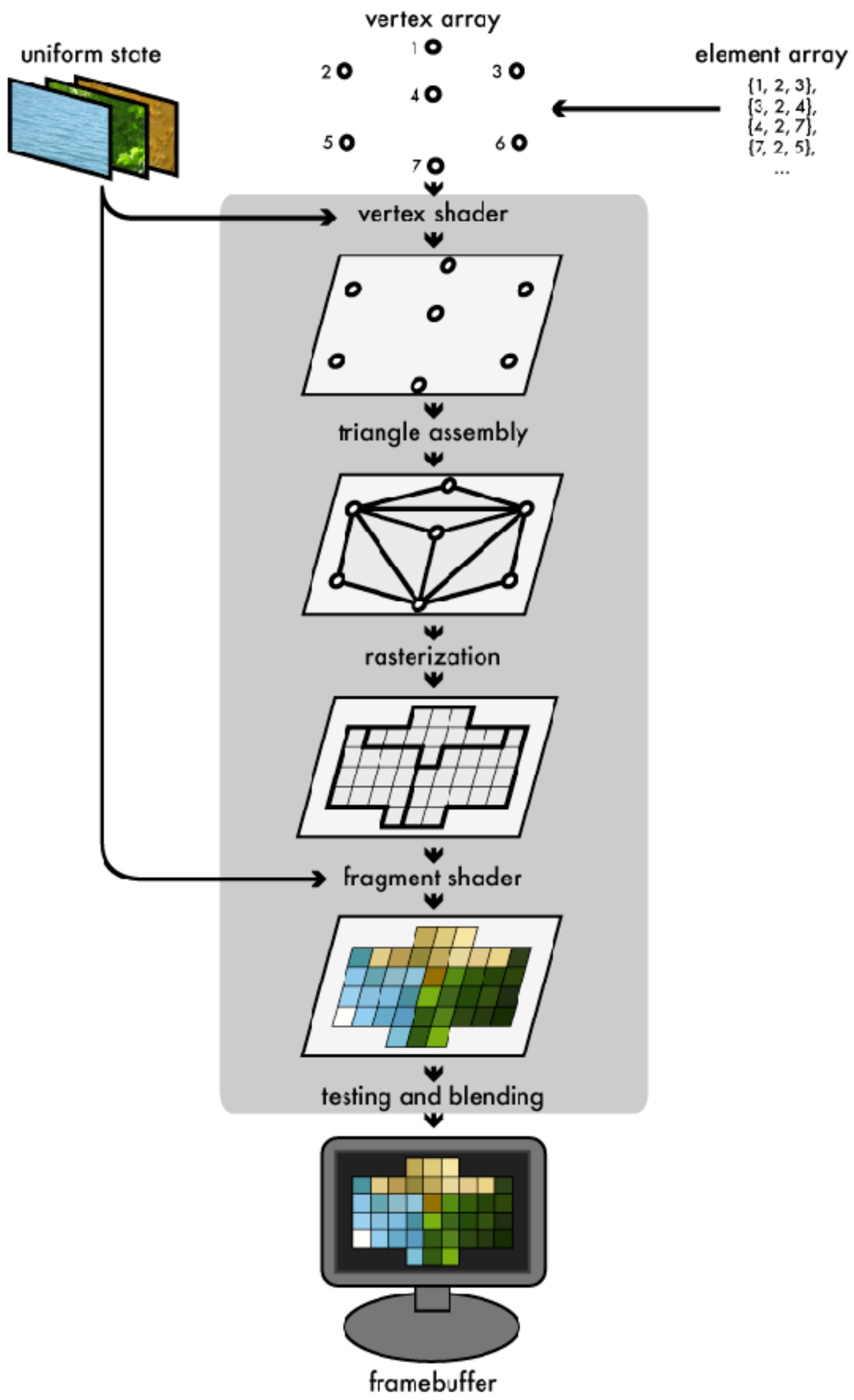
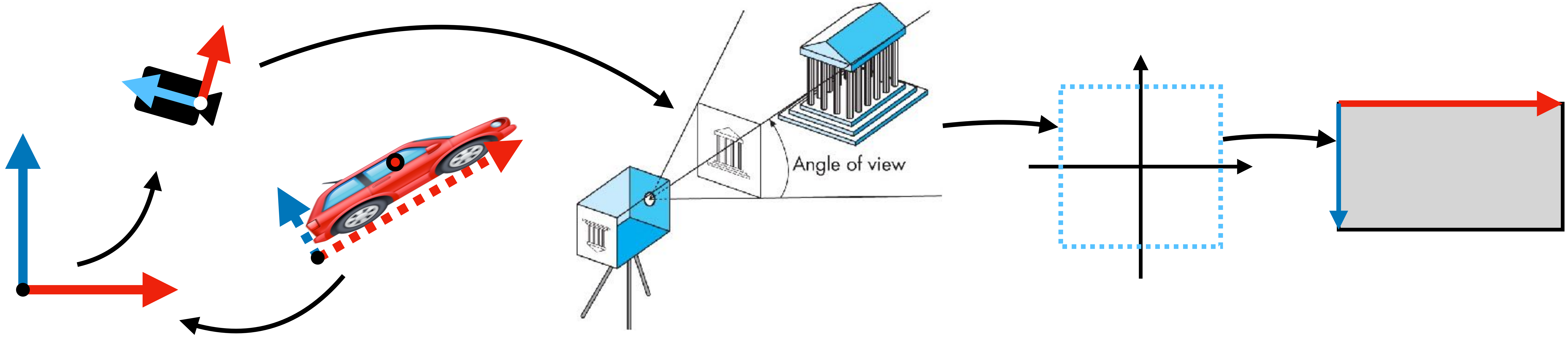
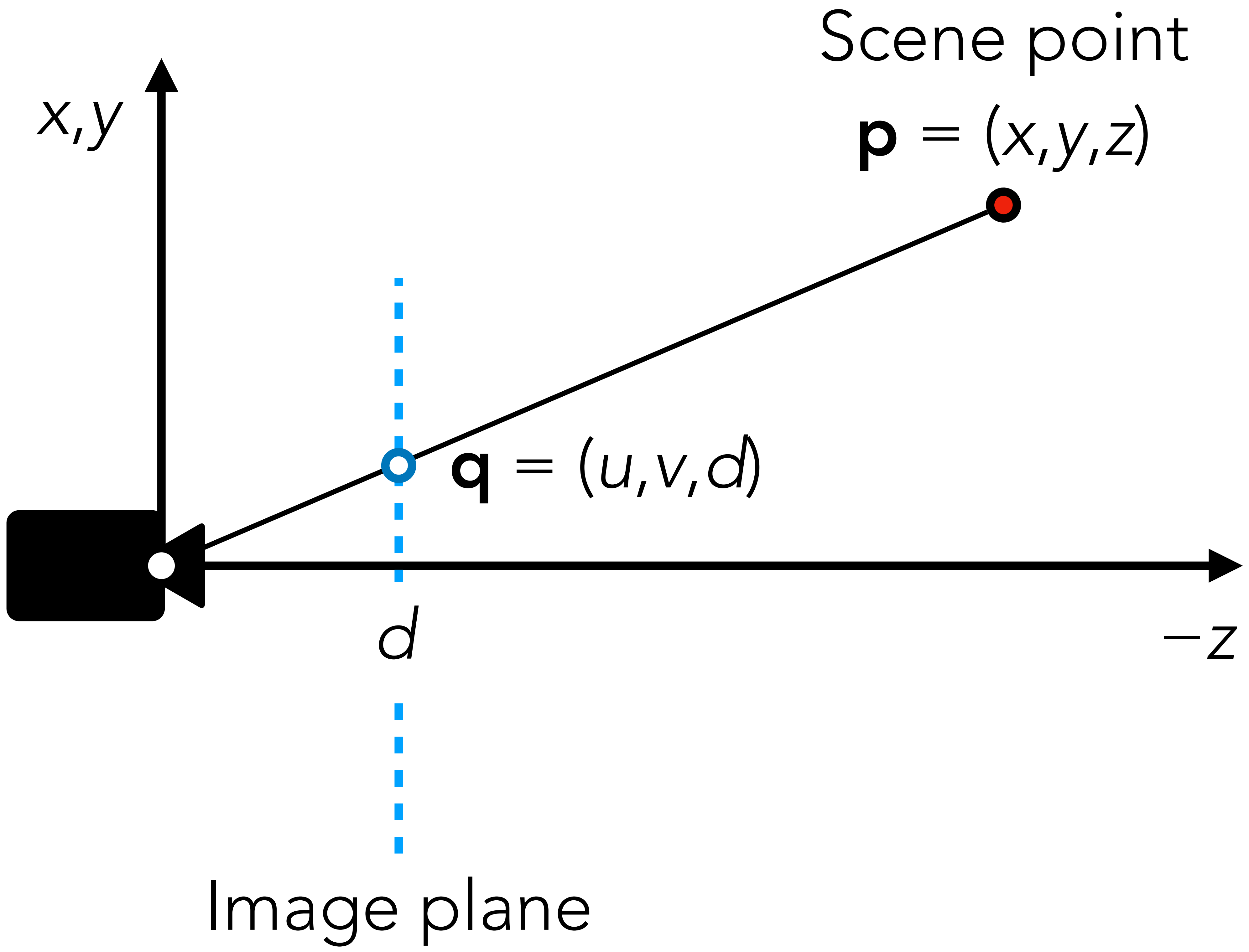

GRAPHICS & VISION
SUMMER SCHOOL
Computer
Graphics
Henrik Wann Jensen



vertex array
1
2
3
4
5
6
7
uniform state
element array
{1, 2, 3},
{3, 2, 4},
{4, 2, 7},
{7, 2, 5},
...
vertex shader
triangle assembly
rasterization
fragment shader
testing and blending
framebuffer



- Object space → world space
- World space → camera space
- Camera space → projection plane (division by z)
- Projection plane → NDC
- NDC → screen coordinates

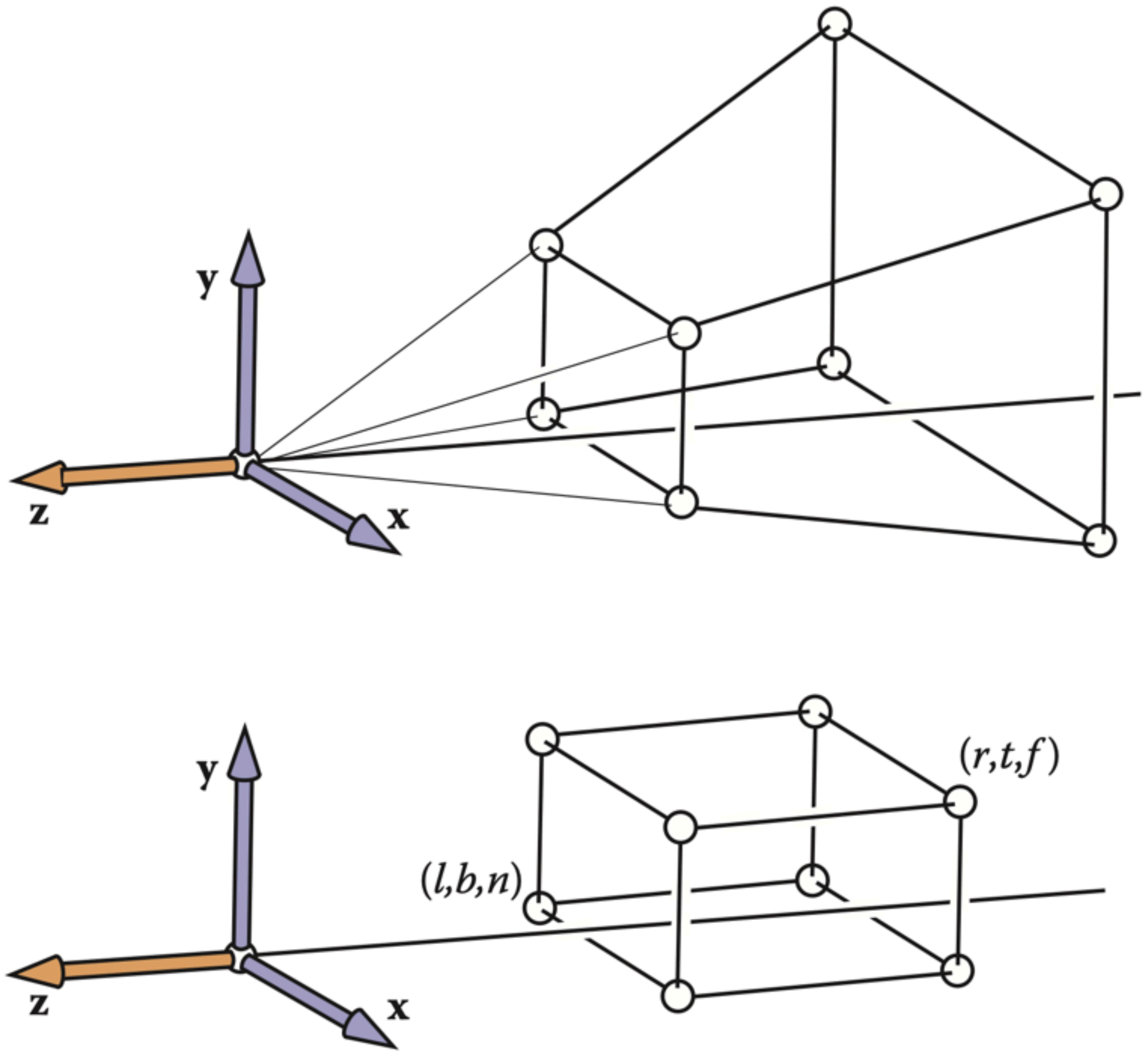


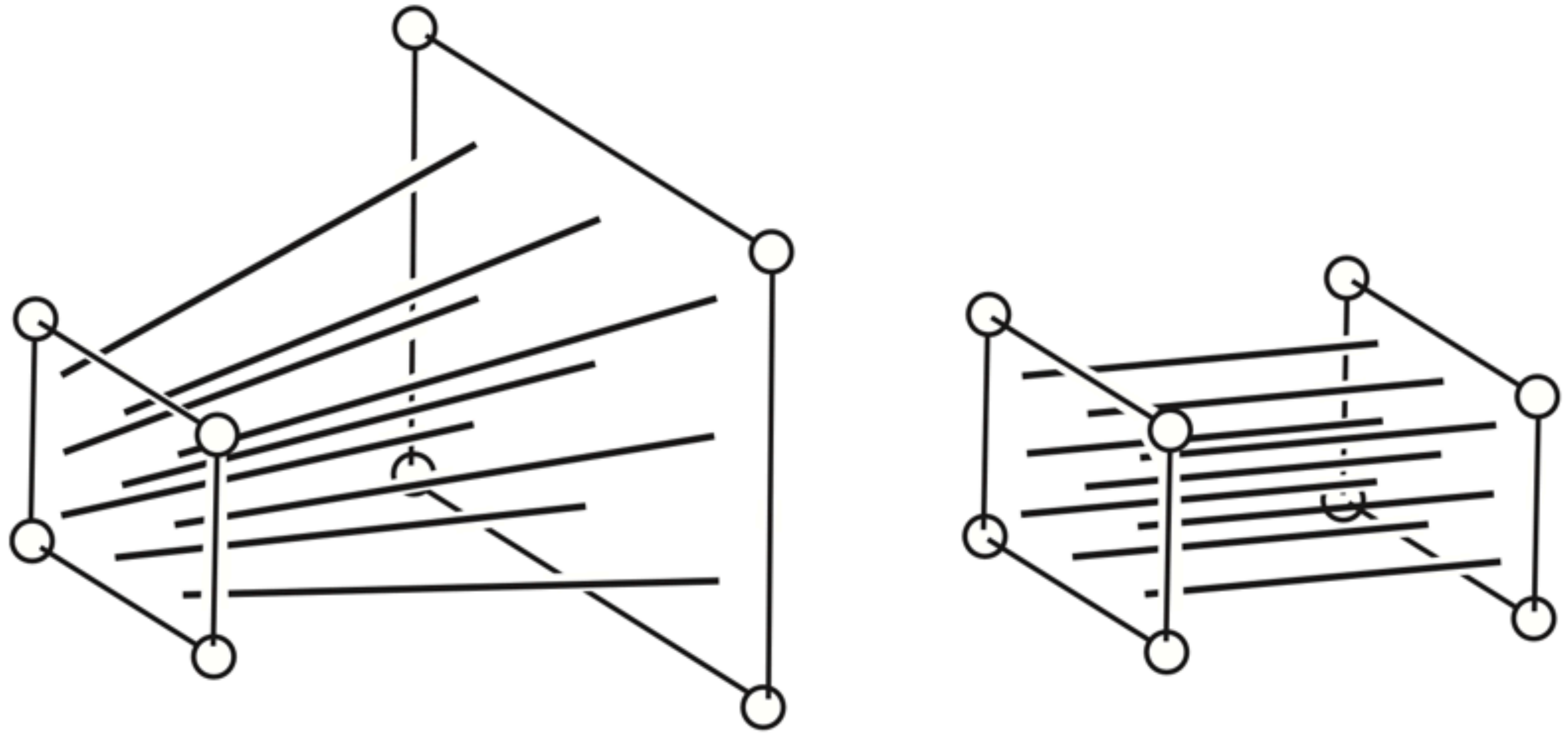
Perspective projection: $(x,y,z) \rightarrow (xd/z,\ yd/z)$

With homogeneous coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 1/d \\ z/d \end{bmatrix} \sim \begin{bmatrix} xd/z \\ yd/z \\ 1/z \\ 1 \end{bmatrix}$$

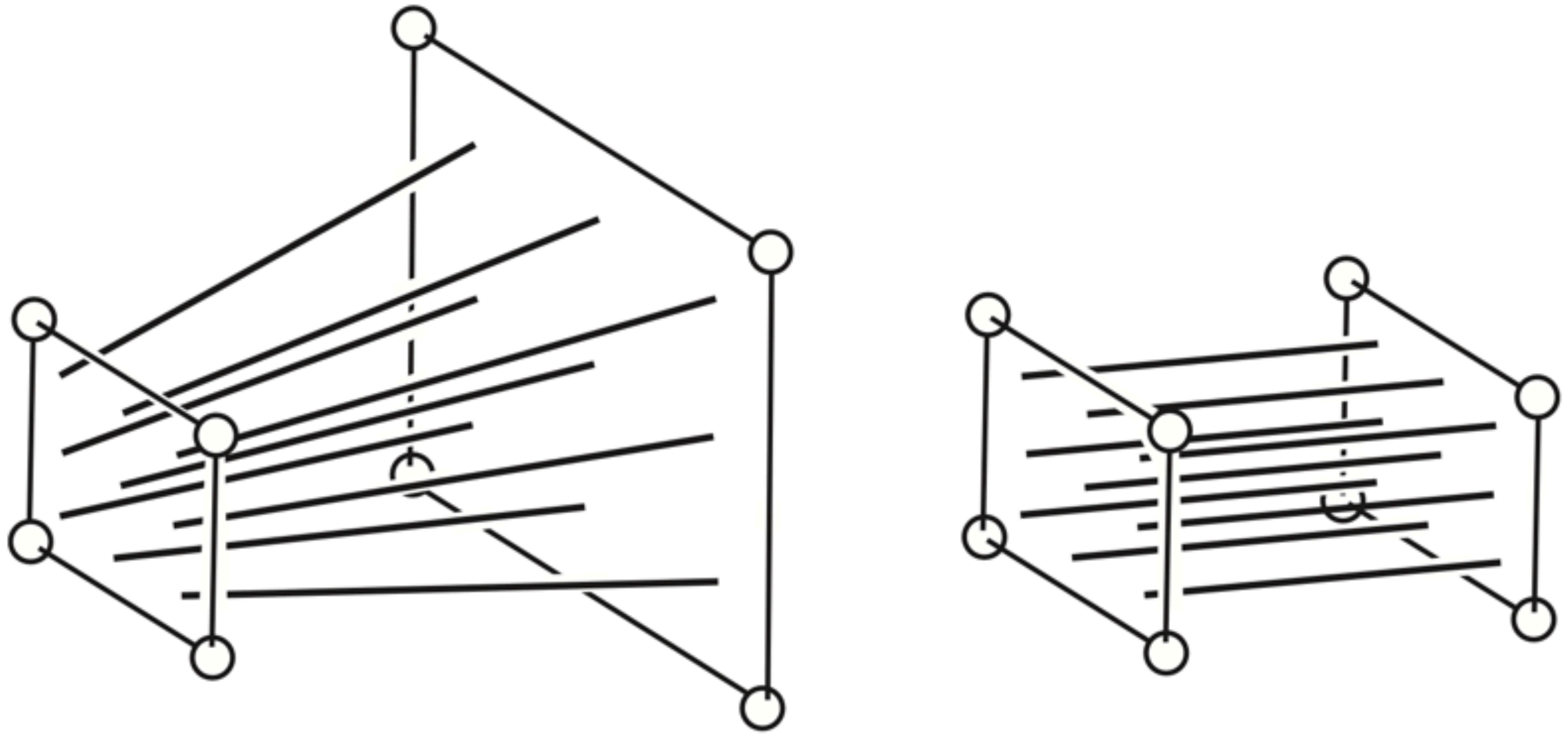
Corresponding matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

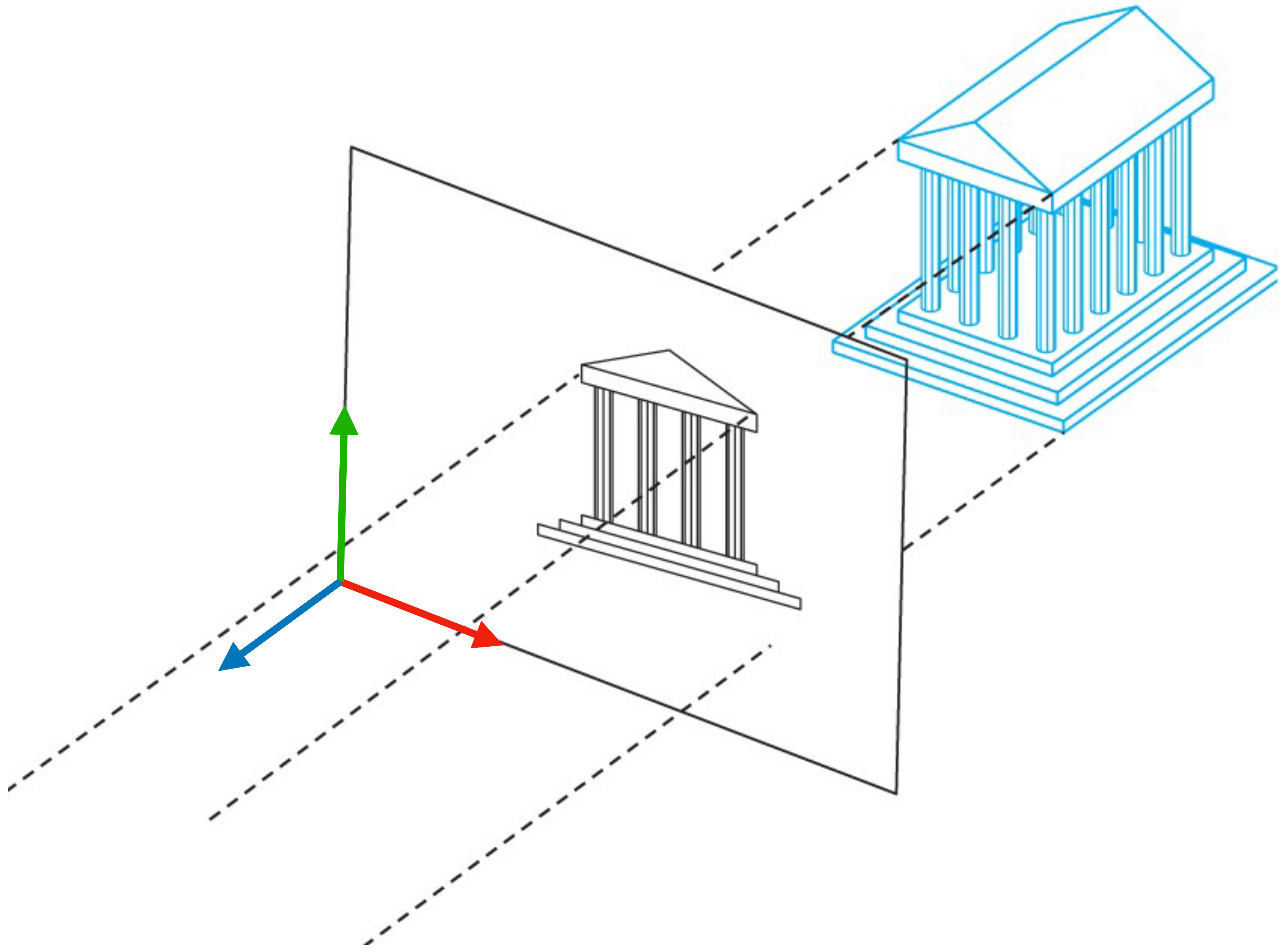




Marschner & Shirley, *Fundamentals of Computer Graphics*

The 4×4 projection matrix turns perspective projection into parallel projection.





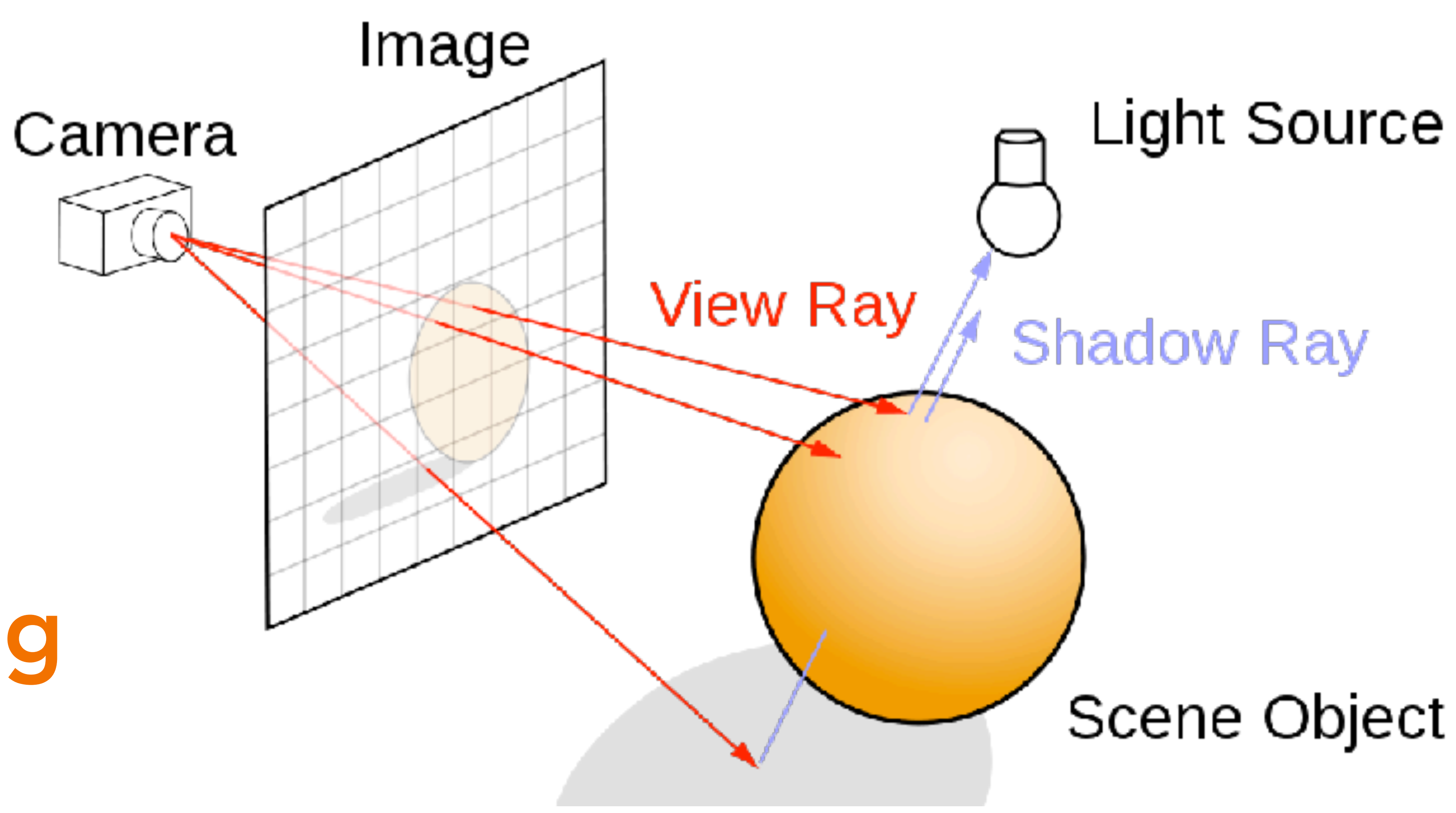
But we still need to do visibility testing even in parallel projection!

Visibility testing

Each pixel's colour should be given by the closest triangle that covers it.

This would be easy if I was rendering per-pixel instead of per-triangle:

for each *sample*:
    for each *triangle* that covers it:
        if *triangle* is closest surface seen so far:
            set *sample*.colour to *triangle*.colour



This is actually the basic idea behind **ray tracing**
(which we will cover later!)

Another way, more compatible with the rasterization pipeline:

for each *triangle*:
  for each *sample* that it covers:
    if *triangle* is closest surface seen by *sample* so far:
      set *sample*.colour to *triangle*.colour

This is what's actually done on the GPU!

Each sample needs to remember the closest depth it has seen, until all the triangles have been drawn.

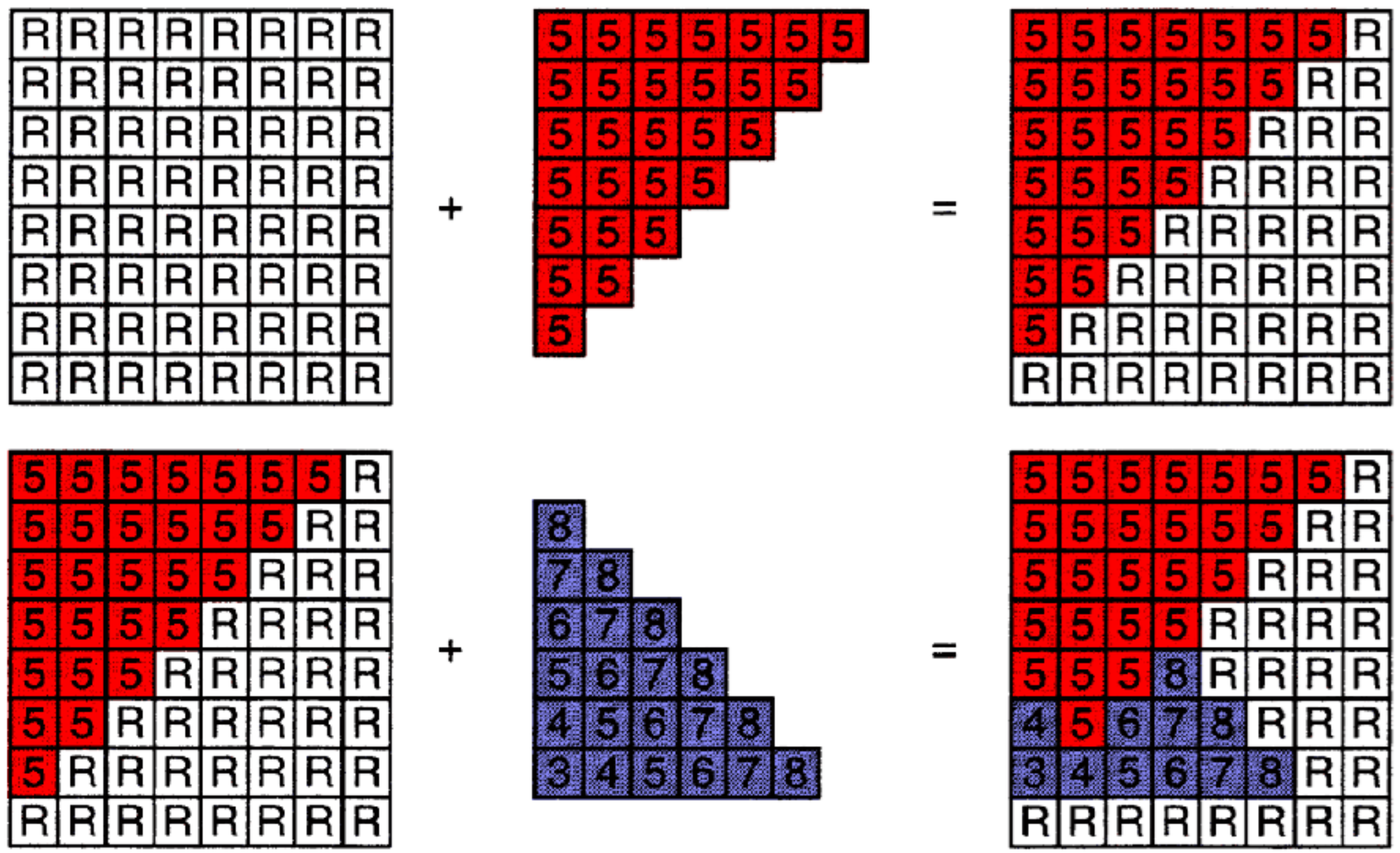
Z-buffering

Framebuffer now contains a colour buffer **and** a depth buffer (a.k.a. z-buffer)

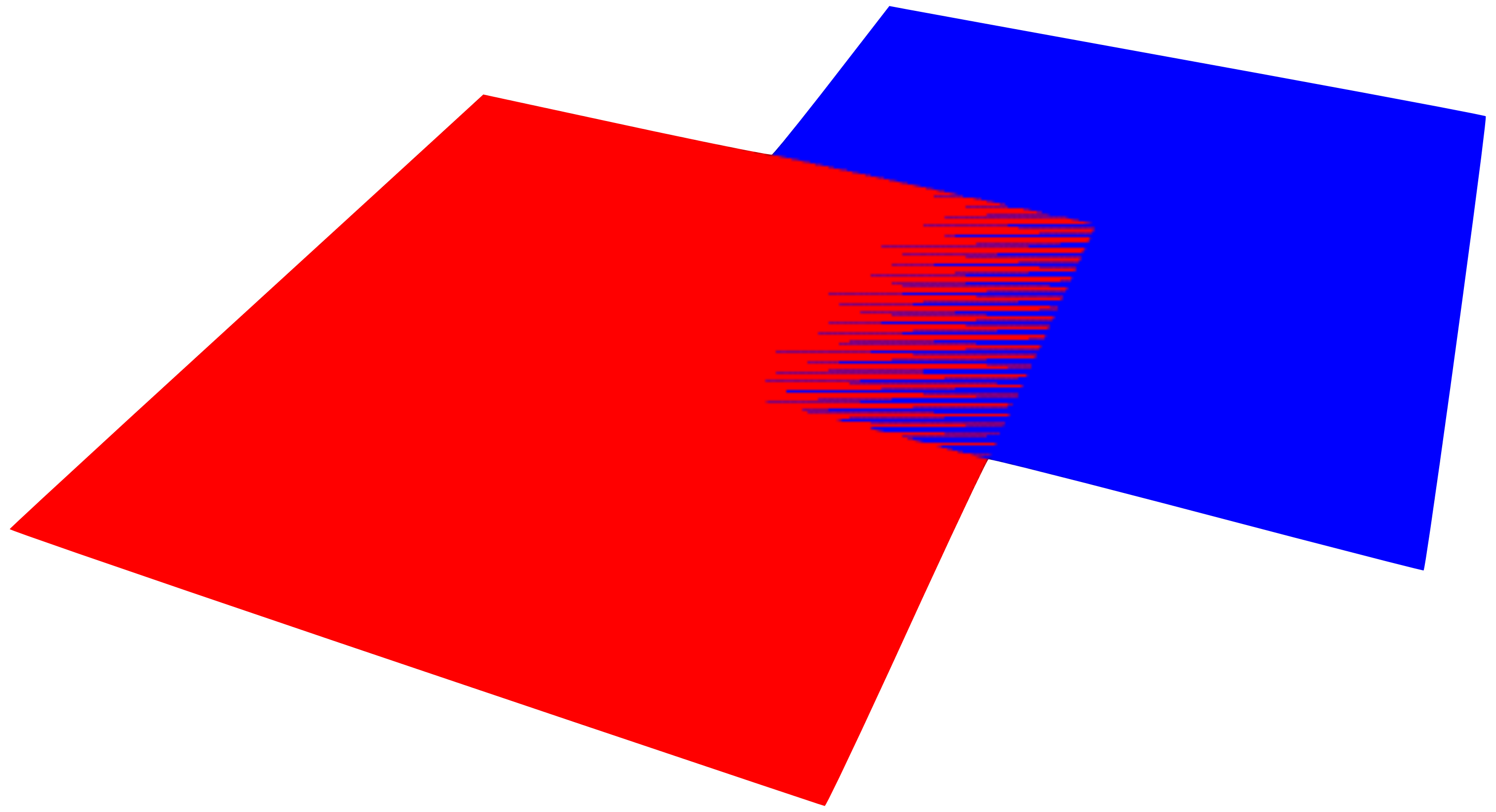
Colour

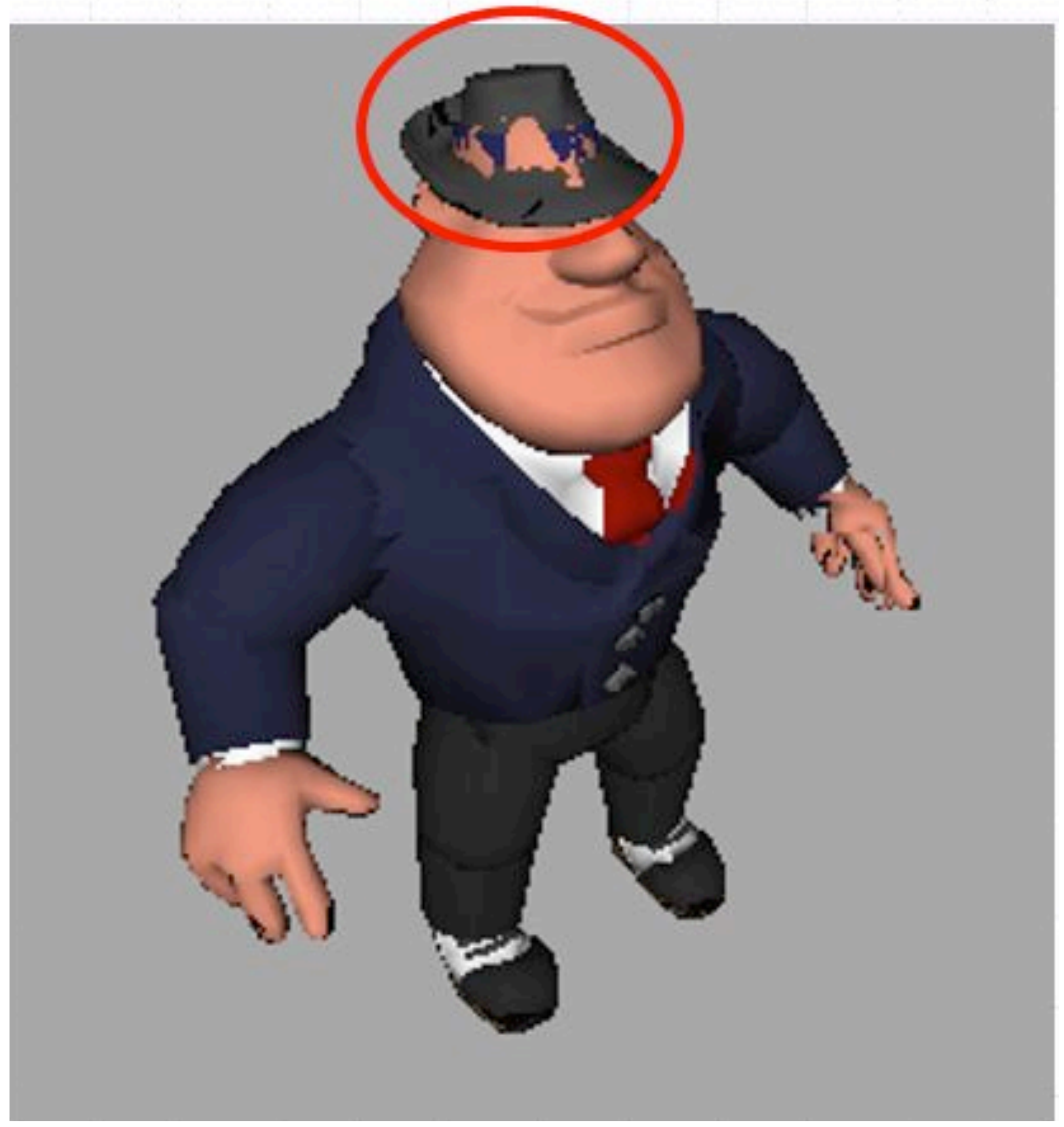
Depth

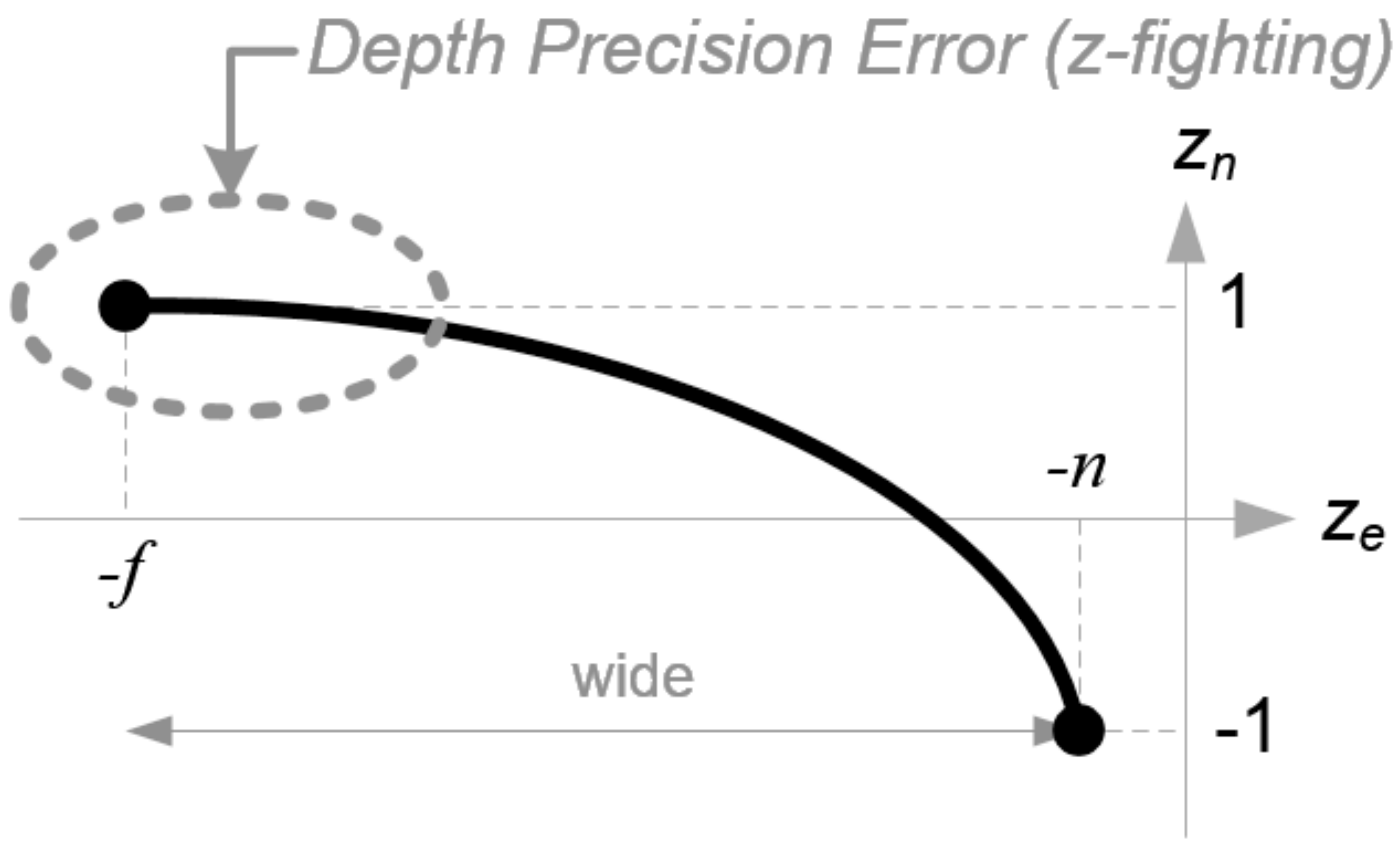
```
drawTriangle(t, rgb):
  for pixels (x,y):
    if isInside(x,y, t):
      z = depth(x,y, t)
      drawSample(x,y,z, rgb)

drawSample(x,y,z, rgb):
  if z < zbuffer[x,y]:
    color[x,y] = rgb
    zbuffer[x,y] = z
  else:
    # do nothing
```



Z-buffer can only store depth up to finite precision!



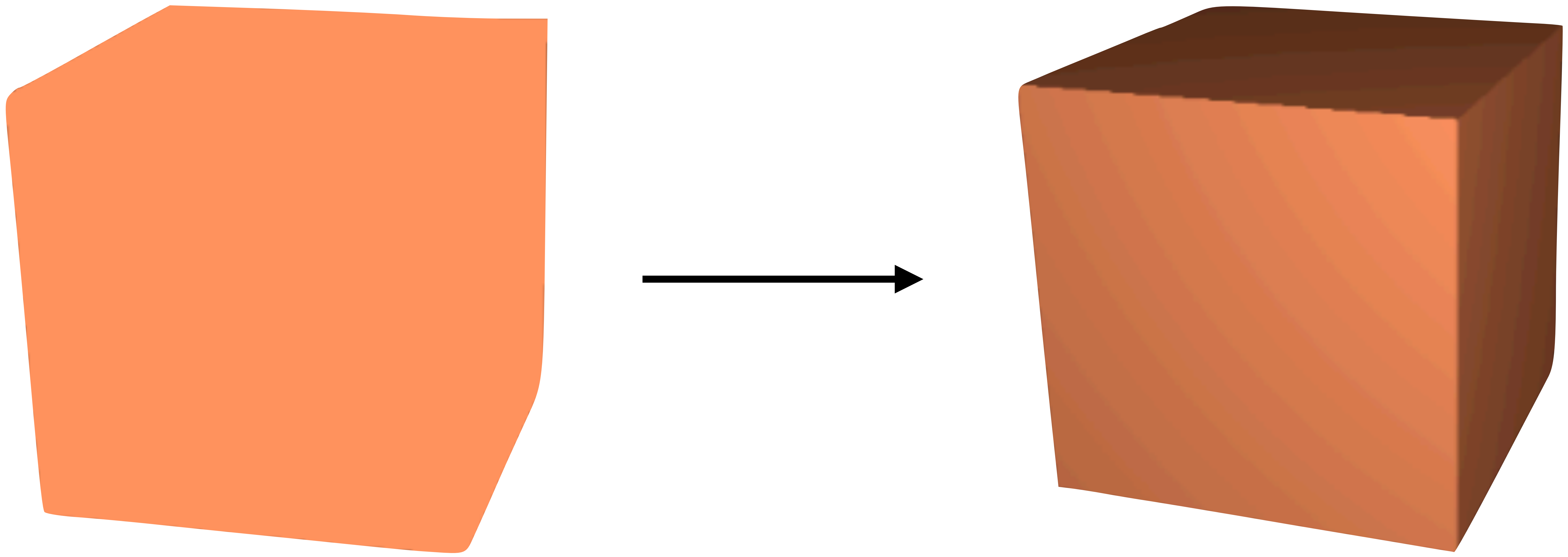




songho.ca

Different surfaces can map to same (rounded) depth: "z-fighting"

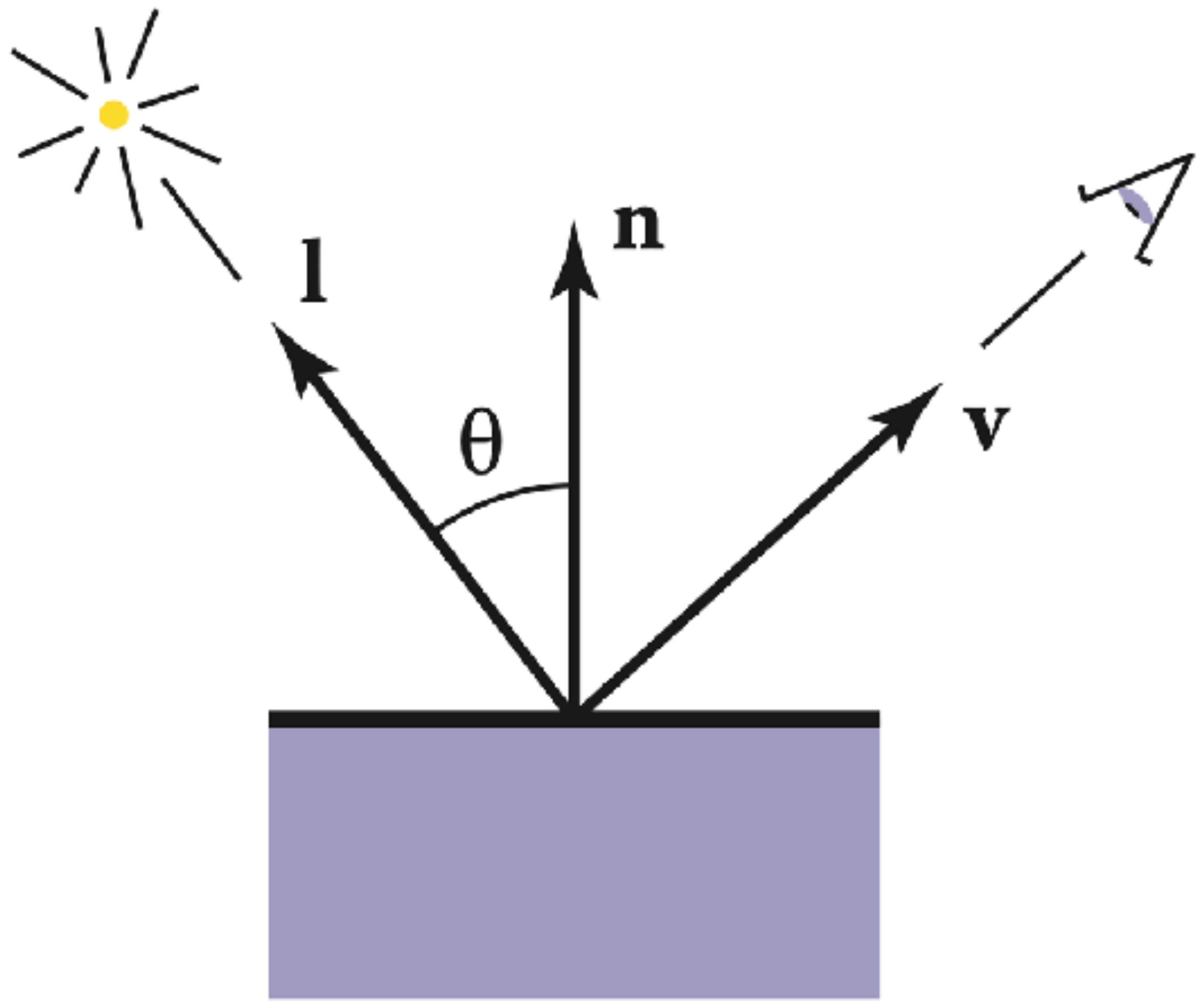
Shading



Local illumination

Light from a light source with some given intensity hits the surface point

- Light direction $\boldsymbol{\ell}$
- Surface normal $\mathbf{n}$
- Viewer direction $\mathbf{v}$



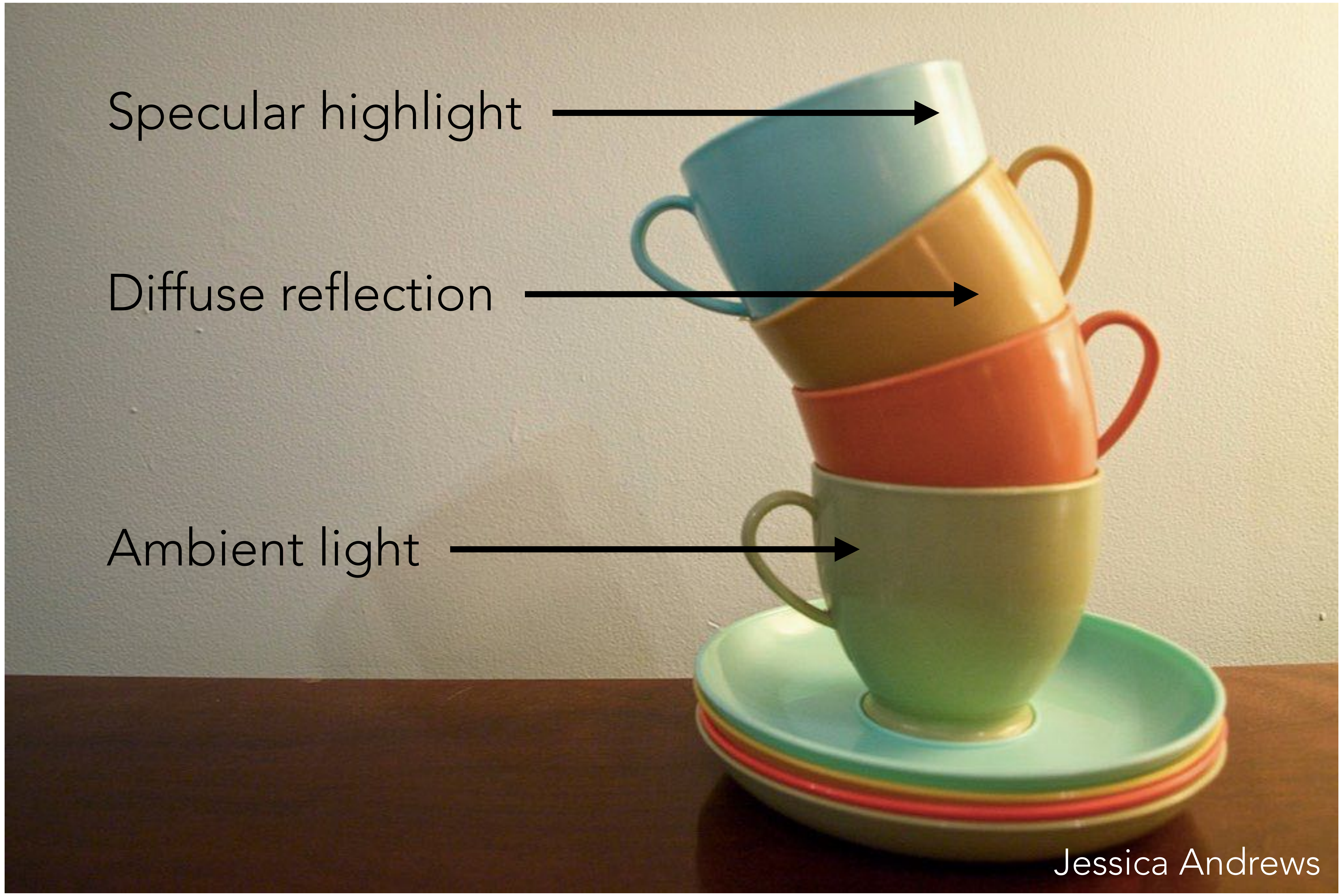
What intensity / colour of light is reflected towards the viewer?

Depends on surface properties (material, colour, roughness, coating, etc.)

For realistic materials, this can be pretty complicated…

Zeltner and Jakob 2018

A very simple shading model



Diffuse reflection: Lambertian model

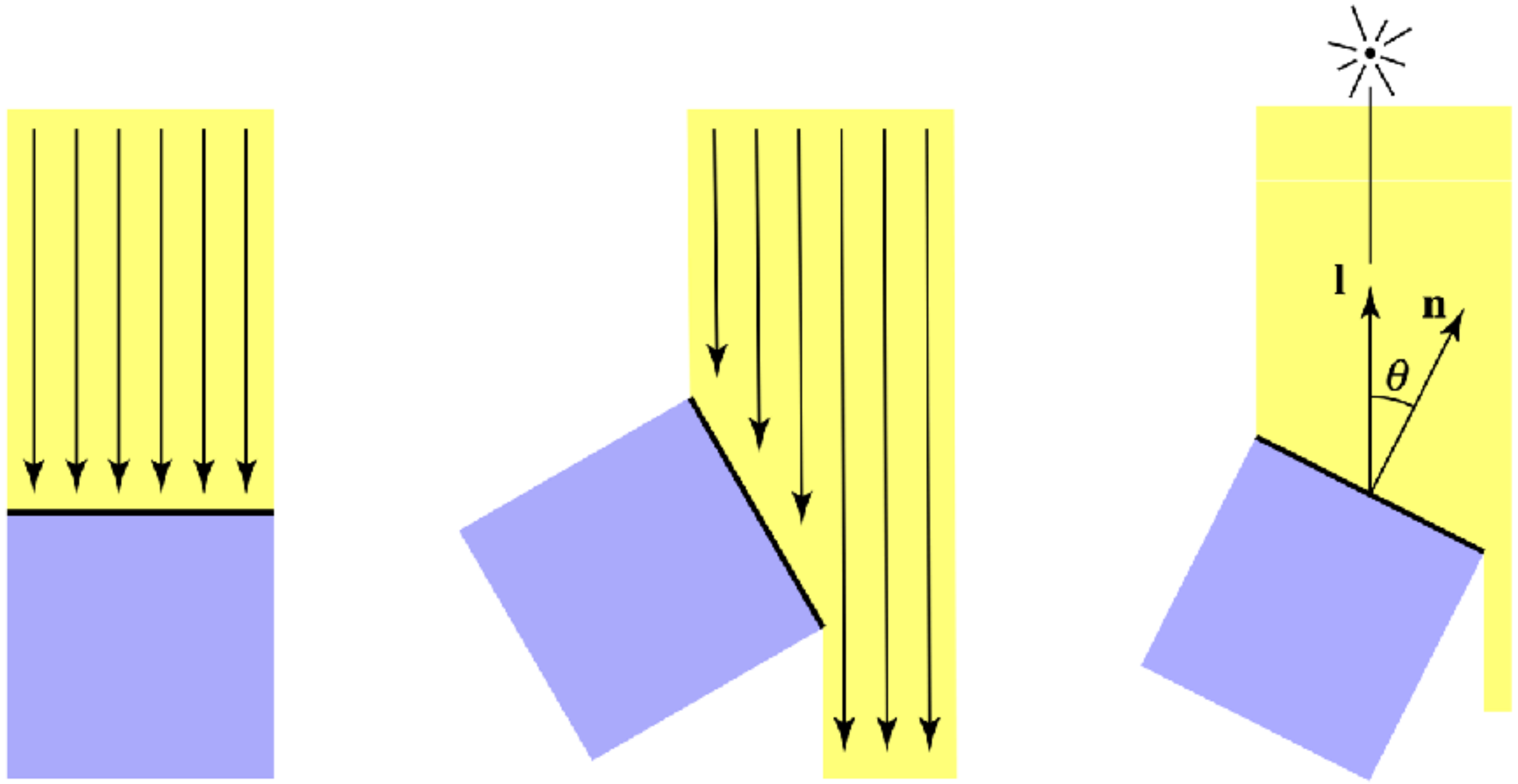
Assume the surface scatters the received light equally in all directions, i.e. the shaded colour is independent of view direction **v**.

But how much light is received?
Light **per unit area** $\propto \cos\theta = \mathbf{n} \cdot \boldsymbol{\ell}$

So, reflected light:

$$L_d = k_d \, I \max(0, \mathbf{n} \cdot \boldsymbol{\ell})$$

diffuse coefficient   incident light

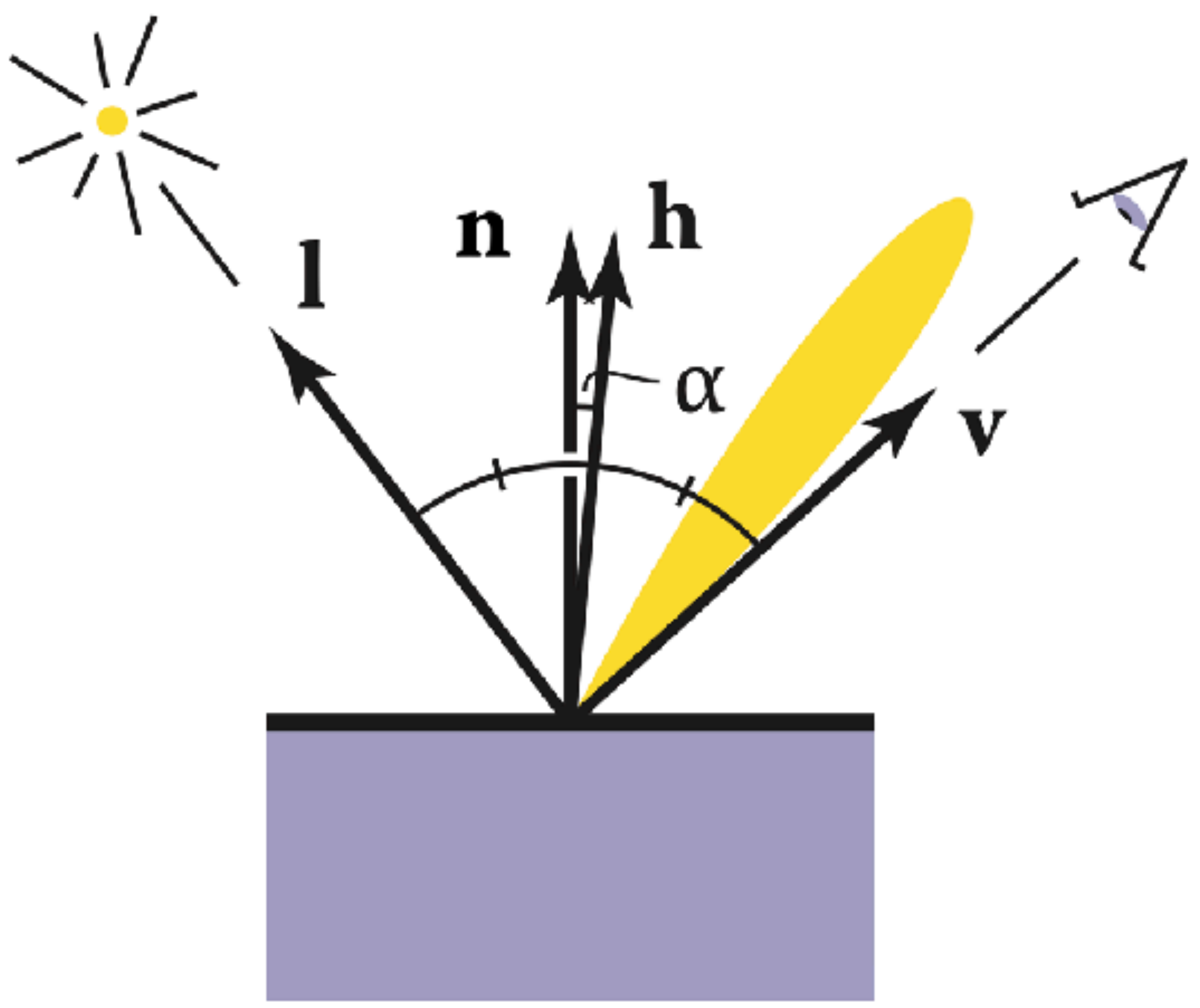


Both k_d and I can (should!) be RGB colours: multiplied componentwise

Specular reflection: Blinn-Phong model

Perfect mirror: Reflection is bright if and only if **v** is exactly "opposite" to $\boldsymbol{\ell}$

$$\text{bisector}(\mathbf{v}, \boldsymbol{\ell}) = \mathbf{n}$$



Shiny surface: Reflection is bright if **v** is **close to** being opposite to $\boldsymbol{\ell}$

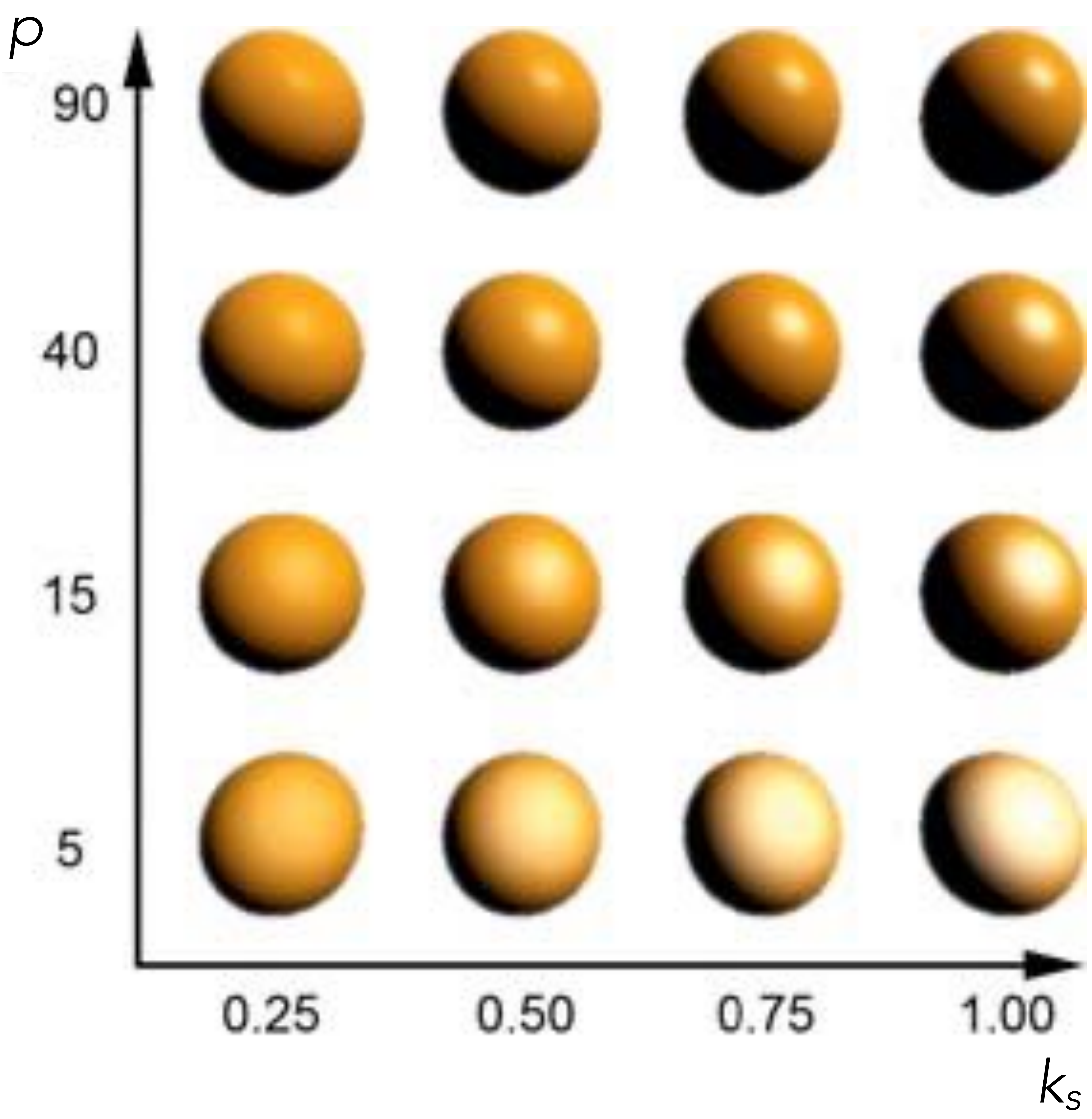
$$\mathbf{h} = \text{bisector}(\mathbf{v}, \boldsymbol{\ell}) = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

halfway vector

$$L_s = k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

Phong exponent

specular coefficient



Hughes et al.

Shading frequency

Per triangle (flat shading)

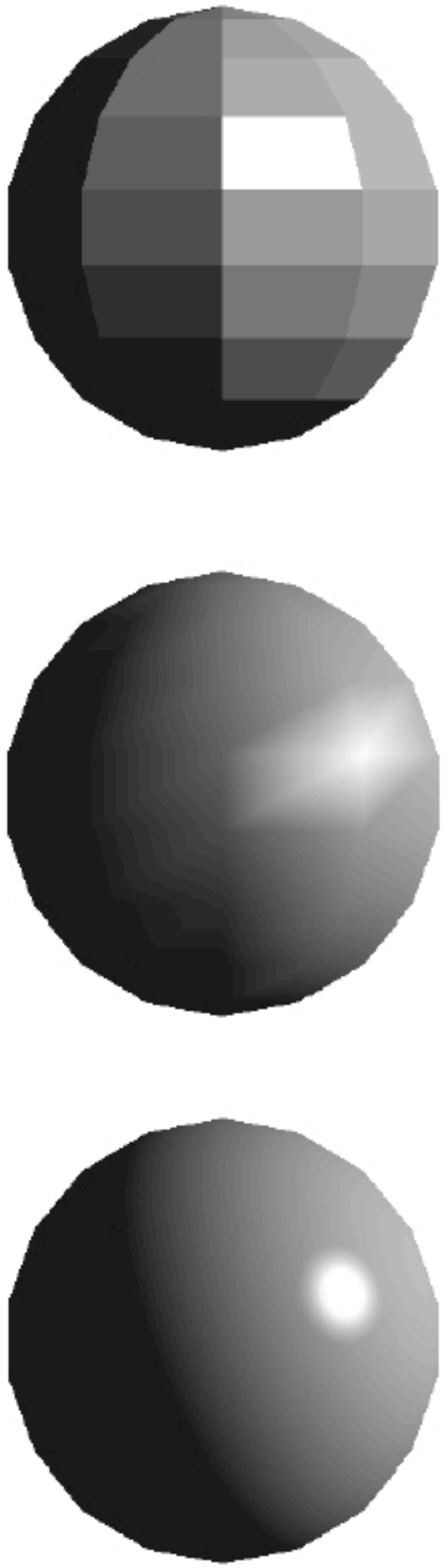
- Not good for surfaces that are supposed to be smooth

Per vertex (“Gouraud shading”)

- Need normal vector at each vertex
- Colour interpolated across triangle

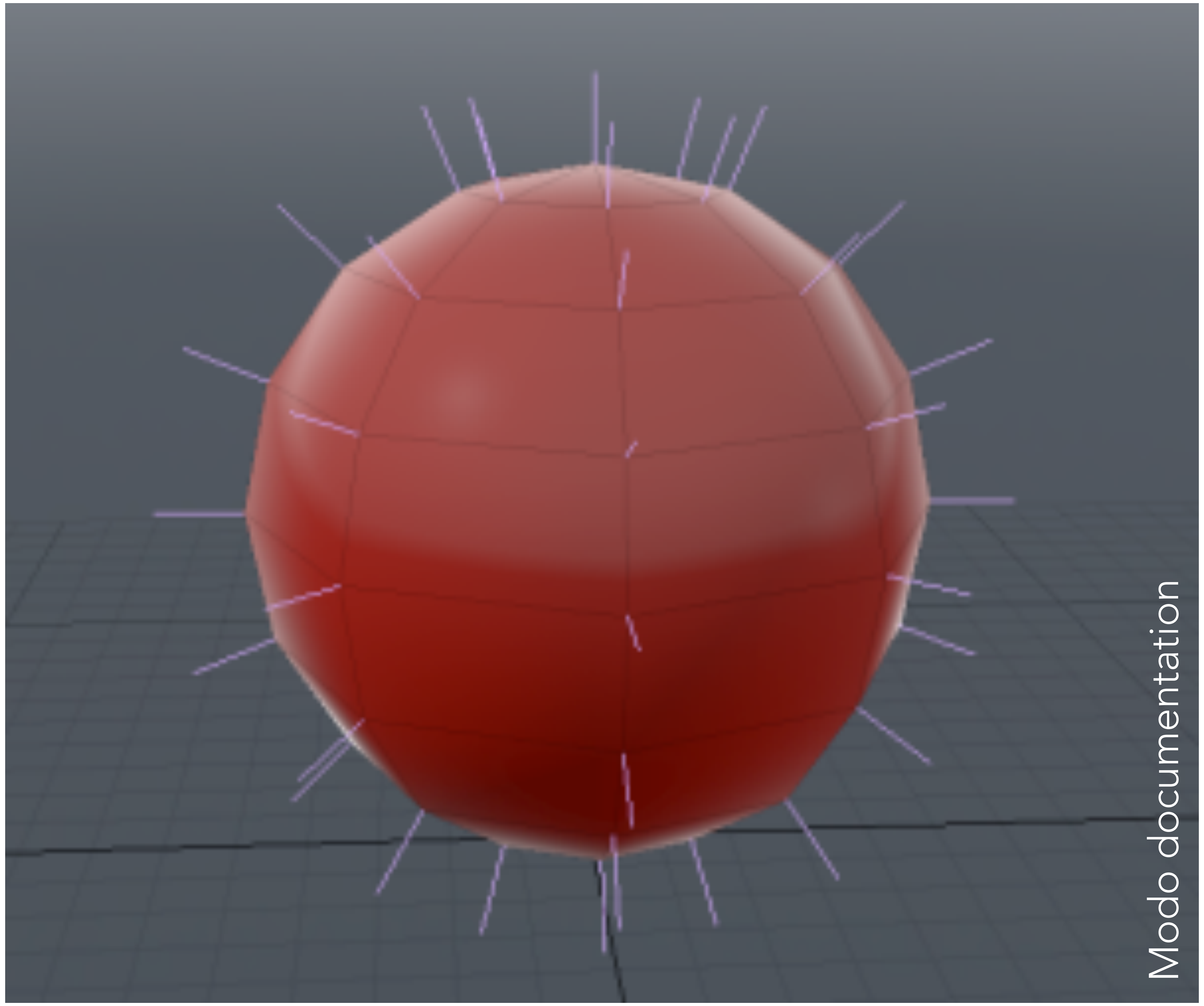
Per pixel (“Phong shading”)

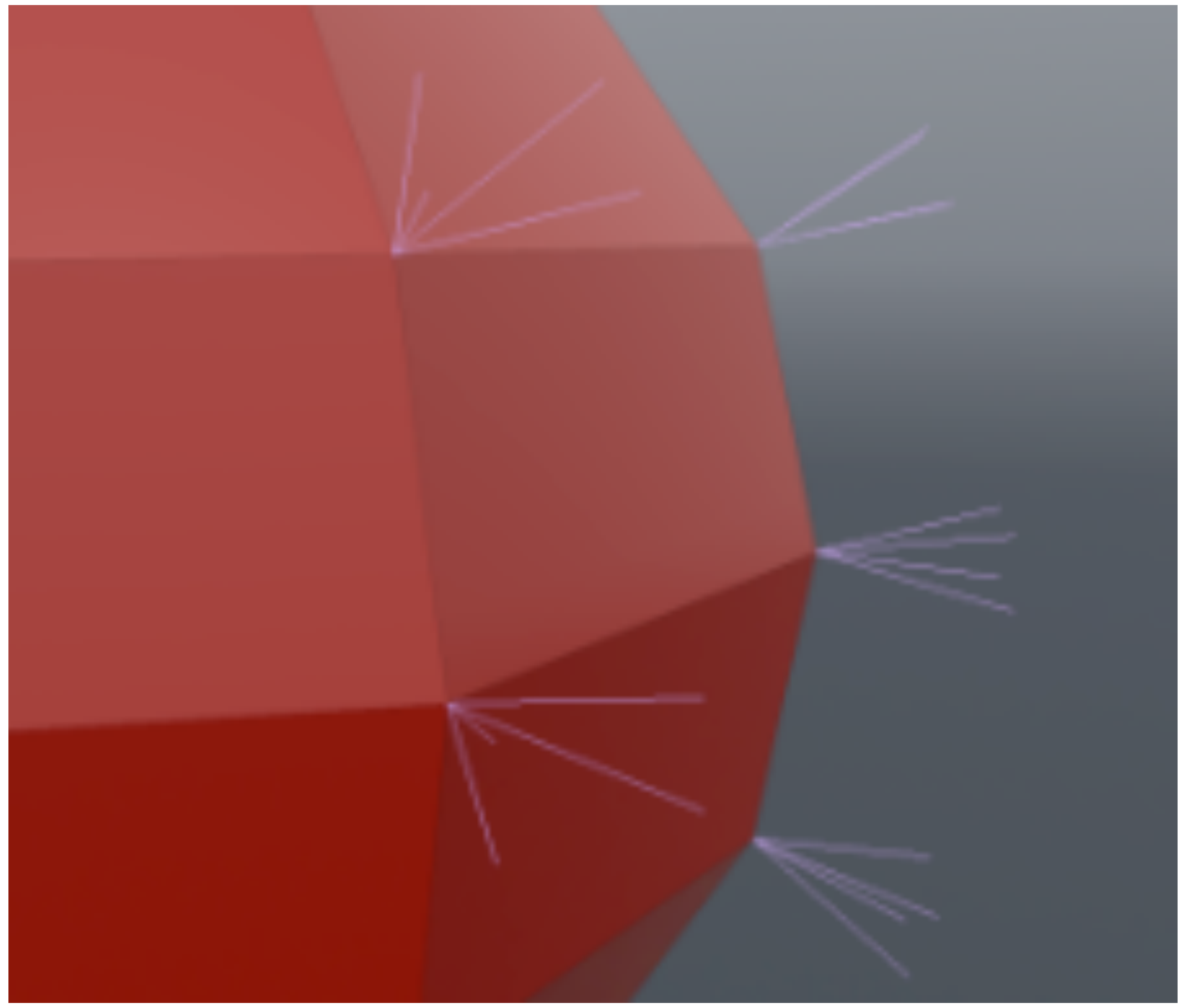
- Vertex normal interpolated across triangle (and then normalized!)



cg2010studio.com

Vertex normals





Light is coming from the right. Why isn't the left side totally black?

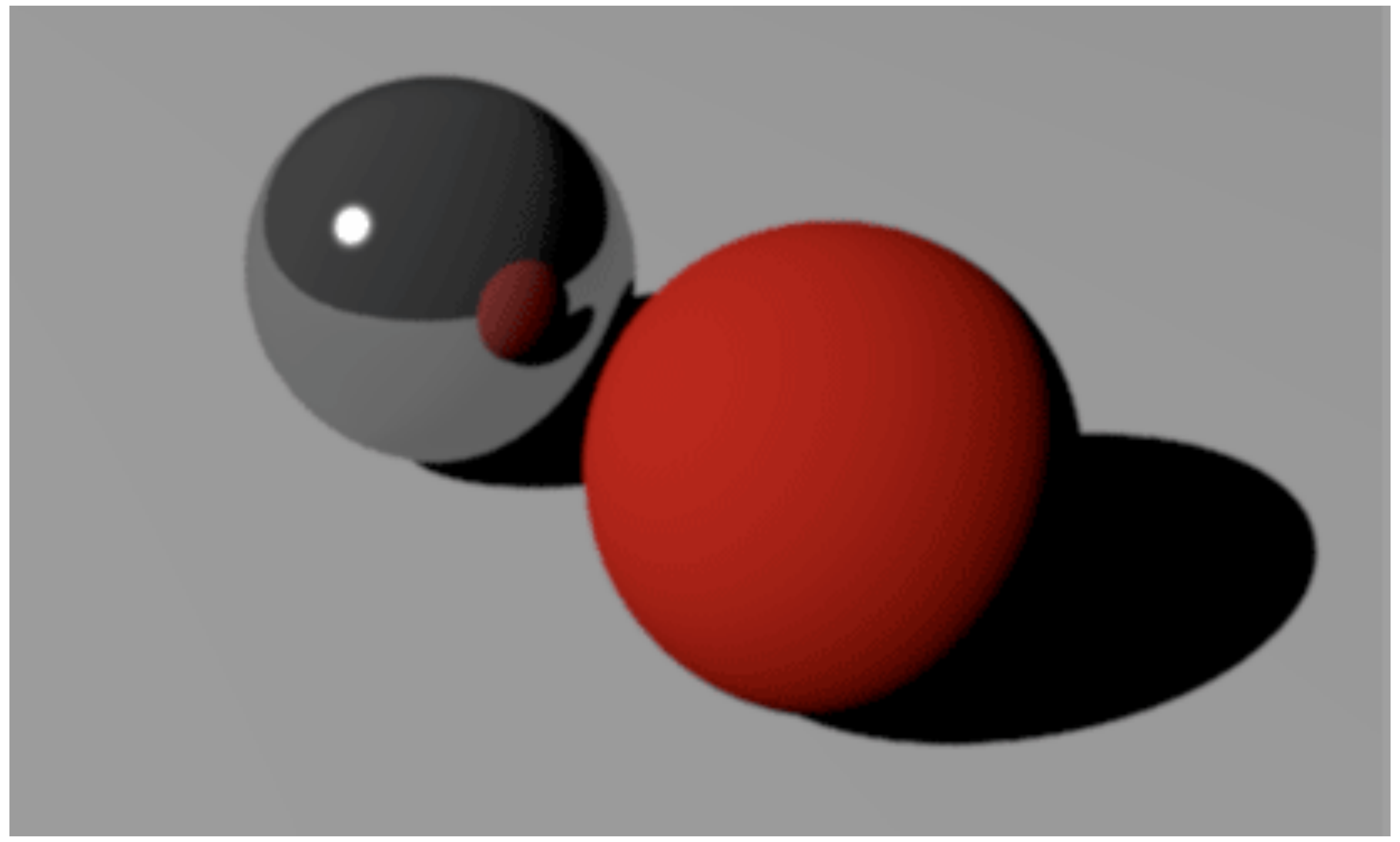
Light is coming from the right. Why isn't the left side totally black?

Ambient light

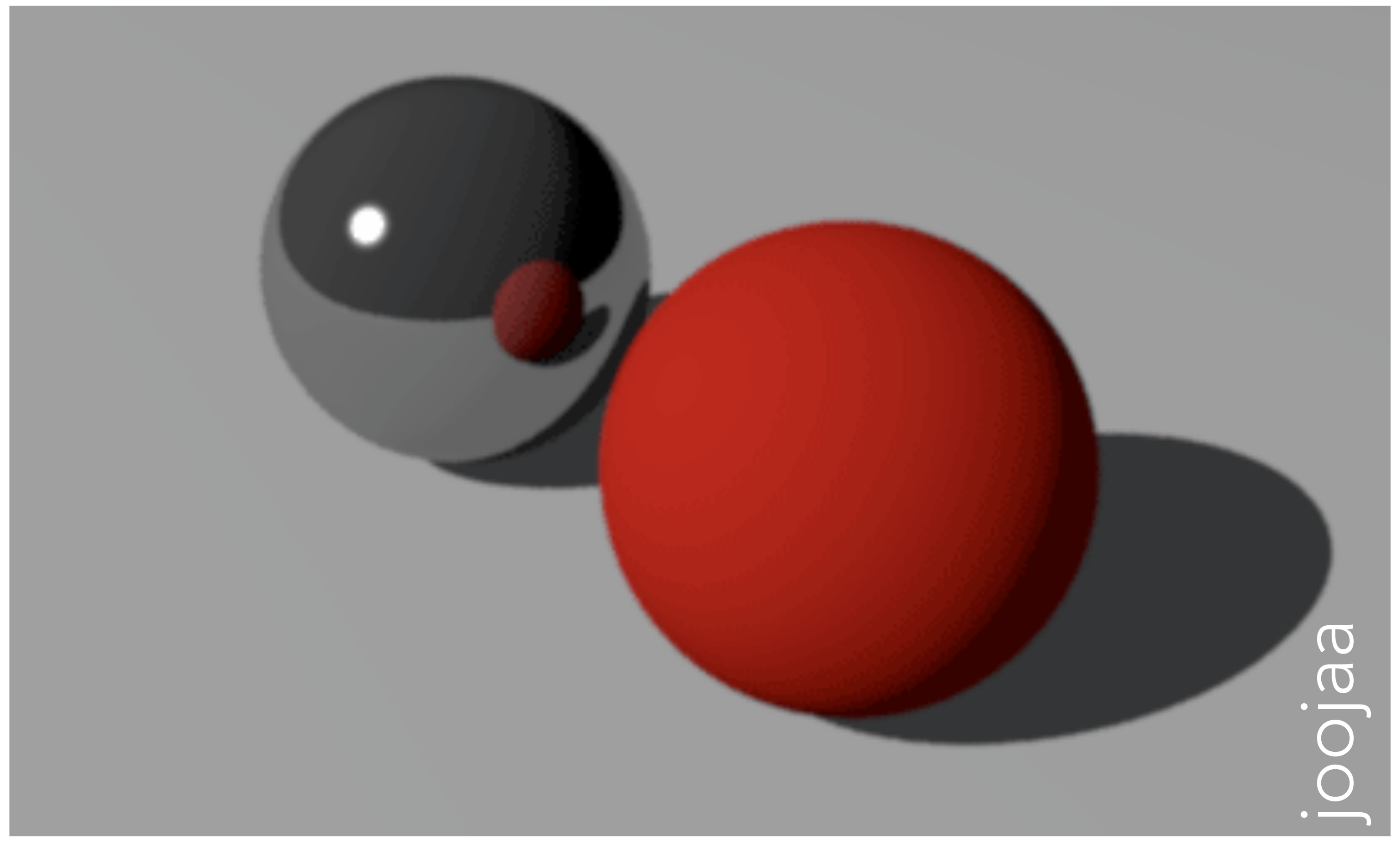
Light bounced around the scene is **nonlocal**: can't compute from **v**, **n**, $\boldsymbol{\ell}$ only

Instead, just assume there is a constant amount of indirect lighting everywhere

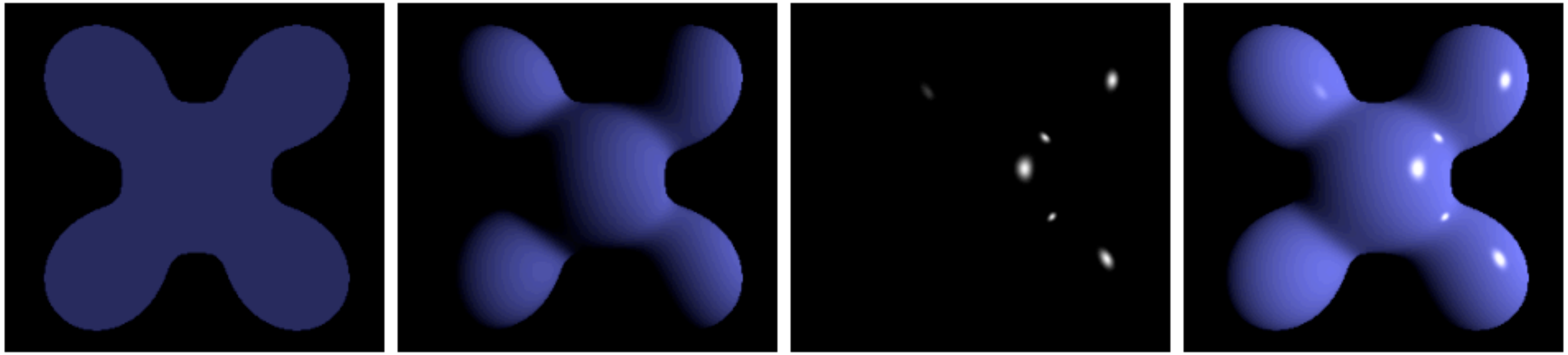
$$L_a = k_a \, I_a$$



Without ambient light



With ambient light



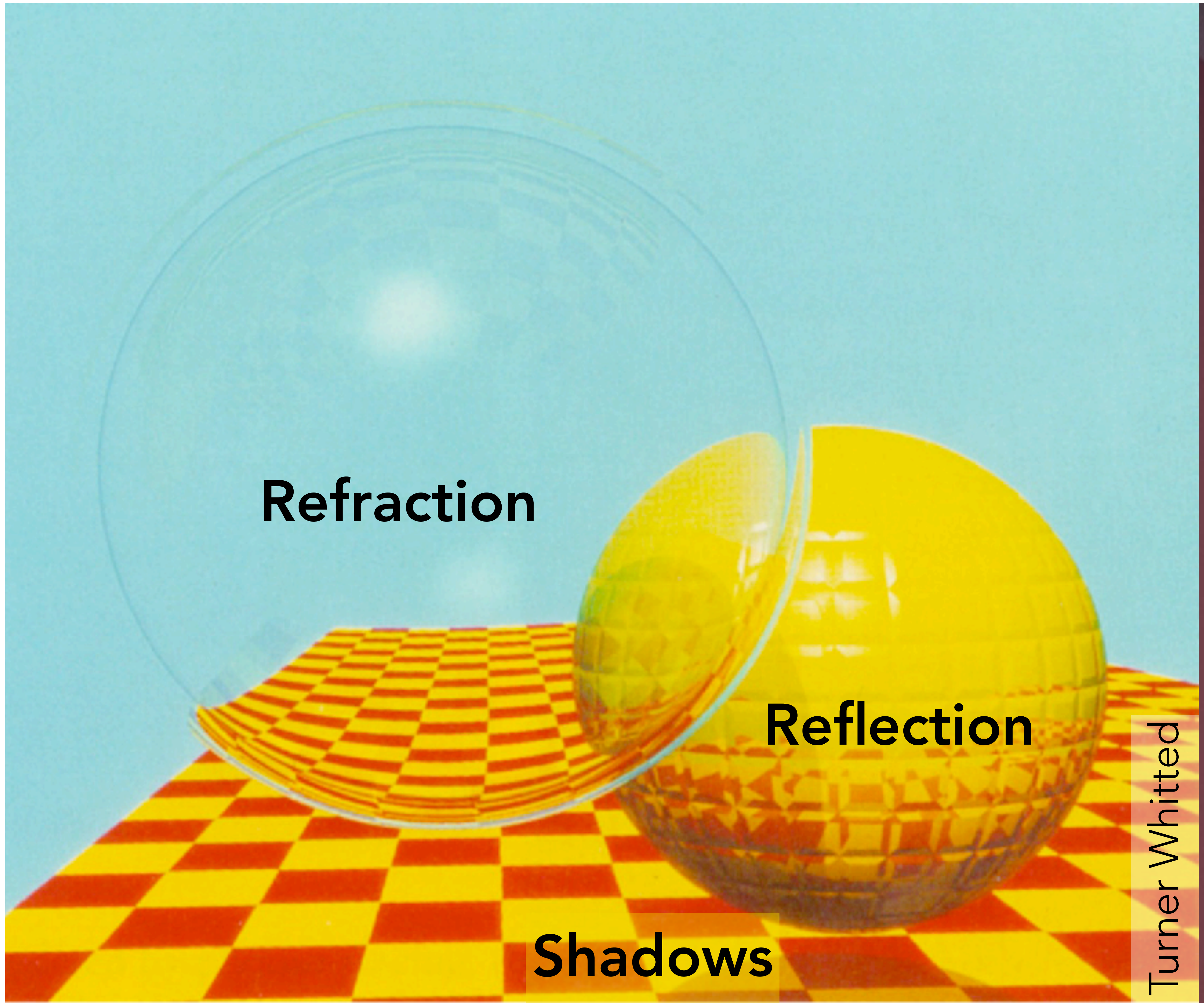
Ambient + Diffuse + Specular = **Blinn-Phong reflectance model**

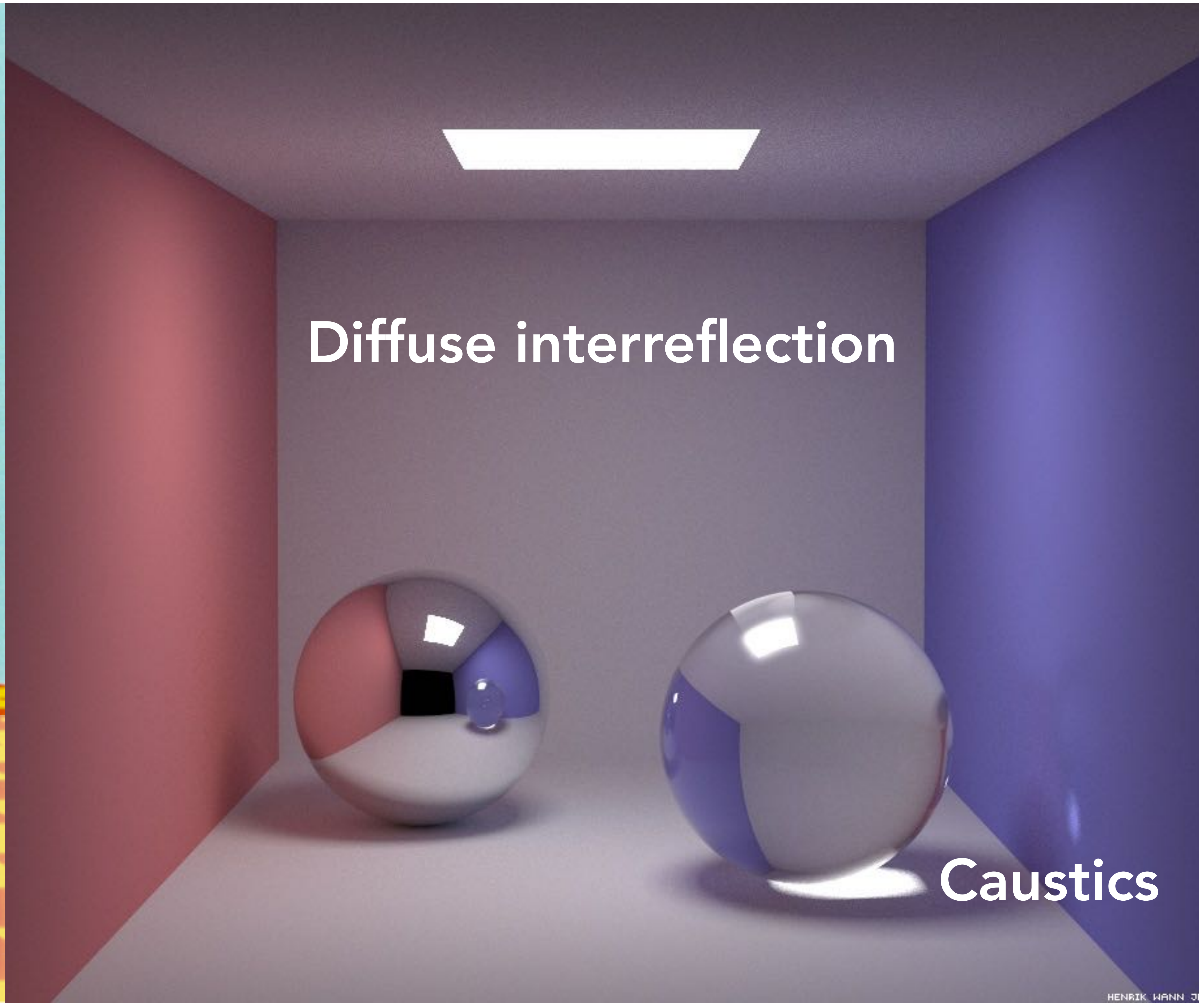
$$\begin{aligned} L &= L_a + L_d + L_s \\ &= k_a\, I_a + k_d\, I \max(0, \mathbf{n} \cdot \boldsymbol{\ell}) + k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^p \end{aligned}$$

k_a, k_d, k_s (colours) and p (scalar) control the material's appearance

If multiple lights I_1, I_2, …: add up diffuse and specular terms for each light

What phenomena are not captured?



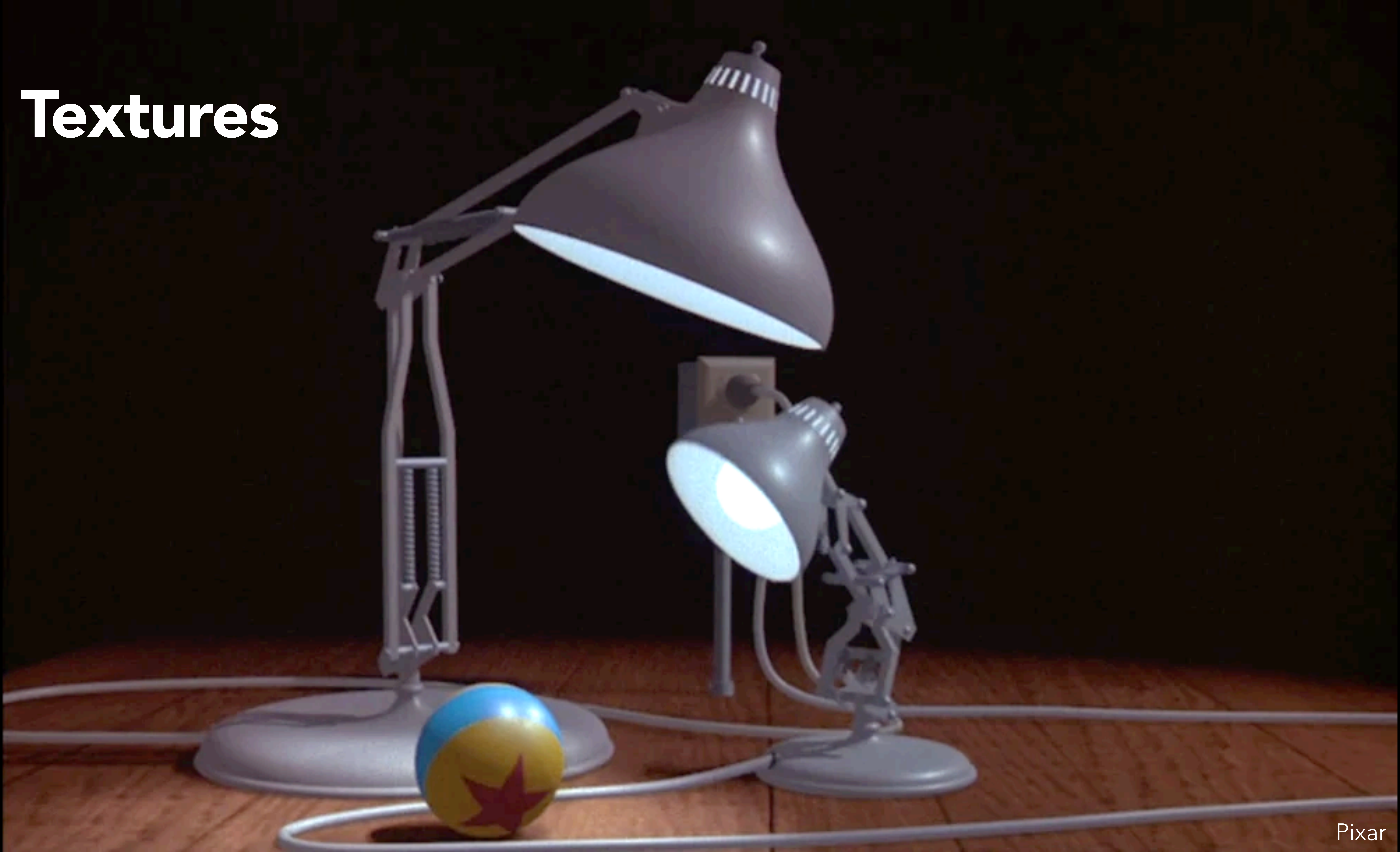


Henrik Wann Jensen

Texture mapping

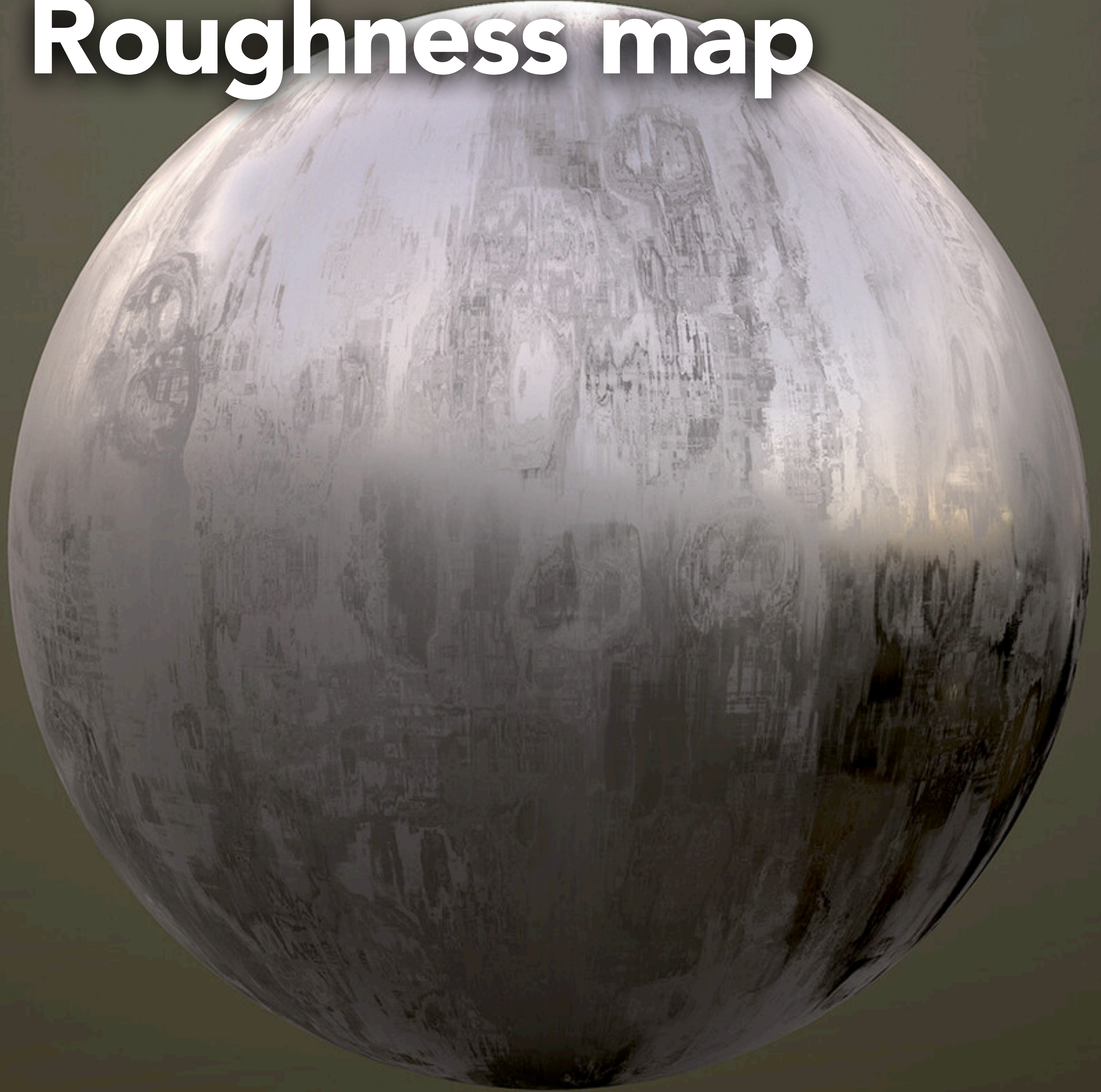
Chan et al.

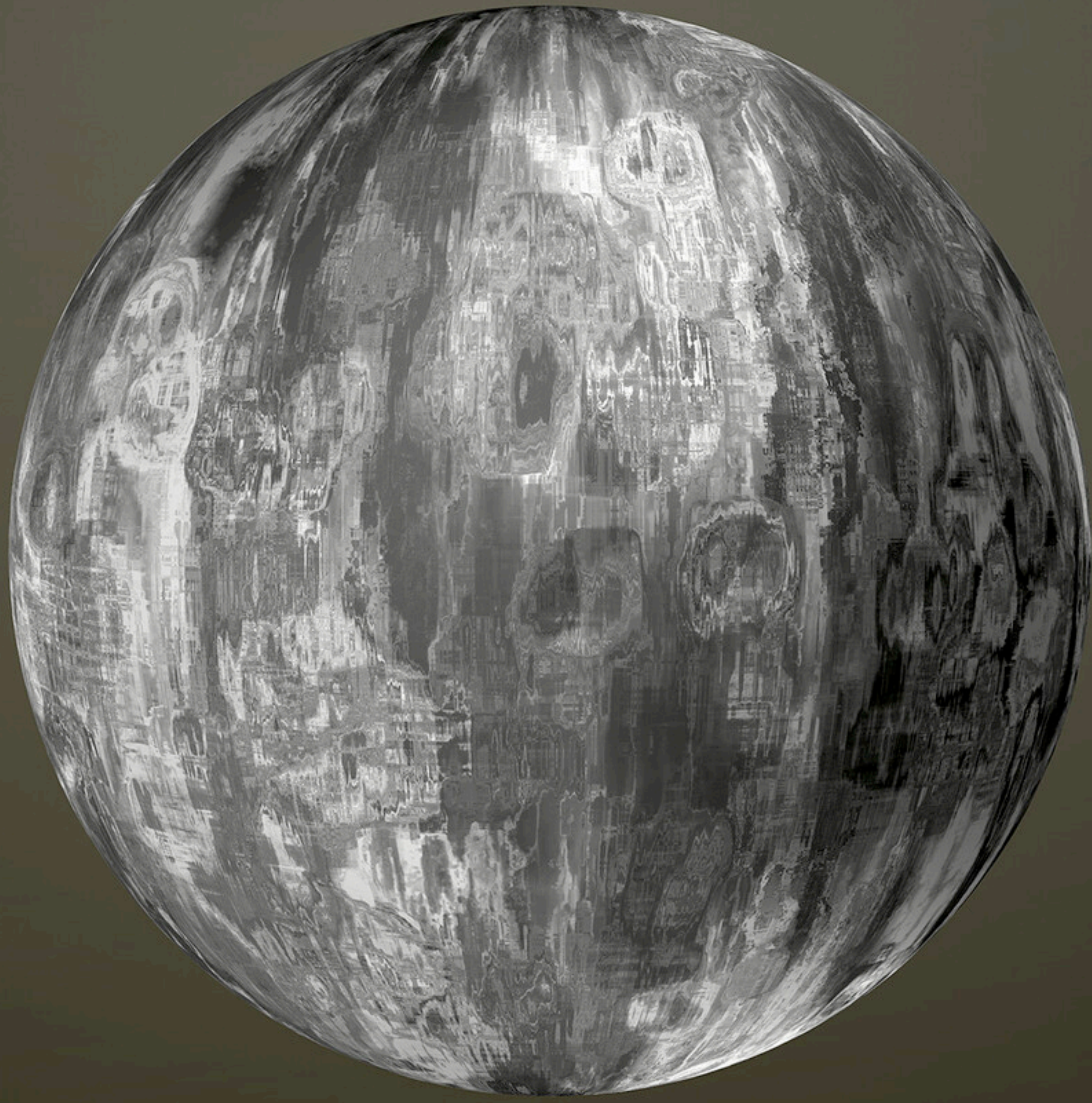
Textures



Pixar

Roughness map

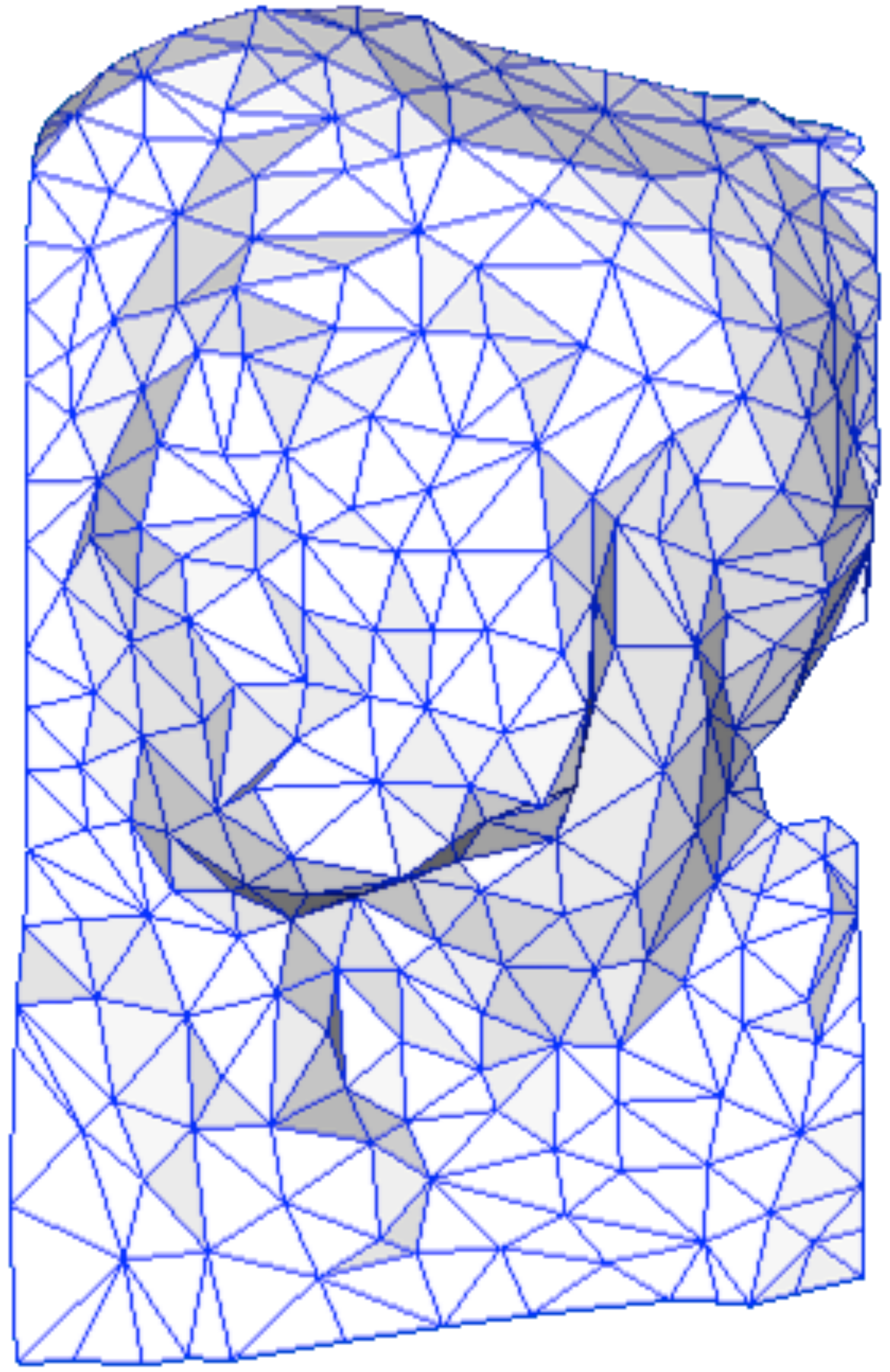




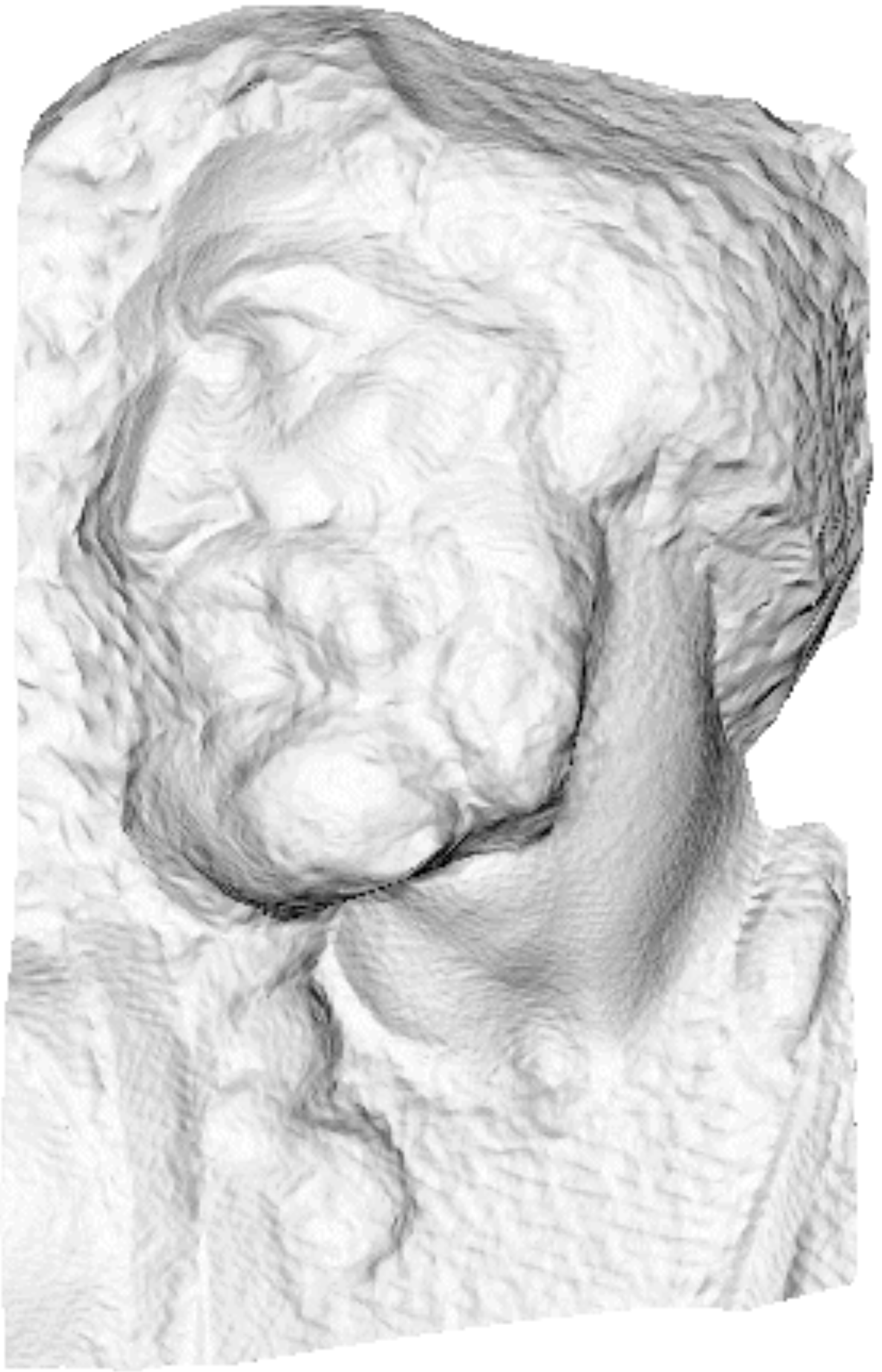
Francesco Saviano

Normal mapping

original mesh
4M triangles



simplified mesh
500 triangles

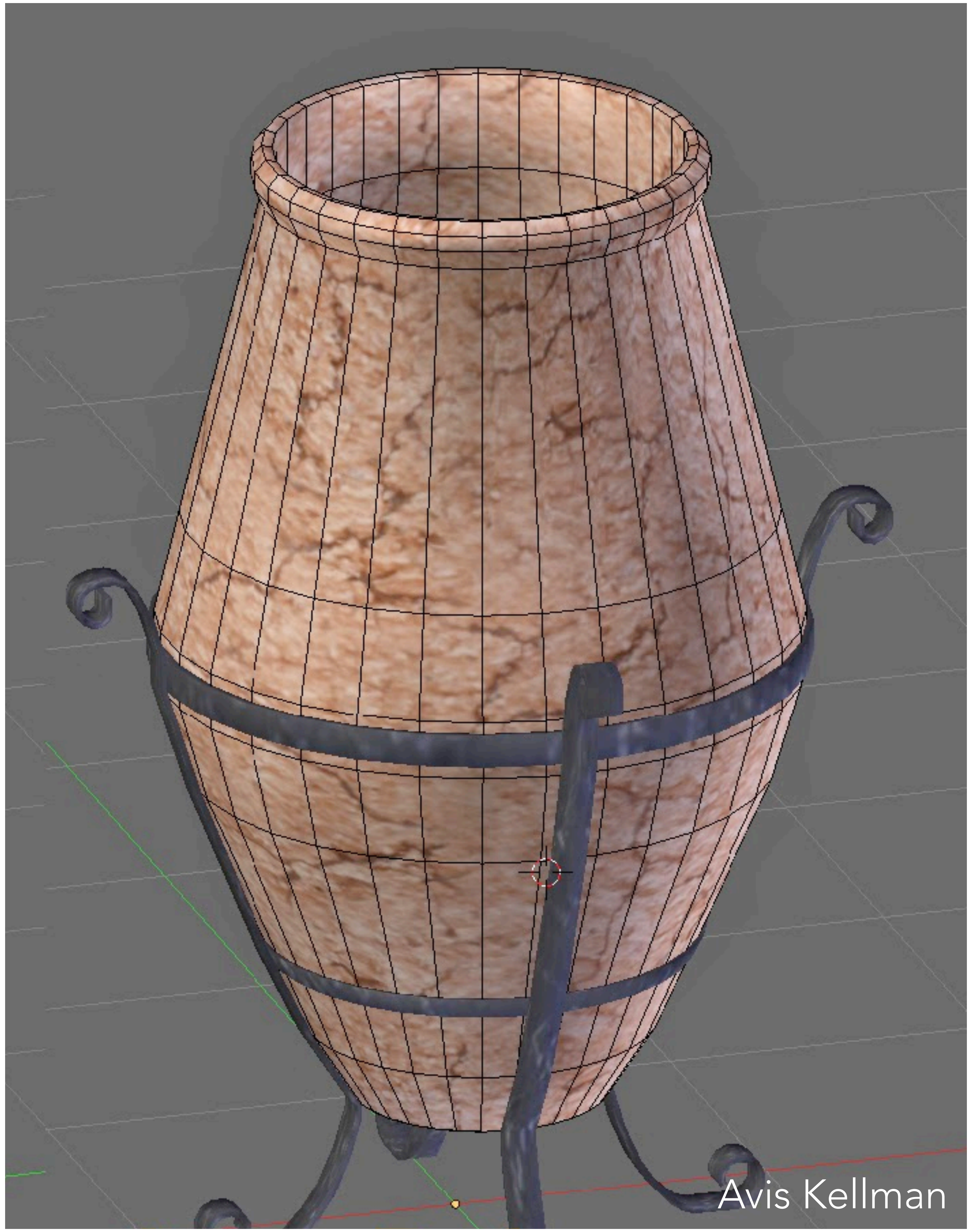


simplified mesh
and normal mapping
500 triangles

Paolo Cignoni

Crytek

RYSE SON OF ROME

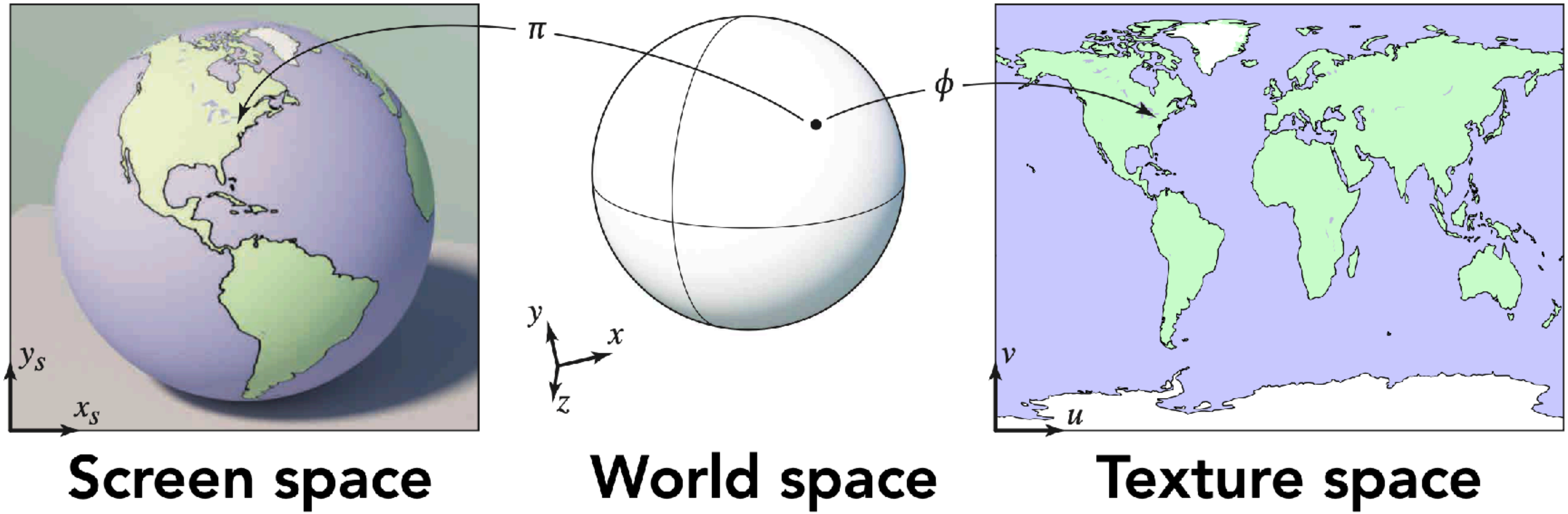


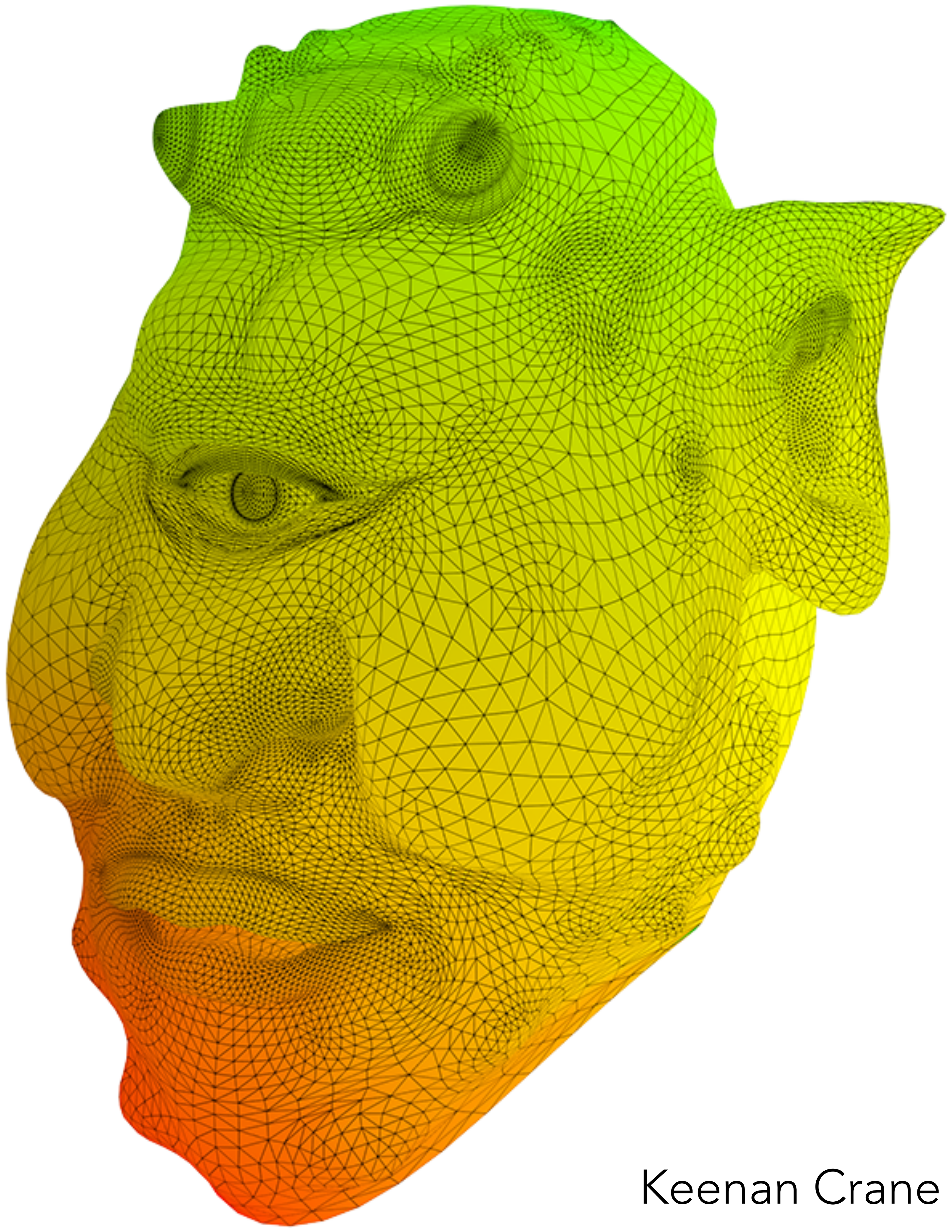
Detail (e.g. colour, roughness, normal, etc.) is some function from surface points to e.g. RGB

Easiest way is to store it in a 2D lookup table $(u,v) \rightarrow (r,g,b)$, i.e. an image!

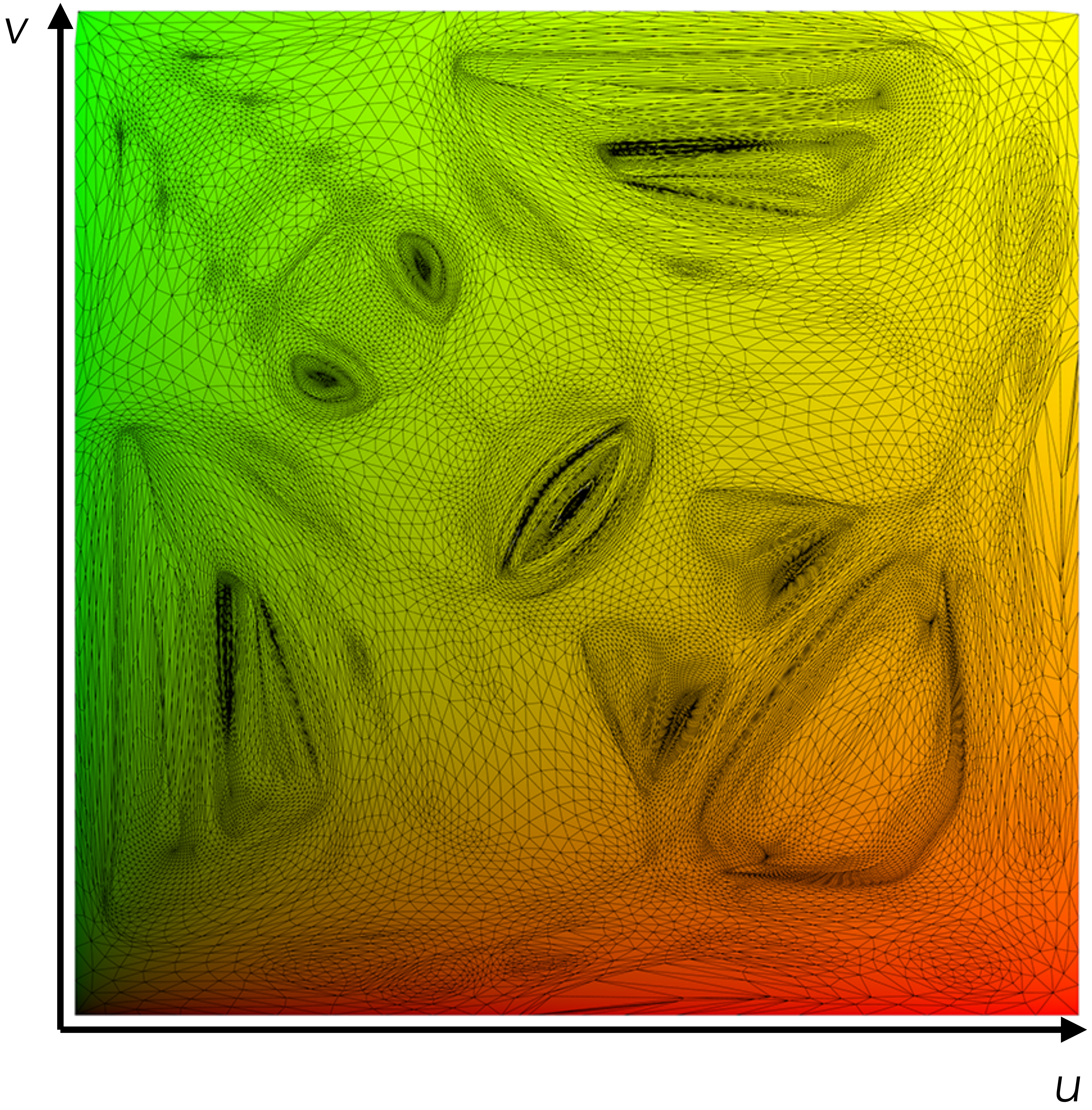
Then we also need to specify for each surface point (x,y,z) which location in the image (u,v) to pick up the colour from: **texture coordinates**

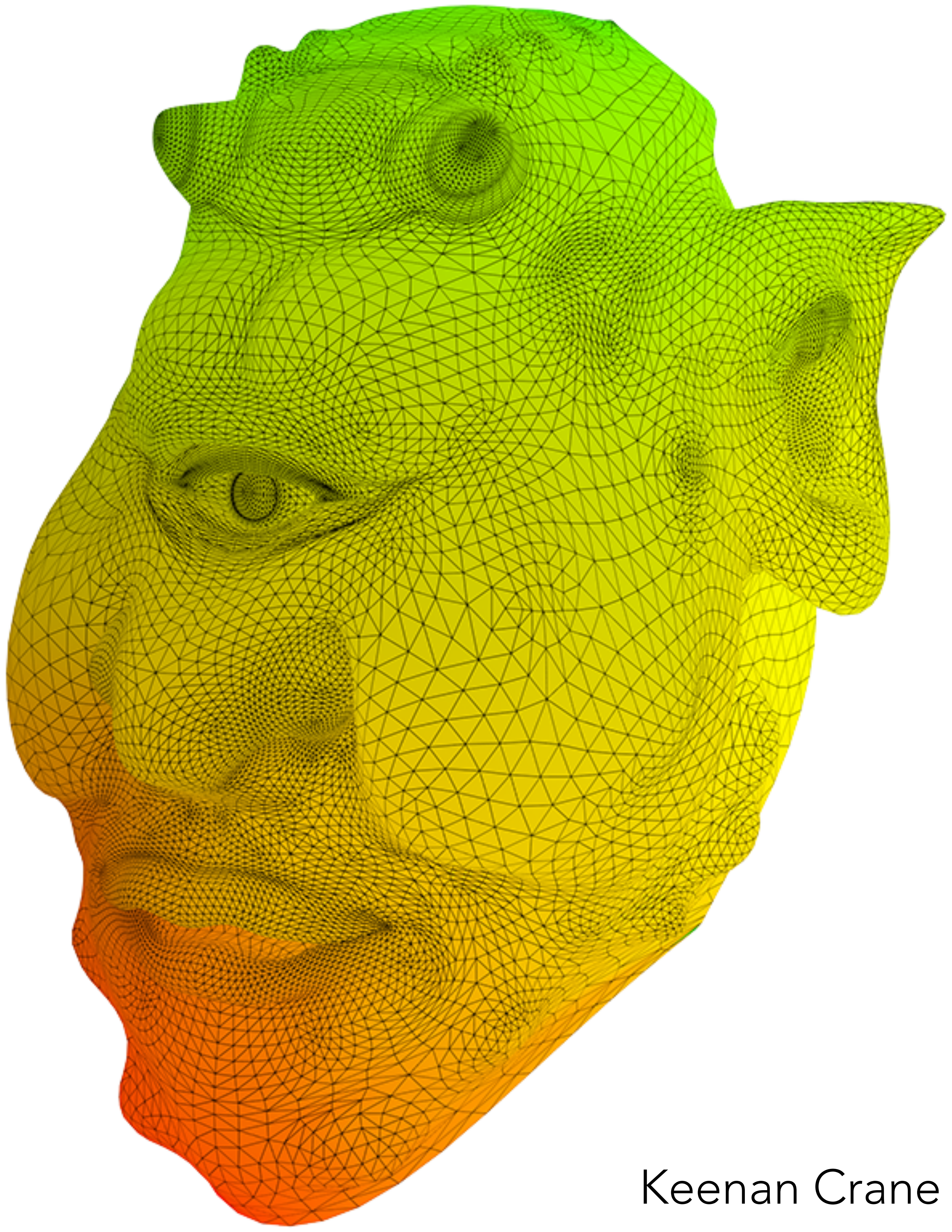
Texture mapping



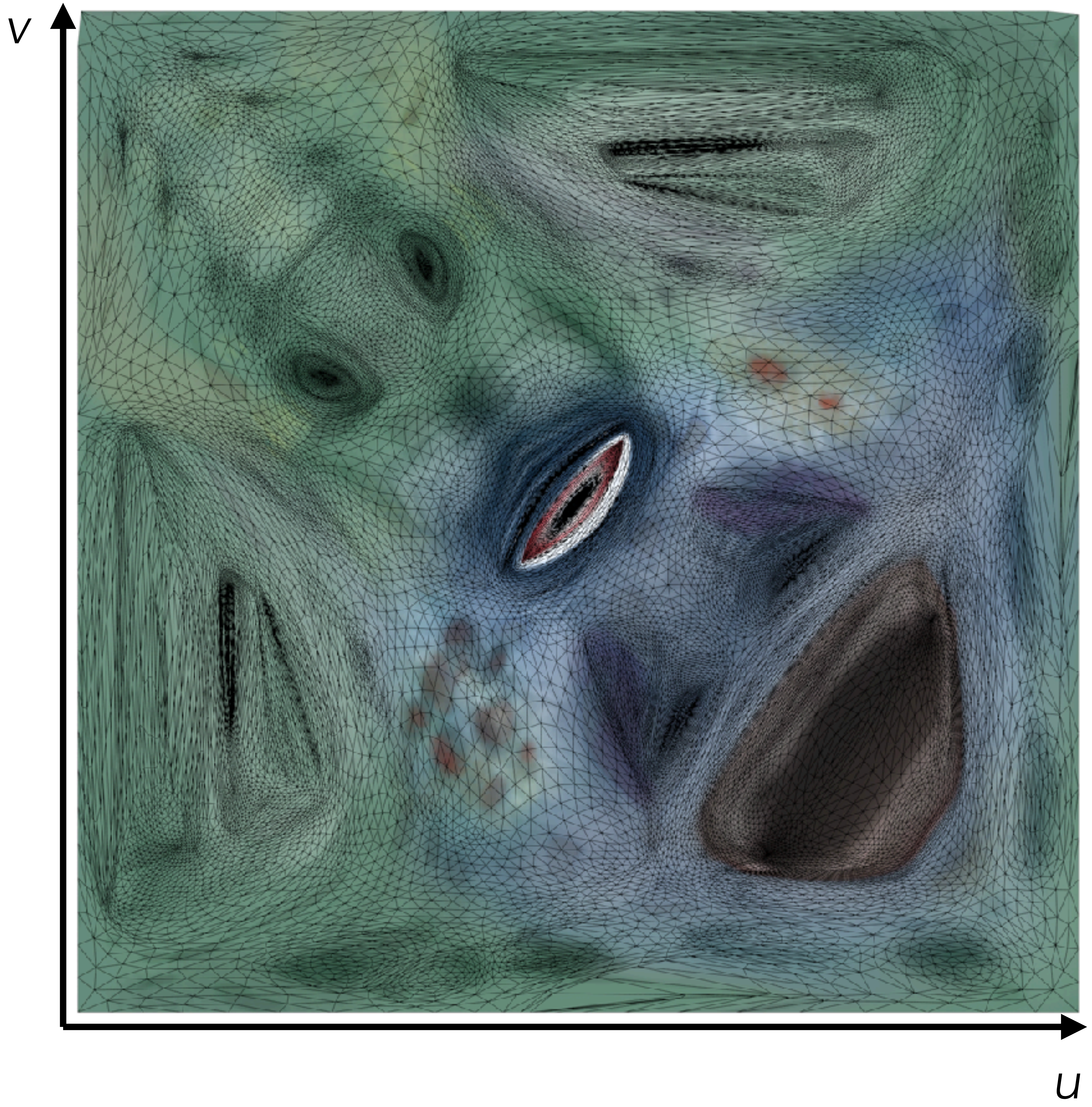


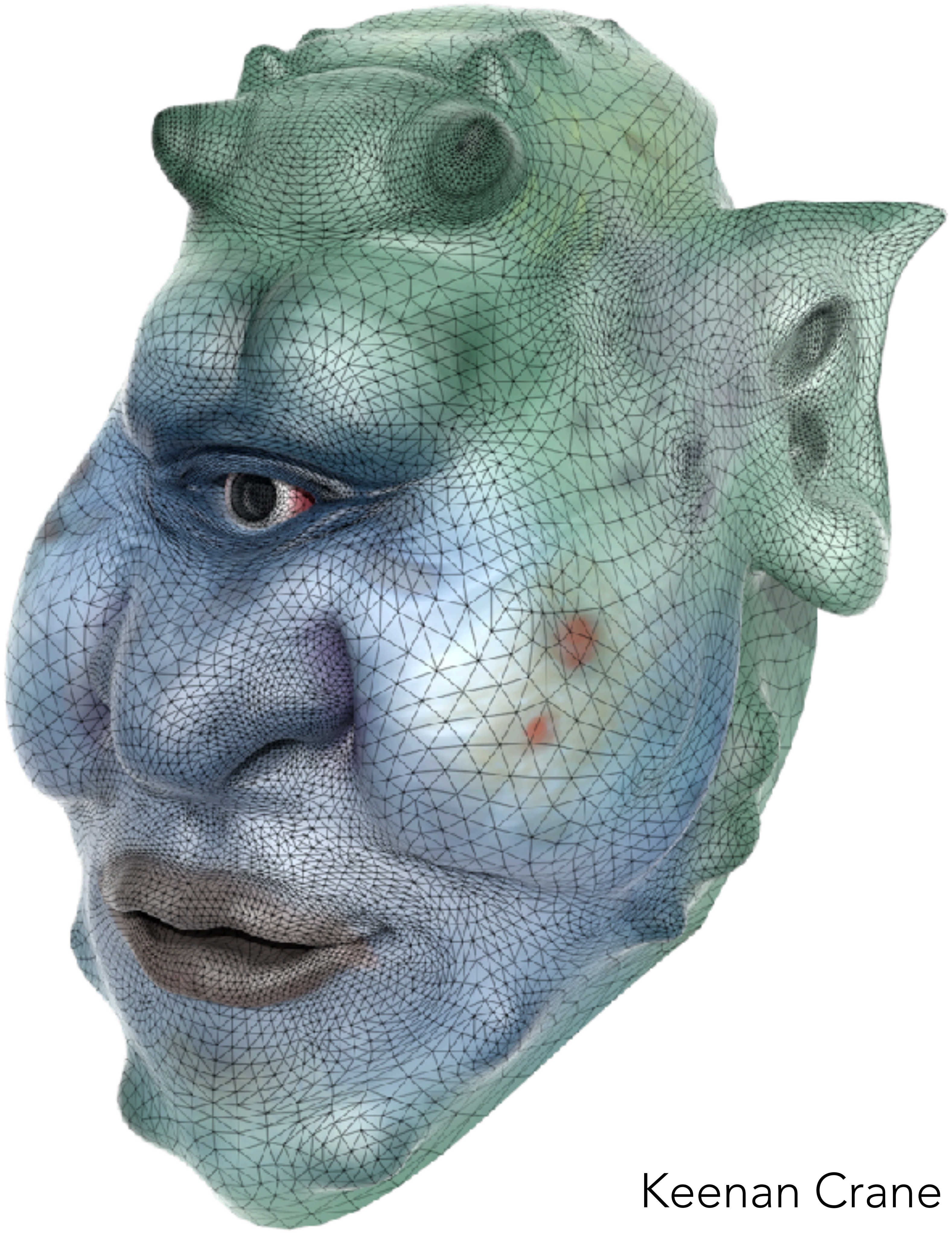
Keenan Crane



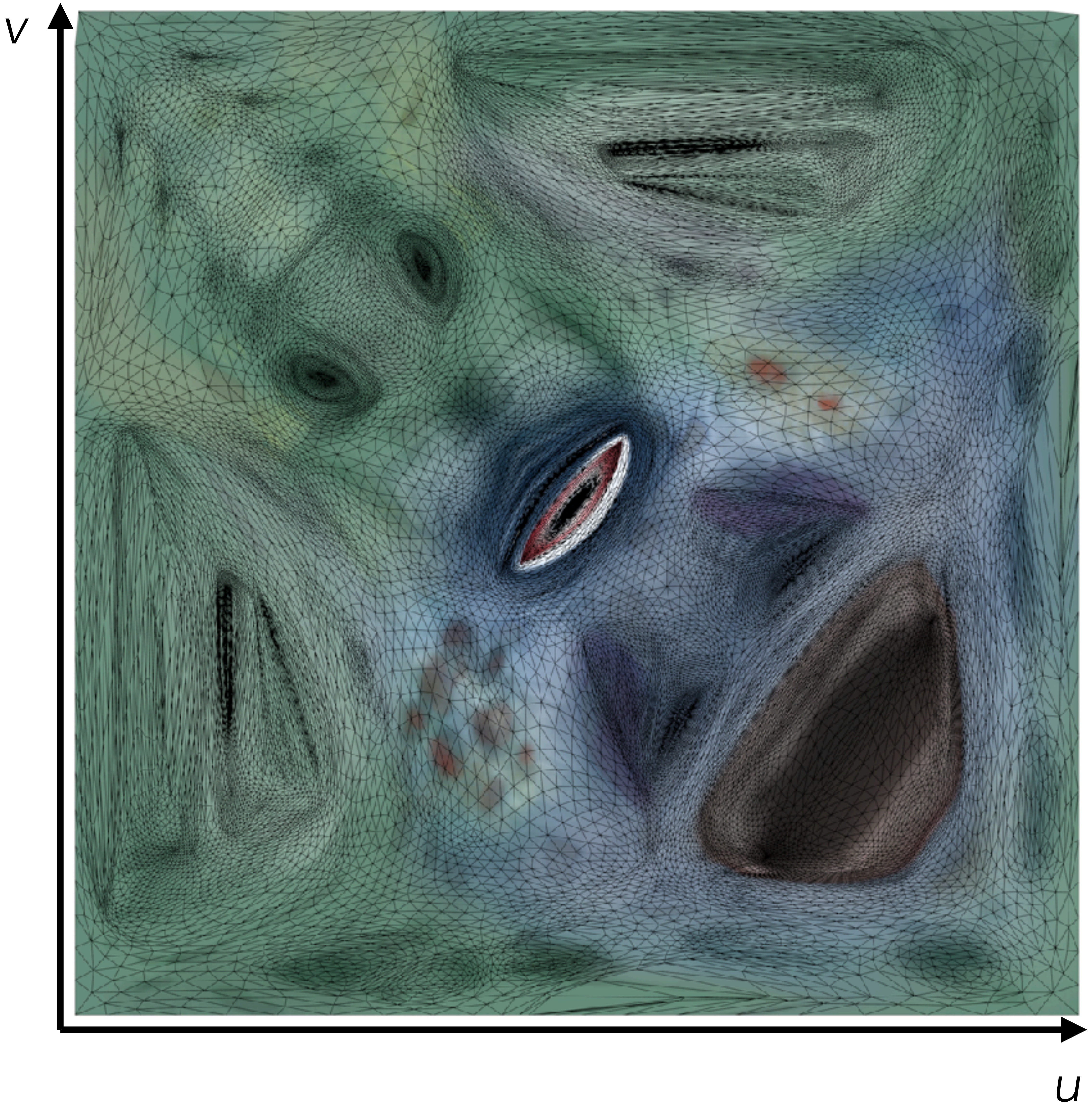


Keenan Crane

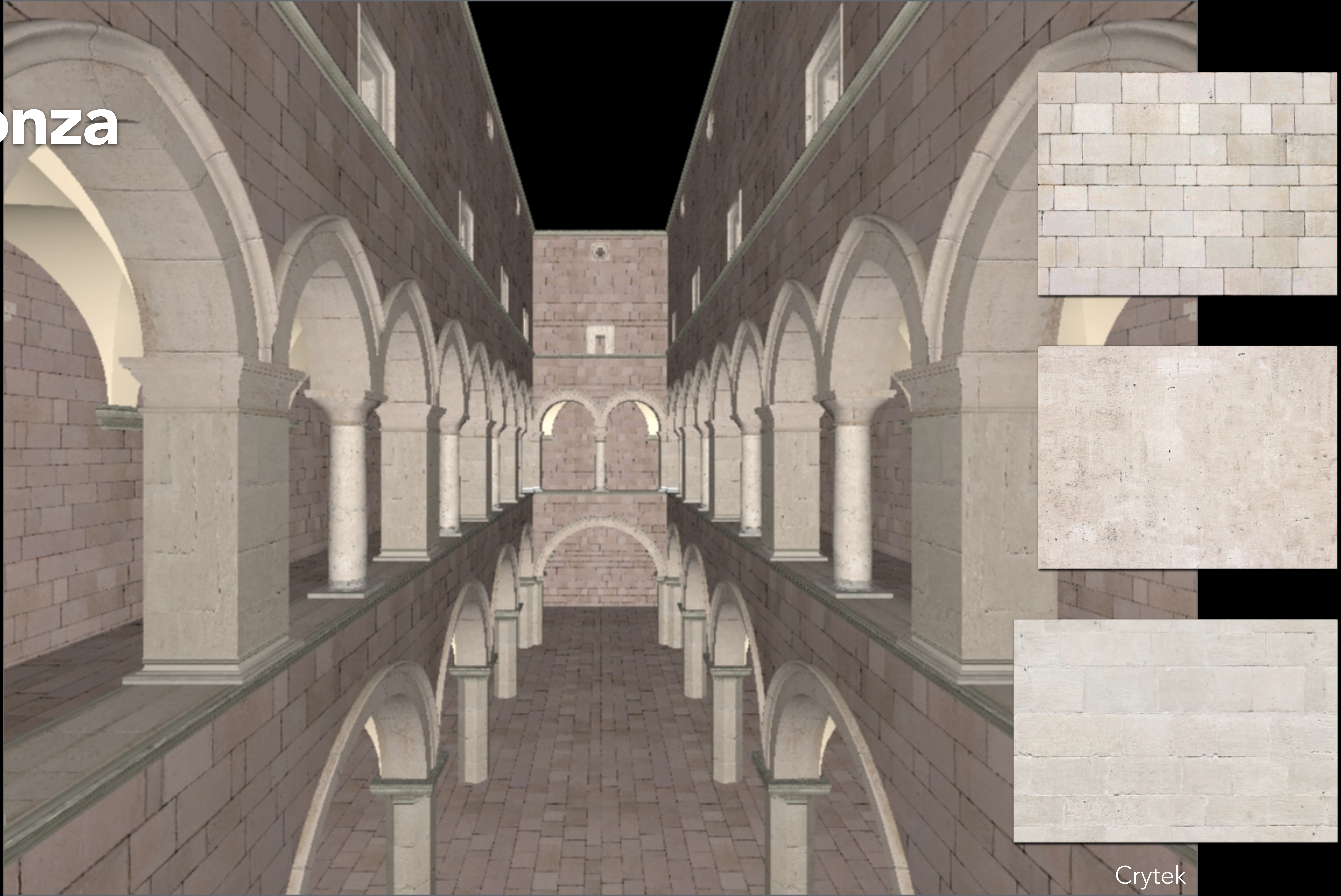




Keenan Crane



Sponza



Crytek

Sponza: texture coordinates

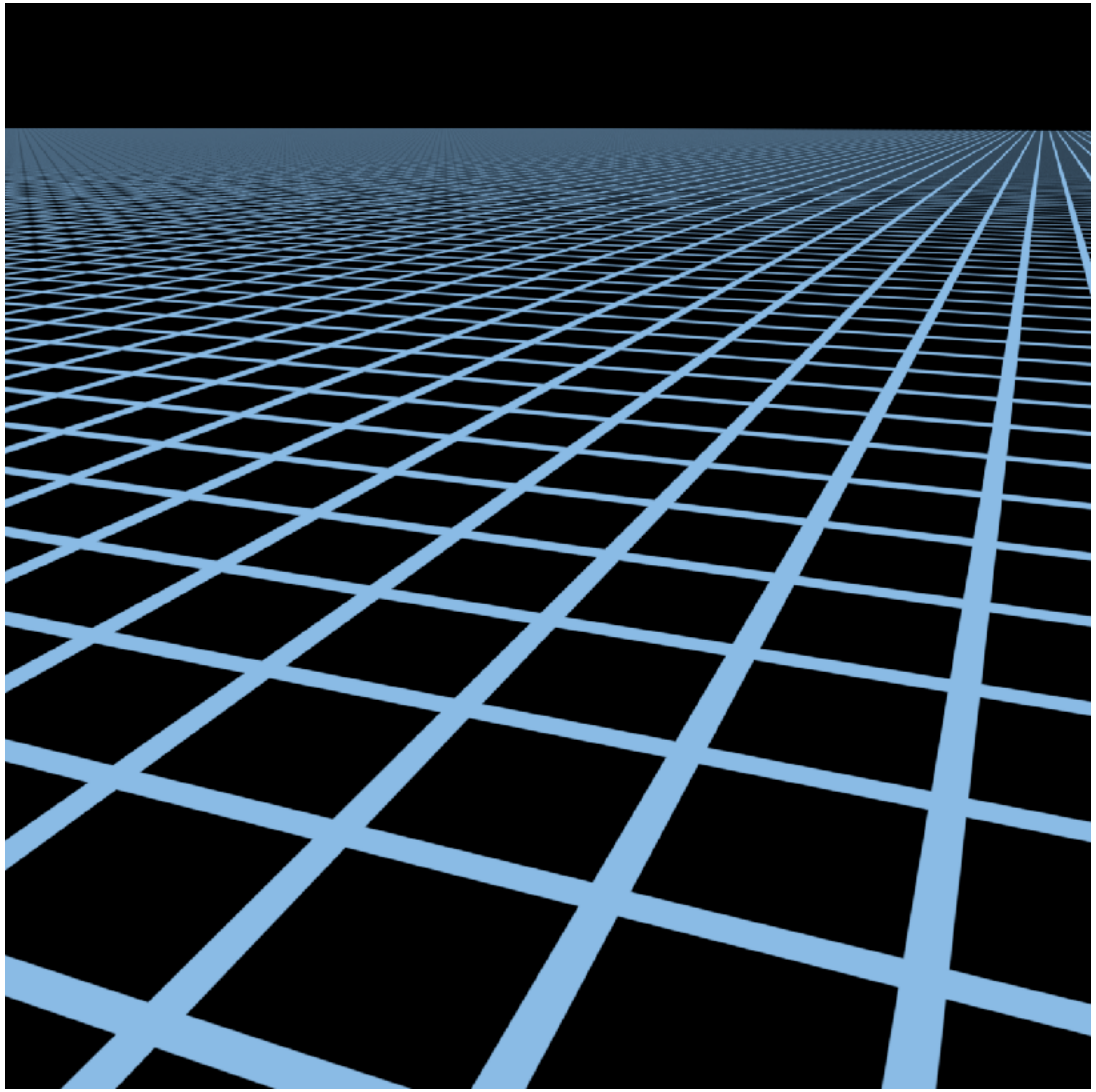
Every point with the same texture coordinates gets the same colour.

Drawing textured triangles

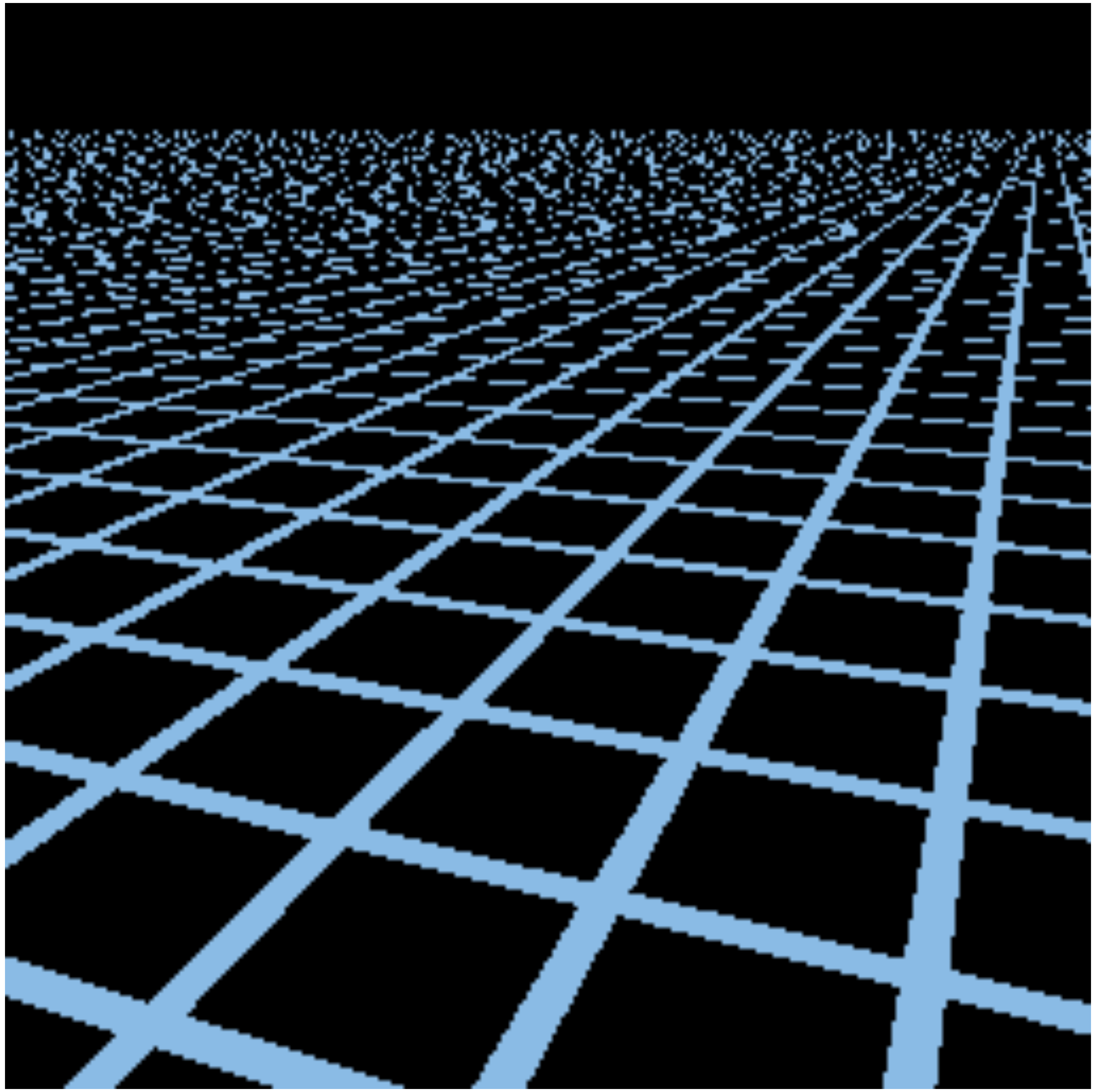
Inputs: (i) mesh with vertex positions (*x*,*y*,*z*) and texture coordinates (*u*,*v*), (ii) texture image

Naïve algorithm:

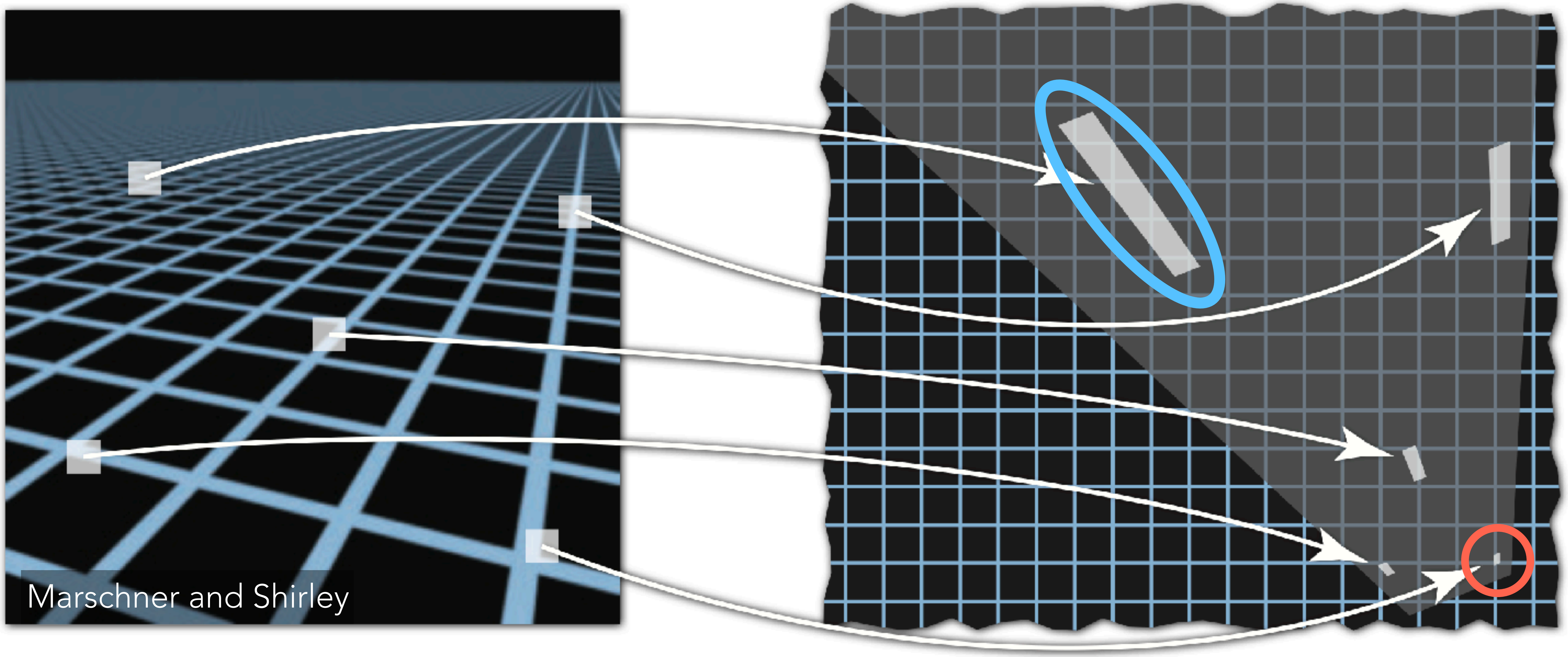
```
for each triangle (i,j,k):
    for each rasterized sample:
        (u,v) = interpolate (ui,vi), (uj,vj), (uk,vk)
        texColor = sample texture at (u,v)
        compute shading with e.g. kd = texcolor
```



High-res reference (1280×1280)

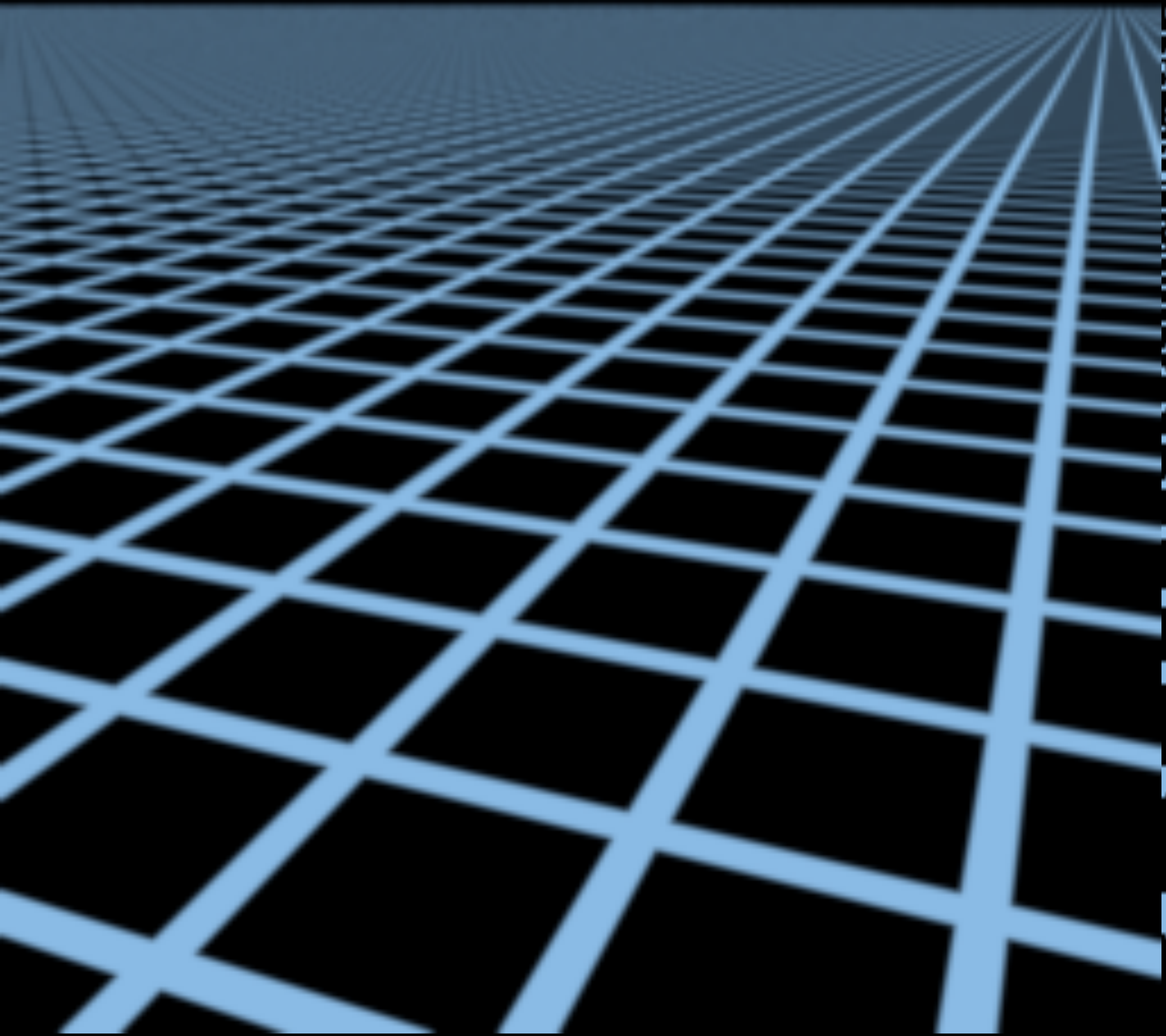


Point sampling (256×256)

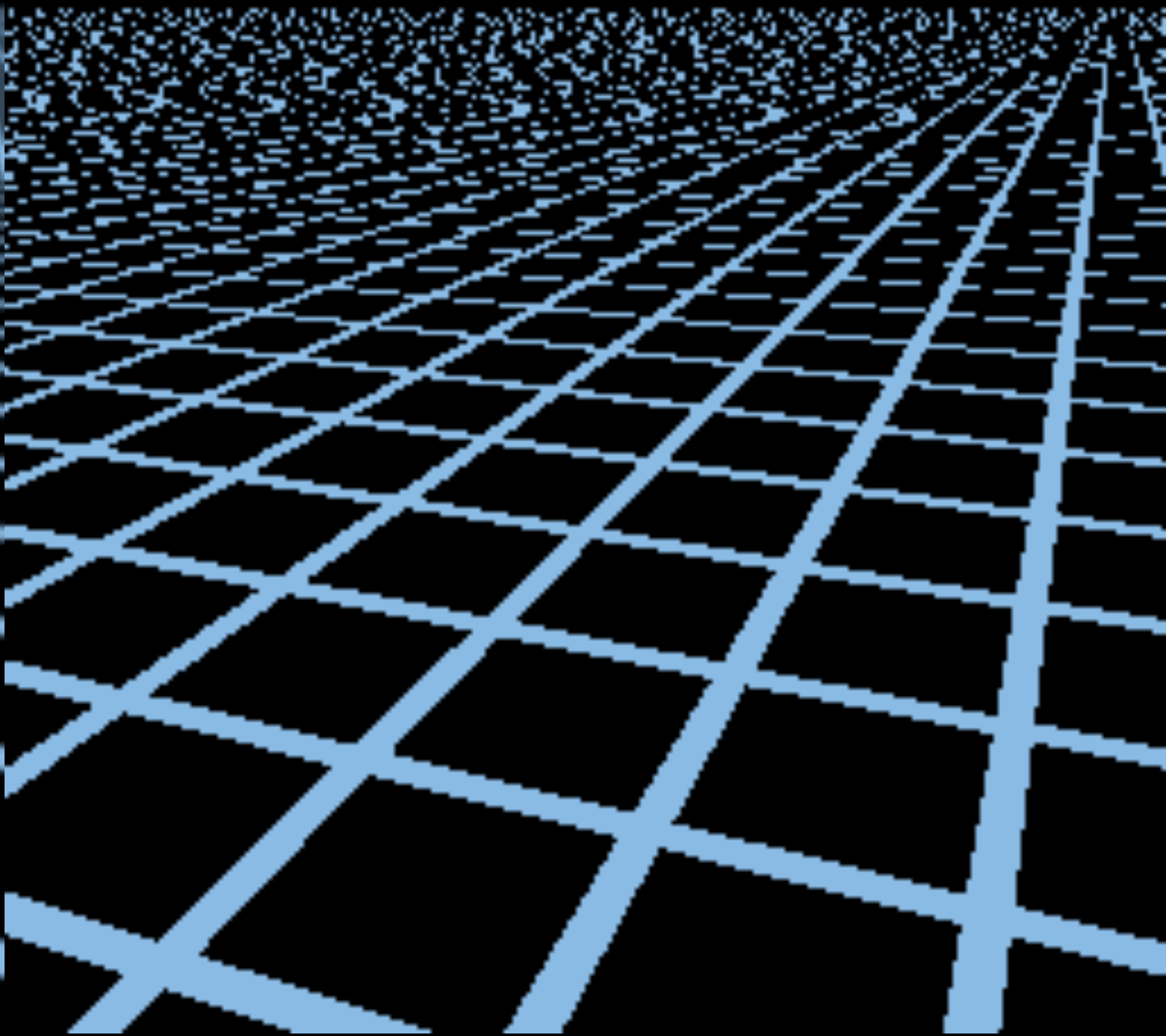


Texture mapping creates a very irregular sampling pattern!

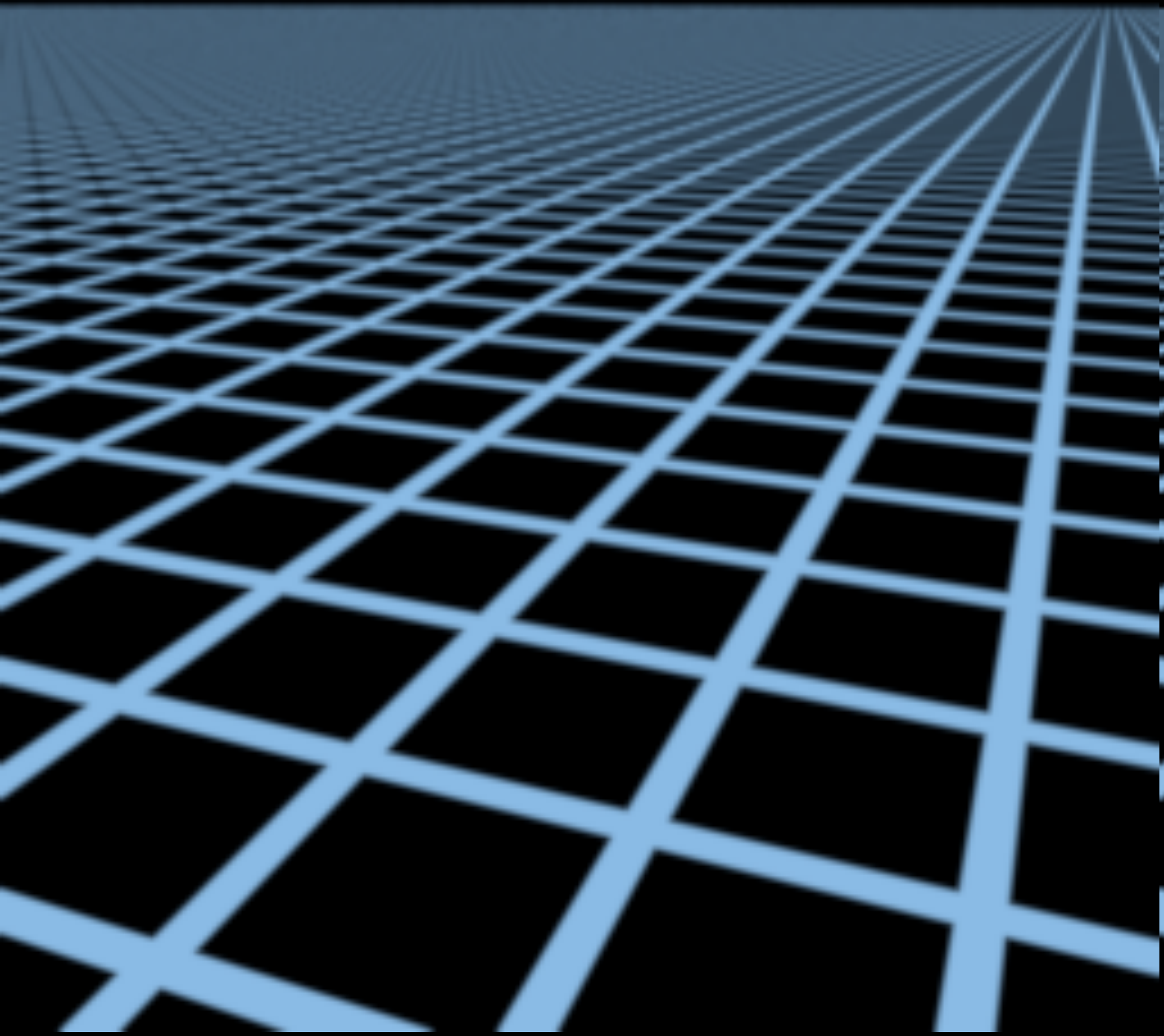
- Some regions are **magnified**: multiple screen samples per texture pixel (**texel**)
- Some regions are "**minified**": multiple texels per sample



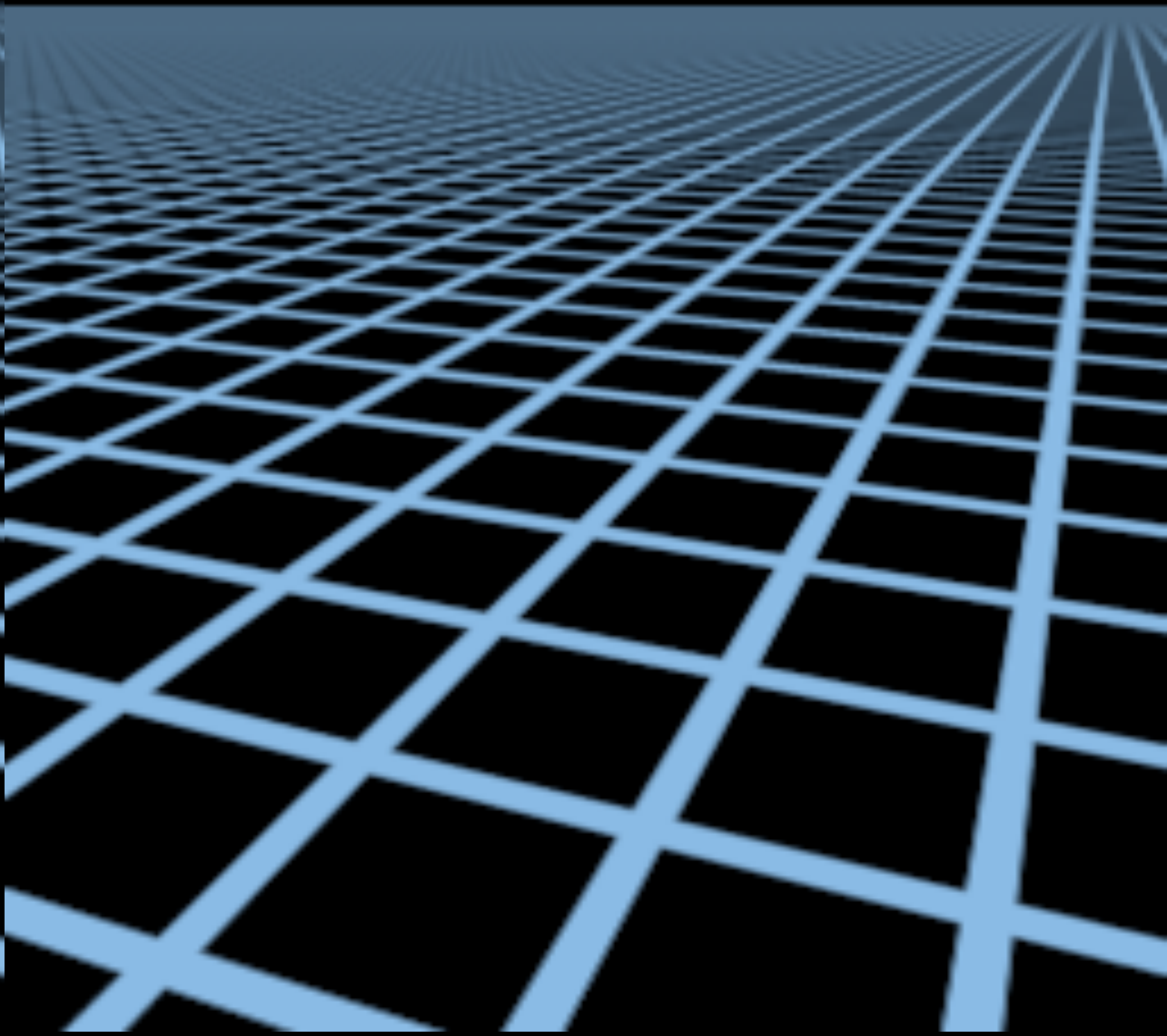
Supersampled reference (256×256, 512 spp)



Point sampling (256×256)



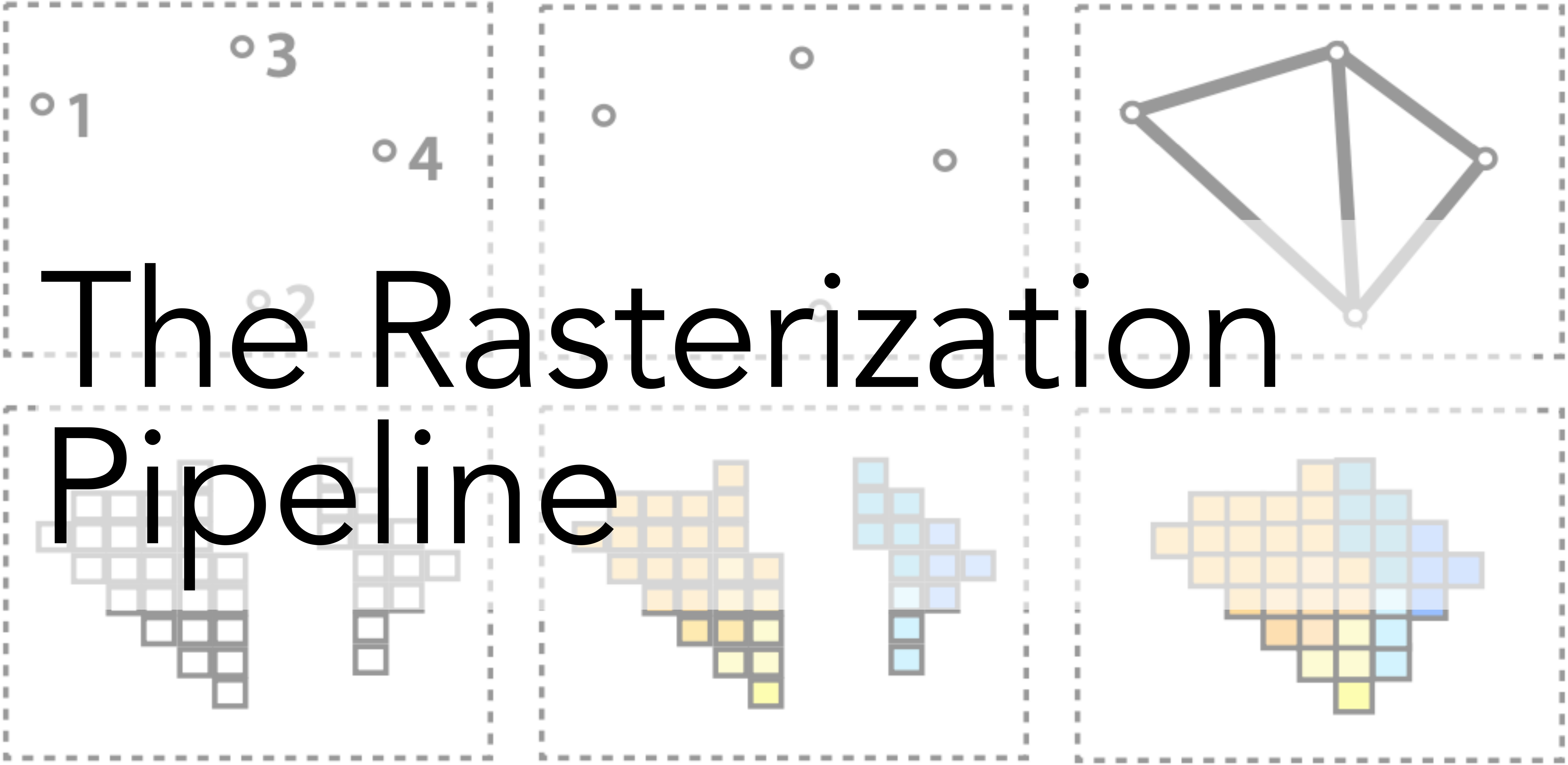
Supersampled reference (256×256, 512 spp)



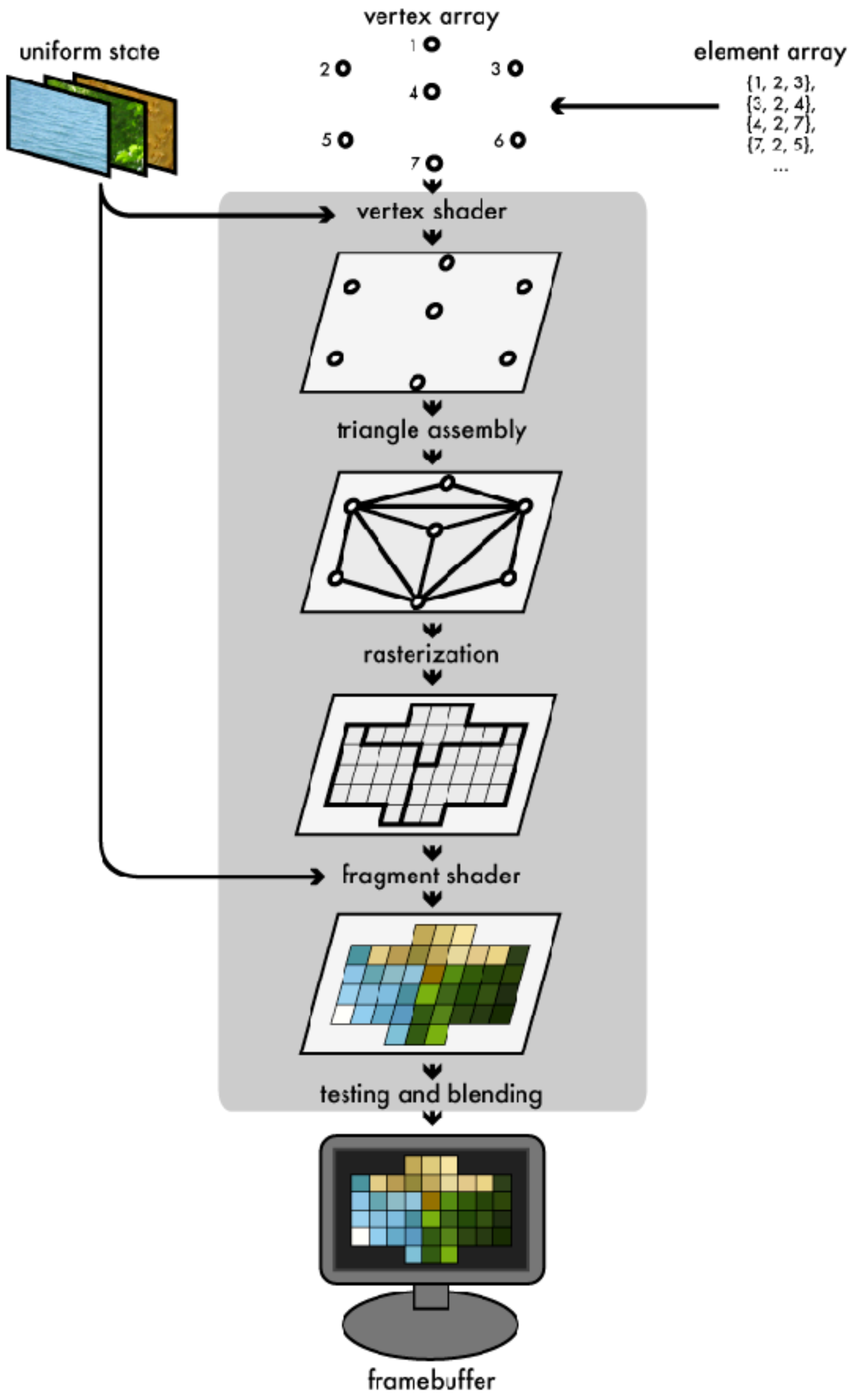
Elliptical weighted average (EWA)

15 minute break

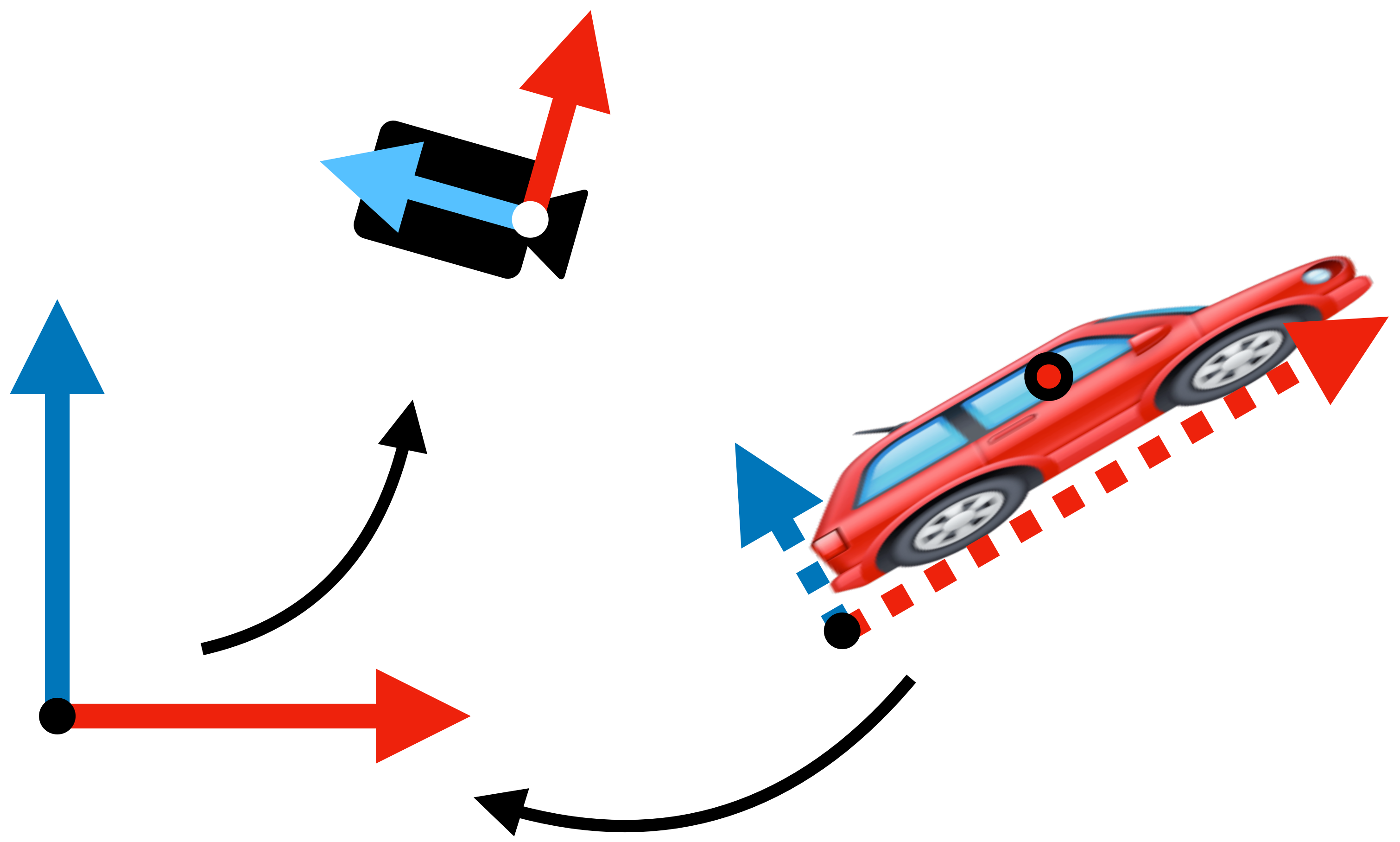
The Rasterization Pipeline



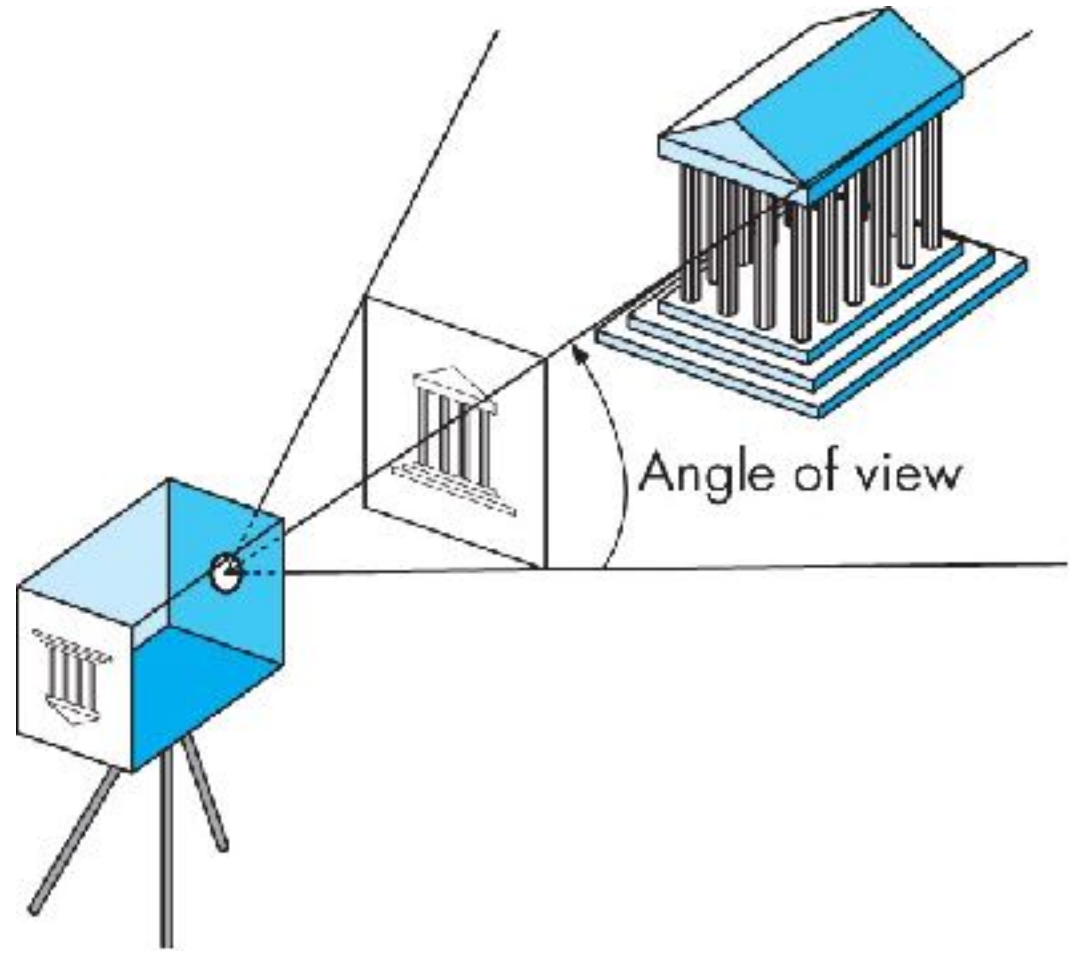
Putting it all together



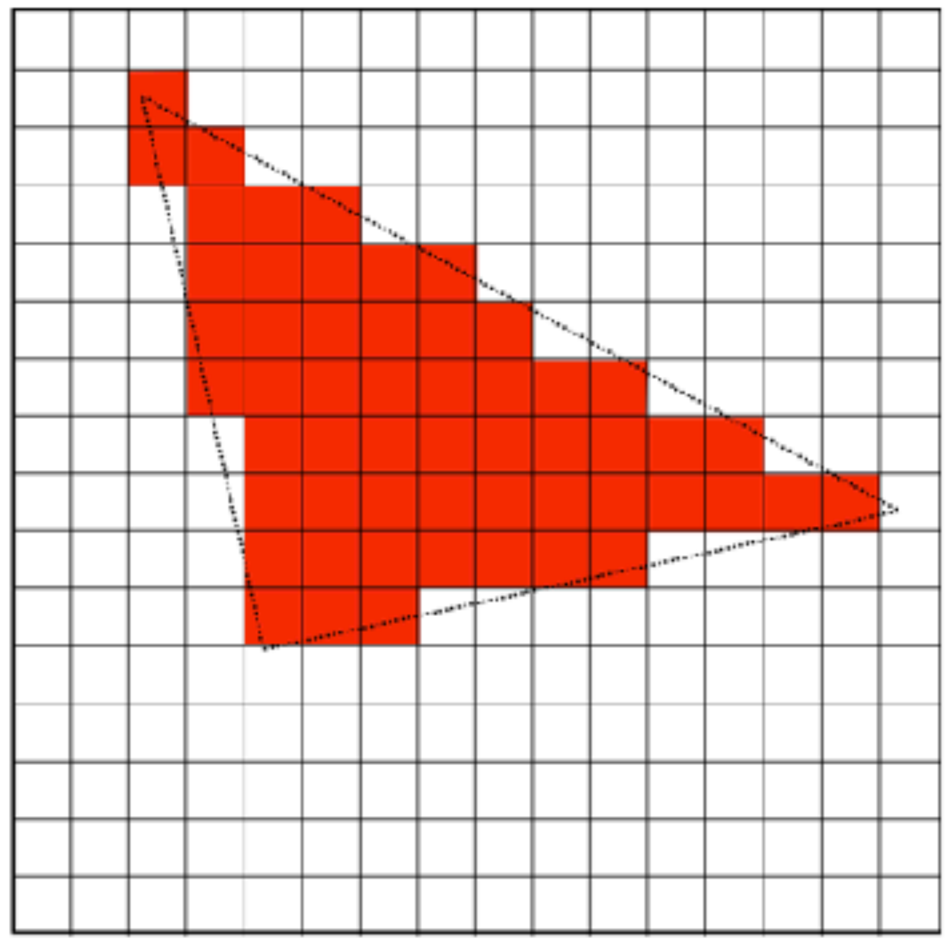
Joe Groff, duriansoftware.com



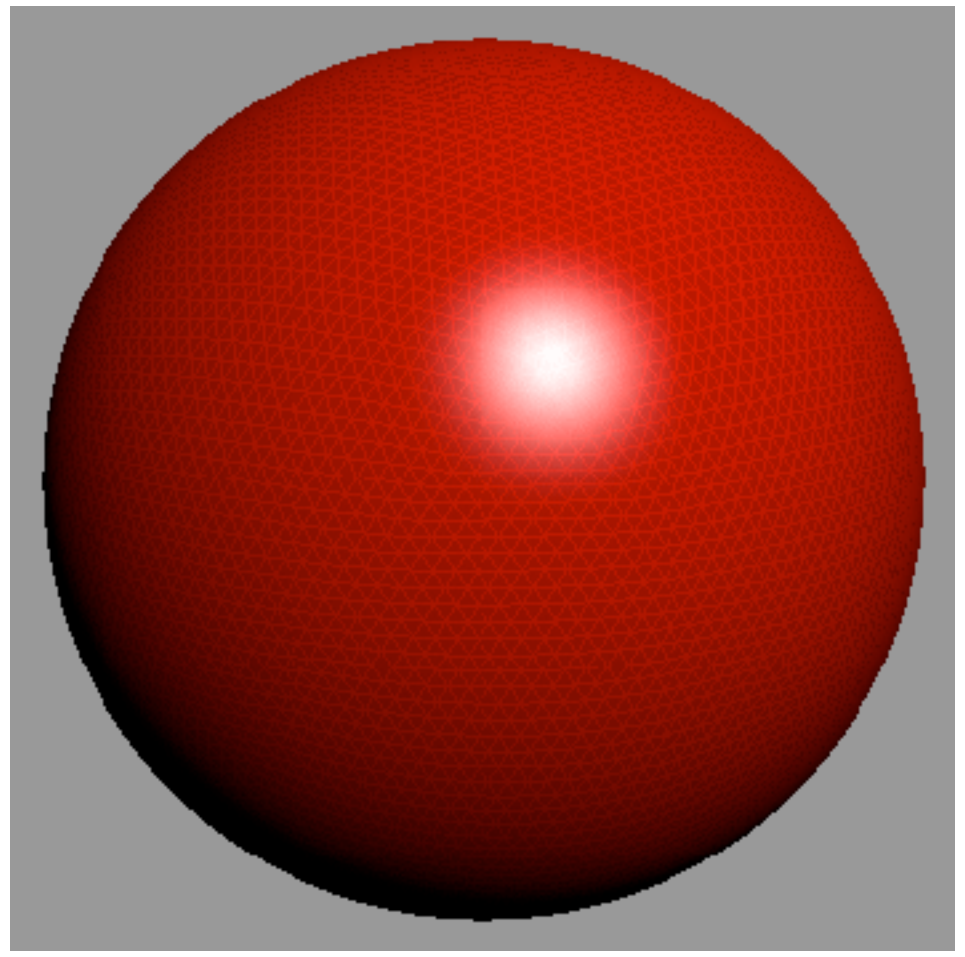
Transformations



Projection



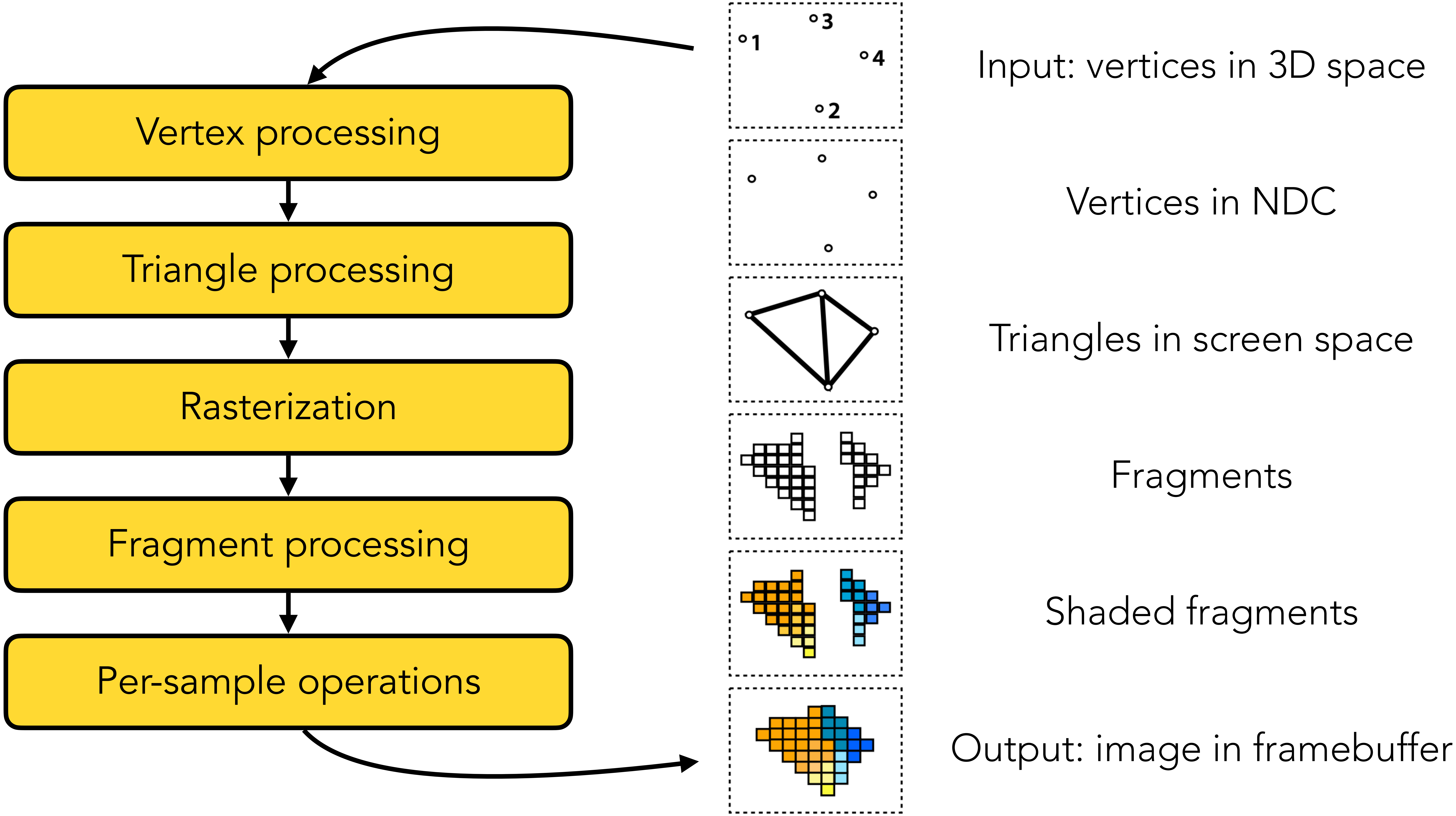
Rasterization



Shading

Texture mapping

Visibility


Vertex processing
Triangle processing
Rasterization
Fragment processing
Per-sample operations
1
2
3
4
Input: vertices in 3D space
Vertices in NDC
Triangles in screen space
Fragments
Shaded fragments
Output: image in framebuffer

Inputs to the pipeline

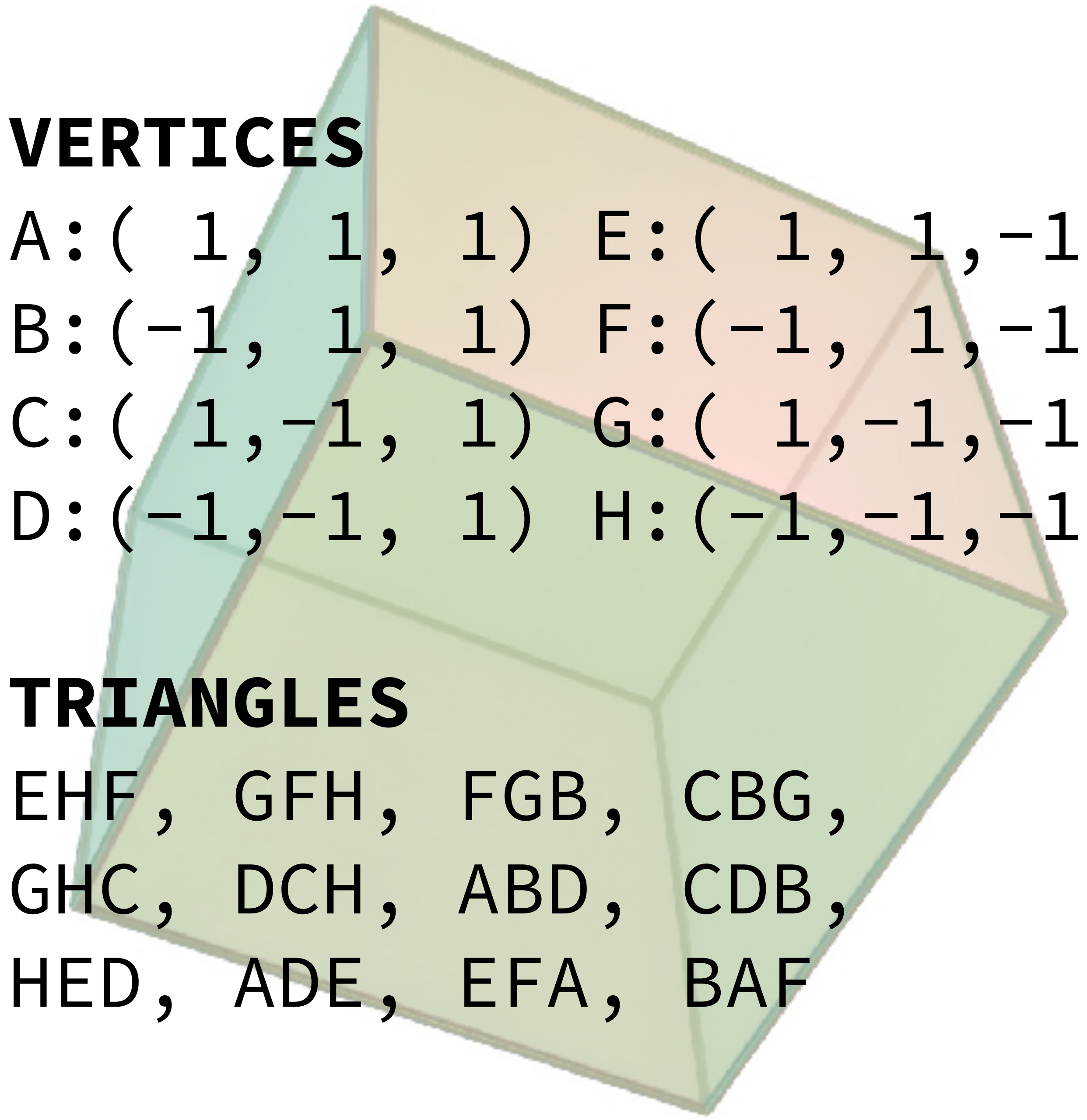
For each object, we have two streams:

- Vertices with various **attributes** (position, colour, texture coordinates, etc.)
- Indices of triangles (or other primitives)

Why? Each vertex is shared between many primitives (on average ~6 triangles!)

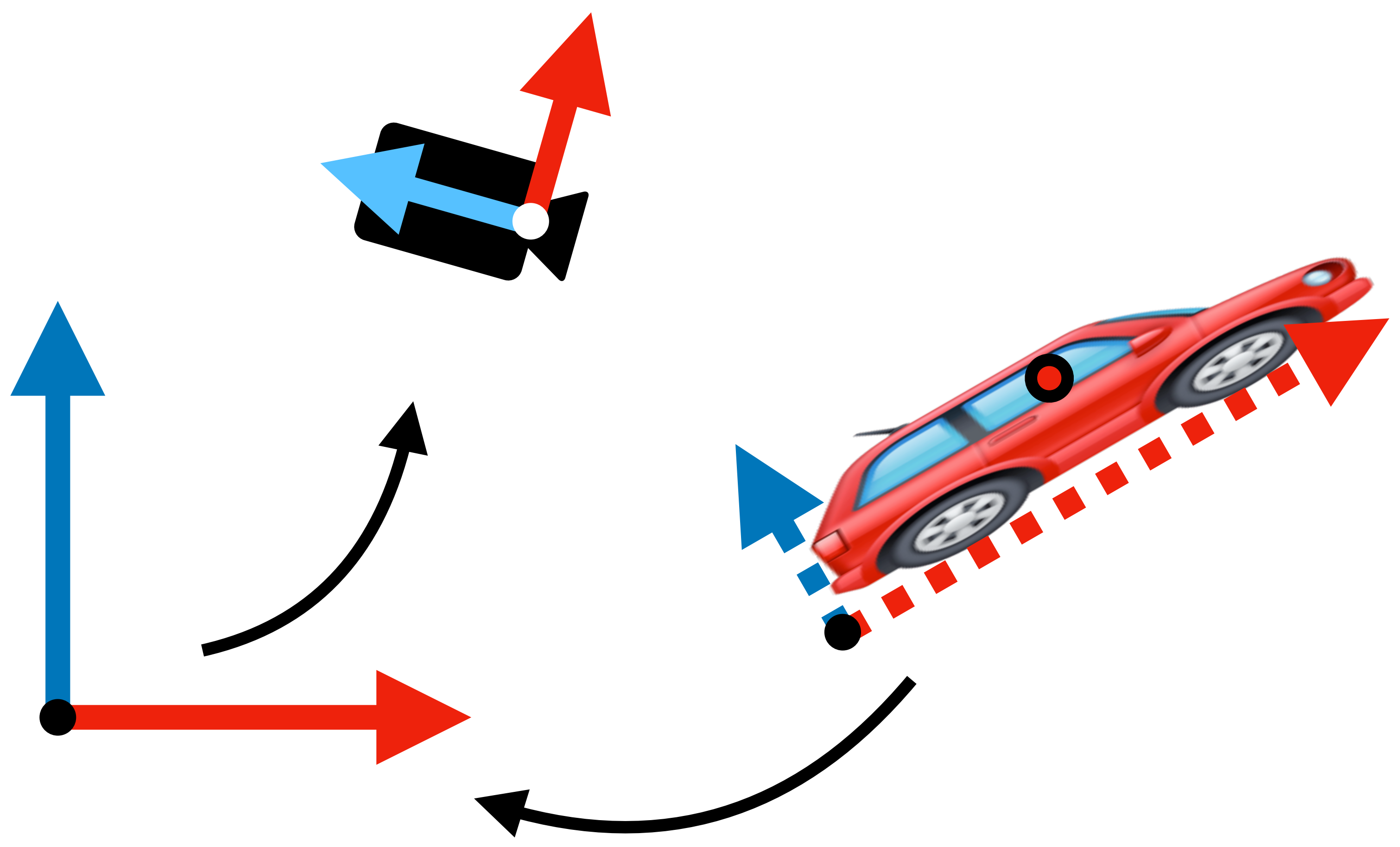
We also have **uniform** data, common to all vertices/triangles of an object:

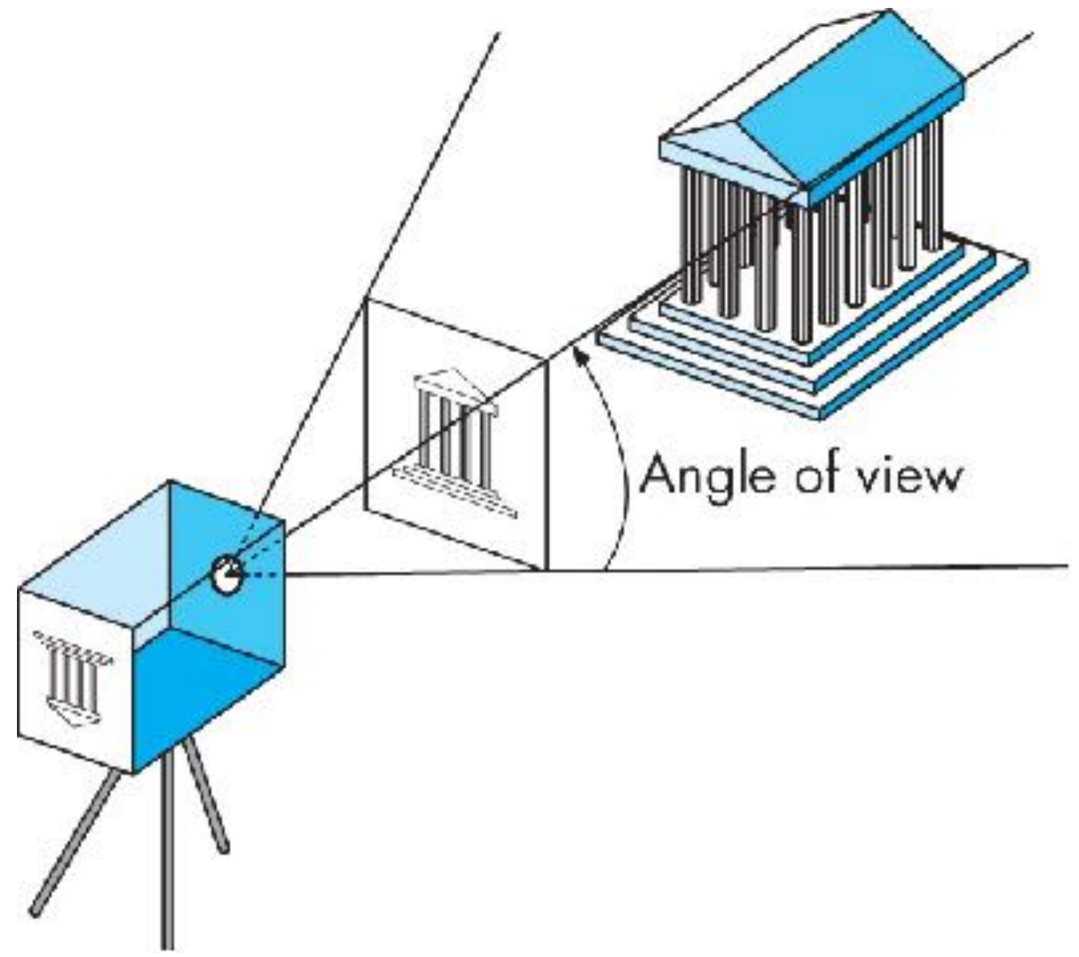
- Transformation matrices, texture images, etc.



```
VERTICES
A:( 1, 1, 1) E:( 1, 1,-1)
B:(-1, 1, 1) F:(-1, 1,-1)
C:( 1,-1, 1) G:( 1,-1,-1)
D:(-1,-1, 1) H:(-1,-1,-1)

TRIANGLES
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

Vertex processing





Every vertex is subject to the same operations:

- Modelling transformation: object space → world space
- Viewing transformation: world space → camera space
- Projection transformation: camera space → normalized device coordinates

This stage is programmable, done by programmer-specified **vertex shader**

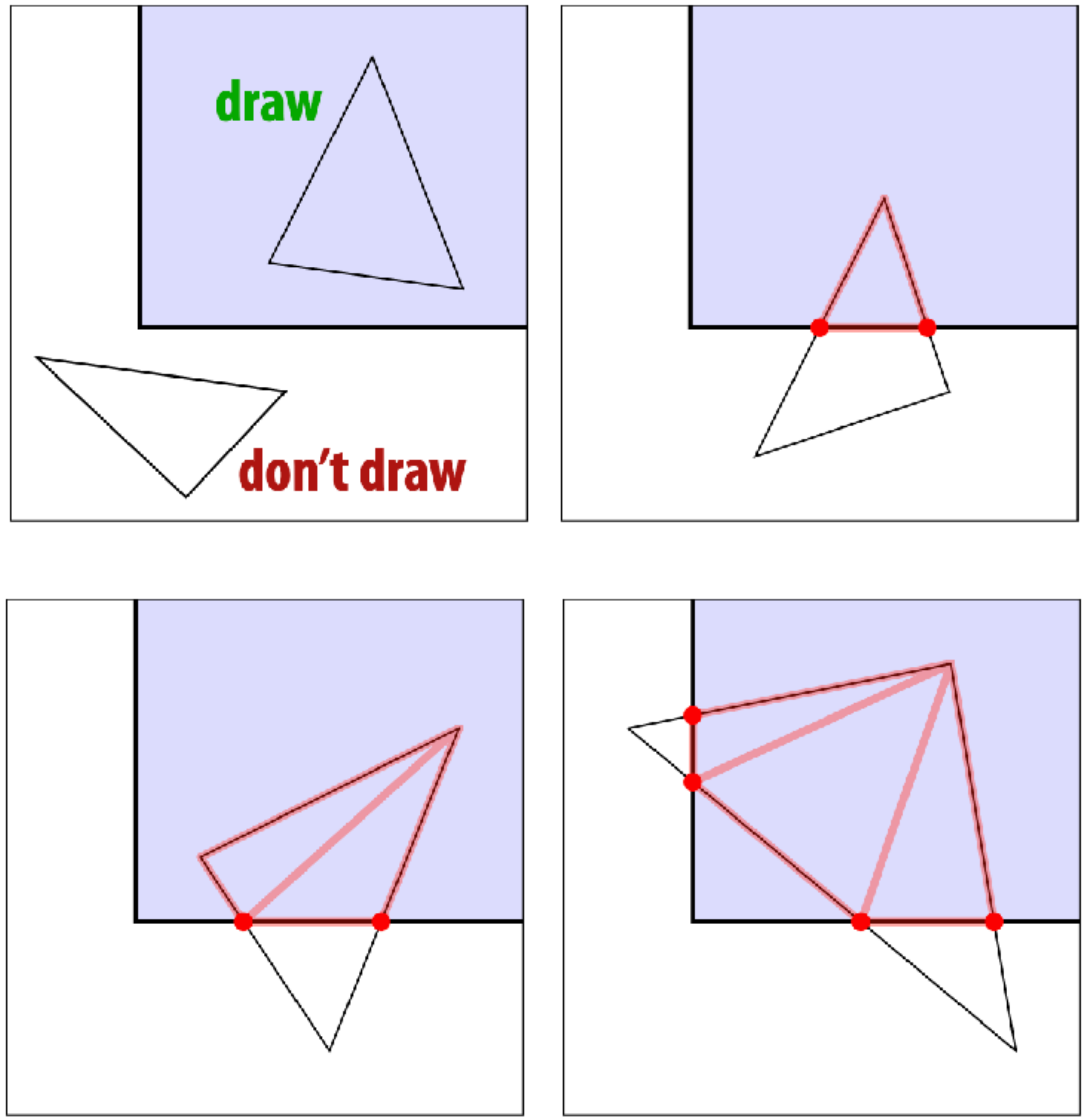
Output: transformed position in NDC (before division)

Triangle processing

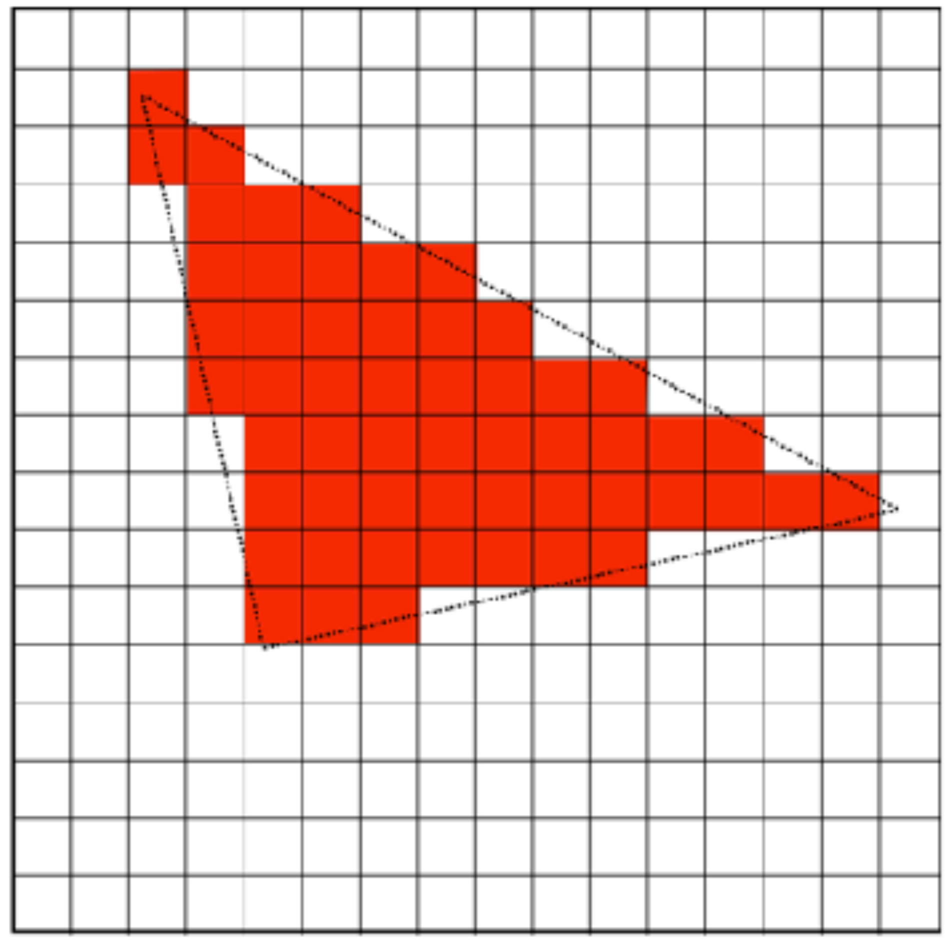
For each triangle (i, j, k):

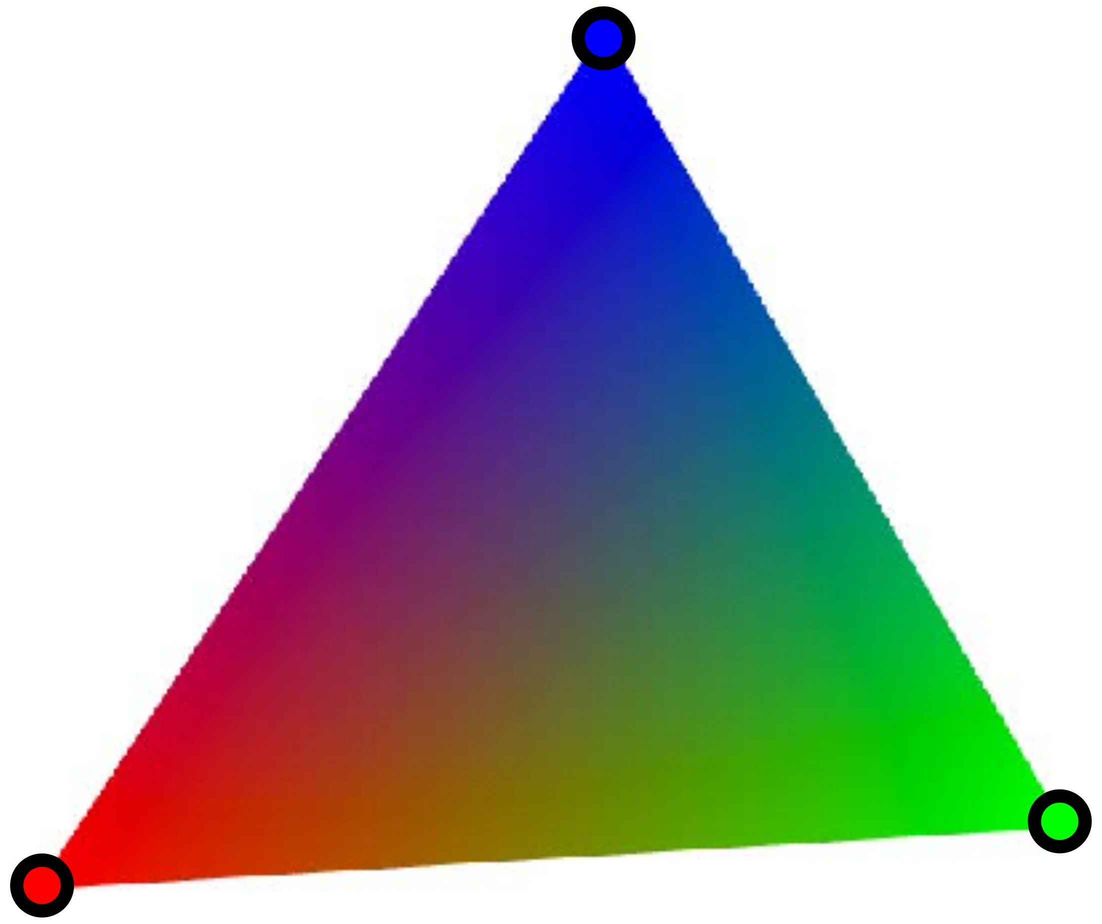
- Get transformed positions $\mathbf{p}_i$, $\mathbf{p}_j$, $\mathbf{p}_k$ of corresponding vertices
- Clip against the canonical view volume $[-1, 1]^3$
- Divide by w and transform to pixel coordinates

Output: clipped triangle(s) in screen space



Rasterization





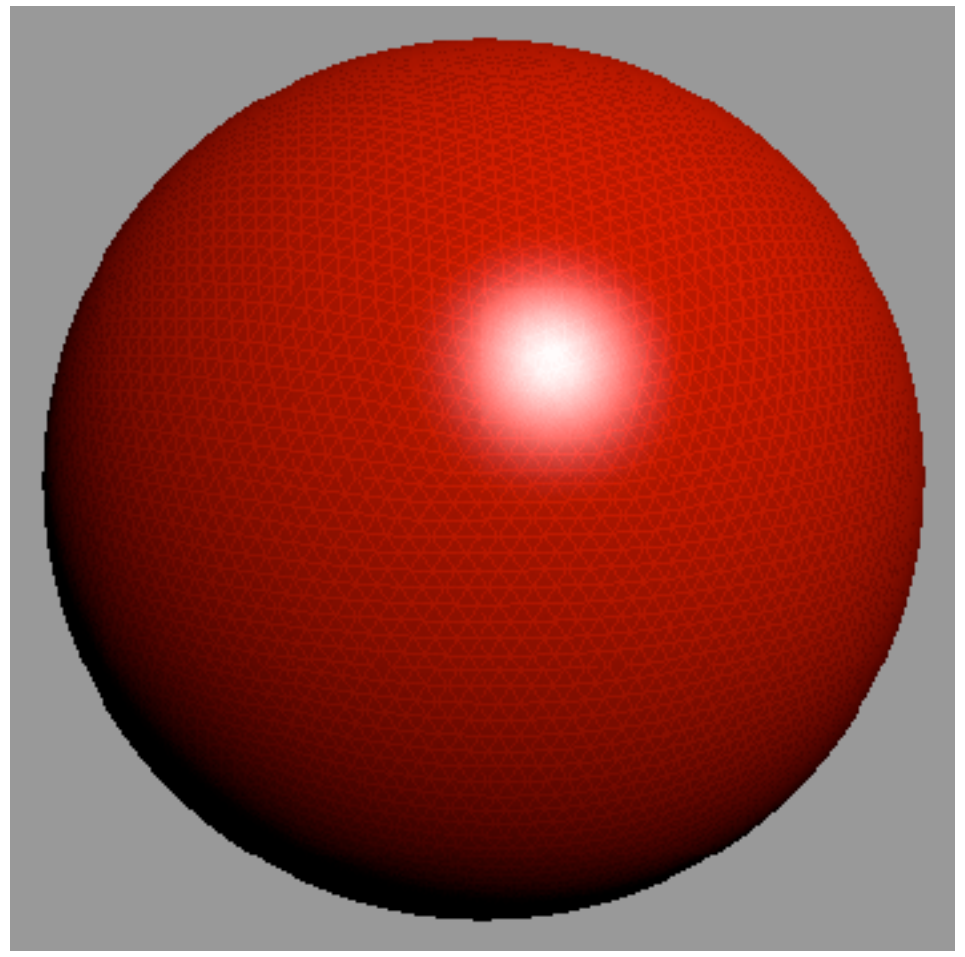
For each triangle, we will produce a set of sample points that it covers.

But also: interpolate the vertex attributes (colour, texture coordinates, etc.) to each covered sample.

We will need these in the next stage!

Output: stream of **fragments**, i.e. sample-sized pieces of triangle with interpolated attributes

Fragment processing



We may want to do some computation to decide the colour of a fragment, e.g.

- Texture lookup
- Lighting computation

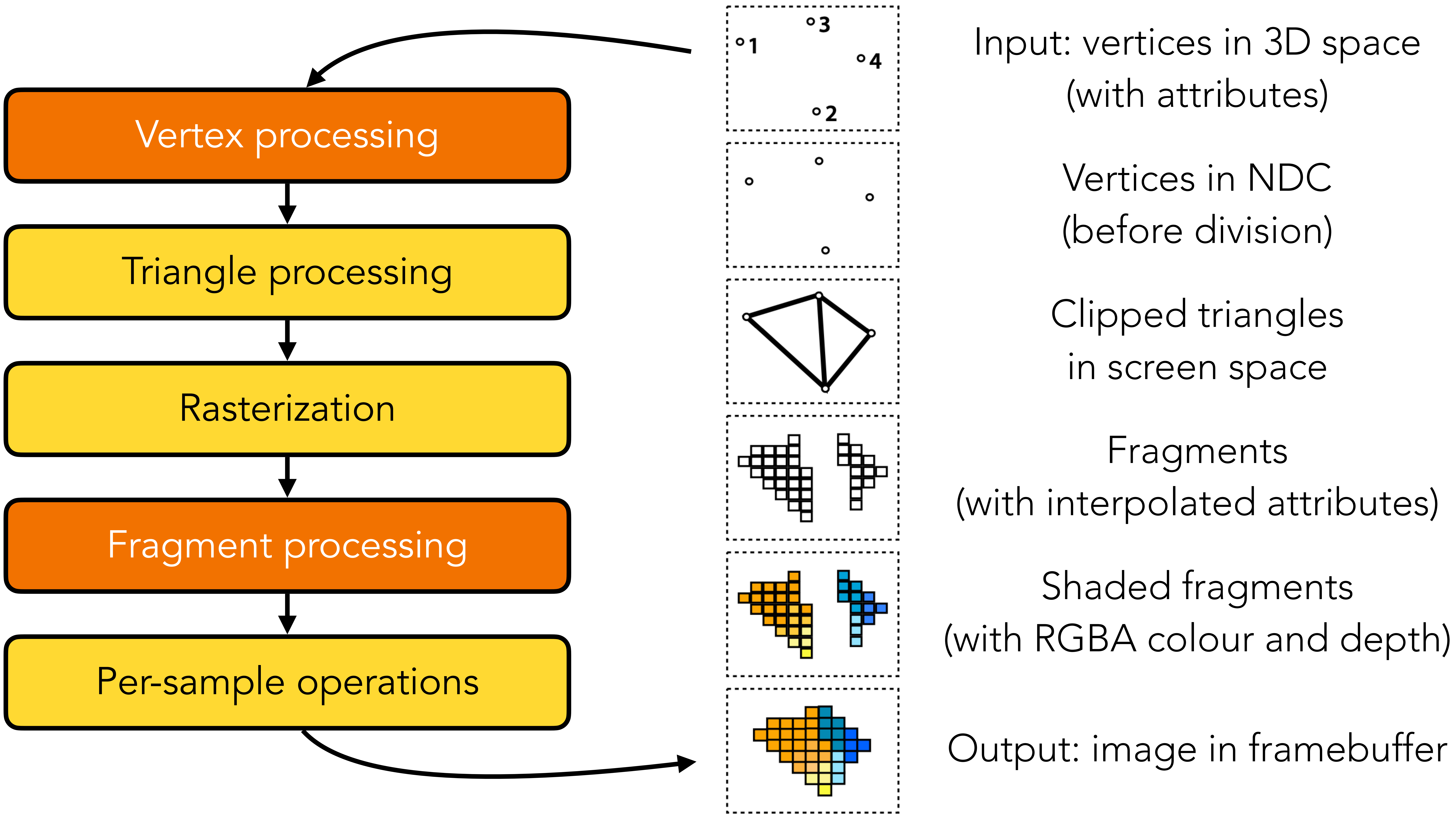
This stage is also programmable: **fragment shader**

Output: fragment colour as a 4-tuple: red, green, blue, alpha (opacity)

Per-sample operations

- Test each sample’s depth vs. z-buffer
- Write its colour to the framebuffer (optionally blending with existing colour if alpha < 1)

Once all this is done for all objects in the scene, the framebuffer contains the final rendered image.


Vertex processing
Triangle processing
Rasterization
Fragment processing
Per-sample operations
1
2
3
4
Input: vertices in 3D space
(with attributes)
Vertices in NDC
(before division)
Clipped triangles
in screen space
Fragments
(with interpolated attributes)
Shaded fragments
(with RGBA colour and depth)
Output: image in framebuffer

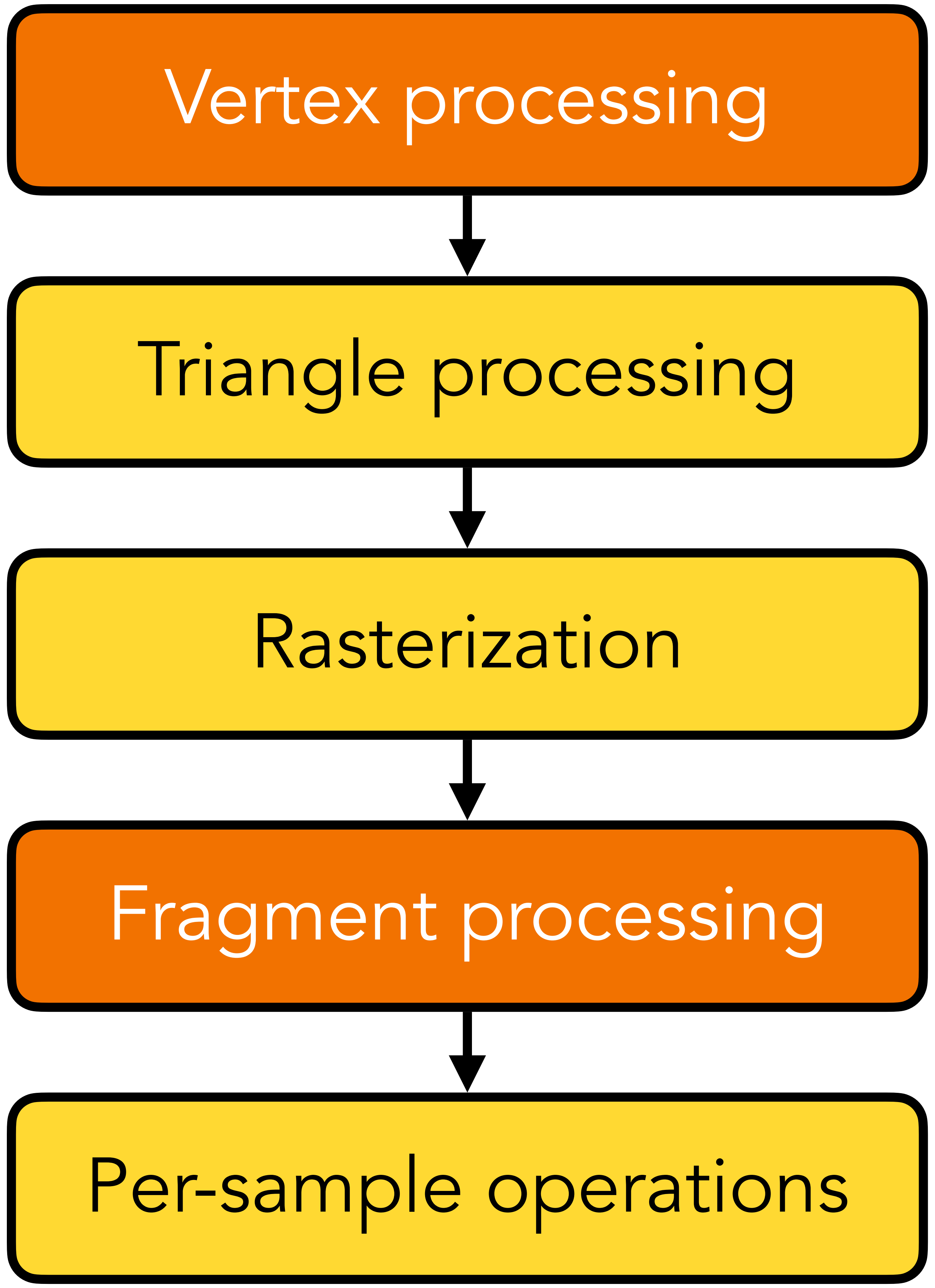
Programmer's view

Initialization:

- Compile vertex and fragment shaders
- Send uniform variables (transformation matrices, texture images, etc.) to GPU
- For each object: send vertex attributes, triangle indices

Per frame, for each object:

- Update uniform variables
- Request draw



The vertex shader typically applies modelling, viewing, projection transformations to compute the NDC position…

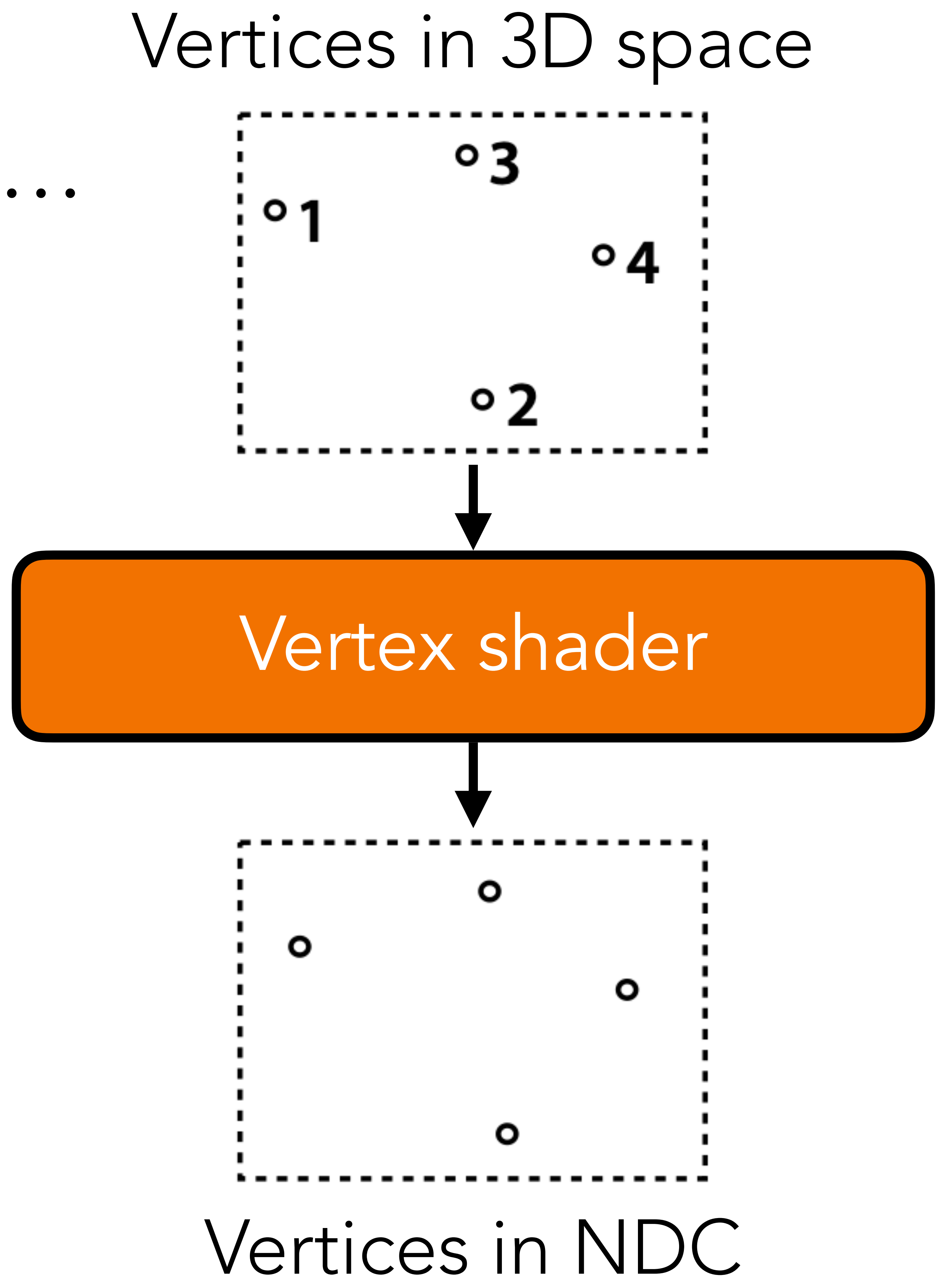
But actually, it is an arbitrary function that can do whatever you want to compute the NDC position!

Runs on each vertex **independently**

- Can't pass information to other vertices
- Can't have side-effects (e.g. no writing to global memory, no print statements)

Inputs: attributes of current vertex, uniform variables

Outputs: vertex position in NDC, other attributes to interpolate to fragments



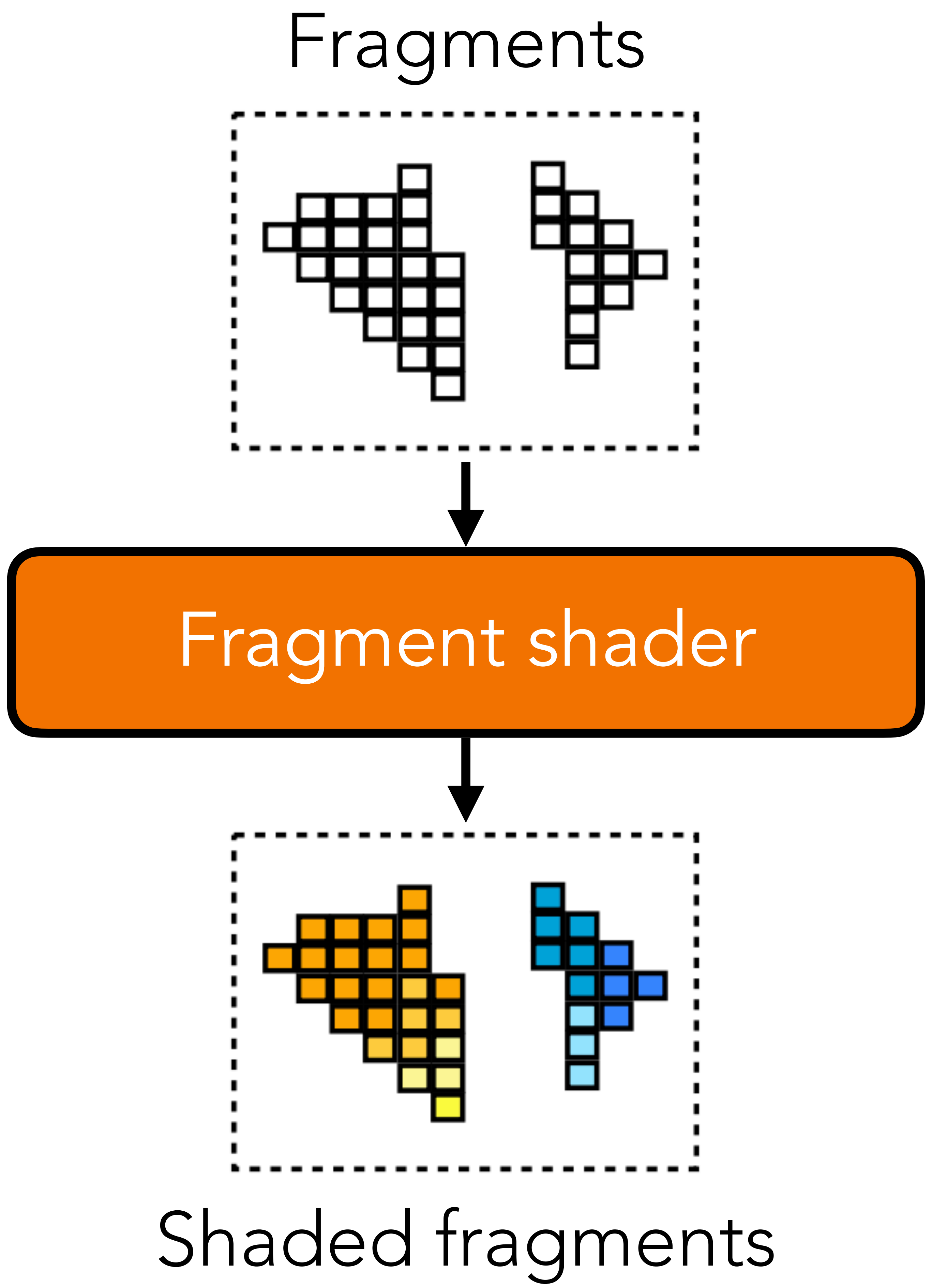
The fragment shader is another arbitrary function.

It can do anything (e.g. texture lookup, lighting computation, etc.) to compute the fragment colour.

Again, runs on each fragment independently

Inputs: attributes interpolated from vertex shader output, uniform variables

Outputs: fragment colour (RGBA), optional: modified fragment depth



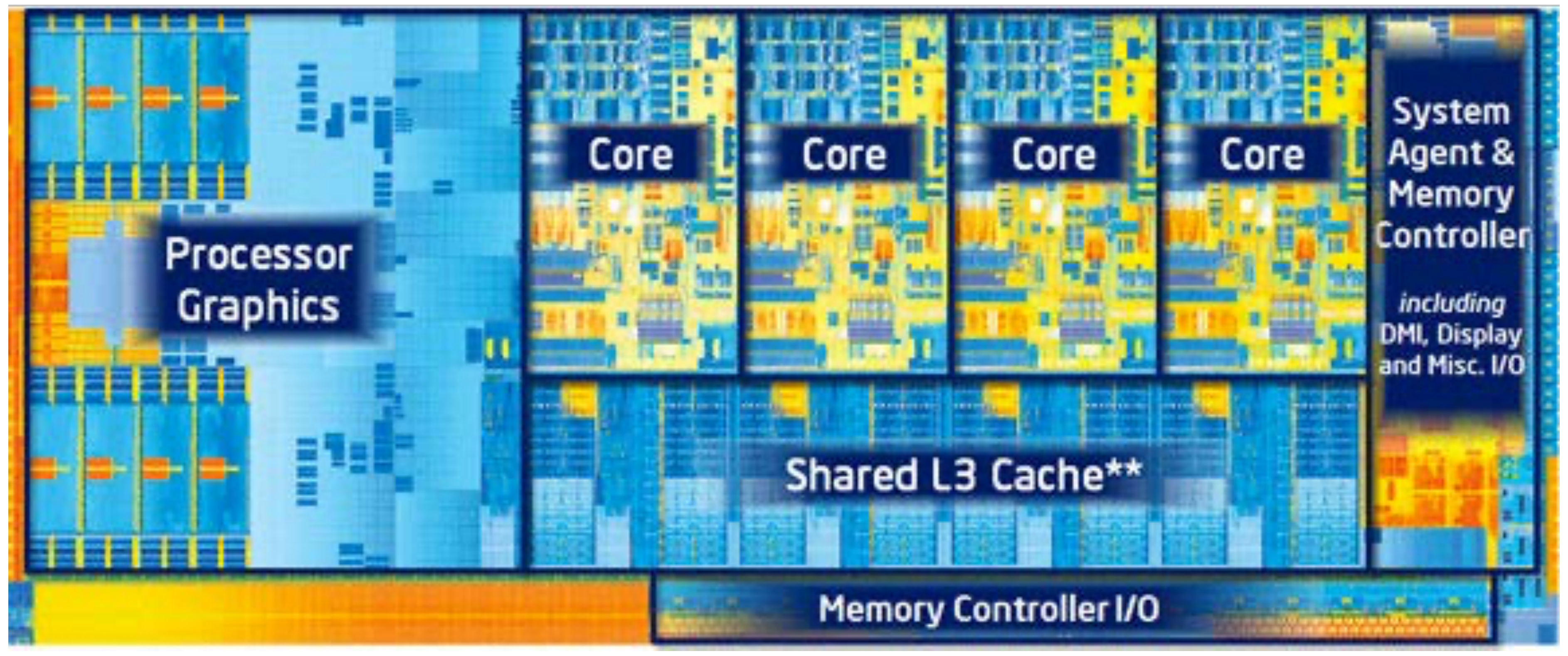
Fragment shader can change fragment depth but not fragment position!

GPUs

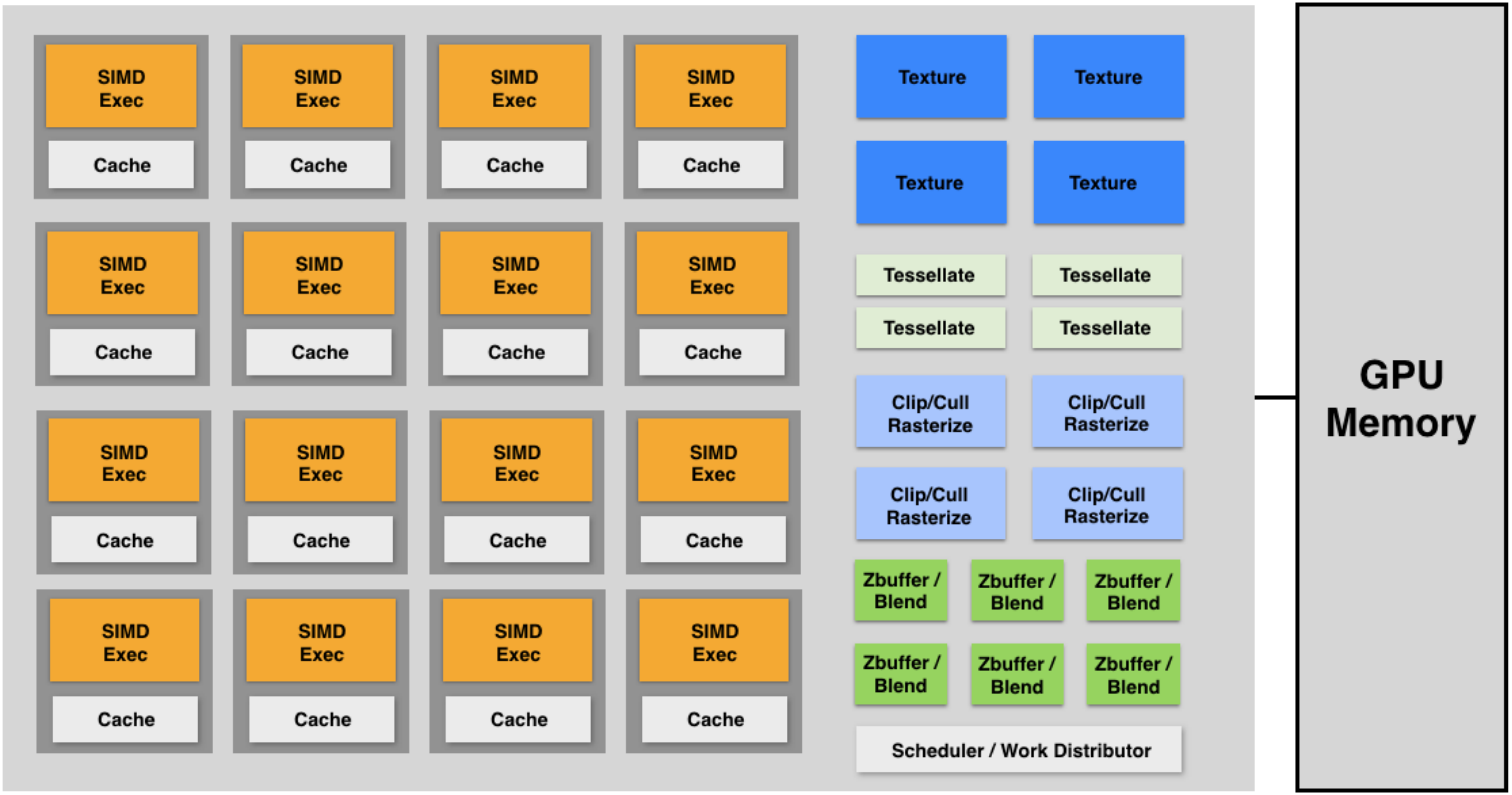
Modern graphics processing units (GPUs) provide a highly parallelized implementation of the rasterization pipeline

- Many SIMD cores for running vertex and fragment shaders in parallel
- Lots of fixed-function hardware for non-programmable stages (clipping, rasterization, texture sampling, z-buffering, etc.)

Discrete GPU card



Integrated GPU (part of CPU die)

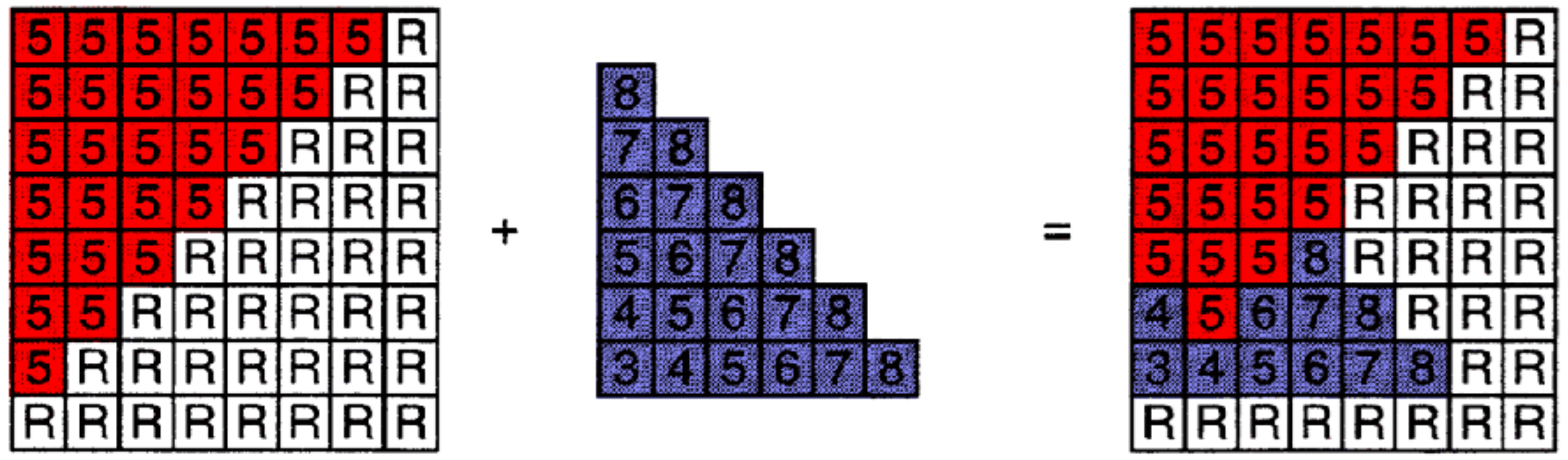

SIMD
Exec
Cache
Texture
Tessellate
Clip/Cull
Rasterize
Zbuffer /
Blend
Scheduler / Work Distributor
GPU
Memory

Ray Tracing

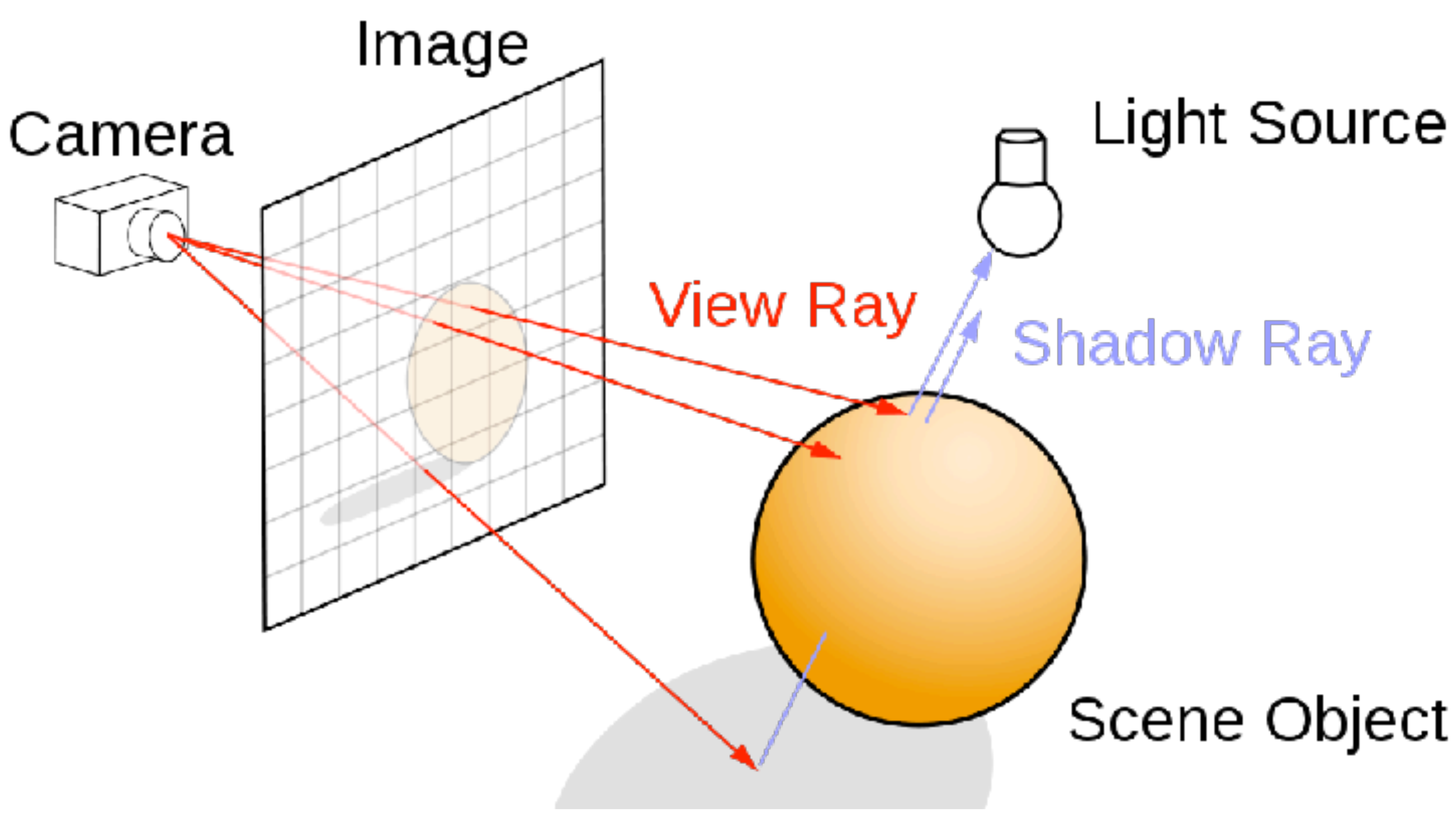
Turner Whitted

Rasterization vs. Ray tracing

for each *shape*:
  for each *sample*:
    get *point* where *shape* covers *sample*
    if *point* is closest point seen by *sample*:
      sample.colour = shade(*point*)



for each *sample*:
  for each *shape*:
    get *point* where *shape* covers *sample*
    if *point* is closest point seen by *sample*:
      sample.colour = shade(*point*)



Ray tracing

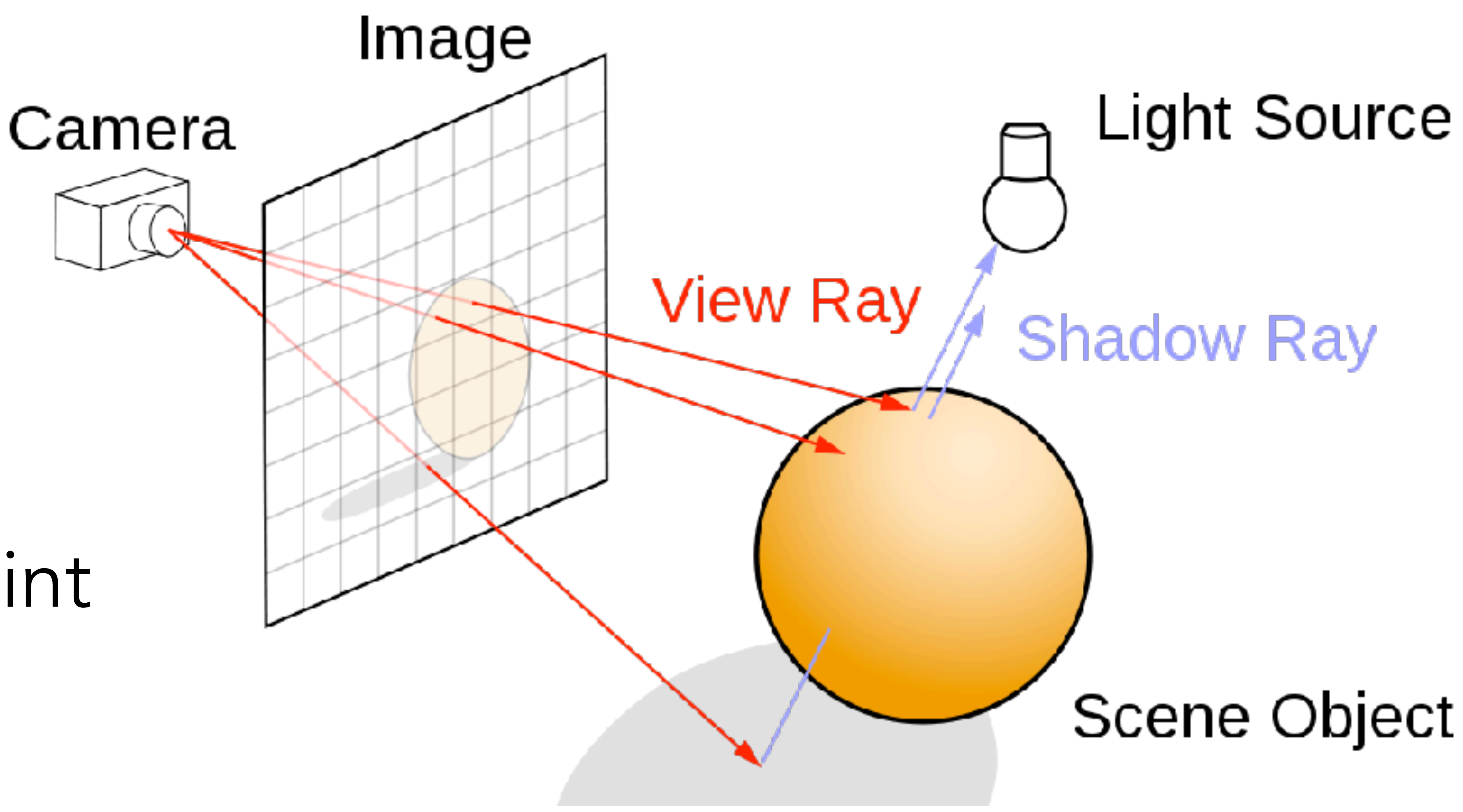
For each sample:

Generate eye ray

Find the closest intersection

Get shaded colour at intersection point

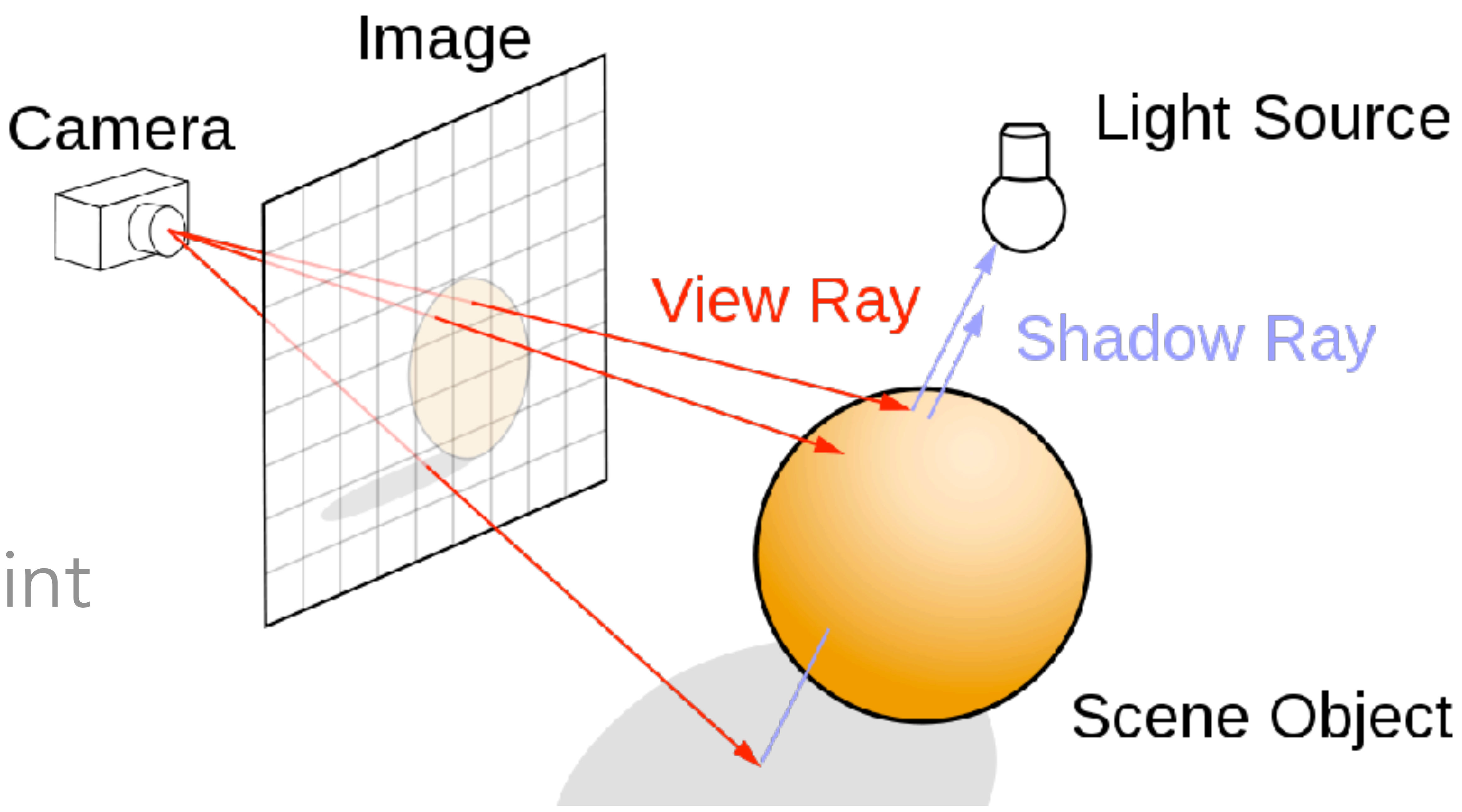
Set sample colour to it



Ray tracing

For each sample:

- **Generate eye ray**
- Find the closest intersection
- Get shaded colour at intersection point
- Set sample colour to it

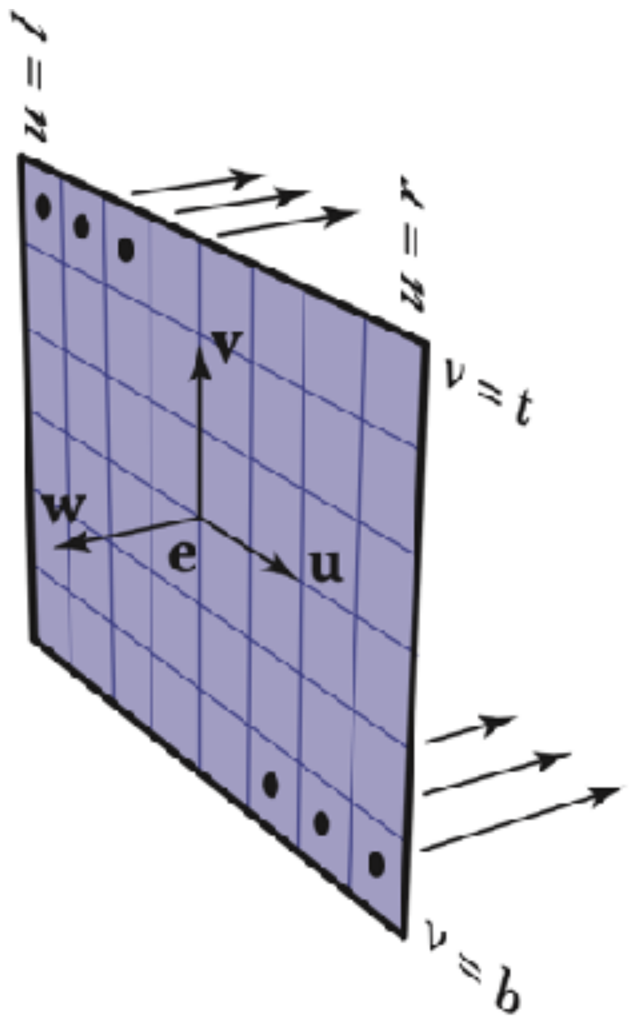


Ray generation

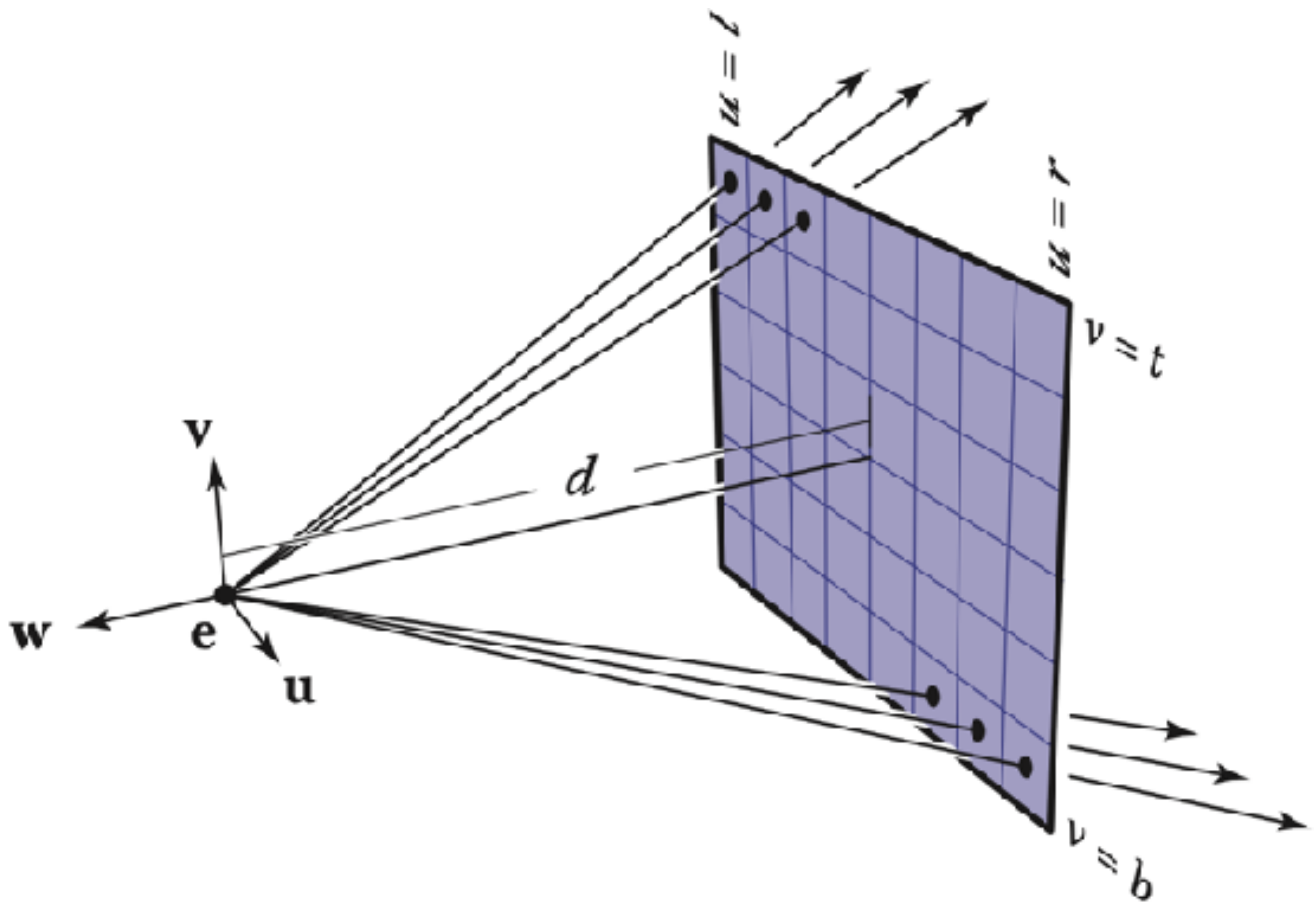
A ray is determined by an origin **o** and a direction **d**.
Any point on the ray is $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ for $t \geq 0$

Each image pixel corresponds to a ray going into the world

- Vertex shader:
  world point → image point
- Ray generation:
  image point → world ray



Parallel projection
same direction, different origins



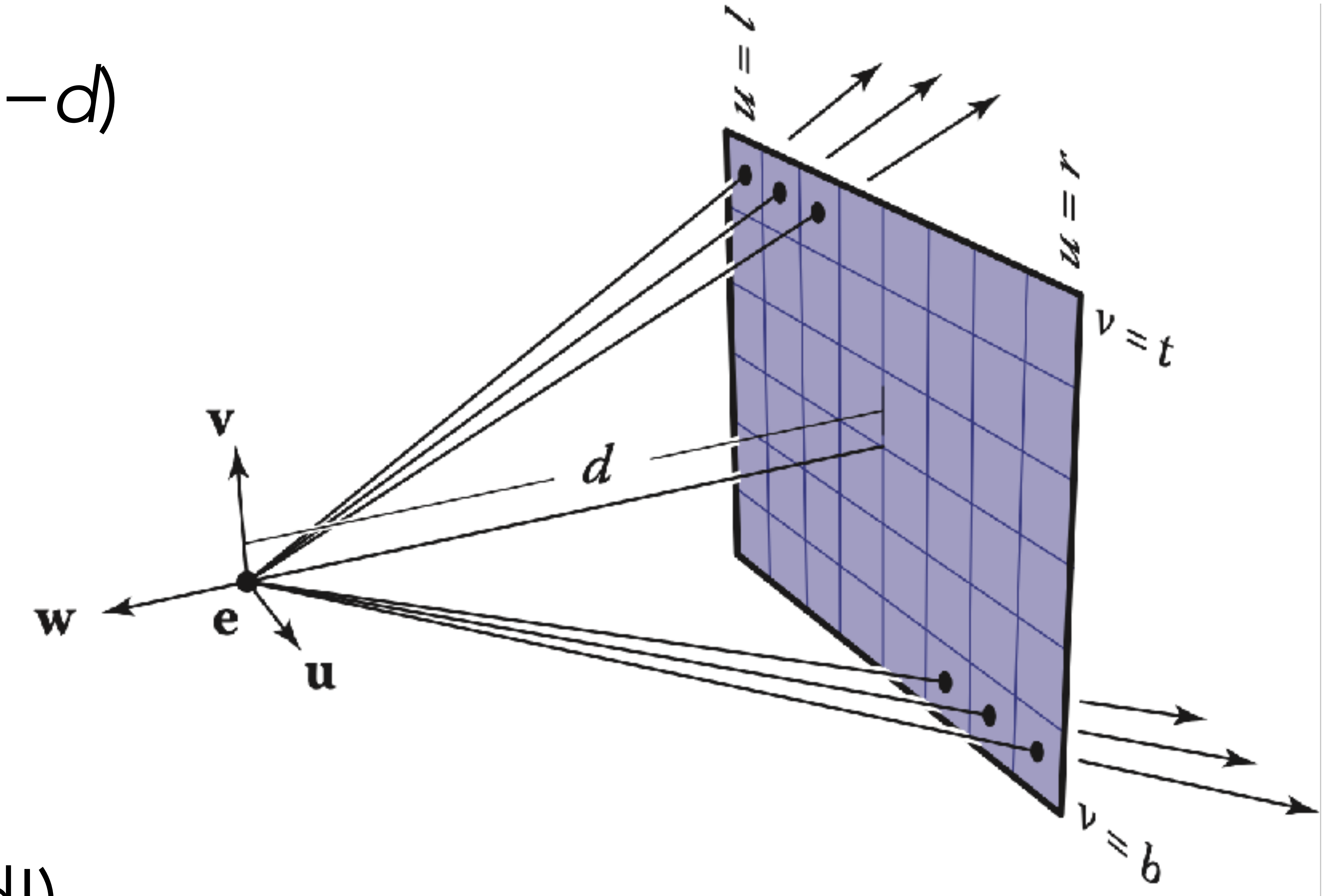
Perspective projection
same origin, different directions

Perspective camera:

- Pixel $(i, j) \rightarrow$ image plane (u, v)
- In camera space, $\mathbf{o} = (0, 0, 0)$, $\mathbf{d} = (u, v, -d)$
- Transform to world space using

$$\mathbf{M}_{\text{view}} = \begin{bmatrix} | & | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(**Note:** We will **not** assume **d** is normalized!)



Perspective projection
same origin, different directions

Ray tracing

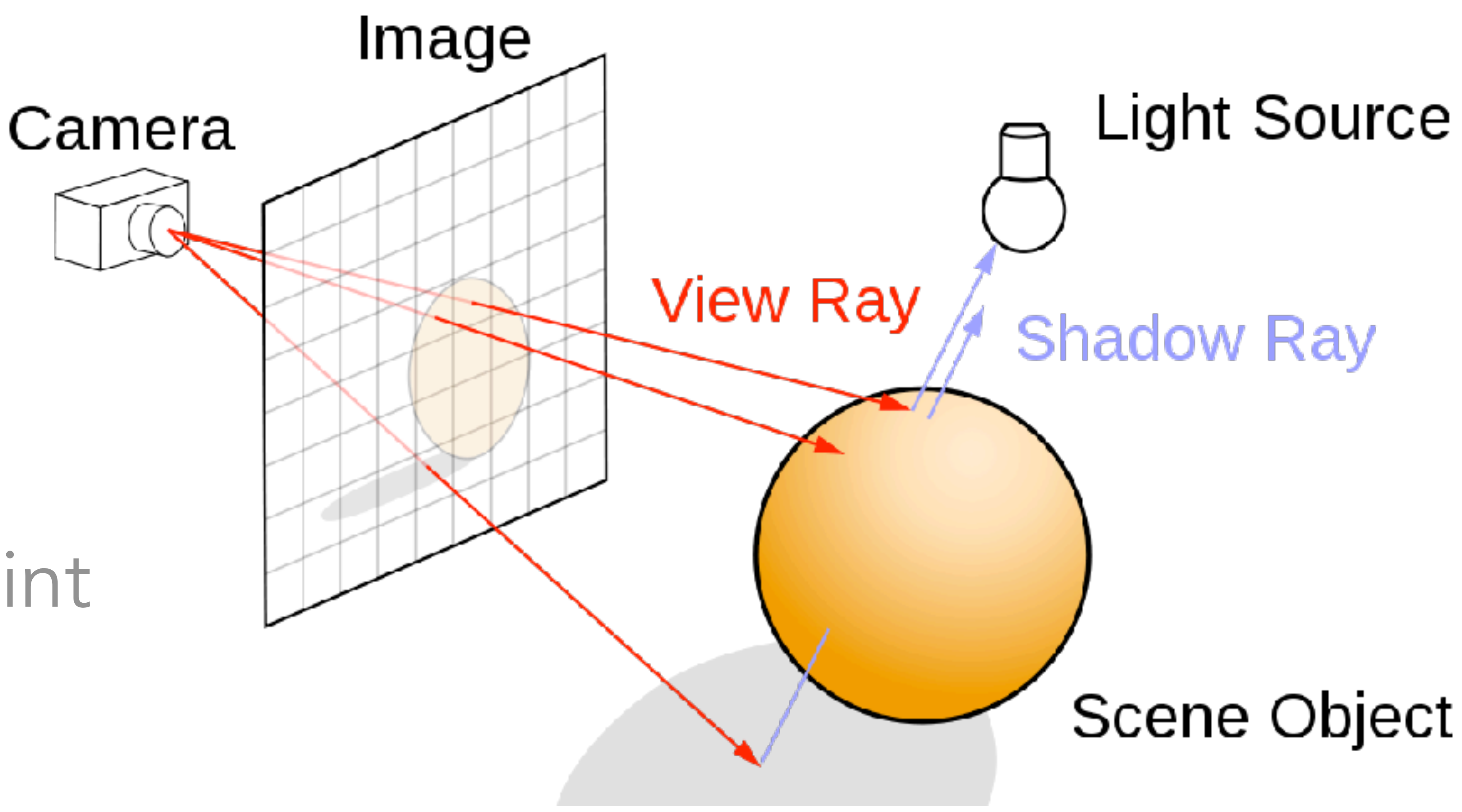
For each sample (x, y):

Generate eye ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$

Find the closest intersection

Get shaded colour at intersection point

Set sample colour to it



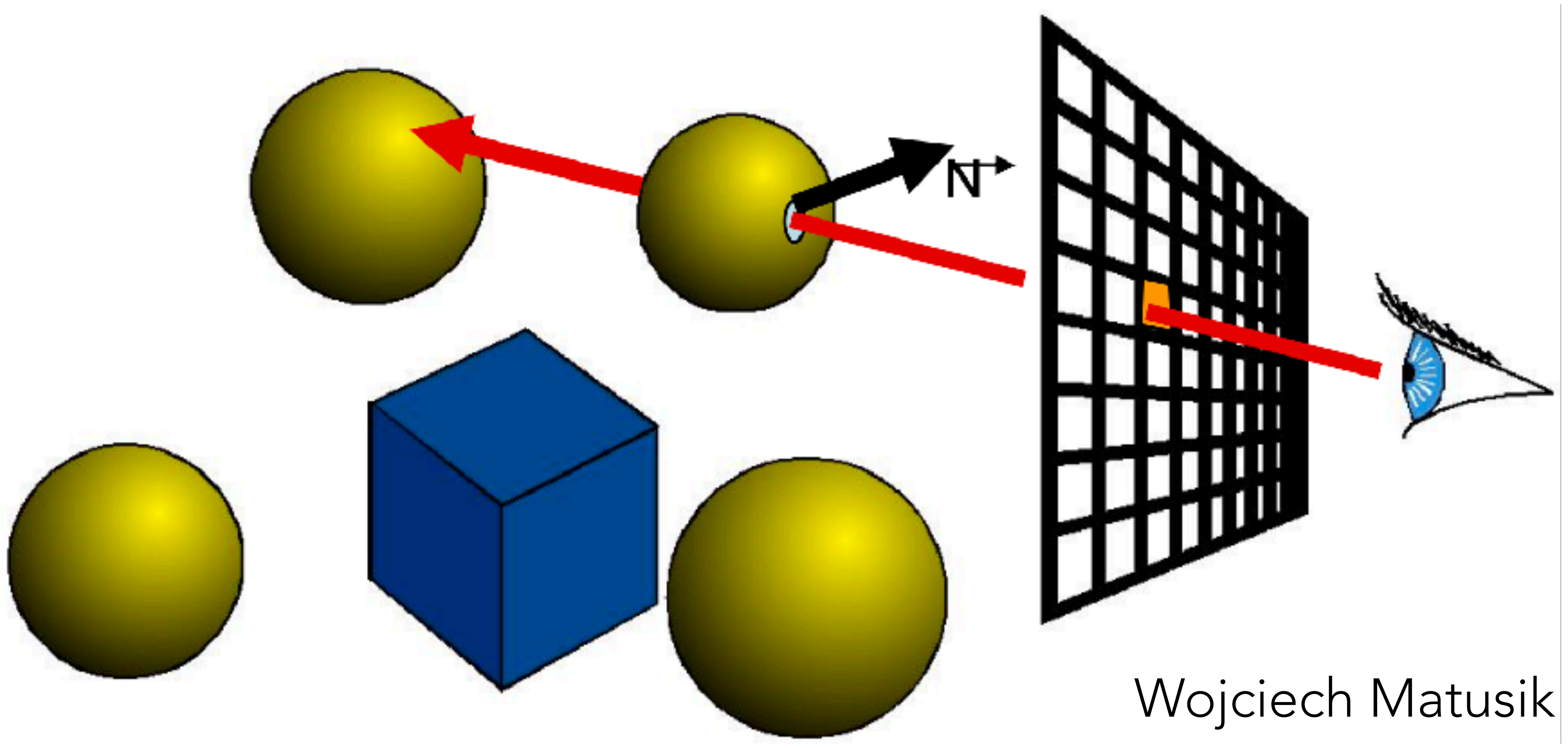
Ray-surface intersection

Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, find closest intersection i.e. minimum t

Return info needed for shading:

- Position $\mathbf{p}$
- Normal $\mathbf{n}$
- Object ID / material properties

(Roughly the same data you would need in a fragment shader)



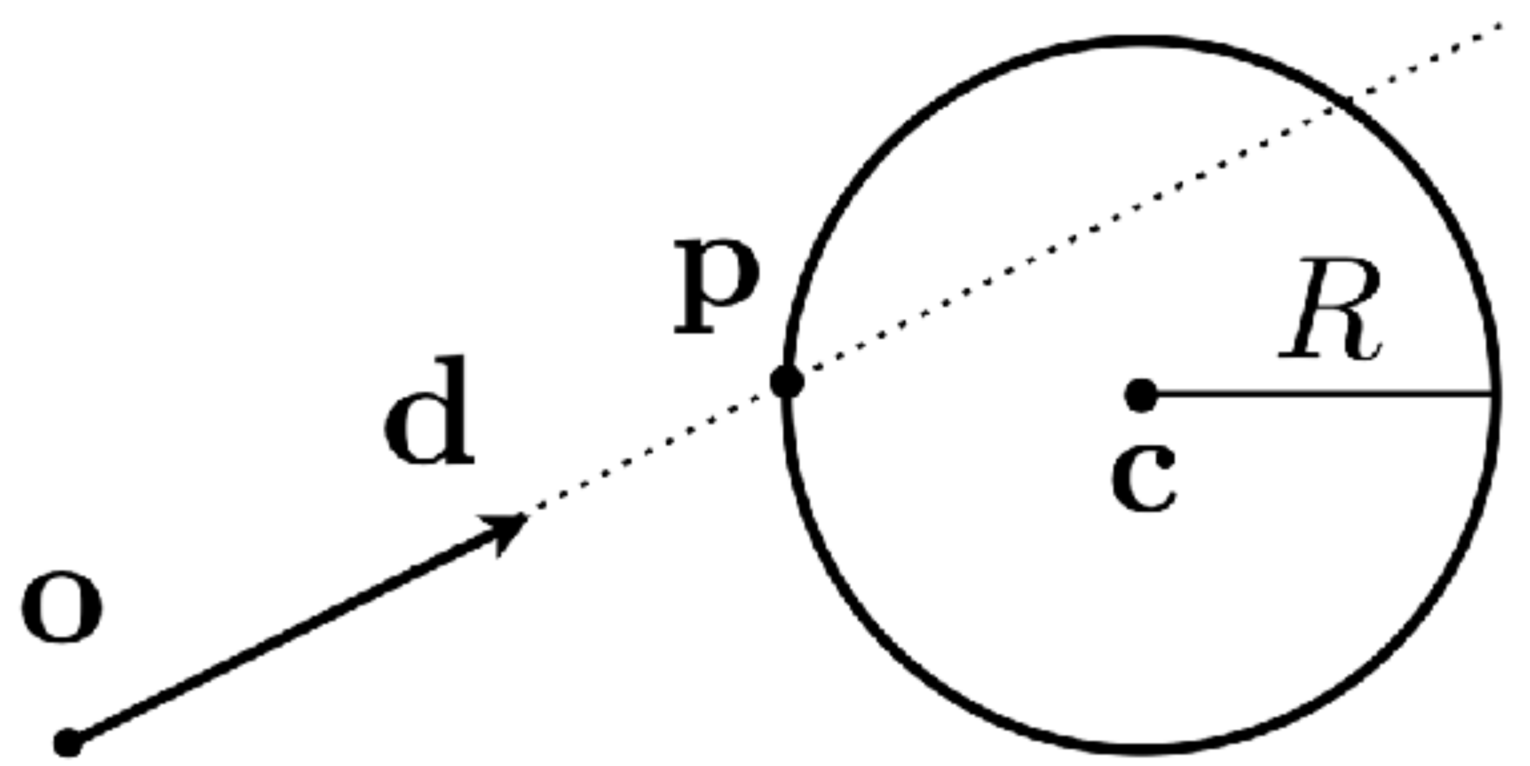
Wojciech Matusik

Ray-sphere intersection

Ray equation: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$

Sphere equation: $\|\mathbf{p} - \mathbf{c}\|^2 = R^2$

Intersection point must satisfy both:



$$\|(\mathbf{o} - \mathbf{c}) + t\mathbf{d}\|^2 = R^2$$

$$\|\mathbf{d}\|^2\, t^2 + 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})\, t + \|\mathbf{o} - \mathbf{c}\|^2 - R^2 = 0$$

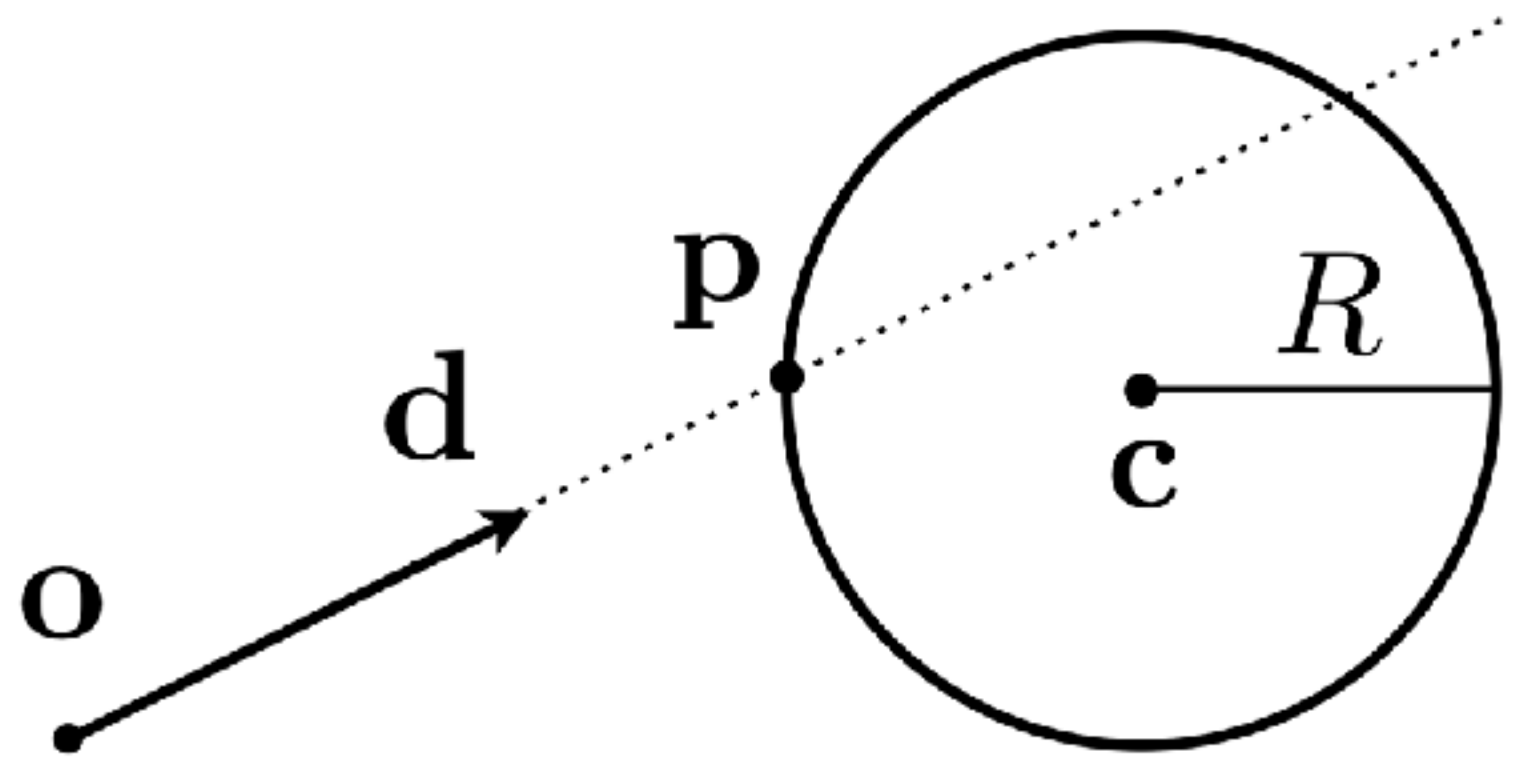
(Recall $\|\mathbf{v}\|^2 = \mathbf{v} \cdot \mathbf{v}$)

Quadratic equation, solve for t

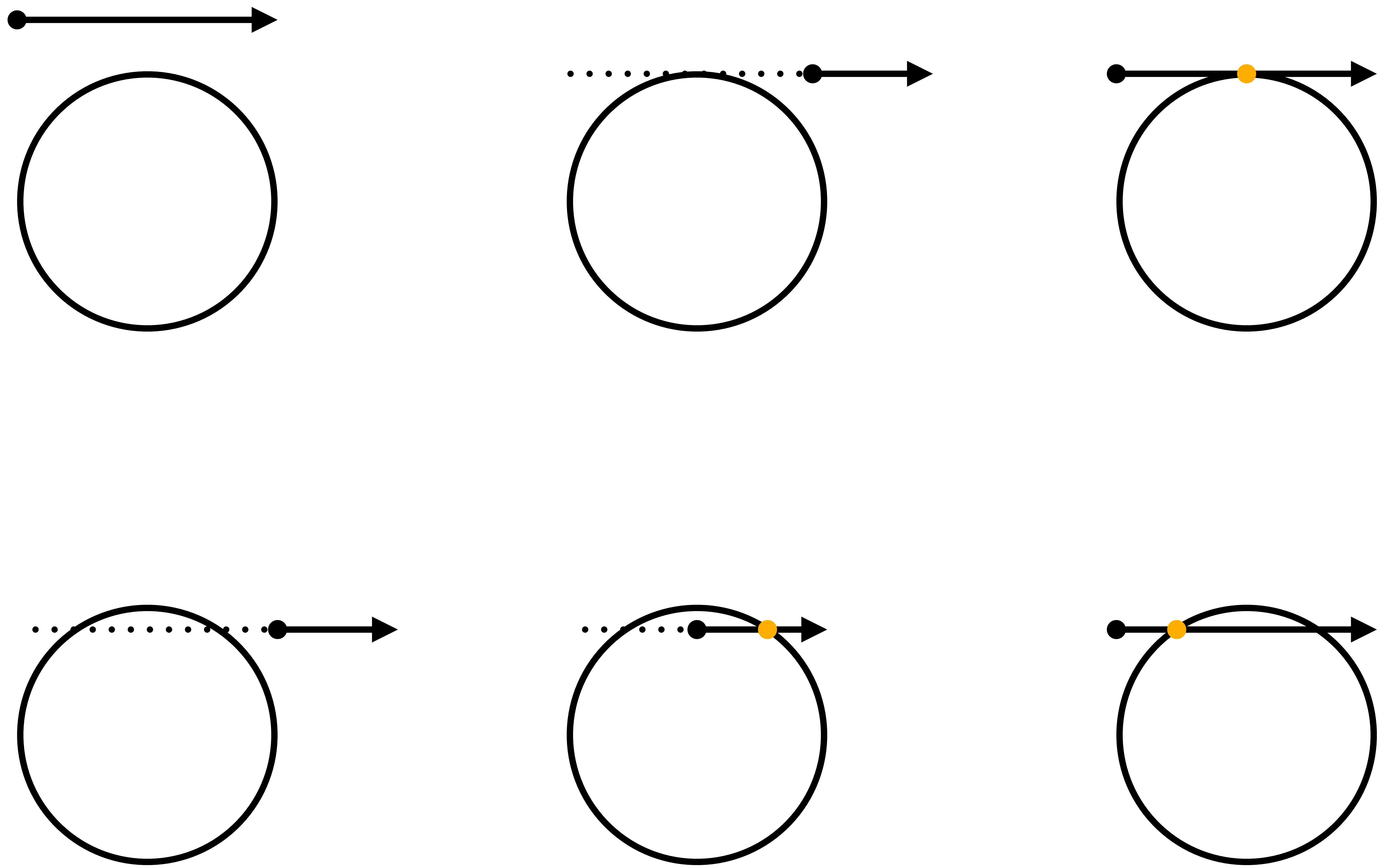
3 cases:

- No solution
- One solution t_1
- Two solutions t_1 and t_2

What do they mean geometrically?



- No solution
- One solution t_1
  - $t_1 < 0$
  - $t_1 > 0$
- Two solutions t_1 and t_2
  - $t_1 < t_2 < 0$
  - $t_1 < 0 < t_2$
  - $0 < t_1 < t_2$



In general: Find all solutions, discard those with $t < 0$, take minimum of remaining

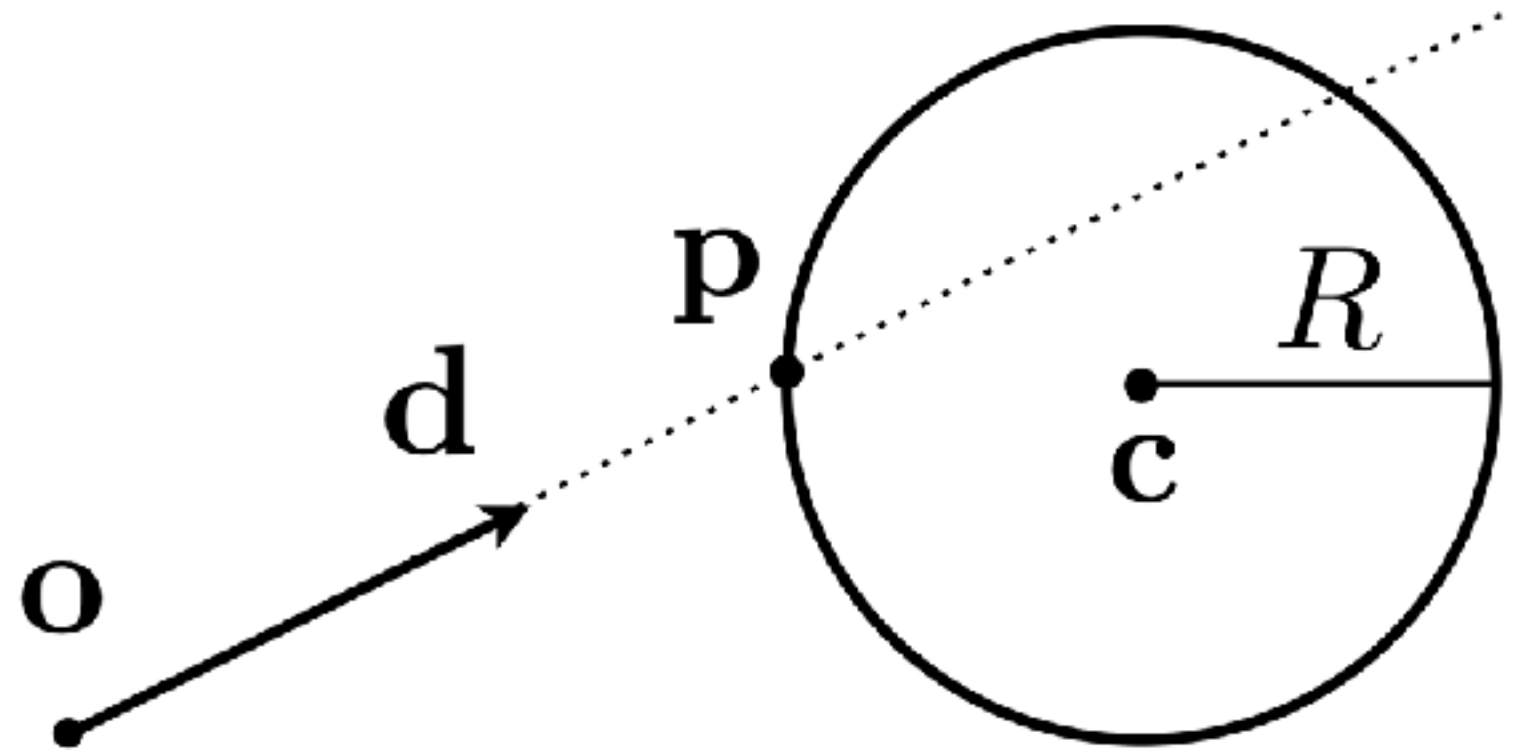
Find t of closest intersection

Then get intersection point from equation of ray:

$$\mathbf{p} = \mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

What about the surface normal?

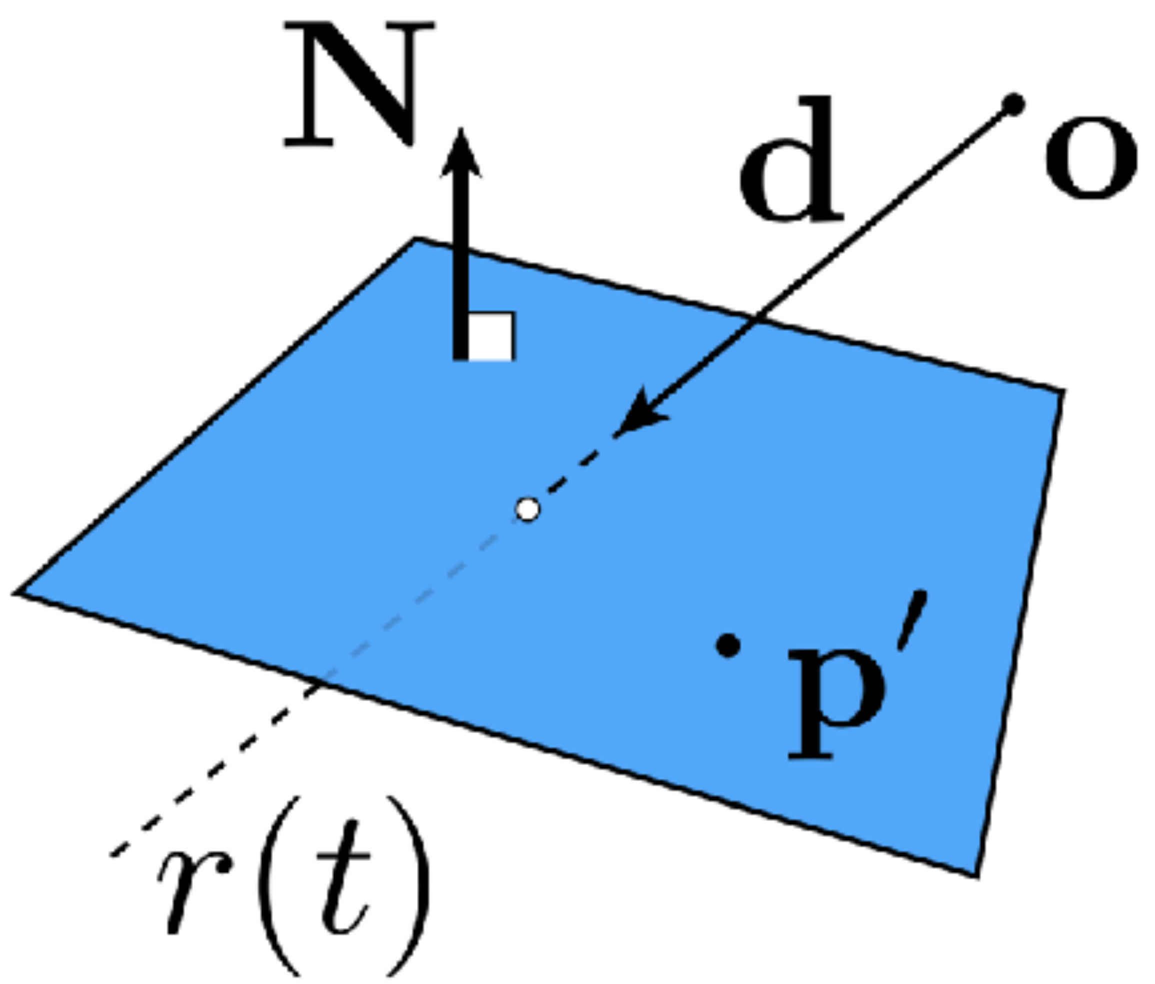
$$\mathbf{n} = (\mathbf{p} - \mathbf{c})/\|\mathbf{p} - \mathbf{c}\|$$



Ray-plane intersection

Plane equation: $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

any known point
on the plane



$$\mathbf{n} \cdot (\mathbf{o} + {\color{red}t}\mathbf{d} - \mathbf{p}_0) = 0$$

$${\color{red}t} = (\mathbf{n} \cdot (\mathbf{p}_0 - \mathbf{o}))/(\mathbf{n} \cdot \mathbf{d})$$

Ray-triangle intersection

Intersect ray with plane, then check if it is inside triangle?