3D Graphics Programming Tools

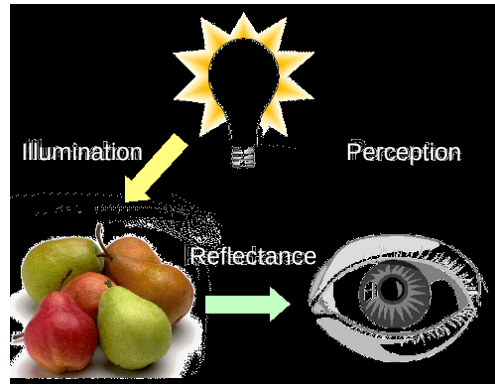
Colour

Today's agenda

- Perception of colour
- Colour spaces
- Colours in OpenGL

Basics of colour

- To understand how to make realistic images, we need a basic understanding of the physics and physiology of colour vision.

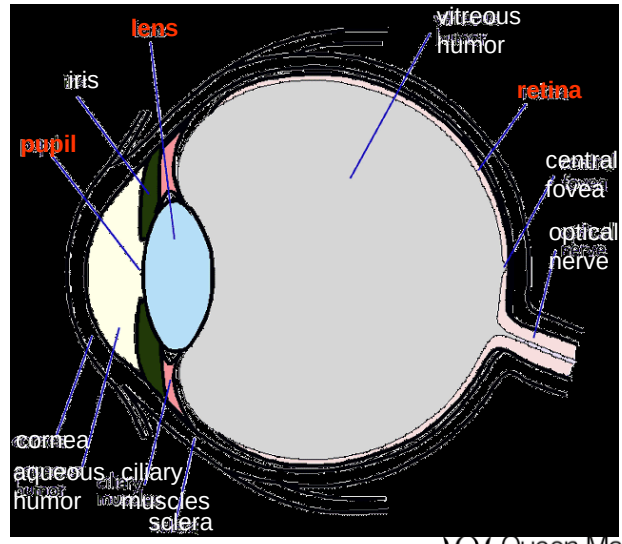


Basics of colour

- Physics
  - Illumination
    - Electromagnetic spectra
  - Reflection
    - Material properties
    - Surface geometry and microgeometry (i.e., polished versus matte versus brushed)
- Perception
  - Physiology and neurophysiology
  - Perceptual psychology

Physiology of vision

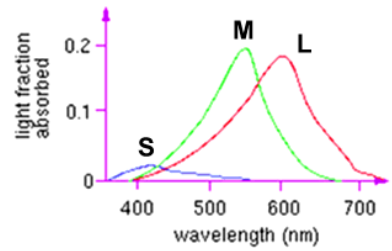
- The eye:



The retina
- Rods
- Cones
  - Color!

Physiology of vision: cones

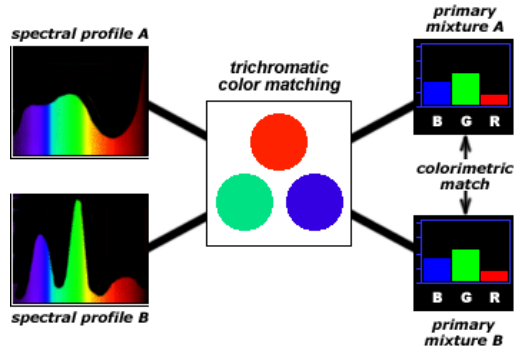
- Cones → three types
  - **L** or **R**, most sensitive to red light (610 nm)
  - **M** or **G**, most sensitive to green light (560 nm)
  - **S** or **B**, most sensitive to blue light (430 nm)



  - Colour blindness results from missing cone type(s)

Perception: metamers

- A given perceptual sensation of colour derives from the stimulus of all three cone types.
- Identical perceptions of colour can thus be caused by very different spectra.

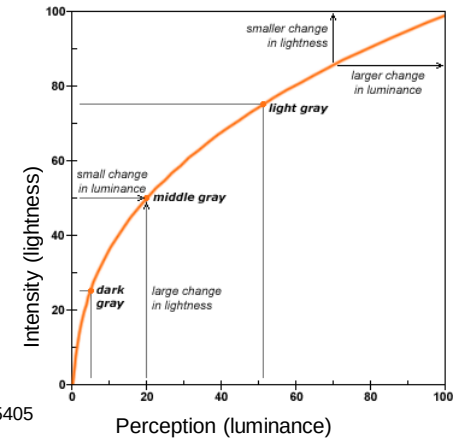


Perception

- Colour perception is also difficult because:
  - It varies from person to person
  - It is affected by adaptation
  - It is affected by surrounding colours

Perception: relative intensity

- We are not good at judging absolute intensity
- We perceive relative intensities, not absolute



Queen Mary University of London

Today's agenda

- Perception of colour
- Colour spaces
- Colours in OpenGL

Queen Mary University of London

Specifying colour

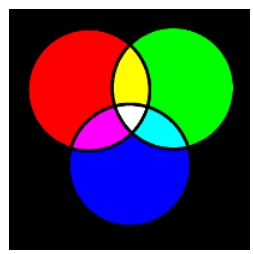
- Colour perception → three quantities
  - *Hue* → Distinguishes between colours like red, green, blue, etc.
  - *Saturation* → How far the color is from a gray of equal intensity
  - *Lightness* → The perceived intensity of a reflecting object

(Sometimes lightness is called *brightness* if the object is emitting light instead of reflecting it)

In order to use colour precisely in computer graphics, we need to be able to specify and measure colours

Colour Models

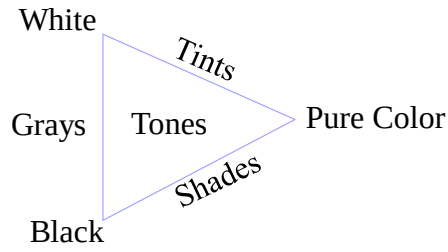
Additive (RGB)



Subtractive (CMYK)

How do artists do it?

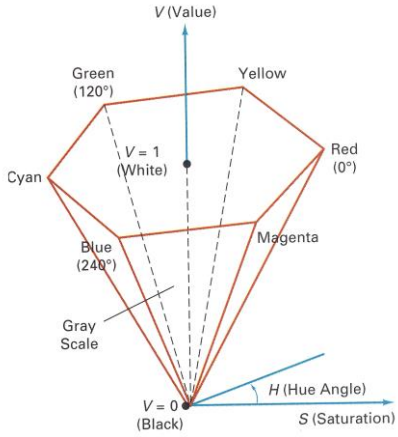
- Artists often specify color as tints, shades, and tones of saturated (pure) pigments
  - Tint
    - Obtained by adding white to a pure pigment, decreasing saturation
  - *Shade*
    - Obtained by adding black to a pure pigment, decreasing lightness
  - Tone
    - Obtained by adding white and black to a pure pigment



HSV colour space

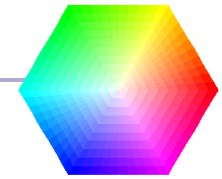
- Computer scientists → use an intuitive colour space that corresponds to tint, shade, and tone:
  - Hue
    - The colour we see (red, green, purple)
  - Saturation
    - How far is the colour from gray
      (pink is less saturated than red, sky blue is less saturated than royal blue)
  - Brightness (Luminance)
    - How bright is the color (how bright are the lights illuminating the object)

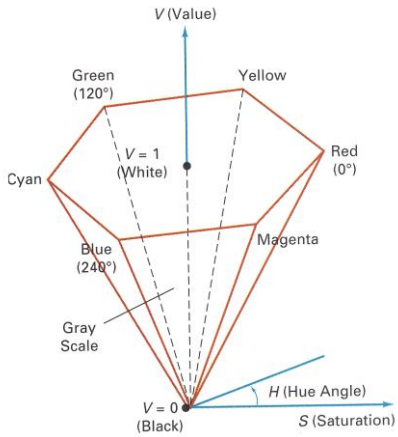
HSV colour model



- Hue (H) is the angle around the vertical axis
- Saturation (S) is a value from 0 to 1 indicating how far from the vertical axis the color lies
- Value (V) is the height of the "hexcone"

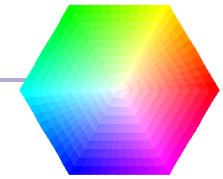
HSV colour model



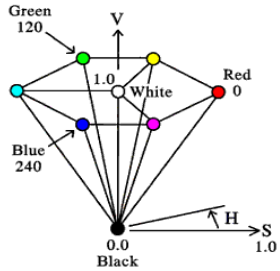


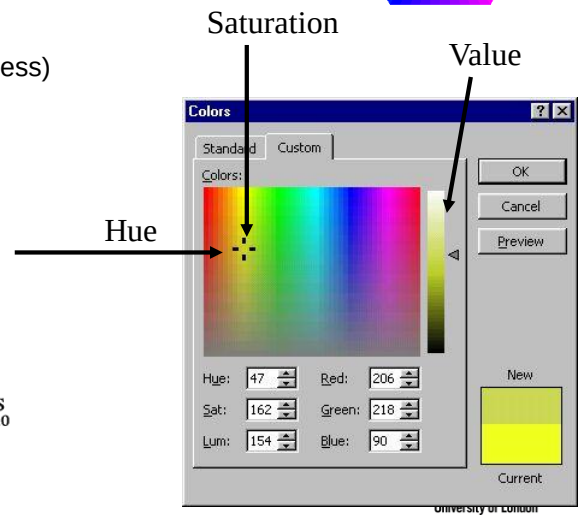
H	S	V	Color
0	1.0	1.0	Red
120	1.0	1.0	Green
240	1.0	1.0	Blue
*	0.0	1.0	White
*	0.0	0.5	Gray
*	*	0.0	Black
60	1.0	1.0	?
270	0.5	1.0	?
270	0.0	0.7	?

HSV colour space



- Intuitive colour space
  - H = Hue
  - S = Saturation
  - V = Value (or brightness)



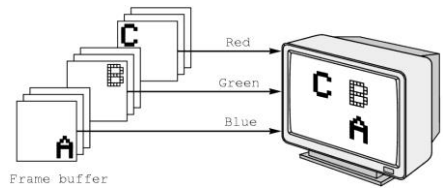


Today's agenda

- Basics of colour
- Perception of colour
- Colour spaces
- Colours in OpenGL

RGB colour in OpenGL

- Each colour component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in **`glColor3f`** the colour values range from 0.0 (none) to 1.0 (all), whereas in **`glColor3ub`** the values range from 0 to 255



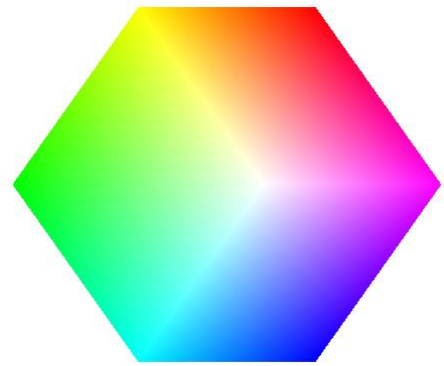
Colour and State

- The colour as set by **`glColor`** becomes part of the state and will be used until changed
  - Colours and other attributes are not part of the object but are assigned when the object is rendered
- We can create conceptual *vertex colours* by code such as:

```
glColor
glVertex
glColor
glVertex
```

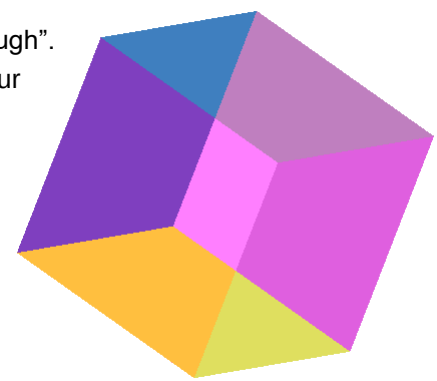
Smooth Colour

- Default is *smooth* shading
  - OpenGL interpolates vertex colours across visible polygons
- Alternative is *flat shading*
  - Colour of first vertex determines fill colour
- **`glShadeModel (GL_SMOOTH)`** or **`GL_FLAT`**



RGBA and Transparency

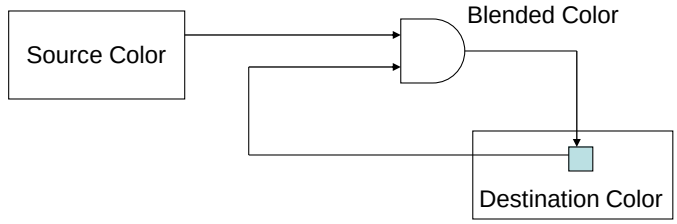
- In RGBA mode, we use a fourth colour component: A or alpha, which is an opacity.
- An opacity of 1.0 means the colour is opaque and cannot be “seen through”.
- An opacity of 0.0 means that a colour is transparent.



```
GLfloat colors[][4] = {{0.5,0.5,0.5,0.5},
{0.0,1.0,0.0,0.5}, {1.0,0.0,1.0,0.5},
{1.0,0.0,0.0,0.5}, {0.0,0.0,1.0,0.5},
{1.0,1.0,0.0,0.5}};

glColor4fv(colors[0]);
```

Blending

- In normal rendering, the alpha value is ignored. We must first enable blending: glEnable(GL_BLEND);
- Alpha values between 0.0 and 1.0 indicate that the material is semitransparent and some of the colors of objects behind it blend with its color.

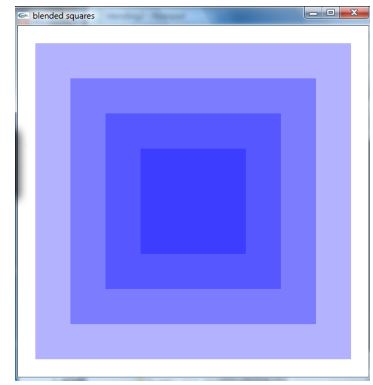


- $(S_rR_s + D_rR_d, S_gG_s + D_gG_d, S_bB_s + D_bB_d, S_aA_s + D_aA_d)$
- glBlendFunc(Glenum source, Glenum destination);

  e.g. glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

Blending - example

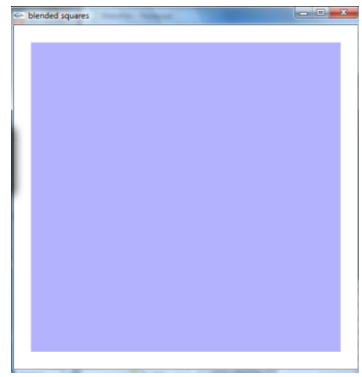
```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor4f(0.0, 0.0, 1.0, 0.3);
  asquare(0.9);
  asquare(0.7);
  asquare(0.5);
  asquare(0.3);
 glutSwapBuffers();
}

void init()
{
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA,
                    GL_ONE_MINUS_SRC_ALPHA);
  glClearColor(1.0, 1.0, 1.0, 1.0);
}
```



Blending - example

- Initially the colour buffer is cleared to an opaque colour (R_c, G_c, B_c, 1)
- When blended with : (R_s, G_s, B_s, A_s) …

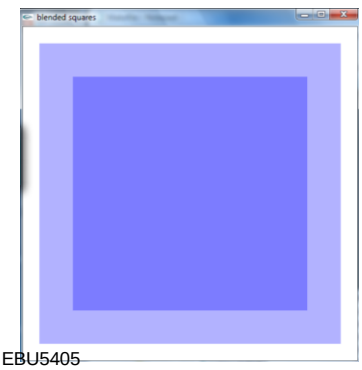
$(A_sR_s + (1-A_s)R_c, A_sG_s + (1-A_s)G_c, A_sB_s + (1-A_s)B_c, A_s^2 + (1-A_s))$



Blending - example

- We add a second polygon with colour ($R_{s'}$, $G_{s'}$, $B_{s'}$, $A_{s'}$)
- The colour where the two polygons overlap will be : …

$(A_{s'}R_{s'} + (1-A_{s'})(A_sR_s + (1-A_s)R_c),\ A_{s'}G_{s'} + (1-A_{s'})(A_sG_s + (1-A_s)G_c),$
$A_{s'}B_{s'} + (1-A_{s'})(A_sB_s + (1-A_s)B_c),\ A_{s'}^2 + (1-A_{s'})\ (A_s^2 + (1-A_s)A_c))$



The order in which we render the polygons Matters !