Computer Graphics

Let there be Light

Konstantin Tretyakov
kt@ut.ee

Oct 16, 2013

Standard Graphics Pipeline

Vertex transform

- $x' = PVMx$
- **Perspective division**
- **Viewport transform**

Culling and clipping

Rasterization

Fragment shading

Visibility tests & blending

Standard Graphics Pipeline

- Vertex transform
- Culling and clipping
  - Normalized frustum culling
  - Back-face culling
  - Clipping
- Rasterization
- Fragment shading
- Visibility tests & blending

Standard Graphics Pipeline

Vertex transform

Culling and clipping

Rasterization

- Triangle rasterization
- Line rasterization
- Attribute interpolation

Fragment shading

Visibility tests & blending

Standard Graphics Pipeline

Vertex transform

Culling and clipping

Rasterization

Fragment shading

Visibility tests & blending

Blending

Z-buffer

Stencil buffer

Standard Graphics Pipeline

Vertex transform

Culling and clipping

Rasterization

Fragment shading

Visibility tests & blending

Next

Standard Graphics Pipeline

Vertex transform — **Determine clip-space position of a triangle**

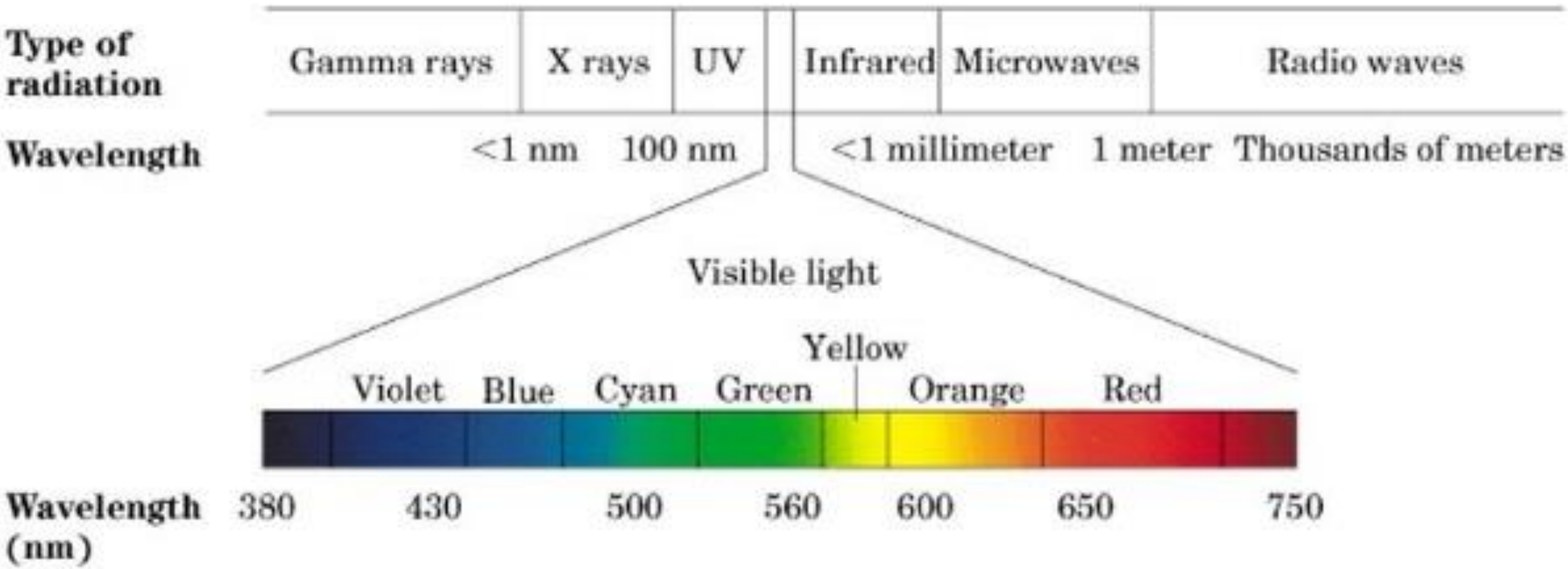
Culling and clipping — **Determine whether the triangle is visible**

Rasterization — **Determine all pixels belonging to the triangle**

Fragment shading — **For each pixel, determine its color**

Visibility tests & blending — **Draw pixel *(if needed)***

Light vs. Color

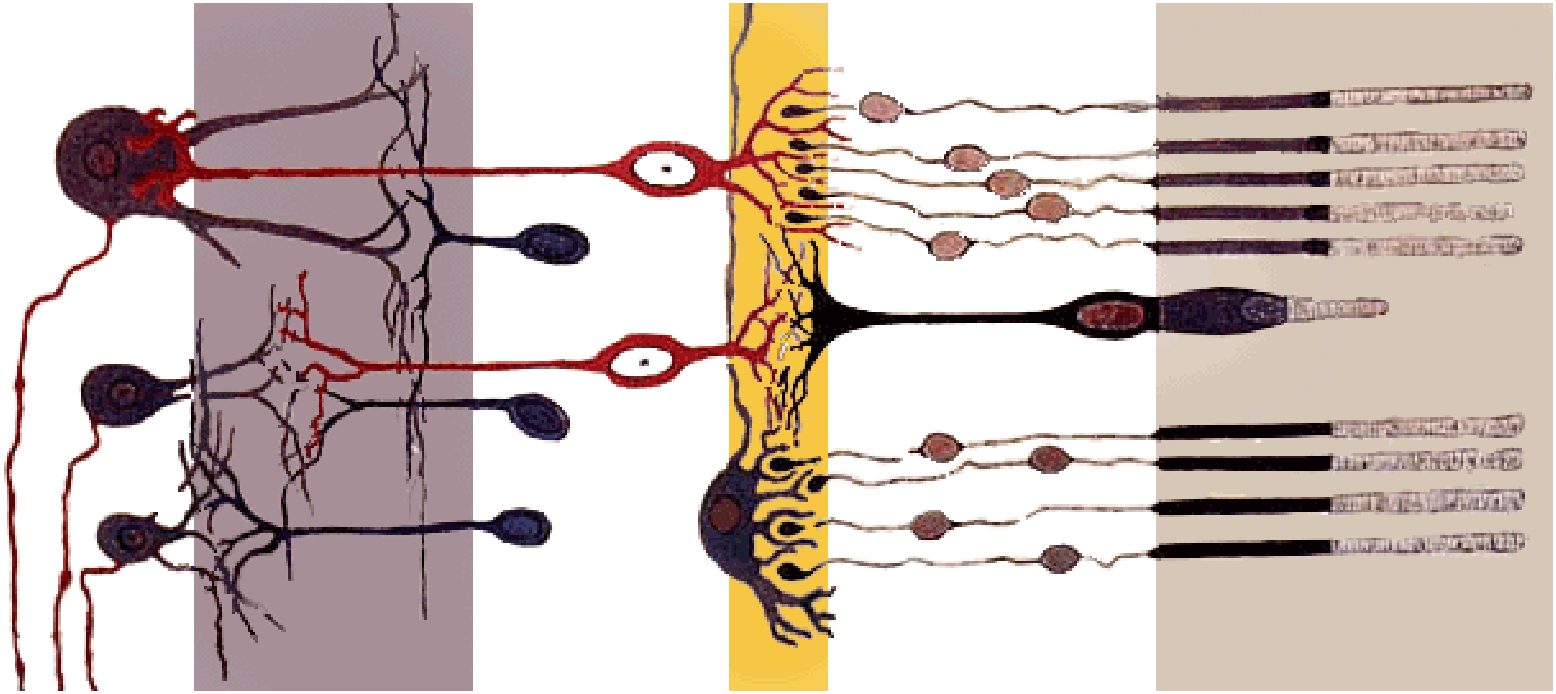


Light vs. Color

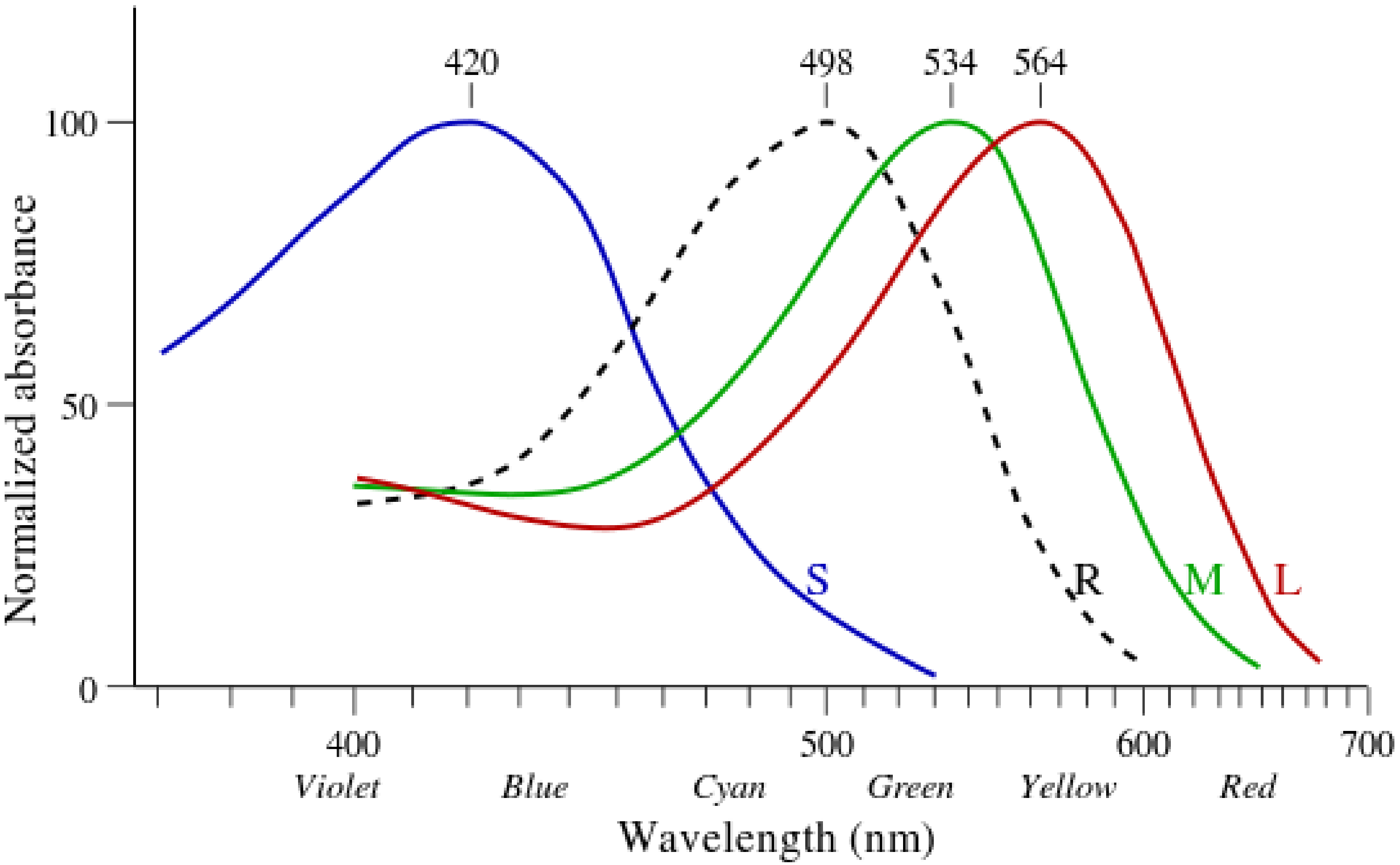
- In principle, a light wave can have a very complex spectrum (e.g. think how complex a sound wave can be).

- However, the receptor cells in our eyes only can only extract crude aggregates of this complex signal.

Rods & Cones



Rods & Cones



420
498
534
564
100
50
0
Normalized absorbance
S
R
M
L
400
500
600
700
Violet
Blue
Cyan
Green
Yellow
Red
Wavelength (nm)

Rods & Cones

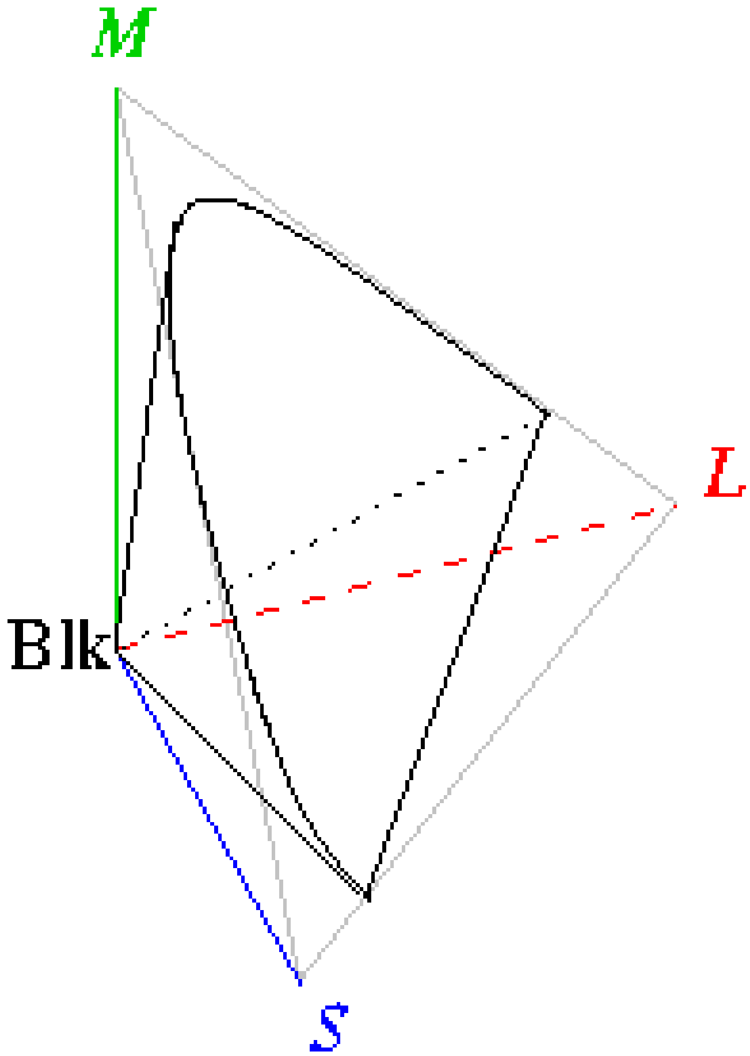
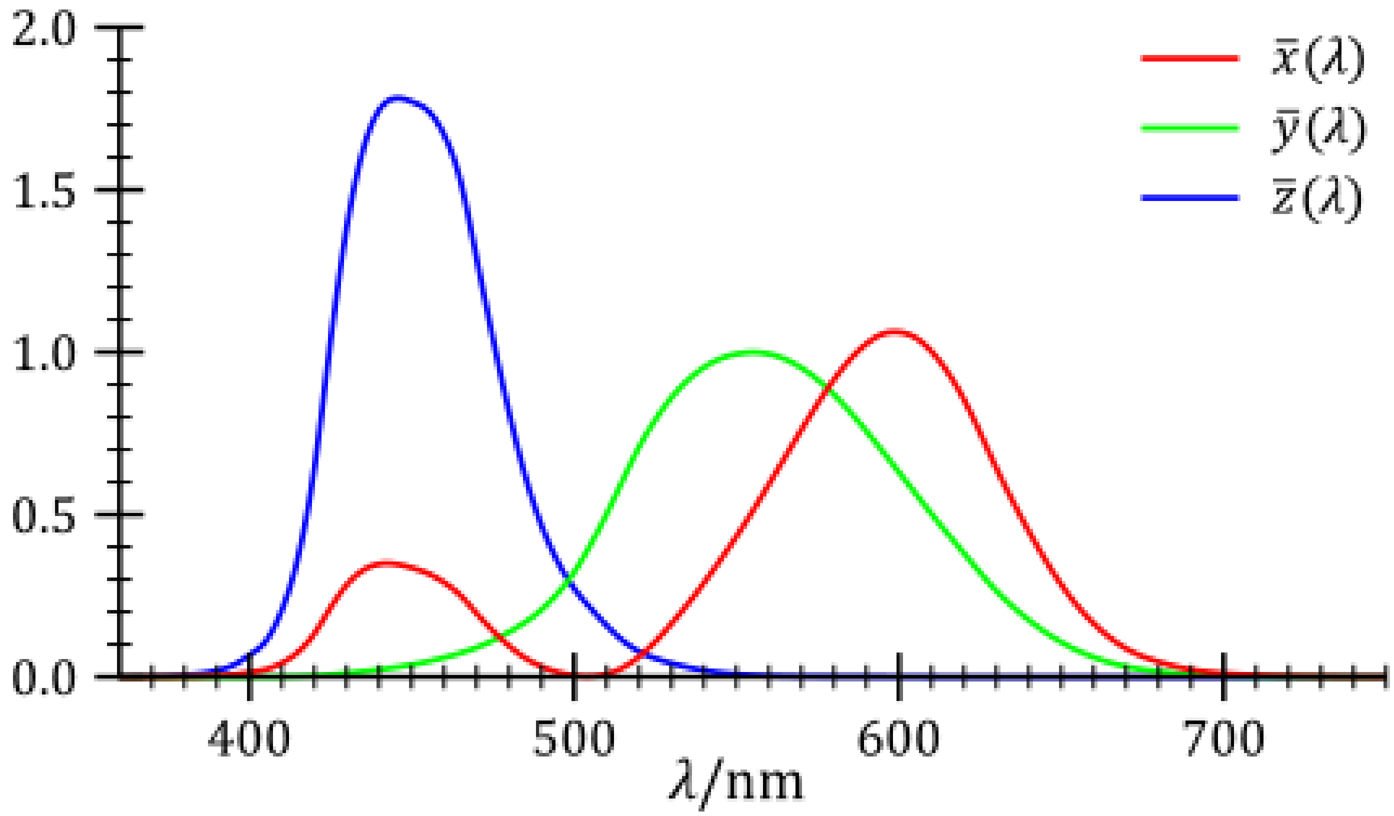


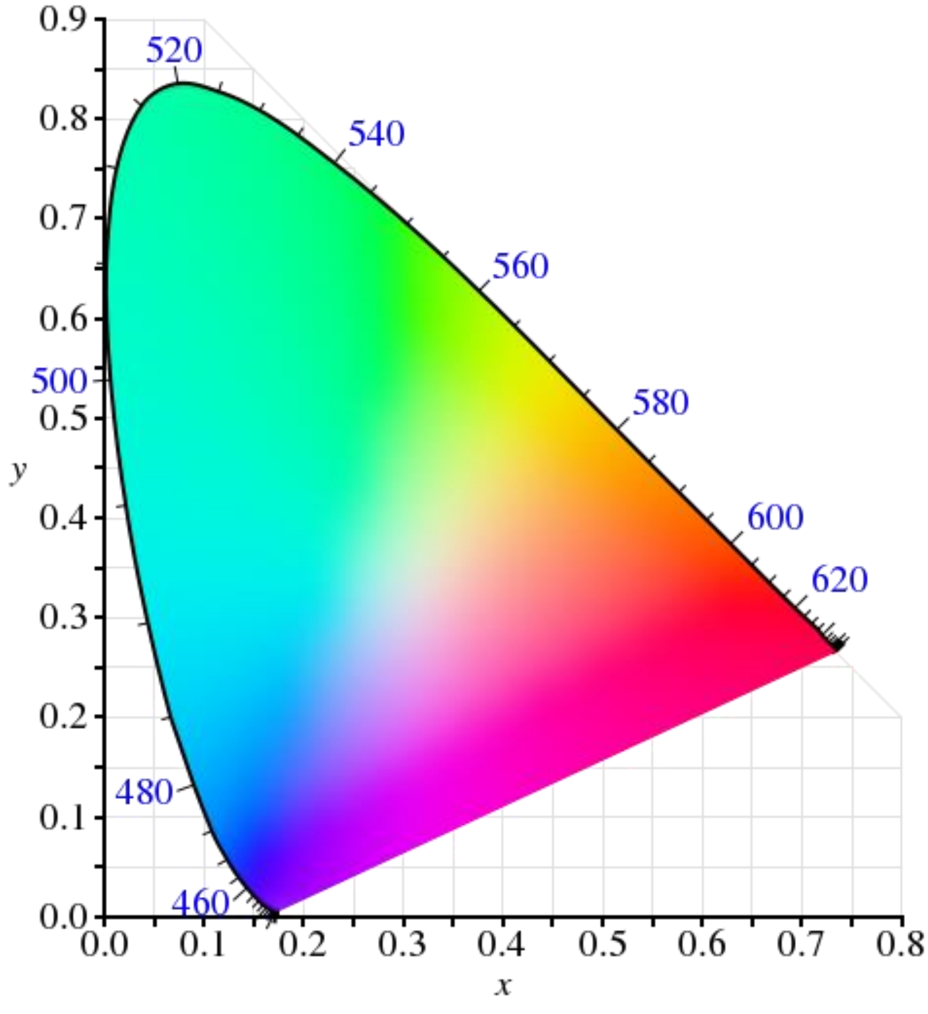
Image by Hankwang, http://en.wikipedia.org/wiki/File:Gamut_full.png

CIE 1931 XYZ

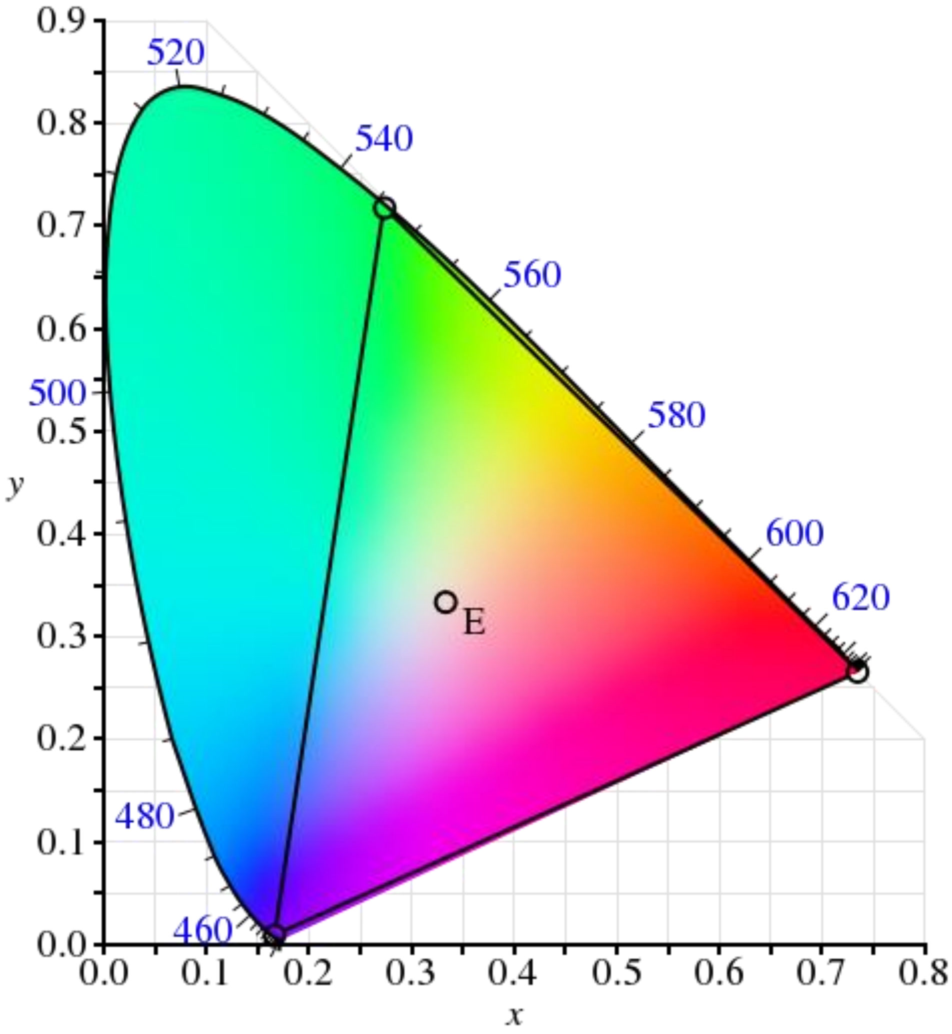


http://en.wikipedia.org/wiki/File:CIE_1931_XYZ_Color_Matching_Functions.svg

CIE 1931 XYZ, xyY

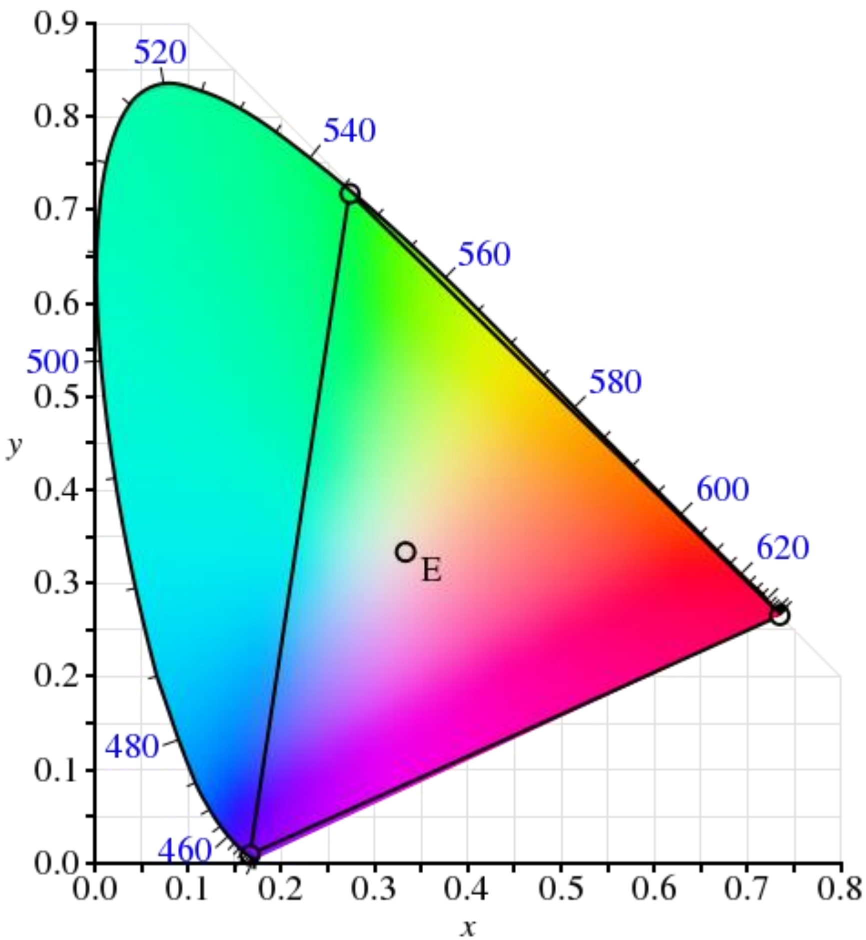


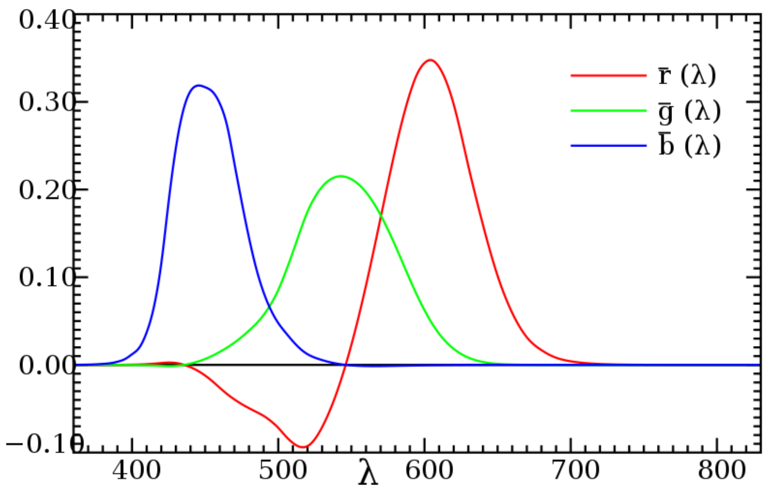
CIE RGB



0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
y
520
540
560
580
600
620
500
480
460
E
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
x

CIE RGB





Matching functions

Linearity of color spaces

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.41847 & -0.15866 & -0.082835 \\ -0.091169 & 0.25243 & 0.015708 \\ 0.00092090 & -0.0025498 & 0.17860 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},$$

Linearity of color spaces

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.41847 & -0.15866 & -0.082835 \\ -0.091169 & 0.25243 & 0.015708 \\ 0.00092090 & -0.0025498 & 0.17860 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},$$

A number of other spaces are obtained as linear transformations of XYZ:

HSV, CMYK, LAB, YUV, …

In this course we'll only deal with RGB.

Various RGB spaces

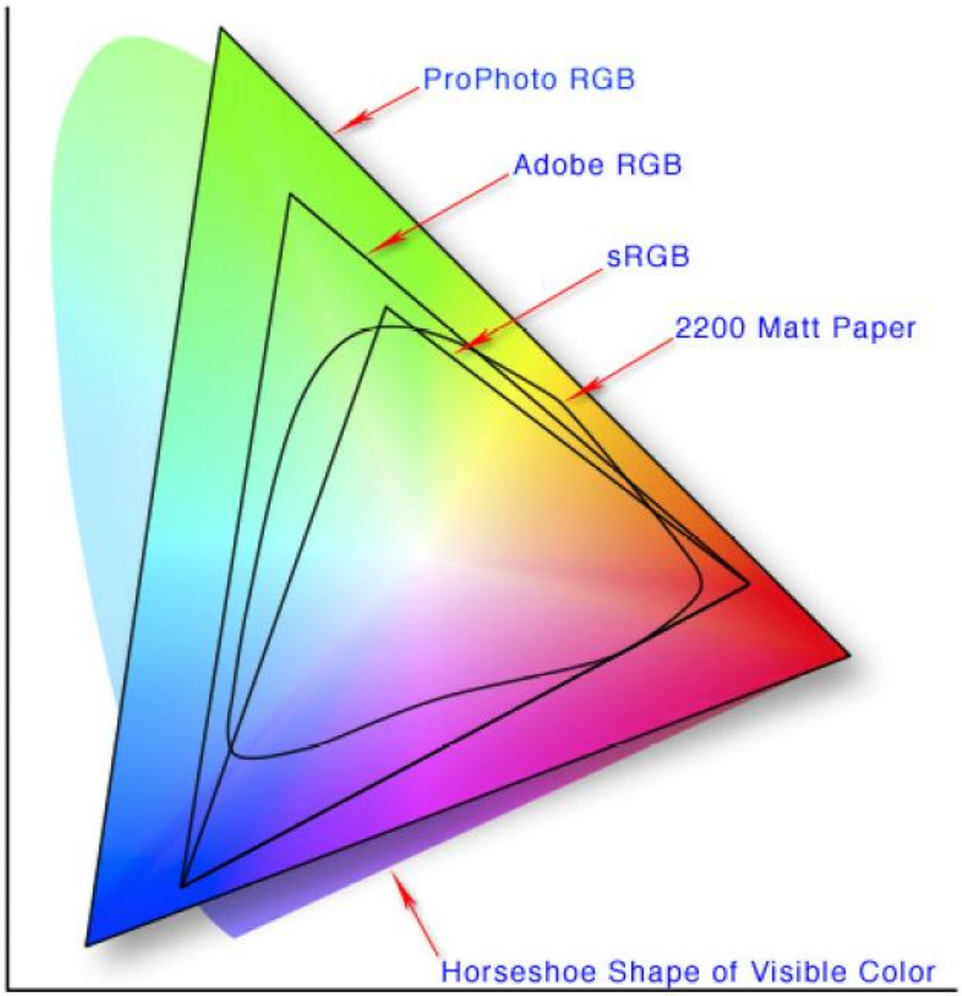


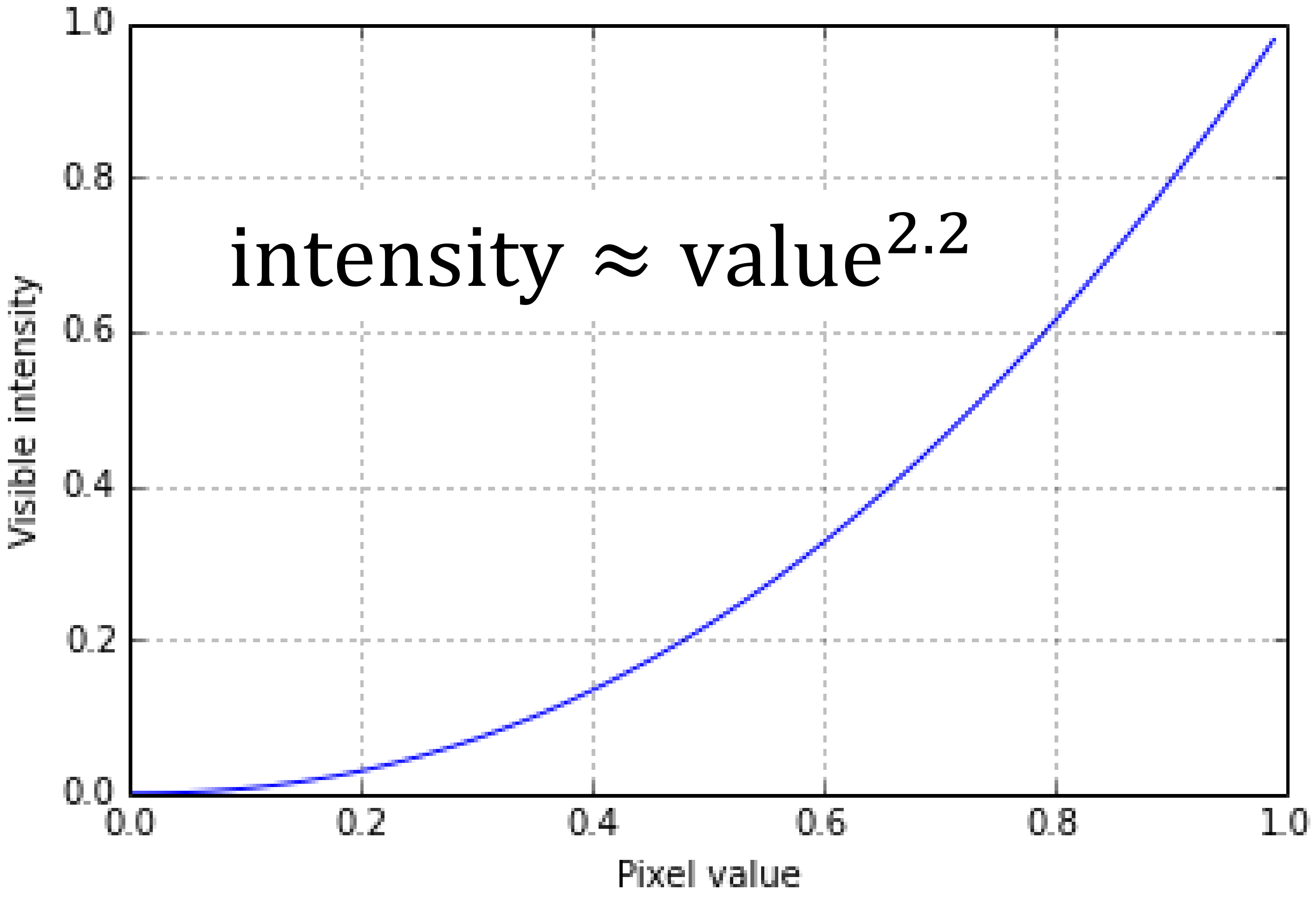
Image by by Jeff Schewe, http://en.wikipedia.org/wiki/File:Colorspace.png

sRGB

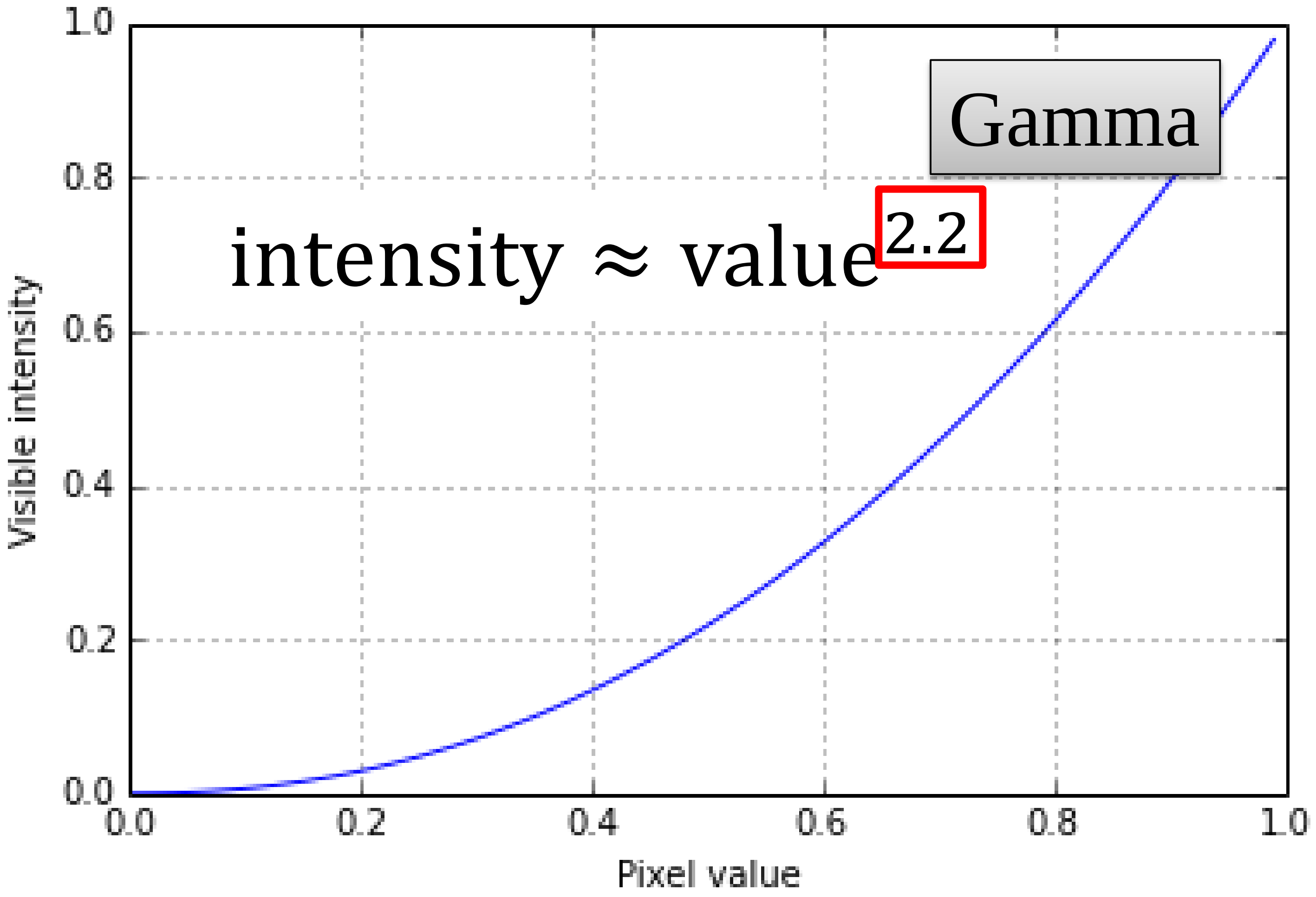
- CIE RGB model is *linear* with respect to light intensities: shining two 0.2 red lights together produces a 0.4 red light.

- However, original CRT monitors were *not* linear in how they work: sending a signal twice as strong would not double light intensity.

CRT response curve



CRT response curve



sRGB

- This led to the standardization of a (non-linear) **sRGB (“standard RGB”)** color space, corresponding to the “typical CRT”.

- CIE RGB $\sim$ $\mathrm{sRGB}^{2.2}$

* Actual transformation slightly more complicated: see http://en.wikipedia.org/wiki/SRGB

sRGB

- This led to the standardization of a (non-linear) **sRGB (“standard RGB”)** color spac… …CRT”.
- CIE I…

Nearly all modern digital display and imaging devices (cameras, monitors, TVs, printers, scanners, etc) use sRGB.

* Actual transformation slightly more complicated: see http://en.wikipedia.org/wiki/SRGB

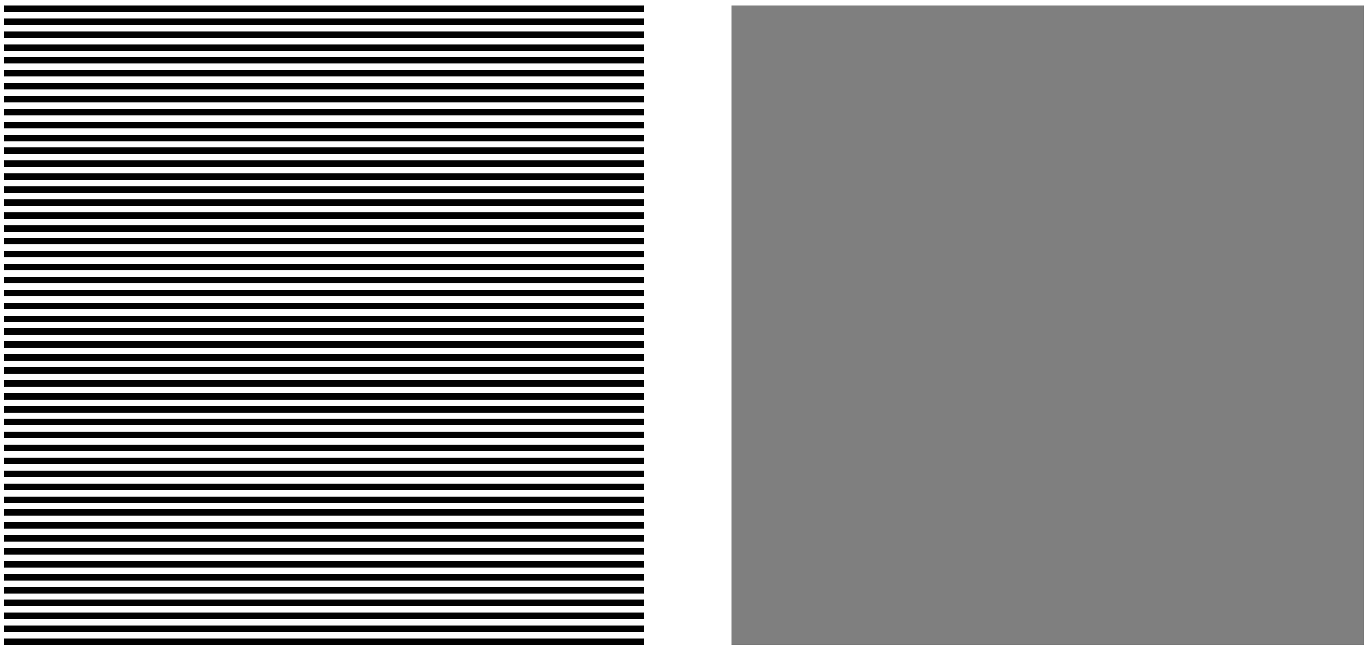
sRGB

- This led to the standardization of a (non-linear) **sRGB (“standard RGB”)** color spa … T”.
- CI

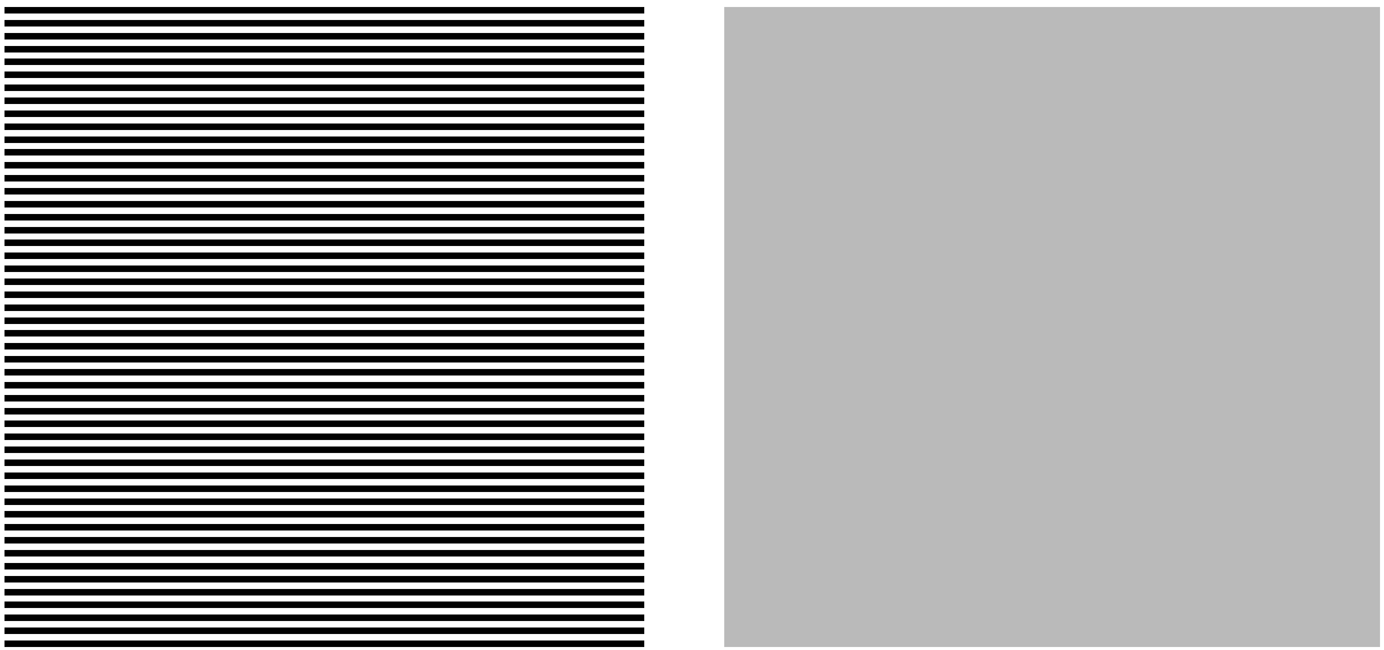
Most images are saved in sRGB

* Actual transformation slightly more complicated: see http://en.wikipedia.org/wiki/SRGB

RGB(0.5) vs. sRGB(0.5)



RGB(0.5) vs. sRGB(0.73)



Gamma correction

- The process of converting from linear intensities to sRGB is called ***gamma correction*** or ***gamma encoding***.

- sRGB ~ CIE RGB$^{\frac{1}{2.2}}$

- The inverse operation is called **gamma decoding**

Light modeling

- When rendering an object, we would like to compute the color of a pixel so that it would imitate physical processes of light scattering from the object.

Light modeling

- When rendering an object, we would like to compute the color of a pixel so that it would imitate physical processes of light scattering from the object.

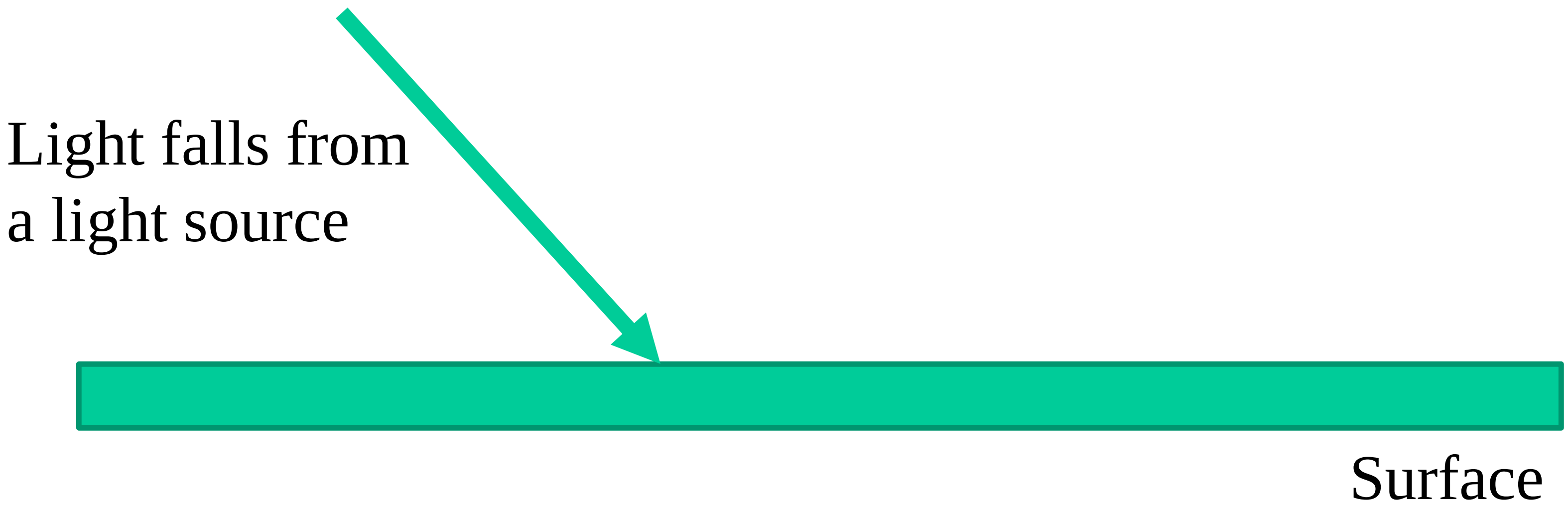
- The actual physics of light is hard to compute. Thus, we shall be using fake approximations instead.

Light modeling

Surface

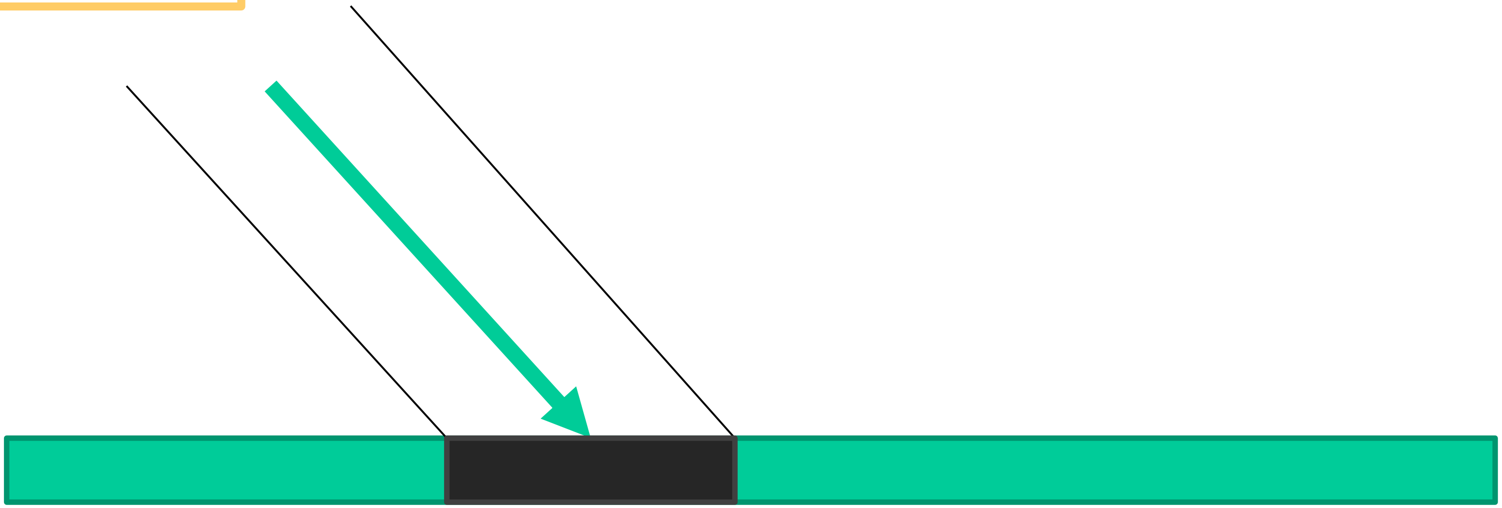
Light modeling

Light source
with intensity *I*



Light modeling

Light source
with intensity I

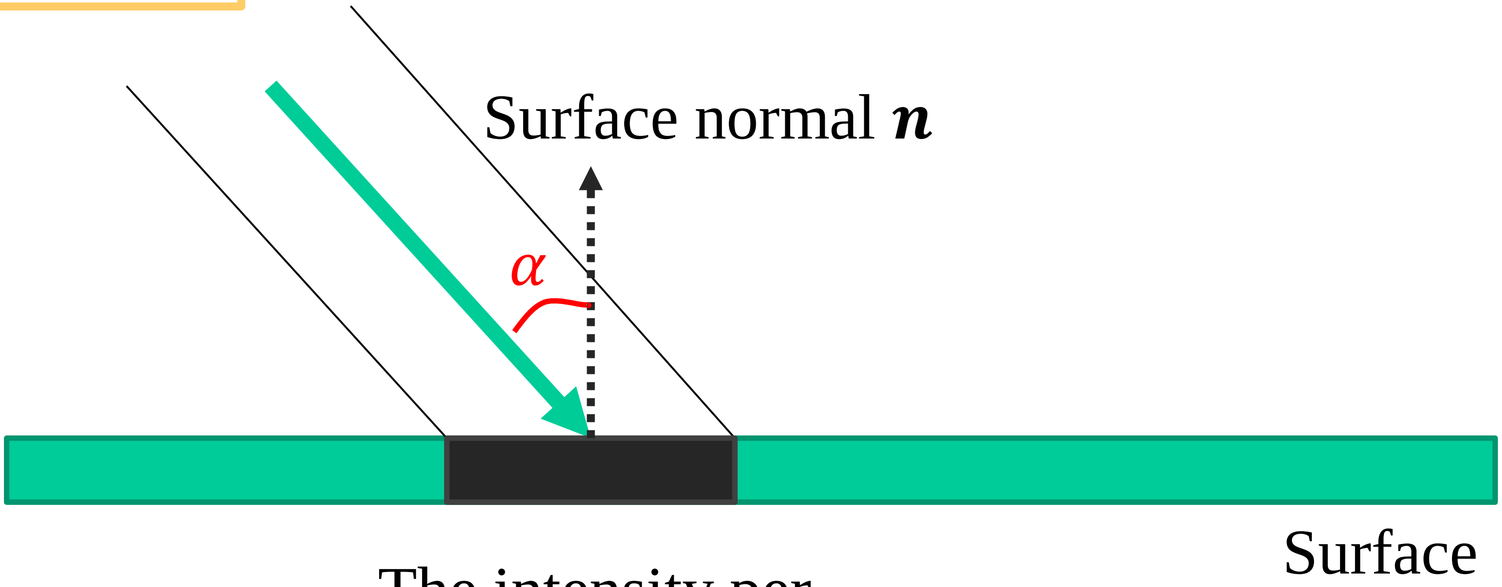


Surface

The more penpendicular
is the light, the greater is its
intensity per area unit.

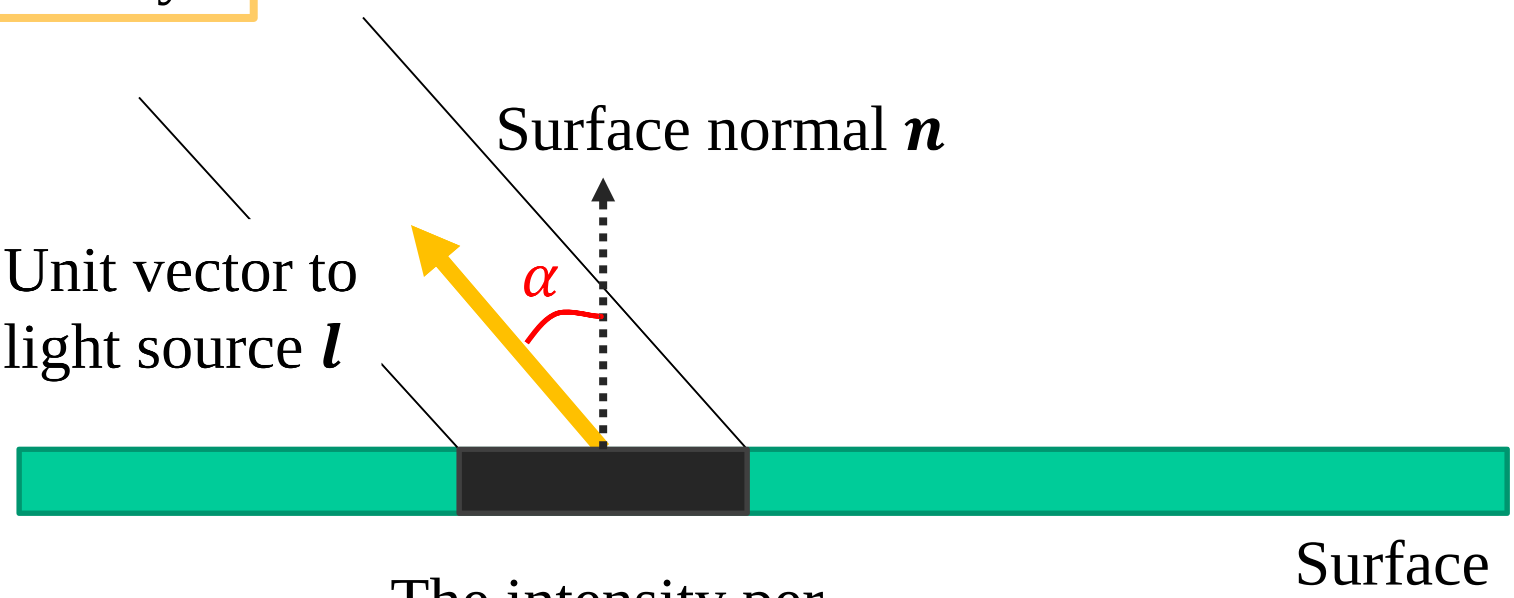
Light modeling

Light source with intensity I



The intensity per unit area scales proportionally to $\cos(\alpha)$

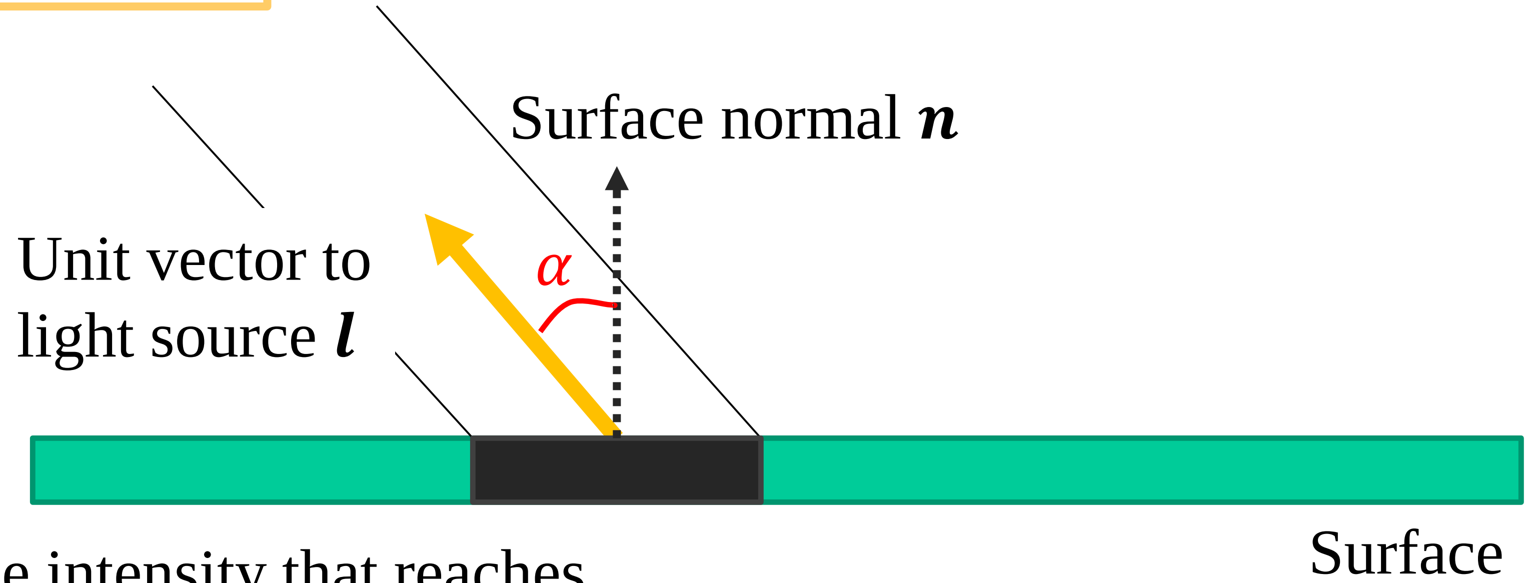
Light modeling



The intensity per unit area scales proportionally to $\cos(\alpha) = \boldsymbol{l}^T\boldsymbol{n}$

Light modeling

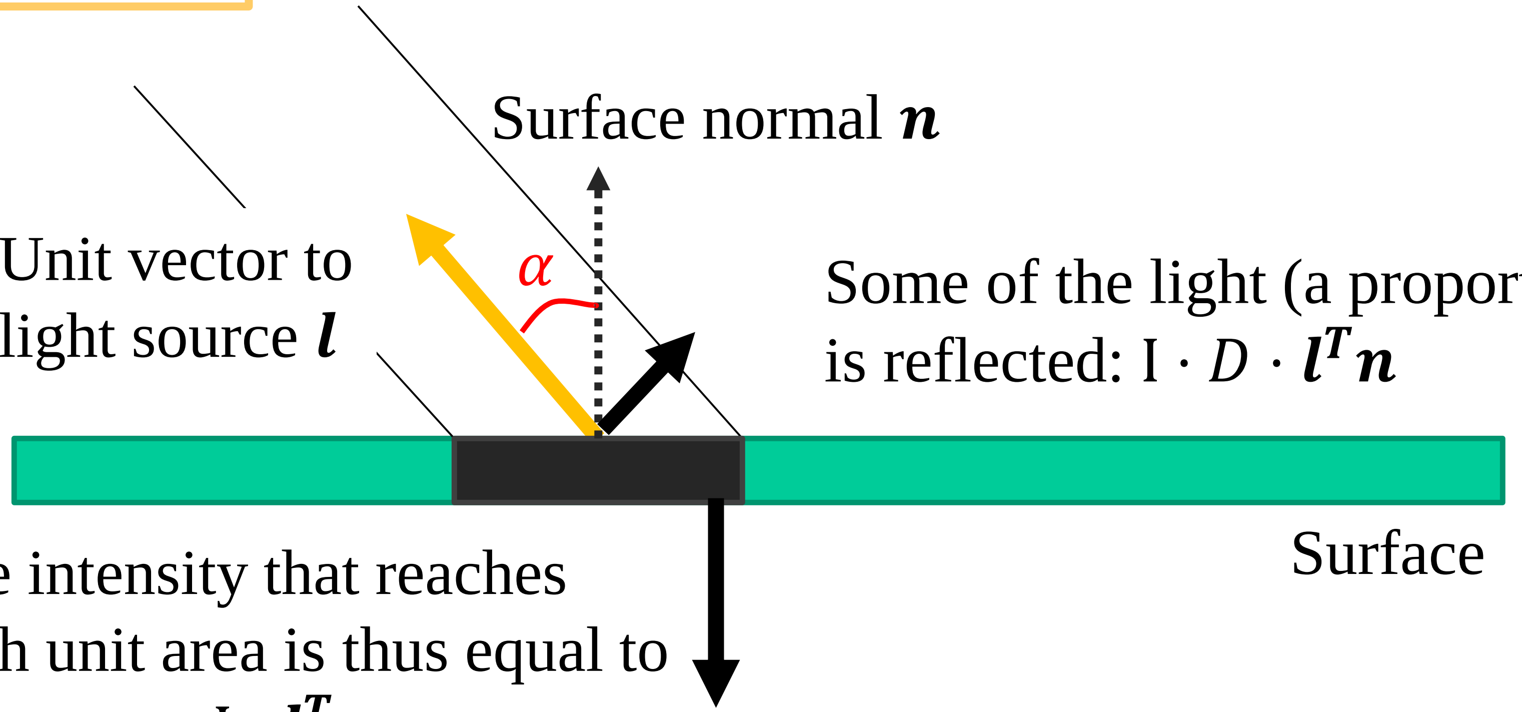
Light source with intensity I



The intensity that reaches each unit area is thus equal to

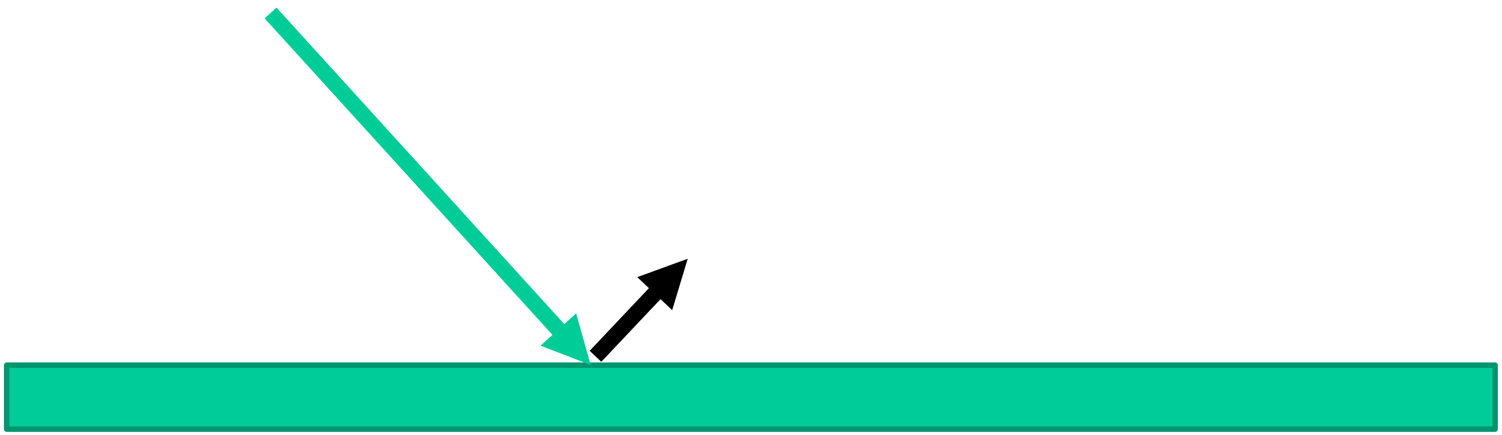
$$\mathrm{I} \cdot \boldsymbol{l}^T \boldsymbol{n}$$

Light modeling



Light source
with intensity *I*

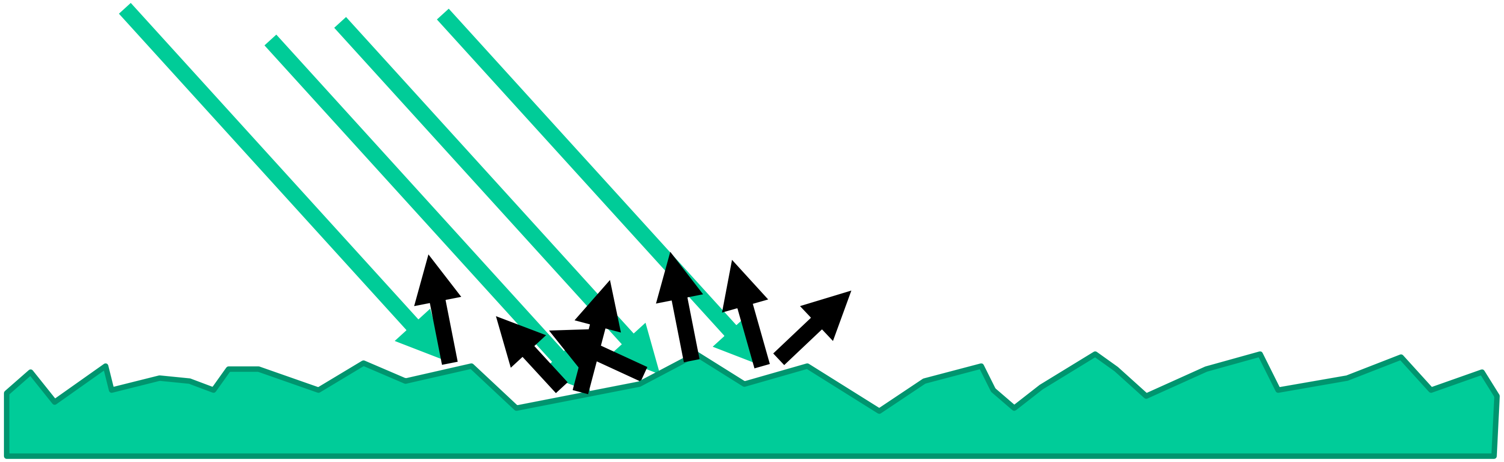
Specular reflection



From a perfectly smooth surface,
light is reflected ideally

Light source with intensity *I*

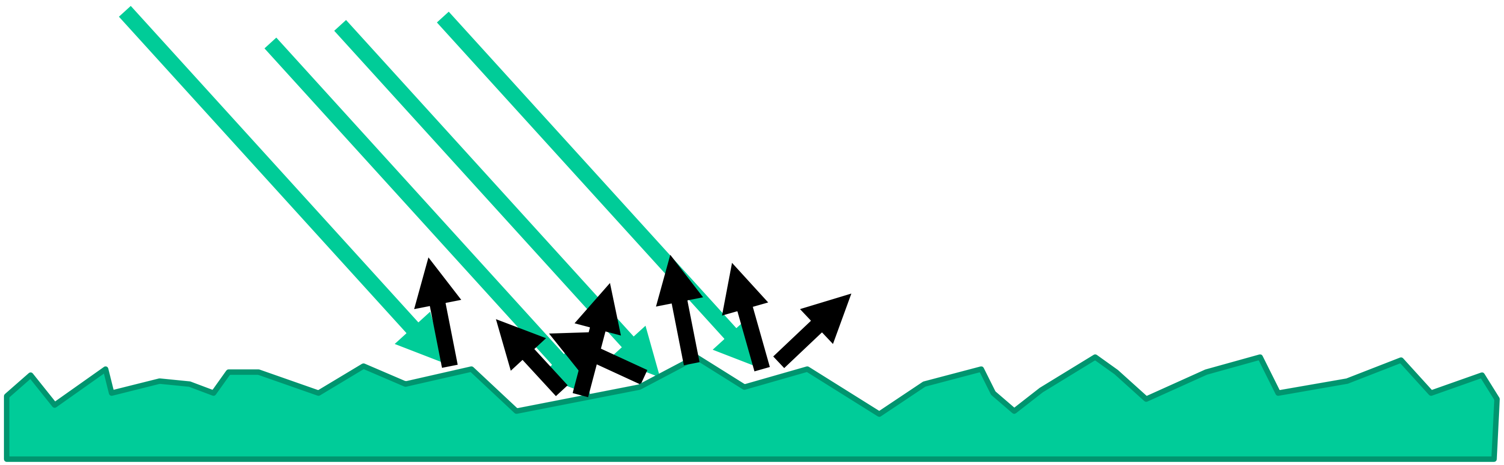
Diffuse reflection



Most surfaces are not perfectly smooth, and can be regarded as consisting of *microfacets* (each of which is a tiny perfect reflector)

Diffuse reflection

Light source
with intensity *I*

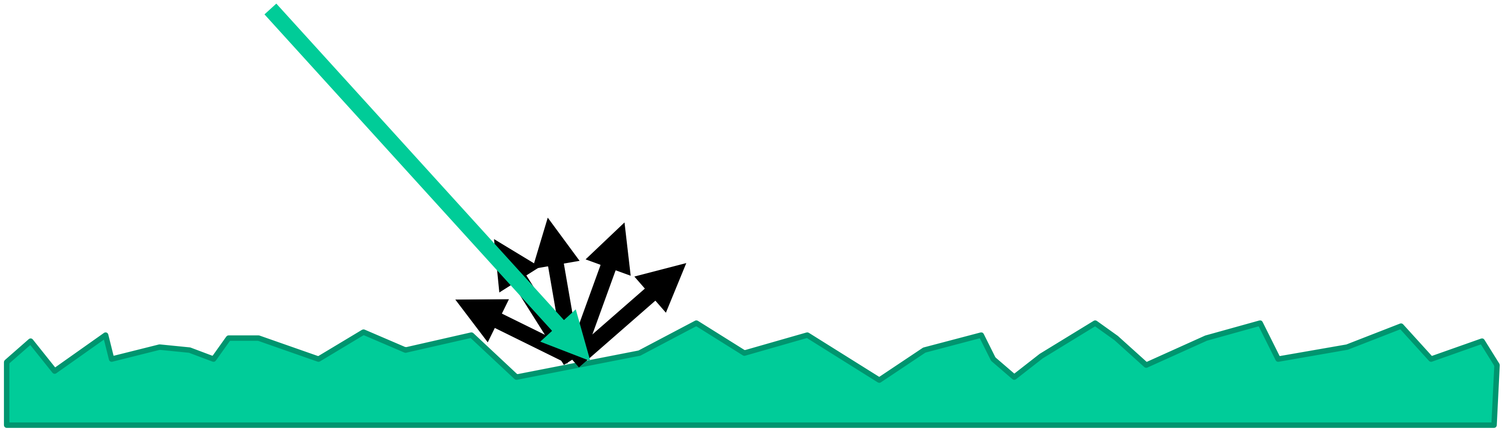


The type of reflection diffusion depends on the distribution of microfacets.

An “ideally” diffusing surface will diffuse light equally in all directions.

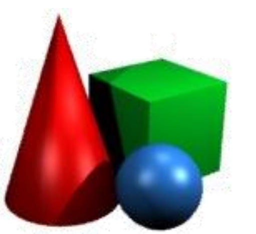
Diffuse reflection

Light source
with intensity I



The type of reflection diffusion depends on the distribution of microfacets.

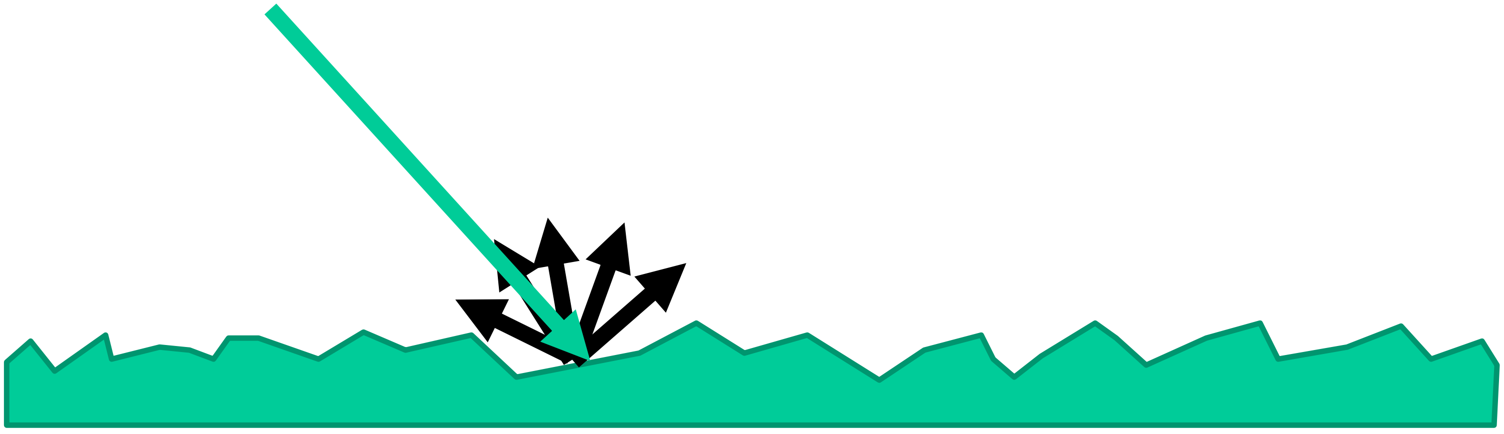
An "ideally" diffusing surface will diffuse light equally in all directions.



* Formally, it is not "equally in all directions", see http://en.wikipedia.org/wiki/Lambert's_cosine_law

Light source with intensity I

Lambertian reflection



Lambertian (diffuse) reflection:

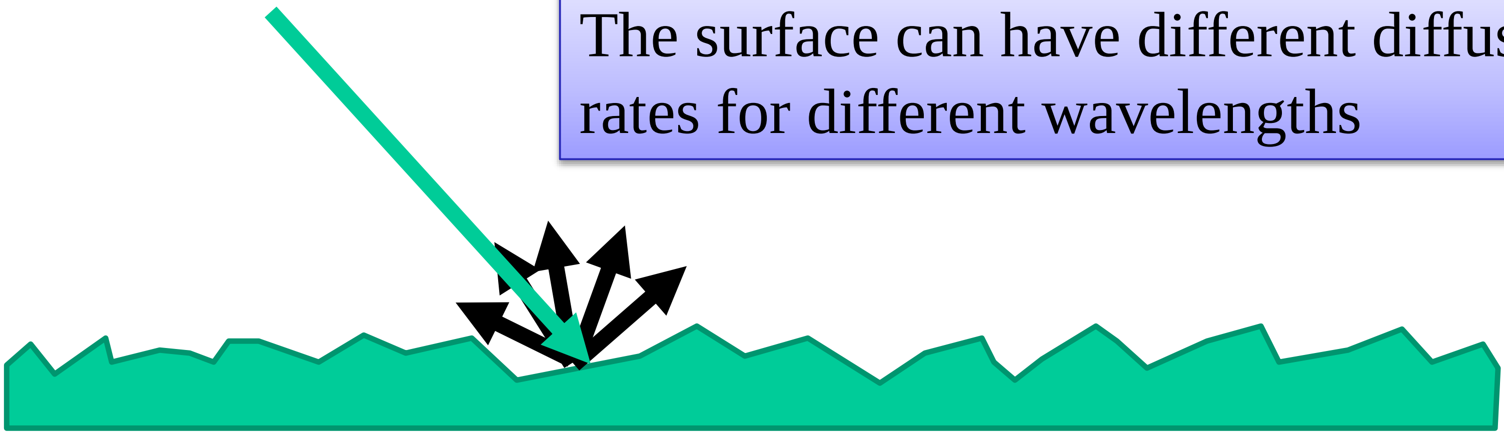
The amount of light reaching the observer is

$$I \cdot D \cdot (\boldsymbol{l}^T \boldsymbol{n})$$

Light source with intensity *I*

Lambertian reflection

The surface can have different diffusion rates for different wavelengths



Lambertian (diffuse) reflection:

The amount of light reaching the observer is

$$I \cdot D \cdot (\boldsymbol{l}^T \boldsymbol{n})$$

Lambertian reflection

Light source with intensity I

We specify separate diffusion coefficient for each color component:
(D_R, D_G, D_B)

Lambertian (diffuse) reflection:

The amount of light reaching the observer is

$$I \cdot D \cdot (\boldsymbol{l}^T \boldsymbol{n})$$

Lighting in OpenGL

The old-school way:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

float direction[] = {0, 0, 1, 0};
glLightfv(GL_LIGHT0, GL_POSITION, direction);

float intensity[] = {1, 1, 1, 1};
glLightfv(GL_LIGHT0, GL_DIFFUSE, intensity);
```

* Of course it’s deprecated, but it is nice & easy for simpler cases.

Lighting in OpenGL

```
float material[] = {1, 0, 0, 1};
glMaterialfv(GL_FRONT, GL_DIFFUSE, material);
```

or

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColor3f(1, 0, 0);
```

Lighting in OpenGL

```
glShadeModel(GL_SMOOTH);
```

or

```
glShadeModel(GL_FLAT);
```

Lighting in OpenGL

```
glShadeModel(GL_SMOOTH);
```

NB: "Old-style" OpenGL does **not** do gamma correction in lighting computations

* Unless your graphics card supports EXT_framebuffer_sRGB, most don't so far.

Lighting in OpenGL

- The new school-way: using GLSL.

Vertex shader

```
smooth out vec4 vertex_color;
uniform vec3 light_dir;
uniform vec4 diffuse_l, diffuse_m;

void main(void) {
  float c = clamp(dot(gl_Normal, light_dir), 0, 1);
  vertex_color = diffuse_l * diffuse_m * c;
  gl_Position = ftransform();
}
```

Fragment shader

```
in vec4 vertex_color;

void main(void) {
  glFragColor = vertex_color;
}
```

Lighting in OpenGL

- The new school-way: using GLSL.

Vertex shader

```
smooth out vec4 vertex_color;
uniform vec3 light_dir;
uniform vec4 diffuse_l, diffuse_m;

void main(void) {
  float c = clamp(dot(gl_Normal, l
  vertex_color = diffuse l * diffuse m * c;
  vertex_color = pow(vertex_color, 1.0/2.2);
  gl_Position = ftransform();
}
```

Gamma correction

Fragment shader

```
in vec4 vertex_color;

void main(void) {
  glFragColor = vertex_color;
}
```

Per-vertex vs per-fragment lighting

- **Per-vertex lighting**: perform lighting computations in the vertex shader, assign resulting colors to vertices, and simply interpolate them.

- **Per-fragment lighting**: perform lighting computations in the fragment shader.

Per-vertex vs per-fragment lighting

- **Per-vertex lighting**: perform lighting computations in the vertex shader, assign resulting colors to vertices, and simply int

NB: "Old-style" OpenGL can only do per-vertex lighting.

- **Per-fragment lighting**: perform lighting computations in the fragment shader.

Ambient light

- To deal with unlit surfaces, we introduce the average level of “ambient” light. It simulates reflections scattered from all over.

- The total light intensity is then:

$$I_A \cdot A + I_D \cdot D \cdot \boldsymbol{l}^T \boldsymbol{n}$$

Ambient light in OpenGL

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0, GL_POSITION, direction);

glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_l);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient_l);

glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse_m);
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient_m);
```

Ambient light in OpenGL

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0, GL_POSITION, direction);

glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_l);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient_l);

glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glColor3f(...);
```

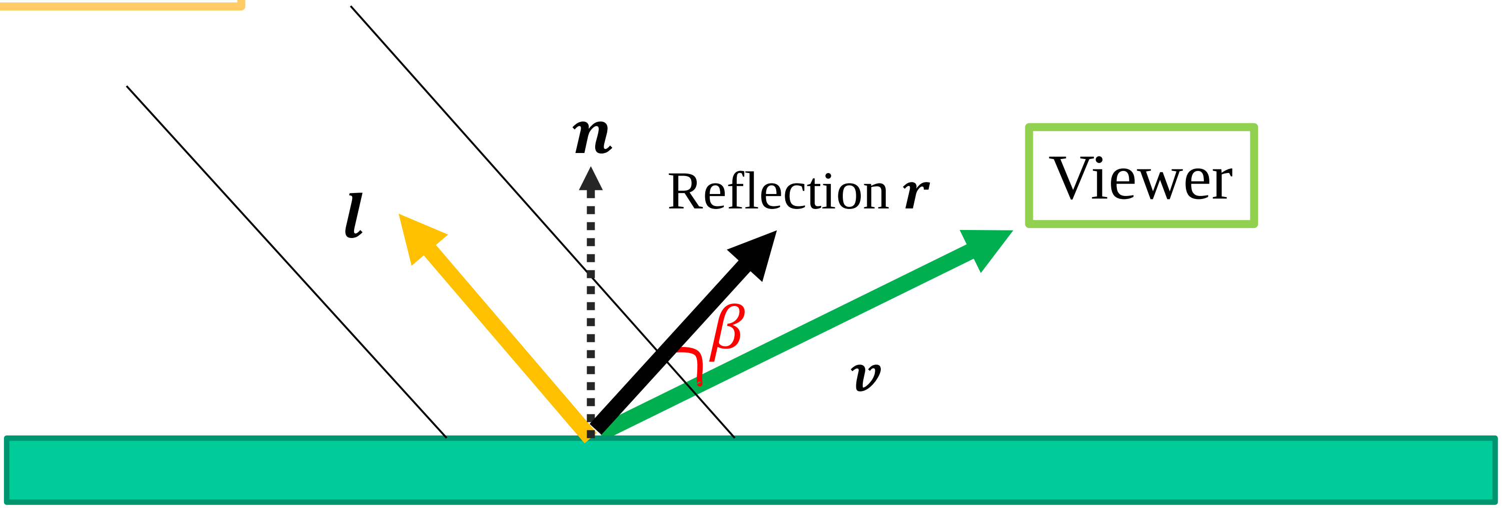
Ambient light in OpenGL

Vertex shader

```
smooth out vec4 vertex_color;
...
uniform vec3 ambient_l, ambient_m;

void main(void) {
  float c = clamp(dot(gl_Normal, light_dir), 0, 1);

  vertex_color = ambient_l*ambient_m +
                 diffuse_m*diffuse_l*c;
  vertex_color = pow(vertex_color, 1.0/2.2);

  gl_Position = ftransform();
}
```

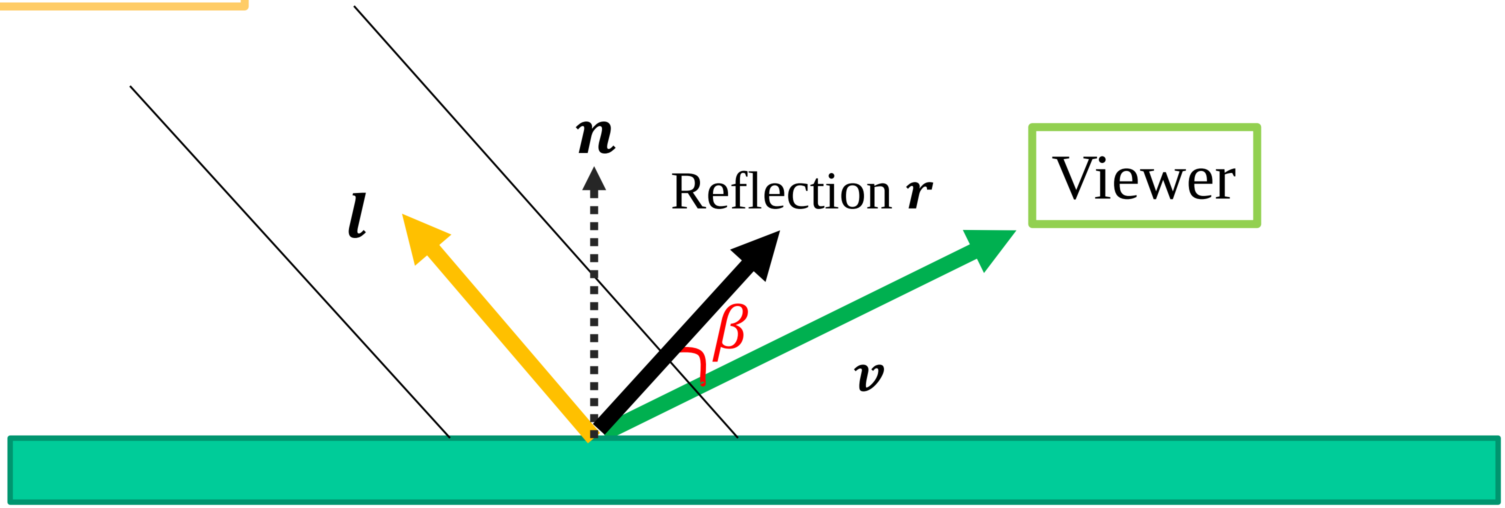
Light source
with intensity I

Specular reflection



When the viewer is watching in a direction close to that of the reflected light, i.e. when β is small, he should see an intense highlight

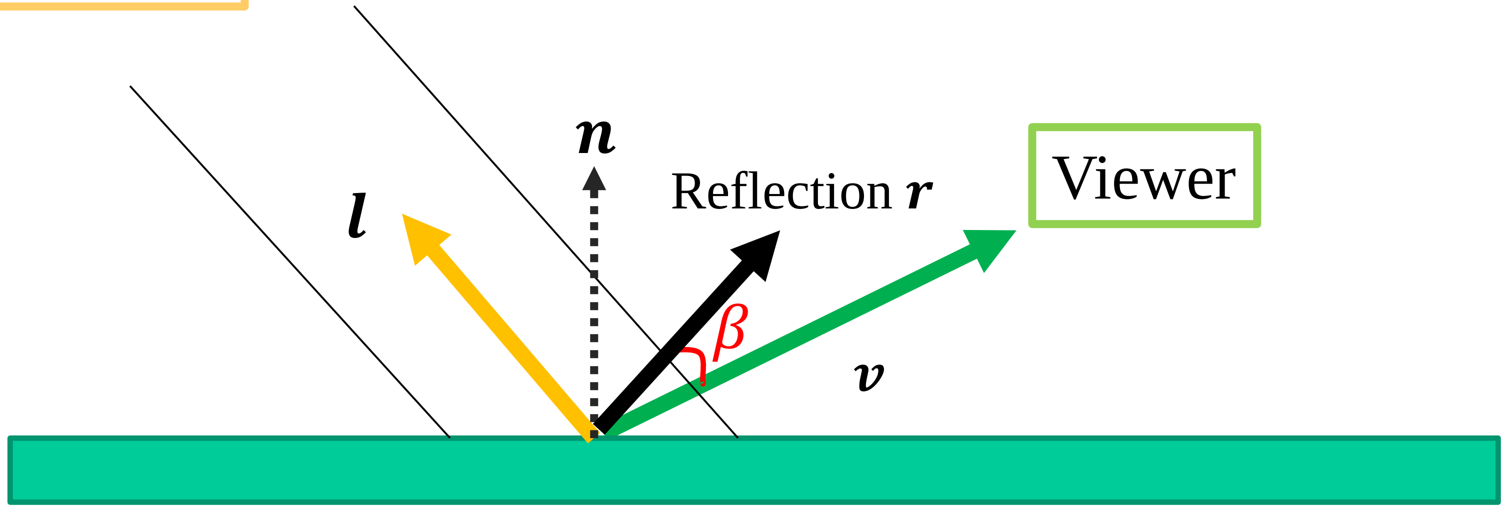
Specular reflection



How much the highlight falls off with β depends on the distribution of microfacets.

Specular reflection

Light source with intensity I



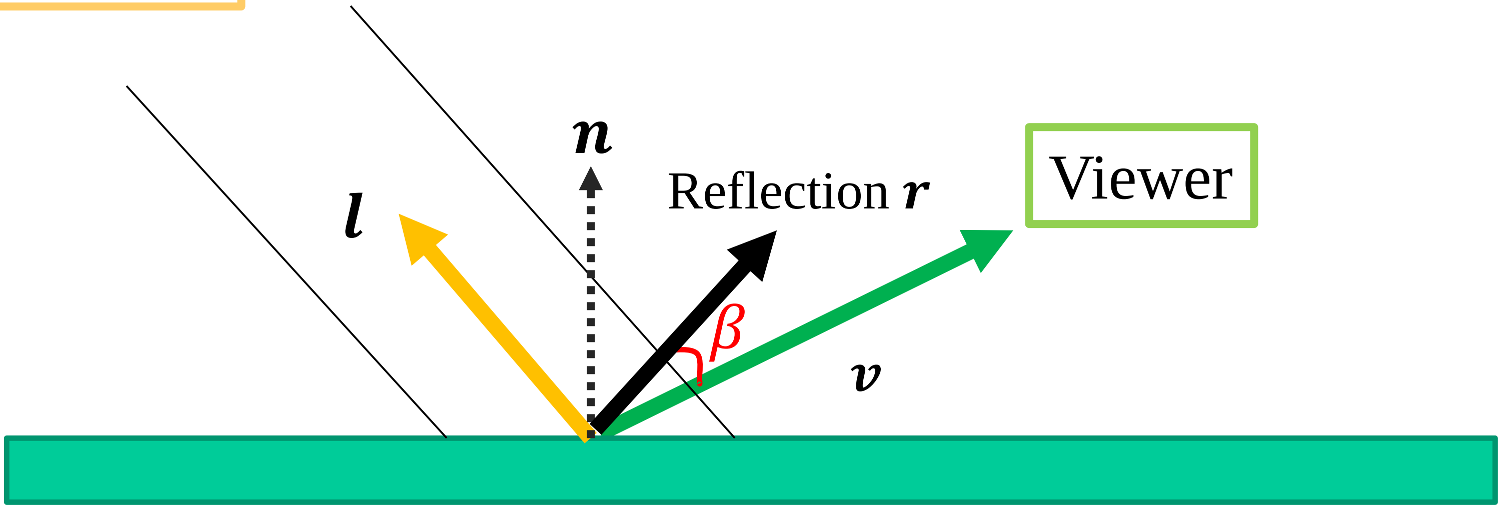
How much the highlight falls off with β depends on the distribution of microfacets.

A Gaussian curve is believed to be a good approximation:

$$\text{intensity of specular reflection} \approx e^{-\left(\frac{\beta}{m}\right)^2}$$

Phong reflection

Light source with intensity I



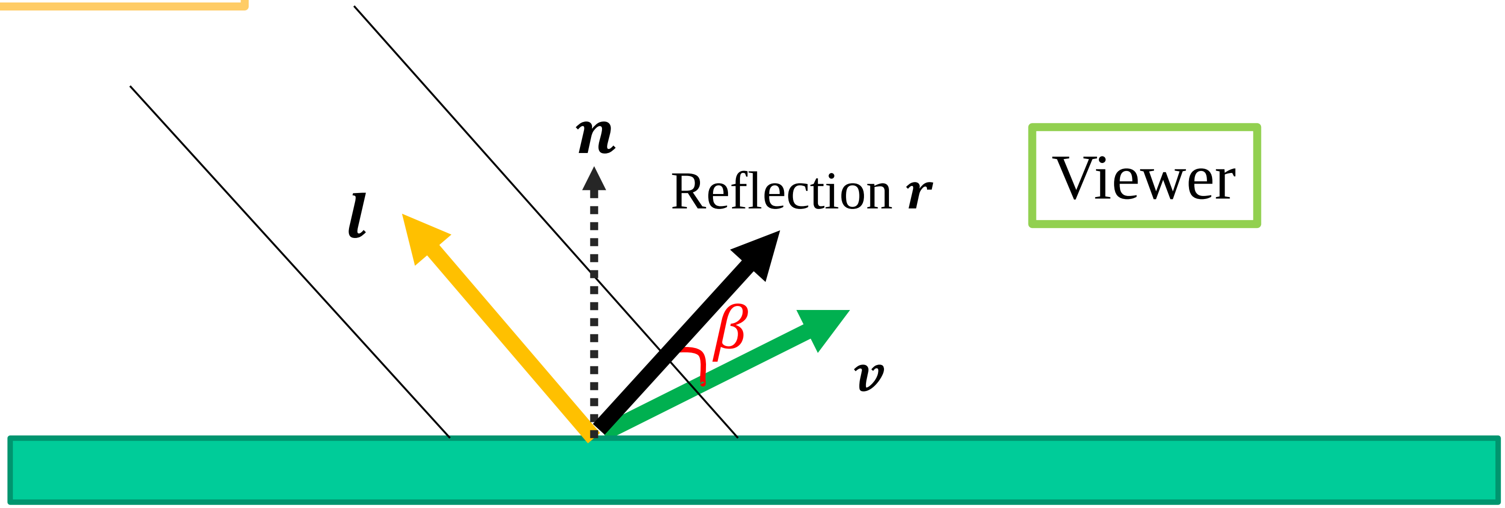
How much the highlight falls off with β depends on the distribution of microfacets.

In practice, the following approximation (*Phong model*) is often used:

$$\text{intensity of specular reflection} \approx \cos(\beta)^s$$

Phong reflection

Light source
with intensity I



How much the highlight falls off with β depends on the distribution of microfacets.

In practice, the following approximation (Phong model) is often used:

$$\text{intensity of specular reflection} \approx (\boldsymbol{r}^T \boldsymbol{v})^s$$

Phong model

- The complete Phong lighting model is thus:

$$I_A \cdot A + I_D \cdot D \cdot \boldsymbol{l}^{\boldsymbol{T}}\boldsymbol{n} + I_S \cdot S \cdot \left(\boldsymbol{r}^{\boldsymbol{T}}\boldsymbol{v}\right)^s$$

Phong model

- The complete Phong lighting model is thus:

$$I_A \cdot A + I_D \cdot D \cdot \boldsymbol{l}^{\boldsymbol{T}}\boldsymbol{n} + I_S \cdot S \cdot \left(\boldsymbol{r}^{\boldsymbol{T}}\boldsymbol{v}\right)^{s}$$

- Note that objects often have different specular and diffuse colors (e.g. red plastic object has white specular highlights).

Lighting in OpenGL

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_l);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient_l);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular_l);

glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse_m);
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient_m);
glMaterialfv(GL_FRONT, GL_SPECULAR, specular_m);
glMaterialf(GL_FRONT, GL_SHININESS, 2.0);
```

Lighting in OpenGL

Vertex shader

```
smooth out vec4 position;
smooth out vec3 normal;

void main(void) {
  position = gl_ModelViewMatrix * gl_Vertex;
  normal = gl_NormalMatrix * gl_Normal;
  gl_Position = ftransform();
}
```

Lighting in OpenGL

Fragment shader

```
in vec4 position;
in vec3 normal;

uniform vec3 l; // Direction to light
uniform vec3 v; // Direction to camera
// Light and material params
uniform vec3 a_l, a_m, d_l, d_m, s_l, s_m;
uniform float shininess;
void main(void) {
    vec3 n = normalize(normal);
    float lambert = clamp(dot(n, l), 0, 1);
    vec3 r = n*2*dot(n, l) - l;
    float phong = clamp(dot(r, v), 0, 1);
    phong = pow(phong, shininess);

    gl_FragColor = a_l*a_m + d_l*d_m*lambert
                 + s_l*s_m*phong;
}
```

High Dynamic Range correction

- If you add many light components together, you may end up with color values exceeding 1.0.
- By default they are clamped to 1.0.
- In reality human, eye automatically normalizes for maximal brightness.

HDR correction

- For scenes with highly lit objects you should remap colors to fit the 0..1 range.
- The easiest option is simply divide the color by the potentially maximal value – this corresponds to tuning the “aperture” of the camera. It is known as *HDR correction*.

Point light sources

- So far we only considered *directional* light.

- If the light source is a point in space, we must attenuate its intensity based on the *distance* to light.

Point light sources

- Physically correct attenuation is achieved using the inverse square law:

$$\text{intensity at distance } d = \frac{I}{k + d^2}$$

Point light sources

- Sometimes using a more general form of attenuation results in a (physically less correct, but visually nicer) result:

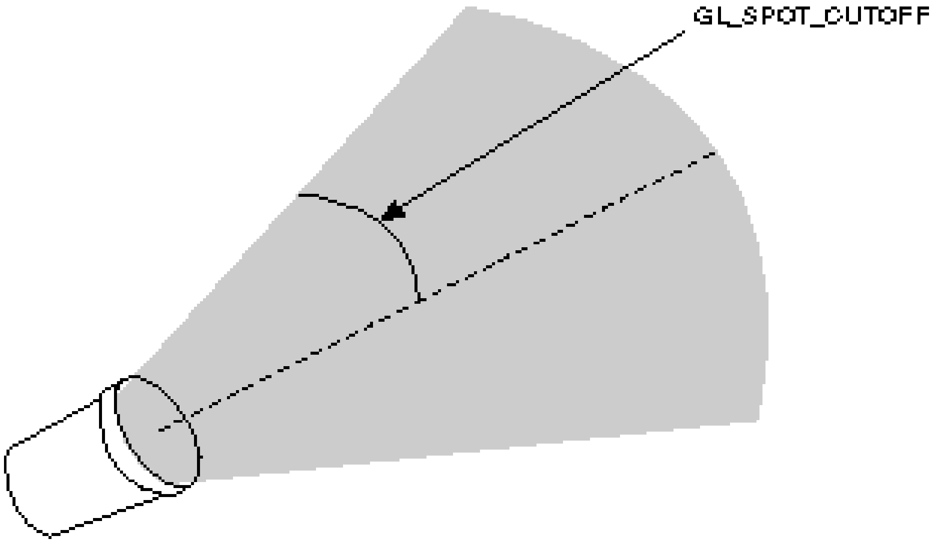
$$\mathrm{I}(d) \approx \frac{I}{k_c + k_\ell d + k_q d^2}$$

Where k_c, k_ℓ, k_q are carefully chosen constants.

Point light sources

```
float position[] = {-1, 2, 3, 1};
glLightfv(GL_LIGHT0, GL_POSITION, position);

glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION,
                     1.0);
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION,
                     1.0);
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION,
                     1.0);
```

Spotlights




```
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
glLightfv(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT0, GL_SPOT_EXPONENT, 1.0);
```

Summary

- Color perception
- CIE XYZ, CIE RGB
- sRGB, Gamma encoding / decoding
- Phong model =
  Ambient, Diffuse, Specular components
- Per-vertex vs Per-fragment lighting
- Fragment shaders
- Directional, point, spotlight light sources
- Gamma correction, HDR correction

Standard Graphics Pipeline

Vertex transform

Culling and clipping

Rasterization

Fragment shading

Visibility tests & blending