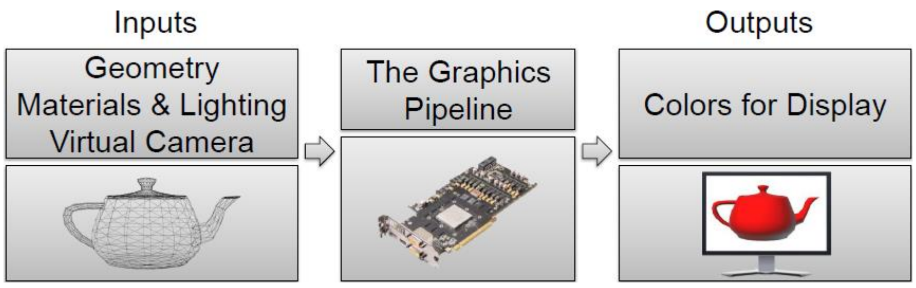
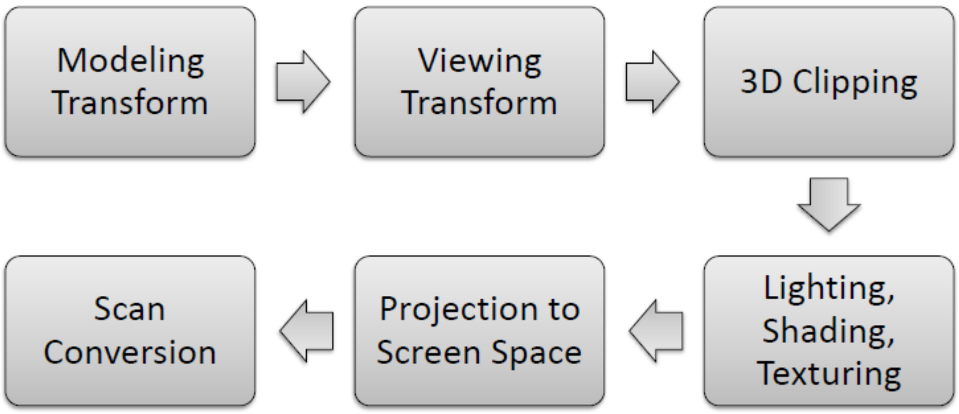
Kevin Wallimann wkevin@student.ethz.ch 16.02.2014

Computer Graphics

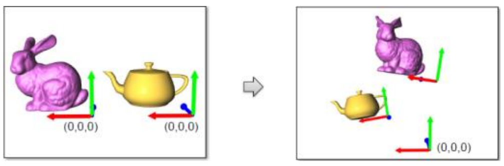


1 Standard Graphics Pipeline

The pipeline consists out of transformations

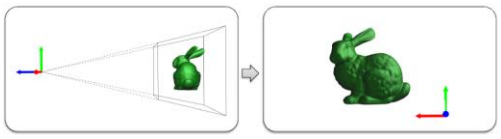


1.1 Modeling Transform



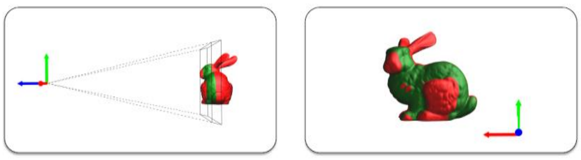
Coordinate Transformation from the object to the world space.

1.2 Viewing Transform



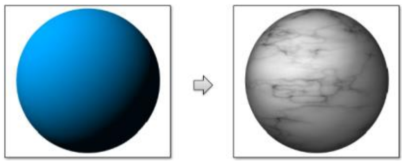
Coordinate Transformation from the world to camera space.

1.3 3D Clipping



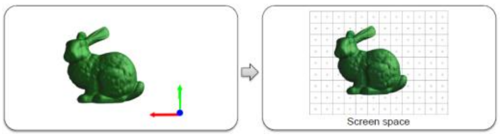
Remove parts of objects outside the frustum.
⇒ Avoid unnecessary computations and numerical instabilities.

1.4 Lighting, Shading and Texturing



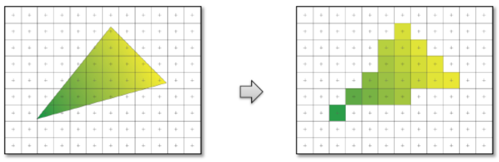
Compute color based on: lighting, materials and surface normal

1.5 Projection to Screen Space



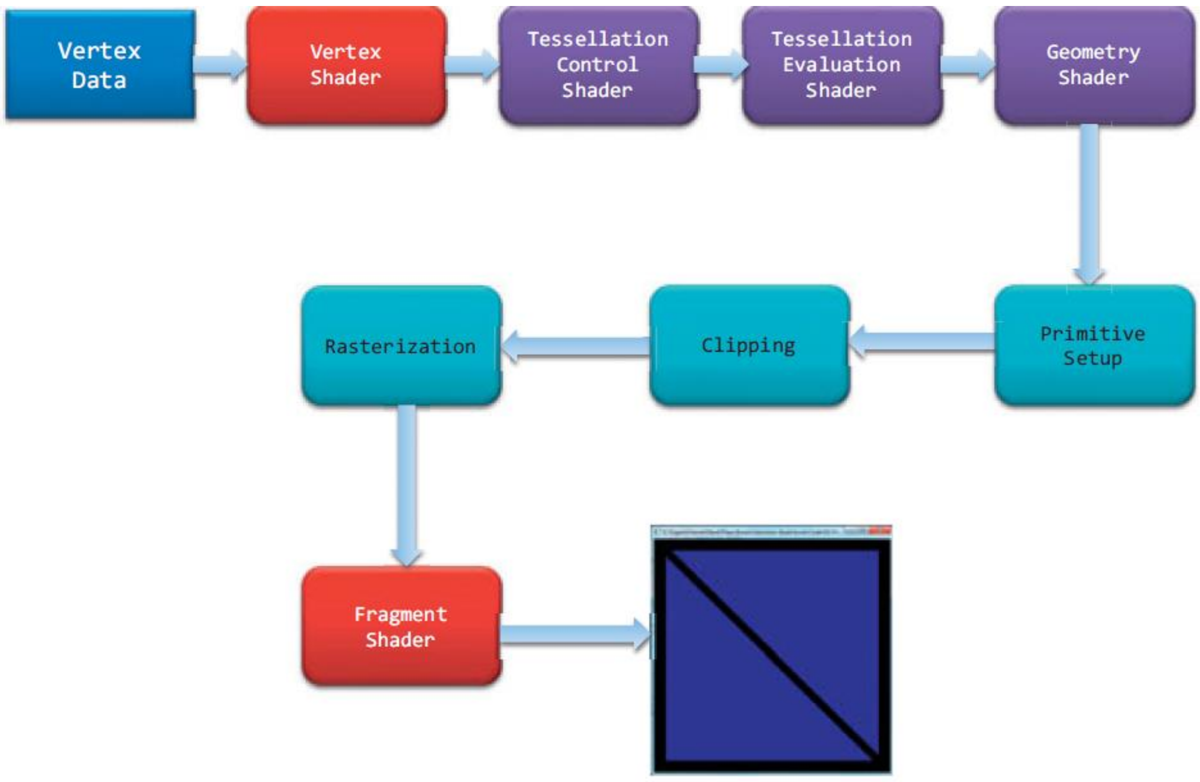
Project from 3D to 2D screen space

1.6 Scan Conversion



Compute fragment attributes and interpolate per-vertex values. For that discretize continuous primitives like triangles, lines and polygons.

2 OpenGL Graphics pipeline



2.1 Responsibilities

1. Vertex Shading (Processing vertex-associated data)
2. Tessellation Shading
3. Geometry Shading (Changing primitives)
4. Primitive Assembly
5. Clipping (Discarding of values outside the viewport)
6. Rasterization (Converting 3d description to 2d description, i.e. compute candidate pixels by interpolation, discard candidate pixels due to depth test)
7. Fragment shading (Texture mapping, final colouring)

- Geometric primitives are for example triangles, lines etc.
- The shading stages can be controlled using shaders (using GLSL)

2.2 Vertex shader

Input: Attributes given per vertex, such as color, surface normal, position, etc.

Output: Modified (or not) attributes per vertex.

2.3 Fragment shader

Input: Per-fragment information from the rasterization stage.

Output: Per-fragment color and depth value, etc. A fragment is not a pixel! A fragment can store color, position, depth, texture coordinates, etc.

3 Light and Colours

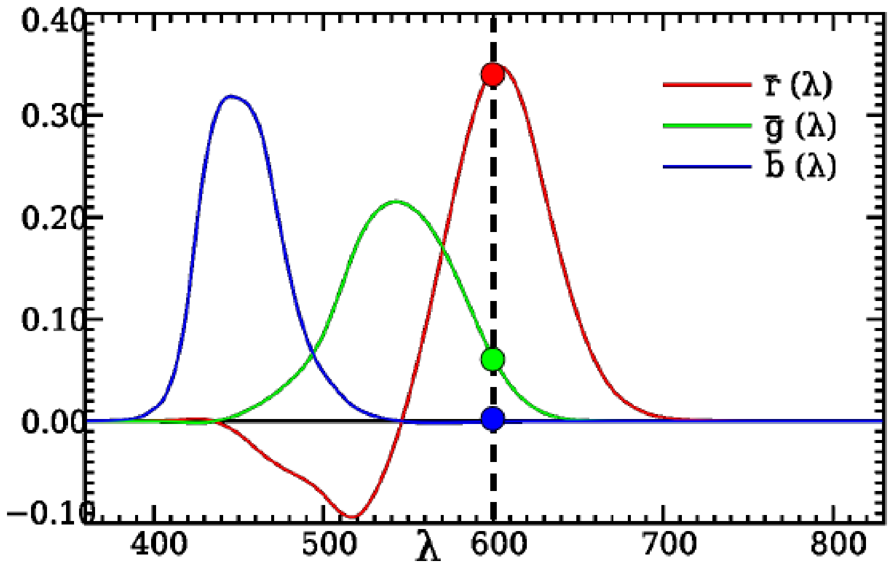
- Physical light: Composed of possibly infinitely many wavelengths. Physical light has infinitely many basis functions (one for each wavelength) and is thus infinite-dimensional.
- Spectral power distribution $P(\lambda)$: Intensity $P(\lambda)$ at wavelength λ
- Human vision: Fundamental input signal through only three types of cones (photoreceptors). Therefore, "human light" has only three basis functions r, g, b and is thus three-dimensional.
- Metamery: Many physical colours can correspond to the same human colour, since the mapping from $\mathbb{R}^\infty$ to $\mathbb{R}^3$ is of course not unique.

3.1 Colour spaces

The three basis functions can in principle chosen arbitrarily (or based on experiments).

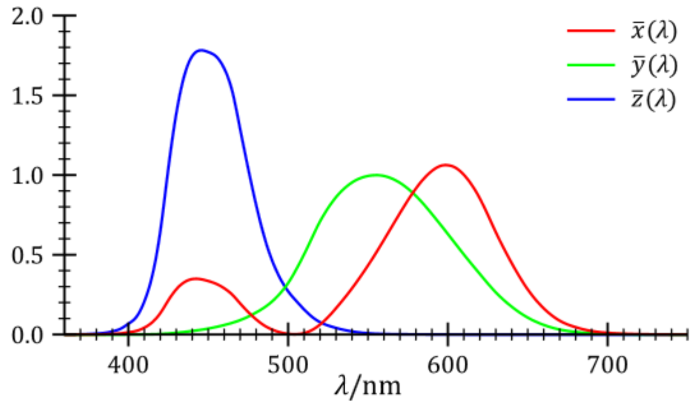
3.1.1 RGB

One example looks as follows (yielding the **CIE RGB** colour space):



- Experiment setup: A pure spectral colour was screened aside a linear combination of red (700nm), green (546nm and blue (435nm) colour. A human user had to adjust intensities of red, green and blue light the sources to match the pure spectral colour.
- Negative values of red: Some colours **cannot** be written as a linear combination of r, g, b In these cases, red was added to the test light.

3.1.2 CIE XYZ space



- Another set of nonnegative basis functions is the above $(\bar{x}, \bar{y}, \bar{z})$, resulting in the CIE XYZ space.
- $\begin{pmatrix} \bar{x}(\lambda) \\ \bar{y}(\lambda) \\ \bar{z}(\lambda) \end{pmatrix} = \begin{pmatrix} 2.36 & -0.515 & 0.005 \\ -0.89 & 1.426 & 0.014 \\ -0.46 & 0.088 & 1.009 \end{pmatrix} \begin{pmatrix} \bar{r}(\lambda) \\ \bar{g}(\lambda) \\ b(\lambda) \end{pmatrix}$
- CIE XYZ and RGB span the same space.

3.1.3 CIE xyY space

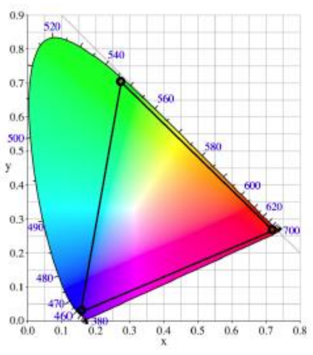
(x,y) characterize color, Y characterizes brightness.

$$X = \int_0^\infty P(\lambda)\bar{x}(\lambda)d\lambda$$
$$Y = \int_0^\infty P(\lambda)\bar{y}(\lambda)d\lambda$$
$$Z = \int_0^\infty P(\lambda)\bar{z}(\lambda)d\lambda$$

P denotes the color stimulus.

The x and y of the CIE xyY Color Space we get by normalization

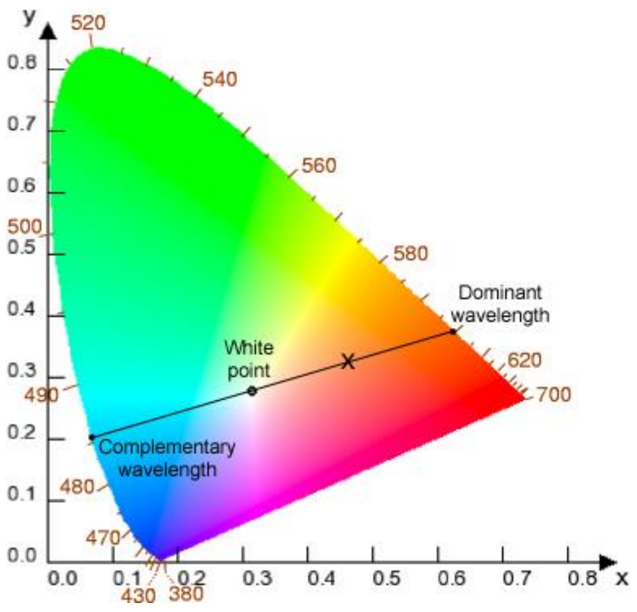
$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z}$$



The plot on the left shows the CIE chromaticity chart.

- Since this is a plot on a xy-plane, all colors have the same brightness.
- Primary colors along curved boundary
- Linear combination of two colors: line connecting two points
- Linear combination of three colors span a triangle (so called **color gamut**). The color gamut is a characteristic for electronic devices.
- RGB is a subspace of the CIE xyY space. (Therefore the chart is not true on a screen or normal printer, since the colors outside the RGB space are drawn with RGB colors)

one of this passage.



The CIE xyY chromaticity chart features some key quantities

White point: The white point is located at (x,y) = (1/3, 1/3)

Dominant / complementary wavelength of the color X: Intersections of the boundary with the straight line defined by the white point and the color X
It holds true that X is between the white point and the dominant wavelength, while the other intersection point is the complementary wavelength.

Non-spectral purples: The line between 700nm and 380nm is called the purple line. Its colors are composed of spectral colors only, i.e. no white or black portions, but they are not spectral colors.

3.1.4 CMY

$$(C, M, Y) = (1,1,1) - (R, G, B)$$

Cyan, Magenta, Yellow, (Black). Used for printers.

3.1.5 YIQ

- Y: Luminance, I: In-phase (orange-blue), Q: quadrature (purple-green)
- Spans the same space as RGB.
- Advantages for natural and skin colors.
- NTSC US-color TV standard

3.1.6 HSV

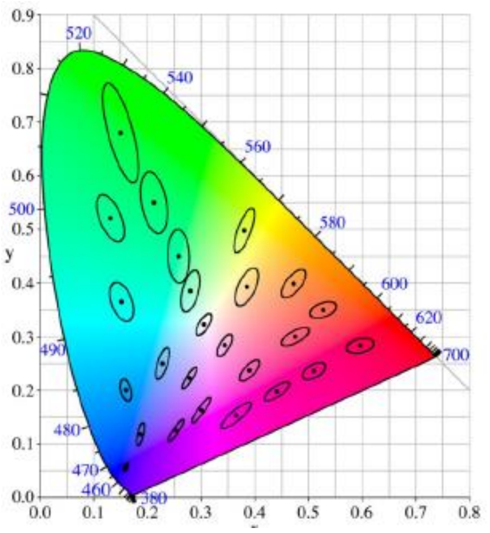
Hue, Saturation, Value (lightness, brightness)

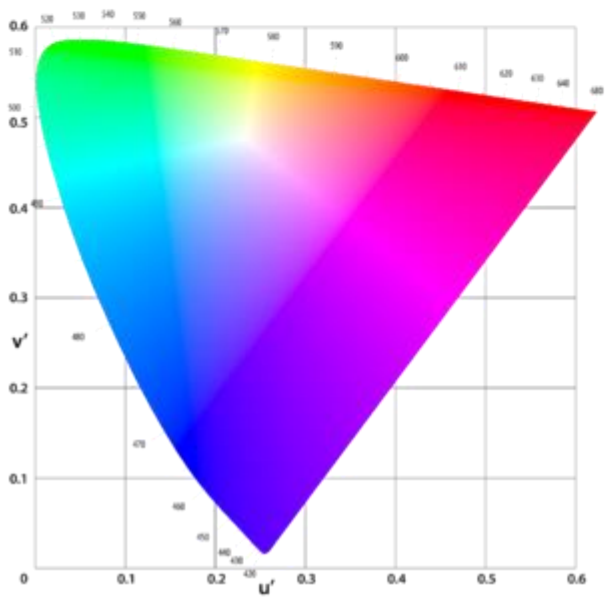
Spans the same space as RGB. In fact RGB cubes are projected to a hexagon.

$$V = \max(R, G, B)$$
$$S = \frac{(\max(R, G, B) - \min(R, G, B))}{\max(R, G, B)}$$
$$H = Hue(V, S, R, G, B)$$

3.1.7 CIELAB, CIELUV

Change CIE xyY such that distances in chart are perceptually uniform. (MacAdams ellipses)





4 Transformations

In the graphics pipeline we have to perform several transformations, such as change of position and orientation of objects, project objects to screen or animate objects.

4.1 2D Transformations

4.1.1 Definitions

A point or vector is represented as usual

$$\boldsymbol{p} = \begin{pmatrix} x \\ y \end{pmatrix}$$

and matrices just as **A**. And by a transformed point we mean

$$\boldsymbol{p}' = A\boldsymbol{p}$$

If the transformation is linear, it holds that

$$A(\alpha\boldsymbol{x} + \beta\boldsymbol{y}) = \alpha A(\boldsymbol{x}) + \beta A(\boldsymbol{y})$$

The same holds for matrices. We speak of a affine mapping if we encounter the form

$$A\boldsymbol{x} + \boldsymbol{b}$$

4.1.2 Transformation in normal Coordinates

4.1.2.1 Translation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \Leftrightarrow \boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{t}$$

4.1.2.2 Scaling

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \boldsymbol{p}' = S\boldsymbol{p}$$

4.1.2.3 Rotation

Rotation by angle θ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \boldsymbol{p}' = R\boldsymbol{p}$$

4.1.3 Homogeneous Coordinates

Affine maps can be linearised by taking homogeneous coordinates

$$\boldsymbol{p} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

A translation (T), scaling (S) or rotation (R) can therefore be written as

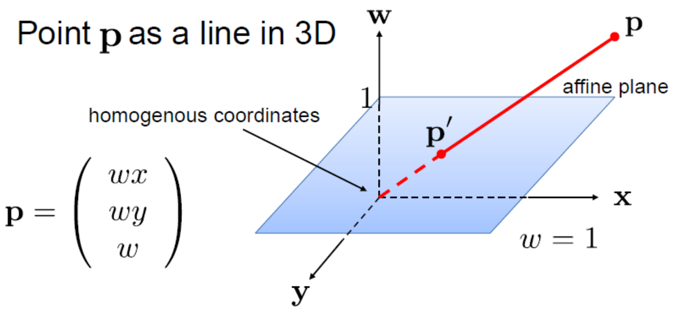
$$\boldsymbol{p}' = A\boldsymbol{p}$$

where A is either

$$T = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We are able to even shear the coordinates along the x- and/or y-axis

$$SH_x = \begin{pmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad SH_y = \begin{pmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Since we have now a 3D vector for an 2D point, a point can be regarded as having infinitely many homogeneous coordinates for any ω.

4.1.3.1 Combining Transformations

We can of course combine multiple transformations by multiplying them, like a rotation followed by a translation as TR. The order plays a role since not every combination of matrices commutes!

4.2 3D Transformations

In 3D the same concepts hold. For the homogeneous coordinates we now need a 4D point

$$\boldsymbol{p} = \begin{pmatrix} x \\ y \\ z \\ \omega \end{pmatrix} \xRightarrow[hyperplane]{} \boldsymbol{p} = \begin{pmatrix} x/\omega \\ y/\omega \\ z/\omega \\ 1 \end{pmatrix}$$

By setting the 4th variable to 1 we project this point in 3D to a hyperplane. Here we have again some matrices for translation (T) and scaling (S)

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

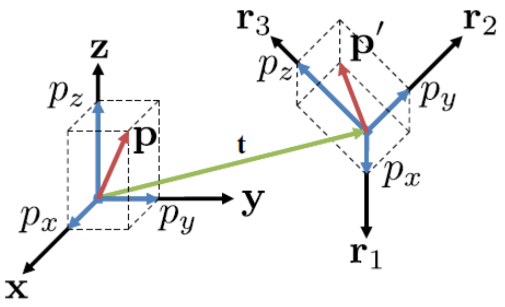
for rotation (R) it looks a bit more complicated, since it depends on the axis on which we are rotating

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We could also take an arbitrary axis **u** which leads to $R(\boldsymbol{u}, \theta)$. Shearing in respect to an principal plane would take the form like

$$SH_{xy} = \begin{pmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ or } SH_{xz} = \begin{pmatrix} 1 & sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4.2.1 Coordinate Systems



A point can be of course represented as a linear combination of orthonormal basis vectors

$$\boldsymbol{p} = p_x\boldsymbol{x} + p_y\boldsymbol{y} + p_z\boldsymbol{z}$$

To change the coordinate system by a rotation and a translation we simply can use R and T like before which looks in vector form like

$$\boldsymbol{p}' = \begin{pmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & \boldsymbol{r}_3 & \boldsymbol{t} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Mind that the $\boldsymbol{r}_i$ and $\boldsymbol{t}$ are vectors. So it's a 4x4 transformation matrix! The $\boldsymbol{r}_i$ correspond to the rotation and $\boldsymbol{t}$ therefore to the transformation.

4.2.2 Transforming Normal Vectors

Think about any surface and a specific normal vector on it. This normal vector has a tangent plane which can by described with 4 parameters

$$\boldsymbol{p} = (x, y, z, 1) \Rightarrow Ax + By + Cz + D = 0 \Leftrightarrow \boldsymbol{n}^{\boldsymbol{T}}\boldsymbol{p} = 0$$

Then the normal $\boldsymbol{n}$ can be written as

$$\boldsymbol{n} = (A, B, C, D)$$

What's the normal to a transformed $\boldsymbol{p}$ after applying the transformation Q

$$n^T\left(\underbrace{Q^{-1}Q}_{I}\right)p = 0$$

$$\left(\underbrace{Q^{-T}n}_{n\prime}\right)^T\left(\underbrace{Qp}_{p\prime}\right) = 0$$

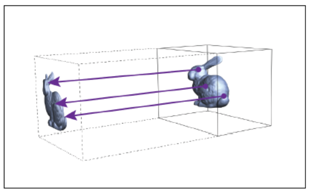
Therefore, the transformation matrix must be inversed and transposed for the normal.

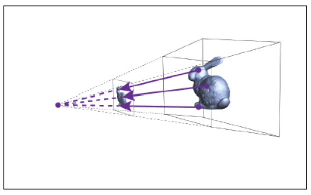
$$\boldsymbol{p}' = Q\boldsymbol{p} \;\Rightarrow\; \boldsymbol{n}' = (Q^{-1})^T\boldsymbol{n}$$

4.3 Projection

It's about reducing dimensionality. In our case we go from 3D to 2D. We distinguish two types of projection

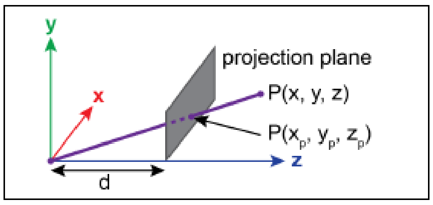
- parallel projection
- perspective projection





4.3.1 Perspective Projection

Let's assume a projection plane that lies in the x/y-plane. It's distance to (0,0,0) is d (0,0,d). So for a arbitrary point $\boldsymbol{p}$ it holds that



$$\boldsymbol{p}_{projected} = \begin{pmatrix} x_p \\ y_p \end{pmatrix} = \begin{pmatrix} x \cdot \frac{d}{z} \\ y \cdot \frac{d}{z} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \cdot \frac{d}{z}$$

So in homogeneous coordinates the transformation looks as follows

$$M_{per}\boldsymbol{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix}$$

To see why this is a perspective projection like described above we have to divide the first 3 components by the 4th to get the 3D coordinates. Normally the 4th component is 1 so we don't care. In this case it is important

$$\boldsymbol{p}_{3D} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \frac{1}{z/d} = \begin{pmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \end{pmatrix}$$

So we see that the first 2 entries correspond to $\boldsymbol{p}_{projected}$. So M_{per} gives us the right result.

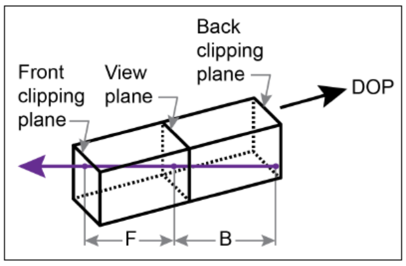
4.3.2 Parallel Projection

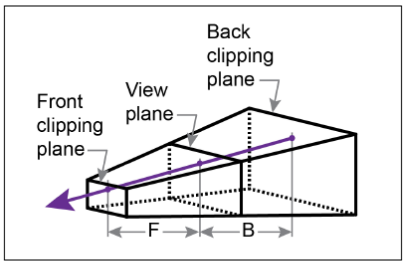
The parallel projection is slightly simpler. It just leaves out the z dimension for a projection plane that is in the x/y-plane

$$M_{ort} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

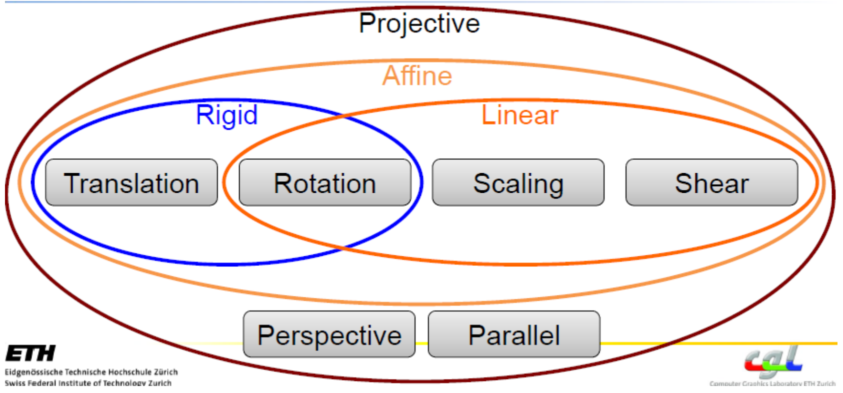
4.3.3 Clipping

For our purposes we have one view plane where we project things on to it. To reduce the computations we have also a front and a back plane. Only objects within this volume gets projected onto the view plane. This is called clipping. Also here we differentiate between parallel and projection clipping



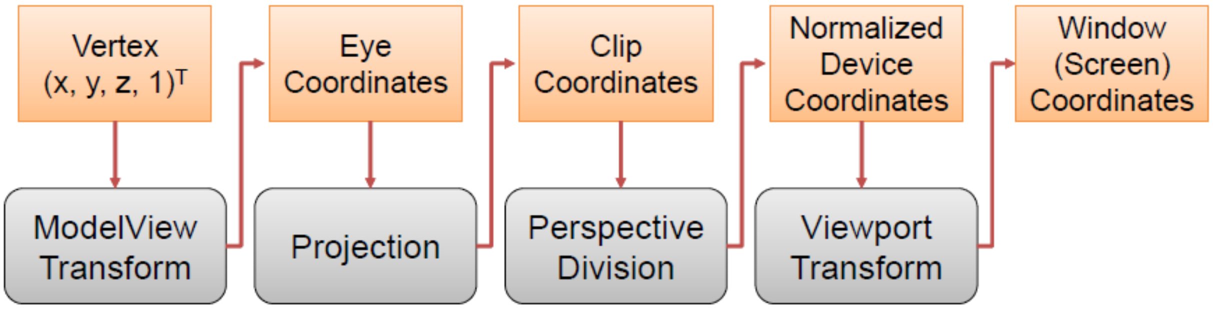


4.3.4 Overview


Projective
Affine
Rigid
Linear
Translation
Rotation
Scaling
Shear
Perspective
Parallel
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
cgl Computer Graphics Laboratory ETH Zurich

4.4 Transformation in OpenGL

In OpenGL many transformations happen



4.4.1 ModelView Transform

First we have to transform the object from the model coordinates (***m***) to the camara coordinates (**c**) which we see. First we transform therefore into world coordinates (***w***) of our scene. This is the same like transforming from one coordinate system to another as described above

$$\begin{pmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & \boldsymbol{r}_3 & \boldsymbol{t} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_x \\ m_y \\ m_z \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{t} + m_x \boldsymbol{r}_1 + m_y \boldsymbol{r}_2 + m_z \boldsymbol{r}_1 \\ 1 \end{pmatrix} = \begin{pmatrix} w_x \\ w_y \\ w_z \\ 1 \end{pmatrix}$$

In a second step we transform now to camara coordinates

$$\begin{pmatrix} \boldsymbol{left} & \boldsymbol{up} & -\boldsymbol{dir} & \boldsymbol{eye} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_x \\ c_y \\ c_z \\ 1 \end{pmatrix} = \begin{pmatrix} w_x \\ w_y \\ w_z \\ 1 \end{pmatrix}$$

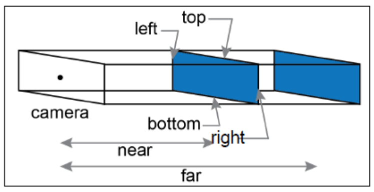
where we have

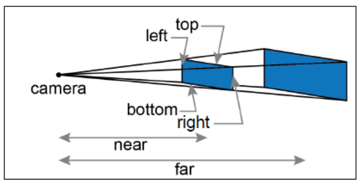
$$\begin{aligned} \boldsymbol{left} &= (1 \quad 0 \quad 0)^{\boldsymbol{T}} \\ \boldsymbol{up} &= (0 \quad 1 \quad 0)^{\boldsymbol{T}} \\ \boldsymbol{dir} &= (0 \quad 0 \quad -1)^{\boldsymbol{T}} \\ \boldsymbol{eye} &= (0 \quad 0 \quad 0)^{\boldsymbol{T}} \end{aligned}$$

as default in OpenGL. We are now in the eye coordinates.

4.4.2 Projection

Now the projection is done. We can make an perspective or an parallel (ortho) projection of the scene. Thereby we also perform the clipping. For this process OpenGL holds 2 functions, where we pick the one which we need in the situation





`glOrtho(left, right, bottom, top, near, far);` `glFrustum(left, right, bottom, top, near, far);`

$$\begin{bmatrix} c'_x \\ c'_y \\ c'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_z \\ 1 \end{bmatrix} \quad \begin{bmatrix} c'_x \\ c'_y \\ c'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2 \cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2 \cdot near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_z \\ 1 \end{bmatrix}$$

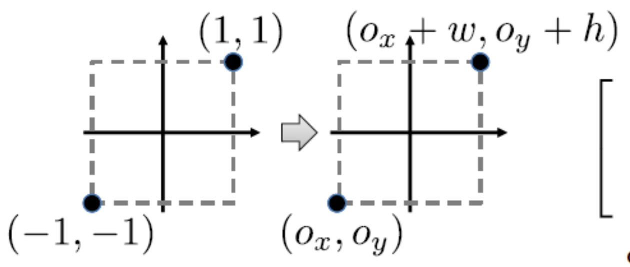
So with *glOrtho()*, respectively *glFrustum()* we can perform the projection. Clipping is now performed by comparing to w (4th coordinate of) which is in our case 1. So we take only the coordinates c'_x, c'_y and c'_z within ±w.

4.4.3 Perspective Division

It is yielded by dividing the clip coordinates by *w*. It is called *perspective division*. It is more like window (screen) coordinates, but has not been translated and scaled to screen pixels yet. The range of values is now normalized from -1 to 1 in all 3 axes.

4.4.4 Viewport Transform

We go now from the normalized device coordinates to the screen coordinates.



$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{w}{2} d_x + (o_x + \frac{w}{2}) \\ \frac{h}{2} d_y + (o_y + \frac{h}{2}) \\ \frac{f-n}{2} d_z + \frac{f+n}{2} \end{bmatrix}$$

`glViewport(`o_x`, `o_y`, w, h);`
`glDepthRange(n, f);`

glViewport() command is used to define the rectangle of the rendering area where the final image is mapped. Therefore we need the origin of the display (ox,oy), the width w and the height h.
And, **glDepthRange()** is used to determine the *z* value of the window coordinates. Here n means near and f far. The window coordinates are computed with the given parameters of the above 2 functions.

4.5 Quaternions

4.5.1 Definition

$$\boldsymbol{q} = a + ib + jc + kd = s + \boldsymbol{v}$$

With $s = a, \boldsymbol{v} = (b, c, d)$

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

4.5.2 Addition

$$a_1 + ib_1 + jc_1 + kd_1 + a_2 + ib_2 + jc_2 + kd_2 = (a_1 + a_2) + i(b_1 + b_2) + j(c_1 + c_2) + k(d_1 + d_2)$$

4.5.3 Magnitude

$$\|\boldsymbol{q}\| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

4.5.4 Unit quaternion

$$\frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}$$

4.5.5 Multiplication

$$\boldsymbol{q_1 q_2} = (s_1 + \boldsymbol{v_1})(s_2 + \boldsymbol{v_2}) = s_1 s_2 - \boldsymbol{v_1} \cdot \boldsymbol{v_2} + s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_1} + \boldsymbol{v_1} \times \boldsymbol{v_2}$$

$$\boldsymbol{q_1 q_2} \neq \boldsymbol{q_2 q_1}$$

4.5.6 Conjugate and inverse

$$\boldsymbol{q} = s + \boldsymbol{v}$$
$$\bar{\boldsymbol{q}} = s - \boldsymbol{v}$$

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

4.5.7 Rotation

u: Rotation axis, 3d

θ: Rotation angle

p: Some point in space, quaternion

For any position $(x, y, z) \in \mathbb{R}^3$, the corresponding quaternion is $q = 0 + ix + jy + kz$

The rotation around u by angle θ of the point p is defined as

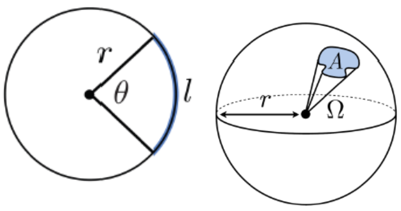
$$p' = qpq^{-1}$$

Where $q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) u$ and $q^{-1} = \cos\left(\frac{\theta}{2}\right) - \sin\left(\frac{\theta}{2}\right)$

5 Lighting and Shading

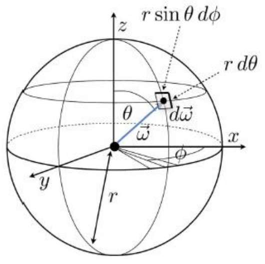
5.1 Definitions

- Angle: $\theta = \frac{r}{l}$
- Solid angle: $\Omega = \frac{A}{r^2}$



A point ω on the unit sphere can easily be written with the zenith θ and azimuth ϕ: $\omega = \omega(\theta, \phi)$

$$\omega_x = \sin\theta\cos\phi\,, \;\; \omega_y = \sin\theta\sin\phi\,, \;\; \omega_z = \cos\theta$$



For some calculations we need the differential of the solid angle

$$dA = (rd\theta)(r\sin\theta\, d\phi)$$

$$d\boldsymbol{\omega} = \frac{dA}{r^2} = \sin\theta d\theta d\phi$$

Note:

$$\Omega = \int_{S^2} d\boldsymbol{\omega} = \int_0^{2\pi}\int_0^{\pi} \sin\theta d\theta d\phi = 4\pi$$

Some notations for the further description of photons and basic quantities

- $\boldsymbol{x}$: Position
- $\boldsymbol{\omega}$: Direction of motion
- λ: Wavelength
- $\frac{hc}{\lambda}$: Energy of single photon
- Φ: flux
- E: irradiance
- B: radiosity
- I: intensity
- L: radiance

5.1.1 Flux

Is the total amount of energy passing through a surface or space per unit time

$$\Phi(\mathrm{A}), \qquad \left[\mathrm{W} = \frac{\mathrm{J}}{\mathrm{s}}\right]$$

Examples are number of photons hitting a wall per second or number of photons leaving a light bulb per second.

5.1.2 Irradiance

Flux per unit area *arriving* at a surface, or in other words: area density of flux

$$E(\boldsymbol{x}) = \frac{d\Phi(\mathrm{A})}{dA(\boldsymbol{x})}, \qquad \left[\frac{\mathrm{W}}{\mathrm{m}^2}\right]$$

A simple visualization would be the number of photons hitting a small patch of a wall per second, divided by the size of the patch.

5.1.3 Radiosity

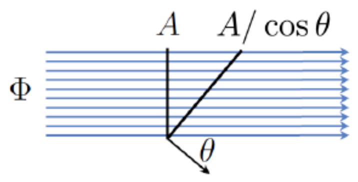
Pretty much the same as irradiance but this time it's about the flux per unit area *leaving* a surface

$$B(\boldsymbol{x}) = \frac{d\Phi(\mathrm{A})}{dA(\boldsymbol{x})}, \qquad \left[\frac{\mathrm{W}}{\mathrm{m}^2}\right]$$

5.1.4 Lambert's Cosine Law

The Irradiance of a surface which isn't perpendicular to a light source is of course lower than the definition above

$$E = \frac{\Phi}{\mathrm{A}/\cos\theta} = \frac{\Phi}{\mathrm{A}}\cos\theta$$



5.1.5 Radiant intensity

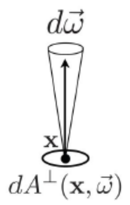
Power flux per unit solid angle = directional density of flux.

5.1.6 Radiance

It is the intensity per unit area. It's the most fundamental quantity for raytracing and remains constant along a ray

$$L(\boldsymbol{x},\boldsymbol{\omega}) = \frac{dI(\boldsymbol{\omega})}{dA(\boldsymbol{x})} = \frac{d^2\Phi(\mathrm{A})}{d\boldsymbol{\omega}dA^{\perp}(\boldsymbol{x},\boldsymbol{\omega})} = \frac{d^2\Phi(\mathrm{A})}{d\boldsymbol{\omega}dA(\boldsymbol{x})\cos\theta}, \qquad \left[\frac{\mathrm{W}}{\mathrm{m}^2\mathrm{sr}}\right]$$

Other radiometric quantities can be expressed in terms of radiance



$$L(\boldsymbol{x},\boldsymbol{\omega}) = \frac{dE(\boldsymbol{x})}{\cos\theta\, d\boldsymbol{\omega}} \Rightarrow \int_{H^2} L(\boldsymbol{x},\boldsymbol{\omega})\cos\theta\, d\boldsymbol{\omega} = E(\boldsymbol{x})$$

So the integration of radiance over the hemisphere leads to the irradiance (same for radiosity). Since the irradiance is dependent on the flux, we can also derive the flux

$$E(\boldsymbol{x}) = \frac{d\Phi(\mathrm{A})}{dA(\boldsymbol{x})} \Rightarrow \int_A \int_{H^2} L(\boldsymbol{x},\boldsymbol{\omega})\cos\theta\, d\boldsymbol{\omega} dA(\boldsymbol{x}) = \Phi(\mathrm{A})$$

Where A is an area.

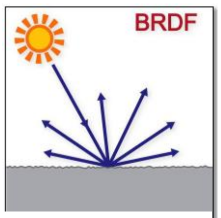
5.1.7 Summary

Flux $\Phi(A)$ $\left[\frac{J}{s} = W\right]$	Irradiance $E(\mathbf{x}) = \frac{d\Phi(A)}{dA(\mathbf{x})}$ $\left[\frac{W}{m^2}\right]$
Radiosity $B(\mathbf{x}) = \frac{d\Phi(A)}{dA(\mathbf{x})}$ $\left[\frac{W}{m^2}\right]$	Intensity $I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}}$ $\left[\frac{W}{sr}\right]$
Radiance $L(\mathbf{x},\vec{\omega}) = \frac{d^2\Phi(A)}{\cos\theta dA(\mathbf{x})d\vec{\omega}}$ $\left[\frac{W}{m^2 sr}\right]$	

5.2 Reflection Models

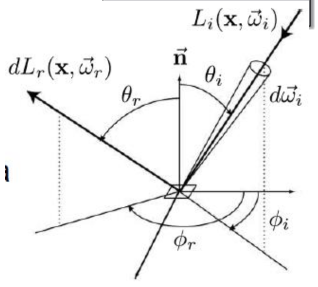
5.2.1 BRDF

BRDF stands for Bidirectional Reflectance Distribution Function.



$$f_r(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_r) = \frac{dL_r(\boldsymbol{x},\boldsymbol{\omega}_r)}{dE_i(\boldsymbol{x},\boldsymbol{\omega}_i)} = \frac{dL_r(\boldsymbol{x},\boldsymbol{\omega}_r)}{dL_i(\boldsymbol{x},\boldsymbol{\omega}_i)\cos\theta_{\mathrm{i}}\, d\boldsymbol{\omega}_i}$$

It describes the irradiance due to a cone of directions around $\boldsymbol{\omega}_i$. It provides a relation between incident radiance and differential reflected radiance.
It is a function which depends on the material and can be measured.
From the BRDF we can derive the reflection equation

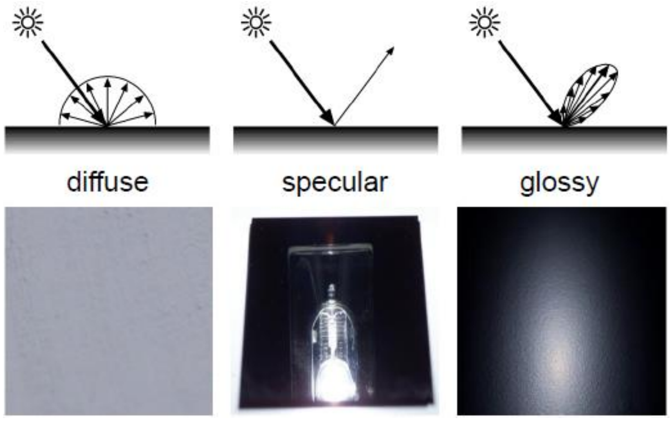


5.2.2 Reflection Equation

It describes a local illumination model. It gives us the reflected radiance due to incident illumination from <u>all</u> directions

$$L_r(\boldsymbol{x},\boldsymbol{\omega}_i) = \int_{\mathrm{H}^2} f_r(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_r) dL_i(\boldsymbol{x},\boldsymbol{\omega}_i)\cos\theta_{\mathrm{i}}\, d\boldsymbol{\omega}_i$$

5.2.3 Simple Reflection Types



5.2.3.1 Ambient Light

Light is scattered by environment and comes from all directions. The reflection is independent of camera and light position and also surface orientation. The reflected intensity I

$$I = I_a k_a$$

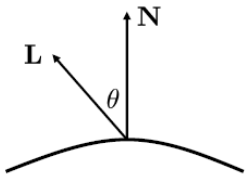
where I_a is the light source and k_a the material parameter for ambient light.

5.2.3.2 Diffuse Reflection

For diffuse reflection the BRDF is constant

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}_i) = \int_{\mathrm{H}^2} f_r(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_r) dL_i(\boldsymbol{x}, \boldsymbol{\omega}_i) \cos\theta_{\mathrm{i}}\, d\boldsymbol{\omega}_i = f_r \int_{\mathrm{H}^2} dL_i(\boldsymbol{x}, \boldsymbol{\omega}_i) \cos\theta_{\mathrm{i}}\, d\boldsymbol{\omega}_i = f_r E_i(\boldsymbol{x})$$

The exact computation would be too slow. That's why OpenGL uses simplified reflection models like Phong illumination.



We can model diffuse reflection with directed light I_p

$$I = I_p k_d \cos\theta = I_p k_d (\boldsymbol{N} \cdot \boldsymbol{L})$$

We see that the reflection in this case is dependent on orientation of surface and the light source position. It still is independent of the camera position.

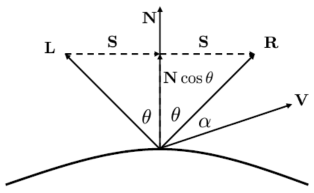
5.2.3.3 Attenuation

To account for attenuation for far away objects we can account for this with

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right)$$

If we combine ambient light and diffuse reflection and include attenuation we get a simple model often used in OpenGL

$$I = I_a k_a + f_{att} I_p k_d (\boldsymbol{N} \cdot \boldsymbol{L})$$



5.2.3.4 Specular Reflection

This type of reflection depends on the angle between the reflection and viewing ray. The angle between the reflectance and the position of the camera (viewing ray) is given by α

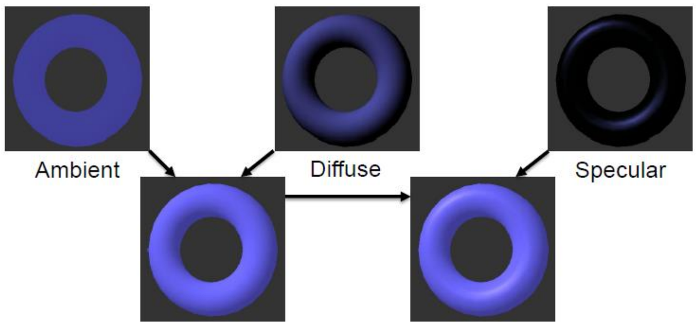
$$\boldsymbol{R} = \boldsymbol{N}\cos\theta + \boldsymbol{S} = 2\boldsymbol{N}\cos\theta - \boldsymbol{L} = 2\boldsymbol{N}(\boldsymbol{N} \cdot \boldsymbol{L}) - \boldsymbol{L}$$

$$\cos\alpha = \boldsymbol{R} \cdot \boldsymbol{V} = (2\boldsymbol{N}(\boldsymbol{N} \cdot \boldsymbol{L}) - \boldsymbol{L}) \cdot \boldsymbol{V}$$

This gives us specular reflection as

$$I = I_p k_s \cos\alpha = I_p k_s (\boldsymbol{R} \cdot \boldsymbol{V})$$

if we model it with a first order cosine.

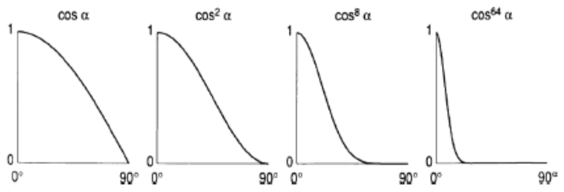


5.2.4 Phong Illumination Model

It approximates specular reflection by cosine powers ($= (\boldsymbol{R} \cdot \boldsymbol{V})^n$)

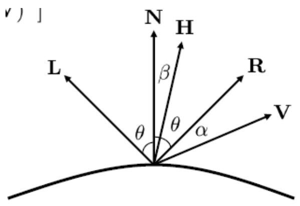
$$I_\lambda = \underbrace{I_{a_\lambda} k_a O_{d_\lambda}}_{Ambient} + \underbrace{f_{att} I_{p\lambda}}_{Attenuation} \left[\underbrace{k_d O_{d_\lambda} (\boldsymbol{N} \cdot \boldsymbol{L})}_{Diffuse} + \underbrace{k_s (\boldsymbol{R} \cdot \boldsymbol{V})^n}_{Specular} \right]$$

So by choosing n we define the behavior of the specular reflection. O_{d_λ} is the color property of the object, i.e. with that parameter we respect the color of the object.



5.2.4.1 *Halfway vector*

Since the computation of $\boldsymbol{R} \cdot \boldsymbol{V}$ takes too much time we can instead use the halfway vector



$$\boldsymbol{H} = \frac{\boldsymbol{L} + \boldsymbol{V}}{\|\boldsymbol{L} + \boldsymbol{V}\|}$$

So we can replace $\boldsymbol{R} \cdot \boldsymbol{V}$ with $\boldsymbol{N} \cdot \boldsymbol{H}$. Note that $\boldsymbol{N} \cdot \boldsymbol{H}$ represents the halfway angle

$$\cos\beta^n = (\boldsymbol{N} \cdot \boldsymbol{H})^n$$

5.2.4.2 *Multiple light sources*

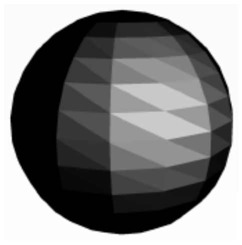
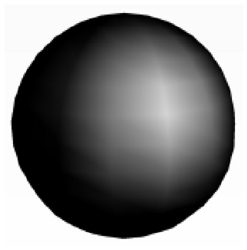
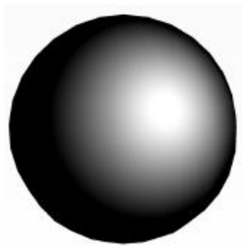
For multiple light sources we end up with

$$I_\lambda = I_{a_\lambda} k_a + \sum_{1 \le i \le m} f_{att_i} I_{p_{\lambda_i}} [k_d (\boldsymbol{N} \cdot \boldsymbol{L}_i) + k_s (\boldsymbol{R}_i \cdot \boldsymbol{V})^n]$$

5.3 Shading models

Shading refers to the process of altering the color of an object/surface/polygon in the 3D scene, based on its angle to lights and its distance from lights to create a photorealistic effect. Shading is performed during the rendering process by a program called a ***shader***.

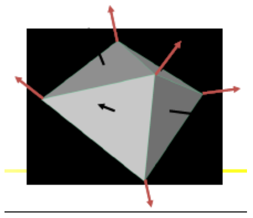
Flat shading	Gouraud shading	Phong shading
Color computed per primitive	Color computed per vertex	Color computed per fragment

No interpolation	Interpolation of vertex colors	Interpolation of vertex normals
Evaluation in object space (vertex shader)	Evaluation in object space (vertex shader)	Evaluation in screen space (fragment shader)
		

5.3.1 Gouraud Shading

Gouraud Shading has the following steps

1. Calculate face normals
2. Calculate vertex normals by averaging
3. Evaluate illumination model for each vertex
4. Interpolate vertex colors bilinearly on the current scan line

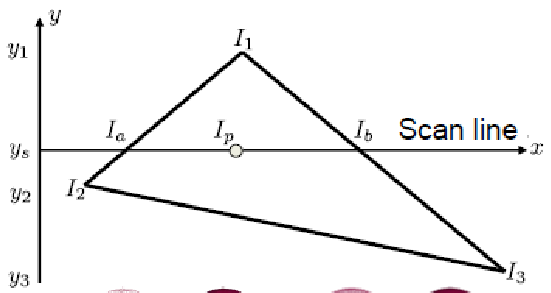


The vertex color interpolation is done bi-linearly on the current ***scan line***.

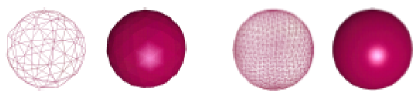
$$\boldsymbol{I}_a = \boldsymbol{I}_1 - (\boldsymbol{I}_1 - \boldsymbol{I}_2)\frac{(y_1 - y_s)}{(y_1 - y_2)}$$

$$\boldsymbol{I}_b = \boldsymbol{I}_1 - (\boldsymbol{I}_1 - \boldsymbol{I}_3)\frac{(y_1 - y_s)}{(y_1 - y_3)}$$

$$\boldsymbol{I}_p = \boldsymbol{I}_b - (\boldsymbol{I}_b - \boldsymbol{I}_a)\frac{(y_b - y_p)}{(y_b - y_a)}$$



For Gouraud Shading the quality depends on the size of the primitives



5.3.1.1 Brightness/Color Interpolation on the computer

On the computer this is not done with the formulas above. Since the differences from one point on the left to the point to the right are constant we can simply perform forward differentiation. That leaves us with additions such that multiplications and divisions aren't needed.

As an example, let's have a closer look on depth interpolation & z-buffering:

Say we have a planar polygon which fulfills the plane equation

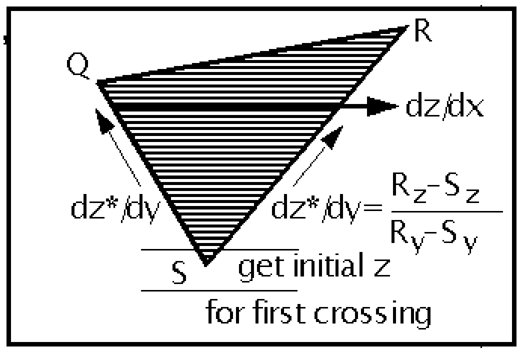
$$Ax + By + Cz + D = 0$$

This equation is linear in all directions. We can now solve for z and take the derivative

$$z = \left(-\frac{A}{C}\right)x + \left(-\frac{B}{C}\right)y - D \Rightarrow \frac{dz}{dx} = const. \& \frac{dz}{dy} = const.$$

The derivative in x and y direction is constant. These constants allows us to do a simple updating scheme by addition:

1. calculate the initial z-value for edges
2. add $\frac{dz^*}{dy}$ along the edges and
3. add $\frac{dz}{dx}$ along coherent spans

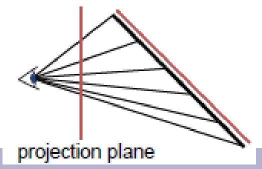


The same general approach is used also in brightness/color interpolation for Gouraud Shading. Then z is the color information.

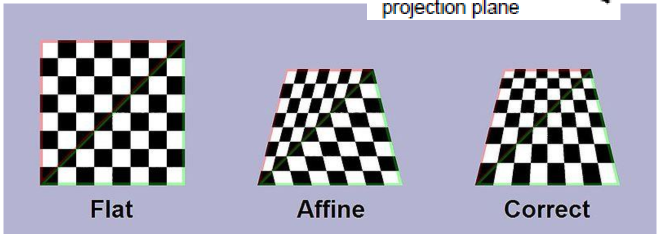
5.3.1.2 Problems with Gouraud Shading

5.3.1.2.1 Perspective distortion

Illumination/colors are evaluated at equal distance in **object** space! After the non-linear perspective projection, the distance are not equal anymore (Perspective distortion). So we need to perform a perspective correction.



Let's have a look on ***texture perspective correction.*** Affine texture mapping directly interpolates a texture coordinate u_α between two endpoints u_0 and u_1:



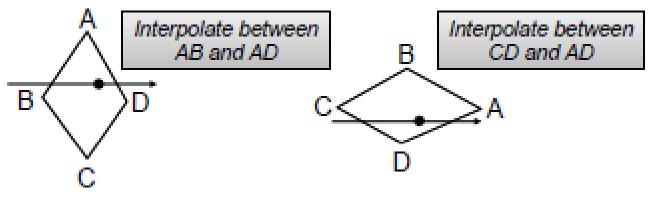
$$u_\alpha = (1-\alpha)u_0 + \alpha u_1$$

where $0 \leq \alpha \leq 1$. Perspective correct mapping interpolates after dividing by depth z, then uses its interpolated reciprocal to recover the correct coordinate:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

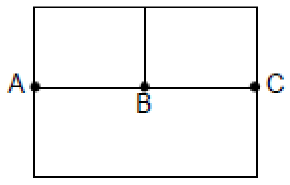
All modern 3D graphics hardware implements perspective correct texturing.

5.3.1.2.2 Orientation Dependence



It depends on the orientation of the face with which edges we interpolate. That's why we can get different results from the same face.

5.3.1.2.3 Shared Vertices



Here we have the problem that for a line scan of the big square the last one (from A to C) should be the same as the line scan for the two small squares of their first line (A to B and B to C).

5.3.2 Phong Shading

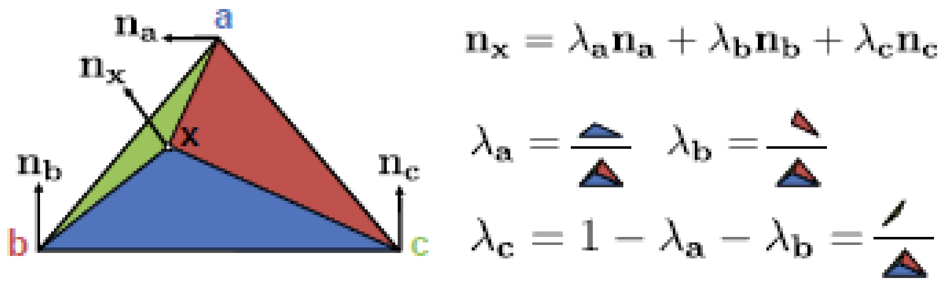
Phong shading is more accurate but also computationally more costly. The extra accuracy is justified for non-diffusive areas with relatively big polygons.

Note: Phong Shading and the Phong illumination model have nothing in common but their name. One is a shading model and the other is a reflection model. It's like comparing a Mitsubishi car with a Mitsubishi solar panel.

With Phong Shading we perform the following steps

1. Barycentric interpolation of normals on the triangles
2. Color of $\boldsymbol{x}$ is determined fragment-wise by the interpolated normal

5.3.2.1 Barycentric Interpolation



$\mathbf{n_a}$
$\mathbf{a}$
$\mathbf{n_x}$
$\mathbf{x}$
$\mathbf{n_b}$
$\mathbf{n_c}$
$\mathbf{b}$
$\mathbf{c}$
$\mathbf{n_x} = \lambda_\mathbf{a}\mathbf{n_a} + \lambda_\mathbf{b}\mathbf{n_b} + \lambda_\mathbf{c}\mathbf{n_c}$
$\lambda_\mathbf{a} =$
$\lambda_\mathbf{b} =$
$\lambda_\mathbf{c} = 1 - \lambda_\mathbf{a} - \lambda_\mathbf{b} =$

Properties:

- Lagrange: $\boldsymbol{x} = \boldsymbol{a} \Rightarrow \boldsymbol{n_x} = \boldsymbol{n_a}$
- Partition of unity: $\lambda_a + \lambda_b + \lambda_c = 1$
- Reproduction: $\lambda_a \boldsymbol{a} + \lambda_b \boldsymbol{b} + \lambda_c \boldsymbol{c} = \boldsymbol{x}$

5.3.2.1.1 Computing λ

With $\lambda_3 = 1 - \lambda_1 - \lambda_2$, the equations for x and y inside the triangle are

$$\begin{aligned} x &= \lambda_1 x_1 + \lambda_2 x_2 + (1 - \lambda_1 - \lambda_2) x_3 \\ y &= \lambda_1 y_1 + \lambda_2 y_2 + (1 - \lambda_1 - \lambda_2) y_3 \\ T \cdot \lambda &= r - r_3 \end{aligned}$$

Where $T = \begin{pmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{pmatrix}$, $\lambda = (\lambda_1, \lambda_2)$, $r = (x, y)$, $r_3 = (x_3, y_3)$

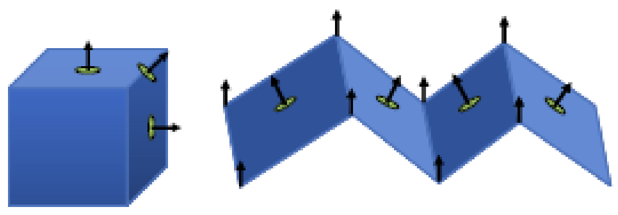
Therefore

$$\lambda = T^{-1} \cdot (r - r_3)$$

Where $\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$

5.3.2.2 Problems with Phong Shading

5.3.2.2.1 Normal not defined/representative



The normal is not defined for every place in the object.

5.4 Transparency

If objects are transparent we have to regard this. Therefore we introduce the Alpha channel by extend the 3D vector RGB to a 4D vector RGBA. 'A' is called the alpha channel.

5.4.1 Alpha Blending

Alpha blending is the process of combining a translucent foreground color with a background color, thereby producing a new blended color. The degree of the foreground color's translucency may range from completely transparent to completely opaque.

Alpha blending is a convex combination of two colors allowing for transparency effects in computer graphics.

The value of the resulting color is given by (**src**=source vector, **dst**=destination vector & **out**=output vector):

1. $out_A = src_A + dst_A(1 - src_A)$
2. $out_{RGB} = \frac{src_{RGB} src_A + dst_{RGB} dst_A (1 - src_A)}{out_A}$
3. $out_A = 0 \Rightarrow out_{RGB} = 0$

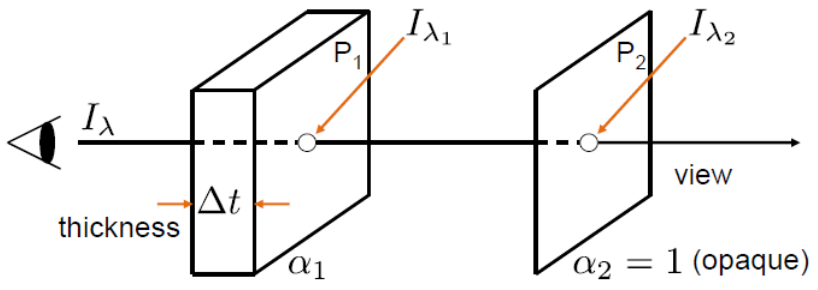
If the destination background is opaque, then $dst_A = 1$, and if you enter it to the upper equation:

1. $out_A = 1$
2. $out_{RGB} = src_{RGB} src_A + dst_{RGB}(1 - src_A)$

The alpha component may be used to blend to red, green and blue components equally, as in 32-bit RGBA, or, alternatively, there may be three alpha values specified corresponding to each of the primary colors for spectral colorfiltering.

Note that the RGB color may be premultiplied, hence saving the additional multiplication before RGB in the equation above. This can be a considerable saving in processing time given that images are often made up of millions of pixels.

If we think about intensity we can construct the following example:



We want to compute the intensity I_λ. We have the Emission of P_1which is $I'_{\lambda_1}$and the intensity filtered by P_1which is I'_{λ_2}. Both deliver the intensity

$$I_\lambda = I'_{\lambda_1} + I'_{\lambda_2} = I_{\lambda_1}\alpha_1\Delta t + I_{\lambda_2} e^{-\alpha_1 \Delta t}$$

When we linearize I'_{λ_2} we get

$$I'_{\lambda_2} = I_{\lambda_2}(1 - \alpha_1 \Delta t) \Rightarrow I_\lambda = I_{\lambda_1}\alpha_1 + I_{\lambda_2}(1 - \alpha_1)$$

Note: This formula of I_λ corresponds to the same as out_{RGB} if we take $dst_A = 1$.

5.4.1.1 Convex Combination

In convex geometry, a convex combination is a linear combination of points(which can be vectors, scalars, or more generally points in an affine space) where all coefficients are non-negative and sum to 1.

More formally, given a finite number of points $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n$ in a real vector space, a convex combination of these points is a point of the form

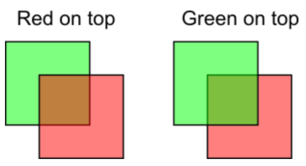
$$\boldsymbol{p} = \alpha_1\boldsymbol{x}_1 + \alpha_2\boldsymbol{x}_2 + \cdots + \alpha_n\boldsymbol{x}_n$$

where the real numbers α_i satisfy:

- $\alpha_i > 0$
- $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$

5.4.2 Ordering

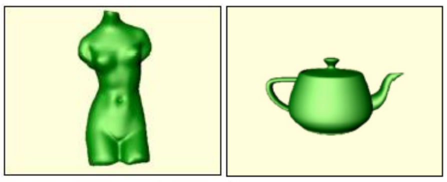
The main issue is that the ordering matters as we see in the picture to the right. To respect that we have to keep track of the position of each polygon in the scene and calculate the alpha blending in the right order!



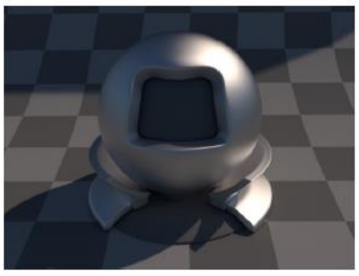
5.5 Further Lighting Models

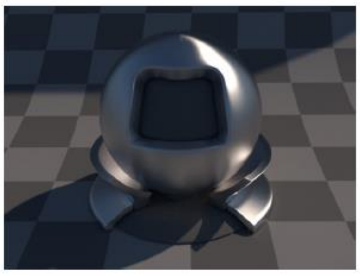
5.5.1 Cook-Torrance

The Cook-Torrance lighting model is a general model for rough surfaces and is targeted at metals and plastics – although it is still possible to represent many other materials.



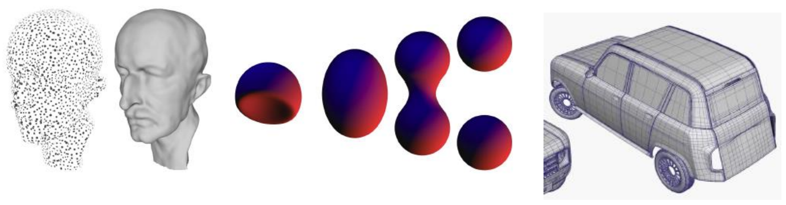
5.5.2 Ashikhmin


Isotropic microfacet distribution


Anisotropic microfacet distribution (au=0.3, av=0.1)

6 Geometry representations

- Parametric surfaces
- Subdivision surfaces
- Point set surfaces
- Implicit surfaces
- Polygonal Meshes

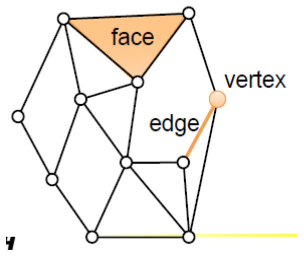


6.1 Polygonal Meshes

A **polygonal mesh** is a set of connected polygons

$$M = \langle V, E, F \rangle$$

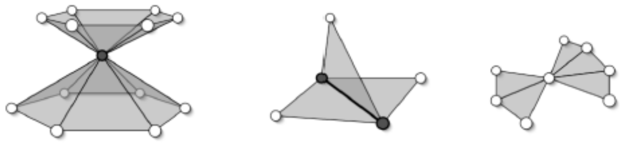
Were V are the vertices, E the edges and F the faces.



Properties:

- Every edge belongs to at least one polygon
- The intersection of two polygons in M is either empty, a vertex, or an edge.

If the surface is locally homeomorphic to a disk, the surface is a manifold. For a manifold mesh, where the following things are not allowed:



6.1.1 Mesh Data Structures

To store a mesh we should store the **geometry** (vertex locations), the **topology** (how vertices are connected (edges/faces)) and **attributes** (normal, color, etc).

7 Texture Mapping

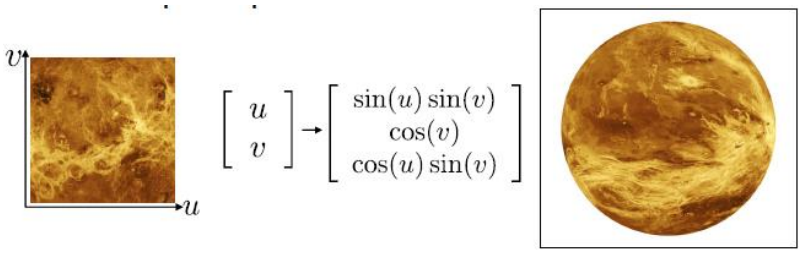
With **texture mapping** we can enhance details without increasing geometric complexity. To perform it we have to overcome some issues:

- Mapping between texture and geometry
- Anti-aliasing and filtering
- Level of detail

7.1 Mapping

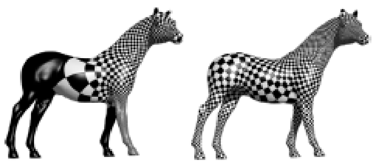
We need a one-to-one mapping between texture and geometry (the function goes from 2D to 3D):

$$f\colon \begin{pmatrix} u \\ v \end{pmatrix} \to \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}$$


v
u
$\begin{bmatrix} u \\ v \end{bmatrix} \to \begin{bmatrix} \sin(u)\sin(v) \\ \cos(v) \\ \cos(u)\sin(v) \end{bmatrix}$

Desirable properties for the function are

- Low distortion (preserve angle and lengths)
- Bijective mapping
- Efficient to compute



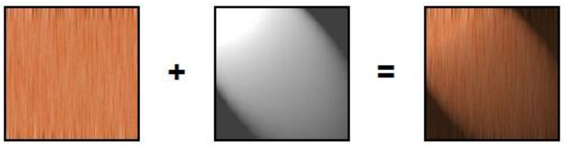
7.2 Anti-aliasing and filtering

As known from the computer vision part, the sampling frequency must be twice as big as the highest frequency in the image. Therefore, a low pass filter (e.g. Gaussian) can be used to reduce the highest frequency.

7.3 Special maps

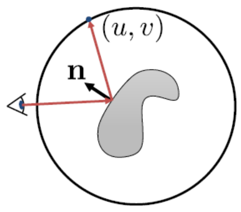
7.3.1 Light maps

Light maps simulate the effect of a local light source. They can be **precomputed** and dynamically adapted.



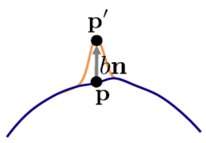
7.3.2 Environment maps

It's about rendering reflective objects efficiently. Texture coordinates are computed by intersecting the reflected ray with the surrounding sphere.



7.3.3 Bump maps

Key idea: Mimic bumps using special maps.



$$\mathbf{n} = \frac{\partial \mathbf{p}}{\partial \boxed{u}} \times \frac{\partial \mathbf{p}}{\partial \boxed{v}} = \mathbf{p}_u \times \mathbf{p}_v$$

Parameter domain

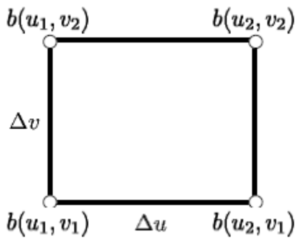
$$\mathbf{p}' = \mathbf{p} + \frac{b\mathbf{n}}{|\mathbf{n}|}$$

$$\mathbf{n}' = \frac{\partial \mathbf{p}'}{\partial u} \times \frac{\partial \mathbf{p}'}{\partial v}$$

$$\mathbf{p}' = \mathbf{p} + \frac{b\mathbf{n}}{|\mathbf{n}|} \qquad \mathbf{n}' = \frac{\partial \mathbf{p}'}{\partial u} \times \frac{\partial \mathbf{p}'}{\partial v}$$

$$\frac{\partial \mathbf{p}'}{\partial u} = \frac{\partial \mathbf{p}}{\partial u} + \frac{\partial}{\partial u}\frac{(b\mathbf{n})}{|\mathbf{n}|}$$

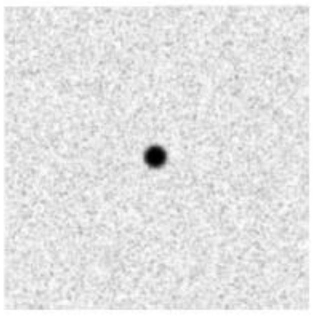
$$\mathbf{n}' = \mathbf{n} + b_u(\mathbf{n} \times \mathbf{p}_u) + b_v(\mathbf{n} \times \mathbf{p}_v)$$

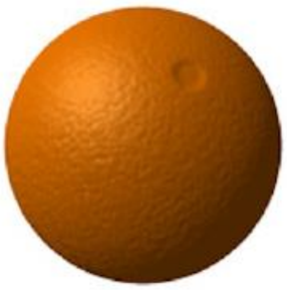


$$b_u = \frac{b(u_2, v_1) - b(u_1, v_1) + b(u_2, v_2) - b(u_1, v_2)}{2\Delta u}$$

$$b_v = \frac{b(u_1, v_2) - b(u_1, v_1) + b(u_2, v_2) - b(u_2, v_1)}{2\Delta v}$$

After the mathematical derivation, it turns out that strong gradients in the map induce bumps. E.g. a pattern with high black / white frequency induces a very bumpy surface as on the orange, while the large black spot in the middle corresponds to the “inlet” on the top of the orange.





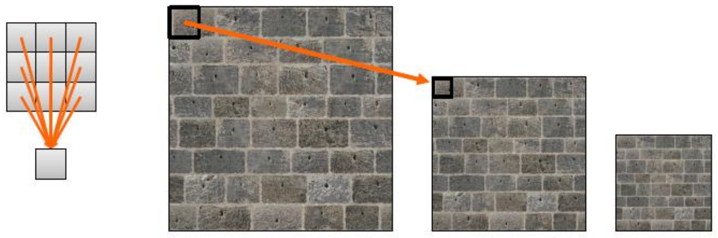
But Bump Mapping has also its limitations:

- No bumps on the silhouette
- No self-occlusions

- No self-shadowing

This can only be overcome by perturbing the geometry based on the textures. This is called **displacement mapping**.

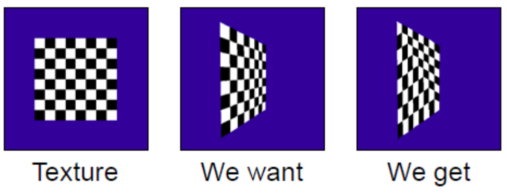
7.4 Level of detail (Mip-Mapping)

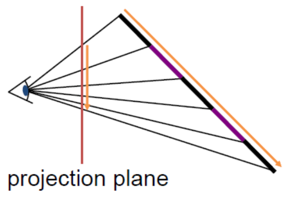


We store down-sampled versions of a texture. This is called **Mip-Mapping**. That way we can use low-res versions for far away objects and can interpolate for in-between depths as we go towards the texture. This **avoids aliasing** and **improves efficiency**. We compute lower resolution versions by weighted averages.

7.5 Perspective Interpolation

Perspective interpolation refers to the process of interpolating texture coordinates of *vertices* to texture coordinates of *pixels*. A linear interpolation in screen-space delivers the wrong result





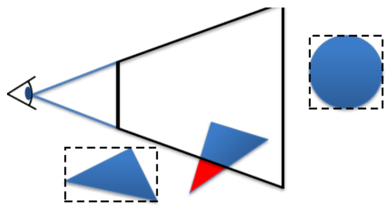
That's because linear variation in world coordinates yields non-linear variation in screen coordinates.

8 Clipping

Clipping is the removal of objects outside the frustum. Reasons for clipping are

- Save computation
- Memory consistency
- Stability: Division by 0

Culling rejects whole objects by bounding volumes



Clipping tears objects apart at the positions where they intersect the frustum.

8.1 Line clipping (Liang-Barsky algorithm)

Parametric form of a line segment

$$\boldsymbol{p}(t) = \boldsymbol{p_0} + (\boldsymbol{p_1} - \boldsymbol{p_0})t \qquad t \in [0,1]$$

Intersection point(s) with a plane (or rectangle)

$$\boldsymbol{n}_i^T(\boldsymbol{p}(t) - \boldsymbol{p}_{E_i}) = 0$$

$$t = \frac{\boldsymbol{n}_i^T(\boldsymbol{p_0} - \boldsymbol{p}_{E_i})}{-\boldsymbol{n}_i^T(\boldsymbol{p_1} - \boldsymbol{p_0})}$$

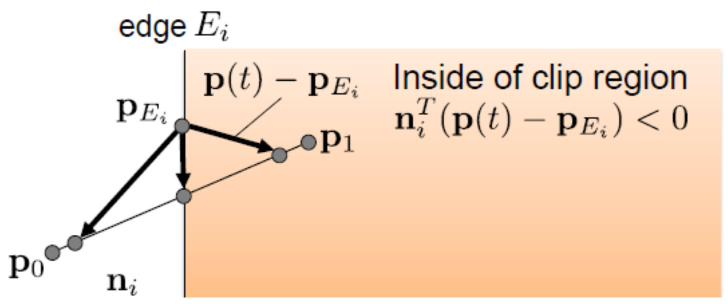


Abbildung 1 Line clipping

If the line goes through the rectangle, there is an entry point (PE) and a leaving point (PL). There is a simple criterion to determine a PE or a PL.

$$n^T d < 0 \Rightarrow PE$$

$$n^T d > 0 \Rightarrow PL$$

8.2 Polygon clipping

For each pair of vertices, perform line clipping.

Problem: If a line is outside the frustum, a turning vertex is needed.

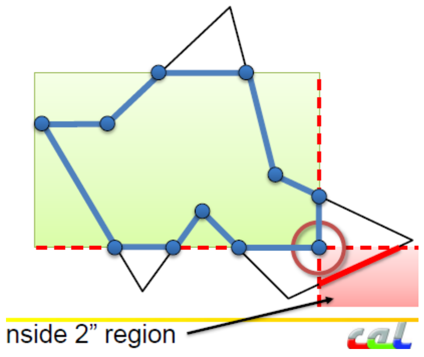
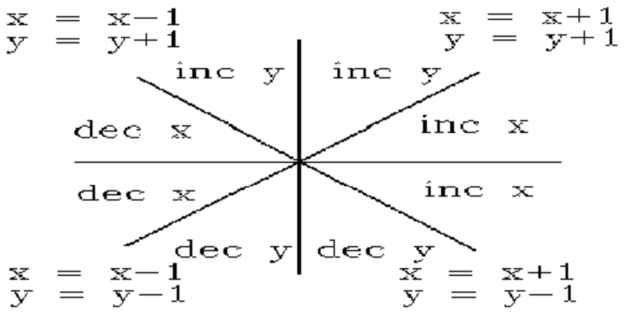


Abbildung 2 Polygon clipping

9 Scan Conversion (aka rasterization)

Given the specification for a straight line, find the collection of addressable pixels which most closely approximates this line. Simplest solutions have the problem of producing gaps if the slope of the line is to high and that they are inefficient, since they often need round functions and multiplications/divisions.

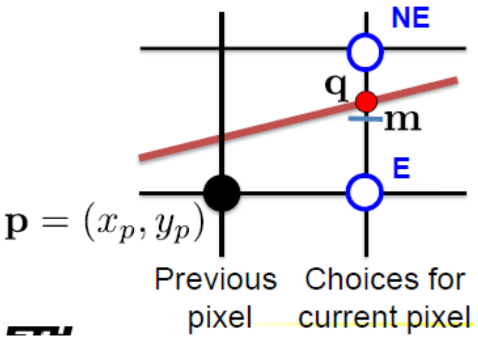
9.1 Bresenham Line



The Bresenham method depends on the the slope of the line which we want to draw. In general it works like:

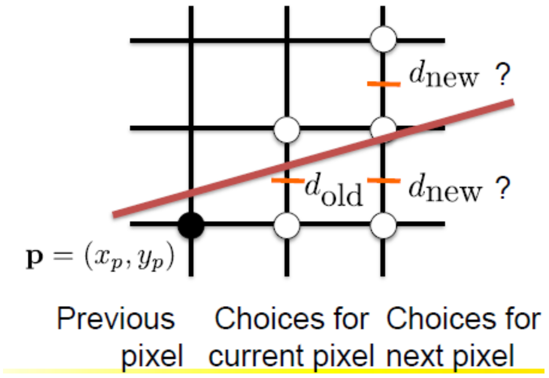
1. Determine in which octant we are working (slope) and define implicit equation $f(x, y) = ax + by + c = 0$
2. Display starting point (x_1, y_1)
3. Calculate with the midpoint $\boldsymbol{m}$ the scalar $d = f(\boldsymbol{m})$. $\boldsymbol{m}$ depends on the octant and describes the midpoint of the two pixel which are considered for the next step $(P_1 \& P_2)$.
4. If $d > 0$ take P_1else P_2
5. Repeat 3 & 4 till you reach (x_2, y_2).

9.1.1 Example: 1st octant



1. We are in octant 1
2. $\boldsymbol{p} = (x_1, y_1) = (x_p, y_p)$
3. $d = f(\boldsymbol{m}) = f\left(x_p + 1, y_p + \frac{1}{2}\right)$
4. If $d > 0$ take NE else E
5. Repeat 3 & 4

There is a faster solution by looking at $d_{new} - d_{old}$.



In the first step d must be computed as above.

It's crucial to see that there are only two possibilities for d_{new}, since the slope is constant.

Furthermore, if NE is chosen for the current pixel, the upper $d_{new} =: d_{new}^{NE}$ will be relevant for the next step, and the lower $d_{new} =: d_{new}^{E}$ if E is chosen, respectively.

The difference between d_{old} and d_{new}^{NE} and d_{new}^{E} is constant, namely:

- $d_{new}^{E} - d_{old} = a = \Delta y$
- $d_{new}^{NE} - d_{old} = a + b = \Delta y - \Delta x$

So, step 3 of the algorithm can be shortened

- If E was chosen at the current pixel: $d_{new} = d_{old} + a$
- If NE was chosen at the current pixel: $d_{new} = d_{old} + a + b$

Caution: The difference between $d_{new} - d_{old}$ depends on the octant, since d itself depends on it!!!

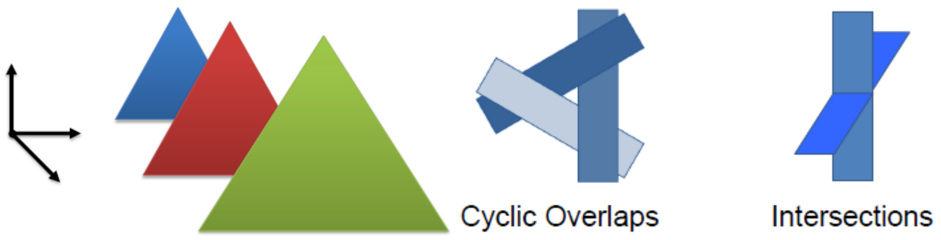
10 Visibility and Shadows

10.1 Visibility

The problem is, that some parts of some surfaces are occluded. We know 2 solutions to that problem

10.1.1 Painter's algorithm

We render objects/polygons from furthest to nearest. So we would first render the blue, then the red and at last the green triangle.
This concept doesn't work well. It doesn't account for **cyclic overlaps and intersections**!

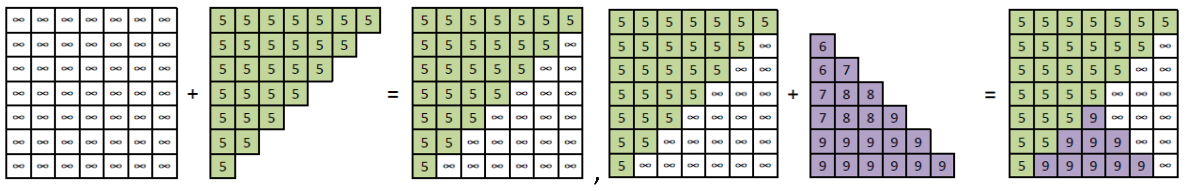


10.1.2 Z-Buffering

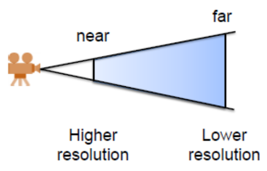
We store depth to the nearest object for each pixel. The algorithm goes like this

1. Initialize all z values to ∞
2. For each polygon:
   If z value of a pixel for this polygon is smaller than the stored z value, replace the stored z value

So this for two triangles (green and purple) this looks like



Also Z-Buffering has a problem: We have only a limited resolution. The resolution is non-linear. To reduce this, set the near plane far from the camera.

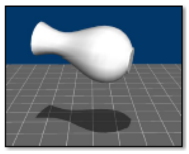


10.2 Shadows

Shadows in a scene give us some **depth cue**, like is the object lost in space, is it before a wall or above a surface? It also tells us something about the light source. For point sources we get hard shadows, where we get soft shadows for area light. And also the distance between the source and the object can be estimated by shadows.

10.2.1 Planar shadows

Draw **projection of the object on the ground**.



Limitations are:

- No self-shadows possible
- No shadows on other objects possible
- Curved surfaces not possible

10.2.2 Projective texture shadows

Here we separate obstacle (gives the shadow) and receiver (gets the shadow of obstacle). First the camera is put into the position of the light source and a black & white image of the obstacle is computed. In a 2nd pass, the image is used as a texture and projected onto the receiver.

Limitations are:

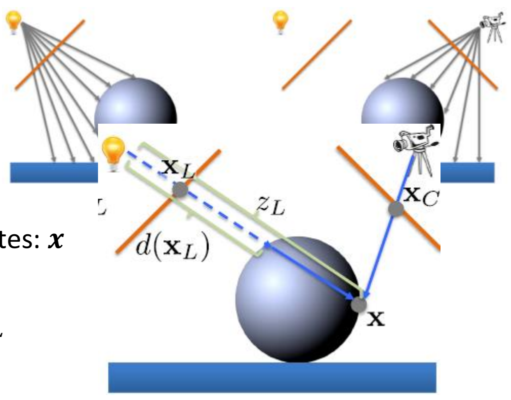
- No self-shadowing possible.
- Obstacle and receiver must be specified.

10.2.3 Shadow Maps

If you looked out from a source of light, all of the objects you can see would appear in light. Anything behind those objects, however, would be in shadow. This is the basic principle used to create a shadow map. The light's view is rendered, storing the depth of every surface it sees (the shadow map). Next, the regular scene is rendered comparing the depth of every point drawn (as if it were being seen by the light, rather than the eye) to this depth map.

As an algorithm this looks like:

1. Compute the depths from the light to the objects, and also the depths from the camera to the objects
2. For each pixel on the camera plane:
    a. Compute the point in world coordinates: $\boldsymbol{x}$
    b. Project point onto the light plane: $\boldsymbol{x}_L$
    c. Compare $d(\boldsymbol{x}_L)$ (shadow map) and z_L
    d. if $d(\boldsymbol{x}_L) < z_L$, $\boldsymbol{x}$ is in shadow

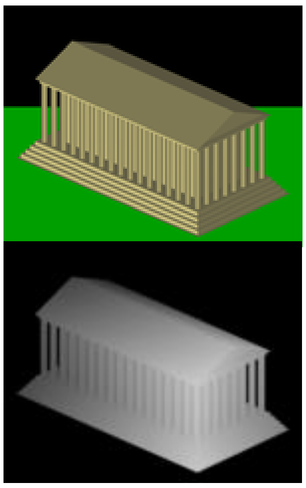


And more in detail:

10.2.3.1 Creating the shadow map

The first step renders the scene from the light's point of view. For a point light source, the projection should be chosen as

- A perspective projection for a spotlight
- An orthographic projection for directional light



From this rendering, the depth buffer is extracted and saved.

For every light source, there must be a depth map. Furthermore, the depth must be recomputed if the light source or the object changes position.

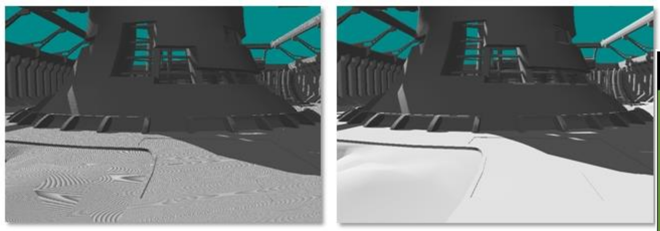
10.2.3.2 Shading the scene

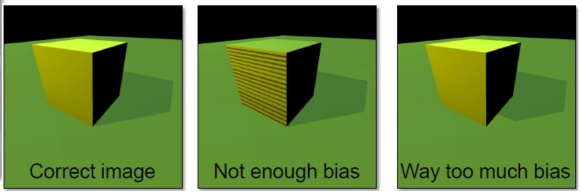
The second step is to draw the scene from the usual camera viewpoint, applying the shadow map. This process has three major components:

1. Find the coordinates of the object as seen from the light
2. Compare that coordinate against the depth map. (Involves coordinate transformation)
3. Draw the object either in shadow or in light.

10.2.3.3 Limitations of Shadow Maps

10.2.3.3.1 Unintentional self-shadowing (aka Shadow Acne)

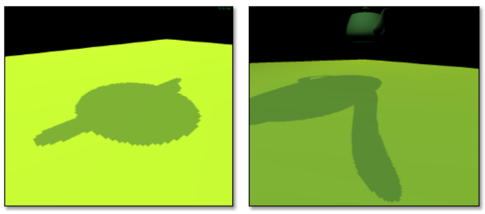




For a visible point with $d(\boldsymbol{x}_L) < z_L$ self-shadowing can occur. By introducing a bias we can avoid it

$$d(\boldsymbol{x}_L) + bias < z_L$$

Choosing the right bias can be very tricky.



10.2.3.3.2 Aliasing

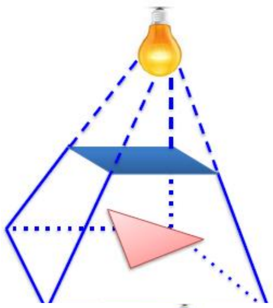
Can occur by undersampling of the shadow map. How can we overcome this problem? By Filtering the depth? No! We can instead **filter the result of the test**

$$d(\boldsymbol{x}_L) + bias < z_L$$

and take a weighted average of comparisons. Bigger filter produces fake soft shadows.

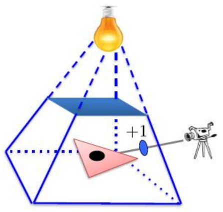
10.2.4 Shadow Volumes

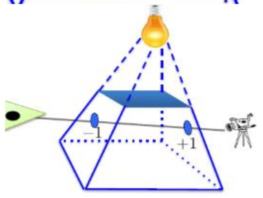
A shadow volume divides the virtual world in two: areas that are in shadow and areas that are not. So we explicitly represent the volume of space in shadow (new geometry). If a polygon is inside the volume, it is in the shadow.



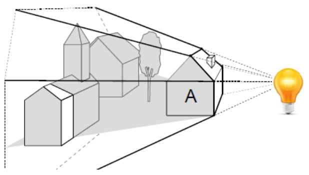
The following algorithm does this:

1. Shoot a ray from the eye (camera)
2. Increment/decrement a counter each time the boundary of the shadow volume is intersected
3. If counter:
   a. > 0: primitive is in shadow
   b. = 0: primitive is not in shadow





To optimize that we can use only the silhouette edges (where a back-facing & front facing polygon meet)



But also this method has its limitations:

- Introduces a lot of new geometry
- Expensive to rasterize long skinny triangles
- Objects must be watertight to use the silhouette optimization (watertight means that if we would fill the object with water, it would not leak)
- Rasterization of polygons sharing an edge must not overlap & not have gap

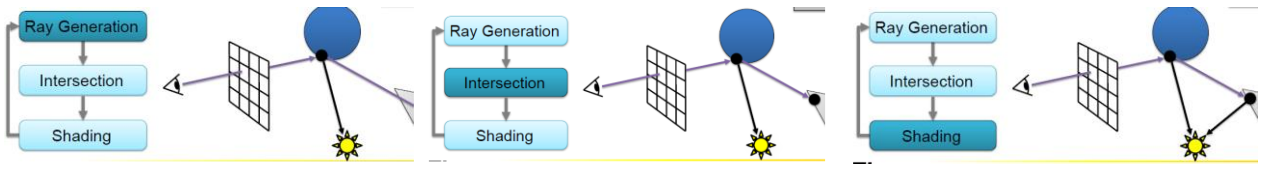
10.2.5 Comparison

Features / limitations	Planar shadows	Texture shadows	Shadow maps	Shadow volumes
Allows objects to cast shadows on themselves (Self-shadowing)	No	No	Yes	Yes
Permits shadows on arbitrary surfaces	No	Yes	Yes	Yes
Generates extra geometric primitives	No	Two-pass rendering: 1st pass: Put camera in position of light source and create shadow map 2nd pass: Cast shadow (apply map)		Yes
Limited resolution of intermediate representation			Yes	

can result in jaggy
shadow artifacts

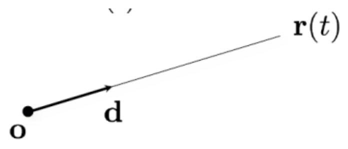
11 Ray Tracing

The Basic pipeline begins in the eye point and searches for all rays that go to the light source. This is called the backward ray tracing:



11.1 Ray-Surface Intersection

We start with a simple equation of a ray



$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$$

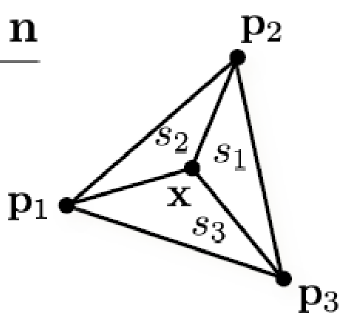
This ray can now intersect with a sphere or a triangle. Given an implicit form of the object we simply can put in $\boldsymbol{r}(t)$ for the point of interest. For example a sphere is given by

$$||\boldsymbol{x} - \boldsymbol{c}||^2 - r^2 = 0$$

So by inserting $\boldsymbol{r}(t)$ for $\boldsymbol{x}$ and solving for t, will lead to the intersection points. The same is possible for other figures like a triangle:

- Barycentric coordinates $\mathbf{x} = s_1\mathbf{p}_1 + s_2\mathbf{p}_2 + s_3\mathbf{p}_3$
- Intersect with triangle's plane

  $(\mathbf{o} + t\mathbf{d} - \mathbf{p}_1) \cdot \mathbf{n} = 0 \qquad t = -\dfrac{(\mathbf{o} - \mathbf{p}_1) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$
- where $\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$
- Compute s_i
- Test $s_1 + s_2 + s_3 = 1 \qquad 0 \leq s_i \leq 1$



Caution: Test functions are important! Ray tracing is often about aborting fast, since most rays don't hit a triangle. So the tests have to be simple. Best is, if they tell us something before we had to compute something.

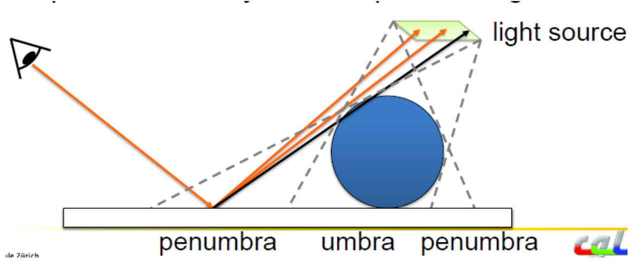
11.2 Shading

Physically exact shading too costly. Simplifying assumptions help:

- split surface reflectance into: diffuse, specular, ambient, transparency terms
- shawod rays to determine if a hit point is in shadow or not

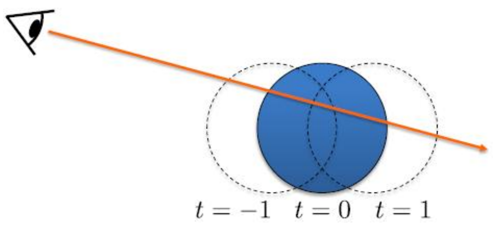
We can add various shadows according to different effects which all behave differently:

11.2.1 Area lights for soft shadows

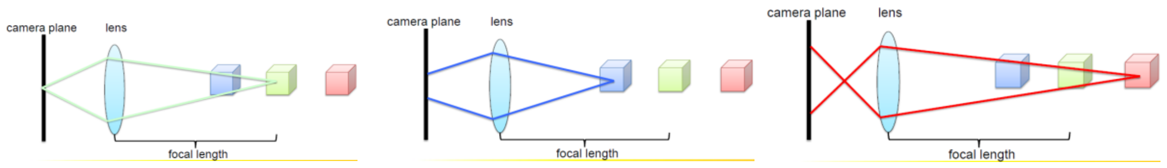


11.2.2 Motion Blur

Sample objects an intersect in time



11.2.3 Depth of field

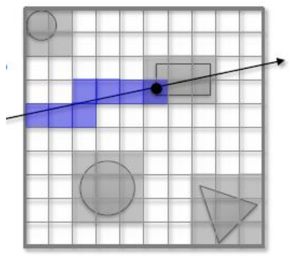


11.3 Acceleration

The problem with ray tracing an complex scenes is the computational cost. Therefore we have to reduce the intersections by

1. uniform grids
2. space partitioning

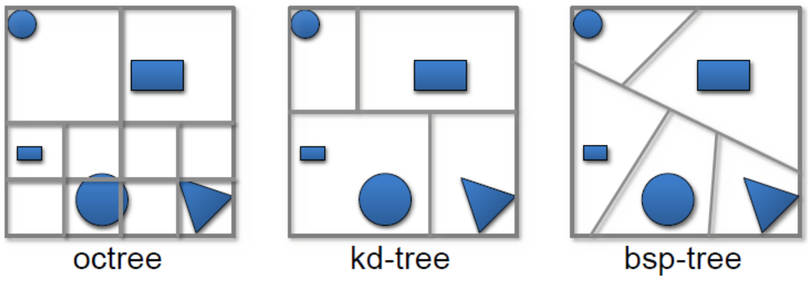
11.3.1 Uniform grids



We first compute the bounding box and set a grid resolution. Then we rasterize objects and store references to those. We incrementally rasterize the ray and stop when an intersection happens. Uniform grids are fast to build and easy to code, but they are non-adaptive to scene geometry.

11.3.2 Space partitioning

Space partitioning is particularly important in computer graphics, especially heavily used in ray tracing, where it is frequently used to organize the objects in a virtual scene. A typical scene may contain millions of polygons. Performing a ray/polygon intersection test with each would be a very computationally expensive task.



Storing objects in a space-partitioning data structure (kd-tree or BSP for example) makes it easy and fast to perform certain kinds of geometry queries – for example in determining whether a ray intersects an object, space partitioning can reduce the number of intersection test to just a few per primary ray, yielding a logarithmic time complexity with respect to the number of polygons.

Space partitioning is also often used in scanline algorithms to eliminate the polygons out of the camera's viewing frustum, limiting the number of polygons processed by the pipeline. There is also a usage in collision detection: determining whether two objects are close to each other can be much faster using space partitioning.