Final Exam Review

CS425: Computer Graphics I

Khairi Reda

Exam format

- Final exam date: **Wednesday 1-3pm (ERF 1023, this room)**
- **Closed books** — notes, slides, books, computers, cell phones **not allowed**
- 20% of the total grade for the course

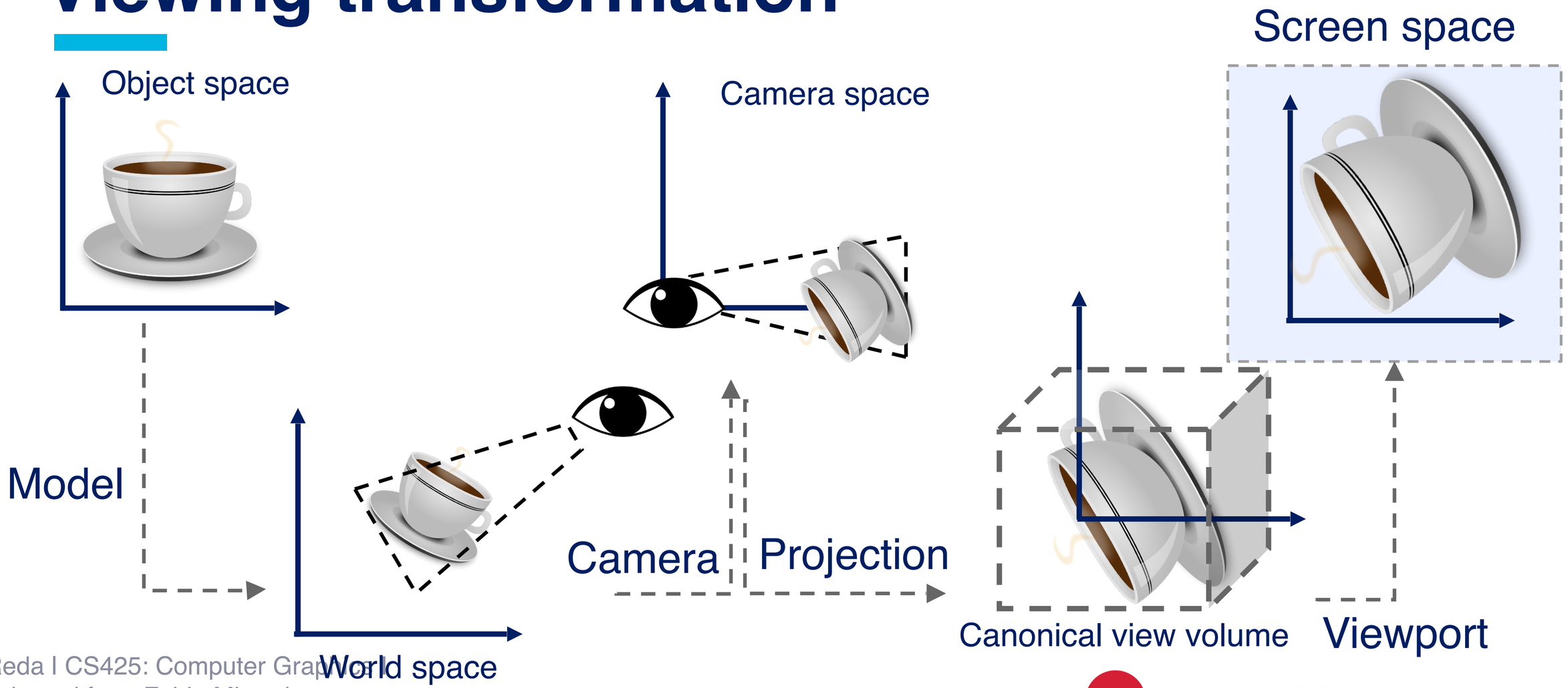
- **Format: Multiple choice + open-ended questions**
  - e.g., compose transformations for a specified model/view config
  - Explain how to shade and compute lighting effect
  - Provide pseudo code for an algorithm (e.g., raster, ray tracer, phong/Gourad shading)
- **Please write legibly!**

Disclaimer

- The following is intended as a review of the main concepts we covered in class
- It is NOT an exhaustive list of everything that will be on the exam
- Feel free to stop me at any point if there are questions
-

Model transformations

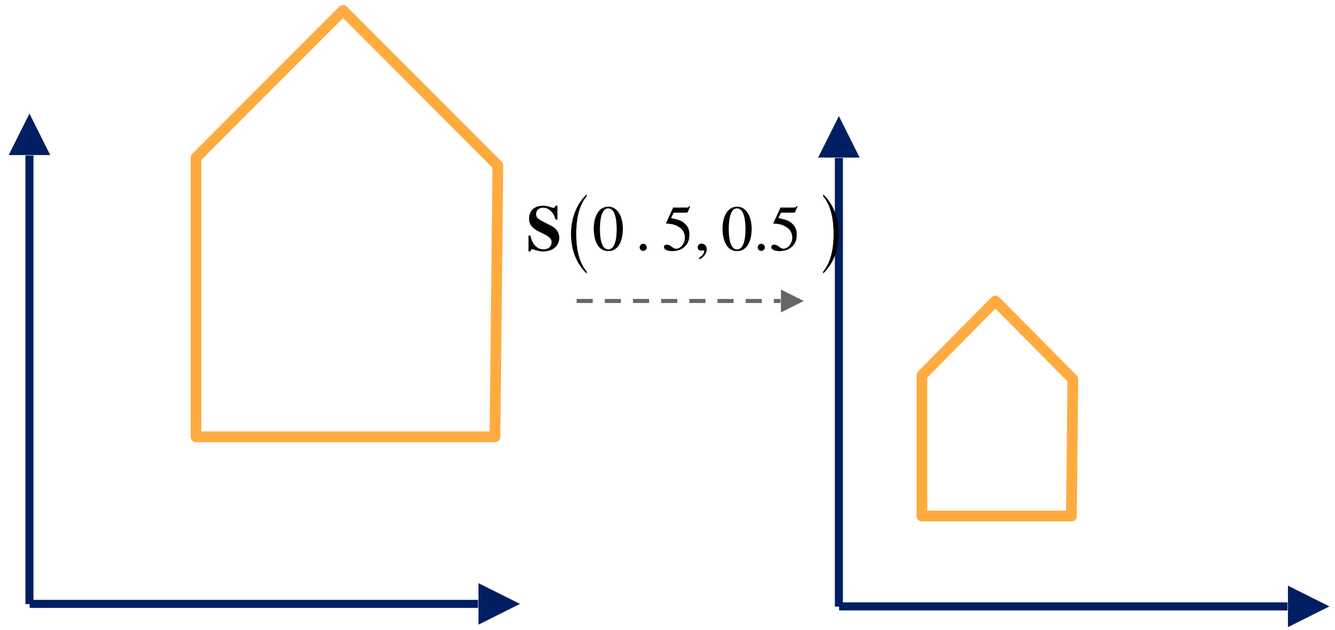
Viewing transformation



Scaling in 2D

- A scaling matrix $\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y)$ scales an entity with factors s_x, and s_y along the x, and y directions.

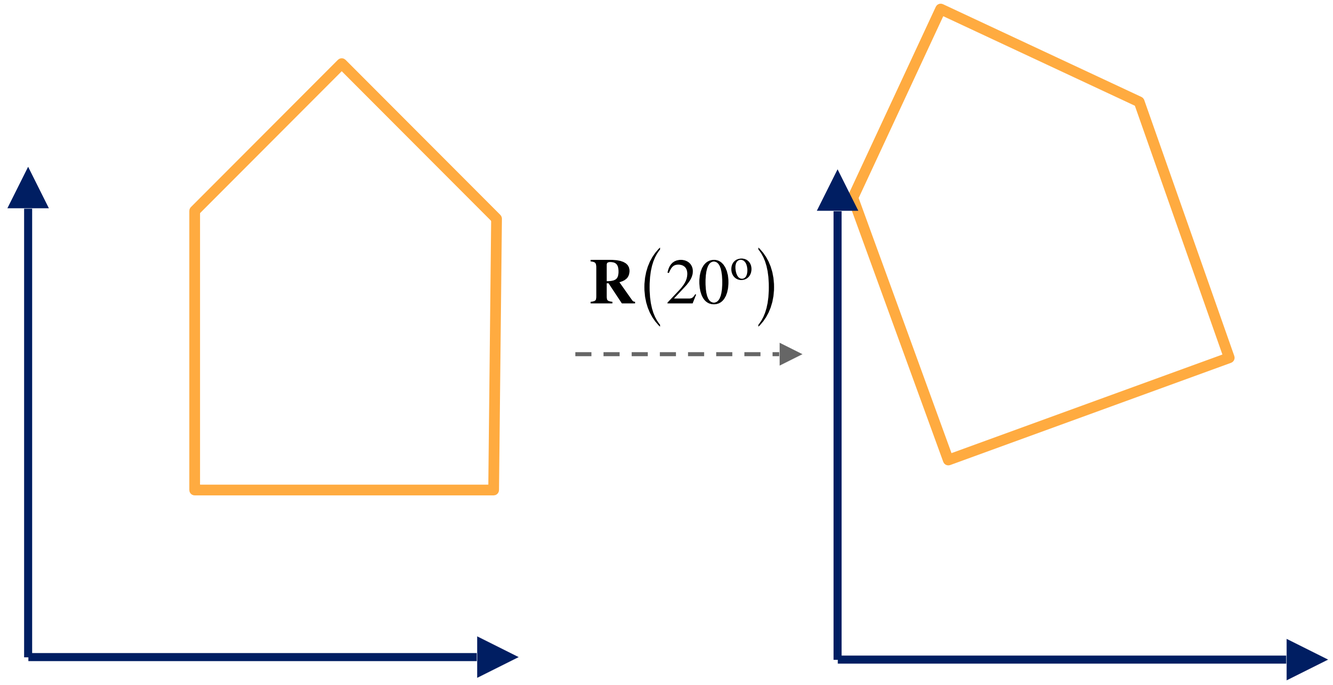
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}}_{\mathbf{S}(s_x, s_y)} \begin{pmatrix} x \\ y \end{pmatrix}$$



Rotation in 2D

- A rotation matrix $\mathbf{R}(\alpha)$ rotates an entity **around the origin** by α.

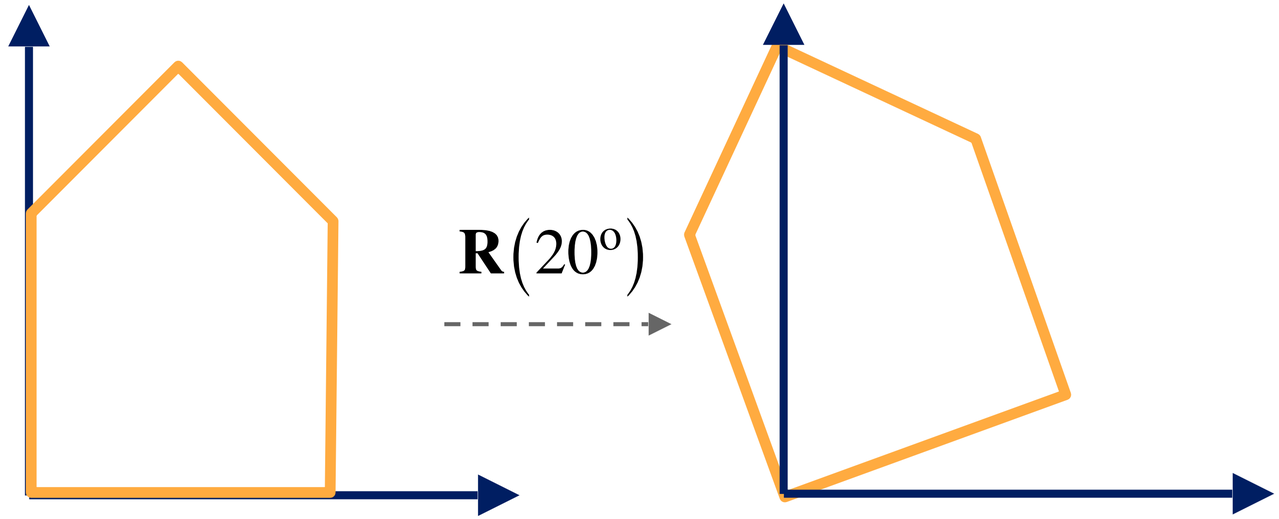
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix}}_{\mathbf{R}(\alpha)} \begin{pmatrix} x \\ y \end{pmatrix}$$



Rotation in 2D

- A rotation matrix $\mathbf{R}(\alpha)$ rotates an entity **around the origin** by α.

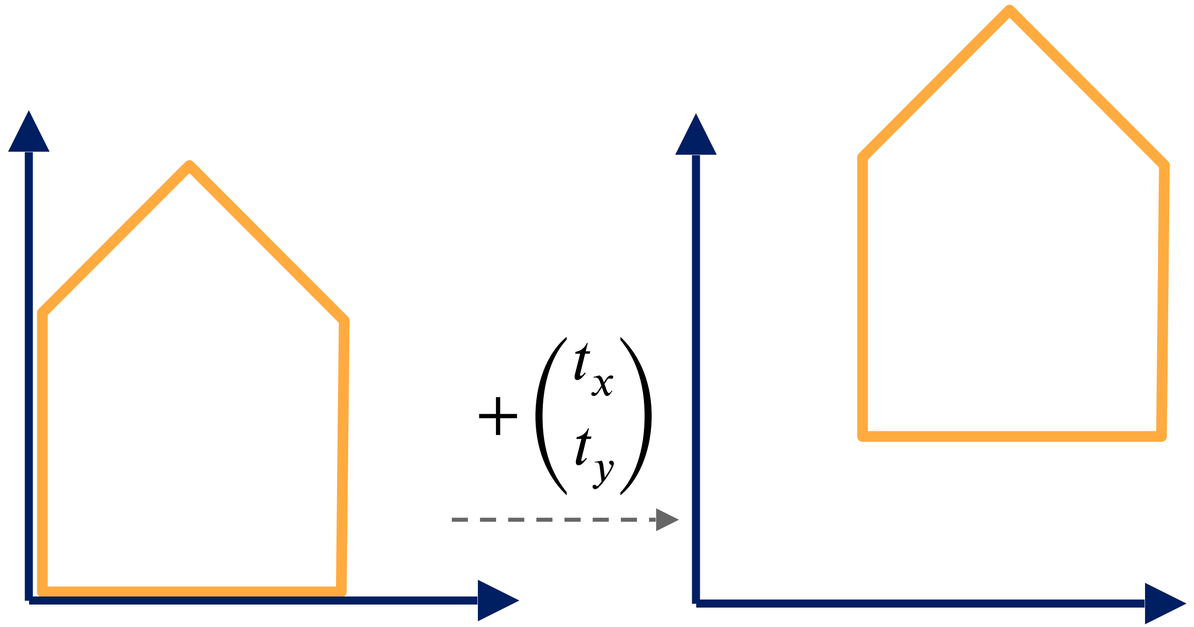
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix}}_{\mathbf{R}(\alpha)} \begin{pmatrix} x \\ y \end{pmatrix}$$



Translation

- Translating an entity?

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$



- Matrix representation?

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{T}(t_x, t_y) \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Homogeneous coordinates

- Add an extra component:
  - 2D point: $(x, y, 1)^T$
  - 2D vector: $(x, y, 0)^T$
- Matrix representation of translations:

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{T}(t_x, t_y)} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

Summary: Transformations in 2D

Scaling

$$\mathbf{S}\left(s_x, s_y\right) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation

$$\mathbf{R}(\alpha) = \begin{pmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Shearing

$$\mathbf{H}(s,t) = \begin{pmatrix} 1 & s & 0 \\ t & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Translation

$$\mathbf{T}\left(t_x, t_y\right) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Concatenation of transformations

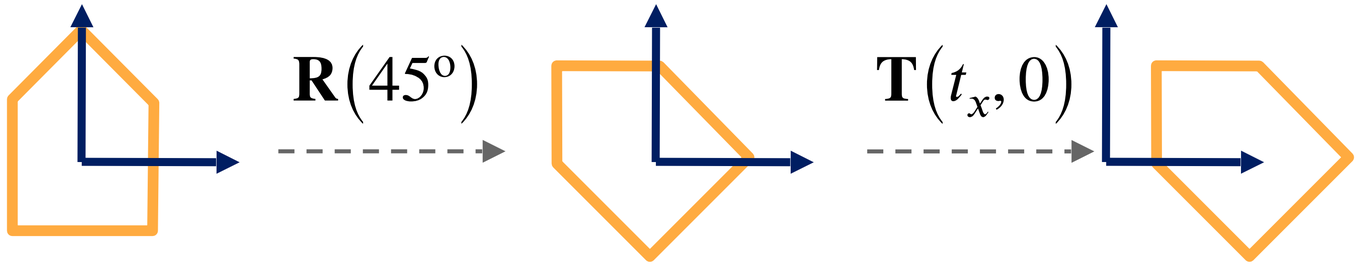
- Sequence of affine maps $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$
- Concatenation by matrix multiplication:

$$\mathbf{A}_n\Big(\ldots\mathbf{A}_2\big(\mathbf{A}_1(\mathbf{x})\big)\Big) = \mathbf{A}_n\ldots\ \mathbf{A}_2\ \mathbf{A}_1\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

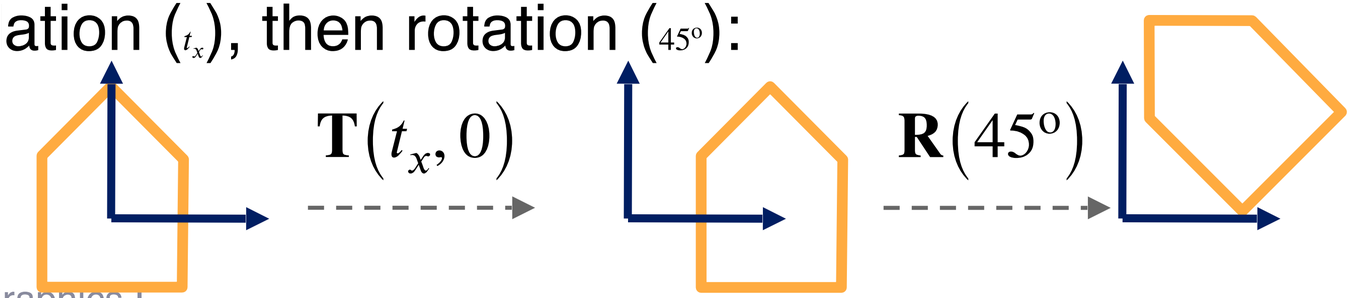
- Very important for performance!
- Matrix multiplication is generally not commutative, ordering is important!

Rotation and translation

- Matrix multiplication is not commutative!
- First rotation (45^o), then translation (t_x):

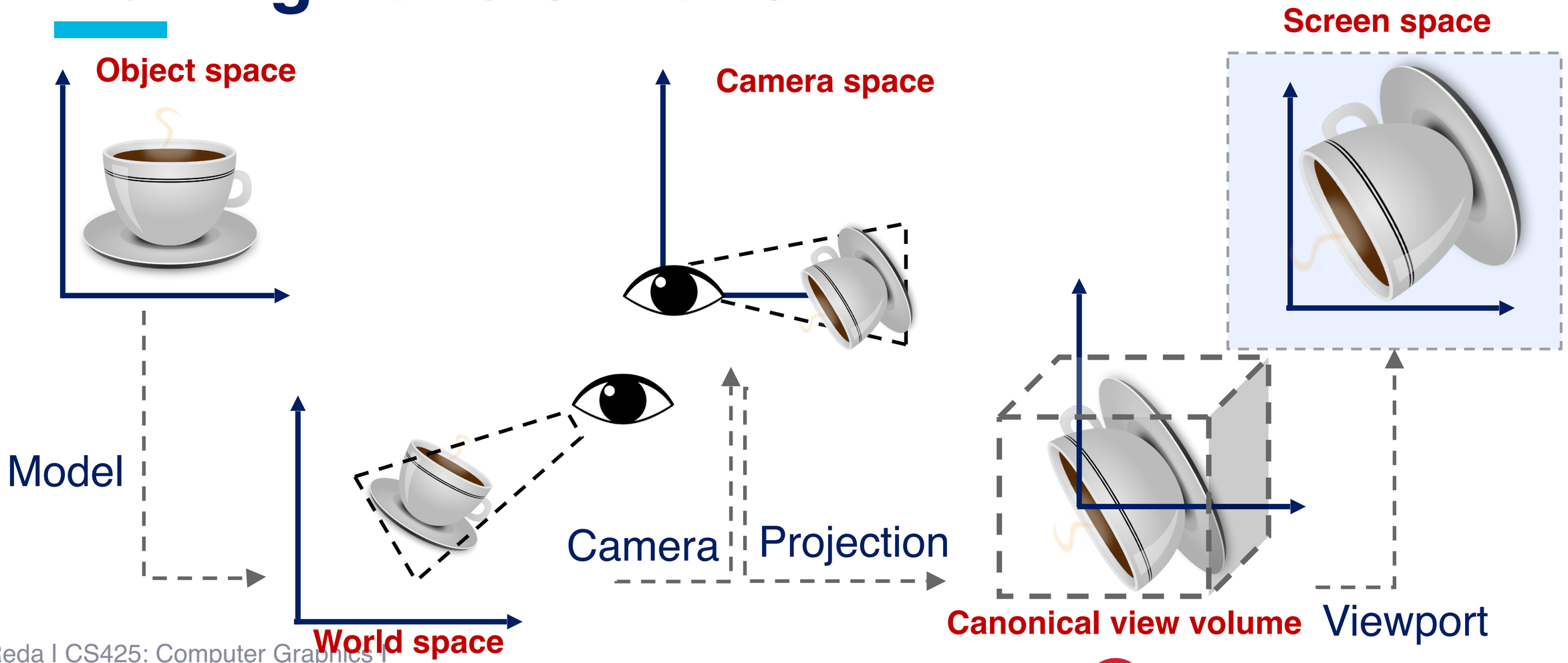


- First translation (t_x), then rotation (45^o):



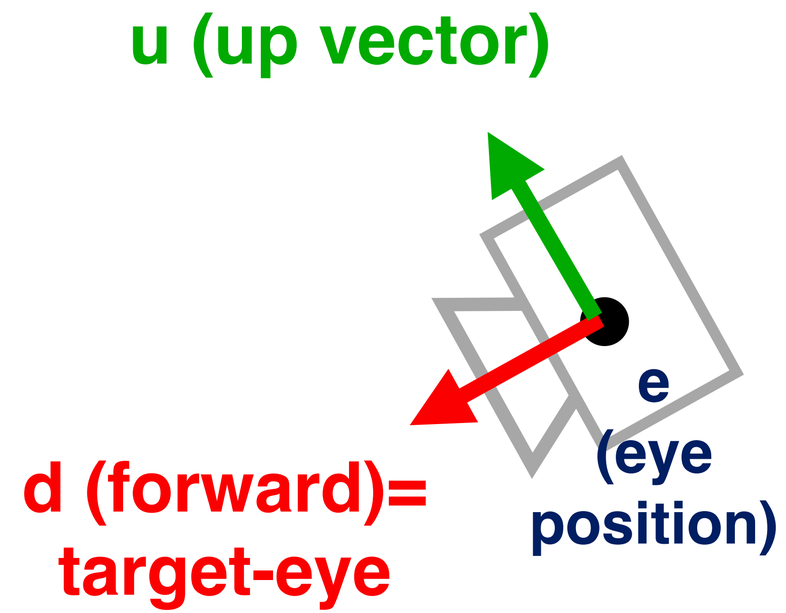
Camera (view) transform

Viewing transformation

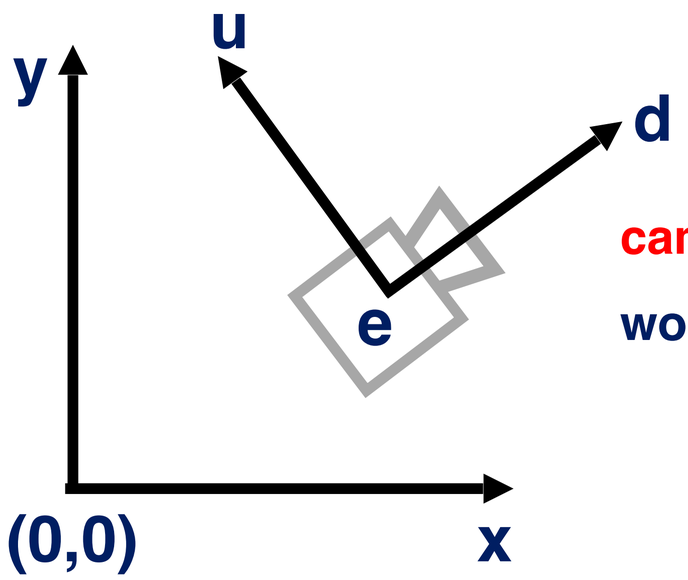


Camera transformation

- Construct the camera reference system:
  - The eye position e.
  - The forward direction d.
  - The view-up vector u.
- We need a matrix that transforms all coordinates into camera coordinates.



Change of frame



Rotation: $R = \begin{pmatrix} d_x & u_x & 0 \\ d_y & u_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Translate: $\mathbf{T} = \begin{pmatrix} 1 & 0 & \mathrm{e_x} \\ 0 & 1 & \mathrm{e_y} \\ 0 & 0 & 1 \end{pmatrix}$

cam2world $\mathbf{M}_{e \to o} = \mathbf{TR}$

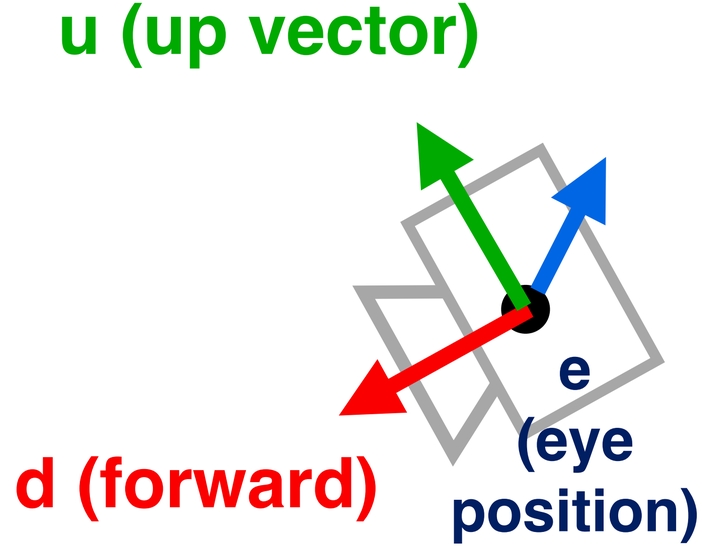
world2cam $\mathbf{M}_{o \to e} = (\mathbf{TR})^{-1} = \mathbf{R}^{-1}\mathbf{T}^{-1}$

We know:

$\mathbf{R}^{-1} = \mathbf{R}^{\mathrm{T}}$ (Pure rotation is orthogonal)

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & -\mathrm{e_x} \\ 0 & 1 & -\mathrm{e_y} \\ 0 & 0 & 1 \end{pmatrix}$$

Camera transformation (3D)



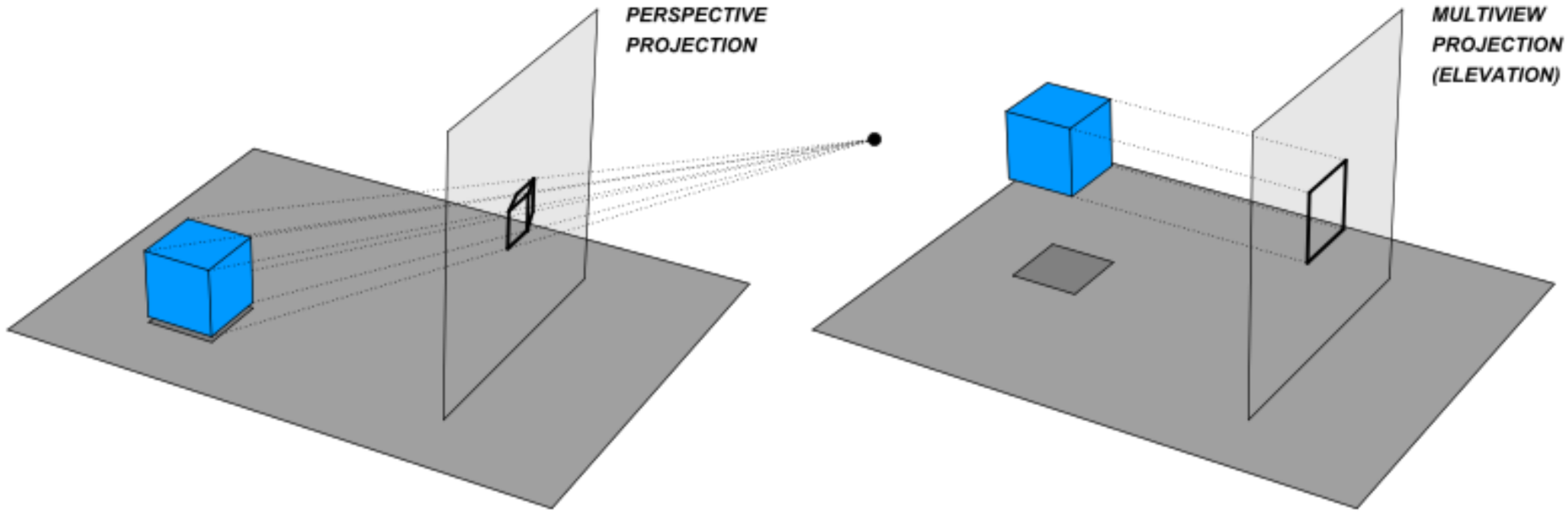
- Construct the camera reference system:
  - The eye position $\mathbf{e}$.
  - The forward direction $\mathbf{d}$.
  - The view-up vector $\mathbf{u}$.
  - Cross product of **u** and **d** to get the 3rd orthogonal vector
- A view matrix transform all coordinates into view coordinates.

$$\mathbf{M}_{camera2world} = \begin{pmatrix} \mathbf{u}\times\mathbf{d} & \mathbf{u} & \mathbf{d} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ (4x4 matrix)

$$\mathbf{M}_{world2camera} = \begin{pmatrix} \mathbf{u}\times\mathbf{d} & \mathbf{u} & \mathbf{d} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

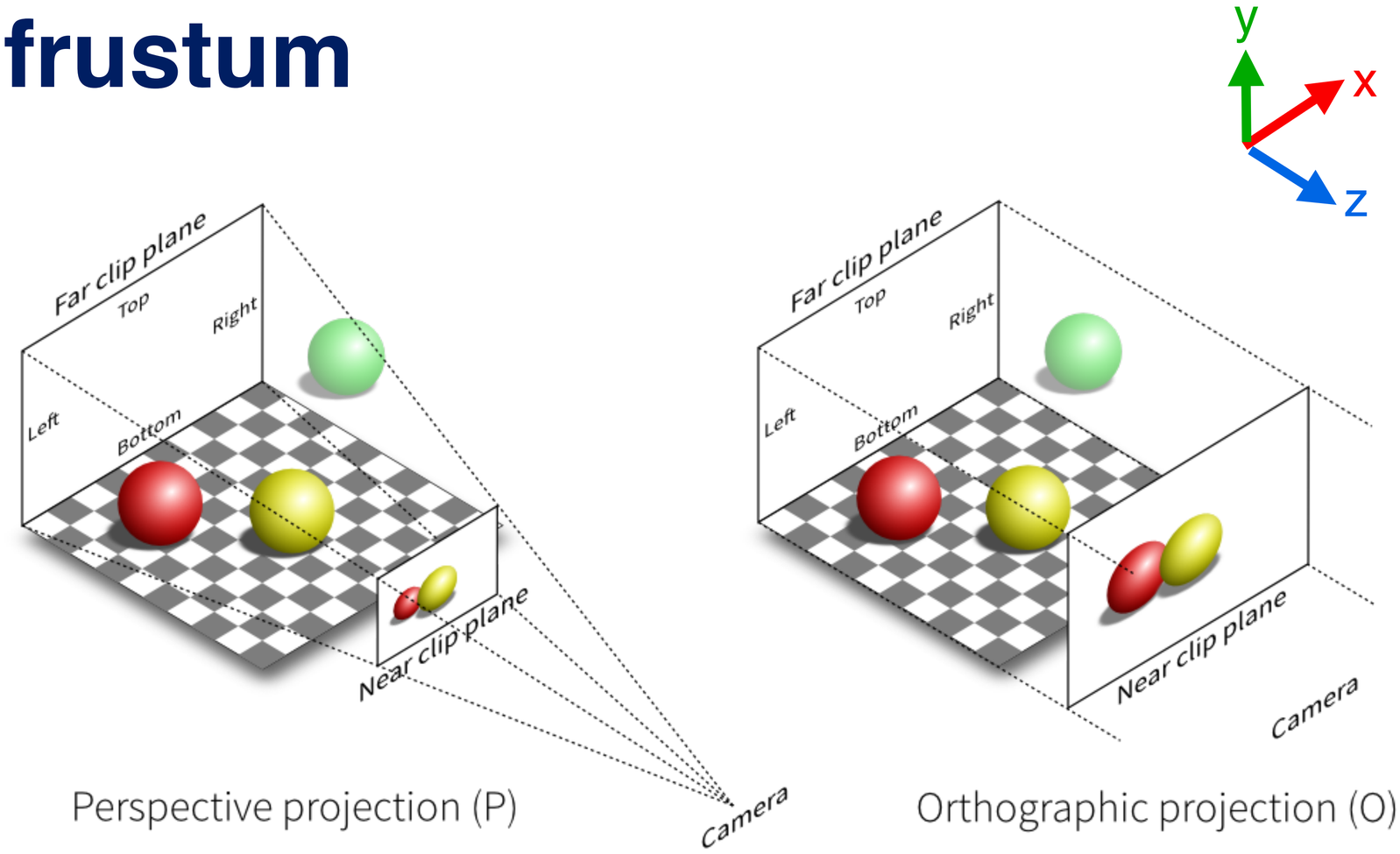
Projection transforms

Orthographic and perspective projections

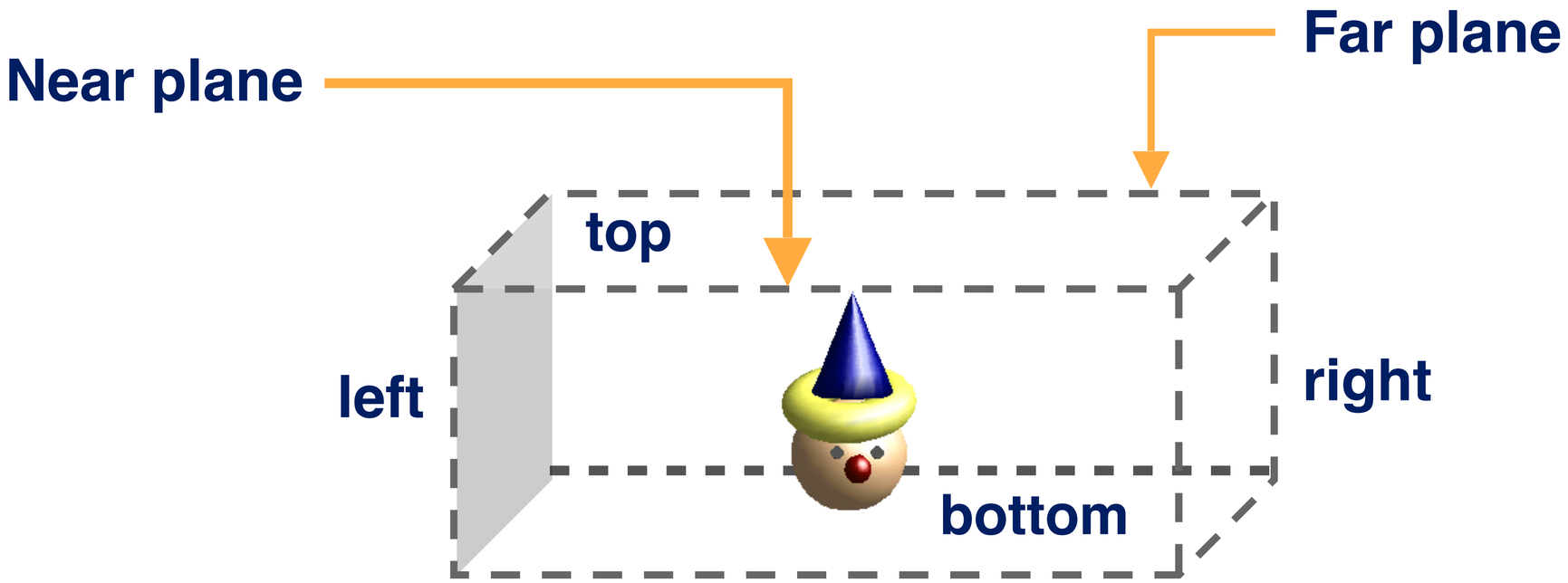


https://en.wikipedia.org/wiki/File:Various_projections_of_cube_above_plane.svg

View frustum



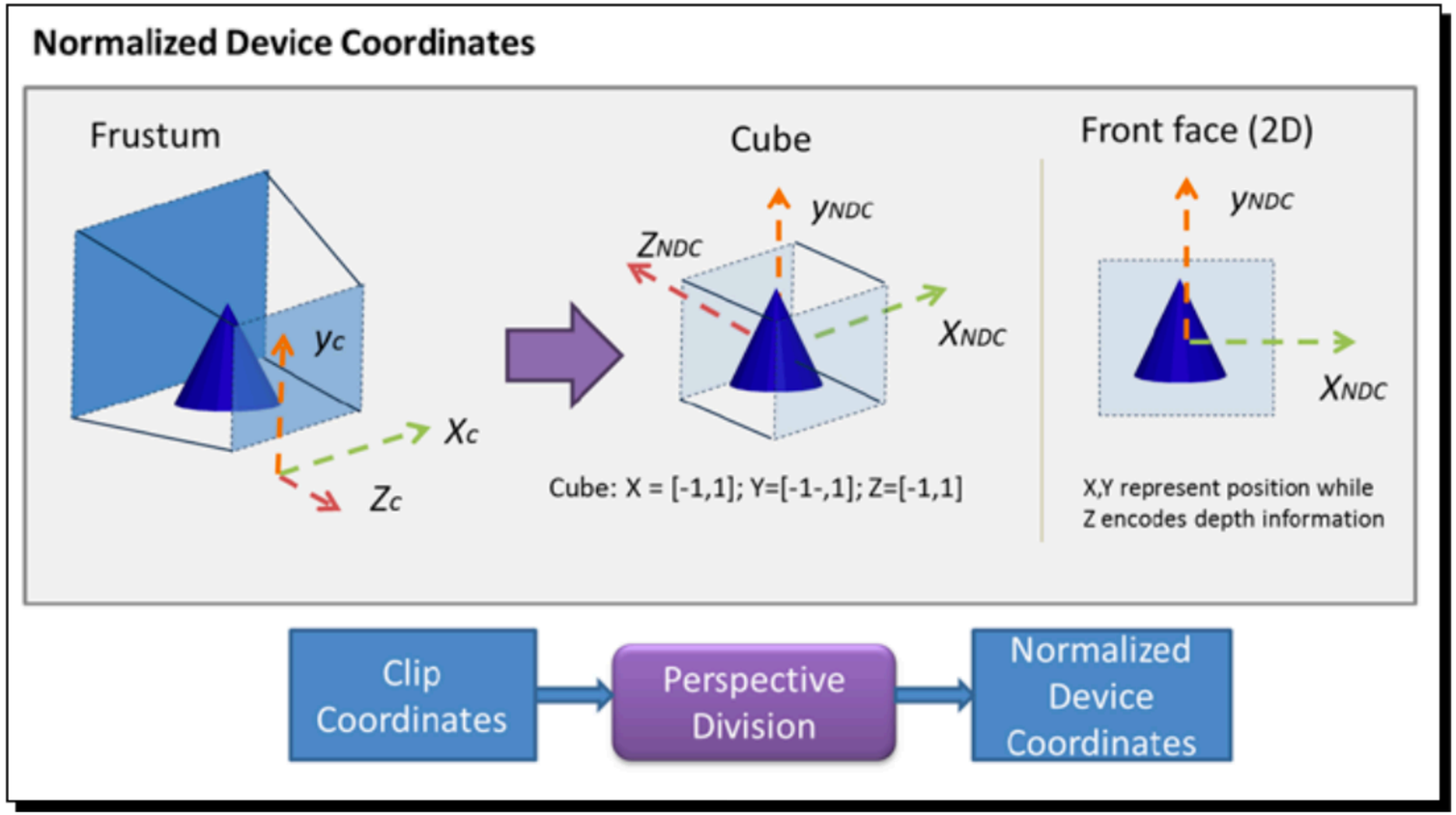
View frustum



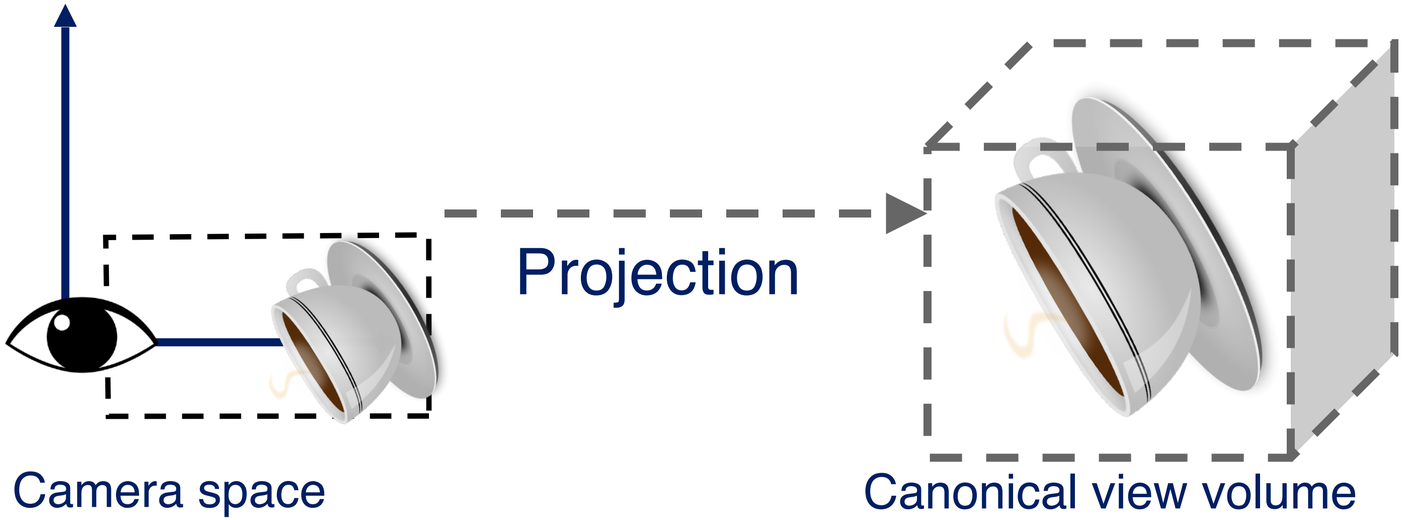
Camera space

$(x_{left}, y_{botton}, z_{near}) \times (x_{right}, y_{top}, z_{far})$

View frustum



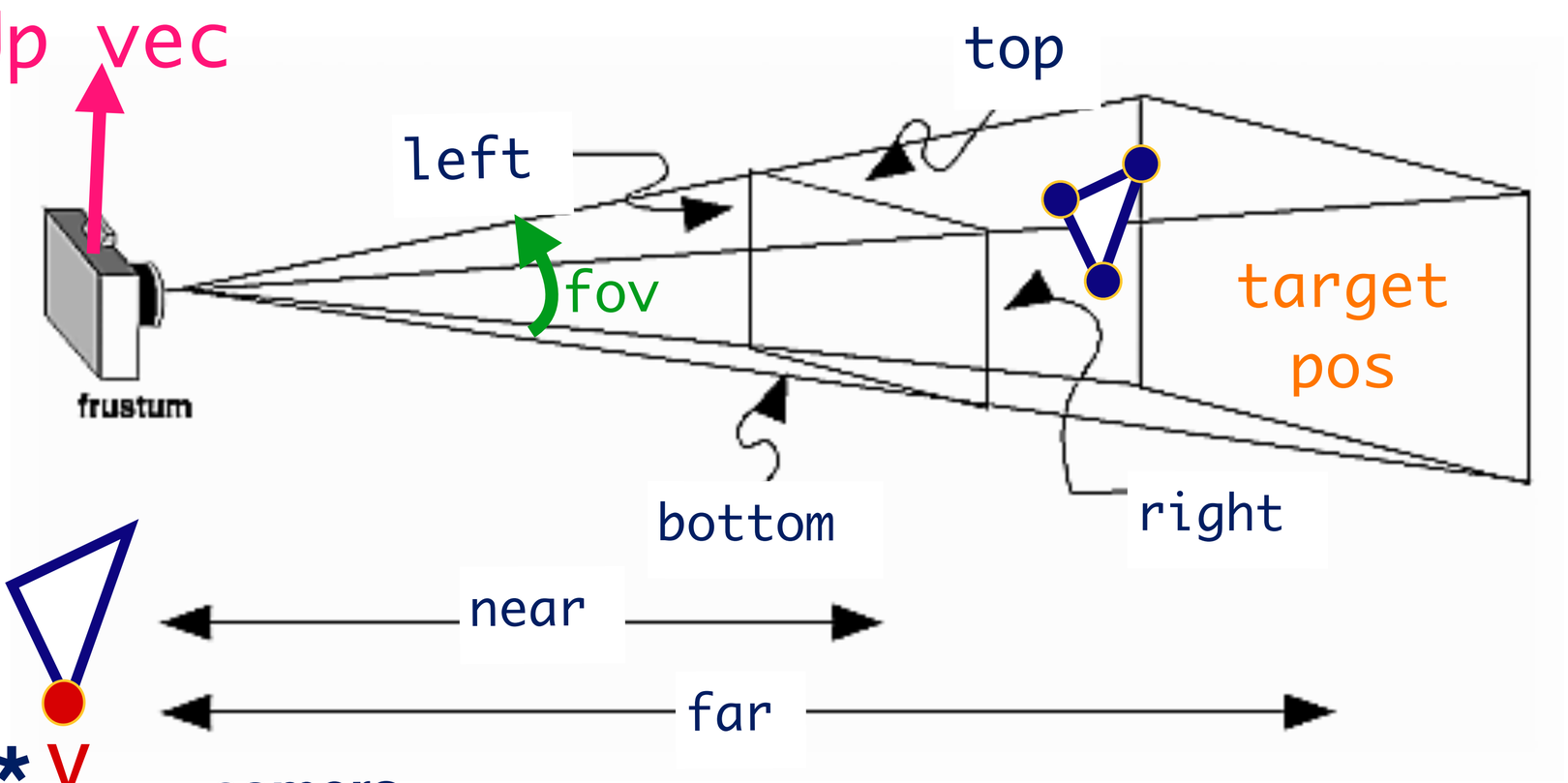
Orthographic transformation



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{(right-left)} & 0 & 0 & \frac{-(right+left)}{(right-left)} \\ 0 & \frac{2}{(top-bottom)} & 0 & \frac{-(top+bottom)}{(top-bottom)} \\ 0 & 0 & \frac{-2}{(far-near)} & \frac{-(far+near)}{(far-near)} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$(-1,-1,-1) \times (1,1,1)$

Anatomy of viewing + projection transform



Any transforms you want to do to your model (scaling, translation, rotation)

projection * view * model * v

camera (view matrix)

```
lookat(eye, target, up)
```

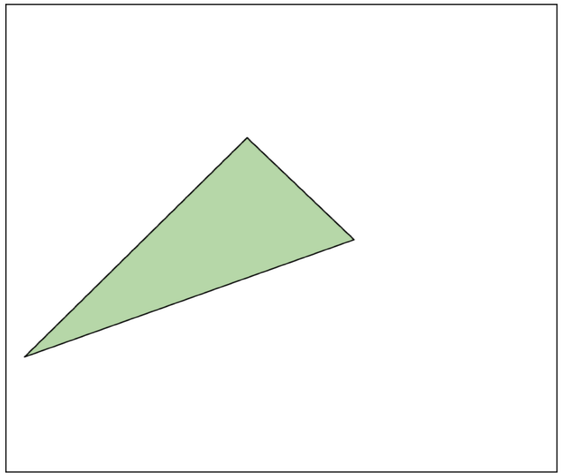
projection matrix

```
perspective(fov, w/h,
near, far)
```

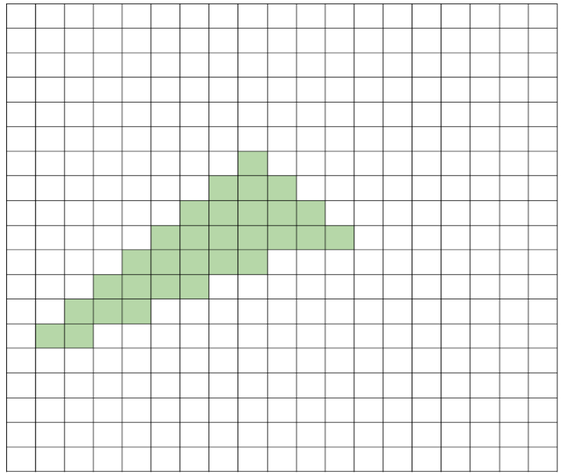
Rasterization

Rasterization

- Process of converting the vertices that are output from the clipping stage to fragments.
- Fragments are potential pixels.



Clipped object in vertex representation



Fragments of the rasterized object

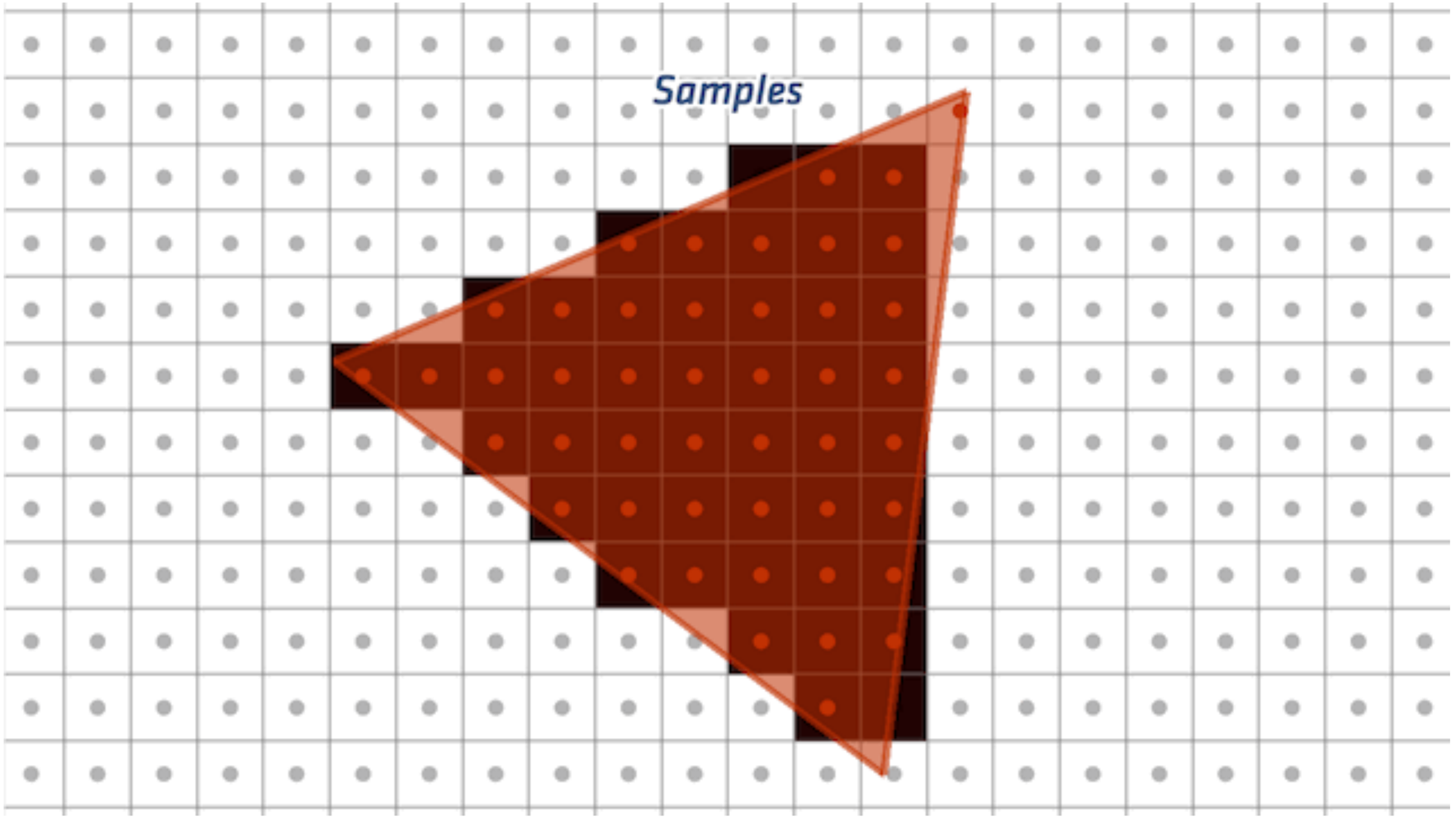


Image Copyright: Andrea Tagliasacchi

Rasterization: Lines

DDA (digital differential analyzer) algorithm:

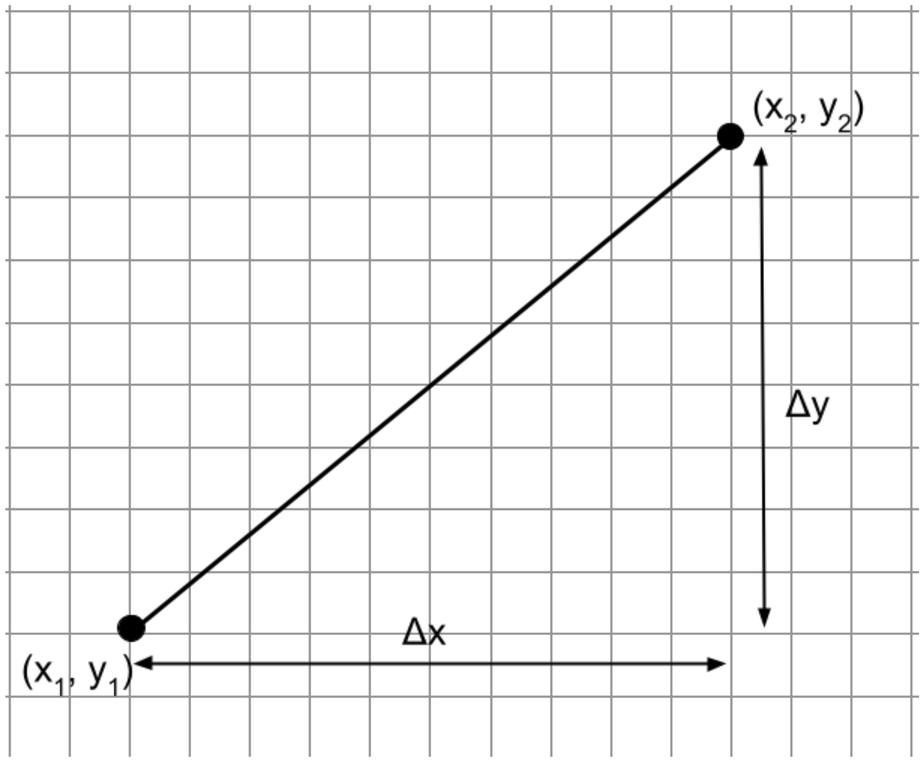
$y = mx + h$

$y_{i+1} = m\,(x_i + \Delta x) + h$

$y_{i+1} = mx_i + h + m\Delta x = y_i + m\Delta x$

If we set $\Delta x = 1$, then

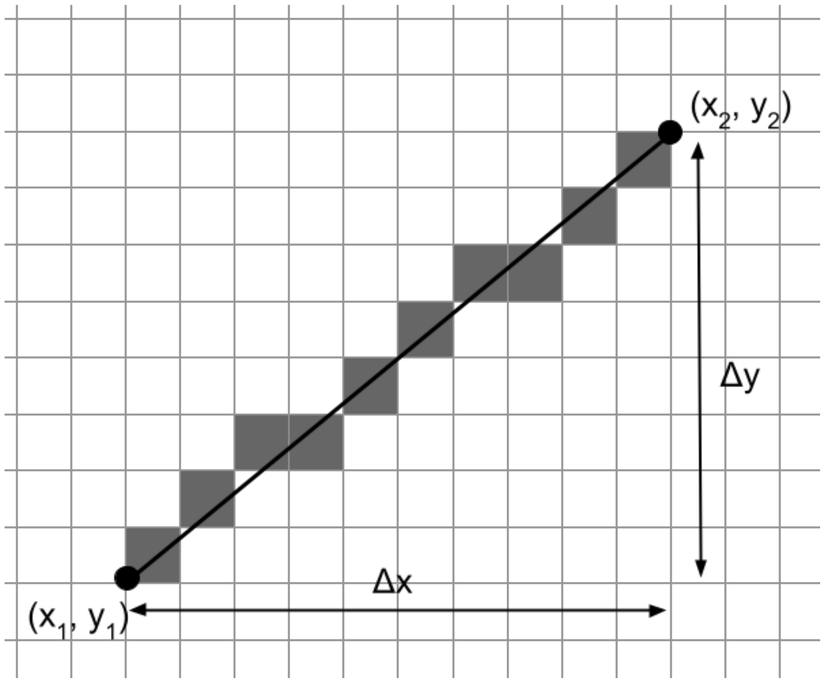
$\mathbf{y_{i+1} = y_i + m}$



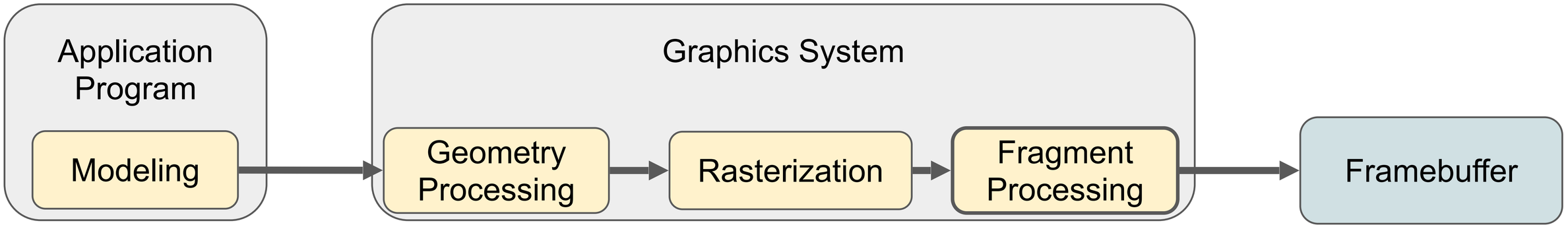
Rasterization: Lines

DDA algorithm

```
for (ix = x1; ix <= x2; ++ix) {
        y += m;
        writePixel(x, round(y), line_color);
}
```



Hidden Surface Removal

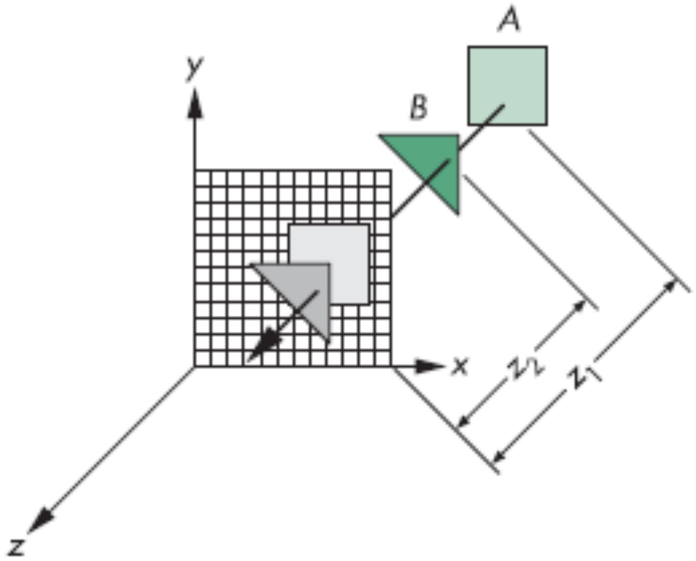


Hidden Surface Removal: The z-Buffer algorithm

A separate buffer to hold the depth information.

Initialized to maximum depth value from center of projection. Color buffer to the background color.

Iteratively rasterize polygons and simultaneously fill the z-buffer.

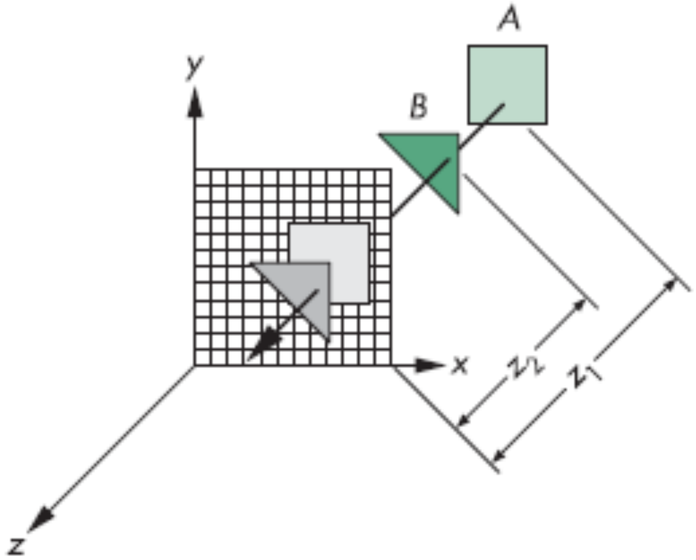


Hidden Surface Removal: The z-Buffer algorithm

Compare depth of incoming fragment with value in z-buffer.

$Depth_{new} > Depth_{z\text{-}buffer}$ we have already rasterized a fragment that is closer to the viewer.

Otherwise the incoming fragment is placed in the color buffer and the z-buffer is updated.



Lighting & shading

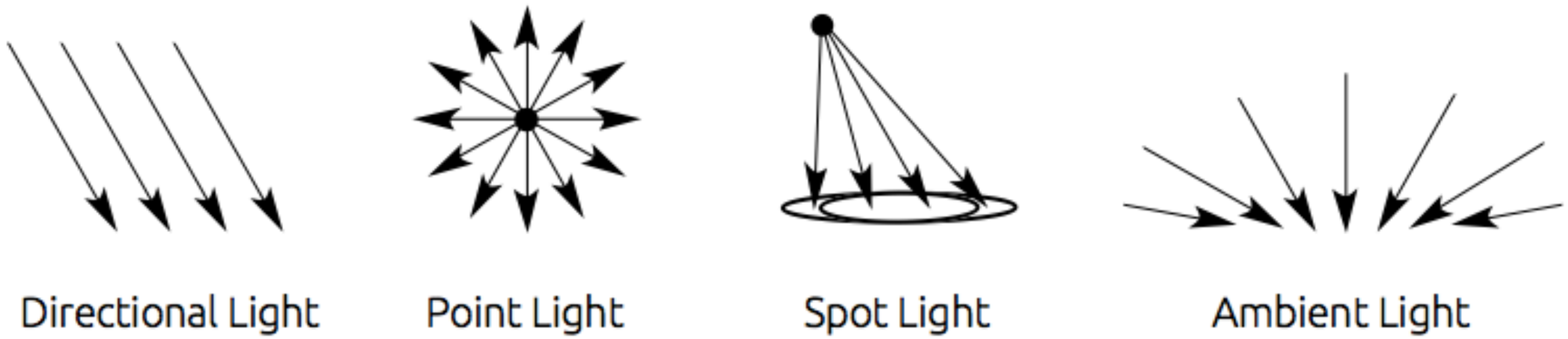
Lighting and shading

- Shading objects so their images appear three-dimensional.

- How can we model light-material interactions?
- We will see how to build a simple reflection model (Phong model) that can be used with real-time graphics hardware.

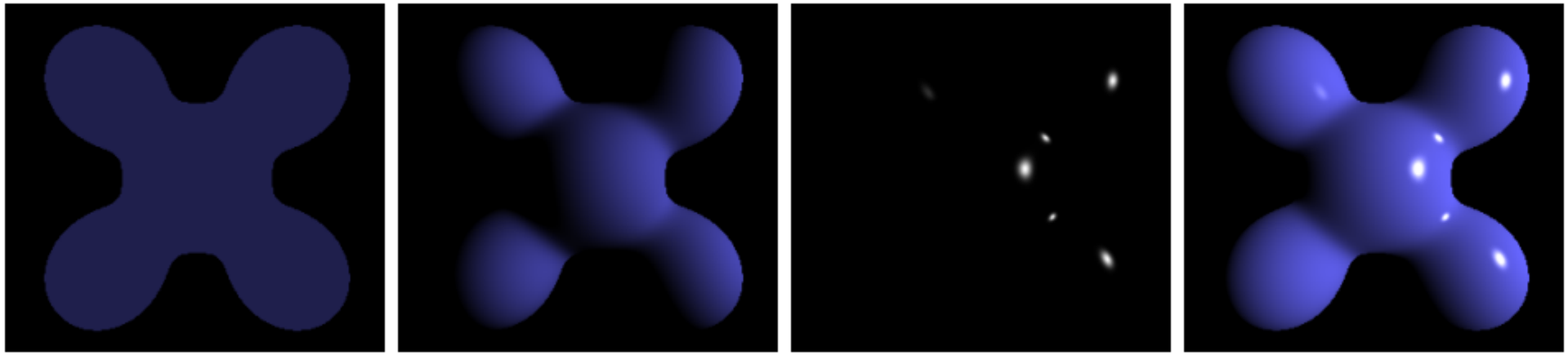
Light sources

- Point: position
- Spotlight: position and direction
- Directional: direction

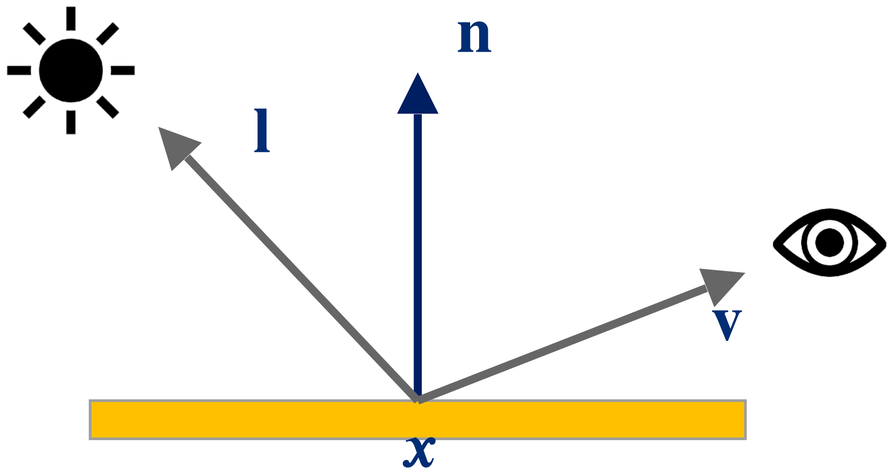


Phong model

$$f_{Phong}(\mathbf{L}_{light}, \mathbf{l}, \mathbf{v}, \mathbf{n}) = k_{ambient}\mathbf{L}_{ambient} + \sum_{m \in lights} k_{diffuse}(\mathbf{l}_m \bullet \mathbf{n})\mathbf{L}_{m,diffuse} + k_{specular}(\mathbf{r}_m \bullet \mathbf{v})^{\alpha}\mathbf{L}_{m,specular}$$



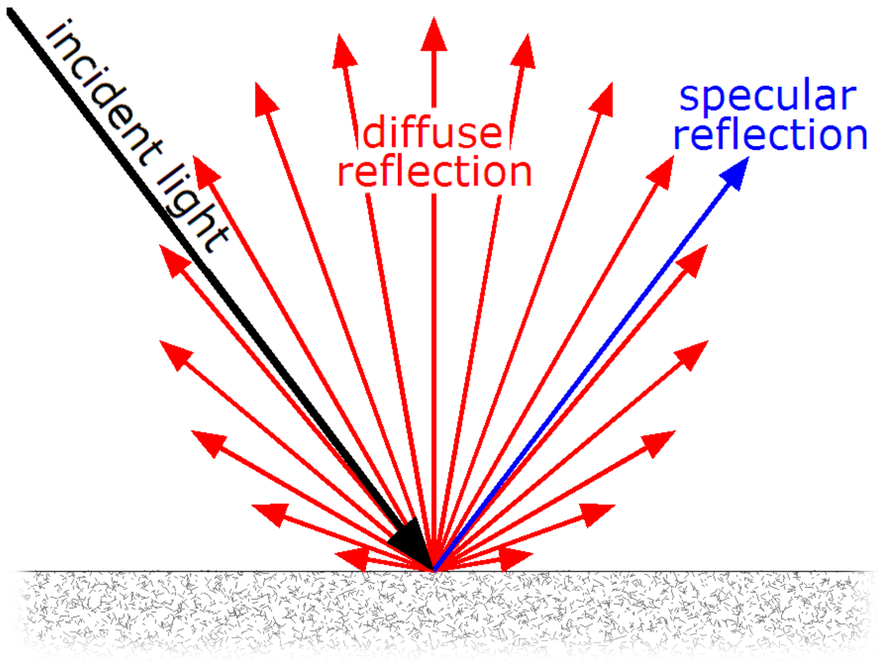
Phong model



$$L(\mathbf{x}, \mathbf{v}) = f_{Phong}(\mathbf{L}_{light}, \mathbf{l}, \mathbf{v}, \mathbf{n})$$

- f_{Phong} computes reflected light into direction $\mathbf{v}$ towards the sensor.

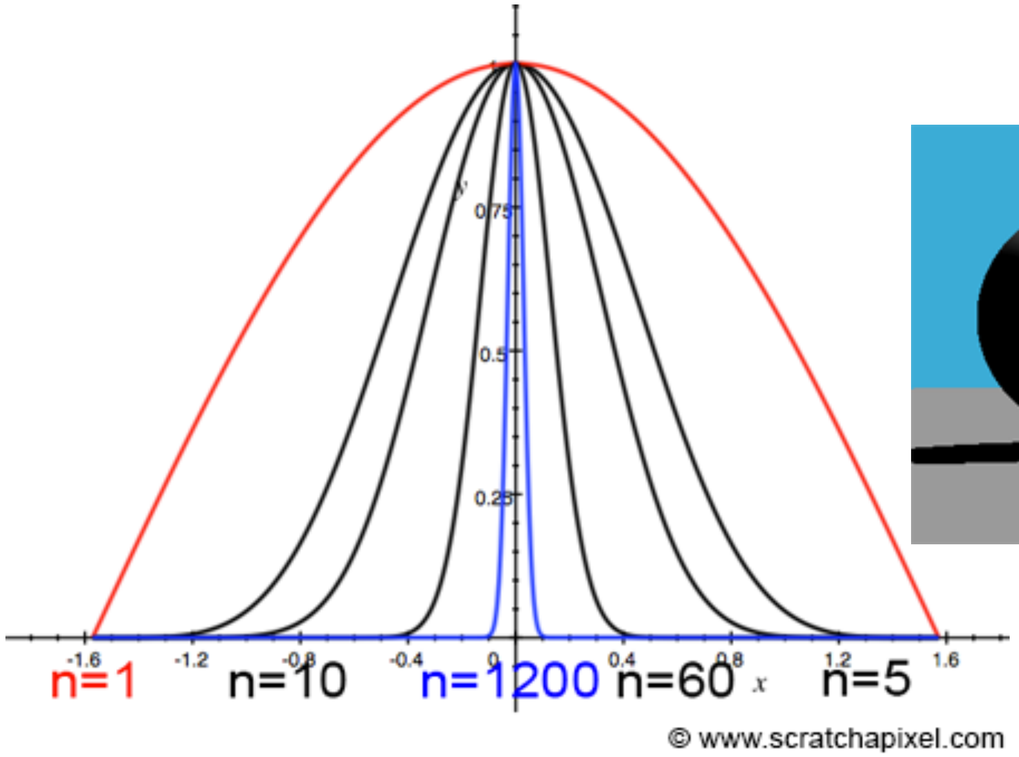
Phong model

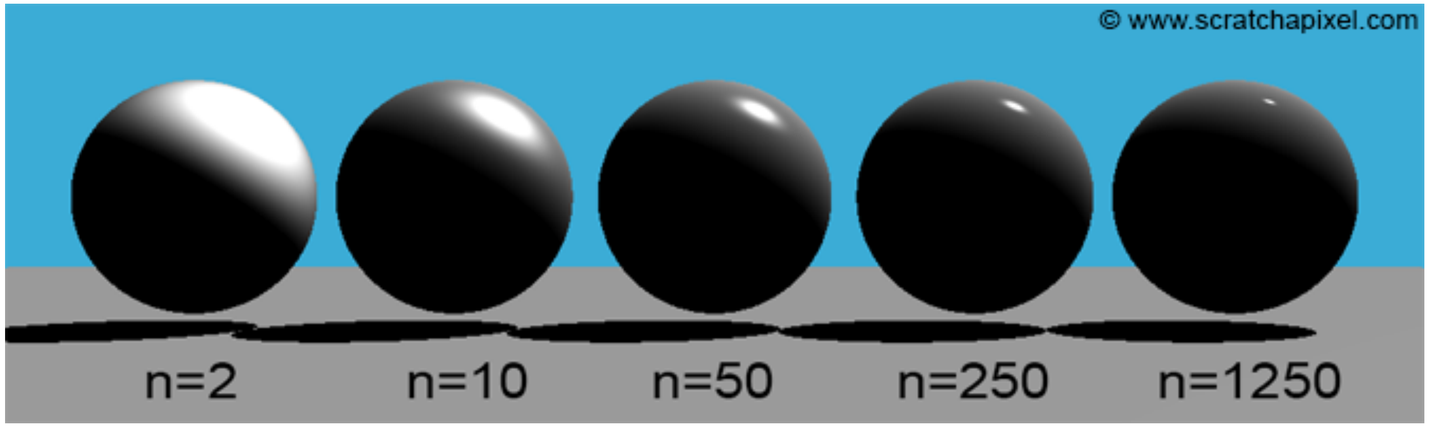


$$L(\mathbf{x}, \mathbf{v}) = f_{Phong}(\mathbf{L}_{light}, \mathbf{l}, \mathbf{v}, \mathbf{n})$$

- f_{Phong} computes reflected light into direction $\mathbf{v}$ towards the sensor.

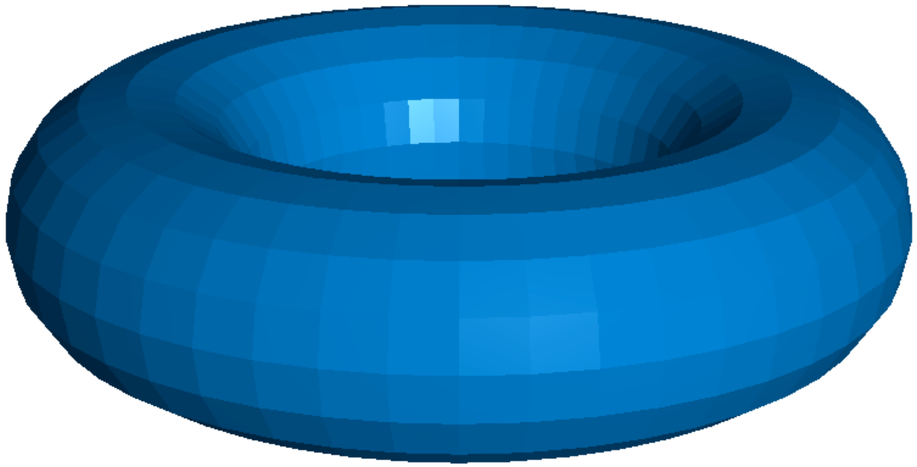
Specular exponent





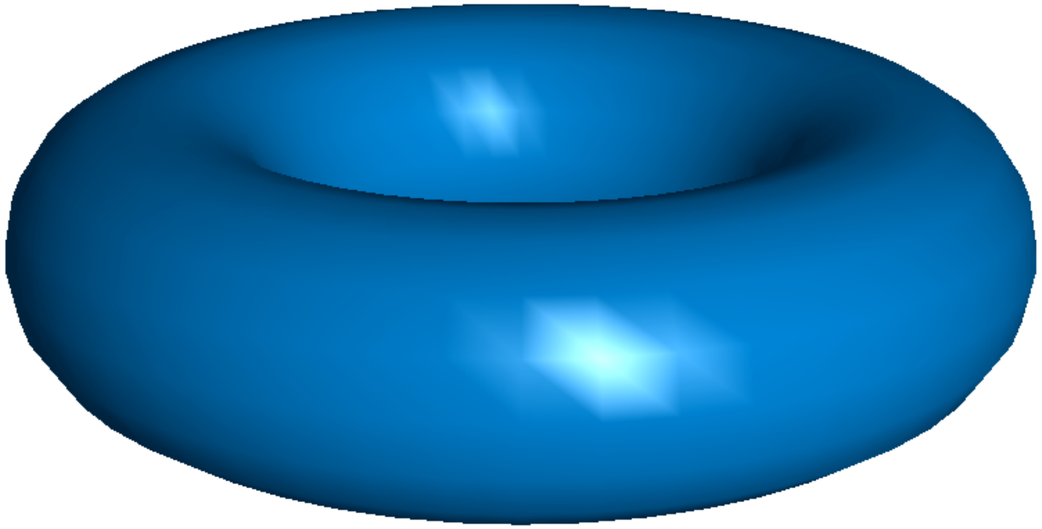
Flat shading

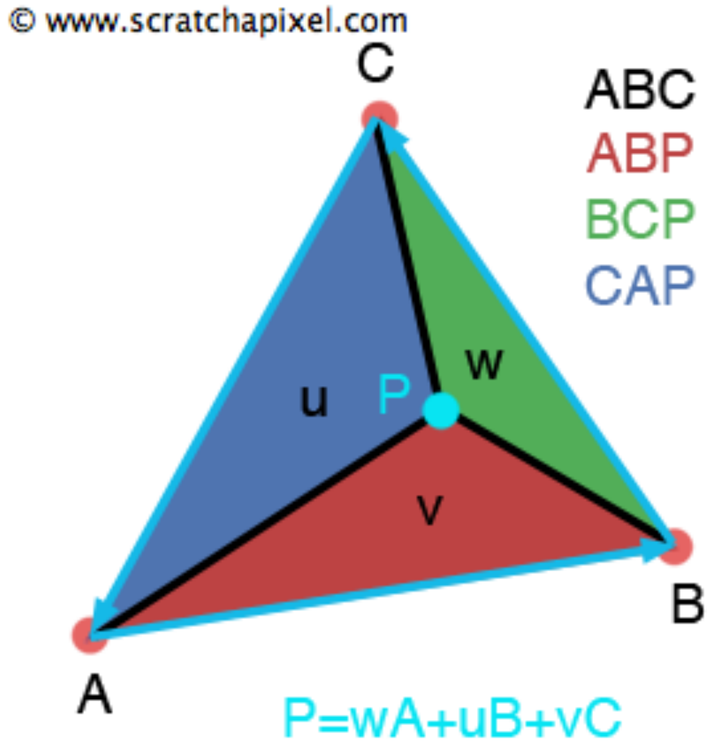
- Compute one color per polygon.
- All pixels in the same polygon are colored by the same color.



Gouraud shading

- Compute one color per vertex.
- Interpolate vertex **colors** across triangles.

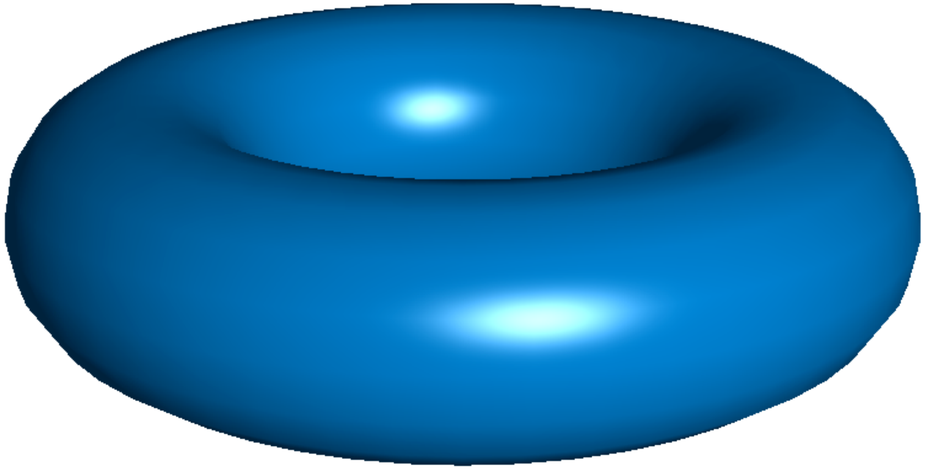


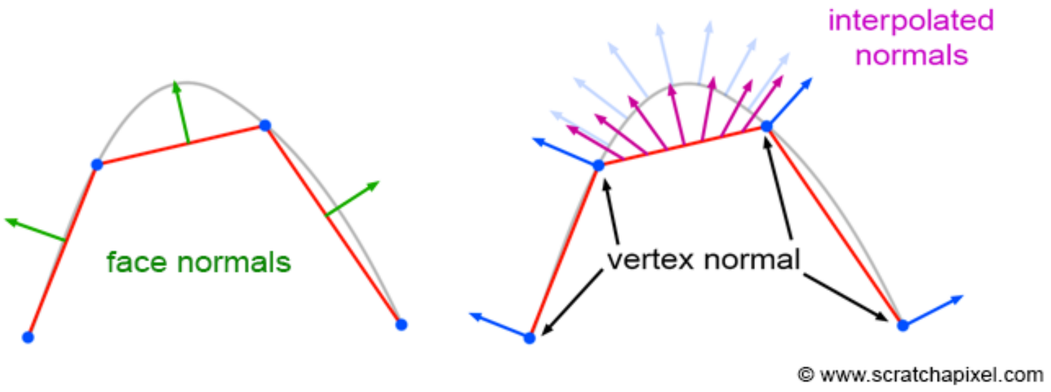


w = Area of BCP / area of ABC
u = Area of CAP / area of ABC
v = Area of ABP / area of ABC

Phong shading

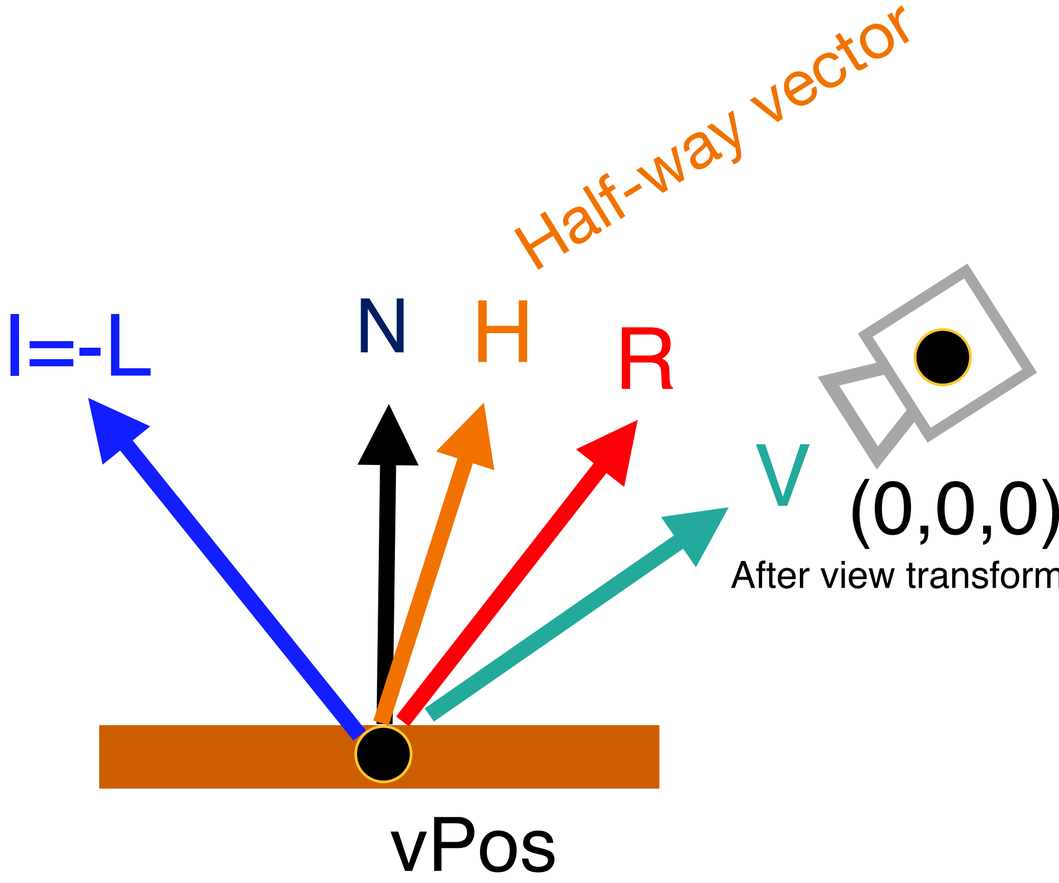
- One color per pixel.
- Interpolates vertex **normals** across triangles.
- Illumination model evaluated at each pixel.





Gouraud vertex shader anatomy

```
4     // position and normal (vertex attributes)
5     layout(location=0) in vec3 aPos;
6     layout(location=1) in vec3 aNormal;
7
8     // model, view, and projection matrices
9     uniform mat4 uModel, uView, uProj;
10
11    // light direction
12    uniform vec3 uLightDir;
13
14    // output the color of the vertex
15    out vec3 vColor;
16    void main()
17    {
18        // transform and normalize the normals
19        vec3 N = normalize(mat3(uModel) * aNormal);
20        vec3 L = normalize(uLightDir);
21
22        vec4 vPos = uView * uModel * vec4(aPos, 1.0);
23        vec3 V = normalize(-vPos.xyz);
24        vec3 R = reflect(-L, N);
25
26        float diffuse = max(dot(N, L), 0.0);
27        float specular = pow(max(dot(R, V), 0.0), 16.0);
28
29        // evaluate model at the vertex
30        vec3 color = vec3(0.2)           // amient
31            + diffuse * vec3(0.6, 0.8, 1.0)     // diffuse
32            + specular * vec3(1.0, 1.0, 1.0);   // specular
33
34        vColor = color;      // vertex color (to be interpolated
35        gl_Position = uProj * uView * uModel * vec4(aPos, 1.0);
36    }
```



$$k_{diffuse}\left(\mathbf{l_m} \bullet \mathbf{n}\right)\mathbf{L}_{m,diffuse}$$

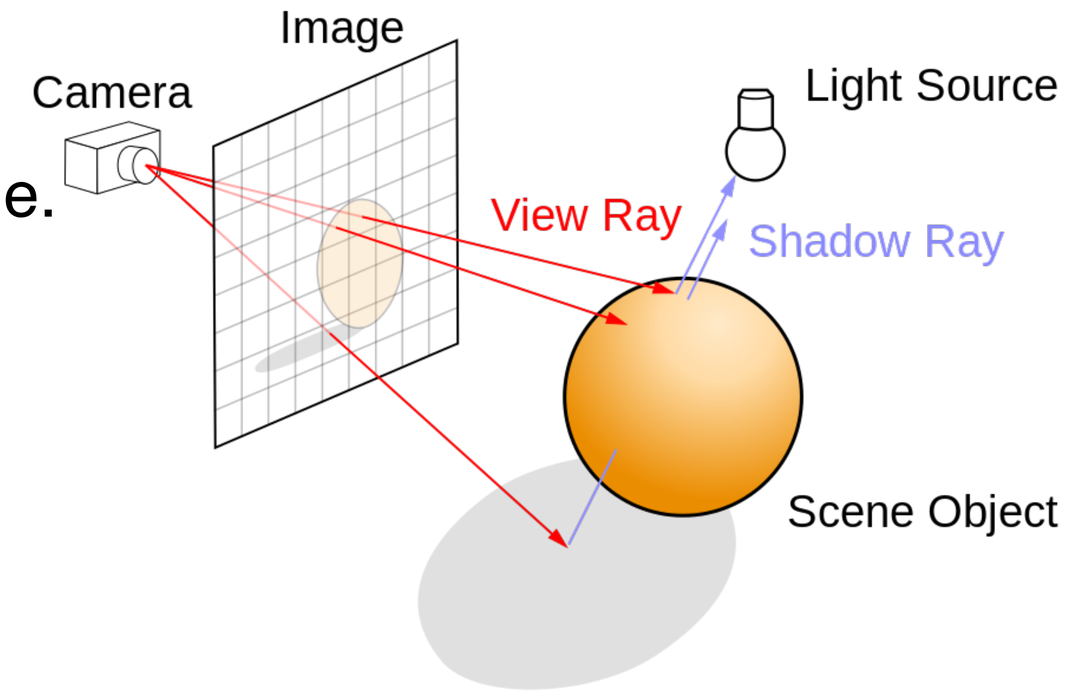
+

$$k_{specular}\left(\mathbf{r_m} \bullet \mathbf{v}\right)^{\alpha}\mathbf{L}_{m,specular}$$

Ray tracing

Basic ray tracing

1. Create ray (one per pixel).
2. Intersect ray with objects in the scene.
3. Shade (compute color of the pixel)



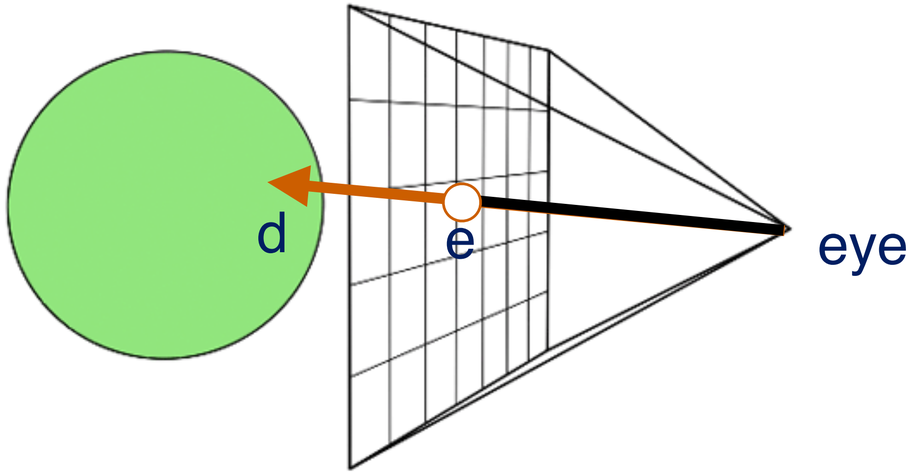
By Henrik - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3869326

Constructing a ray

- Ray equation:

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

- How to construct it?
- We have:
  - Camera position
  - Camera direction
  - Up vector
  - Viewport (width, height)
- We want:
  - Ray described by camera position and pixel center.



scratchapixel.com

Constructing a ray

- Ray equation:

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

- How to construct it?

1. Transform pixel position to camera space:

$$\mathbf{e} = \left(\left(2pixelScreen_x - 1\right) * AspectRatio * \tan\left(\frac{fov}{2}\right), \left(1 - 2pixelScreen_y\right) * \tan\left(\frac{fov}{2}\right), -1\right)$$

2. Ray direction:

$$\mathbf{d} = \left\|\mathbf{e} - \mathbf{eye}\right\|$$

Ray-plane intersection

- Ray:

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

- Plane (normal $\mathbf{n}$, point $\mathbf{p}_0$):

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{p}_0) \bullet \mathbf{n} = 0$$

- Intersection:

$$f(\mathbf{p}(t)) = 0$$

Ray-plane intersection

- Plugging ray equation into plane equation:

$$\left((\mathbf{e} + t\mathbf{d}) - \mathbf{p}_0\right) \bullet \mathbf{n} = 0$$

$$t = \frac{(\mathbf{p}_0 - \mathbf{e}) \bullet \mathbf{n}}{\mathbf{n} \bullet \mathbf{d}} \begin{cases} \mathbf{n} \bullet \mathbf{d} = 0 : \text{no intersection} \\ \mathbf{n} \bullet \mathbf{d} \neq 0 : \textit{one intersection} \end{cases}$$

Ray-sphere intersection

- Ray:

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

- Sphere (radius R and center $\mathbf{c}$):

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2$$
$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \bullet (\mathbf{p} - \mathbf{c}) - R^2 = 0$$

- Intersection:

$$f(\mathbf{p}(t)) = 0$$

Ray-sphere intersection

- Plugging ray equation into sphere equation:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \bullet (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

$$t\mathbf{d} \bullet t\mathbf{d} + 2t\mathbf{d}(\mathbf{e} - \mathbf{c}) + (\mathbf{e} - \mathbf{c}) \bullet (e - \mathbf{c}) - R^2 = 0$$

$$(\mathbf{d} \bullet \mathbf{d})t^2 + 2\mathbf{d} \bullet (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \bullet (\mathbf{e} - \mathbf{c}) - R^2 = 0$$

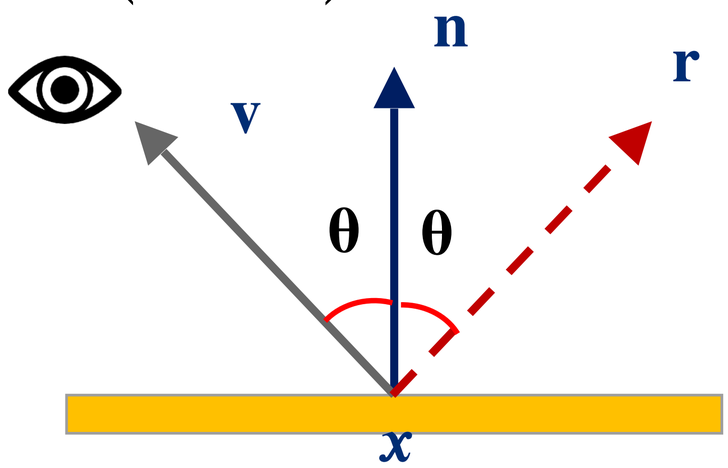
$$At^2 + Bt + C = 0$$

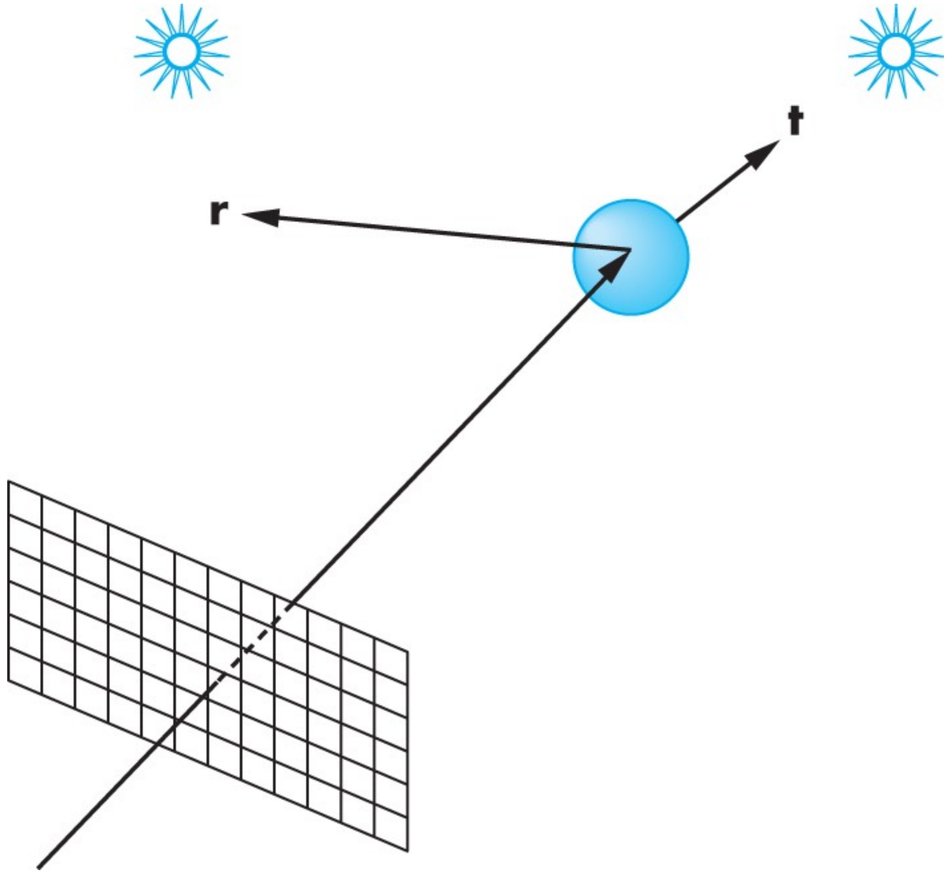
$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \begin{cases} B^2 - 4AC < 0 : no\ intersection \\ B^2 - 4AC = 0 : one\ intersection \\ B^2 - 4AC > 0 : two\ intersections \end{cases}$$

Reflections

- Reflected rays will follow the law of reflection:

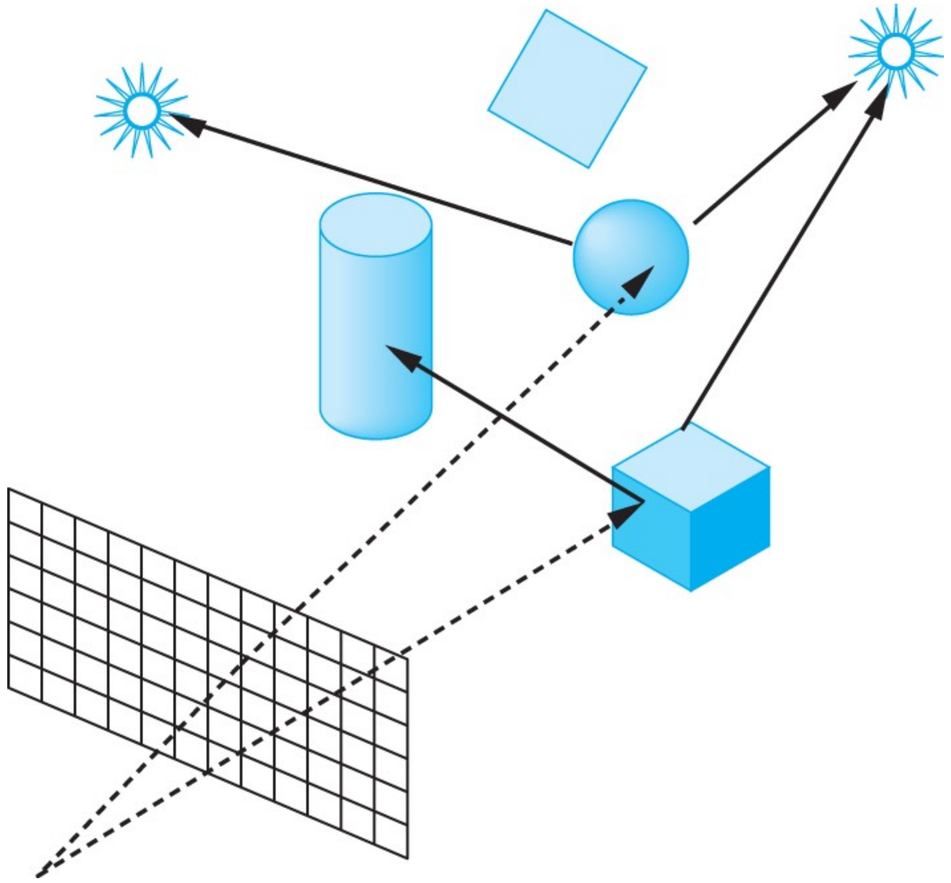
$$\hat{\mathbf{r}} = 2\left(\hat{\mathbf{v}} \bullet \hat{\mathbf{n}}\right)\hat{\mathbf{n}} - \hat{\mathbf{v}}$$





Shadows

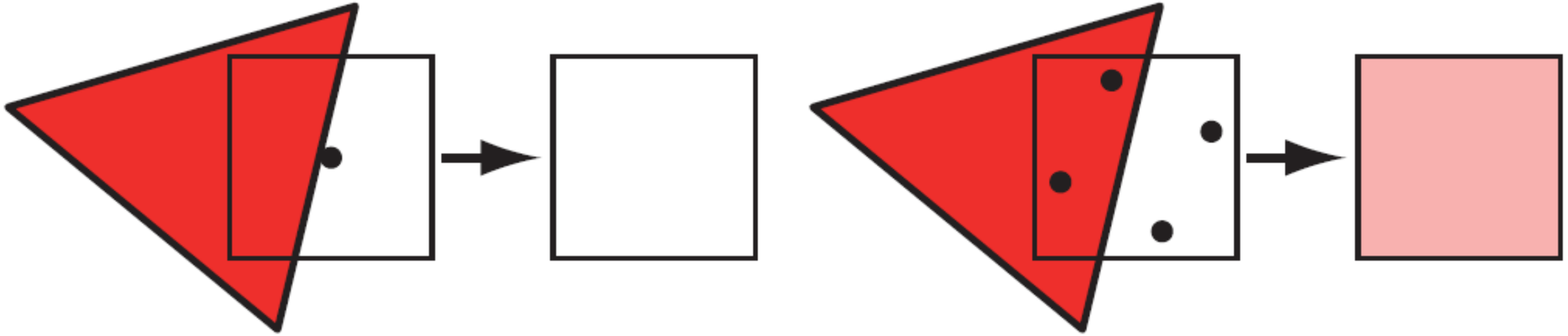
- To check if a point is under shadow or not, cast a ray from each point to the light:
  - If it intersects something before reaching the light, it is in a shadow area.



Antialiasing
&
spatial data structures

Screen-based antialiasing

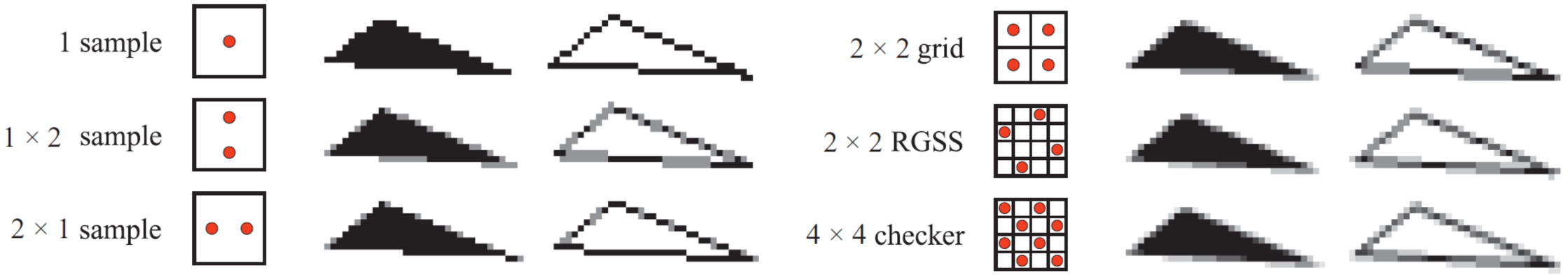
- Operate on the output samples of the graphics pipeline, without any knowledge of the objects being rendered.



$$\mathbf{color}(x, y) = \sum_{i=1}^{n} w_i \mathbf{c}(i, x, y)$$

Supersampling

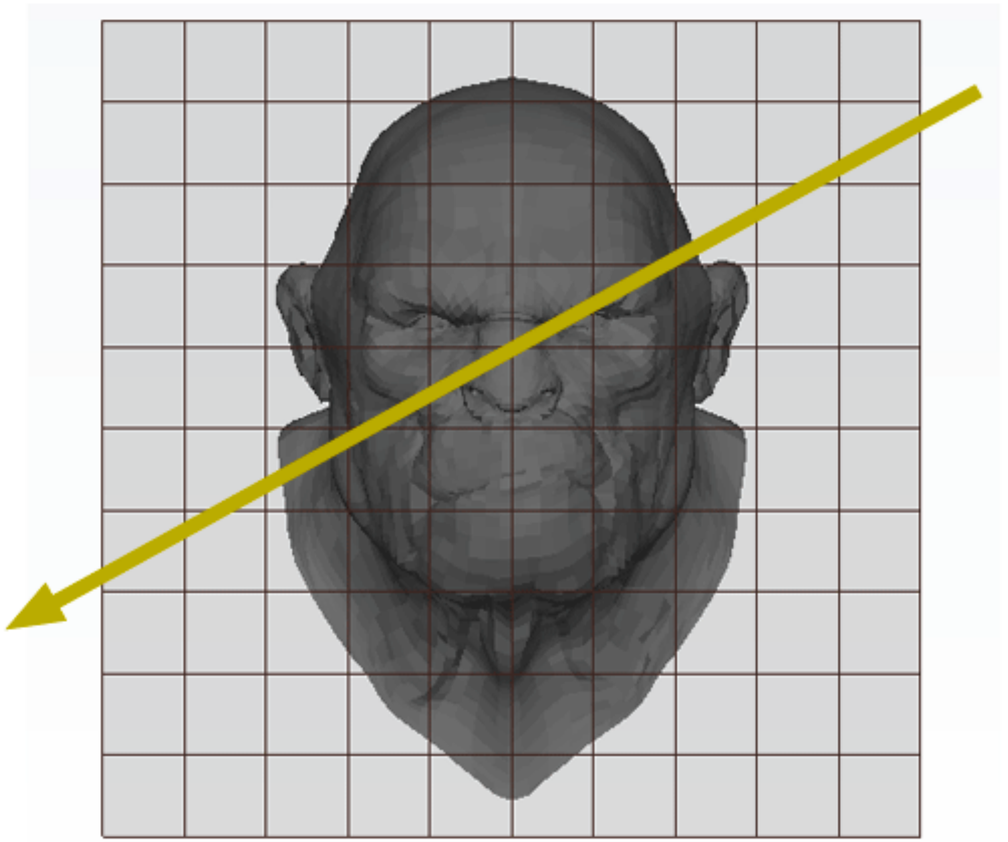
- Spatial antialiasing method.
- Renders the scene at higher resolution, then average neighboring samples.



Real-Time Rendering, 4th Ed.

Uniform grid

- Partition space into equal-sized volumes (i.e., voxels).
- Each cell will contain objects that overlap the voxel.
- Good for uniform data (points are evenly distributed in space).
- Fast construction and queries.



scratchpixel.com

Uniform grid