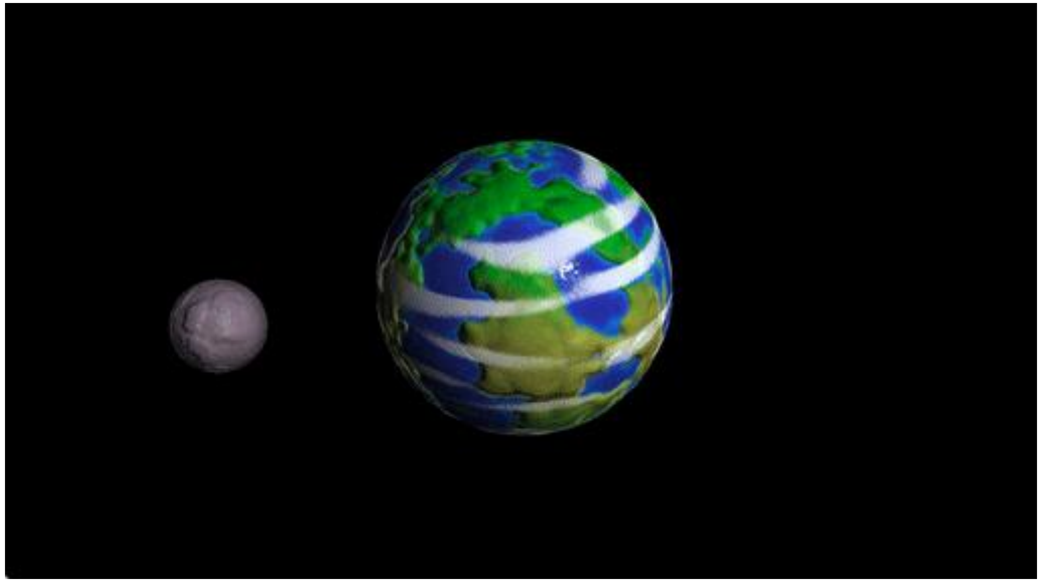
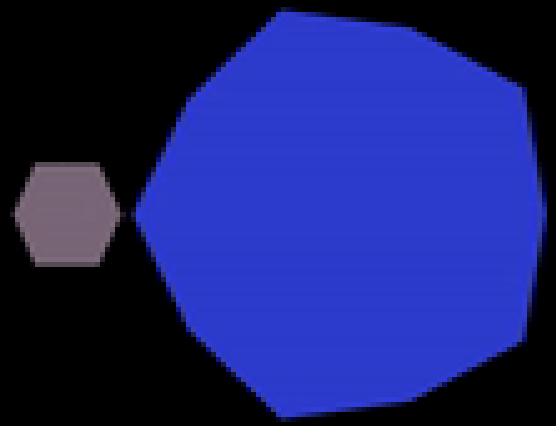
Transformations, Shaders & Rasterization



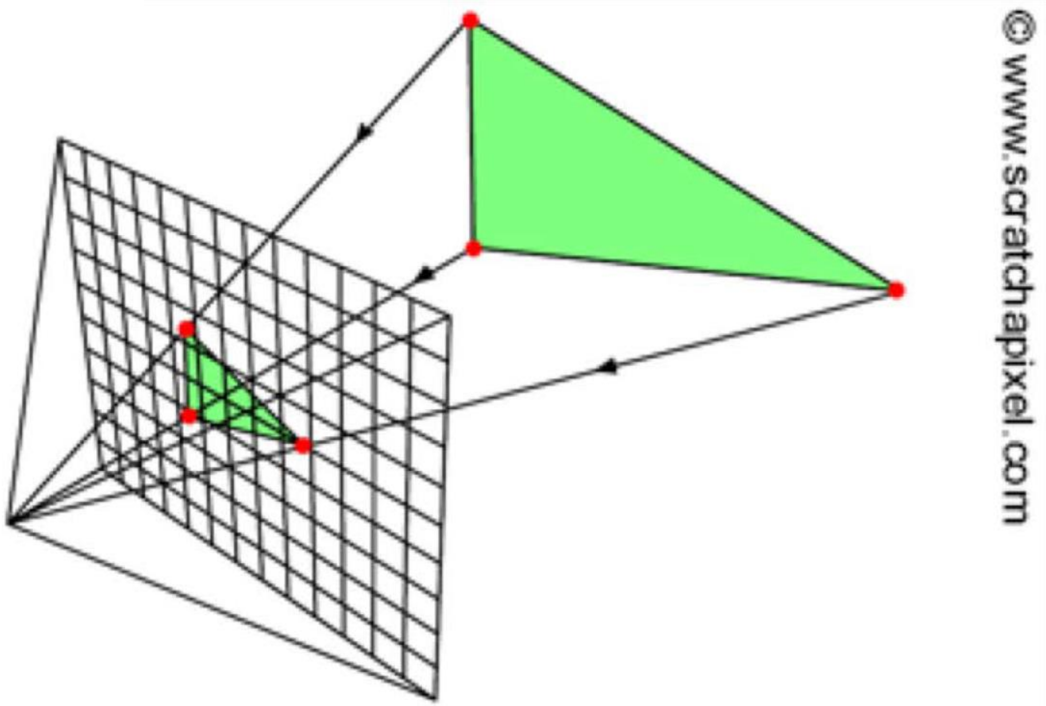
Some Slides/Images adapted from Marschner and Shirley and David Levin

Agenda

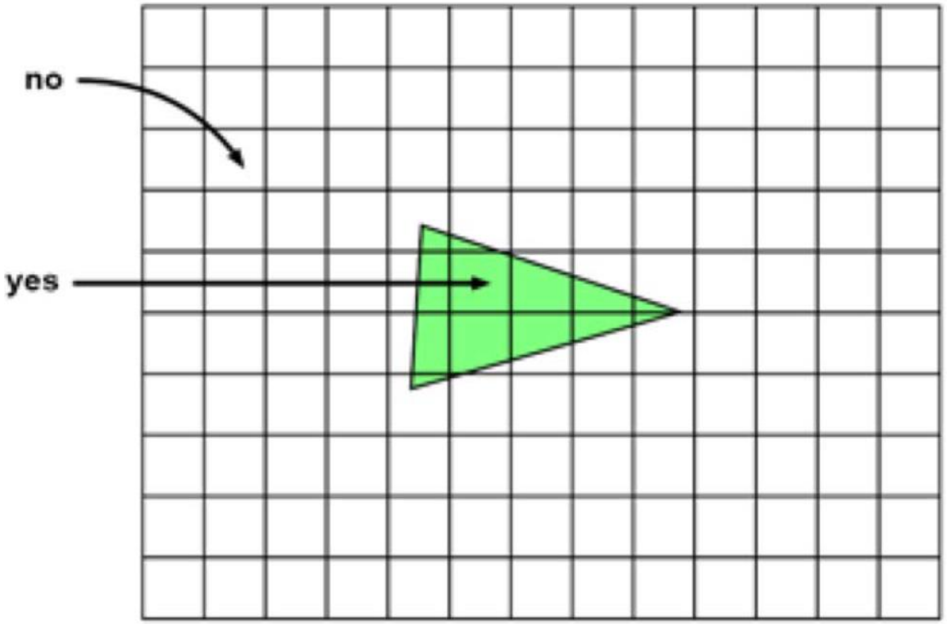
- Rasterization and the Modern Graphics Pipeline
- Transformations
- Shaders
- Normal and Bump Mapping
- Perlin Noise



Rasterization



1. Project Vertices to Image Plane



2. Turn on pixels inside triangle

https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation

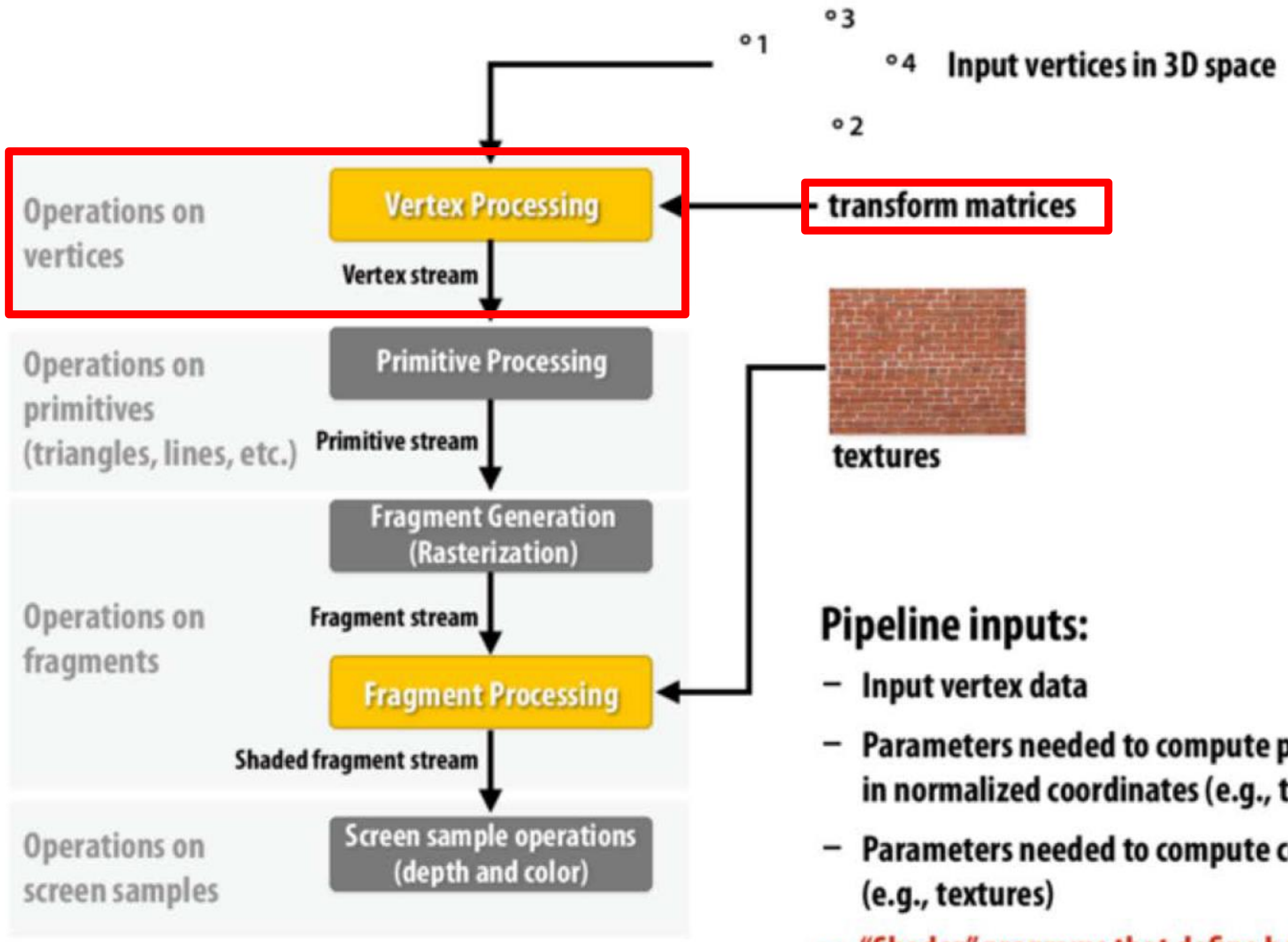
Rasterization

```
// rasterization algorithm
for (each triangle in scene) {
    // STEP 1: project vertices of the triangle using perspective projection
    Vec2f v0 = perspectiveProject(triangle[i].v0);
    Vec2f v1 = perspectiveProject(triangle[i].v1);
    Vec2f v2 = perspectiveProject(triangle[i].v2);
    for (each pixel in image) {
        // STEP 2: is this pixel contained in the projected image of the triang
        if (pixelContainedIn2DTriangle(v0, v1, v2, x, y)) {
            image(x,y) = triangle[i].color;
        }
    }
}
```

https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation

Modern Graphics Pipeline

OpenGL/Direct3D graphics pipeline *



Pipeline inputs:

- Input vertex data
- Parameters needed to compute position on vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)
- "Shader" programs that define behavior of vertex and fragment stages

* several stages of the modern OpenGL pipeline are omitted

Transformations

Transformation/Deformation in Graphics:

A function f, mapping points/vectors to points/vectors.
simple transformations are usually invertible.

$$[x\ y]^T \underset{f^{-1}}{\overset{f}{\rightleftarrows}} [x'\ y']^T$$

Applications:

- Placing objects in a scene.
- Composing an object from parts.
- Animating objects.

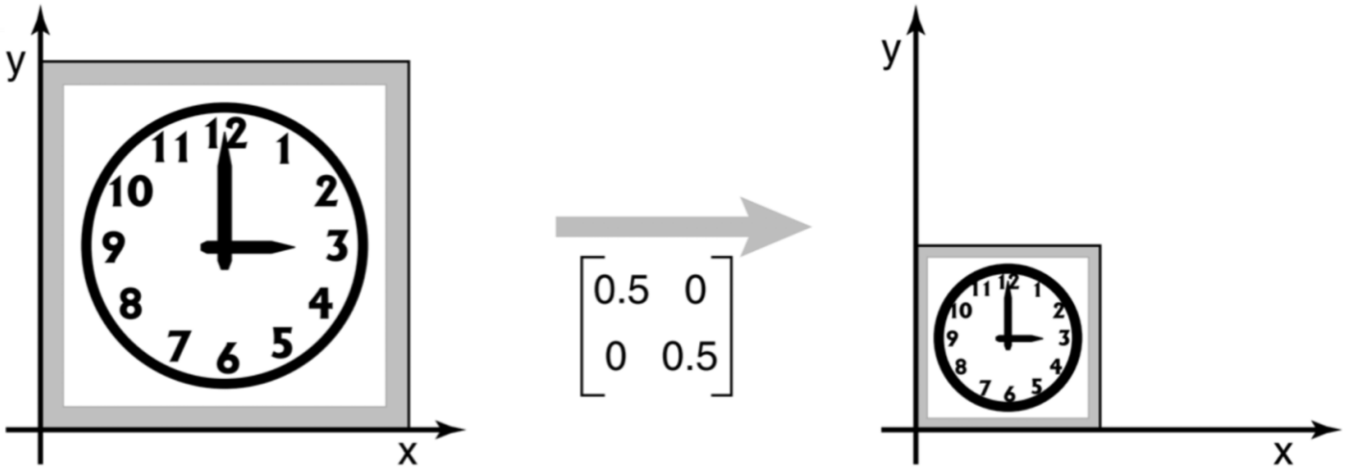
Linear Transformations

A transformation matrix *A* maps a point/vector p=$[x \;\; y]^T$ to *Ap*

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}$$

2D Linear Transformations - Scale

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$

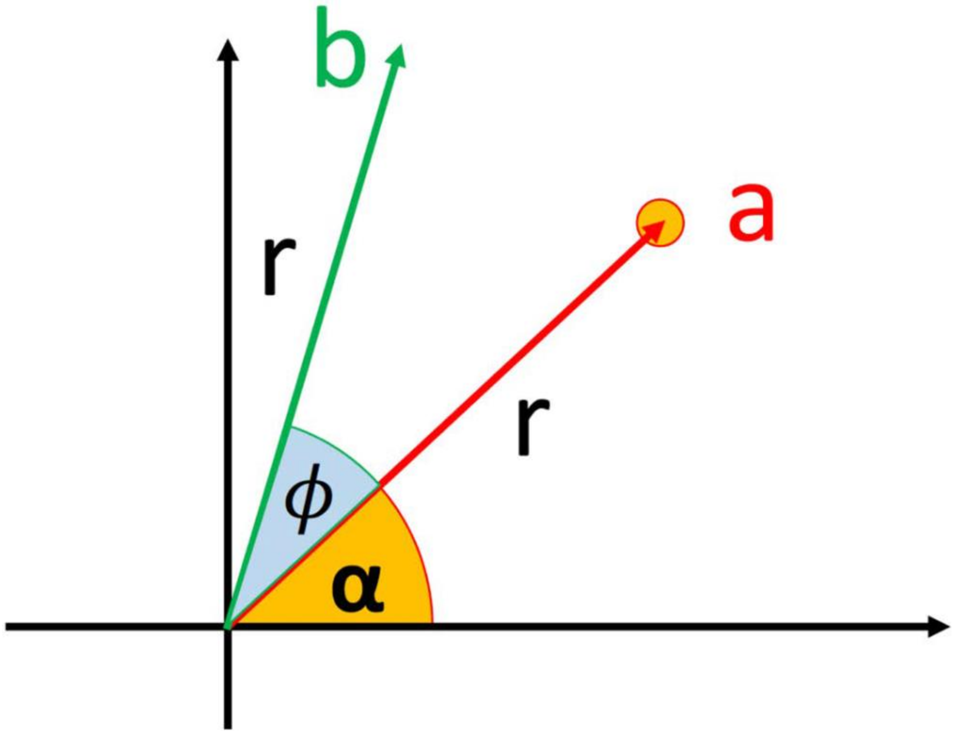


$$\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

When *Sx* = *Sy* we say the scaling is uniform

2D Linear Transformations - Rotation

$$\text{rotate}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

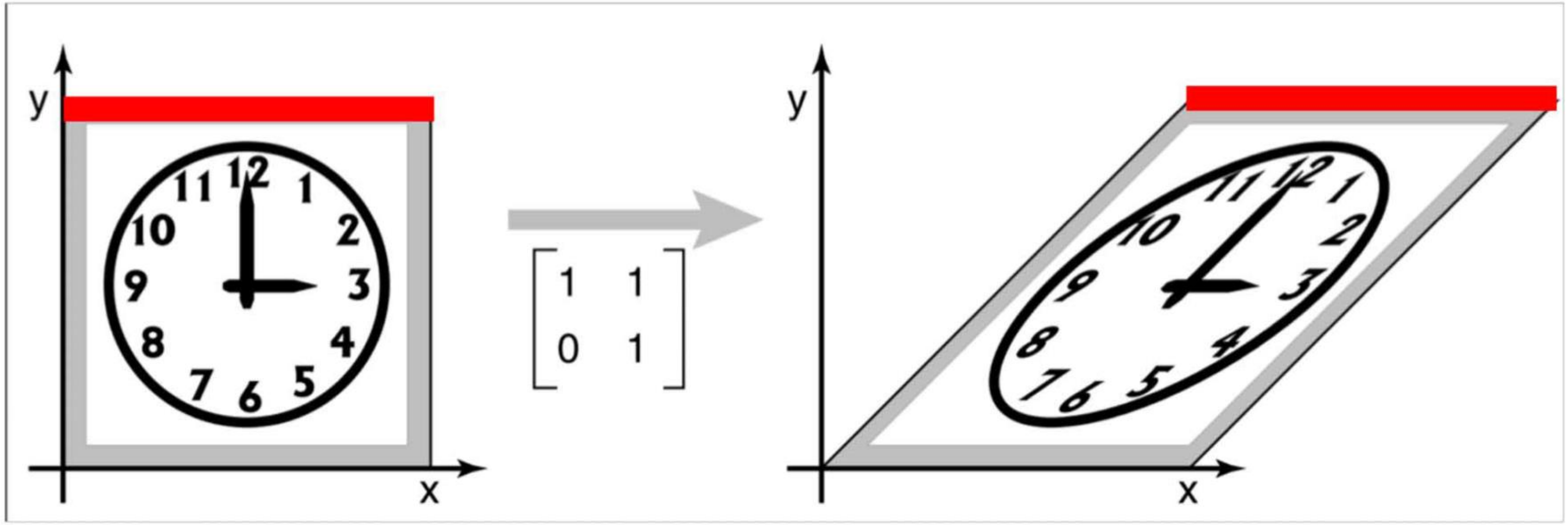


$x = r\ \cos\alpha$
$y = r\ \sin\alpha$

$x' = r\ \cos(\alpha+\varphi) = r\ (\cos\alpha * \cos\varphi - \sin\alpha * \sin\varphi) = x * \cos\varphi - y * \sin\varphi$
$y' = r\ \sin(\alpha+\varphi) = r\ (\cos\alpha * \sin\varphi + \sin\alpha * \cos\varphi) = x * \sin\varphi + y * \cos\varphi$

2D Linear Transformations - Shear

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + sy \\ y \end{bmatrix}$$



These are always the same length

2D Linear Transformations- Translation?

$$T \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

2D Affine Transformations - Translation

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \boxed{1} \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + t_x \\ a_{21}x + a_{22}y + t_y \\ 1 \end{bmatrix}$$

$$Ax + t = b$$

Cartesian ⇔ Homogeneous Coordinates

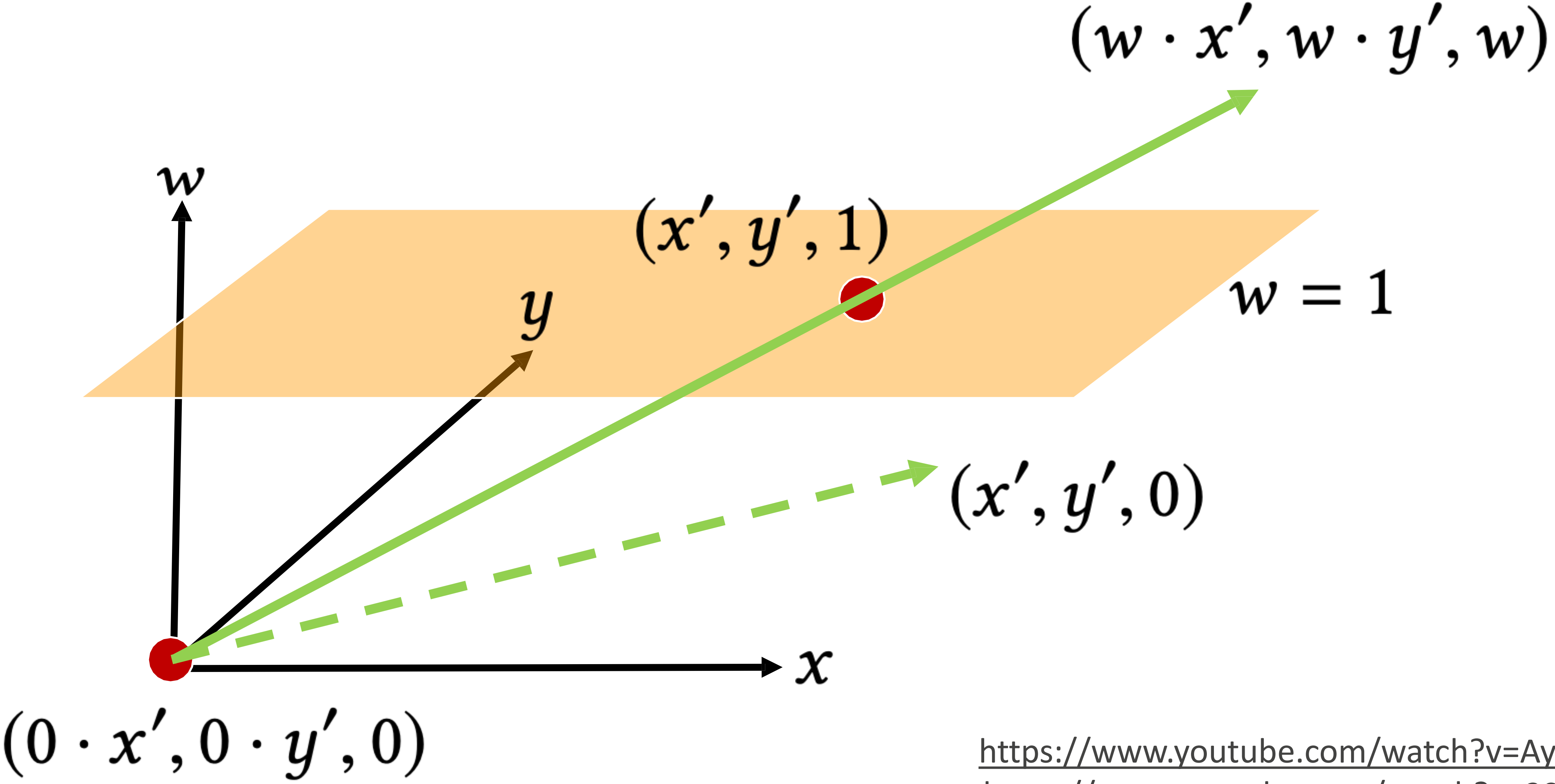
Cartesian $[x\ y]^T$ => Homogeneous $[x\ y\ 1]^T$

Homogeneous $[x\ y\ w]^T$ => Cartesian $[x/w\quad y/w\ 1]^T$

Homogeneous points are equal if they represent the same Cartesian point. For eg. $[4\ {-6}\ 2]^T = [-6\ 9\ {-3}]^T$.

What about w=0?

Geometric Intuition



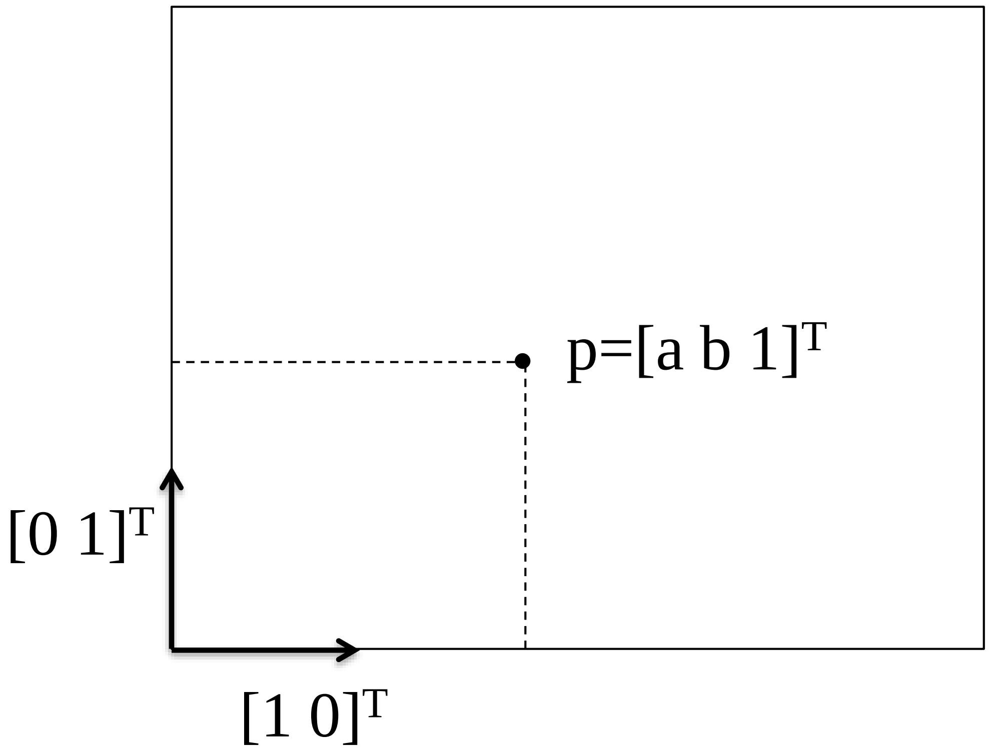
https://www.youtube.com/watch?v=AyW4Y8APjK4
https://www.youtube.com/watch?v=2Snoepcmi9U
https://www.youtube.com/watch?v=Q2uItHa7GFQ

Points at ∞ in Homogeneous Coordinates

$[x\ y\ w]^T$ with $w=0$ represent points at infinity, though with direction $[x\ y]^T$ and thus provide a natural representation for **vectors**, distinct from **points** in Homogeneous coordinates.

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow[\text{homogeneous coordinates}]{\text{Considered as a } \underline{\textbf{\textit{vector}}} \text{ in 3D}} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Points as Homogeneous 2D Point Coords



$$p = a*[1\ 0\ 0]^T + b*[0\ 1\ 0]^T + [0\ 0\ 1]^T$$

basis vectors   origin

Representing 2D transforms as a 3x3 matrix

Translate a point $[x\ y]^T$ by $[t_x\ t_y]^T$:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Rotate a point $[x\ y]^T$ by an angle t :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos t & -\sin t & 0 \\ \sin t & \cos t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Scale a point $[x\ y]^T$ by a factor $[s_x\ s_y]^T$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Properties of 2D transforms

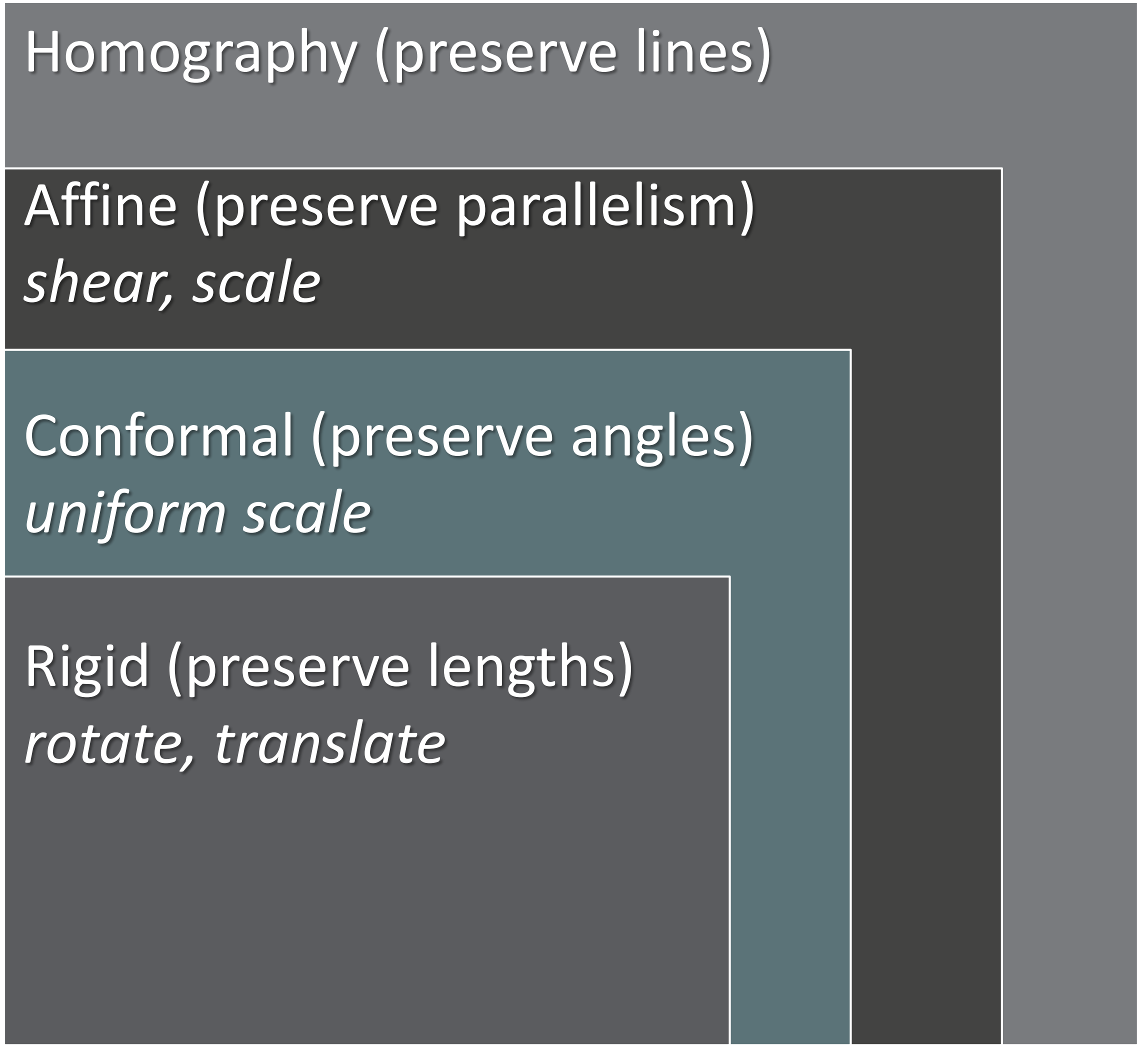
…these 3x3 transforms when invertible are called **Homographies**.
they map **lines** to **lines**.
i.e. points ***p*** on a line ***l*** transformed by a Homography ***H*** produce points ***Hp*** that also lie on a line.

…a more restricted set of transformations also preserve parallelism in lines. These are called **Affine** transforms.

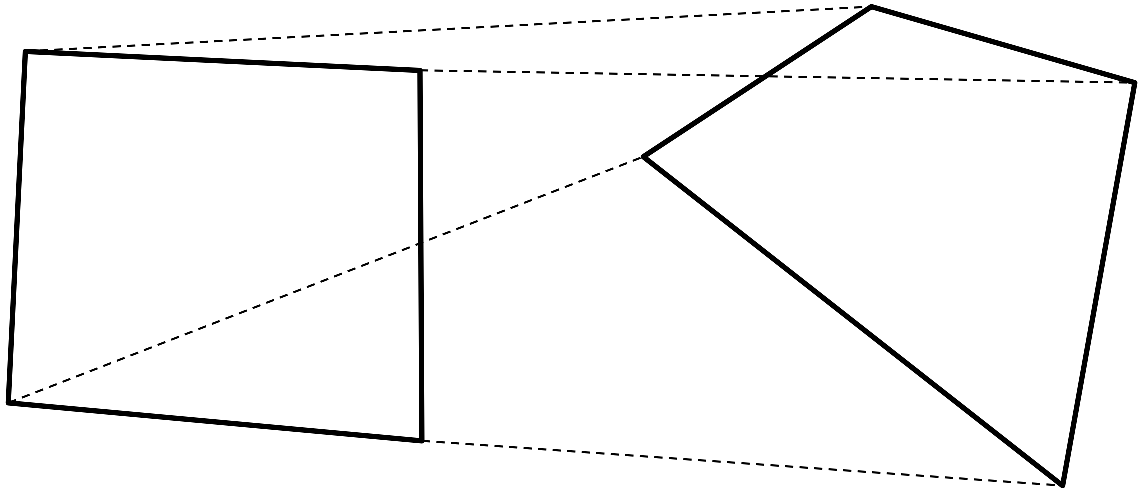
…transforms that further preserve the angle between lines are called **Conformal**.

…transforms that additionally preserve the lengths of line segments are called **Rigid**.

Properties of 2D transforms



Homography: mapping four points



A mapping of 4 points $\mathbf{p_i}$ to $\mathbf{Hp_i}$ uniquely define the 3x3 Homography matrix **H**?

H [x y 1] = [x' y' w'] .
Say **H[2][2]=k**. What does the transform **H'= (1/k)*H** do?
H' [x y 1] = 1/k H [x y 1] = 1/k [x' y' w'] = [x' y' w'].
But **H'[2][2]** = 1.

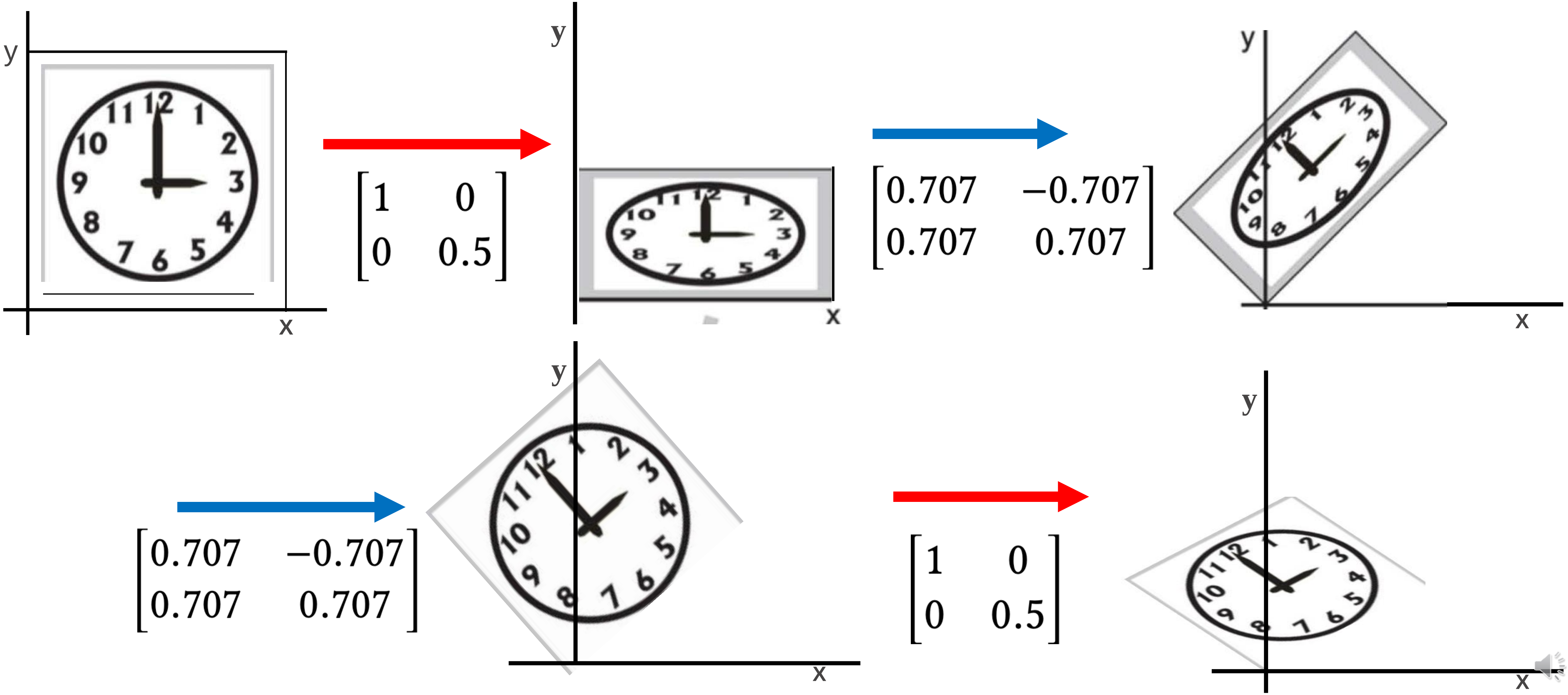
Affine properties: composition

Affine transforms are closed under composition.
i.e. Applying transform (A_1,t_1) (A_2,t_2) in sequence results in an overall Affine transform.

$$p' = A_2 (A_1 p + t_1) + t_2 \quad \Rightarrow (A_2 A_1)p + (A_2 t_1 + t_2)$$

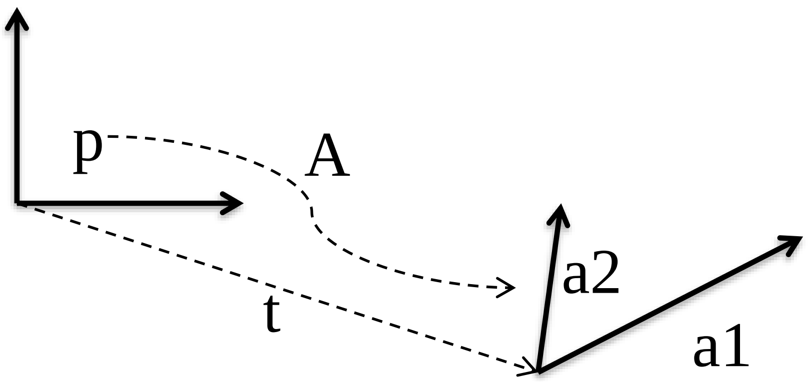
Composing Transformations

Any sequence of linear transforms can be concatenated into a single 3x3 matrix.
In general transforms DO NOT commute (some special combinations are).



Affine transform: geometric interpretation

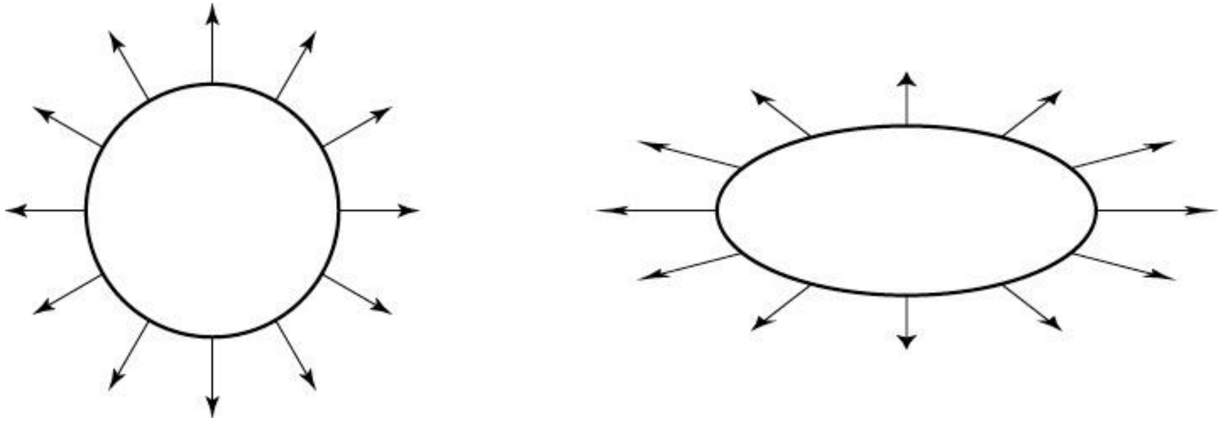
A change of basis vectors and translation of the origin



point p in the local coordinates of a reference frame defined by <a1,a2,t> is

$$\begin{pmatrix} \begin{bmatrix} a1 & a2 \end{bmatrix} & \begin{bmatrix} t \end{bmatrix} \\ 0 \quad 0 & 1 \end{pmatrix}^{-1} \begin{bmatrix} p \end{bmatrix}$$

Transforming tangents and normals



Tangents can be written as the difference of points on an object.
Say **t = p-q.** The transformed tangent **t'= Mp-Mq = M(p-q) = Mt.**

But **n' != Mn** for surface normals. (imagine scaling an object in x-direction)
Say **n'= Hn**. What is **H**?

$$\mathbf{t'}^T \quad \mathbf{n'} \quad = 0$$

$$(\mathbf{Mt})^T (\mathbf{Hn}) = 0$$

$$\mathbf{t}^T (\mathbf{M}^T \mathbf{H})\, \mathbf{n} = 0$$

Remember that $\mathbf{t}^T \mathbf{n} = 0.$ so... $\mathbf{H} = (\mathbf{M}^T)^{-1}$

Representing 3D transforms as a 4x4 matrix

Translate a point $[x\ y\ z]^T$ by $[t_x\ t_y\ t_z]^T$:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

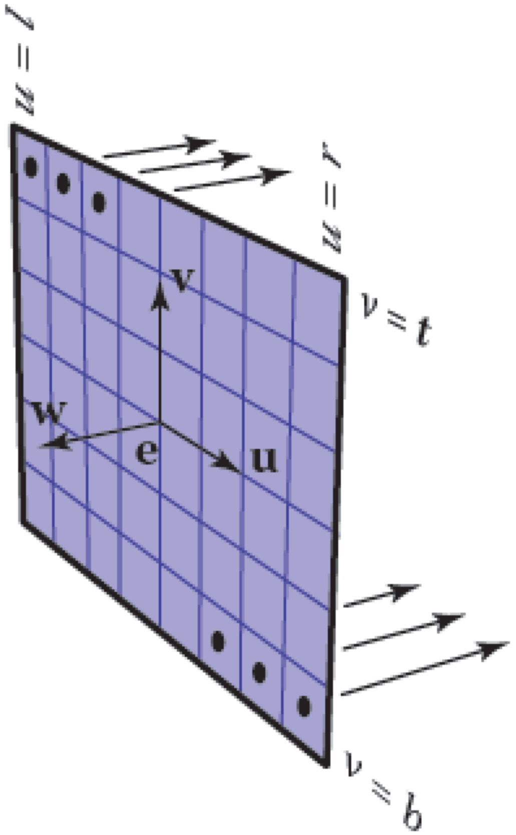
Rotate a point $[x\ y\ z]^T$ by an angle t around z axis:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos t & -\sin t & 0 & 0 \\ \sin t & \cos t & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

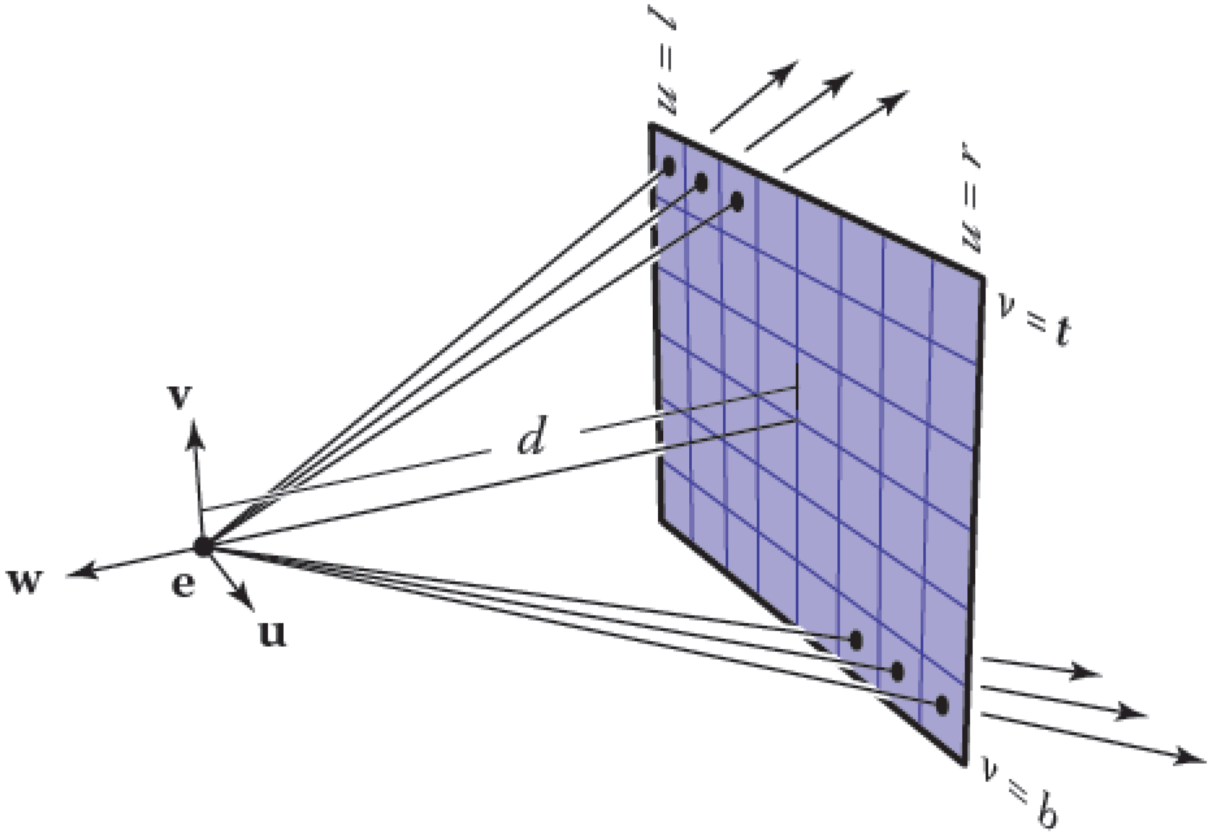
Scale a point $[x\ y\ z]^T$ by a factor $[s_x\ s_y\ s_z]^T$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Reminder: Camera Model

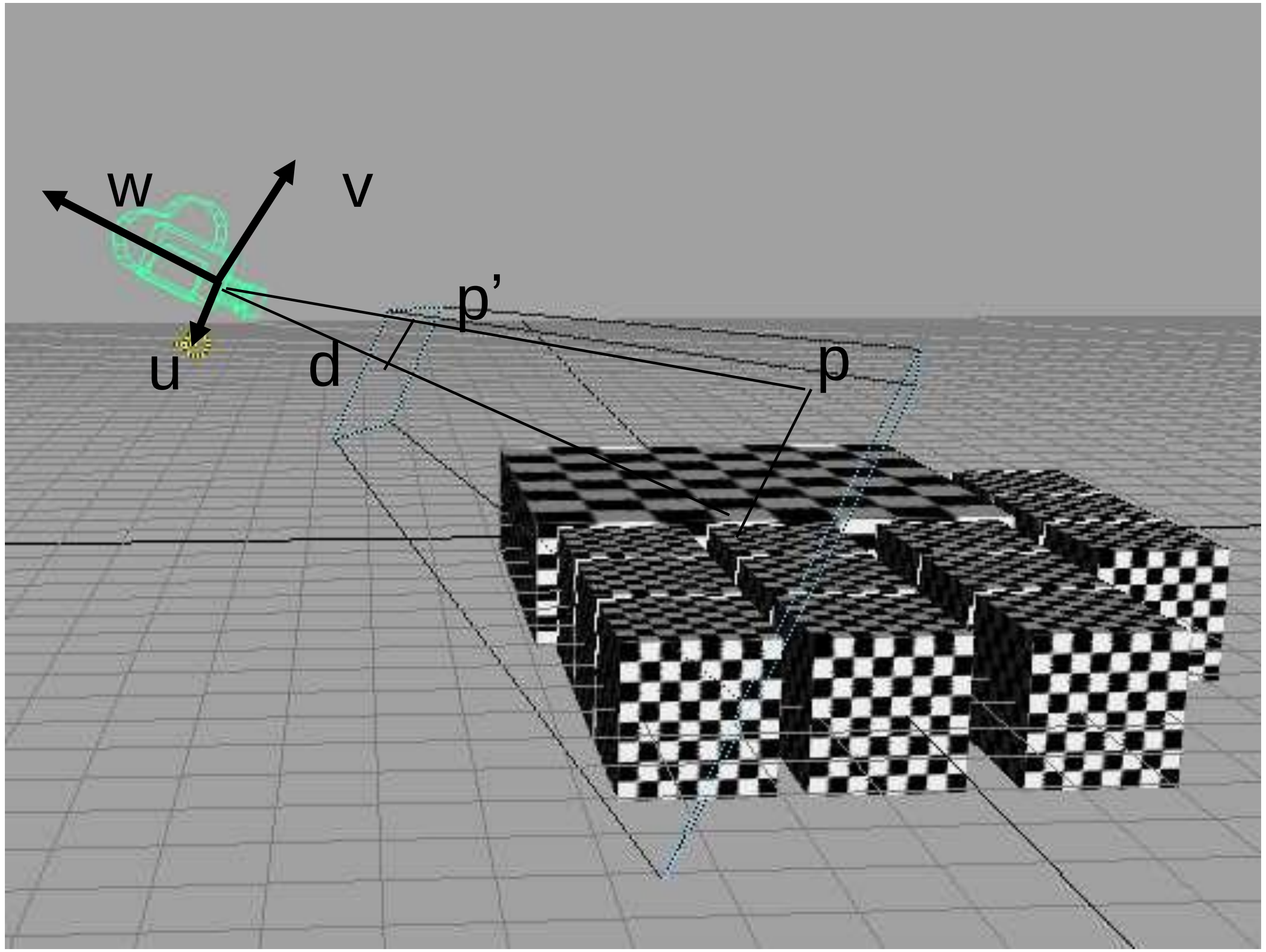


Parallel projection
same direction, different origins



Perspective projection
same origin, different directions

Perspective projection

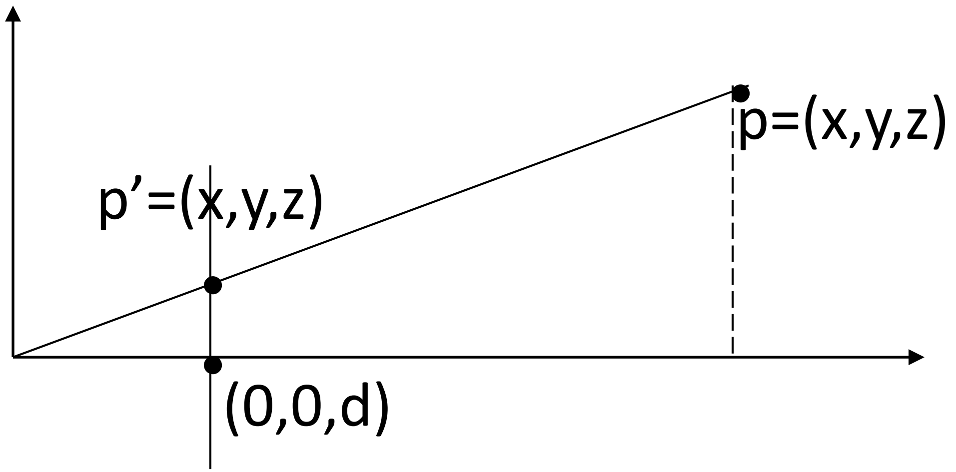


Perspective Projection

$y' = yd/z$

$x' = xd/z$

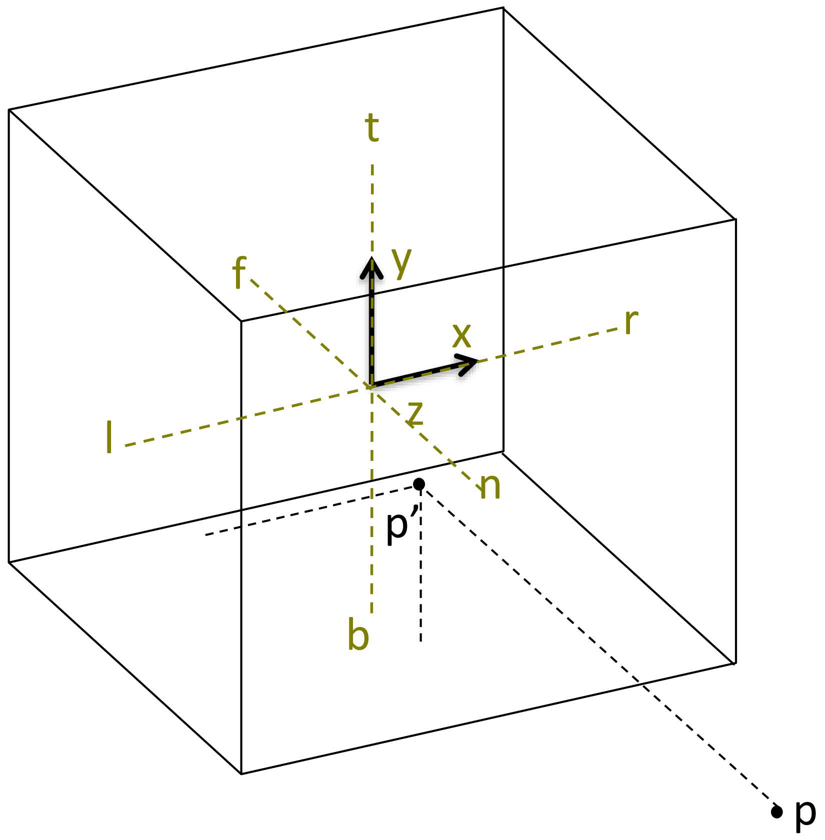
$z' = d$



Insight: **$w' = z/d$**

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

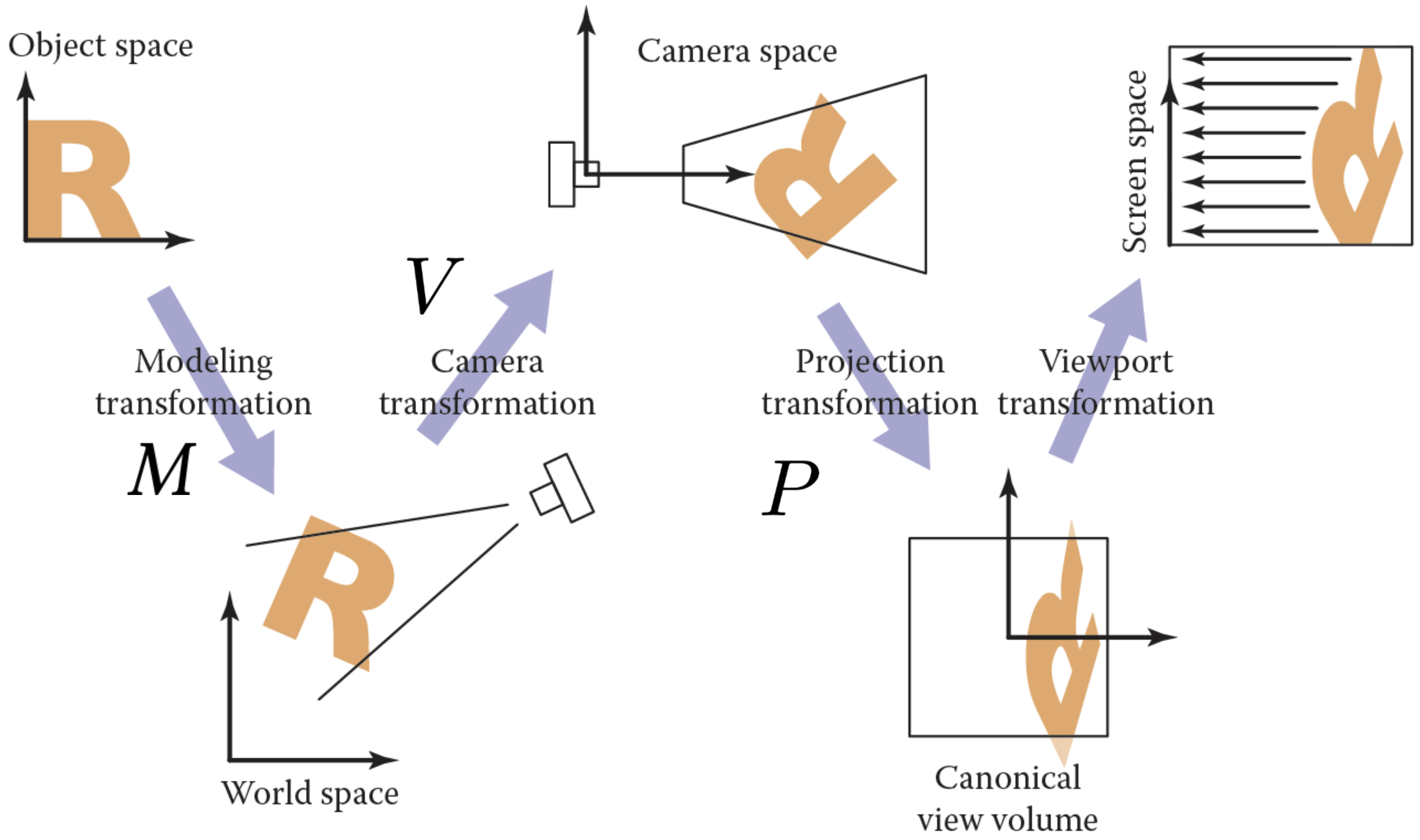
Cannonical view volume



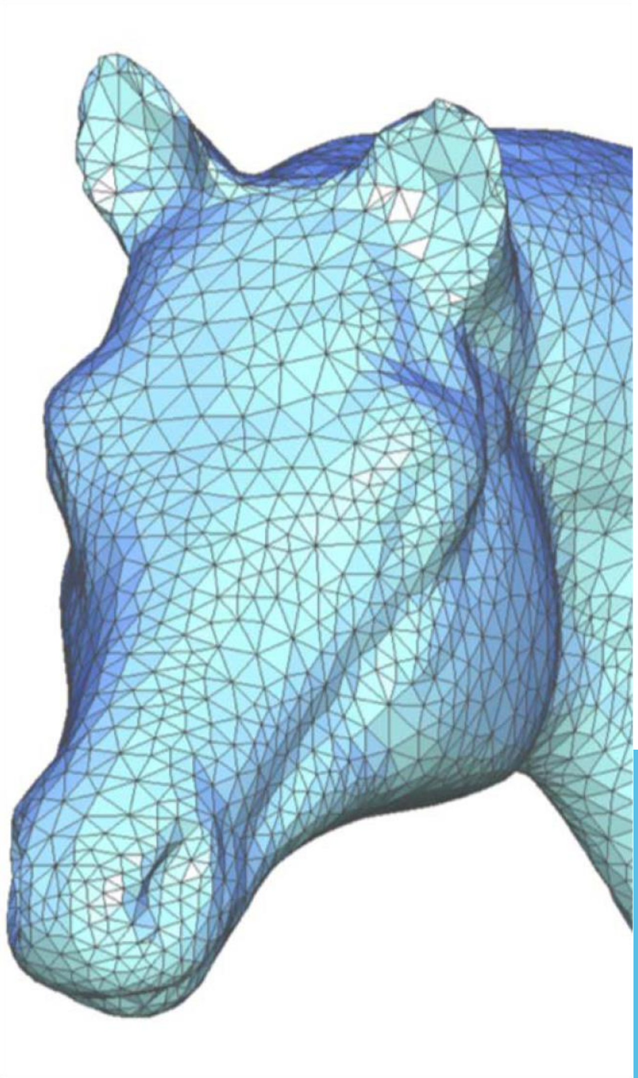
Map a 3D viewing volume
to an origin centered cube
of side length 2.

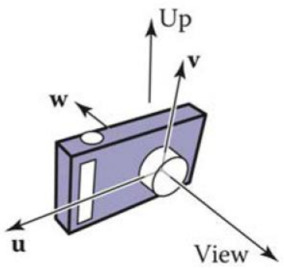
Translate(-(l+r)/2,-(t+b)/2,-(n+f)/2)) *
Scale(2/(r-l), 2/(t-b), 2/(f-n))

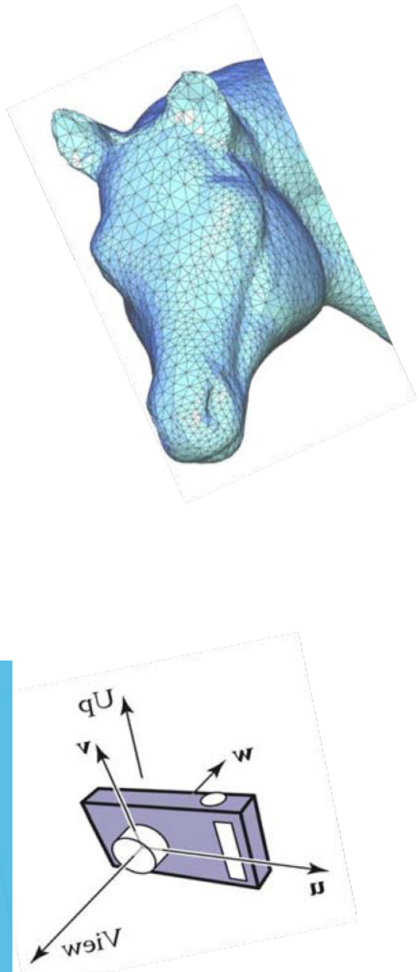
Getting Things Onto The Screen

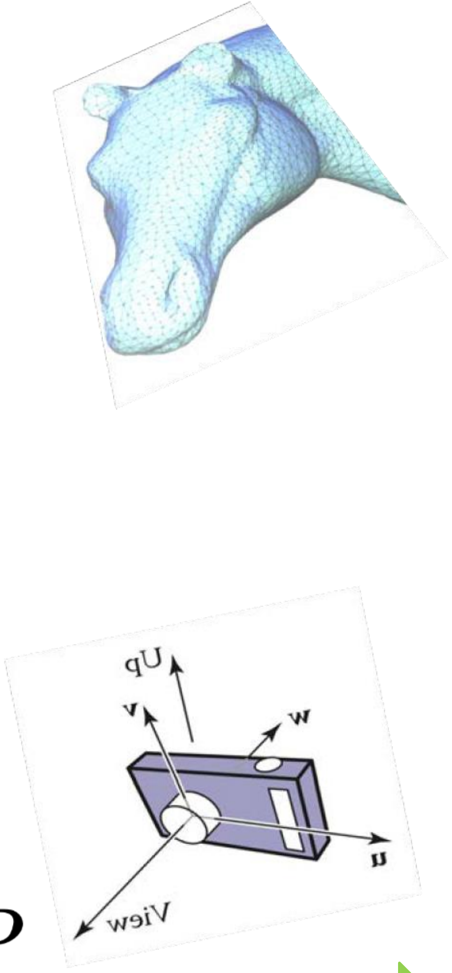


Getting Things Onto The Screen









OpenGL combines these into the ModelView Matrix

M V P

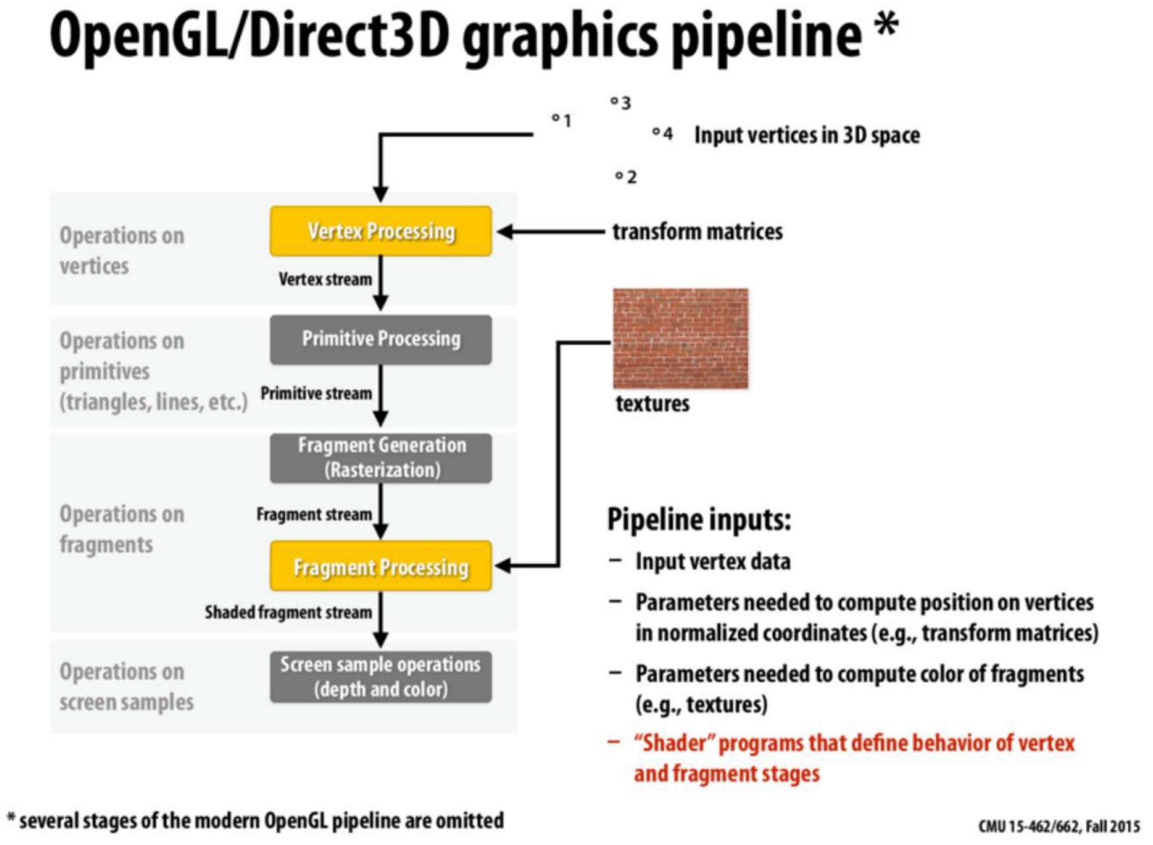
Object Space → World Space → Camera Space → Canonical Space

https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/projection-stage

Modern Graphics Pipeline

Role of CPU

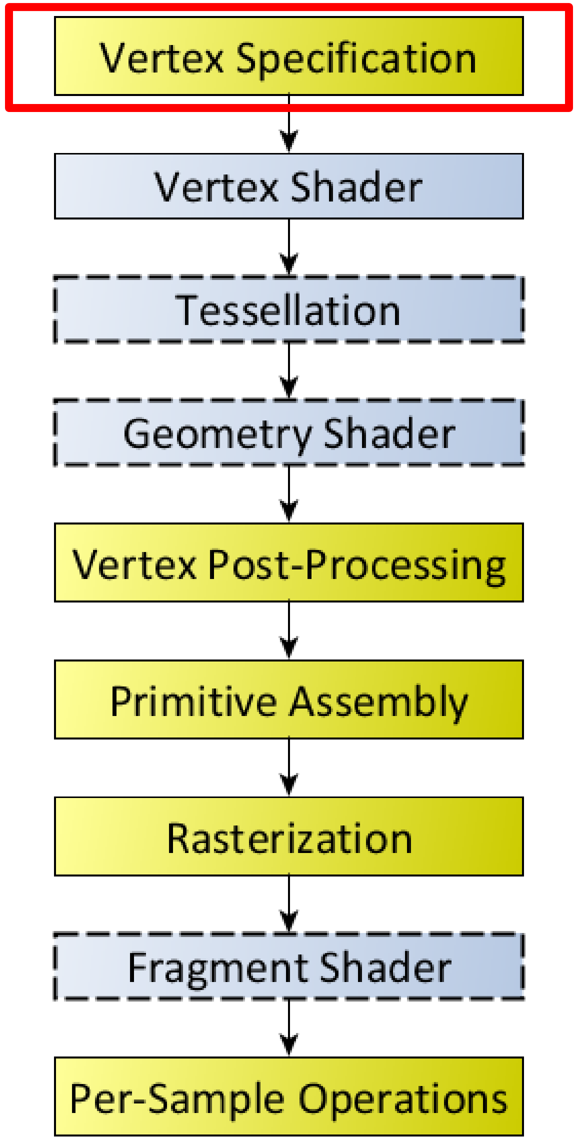
```
main()
  initialize window
  copy mesh vertex positions V and face indices F to GPU
  while window is open
    if shaders have not been compiled or files have changed
      compile shaders and send to GPU
    send "uniform" data to GPU
    set all pixels to background color
    tell GPU to draw mesh
    sleep a few milliseconds
```



Modern Graphics Pipeline

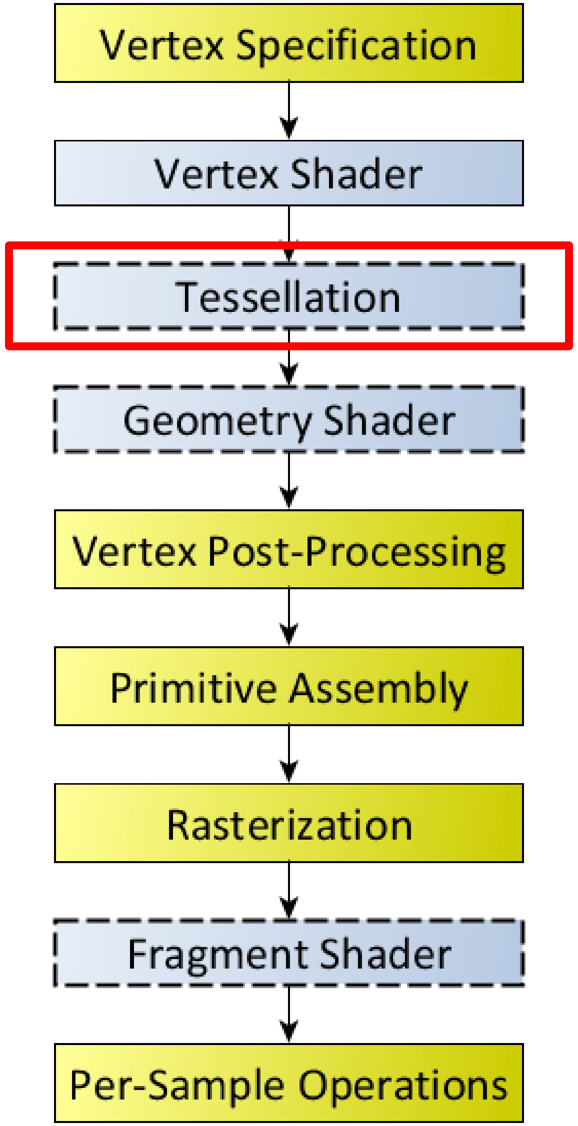
A Vertex Array Object (VAO) is an object which contains one or more Vertex Buffer Objects (VBO), designed to represent a complete rendered object. For eg. this is a diamond consisting of four vertices as well as a color for each vertex.

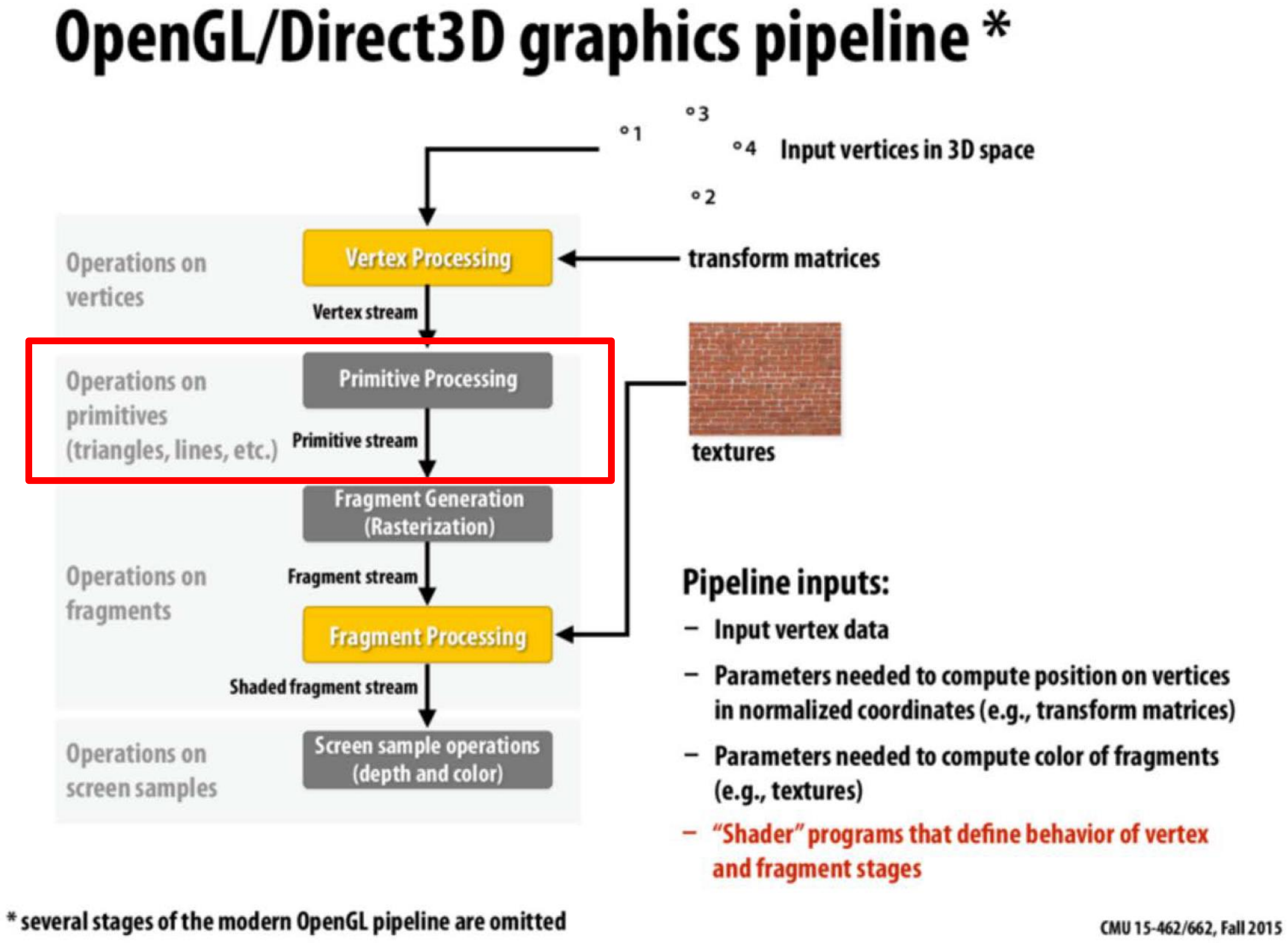
```
/* We're going to create a simple diamond made from lines */
const GLfloat diamond[4][2] = {
{ 0.0, 1.0 }, /* Top point */
{ 1.0, 0.0 }, /* Right point */
{ 0.0, -1.0 }, /* Bottom point */
{ -1.0, 0.0 } }; /* Left point */

const GLfloat colors[4][3] = {
{ 1.0, 0.0, 0.0 }, /* Red */
{ 0.0, 1.0, 0.0 }, /* Green */
{ 0.0, 0.0, 1.0 }, /* Blue */
{ 1.0, 1.0, 1.0 } }; /* White */
```



https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL)

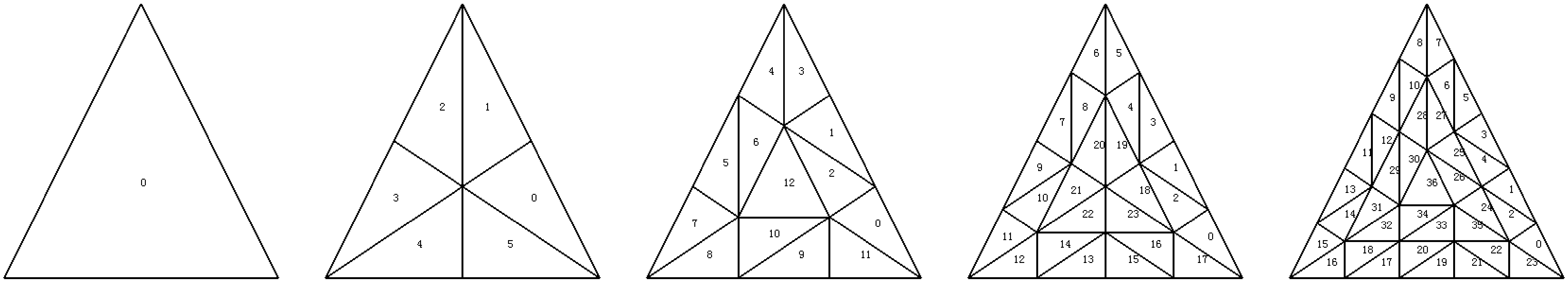
Modern Graphics Pipeline





https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

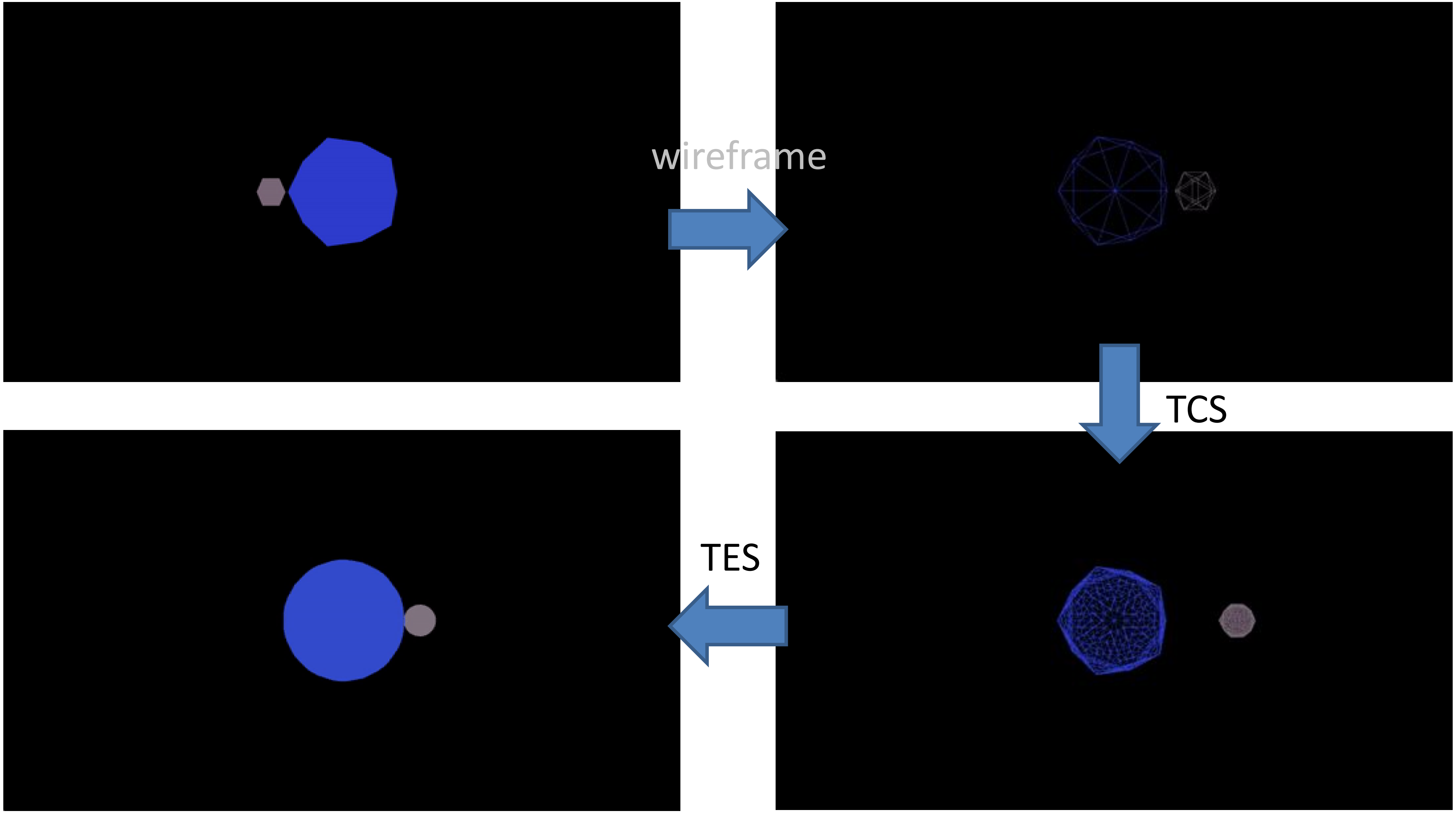
Tessellation Control Shader



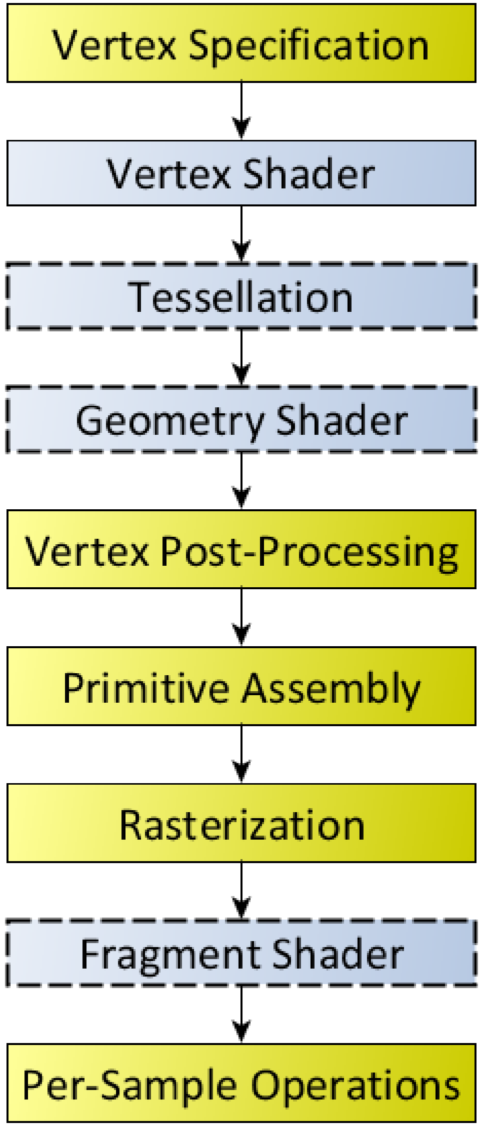
Level 1 Level 2 Level 3 Level 4 Level 5

https://erkaman.github.io/posts/tess_opt.html

Tessellation Evaluation Shader



Modern Graphics Pipeline

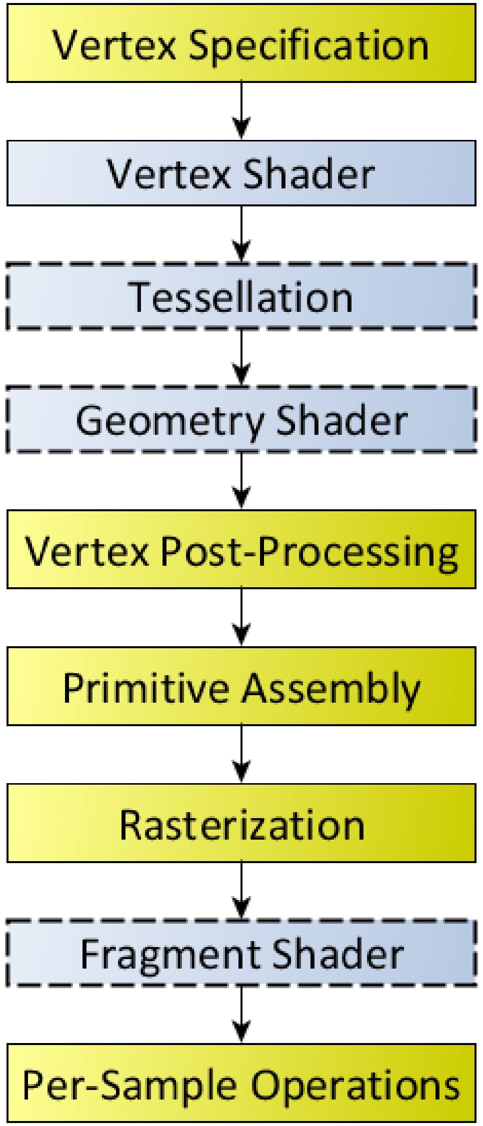


Clipping
Perspective divide and viewport transform to window space

After the perspective projection matrix multiplication in the vertex shader, we need to divide all x,y,z values by w to get them into normalized device coordinates. Then puts things into window space correctly.

https://www.learnopengles.com/tag/perspective-divide/
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

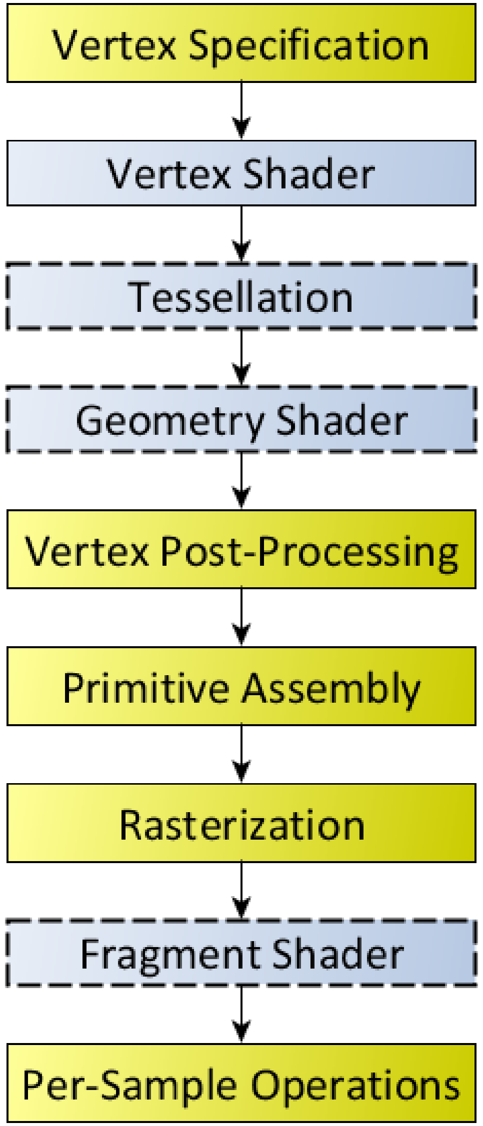
Modern Graphics Pipeline



Fragment generation from primitives
At least one fragment for every pixel area covered by primitive being rasterized

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

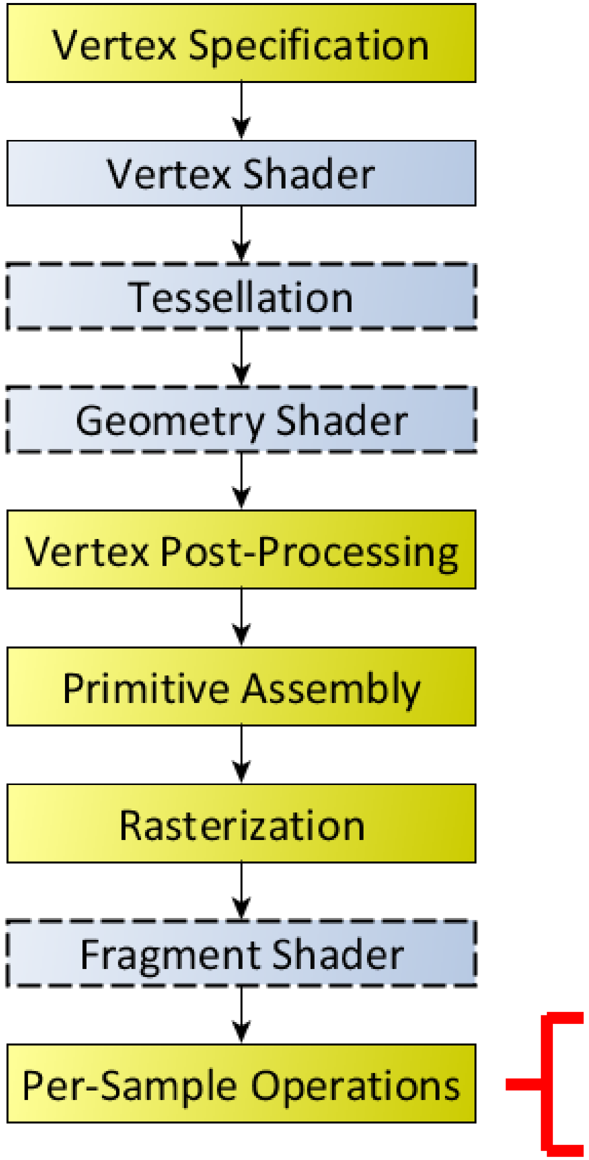
Modern Graphics Pipeline



Programmable shader operates on every fragment
Each output fragment has window space position, depth value, and and zero or more color values

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

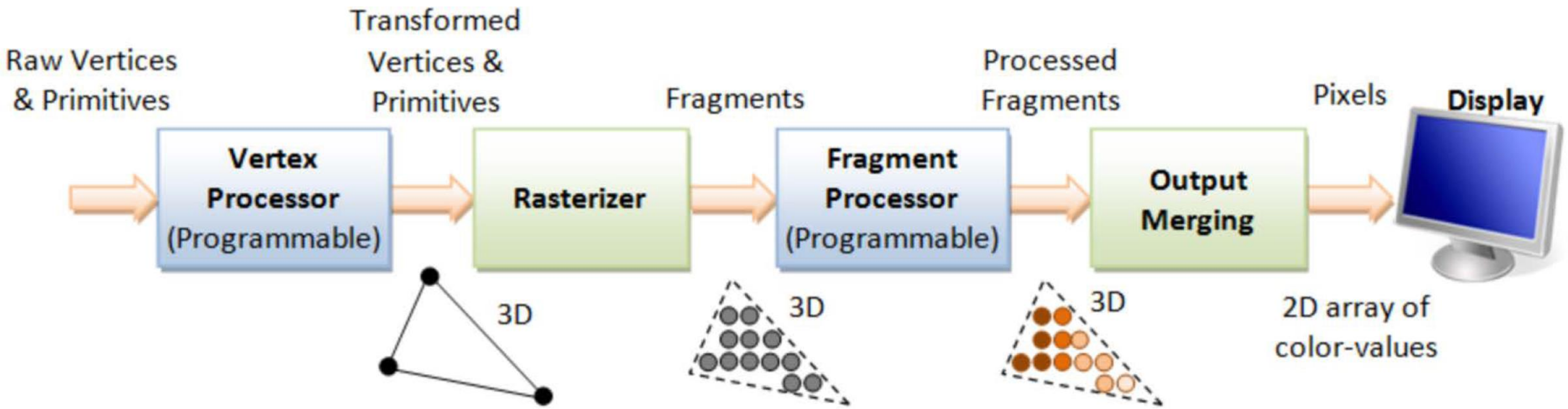
Modern Graphics Pipeline



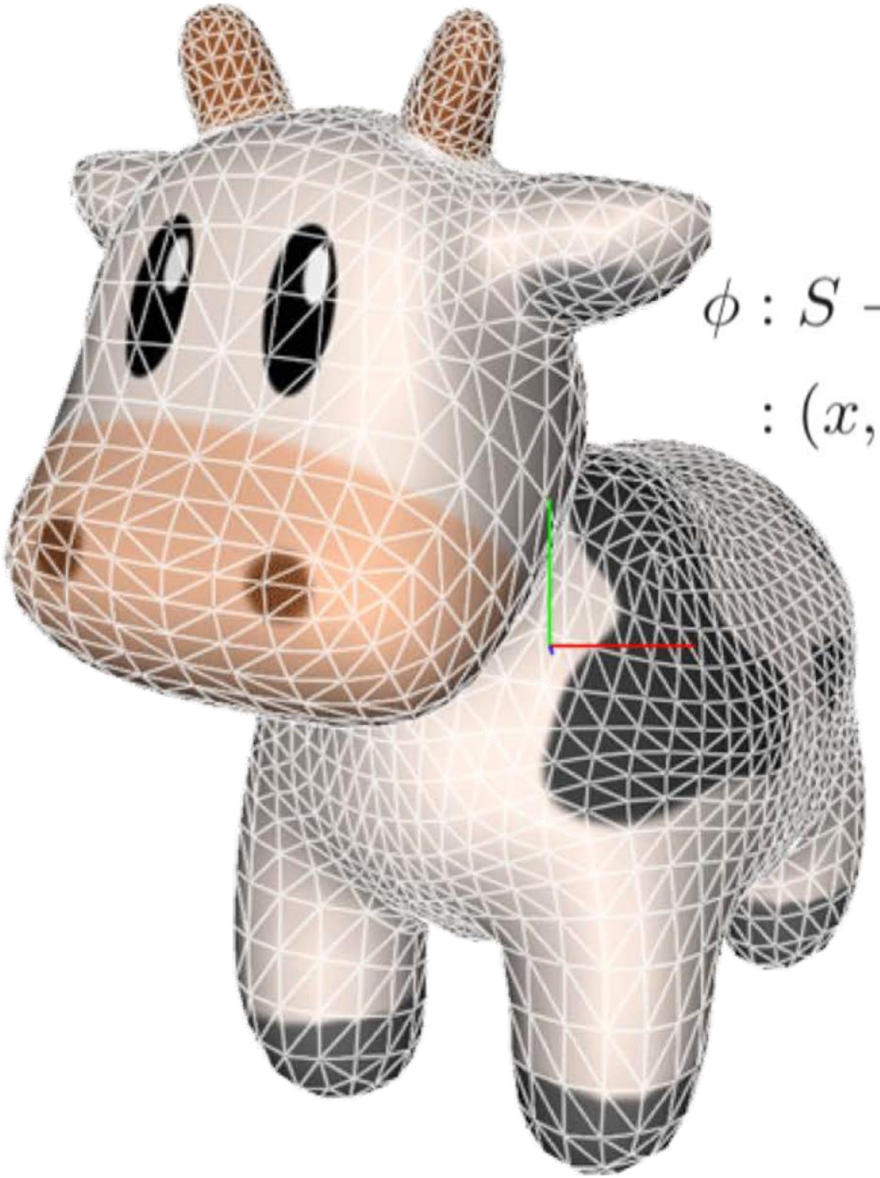
Depth culling, masking, etc.

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

Fragment Shader

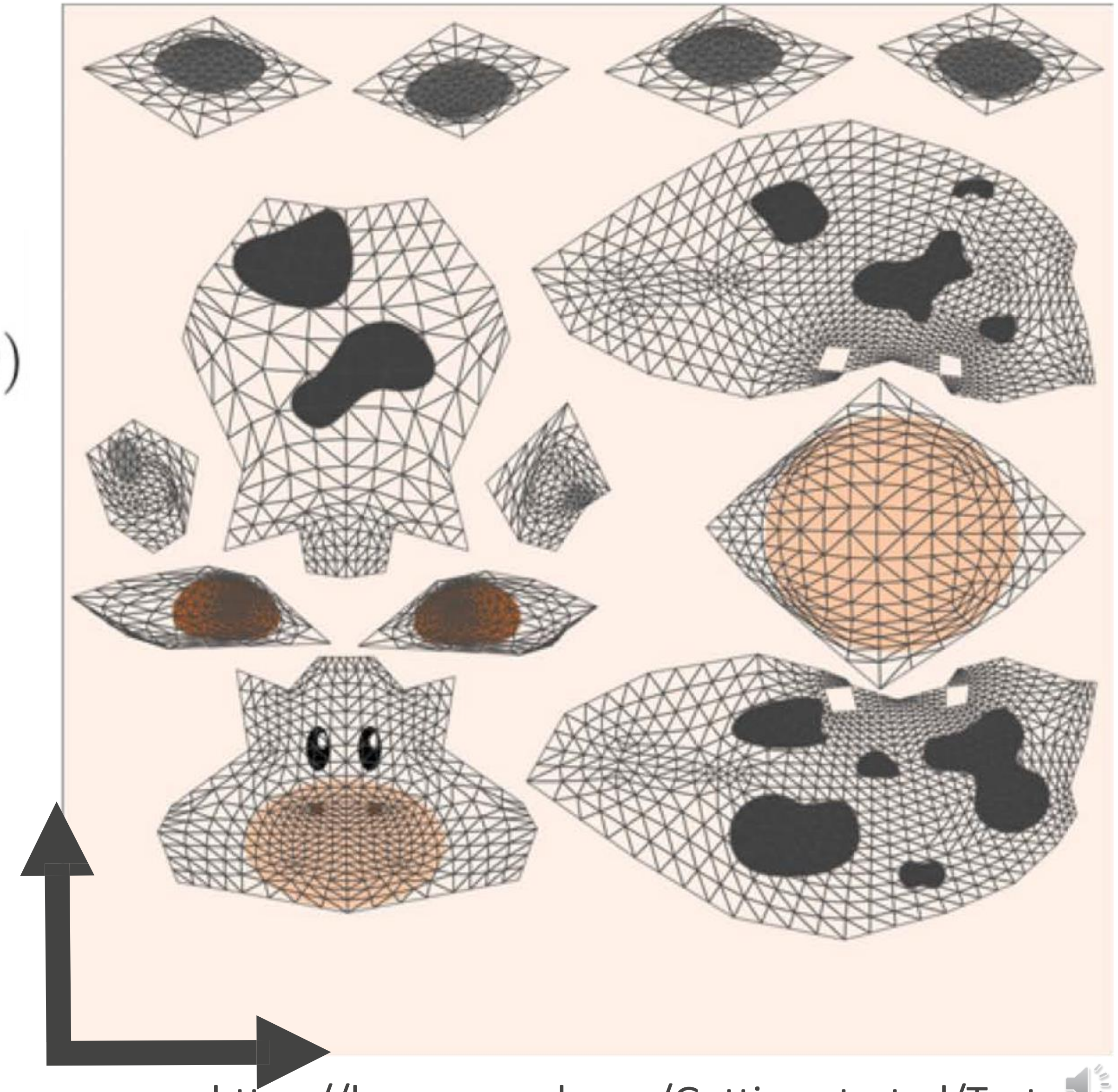


Texture coordinates



$$\phi : S \to T$$

$$: (x, y, z) \mapsto (u, v)$$

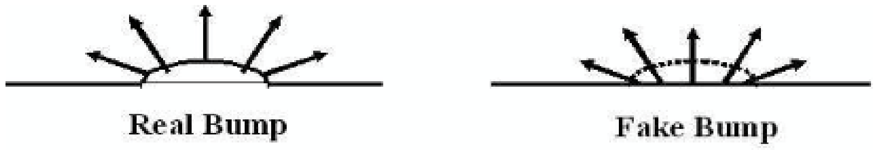


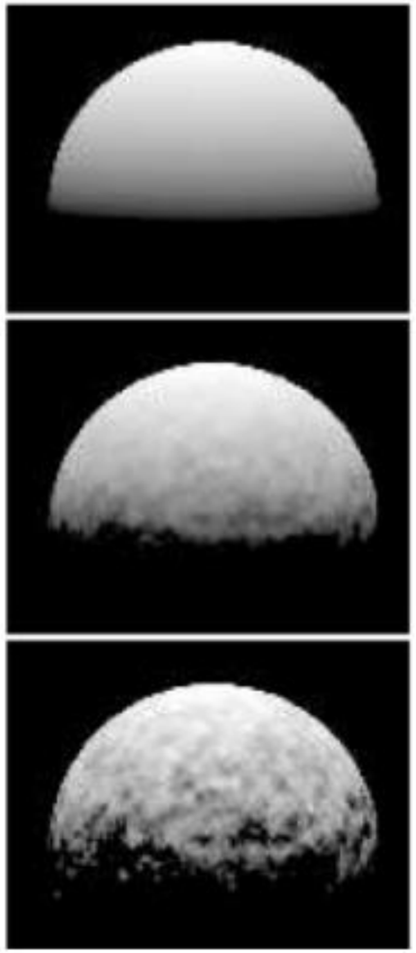
https://learnopengl.com/Getting-started/Textures

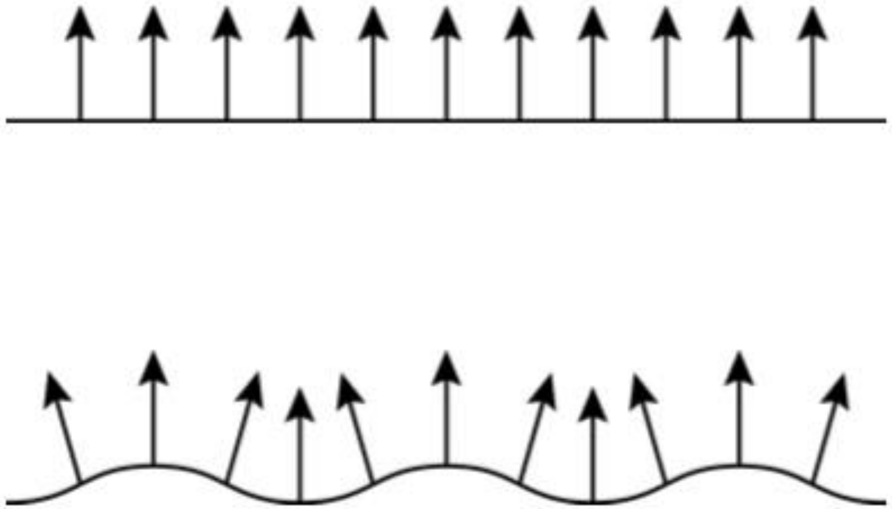
Bump | Normal Mapping

One of the reasons why we apply texture mapping:

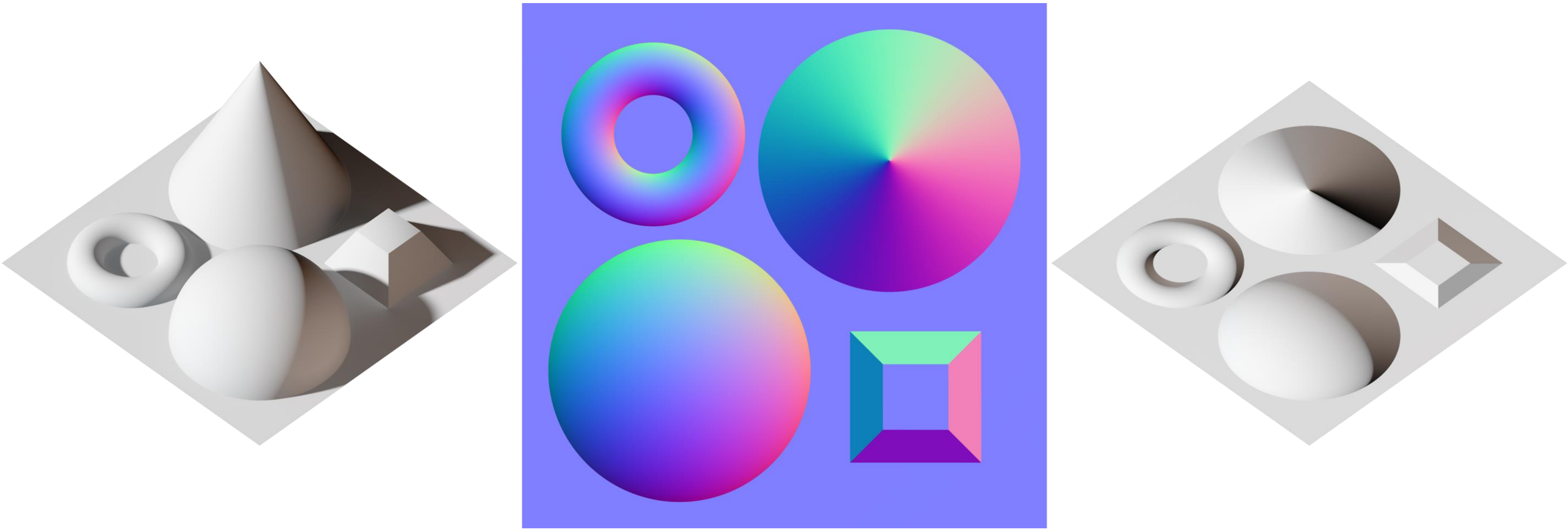
Real surfaces are hardly flat but often rough and bumpy. These bumps cause (slightly) different reflections of the light.



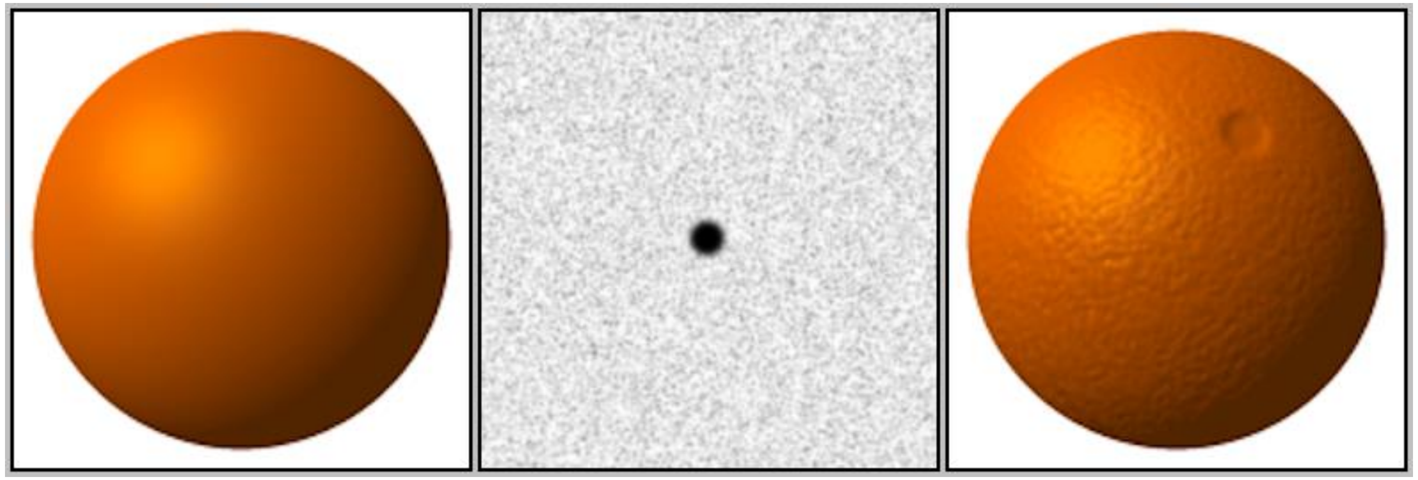




Normal Mapping



Bump Mapping



Bump map B(u,v)

Compute the perceived surface normal for the surface displaced by the bump value **B(u,v)** along its normal **n(u,v)**.

Displaced surface point $\mathbf{p_d(u,v) = p(u,v) + B(u,v)*n(u,v)}$

Displaced surface normal $\mathbf{n_d(u,v) = p'^u_d(u,v) \ X \ \ p'^v_d(u,v)}$

Bump Mapping

Compute the perceived surface normal for the surface displaced by the bump value **B(u,v)** along its normal **n(u,v)**.

Displaced surface point $\mathbf{p_d(u,v) = p(u,v) + B(u,v)*n(u,v)}$

Displaced surface normal $\mathbf{n_d(u,v) = p'^u_d(u,v) \; X \; p'^v_d(u,v)}$

B(u,v) is small

$$\mathbf{p'^u_d(u,v) = p'^u(u,v) + B'^u(u,v)*n(u,v) + \cancel{B(u,v)*n'^u(u,v)}}$$

$$\mathbf{n_d = (p'^u + B'^u\, n) \; X \; (p'^v + B'^v\, n)} \qquad \text{// not showing (u,v)}$$

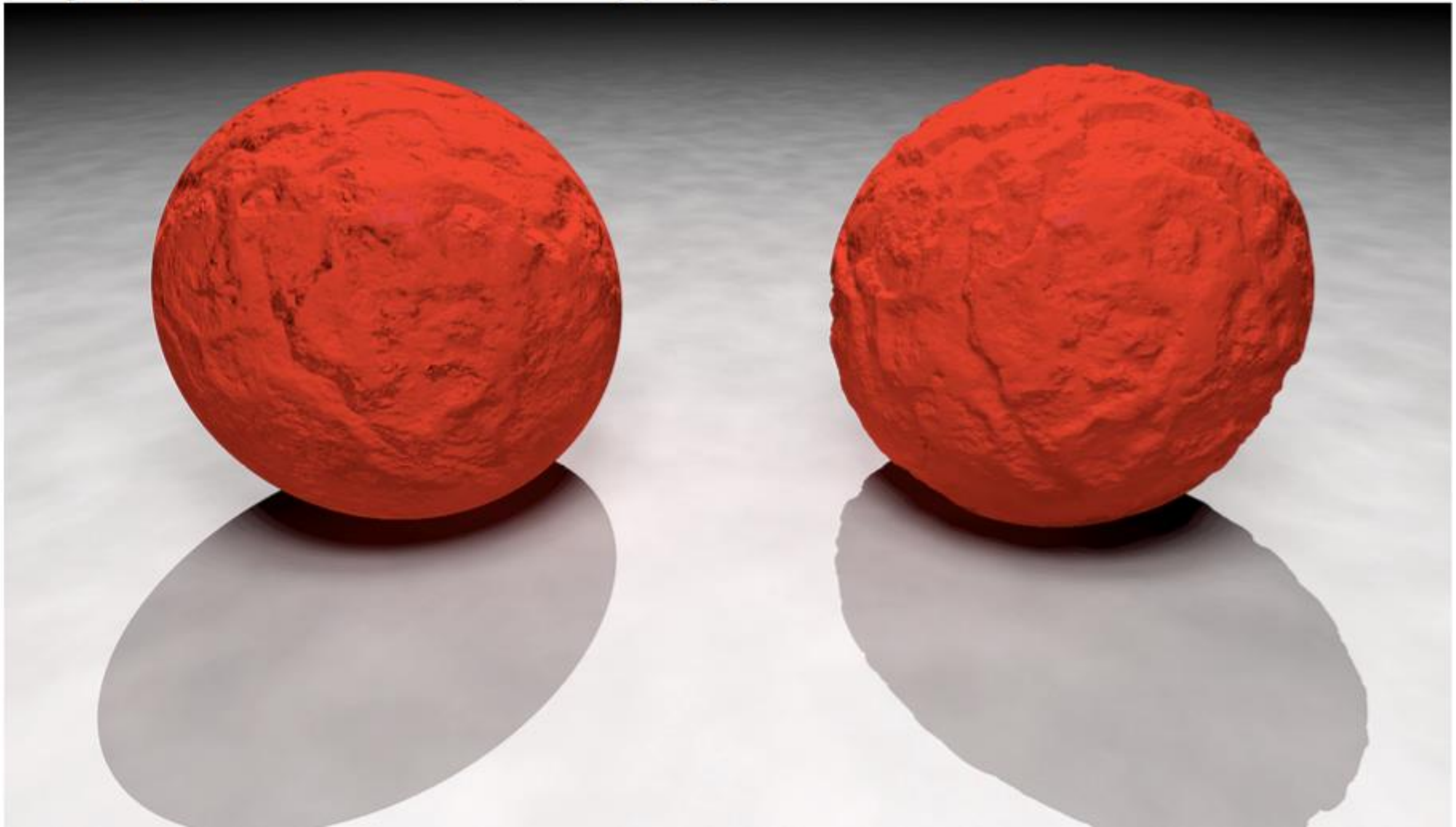
$$\mathbf{n_d = p'^u \; X \; p'^v + p'^u \; X \; n \, B'^v + B'^u \, n \; X \; p'^v + B'^u B'^v \, n \; X \; n}$$

$$\mathbf{n_d = n + B'^v (p'^u \; X \; n) + B'^u (n \; X \; p'^v)}$$

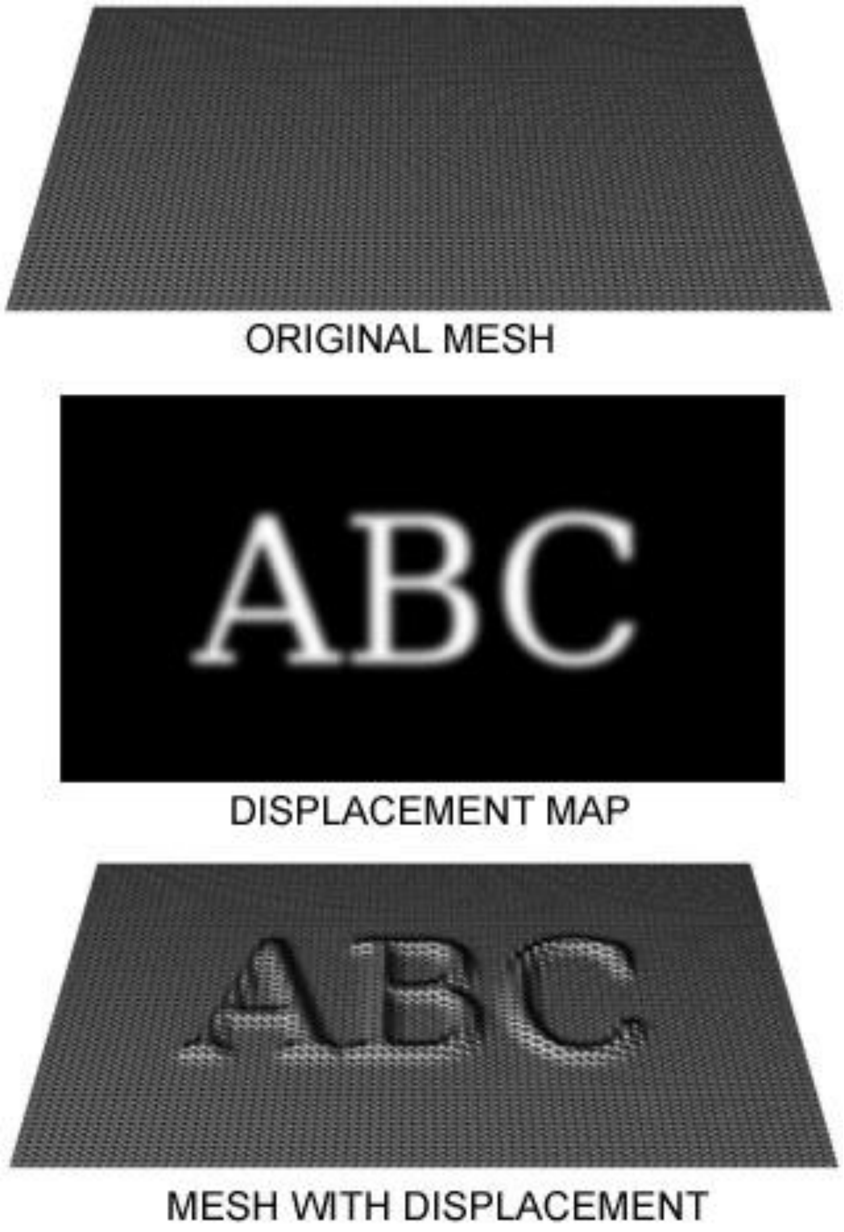
B’ can be precomputed by finite difference and stored as a texture.
p’ can be computed analytically or by finite difference.

Bump | Normal Mapping vs. Geometry

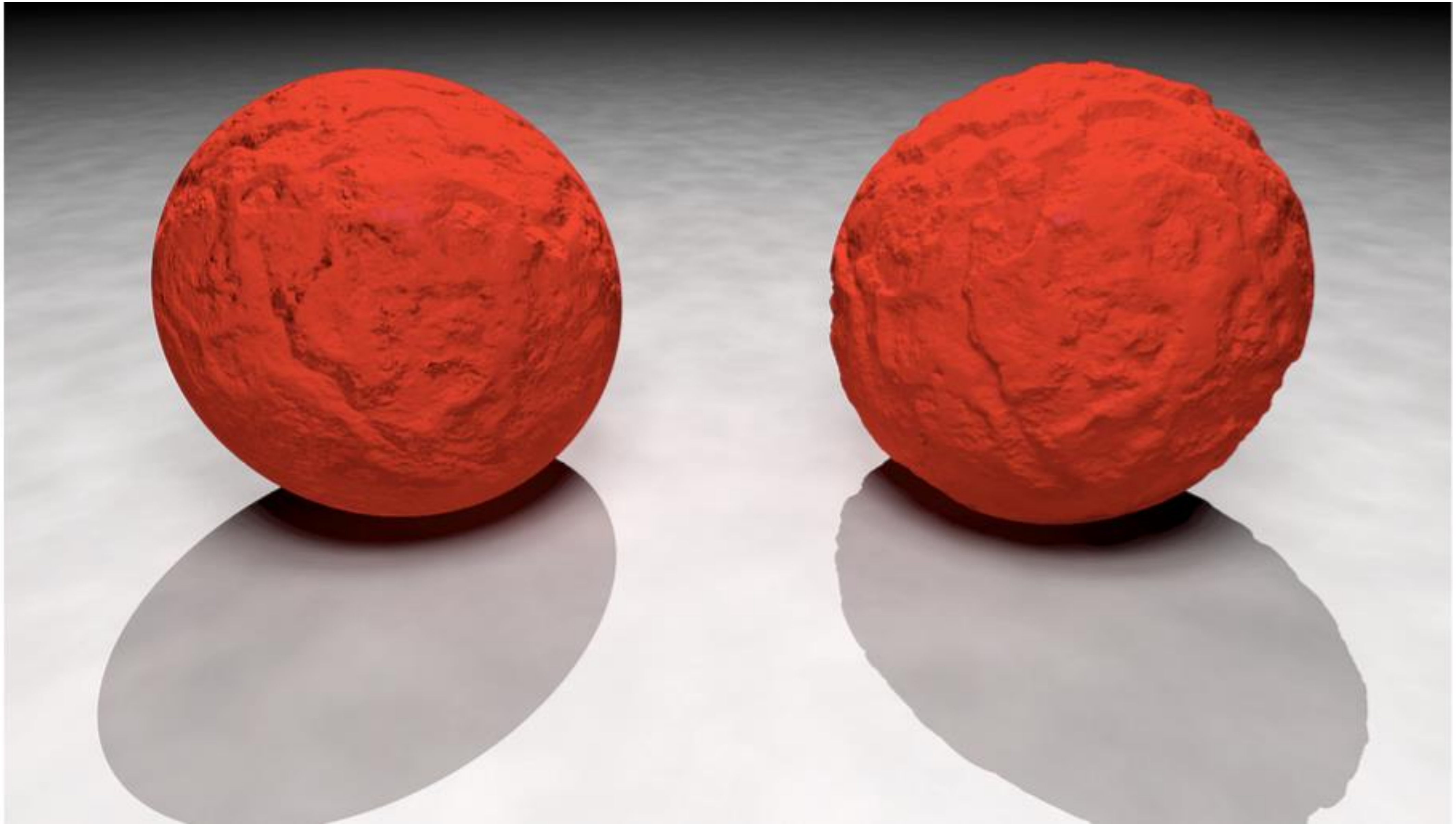
Major problems with bump mapping: silhouettes and shadows



Displacement (Bump) Mapping



Bump | Normal Mapping vs. Displacement Mapping



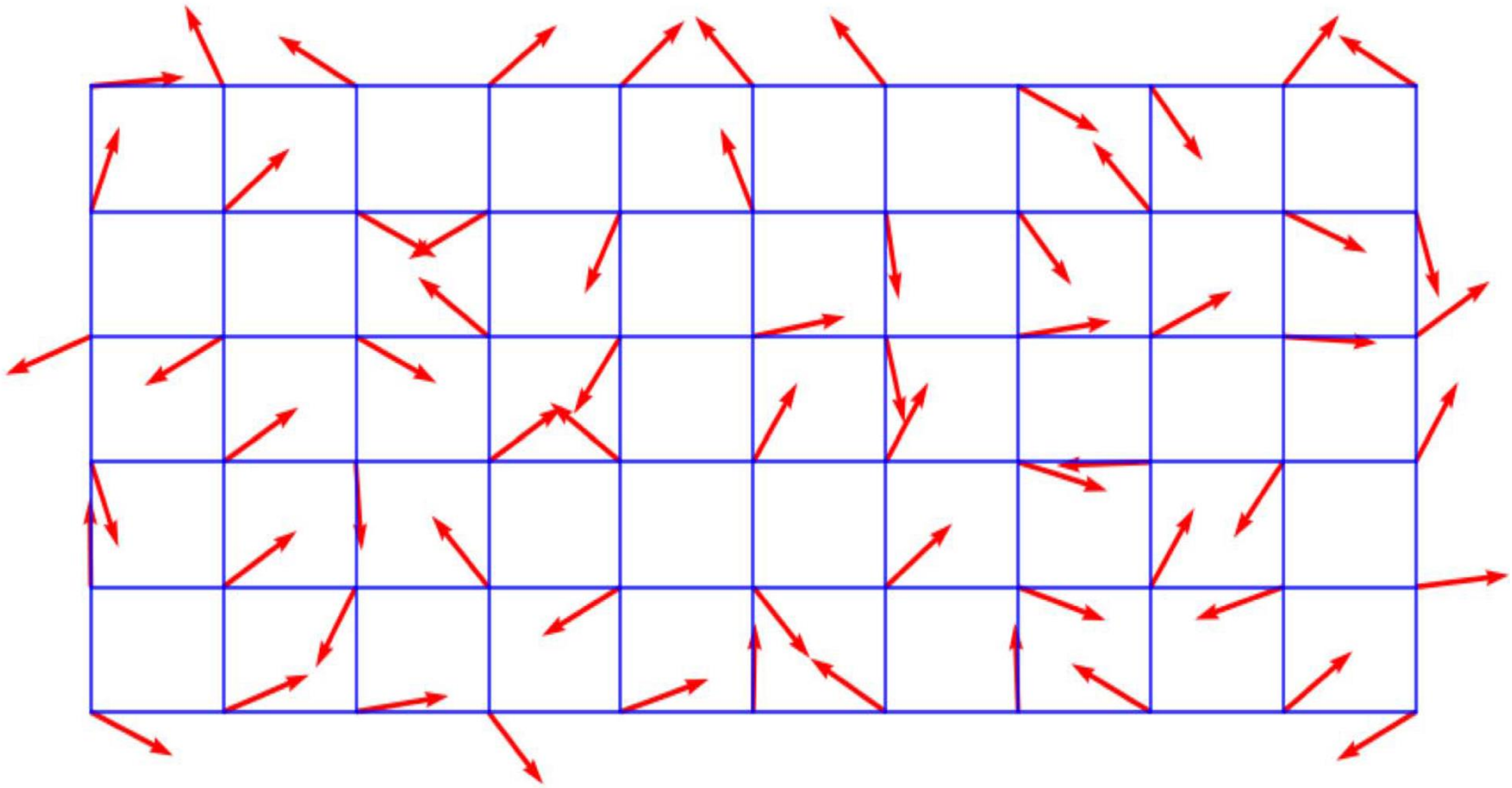
Perlin Noise

Properties of noise:

- well-defined everywhere in space.
- It randomly varies between -1 and 1.
- Its frequency is band-limited.

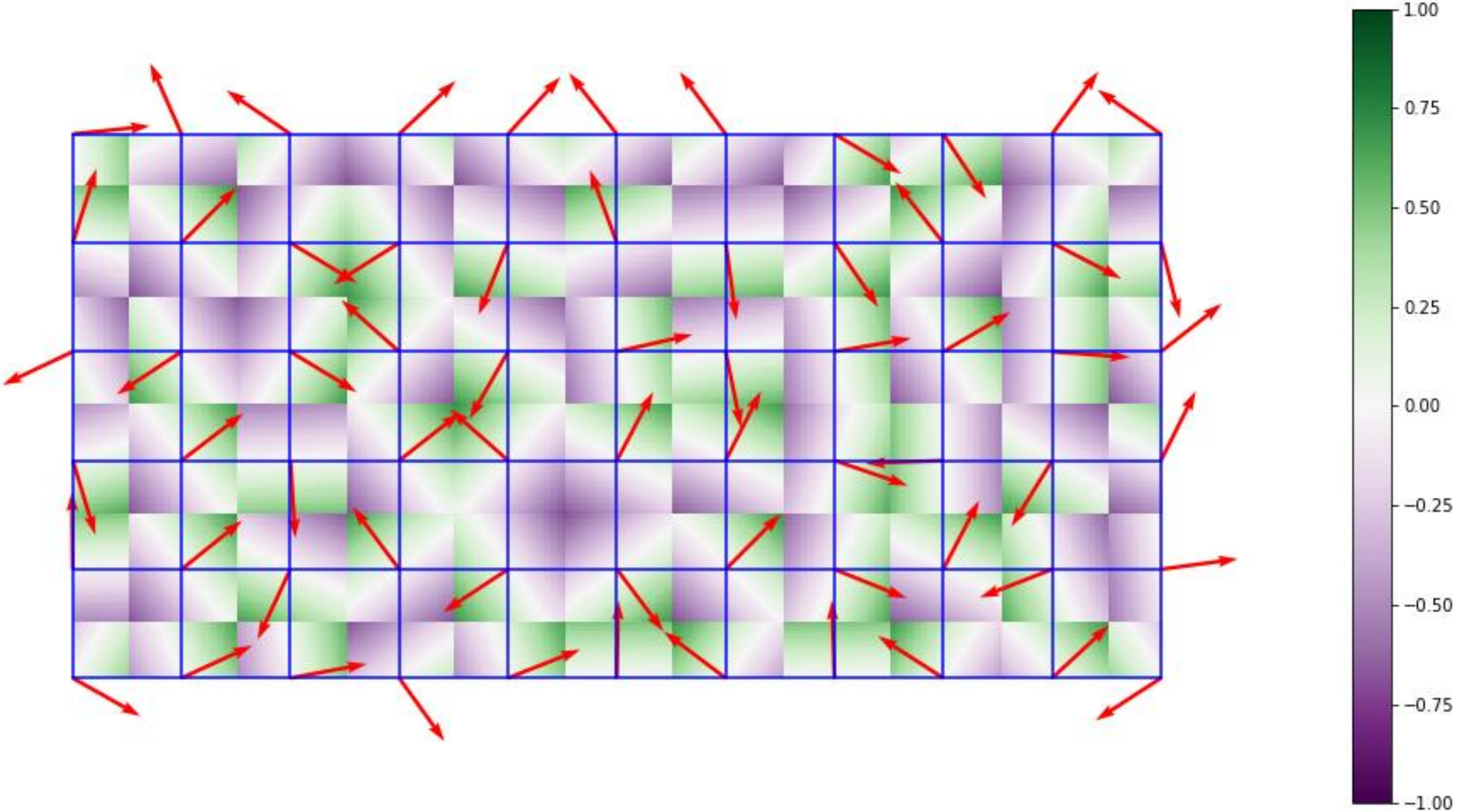
Gradient Noise in 1D:

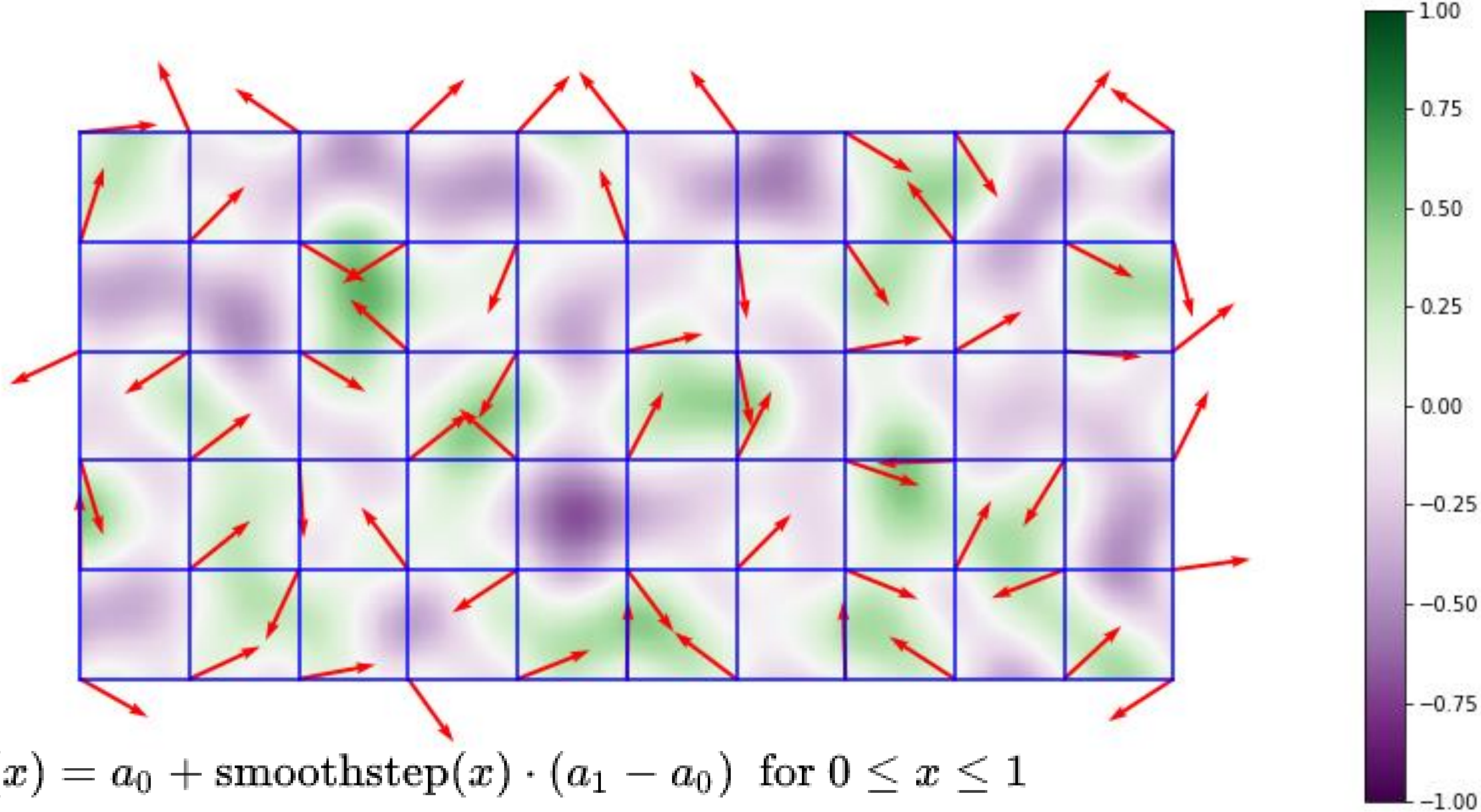
- Define a pseudo-random gradient direction $\mathbf{d_i}$ at each integer point **i**.
- The value at a point **x** relative to an integer neighbour **i** is $\mathbf{(x\text{-}i).\ d_i}$
- Smoothly interpolate between neighboring values for **n(x).**



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

https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise

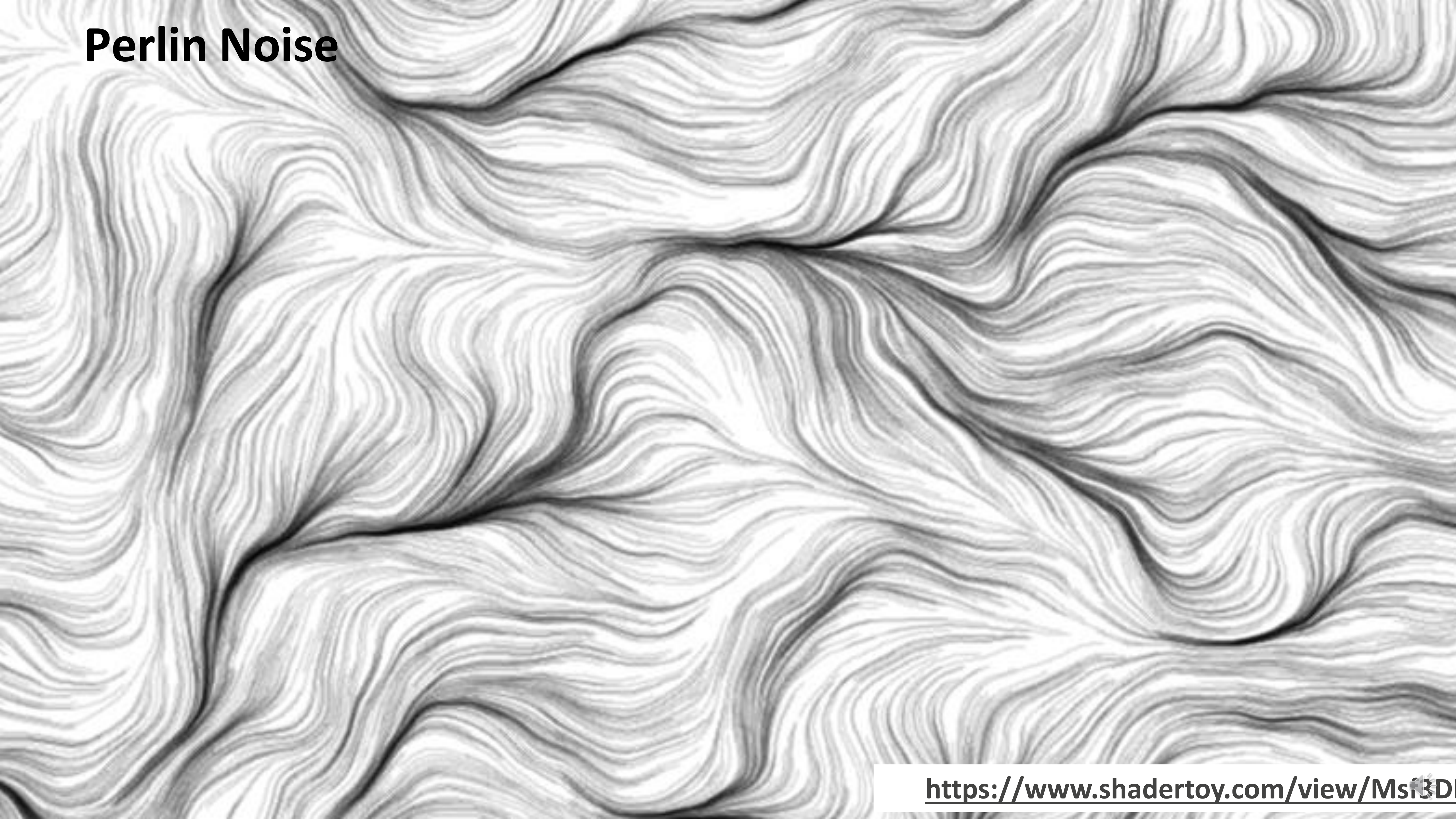

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



$$f(x) = a_0 + \mathrm{smoothstep}(x) \cdot (a_1 - a_0) \ \ \text{for } 0 \leq x \leq 1$$

$$\mathrm{smoothstep}(x) = S_1(x) = \begin{cases} 0 & x \leq 0 \\ 3x^2 - 2x^3 & 0 \leq x \leq 1 \\ 1 & 1 \leq x \end{cases}$$

Perlin Noise



https://www.shadertoy.com/view/Msf3D

Next: Animation