will be hidden by the front ones.

13.1.2 Binary Space-Partitioning Trees

These were used by quake. BSP uses a very intensive pre-processing algorithm, and then can draw the polygons in $O(N)$ time. It is based on the observation that to draw one polygon, you have to draw everything behind its plane, scan convert that polygon, and then draw everything in front of it. This is a classic divide and conquer algorithm, and you can use it to build a binary tree.

You build the tree by picking a random polygon, and then putting everything in front of its plane in one child, and everything behind it in its right child, splitting ones that intersect its plane. You then recurse on either side. You have to be careful to create a balanced tree, but this step is done off-line, and can take as long as you like.

Drawing BSPs........

13.1.3 More Optimisations

The scan line algorithms work by taking each polygon, and using an edge list and active edge list to draw within them. We can actually draw all of the polygons at once by putting all their edges in the active edge list. The only thing you have to be careful about is to check whether an edge is valid (not behind another polygon). This makes it very complicated, and has a large memory requirement. However, if we can do this - we can do the next algorithm, which is better.

13.2 z-buffer

This algorithm is easily implementable in hardware, and doesn't worry about sorting.

You store for each pixel a colour and a depth. When converting a polygon, you use a standard 2D scan line algorithm, but keeping z values, and then before drawing the pixel, you check whether it is 'nearer' than the one currently in the buffer. If it is, you update the colour and depth of the pixel, otherwise you don't.

This doesn't require a sorted polygon list, and is hence quite a lot more efficient. You do need to store colour and depth of each pixel, but thats not a lot more. The only difficulty is calculating the depth value at each pixel in our polygon scan line algorithm.

Naïvly we could just linearly interpolate the z values over the 2D (x, y). This will produce a bad approximation, since x & y depend on z in the projection. It is an easy modification to make to get the correct answer, which is: $\frac{1}{z} = (1-t)\frac{1}{z}_1 + (t)\frac{1}{z}_2$.

Algorithm	Complexity	Notes
Depth Sort	$O(N \log N)$	Need to resolve ambiguities
Scan line	$O(N \log N)$	Memory intensive
BSP Tree	$O(N)$	$O(N \log N)$ preprocessing step
$z$buffer	$O(N)$	Easy to implement in hardware

13.3 Sampling

We have so far assumed that you only fill in the pixels whos centres are within the polygon. This leads to ragged edges, and polygons that only get drawn in certain orientation. These problems are called *aliasing*, and techniques for removing it are called *anti-aliasing*.

13.3.1 Anti-Aliasing 1:

This method assumes that pixels are square, then work out how much of the pixel is covered by each polygon, then averages the colours for that pixel. This is the ideal way to do it, which Pixar use, but it is a hideously expensive algorithm, and involves clipping to every pixel.

13.3.2 Anti-Aliasing 2:

This method increases to resolution we calculate to. We calculate the smaller pixels as before, selecting an appropriate increased resolution you want. Typically about 8×8. Then you take the average of the smaller pixels to get the colour of the larger one. This is slow when they are all the same colour anyway.

13.3.3 A-buffer

This is an optimisation of the above algorithm. We divide the pixels into 3 types. Those completely within a polygon, those completely outside, and those which are partially covered by the polygon. You only need to perform the super-sampling method on those which are partially covered. For each pixel, you store a mask of bits corresponding to which sub pixels are covered by the polygon. Store one of these per polygon partially covering the pixel, and sort by depth. You then in order front-most to rear-most AND the current mask with the bitwise inverse of the nearer masks to find which pixels are visible in that colour. You can then average the colour over the visible sub-pixels. You can also optimise whenever a polygon covers the whole pixel, you can discard all of the ones behind it.

Calculating the masks is done by clipping the polygon to the pixel (only when you know there's an edge pixel), and calculate the mask for each edge, bounded by the right hand side of the pixel (you can store this in a lookup table, since there are a small number of possible edges), and then you XOR them all together.

14 Lecture 11: Illumination and Shading

Up until now we have only considered drawing polygons in one given colour. To render a scene more realistically, we want to calculate the shading on the objects due to lights.

In the real world, every light source emits millions of photons. These bounce off, pass through or are absorbed by objects. In fact most of them are absorbed by objects - which makes this an incredibly bad way to calculate on a computer.

If we just consider the properties of surfaces wrt reflection, there are 3 general types of reflection. Perfect reflection (such as mirrors) sends all the light that hits from one direction out of the corresponding opposite direction. Specular reflection is for metallic surfaces, which have many small facets that perfectly reflect, but at slightly different angles. You have a series of output rays, cantered on the line of perfect reflection, and with less intensity as you move away from the line of perfect reflection. If you look at a really close angle to the specular surface, the facets start to occlude each other, and interfere. The third type is diffuse (Lambertian) reflection, where you get an even spread at all angles, no matter what angle your light comes in.

Some surfaces, such as plastic, have some features of both specular and diffuse reflection.

Obviously different wavelengths of light will be reflected to different amounts by all of these.

14.1 Shading Polygons

To shade polygons we will make lots of assumptions. Firstly we will assume there is only diffuse reflection. Secondly, we will assume that all light falling on a surface is direct from the light source, we will ignore shadows, and assume that all of the light sources are infinitely far away, and can be represented just as a vector. This results in a flat surface being all the same colour, depending on its orientation to the light sources.

This reduces the shading calculation to $I = I_l k_d (N\dot{L})$, which gives you the intensity of light falling on a surfaces. (L is the normalised vector of the light source, N is the normal to the polygon, I_l is the intensity of the light source, K_d is the proportion of light reflected from the surface and I is the resulting intensity of the reflected light).

This formula allows different colours of light (I_l and k_d). The things to watch out for are if $\cos\theta < 0$, or the light source is behind the surface. How you treat this depends if you consider a polygon to be 1 or 2 sided.

14.2 Gouraud Shading

If we want to smoothly shade objects across a polygon, for curved surfaces approximated to polygons, you can use Gouraud shading. This works by calculating the colour at each vertex, and then smoothly interpolating between them. This adds another step to our polygon algorithm. We now project x, y, z, and r, g, b. If we are calculating from a curved surface originally, then we can calculate the normal vectors at all the points as we go. These are needed to calculate the colour at each point. The normal at the vertex is also $N_v = \frac{1}{n}\sum_{i=1}^{n} N_{T_i}$.

14.3 Specular Reflection

We would quite like to be able to model specular reflection as well as diffuse. This is very difficult, so we use approximation. *Phong's Approximation* uses the perfect reflection vector, and a vector to the viewer to get $I = I_l k_s \cos^n \alpha = I_l k_s (R\dot{V})^n$. This is similar to Gouraud shading, but calculate the specular component in addition to the diffuse component. You also have to interpolate the normal vector over the surface.

14.4 Gross Assumptions

We have seen we can deal approximately with specular reflection, but we still take no account of shadows - this needs ray-tracing. You can add local light sources, but this requires you to interpolate the l vector as well. Handling the light reflected from other surfaces can be made *very* approximately, by calculating an ambient amount of light that hits everything in the scene from all angles.

14.5 Multiple Lights

You can calculate the total shading on an object to be the sum of the ambient illumination, plus the sum of all the diffuse and specular reflections. Note that this can result in more than 100% of light reflected, for bad choices of k_a, k_s and k_d.

14.6 Ray Tracing

This is an alternative to polygon scan conversion. For every pixel in the screen you are projecting onto, you plot a ray from the eye, through the pixel, and you calculate which object it intersects first. This is the object you draw at that pixel. The algorithm is quite simple to describe too:

```
select an eye point and a screen plane

FOR every pixel in the screen plane
   calculate the ray from the eye through the pixel
   FOR every object in the sphere
      IF object intersects ray
         IF intersection is closest
            record point and object
         FI
      FI
   ROF
   set colour to that of the closest object
ROF
```

Compared to polygon scan line conversion, this is very expensive, since we must calculate an intersection $n \times m$ times (number of pixels and objects).

Calculating the intersection is quite simple, but you have to be careful to avoid divide by zeros or negative distances from the eye.

14.6.1 Intersection With polygons

You can calculate whether you intersect a polygon simply using the polygon clipping algorithm. You calculate the point at the depth of the polygon, and you just clip the single point against the polygon.

14.6.2 Intersection with Spheres etc

Using the dot product notation for a sphere, and the equation for the ray using the parameter s, solve simultaneously to get a quadratic, and solve the determinant for s to find out if s is real or imaginary. A real value of s implies that the ray intersects the sphere, and the smallest positive value is the closest intersection point. Negative values imply the intersection is behind you.

15 Lecture 12: More Ray Tracing

15.1 Shading with ray tracing

Once we've calculated the intersection point, we calculate the vectors $L_1, \ldots, L_n$ to the light, and calculate the reflections $R_1, \ldots, R_n$, and the normal and viewer vectors to calculate diffuse and specular reflections for that point.

15.2 Shadows

If instead of just calculating the vectors to the lights, we trace a ray to it, we can tell if an object obscures the light ray, and then you don't include the light in the reflection calculations. You have to watch for floating point inaccuracies which put the intersection point slightly within the object, and hence all rays to lights get obscured by itself.

15.3 Reflection

We can now also handle perfect reflection, by tracing rays off reflective objects, to see what it hits. You can set a percentage of light generated from the reflection. You have to ensure you don't get infinite 'bounces'. This can be done either by limiting the depth of reflections, or the percentage contribution we are providing - the last has difficulties with perfect mirrors.

15.4 Transparency

We can also handle transparency and refraction, by specifying how much of the light contribution is from the normal reflection, and how much you can generate by tracing a ray through the object to find things behind it. We can also insert refractions into rays passing through the objects.

15.5 Sampling in ray tracing

We can sample using a single ray through the centre of the pixel. However, we probably want a better resolution, so we can super-sample (just do a large number of them) or adaptive super-sampling, where you use a few well-chosen rays, and only if they differ significantly do you do the rest.

Previously, we've had to use a regular grid of super-samples. This isn't the case in ray tracing. We can use a random selection of rays. This replaces the aliasing artifacts with noise, but the eye is less sensitive to that than to aliasing.

Just a random distribution gives you a lot of noise artifacts, so you can use a *Poisson Disk* algorithm to put the rays on the pixel, which says that no ray may be within ϵ of any other, This produces a really nice effect, but is expensive to calculate. We approximate using a *jittered* method, which divides the pixel into sub pixels, but using a random location in each sub-pixel. This gives a much better image than a regular grid.

15.6 Distributed Ray Tracing

This means distributing the samples over a range. You can distribute over the pixel using super-sampling as we've seen for anti-aliasing, but you can also distribute rays going to a light source over its area, which allows for area, not just point, lights, and gives you soft shadow effects. We can also distribute the camera position, to simulate a camera with a finite aperture lens, and depth of field artifacts. We can also distribute the samples in time to give motion blur effects.

This allows us to produce lots of effects that real cameras have.

Earlier we used an approximation for specular reflection. If you distribute reflection rays around the reflection point, you can calculate specular reflection a lot more realistically. This gives us a much better approximation of the interactions of objects in the scene.

The case this doesn't cover is calculating inter-object reflections with diffuse reflections. You can use a method called radiosity in some cases, but in others there isn't a usable algorithm.

15.6.1 Multiple inter-reflection

Light may reflect off many surfaces on its way from the light to the camera. The general case is $(d|s)*$

Our algorithms can handle some subset of this:

Algorithm	Reflection Type	
Standard ray tracing / polygon scan conversion	One diffuse or specular bounce	$d\|s$
Distributed ray tracing	Multiple specular bounces	$(d\|s)s*$
Radiosity	Multiple Diffuse bounces	$d*$
No General case algorithm		$(d\|s)*$

15.7 Hybrid Algorithm

Polygon scan conversion and ray tracing are our principle two methods of 3D rendering. You can combine these, using polygon scan conversion for flat rendering, but using ray tracing for the detail areas.

16 Lecture 13: Surface Detail

All our work so far has assumed that the objects have been evenly coloured. Mostly we actually want textures on them.

16.1 Texture Mapping

Applying a texture to a polygon, we store another set of coordinates (u, v), which we interpolate along with polygon scan conversion, to calculate which bit of the texture should be drawn. The calculated value won't map to a pixel on the 2D flat texture, because it had be projected. Two methods for calculating what to draw. *Nearest Neighbour* draws the pixel closest to the calculated value. *Bi-linear reconstruction* takes a weighted average of the neighbouring pixels, and this generally produces a better result - if you are scaling the texture larger, then it will have the effect of generating extra pixels between the ones of the texture, which are interpolations of the ones we already have.

Nearest neighbour and bilinear are both very fast and can be done in hardware, but have artifacts. Better results can be obtained from bi-cubic and bi-quadratic (using 16 or 9 squares of interpolation). Bi-cubic runs at a quarter of the speed of bilinear, and is very good. Bi-quadratic is faster than bi-cubic, and nearly as good.

Downsampling can also have problems. If one pixel is a lot larger than a texture pixel, then you will only get the centre texture pixel mapped onto the pixel, when you want an average of quite a lot of them. One way to solve this is to store multiple resolutions, pre-computed by averageing lots of pixels as you go down. Now, when rendering a texture between two of the stored ones, you can do bilinear on both of them, and then linearly interpolate between them to the appropriate size. This is known as trilinear interpolation.

16.2 MIP map

This is an efficient memory arrangement for a multi-resolution colour image. The largest image is split into red, green and blue parts, stored in the three quarters of the map. The top left corner of the MIP map contains the next resolution down, in a similar arrangement.

16.3 Solid textures

Solid textures have a colour defined for every point in space $f(x, y, z)$.

16.4 Texture Map modifications

Texture maps can modify all of the colour components, all the reflection type, transparency - and also the surface normal using a technique called *Bump Mapping*.

17 Lecture 13b: Image Processing

17.1 Filtering

The basic technique is that you move a filter across the image, altering the the pixel values. The filter could be a multi-pixel grid, and for each pixel in the centre you multiply all the filter and pixel values, then sum them to get the value for that pixel, then move the filter.

17.2 Discrete Convolution

In one dimension:

$$I'(x) = \sum_{i=-\infty}^{\infty} h(i) \times I(x-i)$$

where I is the image, and i ranges over the size of the filter h. In 2D we get

$$I'(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(i,j) \times I(x-i, y-j)$$

We have the problem that the edge values don't have enough adjacent pixels to convolve over. Usually, we assume the non-existent pixels are zero, and use that in the sum.

17.2.1 Blurring

We can blur an image by just averaging every pixel with its neighbours - either just straight, or we use a 2D Gaussian for the filter.

17.2.2 Edge Detection

There are several types of edge detection, *Prewitt*, *Sobel* or *Roberts* - in horizontal, vertical or diagonals, and basically having a row of negative values, and row of zeros and a row of positive values - oriented in the direction of the edge to detect. When you have values the same, you average to zero, and where the image varies a lot, it gives you large positive or negative values.

17.3 Median Filters

There are other methods than convolution for doing filtering. Median filtering takes an area of pixels, and uses the median value of those pixels for the centre value. This is very good at removing 'shot' noise, since the individual black/white pixels will be away from the median. On random noise median filtering and Gaussian blur do about the same.

17.4 Point Processing

Point processing can do a lot of things. In the simplest case we can invert an image just by processing an image at a time. We can also change the contrast of a image. This is done by applying a function to the image, that can be seen as a graph.

17.4.1 Gamma Correction

The intensity on a CRT is related to the voltage on the electron gun. This function is $i \propto V^\gamma$ where $1.5 < \gamma < 2.5$, and the voltage is directly proportional the pixel value. Since we want the $i \propto p$, we generate $p' = p^{1/\gamma}$. This is a point processing operation. CRTs generally have $\gamma \approx 2.0$, but needs to be set per monitor.

17.5 Image Compositing

We often want to merge several images together. What the composite operator does, is take two images, and a mask that defines which pixel is taken from each image. We can use point processing to define the mask from one image (e.g. background colour is one mask value, and other pixels are the other mask value). This can produce jagged edge artifacts, so we use a grey-scale mask, which allows us to blend the pixels at the edge as a function of both images.

17.6 Arithmetic Operations

We can apply arithmetic operators to images

17.6.1 Difference

If we want to check the differences between two images, where $d = 1 - |a - b|$. This produces black values where there are differences, and white where there aren't.

18 Lecture 14: Mapping to Monochrome, and Image Compression

Half-toning is a technique for printing shaded images using a binary colour system. For grey-scale to black and white, you map each grey-scale pixel to several black or white dots. Half-tone patterns are quite difficult to get right - at level 3, which three do you turn on. Horizontal/vertical lines aren't good, because large areas of the same colour will have noticeable lines. The rules for half-toning are that you must have a growth sequence (all dots on in level j must be on in $j + 1$. Secondly, you have issues with actually printing, that ink tends to splurge. Isolate dots tend to vanish, and small holes in the middle of areas of ink tend to fill up. You should also only fill up from the centre of the pixel, to simulate a growing dot.

Colour half-tone screens, as we mentioned at the beginning of the course, need to be at different angles

18.1 Dithering

If you can guarantee that ink spread isn't a problem, then you can use a different method. *Ordered dithering* is very similar to half-toning, but you try to separate the dots as much as possible, rather than clumping them in the centre.

18.2 1-to-1 Pixel Mapping

Both the above algorithms work great for mapping one grey-scale pixel to multiple black and white dots. This can be done by making the top half of the colour values -¿ white, and the bottom half to black. A better image can be made from taking the dither map, and using the corresponding dither map as the threshold.

A better method, however, is *Error Diffusion*. Here you round to the nearest black/white, and then distribute the amount you had to round (the *error*) to the adjacent pixels. You start at the top left of the image, and distribute half the error right, and half the error down (positive or negative, depending which way you round) and add it onto the pixel value there before calculating its value. In fact this distribution isn't too good. If we are processing right to left and top to bottom, there are four adjacent unprocessed pixels to the one we are currently rounding. *Floyd-Steinberg* who devised this method used the proportions 1, 3 5 and 7 16ths, with more going right, and then down, then diagonally down and right - otherwise the errors propagate too fast.

18.3 Encoding & Compression

Image data is very large, and tends to be very compressible - since adjacent pixels tend to be very similar. Encoding is therefore possible and necessary.

Encoding is generally broken into several stages:

1. Mapper - maps pixel values to some other set of values, designed to reduce inter-pixel redundancies.

2. Quantiser - designed to reduce psycho-visual redundancies, by reducing the accuracy of the mapper's output

3. Symbol encoder - encodes the quantiser;s output, using standard compression / encoding techniques. Designed to reduce symbol redundancies.

The first stage is designed to optimise the image data to be more easily compressible. Lossless compression misses out the quantiser stage, and you can get back exactly what you put in. This can give you about a 2:1 compression ratio. If you have a good quantiser, you can get a 20:1 compression ratio, without people noticing the difference. For post-processing, and medical, you want to have lossless encoding.

18.3.1 Symbol Encoding

Difference mapping considers the differences between the adjacent pixels. Pixel colour distribution tends to be roughly even, however, differences in pixel values tends to be a very tight Gaussian around zero. This tends to be a lot better for variable length codes.

Predictive mapping looks at more than just the adjacent pixel. As before, when calculating any one pixel, there are four adjacent pixels that we have already drawn. We can therefore use a function of the pixel to the left, and the three pixels above.

18.3.2 Run-length encoding

This is designed to work with smooth computer generated images. c.f. Data structures and Algorithms course. This works by mapping runs of identical pixels to a count and a value. This works really well with many runs of identical pixels, but in the worst case can cause slight expansion!

18.3.3 CCITT Fax encoding

This is a variant on run-length encoding. Since faxes are binary you only need to store lengths, not pixel values, and you can also do it in two dimensions.