EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

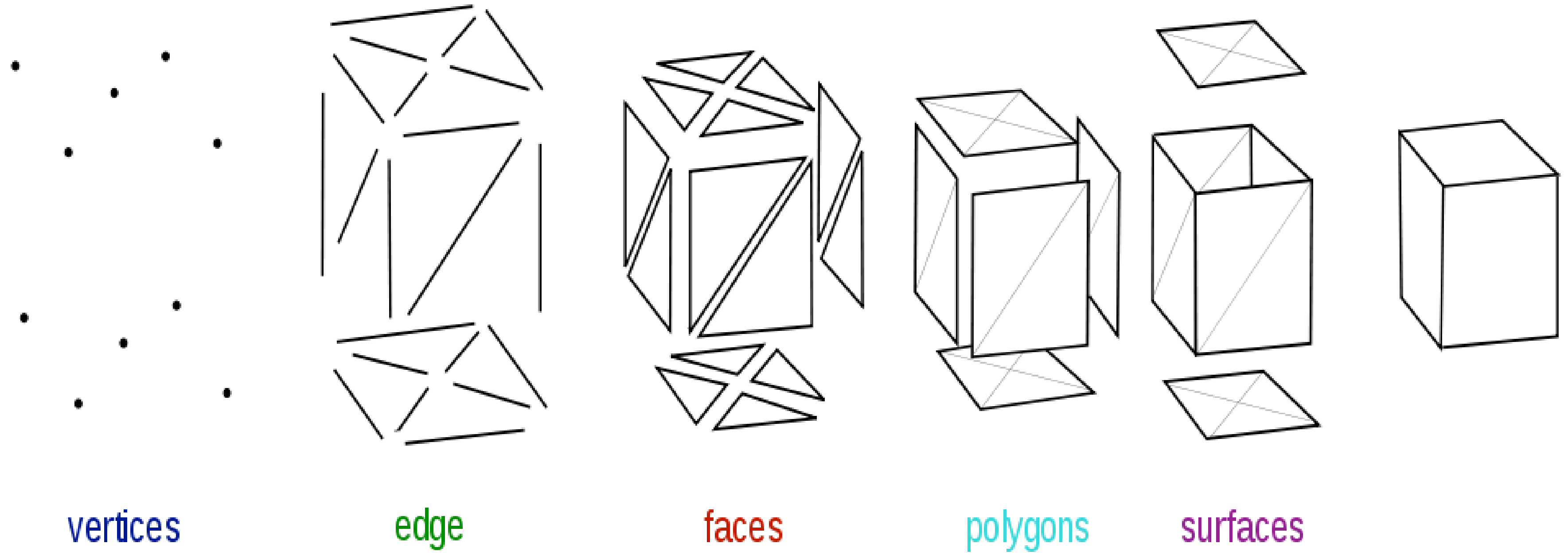
Computer Graphics – Part 2

Scientific Visualization – Summer Semester 2021

Jun.-Prof. Dr. **Michael Krone**

Triangle Meshes

- Describe the surface (boundary) of an object as a set of polygons
  - Mostly use triangles, since they are trivially convex and flat
- Current graphics hardware is optimized for triangle meshes

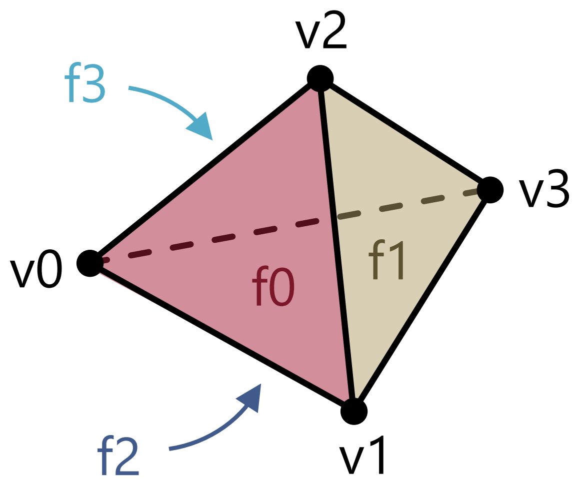


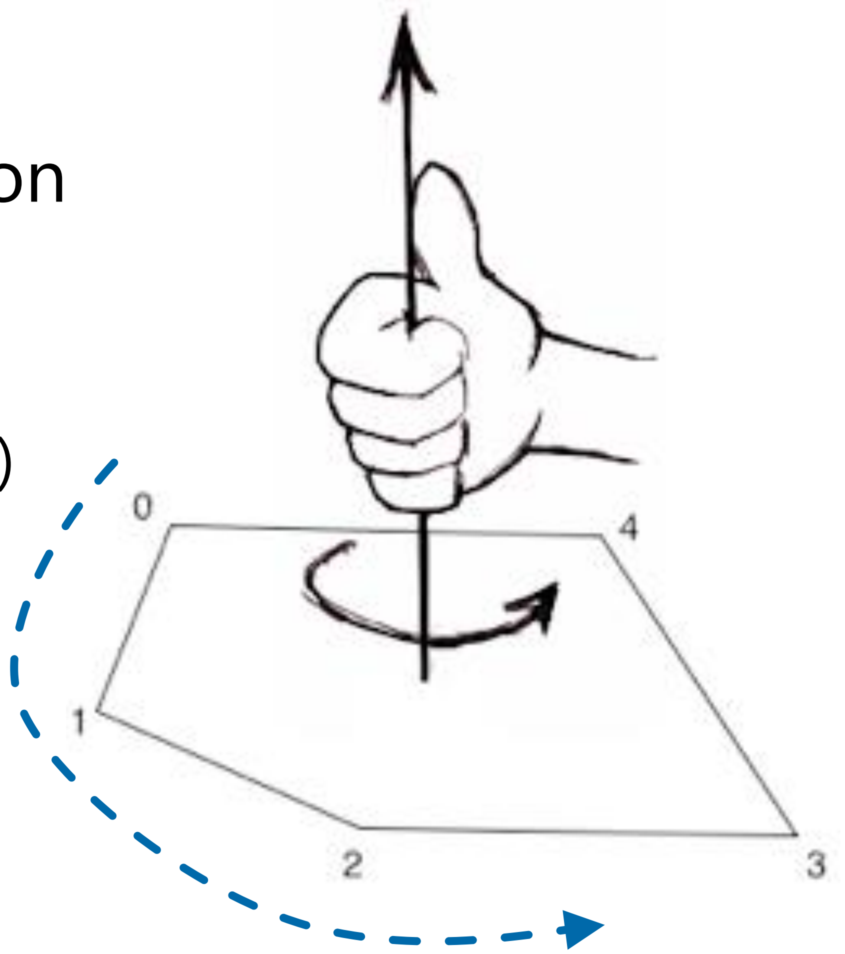
http://en.wikipedia.org/wiki/File:Mesh_overview.svg

Right Hand Rule for Polygons

- A “rule of thumb” to determine the front side (= direction of the normal vector) for a polygon
- Please note: The relationship between vertex order and normal vector is just a convention!
    - Can be defined in OpenGL (clockwise/counter-cw)

Face List	
f0	v0 v1 v2
f1	v1 v3 v2
f2	v0 v3 v1
f3	v0 v2 v3

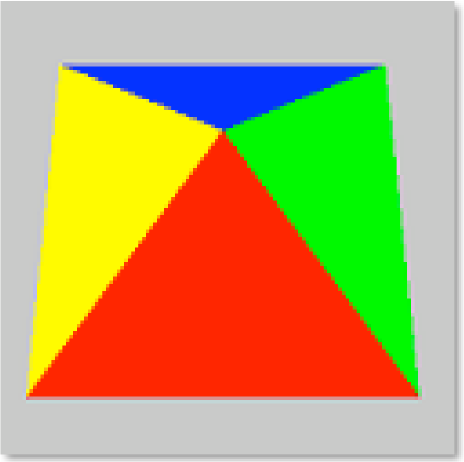


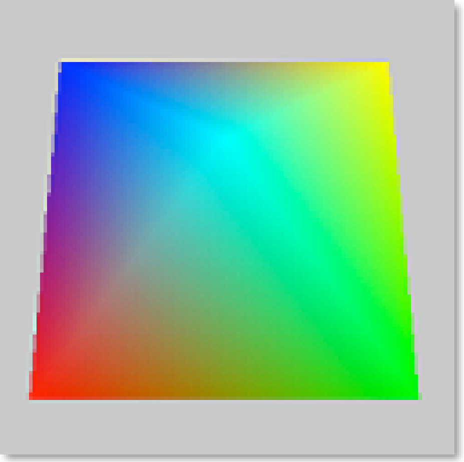


http://www.csse.monash.edu.au/~cema

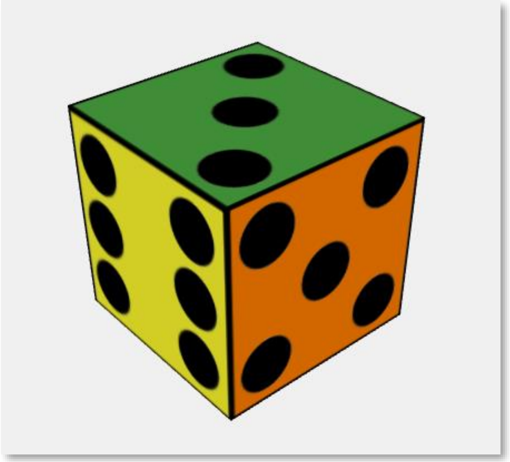
Polygon Meshes: Optional Data

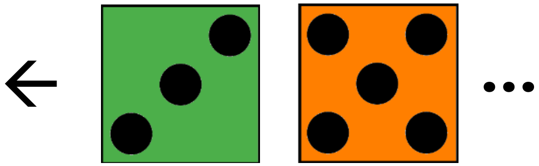
- Color per vertex or per face: produces colored models
- Normal per face:
  - Trivial to compute → cross product!
  - Easy access to front/back information
- Normal per vertex
  - Usually average of face normals
  - Allows free control over the normals
- Texture coordinates per vertex
  - Put images or parts of an image onto the polygons





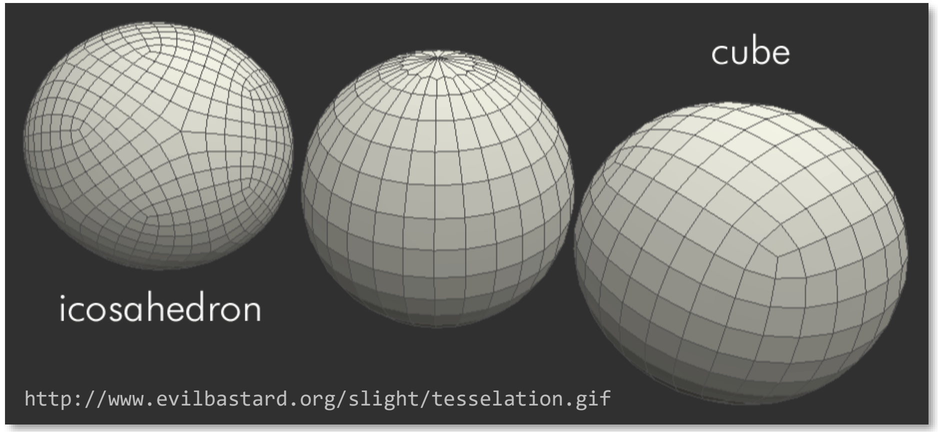
http://en.wikipedia.org/wiki/File:Triangle_Strip.png

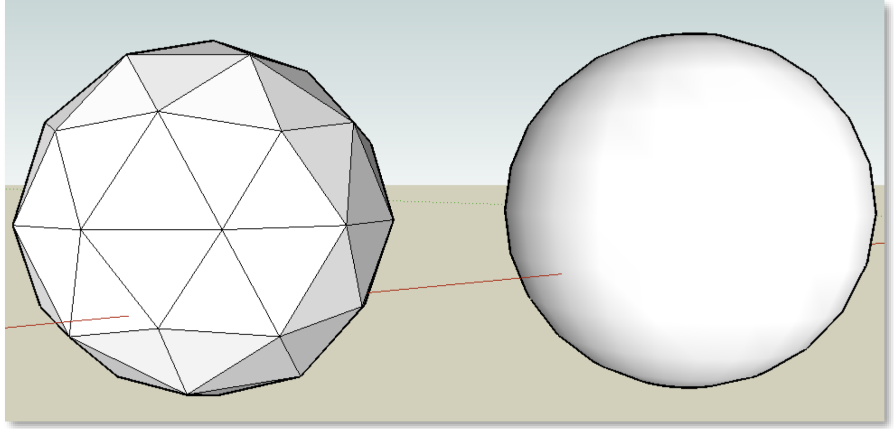




Approximating Primitives by Polygon Meshes

- Trivial for non-curved primitives…
- The curved surface of a cylinder, sphere etc. must be represented by polygons somehow (Tessellation).
- ***Not trivial, only an approximation, and certainly not unique!***
- **Goal:** small polygons for strong curvature, larger ones for areas of weak curvature. ***Why?***

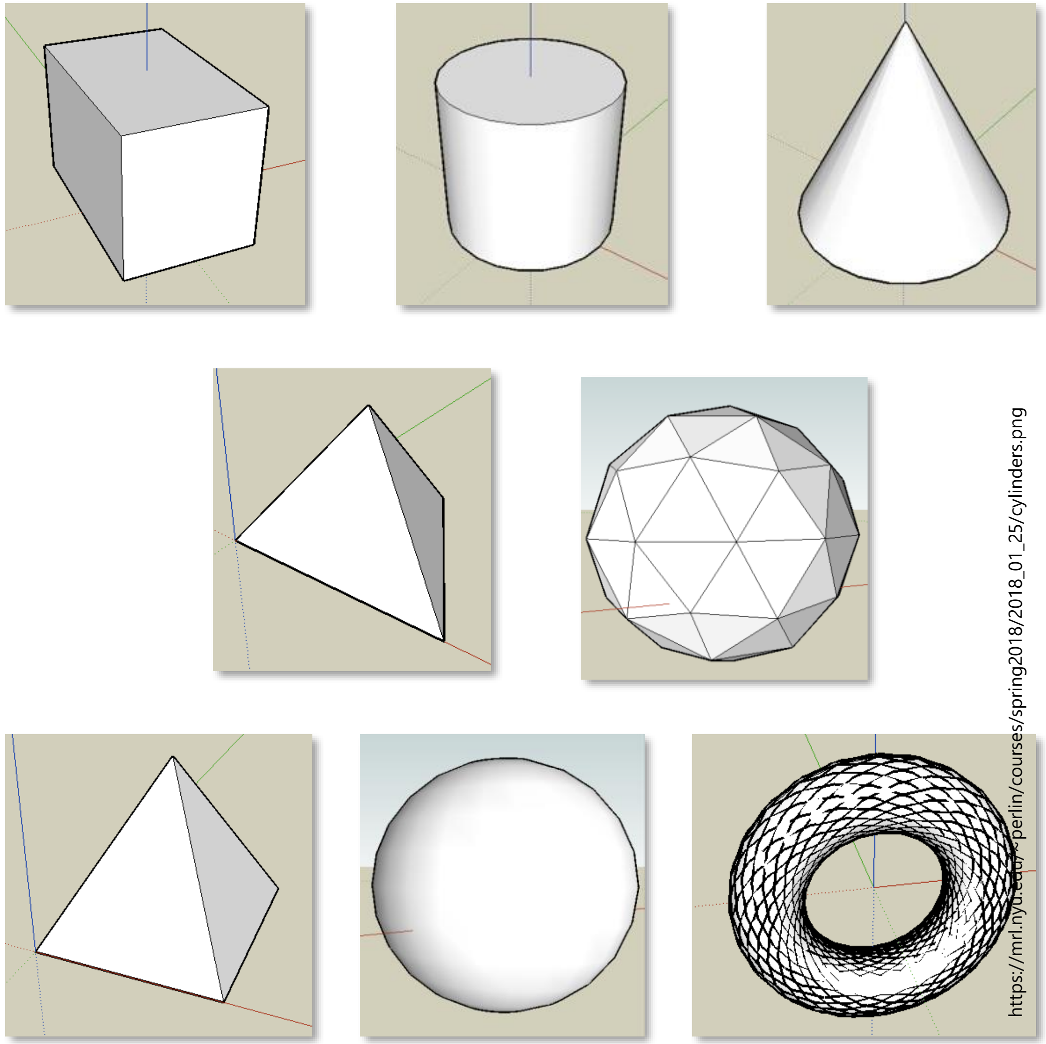




Pre-Tessellated Geometric Primitives in Three.js

- Box
- Cylinder, Cone
- Tetrahedron, Icosahedron,…
  (Platonic solids)
- Pyramid
- Sphere
- Torus („Doughnut/donut")
- …

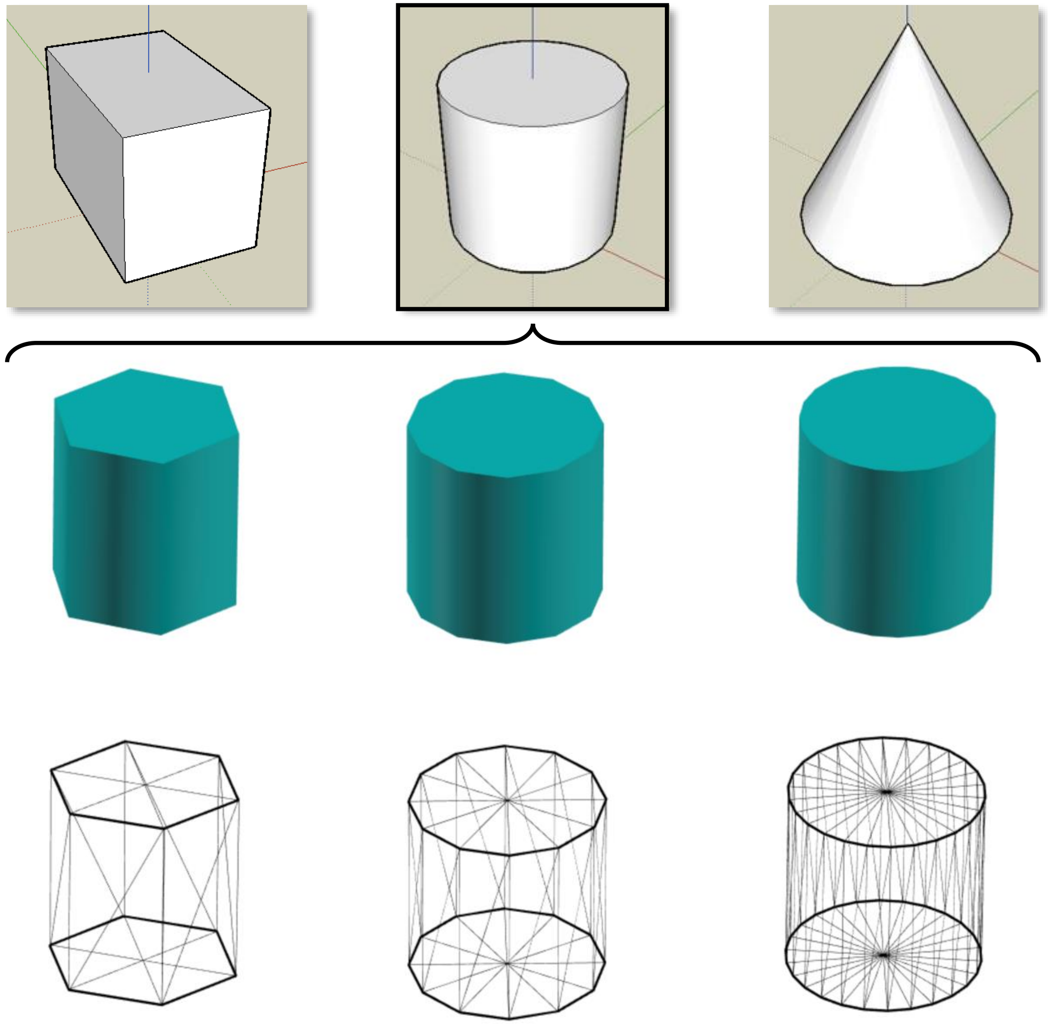
→Adjustable parameters (position, size, number of facets for curved shapes)



Pre-Tessellated Geometric Primitives in Three.js

- Box
- Cylinder, Cone
- Tetrahedron, Icosahedron,… (Platonic solids)
- Pyramid
- Sphere
- Torus („Doughnut/donut“)
- …

→Adjustable parameters (position, size, number of facets for curved shapes)

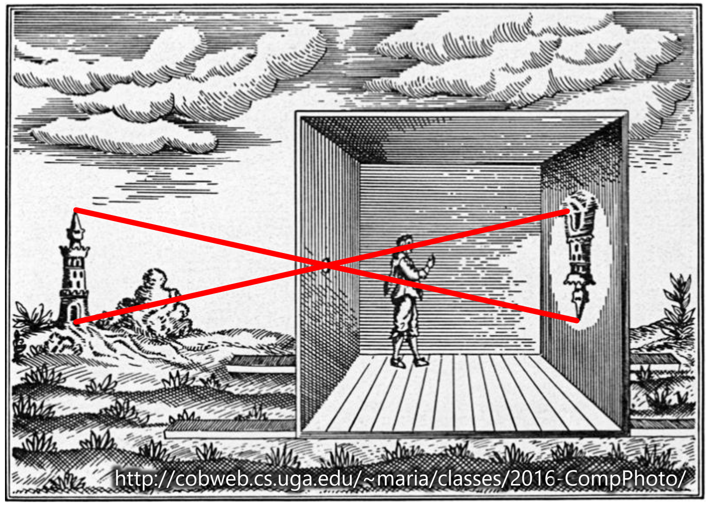


https://mrl.nyu.edu/~perlin/courses/spring2018/2018_01_25/cylinders.png

3D Scenes, Camera, and Projection

- Projections of 3D scenes are also common in art
- General situation:
  - Scene consisting of 3D objects
  - Viewer with defined position and projection surface
  - Projectors (“projection lines”): lines going from points on objects to the projection surface
- Main classification:
  - Parallel projectors or converging projectors
- Assumptions in CG:
  - Objects constructed from flat faces (triangles)
  - Projection surface is a flat plane

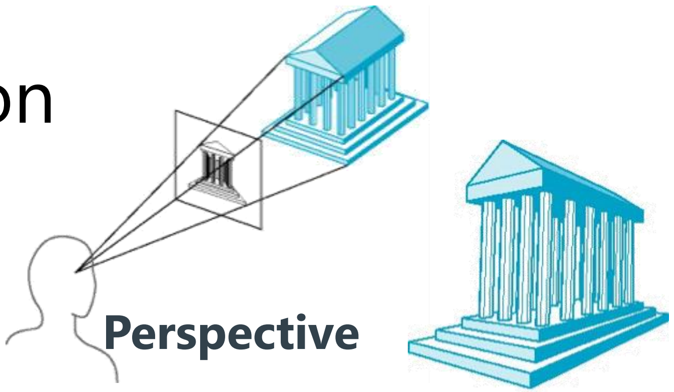
http://www.semioticon.com/seo/P/images/perspective_1.jpg

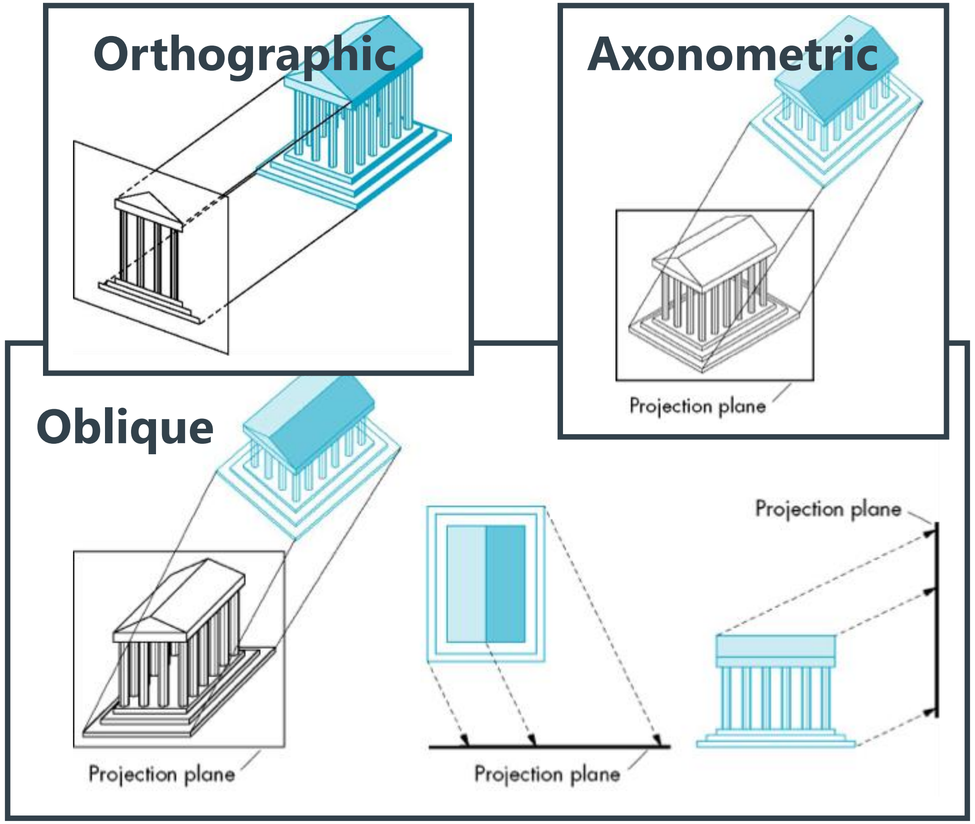


http://cobweb.cs.uga.edu/~maria/classes/2016-CompPhoto/

Projections

- Parallel projections
  - Orthographic projections
  - Axonometric projections (e.g., isometric)
  - Oblique Projection
- Perspective Projection



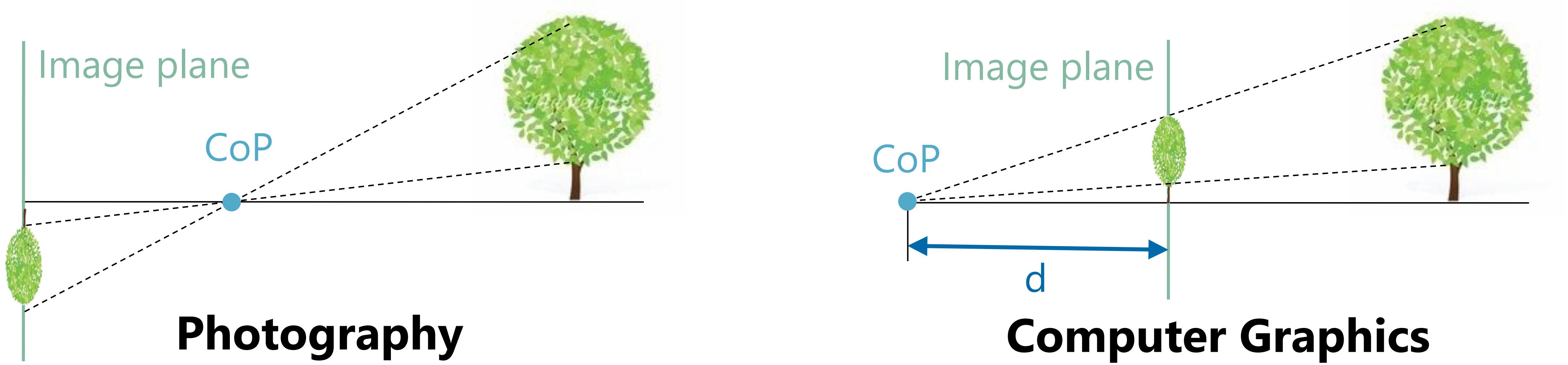


- How to realize projection in Three.js?

```
OrthographicCamera( left, right, top, bottom, near, far);
PerspectiveCamera( field of view (angle), aspect ratio, near, far);
```

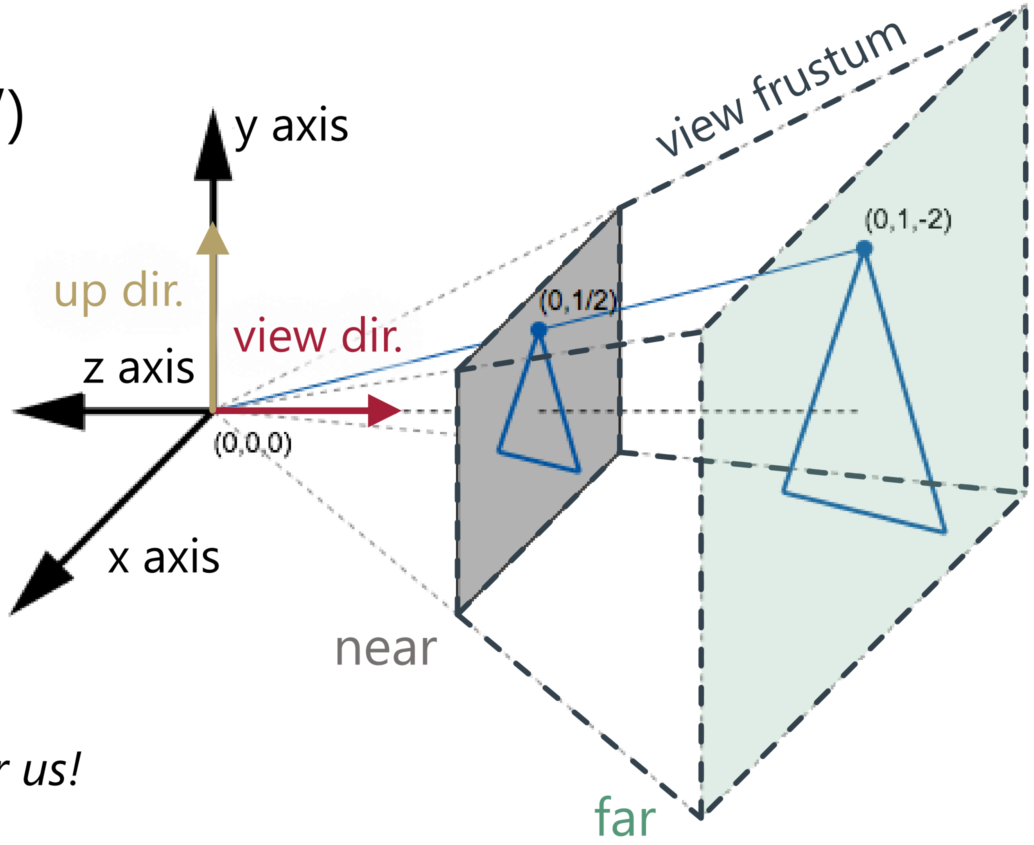
Perspective Projection and Photography

- In photography, the center of projection (CoP) is between the object and the image plane
  - Image on film/sensor is upside down
- In CG perspective projection, the image plane is in front of the CoP
  - Often called “camera position” or “eye position”



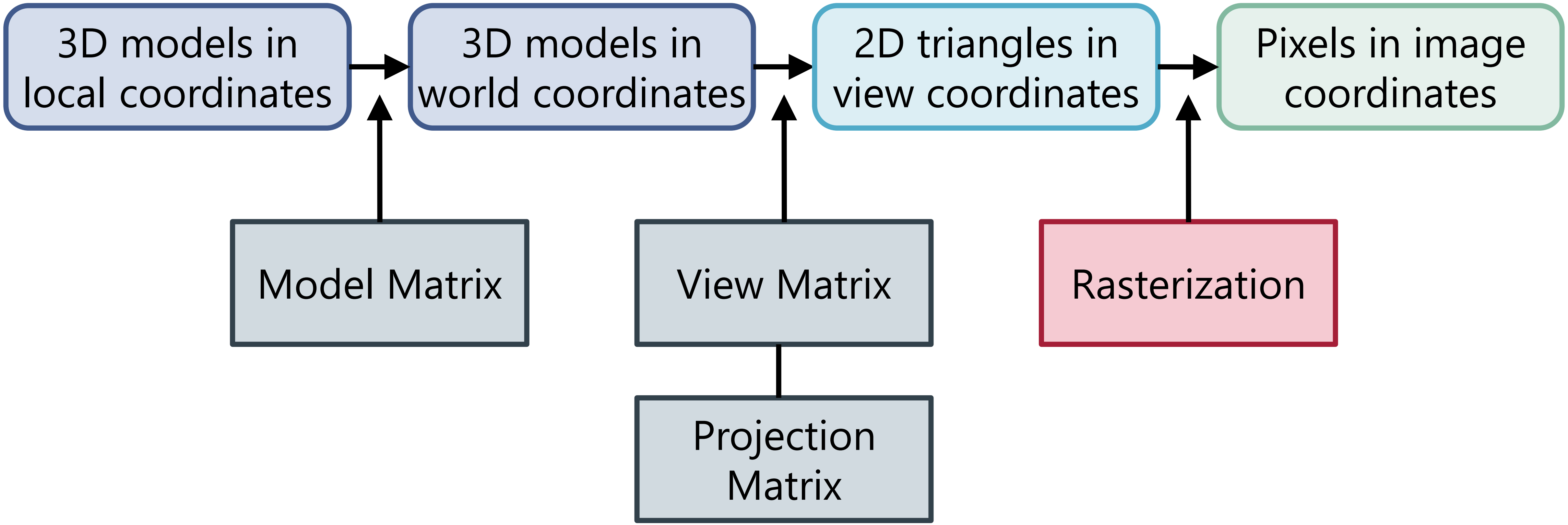
Mathematical Camera Model (Perspective Proj.)

- Virtual Camera
  - Defined by field of view (opening angle), aspect ratio (width/height), near, far
  - Orientation: position, view & up directions
  - → Defines *view frustum* (“truncated pyramid”)
- Two Matrices
  - **View matrix** – transforms *world coordinate* system into *view coordinate system*
  - **Projection matrix** – projects all geometry onto the image plane



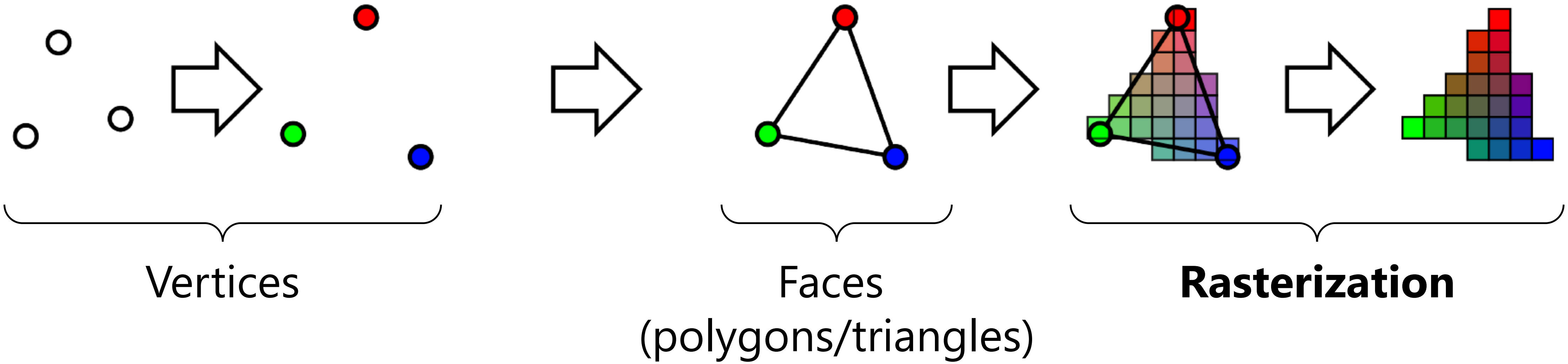
→ Details how to compute view matrix and projection matrix in “Graphische Datenverarbeitung”. *Three.js takes care of this for us!*

The 3D Rendering Pipeline *(simplified version)*



Rasterization

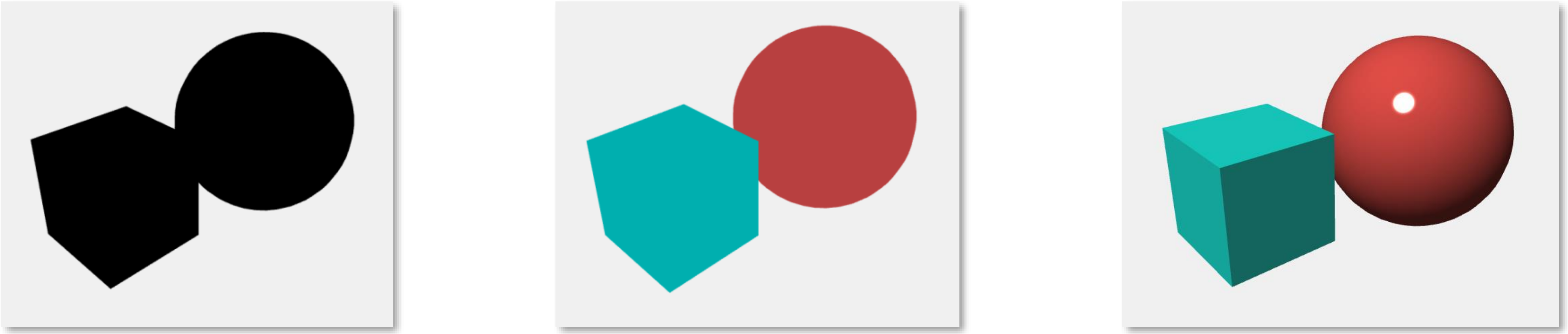
- Transfer objects from projected 2D coordinates to a pixel image
  - Fragments with xy-coordinates in screen space



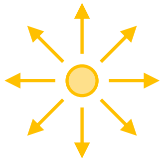
→ ***This is done automatically by the GPU!***

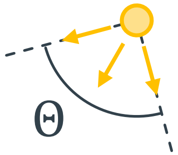
Lights, Materials, and Appearance

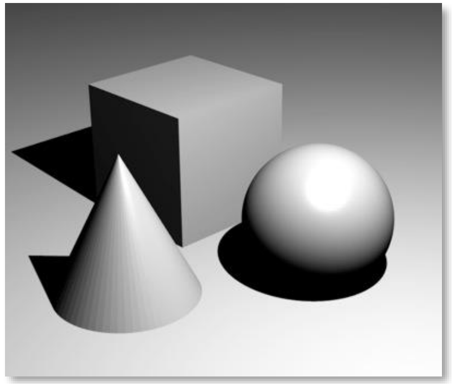
- Light in nature (physics refresher)
  - Can be described as a electromagnetic wave
  - Can also be described as a stream of photons
- Computer Graphics tries to model the physical transport of light
- **Why?**
  - Without light, everything is completely black or flat!
  - The illumination or *shading* gives objects shape



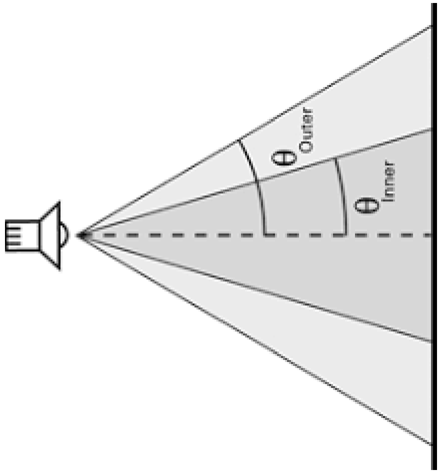
Light Sources



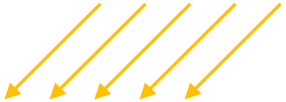




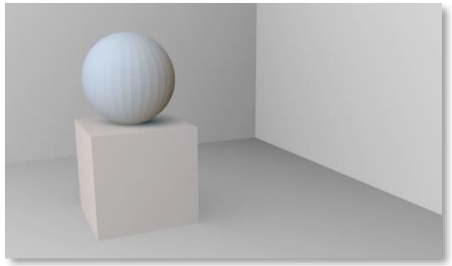
- Point Light
  - Just a position in space, emits light equally in all directions
  - Special case: Spot Light
    - Position and orientation in space, opening angles



- Distant Light / Directional Light
  - Simulates very distant light sources like the sun

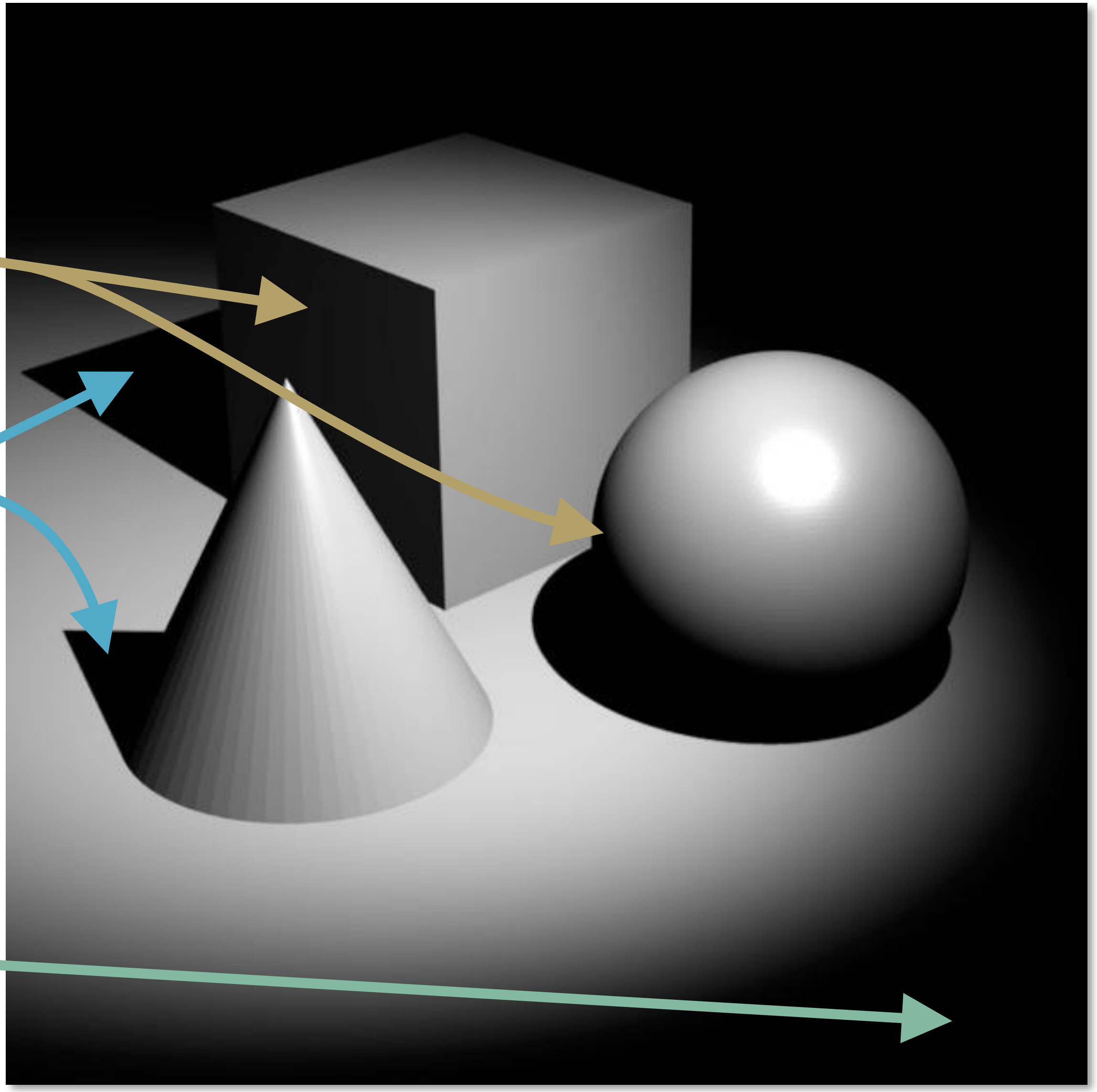


- Ambient Light
  - Equal intensity from all directions (“basic brightness” of scene)
- Area light source
  - Computationally difficult, take very long to render correctly
  - Can be modelled using many point lights in OpenGL/WebGL



Types of Shadows

- Object shadow
  - The side of objects that points away from the light
  - Exists in free space
- Cast shadow / drop shadow
  - The shadow cast onto another object (or the ground)
  - Need another object or ground plane
- Shadow as the absence of light
  - No light source reaches this place

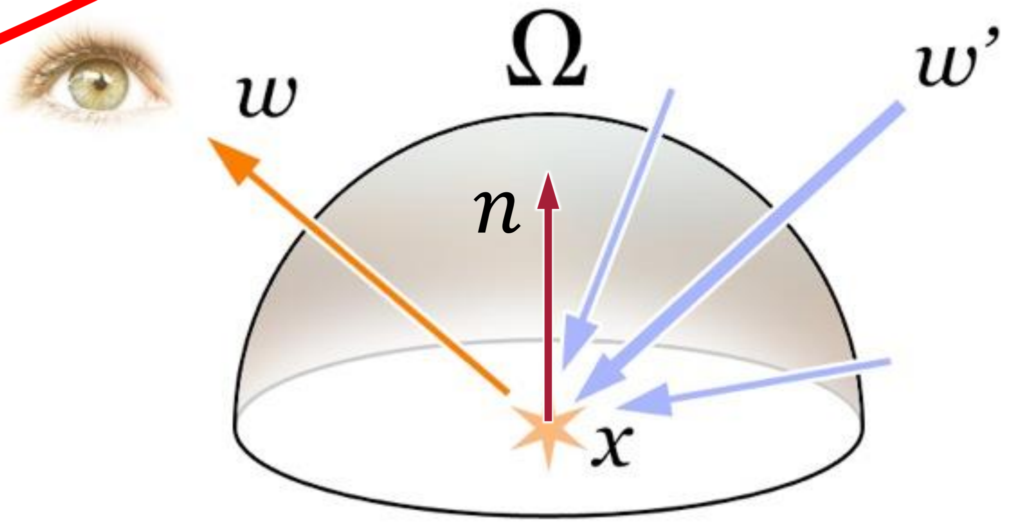


The Rendering Equation [Kajiya '86]

$$\underbrace{I_o(x,\vec{\omega})}\ =\ \underbrace{I_e(x,\vec{\omega})}\ +\ \int_\Omega \underbrace{f_r(x,\vec{\omega}',\vec{\omega})}\underbrace{I_i(x,\vec{\omega}')}\underbrace{(\vec{\omega}'\cdot\vec{n})}d\vec{\omega}'$$

- I_o = outgoing light
- I_e = emitted light
- Reflectance Function
- I_i = incoming light
- Angle of incoming light

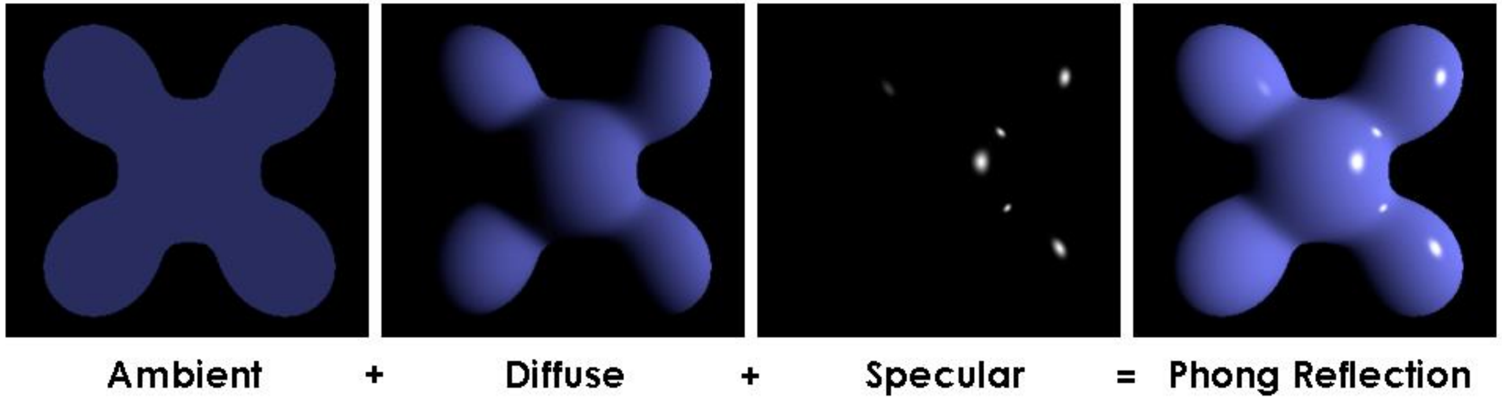
→ *Describes all flow of light in a scene in an abstract way*



http://en.wikipedia.org/wiki/File:Rendering_eq.png

Phong's Illumination Model [Bùi Tường Phong, 1973, PhD thesis]

$$I_o = I_{amb} + I_{diff} + I_{spec}$$



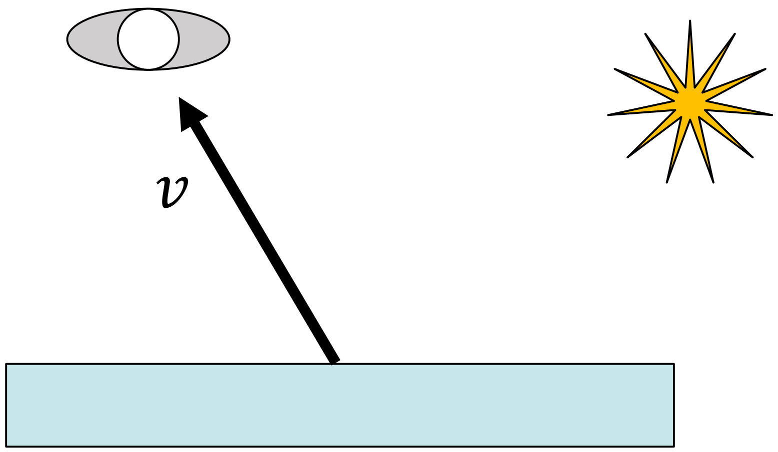
- Strong simplification and specialization of the situation
  - 1 point light source from a clear direction l; viewing direction is given as v
- Only 3 components:
  - Ambient component: reflection of ambient light source from/in all directions
  - Diffuse component: diffuse reflection of the given light source in all directions
  - Specular component: „glossy" reflection creating specular highlights

http://de.wikipedia.org/w/index.php?title=Datei:Phong_components_version_4.png

Ambient Component

- I_a = Intensity of the ambient light source

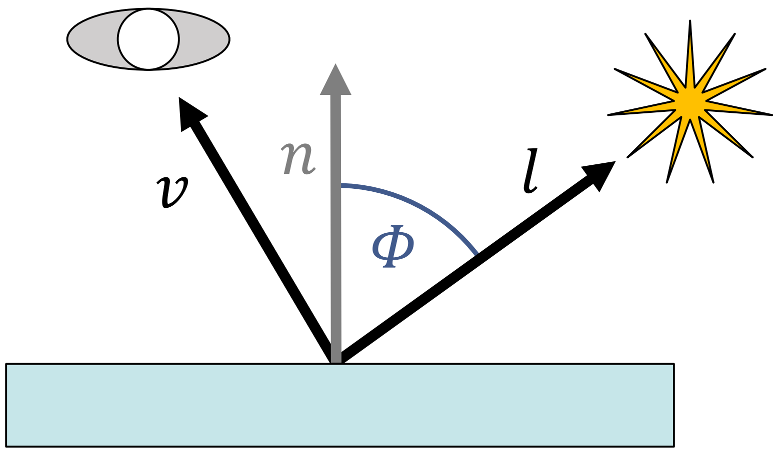
- Independent of any directions
- Can simulate a “glowing in the dark”
- Can be seen as the equivalent to emitted light I_e in the rendering equation



$$I_{amb} = I_a k_a$$

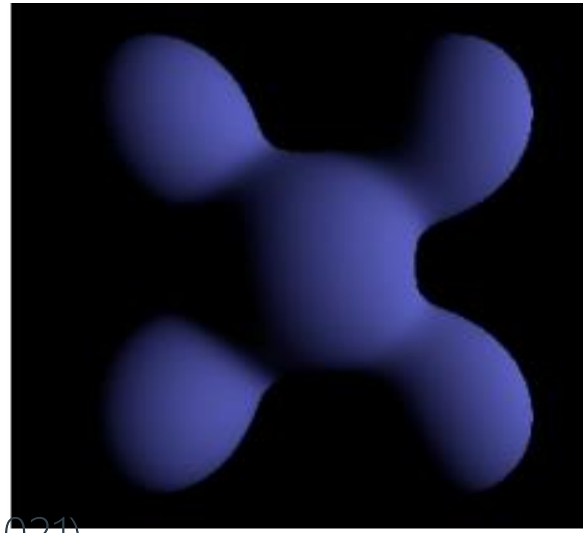
Diffuse Component

- Diffuse reflection is equal in all directions
- Depends on the angle of incident light
  - Light along the surface normal: maximum
  - Light perpendicular to the normal: 0



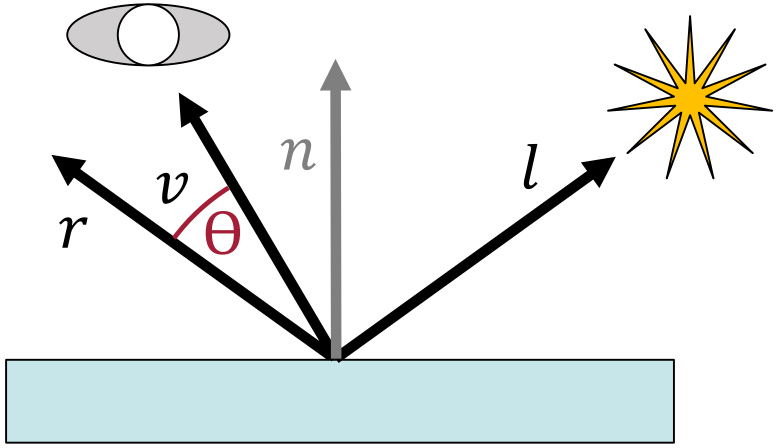
$$I_{diff} = I_i k_d cos\phi = I_i k_d(\vec{l} \cdot \vec{n})$$

- Cosine function describes the energy by which a given area is lit, dep. on angle
  - Hence, cosine is used here
- „Lambertian“ surface
- Visual equivalent in nature: paper



Specular Reflection

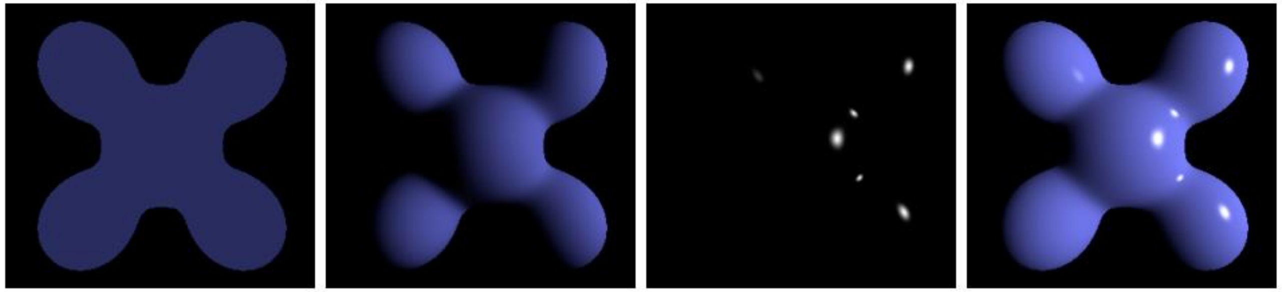
- Let r be the reflection of l on the surface
- Specular reflection depends on the angle between v and r
- $v = -r$: maximum
- v and r perpendicular: minimum
- Function $\cos^n \theta$ behaves correctly
  - Exponent n determines how wide the resulting specular highlight is
  - Other functions could be used as well



$$I_{spec} = I_i k_s cos^n \theta = I_i k_s (\vec{r} \cdot \vec{v})^n$$

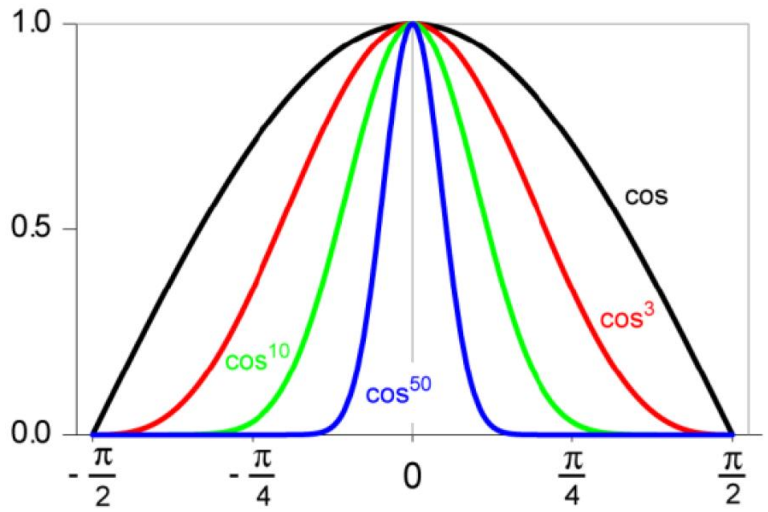
Tweaking the Parameters

$$I_o = I_{amb} + I_{diff} + I_{spec} = I_a k_a + I_i k_d(\vec{l} \cdot \vec{n}) + I_i k_s(\vec{r} \cdot \vec{v})^n$$



Ambient + Diffuse + Specular = Phong Reflection

- Choose $k_s = 0$ for perfectly matte material
- Choose $k_a > 0$ to avoid harsh shadows
- Keep k_a small to avoid "glowing" objects
- Add in some $k_s > 0$ to add gloss
- Adjust the size of specular highlights with n



Tweaking the Parameters

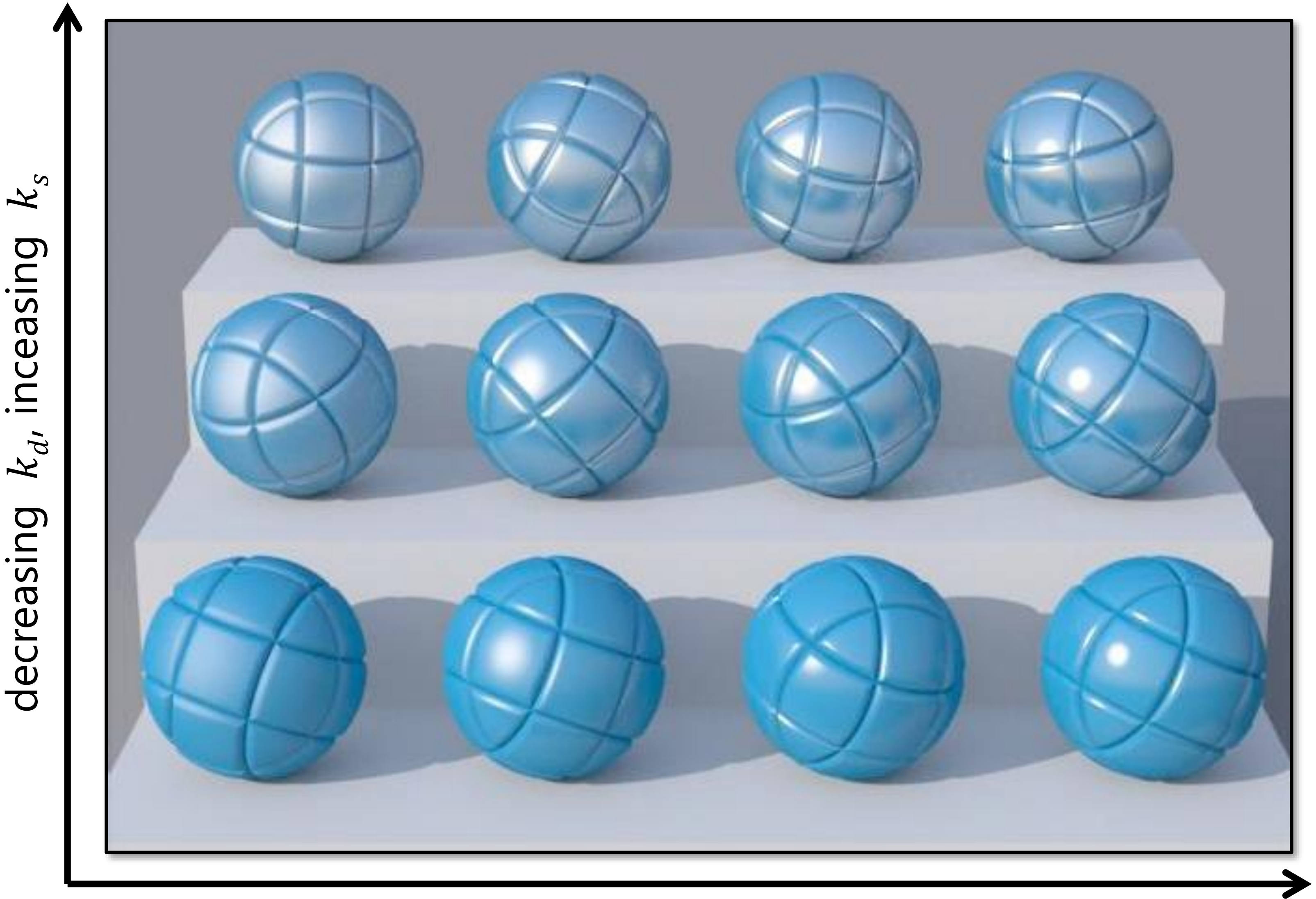
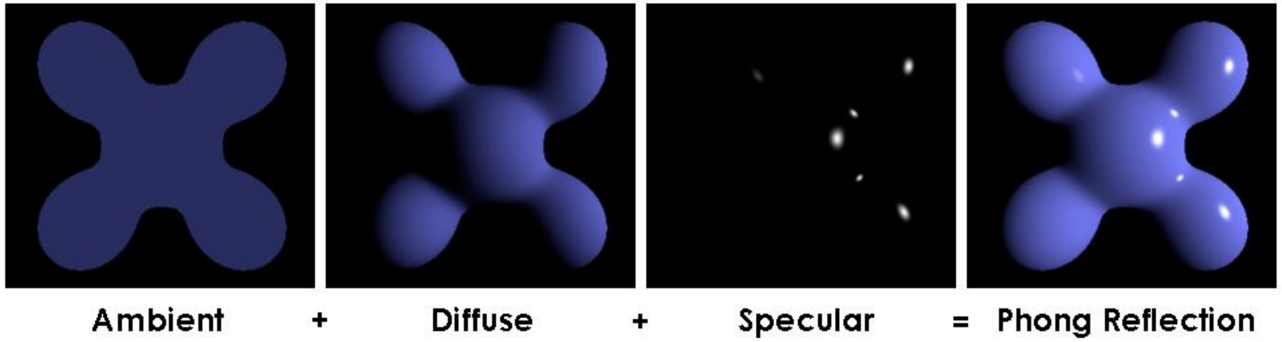


Image: Andrea Weidlich

Phong Illumination Model – Summary

$$I_o = I_{amb} + I_{diff} + I_{spec} = I_a k_a + I_i k_d(\vec{l} \cdot \vec{n}) + I_i k_s(\vec{r} \cdot \vec{v})^n$$

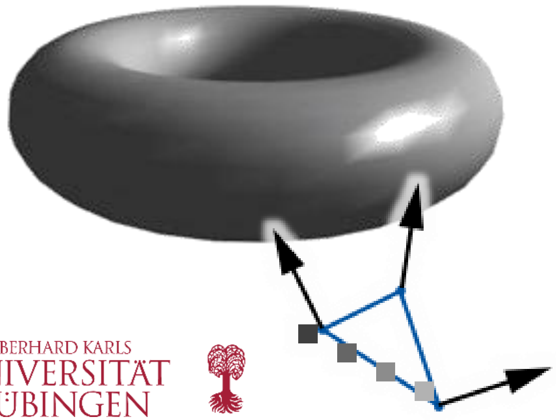


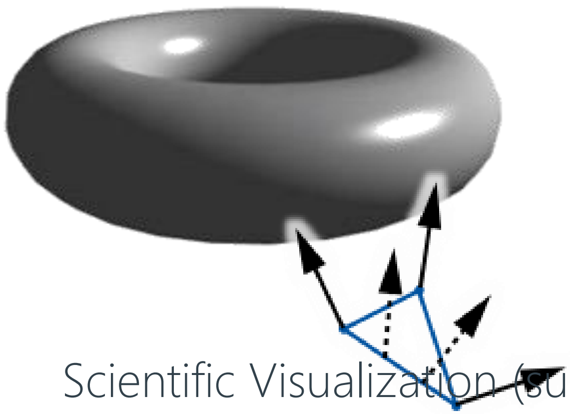
- Simplified approximation of Kajiya’s Rendering Equation
  - Approximates physical light transport
- Local illumination
  - Does not take other objects that might occlude the light source into account

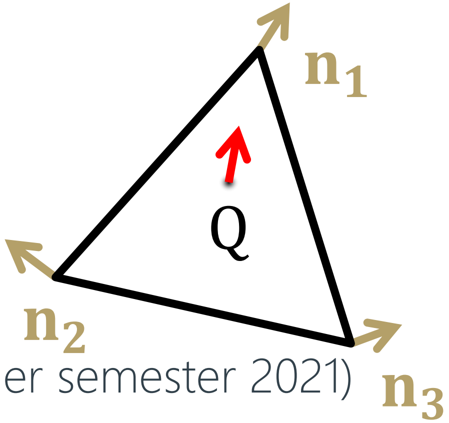
→ Only object shadows, no drop shadows → cheap, fast computation!

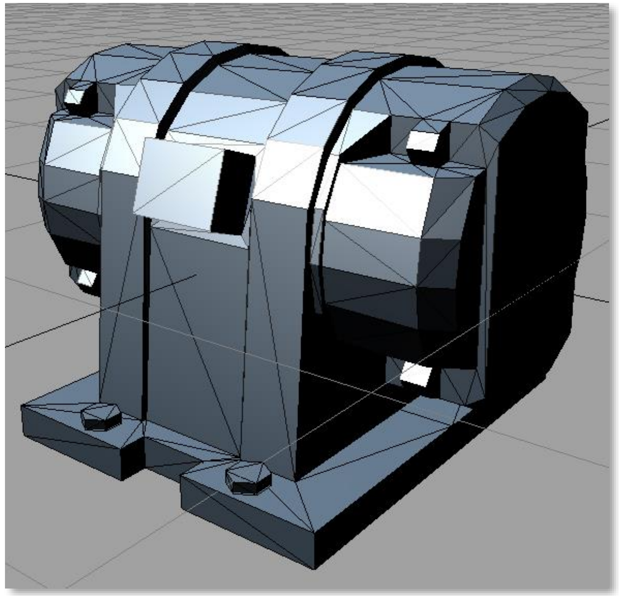
Illumination/Shading of Triangles

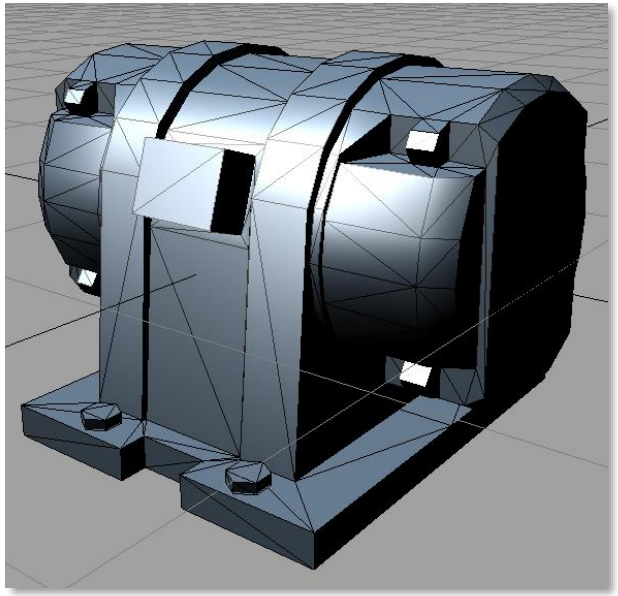
- Compute illumination based on normal and color
  - Can be defined per face (triangle) or per vertex
  - Per-vertex normals lead to smoothly shading
  - Two options for interpolation
    - Compute color per vertex, interpolate between colors
    - Interpolate normals, compute color per pixel
- Shading with interpolated normals is called **Phong Shading** (≠ Phong Illumination Model)





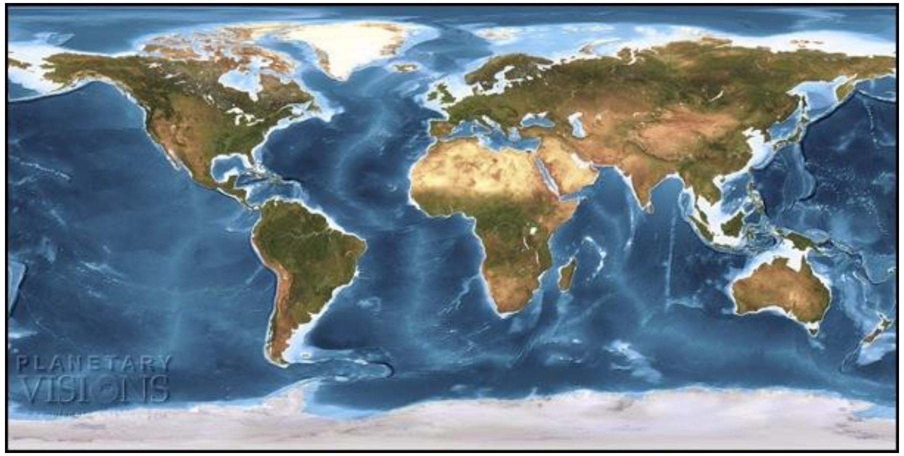


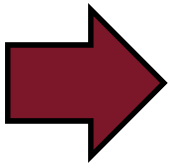




Textures and Mapping

- One of the simplest and oldest ways to achieve good looking objects with simple geometry
- **Idea:** use an image, shrink wrap around the object
  - Use image contents for object surface color: texture map
  - Can be used for other parameters, e.g., normal, elevation, reflection
- **Problem:** what does shrink wrap mean exactly?

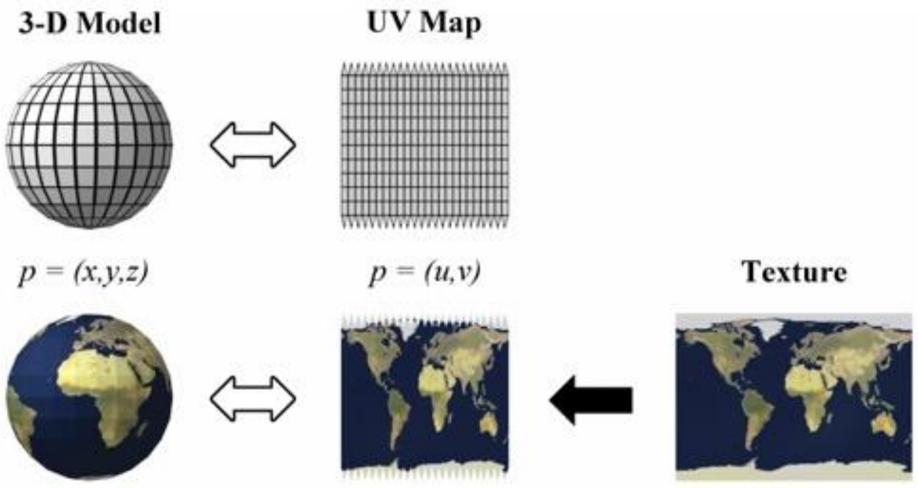




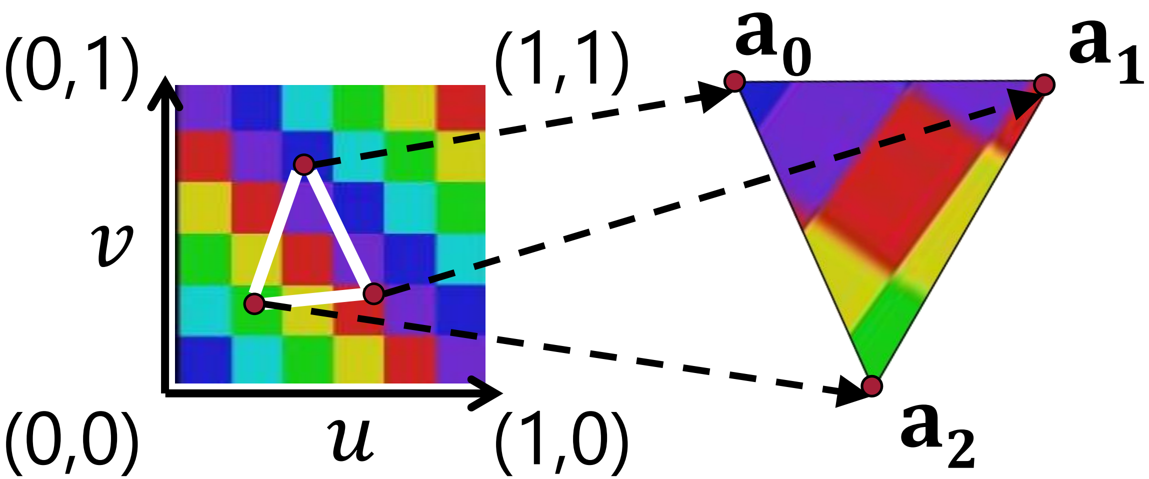
http://www.planetaryvisions.com/Texture_map.php?pid=4128

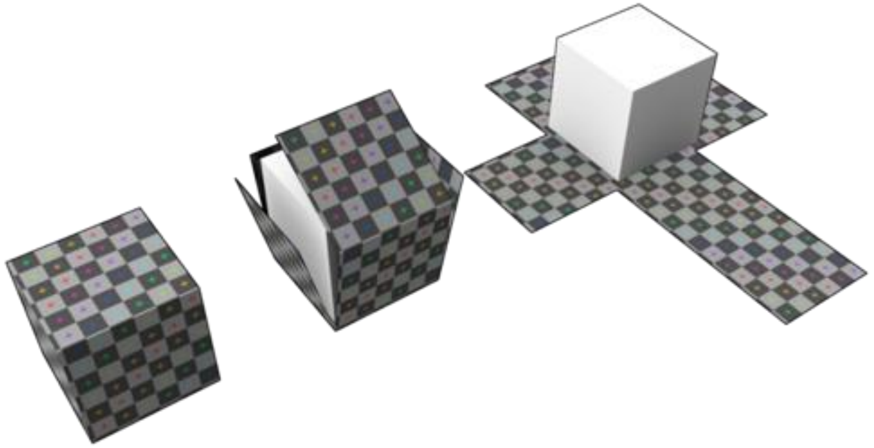
Texture Coordinates and UV Mapping

- Each texture is mapped to a 1×1 square
- Each object defines u, v coordinates
  - Texture coordinates defined per vertex
  - u or $v > 1$: repetition of the texture
- Relatively easy for geometric primitives
  - Three.js provides u, v coordinates



http://upload.wikimedia.org/wikipedia/commons/0/04/UVMapping.png



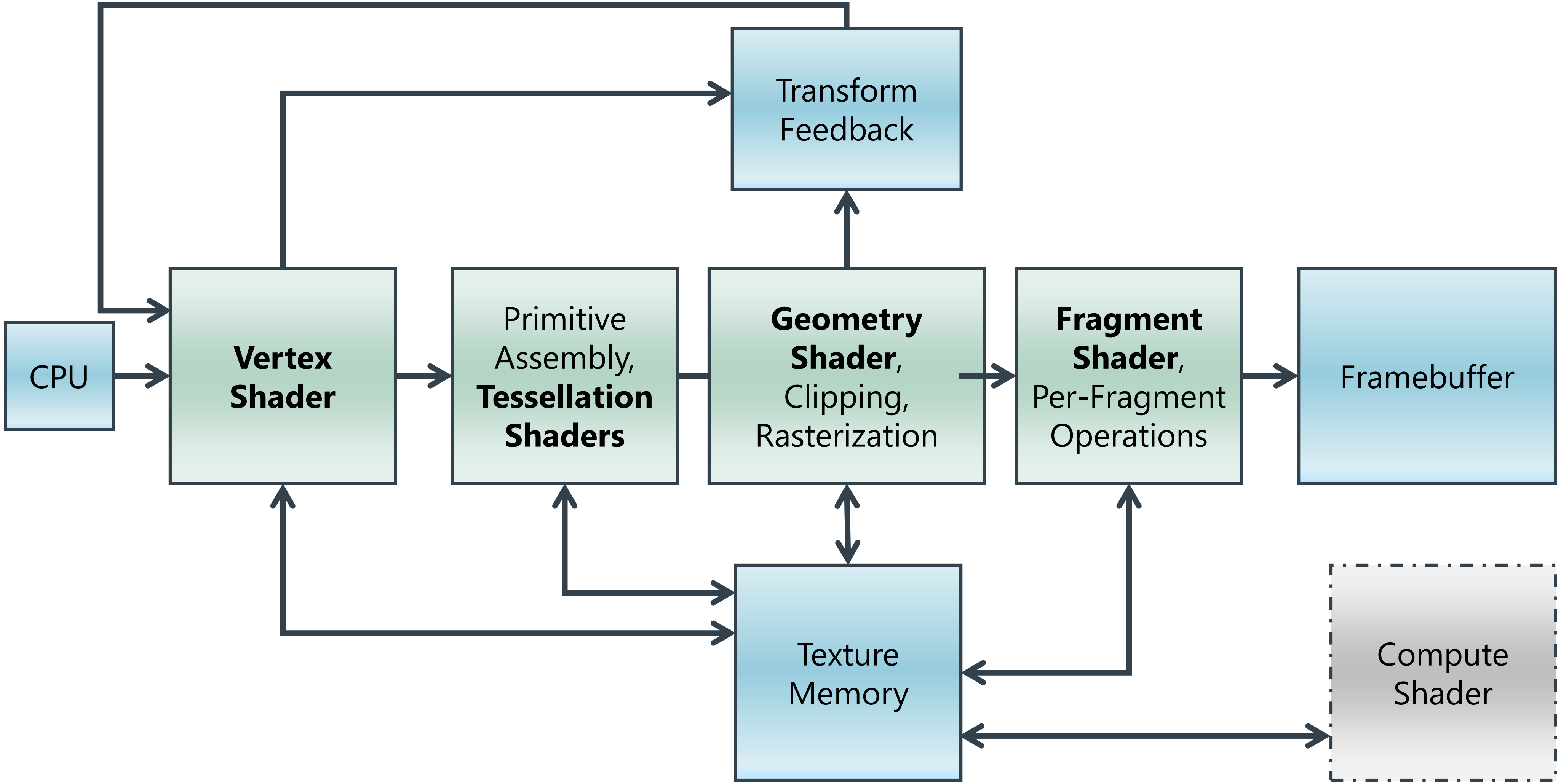


http://en.wikipedia.org/wiki/File:Cube_Representative_UV_Unwrapping.png

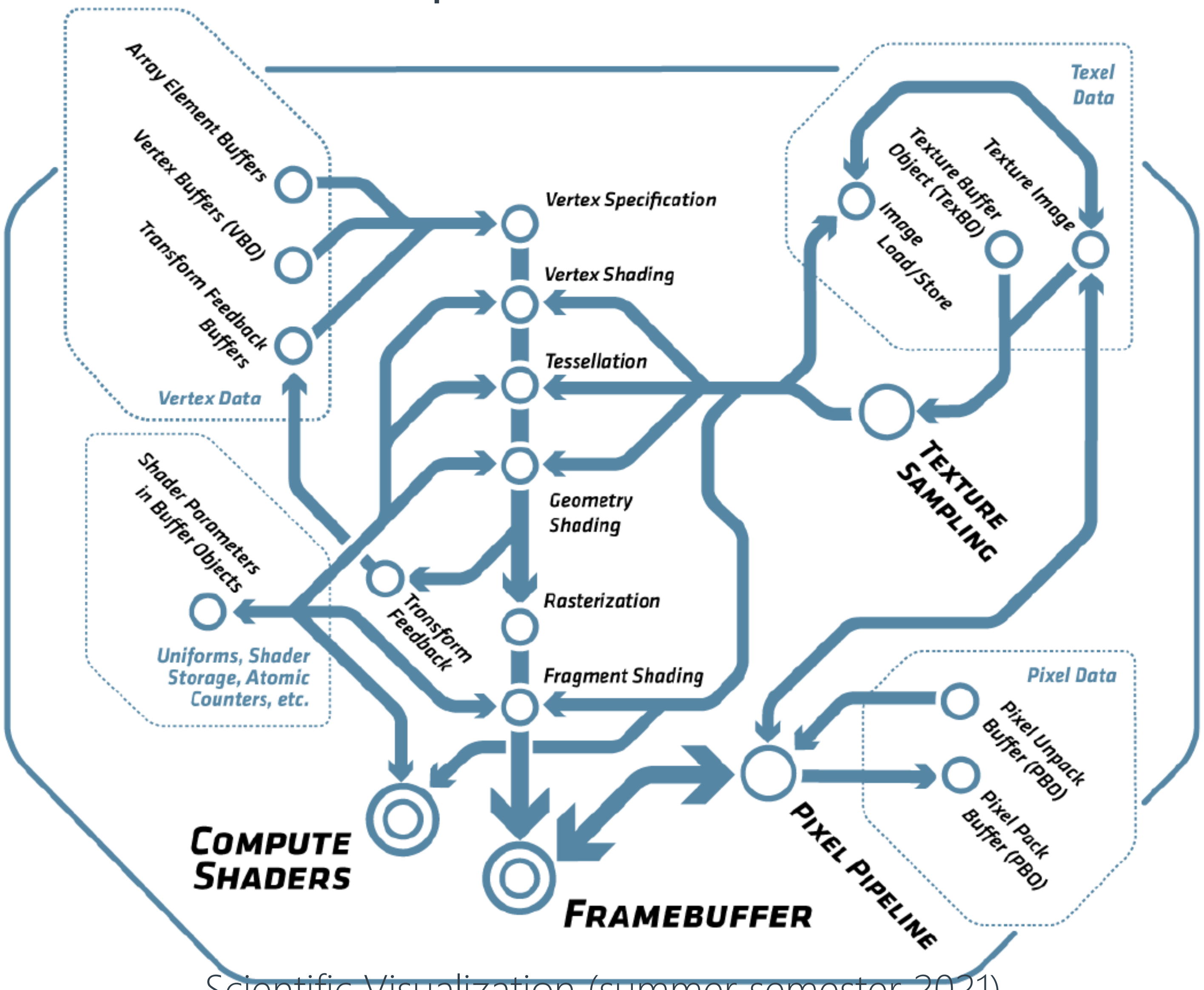
OpenGL / WebGL

- Cross-language, cross-platform graphics API, can interact with GPU to achieve hardware-accelerated rendering
  - ~200 instructions to define geometry and execute typical operations for interactive 3D graphics
  - Implements graphics pipeline
  - Programmable stages: shaders
- GPUs are highly parallel; useful to render interactively
  - Parallelize computations for all vertices/tringles/fragments (pixels) in *shaders*
  - SIMD (single instruction, multiple data) model
    - Different data (e.g., individual vertex positions), but the same operation (e.g., transformation using the same matrices) → no dependencies between vertices/triangles

OpenGL (4.5)-Pipeline *(slightly simplified)*



Full OpenGL (4.5) Pipeline

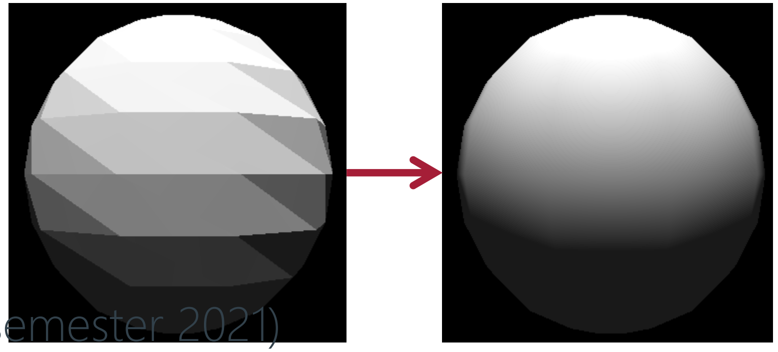


What is a Shader?

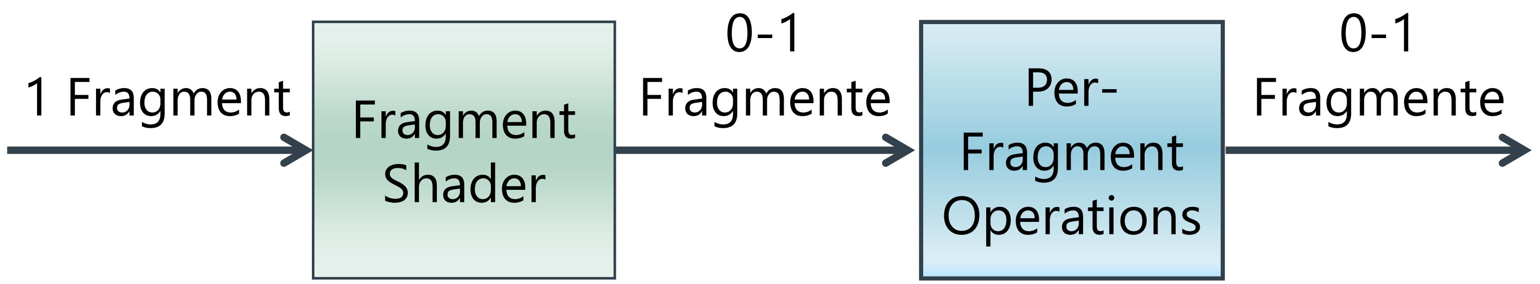
- Code for one of the programmable stages of the graphics pipeline
- 5(+1) Types: Vertex, Tessellation Control, Tessellation Evaluation, Geometry, and Fragment Shader (plus Compute-Shader)
  - WebGL supports only Vertex and Fragment shaders
- Capabilities of these 6 different types is similar
  - Operations and functions are identical
  - Semantics and layout of input and output data varies
- Programmable using high-level programming language "GLSL"
- In modern OpenGL (version ≥3) and WebGL, nothing happens without a Vertex and Fragment Shader

Geometry Processing: Vertex Shader

- Transformation of vertices and their attributes (e.g., normals,...)
- Computation of all attributes that are constant per vertex
  - e.g., transform vertex using model-view-projection matrix; per-vertex shading
- Assign attributes that have to be interpolated per Fragment
  - e.g., normals for per-pixel shading



Fragment Processing



- Fragment Shader
  - Computations per resulting pixel that is written to output buffer
    - Set final output color, per-pixel lighting/shading
  - Input attributes are interpolated within a triangle (can be disabled)
  - Fragments can be discarded
- Per-Fragment operations: Tests, Blending, etc. (more details later)

Example: The smallest (useful) Vertex Shader

User-defined input attribute (no built-in attributes)

User-defined input value (constant for all vertices)

```
in vec3 in_pos;
uniform mat4 TransformationMatrix;

void main() {
    gl_Position = TransformationMatrix * vec4(in_pos, 1.0);
}
```

Built-in output attribute for Vertex Shader (value passed to Fragment Shader)

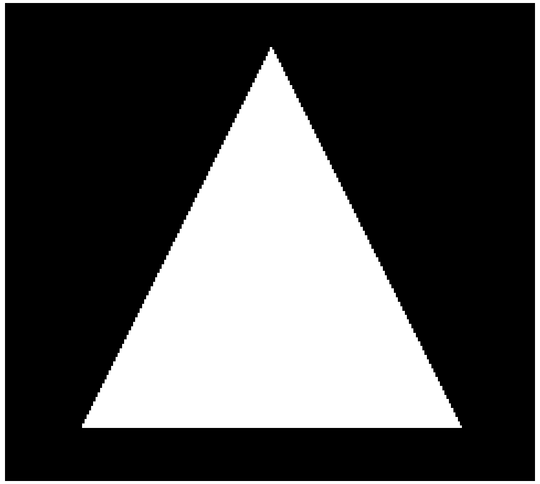
The smallest Fragment Shader

User-defined output attribute (no built-in attributes)

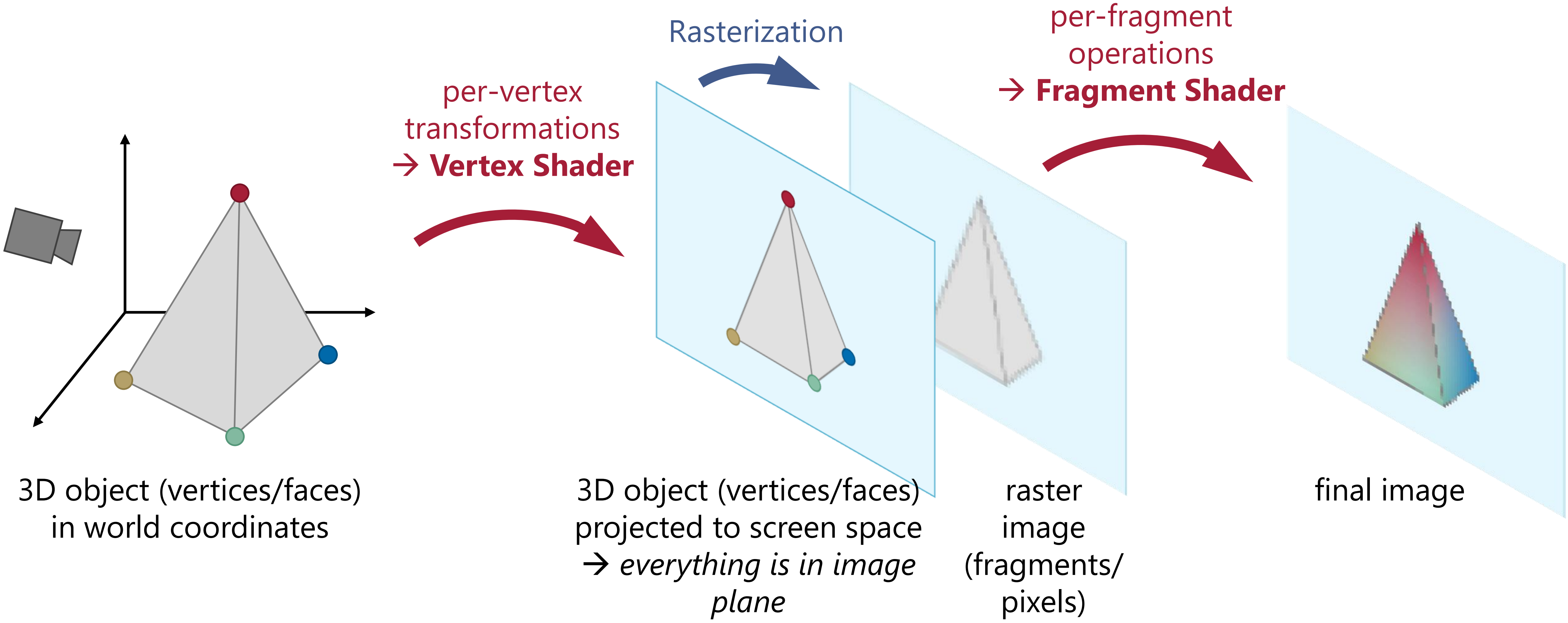
```
out vec4 out_frag_color;

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

Color vector (RGBA) → white

Output:



Graphics/Shader Pipeline Summary



Wrap-up: WebGL & Three.js

- Three.js provides a lot of built-in functions and shaders for different materials and illumination methods
  - User-defined shaders are possible for advanced tasks (we will need to use this later for the exercises)
  - User can pass almost arbitrary (numerical) information to the shaders
- Same goes for the view matrix and the projection matrix
  - Helper functions for camera and projection