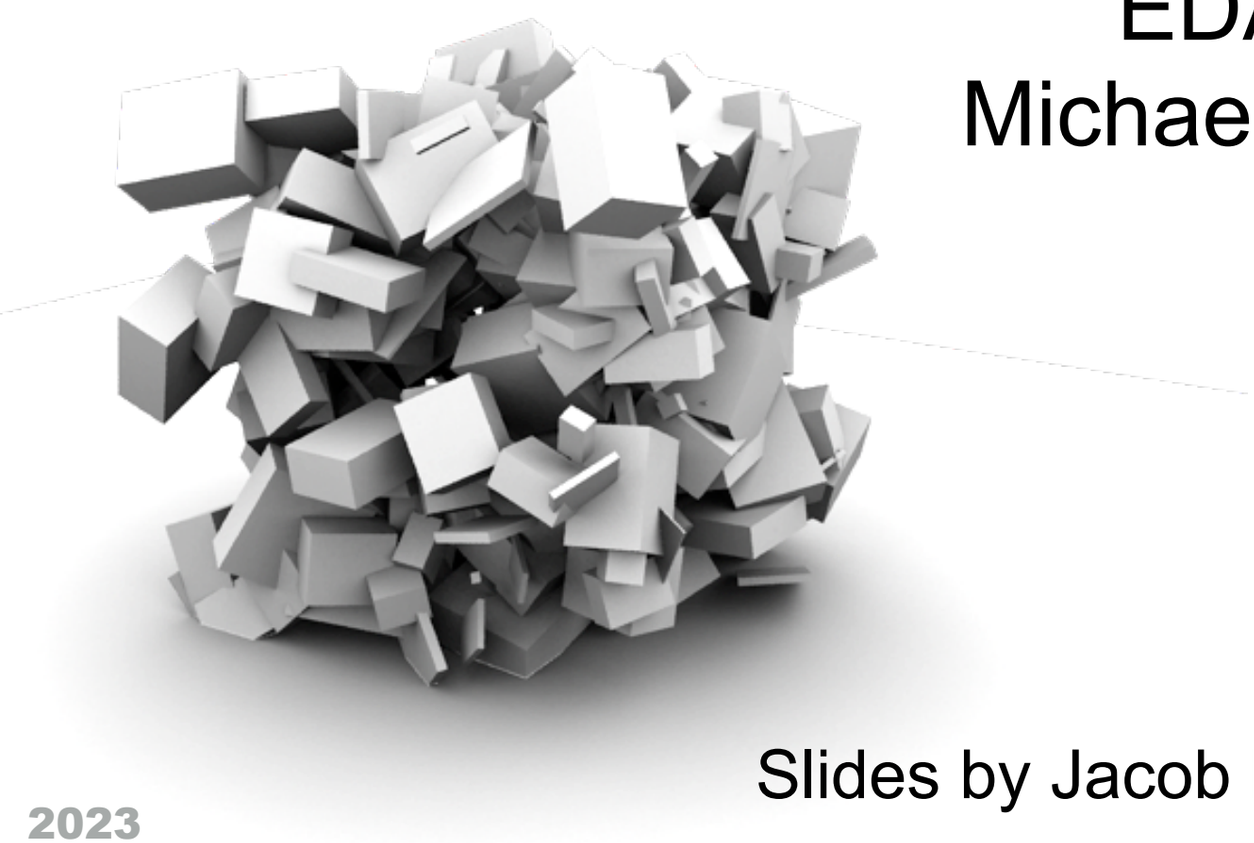
Viewing

EDAF80
Michael Doggett



Slides by Jacob Munkberg 2012-13

2023

Today

- Camera setup
- Viewing and Projection
- Procedural Techniques

Transforms

http://www.realtimerendering.com/udacity/transforms.html

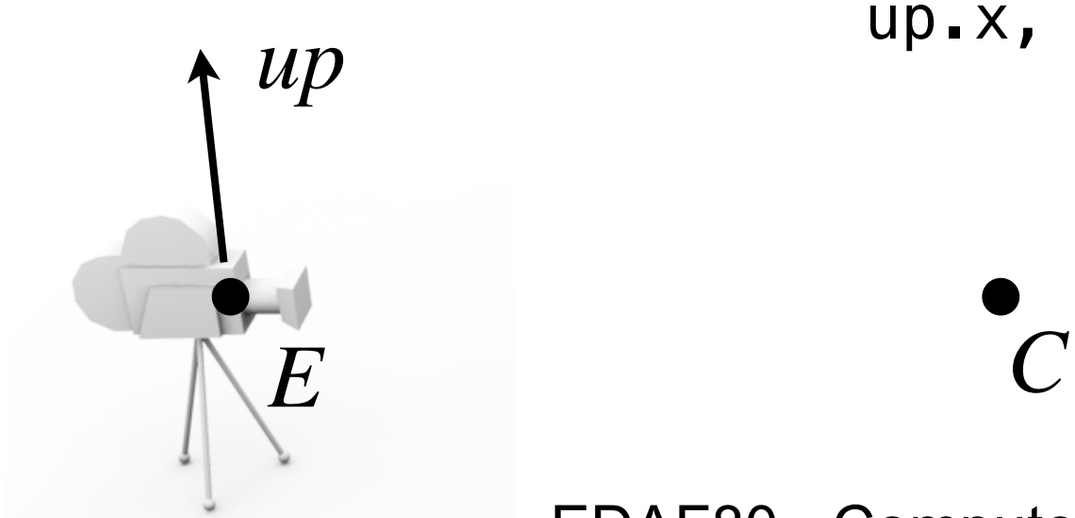
Task at Hand

- Setup an OpenGL camera
- Find matrix that transforms an arbitrary camera to the origin, looking along the negative z axis

Setup Camera Matrix

- LookAt function
  - Takes eye position (E), a position to look at (C) and an up vector (up)
  - Constructs the **View** matrix, i.e., a matrix that transforms geometry (in world space) into the camera's coordinate system (camera space)

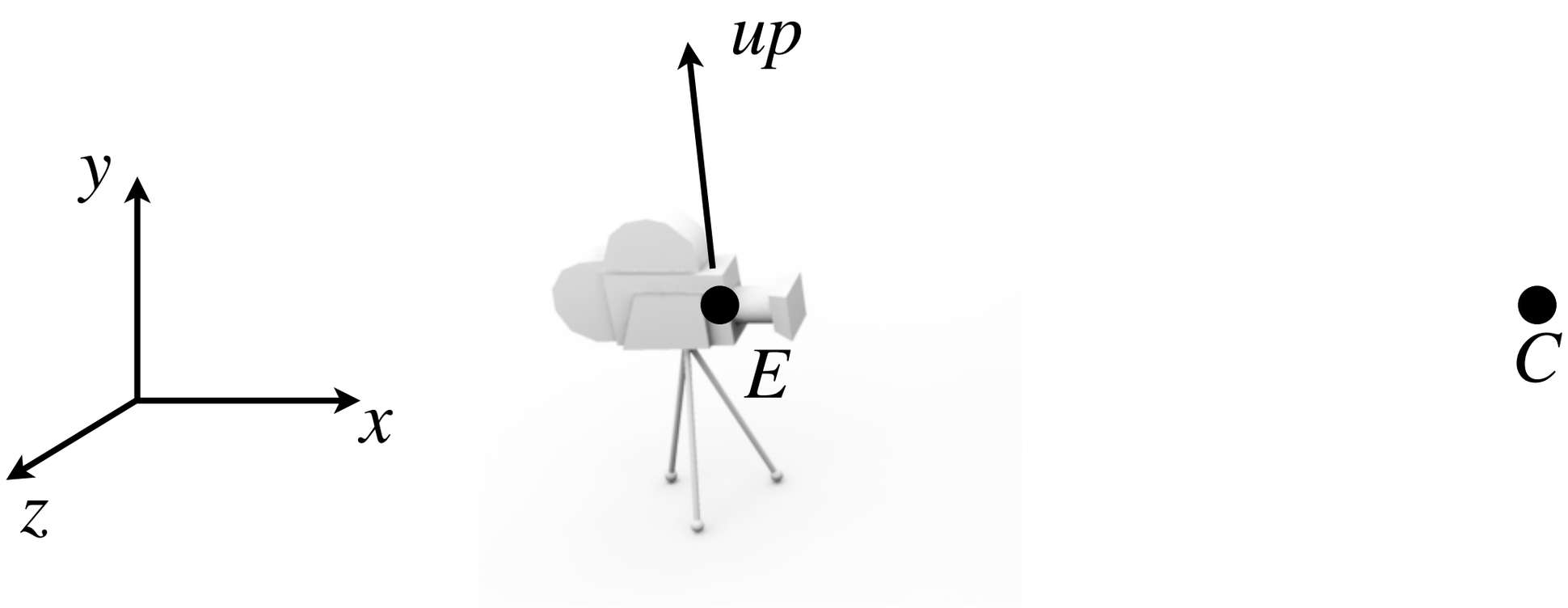
```
mat4 View = LookAt(E.x,E.y,E.z,          // Camera position
                   C.x,C.y,C.z,          // Center of interest
                   up.x, up.y, up.z);   // Up-vector
```



Camera Placement

Derivation from Ravi Ramamoorthi

- Specify camera position (E), center of interest (C) and up-vector (up)



OpenGL convention

- In OpenGL: right-hand coordinate system, looking down $-z$.

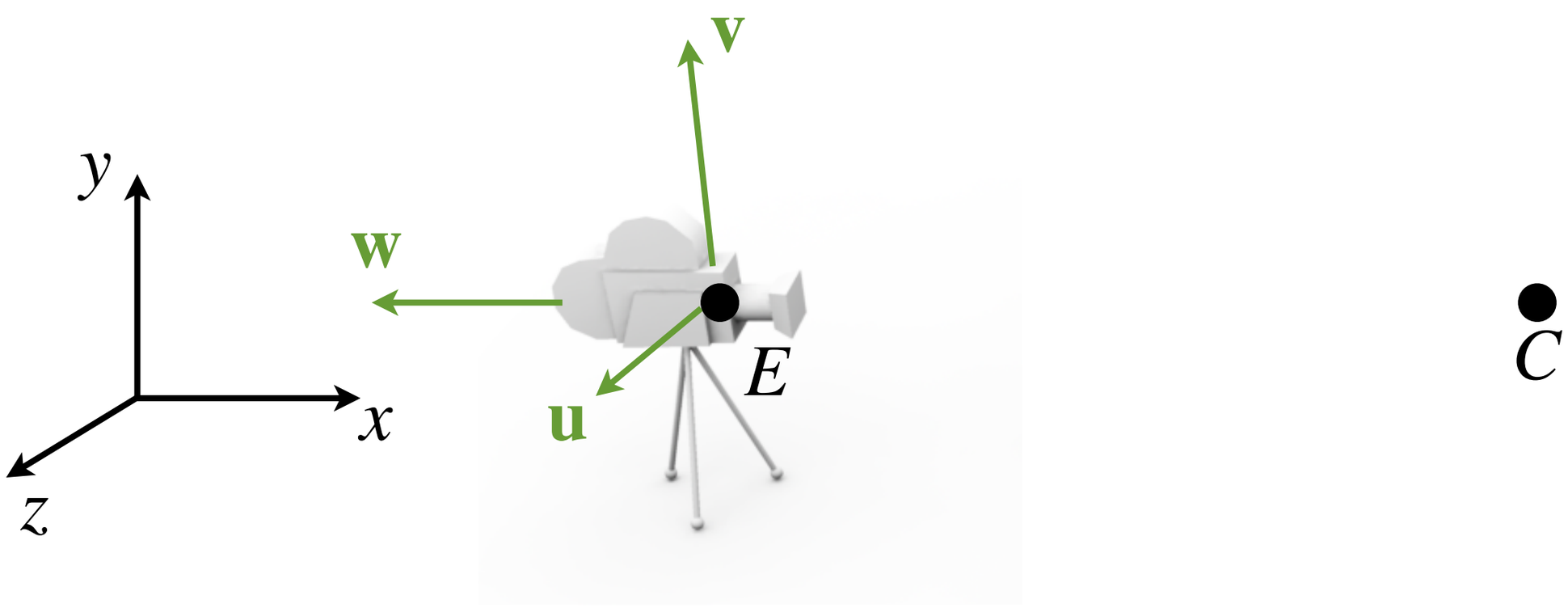
Find orthonormal basis

Derivation from Ravi Ramamoorthi

- OpenGL standard: camera looks along negative z. Choose $\mathbf{w}$ in direction -(C-E)

$$\mathbf{a} = E - C \qquad \mathbf{b} = up$$

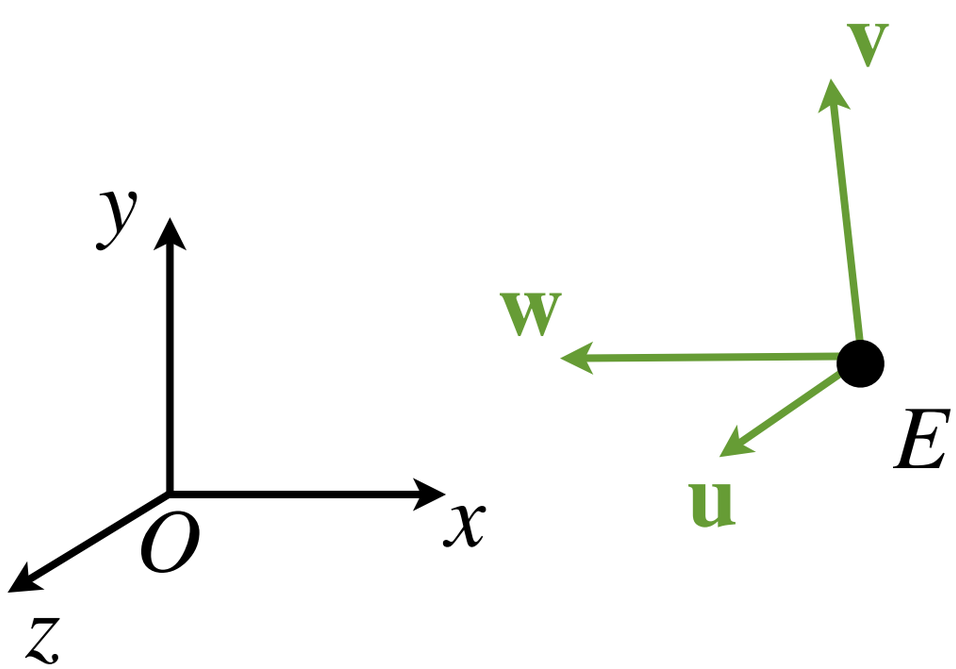
$$\mathbf{w} = \frac{\mathbf{a}}{|\mathbf{a}|} \qquad \mathbf{u} = \frac{\mathbf{b} \times \mathbf{w}}{|\mathbf{b} \times \mathbf{w}|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$



Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Now, we look for matrix that transforms frame $\{\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathrm{E}\}$ to $\{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathrm{O}\}$
- Translation and rotation

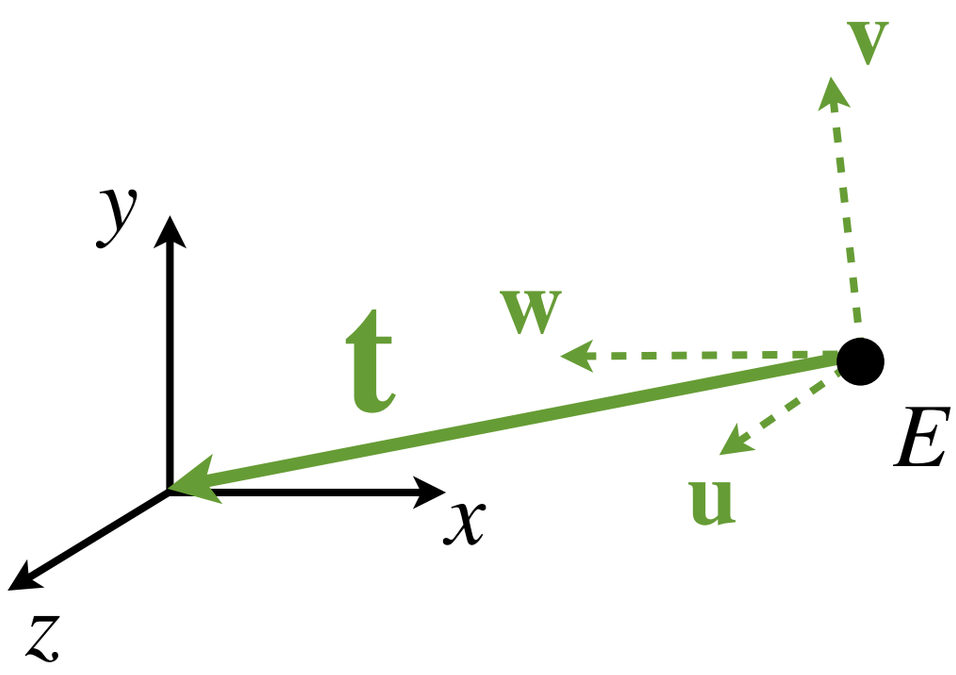


Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Translate $\mathbf{uvw}\mathrm{E}$ frame so that the origin align with the $\mathbf{xyz}\mathrm{O}$ frame

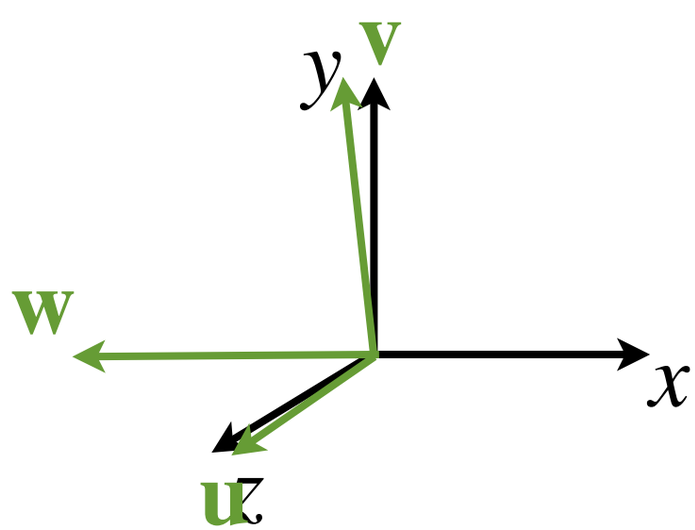
$$\mathbf{t} = \begin{bmatrix} -E_x \\ -E_y \\ -E_z \end{bmatrix}$$



Find orthonormal basis

Derivation from Ravi Ramamoorthi

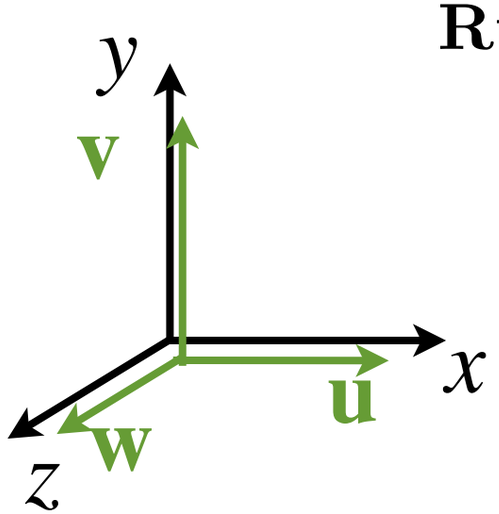
- Translate **uvw**E frame so that the origin align with the **xyz**O frame



Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Then rotate $\mathbf{uvw}$ basis so that the three axes align, $\mathbf{u}$ // $\mathbf{x}$, $\mathbf{v}$ // $\mathbf{y}$ and $\mathbf{w}$ // $\mathbf{z}$
- Rotation matrix given by $\mathbf{R} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix}$
- R rotates vectors $\mathbf{uvw}$ to $\mathbf{xyz}$



$$\mathbf{Ru} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix} \begin{bmatrix} | \\ \mathbf{u} \\ | \end{bmatrix} = \begin{bmatrix} \mathbf{u} \cdot \mathbf{u} \\ \mathbf{v} \cdot \mathbf{u} \\ \mathbf{w} \cdot \mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

$$\mathbf{Rv} = \mathbf{y}, \quad \mathbf{Rw} = \mathbf{z}$$

Camera Placement

Derivation from Ravi Ramamoorthi

- Combine the two transforms
- Move to center, and apply rotation

$$M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate Move to center

Workflow

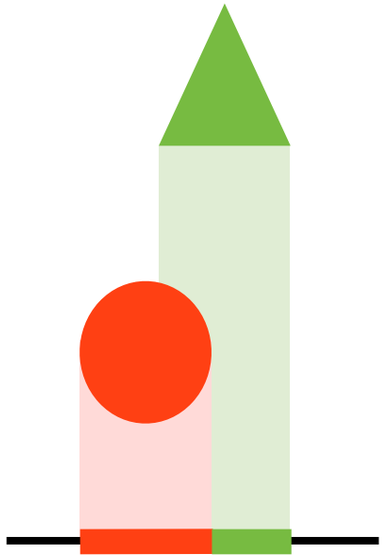
- OpenGL geometry workflow
  - Place camera in scene
  - Find **View** transform that moves camera to origin, looking along $-z$.
  - Place geometry in scene using **Model** (or **World**) transform
  - Setup camera **Projection** matrix (3D->2D)
  - Apply **ModelViewProjection** matrix to all geometry in the scene in vertex shader

Projection

- From 3D to a 2D image
  - Orthographic projection
  - Perspective projection
- Lines map to lines
  - Projective transform **does not** preserve parallel lines, angles or distances!

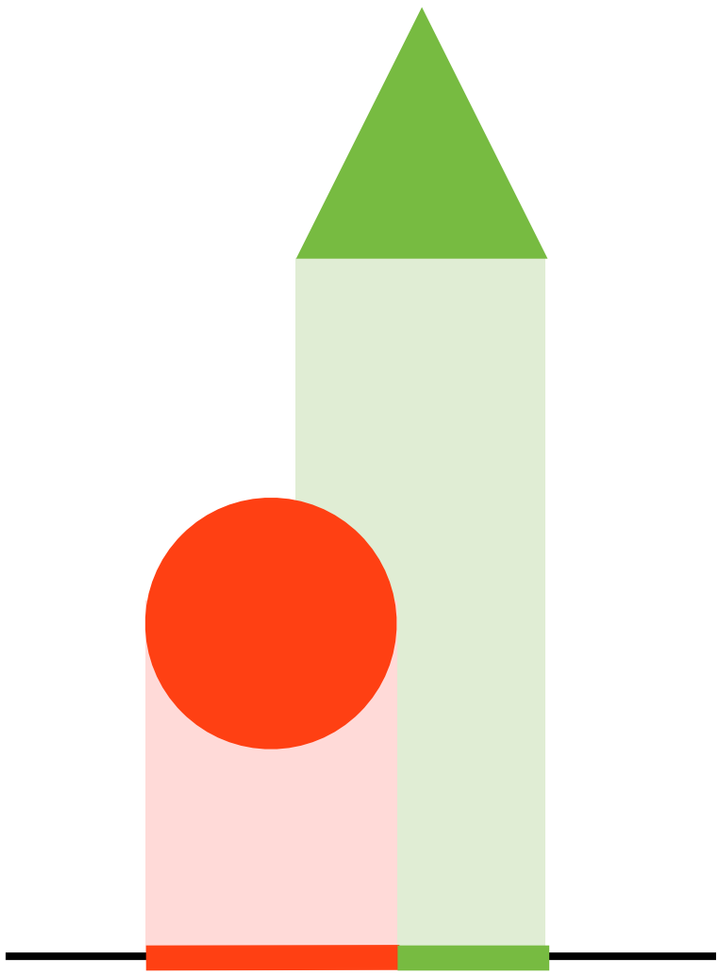
Orthographic Projection

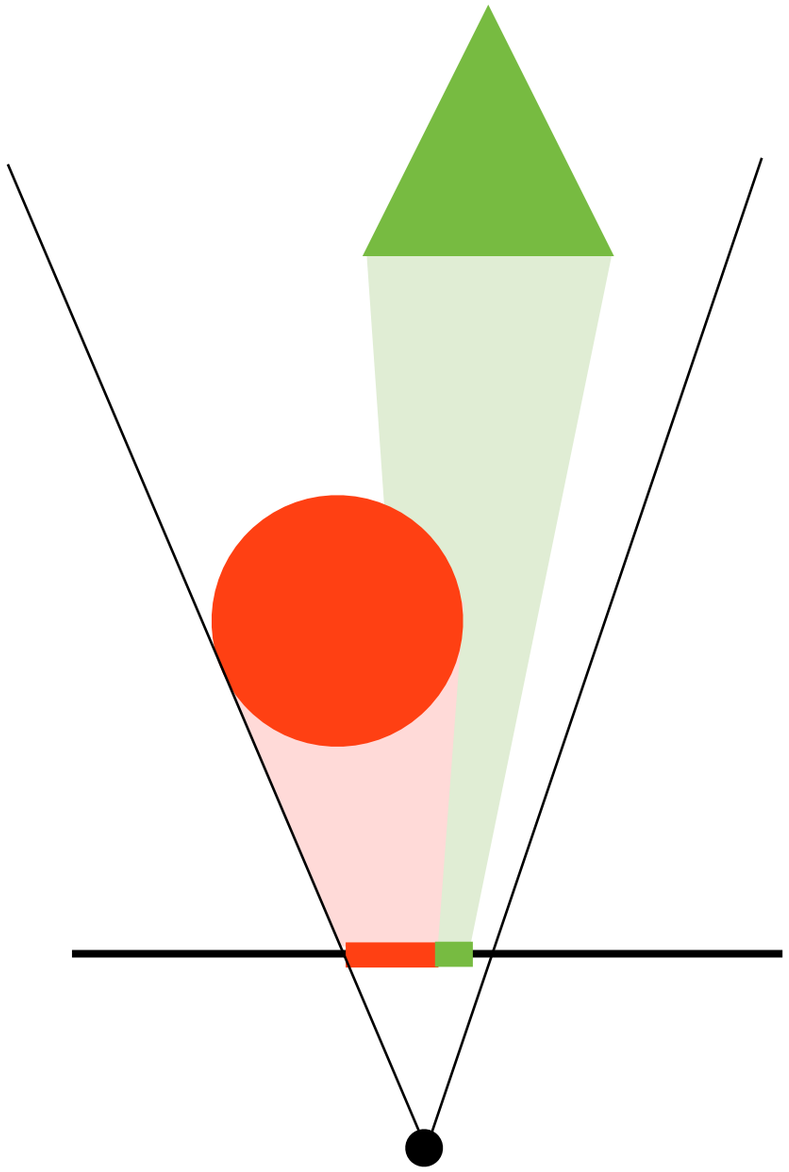
- Drop one coordinate
  - Project onto xy plane: (x,y,z)-> (x,y)
  - Parallel lines remain parallel
  - In homogeneous coordinates:

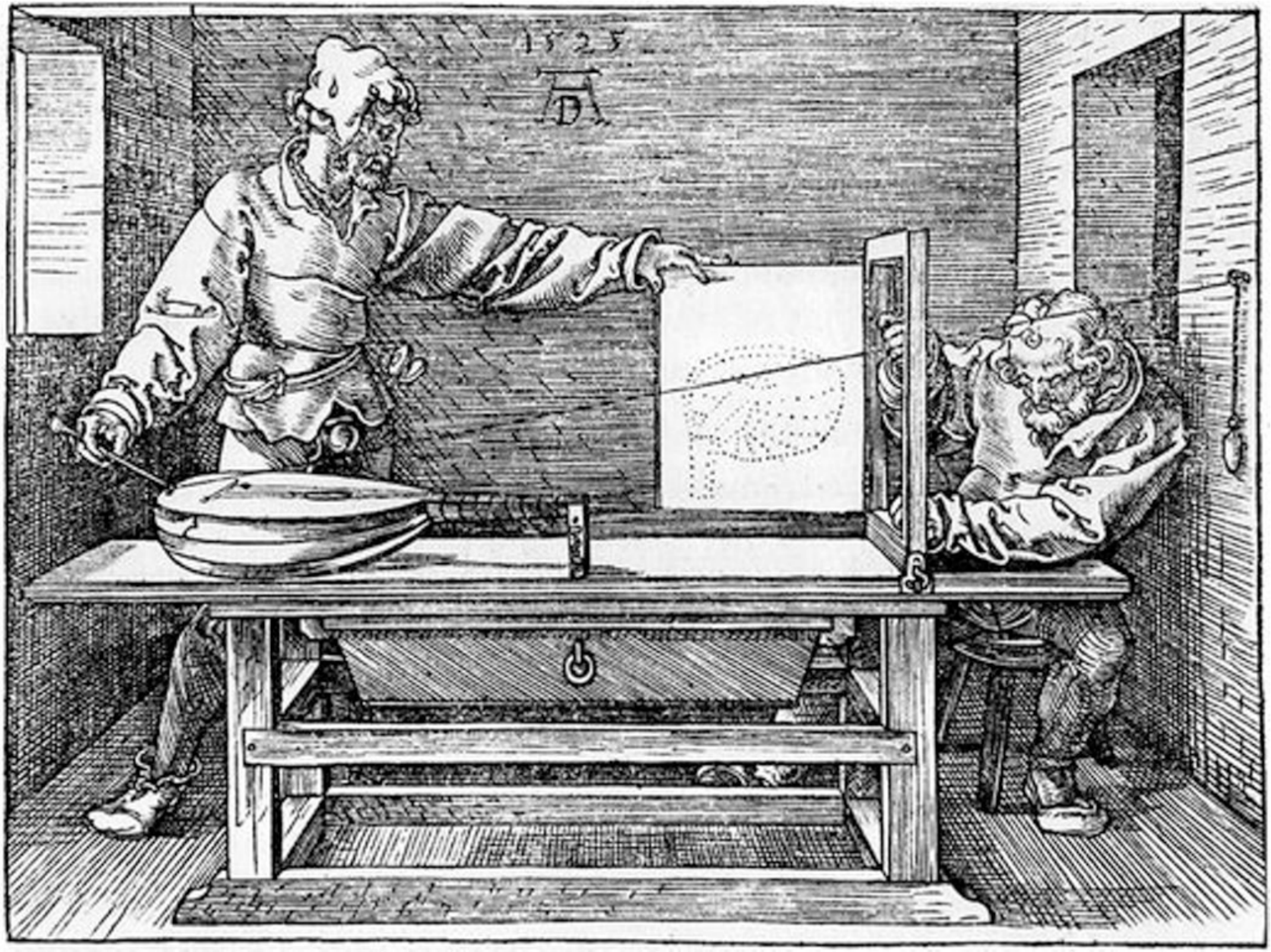


$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Orthographic vs Perspective

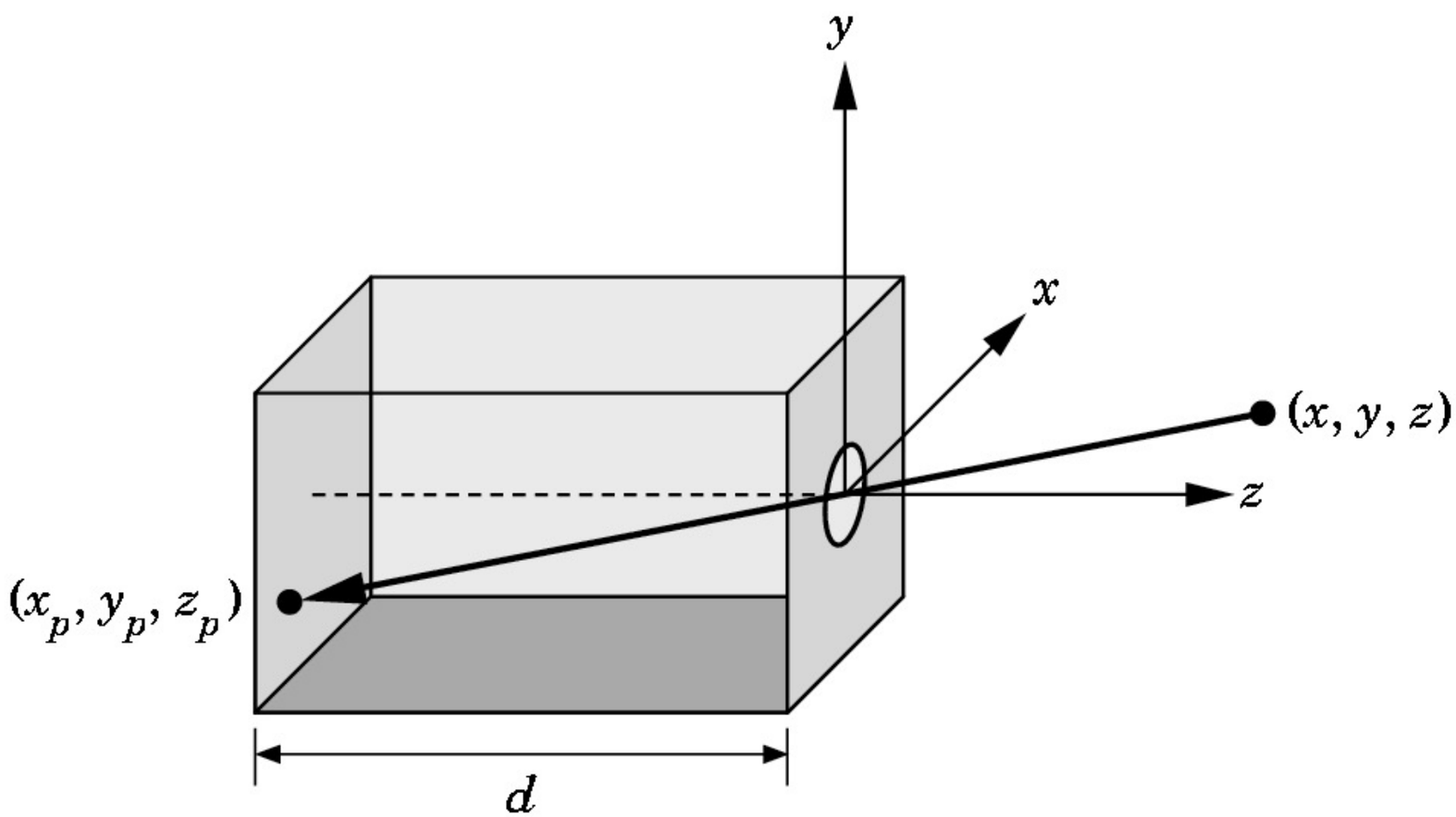


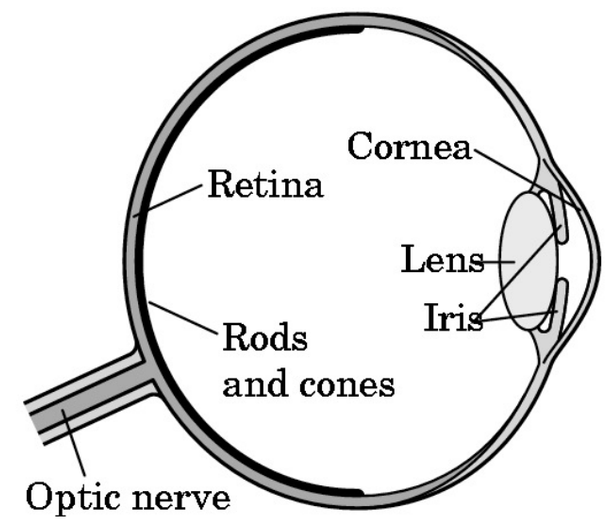




Albrecht Dürer's 1525 woodcut 'Man drawing a Lute'

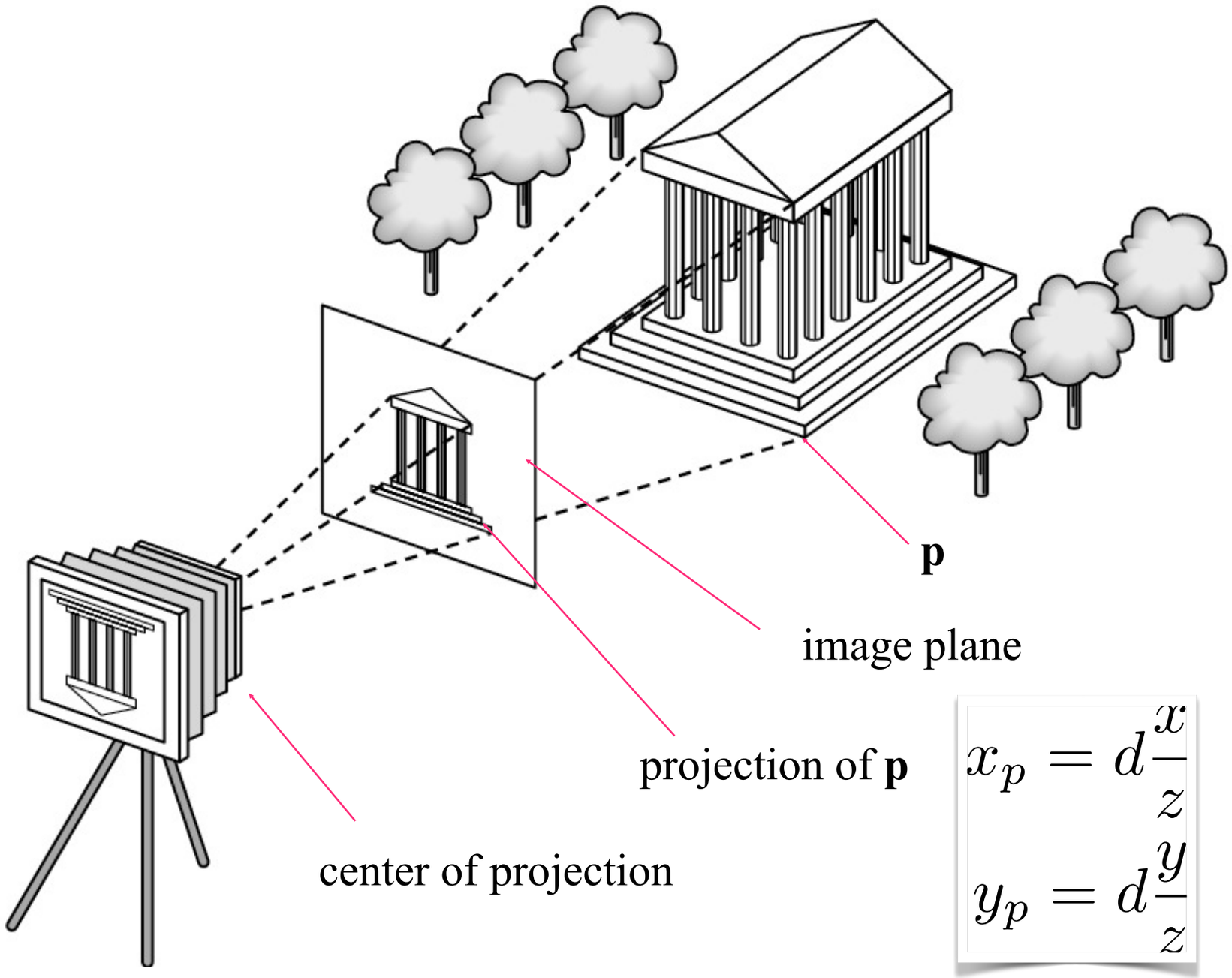
Pinhole Camera





- Projection of a 3D point (x,y,z) on image plane: $x_p = -d\frac{x}{z}, \quad y_p = -d\frac{y}{z}$

Synthetic Camera Model



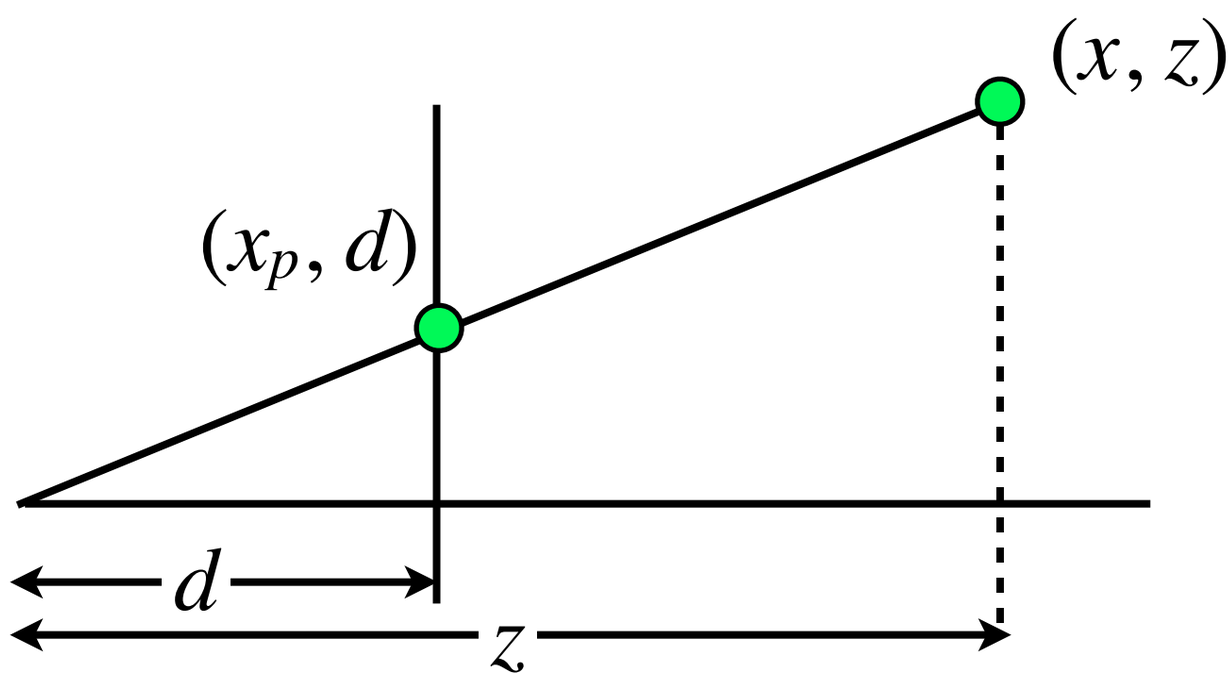
$$x_p = d\frac{x}{z}$$

$$y_p = d\frac{y}{z}$$

Perspective Projection

- More realistic model - objects far away are smaller after projection

$$(x, y, z) \rightarrow (d\frac{x}{z}, d\frac{y}{z})$$



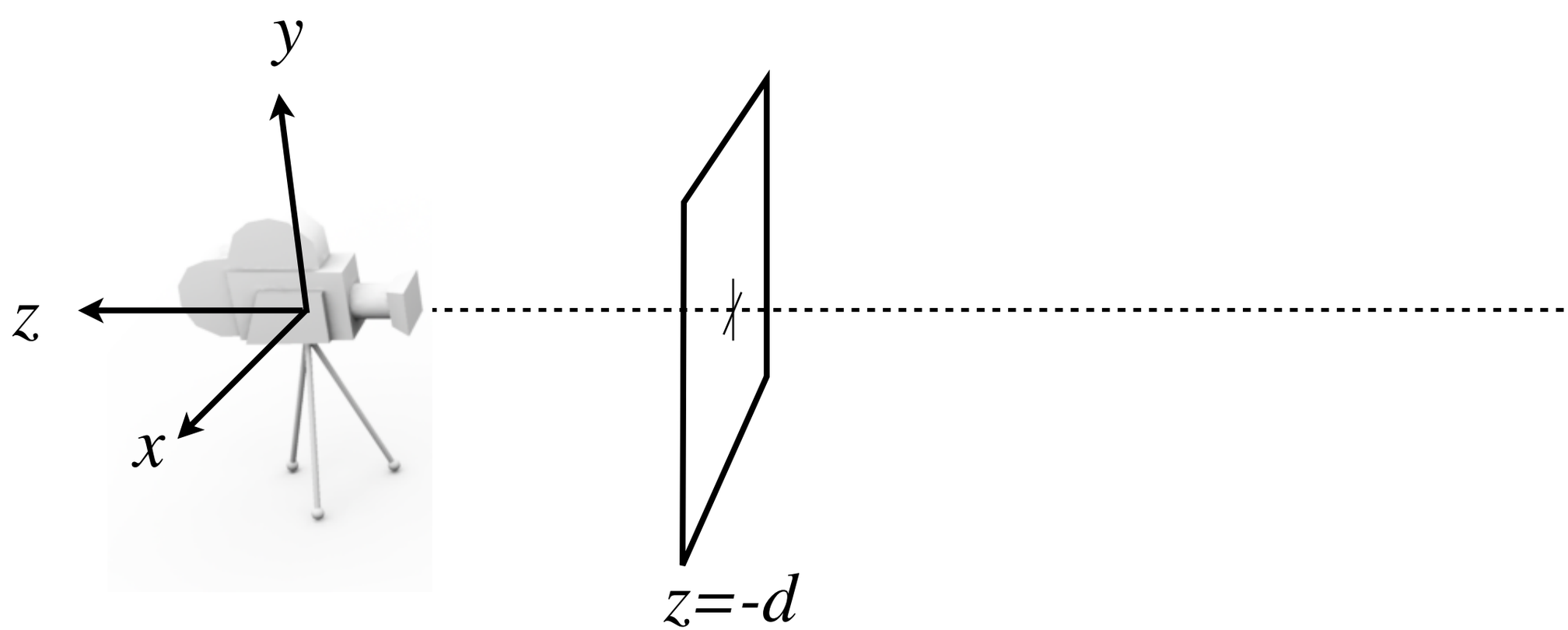
Equal triangles

$$\frac{x_p}{d} = \frac{x}{z}$$

$$x_p = d\frac{x}{z}$$

OpenGL convention

- In OpenGL: right-hand coordinate system, looking down $-z$.
  - The image plane is placed at $z = -d$
  - Visible geometry has negative z-values



Homogeneous Coordinates

- Change our homogeneous coordinate to be scaled

$$(wx, wy, wz, w) = (x, y, z, 1)$$

- Normalize by dividing with w
- Vector: $\mathbf{v} = (x, y, z, 0)$
  - "Point at infinity", pure direction
- Exploit this representation to express projection

Perspective Projection in Matrix Form

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \boxed{-\frac{1}{d}} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{bmatrix} \rightarrow \begin{bmatrix} \boxed{\begin{matrix} -\frac{dx}{z} \\ -\frac{dy}{z} \\ -d \end{matrix}} \\ 1 \end{bmatrix}$$

projected point on image plane $z = -d$

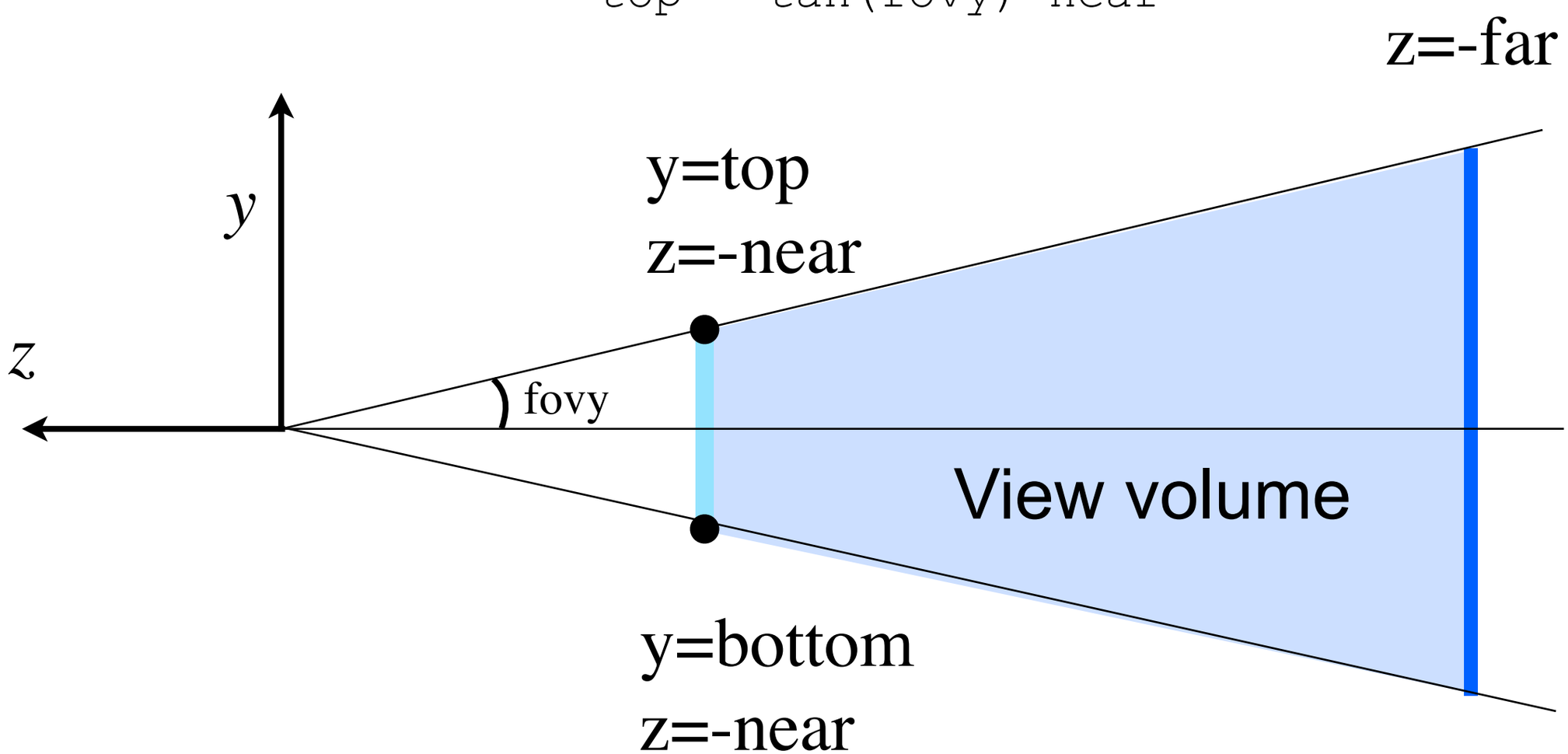
Divide by: $-\frac{z}{d}$

Common standard: Let $d = 1$

Camera in OpenGL

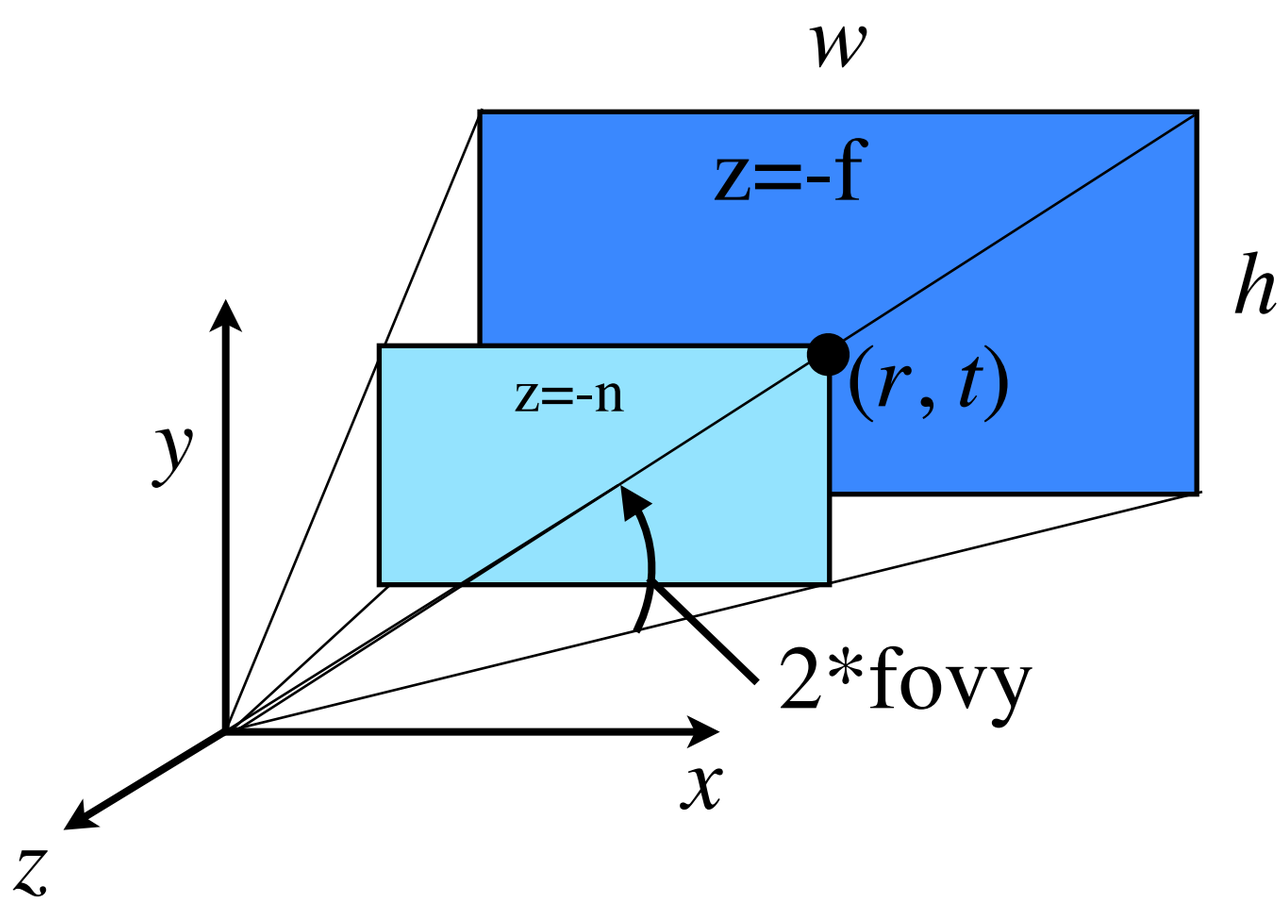
- Perspective camera setup
  - Field of View (fov)

```
top = tan(fovy)*near
```



OpenGL Projection Matrix

```
mat4 proj = Perspective(fovy,aspect,n,f);
```

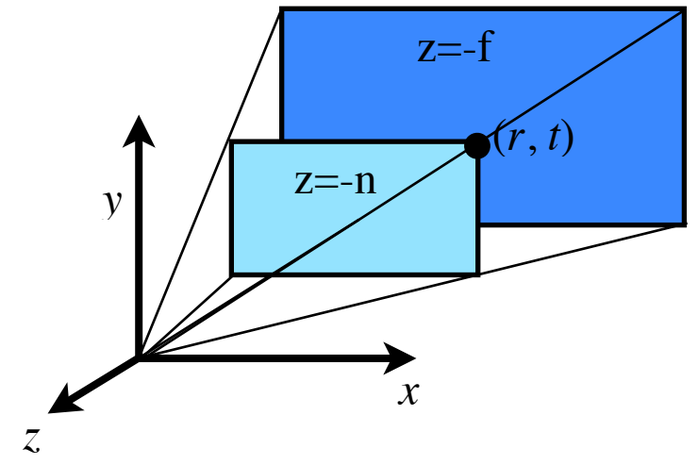



```
aspect = w/h
t = tan(fovy)*n
r = t*aspect
```

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

```
mat4 proj = glm::perspective(fovy,aspect,n,f);
```

Examples



Point at upper right corner at near plane (z=-n)

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} r \\ t \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} n \\ n \\ -n \\ n \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

divide by w

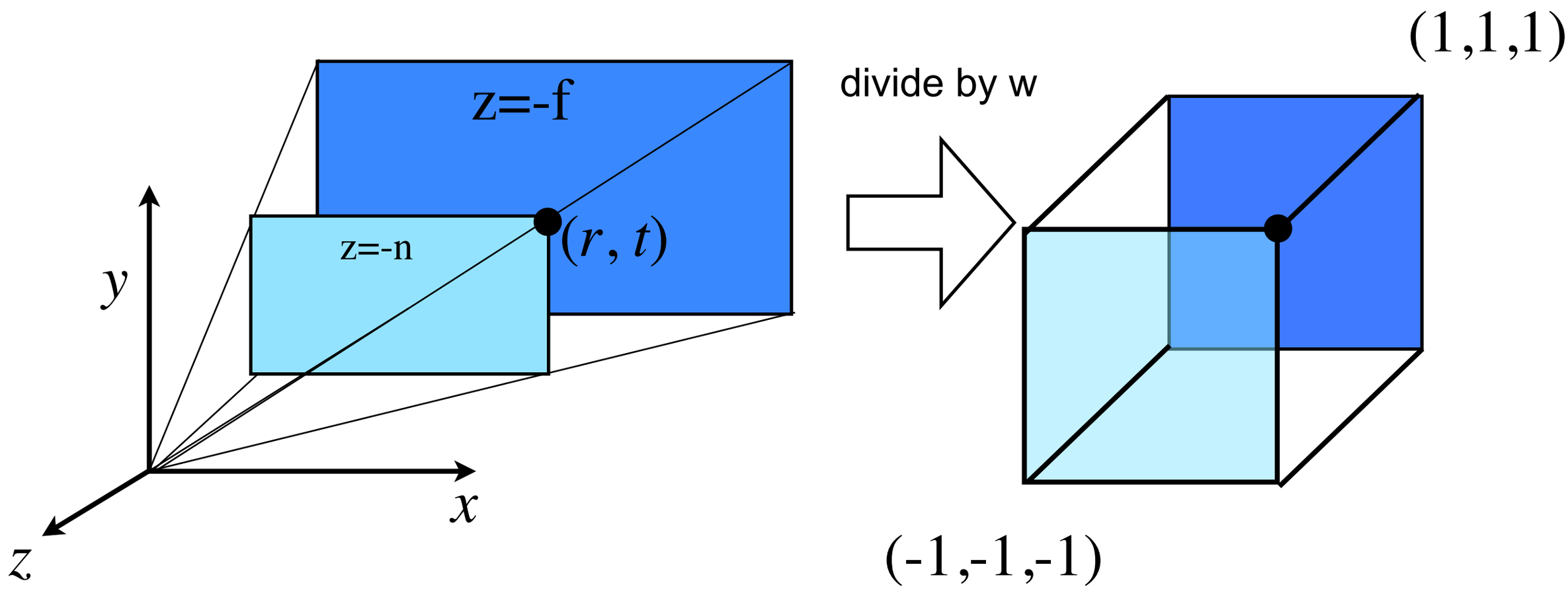
Point along view direction (-z) at far plane (z=-f)

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ f \\ f \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

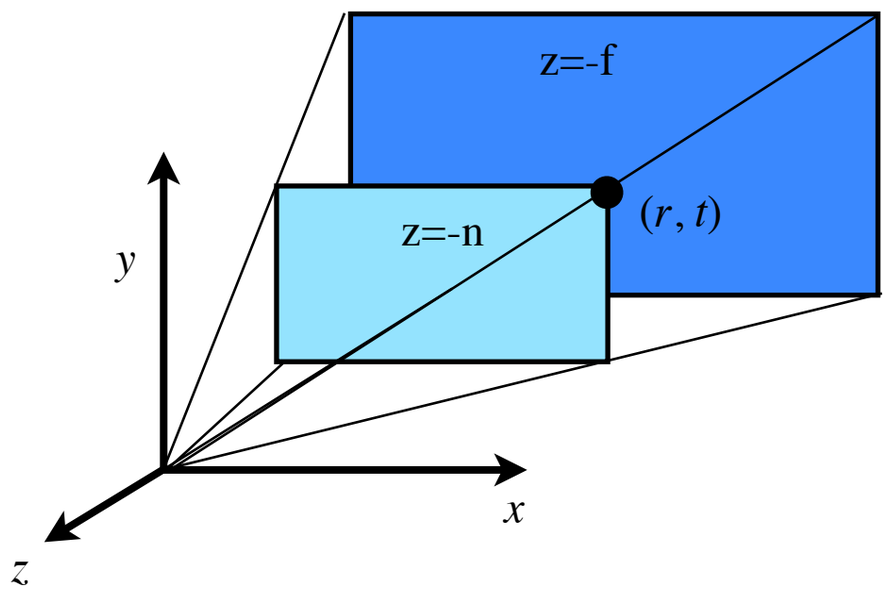
divide by w

OpenGL Projection Matrix

- View frustum volume maps to a cube



New Coordinate Spaces



divide by w

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \\ 1 \end{bmatrix}$$

Projection Matrix | Camera space | Clip space | NDC

Normalized Device Coords

Classification of Transforms

Translation

Rotation

Rigid Body
preserves angles
and distances

Uniform Scaling

Similarity
preserves angles

Non-Uniform Scaling

Shear

Reflection

Affine
preserves
parallel lines

Perspective

Projective preserves lines

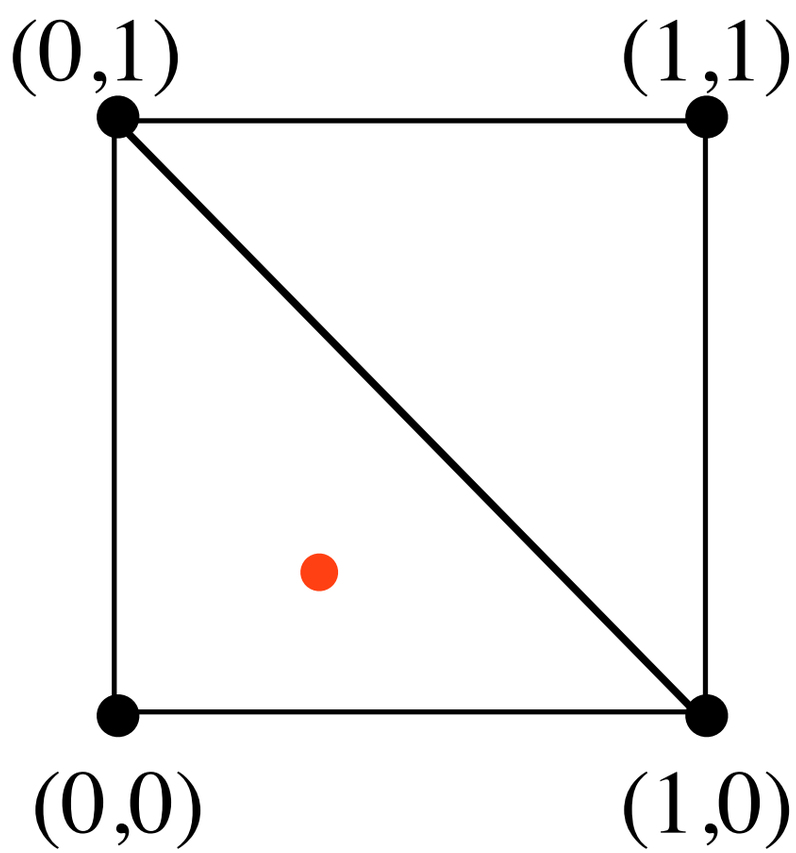
Procedural Techniques

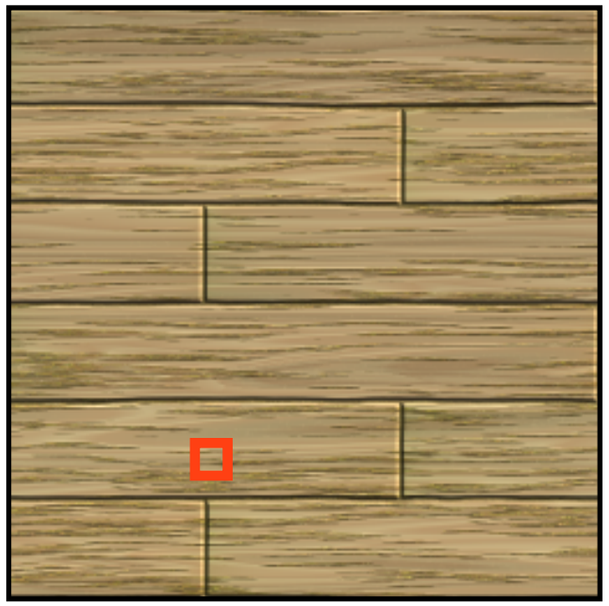
Procedural Techniques

- Instead of describing detail with textures, use mathematical functions
  - Clouds, smoke, fire
- Human visual system very good at detecting patterns/repetitions
- Add randomness for more realism!

Texture Coordinates

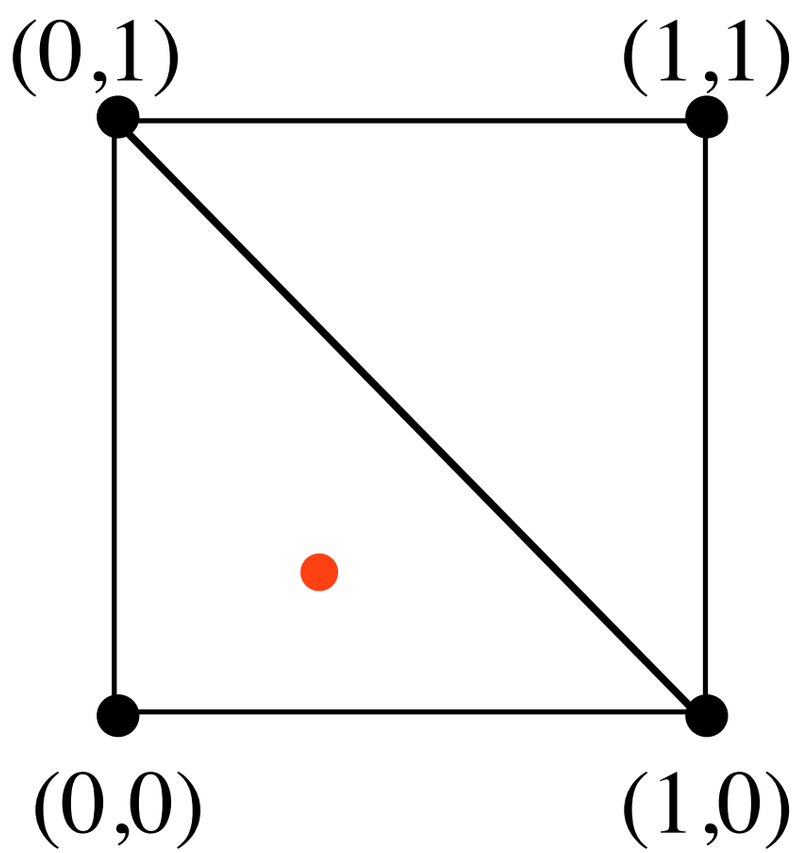
- Specify tex coordinate at each vertex
  - Hardware interpolates texCoord for each pixel
- Use coordinate to lookup in texture

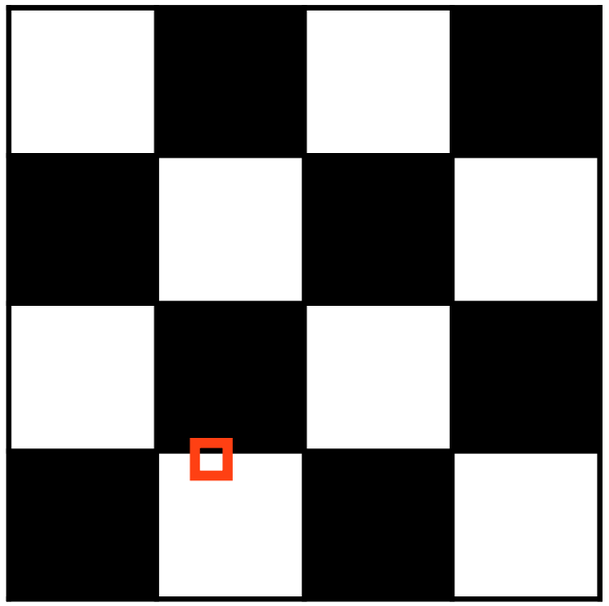




Procedural Texturing

- Use coordinate to define a function that computes the color
  - No need to store (large) texture in memory!

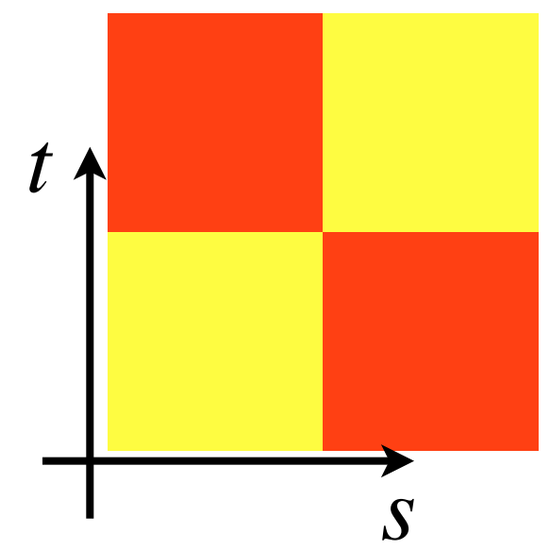




Example: Checkerboard

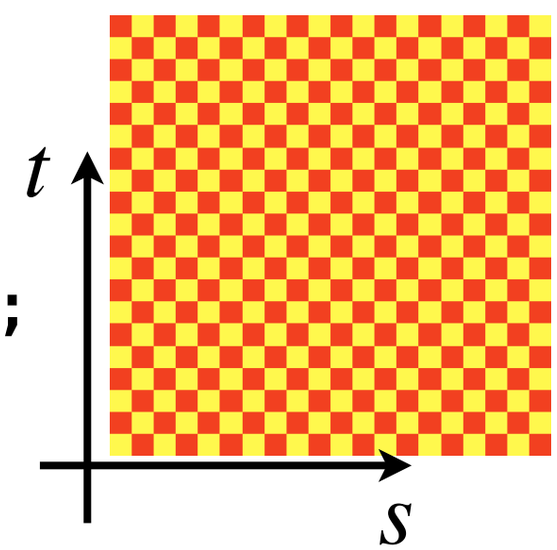
```
float s = texCoord.x;
float t = texCoord.y;
vec3 red    = vec3(1,0,0);
vec3 yellow = vec3(1,1,0);
vec3 c = (s < 0.5) ^^ (t < 0.5) ? red : yellow;
fColor = vec4(c,1.0);
```

A	B	A^^B (XOR)
0	0	0
1	0	1
0	1	1
1	1	0



Generalize to higher frequencies

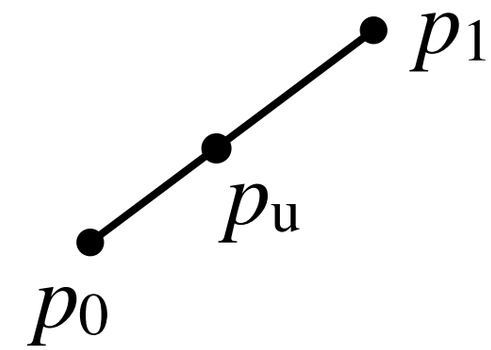
```
float freq = 10.0;
vec3 c =   (mod(freq*s,1) < 0.5)
        ^^ (mod(freq*t,1) < 0.5) ? red : yellow;
```



Bi-linear interpolation

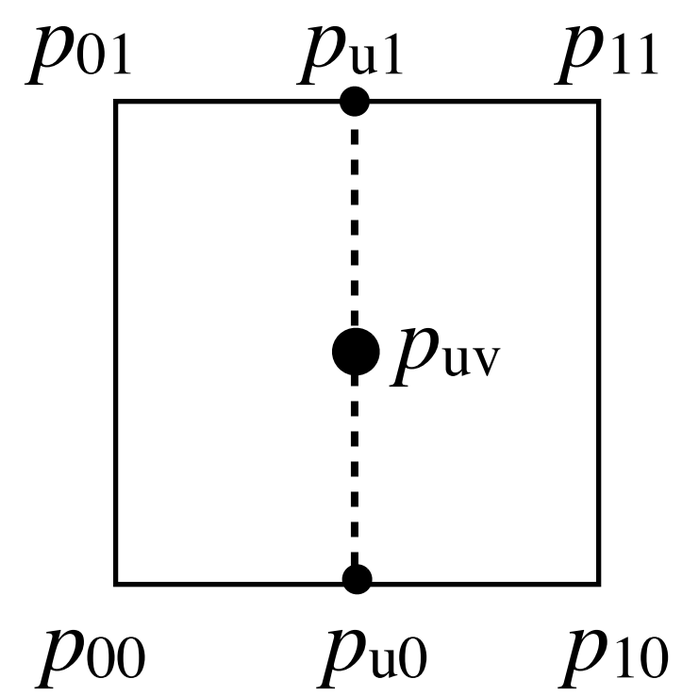
- Remember linear interpolation

$$p_u = (1-u)p_0 + up_1$$



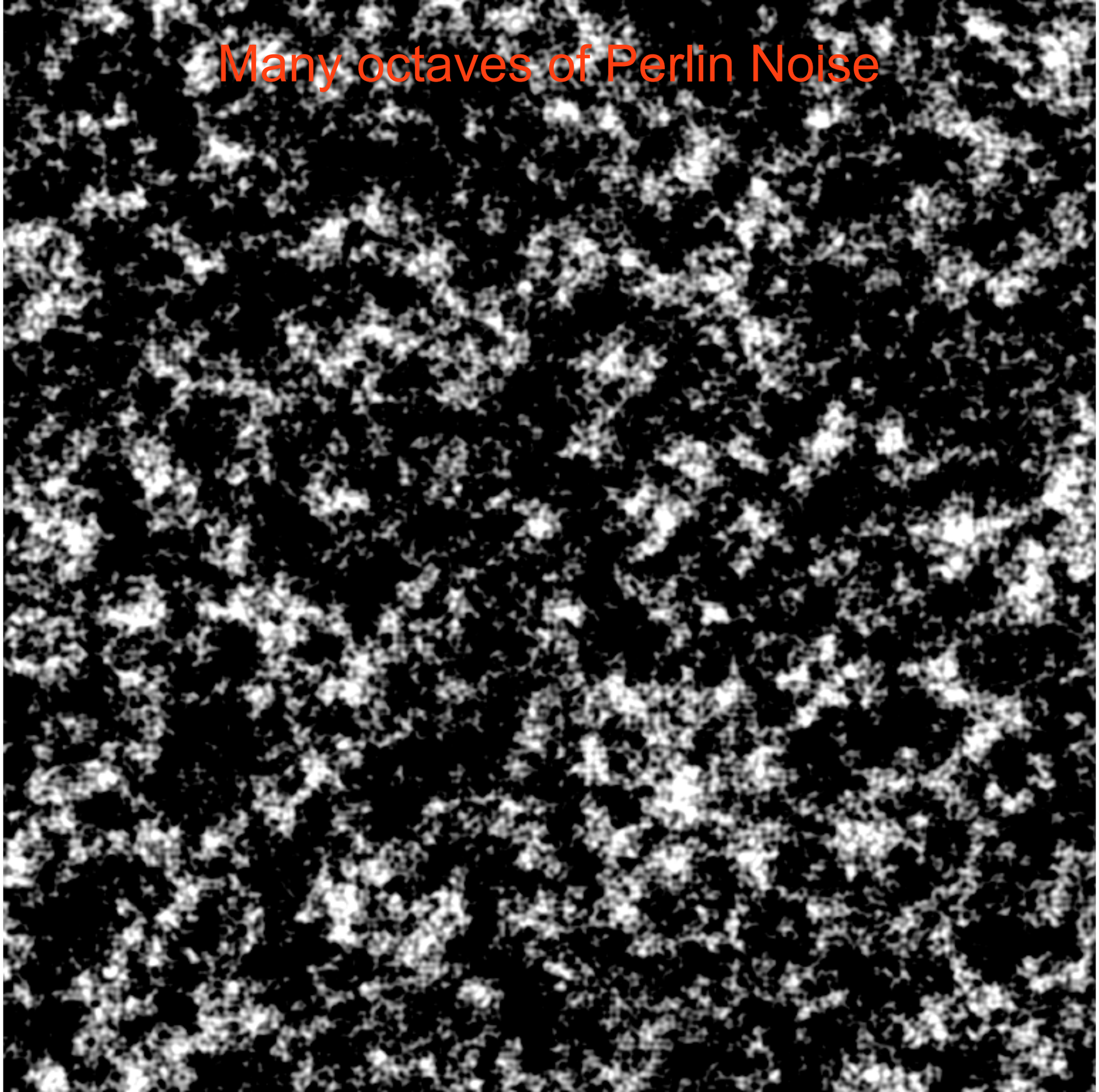
- Bilinear interpolation:

$$p_{u0} = (1-u)p_{00} + up_{10}$$
$$p_{u1} = (1-u)p_{01} + up_{11}$$
$$p_{uv} = (1-v)p_{u0} + vp_{u1}$$



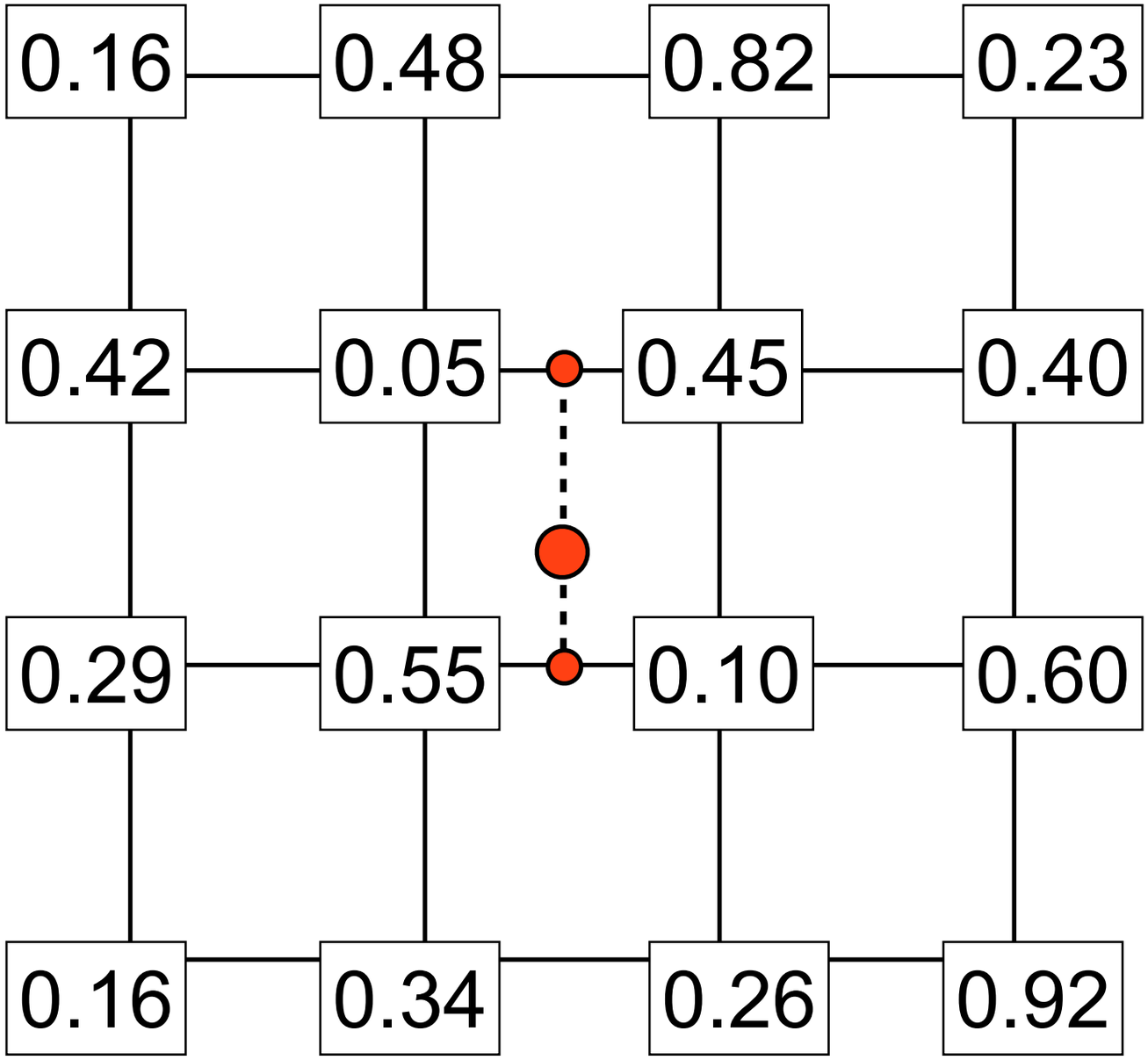
Noise

- A controlled random primitive
  - Bandlimited
  - Repeatable (same input value gives same output)
- Smooth interpolation between random values
  - Assign a value to each integer
  - Interpolate between integer values


Many octaves of Perlin Noise

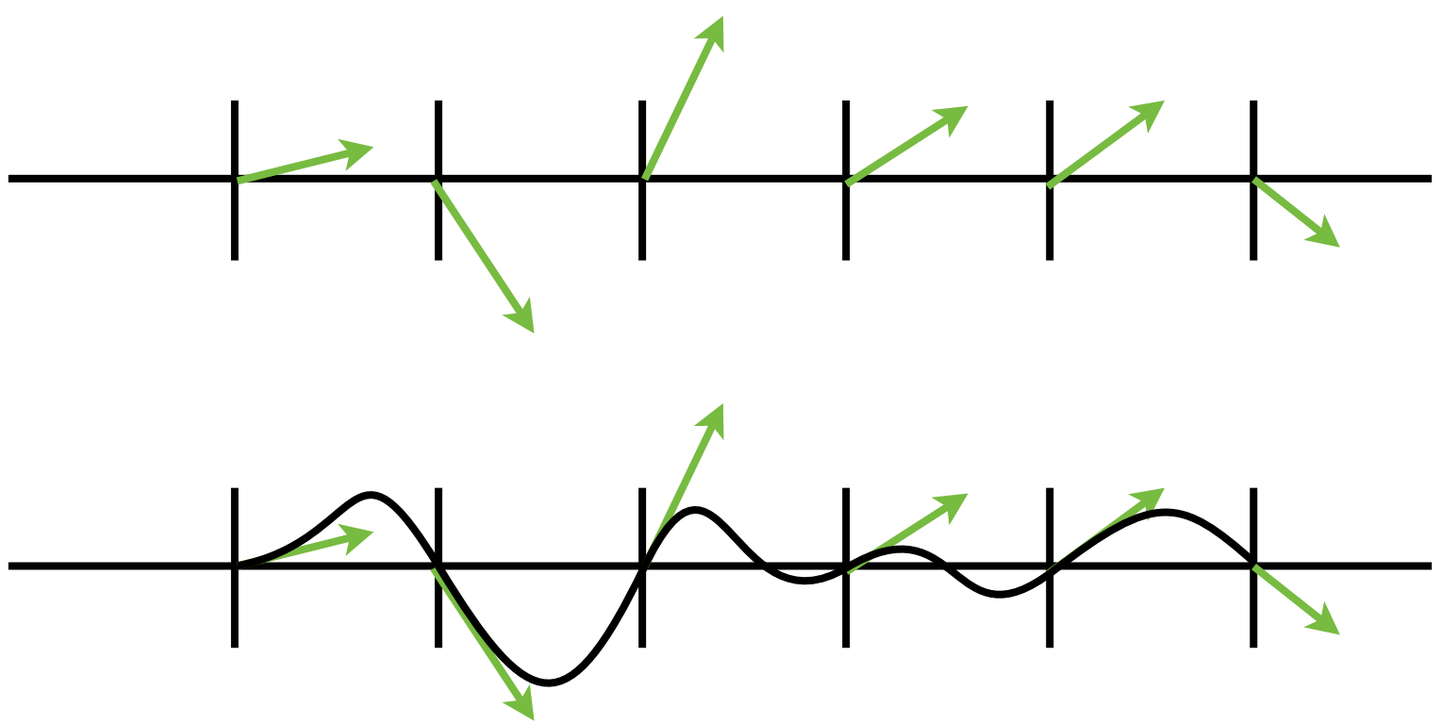
Example: 2D Value Noise

- Assign random value to each vertex in integer grid
- Bi-linear interpolation between values
- Smooths out noise

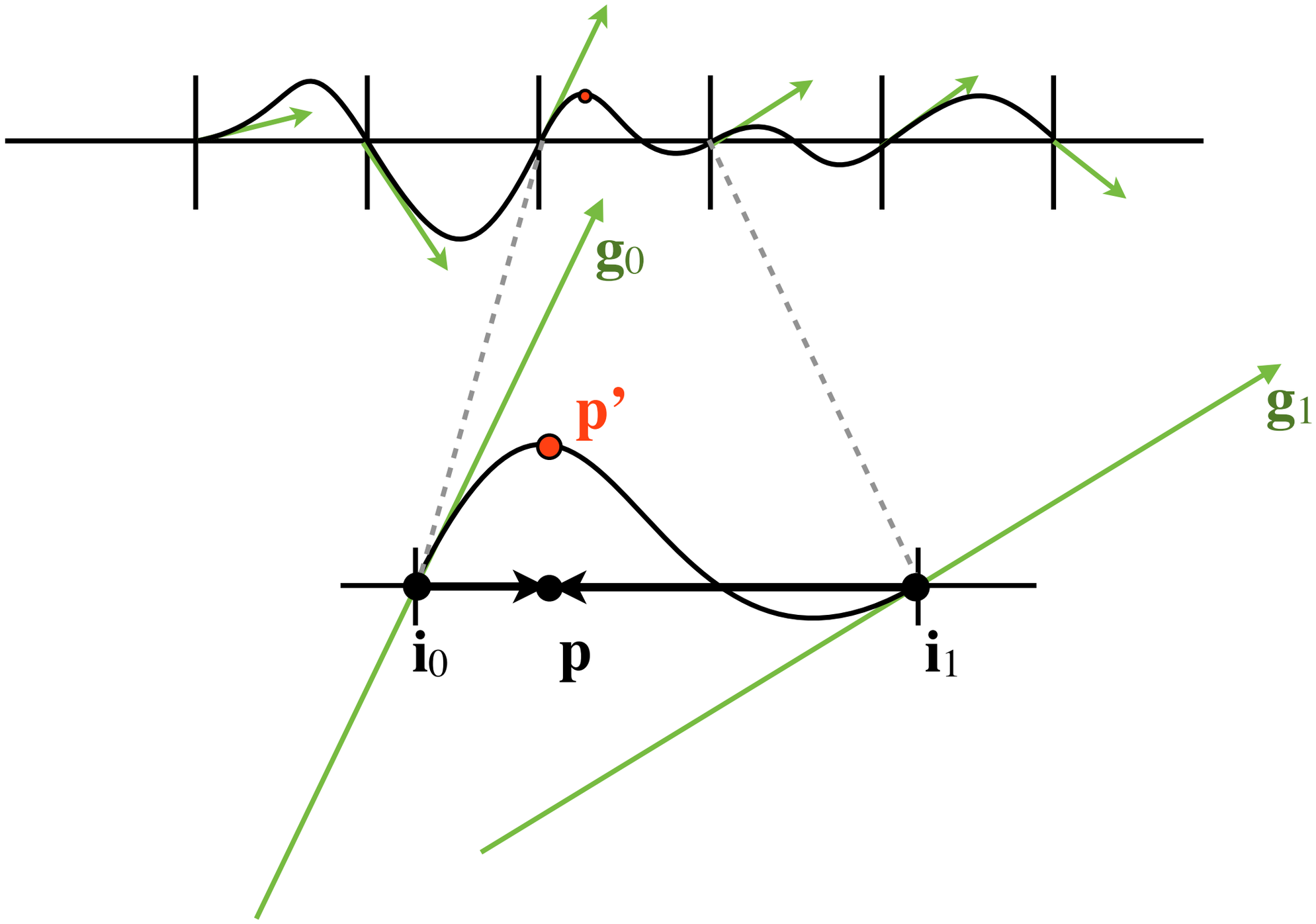


Perlin Noise

- Assign random gradient to each grid point
- Smooth interpolation between gradients



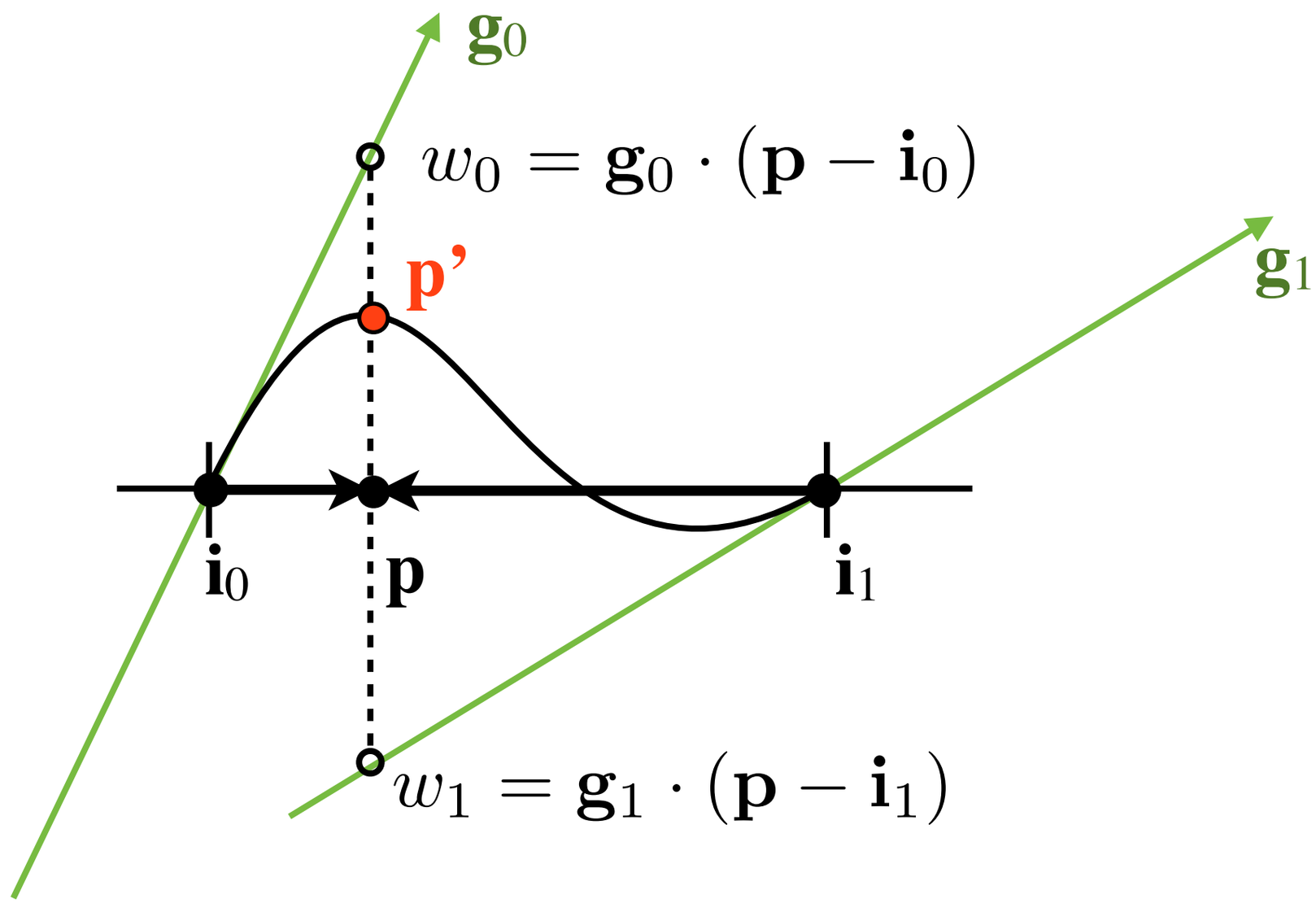
Perlin Noise



What is the value at **p'** ?

Perlin Noise

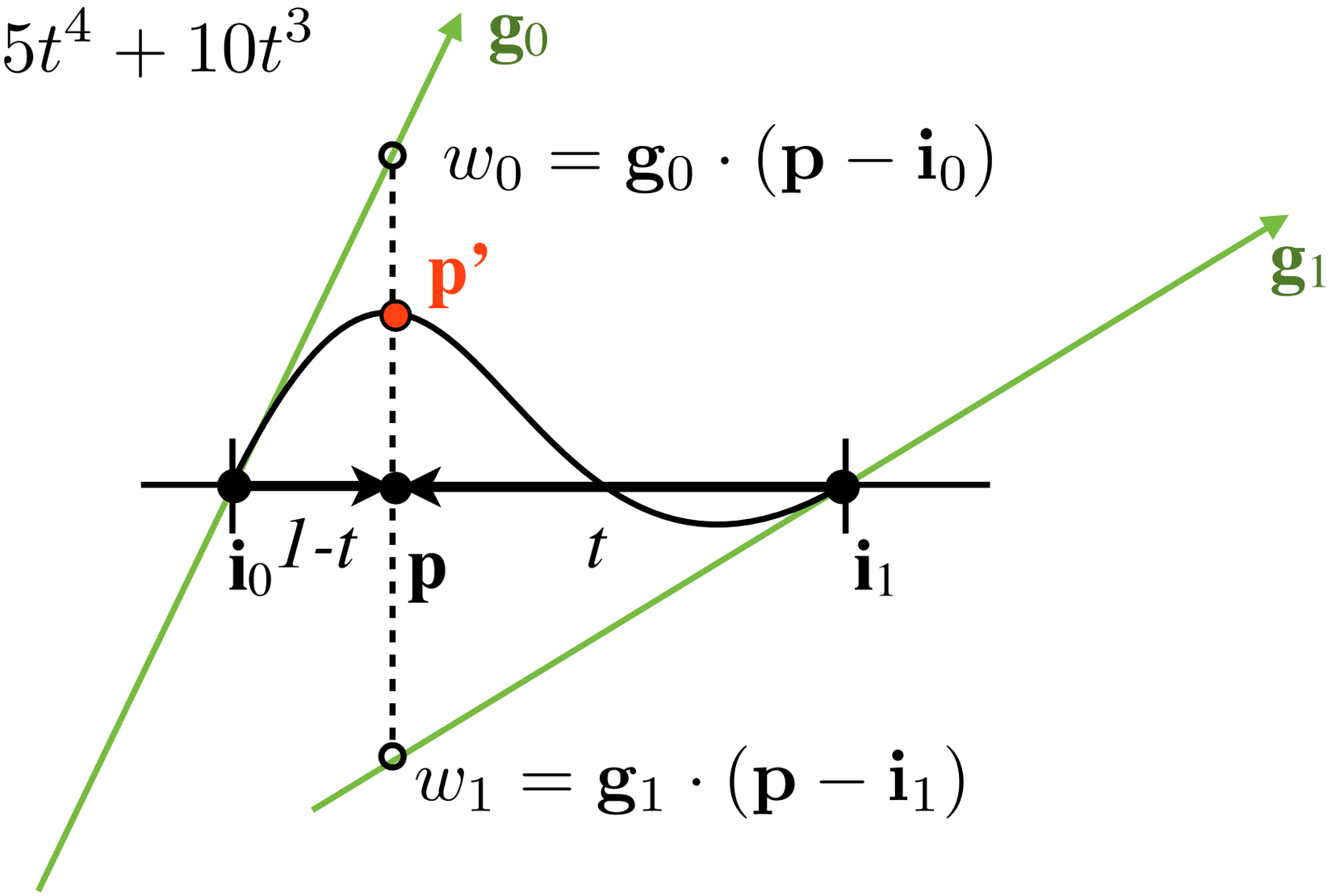
Find weights (scalar product between gradient $\mathbf{g}$ and $\mathbf{p}$-$\mathbf{i}$)



Perlin Noise

Smooth interpolation between weights

$$f(t) = 6t^5 - 15t^4 + 10t^3$$



$$\boxed{\mathbf{p}' = f(|\mathbf{p} - \mathbf{i}_0|)w_0 + f(|\mathbf{p} - \mathbf{i}_1|)w_1}$$

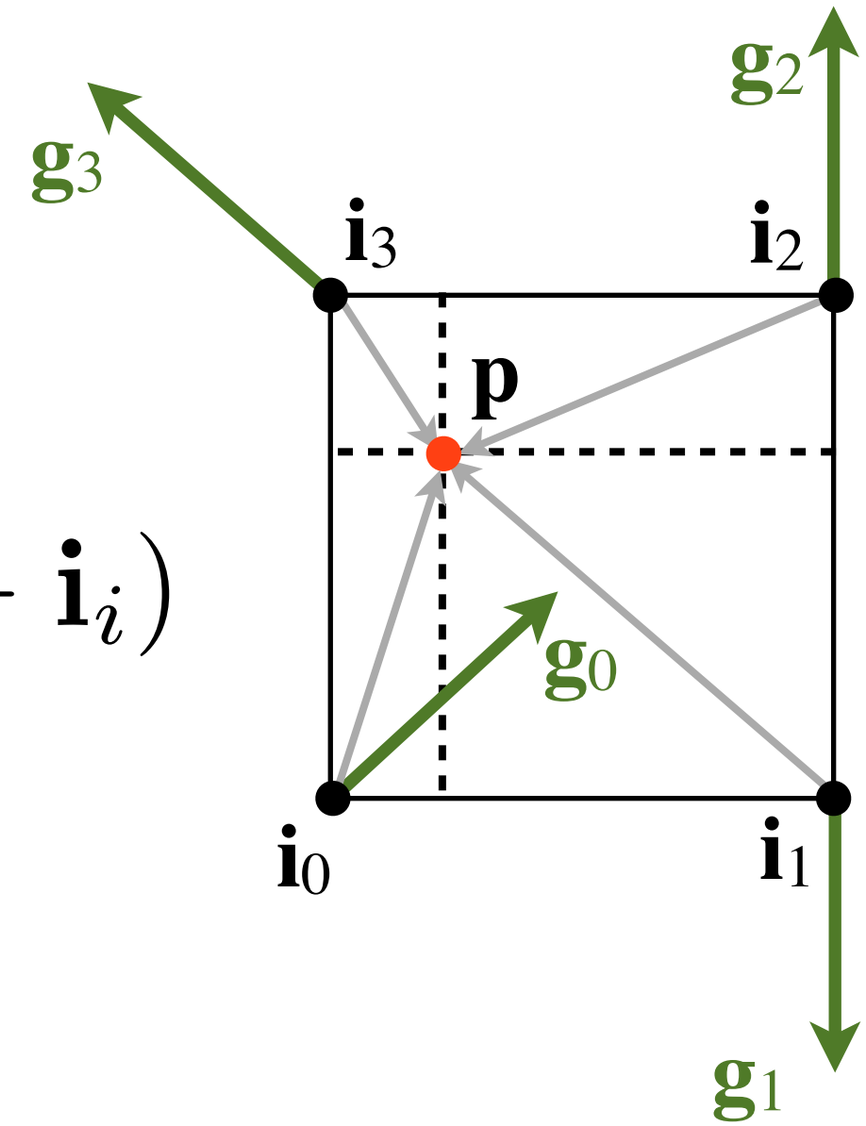
Perlin Noise

- Assign gradient vector, $\mathbf{g}$, to each lattice point $\mathbf{i}$
- Compute gradients weights: $w_i = \mathbf{g}_i \cdot (\mathbf{p} - \mathbf{i}_i)$
- Blend between weights

$$\sum_i f(|\mathbf{p} - \mathbf{i}_i|) w_i$$

  - Use smooth interpolation between the weights

$$f(t) = 6t^5 - 15t^4 + 10t^3$$



Perlin Noise

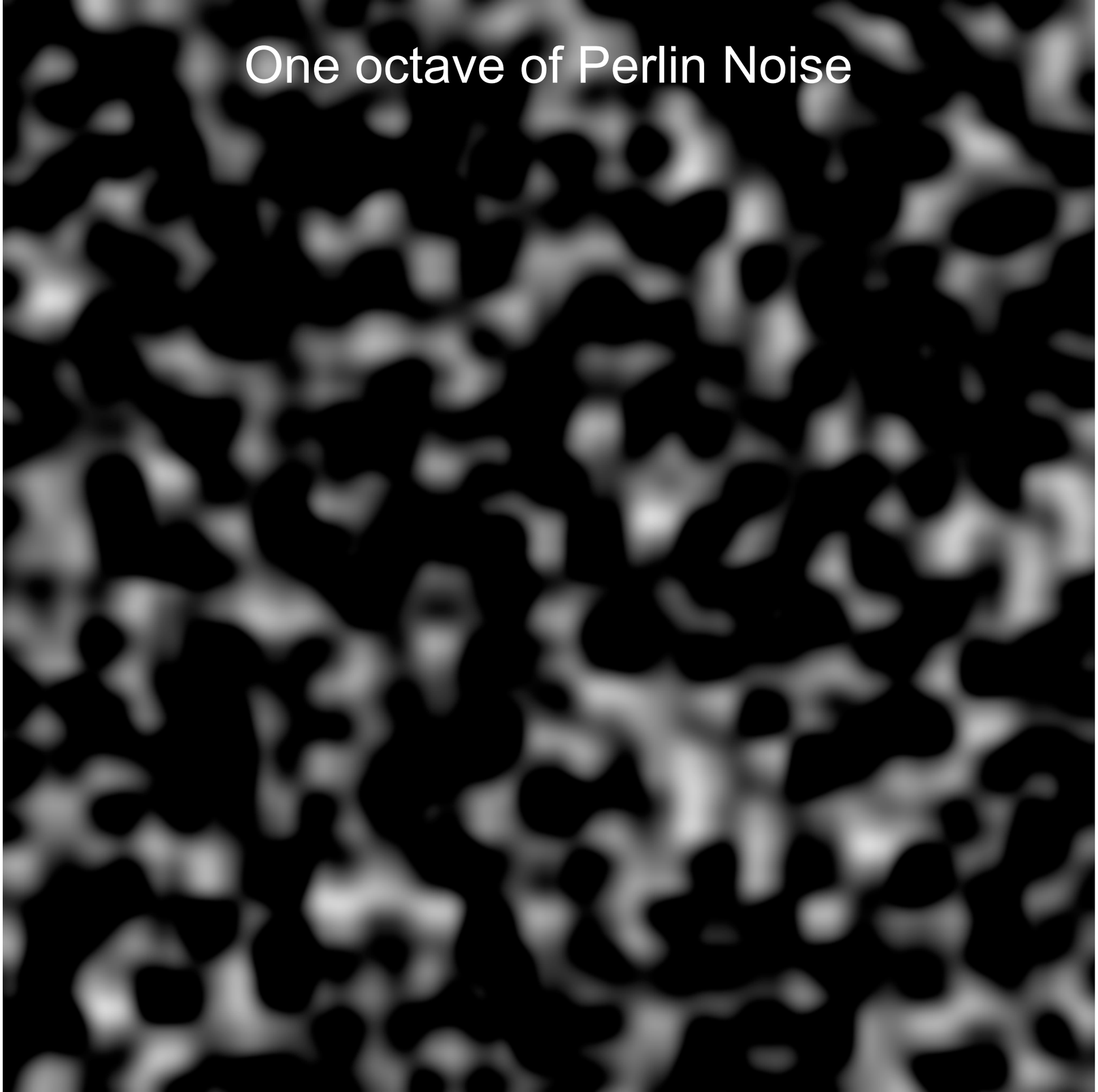
- Perlin noise can be efficiently implemented
  - No need to store large grid of noise values
- Extends to 3D & 4D

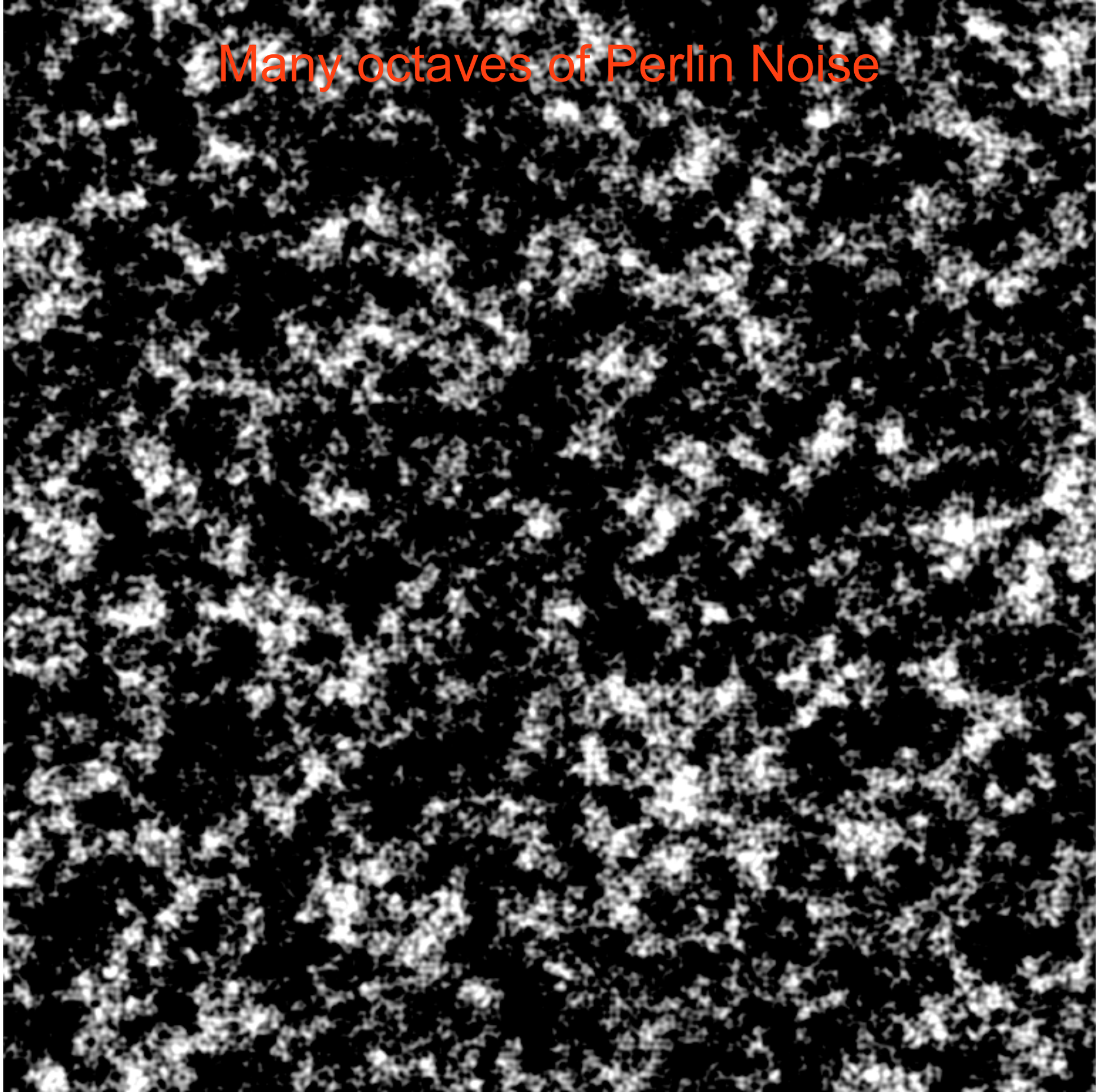
Read more at Read more at https://github.com/stegu/perlin-noise/blob/master/simplexnoise.pdf
Explains Simplex Noise, and also Classic Perlin Noise as presented here.

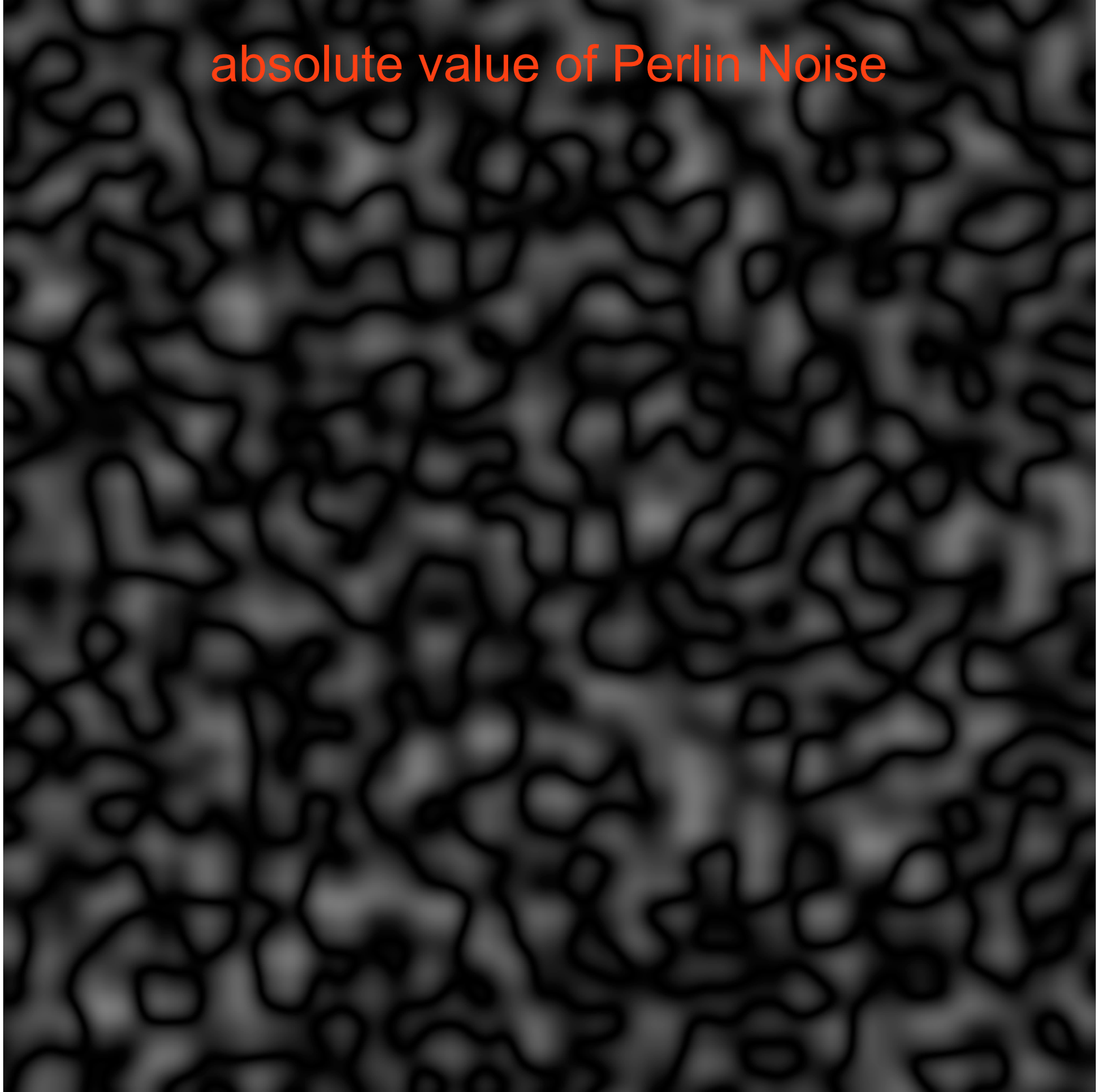
Turbulence

- Sum several octaves of Perlin noise

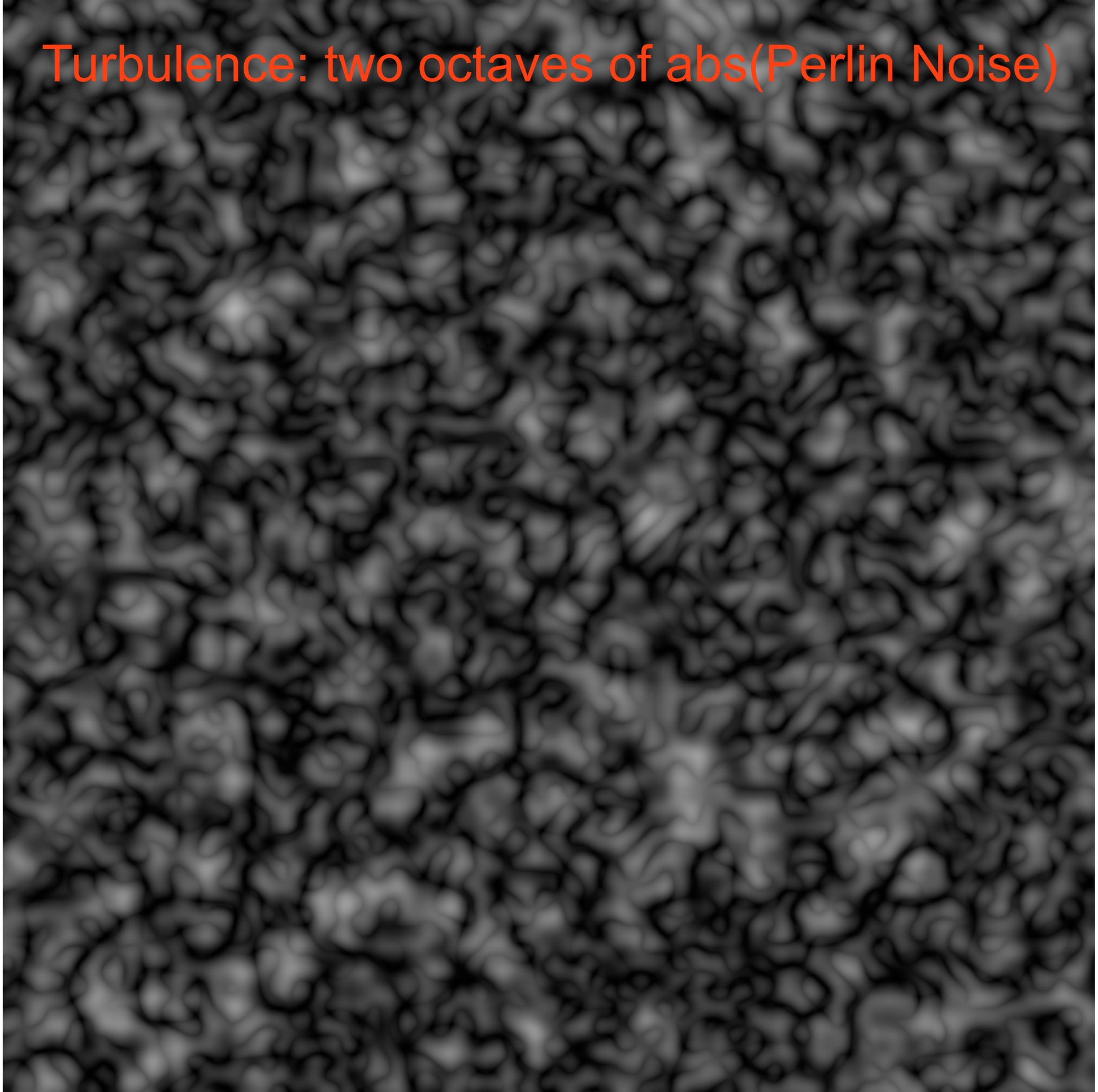
```
float turbulence(vec3 pos, const int octaves)
{
    float sum = 0;
    float omega = 0.6;
    float lambda = 1.0;
    float o = 1.0;
    for (int i=0; i<octaves; ++i)
    {
        sum += abs(o * noise(pos*lambda));
        lambda *= 1.99f;
        o *= omega;
    }
    return sum;
}
```

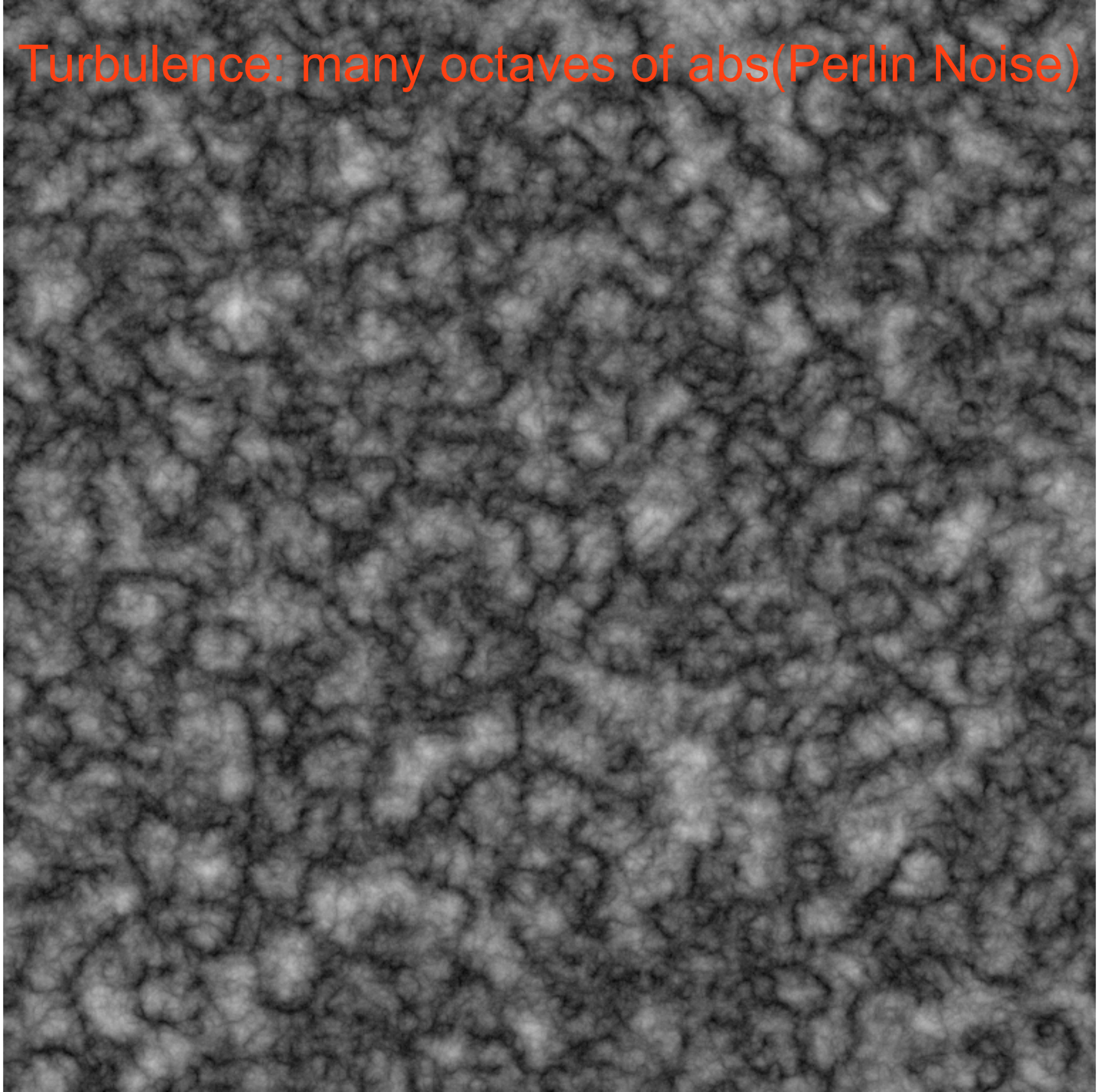

One octave of Perlin Noise


Many octaves of Perlin Noise


absolute value of Perlin Noise

Turbulence: two octaves of abs(Perlin Noise)




Turbulence: many octaves of abs(Perlin Noise)

Next

- Wednesday Lab4 seminar - Water shader
- Procedural video about making a landscape
  - Painting a Landscape with Maths