Computer Graphics

Textures

Konstantin Tretyakov

kt@ut.ee

Oct 23, 2013

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)***

Quiz

- Colors are usually described using three numbers, because ____________________

- Name three different colorspaces: ____________

- The “mother” of all colorspaces is _________

- What colorspace is used by your digital camera to store pictures?

Quiz

- Colors are usually described using three numbers, because *there are three types of cone cells in our eyes*.
- Name three different colorspaces:
  e.g. CIE XYZ, CIE RGB, Adobe RGB, CMYK, sRGB, …
- The “mother” of all colorspaces is *CIE XYZ*
- What colorspace is used by your digital camera to store pictures? *sRGB*

Quiz

- Conversion from RGB to sRGB is called

  ____

  and is performed as follows

  sRGB := _____________

Quiz

- Conversion from RGB to sRGB is called *gamma correction / gamma encoding*

  and is performed as follows

  $$\text{sRGB} \approx \text{RGB}^{\frac{1}{2.2}}$$

Quiz

- Phong lighting model
  - Single light component, single light source.
  - Light source intensity (I_a, I_d, I_s)
  - Material parameters (M_a, M_d, M_s), specular shininess 10
  - Quadratic attenuation

____

Quiz

- Write down the Phong lighting model for:
  - Single light component, single light source.
  - Light source intensity (I_a, I_d, I_s)
  - Material parameters (M_a, M_d, M_s), specular shininess 10
  - Quadratic attenuation

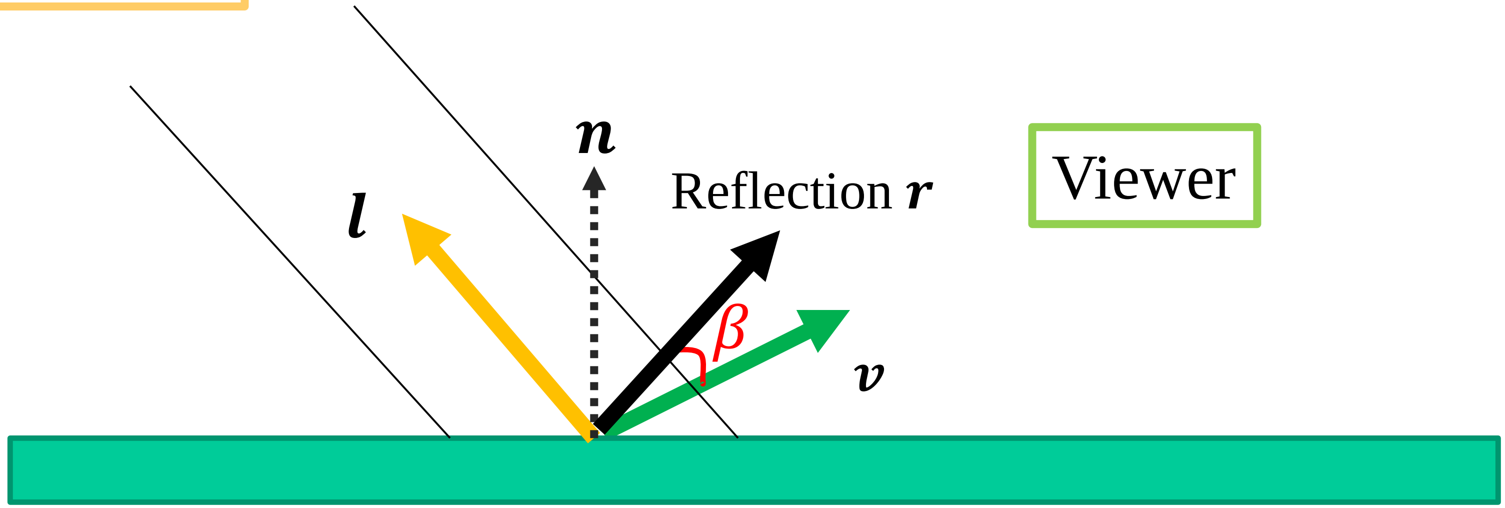
$$I_a M_a + \frac{1}{d^2}\left(I_d M_d (\boldsymbol{n}^T \boldsymbol{l})_+ + I_s M_s {\left(\boldsymbol{r}^{\boldsymbol{T}} \boldsymbol{v}\right)_+}^{10}\right)$$

More basic lighting

- **“Blinn specular term”**
  - Rather than using $\boldsymbol{r}^T\boldsymbol{v}$ in the specular highlight computation, use $\boldsymbol{n}^T\boldsymbol{h}$, where $\boldsymbol{h}$ is the *half-angle*:

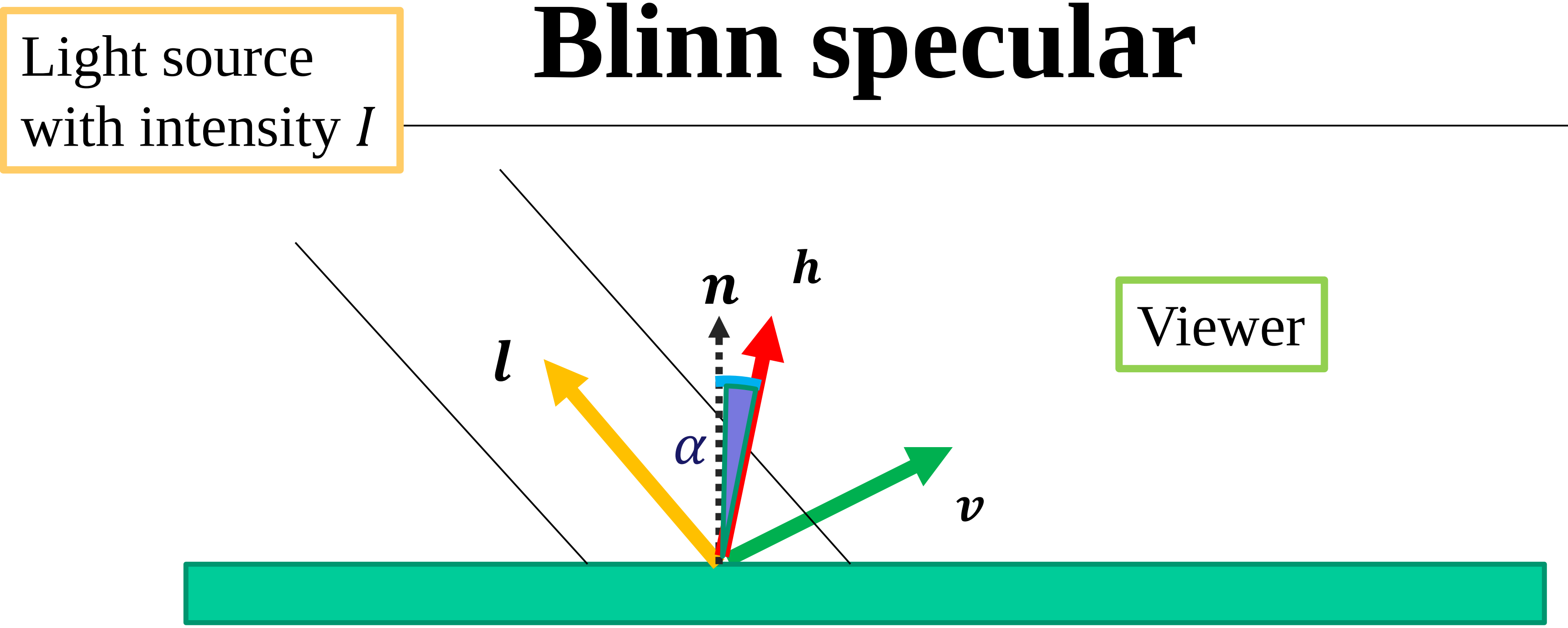
$$\boldsymbol{h} = normalize\left(\frac{\boldsymbol{l}+\boldsymbol{v}}{2}\right)$$

Phong specular



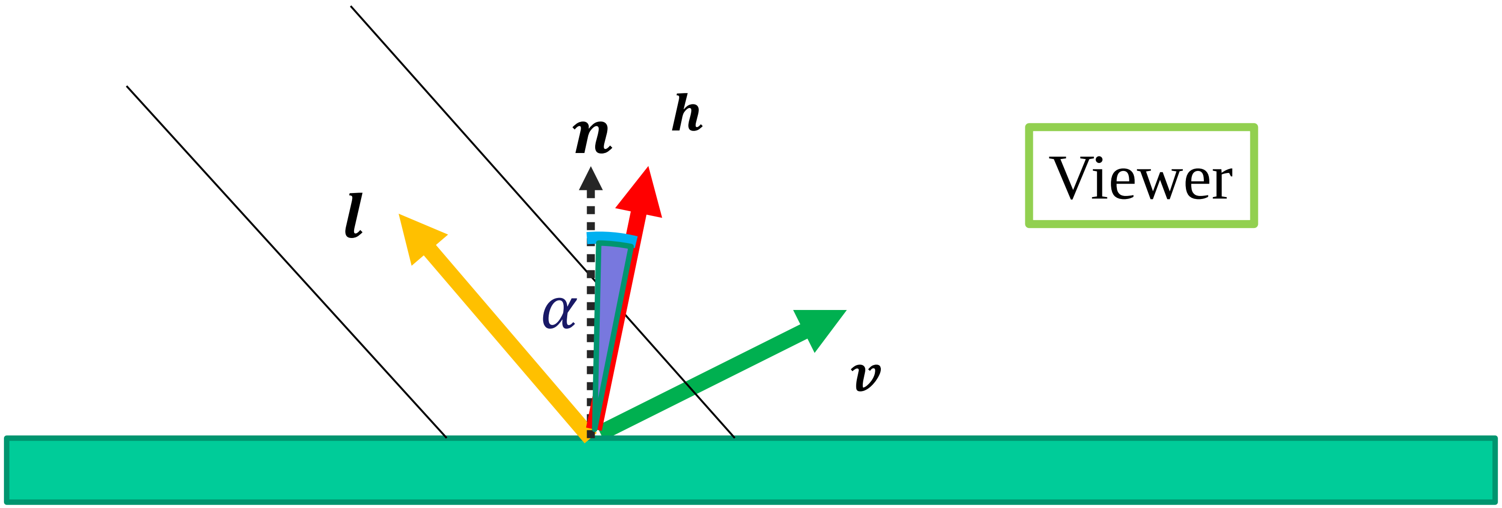
$$(\cos\beta)^s = (\boldsymbol{r}^T\boldsymbol{v})^s$$

Blinn specular

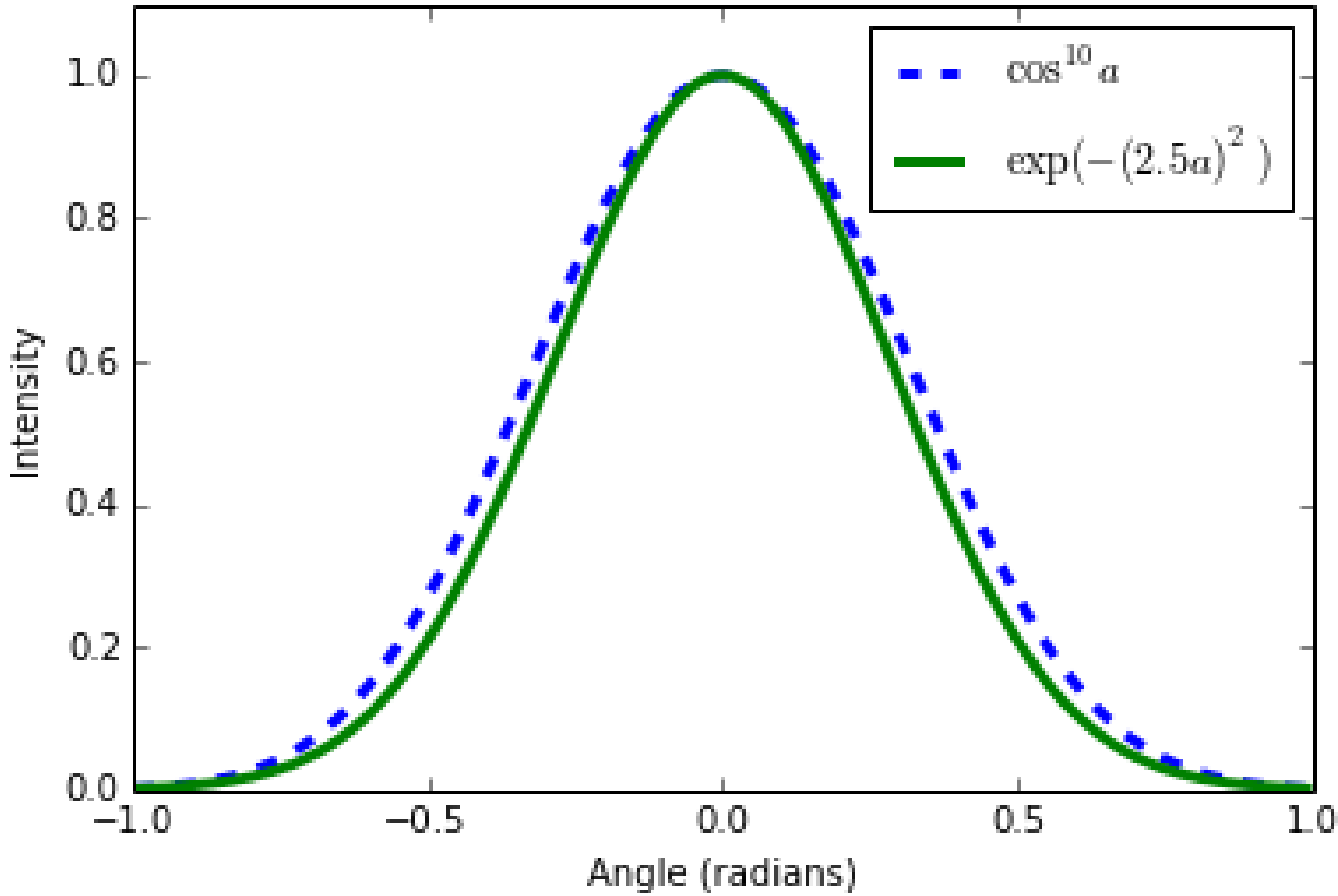


$$(\cos\alpha)^s = (\boldsymbol{n}^T\boldsymbol{h})^s$$

Gaussian-Blinn specular



$$e^{-\frac{\alpha^2}{\sigma^2}} = e^{-\frac{\arccos(\boldsymbol{n}^T\boldsymbol{h})^2}{\sigma^2}}$$



$\cos^{10} a$
$\exp(-(2.5a)^2)$
1.0
0.8
0.6
0.4
0.2
0.0
Intensity
−1.0
−0.5
0.0
0.5
1.0
Angle (radians)

More basic lighting

- **"Spotlight"**, GL_SPOT_xxx
  - Attenuate according to the cosine of the angle between light vector and "spotlight direction"

$$I_a M_a + \frac{1}{d^2}(\ldots)$$

becomes

$$I_a M_a + \left(-\boldsymbol{l}^T \mathbf{spot\_dir}\right)_+^{\mathrm{spot\_exponent}} \cdot \frac{1}{d^2}(\ldots)$$

More basic lighting

- **"Emissive material"**, GL_EMISSION
  - Simply add a constant "emission intensity" term to the light equation:

$$I_a M_a + \text{attenuation} \cdot (\ldots)$$

becomes

$$\boldsymbol{M_e} + I_a M_a + \text{attenuation} \cdot (\ldots)$$

More basic lighting

- **"Fog"**, glEnable(GL_FOG), glFog(..)
  - Weaken color intensity in proportion to distance to the viewer (or fragment depth).

I.e. after computing

$$\text{color} := M_e + I_a M_a + \text{attenuation} \cdot (\ldots)$$

blend the result with the "fog color":

$$t \cdot \text{color} + (1 - t) \cdot \text{fogColor}$$

Where $t = f(\text{distance})$

Fog functions

- **Linear (GL_LINEAR):**
  - $f(d) = \frac{end - d}{end - start}$

- **Exponential (GL_EXP):**
  - $f(d) = e^{-\lambda d}$

- **Exponential quadratic (GL_EXP2):**
  - $f(d) = e^{-(\lambda d)^2}$

Basic lighting summary

- Phong model
  - Ambient, Diffuse, Specular, Emission
  - Per-vertex & per-fragment
- Half-vector & Blinn specular term
- Gauss specular term
- Fog
  - Linear, Exponential, Exponential quadratic

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)***

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

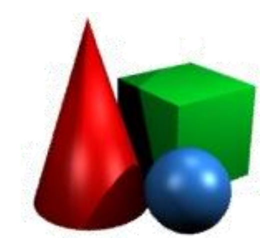
Fixed color

For each pixel, determine its **color**

Lighting model

Visibility tests & blending

Draw pixel *(if needed)*



Standard Graphics Pipeline

Where do we take the colors or lighting model parameters (material, normal, …) from?

Fixed color

Lighting model

color

Standard Graphics Pipeline

So far:

1. Specify globally or per-triangle, or

2. Specify per-vertex and interpolate

Fixed color

Lighting model

color

Standard Graphics Pipeline

Today:
3. Specify with a much higher granularity using a *texture*.

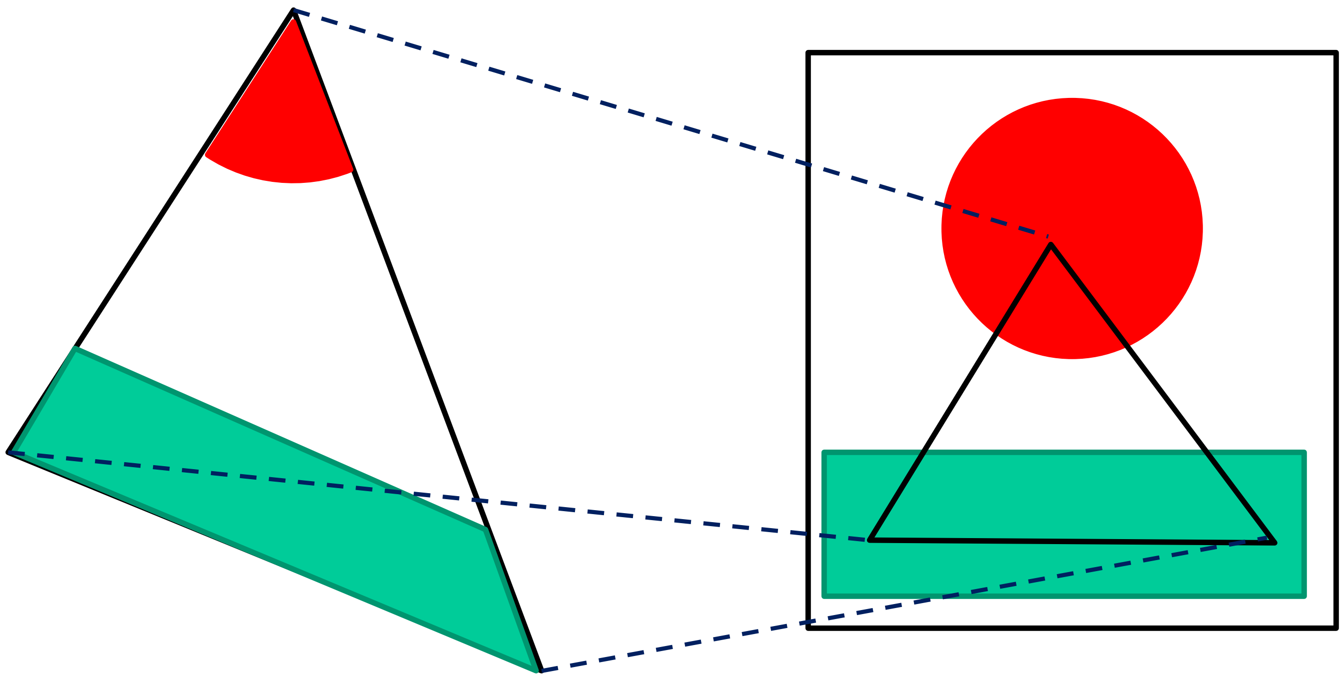
Fixed color

Lighting model

color

Example: Textured triangle

- We can specify color of every pixel of a triangle by mapping it from a *texture image*.



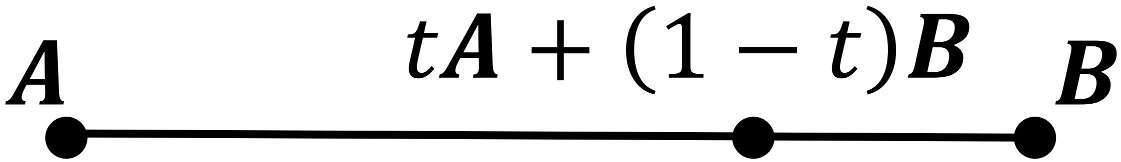
Below the tip of the iceberg

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

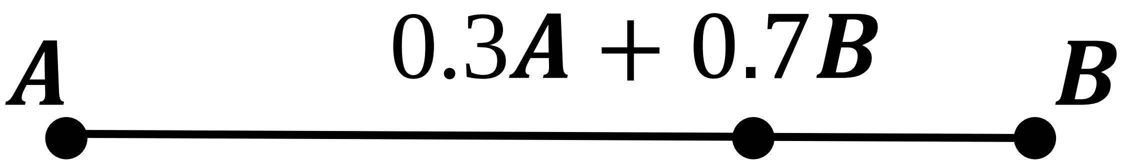
Below the tip of the iceberg

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

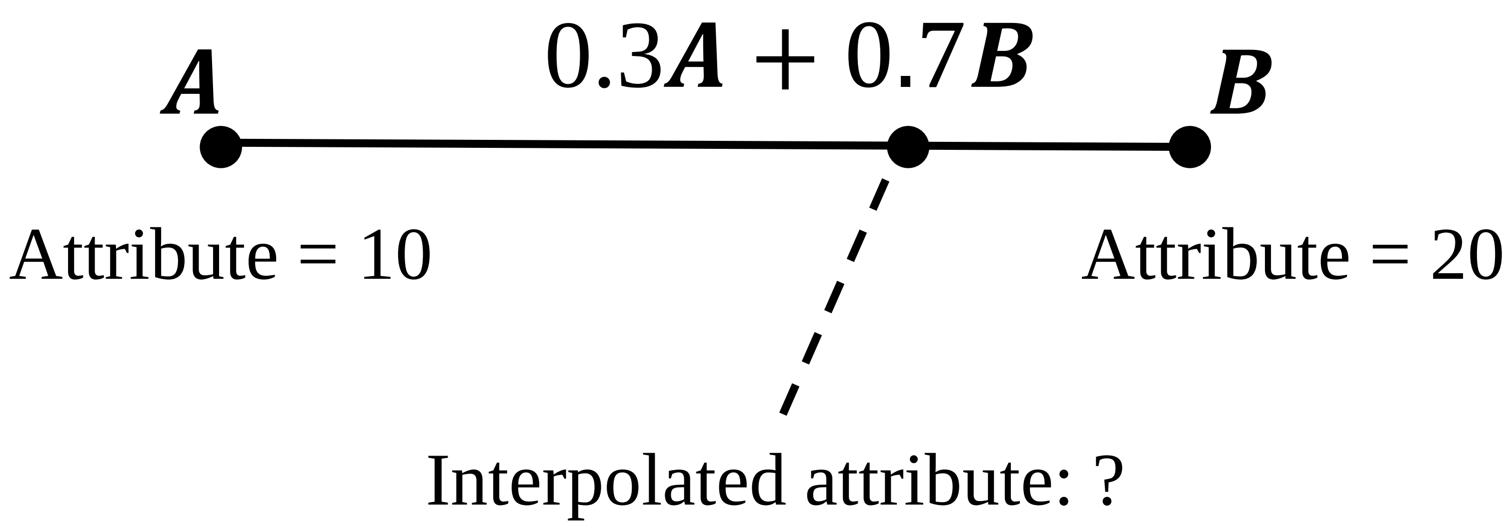
Linear interpolation



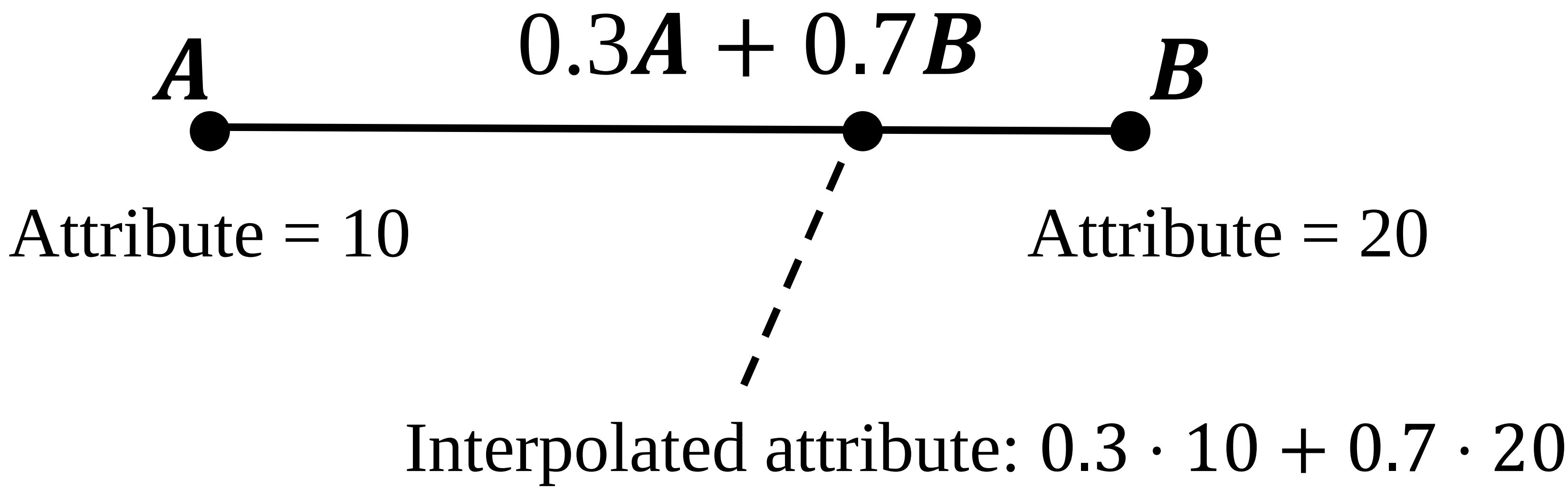
Linear interpolation



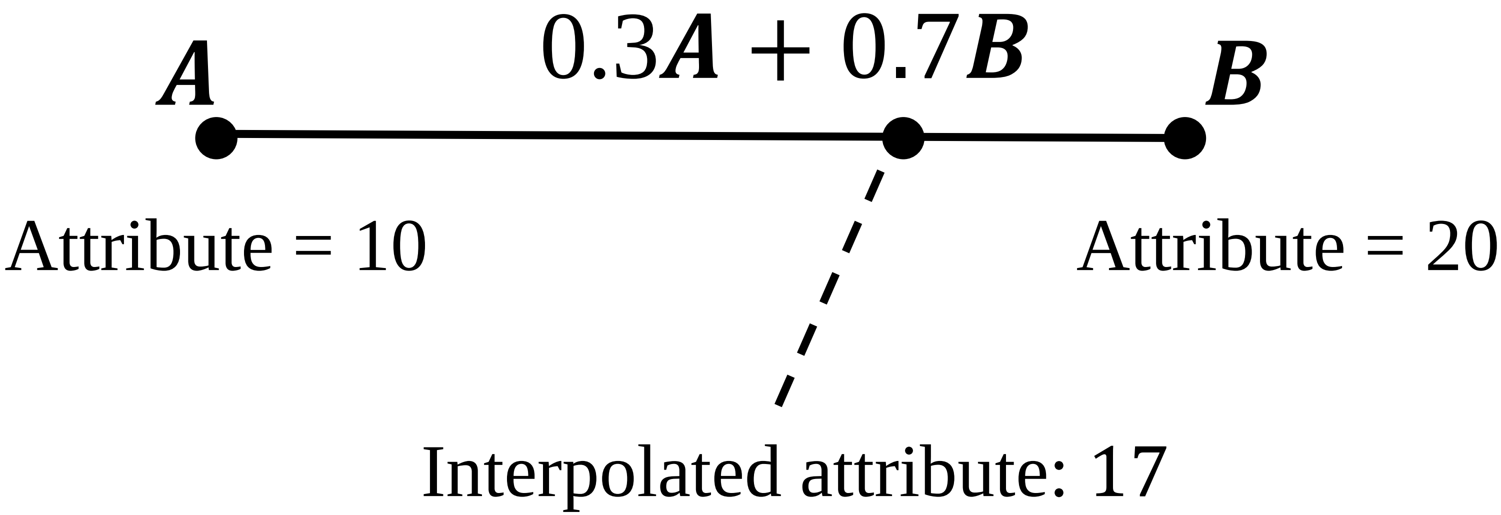
Linear interpolation



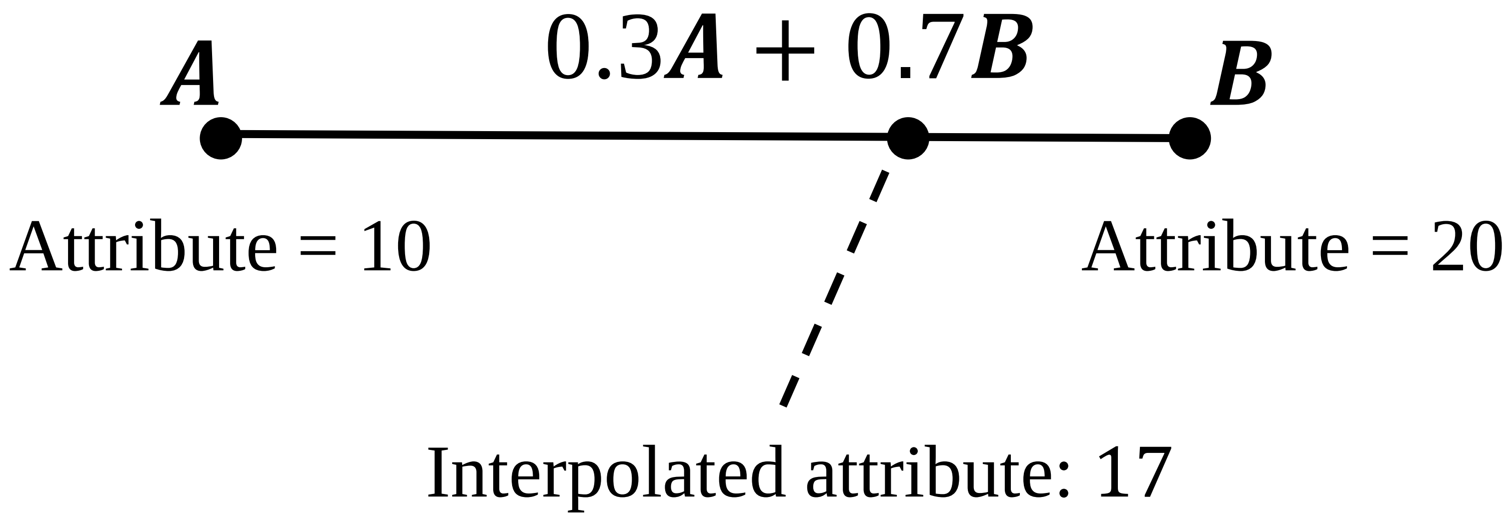
Linear interpolation



Linear interpolation

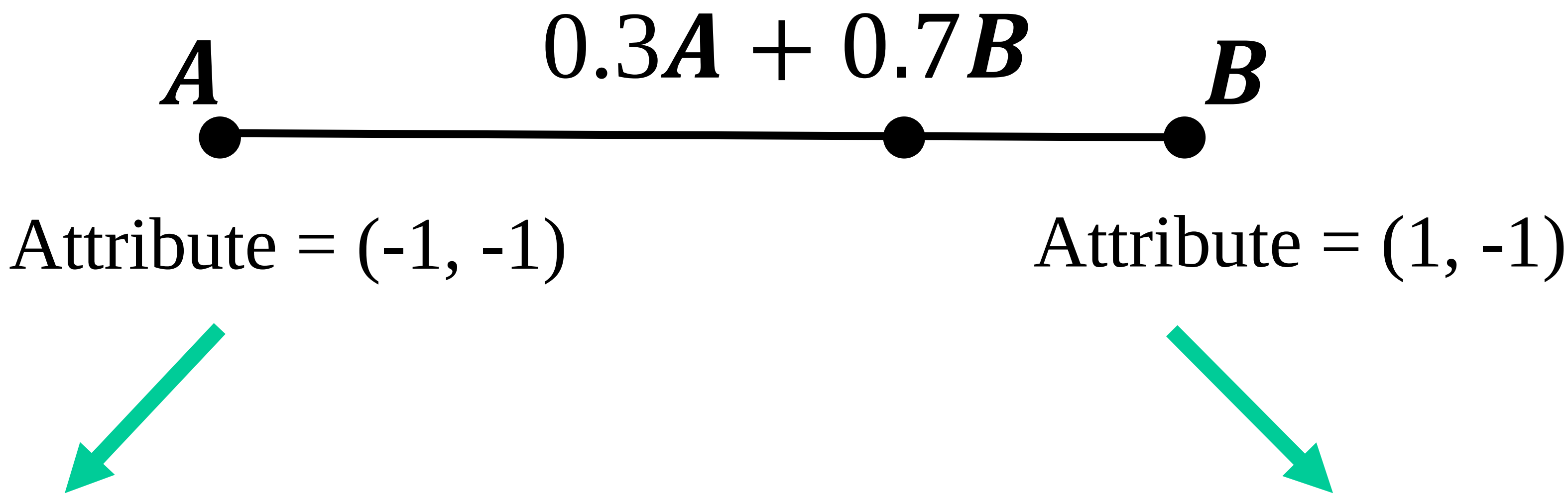


Linear interpolation

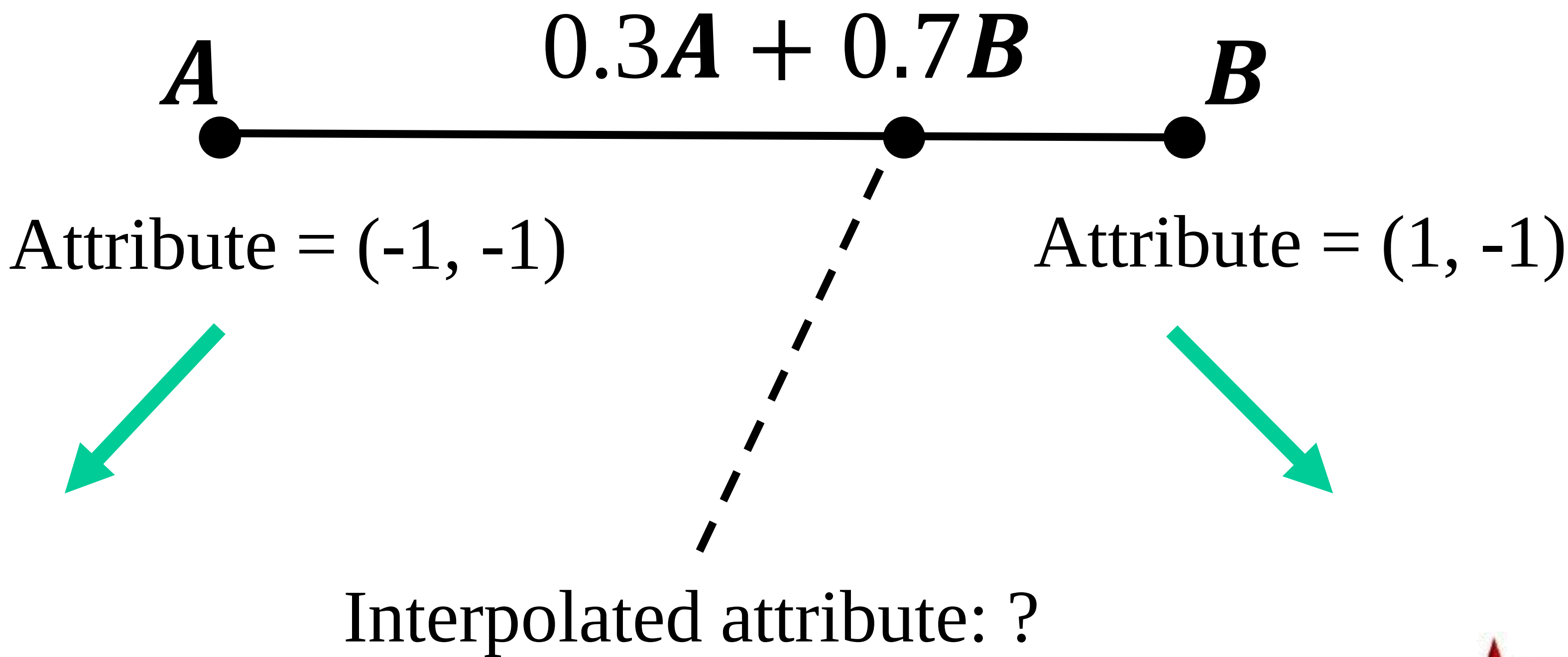


GLSL: `mix(10, 20, 0.3)`

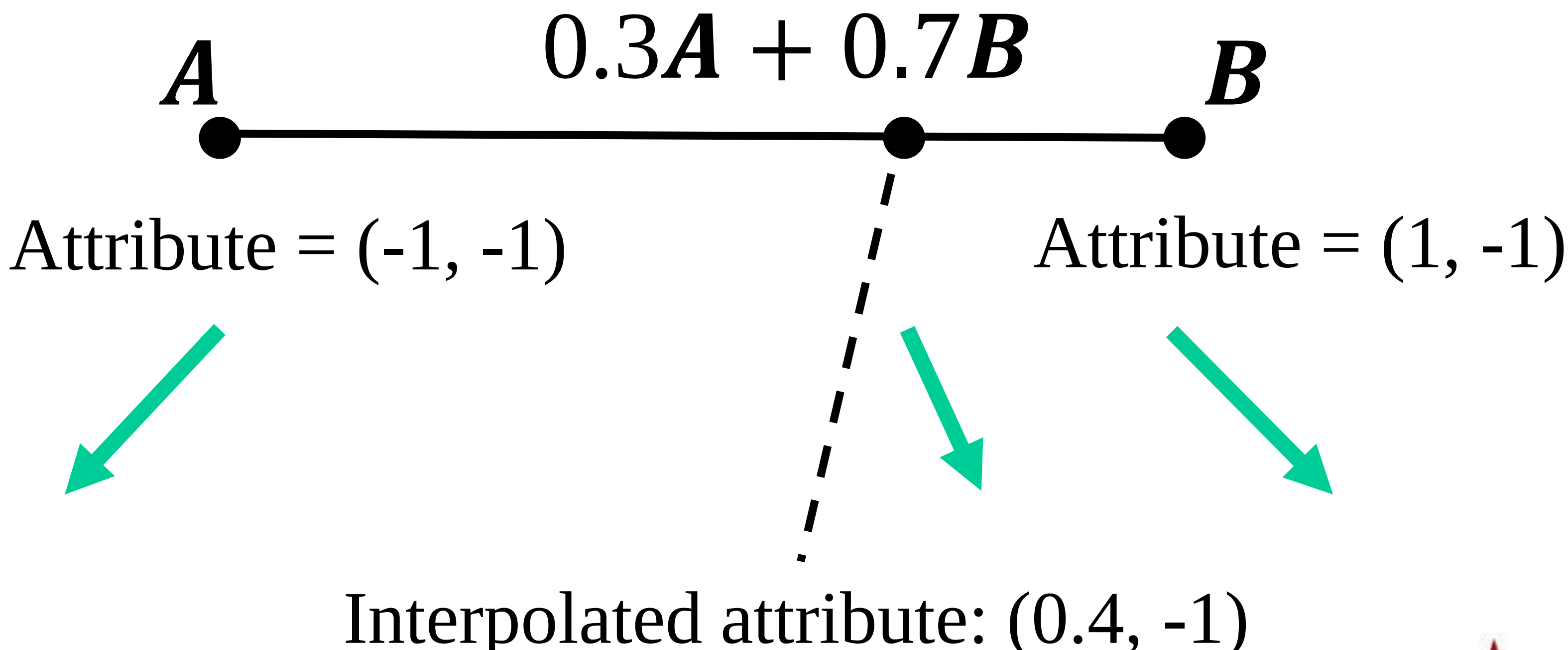
Linear interpolation



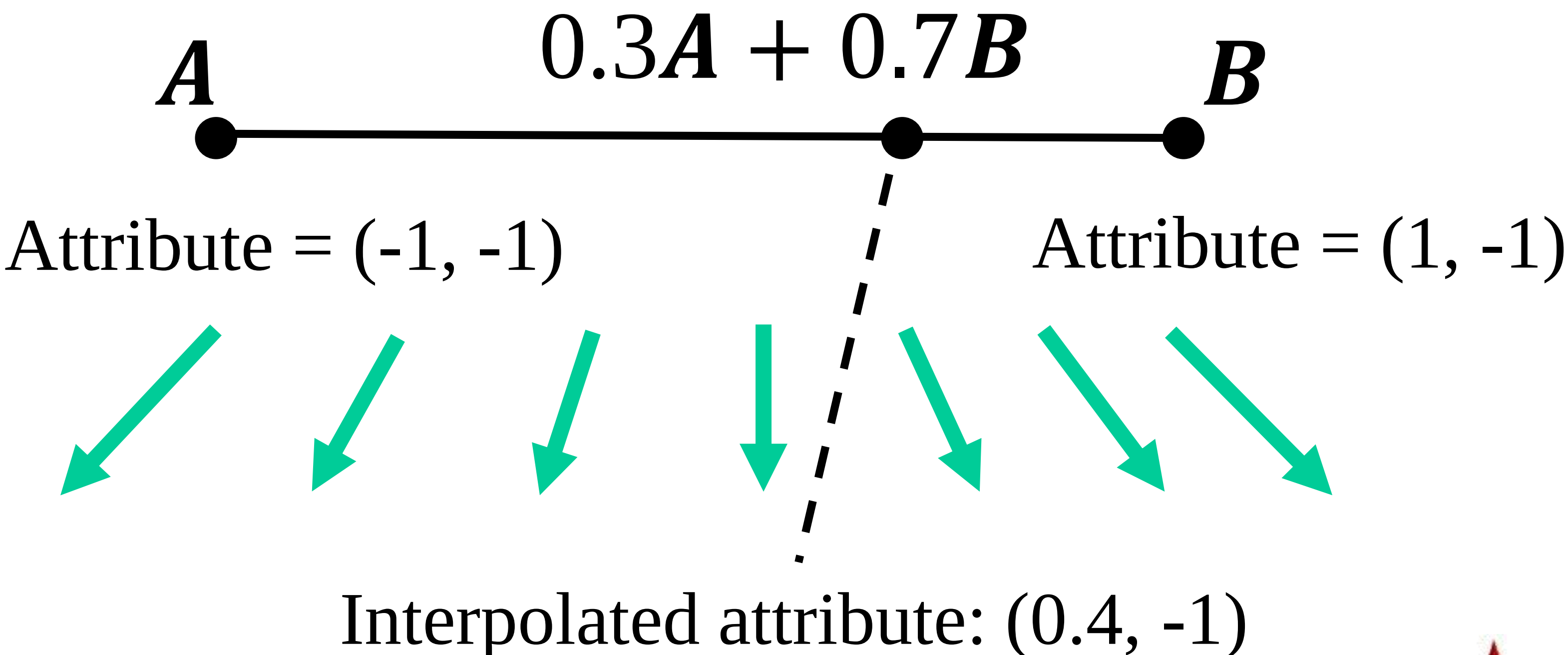
Linear interpolation



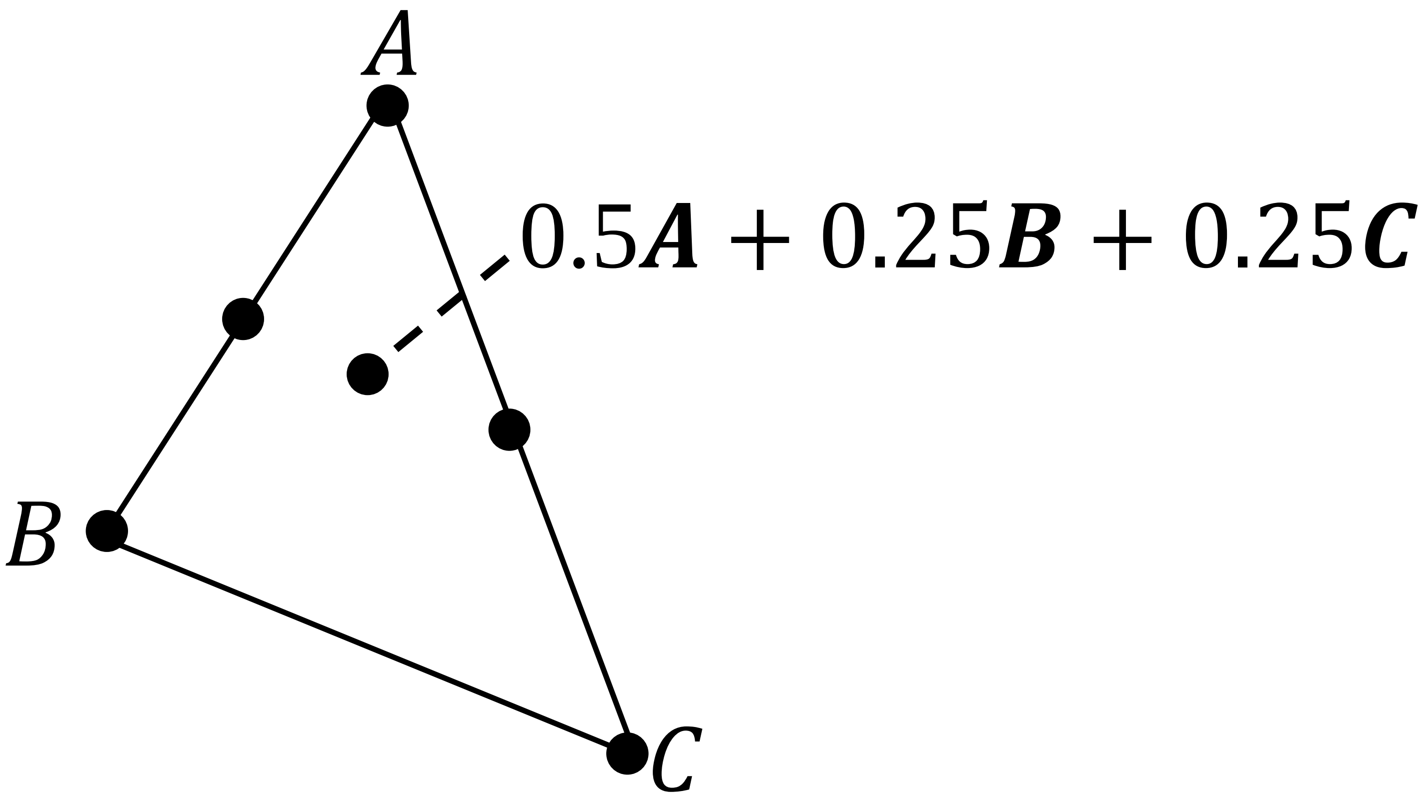
Linear interpolation



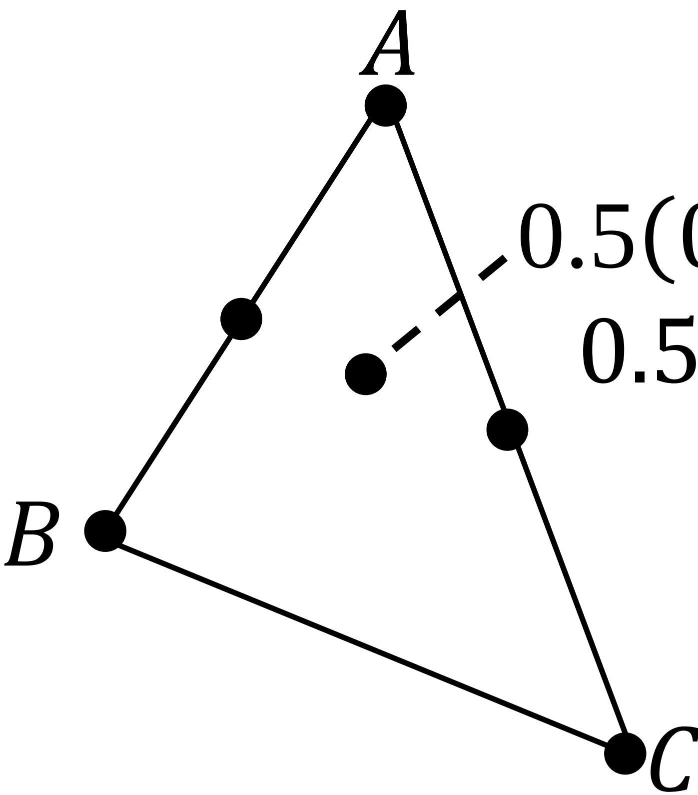
Linear interpolation



Linear interpolation



Linear interpolation

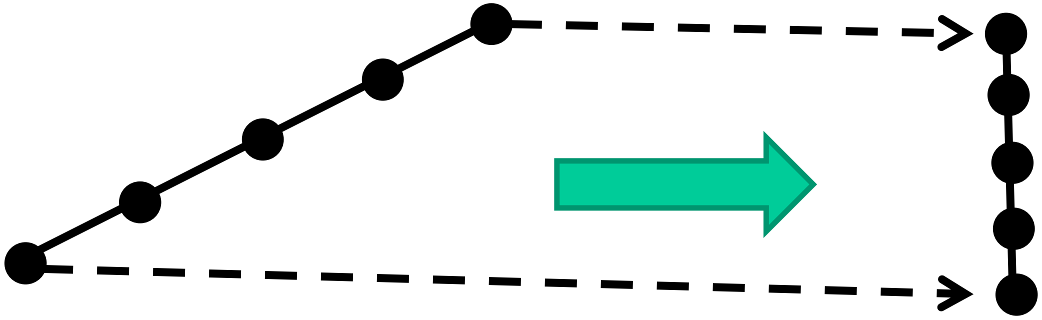


$$0.5(0.5\boldsymbol{A} + 0.5\boldsymbol{B}) + 0.5(0.5\boldsymbol{A} + 0.5\boldsymbol{C})$$

Linear interpolation

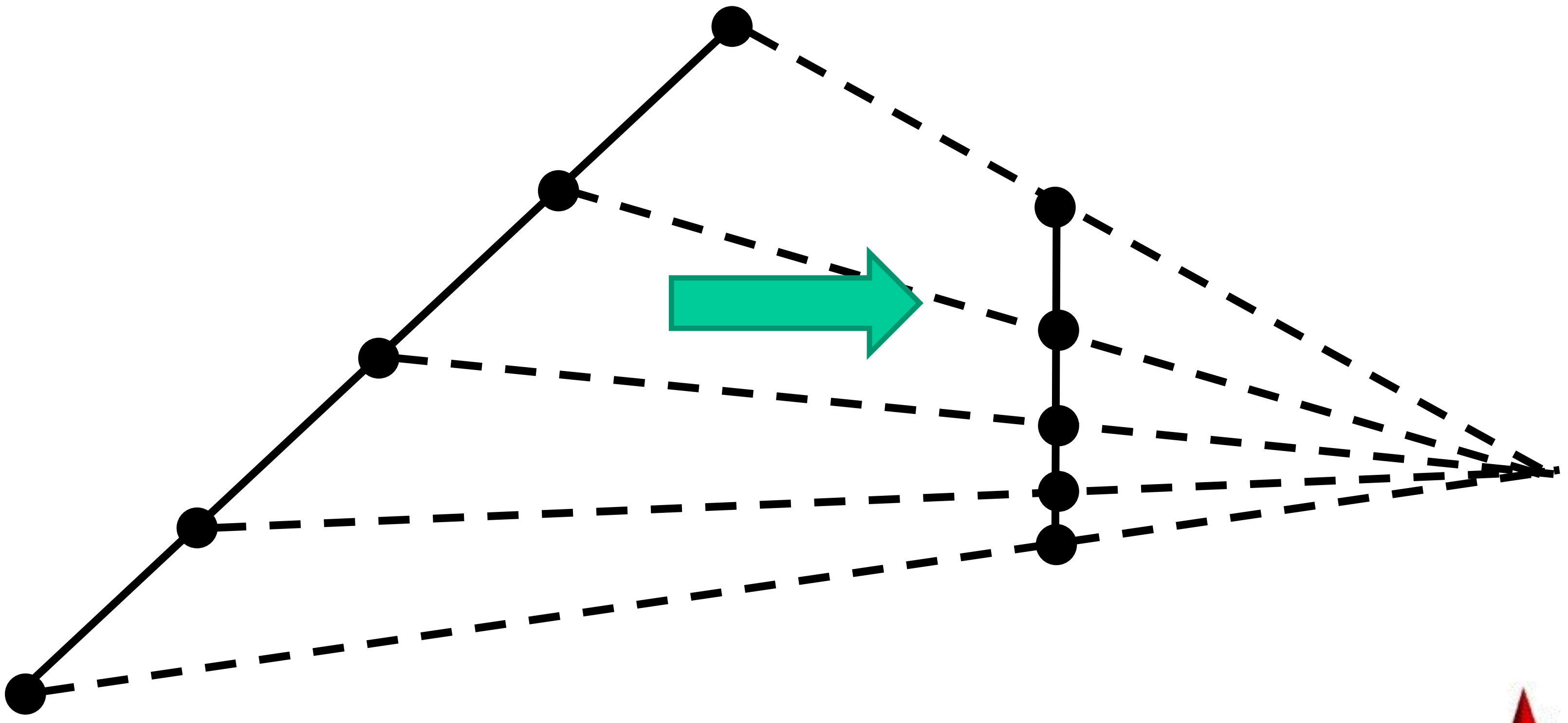
- Linear & affine transformations respect linear interpolation:

$$f(t\boldsymbol{A} + (1-t)\boldsymbol{B}) = tf(\boldsymbol{A}) + (1-t)f(\boldsymbol{B})$$



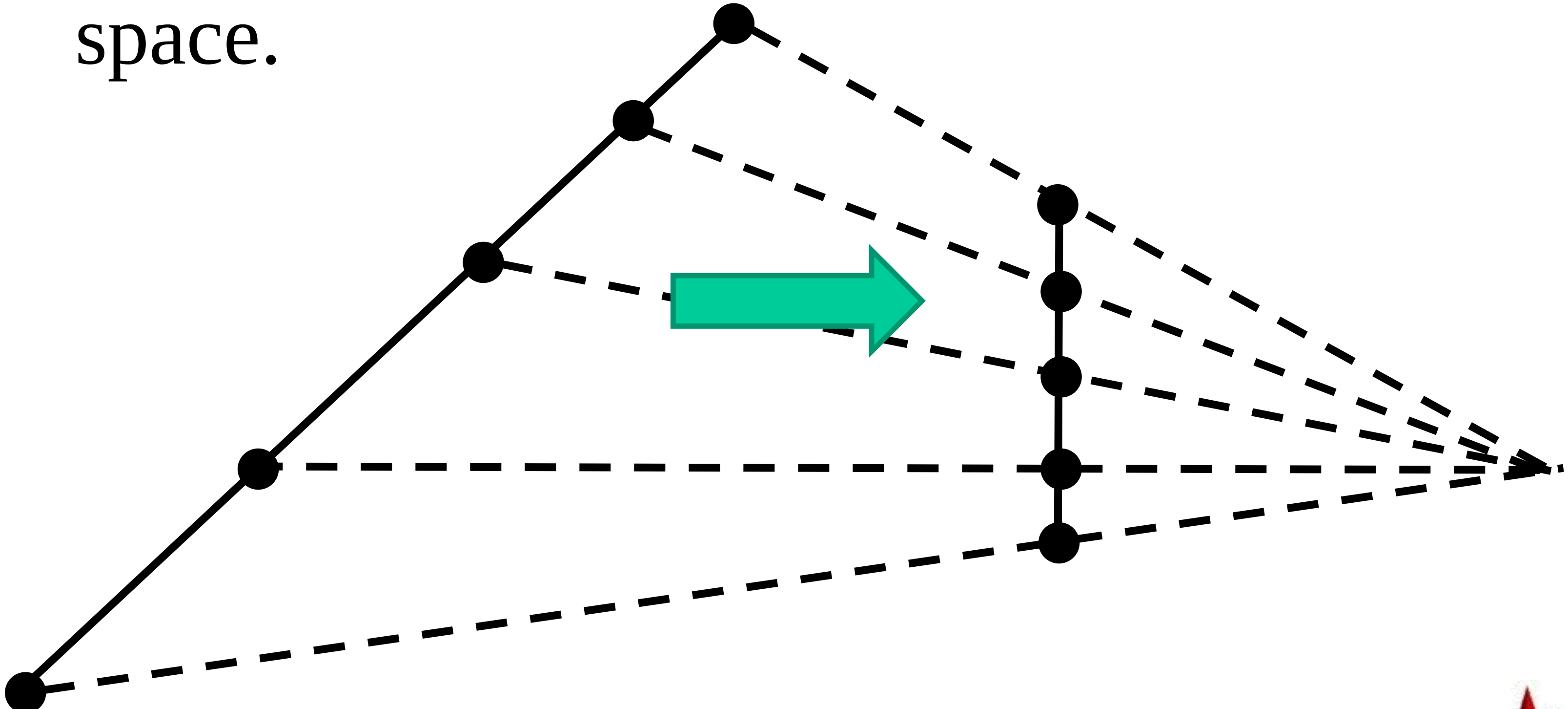
Linear interpolation

- Perspective projection does **not** respect linear interpolation:

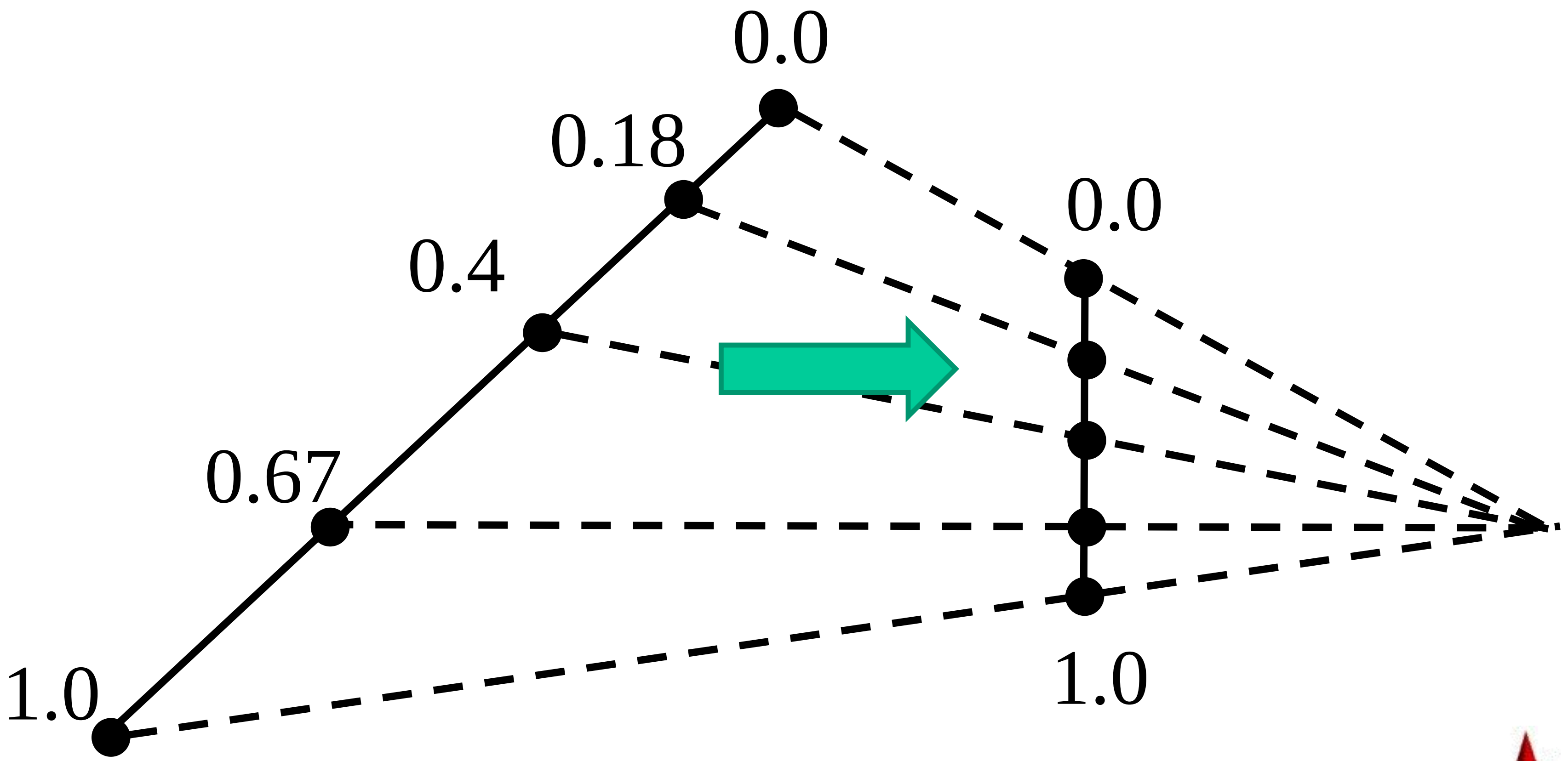


Linear interpolation

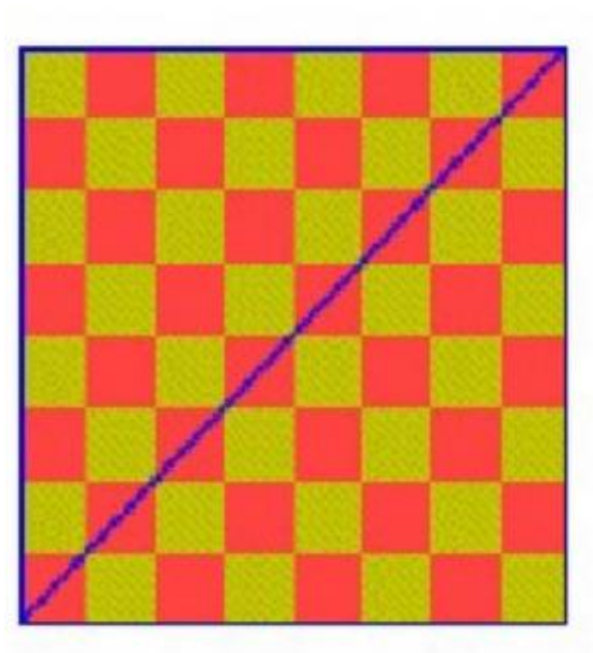
- Consequently, interpolating in window or clip-space **is not the same** as in world-space.

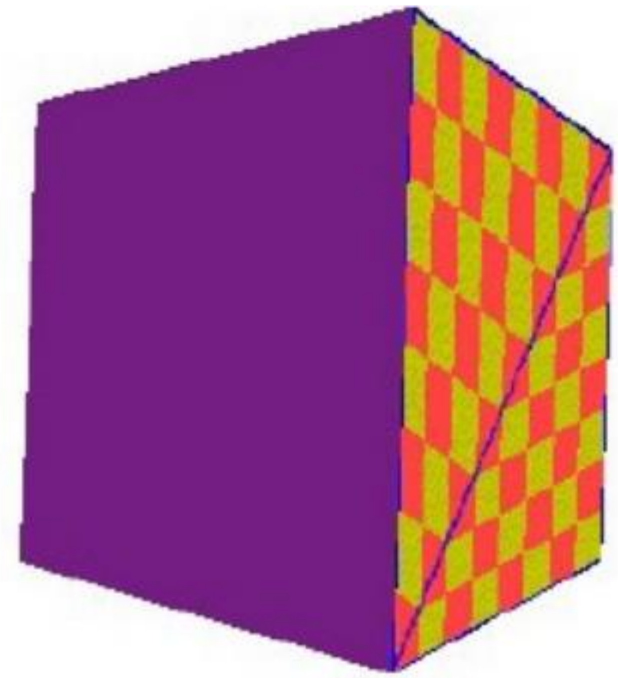


Linear interpolation



Window-space interpolation





Perspective correct interpolation

Let points $\boldsymbol{A}, \boldsymbol{B}$ be perspectively projected to window-coordinates $\boldsymbol{A}^*, \boldsymbol{B}^*$. Let

$$\boldsymbol{C}^* = t\boldsymbol{A}^* + (1-t)\boldsymbol{B}^*$$

Let attribute values F_A and F_B be associated with $\boldsymbol{A}$ and $\boldsymbol{B}$, varying linearly inbetween.

Then for the corresponding point $\boldsymbol{C}$:

$$\frac{F_C}{z_C} = t\,\frac{F_A}{z_A} + (1-t)\,\frac{F_B}{z_B}$$

Perspective correct interpolation

Then

$$\frac{F_C}{z_C} = t\,\frac{F_A}{z_A} + (1-t)\,\frac{F_B}{z_B}$$

In particular,

$$\frac{1}{z_C} = t\,\frac{1}{z_A} + (1-t)\,\frac{1}{z_B}$$

Perspective correct interpolation

Thus:

$$F_C = \frac{t\,\dfrac{F_A}{z_A} + (1-t)\,\dfrac{F_B}{z_B}}{t\,\dfrac{1}{z_A} + (1-t)\,\dfrac{1}{z_B}}$$

Perspective correct interpolation

If we keep the homogeneous w term before the perspective division, we can do:

$$F_C = \frac{t\,\frac{F_A}{w_A} + (1-t)\,\frac{F_B}{w_B}}{t\,\frac{1}{w_A} + (1-t)\,\frac{1}{w_B}}$$

(this makes interpolation logic independent of what projection matrix you set up).

For proofs see P. Heckbert & H. Moreton,
"Interpolation for Polygon Texture Mapping and Shading"

Interpolation in GLSL

- Vertex shaders output attributes, that are interpolated over the triangle and sent to the fragment shader.
- You can specify the kind of interpolation:

```
flat          out vec3 not_interpolated;
smooth        out vec3 perspective_correct;
noperspective out vec3 affine_interpolation;
```

(in the first case, just one of the three vertex attribute values is chosen for all fragments of the triangle)

Texturing

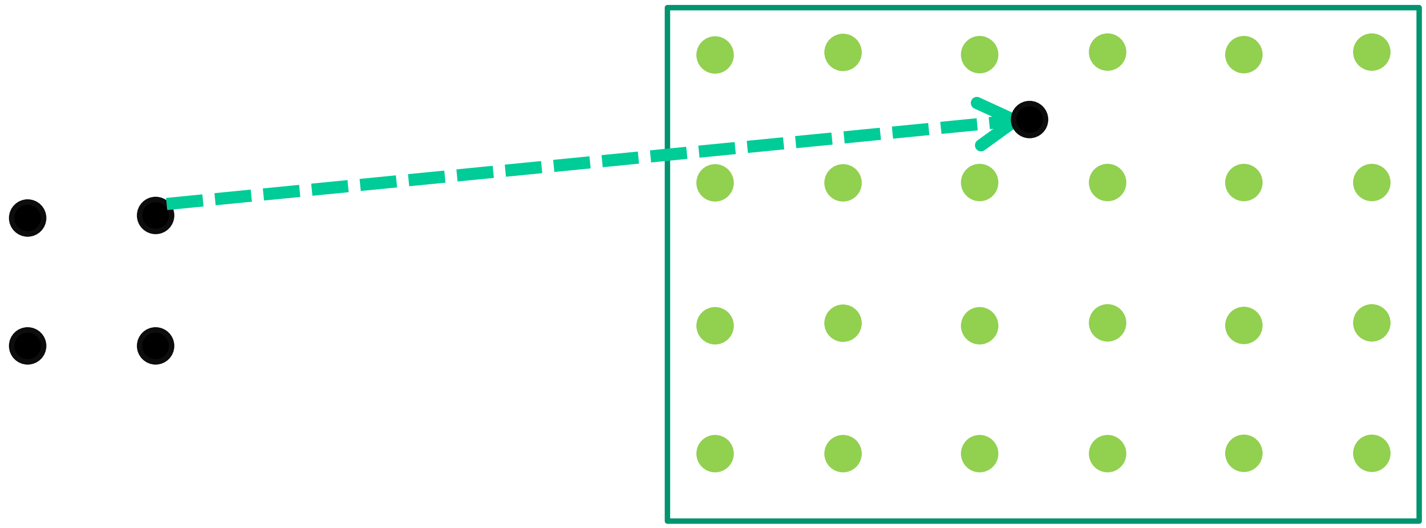
- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

Texture filtering

Consider how a pixel on the screen maps to a texture image

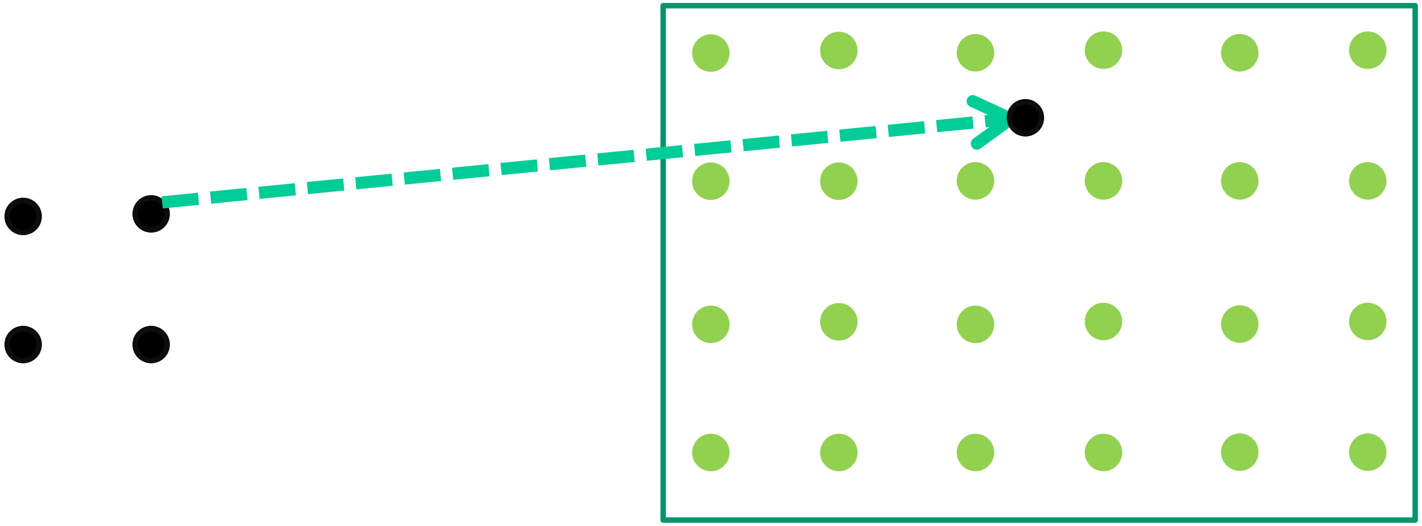


Screen pixels

Texture pixels

Texture filtering

What texel (texture pixel) should we use for this screen pixel?

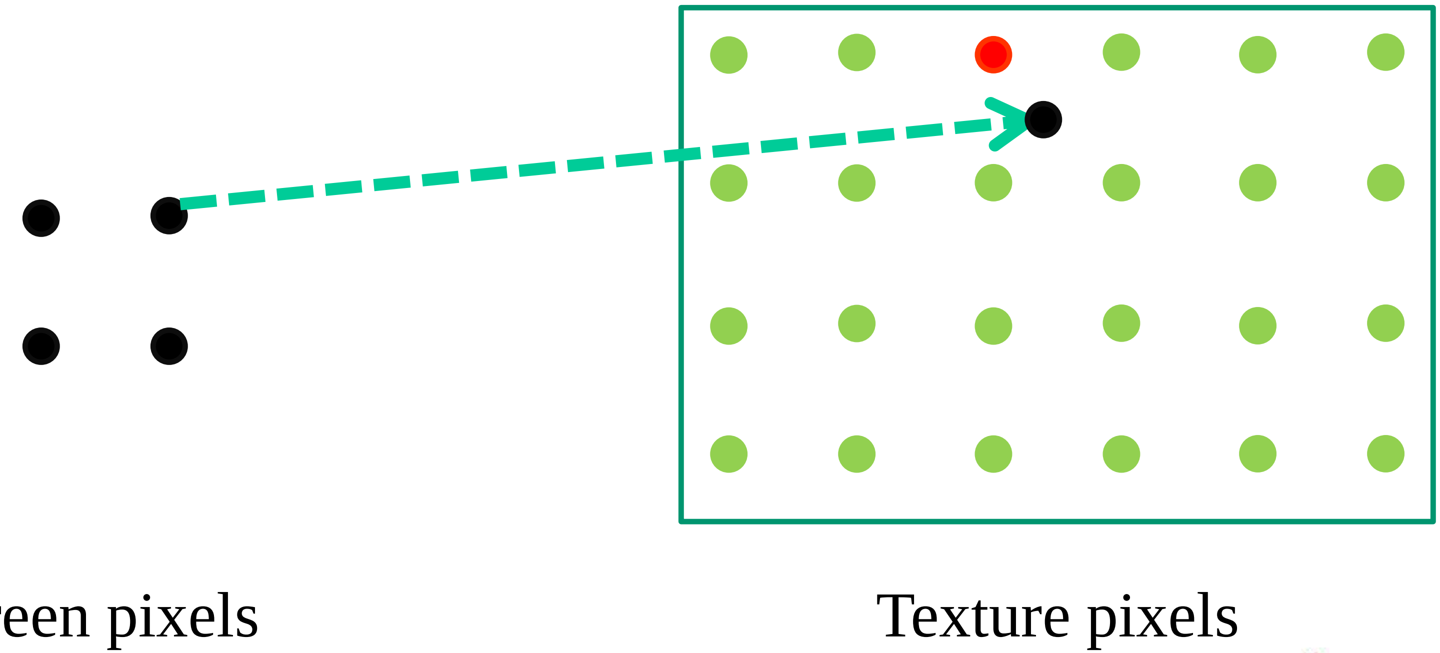


Screen pixels

Texture pixels

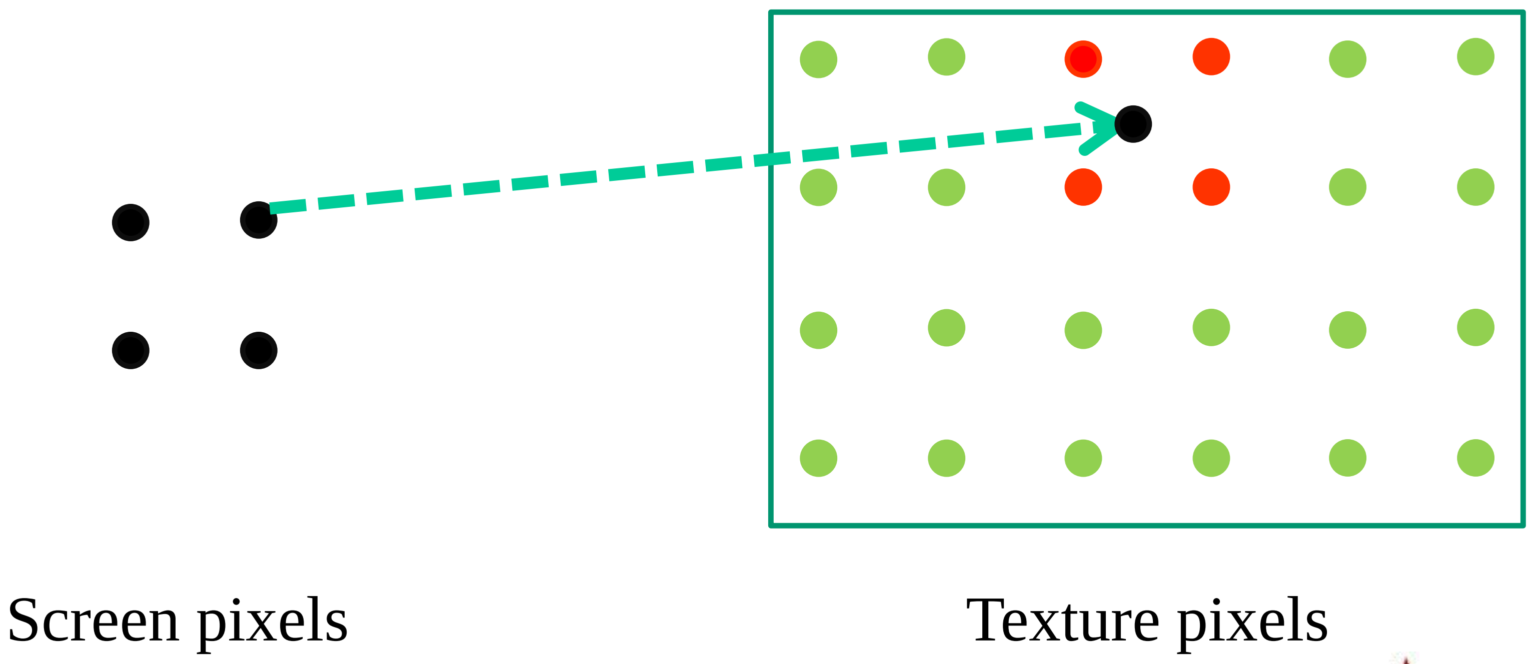
Texture filtering

1. Pick the nearest one



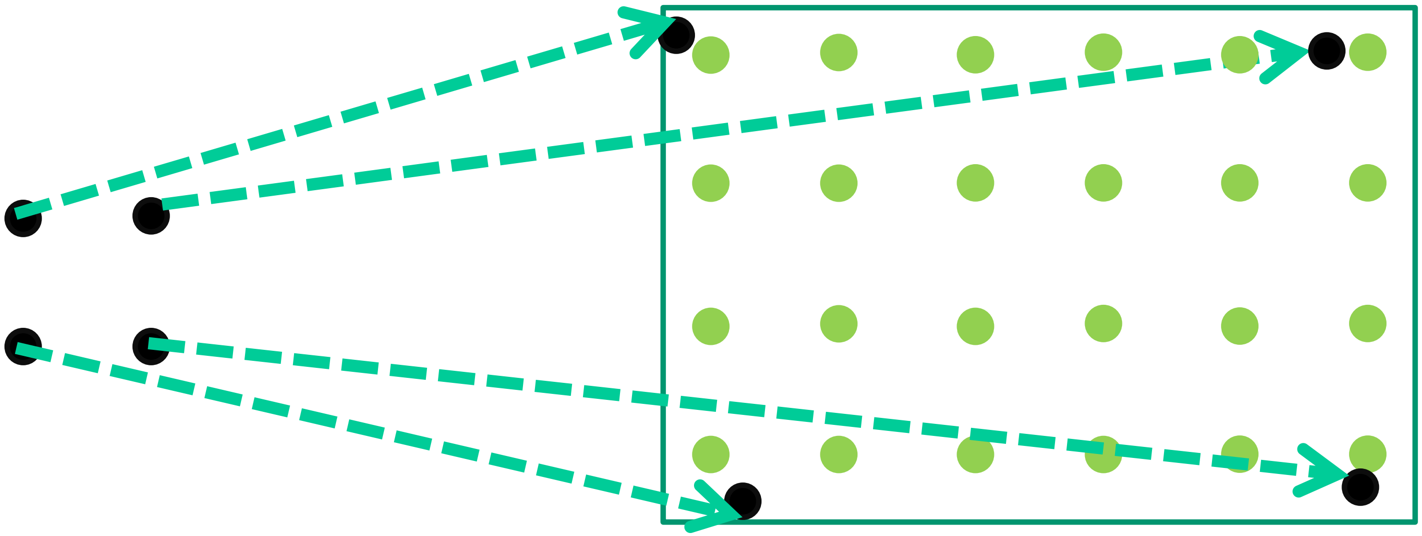
Texture filtering

2. Bilinearly interpolate (i.e. average) four surrounding pixels.



Texture filtering

Consider how four nearby pixels map to the texture:

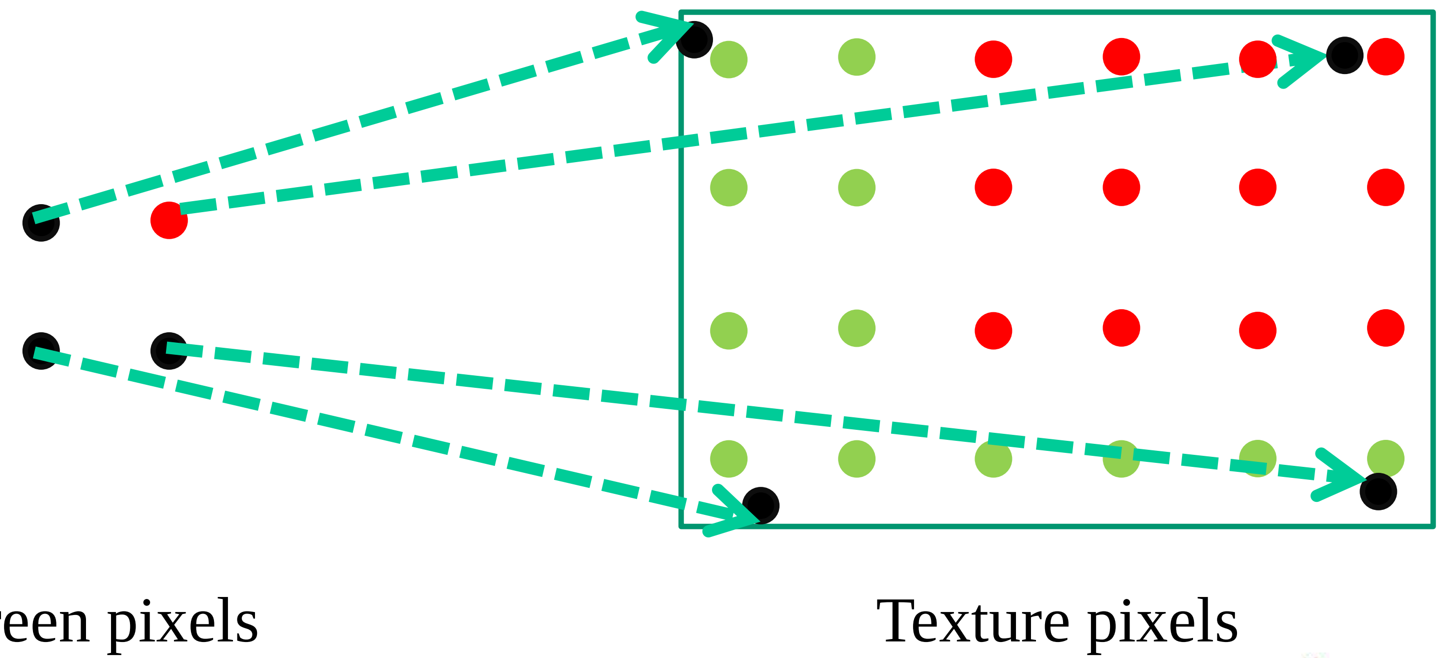


Screen pixels

Texture pixels

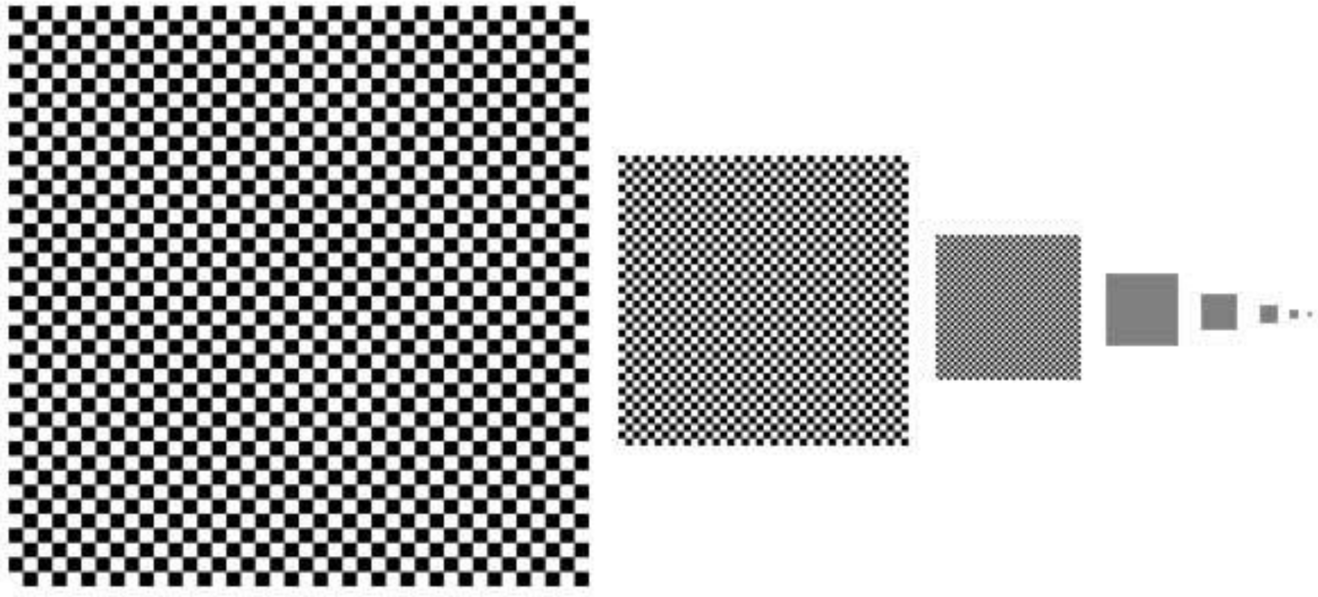
Texture filtering

Now we would like to average over a much larger piece of texture



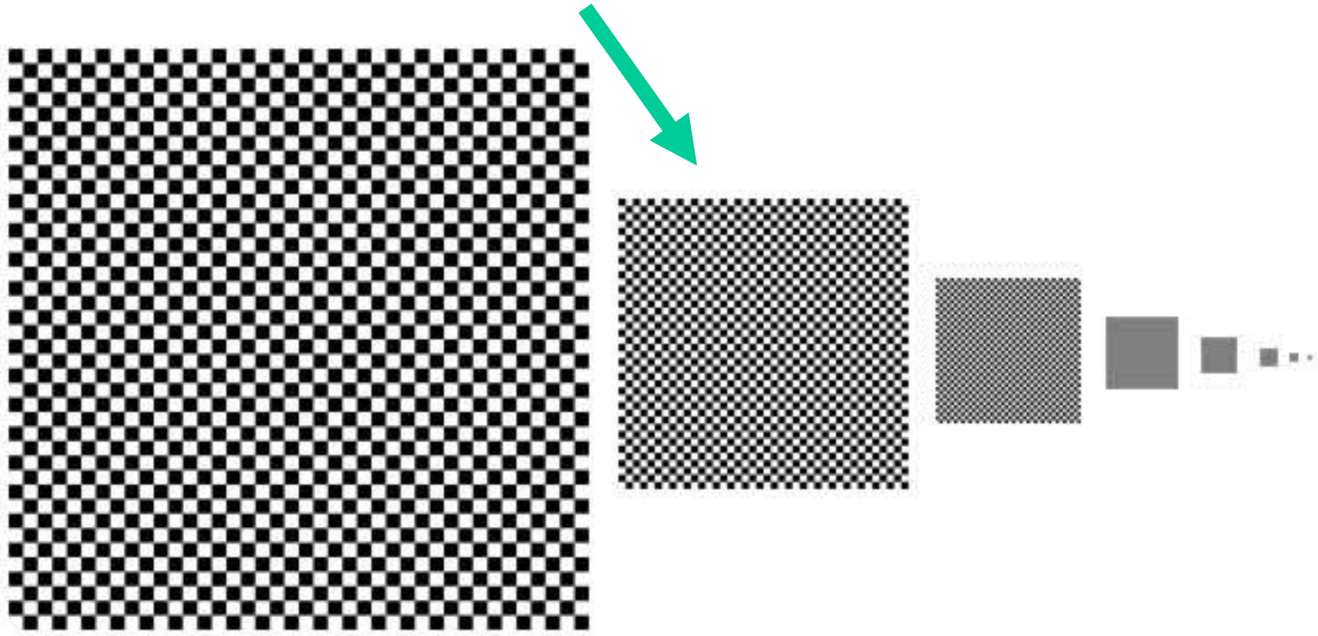
Mipmaps

- To average over large regions of the texture, we pre-average our original texture by computing a set of downscaled versions:



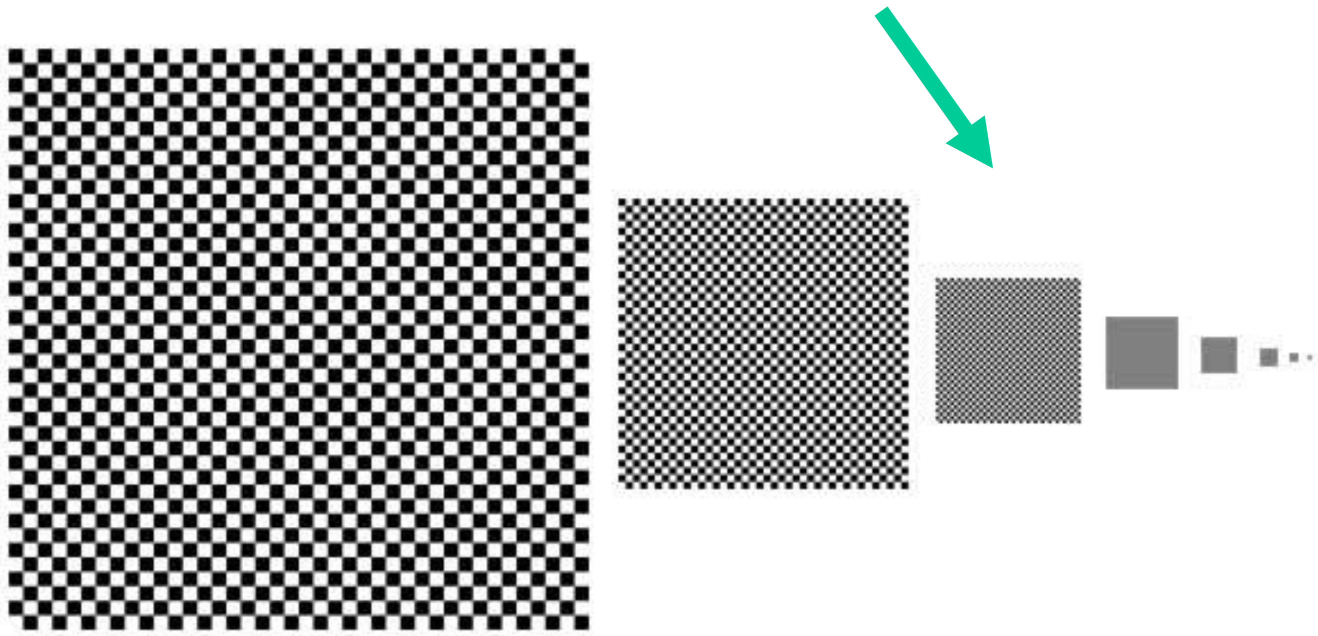
Mipmaps

Level 1 mipmap is downscaled 2x, so its each pixel is an average of 4 original pixels



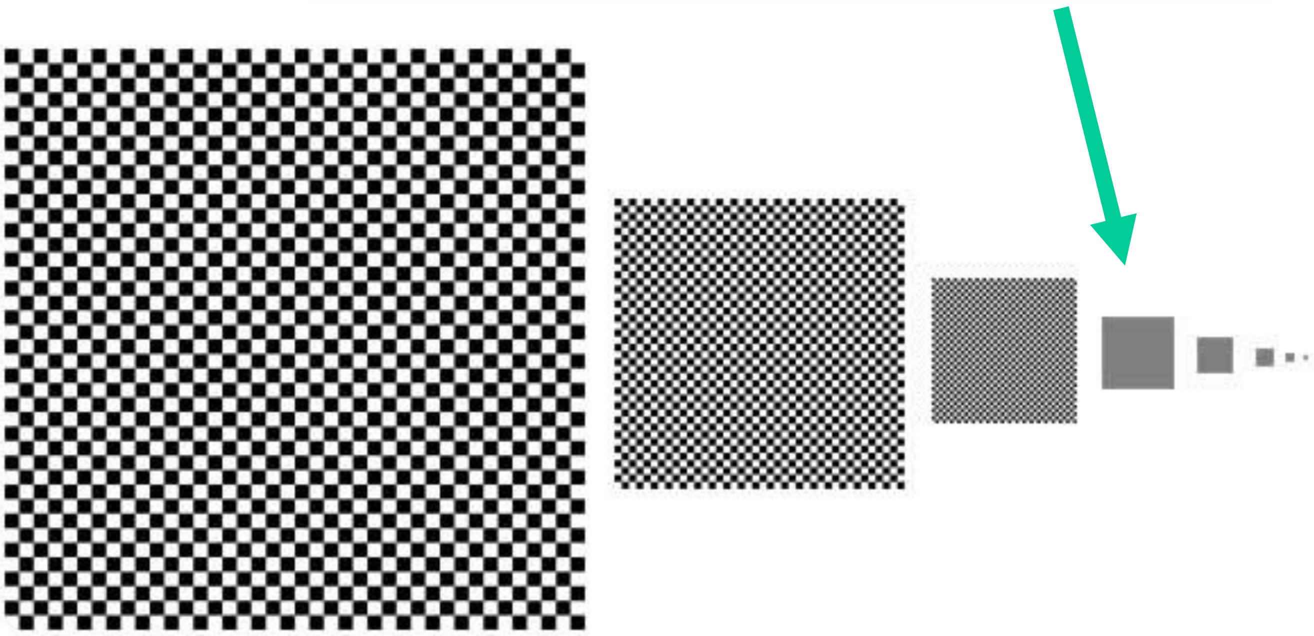
Mipmaps

Level 2 mipmap is downscaled 4x, so its each pixel is an average of 16 original pixels



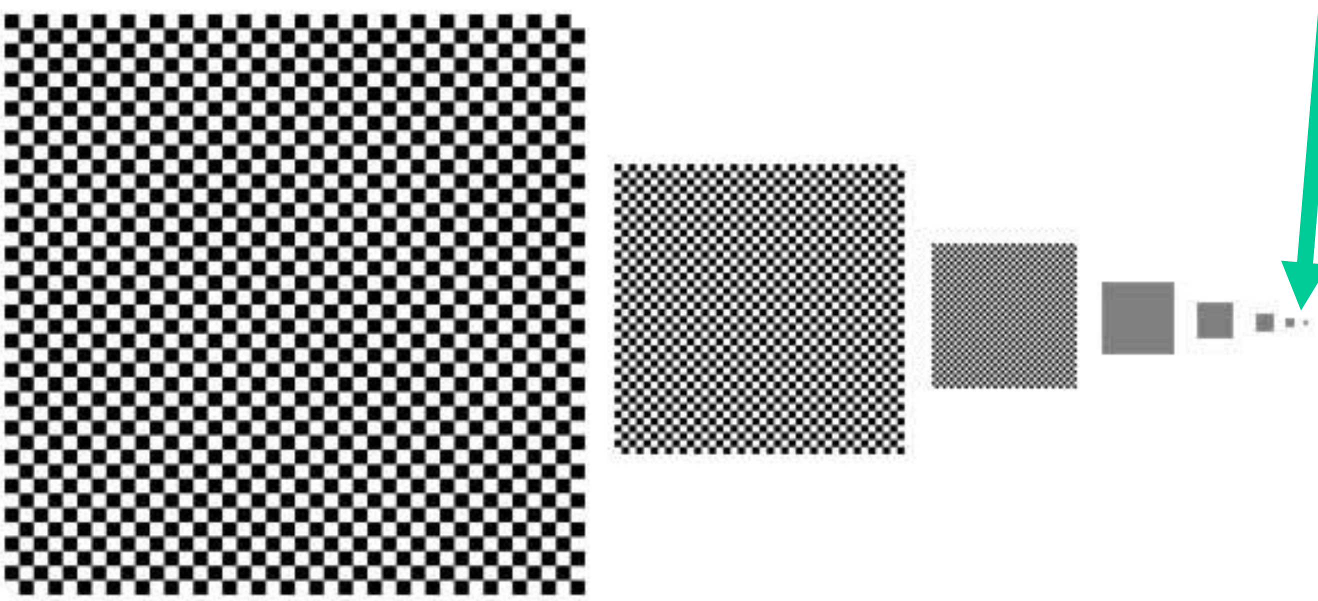
Mipmaps

Level 3 mipmap is downscaled 8x, so its each pixel is an average of 64 original pixels



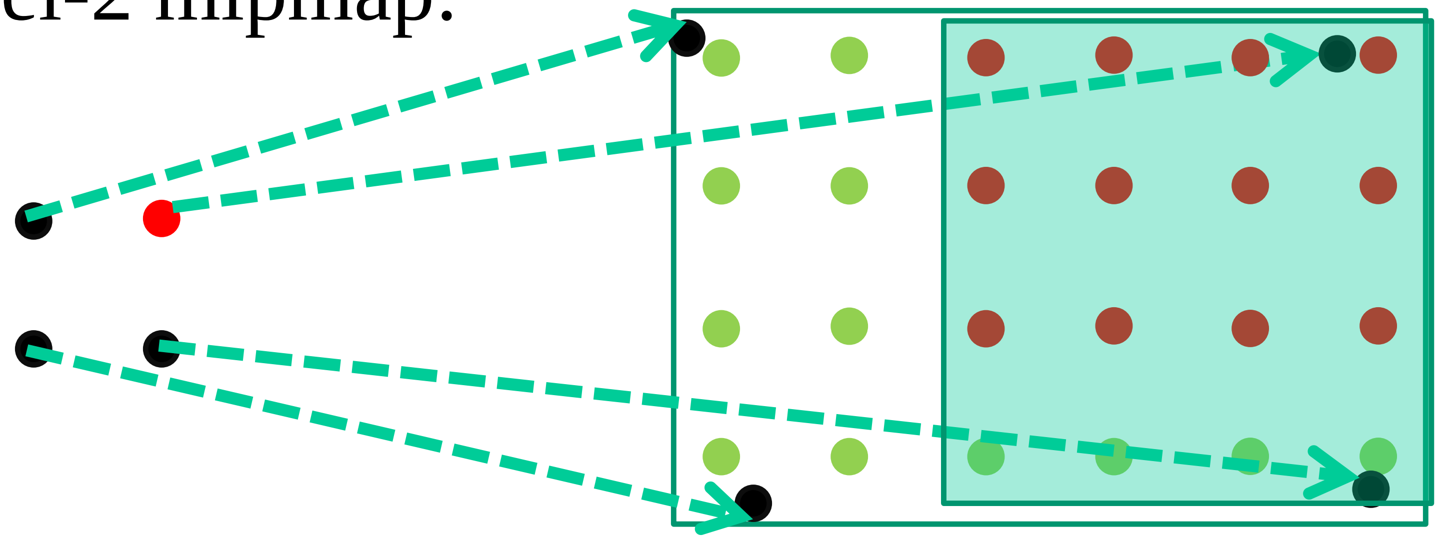
Mipmaps

… etc all the way down to a single-pixel image, which is the overall average of the original image



Texture filtering

Using mipmaps we can efficiently average, e.g the highlighted piece could be looked up in a level-2 mipmap:

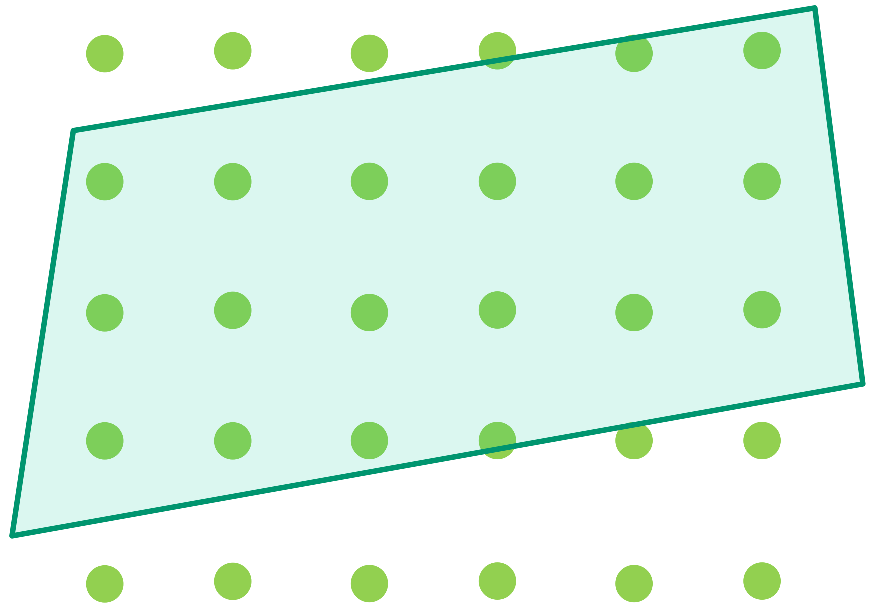


Screen pixels

Texture pixels

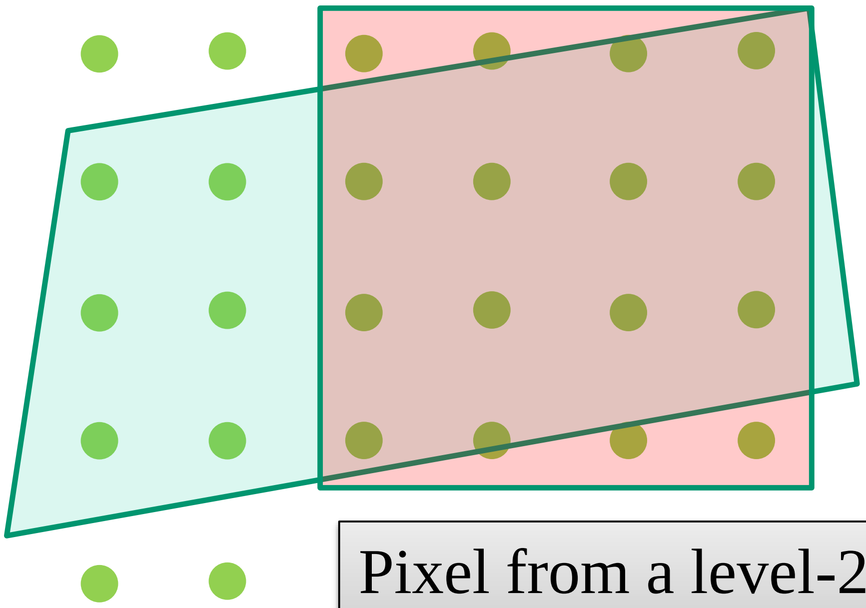
Mipmap-based filtering

- Figure out what region the screen pixel must cover in the texture
- Look up the appropriate average from the mipmaps



Mipmap-based filtering

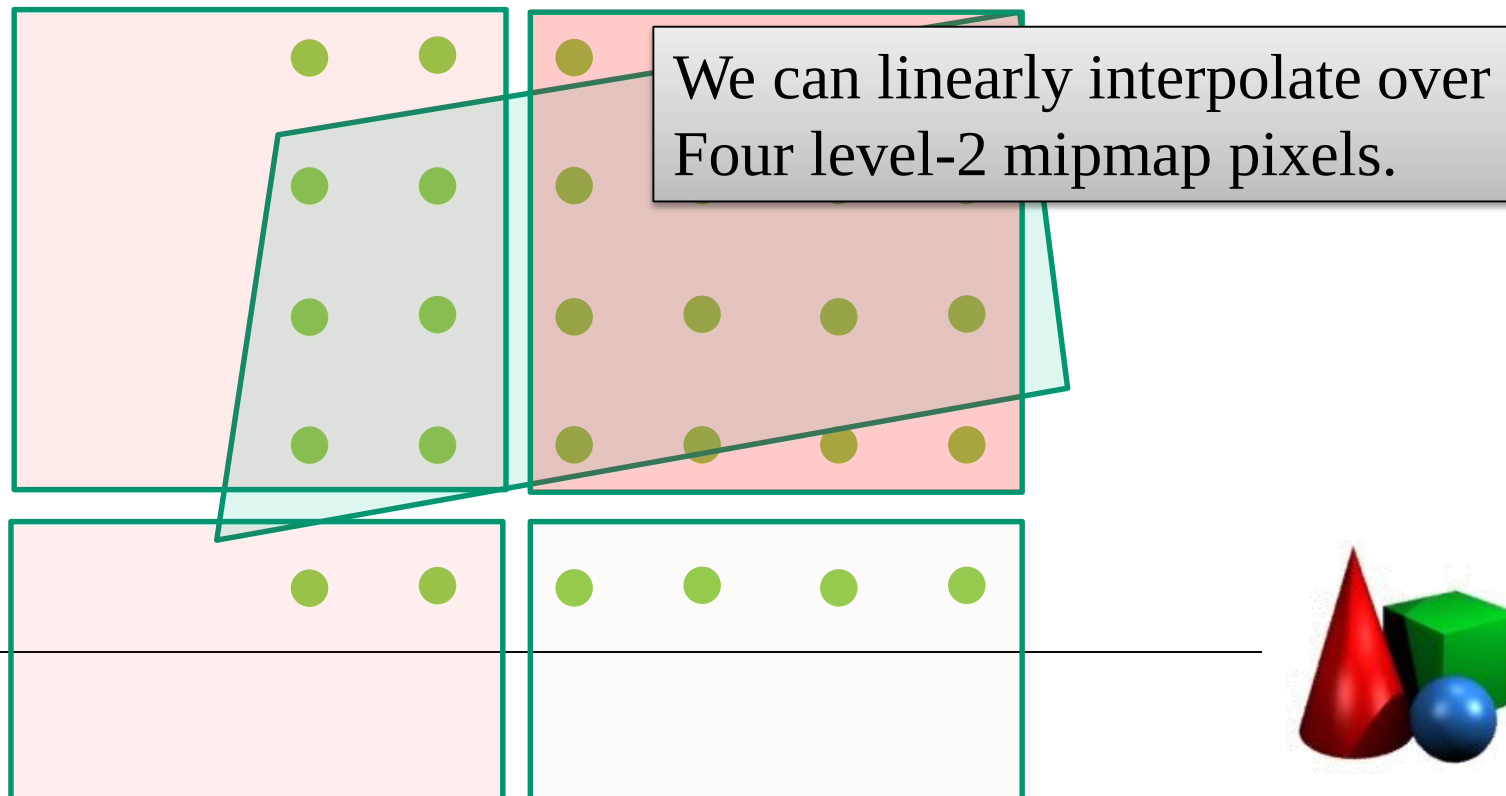
Option 1: Find the scale-wise closest mipmap, and pick a single pixel from it.



Pixel from a level-2 mipmap covers this average.

Mipmap-based filtering

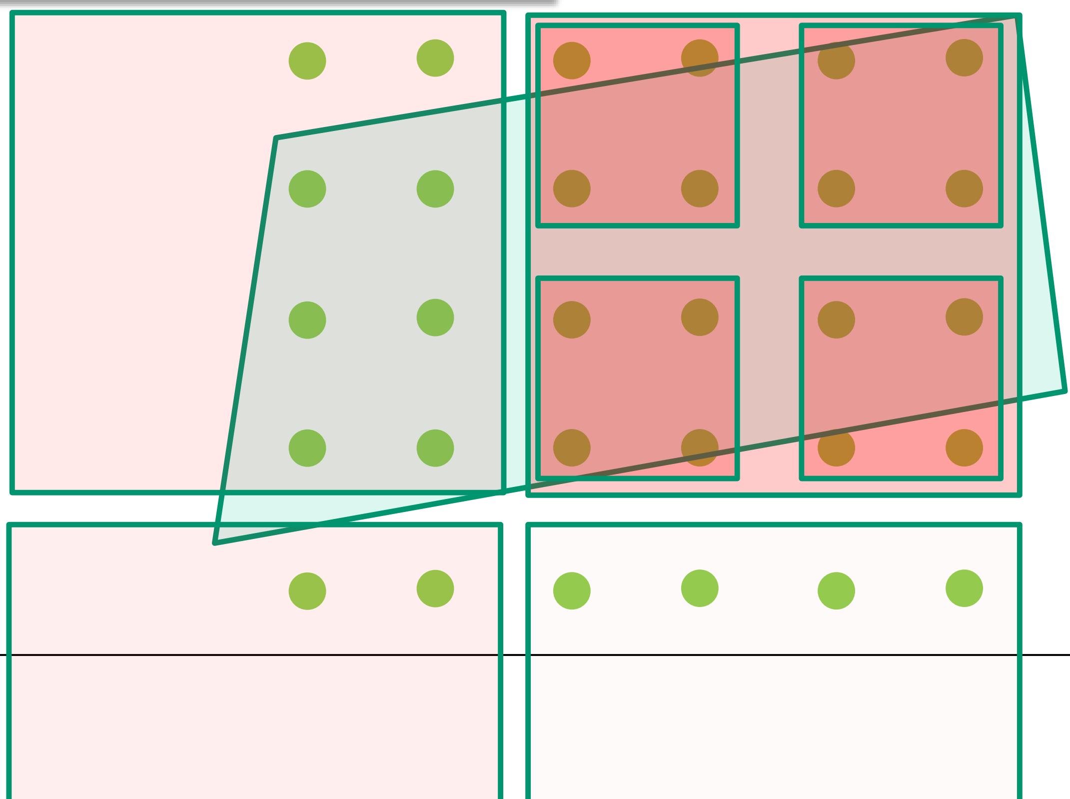
Option 2: Find the scale-wise closest mipmap, pick several pixels from it and interpolate bilinearly as if it was original image



Mipmap-based filtering

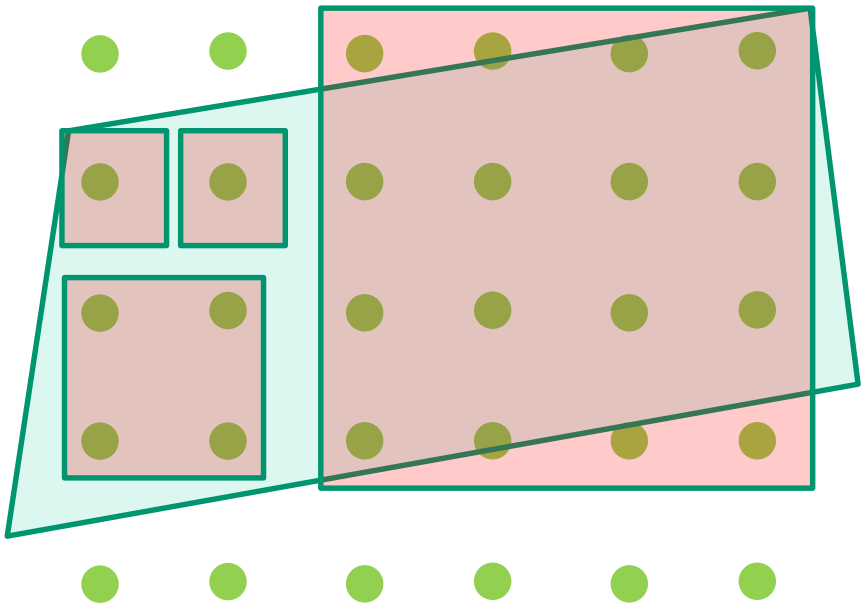
Option 3: Use two different mipmap levels

We can use results from level 1 and level 2-mipmaps, giving perhaps more weight to level 1



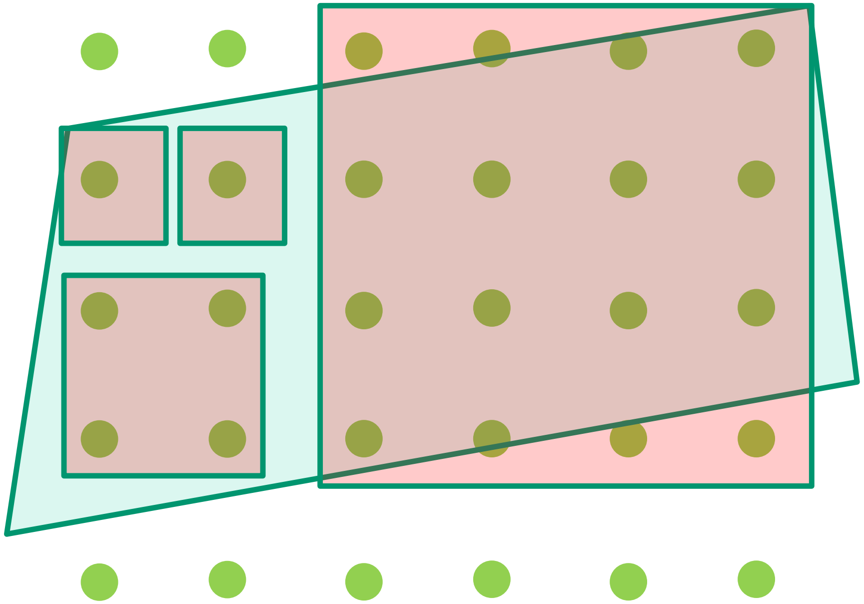
Mipmap-based filtering

Option 4: Be very smart and use many mipmap levels to compute the best approximation to the required average:



Mipmap-based filtering

This approach allows averaging over non-square areas, and is thus referred to as *anisotropic filtering*.



Texture filtering

- `GL_NEAREST`
- `GL_LINEAR`
- `GL_LINEAR_MIPMAP_NEAREST`
- `GL_LINEAR_MIPMAP_LINEAR`
- `GL_TEXTURE_MAX_ANISOTROPY_EXT = (number)`

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

Textures & OpenGL

Loading texture image and mipmaps

```
GLuint textureHandle;
glGenTextures(1, &textureHandle);
glBindTexture(GL_TEXTURE_2D, textureHandle);

glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB8, w, h,
          0, GL_RGB, GL_UNSIGNED_BYTE, image);
glTexImage2D(GL_TEXTURE_2D, 1, GL_SRGB8, w/2,h/2,
          0, GL_RGB, GL_UNSIGNED_BYTE, mipmap1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_SRGB8, w/4,h/4,
          0, GL_RGB, GL_UNSIGNED_BYTE, mipmap2);
...
```

Textures & OpenGL

```
GLuint textureHandle;
glGenTextures(1, &textureHandle);
glBindTexture(GL_TEXTURE_2D, textureHandle);

gluBuild2DMipmaps(GL_TEXTURE_2D, GL_SRGB8, w, h,
          GL_RGB, GL_UNSIGNED_BYTE, image);
```

Loading texture image and mipmaps

Textures & OpenGL

Specifying filtering method

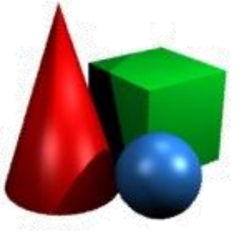
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                          GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                          GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,
             GL_TEXTURE_MAX_ANISOTROPY_EXT, 16.0f);
```

Textures & OpenGL

Using the texture

```
glEnable(GL_TEXTURE2D);
glBindTexture(GL_TEXTURE2D, textureHandle);

glBegin(GL_TRIANGLES);
        glTexCoord2f(0.0, 0.0);
        glVertex3f(0.0, 0.0, 0.0);
        glTexCoord2f(0.0, 1.0);
        glVertex3f(1.0, 0.0, 0.0);
        glTexCoord2f(1.0, 1.0);
        glVertex3f(1.0, 1.0, 0.0);
glEnd();
```



Textures & OpenGL

Automatically generating coordinates

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);

// or

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

Textures & OpenGL

How to interpret texture coordinates beyond [0, 1]?

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                                GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                                GL_REPEAT);
```

Textures & OpenGL

How to combine texture with lighting?

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                                        GL_MODULATE);


glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                                        GL_REPLACE);


glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                                        GL_ADD);
```

Textures & GLSL

Texture in the shader is accessed via a "sampler" object

```
uniform sampler2D myTexture;

void main() {
   gl_FragColor = texture2D(myTexture, gl_TexCoord[0]);
}
```

Textures & GLSL

```
uniform sampler2D myTexture;

void main() {
   gl_FragColor = texture2D(myTexture, gl_TexCoord[0]);
}
```

Of course, it is possible to define multiple sampler objects, each with its own parameters.

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures

Next week

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, reflection maps, shadow maps
- Procedural textures