VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Fundamentals of Computer Graphics

460-2021

Fall 2025

Last update 14. 10. 2025

Fundamentals of Computer Graphics

- Lecturer
  - Tomas Fabian
  - Office

    room EA408, building of FEECS
  - Office hours

    Tuesday 13:00 – 14:00, Thursday 13:00 – 14:00 (all other office hours are by appointment)
  - Email

    tomas.fabian@vsb.cz
  - Web site with additional materials

    http://mrl.cs.vsb.cz/people/fabian/zpg_course.html

Course Targets and Goals

- During the course, students will become familiar with the fundamental principles of 3D computer graphics using the C++ programming language and the OpenGL graphics API (optionally Vulkan), and will gain practical experience with shader programming in GLSL. They will progress through the steps from loading a 3D model to its visualization, including working with cameras, transforming objects and entire scenes, setting up lighting, working with textures, normal maps, shadows, skybox creation, and more.

- You will have hands-on experience with implementation of the here described methods and algorithms.

Course Prerequisites

- Basics of programming (C++)

- Previous courses:
  - NA

- To be familiar with basic concepts of mathematical analysis, linear algebra and vector calculus

Main Topics

- Introduction Computer Graphics. Raster and vector graphics (point, vector, line, curve, etc.). Interpolation.
- Graphics hardware. Introduction to standard rendering pipeline (OpenGL).
- 3D Object representation in CG (polygonal, CSG, procedural, etc.), object topology. Model formats (OBJ, FBX).
- Transformations in CG (move, rotation, scale), projective space.
- Projections (perspective vs. orthogonal projection), camera, clipping, rasterization.
- Colors, human eye, light (pointlight, spotlight, directional light, area light). Color mixing (blending).
- Lighting, local lighting models (Lambert, Phong), global lighting models, BRDF, radiosity, ray-tracing, ambient occlusion, shading.
- Textures in OpenGL texture units, Texel. UV mapping.
- Visible surface algorithms (z-buffer, painter's algorithm). Skybox, skydome.
- Bump mapping, normal mapping. Displacement mapping.
- Shadows in CG, shadow algorithm, shadow maps.
- Curves (Bezier curve) .

Organization of Semester and Grading

- Each lecture will discuss one main topic
- Given topic will be practically realized during the following exercises
- The individual tasks from the exercise will be scored (during the last week of the semester)
- You can earn a total of up to 45 points. The minimum number of points is 20
- The final combined (written and oral) exam covers topics from the previous slide.
- You can earn up to 55 points from the final exam. The minimum number of points is 10

Chat GPT/Copilot AI Policy

- Feel free to use them to prepare your project
- You are good enough to pass the class if you are good enough to verify their outputs. In other words, you need to be able to justify your code
- Beware, their outputs are sometimes completely wrong and they won't let you know.

Study Materials

- **[1] Segal, M., Akeley, K.: *The OpenGL Graphics System*, 2022, 850 pages. (online)**
- [2] Michael, A.: Graphics Programming Black Book. *Coriolis Group, Ames Iowa*, 2001. (online)
- [3] Pharr, M., Jakob, W., Humphreys, G.: *Physically Based Rendering*, Fourth Edition: From Theory to Implementation, MIT Press, 2025, 1312 pages, ISBN 978-0262048026. (online)
- [4] Dutre, P., Bala,K., Bekaert, P. *Advanced global illumination*. AK Peters/CRC Press, 2006.
- **[5] Haines, E., Akenine-Möller, T. (ed.): *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2025, 607 pages, ISBN 978-1484244265. (online)**
- **[6] Marrs, A., Shirley, P., Wald, I (ed.). *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Faller Nature, 2025, 858 pages, ISBN 978-1484271841. (online)**
- [7] Shirley, P., Morley, R. K.: *Realistic Ray Tracing*, Second Edition, AK Peters, 2003, 235 pages, ISBN 978-1568814612.
- [8] Akenine-Moller, T., Haines, E., Hoffman, N.: *Real-Time Rendering*, Fourth Edition, AK Peters/CRC Press, 2018, 1178 pages, ISBN 978-1138627000.
- [9] Dutré, P.: *Global Illumination Compendium*, 2003, 68 pages. (online)
- [10] Ryer, A. D.: *The Light Measurement Handbook*, 1997, 64 pages. (online)
- **[11] Gregory, J.: *Game engine architecture*. AK Peters/CRC Press, 2018.**

Study Materials

- Other free and downloadable materials are at https://www.realtimerendering.com

These are books that are **FREE ONLINE**, ordered by publication date. Do not be fooled by the price; all but one were published as physical books and each has valuable information.


Download for free **Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX**, edited by Adam Marrs, Peter Shirley, and Ingo Wald, Apress, August 4, 2021 (Book's website).


Download for free **Developing Graphics Frameworks with Python and OpenGL**, by Lee Stemkoski and Michael Pascale, CRC Press, July 7, 2021 (Publisher's website).


Download for free **Learn OpenGL - Graphics Programming: Learn modern OpenGL graphics programming in a step-by-step fashion**, by Joey de Vries, Kendall & Welling, June 2020 (Book's website, with fr


Download for free **Ray Tracing Gems**, edited by Eric Haines and Tomas Akenine-Möller, Apress, March 2019 (Book's website, publisher's page, Amazon), ***download for free***.


Read for free **Physically Based Rendering, Third Edition: from Theory to Implementation**, by Matt Pharr, Wenzel Jakob, and Greg Humphreys, Morgan Kaufmann, November 2016 (more information), Pat


Download for free **Ray Tracing in One Weekend**, by Peter Shirley, January 2016 (Code, tweet, blog), ***download for free***, ***read (corrected version) for free***.

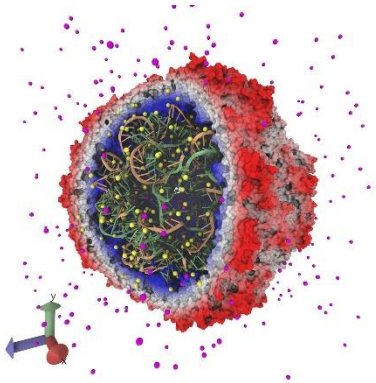

Download for free **Ray Tracing: the Next Week**, by Peter Shirley, March 2016 (Code, tweet, blog), ***download for free***, ***read (corrected version) for free***.

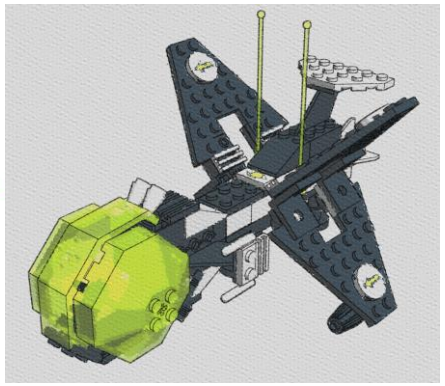

Download for free **Ray Tracing: The Rest Of Your Life**, by Peter Shirley, March 2016 (Code, tweet, blog), ***download for free***, ***read (corrected version) for free***.

- A large list of graphics books is also at https://www.realtimerendering.com/books.html

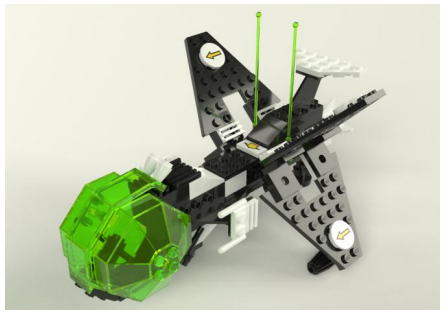
Types of Graphics

- *Non-photorealistic graphics/rendering*
  - Artistic styles
  - Scientific and engineering visualization
  - Data Visualization course





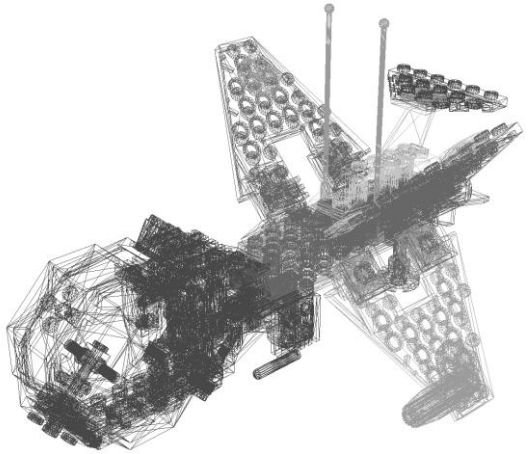
- *Photorealistic graphics/rendering*
  - Simulate the image formation process as precisely as possible
  - Physically plausible light transport through the scene
  - Topic of this course

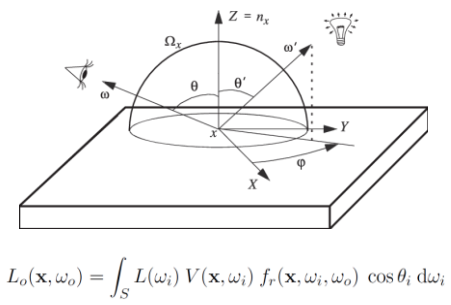


Photorealistic Image Synthesis

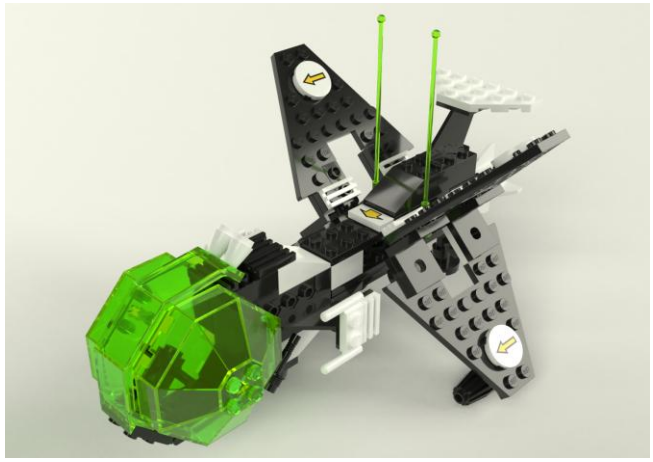
- You will be asked to create an realistically looking image based on a mathematical representation of a real or an artificial world

Real world or your imagination → Mathematical representation of the scene (light sources, geometry, materials, camers) → Mathematical description of light behaviour → Artificial image





$$L_o(\mathbf{x}, \omega_o) = \int_S L(\omega_i)\, V(\mathbf{x}, \omega_i)\, f_r(\mathbf{x}, \omega_i, \omega_o)\, \cos\theta_i\, \mathrm{d}\omega_i$$



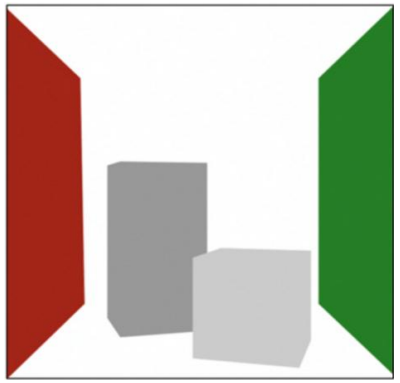
Application Areas

- Film industry – special effects or entire scenes/ films
- High quality rendering for commercials, prints, etc. (CG product images)
- Video game industry – ray tracing has recently entered this area (earlier e.g. prebaked lights)
- Architecture and design, virtual prototyping
- VR and AR (remote assistance and collaboration, conferencing)
- Various kind of simulations (lighting, sound propagation, collision detection, creating artificial datasets, etc.)

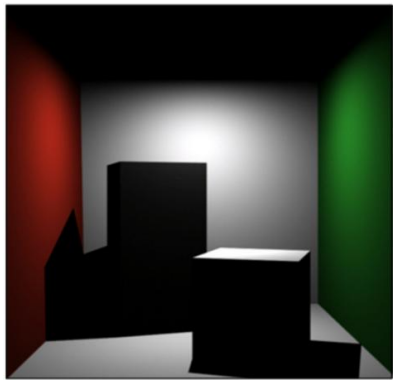
Knowledge Base

- Physics
  - Radiometry and photometry
  - Models of light interaction with various materials
  - Theory of light transport (mainly laws of geometrical optics)
- Mathematics
  - Linear algebra
- Informatics
  - Software engineering
  - Programming
- Visual perception and Art

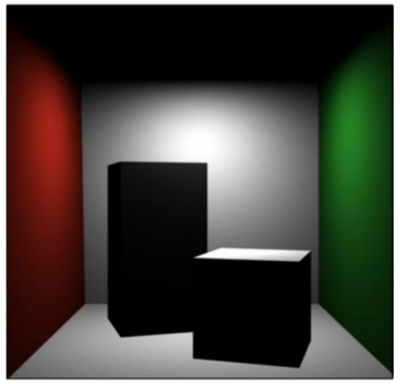
Levels of Realism



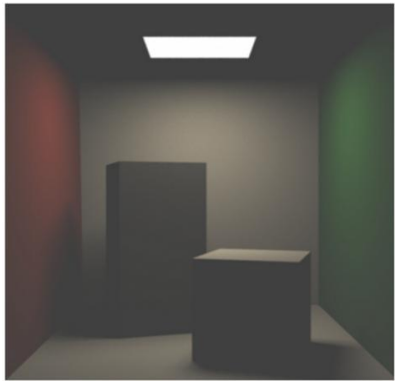
Surface color



Diffuse shading, point light, hard shadows



Diffuse shading, point light, no shadows



Diffuse shading, area light source, soft shadows

Diffuse inter-reflections, area light source

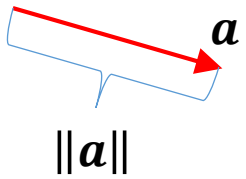
Direct vs. Global Illumination

- Direct illumination
  - A surface point illumination is computed directly from all light sources by the direct illumination model
- Global illumination
  - A surface point illumination is given by the complex light rays interaction with the entire scene

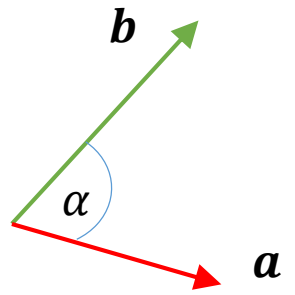
Basic Math Operations

- L2 norm $\|\boldsymbol{a}\| = \sqrt{\boldsymbol{a}_x^2 + \boldsymbol{a}_y^2 + \boldsymbol{a}_z^2} = \sqrt{\boldsymbol{a} \cdot \boldsymbol{a}}$

Unit vector $\hat{\boldsymbol{a}} = \frac{\boldsymbol{a}}{\|\boldsymbol{a}\|}$ and it holds that $\|\hat{\boldsymbol{a}}\| = 1$



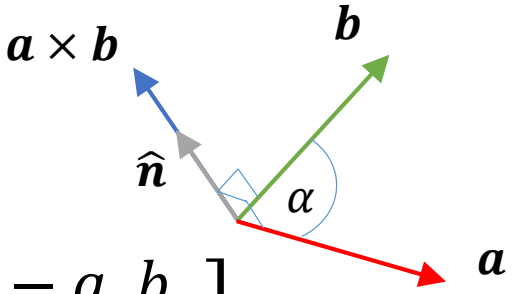
Basic Operators



- Dot product

$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^T \boldsymbol{b} = \sum_i a_i b_i = \|\boldsymbol{a}\| \|\boldsymbol{b}\| \cos \alpha$$

where α is an angle clamped between both vectors



- Cross product

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

Not commutative, follows the right hand rule, it also holds that

$\boldsymbol{a} \times \boldsymbol{b} = \|\boldsymbol{a}\| \|\boldsymbol{b}\| \hat{\boldsymbol{n}} \sin \alpha$ and $\|\boldsymbol{a} \times \boldsymbol{b}\| = \|\boldsymbol{a}\| \|\boldsymbol{b}\| |\sin \alpha|$

Basic Operators

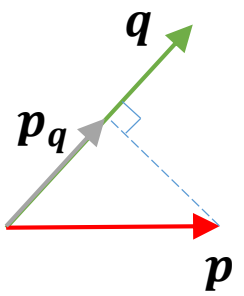
- (Vector) projection of $\boldsymbol{p}$ on $\boldsymbol{q}$

$$\text{proj}_{\boldsymbol{q}}\boldsymbol{p} = \boldsymbol{p}_{\boldsymbol{q}} = \frac{\boldsymbol{p}\cdot\boldsymbol{q}}{\|\boldsymbol{q}\|^2}\boldsymbol{q} = (\boldsymbol{p}\cdot\hat{\boldsymbol{q}})\hat{\boldsymbol{q}} = \frac{1}{\|\boldsymbol{q}\|^2}\underbrace{\begin{bmatrix} q_x^2 & q_xq_y & q_xq_z \\ q_yq_x & q_y^2 & q_yq_z \\ q_zq_x & q_zq_y & q_z^2 \end{bmatrix}}_{=\boldsymbol{q}\boldsymbol{q}^T}\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Matrix notation may be useful for repeated projections

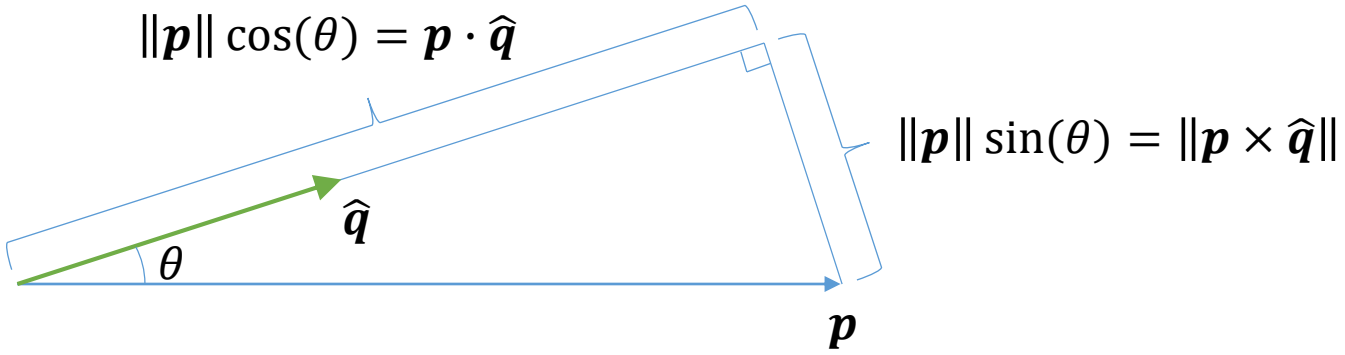
- Scalar projection of $\boldsymbol{p}$ on $\boldsymbol{q}$

$$\text{s.proj}_{\boldsymbol{q}}\boldsymbol{p} = \boldsymbol{p}_{\boldsymbol{q}} = \|\boldsymbol{p}\|\frac{\boldsymbol{p}\cdot\boldsymbol{q}}{\|\boldsymbol{p}\|\|\boldsymbol{q}\|} = \frac{\boldsymbol{p}\cdot\boldsymbol{q}}{\|\boldsymbol{q}\|} = \boldsymbol{p}\cdot\hat{\boldsymbol{q}}$$



Basic Operators

- (Vector) projection of $\boldsymbol{p}$ on $\boldsymbol{q}$



Basic Math Operations

- Other useful formulas

- $\|\boldsymbol{a}-\boldsymbol{b}\|^2 = \boldsymbol{a}\cdot\boldsymbol{a} - 2(\boldsymbol{a}\cdot\boldsymbol{b}) + \boldsymbol{b}\cdot\boldsymbol{b}$

- Lagrange's identity $\|\boldsymbol{a}\times\boldsymbol{b}\|^2 = \|\boldsymbol{a}\|^2\|\boldsymbol{b}\|^2 - (\boldsymbol{a}\cdot\boldsymbol{b})^2$

Rotations

- Counterclockwise rotation in 2D about the origin by angle α

$$\boldsymbol{a}' = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \boldsymbol{a}$$

$\alpha := 45\ \textit{deg}$

$R := \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \quad a := \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad R \cdot a = \begin{bmatrix} 0.707 \\ 0.707 \end{bmatrix}$

- Elementary counterclockwise rotations in 3D

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, R_y = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}, R_z = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Arbitrary rotation may be decomposed into three components (3 Euler angles), non commutative – order of rotations matters (complicated interpolation, gimbal lock)

Rotations

- Counterclockwise rotation of point $\boldsymbol{a}$ around arbitrary unit axis $\hat{\boldsymbol{r}}$ by angle α (Rodrigues' rotation formula)

$$\boldsymbol{a}' = (1 - \cos(\alpha))(\boldsymbol{a} \cdot \hat{\boldsymbol{r}})\hat{\boldsymbol{r}} + \cos(\alpha)\boldsymbol{a} + \sin\alpha\,(\hat{\boldsymbol{r}} \times \boldsymbol{a})$$

$\alpha := 45\ deg$

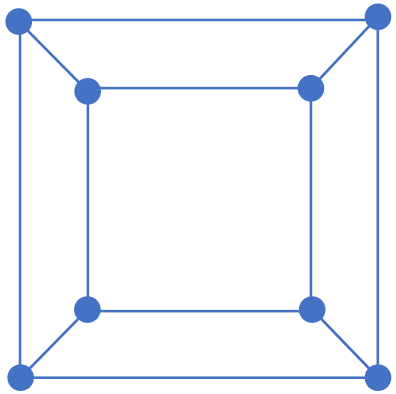
$$r := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad a := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad (1-\cos(\alpha))\cdot(a\cdot r)\cdot r + \cos(\alpha)\cdot a + \sin(\alpha)\ (r \times a) = \begin{bmatrix} 0.707 \\ 0.707 \\ 0 \end{bmatrix}$$

For further reference see http://ksuweb.kennesaw.edu/~plaval/math4490/rotgen.pdf

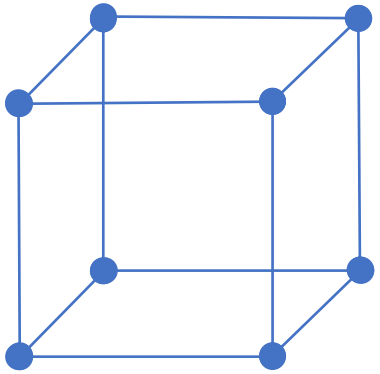
Vertices

Topology is the mathematical study of the properties that are preserved through deformations, twistings, and stretchings of objects. Tearing, however, is not allowed.

- A point (corner or node) where two edges (lines) meet
- The connectedness between the vertices defines a mesh's topology

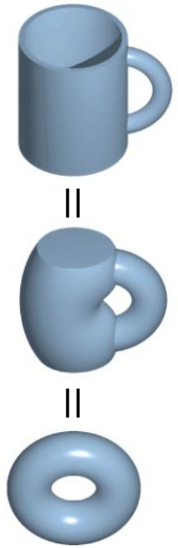


Graph embedding



Topology
Connectivity/neighborhood

Geometry
Position of vertices

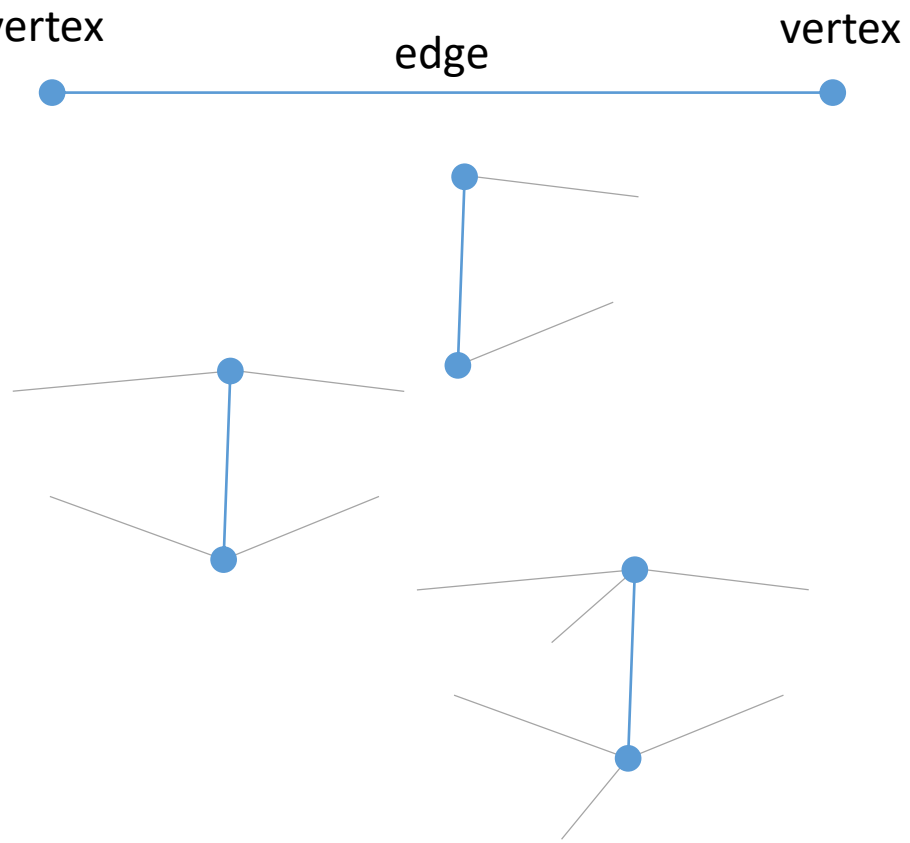


Topological equivalence
(homeomorphism, homotopy)

- Beside the position, vertices may have other attributes: color, normal (tangent, bitangent), texture coordinates, etc.

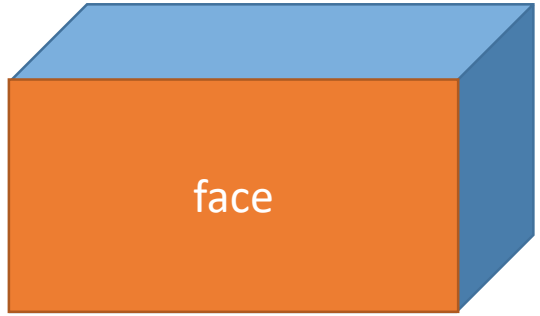
Edges

- Connection of two vertices
- Boundary (1 incident face)
- Regular (2 incident faces)
- Singular (3 or more incident faces)

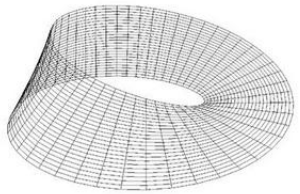


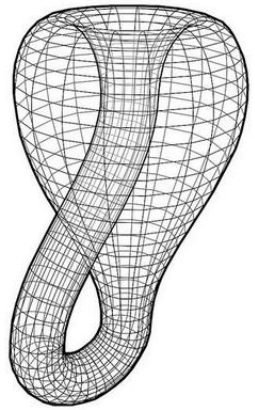
Faces

- The flat surface on a shape or a solid is known as its face



Manifold

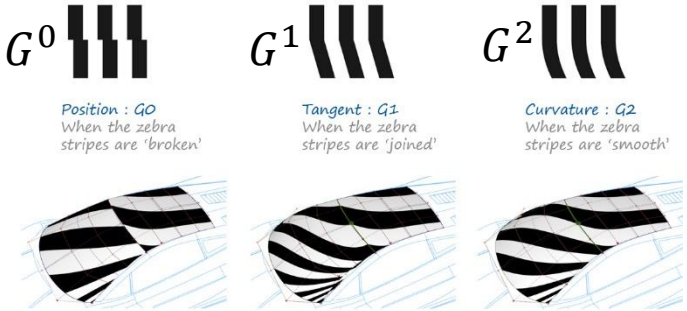




- A n-dimensional manifold is a topological space that locally resembles (homeomorphic) n-dimensional Euclidean space near each point
- 1-manifolds include lines and circles (not a lemniscate or ∞)
- 2-manifold (surface) include plane, sphere (genus 0), torus (genus 1), Klein bottle (not orientable), Möbius strip (closed and not orientable), real projective plane (closed, non-orientable), triangle (smooth manifold with boundary)
- Theorem:
  - Every orientable and closed surface is homeomorphic to a connected sum of tori
  - Every surface is homeomorphic to a connected sum of tori and/or projective planes

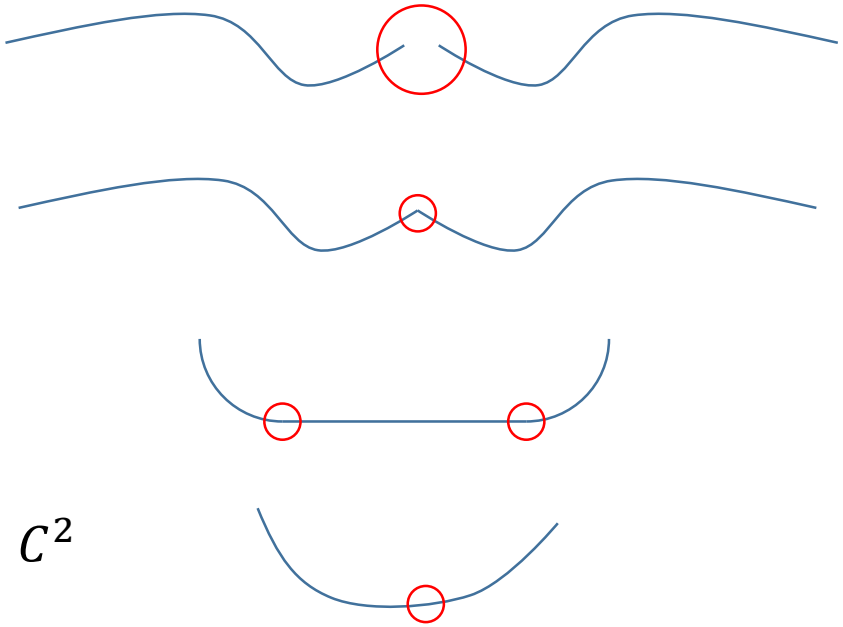
Smoothness

Geometric continuity requires that the parametric derivatives of the two segments be proportional to each other, not equal



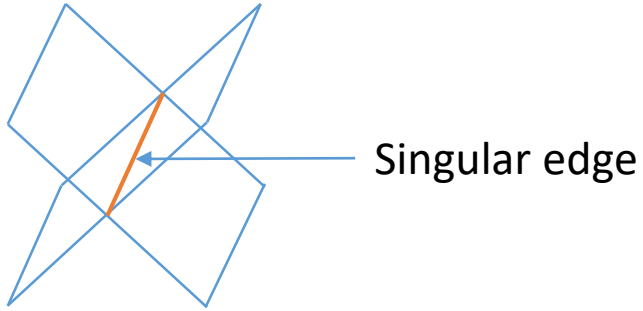
Geometric continuity examples, source: Autodesk Alias Workbench

- Reminder: a function is called C^n continuous if it's n-th order derivative is continuous **everywhere**
- Parametric continuity examples:
  - Not continuous
  - Position continuity $= C^0$
  - Position and tangent continuity $= C^1$
  - Position, tangent, and curvature continuity $= C^2$



Manifold Meshes (with Boundaries)

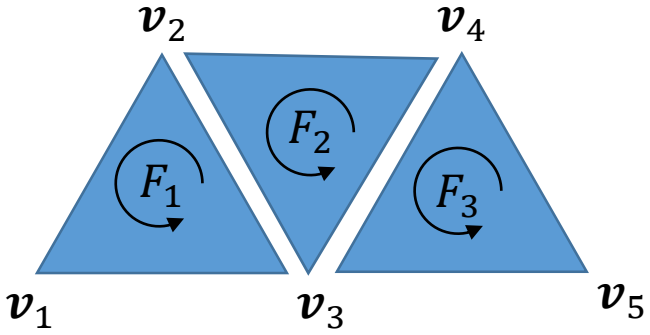
- No singular edges
- No singular vertices



- Simples data structure
  - Polygon soup – list of faces formed by n-tuple of vertices
  - No (explicit) information about adjacency (can be stored in adjacency list)

F_1	$\boldsymbol{v}_1, \boldsymbol{v}_3, \boldsymbol{v}_2$	F_2
F_2	$\boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{v}_4$	F_1, F_2
F_3	$\boldsymbol{v}_3, \boldsymbol{v}_5, \boldsymbol{v}_4$	F_2

Face orientation (CW or CCW) is given by the order in which the vertices are listed



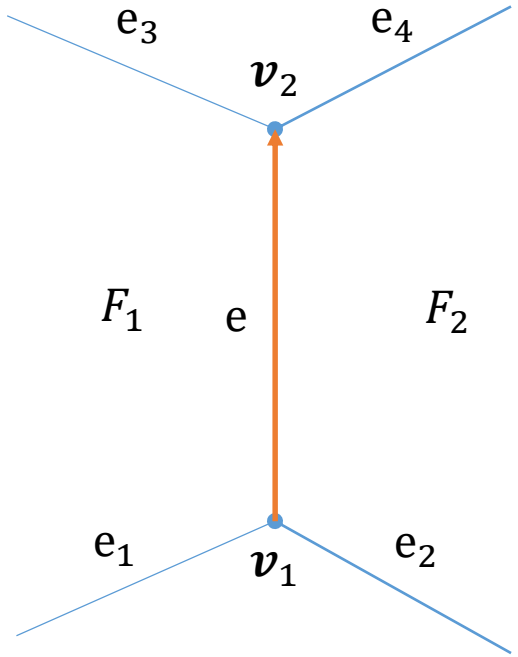
  - Possible extension with indexing (lower memoty req.) (note that shared vertices must have same atributes)

OBJ Format

```
v  14.363998 -19.782551 78.445435
v  14.106861 -19.928997 78.699089
v  13.828163 -19.789648 78.457726
v  14.051285 -19.662575 78.237625
v  14.953951 -19.069986 77.211243
v  14.877973 -19.261126 77.542297

vn -0.111118 0.910786 0.397570
vn 0.132542 0.566302 -0.813439
vn 0.192175 0.622958 -0.758245

f 1//1 2//2 3//3
f 4//1 5//2 6//3
```

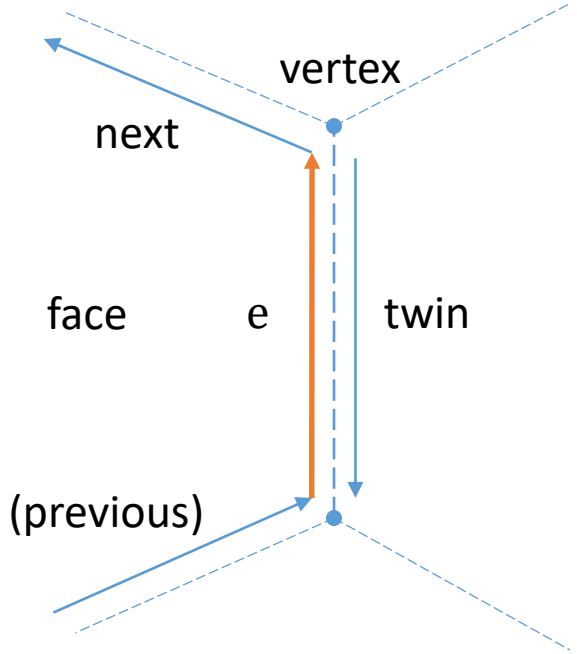
Winged-Edge

- Structure is also valid for non-orientable manifolds



Half-edge Data Structure (DCEL)

- Structure is valid only for orientable manifolds

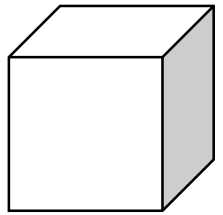


Euler Characteristic χ

The Euler-Poincaré formula describes the relationship of the number of vertices, the number of edges and the number of faces of a manifold. It has been generalized to include potholes and holes that penetrate the solid.

- Simplified Euler's formula for any convex polyhedron's surface

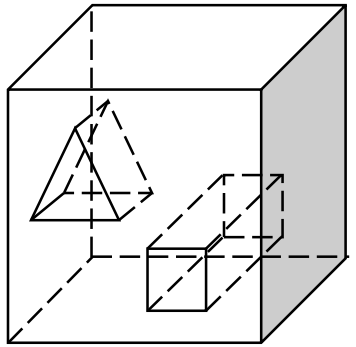
$$V - E + F = 2(S) = \chi$$



$$8 - 12 + 6 = 2(1)$$

- Generalized formula allowing holes

$$V - E + F - L = 2(S - H)$$



$$22 - 33 + 14 - 3 = 2(1 - 1)$$

L … number of holes in faces (inner loops)

H… number of holes passing through the whole solid (genus)

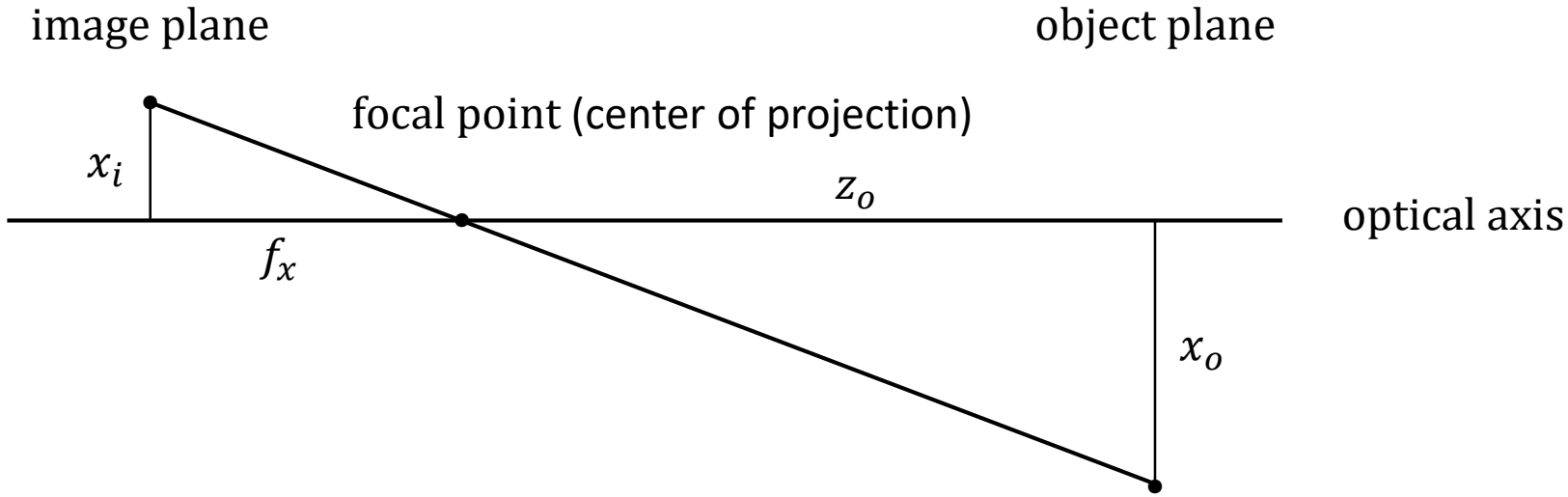
S … number of disjont components (shells, solids)

A homeomorphism is a bijection that is continuous and its inverse is also continuous.

Pinhole Camera Model

Similar triangles: All the corresponding sides have lengths in the same ratio.
$\frac{a}{a'} = \frac{b}{b'} = \frac{c}{c'}$

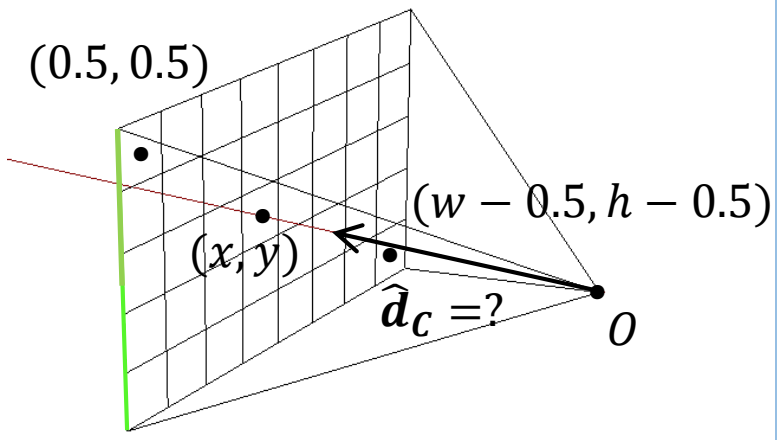
- Idealized model for the optics of a camera defining the geometry of perspective projection

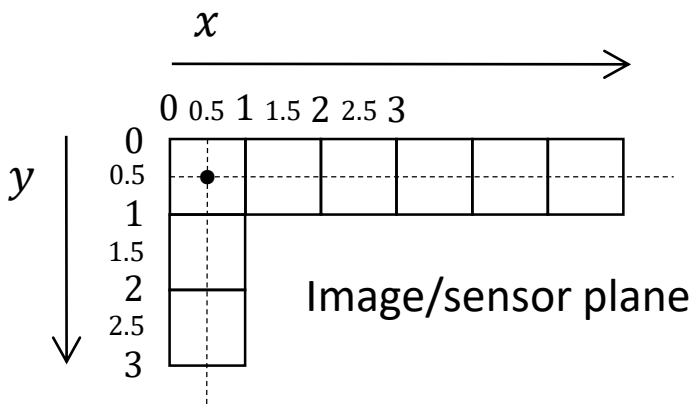


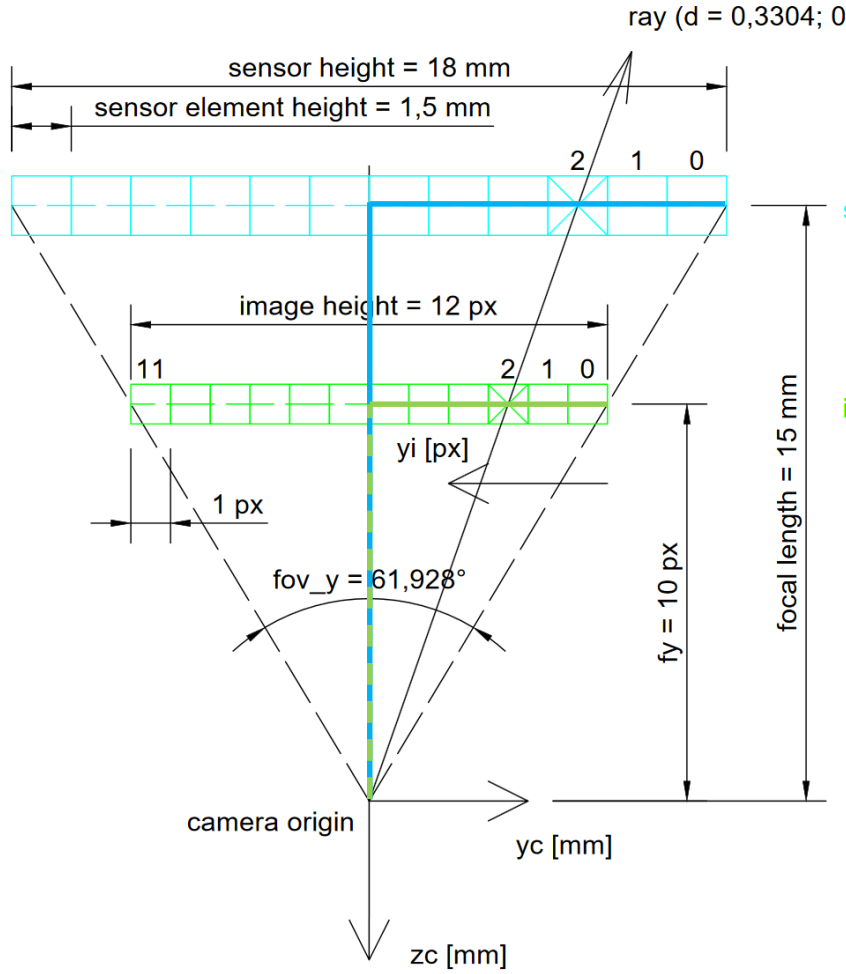
$$\frac{x_i}{f_x} = \frac{x_o}{z_o}, \quad \frac{y_i}{f_x} = \frac{y_o}{z_o}$$

i … image point coordinates
o … object point coordinates

Primary Ray Generation in Camera Space







sensor plane — What would we deal with in the case of a real camera

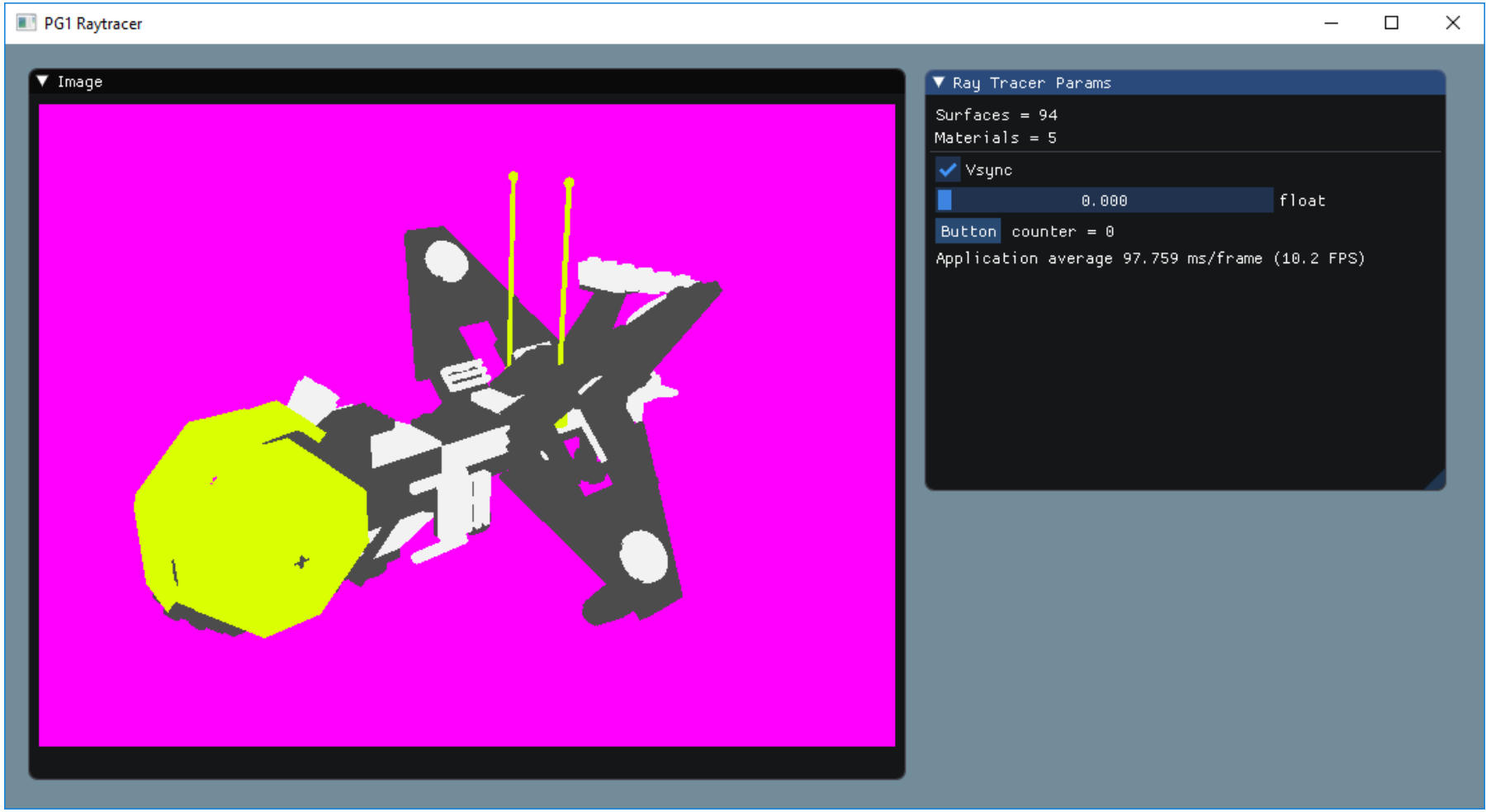
image plane — What will we deal with in the case of an imaginary camera

Known camera parameters:
O, T, width w, height h, fov_y

$f_y = f_x = ?$ (px)

$\boldsymbol{d_C} = (x - w/2\,, h/2 - y\,, -f_y)$

Result of First Exercise

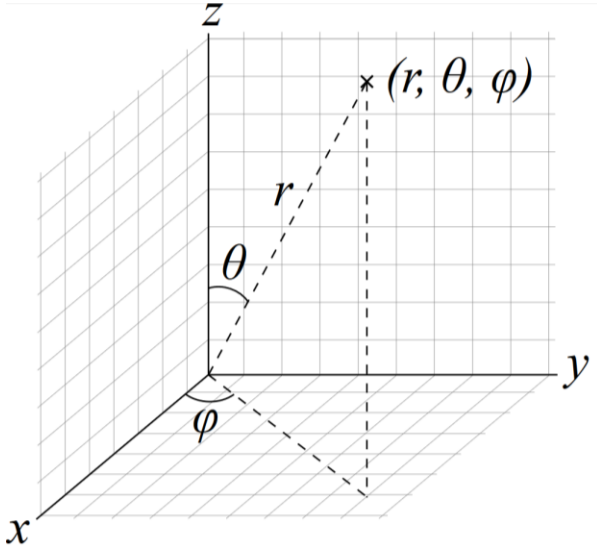


Representation of Direction in 3D

- Cartesian coordinates

$$\widehat{\boldsymbol{d}} = (d_x, d_y, d_z),\ \left\|\widehat{\boldsymbol{d}}\right\| = 1$$

- Spherical coordinates (physics and ISO convention)
  - $\omega = (\theta, \varphi)$
  - Polar angle (theta) $\theta = \cos^{-1} d_z \in \langle 0, \pi \rangle$
  - Azimuthal angle (phi) $\varphi = \tan^{-1} \frac{d_y}{d_x} \in \langle 0, 2\pi)$



$$d_x = \sin\theta \cos\varphi$$
$$d_y = \sin\theta \sin\varphi$$
$$d_z = \cos\theta$$

In C/C++, we can compute the azimuthal angle as $\varphi = \texttt{atan2f}(d_y, d_x) + \begin{cases} 2\pi & d_y < 0 \\ 0 & \text{otherwise} \end{cases}$

Affine and Projective Spaces

- Affine space
  - Set V of vectors and set P of points
  - Affine transformations can be represented by 3×3 matrix
- Projective space
  - Homogeneous coordinates (x, y, z, w)
  - All lines intersect (space contains infinity $(x, y, z, 0)$)
  - Projective transformations can be represented by 4×4 matrix (inc. translation and perspective projection)
  - Cartesian to homogeneous coordinates: $(x, y, z) \rightarrow (x, y, z, 1)$
  - Homogeneous to Cartesian coordinates: $(x, y, z, w \neq 0) \rightarrow (x/w, y/w, z/w)$

Combination of Points

- Let have an affine space A modeled on a vector space V, points $P, Q \in A$, and vectors $v, w \in V$ and axioms $Q - P \in V$ and $P + v \in A$
- Suppose we want to define a combination of points like this
$$\alpha_1 P_1 + \alpha_2 P_2 + \cdots + \alpha_n P_n$$
- At first glance, that doesn't make sense – points cant be added directly. So we can fix one point (say P) as a reference. Then, each point can be expressed as
$$P_i = P_0 + (P_i - P_0), \text{ where } (P_i - P_0) \in V$$
- Plugging this into the combination above yields
$$\alpha_1 P_1 + \cdots + \alpha_n P_n = \alpha_1\big(P_0 + (P_1 - P_0)\big) + \cdots + \alpha_n\big(P_0 + (P_n - P_0)\big)$$
- After simplification we get
$$(\alpha_1 + \cdots + \alpha_n)P_0 + (\alpha_1(P_1 - P_0) + \cdots + \alpha_n(P_n - P_0))$$
- Iff $\sum \alpha_i = 1 \Rightarrow P_0 + \alpha_1(P_1 - P_0) + \cdots + \alpha_n(P_n - P_0)$ is ok as $P_i - P_0 \in V$ and $P_0 + v \in A$

See mrl.cs.vsb.cz/people/fabian/zpg/rcs.pdf for examples

(Model) Transformation Matrix

- With homogeneous coordinates

$$M\boldsymbol{p} = \left[\begin{array}{ccc|c} & R & & \boldsymbol{t} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix}$$

where
$R \in SO(3)$ group
$M \in SE(3)$ Lie group (differentable manifold)
$\boldsymbol{t} \in \mathbb{R}^3$

- Vector $\boldsymbol{t}$ represents translation
- Matrix R represents rotation or scaling or shear or their combinations

$$R_{scaling} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}, R_{shear} = \begin{bmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

View Matrix

- We can simply setup the view matrix as follows

$$\hat{\boldsymbol{z}}_e = \frac{\boldsymbol{e}-\boldsymbol{t}}{\|\boldsymbol{e}-\boldsymbol{t}\|}, \hat{\boldsymbol{x}}_e = \frac{up \times \hat{\boldsymbol{z}}_e}{\|up \times \hat{\boldsymbol{z}}_e\|}, \text{ and } \hat{\boldsymbol{y}}_e = \hat{\boldsymbol{z}}_e \times \hat{\boldsymbol{x}}_e,$$

$$up = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

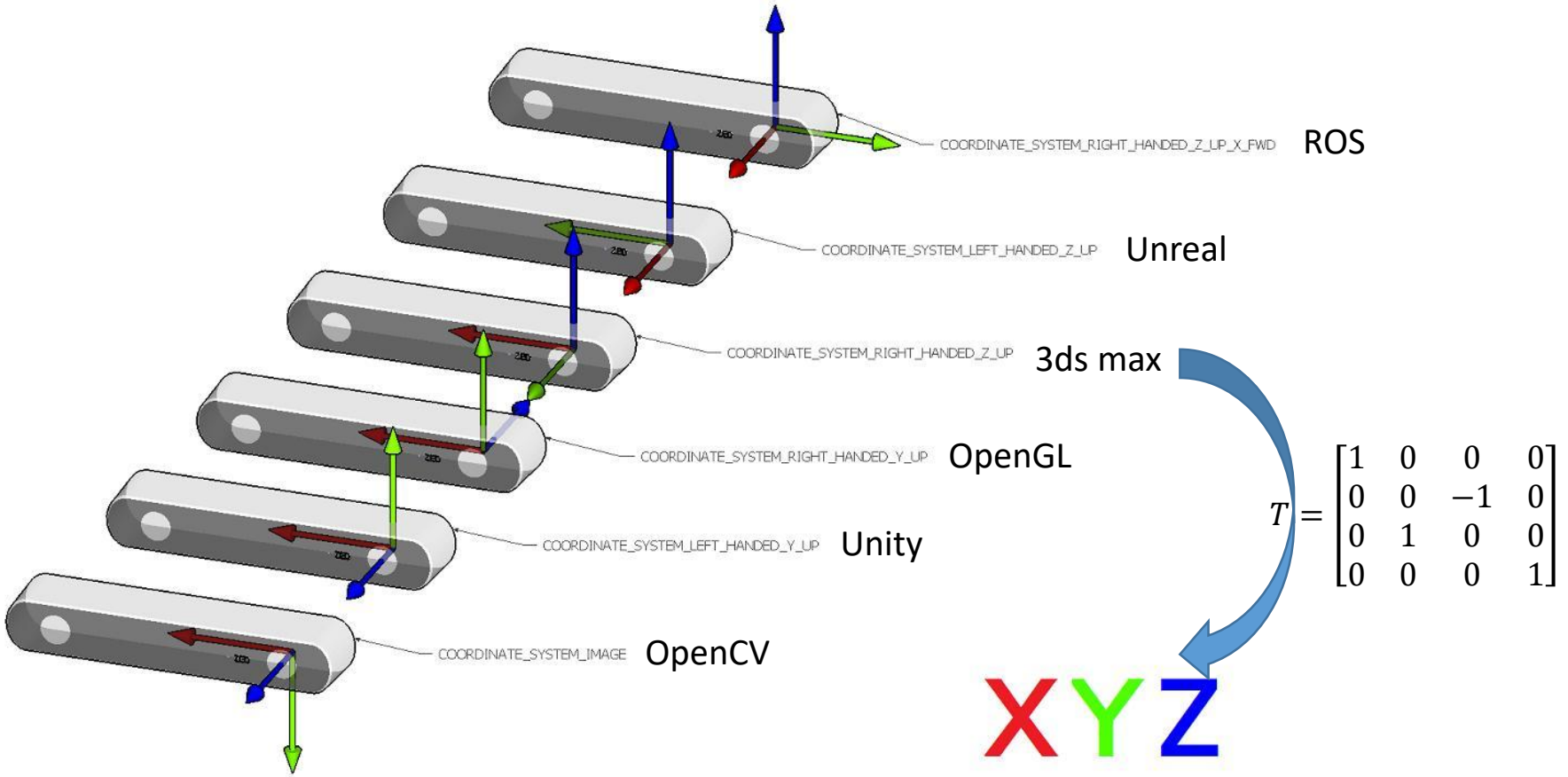
where the vector $\boldsymbol{e}$ represents position of the camera (eye), $\boldsymbol{t}$ is the target position and up is an auxiliary vector marking „up“ direction (a unit vector not parallel to the optical axis)

We can arrange the final transformation matrix

$$V^{-1} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \hat{\boldsymbol{x}}_e & \hat{\boldsymbol{y}}_e & \hat{\boldsymbol{z}}_e & \boldsymbol{e} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ (transforms vectors from eye space to world space)

- V transforms vectors from world space to eye space (and that's what we need now)

Different Coordinate Systems



Projection Matrix 1/4

- From similar triangles we get

$$\frac{v_x'}{v_z'} = \frac{v_x}{v_z} \Rightarrow v_x' = v_z' \frac{v_x}{v_z} = -n \frac{v_x}{v_z}$$

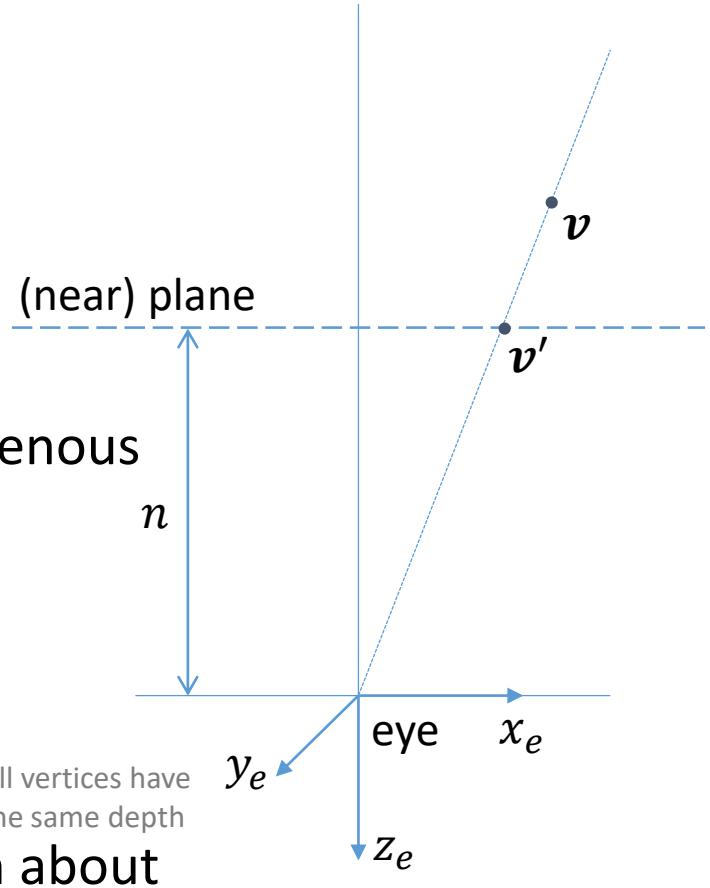
where n is a (positive) distance of (near) projection plane

This transformation can be performed by the following homogenous matrix

$$D = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}; D \begin{bmatrix} \boldsymbol{v} \\ 1 \end{bmatrix} = \begin{bmatrix} nv_x \\ nv_y \\ nv_z \\ -v_z \end{bmatrix} \Rightarrow \boldsymbol{v}' = \begin{bmatrix} n\frac{v_x}{-v_z} \\ n\frac{v_y}{-v_z} \\ -n \end{bmatrix}$$

After perspective division we get

All vertices have the same depth



Note that this perspective transformation discards information about vertex depth and we will not be able to remove hidden surfaces later

Projection Matrix 2/4

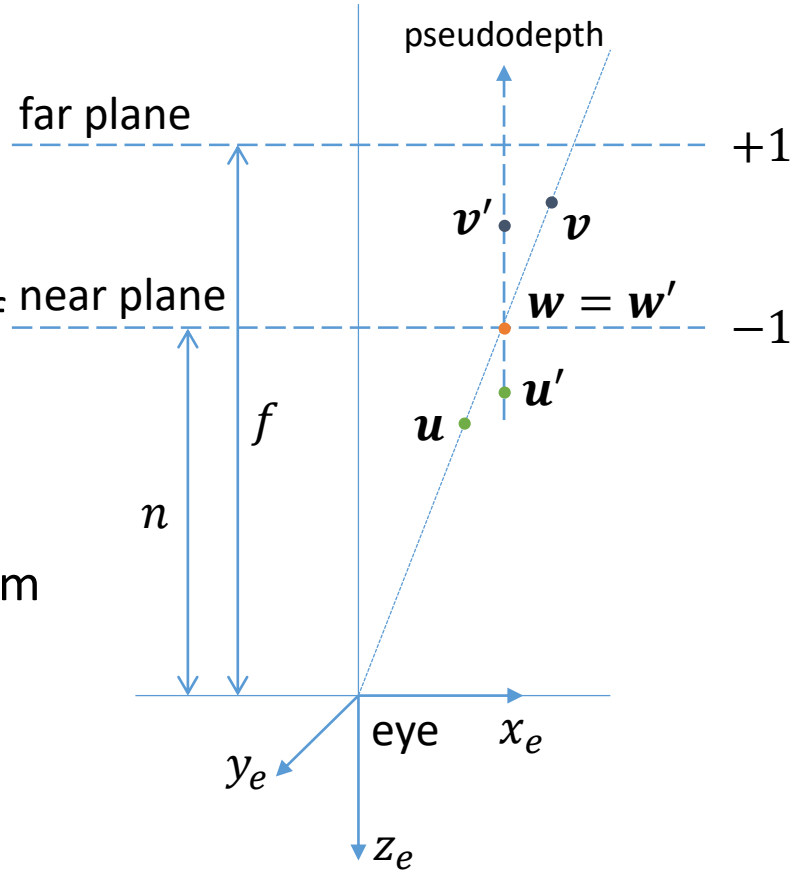
- To fix the depth loss we may incorporate pseudodepth such that points on the near plane will be given a depth of -1 and points on the far plane will be at $+1$
- We just need to solve $v_z' = \frac{av_z+b}{-v_z}$ for a and b what gives us a set of two equations as follows

$$-1 = \frac{a(-n)+b}{-(-n)}, 1 = \frac{a(-f)+b}{-(-f)}$$

Solution $a = \frac{n+f}{n-f}, b = \frac{2nf}{n-f}$ for $f \neq n$ and $fn \neq 0$ yields the new form of projective transformation matrix

$$D = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}; D\begin{bmatrix} \boldsymbol{v} \\ 1 \end{bmatrix} = \begin{bmatrix} nv_x \\ nv_y \\ av_z + b \\ -v_z \end{bmatrix} \Rightarrow \boldsymbol{v'} = \begin{bmatrix} n\frac{v_x}{-v_z} \\ n\frac{v_y}{-v_z} \\ \frac{av_z+b}{-v_z} \end{bmatrix}$$

Instead of a linear transformation of v_z (i.e. nv_z) we can use a more general affine transformation (i.e. $av_z + b$) to solve the problem with the same depth of vertices in a single line of sight.



Note that projected points are in LHS (compare the orientation of z_e and pseudodepth)

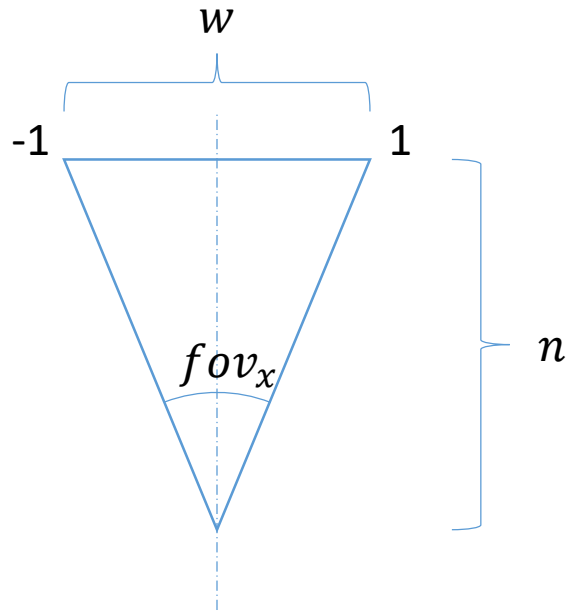
Projection Matrix 3/4

- Normalization transformation to NDC

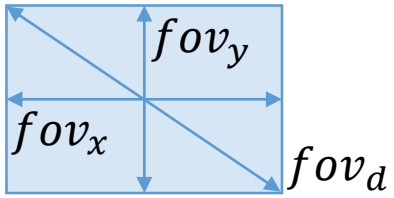
Note that the pseudo depth is already normalized

We need to normalize the actual width of the near plane w into the range of $\langle -1, 1 \rangle$

$$N = \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & \pm 2/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



where $w = 2n \tan(fov_x/2)$ and $h = 2n \tan(fov_y/2) = w/aspect$



Projection Matrix 4/4

- $P = ND = \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & \pm 2/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} =$

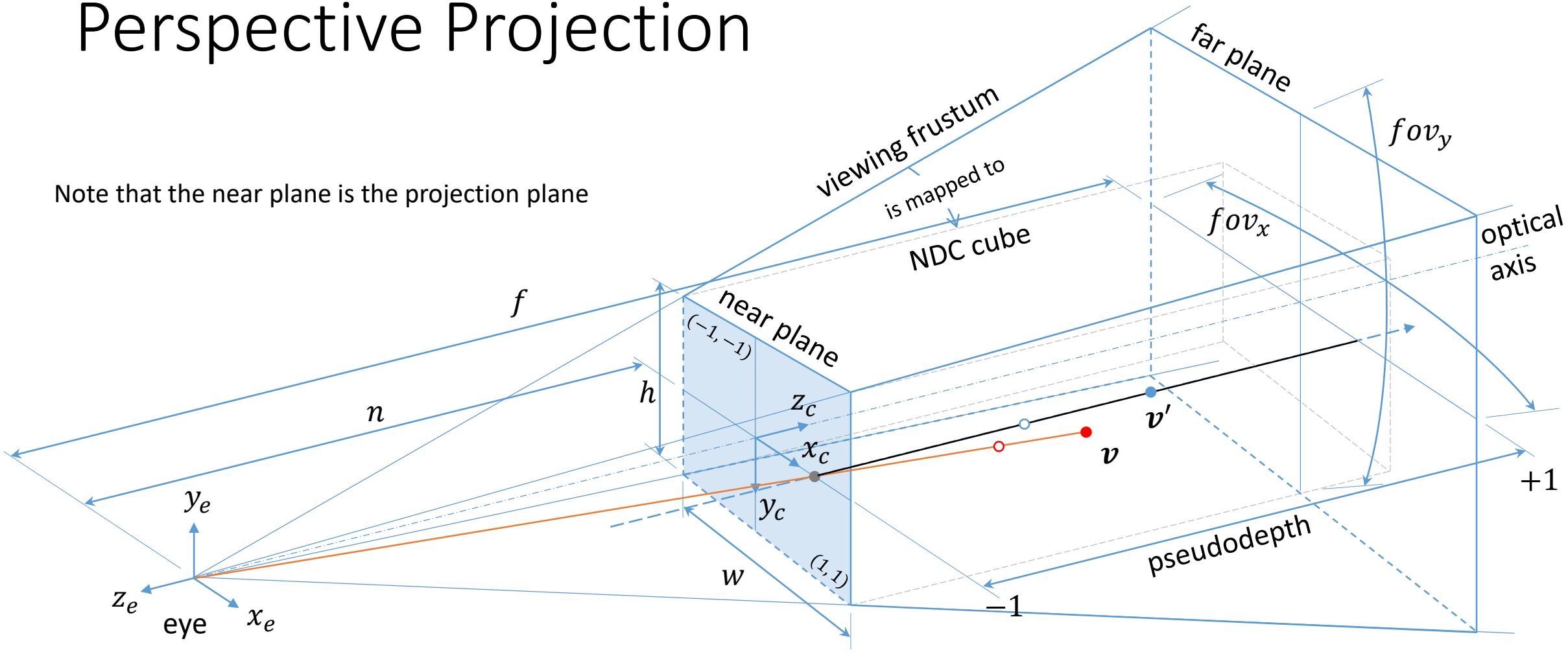
Use + when the clip control origin is lower left and – when upper left

$$= \begin{bmatrix} 1/\tan(fov_x/2) & 0 & 0 & 0 \\ 0 & \pm 1/\tan(fov_y/2) & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where $a = \frac{n+f}{n-f}$ and $b = \frac{2nf}{n-f}$. Also note that $1/\tan(fov_y/2) = aspect/\tan(fov_x/2)$ or $1/\tan(fov_x/2) = 1/(aspect\,\tan(fov_y/2))$

Perspective Projection

Note that the near plane is the projection plane



Orthographic Projection Matrix

- $P = ND_{\text{ortho}} = \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & \pm 2/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 0 & 1 \end{bmatrix}$ = Solve the set of two equations $-1 = a(-n) + b, 1 = a(-f) + b$ for a and b

Also note that w and h are just the physical dimensions of both near and far planes

$$= \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & \pm 2/h & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $a = \frac{2}{n-f}$ and $b = \frac{n+f}{n-f}$.

Example

$v := \begin{bmatrix} 1.6 \\ 0 \\ -4 \\ 1 \end{bmatrix}$ input vertex in eye-space coordinates (m)

Camera (viewing frustum) parameters

$aspect := \frac{4}{3}$ aspect ratio of the front face of the frustum

$fov\_x := 67.38\ deg$ horizontal field of view

$fov\_y := 2 \cdot \operatorname{atan}\left(aspect^{-1} \cdot \tan\left(\frac{fov\_x}{2}\right)\right) = 53.13\ deg$ vertical field of view

$n := 3 \qquad f := 5$ distances of near plane and far plane

Projection transformation

$a := \frac{f+n}{n-f} \qquad b := \frac{2 \cdot f \cdot n}{n-f}$

$M := \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & -4 & -15 \\ 0 & 0 & -1 & 0 \end{bmatrix} \qquad \frac{M \cdot v}{(M \cdot v)_3} = \begin{bmatrix} 1.2 \\ 0 \\ 0.25 \\ 1 \end{bmatrix}$

Normalization transformation

$w := 2 \cdot n \cdot \tan\left(\frac{fov\_x}{2}\right) = 4 \qquad h := 2 \cdot n \cdot \tan\left(\frac{fov\_y}{2}\right) = 3 \qquad h := w \cdot aspect^{-1} = 3$

$N := \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.667 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Perspective projection matrix

$P := N \cdot M = \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -4 & -15 \\ 0 & 0 & -1 & 0 \end{bmatrix} \qquad P \cdot v = \begin{bmatrix} 2.4 \\ 0 \\ 1 \\ 4 \end{bmatrix}$ vertex in 4D clip space coordinates

... and after perspective division we get ...

$\frac{P \cdot v}{(P \cdot v)_3} = \begin{bmatrix} 0.6 \\ 0 \\ 0.25 \\ 1 \end{bmatrix}$ output vertex in 3D NDC coordinates (-)

M should be D matrix here

Review of Coordinate Systems

Model (local, object) space

Model matrix M

World space

View matrix V

View (eye, camera) space

Projection matrix P

Clip space

Perspective division

Normalized device coordinates

Viewport transform

Screen space

Normal Vectors Transformation

- We cannot multiply MV matrix and normals as we do with vertices
- MV matrix contains translation part which will clearly affect (damage) the normal
- Normal vector $\boldsymbol{n}_{ms}$ has to be transformed in a different way, by MV_n matrix

Model-View matrix $\mathrm{MV} = View\ Model$

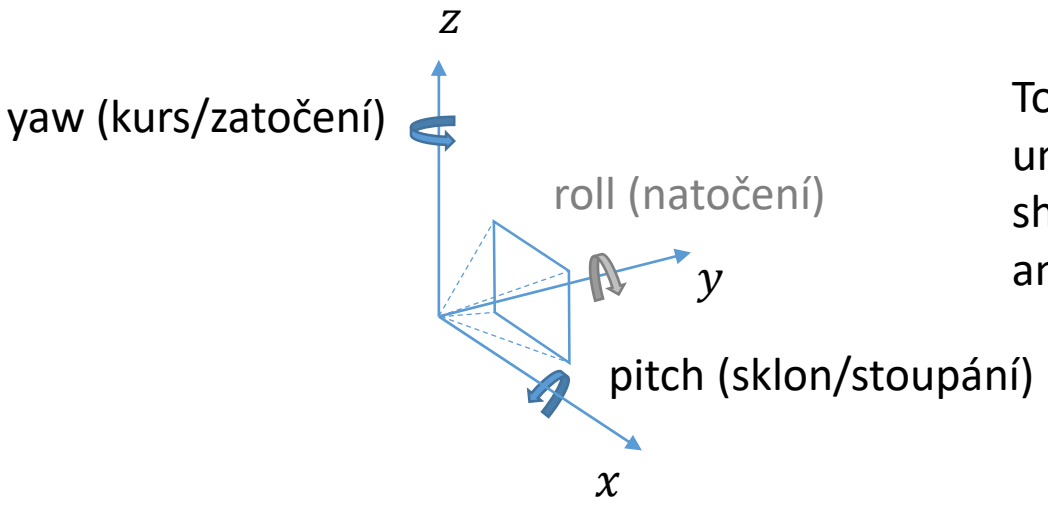
e.g. $\boldsymbol{v}_{es} = \mathrm{MV}\ \boldsymbol{v}_{ms}$

Model-View normal matrix $\mathrm{MV}_n = \left((View\ Model)^{-1}\right)^T$

e.g. $\boldsymbol{n}_{es} = \mathrm{MV}_n \boldsymbol{n}_{ms}$

Camera Movement

- In case of our camera model, all we know are vectors view from and view at (other parameters like field of view or resolution are unimportant here)
- We may use two angles (yaw and pitch) to describe the rotation of our camera around the point view from (our eye). We will not allow camera roll



To avoid gimbal lock and unwanted view flipping, we should restrict the range of pitch angle, e.g. $\pm 80°$

Camera Movement

- The question is how to initialize these two angles from known view from and view at vectors
- If we assume our "zero" rotation position heading toward y-axis, we may compute yaw and pitch angle as follows

```
yaw = atan2f( view_dir.y, view_dir.x ) – M_PI_2;
pitch = acosf( -view_dir.z ) – M_PI_2;
```

- Note that the view_dir is normalized viewing direction vector.

Camera Movement

- After change of any angle, we need to update view at vector.
- To do so, we pitch our initial "zero" rotation vector around x-axis and then we apply yaw around z-axis
- Updating the view at vector is straightforward and the whole process can be summarized as follows

```
Vector3 new_view_dir = Rz( yaw ) * Rx( pitch ) * Vector3( 0, 1, 0 );
new_view_dir.Normalize();
view_at = view_from + new_view_dir;
```

Mouse and Keyboard Inputs in GLFW

- GLFW provides many kinds of input.
- All input callbacks receive a window handle allowing us to reference user pointer (e.g. our class Rasterizer) from callbacks

```
glfwSetWindowUserPointer/ glfwGetWindowUserPointer
```

- Note that the key press event is reported only once and the repeat event occurs after a while what induces a delay.
- Use raw mouse motion (as well as hidden cursor) for better control of the camera rotation
- For further reference, see https://www.glfw.org/docs/3.3/input_guide.html

OpenGL

- Open Graphics Library for rendering 2D and 3D vector graphics
- Modern GPUs accelerate almost all OpenGL operations to achieve real-time framerates
- API released by SGI (OpenGL Architecture Review Board ARB) in 1992
- Since 2006 managed by the consortium Khronos Group
- Multiplatform, cross-language, client-server (same or different address space or computer)
- HW vendor extensions are possible
- Current version is 4.6 (core profile, compatibility profile, shading language GLSL 4.60)
- https://www.khronos.org/registry/OpenGL/index_gl.php

OpenGL

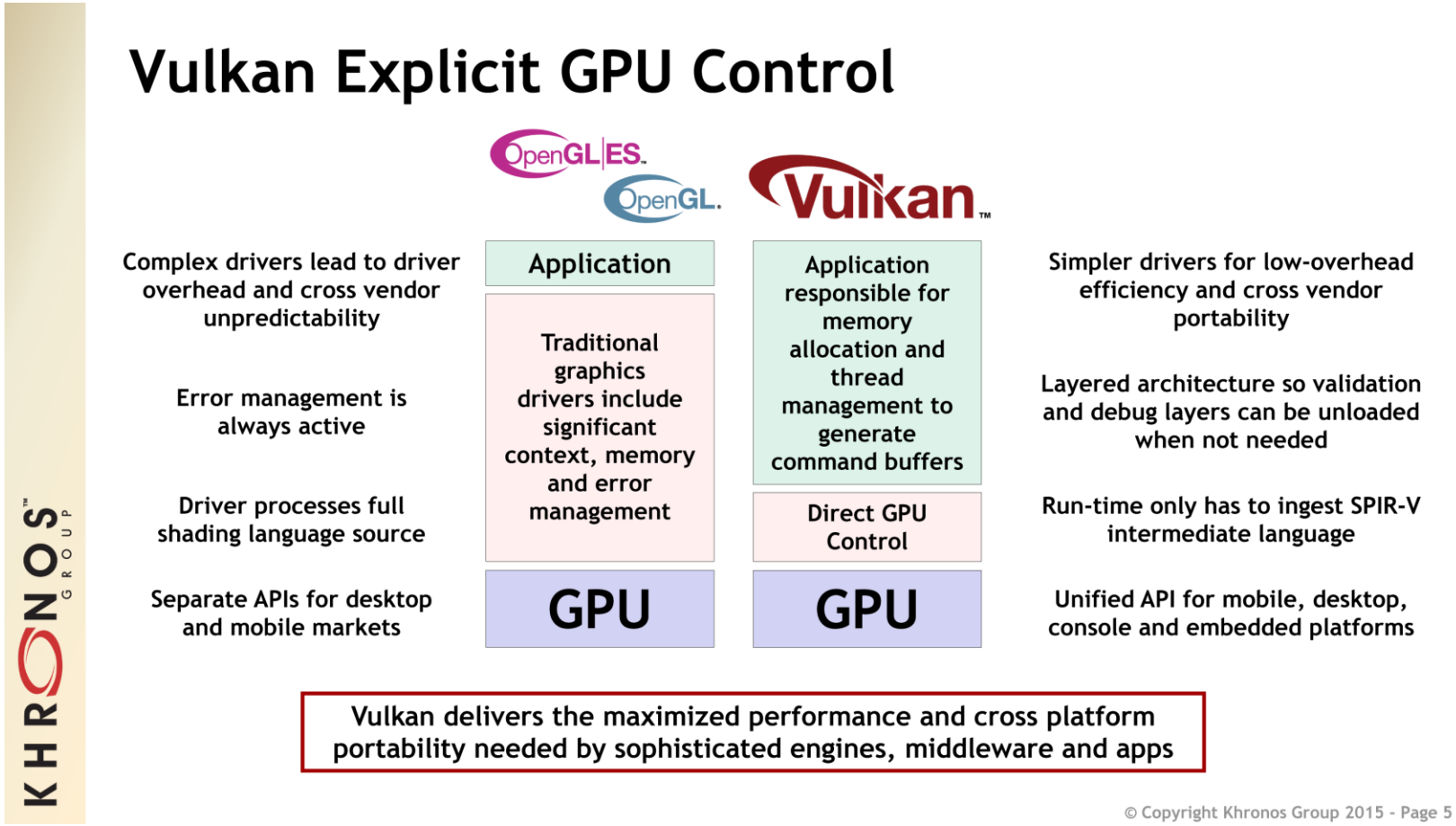
- 1.0 (1992) – first release
- 1.1 (1997) – texture objects
- 1.2 (1998) – 3D textures, BGRA
- 1.2.1 (1998) – ARB extension concept
- 1.3 (2001) – multitexturing
- 1.4 (2002) – depth textures
- 1.5 (2003) – vertex buffer objects
- 2.0 (2004) – shader objects (GLSL)
- 2.1 (2006) – pixel buffer objects, sRGB
- 3.0 (2008) – frame buffer objects
- 3.1 (2009) – instancing, TBO, UBO
- 3.2 (2009) – geometry shader
- 3.3 (2010)
- 4.0 (2010) – tessellation
- 4.1 (2010)
- 4.2 (2011) – atomic counters
- 4.3 (2012) – debug output*
- 4.4 (2013) – bindless textures
- 4.5 (2014) – additional clip control
- 4.6 (2017) – SPIR-V language

For further reference see https://www.khronos.org/opengl/wiki/History_of_OpenGL
(*) https://www.khronos.org/opengl/wiki/Debug_Output

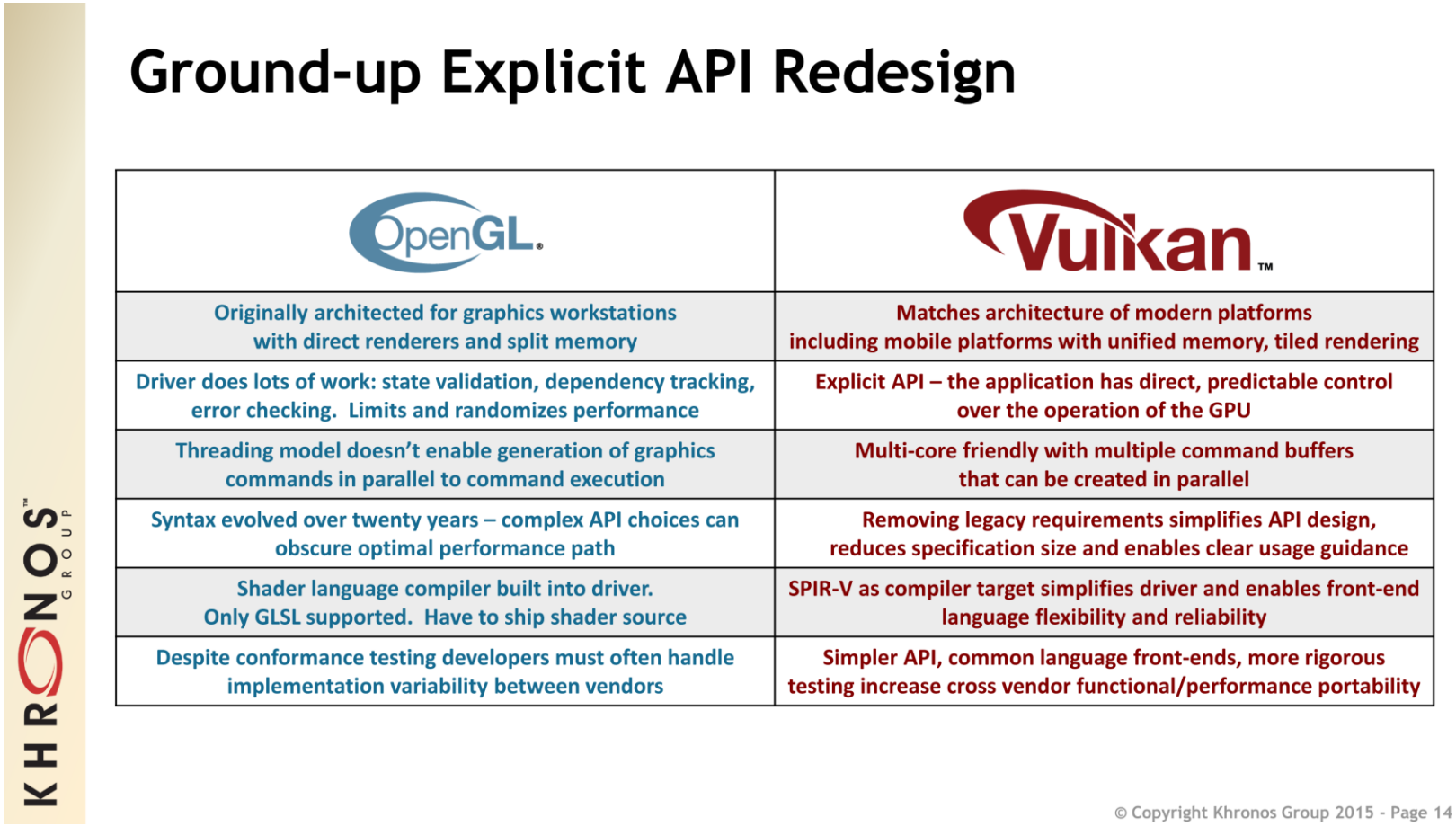
Other Graphics APIs

- SGI IRIS GL, 3dfx Glide, Microsoft DirectX 12, Apple Metal 3, AMD Mantle, Vulkan 1.2 (initially released in 2016)

OpenGL vs Vulkan



OpenGL vs Vulkan



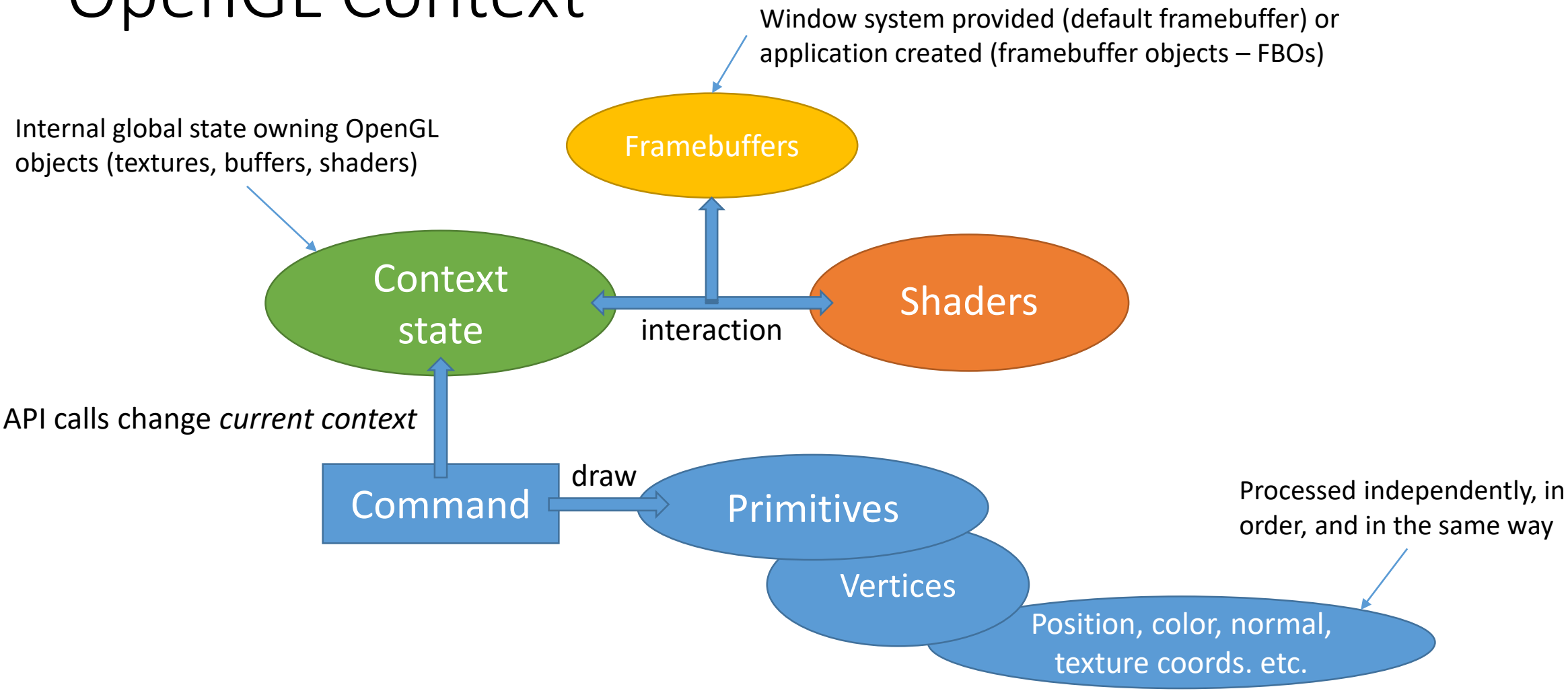
OpenGL

- OpenGL is a pipeline concerned with processing data in GPU memory
  - programmable stages
  - state driven fixed-function stages
- OpenGL ES (subsets of OpenGL + some specific functionality) is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems such as mobile phones, game consoles, and vehicles
- WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES
- SPIR-V is a binary intermediate language for representing graphical-shader stages and compute kernels for multiple Khronos APIs, such as OpenCL, OpenGL, and Vulkan

OpenGL

- OpenCL is an open, royalty-free standard for cross-platform, general purpose parallel programming of processors found in personal computers, servers, and mobile devices, including GPUs.
  - interop methods to share OpenCL memory and image objects with corresponding OpenGL buffer and texture objects

OpenGL Context



OpenGL Context

- OpenGL is a state machine (collection of variables)
- It‘s current state is referred to as the OpenGL context
- C-library API consist of state-changing functions

```
struct Object {
  int option_1;
  float option_2;
};

struct OpenGLContext {
  ObjectName * object_Target = 0;
} opengl_context;

GLuint object_id = 0;
glGenObject( 1, &object_id );
glBindObject( GL_TARGET, object_id ); // bind before usage
// set the properties of object currently bound to the given target
glSetObjectOption( GL_TARGET, GL_OPTION_1, 123 );
glSetObjectOption( GL_TARGET, GL_OPTION_2, 3.14 );
glBindObject( GL_TARGET, 0 ); // set context target back to default
glDelete( 1, &object_id );
```

OpenGL allows binding to several buffers at once as long as they have a different type/target

Phong Reflection Model

- Designed by Bui Tuong Phong in 1975
- Baseline shading method for many rendering applications

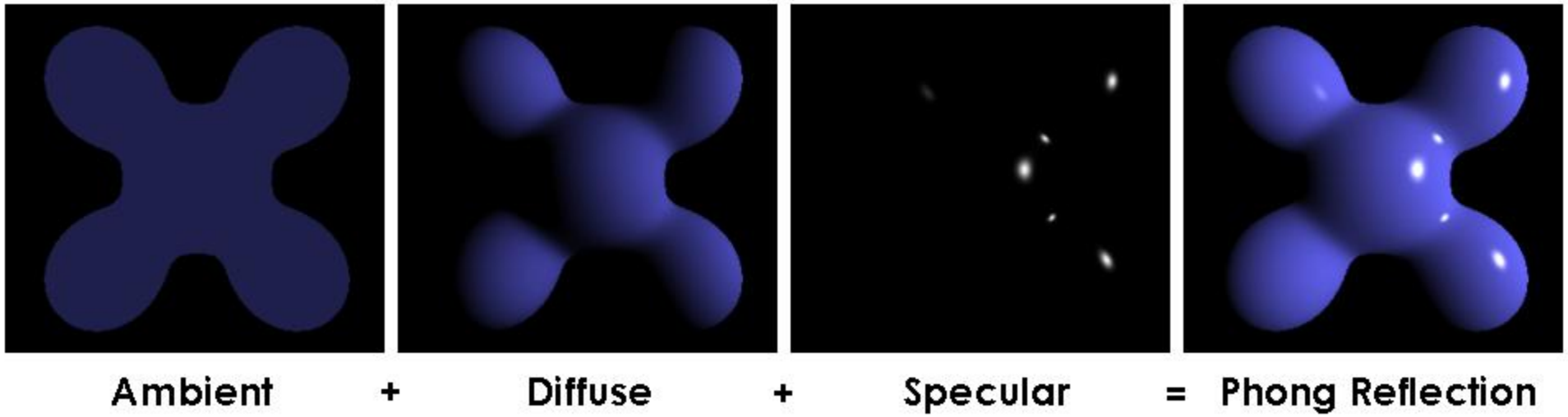
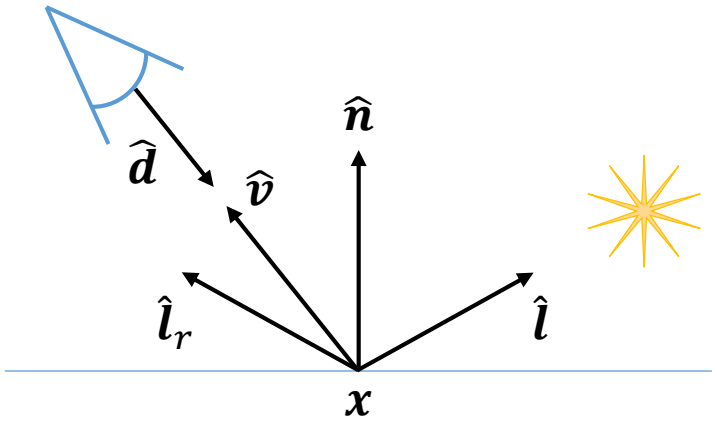


Illustration of the components of the Phong reflection model (Ambient, Diffuse and Specular reflection)
Source: Brad Smith

Phong Reflection Model

- Original definition

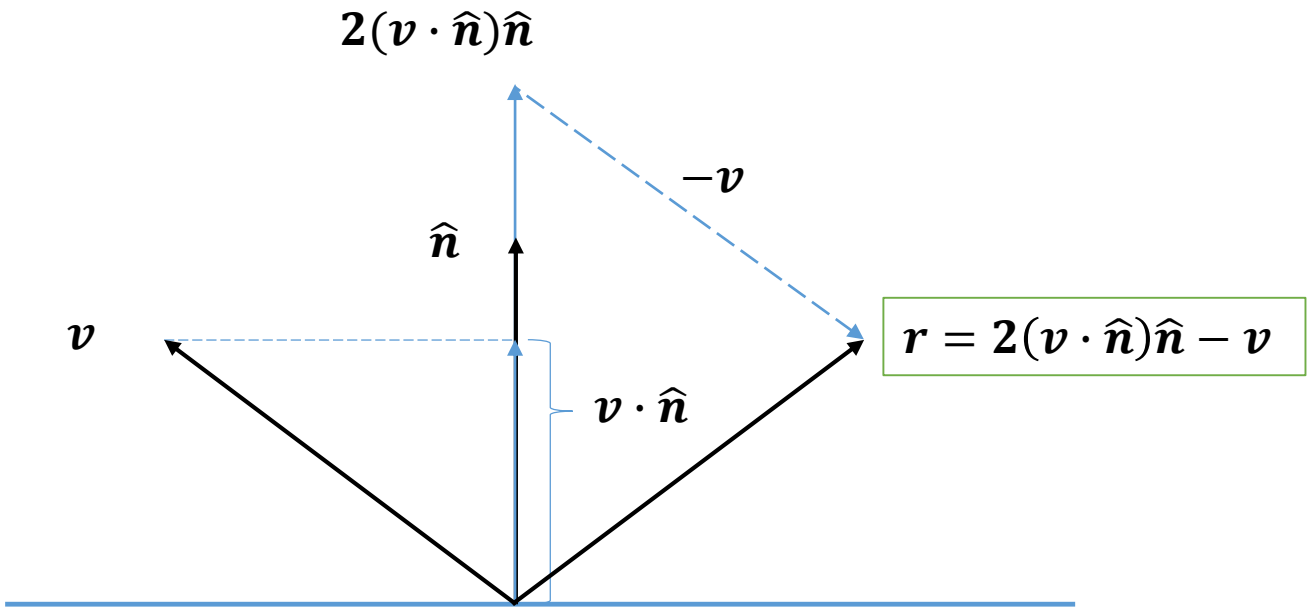


$$\boldsymbol{C} = \boldsymbol{I}_a\boldsymbol{m}_a + \sum_{visible\ lights}(\boldsymbol{I}_d\boldsymbol{m}_d(\hat{\boldsymbol{n}}\cdot\hat{\boldsymbol{l}}) + \boldsymbol{I}_s\boldsymbol{m}_s(\hat{\boldsymbol{v}}\cdot\hat{\boldsymbol{l}}_r)^{\gamma}) \text{ where}$$

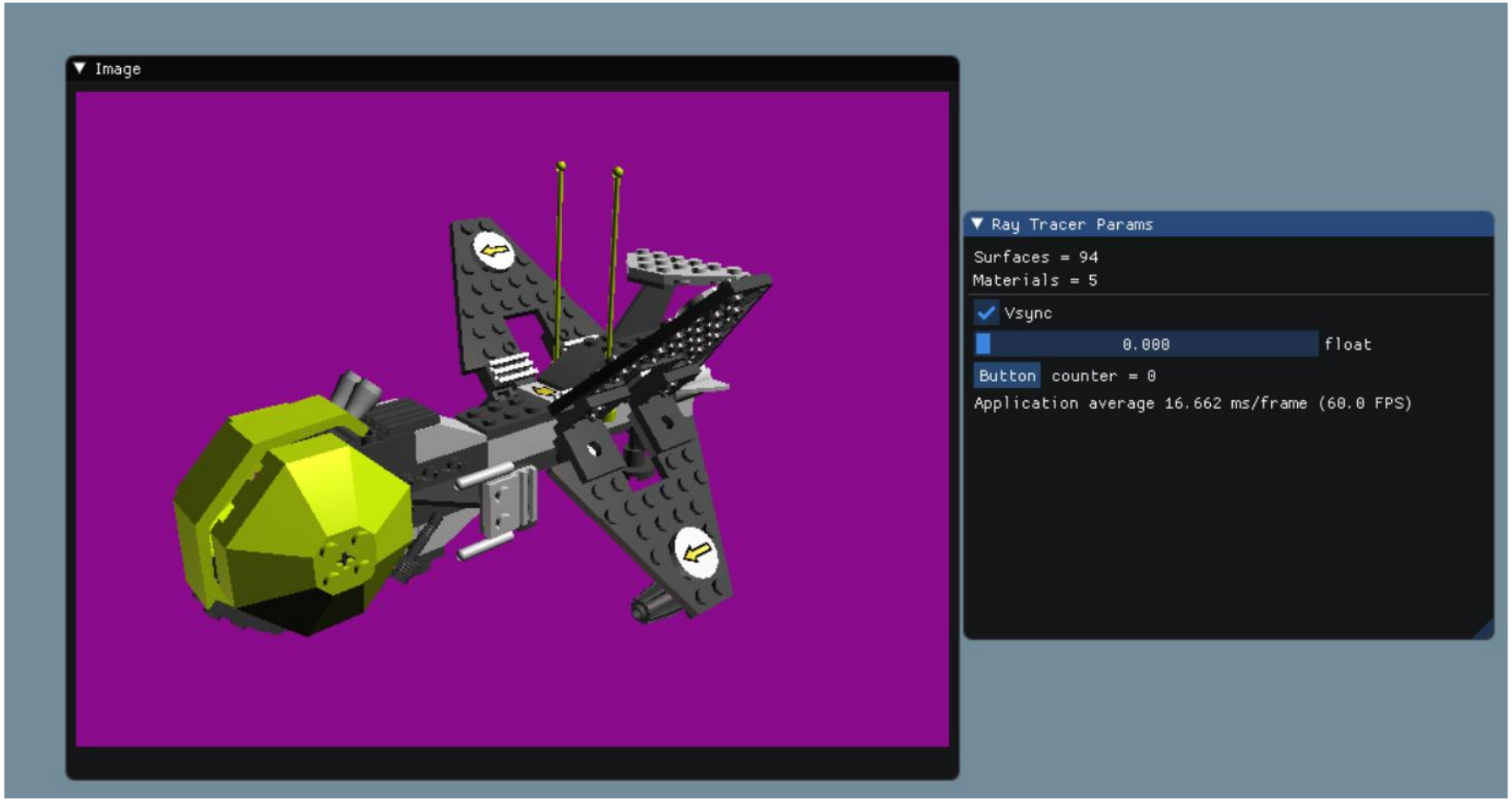
$$\hat{\boldsymbol{l}}_r = 2(\hat{\boldsymbol{n}}\cdot\hat{\boldsymbol{l}})\hat{\boldsymbol{n}} - \hat{\boldsymbol{l}}$$

For a white source set $\boldsymbol{I}_a = \boldsymbol{I}_d = \boldsymbol{I}_s = (1,1,1)$. You can also extend the model with attenuation based on distance from the source.
The material definition includes ambient, diffuse, and specular colors, and a real value γ called shininess

Reflection



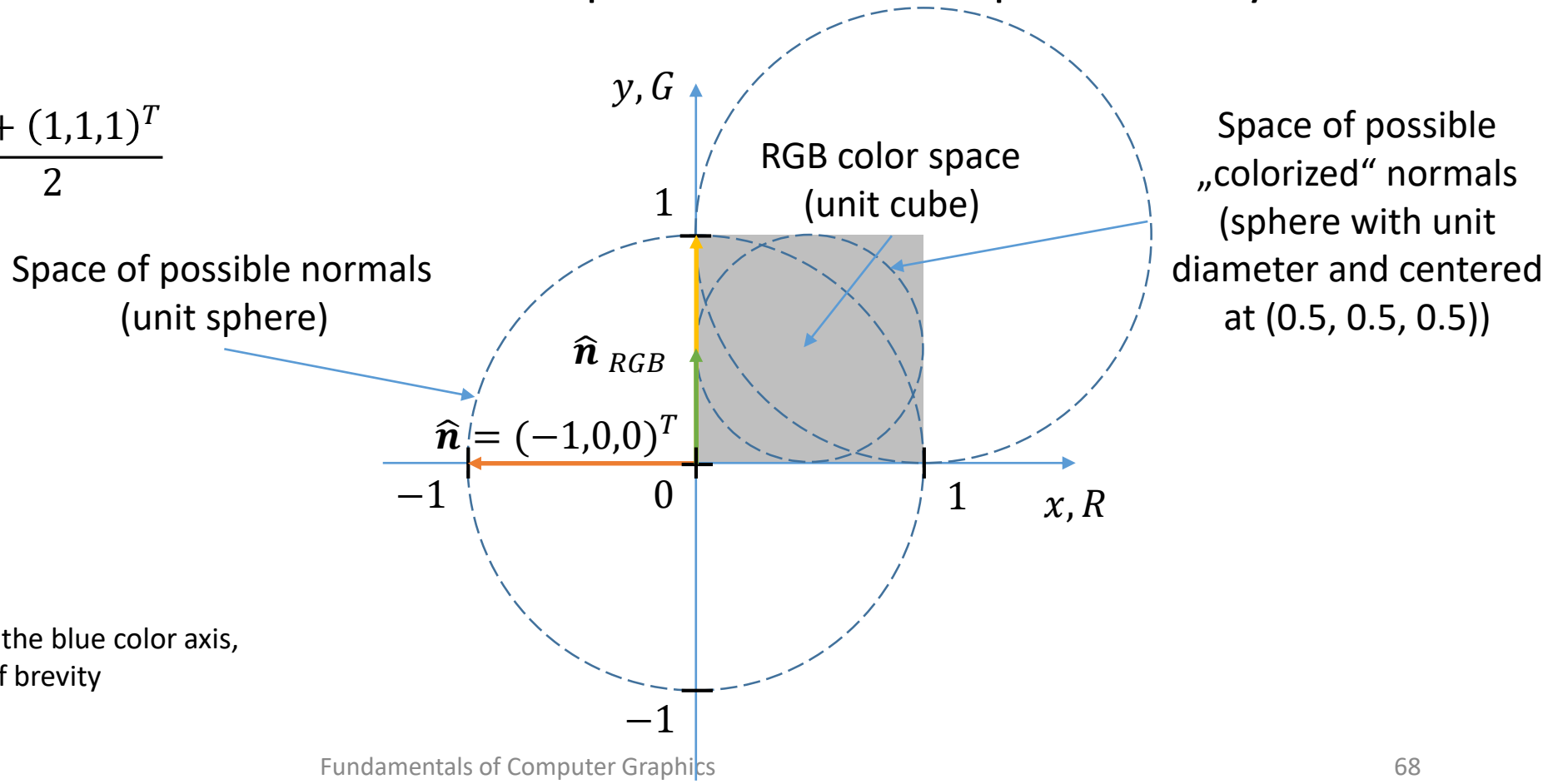
Result of Second Exercise



Normal Shader

- We can easily see that normal cannot be represented in RGB space directly

$$\hat{\boldsymbol{n}}_{RGB} = \frac{\hat{\boldsymbol{n}} + (1,1,1)^T}{2}$$



Note that the z-axis, resp. the blue color axis, are omitted for the sake of brevity

RGB vs sRGB Color Spaces

- RGB color space is any additive color space based on RGB color model that employs RGB primaries (i.e. red, green, and blue chromacities)
- Primary colors are defined by their CIE 1931 color space chromacity coordinates (x, y)
- Specification of any RGB color space also includes definition of white point and transfer function
- sRGB is one of many (but by far the most commonly used) color spaces for computer displays
- Our renderer will produce sRGB images

Specifications of RGB color spaces

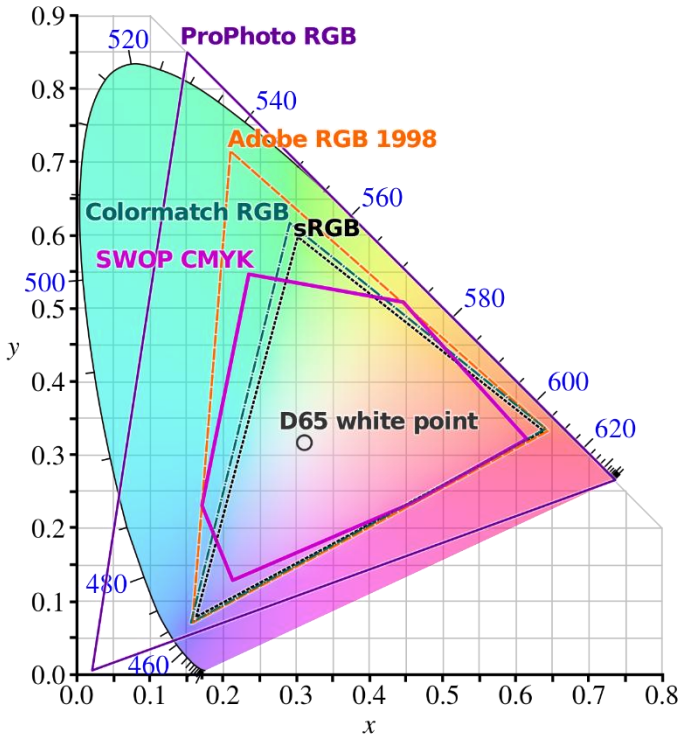
Color space	Standard	Year	Gamut	White point	Primaries						Transfer function parameters				
					Red		Green		Blue		α	β	γ	δ	βδ
					x_R	y_R	x_G	y_G	x_B	y_B	a + 1	$K_0/\varphi = E_t$		φ	K_0
ISO RGB			Limited	floating	floating										
Extended ISO RGB			Unlimited (signed)												
scRGB	IEC 61966-2-2	2003													
sRGB	IEC 61966-2-1	1990, 1996		D65	0.64	0.33	0.30	0.60	0.15	0.06	1.055	0.0031308	$\frac{12}{5}$	12.92	0.04045
HDTV	ITU-R BT.709	1999	CRT	D65	0.64	0.33	0.30	0.60	0.15	0.06	1.099	0.004	$\frac{20}{9}$	4.5	0.018
Adobe RGB 98		1998					0.21	0.71			1	0	$\frac{563}{256}$	1	0
PAL / SECAM	EBU 3213-E, ITU-R BT.470/601 (B/G)	1970					0.29	0.60			1	0	$\frac{14}{5}$	1	0
Apple RGB					0.625	0.34	0.28	0.595	0.155	0.07					
NTSC	SMPTE RP 145 (C), 170M, 240M	1987			0.63		0.31				1.1115	0.0057	$\frac{20}{9}$	4	0.0228
NTSC-J		1987		D93											
NTSC (FCC)	ITU-R BT.470/601 (M)	1953		C	0.67	0.33	0.21	0.71	0.14	0.08	1	0	$\frac{11}{5}$	1	0
eciRGB	ISO 22028-4	1999 (v1), 2007, 2012	Wide	D50							1.16	0.008856	3	9.033	0.08
DCI-P3	SMPTE RP 431-2	2011		Theater	0.68	0.32	0.265	0.69	0.15	0.06	1.055	0.0031308	$\frac{12}{5}$	12.92	0.04045
Display P3	SMPTE EG 432-1	2010		D65											
UHDTV	ITU-R BT.2020, BT.2100	2012, 2016			0.708	0.292	0.170	0.797	0.131	0.046	1.0993	0.018054		4.5	0.081243
Adobe Wide Gamut RGB				D50	0.735	0.265	0.115	0.826	0.157	0.018	1	0	$\frac{563}{256}$	1	0
RIMM	ISO 22028-3	2006, 2012			0.7347	0.2653	0.1596	0.8404	0.0366	0.0001	1.099	0.0018	$\frac{20}{9}$	5.5	0.099
ROMM RGB, ProPhoto RGB	ISO 22028-2	2006, 2013									1	0.001953125	$\frac{9}{5}$	16	0.031248
CIE RGB		1931		E			0.2738	0.7174	0.1666	0.0089					
CIE XYZ		1931	Unlimited		1	0	0	1	0	0	1	0	1	1	0

Source: https://en.wikipedia.org/wiki/RGB_color_space

Color Gamut

- A color gamut is defined as a range of colors that a particular device is capable of displaying or recording
- It usually appears as a closed area of primary colors in a chromaticity diagram. The missing dimension is the brightness, which is perpendicular to the screen or paper
- Color gamut is displayed as a triangular area enclosed by color coordinates corresponding to the red, green, and blue color

CIE 1931 XYZ color space



Color Gamut

- sRGB - by far the most commonly used color space for computer displays
- NTSC – standard for analog television
- Adobe RGB (1998) - encompass most of the colors achievable on CMYK color printers, but by using RGB primary colors on a device such as a computer display
- DCI-P3 – space for digital movie projection from the American film industry
- EBU – European color space surpassing the PAL standard
- Rec. 709 - shares the sRGB primaries, used in HDTVs
- Rec. 2025 – 4K or 8K resolution at 10 or 12 bits per channel
- Remember that 72% NTSC is not sRGB (which is often claimed). Matching the ratios of the color gamut areas does not necessarily guarantee the ability to achieve the same image (the degree of overlap of the triangles is important and not the ratio of their areas).

Linear sRGB and Gamma Compressed sRGB

- Images displayed on monitors are encoded in nonlinear sRGB color space to compensate the transformation of brightness the monitor does
- Our render has to work in linear space as we are using linear operations with colors
- Every texel and material color have to be processed in linear sRGB color model
- Only the final color values stored in framebuffer are converted back to gamma compensated sRGB values
- Issue with blending two sRGB colors in non-linear color space:

$\boldsymbol{C}_{srgb} = \alpha \boldsymbol{A}_{srgb} + (1-\alpha)\boldsymbol{B}_{srgb}$ doesn work well

$\boldsymbol{C}_{rgb} = \alpha \boldsymbol{A}_{rgb} + (1-\alpha)\boldsymbol{B}_{rgb}$ correct

$\boldsymbol{C}_{srgb} = ToSRGB\left(\alpha\, ToRGB\left(\boldsymbol{A}_{srgb}\right) + (1-\alpha)\, ToRGB\left(\boldsymbol{B}_{srgb}\right)\right)$ correct

Note the undesirable dark silhouettes when mixing two colors directly in sRGB space (created in paint.net)

Gamma Correction

- The human visual system response is logarithmic, not linear, resulting in the ability to perceive an incredible brightness range of over 10 decades
- Gamma characterizes the reproduction of tone scale in an imaging system. Gamma summarizes, in a single numerical parameter, the nonlinear relationship between code value (in an 8-bit system, from 0 through 255) and luminance. Nearly all image coding systems are nonlinear, and so involve values of gamma different from unity
- The main purpose of gamma correction is to code luminance into a perceptually-uniform domain, so as optimize perceptual performance of a limited number of bits in each channel

Source: POYNTON, Charles A. The rehabilitation of gamma, 2000.

sRGB Transfer Functions

- Function returns gamma-expanded (or linear) sRGB values from gamma-compressed (or non-linear) sRGB values

```
float c_linear( float c_srgb, float gamma = 2.4f )
{
  if ( c_srgb <= 0.0f ) return 0.0f;
  else if ( c_srgb >= 1.0f ) return 1.0f;

  assert( ( c_srgb >= 0.0f ) && ( c_srgb <= 1.0f ) );

  if ( c_srgb <= 0.04045f )
  {
    return c_srgb / 12.92f;
  }
  else
  {
    const float a = 0.055f;
    return powf( ( c_srgb + a ) / ( 1.0f + a ), gamma );
  }
}
```

- Function returns gamma-compressed (or non-linear) sRGB values from gamma-expanded (or linear) sRGB values

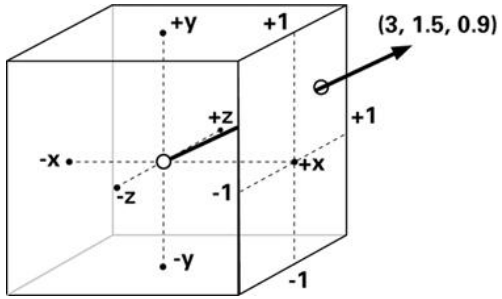
```
float c_srgb( float c_linear, float gamma = 2.4f )
{
  if ( c_linear <= 0.0f ) return 0.0f;
  else if ( c_linear >= 1.0f ) return 1.0f;

  assert( ( c_linear >= 0.0f ) && ( c_linear <= 1.0f ) );

  if ( c_linear <= 0.0031308f )
  {
    return 12.92f * c_linear;
  }
  else
  {
    const float a = 0.055f;
    return ( 1.0f + a )*powf( c_linear, 1.0f / gamma ) - a;
  }
}
```

Tone-mapping

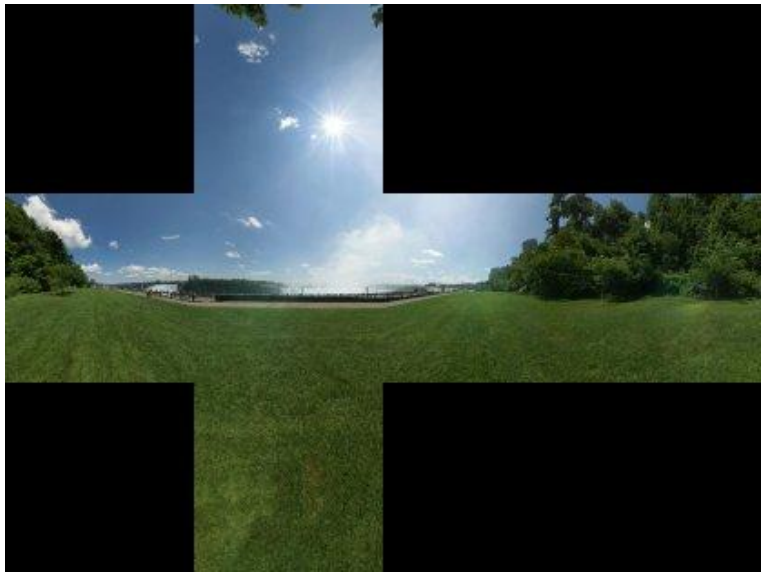
- A process which maps an input image of high dynamic range (HDR) to a limited low dynamic range (LDR)
- Typical output devices such as LCD monitors have LDR (i.e. accept values with a very narrow range of $\langle 0, 1 \rangle$)
- A ray tracer produces linear HDR outputs with a potentially unlimited range of $\langle 0, \infty)$
- We need to scale down these HDR values into LDR somehow
- We can use various (tone)-mapping operators (or functions) to do so

Background (Environment) Images

- Cube map – six square textures $\{\pm x, \pm y, \pm z\}$
- Firstly, select the one of the six maps

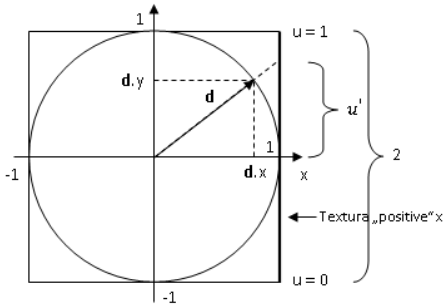


Based on the index of largest component in absolute value of directional vector $\hat{\boldsymbol{d}}$ select the one of the six square maps.



Source: http://www.humus.name

- Secondly, compute u and v coordinates



From similar triangles we can easily see that $\frac{u'}{1} = \frac{d.y}{d.x}$ and after the normalization of u' we get $u = \frac{u'+1}{2}$. The same holds for v coordinate.

Textures Bilinear Interpolation

- Colors (or other values) obtained from textures should be interpolated using bilinear interpolation at least

Nearest neighbor interpolation

1 spp at pixel center

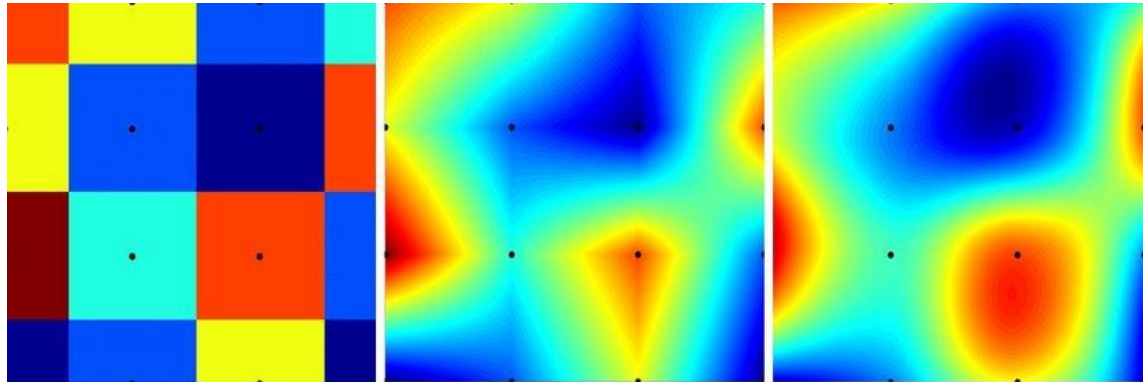
1 random spp

Bilinear interpolation

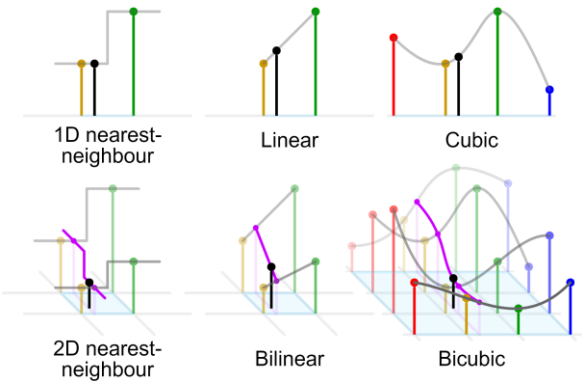
1 spp at pixel center

Bilinear Interpolation

- Number of required samples
  - Nearest neighbor: 1 sample
  - 2D bilinear interpolation: 2×2=4 samples
  - 2D bicubic interpolation: 4×4=16 samples



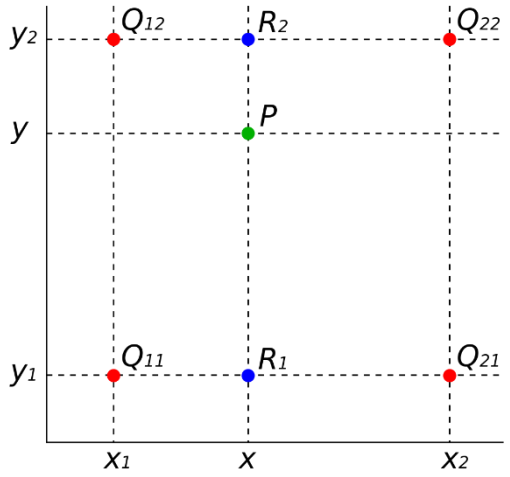
Nearest neighbor Bilinear Bicubic



Source: https://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear Interpolation

- For more detailed explanation refer to the link below



$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

$$\boxed{f(x, y)} \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

$$= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right)$$

$$= \frac{1}{(x_2 - x_1)(y_2 - y_1)} (f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1))$$

This is the resulting value of interpolated quantity at the point P

Source: https://en.wikipedia.org/wiki/Bilinear_interpolation

Composition

- In the case of the image from the previous slide, the color $\boldsymbol{C}^i$ from i-th hit point on opaque dielectric surface is computed as follows

$$\boldsymbol{C}^i = \boldsymbol{C}_{Phong} + \boldsymbol{C}_{refl}\, R(\theta_i)$$

where

Color of the light source, e.g. $(1, 1, 1)$

$$\boldsymbol{C}_{Phong} = \boldsymbol{l}_c \left(\boldsymbol{m}_a + \vartheta(\boldsymbol{p}, \boldsymbol{l}_p) \left(\boldsymbol{m}_d (\hat{\boldsymbol{l}}_d \cdot \hat{\boldsymbol{n}}) + \boldsymbol{m}_s (\hat{\boldsymbol{l}}_r \cdot \hat{\boldsymbol{v}})^{\gamma} \right) \right),$$

Visibility function between the hit point $\boldsymbol{p}$ and the position $\boldsymbol{l}_p$ of an omni light

$\boldsymbol{C}_{refl}$ is color returned by the reflected ray, and R is Schlick's approximation of hit point reflectivity

Composition

6887_allied_avenger.mtl

```
newmtl white_phong
  Ns 20
  Ni 1.460
  Ka 0.01 0.01 0.01
  Kd 0.95 0.95 0.95
  Ks 0.8 0.8 0.8
  shader 3

newmtl black_phong
  Ns 20
  Ni 1.460
  Ka 0.01 0.01 0.01
  Kd 0.1 0.1 0.1
  Ks 0.8 0.8 0.8
  shader 3

newmtl white_phong_4150p04
  Ns 20
  Ni 1.460
  Ka 0.01 0.01 0.01
  Kd 0.95 0.95 0.95
  Ks 0.8 0.8 0.8
  map_Kd 4150p04.jpg
  shader 3

newmtl white_phong_3069bp13
  Ns 20
  Ni 1.460
  Ka 0.01 0.01 0.01
  Kd 0.95 0.95 0.95
  Ks 0.8 0.8 0.8
  map_Kd 3069bp13.jpg
  shader 3

newmtl green_glass
  Tf 0.4 0.001 0.4
  Ni 1.5
  shader 4
```

Notes:

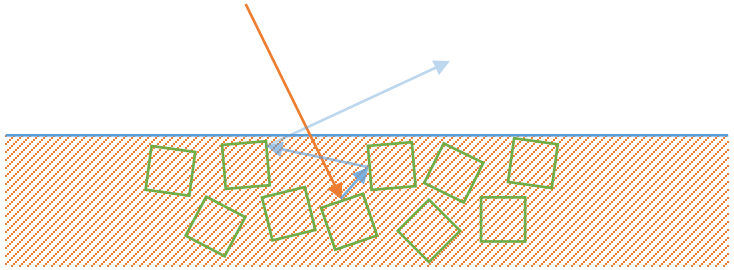
Ka, *Kd*, and *Ks* are treated as RGB values stored in sRGB gamma compressed space to match values stored in texture files

Tf (in case of shader 4 - glass) is treated as RGB value representing attenuation coefficient

Recursion depth is set to 10

Diffuse Material

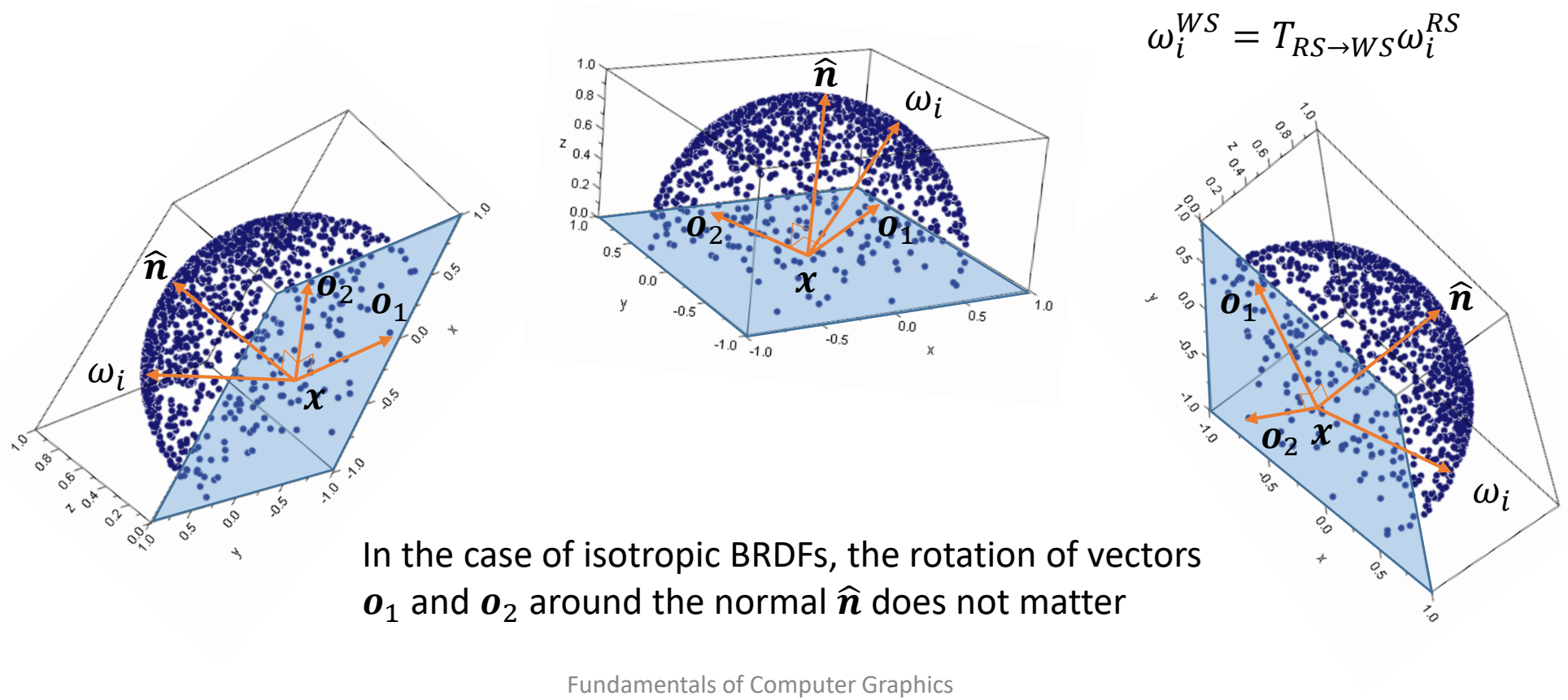
- Incident ray is scattered at many angles
- Ideal diffuse material is said to be Lambertian = equal luminance (radiance) when viewed from all directions lying in „upper“ hemisphere
- Good examples of solid diffuse reflectors are plaster, paper, or polycrystalline materials (exhibit subsurface scattering mechanism caused by internal subdivisions)
- Few materials do not cause diffuse reflection: metals (do not allow light to enter), gases, liquids, glass, and transparent plastics



Local Reference Frame

- Hemisphere samples generated in RS must be (at some point in the ray tracing pipeline) transformed to WS

$$\omega_i^{WS} = T_{RS \to WS} \omega_i^{RS}$$



In the case of isotropic BRDFs, the rotation of vectors $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$ around the normal $\hat{\boldsymbol{n}}$ does not matter

Local Reference Frame

- Derive transformation matrix (RS → WS) from surface normal $\hat{\boldsymbol{n}}$
  - Vector $\hat{\boldsymbol{n}}$ and **any non-parallel** vector $\boldsymbol{a}$ define a plane (we assume that the plane is passing through the origin)
  - This plane has normal $\hat{\boldsymbol{o}}_2$ such that $\hat{\boldsymbol{o}}_2 = \hat{\boldsymbol{n}} \times \boldsymbol{a}$ and by definition, the vector $\hat{\boldsymbol{o}}_2$ is perpendicular to $\hat{\boldsymbol{n}}$. The remaining question is how to construct such a vector $\boldsymbol{a}$?

```
inline Vector3 orthogonal( const Vector3 & n )
{
    return ( abs( n.x ) > abs( n.z ) ) ? Vector3( n.y, -n.x, 0.0f ) : Vector3( 0.0f, n.z, -n.y );
}
```

$\hat{\boldsymbol{o}}_2$ where $\boldsymbol{a} = (0,0,1)$ $\hat{\boldsymbol{o}}_2$ where $\boldsymbol{a} = (1,0,0)$

  - The remaining third axis can be computed as $\hat{\boldsymbol{o}}_1 = \hat{\boldsymbol{o}}_2 \times \hat{\boldsymbol{n}}$ yelding vector perpendicular to both $\hat{\boldsymbol{o}}_2$ and $\hat{\boldsymbol{n}}$
  - Now we can construct a change-of-basis matrix $T_{RS \to WS}$ that transforms vector in the reference (local) space (RS) to the world space (WS)

Local Reference Frame

$$T_{RS\to WS} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \hat{\boldsymbol{o}}_1 & \hat{\boldsymbol{o}}_2 & \hat{\boldsymbol{n}} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

- Inverse transformation can be computed as follows
$$T_{WS\to RS} = T_{RS\to WS}^{-1}$$
- Moreover, the matrix $T_{RS\to WS}$ belongs to a special orthogonal group $SO(3)$, also called the 3D rotation group (matrices of orthonormal basis) for which holds that $QQ^T = I$ for every $Q \in SO(n)$. Also note that for any nonsingular A: $AA^{-1} = I$
- This property allows us to calculate the inversion of the transformation matrix using simpler (and faster) transposition
$$T_{WS\to RS} = T_{RS\to WS}^{-1} = T_{RS\to WS}^{T}$$

Tangent-Bitangent-Normal

Side note: Texture coordinates are interpolated linearly (barycentric interpolation) across the triangle. Hence, the derivatives are all constant and we can calculate tangents/bitangents per triangle.

- $P_1 - P_0 = \boldsymbol{e}_1 = \Delta u_1 \boldsymbol{t} + \Delta v_1 \boldsymbol{b}$
- $P_2 - P_0 = \boldsymbol{e}_2 = \Delta u_2 \boldsymbol{t} + \Delta v_2 \boldsymbol{b}$
- $\Delta u_1 = P_1^u - P_0^u, \Delta v_1 = P_1^v - P_0^v$
- $\Delta u_2 = P_2^u - P_0^u, \Delta v_2 = P_2^v - P_0^v$

$\boldsymbol{e}_{1,2}$ and $\boldsymbol{t}$, $\boldsymbol{b}$ are 3D row vectors

$P_i^{\{u,v\}}$ are u, resp. v, texture coordinates of i-th vertex

… and we want to solve for $\boldsymbol{t}$ and $\boldsymbol{b}$…

- $\begin{bmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \end{bmatrix} = \begin{bmatrix} \Delta u_1 & \Delta v_1 \\ \Delta u_2 & \Delta v_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{t} \\ \boldsymbol{b} \end{bmatrix}$

Transformation matrix $TBN_{TS \to WS} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \hat{\boldsymbol{t}} & \hat{\boldsymbol{b}} & \hat{\boldsymbol{n}} \\ \vdots & \vdots & \vdots \end{pmatrix}$

- $\begin{bmatrix} \boldsymbol{t} \\ \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} \Delta u_1 & \Delta v_1 \\ \Delta u_2 & \Delta v_2 \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \end{bmatrix} = \frac{1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \begin{bmatrix} \Delta v_2 & -\Delta v_1 \\ -\Delta u_2 & \Delta u_1 \end{bmatrix} \begin{bmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \end{bmatrix}$

Tangent-Bitangent-Normal

- It is not necessarily true that the tangent vectors $\hat{\boldsymbol{t}}$ and $\hat{\boldsymbol{b}}$ are perpendicular to each other or to the normal vector $\hat{\boldsymbol{n}}$
- We may assume that these three vectors will be nearly orthogonal. Use Gram-Schmidt orthogonalization proces to fix that
- To find the tangent vectors for a single vertex, we average the tangents for all triangles sharing that vertex in a manner similar to the way in which vertex normals are commonly calculated. In the case that the neighboring triangles have discontinuous texture mapping, vertices along the border are generally already duplicated since they have different mapping coordinates anyway.

The Gram–Schmidt Process

- The Gram–Schmidt process works as follows

$$
\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \operatorname{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}
\end{aligned}
$$

where $\operatorname{proj}_{\widehat{\boldsymbol{u}}}(\boldsymbol{v}) = (\boldsymbol{v} \cdot \widehat{\boldsymbol{u}})\widehat{\boldsymbol{u}}$

Source: https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process

Tangent-Bitangent-Normal

- Using this process, orthogonal (but still unnormalized) tangent vectors $\boldsymbol{t}'$ and $\boldsymbol{b}'$ are given by

$$\boldsymbol{t}' = \boldsymbol{t} - (\boldsymbol{t} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$$

$$\boldsymbol{b}' = \boldsymbol{b} - (\boldsymbol{b} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - (\boldsymbol{b} \cdot \boldsymbol{t}')\boldsymbol{t}' / \cancel{\boldsymbol{t}'^2}$$

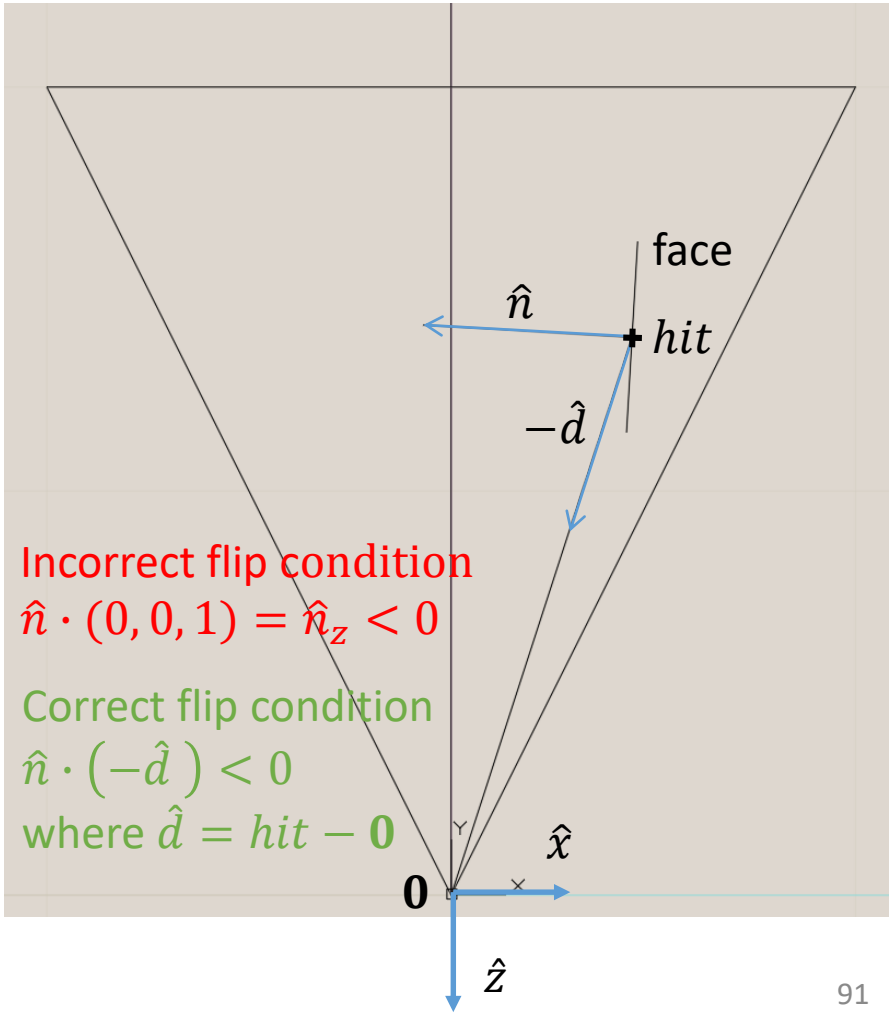
and the new TBN matrix takes the form

$$TBN_{TS \to WS} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \hat{\boldsymbol{t}}' & \hat{\boldsymbol{b}}' & \hat{\boldsymbol{n}} \\ \vdots & \vdots & \vdots \end{pmatrix}$$

Unified Normals

Vertex shader file

```
layout ( location = 0 ) in vec4 in_position_ms; // ( x, y, z, 1.0f )
layout ( location = 1 ) in vec3 in_normal_ms;
uniform mat4 mvn; // Model View
uniform mat4 mvn; // Model View Normal
out vec3 unified_normal_es;
…
void main( void )
{
  …
  unified_normal_es = normalize(( mvn * vec4( in_normal_ms.xyz, 0.0f ) ).xyz);
  vec4 hit_es = mv * in_position_ms; // mv * vec4( in_position_ms.xyz, 1.0f )
  vec3 omega_i_es = normalize( hit_es.xyz / hit_es.w );
  if ( dot( unified_normal_es, omega_i_es ) > 0.0f )
  {
    unified_normal_es *= -1.0f;
  }
  …
}
```



Vertex Buffer

```
glGenVertexArrays( 1, &vao_ );
glBindVertexArray( vao_ );
glGenBuffers( 1, &vbo_ ); // generate vertex buffer object (one of OpenGL objects) and get the unique ID corresponding to that buffer
glBindBuffer( GL_ARRAY_BUFFER, vbo_ ); // bind the newly created buffer to the GL_ARRAY_BUFFER target
glBufferData( GL_ARRAY_BUFFER, sizeof( Vertex )*no_vertices, vertices, GL_STATIC_DRAW ); // copies the previously defined vertex data into the buffer's memory

// vertex position
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, vertex_stride, ( void* )( offsetof( Vertex, position ) ) );
glEnableVertexAttribArray( 0 );
// vertex normal
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, vertex_stride, ( void* )( offsetof( Vertex, normal ) ) );
glEnableVertexAttribArray( 1 );
…
// material index
glVertexAttribIPointer( 5, 1, GL_INT, vertex_stride, ( void* )( offsetof( Vertex, material_index ) ) );
glEnableVertexAttribArray( 5 );
```

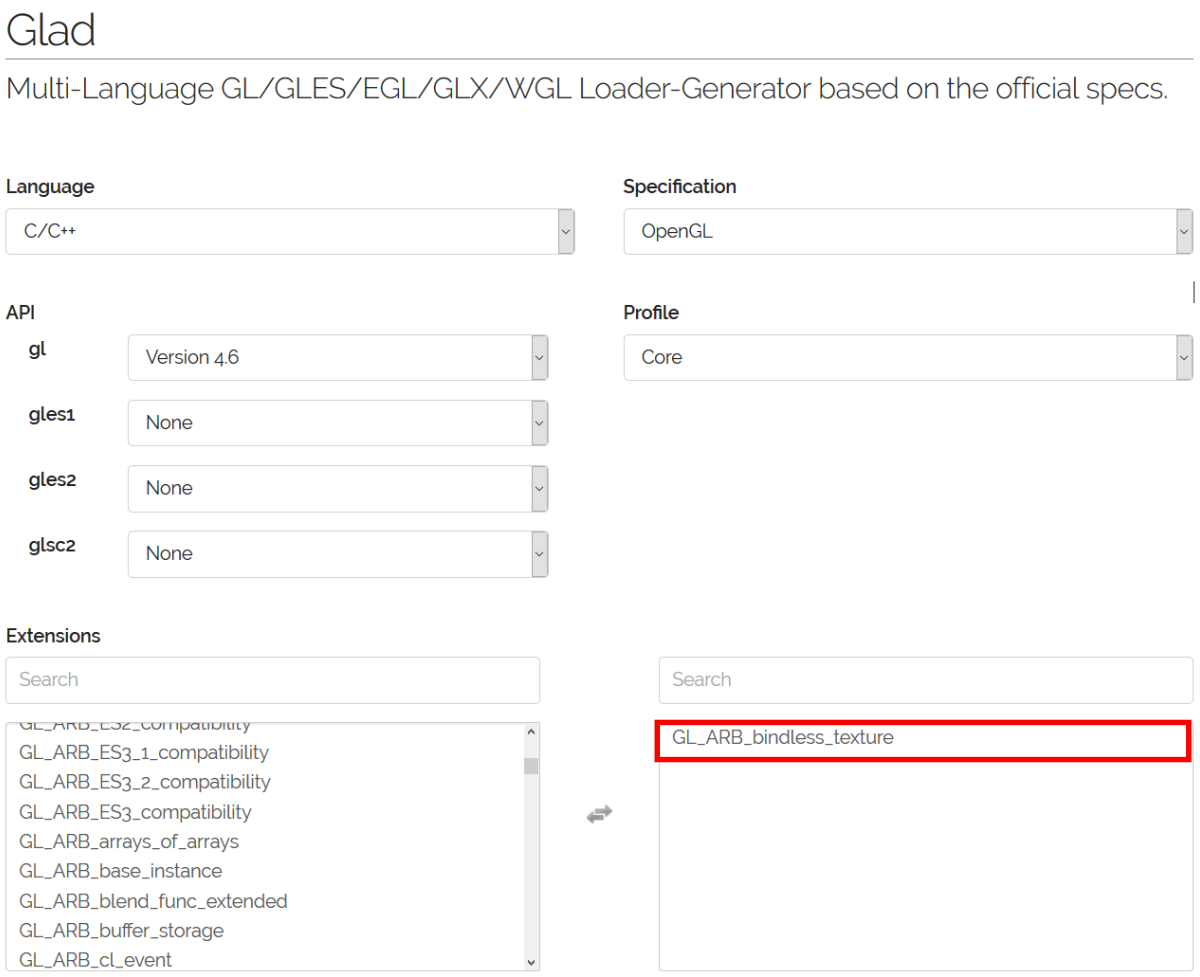
```
#pragma pack(push, 1)
struct Vertex
{
  Vector3 position;
  Vector3 normal;
  Vector3 color;
  Coord2f texture_coords;
  Vector3 tangent;
  int material_index{ 0 };
  char pad[4]; // fill up to 64 B
};
#pragma pack (pop)
```

Bindless Textures

- Classical approach: bound texture to a texture unit (represented as an uniform variable, e.g. sampler2D, in shaders)
  - The number of textures is limited to the number of texture units supported by the OpenGL driver (at least 16)
  - Spending time binding and unbinding textures between draw calls
- If OpenGL reports support for GL_ARB_bindless_texture, we can get around these problems (Intel HD 630 with driver 23.20.16.4944+)
- This ext. allows us to get a handle for a texture and use that handle directly in shaders to refer the underlying texture

Source: OpenGL SuperBible (7th edition)

Adding Extensions to OpenGL



1. Visit https://glad.dav1d.de and fill it accroding the left image
2. Download the generated glad.zip
3. Replace all files in libs/glad directory
4. Rename glad.c to glad.cpp in libs/glad/src
5. Replace all includes in glad.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <glad/glad.h>
```

with the following single line

```
#include "pch.h"
```

Bindless Textures

```
void CreateBindlessTexture( GLuint & texture, GLuint64 & handle, const int width, const int height, const GLvoid * data )
{
  glGenTextures( 1, &texture );
  glBindTexture( GL_TEXTURE_2D, texture ); // bind empty texture object to the target
  // set the texture wrapping/filtering options
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  // copy data from the host buffer
  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR, GL_UNSIGNED_BYTE, data );
  glGenerateMipmap( GL_TEXTURE_2D );
  glBindTexture( GL_TEXTURE_2D, 0 ); // unbind the newly created texture from the target
  handle = glGetTextureHandleARB( texture ); // produces a handle representing the texture in a shader function
  glMakeTextureHandleResidentARB( handle );
}
```

Details on https://www.khronos.org/registry/OpenGL/extensions/NV/NV_bindless_texture.txt

Materials as SSBO with Bindless Textures

MTL file:

```
newmtl white_plastic
 Pr 0.5
 Kd 1.0 1.0 1.0
 map_Kd scuffed-plastic6-alb.png
 map_RMA plastic_02_rma.png
 norm scuffed-plastic-normal.png
```

```
GLMaterial * gl_materials = new GLMaterial[materials_.size()];

int m = 0;
for ( const auto & material : materials_ )
{
  auto tex_diffuse = material.second->texture( Map::kDiffuse );
  if ( tex_diffuse )
  {
    GLuint id = 0;
    CreateBindlessTexture( id, gl_materials[m].tex_diffuse_handle, tex_diffuse->width(), tex_diffuse->height(), tex_diffuse->data() );
    gl_materials[m].diffuse = Color3f( { 1.0f, 1.0f, 1.0f } ); // white diffuse color
  }
  else
  {
    GLuint id = 0;
    GLubyte data[] = { 255, 255, 255, 255 }; // opaque white
    CreateBindlessTexture( id, gl_materials[m].tex_diffuse_handle, 1, 1, data );
    gl_materials[m].diffuse = material->value( Map::kDiffuse );
  }
  m++;
}

GLuint ssbo_materials = 0;
glGenBuffers( 1, &ssbo_materials );
glBindBuffer( GL_SHADER_STORAGE_BUFFER, ssbo_materials );
const GLsizeiptr gl_materials_size = sizeof( GLMaterial ) * materials_.size();
glBufferData( GL_SHADER_STORAGE_BUFFER, gl_materials_size, gl_materials, GL_STATIC_DRAW );
glBindBufferBase( GL_SHADER_STORAGE_BUFFER, 0, ssbo_materials );
glBindBuffer( GL_SHADER_STORAGE_BUFFER, 0 );
```

```
#pragma pack( push, 1 ) // 1 B alignment
struct GLMaterial
{
  Color3f diffuse; // 3 * 4B
  GLbyte pad0[4]; // + 4 B = 16 B
  GLuint64 tex_diffuse_handle{ 0 }; // 1 * 8 B
  GLbyte pad1[8]; // + 8 B = 16 B
};
#pragma pack( pop )
```

see http://www.catb.org/esr/structure-packing/

Materials as SSBO with Bindless Textures

Vertex Shader

```
#version 450 core
// vertex attributes
layout ( location = 0 ) in vec4 in_position_ms;
layout ( location = 1 ) in vec3 in_normal_ms;
layout ( location = 2 ) in vec3 in_color;
layout ( location = 3 ) in vec2 in_texcoord;
layout ( location = 4 ) in vec3 in_tangent;
layout ( location = 5 ) in int in_material_index;
// uniform variables
uniform mat4 mvp; // Model View Projection
uniform mat4 mvn; // Model View Normal (must be orthonormal)
// output variables
out vec3 unified_normal_es;
out vec2 texcoord;
flat out int material_index;
void main( void )
{
  // model-space -> clip-space
  gl_Position = mvp * in_position_ms;
  // normal vector transformations
  vec4 tmp = mvn * vec4( in_normal_ms.xyz, 1.0f );
  unified_normal_es = normalize( tmp.xyz / tmp.w );
  // 3ds max related fix of texture coordinates
  texcoord = vec2( in_texcoord.x, 1.0f - in_texcoord.y );
  material_index = in_material_index;
}
```

Fragment Shader

```
#version 460 core
#extension GL_ARB_bindless_texture : require
#extension GL_ARB_gpu_shader_int64 : require // uint64_t
// inputs from previous stage
in vec3 unified_normal_es;
in vec2 texcoord;
flat in int material_index;
struct Material
{
  vec3 diffuse;
  uint64_t tex_diffuse;
};
layout ( std430, binding = 0 ) readonly buffer Materials
{
  Material materials[]; // only the last member can be unsized
array
};
// outputs
out vec4 FragColor;
void main( void )
{
  FragColor = vec4( materials[material_index].diffuse.rgb *
    texture( sampler2D( materials[material_index].tex_diffuse ),
texcoord ).rgb, 1.0f );
}
```

From the previous slide

```
…
glBindBufferBase( GL_SHADER_STORAGE_BUFFER,
0, ssbo_materials );
…
```

PBR Materials as SSBO with Bindless Textures

Rasterizer::InitMaterials

```
#pragma pack( push, 1 ) // 1 B alignment
struct GLMaterial
{
    Color3f diffuse; // 3 * 4 B
    GLbyte pad0[4]; // + 4 B = 16 B
    GLuint64 tex_diffuse_handle{ 0 }; // 1 * 8 B
    GLbyte pad1[8]; // + 8 B = 16 B

    Color3f rma; // 3 * 4 B
    GLbyte pad2[4]; // + 4 B = 16 B
    GLuint64 tex_rma_handle{ 0 }; // 1 * 8 B
    GLbyte pad3[8]; // + 8 B = 16 B

    Color3f normal; // 3 * 4 B
    GLbyte pad4[4]; // + 4 B = 16 B
    GLuint64 tex_normal_handle{ 0 }; // 1 * 8 B
    GLbyte pad5[8]; // + 8 B = 16 B
};

#pragma pack( pop )
```

Structure packing really matters here. More datails on the **std430** layout rules can be found in OpenGL specification.

Fragment Shader

```
struct Material
{
    vec3 diffuse; // (1,1,1) or albedo
    uint64_t tex_diffuse; // albedo texture

    vec3 rma; // (1,1,1) or (roughness, metalness, 1)
    uint64_t tex_rma; // rma texture

    vec3 normal; // (1,1,1) or (0,0,1)
    uint64_t tex_normal; // bump texture
};

layout ( std430, binding = 0 ) readonly buffer Materials
{
    Material materials[];
};
```

Note that the second option is chosen when the corresponding texture is not available

Bindless Textures on Intel IGPs

- According the GLSL spec, opaque types like sampler2D cannot be used in structures (although some drivers allow that – e.g. NVidia)
- The GL_ARB_gpu_shader_int64 extension is not available on Intel IGPs like HD 630 or Iris 645 thus we cannot simply replace sampler2D with uint64_t in Material structure
- As a consequence, we have to use different 64-bit data type for our bindless texture handles
- Fortunately, we can use **uvec2** data type instead

1. remove GL_ARB_gpu_shader_int64 extension
2. replace uint64_t with uvec2
3. cast uvec2 texture handle to sampler2D in texture function calls
4. C++ part of our code remains the same

Fragment Shader

```
#version 460 core
#extension GL_ARB_bindless_texture : require

…

struct Material
{
  vec3 diffuse;
  //sampler2D tex_diffuse; // not allowed by GLSL spec in structs
  //uint64_t tex_diffuse;    // not available on Intel IGPs
  uvec2 tex_diffuse;
  …
};

…

vec3 diffuse = texture( sampler2D(
materials[material_index].tex_diffuse ), texcoord ).rgb;

…
```

Provoking Vertex

- In case of flat shaded interpolants (e.g. material index), we have to specify from which vertex of a single primitive will be taken
- Call glProvokingVertex to set the desired mode which vertex is to be used as the provoking vertex
  - GL_FIRST_VERTEX_CONVENTION
  - GL_LAST_VERTEX_CONVENTION (default)

Primitive Type of Polygon *i*	First Vertex Convention	Last Vertex Convention
point	*i*	*i*
independent line	2*i* - 1	2*i*
line loop	*i*	*i* + 1, if *i* < *n* 1, if *i* = *n*
line strip	*i*	*i* + 1
independent triangle	3*i* - 2	3*i*
triangle strip	*i*	*i* + 2
triangle fan	*i* + 1	*i* + 2
line adjacency	4*i* - 2	4*i* - 1
line strip adjacency	*i* + 1	*i* + 2
triangle adjacency	6*i* - 5	6*i* - 1
triangle strip adjacency	2*i* - 1	2*i* + 3

Entity Component System (ECS) –

https://github.com/skypjack/entt

- **EnTT uses sparse-set-based component storage, which means components of the same type are stored contiguously in memory.** This makes iteration over entities with certain components extremely fast — critical for game loops and real-time systems like physics, rendering, and AI
- **It also allows operations without the overhead of virtual function calls or inheritance hierarchies**
- **Entities are just IDs, decoupled from data**
- **Components are plain structs or classes**
- **Systems operate on views of entities that have specific components — no rigid inheritance tree needed**
- **Uses C++11/14/17 features** like variadic templates, constexpr, and type-safe identifiers
- No need for macros or code generation — everything is compile-time type-checked
- **Supports move semantics, so components can be moved efficiently without unnecessary copies**

Entity Component System (ECS) – EnTT

https://github.com/skypjack/entt

- EnTT provides on_construct, on_destroy, and on_update signals for components. This allows easy **event-driven programming**:
  - Trigger initialization when a component is added
  - Clean up when an entity is destroyed
  - React to component changes efficiently
- While **ECS is naturally flat, EnTT supports parent-child relationships**, custom tags, and metadata
- You **can implement scene graphs, transform hierarchies, or grouped entities without breaking the ECS paradigm**
- Coupled with views, this makes it easy to traverse complex entity relationships efficiently
- Lightweight and header-only — easy to integrate into any engine
- Well-maintained and widely used, with a strong community

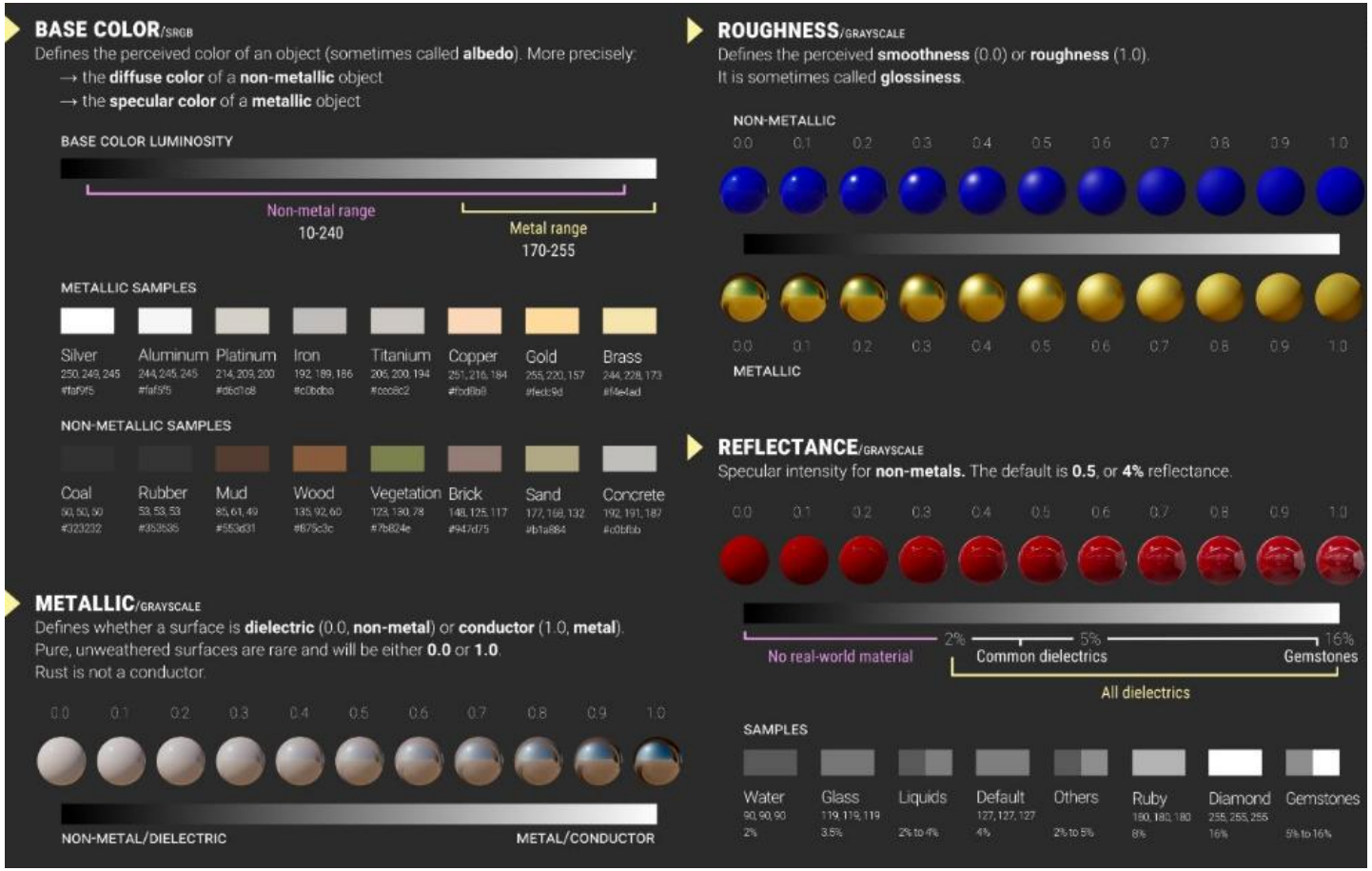
Constructors

```
struct Mesh {
  Mesh() // explicit constructor with no parameters, e.g. Mesh a; calls this constructor
  Mesh( const Mesh & mesh ) // explicit copy constructor, e.g. Mesh b = a; calls this constructor
  Mesh( Mesh && ) noexcept = default; // forces implicit move constructor (because a copy constructor, a copy assignment operator, a destructor (even if default but user - provided), or a move constructor / assignment operator prevent the implicit move constructor),
e.g. Mesh a = make_mesh(); or Mesh b = std::move( a ); calls this constructor

  Mesh & operator=( const Mesh & ) = delete; // removes implicit copy assignment operator,
e.g. b = a; calls this operator
  Mesh & operator=( const Mesh & a ) // re-enable implicit copy assignment operator

  Mesh & operator=( Mesh && ) = delete; // delete copy assignment operator
  Mesh & operator=( Mesh && a ) // re-enable implicit move assignment operator,
e.g. b = std::move( a ) calls this operator

  ~Mesh() // explicit destructor
};
```

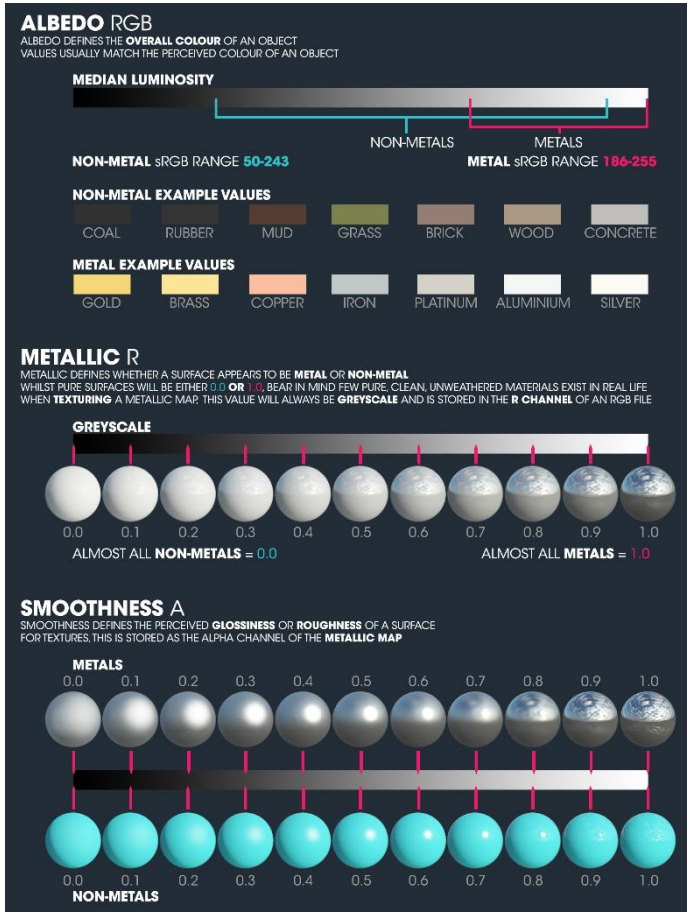
PBR Workflow

- Physically-based material workflows:
  - Metallic-Roughness workflow
    - **Base color (albedo) ≠ diffuse** is represented as a color map without any lighting in the range 30-240 sRGB (for dielectrics) or pure black color (for conductors)
    - **Metalicity** is typically a binary (or linearly interpolated grayscale) texture containing 0‘s (dielectrics) and 1‘s (metals)
    - **Roughness** – a grayscale linear texture in the range 0 (smooth) and 1 (rough)
  - Specular-Glossines workflow
    - **Diffuse** (Albedo) – RGB map
    - **Specular** – RGB map
    - **Glossines** – a grayscale linear texture that describes the surface irregularities that cause light diffusion. It is the inverse of the roughness map

Filament PBR Materials



Unity PBR Materials



PBR Textures and Materials

- On-line sources of free seamless PBR textures with Diffuse, Normal, Displacement, Occlusion, Specularity and Roughness maps:
  - https://cc0textures.com
  - https://texturehaven.com
  - https://www.poliigon.com
  - https://freepbr.com
  - https://3dtextures.me