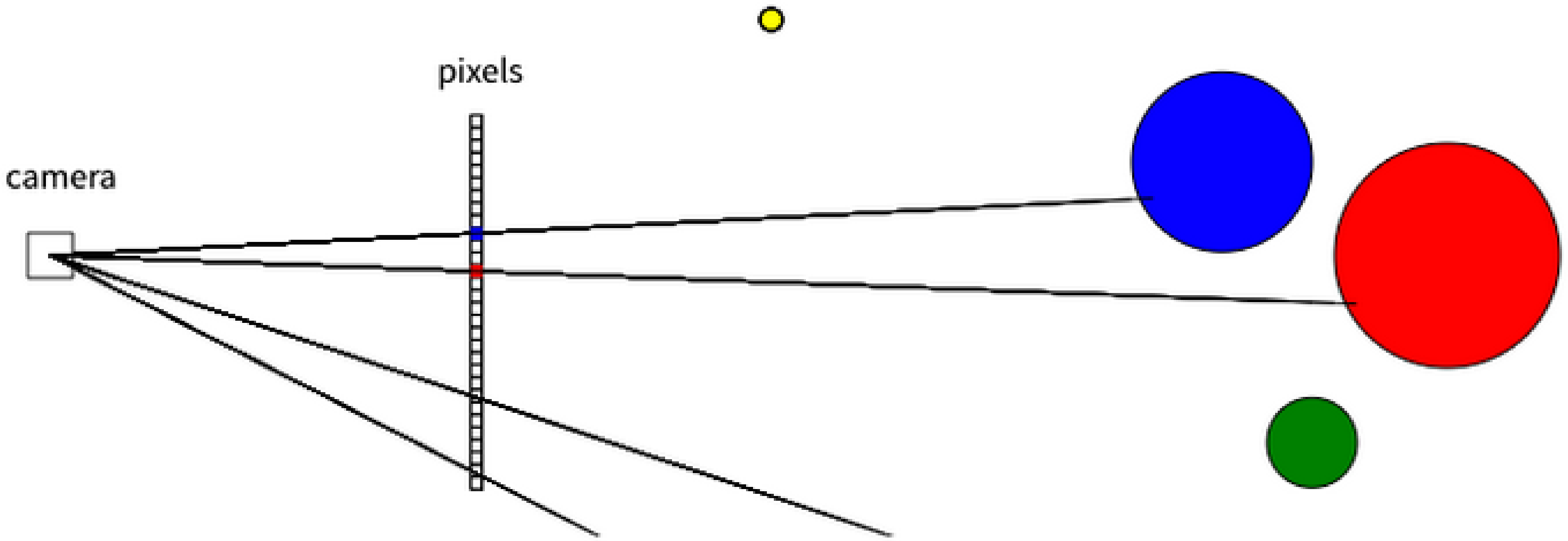
Ray tracing is a pixel-first approach to rendering.



1.) set up view and image plane

2.) for each pixel:

a.) create ray

b.) for each object:

does ray hit object?

keep closest

c.) calculate color

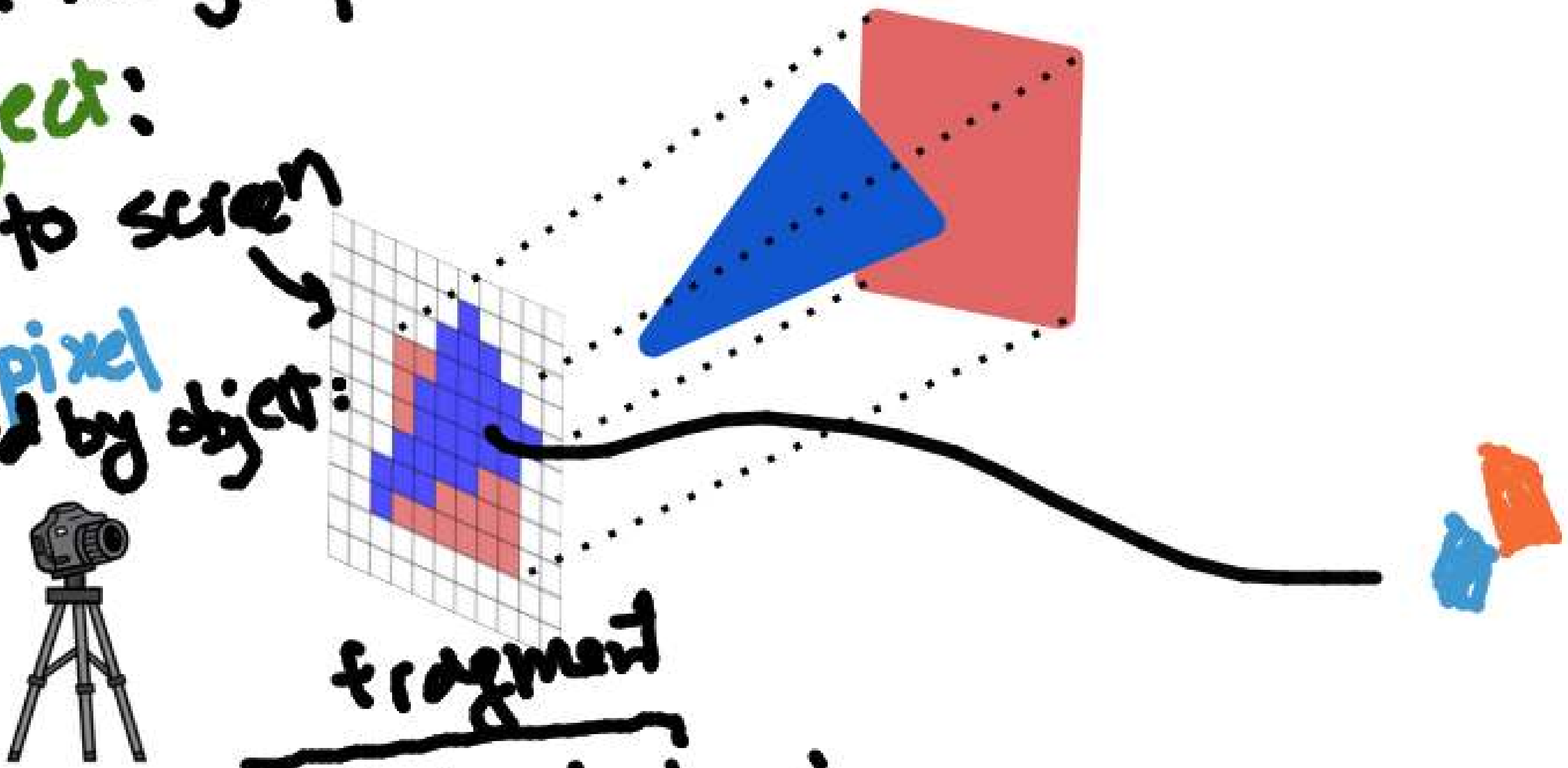
Rasterization is an object-first approach to rendering.

1.) set up view + image plane

2.) for each object:

a) project to screen

b) for each pixel covered by object:



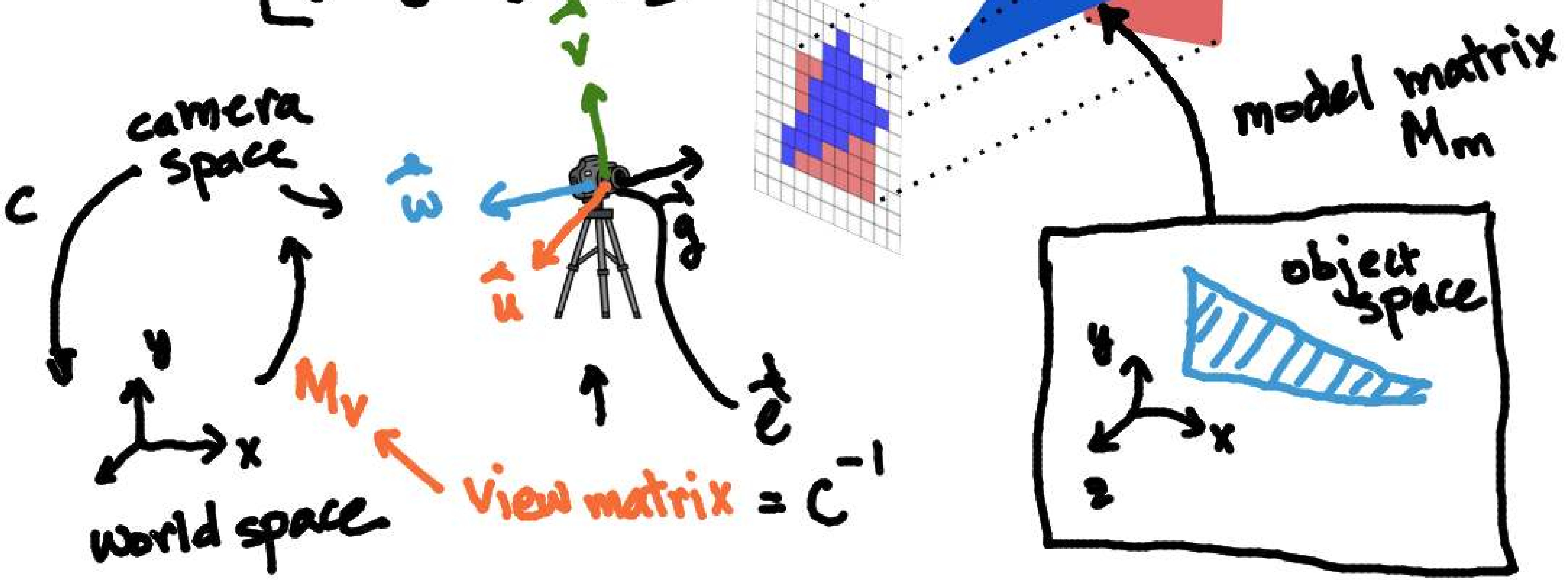
fragment

if depth of the object-pixel pair is less than current depth

c.) calculate color

We need to first transform into the CAMERA frame of reference.

$$C = \begin{bmatrix} u_x & v_x & w_x & e_x \\ u_y & v_y & w_y & e_y \\ u_z & v_z & w_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = TB$$



View Matrix ($\mathbf{M}_v$): transforming from world space *into* camera space.

$$\mathbf{C} = \begin{bmatrix} u_x & v_x & w_x & e_x \\ u_y & v_y & w_y & e_y \\ u_z & v_z & w_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{TB}.$$

unit vectors + orthogonal to each other.

Given the transformation from camera to world space (C = T * B), which matrix will transform from world to camera space? 18

- [ ] inverse(T) * inverse(B)
- [ ] inverse(T) * transpose(B)
- [x] inverse(B) * inverse(T)
- [x] transpose(B) * inverse(T)
- [ ] None of these.

Voting as Anonymous

Send

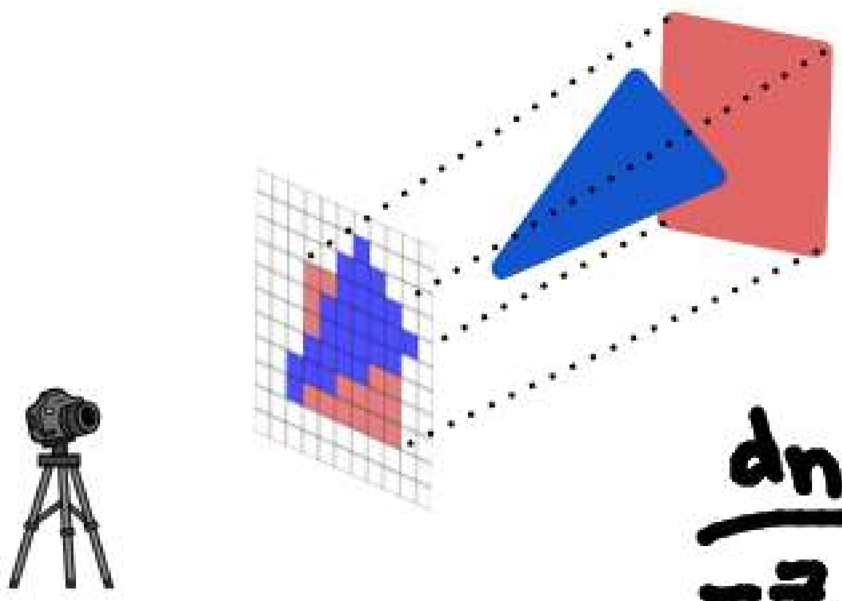
$$M_v = C^{-1} = (TB)^{-1} = \underbrace{B^{-1}}_{\text{orthogonal}} T^{-1}$$

$$B^{-1} = B^T$$

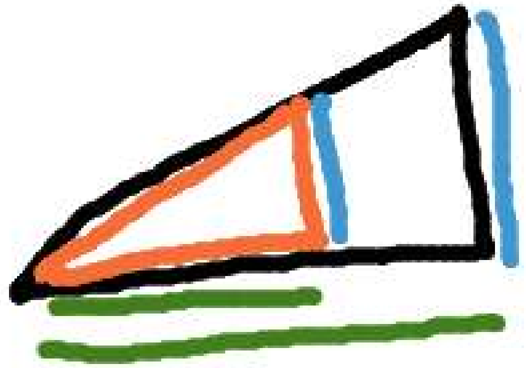
slido.com (event #3097031)

Then we will project vertices onto the image plane.

really want to use a matrix.



similar triangles:

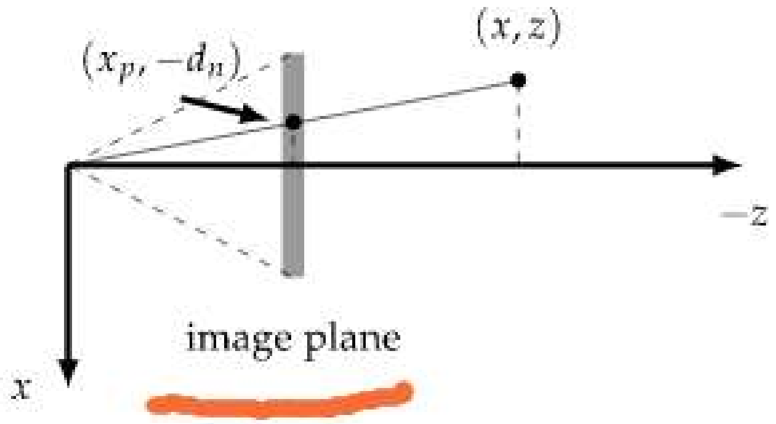


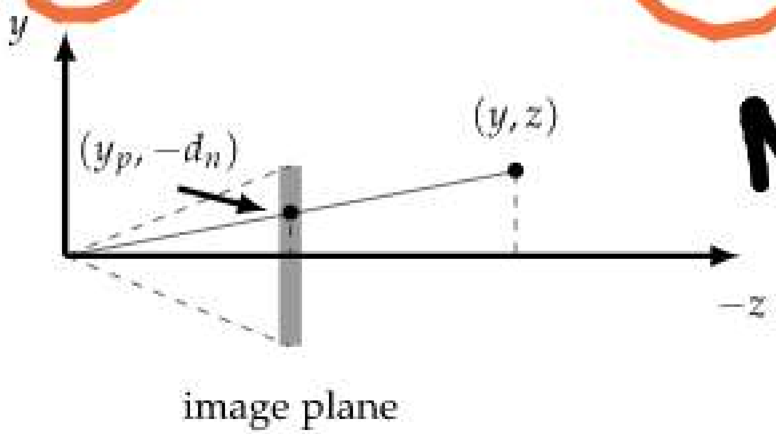
$$\frac{d_n}{-z} = \frac{x_p}{x} \qquad \frac{d_n}{-z} = \frac{y_p}{y}$$

$$x_p = x\left(\frac{-d_n}{z}\right) \qquad y_p = y\left(\frac{-d_n}{z}\right)$$

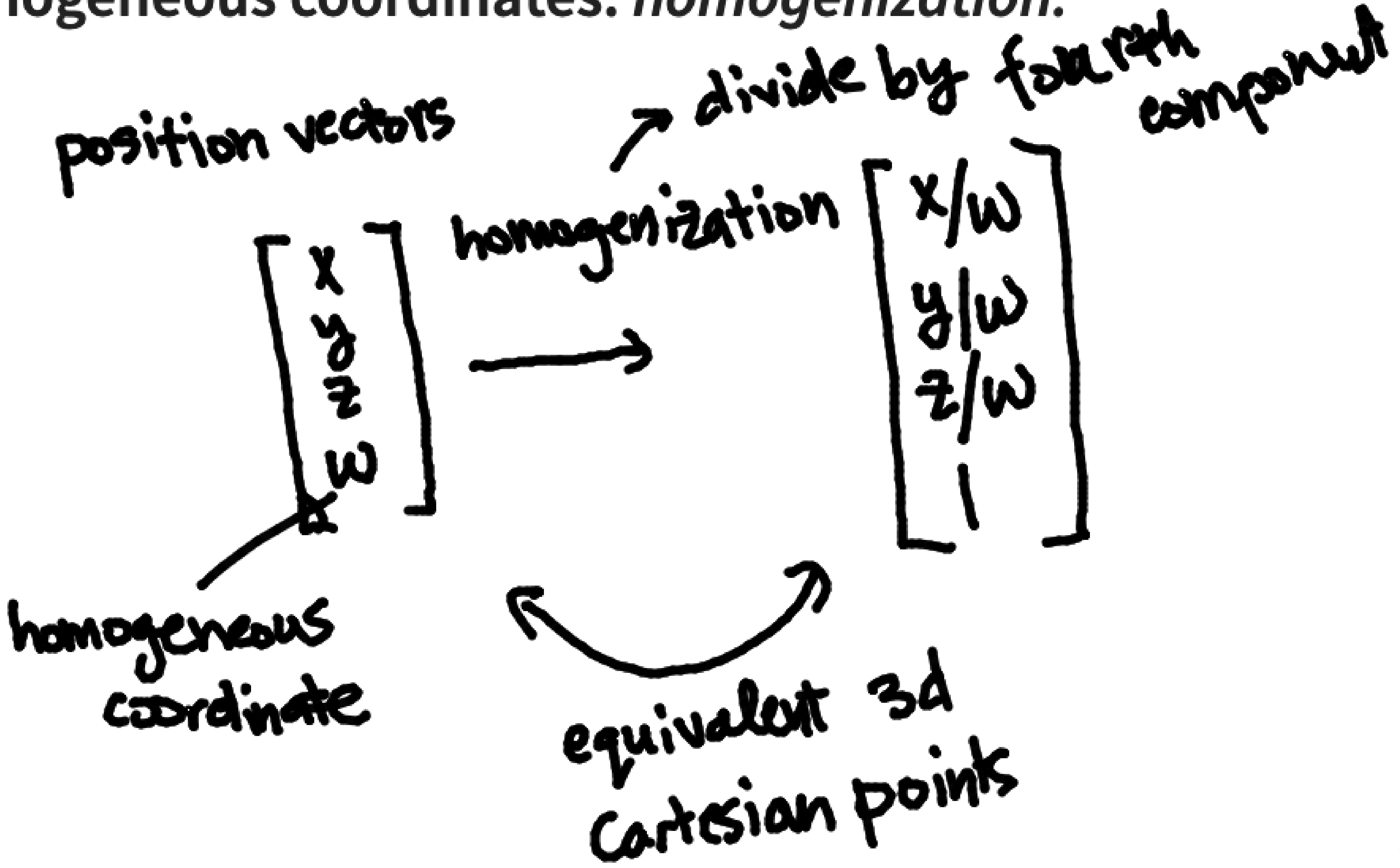
:($\frac{1}{z}$ not linear

$$M_p = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

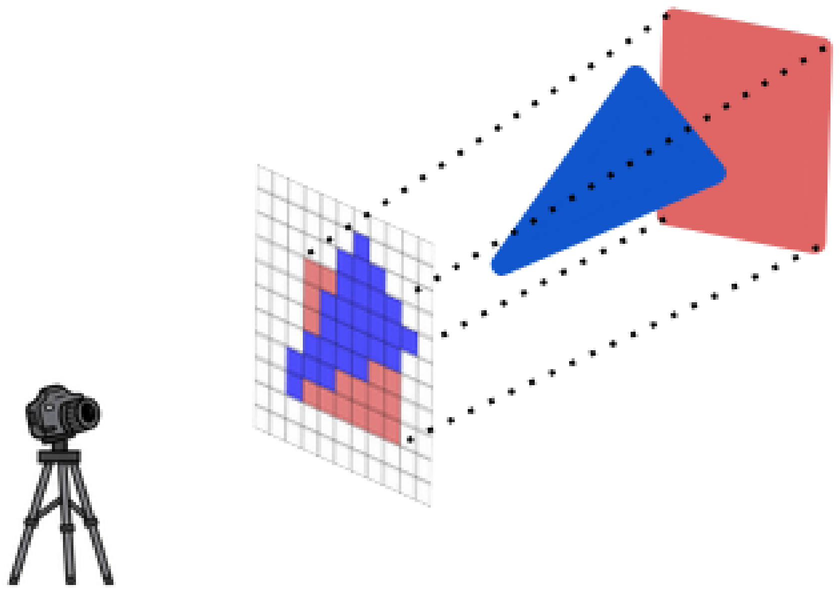




Oh wait, let me tell you one more thing about homogeneous coordinates: *homogenization.*



If we remember to always homogenize the result, then we can write a perspective projection matrix.



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

homogenize

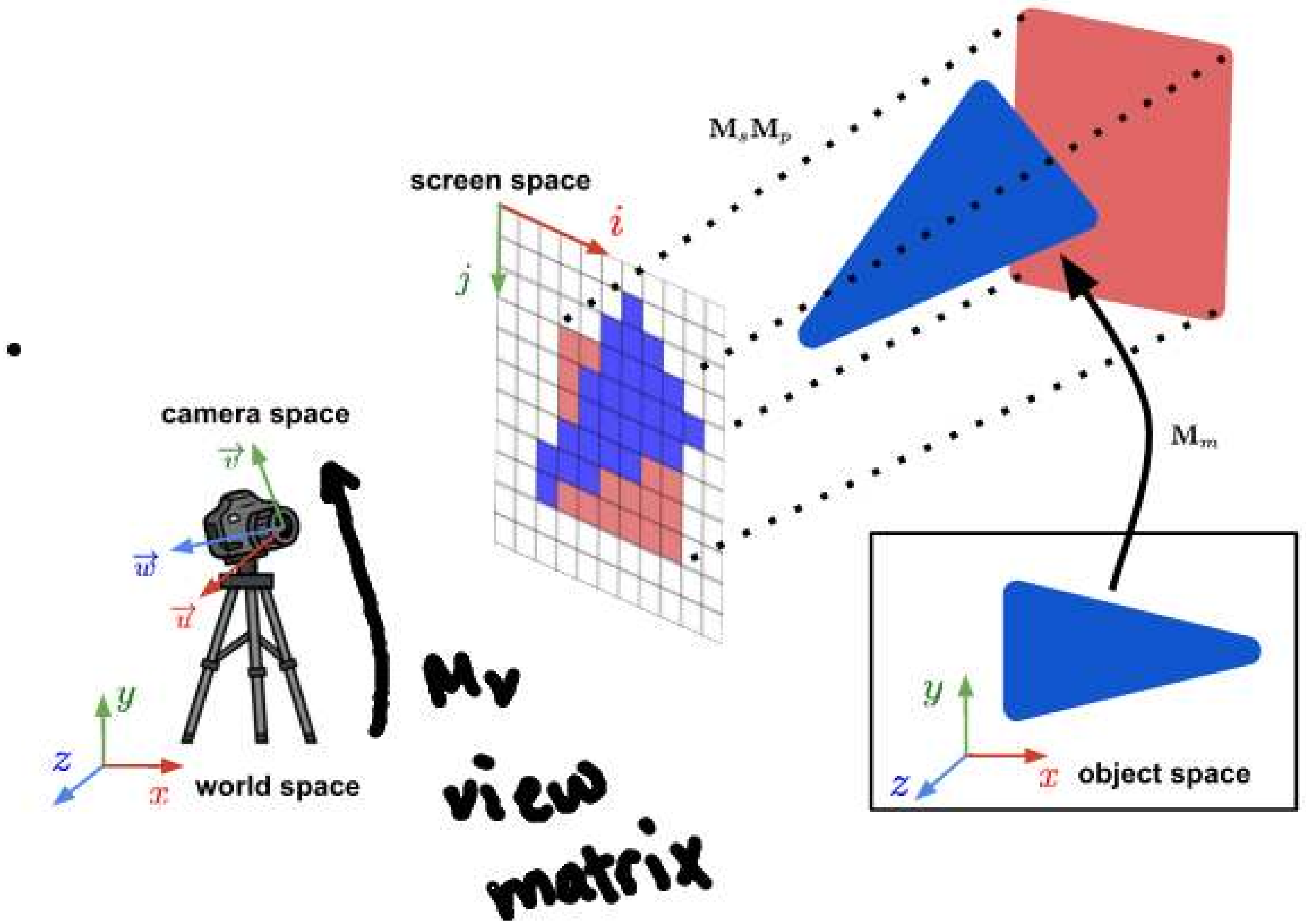
$$x_p = x\left(\frac{-d_n}{z}\right), \quad y_p = y\left(\frac{-d_n}{z}\right), \quad z_p = d_n$$

$$\begin{bmatrix}-d_n x/z\\ -d_n y/z\\ d_n\\ 1\end{bmatrix}$$

$$\frac{x}{(-z/d_n)} = \frac{-d_n}{z}\cdot x$$

$$\frac{y}{(-z/d_n)} = \frac{-d_n}{z}\, y \qquad \frac{-z}{(-z/d_n)} = d_n$$

Nomenclature: Model-View Matrix & MVP matrix.



model-view matrix: $M_v M_m$

MVP matrix

model-view-projection matrix: $M_p M_v M_m$

Introduction to WebGL: GPU-based rasterization API.


```
attribute vec3 a_Position;
attribute vec3 a_Normal;
uniform mat4 u_ModelViewMatrix;
uniform mat4 u_NormalMatrix;
uniform mat4 u_ModelViewProjectionMatrix;
varying vec3 v_Normal;
varying vec3 v_Position;
void main() {
  gl_Position  = u_ModelViewProjectionMatrix * vec4(a_Position, 1.0);
  v_Normal = mat3(u_NormalMatrix) * a_Normal;
  v_Position = (u_ModelViewMatrix * vec4(a_Position, 1.0)).xyz;
};
```

```
precision highp float;

void main() {
  gl_FragColor = vec4(color, 1.0);
}
```

Example GLSL syntax with vectors & matrices:

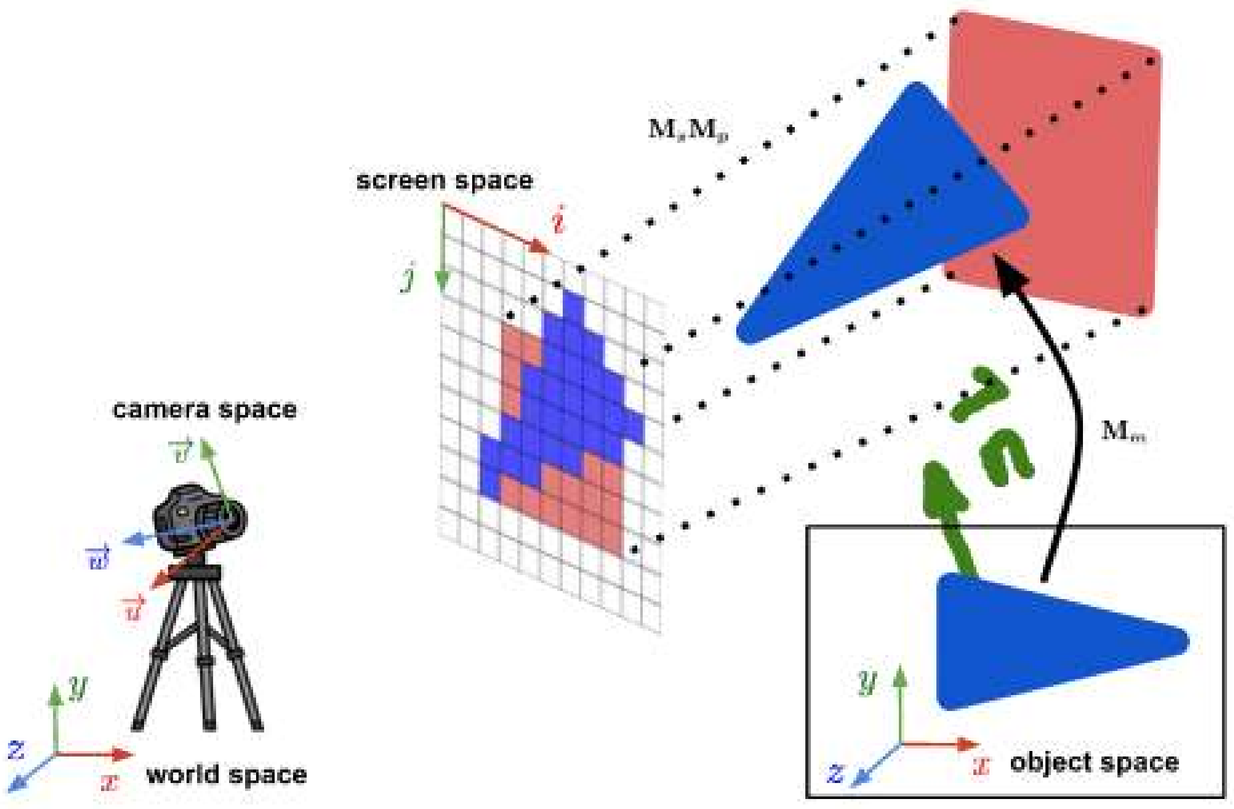
```
vec3 u = vec3(1, 2, 3);
vec3 v = vec3(4, 5, 6);
vec3 u_normalized = normalize(u);
float u_length = length(u);
vec3 u_scaled = 2.0 * u; // 2 * u will result in a compiler error (2 is an integer, but 2.0 is a float)
vec3 u_plus_v = u + v;
vec3 u_minus_v = u - v;
float u_dot_v = dot(u, v);
vec3 u_cross_v = cross(u, v);
vec3 u_times_v_componentwise = u * v;
vec3 reflected = reflect(u, v); // special built-in function to reflect the first vector across the other!
vec3 p = vec3(1, 2, 3); // some position vector
vec4 p_homogeneous = vec4(p, 1.0); // homogeneous representation of p

float u_x = u.x; // or u[0], also u.r
vec2 u_xy = u.xy;

mat3 A = mat3(1, 0, 0, 0, 1, 0, 0, 0, 1); // 3x3 identity, can also use mat3(1.0)
mat3 A_inverse = inverse(A);
mat3 A_transpose = transpose(A);
vec3 A_times_u = A * u;
```

Moving forward: always ask which frame of reference you're in.

When doing a lighting calculation in camera space, the normal vector (originally defined in object space), should be transformed by:



points transformed by model-view (MvMm)

How should normals be transformed? 13

inverse(transpose(Mv * Mm))
69%

inverse(transpose(Mm))
15%

Mv * Mm
8%

inverse(transpose(Mp * Mv * Mm))
8%

Mm
0%

Mv
0%

inverse(transpose(Mv))
0%

Edit response