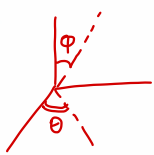
Real time Computer Graphics

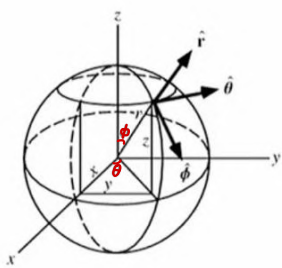
Polygon - the interior of a closed planar connected series of line segments. The edges do not cross each other and exactly two edges meet at every vertex

$x^2+y^2+z^2=R^2$

- Example: a procedural unit sphere

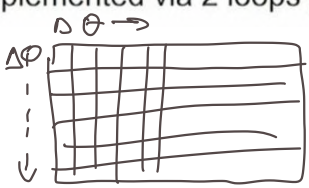
$\phi: [0, \pi]$ Zenith (along longitude)
$\theta: [0, 2\pi]$ Azimuth (along latitude)

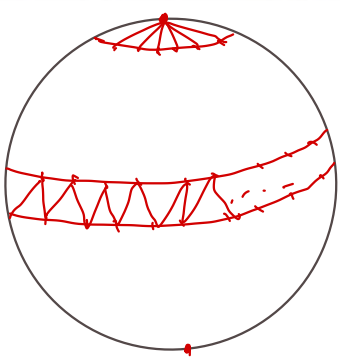




$$x(\theta, \phi) = \cos(\theta) \cdot \sin(\phi)$$
$$y(\theta, \phi) = \sin(\theta) \cdot \sin(\phi)$$
$$z(\theta, \phi) = \cos(\phi)$$

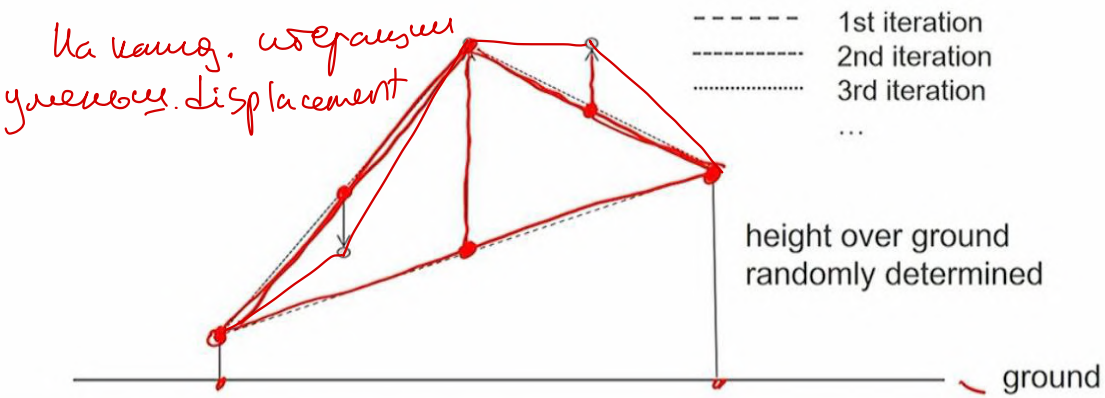
Implemented via 2 loops generating pairs θ, ϕ, and for each pair computing x,y,z



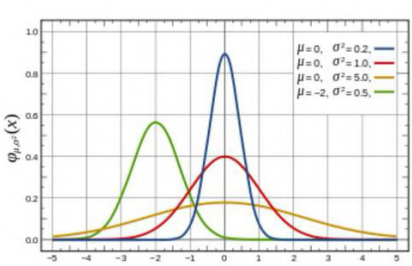


Procedural modelling

- Terrain modelling via random midpoint displacements
  - Interpolation and random displacements D_i (i: iteration)
  - Ever smaller displacements in each iteration



- Random displacements
  - Modelled via random numbers with a certain distribution; distribution defines the „probabilities“ of the occurrence of values of a random variable
  - E.g., uniform [0,1] distribution: all values in [0,1] occur equally likely
  - E.g., normal distribution with expectation value μ and standard deviation σ, i.e., $N(\mu, \sigma^2)$
    - μ: the average of all possible values
    - σ: the variation (spread) from the expectation value
      68.2% of values fall between $\mu \pm \sigma$



Often used is the normal distribution $N(\mu = 0, \sigma^2)$

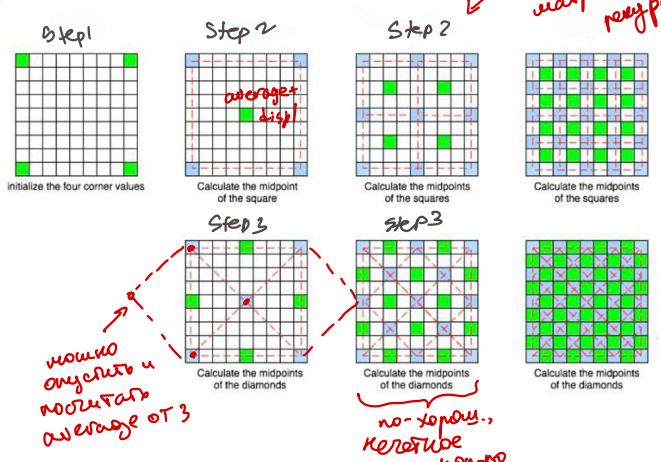
меняем сигму с каждой итерацией.

- Random displacements D_i (i: refinement iteration)
  - D_i are modelled via normal distributed random numbers with expectation value $\mu[D_i] = 0$ and standard deviation $\sigma[D_i] = \sigma_i$
  - Starting with σ_0 at the first level, we want to generate ever smaller random displacements at subsequent iterations
  - Modelled via standard deviation $\sigma_i = \sigma[D_i] = \frac{\sigma_{i-1}}{2^H}$
    H controls fall-off and thus roughness of terrain
  - Computation from $N(0,1)$ distributed random numbers via $N(0,\sigma^2) = \sigma N(0,1)$
  - **Java**: **Class Random**
    double nextGaussian()
    Returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.

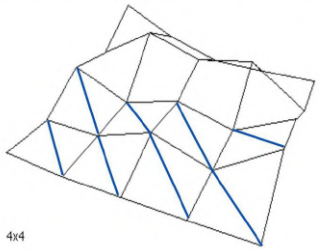
$\sigma_i = \frac{\sigma_{i-1}}{2^H}$

- Diamond-Square-Algorithm
  1. Assign (**random**) values to the corners $N(0, x^2)$
  2. (**Diamond**) Assign the average of four corners plus a random displacement to the midpoint
  3. (**Square**) For each diamond, average four corners and add a random displacement
  4. Repeat step 2 and 3 for a given number of iterations

можно отдельно по осям матрицы! не рекурсивно



- From the 2D height raster to triangles; connect each adjacent block of 2x2 vertices to form 2 triangles:



← дальше просто соединяем диагонали

Polygon meshes – полигональные сетки (из обыч. треугольников)

Valence of a vertex – кол-во выходящих (adjacent) ребер

↑ "валентность"

← тут у v valence 5

How to store triangle mesh:

- Geometry – где расп. вершины в 3D пространстве
- Topology – как вершины и ребра соединены
- Explicit representation – хранить 3 верш. для каждого треугольника

3 верш. · 3 коорд. · F = кол-во floatов

	v_i	v_j	v_k
0			
1			
2			

– high redundancy

Triangle mesh representation – less redundant

- shared vertex ("indexed face set")
- **array of coordinates** of all vertices (3V floats + 3V uchars)

v	x	y	z	c
0	...	...	...	RGB ← color
1	...	...	...	

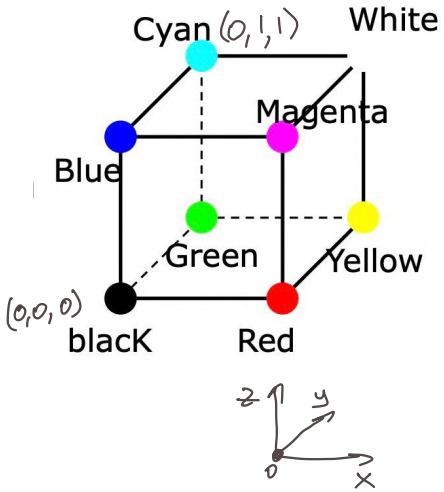
- **array of triangles** with indexes into the vertex list (3F int)

t	i	j	k
0	0	1	4
1	0	2	6

Appearance Modeling

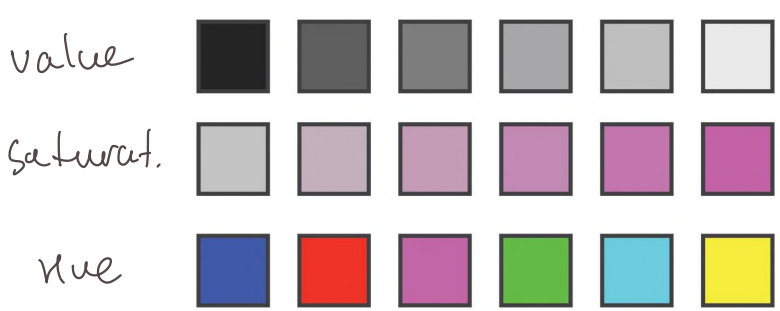
(R, G, B) - (x, y, z) mapping.

CMY (K) (K = Black)
- mix to white in additive model
- mix to black in subtractive model



HSV color model

Hue - dominant wavelength (оттенок)
Saturation - pureness, inverse proport. to amount of white
Value (brightness) - intensity of light



YUV

Y - luminance (brightness) value Y
UV - 2 chrominance (hues) values UV

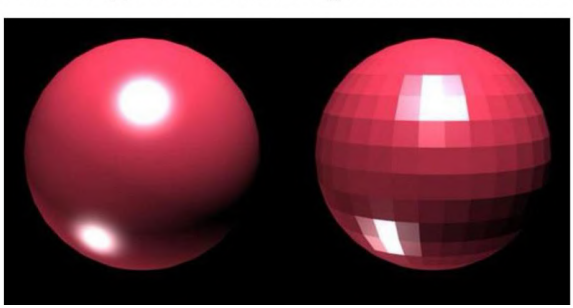
- Used in JPEG image compression:
  Human eye is less sensitive to chrominance than luminance, and by separating chrominance it can be compressed more aggressively

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.14713 & -0.28887 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

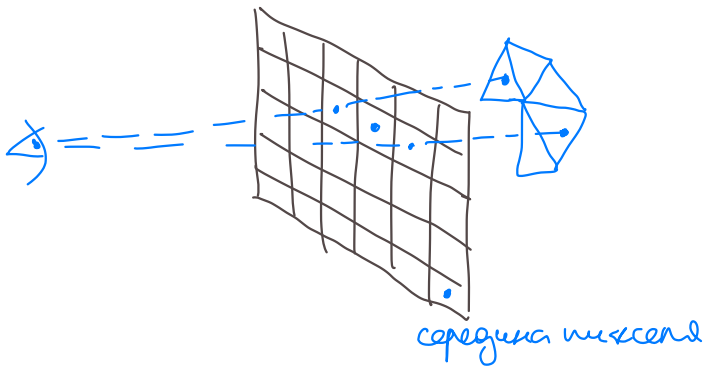
Shading (rendering) - filling triangles with color from vertices

- Colors are assigned to polygon vertices in the modeling stage as additional properties
  - Color can be interpolated across triangles for a smooth appearance
  - Non-smooth appearance when assigned as constant face color

Smooth color shading



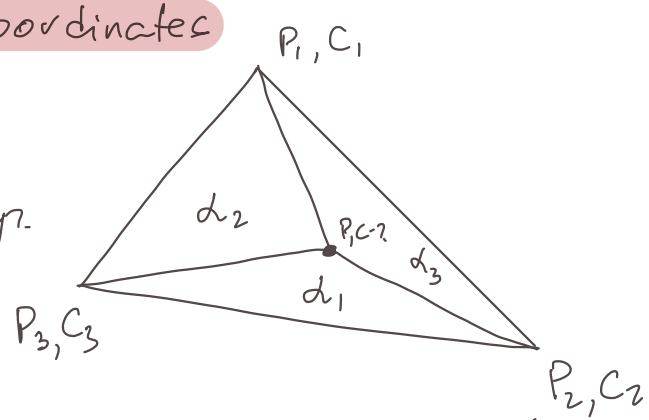
Flat shading - constant color across the triangles



Choose a point seen through the pixels centers and assign their color to the pixels

Color interpolation

Barycentric coordinates



α_n - проекта от площади триг.

$P = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3$ и $\alpha_1 + \alpha_2 + \alpha_3 = 1$

Then:

$$C = \alpha_1 C_1 + \alpha_2 C_2 + \alpha_3 C_3$$

Как найти α - ?

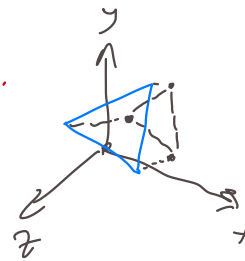
$$\begin{cases} P_x = \alpha_1 P_{x1} + \alpha_2 P_{x2} + \alpha_3 P_{x3} \\ P_y = \alpha_1 P_{y1} + \alpha_2 P_{y2} + \alpha_3 P_{y3} \\ P_z = \alpha_1 P_{z1} + \alpha_2 P_{z2} + \alpha_3 P_{z3} \\ \alpha_1 + \alpha_2 + \alpha_3 = 1 \end{cases}$$

- три переменных, но 4 уравн.

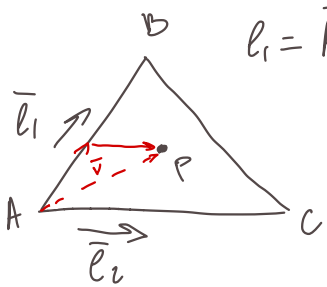
Done for all 3 color comp. RGB

Inside triangle criteria

$$\begin{cases} 0 \le \alpha_1 \le 1 \\ 0 \le \alpha_2 \le 1 \\ 0 \le \alpha_3 \le 1 \end{cases}$$



Возьмем проекцию треугольника на x, y и получим 3 уравнения вместо



$\bar{l}_1 = \bar{AB} = (x_2 - x_1, y_2 - y_1 \dots)$

$A + u\bar{l}_1 + v\bar{l}_2 = P$

$A + u(B-A) + v(C-A) = P$

$A + uB - uA + vC - vA = P$

$(1-u-v)A + uB + vC = P$

w

$u\bar{l}_1 + v\bar{l}_2 = \bar{v} \quad | \cdot \bar{l}_1$

$\begin{cases} (u\bar{l}_1 + v\bar{l}_2)\bar{l}_1 = \bar{v}\bar{l}_1 \\ (u\bar{l}_1 + v\bar{l}_2)\bar{l}_2 = \bar{v}\bar{l}_2 \end{cases}$ $\downarrow l_2$

$\bar{v}\bar{l}_x$ - scalar product

$\bar{l}_1\bar{l}_1 = |\bar{l}_1||\bar{l}_1| \cdot \cos 0 = 1$

$\begin{cases} u\bar{l}_1\bar{l}_1 + v\bar{l}_2\bar{l}_1 = \bar{v}\bar{l}_1 \\ u\bar{l}_1\bar{l}_2 + v\bar{l}_2\bar{l}_2 = \bar{v}\bar{l}_2 \end{cases}$

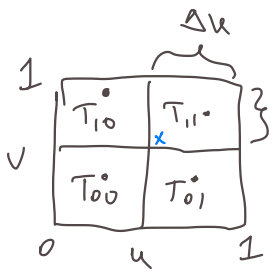
Находим u и v

$w = 1 - u - v$

Texture Mapping

texel - entry in texture mapping (texture element)

Texture map - discretely sampled multi-dimens. function containing material properties, like color



$\frac{1}{T_u}, \frac{1}{T_v}$ - сколько занимает места

$T_{ij} = \left(\frac{\Delta u}{2} + i\Delta u, \frac{\Delta v}{2} + j\Delta v\right)$

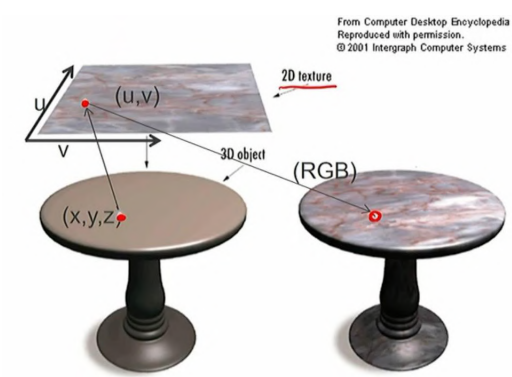
texel center

Δu - width of texels
Δv - height of texels

for x - T_{11} is nearest neighbour

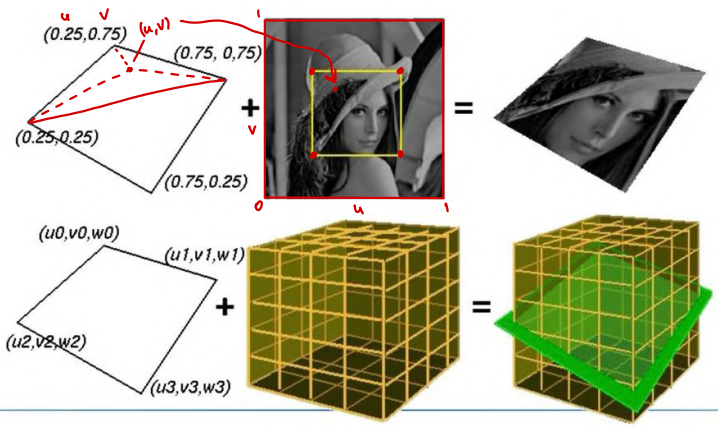
3d obj → texture map

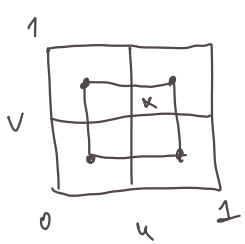
$(x, y, z) \to (u, v) \to RGB$



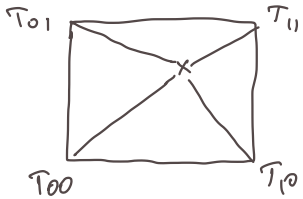
Per-vertex texture coordinates

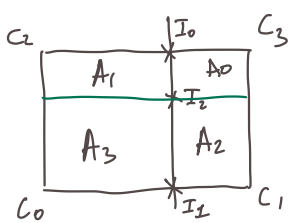
- For a polygon, the mapping is specified via per-vertex texture coordinates; they define the texture part that is mapped to the polygon

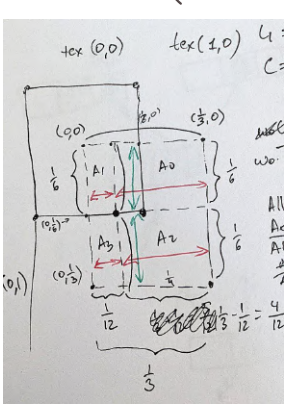




val = tex[0.7, 0.2]







bilinear interpolation

interpolate I_0, then I_1, then I_2

$$b(u,v) = w_0 C_0 + w_1 C_1 + w_2 C_2 + w_3 C_3$$

Lightning

What needs to be specified to lit (illuminate) a model:
- Light source properties
- Appearance properties like reflection, absorption

The received light depends on the orientation of the surface to the light source

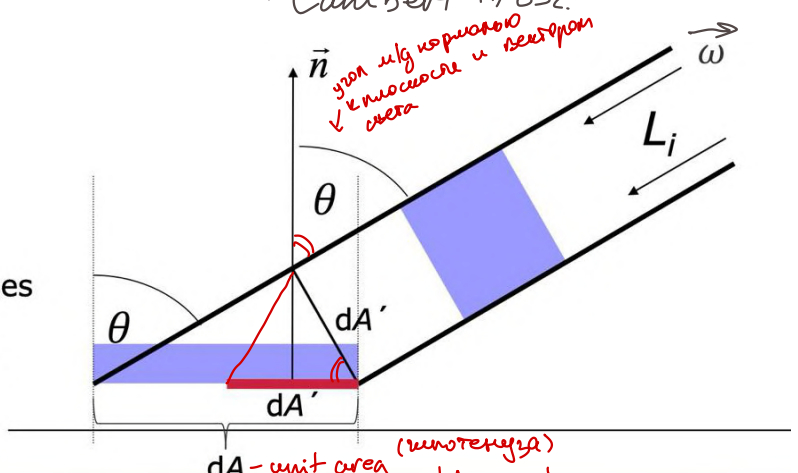
Surface Orientation

Power per unit area (dE) arriving at a surface point x depends on the angle of the surface to the light direction

← Lambert 1783г.

Effectively lit area: $dA = dA' / \cos\theta$

Power distributed over a larger area, power per dA' decreases



light power ↓

$$dE = \cos(\theta) \cdot L_i(\vec{\omega}) d\vec{\omega}$$
$$= (\vec{n} \cdot \vec{\omega}) \cdot L_i(\vec{\omega}) d\vec{\omega}$$

$|\vec{n}| \cdot |\vec{\omega}| \cdot \cos\theta$
1 1

(косинусная)

$dA = dA' / \cos\theta$

$\frac{dA}{dA'} = \cos\theta$ – Ламберт

cos tells us how much one vector points in the same direction as the other

Tangent plane - касательная поверхность

normal vector is always normalized to 1

We also store it:

v	x	y	z	C	T (Texture)	N
0	1.0	1.0	1.0	C_0	tC_0	n_0
1	-1.0	..	...	C_1	tC_1	n_1
2	...	...	...	C_2	tC_2	n_2

t	i	j	k
0	0	1	1
1	0	2	3
2	3	2	1

Light reflection

отражается равномерно во все стороны

не зеркало

Diffuse (Lambertian) reflection at rough material

- Scatters equally likely in any direction, regardless of incoming direction

intensity of the reflected light at point x in direction $\vec{\omega_r}$

$$I_r(x, \omega_r) = k_d \cdot (\vec{n} \cdot \vec{l}) \cdot I_i(x, \omega_i)$$

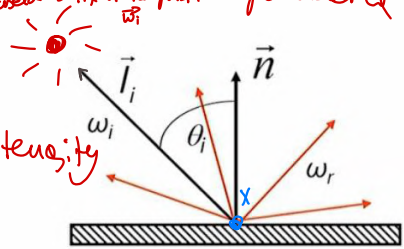
$\vec{n} \cdot \vec{l}$ = $\cos\theta$

Чем меньше $\cos\theta$, тем меньше света

diffuse - рассеянный

I_i = intensity ragious. света в т.x от направл. $\vec{\omega_i}$

- k_d = diffuse material constant $\in (0,1)$
- I = light intensity



I - incident light (падающий интенсивность света)

Ir - reflected light intensity

На самом деле диффузор не от солнца а к солнцу всегда

Light reflection

чем сильнее полируем, тем зеркальнее сферический

отражающий

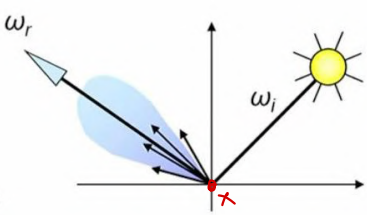
Specular (glossy) reflection at smooth material

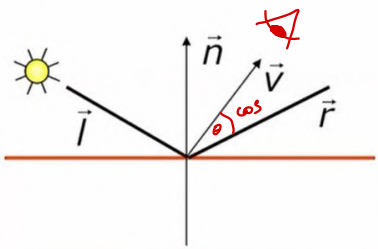
- Light is mostly reflected into the directions around the mirror direction ω_r of ω_i

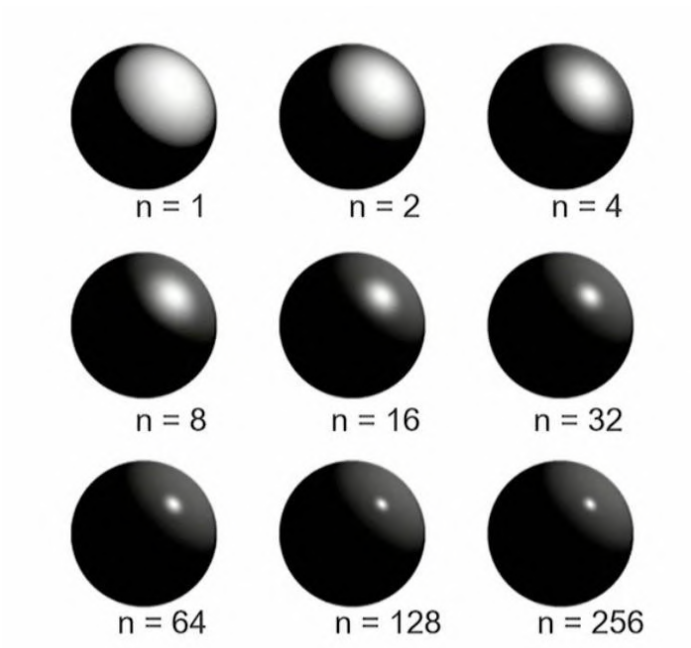
падающий incident light intensity

$$I_r(x, \omega_r) = k_s(\vec{v} \cdot \vec{r})^n I_i(x, \omega_l)$$

- k_s = specular material constant $\in (0,1)$ – сколько света зеркально отраж.
- n = specular exponent, controls extend of highlights





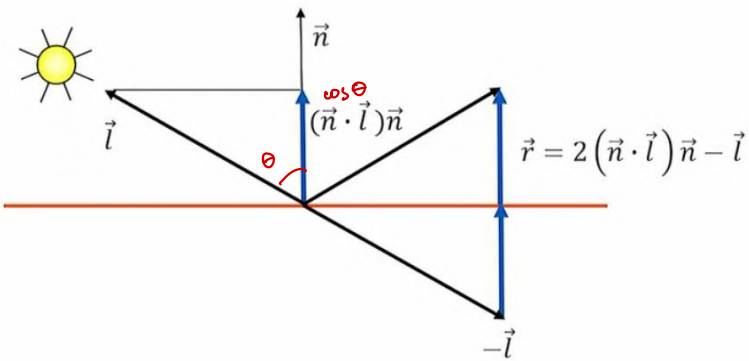


same intensity

changing exponent (and material)

Computing the reflection vector $\vec{r}$ of the incoming light

- $\vec{l}$ is vector from point to light source
- Assuming $\vec{n}$ and $\vec{l}$ are unit length vectors

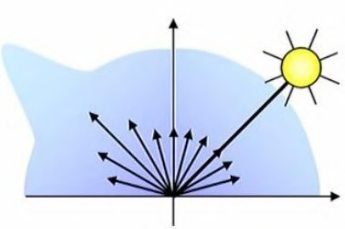


Phong (Phong Bui-Tuong 1975) lighting model

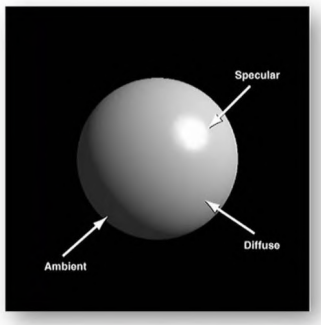
Combines diffuse, specular and ambient terms to model all effects

$$I_r(x,\omega_r) = \begin{array}{c} k_d\left(\vec{l}\cdot\vec{n}\right) I_i(x,\omega_l) + \\ k_s(\vec{r}\cdot\vec{v})^n I_i(x,\omega_l) + \\ k_a I_a \end{array}$$

Ambient term models constant background light

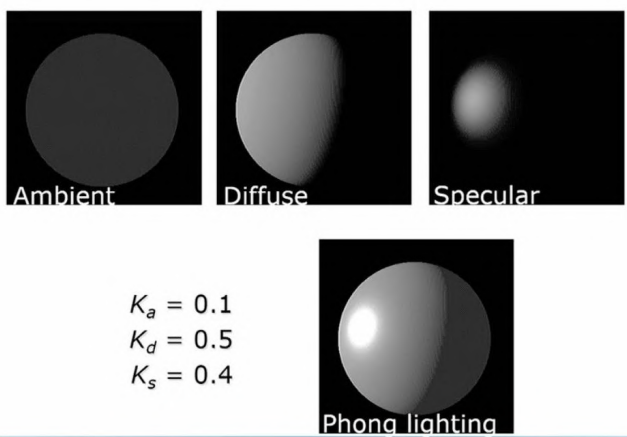


Combines diffuse, specular and ambient terms to model all effects



Phong lighting model

Но эта модель физически неверная!



Вот почти норм:

- With surface color C_S, light source color C_L, and color of background ambient light C_A (окружающий) (На самом деле задаем RGB)

$$C_r(x, \omega_v) = \sum_{\text{sum over all light sources}} k_d(\vec{l} \cdot \vec{n})\, C_S C_L + k_s(\vec{r} \cdot \vec{v})^n C_L + k_a C_S C_A$$

component-wise

(скорее для пластика, не для металлика)

отражение шероход. поверхн. | отражение | background!

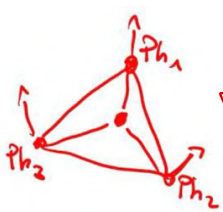
- Note that surface color usually only interacts with diffusely reflected and background light
- Specular highlights have light source color

We do not consider shadows!

Problems …

- Phong is not physically based and can produce materials that are physically impossible – reflecting more light than arrives at them, for instance
- In recent years, PBR – physically based rendering – has become the de-facto standard for lighting models
- Much more complicated formulation of the individual components – for example, specular and diffuse are now tied together via roughness

Rendering

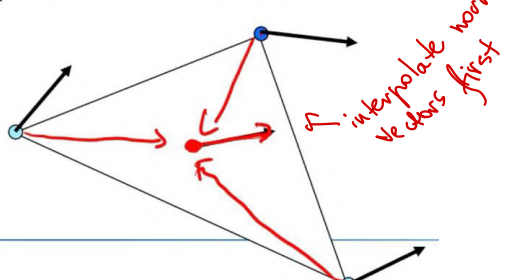


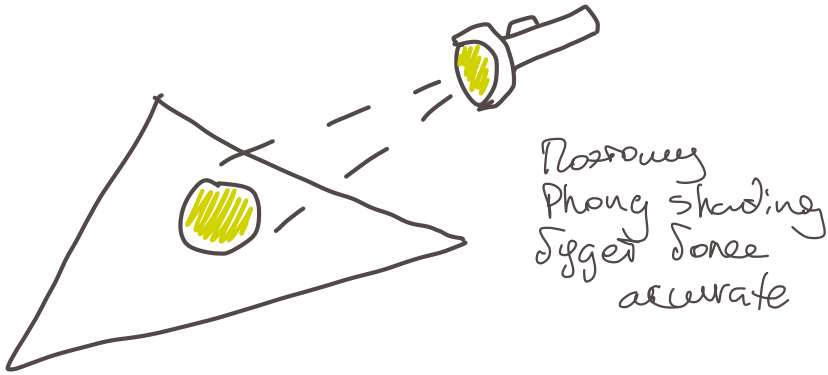
- How to apply Phong illumination

Eval. Phong model 3 times & then interpolate

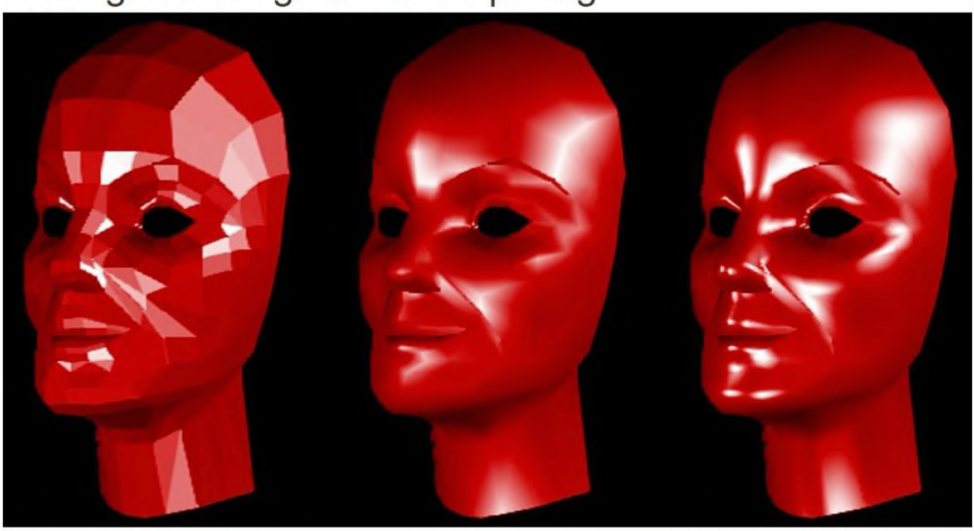
- First approach: compute reflected color per vertex, and interpolate resulting color smoothly - Gouraud shading
- Second approach: Phong shading (1975)
  - Higher quality shading by interpolating per-vertex normals and evaluating for every surface point the illumination model

Evaluate Phong model for every x inside the triangle





Shading: flat vs. gouraud vs. phong



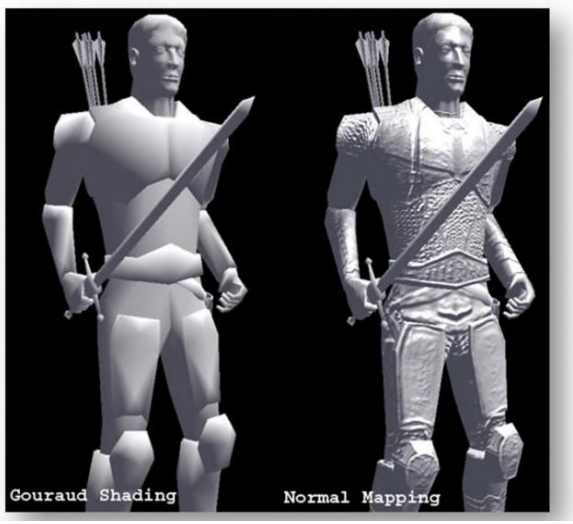
Per-vertex Phong Lighting
Flat Shading

Per-vertex Phong Lighting
Gouraud Colour Shading

Per-Fragment Phong Lighting
Phong Shading

Normal mapping

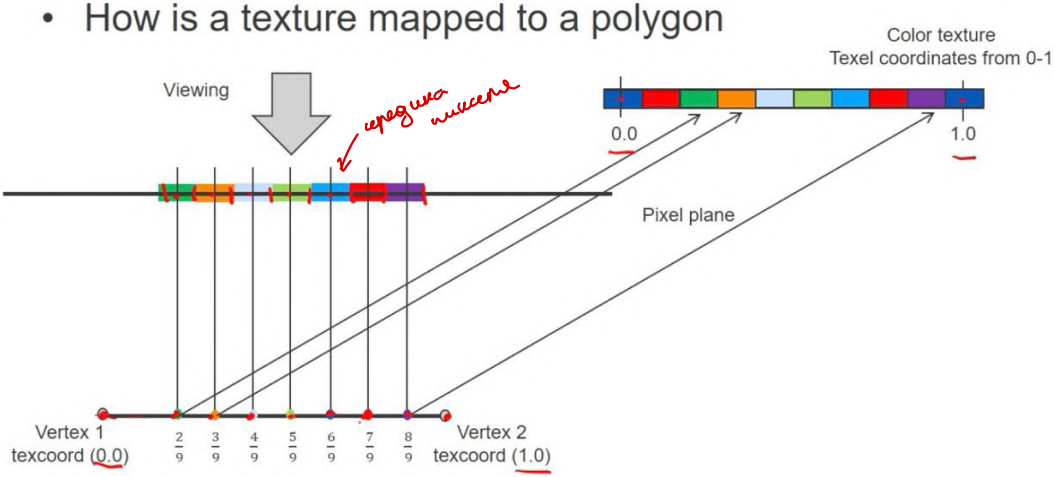
- Motivation: we always try to keep the number of rendered polygons low – less work in the rendering pipeline
- However, we often want to use highly detailed models because they look better



- How can we achieve the appearance of geometric detail without using more geometry?

Texture mapping

- How is a texture mapped to a polygon

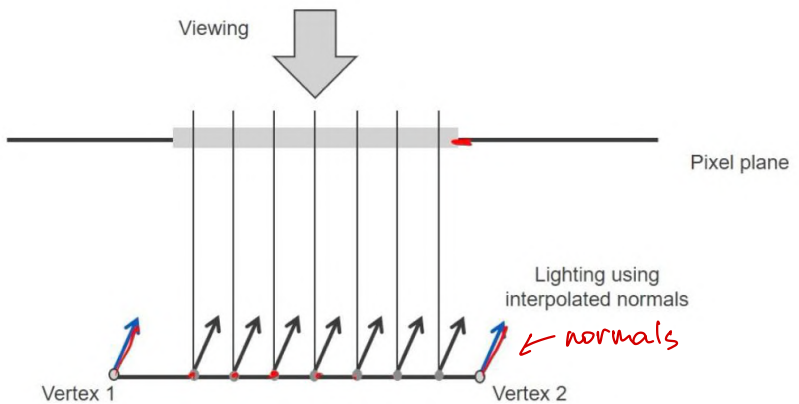


Interpolation of per-vertex texture coordinates
Texture lookup

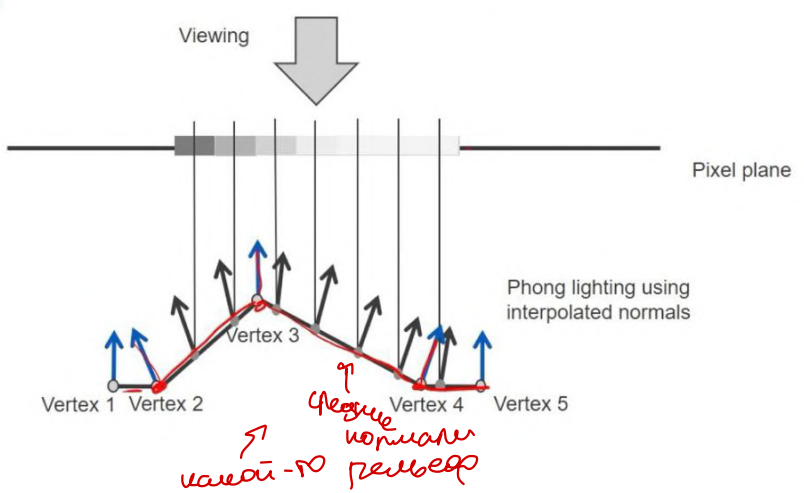
Normal mapping

Lighting and shading

- How is a polygon illuminated



- Bumps can be simulated by adding geometric detail
  - → this results in varying normals
  - → this results in varying brightness when triangles are illuminated



Graphics pipeline

Rasterization based graphics pipeline:

1) User / Driver Stage

2) Vertex / Geometry Stage

transformations

Vertex Stream

Rasterizer

Coverage / Interpolation

3) Pixel Stage

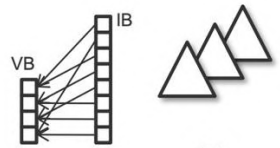
Fragment Stream

Texturing

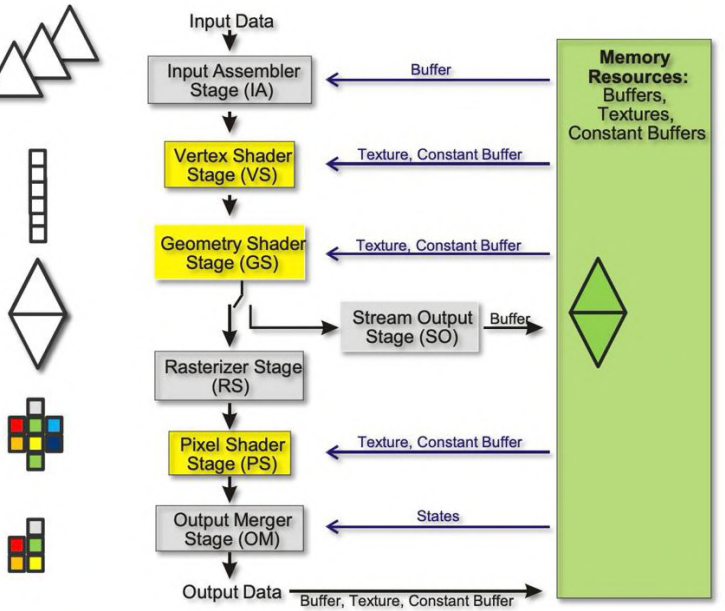
Blending

→ Show

DirectX 10 class hardware pipeline

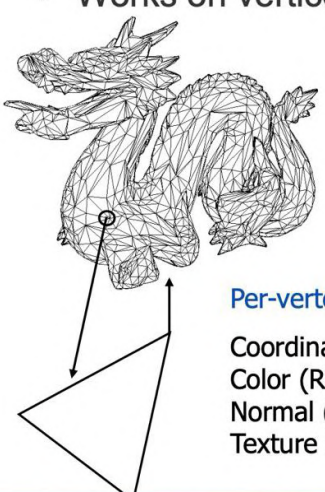


Geometry is stored in vertex, index and attribute buffers (see chapter modelling. shared vertex rep.)



Geometry processing

- Works on vertices; performed in the vertex shader stage

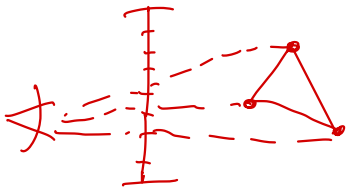


Per-vertex attributes:

Coordinate (x,y,z,1)
Color (RGBα) α - непрозрачность (opacity)
Normal (n_x,n_y,n_z)
Texture coordinate (u,v)

$$\left(\begin{array}{ccc|c} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \hline (. & . & .) & 1/0 \end{array}\right)$$

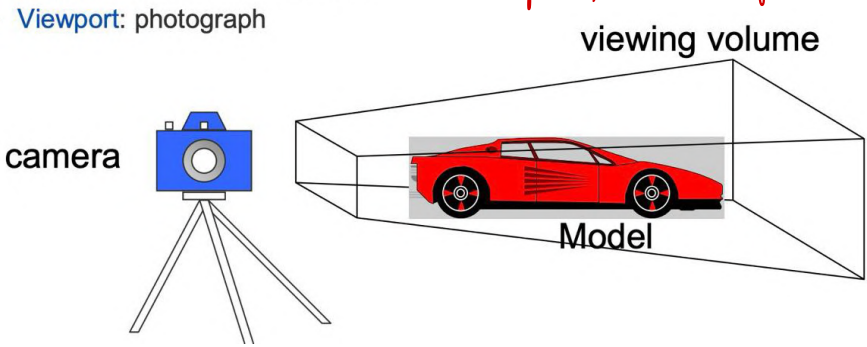
A set of transformation matrices, typically issued via the application program

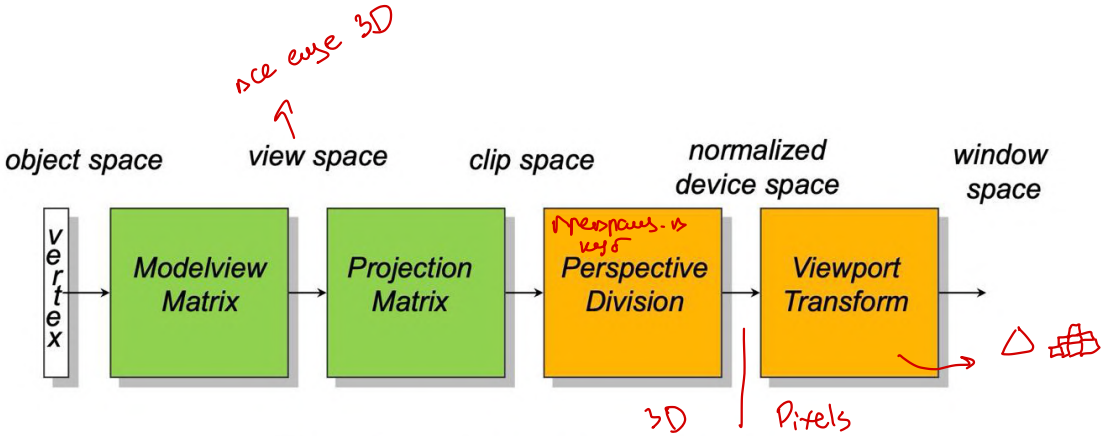


Perspective projection

- Per vertex operations
  - Coordinate/normal transformations
  - Perspective projection of vertices
- Per triangle operations (optional in the geometry shader)
  - Front/back face culling
    - Do not show triangles having a normal that points away from the viewer; can reduce the number of rendered polygons
  - Clipping
    - Remove triangles that are outside the visible space
- Result is projected triangle vertices (in homogeneous coordinates) with attributes

- Transformation – camera analogy
  - Modeling: scale, rotate, translate the model
  - Viewing: position and orientation of the camera
  - Projection: adjust camera lens
  - Viewport: photograph



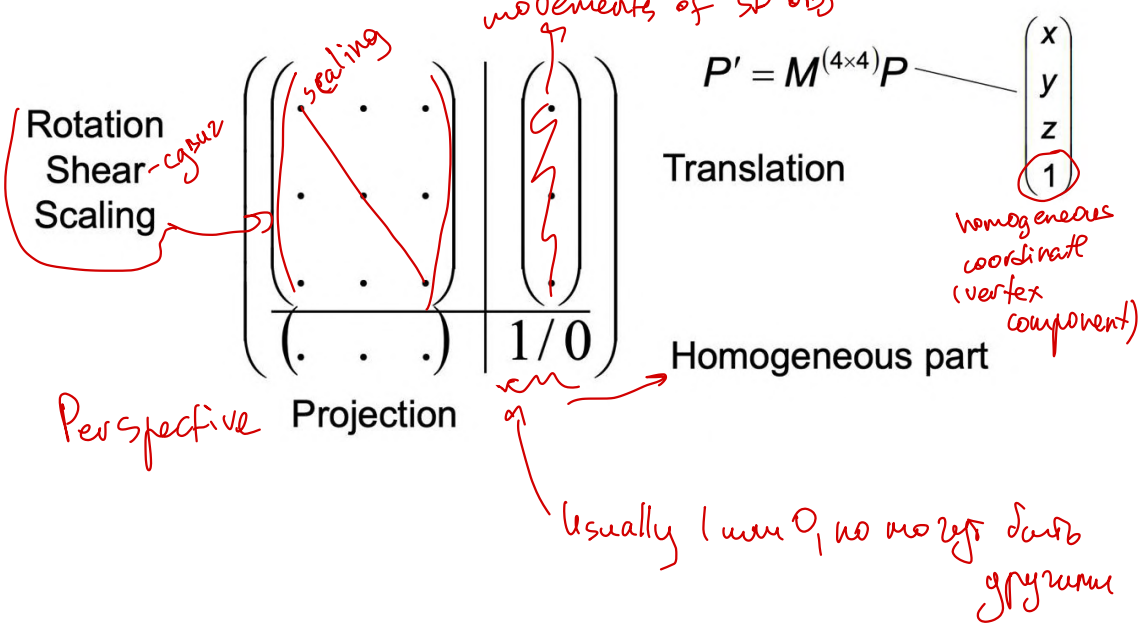


The transformation pipeline

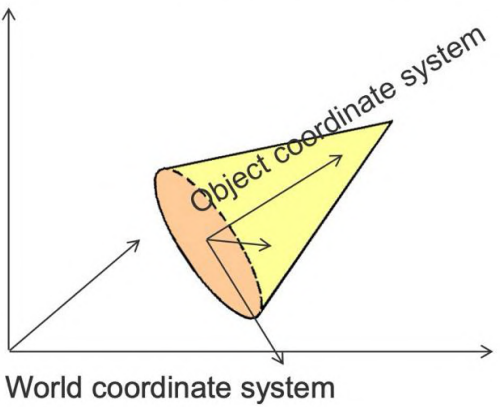
Green: in the vertex shader stage
Orange: in the rasterizer stage

3D Modeling Transformations

- Transformations
  - Transformations in homogeneous coordinates



- What we render is a portion of the global world coordinate space
  - Objects are placed in this space via the modelling transform M_{mod}



Transformations

- A transformation maps points (x,y,z) in a coordinate system to points in the same coordinate system
- Here, we will consider transformations of the form

$$x' = ax + by + cz + d$$
$$y' = ex + fy + gz + h$$
$$z' = ix + jy + kz + l$$

translation - перемещение

which map points (x,y,z) in Cartesian coordinates to points (x', y', z')

- Extended (homogeneous) coordinates (*x*,*y*,*z*,*w*) to capture translations in one transformation matrix

(standard) formulation

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a & b & c \\ e & f & g \\ i & j & k \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} d \\ h \\ l \end{pmatrix}$$

$$p' = Mp + t$$

Homogeneous formulation

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

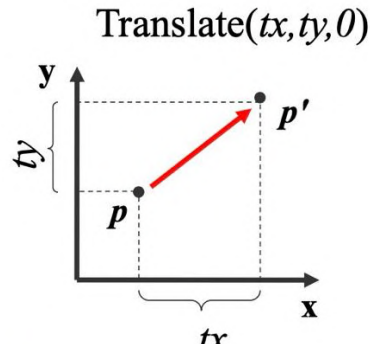
$$p' = Mp$$

у вектора нормали тут 0

Потому что d,h,l — перемещают объект в пространстве

Transformations

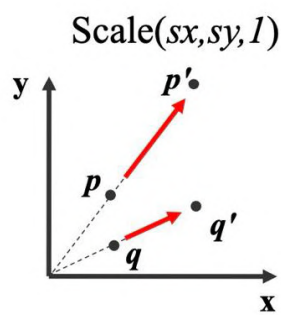
- Translation



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Transformations

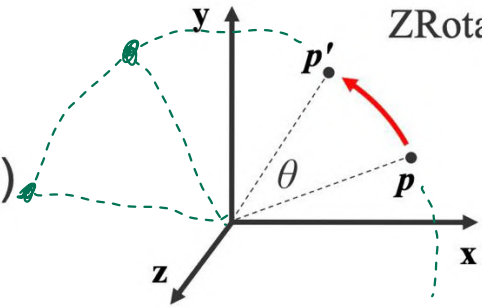
- Scaling



$$\begin{pmatrix} x' \\ y' \\ z' \\ 0/1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0/1 \end{pmatrix}$$

Transformations

- Rotation (about z-axis)



не крутим по z, а крутим вокруг оси z, ↓ получается

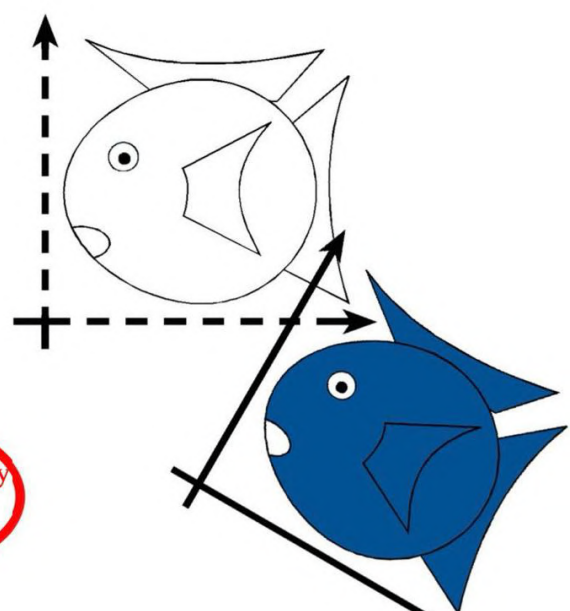
$$\begin{pmatrix} x' \\ y' \\ z' \\ 0/1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0/1 \end{pmatrix}$$

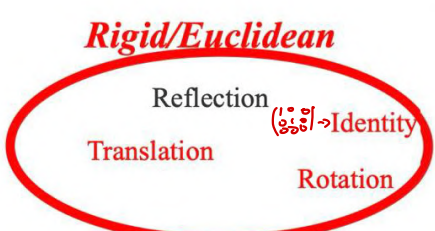
Тогда все перемещ. по этому кругу и есть эти sin, cos

Transformation classes

медленный, твердый

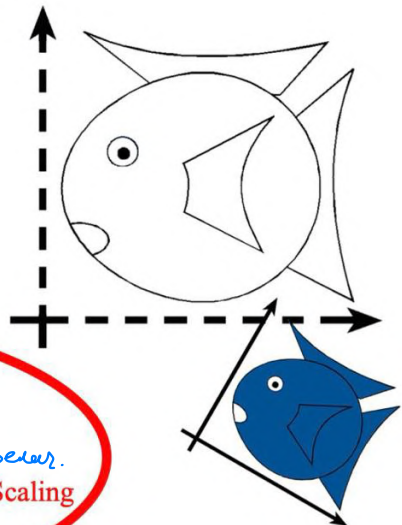
- Rigid body transforms
  - Preserve distances
  - Preserve angles

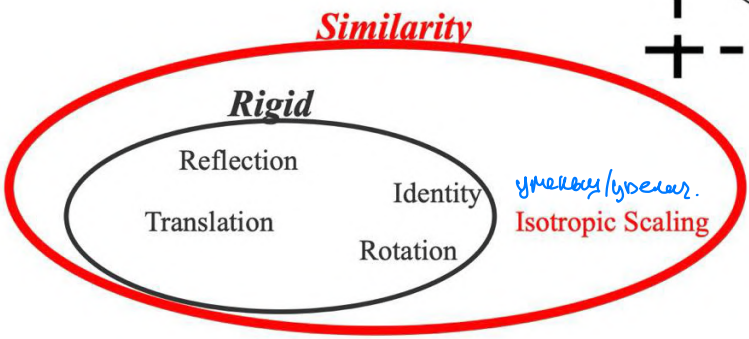




Transformation classes

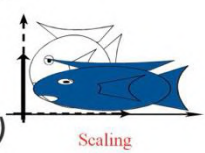
- Similarity transforms
  - Preserve angles

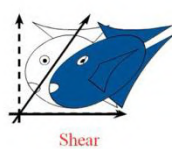


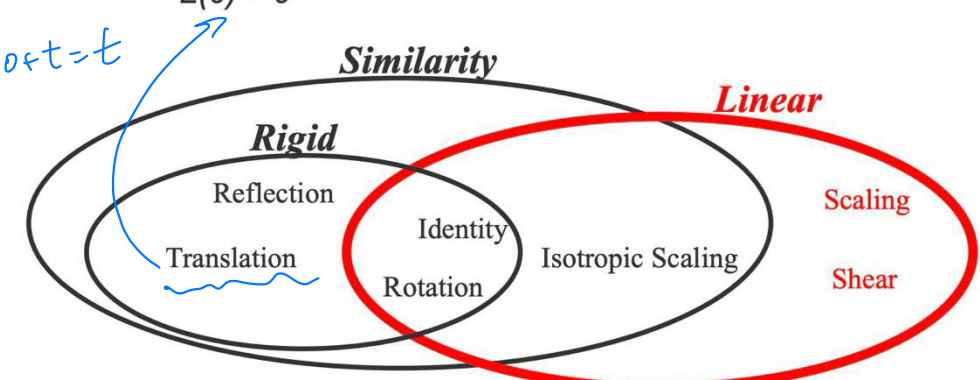


Transformation classes

- Linear transforms
  - $L(p + q) = L(p) + L(q)$
  - $L(ap) = a\,L(p)$
  - $L(0) = 0$



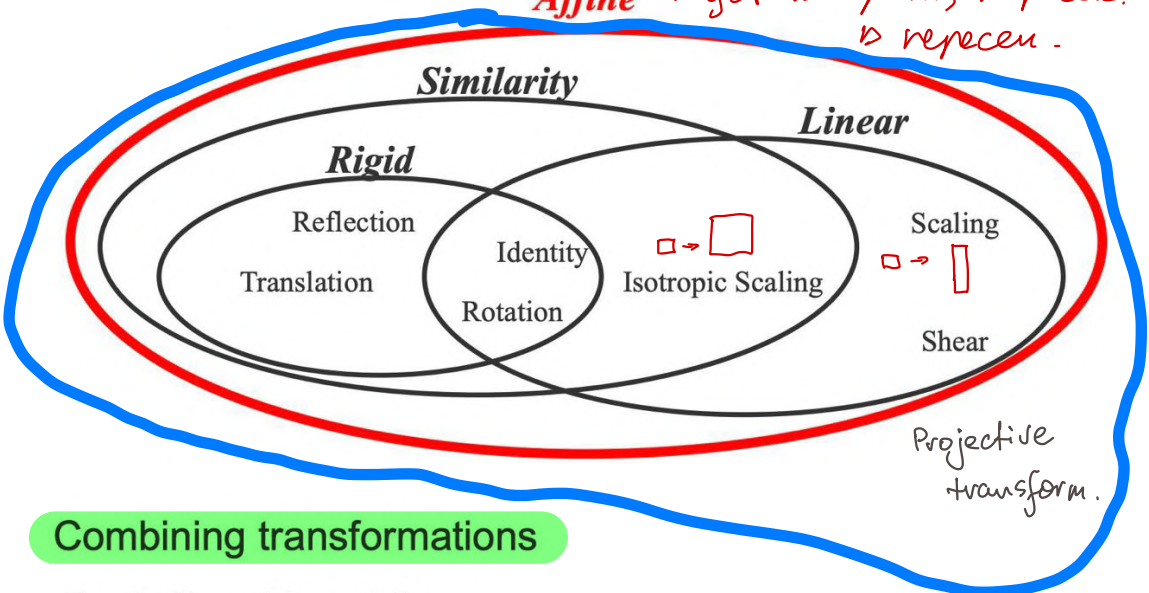




Transformation classes

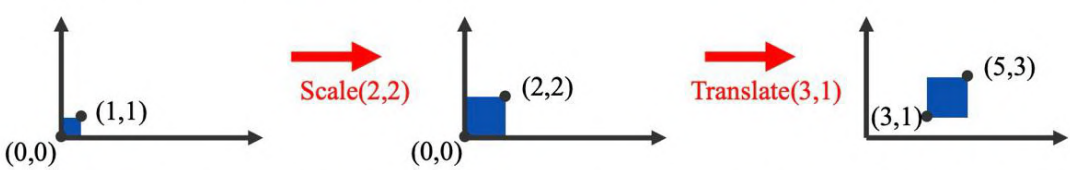
$$T_{affine} = \begin{pmatrix} T_{3\times3} & t \\ \vec{0} & 1 \end{pmatrix}$$

- Affine transforms – (i.e. preserve collinearity & parallelism) отображ. простр. в себя, при котором паралл. прям. переходят в паралл., пересек. – в пересек.
  - Preserve parallel lines



Combining transformations

Scale then Translate

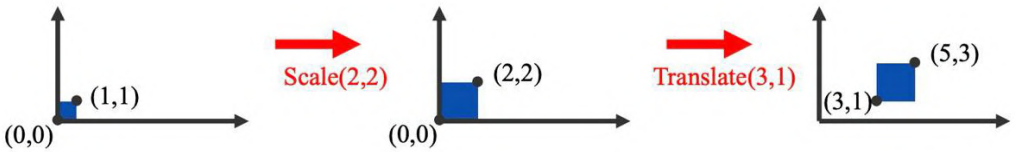


Use matrix multiplication: $p' = T(Sp) = TSp$

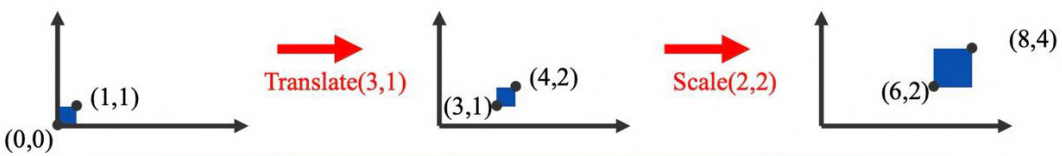
$$TS = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Caution: matrix multiplication is NOT commutative!

Scale then Translate: p' = T (S p) = TS p



Translate then Scale: p' = S (T p) = ST p



- Rotation (counterclockwise around x axis): $T_{rotx} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Косинус всегда или больше нуля

- Rotation (counterclockwise around y axis): $T_{roty} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

$T_{rot}^{-1} = T_{rot}^{T}$

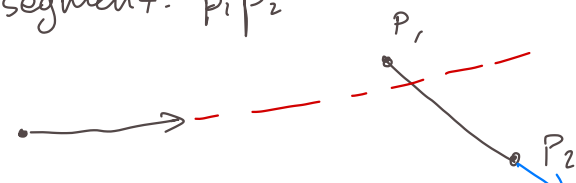
- Rotation (counterclockwise around z axis): $T_{rotz} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- Shearing: $T_{shear} = \begin{pmatrix} 1 & \lambda & \mu & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Reflection by yz Plane $\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Ray: S - start point, $\vec{d}$ - direction

Line segment: P_1P_2



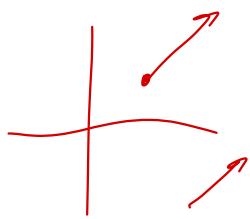
- assume line is also a ray

$$\begin{cases} S + t\vec{d} \\ 0 \le u \le 1 \end{cases} = P_1 + u(\overrightarrow{P_2 - P_1})$$

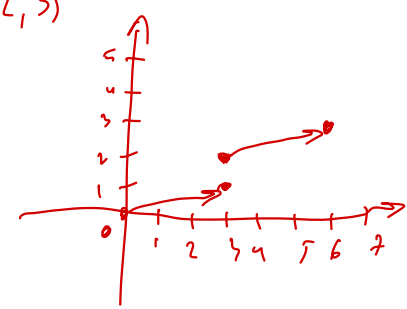
$P_1 + u(P_2 - P_1)$

одно уравнение для каждой координаты

$S + t\vec{d}$



(2, 3)



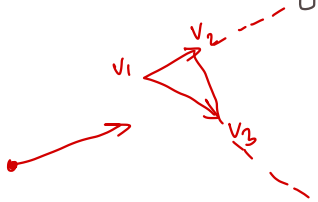
Cases:
1) Паран.
2) Пересек.
3) Лежат на одн. прямой

$(6,3) - (3,2) = (3,1)$

$(3,1) + (3,2) = (6$

$(0,0) + (3,2) = (3,2)$

Test whether a ray with origin S and direct. $\vec{d}$ intersects a triangle with vert. v_1, v_2, v_3



$$S + t\vec{d} = v_1 + u(v_2 - v_1) + v(v_3 - v_1)$$

$0 \le u, v \le 1 \quad u + v \le 1$ – Barycentric coordinates!

$1 - u - v$ при v_1

Test whether a point with coords (x, y, z) is in the interior of a triangle with v. v_1, v_2, v_3.

$$\begin{cases} v_1 + u(v_2 - v_1) + v(v_3 - v_1) = p \\ u + v \le 1 \\ 0 \le u, v \le 1 \end{cases}$$

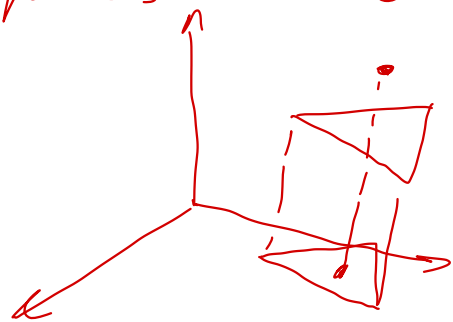
2 unknowns, 3 equations → $\begin{cases} p = uv_1 + vv_2 + wv_3 \\ 1 = u + v + w \Rightarrow w = 1 - u - v \end{cases}$

just eliminate one - it's a 2D projection

только смотреть, что в этой проекции треугольник не превратился в линию!

Еще →

Может статься, что точка будет над треугольн., но в проекции получится, что они пересекаются.

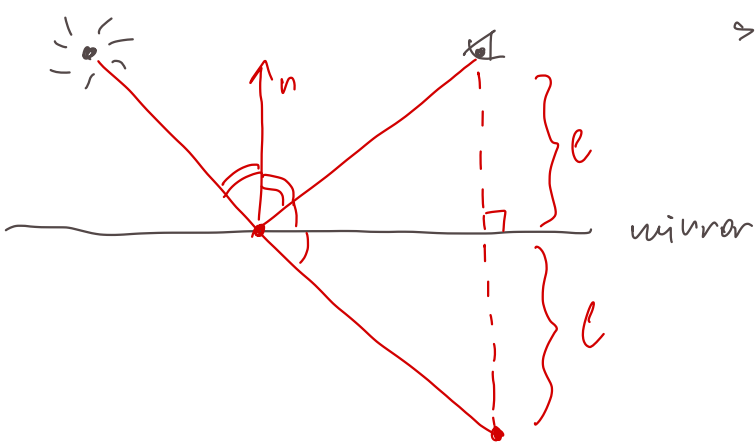


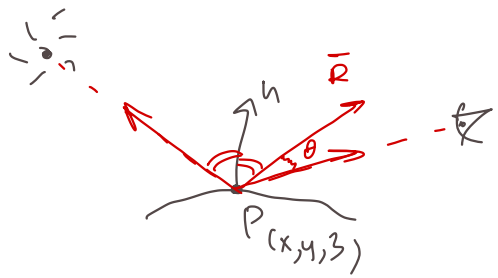
Тогда нужно проверить, что точка принадлежит той же плоскости, что и треуг?

Точка лежит в плоск., если ее можно представить в виде

$$P = \alpha\vec{a} + \beta\vec{b}$$ - линейн.

Does the observer see the refl. -?





$\bar{R}$ - reflection vector

θ - там про диффуз намотивацию.

Все вектора должны идти из P и быть нормализованы!

Нормализация вектора:

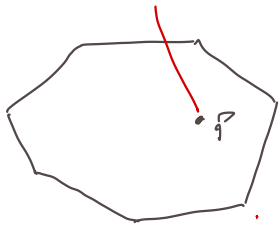
1) Модуль (длина вектора):

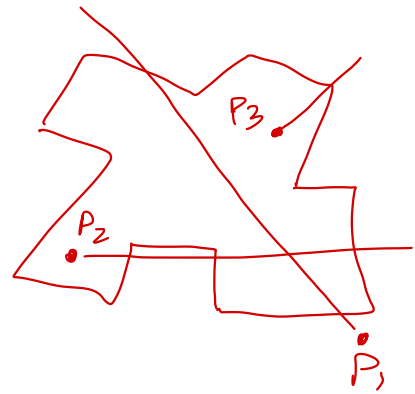
$$|\bar{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

2) Делим компоненты на длину:

$$\vec{v_n} = \left(\frac{v_x}{|\bar{v}|}, \frac{v_y}{|\bar{v}|}, \frac{v_z}{|\bar{v}|} \right)$$

Дан n-polygon. Потестить, что P наход. внутри полигона.

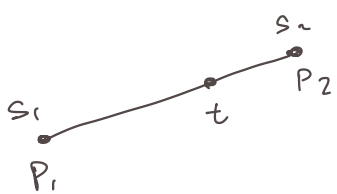




Возьмем и проведем какой-нибудь луч из P и посчитаем все пересечения со сторонами полигона. Тогда, если их четн. кол-во – она внутри, четн.-снаружи

Но! Может как P3 случайно попасть в угол. Поэтому надо проверять углы

↑ тогда уже верно! :(

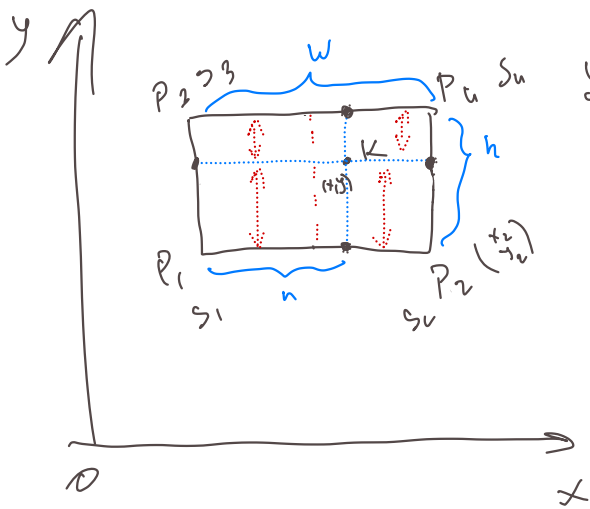


- написать уравн. лин. интерпол.

$$|\overline{S_2 - S_1}| = \sqrt{(P_{2x} - P_{1x})^2 \dots z^2}$$

длина вектора

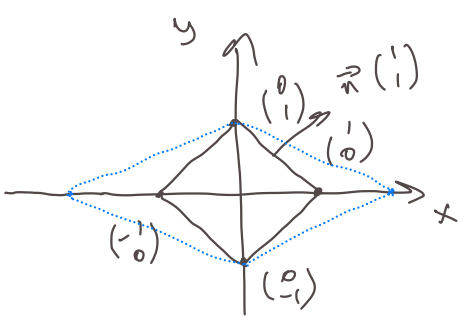
$$S_1 \cdot (d - t) + S_2 t = interpol$$



Уравн. одной билин. интерп.

$$u = \frac{x - x_1}{w} \qquad v = \frac{y_4 - y}{h}$$

$$I_1 = S_1(1-u) + S_2 u$$
$$I_2 = S_3(1-u) + S_4 u$$
$$I_3 = I_1(1-v) + I_2 v$$



Transf. matrix

как растянет по x

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} =$$

точка

т.к. это точка, и если бы было перемещение, оно бы на нее влияло

как растянет по y

$$= \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}$$

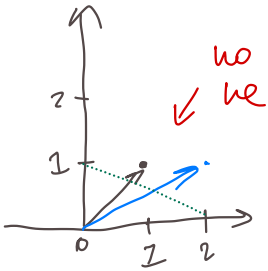
$$TM \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \qquad TM \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \\ 1 \end{pmatrix} \qquad TM \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

↓

меняем вектор нормали

$$TM \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$$

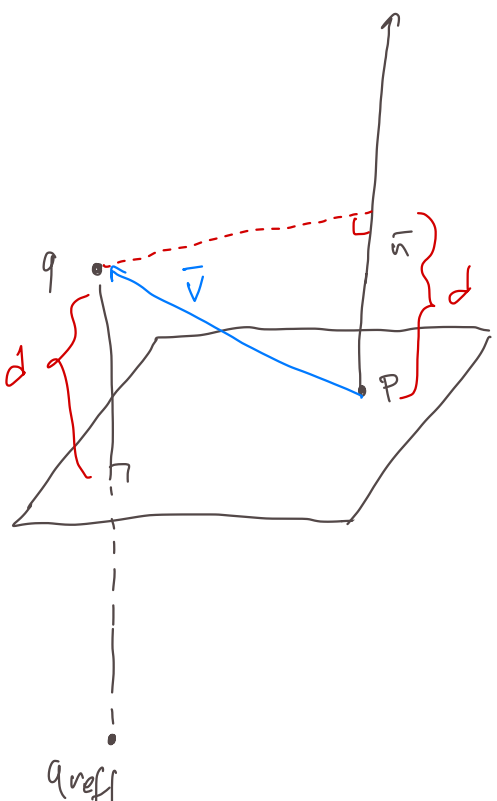
нормаль, если бы было перемещение, оно бы на нее не влияло



но теперь нормаль не 90°!

было
стало
грань ромба (нового)

Reflection transformation matrix



$d = (\bar{v} \cdot \bar{n}) = ((\overline{q-p}) \cdot \bar{n})$

↑ проекция $\bar{v}$ на $\bar{n}$

$q_r = q - 2\bar{n}d =$ ← направл.

$= q - 2\bar{n}((q-p) \cdot \bar{n}) =$

$= q - 2\bar{n}(q \cdot \bar{n}) + 2\bar{n}(p \cdot \bar{n})$

$q_r = q - 2\bar{n}(q \cdot n) + 2\bar{n}K$

$K = (p \cdot n)$

$$\begin{cases} q_{rx} = q_x - 2n_x(q_x n_x + q_y n_y + q_z n_z) + 2n_x K \\ q_{ry} = \dots \\ q_{rz} = \dots \end{cases}$$

Скалярн. произв.

$\bar{a}\cdot\bar{b} = |\bar{a}|\cdot|\bar{b}|\cdot\cos\varphi$

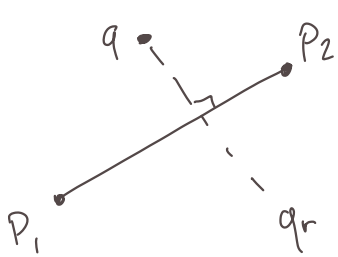
$\bar{a}\cdot\bar{b} = x_1x_2 + y_1y_2 + z_1z_2$

$\bar{a}\perp\bar{b} \Leftrightarrow \bar{a}\cdot\bar{b} = 0$

$$q_{rx} = q_x(1-2n_x^2) - q_y\cdot n_y\cdot 2n_x - q_z\cdot n_z\cdot 2n_x + 2n_x K$$

$$\begin{pmatrix} q_{rx} \\ q_{ry} \\ q_{rz} \\ 1 \end{pmatrix} = \begin{pmatrix} 1-2n_x^2 & -n_y 2n_x & -n_z 2n_x & 2n_x K \\ \dots & 1-2n_x n_y & \dots & 2n_y K \\ \dots & \dots & 1-2n_x n_z & 2n_z K \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} q_x \\ q_y \\ q_z \\ 1 \end{pmatrix}$$

Если бы reflection проходил через начало координат, этого бы не было



Reflect a point onto a line in 3D

Find Normal

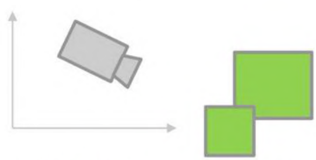
$L(t) =$

1. линия P_1P_2 это $P_1 + t(P_2 - P_1)$
2. $d^2 = (P_{1x} + t(P_{2x} - P_{1x}) - q_x)^2 + (P_{1y} + t(P_{2y} - P_{1y}) - q_y)^2 +$

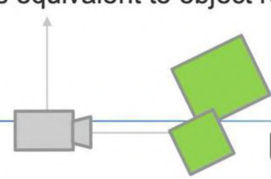
Каквото хрень + derivative → ...z
Не будет на изпе

Viewing Transformations

- To render a portion of the world coordinate space, one positions and orients the camera

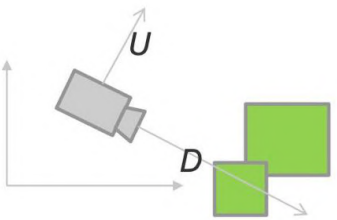


  - The same image can be shot by fixing the camera in world space (eg. at (0.0.0), viewing along (0,0,1)), and transforming objects accordingly
    - Objects transform inversely to the transform that positions and orients the camera
      - e.g. camera movement (forward) is equivalent to movement of objects towards the camera
      - e.g. rotation about α around y-axis is equivalent to object rotation about $-\alpha$ around y-axis
  - This is the viewing transform

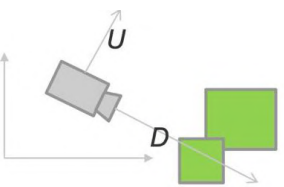


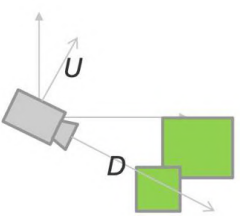
default camera

- How to build the viewing transformation
  - The user specifies:
    - Camera position *C*
    - Viewing direction *D*
    - Up vector *U*



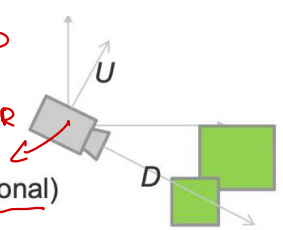
- Move camera *C* to origin:
  - Translation by –*C*: M_{trans}

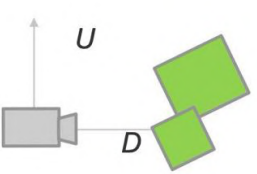




получаем третий ортогональный вектор

- Build orthonormal camera frame:
  - „right“ *R* = *D*x*U* – cross product
  - „zenith“ *U* = *R*x*D* (only if *U*, *D* are not orthogonal)
- Adjust orientation:
  - Matrix [*R*,*U*,*D*] maps $[e_1, e_2, e_3]$ to [*R*,*U*,*D*]
  - So use [*R*,*U*,*D*]$^{-1}$
  - Final transform: M_{view}= [*R*,*U*,*D*]$^{-1}$ M_{trans}

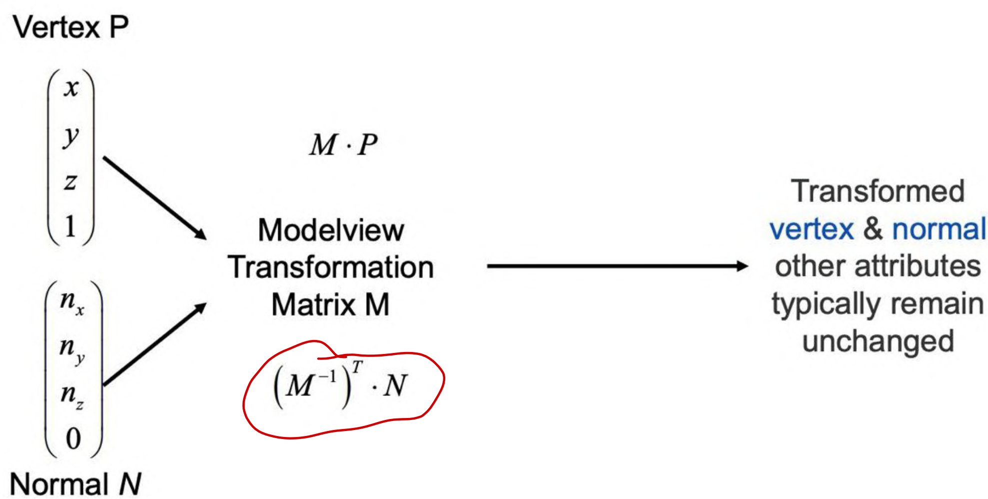




- Viewing and modeling in one single transformation – the modelview transformation $M = M_{view}M_{mod}$

- After the modelview transformation the objects coordinates are in view (or camera) space

- Modelview transformation of vertices and normals



After the transformation should be:

норм. → $n' \cdot v' = 0$ (← вект.), т.е. должна быть такая матр. Q, что

$(Qn)^T(Mv) = 0$ — трансф. || → = ; вект.; матрица преобр.

$M^T = \begin{pmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & c_3 \end{pmatrix}$

$n^T Q^T M v = 0$

since $n^T \cdot v = 0$

$Q^T \cdot M = I \Rightarrow Q^T = M^{-1} \Rightarrow Q = (M^{-1})^T$

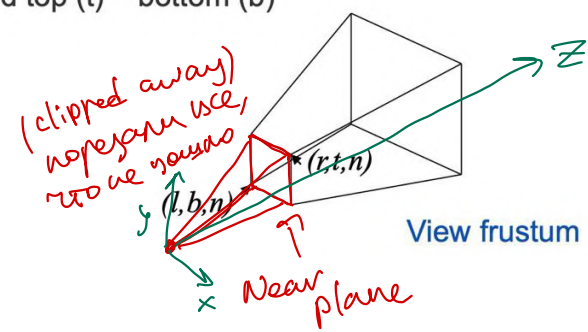
Perspective Projections

- Matrix representation of the standard projection onto the $z = 1$ plane
  - First: bring z component into 4th component of result vector
  - Second: divide by 4th component

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix} \overset{\text{div z projection}}{\equiv} \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix}$$

Чтобы спроецировать на z добавил. в матр. 1 на уровне z, а потом делим все на z.

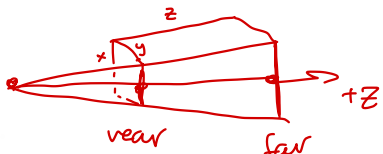
- Assume the origin of camera space at the center of projection (the eye), the viewing direction along z
- Graphics APIs require to select the visible portion of 3D space, ie. the view frustum - вид усеченного конуса
  - Near/far plane (n,f)
  - Width/height of view window given as right (r) – left (l) and top (t) – bottom (b)



- The API projection matrix (n = near, f = far)
  - Not complete, see later

$$\mathbf{M}_P = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

projection on the near plane



точка на near плоскости

- Examples

API: project

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{pmatrix} 0 \\ 0 \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ (n+f)n-nf \\ n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ n(n+f-f) \\ n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ nn \\ n \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 0 \\ n \\ 1 \end{pmatrix}$$

Point on near plane remains on near plane

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ (n+f)f-nf \\ f \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f(n+f-n) \\ f \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ ff \\ f \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix}$$

Point on far plane remains on far plane

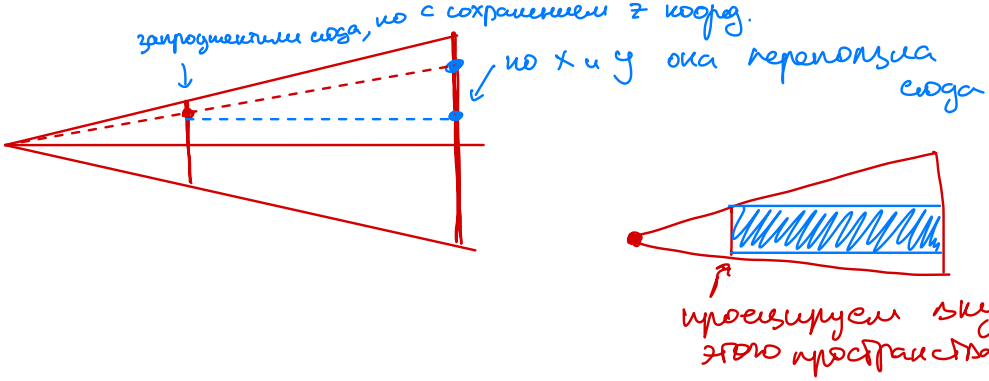
$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{pmatrix} 4 \\ 7 \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ (n+f)n-nf \\ n \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ n(n+f-f) \\ n \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ nn \\ n \end{pmatrix} \equiv \begin{pmatrix} 4 \\ 7 \\ n \\ 1 \end{pmatrix}$$

Point on near plane remains unchanged on near plane

distance from proj. center

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{pmatrix} 4 \\ 7 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ (n+f)f-nf \\ f \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ f(n+f-n) \\ f \end{pmatrix} = \begin{pmatrix} 4n \\ 7n \\ ff \\ f \end{pmatrix} \equiv \begin{pmatrix} 4n/f \\ 7n/f \\ f \\ 1 \end{pmatrix}$$

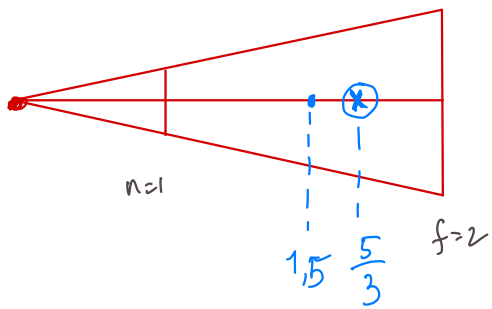
Point on far plane moves on far plane

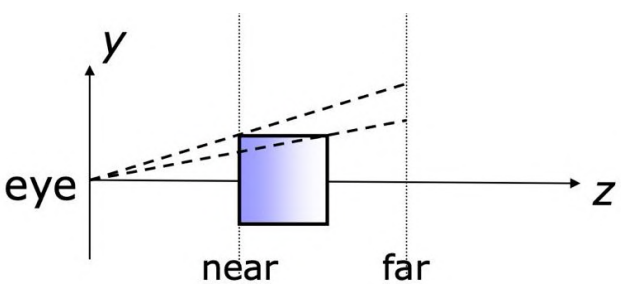


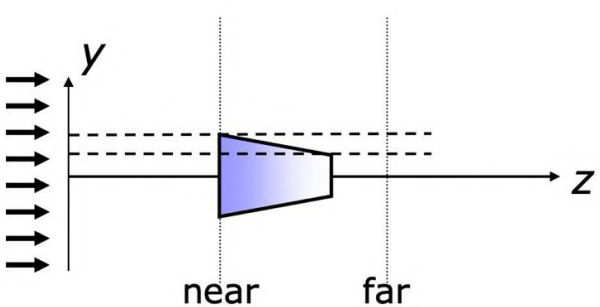
$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} 0 \\ 0 \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ (n+f)z - nf \\ z \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 0 \\ n+f-\frac{nf}{z} \\ 1 \end{pmatrix}$$

Points between near and far move towards far plane

Assume n = 1, f = 2, z = 1.5 $\Rightarrow$ z = 5/3 > 1.5



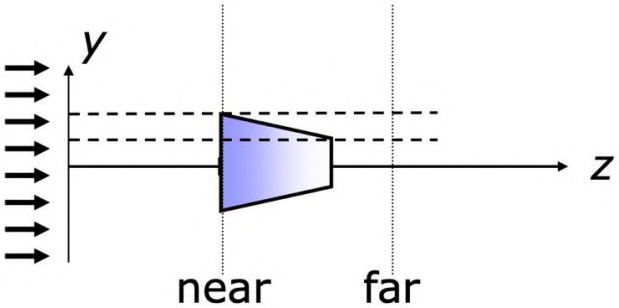




View Coordinates

are perspectively distorted ...

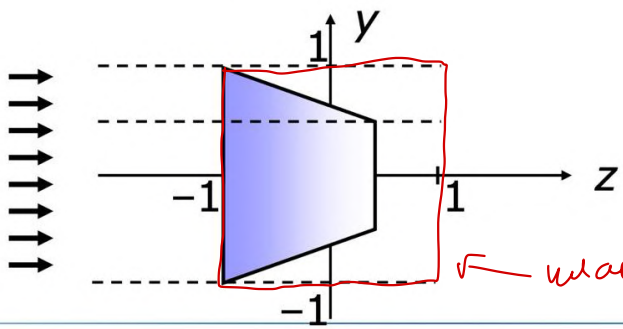
so that x/y coordinates are projected coordinates in near plane



View Coordinates

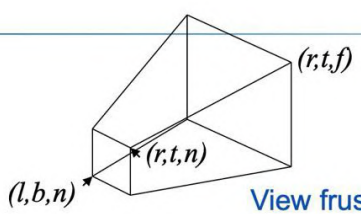
are finally transformed to

Normalized Device Coordinates (NDC)



← кладем из фрустума в куб

Perspective projections



View frustum

- The API projection matrix
  - n = near, f = far, r = right, l = left, t = top, b = bottom

$$\mathbf{M} = \mathbf{M}_O\mathbf{M}_P = \begin{bmatrix} \frac{2}{(r-l)} & 0 & 0 & \frac{-(r+l)}{(r-l)} \\ 0 & \frac{2}{(t-b)} & 0 & \frac{-(t+b)}{(t-b)} \\ 0 & 0 & \frac{2}{(f-n)} & \frac{-(f+n)}{(f-n)} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & (n+f) & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

матрица для преобраз. всего в куб

сжатая матр.

  - Scales the transformed frustum to -1,1 and centers around (0,0,0) via a translation, all vertices not in frustum are outside -1,1

- Modelview and perspective transformation in one single transformation

$$M = M_{perspective} M_{modelview}$$

vertex transformation: $\begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} = M \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

нам делим всё на w!

- Result of vertex shader stage
  - Homogeneous vertex coordinates after perspective projection
  - (modelview transformed) normals
  - Additional attributes like color and texture coordinates

$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad \begin{pmatrix} n_x \\ n_y \\ n_z \\ 0 \end{pmatrix}$

- The transformed, attributed vertex stream is passed to the rasterizer stage
  - The rasterizer performs division by *w* and maps NDC to pixel coordinates
  - For each triangle, the rasterizer determines the pixels covered by this triangle – for each such pixel a fragment is generated
  - Per-vertex attributes are interpolated to each fragment

Normalized device coordinates

если w=1, просто делимся на 1

Vertex Shader

- in vec3 vertexPosition – вектор координат вершины
- in vec3 vertexNormal – вектор нормали
- in vec2 vertexTextureCoordinates – текстурные координаты

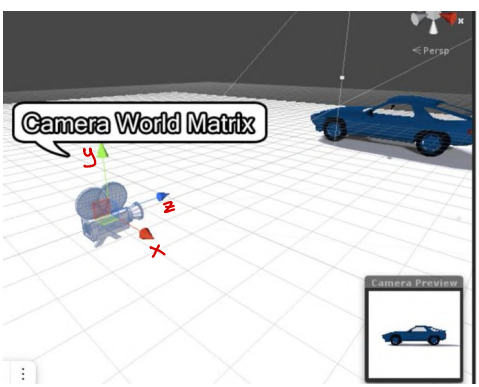
- out vec3 vertexColor

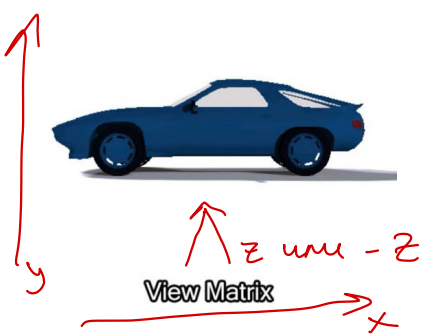
глобальная константа на все шейдеры рендеринга (during single render call)

uniform mat4 mMatrix – матрица преобр. координат объекта в "world coordinates" (model; rotation, transl, scaling)

uniform mat4 vMatrix – матрица преобр. координат мира в коорд. камеры (view; rotation, transl. (usually))

uniform mat4 pMatrix – матрица проекции, преобр. коорд. камеры в коорд. экрана (→ NDC)



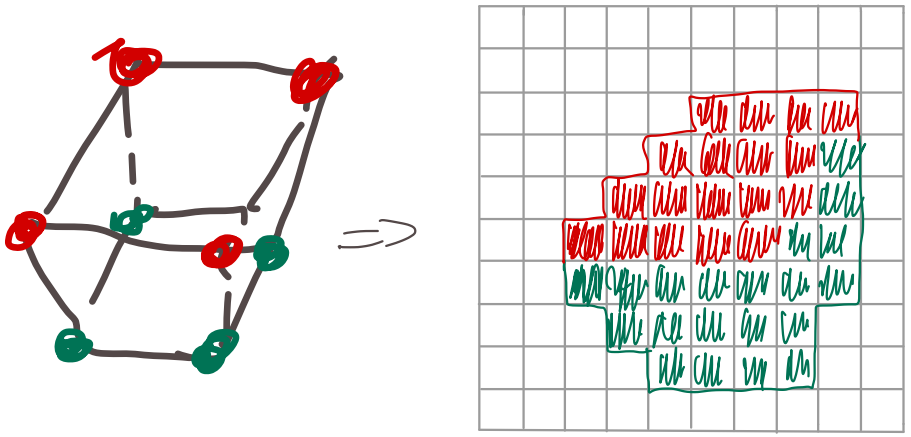


В шейдере пишем: $M_v(M_m(\vdots))$

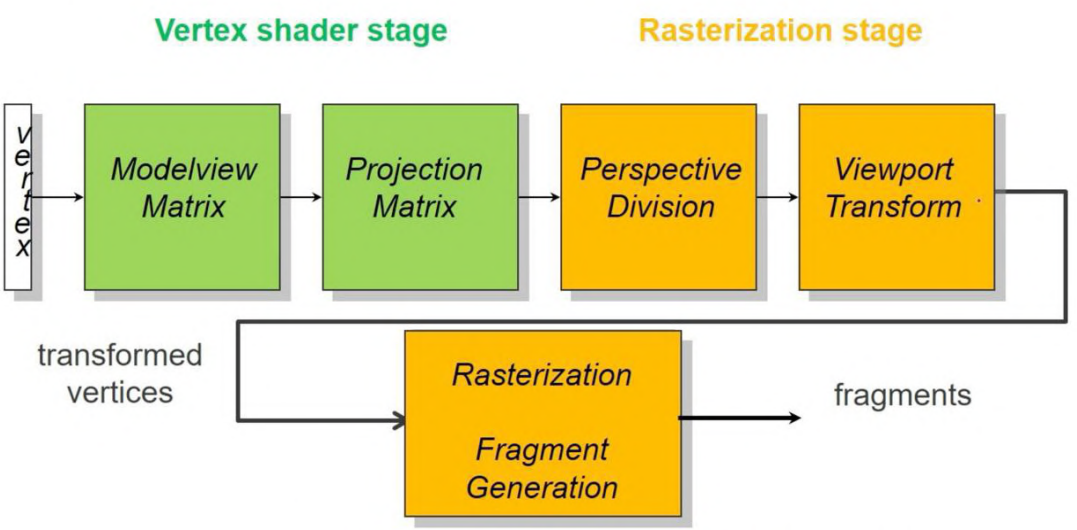
```
vec4 vertexCamSpace = vMatrix * mMatrix * vec4(
                                    vertexPosition, 1.0);
gl_Position = pMatrix * vertexCamSpace;
```

↑ output variable with final position of the vertex on the screen

Fragment Shader



Vertex shader stage



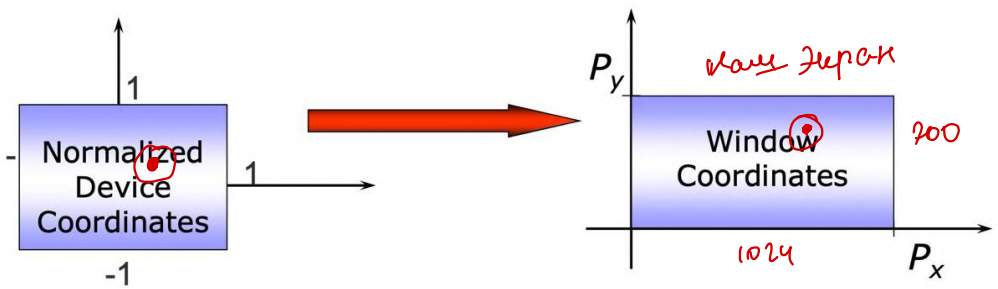
Green: in the vertex shader stage
Orange: in the rasterizer stage

Rasterization

- Viewport transformation maps from NDC (Normalized Device Coordinates) to pixel coordinates

Example: $P_x = 1024$, $P_y = 512$ $\Rightarrow$ $\begin{pmatrix}0.2\\0.2\\0.0\end{pmatrix} \rightarrow \begin{pmatrix}613\\306\\0.5\end{pmatrix}$

– (613, 306) are the pixel coordinates of the vertex



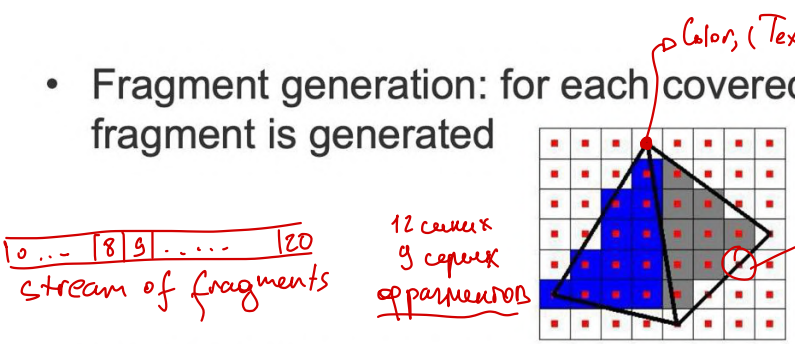
- Viewport transformation maps from NDC (Normalized Device Coordinates) to pixel coordinates

Viewport transformation matrix:

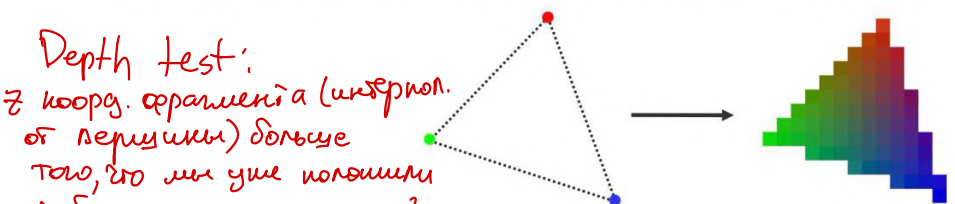
$$\begin{pmatrix}P_x & 0 & 0 & 0\\0 & P_y & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0 & 0.5\\0 & 1 & 0 & 0.5\\0 & 0 & 1 & 0.5\\0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}0.5 & 0 & 0 & 0\\0 & 0.5 & 0 & 0\\0 & 0 & 0.5 & 0\\0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}$$

note: depth (z) values are in [-1,1] after perspective projection are transformed to [0,1] by viewport transformation

- Fragment generation: for each covered pixel, one fragment is generated



- For each fragment: per-vertex attributes (color, normal, z-value, texture coordinates,…) are interpolated



- Result of rasterization stage:

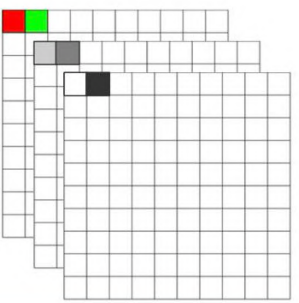
  A set of fragments, each storing its pixel coordinate as well as interpolated z-value, color, normal, texture coordinate etc.

X,Y	z	RGB	u,v	(N_x, N_y, N_z)	…

A fragment is in fact a surface point seen through the respective pixel

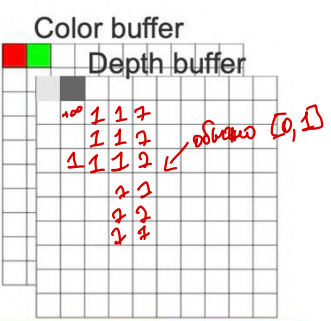
Fragment processing

- On the GPU, multiple buffers are used during rendering
  - Color (pixel) buffer: stores for each pixel a RGBα color, where α is the opacity
  - Depth buffer: stores for each pixel the depth of the closest surface point rendered into this pixel so far

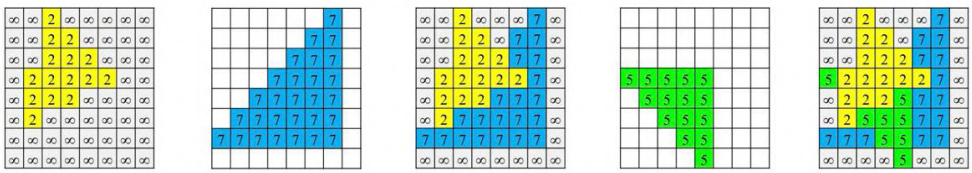


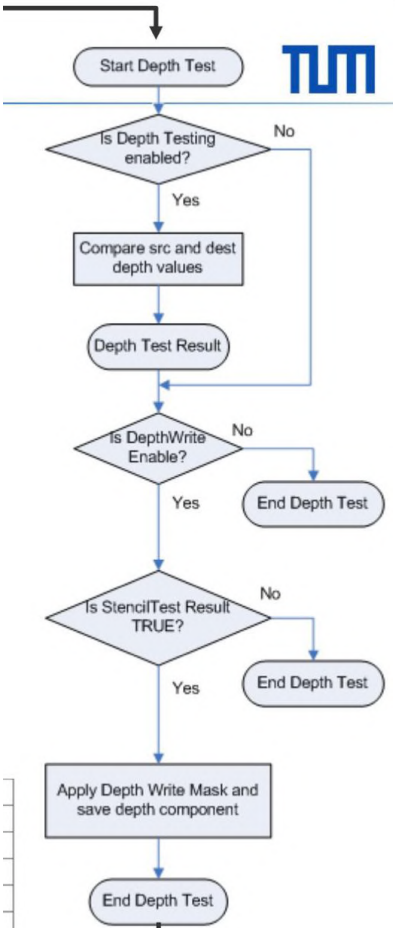
- Before a fragment is written to the color buffer, it undergoes the deth test: the fragment's depth value is tested against the depth value in the depth buffer
  - The depth buffer is initialized with the highest possible value
  - Whenever a fragment writes it's color into the color buffer, it writes it's depth value to the same location in the depth buffer
  - A fragment is allowed to write it's color value only if it passes the depth test

happens in every frame (rendering pass)



- During rendering the depth buffer stores for each pixel the minimum depth value of all fragments which were falling into the pixel
- When multiple surface points fall into the same pixel, only the one closest to the near plane is stored





Depth buffer
Его можно access, но
- depth test сам не programmed!
Но он можно enable/disable

Depth Write Mask – можно оставить тест, но сделать так, что вышестоящие пиксели не будут перезаписывать буфер

Depthfunc – usually "less" – survive if depth is less than in buffer, но может быть more, equal...

- What if the color of more than one fragment should be combined and stored in the color buffer
  - Note that the depth test discards all but the closest fragment and shows its color
- Solution: **disable depth test** and **blend** together the color of all fragments falling into one pixel

- α-blending allows for the combination of fragment (src) and pixel colors (dst) by considering the fragments and pixels opacity
  - α-values correspond to opacity [0,1]
  - Indicates the percentage of light from behind that is blocked

процент света, исходящ. цвета, кот. блок.

Opacity 0,9 => 90% света блок.

- α-blending
  - Assume 2 objects with colors $C_1\alpha_1$ and $C_2\alpha_2$
  - Object 1 is before object 2 with respect to the viewer
  - Consider colors and α-values to blend the 2 objects correctly:

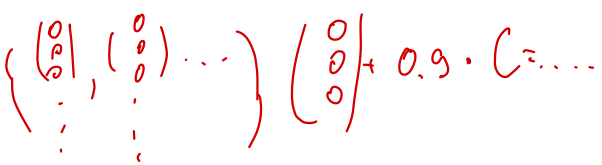
$$C = \alpha_1 C_1 + (1-\alpha_1)\,\alpha_2 C_2$$
$$\alpha = \alpha_1 + (1-\alpha_1)\alpha_2$$

передний объект → ; → сколько этого света видно

Example: $C_1\alpha_1$=(1,0,0,0.8), $C_2\alpha_2$=(0,0,1,0.5)

$$\Rightarrow C = 0.8 \cdot (1,0,0) + (1-0.8) \cdot 0.5 \cdot (0,0,1) = (0.8,0,0.1)$$
$$\alpha = 0.8 + (1-0.8) \cdot 0.5 = 0.9$$

Note that blending is NOT commutative

int color buf. - 

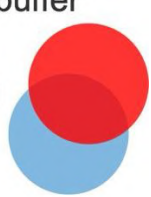
- α-blending of two fragments with colors $C_1\alpha_1$ and $C_2\alpha_2$
  - Render the more distant fragment first into the color buffer, let its color being modulated by its opacity:

$$C_D = \alpha_2 C_2$$

  - Render the closer fragment and let its color being modulated by its opacity

$$C_S = \alpha_1 C_1$$

and blend the colors via $C = C_S + (1-\alpha_1)\,C_D$ into the color buffer



  - Since we render back-to-front, depth test can be disabled or enabled
  - Also accumulation of α not required

Back to Front!

- To blend multiple transparent objects correctly, the following approach is performed:

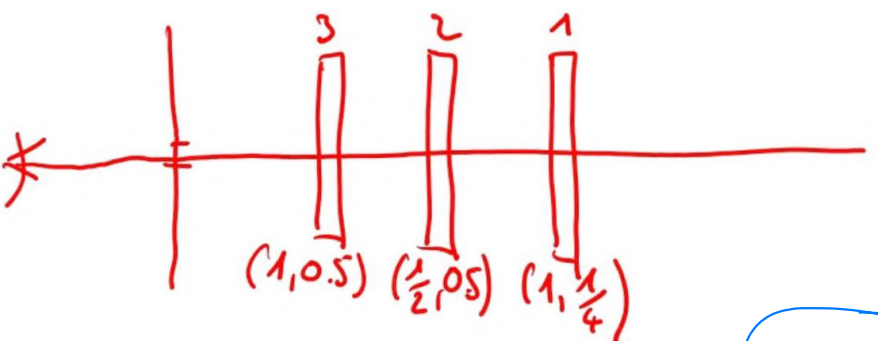
  (1) clear color/opacity (0,0,0,0) in the colorbuffer

  (2) Sort objects from back to front (wrt. decreasing distance to the viewplane)

  (3) Render the objects in sorted order via the blend equation

  $C_D = \alpha_S C_S + (1-\alpha_S) C_D$ – ?

  here, *s* (src) represents the incoming fragment
  and *d* (dst) represents the values in the colorbuffer



$C = (0,0)$

1) $C = (\frac{1}{4} \cdot 1) + (1 - \frac{1}{4}) \cdot C'' = \frac{1}{4} \cdot 1$

2) $C = 0{,}5 \cdot \frac{1}{2} + (1 - 0{,}5) \cdot C$

nur α_2

- To blend multiple transparent and opaque objects correctly, the following approach is performed:

(1) clear color/opacity (0,0,0,0) in the colorbuffer

(2) Sort transparent objects from back to front (wrt. decreasing distance to the viewplane)

(3) Render the opaque objects with depth test / write enabled

(4) Render the transparent objects (with depth test enabled) in sorted order via the blend equation

$C_D = \alpha_S C_S + (1-\alpha_S) C_D$

here, *s* (src) represents the incoming fragment and *d* (dst) represents the values in the colorbuffer

перемещ. каждый раз новый value

потому что в depth test нам уже дали только те, что видим

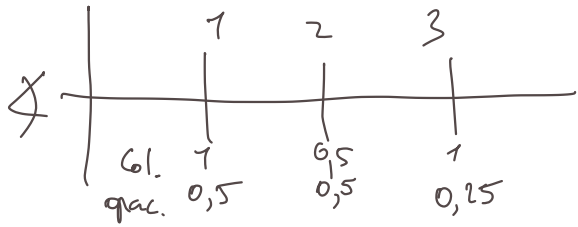
Отдельно рендерим прозр. и непрозр. объекты

just performance thing

Rendering Front to Back is also beneficial

1000000 объектов

opacity 1 - можно останавливаться



$$C = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^{Col}_{\alpha}$$

$$C_i = \underbrace{T_C}_{\text{текущ. зн. буф.}} + \underbrace{(1-\alpha)}_{1-\text{opacity буфера}} (0{,}5 \cdot 1) = 0 + (1-0) \cdot \underbrace{0{,}5 \cdot 1}_{C \cdot \alpha_C} = 0{,}5$$

$$C_2 = \underbrace{0{,}5 \cdot 1}_{\text{текущ. зн. буф.}} + \underbrace{(1-0{,}5)}_{\text{frac. буф.}} \cdot \underbrace{0{,}5 \cdot 0{,}5}_{C \cdot \alpha_C} = 0{,}5 \cdot 1 + 0{,}5 \cdot 0{,}5 \cdot 0{,}5 = 0{,}5 \cdot 1 + 0{,}25 \cdot 0{,}5$$

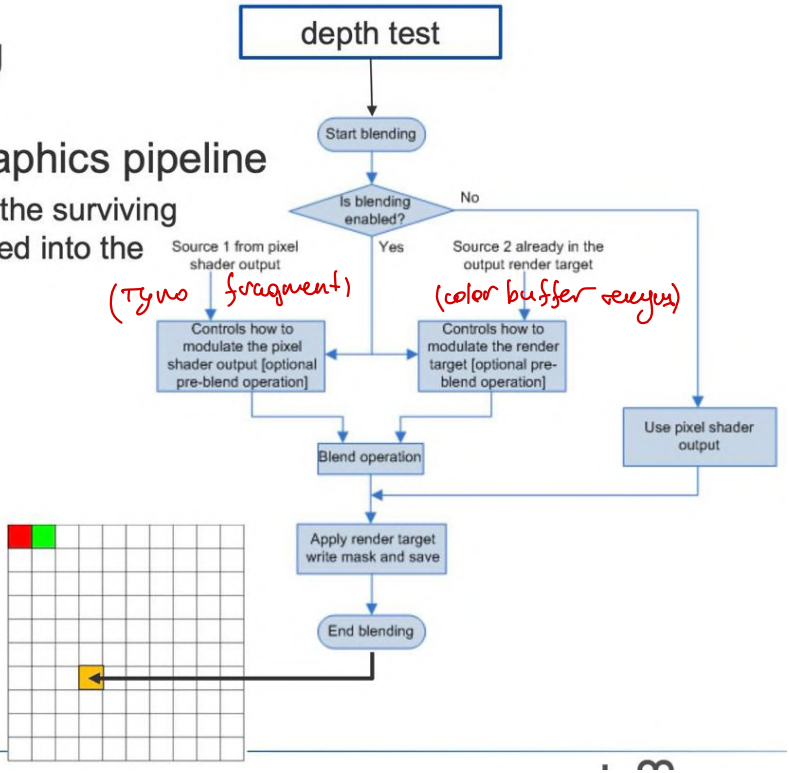
Front to back:

$$Buf_i = \underbrace{C_{i-1} \cdot \alpha_{i-1}}_{\text{буфере}} + \underbrace{(1-\alpha_{bi-1})}_{\text{opacity буф.}} \underbrace{\alpha_C \cdot C}_{\text{нов. объект}}$$

$$\alpha_i = \alpha_{i-1} + (1-\alpha_{i-1}) \cdot \alpha_C$$

Fragment processing

- Blending in the graphics pipeline
  - After the depth test, the surviving fragments are blended into the color buffer



- Blending (in back to front order)
  - It is enabled via DX calls and performed automatically
  - Color/alpha values in the color buffer cannot be accessed directly

$\alpha_s C_s$ - source, current color, что, которым рис.

$\alpha_d C_d$ - то, где > будет

```
// Create an alpha enabled DX blend state description
blendStateDescription.RenderTarget[0].BlendEnable = TRUE;
// αsCs
blendStateDescription.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
// (1-αs) CD
blendStateDescription.RenderTarget[0].DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
// αsCs + (1-αs)CD
blendStateDescription.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
```

$C_s \cdot \alpha_s$

$(1-\alpha_d) \cdot C_d$

back to front

интересно, что можно

Математика

Точка – $p = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$ dehomogenize the point

$p' = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}$, если $w \neq 0$

Derive the transformation matrix that can be used for rotating an object around the origin of rotation $(1, 0, 0)^T$ counterclockwise by 90° around the z axis. origin of object x y z

Hint: The formulas for the rotation matrices given above assume that the origin of rotation is at $(0, 0, 0)^T$. Thus, you need to first translate the origin of rotation to $(0, 0, 0)^T$ and then translate the origin of rotation back to its original position.

$$T_{rot_z} = \begin{pmatrix} \cos 90 & -\sin 90 & 0 & 0 \\ \sin 90 & \cos 90 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_{transl_1} = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad T_{transl_2} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_{res} = T_{transl_2} \times T_{rot_z} \times T_{transl_1}$$

a) Which of the following statements are true and which are wrong? Explain your answers. Answers without a correct explanation will not be graded.

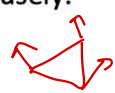
a. When using the Phong illumination model and decreasing the angle between the incoming light direction and the normal vector, the reflection at a point on a purely specular reflecting surface always appears brighter.

Wrong. Phong lightning:

$$I_r(x,\omega_r) = k_d \cdot (\vec{l}\cdot\vec{n}) \cdot I_i(x,\omega_i) + \underbrace{k_s \cdot (\bar{r}\cdot\bar{v}) \cdot I_i(x,\omega_i)}_{\text{specular}} + k_a I_a$$

doesn't depend on α!

b. Gouraud and Phong shading give the same result when the surface reflects only diffusely.

Gouraud:  – calc. color, then interpolate

Phong:  – interpol. normals, then calc. colors

– Gouraud – dcurve colors
Phong – light colors

Wrong.

c. The light reflection at a perfect mirror does not change when the viewer's position changes.

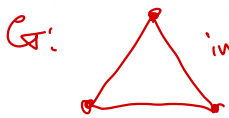
True

d. The intensity of the light that is reflected at a point on a purely diffuse reflector is independent of the direction of the incoming light.

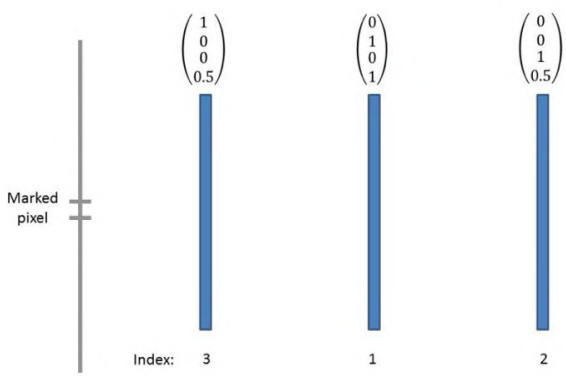
False

$$I_r(x,\omega) = k_d \cdot \underbrace{(\vec{l}\cdot\vec{n})}_{\text{cos}} \cdot I_i(x,\omega_i)$$

b) How many barycentric interpolation operations does the rasterizer have to perform for a triangle that covers 500 pixels when a) Phong shading and b) Gouraud shading is performed? Note that answers without a derivation will not be graded



c) Given is the scene below, which consists of opaque and semi-transparent objects. For each object, its color (first 3 vector components) and opacity (4th component) is given. The objects are rendered in the order of increasing *index* via the rasterization-based rendering pipeline. α-blending is used to render the objects in a realistic way.



a. **(3 Points)** What is the color of the marked pixel if depth testing / writing is enabled and the depth test is set to "Less"?

$\begin{pmatrix} 0.5 \\ 0.5 \\ 0 \\ 1 \end{pmatrix}$

$\alpha_s C_s + (1-\alpha_s)$

b. **(3 points)** What is the color of the marked pixel if depth testing is disabled?

$\begin{pmatrix} 0,5 \\ 0,25 \\ 0,25 \\ 1 \end{pmatrix}$

c. **(4 points)** What is the color of the marked pixel if depth testing is disabled and the objects are rendered in back-to-front order?

$C_0 = 0{,}5 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0{,}5 \end{pmatrix}$

$\alpha_0 = 0{,}5$

$C_1 = 1 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + (1-0) \cdot \ldots = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$

$\alpha_1 = 1$

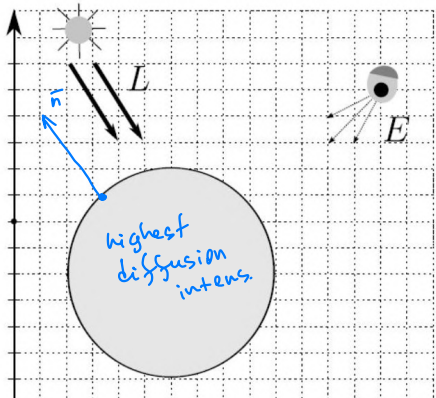
$C_2 = 0{,}5 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + (1-0{,}5) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \cdot 1 = \begin{pmatrix} 0{,}5 \\ 0{,}5 \\ 0 \end{pmatrix}$

$\alpha_2 = 0{,}5 + (1-0{,}5) \cdot 1 = 1$ $\quad 1$

Assignment 3 (Illumination)

A sphere is illuminated by a directional light source, i.e., every point receives light from the same direction L. The intensity of the light is $I_L = 1$. The material constants for the sphere are $k_{diffus} = k_{specular} = 0.5, k_{ambient} = 0$. The specular exponent is $n = \sqrt{2}$. A 2D cross-section of the scene is given in the figure below. The observer is at the eye-point E (indicated by the black dot in the figure).

a) **(4 points)** Draw into the figure the location of the point at which the diffuse reflection is maximal and compute the value of the diffuse reflected intensity.

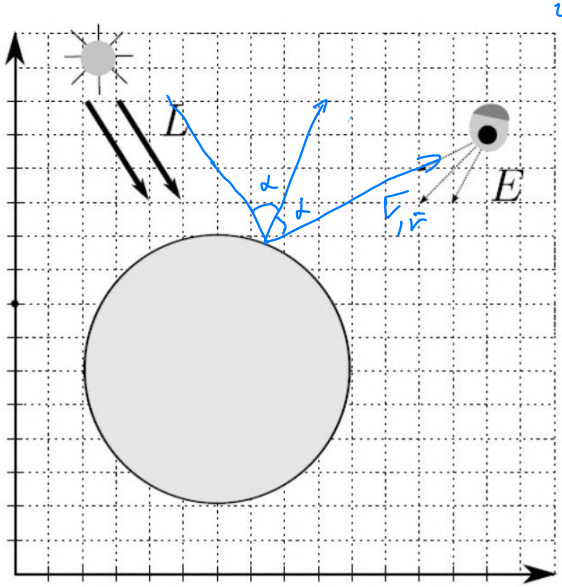


Максим. интенсивн. будет там, где $\cos\alpha = 1$ (нулевой)

b) **(4 points)** Draw into the figure the location of the point at which the specular reflection is maximal and compute the value of the specular reflected intensity.

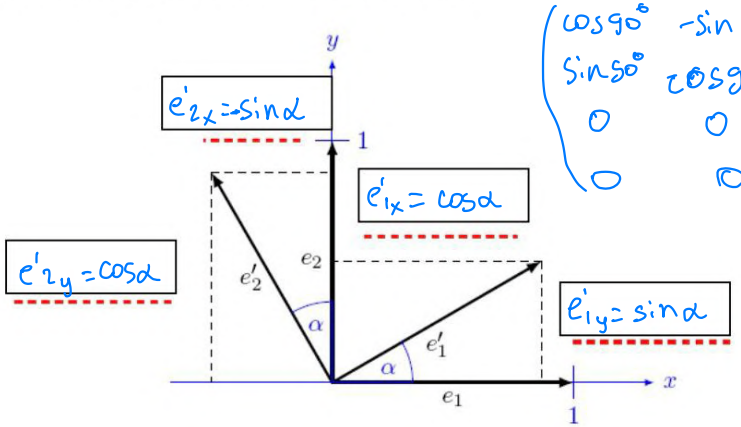
$$I_r(x, w_v) = k_s \cdot (\bar{r} \cdot \bar{v}) \cdot I_l(x, w_l) = 0{,}5 \cdot 1$$

максим. интенсивность будет там, где $(\bar{r} \cdot \bar{v}) = 1$, $\cos = 1$



a) **(4 Points)** In the figure below, the rotation of an angle α about the z-axis is illustrated. Write into the figure the lengths of the *x*- and *y*-components of e'_1 and e'_2, and write down the corresponding homogeneous 4x4 rotation matrix.

$$\begin{pmatrix} \cos 90^\circ & -\sin 90^\circ & 0 & 0 \\ \sin 90^\circ & \cos 90^\circ & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$\begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

b) In the exercises b1) and b2), you are supposed to specify the coefficients of 4×4 transformation matrices for the representation of affine mappings in $\mathbb{R}^3$. The coefficients should be given as precise as possible, i.e., numbers should be given if possible and identifiers else. The same coefficients should be given the same identifier. If a matrix can be specified by concatenation of multiple matrices, the concatenation does not have to be computed explicitly.

Example: isotropic scaling

$$\begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

b1) **(2 Points)** The inverse T^{-1} of the matrix

$$M^{-1} = \begin{pmatrix} s & 0 & 0 & t_x \\ 0 & s & 0 & t_x \\ 0 & 0 & s & t_x \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{s} & 0 & 0 & -\frac{t_x}{s} \\ 0 & \frac{1}{s} & 0 & -//- \\ 0 & 0 & \frac{1}{s} & -//- \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \checkmark$$

это либо S·T, либо T·S

$$M^{-1} \cdot M = I = M^{-1}(T \cdot S) = S^{-1} T^{-1} T \cdot S$$

b2) **(4 Points)** A rotation of 90 degrees about the axis (0,0,1) through the point (4,4,1).

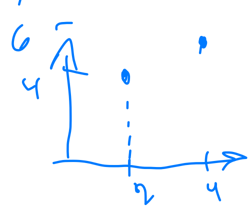
x y z

Мы перепутали. Этот вектор знак. wrong.

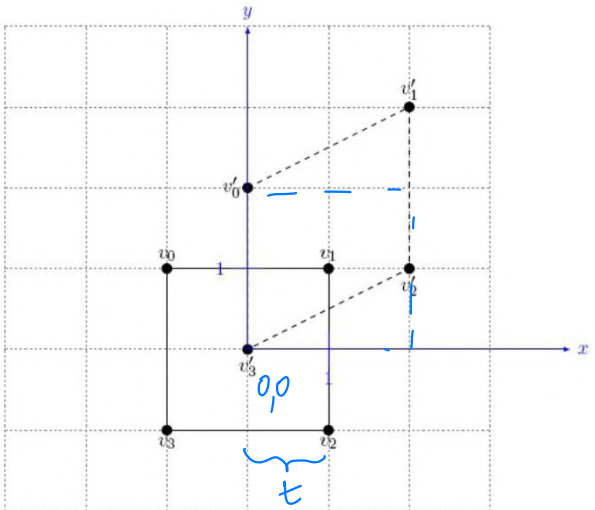
:(

$$\begin{pmatrix} 1 & 0 & 0 & -4 \\ 0 & 1 & 0 & -4 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & -4 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 2+(-4) \\ 4-6 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ -2 \\ 1 \end{pmatrix}$$



c) **(4 points)** Write down the homogeneous 3x3 matrix that transforms the quadrilateral v_0, v_1, v_2, v_3 onto the quadrilateral v_0', v_1', v_2', v_3' in the figure below.



$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1/2 & 1 & 0 & 3/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & t \\ \frac{1}{2}t & 1 & t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -t \\ -t \\ 1 \end{pmatrix} = -t - t^2$$

$$\begin{pmatrix} 1 & 0 \\ 0{,}5 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 \\ 0{,}5 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \to \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 2 \end{pmatrix} \to \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix} \to \begin{pmatrix} 2 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 2 \\ 2 \end{pmatrix} \to \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

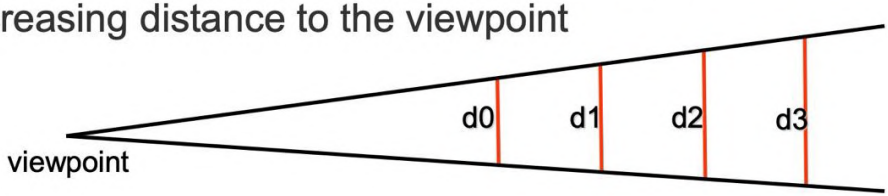
$\frac{1}{2}$

Sampling

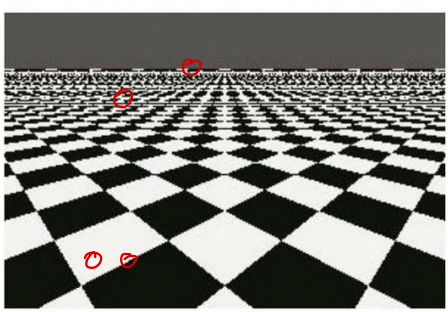
Aliasing - феномен, при котором разные сигналы становятся неразличимыми (похожими друг на друга), when sampled

alias - наложение

- Example: sampling frequency decreases with increasing distance to the viewpoint

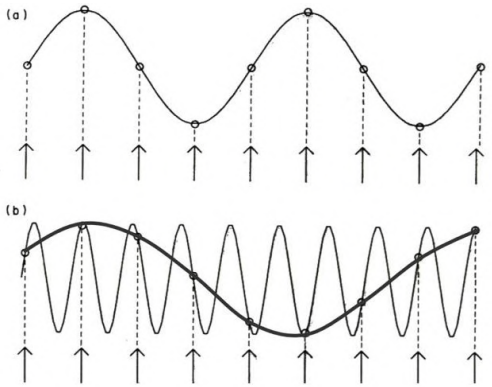


With increasing distance to the viewer and slope of the surface, an ever larger surface area is seen under a pixel, and only 1 fragment per pixel cannot represent that area well

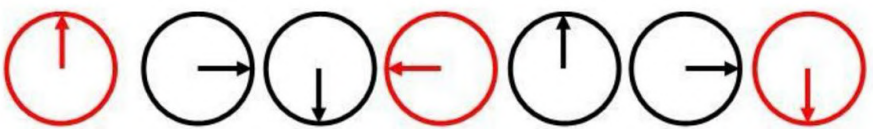


Sampling and Aliasing

- Aliasing artefacts
  - Spatial aliasing

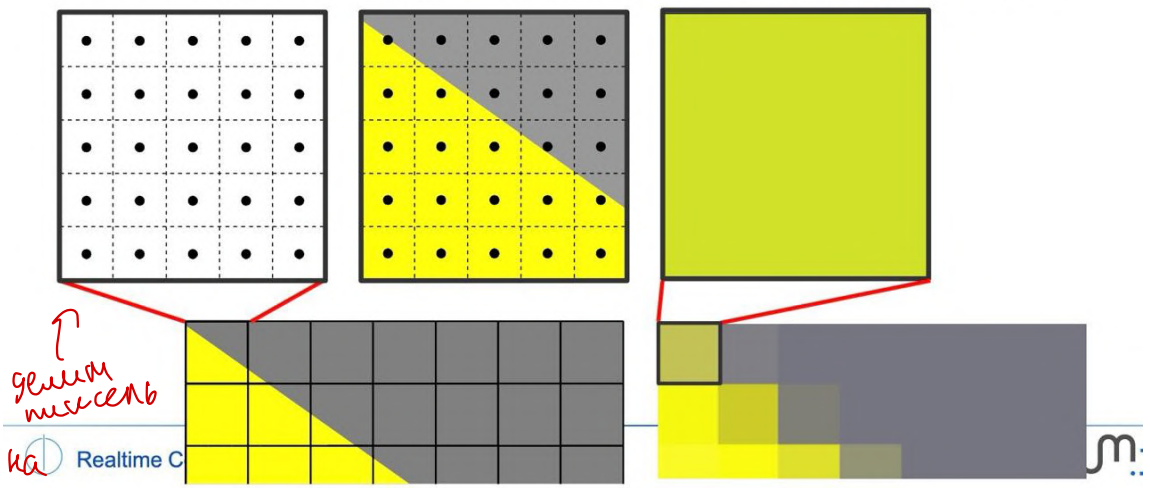


  - Temporal aliasing

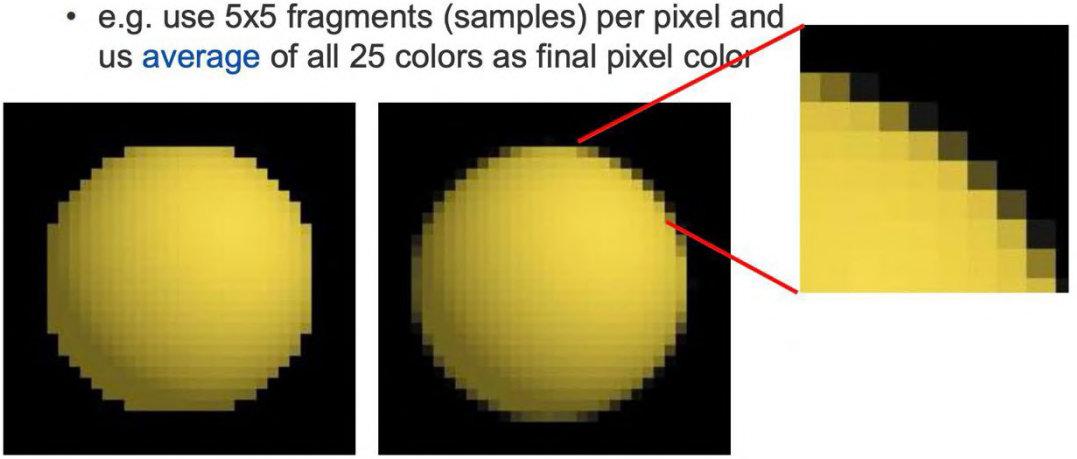


- How can we avoid aliasing caused by an undersampling of the signal (i.e. the sampling frequency is not high enough to cover all details) ?
  - Supersampling - increase sampling frequency
  - Prefiltering - decrease the highest frequency in the signal, i.e. filter the signal before sampling

- **Supersampling**: increase sampling density
  - Use more than one fragment per pixel, i.e. virtually increase the viewport resolution
  - Option 1: regular sampling
    - e.g. use 5x5 fragments per pixel, pixel color is average of 25 colors

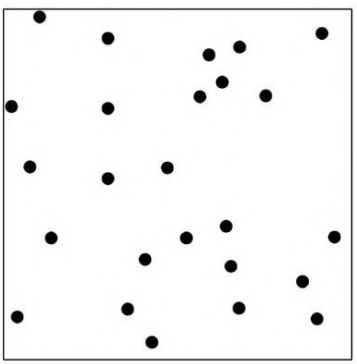


- e.g. use 5x5 fragments (samples) per pixel and us average of all 25 colors as final pixel color

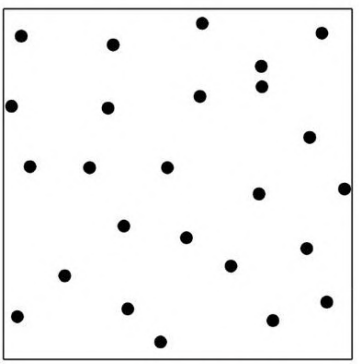


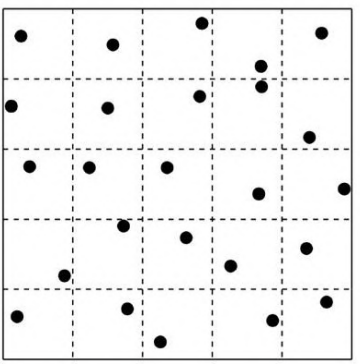
1 sample/pixel 16 samples/pixel

- **Supersampling**: increase sampling density
  - Option 2: jittered random sampling
    - Enforce a more even distribution of samples over pixel while maintaining randomness



25 random samples

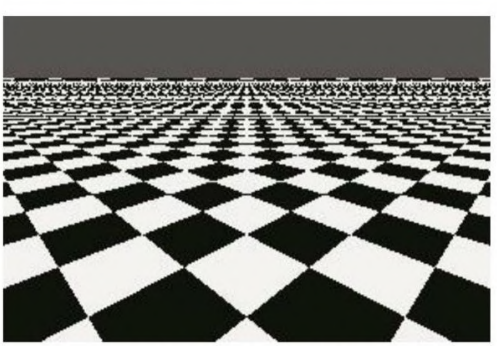




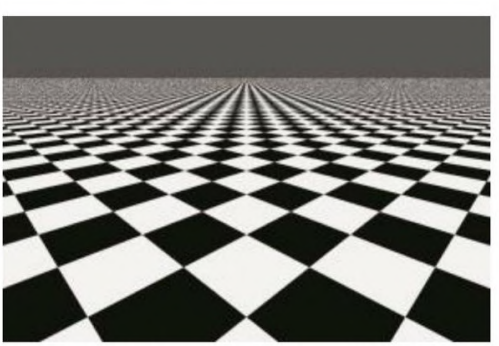
25 jittered samples (1 per grid cell)

Сначала делим на решётку, потом рандомно в каждую клеточку кладем пиксели

  - Example: use jittered sampling to counteract decreasing sampling frequency with increasing distance from viewer



1 sample/pixel



64 jittered samples/pixel

Sampling and Aliasing

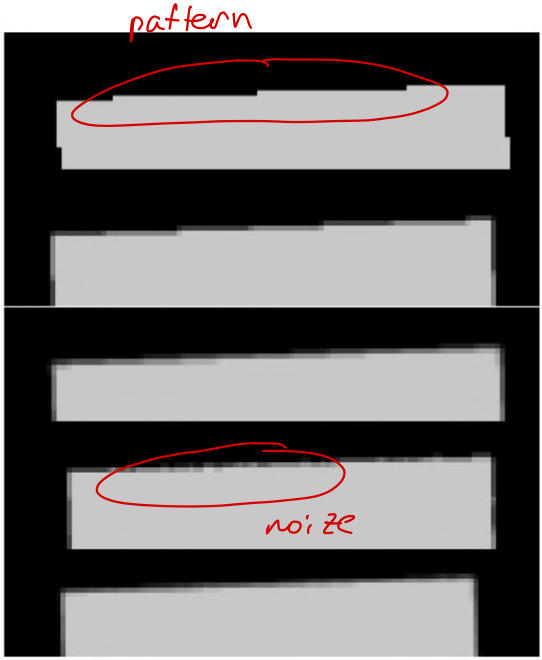
- Comparison
  - Regular, 1x1

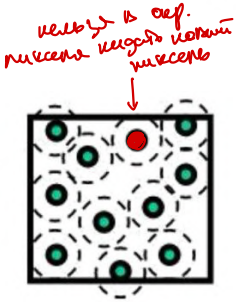
    Regular 3x3

    Regular, 7x7

    Jittered, 3x3

    Jittered, 7x7

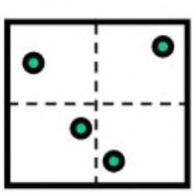




- Supersampling: increase sampling density
  - Option 2: jittered vs poisson-disk sampling

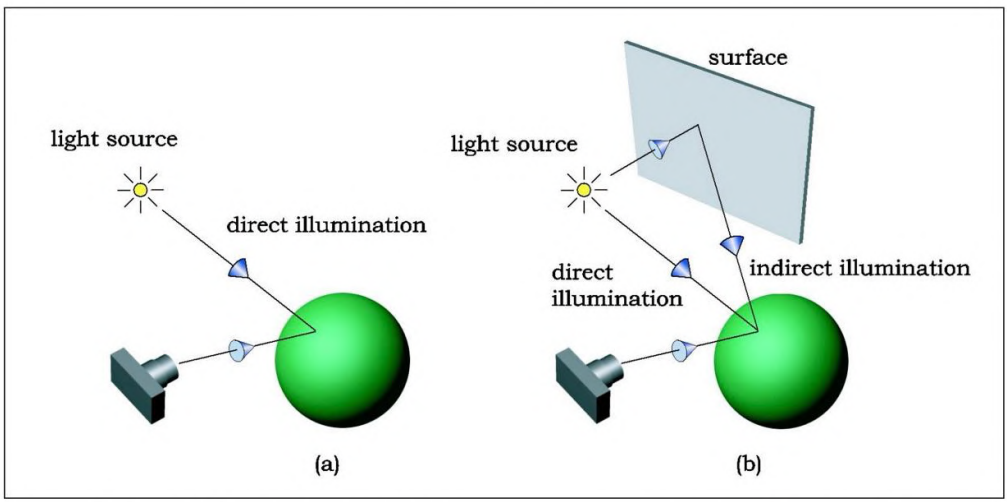
- Poisson-disk sampling
  - Random generation of samples with limit for the minimum distance between samples

- Jittered random sampling
  - Regular partitioning of pixel region
  - One random sample per partition



Ray-tracing

Direct (local) vs indirect (global) illumination

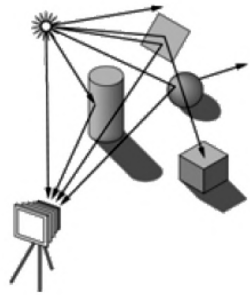


Rasterization-based rendering

„Ray-based“ rendering

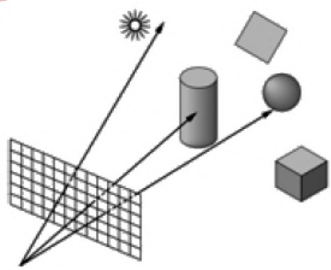
- Ray is a straight path along which light photons travel
- Multiple photons can travel along the same ray at the same time
- Photons originate at some light source and bounce around the geometry in the scene until they hit the eye
- If photons collide with objects they might get absorbed or they change their direction due to the laws of reflection

Basics



- Tracing photons forwards from each light source would lead to an excellent picture of the scene
  - But most photons will never hit the eye

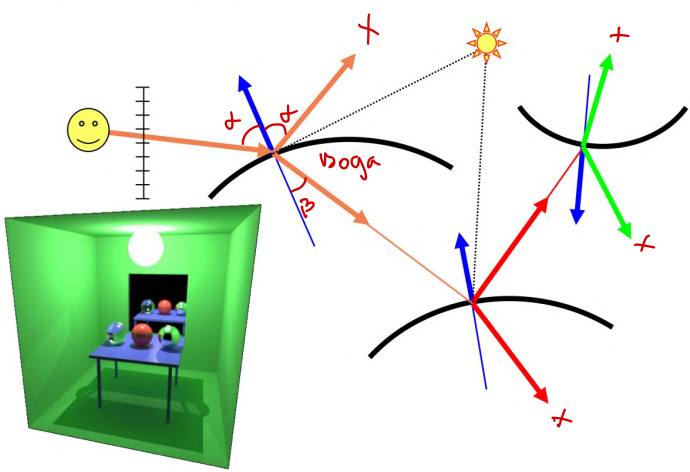
- Therefore, we trace photons backwards from the eye into the scene
  - Requires a sampling structure, i.e. a virtual pixel plane, to determine the rays to follow



ne egzistuojam diffuse ray spawn

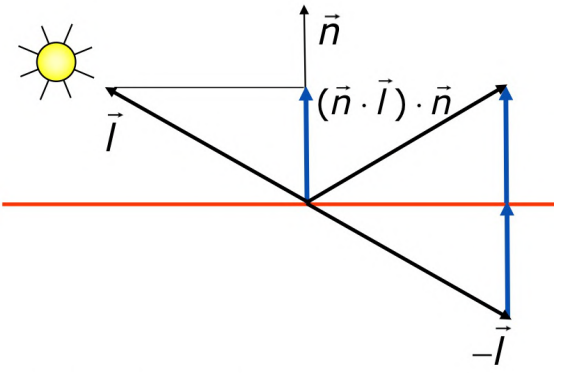
- The classical ray tracing method
  - Recursive computation of paths through the scene

Interaction with objects, i.e. absorption, specular reflection, refraction,



Point-wise evaluation of local illumination model and shadows

Reflection and refraction

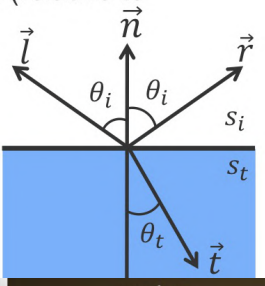


- Snell's law of reflection

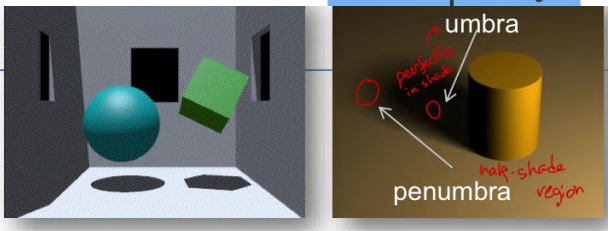
$$\vec{r} = 2(\vec{n}\cdot\vec{l})\cdot\vec{n} - \vec{l}$$

- Law of refraction (Snell)
  - s is the refractive index, i.e. change of propagation speed (relative to speed in vacuum)

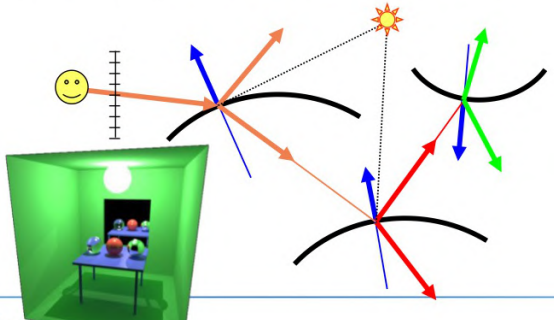
$$s_i \sin\theta_i = s_t \sin\theta_t$$



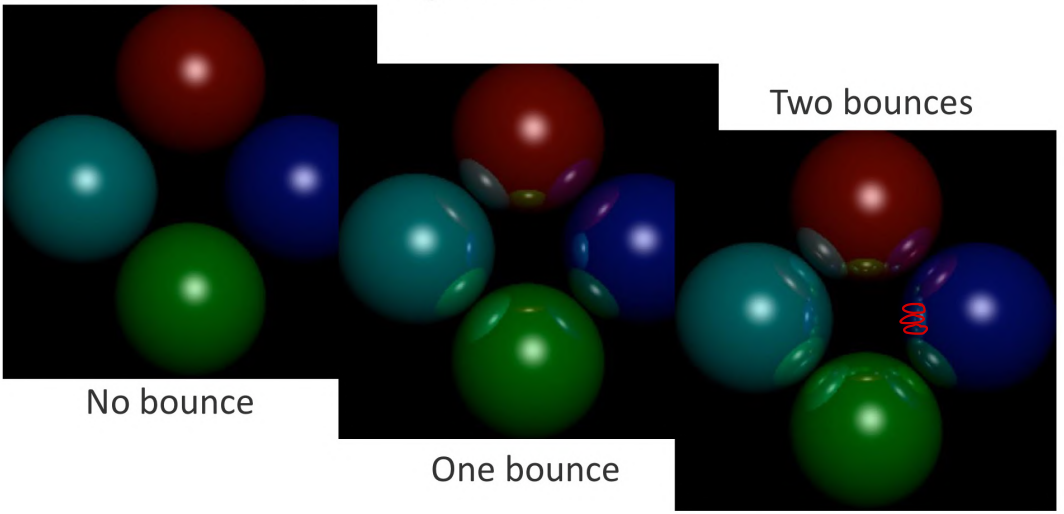
Basics



- The classical ray tracing method
  - The dotted lines represent shadow rays, which are shot to the light source
    - One (point light) or many (area light) shadow rays
  - If the surface point is in shade, nothing happens, else the local illumination from the light source is evaluated



- The basic ray tracing method



- Sampling the image plane with at least one ray per pixel

```
for (each pixel ) {
    ray = getRay(view point, pixel center)
    pixel color = traverse(ray)
}

traverse(ray) {
    compute_first_ray_object_intersection
    compute normal, reflection and refraction ray
    color  = evaluate_local_lighting_model(normal)
    return(color + k_s • traverse(reflect) + k_t • traverse(refract))
}
```

Attention: recursion termination

camera pos + t d

specular < 1

< 1

0,1 · 0,1 · 0,1 ... ≥ 0,001 else Stop

пока

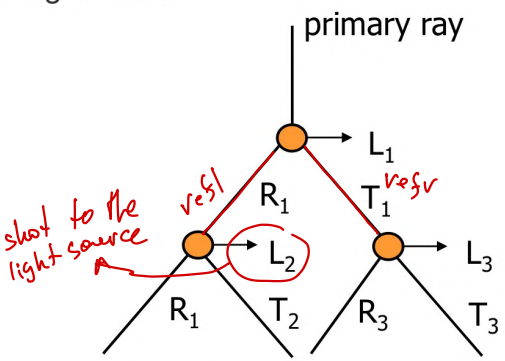
рекурс.

- Recursive ray traversal results in a tree-like computation order
  - Primary, secondary, tertiary etc. rays
  - One or multiple shadow rays to point light source
  - More work in branches but less contribution to the final image due to absorption

$I = I_{direct} + k_s I_{reflected} + k_t I_{refracted}$

$I_{direct} = I_{diffuse} + I_{specular}$

no ambient!



- Trace a ray through each pixel into the scene
- Reflect/refract at specular/transparent surfaces
- Continue until
  - Pure diffuse surface is hit (original Whitted raytracer) or
  - Energy drops below threshold because of attenuation or
  - The ray leaves the scene
- Local illumination from light sources:

$I_{direct} = I_{diffuse} + I_{specular}$

Diffuse reflectors

- Ray tracing of diffuse surfaces (in “Image Synthesis”)
  - Gather contributions from every visible point, not only light source
  - Need many samples in „all directions“
  - Use stochastic (Monte Carlo) techniques
    - Use average of randomly sampled incoming light
  - In combination with importance sampling
    - Shoot more rays into directions that contribute most

- Acceleration techniques for ray tracing
  - Keep in mind that the complexity of ray tracing is **O**(m x p), where m is the number of pixels and p is the number of objects in the scene
    - For each ray we have to check for intersections with every object
  - If we have polygonal objects and the entire scene consists of n polygons, the complexity is **O**(m x n)
    - For each ray we have to check for intersections with every polygon of every object

- Classification of acceleration techniques for ray tracing
  - Faster ray-object intersections
    - Efficient intersectors
  - Fewer ray-object intersections
    - Bounding volumes (boxes, spheres)
    - Space subdivision
  - Fewer rays
    - Adaptive tree-depth control
    - Stochastic sampling

- Faster ray-object intersections
- *A Fast Triangle-Triangle Intersection Test* (1997) Tomas Möller: Journal of Graphics Tools
  - Intersection point $R(t) = \mathbf{o} + t\mathbf{D}$ in barycentric coordinates u,v:

$$
\begin{aligned}
(1-u-v)V_0+uV_1+vV_2 &= o+tD \\
-tD+ uV_1-uV_0+vV_2-vV_0 &= o-V_0 \\
[-D, V_1-V_0, V_2-V_0][t,u,v]^T &= o-V_0
\end{aligned}
$$

- Ray-triangle intersection
  - Solving the system of equations (Cramers rule)

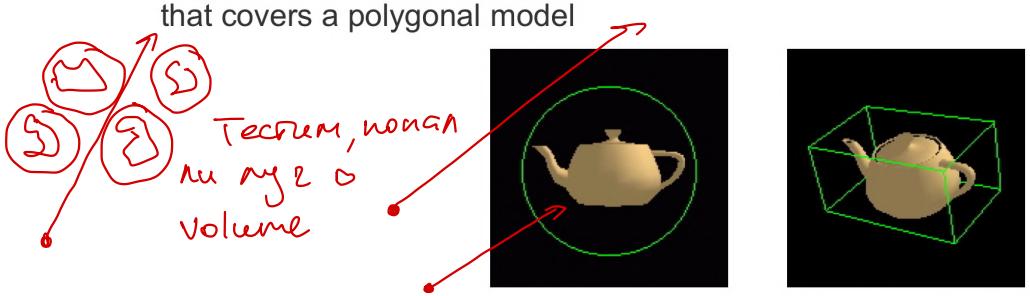
$$
\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{bmatrix} |T,\ E_1,\ E_2| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{bmatrix}
$$

$$
E_1 = V_1 - V_0,\ E_2 = V_2 - V_0 \text{ and } T = O - V_0
$$

$$
\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}
$$

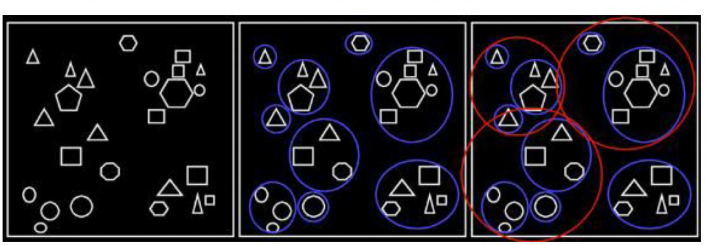
$$
\text{where } P = (D \times E_2) \text{ and } Q = T \times E_1
$$

- Fewer ray-object intersections
  - Bounding volumes (boxes, spheres)
    - A bounding volume is a simple volume description that is guaranteed to contain a more complex object, e.g. a sphere/box that covers a polygonal model

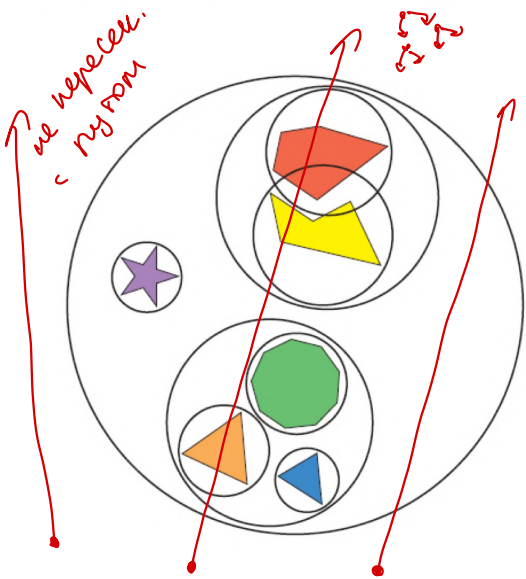


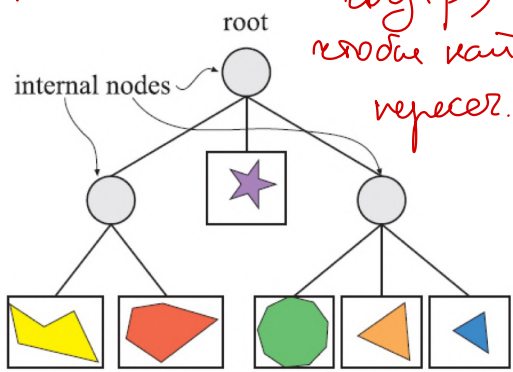
    - Test ray against bounding volume first, and only if it intersects this volume the ray is tested against the polygons

- Fewer ray-object intersections
  - Bounding volumes (boxes, spheres)
    - With simple bounding volumes, ray casting still requires O(p) intersection tests
    - Idea: use tree data structure
      - Larger bounding volumes contain smaller ones etc.
      - Often reduces complexity to O(log(p))



- Hierarchical bounding volumes



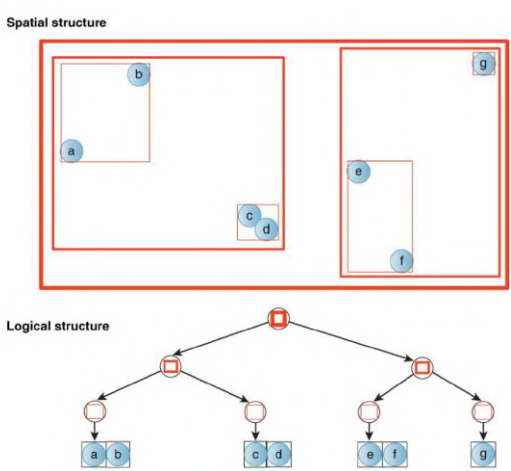


Тогда $\log(p)$ чтобы найти пересеч.

Два способа генерации:

- Сверху вниз
  Обходим.ки root всю сцену, потом выделяем кластеры, в них – более мелкие и тд.
- Снизу вверх
  Считаем volume для каждого объекта, а потом мерджим их

- Hierarchical bounding volumes



s et al. (2013)

- Fewer ray-object intersections using hierarchical bounding volumes
  - Ray intersection algorithm recursively descends tree
    - If ray misses bounding volume, no intersection
    - If ray intersects bounding volume, recurse with enclosed volumes and objects
    - Maintain near and far bounds to prune further
  - Overall effectiveness depends on model and constructed hierarchy

Akenine-Möller et

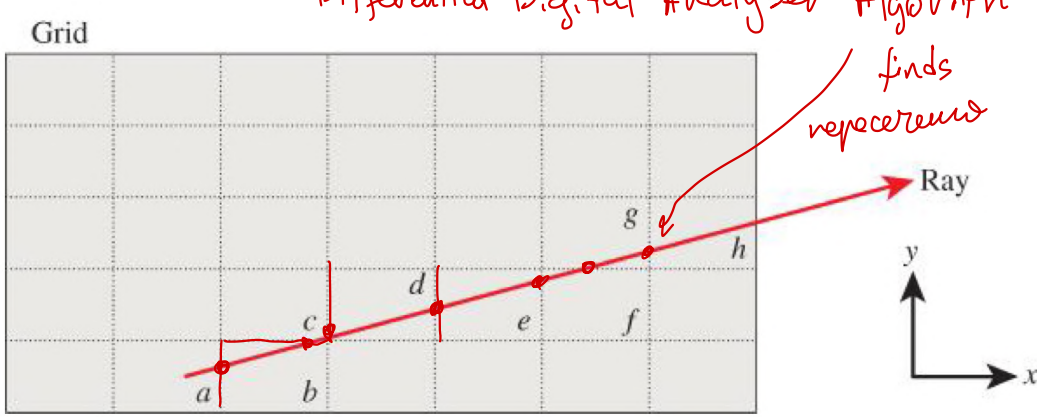
Проверяем все, потому что нужно взять тот, что ближе

Rule of thumb: иметь не больше 7 boxes в листьях

- Alternative to bounding volumes: divide space into disjoint sub-regions (i.e., into partitions)
  - For each region, store a list of objects/polygons contained in this region
  - At run-time, only test sub-regions that are hit
  - General problem of all spatial subdivision schemes: objects/polygons might get cut by region boundaries
  - Solution: either store objects/polygons in all regions they overlap, or cut into halves – increases number of polygons that have to be stored

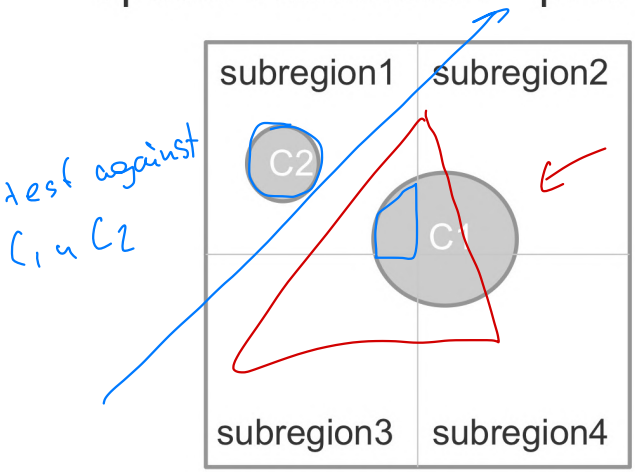
- Uniform spatial subdivision - grids
  - A 3D array of cells, a regular space partition
  - Rays step from cell to cell and test against the objects/polygons in each cell

Differential Digital Analyser Algorithm finds repeatedly



es et al. (2013)

- Spatial subdivision – problems



objectList: C1, C2

multiple references to the same object

partition1: C1 C2

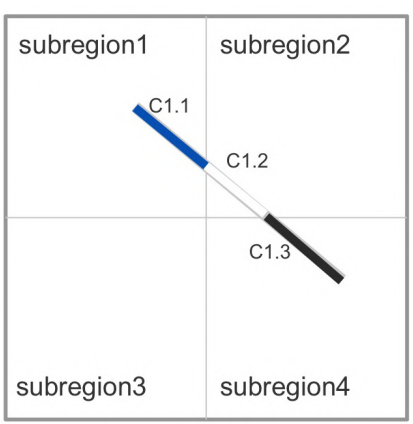
partition2: C1

partition3: C1

partition4: C1

  - Need to store 5 instead of 2 references
  - Per subregion: test against all (also exterior) object parts

- Spatial subdivision – problems



objectList: C1 // one line segment
// C1 is clipped at subregion boundaries

⇒ objectList: C1.1, C1.2, C1.3

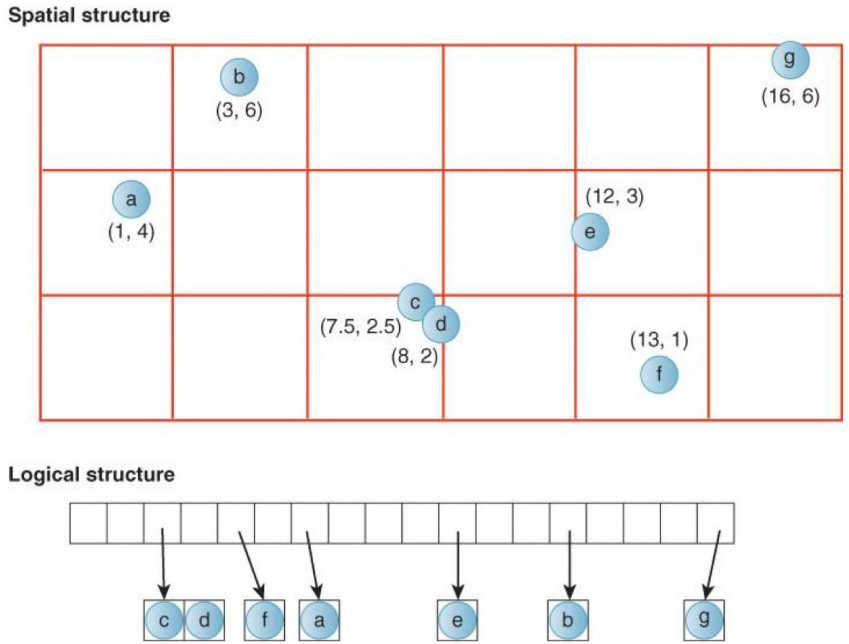
partition1: C1.1

partition2: C1.2

partition3:

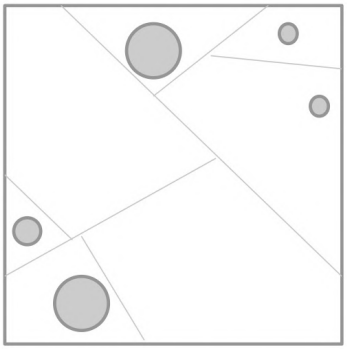
partition4: C1.3

  - Need to store 3 objects instead of 1

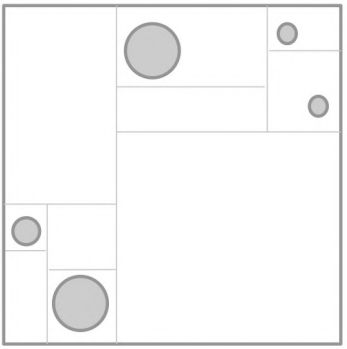


- Properties of regular grids
  - Grid traversal, i.e. from cell to cell, is very fast
  - Resolution of grid
    - Too low: too many polygons per cell
    - Too high: too many empty cells to traverse and store
  - Poor choice when geometry is clustered locally, i.e. many cells do not contain any polygon
  - Non-uniform spatial subdivision is more flexible
    - Can adjust to geometry “density”

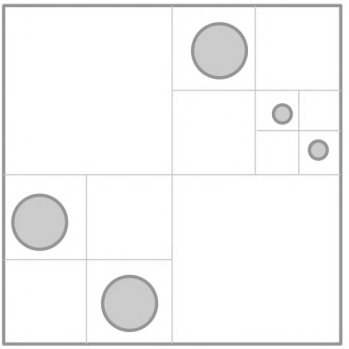
Non-uniform space subdivision



bsp-tree



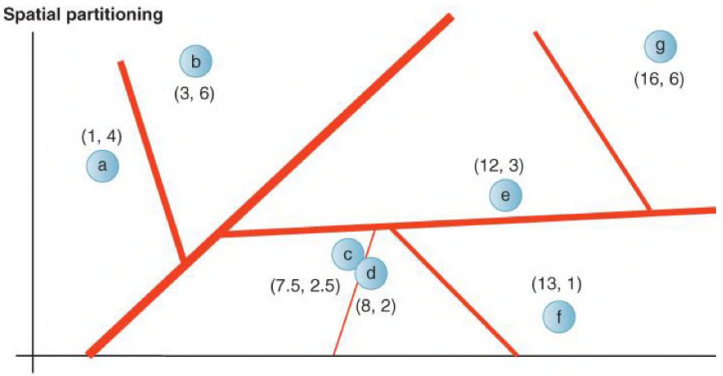
kd-tree

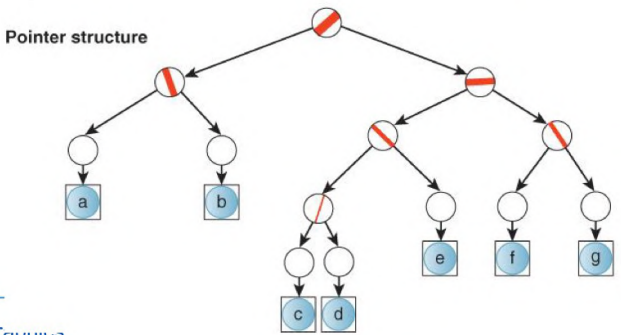


octree

A kd-tree is a specialization of a bsp-tree, where the partitioning planes are axis-aligned, can move arbitrarily, and splits occur in a fixed order, e.g., x – y – z.
In an octree, the partitioning planes are axis-aligned, always half the space, and splits occur simultaneously along x/y/z.

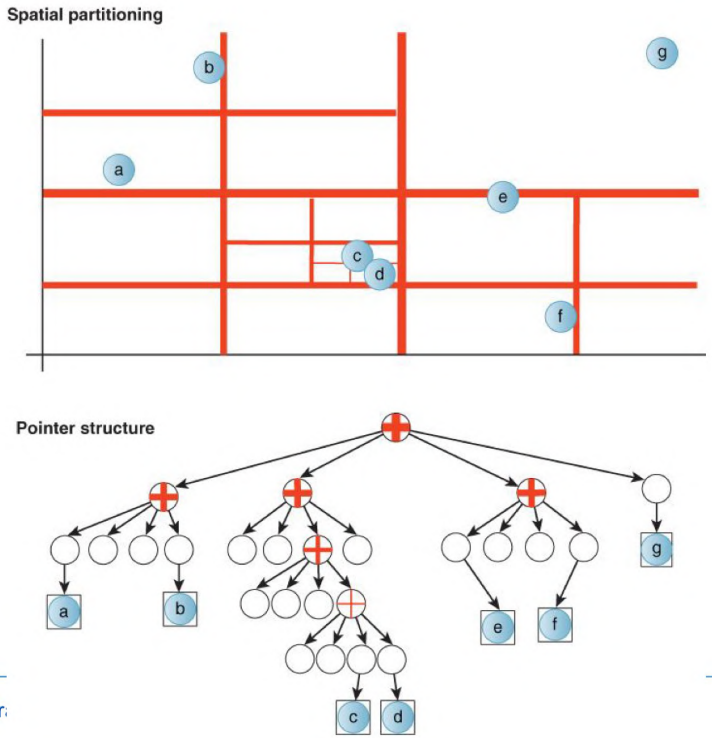
Binary space partitioning





ghes et al. (2013)

Quadtree subdivision

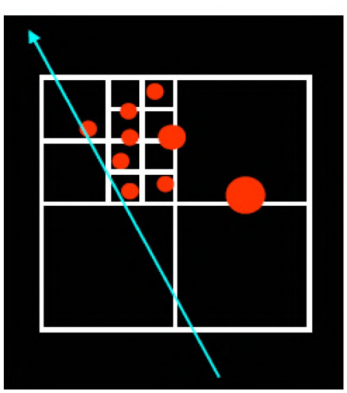


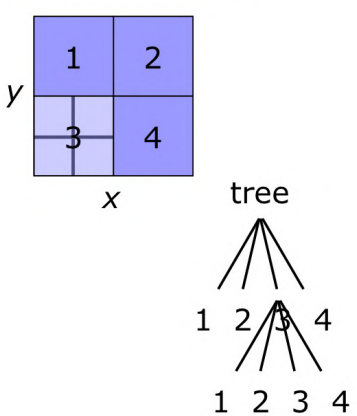
ghes et al. (2013)

Realtime Computer Gr

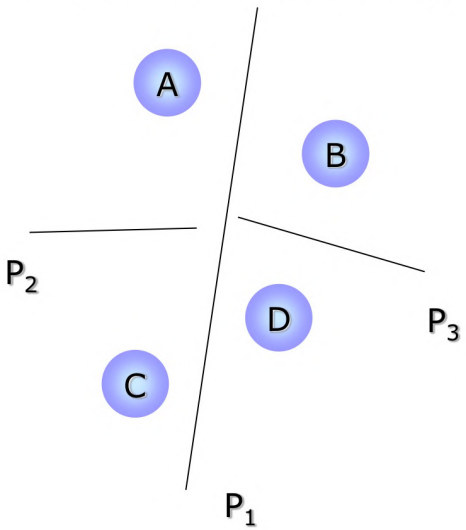
Non-uniform space subdivision

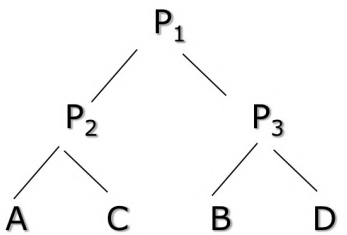
- Adaptive subdivision
  - Recursively test rays agains tree nodes at ever finer resolution levels





- Adaptive subdivision
  - Subdivision and the corresponding BSP-tree



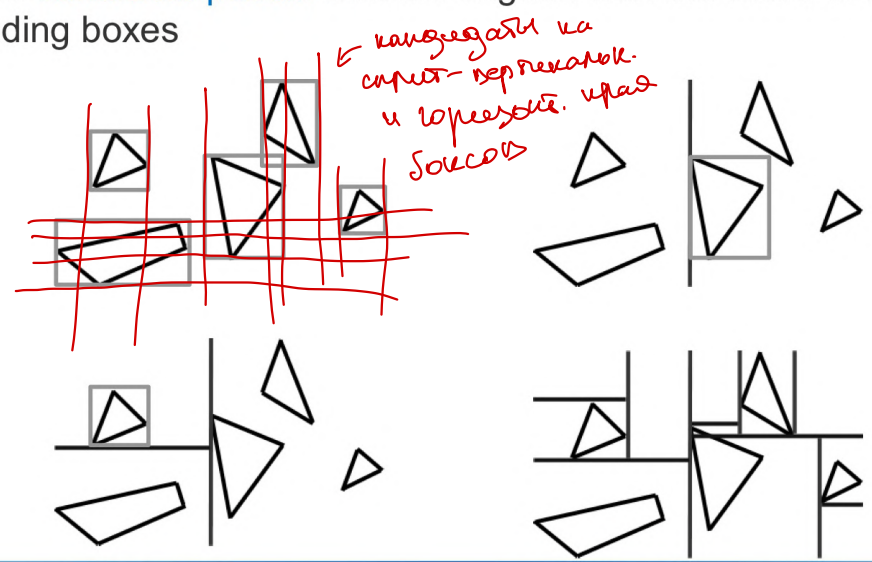


Each node corresponds to a partitioning plane

Leaf nodes store links to objects/polygons

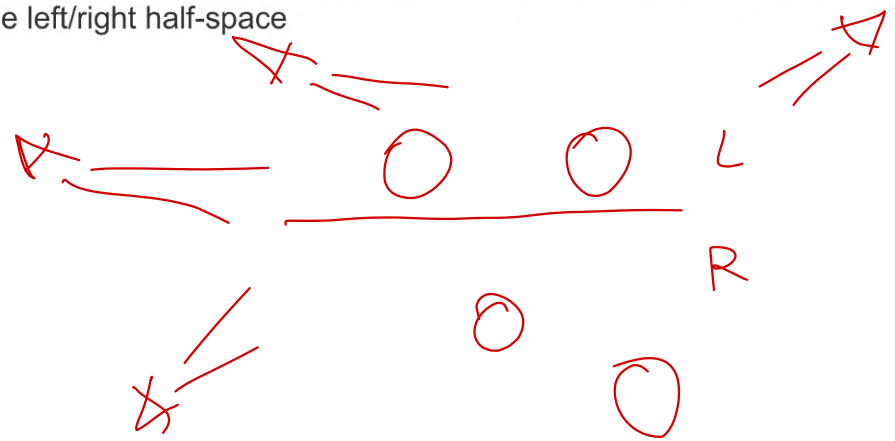
Bsp-tree (kd-Tree) construction

- There are infinitely many candidate planes
- Chose candidate planes that are aligned with the faces of object bounding boxes

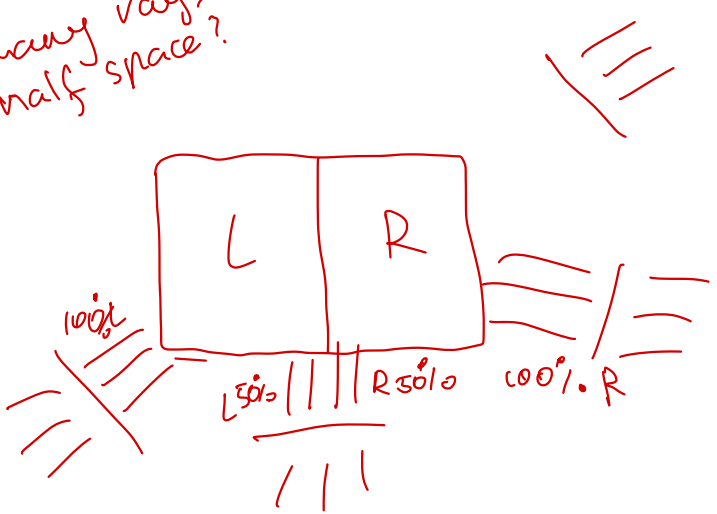


Стоимость выбора сечет-кандидата

- The cost function *C*
  - It should measure the work that has to be done if a certain plane is selected
    - The plane subdivides space into the “left” and “right” half-spaces
  - The work of one ray intersecting the left/right half-space is proportional to the number of objects in the left/right half-space
  - Question: of all rays intersecting the entire space, how many intersect the left/right half-space

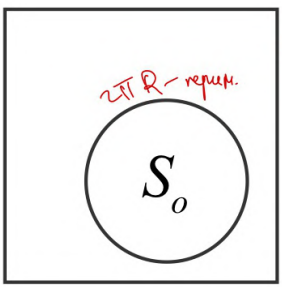


How many rays intersect the half space?



Binary space partitioning

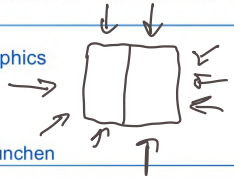
- The probability that an (arbitrary) ray *r* hits a convex shape that is completely inside a convex cell equals



$$p = [r \cap S_o | r \cap S_c] = \frac{\bar{A}_o}{\bar{A}_c} = \frac{S_o}{S_c}$$

Crofton's Theorem (convex objects): $\bar{A}=S/4$, where $\bar{A}$ is the average projected area in all directions and S is the surface area

S_i := surface area of *i*-th object



Binary space partitioning

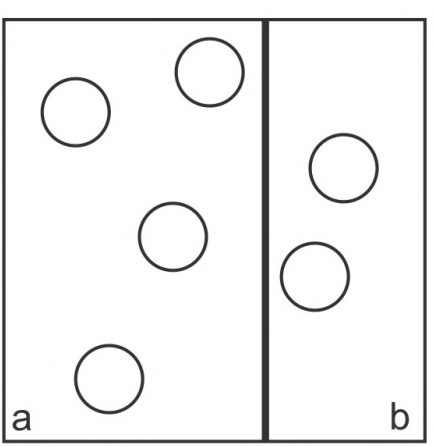
- Assuming intersection time t_i and traversal time t_t, number of objects in *a* and *b* is N_a and N_b, and

$$p_a = \frac{S_a}{S} \quad p_b = \frac{S_b}{S}$$

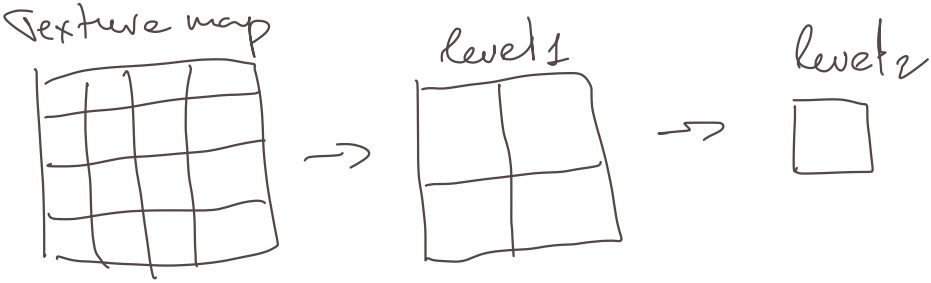
then

(Surface Area Heuristic)

$$C = t_t + p_a N_a t_i + p_b N_b t_i$$



Mipmapping



Trilinear interpolation:

LOD - level of detail → 1) Choose 2 mipmap levels (соседних)

2) Bilinear - на каждом mip level

3) Interpolate between

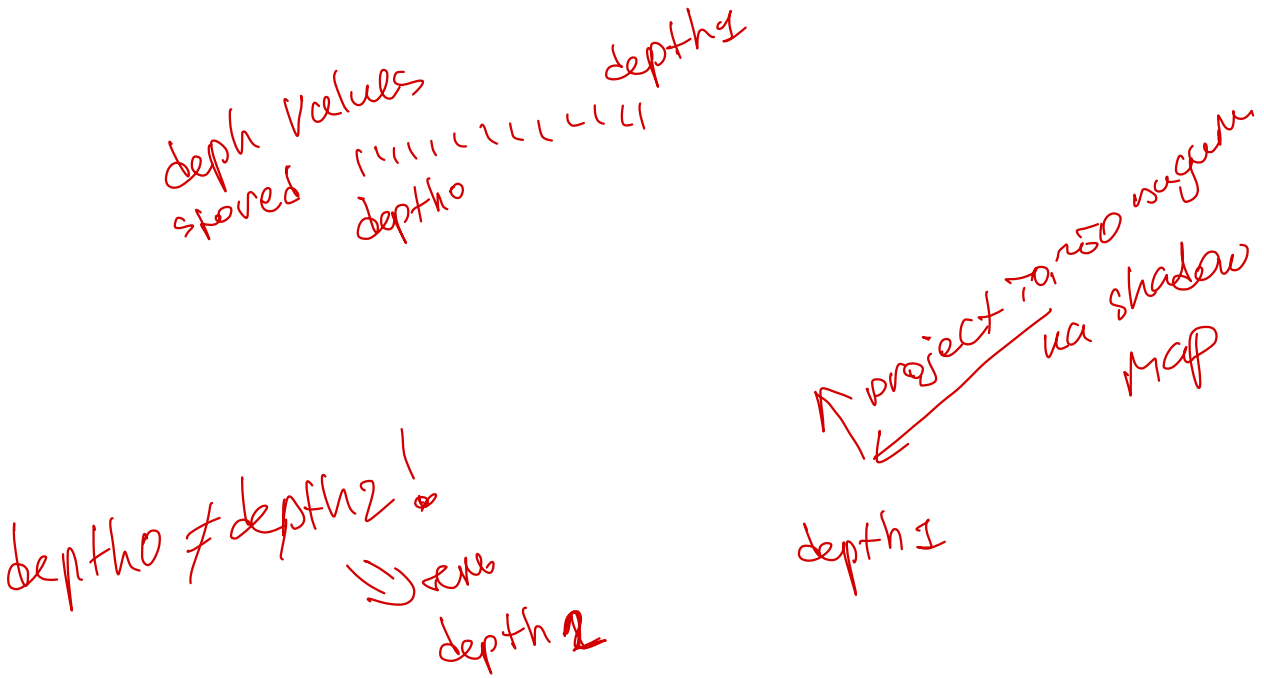
Anisotropic filtering

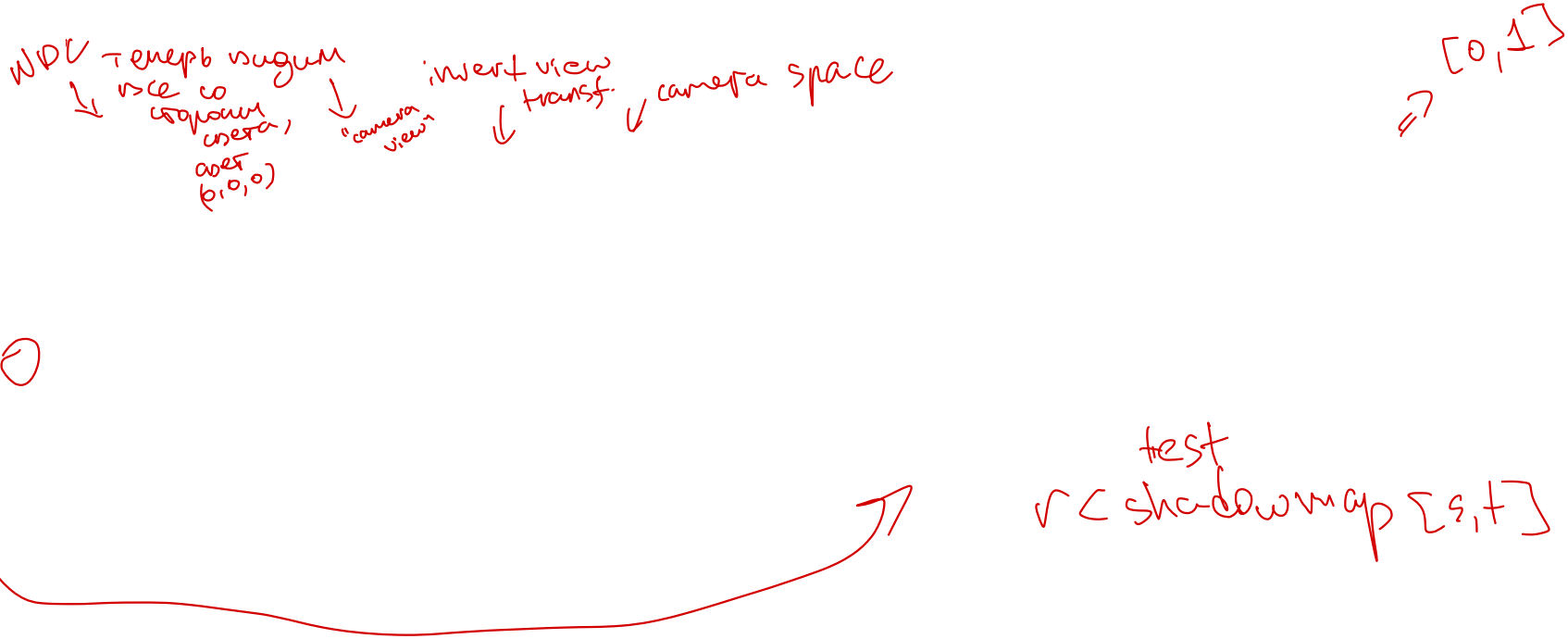
Исп. чтобы текстуры выглядели лучше под sharp angle углом

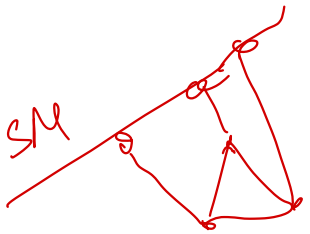
Потому что Mipmapping блюрит


смотрим на пиксель
шадинг
света
если не seen,
то
тень
rendering paths

(view plane)


depth Values
stored
depth0
depth1
project to sun
via shadow
map
depth0 ≠ depth2!
⇒ occ
depth 1
depth 2


WPC
теперь видим
все со
стороны
света,
свет
(0,0,0)
"camera
view"
invert view
transf.
camera space
[0,1]
test
r < shadowmap [s,t]



=> rasterizer interpolates vars from SM