Final Exam Review

Introduction to Computer Graphics

Yu-Ting Wu

About Final Exam

- The final exam will be held physically on **Dec. 26th**, 2022
  - 09:10 – 12:00
  - Room 103
  - 35% of your final grade

- **Scope (theoretical part only)**
  - Scene representation
  - Camera
  - Lighting and shading
  - Textures
  - Transparency
  - GPU graphics pipeline
  - Deferred shading
  - Shadow map
  - Ambient occlusion
  - Ray tracing

About the 17th and 18th Weeks

- There is **NO** class in the 17th week (Jan. 2nd, 2023)
- There is a **physical class** (and recording) in the 18th week
  - Advanced shaders in OpenGL 3.0 and 4.0
  - A case study of the rendering system in Unity
  - You can take back your answering sheet for the final exam

About Homework

- The grade of your HW2 has been announced
  - If you have questions, feel free to mail me
- **The final deadline for all homework is Jan. 8th, 2023**
  - **You can resubmit previous HW if you think it is worth it (but let me know by mail)**

Computer Graphics Overview

- **Digitally synthesize** and **manipulate** a virtual world

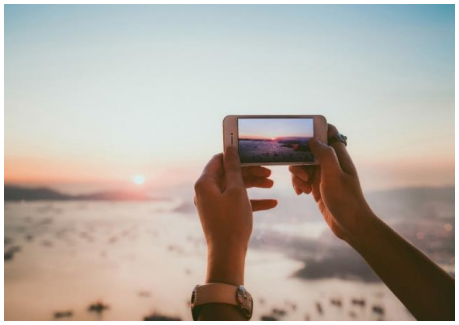
Computer Graphics Overview (cont.)

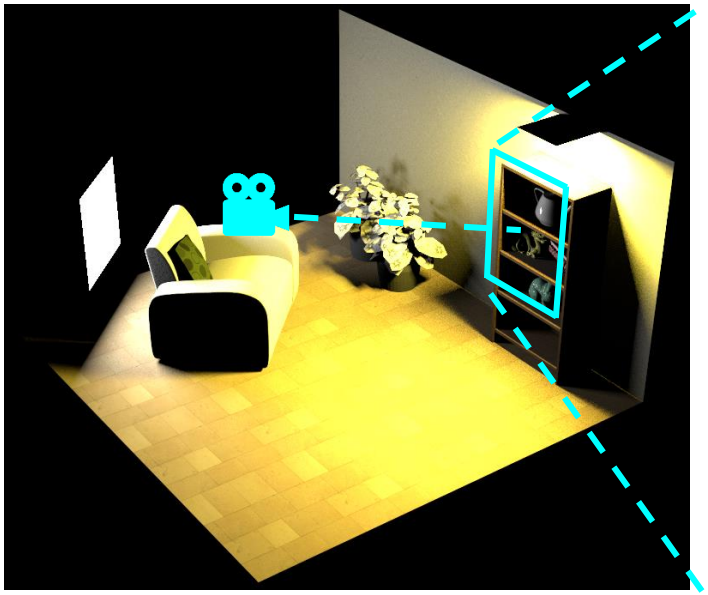
- **Digitally synthesize** and **manipulate** a virtual world
  - Model **geometry** of the 3D objects using triangle meshes
  - Model **materials** of the 3D objects (*Phong* lighting model) and simulate **lighting** (point/directional/spot lights)
  - Simulate more realistic materials (microfacet models and transparency) and lighting phenomena (shadow and ambient occlusion)
  - Simulate more complex light paths (global illumination)




Digital Image Synthesis

- Most displays are 2D, so we need to generate images from the 3D world
- Just like taking a picture with a camera in our daily lives
  - But with a **virtual camera** and a **virtual film**





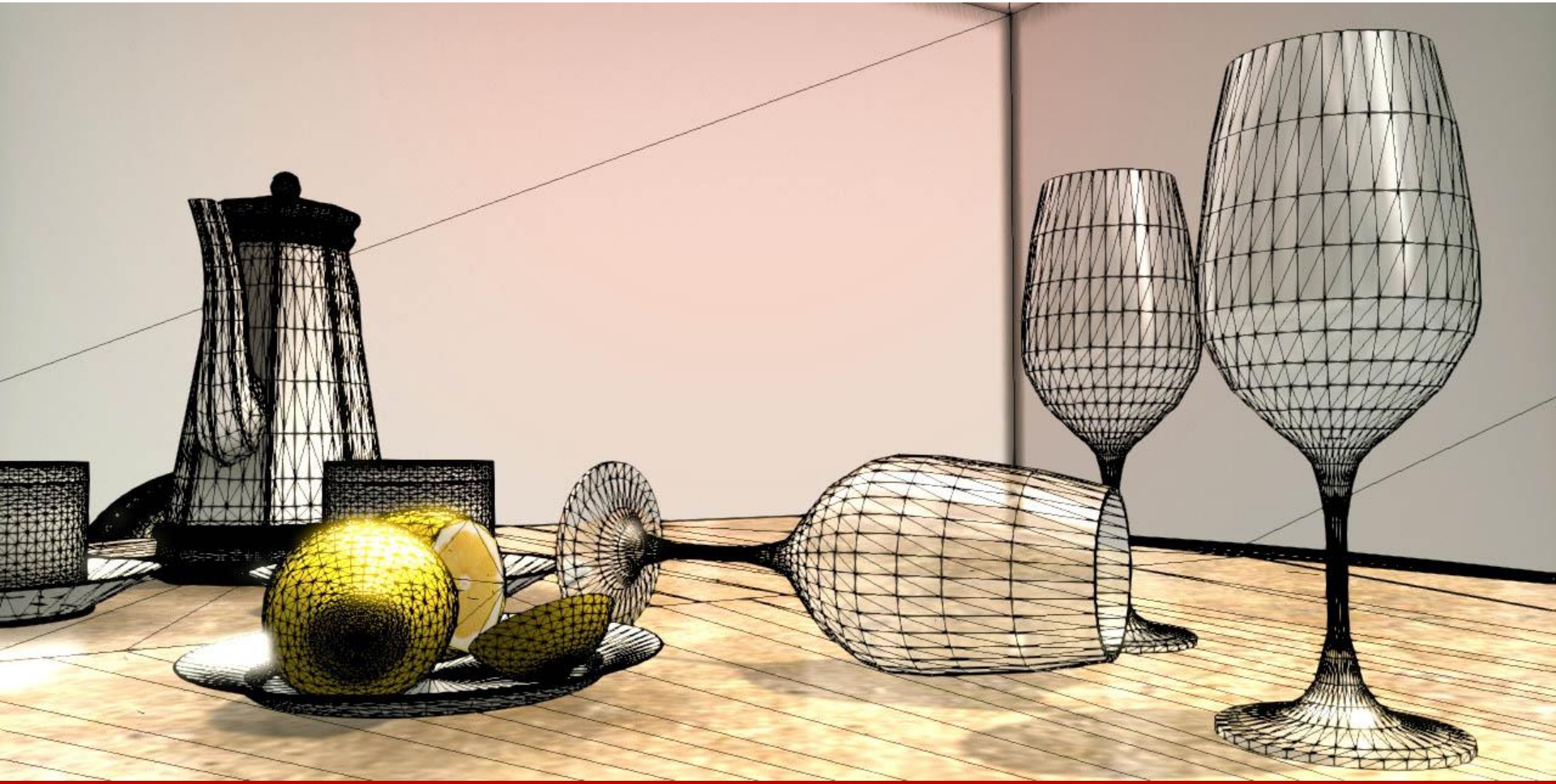
3D virtual world

rendered image

Scene Representation

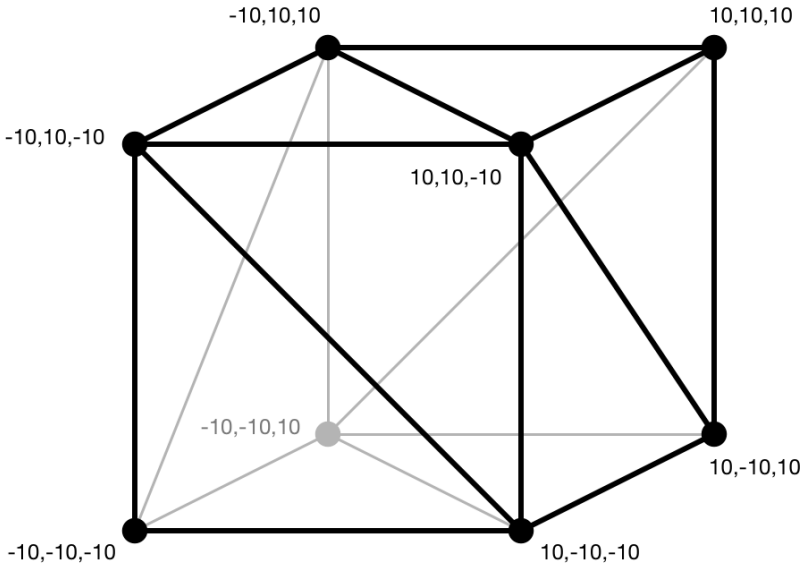
Triangle Mesh

- Use triangle meshes to model the geometry of 3D objects

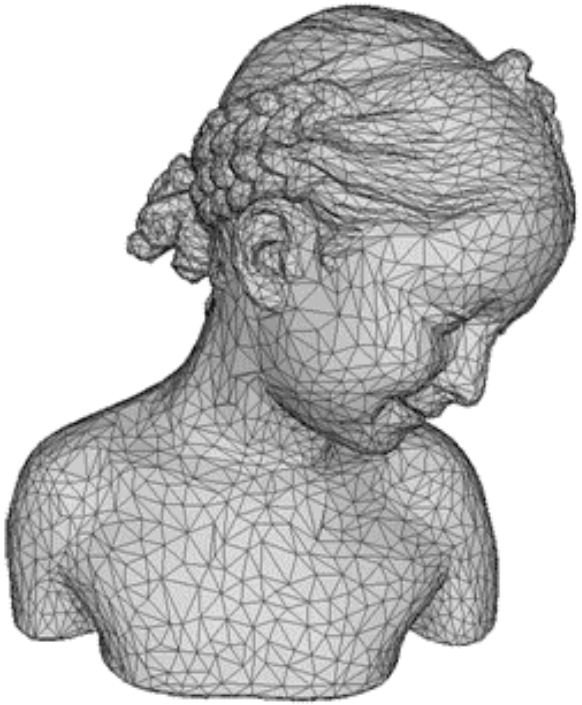


Triangle Mesh (cont.)

- Vertex attributes: 3D position, normal, texture coordinate
- Vertex adjacencies: which vertices form a triangle



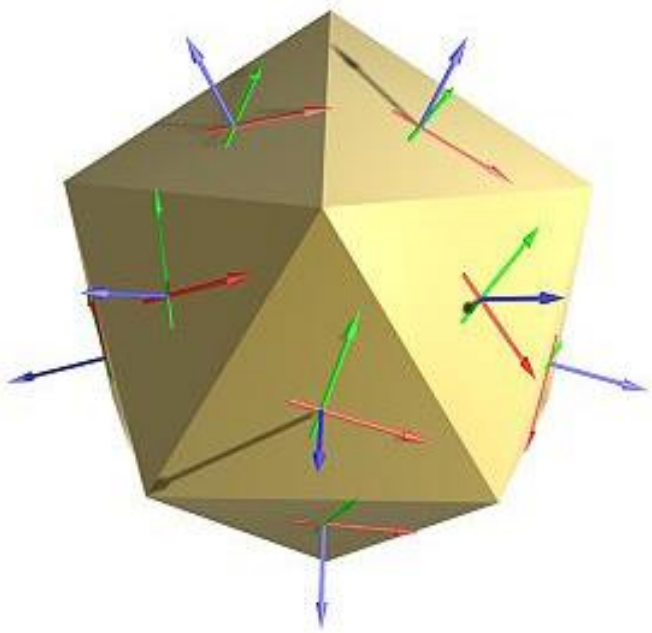
12 triangles



10K triangles

Triangle Mesh (cont.)

- A **surface normal** is a vector that is **perpendicular** to a surface at a particular position
- Represent the orientation of the face
- The length of a normal should be equal to **1**



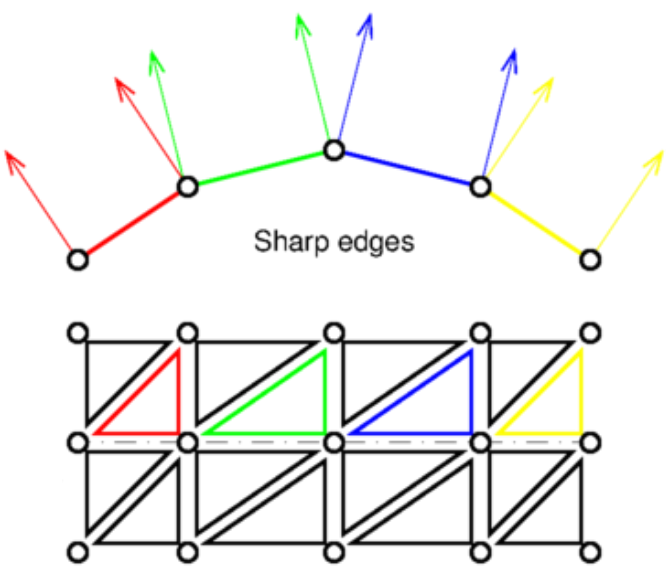
→ **normal (n_x, n_y, n_z)**

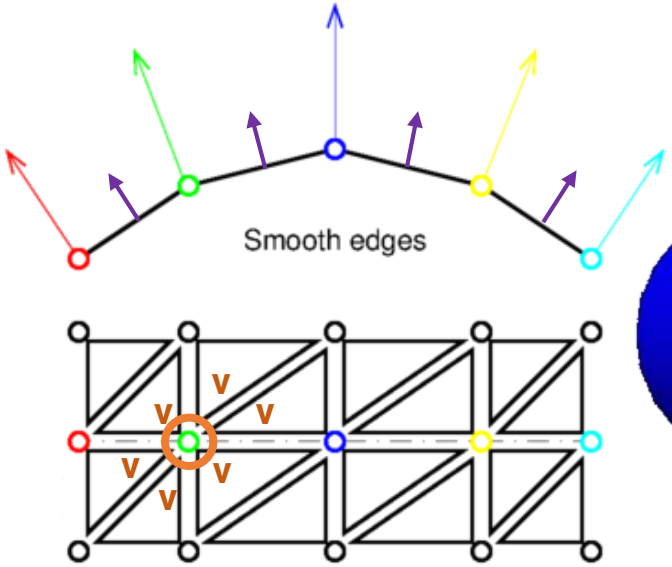
→ tangent

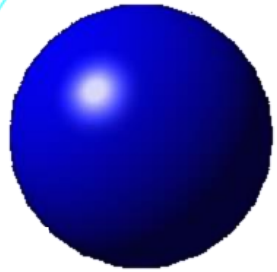
→ binormal

Triangle Mesh (cont.)

- **Vertex normal**
  - Computed by **averaging** the surface normals of the faces that contain that vertex
  - Can achieve more smooth shading

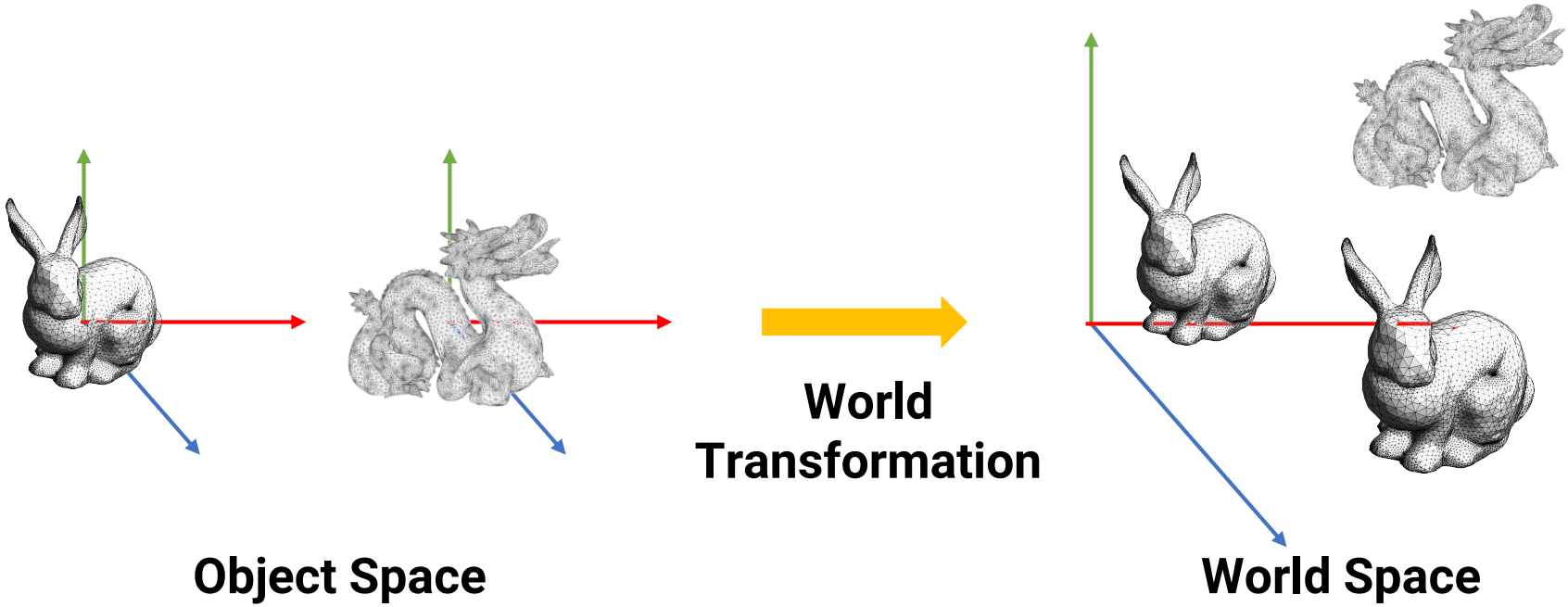






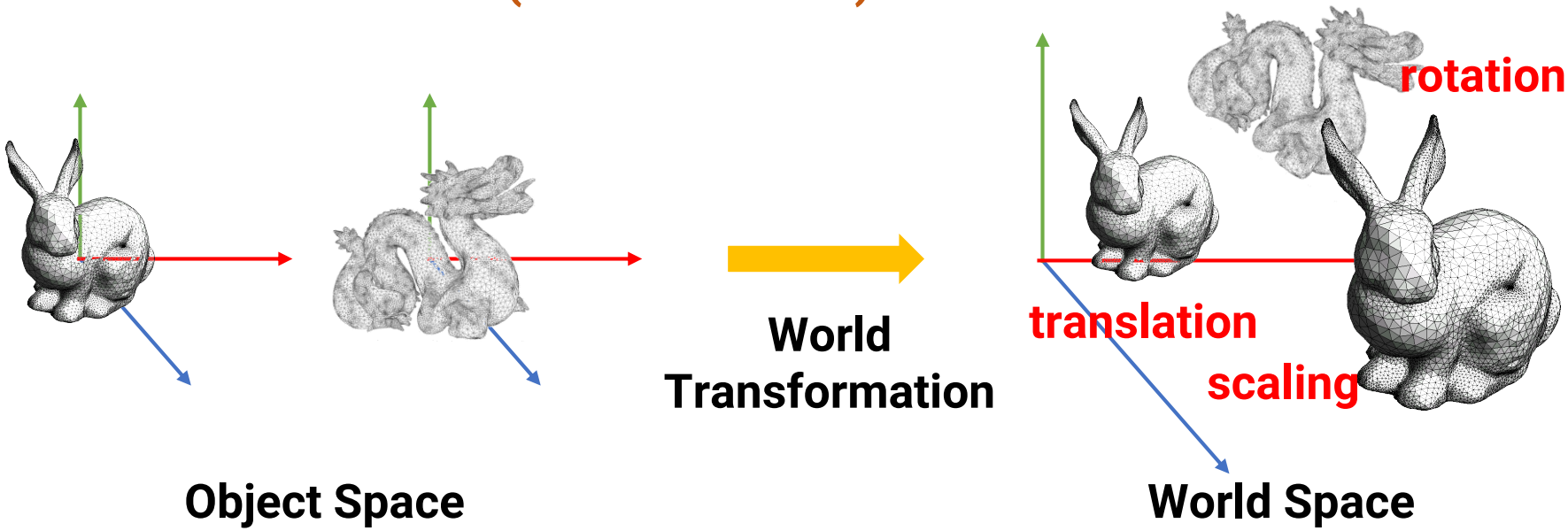
Object Space and World Space

- 3D models are usually defined in **Object Space**
- When building a scene, each object is transformed into a **global** and **unique** space called **World Space**
- The transform is called **World Transformation**



Object Space and World Space (cont.)

- Common 3D transforms for **World Transformation**
  - Translation
  - Scaling
  - Rotation
  - **Combination (concatenation) of transforms**



Common 3D Transforms

- To allow for concatenation, a transform is represented by a **4x4 matrix**

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translation

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scaling

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation w.r.t x-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation w.r.t y-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation w.r.t z-axis

Common 3D Transforms

- Concatenation of transforms

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation w.r.t y-axis | scaling | translation

Homogeneous Coordinate

- To apply a transform, we need to use **homogeneous coordinate**
- For a point

$$(x, y, z) \rightarrow (x, y, z, w = 1)$$

- For a vector (normal)

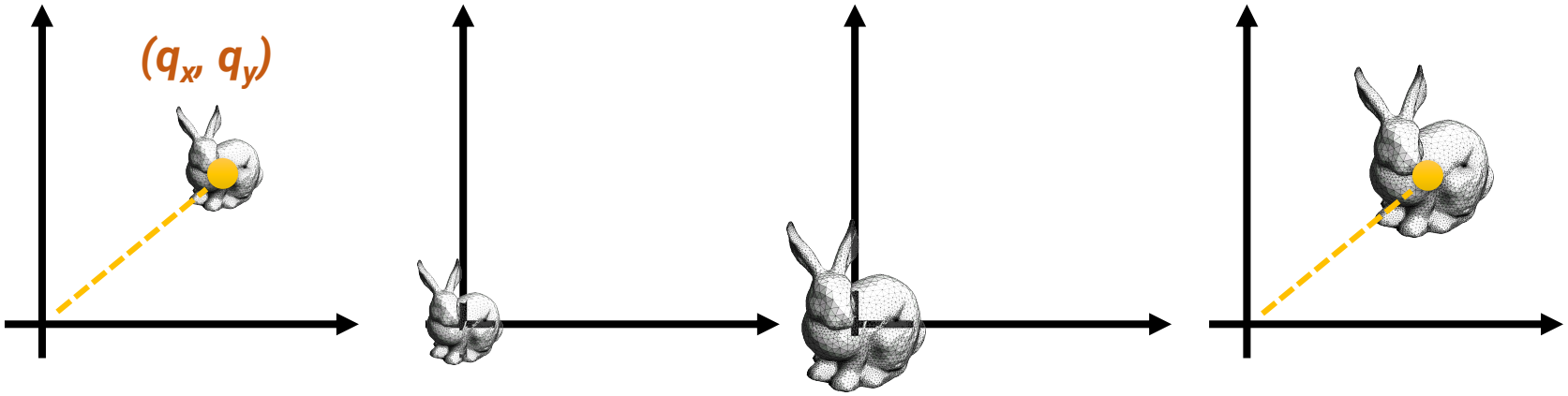
$$(x, y, z) \rightarrow (x, y, z, w = 0)$$ **to avoid translation**

- After transformation, the *xyz* components are **divided by the *w* component** for a transformed point, or **normalized** for a transformed vector (normal)

Issue of Scaling Transform

- If the object is not centered at the origin, it will get a shift
- To scale about an **arbitrary pivot point $Q(q_x,q_y,q_z)$**
  - Translate the objects so that Q will coincide with the origin: $T(-q_x,-q_y,-q_z)$
  - Scale the object: $S(s_x,s_y,s_z)$
  - Translate the object back: $T(q_x,q_y,q_z)$
  - The final scaling matrix can be written as $T(q)S(s)T(-q)$

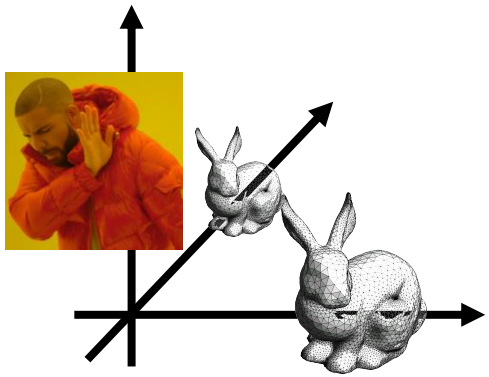
Concatenation of matrices



Issues of Rotation Transform

- The standard rotation matrix is used to rotate w.r.t the x, y, and z axis
- **To rotate locally**
  - Translate the objects so that its center Q will coincide with the origin: $T(-q_x,-q_y,-q_y)$
  - Rotate the object: $R(\Theta)$
  - Translate the object back: $T(q_x,q_y,q_z)$
  - The final rotation matrix can be written as $T(q)R(\theta)T(-q)$

Concatenation of matrices



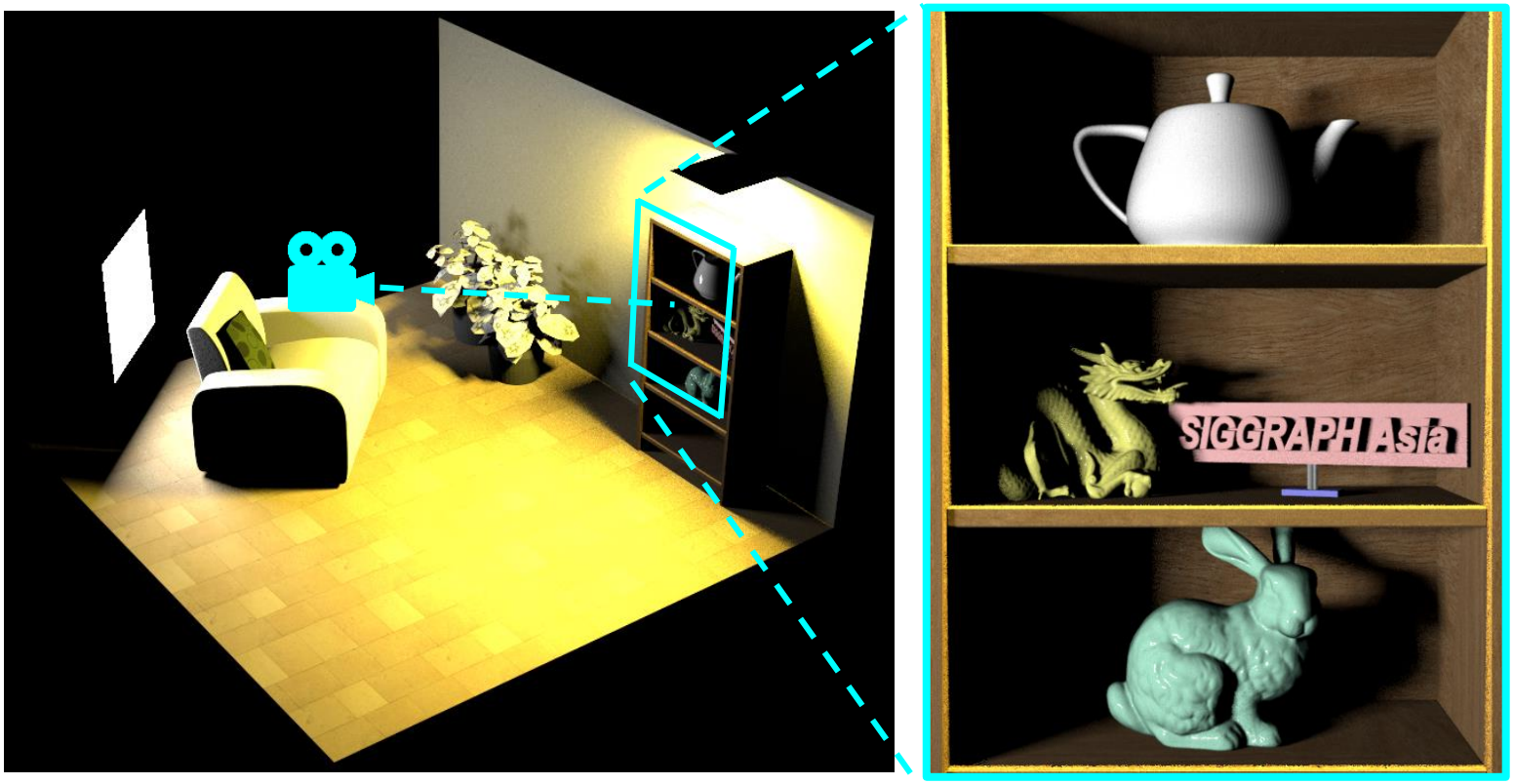
Why use World Transformation?

- **Reuse model:** design a model and use it in several scenes
- **Memory saving:** store a 4x4 matrix instead of duplication of the entire models

Rendering

Recap: Digital Image Synthesis

- Generate an image from the 3D world with a **virtual camera**



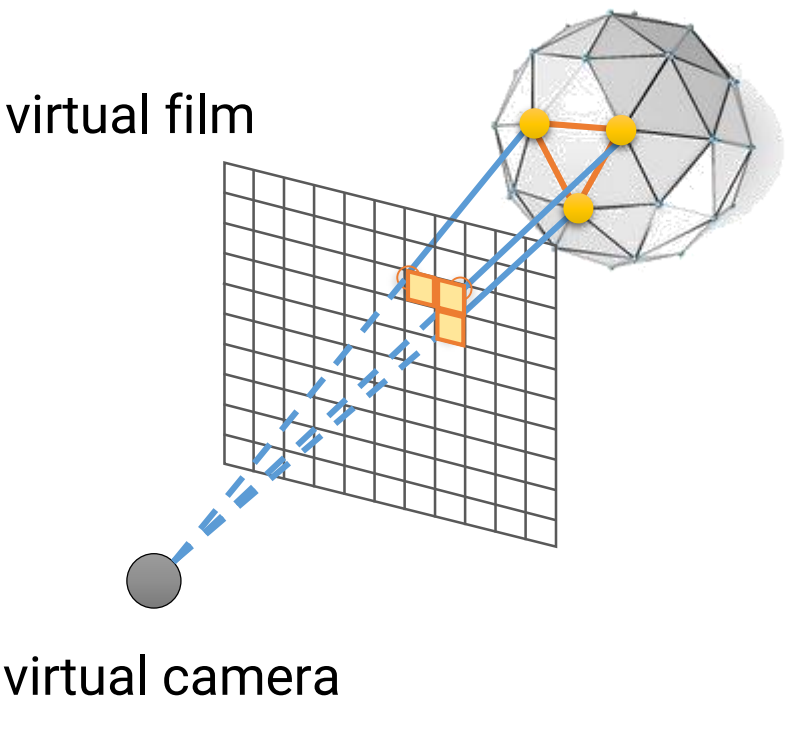
3D virtual world

rendered image

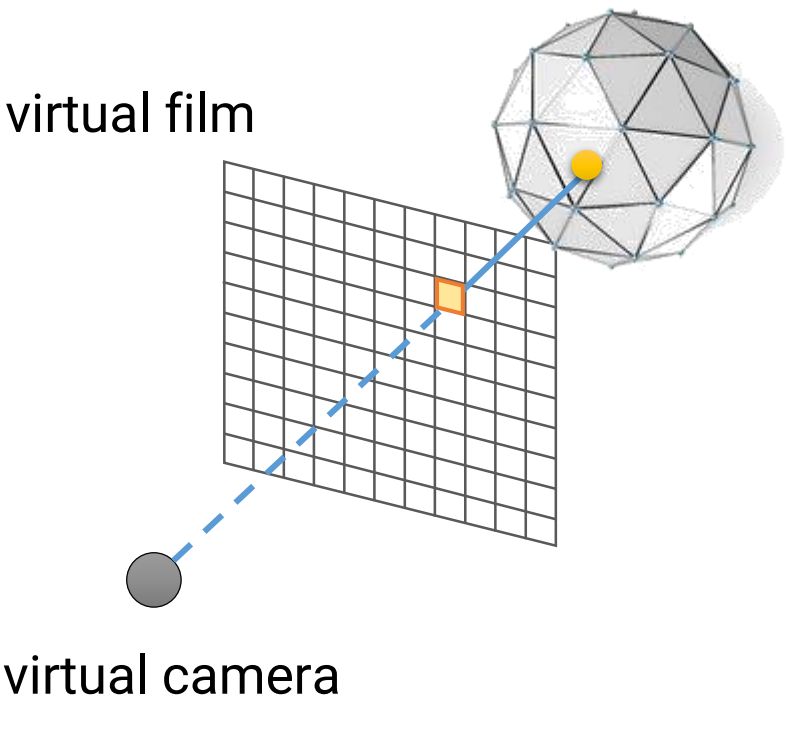
Rasterization and Ray Tracing

- Two ways for generating synthetic images

Rasterization



Ray tracing

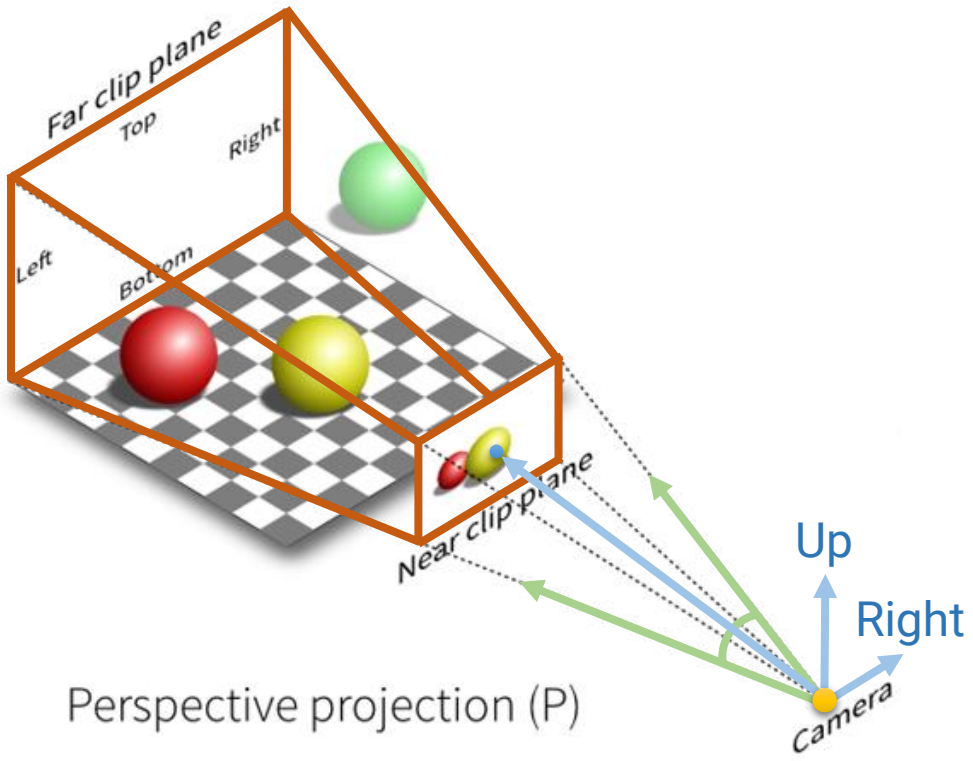


Camera Properties

- The film is **in front of** the camera (to avoid up-side-down)
- **Basic properties**
  - Camera position
  - Viewing direction
  - Camera local frame
  - Field of view
  - Aspect ratio

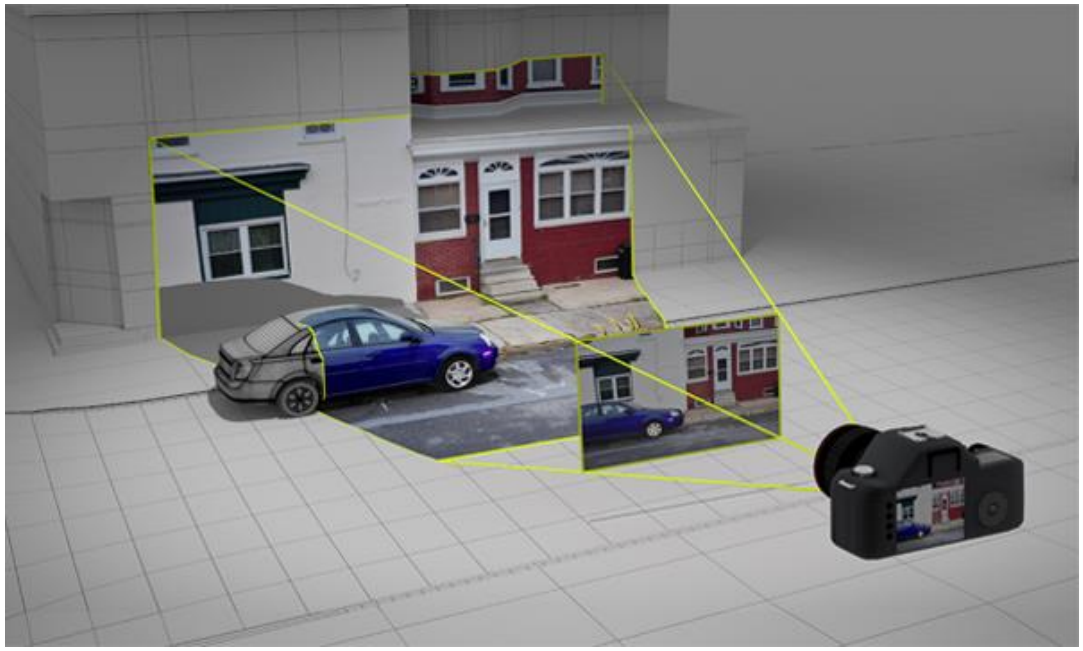
viewing volume (view frustum)

- **Advanced properties**
  - Shutter speed
  - Lens system



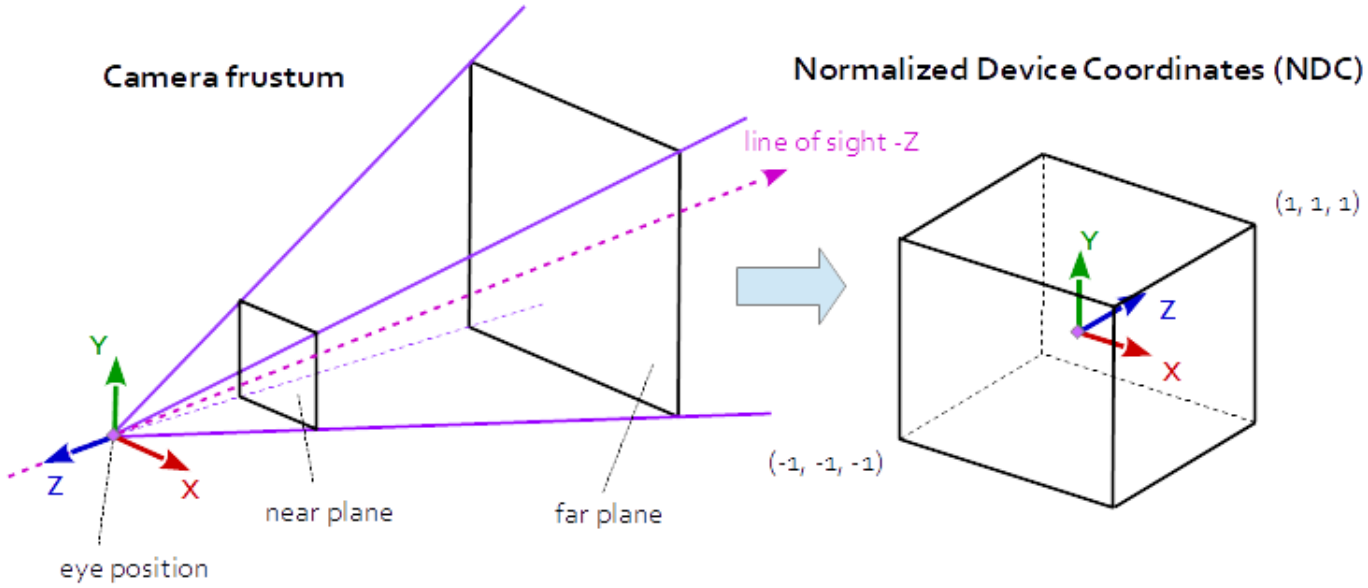
Camera (View) Space

- The camera can be at an arbitrary position and have an arbitrary viewing direction in **World Space**
- This makes the **projection** or **ray generation** difficult in terms of mathematics



Camera (View) Space (cont.)

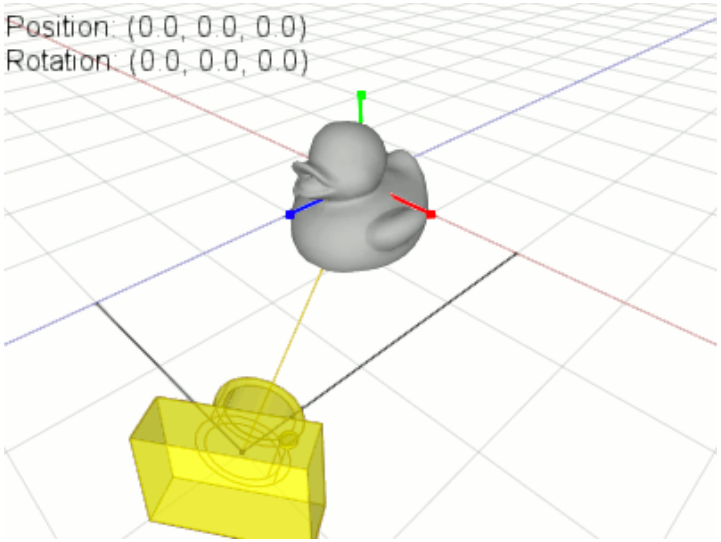
- To keep the math of projection or ray generation simpler, we additionally define a **camera (view, eye) space**
  - In the camera space, the camera is **at the origin *(0, 0, 0)*** and **looking at the negative Z-axis**

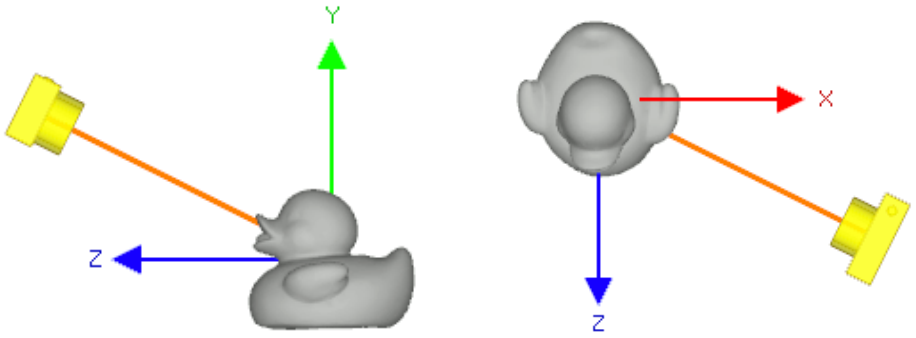


- Vertices are then projected **from Camera Space to NDC**

Camera (View) Transformation

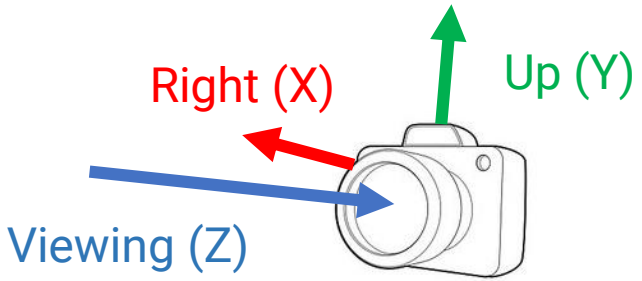
- An object (with all its vertices) is transformed from **World Space** to **Camera Space** by applying **Camera Transformation (4x4 matrix)**
- The camera matrix is a combination of **translation** and **rotation** matrix



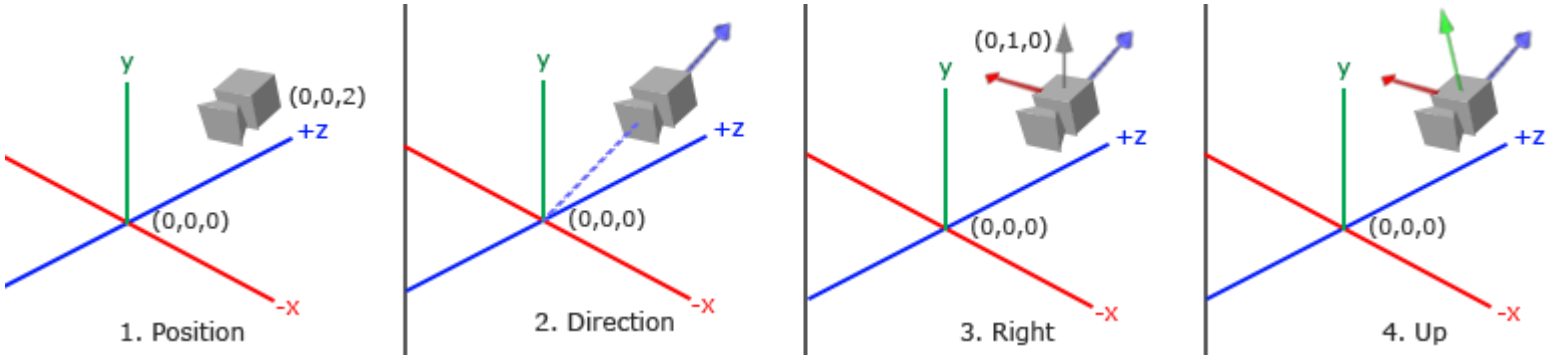


Camera (View) Transformation

- To do this, we need to define the camera's local frame
  - Viewing direction (Z)
  - Right vector (X)
  - Up vector (Y)

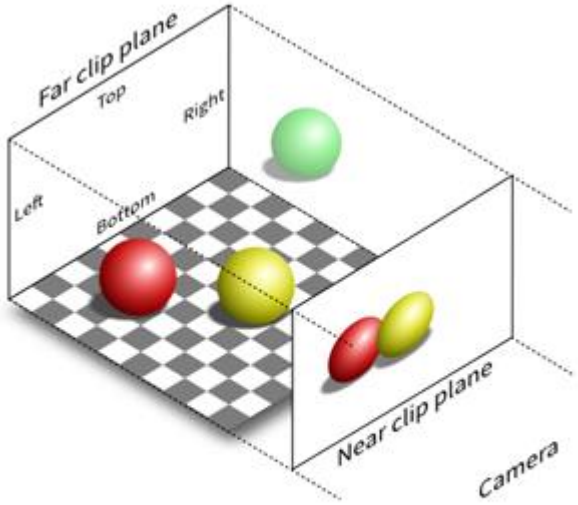


- Process
  - Move it with the inverse translation of the camera's position
  - Rotate the object to match the camera's local frame

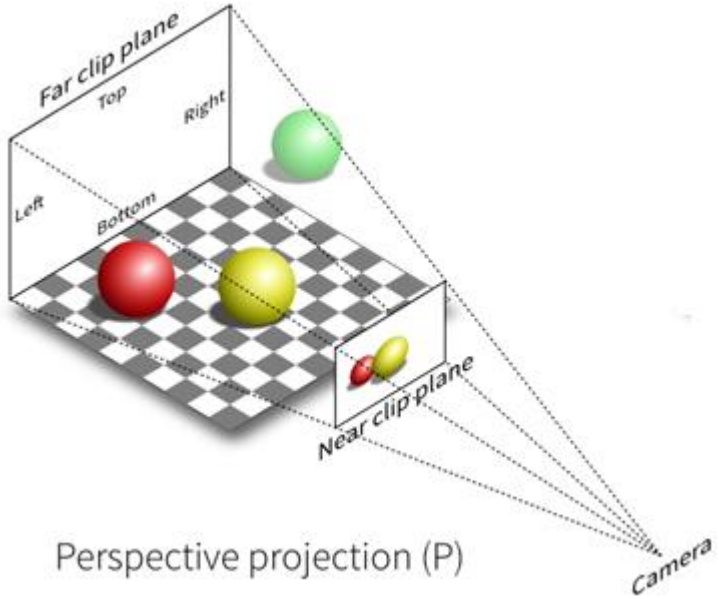


Camera Projection

- Two common projections in Rasterization
  - **Orthographic:** parallel projection with projectors perpendicular to the projection plane
  - **Perspective:** mimic the vision of human eyes; the objects that are farther away appear much smaller



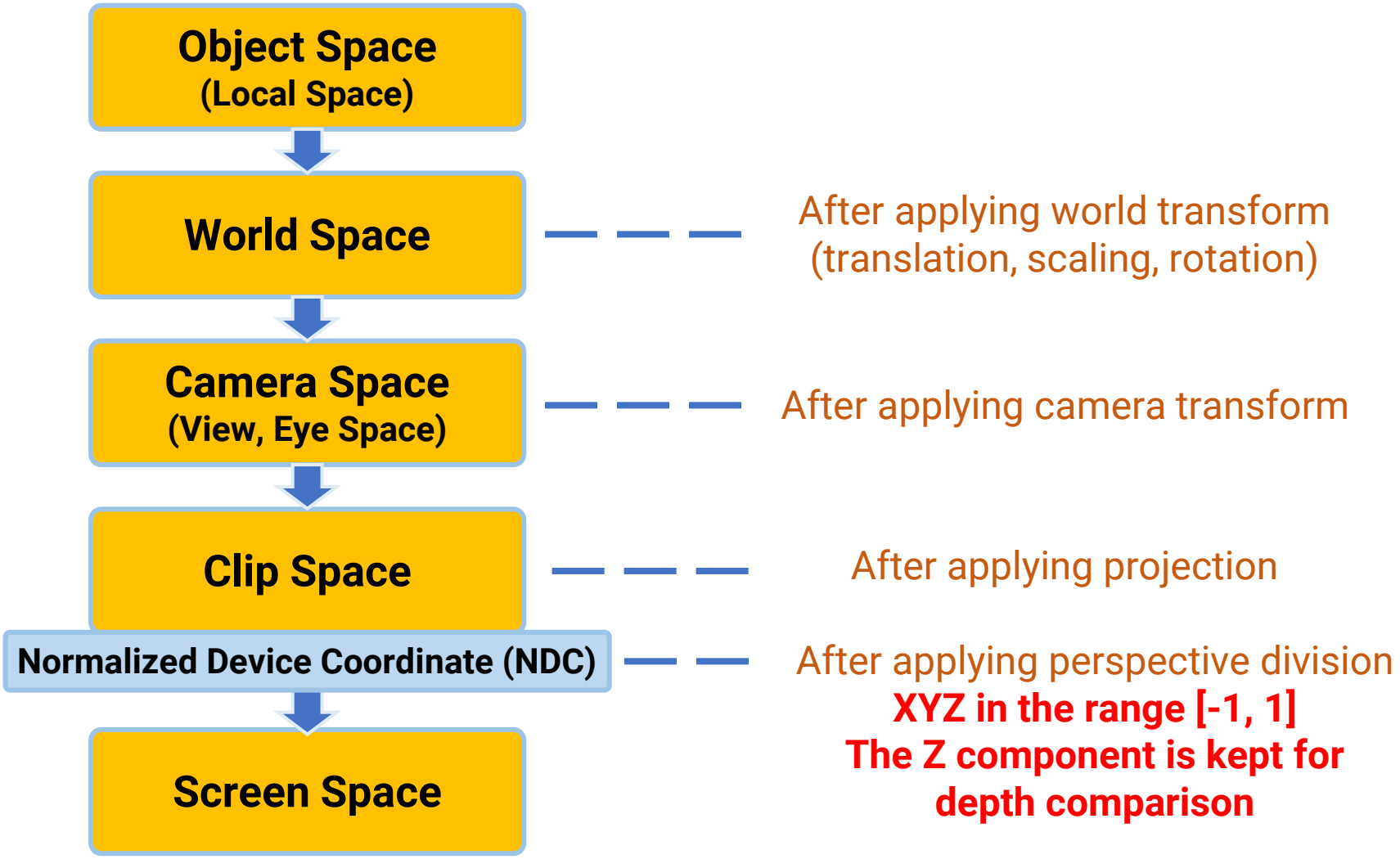
Orthographic projection (O)



Perspective projection (P)

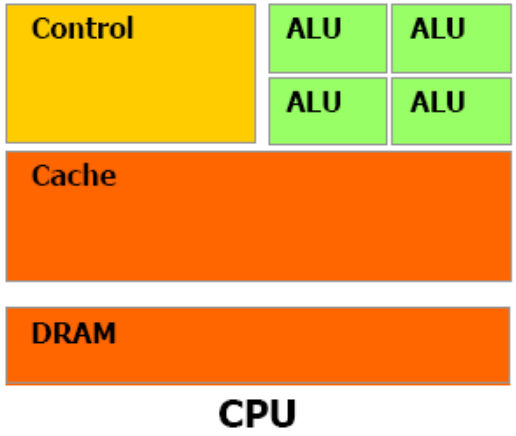
GPU Rendering Pipeline

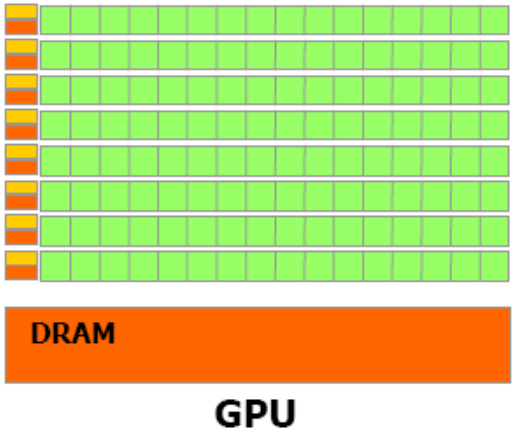
Full Vertex Transformation



Running on GPU! (Rasterization)

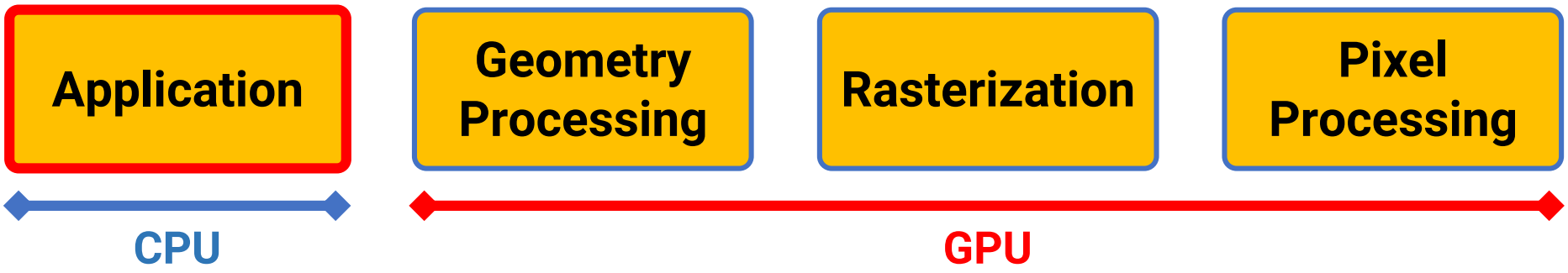
- In complex scenes there are massive vertices, each requires
  - Vertex transformation
  - Shading
- Both tasks are relatively simple
- Better tailored to graphics hardware (GPU)





GPU Rendering Pipeline

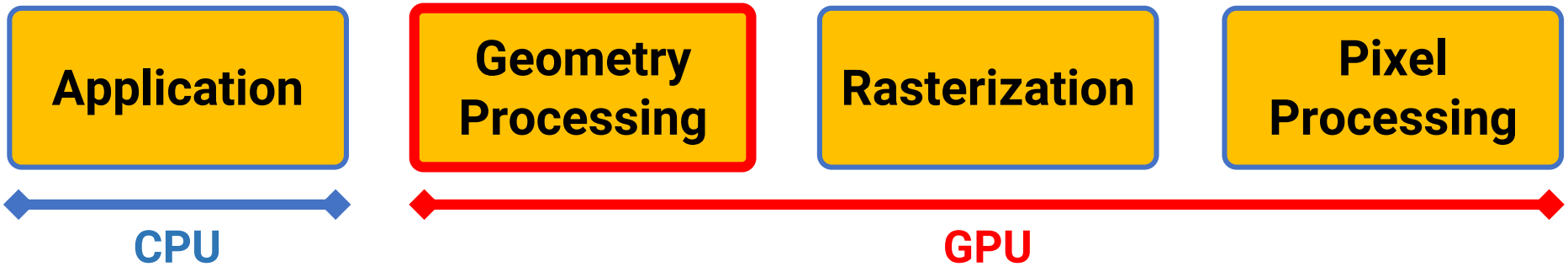
- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages



- Physical simulation
- Animation
- Collision detection
- Global acceleration
- etc.

GPU Graphics Pipeline Overview (cont.)

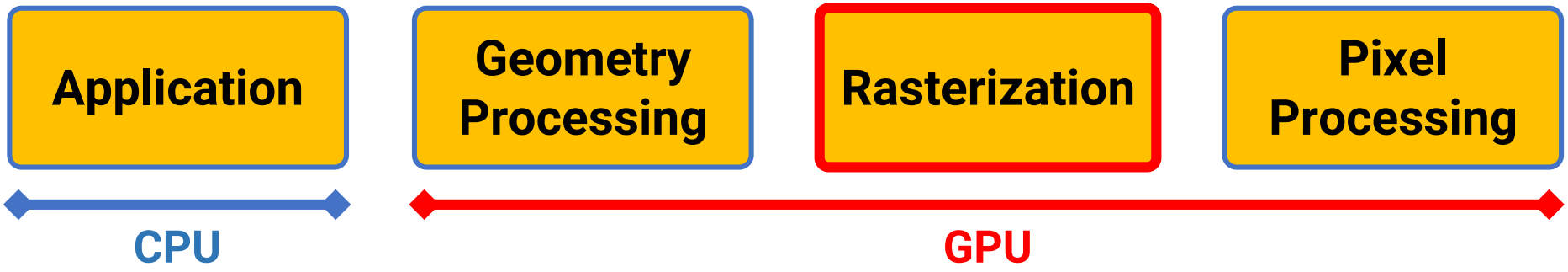
- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages



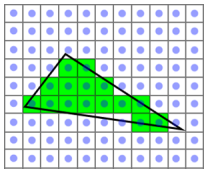
- Vertex transform and projection
- Vertex lighting and shading (rarely used now)
- Geometry assembly
- Clipping
- Culling

GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages



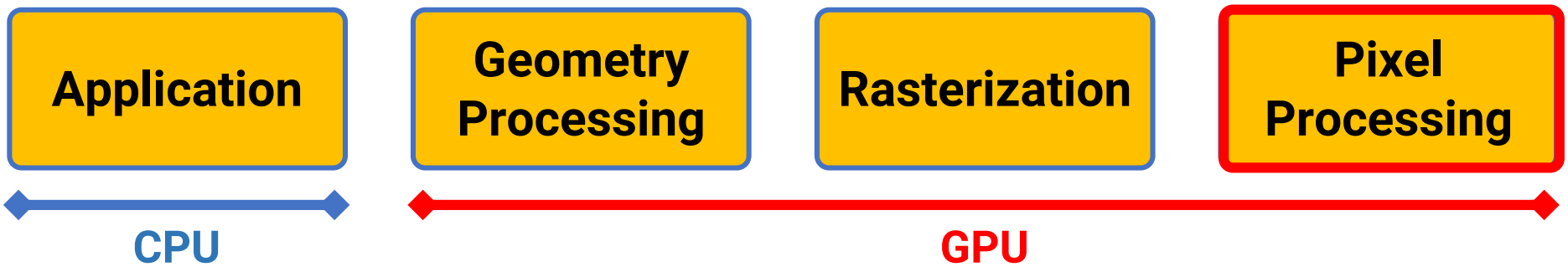
- Triangle setup
- Fragments (pixels) generation
  - Attribute interpolation



from continuous to discrete

GPU Graphics Pipeline Overview (cont.)

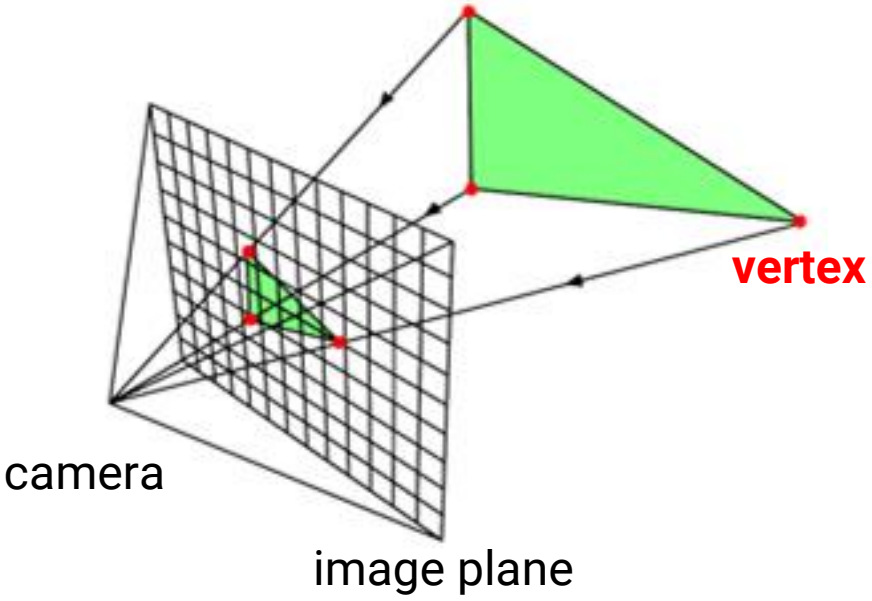
- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages

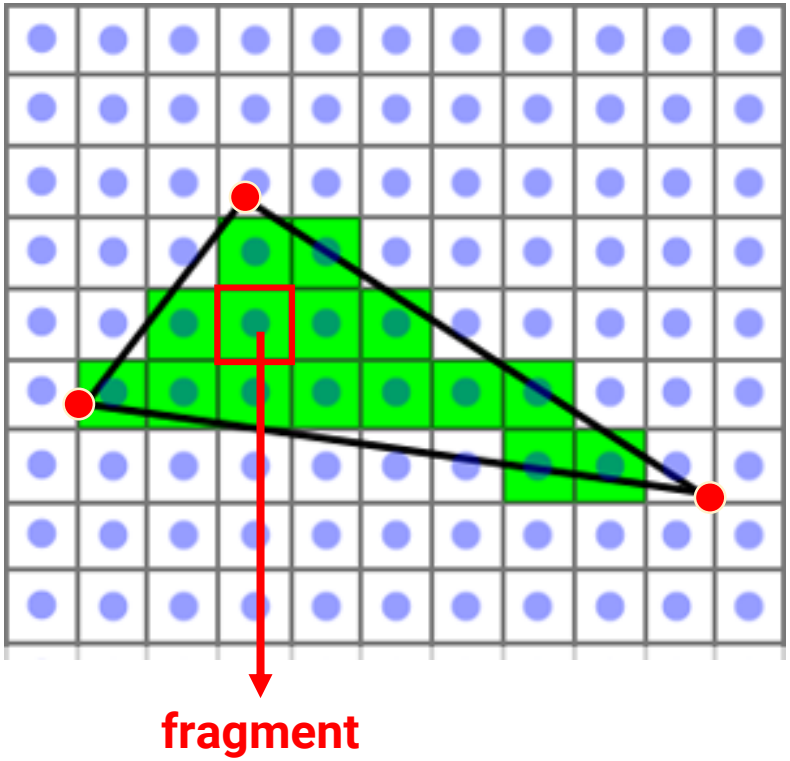


- Pixel shading / Texturing
- Depth testing
- Alpha blending

Rasterization

- Convert **triangles** (**continuous**) into **fragments** (**discrete**, which eventually become the individual **screen pixels**)
- Vertex attributes are **interpolated** across the face





Rasterization (cont.)

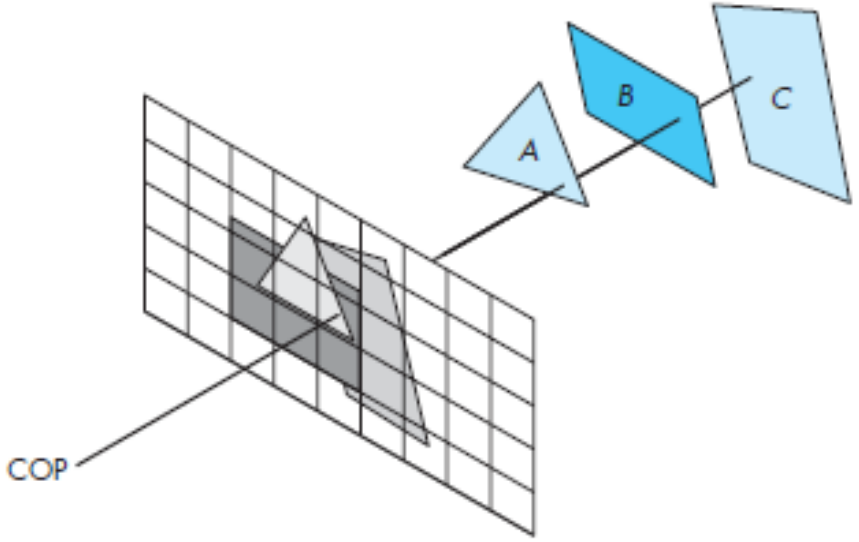
- Convert **triangles** (**continuous**) into **fragments** (**discrete**, which eventually become the individual **screen pixels**)
- Vertex attributes are **interpolated** across the face, including
  - (Lighting) color
  - Texture coordinate
  - Position
  - Normal
  - Anything you want to interpolate

if per-vertex lighting is used

for per-fragment lighting (after OpenGL 2.0)

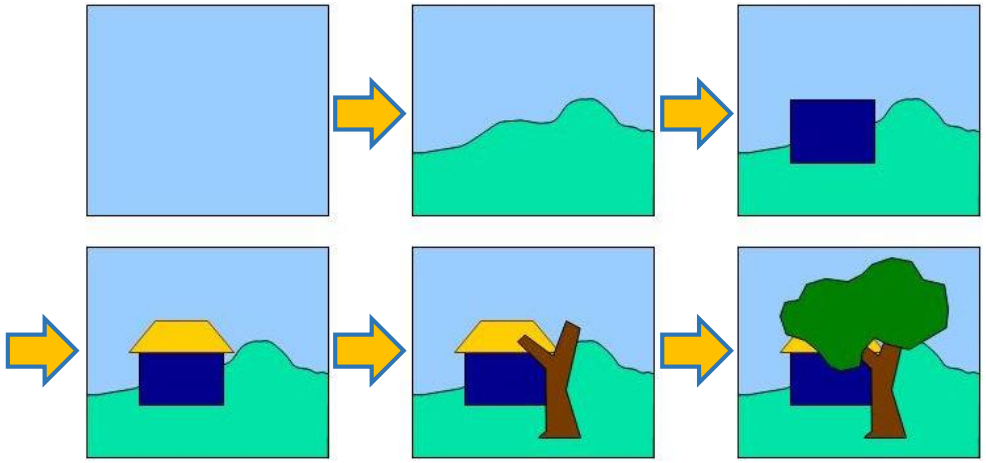
Depth Comparison

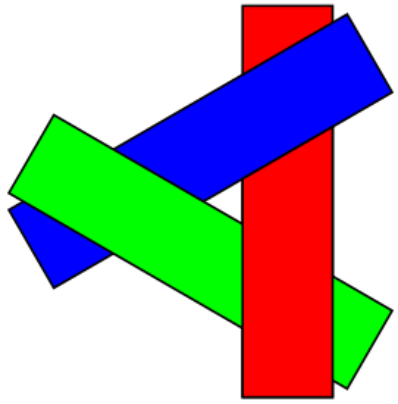
- Several surfaces can project to the same screen location
  - Only the **closest** surfaces from the camera can be seen at each pixel
- How to hide occluded surfaces?
  - Painter's algorithm
  - Z-buffer



Depth Comparison (cont.)

- **Painter's algorithm**
  - Draw objects in an order based on their distances from the camera
  - The farthest object first

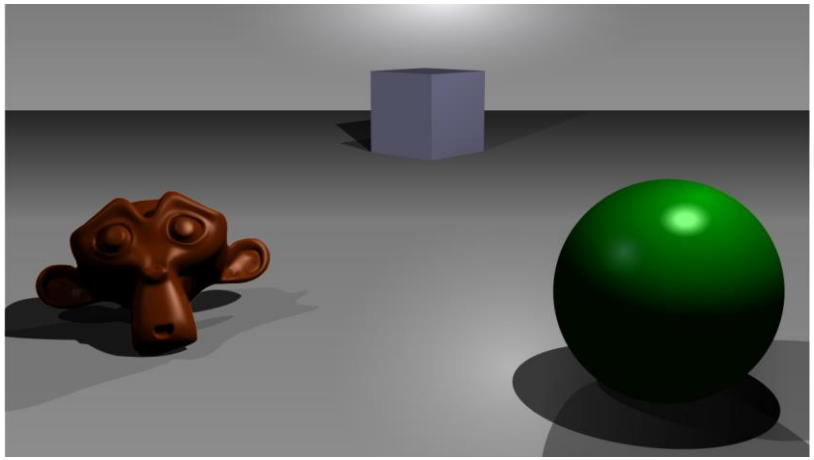




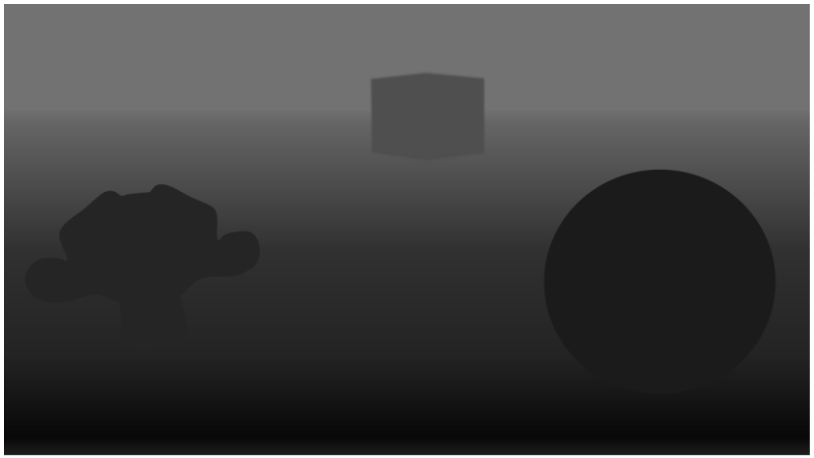
How about this?
(cyclical overlapping)

Depth Comparison (cont.)

- **Z-buffer (current solution)**
  - An additional buffer used to maintain the *z* value of the closest surface to a pixel
  - **Discard** fragments if they have larger depth values than the ones stored in their corresponding positions in the Z buffer



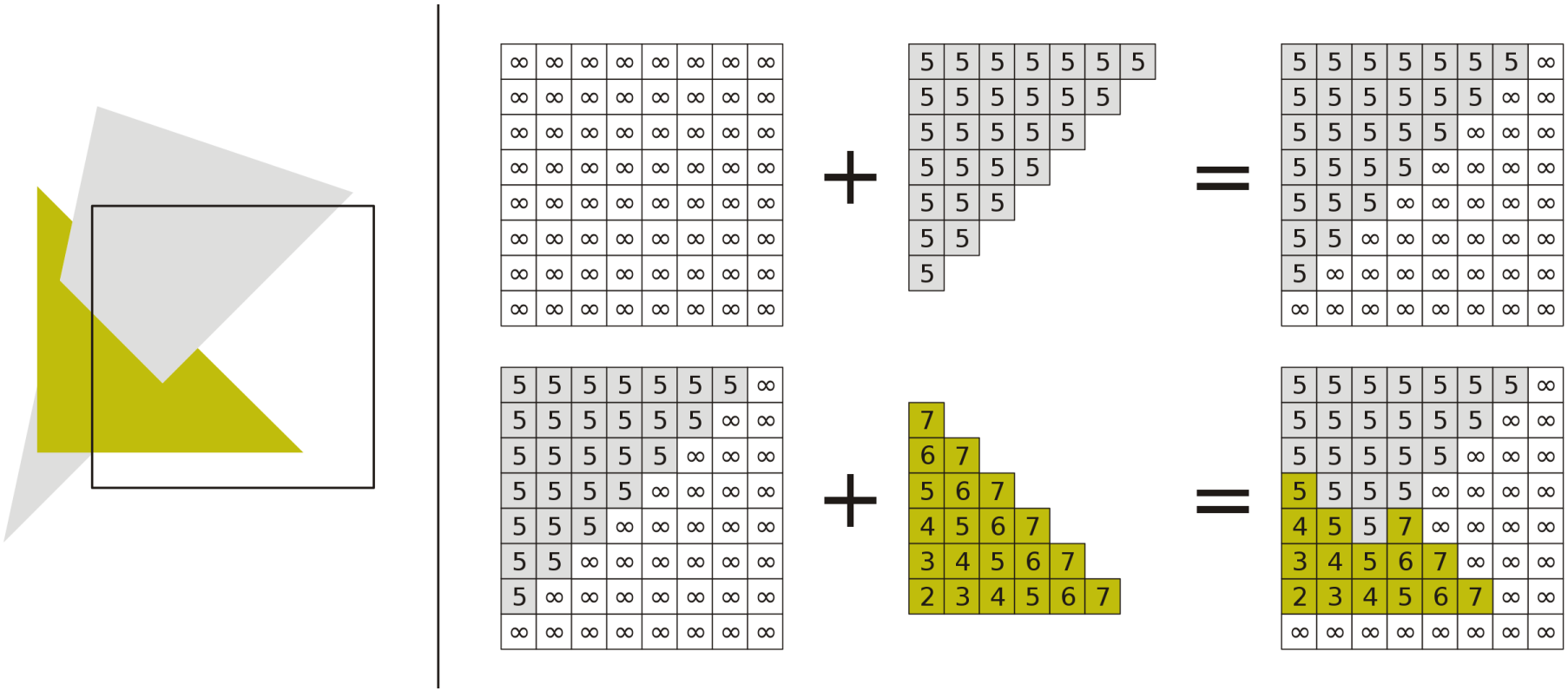
color frame buffer



Z (depth) buffer

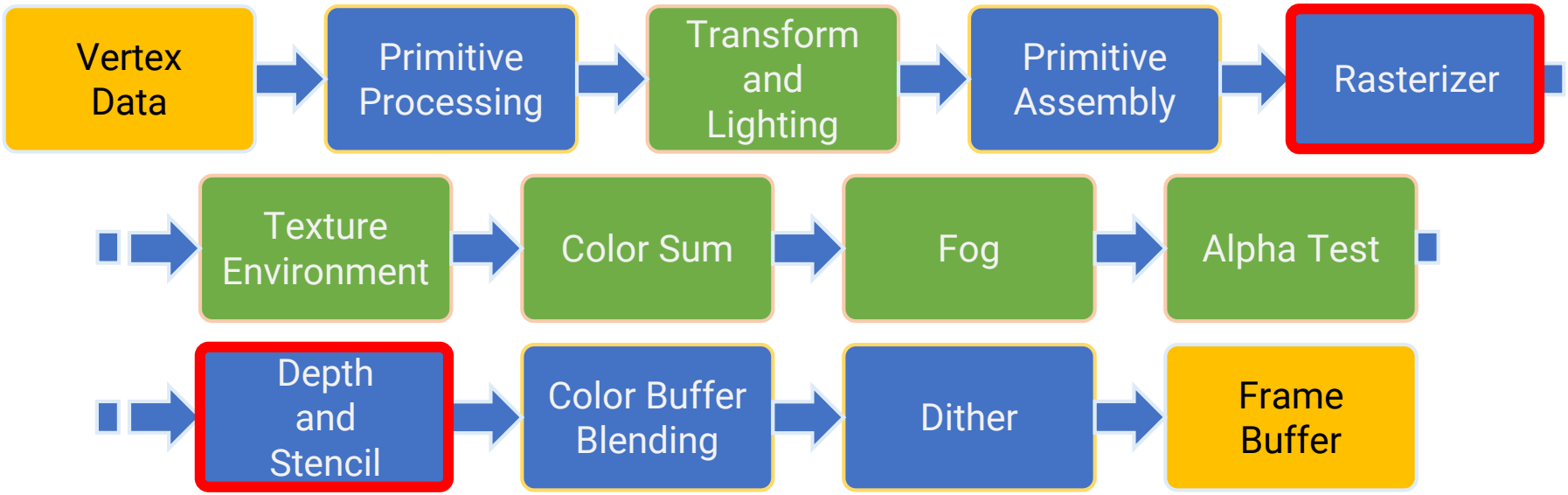
Depth Comparison (cont.)

- Z-buffer (current solution)
  - An example of Z-buffer update



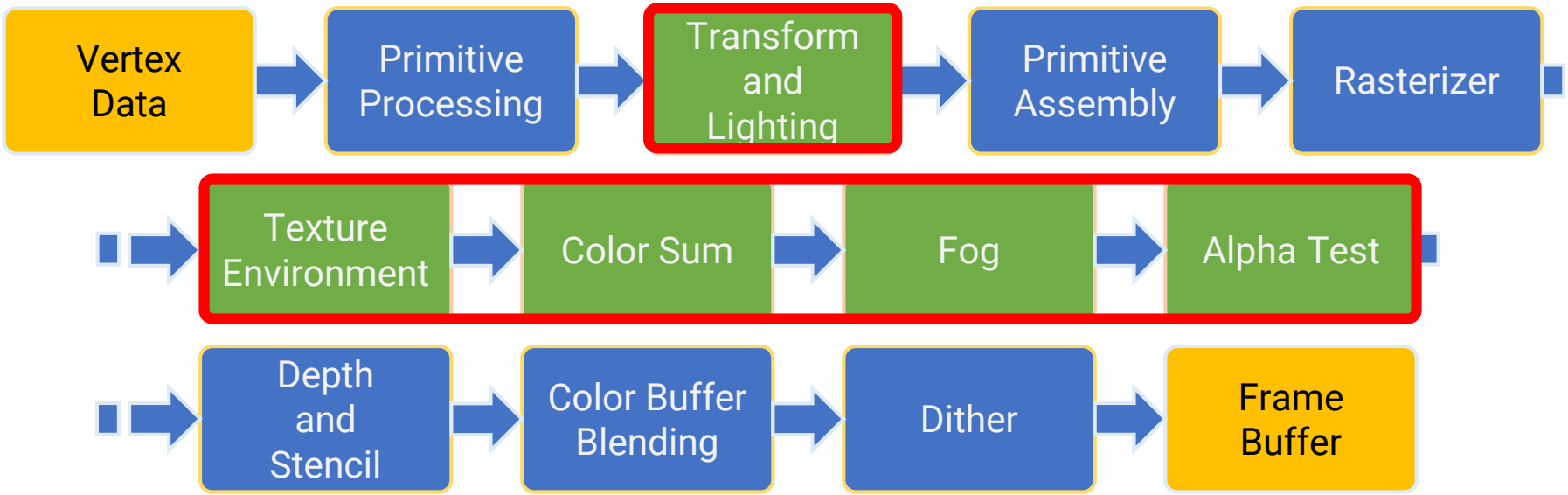
Depth Comparison (cont.)

- Used when OpenGL was first introduced
- All the functions performed by OpenGL are **fixed** and **could not** be modified except through the manipulation of the **rendering states**



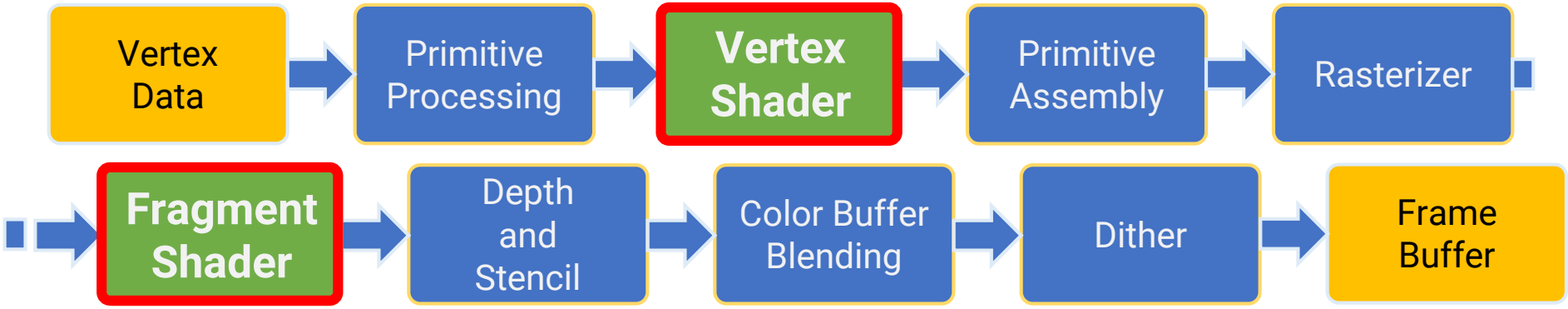
OpenGL (1.1) Fixed Function Pipeline

- All the functions performed by OpenGL are **fixed** and **could not** be modified except through the manipulation of the **rendering states**
- To provide flexibility, the stages shown in green have been replaced by **shaders**



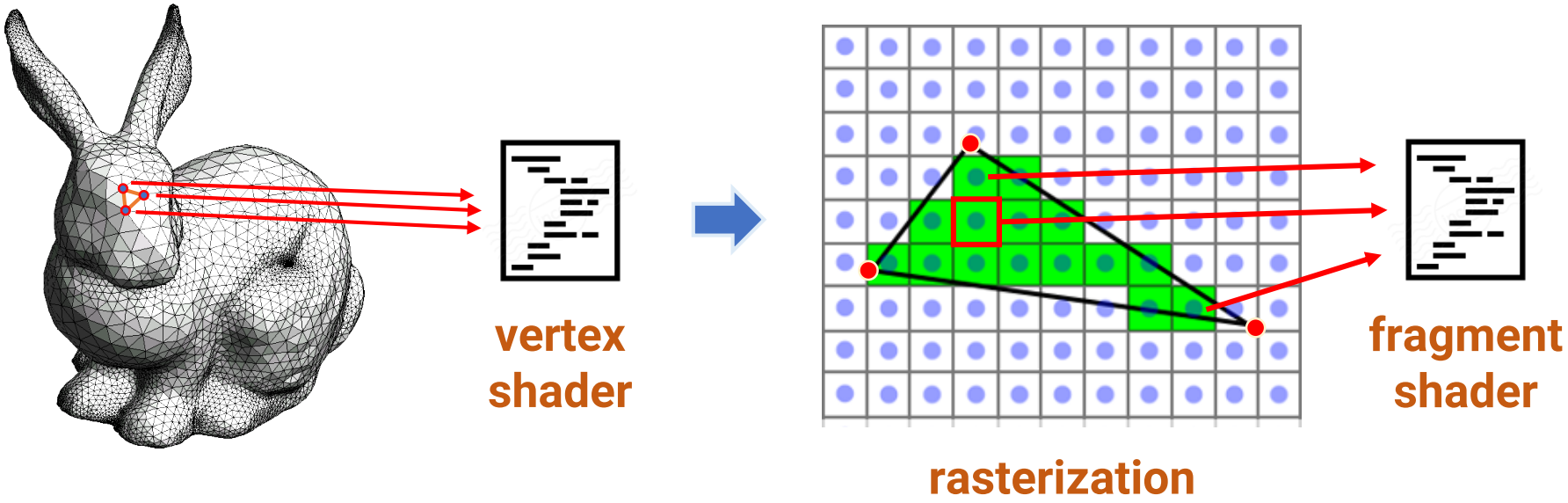
OpenGL (2.0) Graphics Pipeline

- Released in 2004
- Provide the ability to **programmatically** define the vertex transformation and lighting and the fragment operations (with small GPU programs called **shaders**)



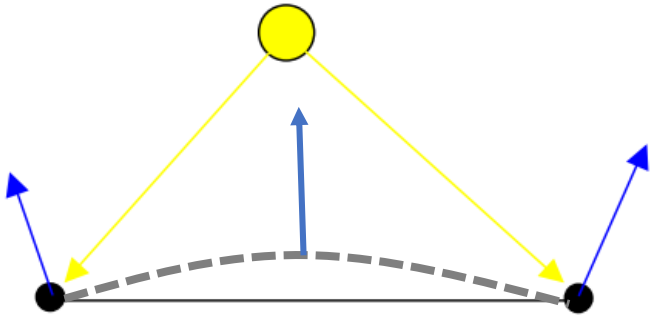
Vertex Shader and Fragment Shader

- **Important concepts**
  - The vertex shader runs **per vertex**
    - **Goal: at least transform a vertex to Clip Space**
  - The fragment shader runs **per (rasterized) fragment**
    - **Goal: calculate the color of the fragment**

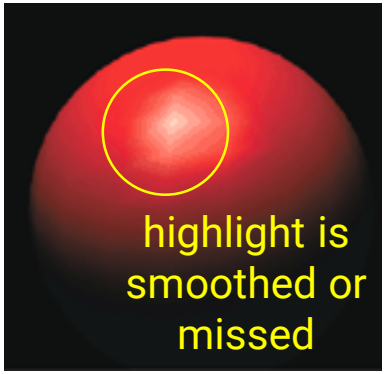


Per-Fragment Shading

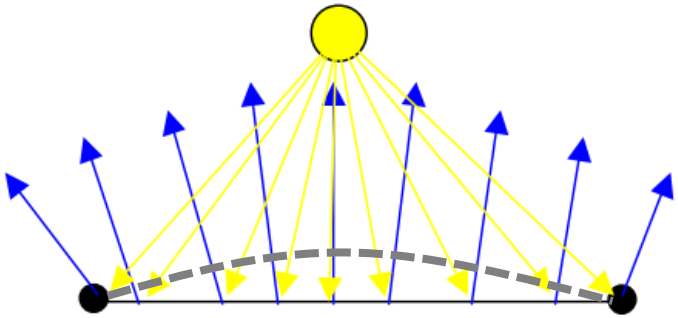
- **Gouraud shading** (interpolated vertex lighting)



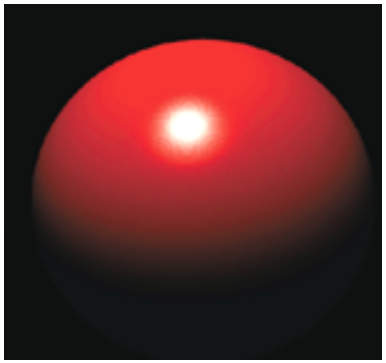
lighting color is interpolated



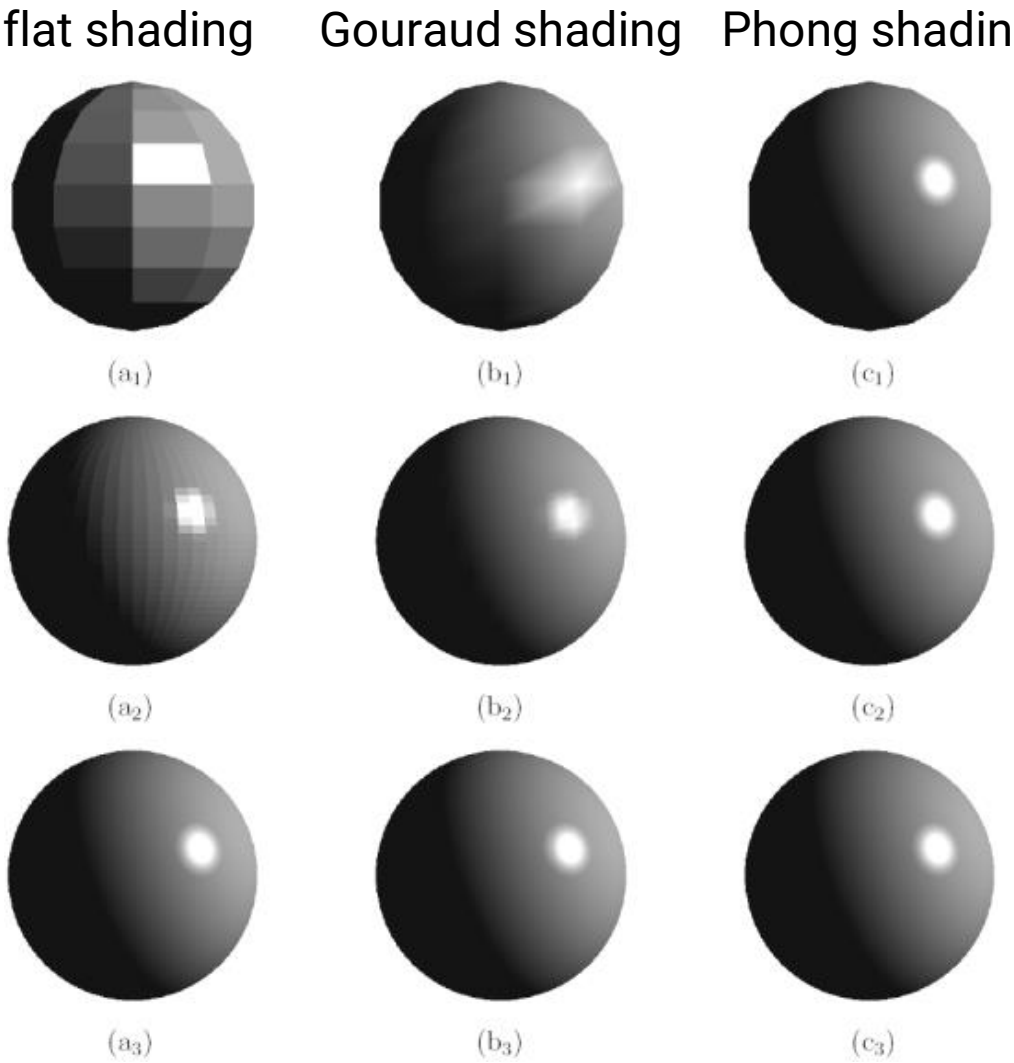
- **Phong shading**



surface normal is interpolated (how? Rasterization!)



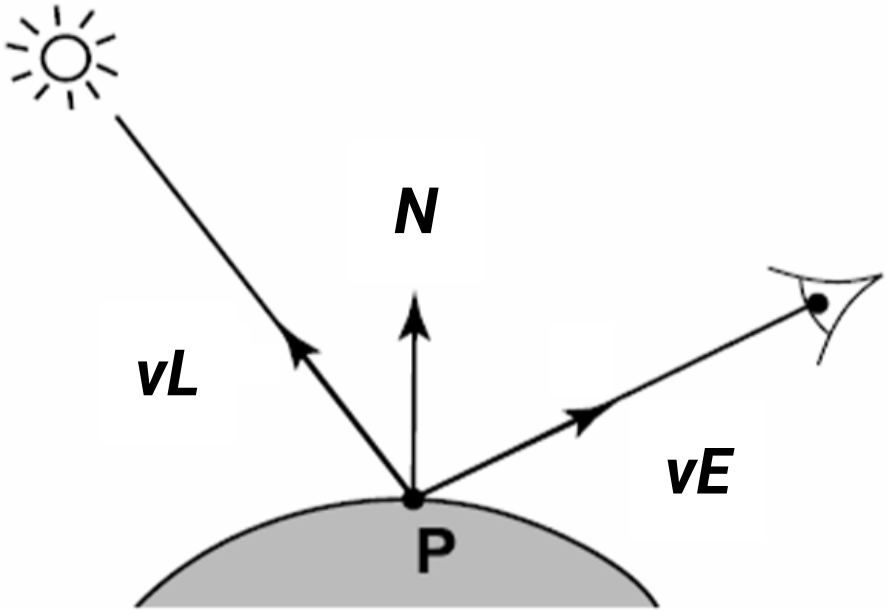
Per-Fragment Shading (cont.)



Lighting and Shading

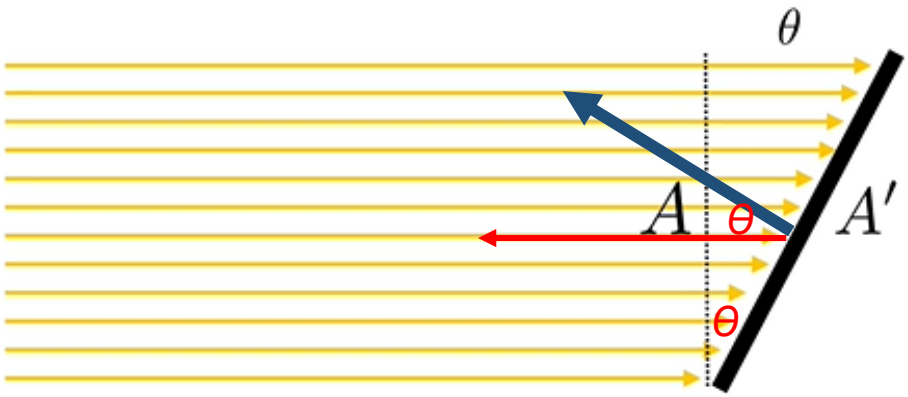
Shading

- The color of a surface is determined by
  - The **light data**
  - The **material property** of the surface
  - The **geometric setting** between the light and the surface



Lambertian Cosine Law

- Illumination on an oblique surface is less than on a normal one
- Generally, illumination falls off as **cosΘ**



$$A = A' \cos\theta$$

$$E = \frac{\Phi}{A'} = \frac{\Phi \cos\theta}{A}$$

Lights in Computer Graphics

- Point light
- Spot light
- Area light

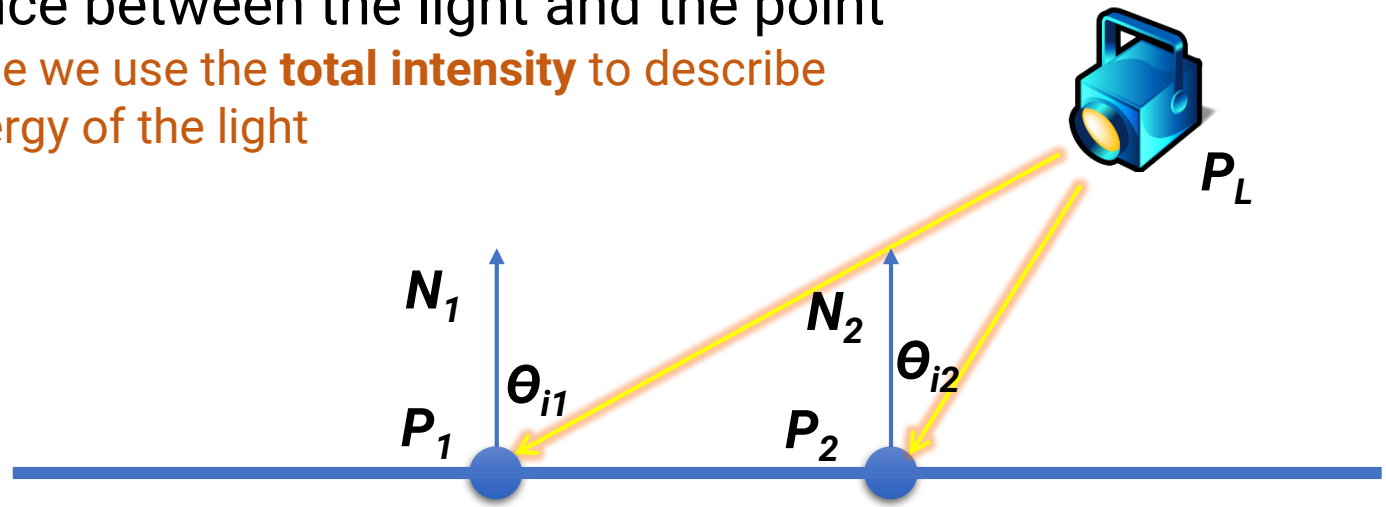
} local lights

- Directional light
- Environment light

} distant lights

Local Light

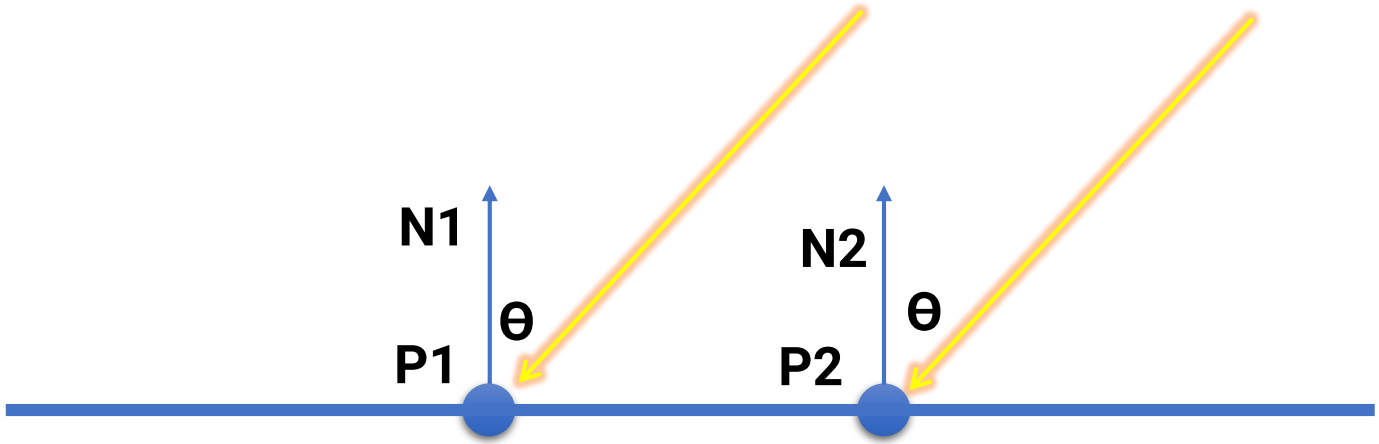
- The distance between a light and a surface is **not** long enough compared to the scene scale
- The position of light needs to be considered during shading
  - **Lighting direction** $vL = |P_L - P|$
  - **Lighting attenuation** is proportional to the square of the distance between the light and the point
    because we use the **total intensity** to describe the energy of the light



Distant Light

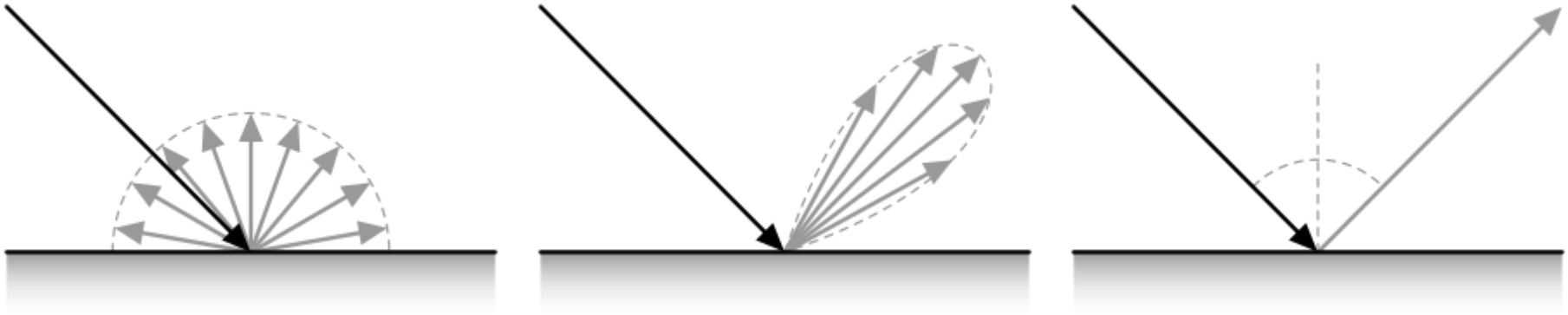
- The distance between a light and a surface is long enough compared to the scene scale and **can be ignored**
  - **Lighting direction** is **fixed**
  - **No lighting attenuation**

    because we use **directional radiance** to describe the energy of the light
- **Directional light (sun)** is the most common distant light



Materials

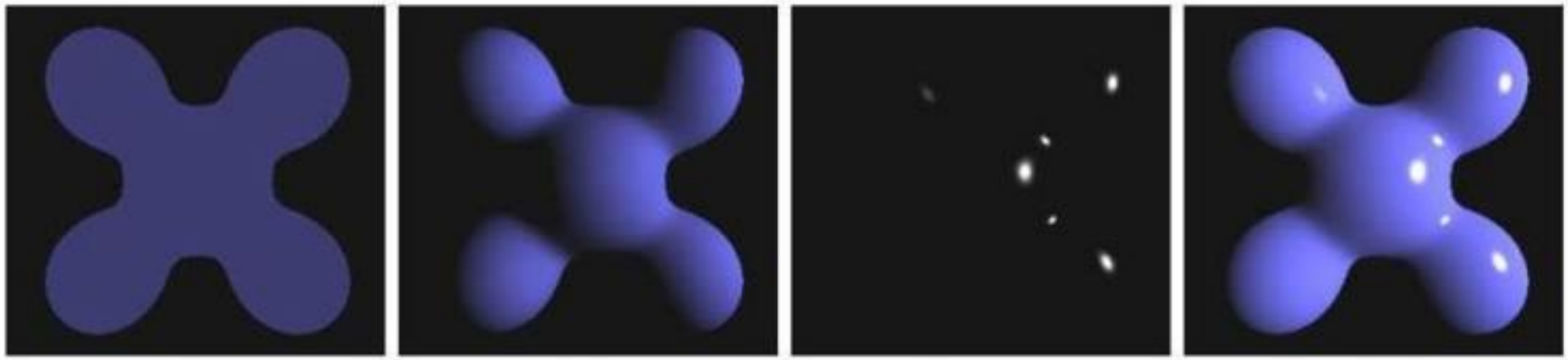
- Highly related to surface types
- The **smoother** a surface, the more reflected light is concentrated in the direction a **perfect mirror** would reflect the light



diffuse glossy specular

Phong Lighting Model

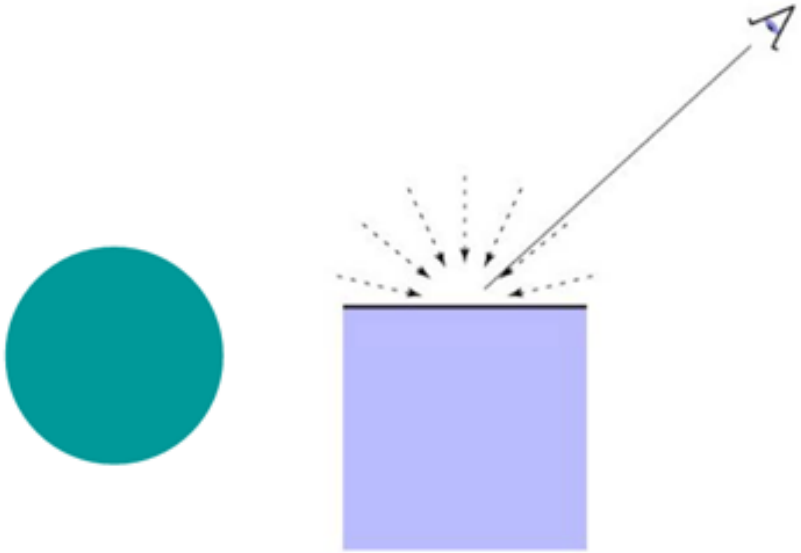
- **Diffuse reflection**
  - Light goes everywhere; colored by object color
- **Specular reflection**
  - Happens only near mirror configuration; usually white
- **Ambient reflection**
  - Constant accounted for global illumination (cheap hack)



ambient diffuse specular

Ambient Shading

- Add constant color to account for disregarded illumination and fill black shadows



the **intensity** of ambient light

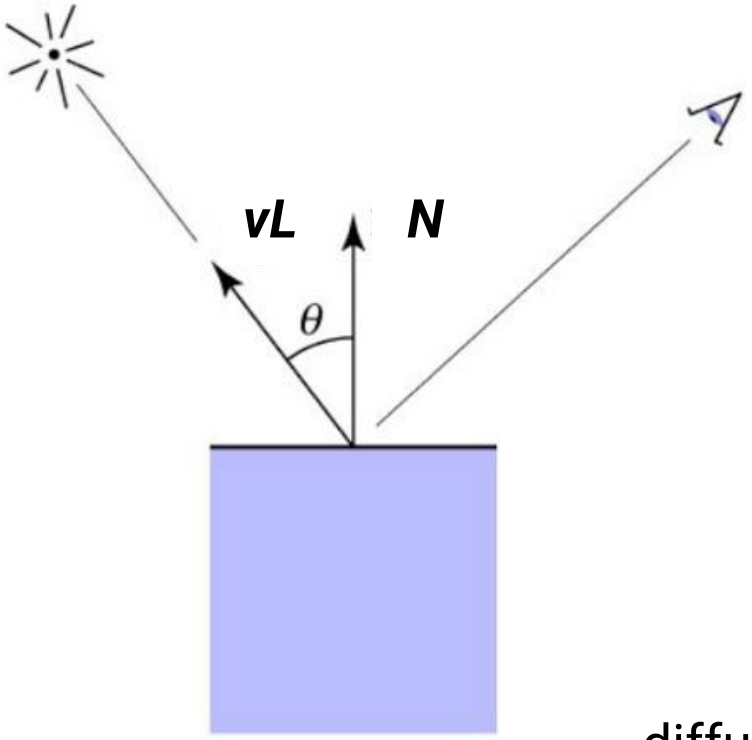
$$L_a = k_a \cdot I_a$$

ambient coefficient

reflected ambient light

Diffuse Shading

- Applies to diffuse or matte surface



illumination from source

$$L_d = k_d \cdot I \cdot \max(0, N \cdot vL)$$

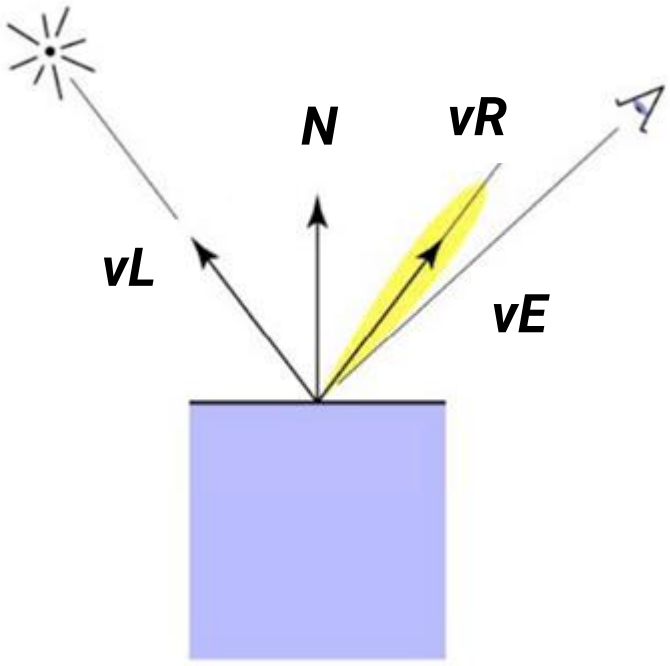
Lambertian law

diffuse coefficient

diffusely reflected light

Specular Shading

- Also known as glossy
- **Phong specular model [1975]**
  - Fall off gradually from the perfect reflection direction



$$vR = vL + 2((N \cdot vL)N - vL)$$
$$= 2(N \cdot vL)N - vL$$

specular exponent

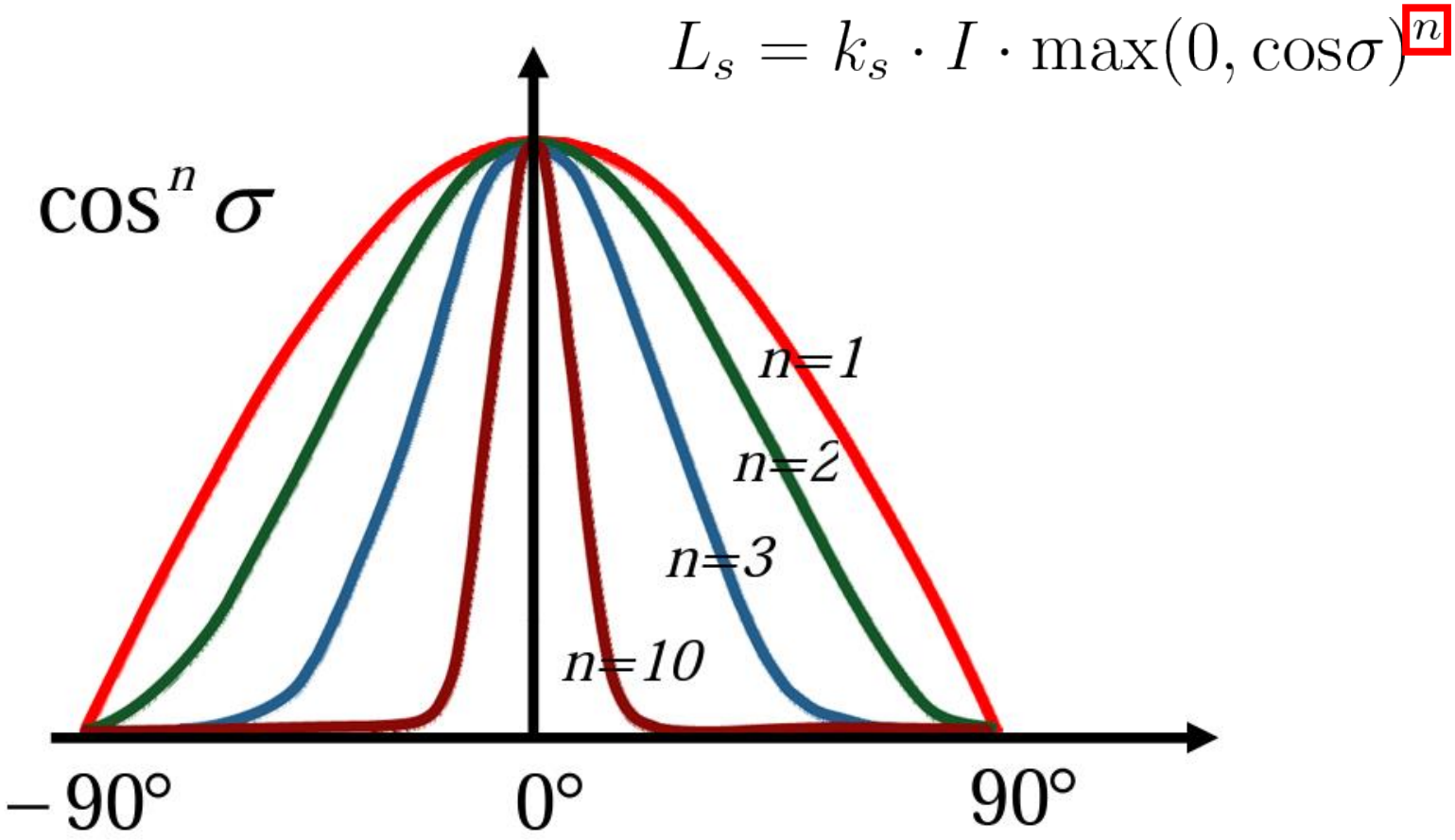
$$L_s = k_s \cdot I \cdot \max(0, \cos\sigma)^n$$
$$= k_s \cdot I \cdot \max(0, vE \cdot vR)^n$$

specular coefficient

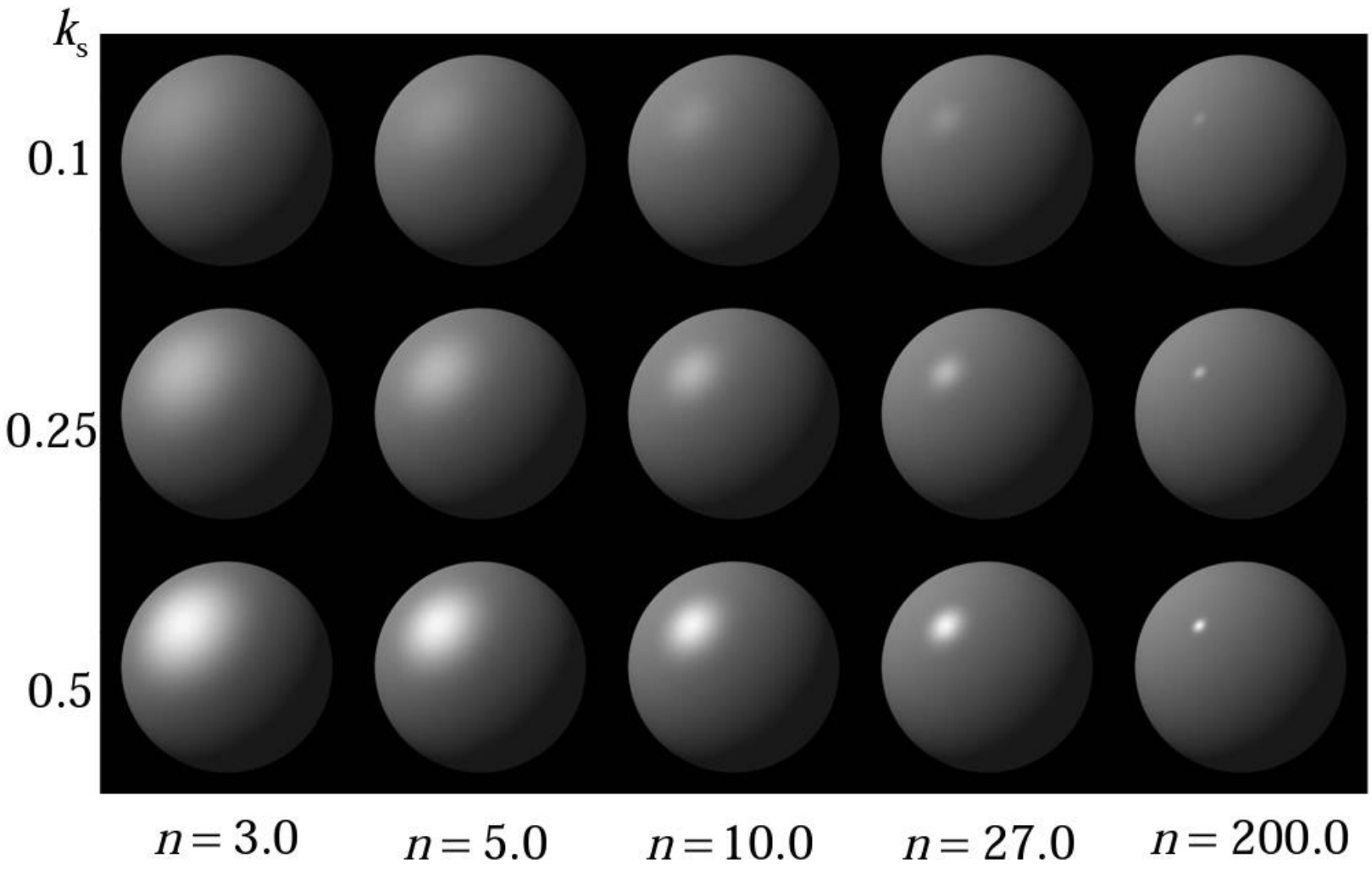
specularly reflected light

Specular Shading (cont.)

- Increase *n* narrows the lobe

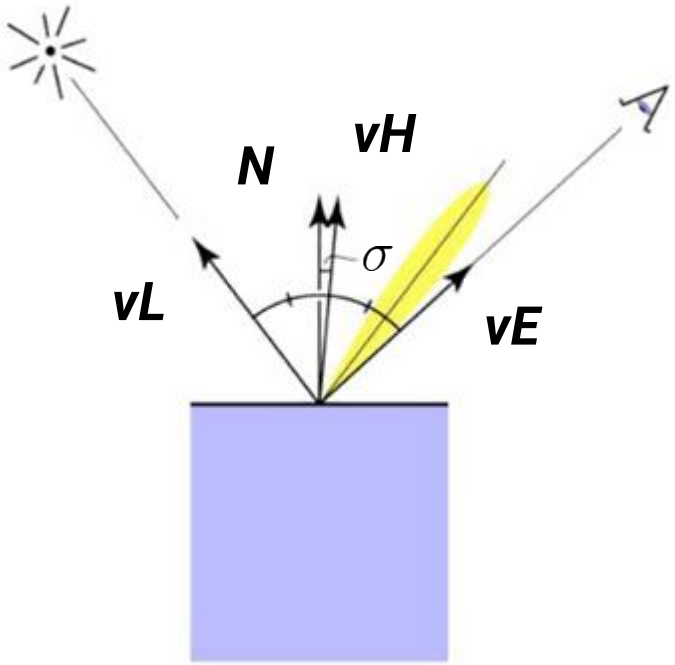


Specular Shading (cont.)



Phong specular Variant: Blinn-Phong

- Rather than computing reflection directly, just compare to normal bisection property
- One can prove $\cos^n(\sigma) = \cos^{4n}(\alpha)$

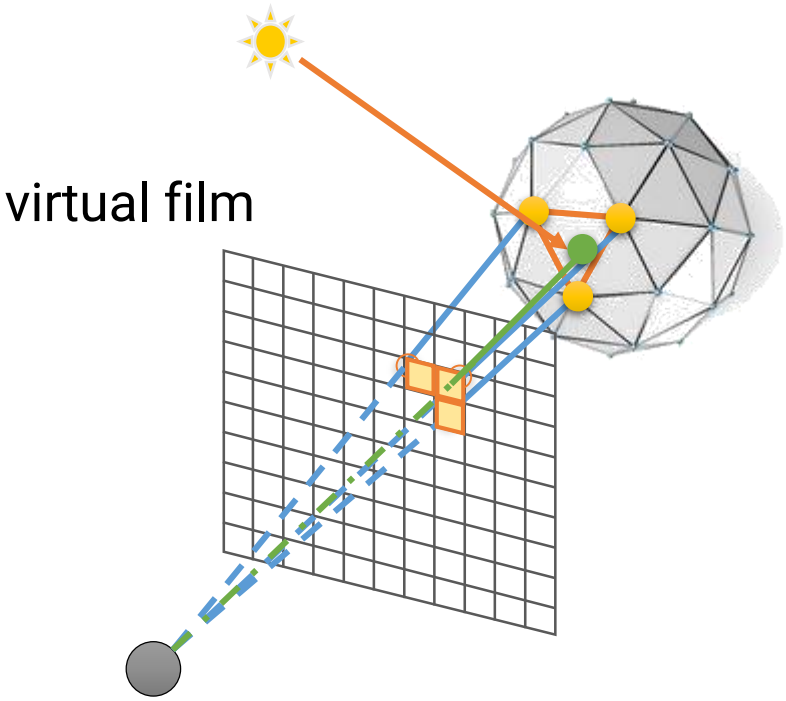


$$vH = \text{bisector}(vL, vE)$$
$$= \frac{(vL + vE)}{\|vL + vE\|}$$

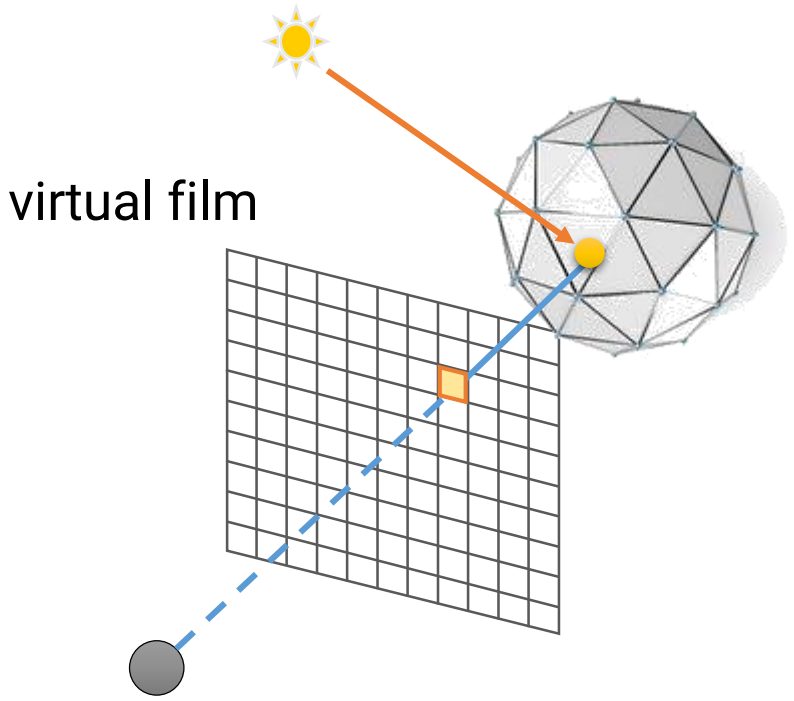
$$L_s = k_s \cdot I \cdot \max(0, \cos\sigma)^n$$
$$= k_s \cdot I \cdot \max(0, N \cdot vH)^n$$

Shading in Rasterization v.s. Ray Tracing

Rasterization

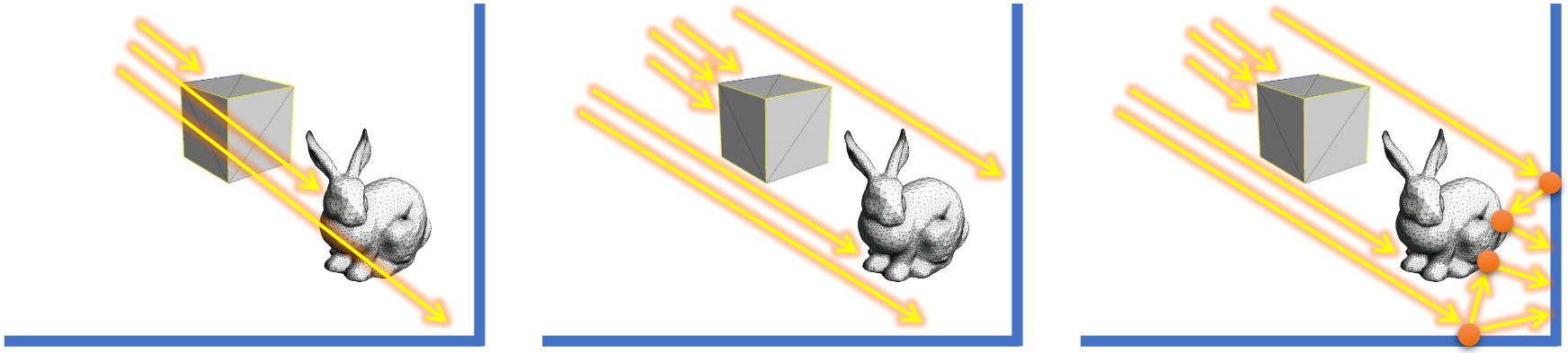


Ray tracing



Local, Direct, and Global Illumination

- Direct illumination considers only the **direct** contribution of lights
- Local illumination can be considered as direct lighting **without occlusion** (all lights are fully visible, no shadows)
- Global illumination includes **multi-bounce** illumination reflected from other surfaces (**recursive** computation!)

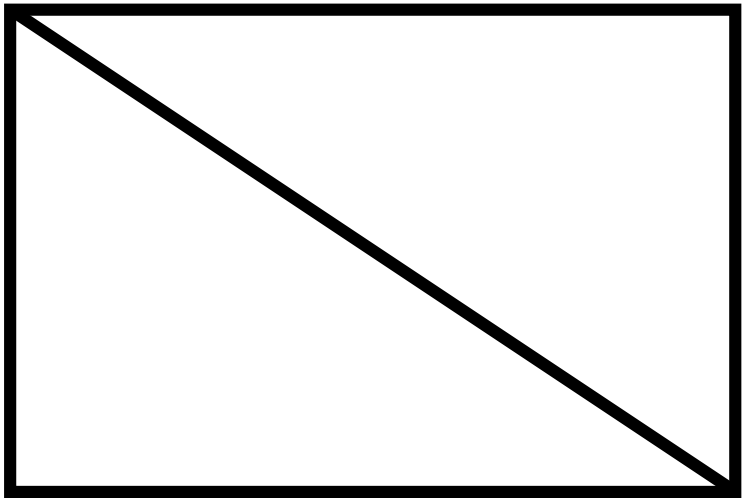


local illumination | direct illumination | global illumination

Textures

Textures

- Can be used to represent **spatially-varying** data
- Can **decouple** materials from the geometry



Geometry: two triangles

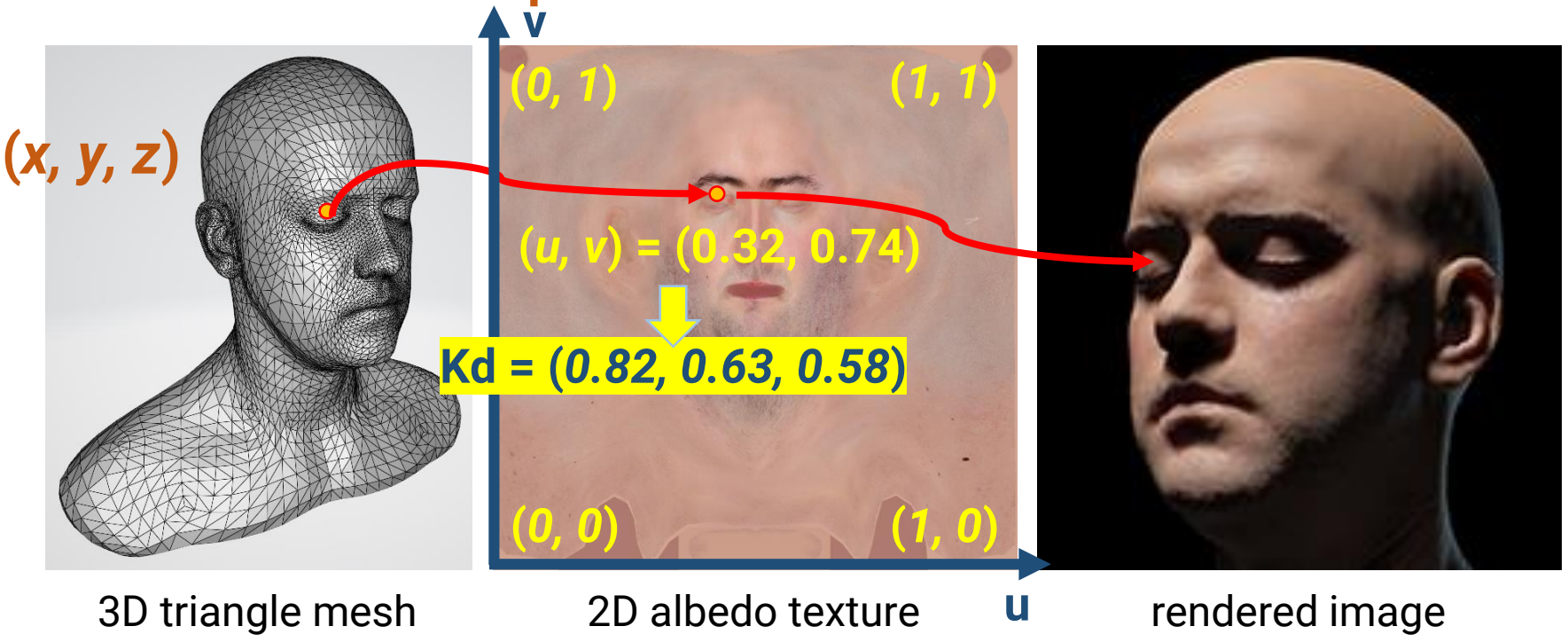
+

Material: 2D image texture

= **complex appearance**

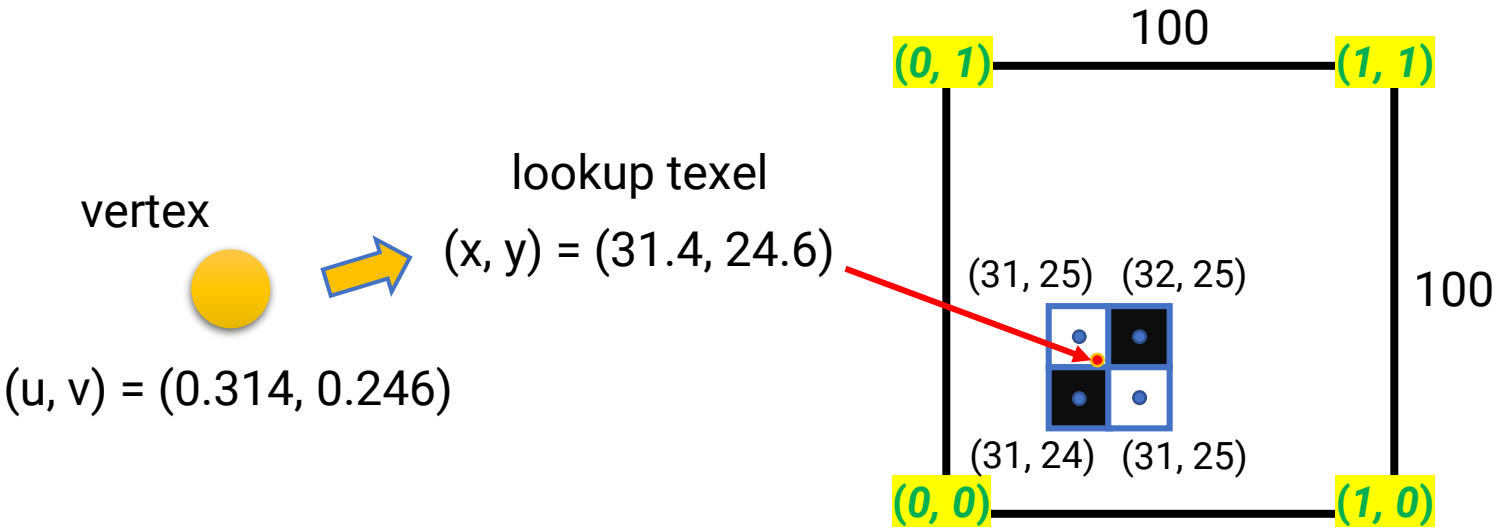
Texture Coordinate

- A coordinate to look up the texture
- The way to map a point on an **arbitrary 3D surface** to a pixel (texel) on an **image** texture
  - Need **surface parameterization**



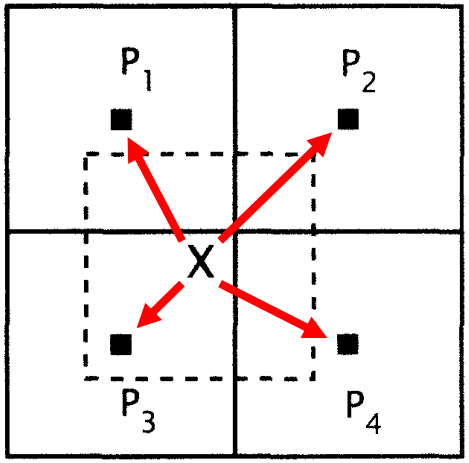
Texture Filtering

- Like an image, the content in a 2D texture is **discretely** represented by texels
- The texture coordinates can be **continuous** (especially after interpolation by the rasterization)
- How to determine the texture value if the lookup point is not at the center of a texel?



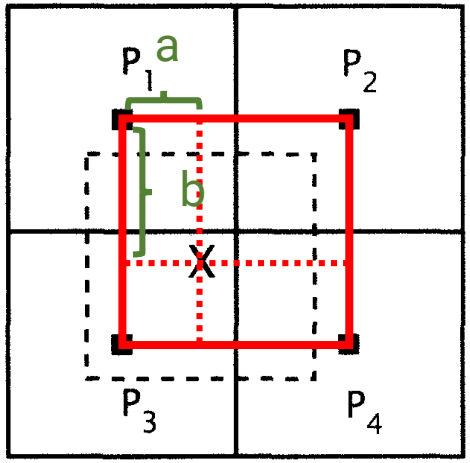
Texture Filtering (cont.)

- Strategies
  - **Nearest neighbor**
  - **Bilinear interpolation**



nearest neighbor

P_3 is closest
Use P_3's pixel value



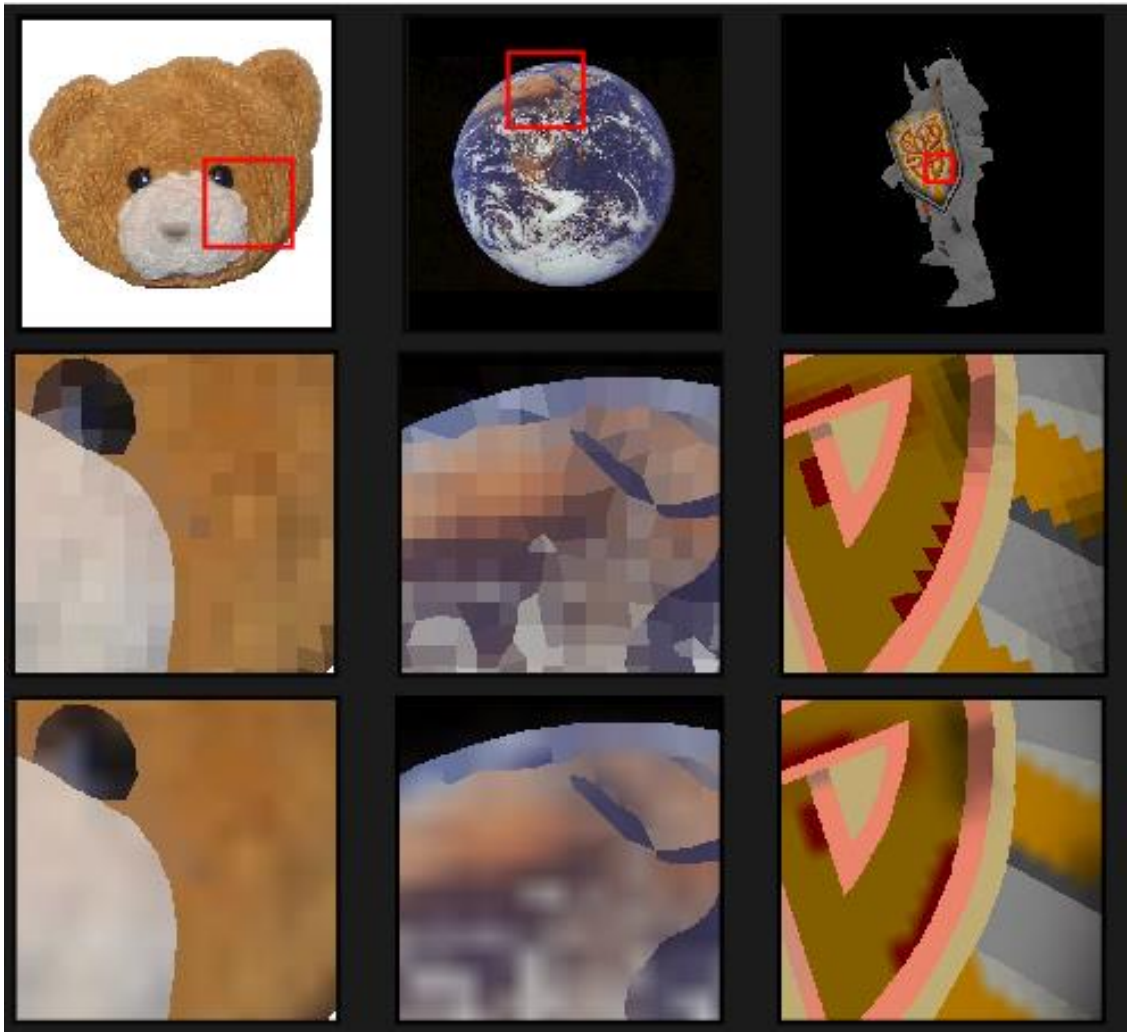
bilinear interpolation

$(1-a)(1-b)P_1 + (a)(1-b)P_2 + (1-a)(b)P_3 + (a)(b)P_4$

Texture Filtering (cont.)

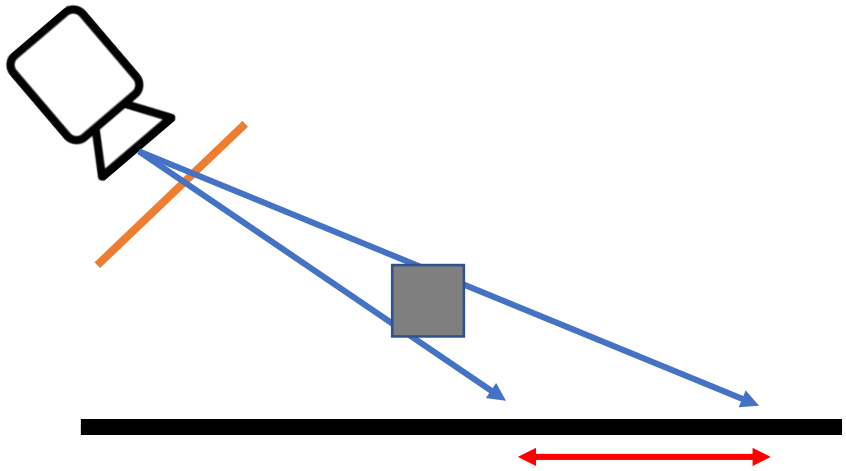
nearest neighbor

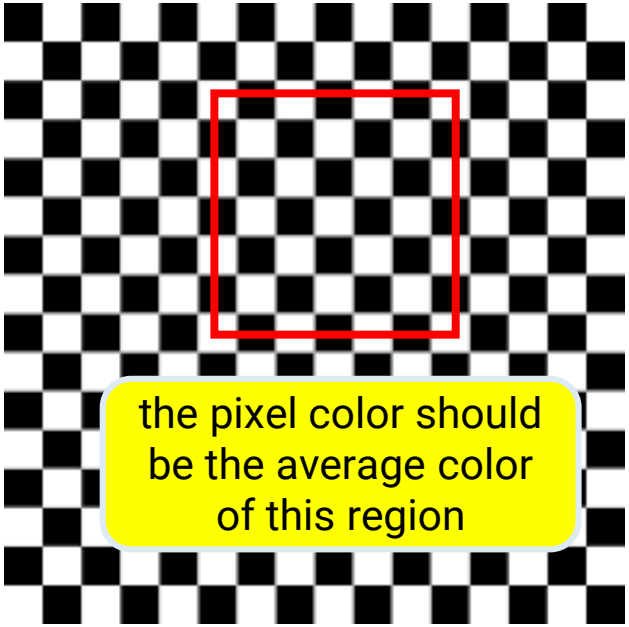
bilinear interpolation



Mipmap

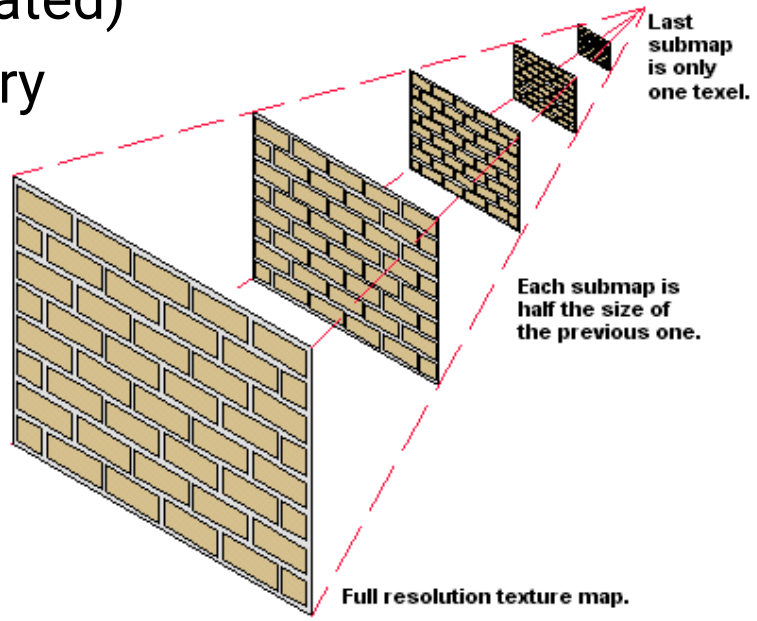
- To avoid aliasing, we should determine the regions a pixel covers (footprint) and average all the texture values inside the regions
- Time-consuming to do this in the run time!



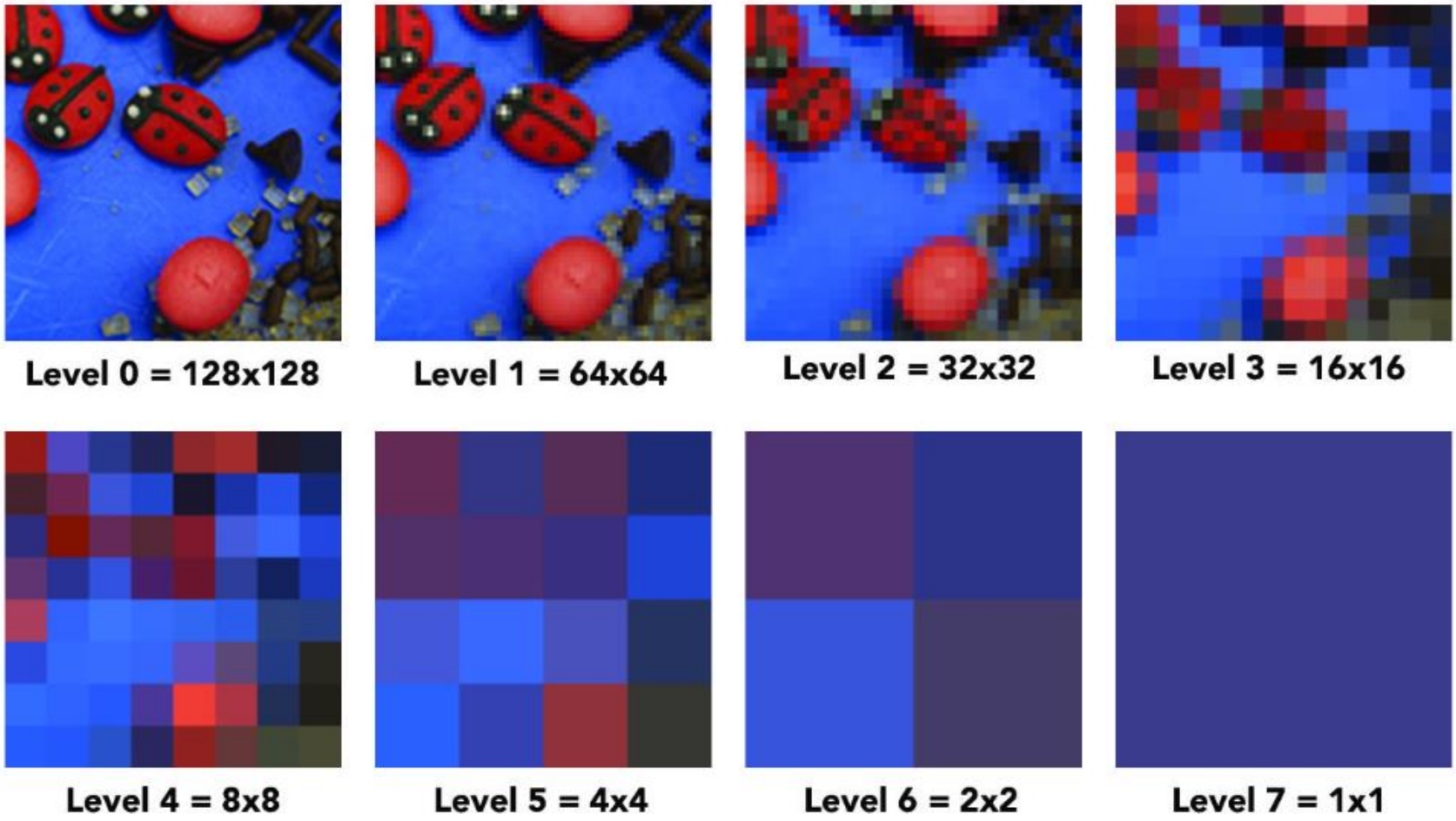


Mipmap (cont.)

- Mipmap provides a clever way to solve this problem
- **Pre-process**
  - Build a **hierarchical representation** of the texture image
  - Each level has a half resolution of its previous level (generated by linearly interpolated)
  - Take at most **1/3** more memory



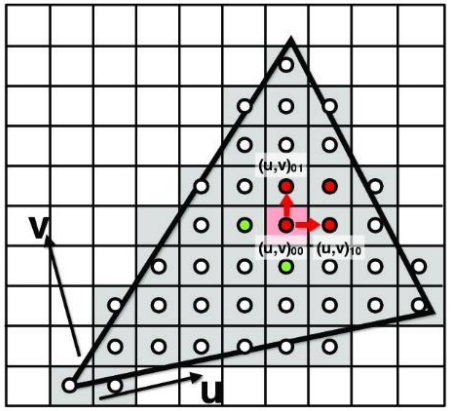
Mipmap (cont.)


Level 0 = 128x128
Level 1 = 64x64
Level 2 = 32x32
Level 3 = 16x16
Level 4 = 8x8
Level 5 = 4x4
Level 6 = 2x2
Level 7 = 1x1

Mipmap (cont.)

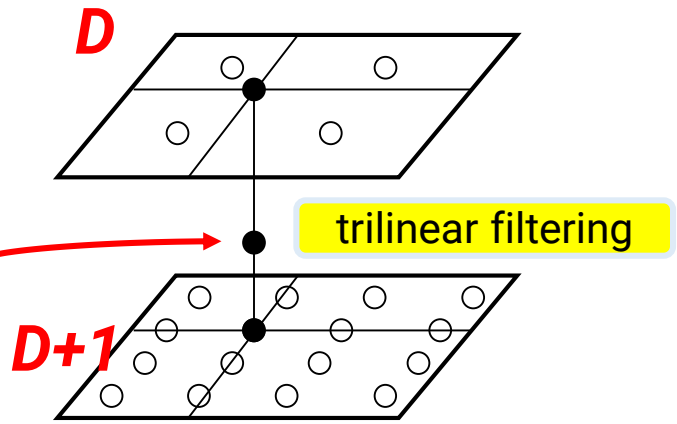
- **Run-time lookup**
  - Use **screen-space texture coordinate** to estimate its footprint in the texture space
  - Choose two levels ***D*** and ***D+1*** based on the footprint
  - Perform linear interpolation at level ***D*** to obtain a value V_D
  - Perform linear interpolation at level ***D+1*** to obtain V_{D+1}
  - Perform linear interpolation between V_D and V_{D+1}

trilinear

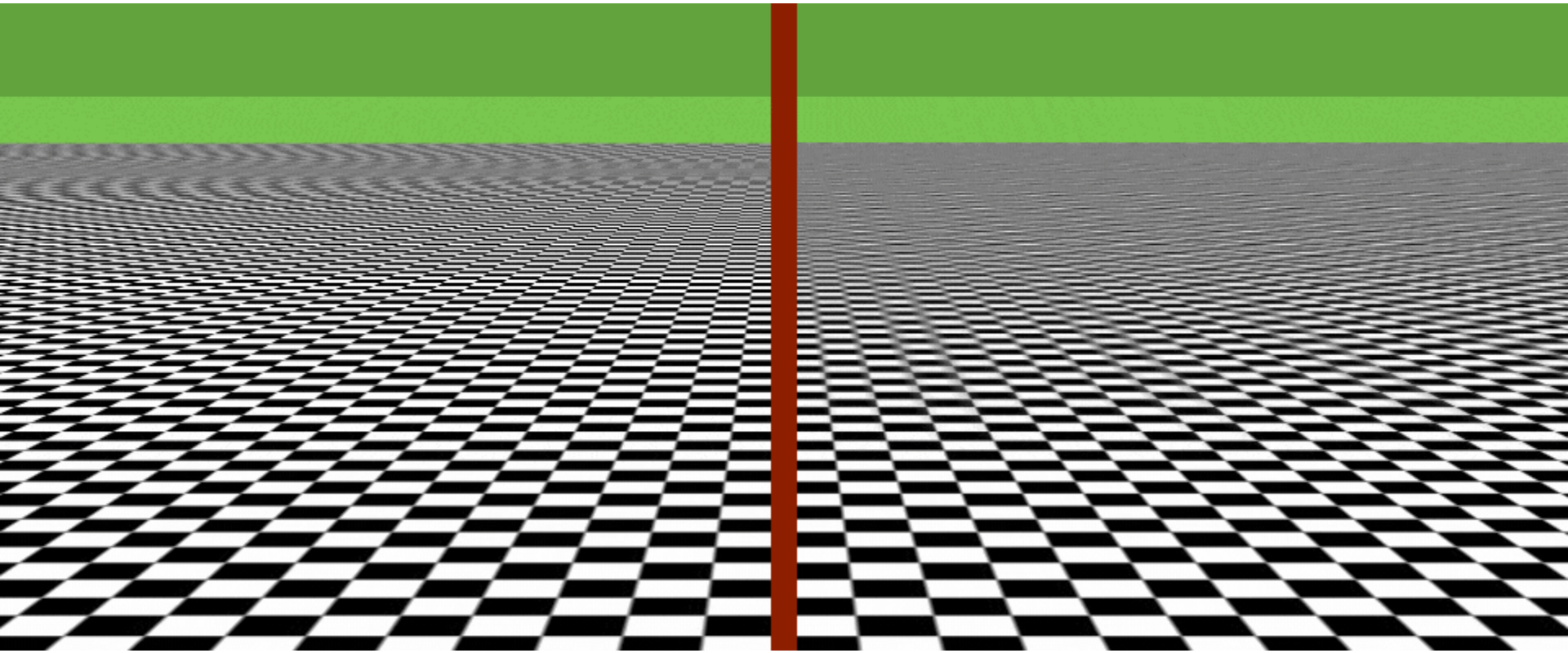


$$\frac{1}{w} = 2^{n-1-l}$$

$$l = n - 1 + \log w$$



Mipmap (cont.)



without mipmap

with mipmap

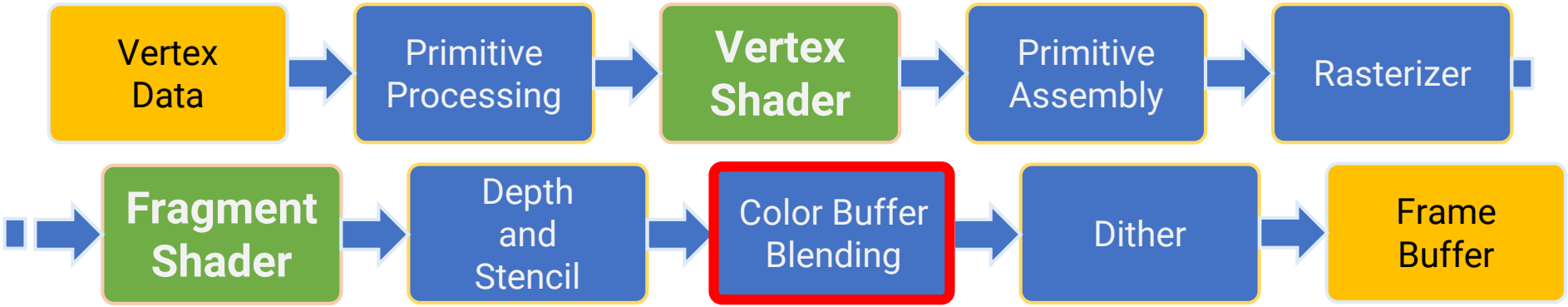
Transparency

Transparency in Rasterization

- Z-buffer provides an efficient way to solve the visibility for opaque objects
  - Keep the closest objects from the camera during the rendering process
- For transparent objects, we will see occluded objects through the transparent one
- However, in rasterization transparency is difficult to resolve **because each polygon only has its own information**
  - It does not know which triangle locates behind, so it cannot determine the pixel color in its fragment shader

Transparency in Rasterization (cont.)

- The GPU rendering pipeline provides a way to simulate transparency
- **Major idea**
  - Render transparent objects **in an order w.r.t their distance to the camera** (farther objects first)
  - When rendering transparent objects, blend the fragment color with the previous results in the color buffer



Blending Equation

- OpenGL provides flexibility to composite the fragment color when rendering transparent objects

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

blending equation

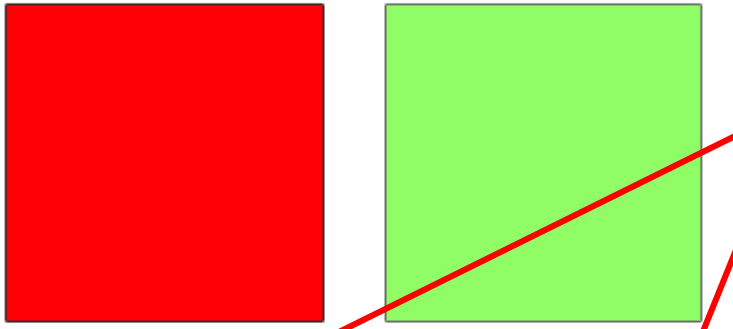
- $\bar{C}_{source}$: the source color vector (the color output by the fragment shader)
- $\bar{C}_{destination}$: the destination color vector (the color vector currently stored in the color buffer)
- F_{source} : the source factor value (set the impact of the alpha value on the source color)
- $F_{destination}$: the destination factor value (set the impact of the alpha value on the destination color)

Blending Equation (cont.)

- Example

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

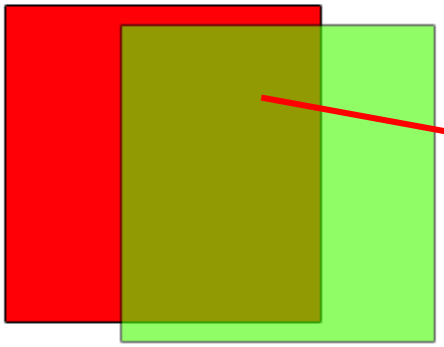
GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA



RGBA = (1.0, 0.0, 0.0, 1.0) (0.0, 1.0, 0.0, 0.6)

Alpha: the transparency of the surface

- For opaque surfaces, alpha = 1
- For totally transparent surfaces, alpha = 0
- For semi-transparent surfaces, alpha is between 0 and 1

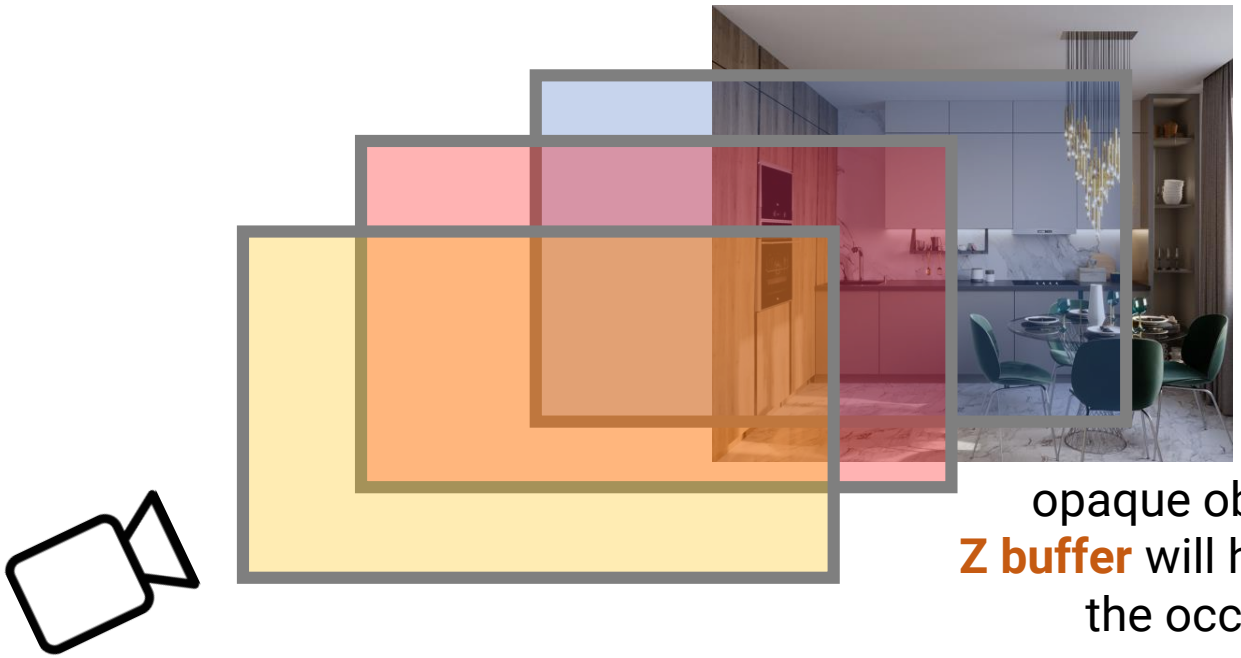


(0.0, 1.0, 0.0, 0.6) * **0.6** +
(1.0, 0.0, 0.0, 1.0) * **(1 – 0.6)**

= (0.4, 0.6, 0.0, 0.76)

Rendering Algorithm for Transparency

- Render **opaque** objects first **in any order**
- Render **transparent** objects **in an order w.r.t their distance to the camera** (farther objects first)
- Each time, blend with the color buffer

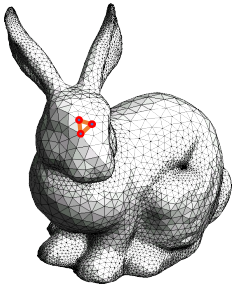


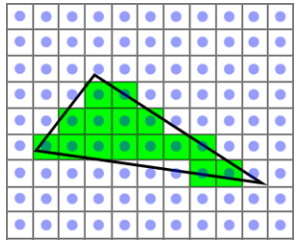
opaque objects:
Z buffer will handle
the occlusion

Deferred Shading

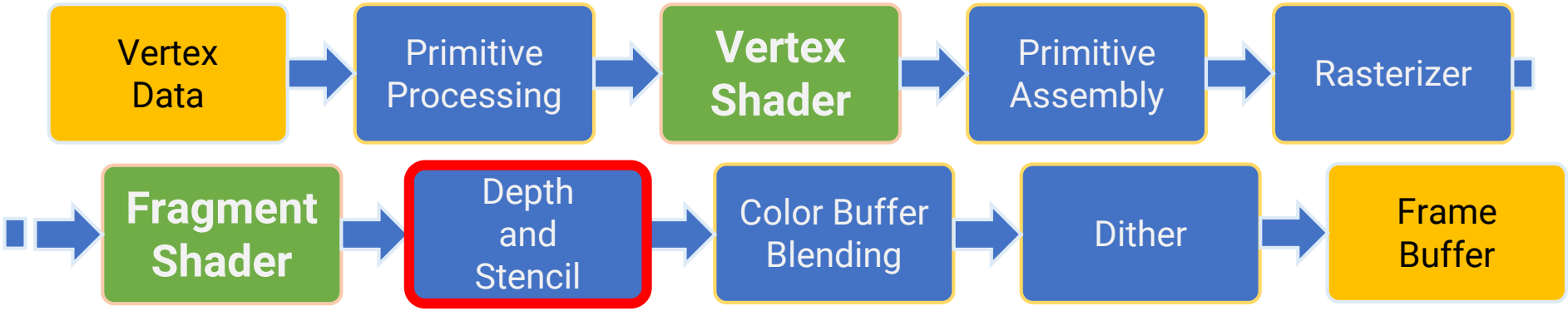
Problem Formulation

- Forward rendering





vertex transform

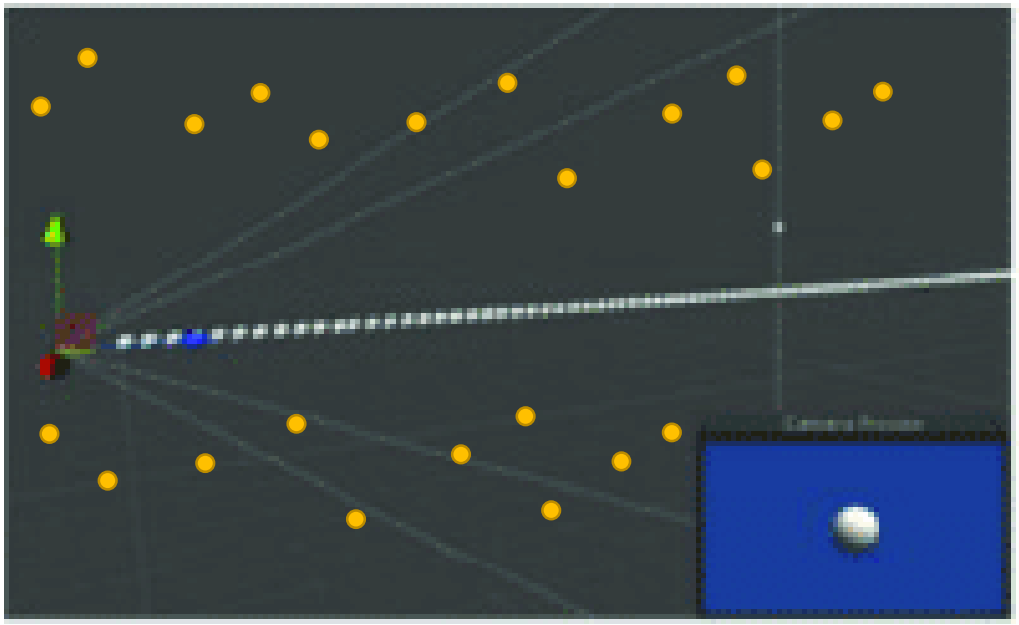


lighting
texturing

fragments are discarded if they failed in the depth test!

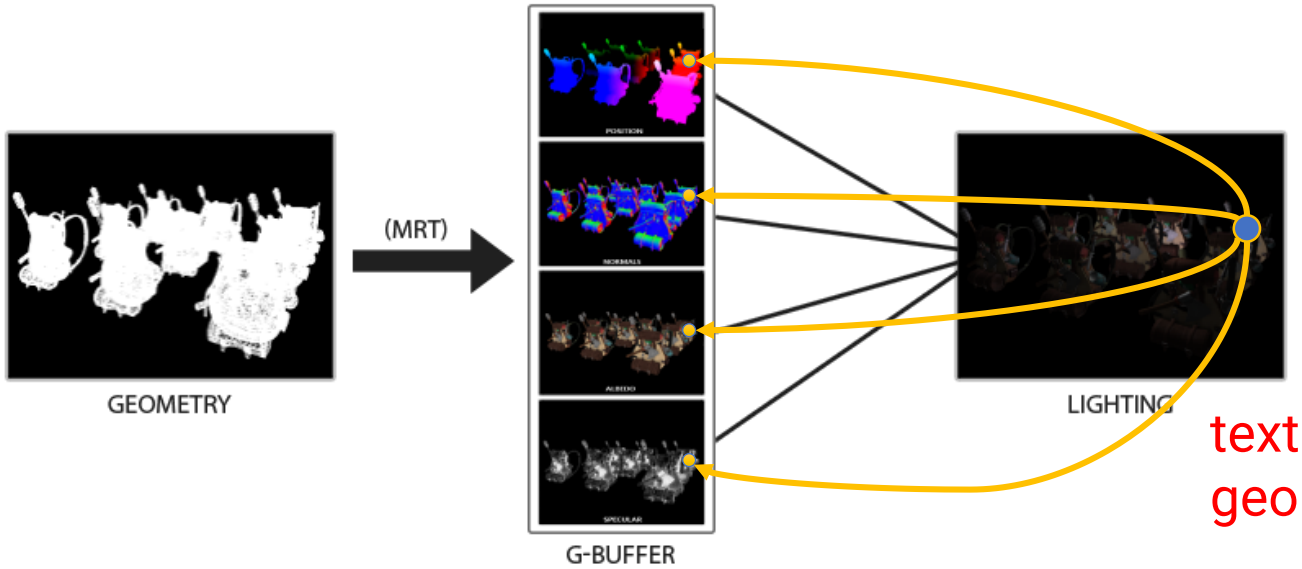
Problem Formulation (cont.)

- Problem of forward rendering
  - In scenes with **many** lights and **complex** layouts, lots of computation resources are wasted on shading the **occluded** surfaces that will finally be discarded!
  - Overdraw per pixel!



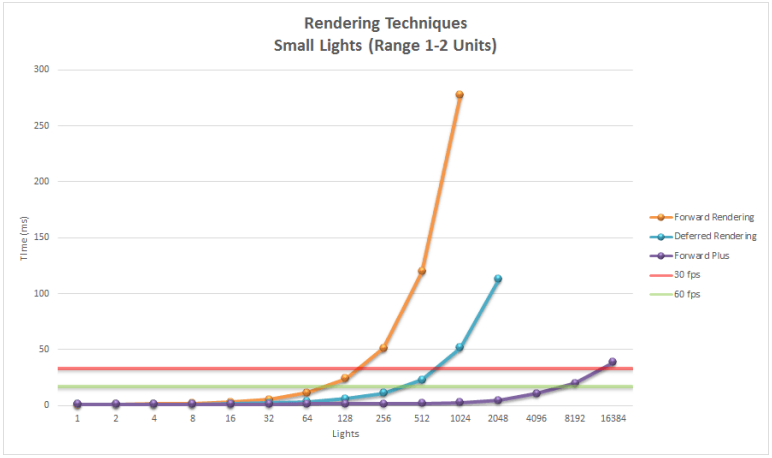
Deferred Shading

- A **Two-pass** rendering algorithm
  - In the first pass, recognize all **visible** surfaces from the camera, store their **geometry** and **material** properties in **geometry buffers (G-buffers)**
  - In the second pass, only compute lighting on the visible surfaces based on the G-buffers



Deferred Shading (cont.)

- **Pros**
  - **Reduce unnecessary lighting computation**

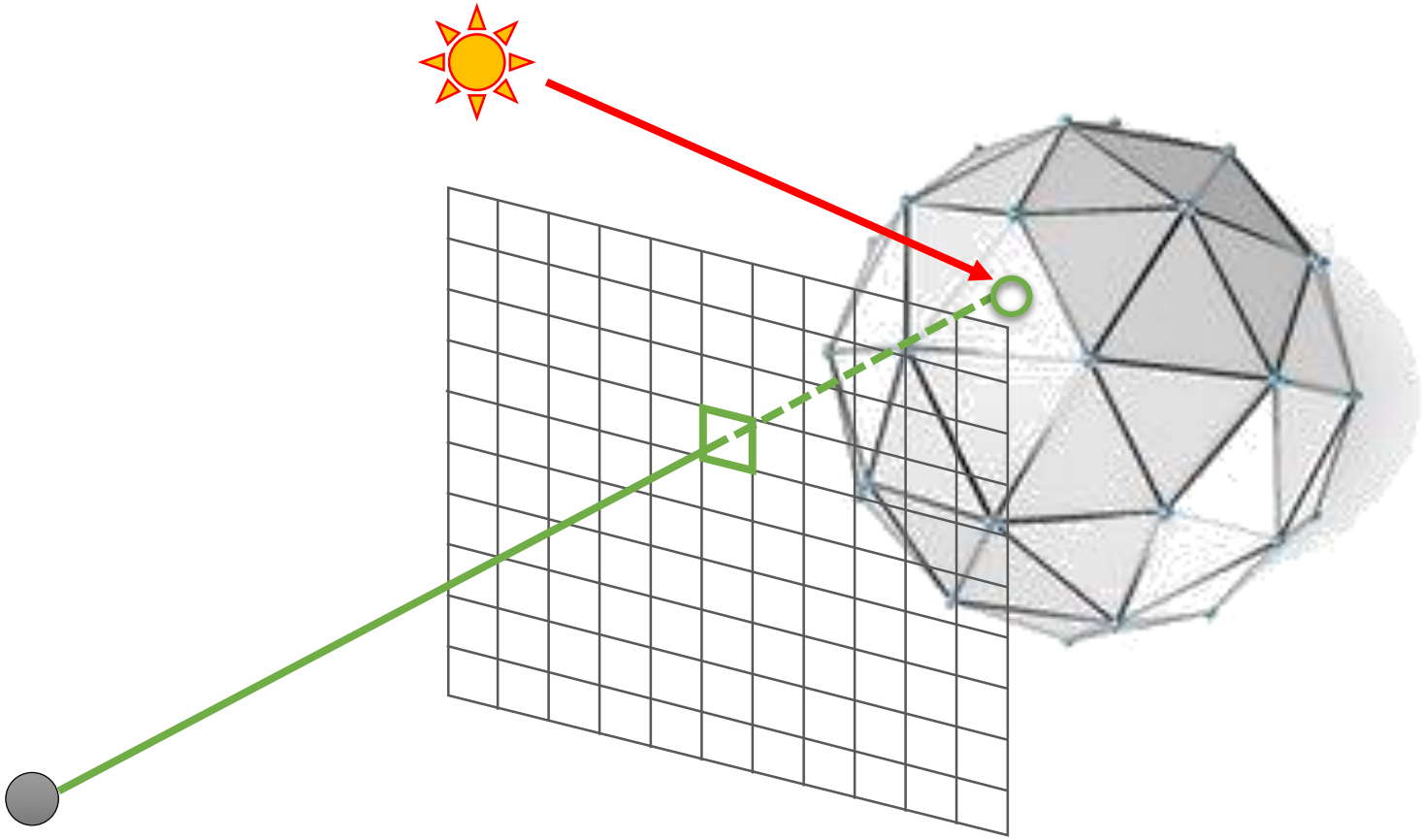


- **Cons**
  - **Large memory bandwidth**
  - **Difficult for multi-sampled anti-aliasing (MSAA)**
  - Transparent objects should be rendered using forward rendering

Shadow Mapping

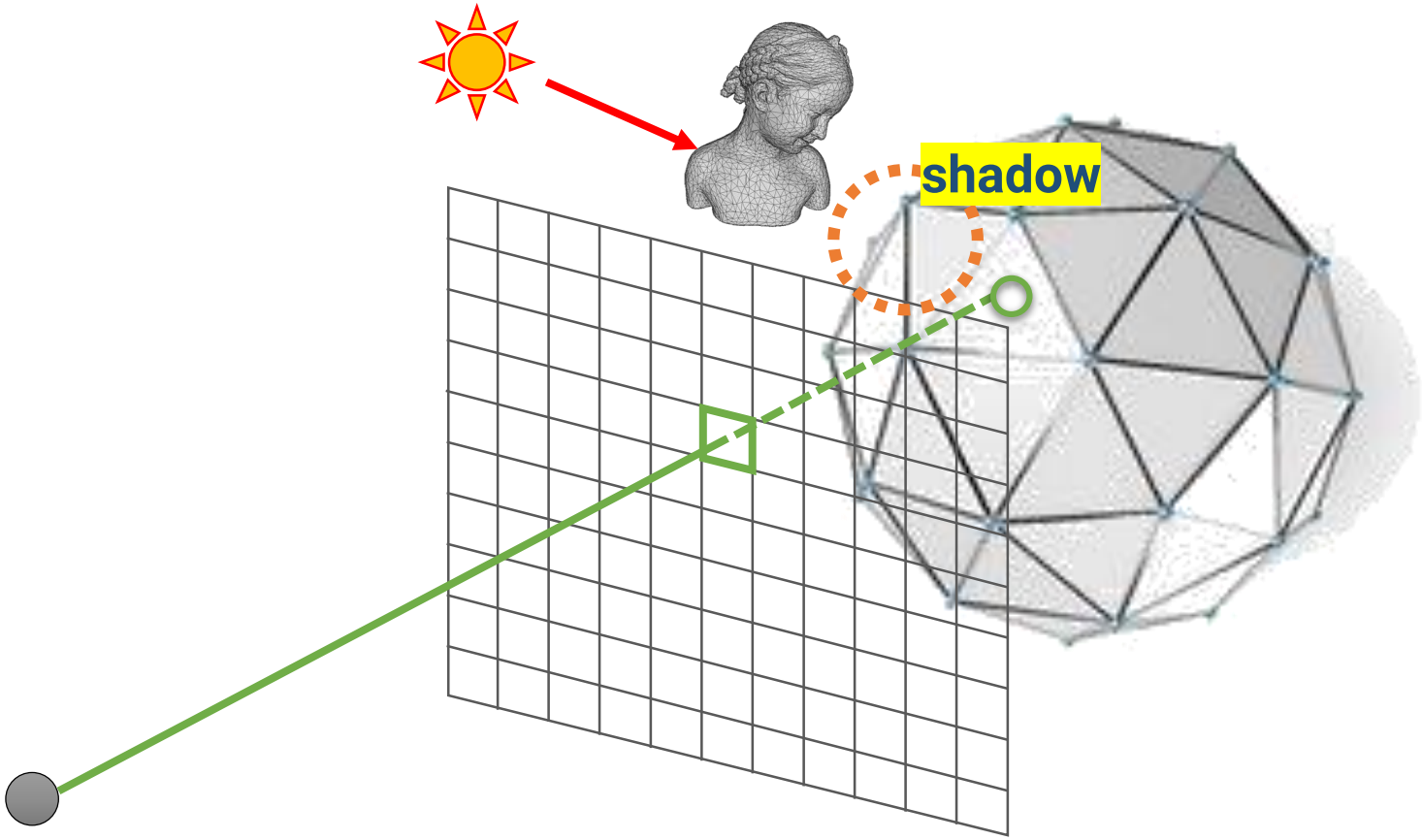
Shadow

- The standard GPU rendering pipeline can only handle unoccluded illumination (local illumination)



Shadow (cont.)

- It is common that a lighting direction is occluded by some other objects

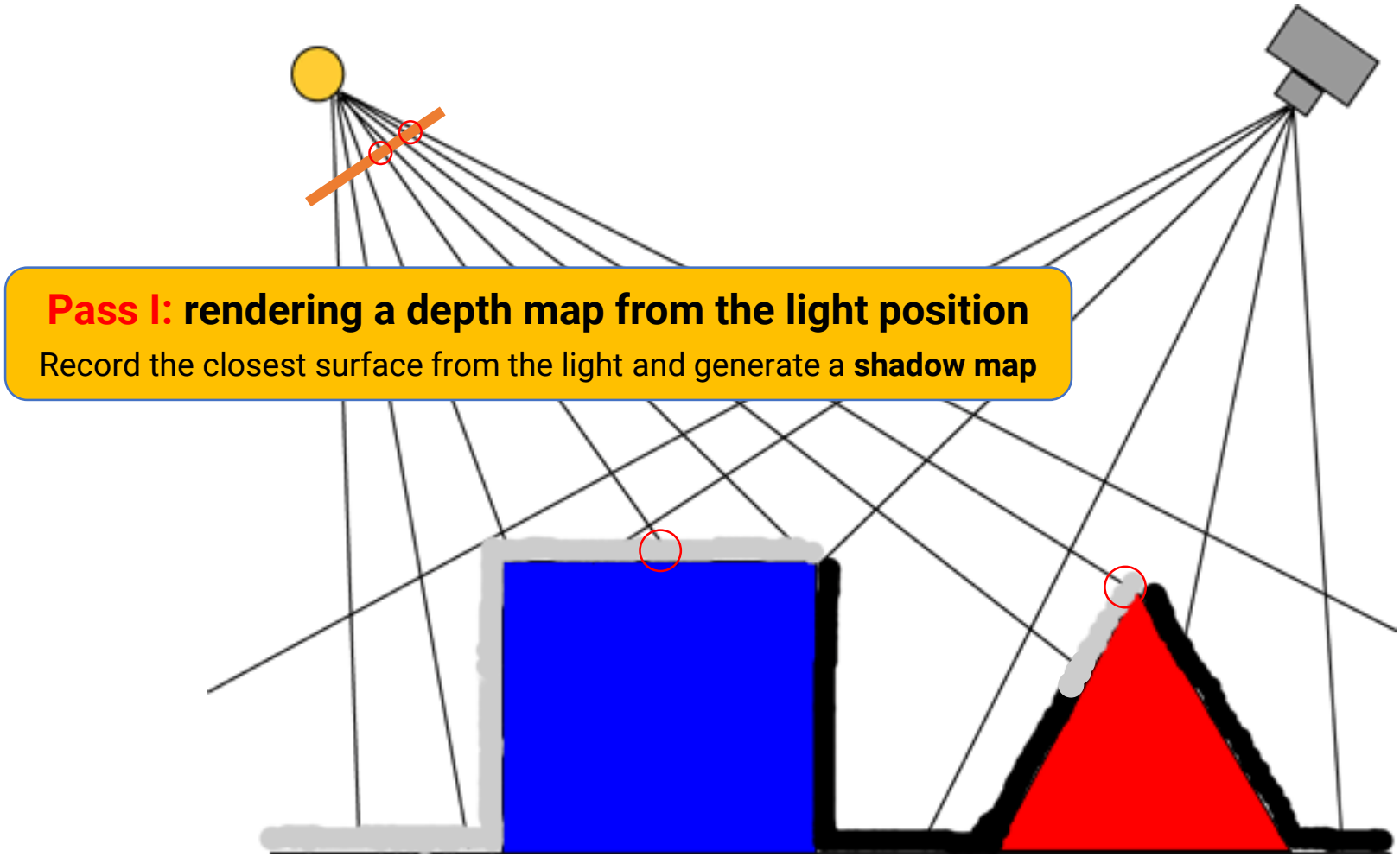


Shadow Map Overview

- Like the case of transparency, rendering shadows is difficult for rasterization because **each polygon only has its own information**
  - It does not know which triangle blocks the light, so it cannot determine the shadow attenuation in its fragment shader

- **Shadow map** is a two-pass rendering technique for simulating shadows using rasterization

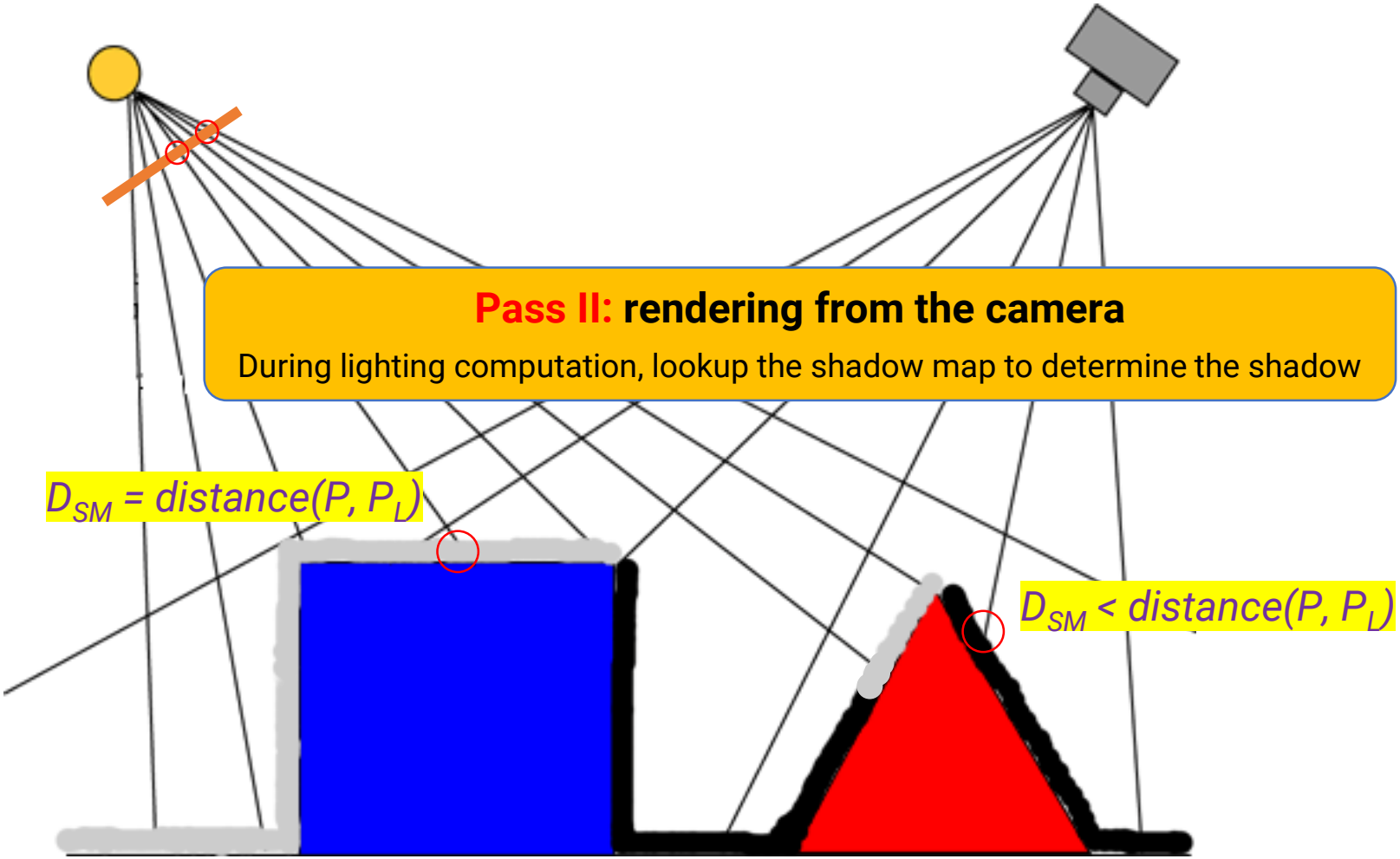
Shadow Map Overview (cont.)



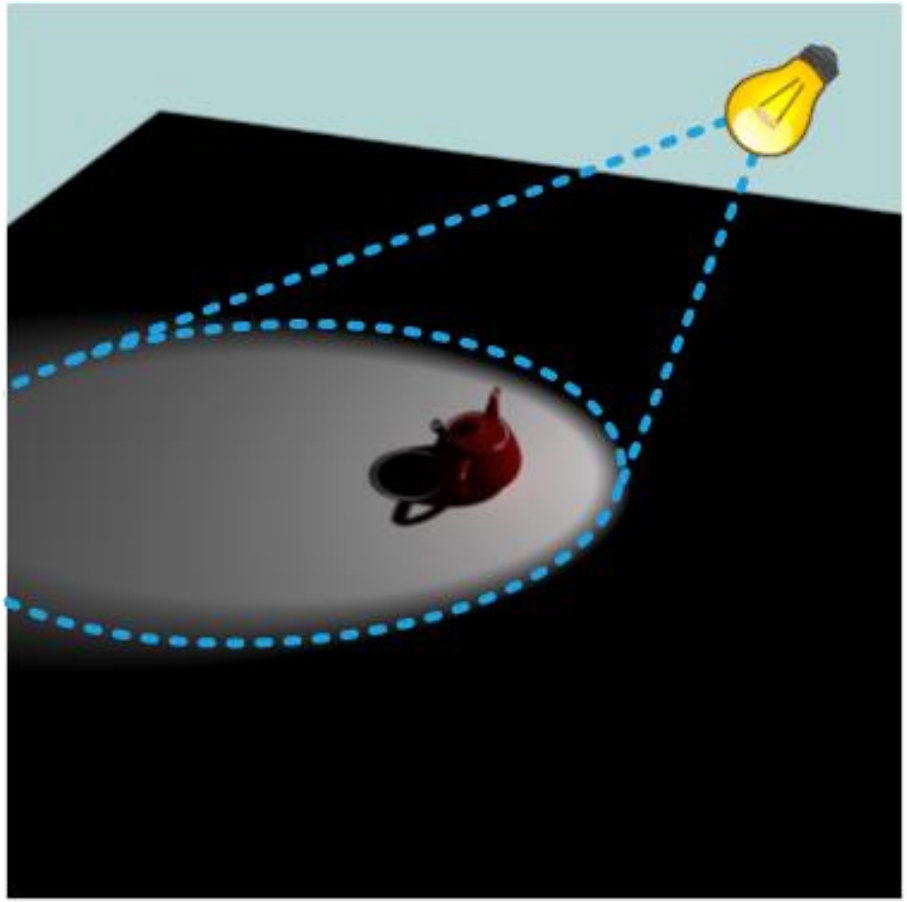
Pass I: rendering a depth map from the light position

Record the closest surface from the light and generate a **shadow map**

Shadow Map Overview (cont.)



Shadow Map Overview (cont.)



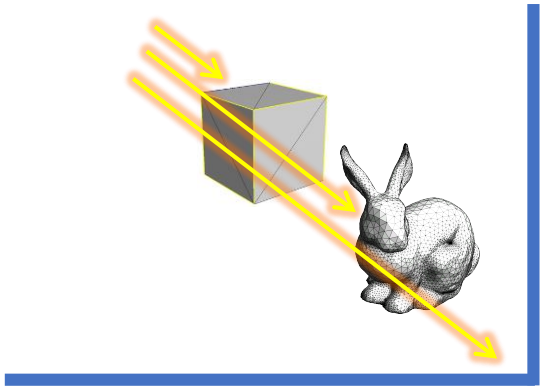
final rendering
(rendering from the camera view)

shadow map
(rendering from the light view)

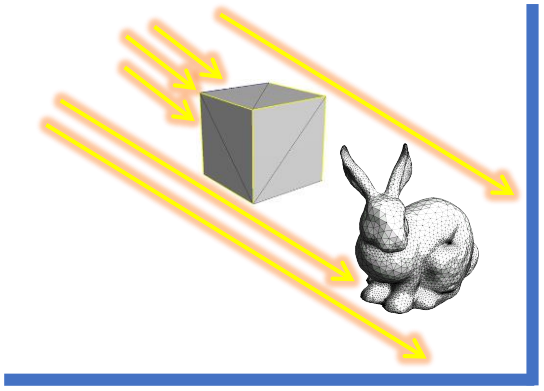
Ambient Occlusion

Recap: Global Illumination

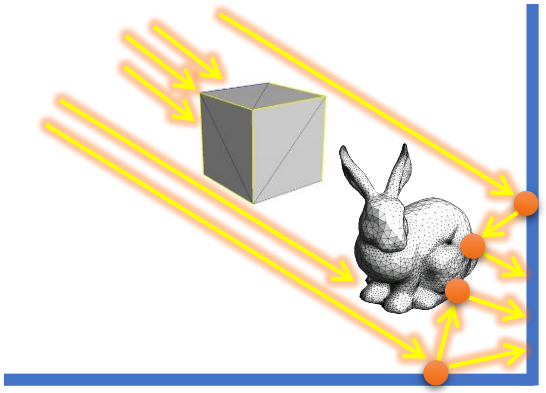
- Global illumination includes multi-bounce lighting
- Very expensive to compute
- In Phong lighting model, a **constant ambient term** is used to account for disregarded illumination
  - However, this produces a “flat”, “non-photo-realistic” appearance



local illumination



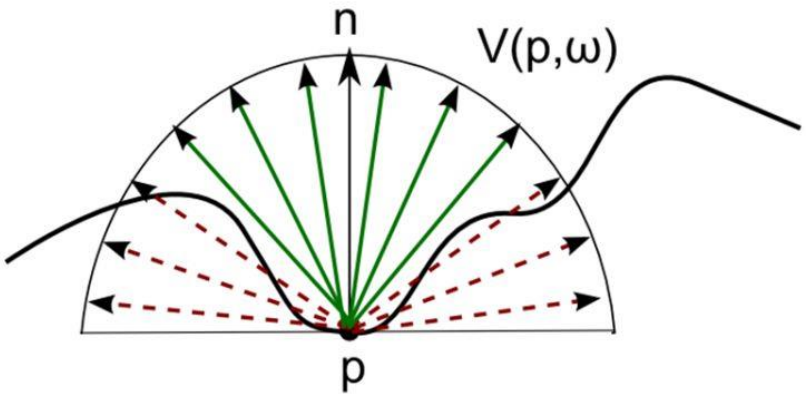
direct illumination



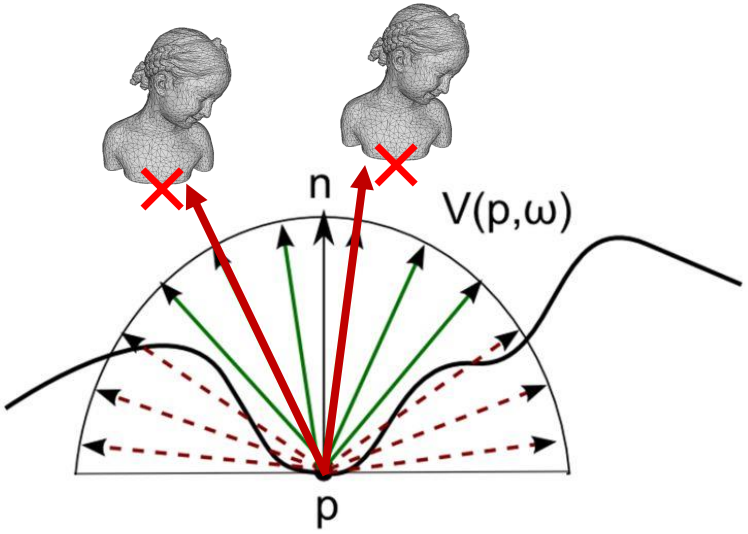
global illumination

Ambient Occlusion (cont.)

- **Ambient occlusion (AO)** is a popular technique to approximate global illumination
  - Modulate ambient light by the surface's **accessibility**
  - Greatly enhance **depth perception** with a relatively low cost



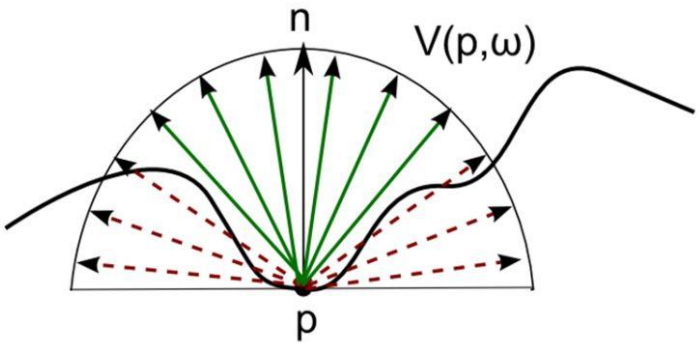
$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\,d\omega,$$



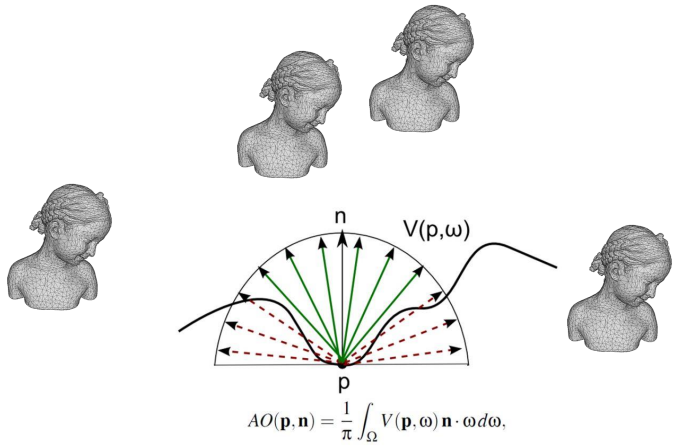
$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\,d\omega,$$

Ambient Occlusion (cont.)

- To compute AO, you need to know whether the ambient light is occluded in a direction
- In ray tracing, you can **trace rays** to determine the visibility
- For rasterization; however, this is difficult because **each polygon only knows its information** (again!)
  - Performance is also an issue!



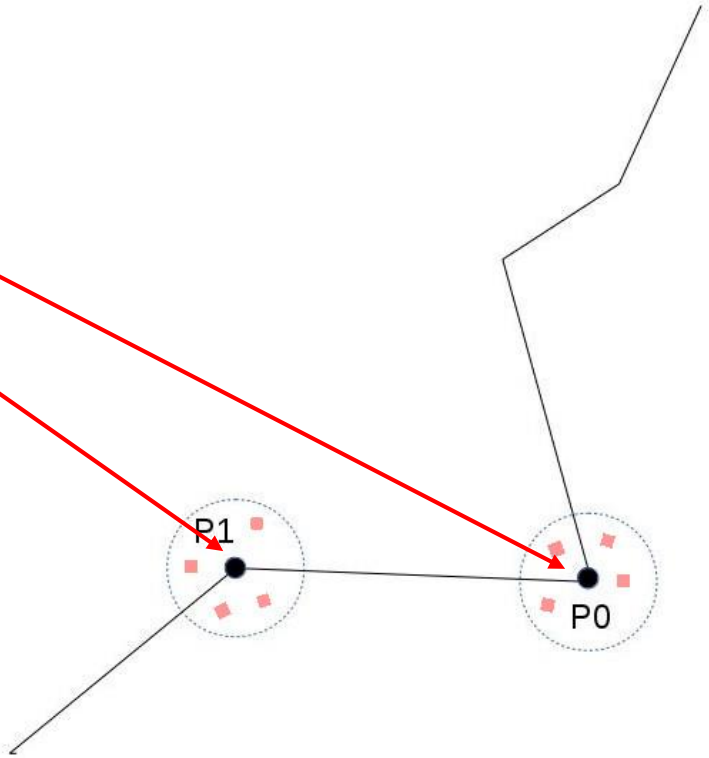
$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$



$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$

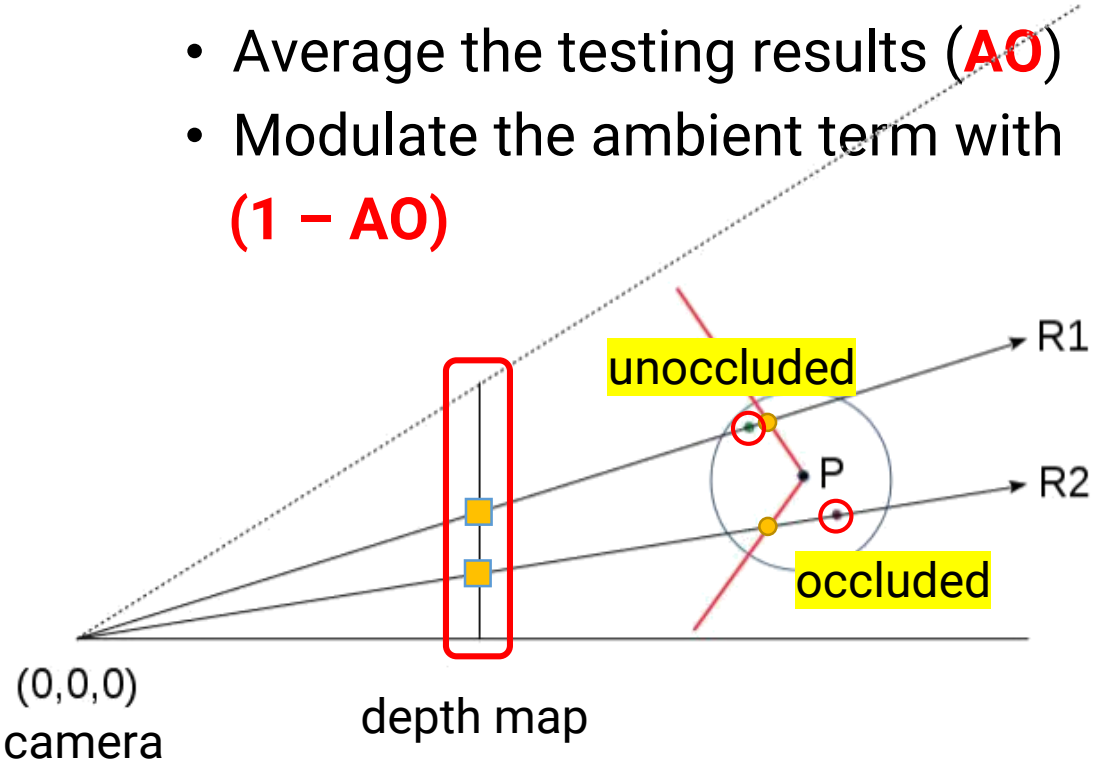
Screen-space Ambient Occlusion (cont.)

- Method
  - Generate samples within a sphere around the shading point (fragment)



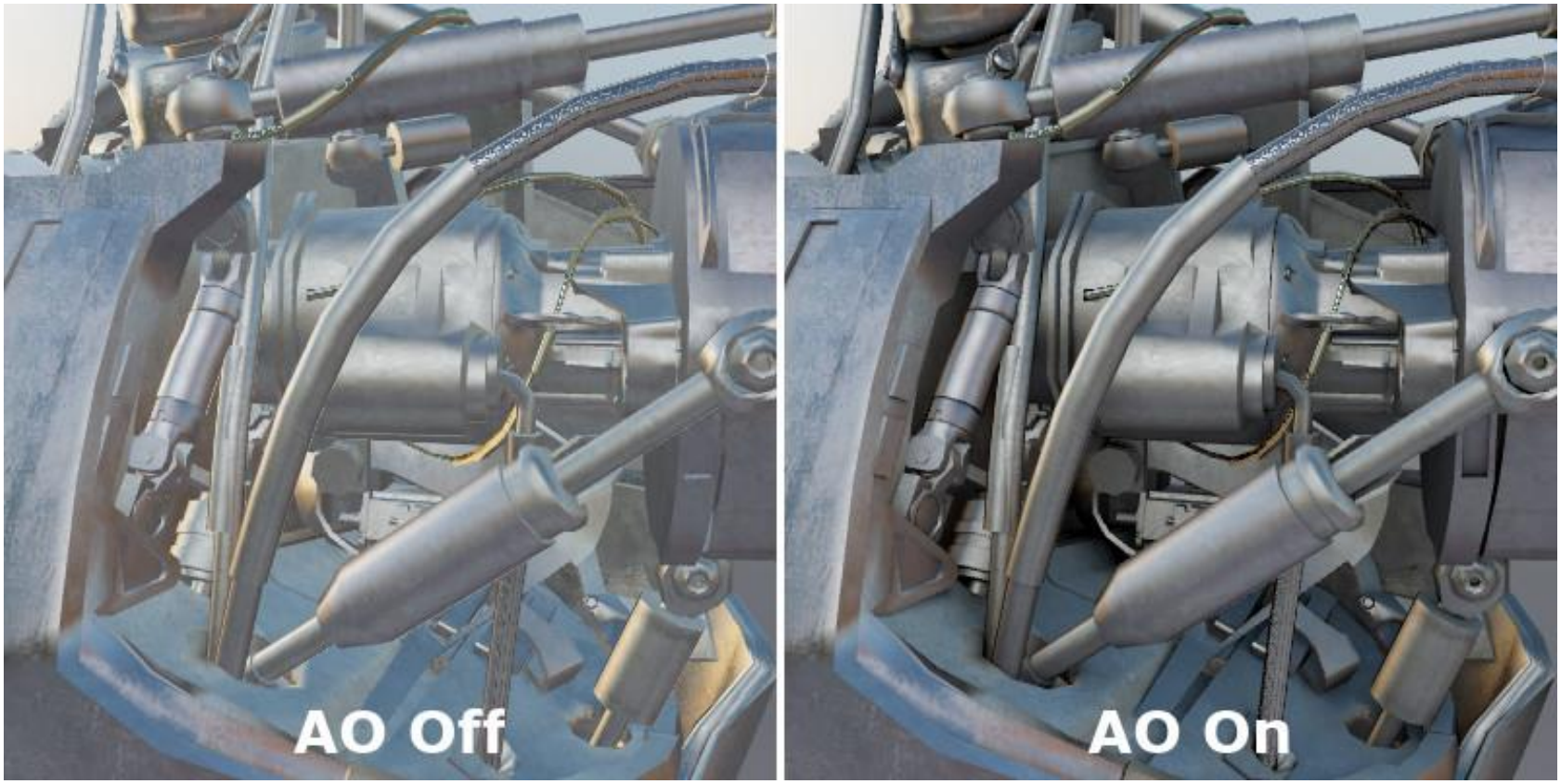
Screen-space Ambient Occlusion (cont.)

- Method
  - Project the samples back to the depth map from the camera
  - Compare the depth values
  - Average the testing results (**AO**)
  - Modulate the ambient term with **(1 – AO)**



Exampled Result

- Ambient occlusion (AO) is a popular technique to approximate global illumination



Global Illumination

Global Illumination

global illumination =

direct illumination **+** **indirect illumination**

local illumination + shadow map

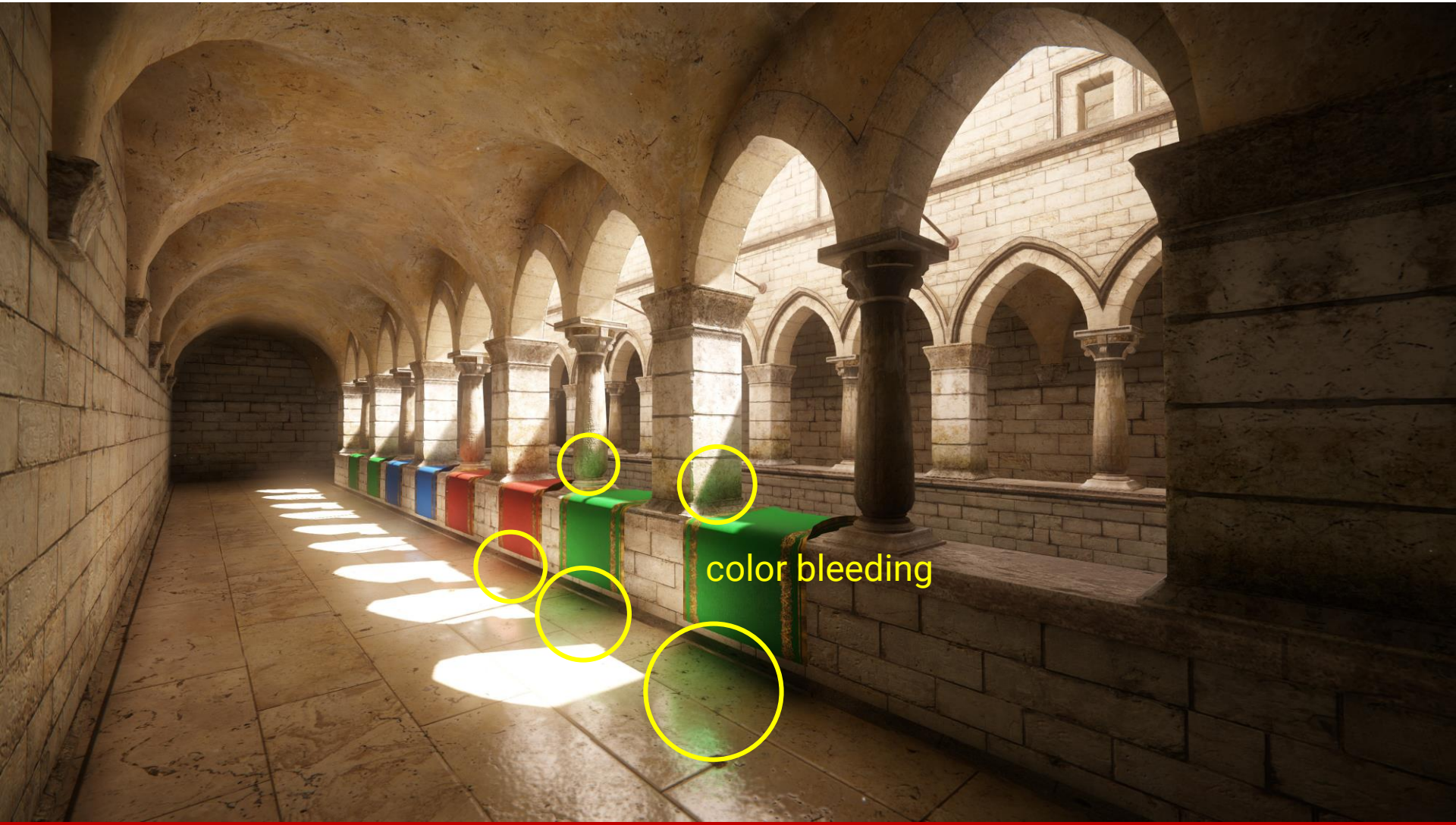
difficult

constant ambient term

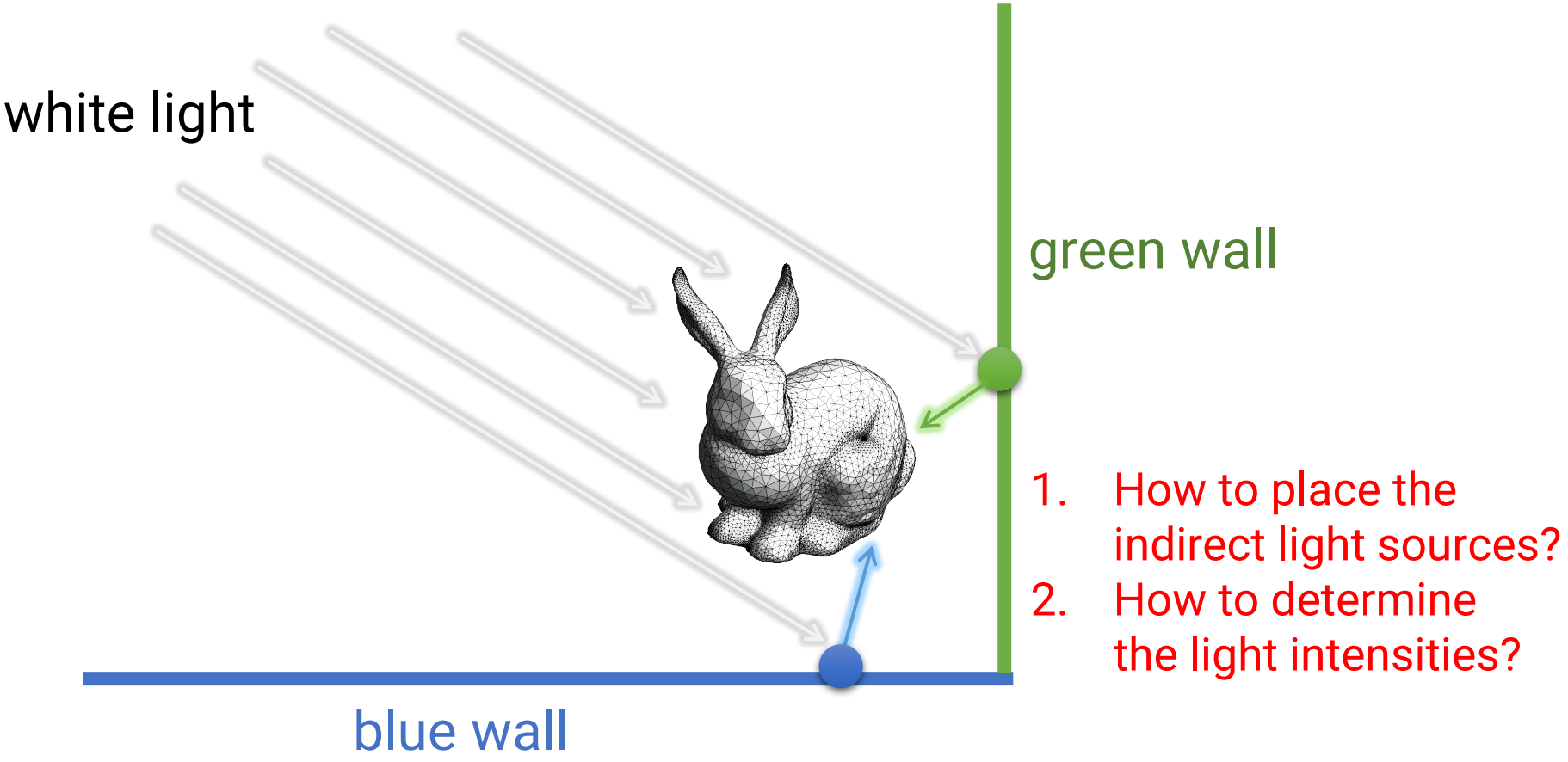
ambient occlusion

both are not good enough

Global Illumination (cont.)



Global Illumination (cont.)



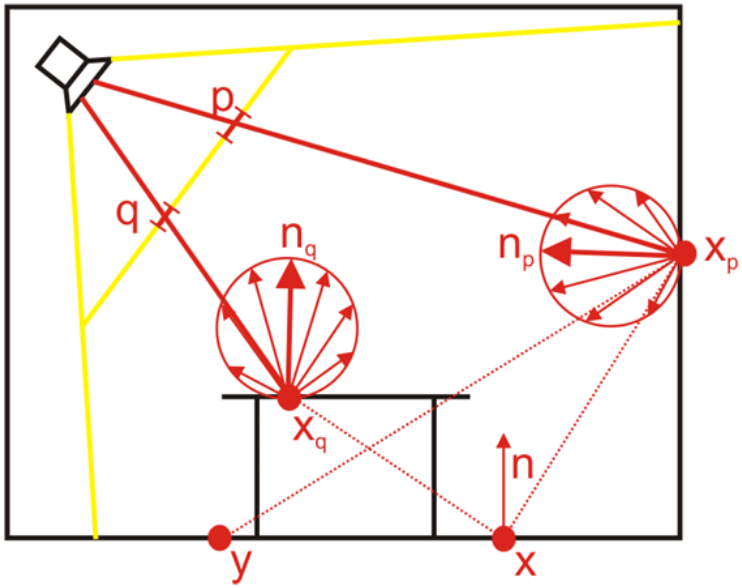
Global Illumination (cont.)

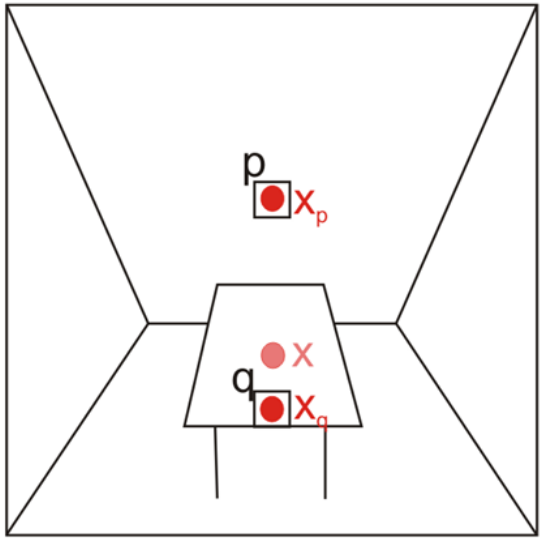
- Indirect illumination is especially difficult for rasterization because …
  - **Each polygon only has its own information**
  - **It does not know which triangle will cast lighting on it**

- In the last two decades, hundreds of research papers focus on this topic to approximate visually-pleasing global illumination in real-time

Reflective Shadow Map

- Proposed by Dachsbacher and Stamminger, I3D 2005
- **Major idea**
  - The closest surfaces from the light can receive the lighting contribution
  - They become the indirect light sources

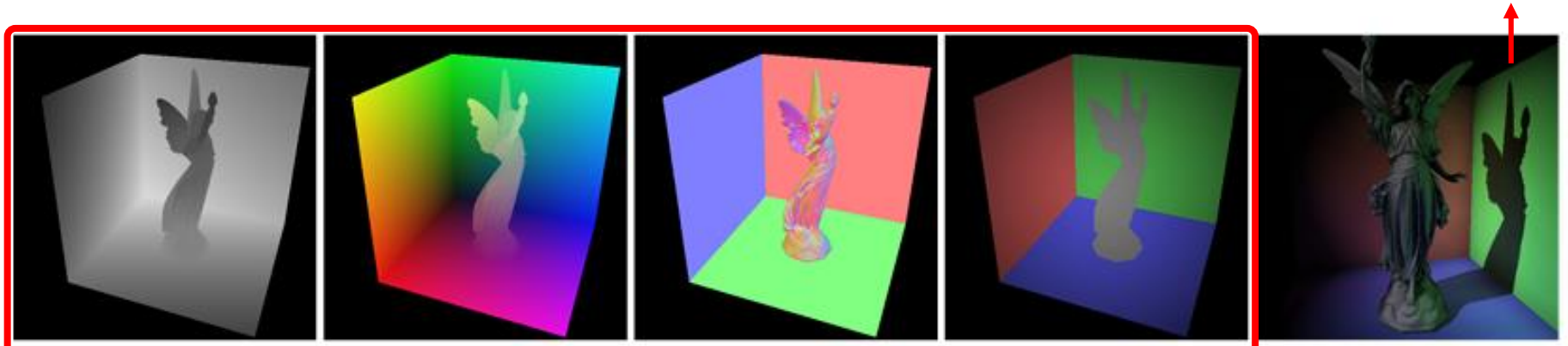




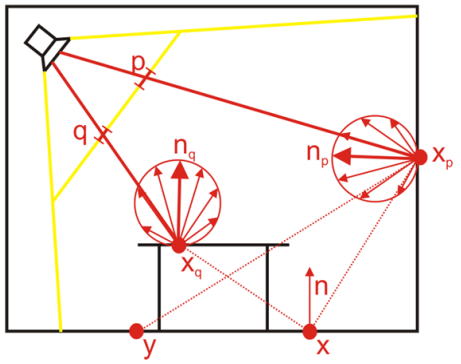
Reflective Shadow Map (cont.)

- Two-pass rendering algorithm

Pass II: render from the camera view



Pass I: render G-buffer from a light view (called RSM)



Reflective Shadow Map (cont.)

- **Pass I:** rendering G-buffer (called **RSM**) from the light view for generating indirect light sources
  - World-space position
  - World-space normal
  - Reflected flux
    - The intensity of the primary light source multiplied by the reflectance of the surface
- **Pass II:** rendering from the camera view
  - Direct lighting is computed by local illumination and shadow mapping
  - Indirect lighting is estimated from the RSM

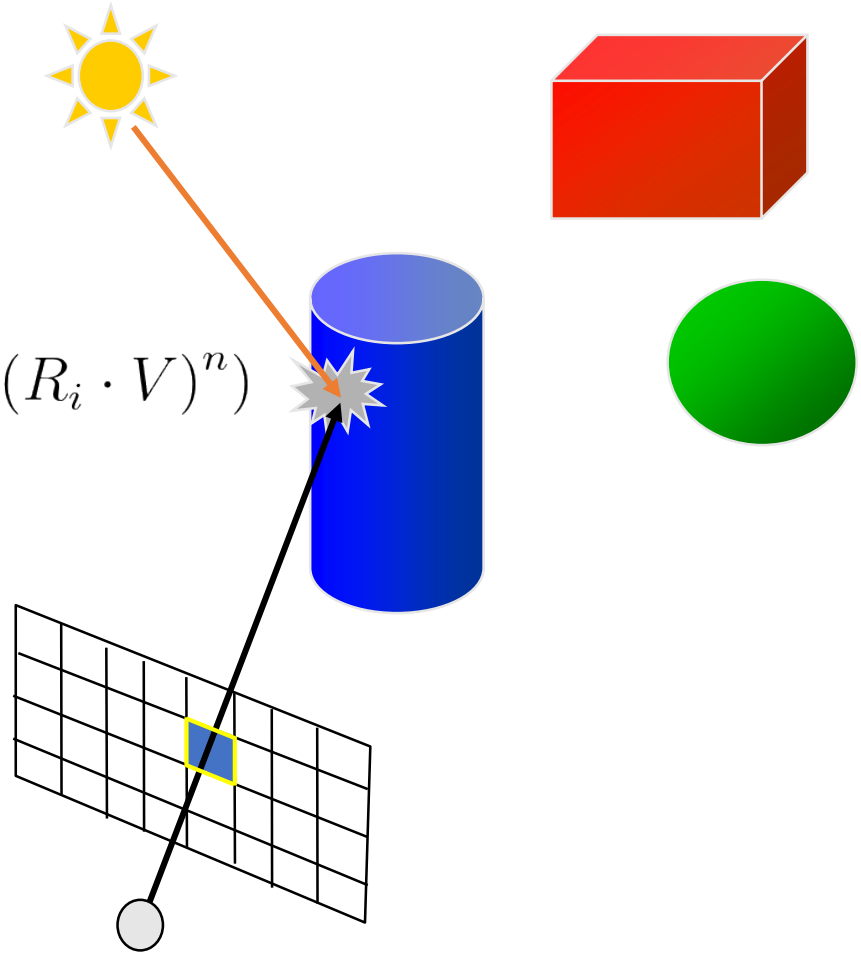
Reflective Shadow Map (cont.)

Ray Tracing

Ray Casting

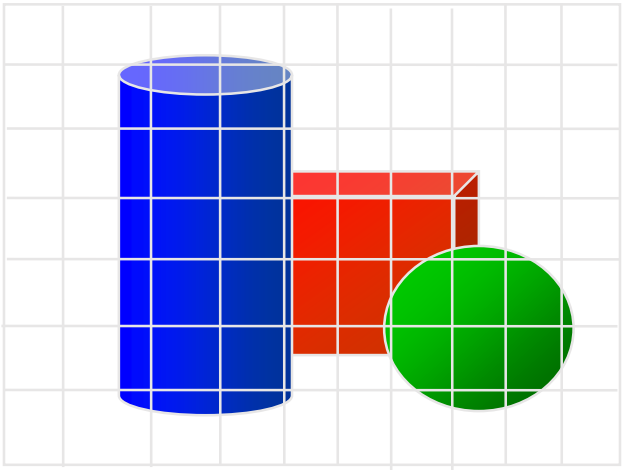
- Proposed by Appel [1968]

$$K_a I_a + \sum_{i=1}^{nls} I_i \left(K_d \left(L_i \cdot N \right) + K_s \left(R_i \cdot V \right)^n \right)$$

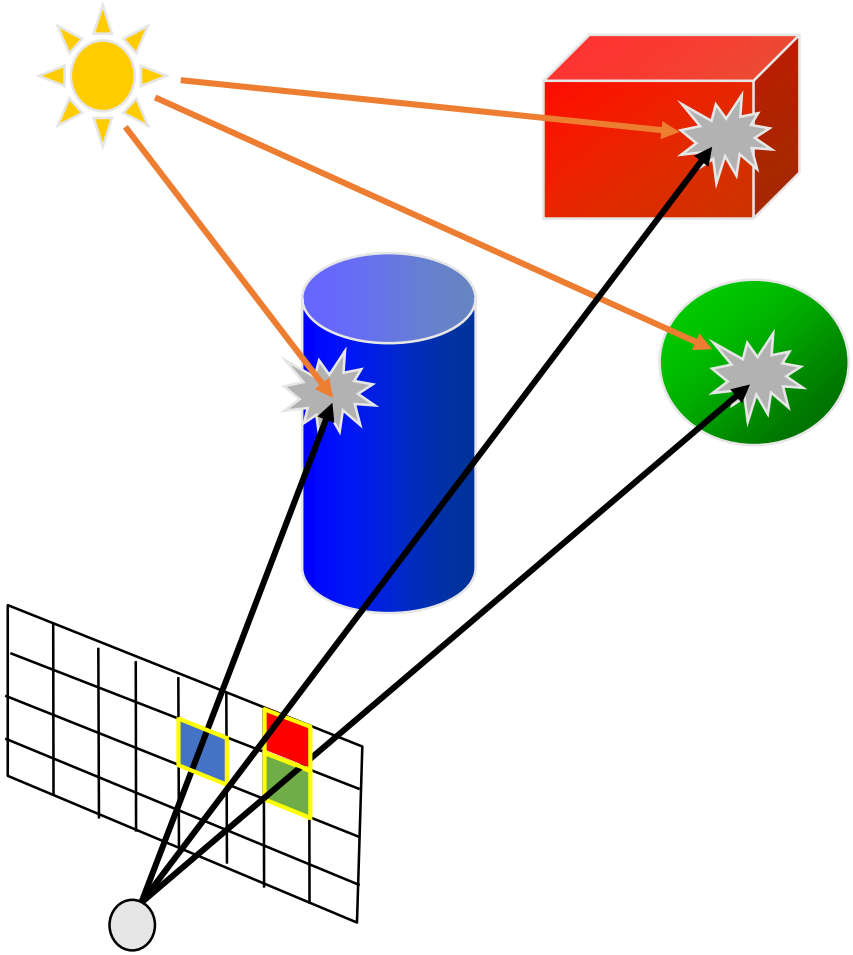


Ray Casting (cont.)

- Proposed by Appel [1968]

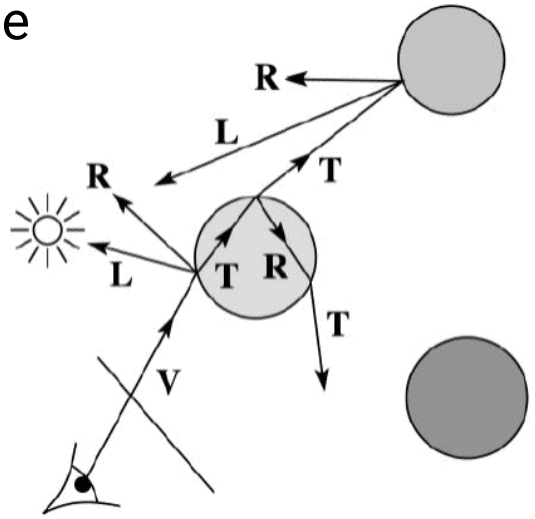


local illumination

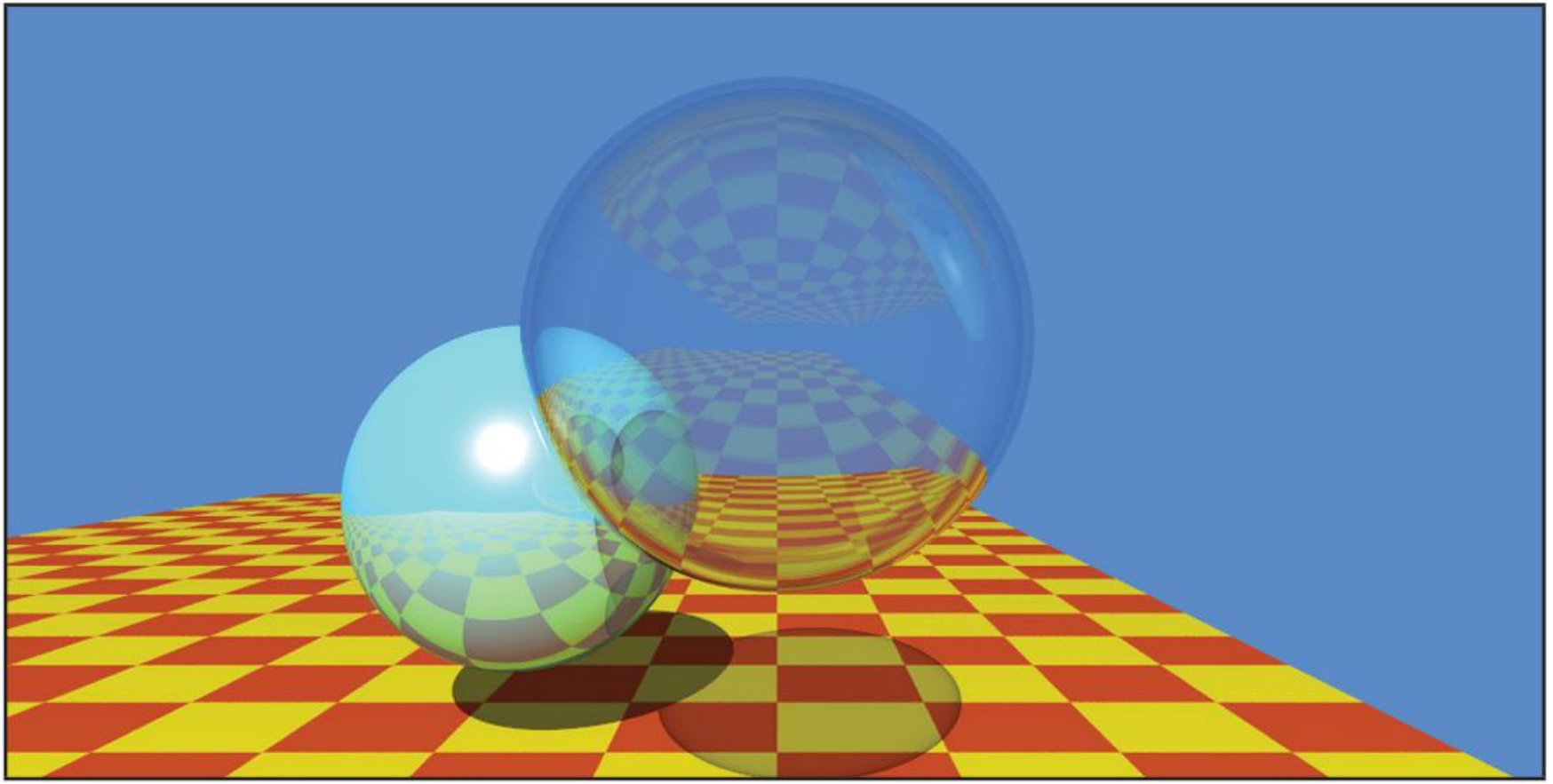


Whitted Ray Tracing

- Proposed by Whitted, 1980
- **Recursive** trace rays for **shadows**, perfect **specular** (e.g., mirror), and perfect **transparent** (e.g., glass) objects
  - For each pixel, trace a primary ray in the direction ***V*** to the first visible surface
  - For each intersection, trace secondary rays including
    - Shadow rays (***L***) to each light source
    - Reflected ray (***R***)
    - Refracted ray (***T***)

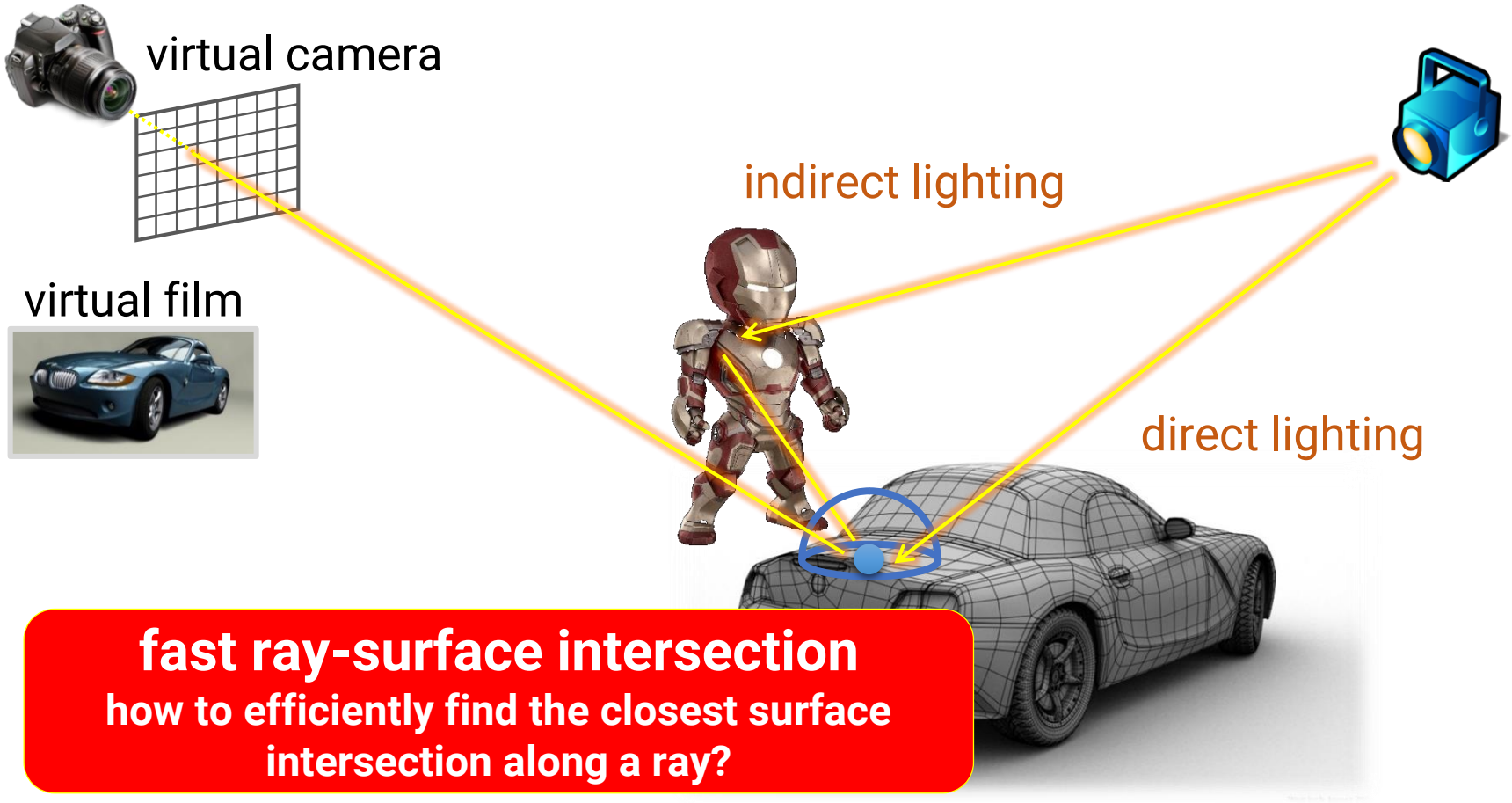


Whitted Ray Tracing (cont.)



Key: Fast Ray-Surface Intersection

- A united approach for different light transport paths



fast ray-surface intersection
how to efficiently find the closest surface intersection along a ray?

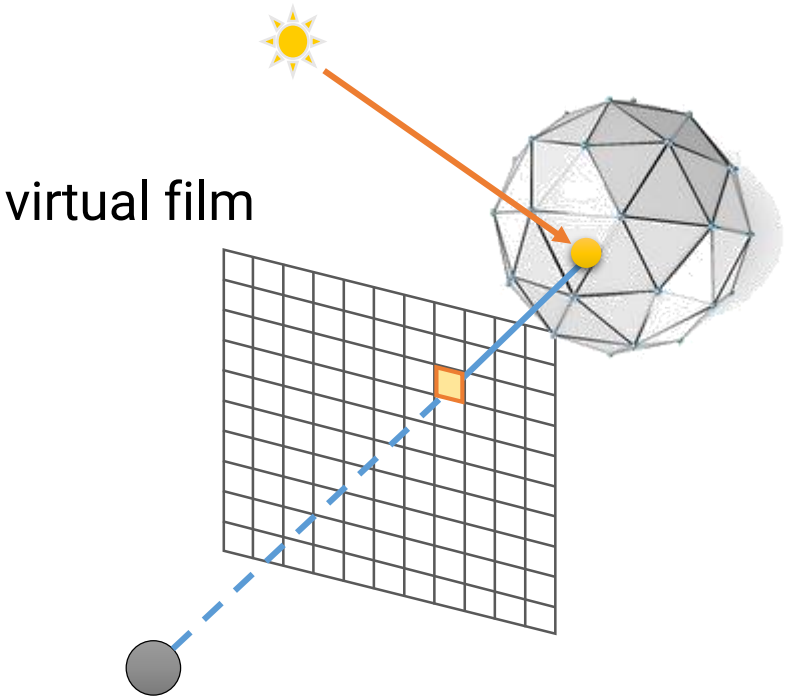
Acceleration Structure

- **Reduce the required number of ray-surface intersection**
- Common acceleration structures
  - Bounding volume hierarchy (BVH)
  - Space subdivision
    - Uniform grid
    - Octree tree
    - Kd tree

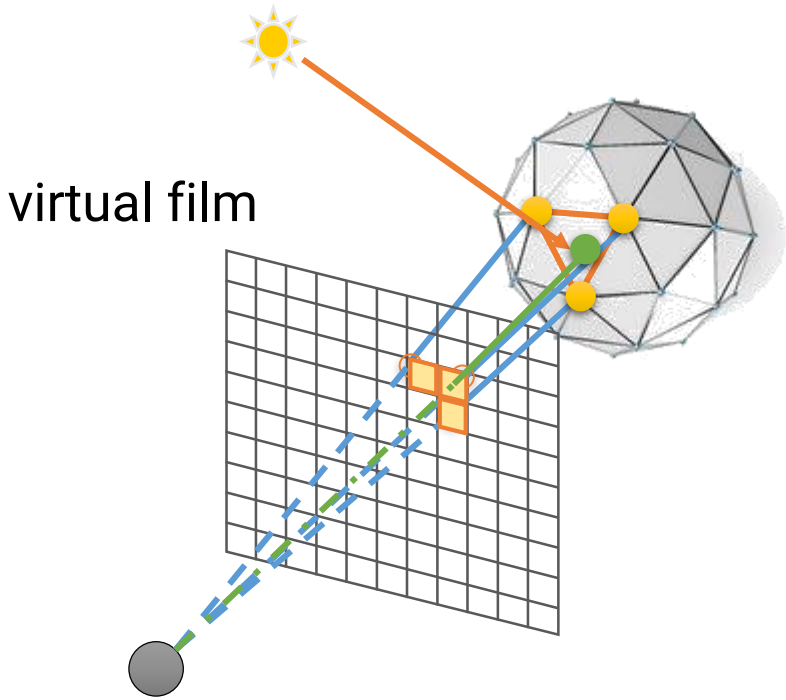
- Please go the slides for details and examples

Ray Tracing v.s. Rasterization

Ray tracing

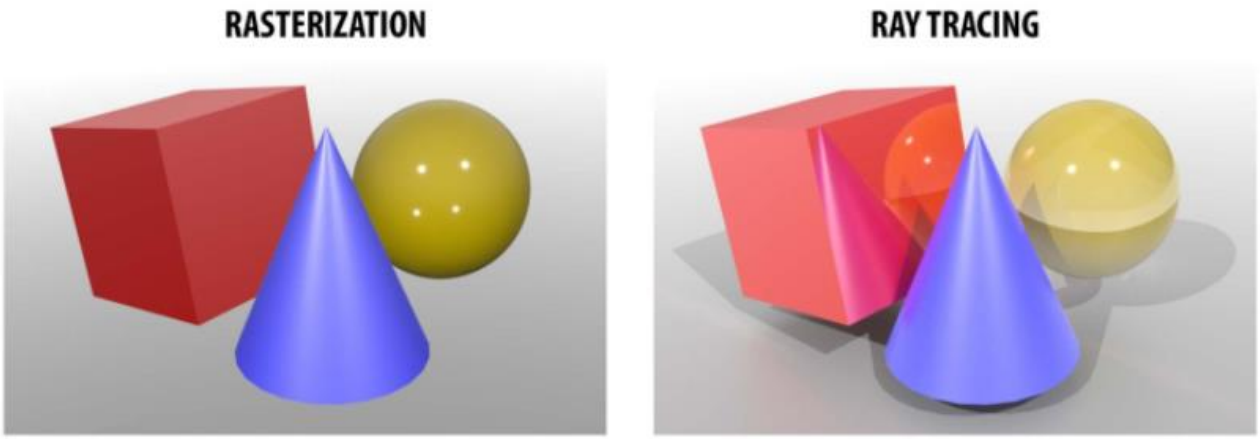


Rasterization



Ray Tracing v.s. Rasterization (cont.)

- Rasterization is **more friendly to hardware** and usually has higher parallelism
- But when we need to **interact with other triangles**, it is much more difficult to simulate effects such as reflection, refraction, shadows, and global illumination
  - Need specialized algorithms

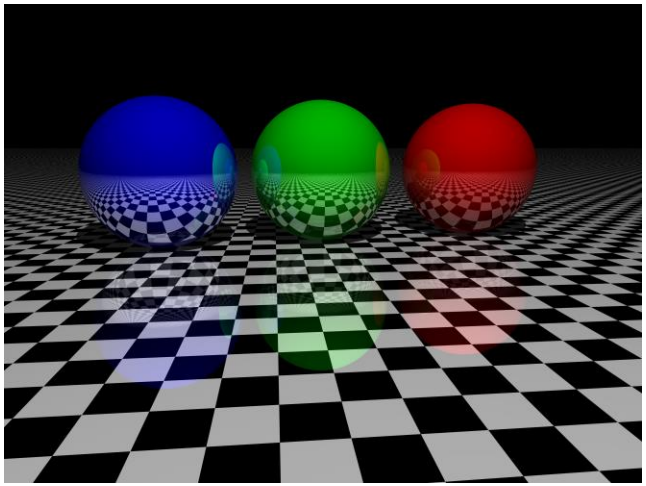


Ray Tracing v.s. Rasterization (cont.)

- **Transparency**
  - **Rasterization**
    - Render the object in order (distant objects first) and blend with the previous result in the color buffer
  - **Ray-tracing:**
    - Trace a secondary (refracted) ray through the object's surface

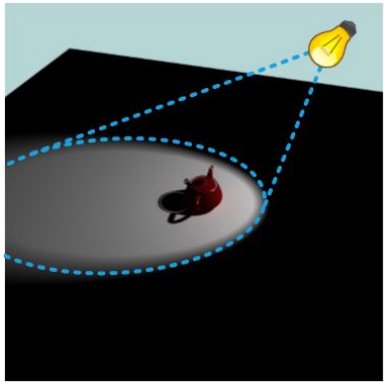
Ray Tracing v.s. Rasterization (cont.)

- **Reflection**
  - **Rasterization**
    - Render the scene into an environment map
    - Look up the environment map in the fragment shader
  - **Ray-tracing:**
    - Trace a secondary (reflected) ray from the object's surface



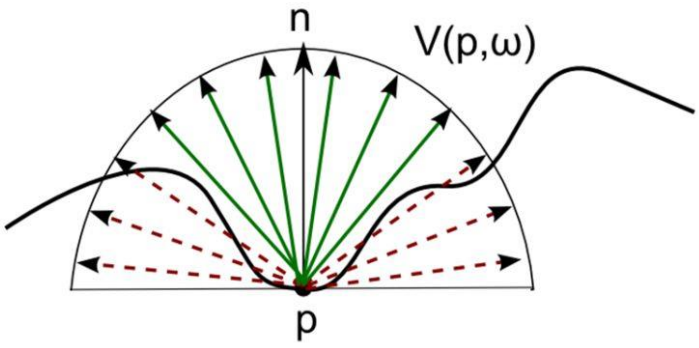
Ray Tracing v.s. Rasterization (cont.)

- **Shadow**
  - **Rasterization (shadow map as an example)**
    - Render a **shadow map** to record the closest surface from each light
    - Look up the map to determine whether a surface point is in shadow or not in the second pass
  - **Ray-tracing**
    - Trace a shadow ray to see if the lighting direction is occluded

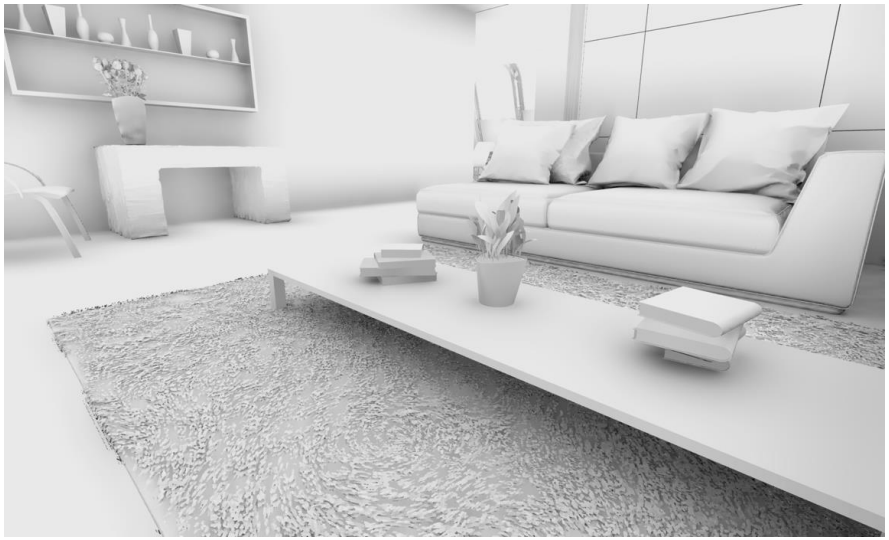


Ray Tracing v.s. Rasterization (cont.)

- **Ambient occlusion**
  - **Rasterization (SSAO as an example)**
    - Use the **depth map** to find nearby occluders in screen space
  - **Ray-tracing**
    - Trace shadow rays to see if a direction is occluded

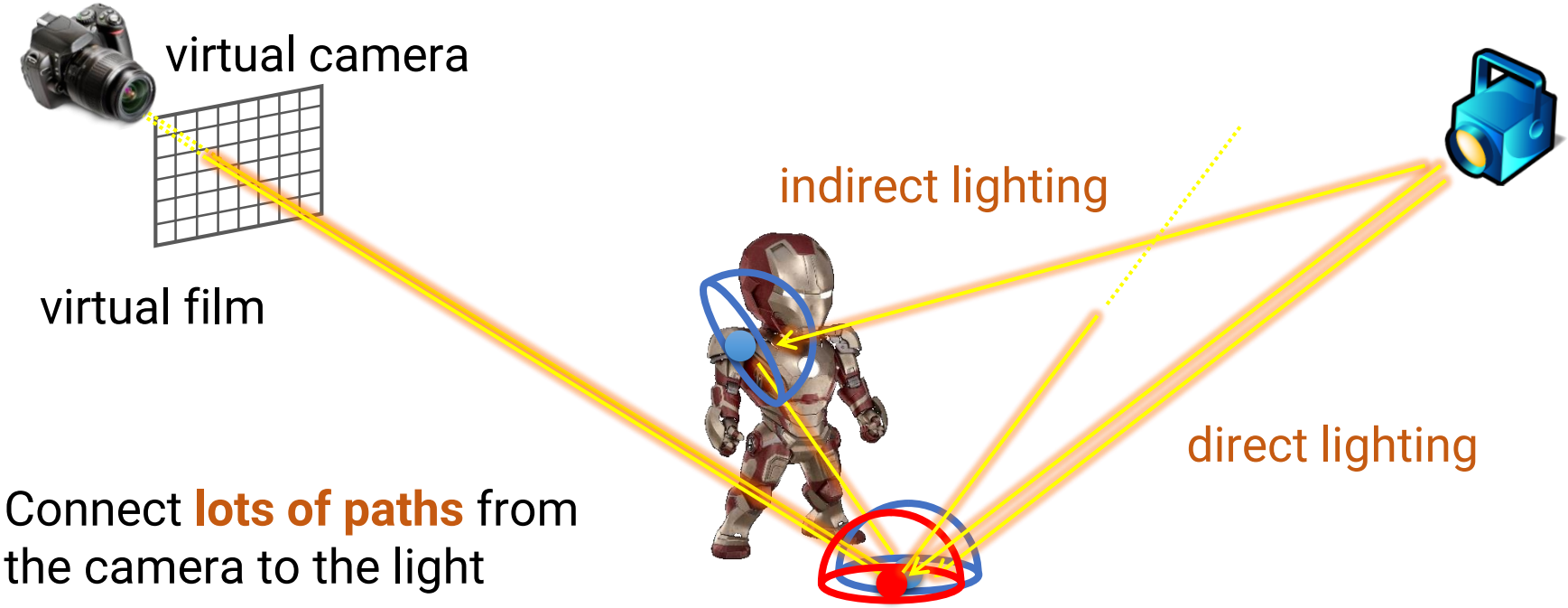


$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$



Ray Tracing v.s. Rasterization (cont.)

- **Global illumination**
  - **Rasterization (RSM as an example)**
    - Render the direct lighting result from the light view
    - Use the results in the first pass to render indirect lighting



Connect **lots of paths** from the camera to the light

Ray Tracing v.s. Rasterization

- **Problems with ray tracing**
  - Ray tracing is more general
  - However, its simulator usually has a slow convergence rate and produces lots of noise when samples are not enough
- **Solution**
  - More rays (expensive)
  - Filtering (bias)

Any Questions?

Good Luck!