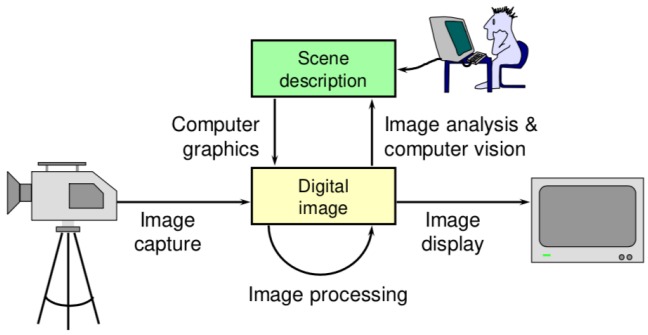
Graphics

Background

Computer Graphics and image processing is the process of taking a scene description and converting it into some sort of image which can be processed.



It is very important, since all visual computer output (and every kind of visual imagery) depends on CG:

- Printed output
- Monitor
- TV, movie special effects and post-production
- Books, magazines, catalogues, brochures, junk mail, newspapers, packaging, posters and flyers.

Vision

There are three requirements for vision:

1. Illumination
2. Objects
3. Eyes

It should be noted that light that we see is only a small subset of the entire electromagnetic spectrum. This is the visible light spectrum, which is between 300nm and 800nm. (**The long wavelength is red and the short wavelength is violet**).

What is an image?

An image is a two dimensional (continuous function) f(x, y) - the value at any point is an intensity / colour. It is important to be noted that an image is not digital per se, the function is completely continuous.

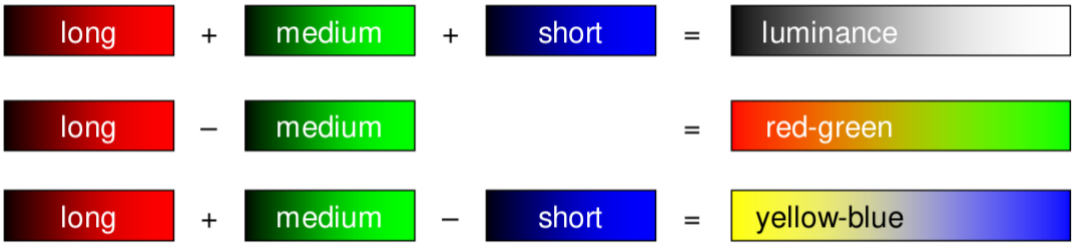
Eye

- **Structure of the eye**
    - The lens focuses light onto the retina (an array of light detection cells in the back of the eye).
        - The shape of the lens can change to change the distance on which we are focusing.
    - The **fovea** is the high-resolution area of the retina (has the greatest concentration of cones).

    - The optic nerve takes signals from the retina to the visual cortex in the brain. It is here the pieces are pieced together and we can perceive an image.
- **Light Detectors**
    - There are two classes of detector: **rods** and **cones**
        - The cones come in three types: sensitive to short, medium and long wavelengths.
    - The cones are concentrated in the **macula**, at the centre of the eye
    - **The fovea is a densely packed region in the centre of the macula**: it contains the highest density of cones (therefore highest resolution vision)

Colour Signals

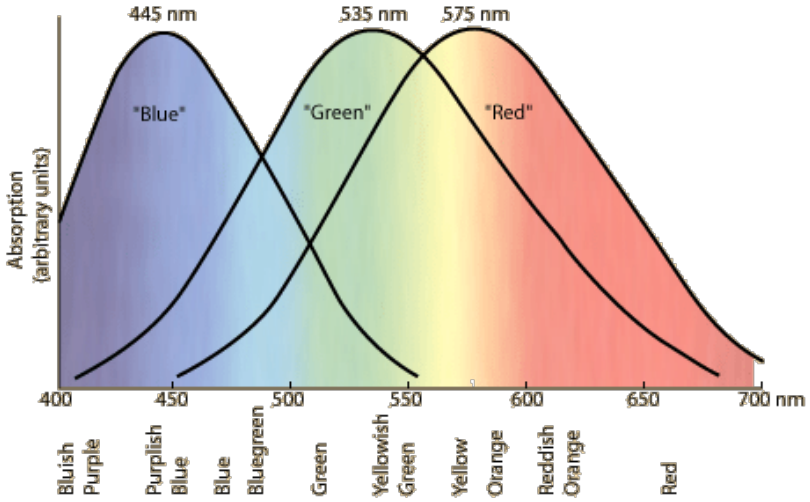
The signals which are sent to the brain are pre-processed by the retina:



It is easy to find the scale for which they correspond to by simply consider what would happen in the maximum and minimum.

These shows why red, green yellow and blue are perceptually important colours.

It is possible, by mixing different amounts of red, green and blue lights, to generate a wide range of responses in the human eye – but it must be noted that not all colours can be created this way.



Colour-Blindness Effects

Colour blindness can be caused by one of two methods:

1. The person is missing a cone (therefore has only got two cones).
2. The person has a shifted cone

a. This means that colours between the two are generally harder to differentiate, therefore may not be able to be differentiated.

Digital Image

This is a contradiction in terms: if you can see it, it's not digital. However, we define a digital image as a sampled and quantised version of a real image. It is a rectangular array of intensity or colour values.

Sampling

A digital image is a rectangular array of intensity values – each value is called a pixel (picture element). The sampling resolution is normally measured in pixels per inch (ppi). This is equivalent to the number of dots per inch (dpi).

- Computer Monitors – 100 ppi
- Lasers and Inkjet printers – 300 – 1200 ppi
- Typesetters – 1000 – 3000 ppi

Image Capture

- Lots of devices can be used:
    - Scanners
        - Line CCD (Charge Coupled Device) in a flatbed scanner
        - Spot detector in a drum scanner
    - Cameras
        - Area CCD
        - CMOS camera chips

Sampling resolution: The sampling resolution determines how many pixels the image should be made into. For each point, the colour of the actual image is averaged into the colour of the digital image which is stored.

Quantisation

Quantisation is the process of converting a continuous value of the intensity of a bit of the image into a digital number stored in a number of bits.
It limits the number of different intensities which can be stored (and limits the maximum brightness that can be stored).
For human consumption, generally only 8 bits are required (though some applications use 10, 12 or 16 – generally to be down-quantised to 8 bits).
The colour is stored normally as a number for each of the red, green and blue – with an intensity (8 bits) for each one.

Storing Images in Memory

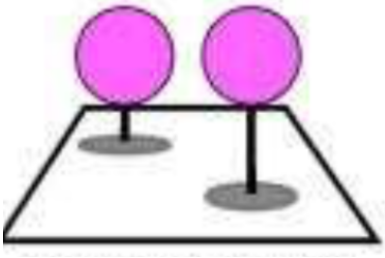
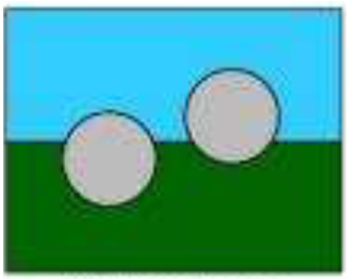
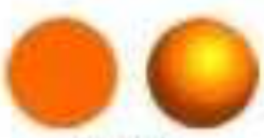
8 bits used to be the de facto standard for greyscale images – but 16 bits is now generally more used.
For 8 bits: pixel[x][y] is stored at base + x + WidthOfImage x y
This is necessary since the memory is one dimensional whereas the images are two dimensional.

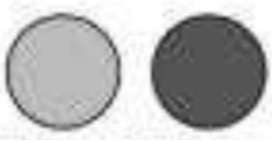
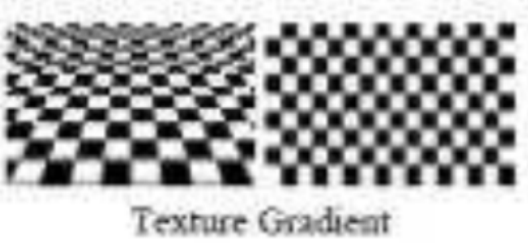
Frame Buffer

Computers have a special piece of memory reserved for storage of the current image being displayed – the frame buffer. This normally consists of **dual ported DRAM (Dynamic RAM)** – this is sometimes referred to as **Video RAM.**

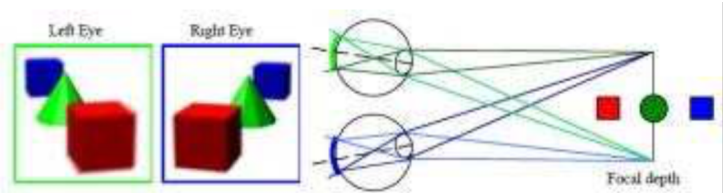
Rendering

Depth Cues

1. Occlusion
    a. The object in front covers the object behind.
    b. 
2. Relative Size
    a. Objects which are further away are smaller than objects nearby.
    b. 
3. Shadow and Foreshortening
    a. The shadow of an object on other objects and the floor as well the shortening of distances as they go further away (**Foreshortening**)
    b. 
4. Distance to horizon
    a. The closer the items are to the horizon, they are further they are away.
    b. 
5. Shading
    a. A 3D object would have shading to do with the texture and shape of the object – for example a sphere would have shadows towards the back of the object.
    b. 
6. Colour

   a. The colour of different objects may give us hints as to whether they are nearer or closer – this would be to do with the light that is near them – for example a blue light source at the back of a scene would make objects them bluer.
7. Relative Brightness
   a. For luminescent objects, depending on how bright they are, you can see how near they are to you – the brighter the closer they are. Also, depending on where the lights are this can also work for other objects. Eg. If the source is at the front of the scene objects the nearer to you would be brighter.
   b. 
8. Atmosphere
   a. If there is a (thick or thin) atmosphere, the nearer the object, the more in focus the object would be – the light would hit lest dissipate less.
   b. 
9. Focus
   a. The more in focus an object is, the closer it is likely to be. You can also tell more practically based on the focal length of the item used to photograph the thing.
   b. 
10. Familiar Size
   a. You can use items of a known size in order to gain an understanding of where the object is.
   b. In case of repeated objects, this would also be possible.
   c. 
11. Texture gradient
   a. A texture gradient (which continues backwards) which can be used to gain an understanding for where objects are.
   b. 

12. 3D images (left and right eye)
   a. In 3D images (or just real-life objects), different images are shown to the eye – therefore when we look at an object (and focus on it), we can see where it is, relative to other objects.

b. 

When we render depth, we can use all of these techniques to ensure that it can be seen where objects are relative to each other when they are displayed as two-dimensional objects.

Drawing Perspective

Throughout history, the perspective tended to be very wrong as artists failed to properly make things appear to be three dimensional.

Examples:

1. Early – Ambrogio Lorenzetti 1332 – Presentation at the Temple
2. Wrong Early – Lorenzo Monaco 1407 – 1409 – Adoring Saints
3. Renaissance Perspective – Filippo Brunelleschi 1413 – Geometrical Perspective (Holy Trinity fresco)
4. More (better) – Carlo Crivelli 1486 – The Annunciation of Saint Emidius

They were also fascinated by the idea of false perspective – the idea that 'impossible' designs could be drawn by warping perspective; Also, by using clever steps and changing the size of objects, you could make someone make someone look much larger than the other person.

Calculating Perspective: In the old days, the perspective would be found (when drawing an object) by using a piece of string from behind the painter going to the point and finding the point it went through the canvas and marking it for a number of points.
Then these could be joined together to make an object in perspective.
This method is known as ALBERTI'S ONE POINT PERSPECTIVE

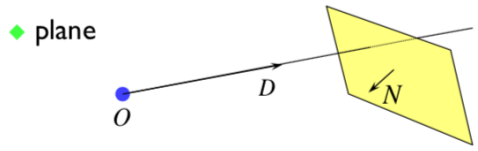
Ray Tracing

- The method involves finding the point on the surface and calculating the illumination.
- Given a set of 3D objects, you shoot a ray from the eye through the centre of every pixel and see what surface it hits – this defines the colour of the pixel.
- It easily **handles reflection, refraction, shadows and blur**
- **But it is very computationally expensive.**

```
select an eye point and a screen plane

FOR every pixel in the screen plane
    determine the ray from the eye through the pixel's centre
    FOR each object in the scene
        IF the object is intersected by the ray
            IF the intersection is the closest (so far) to the eye
                record intersection point and object
            END IF ;
        END IF ;
    END FOR ;
    set pixel's colour to that of the object at the closest intersection point
END FOR ;
```

Calculating intersections of ray with object

Plane:



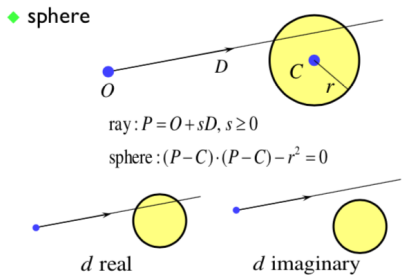
ray: $P = O + sD,\ s \geq 0$

plane: $P \cdot N + d = 0$

$$s = -\frac{d + N \cdot O}{N \cdot D}$$

Polygon or Disc: Intersection of the ray with the plane of the polygon and then check whether the intersection lies within the polygon.

Sphere:



$$a = D \cdot D$$
$$b = 2D \cdot (O - C)$$
$$c = (O - C) \cdot (O - C) - r^2$$
$$d = \sqrt{b^2 - 4ac}$$
$$s_1 = \frac{-b + d}{2a}$$
$$s_2 = \frac{-b - d}{2a}$$

Cylinder:

Deal with it is as two discs as the top and bottom discs then deal with the finite curved surface of the cylinder:

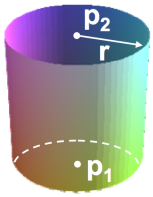
The finite cylinder (without caps) is described by equations:

$(q - p_a - (v_a, q - p_a)v_a)^2 - r^2 = 0$ and $(v_a, q - p_1) > 0$ and $(v_a, q - p_2) < 0$

The equations for caps are:

$(v_a, q - p_1) = 0, (q - p_1)^2 < r^2$ bottom cap

$(v_a, q - p_2) = 0, (q - p_2)^2 < r^2$ top cap



Torus

The easiest way I've found to deal with a torus is by converting the ray into Cartesian form and find the intersection using the general formula of the torus:

$$\left(x^2 + y^2 + z^2 + R^2 - r^2\right)^2 = 4R^2\left(x^2 + y^2\right).$$

If there is no solution then they do not intersect.

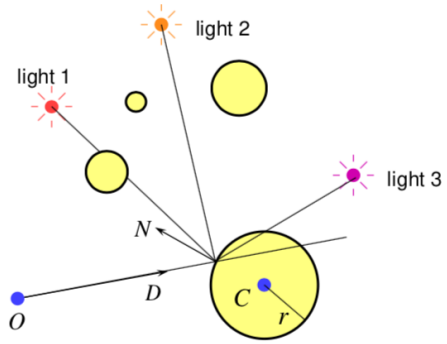
Shading

Once you have the intersection of a ray with the nearest object you can:

- Calculate the normal to the object at the intersection point.
- Shoot rays from that point to all the light sources and calculate the diffuse and specular reflections off the object at that point. **This (plus ambient illumination) gives you the colour of the object (at that point),**

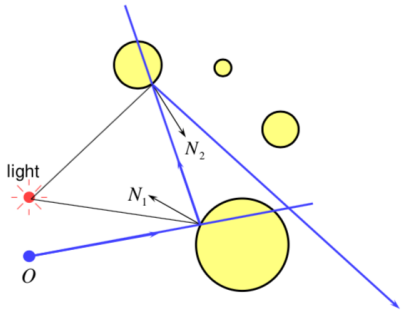
Shadows

When going for the raw from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow. You also need to watch for self-shadowing.



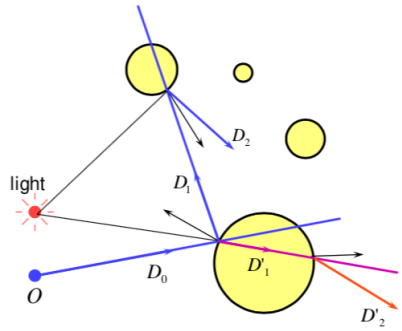
Reflection

If a surface is totally or partially reflective then new rays can be spawned to find the contribution to the pixel's colour given by the reflection.



Transparency and Refraction

Objects can be totally or partially transparent – allowing objects behind the current one to be seen through it. The transparent objects can have refractive indices (bending the rays as they pass through the object). This means the ray can be split into two parts.



Illumination and Shading

Durer's method allows us to calculate what part of the scene is visible in any pixel (using a string to go from the eye to the object).

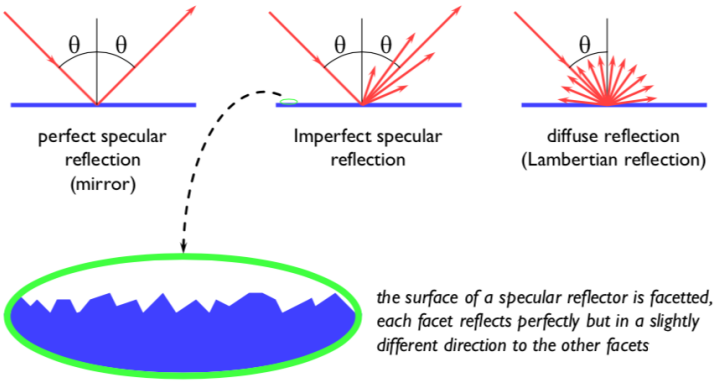
The colour of the pixel depends on:

- Lighting
- Shadows

- Properties of Surface Material

Reflection of Light

Light can be considered to reflect in a few ways – through perfect specular reflection, imperfect specular reflection (still concentrated about perfect reflection) and diffuse reflection (where the light is completely even going everywhere).



Also, the surface can absorb some wavelengths of light, for different types of reflection.

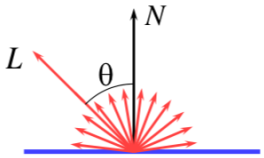
Plastics are good examples of surfaces with **specular reflation in the light's colour** and **diffuse reflection in the plastic's colour.**

In calculating the shading of a surface, we make a few assumptions:

1. There is diffuse reflection **and approximate specular reflection**
2. All light falling on a surface comes directly from a light source
    a. There is no interaction between objects
    b. **However, we can cheat and say that all light reflected off all other surfaces onto a given surface can be amalgamated into a single specular term: ambient illumination and add this to the diffuse and specular reflection.**
3. No objects cast shadows on any other
    a. So, we can treat each surface as if it were the only object in the scene.
4. Light sources are considered to be infinitely distance from the object (lights at infinity)
    a. So, the vector to the light is the same across the whole surface.

We can observe that the colour of a flat surface will be uniform across it, dependent on the colour and position of the object and the colour and position of the light source.

Diffuse Shading



$$I = I_l k_d \cos\theta$$
$$= I_l k_d (N \cdot L)$$

L is a normalised vector pointing in the direction of the light source

N is the normal to the surface

I_l is the intensity of the light source

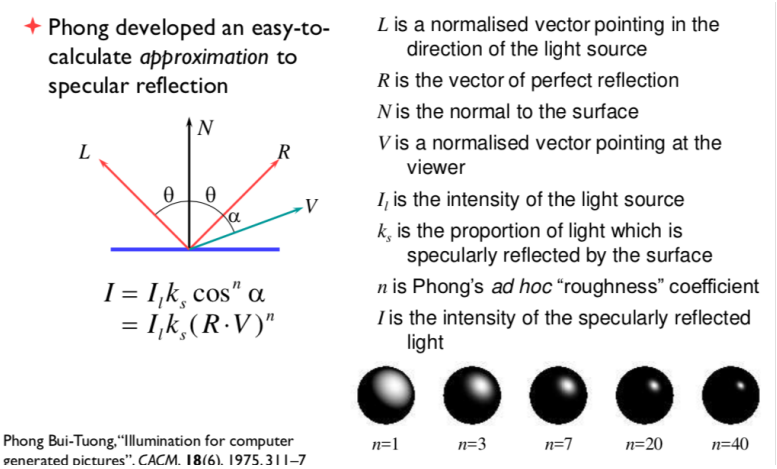
k_d is the proportion of light which is diffusely reflected by the surface

I is the intensity of the light reflected by the surface

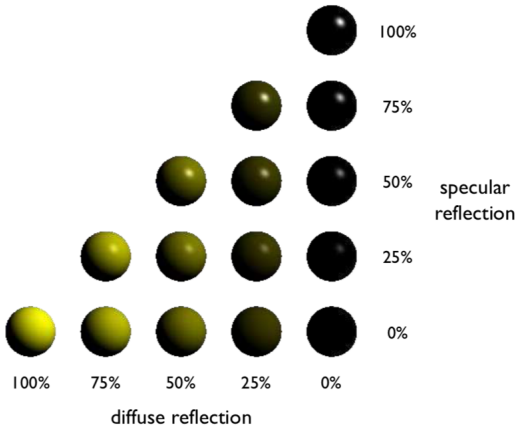
- We can have different Ii and kd for different wavelengths (colours)

- If cos(theta) is less than 0 the light is behind the polygon – so does illuminate this side of the polygon.
- One-sided vs two-sided surfaces
    - **One sided:** Only the side in the direction of the normal vector can be illuminated – therefore if cos(theta) < 0 then both sides are black
    - **Two sided:** The sign of cos(theta) defines which side of the polygon is illuminated.

Specular Reflection



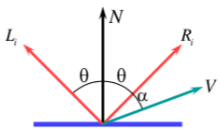
Examples



Shading: Overall reflection

The overall shading is the ambient illumination plus the diffuse and specular reflections from each light source.

$$I = I_a k_a + \sum_i I_i k_d (L_i \cdot N) + \sum_i I_i k_s (R_i \cdot V)^n$$



Sampling

So far, we have assumed that each ray passes through the centre of the pixel, therefore the value of the pixel is the colour of the object which lies under the centre of the pixel.

This leads to:

1. Stair step (jagged) edges to objects
2. Small objects being missed entirely
3. Thin objects being missed or split into small pieces.

Anti-aliasing

These artefacts (and similar ones) are known as aliasing – the methods of removing the effects of aliasing are known as anti-aliasing

In signal processing, aliasing is a precisely defined technical term for a particular kind of artefact, however in computer graphics, its meaning has expanded to include most undesirable effects that can occur in the image. **This is because some of the techniques which remove some of the effects of aliasing will remove others!**

Sampling in Ray Tracing

1. Single Point
    a. Shoots a single ray through the pixel's centre
2. Super-sampling (for anti-aliasing)
    a. Shoot multiple rays through the pixel and average the result
    b. The multiple rays are done through a number of different ways
        1. **Regular Grid**
            a. Divide the pixel into a number of sub-pixels and shoot a ray through each of their centres
            b. However, this can still lead to noticeable aliasing unless a very high resolution sub-pixel grid is used.
        2. **Random**
            a. Shoot pixels at random points in the pixel
            b. Replaces aliasing artefacts with noise artefacts **(however, the eye is far less sensitive to noise than to aliasing)**
        3. **Poisson Disc**
            a. Shoot random points with the proviso that no two rays through the pixel are closer than a set value from each other.
            b. This produces a better-looking image than pure random
            c. However, it is very hard to implement properly.
        4. **Jittered**
            a. Divide pixel into a number of sub-pixels and shoot one ray at a random point in each subpixel – this is an approximation to Poisson disc sampling
            b. **Better than random sampling and easy to implement.**
3. Adaptive super-sampling
    a. Shoot a few rays through the pixel, check the variance of the resulting values
    b. If they are similar (variance is low) – then stop, otherwise shoot some more rays.

Why take multiple samples per pixel: super-sampling is only one reason why we might take multiple sampler per pixel. You can get many effects by distributing the multiple samples over some range – **distributed ray tracing.** This works in one of a couple of ways:

1. Each of the multiple rays shot through a pixel is allocated a random value from the relevant distribution – all effects can be achieved this way with sufficient rays per pixel
2. Or each ray spawns multiple rays when it hits an object

Uses of Distributed Ray Tracing:

1. Distribute the samples for a pixel over the pixel area – used for antialiasing (as described)
2. Distribute the rays going to a light source over some area – allows area light sources in point and directional sources
    a. Produces softer shadows with penumbrae
3. Distribute the camera position over some area – allows the simulation of a camera with a finite aperture lens
4. Distribute the samples in time
    a. Produces motion blur effects on any moving objects

Graphics Pipeline

Since Ray Tracing is computationally so very expensive (only used by hobbyists and for super-high visual quality), we need a faster method.

Therefore, we can:

1. Model surfaces as polyhedra – meshes of polygons
2. Use composition to build scenes
3. Apply perspective transformation and project into the plane of the screen
4. Work out which surface is closest and fill the pixels with the colour of the nearest visible polygon

Since we normally do this, most modern graphics cards have hardware to support this.

Three-Dimensional Objects

Polyhedral surfaces are made up from meshes of multiple connected polygons. **Polygonal meshes are** open or closed and **are manifold or non-manifold.** A mesh is manifold if each edge is incident to only one or two faces (two points at the end of the edge is shared – no points along the edge).

In order to represent a curved surface, we must convert the entire surface into polygons which we can represent in memory. This means approximating the curve.

Triangles

We generally consider triangles since they must be planar, whereas any polygons which are larger than 4 are not necessarily planar and therefore the rotations and other transformations become both ambiguous and may not necessarily do what is predicted. Additionally, the computer hardware (GPUs) is geared towards working with triangles (for this reason) and therefore it is much faster to consider triangles.

Splitting polygons into triangles: This means we generally split all polygons with more than three vertices into triangles – however, it is to be noted that there are often a number of ways that we can split them – we should often consider which is better.

2D Transformations

- 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
  - along vector (x_o, y_o)

$$x' = x + x_o$$
$$y' = y + y_o$$

- shear
  - parallel to x axis
  - by factor a

$$x' = x + ay$$
$$y' = y$$

- scale
  - about origin, factor m

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- rotate
  - about origin, angle θ

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- do nothing
  - identity

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- shear
  - parallel to x axis, factor a

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

However, translations cannot be represented using a simple 2D matrix multiplication, so we switch to **homogeneous 2D co-ordinates**, where an infinite number of homogeneous co-ordinates map to every 2D point – and w=0 represents a point at infinity.

$$(x, y, w) \equiv \left(\frac{x}{w}, \frac{y}{w}\right)$$

We usually take the inverse transform (normal to homogeneous) to be (x, y) = (x, y, 1)

Matrices in homogeneous co-ordinates

- **scale**
  - **about origin, factor** m

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- **rotate**
  - **about origin, angle** θ

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- **do nothing**
  - **identity**

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- **shear**
  - **parallel to** x **axis, factor** a

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

In homogeneous coordinates

$$x' = x + wx_o \qquad y' = y + wy_o \qquad w' = w$$

Concatenating Transformations

We can concatenate transformations by multiplying their matrices (in reverse order to the order in which they are to be performed).

This lets us do things like scale / rotate about an arbitrary point by:

1. Translating the point to the origin
2. Scale / rotate
3. Translate back

3D Transformations

Similarly to 2D transformations:

3D homogeneous co-ordinates

$$(x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

3D transformation matrices

translation

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

identity

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

rotation about *x*-axis

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

scale

$$\begin{bmatrix} m_x & 0 & 0 & 0 \\ 0 & m_y & 0 & 0 \\ 0 & 0 & m_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about z-axis

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

rotation about *y*-axis

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

Model Transformation 1:

We always scale first, then rotate and then translate.

Rotation: Rotation is a multi-step process. We break rotation into steps each of which is a rotation about a principal axis. We work these out by taking the desired orientation back to the original axis-aligned position.

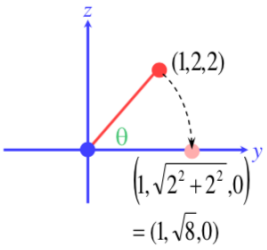
EXAMPLE:

- desired axis: (2,4,5)–(1,2,3) = (1,2,2)
- original axis: y-axis = (0,1,0)

- zero the z-coordinate by rotating about the x-axis

$$\mathbf{R}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

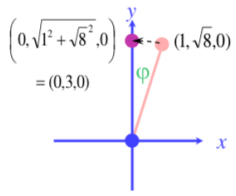
$$\theta = -\arcsin\frac{2}{\sqrt{2^2+2^2}}$$



- then zero the x-coordinate by rotating about the z-axis
- we now have the object's axis pointing along the y-axis

$$\mathbf{R}_2 = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\varphi = \arcsin\frac{1}{\sqrt{1^2+\sqrt{8}^2}}$$



- the overall transformation is:
  - first scale
  - then take the inverse of the rotation we just calculated
  - finally translate to the correct position

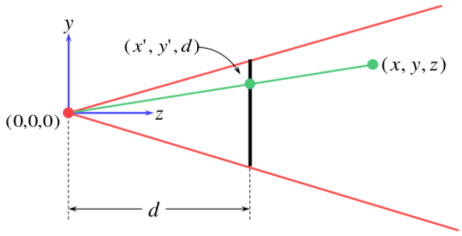
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1^{-1} \times \mathbf{R}_2^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

3D => 2D Projection

In order to make a picture, we project a 3D world to a 2D image, by projecting the three-dimensional world being projected onto a plane.

There are a couple of types of projection, using:

1. Parallel
   a. (x, y, z) -> (x, y)
   b. This is used in CAD, architecture, etc (lengths are preserved)
   c. However, it looks unrealistic.
2. Perspective
   a. (x, y, z) -> (x/z, y/z)
   b. Things get smaller as they get further away
   c. Things look realistic – this is how cameras work.



$$x' = x\frac{d}{z}$$

$$y' = y\frac{d}{z}$$

Projection as a matrix operation

$$\begin{bmatrix} x \\ y \\ 1/d \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = x\frac{d}{z}$$

$$y' = y\frac{d}{z}$$

remember $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$

This is useful in the z-buffer algorithm where we need to interpolate $1/z$ values rather than z values.

$$z' = \frac{1}{z}$$

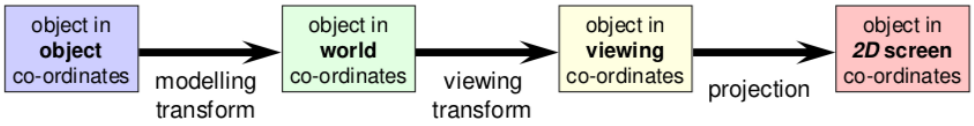
Perspective Projection with an Arbitrary Camera

Assumptions are:

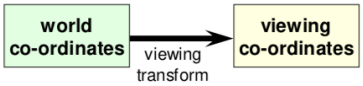
1. Screen centre is at (0, 0, d)
2. The screen is parallel to the xy-plane
3. The z-axis is into the screen
4. The y-axis is up and the x-axis to the right
5. The camera is at the origin

We can either work out a equations for projecting objects about an arbitrary point onto an arbitrary point **or transform all objects into our standard coordinate system (viewing coordinates) and use the above assumptions.**

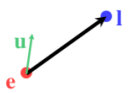
MVP Transformations



Viewing Transformation



- the problem:
  - to transform an arbitrary co-ordinate system to the default viewing co-ordinate system
- camera specification in world co-ordinates
  - eye (camera) at (e_x, e_y, e_z)
  - look point (centre of screen) at (l_x, l_y, l_z)
  - up along vector (u_x, u_y, u_z)
    - perpendicular to **el**



- First, we translate the eye-point to the origin and scale so that eye point to look point vector is the distance from the origin to screen centre d.
- Then we rotate so the line el is aligned with the z-axis.

◆ translate eye point, (e_x,e_y,e_z), to origin, $(0,0,0)$

$$\mathbf{T}=\begin{bmatrix}1&0&0&-e_x\\0&1&0&-e_y\\0&0&1&-e_z\\0&0&0&1\end{bmatrix}$$

◆ scale so that eye point to look point distance, $|\overline{\mathbf{el}}|$, is distance from origin to screen centre, d

$$|\overline{\mathbf{el}}|=\sqrt{(l_x-e_x)^2+(l_y-e_y)^2+(l_z-e_z)^2} \qquad \mathbf{S}=\begin{bmatrix}d/|\overline{\mathbf{el}}|&0&0&0\\0&d/|\overline{\mathbf{el}}|&0&0\\0&0&d/|\overline{\mathbf{el}}|&0\\0&0&0&1\end{bmatrix}$$

◆ need to align line $\overline{\mathbf{el}}$ with z-axis

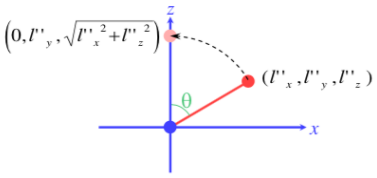
- first transform e and l into new co-ordinate system

$$\mathbf{e}''=\mathbf{S}\times\mathbf{T}\times\mathbf{e}=\mathbf{0} \qquad \mathbf{l}''=\mathbf{S}\times\mathbf{T}\times\mathbf{l}$$

- then rotate e''l'' into yz-plane, rotating about y-axis

$$\mathbf{R}_1=\begin{bmatrix}\cos\theta&0&-\sin\theta&0\\0&1&0&0\\\sin\theta&0&\cos\theta&0\\0&0&0&1\end{bmatrix}$$

$$\theta=\arccos\frac{l''_z}{\sqrt{l''^{\,2}_x+l''^{\,2}_z}}$$

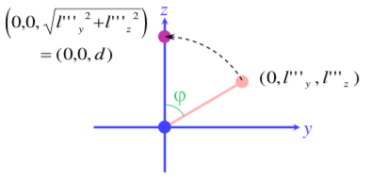


◆ having rotated the viewing vector onto the yz plane, rotate it about the x-axis so that it aligns with the z-axis

$$\mathbf{l}'''=\mathbf{R}_1\times\mathbf{l}''$$

$$\mathbf{R}_2=\begin{bmatrix}1&0&0&0\\0&\cos\varphi&-\sin\varphi&0\\0&\sin\varphi&\cos\varphi&0\\0&0&0&1\end{bmatrix}$$

$$\varphi=\arccos\frac{l'''_z}{\sqrt{l'''^{\,2}_y+l'''^{\,2}_z}}$$



◆ the final step is to ensure that the up vector actually points up, i.e. along the positive y-axis

- actually need to rotate the up vector about the z-axis so that it lies in the positive y half of the yz plane

$$\mathbf{u}''''=\mathbf{R}_2\times\mathbf{R}_1\times\mathbf{u}$$

$$\mathbf{R}_3=\begin{bmatrix}\cos\psi&-\sin\psi&0&0\\\sin\psi&\cos\psi&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$$\psi=\arccos\frac{u''''_y}{\sqrt{u''''^{\,2}_x+u''''^{\,2}_y}}$$

Therefore,

- we can now transform any point in world co-ordinates to the equivalent point in viewing co-ordinate

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- in particular: $\mathbf{e} \rightarrow (0,0,0) \quad \mathbf{l} \rightarrow (0,0,d)$
- the matrices depend only on e, l, and u, so they can be pre-multiplied together

$$\mathbf{M} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T}$$

Illumination and Shading

If we draw polygons with uniform colours around them, we generally get very poor results therefore we must interpolate the colours across the polygons.

In order to interpolate the colours across the polygons, we need:

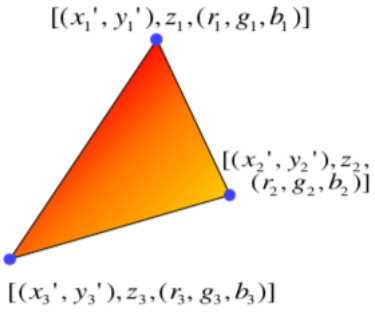
1. A colour at each vertex
2. An algorithm to blend between the colours across the vertex
3. **Normals at each point**
    a. Specular reflection requires this – ambient and diffuse can just work off the first two.

Gouraud Shading

We calculate the diffuse illumination at each vertex by:

- Calculating normal at the vertex and use this to calculate the diffuse illumination at this point
- Normal can be calculated directly.

Then we interpolate the colour between the vertices of the polygon. This means the surface will look smoothly curved, rather than looking like a set of polygons – though the outline will still look polygonal.

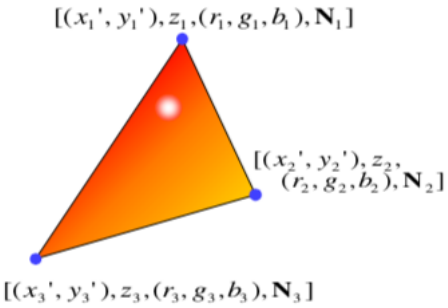


While much better than flat shading it still has the flaw of missing specular highlights – therefore we must think even more carefully.

Phong Shading
The Phong shading is similar to Gouraud shading but calculates the specular components in addition to the diffuse component.

Therefore, we interpolate the normal across the polygon in order to calculate the reflection vector. Then for each point, we calculate the specular reflection and add in the diffuse illumination.

N.B. Phong's approximation to specular reflection ignores (amongst other things), the effects of glancing incidence.



Graphics Hardware and OpenGL

Graphics Processing Unit (GPU)

A GPU is like a CPU (Central Processing Unit) for processing graphics. It is optimised for floating point operations on large arrays of data – with vertices, normal, pixels, etc.

It performs all low-level tasks and a lot of high-level tasks:

- Clipping, rasterization, hidden surface removal – essentially draws triangles very efficiently
- Procedural Shading, texturing, animation, simulation etc
- Video rendering, de and encoding, de-interfacing
- Physics engines

It generally offers full programmability at several pipeline stages – but it is optimised for massively parallel operations.

Why is a GPU fast?

- 3D rendering can be efficiently parallelized
    - Allow us to complete millions of pixels, millions of triangles, all of which have the operations which can be computed independently at the same time.
- Hence, modern GPUs contain lots of SIMD (Single Instruction Multiple Data) which operates on large arrays of data
- It has >> 400 GB/s which has a much higher bandwidth than CPU – but the peak performance can be expected for specific operations

GPU APIs

OpenGL

- Multiplatform
- Open standard API
- Has a focus on general 3D applications – the OpenGL driver manages the resources
- **OpenGL ES**
    - OpenGL ES is a stripped version of OpenGL – removes the functionality that is not necessary on mobile devices
    - It is made for a whole range of devices including:
        - iOS
        - Android
        - Playstation 3
        - Nintendo 3DS
- **OpenGL in Java**
    - The Standard Java API does not include OpenGL interface
    - But several wrapper libraries exist including
        - Java OpenGL (JOGL)
        - Lightweight Java Game Library – LWJGL
            - We use this (version 3)
            - It is better maintained
            - It has access to other APIs (OpenCL, OpenAL, …)
    - We also need to use a linear algebra library
        - JOML (Java OpenGL Math Library)
        - It operates on 2, 3, 4-dimensional vectors and matrices
- **OpenGL History**
    - It originated as a proprietary library IRIS GL made by SGI
    - OpenGL 1.0 (1992)
    - OpenGL 1.2 (1998)
    - OpenGL 2.0 (2004)
        - GLSL
        - Non-power-of-two textures (NPOT)
    - OpenGL 3.0 (2008)
        - It was a major overhaul of the API
        - Many features from previous versions were deprecated.
    - OpenGL 3.2 (2009)
        - Core and Compatability profiles
        - Geometry Shaders
    - OpenGL 4.0 (2010)
        - Catching up with Direct3D 11
    - OpenGL 4.5 (2014)
    - OpenGL 4.6 (2017)

DirectX

- Microsoft Windows / Xbox
- Proprietary API
- It has a focus on games – the application manages the resources.

Vulkan

- It is cross platform and an open standard

- It is a very low-overhead API for high performance 3D graphics compared to OpenGL and DirectX
    - It reduces the CPU load and works better for multi-CPU-core architectures
    - It also gives much finer control of the GPU
- But
    - The code for drawing a few primitives can take 1000s of lines of code
    - It is intended for game engines and code that must be very well optimised.

CUDA

- OpenGL and DirectX are not meant to be used for general purpose computing – physical simulation, machine learning
- CUDA is NVidia's architecture for parallel computing
    - C-like programming language
    - With special API for parallel instructions
    - Requires NVidia GPU

OpenCL

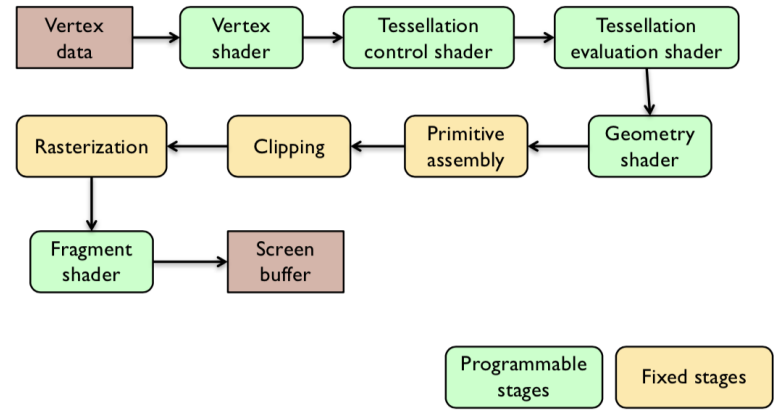
- Similar to CUDA but it is an open standard
- It can run on both the GPU and the CPU
- It is supported by AMD, Intel and NVidia, Qualcomm, Apple …
- Very widely used therefore,

OpenGL Rendering Pipeline

Model

1. CPU Code
    - gl* functions that
        - Create OpenGL objects
        - Copy data between the CPU and the GPU
        - Modify the OpenGL state
        - Enqueue operations
        - Synchronise CPU and GPU
    - C99 Library
    - Wrappers in most programming languages

2. GPU Code
    - Fragment Shaders
    - Vertex Shaders
    - Other shaders
    - It is written in GLSL
        - Similar to C
        - From OpenGL 4.6, they could be written in other languages and compiler to SPIR-V



Vertex Shader
Processes vertices, normal and (u, v) texture coordinates.

Tessellation Control Shader & Tessellation Evaluation Shader
Create new primitives by tessellating existing primitives (patches).

Geometry Shader
This is an optional step – it operates on tessellated geometry. It can create new primitives.

Primitive Assembly
It organises vertices into primitives and prepares them for rendering.

Clipping
It removes vertices so they all lie within the viewport (view frustrum). This means that new primitives might need to be created.

Rasterization
It generates fragments (pixels) to be drawn for each primitive.
It interpolates vertex attributes.

Fragment Shader
Computes the colour for each fragment (pixel). It can look up the colour in the texture and can modify the pixels' depth value.

Working with Buffers

Generating Names

- "name" is like a reference in Java
- glGen* functions create names WITHOUT allocating the actual object.
- From OpenGL 4.5 glCreate* functions create names and allocate the actual object.

Binding Objects

- glBind* functions
- Performs two operations
    - Allocates memory for the particular object (if it does not exist)
    - Makes it active in the current OpenGL context

- Functions operating on OpenGL objects will change the currently bound (or active) object.

Unbinding Objects

- Passing "0" instead of "name" unbinds the active object.
    - glBind(..., 0)

Deleting Objects

- glDelete* functions delete both the objects and its name.

When dealing with buffers, you must be aware that OpenGL is not an Object-Oriented API so we must do things like:

```
int va = glGenVertexArrays();
glBindVertexArray(va); // va
becomes "active" VertexArray

int vertices = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFE
R, vertices); // This adds vertices
to currently bound VertexArray
```

Example OpenGL Code

In order to draw some triangles, we must:

1. Initialise OpenGL
    a. Initialising rendering windows and OpenGL context
    b. Send the geometry (vertices, triangles, normals) to the GPU
    c. Load and compile Shaders
2. Set up inputs
3. Draw a frame
    a. Clear the screen buffer
    b. Set the MVP (model view projection) matrix
    c. Render geometry
    d. Flip the screen buffers
4. Free resources

Initialise

```
int vertexArrayObj = glGenVertexArrays(); // Create a name
glBindVertexArray(vertexArrayObj);          // Bind a VertexArray

float[] vertPositions = new float[] { -1, -1, 0,  0, 1, 0,  1, -1, 0 }; // x, y, z, x, y, z ...
// Java specific code for transforming float[] into an OpenGL-friendly format
FloatBuffer vertex_buffer = BufferUtils.createFloatBuffer(vertPositions.length);
vertex_buffer.put(vertPositions); // Put the vertex array into the CPU buffer
vertex_buffer.flip();                  // "flip" is used to change the buffer from read to
write mode

int vertex_handle = glGenBuffers();    // Get an OGL name for a buffer object
glBindBuffer(GL_ARRAY_BUFFER, vertex_handle); // Bring that buffer object into
existence on GPU
glBufferData(GL_ARRAY_BUFFER, vertex_buffer, GL_STATIC_DRAW); // Load the
GPU buffer object with data
```

Rendering

```
// Step 1: Pass a new model-view-projection matrix to the vertex shader
Matrix4f mvp_matrix; // Model-view-projection matrix
mvp_matrix = new
Matrix4f(camera.getProjectionMatrix()).mul(camera.getViewMatrix());

int mvp_location = glGetUniformLocation(shaders.getHandle(), "mvp_matrix");
FloatBuffer mvp_buffer = BufferUtils.createFloatBuffer(16);
mvp_matrix.get(mvp_buffer);
glUniformMatrix4fv(mvp_location, false, mvp_buffer);

// Step 2: Clear the buffer
glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set the background colour to dark grey
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Step 3: Draw our VertexArray as triangles
glBindVertexArray(vertexArrayObj); // Bind the existing VertexArray object
glDrawElements(GL_TRIANGLES, no_of_triangles, GL_UNSIGNED_INT, 0); //
Draw it as triangles
glBindVertexArray(0); // Remove the binding

// Step 4: Swap the draw and back buffers to display the rendered image
glfwSwapBuffers(window);
glfwPollEvents();
```

GLSL Fundamentals

Shaders

They are small programs executed on a GPU. They are executed for every vertex, each fragment etc.

Data Types

- Basic types
  - float, double, int, uint, bool
- Aggregate types
  - float: vec2, vec3, vec4; mat2, mat3, mat4
  - double: dvec2, dvec3, dvec4; dmat2, dmat3, dmat4
  - int: ivec2, ivec3, ivec4
  - uint: uvec2, uvec3, uvec4
  - bool: bvec2, bvec3, bvec4

```
vec3 V = vec3( 1.0, 2.0, 3.0 );   mat3 M = mat3( 1.0, 2.0, 3.0,
                                                 4.0, 5.0, 6.0,
                                                 7.0, 8.0, 9.0 );
```

Indexing Components

You can index components of aggregate types very easily, in a number of different ways.

- Subscripts: rgba, xyzw, stpq (work exactly the same)
  - float red = color.r;
  - float v_y = velocity.y;

  but also
  - float red = color.x;
  - float v_y = velocity.g;
- With 0-base index:
  - float red = color[0];
  - float m22 = M[1][1]; // second row and column of matrix M

Swizzling

You can also select the elements of the aggregate type:

- vec4 rgba_color(1.0, 1.0, 0.0, 1.0);
- vec3 rgb_color = rgba_color.rgb;
- vec3 bgr_color = rgba_color.bgr;
- vec3 luma = rgba_color.ggg;

Arrays

Arrays work exactly the same as for C:

```
float lut[5] = float[5]( 1.0, 1.42, 1.73, 2.0, 2.23 );
```

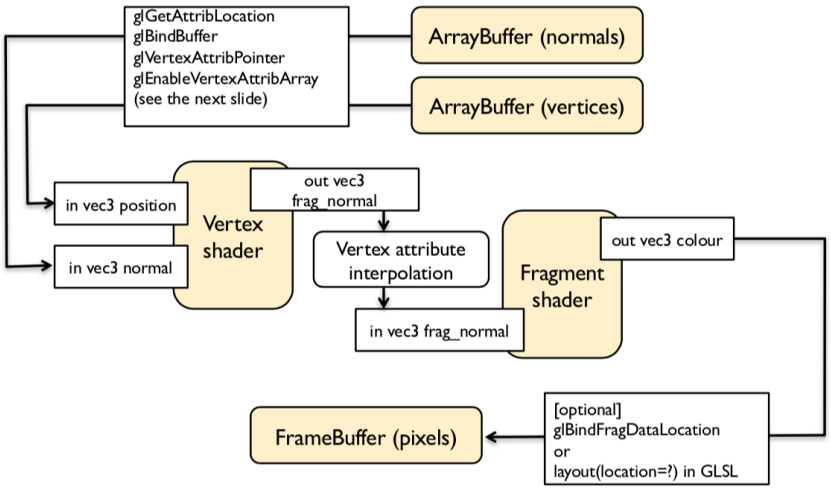
- Size can be checked with "length()"

```
for( int i = 0; i < lut.length(); i++ ) {
  lut[i] *= 2;
}
```

Storage Qualifiers

- `const` – read-only, fixed at compile time
- `in` – input to the shader
- `out` – output from the shader
- `uniform` – parameter passed from the application (Java), constant for the primitive
- `buffer` – shared with the application
- `shared` – shared with local work group (compute shaders only)

Shader Inputs and Outputs



We specify input to a vertex shader by calling a 'function' with the parameters of the variable that the shader should have.

For inputs, we add it to a buffer, enabling it and telling it where and of what form the variables are before enabling the variable:

```
// Get the locations of the "position" vertex attribute variable
in our shader
int position_loc = glGetAttribLocation(shaders_handle,
"position");
// If the vertex attribute found
if (position_loc != -1) {
  // Activate the ArrayBuffer that should be accessed in the
shader
  glBindBuffer(GL_ARRAY_BUFFER, vertex_handle);
  // Specifies where the data for "position" variable can be
accessed
  glVertexAttribPointer(position_loc, 3, GL_FLOAT, false, 0, 0);
  // Enable that vertex attribute variable
  glEnableVertexAttribArray(position_loc);
}
```

For uniforms, we simply define the uniform in the shader, before adding it to the buffer and giving the buffer to glsl.

- In shader:

```
uniform mat4 mvp_matrix; // model-view-projection matrix
```

- In Java:

```
Matrix4f mvp_matrix; // Matrix to be passed to the shader
...
int mvp_location = glGetUniformLocation(shaders.getHandle(),
                   "mvp_matrix");
FloatBuffer mvp_buffer = BufferUtils.createFloatBuffer(16);
mvp_matrix.get(mvp_buffer);
glUniform**Matrix4fv**(mvp_location, false, mvp_buffer);
```

Name of the method depends on the data type. For example, glUniform3fv for Vector3f

Operators

- Arithmetic: + - ++ --
  - Multiplication:
    - vec3 * vec3 – element-wise
    - mat4 * vec4 – matrix multiplication (with a column vector)
- Bitwise (integer): <<, >>, &, |, ^
- Logical (bool): &&, ||, ^^
- Assignment:

```
float a=0;
a += 2.0; // Equivalent to a = a + 2.0
```

Math

- Trigonometric:
  - `radians( deg ), degrees( rad ), sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh`
- Exponential:
  - `pow, exp, log, exp2, log2, sqrt, inversesqrt`
- Common functions:
  - `abs, round, floor, ceil, min, max, clamp, …`
- And many more

Flow Control

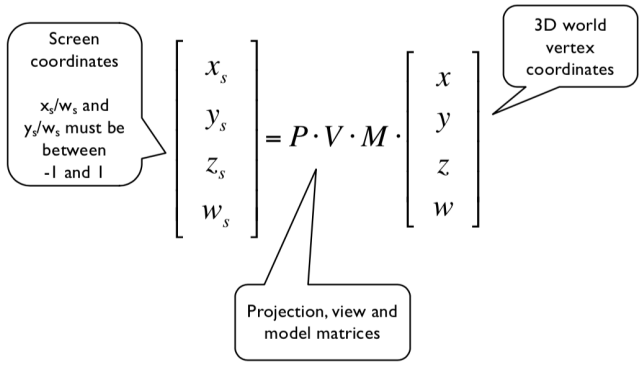
```
if( bool ) {
  // true
} else {
  // false
}

switch( int_value ) {
  case n:
    // statements
    break;
  case m:
    // statements
    break;
  default:
}
for( int i = 0; i<10; i++ ) {
   ...
}

while( n < 10 ) {
 ...
}

do {
 ...
} while ( n < 10 )
```

Transformations and Vertex Shaders

In order to go from **Objects coordinates to World Coordinates**, you need to use the Model matrix to change the object. It could be different for every object.

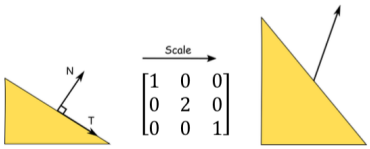
In order to go from **World coordinates to View coordinates (where the camera is at the origin, pointing at –z) you need the view matrix.** This is common for all objects in the scene.

Finally, to go from the **View coordinates to the Screen coordinates (where x and y are in range -1 to 1) and the entire scene is projected to a '2D plane') you use the projection matrix which is defined for every object.** It is important to note that the z coordinate is still maintained in this step since it is required for depth testing (which object is in front).



Transforming Normal Vectors

- Transformation by a nonorthogonal matrix does not preserve angles



- Since:
  $N \cdot T = 0$
  $N' \cdot T' = (GN) \cdot (\mathrm{MT}) = 0$
  Normal transform
  Transformed normal and tangent vector
  Vertex position transform
- We can find that: $G = (M^{-1})^T$
  - Derivation shown on the visualizer

Raster Buffers

Render Buffers

Colour:	GL_FRONT	GL_BACK	Four components: RGBA Typically 8 bits per component
In stereo:	GL_FRONT_LEFT	GL_FRONT_RIGHT	
	GL_BACK_LEFT	GL_BACK_RIGHT	
Depth:	DEPTH		To resolve occlusions (see Z-buffer algorithm) Single component, usually >8 bits
Stencil:	STENCIL		To block rendering selected pixels Single component, usually 8 bits.

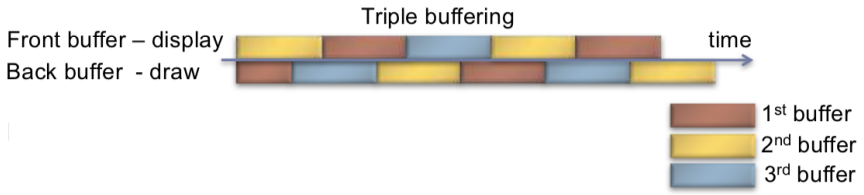
Double Buffering

In order to avoid flicker and tearing, two buffers (rasters) are used:

- The front buffer shows what is shown on the screen
- The back buffer is not shown and the GPU draws into that buffer

Then, when the drawing is finished, you swap the front and back buffers – ie then use the 2nd buffer to display and use the first to draw in.

However, by introducing a **third buffer**, you can eliminate the gaps which reduce the efficiency of the system.



However, while more efficient this has two issues:

1. Requires more memory
2. Leads to a larger delay between drawing a displaying a frame – though this could happen quicker per frame.

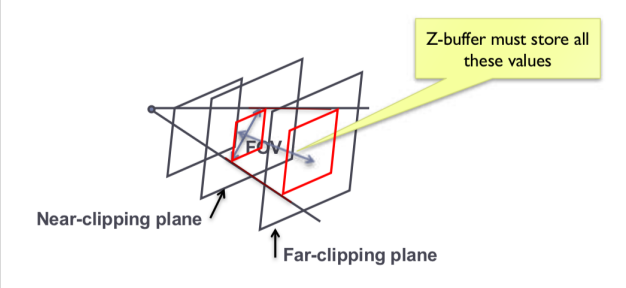
Z-Buffer

1. Initialise the depth buffer and image buffer for all pixels
    a. Color (x, y) = BackgroundColour
    b. Depth (x, y) = z_far // position of far clipping area.
2. For every triangle
    a. Calculate z for current (x, y)
    b. If (z < Depth (x, y))
        i. Depth (x, y) = z
        ii. Color (x, y) = Polygon_Color (x, y)

Clipping

You define the view frustrum by defining the near-clipping plane and the far-clipping plane which defines the objects which are visible.

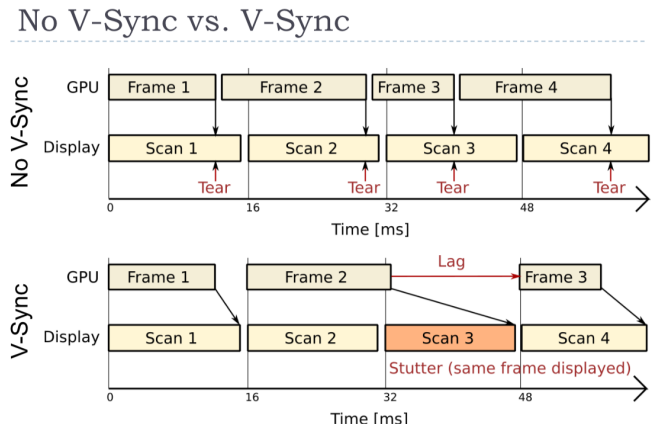
It is a frustrum since you are observing from a set point (camera).



Vertical Synchronisation: V-Sync

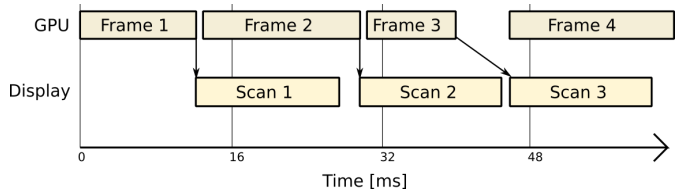
Since pixels are copied from the colour buffer to the monitor row-by-row, you can get **tearing artefacts.** This occurs when the front and back buffer are swapped in the middle of the process of copying – therefore the upper part of the screen contains the previous frame and the lower part contains the current frame.

In OpenGL, you can use the glwfSwapInterval(1) command to wait until the last raw is copied to the display before swapping the buffers.



Note, here, the scan happens a set number of times per second.

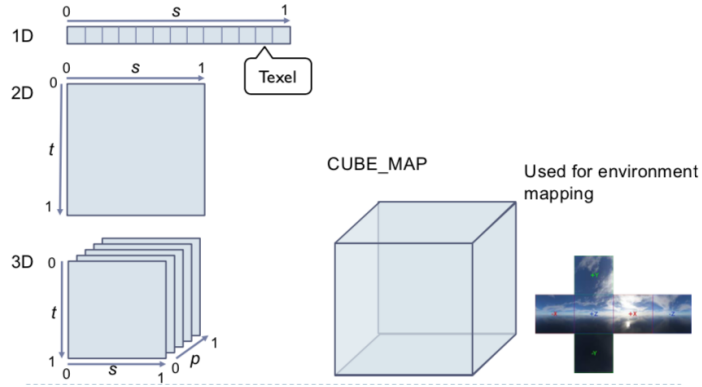
FreeSync (AMD) and G-Sync (Nvidia)
This makes use of adaptive sync, where the graphics card controls the timing of frames on the display. This can save power where useful and also reduce lag for real-time graphics.



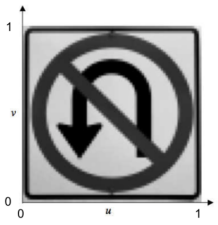
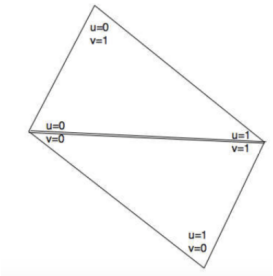
Textures

OpenGL Texture Types

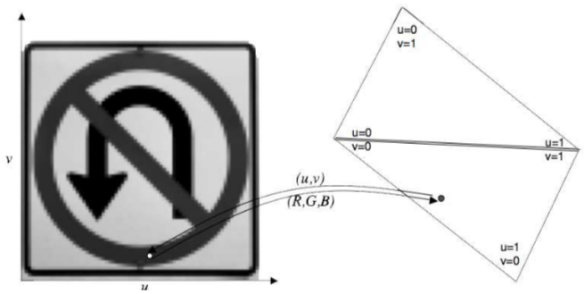
1. In **1D**, it is effectively an array of texels
    a. Therefore, textures can have any size but the sizes that are powers of 2 (2^n) may give better performance.
2. In **2D,** it is an array of arrays of texels.
3. In **3D,** it is an array of 2D textures.
4. In a **Cube Map,** we represent a texture of a surface of a cube using the map of it.



Texture Mapping

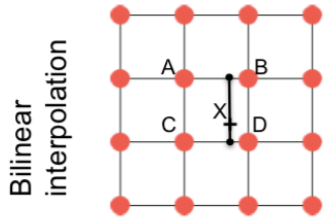
1. Define a texture mapping function (image) – T(u, v) where (u, v) is the texture coordinates.
    a. 
2. Define the correspondence between the vertices on the 3D object and the texture coordinates.
    a. 

3. For every surface point compute texture coordinates – using the texture function to get the texture value. This gets used as a colour or reflectance.

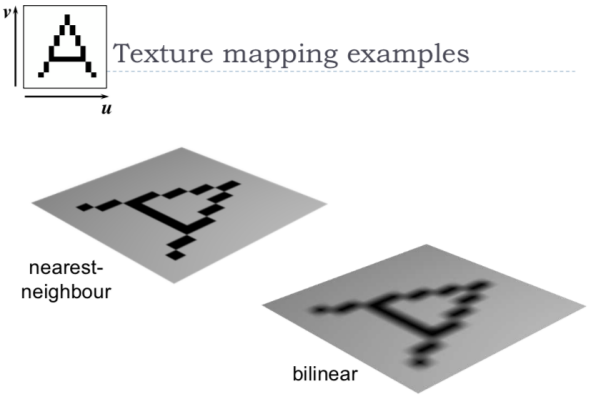
a. 

Sampling

Nearest neighbour sampling: Pick the texture of the nearest texel.
Bilinear Interpolation: Interpolate to find the texture of the point – first interpolating for the x-axis and then interpolating for the y-axis.



Interpolate first along x-axis between AB and CD, then along y-axis between the interpolated points.

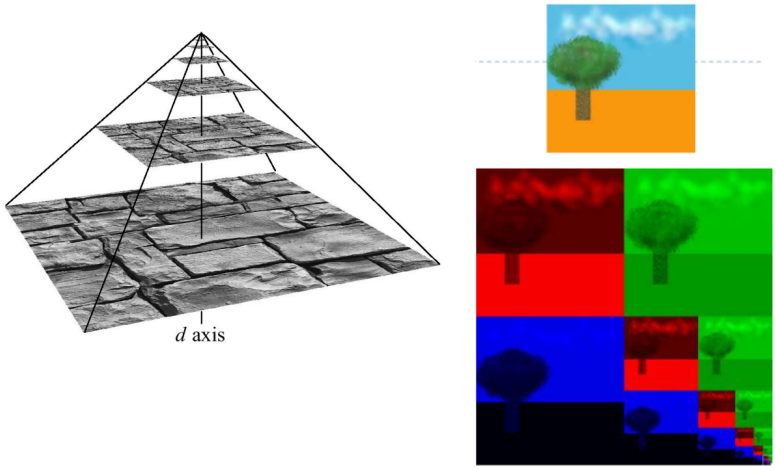


1. **Up-sampling**
    a. More pixels than texels
    b. Values need to be interpolated.
    c. **You will always get visible artefacts – the only way to prevent this is to ensure that the texture map is of a sufficiently high resolution so it doesn't happen.**
    d. **Nearest-Neighbour**
        i. You get blocky artefacts
    e. **Bilinear**

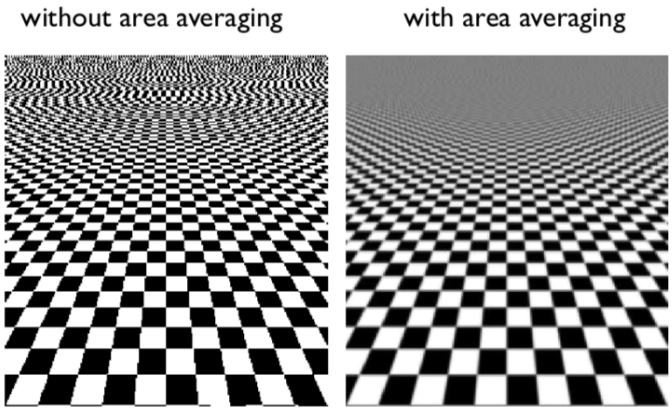
   i. You get blurry artefacts
2. **Down-sampling**
   a. There are fewer pixels than texels
   b. The values need to be averaged over an area of the texture
      i. Using a **mipmap**
      ii. You need to average the texture across that area, not just take a sample in the middle of the area.

Mipmap

Textures can be stored at multiple resolutions as a mipmap – each level of the pyramid is half the size of the lower level. It provides pre-filtered texture (area-averaged) when screen pixels are larger than the full resolution texels. The mipmap requires a third of the original texture size to store.



In order to use the mipmap, we find the level that it has a 1-1 mapping and combine the three items and recombine them to form a single texture map.



Texture Tiling: Repetitive patterns can be represented as texture tiles. It is made so that the texture folds over.

Texture Atlas: A single texture can be used and referenced for multiple surfaces and objects.

Bump Mapping

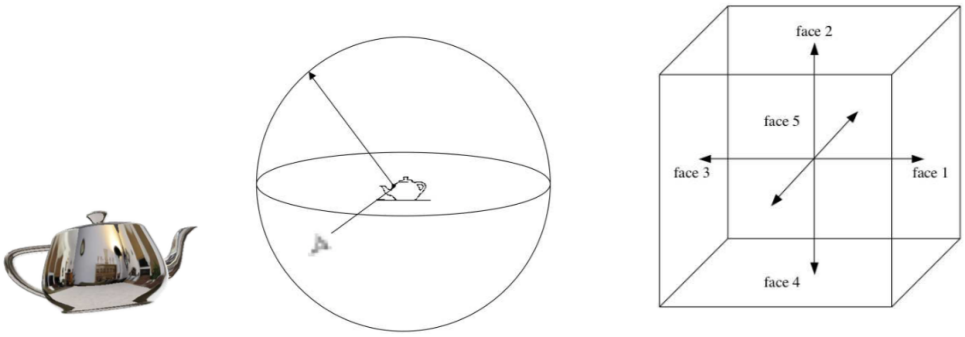
Bump mapping is a special kind of texture that modifies the surface normal (a vector that is perpendicular to a surface). The surface is still flat but shading appears as on an uneven surface. This is easily done **in fragment shaders.**

Displacement Mapping

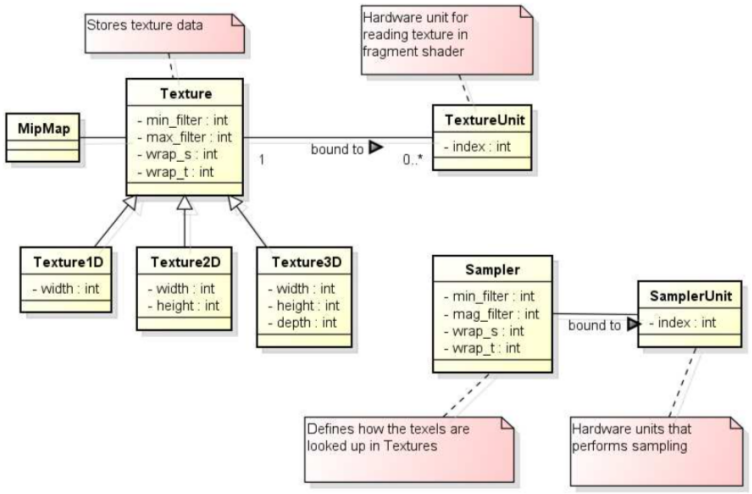
Displacement Mapping is a texture that modifies the surface. It has better results than bump mapping since the surface is not flat. This requires **geometry shaders.**

Environment Mapping

We use environment mapping to show the environment reflected by an object. In order to do this, we use an environment cube – with each face capturing the environment in that direction and therefore we can recombine that to show something.



Texture Objects in OpenGL



Setting up a texture

```
// Create a new texture object in memory and bind it
int texId = glGenTextures();
glActiveTexture(textureUnit);
glBindTexture(GL_TEXTURE_2D, texId);

// All RGB bytes are aligned to each other and each component is
1 byte
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Upload the texture data and generate mipmaps
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tWidth, tHeight, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, buf);
glGenerateMipmap(GL_TEXTURE_2D);
```

Texture parameters

```
//Setup filtering, i.e. how OpenGL will interpolate the pixels
when scaling up or down
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
```

How to interpolate in 2D

How to interpolate between mipmap levels

```
//Setup wrap mode, i.e. how OpenGL will handle pixels outside of
the expected range
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
```

Fragment shader

```
#version 330
uniform sampler2D texture_diffuse;
in vec2 frag_TextureCoord;

out vec4 out_Color;

void main(void) {
	out_Color = texture(texture_diffuse, frag_TextureCoord);
}
```

Rendering

```
// Bind the texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indicesCount, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

glBindTexture(GL_TEXTURE_2D, 0);
```

Frame Buffer Objects

Instead of rendering to the screen buffer (back buffer – GL_BACK), an image can be rendered to an off-screen buffer: this is a Texture or RenderBuffer. It is faster to render to a RenderBuffer than a Texture but it cannot be copied, pixels can only be copied.

They can be used for:

1. Post-processing, tone-mapping, blooming, etc.
2. Reflections (in water), animated textures.
3. When the result of rendering is not shown on the screen – it is saved to disk.

Using a FBO

1. Create a FBO object
2. Attach a Texture (colour) and a RenderBuffer (depth)

```
int color_tex = glGenTextures();
glBindTexture(GL_TEXTURE_2D, color_tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 256, 256, 0, GL_BGRA,
GL_UNSIGNED_BYTE, NULL);

int myFBO = glGenFramebuffers();
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
//Attach 2D texture to this FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, color_tex, 0);
int depth_rb = glGenRenderbuffers();
glBindRenderbuffer(GL_RENDERBUFFER, depth_rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24,
256, 256);
//Attach depth buffer to FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depth_rb);
```

- Render

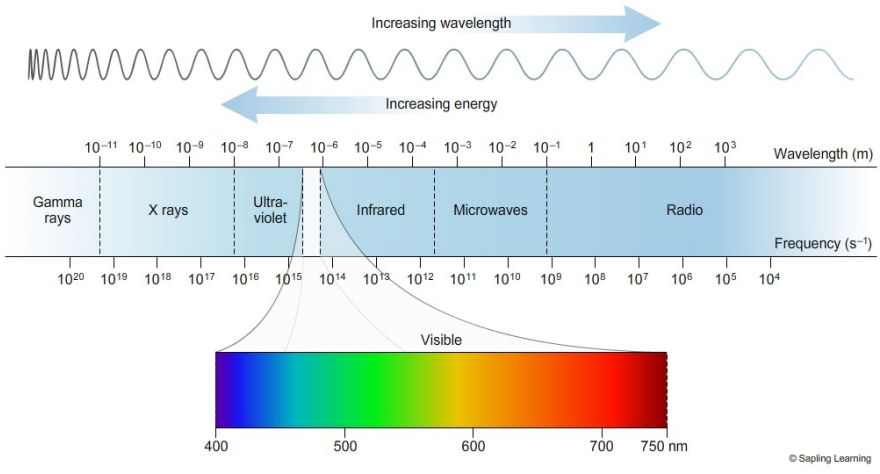
```
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Render

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Colour

Definitions

Electromagnetic Spectrum

Visible Light is EM waves of wavelength in the range of 380nm to 730nm. The earth's atmosphere lets a lot of light in this wavelength band through. It is higher in energy than thermal infrared so heat does not interfere with vision.



Colour

Colour is the result of our perception – there is no physical definition of colour.
However, for emissive displays / objects:

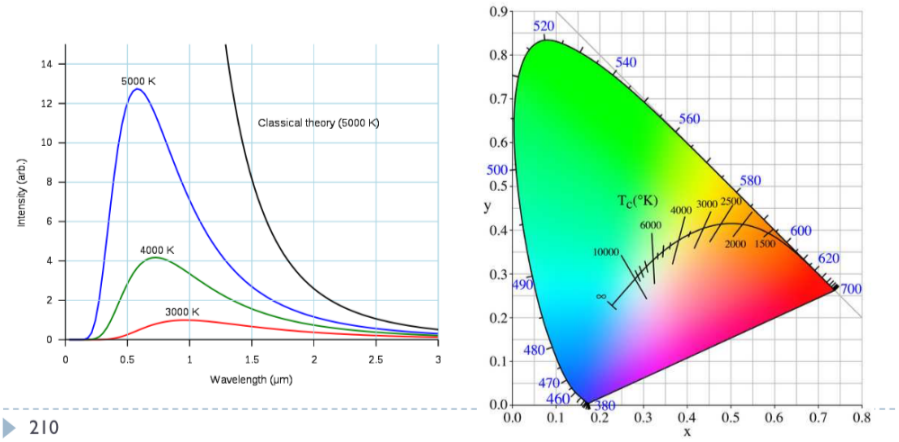
$$colour = perception\ (spectral\ emission)$$

For reflective displays / objects:

$$colour = perception\ (illumination * reflectance)$$

Black body radiation

Black body radiation is radiation emitted by a perfect absorber at a given temperature – Graphite is a good approximation of a black body.



210

It is related by:

$$W = \lambda_{max} T$$

where $lambda_{max}$ is the wavelength of maximum intensity.

Correlated colour temperature

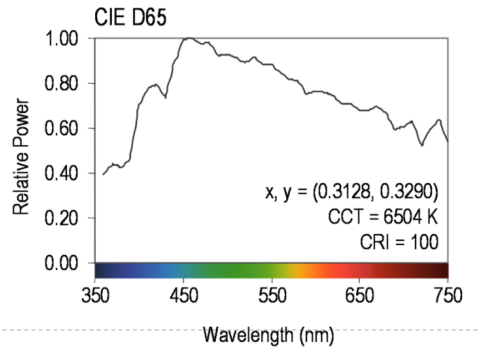
The correlated colour temperature is the temperature of a black body radiator that produces light most closely matching the particular source:

- Typical north-sky light – 7500K
- Typical average daylight – 6500K
- Domestic tungsten lamp (100 – 200W) – 2800K
- Domestic tungsten lamp (40 – 60W) – 2700K
- Sunlight at sunset – 2000K

It is used to describe the colour of the illumination – **illumination is the source of light.**

Standard illuminant D65

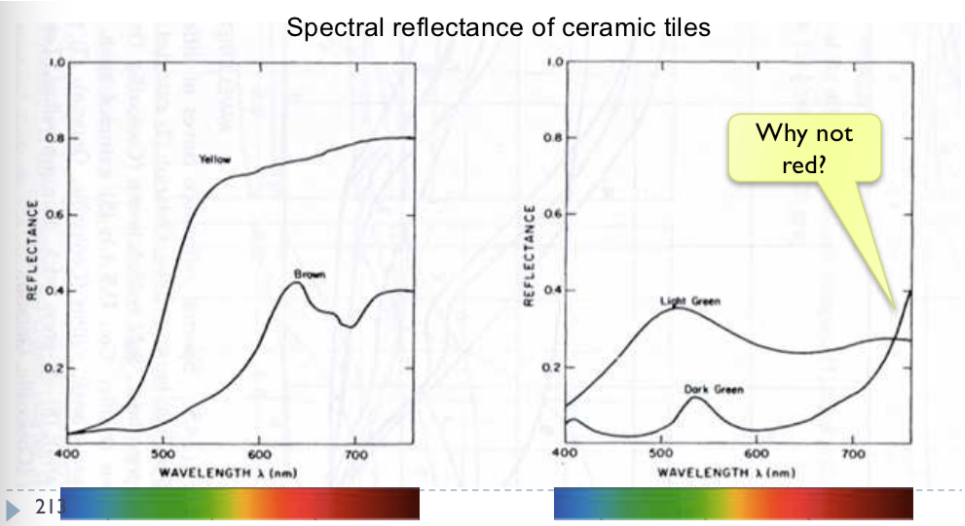
This is a standardises illuminant of midday sun in Western Europe / Northern Europe. This has a colour temperature of approximately 6500K.



Reflectance

Most of the light we see is reflected from objects – these objects absorb and reflect a certain part of the light spectrum.

Example:



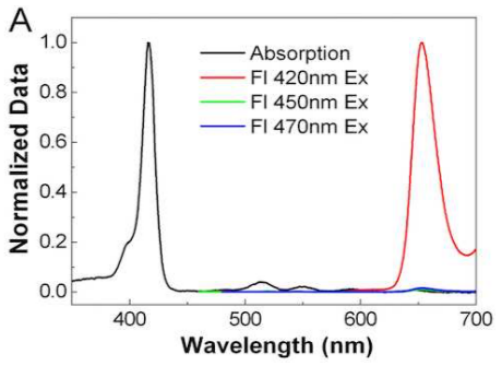
$$Reflected\ Light\ (L) = Illumination\ (I) * Reflectance\ (R)$$

This shows that the same object may appear to have different colours depending on the different illumination.

Fluorescence

Fluorescence is the emission of light by a substance that has absorbed light or other EM radiation. It is a form of **luminescence.** In most cases, the emitted light has a longer wavelength and therefore lower energy than the absorbed radiation.

Most striking when absorbed radiation is in UV and thus invisible while the emitted light is in the visible region.

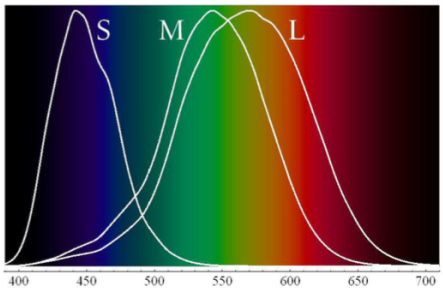


Colour Vision

We 'see' colour through the use of cones – which are the photoreceptors responsible for colour vision. They only work in daylight – when there is not enough light.

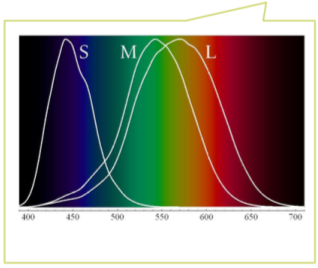
We have three types of cones:

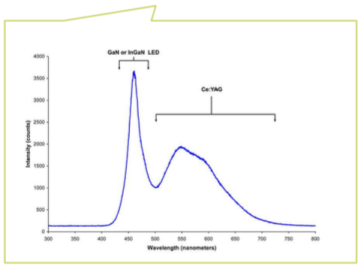
- S – sensitive to short wavelengths
- M – sensitive to medium wavelengths
- L – sensitive to long wavelengths



Sensitivity curves – probability that a photon of that wavelengths will be absorbed by a photoreceptor

- cone response = sum(sensitivity * reflected light)





Although there is an infinite number of wavelengths, we have only three photoreceptor types to sense differences between light spectra

Formally

$$R_S = \int_{380}^{730} S_S(\lambda) \cdot L(\lambda) d\lambda$$

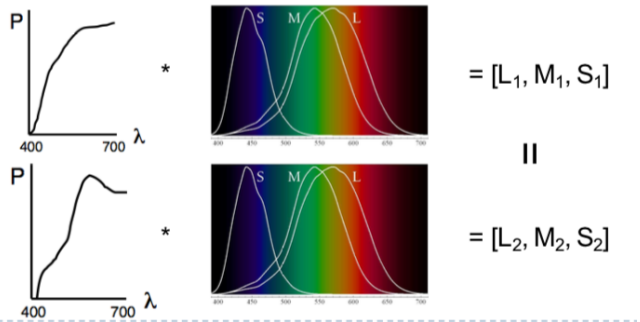
Index S for S-cones

217

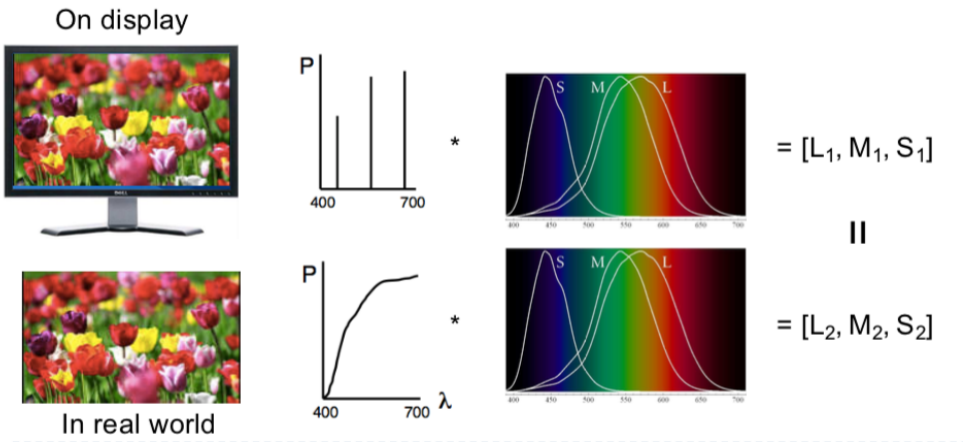
Metamerism

Due to the specific sensitivity curves, even if two light spectra are different, they may appear to have the same colour.

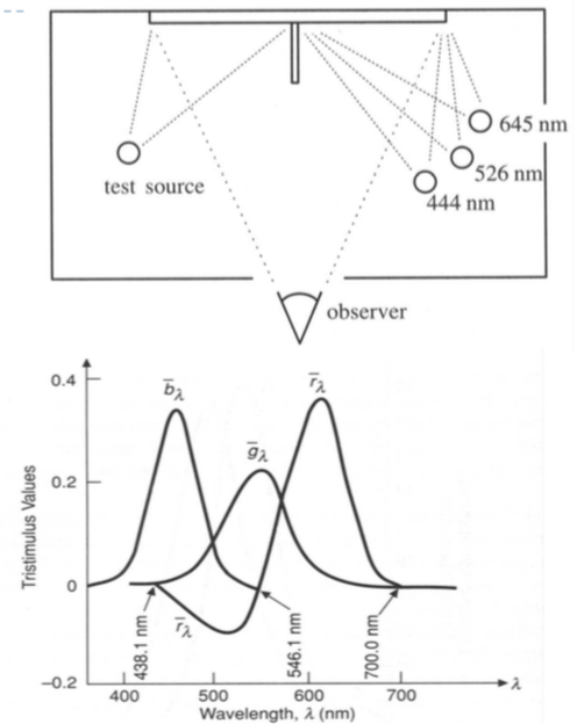
The light spectra that appear to have the same colour are called **metamers.**



We use this in the real world for displays – by displaying light in three specific wavelengths rather than continuous to replicate real world images.



It can be shown that any colour can be matched using three linear independent reference colours (**Tristimulus Colour Representation**). **However, it may require NEGATIVE contribution to test colour.** The matching curves describe the value for matching monochromatic spectral colours of equal intensity (with respect to a certain set of primary colours).

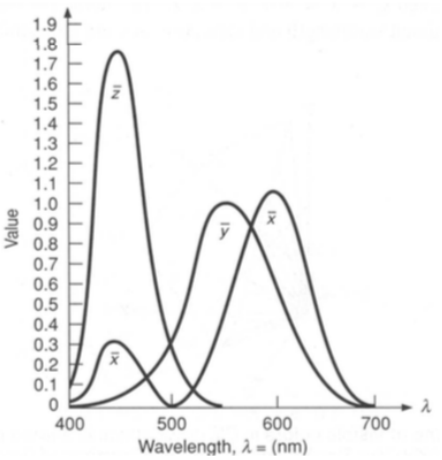


Standard Colour Space CIE-XYZ

1. CIE Experiments [Guild and Wright, 1931]
    a. Colour matching experiments
    b. Group ~12 people with normal colour vision
    c. 2-degree visual field (fovea only)
2. CIE 2006 XYZ
    a. Derived from LMS colour matching functions by Stockman & Sharpe
    b. S-cone response differs the most from CIE 1931
3. CIE-XYZ Colour Space
    a. **Goals**

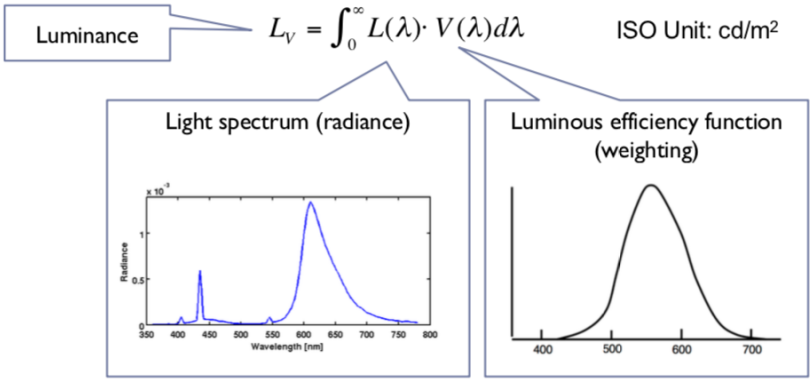
1. Abstract from concrete primaries use in experiment
2. All matching functions are positive
3. One primary is roughly proportional to light intensity

4. **Standardized Imaginary Primaries (CIE XYZ 1931)**
   a. Could match all physically realisable colour stimuli
   b. Y is roughly equivalent to luminance – the shape is similar to luminous efficiency curve
   c. Monochromatic spectral colours form a curve in 3D XYZ-space.
   d. Cone sensitivity curves can be obtained by a linear transformation of CIE XYZ
   e.



Luminance

Luminance is the perceived brightness of light, adjusted for the sensitivity of the visual system to wavelengths.



CIE Chromaticity Diagram

The chromaticity values are defined in terms of x, y, z:

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \quad \therefore \quad x+y+z=1$$

This ignores luminance and can be plotted as a 2D functions.

Pure colours (single wavelength) all lie along the outer curve and all other colours are a mix of pure colours and hence lie inside the curve. The points outside the curve do not exist as colours.

Displayable Colours
All physically possible and visible colours form a solid in XYZ space.

Each display device can reproduce a subspace of that space – a chromacity diagram is a slice taken from a 3D solid in XYZ space.

The Colour Gamut is the solid in a colour space – it is usually defined in XYZ to be device-independent.

Representing Colours

We need a mechanism to represent colour in the computer by some set of numbers. It needs to be:

- Small – so it can be quantised to be a small number of bits
  - Linear and Gamma Corrected RGB, sRGB
- Set of numbers which are easy to interpret
  - Munsell's artists' scheme
  - HSV, HLS
- Such that Euclidean colour differences are perceptually uniform
  - CIE Lab, CIE Luv

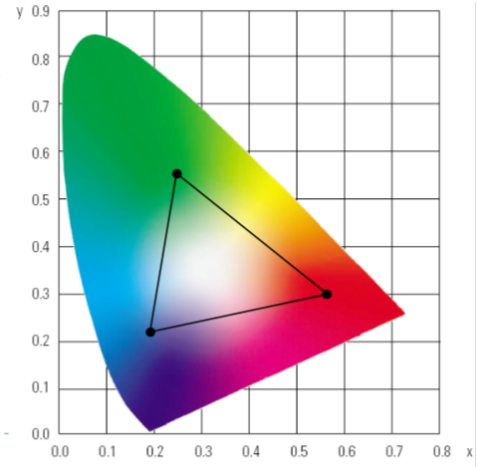
RGB Space

RGB space is cube of combining red, green and blue lights. It is used for all display devices – TVs, CRT monitors, LCDs, etc.

The device puts its own physical limitations on:

1. Range of colours
2. Brightest colours
3. Darkest colour

The red, green and blue primaries each map to a point in XYZ space and any triangle within the resulting triangle can be displayed – any colour outside the triangle cannot be displayed.



CMY Space

Printers make colours by mixing coloured inks. The important difference between inks (CMY) and lights (RGB) is that, while lights emit light and inks absorb light.

- **Cyan absorbs red, reflects blue and green**

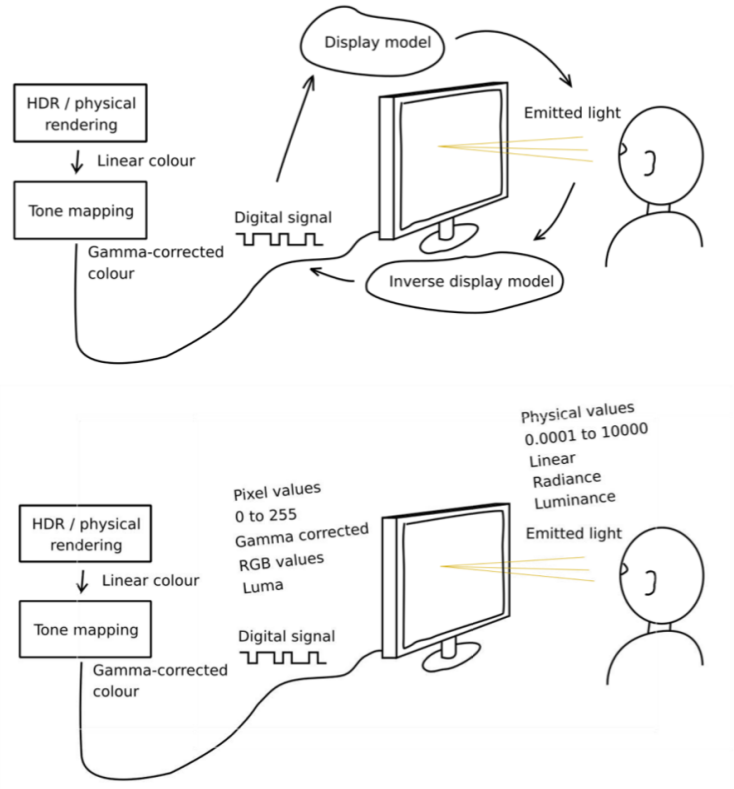
- **Magenta absorbs green, reflects red and blue**
- **Yellow absorbs blue, reflects green and red.**

It is effectively the inverse of RGB – therefore still a cube. However, it still doesn't really work – therefore in real printing, we use black (key) as well as CMY **(CMYK Space).**

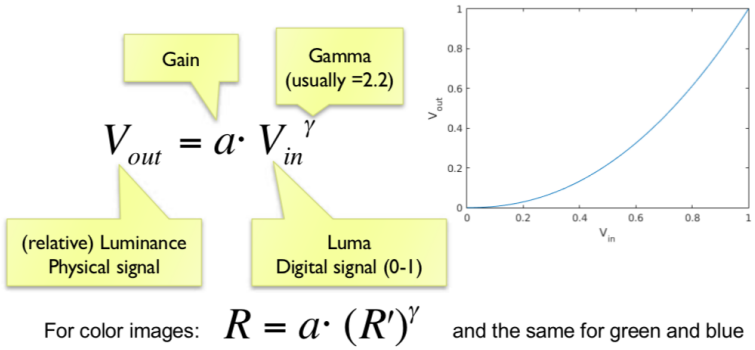
We use black because:

1. Inks are not perfect absorbers
2. Mixing C, M, Y gives a muddy grey, not black
3. Lots of text is printed in black: therefore, trying to align C, M and Y perfectly for black text would be a nightmare.

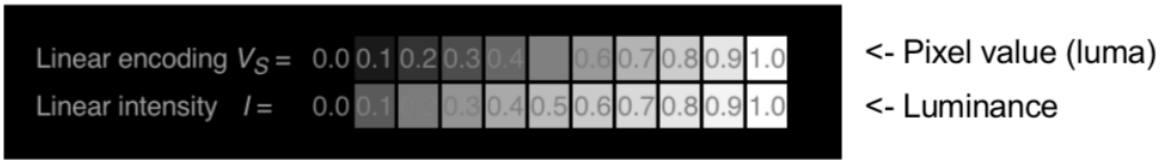
Linear vs. Gamma-Corrected Values



Gamma correction is used to encode luminance or tri-stimulus colour values (RGB) in imaging systems.



For color images: $R = a \cdot (R')^{\gamma}$ and the same for green and blue



- Gamma corrected pixel values give a scale of brightness levels that is more perceptually uniform
- At least 12 bits (instead of 8) would be needed to encode each colour channel without gamma correction.
    - The response of the CRT gun

Luma

Luma is the pixel brightness in gamma corrected units:

- $L' = 0.2126R' + 0.7152G' + 0.0722B'$
- $R', G' and\ B'$ are gamma corrected colour values
- It is used in image / video coding.

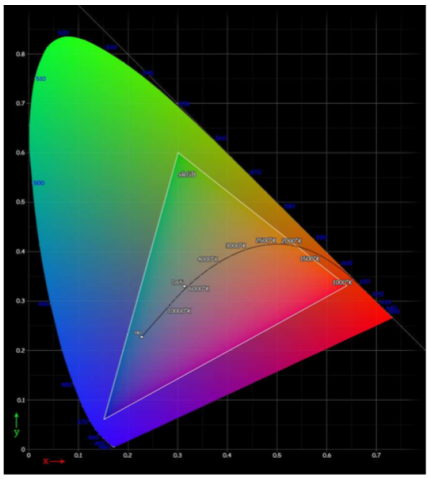
RELATIVE LUMINANCE is often approximated with:

- $L = 0.2126R + 0.7152G + 0.0722B = 0.2126(R')^{y} + 0.7152(G')^{y} + 0.0722(B')^{y}$
- Where R, G and B are linear colour values
- **Luma** and **luminace** are different quantities despite similar formulas

sRGB Space

sRGB space is not standard – colours may differ based on the choice of primaries. **sRGB is a standard colour space which most displays try to imitate:**

Chromaticity	Red	Green	Blue	White point
x	0.6400	0.3000	0.1500	0.3127
y	0.3300	0.6000	0.0600	0.3290
z	0.0300	0.1000	0.7900	0.3583



To get from XYZ to sRGB:

1. Linear Colour Transform:

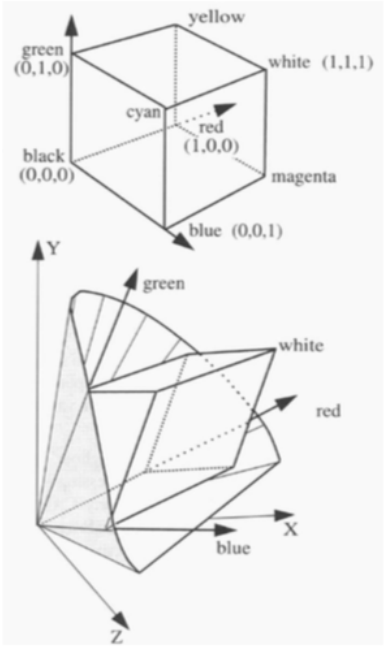
$$\begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} = \begin{bmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

    a.
2. Non-linearity

$$C_{\text{srgb}} = \begin{cases} 12.92C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308 \\ (1+a)C_{\text{linear}}^{1/2.4} - a, & C_{\text{linear}} > 0.0031308 \end{cases}$$

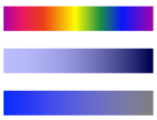
a. $a = 0.055$

b.

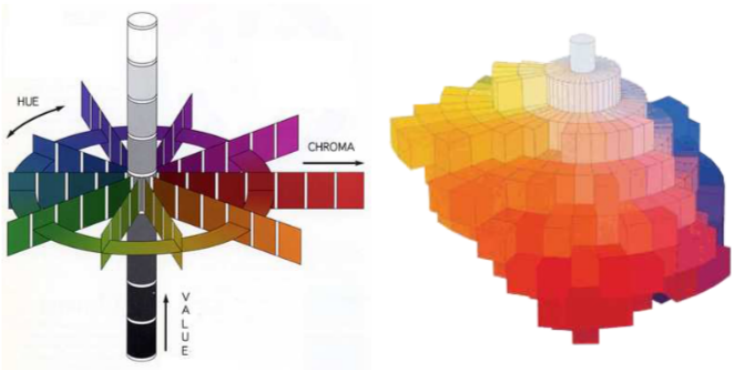


Munsell's Colour Classification System

- Invented by Albert H. Munsell, an American artist in 1905, in an attempt to systematically classify colours.
- Has three axes
  - hue ➢ the dominant colour
  - value ➢ bright colours/dark colours
  - chroma ➢ vivid colours/dull colours



- It can be represented using a 3D graph



- Any two adjacent colours are a standard 'perceptual' distance apart
  - It is worked out by testing it out on people
  - It is a highly irregular space
    - Vivid yellow is much brighter than vivid blue

Colour spaces conclusion

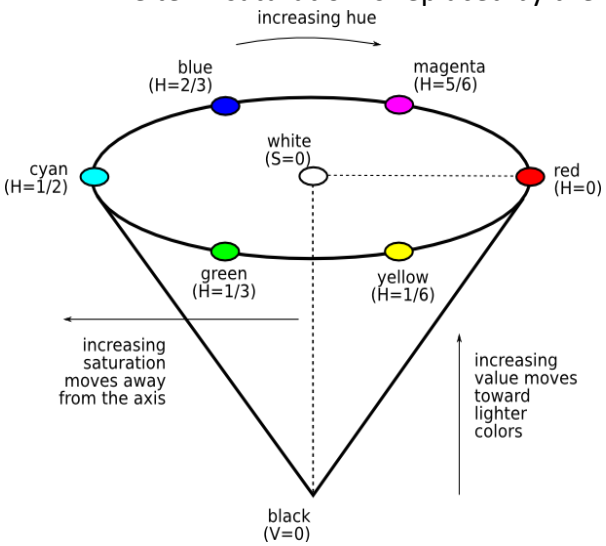
- RGB and CMY are based on the physical devices to produce the coloured output
  - They are difficult for humans to use to select colours

- Munsell's colour system is much more intuitive
    - Hue – what is the principal colour?
    - Value – how light or dark is it?
    - Chroma – how vivid or dull is it?
- Therefore, produced basic transformations of RGB which resemble Munsell's human friendly system.

HSV

Also has three axes (defined the same as Munsell's):

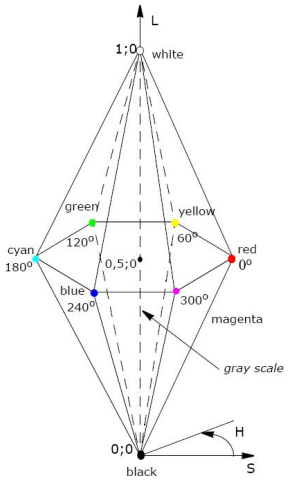
1. Hue and value have the same meaning
2. The term saturation is replaced by the term chroma



It was designed by Alvy Ray Smith in 1978 with a precise algorithm to convert between HSV and RGB.

HLS

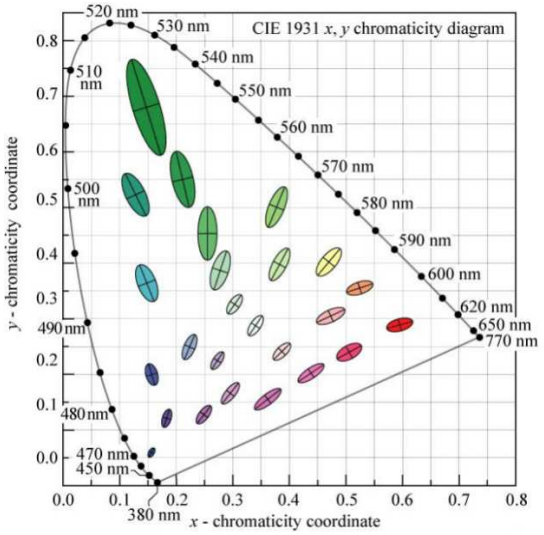
- HLS is a simple variation of HSV where hue and saturation have the same meaning and the term **lightness replace the term value.**
- It is designed to address the complaint that **HSV has all pure colours having the same lightness / value as white.**



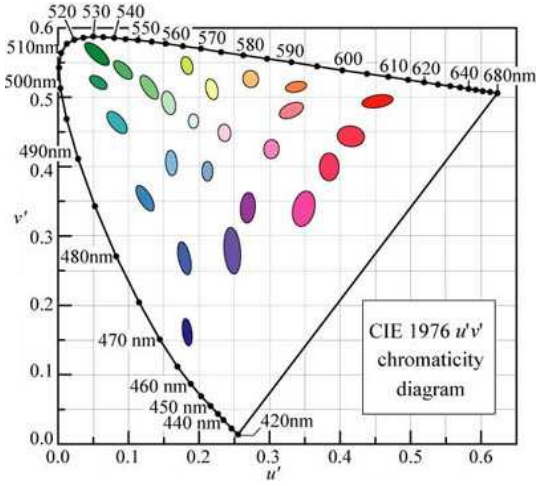
H = Hue
S = Saturation
L = Lightness

It was designed by Metrick in 1979 and as with HSV an algorithm is provided to convert between HLS and RGB.

However, in CIE xy chromatic colours and any derived things (RGB, HLS, HSV), the colours aren't perceptually uniform. You can show this by using graphs with MacAdam ellipses which groups together visually indistinguishable colours:



In CIE xy chromatic coordinates



In CIE u'v' chromatic coordinates

CIE L*u*v* and u'v'

This was made in order to be approximately perceptually uniform.

- u'v' chromacity

$$u' = \frac{4X}{X+15Y+3Z} = \frac{4x}{-2x+12y+3}$$
$$v' = \frac{9Y}{X+15Y+3Z} = \frac{9y}{-2x+12y+3}$$

- CIE LUV

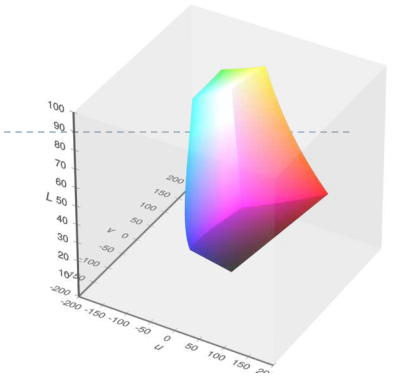
Lightness:
$$L^* = \begin{cases} \left(\frac{29}{3}\right)^3 Y/Y_n, & Y/Y_n \le \left(\frac{6}{29}\right)^3 \\ 116(Y/Y_n)^{1/3} - 16, & Y/Y_n > \left(\frac{6}{29}\right)^3 \end{cases}$$

Chromacity coordinates:
$$u^* = 13L^* \cdot (u' - u'_n)$$
$$v^* = 13L^* \cdot (v' - v'_n)$$

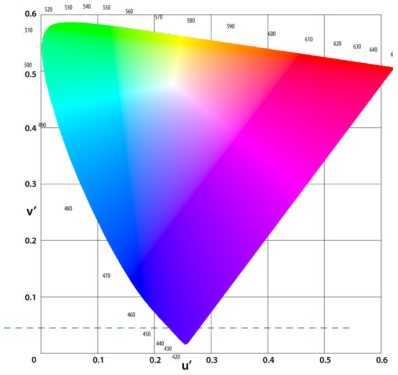
Colours less distinguishable when dark

- Hue and chroma

$$C^*_{uv} = \sqrt{(u^*)^2 + (v^*)^2}$$
$$h_{uv} = \text{atan2}(v^*, u^*),$$



sRGB in CIE L*u*v*



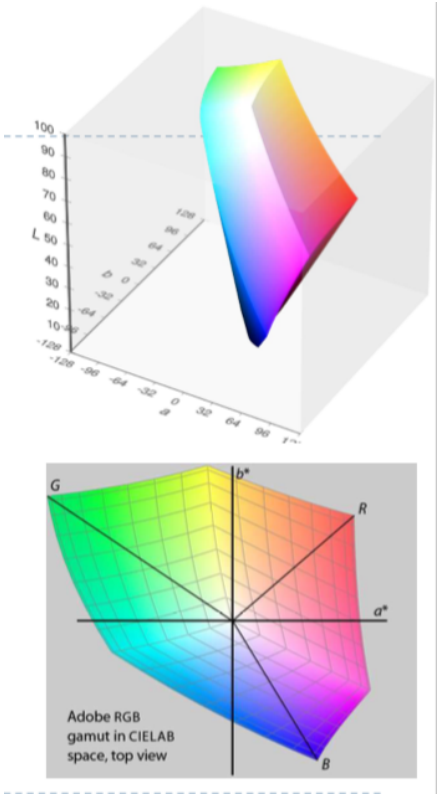
CIE L*a*b* colour space

This is another perceptually approximately uniform colour space:

$$L^* = 116 f\left(\frac{Y}{Y_\mathrm{n}}\right) - 16$$
$$a^* = 500\left(f\left(\frac{X}{X_\mathrm{n}}\right) - f\left(\frac{Y}{Y_\mathrm{n}}\right)\right)$$
$$b^* = 200\left(f\left(\frac{Y}{Y_\mathrm{n}}\right) - f\left(\frac{Z}{Z_\mathrm{n}}\right)\right)$$

Trichromatic values of the white point, e.g.
$X_\mathrm{n} = 95.047,$
$Y_\mathrm{n} = 100.000,$
$Z_\mathrm{n} = 108.883$

$$f(t) = \begin{cases} \sqrt[3]{t} & \text{if } t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & \text{otherwise} \end{cases}$$
$$\delta = \frac{6}{29}$$



- Chroma and hue

$$C^* = \sqrt{a^{*2} + b^{*2}}, \qquad h^\circ = \arctan\left(\frac{b^*}{a^*}\right)$$

Lab space

Lab space is a space which contains all the colours that a human can perceive. The image shows a visualisation of this. We note:

1. Human perception of colour is not uniform
2. Perception of colour diminishes at the white and black ends of the Lightness axis.
3. The maximum perceivable chroma differs for different hues.