Computer Graphics

Michael Prinzinger

April 14, 2007

Contents

Preface

This script tries to merge information on computer graphics and interactive computer graphics from the lectures, the exercises, the books: Fundamentals Of Computer Graphics (from Peter Shirley) and Real Time Rendering (from Möller and Akenine-Haines) and last but not least intuition and idea that arose from talking to Professor Stamminger and talking to and the help from Christian Graef and Arian Baer.
In it I tried to present the information in the way we understood it with many hints and pictures that help understanding it. I think it is a valuable secondary resource, in case you did not understand a certain topic or want to know more about it.
As a merge from the above sources the contents exceed the lecture at many places, you simply have to decide for yourself how much you want to know and where to stop (I think knowing a little bit more than necessary does no harm, and instead you are more self confident and get a better overall understanding of the general problems and methods of computer graphics, because they repeat over and over again in different context. Once you had this realization, you can rest assured that you will pass the exam splendidly).

0.1 Definitions

Graphics

Computer Graphics Any use of computers to create or manipulate images.

Modeling mathematical specification of shape and appearance properties

Rendering creation of shaded images

Animation illusion of motion through sequence of images

Interactivity allowing the user to interact with the scene, immediately displaying the results (e.g. grab & drop) 5-6 fps

Real-Time render changes in the scene fast enough, that illusion of motion is created 20-60 fps

Units

Pixel PICture ELement. The smallest unit on the screen.

Texel TEXture ELement. The smallest unit on a texture.

Fragment Before the scene is rendered on the screen it often is rendered to a buffer having a greater (theoretically also lesser) resolution. Since some of the buffers elements are cast to one pixel, we introduce the term fragment for disambiguation.

Buffers

Z-Buffer A buffer identical to the framebuffer containing depth information for every pixel. Thus by having new objects come into view, we can compare the object's pixels' depth with the value stored in the Z-Buffer and draw or discard them accordingly.

W-Buffer An alternative to Z-Buffering. Instead of depth value the homogeneous perspective w coordinate is stored. The advantage of this method is having uniform depth values.

Stencil Buffer The stencil buffer is another duplicate of the framebuffer containing integer values (1 byte per pixel). It is mainly used to limit the area of rendering: Render only to pixels highlighted on the stencil buffer (e.g. for drawing shadows). It can be efficiently combined with the depth buffer, for example every time a depth test fails increase the pixels integer value on this position in the stencil buffer by 1.

Framebuffer A certain chunk of memory used for display on screen. Graphics intended to be written to the screen is written to the framebuffer.

Double Buffer Writing to the framebuffer while the monitor's photon cannon is displaying it's content, leads to flickering and artifacts. Therefore a technique called double buffering is commonly used. Graphics are first written to the double buffer (another framebuffer) and once the photon cannon reaches the bottom the two buffers are swapped.

Triple Buffer Double Buffering still can lead to artifacts: Image the pipeline just writing to the double buffer, when it is switched with the framebuffer. In this case no flickering will be seen, but depending on the amount of change, the scene's integrity will be broken for an instant. Therefore triple buffering was suggested. The rendering is started on the triple buffer, once a screen update has been made, double and triple buffer are swapped and rendering is completed on the double buffer, then in the next step it is displayed to the viewer. Another advantage is that during the rendering of the double buffer, the triple buffer can be cleared, which takes an considerable amount of time. Therefore using triple buffering more frames per second can be displayed than using double buffering. A disadvantage you should take into account is the latency of 3 frames. A user command/input will only have no effect on the next two frames.

Accumulation Buffer A buffer used to gather images of an object with set operations. It is mainly used to generate motion blurs, but also for soft shadows or depth antialiasing. Usually out of this set a single image with higher precision is created including the motion blur effect.

G Buffer A buffer used for deferred shading (see 8.4.4). In short in it we store every piece of information we need for an accurate lighting computation, so that we are able to perform the lighting stage anywhere in the pipeline.

Some words on the required memory. If we assume 1280x1024 pixels with true color, results in 8 bit per color channel = 3.75 MB. Using double buffering we need twice as much: 7.5 MB. The Z-Buffer with 24 bit per pixel requires 3.75 MB. Adding an accumulation buffer with 48 bit and a stencil buffer with 8 bit per pixel would result in 8,75 MB. Summing up to a total of 20 MB.

Computer

CPU Central Processing Unit

GPU Graphics Processing Unit

FLOPS Float Operations Per Second

0.2 Bibliography

Fundamentals of Computer Graphics by Peter Shirley (Second Edition)

A very good book that covers all the basics. If even with this script you have not completely grasped a certain topic about the very fundamentals, open the book and read the **whole** chapter about it. If however this topic of yours is also to be found in the book Real Time Rendering (see below), try the other one first. Although Shirley and Friends give real good explanations, the authors of the Real Time Book even surpass his explanations.

A word on the edition. The first edition was written by Shirley alone, the second one by Shirley and seven other authors, which added some minor changes to existing chapters and added completely new chapters on their own. So try to get hold of the second one.

Real Time Rendering by Tomas Akenine-Möller and Eric Haines (Second Edition)

In my opinion this book even surpasses Shirley's. The authors really give intuitive and splendid explanations going hand in hand with huge amounts of excellent pictures, figures and graphics illustrating what is being explained. Furthermore they cover the topics really good and can enrich your knowledge about the topics covered in the lecture. I've almost read through all the chapters and did never regret even one. If I didn't understand a topic with the slides and the lecture, I usually did understand it after reading through the corresponding chapter, if existing.

Other Resources

Apart from the slides you have from the lecture, you should from time to time try to find different explanations of illustratory applets with google.
The page of the lecture of Professor Stamminger (Interactive Computer Graphics `http://www9.informatik.uni-erlangen.de:81/Teaching/SS2006/InCG/`

Material) offers a great fundus of additional free internet resources. Also the page of Möller's and Haines' Book (Real Time Rendering http://www.realtimerendering.com/) and of Shirley's book (Fundamentals Of Computer Graphics http://www.cs.utah.edu/~shirley/books/fcg2/) offer nice slides based on their books and various other helpful links. Last but not least Wikipedia frequently helps with different approaches to topics.

0.3 Copyright

This document is to be seen as OpenSource and I would be happy if anyone decides to enrich this script by adding additional concepts, better explanations or additional examples and illustrations and of course correcting all the mistakes, that I made. The sources (.lyx or .tex) can be acquired by writing a short e-mail to me: michaelprinzinger@gmx.de. However as in the GNU-Licence, I hereby fobid anyone to postulate money for this document or use parts from it for commercial works. It is meant to be a free help for students all over the world and it should remain free.

1 Application

Movies

computer generated foregrounds, Animations, special effects

Games

the drive behind graphics development

Computer Aided Design (CAD)

architecture, products, cars, planes, mechanical parts

Education & Training

simulation of realistic environments, flight simulator

Visualization

medical applications: model ling of (parts of) the human body

Virtual Reality (VR)

Immersion, response to head motion, stereo pictures, additional components (Sound, Force Feedback)

2 Math

2.1 Vectors

2.1.1 Properties

orthogonal vectors building a right angle: $\vec{u} \cdot \vec{v} = 0$

orthonormal orthogonal vectors having length 1: $\vec{u} \cdot \vec{v} = 0$ and $\|\vec{u}\| = \|\vec{v}\| = 1$

construction use Gram-Schmidt Orthognoalization / Orthonormalization

2.1.2 Operations

length $\|\vec{a}\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$

scalar product $\vec{a}^T \cdot \vec{b} = \|\vec{a}\| \left\|\vec{b}\right\| \cos\phi$

> **usage**: compute the angle between two vectors.

$$\vec{a}^T \cdot \vec{b} = \begin{array}{cc} & \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \\ \begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} & a_1b_1 + a_2b_2 + a_3b_3 \end{array}$$

tensor product $\vec{a} \cdot \vec{b}^T = M^{n \times n}$

> **usage:** combine two vectors to a matrix. e.g. for combining two 1D functions to one 2D one (see Bézier Curves & Splines 10.2).

$$\vec{a} \cdot \vec{b}^T = \begin{array}{cc} & \begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \\ \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} & \begin{pmatrix} a_1b_1 & a_2b_1 & a_3b_1 \\ a_1b_2 & a_2b_2 & a_3b_2 \\ a_1b_3 & a_2b_3 & a_3b_3 \end{pmatrix} \end{array}$$

cross product $\left\|\vec{a} \times \vec{b}\right\| = \|\vec{a}\| \left\|\vec{b}\right\| \sin\phi$

> **usage**: compute a third vector perpendicular to $\vec{a}$ and $\vec{b}$ (3D)

orthogonal projection $\vec{a} \to \vec{b} = \frac{\vec{a} \cdot \vec{b}}{\|\vec{b}\|} = \vec{a} \cdot \cos\phi$

2.2 Matrices

$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$

Quadratic $A^{n \times m}$ where $n = m$

Identity $A^{n\times n}$ having 1 on the diagonal and 0 everywhere else

$$I^{2\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Transpose $A^T : A^{n\times m} \to A^{m\times n}$ switch rows with columns

Adjoint $\bar{A}$. This matrix has the entries

$$a_{ij} := \det(A_{ij}) \cdot \begin{cases} -1 & \text{if } (i+j) \text{ odd} \\ 1 & \text{if } (i+j) \text{ even} \end{cases}$$

where A_{ij} means the matrix resulting from A when removing the i^{th} row and the j^{th} column. The resulting matrix is called the **cofactor matrix**. Take its transpose to get the adjoint matrix $\bar{A}$.
The adjoint has the following nice property:

$$A \cdot \bar{A} = \det(A) \cdot I_n$$

Inverse $A \cdot A^{-1} = I$

$$A^{-1} = \frac{1}{\det(A)}\bar{A}$$

2.2.1 Determinants

Vectors

The determinant of two vectors, $\vec{a}, \vec{b}$is a parallelogram.

$$\left|\vec{a}\vec{b}\right| = x_a y_b - y_a x_b$$

Having three vectors it is a cube with parallel parallelograms as sides.

$$\left|\vec{a}\vec{b}\vec{c}\right| = x_a y_b z_c - x_a y_c z_b - x_b y_a z_c + x_c y_a z_b + x_b y_c z_a + x_c y_b z_a$$

Matrices

There are several methods to compute the determinant of a given matrix. Look them up in a linear algebra script.

2.2.2 Eigenvalues & Eigenvectors

Condition matrix A has to be quadratic.

Eigenvalues

$$A\vec{x} = \lambda\vec{x}$$

where λ is called eigenvalue.
$A\vec{x} = I\lambda\vec{x}$
$(A - \lambda I)\,\vec{x} = 0$
$\begin{pmatrix} a_{11}-\lambda & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22}-\lambda & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33}-\lambda & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix}$
solve this to get $\lambda_1 \ldots \lambda_n$.

Eigenvectors

Plug in $\lambda_1 \ldots \lambda_n$ into A results in the eigenvectors.

Singular Value Decomposition (SVD)

For non quadratic matrices.

Benefit singular values, eigenvalues, orthonormal basis, pseudo inverse, condition

Singular Values σ
for symmetric matrices they are equal to the eigenvalues
in case A is not quadratic we have $A = MM^T$ and thus the singular values would be $\sigma_{MM^T} = \sqrt{\lambda_A}$

Decompose $A \in \mathbb{R}^{n\times m}$ into $A = U\Sigma V^T$, where

$$\Sigma = \begin{pmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \sigma_3 \end{pmatrix}$$

with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$
$U \in \mathbb{R}^{m\times n}$ and $U \cdot U^T = I$ (orthonormal column vectors)
$V \in \mathbb{R}^{n\times n}$ and $V \cdot V^T = V^T \cdot V = I$ (orthonormal column and row vectors)

Condition $\kappa := \frac{\sigma_1}{\sigma_n}$

If κ is close to one, the problem is well conditioned, if it is large the problem is unstable

Pseudoinverse $A^t = V\Sigma' U^T$ where Σ' results from Σ when replacing all singular values by their reciprocal values $\left(\sigma_i \mid \frac{1}{\sigma_i} \to \Sigma'\right)$

2.3 Coordinate Systems

world without explicitly storing the coordinates of the origin, we usually have one world coordinate system, where local object coordinate system will be place in.

local a local coordinate system refers to an object. If it is placed in a world coordinate system, a mapping must be made to access points in the object relative to the world coordinate system.

eye a perspective space with viewing coordinates.

screen the screen space is the coordinate system of the computer screen

mapping

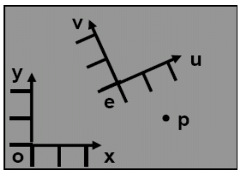


Figure 1: mapping from one coordinate system into another

The world origin is o, the local one e. The world basis vectors are denoted x, y, z, the local ones u, v, w. Then

$$e = (x_e, y_e, z_e) = o + x_e \cdot x + y_e \cdot y + z_e \cdot z$$

Additionally the local coordinate system may be rotated. Model ling both rotation and translation by a matrix we can easily move forwards and backwards through different coordinate systems (see 5).

Example

$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}$$

Mapping the eye space to the screen space requires a mapping to the view frustum (a unit cube) with normalized coordinates first.

2.3.1 Cartesian Coordinates

We use a orthonormal basis vector system, with the three basis vectors x-axis $(1, 0, 0)$, y-axis $(0, 1, 0)$ and z-axis $(0, 0, 1)$.

$$p = (x_0, y_0, z_0) = x \cdot x_0 + y \cdot y_0 + z \cdot z_0$$

2.3.2 Polar Coordinates

We use two parameters to describe any point in the coordinate system: distance from the origin r and angle between coordinate axes and the vector ϕ.

$$p = (r_0, \phi_0)$$

2.3.3 Barycentric Coordinates

Mainly used for interpolating color values on triangles. We use non-orthogonal basis vectors. a is the origin and $(b - a)$ and $(c - a)$ the basis vectors.

$$p = (\beta, \gamma) = a + \beta\,(b - a) + \gamma\,(c - a)$$

$$p = (1 - \beta - \gamma)\,a + \beta b + \gamma c$$

$$\alpha \equiv 1 - \beta - \gamma$$

$$p = \alpha a + \beta b + \gamma c$$

resulting in the constrain that $\alpha + \beta + \gamma = 1$.

2.3.4 Homogeneous Coordinates

Used for matrix transformations. They are based on projective geometry and irreplaceable useful in graphic transformations. The idea is to artificially increase the dimension.

So being in 2D, we would result in having three coordinates

$$(x, y) \to (x, y, 1)$$

where 1 is the homogeneous coordinate. See $(x, y, 1)$ as the line $\alpha \cdot x, \alpha \cdot y, \alpha \mid \alpha \in \mathbb{R}^3$

Direction And Location

The homogeneous coordinate w acts as a kind of pointer to a location (translation from the origin). But often we want a vector to store a direction rather than a location. In the latter case we simply set $w = 0$ ind the first case $w = 1$.

Dehomogenization

If our vector is $\vec{p} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$, the dehomogenized vector is $\vec{p} = \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{bmatrix}$

Properties

Homogeneous coordinates have some very useful properties justifying their usage

- any two lines intersect in one point
- points at infinity
- affine transformation become linear
- preserves cross-ratio

2.3.5 Mappings

Cartesian -> Barycentric

$$\begin{bmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_p - x_a \\ y_p - y_a \end{bmatrix}$$

Imagining a lines AC and AB passing through a barycentric triangle, we can get β, γ and α by:

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$

$$\gamma = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}$$

$$\alpha = 1 - \beta - \gamma$$

where $f_{ab}(x, y)$ can implicitly written as

$$f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

A third possibility is using areas A_a, A_b, A_c resulting from drawing lines from the center to the three points ($A = A_a + A_b + A_c$):

$$\alpha = \frac{A_a}{A} = \frac{\vec{n} \cdot \vec{n}_a}{\|\vec{n}\|^2}$$

$$\beta = \frac{A_b}{A} = \frac{\vec{n} \cdot \vec{n}_b}{\|\vec{n}\|^2}$$

$$\gamma = \frac{A_c}{A} = \frac{\vec{n} \cdot \vec{n}_c}{\|\vec{n}\|^2}$$

in the 3D case, we can use normal vectors instead of the area.

2.4 Implicit Functions

Implicit Lines

The common line definition is:

$$y = m \cdot x + t$$

the implicit form is easily obtained by:

$$y - m \cdot x - b = 0$$

where m is the slope (Steigung) and b the y-value, where the line crosses the y-axis.

Since this form still lacks some lines like $x = 0$ where m would be infinite large, we advance to the more general

$$ax + by + c = 0$$

Any point (x_0, y_0) on this line must satisfy the equation: $ax_0 + by_0 + c = 0$

Distance Point to Line:

The distance from point (x_1, y_1) to the line $ax + by + c = 0$ is

$$\text{distance} = \frac{f\left(x_1, y_1\right)}{\sqrt{a^2 + b^2}}$$

If (a, b) is a unit vector, the distance is directly given by $f\left(x, y\right)$.

Implicit Circles

A circle with center (c_x, c_y) and radius r has the implicit form

$$\left(x - c_x\right)^2 + \left(y - c_y\right)^2 - r^2 = 0$$

Implicit Ellipsis

A ellipse with center (c_x, c_y) and minor and major semi-axes a and b

$$\frac{\left(c - c_x\right)^2}{a^2} + \frac{\left(c - y_x\right)^2}{b^2} - 1 = 0$$

Given: function $f\left(x, y, z\right)$, point $\vec{p} = (x, y, z)$

Surface Normal

The surface normal is given by the gradient

$$\vec{n} = \nabla f\left(x, y, z\right) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$$

Implicit Planes

$$P : (\vec{p} - \vec{a}) \cdot \vec{n} = 0$$

A plane P given by three points $\vec{a}, \vec{b}, \vec{c}$

$$\vec{n} = \left(\vec{b} - \vec{a}\right) \times (\vec{c} - \vec{a})$$

$$P : (\vec{p} - \vec{a}) \cdot \left(\left(\vec{b} - \vec{a}\right) \times (\vec{c} - \vec{a})\right) = 0$$

Implicit Spheres

$$f\left(x, y, z\right) = \left(x - c_x\right)^2 + \left(y - c_y\right)^2 + \left(z - c_z\right)^2 - r^2 = 0$$

2.5 Parametric Functions

Parametric Functions use parameters to describe the function.

Parametric Lines

A parametric line passing through points $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$ can be written as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t\left(x_1 - x_0\right) \\ y_0 + t\left(y_1 - y_0\right) \end{bmatrix}$$

$$p\left(t\right) = p_0 + t\left(p_1 - p_0\right)$$

Parametric Circles

A circle with center (c_x, c_y) and radius r can be written as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_x + r \cdot \cos\phi \\ c_y + r \cdot \sin\phi \end{bmatrix}$$

Parametric Ellipsis

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_x + a \cdot \cos\phi \\ c_y + b \cdot \sin\phi \end{bmatrix}$$

3D parametric surfaces have the form

$$x = f\left(u, v\right)$$

$$y = g\left(u, v\right)$$

$$z = h\left(u, v\right)$$

Parametric Spheres

Consider a sphere, that's center is at the origin having radius r

$$x = r \cdot \cos\phi \sin\theta$$

$$y = r \cdot \sin\phi \cos\theta$$

$$z = r \cdot \cos\theta$$

where ϕdenotes the longitude (angle between x-axis the y-axis and the vector on the xy-plane) and θ denotes the latitude (angle between the z-axis the xy-plane and the vector). See **FoCG** p.41 .

$$\theta = \text{acos}\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right)$$

$$\phi = \text{acos}\,(y, x)$$

By that we get

$$\vec{x} = r \cdot \begin{pmatrix} r \cdot \cos\phi \sin\theta \\ r \cdot \sin\phi \cos\theta \\ r \cdot \cos\theta \end{pmatrix} + \vec{c}$$

2.6 Curves

linear $p(t) = c_1 t + c_0$

quadratic $p(t) = c_2 t^2 + c_1 t + c_0$

cubic $p(t) = c_3 t^3 + c_2 t^2 + c_1 t + c_0$

2.7 Polynomials

Bernstein

$$B_i^n(x) = \left(\begin{array}{c} n \\ i \end{array}\right) (1 - x)^{n-i} x^i$$

Lagrange

$$L_i(x) = \prod_{\substack{j = 0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j}$$

Legendre

$$P_n(x) = \frac{1}{2\pi i} \int \sqrt{(1 - 2tx + t^2)} t^{-n-1} dt$$

Splines

$$b_0(t) = \frac{1}{6}t^3$$

$$b_1(t) = \frac{1}{6} \cdot \left(-3t^3 + 3t^2 + 3t + 1\right)$$

$$b_2(t) = \frac{1}{6} \cdot \left(3t^3 - 6t^2 + 4\right)$$

$$b_3(t) = \frac{1}{6} \cdot \left(1 - t^3\right)$$

$$b_n(t) = (n+1) \sum_{i=0}^{n+1} \omega_{i,n} (t - t_i)^n$$

where

$$\omega_{i,n} = \prod_{\substack{j = 0 \\ j \neq i}}^{n+1} \frac{1}{t_j - t_i}$$

2.8 Linear Interpolation

Linear Interpolation is the process of passing through a geometric surface by a parameter t.

E.g. as we have already seen:

$$p = (1 - t)\,a + t \cdot b$$

is a linear interpolation. It is linear, because t and $t - 1$ are linear polynomials of t.

Interpolating through a set of points on the x-axis having assigned a height y_i to each point x_i, interpolating over those height values, we get

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i)$$

Consecutively you can think of the x-values as 3D vectors and the y-values as color values.

2.9 Triangles

Triangles usually are the fundamental primitives for graphics programs. Most commonly their vertices store a color value, which is then interpolated across the triangle. To make this interpolation straight forward, we will use barycentric coordinates.

Given: triangle $\triangle ABC$

Area(2D) area $= \frac{1}{2}\left|x_a y_b + x_b y_c + x_c y_a - x_a y_c - x_b y_a - x_c y_b\right|$

Internal Point a point p is inside the triangle if and only if $0 < \alpha < 1$, $0 < \beta < 1$, $0 < \gamma < 1$.

Edge one point is zero, the other two between zero and one

Vertex two points are zero, the other one is one

Normal Vector $\vec{n} = \left(\vec{b} - \vec{a}\right) \times (\vec{c} - \vec{a})$ (a vector perpendicular to the triangle edges)

Area(3D) area $= \frac{1}{2}\left\|\left(\vec{b} - \vec{a}\right) \times (\vec{c} - \vec{a})\right\|$

2.10 Quaternions

The quaternions $\mathbb{H}$ can be seen as an extension to the body of complex numbers $\mathbb{C}$.

Quaternions	Complex Numbers
$\mathbb{H} = \mathbb{R} \times \mathbb{R}^3$	$\mathbb{C} = \mathbb{R} \times \mathbb{R}$
$q = (q_0, \vec{q}) = a + ib + jc + kd$	$z = x + iy$
$n_+ = (0, \mathcal{O})$	$n_+ = (0, 0)$
$n. = (1, \mathcal{O})$	$n. = (1, 0)$
$i. = \frac{1}{\|q\|}(q_0, -\vec{q})$	$i. = \frac{1}{\|z\|}(x, -y)$
$q = \|q\|(\cos(t), \sin(t) \cdot \vec{n}_0)$	$z = \|z\|(\cos(t), \sin(t))$

2.11 Miscellaneous

Angle

The angle θ of a circular arc of length l and radius r is equal to

$$\theta = \frac{l}{r}\,[rad]$$

Example: Circle

$l = 2\pi r$
$\theta = \frac{l}{r} = \frac{2\pi r}{r} = 2\pi\,[rad]$

Solid Angle

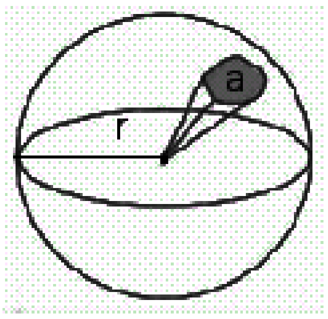


Figure 2: solid angle

A solid angle is the equivalent to the angle in 3D. The angle Ω with a spherical area a is equal to

$$\Omega = \frac{a}{r^2}\,[\textbf{stearradians}]$$

Example: Sphere

$a = 4\pi r^2$
$\Omega = \frac{a}{r^2} = \frac{4\pi r^2}{r^2} = 4\pi\,[sr]$

Solid Angel Differential

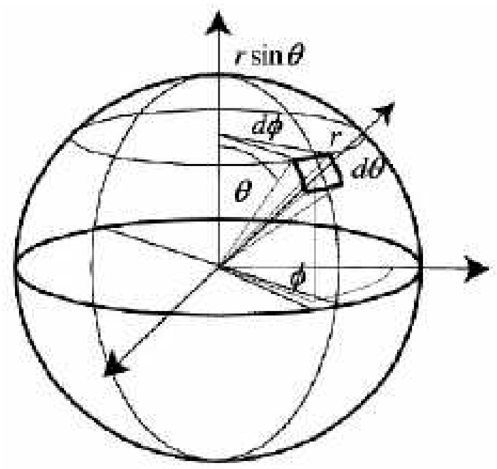


Figure 3: Solid Angle Differential

For light purposed we need to differentiate the solid angle:

arc length $[\theta, \theta + d\theta] : rd\theta$

arc length $[\phi, \phi + d\phi] : r \sin\theta d\phi$

area differential $dA = (rd\theta)\,(r \sin\theta d\phi) = r^2 \sin\theta d\theta d\phi$

angle differential $d\omega = \frac{dA}{r^2} = \sin\theta d\theta d\phi\,[sr]$

We can use this result to integrate over the entire sphere and get the solid angle S

$$S = \int_0^{\pi}\int_0^{2\pi} \sin\theta d\theta d\phi = 4\pi\,[sr]$$

Ratio (Teilungsverhältniss)

Having three points A_1, A_2, A_3 on a line, we have the ratio

$$\frac{|A_1A_2|}{|A_2A_3|}$$

Crossratio (Doppelverhätniss)

Having four points A_1, A_2, A_3, A_4 on a line, we can define a crossratio

$$\frac{\frac{|A_1A_2|}{|A_2A_3|}}{\frac{|A_2A_3|}{|A_3A_4|}}$$

3 Raster Algorithms

Pixel (**picture element**) a single element of a raster display indexed by row and column (i, j)

Raster rectangular array of pixels

Scanline row of pixels in the raster

3.1 Display Types

3.1.1 Vector Display

advanced oscilloscope, controlled by horizontal/vertical plate voltage, creation of whole objects (i.e. vectors) instead of single pixels

+ high resolution, interactivity, scaling

- few colors, wire frames without surfaces, low complexity, expensive

3.1.2 Raster Display

Cathode Ray Tube (CRT) traditional monitor with blobby pixels associated with a patch of phosphor, that's glow depend on the electron beam's intensity (color CRTs have three beams red, blue, green)

Liquid Crystal Display (LCD) almost perfect squares act as filters, which vary their opacity to darken a back light. They do this by liquifying when shot at with heat

Framebuffer memory array in which an image is stored, before it is displayed on the screen

+ filled surfaces, color variation per pixel (lighting, shading), real time refresh

- aliasing: artifacts, moire patterns, difficult selective update, discrete sampling, jaggies

3.2 Line Rasterization

Given start and end point, we want an algorithm that draws a line between them. Usually only integers are respected (i.e. whole pixels are used for the line).

Ray Acceleration

Draw every pixel the line touches.

+ fast

- ugly

Bresenham Algorithm (Midpoint Algorithm)

Makes use of a implicit form of the line:

$$f(x, y) = (y_0 - y_1)\,x + (x_1 - x_0)\,y + x_0 y_1 - x_1 y_0 = 0$$

where $(x_0 < x_1)$. The key idea of the algorithm is the line's slope m

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

The algorithm assumes the line to proceed more horizontally than vertically from start to end point, so the next pixel is either on the same level $(x+1, y)$ or one above $(x+1, y+1)$. All other cases can be deduced straight forwardly (e.g. for the vertical case, switch y and x. Now the idea is to look at the midpoint between those candidates $(x+1, y+0.5)$ and compute whether the line goes above or below it and make a decision accordingly. We can get the distance between point and line as explained in 2.4 by simply evaluating $f(x+1, y+0.5)$. Since

$x_1 > x_0$, $(x_1 - x_0)$ will always be positive. Thus we can read whether the line is below or above the point, by looking if $(x_1 - x_0)\,y$ has increased or decreased.

Algorithm 1 Bresenham Algorithm

```
    $y = y_0$
    for $x = x_0$to $x_1$ do
        draw$(x, y)$
        if($f\,(x + 1, y + 0.5) < 0$) then
            $y = y + 1$
```

For more efficiency, we can reuse previous results using the following properties
$f\,(x + 1, y) = f\,(x, y) + (y_0 - y_1)$
$f\,(x + 1, y + 1) = f\,(x, y) + (y_0 - y_1) + (x_1 - x_0)$

```
    $y = y_0$
    $d = f\,(x_0 + 1, y_0 + 0.5)$
    for $x = x_0$to $x_1$ do
        draw$(x, y)$
        if($d < 0$) then
            $y = y + 1$
            $d = d + (y_0 - y_1) + (x_1 - x_0)$
        else
            $d = d + (y_0 - y_1)$
```

We still have a real operation when adding 0.5, yet the code uses only integers apart from that. We can outmaneuver this by multiplying with 2.

$$d = 2f\,(x_0 + 1, y_0 + 0.5)$$

$$d = d + 2\,(y_0 - y_1) + 2\,(x_1 - x_0)$$
$$d = d + 2\,(y_0 - y_1)$$

If the line is very diagonal, it will have fewer pixels than a straight line and thus appear less bright. As a solution you may take the distance to the midpoint d and use it to adjust the pixels brightness according to d. For grey-scale color $\frac{1}{\sqrt{2}\cos\alpha}$ has proven to be a good compensation.

3.3 Triangle Rasterization

Gouraud Interpolation

Determine the triangles pixels colors by interpolating the color at it's vertices:

$$c = \alpha c_0 + \beta c_1 + \gamma c_2$$

where (α, β, γ) are the pixel's/point's barycentric coordinates (see 2.3.3).

If the pixel is on the edge of two adjacent triangles, there is no "right one" to assign it to. Therefore we just decide for one of them, as long as the decision is well defined. One solution is to choose a random off screen point, and make the decision depending on it's position.

Antialiasing

The edges of triangles will appear pretty "jaggy" blurry on the screen. A simple solution for this problem is to allow pixels to be half on (αvalue).

Box Filter: One easy method is to underlay a rectangle and use it as a filter, where the pixel's color is set to the average values inside the rectangle.

3.4 Polygon Rasterization

If we are dealing with polygons in general rasterization is getting a bit thougher. Our task is still to draw all pixels within a polygon.

Seed Fill

Algorithm 2 Seed Fill

```
1. draw polygon edges with the Bresenham algorithm (see 3.2)
2. randomly pick a point within the polygon and draw it
3. ↺ recursively check all neighbouring pixels for being inside and draw them
```

- deep recursion (stack overflow), inefficient, no shading

2D Scan Conversion

Use the edges to partition the screen into outcode areas and apply α-clipping (see 3.5), painting every pixel inside.

- a lot of useless computation, highly inefficient

+ slightly better when using small screen bounding boxes, instead of the entire screen

Scanline

The idea is to proceed scanline per scanline from bottom to top, to find intersections with the polygon and draw between the intersection points. The x-value to start the line at can be determined by storing the lowest x-coordinate of the edges and keep this one up to date by adding the reciprocal slope $\frac{1}{m} = \frac{\Delta x}{\Delta y}$ each time we climb a line higher.

Edge Table a list of all edges of the form

y_{lower}	x_{lower}	y_{upper}	$\frac{1}{m} = \frac{\Delta x}{\Delta y}$	$\rightarrowtail$next edge

these nodes are sorted by y_{lower}. $\frac{1}{m}$ is the increment required to step a line higher.

Active Edge Table a list of edges that are intersecting with the current scanline

$x_{\text{intersect}}$	y_{upper}	$\frac{1}{m} = \frac{\Delta x}{\Delta y}$	$\rightarrowtail$next edge

sorted by $x_{\text{intersect}}$. The current intersection point is $(x_{\text{intersect}}, y_{\text{scan}})$

Algorithm 3 Scanline

```
1. initialize Edge Table (ET)
2. set Active Edge Table (AET) to $\emptyset$: AET = NULL
3. draw all horizontal lines
4. $y_{\text{scan}} = y_{\text{lower}}$ of the first ET entry
5. do
     • move all edges with $y_{\text{scan}} == y_{\text{lower}}$ from ET to AET
     • sort AET
     • draw lines:
         – AET[0].x, $y_{\text{scan}}$ to AET[1].x, $y_{\text{scan}}$
         – AET[2].x, $y_{\text{scan}}$ to AET[3].x, $y_{\text{scan}}$
         – ...
     • $y_{\text{scan}}++$
     • remove all edges with $y_{\text{upper}} \leq y_{\text{scan}}$ from the AET
     • $x = x + \frac{1}{m}$
   ↻ while AET$\neq\emptyset$
```

+ fast, efficient, allows a good combination with shading

3.5 Line Clipping

The task of clipping is, to only draw what is inside the visible area (e.g. the rectangle of the monitor). Now we haven given start and end points of lines, if they're both inside the clipping rectangle, we draw the line. Yet even if they are both outside, it is not given, that the line between does not cross the visible rectangle.

Cohen Sutherland

We partition the image into nine areas by lengthening the rectangles edges. Then we assign each area with an outcode (see figure).

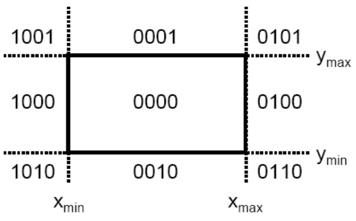


Figure 4: outcodes for the clipping rectangle

The four Boolean correspond to: $|x < x_{\text{min}}|\,|x > x_{\text{max}}|\,|y < y_{\text{min}}|\,|y > y_{\text{max}}|$ where $x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}}$ refer to the lower left and the upper right corner of the clipping rectangle.

Algorithm 4 Cohen Sutherland

1. determine the outcodes for the start and end points P_1 and P_2
2. check `Trivial Accept`: both points are inside

$$\text{outcode}\,(P_1) \vee \text{outcode}\,(P_2) = 0$$

   $\rightarrow$ draw the entire line
3. check `Trivial Reject`: both points are outside in respect to one edge

$$\text{outcode}\,(P_1) \wedge \text{outcode}\,(P_2) \neq 0$$

   $\rightarrow$ draw nothing
4. find intersection points S_1, S_2 where the line intersects with edges. Replace P_i by the nearest intersection point.
   ⥀ restart at 1.

α-clipping

α-clipping adds an Improvement to the Cohen Sutherland algorithm by introducing **Window Edge Coordinates (WEC)** For both points of the line we determine four WECs:

$$\text{WEC}_{\text{left}}\,(P) = p_x - x_{\text{min}}$$

$$\text{WEC}_{\text{right}}\,(P) = x_{\text{max}} - p_x$$

$$\mathrm{WEC}_{\mathrm{bottom}}(P) = p_y - y_{\min}$$
$$\mathrm{WEC}_{\mathrm{top}}(P) = y_{\max} - p_y$$

If $\mathrm{WEC}_E(P) < 0$ then P is outside in respect to edge E. This can be used for an efficient outcode generation.

For α-clipping we choose the parameter form of a line: $\overline{P_1P_2} = \{p = p_1 + \alpha(p_2 - p_1), \alpha \in [0,1]\}$. The value of this parameter α for getting an intersection point S with an edge E can be determined by

$$\alpha_S = \frac{\mathrm{WEC}_E(P_1)}{\mathrm{WEC}_E(P_1) - \mathrm{WEC}_E(P_2)}$$

Algorithm 5 α-clipping

```
1. compute the eight WEC for $P_1$ and $P_2$
2. compute the outcodes (take the sign of the WECs)
3. check Trivial Accept and Trivial Reject
4. $\alpha_{\min} = 0$, $\alpha_{\max} = 1$
5. ↻ for every $E$ where an outcode is set
     • $\alpha_S = \frac{\mathrm{WEC}_E(P_1)}{\mathrm{WEC}_E(P_1) - \mathrm{WEC}_E(P_2)}$
     • if $\mathrm{outcode}_E(P_1) \rightarrow \alpha_{\min} = \max\{\alpha_{\min}, \alpha_S\}$
     • else if $\mathrm{outcode}_E(P_2) \rightarrow \alpha_{\max} = \min\{\alpha_{\max}, \alpha_S\}$
6. if $\alpha_{\min} > \alpha_{\max}$ → return empty line
   else → return $(p_1 + \alpha_{\min} \cdot (p_2 - p_1), p_1 + \alpha_{\max} \cdot (p_2 - p_1))$
```

If we are dealing not with an rectangle, but with a **Convex Clipping Domain** we simply apply α-clipping with one WEC per edge. If however, we have a **Concave Clipping Domain** we have to partition it into convex ones and merge the results.

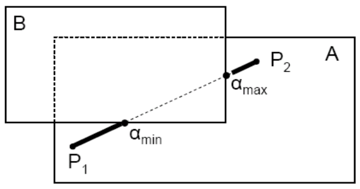


Figure 5: reversed clipping in a x-window system

In X-window systems we often have multiple windows overlapping. In this case we may also apply α-clipping, yet we have to reverse the results (do not draw what's inside).

3.6 Polygon Clipping

Similar to line clipping, but now we have a complete polygon to clip against a clipping rectangle.

Sutherland Hodgeman

The idea is to clip against all edges consecutively and when appropriate add intersection points or polygon vertices to the final set of vertices. Doing this we have to differentiate four different classes:

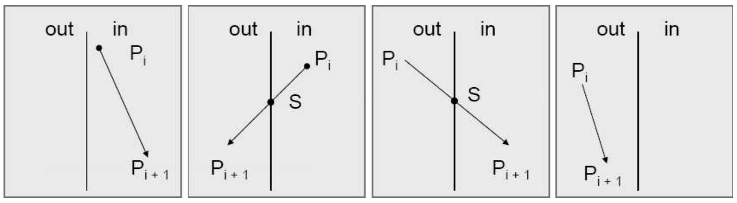


Figure 6: Sutherland Hodgeman Classes

inside/inside add P_{i+1} to the set of vertices

inside/outside compute and add intersection point S

outside/inside compute intersection point S and add S and P_{i+1}

outside/outside do nothing

Doing this check consecutively for all vertices ($P_1P_2 \rightarrow P_2P_3 \rightarrow \cdots \rightarrow P_nP_1$) for all four edges, we can return a set of vertices defining the visible polygon.

3.7 Culling

When an entire triangle lies outside the view volume, it can be culled. Culling means elimination of a triangle or a whole object from the pipeline. See the Chapter about Occlusion & Visibility 7.7.

3.8 Antialiasing

In general aliasing occurs when Nyquist's sampling theorem was hurt (see 11.5.1), therefore the best way, if possible, is to use a higher sampling frequency.

3.8.1 Line

Line's often appear often jagged having aliasing artifacts. Methods to counter this are:

- treat them as a one pixel wide quadrilateral blended with the background
- consider an infinitely thin object with a halo
- use a anti aliased texture

3.8.2 Screen Based

A technique often used for screen based antialiasing is weighted interpolation of neighbouring pixels:

$$p(x,y) = \sum_{i=1}^{n} \omega_i c(i,x,y)$$

where ω_i are weights describing the contribution of a neighbouring pixel and $c(i,x,y)$ returns the color of neighbouring pixel i. Note that the weights have to sum up to 1: $\sum_{i=1}^{n} \omega_i = 1$.
Other methods include

Full Scene Antialiasing (FSAA) Render the image at a higher resolution and average neighbouring pixels. This is usually combines with Mip-Mapping (see 11.5.2), but instead of choosing the Mip-Map level by the longer side of the parallelogram, we take the smaller side and thus choose a Map in higher resolution than usual, but also render the object in higher resolution than the final image on the screen has. Now for every pixel in the screen we check the n closest fragments in the higher resolution image and average them to determine the pixel's value at the current position. The more parallel fragment pipelines we have, the more efficient this n per 1 look-up can be realized.

Anisotropic Filtering see FSAA above

Accumulation Buffer Use the accumulation buffer (see 0.1) to use multiple passes blending over each other

Multisampling compute a polygon's grid coverage

Stochastic Sampling (Jittering) instead of sampling uniformly, sample randomly. This results in uniform noise added to the resulting image, but the human vision system is very forgiving to uniform (or white) noise

Gamma Correction see next chapter 4.8

4 Color

Color as perceived by human being is always a three dimensional problem, since the human eye differentiates three kinds of cones for color perception and rods, sensors detecting brightness and darkness (**Tristimulus Theory**). Ours are especially sensitive to red, green and blue.

8 Bit Byte entries in a pseudo color framebuffer point to a look-up table with color values

24/32 Bit 1 Byte per color (True Color)

Using multipass rendering techniques the color depth becomes especially important. If the precision (depth) used for one pixel's color is to low, nasty visible quantization artifacts will occur.

4.1 Light

Seeing works with light entering the eye hitting the retina. We describe this light signal and its wavelength as the radiance.

radiance $L(\lambda)$ radiance is the intensity of light with a certain direction and wavelength

wavelength $\lambda = \frac{c}{f}$ with c as the speed of light and f as frequency.
light has a huge spectrum of wavelengths, of which a small part are visible as color. Other spectra are UV, infrared, microwaves, radio waves, x-rays, gamma-rays

hue the seen color, i.e. the dominant wavelength. E.g. the hue of pink is red.

saturation color intensity (how far is it from grey of equal intensity)

brightness the light energy, the emitted light. It is also called **luminance**. E.g. darkblue and lightblue.

Response

The human eye perceives light not linearly, but is more sensitive to certain spectra; which also is true for cameras. Therefore we can define a response to light:

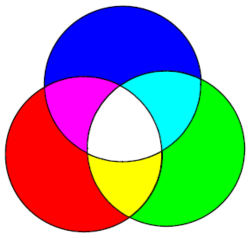
$$\text{response} = k \int w(\lambda)\, L(\lambda)\, d\lambda$$

where $w(\lambda)$ is the response function and k a hardware dependent (organic dependent) constant.

Color

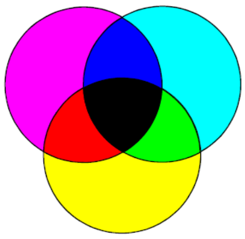
The phenomena of color is based on different wavelengths of light. E.g. red is around $4.3 \cdot 10^{14}$ outcode

4.2 RGB



Using red, green and blue (RGB) as basis colors, we have an additive color system. This is suited for monitors, which work with additive light.

4.3 CMY



Having actual paint, like with printers, we fall back to the well known subtractive color system. For this is is common to use cyan, magenta and yellow (CMY) as the three basis colors.

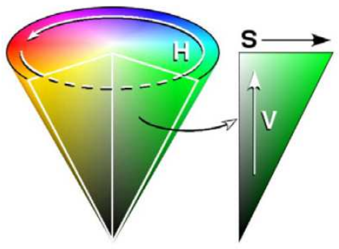
CMYK

Often the CMY model is extended by the color black. This model is mainly used for printing devices, since it would be more costly to mix black out of cyan, magenta and yellow.

4.4 YIQ

The YIQ color model is traditionally used in NTSC-television. The variable Y solely contains the luminance necessary for black & white television and I, Q hold additional color information.

4.5 HSV



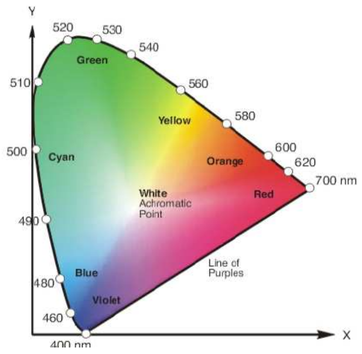
The Hue, Saturation, Value (Brightness) color model is a more intuitive color model derived form the RGB model. In the HSV model color is more defined by it's properties (hue, saturation and brightness) than parted into three different colors.

hue given as angle between $[0°, 360°]$

saturation the distance from grey $[0, 1]$

value from white to black $[0, 1]$

4.6 XYZ



The XYZ color model is based on the Tristimulus Theory and attempts to standardize color values. In contrast to physical reality the CIE[1] developed a model in which any linear combination between colors is possible, even if it is contradicting reality. In addition there is one grey light without hue information (color saturation) and two with zero luminance and only hue information. We have

$$X/Y/Z = 683 \int_{380}^{800} \bar{x}/\bar{y}/\bar{z}\,(\lambda)\, L\,(\lambda)\, d\lambda$$

where 683 is a constant to conform luminance standards and $[380, 800]$ is the range of visible light. Y returns the luminance (brightness). The big advantage

[1]Commssion Internationale de L'Eclairage

of this approach is that in contrast to the previously discussed models, this model is hardware independent.

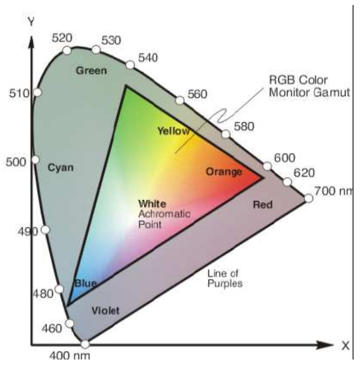


Figure 7: Comparison with the RGB color model

In contrast to other models the XYZ is therefore able to model every visible color, however monitors are not, since they are limited to the colors they can produce by adding three light beams. The triangle represents the monitor gamut.

There is an addition called **isotopic luminance**. This is what you see at night, when you look at a world lit by moonlight. Although it is not possible to deduce the isotopic V directly from X, Y, Z there is a good approximation

$$V = Y \left[1.33 \left(1 + \frac{Y + Z}{X} \right) - 1.68 \right]$$

Alternatively you can add V as fourth value.

4.7 Alpha Blending

The α refers to the degree of visibility of a pixel. If a pixel is only half visible ($\alpha = 0.5$), we want to see half of the pixel behind it (e.g. glass, water). Having c_f referring to the foreground pixel's color and c_b to the color of the background pixel, we get a kind of interpolation:

$$c = \alpha c_f + (1 - \alpha) c_b$$

4.8 Gamma

Monitors are non-linear in respect to the input intensity a (0.5 input intensity can be displayed as 0.25). This degree of freedom is referred to as gamma value γ.

$$\text{displayed intensity} = M \cdot a^\gamma$$

where M is the monitors maximum intensity.

$$a = 0.5^{\gamma} \rightarrow \gamma = \frac{\ln 0.5}{\ln \alpha}$$

To find a you can for example let your monitor display two images: a black & white checkerboard pattern and a grey value image at intensity 0.5. Fiddling on the intensity you can find the grey value that corresponds to the checkerboard, which will also look like grey. Having this you can deduce a.

Having γ we can correct this non-linearity by the transform $a = a^{\frac{1}{\gamma}}$
Without Gamma Correction we will encounter the following phenomena:

- color interpolation is not linear (without Gamma Correction mid tones will appear too dark)
- color fidelity: colors will differ from their true hue
- distance-squared fall-off: colors fade out to darkness way too fast
- dithering: blending of colors (see Screen Door Transparency 7.6), appears only with color depths below 24bit
- aliasing (Attention: gamma anti-aliased images for CRTs look jagged on LCDs)
- problems with the use of anti-aliased textures (MipMapping has to take gamma correction into account)

4.9 Fog

There are four arguments for using fog:

1. increasing realism
2. helps viewer to determine distances
3. helps culling (far objects are hidden in fog), avoids far-plane pop ups
4. implemented in hardware

Adding fog to a scene the user sets two variables

fog color c_f

fog factor $f \in [0; 1]$ decreasing with the distance from the viewer

If c_s denotes the color resulting from shading, the color added fog c_p computes as

$$c_p = f \cdot c_s + (1 - f)\, c_f$$

f computes by the distance given in z-values

$$f = \frac{z_{\text{end}} - z_p}{z_{\text{end}} - z_{\text{start}}}$$

where start and end denote the fogged area or exponentially falling fog

$$f = e^{-d_f z_p}$$

where d_f controls the fog's density.

4.10 Color Conversion

RGB → CMY

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ B \\ G \end{pmatrix}$$

CMY → CMYK

$$\begin{pmatrix} K \\ C \\ M \\ Y \end{pmatrix} = \begin{matrix} \min\{C, M, Y\} \\ C - K \\ M - K \\ Y - K \end{matrix}$$

RGB → YIQ

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \begin{pmatrix} R \\ B \\ G \end{pmatrix}$$

RGP→HSV

$V = \max\{R, G, B\}$
$S = \frac{\max\{R,G,B\} - \min\{R,G,B\}}{\max\{R,G,B\}}$

$$H = \begin{cases} 60 \cdot \frac{G-B}{\max\{R,G,B\}-\min\{R,G,B\}} & \text{if } \max\{R,G,B\} = R \\ 60 \cdot \frac{B-R}{\max\{R,G,B\}-\min\{R,G,B\}} + 120 & \text{if } \max\{R,G,B\} = G \\ 60 \cdot \frac{R-G}{\max\{R,G,B\}-\min\{R,G,B\}} + 120 & \text{if } \max\{R,G,B\} = B \end{cases}$$

RGB → XYZ

Since the XYZ color model is the only discussed model, that is hardware independent, it is hard to convert from the other models, because hardware information is required.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{pmatrix} \begin{pmatrix} R \\ B \\ G \end{pmatrix}$$

where $X_rY_rZ_r$ refers to and description of the monitor's red channel in the XYZ color model. By linear algebra this conversion can be reduced, so that only Y_r, Y_g, Y_b must be known. Additionally those three values can be approximated numerically, when now hardware information is given. However the XYZ scale can be directly converted to grey scale RGB color

$$Y = 0.2125R + 0.7154G + 0.0721B$$

5 Transformation Matrices

In general we want to use matrices to change a set of vectors representing an object.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} + x & a_{12} + y \\ a_{21} + x & a_{22} + y \end{bmatrix}$$

We can partition these changes into categories:

Types

- **Rigid Transformations**: preserving distances and angles
  - identity, rotation, translation
- **Similitudes**: preserving angles, preserving width/height ratio
  - isotropic scaling
- **Linear Transformations**
  - scaling, reflection, shearing
- **Affine Transformations**: parallel lines remain parallel, line ratios are preserved
  - translation
- **Projective Transformations**: parallel lines intersect at points at infinity, preserves cross ratio)
  - projective transformation

5.1 Scaling

Scaling changes length and direction.

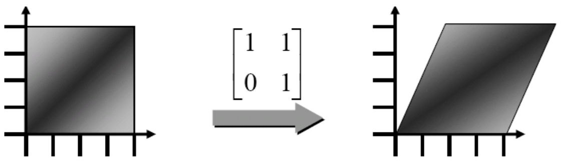


Figure 10: horizontal shearing

isotropic scaling

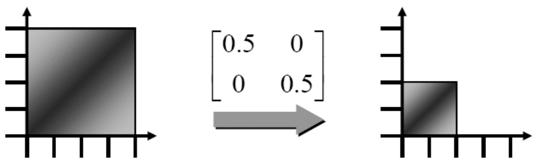


Figure 8: Isotropic Scaling

$$\mathrm{scale}\,(s) = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$

scaling

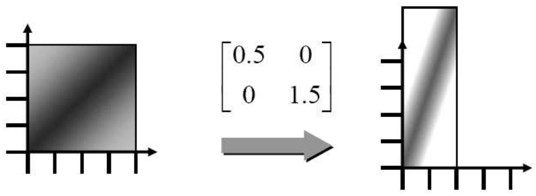


Figure 9: scaling

$$\mathrm{scale}\,(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

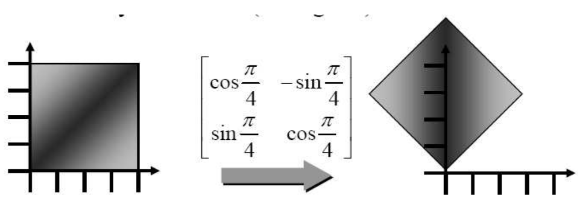
5.2 Shearing

Shearing pushes objects sideways.

horizontal $\mathrm{shear}\,(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$

vertical $\mathrm{shear}\,(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$

5.3 Rotation



$\begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix}$

Figure 11: Rotation

Rotation rotates a vector around a certain angle. We rotate around the origin.

$$\text{rotate}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

5.3.1 Arbitrary Rotations In 3D

Orthogonal Matrices

3D rotation matrices are orthogonal and preserve the orientation.

$$O^T \cdot O = I$$

$$\det(O) = 1$$

The rows are three arbitrary orthogonal unit vectors (i.e. orthonormal) and the columns are three different orthogonal unit vectors (i.e. orthonormal):

$$R_{uvw} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}$$

with

$$\vec{u} \cdot \vec{u} = \vec{v} \cdot \vec{v} = \vec{w} \cdot \vec{w} = 1$$

$$\vec{u} \cdot \vec{v} = \vec{u} \cdot \vec{w} = \vec{v} \cdot \vec{w} = 0$$

Therefore

$$R_{uvw} \cdot \vec{u} = \begin{bmatrix} \vec{u} \cdot \vec{u} \\ \vec{v} \cdot \vec{u} \\ \vec{w} \cdot \vec{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = x$$

and $R_{uvw} \cdot \vec{v} = y$, $R_{uvw} \cdot \vec{w} = z$

Note The inverse of an orthogonal matrix is its transpose: $R_{uvw}^{-1} = R_{uvw}^T$

Rotation About An Arbitrary Axis/Vector

So we have found out that we can create arbitrary rotation matrices from a orthonormal basis. If we want for example to rotate about an arbitrary axis/vector $\vec{a}$, we

1. build an orthonormal basis with this vector $\vec{w} = \vec{a}$
2. rotate the uvw basis to the canonical basis
3. rotate around the z-axis
4. rotate back to the uvw basis

Three Euler Rotations

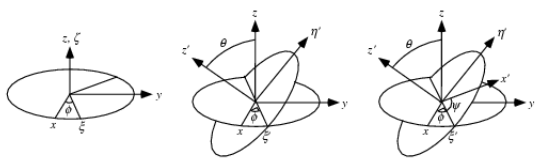


Figure 12: Euler Rotations

Euler found out that any rotation in 3D can be described using three angles (ϕ, θ, ψ) (the **Euler Angles**). If we put these into rotation matrices, we can make one matrix out of them: $A = BCD$

Usually the first rotation ϕ is about the z-axis, the second θ about the x-axis and the third ψ about the z-axis again. This draws

$$B = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$

$$D = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The theory behind is that we are using a local coordinate e'_x, e'_y, e'_z system defined by the three Euler angles

$$\phi = \langle(e_z, e'_z)\rangle$$

$$\theta = \langle(e_x, L)\rangle$$

$$\psi = \langle (L, e'_x) \rangle$$

where L is the intersecting line between $e_x e_y$ and $e'_x e'_y$. So instead of a real rotation, we are just translating another coordinate system and the Euler angles store the relationship between both coordinate systems.

- There is a problem with Euler angles called **gimbal lock**: We want to rotate around the z-axis. First we rotate by 90° around the x-axis (pitching), make no y rotation, finally any rotation around the z-axis. At this point this z-axis rotation actually corresponds to a rotation around the y-axis[2]. This problem does not appear when using Quaternions (see below).

Rotation Axis And Angle

In this method we describe an arbitrary rotation be giving an axis n and an angle ω.

1. partition a vector $\vec{x}$ into a parallel part $x_{\|} = \langle x \mid n_0 \rangle \, n_0$ and an orthogonal part $x_{\perp} = x - x_{\|}$
2. rotate the orthogonal component: $\vec{x} = x_{\perp} \cos(\omega) + (n_0 \times x_{\perp}) \sin(\omega)$
3. add the parallel part: $\vec{x} = \vec{x} + x_{\|}$

This can of course be represented by a matrix by mapping unit vectors.

If we are given an orthogonal matrix O we can find the axis of rotation n by looking for the eigenvector to the eigenvalue of 1. The angle ω can be determined by:

$$\omega = \arccos\left(\frac{\operatorname{trace}(O) - 1}{2}\right)$$

because

$$\operatorname{trace}(O) = 1 + 2 \cdot \cos(\omega)$$

Complex Numbers and Quaternions

Any 2D rotation can be described by a complex number z_0

$$z = n \cdot z_0$$

And any 3D rotation of point $\vec{v}$ about axis $\vec{n}$ and angle ω can be described by a quaternion q (see 2.10)

$$\operatorname{rotate}(\vec{n}, \omega) = \cos\frac{\omega}{2} + \sin\frac{\omega}{2} \cdot \frac{\vec{n}}{\|\vec{n}\|}$$

[2]Try it with your head. Pitching / x-axis rotation means moving your head towards your shoulder. If you now try to rotate it in z-direction (forward), you actually rotate around your y-axis (imagine an extension of your spline through your head.

the result is of the form

$$q^{-1} \cdot (0, \vec{v}) \cdot q$$

Matrix representation can be acquired mapping the three unit vectors

Axis n and angle ω can also be acquired easily.

$$\vec{n} = \vec{q}$$

$$\cos\left(\frac{\omega}{2}\right) = \frac{q_0}{|q|}$$

Two Planar Reflections

A rotation has the form

$$\text{rotate}\,(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

a reflection can also be written in terms of trigonometry

$$\text{reflect}\,(\phi) = \begin{bmatrix} \cos 2\phi & \sin 2\phi \\ \sin 2\phi & -\cos 2\phi \end{bmatrix}$$

Therefore we get the equation

$$\text{reflect}\,(\theta)\,\text{reflect}\,(\phi) = \text{rotate}\,(2\,(\theta - \phi))$$

allowing us to express any rotation by two planar reflections.

Note that rotation matrices have a determinant of 1 while reflection matrices have determinant -1.

5.4 Reflection

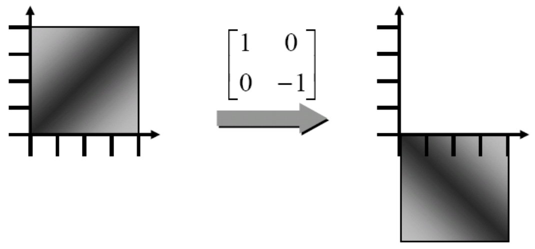


Figure 13: Reflection

Reflecting an object on an axis.

x-axis $\text{reflect}\,(s) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

y-axis $\text{reflect}\,(s) = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$

Later we will introduce an ordering of vertices of a triangle (see 10.1.8). A reflection might distort this ordering resulting in wrong illumination and lighting. To determine whether a matrix is reflective, compute it's determinant and check the sign: -1 means reflective.

5.5 Translation

The problem with translation is that talking about the other transformations we have seen every vector as offset from the origin, what makes them unmovable. Therefore we artificially move a dimension up using homogeneous coordinates:

Translation usually is performed by adding a translation vector $\vec{t}$:

$$\begin{bmatrix} a_{11} & a_{12} & t_1 \\ a_{21} & a_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_1 \\ y + t_2 \\ 1 \end{bmatrix}$$

after dehomogenization, we have what we wanted.

5.6 Composition of Transformations

A composition of matrix transformations corresponds to a matrix multiplication of the transformation matrices involved. However matrix multiplication is not commutative meaning it does matter whether you do a rotation before scaling or a scaling before rotating.

Because it is associative we can combine all transformations into a single matrix and use this matrix to transform all involved vectors only once.

Note matrices are multiplied from right to left

$$M = RS$$

means first a shearing is applied and then a rotation.

Decomposition

The opposite is of course possible as well. Using for example SVD (see 2.2.2) we can decompose the matrix into a diagonal part (reflection and scaling) and orthonormal/orthogonal parts (rotation). Interesting is that any transformation can be decomposed into two rotations and one scaling:

$$A = R_2 \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} R_1$$

A rotation on the other side can be decomposed into three consecutive shearings:

$$\begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} 1 & \frac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\phi & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{bmatrix}$$

This is important since shearing is a very effective raster operation.

5.7 Transforming Normal Vectors

One problem is that we cannot apply the same transformation matrix both to the object and to the object's normal vectors. Consider a shearing, the direction of the y-vectors are not changed, yet the form changes and the y-normal vector's direction is no longer perpendicular to the surface.

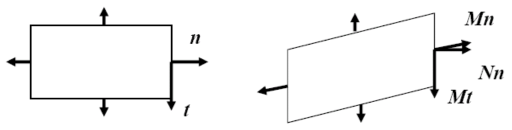


Figure 14: Transforming Normals
The normal vectors are dealt wrongly with the transformation matrix applied to the object

Therefore we need to deduce separate transformation matrices for the normal vectors. We start with the fact, that the normal vector $\vec{n}$ and a tangent vector $\vec{t}$ are perpendicular:

$$\vec{n}^T \cdot \vec{t} = 0$$

We add an identity matrix

$$\vec{n}^T \cdot \vec{t} = \vec{n}^T \cdot I \cdot \vec{t} = \vec{n}^T M^{-1} M \cdot \vec{t} = 0$$

and change the order

$$\left(\vec{n}^T M^{-1}\right)(Mt) = \left(\vec{n}^T M^{-1}\right)\vec{t}_M = 0$$

$$\vec{n}_M^T = \vec{n}^T M^{-1}$$

$$\vec{n}_M = \left(\vec{n}^T M^{-1}\right)^T = \left(M^{-1}\right)^T \vec{n}$$

$$N = \left(M^{-1}\right)^T$$

If however we know our matrix to be orthogonal (e.g. it has been formed by rotations), we can take the matrix itself for transforming the normals, because the inverse of an orthogonal matrix is its transpose and two transpositions cancel each other out. Finally since rotations and translations are rigid body transforms (the shape is not changed) the matrix will return a unit normal vector. In case of uniform scaling, the matrix can be used to transform the normals, yet the resulting normals have to be normalized (they are no unit vectors).

5.8 Windowing Transforms

Often we will need to scale a window in a X-window system. The most easy way to create a correct matrix for this task is to see it as three different transforms.

1. move the lower left point of the window to the origin
2. scale the window rectangle
3. move the lower left point to the target position

5.9 Inverse Transformations

Since ordinary matrix inversion is a costly operation, we can use background knowledge to quicken the inversion:

translation $\vec{t} \to -\vec{t}$

rotation $R \to R^T$

scaling $\text{scale}\,(s_x, s_y, s_z) \to \text{scale}\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$

Curiously enough we can make use of the SVD, because of the fact that we can partition any transformation into

$$M = R_1 \,\text{scale}\,(s_x, s_y, s_z)\, R_2$$

the inverse is simply

$$M^{-1} = R_1^T \,\text{scale}\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right) R_2^T$$

5.10 Projective Transformations

Projective Transformations have 1,2 or 3 **vanishing points**. These are points where virtually parallel lines intersect in the perspective space. These 3 vanishing points v_i can be found at the bottom row of or homogeneous 3D transformation matrix.

$$\begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ v_1 & v_2 & v_3 & 1 \end{bmatrix}$$

t_i describe a translation of the object.

5.11 Big M

All together we now have a matrix of the form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ v_1 & v_2 & v_3 & 1 \end{bmatrix}$$

where a_{ij} denote the linear part for all linear transformations, t_i the affine part for translations and v_i the part for projective transformations.

6 Viewing

In this chapter orthographic and perspective projection as well as dealing with occlusion and hidden lines is discussed.

orthographic three dimensional objects are displayed on the two dimensional screen, but without perspective viewing. That means parallel lines are still parallel in the orthographic 3D model

perspective perspective also refers to the displaying of 3D objects, yet as seen by a camera/eye. That means parallel lines will intersect at some point on the horizon.

occlusion When modeling 3D scenes some objects will be in front of others, some will be partially occluded. We got to find out which are in front.

hidden lines both in orthographic and perspective transformations we will have to deal with hidden lines (e.g. of wire frames). Since this easily leads to artifacts and a wrong perspective/orthographic impression we will discuss methods to deal with these phenomena

6.1 Canonical View Volume

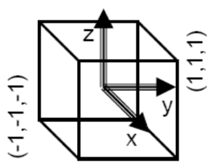


Figure 15: Canonical View Volume

The canonical view volume refers to a cube with the dimensions $(x, y, z) \in [-1, 1]^3$. It serves as a intermediary between any viewing transformation and the screen (Clipping is much easier inside this volume). Let n_x, n_y be the pixels on the screen, then $x = -1$ will be mapped to the left half, $x = 1$ to the right half, $y = -1$ to the bottom half and $y = 1$ to the top of the screen.

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}$$

maps the pixels of the canonical view volume to real pixel centers ($[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$) of screen pixels.
Coordinates inside this volume are called **normalized device coordinates**.
Advantages of using this intermediary step are

- the transformation can be expressed as a 4×4 matrix

- projection to the 2D screen becomes easier (throw away z)
- clipping against the unit cube is more efficient than against the frustum
- maintains relative depths (important for the Z-Buffer)

6.2 Orthographic Projection

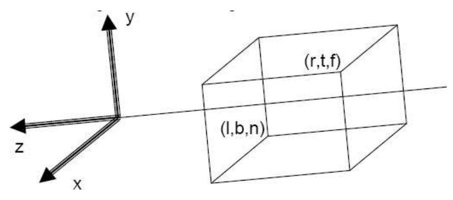


Figure 16: Orthographic View Volume

Having a general **orthographic view volume** we differentiate the six planes of it by:

left plane $l = x$

right plane $r = x$

bottom plane $b = y$

top plane $t = y$

near plane $n = z$

far plane $f = z$

Note that $n > f$!
Usually the camera's or user's head is pointing to the y-direction and looking into $-z$-direction. Furthermore the y-direction in upwards, x-direction is sidewards and z-direction in/outwards.

Mapping to the Canonical View Volume

For that we first move the orthographic view volume to the origin and then do a scaling:

$$\begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

If the matrix that maps the canonical view volume to screen coordinates is added at the left, we can directly map to screen coordinates. The resulting matrix is called M_o and we get:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = M_o \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

6.3 Viewing Direction

We often want to change the viewing direction, if e.g. the user or camera moves it's head. For specifying the view direction we define three variables:

eye position e, the position the eye sees from

gaze direction g, the direction the viewer is looking

view-up vector t, any vector bisecting the viewer's head, where "up" is for the viewer

Furthermore we define a special coordinate system for viewing with axes u, v, w and the origin at e. Then we get:

$$w = -\frac{g}{\|g\|}$$

$$u = \frac{t \times w}{\|t \times w\|}$$

$$v = w \times u$$

Mapping the viewing coordinates to the orthographic view volume

Again we first move the viewing coordinate system to the origin of the orthographic view volume and then align uvw to xyz.

$$M_v = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And again we can combine the latter two matrices to directly transform to screen coordinates:

$$M = M_o M_v$$

6.4 Perspective Projection

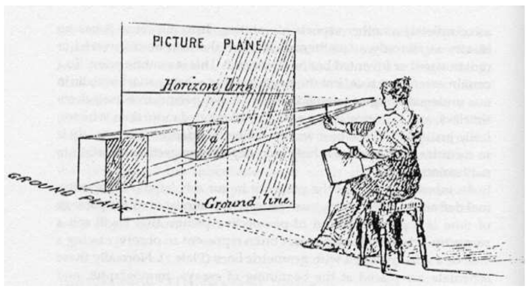


Figure 17: Image Plane

The mathematical idea is to think of an image plane between the viewer and the object. Now for every point in the viewing plane, think of a line pointing directly to the viewer's eye. This line intersects with some point of the object. Draw this point on the pixel the line started from.

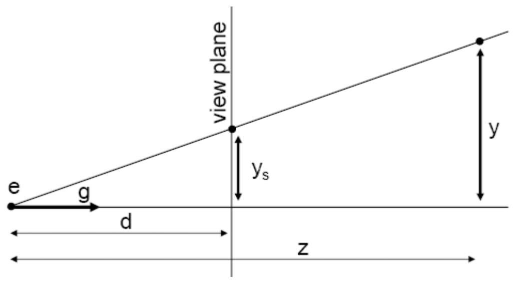


Figure 18: Image Plane with variables

$$y_s = \frac{d}{z} y$$

The key of dealing with perspective projections are the homogeneous coordinates (see 2.3.4). They allow to use linear functions i.e. matrices even for those transformations. If we are done with our object transformations we can use a projective matrix M_p for mapping to the orthographic view frustum:

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & \frac{1}{n} & 0 \end{bmatrix}$$

because of the dehomogenization (division by w), we can scalar multiply M_p by to make it more pretty (no divisions):

$$M_p = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The third entry in the last column has a special meaning. Since applying perspective projection depth information would get lost, we use this position to map the original z-value to the homogeneous slot w. If we later dehomogenize, we effectively divide by the $z-$coordinate and get the perspective view effect.

Mapping the perspective model to the orthographic view frustum

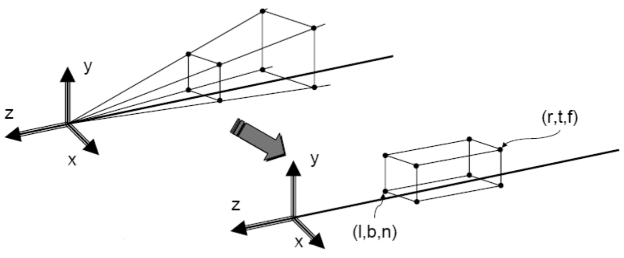


Figure 19: Perspective Projection

Thanks to the homogeneous coordinates we still found a matrix to map back to the orthographic view frustum. Therefore once again we can combine our matrices to directly map to the canonical view volume. Note that in general this cannot be done, because the perspective matrix destroys angles and ratios, which we would need for lighting calculations. So in general we will perform the lighting stage after model and view and then apply the perspective matrix.

$$M = M_o M_p M_v$$

$$M = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2n \cdot f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Items

Diagonal transforming the view frustum range $[-\infty, \infty]$ to the canonical view volume $[-1, 1]$. $\frac{1}{\Delta x}$ gives $[0, 1]$ and the 2 in the enumerator gives $[-1, 1]$. The n and $n + f$ is because of the perspective.

$\frac{r+l}{r-l}, \frac{t+b}{t-b}$ shearing factors. The frustums side planes needs to be sheared to a cube. n, f are already sheared correctly.

-1 very important. By this factor the $z-$coordinates are written to the $w-$coordinate. That means dehomogenizing means division by z and therefore getting the perspective into 2D coordinates.

$\frac{2n \cdot f}{n-f}$ As mentioned above we will divide by z when dehomogenizing. That would mean to loose our z-values. Therefore this term allows us to have the original z-values (depth values) in the $z-$coordinate after dividing it by z. This is important, because we will later need these depth values, when we want to determine which object to draw (i.e. filling the Z-Buffer).

Properties

- maps lines to lines, triangles to triangles and planes to planes
- point/vector ordering may change (because ∞ is mapped to a finite point)
- line segments can be split (because ∞ is mapped to a finite point)
- maps parallel lines to lines intersecting at infinity
- points at infinity (vanishing points)

6.5 Field Of View (Camera Transformations)

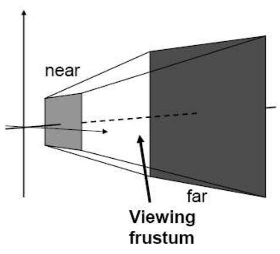


Figure 20: Field Of View

A camera is defined by intrinsic and extrinsic parameters:

extrinsic position and rotation

intrinsic focal length (Brennweite) and aperture (opening)

Extrinsic Transformations

Camera position is at the origin, view direction is $-z$ and up is y. Extrinsic transformations change these three values.

`OpenGL gluLookAt(`$\text{eye}_x, \text{eye}_y, \text{eye}_z, \text{at}_x, \text{at}_y, \text{at}_z, \text{up}_x, \text{up}_y, \text{up}_z$`)`

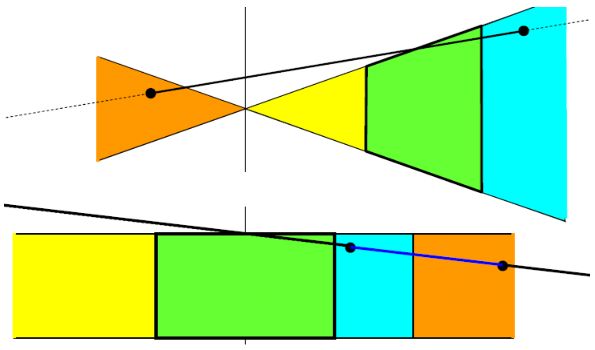


Figure 22: Mapping Of Z illustrated with colors
The vertical line represents the position of the eye. The orange part behind the eye is mapped beyond the blue part representing everything beyond the far plane+∞. The green part is the view frustum and below the canonical view volume. The yellow part between the eye and the near plane remains.

Intrinsic Transformation

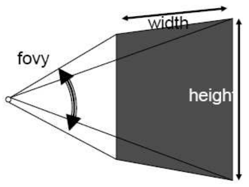


Figure 21: View Frustum

Describes the projection frustum. The image plane is located at $n = -z$. The viewing frustum is important, because it corresponds to the visible world. Everything outside will be clipped to this frustum.

OpenGL `gluPerspective(fovy, aspect, near, far)` or `glFrustum(left, right, top, bottom, near, far)` for asymmetric frustum

fovy opening angle of frustum along y-axis (typical $45° - 60°$)

aspect $\frac{\text{width}}{\text{height}}$, e.g. $\frac{800}{600} = 1.\bar{3}$

near/far distance between origin and near/far plane, e.g. $n = 10cm, f = 100m$

Note a tighter view frustum makes the occlusion test (Z-Buffer) easier

6.6 Mapping Of Z

Due to homogeneous coordinates and the special normalization matrix storing values of z in w, we have to know what happens with out z-values. What will

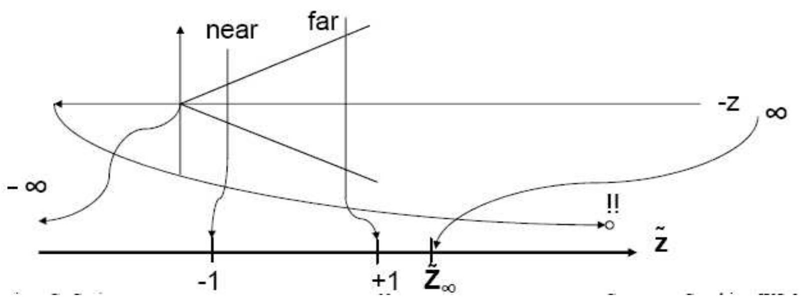


Figure 23: Mapping Of Z illustrated with arrows
The arrows indicate where the corresponding points are mapped to.

happen is that every z-value behind the eye will be mapped to beyond $+\infty$. But since we have $+\infty$ as a vanishing point (last row, third column of the projection matrix), we get finite points for these values again. Instead the eye point is mapped to $-\infty$ when dehomogenizing the coordinates.

Consequences

- ordering of points on a line changes
- clipping draws wrong results (see below)

6.7 Clipping In Homogeneous Coordinates

As mentioned above the ordering of points may change with perspective projection. If we would do a standard clipping we miss parts of line segments, that had been split during perspective projection. For example imagine a line starting from a point behind the eye and ending inside the frustum. After dehomogenization the point behind the eye will be mapped to some positive point behind the far plane, the point in the frustum will remain. Displaying the line between both points we get a line that wrongly leave through the far plane instead of the near plane. This line will be wrongly clipped by the far plane.
The solution is to perform clipping in homogeneous coordinates.
That means we perform standard α-clipping to the six faces of the unit cube resulting in six WEC per point: $\text{WEC}_r\,(x, y, z, w) = w - x$, $\text{WEC}_l\,(x, y, z, w) = w + x$, $\text{WEC}_t\,(x, y, z, w) = w - y$, $\text{WEC}_b\,(x, y, z, w) = w + y$, $\text{WEC}_f\,(x, y, z, w) = w - z$, $\text{WEC}_n\,(x, y, z, w) = w - z$

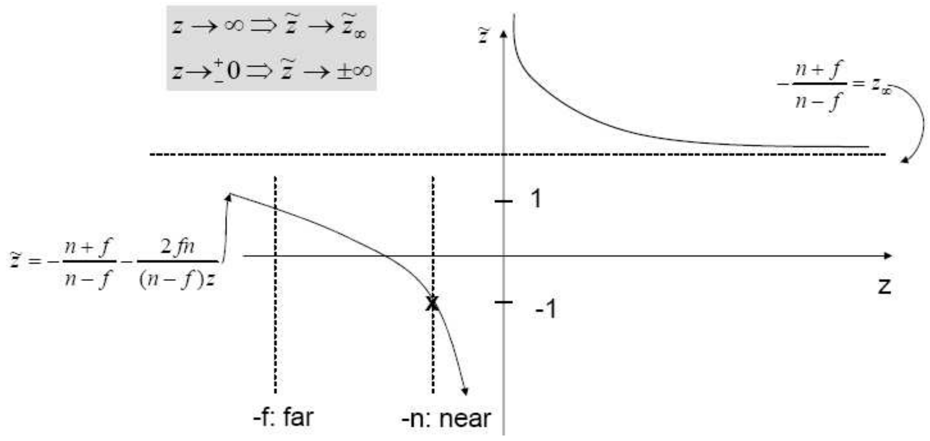
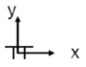


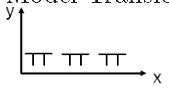
Figure 24: Mapping Of Z illustrated with asymptotes
z: before the projection
$\tilde{z}$: after the projection
The asymptotes indicate how the values behave after the projection. The left most point of the graph is connected with the right most point, mapping infinity to a finite point. On the other hands finite points between the near plane and the eye ($\tilde{z}$-axis) are mapped to infinity.

6.8 Viewing Pipeline

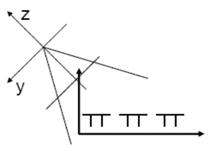
1. Geometric Transformation, Lighting, Clipping → *model coordinates*



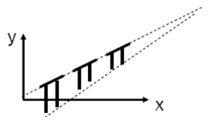
2. Model Transformation → *world coordinates*



3. Viewing: Camera Transformations → *camera/eye coordinates*



4. Perspective Transformations



   (a) Normalization Transformation → *normalized homogeneous coordinates*
   (b) Clipping In Homogeneous Coordinates
   (c) Dehomogenization → *screen coordinates*

5. Viewport Transformation → *window coordinates*

6. Rasterization → *device coordinates*

7 Occlusion & Visibility

In scenes we will always be faced with the problem of multiple objects occluding each other. So we need a way to determine which object is at front and shall be painted.

7.1 Painter's Algorithm

1. sort objects from back to front
2. render them in this order

In this way front objects will simply be painted over the back ones.

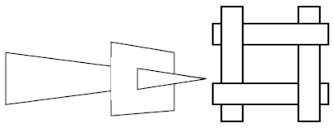


Figure 25: penetration and cyclic occlusion

- painting the back objects is unnecessary
- the sorting of several million triangles is highly inefficient
- cannot handle penetration and cyclic occlusion

7.2 Binary Space Partitioning (BSP)

Binary Space Partitioning is a kind of painter's algorithm, but much more effective, since it lacks the disadvantages of sorting, unnecessary painting and can handle penetration and cyclic occlusion.
The idea is to use the implicit representation of a plane (see 2.4) to make advantage of the easy way to access distances to this plane. Now we pick a triangle that best subdivides the scene in half[3] and build a plane, so that the

[3]usually the triangle that's plane has the lowest number of intersections, this strategy is called the **least crossed criterion**

triangle completely lies on this plane. We assume for now that no other triangle penetrates this plane. Now depending on the eye position e we can decide the safest drawing order (first draw triangles on the side where the eye is located). Thinking of this process recursively like a tree (the BSP-Tree) we can give an overall ordering for all triangles.

Penetration

We assumed that no triangle penetrates our plane, still we can handle this case by cutting the penetrating triangle into two and handle them as three separate triangles, which we add to the BSP-Tree.

Slivers

More often than assumed it will come to a case where a triangle only penetrates a plane tightly with a vertex. In this case the triangle will be cut into three triangles. One of which is a sliver and one of almost zero size. We do better in detecting this special case and leave the triangle untouched.

Axis Aligned Binary Space Partitioning

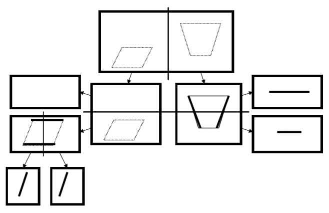


Figure 26: Axis Aligned Binary Space Partitioning

Alternatively we can use planes that cut the complete scene in half and test against them (see figure). Always the line is chosen that subdivides the scene best into halves (horizontal or vertical (axis aligned)).

7.3 Ray Tracing

Cast a ray through each screen pixel to find the first intersection (for Ray Tracing see 9)

7.4 Z-Buffer

The idea is to create a buffer equal to the size of the image, where there is depth information stored for every pixel. The depth value corresponds to the z-coordinate after normalization. Occlusion checks can than be performed by simple depth comparison. The buffer is initialized by the far plane.

Creation The Z-Buffer is filled during rendering/rasterization. The first object's depth values drawn on the screen are stored in the Z-Buffer. Afterwards in case of an successful test (the new object's depth value is closer to the viewer), they are overwritten.

Usage It is used when it is filled, during rendering/rasterization. Check a pixel's depth value, if it is closer to the screen draw it and update the Z-Buffer value for this pixel, otherwise discard it.

Deletion After rendering the buffer's allocated memory can be freed.

Issues & Strengths

- requires fast memory

- non-uniform depth values. The closer we are to the far-plane the smaller the difference in the depth values become. Therefore precision is of fundamental importance for Z-Buffering

- the precision is hardware dependent (the more far away near and far plane are, the more precision comes into play) $\rightarrow$ depth difference becomes very small for distant objects (try to move the near plane as close as possible to the far plane! moving the far plane on the other hand does not help)

- Z-Buffer fighting

+ can handle cyclic occlusion and penetration

OpenGL

```
glEnable(GL_DEPTH_TEST)
glClear(GL_DEPTH_BUFFER_BIT)
```

Strategies

Z-Buffer Fighting This phenomena means the situation when the value in the Z-Buffer so much resembles the current one, that a jumping for and back with each screen update can be seen.

Polygon Offset Polygon Offset means to push certain values inside the depth buffer a little towards the near plane. Since they are not touched on the real framebuffer, this change cannot be seen. Still once the next depth check occurs, this offset will come in handy and avoid Z-Buffer fighting.

[A] Hidden Line Rendering (Wire frame Rendering)

1. render polygon as a wireframe (render to the Z-Buffer only, not to the screen)
2. render the polygon a second time as a solid, using a polygon offset (pushing each depth value in the Z-Buffer towards the near plane by this offset)

[B] Haloing Make use of gaps between hidden lines to emphasize depth perception and avoid Z-Buffer fighting

1. render the polygon as wireframe using thick lines (render only to depth buffer, not to screen)
2. render the lines again with normal thickness and polygon offset

This can also be done with two different kind of colors instead of different thickness.

7.5 W-Buffer

The W-Buffer is an alternative to the Z-Buffer. Instead of depth values the homogeneous w coordinate is stored. The advantage of this method is having uniform "depth" values. Thus we do not need to pay attention to the distance between near and far plane. However a disadvantage is that we cannot linearly interpolate between those values of w (logarithmic scale, because of perspective projection).

7.6 Transparency

Instead of occlusion objects from behind might shine through objects in front, if these are transparent (e.g. a tree behind a window).

Screen Door Transparency

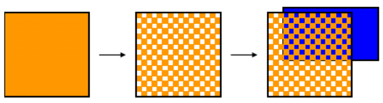


Figure 27: Screen Door Transparency

Instead of a solid objects, the transparent object is drawn as a checkerboard pattern, where the number of gaps depends on the α-value. By this technique objects behind may shine through these gaps.

- worsens if more than one objects can be seen through the transparent objects.
- if two overlapping transparent objects share the same α-value, they have the same number of gaps on the same position and the rear one cannot be seen

Blended Transparency

Draw the objects in the order given by the depth test.

- The depth buffer draws the foremost object first. That means, given the scenario, the transparent object is drawn before the opaque one occluding it, no transparent effects will be seen.

Delayed Blended Transparency

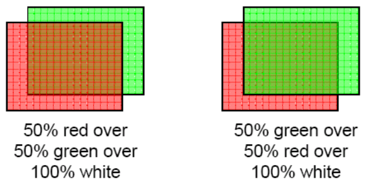


Figure 28: Problems with Delayed Blended Transparency

Draw opaque objects first, then continue with depth buffer test for transparent objects.

- still wrong results when transparent objects in front are rendered before transparent objects in the rear, because blending is not commutative

Sorted Blended Transparency

Draw opaque objects first, then sort the transparent ones from back to front.

- still problems occur when objects intersect and no precise ordering can be given

7.7 Culling

When an entire triangle lies outside the view volume, it can be culled. Culling means elimination of a triangle or a whole object from the pipeline. However in practice perfect culling (i.e. of every single triangle, primitive) is more expensive than letting the Clipping module take care of them. Yet if we use bounding volumes around groups of triangles culling can become very useful. We only check whether a whole bounding volume lies outside our volume, eliminate if it does or pass it on to Clipping if it doesn't.

Culling in hardware is very difficult, because it is not supported pretty well and the entire scene has to be known. So often it is performed in the application stage, where the entire scene is known or solved by precomputation. However it can and is performed at any stage of the pipeline as well.

7.7.1 Back Face Culling

The idea of back face culling is that only those faces can be seen, that are facing towards the user. Other will face the backside of objects and this be invisible to the viewer. The orientation of a face can be checked by examining the outwards facing normal of it in respect to the view vector. The face is visible if

$$\vec{v} \cdot \vec{n} > 0$$

Dealing with polygons the orientation is implicitly coded in the ordering of the vertices (see 10.1.8). The orientation can be tested using the vector product (thumb, indexing finger and middle finger). A polygon $\triangle_{abc}$ with $p = a - b$ and $q = c - a$ is ordered counter clockwise if

$$(p \times q)_z > 0$$

The question remaining is when to perform back face culling.

world space before the scene is transformed to screen space

+ fast (faces are sorted out quickly)

- real normal is needed

screen space (after screen space transformation)

+ no normal needed, more general

+ supported by OpenGL

- more expensive

Clustered Back Face Culling

A small extension to standard back face culling where polygon groups sharing a similar normal are either rendered together or discarded altogether (If part of the groups is front facing, the other not, all are rendered). These "normal sharing" mathematically effects in a **normal cone**, a truncated cone containing all the normals and points of a set. Now the viewing direction is compared to the normals of the cone, and all points associated to normals which differ significantly are discarded. The normals $\vec{n}$ of a cone are front facing the viewer $\vec{v}$ if

$$\vec{n} \cdot \left(\frac{\vec{v} - \vec{f}}{\left\| \vec{v} - \vec{f} \right\|} \right) \geq \sin(\alpha)$$

where α is the opening angle of the cone. This works because even if only part of the polygons are facing towards the viewer, the unincidentally rendered are later discarded during clipping.

7.7.2 View Frustum Culling

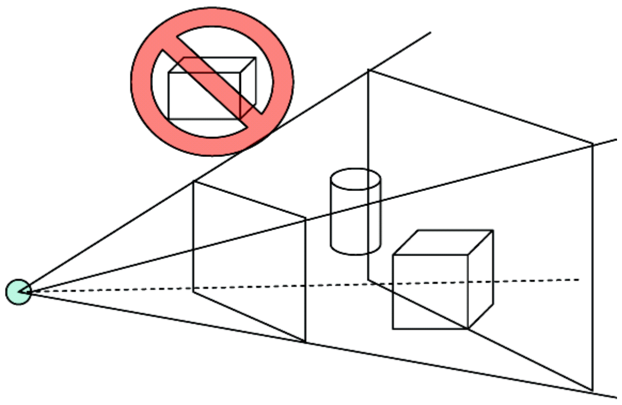


Figure 29: View Frustum Culling
all objects outside of the frustum are culled

The idea of view frustum culling is to check objects against the view frustum, i.e. against what the view is actually seeing. Doing this by polygons would be tedious and in fact the efficiency gain is worse than skipping culling altogether and let the clipping module clip the frustum. However if we see the objects as complete single entities, we gain a efficiency bonus.

Bounding Volumes

For that we must enclose them into geometric bounding volumes (e.g. cube, sphere). Now we check the bounding volume by simple implicit geometry and cull the object if frustum and volume share no common points and render it if they share all or only some points. In the latter case we again leave the precision work to the clipping module.

Bounding Sphere Finding sphere exactly fitting the object is complex and since we look for speed, we go with sphere that will be bigger than the object but easily computed: We take the center of mass as the center of the sphere and choose the radius to cover all object vertices. A great advantage of bounding spheres is that they are invariant to rotations.

Axis Aligned Bounding Box Creation is easy. A box is described by six values: $x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}, z_{\max}$. These values are simply the maximum/minimum of all vertices, e.g. $x_{\min} = \min\{x\text{-value of all vertices}\}$. A disadvantage is that axis aligned bounding boxes are not invariant to rotations and must be adapted for rotation. Like the sphere approach, this approach results in much bigger boxes than objects inside.

Oriented Bounding Box A bounding box tightly fit to the object which considers rotation. However the computation and intersection tests are more difficult. Computation include: finding the center, compute the covariance matrix (think of the center as mean), Eigenvalue analysis, use Eigenvectors as directions (basis) and use them a box that includes all vertices of the object.

Bounding Hierarchies

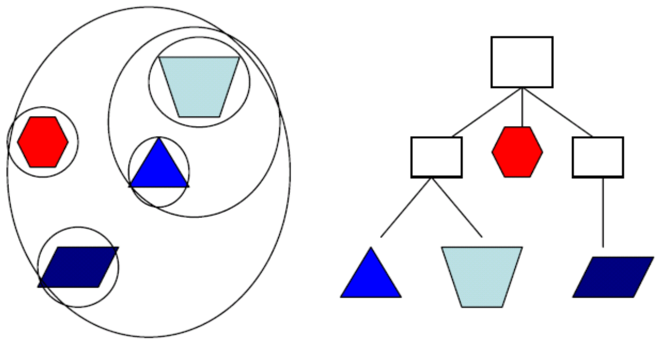


Figure 30: View Frustum Culling
all objects outside of the frustum are culled

Even more can be gained if we build a hierarchy of bounding volumes, i.e. a bounding volume around several bounding volumes. If a big volume lies completely in the volume, we render all its children objects; if it shares points with the frustum, we check each volume inside it separately, if not all objects inside are culled.

Other hierarchies can also be applied. They are discussed later on in the chapters about Ray Tracing, see 9.3 and the chapter on Occlusion for Binary Space Partitioning , see 7.2. However they usually perform worse under dynamic changing scenes. On the other hand the deliver better results and should be used for static scenery.

Intersection Test

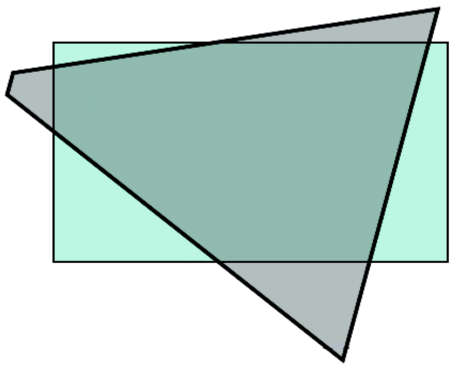


Figure 31: View Frustum Intersection
Intersection tests with the frustum are not trivial, even if all vertices of an object are outside, parts of the object might be in.

The intersection test is not as easy as it might appear (see figure). Therefor we apply a strategy similar to those from clipping:

We model the frustum as six intersecting planes (half-spaces) and use implicit plane representation. Then a point x lies in the half-space partitioned by the plane spanned by plane point p and normal $\vec{n}$, if

$$(x - p) \cdot \vec{n} > 0$$

We repeat this test for all six planes and combine the results to know whether the point lies inside in respect to all planes or not.

A modification is to test only the most critical point (i.e. the point closest) of the object against the current plane. To figure this critical point out, we comparing the object's vertices minimal and maximal in respect to a coordinate direction to the plane, to the half-space's normal. Then the object is inside if the critical point is inside, outside if the critical point as well as it's opposite corner lies outside and partly inside otherwise. Taking errors into account we increase the test efficiency by using the following rule of decision:

box within all half-spaces → render
box outside one half-spaces → cull
otherwise → render

In fact the outside test can decide an object as outside, which has parts inside. This is called a critical error.

7.7.3 Occlusion Culling

Finally occlusion culling intends to cull objects that are occluded, i.e. hid by other objects. This is important since only using the Z-Buffer several pixels will be drawn multiple times, in fact whole objects later invisible will be drawn first. While Occlusion culling intends to remove these objects, there are other methods that try to sort the objects to only render the ones in front (see Chapter about Occlusion & Visibility 7). In general aspects that have proven to be maxims for good occlusion culling are:

- **Occluder Combination**: combine several occluders to one big occluder whenever possible
- **Occluder Choice**: try to use the best occluders for checking occlusion; best occluders are: big objects and objects close to the viewer
- **Precomputation**: precompute as much as possible, but keep interactivity in mind
- **Validity Over Time**: try to keep the current occlusion computations valid for as long as possible (e.g. as long as the user is within one room or cell) and don't compute everything from scratch each frame
- **Level Of Detail**: use object models in higher resolution for objects close to the eye, and lower resolution ones for distant objects (see 10.7)
- **Hierarchy & Bounding Box**: hierarchies and bounding volumes over the scene are very useful, but it is usually hard to update them for quickly changing dynamic scenes

Potentially Visible Sets (PVS)

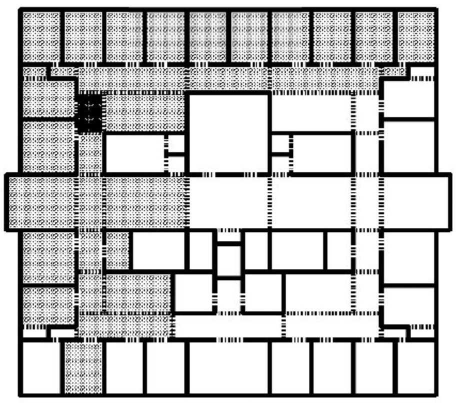


Figure 32: Potentially Visible Sets separate the scene into arbitrary cells

This first approach separates the scene into cells and computes which cells can be seen from a particular cell. This is done with precomputation. Then during rendering we check the cell, the camera is positioned in and only render the cells/sets that are potentially visible from here. Since it heavily uses precomputation, rendering time is efficient while we have a high memory load. Furthermore the computations for a cell can usually be kept for a couple of frames, as long as the viewer remains in that same cell.

The visibility of other cells can be precomputed using visibility rays (see Chapter about Ray Tracing 9).

Portal Visibility

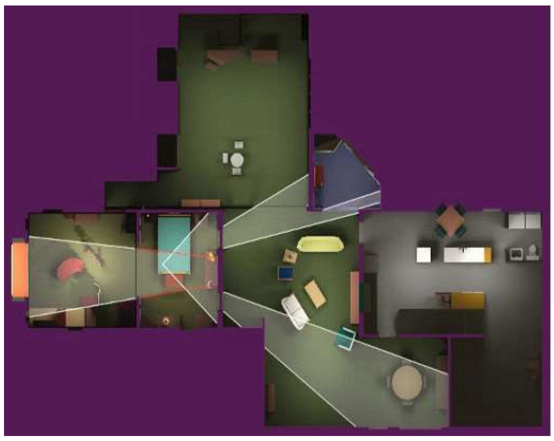


Figure 33: Portal Visibility

In the Portal Visibility approach subdivide the scene into cells that are connected by portals (e.g. doors, windows, holes). It is a very excellent method for rendering the inside of buildings. Now the algorithm goes:

- find the cell, the camera is located in
- render this cell
- for all portals in the cell: if portal is visible, render neighbouring cell recursively

To check whether a portal is visible, we can check its bounding box against the view frustum (see previous point "View Frustum Culling" above). Further more this allows for a special integration of mirrors, by seeing mirrors simple as a special kind of portal (sigh is reversed).

OpenGL Occlusion Test

OpenGL implements its own occlusion culling strategy by offering a special render mode called OccCull-mode. In this mode actually nothing is rendered, instead the number of potentially rendered pixels is counted. Now this can be used for culling like this:

- set the counter to zero
- render the object's bounding box in OccCull-mode
- render the object if the counter is above a threshold[4]

The disadvantage of this very simple test is, that it stalls the pipeline and that even counting can be expensive for thousands of triangles. This behaviour can be improved by installing several counters and parallelize the counting render mode. Then the stalling of the pipeline does only occur one time instead of one per object. This method is especially useful for very distant or complex objects (leafs on a tree).

7.7.4 Hierarchical Z-Buffer

A hierarchical z-buffer has several levels. At bottom level 0, the actual z-values are stored. One level above the highest value among a group of children z-values is stored and so on.
Now if we want to test an object for culling we project the bounding box of the object and search for the node in the Z-Tree that completely covers it and compare depth values. This leads to many fast rejects. If the object is not hidden, we proceed to its children. Passing level 0 we eventually render it.
For this to work the Z-Buffer hierarchy must be kept consistent. Using a tree-like structure this is easy: Once a value changes, this change simply propagates upwards.
This method works best when the scene is rendered from front to back. Check the chapter about Occlusion & Visibility on page 7 for methods to achieve such render orderings.

7.7.5 Occlusion Horizons

Occlusion Horizon is a specialized culling technique for urban scenes. We separate the scene's objects into: plain ground, opaque buildings, buildings on ground. Apart from that we geometrically subdivide the scene into equally spaced quads. And we reduce the dimension a little bit to $2\frac{1}{2}$ by using height fields for the height of buildings. For each building we compute a set of prisms that are completely inside (inner hull) and a set of prisms describing the bounding hull (outer hull).

Now we traverse the quadtree while moving away from the camera and process the buildings with increasing distance to the camera. We keep track of a so called **occlusion horizon** (e.g. for x, the current maximum value of x). Now for every building being processed, we check whether the building is behind the horizon ($\rightarrow$ invisible) or in front of it ($\rightarrow$ render building, adapt horizon).

[4]Objects only having a few pixels visible can usually be culled without great loss. Alternatively the object can be rendered in a highly simplified way.

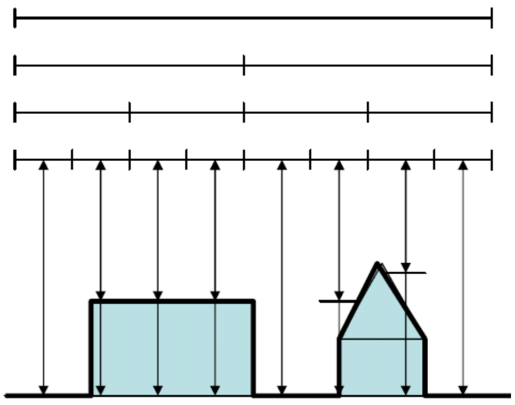


Figure 34: Occlusion Horizon

This occlusion horizon is implemented as a constant piecewise function accessed by x, stored as binary tree with values of y (see figure). Tests against the horizon are performed with the buildings outer hulls and updated of the horizon are made with the inner hulls.

Caveat

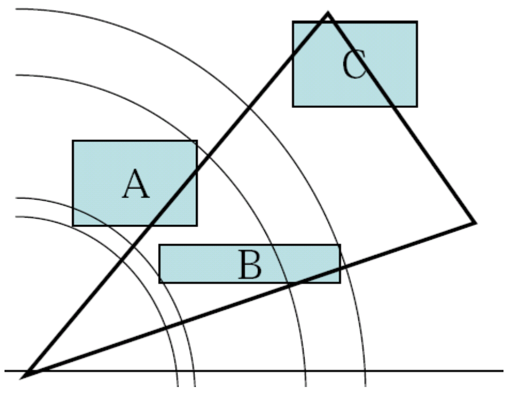


Figure 35: Occlusion Horizon: Difficult Occlusion

However it's not always that easy, as the figure above shows. Although buildings A is close to the camera, B occludes A. To avoid such scenarios we introduce a priority queue with new buildings (entered at the current horizon) sorted by maximum distance. Buildings from the queue are only then added to the horizon once their minimum distance is smaller than the maximum distance from the queue. With the queue, in the above scenario A would still be in this queue, when B is tested:

- test A - visible! - render A, put A into queue
- test B - visible! - render B, put B into queue
- A and B definitely in front of C - insert A and B into horizon
- test C...

Last but not least the idea of the horizons can be connected with Mip Mapping (see 11.5.2). Depending on the distance of the current horizon from the eye, buildings and other objects can be rendered using a lower resolution. Objects at horizons close to the eye should be rendered with an extra high resolution for details.

7.7.6 Dual Ray Space Occlusion Culling

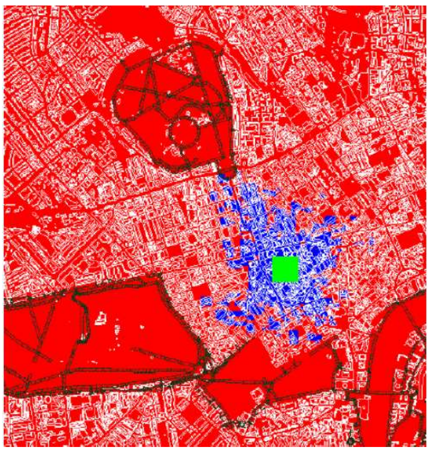


Figure 36: London
With Dual Ray Space Occlusion Culling the PVS for a single cell of size 100x100 m^2 can be rendered in 2.5 s

Imagine the map of London (see image) with an area of $160km^2$, this could be rendered by about $1,7 \cdot 10^6$ polygons. The goal Dual Ray Space Occlusion Culling is aiming at, is that a user can virtually walk through the streets, while the data is downloaded via the internet. Only what user can see is rendered, and while he walks data for buildings that soon come into vision is downloaded.

The strategy sounds familiar, we separate London into cells and determine the PVS (see above) for each cell. The currently valid PVS is kept on the client as well as the PVS of all neighbouring cells. As in occlusion horizons we restrict the dimension to pseudo 3D ($2\frac{1}{2}$D). In addition we also take occlusion horizons' quadtree idea to recursively determine the nodes of the quadtree that are visible. But how do we decide whether a quadtree node is visible from another or not? We solve this by a new idea:

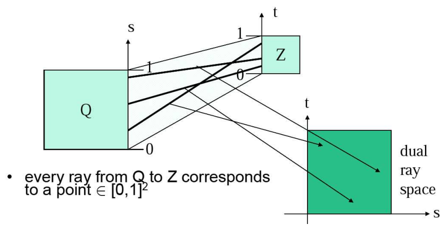


Figure 37: Dual Ray Space Occlusion Culling

First we find the two faces of Z and Q facing each other.
Then we model them as parametrized line segments: $Q_0+s\left(Q_1-Q_0\right), s \in [0;1]$, $Z_0+t\left(Z_1-Z_0\right), t \in [0;1]$
And now we transform this model to a Dual Space, where every Ray in our current space is represented by a point.

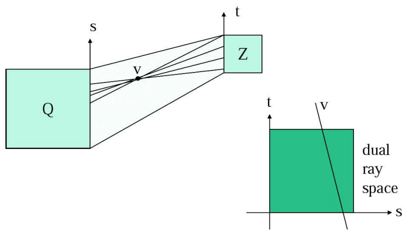


Figure 38: A line in the Dual Ray Space

For example all rays between Q and Z that hit a random point v will build a line in this Dual Space.

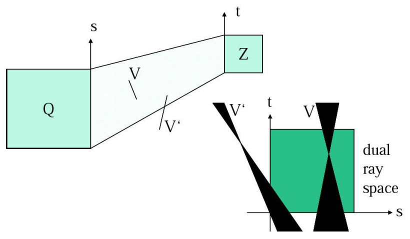


Figure 39: A double triangle in the Dual Ray Space

Now consider a whole line segment between Q and Z, all rays intersecting this segment build a double triangle in the dual space.
Slowly the idea becomes clear, we represent the space of all rays in the dual ray space as image $[0;1]^2$. Then we render this image black and compute for each segment between Q and Z the corresponding double triangle. After all segments

have been processed, we check whether there are any black pixels left, if there are not, we assume total occlusion.

8 Lighting

8.1 Light

To understand algorithms of lighting it is important to understand light, respectively the measuring of such called **radiometry** or the human perception of light referred to as **photometry**.

8.1.1 Radiometry

Radiometry is the physical measurement of light. Since light is a form of energy, we measure it in joule $[J]$.

Photons

Light can also be seen as a large amount of photons. A photon is a light quantum having a **position**, a **direction** and a **wavelength** λ given in nanometer $1nm = 10^{-9}m$. Furthermore light has **speed** c (depended on the material it passes through) and a **frequency** $f = \frac{c}{\lambda}$. Finally the amount of **energy** q carried is given by

$$q = hf = \frac{hc}{\lambda}$$

where $h = 6.63 \cdot 10^{-34}$ is **Plank's Constant**.

Energy

The energy of light in general (**radiant energy**, Strahlungsenergie) Q is then simply the sum of the single photon's energy q_i.

$$Q = \sum_{i-l}^{n} \frac{hc}{\lambda_i} [J]$$

Furthermore we can give Q relative to the wavelength λ by integrating over an interval on the wavelength:

$$Q = \frac{\Delta q}{\Delta \lambda}$$

Then we can give Q relative to time (**radiant flux**, Strahlungsleistung)

$$\Phi = \frac{dQ}{dt} \left[W = \frac{J}{s} \right]$$

Example

A light bulb with 100 Watt emitting about 5% as light, then the radiant flux Φ would be: $\Phi = 5W$

The radiant energy Q, relative to a surface (**flux density**, Flußdichte) is given by

$$\frac{d\Phi}{dA}\left[\frac{W}{m^2}\right]$$

Irradiance

Irradiance is the amount of ingoing light that hit a certain point, the **incident light** or **incident flux density** (Strahlungsenergie).

$$E = \frac{d\Phi_{\text{in}}}{dA}\left[\frac{W}{m^2}\right]$$

Radiosity

Radiosity is the light emitting from a surface. It is called **exitant light** or **exitant flux density** (spezifische Ausstrahlung) and measured the same way:

$$B = \frac{d\Phi_{\text{out}}}{dA}\left[\frac{W}{m^2}\right]$$

Example

- A big area light with $\Phi = 5W$ and $A = 1m^2$ has a radiosity of $5\frac{W}{m^2}$
- A small area light with $\Phi = 5W$ and $A = 100cm^2$ has a radiosity of $500\frac{W}{m^2}$

Radiant Intensity

Radiant Intensity (Strahlungsintensität) is used for point light sources and gives the emitted light per solid angle (see 2.11)

$$I = \frac{d\Phi}{d\omega}\left[\frac{W}{sr}\right]$$

This means, if by tilting the area is getting smaller, the same light is getting brighter (the light becomes more compact).

Approximation Usually the dome is neglected, just the area is taken into account

Note By $/r^2$ dimension and size leaves the formula. Thus dimension and size of the sphere does not influence the outcome.

Radiance

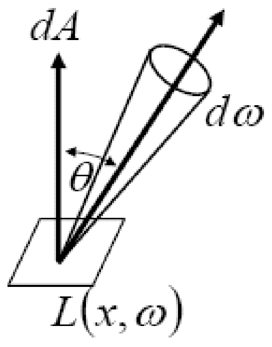


Figure 40: Radiance

Finally radiance combines flux density with radiant intensity (Strahldichte) adding a directional dependency to radiosity/irradiance.

$$L\left(x,\omega\right)=\frac{d^2\Phi}{d\omega\cdot dA\cdot\cos\theta}\left[\frac{W}{sr\cdot m^2}\right]$$

where $d^2\Phi$ is the flux[5], $d\omega$ is the differential solid angle and dA the differential of the Area ($\cos\theta$ is explained below). This results in the **brightness**. This is the value measured by cameras. Radiance is also the most important radiometric unit for computer graphics, since it is exactly what we want to store in a pixel.

Intuition

We want to measure how many photons originate from a certain point x into a direction ω. We use a light sensitive sensor as cameras use and place it above the surface. Now to get sure only the rays originating from our point we need to enclose the sensor by a black cone which has a broad opening towards x and a small towards the sensor, so that rays coming from another direction are absorbed by its black walls when bouncing.

Still some rays will get through, therefore we need to make the cone infinitesimal small ($d\omega$).

Still some rays will reach the sensor, cause it has a certain area size, therefore we need to make this area infinitesimal small (dA).

Still our sensor will be oriented in some kind, therefore we need to take this orientation into account ($\cos\theta$)

(e.g. vertical, horizontal or in between).

Now we get only the desired photons. If on the other hand we want to measure irradiance we place the sensor on the surface.

[5]d^2because we differentiate two times

Invariant Radiance is constant along a ray

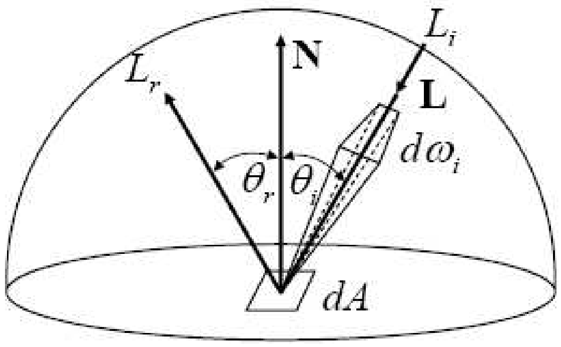


Figure 41: Relationship between incident and reflected light

An important relationship is between incident light L_i and the light reflected L_r and transmitted by the surface A

$$E_i = L_i \left(\vec{n} \cdot \vec{l}\right) d\omega_i$$

$$E_i = L_i \cos\theta_i d\omega_i$$

Overview

Table 1: Physical Light Overview

Measure	Meaning	Modeling
Flux	general light flow without differentiation	
Intensity	light per angle (e.g. the intensity of a light bulb)	power
Radiosity	light per area (all directions)	diffuse
Radiance	light per area into a direction	specular
Irradiance	incoming light per area from any direction	L_{in}

8.1.2 Photometry

Where radiometry covers the physical measurement of light, photometry covers the human measurement of light. The human system is only capable of perceiving a limited range of radiation. Furthermore the human response system is not linear: Some wavelengths appear brighter than others (e.g. red).

The average human vision capabilities (daytime) are covered in lumen $V(\lambda)\left[\frac{lm}{W}\right]$. L_v covers how bright a certain wavelength is perceived.

$$L_v = k_m \int L(\lambda)\, V(\lambda)\, d\lambda$$

B_v, Q_v, Φ_v are expressed correspondingly.

8.2 Lighting

Lighting is the simulation of physical light to make a 3D scene look real. However a real approximation takes far too long, so that we make a lot of approximations, simplifications and hacks. Our task is to compute the luminous intensity at a point in the scene.

8.2.1 Simplifications

Since we are far from able to model light physically correct, we often are forced to make some of these common simplifications:

- no interactions between wavelengths (e.g. fluorescence)

Figure 42: fluorescence

- time invariance (distribution remains constant over time, e.g. phosphorescence)

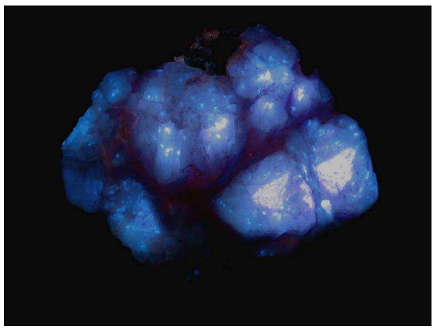


Figure 43: phosphorescence

- light transport in vacuum (no intermediary medium, emission and absorption just on objects, e.g. smoke, mist)
- isotropic objects (identical material)
- direct illumination (no or limited reflected illumination)

Light Hitting A Surface can be

- absorbed
- scattered
- reflected
- refracted
- transmitted

8.3 Illumination

Transport of energy (in particular, the luminous flux of visible light) from light sources to surfaces & surface to eye. There are two major components of illumination:

- light source

  a light source has a certain **spectrum** (color), a **direction** and a **shape** (e.g. point light source, parallel light, area light source).

- surface properties

  a surface has a **reflectance spectrum** (color), a **position**, an **orientation** (given by a **surface normal** at every point) and a **micro structur**e (important for scattering and reflection)

Local Illumination

Illumination by one or several light sources (point, parallel). This results in having no shadows. An example is Phong Lighting.

Global Illumination

Global light exchange (area light sources). Slower but with shadows and higher quality. An example is Ray Tracing

8.3.1 Light Sources

Point Light Source

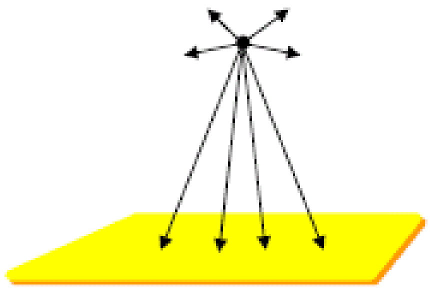


Figure 44: Point Light Source

Light is equally emitted in all direction originating from a single point. Thus the light direction towards a surface varies for every surface point. Thus we have to compute a normalized light direction vector for every point:

$$l = \frac{p - x}{\|p - x\|}$$

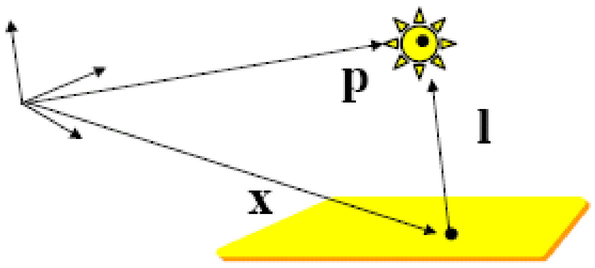


Figure 45: direction of light for point light source

Directional / Parallel Light

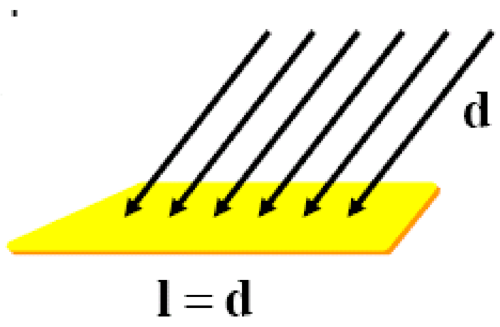


Figure 46: Parallel Light

Light is modeled by parallel rays originating from a quasi infinite distant light source (e.g. the sun). The direction of the surface relative to the light direction becomes important.

Spotlight

A point light source with parallel light. Outside the spotlight, the light remains parallel but fades away in intensity, limiting the light to a certain area. A mixture of the two above.

Area Light Source

Defined by a 2D emissive surface (e.g. a flashlight). Area light sources are capable of creating soft shadows.

8.3.2 Phong Lighting Model

Properties

- local illumination
- heuristic, no physical simulation
- fast

Variables

Reflected Light

In Phong Lighting, reflected light does not exist per se and is therefore approximated by three components:

Ambient Light Indirect light modeled by a constant (omnipresent light)

Variable	meaning
$\vec{l}$	direction of light
I	light Intensity
$\vec{v}$	vector towards the eye
$\vec{n}$	surface normal
k	surface constant (color)
n_{shiny}	empirical constant (spread of the highlight)
$\vec{r}$	ideal reflectance vector
$\vec{h}$	halfway vector

Table 2: Phong Lighting Variables

Diffuse Light reflection from rough surfaces (uniform into all direction)

Specular Light reflection from glossy (no perfect mirrors) surfaces

$$L_{\text{total}} = L_{\text{ambiebt}} + L_{\text{diffuse}} + L_{\text{specular}}$$

Ambient Light

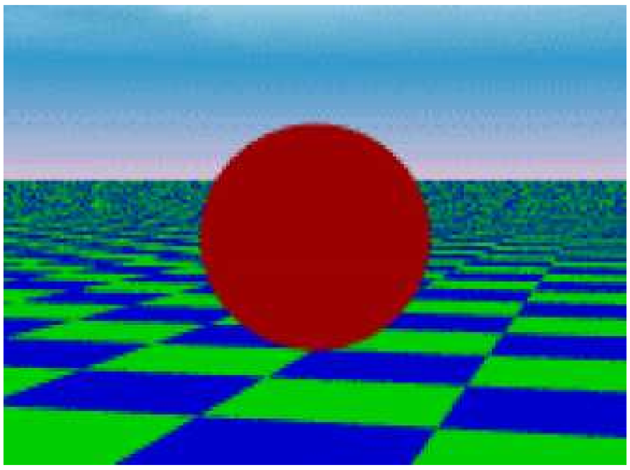


Figure 47: Ambient Light

Covers objects that are not directly light, but that would be still visible by indirect illumination.

$$L_{\text{amb}} = k_{\text{amb}} I_{\text{amb}}$$

Some properties of ambient light:

- no physical base (necessary because indirect/global illumination is skipped)

- better results by giving ambient light per light source, so that if one light source is turned off, it's ambient light is removed from the objects
- if a surface is not covered by any light source, only ambient light is applied giving the surface an uniform look and no 3D features (there often an additional light source called **head light** is used above the viewer to make sure everything visible is at least lit by some direct light source.

Diffuse Light

In principle the scattering/reflection of light depends on the surface's micro structure. In Phong Lighting we assume rough surfaces to be equally rough scattering light equally in all directions.

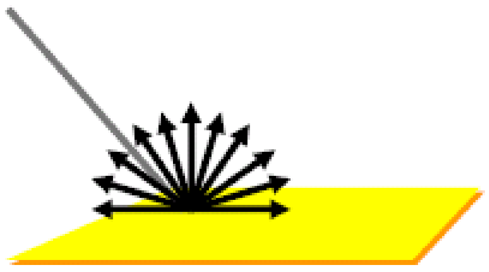


Figure 48: Reflection from equally rough surfaces

For this case we may apply **Lambert's Cosine Law**:

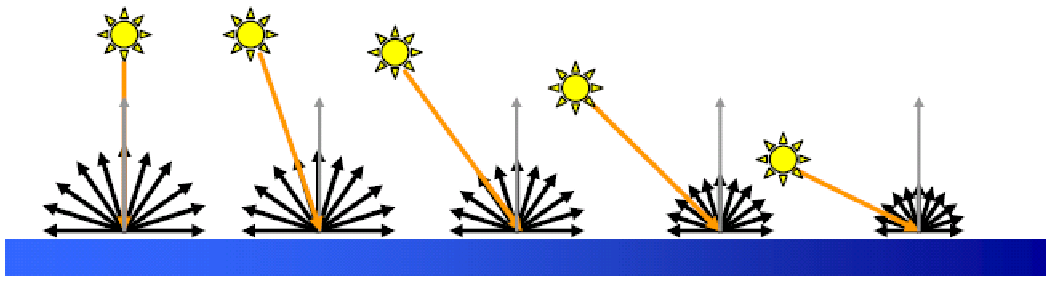


Figure 49: Lambert's Cosine Law
Reflected radiant intensity in any direction varies as the cosine of the angle between light direction and surface normal.

Therefore we require the angle between the light direction $\vec{l}$ and the surface normal $\vec{n}$ to compute diffuse light:

$$L_{\text{diff}} = k_{\text{diff}} I_{\text{In}} \cos\theta$$

$$L_{\text{diff}} = k_{\text{diff}} I_{\text{In}} \left(\vec{n} \cdot \vec{l}\right)$$

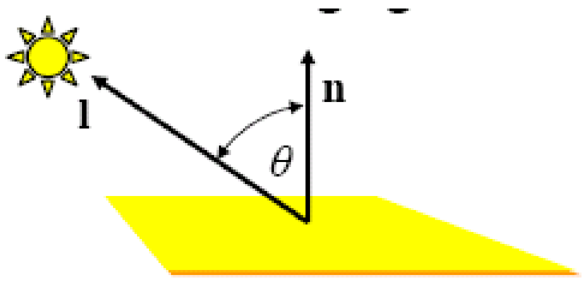


Figure 50: Diffuse Light

As you can see the view direction does not appear in the formula. This means the diffuse light is view independent and thus looks the same from any direction. The angle θ gives us more information. If

- $\left(\vec{n} \cdot \vec{l}\right) < 0$: light is below the surface
- $(\vec{n} \cdot \vec{v}) > 0$: eye is below the surface

In both cases we clamp I to zero.
Some properties of diffuse light:

- diffuse light is view independent
- diffuse light is based on Lambert's cosine law and with that based on real world physics

Specular Light

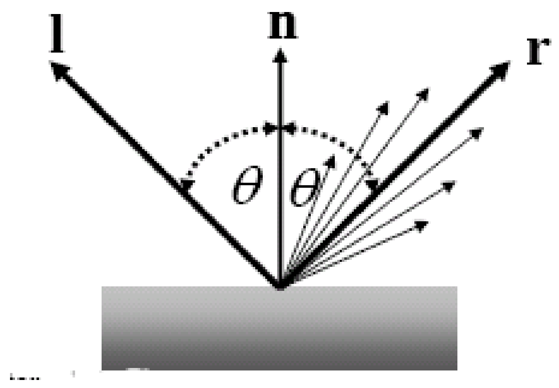


Figure 51: Specular Light

Reflection for glossy materials, e.g. polished metal. Light causes a bright spot on this surface (**specular highlight**). This highlight depends on the direction the viewer looks at the surface, thus it is view dependent. Specular light approaches mirror like reflectance (see Figure).

$$L_{\text{spec}} = k_{\text{spec}} I_{\text{In}} \cos{(\phi)}^{n_{\text{shiny}}}$$

$$L_{\text{spec}} = k_{\text{spec}} I_{\text{In}} \left(\vec{v} \cdot \vec{r}\right)^{n_{\text{shiny}}}$$

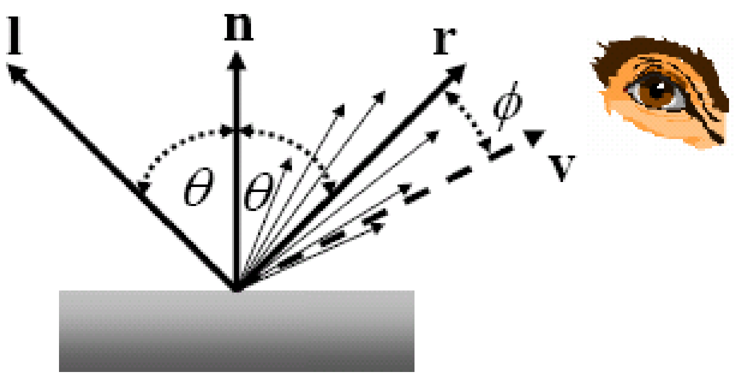


Figure 52: Specular Light is view dependent

where n_{shiny} determines the spread of the highlight. A large n_{shiny} makes for a rather glossy (narrow) highlight, and a small n_{shiny} a rather diffuse one. The vector $\vec{r}$ of ideal reflectance can be computed as

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

Some properties of specular light:

- view dependent (the highlight moves with the viewer)
- halfway vector reduces computation time
- no physical base or validity
- looks unrealistic with per-pixel shading (Phong Shading, see below 8.4.3)

Complete Reflected Light

$$L_{\text{reflected}} = k_{\text{amb}} I_{\text{amb}} + \sum_{i}^{\#\text{lights}} I_{\text{in}_i}\left(k_{\text{diff}}\left(\vec{n}\cdot\vec{l}_i\right) + k_{\text{spec}}\left(\vec{v}\cdot\vec{r}_i\right)^{n_{\text{shiny}}}\right)$$

Halfway Vector Approach

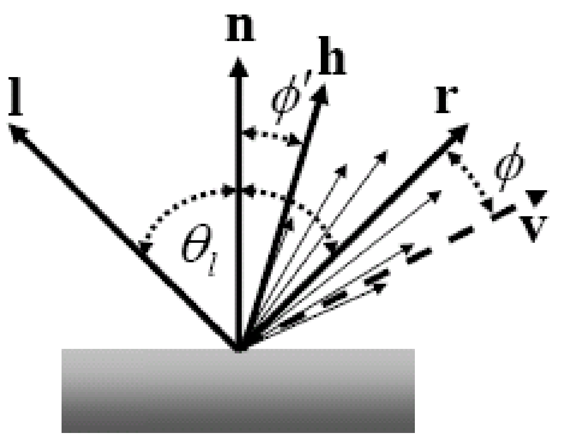


Figure 53: Halfway Vector Approach

Computing the ideal reflectance vector $\vec{r}$ is costly. The Halfway Vector Approach comes close to the results of using $\vec{r}$ but is far more efficient. The idea is that this halfway vector $\vec{h}$ is exactly equal to $\vec{n}$ if the view direction $\vec{v}$ is parallel to the reflection direction $\vec{r}$. When it deviates from $\vec{n}$, the angle of deviation is $\phi' = \frac{\phi}{2}$. Therefore the inner product $\left(\vec{n}\cdot\vec{h}\right)$ equals $\cos\left(\frac{\phi}{2}\right)$. The halfway vector specular formula is:

$$\vec{h} = \frac{\vec{l}+\vec{v}}{\left\|\vec{l}+\vec{v}\right\|}$$

Using $\vec{h}$ draws

$$L_{\text{spec}} = k_{\text{spec}} I_{\text{In}} \cos(\phi')^{n_{\text{shiny}}}$$

$$L_{\text{spec}} = k_{\text{spec}} I_{\text{In}} \left(\vec{n} \cdot \vec{h}\right)^{n_{\text{shiny}}}$$

If the light is directional and the view parallel (orthographic), both $\vec{l}$ and $\vec{v}$ become constant, resulting in a constant $\vec{h}$.

Attenuation

For parallel light there is no attenuation, since we have parallel light rays everywhere, therefore we consider a point lightsource at position p, which is d away from the surface point x it is lighting ($d = \|p - x\|^2$).

$$I_{\text{In}} = \frac{I}{\|p - x\|^2}$$

But this approach is problematic for close and distant light sources. Therefore we add some constants and build a polynomial:

$$I_{\text{In}} = \frac{I}{c_0 + c_1 d + c_2 d^2}$$

where c_i model for example atmospheric attenuation, smoke or vacuum.

Note that a physically correct attenuation proceeds quadratic in respect to the distance to the light source. However we usually do not model this because a) the sun is too far away (hardware float precision), b) it results in very unrealistic looking light and c) we can model additional atmospheric attenuation like smoke.

Material Colors

The material constants k_i may consist of two parts:
a brightness value $k \in [0, 1]$
a color frequency $O_\lambda, \lambda \in RGB\alpha$
This allows the mixing of lightsource and material color frequency

8.3.3 Torrance-Sparrow Light Model

The Torrance-Sparrow Light Model tries to model physics. Here we do not assume equally rough surfaces, but consider isotropic collections of planar microscopic facets.

$$L = I_{\text{In}} \cdot \frac{F\left(\vec{l} \cdot \vec{h}\right) G\left(\vec{n} \cdot \vec{v}, \vec{n} \cdot \vec{l}\right) D\left(\vec{n} \cdot \vec{h}\right)}{\pi\left(\vec{n} \cdot \vec{v}\right)\left(\vec{n} \cdot \vec{l}\right)}$$

Variable	Meaning
$\vec{n}$	standard surface normal
$\vec{h}$	normal for rough surfaces (current normal wandering across the hill)
$\vec{v}$	view vector
$\vec{l}$	light vector
π	accounts for surface roughness
$D\left(\vec{n}\cdot\vec{h}\right)$	distribution of micro facets / normals (Gaussian)
$G\left(\vec{n}\cdot\vec{v},\vec{n}\cdot\vec{l}\right)$	attenuation, masking and self shadowing
$F\left(\vec{l}\cdot\vec{h}\right)$	Fresnel Term
$(\vec{n}\cdot\vec{v})\left(\vec{n}\cdot\vec{l}\right)$	~maybe~ for the specular highlight?

The **Fresnel Term** describes the relation between incoming and reflected light and takes surface properties (glass, water) into account.

Self shadowing means that reflected light bounces against a facet:

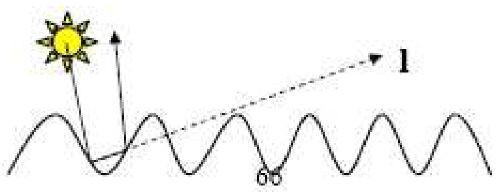


Figure 54: Self-Shadowing

Schlick gives an fast and efficient approximation of the complex Fresnel Term:

$$F = f_\lambda + (1 - f_\lambda)\left(1 - \vec{v}\cdot\vec{h}\right)$$

where f_λ is the Fresnel reflectance of the material at normal incidence.

8.4 Shading

Shading is a kind of parent to lighting, it decides for which pixels lighting is computed and how these values are interpolated across a face.

8.4.1 Flat Shading

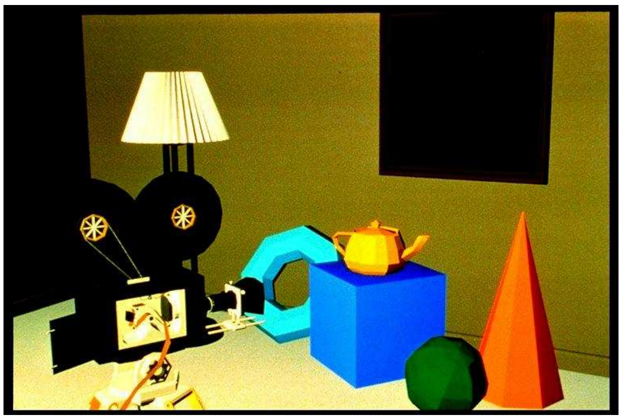


Figure 55: Flat Shading

The polygon is partitioned into faces. Each face has a uniform surface normal. Therefore we compute lighting for a single point on the face, and take it for the rest.

- inaccurate for faceted objects

`OpenGL glShadeModel(GL_FLAT)`

8.4.2 Gouraud Shading

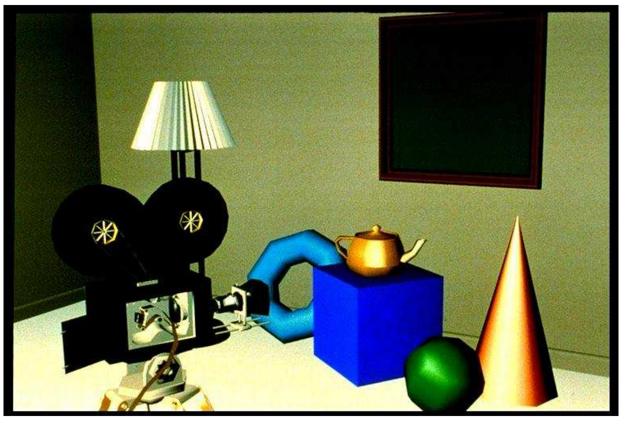


Figure 56: Gouraud Shading

Instead of applying Phong Lighting with surface normals for complete faces, we apply it with vertex normals. Each vertex of a polygon is assigned a normal, which is an average by the surface normals of surfaces contributing to this vertex (wireframe).

$$\vec{n}_v = \frac{\sum_i^{\#\text{faces}} \vec{n}_i}{\left|\sum_i^{\#\text{faces}} \vec{n}_i\right|}$$

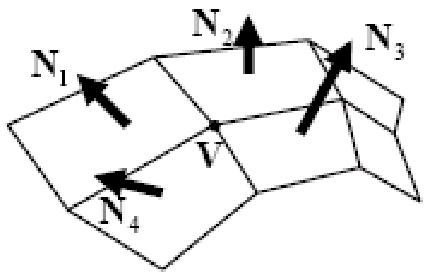
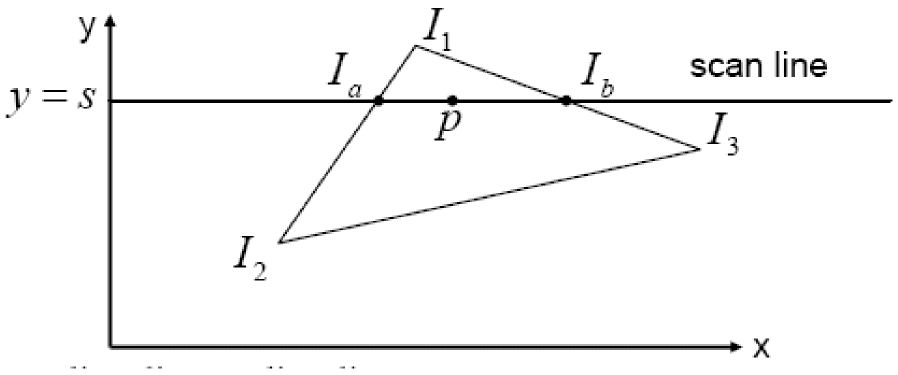


Figure 57: vertex normals

The next step is then to interpolate the vertex values across the faces.

`OpenGL glShadeModel(GL_SMOOTH)`

Algorithm 6 Gouraud Shading



1. apply Phong Lighting Model to vertices I_1, I_2, I_3
2. interpolate these values along the edges $\rightarrow I_a, I_b$

$$I_a = \frac{y_s - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_s}{y_1 - y_2} I_2$$

$$I_b = \frac{y_s - y_3}{y_1 - y_3} I_1 + \frac{y_1 - y_s}{y_1 - y_3} I_2$$

3. use scanline algorithm to interpolate between the edges $\rightarrow I_p$

$$I_p = \frac{x_b - x_p}{x_b - x_a} I_a + \frac{x_p - x_a}{x_b - x_a} I_b$$

As with the scanline algorithm a incremental update can be found to makes things faster/numerically more stable

- fails to capture spotlight effects
- Through the interpolations highlights are smeared and light decreases slower

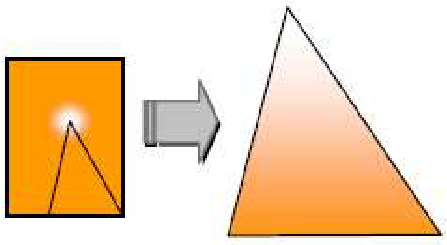


Figure 58: Gouraud Shading smears highlights

Gouraud Shading can be artificially modified to perform Phong Shading.
This is done by making the surfaces (triangles) smaller than pixels, so that effectively shading per pixel is performed.
This is done because many graphics hardware support Gouraud Shading, but not Phong Shading.

8.4.3 Phong Shading

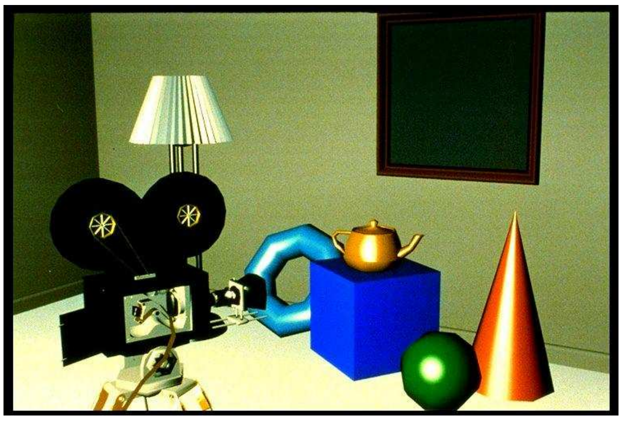


Figure 59: Phong Shading

1. compute **vertex normals** at each polygon vertex
2. interpolate these **normals** across the face
3. recompute lighting for each pixel with the **interpolated normal**

The interpolation of the normals works just as the interpolation of light in Gouraud Shading.

+ looks really good

+ good highlights → implement a highlight test ($\left(\vec{n} \cdot \vec{h}\right) \geq \tau$, Threshold τ) and use Phong Shading only for faces with highlights

+ correct size

- high costs

- three vector components

- constant renormalization necessary (square root) → interpolate scalar products instead, saves renormalization

- huge amount of lighting calculations

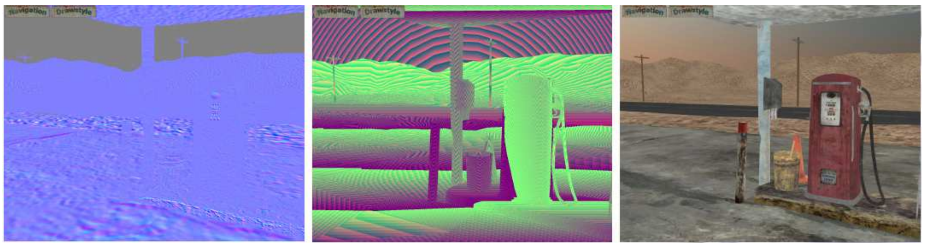
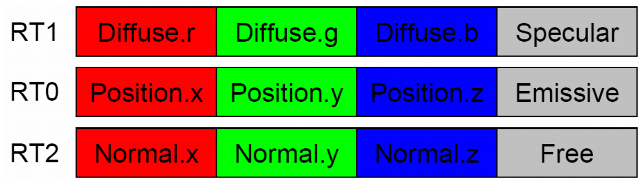


Figure 60: Deferred Shading
We store all parameters important for shading in RGBα Render Targets (textures)



RT1	Diffuse.r	Diffuse.g	Diffuse.b	Specular
RT0	Position.x	Position.y	Position.z	Emissive
RT2	Normal.x	Normal.y	Normal.z	Free

Figure 61: Storing shading parameters in the RGBα channels of three Render Targets
Note that 16bit are overkill for diffuse reflectance.

8.4.4 Deferred Shading

The idea of deferred shading is postpone shading as far behind as possible. We do this by saving all we need for shading during the modeling stages: pixel position, normals, light/color: diffuse and specular albedo, material. To effectively store all these values, we can use Multiple Render Targets or Multiple Texturing (see 11.15) and make use of the individual RGBα channels (see figure). This patch of memory is then commonly referred to as **G-Buffer**.

Having all parameters, that we need, we can reduce the complexity significantly and result in the following procedure:

+ worst case: $O\,(\text{objects} + \text{light sources})$ (other shading techniques have: $O\,(\text{objects} \cdot \text{light sources})$)

+ works best for depth complex scenes with multiple light sources

Algorithm 7 Deferred Shading

```
For each object:
→render lighting properties to the G-Buffer.
For each light:
→ framebuffer + BRDF(G-Buffer, light)
```

+ models many small light sources just as fast as one big one

+ allows for the integration of all popular shadow methods

8.5 Shadows

Computing shadows is not very easy, since the entire scene has to be known to decide whether a point lies in shadow (does the light hit the point, is the point occluded by another object, is it self-occluding). However the pipeline is a sequential process where one triangle is rendered after the other, but each new triangle could cast shadows to the previous ones.

8.5.1 Planar Shadows

Another idea is to generate a 2D projection of an object onto a plane:

1. render ground
2. render object
3. set matrices to the desired projection
4. render the shadow in black

Problems

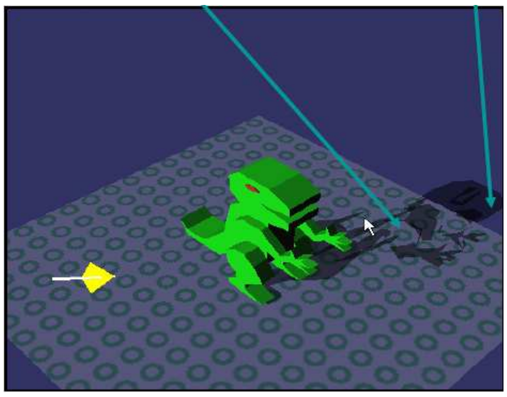


Figure 62: Problems occurring with planar shadows

- shadow outside of polygon ground
- Z-Buffer fighting (because the shadow is so fine)

These can be solved by using the stencil buffer:

1. render the object

2. render the ground and set stencil buffer to 1 for ground pixels
3. turn off Z-Test
4. render shadow where stencil buffer is equal to 1

Properties

+ fast

+ simple

- only for shadows on planar object

- no self shadowing

8.5.2 Light Maps

One idea arising from this problem is to precompute shadows by **light maps**. Light maps are textures that store the light conditions of a static scene in an image. Often light maps are called static shadow maps.

8.5.3 Shadow Maps

Shadow Maps are more general than planar shadow projection and allow for the casting of curved shadows on curved surfaces, however this technique requires two rendering passes: One from the "view" of the light source and one from the camera.

Light View everything that is visible from here, must be lit. All hidden parts are in shadow. Since we are only interested in how deep the objects are located, we only save the depth values (Z-Buffer).

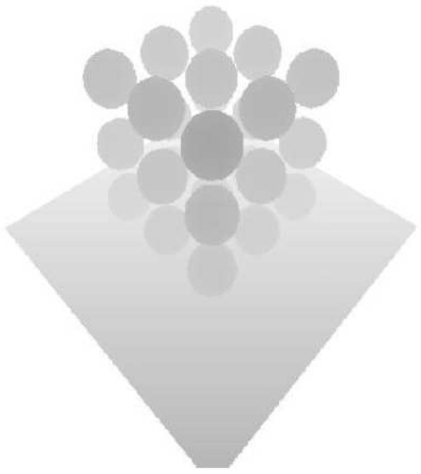


Figure 63: The scene as seen from the light. Only the depth values are stored.

Render Pass now we transform our (x, y, z) coordinates to access the shadow map (x', y', z') (see texture mapping 11 for details). Then we check whether:

$z' = \text{shadow}\,(x', y') \rightarrow$ pixel is lit

$z' > \text{shadow}\,(x', y') \rightarrow$ pixel is in shadow (render it black)

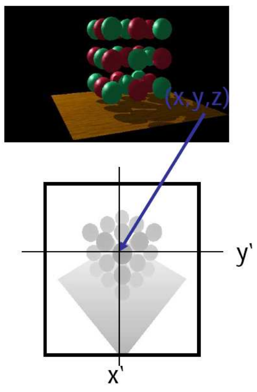


Figure 64: In the second pass, the shadow map is accessed to determine whether a pixel is lit or in shadow.

Dimmed Shadows

Even nicer results can be obtained, when the shadow map returns grey values instead of white and black. Then the shadows are dimmed and not entirely black.

Colored Shadows

But still the shadows will not contain any of the materials property color. If we want the shadows to return even a full spectrum of color, taking into account material colors we just have to render the scene before the first pass with only ambient light turned on. However this results in having three rendering passes.

Curvature Shadows

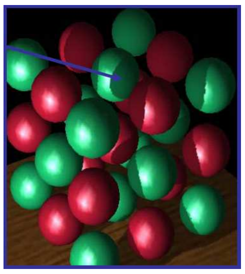


Figure 65: The blue arrow shows where the curvature can be seen in the shadow

We can even make curvatures visible in shadows by computing diffuse reflection, although we are in shadow. We simply darken the ("not" incoming) light by some factor and apply Lambert's cosine law. To make the curvature look good we can use a so called fragment shader.

Spotlight Shadows

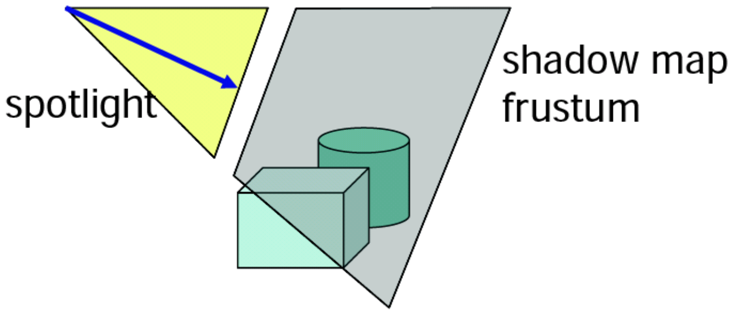


Figure 66: Spotlight shadows can be created by using the camera's frustum as a shadow frustum

If we are dealing with spotlights the shadows will be limited to the light spread of the spotlight. We can obtain this effect by using the camera's view frustum as a shadow frustum when creating the shadow map. The camera's frustum parameters are adjusted to fit the spotlight (view direction is spot direction, spot angle is fovy).

Directed Light Shadows

In case of a parallel/directed light source, we use the planar shadow method from above if possible (surface for the shadow required). If not we have to use an orthographic projection rather than a perspective one to render the lightview.

Omnidirectional Light

In case the point light is outside of the scene (what we have assumed), we can use the standard methods presented. If however the point light is within the scene we speak of an omnidirectional light and need a variation of the method. A solution is to use a cube environment map for creating six shadow maps (or a parabolic for two) and use the reference techniques of environment texture mapping (see 11.10).

Properties

static one render pass, only one shadow map

dynamic two/three render passes, shadow map generated per frame

- machine precision allows only for the test $z' \approx \text{shadow}(x', y')$, but not $z' = \text{shadow}(x', y')$ (render with a small depth offset, `glPolygonOffset`)

  alternatively to real depth values polygons or even whole objects can be assigned an ID. During rendering this ID is compared, and if it matches the polygon/object is lit.

- aliasing: the resolution of the saved shadow map easily becomes visible (worst for light from opposite direction: the shadow will be projected right to the near plane and be huge, best for a miner's lamp above the object: straight projection without perspective distortion.)

- the light is assumed to be outside the scene. If it isn't the "light view" trick won't work.

  for this case environment maps could be used (e.g. a resp. six cube maps)

Aliasing can be met with antialiasing techniques such as linear interpolating the shadow map (see 3.8). However because depth values may have hard jumps, **percentage closest filtering** is suggested: We compare the current pixel's depth value with the surrounding ones and only take those for bilinear interpolation that's depth value equals at least X percent of the pixel's depth value.

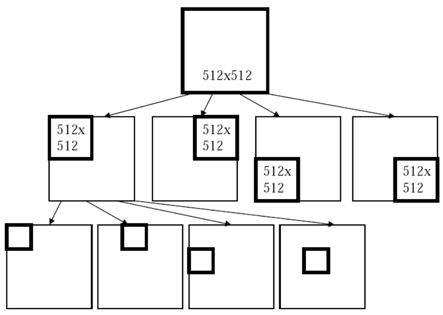


Figure 67: Adaptive shadow maps are ordered and accessed in a tree structure

Another idea are **Adaptive Shadow Maps** where similar to Mip-Mapping (see 11.5.2) shadow maps are stored in different sizes (here the resolution remains the same, instead the size varies depending on the position on the screen, see figure). Then depending on the screen section the appropriate is chosen. The shadow maps are ordered and accessed in a quadtree structure (see figure). To choose which shadow map is appropriate, Mip-Mapping hardware supported technique can be exploited.

+ very good shadows

- many passes until the quadtree has been cut

- difficult if the camera turns around

- not yet applicable in dynamic scenes

A third method to meet shadow map aliasing especially for large scenes are **Light Space Perspective Shadow Maps**. The idea is to use a projective mapping to shift resolution to regions close to the camera. A suggestion for where to place the eye for this projective projection p_{opt} (this is the degree of freedom) is

$$p_{opt} = n + (n \cdot f)^{\frac{1}{2}}$$

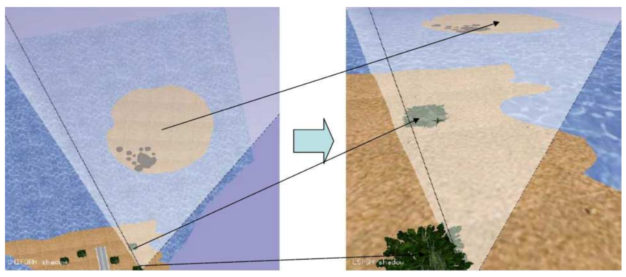


Figure 68: The shadow map resolution changes with a projective mapping

+ only one shadow map

8.5.4 Soft Shadows

Maybe the biggest problem dealing with shadows is the creation of soft shadows. Soft shadows mean shadows respecting attenuation (by distance to the occluder) and a blurred border line. Alas simple blurring of the border line by low-pass filtering does not in the results we want. If computation time is unimportant we can simulate an area light source by several point light sources (at least 64 necessary to avoid artifacts) and blend between them.

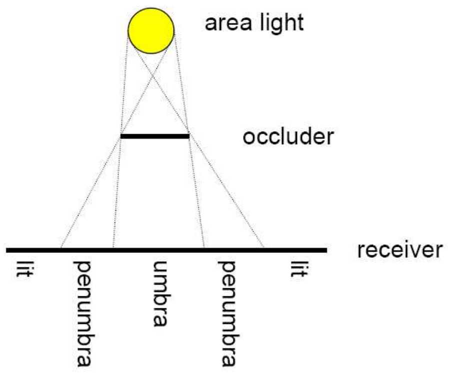


Figure 69: Partitioning of the scene into umbra, penumbra and lit areas.

Modeling real area light sources we come up with a model the separates the scene into three regions: umbra, penumbra and lit (see figure).

Penumbra Maps

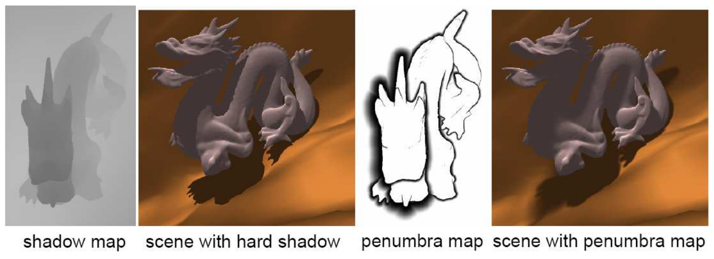
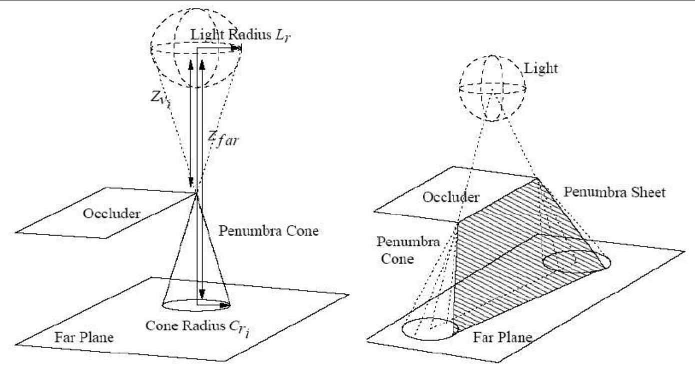


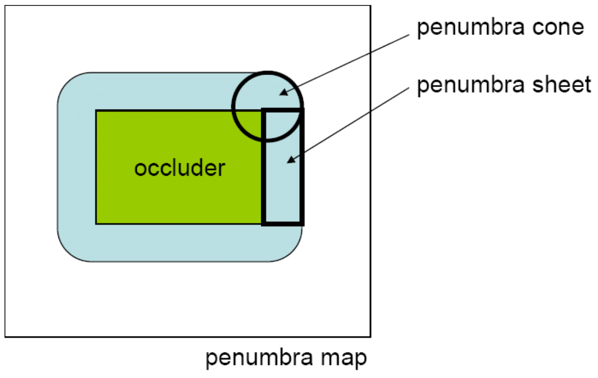
Figure 70: Shadow mapping combined with a penumbra map to soften the shadow outlines

Penumbra Maps are an extension to Shadow Maps respecting soft shadows. We simplify the model by modeling an arbitrary area light source with a "disc/sphere" light source modeled again by a point light source and a radius. Then we add a second shadow map, which we call **penumbra map**. This image contains penumbra values between $[0;1]$. This value is used to modulate the already computed hard shadow turning it soft (see image). The question remains how to compute this penumbra map:

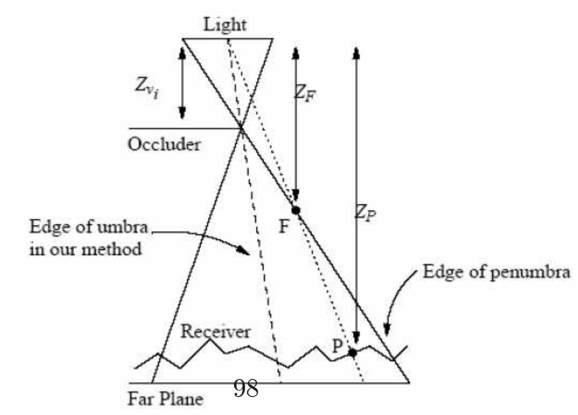
Algorithm 8 Creating Penumbra Maps



1. compute occluder's silhouettes (see shadow volumes 8.5.4)
2. generate penumbra cones (silhouette vertices) and sheets (silhouette edges)
3. render penumbra cones and sheet in light view
4. take the blue hulls/outlines for penumbra regions (use the Z-Buffer to obtain blue region)



compute penumbra value for each pixel



$$pen = \frac{(z_F - z_{vi})}{(z_P - z_{vi})}$$

where z_{vi} is the distance of the occluder to the center of the light source, z_F the distance of the fragment and z_P the distance of the receiver (taken from shadow map) . z_F lies on the intersection of the penumbra edge with the line from the pixel $z_p$to the light source's center and determines the penumbra value. Now if:

$z_F = z_{vi}$ the penumbra value results in $pen = \frac{0}{z_p - z_{vi}} = 0$, meaning black a total shadow

+ can be done by a fragment shader

- because of modeling any area lightsource as disc, we get rectangular light sources, which prefer one direction, wrong (shadow is harder for one direction as for the other)

- having a disc or sphere the proportions of the penumbra value determination methods get wrong, because e.g. 40% of the diameter does not correspond to 40% of the sphere's volume (disc's area), because it gets broader in the middle. We can account for this by using this transformation:

$$pen' = 3pen^2 - 2pen^3$$

- overlapping: how to deal with overlapping penumbras (many different scenarios, see figure)

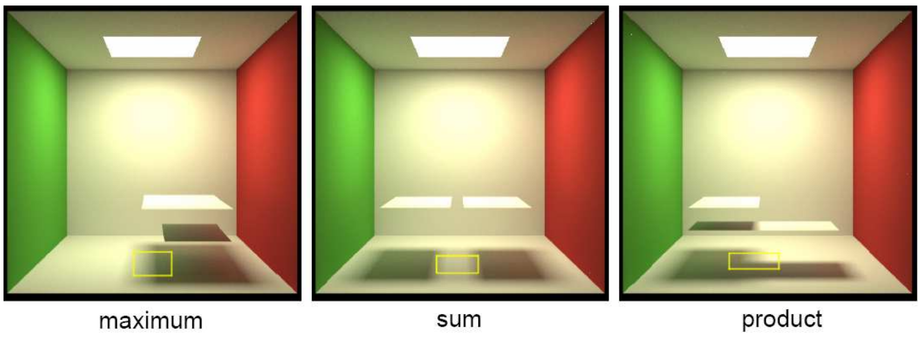


Figure 71: Overlapping Penumbras

As you can see there are several different cases, that would need to be differentiated, when penumbras overlap

Shadow Volumes

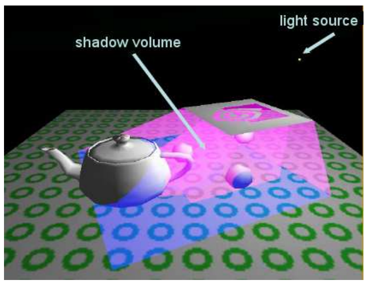


Figure 72: Highlighted Shadow Volume

Shadow Volumes describe the boundary surfaces between lit and shadowed regions. Their application is restricted to watertight, convex surfaces[6]. Imagine connecting all vertices of a triangle with the light source (point) and extending these lines into infinity. Above the triangle we will have a pyramid and below a truncated pyramid. Everything that lies within this truncated pyramid is within the shadow of the triangle. This truncated pyramid is what we call a shadow volume.

Algorithm 9 Shadow Volume Algorithm

```
1. render the scene without shadows
2. generate shadow volume
3. determine for every pixel, whether it is within the volume
4. dim pixels within the shadow volume
```

Construction: Triangle

take the triangle and shadow surfaces of the edges connected to the light source's center.

Construction: Triangle Mesh

take the triangles facing the light source, and shadow the surfaces of the object's silhouette. Use the scalar product between surface normal and light direction to determine the light facing triangles:

triangles where $\left(\vec{n} \cdot \vec{l}\right) > 0$ are facing the light source

triangles where $\left(\vec{n} \cdot \vec{l}\right) < 0$ aren't

If with two triangles A and B, A is facing the light source and B ain't, then we add the edge between A and B to the silhouette of the object.

[6]We can break these restrictions by elongating the surface bottom wards or copying it and sticking it to the backside. Because the problem arises with surfaces having no thickness and normals only for one side. Also take special care with the borderline.

Algorithm 10 Constructing The Shadow Volume

```
1. determine $\left(\vec{n} \cdot \vec{l}\right)$ for all triangles and store it as signum $+/-$
2. for each neighbouring triangle (use face list for topology information, see 10.1.5):
   if the signi of both triangles differ
      (a) extrude the common edge $[p, q]$ away from the light source towards $\infty$
      (b) add the resulting surface to the shadow volume
```

Multiple Occluders

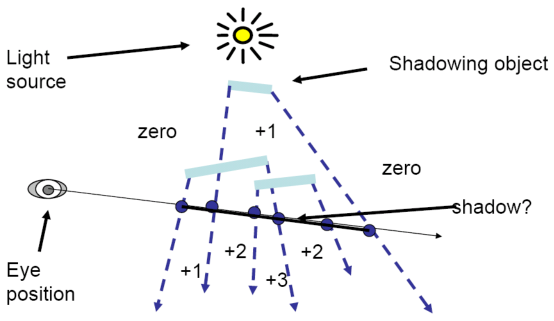


Figure 73: Dealing with multiple occluders

Having multiple occluders we build shadow volumes for each of them. But then the problem arises that we can enter and leave shadow volumes. One possibility is to set up a counter, following the eye incrementing when entering and decrementing when leaving a shadow volume. When the counter is greater than zero, we are within shadow. However this is not very efficient (e.g. because of intersection computations).
Another idea is to make use of the stencil buffer:

Algorithm 12 Shadow Volumes with Vertex Shaders

Two passes, for front and back facing triangles.

```
1. send all edges to the vertex shader as degenerated quad literals
2. check the edges for being silhouette edges (see above)
    (a) if they are: two of their edges are projected away from the lightsource
    (b) if they aren't: render them as degenerate polygons covering no pixels
3. all the transformed quad literals now define the shadow volume sides
```

Algorithm 11 Z-Pass Algorithm

```
1. render the scene with Z-Buffer turned on
2. lock writing to Z-Buffer and framebuffer (but leave the Z-Test enabled)
3. render shadow volumes facing the viewer: for every rendered pixel, increment the stencil buffer on this position
4. render shadow volumes back facing the viewer: for every rendered pixel, decrement the stencil buffer on this position
```

Now every pixel where the stencil buffer is greater then zero is in shadow.
Step 3 and 4 can be rendered in one pass, if the shadow columns do not overlap.

Properties

infinite shadow volumes use infinity point with $w = 0$

+ only one pass for rendering shadow volumes (front/back facing is supported by `OpenGL`)

+ optimal quality

+ less silhouette edges than vertices

+ no sampling problems (does not use texture maps)

- restricted to watertight convex surfaces

- limited depth of the stencil buffer (8 bits, max counter 255)

  simply use another buffer, e.g. color or α-buffer

- determining the silhouette in software is very expensive

- rendered shadow volumes are very large (high fill rate necessary), especially close to the light source, rasterizer becomes a bottleneck

- viewer in shadow: counter values are wrong (determine alternate global counter start value, place the viewer far away from the near plane)
- too close near plane: the near plane might be beyond the entry point of the first shadow volume and remove it (different counter starting values)

  There is an alternative algorithm called **z-fail**, which reverses the order of the z-pass algorithm, but z-fail encounters the same problem with the far-plane (we start at the far plane and increase/decrease the stencil, once a depth test fails). However if we move the far-plane to ∞ the problem is solved. Note that because of matrix calculations the other way round, i.e. setting the near-plane to 0 is not possible (choosing even $-\infty$ objects will be perspectively stretched to infinity).

  Disadvantages of this method is hardware dependency and that in general the z-pass method will fill less pixel overall and thus be faster.
- disadvantage of the z-fail method is, that a lot more pixels have to be rendered than using z-pass. Because usually the scene goes on quite a lot after the view frustum far-plane.

In general it is important to find out where the bottleneck is located. For example using simpler models for the occluders won't help when using the shadow volume approach, since its bottleneck is usually the fill-rate. Once the bottleneck is spotted a common approach is to use several simplifications (e.g. simpler model, lower refresh rate for shadow information) and take wrong, inaccurate shadows into account to get a acceptable frame rate (the idea is that it is hard to determine the correctness of a shadow by the eye alone anyway)

8.6 Motion Blur

Motion Blur is a feature added to moving objects to support the illusion of movement. A simple idea is, taking for example a sword slicing through the air, to add copies of polygon at previous positions setting their α-value for blending with the background. For this effect we need to gather a series of images. For this purpose a special extra big buffer, called **accumulation buffer** has been set up (see also 0.1).

To make this process fit for real time, we do not gather n images before displaying them, but use image operations. If we rendered the $n + 1^{th}$ image, we subtract the image $n - x$ from the buffer and add the new image $n + 1$. By this we only need two renderings per frame.

An alternative is to use vertex shaders (see 15) for the second pass. In the first pass we render the object normally, in the second a vertex shader applies the previous frame's and the current frame's transformation to each vertex. The difference between both gives a motion vector: Check the dot product between motion vector and surface normal, whether the vertex is facing away from the motion:

if it is facing away: take the vertex's previous position as output

if it is facing motion: take the vertex's current position
The length of the motion vector can be used to determine the α-values for blending.

8.7 Reflection

A common way to render reflections without Ray Tracing (see 9) is to render the real scene again mirrored on the reflecting surface (e.g. written to a texture map). Then these mirrored rendered objects are made semitransparent. To avoid rendering over opaque areas the stencil buffer can be used to hide them from rendering. Also remember to change from back-face culling to front-face culling or turn it off (slower rendering), since everything will be reversed.

9 Ray Tracing

The idea of ray tracing is to cast a ray for every pixel on the screen to the eye and follow it through all objects it intersects with. By that interactions between objects become feasible. Especially shadows and reflections become easy prey. We differentiate four kinds of rays:

- **primary rays**: ray from the eye through the screen pixel
- **shadow rays**: once a primary ray hits an object, a secondary/shadow ray is sent towards the light source
- **reflected rays**: once a shadow ray hits a light source, it is reflected according to surface properties
- **transmitted rays**: if the object is translucent, in addition to reflectance transmission rays are created and the result is carried back to the first intersection point

9.1 Viewing

The principle is very similar to what we got to know in perspective transformations, we simply draw the first object the ray, a 3D directed line with origin e and uwv coordinates, intersects with.

$$\vec{e} + t \cdot \vec{d}$$

where $\vec{b}$ is the vectors direction. We can replace $\vec{d}$ with $(\vec{s} - \vec{e})$, where $\vec{s}$ is the pixel on the screen we are processing. $\vec{s}$'s coordinates can be found be transforming the screen coordinates into the uvw-coordinate system.

$$w_s = \vec{n}$$

$$u_s = l + (r - l)\,\frac{i + 0.5}{n_x}$$

$$v_s = b + (t - b)\,\frac{j + 0.5}{n_y}$$

where (i, j) are the pixel's indices. Thus

$$s = \vec{e} + u_s\vec{u} + v_s\vec{v} + w_s\vec{w}$$

Note If $t < 0$ the object is behind the eye, and we don't have to render it.

Of course this method is highly view dependent, meaning, once the viewing changes, we have to recalculate everything.

9.2 Lighting

Lighting can also be done by casting rays. Once our eye-ray intersects with an object we send a ray from this intersection point towards the light source and compute lighting. If it doesn't hit the light source, we are inside a shadow. If it does we cast reflection and transmission rays according to the surface properties and then shade the pixel with the following component:

- direct illumination
  - material properties (color)
  - surface normal
  - light from the light source
- indirect illumination
  - incoming light from reflected rays
  - incoming light from transmitted rays

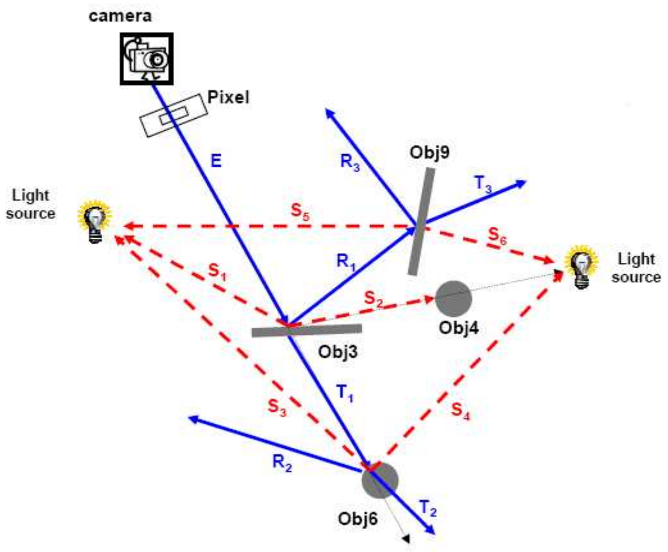


Figure 74: Recursive Ray Tracing

We see that Ray Tracing is a highly recursive procedure. This sounds much like a physical simulation of sun rays, yet it isn't. If we'd to simulate reality, we ought to start from the sun and cast rays on any point on any object and then into all directions. If one of there rays hits the eye, we can see it and render it. But the probability for a ray hitting the eye is really low.

Shadows

If a shadow ray does intersect with an object on it's way to the light source, we are in a shadow. We can basically use the same algorithm as for viewing rays, but we can simplify it: Since we are not interested in the closest intersecting object, we can stop the algorithm, when we find the first intersection with an object.

Soft Shadows

Soft shadows can be obtained by modeling an area light source by a number of point lightsources. However since casting rays for every of these point light sources would be tedious and the resulting shadows would still show hard visible boundaries of values of grey, an idea is to randomly select one or more of these point light sources for every ray.

Reflection

In case the shadow ray did not intersect with an object and we are dealing with a reflecting surface, we cast a reflection ray. The direction of the ideal reflection

$\vec{r}$ can be computed by

$$\vec{r} = \vec{v} + 2\left(\vec{v} \cdot \vec{n}\right)\vec{n}$$

In reality color is reflected differently depending on the color of the reflecting material. E.g. gold reflects yellow better than blue. We can respect this by adding a function to determine the reflection ray's color

$$\text{color} c = c + c_s \text{raycolor}\left(\vec{p} + s\vec{r}, s \in [\varepsilon, \infty[\right)$$

Transparency / Refraction

As with reflection we send transmission rays, if the surface material is translucent. Refraction is a bit different from reflection. The transmission ray will be bent, like e.g. a sun ray entering water. Thus the next object it hits, will appear translocated on the transmissive surface. We can make use of **Snell-Descartes Law** to compute the refraction angle:

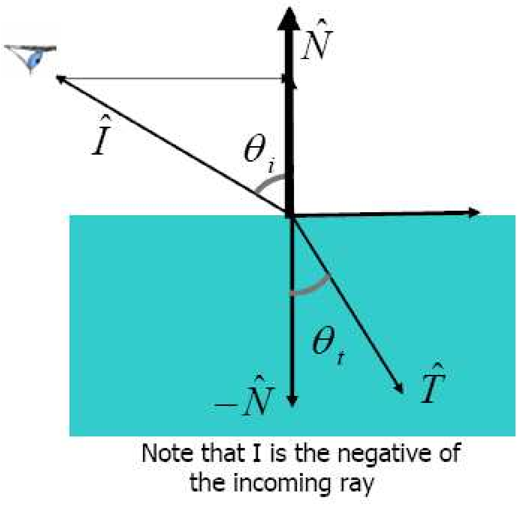


Figure 75: Snell-Descartes Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_i}{\eta_t} = \eta_r$$

$$\cos^2\theta_t = 1 - \frac{\vec{n}^2\left(1 - \cos^2\theta_i\right)}{\vec{n}_t^2}$$

where η describes the refraction property of the material and $\vec{n}$ is the surface normal, whereas $\vec{n}_t$ is the bend surface normal.

Epsilon ε

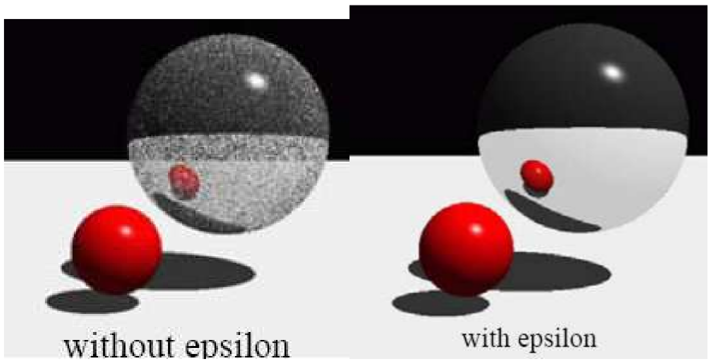


Figure 76: Add a small constant ε to counter numerical instability

Because of numerical instability, we always should add a small constant ε to the rays mentioned above, else the first intersection might be with the surface itself and result in unwanted self-shadowing.

Adaptive Depth Control

This recursive creating of reflection and transmission might never end. Therefore we should add thresholds:

Number Of Reflections ρ: The number of reflections threshold obviously stops if the ray has been reflected more than ρ times

Intensity τ: The intensity thresholds stops reflection when the reflection ray's intensity drops beneath τ. To reach a drop down of intensity we consider each materials individual attenuation properties.

9.3 Intersection

We can determine intersections by using implicit representations of our ray and of our objects.

Sphere

An implicit sphere is given by

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - r^2 = 0$$

or with vectors

$$(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) - r^2 = 0$$

Now we simply plug-in our implicit ray as a “point” on this sphere and check whether we still get 0:

$$\left(\vec{e}+t\vec{d}-\vec{c}\right)\cdot\left(\vec{e}+t\vec{d}-\vec{c}\right)-r^2=0$$

rearranging for t we can get a simple quadratic equation:

$$\left(\vec{d}\cdot\vec{d}\right)t^2+2\vec{d}\cdot\left(\vec{e}-\vec{c}\right)t+\left(\vec{e}-\vec{c}\right)\cdot\left(\vec{e}-\vec{c}\right)-r^2=0$$

Triangle

For triangles we can use the handy barycentric coordinates (see 2.3.3). Our implicit triangle was given by

$$\vec{a}+\beta\left(\vec{b}-\vec{a}\right)+\gamma\left(\vec{c}-\vec{a}\right)$$

Now for intersection we set them equal

$$\vec{e}+t\vec{d}=\vec{a}+\beta\left(\vec{b}-\vec{a}\right)+\gamma\left(\vec{c}-\vec{a}\right)$$

And we remember, that if $\beta,\gamma>0$ and $\beta+\gamma<1$ the point lies within the triangle.

Polygon

Our given polygon has m vertices $\vec{p_1}\dots\vec{p}_m$ and the surface normal $\vec{n}$. We start with checking whether the ray hits the plane the polygon is lying in

$$\left(\vec{p}-\vec{p_1}\right)\cdot\vec{n}=0$$

by plugging the implicit ray in as a point

$$\left(\vec{e}+t\vec{d}-\vec{p_1}\right)\cdot\vec{n}=0$$

solving for t

$$t=\frac{\left(\vec{p_1}-\vec{e}\right)\vec{n}}{\vec{d}-\vec{n}}$$

By that we find the point $\vec{p}$ where the ray hits the plane.

Now secondly we check whether $\vec{p}$ is inside the polygon or not. We do this by projecting polygon and point unto the most parallel coordinate-plane (e.g. by throwing away the biggest component in the normal vector) and create yet another ray, starting from $\vec{p}$ having direction $(\vec{p_i}-\vec{p})$. We only allow for positive t and check whether this ray intersects one or two time with the polygon edges (by checking consecutively against all edges).

One $\vec{p}$ is inside the polygon

Two $\vec{p}$ enters and leaves the polygon and must be outside

Attention: handle intersections at vertices and along edges with special care. Further more concave polygons can also lead to wrong decisions and more then 2 intersections.

Because of such an amount of special cases, most commonly testing per triangle and tessellation is preferred whenever possible.

Acceleration Techniques

We now have a stable method to compute intersections between rays and objects, however if we do these intersection test for each ray and each objects, we take 9x% of the computation time & power. But there are some strategies for accelerating the procedure of finding the first intersection:

- bounding volumes
- space partitioning
- ray coherence

Bounding Volumes

For each object we add a simply geometric object that completely surrounds it. E.g. a rectangle or a sphere. Then we intersect with those bounding volumes. In case the ray hits one of the we do an intersection test with the object inside. Another advantage is, that at the firsts step we do not even need to know where the ray hit our bounding volume, which makes the intersection test much easier.

Hierarchical Bounding Volumes

At a next step we might combine several objects, which are close to each other, to one big bounding volume (e.g. table + chairs + fruit bowl). If the big volume is hit, we intersect with the smaller ones inside and eventually with the objects themselves.

Uniform Space Partitioning

We partition the space into any number of uniform quadrants and only check objects which are (partly) inside the quadrants, the rays hits. Furthermore we can make use of the techniques developed for the Bresenham Algorithm (see 3.2). With that we do not need to check every space part but like with the line, only the next of the upper next one.

Octree Space Partitioning

Again we can move one level higher and repartition every quadrant. If the ray passes a quadrant we recursively scale down the partitioning for this quadrant and then check for objects within the passed by smaller quadrants.

Ray Coherence

The idea is to combine several rays into a bundle of rays.

9.4 Different Usage

Instead of doing only ray tracing, ray tracing can be used as an auxiliary technique for standard rendering:

- to generate high class textures (see 11 for all kinds of different textures)
- for vertex shaders: only cast rays from vertices (disadvantage: Gouraud Shading Artifacts)

9.5 Limits

Caustics

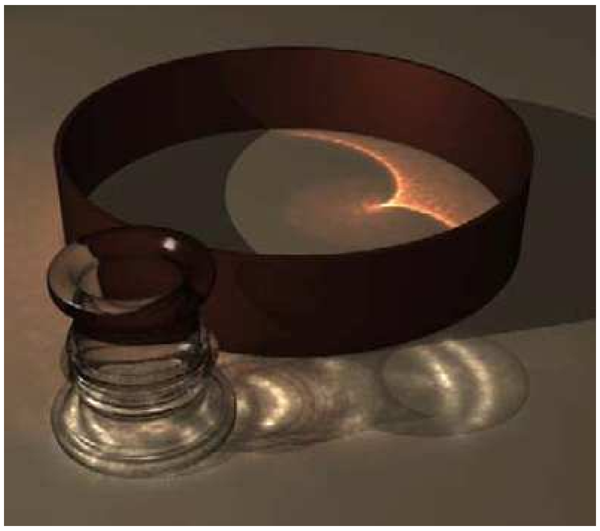


Figure 77: Caustics

Appear with simple reflection at certain angle at the interior side of a shiny cylinder and result in complicated curves (see figure).
Caustics can be modeled using photon maps (forward ray tracing):

1. Shoot a huge amount of photons from the light source.
2. Store their hitpoints in some 3D buffer
3. Get photon density: Use clustering algorithms to find hot spots (e.g. for every hit point, get the 50 closest hit points and calculate the max distance to the chosen hitpoint)

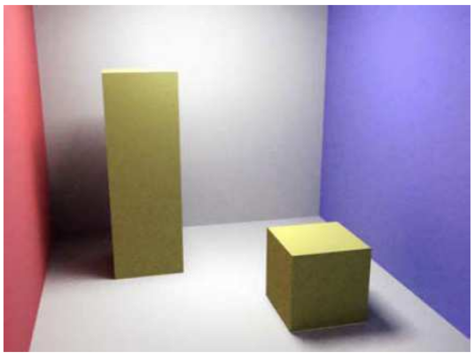


Figure 79: Color Bleeding

4. The denser the region, the more caustics we render

Color Transmission

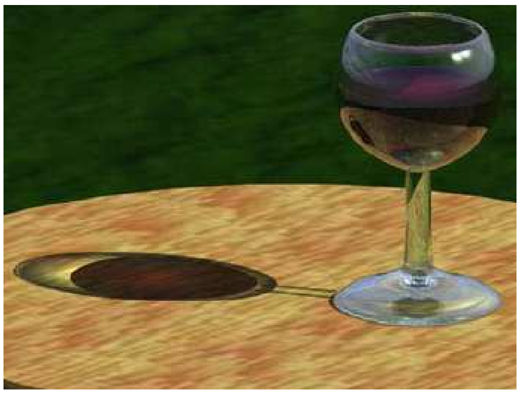


Figure 78: Color Transmission

Color Transmission means that to the shadow of an object color is added due to the translucent property of the object (see figure).

Color Bleeding

The transfer of color between nearby objects, caused by the colored reflection of indirect light.

9.6 Properties

+ enormously parallel: each ray could be cast in parallel, if we'd had enough parallel computational power ray tracing would exceed all other methods is speed and quality

+ global illumination

+ very excellent results

+ combines various different illumination aspects in one ray (reflection, refraction, transparency, shadows, soft shadows, global illumination, ...)

- very slow with current unfit hardware

10 Modeling

Modeling is all about choosing the right representation for the objects in a scene.

Postulations

- good representation of the object
- easy to render
- memory/runtime requirements
- interaction properties/possibilities
- creation process

10.1 Polygon Meshes

Polygons are the basis for most 3D applications, they can be rendered easily and express almost every object given due conversion time. Usually either triangle or quadrilateral meshes are used.

Polygon An ordered set of vertices: $P_0, P_1, \ldots, P_n$

Polygon Mesh A collection of polygons, such that any intersection between polygons of the mesh is either at a vertex or across an edge.

`OpenGL glBegin(GL_POLYGON): glVertex3fv(`P_0`); ... glVertex3fv(`P_n`); glEnd();`

It is important to differentiate between topology and geometry of a mesh:

Topology neighbourhood relations

Geometry the position of the vertices xyz-coordinates

In general whole object's can not be represented by a single polygon mesh. So our goal is to find the ideal decomposition into smaller polygon meshes. However the complexity of the stated problem is NP-complete.

10.1.1 Indexed Face Set (Shared Vertex Set)

The idea of indexed face sets is to use two separate lists:

vertex list capturing geometry (coordinates)

face list capturing topology (which vertices form faces)

10.1.2 Triangle Strips

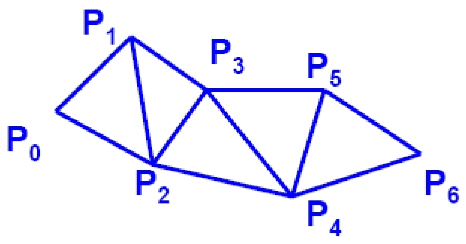


Figure 80: Triangle Strip

We try to model both geometry and topology in one list, by using a sequence of vertices, where every three vertices form a face. Of course this means we have to do a through ordering.

Example

Given is the list: $P_0P_1P_2P_3, P_4, \ldots, P_n$
This corresponds to the faces: $P_0P_1P_2; P_1P_2P_3; P_2P_3P_4;$...

OpenGL `glBegin(GL_TRIANGLE_STRIP): glVertex3fv(`P_0`); ... glVertex3fv(`P_n`); glEnd();`

General Triangle Strips The generalization means that both edges of an end triangle can be used to continue the triangle strip. If this method is not available (e.g. it isn't in OpenGL), you can insert dummy triangles to choose the edge you want to continue with. The advantage of general triangle strips is, that they can become much longer (see Stripification below 10.1.9)

10.1.3 Triangle Fans

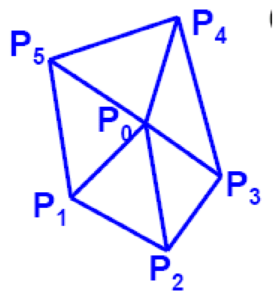


Figure 81: Triangle Fan

Triangle fans are very similar to triangle strips, except that every face starts at the same point P_0.

Example

Given is the list: $P_0P_1P_2P_3, P_4, \ldots, P_n$
This corresponds to the faces: $P_0P_1P_2; P_0P_2P_3; P_0P_3P_4;$...

```
OpenGL glBegin(GL_TRIANGLE_FAN): glVertex3fv(P0); ... glVertex3fv(Pn);
     glEnd();
```

10.1.4 Quad Strips

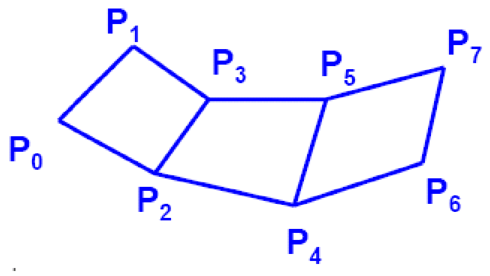


Figure 82: Quad Strips

Similar to triangle strips, but every four vertices form a face, and the interpretation of the ordering is different according to quadrangles.

Example

Given is the list: $P_0P_1P_2P_3, P_4, P_5, P_6, P_7, \ldots, P_n$

This corresponds to the faces: $P_0P_1P_3P_2; P_2P_3P_5P_4; P_4P_5P_7P_6;$...

OpenGL `glBegin(GL_QUAD_STRIP): glVertex3fv(`P_0`); ... glVertex3fv(`P_n`); glEnd();`

10.1.5 Enhanced Indexed Face List

Apart from modeling we often will want to access and change the rendered model. For that we need an efficient way to answer calls like: what are adjacent triangles, which triangles share an edge, which faces share a vertex or which edges share a vertex, therefore we might want a data structure allowing for faster access to those relations: the **Enhanced Face List**.

We enhance the face list by three reference pointer to the three neighbouring triangles by e.g. a pointer to the third vertex creating the neighbouring triangles.

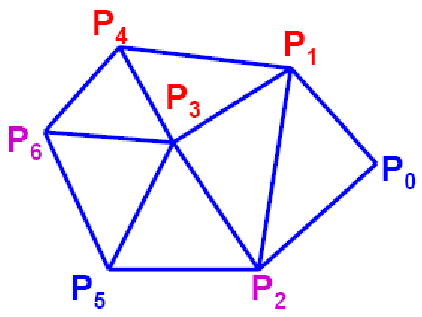


Figure 83: Enhanced Face List Example

vertex list Triangle$_0 = x_0, y_0, z_0$, Triangle$_1 = x_1, y_1, z_1$, Triangle$_2 = x_2, y_2, z_2$, Triangle$_3 = x_3, y_3, z_3$

face list Face$_0 = 0, 1, 2$, Face$_1 = 3, 2, 5$, Face$_2 = 1, 4, 3$, Face$_3 = 3, 5, 2$

enhanced face list Face$_0 = 3, -1, -1,$ Face$_1 = 5, 0, 4,$ Face$_2 = 6, 2, -1,$ Face$_3 = -1, 1, 6$

10.1.6 Directed Edges

This problem is commonly solved by giving edges a direction. This is by replacing the face list with a list of directed edges, with two entries

- start vertex of edge
- pointer to the opposite edge index

indexed by an edge index.

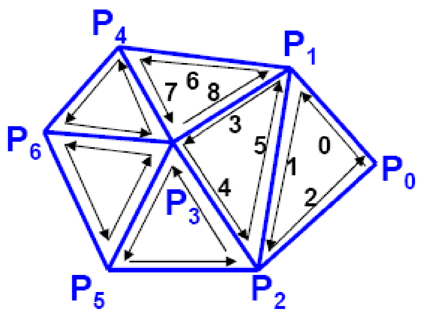


Figure 84: Directed Edges

edge list $\mathrm{Edge}_0 = 0, -1$, $\mathrm{Edge}_1 = 1, 5$, $\mathrm{Edge}_2 = 2, -1$, $\mathrm{Edge}_3 = 1, 8$, $\mathrm{Edge}_4 = 3, x$, $\mathrm{Edge}_5 = 2, 1$

10.1.7 Normal Vectors

The normal vectors of surfaces can be obtained either in the design process (e.g. when using NURBS or implicit surfaces) or taken as average of the involved edges' normals:

$$\vec{n}_0 = \sum_{i=1}^{n} P_i \cdot P_{i+1}, P_{n+1} = P_1$$

10.1.8 Face Orientation (Back Face Culling)

This is rather important, since usually only one side of the object is visible (unless the viewer is inside the object, or there are holes in it). Thus we want only to render the faces facing front (towards the eye) and leave the rest unrendered, this effects in about 50% less polygons to render. The idea is to implicitly store the orientation in the ordering of the vertices:

clockwise face is seen from the back

counter-clockwise face is seen from the front

Properties

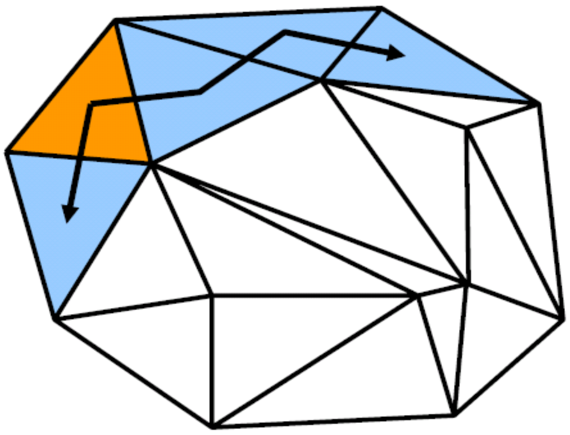
- approximation to smooth geometry (no silhouettes)
- very large number of polygons
- very bad interactivity
- difficulty to increase/decrease the resolution of the object
- difficult to extract geometrical information (e.g. curvature)

10.1.9 Stripification

A simple approach to find a decomposition into triangle strips:

Algorithm 13 Simple Stripification

```
1. randomly select an unused triangle
2. start a triangle strip along one edge
   ⟳2: until an used edge has been reached
3. continue triangle strip into opposite direction
   ⟳3: until an used edge has been reached
4. ⟳1: until the polygon is completely decomposed
```



The SGI approach adds a little improvement: The starting triangle (orange) is not selected randomly, but by the number of least unused neighbours (if ambivalent, check the neighbours as well)

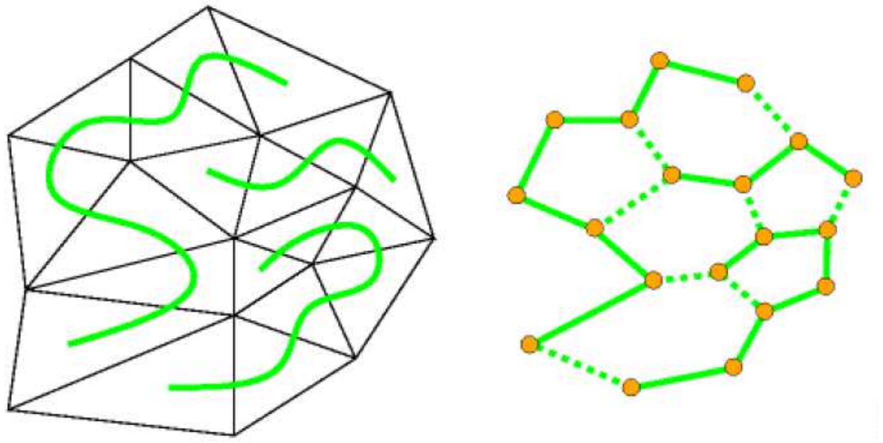
Even better results can be obtained by using general triangle strips, since then the strips can become much longer. Furthermore better usage of the vertex cache (see below) can be made:

Algorithm 14 Tunneling

Each triangle is seen as a node in a graph.
Then we use graph algorithms to find paths between the nodes, without using a node twice.
Eventually we seek for "edges" connecting two end points of such paths (dotted lines). These are called **tunnels**.
The so discovered "best" path is then the general triangle strip.

1. generate a trivial path set of triangles (e.g. empty set, all isolated)
2. ⭮ for each path endpoint o_i:
   → for each other path endpoint o_j: search for a tunnel $o_i o_j$
3. was a tunnel found?
   (a) `TRUE`: swap all dashed and solid edges →2
   (b) `FALSE`: Return path as general triangle strip

With every swap the number of triangle strips gets effectively reduced by one.



Note that even tunneling does not return the global minimum of strips, yet the results are pretty good. For example a bunny object consisting of 70.000 triangles results in about 700 strips when using SGI Stripification and in 158 when using tunneling.

> **Note** Be careful about the triangle's orientation (vertex ordering)! When trying to continue a strip at one of its endpoints, the strip might suddenly end in a triangle that's order is different to the original starting triangle. However this new triangle will be chosen as starting triangle for the strip, so the order of the complete strip is reversed resulting in wrong rendering.

10.1.10 Vertex Cache

The vertex cache is a cache for processed vertices, normals, texture coordinates or color arrays. The idea is that for example in a triangle strip the same vertex's attributes will be used multiple times in short notice (up to 6 times). Therefore a little cache for the last n processed vertices can give us an enormous speed improvement. However for the look-up statement to work, we require indexed face sets (without we don't know whether vertex$_i$ equals vertex$_j$). In the optimum case (see figure below), which is not as rare as you think (e.g. tessellation of Bézier Surfaces), half of the vertices are already in the cache or in other words, fetching one single vertex can lead up to two new triangles.

Example

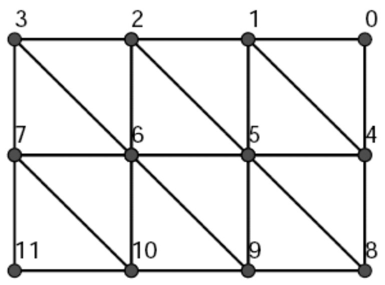


Figure 85: Triangle Strips, where the vertex cache can be optimally used (a vertex may be called up to 6 times)

If we use a vertex cache of size 7 for this strip, we reuse half of the vertices:
Cache: $\boxed{7}\boxed{4}\boxed{1}\boxed{5}\boxed{2}\boxed{6}\boxed{3}$, the vertices 4,5,6 and 7 are used again.
If however we limit the size to 6:
Ch ache: $\boxed{3}\boxed{7}\boxed{1}\boxed{5}\boxed{2}\boxed{6}$, the vertices 4,5,6 and 7 are overwritten, before we can reuse them, and we end in no reuses.
A typical size for vertex caches is $n = 16, 32, 64$. The optimal ratio between processed vertices and triangles is 2.
A combination between Stripification using the Vertex Cache is to stop the strip, once the cache overruns. In this case the next strip started will make reuse of at least half of the cached vertices.

Triangle Strip Length

Note that we need to distinguish two cases to determine the optimal triangle strip length. This differentiation is made on the used data-type. If we use indexed face sets, only then will we be able to make use of the vertex cache, and the optimal length of an indexed face set should be limited by the vertex cache capacity. If we however have simple triangle-strips without topology information, the rule: the longer the better, counts and we might use tunneling to greedily get longer ones.

Indexed Face Sets limit size to vertex cache capacity

Triangle Strips the longer the better

10.2 Parametric Surfaces

We can decide between using polynomial or rational curves and between using global or piecewise models.

42	global	piecewise
polynomial	Bézier	B-Splines
rational	rat. Bézier	NURBS

Instead of simple monomials (x^n) we will use more suitable basis functions:

Bézier Bernstein Polynomials (see 2.7)

B-Splines B-Spline basis functions (see 2.7)

A curve is then represented by a polynomial linear combination of one these basis functions and so called **control points** c_n

$$F(x) = \sum_{i=0}^{n} c_i B_i(x)$$

This is only one possible kind of representation, we could also use implicit curves (see 2.4):

$$f(x, y, z) = 0$$

where the implicit function f returns 0 if the point (x, y, z) lies on the curve. Implicit representations are especially useful for geometric primitives like spheres or planes, where fixed formulas exist. In the other cases Computer Graphic Designer usually prefer parametric curves, since because of the free parameter, they are easier to sample and to draw.

Interpolation

Having these control points c_n we have to estimate the values in between, approximating them by a polynomial. One problem is, that whilst a polynomial interpolating points c_n is unique for every degree, a curve has infinite many representations. The process of transforming one representation of a curve into another of the same curve is called **reparametrization**. We can make use of this to find a representation that is most convenient for our application.

Apart from that polynomials are functions, that means for every x there is one and only one y. Yet for a curve, there can be more than one y (e.g. a circle). Secondly polynomials of a high degree tend to oscillate (overfitting, see Pattern Recognition). Therefore instead of a global model often a piecewise model appears to be more fitting.

Linear Interpolation Linear Interpolation means to find the simplest curve between any number of points and distributing the values of these points in between linearly. E.g. for two points we have

$$p(t) = t \cdot p_0 + (1-t)\, p_1, t \in [0;1]$$

Bilinear Interpolation Bilinear Interpolation means linear interpolation in two directions (across a patch).

$$p(s,t) = (1-s)(1-t)\, p_{00} + s(1-t)\, p_{10} + (1-s)\, t \cdot p_{01} + s \cdot t \cdot p_{11}$$

where $s, t \in [0;1]$ are the free parameters defining the patch.

Trilinear Interpolation Trilinear Interpolation means linear interpolation in three directions (through a room).

Approximation

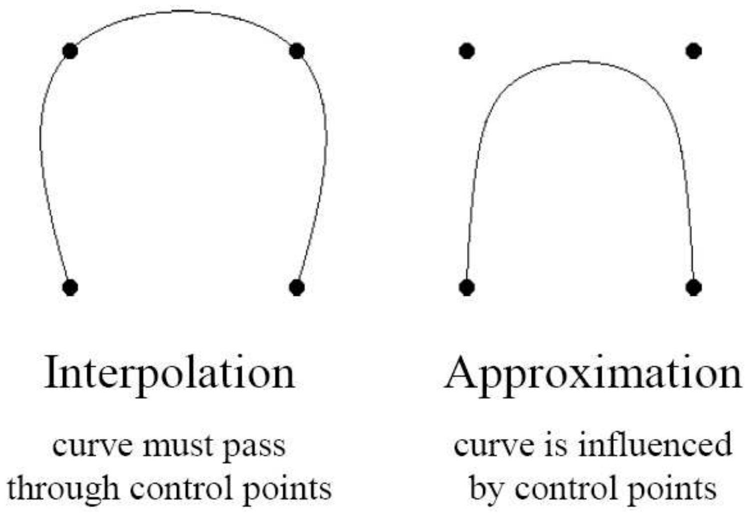

Interpolation
curve must pass through control points
Approximation
curve is influenced by control points

Figure 86: Difference between interpolation and approximation

We differentiate the terms interpolation and approximation in so far, that with interpolation the curve/polynomials must pass every control point, while with approximation they only influence the curve's graph.

10.2.1 Bézier Curves

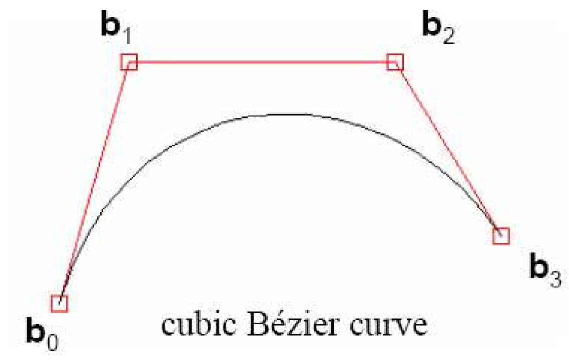


Figure 87: A Bézier Curve with four control points: b_0, b_1, b_2, b_3

Bézier curves chose the later idea and approximate control points rather than interpolate them. Exceptions are the end points of the curve, which are interpolated. As you can see in the figure above, the direct lines between the points are tangents of the actual curve. The degree of the Bézier curve is the number of control point minus one.

An intuitive way to understand how we can draw such a non discrete function, i.e. a perfect curve, to the screen is illustrated by corner cutting

Corner Cutting

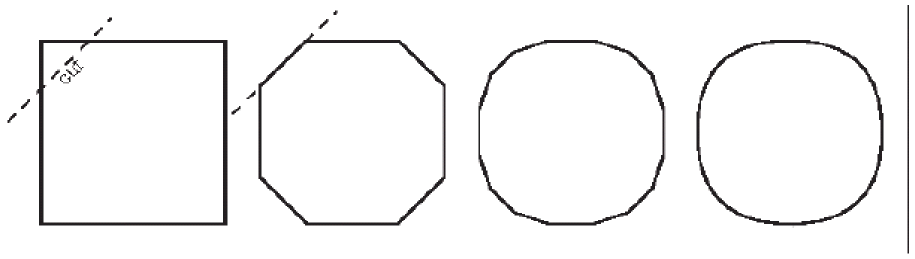
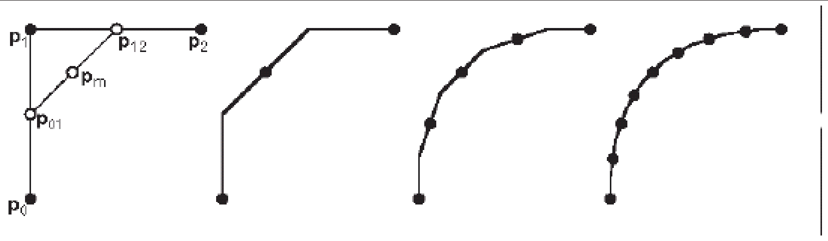


Figure 88: Corner Cutting

Corner Cutting means to successively cut the corners off, to make the corner points more smooth. We cut them of by spanning lines between a corner point and cut/clip off the outside. The limes of infinite many subdivisions is indeed a smooth curve.

Algorithm 15 Corner Cutting



```
subdivide($p_0, p_1, p_2$) {
    $p_{01} = \frac{p_0+p_1}{2}$
    $p_{12} = \frac{p_1+p_2}{2}$
    $p_m = \frac{p_{01}+p_{12}}{2}$
    subdivide($p_0, p_{01}, p_m$)
    subdivide($p_m, p_{12}, p_2$)
}
```

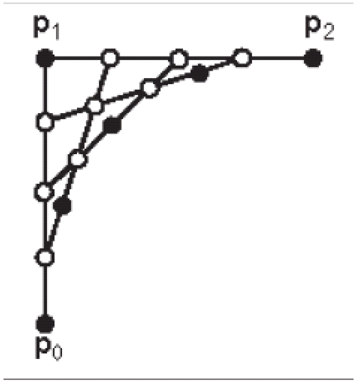


Figure 89: The midpoints (black points) resulting from corner cutting, make up the curve approximating the control point p_1 and are thus those, which we draw on the screen. Some threshold can determine the number of subdivisions.

We use corner cutting to approximate the control points and draw our “smooth” curve on the screen. We only use the midpoints p_m resulting from corner cutting to define the shape of our curve (see figure above).

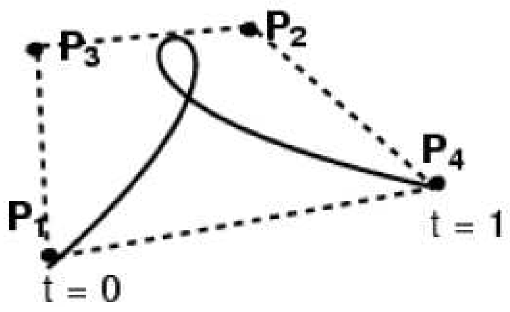


Figure 91: Ordering of Bézier control points

Note that the order of the control points is very important, as illustrated in this figure.

Algorithm Of Casteljau

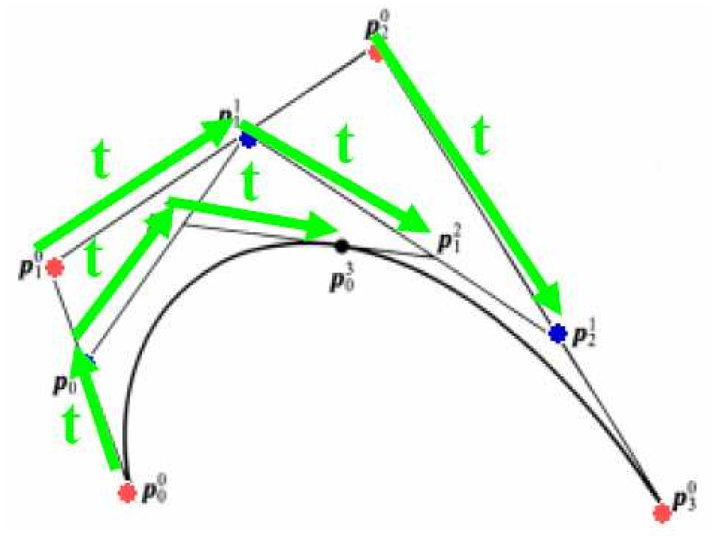


Figure 90: Algorithm of Casteljau

An generalization to corner cutting is a number of successive linear interpolations called the algorithm of Casteljau.

Order

Continuity

Of course we also will have to connect Bézier Curves. In this case we want to assure that we have at least C^0 and C^1 continuity[7]:

We have

C^0 if the graph has no gaps

C^1 if the tangent vectors match (no sharp corners)

With these assurances we have a curve without gaps and sharp corners. However for some applications continuity up to C^5 is useful, so we bid special care depending on the application. For example C^2 continuity is needed, when the object is in motion and we want the motion to be smooth.

Quadratic Bézier Curves

Quadratic Bézier Curves have only one approximation (control) point p_1. For $t \in [0;1]$ we get

$$p(t) = (1-t)\left((1-t)\,p_0 + t \cdot p_1\right) + t\left(t \cdot p_0 + (1-t)\,p_1\right) + t \cdot p_2$$

$$p(t) = \left(1-t^2\right) p_0 + 2\,(1-t)\,p_1 + t^2 p_2$$

thus we result in having the weights $w_0 = \left(1-t^2\right), w_1 = 2\,(1-t)\,, w_2 = t^2$. These weights or control points are often referred to as **blending functions**.

So the curve is the weighted average of the control points:

$$p(t) = \sum_{i=0}^{n} w_i(t)\,p_i$$

The weights must always sum up to 1 and we allow no negative weights.

Cubic Bézier Curves

Cubic Bézier Curves are also based on the subdivision procedure and have one additional point to approximate. This results in 4 control points $w_0 = (1-t)^3, w_1 = 3\,(1-t)^2\,t, w_2 = 3\,(1-t)\,t^2, w_4 = t^3$

Bézier Curves Of Higher Order

Dealing with Bézier Curves of any order, we can find a generalization of the blending functions using Bernstein binomial coefficients:

$$w_i^n(t) = \frac{(n-1)!}{i!\,(n-i)!} \cdot (1-n)^{n-i-1}\,t^i$$

[7] C^ncontinuity means the function is continuous and all of it's derivatives up to the n^{th} also are.

Bézier Surfaces

Bézier Surfaces are a generalization of Bézier Curves in 3D. A Bézier Surface is also given by the average of all control points:

$$p(s,t) = \sum_{j=0}^{m} \sum_{i=0}^{n} w_{ij}(s,t) \cdot p_{ij}$$

where the blending functions w_{ij} must be continuous.

There is a great property of Bézier Surfaces (continuous blending functions), that allows us to separate the 2D blending functions into two 1D ones.

$$p(s,t) = \sum_{j=0}^{m} \sum_{i=0}^{n} w_j(s) \cdot w_i(t) \cdot p_{ij}$$

where $w_j(s) \cdot w_i(t)$ is a tensor product (see 2.1.2), since one goes in x and the other in y direction. This also means we can give two 1D curves to make a surface. We simply have to find a matrix representation of our function (which we can, because we are dealing with linear functions) and handle these matrices as they would be vectors and apply the tensor product.

Properties

+ the most striking advantage is that an object described by such curves is completely resolution independent and will show no signs of aliasing.

+ easy adjustment: the curve's shape is manipulated by manipulating control points (define tangents on the curve)

+ the curve always remains in the convex hull of control points

+ affine transformations on the curve $\triangleq$ affine transformations of the control points (**affine invariant**)

- the curve depends on all control points, so changing a single one reshapes the whole curve (this can sometimes be an advantage as well)

- many control points lead to a high-degree polynomial (degree = number of control points minus one)

A small simple applet to play with Bézier curves can be found here Bézier Curve Applet `http://www2.mat.dtu.dk/people/J.Gravesen/cagd/decast.html`. Look how moving a single control point influences the whole curve. If you still want to see more, here's an applet illustrating Bézier Surfaces Bézier Surface Applet `http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-96to97/anson/BezierPatchApplet/index.html`

10.2.2 Uniform B-Splines[8]

A way to avoid both negative properties mentioned above is to use splines. Splines are polynomials of a lower degree that are combined to approximate a polynomial of a higher degree. Basis functions for B-Splines can be looked up here: 2.7. With these basis functions a B-Spline is defined as:

$$S(t) = \sum_{i=0}^{n-1} p_i b_i(t)$$

Now geometrically speaking we combine these splines by shifting the basis functions to given so called **knot points** k_i, which serve as connection points between the splines. In our case we choose them uniform.

$$k_i = k_{i-1} + 1$$

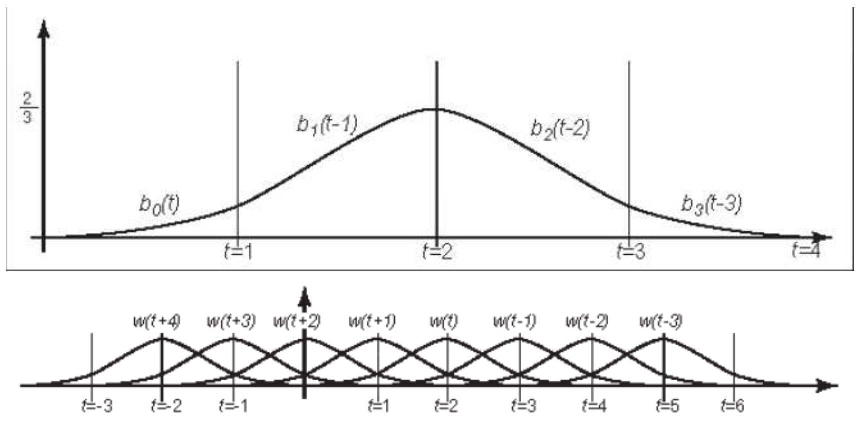


Figure 92: Shifting the splines basis function to the control points

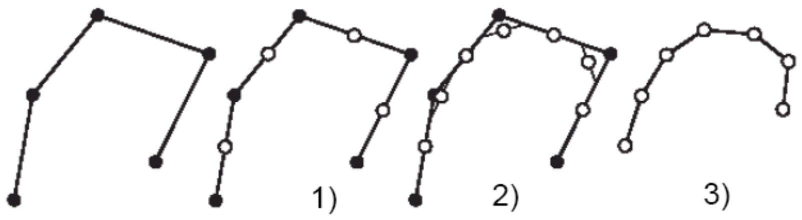
This choice for knot points in uniform distance results in having the same spline over and over again, only translated to the knot point. This also means we don't have to store the knot points, since they can be created automatically and need only to store the control points.

Subdivision Process

The subdivision process is similar to the Bézier case, yet we do not take the midpoints. Instead we use the midpoints between midpoints:

[8]From Basis-Splines, cause they are all created from the same set of basis functions

Algorithm 16 Subdivision Process for B-Splines



1. Choose midpoints in each segment of the control polygon
2. Connect midpoints of these and the original control points
3. Also use midpoints of the corner segments

Cubic B-Splines

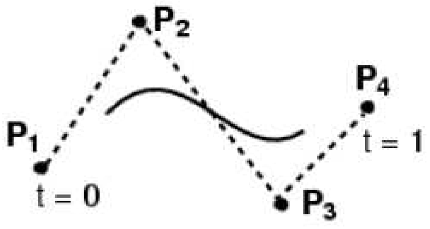


Figure 93: Cubic B-Spline

Like with cubic Bézier Curves we have four points, but now, even the end points are not necessarily on the curve. All the properties of Bézier Curves do also count here, but only locally: local convex hull, locally continuous, local control by control points.

Properties

+ every spline has C^2-continuity
+ constant degree of basis functions: more efficient and more numerically stable
+ local control of control points: effects are only local
+ bound by the convex hull of the points
+ affine invariant

- only an approximation like polygons, no accurate modeling

- no control points on the curve (the curve will be defined by parameter values)
- removing of control points can lead to a complete restructuring of the whole curve, since the number of control points between two knot points is constant

A series of applets illustrating uniform B-Splines can be found here: B-Spline Applet `http://www.ibiblio.org/e-notes/Splines/Basis.htm`. Look how the B-Spline functions all look equal, expect for being translated to the knot points. Try to move the control points to influence the curve. Moving one control point will only influence the part of the curve, that is dependent on it.

You can also try to destroy the uniform spacing of the knot points on the right side of the applets and see how the representation changes.

10.2.3 NURBS

Non Uniform Rational B-Splines (NURBS) are a generalization of B-Splines

rational ratio of two polynomials instead of one cubic one (results in an exact representation of conics (e.g. cylinders, circles)

non-uniform different spacing between knot points (results in an easier adding and deleting of control points, simply add the point 2.5 between $2,3 \rightarrow 2,2.5,3$ or simply remove the point 3 between $2,3,4 \rightarrow 2,4$)

These changes mean that we have to define and store two knot sequences for x and y direction; $w_i(x), w_j(y)$. We can combine them for a knot matrix defining a surface, like we did with Bézier Curves, by combining them with a tensor product.

$$N(x,y) = \sum_{j=0}^{m} \sum_{i=0}^{n} w_i(x) \cdot w_j(y) P_{ij}$$

or written with the ratio

$$N(\vec{u}) = \frac{\sum_{i=0}^{n} h_i p_i w_{i,k,\vec{t}}(\vec{u})}{\sum_{i=0}^{n} h_i w_{i,k,\vec{t}}(\vec{u})}$$

where $\vec{t}$ is the knot vector and k the B-Spline degree parameter. Where P_{ij} is an array (a matrix) containing all the control points.

10.3 Constructive Solid Geometry (CSG)

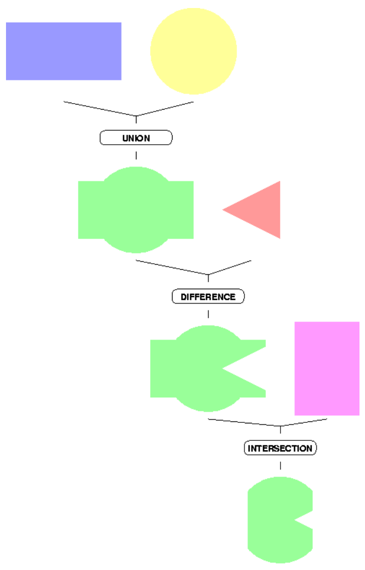


Figure 94: Basic Operations of Constructive Solid Geometry

The idea of Constructive Solid Geometry is to use a set of operations to combine solid shapes. These operations can be seen as operations on sets ($\cup, \cap, -$).

+ can efficiently be combined with ray tracing

10.4 Subdivision Surfaces

10.5 Procedural Models

Procedural models provide procedures that can generate points on a curve (model), that are neither implicit nor parametric. A good example for procedural models are fractals.

10.6 Hierarchical Modeling

10.6.1 Scene Tree / Scene Graph

The idea of hierarchical modeling is to gather objects as chunks. The root is the scene itself, partitioned by object groups that share a certain geometry (e.g. tables), partitioned by single objects, partitioned by object parts, partitioned by primitives. This hierarchy is called **scene tree**. When focusing on shared geometric properties, we speak of **scene graphs** rather than trees.

10.6.2 Scene Description

A scene consists out of: Camera, Light, Background, Materials and Objects. We describe each of them separately.

10.6.3 Class Hierarchy

A kind of object oriented approach. For example a possible super class is `Object3D`. This super class is inherited by `Sphere`, `Cylinder`, `Plane`, `Triangle` or `Group`.

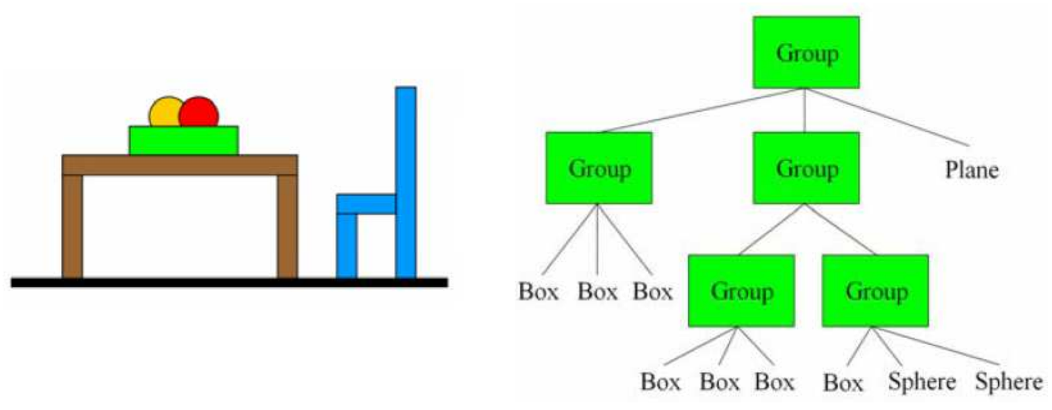


Figure 95: Organization tree using a class hierarchy

Using such a class hierarchy, we can describe the scene as a tree of groups.

In this approach we can define materials inside of group classes for each group member.

Scene Transformations

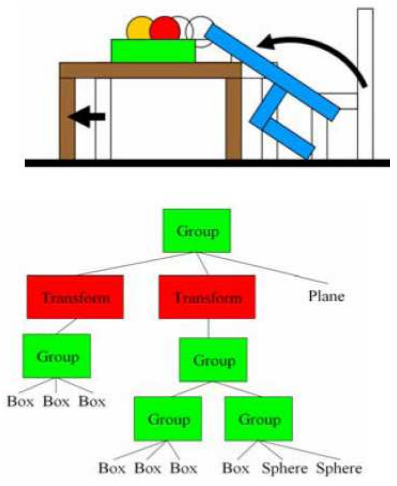


Figure 96: Scene Transformations

Adding a transformation class as Object3D, we can also describe transformations within the scene.

Add a class `Transformation` as an `Object3D`. By making `Transformation Object 3D` we can logically place them in the organization tree and order all affected real object groups below it.

10.6.4 Scenegraph API

`OpenGL` is powerful, but we have to create objects from the bottom by lines of code, and have little assistance in picking and transforming objects during the creation. Furthermore `OpenGL` is more hardware than user oriented and the process of creation is imperative rather than descriptive (which would be more intuitive). Now scenegraph APIs are usually based on OpenGL and therefore share it's advantages (e.g. hardware independent), yet they offer the above mentioned features making the process of creating easier and more intuitive.

A typical scenegraph API covers

- scene description: geometry and attributes (hierarchical modeling)
- reutilization: leads to DAGs
- validity and propagation attributes
- a GUI for easy modeling and arranging of objects
- typical basic elements/nodes: camera, light sources, background, shape, group, geometry, transformation, root

Some popular APIs are:

- OpenInventor (SGI)
- Java 3D
- OpenScenegraph

10.7 Level Of Detail (LOD)

Representing a model in detail may not always be good, especially if we look for speed. For example imagine a car close to the far plane, very distant to the viewer. To model this car taking up so few pixels, a really simple model is sufficient. However if the car is right before the user, we need it with every detail, we can get. Now the idea of LOD is to provide objects and textures in a different level of detail resp. resolution, decide which LOD is best for the objects in the scene and provide a way to switch between different LODs for interactivity.

Also note the LOD rendering is completely solved by curves, NURBS etc. (see previous sections), since they provide different LODs in their geometric description.

10.7.1 LOD Creation

First of all to select and switch between LODs, we need different LODs per se.

Handmade the most straightforward method is to provide them yourself. Advantage is that they will be asjusted to the application and can be tested to look good. On the other hand this takes time and makes them application dependent.

Edge Collapse move two vertices forming an edge to one point making an edge collapse. One collapse removed two triangles, three edges and one vertex. Supplying a history of collapses, we can lostless reproduce the higher LOD and don't have to save different LODs per se.

Contraction a generalization of edge collapse allowing edges and triangles to collapse as well. Also very important is to avoid critical contractions at any cost (critical in the sense of hugely deforming the object), however those can be easily detected by the direction change of affected surface normals.

Bump Maps a curious idea is to turn actual geometry information into a bump map and render the object flat (just take the normals and put them into a normal map).

Actually "**maintaining surface properties**", like avoiding direction changes of surface normals, has proven to be a good cost function. Another good cost function is based on the "**least perceptible change**" in the resulting image. It is measured by a simple distance comparison between the images resulting from both LODs, however simple it is expensive to compute.

10.7.2 LOD Switching

LOD Switching is quite important, because without a proper strategy a significant blopping between LODs will be visible. In the worst case the levels will rapidly switch for and back resulting in blopping and flickering.

Blending doing a blend over two LODs over a short period of time. For example by rendering the old LOD opaque and the new with increasing α-value (- blending is very expensive).

Alpha α-LODs actually use only a single model, but this model's αvalue increases with distance to the viewer disappearing at some point all together. After this disappearing a significant speed up will happen, but in contrast to the speed idea of LOD this method will result in no efficiency gain, while the object is still visible.

CLOD standing for Continuous Level Of Detail. The idea is to provide one complex model and successively derive less complex models from it (e.g. 2 pixels less complex per stage). The idea is to successively shrink all edges until both endpoints meet and the edge will entirely disappear. Each "model" must thus contain a pointer to the next LOD (some LODS will look ugly, the object always appear to be changing).

10.7.3 LOD Selection

Range The most forward way is to place the decision on the distance to the viewer.

Projected Area In this method the bounding volume of the object is projected onto the screen and the number of pixels is counted to determine the LOD (requires approximation of solid angles).

Projected Pixel Another possibility is to project a pixel onto a associated texture map (if given) and measure the number of textures influencing it. This is especially useful for finding the right Mip Mapping (see 11.5.2) LOD resp. resolution. E.g. by using the longer edge of the parallelogram formed by the pixel's cell as a measure.

Object Type E.g. a clock on the wall is less important than a wall.

Focus The viewer's focus determines the LOD. E.g. during a soccer game the area around the ball needs a high LOD, whereas the other playground can be rendered at a low LOD.

Since almost the only value of using LODs is a gain in speed, in general an objective function can be approached and used as a metric for selection:

$$\sum^{\#\text{ objects}} \frac{\text{Benefit}\,(O, L)}{\text{Cost}\,(O, L)}$$

where O is the object rendered and L the associated level of detail. This can be especially useful when we want to guarantee a minimum frame rate.

11 Texture Mapping

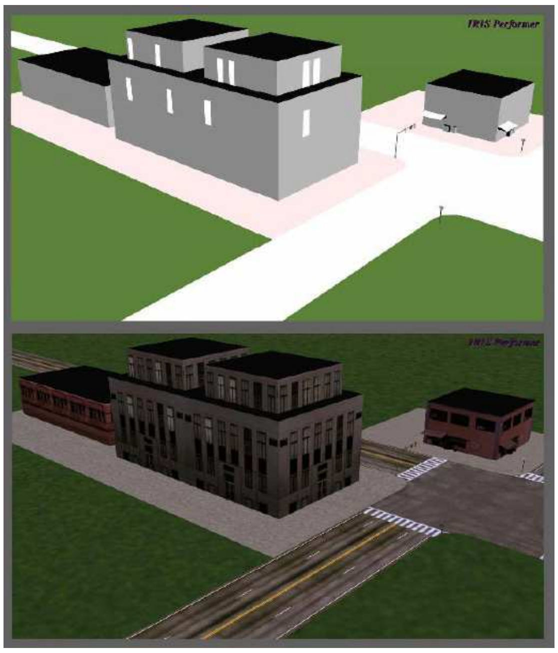


Figure 97: A scene with and without texture mapping

The problem of the techniques we introduced so far, is that if we really want a detailed surface on an object, the means various different materials, height differences, colors and other features, the modeling process would become really complex and inefficient. Therefore Ed Catmun and Jim Blinn thought of something else that works much like wallpapers on walls. Instead of really modeling the outward appearance we define a 2D image and wrap it around the object (1D and 3D "images" are also possible). This is called texture mapping. Effectively we have to find a coordinate mapping from the image coordinates to our object coordinates. We differentiate **static textures** (raster images) or

procedural textures that are computed on the fly:

$$T : \mathbb{R}^2 \rightarrow \mathrm{RBG}\,(A)$$

texel a pixel in the texture (from texture element)

Properties

+ adds visual complexity to objects in a simple way

+ great performance compared to "real" modeling

+ can even be used for reflectance properties (see environment maps)

- dependent on the rasterization method (ray tracing, scanline deliver different results). Solution: Do Perspective Interpolation 11.2

11.1 Noise Textures

Noise Textures are an example for procedural textures. We randomly assign color values of a certain range to get something like a TV static. This is also called **white noise**, because it's following an uniform distribution. For a more smooth noise we can use a technique called **Perlin Noise**. Key features of Perlin Noise is to use a lattice and color vectors rather than color values and interpolate between them using weighting function ω.

$$n\,(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor+1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor+1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor+1} \Omega_{ijk}\,(x-i, y-j, z-k)$$

$$\Omega_{ijk}\,(u, v, w) = \omega\,(u)\,\omega\,(v)\,\omega\,(w)\,(\Gamma_{ijk} \cdot (u, v, w))$$

$$\omega\,(t) \quad = \quad \begin{cases} 2\,|t|^3 + 3\,|t|^2 + 1 & \text{if } |t| < 1 \\ 0 & \text{otherwise} \end{cases}$$

where Γ contains a hash function ϕ for accessing precomputed unit vectors in an array G:

$$\Gamma_{ijk} = G\,(\phi\,(i + \phi\,(j + \phi\,(k))))$$

11.2 2D Texture Mapping

We are given a texture image of size (n_x, n_y) and have texture coordinates u, v to access texels on the texture. Often 2D texture mapping is done by first mapping texels to every vertex and then interpolating between them.

1. Normalization: First we see that we limit the texture range to $[0; 1]$, values outside of this range can for example be computed by a periodic extension of the texture or by clamping (both discussed later on 11.7).

2. Interpolation: A pixel value usually does not directly correspond to one single texture value, but lies e.g. close to the center of four neighboured pixel. In such a case we can apply a interpolation technique like: Nearest Neighbour, Bilinear or Trilinear Interpolation.

   (a) Nearest Neighbour: Take the texture value closest to the pixel

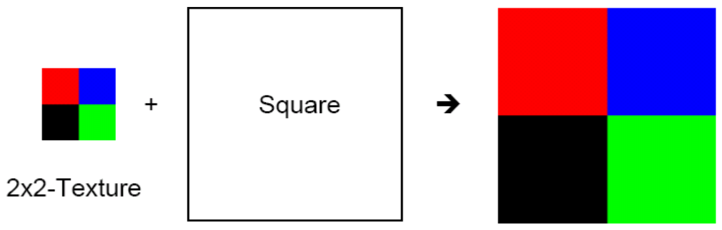


Figure 98: Nearest Neighbour Interpolation

   (b) Bilinear Interpolation: Interpolate between neighbouring textures close to the pixel

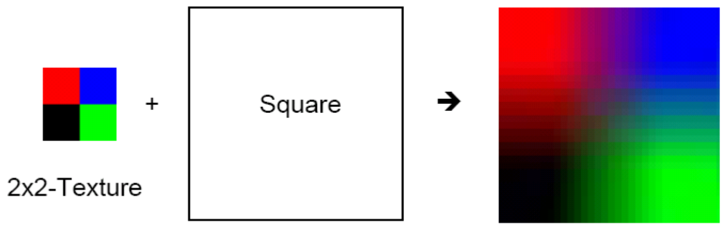


Figure 99: Bilinear Interpolation

   (c) Trilinear Interpolation: The same as bilinear interpolation for textures close to the pixel in three directions (3D)

Texture Coordinates

The texture coordinates u, v can be gotten by:

- **delivered by model data**
  The u, v coordinates are generated during the modeling phase (e.g. parametric surfaces) and stored in a second list next to the vertex list. This means every vertex in the list has both x, y, z coordinates as well as u, v coordinates. This also means that we can easily add additional features to vertices.

- **run time computation (parametrization)**

  This is trivial for geometric primitives like spheres or cubes. For other objects, we can enclose them into a geometric primitive and project from this enclosing primitive onto the object. Of course the results vary for each method, therefore it is application dependent which to choose. The easiest way is to use planar projection (see figure).
  Possible are: parallel (planar) projection, cubical, cylindrical or spherical projection

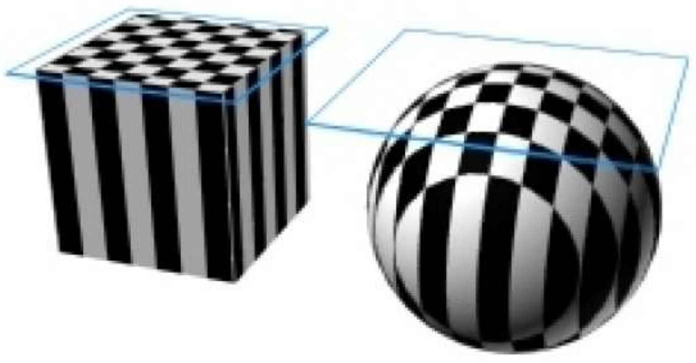


Figure 100: Planar Projection

- **automatic generation from vertex coordinates**

  This is what OpenGL does (`glTexGen()`). Think of it as a dia/beamer projection (see figure). We do this by giving a fixed rule how to map coordinates for all objects by defining a linear function, i.e. a matrix. Now by this matrix we can define an arbitrary projection, e.g. an orthographic one, using only the linear 3×3 part or a perspective making use of the last row of the whole 4×4 matrix. With the latter one we can e.g. perform a dia projection from the light source. Using vertex shader, we can get even more sophisticated projections by defining rules how to transform vertices.

  One advantage of this method is that we can manipulate the texture coordinates by this 4×4 texture matrix (which has its own texture stack). Not that we don't have direct access to world coordinates, therefore we need to apply the object's coordinates first to the ModelView matrix before we can throw them into the texture matrix.

$$p_{\text{texture}} = M_{\text{texture}} M_{\text{ModelView}} p_{\text{object}}$$

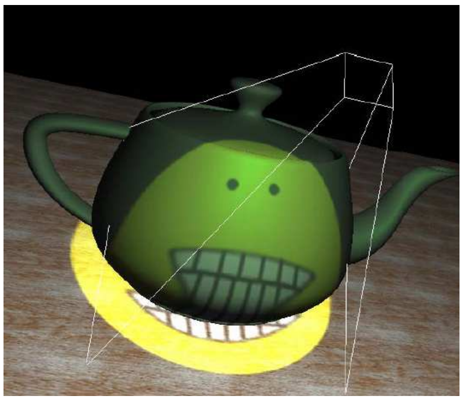


Figure 101: Dia Projection

Usage With this method you can for example go out into the RealWorld© take a picture with a digicam and use it as dia-texture. However the most used application are shadow maps.

Example: Sphere / Runtime Computation

1. Get polar coordinates for the vertex (x, y, z) on a sphere with center (x_c, y_c, z_c) and radius r:

$$x = x_c + r \cdot \cos\phi \sin\theta$$

$$y = y_c + r \cdot \sin\phi \sin\theta$$

$$z = z_c + r \cdot \cos\theta$$

$$\phi = \arctan 2\,(y - y_c, x - x_c)$$

$$\theta = \arccos\left(\frac{z - z_c}{r}\right)$$

2. Now we can easily get 2D surface coordinated for this polar coordinates by dividing by the spherical component π:

$$u = \frac{\phi + \pi}{2\pi}$$

$$v = \frac{\pi - \theta}{\pi}$$

Current graphics card allow for loading the computation code for rather than the texture coordinates itself to the card. These are called **vertex programs**.

Rasterization: Perspective Interpolation

A problem is that the texturing method is currently rasterization dependent and scanline interpolation will even distort our textures, since the used barycentric coordinates do not respect the distortion of the perspective transformation of the texture. The solution is pretty straightforward and the idea is already know from clipping in homogeneous coordinates: We do the interpolation in the perspective space.

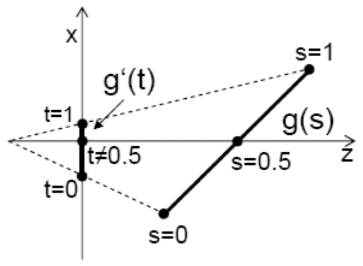


Figure 102: Perspective Interpolation

On the figure you can see above, the process is illustrated. s is the texture coordinate in world space, and t in screen space. As you can guess, the interpolation in both spaces is not the same, therefore we look for a mapping $t \to s$ that allows for a correct interpolation.

$$\begin{pmatrix} x \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ z_0 \end{pmatrix} + s \begin{pmatrix} x_1 - x_0 \\ z_1 - z_0 \end{pmatrix}$$

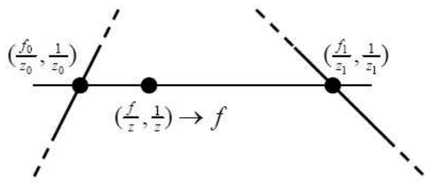

$(\frac{f_0}{z_0}, \frac{1}{z_0})$
$(\frac{f_1}{z_1}, \frac{1}{z_1})$
$(\frac{f}{z}, \frac{1}{z}) \to f$

Figure 103: So instead of the standard rasterization we now interpolate with the values returned by the mapping to the perspective space.

11.3 1D Texture Mapping

1D texture maps are often used for visualization. E.g. for scalar fields (color coding).

11.4 3D Texture Mapping

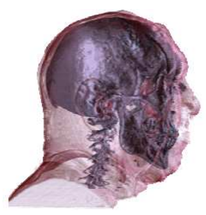


Figure 104: A 3D map to model the inside of a human head

Especially for visualization. For example medical modeling of organs or systems of the human body or for technical modeling of machines.With 3D maps volume effects can be obtained.

11.5 Texture Antialiasing

Often the object we want to wrap out texture around is larger or smaller than the texture. In this case we need a larger or smaller form of or texture gotten by expanding or shrinking it's resolution. However this easily leads to visual artifacts (called Aliasing). because the sampling theorem is hurt:

11.5.1 Sampling Theorem

Nyquist once stated an important theorem about the sampling of a signal. The sampling frequency f_s must be at least twice as high than the highest frequency occurring in the signal f_o, else the signal's representation will not be accurate:

$$f_s \geq 2 \cdot f_o$$

In our case that means the function's frequency sampling the texture must be twice as high as the texture's frequency.
Antialiasing methods and algorithms have been thought of to counter this effect or keep the sampling theorem valid.

11.5.2 Mip Mapping[9]

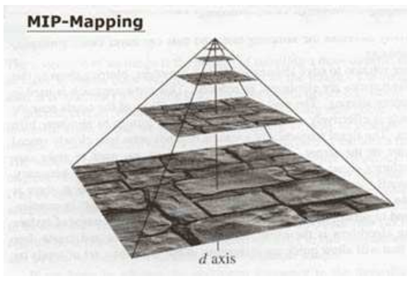


Figure 105: Mip-Mapping

The idea of Mip Mapping is to provide the same texture in several different resolutions (correspond to different frequencies). Than when it comes to sampling we choose the texture, whose frequency is most fitting the sampling frequency (e.g. by taking the highest absolute value of the following differentials measuring how much texels contribute to one pixel projected to the texture map: $\left\{\frac{\partial u}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial x}\right\}$ as a measure, for more selection methods, refer to the section on Level Of Detail Selection 10.7.3).

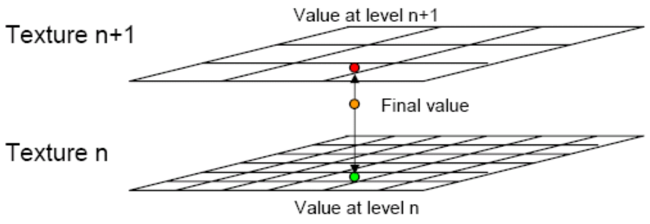


Figure 106: Trilinear Interpolation with Mip-Mapping

Mip Mapping also provides an interesting possibility for doing trilinear interpolation using 2D textures. In a first step we do bilinear interpolation between two succeeding textures of the Mip Mapping hierarchy and then linear interpolate between the two resulting values. The result is a three dimensional looking image.

+ takes a constant amount of time no matter the resolution

- only squared areas can be retrieved, this leads to **overblurring** of rectangular scenes, if they minimized/maximized too much

[9]MIP = multum in parvo (Latin: many things in a small place)

11.5.3 Ripmapping

The ripmapping technique tries to avoid the overblurring appearing with mip mapping. The idea is simple, we extend mip mapping as to include down sampled rectangular areas as subtextures that can be accessed. Two more parameters are used to access this rip map, but they can be computed on the fly by using the pixel cell's u and v extents on the texture.

+ no overblurring

- very memory intensive

11.5.4 Summed-Area Table

The "Table" is referring to an underlying array, having the size of the texture but more bits. Now on every position in the array all pixels included by the rectangle having lower left point $[0, 0]$ and the position as upper right point are summed, divided by their number and stored in this position. By that we can compute the average of any arbitrary rectangle within the texture (by simple subtractions).

+ less overblurring than mip mapping (only at the diagonals)

- memory intensive

11.6 Blending Functions

When we found the texture coordinates we want to have at a certain point on our object, we have several possibilities how to proceed further. These include

replace simply replace the underlying object point with the texture value

decal like replace, but apply α-blending

modulate multiply the surface color with the texture value (also called **multiplicative blending**)

In the first case already computed lighting will be overwritten and the object will appear to glow on its own account (**glow texture**).

11.7 Corresponder Functions

Corresponder functions describe what is to be done with pixels outside of the normalized texture range $[0; 1[$:

wrap repeat tile The texture image is repeated across its borders. For example the value at 1.2 equals the value at 0.2.

mirror

The texture image is repeated, but mirrored. For example the value at 1.2 equals the value at 0.8.

clamp

Values outside the range are "clamped" to the closest edge. There is an alternative called clamp to border, where a special border color is defined, where to cast outside values to.

11.8 Bump Maps

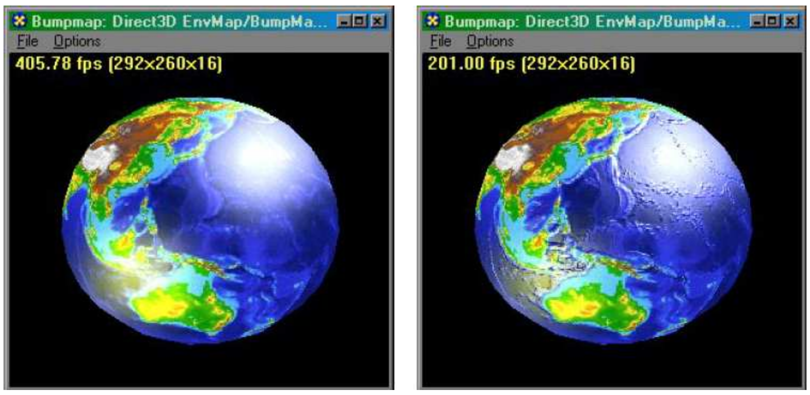


Figure 107: Bump Maps adding height features

As discussed at the beginning of the chapter there are additional features we might want to add to mere color wallpapers, to make them look more realistic. One approach to do this are bump maps. A bump map is an "image" which contains height information for a texture map. So at each point of the texture we know the depth/height of it.

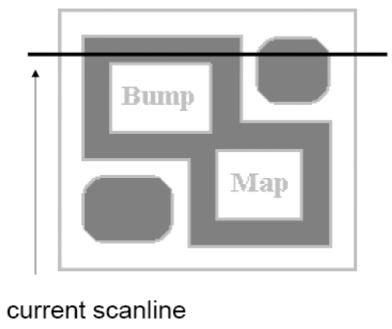


Figure 108: Bump Maps contain height information

We can make use of this knowledge by developing a way to alter the surface normal at this position according to the texel's height value. This does not result into an actual change of shape, yet due to shading being differently applied at this point, it looks like it was. This change in normals can be obtained by sampling the bump map and using partial derivatives (gradient) to express the changes in height and perturb the normals with them.

Emboss Bump Maps

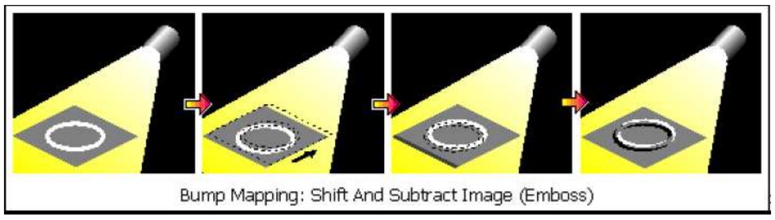


Figure 109: Emboss Bump Map

Emboss Bump Maps is an approximation to standard bump mapping, that is far more efficient, since it skips lighting calculation at each pixel. The idea is to render the bump map as an image, translate the texture towards the light source, render it again as a subtractive texture (see below for multi texturing 11.15):

$$L \cdot T(s) - T(s + \Delta s)$$

Gouraud Bump Maps

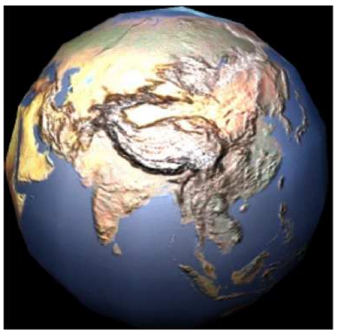


Figure 110: Gouraud Bump Map

In contrast to Emboss Bump Maps, Gouraud Bump Maps are a complification. Instead of changing the surface normals, the bump map changes lighting normal per vertex. It requires a high geometric resolution and is hardly useful.

Per-Pixel Bump Maps / Dot Product Bump Maps

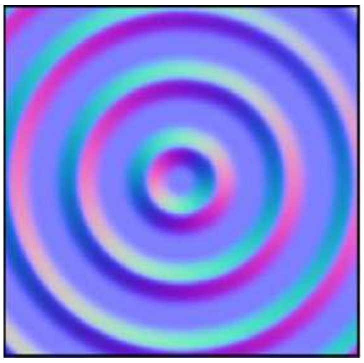


Figure 111: Normal Map used for bump mapping

Instead of height, the bump texture contains normals (x, y, z coded in RGB). So we read the normal from the texture $\vec{n}$, interpolate $\vec{v}$ and $\vec{l}$ and normalize them and finally compute lighting with $\vec{n}, \vec{l}, \vec{v}$. However in a first step we still need per vertex operations to map world space coordinates to the texture coordinates.

Parallax Bump Maps

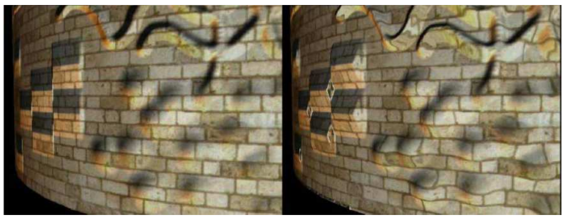


Figure 112: Left: standard bump map, Right: parallax bump map

Parallax is the apparent shift of an object against a background caused by a change in observer position. Standard bump mapping does not cover this visual effect, but they can be elegantly extended to provide this effect. We estimate the parallax due to the bump texture and apply the effect by adding a offset to texture coordinates.

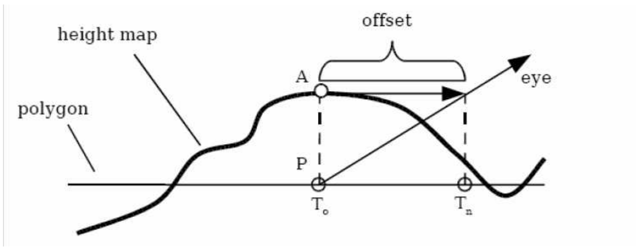


Figure 113: Computation of the parallax offset

T_0 the actual point the eye would see without bump mapping

A the point T_0 offseted according to the bump map

T_n corrected point

B what the eye would see, if the bump was real (by offsetting the corrected T_n)

$$T_n = T_0 + H \cdot \frac{e_x e_y}{e_z}$$

where H is the height according to the bump map.

Limit the offset for grazing angles

$$T_n = T_0 + H \cdot e_x e_y$$

T_n corresponds to the gradient and can be found by for example using Newton Iteration.

Properties

+ efficient: simple geometry stage,

+ visually complex

- no change in geometry: shadows are not affected by the bumps, silhouettes are unaffected

- looks still flat, when viewed from the side

11.9 Displacement Maps

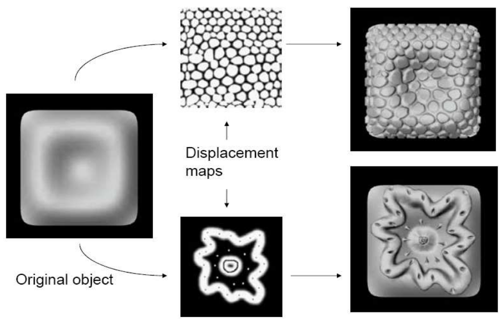


Figure 114: Displacement Maps

In contrast to bump maps, displacement maps really do change the geometry of objects. Surface points are displaced according to a displacement map commonly towards the surface normal.

11.10 Environment Maps

Environment Maps are textures that allow for a mirroring of the background. They can be implemented as cube maps (6 textures), sphere maps (1 texture) or paraboloid maps (2 textures).

Cube Maps

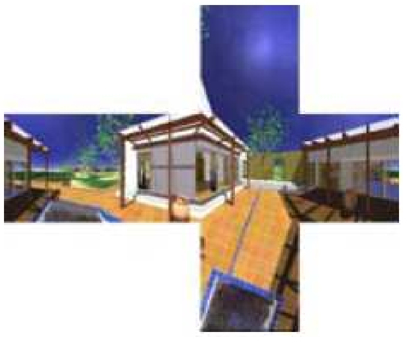


Figure 115: Environment Map with a cube

We think of a cube surrounding the whole scene having one texture on each side. We access the cube's textures by casting a ray from the center of the scene.

1. compute reflection ray $\vec{r}$ for the surface point (where the eye vector would be reflected to)
$$\vec{r} = 2\,(\vec{e} \cdot \vec{n})\,\vec{n} - \vec{e}$$

2. find the corresponding cube sub-texture: Choose the highest absolute[10] value among the three coordinates and determine the sub-texture by it's sign

   E.g. $\vec{r} = (-8, 2, 1)$ has highest absolute coordinate $|-8|$, the sign is $-$ resulting in the left sub-texture

3. get the texture coordinates (u, v) by the intersection of ray and sub-texture. This can easily be obtained by dividing the other two coordinates by the absolute value of the one chosen in 2. and scaling the unit cube range $[-1; 1]$ to the texture range $[0; 1]$ by adding 1 and dividing by 2.

   E.g. $\vec{r} = (-8, 2, 1)$ results in $\frac{2}{|-8|}$ and $\frac{1}{|-8|}$ and after scaling the range: $u = \frac{\frac{2}{|-8|}+1}{2}$ and $v = \frac{\frac{1}{|-8|}+1}{2}$

+ uniform sampling characteristics (no excessive number of pixels at a pole, like spherical ones)

+ the six faces are easy to compute

+ view independent

[10]If two coordinates have equal absolute values, we are on a border between two cube maps and can choose any of them. This however will seldom happen, because of hardware precision failures. However we can account for it, by putting these border lines into both neighbouring textures.

Spherical Maps

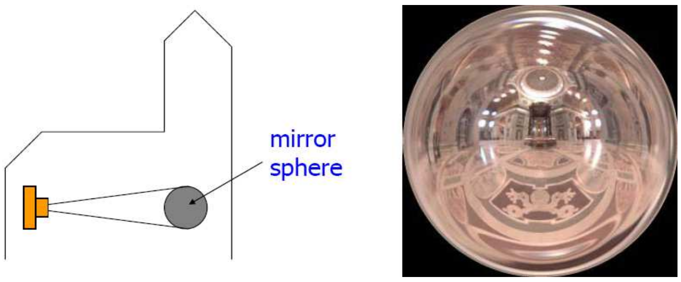


Figure 116: Environment Map on a sphere

We make some assumptions to make this process more efficient:

- parallel camera rays (uniform direction e_0)
- environment map is infinitely far away (color depends only on the direction of reflection $\vec{r}$)
$$\vec{r} = 2\,(\vec{e}\cdot\vec{n})\,\vec{n} - \vec{e}$$

With these assumptions made, our environment map odes only need to store one color value for every direction of reflection. After $\vec{r}$ has been normalized, texture coordinates can be gotten by

$$u = \frac{r_x + 1}{2}$$

$$v = \frac{r_y + 1}{2}$$

The sphere texture image is created/recorded by placing a perfectly mirroring sphere in the middle of the scene and save the reflection (also called “probe”, see image above).

`OpenGL glTexGenfv(GL_S,GL_SPHERE_MAP,0)`

+ no seam at the border of the texture

- irregular sampling at the boundary, because many pixels are mapped closely on the sphere’s poles (use other environment maps, e.g. cube maps or parabolic maps)

- moving between two points is not linear (no linear interpolation possible)

- only valid for one viewing direction (no environment rotation)

Parabolic Maps

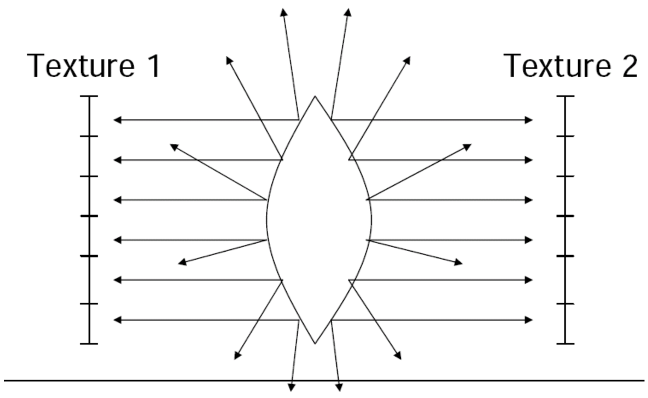


Figure 117: Parabolic Environment Map

Parabolic Maps are very similar to spherical environment maps, yet they use two textures and mirror the environment at two paraboloids rather than a perfect sphere. All reflection rays share the same origin and viewing rays are parallel to the z-axis. The paraboloid is given by:

$$f(x, y) = \frac{1}{2} - \frac{1}{2}\left(x^2 + y^2\right)$$

The we get the texture coordinates from the reflection vector:

$$u = \frac{r_x}{1 + r_z}$$
$$v = \frac{r_y}{1 + r_z}$$

This works especially well if the hardware allows to reflection vector to texture coordinates.

+ view independent

+ uniform sampling (linear interpolation) even better as with cube maps

- very hard to create

Properties of Environment Maps

+ supported by hardware

- for planar objects (flat objects) the color becomes unrealistically constant (worst for orthographic projection)

- the color of a point on the reflecting surface does not only depend on the reflection vector $\vec{r}$ but rather on the area of a cone having it's peak in the point (use **prefiltered environment maps** accounting for this).

11.11 Environment Bump Maps

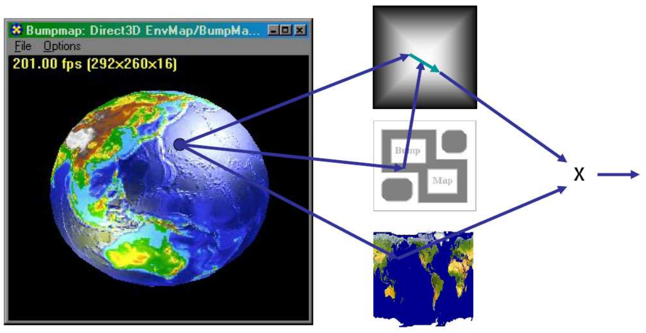


Figure 118: Environment Bump Map with normal texture, environment map (+light source) and a bump map

There exists an interesting combination of bump and environment maps, that will be presented here. The first aspect is that we will perform lighting via a texture:

lighting via texture

a 2D texture that maps surface normals $\vec{n}$ to color $L_{\vec{n}}$
Thus we result in having three texture maps: standard texture, environment map + light source, bump map (see picture). The bump map returns an offset for accessing the environment map (changing the normal):

$$L = L_{\vec{n}} \cdot \text{Brightness}\,(\text{environment}) + \text{bump offset}$$

11.12 Interactive Horizon Maps

The disadvantage of bump maps is that although the bumps look good, they provide no real geometrical bumps and therefore those bumps do not cast shadows. Horizon Maps try to counter this by storing the horizon around a point in texture maps, enabling to decide whether a point lies in shadow or not. The height of the horizon simply depends on the direction (i.e. the angle), so the direction will give us access to the map. If the light source lies below the horizon we are in shadow.

Following this idea we precompute horizon heights for each pixel in at least eight directions (N, NE, E, SE, S, SW, W, NW). Then during lighting calculation we compute the height angle of the light source and check whether the height of the horizon surpasses the light source's height.

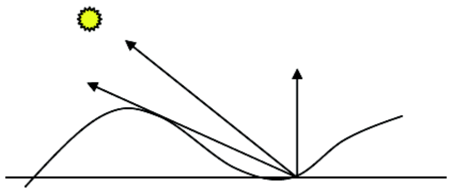


Figure 119: Does the light ray lie above the horizon?

By this eight directions we sample the horizon enclosing our point in 8 points. Now having a point we could simply interpolate the two involved samples, however the results are very bad. Therefore we rather use the samples as coefficients of basis functions (stored in textures, one for each direction) and use them to evaluate the height in between two directions with weighted interpolation (coefficients = weights).
Since the horizon height samples are 1D float values, we will be able to store them in merely two textures, storing four samples in one texture's RGBα-channels.

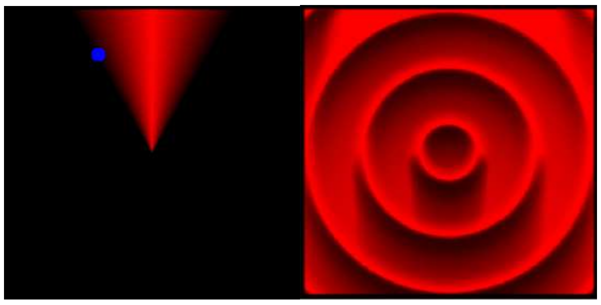


Figure 120: North basis function texture and the resulting horizon map using only this basis texture
In the figure on the left we see the basis function for the direction north. If we have a point and access a direction influenced by the north basis function, we will result in a horizon map like the figure on the right. The bright circles are totally lit and not occluded by horizons in the north. The closer we go north from such a circle the closer we will get to a northern horizon and the more we will be in shadow.
For example we assume the position to be the blue point on the left figure. Then the horizon height will depend on something about 20% on the northern horizon sample point and about 80% on the north western one. All other samples contribute 0% to the interpolated height.

11.13 Shadow Maps

The idea of shadow maps is to store a map, which we can access, if we want to know whether a point is lit or not. This can be done by rendering the scene with the eye at the light source, then naturally every position not lit, lies in shadow, since the light cannot reach it. We then store this result in a picture we can use as a shadow map. These kind of texture maps are excessively discussed in the chapter about shadows on page 8.5.3.

11.14 Illumination In Textures

Another different way of making use of textures is to use them for lighting calculation. This of course is limited to scenes where the light remains the same from any angle, (e.g. a chamber with one light source on the ceiling). Using textures we can efficiently realize the Torrance-Sparrow Light Model (see 8.3.3). Remember the color was given by

$$L = I_{\text{In}} \cdot \frac{F\left(\vec{l} \cdot \vec{h}\right) G\left(\vec{n} \cdot \vec{v}, \vec{n} \cdot \vec{l}\right) D\left(\vec{n} \cdot \vec{h}\right)}{\Pi\left(\vec{n} \cdot \vec{v}\right)\left(\vec{n} \cdot \vec{l}\right)}$$

we reorder the formulated

$$L = F\left(\vec{l} \cdot \vec{h}\right) D\left(\vec{n} \cdot \vec{h}\right) \frac{G\left(\vec{n} \cdot \vec{v}, \vec{n} \cdot \vec{l}\right)}{\Pi\left(\vec{n} \cdot \vec{v}\right)\left(\vec{n} \cdot \vec{l}\right)} \cdot I_{\text{In}}$$

and then store the functions $F\left(\vec{l} \cdot \vec{h}\right) D\left(\vec{n} \cdot \vec{h}\right)$ in a first texture with $u = \vec{l} \cdot \vec{h}$ and $v = \vec{n} \cdot \vec{h}$ as texture coordinates and $\frac{G\left(\vec{n} \cdot \vec{v}, \vec{n} \cdot \vec{l}\right)}{\Pi\left(\vec{n} \cdot \vec{v}\right)\left(\vec{n} \cdot \vec{l}\right)}$ in a second one (for color), where we use $s = \vec{n} \cdot \vec{v}$ and $t = \vec{n} \cdot \vec{l}$ as texture coordinates.

Algorithm 17 Illumination In Textures

```
1. set vertex color to $I_{\text{In}}$
2. turn on texture #1
3. use $u, v$ as texture coordinates
4. render the scene
5. set vertex color to 1 (white)
6. turn on texture #2
7. use $s, t$ as texture coordinates
8. render the scene with multiplicative blending (multi texturing, see above
   11.6)
```

Due to the resulting gain in efficiency Phong Shading can be used.

Optimization

If we simplify the physical model again by assuming parallel light, $\vec{l}$ becomes constant. If we further assume a parallel viewer $\vec{v}$ becomes constant. Assuming both even $\vec{h}$ becomes constant. If light and viewer are indeed far from the object this assumption is justified.

In this case we use simply the normal $\vec{n}$ as texture coordinates and generate u, v, s, t automatically by having stored $\vec{l}, \vec{v}$ in a texture matrix. This further allows us to use `OpenGL` display lists (see 14).

If we are still not content we can even store the diffuse reflection $\left(1 - \vec{f}\right)$ in the texture. The α-channels of both textures are not used yet, so we can store the diffuse reflection in the α-channel of the first texture. In this case however, we must add a third render pass at the end of the algorithm for diffuse illumination and blending:

$$L = \alpha\,(\text{destination}) \cdot L\,(\text{source}) + 1 \cdot \text{destination}\,(\text{source})$$

where $\alpha\,(\text{destination})$ corresponds to $1 - \vec{f}$, $L\,(\text{source}) + 1$ corresponds to the diffuse fraction and destination (source)to the specular fraction.

+ if changes in $\vec{l}, \vec{v}$ are required from time to time, the optimization can still be used, simply the texture matrix has to be recomputed

- two/three passes required

11.15 Multi Texturing

Multi Texturing describes a method to apply multiple textures in a single rendering pass for one object. Today's graphics hardware supports this. It defines operations to add, subtract, multiply, etc. textures with/from each other. The advantage against simply applying multiplicative blending (see 11.6) is that instead of rendering multiple texture one after another, we get the textures rendered in one single pass.

Note To avoid visual quantization artifacts, choose an appropriate color model (24bit, 32bit)

+ supported by modern boards

+ only one rendering required

11.16 Texture Cache

A scene might contain a high number of textures, which are consequently accessed. Therefore most graphics hardware offers a cache for textures. Usually the textures should be kept small. An exception is to combine small textures in a mosaic like pattern on a larger textures. In this case we have implicit smaller textures, but save the overhead for switching textures.

Last Recently Used (LRU)

Now to make good use of the texture cache, each texture is assigned a time stamp. Every time a texture is called, it gets a new time stamp assigned. If the cache is full the texture with the oldest time stamp will be dropped. In case of a draw OpenGL and DirectX offer additional priority assignments.

Most Recently Used (MRU)

MRU checks the texture currently being swapped out of the texture cache, whether it has been used in the current frame. If it was, it is kept. While being in one frame MRU should be preferred to LRU, since otherwise every single texture of the frame would first be swapped in. Leaving a frame, we switch back to LRU.

Prefetching

As the name suggests prefetching loads the textures into the texture cache, before they are needed or required. By that a lot of latency can be hidden. Of course this technique requires a good precomputation of which textures are required at a future time.

11.17 Texture Compression

Textures are images and images can be compressed by e.g. JPEG or PNG compression. Now this would allow a faster loading and a better usage of the texture cache, however the decoding algorithms for JPEG and PNG are to complex to put them in hardware. Therefore SGI has created a special texture image compression format that is especially easy to decompose: **S3TC** (S3 Texture Compression). The main disadvantage is that this format is lossy, i.e. it cannot be recreated without information loss. If a texture image shows especial color depth at a certain region, this will be lost. Furthermore S3TC should never be used when dealing with normal maps used for bump maps.

12 BRDF (Bidirectional Reflection Distribution Function)

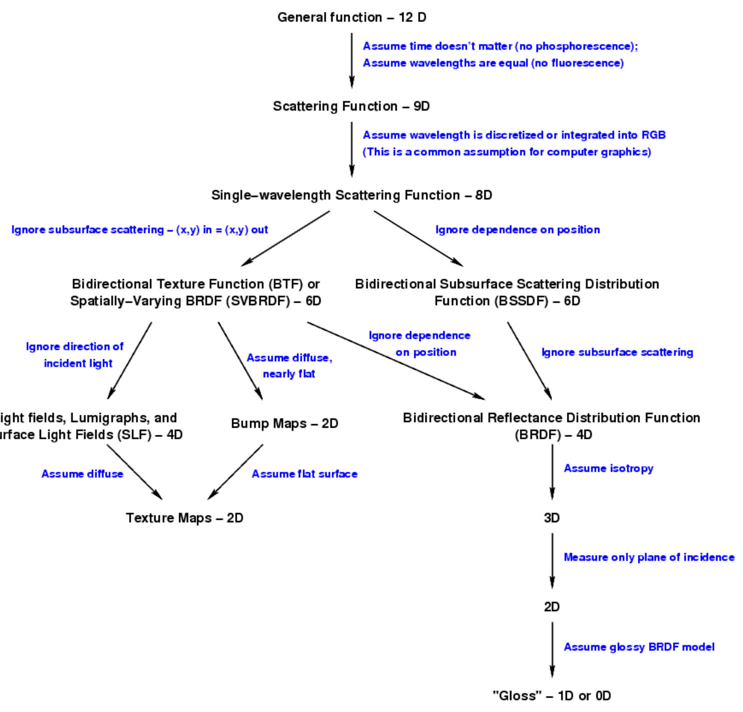

General function – 12 D
Assume time doesn't matter (no phosphorescence); Assume wavelengths are equal (no fluorescence)
Scattering Function – 9D
Assume wavelength is discretized or integrated into RGB (This is a common assumption for computer graphics)
Single–wavelength Scattering Function – 8D
Ignore subsurface scattering – (x,y) in = (x,y) out
Ignore dependence on position
Bidirectional Texture Function (BTF) or Spatially–Varying BRDF (SVBRDF) – 6D
Bidirectional Subsurface Scattering Distribution Function (BSSDF) – 6D
Ignore direction of incident light
Assume diffuse, nearly flat
Ignore dependence on position
Ignore subsurface scattering
Light fields, Lumigraphs, and Surface Light Fields (SLF) – 4D
Bump Maps – 2D
Bidirectional Reflectance Distribution Function (BRDF) – 4D
Assume diffuse
Assume flat surface
Assume isotropy
Texture Maps – 2D
3D
Measure only plane of incidence
2D
Assume glossy BRDF model
"Gloss" – 1D or 0D

Figure 121: From physical radiance to BRDFs and other lighting/shading methods

12.1 Maxims

- plausible (obey energy conservation, reciprocity)
- anisotropy
- intuitive parameters (like in Phong Lighting)
- Fresnel behaviour (for peculiarity)
- non-Lambertian diffuse term (for a diffuse term with energy conservation for the Fresnel term)
- Monte Carlo support (to support Ray Tracing)

A BRDF which manages to fit all these maxims is called: **Fresnel-weighted Phong-style anisotropic cosine lobe model**.

12.2 Theory

Bidirectional Reflection Distribution Functions describe how light is reflected from a surface. To describe this a BRDF covers:

- material properties
- incoming/outgoing azimuth and elevation angles
- incoming light's wavelength
- surface area

You can see BRDFs as giving the probability that an incoming photon will leave in a particular direction. So they relate incoming and outgoing radiance, but they do not describe physical material light interactions. It makes another simplification by neglecting scattering of light within a surface, and only takes into account light coming from above and being reflected at one specific point (A function type modeling surface scattering are **Bidirectional Surface Scattering Reflectance Distribution Function** (**BSSRDF**) which will not be discussed).

The outgoing radiance for a given point x and light L_{in} incoming at angle ω_{in} is

$$L(x, \omega_{out}) = \int_{\Omega} f(x, \omega_{in}, \omega_{out}) \, L_{in}(x, \omega_{in}) \, (\omega_{in} \cdot \vec{n}_x) \, d\omega_{in}$$

where f is the BRDF. It returns for some incoming light direction ω_{in} what percentage of light leaves at some exitant direction ω_{out}. The second term denotes the radiance arriving at point x from direction ω_{in}. The last term is just the application of Lambert's cosine law for diffuse surfaces: $\cos(\omega_{in}) = (\omega_{in} \cdot \vec{n}_x)$. Since we are interested only in the light that will turn out on point x we integrate over all incoming and outgoing light angles $\int_{\Omega}$.

If the surface is diffuse, the BRDF f becomes constant ($f(x, \omega_{in}, \omega_{out}) = \rho(x)$, with $\rho(x) \in]0;1[$ where 0 means perfect reflectance and 1 no reflectance)

$$L(x, \omega_{out}) = \rho(x) \int_{\Omega} L_{in}(x, \omega_{in}) \cos(\omega_{in}) \, d\omega_{in}$$

If we are dealing with a single point lightsource the equation further simplifies to

$$L(x, \omega_{out}) = \rho(x) \, L_{in}(x, \omega_{in}) \cos(\omega_{in})$$

where $\cos(\omega_{in})$ can be computed by the surface normal at the corresponding point:$\cos(\omega_{in}) = (\omega_{in} \cdot \vec{n}_x)$
Having more than one light source we discretize the integral to a sum and sum up over all light sources.
If we for example take the Phong Light Model, (see 8.3.2) the BRDF f becomes

$$f_{\text{phong}}(x, \omega_{in}, \omega_{out}) = \frac{k_s \left(\vec{n} \cdot \vec{h}\right)^{n_{\text{shiney}}}}{\vec{n} \cdot \omega_{in}}$$

in this way other models like the Torrance-Sparrow Light Model can be used.

Microfacets

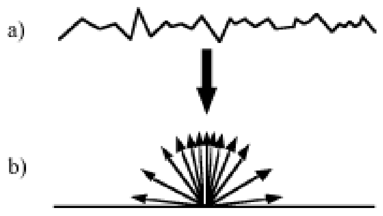


Figure 122: Surface microfacets.
a) The surface is assumed to be made of millions of tiny facets. The facets are used to find a probability distribution of facet normal directions.
b) The surface is rendered as a geometrically flat surface with the normal distribution used to reproduce the shading effects of the facets.

Surfaces will seldom be nice and flat, in fact even those which look flat have a microscopic rough structure. We introduce the concept of microfacets which model this micro structure and thus describe how surfaces behave. Microfacets are tiny mirrors on the surface with random size and angle (see figure). Instead of random (uniform), a Gaussian distribution of sizes and angles is assumed, because they are better to work with.

Microfacets cover:

- specular reflection (by direct reflection)
- diffuse reflection (inter reflection or scattering)
- self shadowing (facets shadow each other)
- refraction (use Fresnel Reflectance for dielectrics F[11])

Properties

- BRDF do not cover anisotropy[12]

  They could if we would add a second type of angle (ϕ_{in}, ϕ_{out}) to the BRDF. However this can hardly be covered with graphics hardware and methods.

12.3 Praxis (Implementation)

A first idea is to evaluate BRDF on every vertex in the scene. However as usual this leads to Gouraud artifacts, when changes are either smeared away or overemphasized. As discussed earlier this can be countered by fine structuring, however then we have a bottleneck and lack performance (using vertex shaders, the performance goes slightly up).

[11] F describes the reflectance of a surface at various angles
[12] the property of being directionally dependent

A second common idea is to precompute as much as possible and store it in a texture map. For isotropic surfaces the BRDF needs three variables, so we would be able to store everything in a three dimensional texture map. Again the usual lacks of this method are sampling problems, noise, gaps and memory-intensiveness.

12.3.1 Factorization

A more sophisticated implementation uses factorizations of the BRDF basis functions into a sum of two term products. The idea is to factorize the four dimensional (four variables) BRDF into two texture maps. Then we multiply the two values from both maps and sum them up:

$$f\left(x, \omega_{in}, \omega_{out}\right) \approx \sum_{j=1}^{n} p_j\left(\omega_{in}\right) \cdot q_j\left(\omega_{out}\right)$$

where p and q denote access functions to the two texture maps. Looking at the term closely, we see that factorizations tries to separate the BRDF into a function covering the incoming light and one covering the exitant light. The texture maps itself are accessed like environment maps (cube, parabolic or sphere (best)), which cover a similar task (reflection).

Properties

- rendering artifacts from using texture maps and interpolation (minor)
- two texture accesses for every light source
- limited to point and directional light sources

12.3.2 Environment Map Filtering

Environment Map Filtering extends the environment map concept (see 11.10) from mirror like reflection to glossy and diffuse reflection. The idea is to filter the result of the environment map access. For example by blurring it the specular reflection will appear rougher.

Now either we hope for the forgivingness of the eye and blur the whole map uniformly/linearly or we use a equation called **Phong Specular Equation** to filter it non-linearly. This equation determines a weight for each light direction depending how much every texel contributes relative to the direction. So the light color is given by the ambient light and the diffuse light resulting from an environment map covering the radiance of an environment (light + reflected lights with contributions falling off according to Lambert's Cosine Law, 8.3.2). This kind of environment map is also called **Irradiance Map**.

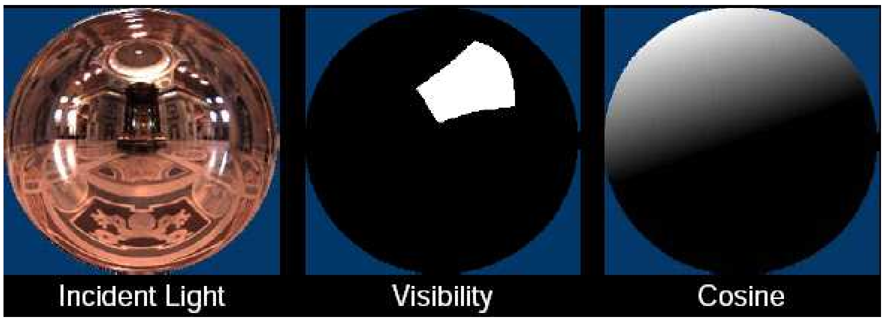


Figure 123: The three major components of PRT
The first sphere is the environment map covering lighting $L_{in}(p, s)$
The second sphere covers visibility (shadows) $V(p, s)$
The third sphere covers Lambert's cosine law for reflection $\cos(s) = (s \cdot \vec{n})$

Properties

- wrongly assumes the same specular lobe for all viewing/surface directions yielding the same reflection direction (this is only valid for perfect mirrors)

  This assumption is necessary to be able to restrict to one single environment map. Accordingly this problem can be covered by using multiple Environment Maps. In fact interpolation and blending between 20 sphere maps already draw high quality results.

- view dependency: by storing every information in a single sphere map, we also have view dependent specular reflection stored

  use two sphere maps instead and use one for view-dependent and one for view-independent radiance information

- environment maps assume light sources and objects to be distant

- the dynamic range of light is limited to 8 bits per color channel, yet direct lighting from a light source is hundreds of times brighter than indirect illumination. So 8 bits do not suffice to cover the full range of incident illumination (as it is needed in the environment map).

12.4 Precomputed Radiance Transfer (PRT)

Precomputed Radiance Transfer is a global illumination technique covering BRDFs (with precomputed environment maps), soft shadows and inter reflection. The key feature of PRT are spherical harmonics that make up the environment map. Advantages in comparison with previous BRDF techniques covered so far are:

- PRT is fast and simple (can be done in a vertex shader)

- interactive: the environment can be changed dynamically

Furthermore PRT allows for arbitrary illumination (direct, indirect, caustics) and for any kind of light transportation. Because of the environment map covering lighting, illumination can come from any direction. However objects need to be static (environment map) and interactions between objects is very limited (e.g. color bleeding).

Spherical Harmonics

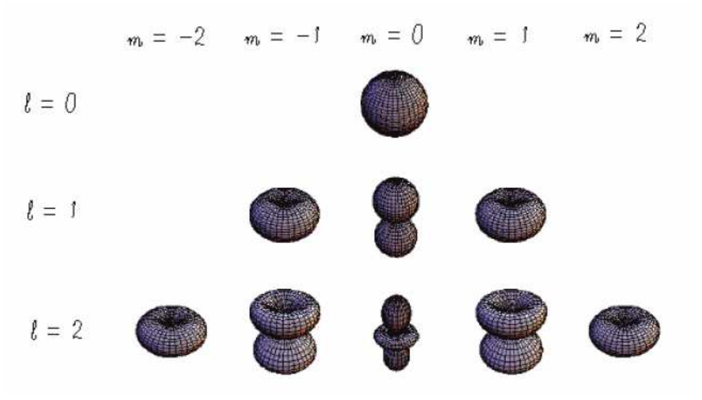


Figure 124: Spherical Harmonics
l is called band and m is bound by $-l \leq m \leq l$

Spherical Harmonics are a set of basis functions with a spherical domain. They can be used to represent spherical functions with a set of coefficients:

$$f(\theta, \phi) = \sum_{i=1}^{n} f_i Y_i(\theta, \phi)$$

where f is a spherical function, angles (θ, ϕ) parametrize the spherical domain and Y_i are complex basis functions with coefficients f_i.

$$Y_l^m(\theta, \phi) = K_l^m e^{\Im(\phi)} P_l^{|m|} \cos(\theta)$$

P_l^m are Legendre Polynomials (see 2.7) and normalization coefficients K_l^m are

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi \cdot (l-|m|)!}}$$

The higher the number of basis functions n, the more precise the outcome (typical are about 25).

Now we want a real values basis function and thus differentiate:

$$y_l^m = \begin{cases} \sqrt{2}\Re\left(Y_l^l\right) & m > 0 \\ \sqrt{2}\Im\left(Y_l^l\right) & m < 0 \\ Y_l^0 & m = 0 \end{cases}$$

Since y_l^m build an orthonormal basis, we can easily find the coefficients f_i by using the function scalar product

$$f \circ g = \int_0^1 f(x)\, g(x)\, dx$$

$$f_i = f_l^m \circ y_l^m$$

Computation: Incident/Exitant Light

So what we do is to approximate the lighting function with a polynomial using Spherical Harmonics as basis functions and the environment map as coefficients. We compute the incident light L_{in} as

$$L_{in} = \frac{1}{n} \sum_{j=1}^{n} L_{in}^{j}(\theta)\, y_j(\phi)$$

where (θ, ϕ) are used as indices for accessing the environment map. The environment is split up like in environment map filtering. Where $L_{in}^{j}$is a BRDF. Often L_{in} is constant for a whole object. If it isn't we can take some samples of the object and interpolate them.

Next we check the exitant light L_{out} for every vertex p and texture map access $s = (\theta, \phi)$:

$$L_{out}(p) = \rho(p) \frac{1}{n} \sum_{j=1}^{n} L_{in}^{j}(p, s)\, H_j(p, s)$$

where $H(p, s) = s{\cdot}\vec{n}_p$ transformed to the spherical domain: $H(p, s) = \sum_{i=1}^{n} H_i y_i(p, s)$

Shadows

Having this formula we can easily include shadow by an additional term $V(p, s)$

$$L_{out}(p) = \rho(p) \int_{\Omega} L_{in}(p, s)\, V(p, s)\, (s \cdot \vec{n}_p)\, ds$$

where $V(p, s) = 1$ if the point p sees the environment in direction s, and 0 if not.

Inter Reflections

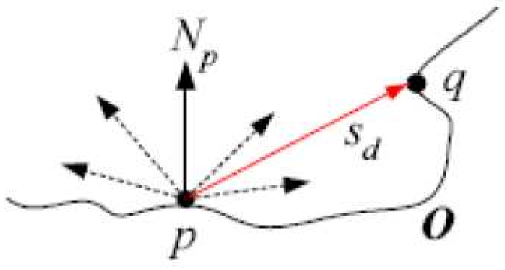


Figure 125: PRT with inter reflections
From the point p direction s does not return environment map texels, therefor we use the exitant light of q

In case p does not see the environment in direction s ($V(p, s) = 0$), it might see it's own surface point q (see figure). In this case we use the exitant light of q to model the interreflection between those two points. We can compute this value by assuming $L_{in}$to be the same at p and q. Then we just apply a global illumination methods like ray tracing and compute the diffuse color for p.

Shadows combined with inter reflections result in having soft shadows!

Properties

+ extension: translucent objects

+ extension: all-frequency lighting (wavelet basis instead of spherical harmonics)

- glossy reflection: exitant radiance depends on viewing direction

- the more specular the surface, the more basis functions are needed

- very bad for **high frequency light**: High frequency light is an euphemism and has nothing to do with wavelength. Every real lightsource lites the area perpendicular and closest to it more than the surrounding one building a light disc of intensity on it. This disc is brightest in the center. Transforming this disc into a frequency diagram we will encounter a peak at the center. If this peak if very high, we talk about high frequency light, if it's low and broad about low frequency light. For example a point light source will have a Dirac peak/impulse. Now to model such a peak we'd need spherical harmonics or wavelets of a very high degree. Apart from that that "a very high degree" per se is a problem, spherical harmonics of a high degree are very similar and show almost no differences.

Alternative Computation

An alternative algorithm that is faster, because the BRDF is computed with L_{in} in SH basis

1. transform incident light L_{in} to transferred light L'_{in} ignoring the object at p

   this gives the local incident light for p, as well as self shadowing and inter reflection (L_{in}, L'_{in} in SH basis)

2. now apply BRDF for every vertex using L'_{in}

13 Rendering Pipeline

A scene is described by geometry, material properties, viewing and lighting. But the question is in what order should or rather can we perform these steps and convert the 3D scene description to a 2D raster image.

Pipeline 1 (Single Stages)

Step	Actions	Variables	Coordinates
Application	interactivity, collision detection	pixel/color	screen coordinates
Model Transforms	translation, rotation, ...	vertices/normals	model coordinates: (u, v, w)
World Transforms	translation, rotation, ...	vertices/normals	world coordinates: (x, y, z)
Viewing Transforms	perspective projection	vertices/normals	viewing coordinates: (e, g, t)
Illumination	lighting	vertices/color	world coordinates
Projection	normalizing transforms	vertices/color	normalized coordinates
Window Mapping	?	?	?
Clipping	Z-Buffer	fragments, depth/color	normalized device coordinates
Rasterization	shading	pixel/color	screen coordinates
Texturing	texture mapping	texel	texture coordinates
Framebuffer	window to view port	pixel/color	screen coordinates

Pipeline 2 (Culling/Clipping)

backface culling $\rightarrow$ modeling transforms$\rightarrow$ clipping $\rightarrow$ homogeneous divide $\rightarrow$ shading, lighting $\rightarrow$ rasterization

Pipeline 3 (Vertex, Primitives, Fragments)

1. Application: Custom Operations

   (a) collision Detection

   (b) Interactivity (e.g. drag & drop)

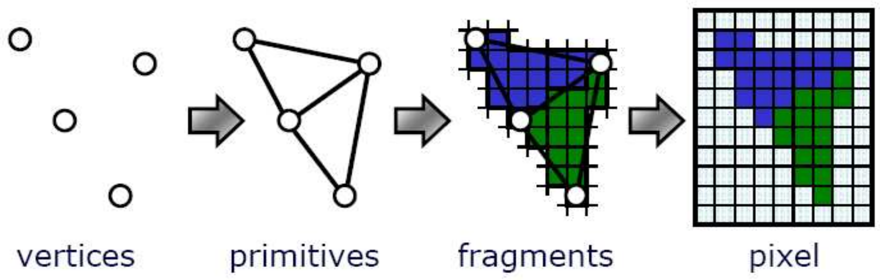


Figure 126: From vertices to fragments

2. Geometry: Vertex Operations
    (a) Affine Transformations (transformation matrices)
    (b) Illumination (local Illumination at the vertices)
    (c) Primitive Assembly (lines, triangles)
    (d) Projection (Normalizing Transform, Unit Cube, Z-Values)
3. Rasterization: Operations on Primitives
    (a) Polygon Rasterization (decompose primitives to pixel fragments)
    (b) Shading (with the fragments)
    (c) Texture Generation (Interpolation of texture coordinates / texture values)
    (d) Texture Mapping (Projection unto the object)
4. Fragment Operations: Operations of Fragments and Pixels
    (a) α Test (reject fragments above a certain α-value)
    (b) Stencil Test (reject fragments with stencil buffer enabled)
    (c) Depth Test (reject fragments where the depth test fails)
    (d) α Blending (combine values of color fragments)
    (e) Fog (a fragment is blended with a fog color)

Runtime Considerations

The speed of a single data packet is determined by the sum of all stages on the Pipeline, but the overall throughput is determined by the slowest stage, referred to as **bottle neck** (e.g. if there are two stages under 2 minutes and one requiring 3 minutes for assembling a car, one car can be completed every 3 minutes). The event if the whole process is stand to wait for a certain stage, is called **stalling**. Optimizations include

Sequentiation Partition the the bottle neck into two sequential stages

Parallelization Insert parallel pipelines at vertex and pixel operations step

Sorting Sort polygons by material (render $1 \dots X$ with the same texture, much faster than per triangle)

Potential bottle necks include

Application data generation, data transfer

$\rightarrow$ this stage is done in software, so optimize the code. a good code here can also fasten the next two stages. furthermore you might be able to make use of parallel processors.

Geometry lighting computation, number of light sources, number of triangles, complex per vertex computations

Rasterization degree of occlusion (e.g. leaves on a tree), mutlitexturing, complex per pixel computation

14 OpenGL

OpenGL is a hardware independent version of GL (Graphics Library) from Silicon Graphics. A review board out of consortium of graphics companies is maintaining the language. OpenGL is specialized but not limited to 3D scenery.

Properties

- hardware abstraction: API (application programmer interface)
- low level hardware optimized
- hardware independent
- boundless extensions
- high level modeling: scene graphs
- window system interfaces: GLUT, GLX, AGL, WGL

Syntax

functions `gl`-Prefix: `glClear`, `glPolygonMode`

constants in CAPITAL LETTERS: `GL_POLYGON`, `GL_RGB`

datatypes `GL`-Prefix: `GLbyte`, `GLdouble`, `GLfloat`

Libraries

- GLU:

  prefix glu

  content advanced routines, B-splines, complex objects

- OpenInventor

  content object oriented toolkit, scene graphs

- Graphics Display: GLUT (`glut`) OS independent, GLX (`glx`) X-Window System, AGL (`agl`) Apple, WGL (`wgl`) Windows

Matrix Stacks

Since the matrices needed for transformations will be used more than one time, we should store them in a kind of stack `push(Matrix m)` and re-access them we needed `pop()`.
With this stack we can easily include a rendering command.

```
Group::render()
push(Matrix transformation);
forall children c : c.render();
pop();
```

`push matrix` duplicate current matrix $m \rightarrow m'$. apply m' to the matrix on top of the stack $\bar{m} \rightarrow m*$.

$m*$	$\bar{m}$	$\cdots$	$\cdots$

We have a matrix history: the last step is always multiplied with the newest to create the stack entry.

`pop matrix` remove the top matrix from the stack.

OpenGL differentiates between two different types of matrix each having it's own matrix stack:

`GL_PROJECTION` normalization (`gluPerspective()`)

`GL_MODELVIEW` camera, modeling (`gluLookAt(), glTranslate(), glScale(), glRotate(),...`)

By the command `glMatrixMode()` we may choose on which kind of matrix we currently want to work.

`GL_TEXTURE` In fact there is a third kind of matrix assigned a third separate stack of it's own. The texture matrix. This matrix is used for projecting textures onto objects. It can be defined by: `glTexCoords4f(s, t, r, q)`

Output Primitives

The definition of primitives always starts with `glBegin(Primitive Type)` and end with `glEnd()`. Single vertex coordinates can be set by vertex position (`glVertex(position)`). Primitives are:
points (`GL_POINTS`), straight lines (`GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP`), circles, other conic sections, quadratic surfaces, spline curves and surfaces, polygon color areas, character strings.
Additionally size and color can be set.

Display List

Display Lists are rendering macros (e.g. for rendering a chair) that are directly loaded unto the graphics card and can easily be recalled multiple times. This gives a speed improvement by a factor of 10.

```
glNewList(1, glCompile); ... glEndList(); ... glCallList(1);
```

Cartesian Reference Frame

The coordinate frame for screen display can be set by `gluOrtho2D(xmin, xmax, ymin, ymax);`

15 Programmable Graphics Hardware

Configuration matrices, lighting parameters (Phong), texture maps

Problems portability, short innovation cycle, vendor dependent, too many extensions (OpenGL)

Unification Upload per-vertex and per-pixel code directly to the graphics device. Vertex and Pixel stages on the pipeline are replaced by programmable units.

+ vendor independent

+ effect libraries

Having several fragments per pixels, which can even be shared by multiple pixels, we can do antialiasing. If we'd simply say fragment = pixel, we would encounter ugly aliasing effects.

Pixel average value of multiple fragments that is displayed.

Fragment one sample of a high resolution image.

Antialiasing By rendering the image at a higher resolution and scaling it down.

Vertex Unit (Vertex Shader)

The vertex unit deals with all per-vertex operations like transformations and lighting per vertex (see 13, Pipeline 3). Per-Vertex operations are required every then, when data is changing slowly enough to only change the vertices (and risk minor artifacts). A vertex program is built out of three components:

Input `glVertex(), glNormal(), glColor(), glTexCoord()`

Parameters constants like light direction $\vec{l}$, material

Output normalized 3D screen position, color, secondary colors (glossy), texture coordinates, ...

Currently a creation of new vertices or removal of existing ones is not possible (added in 2006/2007, XBox 360°). There is no such thing as an early return statement or flow control statements[13], that means the hardware will need exactly the same time for each vertex. Also, small vertex programs run faster. Vertex shaders can be used for

- shadow volume creation (see 8.5.4)
- lens effects (e.g. underwater)
- object definition (making a mesh only once)
- object twist, bend, taper operations
- procedural deformations (flag, cloth movement)
- primitive creation (send degenerated meshes)
- page curls, heat haze, water ripples

Fragment Unit (Fragment Shader, Pixel Shader)

The fragment unit deals with per-fragment or per-pixel operations like texturing or phong-shading (see 13, Pipeline 3). Per-Pixel/Fragment operations are needed to accurately capture rapidly varying changes. A fragment program is also built of three components:

Input interpolated color, interpolated depth, interpolated texture coordinates, textures

[13]Of course those can be simulated by register swaps as in Assembler

Parameters constants like light direction $\vec{l}$, material, ...

Output color, depth

The light direction is given in the eye-space, because of the OpenGL ModelView matrix (we can only work in the eye-space).

$\Rightarrow$also transform the normals into eye-space before passing them to the shader

Like Vertex Shaders flow control and early return is not possible without tricks. Pixel Shader use an API instead of free programming code, resulting in hardware dependent optimized code.

Possibilities include

- customized texture mapping (bump mapping, environment mapping)
- accessing multiple textures (environment bump mapping)
- texture projection
- killing whole fragments (not rendering them)
- clipping with arbitrary "planes", e.g. a sphere
- multiple passes before rendering (allows for complex rendering techniques)
- rendering to a texture (e.g. to store multipass results, if no multi passing is available)
- Torrance-Sparrow Lighting using the Schlick approximation

Programs

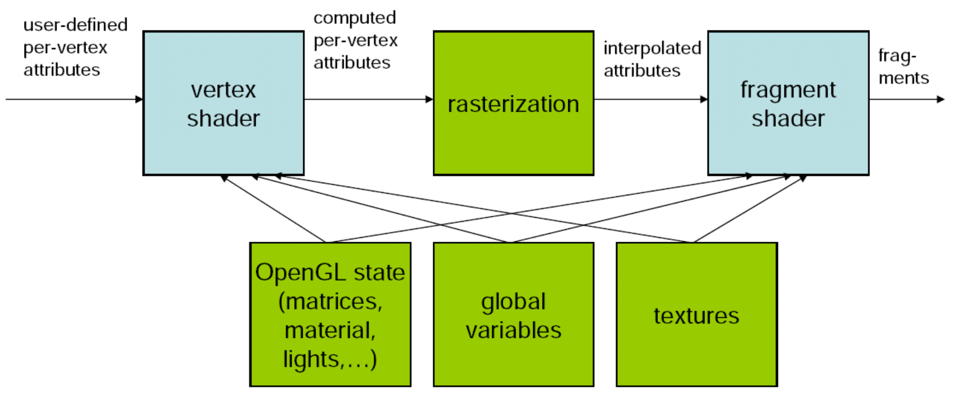

user-defined per-vertex attributes
vertex shader
computed per-vertex attributes
rasterization
interpolated attributes
fragment shader
frag-ments
OpenGL state (matrices, material, lights,...)
global variables
textures

Figure 127: OpenGL Shading Language

ARB_VERTEX_PROGRAM assembler like vertex shader

ARB_FRAGMENT_PROGRAM assembler like fragment shader

The program itself is passed as string: `glProgramStringARB(enum target, enum format, size len, const ubyte* program)`
And bound by: `glBindProgramARB(enum target, uint program)`

Shading Language mainly macros for the ARB_PROGRAMS

OpenGL Shading Language (GLSL) C-like language, global state/variables for the passing of parameters (e.g. `state.material, state.light`), result parameters (`result.color, result.depth`), flow control (loops, branches, conditionals)

vertex shader `glPosition = glModelViewProjecjtionMatrix * gl_Vertex;`

fragment shader `glFragColor = vec4(1,0,0,1);`

These shaders can be included into OpenGL code by: `shader = glCreateShaderObject(), glShaderSource(shader, char* source), glCompileShader(shader)`

Variable Qualifiers

attribute application defined vertex attribute (vertex shader input)

uniform application defined global variable (vertex/fragment shader input)

varying computed by vertex shader, interpolated by rasterization step, sent to fragment shader (vertex shader output, fragment shader input)

const constant variables (e.g. π)

16 History

Some Numbers

fps complex global illumination (1 frame per day), movies (1 frame per 8 minutes), interactivity (5 fps), games (50 fps)

throughput 10^6 pixel with 20 fps:

- processing $20 \cdot 10^6$ pixels per second
- 50 cycles per pixel (1 GHZ CPU)
- 3 bytes per pixel (~60 MB)

triangles games: 100.000 triangles, cave: 40.000 triangles (20 fps), reality: $80 \cdot 10^6$ triangles

per triangle we have three vertex coordinates, material properties, a texture

rendering perspective projection to screen, occlusion computation, rasterization, illumination, texturing

GPU speed doubles every 6-12 months (CPU 18 month), denser, more transistors and FLOPS than CPU (CPU are more flexible)

Today: $600 \cdot 10^6$ vertices / second, $6.4 \cdot 10^9$ pixels / second, 6 parallel vertex stages, 16 parallel pixel stages

History

Sinclair ZX81 (1982) complete pipeline is performed by the CPU

Commodore 64 (1982) graphic chips generates video signal (after CPU has written to the framebuffer)

Atari ST (1985) GPU deals with 2D graphics operations

SGI Indy (1993) GPU does rasterization step

SGI O_2 (1996) GPU does transformations and rasterization

SGI Onyx,Nvidia,ATI GPU does the entire pipeline

Today programmable stages

17 Virtual Reality

Virtual Reality in general is a computer generated world, that can be manipulated by the user. It's all about immersion, the feeling that what surrounds you is really real. For reaching immersion not only vision, should be considered, VR tries to capture other senses as well:

Senses

- vision: real time graphics, stereo vision
- sound: surround sound
- haptics: force feedback, input resistors
- smell
- taste (not yet given)

Input

The manipulation can be achieved with 3D mouses, spaceballs, data gloves, tracking devices or whole data suits. Another goal for data input is that the camera or eye can be moved by the user by moving the head. Even more difficult is eye tracking. Furthermore the ability to move objects and grab and drop are favorable immersion boosts.

Output (Stereo Vision)

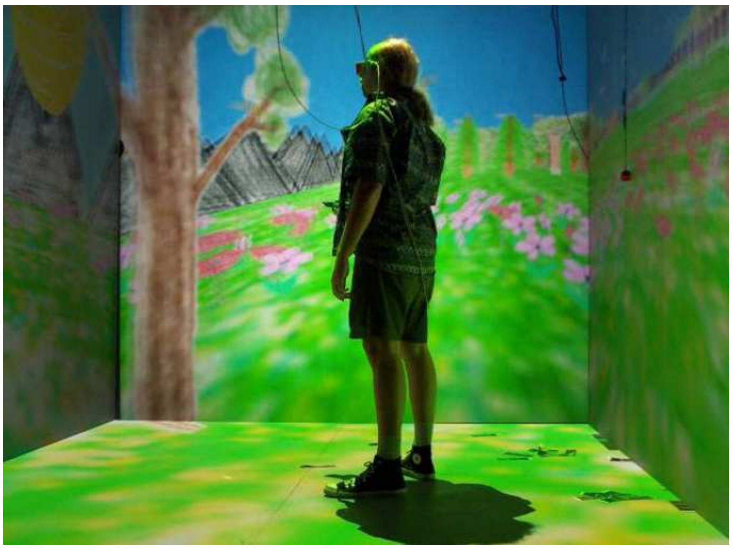


Figure 128: CAVE
An output system called CAVE for an example for a great level of immersion.

Graphics are often displayed by special **Head-Mounted Devices (HMD)**, special beamer technology or whole rooms. The special is referring to giving the possibility of stereo vision, which means to separate images for the left and for the right eye. The first device, the HMD, achieves this by supplying one LCD screens for each eye. HMD can easily combined with sound output and position tracking input. However they are very heavy and uncomfortable and thus reduce immersion. A softer version of head glasses are **Shutter Glasses**. They alternately blacken the left and the right eye, thus providing the right frames, simulate spatial viewing. However apart from them still being somewhat uncomfortable, the images appear darkened synchronization must be assured.

Talking about **beamers** we can use two separate beamers projecting their images through projection filters. Polarization filters let pass light only in one direction. Supplying the user with glasses which have two projection filters with the corresponding directions, the images can be separated for stereo view again. This is what 3D cinemas usually do. This is usually done by front projection, however then often the user shadows the projection by his geometrical physical form. Using a mirror we can use back projection as well, avoiding this problem. However then we need some room behind the screen.

A third alternative are **work benches**. The screen is like a drawing desk where upon the image is projected. The user's position is tracked and the image is adjusted accordingly to provide 3D vision.

Finally we can use a whole room or chamber to maximize the level of immersion. This is called **Cave Automatical Virtual Environment (CAVE)**. This is one of the most expensive VR architectures, since we need six (twelve for passive stereovision) beamers for every wall of the chamber including a beamer for the bottom floor underneath the room. Furthermore the computation and synchronization takes the power of graphics clusters.

Stereo Projection

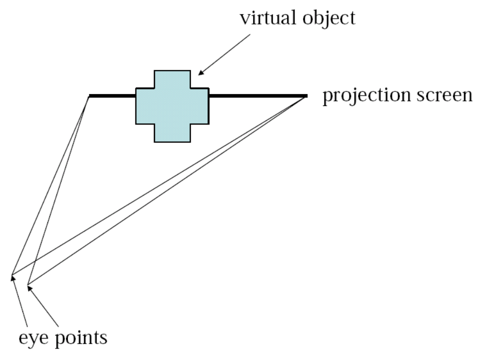


Figure 129: Stereo Projection

Assuming we know the position of the viewer, resp. her eyes, how can we calculate the right stereo images?

A straightforward method is to place the camera successively onto the left and the right eye and render the image towards the center of the "screen". However this method fails when the viewer is not centered and looks at the screen at a different angle.

Another method is calls **sheared perspective**. The projection screen is the image plane and we allow the eye point to be anywhere. This means our viewing frustum becomes sheared. OpenGL offers a sheared viewing frustums called `glFrustum`.

List of Algorithms

List of Figures

List of Tables

Index