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

Computer Graphics II

460-4033

Spring 2025

Last update 19. 4. 2025

Computer Graphics II

- Lecturer
  - Tomas Fabian
  - Office

    room EA408, building of FEECS
  - Office hours

    Tuesday 13:30 – 15:30 (all other office hours are by appointment)
  - Email

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

    http://mrl.cs.vsb.cz/people/fabian/pg2_course.html

Course Targets and Goals

- Further extend the techniques of photorealistic image synthesis introduced in the previous course.
- You will have hands-on experience with implementation of the here described methods and algorithms for creating synthetic images in real-time.
- Mastering selected libraries for real-time image synthesis.

Course Prerequisites

- Basics of programming (C++)

- Previous courses:
  - Fundamentals of Computer Graphics (ZPG), Computer Graphics I (PG I)

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

Main Topics

- OpenGL pipeline (core-profile, GLSL)
- Normal mapping
- Shadows
- Deferred rendering
- PBR and global illumination
- Ray tracing at interactive frame rates on GPUs

- Signed distance field rendering
- 3D fractals rendering

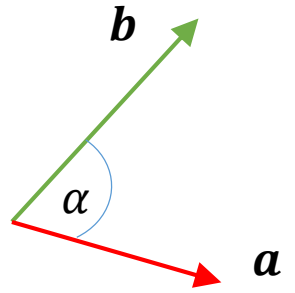
Organization of Semester and Grading

- Each lecture will discuss one main topic
- Given topic will be practically realized during the following exercise
- The individual tasks from the exercise will be scored (during the last week of the semester)
- You can earn up to 100 points in total from all exercises
- The course is not completed by an exam, only by credit, so your activity during the exercises is really important

Study Materials

- Sojka, E.: *Počítačová grafika II: metody a nástroje pro zobrazování 3D scén*, VŠB-TU Ostrava, 2003, ISBN 80-248-0293-7. (online)
- Sojka, E., Němec, M., Fabián, T.: *Matematické základy počítačové grafiky*, VŠB-TU Ostrava, 2011. (online)
- Pharr, M., Jakob, W., Humphreys, G.: *Physically Based Rendering*, Third Edition: From Theory to Implementation, Morgan Kaufmann, 2016, 1266 pages, ISBN 978-0128006450. (online)
- Haines, E., Akenine-Möller, T. (ed.): *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019, 607 pages, ISBN 978-1484244265. (online)
- Marrs, A., Shirley, P., Wald, I (ed.). *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Springer Nature, 2021, 858 pages, ISBN 978-1484271841. (online)
- Shirley, P., Morley, R. K.: *Realistic Ray Tracing*, Second Edition, AK Peters, 2003, 235 pages, ISBN 978-1568814612.
- Akenine-Möller, T., Haines, E., Hoffman, N.: *Real-Time Rendering*, Fourth Edition, AK Peters, 2018, 1198 pages, ISBN 978-1351816151.
- Dutré, P.: *Global Illumination Compendium*, 2003, 68 pages. (online)
- Ryer, A. D.: *The Light Measurement Handbook*, 1997, 64 pages. (online)
- Segal, M., Akeley, K.: *The OpenGL Graphics System*, 2019, 850 pages. (online)
- Michael, A.: Graphics Programming Black Book. *Coriolis Group, Ames Iowa*, 2001. (online)

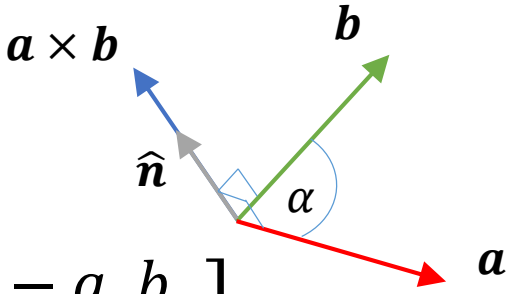
Basic Operators



- 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

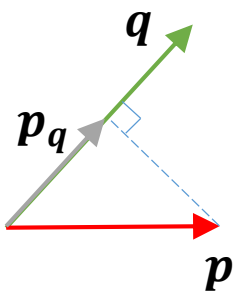
- (Vector) projection of $\boldsymbol{p}$ on $\boldsymbol{q}$

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

Matrix notation may be useful for repeated projections

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

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



Rotations

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

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

$$\alpha := 45\ \textit{deg} \qquad R := \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \quad a := \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad R \cdot a = \begin{bmatrix} 0.707 \\ 0.707 \end{bmatrix}$$

- 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)

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

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

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

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

Quaternions

- Generalization of complex numbers in 4D space

$$\boldsymbol{q} = \underbrace{(\boldsymbol{v}, w)}_{\text{ordered pair}} = \underbrace{\boldsymbol{i}x + \boldsymbol{j}y + \boldsymbol{k}z}_{\text{imaginary part}} + \underbrace{w}_{\text{real part}} = \boldsymbol{v} + w$$

- $i^2 = j^2 = k^2 = -1, jk = -kj = i\ , ki = -ik = j, ij = -ji = k$

- Invented by Sir Hamilton in 1843, used in graphics since 1985
- Quaternion is a geometrical operator to represent the relationship (relative length and relative orientation) between two vectors in 3D space

Quaternions

- Addition

$$(\boldsymbol{v}_1, w_1) + (\boldsymbol{v}_2, w_2) = (\boldsymbol{v}_1 + \boldsymbol{v}_2, w_1 + w_2)$$

- Multiplication (associative but not commutative)

$$(\boldsymbol{v}_1, w_1)(\boldsymbol{v}_2, w_2) = (\boldsymbol{v}_1 \times \boldsymbol{v}_2 + w_2\boldsymbol{v}_1 + w_1\boldsymbol{v}_2, w_1w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2)$$

- Conjugation

$$\boldsymbol{q}^* = (\boldsymbol{v}, w)^* = (-\boldsymbol{v}, w)$$

- Norm

$$\|\boldsymbol{q}\| = \|\boldsymbol{q}^*\| = \sqrt{\boldsymbol{q}\boldsymbol{q}^*} = \sqrt{x^2 + y^2 + z^2 + w^2}$$

Quaternions

- Unit quaternion (quaternion of norm one, versor, represents a specific rotation in 3D space – see next slide)

$$\|\boldsymbol{q}\| = 1$$

- Real quaternion

$$(\mathbf{0}, w) = w$$

- Reciprocal (inverse)

$$\boldsymbol{q}^{-1} = \frac{\boldsymbol{q}^*}{\|\boldsymbol{q}\|^2}$$

- Multiplication by a scalar s

$$s\boldsymbol{q} = (\mathbf{0}, s)(\boldsymbol{v}, w) = (s\boldsymbol{v}, sw)$$

Quaternions

- Ambiguity

$\boldsymbol{q}$ and $-\boldsymbol{q}$ represents the same rotation

- Unit quaternion expressed by goniometry

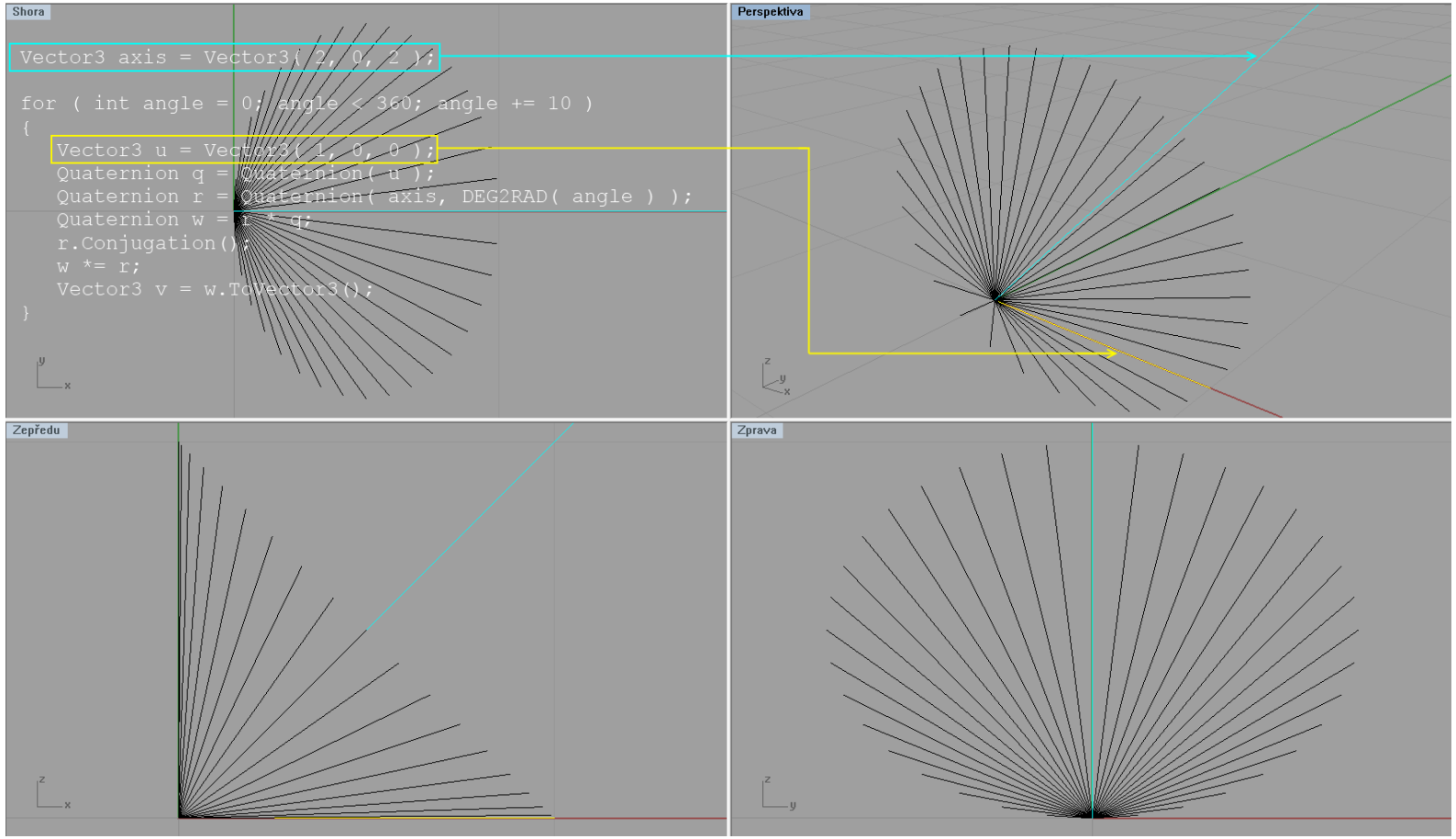
$$\boldsymbol{q} = (\hat{\boldsymbol{v}} \sin\alpha\,, \cos\alpha)$$

where $\hat{\boldsymbol{v}}$ is unit axis of rotation and α is rotation angle

- Counterclockwise rotation of point $\boldsymbol{a} = (x, y, z, 0)$ around $\hat{\boldsymbol{v}}$ by angle 2α

$$\boldsymbol{a}' = \boldsymbol{q}\boldsymbol{a}\boldsymbol{q}^{-1} = \boldsymbol{q}\boldsymbol{a}\boldsymbol{q}^{*}$$

Quaternions



Spherical Linear Interpolation (SLERP)

- Two quaternions $\boldsymbol{q}$ and $\boldsymbol{r}$ and real parameter $0 \leq t \leq 1$
- Interpolated quaternion

$$\text{slerp}(\boldsymbol{q}, \boldsymbol{r}, t) = \boldsymbol{q}(\boldsymbol{q}^* \boldsymbol{r})^t = \frac{\sin\big(\alpha(1-t)\big)}{\sin\alpha}\boldsymbol{q} + \frac{\sin(\alpha t)}{\sin\alpha}\boldsymbol{r}$$

This works for quaternions

This works for unit-length elements of any arbitrary-dimensional inner product space provided that $\alpha = \cos^{-1}(\boldsymbol{q} \cdot \boldsymbol{r})$

represents the shortest spherical arc between $\boldsymbol{q}$ and $\boldsymbol{r}$

- In other words, it describes interpolation between $\boldsymbol{q}$ and $\boldsymbol{r}$ with constant angular velocity
- Power of a unit quaternion

$$\boldsymbol{q}^t = (\hat{\boldsymbol{v}}\sin\alpha + \cos\alpha)^t = e^{t\alpha\hat{\boldsymbol{v}}} = \hat{\boldsymbol{v}}\sin(t\alpha) + \cos(t\alpha)$$

Comparing Rotations

- http://www.boris-belousov.net/2016/12/01/quat-dist/
- https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation

Normalized Linear Interpolation (NLERP)

- While SLERP interpolates along a great arc between two quaternions, it is also possible to interpolate along a straight line (in four-dimensional quaternion space) between those two quaternions. The resulting interpolant is *not* part of the unit hypersphere, i.e. the interpolated values are *not* unit quaternions. However, they can be normalized to become unit quaternions. This is called "**n**ormalized **l**inear int**erp**olation", in short NLERP. The resulting interpolant travels through the same quaternions as SLERP does, but it doesn't do it with constant angular velocity.

Source: https://splines.readthedocs.io/en/latest/rotation/slerp.html

Summary of rotations

- Matrices
  - + HW support, efficient transformations
  - - memory requirements
- Rotation axis and angle (Rodrigues' rotation formula)
  - + memory efficient, similar to quaternion
  - - inefficient composition and interpolation
- Quaternion
  - + memory efficient, composition, interpolation
  - - inefficient transformation

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: $(x, y, z, w \neq 0) \rightarrow (x/w, y/w, z/w)$

(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

$$\hat{\boldsymbol{z}}_e = \frac{\boldsymbol{e}-\boldsymbol{t}}{\|\boldsymbol{e}-\boldsymbol{t}\|}, \hat{\boldsymbol{x}}_e = \frac{up \times \hat{\boldsymbol{z}}_e}{\|up \times \hat{\boldsymbol{z}}_e\|}, \text{ and } \hat{\boldsymbol{y}}_e = \hat{\boldsymbol{z}}_e \times \hat{\boldsymbol{x}}_e, \qquad up = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

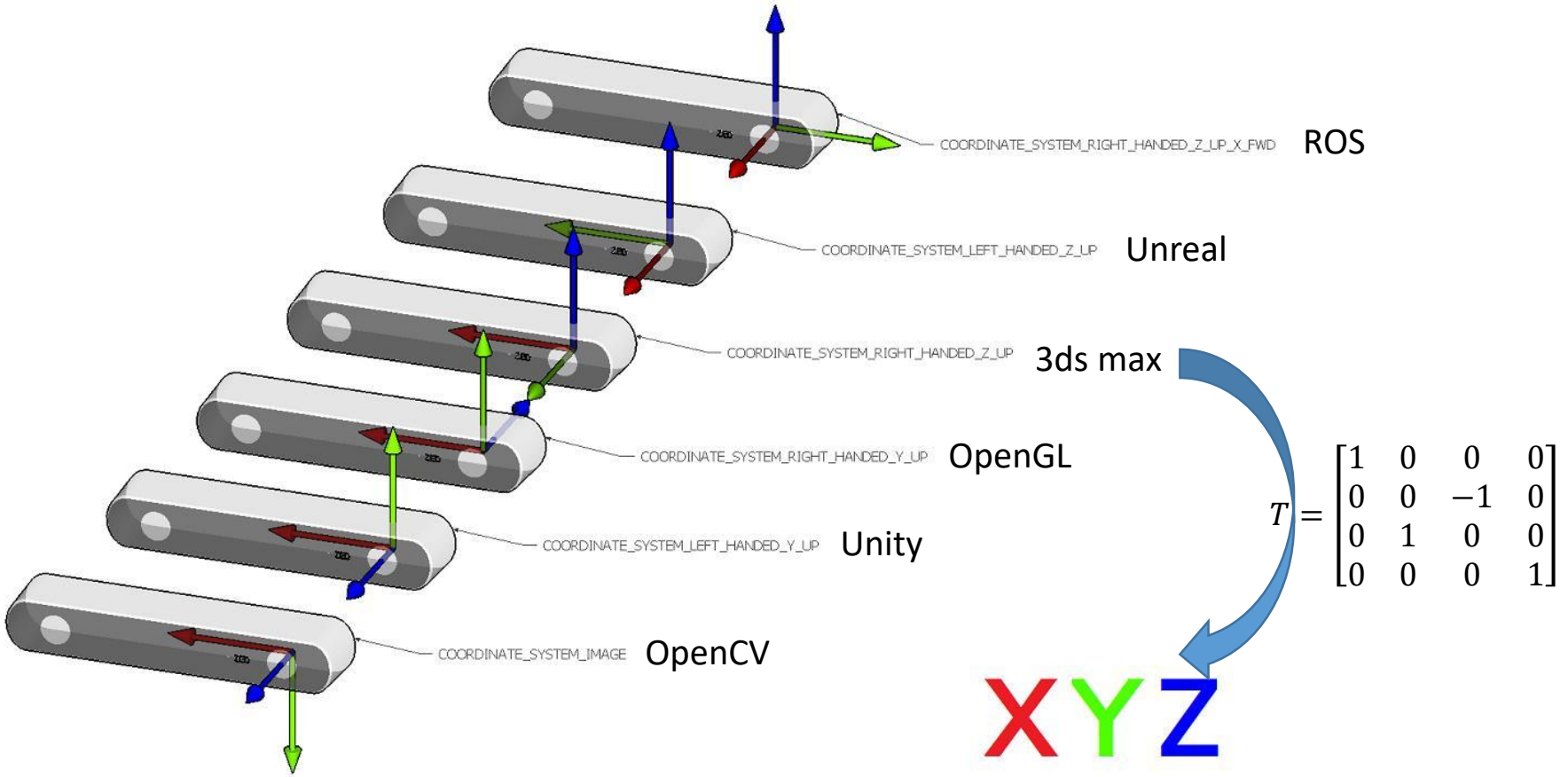
where the vector $\boldsymbol{e}$ represents position of the camera (eye), $\boldsymbol{t}$ is the target position and up is an auxiliary vector marking „up“ direction (a unit vector not parallel to the optical axis)

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

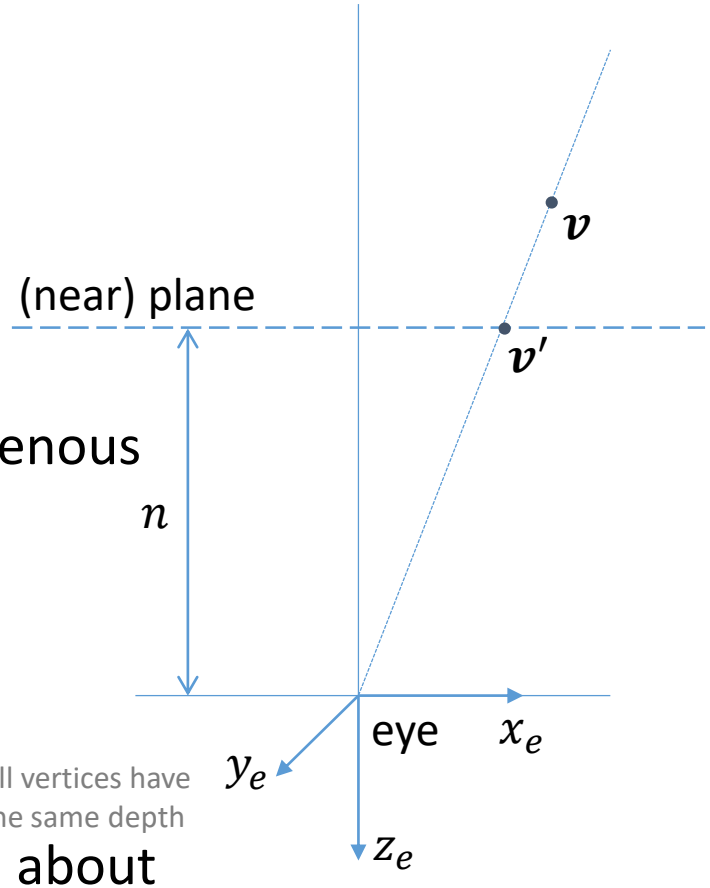
This transformation can be performed by the following homogenous matrix

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

After perspective division we get

$$\Rightarrow \quad \boldsymbol{v}' = \begin{bmatrix} n\frac{v_x}{-v_z} \\ n\frac{v_y}{-v_z} \\ -n \end{bmatrix}$$

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

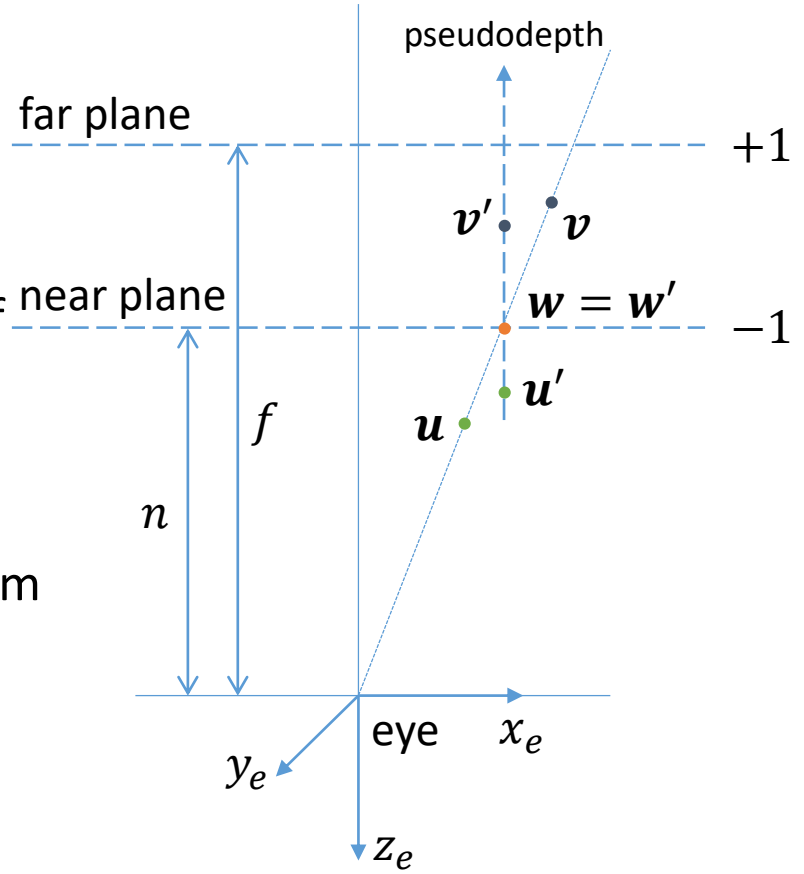
- To fix the depth loss we may incorporate pseudodepth such that points on the near plane will be given a depth of -1 and points on the far plane will be at $+1$
- We just need to solve $v_z' = \frac{av_z+b}{-v_z}$ for a and b what gives us a set of two equations as follows

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

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)

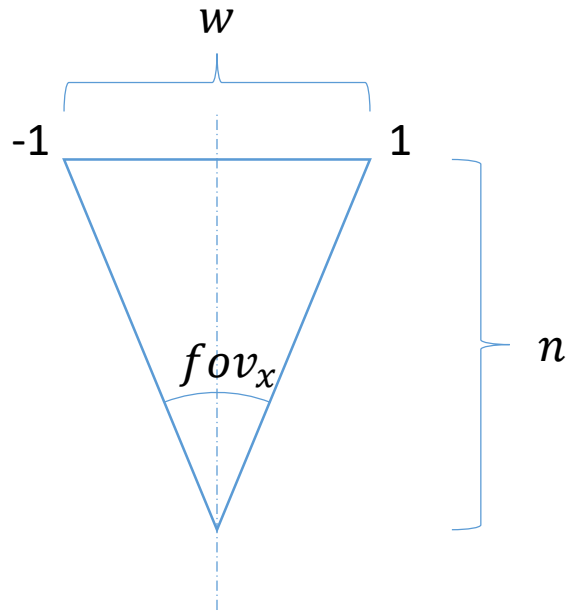
Projection Matrix 3/4

- Normalization transformation to NDC

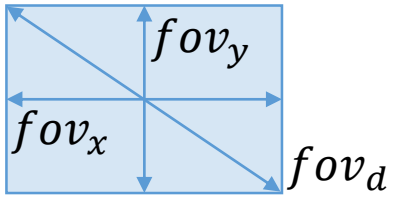
Note that the pseudo depth is already normalized

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

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

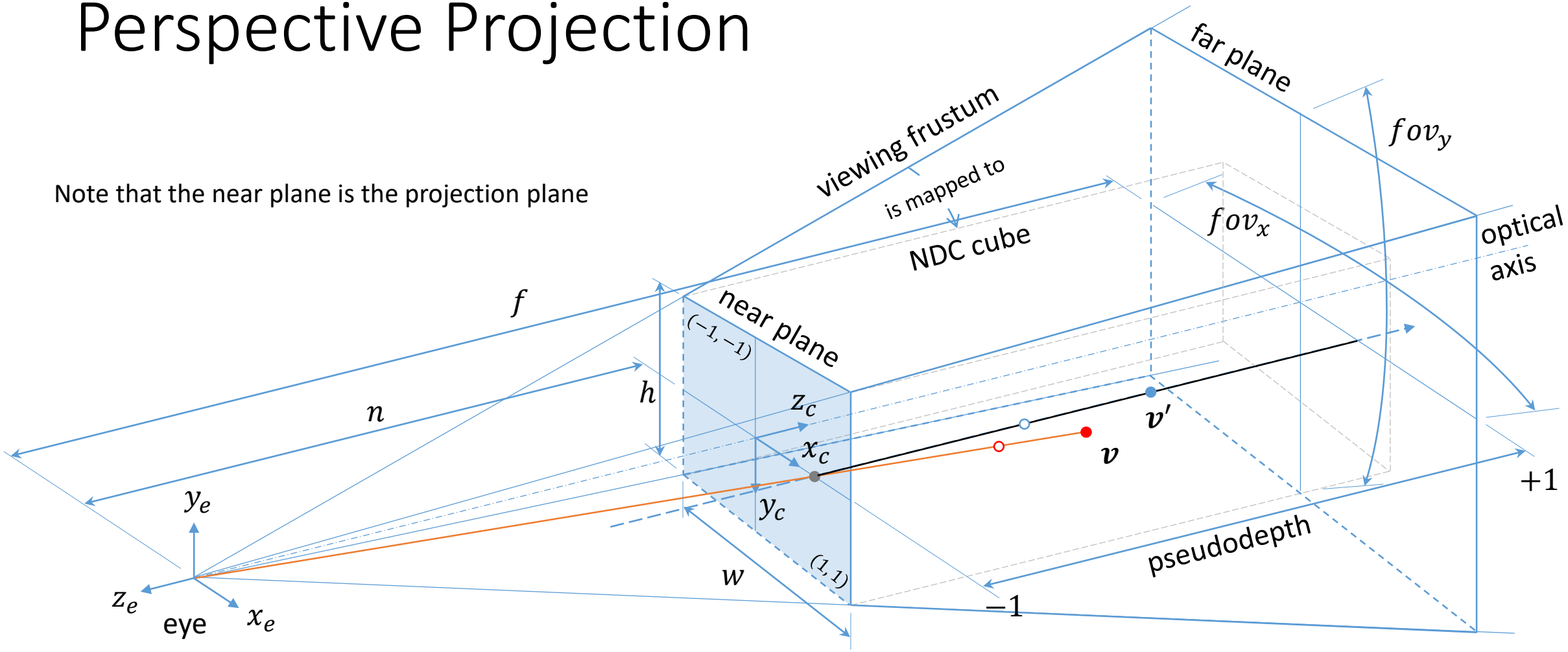
Use + when the clip control origin is lower left and – when upper left

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

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 $\mathrm{MV} = View\ Model$

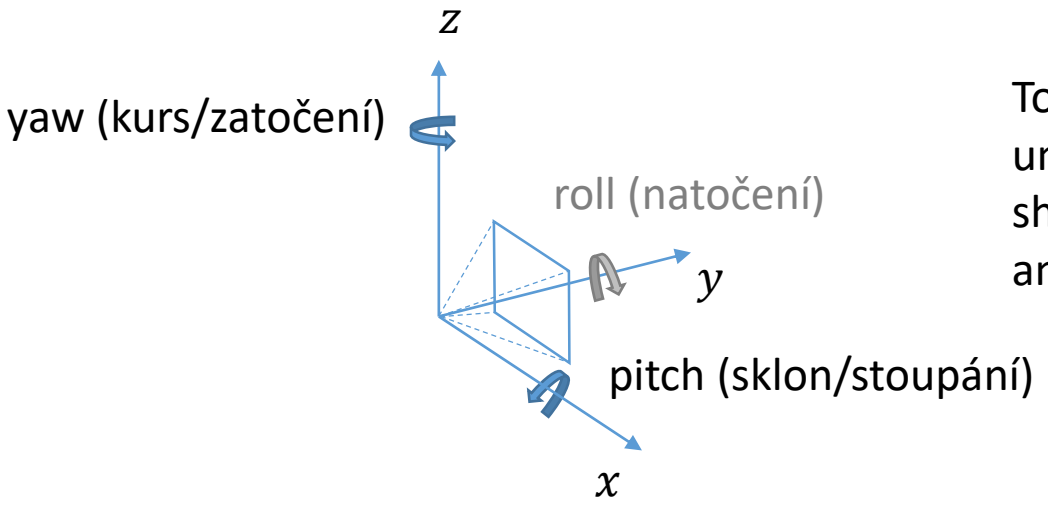
e.g. $\boldsymbol{v}_{es} = \mathrm{MV}\ \boldsymbol{v}_{ms}$

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

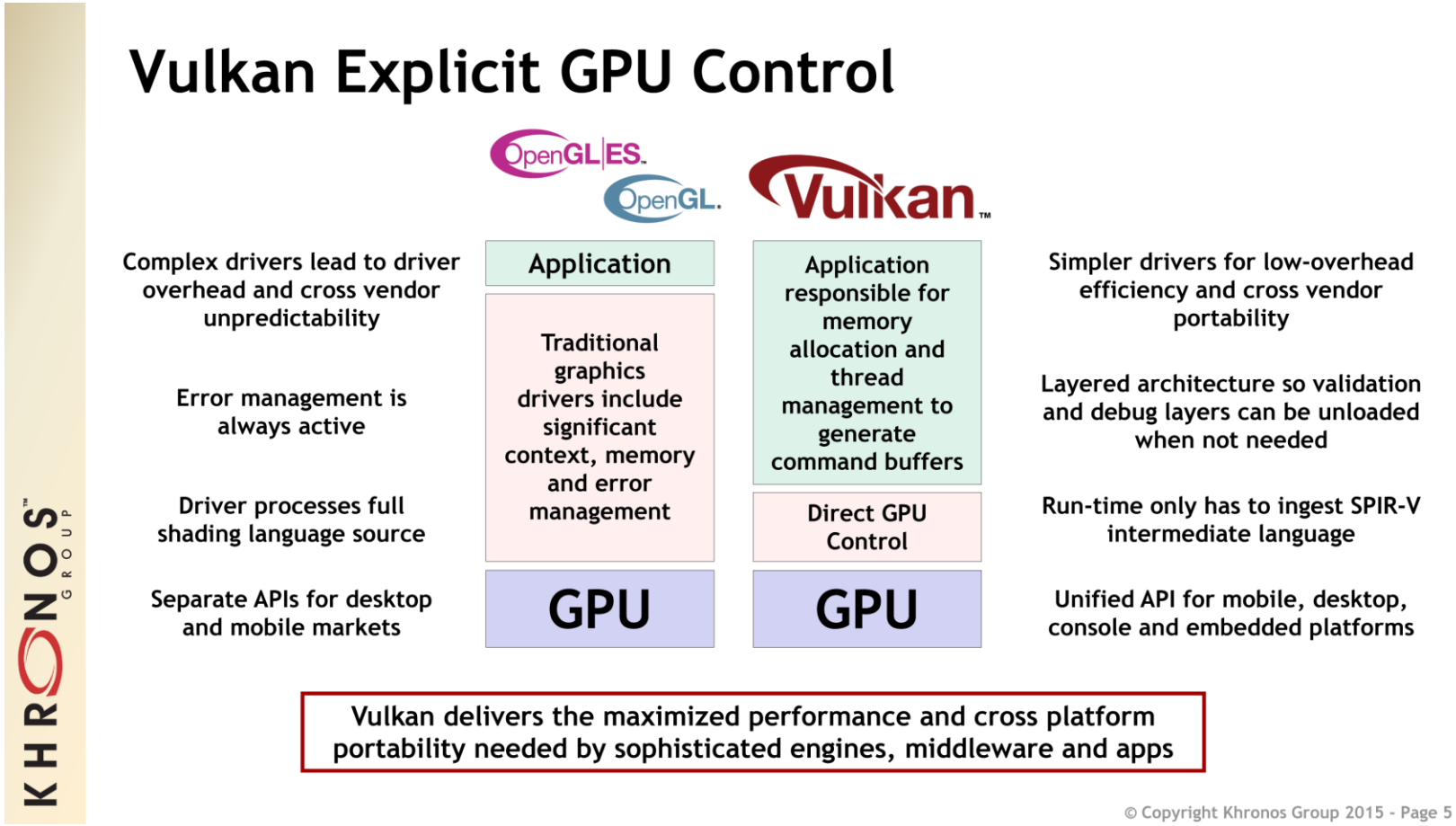
- 1.0 (1992) – first release
- 1.1 (1997) – texture objects
- 1.2 (1998) – 3D textures, BGRA
- 1.2.1 (1998) – ARB extension concept
- 1.3 (2001) – multitexturing
- 1.4 (2002) – depth textures
- 1.5 (2003) – vertex buffer objects
- 2.0 (2004) – shader objects (GLSL)
- 2.1 (2006) – pixel buffer objects, 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

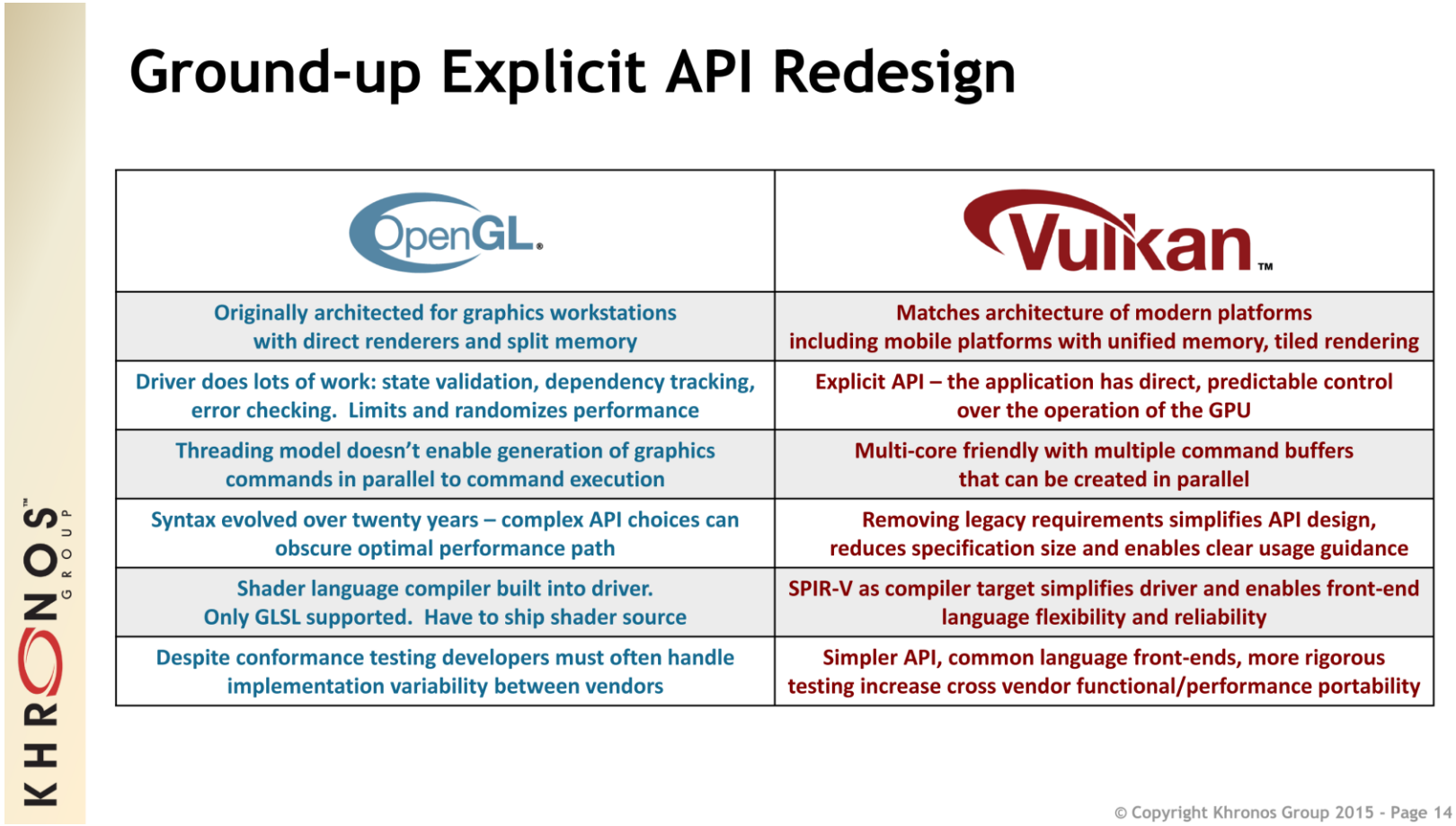
Other Graphics APIs

- 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



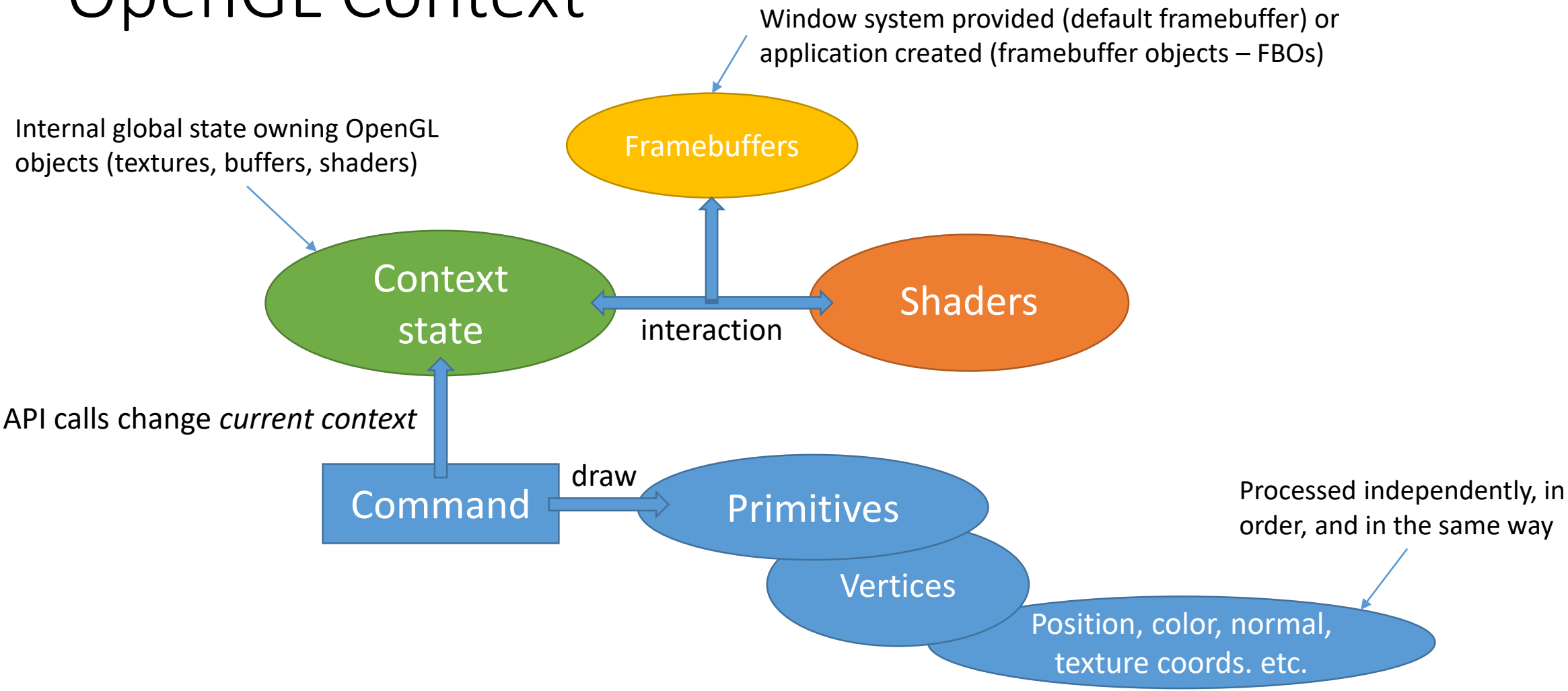
OpenGL

- 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

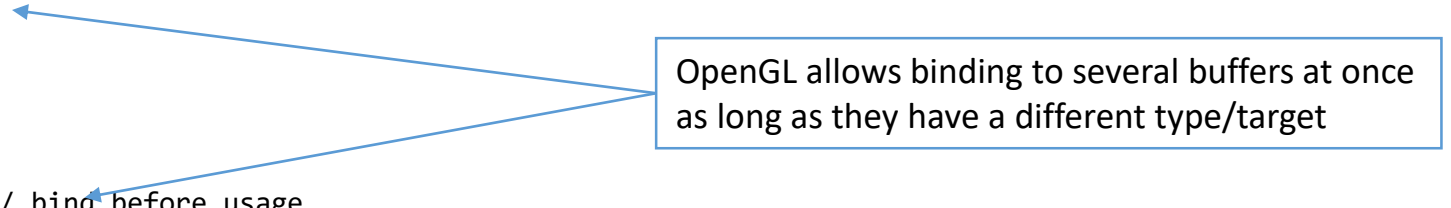
OpenGL Context



OpenGL Context

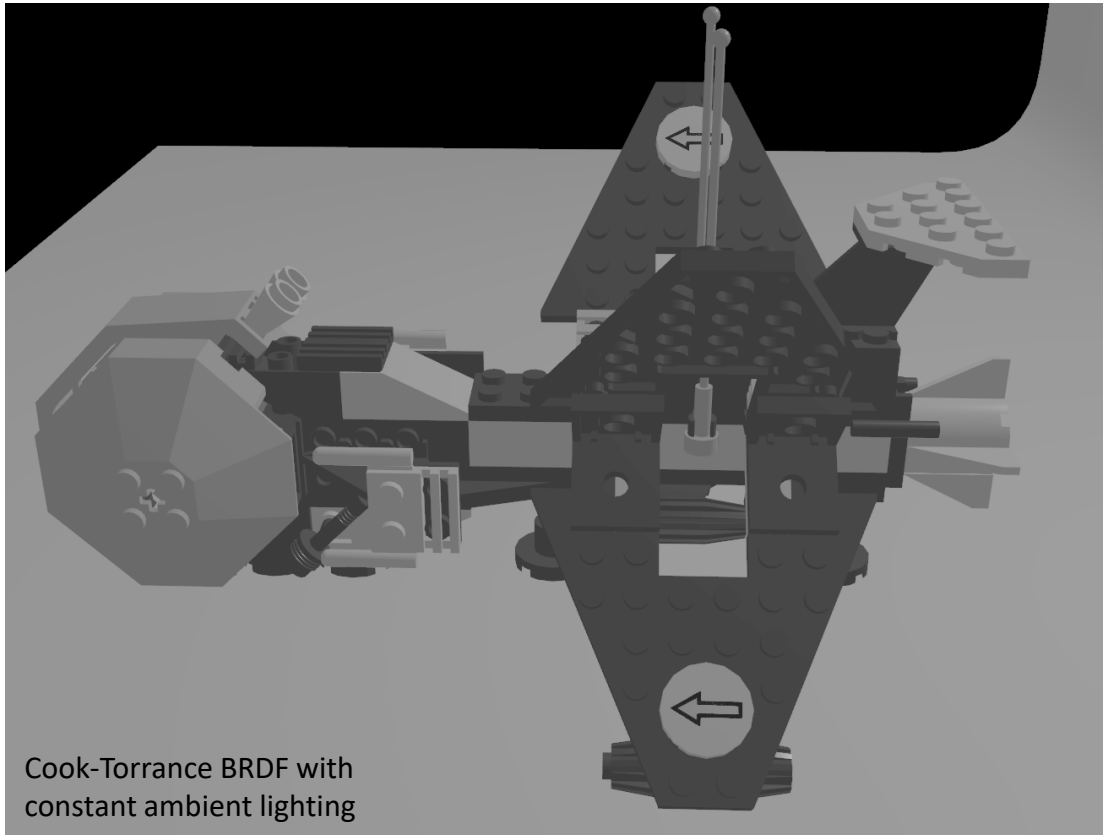
- OpenGL is a state machine (collection of variables)
- It‘s current state is referred to as the OpenGL context
- C-library API consist of state-changing functions

```
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 );
```

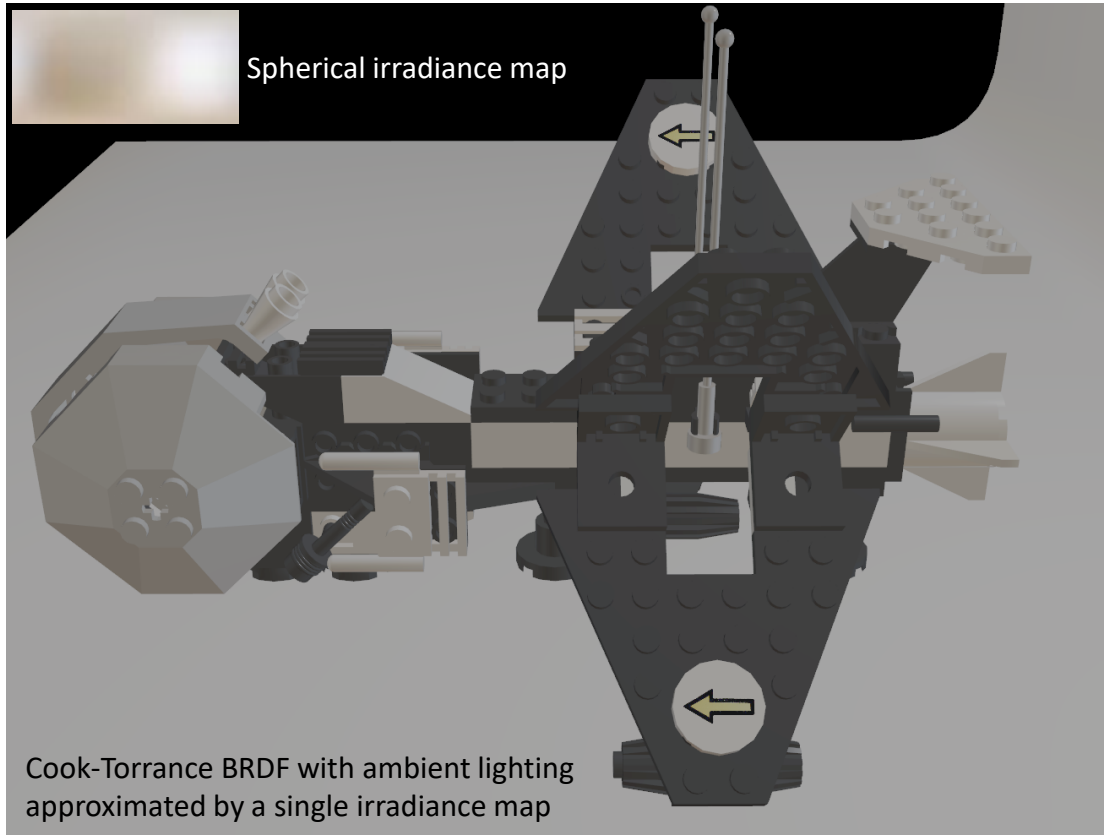


(Global) Illumination in Rasterization

- We will start with the diffuse part of the RE

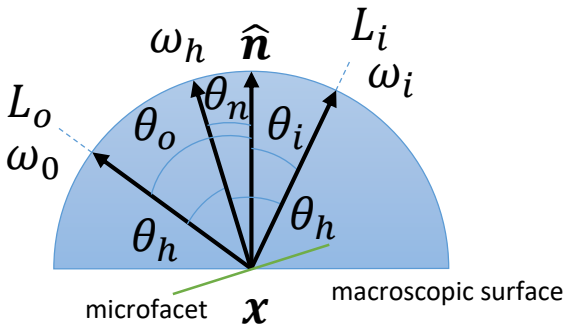


Cook-Torrance BRDF with
constant ambient lighting



Cook-Torrance BRDF with ambient lighting
approximated by a single irradiance map

Cook Torrance BRDF Revisited



- Outgoing radiance

$$L_o(\boldsymbol{x}, \omega_0) = L_r^D(\boldsymbol{x}, \omega_0) + L_r^S(\boldsymbol{x}, \omega_0)$$

- Diffuse part

$$L_r^D(\boldsymbol{x}, \omega_0) \approx \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i) \frac{albedo}{\pi} \frac{\cos\theta_i}{pdf(\omega_i)}$$

Lambert BRDF

Directions ω_i (resp. θ_i) are obtained via cosine weighted hemisphere sampling

- Specular part

$$L_r^S(\boldsymbol{x}, \omega_0) \approx \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i) \frac{D(\alpha, \theta_m)F(\theta_h)G(\theta_o, \theta_i)}{4\cos\theta_0 \cos\theta_i} \frac{\cos\theta_i}{pdf(\omega_i)}$$

Cook Torrance BRDF

Directions ω_i (resp. θ_i) are obtained via TR (GGX) sampling

Microfacet Normal Distribution Function GGX

- Describes the distribution (concentration) of microfacets with normals equal to ω_h

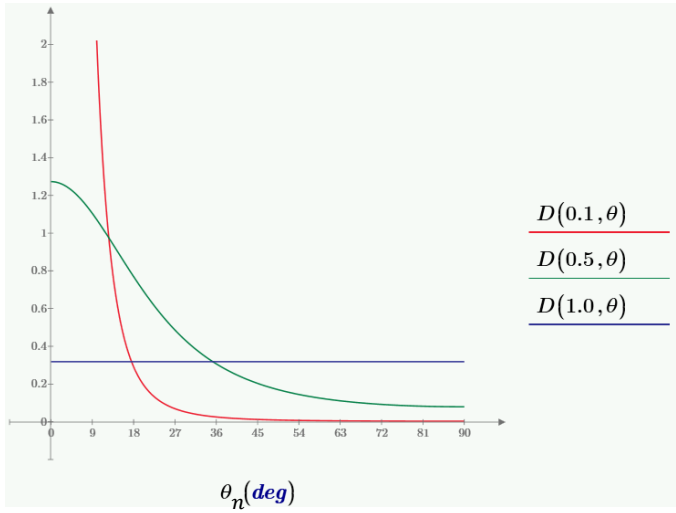
Disney's choice of GGX/Trowbridge-Reitz

$$D(\alpha, \theta_m) = \frac{\alpha^2}{\pi\left((\cos\theta_m)^2(\alpha^2 - 1) + 1\right)^2}$$

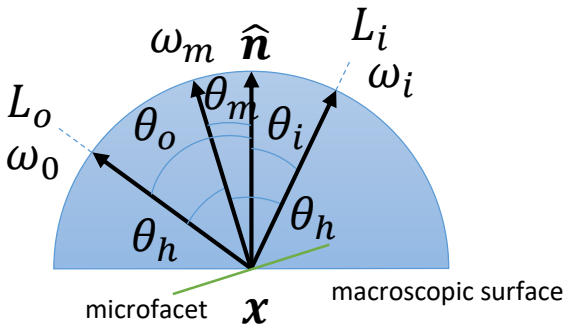
where $\alpha \in \langle 0.001, 1 \rangle$ is related to the roughness

and $\cos\theta_m = \hat{\boldsymbol{n}} \cdot \omega_m$

$$\alpha = roughness^2$$



TR (GGX) Sampling



- Sample microfacet normal ω_h with GGX distribution

$$\varphi_m = 2\pi\xi_1 \text{ and } \theta_m = \tan^{-1}\left(\alpha\sqrt{\frac{\xi_2}{1-\xi_2}}\right) = \cos^{-1}\left(\sqrt{\frac{1-\xi_2}{\xi_2(\alpha^2-1)+1}}\right)$$

where $\xi_{1,2} \in \langle 0,1)$ are random numbers with uniform distribution

- Conversion from spherical to Cartesian coordinates yields ω_m
- Compute probability of ω_m

$$pdf(\omega_m) = \frac{\alpha^2 \cos\theta_m}{\pi\left((\cos\theta_m)^2(\alpha^2-1)+1\right)^2} = D(\alpha,\theta_m)\cos\theta_m$$

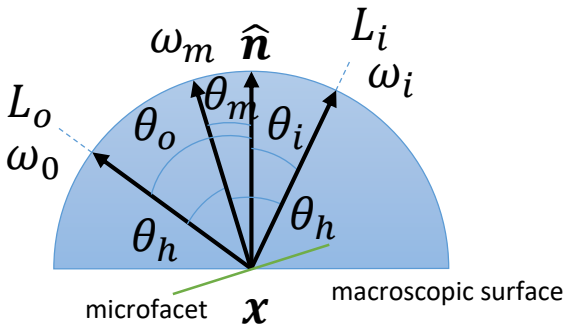
- Get the probability of ω_i via transformation

$$pdf(\omega_i) = \frac{pdf(\omega_m)}{4\cos\theta_h} = \frac{D(\alpha,\theta_m)\cos\theta_m}{4\cos\theta_h}$$

- Finally, get new sample direction ω_i

$$\omega_i = reflect(\omega_o, \omega_m)$$

Fresnel Reflection Coefficient



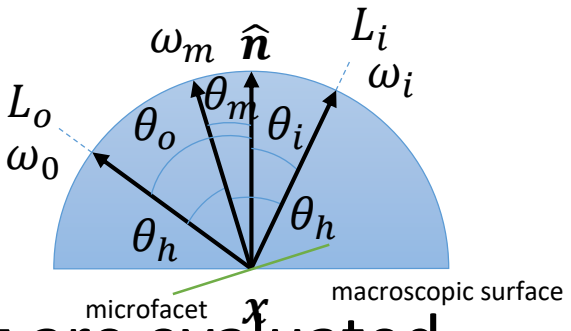
$$F(\theta_h) = F_0 + (1 - F_0)(1 - \cos\theta_h)^5$$

where $\cos\theta_h = \omega_h \cdot \omega_o = \omega_h \cdot \omega_i$ and F_0 is the specular reflectance at normal incidence (or $F_0 = \left(\frac{1-ior}{1+ior}\right)^2$)

In Schlick's approximation, we can replace the costly power by using the spherical Gaussian approximation to obtain a slightly more efficient formula

$$F(\theta_h) = F_0 + (1 - F_0)2^{(-5.55473\cos\theta_h - 6.98316)\cos\theta_h}$$

Geometric Attenuation Factor



- Height correlated form, where masking and shadowing are evaluated separately using the Smith G_1 function

$$G_2^{GGX}(\alpha, \theta_o, \theta_i) =$$

$$= \frac{2 \cos\theta_o \cos\theta_i}{\cos\theta_o \sqrt{\alpha^2 + (1-\alpha^2)(\cos\theta_i)^2} + \cos\theta_i \sqrt{\alpha^2 + (1-\alpha^2)(\cos\theta_o)^2}}$$

G_1 gives the fraction of microfacets with normal ω_m that are visible along the view vector ω_o, resp. ω_i

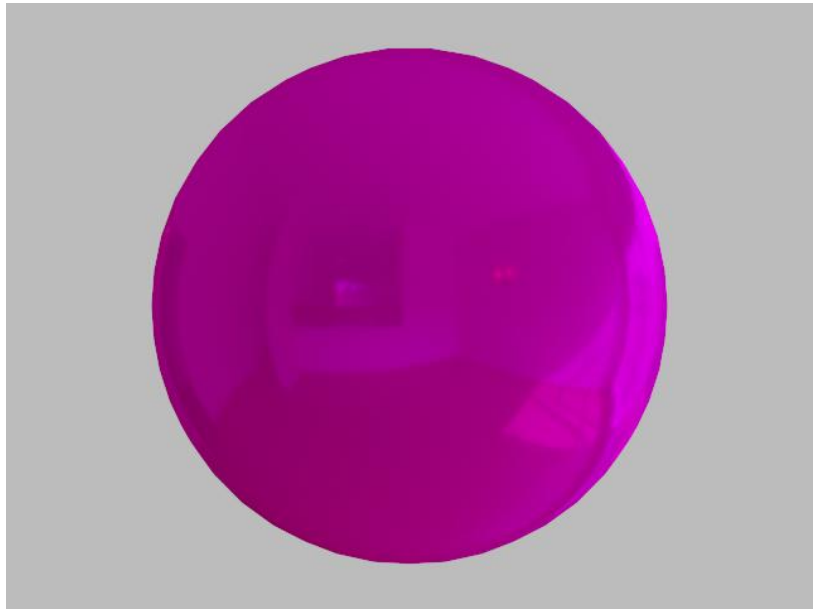
Our PBR Workflow: Metalness-Roughness

- Plastics

Ni = **2.0**; Pr = 0.01

Kd = (1, 0, 1); Ks = (1, **1**, 1)

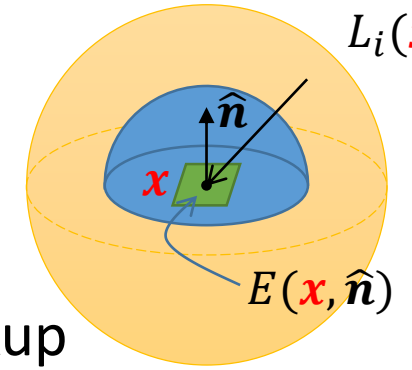
- Metals



Ni = **0.1**; Pr = 0.01

Kd = (1, 0, 1); Ks = (1, **0**, 1)

Diffuse Part and Irradiance Map

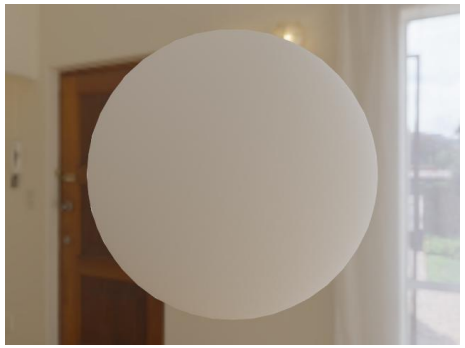


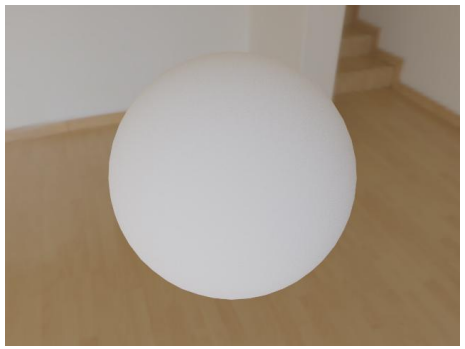
- Replace sampling of diffuse term by a single 2D irradiance map lookup

$$L_r^D(\boldsymbol{x}, \omega_0) = \frac{albedo}{\pi} E(\boldsymbol{x}, \hat{\boldsymbol{n}})$$

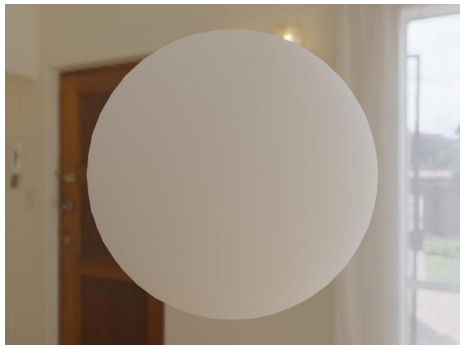
$L_i(\mathbf{0}, \omega_i)$

Hi-res HDR spherical background





Path traced reference (200 spp)



OpenGL (1 spp)

Precompute irradiance

$$E(\boldsymbol{x}, \hat{\boldsymbol{n}}) = \int_{\Omega(\hat{\boldsymbol{n}})} L_i(\boldsymbol{x}, \omega_i) \cos\theta_i \mathrm{d}\omega_i$$

Low-res HDR irradiance map

Diffuse Part and Irradiance Map

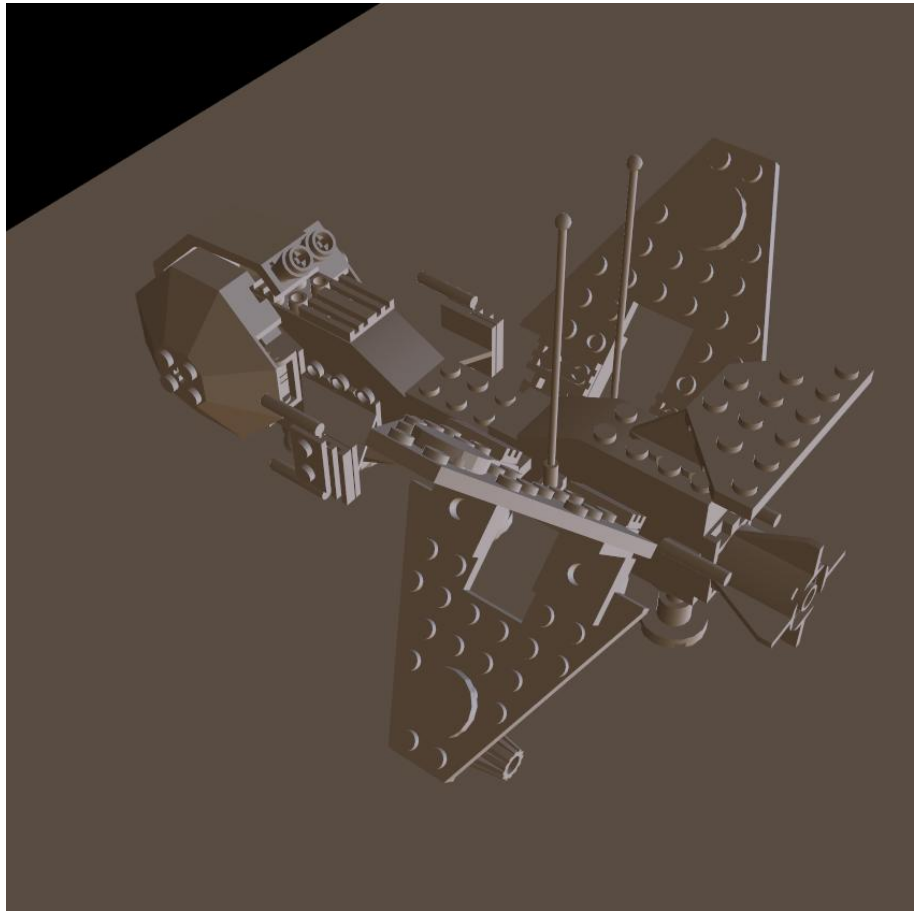
- In case of cosine-weighted sampling of ω_i, we can reduce the amount of computational costs a bit further

$$L_r^D(\boldsymbol{x}, \omega_0) = \frac{albedo}{\cancel{\pi}} \left[E'(\boldsymbol{x}, \hat{\boldsymbol{n}}) \approx \frac{1}{N} \sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i) \frac{\cancel{\cos\theta_i}}{\left[pdf(\omega_i) = \frac{\cancel{\cos\theta_i}}{\cancel{\pi}} \right]} \right]$$

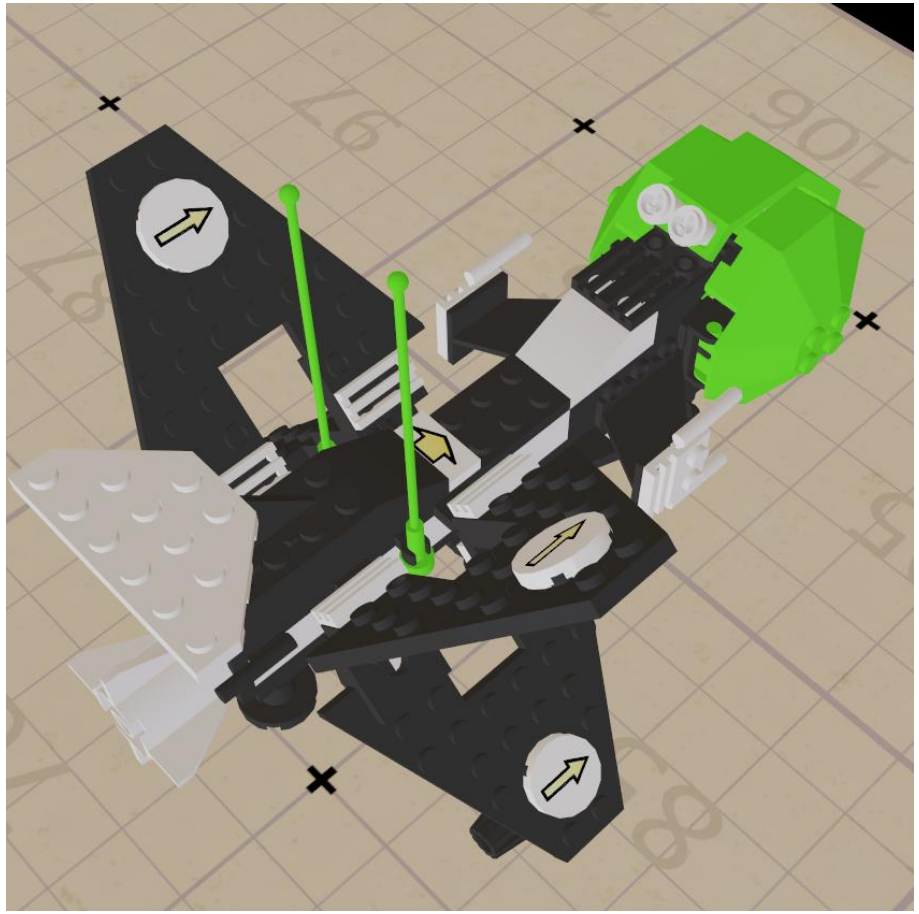
Low-res HDR irradiance map
$E(\hat{\boldsymbol{n}})$ (with π)

Low-res HDR irradiance map
$E'(\hat{\boldsymbol{n}})$ (without π)

Diffuse Part and Irradiance Map



$E(\hat{\boldsymbol{n}}) \cdot 0.15$



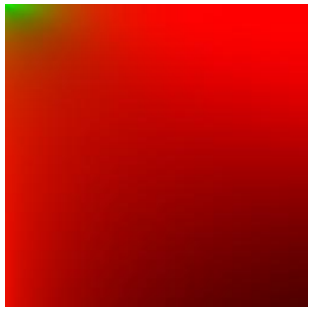
$ToneMapping\left(L_r^D(\boldsymbol{x}, \omega_0)\right)$

Specular Part

- Specular part is not constant over the integral (depends on ω_o and ω_i)
- Split sum approximation (by Epic Games) - this approximation is exact for a constant L_i and fairly accurate for common environments
- Many samples still need to be taken but each separate sum can be **precalculated**

Pre-filtered environment map BRDF integration map

$$L_r^S(\boldsymbol{x}, \omega_0) \approx \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i) f_r(\omega_o, \omega_i) \frac{\cos\theta_i}{pdf(\omega_i)} \approx \left(\frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i)\right) \cdot \left(\frac{1}{N}\sum_{i=1}^{N} f_r(\omega_o, \omega_i) \frac{\cos\theta_i}{pdf(\omega_i)}\right)$$



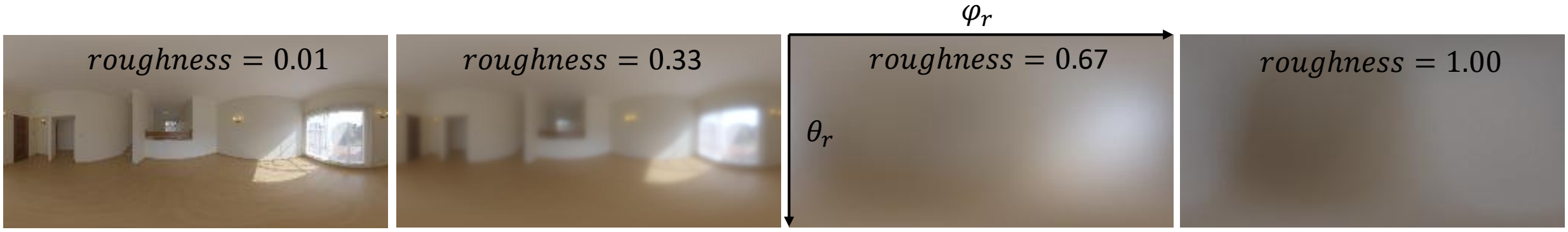
Pre-filtered Environment Map

When dealing with a beam that is nearly parallel to a surface, it is sometimes more useful to refer to the angle between the beam and the surface, rather than that between the beam and the surface normal, in other words 90° minus the angle of incidence. This small angle is called a **glancing angle** or **grazing angle**. Incidence at grazing angles is called "grazing incidence".
Source: https://en.wikipedia.org/wiki/Angle_of_incidence_(optics)

- We assume that $\omega_o = \hat{\boldsymbol{n}} = \omega_r$ (map is unaware of viewing direction, this approximation will remove lengthy reflections at grazing angles)

$$PrefEnvMap(\omega_r, \alpha) = \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}, \omega_i)$$

where the direction ω_i comes from GGX sampling of specular lobe around ω_r and the „width“ of specular lobe is given by roughness



Pre-filtered Environment Map

$alpha = 0.999, 0.750, 0.500, 0.250, 0.100, 0.010, 0.001$

maps look too blurry

$roughness = 0.999, 0.750, 0.500, 0.250, 0.100, 0.010, 0.001$

$alpha = roughness^2$

maps have better distribution

$$\left\|\left(\sqrt{1-(\cos\theta_o)^2}, 0, \cos\theta_o\right)\right\| = \sqrt{1-(\cos\theta_o)^2+(\cos\theta_o)^2} = 1$$

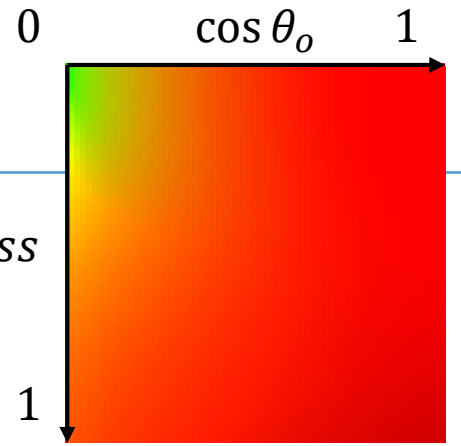
BRDF Integration Map 1/6

- This low-res map is generated for every BRDF we want to use in our shader

$$BRDFIntMap(\cos\theta_o, \alpha) = \frac{1}{N}\sum [L_i(\boldsymbol{x}, \omega_i) = 1] f_r^{CT}(\omega_o, \omega_i)\frac{\cos\theta_i}{pdf(\omega_i)}$$

Here we assume $\hat{\boldsymbol{n}} = (0, 0, 1)$, thus $\omega_o = \left(\sqrt{1-(\cos\theta_o)^2}, 0, \cos\theta_o\right)$ and, like in the previous case, directions ω_i and ω_h come from GGX sampling of specular lobe around $\hat{\boldsymbol{n}}$ and the "width" of lobe is given by roughness

Red channel is a scale and green represents a bias value to the surface's Fresnel response, i.e. $F_0 s + b$



Motivation: we would like to isolate the part that depends on the specular color of the material, which is controlled by the Fresnel term F, so that the resulting integration map depends only on the C-T BRDF

BRDF Integration Map 2/6

- Derivation of the scale s and bias b…

$$\int_{\Omega} \left[f_r^{CT}(\omega_o, \omega_i) = \frac{DFG}{4 \cos\theta_0 \cos\theta_i} \right] \cos\theta_i \, \mathrm{d}\omega_i = \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} F \cos\theta_i \, \mathrm{d}\omega_i =$$

Extract F term from the f_r^{CT}

$$= \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [F \approx F_0 + (1 - F_0) \overbrace{(1 - \cos\theta_h)^5}^{A}] \cos\theta_i \, \mathrm{d}\omega_i =$$

Replace F term with Schlick's approx.

$$= \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [F_0 + (1 - F_0)A] \cos\theta_i \, \mathrm{d}\omega_i = \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [F_0(1 - A) + A] \cos\theta_i \, \mathrm{d}\omega_i =$$

$$= \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [F_0(1 - A)] \cos\theta_i \, \mathrm{d}\omega_i + \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [A] \cos\theta_i \, \mathrm{d}\omega_i =$$

Additional rule - integral of a sum of two functions is the sum of the integrals of each function

$$= F_0 \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [1 - A] \cos\theta_i \, \mathrm{d}\omega_i + \int_{\Omega} \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [A] \cos\theta_i \, \mathrm{d}\omega_i =$$

Extract the specular reflectance F_0

BRDF Integration Map 3/6

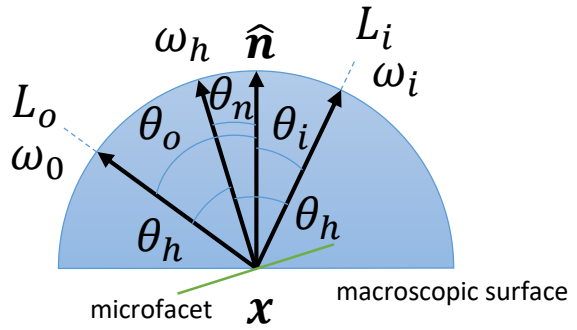
$$= F_0 \int_\Omega \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [1 - A] \cos\theta_i \, \mathrm{d}\omega_i + \int_\Omega \frac{f_r^{CT}(\omega_o, \omega_i)}{F} [A] \cos\theta_i \, \mathrm{d}\omega_i =$$

Plug the f_r^{CT}

$$= F_0 \int_\Omega \frac{D\cancel{F}G}{4 \cos\theta_0 \cancel{\cos\theta_i F}} [1 - A] \cancel{\cos\theta_i} \, \mathrm{d}\omega_i + \int_\Omega \frac{D\cancel{F}G}{4 \cos\theta_0 \cancel{\cos\theta_i F}} [A] \cancel{\cos\theta_i} \, \mathrm{d}\omega_i =$$

$$= F_0 \int_\Omega \frac{DG}{4 \cos\theta_0} [1 - A] \mathrm{d}\omega_i + \int_\Omega \frac{DG}{4 \cos\theta_0} [A] \mathrm{d}\omega_i = F_0 s + b$$

BRDF Integration Map 4/6

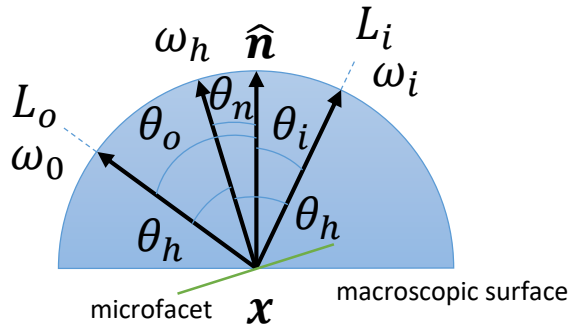


$$s = \int_{\Omega} \frac{DG}{4\cos\theta_0}[1 - A]\mathrm{d}\omega_i \approx \frac{1}{N}\sum^{N} \frac{DG}{4\cos\theta_0}[1 - A]\frac{1}{pdf(\omega_i)} =$$

$$= \frac{1}{N}\sum^{N} \frac{\cancel{D}G}{4\cos\theta_0}[1 - A]\frac{4\cos\theta_h}{\cancel{D}\cos\theta_n} = \frac{1}{N}\sum^{N} \frac{G\cos\theta_h}{\cos\theta_0\cos\theta_n}[(1 - (1 - \cos\theta_h)^5)]$$

Note that $pdf(\omega_i) = \frac{pdf(\omega_h)}{4\cos\theta_h} = \frac{D(\theta_n)\cos\theta_n}{4\cos\theta_h}$

BRDF Integration Map 5/6

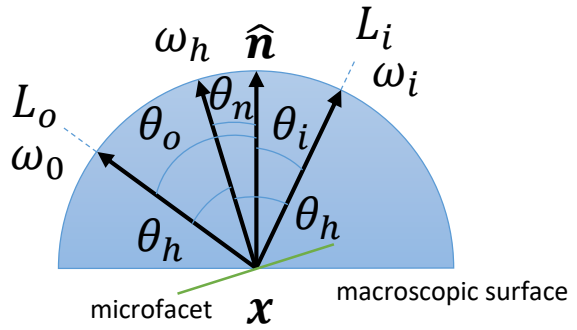


$$b = \int_{\Omega} \frac{DG}{4\cos\theta_0}[A]\mathrm{d}\omega_i \approx \frac{1}{N}\sum^{N} \frac{DG}{4\cos\theta_0}[A]\frac{1}{pdf(\omega_i)} =$$

$$= \frac{1}{N}\sum^{N} \frac{\cancel{D}G}{4\cos\theta_0}[A]\frac{4\cos\theta_h}{\cancel{D}\cos\theta_n} = \frac{1}{N}\sum^{N} \frac{G\cos\theta_h}{\cos\theta_0\cos\theta_n}[(1-\cos\theta_h)^5]$$

Note that $pdf(\omega_i) = \frac{pdf(\omega_h)}{4\cos\theta_h} = \frac{D(\theta_n)\cos\theta_n}{4\cos\theta_h}$

BRDF Integration Map 6/6



$$\int_{\Omega} f_r^{CT}(\omega_o, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i = F_0 s + b$$

$$BRDFIntMap(\cos\theta_o, \alpha) = \begin{pmatrix} s(\cos\theta_o, \alpha) \\ b(\cos\theta_o, \alpha) \end{pmatrix}$$

Complete PBR IBL Shader

```
newmtl Mat025 # metal/roughness workflow
  Pr 0.4 # roughness
  Pm 1.0 # metallic
  Kd 0.913 0.921 0.925 # albedo (base color)
```

$$L_o(\boldsymbol{x}, \omega_0) = (L_r^D(\boldsymbol{x}, \omega_0) + (F_0 s + b) L_r^S(\boldsymbol{x}, \omega_0)\ )\ ao$$

Optional ambient occlusion

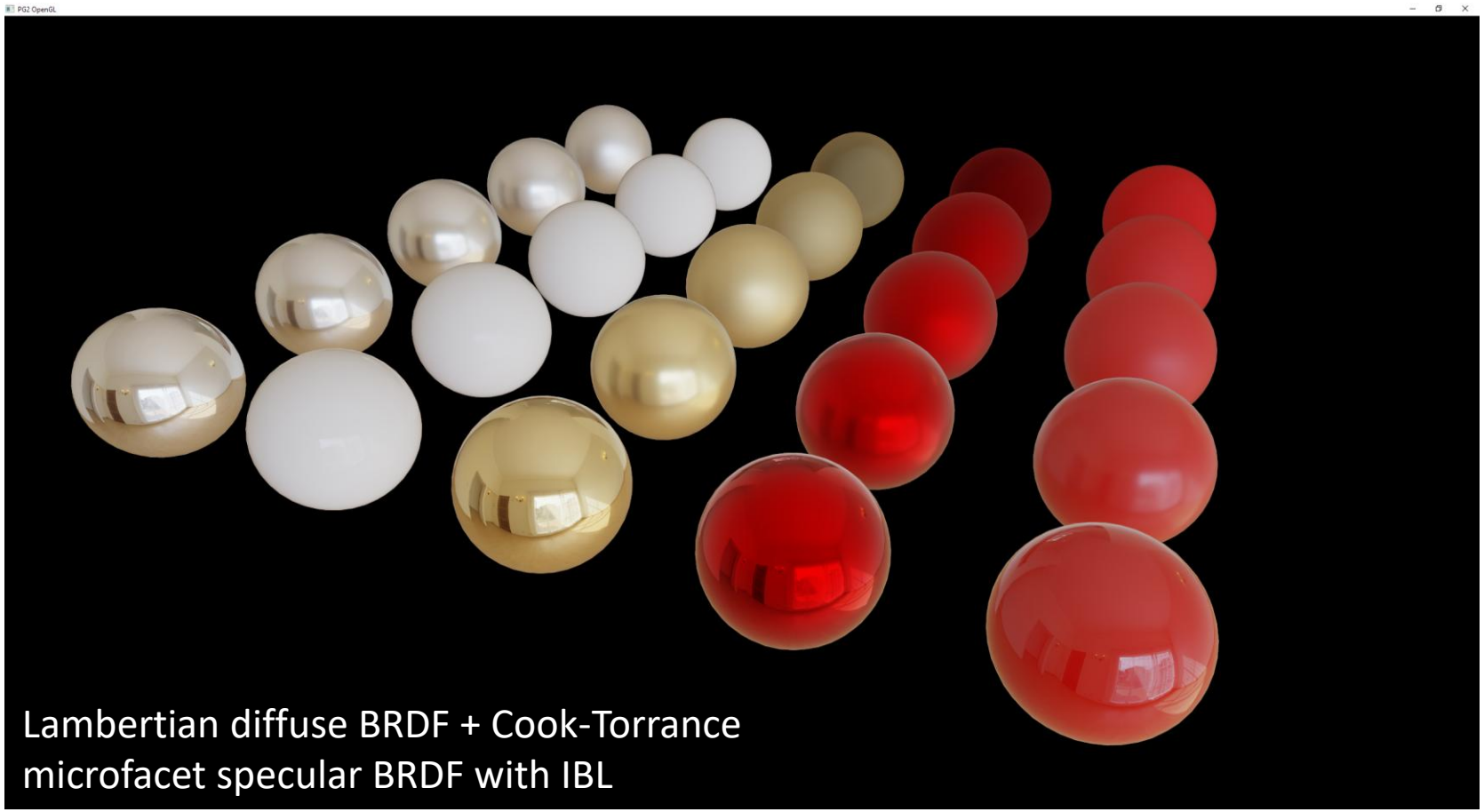
Common Fresnel factor at perpendicular incidence for all dielectrics

- $F_0 = \text{mix}(\text{vec3}(0.04), albedo, metallic)$
- $F_d = \big(vec3(1) - F(\cos\theta_o\,, F_0)\big)(1 - metallic)$
- $(s, b) = BRDFIntMap(\cos\theta_o, roughness)$

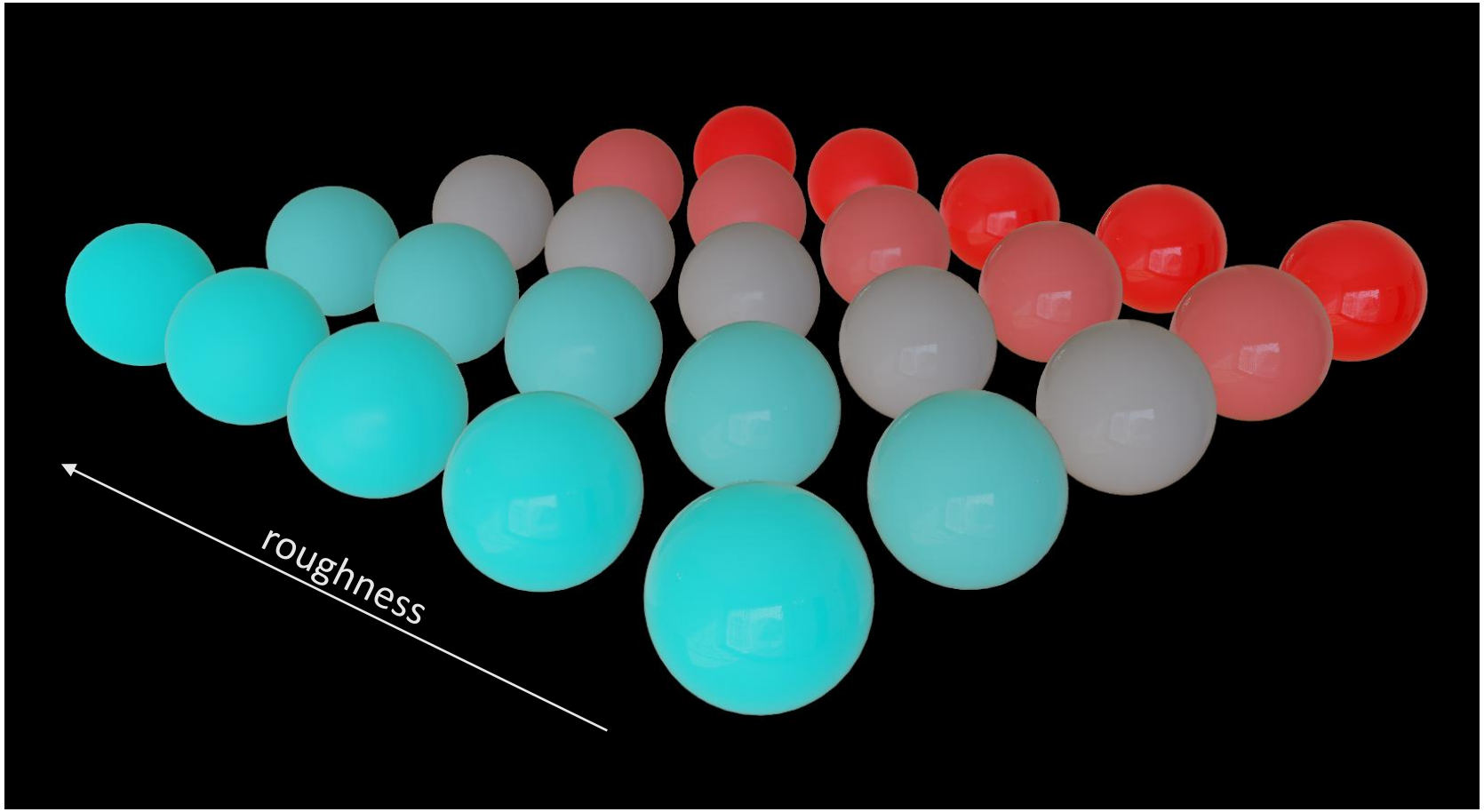
In GLSL, the function
`genType reflect(genType I, genType N)`
returns I - 2 * dot(N, I) * N, so we should write
`vec3 omega_i = reflect(-omega_o, n);`

- $L_r^D(\boldsymbol{x}, \omega_0) = F_d \frac{albedo}{\pi} IrradianceMap(\hat{\boldsymbol{n}})$
- $L_r^S(\boldsymbol{x}, \omega_0) = PrefEnvMap(\omega_i, roughness);\ \omega_i = \text{reflect}(\omega_0, \hat{\boldsymbol{n}})$

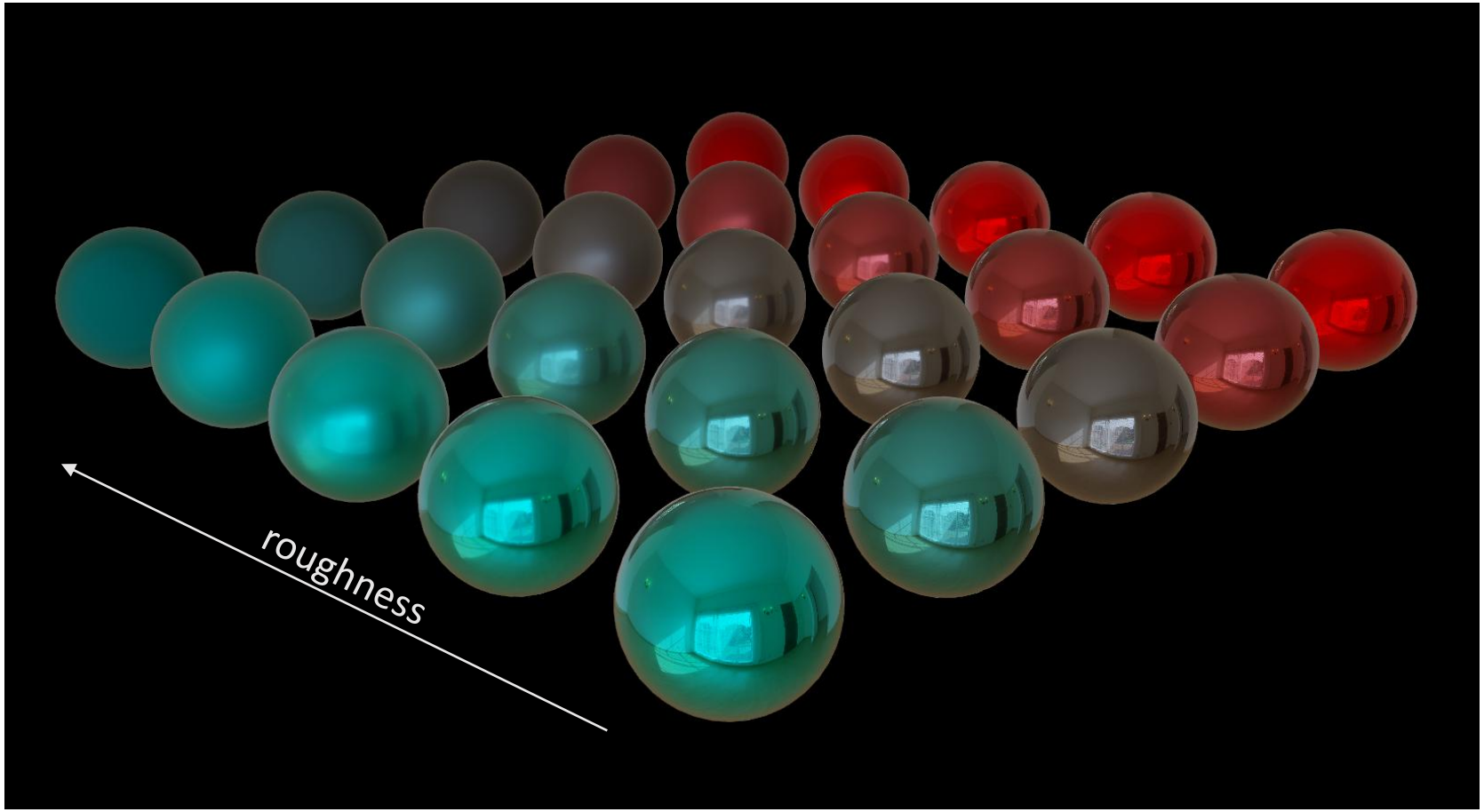
PBR IBL Shader Output



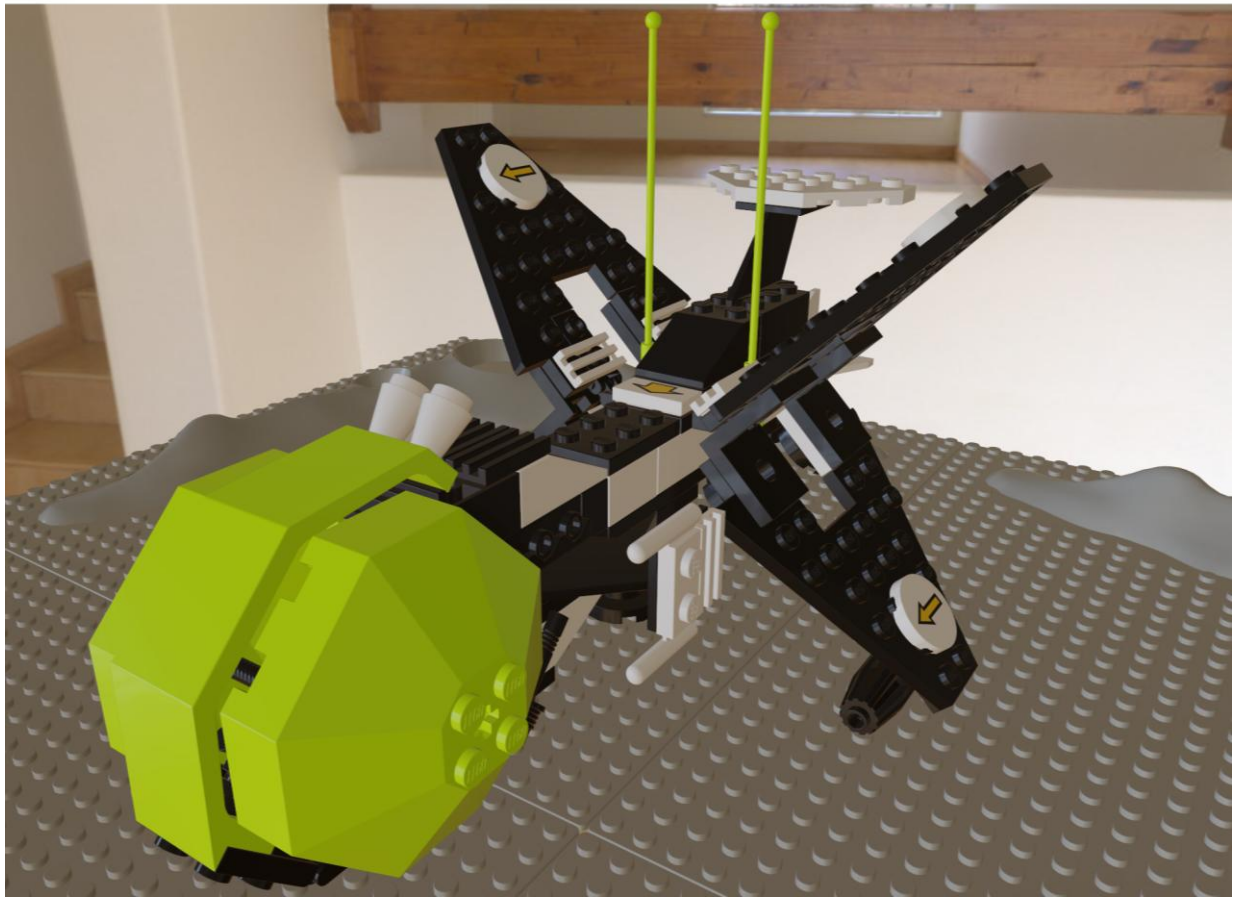
Results (Dielectrics)

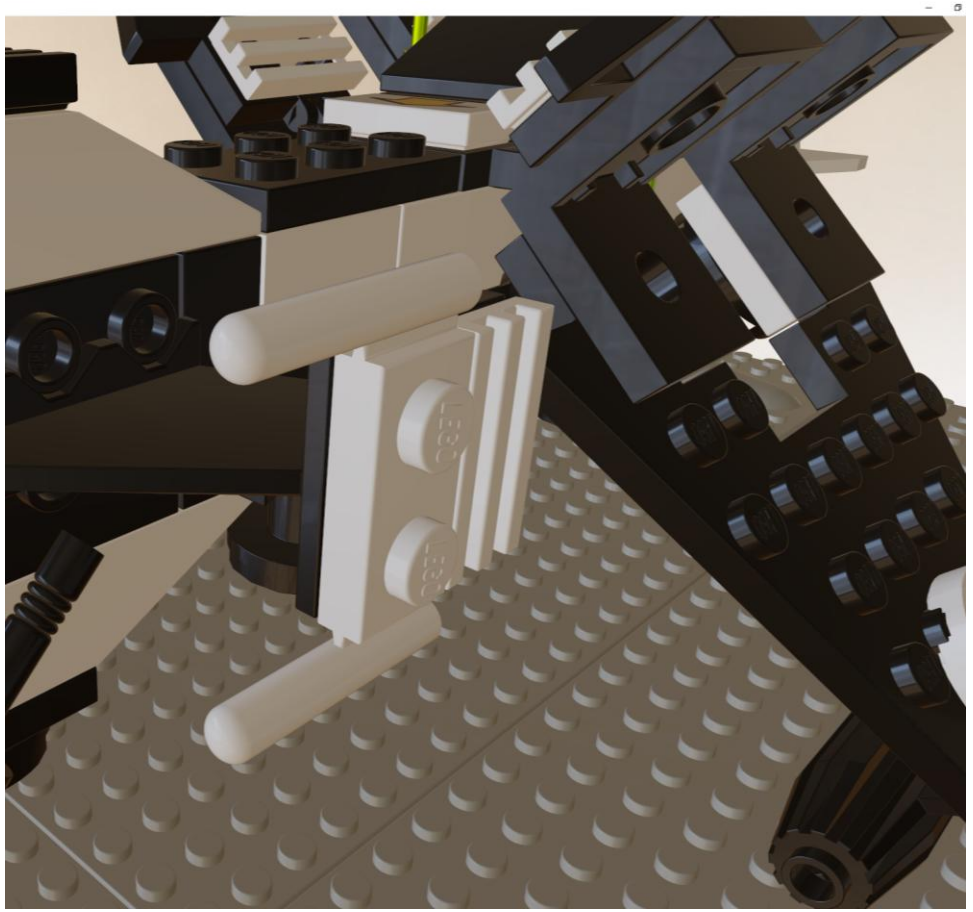


Results (Metals)

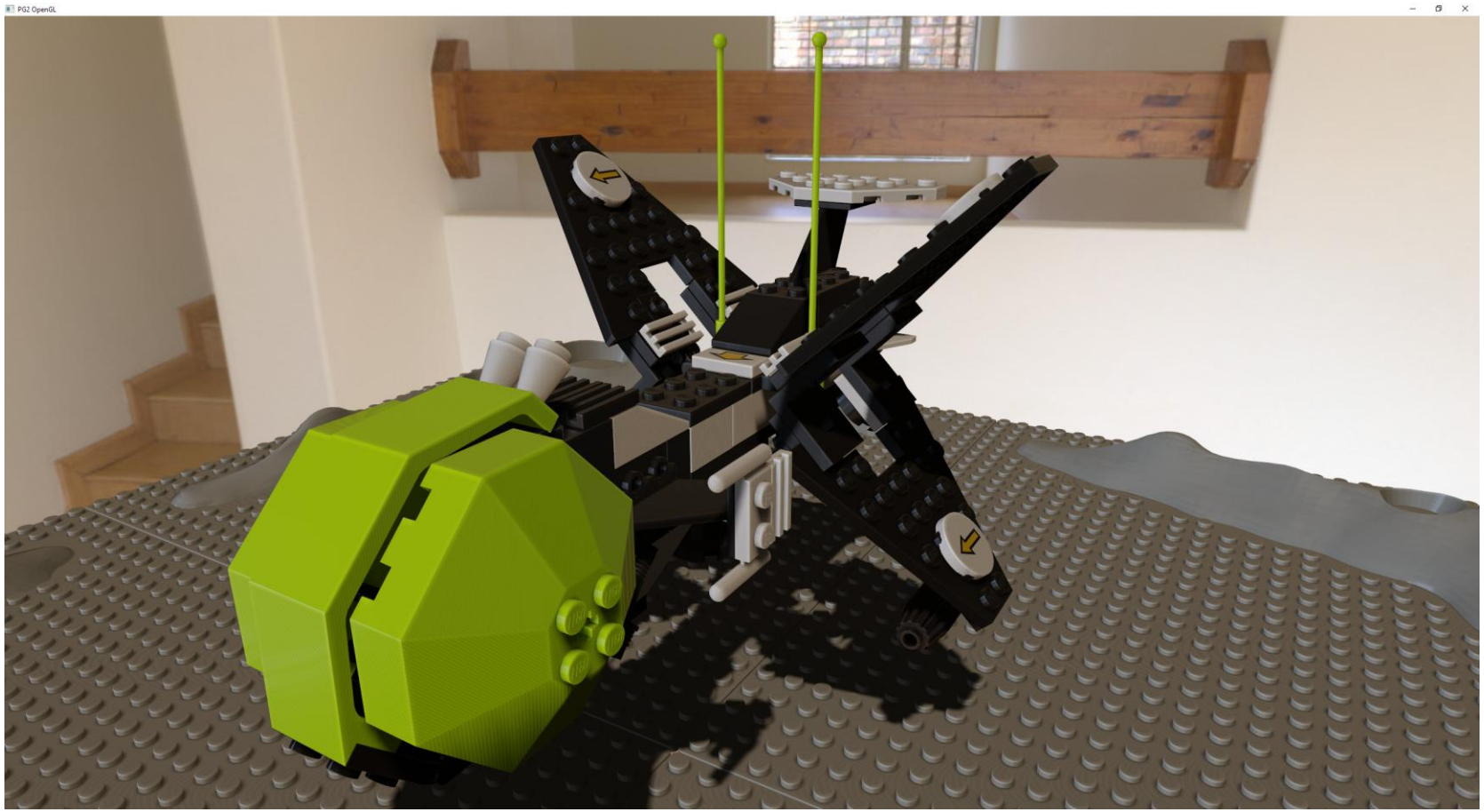


PBR IBL Shader Output





PBR IBL Shader with Shadow Maps Output



Texture Lookup With Explicit Level-of-Detail

- To get interpolated values from the closest levels of $PrefEnvMap$, we can use GLSL function textureLod(tex, uv, lod) where lod is float in the range <0, max_level>
- This function returns interpolated pixel values from closest levels

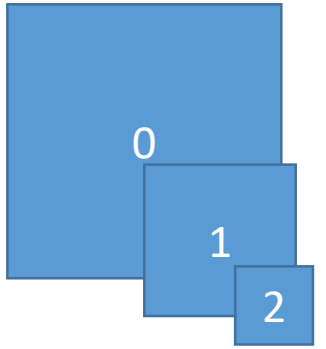
0 base level $width \times height$

1 level $width/2 \times height/2$

2 level $width/4 \times height/4$

…

max level



Texture Lookup With Explicit Level-of-Detail

```
const GLint max_level = GLint( file_names.size() ) - 1; // assume we have a list of images representing different levels of a map

glGenTextures( 1, &tex_prefiltered_env_map_ );
glBindTexture( GL_TEXTURE_2D, tex_prefiltered_env_map_ );
if ( glIsTexture( tex_prefiltered_env_map_ ) ) {
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );

  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0 );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, max_level );

  int width, height;
  GLint level;

  for ( level = 0; level < GLint( file_names.size() ); ++level ) {
    Texture3f prefiltered_env_map = Texture3f( file_names[level] );
    // for HDR images use GL_RGB32F or GL_RGB16F as internal format !!!
    glTexImage2D( GL_TEXTURE_2D, level, GL_RGB32F, prefiltered_env_map.width(), prefiltered_env_map.height(), 0, GL_RGB, GL_FLOAT,
prefiltered_env_map.data() );

    width = prefiltered_env_map.width() / 2;
    height = prefiltered_env_map.height() / 2;
  }
}

glBindTexture( GL_TEXTURE_2D, 0 );
```

Fragment Shader

```
…
uniform sampler2D map; // per scene background
uniform int max_level;
…
vec3 texel = textureLod( map, uv, x * max_level
).rgb; // x is in <0,1>
…
```

General Textures (e.g. RGB 8UC3)

```
// OpenGL texture initialization
glGenTextures( 1, &tex );
glBindTexture( GL_TEXTURE_2D, tex );
if ( glIsTexture( tex ) ) {
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR ); // initial value is GL_NEAREST_MIPMAP_LINEAR
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR ); // initial value is GL_LINEAR

  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB8, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, ptr_data );
  //glGenerateMipmap( GL_TEXTURE_2D ); // note that the default value of GL_TEXTURE_MIN_FILTER is
GL_NEAREST_MIPMAP_LINEAR, so we have to use the build-in mipmap generator
}
glBindTexture( GL_TEXTURE_2D, 0 ); // unbind
…
// In the main loop
…

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, tex_irradiance_map_ );
GLint location = glGetUniformLocation( program, "my_tex" );
glUniform1i( location, 0 );
…
```

Fragment Shader

```
…
uniform sampler2D my_tex;
…
vec3 color = texture( my_tex, uv ).rgb;
…
```

Bump vs Normal vs Displacement Maps

- Bump maps – grayscale height map, a measure of surface indentation or extrusion, phased out in favor of normal maps
- Normal maps – three-channel texture with local normals, add fake details (the silhouettes still look smooth)
- Displacement maps – physically displace the mesh to which they are applied. As a result of this additional geometry, it's hard to beat the results of a displacement map. Since the surface is actually modified, the silhouette reflects the additional geometry

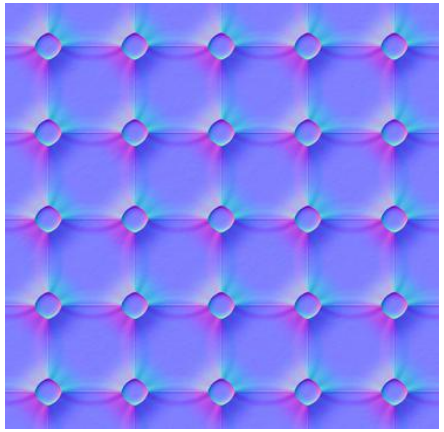
- Combination is possible – utilize displacement for the big changes to the geometry and then the normal or bump for the fine detail

```
vec3 normal_ls = normalize( texture( tex_normal, texcoord ).xyz ) * 2.0f - vec3( 1.0f ) );
```

Normal Mapping

- Technique for simulating bumps and wrinkles on the surface of an object while the geometry stays untouched
- What we need
  - A normal map with normals stored in local (tangent) space

$$\hat{n}_{LS} = 2\ NormalMap(u, v) - (1, 1, 1)^T$$



normal map

  - A way how to transform local normals to world space

We need to find TBN matrix which will convert local normals from normal map to proper world space coordinates of our scene

$$\hat{n}_{WS} = TBN\ \hat{n}_{LS}$$

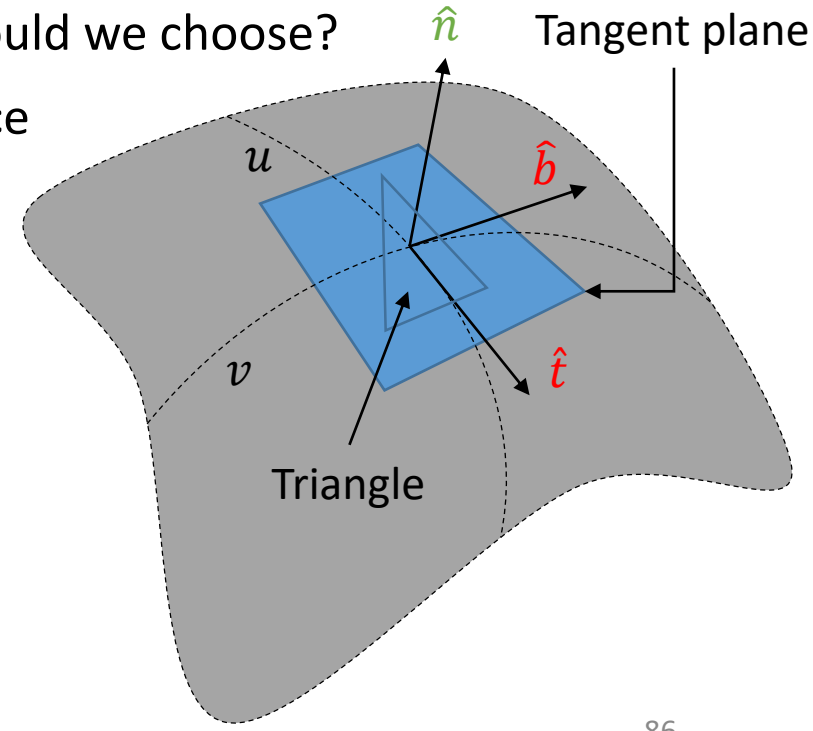
$\hat{n}_{WS}$ will be our new normal we are working with

Local Tangent Space

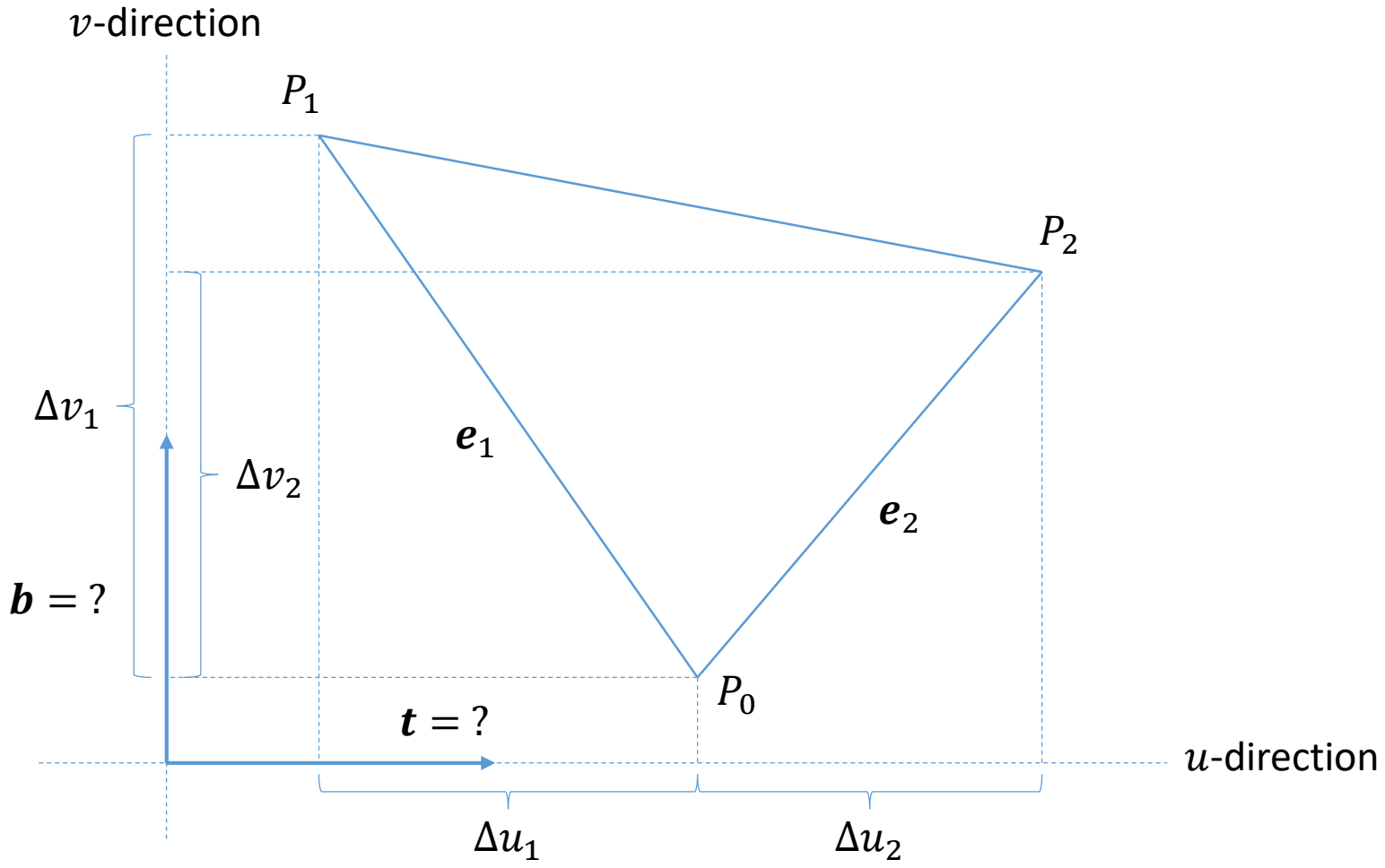
- Defined by three orthonormal vectors – tangent $\hat{t}$, bitangent $\hat{b}$, and normal $\hat{n}$
- Vector $\hat{n}$ is given, so we need to come up with the remaining two vectors
- Vector $\hat{t}$ should be parallel to the surface
- Problem: But there are infinitely many such vectors, which one should we choose?
- Constrain: Our choice should be consistent across the whole surface
- Solution: We may orient the tangent vector in the same direction

that has our u texture coordinate

- After obtaining tangent vector, getting a bitangent is simple



Tangent-Bitangent-Normal



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.

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

P_i is the position of i-th vertex

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

- $\Delta u_1 = P_1^u - P_0^u, \Delta v_1 = P_1^v - P_0^v$
- $\Delta u_2 = P_2^u - P_0^u, \Delta v_2 = P_2^v - P_0^v$

$P_i^{\{u,v\}}$ 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 $\hat{\boldsymbol{n}}$
- But we may safely assume that these three vectors will be nearly orthogonal. To ensure the orthogonality, we may use Gram-Schmidt orthogonalization process.

- 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 the projection operator $\operatorname{proj}_{\widehat{\boldsymbol{u}}}(\boldsymbol{v}) = (\boldsymbol{v} \cdot \widehat{\boldsymbol{u}})\widehat{\boldsymbol{u}}$ projects the vector $\boldsymbol{v}$ orthogonally onto a line spanned by unit vector $\widehat{\boldsymbol{u}}$

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

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

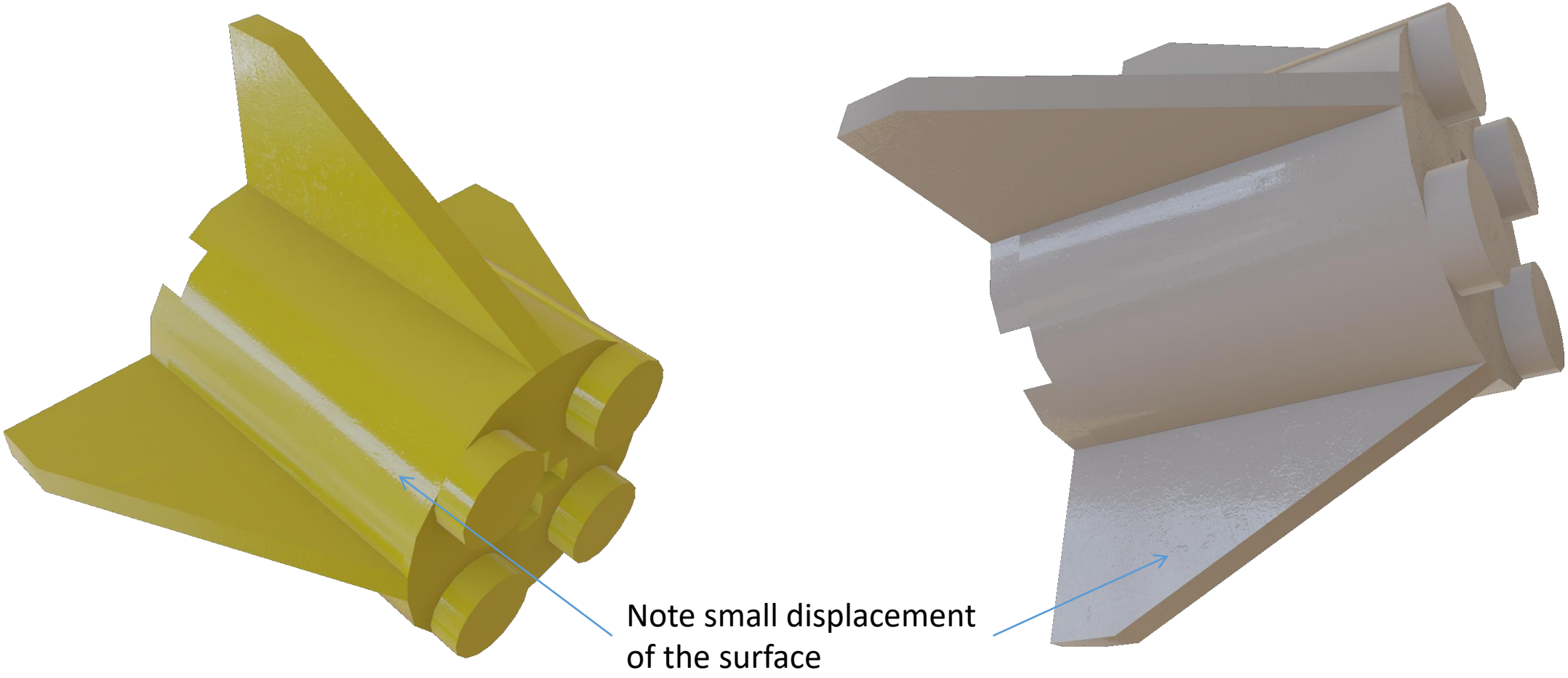
$$\boldsymbol{b}' = \boldsymbol{b} - (\boldsymbol{b} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - (\boldsymbol{b} \cdot \boldsymbol{t}')\boldsymbol{t}'/t'^2$$

  and the new TBN matrix takes the form

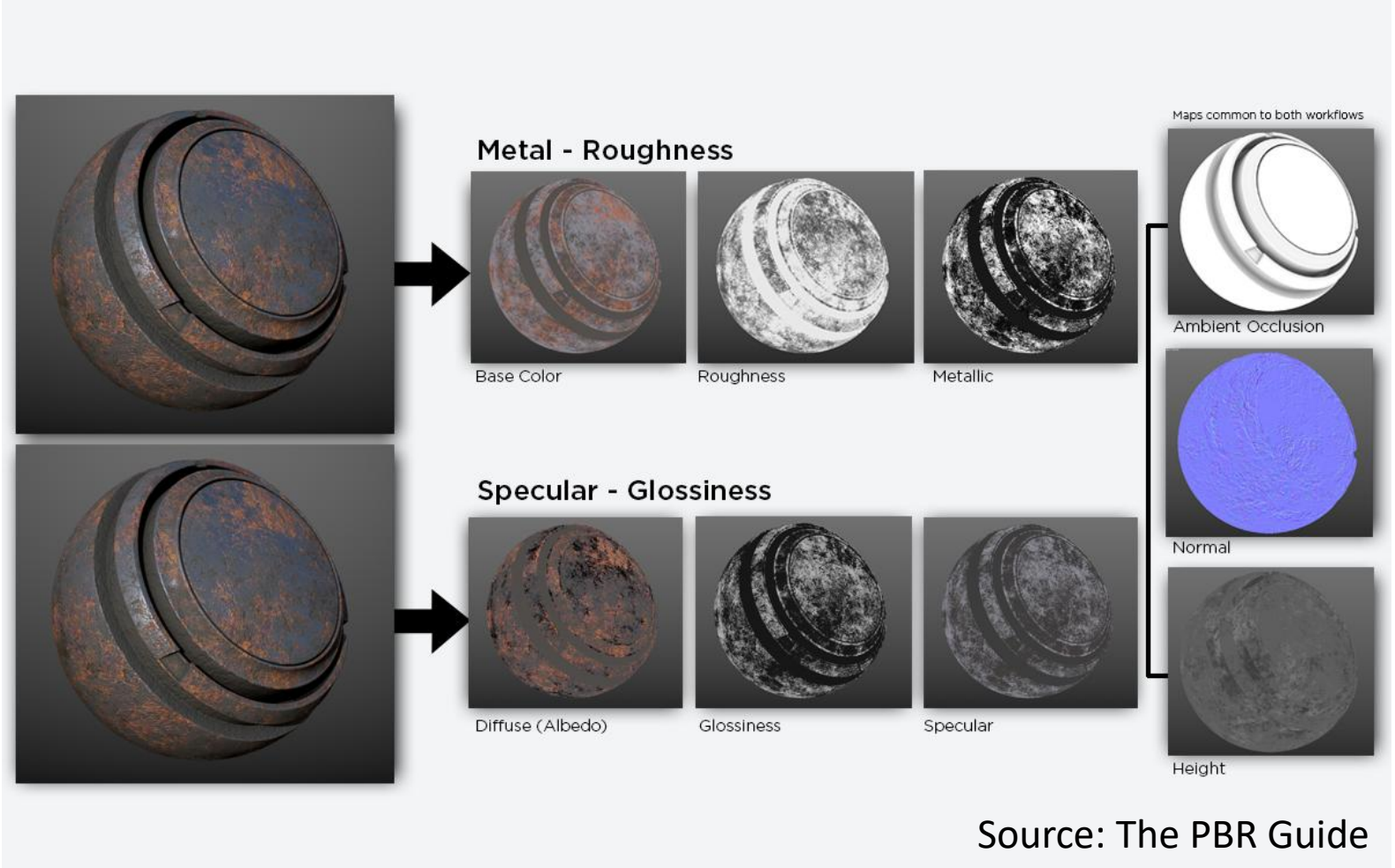
$$TBN_{TS \to WS} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \hat{\boldsymbol{t}}' & \hat{\boldsymbol{b}}' & \hat{\boldsymbol{n}} \\ \vdots & \vdots & \vdots \end{pmatrix}$$

Note that $\hat{\boldsymbol{b}}' = \hat{\boldsymbol{n}} \times \hat{\boldsymbol{t}}'$

Normal Mapping Results



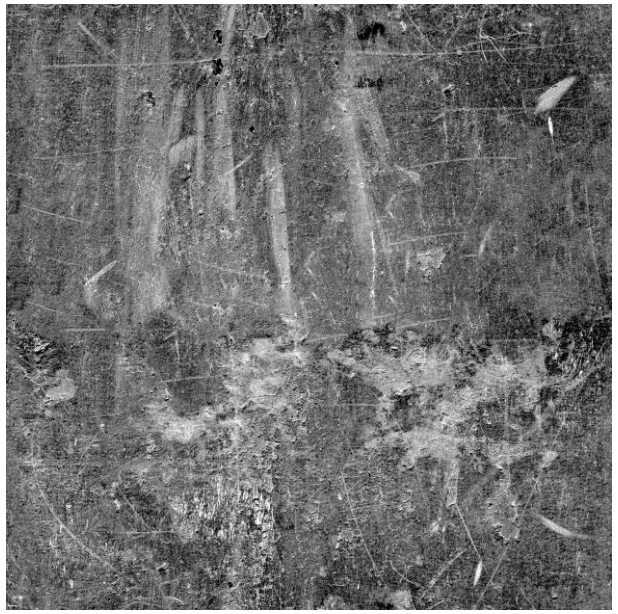
PBR Workflows



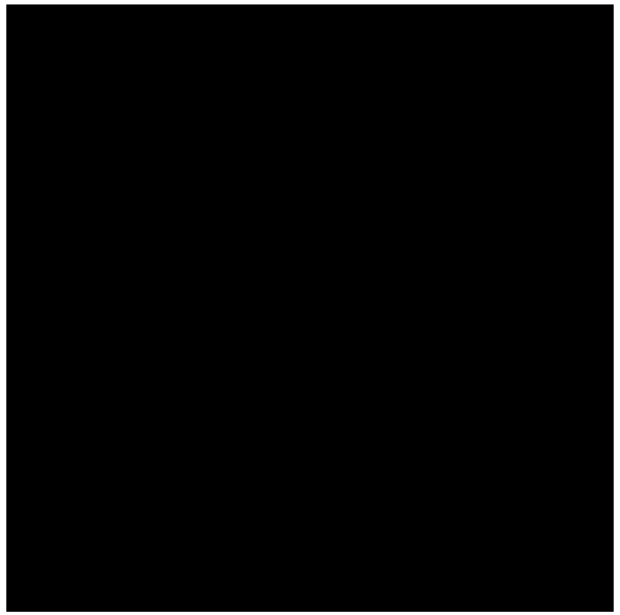
Source: The PBR Guide

RMA Texture

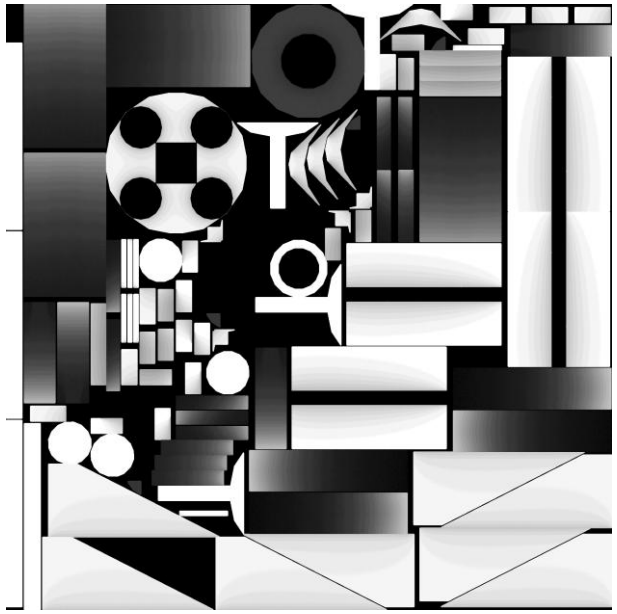
- Metal/roughness workflow: base color (often similar to or the same as the albedo map) + RMA + normal



Roughness



Metallic



Ambient occlusion

Nvidia OptiX

- Low-level ray tracing engine and API exploiting computational potential of modern GPUs (similar to Intel Embree)
- Provides abstraction for the ray tracing pipeline (contains only general parts, not an assembled ray tracer)
- Consist of data structures (API) and user-provided programs (CUDA)
- May operate in conjunction with OpenGL or DirectX as a hybrid ray tracing-rasterization application

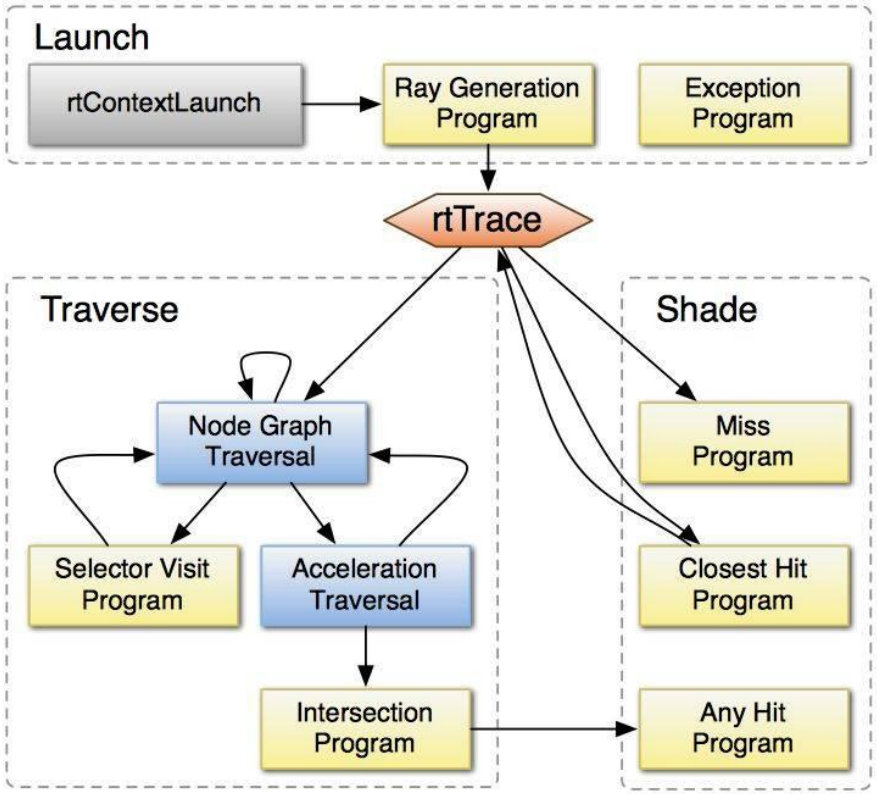
Programming model

Retained = library stores a model of the scene in memory, API can be simpler to use

- OptiX is an object-based C API implementing a simple retained (vs. immediate) mode object hierarchy
- User-provided CUDA C-based functions
  - Ray initialization, intersections, shading, spawning of new rays
  - Compiled to PTX virtual assembly language
- User-specified payload
  - Rays can carry color, recursion depth, importance, …

Programming model

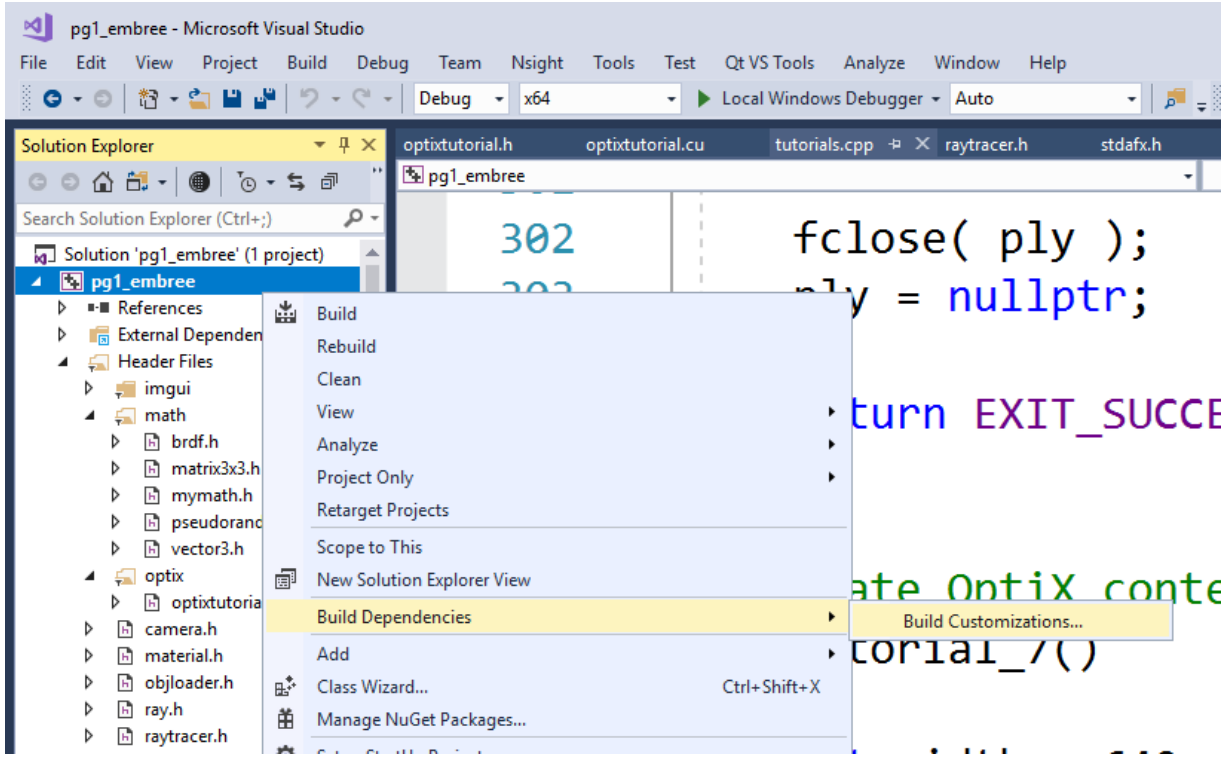
- Object model
  - Objects are created, destroyed, modified and bound with the C API
- Component programs
  - Eight types of programs
- Variables
  - Communicating data to programs
- Buffers
  - Pass data between the host and the device
- Execution model
  - Launching a ray generation program with special semantic variable providing run-time index of current pixel

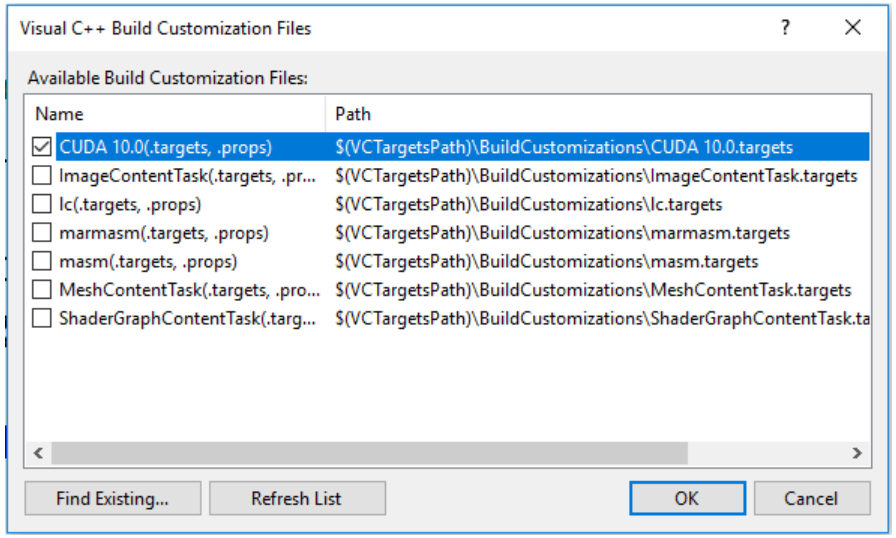


VS 2017 and OptiX 6.0.0

- HW requirements
  - Nvidia GPU Maxwell or newer (2014+), e.g. GeForce GTX 745, GTX 750/750 Ti, GTX 850M/860M (GM107) and GeForce 830M/840M (GM108)
- SW requirements
  - Min. 418.81 display driver under Windows or the 418.30 under Linux
  - CUDA SDK 10.0.130 (also older 9.2 works well)
  - Visual Studio 2017 (tested on 15.9.7 with Windows SDK 10.0.17763.0)

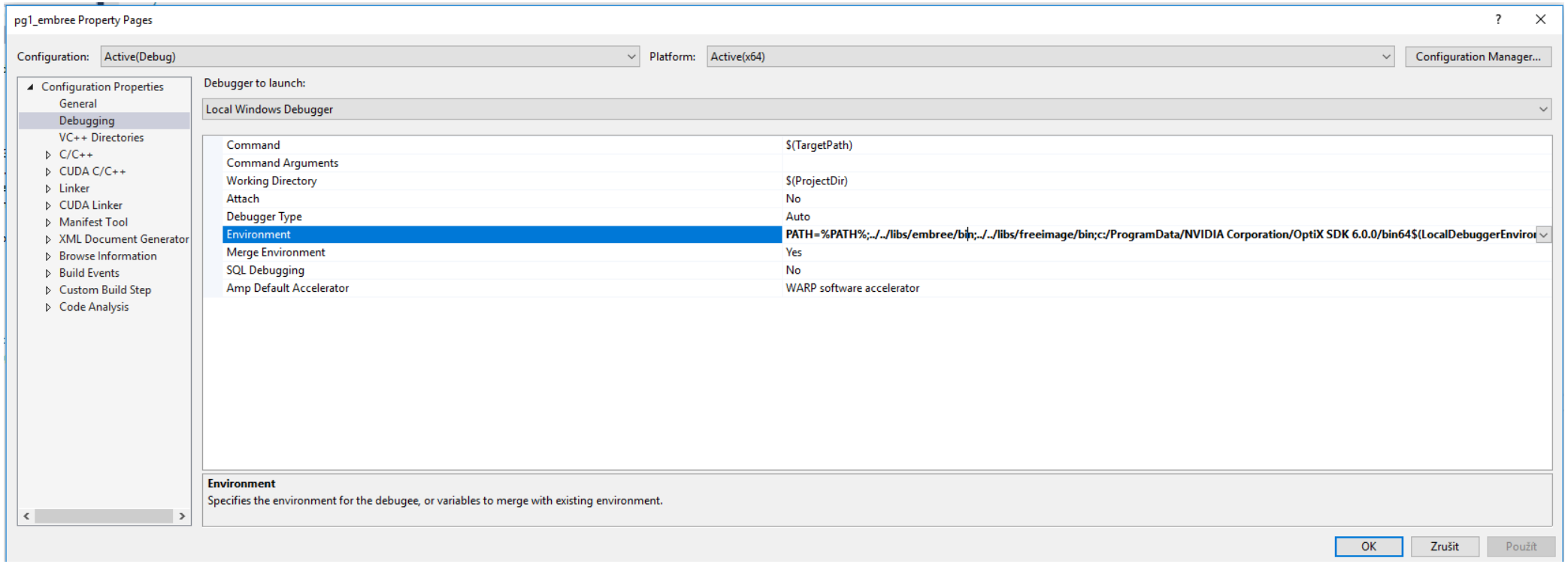
VS 2017 and OptiX





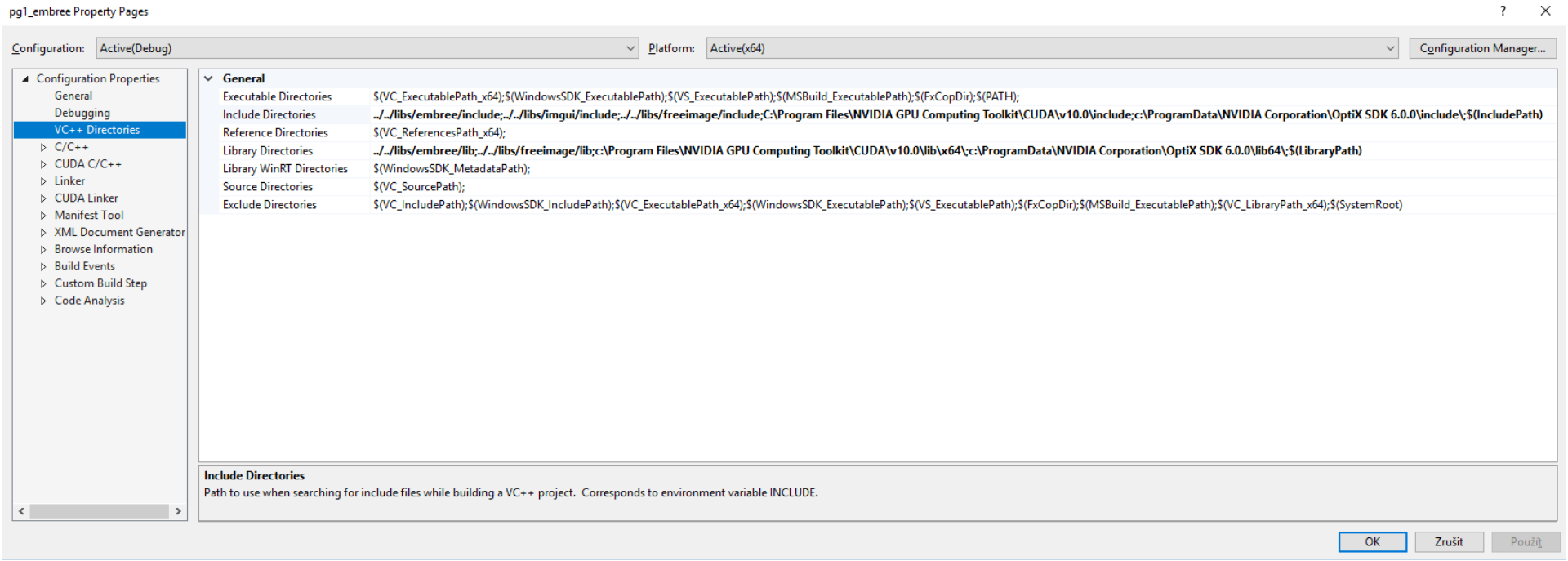
VS 2017 and OptiX

- Dll not found?
- PATH=%PATH%;c:/ProgramData/NVIDIA Corporation/OptiX SDK 6.0.0/bin64



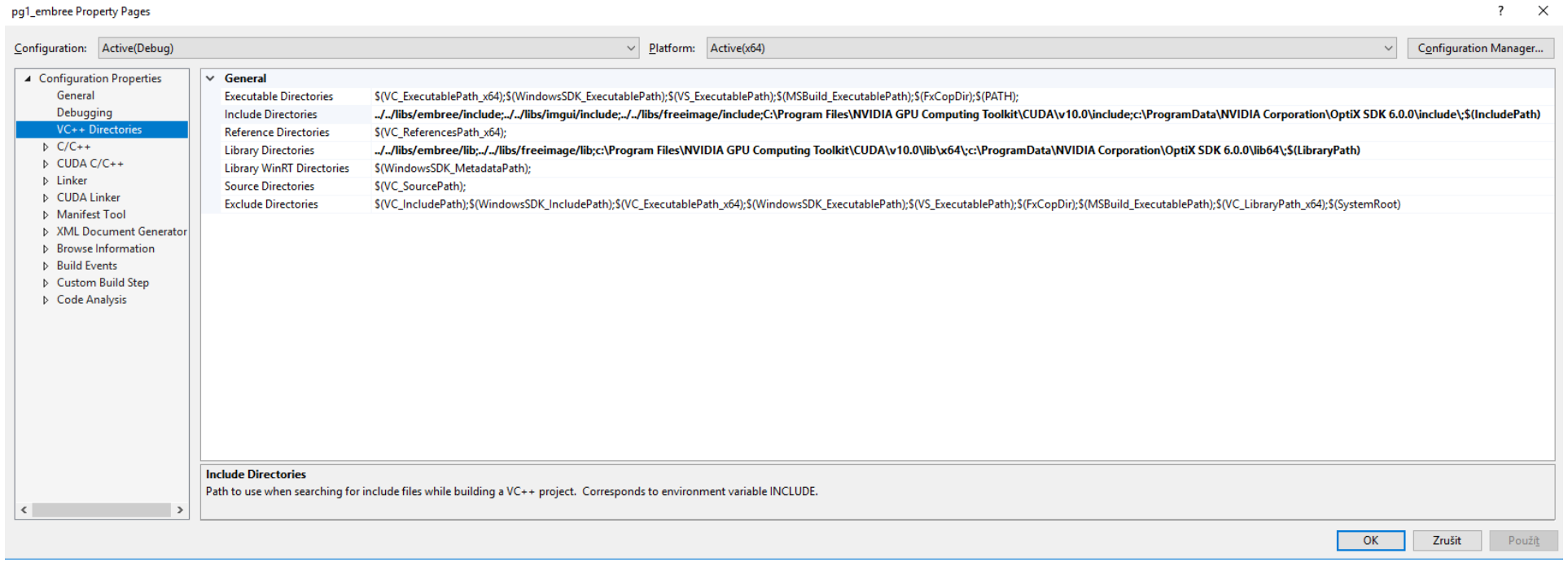
VS 2017 and OptiX

- Header files not found?
- c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\include
- c:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\include



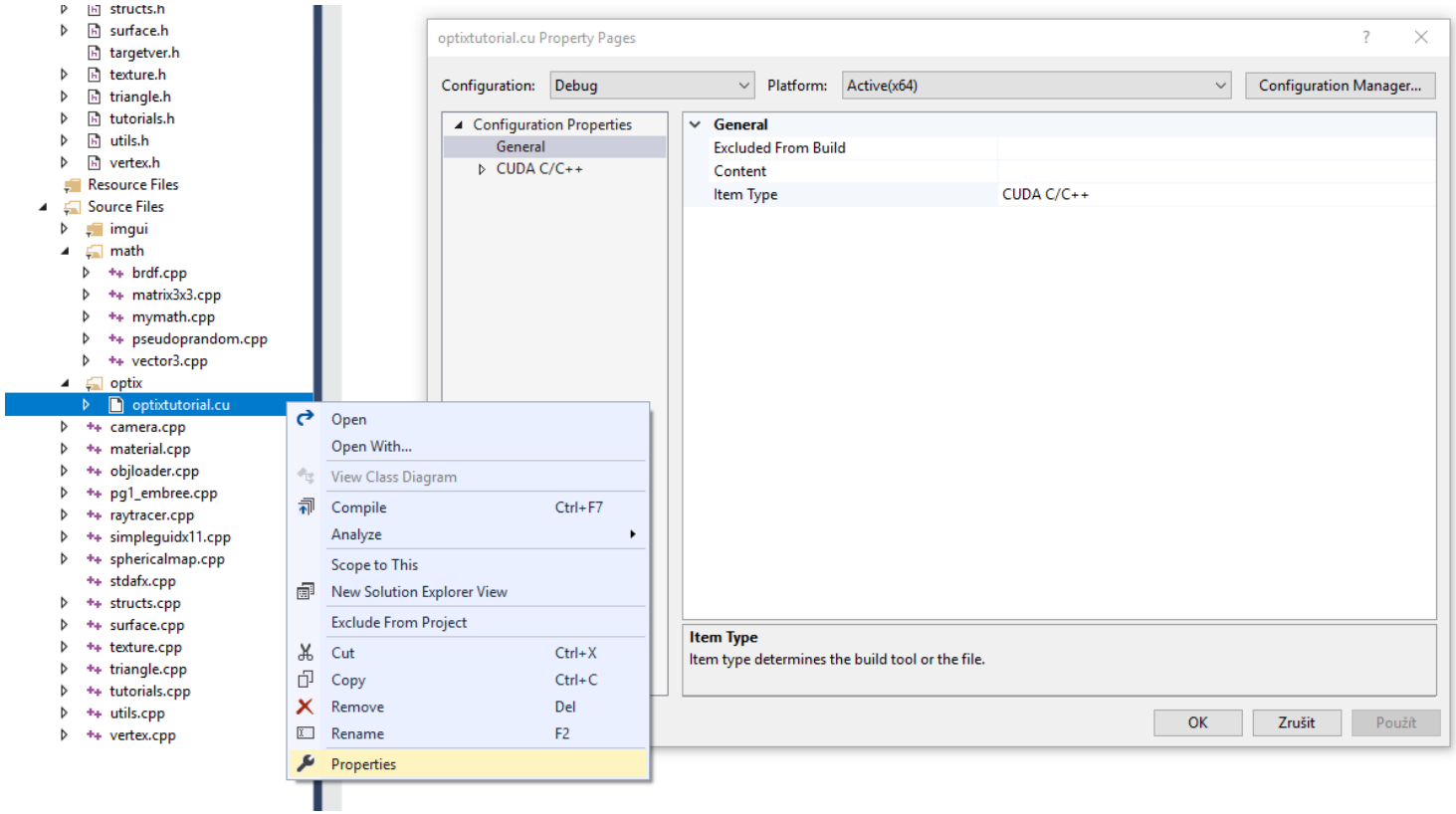
VS 2017 and OptiX

- Static librarie not found?
- c:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\lib64\



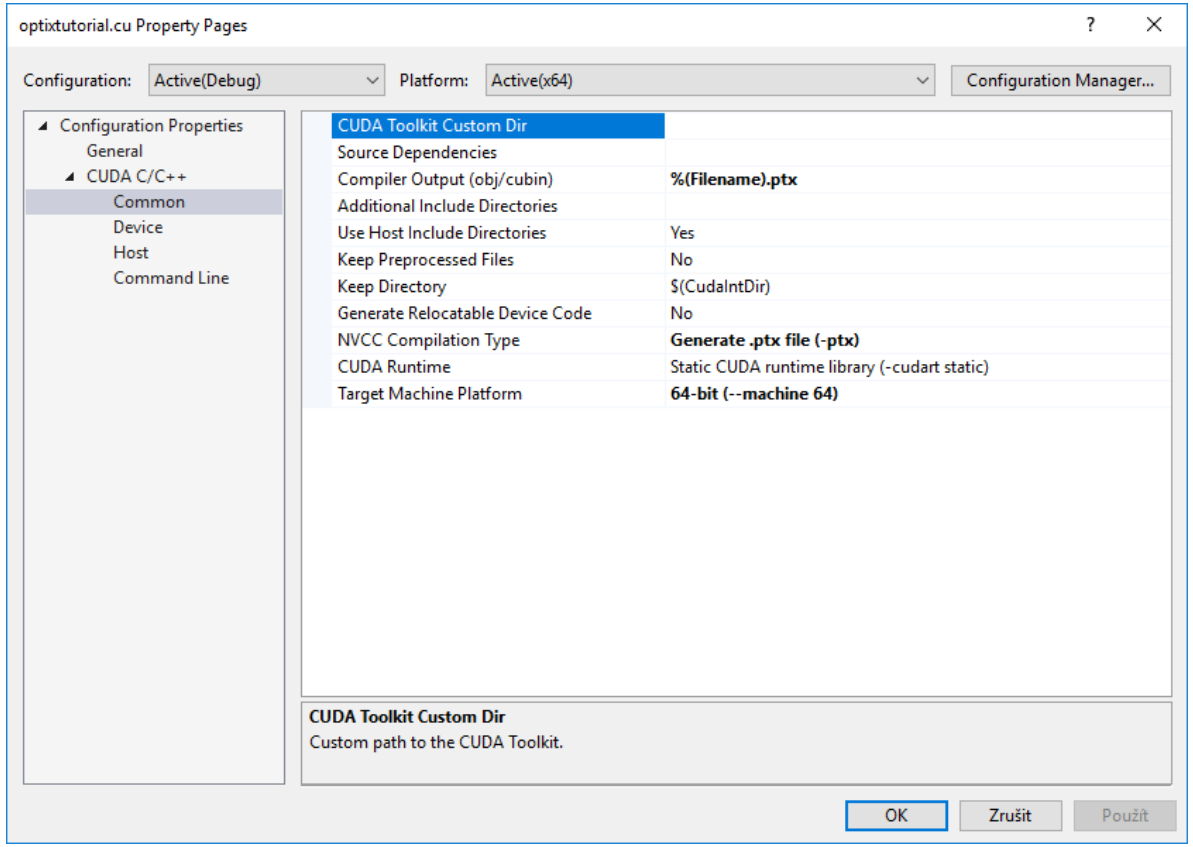
VS 2017 and OptiX

- CU files must be compiled with NVCC (set the field Item Type to CUDA C/C++)



VS 2017 and OptiX

- NVCC compiler output must be set to PTX and 64-bit target



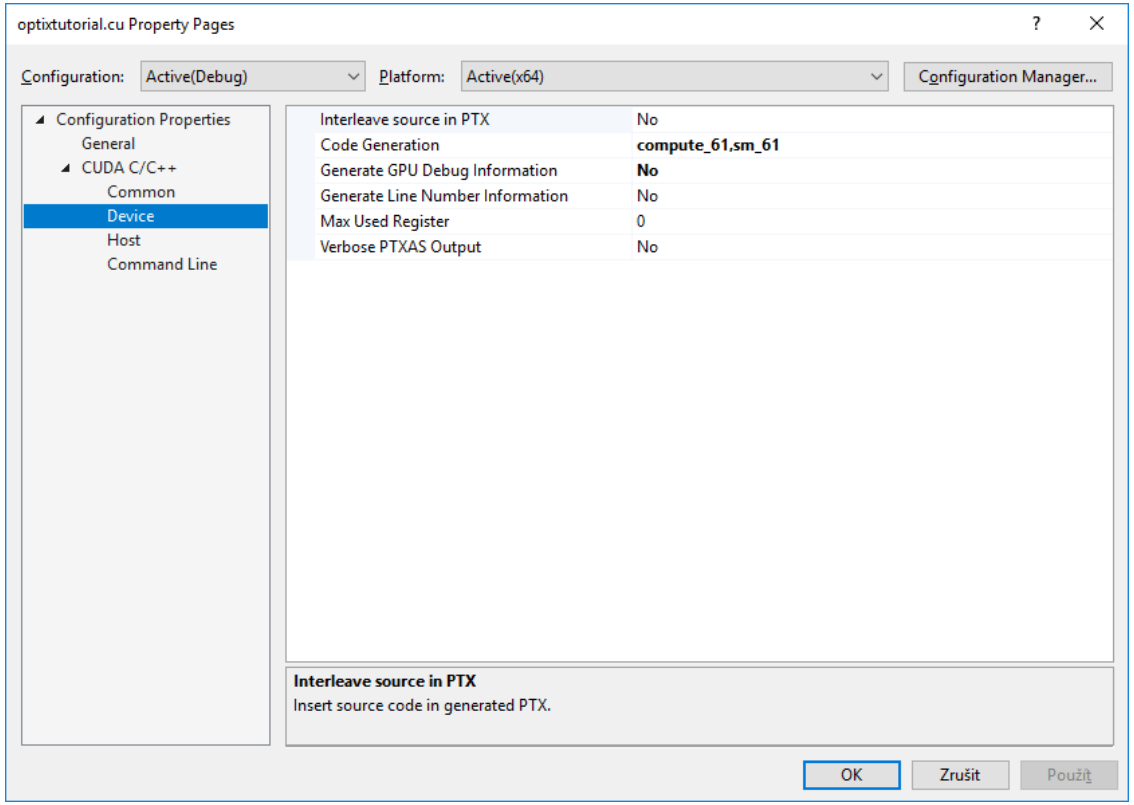
VS 2017 and OptiX

- Make sure to generate the code for your graphics card's level of compute capability

For Maxwell generation
compute_50, sm_50

For Pascal generation
compute_61, sm_61

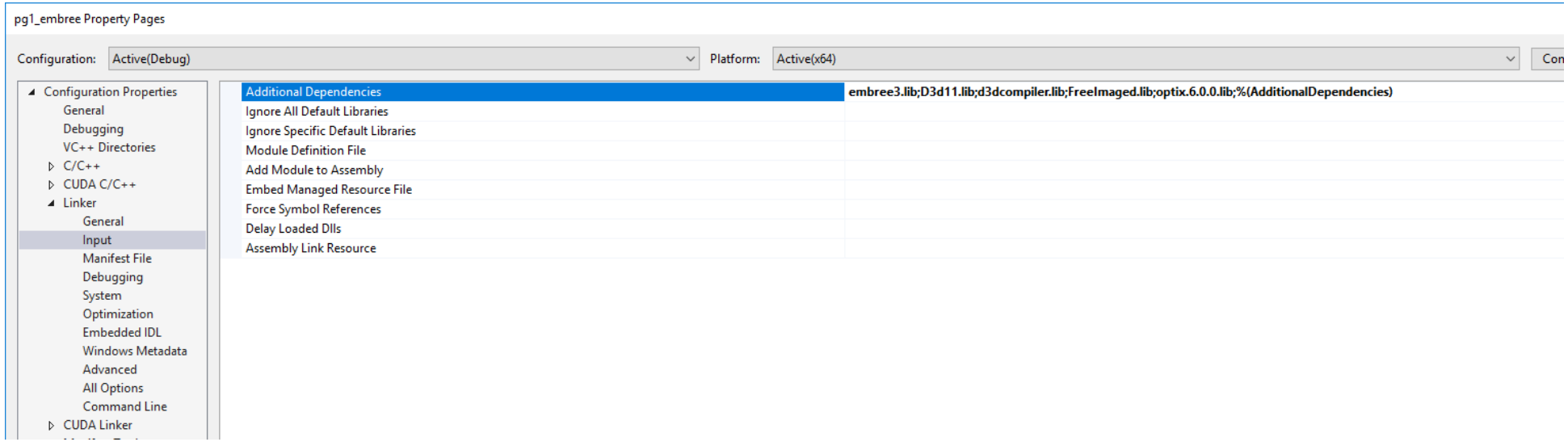
For Turing generation
compute_75, sm_75



VS 2017 and OptiX

- Unresolved external symbols?

Add optix.6.0.0.lib to Additional Dependencies



VS 2017 and OptiX

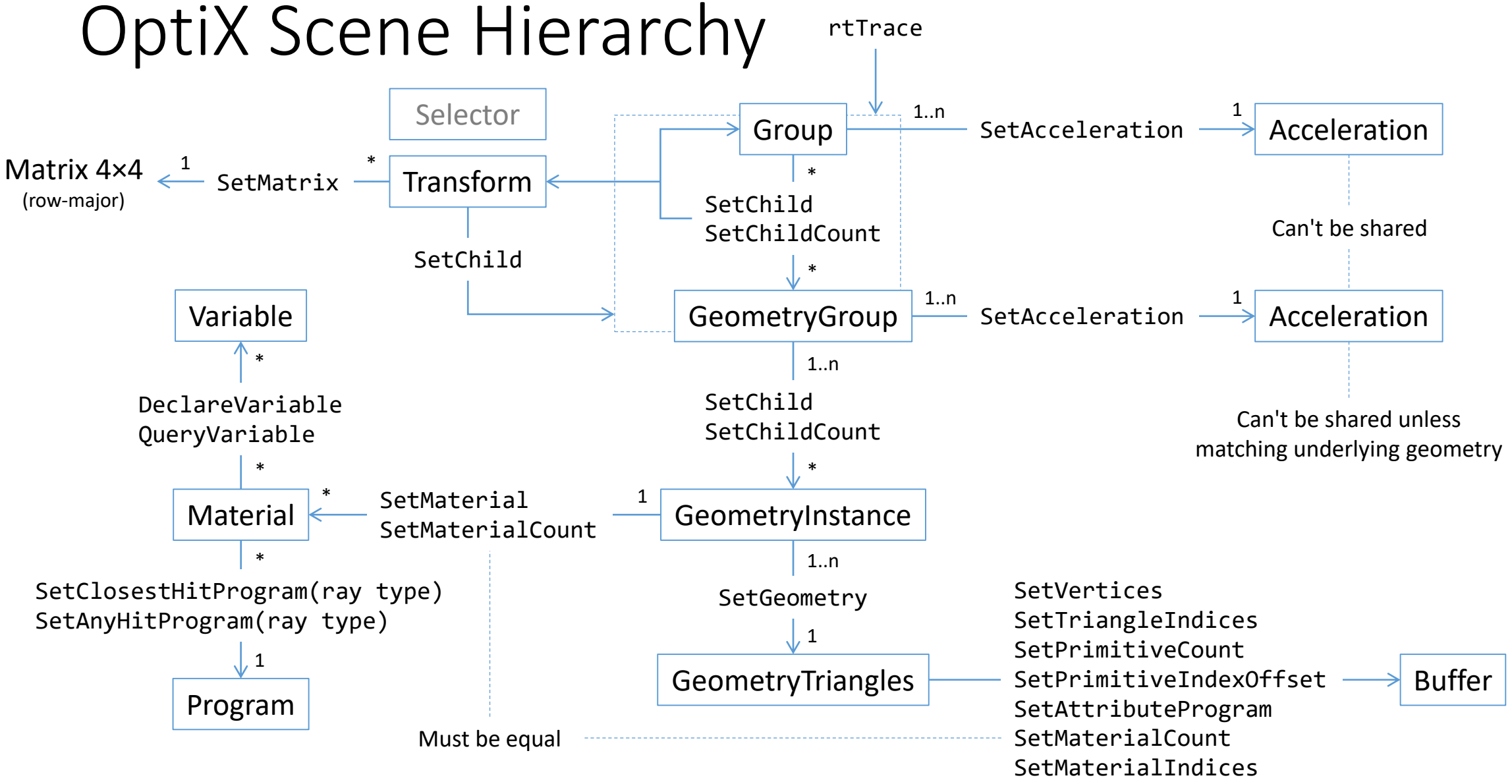
- Add following lines into the stdafx.h/pch.h file

```
#define NOMINMAX
// std libs go here
...
// Nvidia OptiX 6.0.0
#include <optix.h>
#include <optix_world.h>
```

- In tutorials.cpp

Use namespace qualifiers, e.g. optix::float3

OptiX Scene Hierarchy



Graph Nodes

- Ray tracing starts by calling the rtTrace function with the root of the graph as the parameter
- Graph is an assembly of various nodes provided by OptiX API
- Its bottom level contains geometric objects (e.g. triangles), upper levels consist of collections of objects (e.g. groups)
- It‘s not a scene graph in the classical sense
- It‘s a binding of different programs or actions to portions of the scene

HW Accelerated Triangles

- RTgeometrytriangles complements the RTgeometry type
- Applications should try to make use of the RTgeometrytriangles type whenever possible
- Provide hardware-accelerated primitive intersection (Turing arch.)
- Two types of triangles
  - indexed
  - unorganized collections (soap of triangles)

Materials

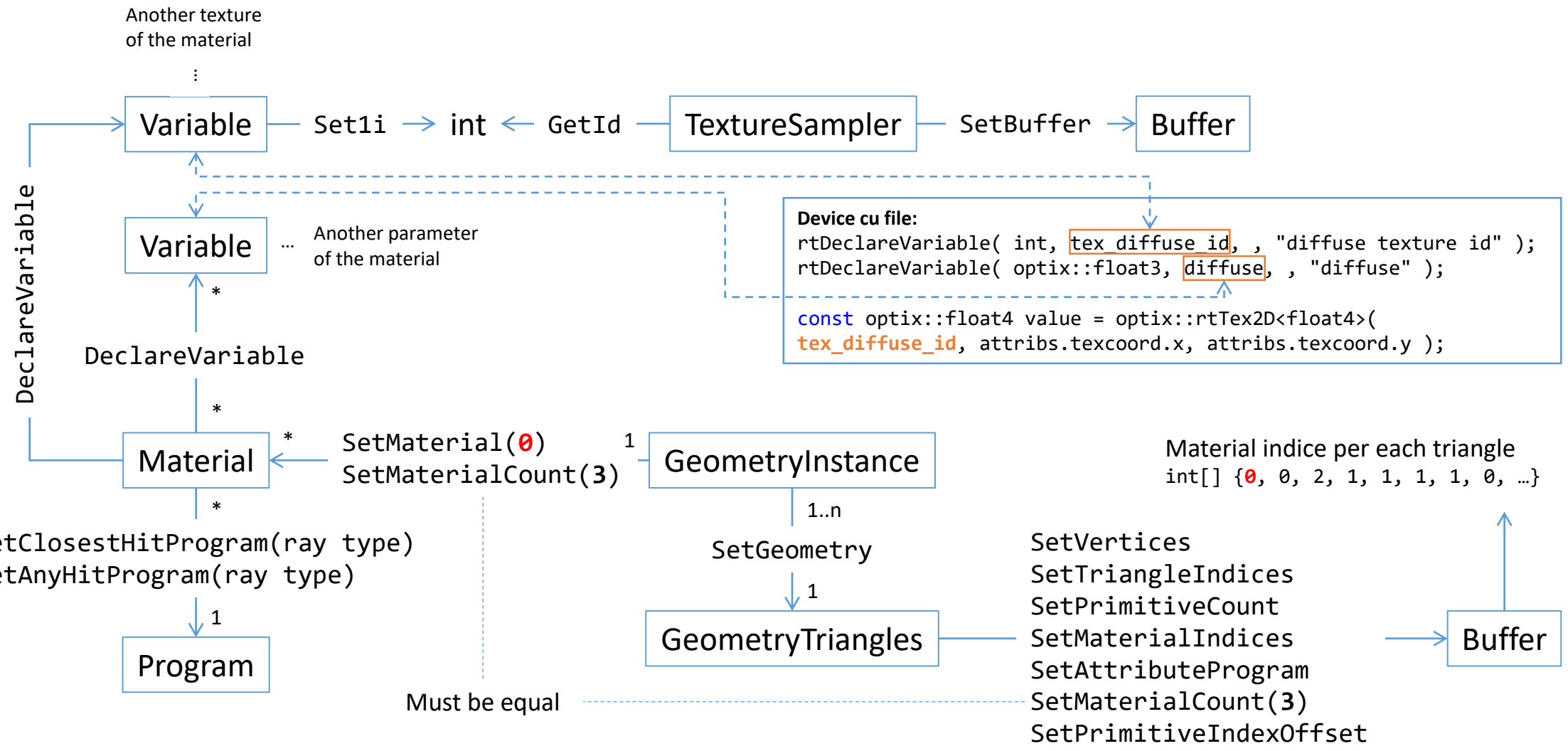
- Material encapsulates the actions that are taken when a ray intersects a primitive associated with a given material (e.g. computing a reflectance color, tracing additional rays, ignoring an intersection, and terminating a ray)
- Arbitrary parameters can be provided to materials by declaring program variables
- Two types of programs may be assigned to a material:
  - closest-hit programs
  - any-hit programs (executed for each potential closest intersection)
- Both types of programs are assigned to materials per ray type

Multi-materials

- Material slot is assigned to each triangle of a Rtgeometrytriangles
- SetMaterialCount of GeometryTriangles sets the number of materials
- Must be equal to the number of materials set at the GeometryInstance
- Mapping is set via function SetMaterialIndices of GeometryTriangles
- Actual materials are set at the GeometryInstance

Multi-materials

During traversal, the material with proper id is set in accordance with the material index associated with each triangle of geometry instance.



Main Communication Means in OptiX

- Variables
  - Similar to uniforms in other systems
  - Constants, system values, buffers, textures, ...
- Ray Payload
  - Arbitrary data associated with ray
  - In/out from rtTrace to Any-hit, Closest-hit, Miss
- Attributes
  - Arbitrary data associated with hit
  - Generated in Intersection program
  - Consumed by Any-hit, Closest-hit

Variables

- `rtDeclareVariable(type, name, semantic, annotation);`
  - `rtDeclareVariable(float3, eye, , ) = make_float3( 1.0f, 2.0f, 3.0f );`
  - `rtDeclareVariable(rtObject, top_object, , );`
  - `rtDeclareVariable(uint2, index, rtLaunchIndex, );`
- Variables communicate data to programs
- Well-defined set of scopes that will be queried for a definition of a variable
- String annotation may be interpreted by the host program as a human-readable description of the variable

Internal Semantic Variables

Semantic name	*Access*	*Description*	Ray generation	Exception	Closest hit	Any hit	Miss	Intersection	Bounding box
`rtLaunchIndex`	read only	The unique index identifying each thread launched by `rtContextLaunch`*n*`D`.	✓	✓	✓	✓	✓	✓	
`rtCurrentRay`	read only	The state of the current ray.			✓	✓	✓	✓	
`rtPayload`	read/write	The state of the current ray's payload of user-defined data.			✓	✓	✓		
`rtIntersectionDistance`	read only	The parametric distance from the current ray's origin to the closest intersection point yet discovered.			✓	✓		✓	
`rtSubframeIndex`	read only	The unique index identifying each subframe in a progressive launch. Zero for non-progressive launches.	✓	✓	✓	✓	✓	✓	

Attribute Variables

- Variables declared with user-defined semantics called attributes
- Attribute variables provide a mechanism for communicating data between the intersection program and the shading programs (e.g. surface normals and texture coordinates)

- rtDeclareVariable(float3, normal, attribute normal_vec,);

- Attributes for triangles with built-in intersection work differently, see Atribute Programs

Attribute Programs

- The attribute program is executed after the successful intersection of the ray with a triangle and before the execution of an any-hit and closest-hit program
- If no AP is defined, the triangle intersection barycentric coordinates are available

```
rtDeclareVariable( float2, barycentrics, attribute rtTriangleBarycentrics, );
```

Buffers

Host cpp file

```
RTvariable normals;
rtContextDeclareVariable( context, "normal_buffer", &normals );
RTbuffer normal_buffer;
rtBufferCreate( context, RT_BUFFER_INPUT, &normal_buffer );
rtBufferSetFormat( normal_buffer, RT_FORMAT_FLOAT3 );
rtBufferSetSize1D( normal_buffer, 3 );
{
  optix::float3 * data = nullptr;
  rtBufferMap( normal_buffer, ( void** )( &data ) );
  data[0].x = 1.0f; data[0].y = 0.0f; data[0].z = 0.0f;
  data[1].x = 0.0f; data[1].y = 1.0f; data[1].z = 0.0f;
  data[2].x = 0.0f; data[2].y = 0.0f; data[2].z = 1.0f;
  rtBufferUnmap( normal_buffer );
  data = nullptr;
}
rtBufferValidate( normal_buffer );
rtVariableSetObject( normals, normal_buffer );
```

Device cu file

```
rtBuffer<optix::float3> normal_buffer;
```

Triangle Attributes

Host cpp file

```
RTgeometrytriangles geometry_triangles;
…
RTprogram attribute_program;
rtProgramCreateFromPTXFile( context_, "optixtutorial.ptx", "attribute_program", &attribute_program );
rtProgramValidate( attribute_program );
rtGeometryTrianglesSetAttributeProgram( geometry_triangles, attribute_program );
rtGeometryTrianglesValidate( geometry_triangles );
```

Device cu file

```
struct TriangleAttributes
{
  optix::float3 normal;
  optix::float2 texcoord;
};

rtBuffer<optix::float3, 1> normal_buffer;
rtBuffer<optix::float2, 1> texcoord_buffer;
rtDeclareVariable( TriangleAttributes, attribs, attribute attributes, "Triangle attributes" );

RT_PROGRAM void attribute_program( void )
{
  const optix::float2 barycentrics = rtGetTriangleBarycentrics();
  const unsigned int index = rtGetPrimitiveIndex();
  const optix::float3 n0 = normal_buffer[index * 3 + 0];
  …
  attribs.normal = optix::normalize( n1 * barycentrics.x + n2 * barycentrics.y +
    n0 * ( 1.0f - barycentrics.x - barycentrics.y ) );
}
```

Multi-Materials

Host cpp file

```
RTgeometrytriangles geometry_triangles;

…

RTvariable material_indices;

rtContextDeclareVariable( context, "material_index_buffer",
&material_indices );

RTbuffer material_index_buffer;

rtBufferCreate( context, RT_BUFFER_INPUT, &material_index_buffer
);

rtBufferSetFormat( material_index_buffer, RT_FORMAT_UNSIGNED_BYTE
);

rtBufferSetSize1D( material_index_buffer, no_triangles ) ); // to
be used, must be larger than 1
```

Device cu file

```
rtBuffer<optix::uchar1> material_index_buffer;
```

Camera Parameters

Host cpp file

```
// set values of OptiX variables

RTvariable focal_length;

rtProgramDeclareVariable( primary_ray, "focal_length",
&focal_length );

rtVariableSet1f( focal_length, camera_.focal_length() );


// set view_from (use rtVariableSet3f)

…


// set M_c_w (use rtVariableSetMatrix3x3fv)

…
```

Device cu file

```
rtDeclareVariable( optix::float3, view_from, , );

rtDeclareVariable( optix::Matrix3x3, M_c_w, , "camera to world
space transformation matrix" );

rtDeclareVariable( float, focal_length, , "focal length in
pixels" );


RT_PROGRAM void primary_ray( void )

{

  const optix::float3 d_c = make_float3( launch_index.x -
output_buffer.size().x * 0.5f, output_buffer.size().y * 0.5f -
launch_index.y, -focal_length );

  const optix::float3 d_w = optix::normalize( M_c_w * d_c );

  optix::Ray ray( view_from, d_w, 0, 0.01f );

  …

}
```

Handling User Inputs in ImGui

- simpleguidx11.h

```
private:
  ImGuiIO io_;
```

- simpleguidx11.cpp

```
int SimpleGuiDX11::MainLoop() {
  …
  ImGui::NewFrame();
  io_ = ImGui::GetIO();
  Ui();
  …
}
```

- raytracer.cpp

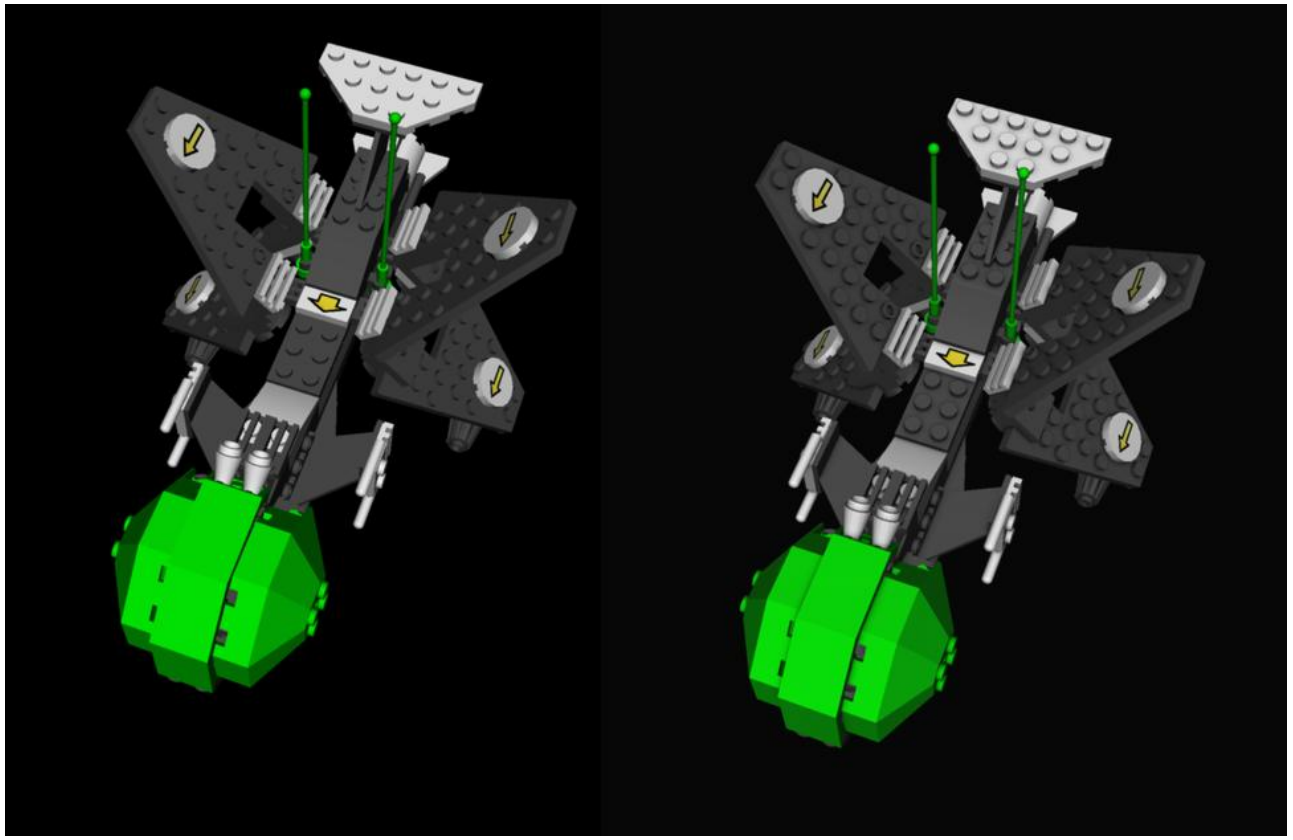
```
int Raytracer::Ui() {
…
const float dt = ImGui::GetIO().DeltaTime;
if ( io_.KeysDown[87] ) // W key
  camera_.MoveForward( dt );
if ( io_.KeysDown[83] ) // S key
  camera_.MoveForward( -dt );
…
camera_.Update();
…
}
```

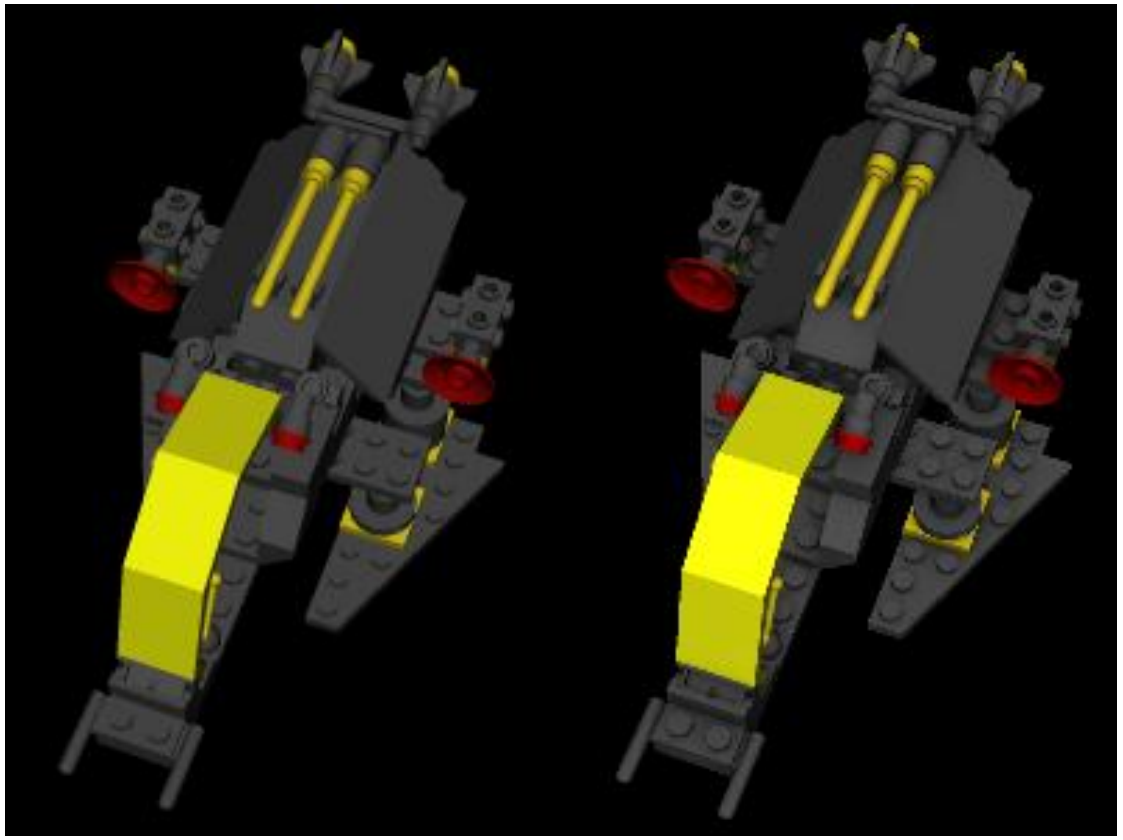
Scope for Device Functions

	Ray generation	Exception	Closest hit	Any hit	Miss	Intersection	Bounding box	Visit	Bindless callable program
`rtTransform*`			✓	✓	✓	✓	✓	✓	
`rtTrace`	✓		✓		✓				
`rtThrow`	✓		✓	✓	✓	✓	✓	✓	✓
`rtPrintf`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`rtTerminateRay`				✓					
`rtIgnoreIntersection`				✓					
`rtIntersectChild`								✓	
`rtPotentialIntersection`						✓			
`rtReportIntersection`						✓			
Callable program	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 8 – Scopes allowed for device-side API functions

Ambient Occlusion





Ambient Occlusion

- Real-time approximation based on pixel depth (OpenGL)
  - Screen space ambient occlusion (SSAO)
  - Horizon-based ambient occlusion (HBAO)
  - VolumetricAO, ambient obscurance, etc.
- True ambient occlusion

Remember:

```
V = 1
I = integrate( (V*(cos(t)))*sin(t), (t, 0, pi/2), (p, 0, 2*pi) )
print(I)

Return pi
```

$$A(x) = \frac{1}{\pi} \int_{\Omega} V(x, \omega_i) \, (\hat{\boldsymbol{n}} \cdot \omega_i) \mathrm{d}\omega_i$$

- $V(x, \omega_i)$ … visibility function at x, returns 0 if x is occluded in the direction ω_i and 1 otherwise

Pseudo (or Quasi) Random Numbers

Device cu file

```
#include <curand_kernel.h> // also add curand.lib to Project Properies -> Linker -> Input -> Additional Dependencies
struct Payload {
  …
  curandState_t state;
};
rtDeclareVariable( Payload, payload, rtPayload, "ray payload" );

RT_PROGRAM void primary_ray( void ) {
  Payload ;
  curand_init( launch_index.x + launch_dim.x * launch_index.y, 0, 0, &payload.state );
  …
}

float ksi = curand_uniform( &payload.state ); // anywhere in the code where the payload with the state of RNG is available
```

Performance Hints

- C:\Program Files\NVIDIA Corporation\NVSMI\nvidia-smi.exe dmon
- http://raytracing-docs.nvidia.com/optix_6_0/guide_6_0/index.html#performance#performance-guidelines

OpenGL Pipeline

List of all available shader types:
GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER, GL_COMPUTE_SHADER

Transforms 3D coordinates into different 3D coordinates and allows some basic processing on the vertex attributes

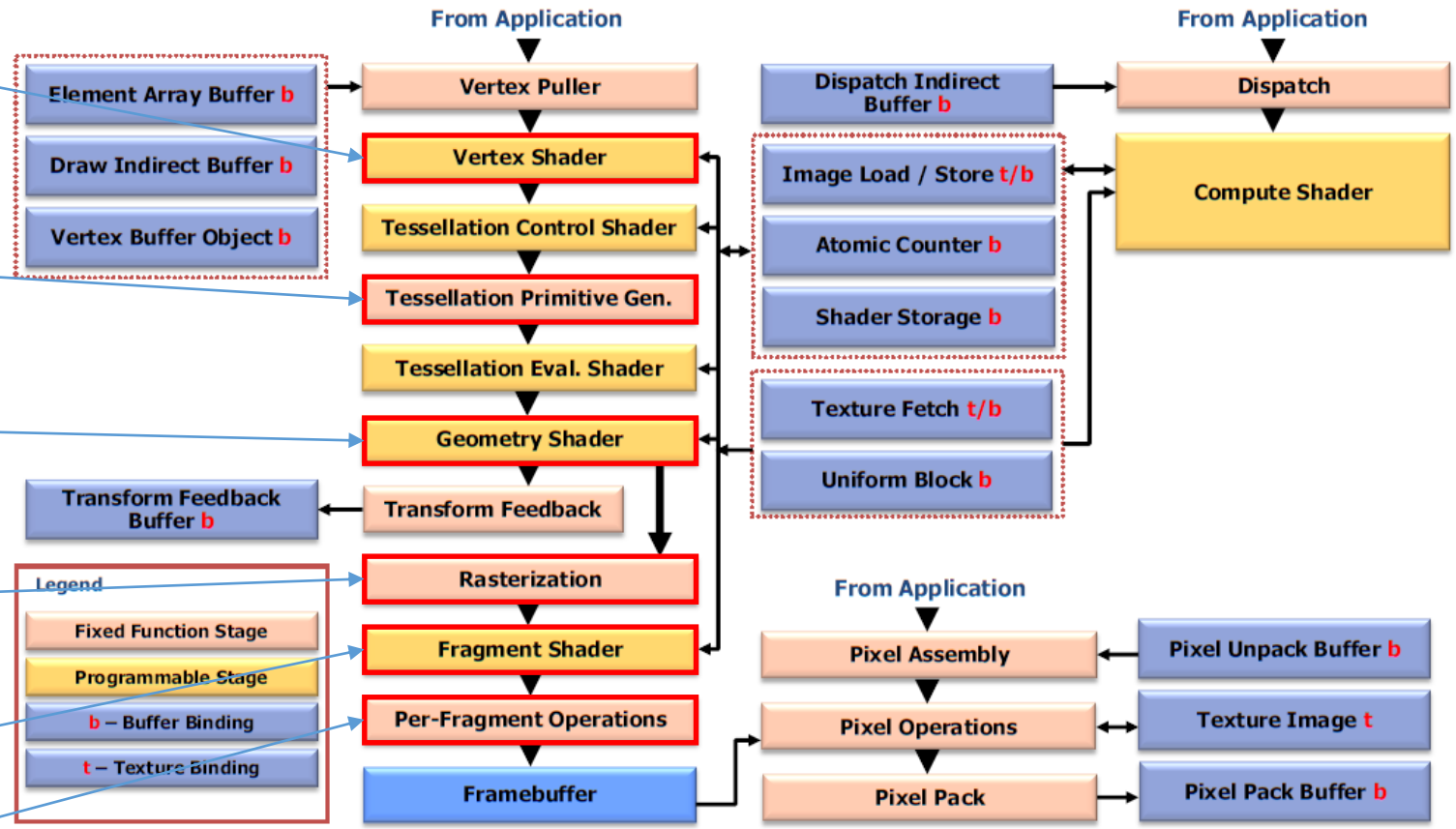
Assembles all the points in the primitive shape (e.g. point, line, triangle)

Emmits new vertices to generate new primitives

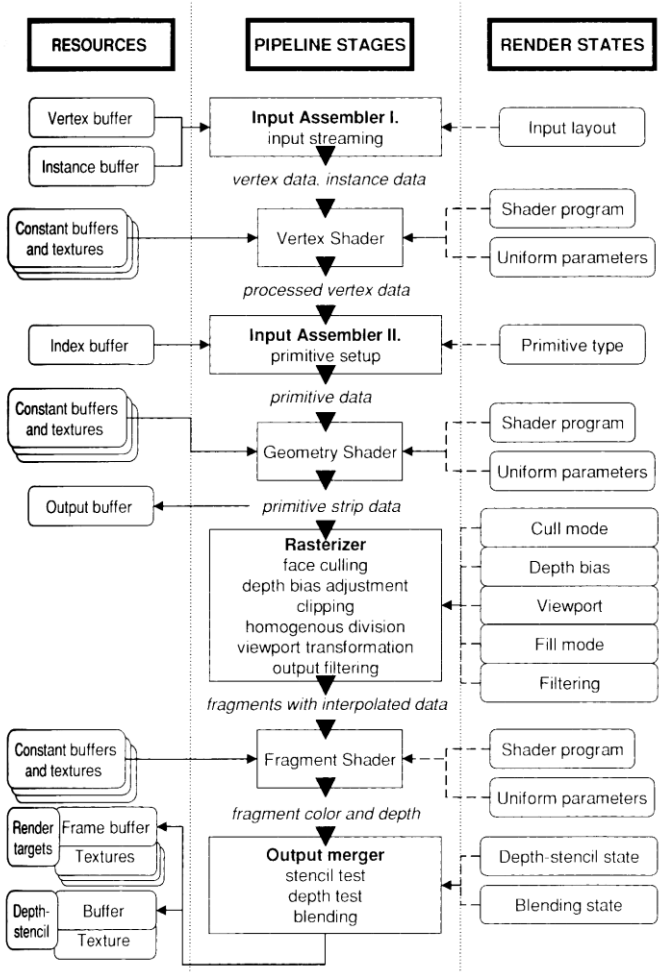
Maps the resulting primitives to the corresponding pixels and performs clipping

Calculate the final color of a pixel

Alpha test and blending stage



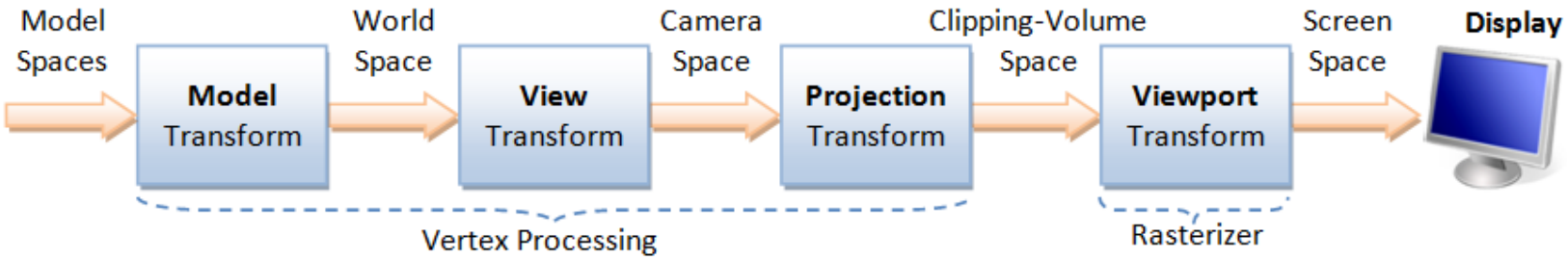
General Pipeline of Shader Model 4.0



Source: SZIRMAY-KALOS, László; SZÉCSI, László; SBERT, Mateu. GPU-based techniques for global illumination effects. *Synthesis Lectures on Computer Graphics and Animation*, 2008

Coordinates Transform Pipeline

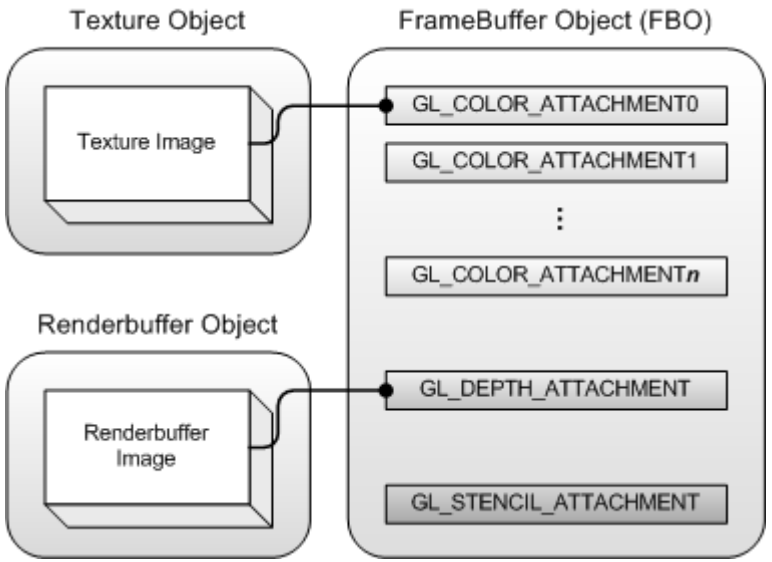
After this point, only the vertices/fragments inside the range <-1.0, 1.0> (NDC) on all three axis will be visible



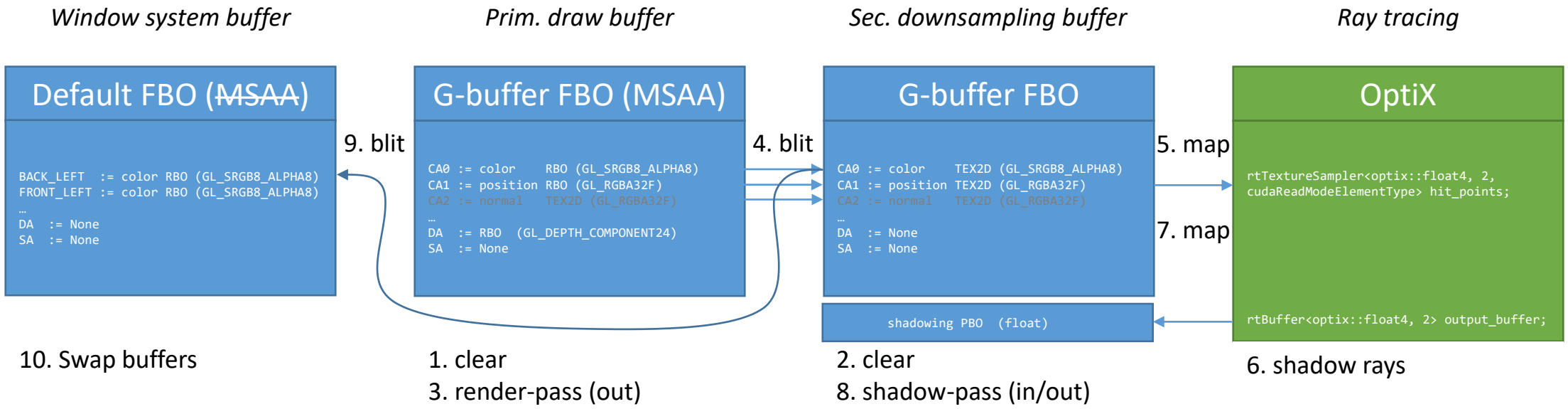
Framebuffer (Framebuffer Object, FBO)

- Create Framebuffer object (default framebuffer with renderbuffers may exist)
- Default framebuffer (typically) consists of two color buffers (GL_FRONT_LEFT, GL_BACK_LEFT; 3×8 bits), depth buffer (24 bits), and stencil buffer (8 bits); the type of elements is GL_UNSIGNED_NORMALIZED
- Attach Renderbuffers (optimized, MSAA) or Textures (post-pass shaders) as color, depth or stencil buffers
- Don‘t forget to check FBO status (glCheckFramebufferStatus)

Framebuffer



Framebuffer (Deferred Rendering)



Copy a rectangle of pixel values from one region of a **read framebuffer** to another region of a **draw framebuffer**

```
glBindFramebuffer( GL_READ_FRAMEBUFFER, fbo_src );
glReadBuffer( GL_COLOR_ATTACHMENTi );
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, fbo_dst );
glDrawBuffer( GL_COLOR_ATTACHMENTj );
glBlitFramebuffer( 0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST );
```

In case of `GL_DEPTH_BUFFER_BIT` or `GL_STENCIL_BUFFER_BIT` we don't need to call `glReadBuffer`

Custom Framebuffer

```
int msaa_samples = 0;
glGetIntegerv( GL_SAMPLES, &msaa_samples );

GLuint fbo, rbo_color, rbo_depth;
glGenFramebuffers( 1, &fbo );
glBindFramebuffer( GL_FRAMEBUFFER, fbo );

// color renderbuffer
glGenRenderbuffers( 1, &rbo_color );
glBindRenderbuffer( GL_RENDERBUFFER, rbo_color );
glRenderbufferStorage( GL_RENDERBUFFER, GL_RGBA8, width, height );
glRenderbufferStorageMultisample( GL_RENDERBUFFER, msaa_samples, GL_SRGB8_ALPHA8, width, height );
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER, rbo_color );

// depth renderbuffer
glGenRenderbuffers( 1, &rbo_depth );
glBindRenderbuffer( GL_RENDERBUFFER, rbo_depth );
glRenderbufferStorage( GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, width, height );
glRenderbufferStorageMultisample( GL_RENDERBUFFER, msaa_samples, GL_DEPTH_COMPONENT24, width, height );
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rbo_depth );

if ( glCheckFramebufferStatus( GL_FRAMEBUFFER ) != GL_FRAMEBUFFER_COMPLETE ) return -1;
```

Renderbuffer Object (RBO)

- Optimized for exclusive use with FBOs as render targets
- RBOs natively support multisampling (MSAA)
- Pixel transfer operations may be used to read and write from/to RBOs

Pixel Buffer Object (PBO)

- Buffer object used for asynchronous pixel transfers
- Similar to VBOs but use GL_PIXEL_PACK_BUFFER and GL_PIXEL_UNPACK_BUFFER targets

```
glGenBuffers( 1, &pbo );
glBindBuffer( GL_PIXEL_UNPACK_BUFFER, pbo );
const int pbo_size = width * height * pixel_size;
glBufferData( GL_PIXEL_UNPACK_BUFFER, pbo_size, 0, GL_STREAM_DRAW );
glBindBuffer( GL_PIXEL_UNPACK_BUFFER, 0 );
```

Performance hint

Allocates only a memory space of given size

Using Custom FBO in Render Loop

```
while ( !glfwWindowShouldClose( window ) ) // render loop
{
  // bind custom FBO and clear buffers
  glBindFramebuffer( GL_FRAMEBUFFER, fbo );
  glDrawBuffer( GL_COLOR_ATTACHMENT0 );
  glClearColor( 0.2f, 0.3f, 0.3f, 1.0f );
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

  // TODO activate shader program, setup uniforms and draw the scene

  // copy the color attachment from the custom FBO to the default FBO
  glBindFramebuffer( GL_READ_FRAMEBUFFER, fbo ); // bind custom FBO for reading
  glReadBuffer( GL_COLOR_ATTACHMENT0 ); // select it‘s first color buffer for reading
  glBindFramebuffer( GL_DRAW_FRAMEBUFFER, 0 ); // bind default FBO (0) for writing
  glDrawBuffer( GL_BACK_LEFT ); // select it‘s left back buffer for writing

  glBlitFramebuffer( 0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST );// copy
  //glBlitFramebuffer( 0, 0, width, height, 0, 0, width, height, GL_DEPTH_BUFFER_BIT, GL_NEAREST );

  // do the double buffering on the default FBO and process incomming events
  glfwSwapBuffers( window );
  glfwSwapInterval( 1 ); // introduce a 1-frame delay to prevent from tearing
  glfwPollEvents();

}
```

In case of two or more color buffers to be drawn into:

```
GLenum draw_buffers[2] = {
GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers( 2, draw_buffers );
```

We need some way how to specify which color attachment we want to use and the answer is glRead/DrawBuffer(s)

Resizing Custom FBO

```
static void Rasterizer::framebuffer_resize_callback( GLFWwindow * window, int width, int height ) {

  Rasterizer * rasterizer = ( Rasterizer * )glfwGetWindowUserPointer( window );
  rasterizer->Resize( width, height );

}

…

glfwSetWindowUserPointer( window_, this );
glfwSetFramebufferSizeCallback( window_, framebuffer_resize_callback );

…

int Rasterizer::Resize( const int width, const int height ) {

  glViewport( 0, 0, width, height );

  camera.Update( width, height );  // we need to update camera parameters as well

  // delete custom FBO with old width and height dimensions

  glBindFramebuffer( GL_FRAMEBUFFER, 0 );

  glDeleteRenderbuffers( 1, &rbo_color );
  glDeleteRenderbuffers( 1, &rbo_depth );
  glDeleteFramebuffers( 1, &fbo );

  InitFramebuffer(); // reinitialize custom FBO from scratch

}
```

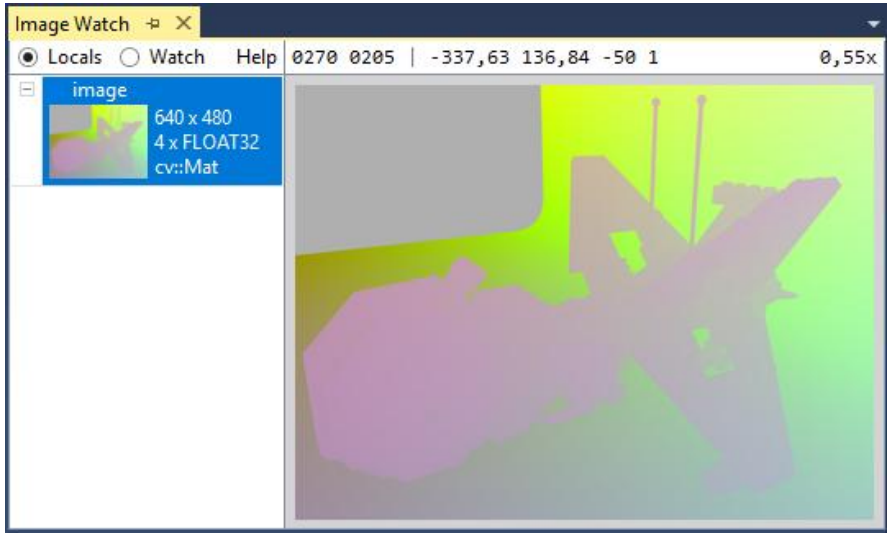
Textures as Color Attachments

```
glGenTextures( 1, &tex_positions );
glBindTexture( GL_TEXTURE_2D, tex_positions );
//glBindTexture( GL_TEXTURE_2D_MULTISAMPLE, renderedTexture );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB32F, width, height, 0, GL_RGB, GL_FLOAT, 0 );


glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );

glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, tex_positions, 0 );
//glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D_MULTISAMPLE, tex_positions, 0 );
```

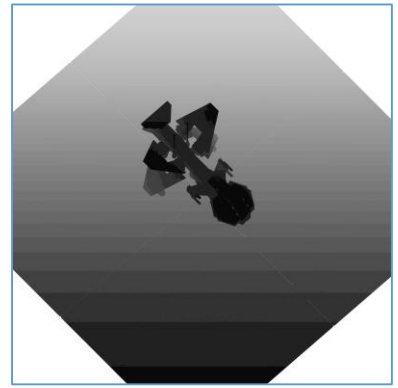
Don't forget to recreate the textures with proper width and height when resizing the FBO

Framebuffer Debug

```
cv::Mat image = cv::Mat::zeros( cv::Size( width, height ), CV_32FC4 );
glBindFramebuffer( GL_READ_FRAMEBUFFER, fbo ); // source fbo
glReadBuffer( GL_COLOR_ATTACHMENT1 ); // source color buffer
glReadPixels( 0, 0, width, height, GL_RGBA, GL_FLOAT, image.data );
cv::flip( image, image, 0 ); // in case of GL_LOWER_RIGHT
```



Framebuffer Debug (Depth Component)

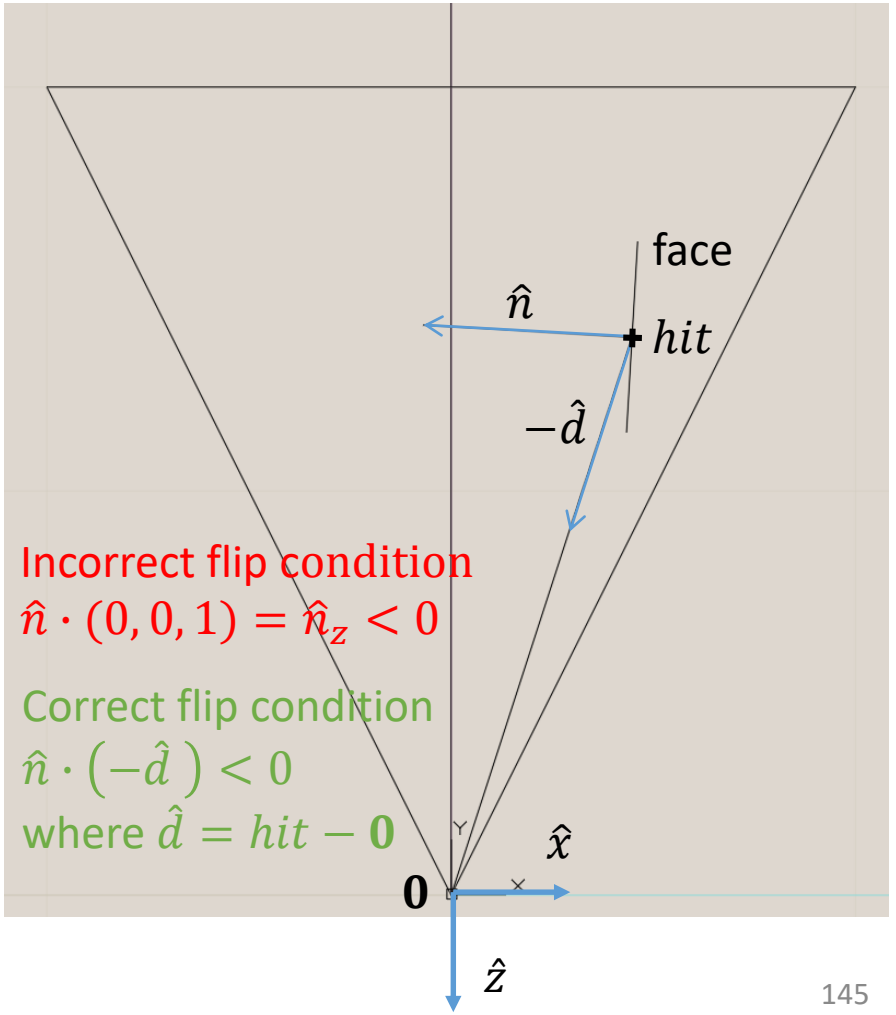


```
float * db = new float[shadow_width * shadow_height];
glReadPixels( 0, 0, shadow_width, shadow_height, GL_DEPTH_COMPONENT, GL_FLOAT, db );

FILE * file = fopen( "depth.ppm", "wt" );
fprintf( file, "P3\n%d %d\n255\n", shadow_width, shadow_height );
for ( int y = 0; y < shadow_height; ++y ) {
  for ( int x = 0; x < shadow_width; ++x ) {
    const int value = int( 255.0f * powf( db[y * shadow_width + x], 5.0f ) ); // value 5 is arbitrary
    fprintf( file, "%d %d %d\t", value, value, value );
  }
  fprintf( file, "\n" );
}
fclose( file );
delete[] db;
```

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 );
```

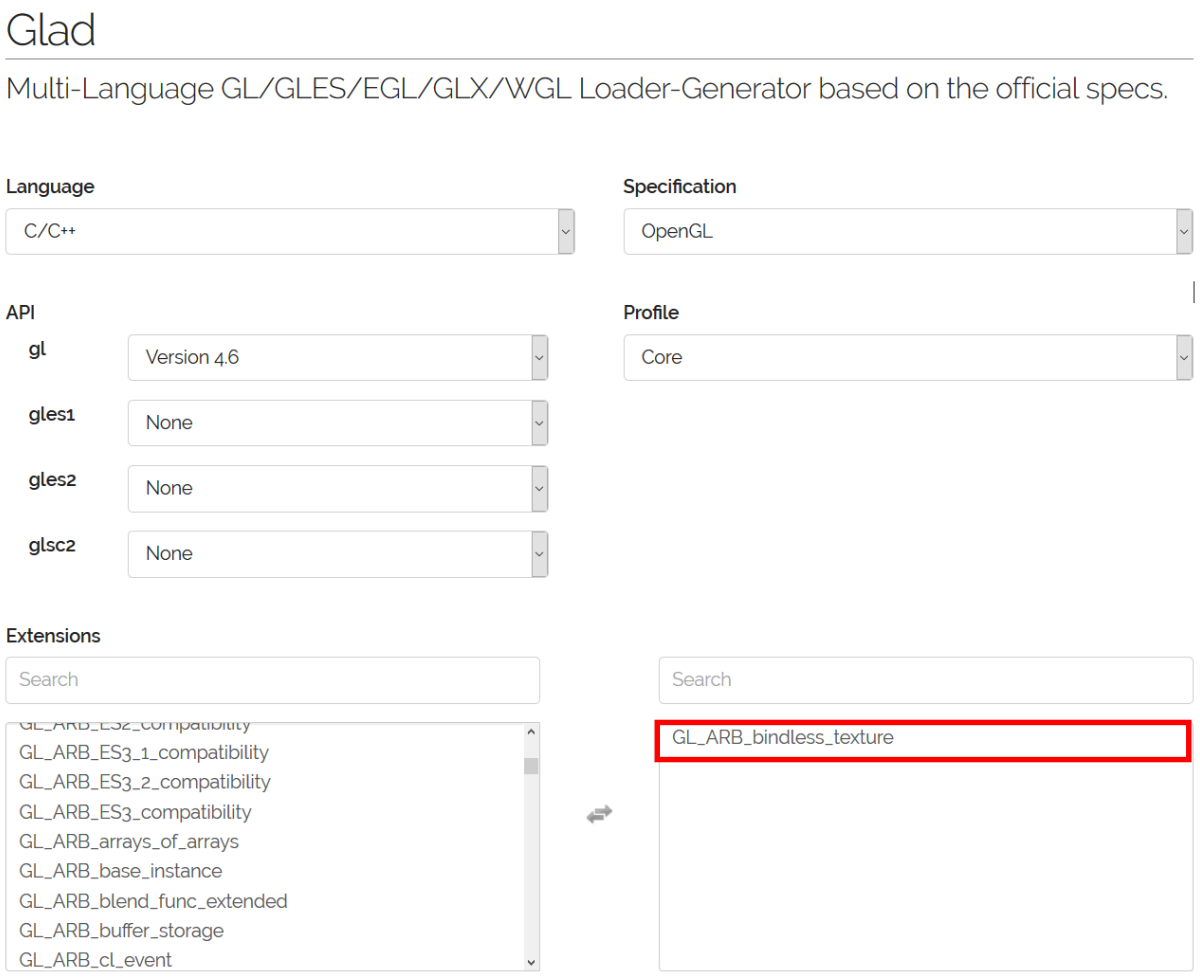
```
#pragma pack(push, 1)
struct Vertex
{
  Vector3 position;
  Vector3 normal;
  Vector3 color;
  Coord2f texture_coords;
  Vector3 tangent;
  int material_index{ 0 };
  char pad[4]; // fill up to 64 B
};
#pragma pack (pop)
```

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)

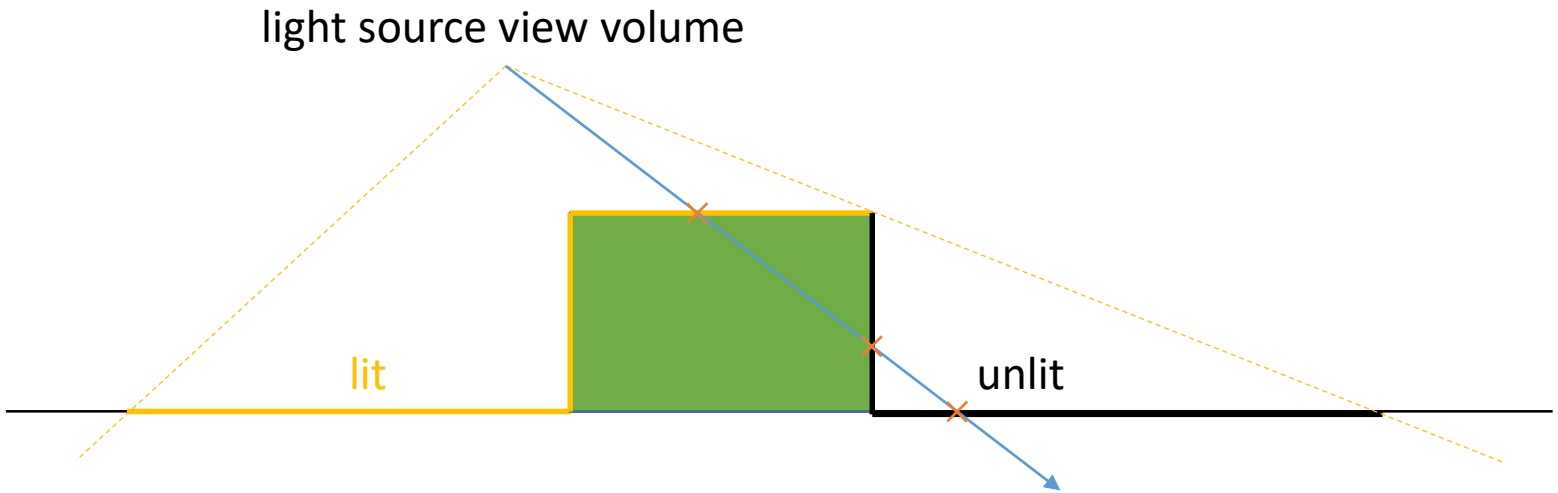
Primitive Type of Polygon *i*	First Vertex Convention	Last Vertex Convention
point	*i*	*i*
independent line	2*i* - 1	2*i*
line loop	*i*	*i* + 1, if *i* < *n* 1, if *i* = *n*
line strip	*i*	*i* + 1
independent triangle	3*i* - 2	3*i*
triangle strip	*i*	*i* + 2
triangle fan	*i* + 1	*i* + 2
line adjacency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

Shadow Mapping

- Shadows are important for realism as they create a sense of depth
- Two common real-time techniques: Shadow Volumes and **Shadow Mapping**
- Unfortunately, a perfect shadow algorithm doesn't exist yet and some artifacts are unavoidable
- SM can handle spot lights, directional lights, and omni lights with cube mapping
- SM is quite simple to implement (but not that simple like shadow rays in RT)
- In terms of implementation, SM consists of two passes
  - First pass – depth (or shadow) map is generated from the light's perspective
  - Second pass – the scene is rendered with shadow mapping (i.e. shadowing of every fragment is determined based on the depth map from the first pass)

Shadow Mapping

- The basic idea of the shadow mapping is similar to shadow rays in RT
- If we render the scene from the perspective of the light source then everything we see from that light is lit and everything else is in shadow



- More elaborated scheme of the shadow mapping algorithm is on the slide with the second pass description

Shadow Mapping

- Before we can start, we have to prepare our Rasterizer class a bit

```
Class Rasterizer {
…
public:
  int InitShadowDepthbuffer(); // initialize shadow (depth) map texture and framebuffer for the first pass

private:
  int shadow_width_{ 1024 }; // shadow map resolution
  int shadow_height_{ shadow_width_ };
  GLuint fbo_shadow_map_{ 0 }; // shadow mapping FBO
  GLuint tex_shadow_map_{ 0 }; // shadow map texture
  GLuint shadow_program_{ 0 }; // collection of shadow mapping shaders
};
```

Shadow Mapping

```
// this is only for depth map debug
GLuint rbo_color;
glGenRenderbuffers( 1, &rbo_color );
glBindRenderbuffer( GL_RENDERBUFFER, rbo_color );
glRenderbufferStorage( GL_RENDERBUFFER, GL_RGBA8,
 shadow_width_, shadow_height_ );
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
  GL_RENDERBUFFER, rbo_color );
```

- Init shadow map texture as a depth component of a new framebuffer

rasterizer.cpp

```
void Rasterizer::InitShadowDepthbuffer() // must be called before we enter the main render loop
{
  glGenTextures( 1, &tex_shadow_map_ ); // texture to hold the depth values from the light's perspective
  glBindTexture( GL_TEXTURE_2D, tex_shadow_map_ );

  // GL_DEPTH_COMPONENT ... each element is a single depth value. The GL converts it to floating point, multiplies by the signed scale
  // factor GL_DEPTH_SCALE (1), adds the signed bias GL_DEPTH_BIAS (0), and clamps to the range [0, 1] - this will be important later

  glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, shadow_width_, shadow_height_, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0 );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER );
  const float color[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // areas outside the light's frustum will be lit
  glTexParameterfv( GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, color );
  glBindTexture( GL_TEXTURE_2D, 0 );

  glGenFramebuffers( 1, &fbo_shadow_map_ ); // new frame buffer
  glBindFramebuffer( GL_FRAMEBUFFER, fbo_shadow_map_ );
  glFramebufferTexture2D( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, tex_shadow_map_, 0 ); // attach the texture as depth
  glDrawBuffer( GL_NONE ); // we don't need any color buffer during the first pass
  glBindFramebuffer( GL_FRAMEBUFFER, 0 ); // bind the default framebuffer back
}
```

Shadow Mapping

First pass (shaders)

Vertex shader

```
#version 460 core
// vertex attributes
layout ( location = 0 ) in vec4 in_position_ms;

// uniform variables
// Projection (P_l)*Light (V_l)*Model (M) matrix
uniform mat4 mlp;

void main( void )
{
  gl_Position = mlp * in_position_ms;
}
```

Fragment shader

```
#version 460 core

void main( void )
{
  // by default the z component of gl_FragCoord is assigned
to gl_FragDepth so we don't have to do anything here
  //gl_FragDepth = gl_FragCoord.z;

  // depth map debug
  gl_FragColor = vec4(vec3(gl_FragCoord.z), 1.0f);
}
```

```
// save shadow map after depth pass completion
Texture4f shadow_map( shadow_width_, shadow_height_ );
glReadBuffer( GL_COLOR_ATTACHMENT0 );
glReadPixels( 0, 0, shadow_width_, shadow_height_ ,
  GL_BGRA, GL_FLOAT, shadow_map.data() );
shadow_map.Save("shadow_map.exr");
```

Shadow Mapping

First pass (rendering loop)

rasterizer.cpp

```
// --- first pass ---
// set the shadow shader program and the viewport to match the size of the depth map
glUseProgram( shadow_program_ );
glViewport( 0, 0, shadow_width_, shadow_height_ );
glBindFramebuffer( GL_FRAMEBUFFER, fbo_shadow_map_ );
glClear( GL_DEPTH_BUFFER_BIT );

// set up the light source through the MLP matrix
Matrix4x4 mlp = BuildMLPMatrix();
SetMatrix4x4( shadow_program_, mlp.data(), "mlp" );

// draw the scene
glBindVertexArray( vao_ );
glDrawArrays( GL_TRIANGLES, 0, no_triangles_ * 3 );
glBindVertexArray( 0 );

// set back the main shader program and the viewport
glBindFramebuffer( GL_FRAMEBUFFER, 0 );
glViewport( 0, 0, camera_.width(), camera_.height() );
glUseProgram( shader_program_ );
```

Shadow Mapping

Second pass

Coordinates of point $\boldsymbol{a}$ in light's clip space
$\boldsymbol{a}_{lcs} = (P_l V_l M)\boldsymbol{a}_{ms}$

Texture coordinates of point $\boldsymbol{a}$ in depth map
$\boldsymbol{a}_{ts} = (\boldsymbol{a}_{lcs}.xy + 1)/2$

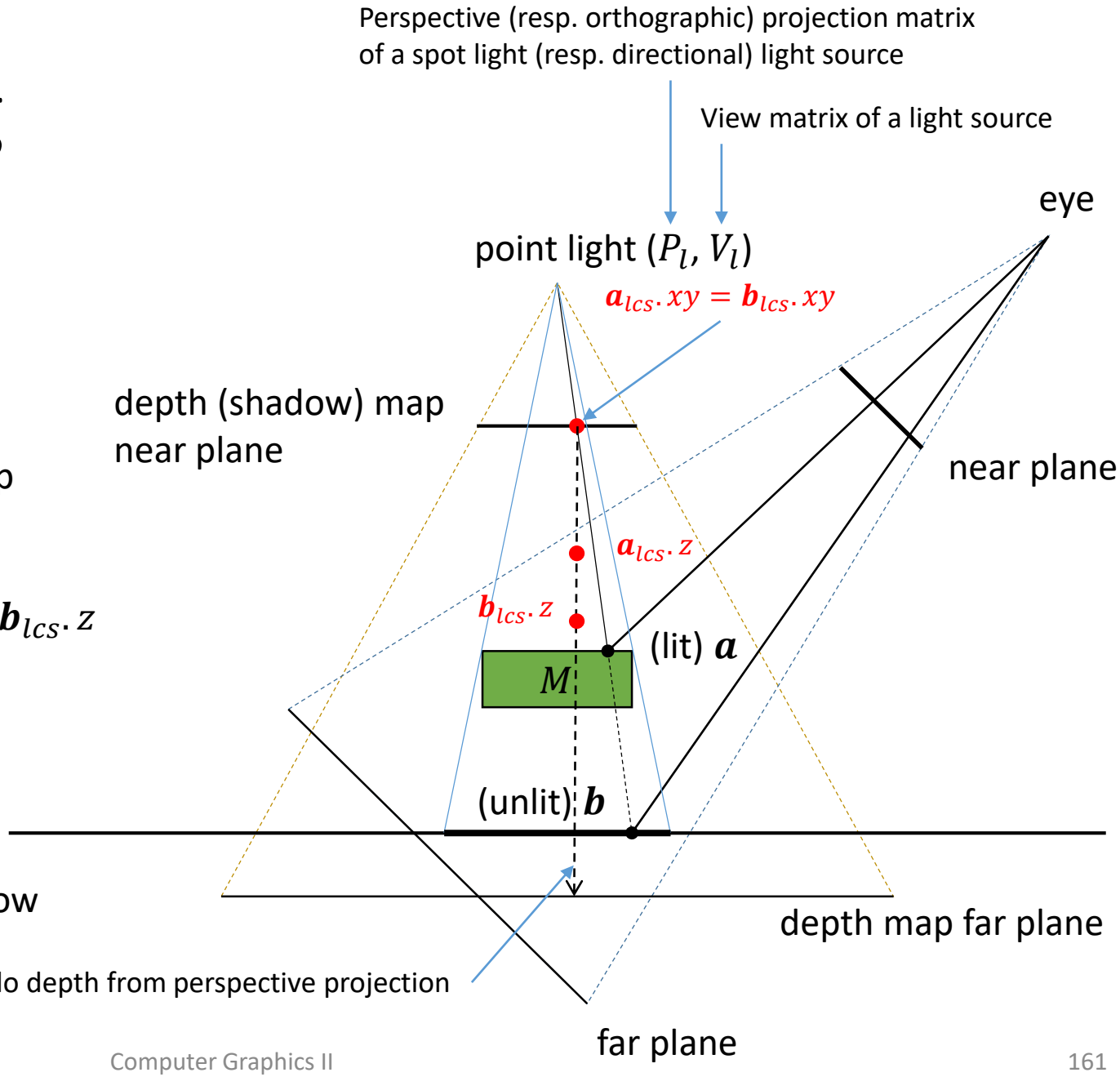
Note that $\boldsymbol{a}_{lcs}.xy = \boldsymbol{b}_{lcs}.xy$ and $\boldsymbol{a}_{lcs}.z < \boldsymbol{b}_{lcs}.z$

shadow_map$(\boldsymbol{a}_{ts}) \approx \boldsymbol{a}_{lcs}.z \rightarrow \boldsymbol{a}$ is lit

here is the catch aka acne artefacts

In case of point $\boldsymbol{b}$
shadow_map$(\boldsymbol{b}_{ts}) < \boldsymbol{b}_{lcs}.z \rightarrow \boldsymbol{b}$ is in shadow

lcs … light clip space
ts … texture space



Shadow Mapping

Second pass (shaders; only new lines are shown)

Vertex shader

```
#version 460 core
// vertex attributes
layout ( location = 0 ) in vec4 in_position_ms;
…

// uniform variables
// Projection (P_l)*Light (V_l)*Model (M) matrix
uniform mat4 mlp;

// output variables
out vec3 position_lcs; // this is our point a (or b) in lcs

void main( void )
{
  …
  tmp = mlp * vec4( in_position_ms.xyz, 1.0f );
  position_lcs = tmp.xyz / tmp.w;
  …
}
```

Fragment shader

```
#version 460 core
…
// inputs from previous stage
in vec3 position_lcs;

uniform sampler2D shadow_map; // light's shadow map

void main( void )
{
  …
  // convert LCS's range <-1, 1> into TC's range <0, 1>
  vec2 a_tc = ( position_lcs.xy + vec2( 1.0f ) ) * 0.5f;
  float depth = texture( shadow_map, a_tc ).r;
  // (pseudo)depth was rescaled from <-1, 1> to <0, 1>
  depth = depth * 2.0f - 1.0f; // we need to do the inverse
  float shadow = ( depth + bias >= position_lcs.z )? 1.0f :
0.25f ); // 0.25f represents the amount of shadowing
  …
  vec3 L_o = ( k_D * L_r_D + ( k_S * brdf_integr.x +
brdf_integr.y ) * L_r_S ) * ao * shadow;
  …
}
```

Shadow Mapping

Second pass (rendering loop)

rasterizer.cpp

```
// --- second pass ---
…
// everything is the same except this line
SetMatrix4x4( shader_program_, mlp.data(), "mlp" );
// and also don't forget to set the sampler of the shadow map before entering rendering loop
glActiveTexture( GL_TEXTURE3 );
glBindTexture( GL_TEXTURE_2D, tex_shadow_map_ );
SetSampler( shader_program_, 3, "shadow_map" );
…
```

Fragment shader

```
…
uniform sampler2D shadow_map;
…
```

glutils.cpp

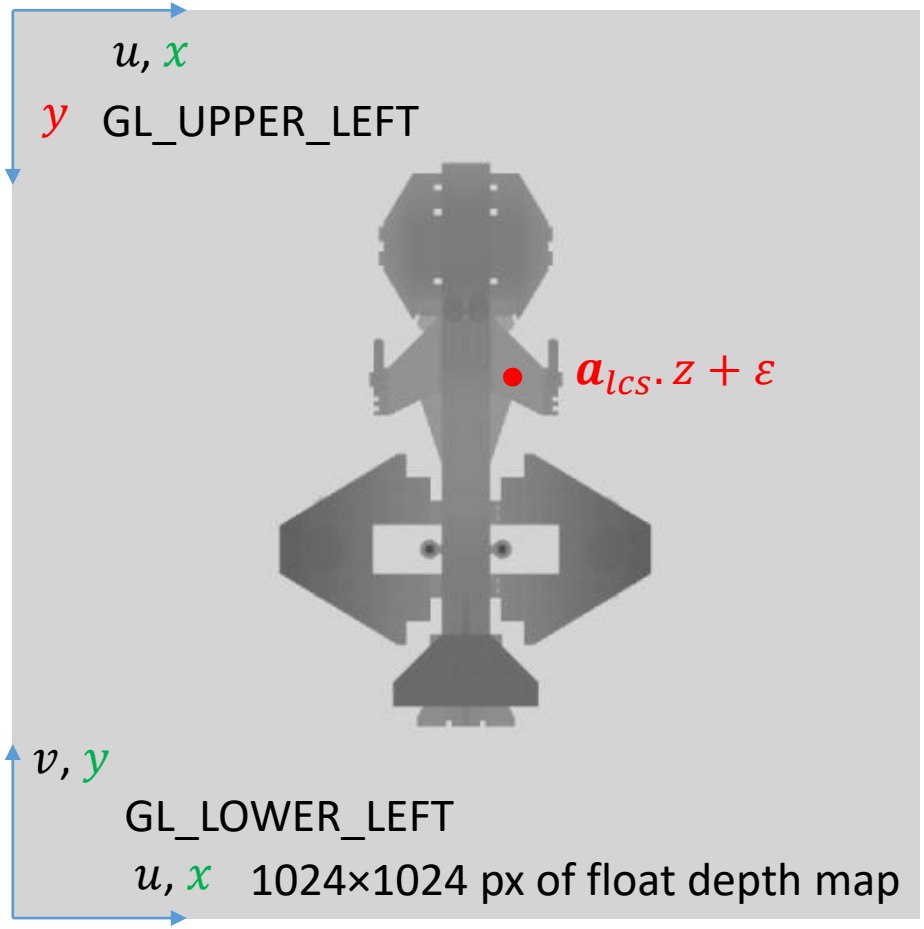
```
void SetSampler( const GLuint program, GLenum texture_unit, const char * sampler_name ) {
  const GLint location = glGetUniformLocation( program, sampler_name );

  if ( location == -1 ) {
    printf( "Texture sampler '%s' not found in active shader.\n", sampler_name );
  } else {
    glUniform1i( location, texture_unit );
  }
}
```

Shadow Mapping

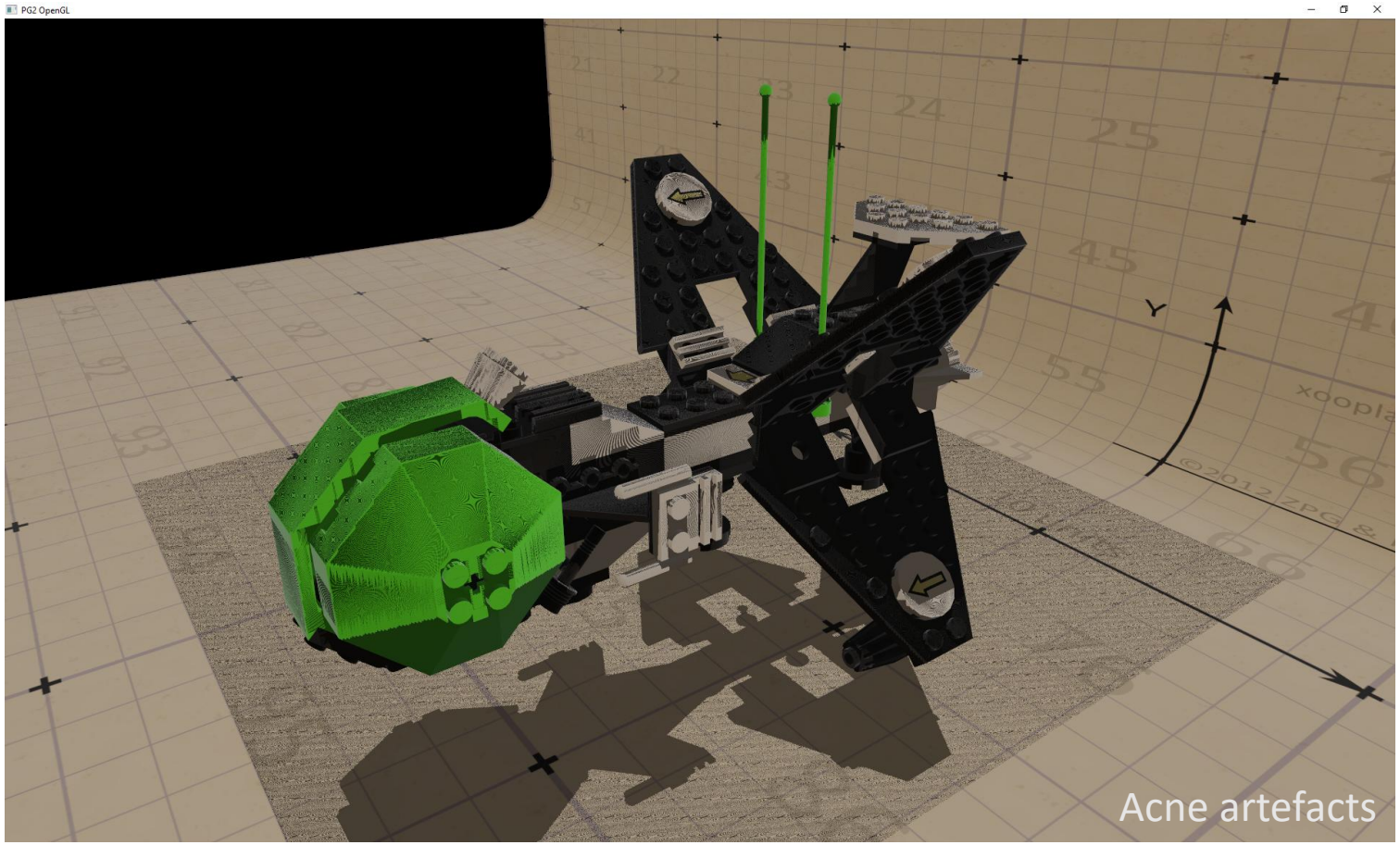
- Depth (shadow) map of a spot light consists of (pseudo)depths of scene points as viewed from the perspective of lights viewing frustum
- Even for float depth maps holds that we introduce some small error (ε) which will exhibit as randomly scattered lit/unlit patches (acne artefact)
- This can be partially fixed by introducing a small bias such that the fragments are not incorrectly considered below the surface

  shadow_map$(\boldsymbol{a}_{ts}) + bias \geq \boldsymbol{a}_{lcs}.z \rightarrow \boldsymbol{a}$ is lit
- And how much is the bias? Well, it depends on many factors, try 0.001. Also note that the resolution of the depth map matters.

Note that the NDC of the viewport does not match texture coordinates of the attached depth component if GL_UPPER_LEFT clip control origin is used thus use GL_LOWER_LEFT instead



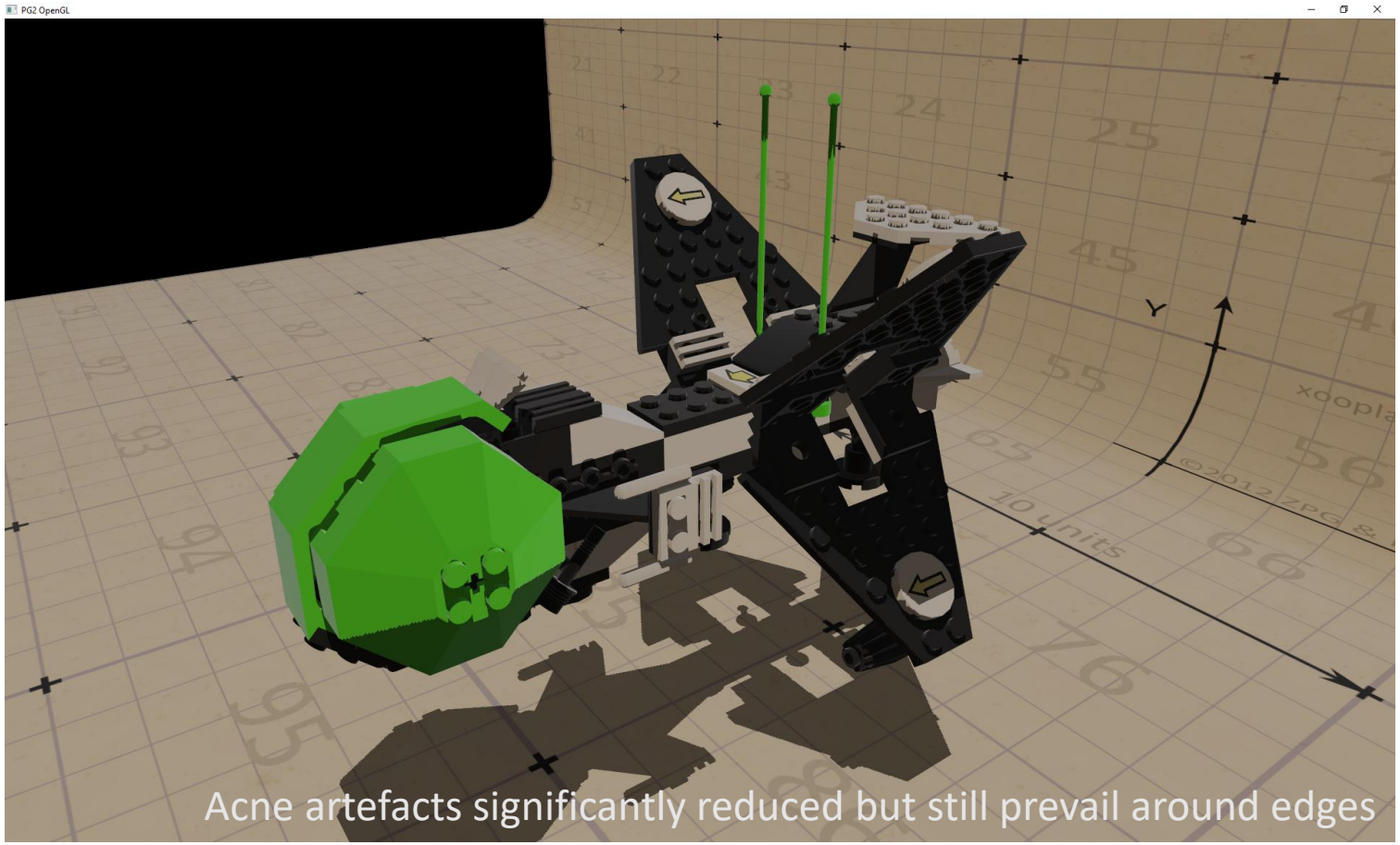
Shadow Mapping



Shadow map
1024×1024 px

No bias
No PCF

Shadow Mapping



Shadow map
1024×1024 px

Bias 0.001
No PCF

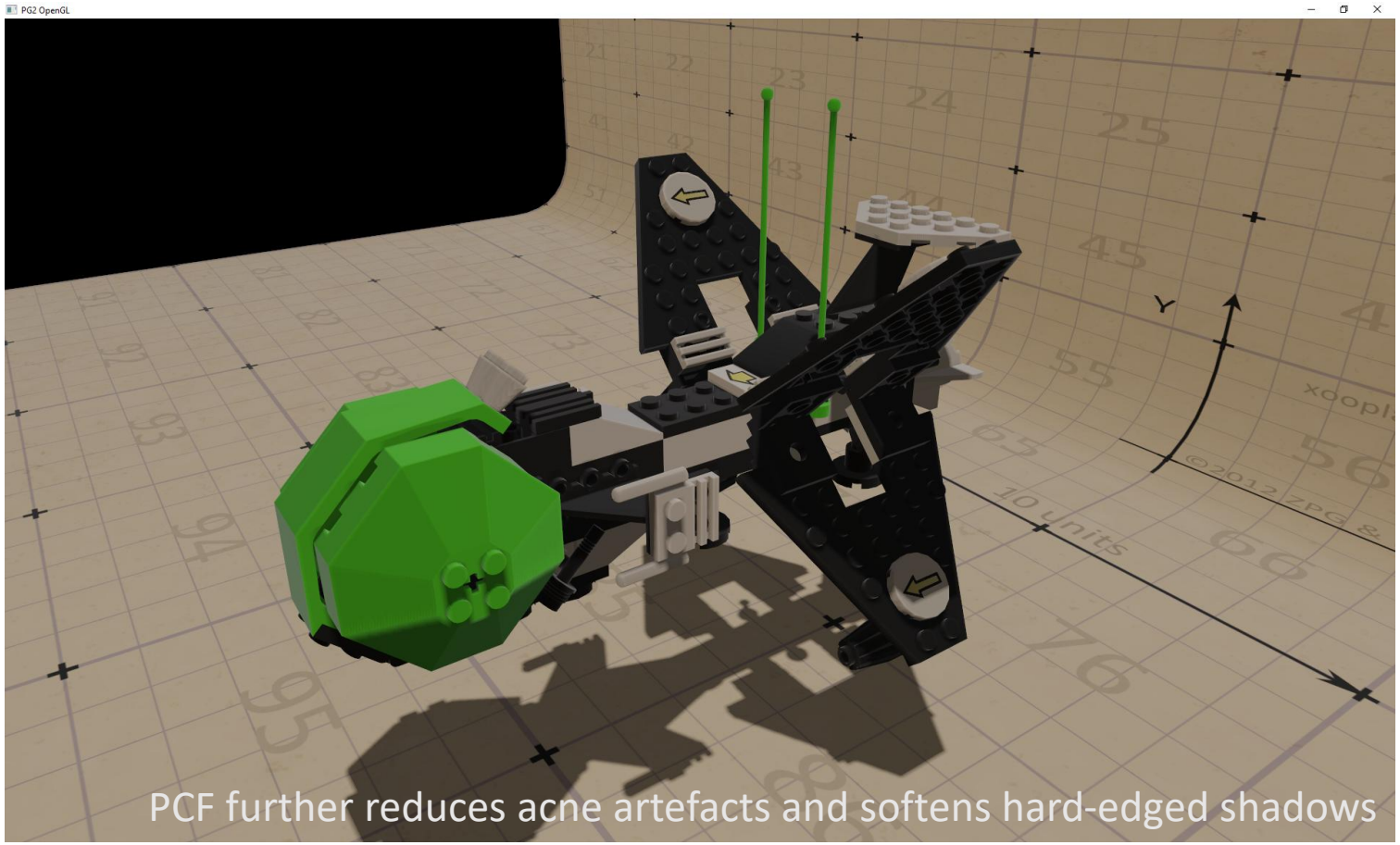
Shadow Mapping

- Another option how to (partially) tackle with artefacts is to involve some sort of shadow map filtering, e.g. percentage-closer filtering (PCF)

Fragment shader

```
…
vec2 shadow_texel_size = 1.0f / textureSize( shadow_map, 0 ); // size of a single texel in tex coords
const int r = 2; // search radius, try different values
float shadow = 0.0f; // cumulative shadowing coefficient
for ( int y = -r; y <= r; ++y ) {
  for ( int x = -r; x <= r; ++x ) {
    vec2 a_tc = ( position_lcs.xy + vec2( 1.0f ) ) * 0.5f;
    a_tc += vec2( x, y ) * shadow_texel_size;
    float depth = texture( shadow_map, a_tc ).r;
    depth = depth * 2.0f - 1.0f;
    shadow += ( depth + bias >= position_lcs.z )? 1.0f : 0.25f );
  }
}
shadow *= ( 1.0f / ( ( 2 * r + 1 ) * ( 2 * r + 1 ) ) ); // compute mean shadowing value
// use the shadow value as in the case of non-pcf code
…
```

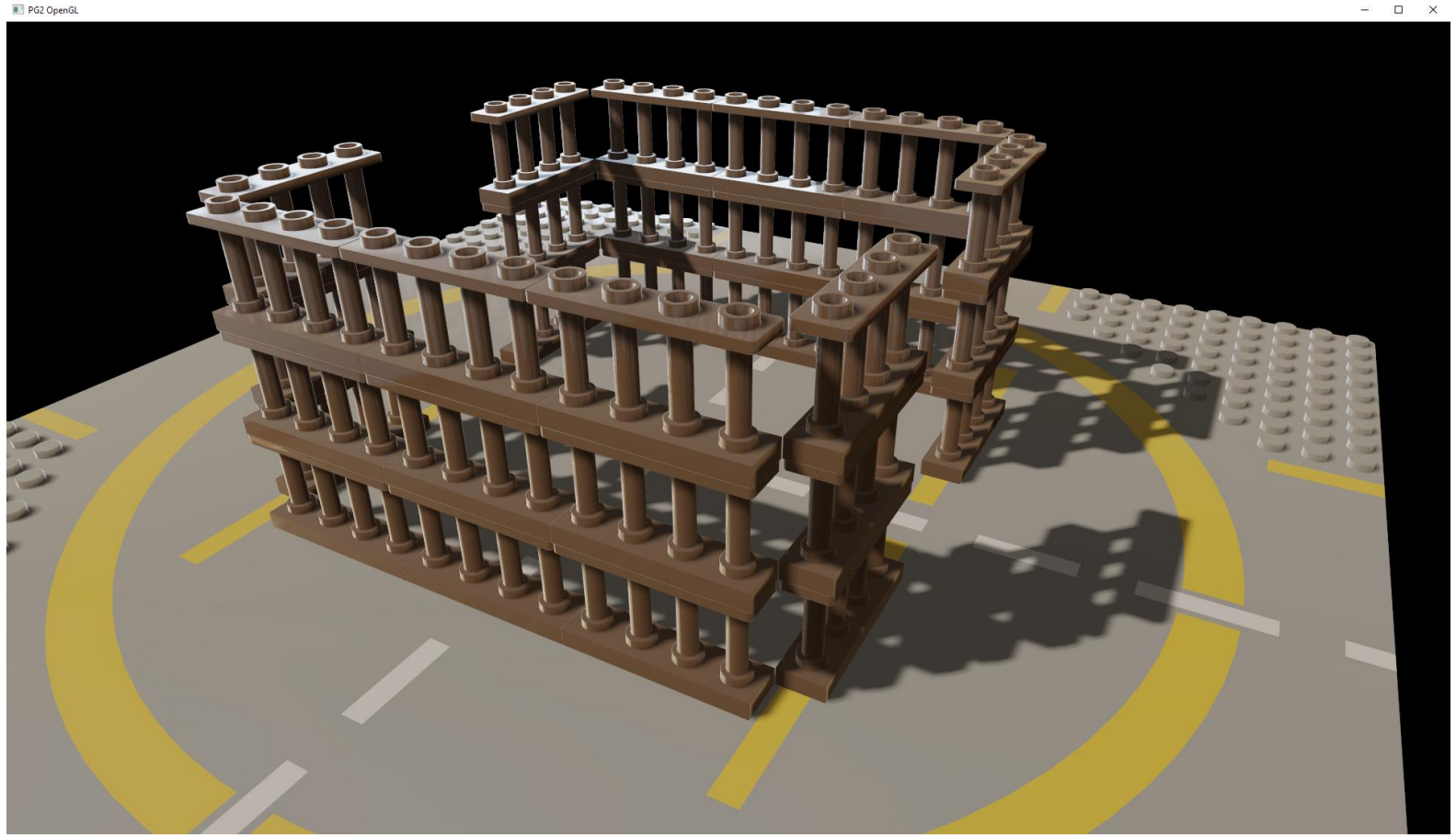
Shadow Mapping



Shadow map
1024×1024 px

Bias 0.001
PCF 3×3

Shadow Mapping



Shadow map
4096×4096 px

Directional light
(orthogonal projection)

Near = 20
Far = 250

Shadow diameter = 50

Bias 0.001
PCF 5×5

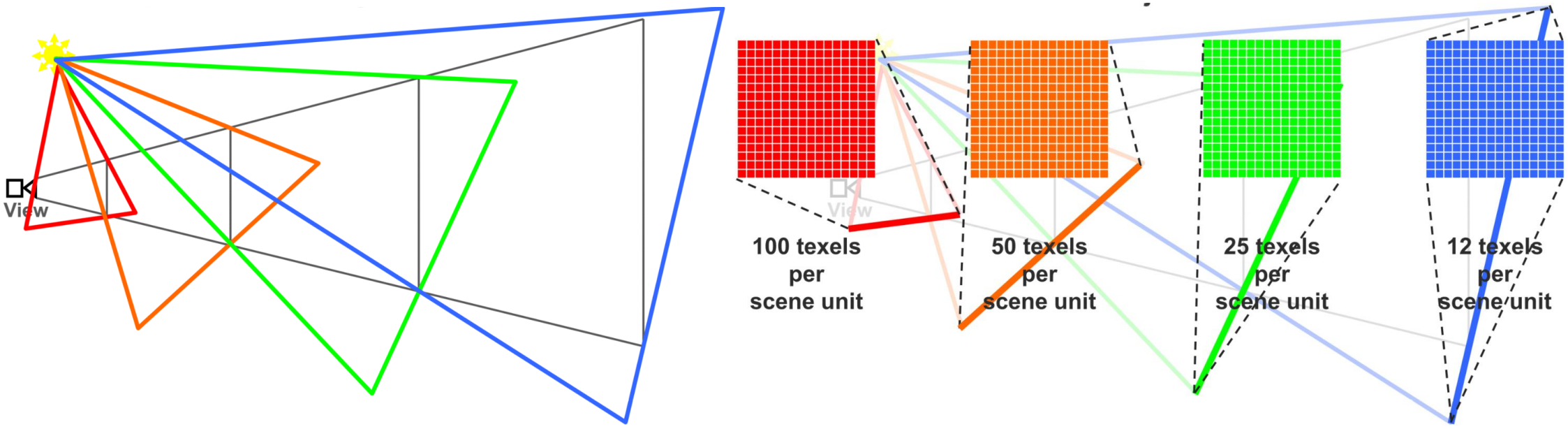
Shadow Map Artifacts

Source: https://docs.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps

- **Perspective aliasing** - occurs when the mapping of pixels in view space to texels in the shadow map is not a one-to-one ratio. Near the eye, the pixels are closer together, and many pixels map to the same shadow texels. Partial solutions are PSMs, LSPSMs, and LogPSMs, better solution are CSMs.
- **Projective aliasing** - occurs when the mapping between texels in camera space to texels in light space is not a one-to-one ratio, e.g. the tangent plane of the geometry becomes parallel to the light rays.
- **Shadow acne and erroneous self-shadowing**
- **Peter Panning** - shadows appear to be detached from and to float above the surface when there is insufficient precision in the depth buffer. Calculating tight near planes and far planes may help.

Cascaded Shadows Maps

- View frustum is covered with multiple shadow frustums
- Since shadow maps have the same resolution, the density of shadow map pixels goes down as we move away from the viewpoint



Ray Traced Shadows

See Nvidia OptiX Programming Guide, Section 8 – Interop. with OpenGL
https://raytracing-docs.nvidia.com/optix_6_0/guide_6_0/index.html#opengl#interoperability-with-opengl

```
rtBufferCreateFromGLBO
rtTextureSamplerCreateFromGLImage
```

- The goal is to create precise ray traced shadows with OptiX
- We will use three rendering passes
  - Geometry pass produces color and position (in WS) for each fragment in fragment shader:

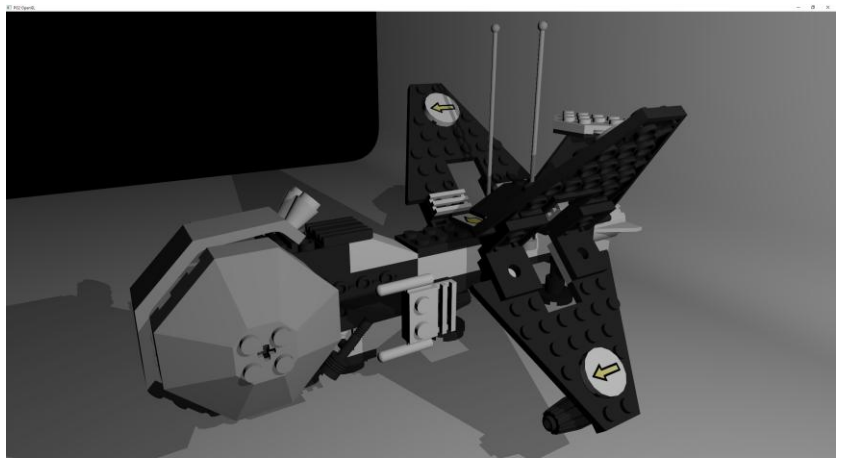
```
layout ( location = 0 ) out vec4 FragColor;
layout ( location = 1 ) out vec4 Position;
```

  - In shadow pass, rays from hit positions are traced via OptiX toward the light and potential occlusions are stored in shadow buffer (0 or 1)

```
rtTextureSampler<optix::float4, 2, cudaReadModeElementType> hit_points;
rtBuffer<optix::float4, 2> shadow_buffer;
```

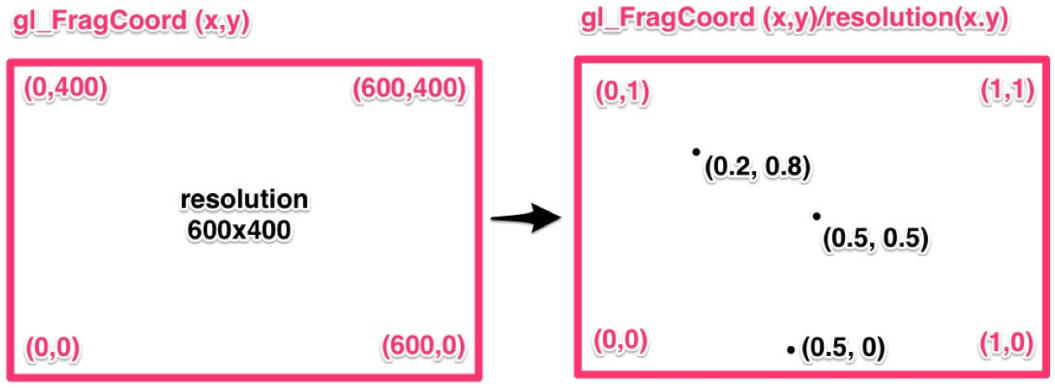
  - Third pass multiplies colors and attenuation buffer in fragment shader

```
uniform sampler2D color_map;
uniform sampler2D shadow_map;
uniform vec2 screen_size = vec2( 1.0f / 640.0f, 1.0f / 480.0f );
layout ( location = 0 ) out vec4 FragColor;

void main( void ) {
  vec2 tc = gl_FragCoord.xy * screen_size;
  vec4 color = texture( color_map, tc ).xyzw;
  vec4 shadow = texture( shadow_map, tc ).xyzw;
  FragColor = color * shadow;
}
```



Composition Pass in General

```
#version 450 core

layout ( location = 0 ) in vec4 in_position_cs;

void main( void ) {
  gl_Position = in_position_cs;
}
```




```
#version 450 core

uniform sampler2D color_map;
uniform sampler2D shadow_map;

uniform vec2 screen_size = vec2( 1.0f / 640.0f,
1.0f / 480.0f ); // should be set to the actual
screen size

layout ( location = 0 ) out vec4 FragColor;

void main( void ) {
  vec2 tc = gl_FragCoord.xy * screen_size;

  vec4 color = texture( color_map, tc ).xyzw;
  vec4 shadow = texture( shadow_map, tc ).xyzw;

  // lighting calculation with all the samplers

  FragColor = color;
}
```

Stencil Buffer

- A method to mask a section of the back (color) buffer
- More flexible then simpler scissor region (rendering takes place within a rectangular section)
- Used in multi-pass rendering algorithms to achieve special effects (decals, outlining, CSG, stencil shadows)
- Represented as a screen-sized buffer like depth and color buffers with at least 8-bit depth (typically, 32-bit buffer shared with 24-bit depth)
- Values in stencil buffer can be modified on per-pixel basis
- We can allow or discard a fragment drawn into a pixel based on the stencil value
- If there is no stencil buffer it is as if the stencil test always passes
- Initially, the stencil test is disabled

Stencil Buffer Test

- **We need to explicitly enable or disable stencil test**

`glEnable/Disable( GL_STENCIL_TEST )`

- **When enabled, all rendering calls will influence the stencil buffer**
- The actual outcomes are specified with `glStencilOp` and `glStencilFunc`
- The stencil operations are still processed when the test is disabled; the operation proceeds as if the stencil test had passed
- When the stencil test is enabled, the testing logic of the reference stencil value against the framebuffer stencil value is skipped
- Note that if the depth test fails, the stencil test no longer determines whether a fragment is drawn or not, but these fragments can still affect values in the stencil buffer

Stencil Buffer Clear

- **The stencil buffer can be cleared** along with the color buffer

```
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

- Initial value of the stencil buffer can be specified by

```
glClearStencil( 0 );
```

Stencil Buffer Mask

- **Mask enables or disables writing of individual bits in the stencil planes**
- Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer (stencil plane)
- Where a 0 appears, the corresponding bit is write-protected
- Initially, all bits are enabled for writing

```
glStencilMask(0xFF) // enables all bits for writing
```

Stencil Buffer Functions

- Front and back function and reference value for stencil testing **specify the conditions under which a fragment passes the stencil test**
- Possible functions are: GL_NEVER (always fails), GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS (always passes)
- Reference value is also used by other functions
- Stencil test passes `if ( ref & mask ) OP ( stencil & mask )`

```
glStencilFunc( GL_EQUAL, 0, 0xFF ); // accept fragments with stencil value equal to 0
```

- Use `glStencilFuncSeparate` to set front and back stencil state to different values

Stencil Buffer Operations

- **What happens when a stencil test passes or fails** is determined by

`glStencilOp`

- Function takes 3 parameters, which control what happens when

1. a stencil test fails
2. a stencil test passes but then the fragment fails the depth test
3. both the stencil and depth test pass (or the stencil test passes and depth testing is disabled)

- List of all possible actions: GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP, GL_INVERT
- The reference value for GL_REPLACE is set by `glStencilFunc`

Stencil Buffer Operations

- **What happens when a stencil test passes or fails** is determined by

`glStencilOpSeparate` separately **with respect to the side of the face**

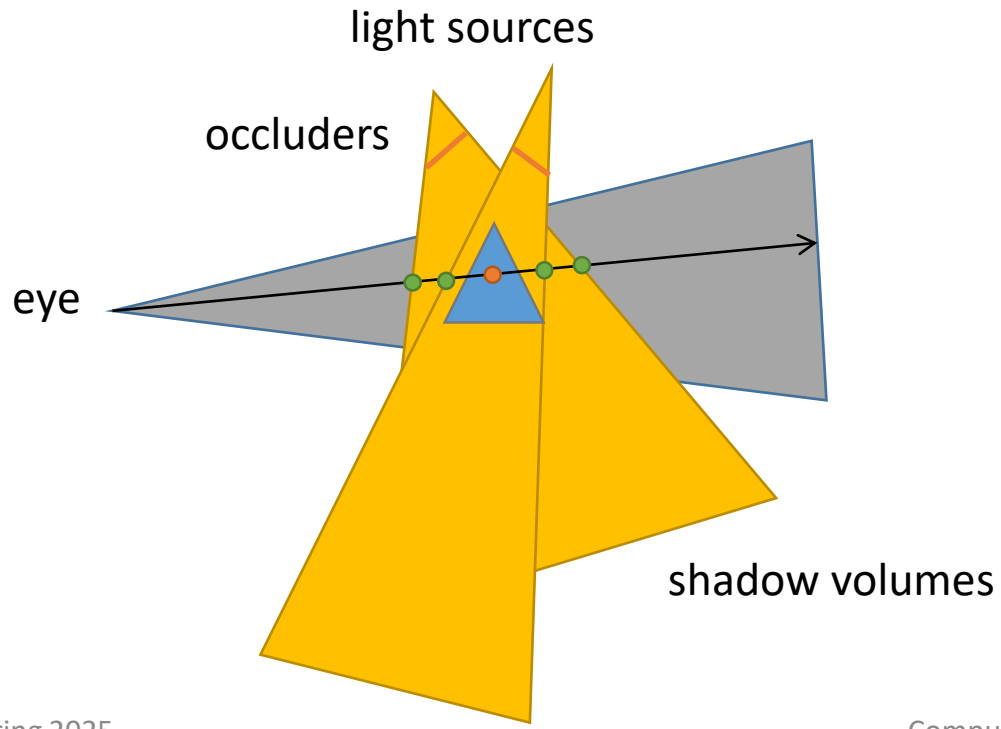
- Function takes 3 parameters as `glStencilOp` does plus the face side (GL_FRONT, GL_BACK, GL_FRONT_AND_BACK)
- Examples
- glStencilOp(GL_ZERO, GL_KEEP, GL_KEEP) - If the stencil test fails, this will set the stencil buffer at that test location to zero. If the stencil test passes, even if the depth test fails, then the stencil buffer is left alone.
- glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE) - If the stencil test fails, or passes when the depth test fails, nothing happens. If however both the stencil and depth test pass, the value in the stencil buffer is replaced by the current ref value of the stencil func.
- glStencilOp(GL_KEEP, GL_REPLACE, GL_KEEP) - If the stencil test passes, but the depth test fails, the value in the stencil buffer will be replaced. This can be used to determine where one object intersects with another, as the parts of object A that intersect object B will fail the depth test.

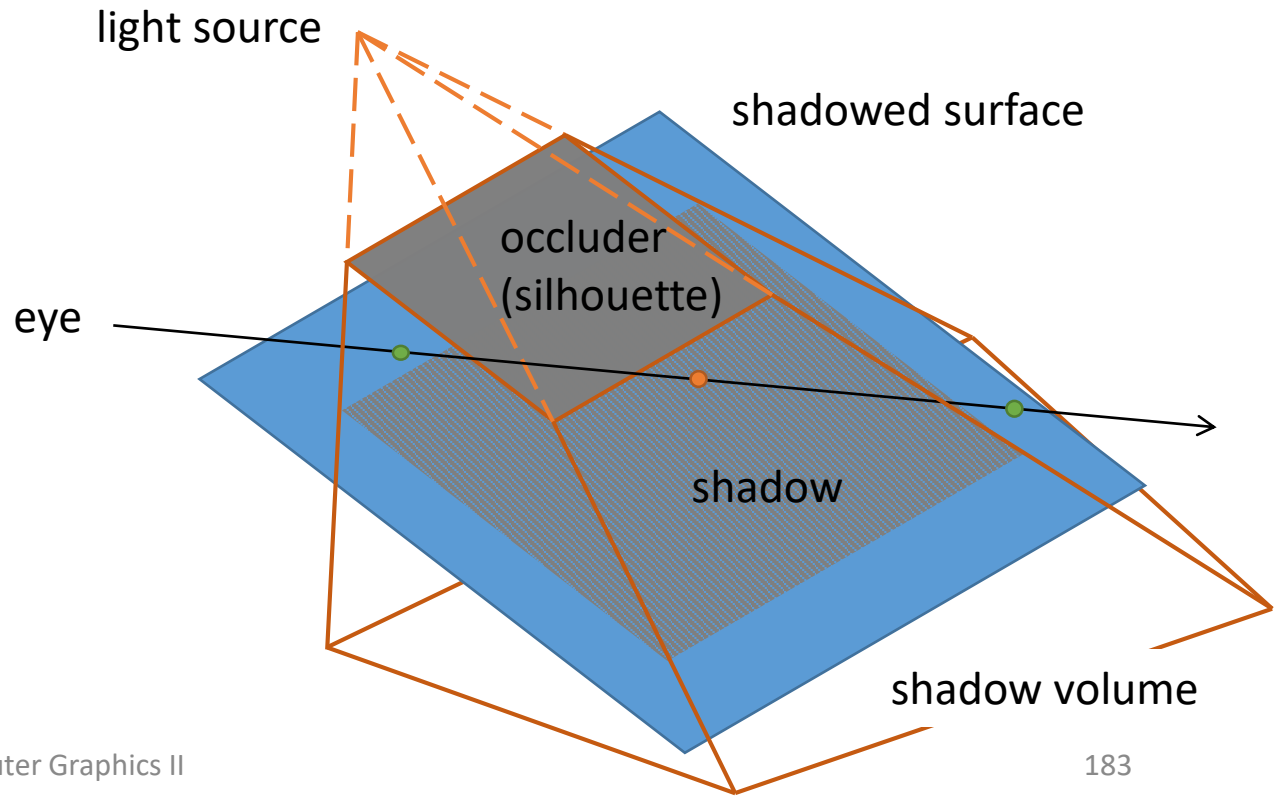
Shadow Volumes

- Shadow volumes are rendered with pixel accuracy (sharp, hard shadows)
- Suitable for perfect point (omni) lights, spot lights, and directional lights
- Can handle many difficult-to-shadow scenes
- Shadow casting mesh must be a crack-free surface
  - It must be a topological 2-manifold, i.e. a topological space whose points all have open disks as neighborhoods
  - It must be homeomorphic to the sphere
  - It must look locally like the plane everywhere
- We must be able to find the silhouette edges of every shadow caster and build the corresponding shadow volume

Shadow Volumes

- Multiple light sources are handled with multiple passes
- Far edges of volume sides are projected onto an infinitely large sphere



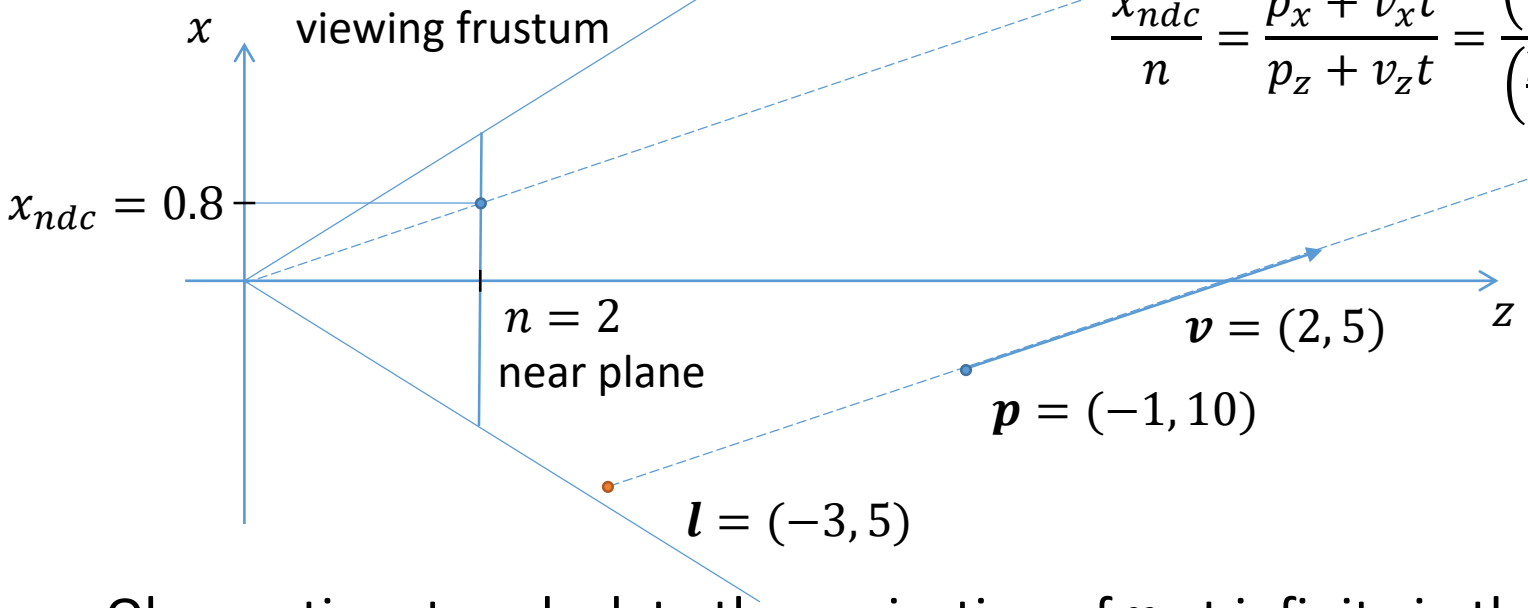


Working with points at infinity

infinitely large sphere
$R = \infty$

$\boldsymbol{p}_{\boldsymbol{v}}^{\infty} = \boldsymbol{p} + \boldsymbol{v}\infty$

x_{ndc} is x-coordinate of $\boldsymbol{p}_{\boldsymbol{v}}^{\infty}$ ($\boldsymbol{p}$ projected at infinity in the direction of $\boldsymbol{v}$) in NDC



$$\frac{x_{ndc}}{n} = \frac{p_x + v_x t}{p_z + v_z t} = \frac{\left(\frac{p_x}{t} + v_x\right)\cancel{t}}{\left(\frac{p_z}{t} + v_z\right)\cancel{t}} = \left.\frac{\frac{p_x}{t} + v_x}{\frac{p_z}{t} + v_z}\right| \overset{t\to\infty}{=} \frac{v_x}{v_z}$$

$$x_{ndc} = n\frac{v_x}{v_z}$$

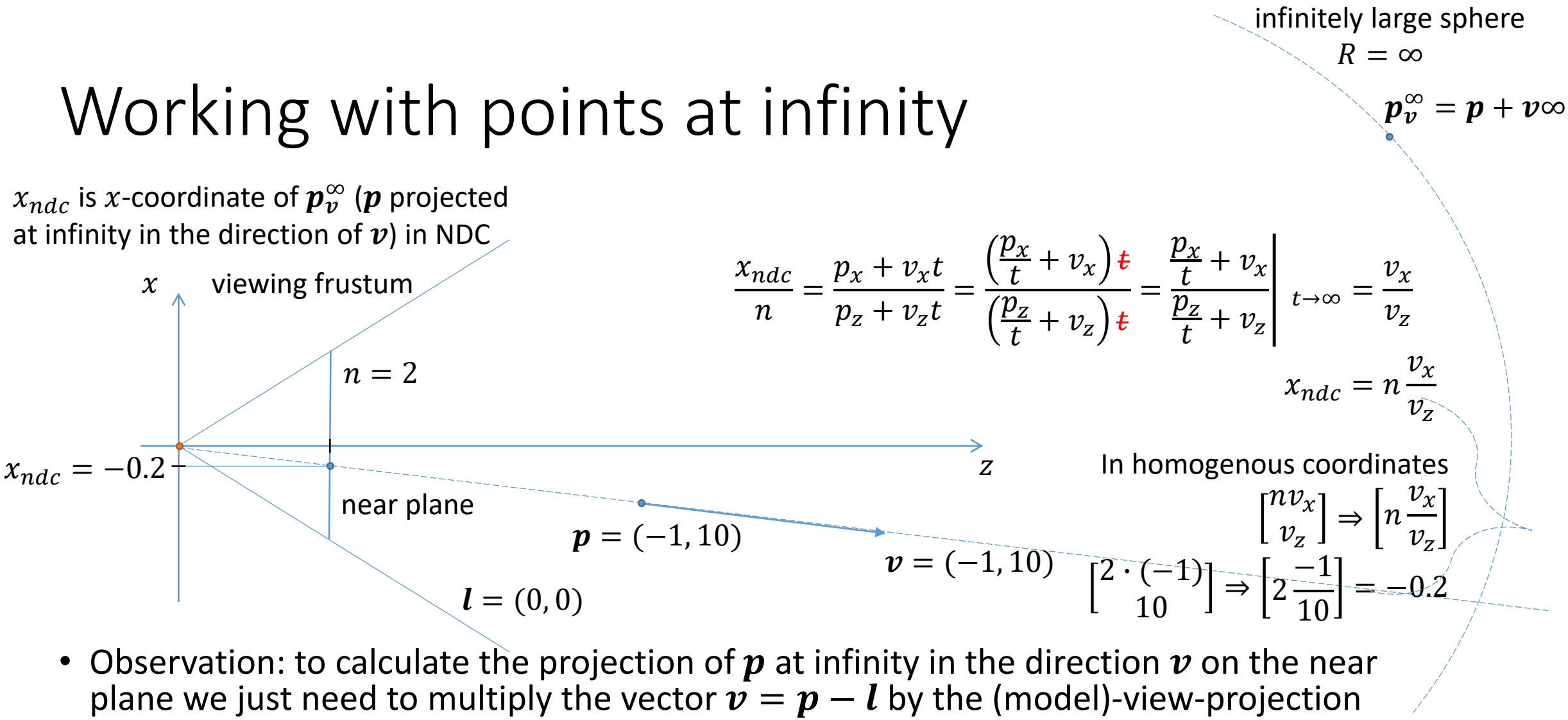
In homogenous coordinates

$$\begin{bmatrix} nv_x \\ v_z \end{bmatrix} \Rightarrow \left[n\frac{v_x}{v_z}\right]$$

$$\begin{bmatrix} 2 \cdot 2 \\ 5 \end{bmatrix} \Rightarrow \left[2\frac{2}{5}\right] = 0.8$$

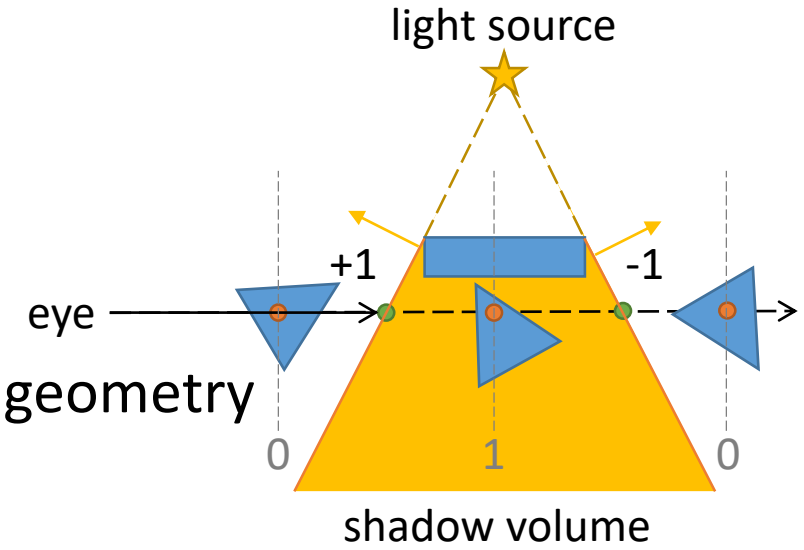
- Observation: to calculate the projection of $\boldsymbol{p}$ at infinity in the direction $\boldsymbol{v}$ on the near plane we just need to multiply the vector $\boldsymbol{v} = \boldsymbol{p} - \boldsymbol{l}$ by the (model)-view-projection matrix and apply perspective divide on it (gl_Position = mvp * vec4($\boldsymbol{p}$.xyz – $\boldsymbol{l}$, 0.0f);)

Working with points at infinity



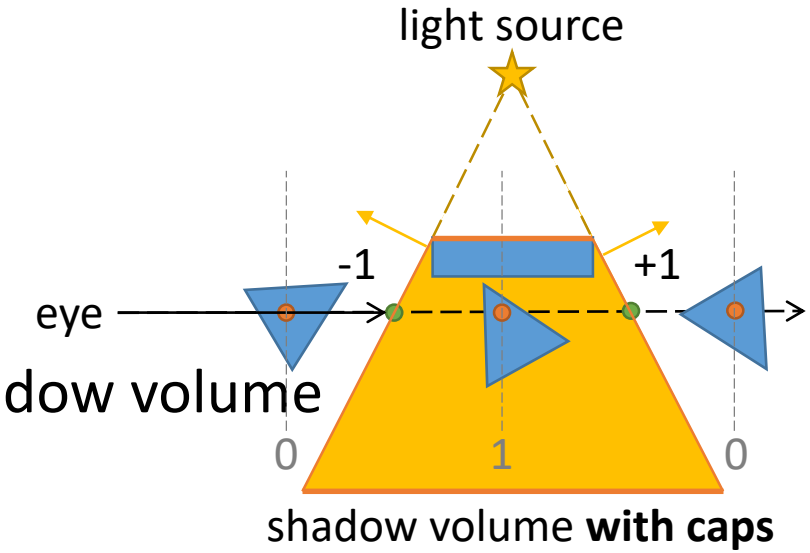
- Observation: to calculate the projection of $\boldsymbol{p}$ at infinity in the direction $\boldsymbol{v}$ on the near plane we just need to multiply the vector $\boldsymbol{v} = \boldsymbol{p} - \boldsymbol{l}$ by the (model)-view-projection matrix and apply perspective divide on it (gl_Position = mvp * vec4($\boldsymbol{p}$.xyz, 0.0f);)

Z-pass algorithm



- 1. Do the depth pass (with ambient lighting) with scene geometry
- 2. Clear the stencil buffer (set to zero)
- 3. Draw shadow volume in stencil buffer (only fragments of the shadow volume placed in front of the nearest geometry are considered)
  - no depth and no color writes
  - store +1 to the stencil buffer for front-facing triangles
  - store -1 to the stencil buffer for back-facing triangles
- 4. Do the lighting pass: every fragment outside the shadow volume (i.e. fragment with zero value in the stencil buffer) receives full light

Z-fail algorithm

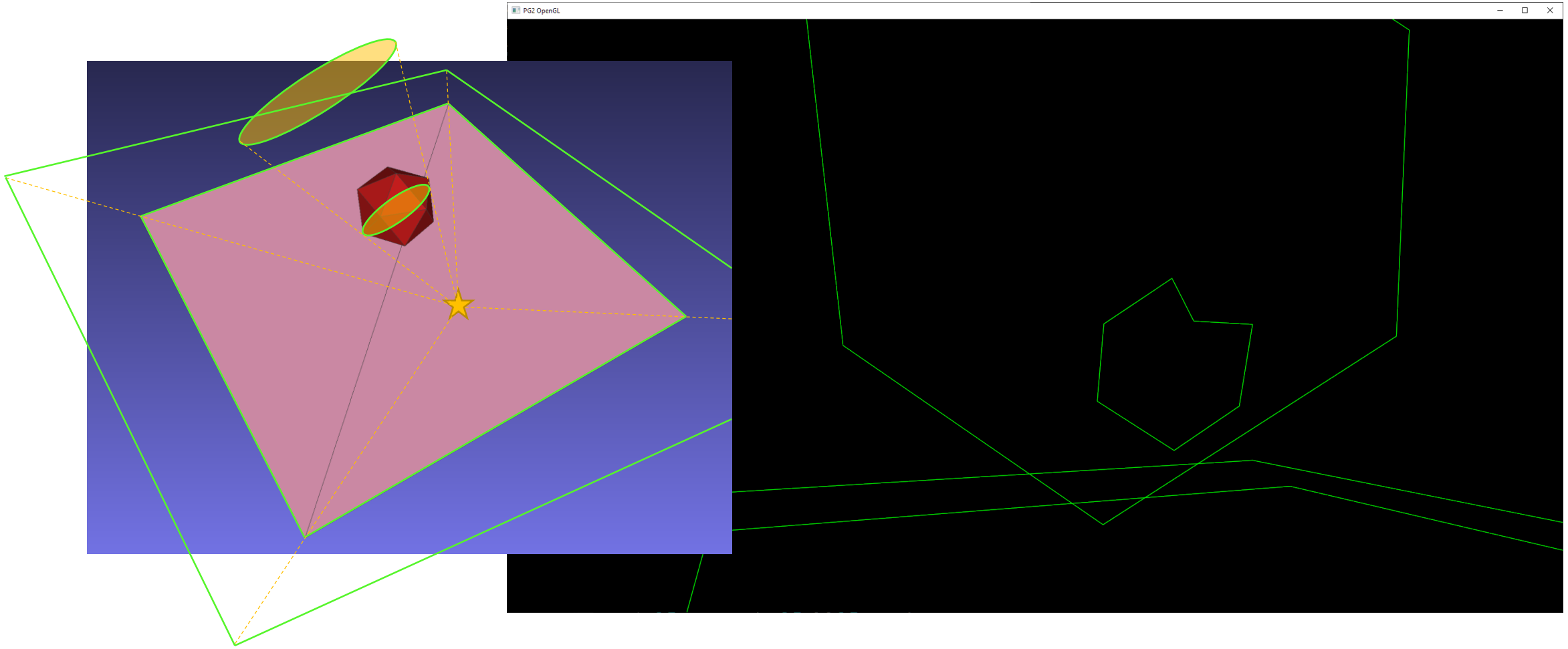


- Z-pass has a problem when the camera is inside the shadow volume
- Z-fail (Carmac's reverse, 2000) solves this problem
- Instead of counting the ray-volume intersections in front of the actual geometry, we can count the intersections behind it (i.e. when the depth test fails)
- We write to the stencil buffer if the depth test fails
  - store -1 to the stencil buffer for front-facing triangles
  - store +1 to the stencil buffer for back-facing triangles
- Z-fail works for any case, but unlike z-pass, it must be assured that the volume is closed both at its front end and at its back end

Shadow Volume

- We can generate a dedicated vertex buffer with shadow volumes for each shadow caster on CPU but this would be very impractical, especially in case of dynamic lights or shadow casters
- We will limit ourselves to closed 2-manifold polygon meshes (no holes, cracks, or self-intersections)
- Rendering of shadow volumes consists of three steps
  - Rendering the front cap
  - Rendering the back cap
  - Rendering the shadow caster extruded silhouette (the sides of volume)

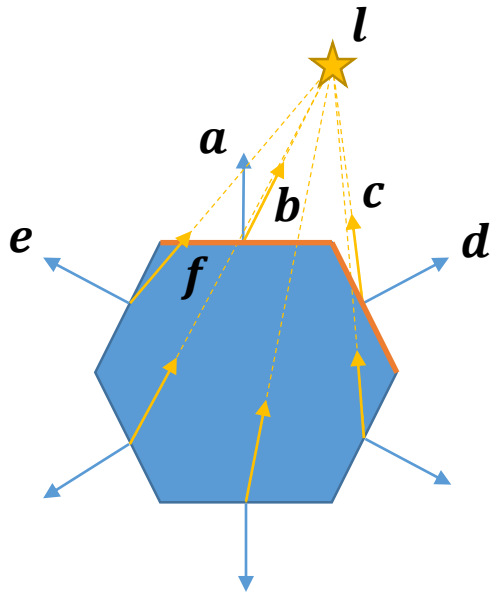
Extruded Silhouette Example



Shadow Caster Front Cap

- The following condition applies to the faces of the front cap of the shadow volume

```
if ( dot( omega_i, face_normal ) > 0 ) {
  // face is a front cap
}
```



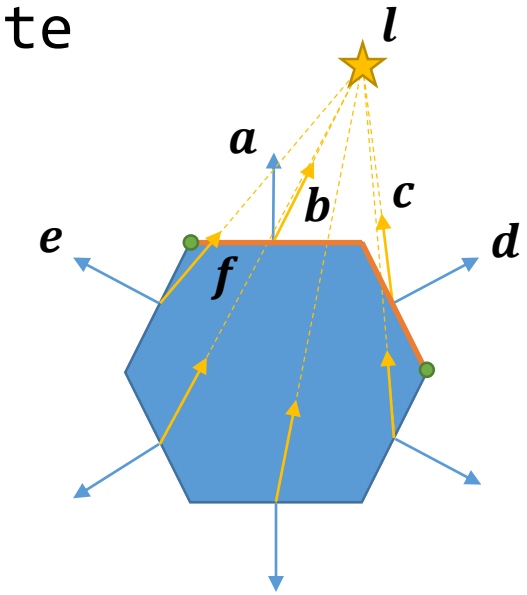
$\boldsymbol{a} \cdot \boldsymbol{b} > 0$
$\boldsymbol{e} \cdot \boldsymbol{f} < 0$

Note that the direction ω_i can be represented as $\frac{\boldsymbol{l}-\boldsymbol{p}}{|\boldsymbol{l}-\boldsymbol{p}|}$ where $\boldsymbol{p}$ is any point of the given face

Shadow Caster Silhouette

- The following condition applies to the edges of the silhouette

```
if ( sign( dot( omega_i, face_normal ) ) !=
     sign( dot( omega_i, adjacent_face_normal ) ) ) {
  // edge is a silhouette
}
```



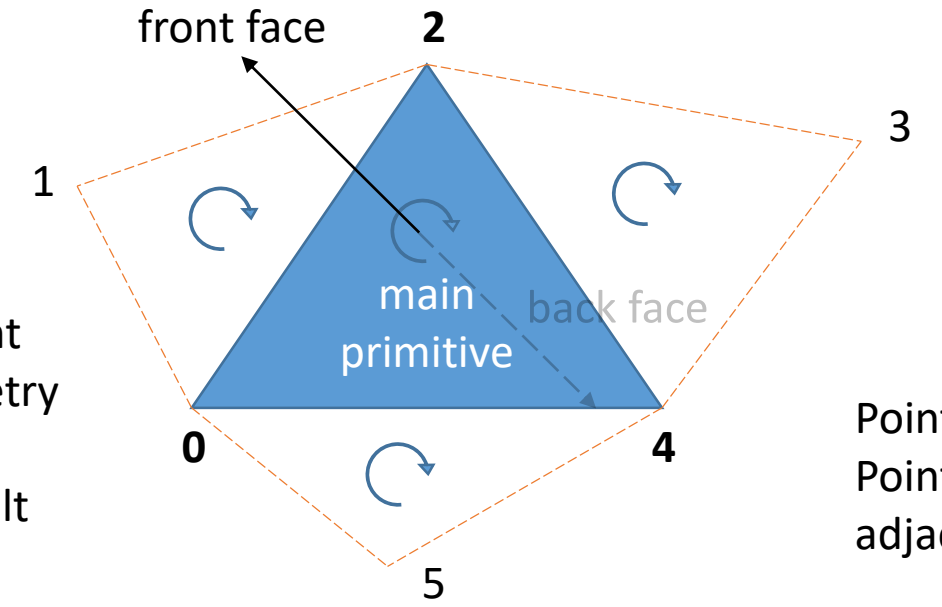
$$\text{sgn}(\boldsymbol{a}\cdot\boldsymbol{b}) = \text{sgn}(\boldsymbol{c}\cdot\boldsymbol{d})$$
$$\text{sgn}(\boldsymbol{a}\cdot\boldsymbol{b}) \neq \text{sgn}(\boldsymbol{e}\cdot\boldsymbol{f})$$

Also note that we need some additional information about the topology of our shadow caster – the triangle adjacency

Note that the direction ω_i can be represented as $\frac{\boldsymbol{l}-\boldsymbol{p}}{|\boldsymbol{l}-\boldsymbol{p}|}$ where $\boldsymbol{p}$ is any point of the given face

Adjacency Primitives

- We will deal with GL_TRIANGLES_ADJACENCY further
- Each triangle edge has an adjacent vertex that can be accessed in VS, TS, and GS
- 6 vertices per main function call are given, 6N in total (where N is the number of triangles to draw)



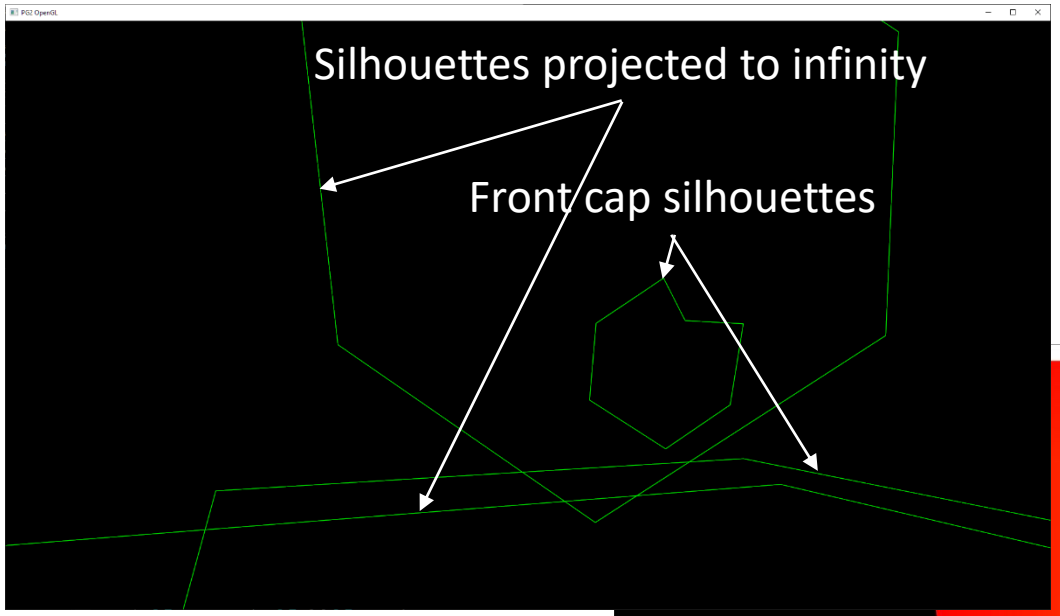
Note that the default OpenGL ordering of vertices is CW (front face). Front faces of the geometry stored in OBJ files are CCW
glFrontFace(GL_CW); // default
glFrontFace(GL_CCW); // OBJ

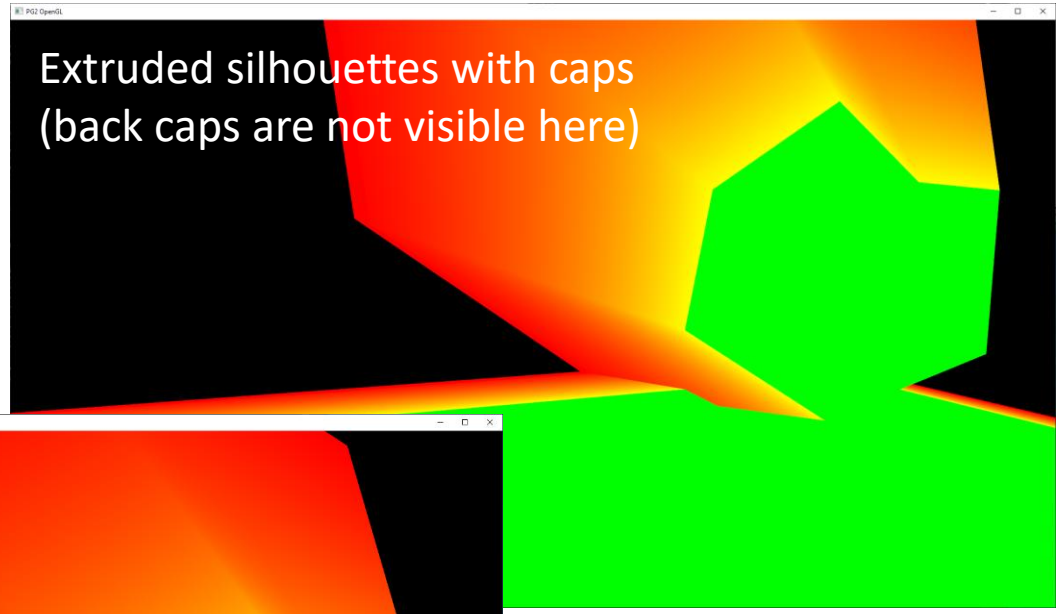
Points 0, 2, and 4 define the triangle
Points 1, 3, and 5 define vertices of adjacent triangles

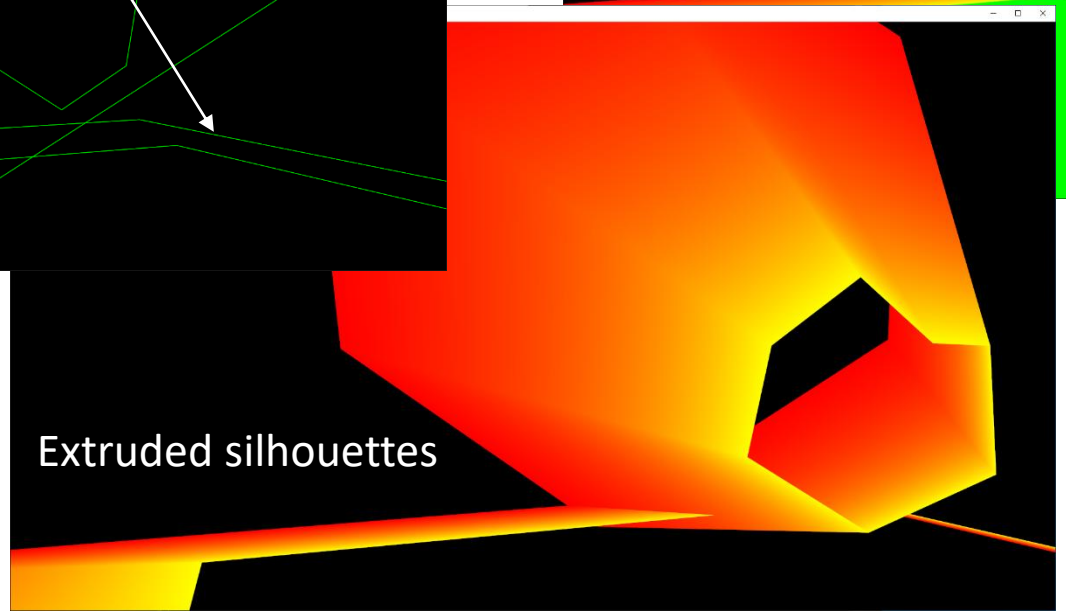
Shadow Volume Ex.

on the next slide why

When rendering shadow volume to the stencil buffer, we need to allow drawing (pass) fragments outside the NDC depth range (-1, 1)
glEnable(GL_DEPTH_CLAMP); and glDepthFunc(GL_LEQUAL);
The depth test must be enabled even if we only draw to the stencil buffer

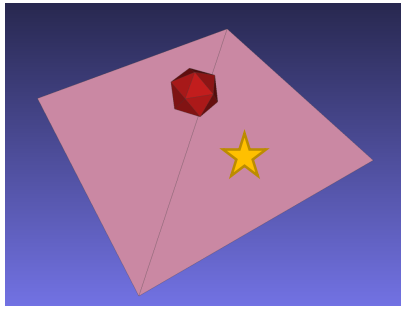






This is only for visualization and debugging purposes as the shadow volumes are drawn only into the stencil buffer

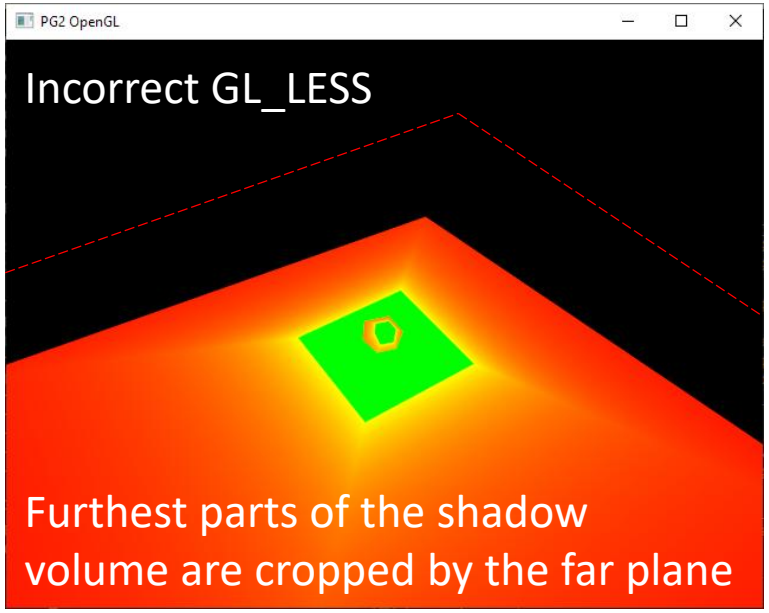
Test scene

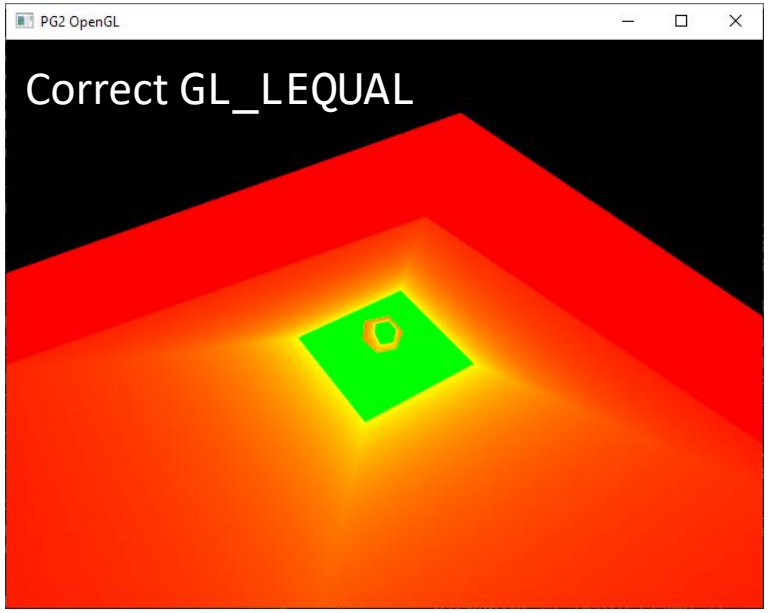


GL_LESS or GL_LEQUAL?

```
glEnable( GL_DEPTH_CLAMP );
glEnable( GL_DEPTH_TEST );
glDepthFunc( GL_LESS or GL_LEQUAL );
```

- The default clear value for the depth is 1.0, which is equal to the depth of the far clipping plane and therefore the furthest depth that can be represented. All fragments will be closer than that, except for fragments corresponding to vertices projected to infinity (w = 0), so they will be inadvertently discarded if the depth func is set to GL_LESS. The depth test for furthest fragments will fail here because 1 is not less than 1. That's why we should use less or equal (GL_LEQUAL) here instead.

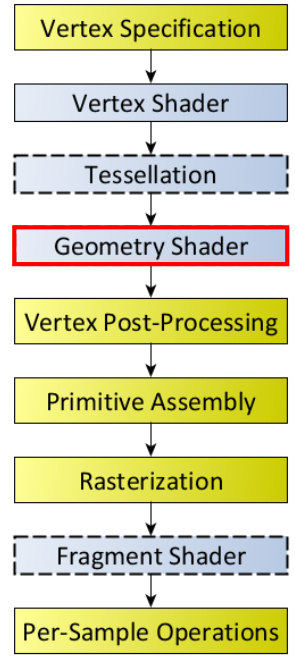




GS is core in 4.6. While geometry shaders have had previous extensions like GL_EXT_geometry_shader4 and GL_ARB_geometry_shader4, these extensions expose the API and GLSL functionality in very different ways

Geometry Shader

- Geometry shaders process each incoming primitive returning zero or more output primitives
- Additional topologies that geometry shaders made available
  - GL_LINES_ADJACENCY
  - GL_LINE_STRIP_ADJACENCY
  - **GL_TRIANGLES_ADJACENCY**
  - GL_TRIANGLE_STRIP_ADJECENCY



Sources: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
https://www.khronos.org/opengl/wiki/Geometry_Shader

Geometry Shader Inputs

- Geometry shaders provide the following built-in input variables for vertices

```
in gl_PerVertex {
  vec4 gl_Position; // meaning is given by prior stages
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[];
```

- There are also input values based on primitives
- gl_PrimitiveIDIn - the current input primitive's ID, based on the number of primitives processed by the GS since the current drawing command started
- gl_InvocationID - the current instance, as defined when instancing geometry shaders

Geometry Shader Outputs

- Geometry shaders provide the following built-in output variables for vertices

```
out gl_PerVertex {
   vec4 gl_Position; // the clip-space output position of the current vertex
   float gl_PointSize;
   float gl_ClipDistance[];
};
```

- and there are also output values gl_PrimitiveID, gl_Layer, and gl_ViewportIndex
- For additional information consult https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)

Geometry Shader Sample Code

Input layout qualifier: points, lines, lines_adjacency, triangles or triangles_adjacency

Output layout qualifier: points, lines_strip or triangles_strip

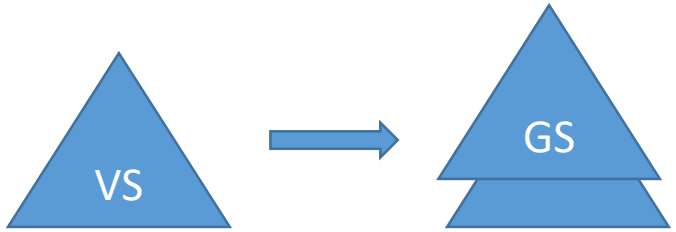
Maximum number of emitted vertices

② geometry shader

```
#version 460 core
layout ( triangles_adjacency ) in;
layout ( triangle_strip, max_vertices = 6 ) out;
uniform mat4 mvp; // (Model) View Projection matrix
void main() {
  // built-in input variable (array) gl_in[].gl_Position is expected to contain world-space coordinates of triangle (and adjacent) vertices
  // in this example, GL_TRIANGLES_ADJACENCY is expected
  const vec3 v0 = gl_in[0].gl_Position.xyz;
  const vec3 v1 = gl_in[2].gl_Position.xyz;
  const vec3 v2 = gl_in[4].gl_Position.xyz;
  // generate the first triangle
  gl_Position = mvp * v0; // we should end up with the clip-space output position of the current vertex here
  EmitVertex();
  gl_Position = mvp * v1;
  EmitVertex();
  gl_Position = mvp * v2;
  EmitVertex();
  EndPrimitive();
  // generate the second triangle 10 units above the first one
  gl_Position = mvp * ( v0 + vec4( 0.0f, 0.0f, 10.0f, 0.0f ) );
  EmitVertex();
  gl_Position = mvp * ( v1 + vec4( 0.0f, 0.0f, 10.0f, 0.0f ) );
  EmitVertex();
  gl_Position = mvp * ( v2 + vec4( 0.0f, 0.0f, 10.0f, 0.0f ) );
  EmitVertex();
  EndPrimitive();
}
```

① ***vertex shader***

```
#version 460 core
layout ( location = 0 ) in vec4 in_position_ms;

void main( void ) {
  gl_Position = in_position_ms;
}
```



③ ***fragment shader***

```
#version 460 core
layout ( location = 0 ) out vec4 FragColor;

void main( void ) {
  FragColor = vec4( 0.0f, 0.0f, 1.0f, 1.0f );
}
```

Shadow Volume Loop 1/6

- Setup camera as usual
- Depth pass with ambient lighting
- This is a standard pass with output to then color buffer and the depth buffer

```
glStencilMask( 0xFF );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
```

- Shadow pass
  - Disable writing to the color buffer and the depth buffer

```
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glDepthMask( GL_FALSE );
```

  - Enabled GL_DEPTH_CLAMP, depth test is still enabled

```
glEnable( GL_DEPTH_CLAMP );
```

Shadow Volume Loop 2/6

- Shadow pass cont.
    - Disable face fulling and define the front and back facing polygons as needed

```
glDisable( GL_CULL_FACE );
glFrontFace( GL_CCW );
```

    - Enable stencil test and writing to the stencil buffer

```
glEnable( GL_STENCIL_TEST );
glStencilMask( 0xFF );
```

    - Set the appropriate stencil operations for front and back faces separately
    - Set the stencil reference value to 0 and mask to 255

Shadow Volume Loop 3/6

- Shadow pass cont.
    - Set the appropriate stencil operations for front and back faces separately

```
glStencilOpSeparate( GL_FRONT, GL_KEEP, GL_DECR_WRAP, GL_KEEP );
glStencilOpSeparate( GL_BACK, GL_KEEP, GL_INCR_WRAP, GL_KEEP );
```

    - Set the stencil reference value to 0 and mask to 255

```
glStencilFunc( GL_ALWAYS, 0, 0xFF );
```

Shadow Volume Loop 4/6

- Shadow pass cont.
  - Draw the geometry with the following triplet of shaders
    - Simple pass-through vertex shader
    - Geometry shader that finds silhouette edges, extrudes them to form the shadow volume sides and puts the front and back cap
    - Fragment shader may be empty

Shadow Volume Loop 5/6

- Lighting pass
  - Enable drawing to the color buffer (and the back buffer)

```
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
glDepthMask( GL_TRUE );
```

  - Disable depth camp and change depth func to GL_LEQUAL

```
glDisable( GL_DEPTH_CLAMP );
glDepthFunc( GL_LEQUAL );
```

  - Enable (back) face culling (optional)

```
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );
```

Shadow Volume Loop 6/6

- Lighting pass cont.
  - Keep enabled stencil test but disable writing to the stencil buffer

```
glEnable( GL_STENCIL_TEST );
glStencilMask( 0 );
glStencilOpSeparate( GL_FRONT_AND_BACK, GL_KEEP, GL_KEEP, GL_KEEP );
```

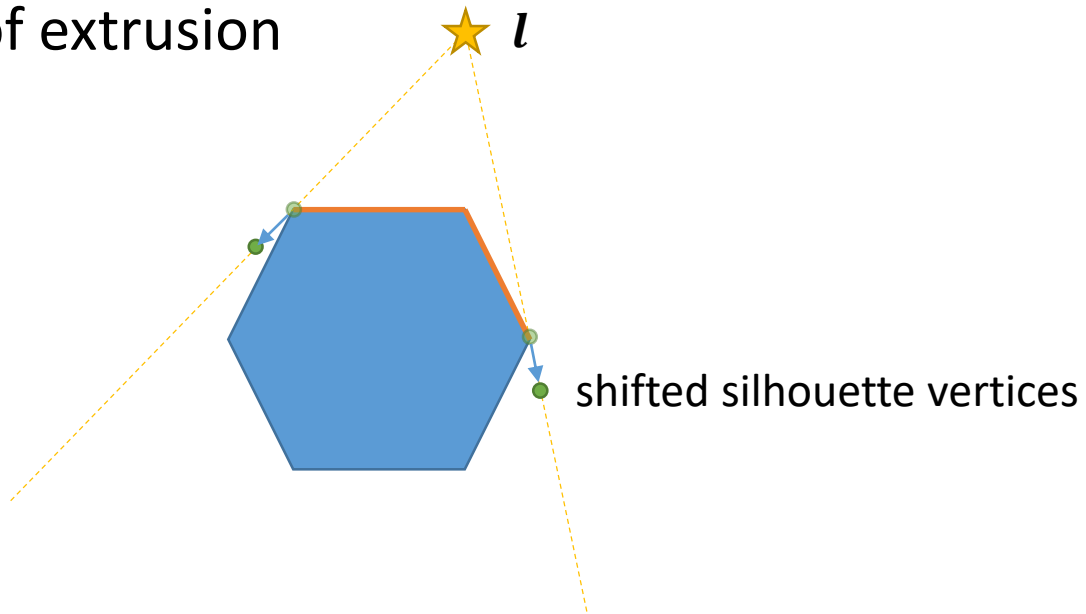
  - Set the reference stencil value to 0

```
glStencilFunc( GL_EQUAL, 0, 0xFF );
```

  - Draw the scene with lighting shader (stencil test will accept only the lit fragments with zero stencil value)

Shadow Volume Final Remark

- We must ensure that the front cap of the shadow volume is generated slightly behind the actual geometry of the shadow caster. This will prevent interference between the shadow volume and the illuminated part of the geometry
- This can be done easily in the geometry shader by slight shift of silhouette vertices in the direction of extrusion



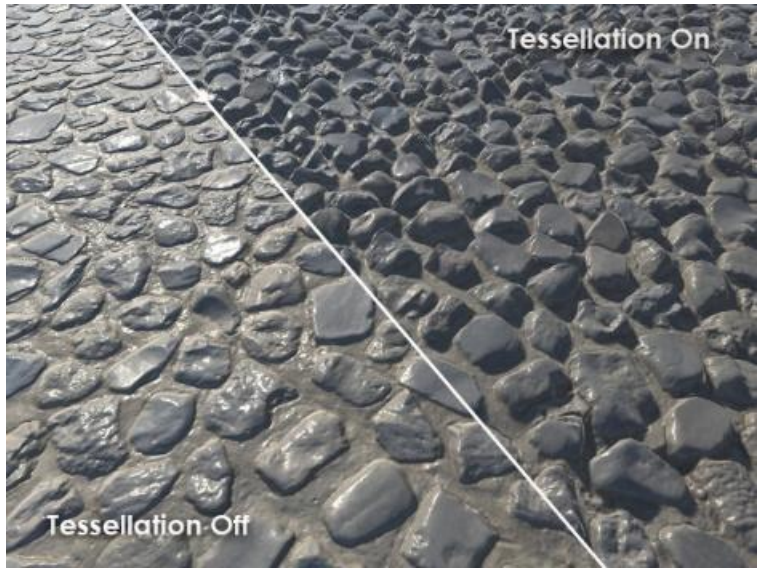
Hardware Tessellation

- Subdivision of polygons or lines
- Dynamic Level of Detail (LoD) – adjusts tessellation level according to the camera distance or screen size of the visible polygons
- Rendering of parametric surfaces or curves – store only control points in VBOs and evaluate the actual surface on the fly
- Displacement mapping – changes the objects silhouette not just normals

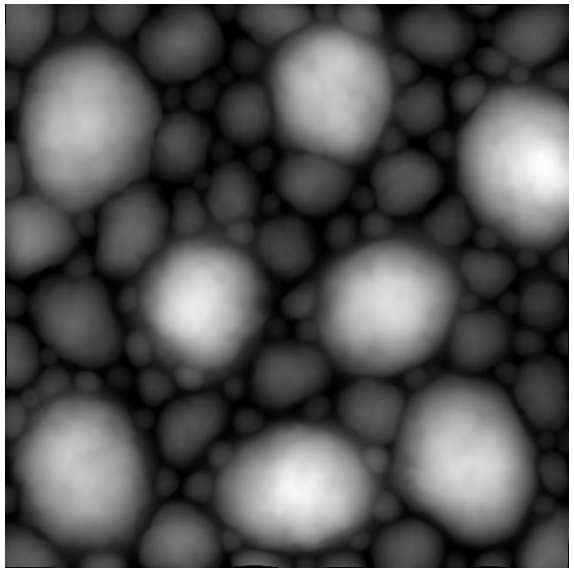
Source: Unigine Heaven Benchmark

Hardware Tesselation

- Use tessellation to increase detail
  - reduce your vertex count in VBO to minimum level of detail (LOD)
  - add in detail using a texture heightmap



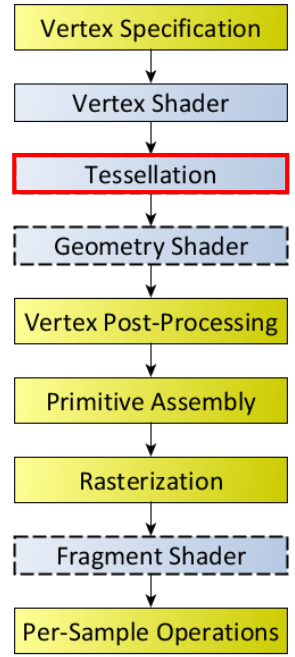
Source: https://unityfreaks.com/asset/3999



Example of a height map

Tessellation Shaders

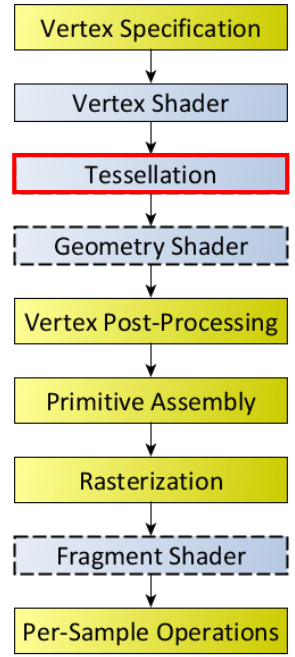
- Part of programmable vertex processing pipeline introduced in OpenGL 4.0 (2010) and core since 4.6 (2017)
- Tessellation is a process that reads a patch primitive and generates new primitives
- The new generated primitives are formed by subdividing a single triangle or quad primitive
- Generally, the process of tessellation involves subdividing a patch of some type, then computing new vertex values (position, color, texture coordinates) for each of the vertices generated by this process
- Tessellation shaders are faster than geometry shaders



Use Cases of GS and TS

- Use GS when you want to convert an input topology into a different output topology
- Use GS when you need some sort of geometry processing after TS

- Use TS when on of the build-in tessellation patterns suits your needs
- Use TS when you need more than 6 input vertices to define the surface being tessellated
- Use TS when you need more output vertices than a GS can provide



What is Patch?

- A patch primitive is a general-purpose primitive, where every n vertices is a new patch primitive
- The number of vertices per patch can be defined on the application-level

```
void glPatchParameteri(GL_PATCH_VERTICES, value);
```

where value is in the range [1, GL_MAX_PATCH_VERTICES)

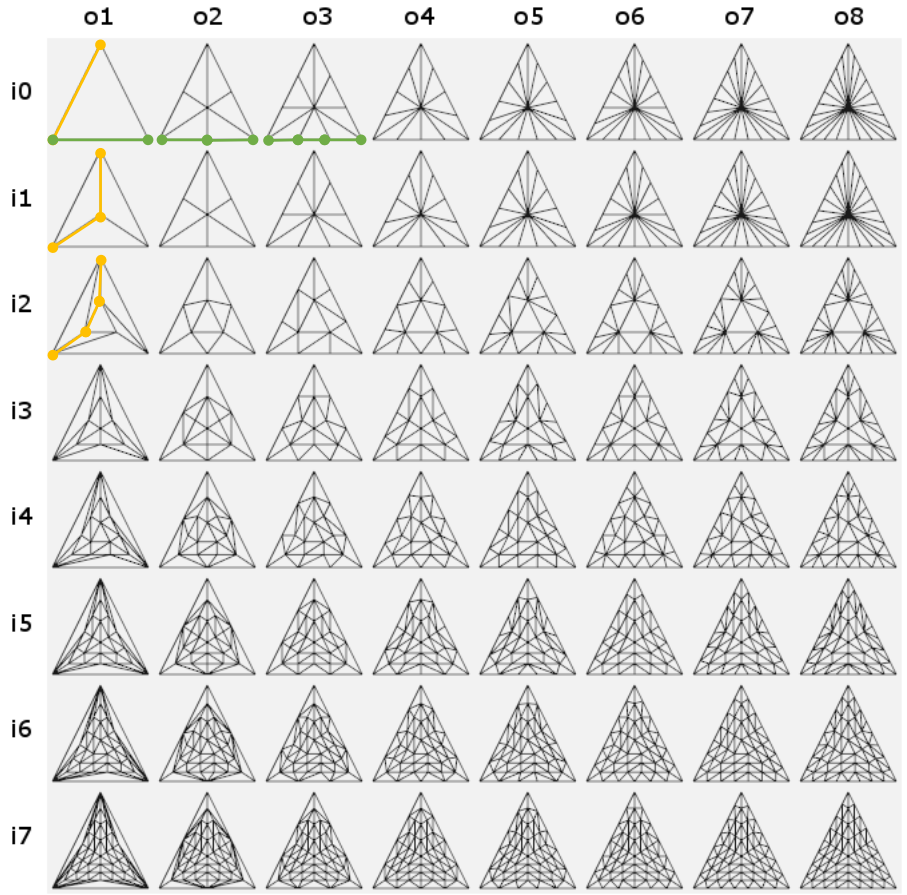
- The maximum number of patch vertices is implementation-dependent, but will never be less than 32
- Patch primitives are always a sequence of individual patches; there is no such thing as a "patch strip" or "patch loop" or such

Tessellation Levels

- Inner and outer tesselation levels

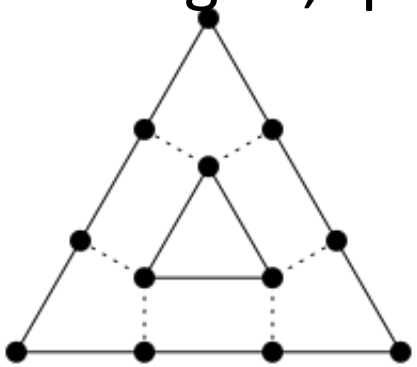
```
gl_TessLevelOuter[4];
gl_TessLevelInner[2];
```

- The amount of tessellation that is done over the abstract patch type (defines how many segments an edge is tessellated into)
- Outer tesselation levels make it possible for two or more patches to properly connect
- Triangles uses one inner level and 3 outer levels

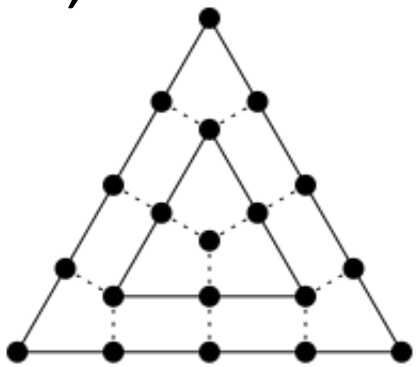


What is Tessellation Pattern/Primitive

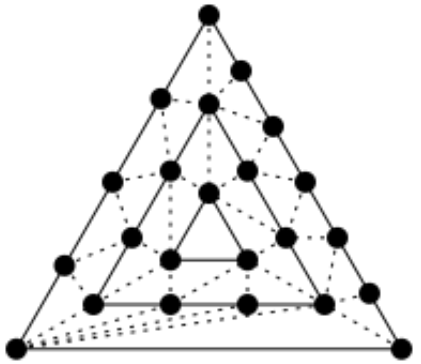
- Triangles, quads, isolines



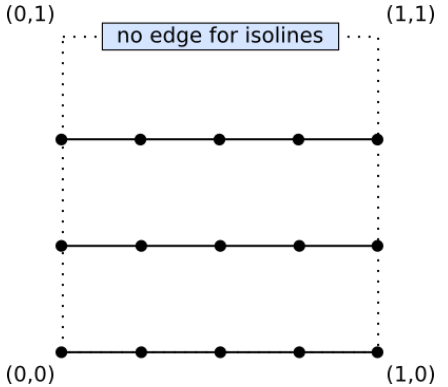
Inner Tess = 3

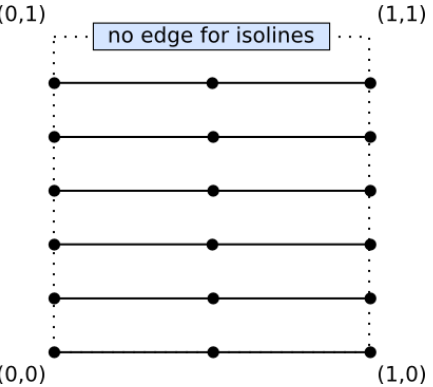


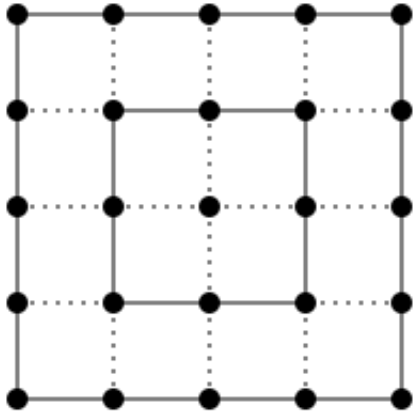
Inner Tess = 4



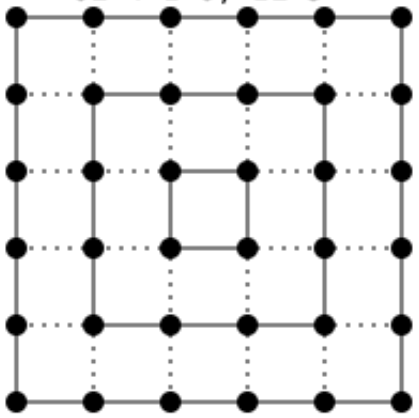
OL 4 1 6; IL 5



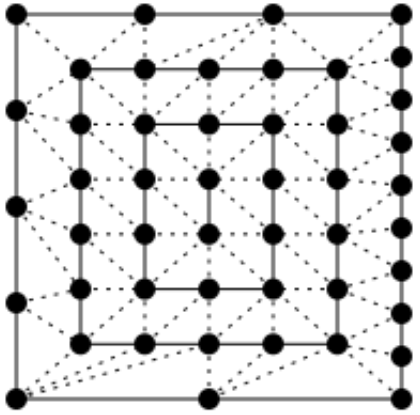




Inner Tess = 4 4



Inner Tess = 5 5

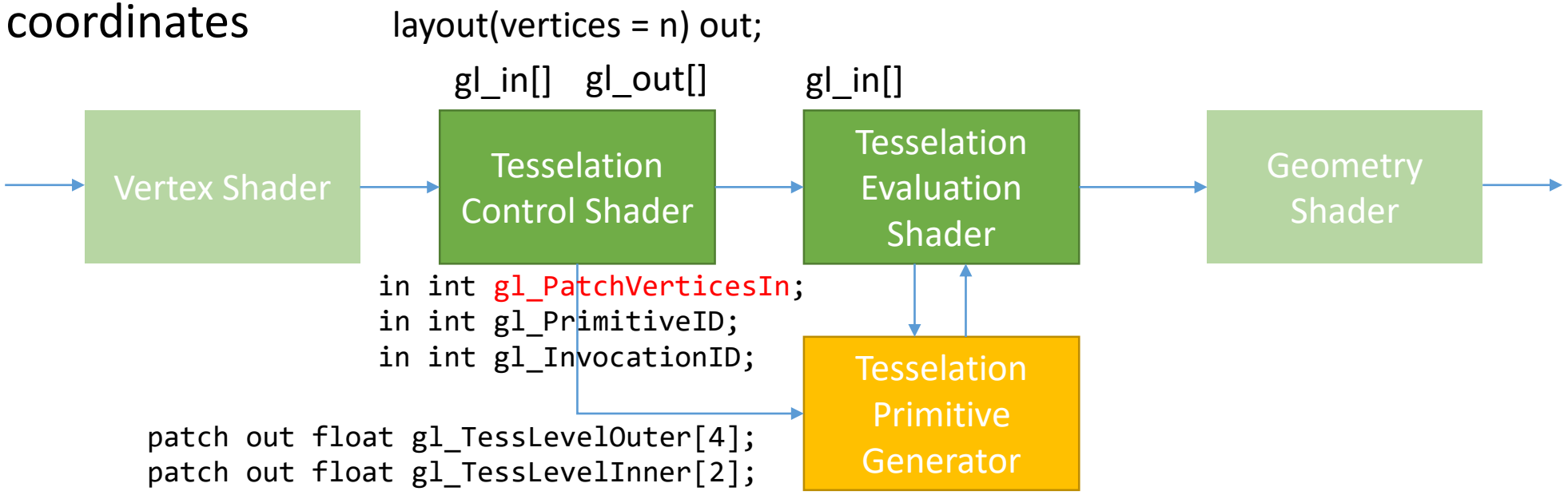


OL 4 2 9 3; IL 6 7

Tessellation Shaders

```
in gl_PerVertex {
  vec4 gl_Position;
  float gl_PointSize;
  vec4 gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices], gl_out[];
```

- Tessellation functionality is controlled by two types of tessellation shaders
  - Tessellation control shaders (optional) – controls how much tessellation a particular patch gets
  - Tessellation evaluation shaders (mandatory) – evaluates the surface in uvw coordinates



Tessellation Control Shader

- Transforms the input coordinates to a regular surface representation
- It also computes the required tessellation level based on distance to the eye, screen space spanning, hull curvature, or displacement roughness
- There is one invocation per output vertex

```
glCreateShader( GL_TESS_CONTROL_SHADER  ); # also the VS must be preset
…
glPatchParameteri( GL_PATCH_VERTICES, number of vertices per patch );
glDrawArrays( GL_PATCHES, 0, number of vertices );
```

Tessellation Control Shader Inputs

- User-defined inputs can be declared as unbounded arrays

```
in vec2 texCoord[];
```

- Built-in input variables

```
in int gl_PatchVerticesIn; // the number of vertices in the input patch
in int gl_PrimitiveID; // the index of the current patch
in int gl_InvocationID; // the index of the TCS invocation within this patch

in gl_PerVertex {
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices]; // variables have only the meaning the vertex shader
that passed them gave them
```

Tessellation Control Shader Outputs

- A user-defined per-vertex output variable would be defined as such

```
out vec2 vertexTexCoord[];
```

- Per-patch output variables

```
patch out vec4 data;
```

- Defining how many vertices this patch will output

```
layout( vertices = n ) out;
```

- Built-in patch output variables (the outer and inner tessellation levels used by the tessellation primitive generator)

```
patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

Tessellation Evaluation Shaders

- A user-defined per-vertex output variable and per-patch output variables as in TCS
- Built-in input variables

```
in vec3 gl_TessCoord; // the triplet of <0,1> coordinates


in gl_PerVertex {
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[6];
} gl_in[];
```

layout({ triangles | quads | isolines } , { equal_spacing | fractional_even_spacing | fractional_odd_spacing } , { ccw | cw } , point_mode) in;

Tessellation Primitive Generator

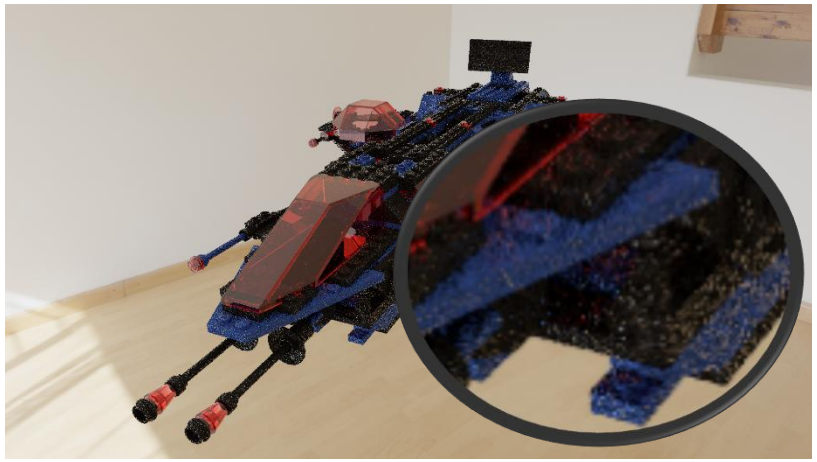
- Primitive generation is a fixed-function stage responsible for creating a set of new primitives from the input patch
- It is executed only if a TES is active in the current pipeline
- It consumes all vertices from the TCS and emits vertices for the triangles, quads, or isolines primitives

Geometry Instancing

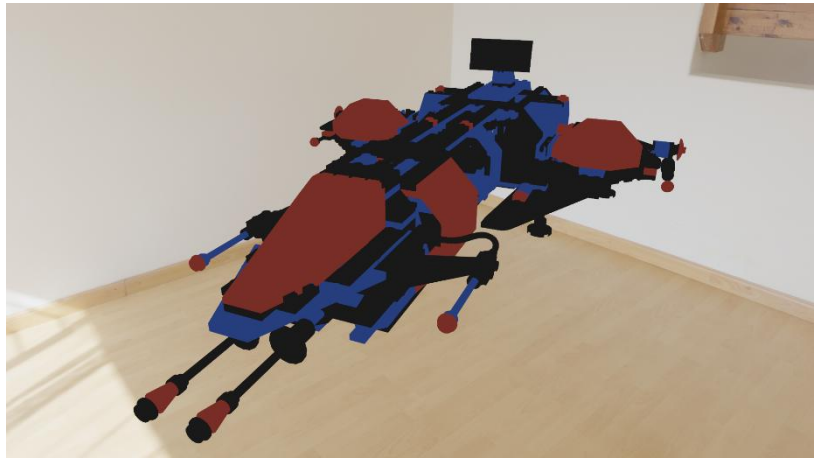
- Replace glDrawArrays and glDrawElements with glDrawArrays**Instanced** and glDrawElements**Instanced**
- These functions take one additional parameter - instance count
- Built-in variable in the vertex shader called **gl_InstanceID**
- This variable may be used to index into a large array of position values, e.g. arrays of uniforms, UBOs, or SSAOs
- Instanced arrays are an alternative approach for storing data that is unique per instance instead of gl_InstanceID

Image Denoising

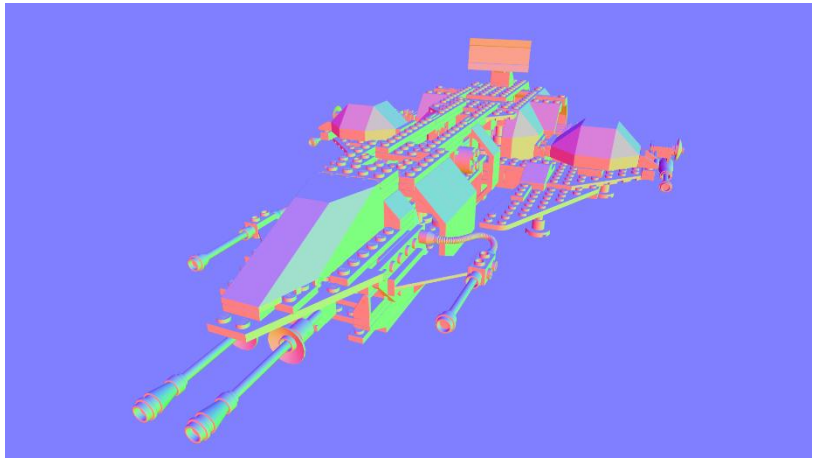
- Intel Open Image Denoise – high-performance denoising library for ray tracing
- https://openimagedenoise.github.io
- Works for a wide range of samples per pixel (1 spp to almost fully converged)
- See the simple example code snippet in the API documentation



LDR or HDR noisy color values in $\langle 0,1 \rangle$



First hit albedo values in $\langle 0,1 \rangle$ (antialiased)



World-space or view-space normals $\langle -\mathbf{1}, \mathbf{1} \rangle$ (antialiased)

Original 2 spp color image

Denoised 2 spp color image, Full HD

Denoised 2 spp color image + albedo, all Full HD


Denoised 2 spp color image + albedo + normal, all Full HD
Filter completed in 0.6 s on AMD Ryzen 3900X

Real-Time Polygonal-Light Shading

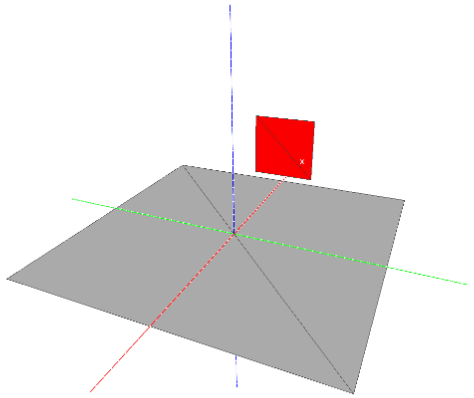
- Problem was that analytic solutions to polygonal lighting were limited to cosine-like distributions
  - Integrating a clamped cosine over a polygonal domain was solved by Lambert centuries ago with complexity $O(n)$.
  - Integrating Phong distributions was solved by Arvo in 1995 with complexity $O(n\ e)$.
  - Approximate solution to the Phong-polygon integration was devised by Lecocq in 2016 with complexity $O(n)$.
- Based on the Linearly Transformed Spherical Distributions (LTSDs), Heitz derived a new method called Linearly Transformed Cosines (LTC) for polygonal shading with physically based BRDFs with complexity $O(n)$

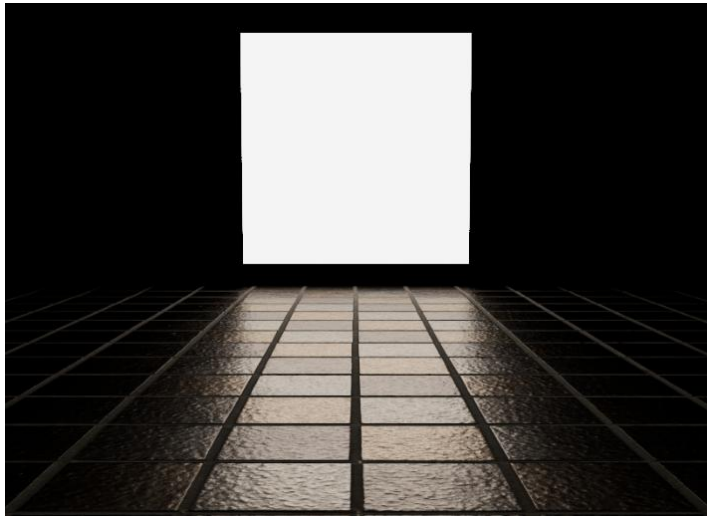
n ... number of polygon vertices
e … Phong exponent (shininess)

Sources:
LAMBERT, Johann Heinrich. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. sumptibus vidvae E. Klett, typis CP Detleffsen, 1760.
ARVO, James. Applications of irradiance tensors to the simulation of non-lambertian phenomena. In: *proc. of Computer graphics and interactive techniques*. 1995.
LECOCQ, Pascal, et al. Accurate analytic approximations for real-time specular area lighting. In: *proc. of the 20th ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games*. 2016.
HEITZ, Eric, et al. Real-time polygonal-light shading with linearly transformed cosines. ACM Transactions on Graphics (TOG), 2016.

Real-Time Polygonal-Light Shading

- Sample scene containing textured quad and rectangular light source





```
newmtl white_light
  Ke 10.0 10.0 10.0
  illum 1

newmtl slab_tiles
  Kd 1.0 1.0 1.0
  map_Kd ../textures/square_tiles/square_tiles_03_diff_4k.png
  Pr 1.0
  map_Pr ../textures/square_tiles/square_tiles_03_rough_4k.png
  map_Kn ../textures/square_tiles/square_tiles_03_nor_gl_4k.png
  Ke 0.0 0.0 0.0
  illum 5
```

Linearly Transformed Cosines

- Shading with diffuse polygonal light means evaluating the illumination integral I over a polygonal domain P

$$I = \int_P L(\omega_i) f_r(\omega_o, \omega_i) \cos(\theta_i)\, \mathrm{d}\omega_i \approx \int_P L(\omega_i) D(\omega_i) \mathrm{d}\omega_i = L \int_P D(\omega_i) \mathrm{d}\omega_i$$

$$= L \int_{P_o} D_o(\omega) \mathrm{d}\omega = L\, E(P_o)$$

The radiance of the polygonal light must be spatially constant

where computing the irradiance E is done by classic Lambert formula

$$E(\boldsymbol{p}_0, \ldots \boldsymbol{p}_{n-1}) = \frac{1}{2\pi} \sum_{i=0}^{n-1} \operatorname{acos}(\hat{\boldsymbol{p}}_i \cdot \hat{\boldsymbol{p}}_j) \left(\frac{\hat{\boldsymbol{p}}_i \times \hat{\boldsymbol{p}}_j}{\|\hat{\boldsymbol{p}}_i \times \hat{\boldsymbol{p}}_j\|} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right); j = (i+1) \,\%\, 2$$

assuming that the vertices of the polygonal light P are transformed as follows

$$P_o = M^{-1} P$$

Linearly Transformed Cosines

- The polygonal light P (originally represented by the set of vertices p_i^{WS} in world space) is transformed into a local frame such that the origin is located at the current hit point p (world coordinates of the fragment), the local x-axis ($\widehat{\boldsymbol{o}}_1$) is oriented in the direction of ω_o, and the local z-axis is surface normal ($\widehat{\boldsymbol{n}}$)

$$\boldsymbol{p}_i = LST \cdot \left(\boldsymbol{p}_i^{WS} - \boldsymbol{p}\right)$$

$$LST = \begin{bmatrix} \vdots & \vdots & \vdots \\ \widehat{\boldsymbol{o}}_1 & \widehat{\boldsymbol{o}}_2 & \widehat{\boldsymbol{n}} \\ \vdots & \vdots & \vdots \end{bmatrix}^{-1} \quad \text{and } \widehat{\boldsymbol{o}}_1 = \underbrace{\omega_o - \widehat{\boldsymbol{n}}(\omega_o \cdot \widehat{\boldsymbol{n}})}_{\text{Gram–Schmidt process}}, \widehat{\boldsymbol{o}}_2 = \widehat{\boldsymbol{n}} \times \widehat{\boldsymbol{o}}_1$$

- That is, the direction ω_o represented in the local frame is equal to $(\sin(\theta_o), 0, \cos(\theta_o))$. This is critical for the correct orientation of the polygon P with respect to the used parameterization of the matrix M, resp. M^{-1}.

Linearly Transformed Cosines

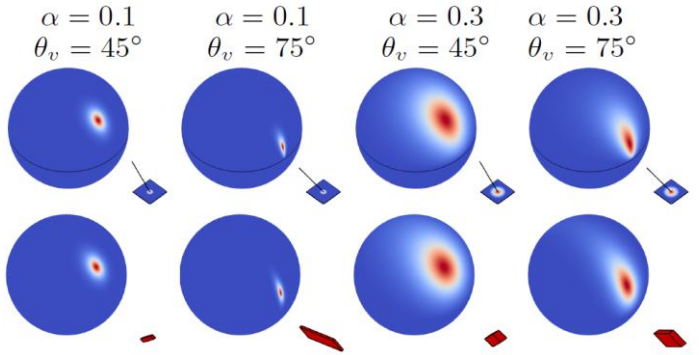
- For an isotropic material, the BRDF depends on the incident direction $\omega_o = (\sin(\theta_o), 0, \cos(\theta_o))$ and the parameter α (i.e. squared roughness)
- The matrix M comes from fitting the **cosine-weighted BRDF** to the desired BRDF for a pair of parameters $\cos(\theta_O)$ and α over the light directions ω_i

$$M(\cos(\theta_O), \alpha) = \begin{bmatrix} a & 0 & b \\ 0 & c & 0 \\ d & 0 & 1 \end{bmatrix}$$

Original distribution
D_0 – clamped cosine

Target distribution
D (e.g. T-R/GGX)

Fitted distribution



- Detailed derivation and additional materials including matrices M^{-1} with amplitudes can be found at https://eheitzresearch.wordpress.com/415-2/

An example of the equality for TR-GGX BRDF

$$\frac{DG}{4\cos(\theta_v)\,\cancel{\cos(\theta_l)}}\,\cancel{\cos(\theta_l)} = amplitude\,\frac{1}{\pi}\cos(\theta_o)\,\frac{|M^{-1}|}{\|M^{-1}\omega\|^3}$$

Linearly Transformed Cosines

- The computation of the target distribution D from the base distribution D_o (both are cos-weighted) with knowledge of M^{-1} for a particular $\cos(\theta_o)$ and α is as follows (note that $\omega_o = \frac{M^{-1}\omega}{\|M^{-1}\omega\|}$)

The Jacobian of the spherical transformation measures the change of solid angle

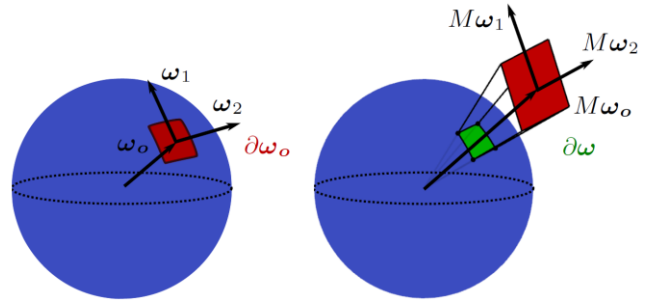
$$D(\omega) = D_o(\omega_o)\boxed{\frac{\partial\omega_o}{\partial\omega}} = D_o\left(\frac{M^{-1}\omega}{\|M^{-1}\omega\|}\right)\boxed{\frac{|M^{-1}|}{\|M^{-1}\omega\|^3}}$$

where the base distribution reads

$$D_o(\omega_o) = \frac{1}{\pi}\cos(\theta_o)$$

and ω stands here for a viewing direction

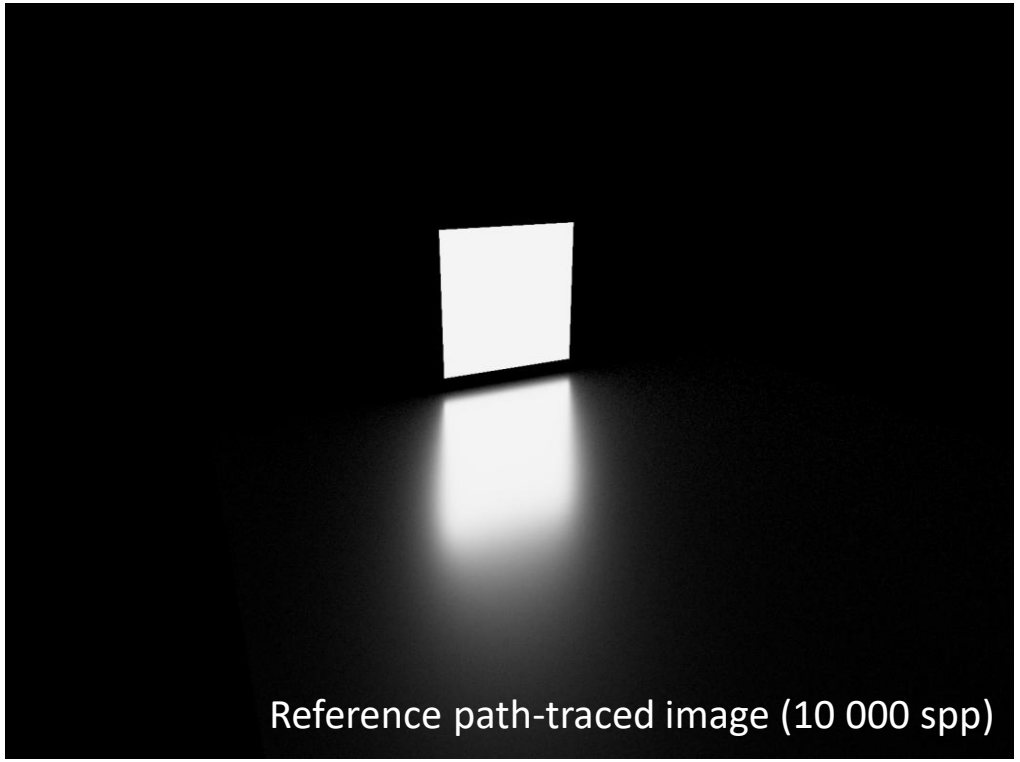
Do not confuse D with mere NDF here

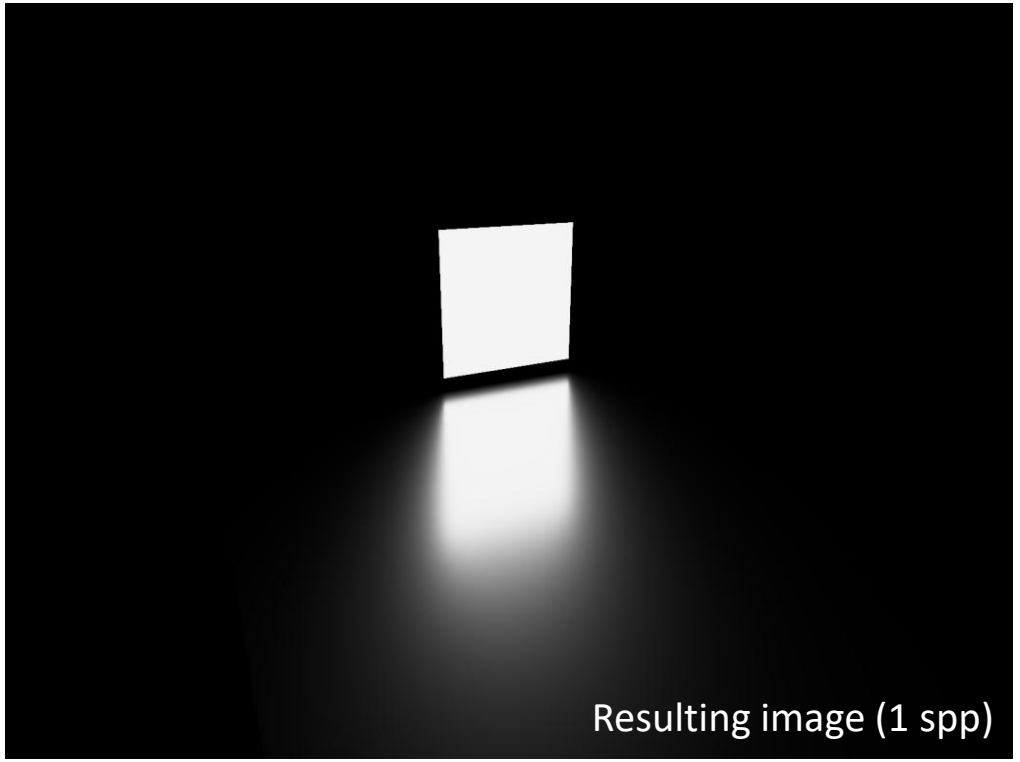


$$\partial\omega = \partial\omega_o A\frac{\cos(\theta)}{r^2}$$

Linearly Transformed Cosines

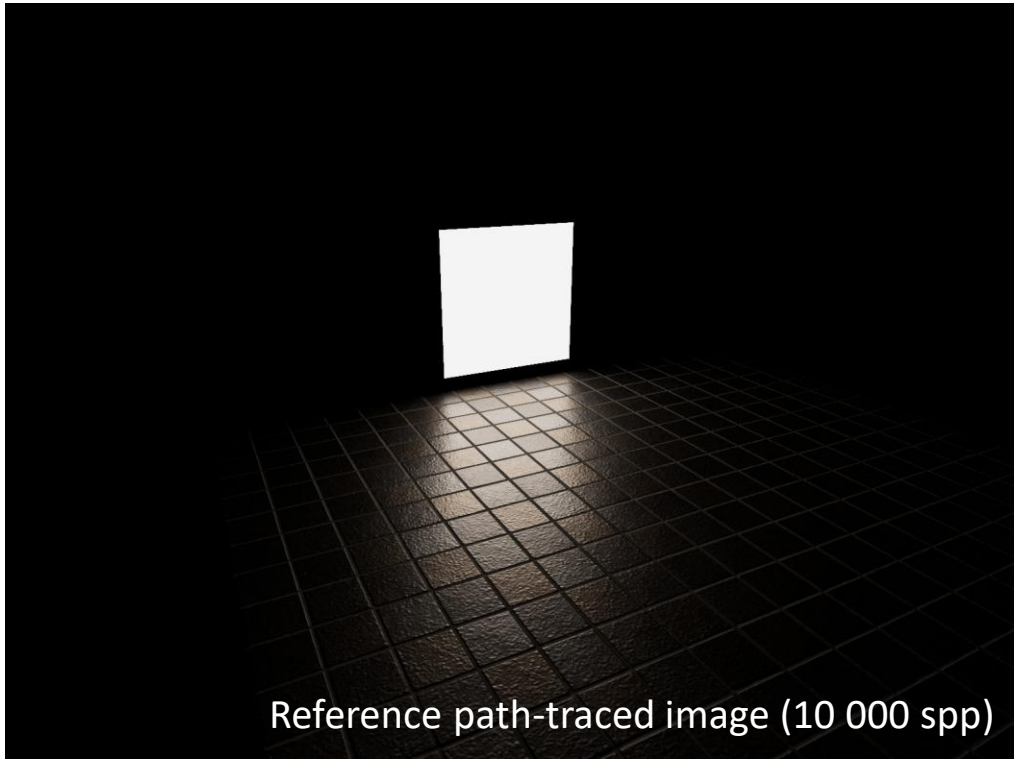
- Test scene: BRDF TR-GGX, roughness 0.2, $L_e = (10, 10, 10)$, TMO Reinhard

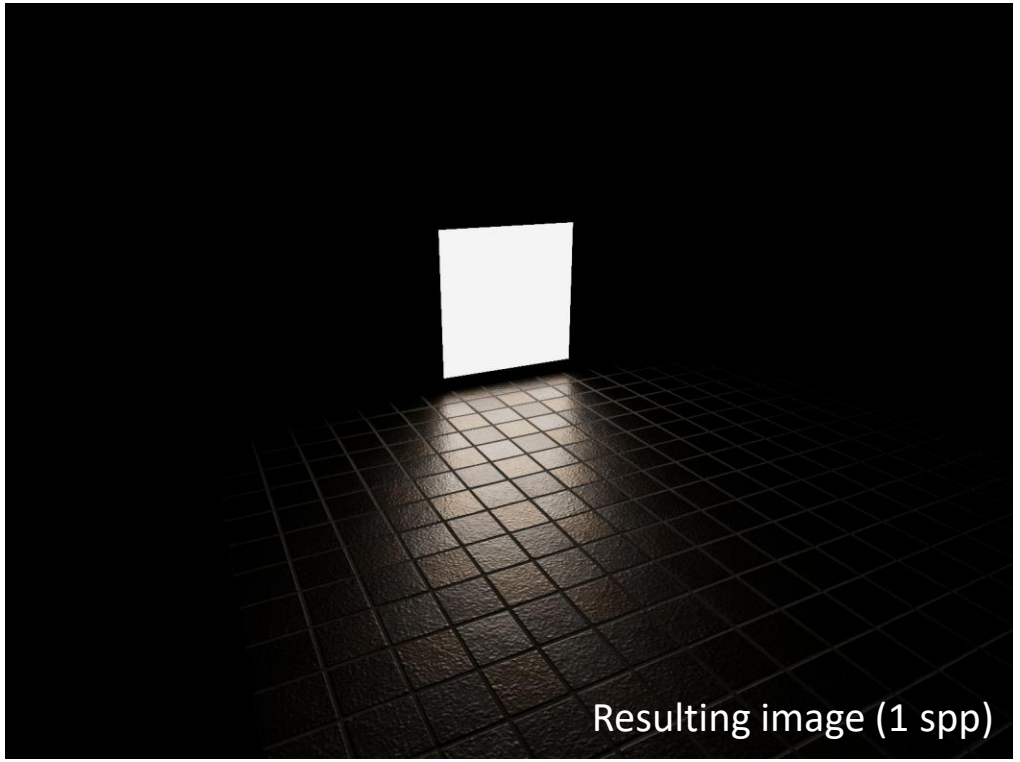




Linearly Transformed Cosines

- Test scene: BRDF TR-GGX, roughness by texture, $L_e = (10, 10, 10)$, TMO Reinhard
  Note that only the specular component is considered here and Fresnel is not used ($F = \text{base color}$)


Reference path-traced image (10 000 spp)


Resulting image (1 spp)