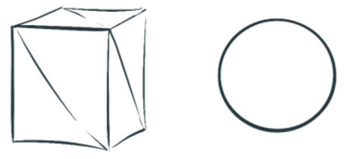
$$L_{\mathrm{o}}(x, \omega_{\mathrm{o}}) = L_{\mathrm{e}}(x, \omega_{\mathrm{o}}) + \int_{\Omega} L_{\mathrm{i}}(x, \omega_{\mathrm{i}}) f_{\mathrm{r}}(x, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}) |\cos\theta_{\mathrm{i}}| d\omega_{\mathrm{i}}$$

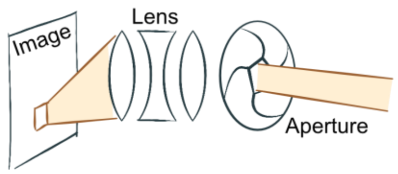
Light transport basics

Computer Graphics 24/25 – Lecture 2

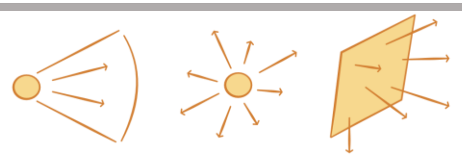
Last time



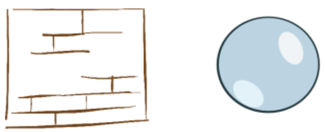
Geometries



Camera model

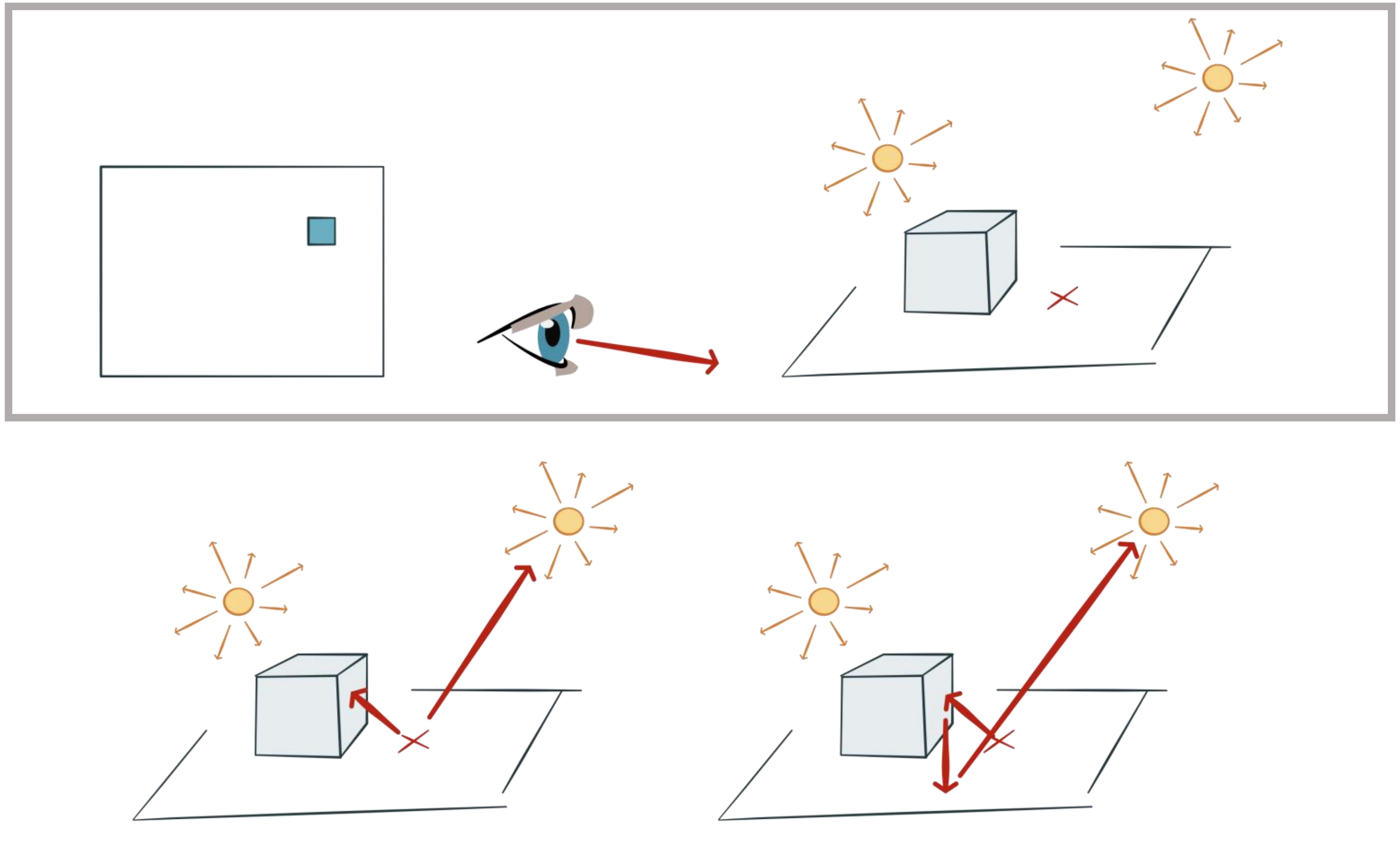


Light sources

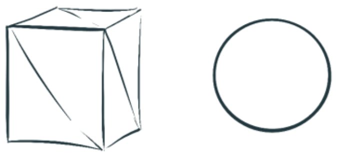


Materials & Textures

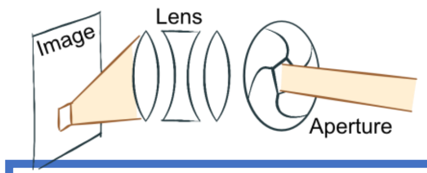
Volumes



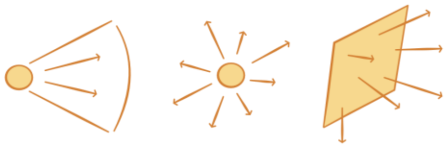
Today



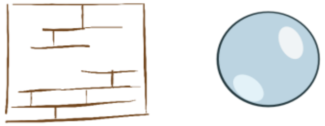
Geometries



Camera model

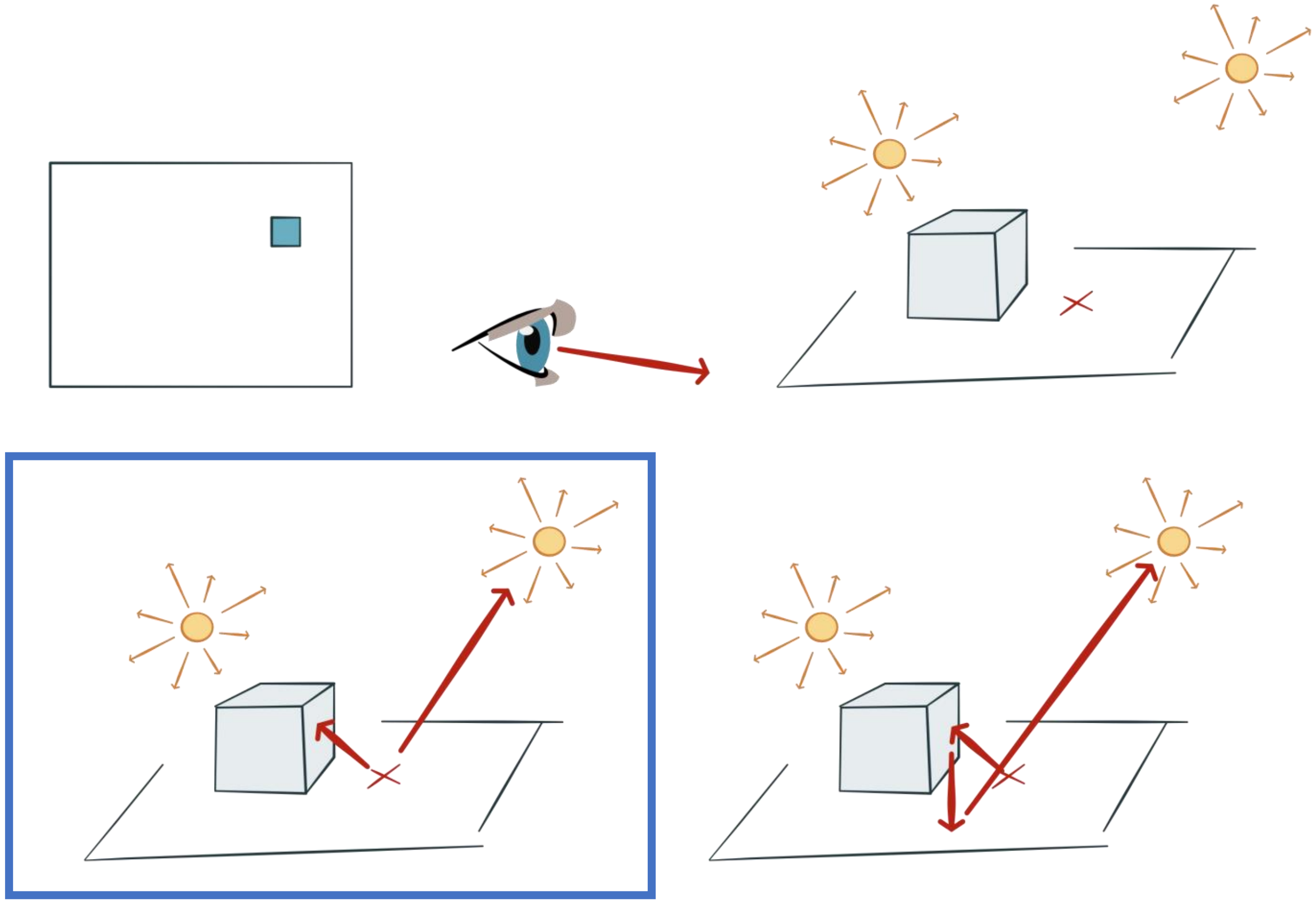


Light sources



Materials & Textures

Volumes



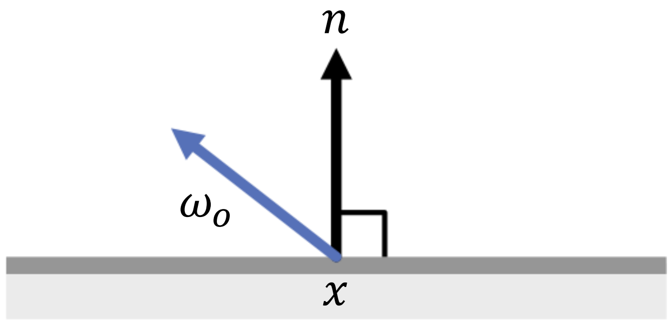
Rendering equation

The heart of rendering

... and a **guaranteed** exam question

$$\boxed{L_o(x,\omega_o)} = \boxed{L_e(x,\omega_o)} + \boxed{\int_\Omega L_i(x,\omega_i) f_r(x,\omega_i,\omega_o) |\cos\theta_i| d\omega_i}$$

Outgoing light — Emitted light — Reflected light

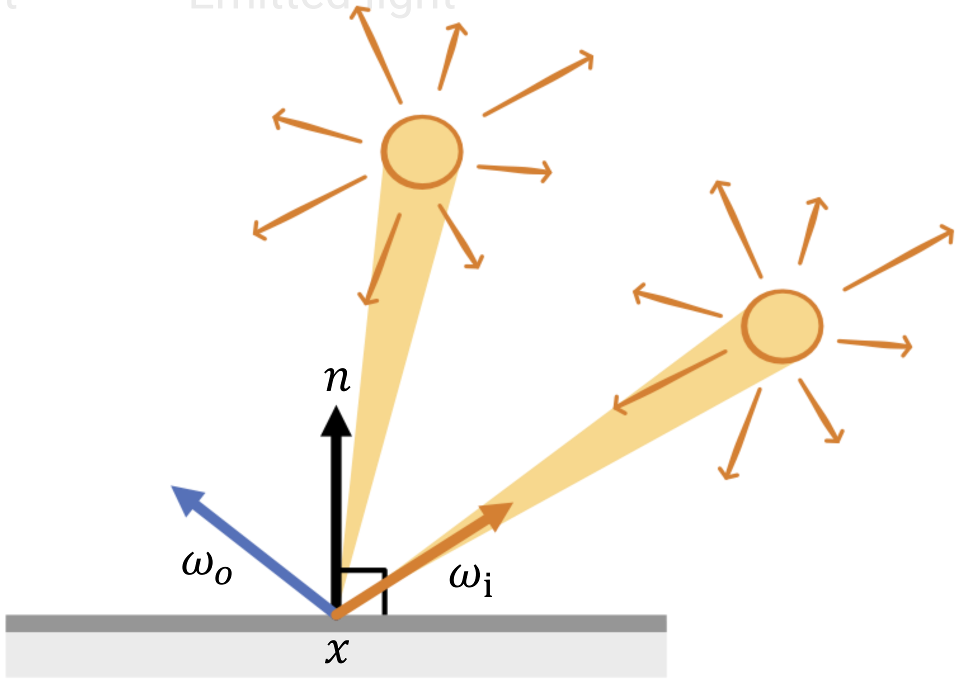


The heart of rendering

... and a **guaranteed** exam question

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \boxed{\int_\Omega L_i(x, \omega_i) f_r(x, \omega_i, \omega_o) |\cos\theta_i| d\omega_i}$$

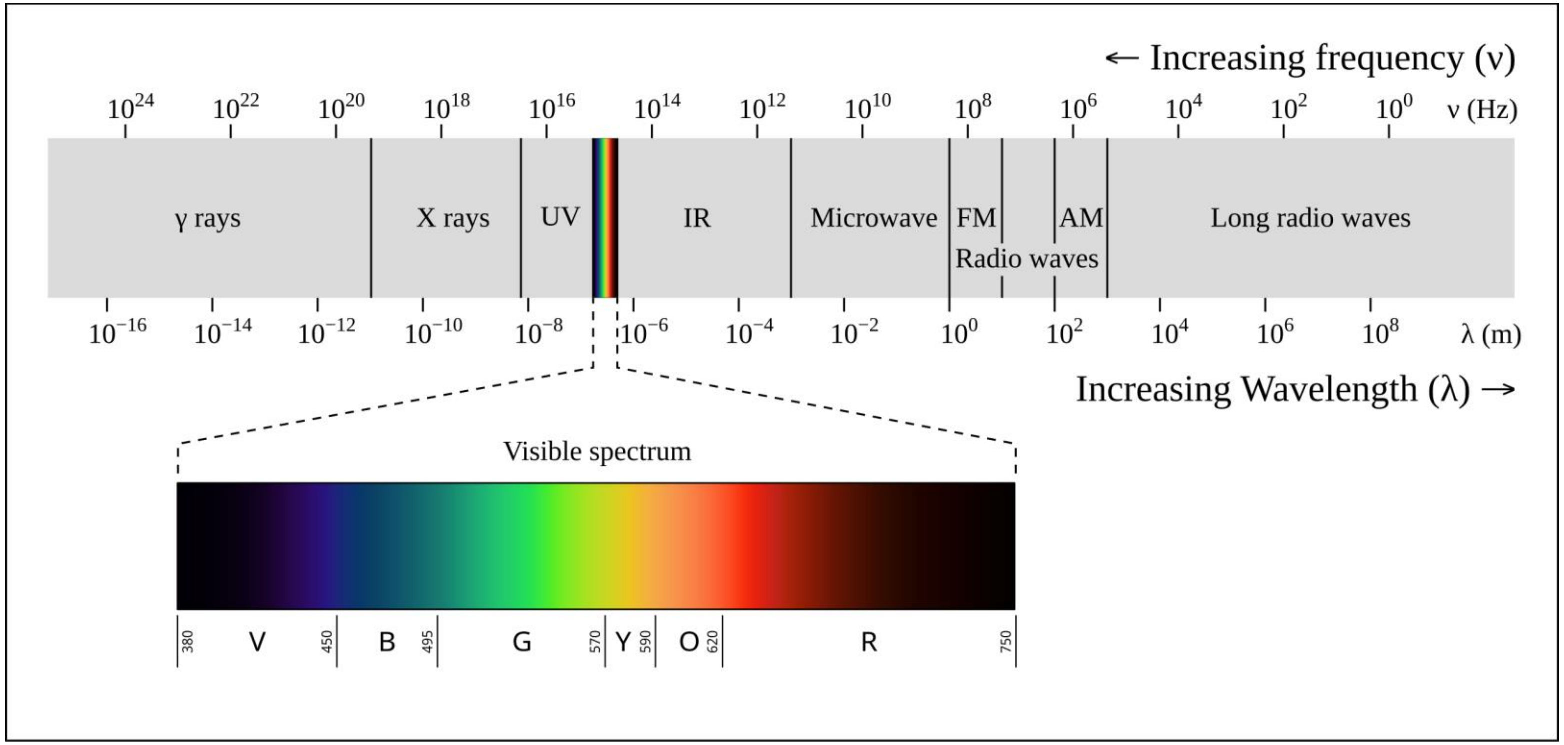
Outgoing light Emitted light Reflected light



"sum" over light from all directions

To define this properly, let’s look at some physics

- Visible light is a type of electromagnetic radiation
- We use *radiometry* to measure it



Flux Φ (radiometric power)

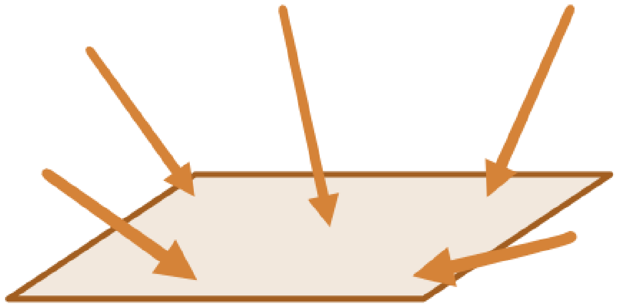
- Energy per unit time
  - Definition: Time derivative of the energy $\Phi = \frac{dQ}{dt}$
  - Unit: Watt (W)
- We measure / compute it at an instant in time

Not this, though

- Electric power of an equivalent incandescent light bulb
- Flux is this power minus heat loss
  - (Though, technically, "heat loss" is mostly flux in IR spectrum)

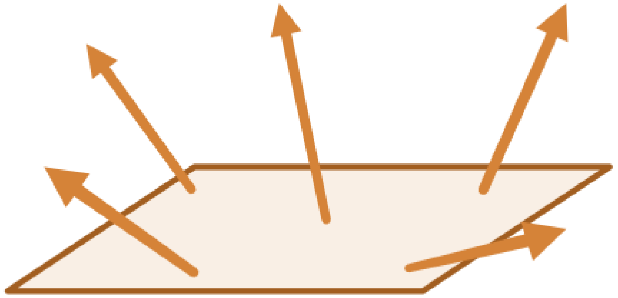
Quantifying the amount of light that reaches / leaves a surface

- Irradiance is the incoming power per unit area
  - $E = \frac{d\Phi}{dA}$
  - unit: $\frac{W}{m^2}$



Irradiance

- Radiosity is the outgoing (emitted or reflected) power per unit area
  - $R = \frac{d\Phi}{dA}$
  - unit: $\frac{W}{m^2}$



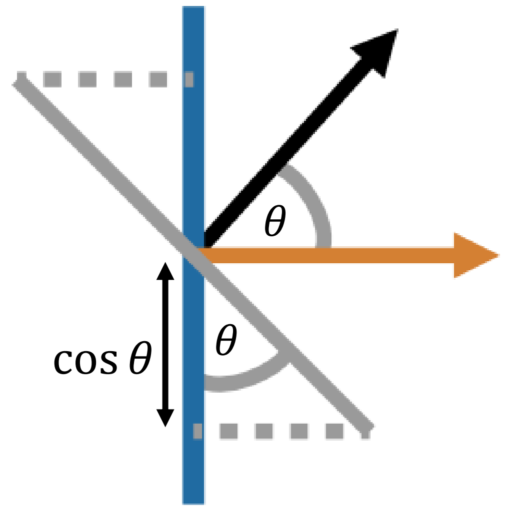
Radiosity

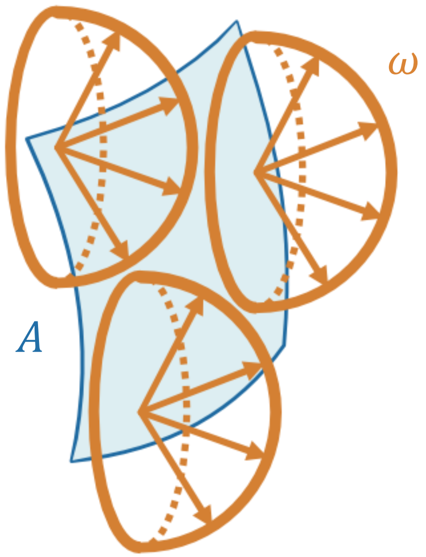
Radiance is our main quantity of interest

- A *directional* quantity: Power per "direction" and projected area

$$L = \frac{d\Phi}{d\omega\, dA \cos\theta}$$

To define this properly, we need a *measure*





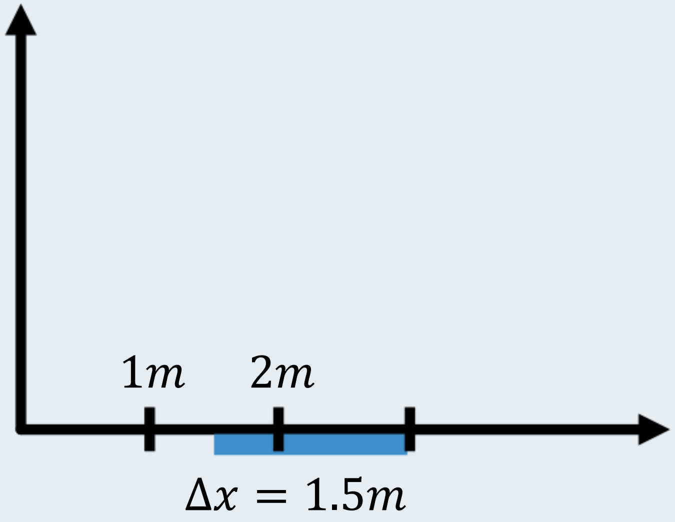
💡 That's why we have a cosine in the rendering equation

Integrals and derivatives are defined via a *measure*

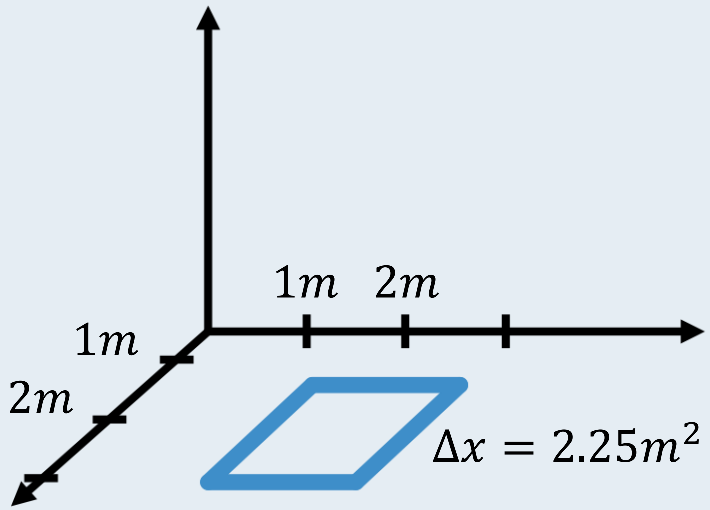
$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

This needs a notion of "how big is Δx?"

Examples



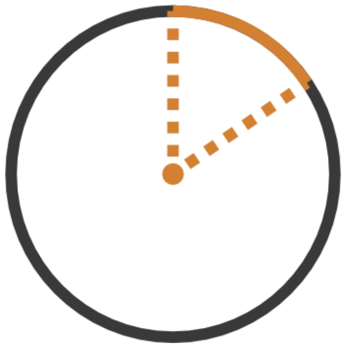
1D length measured in meters

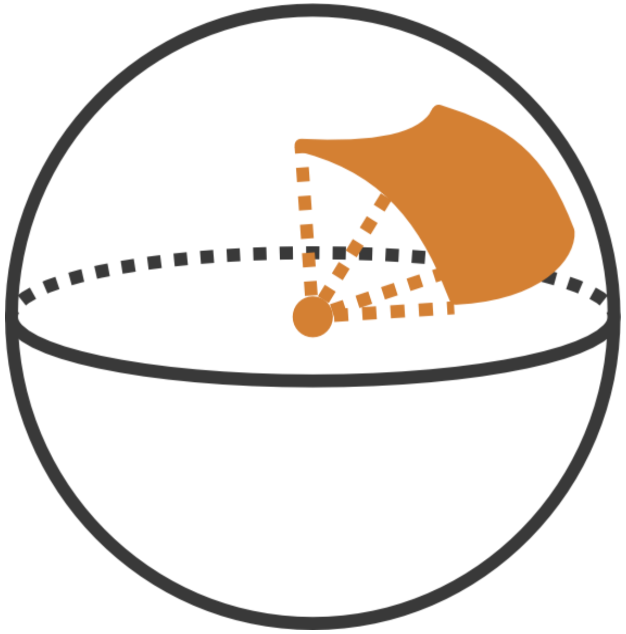


2D area measured in square meters

The solid angle: A measure for directions

- Remember radian ("rad")?
  - Measures 2D angle as the corresponding length on the unit circle
- A *solid angle* is the 3D analog of a 2D angle
  - Its unit is the steradian ("sr")
  - Measures the area on the unit sphere
- And a set of directions corresponds to a patch on the unit sphere





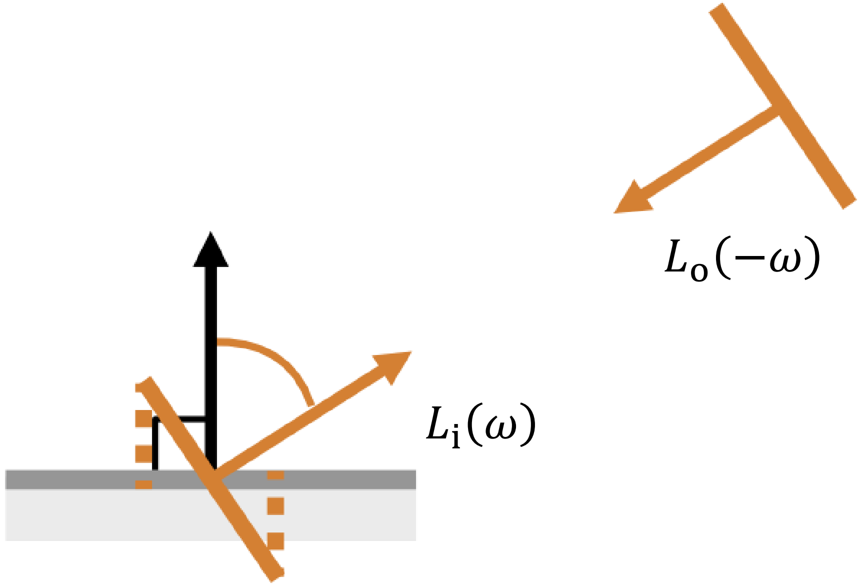
Radiance is power per solid angle per projected area

$$L = \frac{d\Phi}{d\omega\; dA \cos\theta}$$

Unit: $\frac{W}{\text{sr}\; m^2}$

💡 That's why we have a cosine in the rendering equation

Radiance remains constant (in vacuum)



$$L_\mathrm{i}(\omega) = L_\mathrm{o}(-\omega)$$

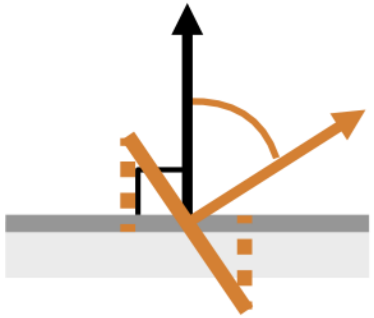
Independent of geometry and distance!

Getting back to the rendering equation

$$L_{\mathrm{o}}(x,\omega_{\mathrm{o}}) = L_{\mathrm{e}}(x,\omega_{\mathrm{o}}) + \int_{\Omega} L_{\mathrm{i}}(x,\omega_{\mathrm{i}}) f_{\mathrm{r}}(x,\omega_{\mathrm{i}},\omega_{\mathrm{o}}) |\cos\theta_{\mathrm{i}}| d\omega_{\mathrm{i}}$$

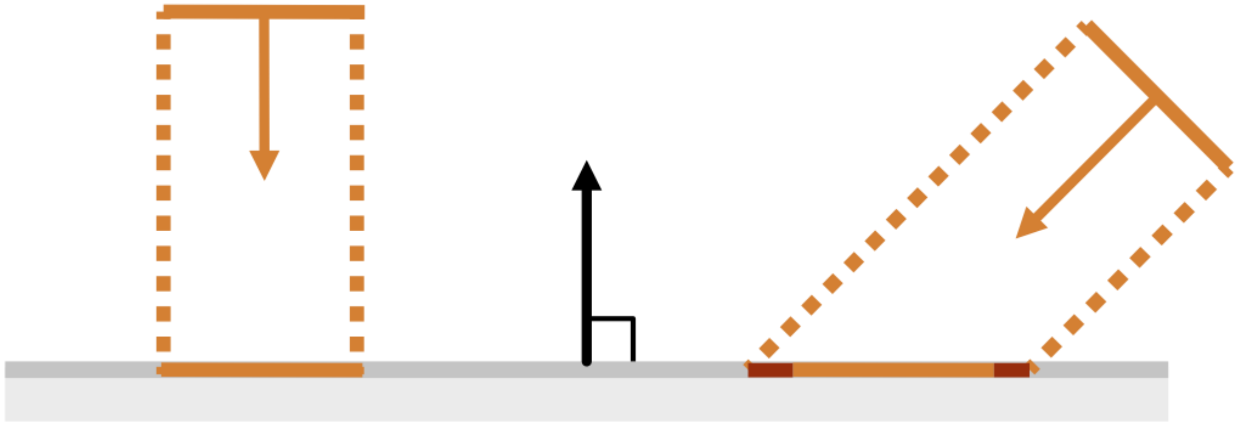
Outgoing radiance is emitted radiance plus integral over the unit sphere, measured by solid angle, of the incoming radiance, modified by the BRDF and cosine

- Why the cosine?
  - Cancels out the same $\cos\theta$ in the definition of radiance
  - Because we want the power per area at x
  - And L_i is power per area *perpendicular* to ω_i



An intuitive take on the cosine

- If the same “amount of” light arrives from different angles...
- The one from the grazing angle is “spread over a larger area”



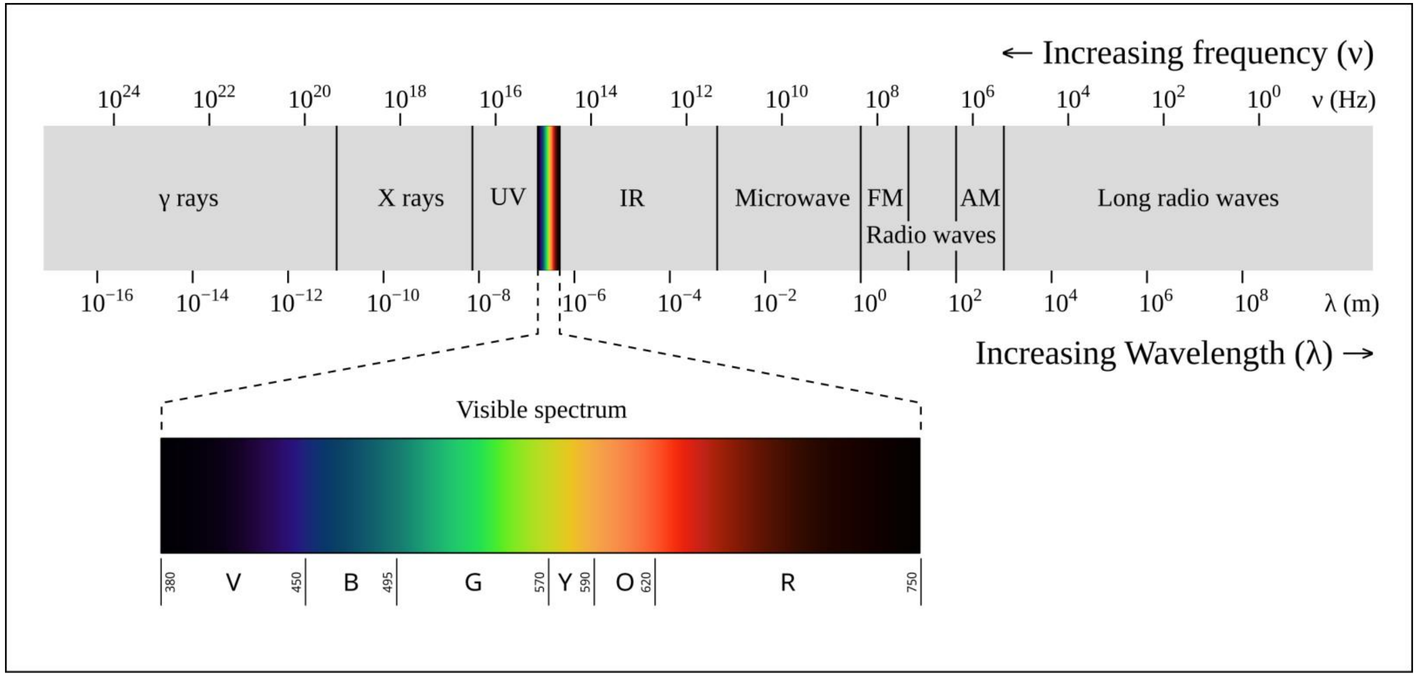
Don’t want to take my word for it? Try shining light with your phone onto your table!

What about color?

- Ideally: consider the spectral radiance, i.e., radiance per wavelength λ

$$L_\lambda = \frac{L}{d\lambda}$$

- Common simplification: RGB
  - Red, Green, Blue color channels
  - More on that in lecture 4



Reading materials

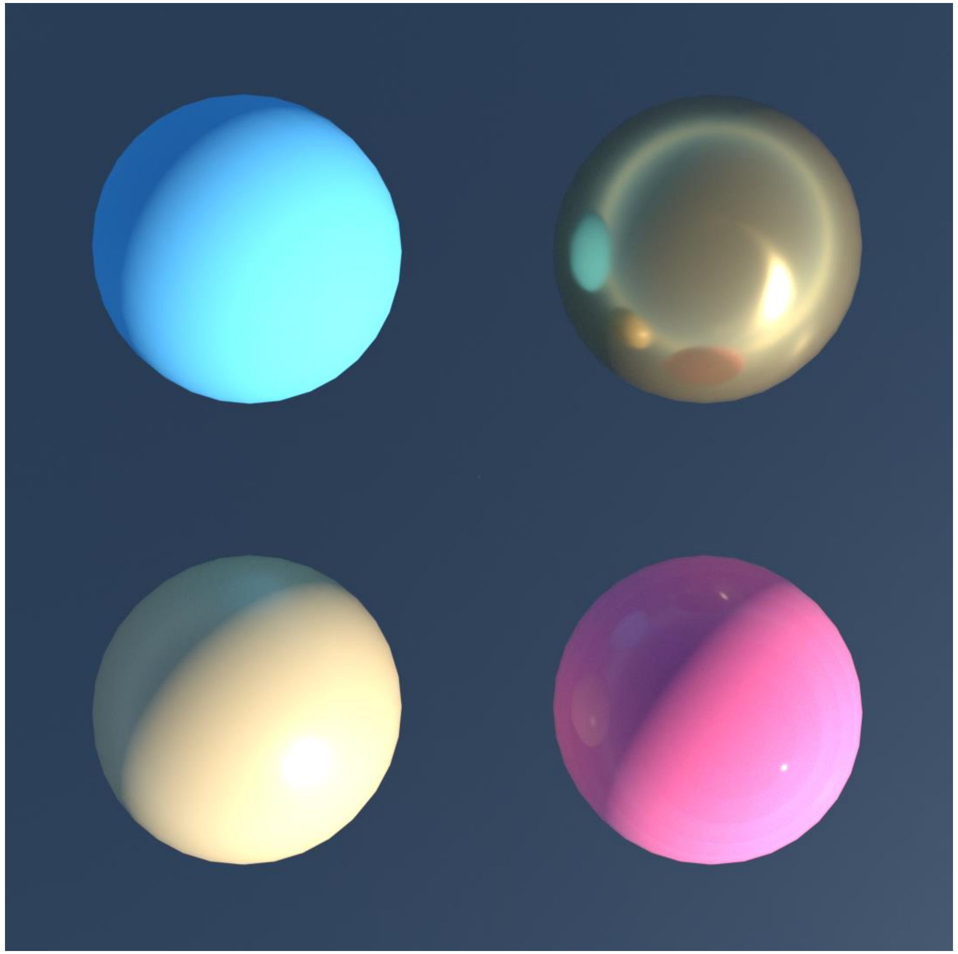
- https://pbr-book.org/4ed/Radiometry,_Spectra,_and_Color/Radiometry
- Kajiya, James T. 1986. "The rendering equation."
- *Chapter 3 of:* Veach, Eric. 1997. Robust Monte Carlo Methods for Light Transport Simulation. PhD thesis. https://graphics.stanford.edu/papers/veach_thesis/thesis-bw.pdf

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) \boxed{f_r(x, \omega_i, \omega_o)} |\cos\theta_i| d\omega_i$$

The BRDF

Bidirectional Reflectance Distribution Function

The BRDF models the surface appearance



The BRDF – $f_\mathrm{r}(x, \omega_\mathrm{i}, \omega_\mathrm{o})$

- Describes the fraction of light from ω_i that is reflected to ω_o at point x
- Must satisfy some properties:
    - Energy conservation (technically, conservation means $=$, but we allow absorption):

$$\int_\Omega f_\mathrm{r}(x, \omega_i, \omega_o) \cos\theta_i \, d\omega_i \leq 1$$

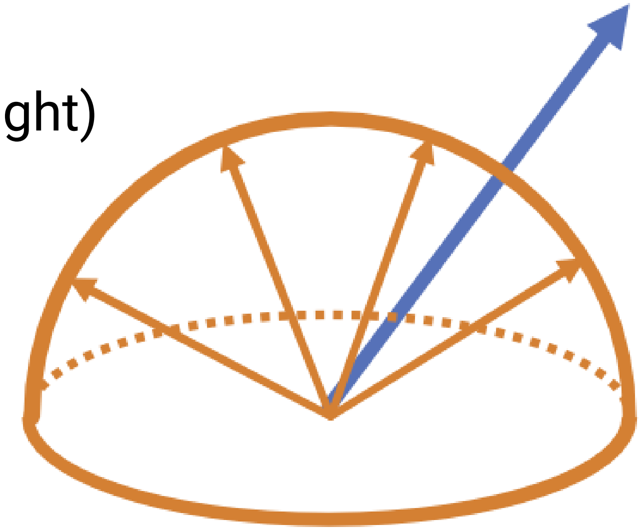
    - Reciprocity (typically but *not always* equivalent to symmetry):

$$f_\mathrm{r}(x, \omega_i, \omega_o) = f_\mathrm{r}(x, \omega_o, \omega_i)$$

A Lambertian diffuse BRDF

- Reflects light equally in all directions
- This is ***not*** $f_r = \rho$
  - ρ is the albedo – the color of the reflected light
- Because

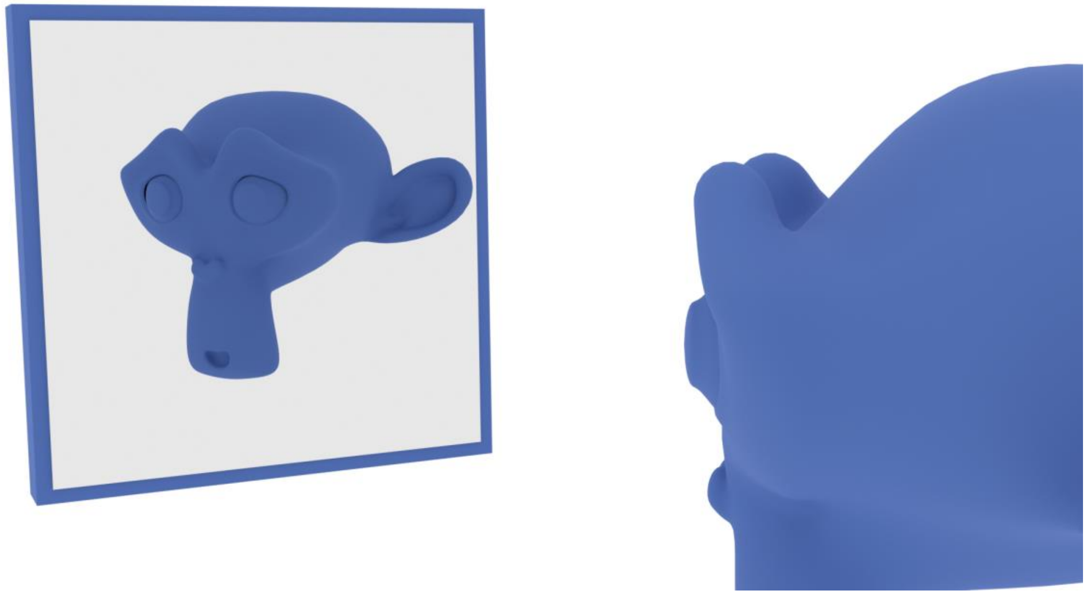
$$\int_{\Omega} 1 \cos\theta \, d\omega = \pi > 1$$



- Violates energy conservation (the surface would appear to generate light)
- The **correct** diffuse BRDF is

$$f_r = \rho \frac{1}{\pi}$$

Perfect mirror reflection



- $L_r(\omega) = L_i(\omega_r)$

- How to encode that as a BRDF?
  - Dirac delta distribution $\delta(x)$, defined as

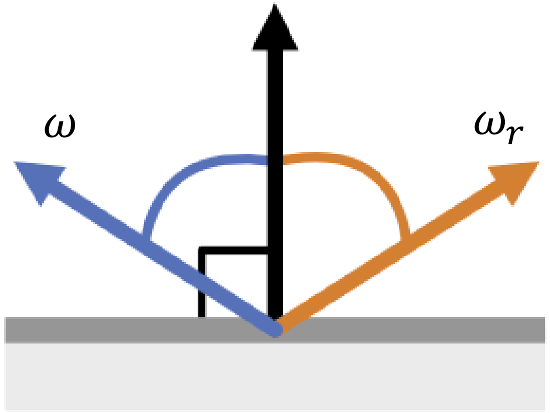
$$f(0) = \int_X f(x)\delta(x)dx$$

  - Intuition: $\delta(x)$ is zero everywhere, except *exactly* at 0, where it is infinitely large

- Perfect mirror BRDF:

$$= f_r(x, \omega_i, \omega)$$

$$L_r(\omega) = \int_\Omega \frac{\delta(\omega_i - \omega_r)}{\cos\theta_r} L_i \cos\theta_i \, d\omega_i = L_i(\omega_r)$$



To be continued...

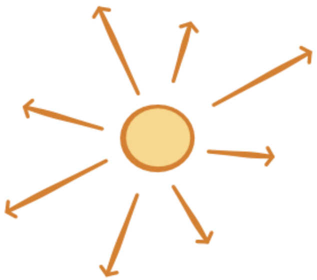
(in 2 weeks)

Reading materials

- *Nicodemus, Fred (1965). "Directional reflectance and emissivity of an opaque surface". Applied Optics.* ***4*** *(7): 767–775*

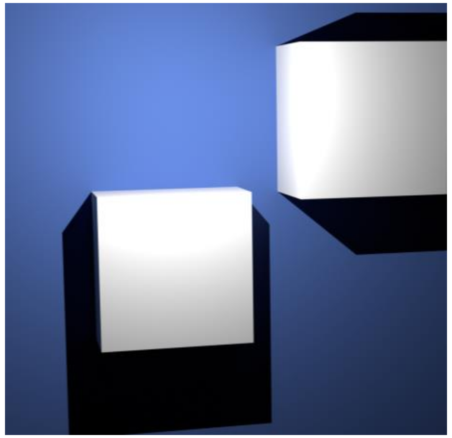
Computing simple illumination

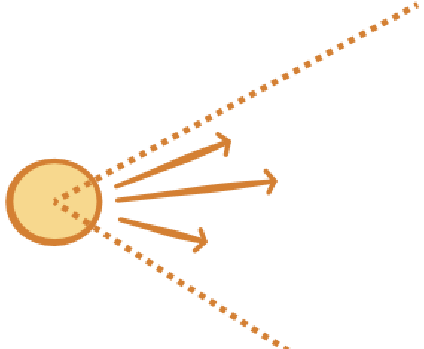
Point, spot, and directional lights



Point light

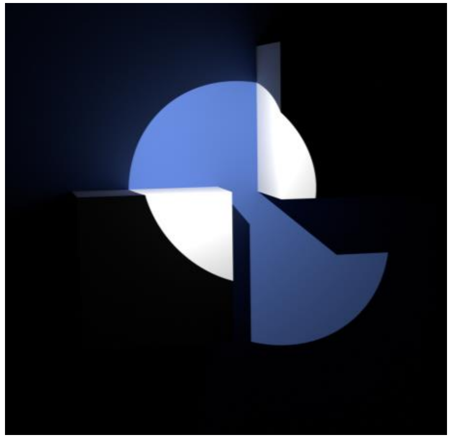
- Emits total power ϕ in all directions

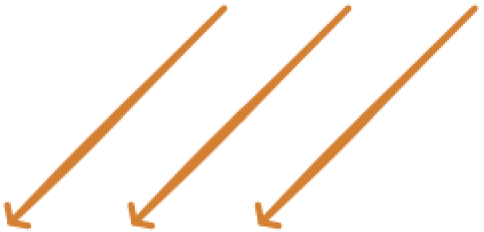




Spot light

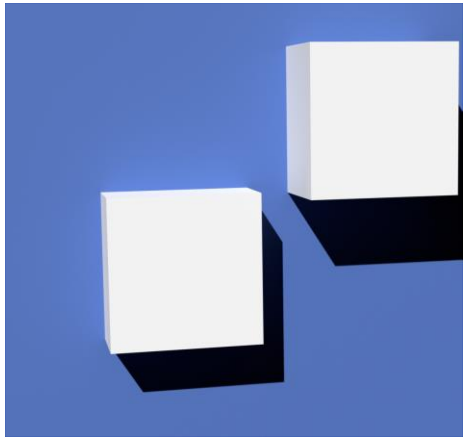
- A point light restricted to a cone



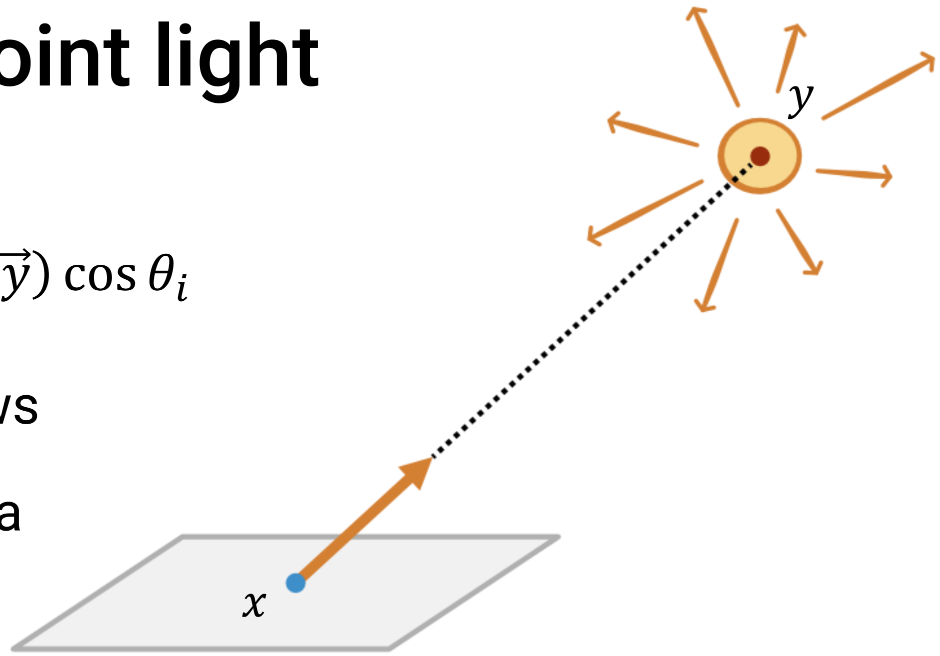


Directional light

- Infinitely far away light
- Exactly one direction

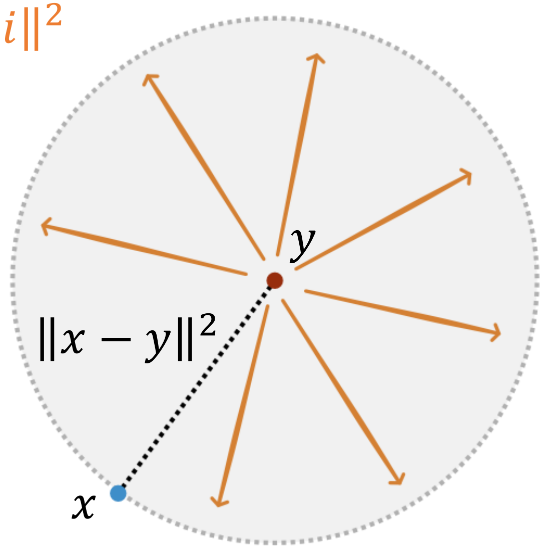


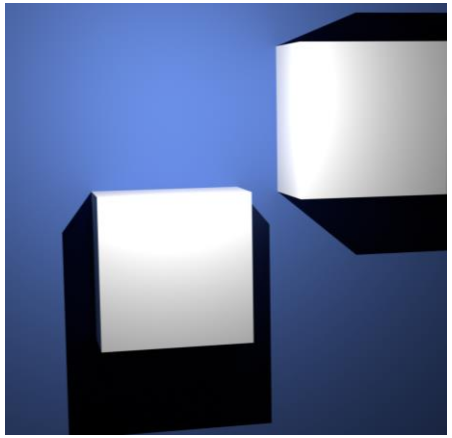
Computing direct illumination from a point light



$$L_o = \frac{\phi}{4\pi\|x-y\|^2} V(x,y) f_r(x, \omega_o, \overrightarrow{xy}) \cos\theta_i$$

- $V(x,y)$ is a binary visibility term, we ray-trace it to get shadows
- “Radiance” makes no sense for a single point: there is no area
  - The total power ϕ spreads spherically
  - Surface area of the sphere at x is $4\pi\|x-i\|^2$



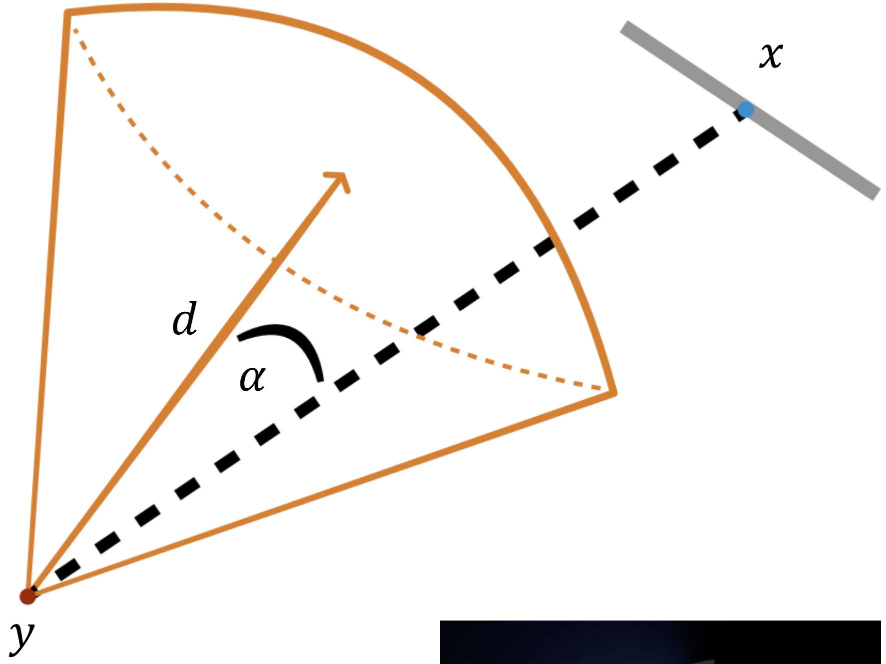


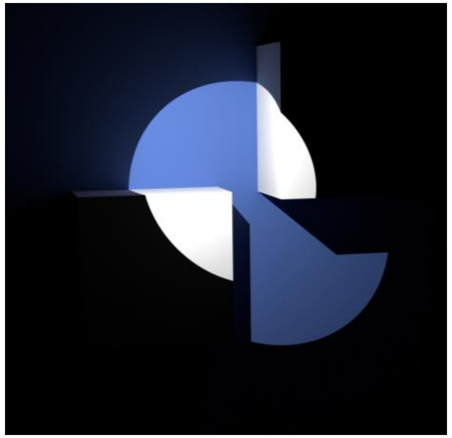
Computing direct illumination from a spot light

- Almost the same as for the point light
- Need to check if we are within the cone
  - Compare α to opening angle

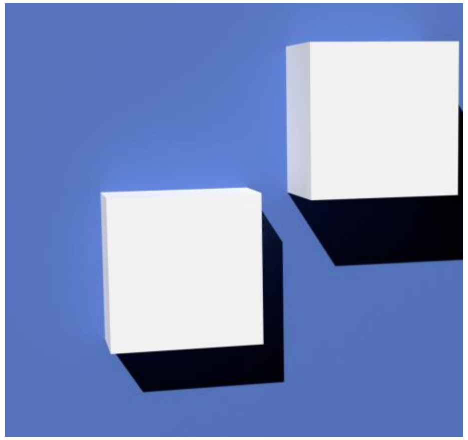
$$\cos\alpha = \frac{\langle d, \overrightarrow{yx} \rangle}{\|x - y\|}$$

- Bonus: intensity falloff as we move away from the center
  - Multiply ϕ by some factor computed from $\cos\alpha$





Computing direct illumination from a directional light



- Assumption: infinitely far away light source with emitted radiance L_e
- Light arrives from a single direction ω
- Rendering equation simplifies to

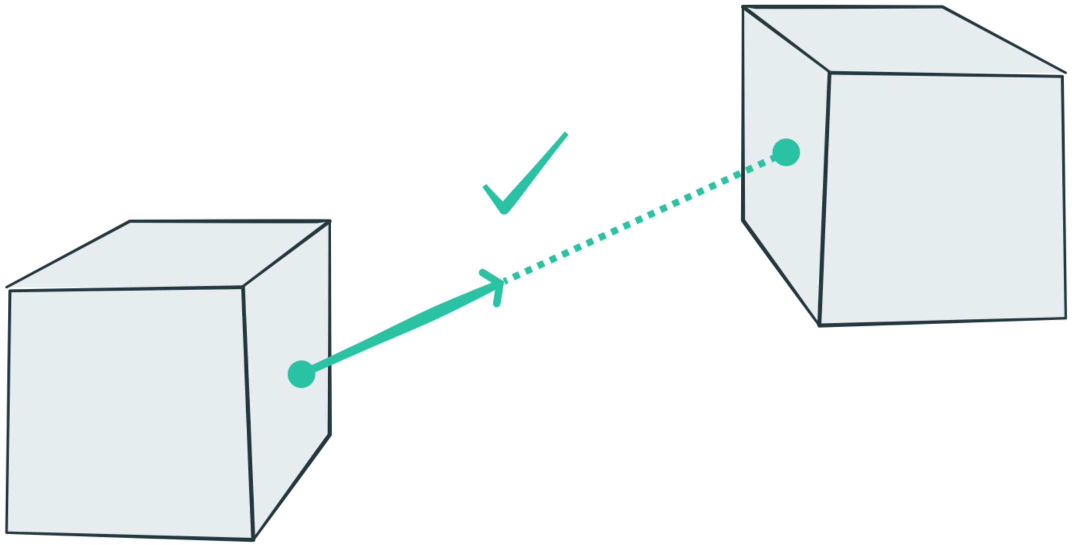
$$L_o(x, \omega_o) = V(\omega) L_e f_r(x, \omega_o, \omega) \cos\theta$$

  - $V(\omega)$ checks if there is any geometry in this direction, at any distance (ray traced)

Shadow rays, aka “any-hit” ray queries

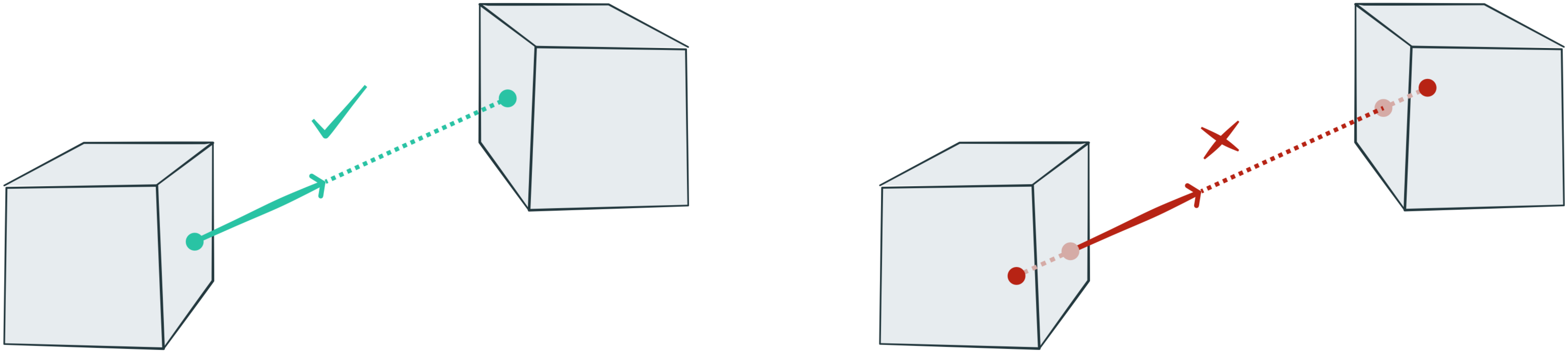
- Trace ray between two points
- Can terminate *on first hit* – distance is irrelevant

  ➔ Faster than “normal” closest-hit rays

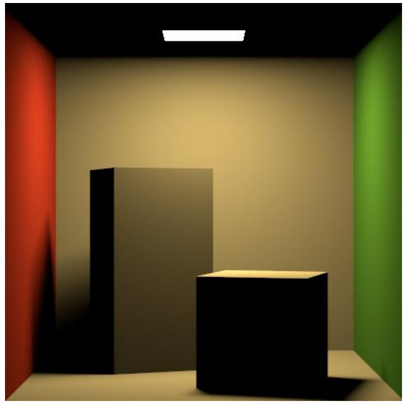


Self intersections are a problem

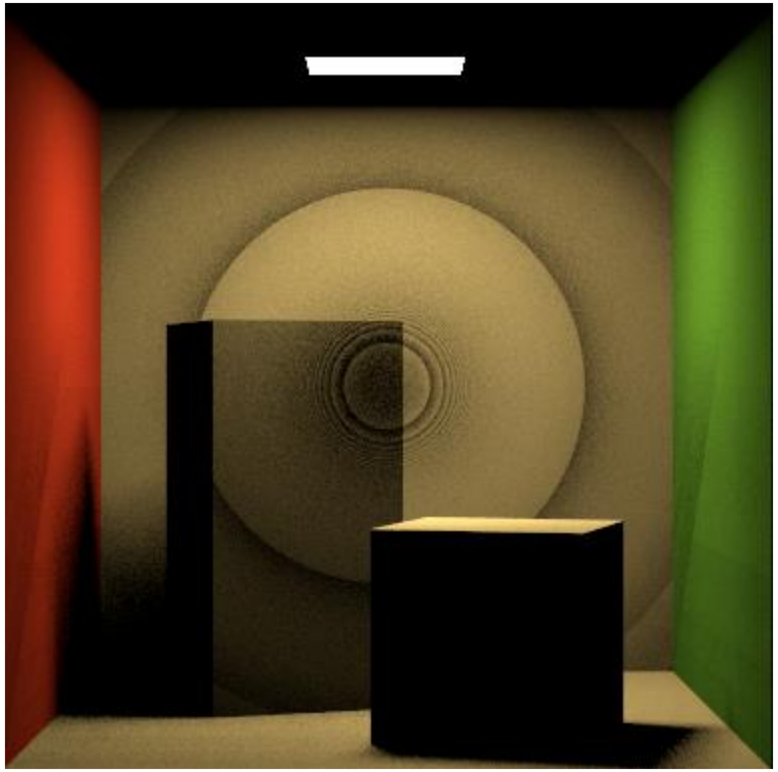
- Computers have limited accuracy
- Floating point errors can cause shadow tests to fail

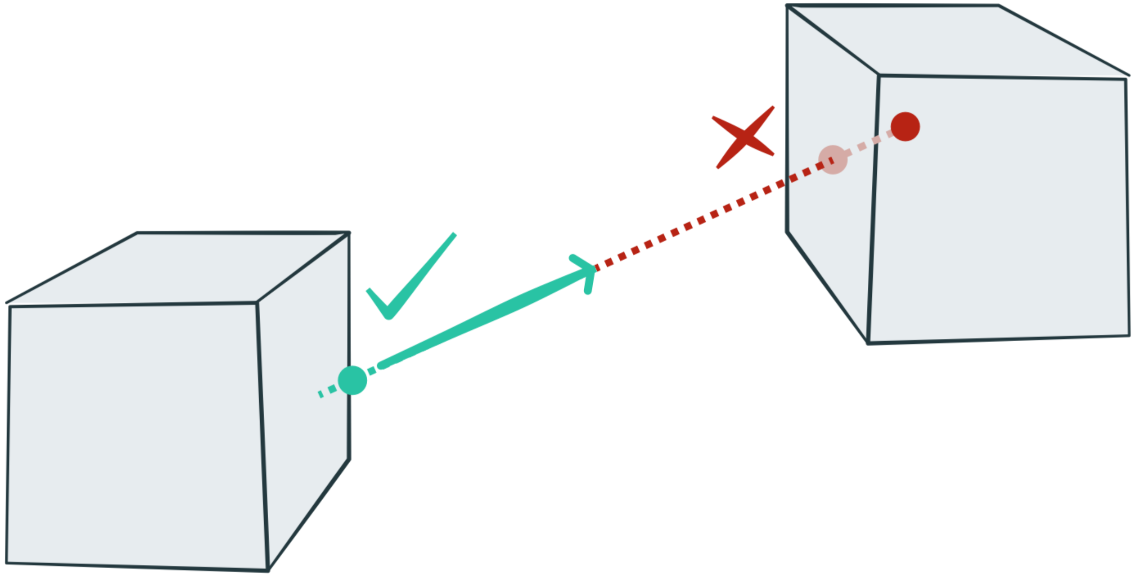


Self-intersection on the illuminated surface

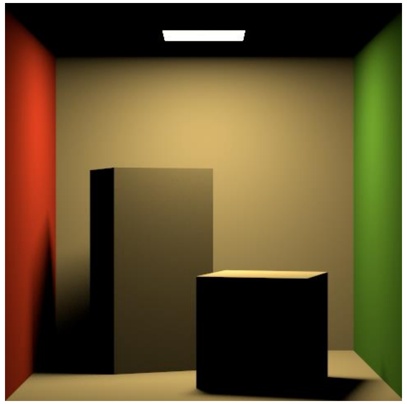


- Move origin away from the surface
- And/Or: ignore hits closer than a small minimum distance

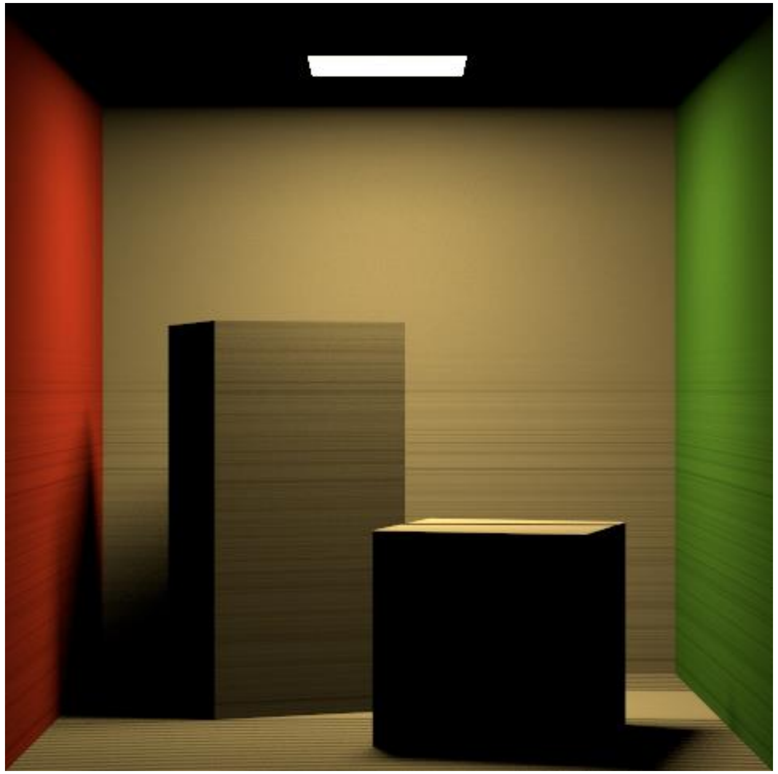


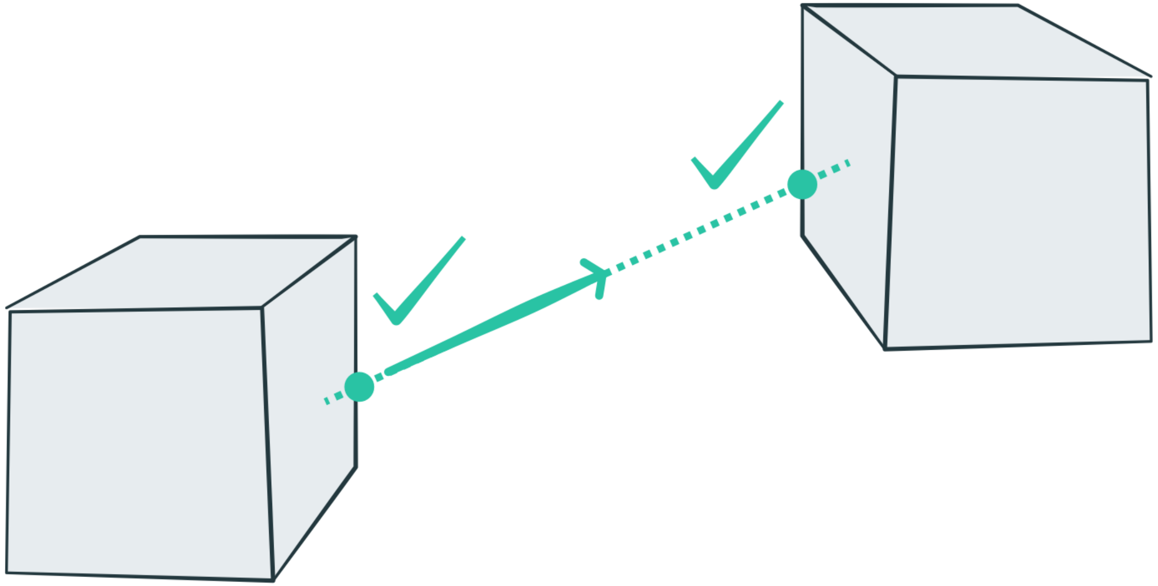


Self-intersection on the light source



- Set maximum distance shorter than actual distance





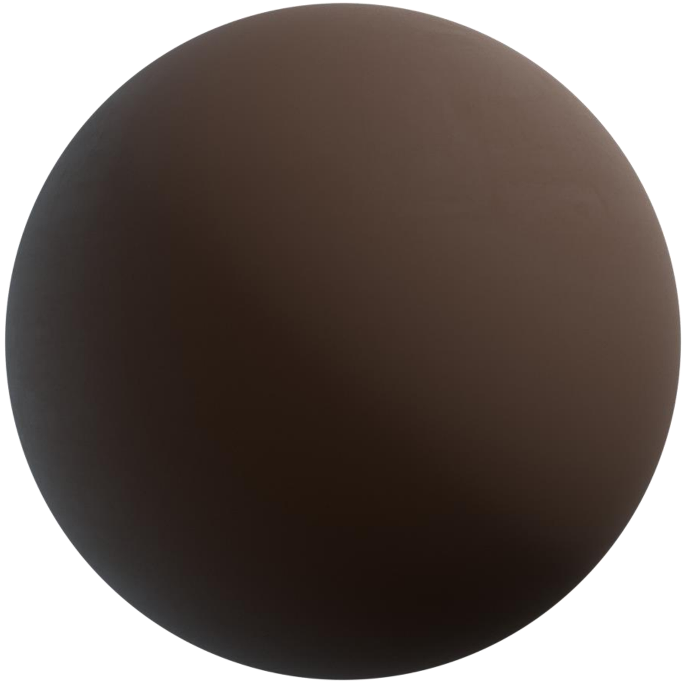
Reading materials

- https://pbr-book.org/4ed/Light_Sources
- *https://www.pbr-book.org/4ed/Shapes/Managing_Rounding_Error*

Textures

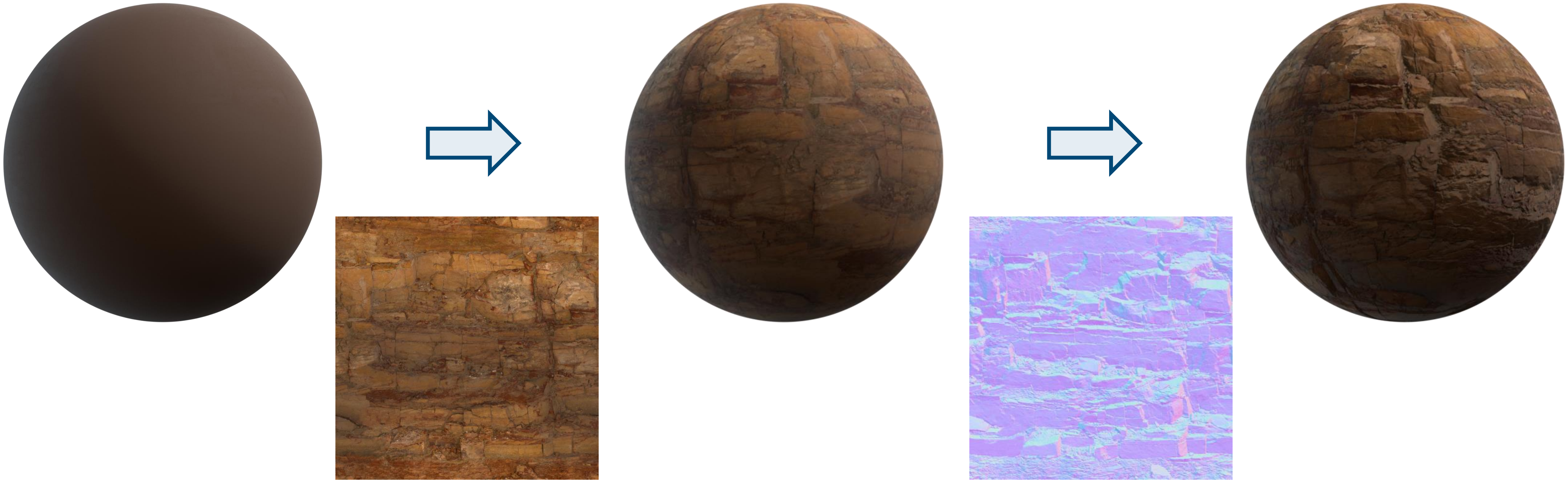
Textures are an inexpensive way to add detail

… well, compared to excessive geometry detail



https://polyhaven.com/a/cliff_side

Textures can control arbitrary parameters

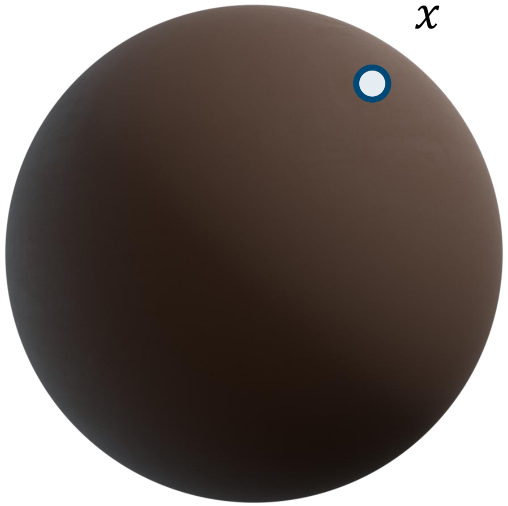


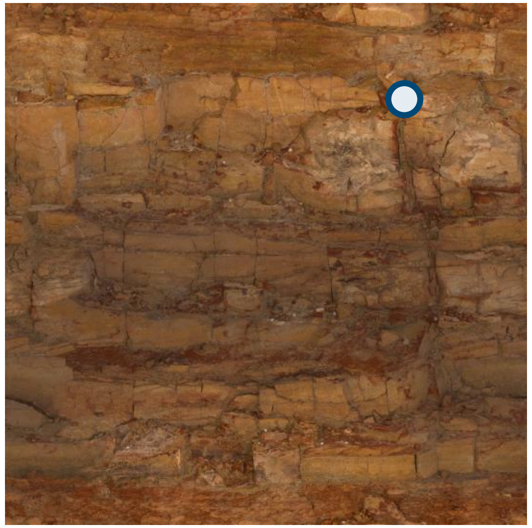
Base color

Normal

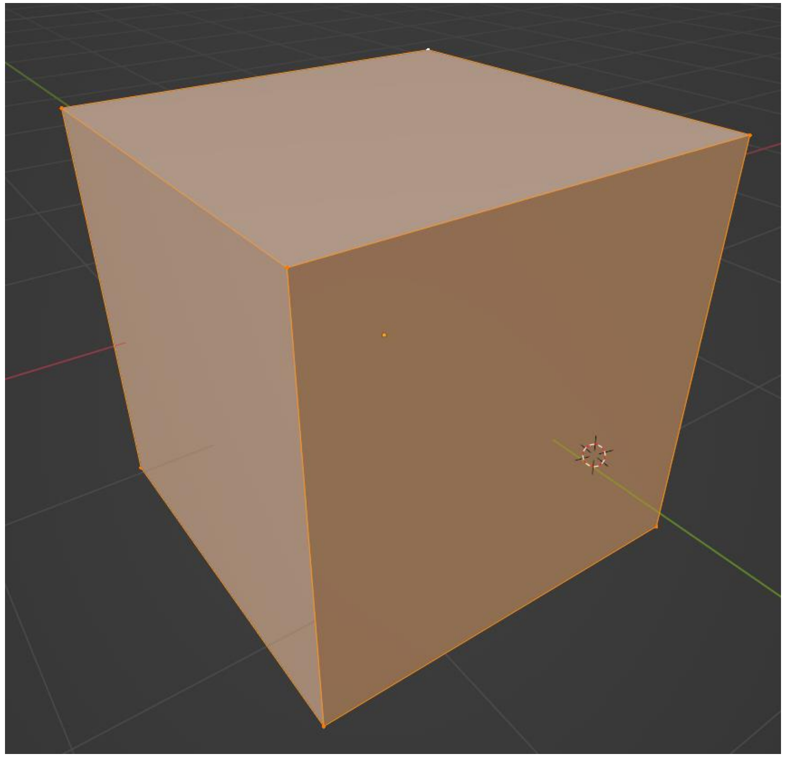
A texture is a function $t(x)$

- Maps surface point x to a parameter value
  - Here: $t(x)$ assigns a material color based on an image
- Two main types:
  - Image textures
  - Procedural textures

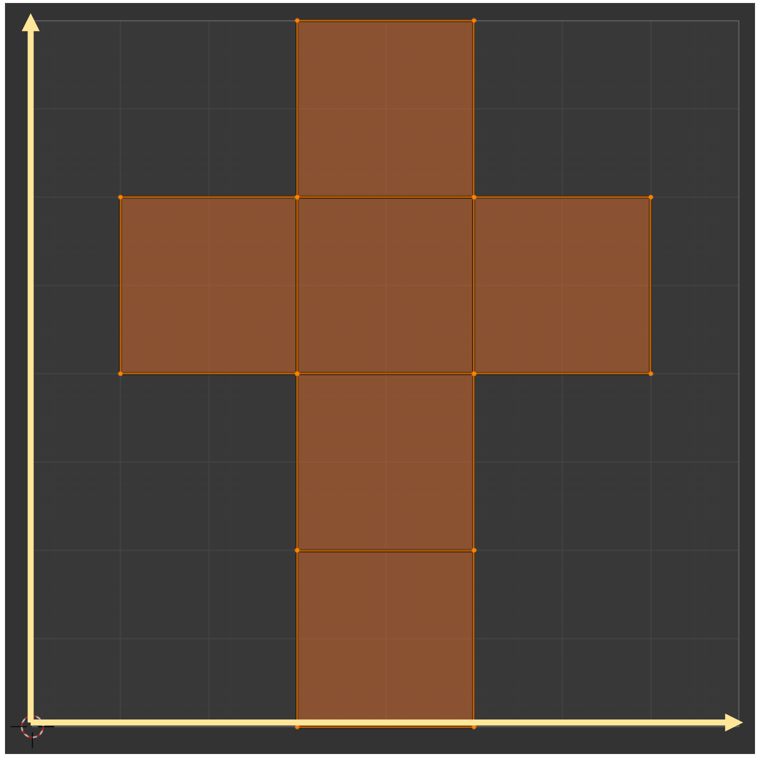




Surface parametrization of a cube

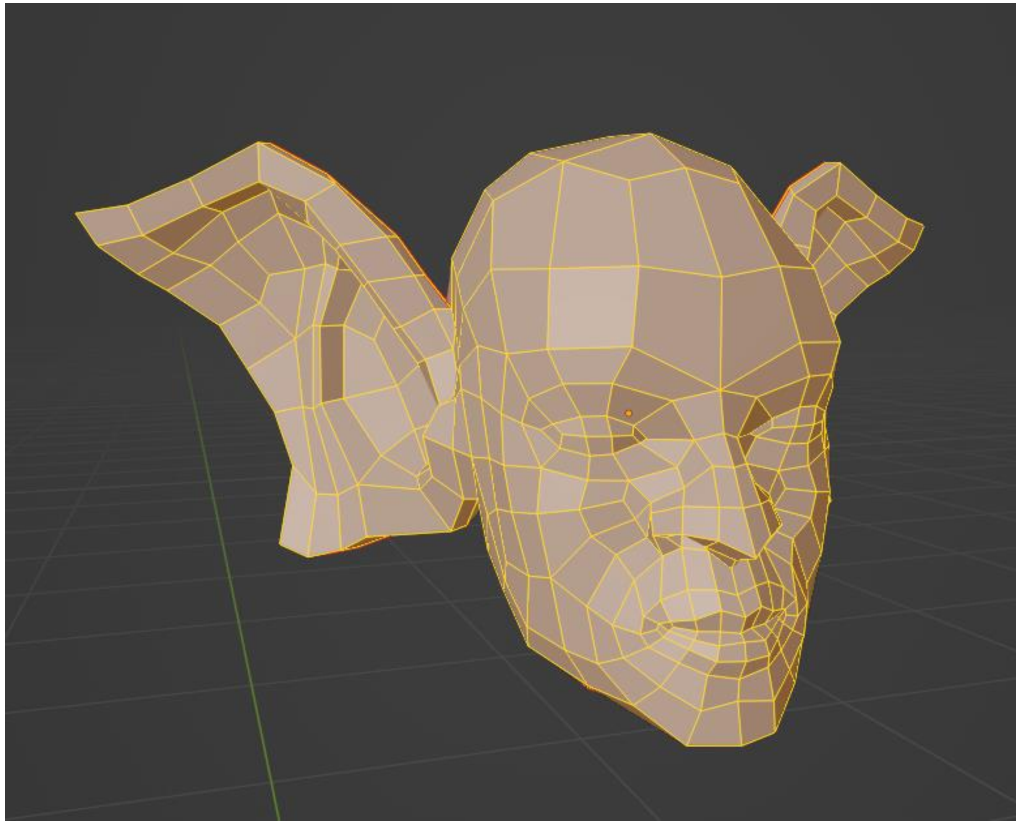


Geometry

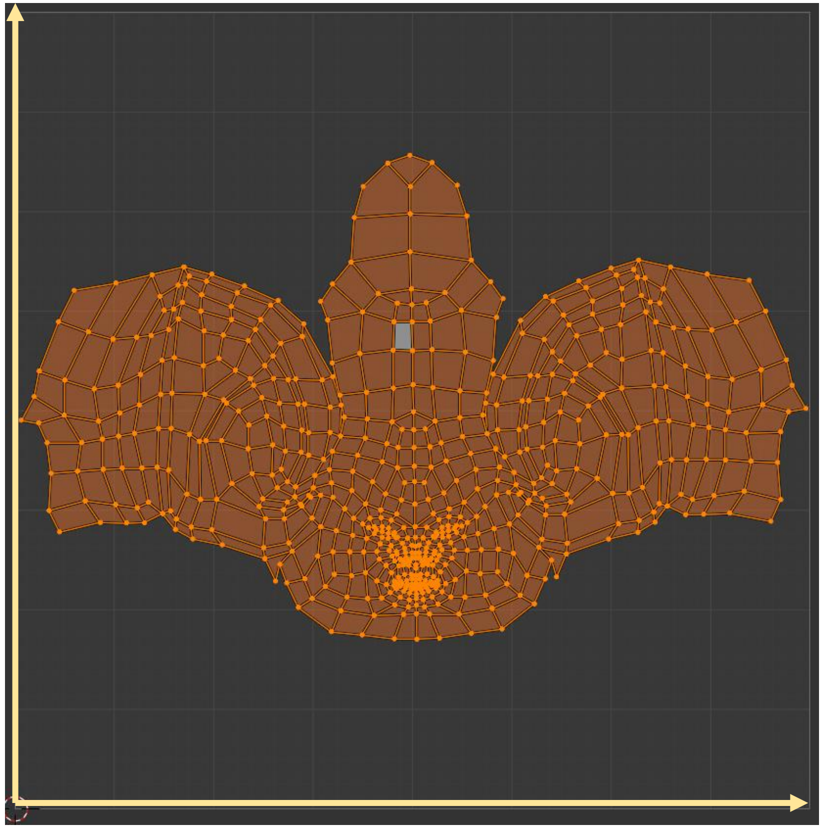


UV Map

Surface parametrization of a humanoid head



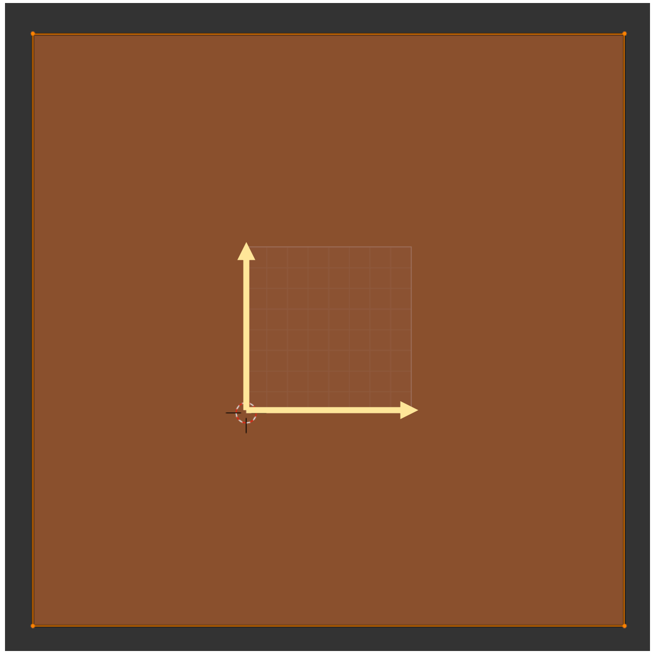
Geometry



UV Map

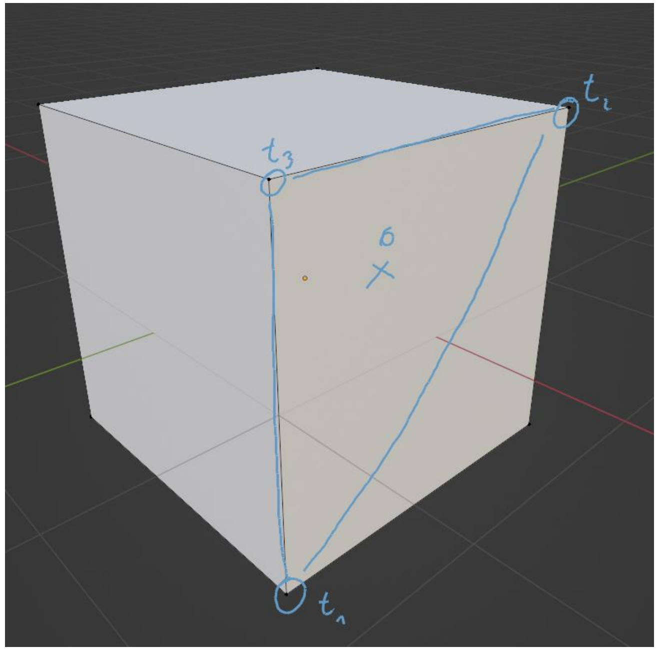
Tiling: repeating a texture over a larger surface

- Texture coordinates outside the $[0,1]$ range
- *Border handling* dictates how these are mapped back onto the image
  - Here: “repeat”: $u' = u \bmod 1$
  - Alternatives: mirror, clamp



Barycentric coordinates and vertex attributes

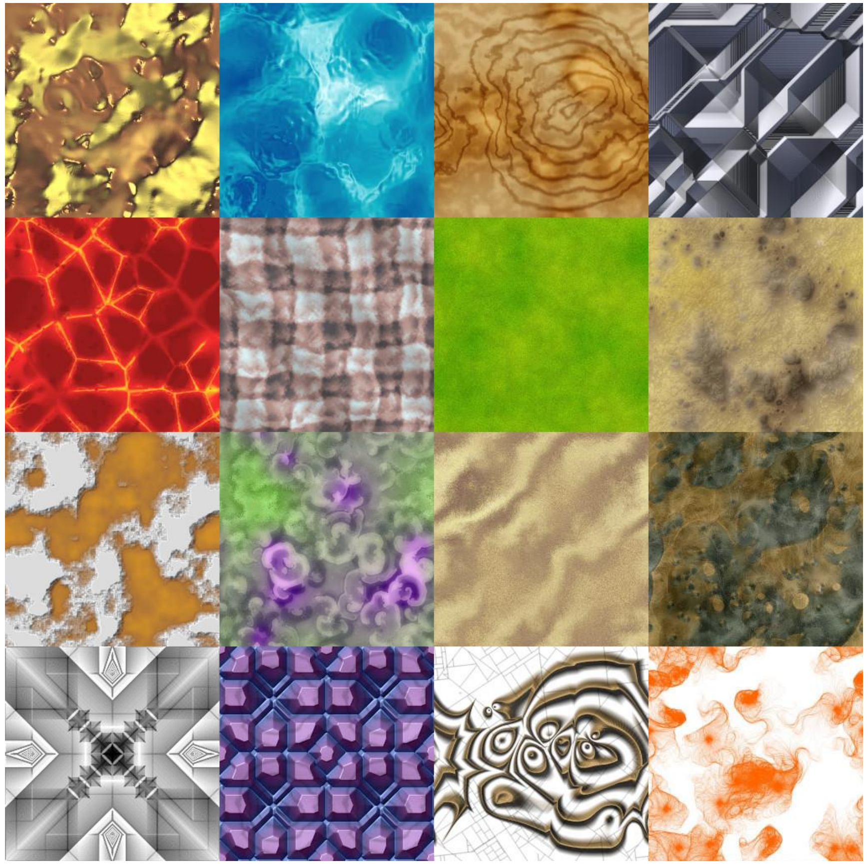
- How to get the texture coordinate at a hit point x?
- Vertices p_1, p_2, p_3 store their texture coordinates t_1, t_2, t_3
- Barycentric coordinates (u, v)
  - $x = up_1 + vp_2 + (1 - u - v)p_3$
  - Interpolate the triangle corners to get x
- We can interpolate any *vertex attribute* the same way
  - $t_x = ut_1 + vt_2 + (1 - u - v)t_3$



Procedural textures: Describing patterns via math

- Many possibilities:
  - (quasi) random noise
  - Voronoi patterns
  - Tiles and chessboards
  - ...
  - And all combinations of those!

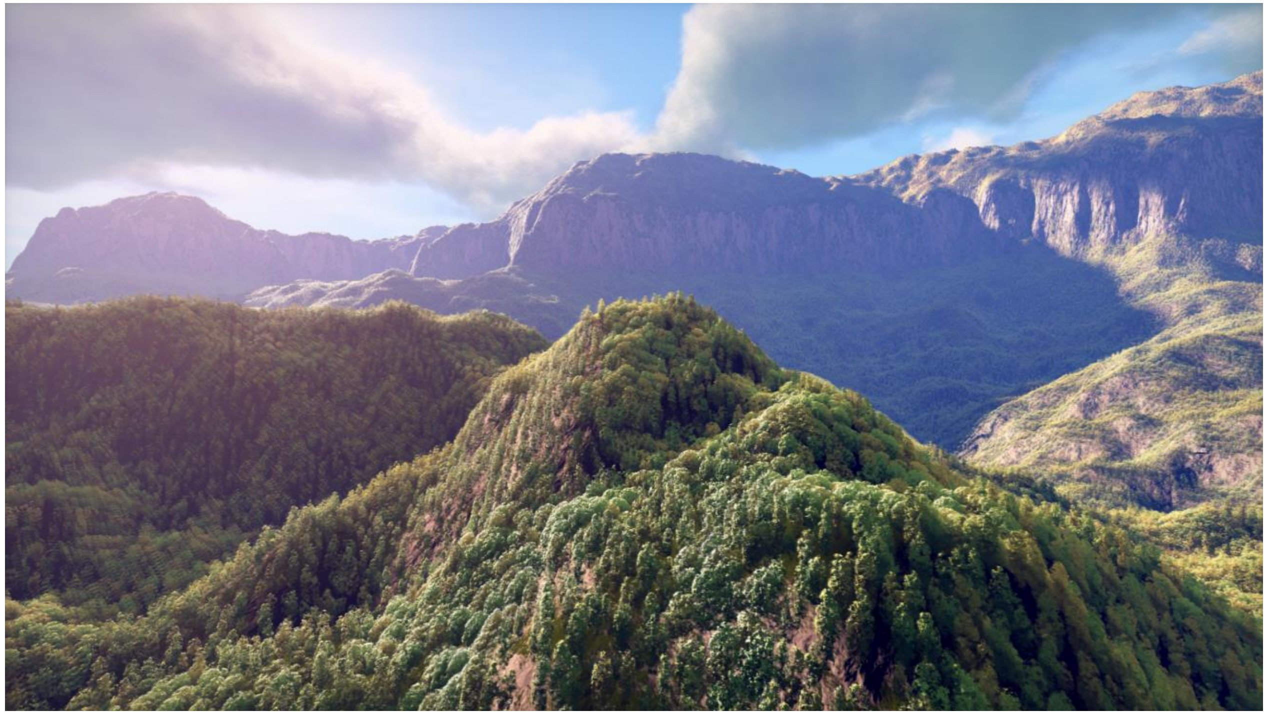
- Benefits: low memory cost, infinite resolution
- Drawbacks:
  - Tricky to find functions and parameters to get a desired look
  - (sometimes) high evaluation cost



https://opengameart.org/content/40-procedural-textures

Example: Procedural Noise

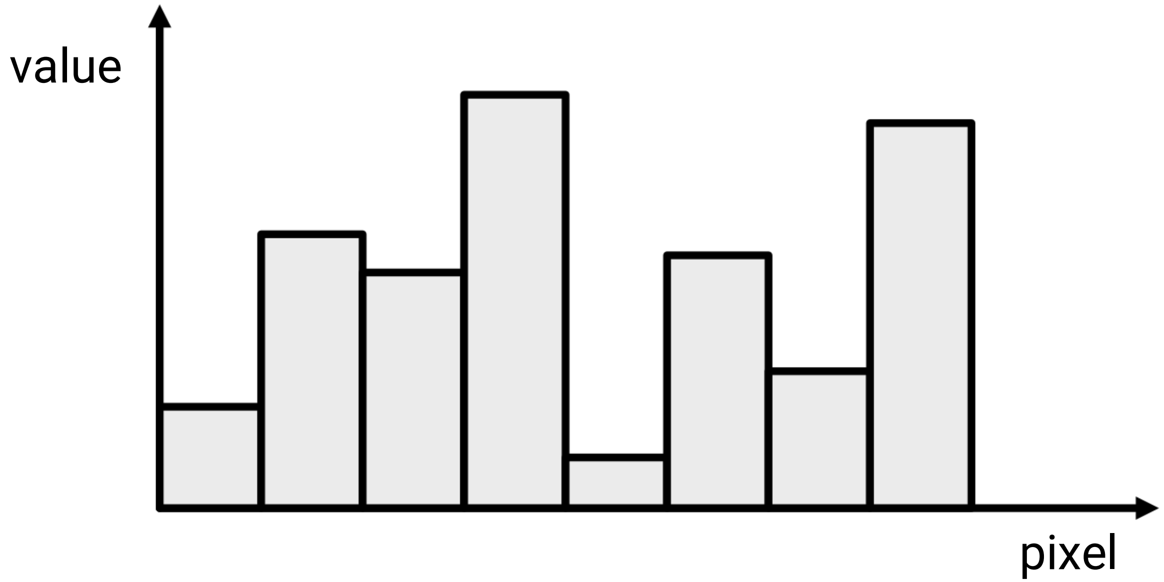
- A great way to add detail to huge scenes
- Or to make your renderings look less “synthetic”

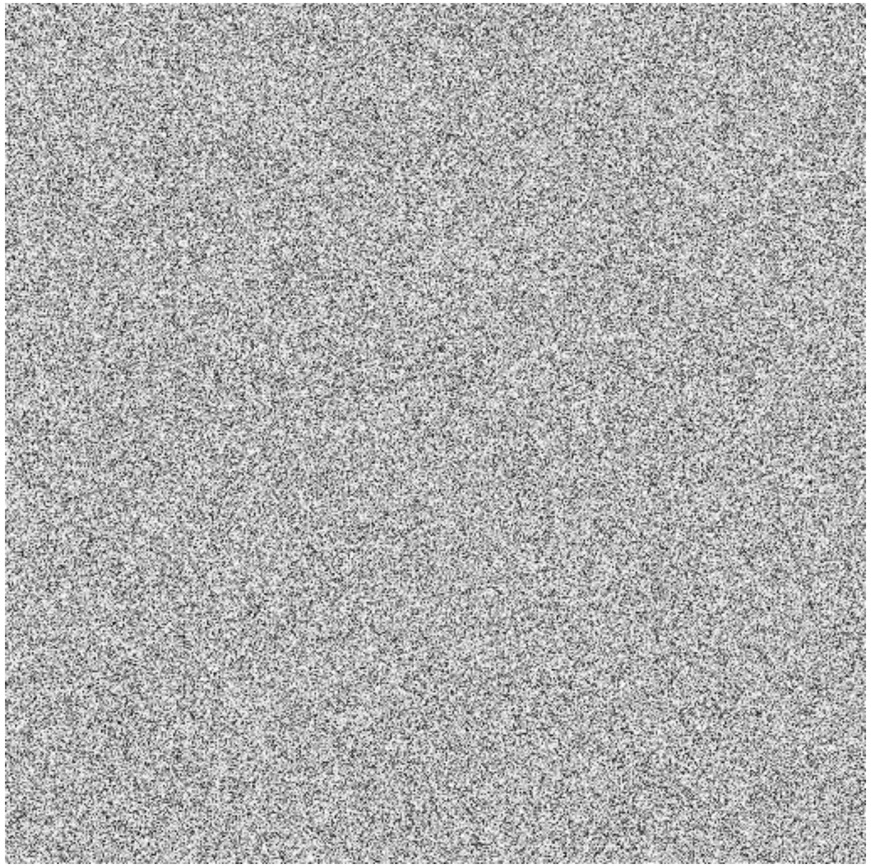


https://www.shadertoy.com/view/4ttSWf

Value noise: Every pixel (or tile) gets a random value

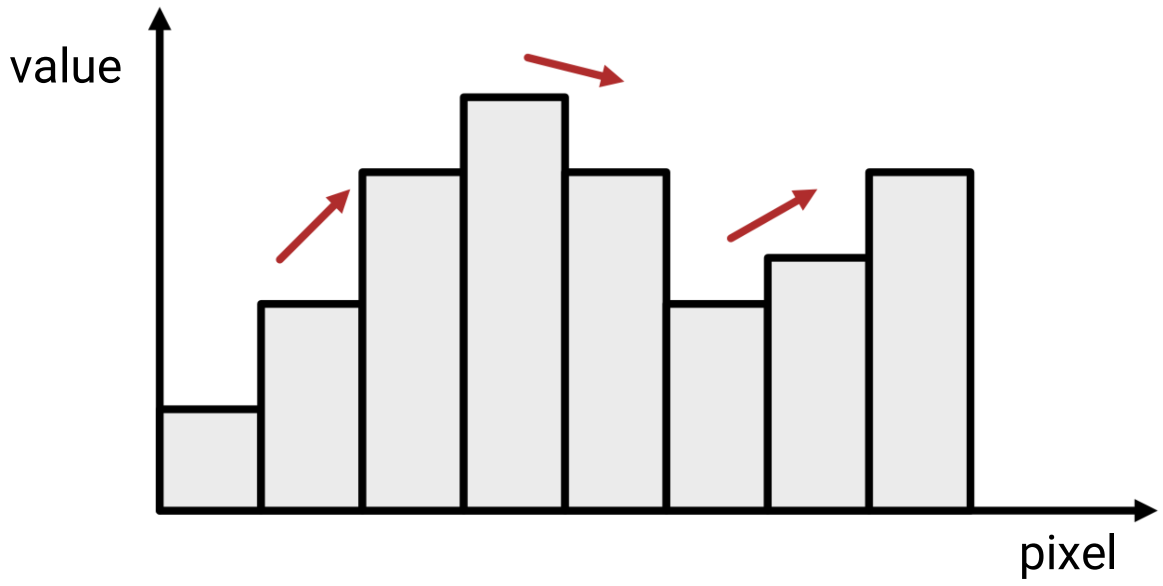
```
float ValueNoise(uint seed, Vector2 p) {
    return RandomValue(seed, p);
}
```

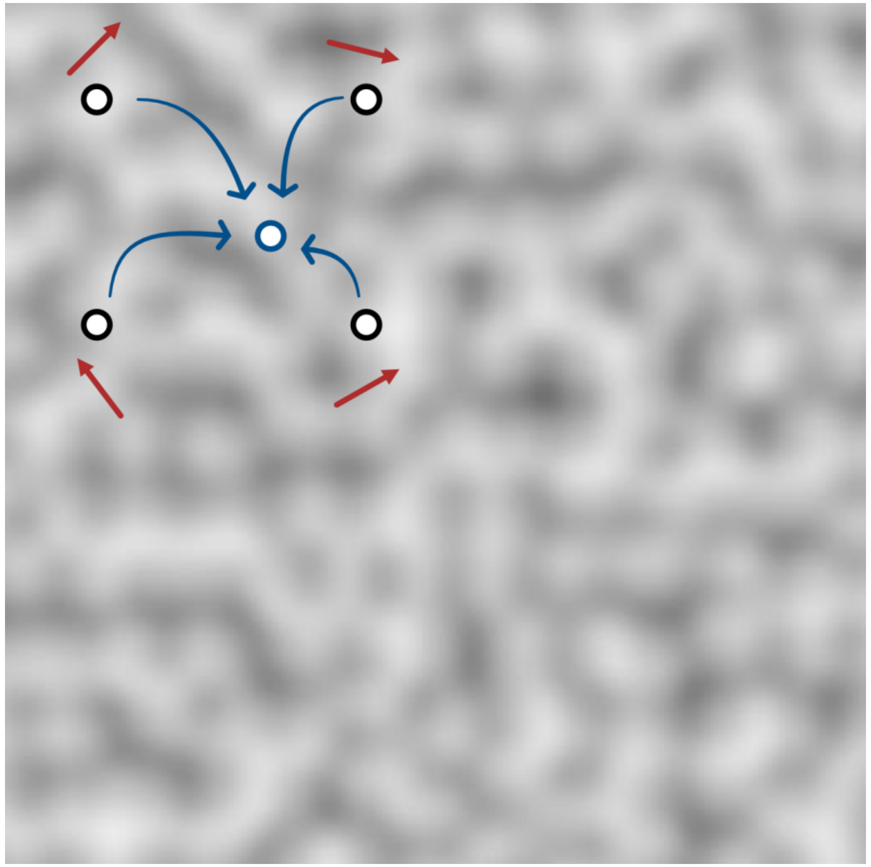




Value noise → Grainy result

Gradient noise: Compute values from random gradients

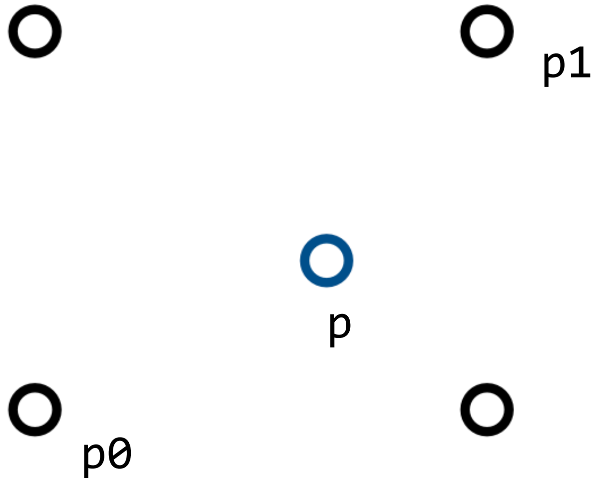




Gradient noise ➔ Smooth result

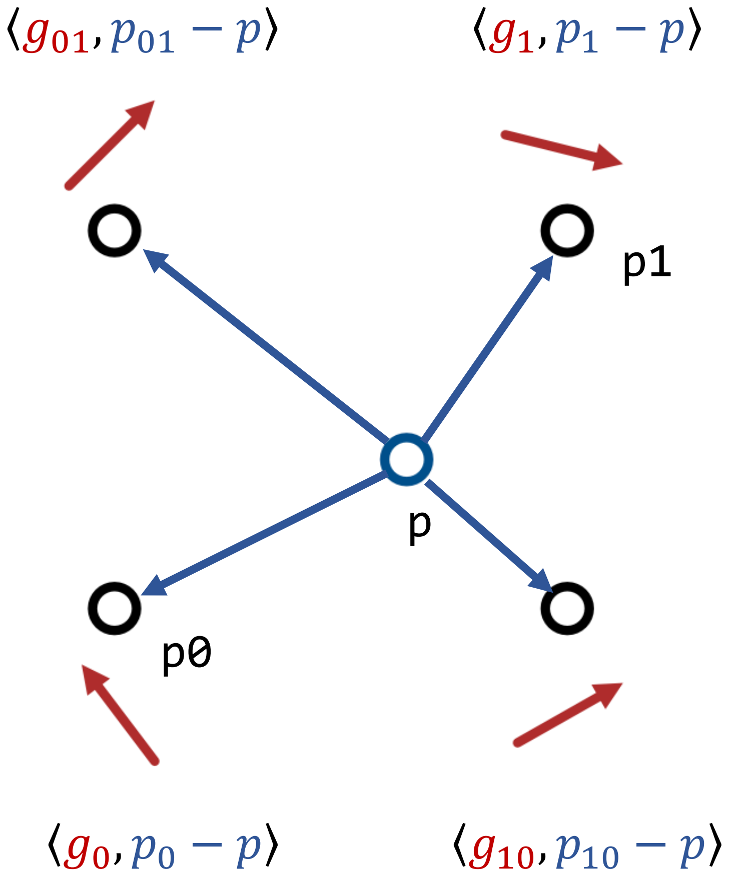
Perlin noise: Random gradients at grid points

```
float PerlinNoise(uint seed, Vector2 p) {

    Vector2 p0 = new(float.Floor(p.X), float.Floor(p.Y));
    Vector2 p1 = p0 + Vector2.One;

}
```



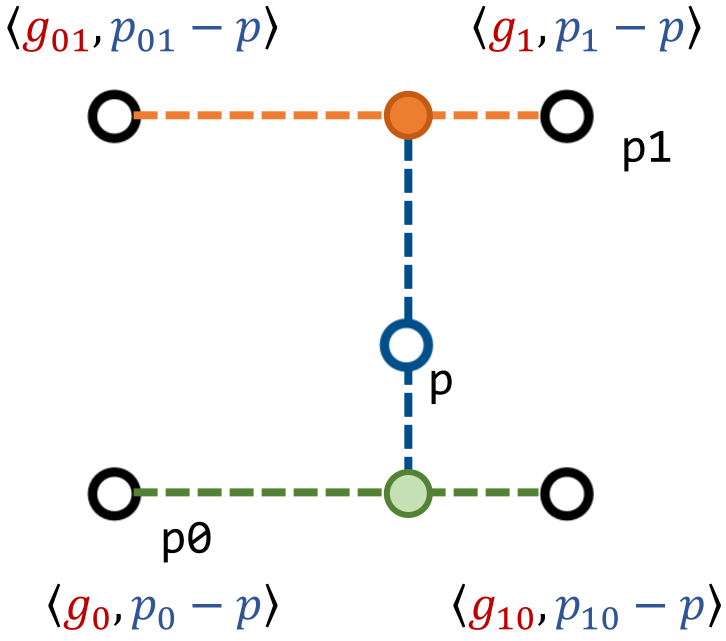
Perlin noise: Compute dot products with gradients

```
float PerlinNoise(uint seed, Vector2 p) {
    float DotGridGradient(Vector2 gridPos)
    => Vector2.Dot(p - gridPos, RandomGradient(seed, gridPos));

    Vector2 p0 = new(float.Floor(p.X), float.Floor(p.Y));
    Vector2 p1 = p0 + Vector2.One;



        DotGridGradient(p0),
        DotGridGradient(new(p1.X, p0.Y)),



        DotGridGradient(new(p0.X, p1.Y)),
        DotGridGradient(p1),




}
```

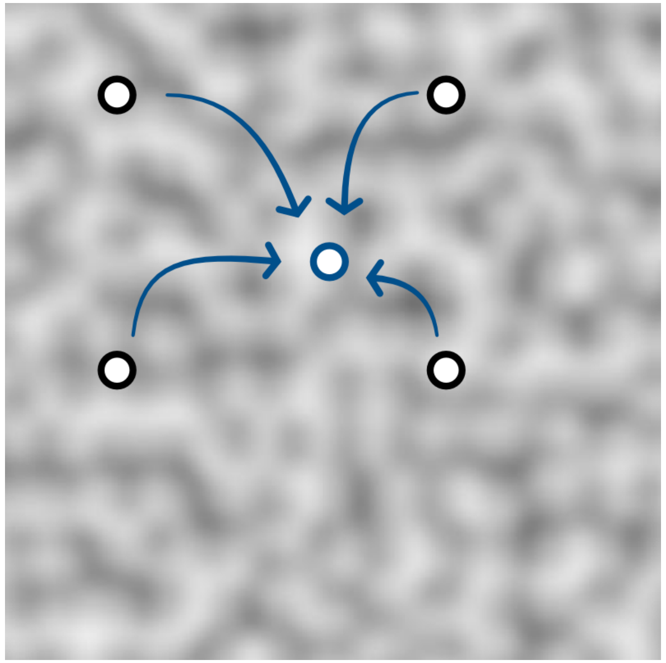


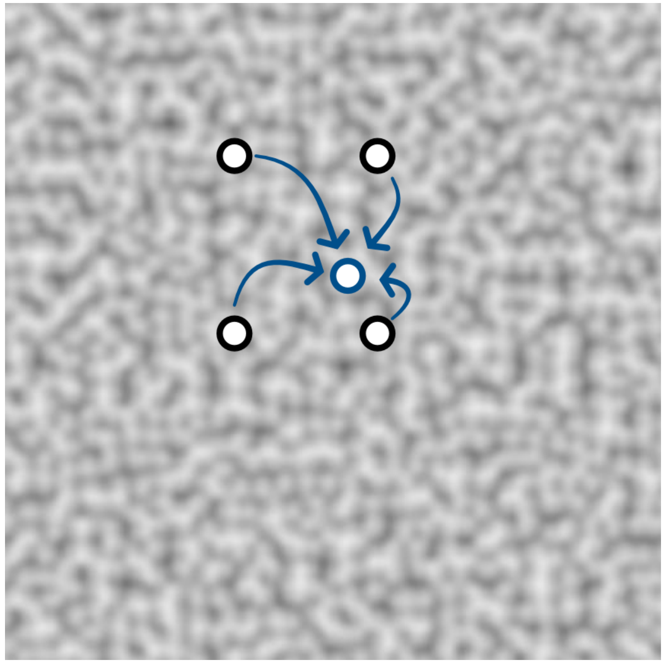
Perlin noise: Bicubic interpolation of dot products

```
float PerlinNoise(uint seed, Vector2 p) {
    float DotGridGradient(Vector2 gridPos)
    => Vector2.Dot(p - gridPos, RandomGradient(seed, gridPos));

    Vector2 p0 = new(float.Floor(p.X), float.Floor(p.Y));
    Vector2 p1 = p0 + Vector2.One;
    Vector2 offset = p - p0;

    float valX1 = Interpolate(
        DotGridGradient(p0),
        DotGridGradient(new(p1.X, p0.Y)),
        offset.X
    );

    float valX2 = Interpolate(
        DotGridGradient(new(p0.X, p1.Y)),
        DotGridGradient(p1),
        offset.X
    );

    float val = Interpolate(valX1, valX2, offset.Y);

    return float.Clamp(0.5f * (val + 1), 0, 1);
}
```



Noise frequency can be tweaked via the grid resolution





Mixing multiple frequencies ➔ More natural result

$$\mathrm{Noise}(p) = \sum_i a_i \,\mathrm{Perlin}(f_i\, p)$$

```
float MixedNoise(uint seed, IEnumerable<(float Frequency, float Amplitude)> components, Vector2 p) {
    float noise = 0.0f;
    foreach (var component in components)
        noise += PerlinNoise(seed, p * component.Frequency) * component.Amplitude;
    return noise;
}
```

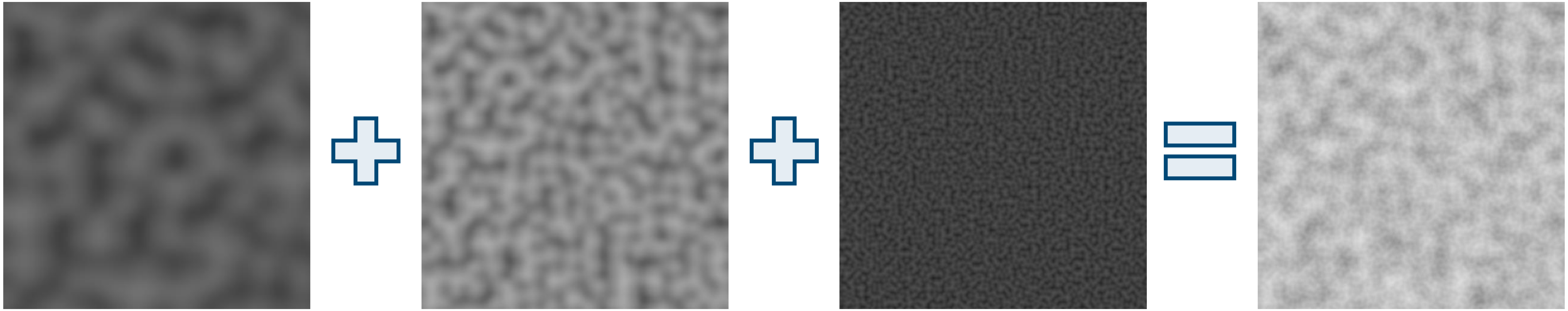


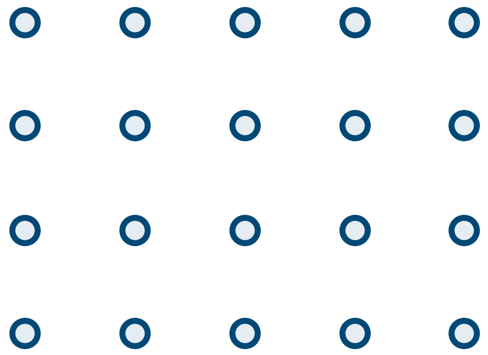
Image Textures

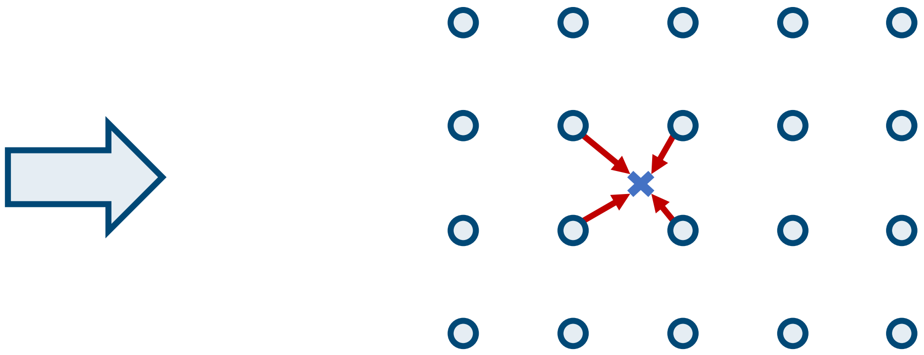
- Obtained via:
  - Painting
  - Photographs (+ manipulation)
  - Simulation

- Benefits: extremely flexible, intuitive creation
- Drawbacks: limited resolution and huge memory cost

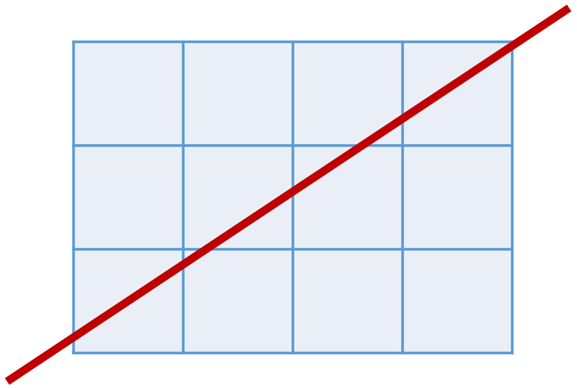
An image is a grid of values

- Pixels are point samples





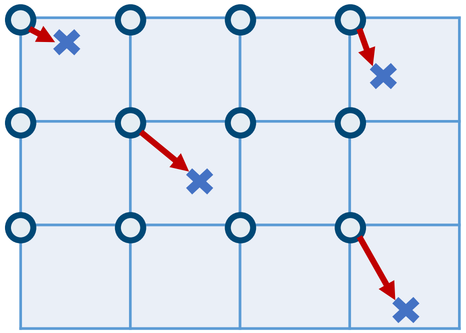
- ***Not*** little squares



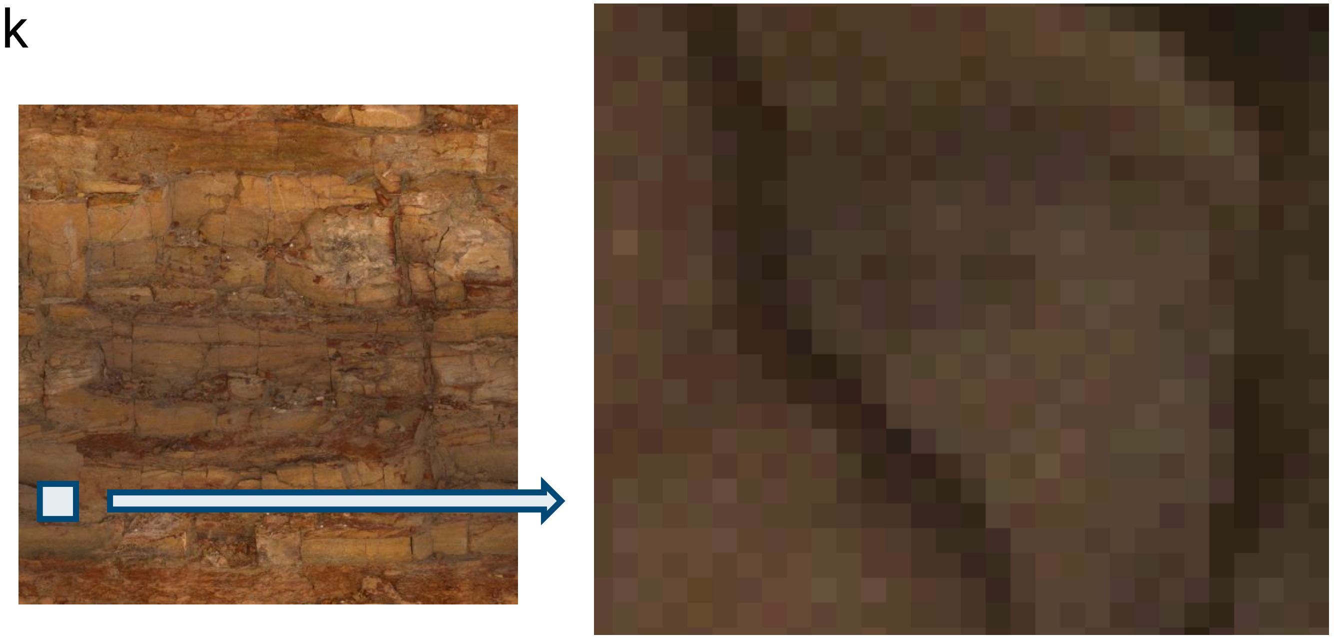
To get a value at a position

we need to interpolate

Nearest-neighbor interpolation

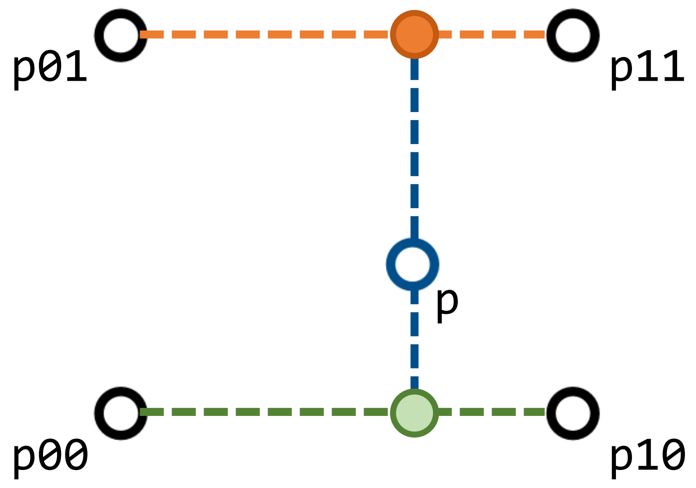


- Round the texture coordinate down to integer:
  - $\lfloor u \cdot w \rfloor$
  - $\lfloor v \cdot h \rfloor$
  - w and h are image width and height in pixels
- Fast, but results in a blocky / “pixelated” look
  - ... though that is sometimes desired!



Bilinear interpolation

- Linearly interpolate between left and right
  - $p_1 = t_x p_{01} + (1 - t_x) p_{11}$
  - $p_0 = t_x p_{00} + (1 - t_x) p_{10}$
- Linearly interpolate result vertically
  - $p = t_y p_0 + (1 - t_y) p_1$



- For even smoother results:
  - Bicubic interpolation

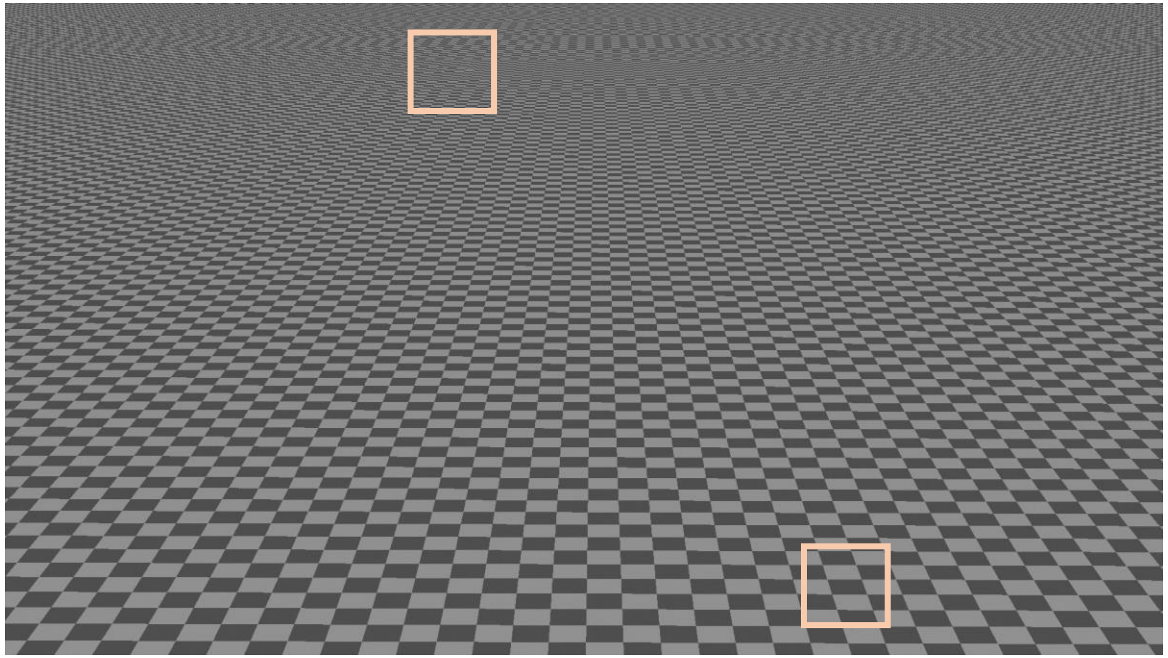
Reading materials

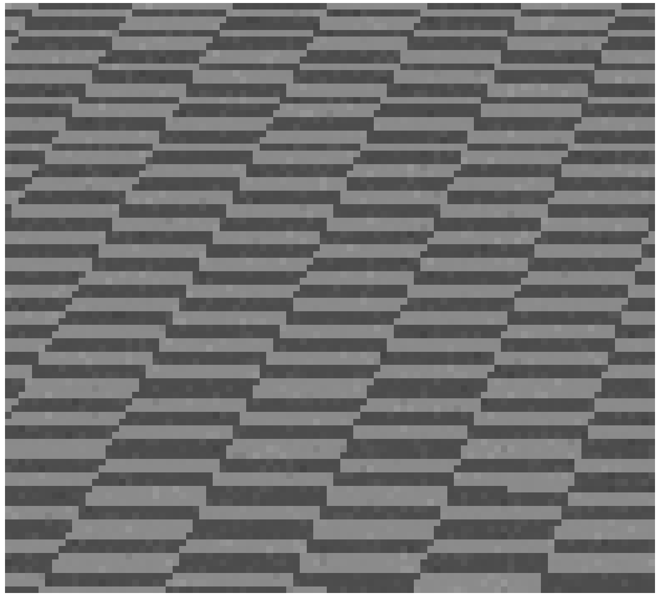
- Ken Perlin. 1985. An Image Synthesizer.
- https://pbr-book.org/4ed/Textures_and_Materials

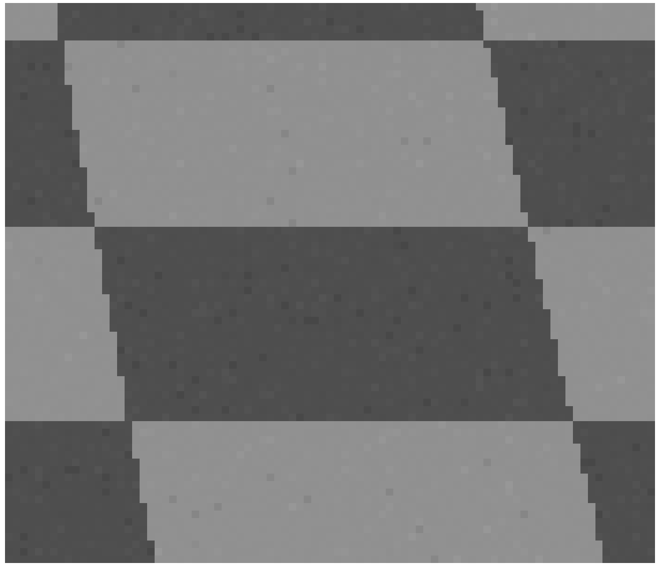
Anti-aliasing and texture filtering

What is aliasing?

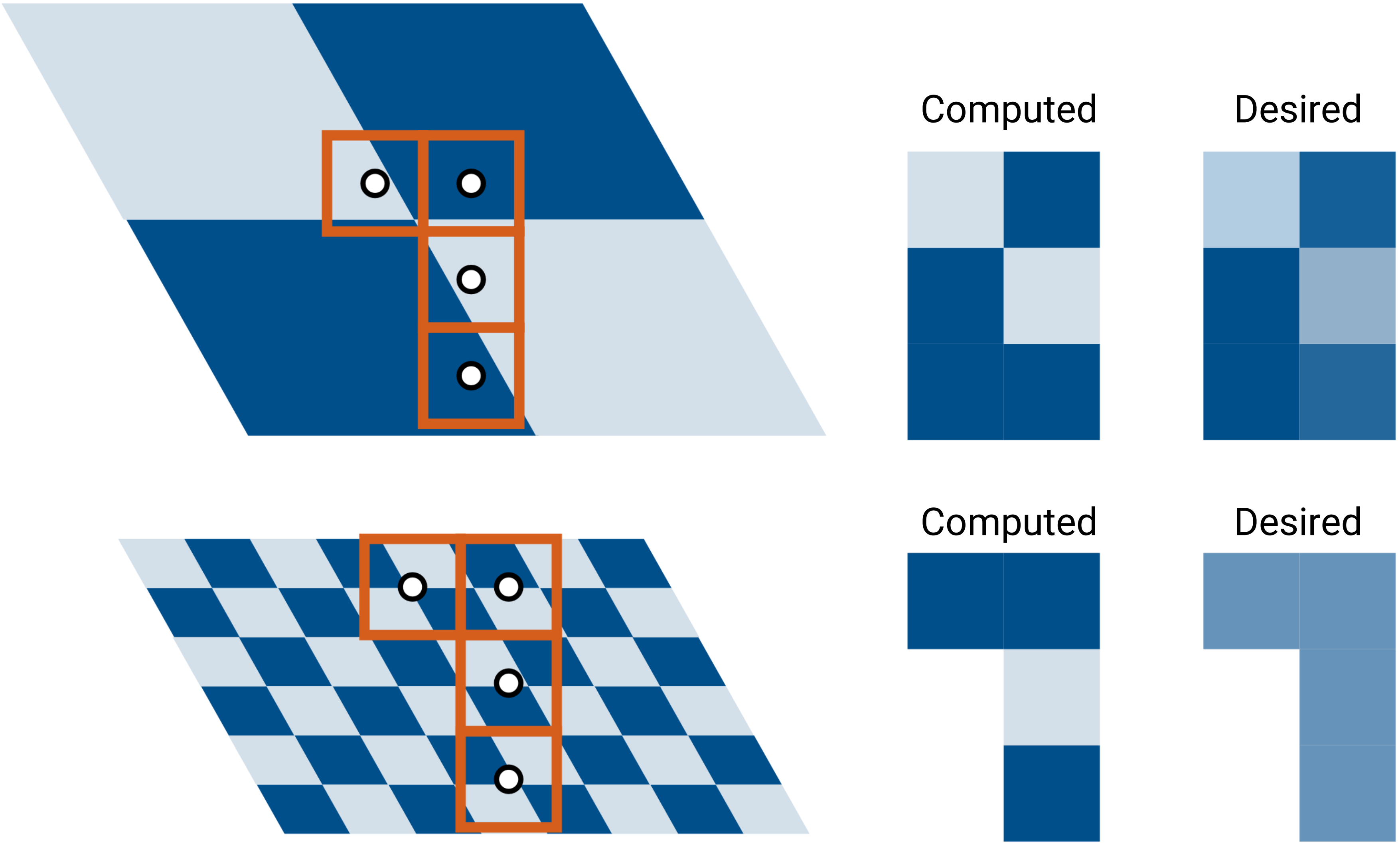
- Jagged edges and distorted shapes
- If the sample count is too low to capture all details



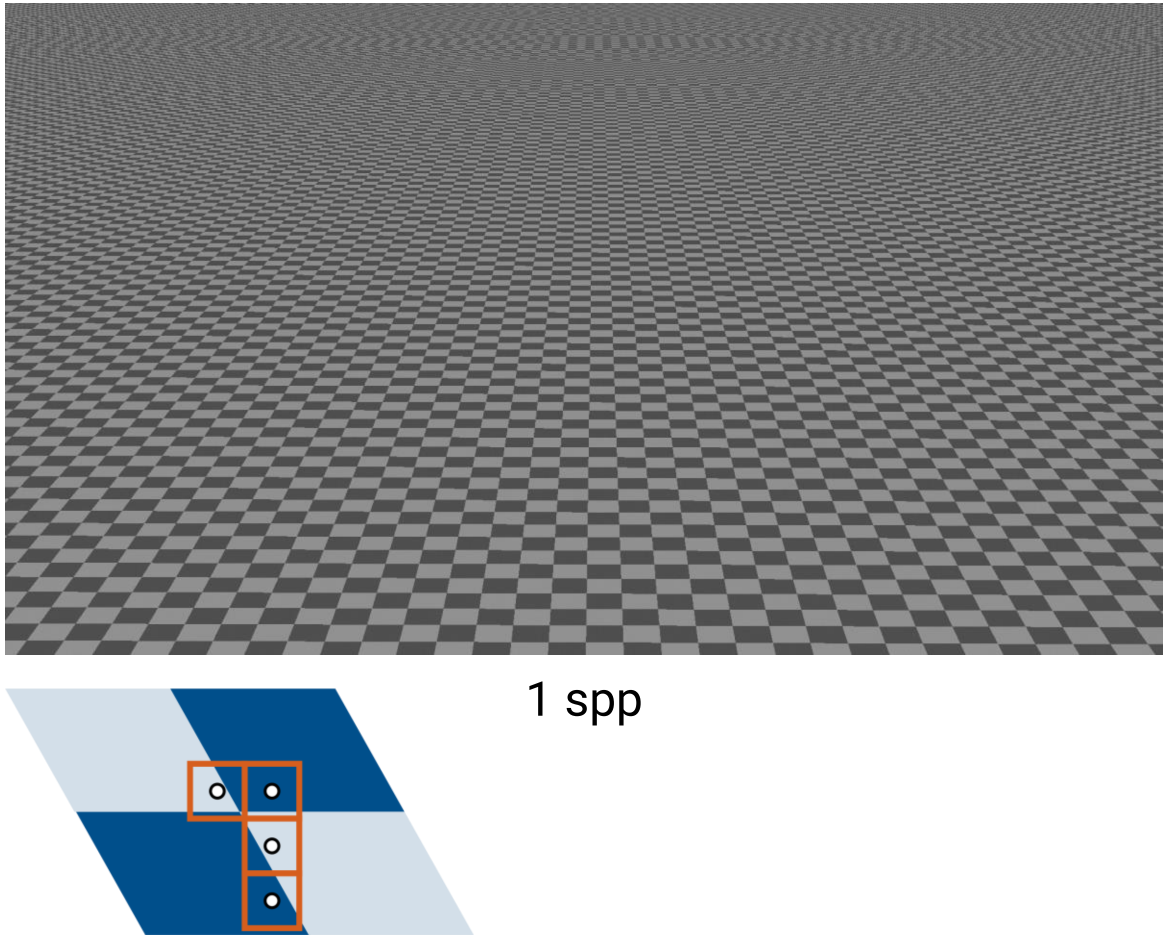




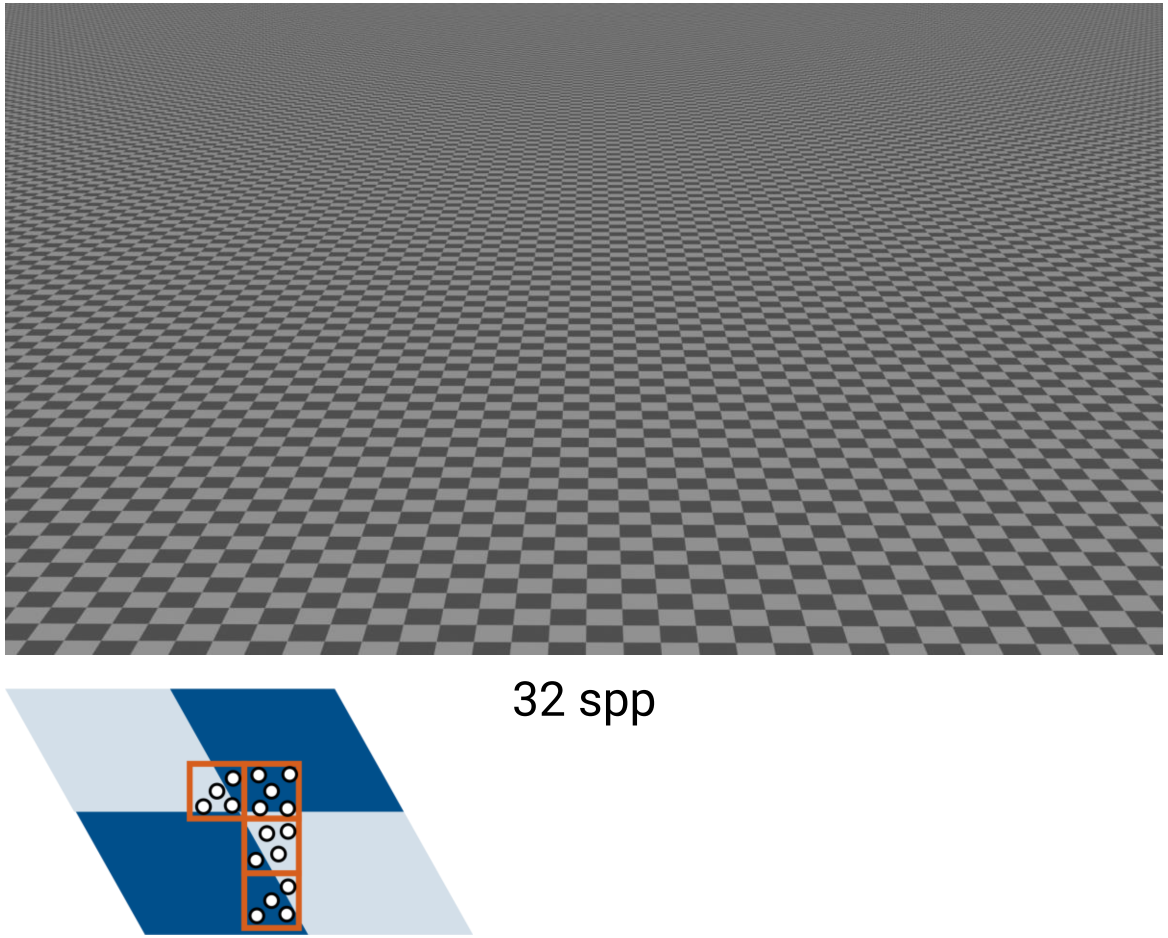
Example values with Alias



Naively getting rid of alias is simple: Just use more samples

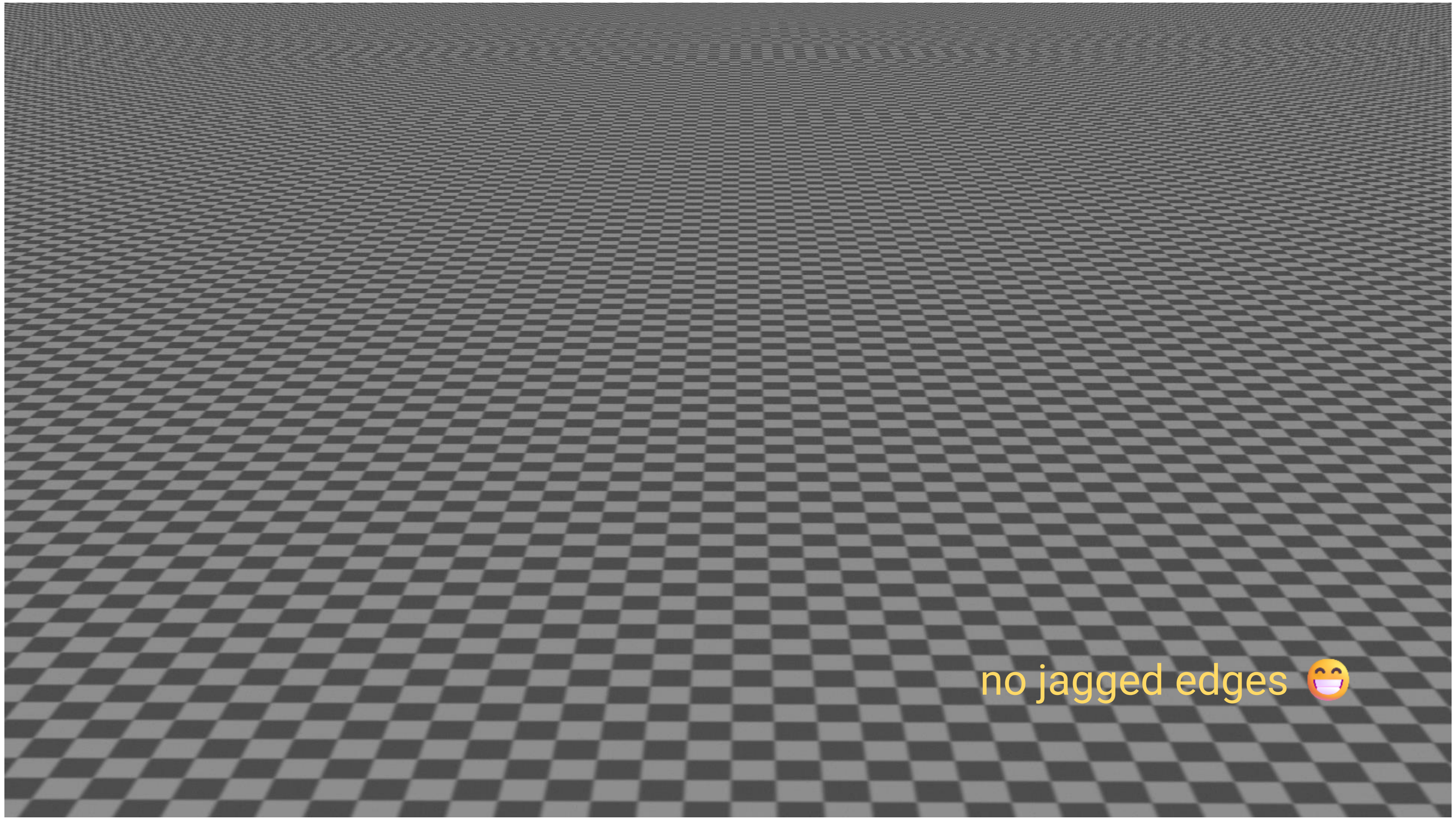


1 spp



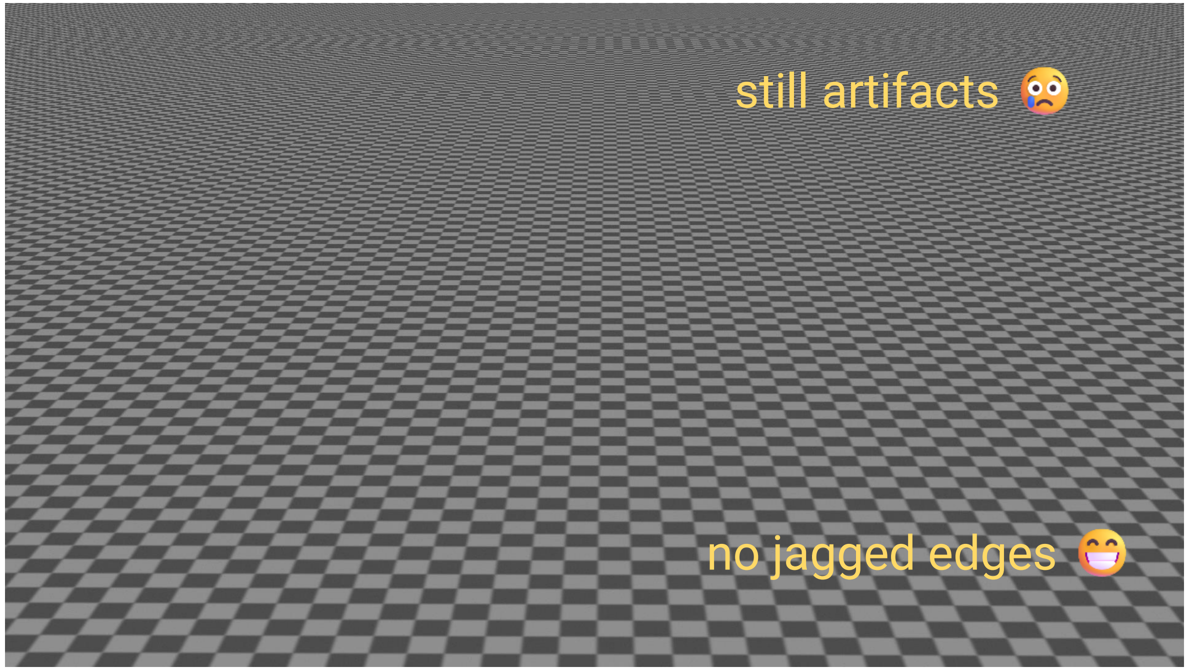
32 spp

Alias can be avoided by *prefiltering* the texture

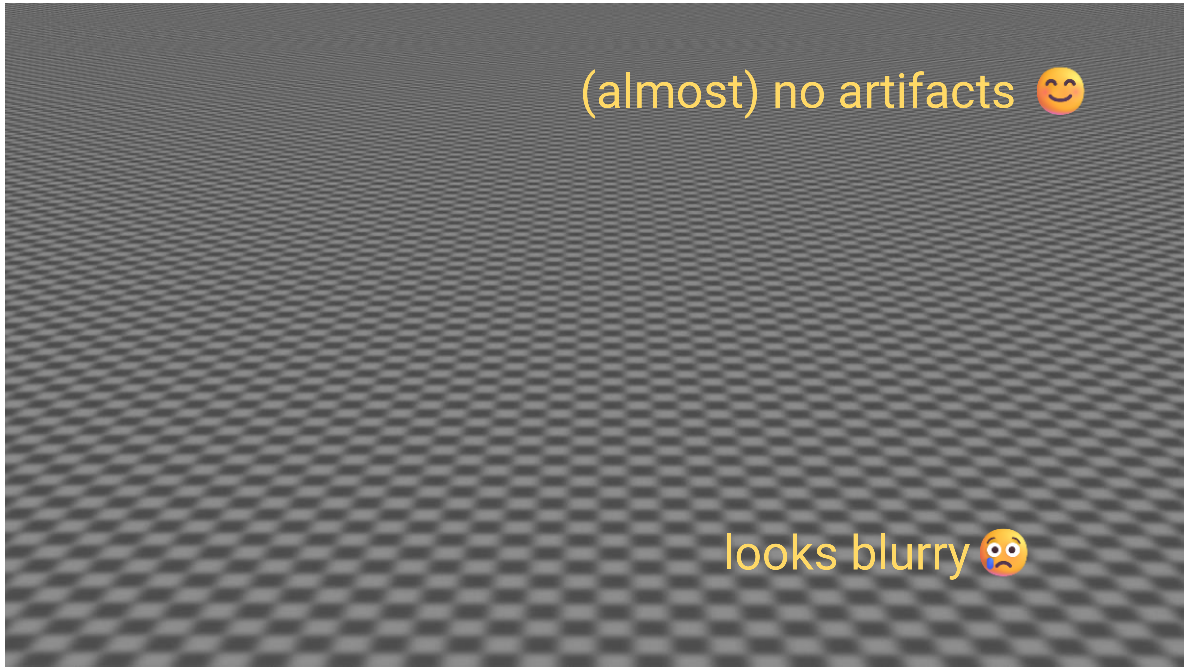


2px Gaussian blur

But how much filtering do we need?



2px Gaussian blur

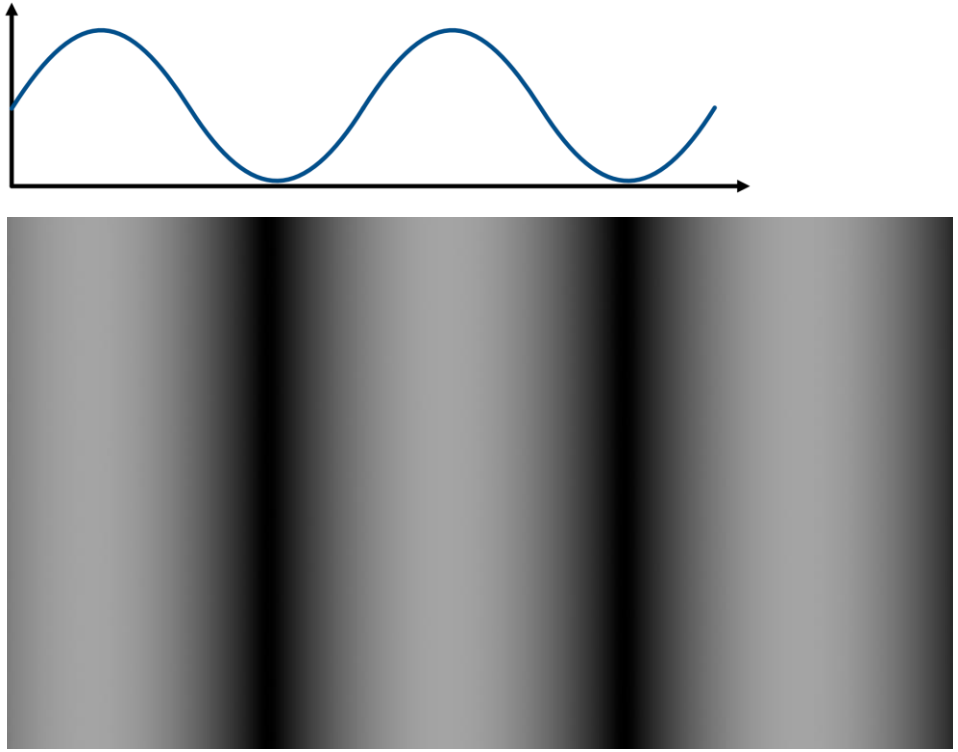


5px Gaussian blur

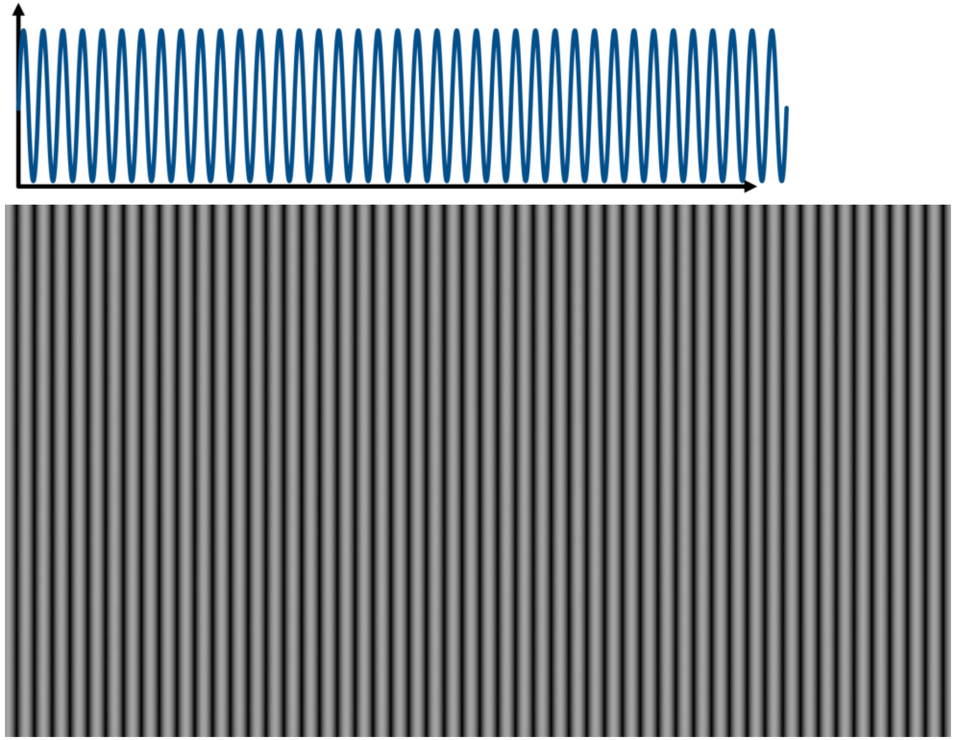
To answer that, we turn to Fourier analysis

High-frequency and low-frequency images

- Some trivial examples



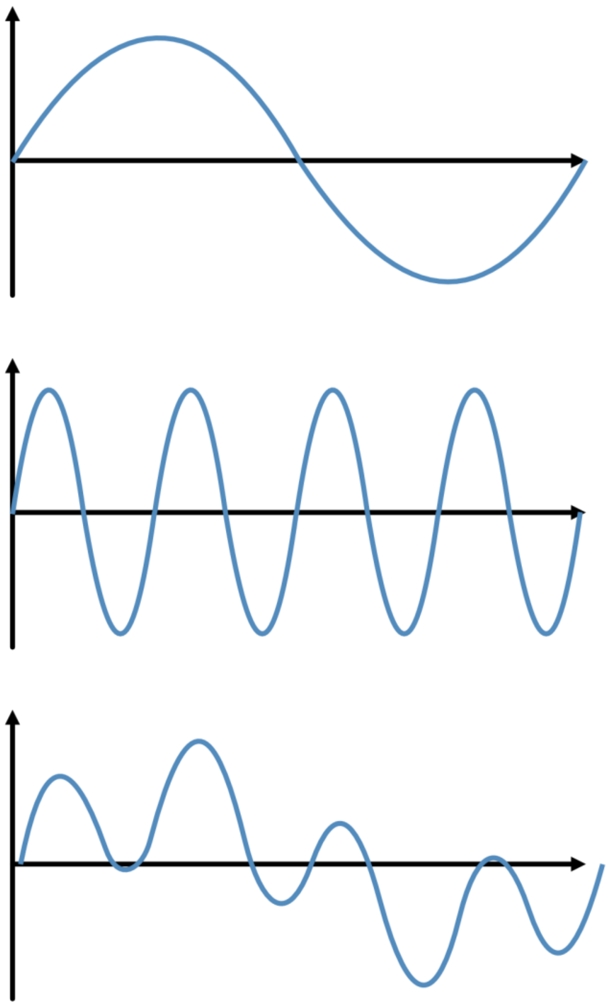
Low frequency = slowly changing signal



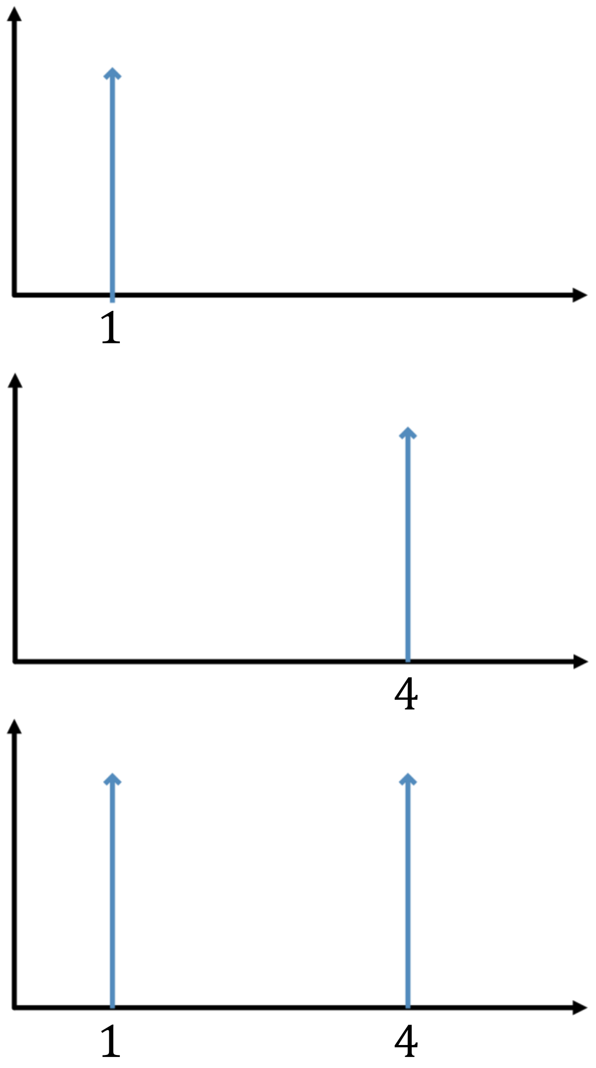
High frequency = rapidly changing signal

Images typically contain many different frequencies

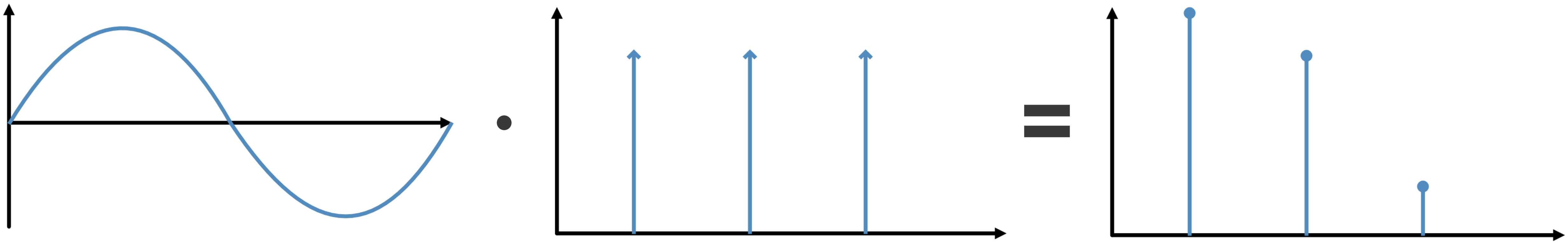
Fourier analysis decomposes a signal into its frequencies



$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\xi x} dx$$



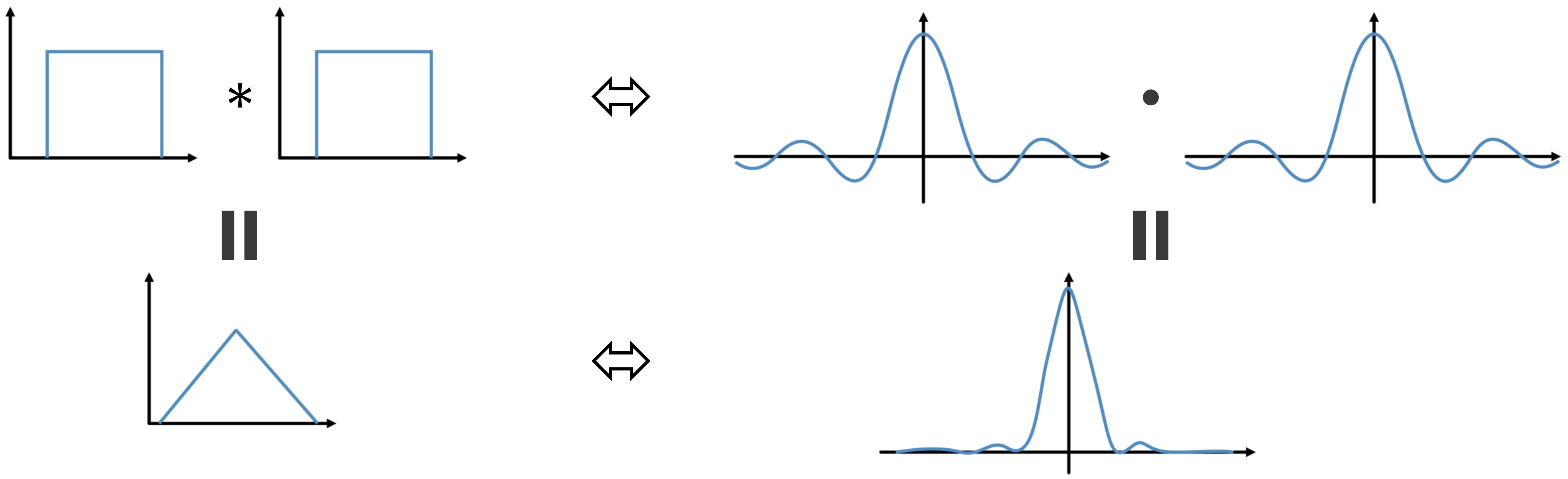
Sampling as a multiplication with a comb function



“slight” stretch of what a Dirac delta is...
But good enough for our intuition 😅

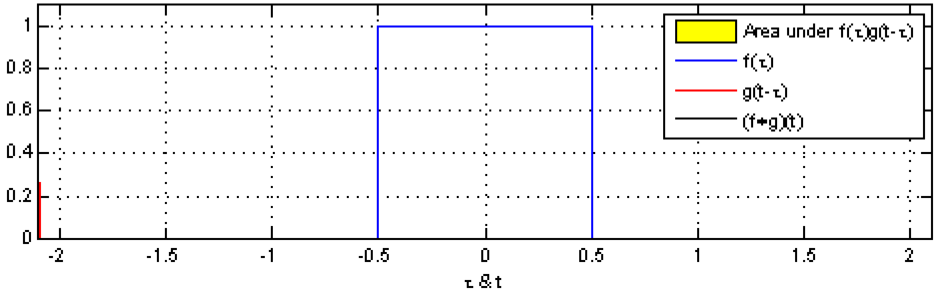
Multiplication in space ⇔ convolution in frequency domain

Convolution in space ⇔ Multiplication in frequency domain



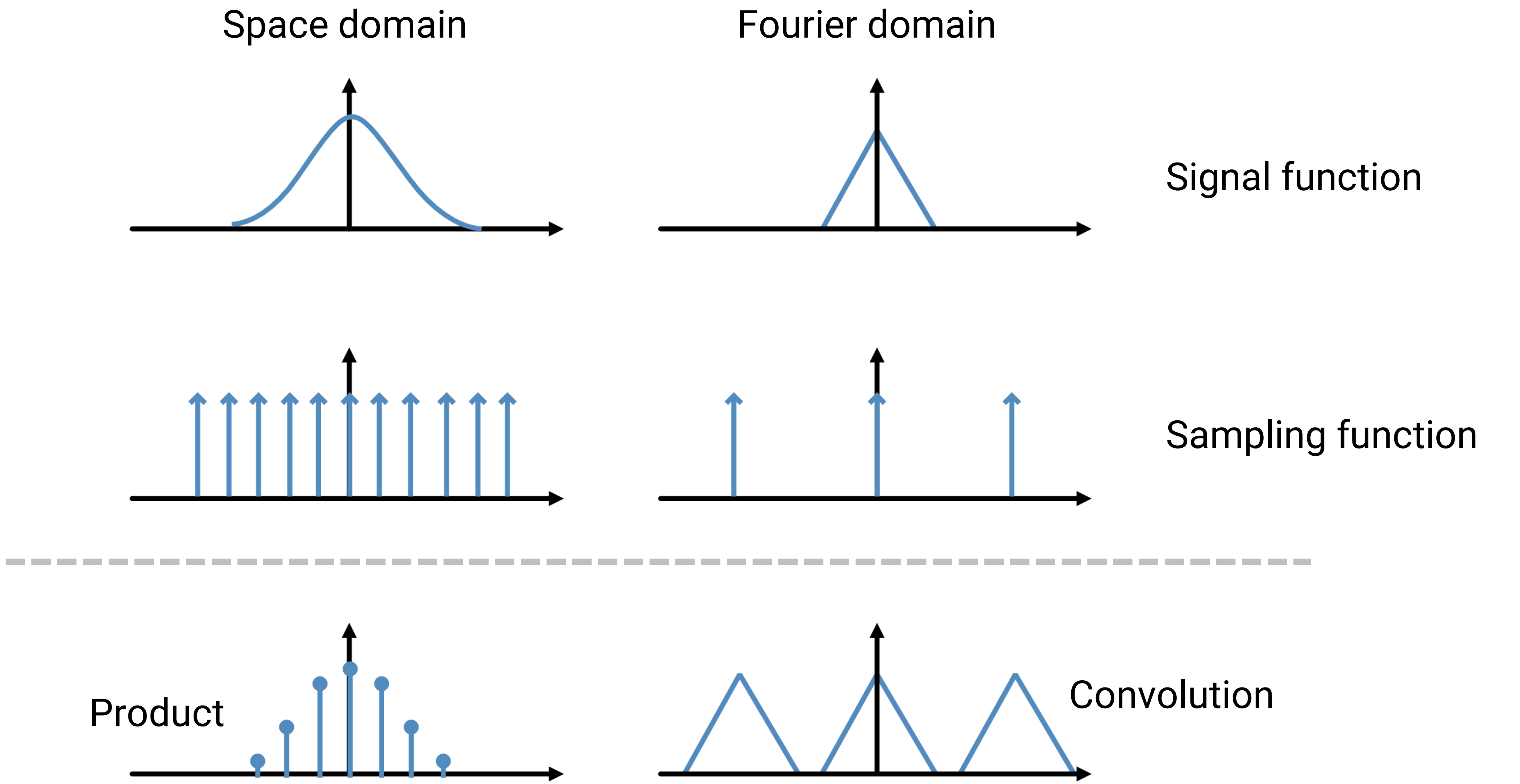
Convolution (a fancy term for filtering)

$$f(x) * g(x) = \int f(x')g(x - x')dx'$$

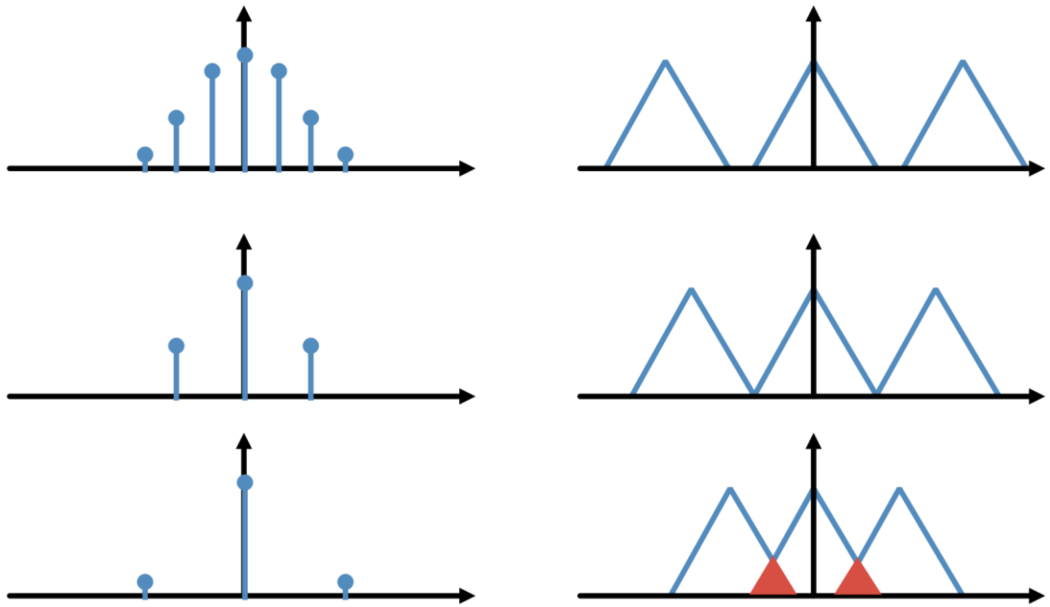


https://en.wikipedia.org/wiki/Convolution

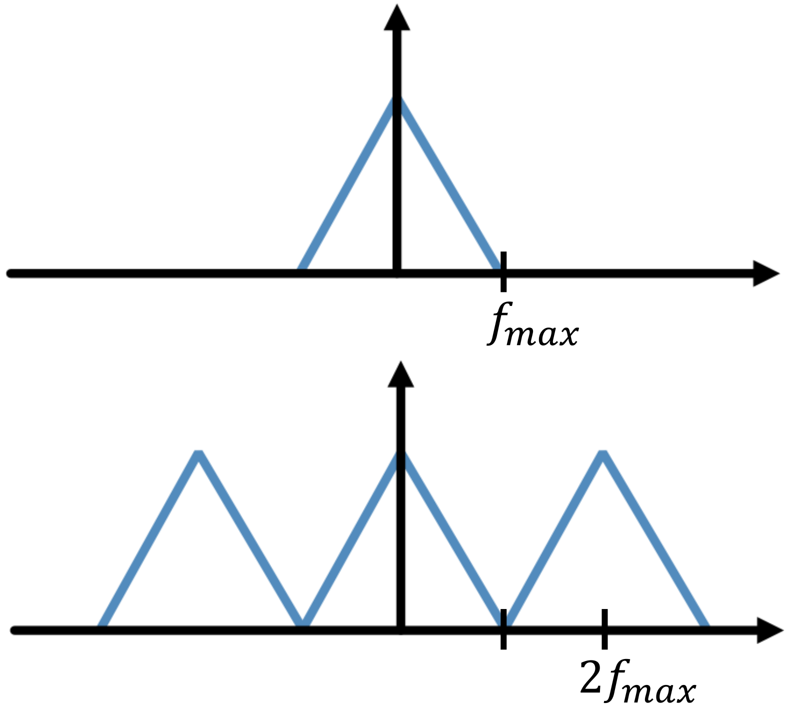
In Fourier space: Sampling is a convolution with a comb



Alias ⇔ Overlap



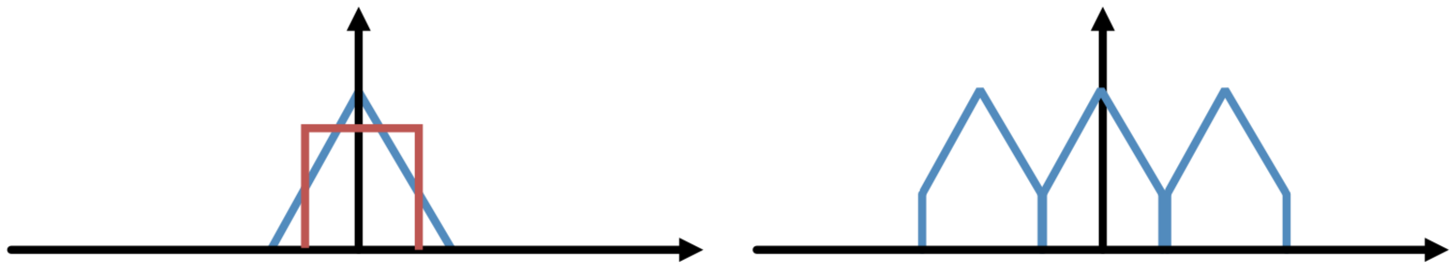
Nyquist theorem:
No alias if sample rate ≥ highest signal frequency



Minimal sample distance without overlap

Prefiltering: remove all frequencies above the sample rate

- Ideal low-pass: multiplication with a box in frequency domain



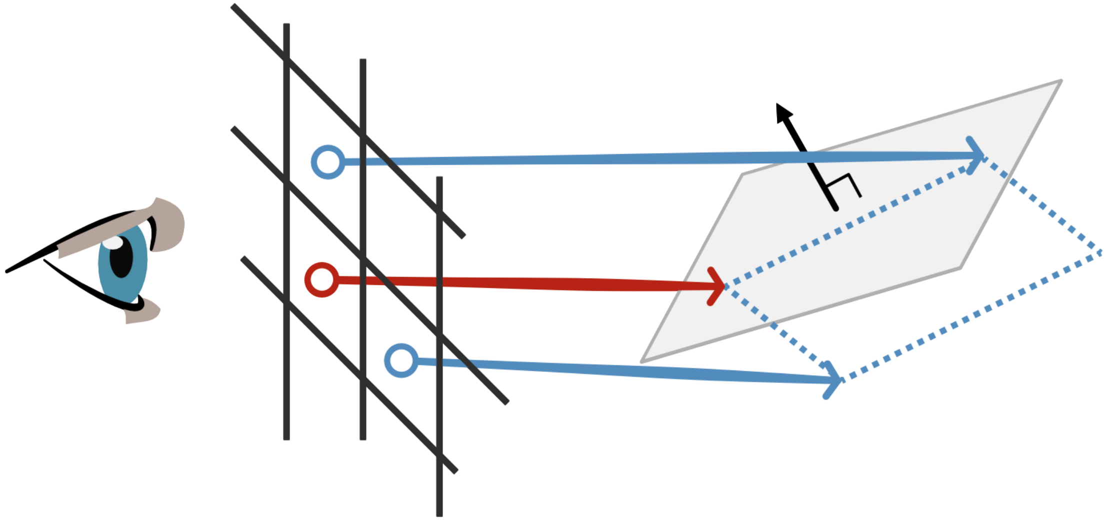
  - We loose high-frequency detail
  - But we don’t destroy more than that (as alias does!)

- In image space: convolution with a sinc
  - Costly: infinite support
  - But we can use a cheaper, non-ideal, low-pass filter as a surrogate

Texture filtering in practice

Ray differentials to determine the pixel footprint

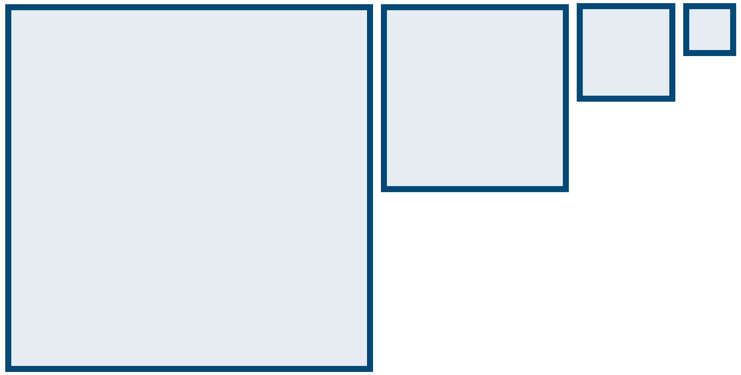
- Track additional rays to the side and above
  - Not actually intersected with the geometry
  - Hitpoints approximated from the normal n, assuming planar surface



- With 1 spp, the texture should be low-pass filtered with roughly this shape
  - But how to do that efficiently?

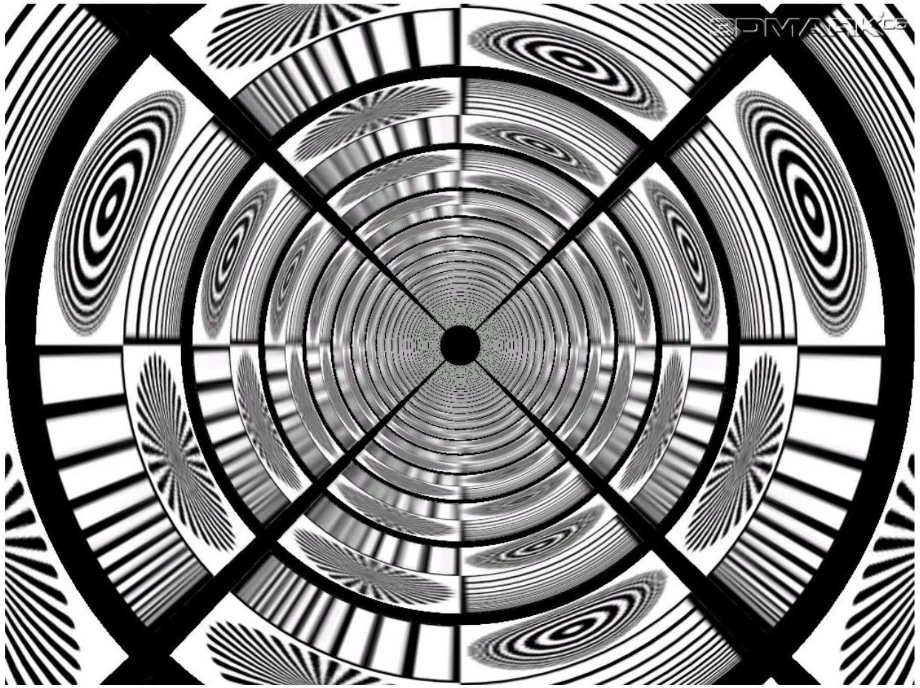
MIP-maps (Multum In Parvo = “much in little”)

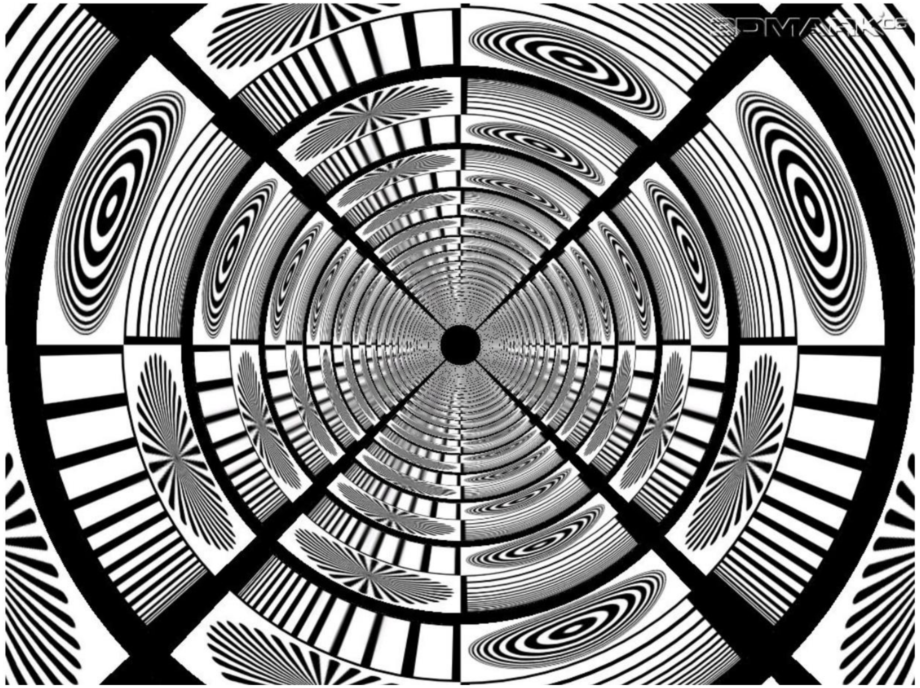
- Store image at multiple resolutions
  - Computed in pre-process
  - Each level uses half the resolution of the previous



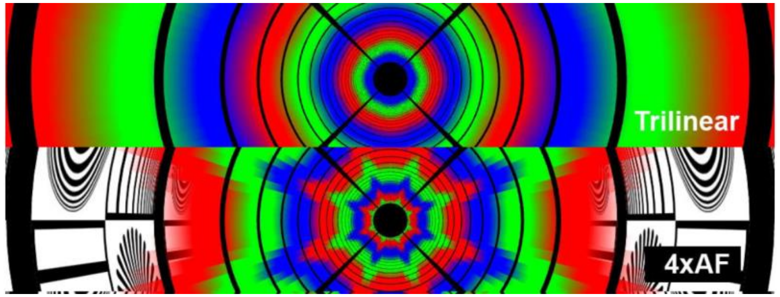
- Little overhead
  - Less than twice the memory of the original texture
- Rendering
  - Pick level based on pixel footprint
  - Interpolate (to avoid visible, abrupt transitions)

Trilinear (left) vs anisotropic (right)





Anisotropic filtering reduces *unnecessary* blur



Color = MIP level

Reading materials

- https://pbr-book.org/4ed/Textures_and_Materials/Texture_Sampling_and_Antialiasing

- For more on Fourier transforms, image filtering, etc

  ➔ Image Processing and Computer Vision (IPCV) core lecture

Summary

Topics in this block

- Rendering equation
- Basic radiometry
- Simple BRDF models
- Simple light sources
- Textures

➔ Now we can render textured diffuse surfaces and mirrors under simple direct illumination