



## Introduction to Graphics

Computer Science Tripos Part 1A/1B  
Michaelmas Term 2018/2019

Department of  
Computer Science  
and Technology  
The Computer Laboratory

William Gates Building  
15 JJ Thomson Avenue  
Cambridge  
CB3 0FD

[www.cst.cam.ac.uk](http://www.cst.cam.ac.uk)

---

This handout includes copies of the slides that will be used in lectures. These notes do not constitute a complete transcript of all the lectures and they are not a substitute for text books. They are intended to give a reasonable synopsis of the subjects discussed, but they give neither complete descriptions nor all the background material.

Material is copyright © Neil A Dodgson, Peter Robinson & Rafał Mantiuk, 1996-2018, except where otherwise noted.

All other copyright material is made available under the University's licence. All rights reserved.

---

## Introduction to Computer Graphics

**Peter Robinson & Rafał Mantiuk**

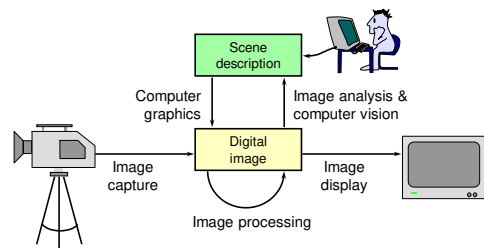
[www.cl.cam.ac.uk/~pr](http://www.cl.cam.ac.uk/~pr) & [~rkm38](http://~rkm38)

Eight lectures & two practical tasks for Part IA CST

Two supervisions suggested

Two exam questions on Paper 3

## What are Computer Graphics & Image Processing?



## Why bother with CG & IP?

- ✦ All visual computer output depends on CG
  - ◆ printed output (laser/ink jet/phototypesetter)
  - ◆ monitor (CRT/LCD/plasma/DMD)
  - ◆ all visual computer output consists of real images generated by the computer from some internal digital image
- ✦ Much other visual imagery depends on CG & IP
  - ◆ TV & movie special effects & post-production
  - ◆ most books, magazines, catalogues, brochures, junk mail, newspapers, packaging, posters, flyers



## Course Structure

- ✦ **Background**
  - ◆ What is an image? Human vision. Resolution and quantisation. Storage of images in memory. [1 lecture]
- ✦ **Rendering**
  - ◆ Perspective. Reflection of light from surfaces and shading. Geometric models. Ray tracing. [2 lectures]
- ✦ **Graphics pipeline**
  - ◆ Polygonal mesh models. Transformations using matrices in 2D and 3D. Homogeneous coordinates. Projection: orthographic and perspective. Rasterisation. [2 lectures]
- ✦ **Graphics hardware and modern OpenGL**
  - ◆ GPU APIs. Vertex processing. Fragment processing. Working with meshes and textures. [1 lectures]
- ✦ **Colour and tone mapping**
  - ◆ Colour perception. Colour spaces. Tone mapping [2 lectures]

## Course books

- ✦ *Fundamentals of Computer Graphics*
  - ◆ Shirley & Marschner
  - CRC Press 2015 (4<sup>th</sup> edition)
- ✦ *Computer Graphics: Principles & Practice*
  - ◆ Hughes, van Dam, McGuire, Sklar et al.
  - Addison-Wesley 2013 (3<sup>rd</sup> edition)
- ✦ *OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 4.5 with SPIR-V*
  - ◆ Kessenich, Sellers & Shreiner
  - Addison Wesley 2016 (7<sup>th</sup> edition and later)




## Computer Graphics & Image Processing

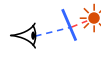
- ✦ **Background**
  - ◆ What is an image?
  - ◆ Human vision
  - ◆ Resolution and quantisation
  - ◆ Storage of images in memory
- ✦ **Rendering**
- ✦ **Graphics pipeline**
- ✦ **Rasterization**
- ✦ **Graphics hardware and modern OpenGL**
- ✦ **Colour and tone mapping**

### What is required for vision?


- ✦ illumination
  - some source of light
- ✦ objects
  - which reflect (or transmit) the light
- ✦ eyes
  - to capture the light as an image



direct viewing

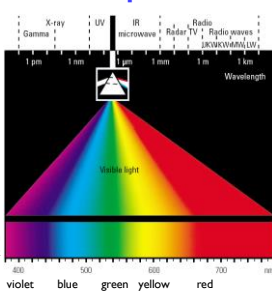


transmission



reflection

### The spectrum



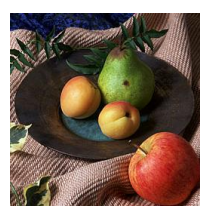
visible light is only a tiny part of the whole electromagnetic spectrum

the short wavelength end of the spectrum is violet

the long wavelength end of the spectrum is red

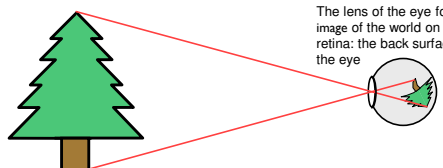
### What is an image?

- ✦ two dimensional function
- ✦ value at any point is an intensity or colour
- ✦ not digital!



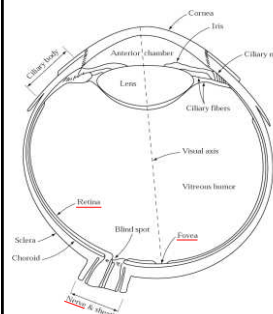
### The workings of the human visual system

- ✦ to understand the requirements of displays (resolution, quantisation and colour) we need to know how the human eye works...



The lens of the eye forms an image of the world on the retina: the back surface of the eye

### Structure of the human eye



- ✦ the **retina** is an array of light detection cells
- ✦ the **fovea** is the high resolution area of the retina
- ✦ the **optic nerve** takes signals from the retina to the visual cortex in the brain

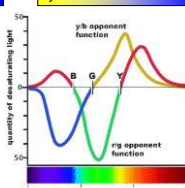
Fig. 2.1 from Gonzalez & Woods

### Light detectors in the retina

- ✦ two classes
  - ◆ rods
  - ◆ cones
- ✦ cones come in three types
  - ◆ sensitive to **short**, **medium** and **long** wavelengths
  - ◆ allow you to see in colour
- ✦ the cones are concentrated in the macula, at the centre of the retina
- ✦ the fovea is a densely packed region in the centre of the macula
  - ◆ contains the highest density of cones
  - ◆ provides the highest resolution vision


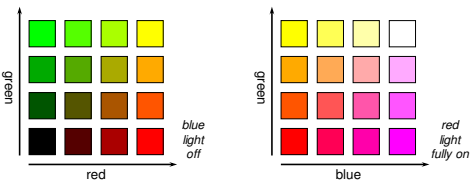
### 13 Colour signals sent to the brain

- ◆ the signal that is sent to the brain is pre-processed by the retina
  - long + medium + short = luminance
  - long - medium = red-green
  - long + medium - short = yellow-blue
- ◆ this theory explains:
  - colour-blindness effects
  - why red, yellow, green and blue are perceptually important colours
  - why you can see e.g. a yellowish red but not a greenish red



### 14 Mixing coloured lights

- ◆ by mixing different amounts of red, green, and blue lights we can generate a wide range of responses in the human eye

- ◆ not all colours can be created in this way

### 15 What is a digital image?

- ◆ a contradiction in terms
  - ◆ if you can see it, it's not digital
  - ◆ if it's digital, it's just a collection of numbers
- ◆ a sampled and quantised version of a real image
- ◆ a rectangular array of intensity or colour values

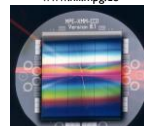
### 16 Sampling

- ◆ a digital image is a rectangular array of intensity values
- ◆ each value is called a *pixel*
  - ◆ "picture element"
- ◆ sampling resolution is normally measured in pixels per inch (ppi) or dots per inch (dpi)
  - ◆ computer monitors have a resolution around 100-200 ppi
  - ◆ laser and ink jet printers have resolutions between 300 and 1200 ppi
  - ◆ typesetters have resolutions between 1000 and 3000 ppi


### 17 Image capture

- ◆ a variety 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


area CCD  
[www.hil.mpg.de](http://www.hil.mpg.de)



flatbed scanner  
[www.nuggetlab.com](http://www.nuggetlab.com)

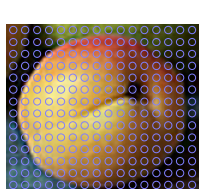



Heidelberg drum scanner



The image of the Heidelberg drum scanner and many other images in this section come from "Handbook of Print Media", by Helmut Kipphan, Springer-Verlag, 2001

### 18 Image capture example

A real image

A digital image

19

### Sampling resolution

256×256	128×128	64×64	32×32
2×2	4×4	8×8	16×16

20

### Quantisation

- ✦ each intensity value is a number
- ✦ for digital storage the intensity values must be quantised
  - limits the number of different intensities that can be stored
  - limits the brightest intensity that can be stored
- ✦ how many intensity levels are needed for human consumption
  - 8 bits often sufficient
  - some applications use 10 or 12 or 16 bits
  - more detail later in the course
- ✦ colour is stored as a set of numbers
  - usually as 3 numbers of 5–16 bits each
  - more detail later in the course

21

### Quantisation levels

8 bits (256 levels)	7 bits (128 levels)	6 bits (64 levels)	5 bits (32 levels)
1 bit (2 levels)	2 bits (4 levels)	3 bits (8 levels)	4 bits (16 levels)

22

### Storing images in memory

- ✦ 8 bits became a *de facto* standard for greyscale images
  - ◆ 8 bits = 1 byte
  - ◆ 16 bits is now being used more widely, 16 bits = 2 bytes
  - ◆ an 8 bit image of size  $W \times H$  can be stored in a block of  $W \times H$  bytes
  - ◆ one way to do this is to store `pixel[x][y]` at memory location  $base + x + W \times y$ 
    - memory is 1D, images are 2D

base

$base + 1 + 5 \times 2$

23

### Colour images

- ◆ tend to be 24 bits per pixel
  - 3 bytes: one red, one green, one blue
  - increasing use of 48 bits per pixel, 2 bytes per colour plane
- ◆ can be stored as a contiguous block of memory
  - of size  $W \times H \times 3$
- ◆ more common to store each colour in a separate "plane"
  - each plane contains just  $W \times H$  values
- ◆ the idea of planes can be extended to other attributes associated with each pixel
  - alpha plane (transparency), z-buffer (depth value), A-buffer (pointer to a data structure containing depth and coverage information), overlay planes (e.g. for displaying pop-up menus) — see later in the course for details

24

### The frame buffer

- ✦ most computers have a special piece of memory reserved for storage of the current image being displayed

B  
U  
S

- ✦ the frame buffer normally consists of dual-ported Dynamic RAM (DRAM)
  - ◆ sometimes referred to as Video RAM (VRAM)

25

### Computer Graphics & Image Processing

- + Background
- + **Rendering**
  - ◆ Perspective
  - ◆ Reflection of light from surfaces and shading
  - ◆ Geometric models
  - ◆ Ray tracing
- + Graphics pipeline
- + Graphics hardware and modern OpenGL
- + Technology

26

### Depth cues

27

### Rendering depth

28

### Perspective in photographs

Gates Building – the rounded version (Stanford)

Gates Building – the rectilinear version (Cambridge)

29

### Early perspective

- + Presentation at the Temple
- + Ambrogio Lorenzetti 1342
- + Uffizi Gallery Florence

30

### Wrong perspective

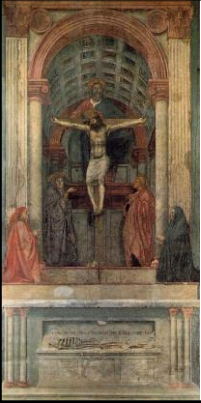
- + Adoring saints
- + Lorenzo Monaco 1407-09
- + National Gallery London



31

### Renaissance perspective

- ✦ Geometrical perspective  
Filippo Brunelleschi 1413
- ✦ Holy Trinity fresco
- ✦ Masaccio (Tommaso di Ser Giovanni di Simone) 1425
- ✦ Santa Maria Novella  
Florence
- ✦ *De pictura* (On painting)  
textbook by Leon Battista Alberti  
1435



32


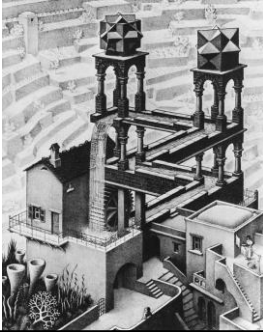
### More perspective

- ✦ The Annunciation with Saint Emidius
- ✦ Carlo Crivelli 1486
- ✦ National Gallery London



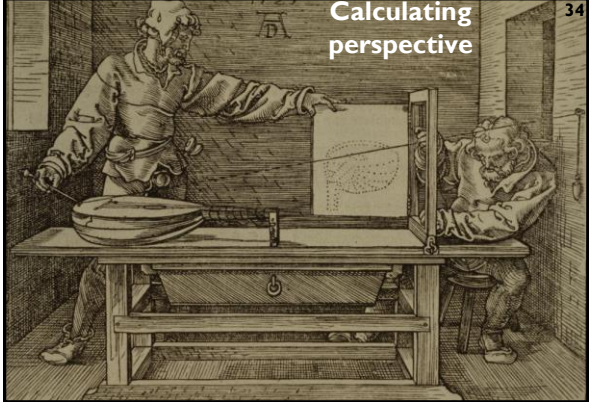
33

### False perspective

34

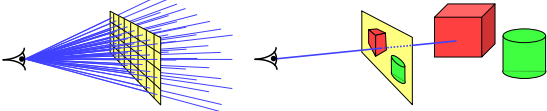
### Calculating perspective



35

### Ray tracing

- ✦ Identify point on surface and calculate illumination
- ✦ Given a set of 3D objects, shoot a ray from the eye through the centre of every pixel and see what surfaces it hits




shoot a ray through each pixel

whatever the ray hits determines the colour of that pixel

36

### Ray tracing: examples



ray tracing easily handles reflection, refraction, shadows and blur

ray tracing is computationally expensive

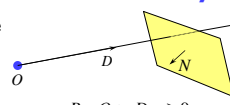


### Ray tracing algorithm 37

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 ;

### Intersection of a ray with an object 1 38

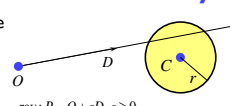
- ◆ plane
 

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

$$\text{plane: } P \cdot N + d = 0$$

$$s = -\frac{d + N \cdot O}{N \cdot D}$$
- ◆ polygon or disc
  - intersection the ray with the plane of the polygon
  - as above
  - then check to see whether the intersection point lies inside the polygon
  - a 2D geometry problem (which is simple for a disc)

### Intersection of a ray with an object 2 39

- ◆ sphere
 

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

$$\text{sphere: } (P - C) \cdot (P - C) - r^2 = 0$$

$$a = D \cdot D$$

$$b = 2D \cdot (O - C)$$

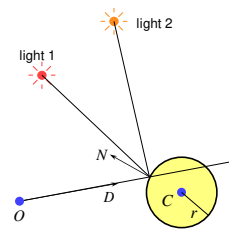
$$c = (O - C) \cdot (O - C) - r^2$$

$$d = \sqrt{b^2 - 4ac}$$

$$s_1 = \frac{-b + d}{2a}$$

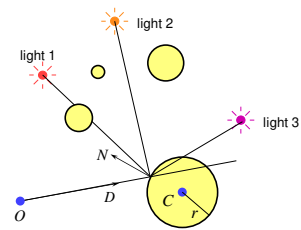
$$s_2 = \frac{-b - d}{2a}$$
- ◆ cylinder, cone, torus
  - all similar to sphere
  - try them as an exercise

### Ray tracing: shading 40



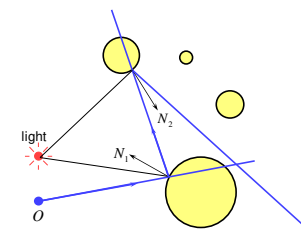
- ◆ once you have the intersection of a ray with the nearest object you can also:
  - calculate the normal to the object at that intersection point
  - shoot rays from that point to all of the light sources, and calculate the diffuse and specular reflections off the object at that point
  - this (plus ambient illumination) gives the colour of the object (at that point)

### Ray tracing: shadows 41



- ◆ because you are tracing rays 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
  - also need to watch for self-shadowing

### Ray tracing: reflection 42



- ◆ 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
  - this is perfect (mirror) reflection

### Ray tracing: transparency & refraction

- ◆ objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- ◆ transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- ◆ transparency + reflection means that a ray can split into two parts

### Illumination and shading

- ✦ Dürer's method allows us to calculate what part of the scene is visible in any pixel
- ✦ But what colour should it be?
- ✦ Depends on:
  - ◆ lighting
  - ◆ shadows
  - ◆ properties of surface material

### How do surfaces reflect light?

perfect specular reflection (mirror)      imperfect specular reflection      diffuse reflection (Lambertian reflection)

the surface of a specular reflector is faceted, each facet reflects perfectly but in a slightly different direction to the other facets

Johann Lambert, 18<sup>th</sup> century German mathematician

### Comments on reflection

- ◆ the surface can absorb some wavelengths of light
  - e.g. shiny gold or shiny copper
- ◆ specular reflection has "interesting" properties at glancing angles owing to occlusion of micro-facets by one another

- ◆ plastics are good examples of surfaces with:
  - specular reflection in the light's colour
  - diffuse reflection in the plastic's colour

### Calculating the shading of a surface

- ◆ gross assumptions:
  - there is only diffuse (Lambertian) reflection
  - all light falling on a surface comes directly from a light source
    - there is no interaction between objects
  - no object casts shadows on any other
    - so can treat each surface as if it were the only object in the scene
  - light sources are considered to be infinitely distant from the object
    - the vector to the light is the same across the whole surface
- ◆ observation:
  - the colour of a flat surface will be uniform across it, dependent only on the colour & position of the object and the colour & position of the light sources

### Diffuse shading calculation

$$\begin{aligned}
 I &= I_l k_d \cos \theta \\
 &= I_l k_d (N \cdot L)
 \end{aligned}$$

use this equation to calculate the colour of a pixel

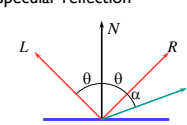
- $L$  is a normalised vector pointing in the direction of the light source
- $N$  is the normal to the surface
- $I_l$  is the intensity of the light source
- $k_d$  is the proportion of light which is diffusely reflected by the surface
- $I$  is the intensity of the light reflected by the surface

### Diffuse shading: comments 49

- ◆ can have different  $I_i$  and different  $k_d$  for different wavelengths (colours)
- ◆ watch out for  $\cos\theta < 0$ 
  - implies that the light is behind the polygon and so it cannot illuminate this side of the polygon
- ◆ do you use one-sided or two-sided surfaces?
  - one sided: only the side in the direction of the normal vector can be illuminated
    - if  $\cos\theta < 0$  then both sides are black
  - two sided: the sign of  $\cos\theta$  determines which side of the polygon is illuminated
    - need to invert the sign of the intensity for the back side
- ◆ this is essentially a simple one-parameter ( $\theta$ ) BRDF

### Specular reflection 50


◆ Phong developed an easy-to-calculate approximation to specular reflection



$$I = I_i k_s \cos^n \alpha$$

$$= I_i k_s (R \cdot V)^n$$

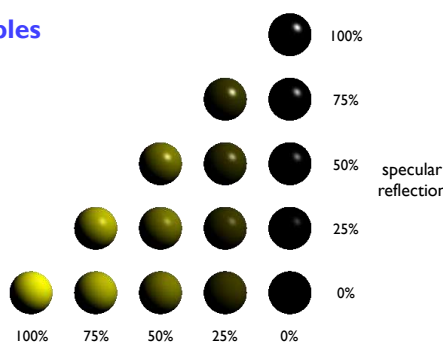
$L$  is a normalised vector pointing in the direction of the light source  
 $R$  is the vector of perfect reflection  
 $N$  is the normal to the surface  
 $V$  is a normalised vector pointing at the viewer  
 $I_i$  is the intensity of the light source  
 $k_s$  is the proportion of light which is specularly reflected by the surface  
 $n$  is Phong's *ad hoc* "roughness" coefficient  
 $I$  is the intensity of the specularly reflected light



$n=1$     $n=3$     $n=7$     $n=20$     $n=40$

Phong Bui-Tuong, "Illumination for computer generated pictures", *CACM*, 18(6), 1975, 311-7

### Examples 51

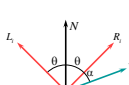


diffuse reflection                      specular reflection

### Shading: overall equation 52

◆ the overall shading equation can thus be considered to be the ambient illumination plus the diffuse and specular reflections from each light source

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



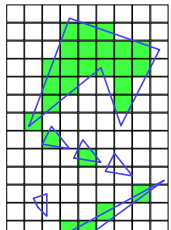
■ the more lights there are in the scene, the longer this calculation will take

### The gross assumptions revisited 53

- ◆ diffuse reflection
- ◆ approximate specular reflection
- ◆ no shadows
  - need to do ray tracing or shadow mapping to get shadows
- ◆ lights at infinity
  - can add local lights at the expense of more calculation
    - need to interpolate the  $L$  vector
- ◆ no interaction between surfaces
  - cheat!
    - assume that all light reflected off all other surfaces onto a given surface can be amalgamated into a single constant term: "ambient illumination", add this onto the diffuse and specular illumination

### Sampling 54

- ◆ we have assumed so far that each ray passes through the centre of a pixel
  - i.e. the value for each pixel is the colour of the object which happens to lie exactly under the centre of the pixel
- ◆ this leads to:
  - stair step (jagged) edges to objects
  - small objects being missed completely
  - thin objects being missed completely or split into small pieces

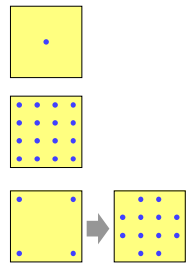


### Anti-aliasing 55

- ◆ these artefacts (and others) are jointly known as aliasing
- ◆ methods of ameliorating 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
  - in computer graphics its meaning has expanded to include most undesirable effects that can occur in the image
    - this is because the same anti-aliasing techniques which ameliorate true aliasing artefacts also ameliorate most of the other artefacts

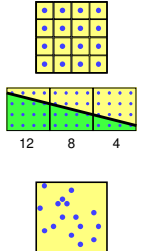
### Sampling in ray tracing 56

- ◆ single point
  - shoot a single ray through the pixel's centre
- ◆ super-sampling for anti-aliasing
  - shoot multiple rays through the pixel and average the result
  - regular grid, random, jittered, Poisson disc
- ◆ adaptive super-sampling
  - shoot a few rays through the pixel, check the variance of the resulting values, if similar enough stop, otherwise shoot some more rays



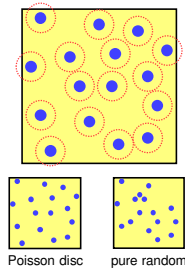
### Types of super-sampling I 57

- ◆ regular grid
  - divide the pixel into a number of sub-pixels and shoot a ray through the centre of each
  - problem: can still lead to noticable aliasing unless a very high resolution sub-pixel grid is used
- ◆ random
  - shoot  $N$  rays at random points in the pixel
  - replaces aliasing artefacts with noise artefacts
    - the eye is far less sensitive to noise than to aliasing



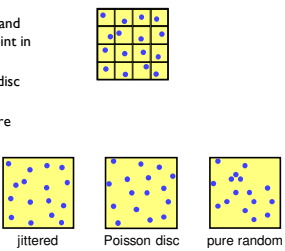
### Types of super-sampling 2 58

- ◆ Poisson disc
  - shoot  $N$  rays at random points in the pixel with the proviso that no two rays shall pass through the pixel closer than  $\epsilon$  to one another
  - for  $N$  rays this produces a better looking image than pure random sampling
  - very hard to implement properly



### Types of super-sampling 3 59

- ◆ jittered
  - divide pixel into  $N$  sub-pixels and shoot one ray at a random point in each sub-pixel
  - an approximation to Poisson disc sampling
  - for  $N$  rays it is better than pure random sampling
  - easy to implement



### More reasons for wanting to take multiple samples per pixel 60

- ◆ super-sampling is only one reason why we might want to take multiple samples per pixel
- ◆ many effects can be achieved by distributing the multiple samples over some range
  - called *distributed ray tracing*
    - N.B. *distributed* means distributed over a range of values
- ◆ can work in two ways
  - each of the multiple rays shot through a pixel is allocated a random value from the relevant distribution(s)
    - all effects can be achieved this way with sufficient rays per pixel
  - each ray spawns multiple rays when it hits an object
    - this alternative can be used, for example, for area lights

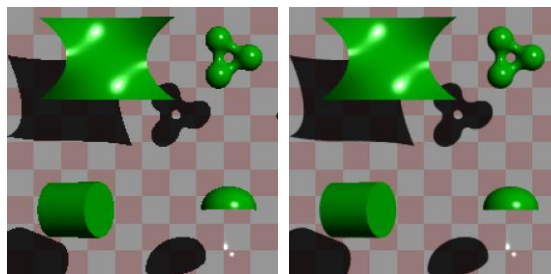
61

### Examples of distributed ray tracing

- distribute the samples for a pixel over the pixel area
  - get random (or jittered) super-sampling
  - used for anti-aliasing
- distribute the rays going to a light source over some area
  - allows area light sources in addition to point and directional light sources
  - produces soft shadows with penumbræ
- distribute the camera position over some area
  - allows simulation of a camera with a finite aperture lens
  - produces depth of field effects
- distribute the samples in time
  - produces motion blur effects on any moving objects

62

### Anti-aliasing



one sample per pixel

multiple samples per pixel

63

### Area vs point light source



an area light source produces soft shadows

a point light source produces hard shadows

64

### Finite aperture



left, a pinhole camera

below, a finite aperture camera

below left, 12 samples per pixel

below right, 120 samples per pixel

note the depth of field blur: only objects

at the correct distance are in focus



65

### Computer Graphics & Image Processing

- ✦ Background
- ✦ Rendering
- ✦ Graphics pipeline
  - ◆ Polygonal mesh models
  - ◆ Transformations using matrices in 2D and 3D
  - ◆ Homogeneous coordinates
  - ◆ Projection: orthographic and perspective
- ✦ Rasterization
- ✦ Graphics hardware and modern OpenGL
- ✦ Colour and tone mapping

66

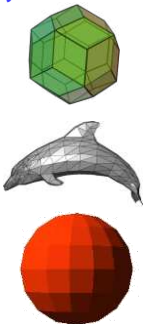
### Unfortunately...

- ✦ Ray tracing is computationally expensive
  - ◆ used for super-high visual quality
- ✦ Video games and user interfaces need something faster
- ✦ Most real-time applications rely on **rasterization**
  - ◆ Model surfaces as polyhedra – meshes of polygons
  - ◆ Use composition to build scenes
  - ◆ Apply perspective transformation and project into plane of screen
  - ◆ Work out which surface was closest
  - ◆ Fill pixels with colour of nearest visible polygon
- ✦ Modern graphics cards have hardware to support this
- ✦ Ray tracing starts to appear in real-time rendering
  - ◆ The latest generation of GPUs offers accelerated ray-tracing
  - ◆ But it still not as efficient as rasterization

67

### Three-dimensional objects

- ◆ Polyhedral surfaces are made up from meshes of multiple connected polygons
  - open or closed
- ◆ Polygonal meshes
  - open or closed
  - manifold or non-manifold
- ◆ Curved surfaces
  - must be converted to polygons to be drawn

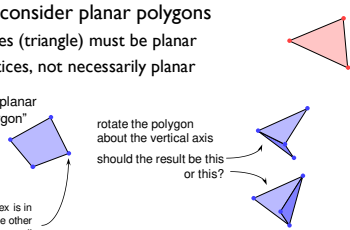


68

### Surfaces in 3D: polygons

- ✦ Easier to consider planar polygons
  - ◆ 3 vertices (triangle) must be planar
  - ◆ > 3 vertices, not necessarily planar

a non-planar "polygon" rotate the polygon about the vertical axis should the result be this or this?

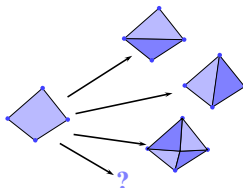


this vertex is in front of the other three, which are all in the same plane

69

### Splitting polygons into triangles

- ◆ Most Graphics Processing Units (GPUs) are optimised to draw triangles
- ◆ Split polygons with more than three vertices into triangles



which is preferable?

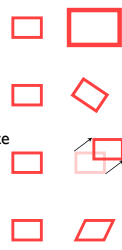
70

### 2D transformations

- ✦ scale
- ✦ rotate
- ✦ translate
- ✦ (shear)

✦ why?

- ◆ it is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size
- ◆ any reasonable graphics package will include transforms
  - 2D → Postscript
  - 3D → OpenGL



71

### Basic 2D transformations

- ◆ scale
  - about origin
  - by factor  $m$
$$x' = mx$$

$$y' = my$$
- ◆ rotate
  - about origin
  - by angle  $\theta$
$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$
- ◆ translate
  - along vector  $(x_o, y_o)$
$$x' = x + x_o$$

$$y' = y + y_o$$
- ◆ shear
  - parallel to  $x$  axis
  - by factor  $a$
$$x' = x + ay$$

$$y' = y$$

72

### Matrix representation of transformations

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



73

### Homogeneous 2D co-ordinates

- translations cannot be represented using simple 2D matrix multiplication on 2D vectors, so we switch to homogeneous co-ordinates
- $(x, y, w) \equiv \left(\frac{x}{w}, \frac{y}{w}\right)$
- an infinite number of homogeneous co-ordinates map to every 2D point
- $w=0$  represents a point at infinity
- usually take the inverse transform to be:  $(x, y) \equiv (x, y, 1)$

74

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

75

### Translation by matrix algebra

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

In homogeneous coordinates  
 $x' = x + wx_0$        $y' = y + wy_0$        $w' = w$

In conventional coordinates  
 $\frac{x'}{w'} = \frac{x}{w} + x_0$        $\frac{y'}{w'} = \frac{y}{w} + y_0$

76

### Concatenating transformations

- often necessary to perform more than one transformation on the same object
- can concatenate transformations by multiplying their matrices
- e.g. a shear followed by a scaling:

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

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

77

### Transformation are not commutative

be careful of the order in which you concatenate transformations

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

rotate then scale      scale

$$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

scale then rotate      rotate

78

### Scaling about an arbitrary point

- scale by a factor  $m$  about point  $(x_0, y_0)$
- translate point  $(x_0, y_0)$  to the origin
- scale by a factor  $m$  about the origin
- translate the origin to  $(x_0, y_0)$

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

$$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Exercise: show how to perform rotation about an arbitrary point

79

### 3D transformations

- 3D homogeneous co-ordinates  
 $(x, y, z, w) \rightarrow (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{1}{w})$
- 3D transformation matrices

<p>translation</p> $\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$	<p>identity</p> $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	<p>rotation about x-axis</p> $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
<p>scale</p> $\begin{bmatrix} m_x & 0 & 0 & 0 \\ 0 & m_y & 0 & 0 \\ 0 & 0 & m_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	<p>rotation about z-axis</p> $\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	<p>rotation about y-axis</p> $\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

80

### 3D transformations are not commutative

90° rotation about z-axis      90° rotation about x-axis

90° rotation about x-axis      90° rotation about z-axis

opposite faces

81

### Model transformation 1

- the graphics package Open Inventor defines a cylinder to be:
  - centre at the origin, (0,0,0)
  - radius 1 unit
  - height 2 units, aligned along the y-axis
- this is the only cylinder that can be drawn, but the package has a complete set of 3D transformations
- we want to draw a cylinder of:
  - radius 2 units
  - the centres of its two ends located at (1,2,3) and (2,4,5)
    - its length is thus 3 units
- what transforms are required? and in what order should they be applied?

82

### Model transformation 2

- order is important:
  - scale first
  - rotate
  - translate last
- scaling and translation are straightforward

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scale from size (2,2,2) to size (4,3,4)

$$T = \begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate centre of cylinder from (0,0,0) to halfway between (1,2,3) and (2,4,5)

83

### Model transformation 3

- rotation is a multi-step process
  - break the rotation into steps, each of which is rotation about a principal axis
  - work these out by taking the desired orientation back to the original axis-aligned position
- the centres of its two ends located at (1,2,3) and (2,4,5)
- desired axis:  $(2,4,5) - (1,2,3) = (1,2,2)$
- original axis: y-axis = (0,1,0)

84

### Model transformation 4

- 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

$$R_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

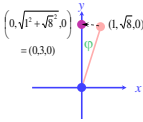
$$\theta = -\arcsin \frac{2}{\sqrt{2^2 + 2^2}}$$

### Model transformation 5

85

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

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

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


### Model transformation 6

86

the overall transformation is:

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

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

### Application: display multiple instances

87

- transformations allow you to define an object at one location and then place multiple instances in your scene

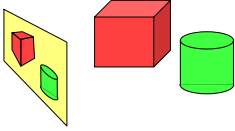


### 3D ⇔ 2D projection

88

to make a picture

- 3D world is projected to a 2D image
  - like a camera taking a photograph
  - the three dimensional world is projected onto a plane



The 3D world is described as a set of (mathematical) objects

e.g. sphere    radius (3.4)  
                  centre (0,2,9)

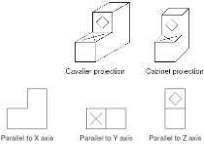
e.g. box        size (2,4,3)  
                  centre (7, 2, 9)  
                  orientation (27°, 156°)

### Types of projection

89

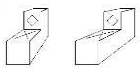
parallel

- e.g.  $(x, y, z) \rightarrow (x, y)$
- useful in CAD, architecture, etc
- looks unrealistic



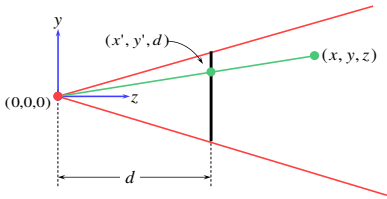
perspective

- e.g.  $(x, y, z) \rightarrow (\frac{x}{z}, \frac{y}{z})$
- things get smaller as they get farther away
- looks realistic
  - this is how cameras work



### Geometry of perspective projection

90



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

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

### Projection as a matrix operation

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

$x' = x \frac{d}{z}$   
 $y' = y \frac{d}{z}$   
 $z' = \frac{1}{z}$

remember  $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$

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

### Perspective projection with an arbitrary camera

- we have assumed that:
  - screen centre at  $(0,0,d)$
  - screen parallel to  $xy$ -plane
  - $z$ -axis into screen
  - $y$ -axis up and  $x$ -axis to the right
  - eye (camera) at origin  $(0,0,0)$
- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary point onto an arbitrary plane
  - transform all objects into our standard co-ordinate system (viewing co-ordinates) and use the above assumptions

### A variety of transformations

object in object co-ordinates  $\xrightarrow{\text{modelling transform}}$  object in world co-ordinates  $\xrightarrow{\text{viewing transform}}$  object in viewing co-ordinates  $\xrightarrow{\text{projection}}$  object in 2D screen co-ordinates

- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can be specified directly in viewing co-ordinates, or directly in world co-ordinates
- this is a useful set of transforms, not a hard and fast model of how things should be done

### Model, View, Projection matrices

Object coordinates  $\xrightarrow{\text{Model matrix}}$  World coordinates

To position each object in the scene. Could be different for each object.

### Model, View, Projection matrices

World coordinates  $\xrightarrow{\text{View matrix}}$  View (camera) coordinates

To position all objects relative to the camera

Camera at the origin, pointing at  $-z$

### Model, View, Projection matrices

View (camera) coordinates  $\xrightarrow{\text{Projection matrix}}$  Screen coordinates

To project 3D coordinates to a 2D plane. Note that  $z$  coordinate is retained for depth testing.

The default OpenGL coordinate system is right-handed

$x$  and  $y$  must be in the range  $-1$  and  $1$

### All together

Screen coordinates  
 $x_s/w_s$  and  $y_s/w_s$  must be between -1 and 1

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ w_s \end{bmatrix} = P \cdot V \cdot M \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

3D world vertex coordinates

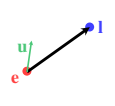
Projection, view and model matrices

97

### Viewing transformation 1

world co-ordinates  $\xrightarrow{\text{viewing transform}}$  viewing co-ordinates

- ★ 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  $\vec{el}$



98

### Viewing transformation 2

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

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

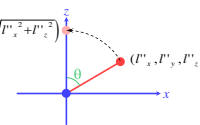
- ◆ scale so that eye point to look point distance,  $|\vec{el}|$ , is distance from origin to screen centre,  $d$

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

99

### Viewing transformation 3

- ◆ need to align line  $\vec{el}$  with z-axis
  - first transform  $e$  and  $l$  into new co-ordinate system  
 $e'' = S \times T \times e = 0 \quad l'' = S \times T \times l$
  - then rotate  $e''l''$  into  $yz$ -plane, rotating about  $y$ -axis

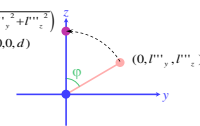
$$R_1 = \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \theta = \arccos \frac{l''_z}{\sqrt{l''_x{}^2 + l''_z{}^2}}$$


100

### Viewing transformation 4

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

$$l''' = R_1 \times l''$$

$$R_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \phi = \arccos \frac{l'''_z}{\sqrt{l'''_y{}^2 + l'''_z{}^2}}$$


101

### Viewing transformation 5

- ◆ 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

$$u'''' = R_2 \times R_1 \times u$$

$$R_3 = \begin{bmatrix} \cos \psi & -\sin \psi & 0 & 0 \\ \sin \psi & \cos \psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \psi = \arccos \frac{u''''_y}{\sqrt{u''''_x{}^2 + u''''_y{}^2}}$$

why don't we need to multiply  $u$  by  $S$  or  $T$ ?  
 $u$  is a vector rather than a point, vectors do not get translated  
  
 scaling  $u$  by a uniform scaling matrix would make no difference to the direction in which it points

102

103

### Viewing transformation 6

world  
co-ordinates

$\xrightarrow{\text{viewing transform}}$

viewing  
co-ordinates

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

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

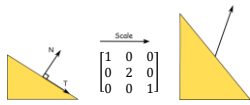
- ◆ in particular:  $\mathbf{e} \rightarrow (0,0,0)$   $\mathbf{1} \rightarrow (0,0,d)$
- ◆ the matrices depend only on e, l, and u, so they can be pre-multiplied together

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

104

### Transforming normal vectors

- + Transformation by a nonorthogonal matrix does not preserve angles



$\xrightarrow{\text{Scale}}$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- + Since:  $N \cdot T = 0$

Normal transform

$$N' \cdot T' = (GN) \cdot (MT) = 0$$

Transformed normal and tangent vector

Vertex position transform

- + We can find that:  $G = (M^{-1})^T$
- ◆ Derivation shown on the visualizer

104



### Introduction to Computer Graphics

- ✦ Background
- ✦ Rendering
- ✦ Graphics pipeline
- ✦ **Rasterization**
- ✦ Graphics hardware and modern OpenGL
- ✦ Colour

▶ 105

### Rasterization

- ▶ Efficiently draw (thousands of) triangles
  - ▶ Interpolate vertex attributes inside the triangle
- ▶ **Barycentric coordinates** are used to interpolate colours, normals, texture coordinates and other attributes inside the triangle

▶ 106

### Barycentric coordinates

- ▶ Find barycentric coordinates of the point (x,y)
- ▶ Given the coordinates of the vertices
- ▶ Derivation in the lecture

$$\alpha = \frac{f_{cb}(x,y)}{f_{cb}(x_a,y_a)} \quad \beta = \frac{f_{ac}(x,y)}{f_{ac}(x_b,y_b)}$$

$f_{ab}(x,y)$  is the implicit line equation:  
 $f_{ab}(x,y) = (y_b - y_a)x + (x_b - x_a)y + x_a y_b - x_b y_a$

▶ 107

### Triangle rasterization

```

for y=y_min to y_max do
  for x=x_min to x_max do
    alpha = f_cb(x,y)/f_cb(x_a,y_a)
    beta = f_ac(x,y)/f_ac(x_b,y_b)
    gamma = 1 - alpha - beta
    if (alpha > 0 and beta > 0 and gamma > 0) then
      c = alpha*a + beta*b + gamma*c
      draw pixels (x,y) with colour c
  
```

- ▶ Optimization: the barycentric coordinates will change by the same amount when moving one pixel right (or one pixel down) regardless of the position
- ▶ Precompute increments  $\Delta\alpha, \Delta\beta, \Delta\gamma$  and use them instead of computing barycentric coordinates when drawing pixels sequentially

▶ 108

### Illumination & shading

- ✦ Drawing polygons with uniform colours gives poor results
- ✦ Interpolate colours across polygons

### Gouraud shading

- ▶ for a polygonal model, calculate the diffuse illumination at each vertex
- ▶ calculate the normal at the vertex, and use this to calculate the diffuse illumination at that point
- ▶ normal can be calculated directly if the polygonal model was derived from a curved surface
- ▶ interpolate the colour between the vertices across the polygon
- ▶ surface will look smoothly curved
  - ▶ rather than looking like a set of polygons
  - ▶ surface outline will still look polygonal

▶ 110 Henri Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Trans Computers*, 20(6), 1971

### Flat vs Gouraud shading

- note how the interior is smoothly shaded but the outline remains polygonal

**Flat**                      **Gouraud**

<http://computer.howstuffworks.com/question484.htm>                      111

### Phong shading

- similar to Gouraud shading, but calculate the specular component in addition to the diffuse component
- therefore need to interpolate the *normal* across the polygon in order to be able to calculate the reflection vector
- N.B. Phong's approximation to specular reflection ignores (amongst other things) the effects of glancing incidence (the Fresnel term)

112

### Occlusions (hidden surfaces)

Simple case

More difficult cases

113

### Z-Buffer - algorithm

- Initialize the depth buffer and image buffer for all pixels  
 $colour(x, y) = Background\_colour,$   
 $depth(x, y) = z_{max}$  // position of the far clipping plane
- For every triangle in a scene do**
  - For every fragment (x, y) representing this triangle do**
    - Calculate z for current (x, y)
    - if** ( $z < depth(x, y)$ ) and ( $z > z_{min}$ ) **then**
      - $depth(x, y) = z$
      - $colour(x, y) = Polygon\_Colour(x, y)$

114

### View frustum

- Controlled by camera parameters: near-, far-clipping planes and field-of-view

115



### Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- Rasterization
- Graphics hardware and modern OpenGL**
  - GPU & APIs
  - OpenGL Rendering pipeline
  - Example OpenGL code
  - GLSL
  - Textures
  - Raster buffers
- Colour & Tone mapping

116


### What is a GPU?

- ▶ Graphics Processing Unit
- ▶ Like CPU (Central Processing Unit) but for processing graphics
- ▶ Optimized for floating point operations on large arrays of data
  - ▶ Vertices, normals, pixels, etc.

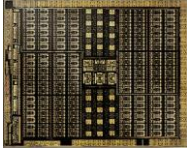
▶ 117

### CPU vs. GPU



AMD Epyc

32 CPU cores  
2.2-3.2 GHz  
14nm  
19,200,000,000 transistors



Nvidia Turing Quadro RTX 8000

4608 unified shaders  
288 texture mapping units  
96 render output units  
72 streaming multiprocessors  
576 tensor cores  
12nm  
18,600,000,000 transistors

▶ 118

### What does a GPU do

- ▶ Performs all low-level tasks & a lot of high-level tasks
  - ▶ Clipping, rasterisation, hidden surface removal, ...
    - ▶ Essentially draws millions of triangles very efficiently
  - ▶ Procedural shading, texturing, animation, simulation, ...
  - ▶ Video rendering, de- and encoding, deinterlacing, ...
  - ▶ Physics engines
- ▶ Full programmability at several pipeline stages
  - ▶ fully programmable
  - ▶ but optimized for massively parallel operations



▶ 119

### What makes GPU so fast?

- ▶ 3D rendering can be very efficiently parallelized
  - ▶ Millions of pixels
  - ▶ Millions of triangles
  - ▶ Many operations executed independently at the same time
- ▶ This is why modern GPUs
  - ▶ Contain between hundreds and thousands of SIMD processors
    - ▶ Single Instruction Multiple Data – operate on large arrays of data
  - ▶ >>400 GB/s memory access
    - ▶ This is much higher bandwidth than CPU
    - ▶ But peak performance can be expected for very specific operations


▶ 120

### GPU APIs (Application Programming Interfaces)

<p><b>OpenGL</b></p> <ul style="list-style-type: none"> <li>▶ Multi-platform</li> <li>▶ Open standard API</li> <li>▶ Focus on general 3D applications                             <ul style="list-style-type: none"> <li>▶ Open GL driver manages the resources</li> </ul> </li> </ul>		<p><b>DirectX</b></p> <ul style="list-style-type: none"> <li>▶ Microsoft Windows / Xbox</li> <li>▶ Proprietary API</li> <li>▶ Focus on games                             <ul style="list-style-type: none"> <li>▶ Application manages resources</li> </ul> </li> </ul>	
--	---	--	---

▶ 121

### One more API



- ▶ Vulkan – cross platform, open standard
- ▶ Low-overhead API for high performance 3D graphics
- ▶ Compared to OpenGL / DirectX
  - ▶ Reduces CPU load
  - ▶ Better support of multi-CPU-core architectures
  - ▶ Finer control of GPU
- ▶ But
  - ▶ The code for drawing a few primitives can take 1000s line of code
  - ▶ Intended for game engines and code that must be very well optimized

▶ 122

## And one more



- ▶ **Metal (Apple iOS8)**
  - ▶ low-level, low-overhead 3D GFX and compute shaders API
  - ▶ Support for Apple A7, Intel HD and Iris, AMD, Nvidia
  - ▶ Similar design as modern APIs, such as Vulkan
  - ▶ Swift or Objective-C API
  - ▶ Used mostly on iOS

▶ 123

## GPGPU - general purpose computing

- ▶ **OpenGL and DirectX are not meant to be used for general purpose computing**
  - ▶ Example: physical simulation, machine learning
- ▶ **CUDA – Nvidia's architecture for parallel computing**
  - ▶ C-like programming language
  - ▶ With special API for parallel instructions
  - ▶ Requires Nvidia GPU
- ▶ **OpenCL – Similar to CUDA, but open standard**
  - ▶ Can run on both GPU and CPU
  - ▶ Supported by AMD, Intel and NVidia, Qualcomm, Apple, ...



▶ 124

## GPU and mobile devices



- ▶ **OpenGL ES 1.0-3.2**
  - ▶ Stripped version of OpenGL
  - ▶ Removed functionality that is not strictly necessary on mobile devices
- ▶ **Devices**
  - ▶ iOS: iPhone, iPad
  - ▶ Android phones
  - ▶ PlayStation 3
  - ▶ Nintendo 3DS
  - ▶ and many more



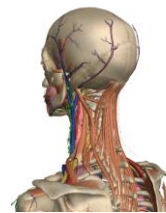
OpenGL ES 2.0 rendering (iOS)

▶ 125

## WebGL



- ▶ JavaScript library for 3D rendering in a web browser
- ▶ **WebGL 1.0** - based on OpenGL ES 2.0
- ▶ **WebGL 2.0** – based on OpenGL ES 3.0
  - ▶ Chrome and Firefox (2017)
- ▶ Most modern browsers support WebGL
- ▶ Potentially could be used to create 3D games in a browser
  - ▶ and replace Adobe Flash

<http://zygotebody.com/>

▶ 126

## OpenGL in Java

- ▶ Standard Java API does not include OpenGL interface
- ▶ But several wrapper libraries exist
  - ▶ Java OpenGL – JOGL
  - ▶ Lightweight Java Game Library - LWJGL
- ▶ We will use LWJGL 3
  - ▶ Seems to be better maintained
  - ▶ Access to other APIs (OpenCL, OpenAL, ...)
- ▶ We also need a linear algebra library
  - ▶ JOML – Java OpenGL Math Library
  - ▶ Operations on 2, 3, 4-dimensional vectors and matrices

▶ 127

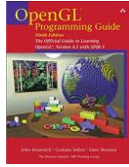
## OpenGL History

- ▶ Proprietary library IRIS GL by SGI
- ▶ OpenGL 1.0 (1992)
- ▶ OpenGL 1.2 (1998)
- ▶ OpenGL 2.0 (2004)
  - ▶ GLSL
  - ▶ Non-power-of-two (NPOT) textures
- ▶ OpenGL 3.0 (2008)
  - ▶ Major overhaul of the API
  - ▶ Many features from previous versions deprecated
- ▶ OpenGL 3.2 (2009)
  - ▶ Core and Compatibility profiles
- ▶ Geometry shaders
- ▶ OpenGL 4.0 (2010)
  - ▶ Catching up with Direct3D 11
- ▶ OpenGL 4.5 (2014)
- ▶ OpenGL 4.6 (2017)
  - ▶ SPIR-V shaders

▶ 128

### How to learn OpenGL?

- ▶ Lectures – algorithms behind OpenGL, general principles
- ▶ Tick 2 – detailed tutorial, learning by doing
- ▶ References
  - ▶ **OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V** by John Kessenich, Graham Sellers, Dave Shreiner ISBN-10: 0134495497
  - ▶ OpenGL quick reference guide <https://www.opengl.org/documentation/glsl/>
  - ▶ Google search: „man gl.....”



▶ 129

### OpenGL rendering pipeline

▶

### OpenGL programming model

#### CPU code

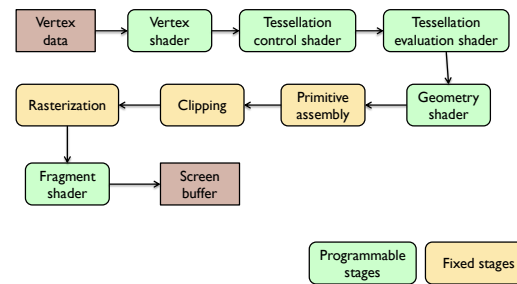
- ▶ **gl\*** functions that
  - ▶ Create OpenGL objects
  - ▶ Copy data CPU-<->GPU
  - ▶ Modify OpenGL state
  - ▶ Enqueue operations
  - ▶ Synchronize CPU & GPU
- ▶ **C99 library**
- ▶ Wrappers in most programming language

#### GPU code

- ▶ **Fragment shaders**
- ▶ **Vertex shaders**
- ▶ **and other shaders**
- ▶ **Written in GLSL**
  - ▶ Similar to C
  - ▶ From OpenGL 4.6 could be written in other language and compiled to SPIR-V

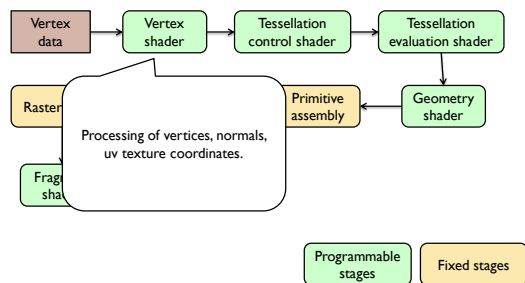
▶ 131

### OpenGL rendering pipeline



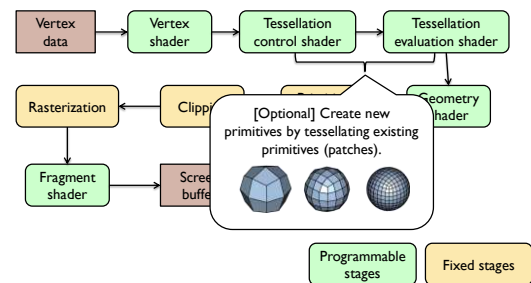
▶ 132

### OpenGL rendering pipeline

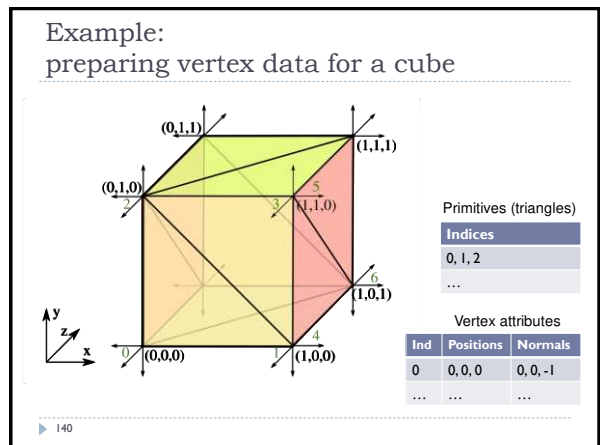
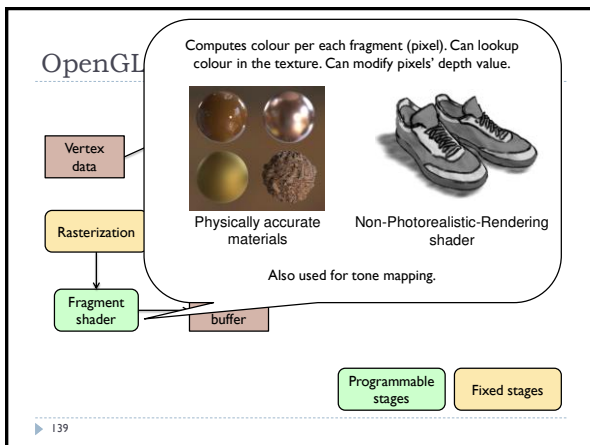
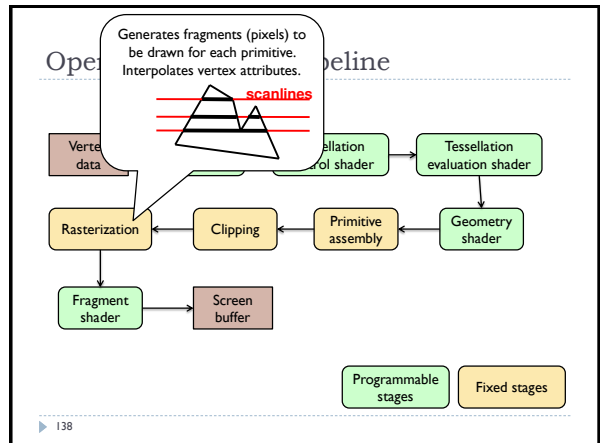
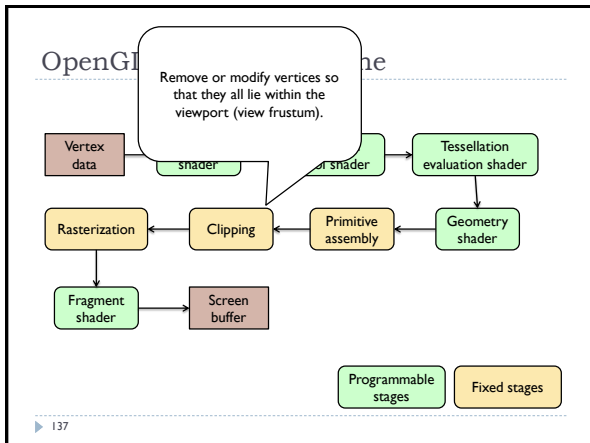
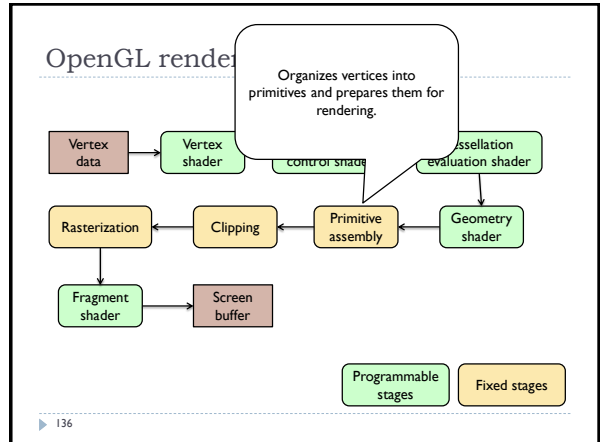
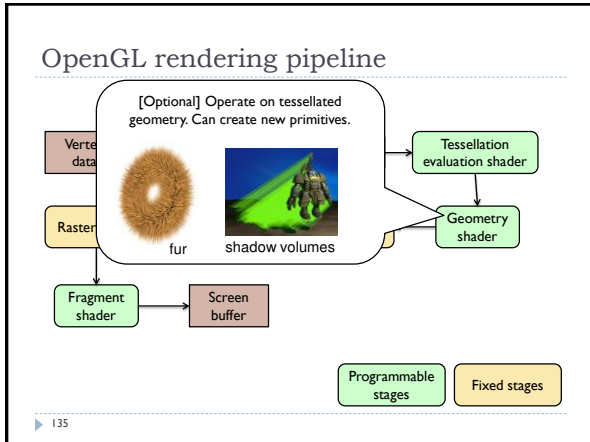


▶ 133

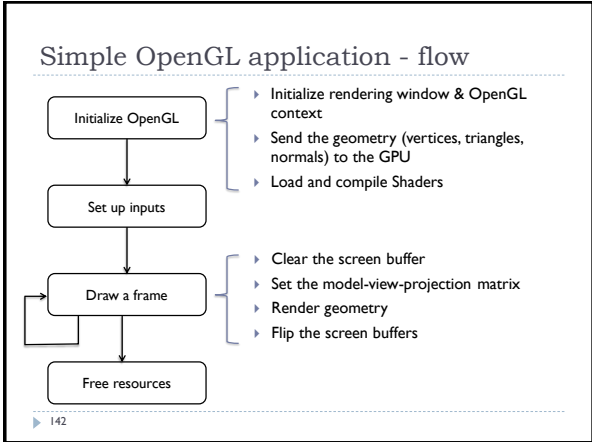
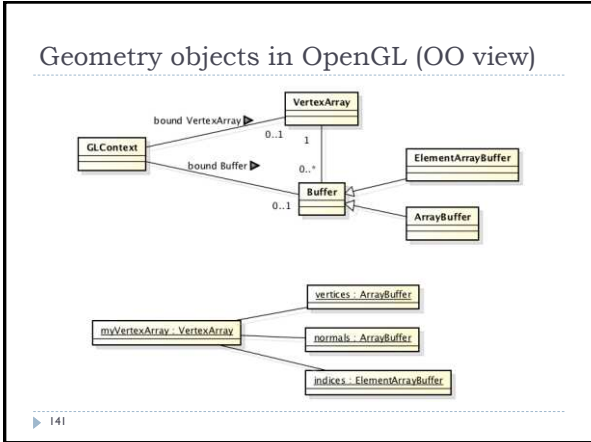
### OpenGL rendering pipeline



▶ 134







## GLSL - fundamentals

143

## Shaders

- Shaders are small programs executed on a GPU
  - Executed for each vertex, each pixel (fragment), etc.
- They are written in GLSL (OpenGL Shading Language)
  - Similar to C and Java
  - Primitive (int, float) and aggregate data types (ivec3, vec3)
  - Structures and arrays
  - Arithmetic operations on scalars, vectors and matrices
  - Flow control: if, switch, for, while
  - Functions

144

### Example of a vertex shader

```

#version 330
in vec3 position;           // vertex position in local space
in vec3 normal;            // vertex normal in local space
out vec3 frag_normal;      // fragment normal in world space
uniform mat4.mvp_matrix;    // model-view-projection matrix

void main()
{
    // Typically normal is transformed by the model matrix
    // Since the model matrix is identity in our case, we do not modify normals
    frag_normal = normal;

    // The position is projected to the screen coordinates using.mvp_matrix
    gl_Position =.mvp_matrix * vec4(position, 1.0);
}
    
```

Why is this piece of code needed?

145

## 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);

146

## Indexing components in aggregate types

- ▶ 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

▶ 147

## Swizzling

You can 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;
```

▶ 148

## Arrays

- ▶ Similar to 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;
}
```

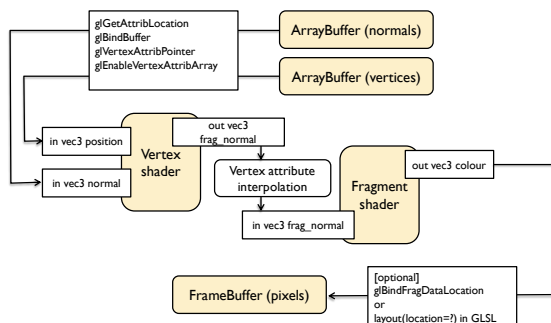
▶ 149

## 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)
- ▶ Example: `const float pi=3.14;`

▶ 150

## Shader inputs and outputs



▶ 151

## GLSL 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
```

▶ See the quick reference guide at:  
<https://www.khronos.org/opengl/documentation/gsl/>

▶ 152

### GLSL 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**
- ▶ See the quick reference guide at:  
<https://www.opengl.org/documentation/glsl/>

▶ 153

### GLSL flow control

```

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

switch( int_value ) {
    case n:
        // statements
        break;
    case m:
        // statements
        break;
    default:
        // statements
}

for( int i = 0; i < 10; i++ ) {
    ...
}

while( n < 10 ) {
    ...
}

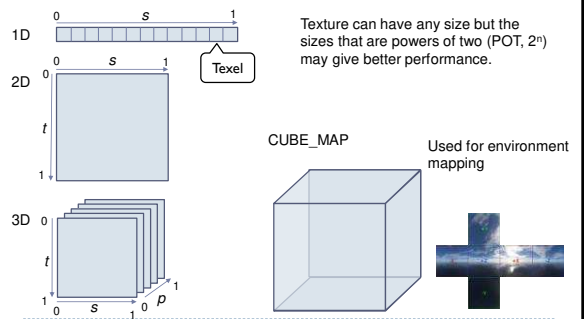
do {
    ...
} while ( n < 10 )
    
```

▶ 154

### Textures



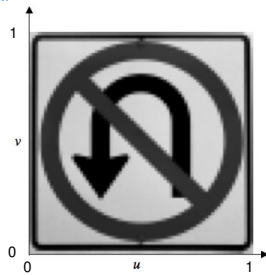
### (Most important) OpenGL texture types



▶ 156

### Texture mapping

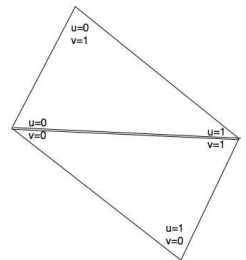
- ▶ 1. Define your texture function (image)  $T(u,v)$
- ▶  $(u,v)$  are texture coordinates



▶ 157

### Texture mapping

- ▶ 2. Define the correspondence between the vertices on the 3D object and the texture coordinates



▶ 158

### Texture mapping

▶ 3. When rendering, for every surface point compute texture coordinates. Use the texture function to get texture value. Use as color or reflectance.

▶ 159

### Sampling

Texture

Up-sampling  
More pixels than texels  
Values need to be interpolated

Down-sampling  
Fewer pixels than texels  
Values need to be averaged over an area of the texture (usually using a mipmap)

▶ 160

### Nearest neighbor vs. bilinear interpolation (upsampling)

Nearest neighbour

Bilinear interpolation

Pick the nearest texel: D

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

▶ 161

### Texture mapping examples

nearest-neighbour

bilinear

▶ 162

### Up-sampling

nearest-neighbour

*blocky artefacts*

bilinear

*blurry artefacts*

- ▶ if one pixel in the texture map covers several pixels in the final image, you get visible artefacts
- ▶ only practical way to prevent this is to ensure that texture map is of sufficiently high resolution that it does not happen

▶ 163

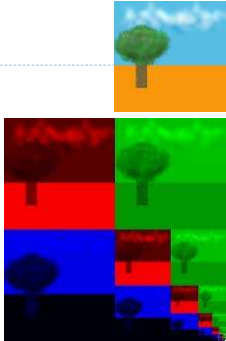
### Down-sampling

- ▶ if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

▶ 164

## Mipmap

- ▶ Textures are often stored at multiple resolutions as a mipmap
  - ▶ Each level of the pyramid is half the size of the lower level
  - ▶ Mipmap resolution is always power-of-two (1024, 512, 256, 128, ...)
- ▶ It provides pre-filtered texture (area-averaged) when screen pixels are larger than the full resolution texels
- ▶ Mipmap requires just an additional 1/3 of the original texture size to store
- ▶ OpenGL can generate a mipmap with `glGenerateMipmap(GL_TEXTURE_2D)`

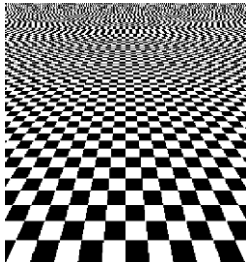


This image is an illustration showing only 1/3 increase in storage. Mipmaps are stored differently in the GPU memory.

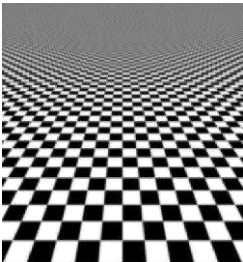
▶ 165

## Down-sampling

without area averaging



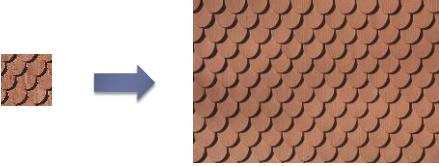
with area averaging



▶ 166

## Texture tiling

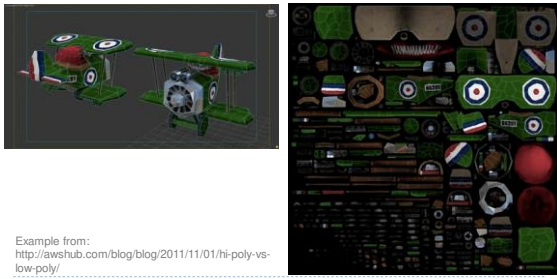
- ▶ Repetitive patterns can be represented as texture tiles.
- ▶ The texture folds over, so that
  - ▶  $T(u=1.1, v=0) = T(u=0.1, v=0)$



▶ 167

## Multi-surface UV maps

- ▶ A single texture is often used for multiple surfaces and objects




Example from:  
<http://awshub.com/blog/blog/2011/11/01/hi-poly-vs-low-poly/>

▶ 168

## Bump (normal) mapping

- ▶ Special kind of texture that modifies surface normal
  - ▶ Surface normal is a vector that is perpendicular to a surface
- ▶ The surface is still flat but shading appears as on an uneven surface
- ▶ Easily done in fragment shaders

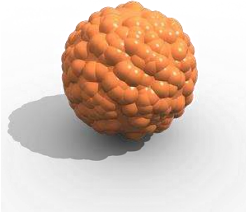


From Computer Desktop Encyclopedia  
 Reproduced with permission.  
 ©2001 Interglyph Computer Systems

▶ 169

## Displacement mapping

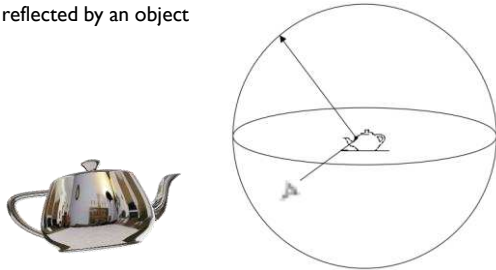
- ▶ Texture that modifies surface
- ▶ Better results than bump mapping since the surface is not flat
- ▶ Requires geometry shaders



▶ 170

## Environment mapping

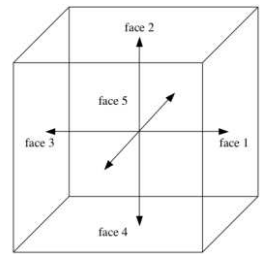
- ▶ To show environment reflected by an object



▶ 171

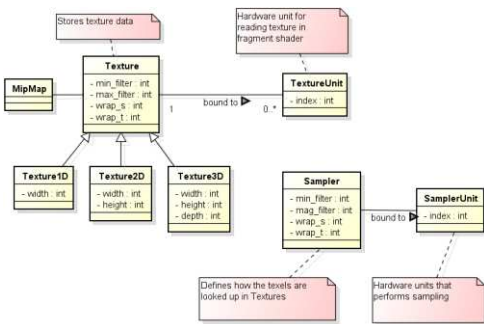
## Environment mapping

- ▶ Environment cube
- ▶ Each face captures environment in that direction



▶ 172

## Texture objects in OpenGL



▶ 173

## 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);
```

▶ 174

## 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);
```

▶ 175

## Raster buffers (colour, depth, stencil)

▶

### Render buffers in OpenGL

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.

177

### Double buffering

- To avoid flicker, tearing
- Use two buffers (rasters):
  - Front buffer – what is shown on the screen
  - Back buffer – not shown, GPU draws into that buffer
- When drawing is finished, swap front- and back-buffers

178

### Triple buffering

- Do not wait for swapping to start drawing the next frame

Shortcomings

- More memory needed
- Higher delay between drawing and displaying a frame

179

### Vertical Synchronization: V-Sync

- Pixels are copied from colour buffer to monitor row-by-row
- If front & back buffer are swapped during this process:
  - Upper part of the screen contains previous frame
  - Lower part of the screen contains current frame
  - Result: tearing artefact
- Solution: When V-Sync is enabled
  - `glfwSwapInterval(1);`
  - `glSwapBuffers()` waits until the last row is copied to the display.

180

### No V-Sync vs. V-Sync

181

### FreeSync (AMD) & G-Sync (Nvidia)

- Adaptive sync
  - Graphics card controls timing of the frames on the display
  - Can save power for 30fps video of when the screen is static
  - Can reduce lag for real-time graphics

182

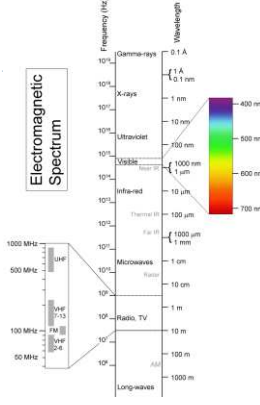


### Colour and colour spaces

183

### Electromagnetic spectrum

- ▶ Visible light
  - ▶ Electromagnetic waves of wavelength in the range 380nm to 730nm
  - ▶ Earth's atmosphere lets through a lot of light in this wavelength band
  - ▶ Higher in energy than thermal infrared, so heat does not interfere with vision






184

### Colour

- ▶ There is no physical definition of colour – colour is the result of our perception
- ▶ For emissive displays / objects
 

colour = perception( spectral\_emission )
- ▶ For reflective displays / objects
 

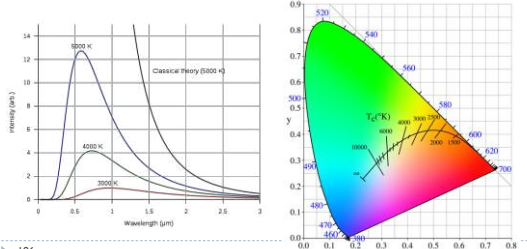
colour = perception( illumination \* reflectance )

185

### Black body radiation


- ▶ Electromagnetic radiation emitted by a perfect absorber at a given temperature
- ▶ Graphite is a good approximation of a black body



186

### Correlated colour temperature

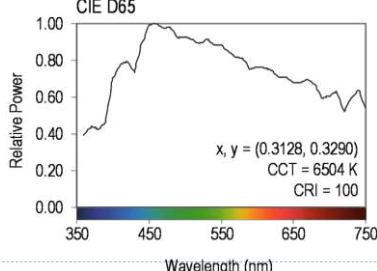
- ▶ The temperature of a black body radiator that produces light most closely matching the particular source
- ▶ Examples:
  - ▶ Typical north-sky light: 7500 K
  - ▶ Typical average daylight: 6500 K
  - ▶ Domestic tungsten lamp (100 to 200 W): 2800 K
  - ▶ Domestic tungsten lamp (40 to 60 W): 2700 K
  - ▶ Sunlight at sunset: 2000 K
- ▶ Useful to describe colour of the **illumination** (source of light)



187

### Standard illuminant D65

- ▶ Mid-day sun in Western Europe / Northern Europe
- ▶ Colour temperature approx. 6500 K



188



### Reflectance

- ▶ Most of the light we see is reflected from objects
- ▶ These objects absorb a certain part of the light spectrum

Spectral reflectance of ceramic tiles

REFLECTANCE

WAVELENGTH λ (nm)

Why not red?

▶ 18

### Reflected light

$$L(\lambda) = I(\lambda)R(\lambda)$$

- ▶ Reflected light = illumination \* reflectance

CIE D65

Relative Power

Wavelength (nm)

x, y = (0.3128, 0.3290)

CCT = 6504 K

CRI = 100

REFLECTANCE

WAVELENGTH λ (nm)

The same object may appear to have different color under different illumination.

▶ 190

### Fluorescence

Normalized Data

Wavelength (nm)

Absorption

FI 420nm Ex

FI 450nm Ex

FI 470nm Ex

Normalized Data

Wavelength (nm)

Absorption

FI 420nm Ex

FI 450nm Ex

FI 470nm Ex

From: <http://en.wikipedia.org/wiki/Fluorescence>

▶ 191

### Colour vision

- ▶ Cones are the photoreceptors responsible for color vision
- ▶ Only daylight, we see no colors when there is not enough light
- ▶ 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. S, M and L curves are normalized in this plot.

▶ 192

### Perceived light

- ▶ 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

▶ 193

### Metamers


- ▶ 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**
- ▶ Example:
  - ▶ I = [L<sub>1</sub>, M<sub>1</sub>, S<sub>1</sub>]
  - ▶ II = [L<sub>2</sub>, M<sub>2</sub>, S<sub>2</sub>]


▶ 194

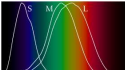
### Practical application of metamerism

- ▶ Displays do not emit the same light spectra as real-world objects
- ▶ Yet, the colours on a display look almost identical

On the display









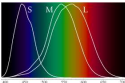
= [L<sub>1</sub>, M<sub>1</sub>, S<sub>1</sub>]

II

In real world





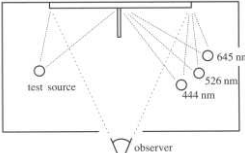


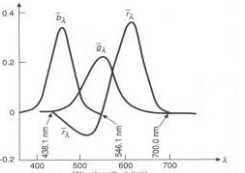
= [L<sub>2</sub>, M<sub>2</sub>, S<sub>2</sub>]

▶ 195

### Tristimulus Colour Representation

- ▶ **Observation**
  - ▶ Any colour can be matched using three linear independent reference colours
  - ▶ May require "negative" contribution to test colour
  - ▶ Matching curves describe the value for matching monochromatic spectral colours of equal intensity
  - ▶ With respect to a certain set of primary colours





▶ 196

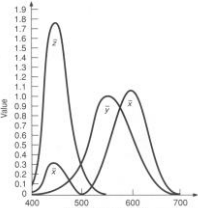
### Standard Colour Space CIE-XYZ

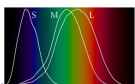
- ▶ **CIE Experiments [Guild and Wright, 1931]**
  - ▶ Colour matching experiments
  - ▶ Group ~12 people with „normal“ colour vision
  - ▶ 2 degree visual field (fovea only)
- ▶ **CIE 2006 XYZ**
  - ▶ Derived from LMS color matching functions by Stockman & Sharpe
  - ▶ S-cone response differs the most from CIE 1931
- ▶ **CIE-XYZ Colour Space**
  - ▶ Goals
    - ▶ Abstract from concrete primaries used in experiment
    - ▶ All matching functions are positive
    - ▶ Primary „Y“ is roughly proportionally to light intensity (luminance)

▶ 197

### Standard Colour Space CIE-XYZ

- ▶ **Standardized imaginary primaries CIE XYZ (1931)**
  - ▶ Could match all physically realizable colour stimuli
  - ▶ Y is roughly equivalent to luminance
    - ▶ Shape similar to luminous efficiency curve
  - ▶ Monochromatic spectral colours form a curve in 3D XYZ-space





Cone sensitivity curves can be obtained by a linear transformation of CIE XYZ

▶ 198

### Luminance

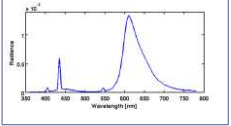
- ▶ **Luminance – how bright the surface will appear regardless of its colour. Units: cd/m<sup>2</sup>**

Luminance

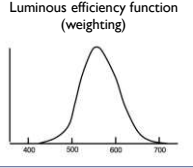
$$L_V = \int_{350}^{700} kL(\lambda)V(\lambda)d\lambda$$

$$k = \frac{1}{683.002}$$

Light spectrum (radiance)



Luminous efficiency function (weighting)



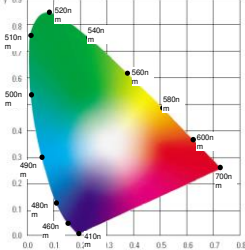
▶ 199

### CIE chromaticity diagram

- ▶ *chromaticity* values are defined in terms of x, y, z

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

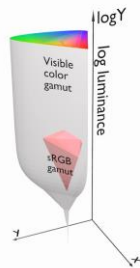
- ▶ ignores luminance
- ▶ can be plotted as a 2D function
- ▶ pure colours (single wavelength) lie along the outer curve
- ▶ all other colours are a mix of pure colours and hence lie inside the curve
- ▶ points outside the curve do not exist as colours



▶ 200

### Visible vs. displayable colours

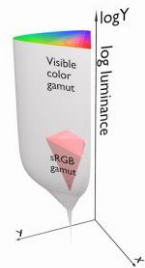
- ▶ All physically possible and visible colours form a solid in XYZ space
- ▶ Each display device can reproduce a subspace of that space
- ▶ A chromacity diagram is a slice taken from a 3D solid in XYZ space
- ▶ Colour Gamut – the solid in a colour space
  - ▶ Usually defined in XYZ to be device-independent



▶ 201

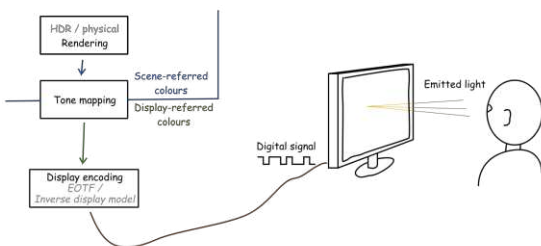
### Standard vs. High Dynamic Range

- ▶ **HDR** cameras/formats/displays attempt capture/represent/reproduce (almost) all visible colours
  - ▶ They represent scene colours and therefore we often call this representation *scene-referred*
- ▶ **SDR** cameras/formats/devices attempt to capture/represent/reproduce only colours of a standard sRGB colour gamut, mimicking the capabilities of CRT's monitors
  - ▶ They represent display colours and therefore we often call this representation *display-referred*



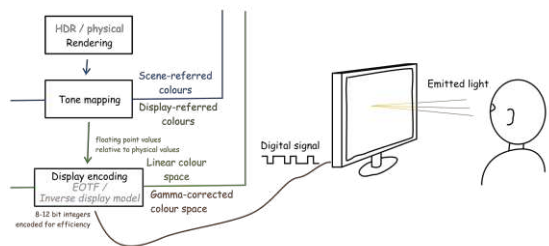
▶ 202

### From rendering to display



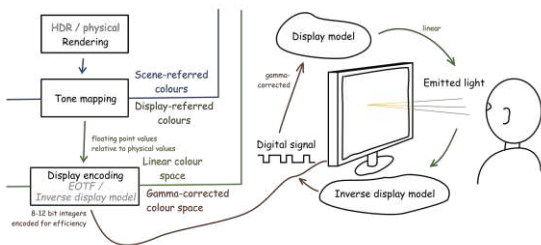
▶ 203

### From rendering to display



▶ 204

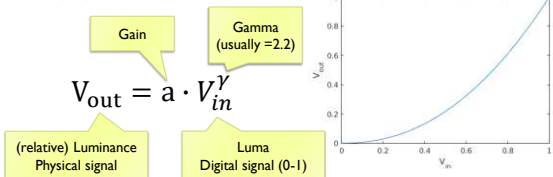
### From rendering to display



▶ 205

### Display encoding for SDR: gamma correction

- ▶ Gamma correction is often used to encode luminance or tri-stimulus color values (RGB) in imaging systems (displays, printers, cameras, etc.)



Inverse:  $V_{in} = \left(\frac{1}{a} \cdot V_{out}\right)^{\frac{1}{\gamma}}$

Colour: the same equation applied to red, green and blue colour channels.

▶ 206

### Why is gamma needed?

- ▶ *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 color channel without gamma correction
- ▶ And accidentally it was also the response of the CRT gun

207

### Luma – gray-scale pixel value

- ▶ **Luma** - pixel brightness in *gamma corrected* units  
 $L' = 0.2126R' + 0.7152G' + 0.0722B'$ 
  - ▶  $R', G'$  and  $B'$  are *gamma-corrected* colour values
  - ▶ Prime symbol denotes *gamma corrected*
  - ▶ Used in image/video coding
- ▶ Note that relative **luminance** is often approximated with  
 $L = 0.2126R + 0.7152G + 0.0722B$   
 $= 0.2126(R')^\gamma + 0.7152(G')^\gamma + 0.0722(B')^\gamma$ 
  - ▶  $R, G,$  and  $B$  are *linear* colour values
  - ▶ Luma and luminance are different quantities despite similar formulas

208

### Standards for display encoding

Display type	Colour space	EOTF	Bit depth
Standard Dynamic Range	ITU-R 709	2.2 gamma / sRGB	8 to 10
High Dynamic Range	ITU-R 2020	ITU-R 2100 (PQ/HLG)	10 to 12

**Colour space**  
What is the colour of "pure" red, green and blue

**Electro-Optical Transfer Function**  
How to efficiently encode each primary colour

209

### How to transform between RGB colour spaces?

- ▶ From ITU-R 709 RGB to XYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix}_{R709toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709}$$

Relative XYZ of the red primary

Relative XYZ of the green primary

Relative XYZ of the blue primary

210

### How to transform between RGB colour spaces?

- ▶ From ITU-R **709** RGB to ITU-R **2020** RGB:  
 $\begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R2020} = M_{XYZtoR2020} \cdot M_{R709toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709}$
- ▶ From ITU-R **2020** RGB to ITU-R **709** RGB:  
 $\begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709} = M_{XYZtoR709} \cdot M_{R2020toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R2020}$
- ▶ Where:  
 $M_{R709toXYZ} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix}$  and  $M_{XYZtoR709} = M_{R709toXYZ}^{-1}$   
 $M_{R2020toXYZ} = \begin{bmatrix} 0.6370 & 0.1446 & 0.1689 \\ 0.2627 & 0.6780 & 0.0593 \\ 0.0000 & 0.0281 & 1.0610 \end{bmatrix}$  and  $M_{XYZtoR2020} = M_{R2020toXYZ}^{-1}$

211


### Representing colour

- ▶ We need a mechanism which allows us to represent colour in the computer by some set of numbers
  - ▶ A) preferably a small set of numbers which can be quantised to a fairly **small number of bits** each
    - ▶ Linear and gamma corrected RGB, sRGB
  - ▶ B) a set of numbers that are **easy to interpret**
    - ▶ Munsell's *artists'* scheme
    - ▶ HSV, HLS
  - ▶ C) a set of numbers in a 3D space so that the (Euclidean) distance in that space corresponds to approximately **perceptually uniform** colour differences
    - ▶ CIE Lab, CIE Luv

212

### RGB space

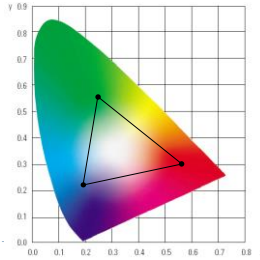
- ▶ Most display devices that output light mix red, green and blue lights to make colour
  - ▶ televisions, CRT monitors, LCD screens
- ▶ Nominally, *RGB* space is a cube
- ▶ The device puts physical limitations on:
  - ▶ the range of colours which can be displayed
  - ▶ the brightest colour which can be displayed
  - ▶ the darkest colour which can be displayed



▶ 213

### RGB in XYZ space


- ▶ CRTs and LCDs mix red, green, and blue to make all other colours
- ▶ the red, green, and blue **primaries** each map to a point in *CIE xy* space
- ▶ any colour within the resulting triangle can be displayed
  - ▶ any colour outside the triangle cannot be displayed
  - ▶ for example: CRTs cannot display very saturated purple, turquoise, or yellow



▶ 214


### CMY space

- ▶ printers make colour by mixing coloured inks
- ▶ the important difference between inks (*CMY*) and lights (*RGB*) is that, while lights *emit* light, inks *absorb* light
  - ▶ cyan absorbs red, reflects blue and green
  - ▶ magenta absorbs green, reflects red and blue
  - ▶ yellow absorbs blue, reflects green and red
- ▶ *CMY* is, at its simplest, the inverse of *RGB*
- ▶ *CMY* space is nominally a cube



▶ 215

### CMYK space




- ▶ in real printing we use black (key) as well as *CMY*
- ▶ why use black?
  - ▶ inks are not perfect absorbers
  - ▶ mixing  $C + M + Y$  gives a muddy grey, not black
  - ▶ lots of text is printed in black: trying to align  $C, M$  and  $Y$  perfectly for black text would be a nightmare

▶ 216

### Munsell's colour classification system

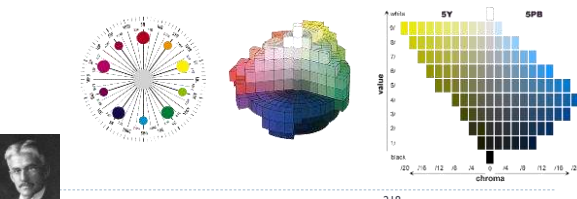
- ▶ three axes
  - ▶ hue ▶ the dominant colour
  - ▶ value ▶ bright colours/dark colours
  - ▶ chroma ▶ vivid colours/dull colours
- ▶ can represent this as a 3D graph



▶ 217

### Munsell's colour classification system

- ▶ any two adjacent colours are a standard "perceptual" distance apart
  - ▶ worked out by testing it on people
  - ▶ a highly irregular space
    - ▶ e.g. vivid yellow is much brighter than vivid blue



▶ 218  
invented by Albert H. Munsell, an American artist, in 1905 in an attempt to systematically classify colours

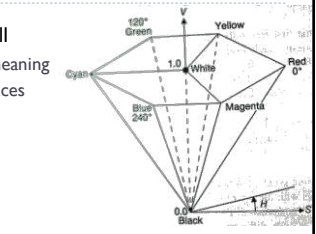
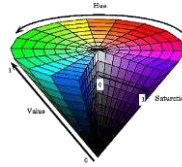
### Colour spaces for user-interfaces

- ▶ *RGB* and *CMY* are based on the physical devices which produce the coloured output
- ▶ *RGB* and *CMY* are difficult for humans to use for selecting 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?
- ▶ computer interface designers have developed basic transformations of *RGB* which resemble Munsell's human-friendly system

219

### HSV: hue saturation value

- ▶ three axes, as with Munsell
- ▶ hue and value have same meaning
- ▶ the term "saturation" replaces the term "chroma"

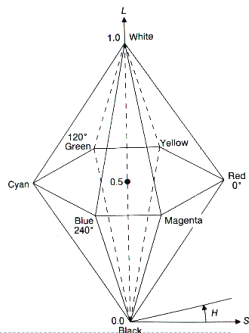


- ◆ designed by Alvy Ray Smith in 1978
- ◆ algorithm to convert *HSV* to *RGB* and back can be found in Foley et al., Figs 13.33 and 13.34

220

### HLS: hue lightness saturation

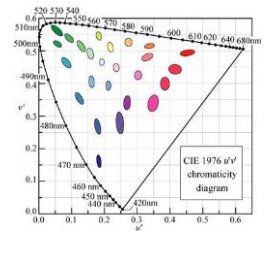
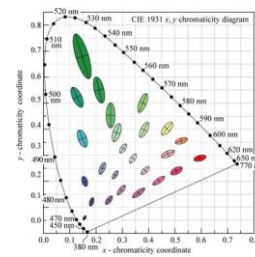
- ▶ a simple variation of *HSV*
  - ◆ hue and saturation have same meaning
  - ◆ the term "lightness" replaces the term "value"
- ▶ designed to address the complaint that *HSV* has all pure colours having the same lightness/value as white
  - ◆ designed by Metrick in 1979
  - ◆ algorithm to convert *HLS* to *RGB* and back can be found in Foley et al., Figs 13.36 and 13.37



221

### Perceptually uniformity

- ▶ MacAdam ellipses & visually indistinguishable colours



In CIE xy chromatic coordinates

In CIE u'v' chromatic coordinates

222

### CIE L\*u\*v\* and u'v'

- ▶ Approximately perceptually uniform
- ▶ u'v' chromaticity

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

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

#### ▶ CIE LUV

$$\text{Lightness } L^* = \begin{cases} \left(\frac{29}{27}\right)^3 Y/Y_n & Y/Y_n \leq \left(\frac{6}{29}\right)^3 \\ 116(Y/Y_n)^{1/3} - 16 & Y/Y_n > \left(\frac{6}{29}\right)^3 \end{cases}$$

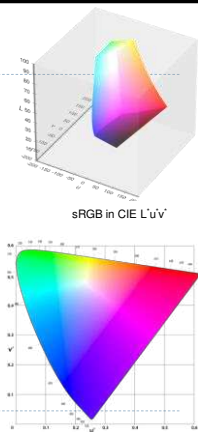
$$\text{Chromaticity coordinates } \begin{cases} u^* = 13L^* \cdot (u' - u'_n) \\ v^* = 13L^* \cdot (v' - v'_n) \end{cases}$$

Colours less distinguishable when dark

#### ▶ Hue and chroma

$$C_{uv}^* = \sqrt{(u^*)^2 + (v^*)^2}$$

$$h_{uv}^* = \text{atan2}(v^*, u^*)$$



223

### CIE L\*a\*b\* colour space

- ▶ Another approximately perceptually uniform colour space

$$L^* = 116f\left(\frac{Y}{Y_n}\right) - 16$$

$$a^* = 500\left(f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right)$$

$$b^* = 200\left(f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right)$$

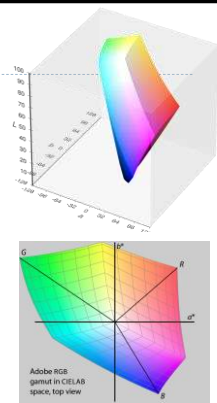
Trichromatic values of the white point, e.g.  
 $X_n = 95.047,$   
 $Y_n = 100.000,$   
 $Z_n = 108.883$

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

$$\delta = \frac{6}{29}$$


#### ▶ Chroma and hue

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



224





**Lab space**

- ▶ this visualization shows those colours in *Lab* space which a human can perceive
- ▶ again we see that human perception of colour is not uniform
  - ▶ perception of colour diminishes at the white and black ends of the *L* axis
  - ▶ the maximum perceivable chroma differs for different hues

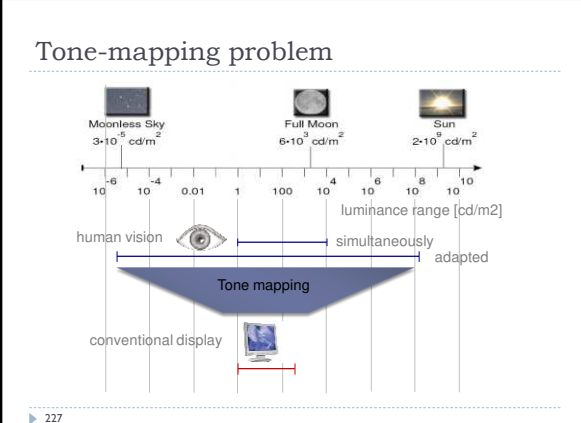
225

**Colour - references**

- ▶ Chapters „Light” and „Colour” in
  - ▶ Shirley, P. & Marschner, S., *Fundamentals of Computer Graphics*
- ▶ Textbook on colour appearance
  - ▶ Fairchild, M. D. (2005). *Color Appearance Models* (second.). John Wiley & Sons.

226

**Tone-mapping problem**



luminance range [cd/m<sup>2</sup>]

human vision (simultaneously adapted)

conventional display

▶ 227

**Why do we need tone mapping?**

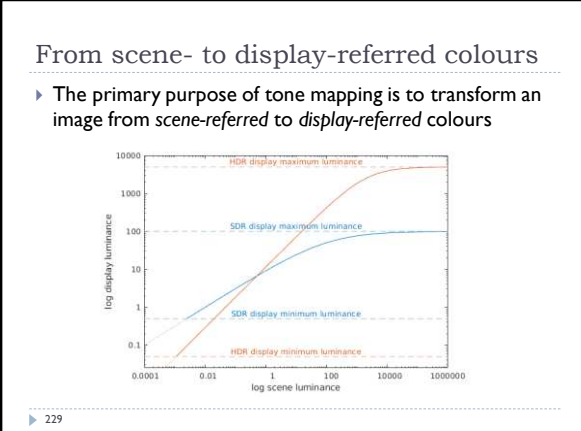
- ▶ To **reduce dynamic range**
- ▶ To **customize the look** (colour grading)
- ▶ To **simulate human vision** (for example night vision)
- ▶ To **simulate a camera** (for example motion blur)
- ▶ To **adapt** displayed images to a display and **viewing conditions**
- ▶ To make rendered images look **more realistic**
- ▶ To map from **scene- to display-referred** colours

▶ Different tone mapping operators achieve different goals

▶ 228

**From scene- to display-referred colours**

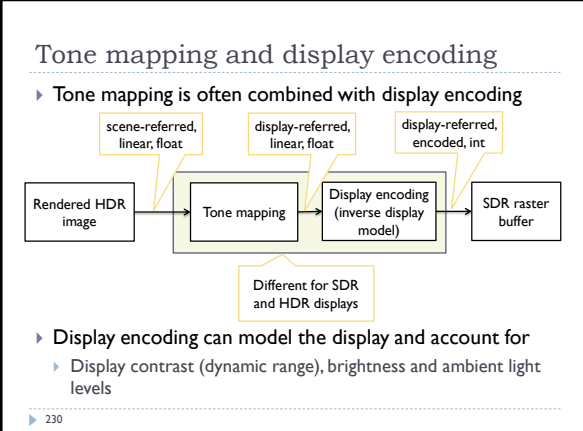
- ▶ The primary purpose of tone mapping is to transform an image from *scene-referred* to *display-referred* colours



▶ 229

**Tone mapping and display encoding**

- ▶ Tone mapping is often combined with display encoding



- ▶ Display encoding can model the display and account for
  - ▶ Display contrast (dynamic range), brightness and ambient light levels

▶ 230

### sRGB textures and display coding

- ▶ OpenGL offers sRGB textures to automate RGB to/from sRGB conversion
  - ▶ sRGB textures store data in gamma-corrected space
  - ▶ sRGB converted to (linear) RGB on texture look-up (and filtering)
    - ▶ Inverse display coding
  - ▶ RGB to sRGB conversion when writing to sRGB texture
    - ▶ with glEnable(GL\_FRAMEBUFFER\_SRGB)
    - ▶ Forward display coding

▶ 231

### Basic tone-mapping and display coding

- ▶ The simplest form of tone-mapping is the exposure/brightness adjustment:

$$R_d = \frac{R_s}{L_{white}}$$

Annotations: Display-referred red value, Scene-referred, Scene-referred luminance of white

- ▶ R for red, the same for green and blue
- ▶ No contrast compression, only for a moderate dynamic range

- ▶ The simplest form of display coding is the "gamma"

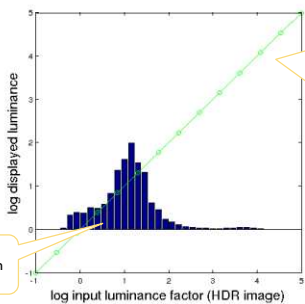
$$R' = (R_d)^{\frac{1}{\gamma}}$$

Annotations: Prime (') denotes a gamma-corrected value, Typically  $\gamma=2.2$

- ▶ For SDR displays only

▶ 232

### Tone-curve

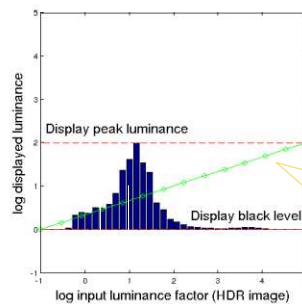


The „best“ tone-mapping is the one which does not do anything, i.e. slope of the tone-mapping curves is equal to 1.

Image histogram

▶ 233

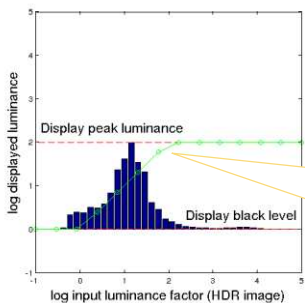
### Tone-curve



But in practice contrast (slope) must be limited due to display limitations.

▶ 234

### Tone-curve

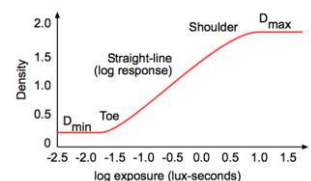


Global tone-mapping is a compromise between clipping and contrast compression.

▶ 235

### Sigmoidal tone-curves

- ▶ Very common in digital cameras
  - ▶ Mimic the response of analog film
  - ▶ Analog film has been engineered over many years to produce good tone-reproduction
- ▶ Fast to compute



▶ 236



### Sigmoidal tone mapping

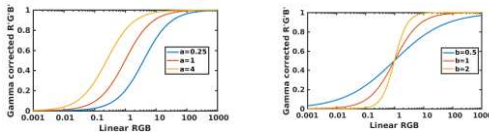
Simple formula for a sigmoidal tone-curve:

$$R'(x, y) = \frac{R(x, y)^b}{\left(\frac{L_m}{a}\right)^b + R(x, y)^b}$$

where  $L_m$  is the geometric mean (or mean of logarithms):

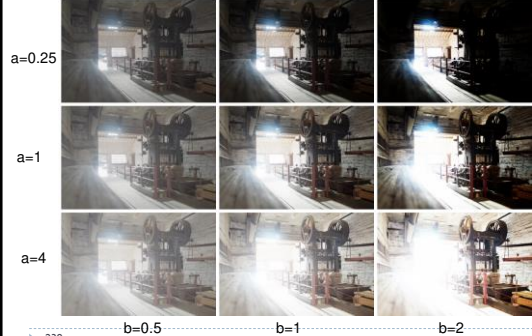
$$L_m = \exp\left(\frac{1}{N} \sum_{(x,y)} \ln(L(x, y))\right)$$

and  $L(x, y)$  is the luminance of the pixel  $(x, y)$ .



237

### Sigmoidal tone mapping example



238

### Glare Illusion



\*Alan Wake\* © Remedy Entertainment

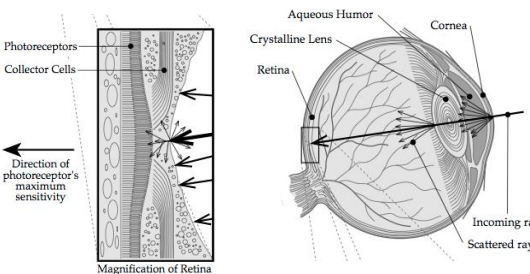
239

### Glare Illusion



240 Computer Graphics HDR rendering in games

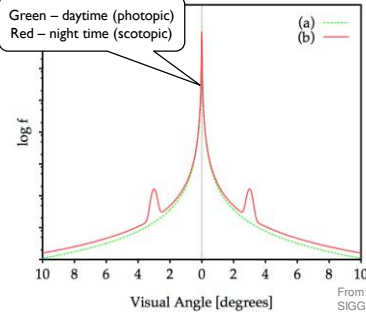
### Scattering of the light in the eye



From: Sekuler, R., and Blake, R. Perception, second ed. McGraw- Hill, New York, 1990

241

### Point Spread Function of the eye



- ▶ What portion of the light is scattered towards a certain visual angle
- ▶ To simulate:
  - ▶ construct a digital filter
  - ▶ convolve the image with that filter

From: Spencer, G. et al. 1995. Proc. of SIGGRAPH. (1995)

242

### Selective application of glare

- ▶ A) Glare applied to the entire image  
 $I_g = I * G$  (Glare kernel (PSF))  
 Reduces image contrast and sharpness
- ▶ B) Glare applied only to the clipped pixels  
 $I_g = I + I_{clipped} * G - I_{clipped}$   
 where  $I_{clipped} = \begin{cases} I & \text{for } I > 1 \\ 0 & \text{otherwise} \end{cases}$   
 Better image quality

▶ 243

### Selective application of glare

Original image

- ▶ A) Glare applied to the entire image
- ▶ B) Glare applied to clipped pixels only

▶ 244

### Glare (or bloom) in games

- ▶ Convolution with large, non-separable filters is too slow
- ▶ The effect is approximated by a combination of Gaussian filters
  - ▶ Each filter with different "sigma"
- ▶ The effect is meant to look good, not be accurate model of light scattering
- ▶ Some games simulate a camera rather than the eye

▶ 245

### References: Tone-mapping

- ▶ Tone-mapping
  - ▶ REINHARD, E., HEIDRICH, W., DEBEVEC, P., PATTANAIK, S., WARD, G., AND MYSZKOWSKI, K. 2010. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann.
  - ▶ MANTIUK, R.K., MYSZKOWSKI, K., AND SEIDEL, H. 2015. High Dynamic Range Imaging. In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 1–42.  
<http://www.cl.cam.ac.uk/~rkm38/pdfs/mantiuk15hdri.pdf> (Chapter 5)

▶ 246