CS4160

COMPUTER GRAPHICS

class 2

1

1

today's class

rendering architecture overview
mathematical preliminaries

<break>

light & images
the OpenEXR image format

2

2

sep 22: class reschedule

apologies!

choices:

wednesday (sep 21) 6:30-8 pm

or thursday (sep 22) 8-9:30am

3

3

TA office hours

He:	T 3:00 - 5:00
Justin:	W 4:00 - 6:00
Ray:	T/Th 1:00-2:00

[in 122A Mudd]

[changes will be broadcast]

4

4

courseworks/canvas - set to “notify”

make sure you are getting broadcast msgs

check in every once in a while!

getting help

1 - try to solve the problem! (notes/book/canvas)

2 - is it coding?

- if YES: have you used the debugger?
(if not, go to step 1)

3 - post to canvas - include the words “I used the debugger, and the result was…”, and include image if possible.

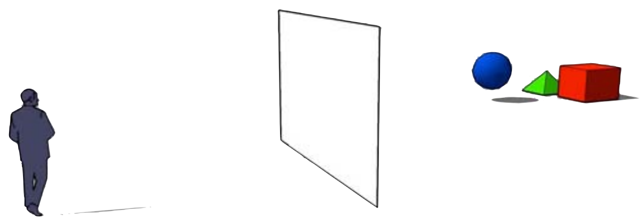
4 - go to TA office hours

5 - if the above fails to solve your problem, contact me

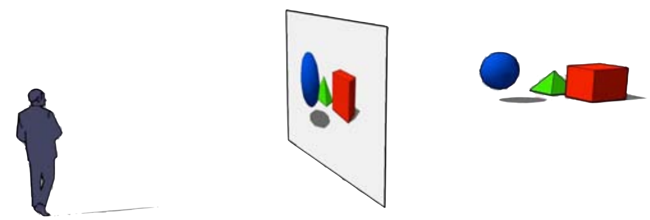
renderer architectures for 3d graphics : an overview

rendering

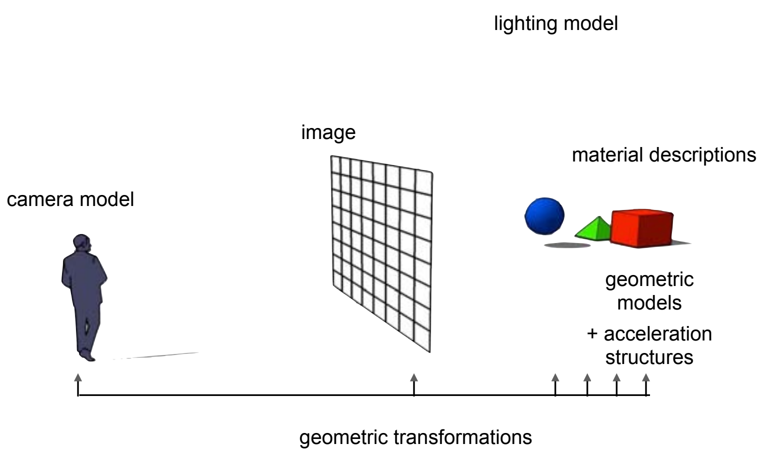
rendering



rendering



rendering overview



rendering

2 common methods:

- raytracing

  examples: Mental Ray, Blue Sky's CGIStudio, etc. etc.

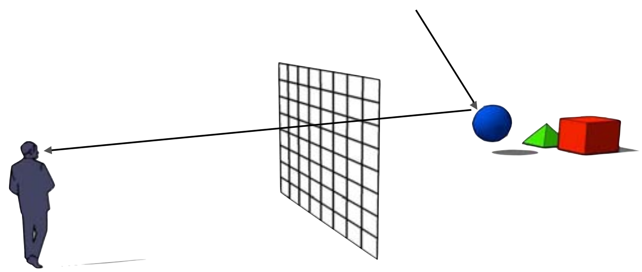
- pipeline rendering

  called “object-order rendering” in the book

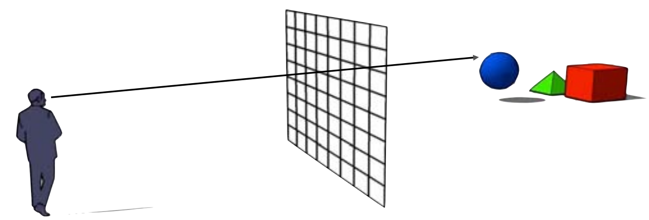
  often called “scanline rendering”

  examples: Pixar's PRMan, Maya's default renderer, most commercial renderers, 99.99% of hardware renderers

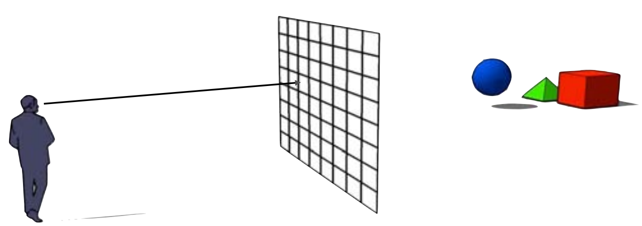
ray tracing



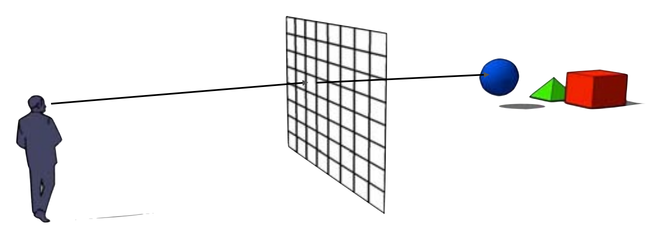
ray tracing



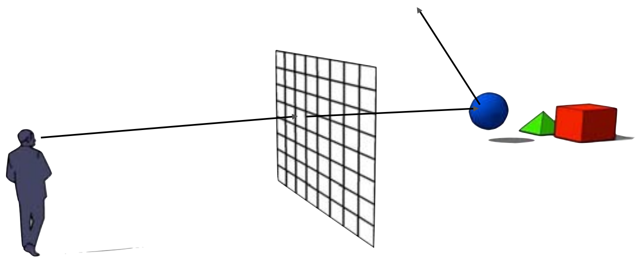
ray tracing



ray tracing



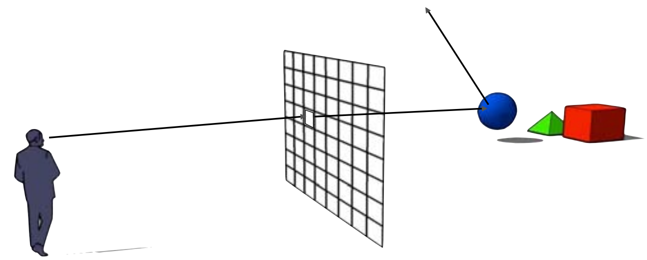
ray tracing



17

17

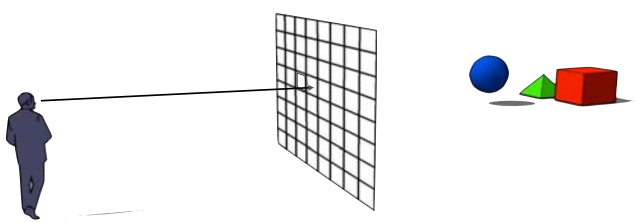
ray tracing



18

18

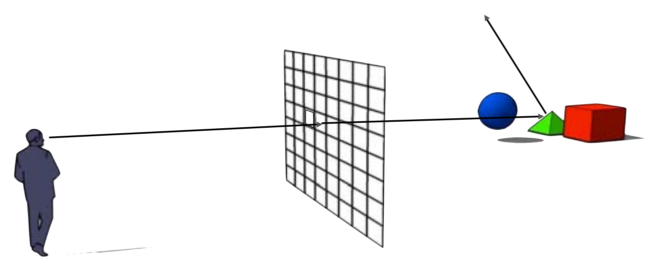
ray tracing



19

19

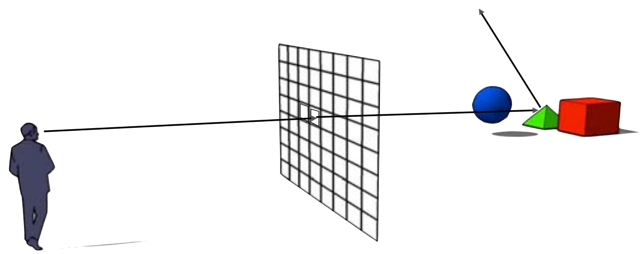
ray tracing



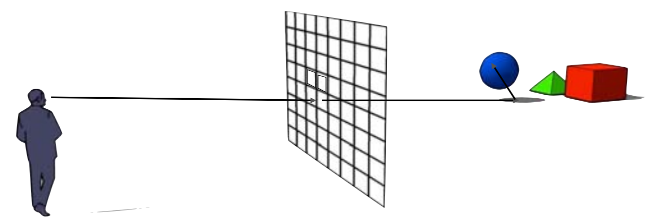
20

20

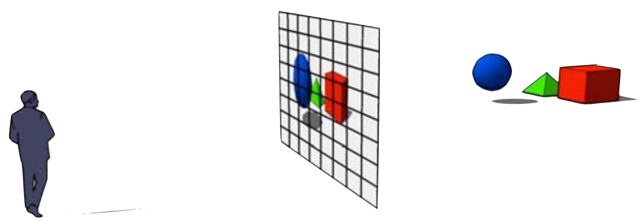
ray tracing



ray tracing



ray tracing



ray tracing

operations:

ray-object intersection

shading: lighting & materials calculation

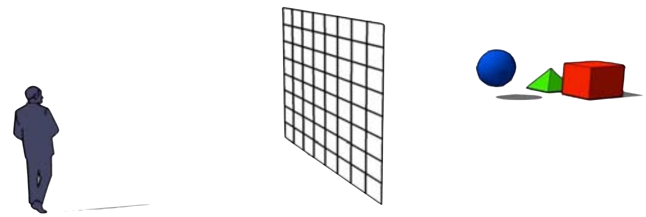
geometric transformations

ray tracing

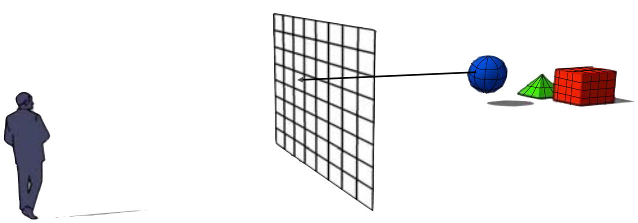
"image-order rendering"

```
for i = 1 to image_width {
  for j = 1 to image_height {

    generate "ray" from focal point through pixel (i,j)
    which_obj = first object ray intersects in scene
    calculate the shading on which_obj at intersection
    pixel (i,j) = result of shading calculation

    }
}
```

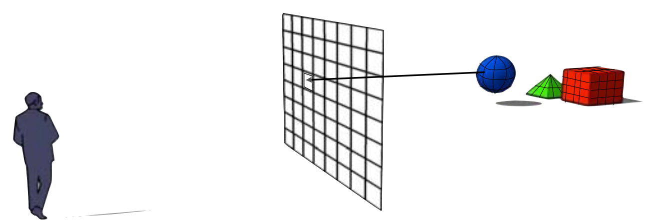

pipeline rendering



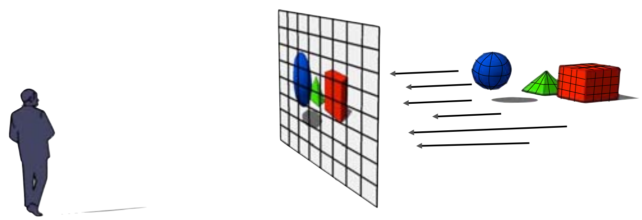
pipeline rendering



pipeline rendering



pipeline rendering



pipeline rendering

geometric operations:

"dicing" i.e tesselation

shading: lighting & materials calculation

geometric transformations

perspective projections

optimized sorting for handling occlusion

pipeline rendering

“object-order rendering”

```
for i = 1 to num_objects {

     split object(i) into fragments
     calculate the shading on each fragment
     project fragments to pixels
     pixel keeps color of closest fragment

}
```


ray tracing vs. pipeline rendering

raytracing has an intuitive geometric analogy that extends to many phenomena (shadows, reflections, transparency, etc.)

pipeline methods are often much faster

each has extensions for more realistic rendering: radiosity and global illumination

for interactive rendering, pipeline methods dominate, while for physically-based simulation of light transport, ray tracers dominate.

mathematical preliminaries

mathematical preliminaries

points

vectors

coordinate systems

equation forms

linear interpolation

barycentric combinations

points

denotes a location in 1D, 2D, 3D, or *n*D

a = [1 -2] **b** = [2 -1 3]

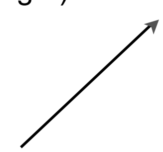
this location is with respect to a particular **coordinate system** (to be defined later)

individual components referred to as $\mathbf{a}_x$, $\mathbf{a}_y$, $\mathbf{a}_z$ (or generally, $\mathbf{p}_x$, $\mathbf{p}_y$, $\mathbf{p}_z$)

the number of components is referred to as the **dimension**

vectors

vectors describes both an orientation and a magnitude (i.e. “direction” and length)

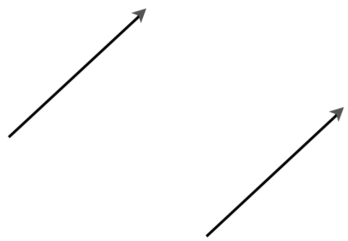


in graphics, vectors are used to denote:

- geometric properties (like surface orientations)
- the direction that light follows as it illuminates a scene
- reflectance properties of materials
- and many other phenomena

vectors

two vectors are **equivalent** if they have the same orientation and magnitude (like these):

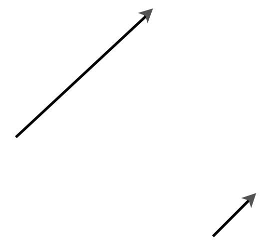


37

37

vectors

these two vectors have the same orientation, but different magnitudes, and so are **not** equivalent

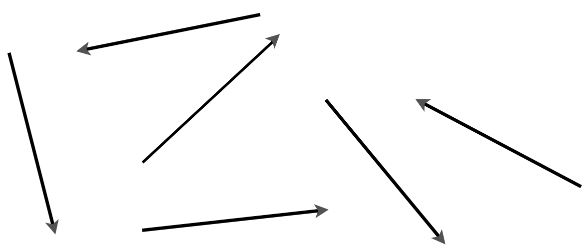


38

38

vectors

these vectors have the same magnitude, but different orientations, and so are **not** equivalent:



39

39

vectors

vectors are represented by ordered tuples of values, e.g.:

in 2D: [3 5]

in 3D: [2 4 9]

as with points, the number of values specifies the **dimension** of the vector

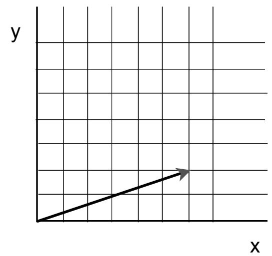
40

40

vectors

vectors should be visualized by imagining them in the canonical coordinate system ("Real Space"), aligned with the origin:

the vector [6 2]:

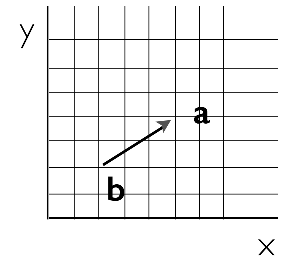


vectors

vectors may be constructed by the difference between two points:

v = a - b

$$\begin{bmatrix}3\\2\end{bmatrix} = \begin{bmatrix}5\\4\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix}$$

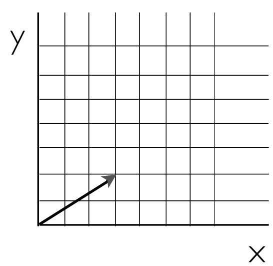


vectors

therefore, the vector [3 2] is the same as the vector formed by subtracting the points:

$$\begin{bmatrix}3\\2\end{bmatrix} = \begin{bmatrix}3\\2\end{bmatrix} - \begin{bmatrix}0\\0\end{bmatrix}$$

$$\begin{bmatrix}3\\2\end{bmatrix} = \begin{bmatrix}5\\4\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix}$$



vectors

vectors may be developed this way in any number of dimensions

v = a - b

2D: $\begin{bmatrix}3\\3\end{bmatrix} = \begin{bmatrix}5\\4\end{bmatrix} - \begin{bmatrix}2\\1\end{bmatrix}$

3D: $\begin{bmatrix}2\\2\\0\end{bmatrix} = \begin{bmatrix}1\\2\\1\end{bmatrix} - \begin{bmatrix}-1\\0\\1\end{bmatrix}$

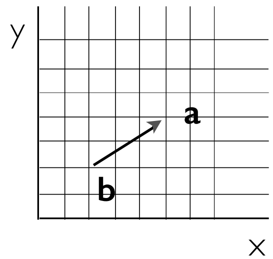
etc.

point + vector

vectors may be added to points - you can visualize this as a displacement:

a = b + v

$$\begin{bmatrix}5\\4\end{bmatrix} = \begin{bmatrix}3\\2\end{bmatrix} + \begin{bmatrix}2\\2\end{bmatrix}$$



vector magnitude

the magnitude (or "length") of a vector **v** is given by:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$$

where: $$v = \begin{bmatrix}v_1\\v_2\\\ldots\\v_n\end{bmatrix}$$

unit vectors

a vector of length 1, i.e.

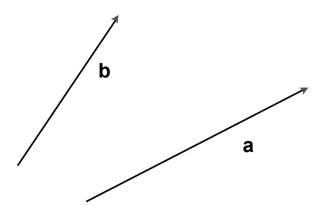
$$\|v\| = 1$$

is called a <u>unit vector</u>, and is commonly used in shading calculations

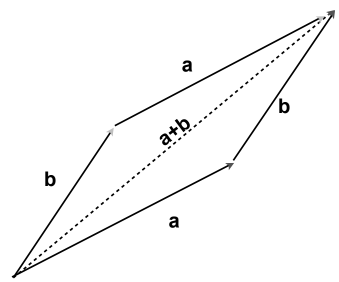
any nonzero vector may be **normalized** to unit length by:

$$\bar{v} = \frac{v}{\|v\|}$$

vector addition

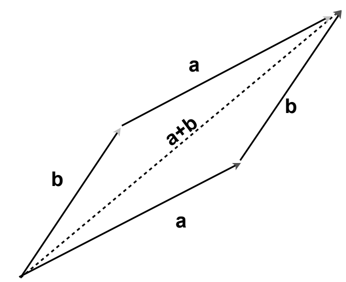


vector addition



n.b.: a+b = b+a

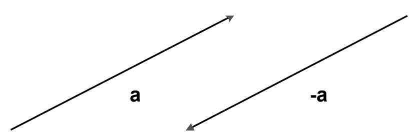
vector addition



$\mathbf{a} = [a_1\ a_2\ ...\ a_n]$, $\mathbf{b} = [b_1\ b_2\ ...\ b_n]$

$\mathbf{a+b} = [a_1+b_1\ \ a_2+b_2\ \ ...\ \ a_n+b_n]$

vector negation

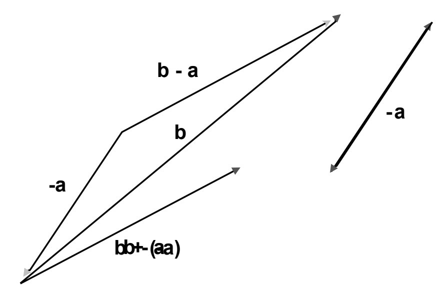


-a has the same magnitude, but opposite orientation, to **a**

vector subtraction

now that we have negation, we can define subtraction as:

b - a = b + (-a)

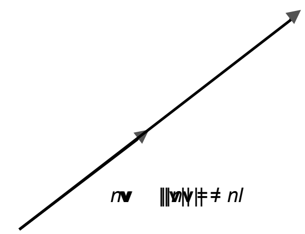


vector multiplication: scalar product

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix}, with \, \|v\| = l$$

$$n \cdot v = \begin{bmatrix} n \cdot v_1 \\ n \cdot v_2 \\ \dots \\ n \cdot v_n \end{bmatrix}, with \, \|n \cdot v\| = nl$$

vector multiplication: scalar product



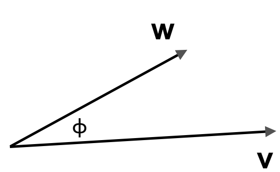
vector multiplication: dot product

for:

v = [v_1 v_2 ... v_n] and **w** = [w_1 w_2 ... w_n]

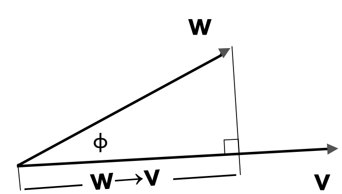
v·**w** = $v_1w_1 + v_2w_2 + \ldots + v_nw_n$ also:

v·**w** = ||**v**|| ||**w**|| cos ϕ



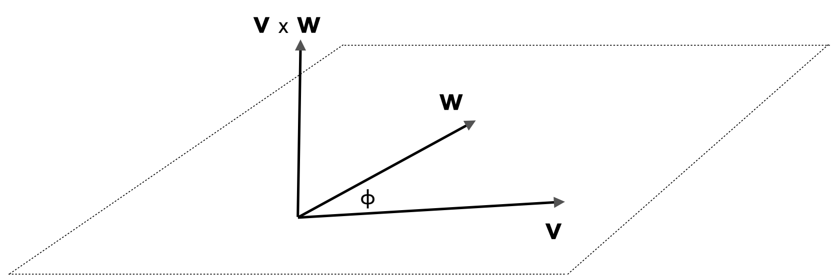
vector multiplication: dot product

useful geometric technique: projection of one vector onto another:



$$w \rightarrow v = \|w\| \, cos\phi = \frac{w \cdot v}{\|v\|}$$

vector multiplication: cross product



57

57

vector multiplication: cross product

$$v \times w = \begin{bmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{bmatrix}$$

58

58

vector multiplication: cross product

the cross product **v** x **w** is **perpendicular** to the constituent vectors **v**, **w**

its **magnitude** is given by:

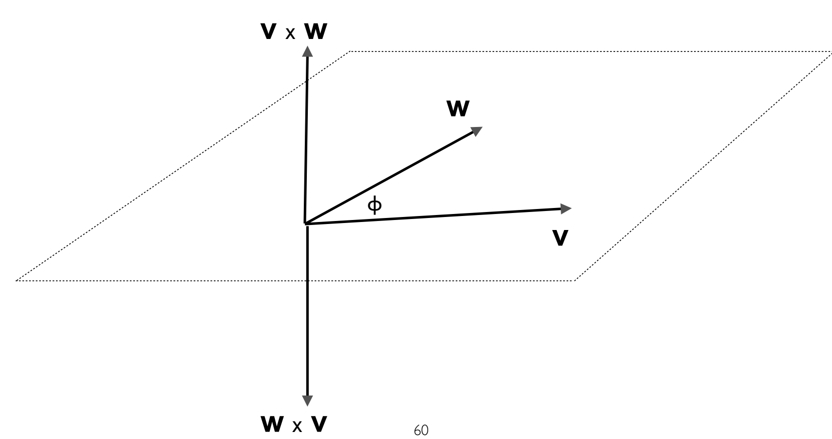
$$\| \mathbf{v} \times \mathbf{w} \| = \| \mathbf{v} \| \, \| \mathbf{w} \| \sin \phi$$

its **direction** is given by the **right-hand rule**, i.e. considering a counterclockwise rotation from **v** to **w**, **v** x **w** points “out” of the clock face

59

59

vector multiplication: cross product



60

60

uses of dot and cross products

both the dot product and the cross product can be used to get the angle between two vectors:

$$\mathbf{v} \cdot \mathbf{w} = ||v||\ ||w||\ \cos \phi$$

$$||\ \mathbf{v} \times \mathbf{w}\ || = ||\ \mathbf{v}\ ||\ ||\ \mathbf{w}\ ||\ \sin \phi$$

the dot product is useful for finding the projection of one vector on another

the cross product is useful for making 3-dimensional **coordinate systems** from two vectors

vector bases

any 2D vector can be written as a combination of 2 non-zero 2D vectors that are linearly independent (int this case, that means not parallel).

$\mathbf{c} = 1.7\mathbf{b} + 1.1\mathbf{a}$

vector bases

these 2 vectors form a **basis**

a 2D **basis**, in combination with an origin point forms a 2D <u>coordinate system</u>

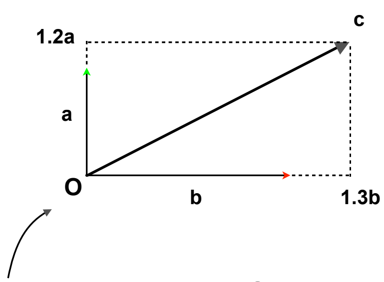
vector bases

any ***n***-dimensional vector can be written as a combination of ***n*** non-zero ***n***-dimensional vectors that are linearly independent (i.e. none can be composed of scaled sums of the others).

these vectors form an ***n***-dimensional **basis**

with an origin, they form a <u>coordinate system</u> in **n** dimensions

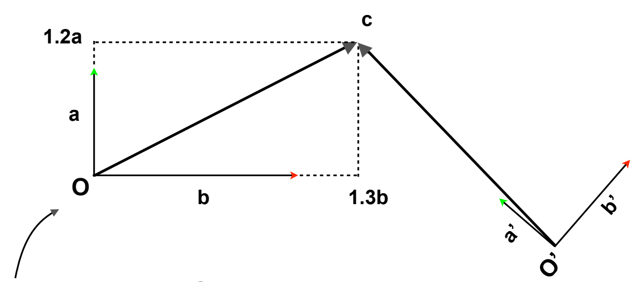
intro to coordinate systems



coordinate system's origin point

$$\mathbf{c} = \mathbf{O} + 1.3\mathbf{b} + 1.2\mathbf{a}$$

intro to coordinate systems



coordinate system's origin point

$$\mathbf{c} = \mathbf{O} + 1.3\mathbf{b} + 1.2\mathbf{a}$$

$$\mathbf{c} = \mathbf{O'} + .1\mathbf{b'} + 4.1\mathbf{a'}$$

intro to coordinate systems

points in space have different coordinates depending on the coordinate system used

choice of coordinate system is arbitrary

a specific coordinate system is often chosen to:

- help simplify the mathematics
- simplify a problem conceptually

choosing coordinate systems

example: car dashboard

in a moving car, the dashboard's location with respect to the earth is constantly changing

(here, the earth provides the coordinate system)

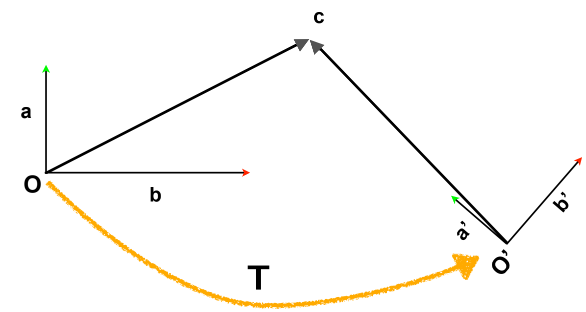
choosing coordinate systems

example: car dashboard

however, with respect to a fixed part of the car itself (e.g. the driver), the dashboard is at a fixed location

intro to coordinate systems

points represented in one coordinate system may be easily represented in another by using **geometric transformations**



$\mathbf{c} = \mathbf{O} + 1.3\mathbf{b} + 1.2\mathbf{a}$ $\quad$ $\mathbf{T}(\mathbf{O} + 1.3\mathbf{b} + 1.2\mathbf{a}) = \mathbf{O'} + .1\mathbf{b'} + 4.1\mathbf{a'}$

coordinate systems

coordinate systems allow us to uniquely define:

- points
- vectors

geometric transformations allow us to define:

- translation
- rotation
- scaling
- shearing
- etc.

...as transformations between coordinate systems

operations on vectors

vectors may be added, subtracted, rotated & otherwise transformed:

$$\mathbf{v} = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \alpha_3\mathbf{v}_3 + \ldots + \alpha_n\mathbf{v}_n$$

reminder: vectors invariant under translation.

operations on points

translation, rotation, & other geometric transformations:

p' = **p** + **v**

subtraction, but NO arbitrary addition!

linear combinations for points

while vectors may be combined in arbitrary linear combination, points may not!

$\mathbf{x} = \alpha\mathbf{p} + \beta\mathbf{q}$ true only if $\alpha + \beta = 1$

why? because translating **x**, **p**, **q** by **v** will not result in a valid equality if $\alpha + \beta \neq 1$

equations: explicit

form: $y = f(x)$

examples:

$$y = mx + c$$
$$y = x^2$$
$$y = \sin(x)$$

problems: axis-dependent, ill-defined slopes, difficult to represent bounded surfaces

equations: implicit

form: $0 = f(x, y)$

examples:

$$x^2 + y^2 - r^2 = 0$$
$$ax^2 + 2hxy + by^2 + 2gx + 2fy + c = 0$$

problems: difficult to generate points directly

equations: parametric

form:

$$x = x(u) = f_1(u)$$
$$y = y(u) = f_2(u)$$
$$z = z(u) = f_3(u)$$

examples:

$x(t) = \cos(t), y(t) = \sin(t)$ circle

$x(t) = x_1 + t(x_2 - x_1)$
$y(t) = y_1 + t(y_2 - y_1)$ line through (x_1,y_1) and (x_2,y_2)

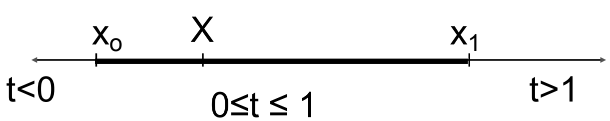
interpolation

the process of computing a curve/surface that includes a given set of points

[vs. approximation: computing a curve that remains "near" a set of points]

linear interpolation

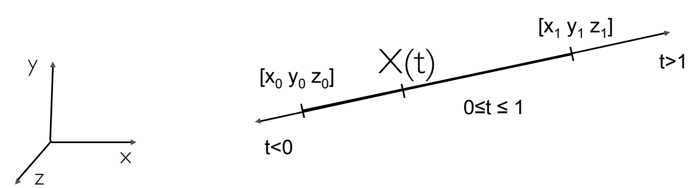
$$X(t) = (1\text{-}t)\cdot x_0 + t\cdot x_1$$



linear interpolation

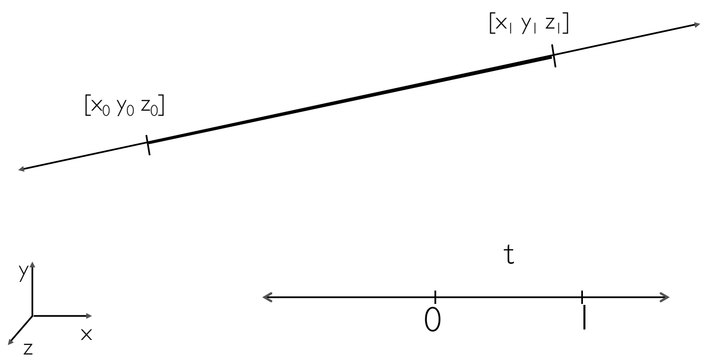
$$X(t) = (1-t)x_o + tx_1$$

$$X(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = (1-t)\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$



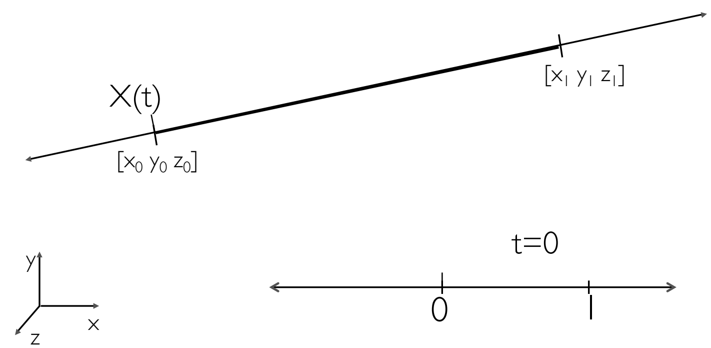
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



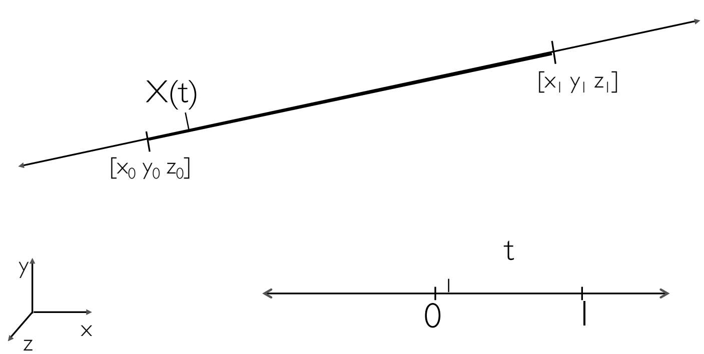
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



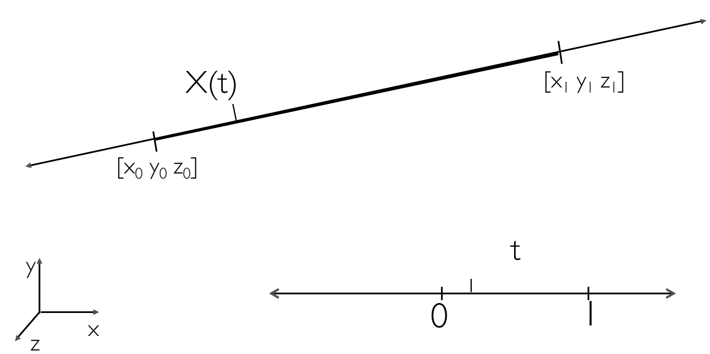
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



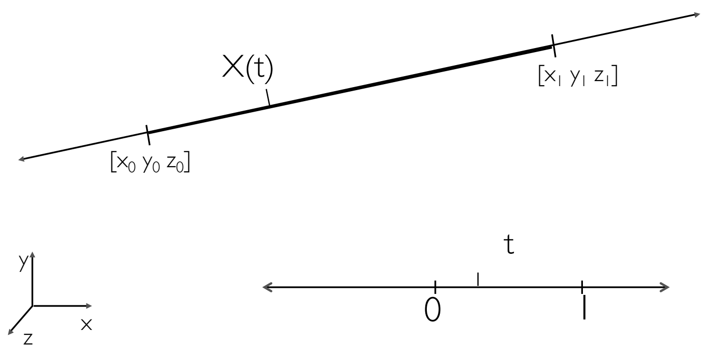
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



linear interpolation

$$X(t) = (1-t)x_o + tx_1$$

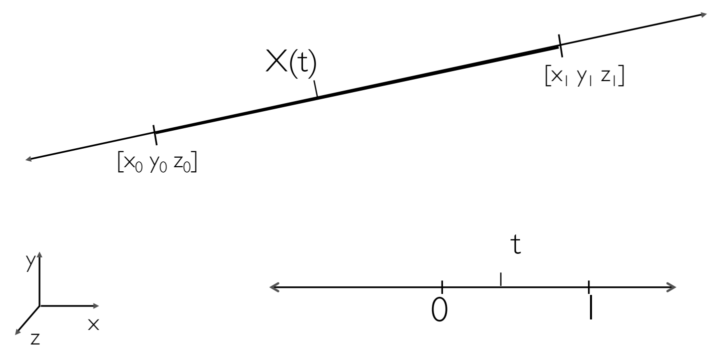


85

85

linear interpolation

$$X(t) = (1-t)x_o + tx_1$$

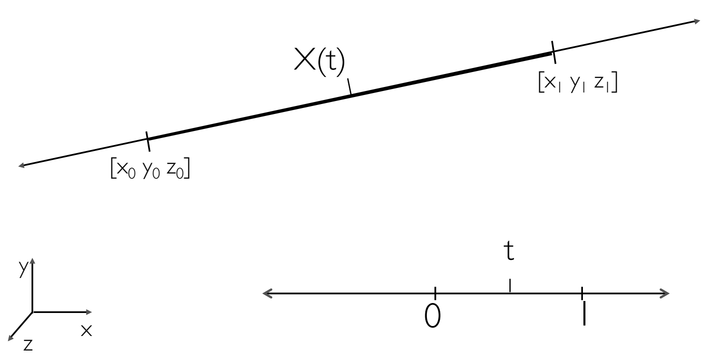


86

86

linear interpolation

$$X(t) = (1-t)x_o + tx_1$$

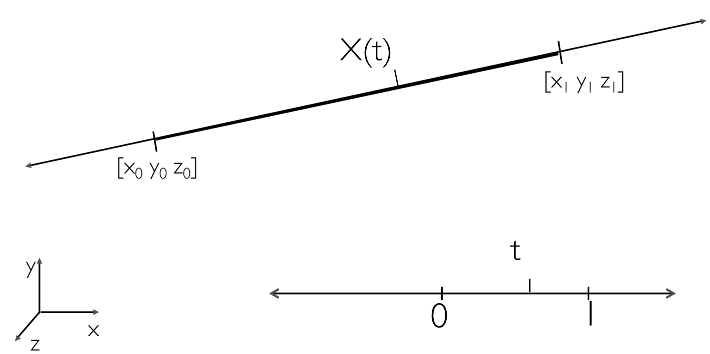


87

87

linear interpolation

$$X(t) = (1-t)x_o + tx_1$$

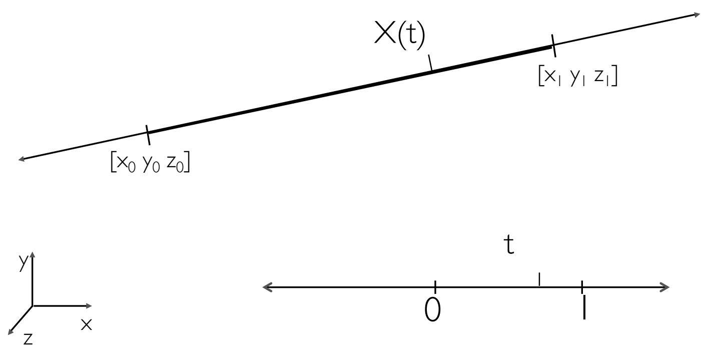


88

88

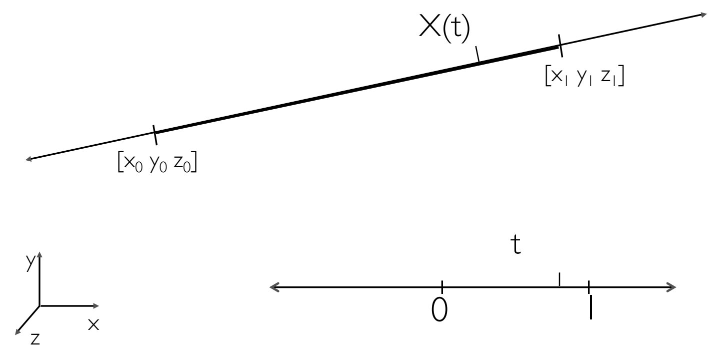
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



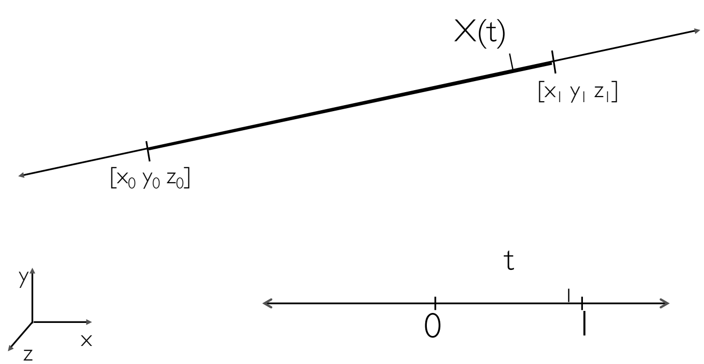
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



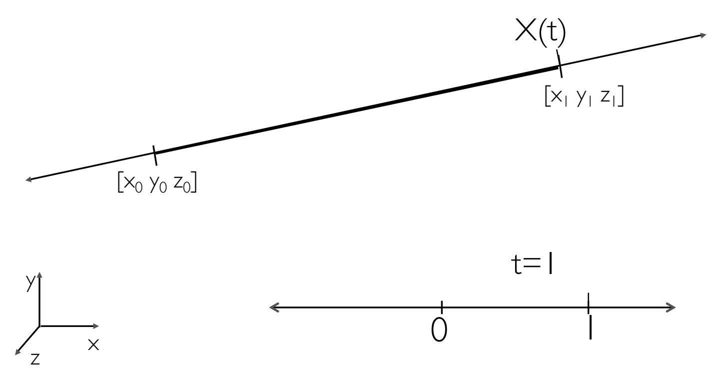
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



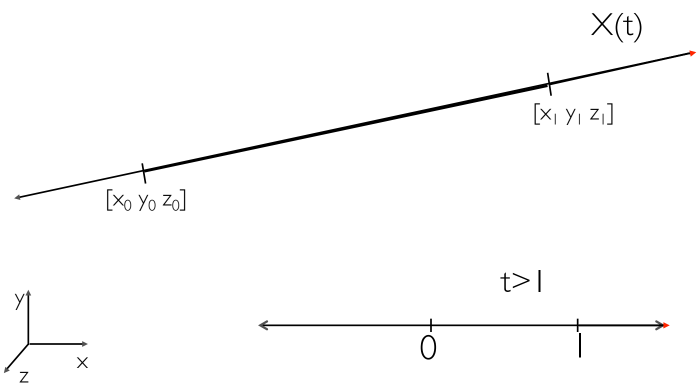
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



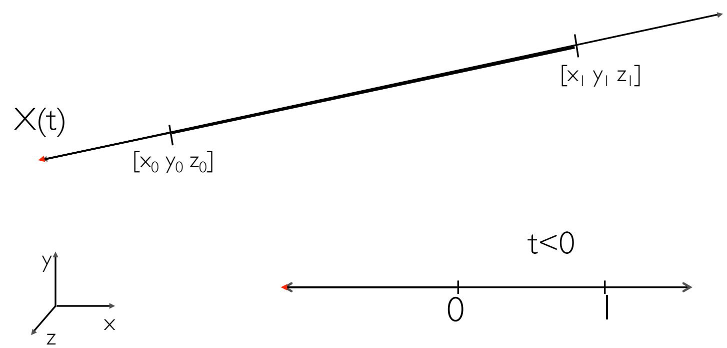
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



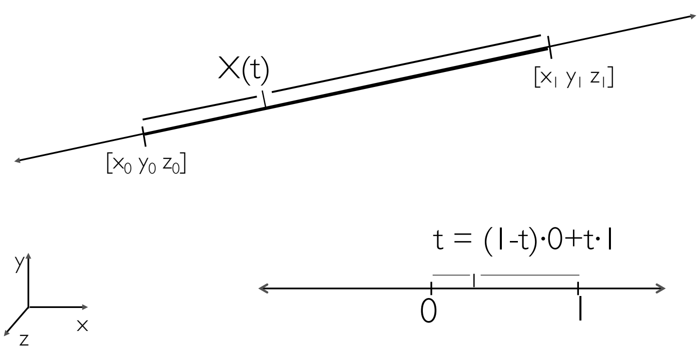
linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



linear interpolation

$$X(t) = (1-t)x_o + tx_1$$



some notes on light & vision

dual nature of light

"wave-particle duality": light can behave as:

a wave - as evidenced by the double-slit experiment

a particle - as evidenced by the photoelectric effect

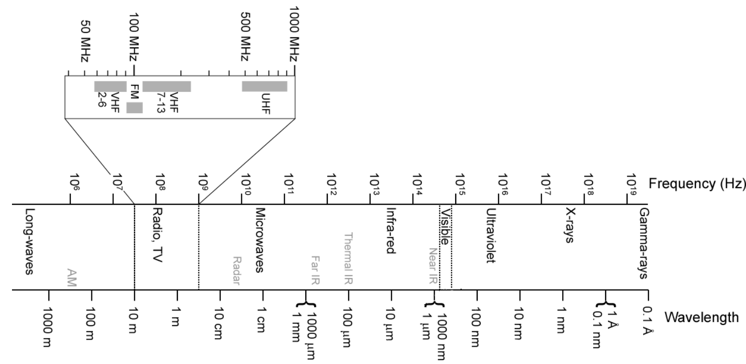
in this class, we will almost always be discussing light in terms of geometric optics, which models light as a particle (i.e. photons).

but to discuss color we must use the wave properties.

97

97

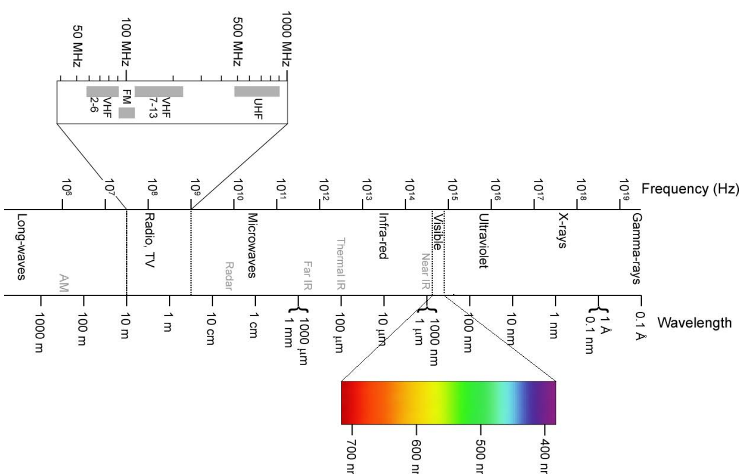
the electromagnetic spectrum



98

98

the visible spectrum

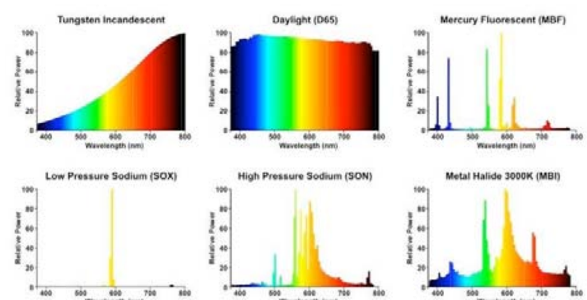


99

99

spectral power distribution (SPD)

energy emitters (and reflectors) can be characterized by their SPD: a graph of "intensity" vs. frequency



note: "intensity" is really a vague term - here we mean radiant exitance (radiant flux per area).

100

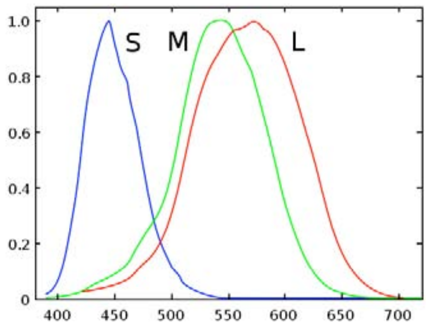
100

tristimulus theory of color vision

the human eye has 3 types of color receptors (*cones*)

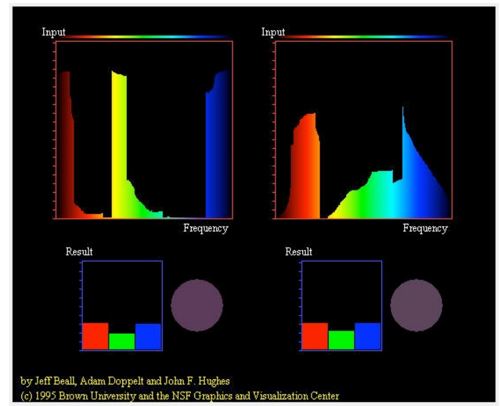
incident light is reduced to a function of 3 signals by the eye

the response is a function of wavelength for each type of receptor



metamers

metamers are two emitters (or reflectors) with different SPDs, but which appear the same (i.e. perceptually).



Q: how many component colors do we use?

for monitors, 3: RGB space (for manufacturing simplicity)

more general devices may have higher dimensions (e.g. CMYK space)

all are approximations to the CIE XYZ color space (which defines the range of human color perception)

[note: a *color spaces* describes the entire range of wavelengths possible with a specific representation]

so what 3 colors do we use?

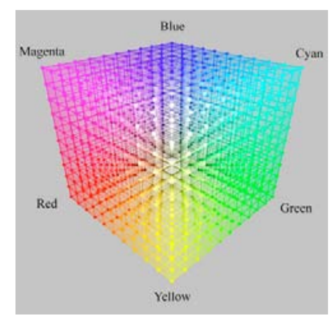
RGB uses additive color mixing of red, green & blue:

red + green = yellow

green + blue = cyan

blue + red = magenta

red + green + blue = white



if we also allow intensity of each component to be controlled, we can go from black to white, and the corresponding shades of colors as well.

encoding RGB

colors are encoded as triples: [r g b]

we often illustrate colors though normalized encoding: each element is in [0 1]

e.g.

black = [0 0 0]

white = [1 1 1]

red = [1 0 0]

green = [0 1 0]

blue = [0 0 1]

yellow = [1 1 0]

magenta = [1 0 1]

cyan = [0 1 1]

gray = [***a a a***] (for any ***a*** on [0 1])

blue = [0 0 1]

in implementation, each channel will use the entire range of values of its data type

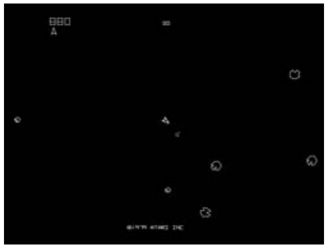
images

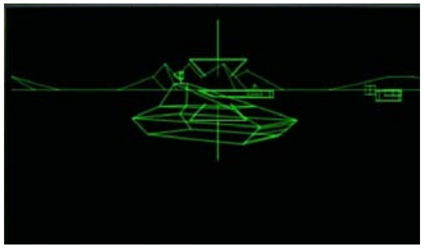
what is an image?

an image is a 2D representation of energy (usually light, sometimes sound)

for modern devices - <u>raster devices</u> - an image is a rectangular set of samples of a continuous 2D function

before raster devices: vector devices!





advantages: extremely high resolution

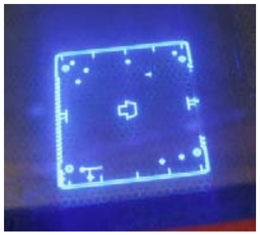
disadvantages: slow, unreliable, hard to do color

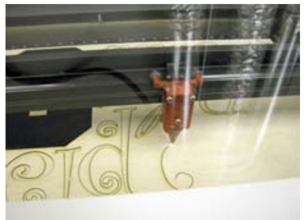
(we still have some vector devices around)

pen plotters

laser cutters

rapid prototyping machines





rapid prototyping machine drawing a part

laser cutters cutting metal (top) and engraving wood

raster devices

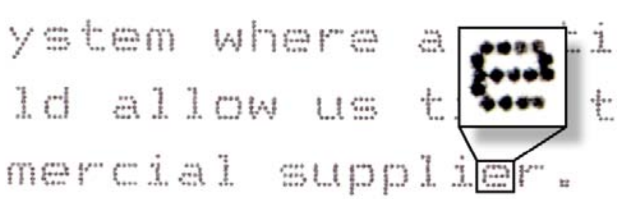
raster devices use arrays of pixels or sensors to display or acquire colors

n.b. raster devices are <u>resolution-dependent</u>, measured in one of:

dots-per-inch (dpi) - usually for print processes

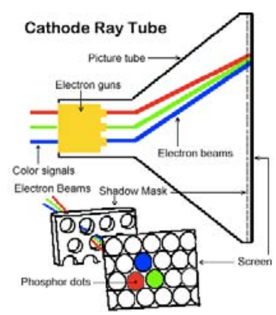
pixels-per-inch (ppi) - usually for monitors

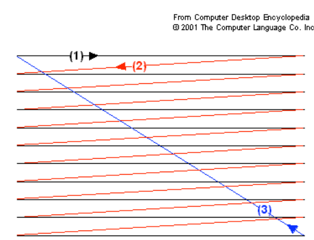
raster hardcopy devices



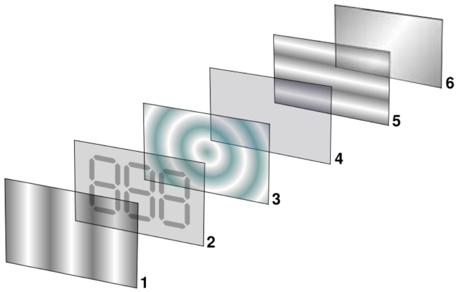
both dot matrix and inkjet printers are raster devices

raster devices - CRT



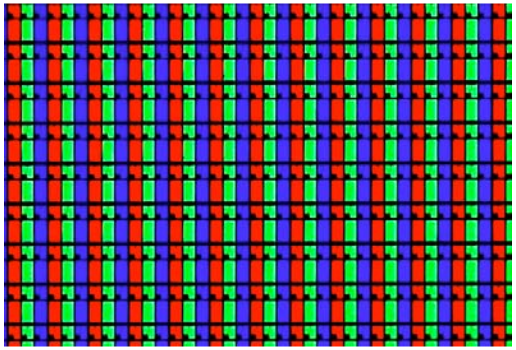


raster devices - LCD display



layers in an LCD display: 1,5: polarizing filters. 2, 4 electrode plates. 3: liquid crystal. 6: backlight

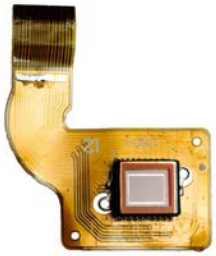
LCD display - up close

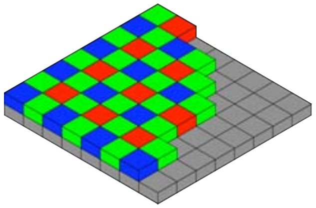


input devices

digital cameras, film & “flatbed” scanners

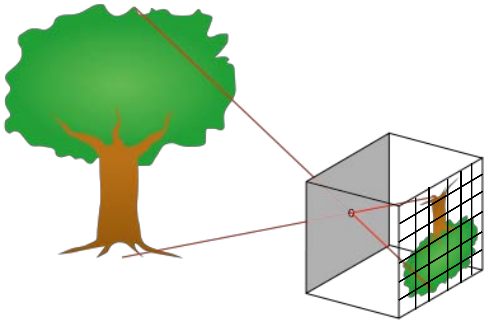
(all have resolutions <u>much</u> higher than displays)





camera CCD and color filter array (image is processed using *demosaicing)*

image formation



light on sensor as a 2D function

$\mathbf{I}(x, y): R^2 \rightarrow V$

raster image from I(x,y)

how do we go from continuous $\mathbf{I}$(x,y) to pixel value in a raster image?

camera or scanner pixel (i.e. input device):

measurement of the average "intensity" over some small area around the pixel

but: unless you know very specifically the device an image came from, all you can assume is:

a pixel is a point sample and image is an array of point samples

display pixels

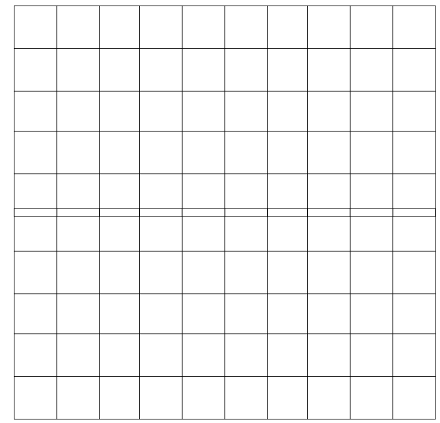
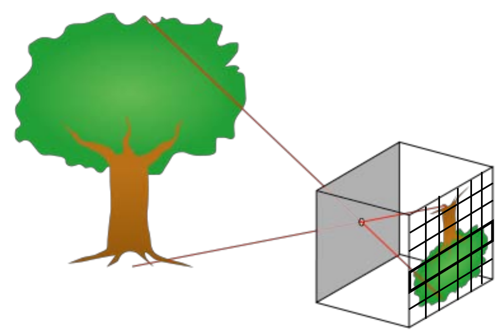


image pixels (samples)

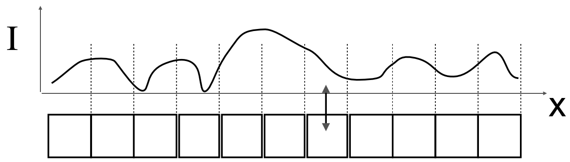
image formation



light on sensor as a 2D function

$\mathbf{I}(x, y) : R^2 \rightarrow V$

1d pixel array



each pixel gets average intensity for its interval of I(x)

image pixel values

date stored in pixel values:

- 1 value for grey scale, OR
- 3 values: 1 each for red, green and blue
- alpha : a separate value designating opacity

each of red/blue/green/alpha is called a **channel**

Q: what data type should each value be?

- depends on both the device and the available memory
- e.g. 10 megapixel camera, 32-bit data for each color = 115MB!

pixel data types and applications

1 bit : (b&w) - hires text displays (e.g. digital ink)

8 bit RBG (24 bits per pixel): web & email images

12-16 bit RBG (36-42 bits per pixel): highend digital cameras

16-bit greyscale (16 bits per pixel): medical imaging

16-bit floating-point RGB (48 bits per pixel): HDR images & rendering

32-bit floating-point RGB (96 bits per pixel): HDR images & rendering

what does color depth give us?

data type - bits & format **per channel**	description	# of possible colors per channel	# of possible colors overall
1	black & white	2	2
8 fixed precision	RGB (+A) “24-bit color”	256 per channel (0-255)	16 million
16 fixed precision	greyscale “16-bit greyscale”	65536 (0-65535)	65536
16 fixed precision	RGB (+A) “48-bit color”	65536 (0-65535)	281 trillion
16 floating point	HDR RGB (+A) “half precision”	65536 (6x10	281 trillion
32 floating point	HDR RGB (+A) +Z	4.3 trillion	!!!!!

dynamic range

dynamic range specifies the maximum contrast possible, and is a major factor in image quality

$R_d = I_{max} / I_{min}$ (i.e. maximum intensity divided by minimum intensity)

note:
natural illumination has a dynamic range of about 10^6

dynamic range

$R_d = I_{max} / I_{min}$ (i.e. maximum intensity divided by minimum intensity)

typical values:

- desktop display: 20:1 to 100:1 (best)
- photograph, mono print: 30:1
- photograph, transparency: 1000:1
- high dynamic range (HDR) display: 10000:1

dynamic range

how many levels are needed?

rule of thumb: < 2% between adjacent intensity:

- min: 0;
- value 1: I_{min}
- value 2: $(1.02) \cdot I_{min}$
- value 3: $(1.02)^2 \cdot I_{min}$
- value 4: $(1.02)^3 \cdot I_{min}$
- etc.

this ends up being about 12 steps per unit of dynamic range

e.g. if we want a R_d of 20:

20 * 12 = 240 steps

(just enough in an 8-bit fixed-precision channel: 0-255)

dynamic range - integer vs. floating point

in 48-bit integer color, the dynamic range per channel is:

65535 / 1 = 65535

in “half” precision format, the dynamic range is:

$6.5 \times 10^4 \ / \ 6 \times 10^{-8} = 1 \times 10^{12}$

Q: how can the ranges be different when the # of possible colors is the same?

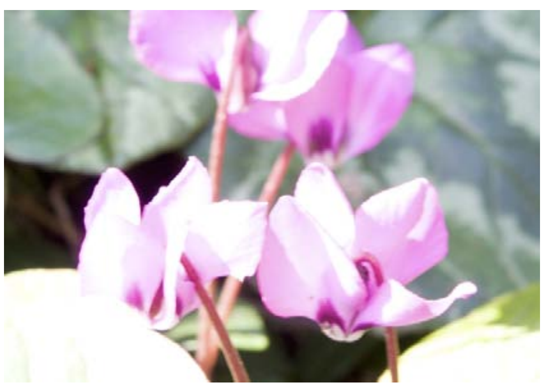
effect of reduced bit depth (i.e. not enough levels)

129

quantization artifact - “banding”

130

“clipping” - highlights all set to max



131

alpha

alpha is a fourth per-pixel channel, in addition to R, G, & B

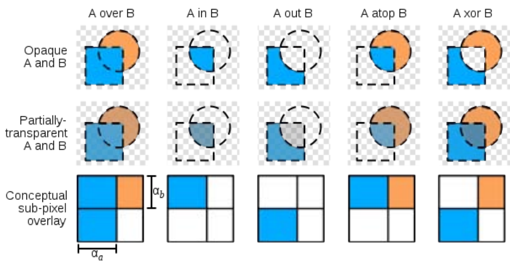
it allows one to specify transparency, useful when images are being composited

e.g. “over” operator for pixels:

final color = fg_color * α + bg_color * $(1-\alpha)$

132

alpha compositing



133

133

The OpenEXR Image Format

134

134

OpenEXR

open-source HDR format developed by ILM

supports 16-bit (“half”) & 32-bit floating point, and 32-bit integer pixels

multiple lossless compression algorithms

scan-line and tile- based methods

portable & extensible

compatible with graphics hardware

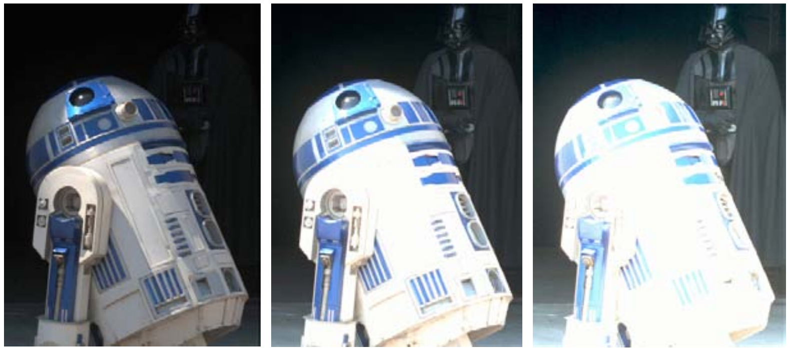
c++-based

see: www.openexr.com

135

135

OpenEXR example



136

136

OpenEXR image file interface

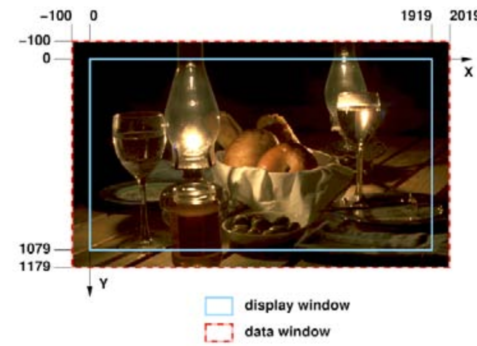
we will be using the “simple” OpenEXR interface

we will be constrained to using the “half” pixel type

we can only read/write RBGA files

we will only be addressing scanline-based images (though the simple interface supports tiling as well)

OpenEXR image coordinates



OpenEXR “half” pixels

```
struct Rgba
{
   half r; // red
   half g; // green
   half b; // blue
   half a; // alpha (opacity) (convention: 0 to 1)
};
```


the “half” format

16-bit floating-point format for each channel (R, G, B, A, etc)

65536 possible values for each channel

minimum positive value: $6x10^{-8}$

maximum positive value: $6.5x10^{4}$

dynamic range: $6.5x10^{4}$ / $6x10^{-8}$ = $1x10^{12}$

writing an image file

```
void
writeRgba (const char fileName[],
           const Rgba *pixels,
           int width,
           int height)
{
    //
    // Write an RGBA image using class RgbaOutputFile.
    //
    //  - open the file
    //  - describe the memory layout of the pixels
    //  - store the pixels in the file
    //


    RgbaOutputFile file (fileName, width, height, WRITE_RGBA);
    file.setFrameBuffer (pixels, 1, width);
    file.writePixels (height);
}
```

141

141

reading an image file

```
void
readRgba (const char fileName[],
          Array2D<Rgba> &pixels,
          int &width,
          int &height)
{
    //
    // Read an RGBA image using class RgbaInputFile:
    //
    //  - open the file
    //  - allocate memory for the pixels
    //  - describe the memory layout of the pixels
    //  - read the pixels from the file
    //

    RgbaInputFile file (fileName);
    Box2i dw = file.dataWindow();

    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;
    pixels.resizeErase (height, width);

    file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y * width, 1, width);
    file.readPixels (dw.min.y, dw.max.y);
}
```

142

142

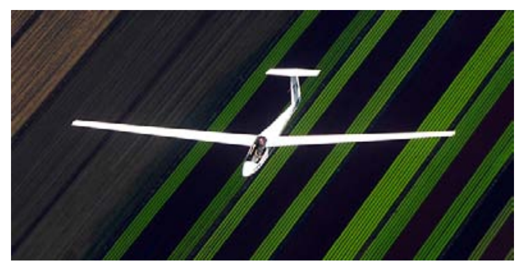
example main

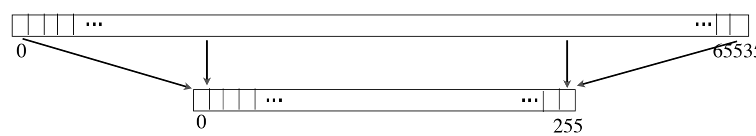
```
int
main (int argc, char *argv[])
{
    try
    {
        int w, h;
        Array2D<Rgba> p;

        readRgba ("myimage.exr", p, w, h);

        cout << "width height is " << w << "  " << h << endl;

        // convert to a red-only image:
        for (int y = 0; y < h; ++y)
        {
            for (int x = 0; x < w; ++x)
            {
                Rgba &px = p[y][x];   // get the pixel
                px.g = 0;  px.b = 0;  px.a = 1;  // note: px.r left alone!
            }
        }

        writeRgba ("myimage_red.exr", &p[0][0], w, h);
    }

    catch (const std::exception &exc)
    {
        std::cerr << exc.what() << std::endl;
        return 1;
    }

    return 0;
}
```

143

143

exposure & compression in EXR format: displaying “half” format on 8-bit RGB

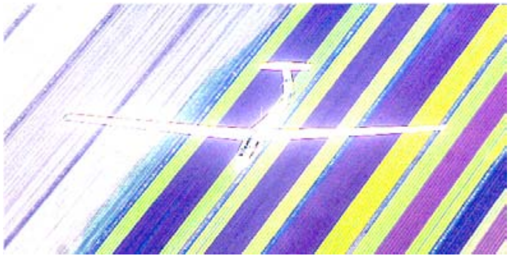


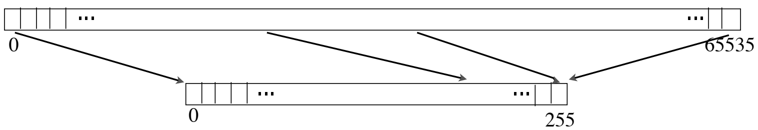


144

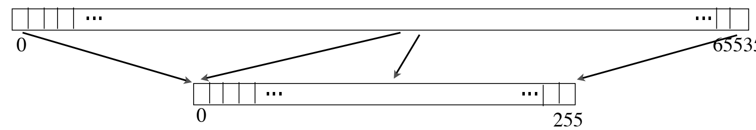
144

exposure & compression in EXR format: displaying “half” format on 8-bit RGB





exposure & compression in EXR format: displaying “half” format on 8-bit RGB



online resources

on the class resources page:

- openEXR technical intro
- reading & writing openEXR files

online:

- www.openexr.com

on CLIC machines:

- libraries, test programs, and test images

Ray Tracing

introduction

basic idea:

- trace the path light follows though a scene.
- model its material interactions.
- iterate. a lot.

149

introduction

strength of the algorithm:

- very intuitive solution to visible surface determination
- extends to many secondary phenomena (i.e. shadows, refraction, multiple lights, etc.)
- produces very realistic images

weaknesses:

- not good for some common secondary phenomena (diffuse inter-reflection)
- slow

150

ray traced example scene

151

ray traced example scene

152

ray traced example scene

153

ray traced example scene

154