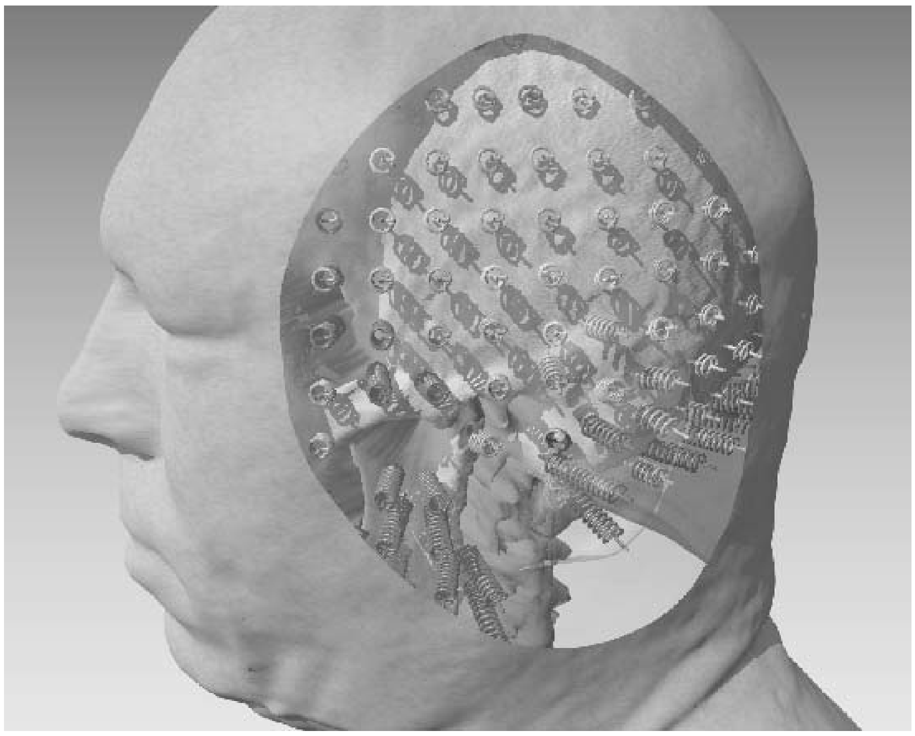
GRAPHISCHE DATENVERARBEITUNG



Prof. Dr. Markus Gross
Computer Graphics Laboratory
Institut für Visual Computing
Departement Informatik

Eidgenössische
Technische Hochschule
Zürich

The script is under construction.

Contents

List of Figures vii

List of Tables xiii

1 Introduction 1

1.1 Definitions of Graphical Data Processing . . . 1
1.2 Areas of graphic data processing . . . 2
1.3 Fundamentals of realistic, computer-generated images . . . 3
1.4 Rendering Pipeline . . . 5
1.5 Visualization Pipeline . . . 5

2 Light and Colors 9

2.1 Basics of sensory physiology and physics . . . 9
2.2 Units of measurement for light . . . 10
2.2.1 Light flow . . . 10
2.2.2 Light intensity . . . 10
2.2.3 Illuminance . . . 11
2.2.4 Luminance . . . 12
2.3 Definition and Physiology of Color . . . 14
2.3.1 definition . . . 14
2.3.2 Physiology . . . 15
2.4 The norm valence system of the CIE . . . 15
2.4.1 CIE standard valence curves . . . 15
2.4.2 The CIE Chart . . . 20
2.4.3 Important color spaces for practice . . . 21
2.4.4 Monitor calibration . . . 23

2.5 color spaces 23
2.5.1 RGB color space 23
2.5.2 CMY color space 23
2.5.3 HSV color space 24
2.5.4 YIQ color space 26
2.5.5 Perception-oriented color spaces 27

3 Geometric transformations 31
3.1 Introduction 31
3.2 2D transformations 32
3.2.1 Translation 32
3.2.2 Scaling 32
3.2.3 Rotation 32
3.2.4 Homogeneous coordinates 33
3.2.5 Shear 36
3.2.6 Scaling and rotation in homogeneous coordinates 36
3.2.7 Concatenation of 2D transformations 37
3.3 Coordinate systems 39
3.3.1 World, object and camera coordinates 39
3.3.2 Windows and Viewports 39
3.3.3 Calculation bases 40
3.4 3D transformations in homogeneous coordinates 42
3.4.1 Right and left systems 42
3.4.2 Translation 44
3.4.3 Scaling 44
3.4.4 Rotation 44
3.4.5 Inversion 45
3.4.6 Shear 45
3.4.7 Transformation of normal vectors 46
3.4.8 Composite 3D Transforms 46
3.4.9 Example application flight simulation 49
3.5 3D rotations and translations with quaternions 51
3.5.1 Definition and Properties 51
3.5.2 Quaternions of length one 52
3.5.3 3D rotations using unit quaternions 53
3.5.4 Translations and concatenations 54

4 Projections for 3D representation 57
4.1 Fundamentals of planar projections 57
4.1.1 Parallel Projection 58
4.1.2 Perspective projections 61
4.1.3 Coordinate systems and viewports 62
4.1.4 Clipping Planes 63
4.2 Mathematics of projection types 65
4.2.1 Perspective projection 66
4.2.2 parallel projection 68

4.2.3 General formulation . . . 68
4.2.4 summary . . . 70

5 Clipping 73
5.1 Introduction . . . 73
5.2 Line clipping in 2D . . . 73
5.2.1 brute force method . . . 74
5.2.2 Cohn-Sutherland Algorithm . . . 75
5.2.3 Parametric line clipping (Liang-Barsky / Cyrus-Beck) . . . 79
5.3 Polygon clipping in 2D . . . 84
5.3.1 Identification of convex polygons . . . 85
5.3.2 Sutherland-Hodgeman algorithm . . . 86
5.3.3 Liang-Barsky polygon clipping . . . 89
5.4 Line clipping in 3D . . . 92
5.4.1 Cohn-Sutherland Algorithm . . . 92
5.4.2 Parametric Clipping . . . 94
5.5 Clipping in homogeneous coordinates . . . 95

6 Scan Conversion 97
6.1 Lines . . . 97
6.1.1 First incremental algorithm (Digital differential analyzer) . . . 97
6.1.2 Bresenham's algorithm . . . 99
6.2 Circles(Bresenham) . . . 102
6.2.1 Derivation from implied circle equation . . . 102
6.2.2 Elimination of real arithmetic . . . 104
6.2.3 Improvement by second-order partial differences . . . 105
6.3 Ellipses . . . 107
6.4 Scan conversion of polygons . . . 110
6.4.1 A three-step algorithm . . . 112
6.4.2 Horizontal edges . . . 112
6.4.3 Problems with long thin polygons (slivers) . . . 113
6.4.4 Edge coherence . . . 113
6.4.5 The Active Edge Table data structure (AET) . . . 115

7 Hidden Line and Hidden Surface Algorithms 117
7.1 Hidden Line Algorithms . . . 117
7.1.1 Back Face Culling . . . 117
7.1.2 Appel's algorithm . . . 118
7.2 Hidden Surface Algorithms . . . 120
7.2.1 Z buffering . . . 120
7.2.2 Depth sorting (Newell-Newell-Sancha) . . . 122
7.2.3 BSP trees . . . 126
7.2.4 Warnock's Algorithm . . . 128
7.2.5 Weiler-Atherton Algorithm . . . 129

8 Lighting, Shading and Texturing 137
8.1 Lighting . . . 137
8.1.1 Ambient Light . . . 137
8.1.2 Diffuse Reflection . . . 138
8.1.3 Attenuation . . . 138
8.1.4 Colors . . . 139
8.1.5 Depth Cueing . . . 140
8.1.6 Specular Reflection (Directional Reflection) . . . 140
8.1.7 Phong's lighting model . . . 141
8.1.8 Modeling light sources . . . 144
8.2 Shading . . . 145
8.2.1 Constant shading . . . 145
8.2.2 Gouraud shading . . . 145
8.2.3 Phong shading . . . 147
8.3 Transparency and refraction . . . 149
8.3.1 Derivation of the refraction vector . . . 151
8.3.2 Neglecting refraction (α *blending)* . . . 152
8.4 cast shadow . . . 153
8.4.1 Scan line shadow calculation . . . 154
8.4.2 Shadow Volumes . . . 154
8.5 Texture Mapping . . . 157
8.5.1 Mapping of brightness and color functions . . . 157
8.5.2 Reflection Mapping . . . 160
8.5.3 Aliasing Effects in Texture Mapping . . . 160
8.5.4 Bump mapping . . . 165

9 The Open Graphics Library (OpenGL) 169
9.1 Introduction . . . 169
9.1.1 Graphics systems and standards . . . 169
9.1.2 The OpenGL . . . 170
9.1.3 OpenGL organizational principles . . . 172
9.2 The OpenGL Pipeline . . . 173
9.2.1 Graphic primitives . . . 173
9.2.2 Transformations . . . 173
9.2.3 Lighting Models . . . 176
9.2.4 Texture Mapping . . . 177
9.2.5 Clipping and projection . . . 179
9.2.6 Raster scan conversion and antialiasing . . . 180
9.2.7 Pixels and Bitmaps . . . 181
9.2.8 The framebuffer . . . 182
9.2.9 Variable . . . 183
9.3 Integration into window systems . . . 184

10 Recursive Raytracing 189
10.1 Global Illumination . . . 189
10.2 Abstract description by Kajiya's rendering equation . . . 190

10.3 Recursive Raytracing 191
10.3.1 Schematization of the algorithm by ray tree 193
10.3.2 Recursive definition of the local lighting model 193
10.3.3 Typical ray tracing algorithm 194
10.3.4 Adaptive tree depth control 196
10.3.5 Ray box intersection calculation 196

11 Antialiasing 203

11.1 Definitions 203
11.1.1 Folding 203
11.1.2 Fourier Transform 204
11.1.3 Elementary Relations 206
11.2 Sampling 206
11.2.1 Sampling of One-Dimensional Functions 206
11.2.2 Sampling of Two-dimensional Functions 210
11.3 Antialiasing Methods 212
11.3.1 Band Limitation through Filtering 212
11.3.2 Filtering of Textures 215
11.3.3 Raytracing Supersampling Methods 216
11.3.4 Adaptive Supersampling 217
11.3.5 Stochastic Supersampling 217

List of Figures

1.1 Cyclic action unit of graphic data processing and image processing 2
1.2 Main research areas of graphic data processing 3
1.3 The rendering pipeline 6
1.4 The visualization pipeline 7

2.1 Light is the visible part of the electromagnetic spectrum 10
2.2 Spectral brightness sensitivity of the human eye a) Photopic domain $V(\lambda)$ b) Scotopic region $V'(\lambda)$ 11
2.3 Typical relative spectral power density distribution of a light source 12
2.4 Illustration of the geometric arrangement of transmitter and receiver 12
2.5 Goniometric diagram: luminous intensity distribution curve for a physical light source (far field of a surface element). 13
2.6 Relative spectral energy distribution of sunlight in different atmospheric layers 14
2.7 Spectral discrimination properties 16
2.8 Relative spectral sensitivity of the human eye 17
2.9 Neural coding of color perception 17
2.10 Schematic test setup for determining the standard valence curves of the CIE 1931 18
2.11 Spectral evaluation functions $\overline{r}, \overline{g}, \overline{b}$ for the primary valences R,G,B 18
2.12 Normalized spectral weighting functions x, y, z for the standard primaries X, Y, Z 19
2.13 CIE chart with some characteristic color locations 21
2.14 Nomenclature for areas of similar color types in the CIE chart 22
2.15 color coordinates in the CIE chart 22
2.16 RGB color space 24
2.17 CMY color space 24
2.18 HSV color space 25

2.19 Distinction thresholds as ellipses in the McAdams experiment for different color locations (scaled 10 times) 28
2.20 Nonlinear distortions of the La*b* and Lu*v* color spaces 29

3.1 interpretation of the two-dimensional homogeneous coordinates in 3D 34
3.2 World and object coordinates 39
3.3 Relationship between world and screen coordinates, windows and viewports . . 40
3.4 Non-uniform scaling and multiple viewports for the same window 40
3.5 mapping of a window into a viewport 41
3.6 Clipping at window and viewport 42
3.7 Right-handed coordinate system 43
3.8 Left-handed coordinate system 43
3.9 Object and world coordinates for flight simulation 50
3.10 Arrangement of characteristic vectors for 3D rotation 53
3.11 concatenation of translation and rotation 55
3.12 Movement sequence of a camera coordinate system. 55

4.1 Parallel and perspective projection 57
4.2 Overview of the variants of the planar projection 58
4.3 Construction of Three Orthographic Projections 59
4.4 Construction of an Isometric Projection of a Unit Cube 59
4.5 Isometric Projection of the Unit Vectors with the Projection Direction $(1, 1, 1)$. 59
4.6 Construction of an Oblique Projection 60
4.7 Cavalier Projection of the Unit Cube 60
4.8 Cabinet Projection of the Unit Cube 61
4.9 Illustration of the Angles in Cavalier and Cabinet Projection 61
4.10 Perspective projection of the unit cube with a vanishing point 62
4.11 Perspective 2-point projection of the unit cube (in this drawing, the projection plane corresponds to the drawing plane. 63
4.12 Example of a 3 point projection (projection plane intersects all 3 coordinate axes) compared to the 2 point projection of the same object (only the x and y axes are intersected 64
4.13 Plane of projection in the legal system 64
4.14 Right-hand reference coordinate system given by the u,v, and n-axis (camera or viewing coordinate system) 64
4.15 Viewing volumes for parallel and perspective projection 65
4.16 Plane of projection in the left system 65
4.17 Clipping planes with perspective projection 65
4.18 Clipping planes in oblique parallel projection 66
4.19 Perspective projection 66
4.20 Alternative perspective projection into the (z=0)-plane 68
4.21 Illustration for the derivation of a general formulation for the projection 69
4.22 Transition from world coordinates to camera coordinates 71

5.1 Line clipping in 2D 74
5.2 Polygon clipping in 2D 74
5.3 4-bit codes of the different regions 75

5.4 Illustration of the Cohen-Sutherland clipping 77
5.5 Dot product for three points, one outside, one inside and one on the clipping rectangle 80
5.6 Lines diagonal to the clipping rectangle 81
5.7 difference between convex and concave polygon 85
5.8 Sign of the cross products for testing convex polygons 85
5.9 Polygon clipping at a rectangular clipping window 86
5.10 Relationship when clipping edges on lines 87
5.11 visibility test 87
5.12 Flowchart of the Sutherland-Hodgeman algorithm 88
5.13 Pre-sectioning of the plane 89
5.14 Different positions of the starting point in the sections 90
5.15 necessity of a point in window vertex 91
5.16 The two possibilities of the intersection of a line with the window 91
5.17 Clipping volumes in 3D 93

6.1 scan-converted line 98
6.2 Incremental calculation of (x_i, y_i) 98
6.3 Points M and Q, as well as pixels N and NE in the Bresenham algorithm . . . 100
6.4 increments in the Bresenham algorithm related to the respective quadrant . . . 102
6.5 Eight symmetrical points on the circle 102
6.6 Pixel grid with the points M, E and SE 103
6.7 Example of creating a quadrant using symmetry 105
6.8 Standard ellipse at origin 108
6.9 Two regions of an ellipse defined by the $45°$ tangent 108
6.10 polygon and scan line 111
6.11 Span of a polygon: extrema are shown in black, inner points are grey: a) Extrema calculated with midpoint algorithm (Bresenheam) b) extrema within the polygon 111
6.12 Horizontal edges in a polygon 113
6.13 Scan conversion of Silvers 114
6.14 ET for the example polygon from Fig. 6.10 115
6.15 AET for scan lines 9 and 10 from Fig. 6.10 116

7.1 Illustration of back face culling: grey: *Back-facing polygons* black: *Front-facing polygons* 118
7.2 Quantitative invisibility of lines 119
7.3 Contour line runs in front of the polygon edge 119
7.4 Example of how the Z-buffer works 121
7.5 Interpolation of z values by incrementing 122
7.6 Different situations for the spatial arrangement of polygons 122
7.7 Test 4 is fulfilled 123
7.8 Test 4 is not fulfilled, test 5 is fulfilled 123
7.9 Example of deciding tests 4 and 5 125
7.10 Clustering of a polygon scene and associated BSP tree 126
7.11 Examples for building BSP trees 132

7.12 BSP tree traversal and resulting scan conversion order for two different projections of the same scene . . . 133
7.13 Illustration of case 4 for deciding whether a further subdivision is necessary for quadtree methods . . . 133
7.14 Subdivision into square cells . . . 134
7.15 Subdivision around the polygon vertices marked with circles . . . 135
7.16 Examples for the Weiler-Atherton algorithm (a) original scene (b) Polygons clipped to A (c) inside list of A clipped to C (d) visible parts within A (e) Polygons clipped to B (f) All visible parts of the scene . . . 136

8.1 Diffuse reflection . . . 138
8.2 Diffuse reflection: Ratio between incident and reflected light intensity depending on the size of the radiating comparison to the reflecting surface dA/(dA/$\cos\theta$) 139
8.3 Sigmoids for calculating the attenuation . . . 140
8.4 Vectors and angles in specular reflection: vector to light source L, surface normal N, reflection vector R, vector to viewer V . . . 141
8.5 Calculation of the reflection vector . . . 141
8.6 Characteristics of the $cos^n\alpha$ function for different n . . . 142
8.7 Propagation map for different n . . . 143
8.8 Warn's lighting model and spotlight . . . 144
8.9 Goniometric diagrams of Warn's light sources and a point light source . . . 145
8.10 vertex and surface normals . . . 146
8.11 Intensity interpolation along polygon edges and scan lines . . . 146
8.12 Illuminated sphere approximated by different numbers of polygons as wireframe and as solid . . . 147
8.13 Interpolation of the normal vector in Phong shading . . . 148
8.14 Interpolation of the vertex normals . . . 148
8.15 Flat, Gouraud and Phong Shading in comparison . . . 149
8.16 Refraction in the water glass . . . 149
8.17 Snell's law of refraction . . . 150
8.18 Refractive index as a function of wavelength (left) and some typical indices measured at a fixed wavelength (right). . . . 150
8.19 Arrangement for the derivation of the vector **T** . . . 151
8.20 Absorption factor in α blending . . . 153
8.21 Scan-Line Shadow Algorithm . . . 155
8.22 Shadow volume defined by the light source and the shadow-casting polygon . . 156
8.23 Calculating the shadow cast with the help of shadow volumes . . . 156
8.24 Texture mapping from pixel to surface into texture map . . . 157
8.25 Mapping a line texture to a spherical octant . . . 158
8.26 Coordinate relationships in texture mapping to a cylinder . . . 160
8.27 Calculation of texture coordinates in reflection mapping . . . 161
8.28 Illustration of Reflection Mapping . . . 162
8.29 Texture mapping by object space subdivision . . . 162
8.30 projection of the pixel in image space onto the texel in texture space . . . 165
8.31 Core points for calculating partial derivatives with finite differences . . . 166

8.32 Texture mapping methods in comparison (from left to right: modulation of colors - or brightness values, bump mapping, displacement mapping and reflection mapping) . . . 167

9.1 Block diagram of data flow in OpenGL . . . 172
9.2 Types of geometric primitives in OpenGL . . . 174
9.3 Relationship between coordinates and attributes of a corner point . . . 175
9.4 sequence of vertex transformations to represent primitives . . . 176
9.5 a) Monochrome triangle b) Gouraud-shaded triangle c) More complex shaded object . . . 177
9.6 Utah teapots with different surface properties . . . 178
9.7 mipmap representation of a texture . . . 179
9.8 a) Typical scene from many textured polygons (driving simulation) b) Contouring through texture mapping c) reflection mapping . . . 179
9.9 Viewing Volume of Perspective Projection . . . 180
9.10 Determining the percentage coverage of a pixel . . . 181
9.11 Bitplanes of the framebuffer . . . 182
9.12 GLX client, X server and OpenGL renderer . . . 185
9.13 Output of the sample program above . . . 188

10.1 Principle of ray casting . . . 190
10.2 Influence of reflection and refraction on lighting . . . 190
10.3 REflection and transmission on the way eye-light source . . . 192
10.4 Path of a line of sight in an example scene . . . 193
10.5 Resulting binary ray tree for the example scene in Figure 10.4 . . . 194
10.6 Flow chart for recursive ray tracing (for the meaning of the nomenclature see the following page) . . . 198
10.7 Traversal of the ray tree in the algorithm from Figure 10.6 . . . 199
10.8 Calculation of the intensity at an intersection . . . 200
10.9 Ray box intersection calculation for slabs perpendicular to the main axes . . . 201

11.1 (a) 2D block function (b) Corresponding Fourier spectrum (c) Spectrum shown as intensity distribution (logarithmically scaled) . . . 205
11.2 Illustration of sampling and aliasing (from the complex-valued Fourier transform, the magnitude is plotted) . . . 208
11.3 Illustration of the sampling of time-limited signals . . . 209
11.4 Illustration of the Discrete Fourier Transform . . . 210
11.5 2D momentum combs . . . 211
11.6 Representation of a sampled, band-limited 2D function in frequency space . . . 212
11.7 Design criteria for interpolation filters . . . 213
11.8 Different 1D filters and their Fourier transforms . . . 214
11.9 Same shapes on screen correspond to different shapes in texture space . . . 216
11.10Local variant filter: Circular region in screen space (left) are mapped to elliptical region in texture space. These regions differ in size, eccentricity and orientation. Dots mark pixel centers. . . . 216
11.11Example of super sampling: Four corner rays and one center ray are traced for each pixel . . . 217

11.12 Example of adaptive super sampling 218
11.13 Jittering: Each sampling point is "jiggled" by two uncorrelated random values in the x and y direction 219
11.14 Poisson samples (a) versus jittering (b) 219
11.15 FFRNS for $l_{k_0}\beta = 0.5$ and $l_{k_0}\beta = 0.95$ 221
11.16 FFRNS for uniformly distributed jitter 222
11.17 FFRNS for $\alpha = 1.0$ and $\alpha = 0.5$ 222

List of Tables

2.1 Table of some typical illuminance levels or luminance levels 13

5.1 Table of calculations required for parametric clipping 82
5.2 Table of calculations needed for parametric 3D clipping 94

8.1 Mapping of the lines from Fig. 8.26 (a) to the spherical octants in Fig. 8.26 (b) . 159

Introduction

1.1 Definitions of Graphical Data Processing

Computer graphics concerns the pictorial synthesis of real or imaginary objects from their computer-based models, whereas the related field of image processing (also called picture processing) treats the converse process: the analysis of scenes, or the reconstruction of models of 2D or 3D objects from their pictures.

James D Foley

Graphic data processing deals with the creation and management of computer-aided, attributed, geometric models and the generation of images from these models.

- It thus creates access to the visual interpretation of complex relations and semantics and thus increases the gain in knowledge in many areas of science and technology.
- It is interdisciplinary and integrative.
- It is complementary to image processing and supplements it to form a cyclic effect unit.

In contrast, the definition of visualization, also often called scientific visualization, is far more general. This term was coined by an expert committee of the National Science Foundation established in 1987 as follows:

"Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science.

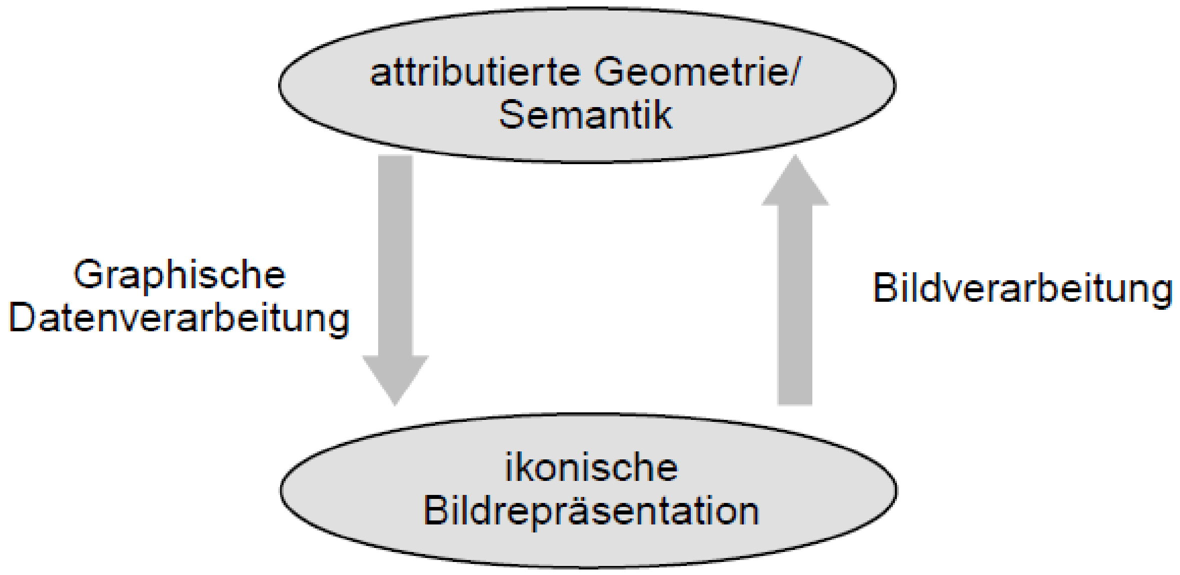

attributierte Geometrie/ Semantik
Graphische Datenverarbeitung
Bildverarbeitung
ikonische Bildrepräsentation

Figure 1.1: *Cyclic action unit of graphic data processing and image processing*

Visualization embraces both image understanding and image synthesis. That is, visualization is a tool both for interpreting image data fed into a computer, and for generating images from complex multidimensional data sets. It studies those mechanisms in humans and computers which allow them in concert to perceive and communicate visual information. Visualization unifies the largely independent but convergent fields of:

- *Computer Graphics*
- *Image processing*
- *Computer vision*
- *Computer-aided design*
- *Signal processing*
- *User interface studies"*

NSF Report on Visualization in Scientific Computing, 1987

1.2 Areas of graphic data processing

Graphic data processing, to which the following lecture provides an introduction, includes a wide variety of work areas from modeling and CAD to graphic systems and iconic image processing. This is illustrated in the following picture, whereby only a part of the subject areas is covered within the lecture cycle.

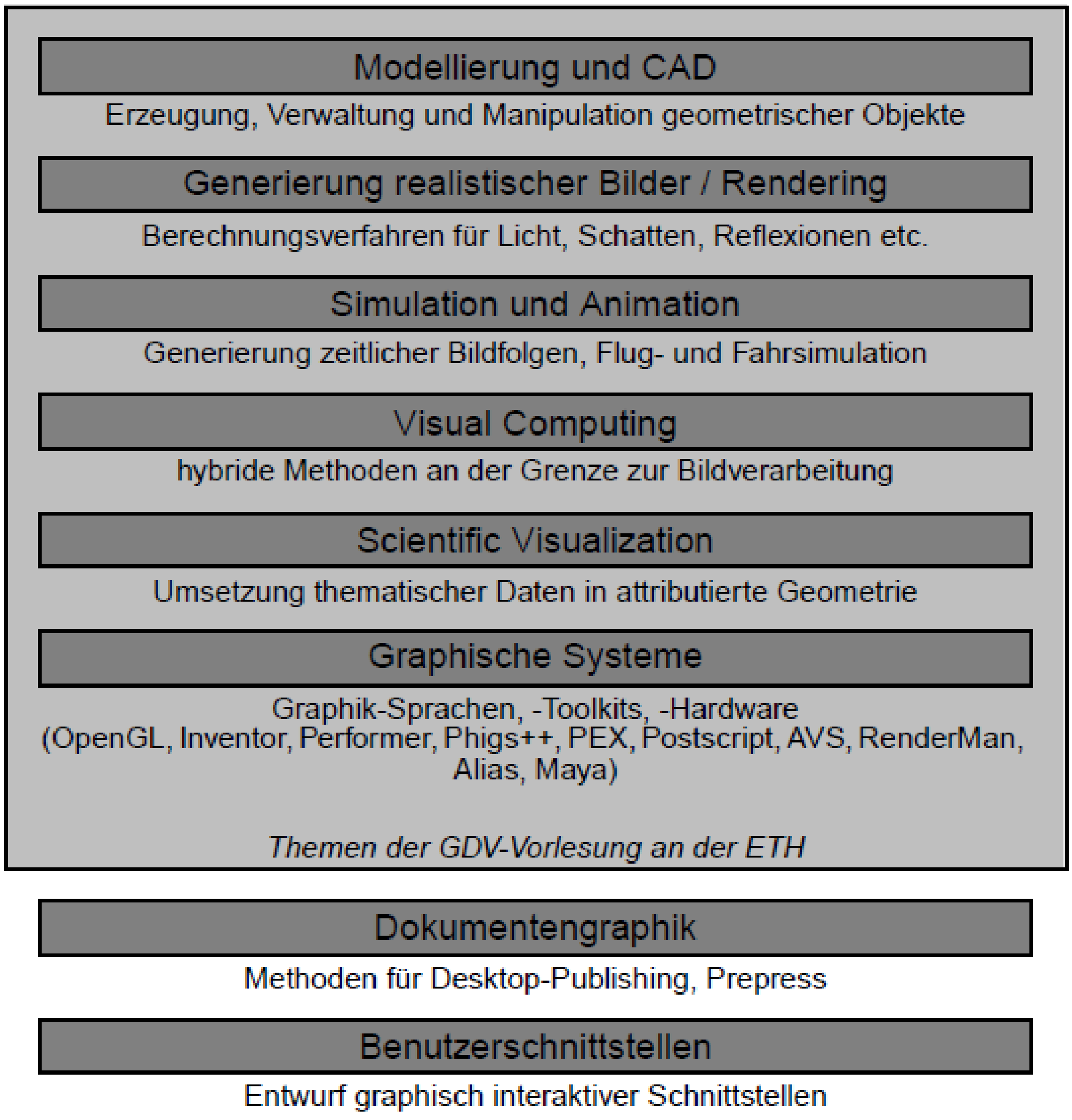

Modellierung und CAD
Erzeugung, Verwaltung und Manipulation geometrischer Objekte
Generierung realistischer Bilder / Rendering
Berechnungsverfahren für Licht, Schatten, Reflexionen etc.
Simulation und Animation
Generierung zeitlicher Bildfolgen, Flug- und Fahrsimulation
Visual Computing
hybride Methoden an der Grenze zur Bildverarbeitung
Scientific Visualization
Umsetzung thematischer Daten in attributierte Geometrie
Graphische Systeme
Graphik-Sprachen, -Toolkits, -Hardware (OpenGL, Inventor, Performer, Phigs++, PEX, Postscript, AVS, RenderMan, Alias, Maya)
Themen der GDV-Vorlesung an der ETH
Dokumentengraphik
Methoden für Desktop-Publishing, Prepress
Benutzerschnittstellen
Entwurf graphisch interaktiver Schnittstellen

Figure 1.2: *Main research areas of graphic data processing*

1.3 Fundamentals of realistic, computer-generated images

A sequence of processing steps is required for computer-aided image generation. Starting from an initial object modeling, the scene is transformed, shading, lighting and texturing are calculated, the hidden edges are eliminated, invisible areas are clipped, the scene is projected into the image plane and finally mapped to individual discrete image elements, so-called pixels. This sequence of processing steps is also called *graphics pipeline* or *rendering pipeline*, although the order of the individual steps can vary depending on the method used.

1. *Item descriptions*

- geometry (in the form of primitives, hierarchies)
- material (texture, reflection properties, color, physical properties)
- Topological properties

2. *Transformation and projection methods*
    - 3D transformations (affine)
    - perspective / parallel projection
    - camera model (lens as pinhole)
3. *Lighting and Shading*
    - Description of the light sources (color, spectral characteristics)
    - reflection model (diffuse, specular)
    - Transparency and shadows
    - multiple reflections
    - texturing
    - Cast shadow
    - volume effects
4. *Hidden lines and areas*
    - Hidden Line
    - Hidden Surface
    - backface culling
5. *clipping*
    - 2D clipping for display
    - 3D-Clipping at the viewing pyramid
6. *scan conversion*
    - Conversion of the projected geometric primitives into discrete pixel values
7. *Display*
    - Mono/Stereo
    - Head mounted display
    - holograms

1.4 Rendering Pipeline

The *Rendering Pipeline* describes the path from a geometric, attributed model to the (image) output. This is shown as an example in the following image. To achieve interactivity, the pipeline must be reversible (*interaction pipeline*), i.e. the path from the pixel to the geometry and its attributes must be clearly known.

1.5 Visualization Pipeline

The often quoted *visualization pipeline* describes the process of mapping thematic data (of any format) via attributed geometry into images. The rendering pipeline is a part of the visualization pipeline.

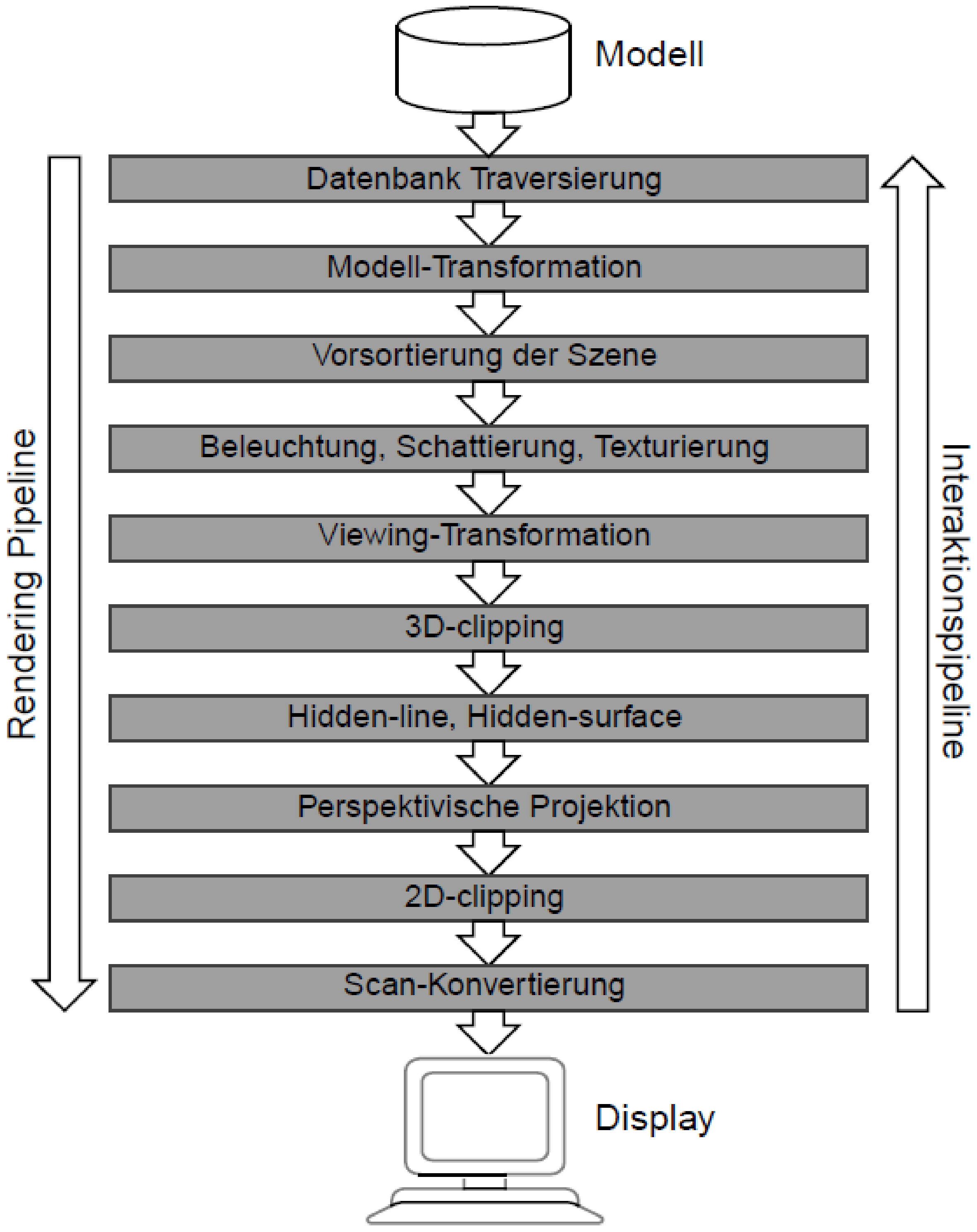


Figure 1.3: *The rendering pipeline*

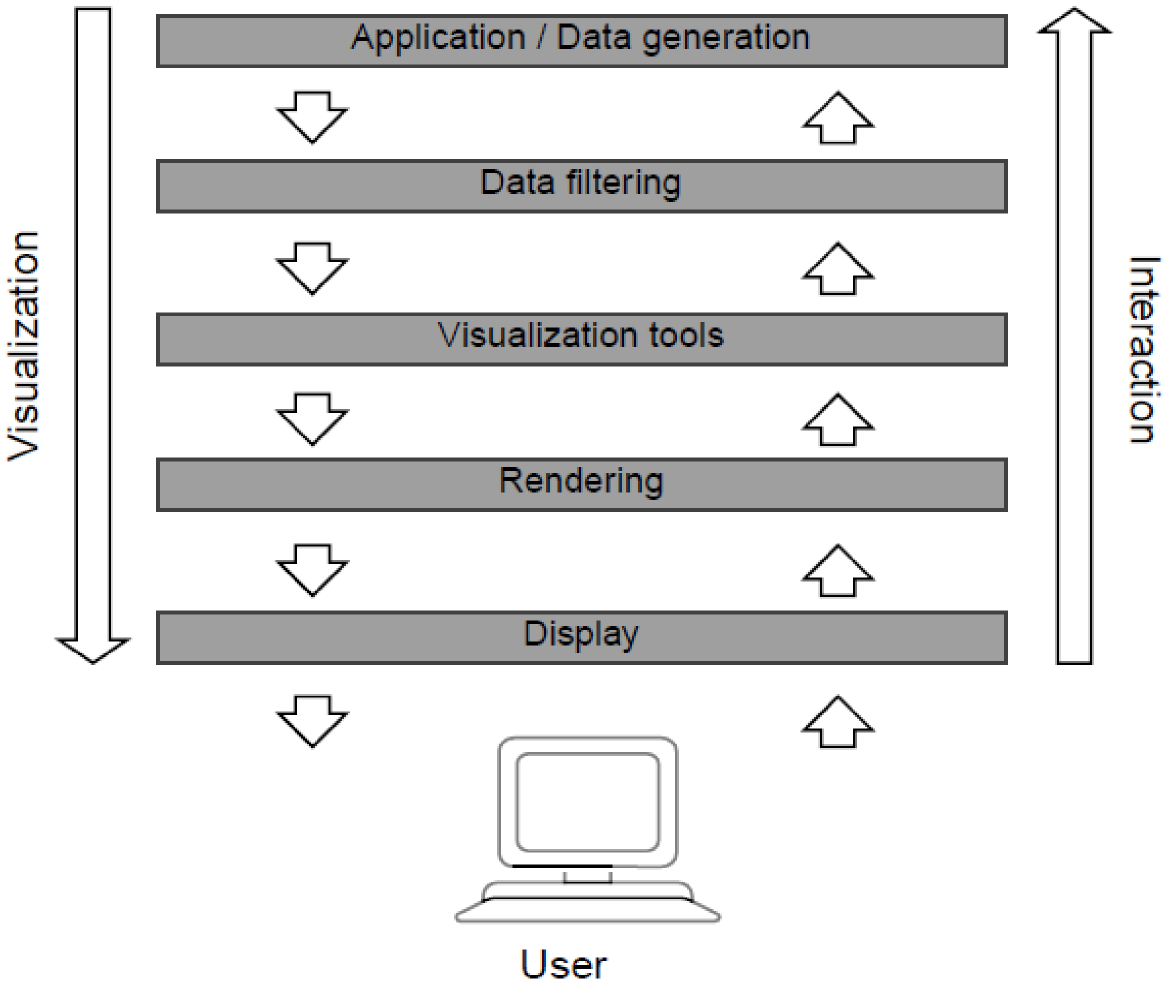


Figure 1.4: *The visualization pipeline*

Light and Colors

2.1 Basics of sensory physiology and physics

Light is the visible part of the electromagnetic spectrum. The following relationship applies to electromagnetic radiation: The energy E of a quantum of light is given by:

$$E = \frac{hc}{\lambda} \qquad \textit{h:Plank's constant} \qquad \textit{c:speed of light} \qquad \lambda\textit{:wavelength}$$

Thus the energy of the radiation is directly proportional to the frequency. However, as shown in Fig. 2.1, the physiological light stimulus only works from about 380 - 780nm and thus in a small part of the entire spectrum. It therefore does not correlate directly with the electromagnetic energy.

The physiological *brightness sensitivity* $V(\lambda)$ describes the sensitivity of the human eye and varies over the visible spectrum. The graph below shows the relative spectral sensitivity of the human eye as a function of wavelength. A distinction is made between curves for the *photopic range* (day vision, cones, light-adapted eye) and for the *Scotopic range* (night vision, rods, dark-adapted eye).

The aim of lighting technology is to define physical masses that take the above distribution into account. For this purpose, the following sizes were introduced.

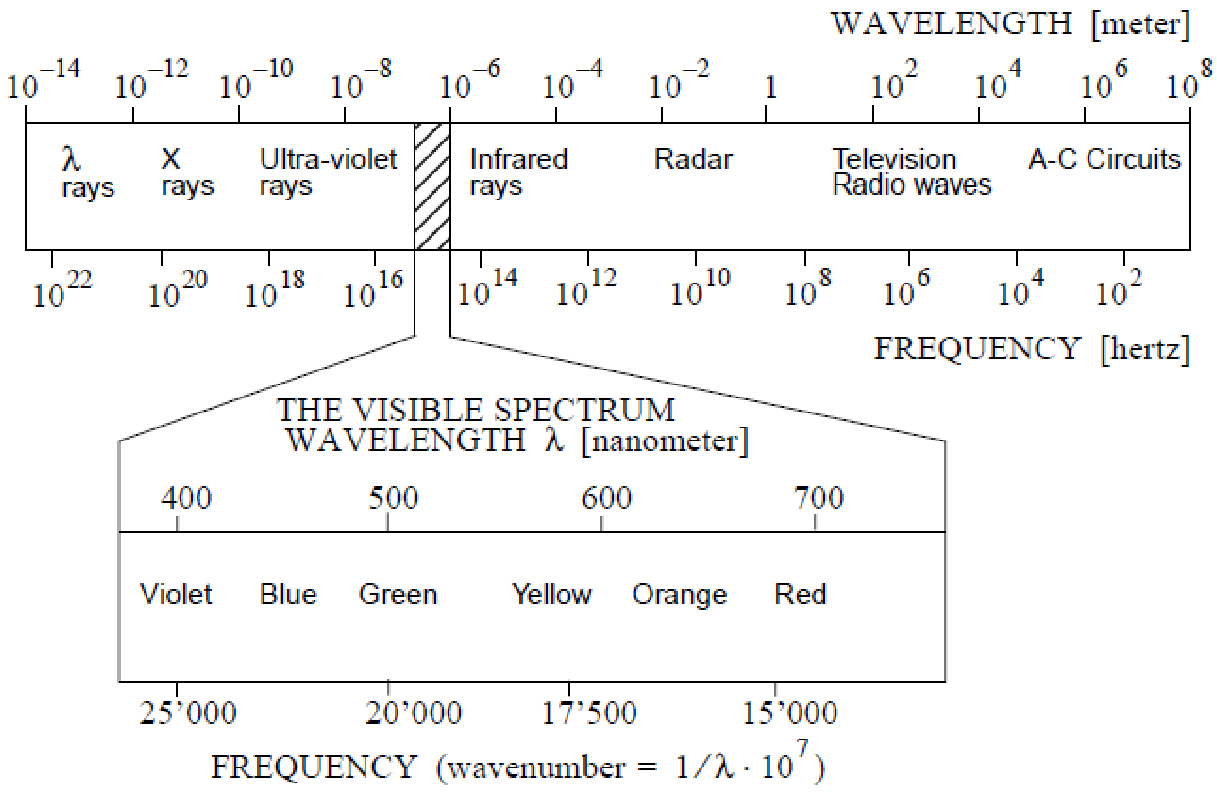


Figure 2.1: *Light is the visible part of the electromagnetic spectrum*

2.2 Units of measurement for light

2.2.1 Light flow

The *luminous flux F* describes the power emitted by a light source with the spectral power density distribution *P(λ)*, evaluated with the spectral brightness sensitivity *V(λ)*.

$$F = const \cdot \int_{380nm}^{780nm} P(\lambda)V(\lambda)\,d\lambda \qquad \text{const: 683 lm/W} \tag{2.1}$$

The unit of luminous flux is the *lumen* $[lm]$.

2.2.2 Light intensity

Of particular interest is the distribution of the luminous flux over the hemisphere defined with the light source as the center. The following arrangement is given for this, with the light source and light receiver each being shown as surface elements.

The *luminous intensity I* is the luminous flux emitted into a solid angle element $d\omega_1$. A hemisphere is placed around the radiator (Fig. 2.4). For every point on the hemisphere there is a

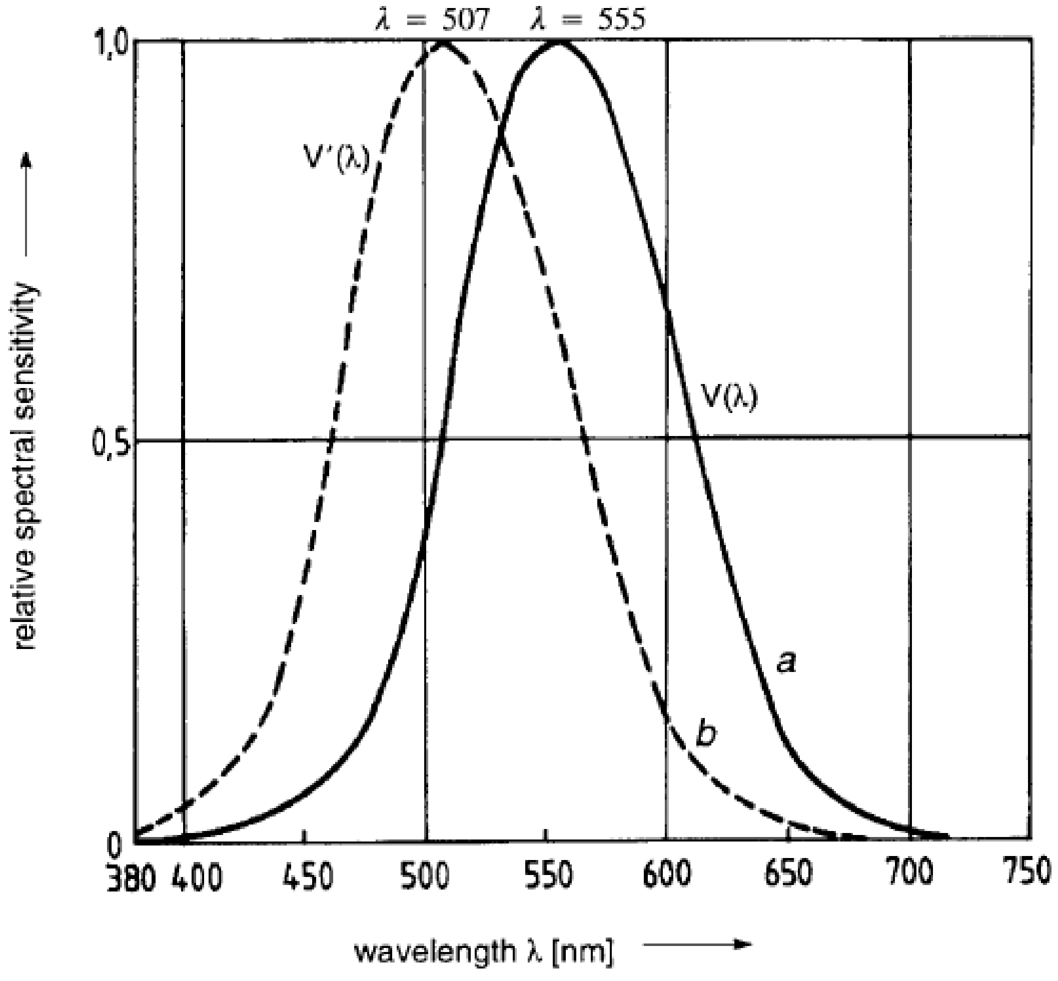


Figure 2.2: *Spectral brightness sensitivity of the human eye*
a) Photopic domain $V(\lambda)$
b) Scotopic region $V'(\lambda)$

certain luminous intensity. Sections through the resulting luminous intensity distribution body are represented by so-called goniometric diagrams (Fig. 2.5).

$$I = \frac{dF}{d\omega_1} \tag{2.2}$$

The unit of luminous intensity is the *candela* $[cd]$.

2.2.3 Illuminance

To characterize the light flow, which acts on a surface depending on the current geometry of the scene, another variable is required.
The part of the luminous flux that arrives on a surface element dA_2 is called *illuminance B*.

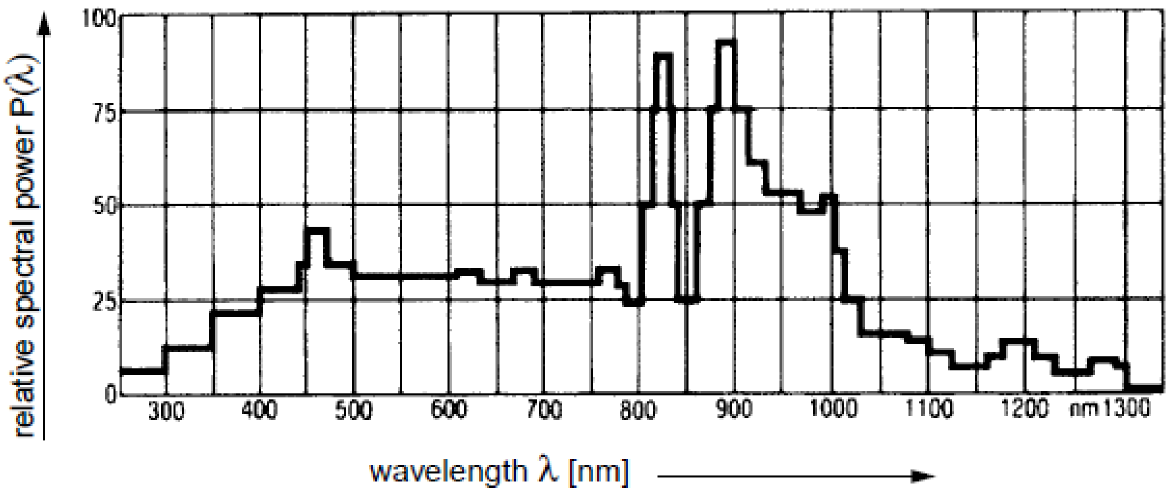


Figure 2.3: *Typical relative spectral power density distribution of a light source*

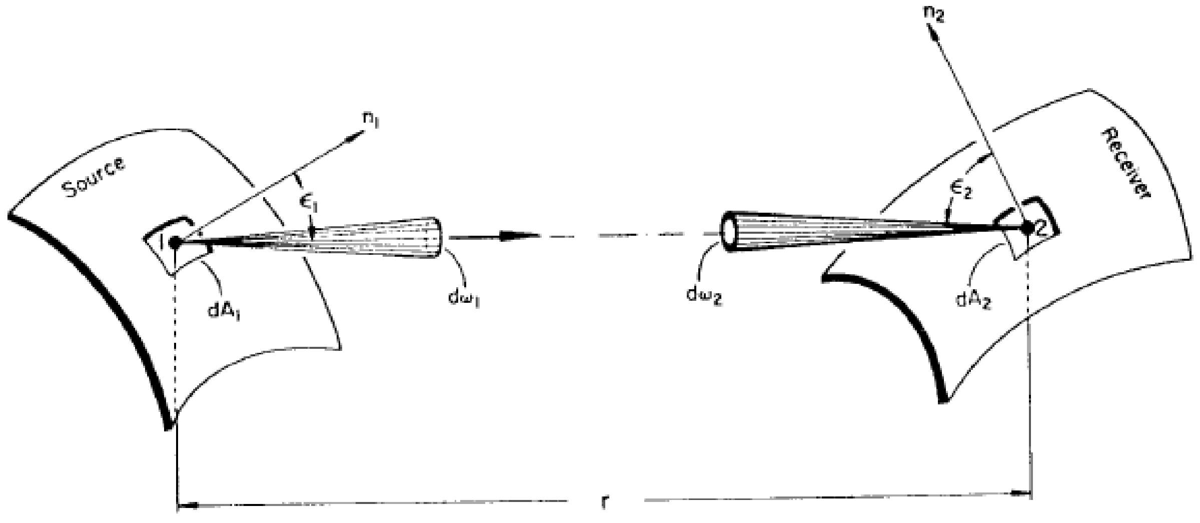


Figure 2.4: *Illustration of the geometric arrangement of transmitter and receiver*

$$B = \frac{dF}{dA_2} \tag{2.3}$$

The unit of illuminance is *lux* $[lx]$.

2.2.4 Luminance

Due to the geometric relationships between the surface normal direction and the direction of emission, represented by the angle ϵ_1, an "effective" differential luminous intensity results, which describes the emission of each surface element dA_1 of the radiator through the solid angle element $d\omega_1$. This is the *luminance* Y:

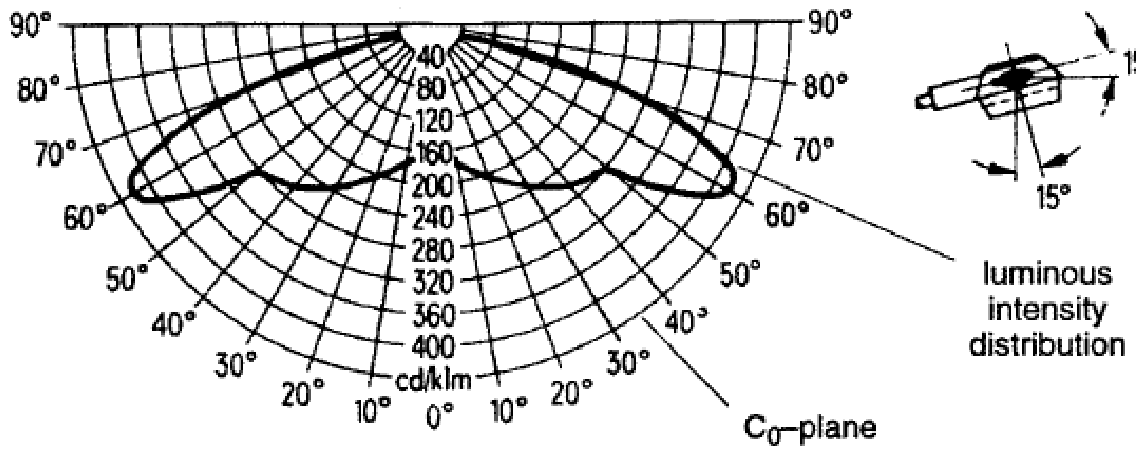


Figure 2.5: *Goniometric diagram: luminous intensity distribution curve for a physical light source (far field of a surface element).*

Table 2.1: *Table of some typical illuminance levels or luminance levels*

situation	$\mathbf{E_a}$ [lux]	**surface type**	$\mathbf{Y}$ [cd m^2]
clear sky in summer	15×10^4	grass	2900
overcast sky	16×10^3	grass	300
textile inspection	1500	light gray cloth	140
office work	500	white paper	120
heavy engineering	300	steel	20
good street lighting	10	concrete road surface	1.0
moonlight	0.5	asphalt road surface	0.001

$$Y = \frac{d^2F}{dA_1 \cdot cos\epsilon_1 \cdot d\omega_1} \tag{2.4}$$

The unit of illuminance is $candela/m^2$ $[cd/m^2]$.

`Luminance is one of the most well-known units for quantifying light`

The following table gives some values for illuminance and luminance that can occur in typical situations.

2.3 Definition and Physiology of Color

2.3.1 definition

The concept of color is very intuitive and it is generally difficult to find an accurate definition of color. A variant is given below:

> *"Color is that aspect of visual perception by which an observer may distinguish differences between two structure-free fields of view of the same spatial and temporal properties, such as may be caused by differences in the spectral composition of the radiant energy concerned in the observed."*

In technical applications, the color is often determined by a color temperature. This temperature (in K) generally corresponds to the temperature that a black Planckian radiator would have to have in order to glow with the corresponding color. The *black body* according to Max Planck is

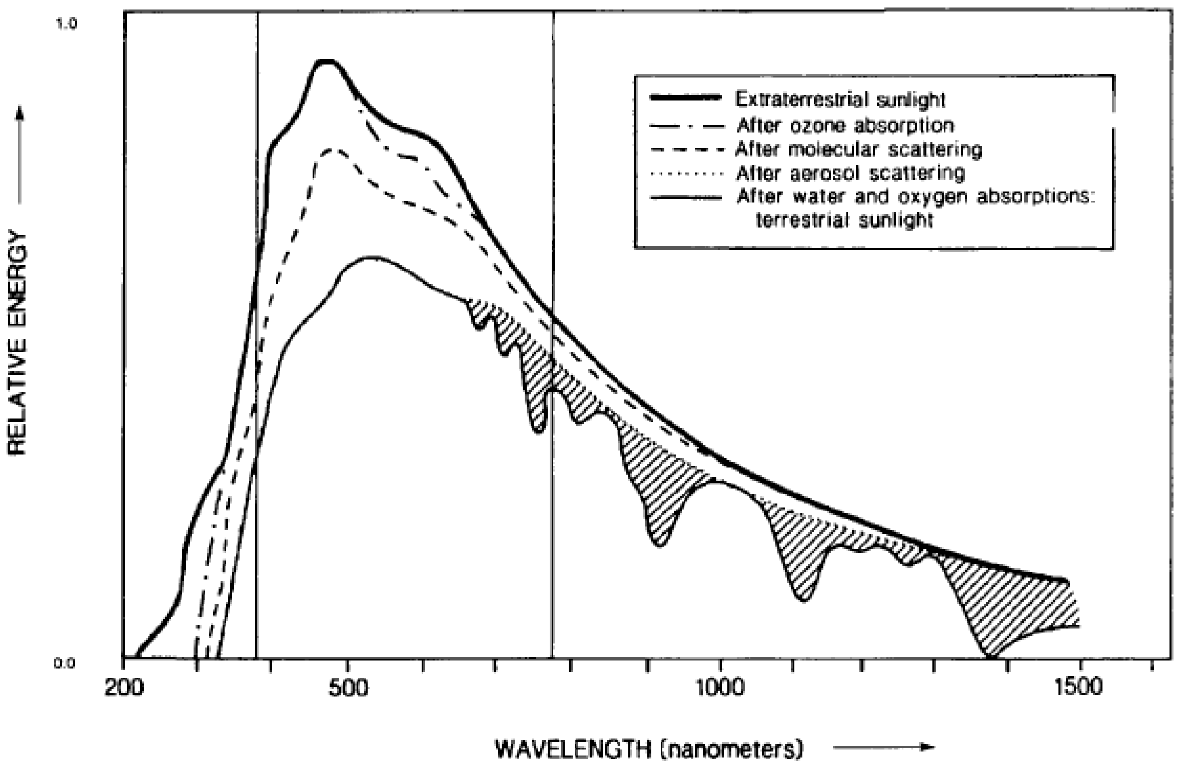


Figure 2.6: *Relative spectral energy distribution of sunlight in different atmospheric layers*

a body that converts all incident radiation into heat, regardless of wavelength and temperature, or all heat energy supplied to it into radiation. The ideal black body does not exist; Bodies with similar properties are called *grey radiators* (platinum, carbon, soot, tungsten). An example is iron, which when heated initially glows deep red, then light yellow to blue-white. With the help of Planck's radiation formula one finds the spectral energy distribution of a blackbody at a given temperature:

$$V_{\lambda T} = \frac{c_1}{\lambda^5(e^{c_2/\lambda T} - 1)} \tag{2.5}$$

$c_1 = 3.74 \cdot 10^{-12}\ W/cm^2$

$c_2 = 14.32$

λ : wavelength

T: absolute temperature in Kelvin

For example, the sun is a Planckian radiator at 6500 K.

2.3.2 Physiology

The color of a self-illuminating object is obviously determined by its spectral power density distribution $P(\lambda)$. A key goal of colorimetry is to define units of measure for color. However, it must be noted that the spectral discrimination properties of the human visual system vary across the visible spectrum, as shown in Fig. 2.7. From an anatomical and physiological point of view, the following two aspects must be considered:

The eye initially has three different sensitivity functions and cone types in the near-sighted (photopic) area, which are reminiscent of the well-known three-color systems from video technology. However, the neuronal coding in the ganglion cells is antagonistic in the sense of opposite colors red-green and blue-yellow. This is illustrated in Fig. 2.9. The aim is therefore to find a system of measurement for colors that takes into account the physical and physiological conditions mentioned.

2.4 The norm valence system of the CIE

2.4.1 CIE standard valence curves

In 1931 and 1964, the *Commission Internationale de l'Eclairage* (CIE) determined the so-called norm valence system through experiments.

Mixed colors were generated from three primaries $\lambda_1 = 435.8nm$, $\lambda_2 = 546.1nm$ and $\lambda_3 = 700.0nm$ and compared with pure spectral colors by test persons. The aim of the experiment was to find out whether all spectral colors can be additively mixed using three different primary colors. Fig. 2.10 shows the experimental arrangement. The following curves were obtained as a result of the experiment: It can be seen that the function $\overline{r}(\lambda)$ assumes negative values in a certain spectral range. There are therefore spectral colors that cannot be generated purely additively from the three primary valences R, G, B. This is a result which is independent of the

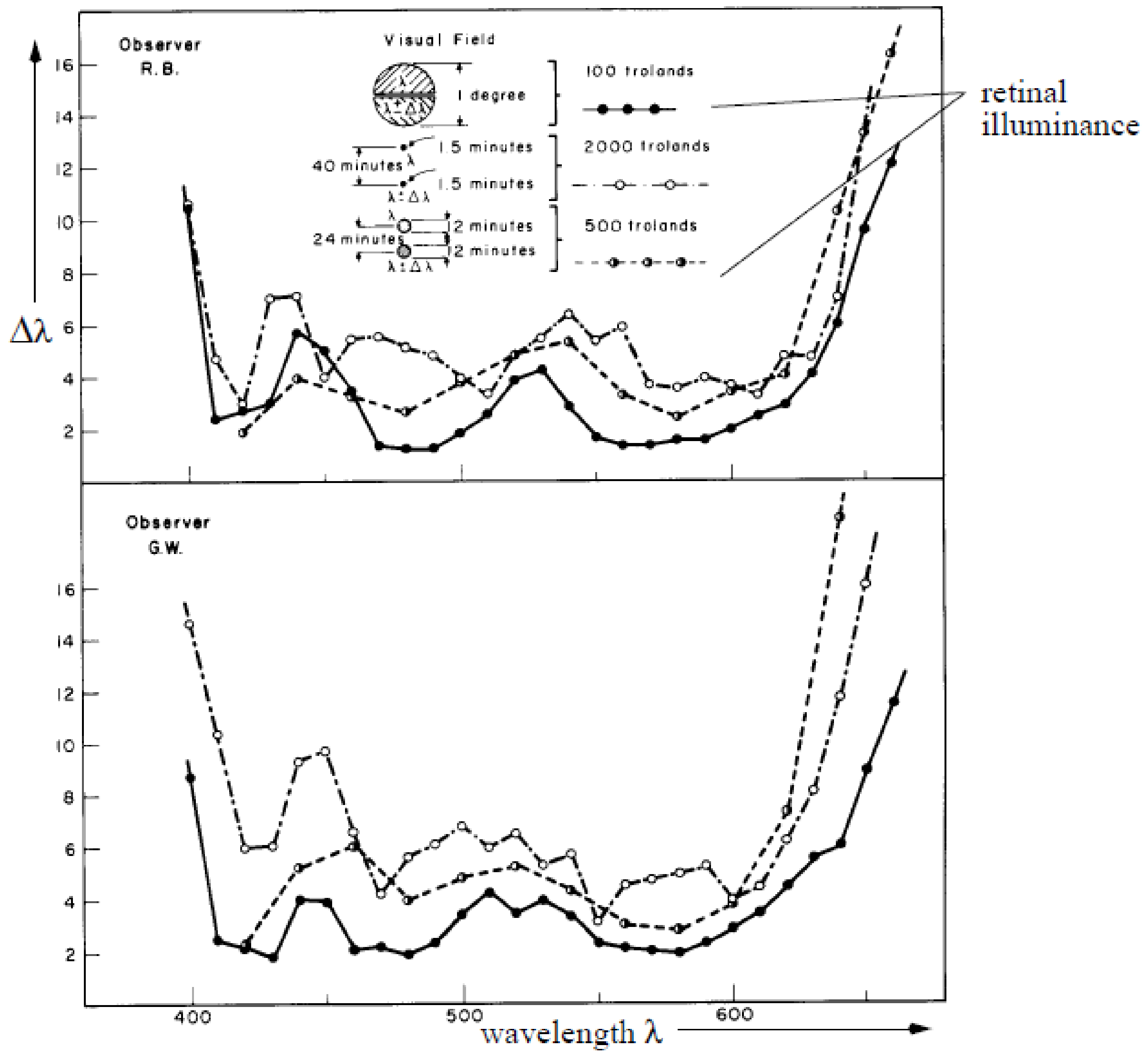


Figure 2.7: *Spectral discrimination properties*

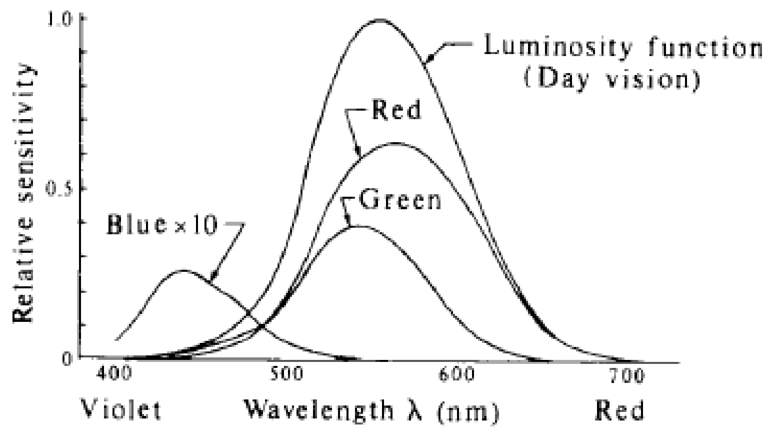


Figure 2.8: *Relative spectral sensitivity of the human eye*

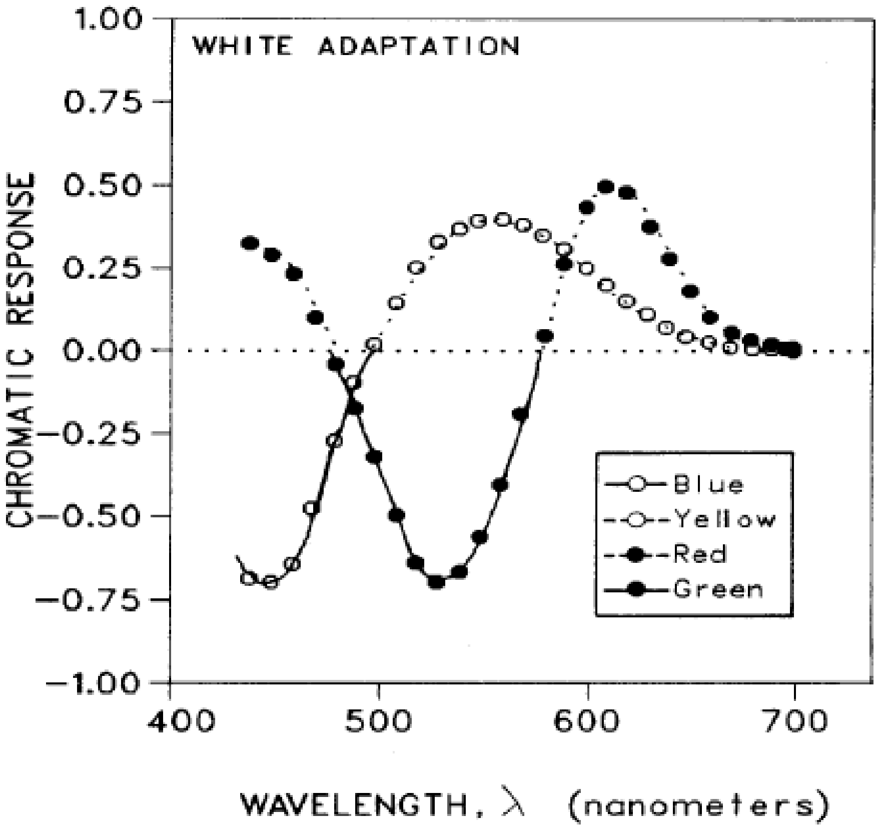


Figure 2.9: *Neural coding of color perception*

chosen wavelengths of the three primary colors. The width and location of the negative area are affected, but not the fact that there is one. The three primaries are therefore replaced by the CIE *standard primaries* or norm valences *X, Y, Z*, which allow purely additive generation of all visible colors. The boundary conditions are:

- positivity of the curves

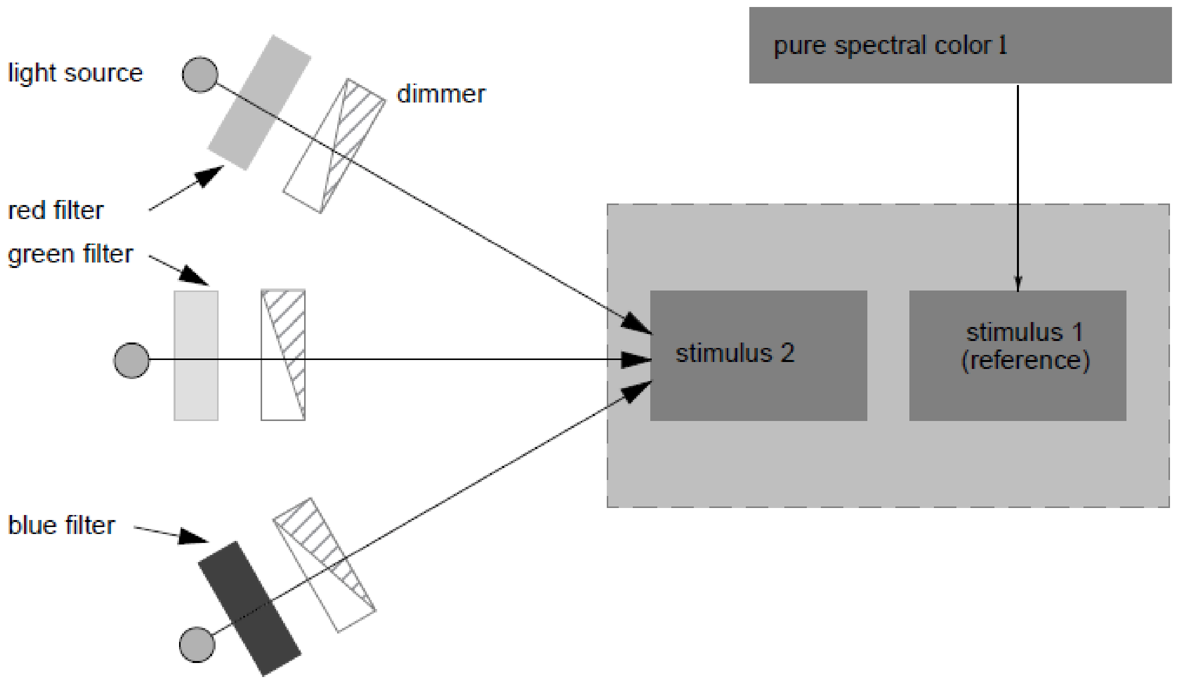


Figure 2.10: *Schematic test setup for determining the standard valence curves of the CIE 1931*

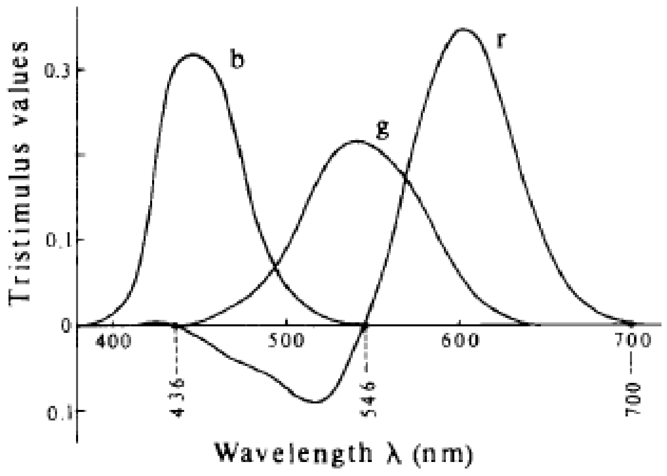


Figure 2.11: *Spectral evaluation functions* $\overline{r}, \overline{g}, \overline{b}$ *for the primary valences R,G,B*

- Energy normalization by equal areas under the curves
- Equality of $y(\lambda)$ and $V(\lambda)$

A normalization transformation results in the normalized spectral weighting functions $\overline{x}_{10}(\lambda)$, $\overline{y}_{10}(\lambda)$, $\overline{z_{10}}(\lambda)$ for the norm valences *X, Y, Z*:

$$\overline{x}_{10}(\lambda) = 0.341\overline{r}(\lambda) + 0.189\overline{g}(\lambda) + 0.388\overline{b}(\lambda) \tag{2.6}$$
$$\overline{y}_{10}(\lambda) = 0.139\overline{r}(\lambda) + 0.837\overline{g}(\lambda) + 0.073\overline{b}(\lambda) \;\; \overline{z}_{10}(\lambda) \;\; = 0.000\overline{r}(\lambda) + 0.040\overline{g}(\lambda) + 2.062\overline{b}(\lambda)$$

The resulting curves are shown in Fig. 2.12.

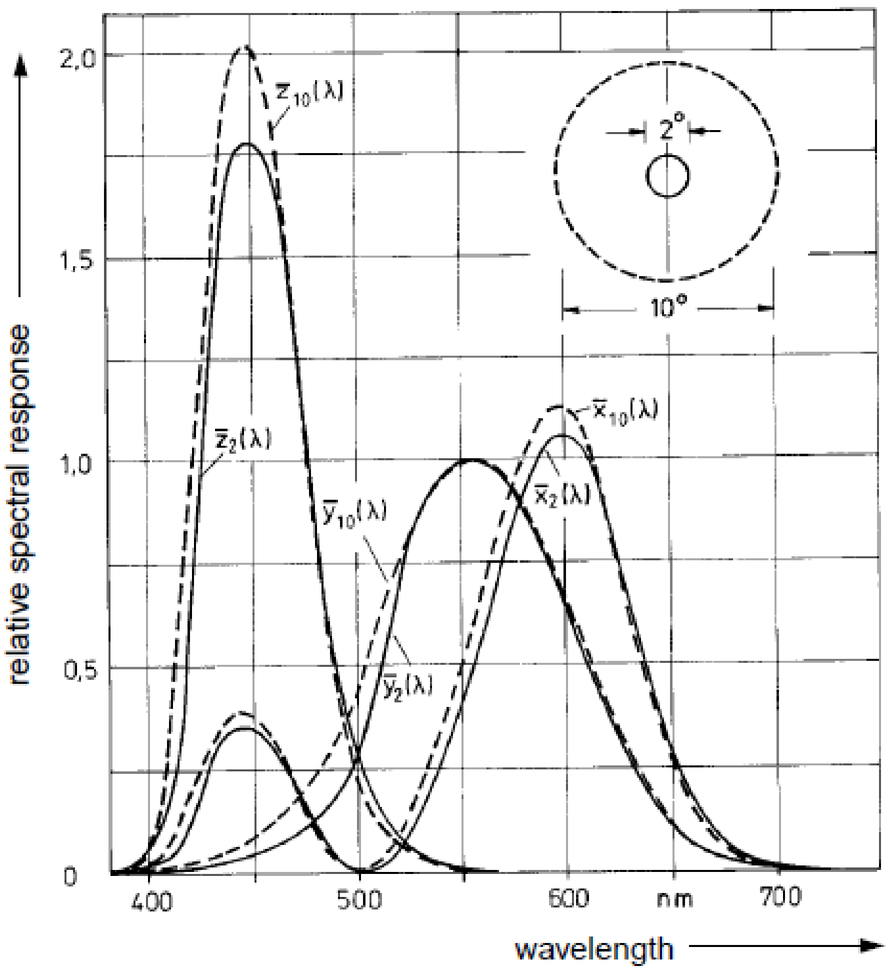


Figure 2.12: *Normalized spectral weighting functions x, y, z for the standard primaries X, Y, Z*

$\overline{y}_2(\lambda)$ corresponds to the light perception $V(\lambda)$ (sensitivity of the eye to light of the same intensity of different wavelengths, see Fig. 2.12). Thus, each color of a self-luminous object can be described by a triple *(X, Y, Z)* using its spectral power density distribution $P(\lambda)$.

$$X = \int_{380nm}^{780nm} P(\lambda)\overline{x}(\lambda)\, d\lambda \tag{2.7}$$
$$Y = \int_{380nm}^{780nm} P(\lambda)\overline{y}(\lambda)\, d\lambda$$
$$Z = \int_{380nm}^{780nm} P(\lambda)\overline{z}(\lambda)\, d\lambda$$

This allows a geometric interpretation in a three-dimensional Euclidean coordinate system: The color locus is described as a vector $\mathrm{f} = (X, Y, Z)$ **and can be clearly identified. The addition of two colors**

f_1 and f_2 is simply a vector addition in space: $f_1+f_2 = (X_1+X_2, Y_1+Y_2, Z_1+Z_2)$

2.4.2 The CIE Chart

A two-dimensional chart is much better suited for the practical identification of colors, whereby the chromaticity should be processed separately from the brightness (luminance). For this purpose, a further normalization is carried out, which supplies the following variables:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z} \qquad z = 1 - xy \tag{2.8}$$

This leads to the famous CIE chart, which is shown in Fig. 2.13 with some typical color locations. It plots x over y and allows each color to be uniquely characterized. The horse-shoe delimiting the range of visible colors results from the projection of pure spectral colors into the diagram.
Other characteristic features of the chart are:

- white point
- Isolines of saturation or whiteness of different colors
- Isolines of the chromaticity
- Planckian radiator and color temperature: This curve has a characteristic progression from the deep red edge of the chart to the white point and into the weakly saturated blue.
- Dominant wavelength
- Purple Line

Areas of similarly perceived color tones are shown in Fig. 2.14.

Modern graphics monitors or video cameras often allow the color temperature to be calibrated. The coordinates of the white point are associated with the color temperature. The following temperatures and coordinates are important:
A: light bulb, 0.448, 0.408
B: Sun in the afternoon, 0.349, 0.352
C: Overcast, 0.310, 0.316b
D65: Sun, 0.310, 0.316
Sony monitor: optionally 9300 K, 6500 K, 5000 K

It should be noted that the transformation into the CIE chart can be viewed as a projection of the position vector *f* of a color value into a plane $X+Y+Z=1$. This plane is perpendicular to the space diagonal in the three-dimensional Euclidean XYZ system.

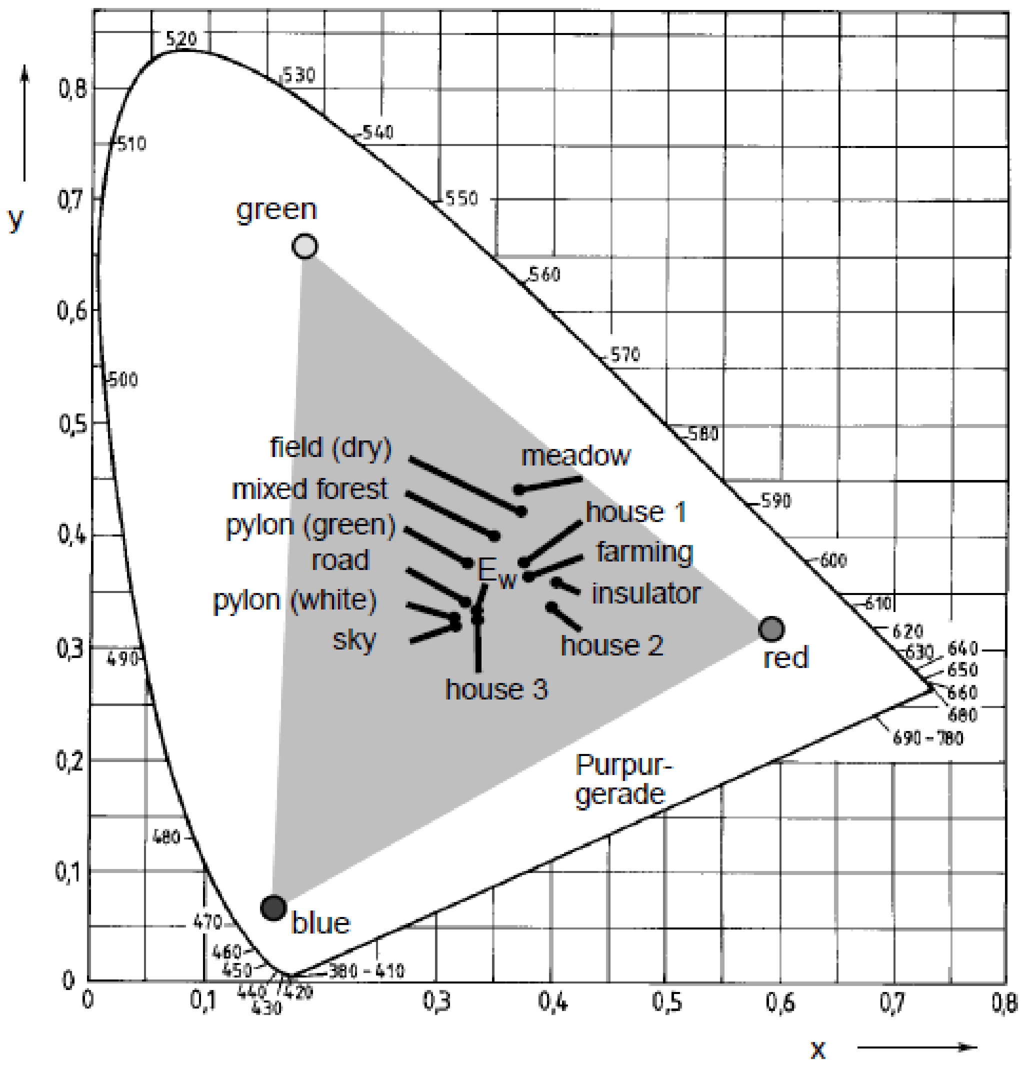


Figure 2.13: *CIE chart with some characteristic color locations*

2.4.3 Important color spaces for practice

The color coordinates of screen phosphors are also defined by the CIE chart. These are uniform for different television standards. The resulting areas (triangles) for various practical cases are drawn in Fig. 2.15. Due to the vector representation of individual colors and the resulting mixing laws as vector additions, it can be seen that every mixed color that is additively composed of three primary valences must always lie within the inscribed triangle. Therefore, pure spectral colors cannot always be generated by additive mixing (barycentric coordinates in the second part of the script).
For typical phosphorus coordinates one finds, for example:

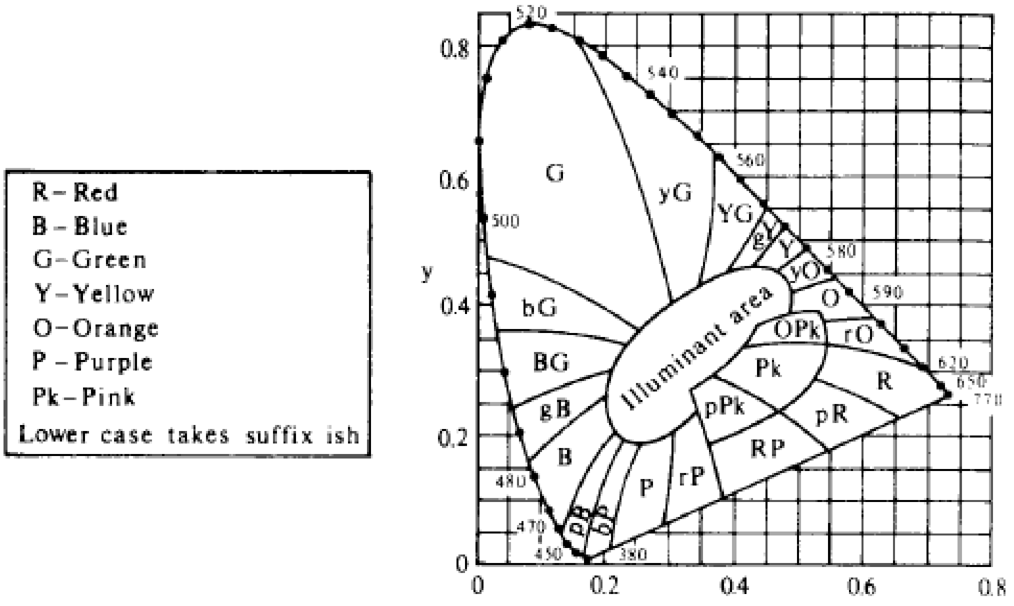


Figure 2.14: *Nomenclature for areas of similar color types in the CIE chart*

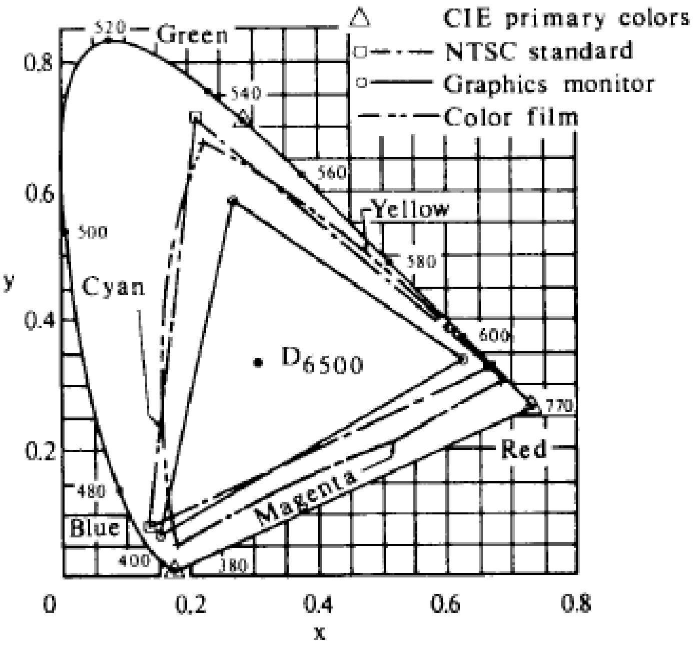


Figure 2.15: *color coordinates in the CIE chart*

short-luminous phosphors
(short-presistance phosphors)

	Red	**Green**	**Blue**
x	0.61	0.29	0.15
y	0.35	0.59	0.063

long-luminous phosphors
(long- persistence phosphors)

	Red	**Green**	**Blue**e
x	0.62	0.21	0.15
y	0.33	0.685	0.063

Each point (X, Y, Z) can be transformed to *(R, G, B)* by inverting the matrix in equation (2.9):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_R & X_G & X_B \\ Y_R & Y_G & Y_B \\ Z_R & Z_G & Z_B \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.9}$$

2.4.4 Monitor calibration

A basic problem of photorealistic visualization is the reproduction of colors on a monitor. If the spectral power density distributions of sources or surfaces are known from a calculation method, they can first be transformed into the standard valence system.
If the phosphorus coordinates $(x_R, y_R), (x_G, y_G)\text{and}(x_B, y_B)$ are given, X, Y, Z can be found with the transformation

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_R C_R & x_G C_G & x_B C_B \\ y_R C_R & y_G C_G & y_B C_B \\ (1 - x_R - y_R) \cdot C_R & (1 - x_G - y_G) \cdot C_G & (1 - x_B - y_B) \cdot C_B \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.10}$$

The unknowns C_R, C_G, C_B result from a known white point $(X_\omega, Z_\omega, Z_\omega)$ for $R = G = B = 1$

$$\begin{bmatrix} X_\omega \\ Y_\omega \\ Z_\omega \end{bmatrix} = \begin{bmatrix} x_R & x_G & x_B \\ y_R & y_G & y_B \\ (1 - x_R - y_R) & (1 - x_G - y_G) & (1 - x_B - y_B) \end{bmatrix} \cdot \begin{bmatrix} C_R \\ C_G \\ C_B \end{bmatrix}$$

This allows screens that operate in an RGB system to be calibrated. Incidentally, the second line of the above matrix corresponds to an equation for extracting gray levels from color images.

2.5 color spaces

2.5.1 RGB color space

The standard valence system of the CIE is a very technical framework, which is used in particular for exact reproductions in the paint, printing or textile industry. Since monitors usually work with RGB, the additive RGB system is often used in graphic data processing. The coordinates in the RGB color space are normalized to $[0, 1]$.

2.5.2 CMY color space

Complementing this is the subtractive CMY system, which is frequently used in printing technology in particular, in which cyan, magenta and yellow represent the primary valences. The

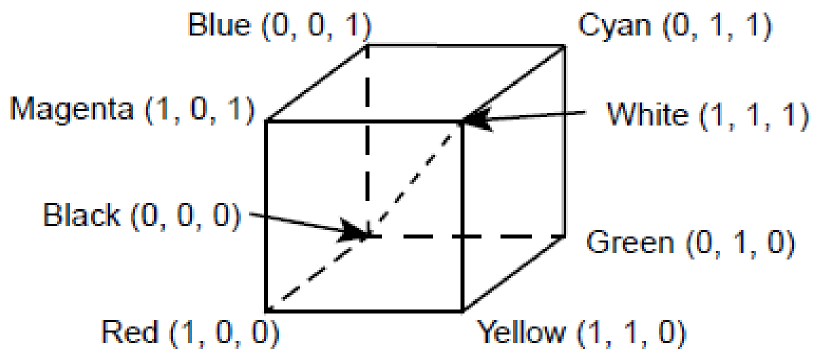


Figure 2.16: *RGB color space*

following applies:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \text{ or } \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \tag{2.11}$$

In contrast to the RGB system, vectorial additions of individual colors result in a subtraction of brightness.

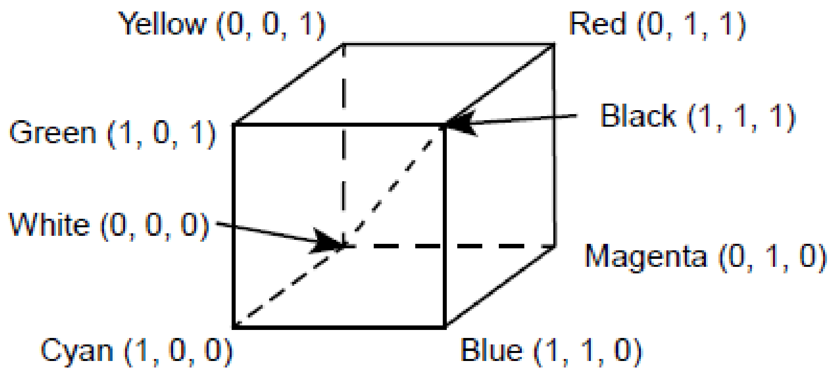


Figure 2.17: *CMY color space*

2.5.3 HSV color space

Due to the difficulty of mixing colors in RGB, spaces have been defined that are based on natural criteria such as *Hue* (hue), *Saturation* (saturation) and *Value* (brightness). The color spaces described in this way can be found in various variants and are now part of almost all presentation software or window systems. For example, you get a hexagonal color body as shown in Fig. 2.18.

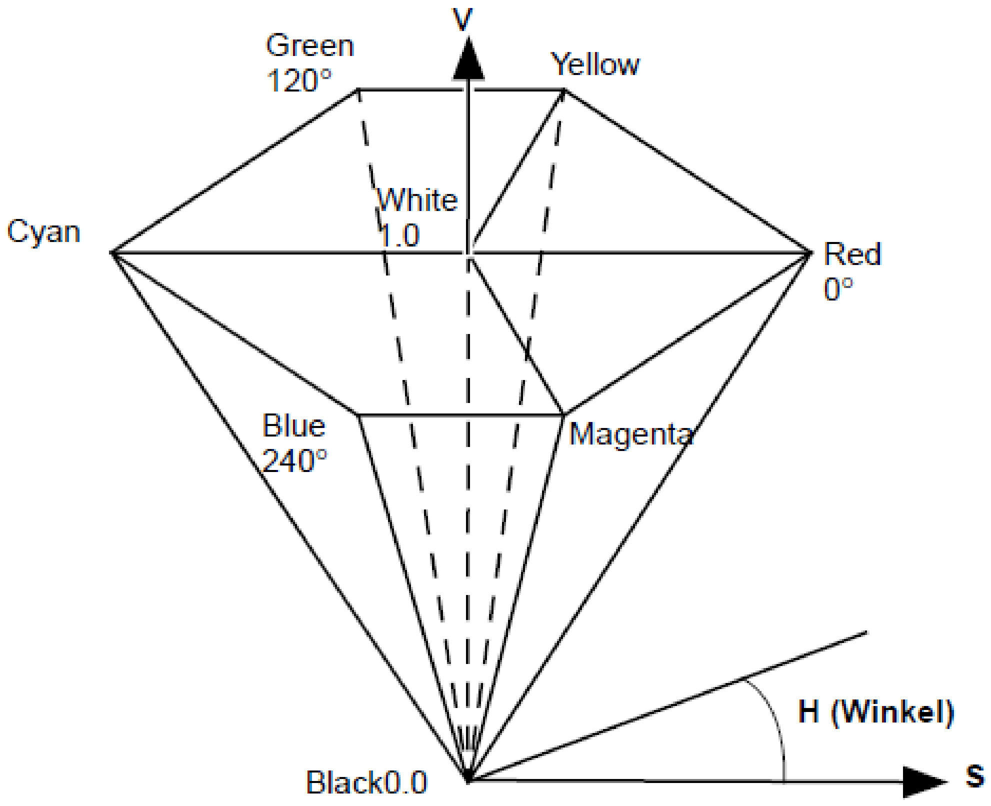


Figure 2.18: *HSV color space*

All possible color types of the CIE chart are mapped to a 360° color circle. The color wheel is divided into six fields, which represent the hues in the order of the CIE chart. For $V = 0$ the values for S and H are irrelevant, for $S = 0$ H is irrelevant, while V determines the gray value. Example: Conversion from RGB to HSV with $R, G, B, S, V \in [0, 1]$ and $H \in [0^\circ, 360^\circ]$

```
/* determine value */
v = max(r, g, b);
/* determine saturation */
temp = min(r, g, b);
if (v == 0)
        s = 0;
else
        s = (v – temp)/v;
/* determine hue */
if (s == 0)
        h = −1; /* undefined */
else {
        cr = (v – r)/(v – temp);
        cg = (v – g)/(v – temp);
        cb = (v – b)/(v – temp);
        if (r == v)
```

```
            h = cb − cg;
            /∗ color between yellow and magenta ∗/
    if (g == v)
            h = 2 + cr − cb;
            /∗ color between cyan and yellow ∗/
    if (b == v)
            h = 4 + cg − cr;
            /∗ color between magenta and cyan ∗/
    h = 60∗h;   /∗ convert to degrees ∗/
    if (h < 0)
            h += 360;   /∗ prevent negative value ∗/
}
```

Example: Conversion from HSV to HSV with $R, G, B, S, V \in [0, 1]$ and $H \in [0°, 360°]$

```
if (s == 0)  {  /∗ achromatic case ∗/
    if (h == −1)   /∗ undefined ∗/
            r = g = b = v;
    else
            error;
}
else {   /∗ chromatic case ∗/
    if (h == 360) h = 0;
    h /= 60;
    i = floor(h); f = h − i;
    m = v∗(1 − s);
    n = v∗(1 − s∗f);
    k = v∗(1 − s∗(1 − f));
    if (i == 0) r = v, g = k, b = m;
    if (i == 1) r = n, g = v, b = m;
    if (i == 2) r = m, g = v, b = k;
    if (i == 3) r = m, g = n, b = v;
    if (i == 4) r = k, g = m, b = v;
    if (i == 5) r = v, g = m, b = n;
}
```

2.5.4 YIQ color space

The YIQ color space is the coding system of the NTSC color television standard. The total bandwidth of around 4.5 MHz is divided up in such a way that 2.4 MHz is available for the luminance, 1.5 MHz for the in-phase component (orange-cyan, skin colors) and 0.6 MHz (green-magenta) for the quadrature component. This is based on the psychophysical properties of the human visual system.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.12}$$

The phosphor has the following color coordinates in NTSC:

	Red	**Green**	**Blue**
x	0.67	0.21	0.14
y	0.33	0.71	0.08

2.5.5 Perception-oriented color spaces

A fundamental problem in colorimetry is calculating the distance between two colors. It would be obvious to calculate the Euclidean distance of the individual color locations in the standard valence system. However, an experiment by *McAdams* has shown that the differentiation thresholds for individual colors are distributed ellipsoidally around the color coordinates in the CIE chart. For this purpose, certain color locations (centers of the ellipses) had to be mixed by two colors whose connecting line ran through the color to be mixed. The resulting lack of focus when setting the mixing ratio led to the ellipsoidal differentiation thresholds shown in Fig. 2.19. These in turn depend on the selected adaptation level and on the position in the chart.
Based on this knowledge, attempts have now been made to rectify the ellipses as much as possible into circles of the same radii. This has led to a whole series of different, so-called uniform color spaces, which emerge from the norm valence system by means of heuristic non-linear transformations. In those color spaces, metric distances correspond to perceived ones.

*La*b* color space*

$$\begin{aligned} L^* &= 25\left[\frac{100Y}{Y_\omega}\right]^{1/3} - 16 \\ a^* &= 500\left[\left(\frac{X}{X_\omega}\right)^{1/3} - \left(\frac{Y}{Y_\omega}\right)^{1/3}\right] \\ b^* &= 200\left[\left(\frac{Y}{Y_\omega}\right)^{1/3} - \left(\frac{Z}{Z_\omega}\right)^{1/3}\right] \end{aligned} \tag{2.13}$$

$(X_\omega, Y_\omega, Z_\omega)$: Coordinates of the white point

*La*v* color space*

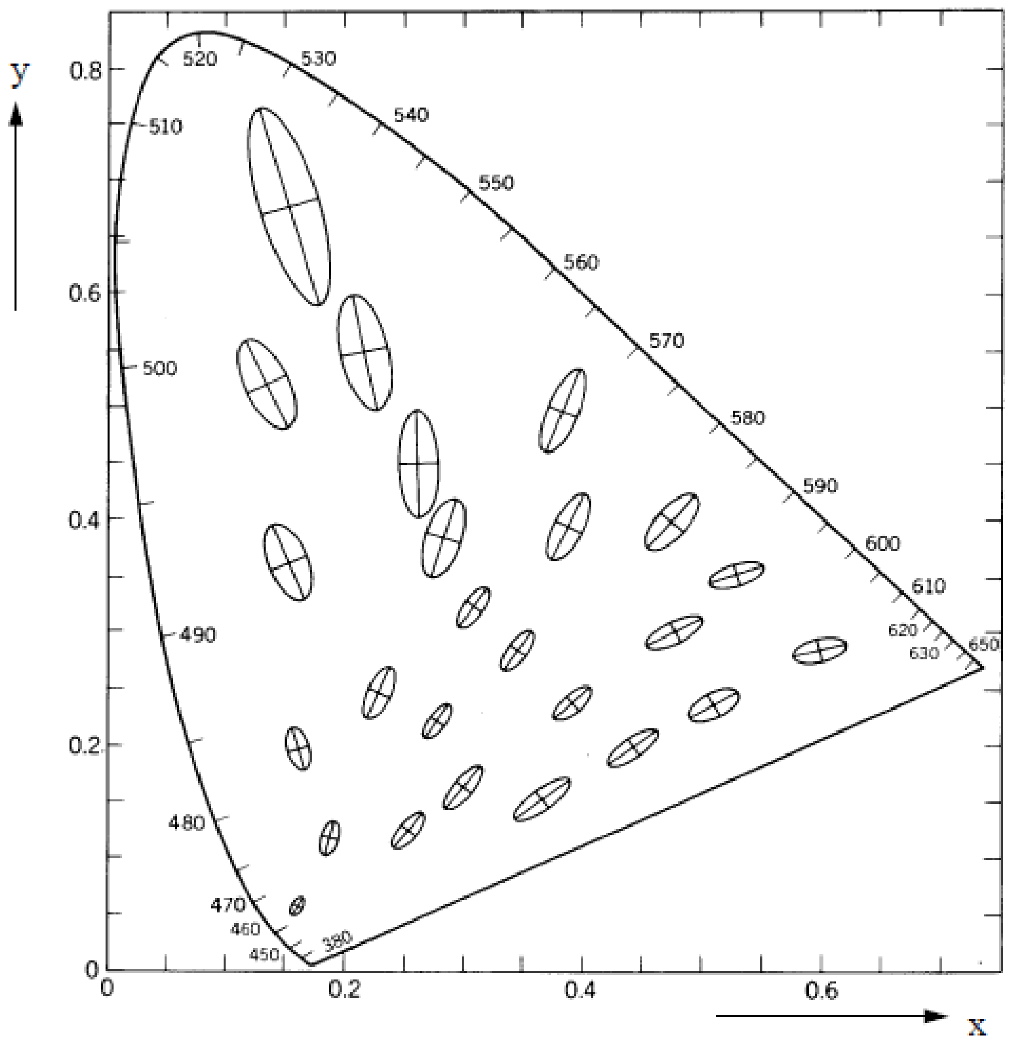
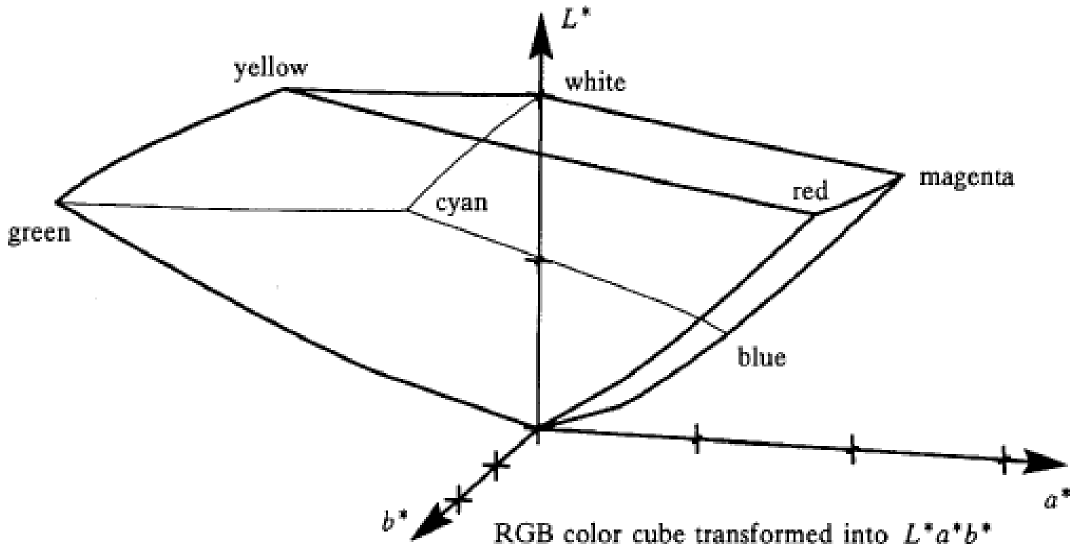


Figure 2.19: *Distinction thresholds as ellipses in the McAdams experiment for different color locations (scaled 10 times)*

$$
\begin{aligned}
u &= \frac{4X}{X + 15Y + 3Z} \\
v &= \frac{9X}{X + 15Y + 3Z} \\
L^* &= 25\sqrt[3]{\frac{100Y}{Y_\omega}} - 16 \\
u^* &= 13L^*(u - u_\omega) \\
v^* &= 13L^*(v - v_\omega) \\
&(Y_\omega, u_\omega, v_\omega)\text{: \textit{coordinates of the white point}}
\end{aligned} \tag{2.14}
$$

The last two color systems *La*b** and *Lu*v** cause non-linear distortions, which are shown in Fig. 2.20 for the RGB and CMY color spaces.



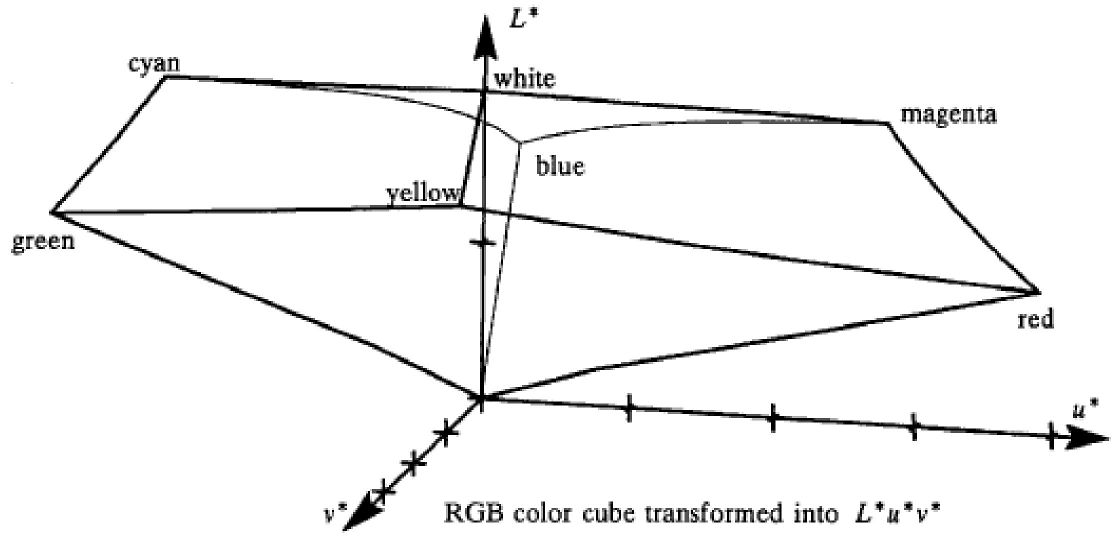


Figure 2.20: *Nonlinear distortions of the La*b* and Lu*v* color spaces*

Geometric transformations

3.1 Introduction

Geometric primitives can be moved, rotated and scaled using 2D and 3D transformations. This allows complex scenes composed of individual primitives to be generated and processed. The mathematical basics of 2D and 3D transformations are discussed below. In particular, the concept of *homogeneous coordinates* is of central importance.

First some definitions of terms:

1. A mapping $A : x \rightarrow x'$ is called *linear* if:

$$A(\alpha x + \beta y) = \alpha A(x) + \beta A(y) \tag{3.1}$$

   If a vector **x** is multiplied by a matrix **A**, then $\mathbf{A} \cdot \mathbf{x} = \mathbf{x}'$ is a linear equation.

2. A mapping $B : x \rightarrow x'$ is called *affine* if it is of the form

$$x' = A(x) + t = B(x) \tag{3.2}$$

   enough. That is, an affine mapping B can be decomposed into a linear mapping A and into a translation by the vector **t**.

3.2 2D transformations

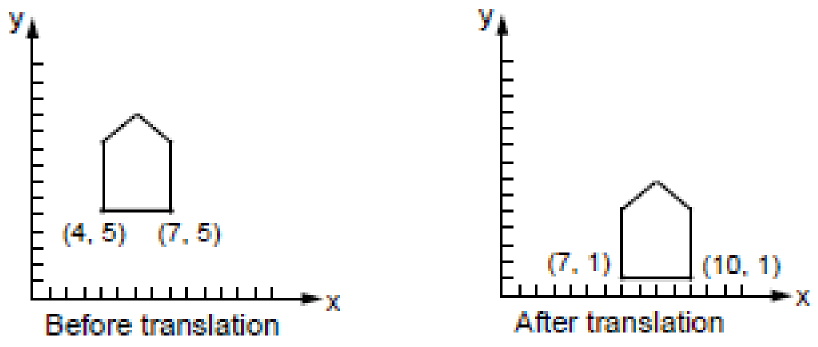
3.2.1 Translation

The displacement of a point **P**(x, y) by *(dx, dy)* into a point **P** *(x', y')* is described by:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix} \Rightarrow \mathbf{P'} = \mathbf{P} + \mathbf{T} \tag{3.3}$$

An object is translated by applying the above equation to all vertices.

Example: Translation by (3, -4):



3.2.2 Scaling

If the point **P**(x, y) is to be scaled by *(sx, sy)*, the following applies:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \mathbf{P'} = \mathbf{S} \cdot \mathbf{P} \tag{3.4}$$

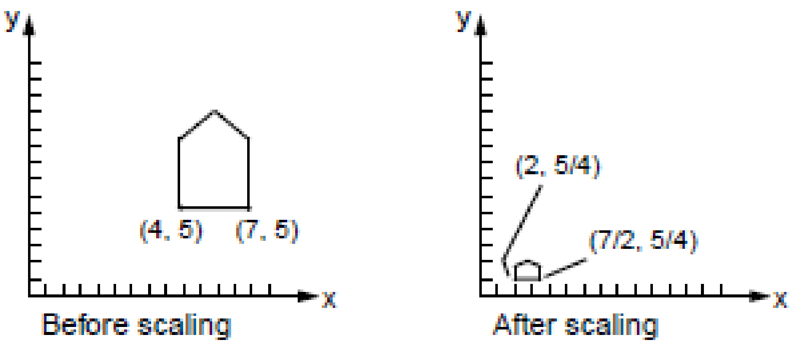
Example: $s_x = 1/2, s_y = 1/4,$

3.2.3 Rotation

If a point **P** is to be rotated by the angle θ, the following applies:

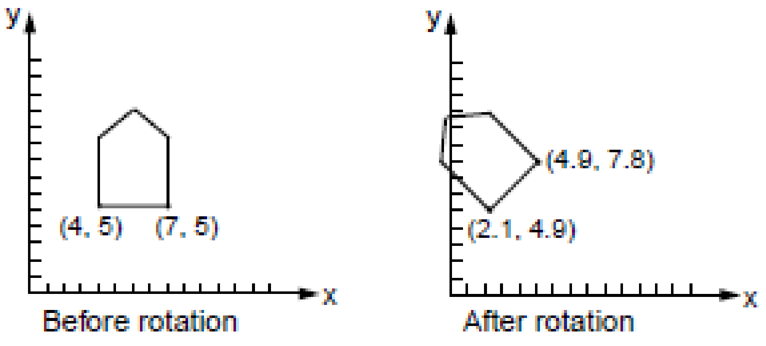
$$x' = x \cdot cos\ \theta - y \cdot sin\ \theta, y' = x \cdot sin\ \theta + y \cdot cos\ \theta \tag{3.5}$$

or



$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\,\theta & -sin\,\theta \\ sin & \\ theta & cos \\ theta & \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \mathbf{P'} = \mathbf{R} \cdot \mathbf{P} \tag{3.6}$$

R is the 2D rotation matrix. Angles are defined mathematically positive. This equation is derived using simple, trigonometric relationships.
Example: Rotation around $\frac{\pi}{4}$ The center of rotation is the origin of the coordinates



A rotation around any center requires the concatenation of several individual transformations.

3.2.4 Homogeneous coordinates

The matrix notation of the three presented transformations are:

$$\begin{aligned} P' &= T + P \\ P' &= S \cdot P \\ P' &= R \cdot P \end{aligned} \tag{3.7}$$

The translation is treated as an addition. As can easily be seen, the sum of the individual mappings describes an affine mapping. Uniform treatment as multiplication is desirable. For

this purpose, the so-called *homogeneous coordinates* are introduced. They can be interpreted simply as adding an extra dimension to the vectors.
Points **P** in 2D are provided with an additional coordinate W.

$$\mathbf{P} = (x, y, W) \tag{3.8}$$

Every point in 2D thus has an infinite number of homogeneous coordinates, depending on the choice of W.

P = (1, 2) can be written as (1, 2, 1) or (3, 6, 3) or (-4, -8, -4) etc. ($W \neq 0$).
General:

- **P** = (x,y,W) and **P'** = $\left(\frac{x}{W}, \frac{y}{W}, 1\right)$ are equal points in 2D
- $\left(\frac{x}{W}, \frac{y}{W}\right)$ are the Cartesian, two-dimensional coordinates of the point **P** in the homogeneous coordinate system.

meaning: Since points obtained from one another by multiplication by a scalar t in homogeneous coordinates are equal, each point **P**(x, y) can be viewed as a straight line in the three-dimensional space of homogeneous coordinates.

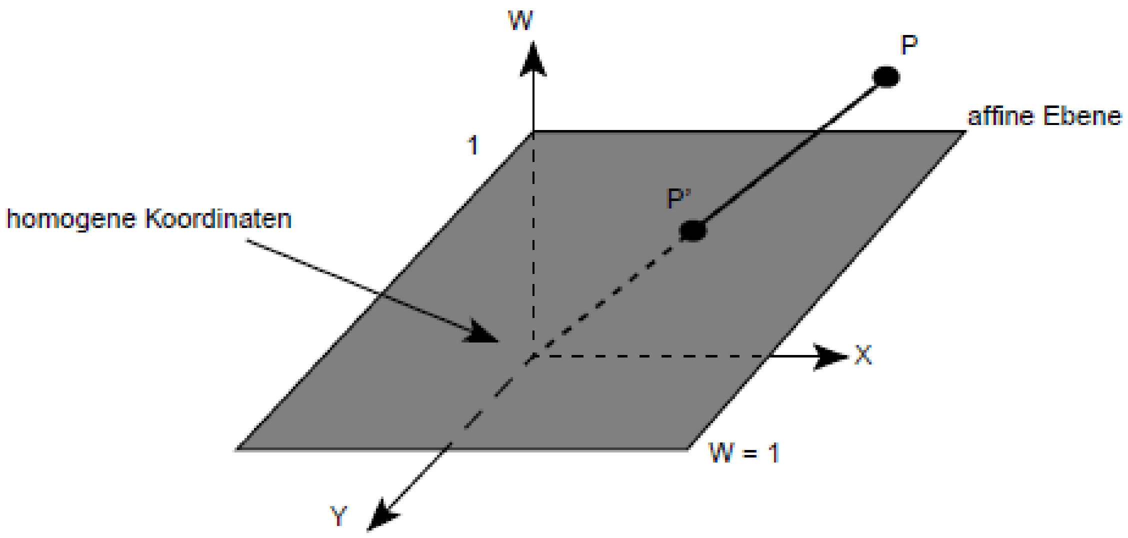


Figure 3.1: *interpretation of the two-dimensional homogeneous coordinates in 3D*

By *homogenizing* (division by W) the point $\mathbf{P} = \left(\frac{x}{W}, \frac{y}{W}, 1\right)$ results, which lies in the $W = 1$ plane. This allows using a 3x3 matrix **T** of the form

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \Rightarrow \mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \tag{3.9}$$

to describe the translation

A point in homogeneous coordinates is therefore a straight line in a higher-dimensional space. A division by the homogeneous coordinate W corresponds to a projection into the plane W=1.

Example: Successive translation by $T(t_{x_1}, t_{y_1}$ and $T(t_{x_2}, t_{y_2}$. The following applies:

$$P' = T(t_{x_1}, t_{y_1}) \cdot P$$
$$P'' = T(t_{x_2}, t_{y_2}) \cdot P'$$

and thus:

$$P'' = T(t_{x_2}, t_{y_2}) \cdot (T(t_{x_1}, t_{y_1}) \cdot P) = ((t_{x_2}, t_{y_2})\ cdotT(t_{x_1}, t_{y_1})) \cdot P$$

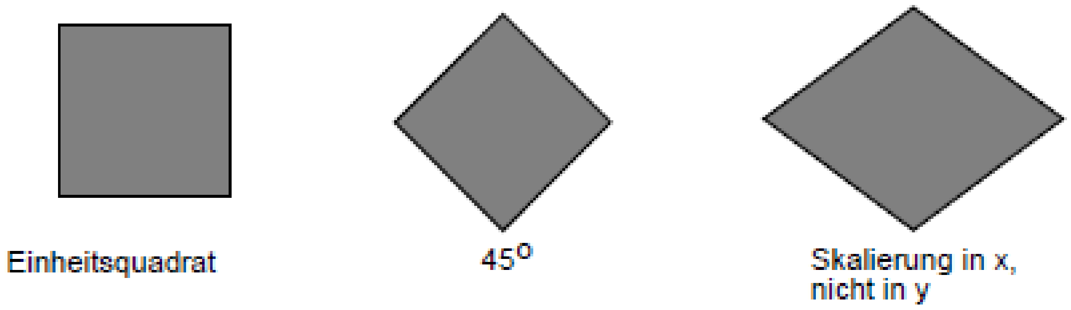
In matrix notation:

$$\begin{bmatrix} 1 & 0 & t_{x_2} \\ 0 & 1 & t_{y_2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x_1} \\ 0 & 1 & t_{y_1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0t_{x_1} + t_{x_2} & \\ 0 & 1 & t_{y_1} + t_{y_2} \\ 0 & 0 & 1 \end{bmatrix}$$

You can see the addition of the individual translation components, despite the matrix product notation used.

If a concatenation of individual transformations also includes non-uniform scaling operations, the result is an affine overall transformation that maintains the parallelism of lines, but not lengths and angles.

Example: Successive rotation and non-uniform scaling (s_x, s_y) of a square:



3.2.5 Shear

Shearing can be done separately along the x-axis or the y-axis.
The *shear matrix in x* is defined as follows:

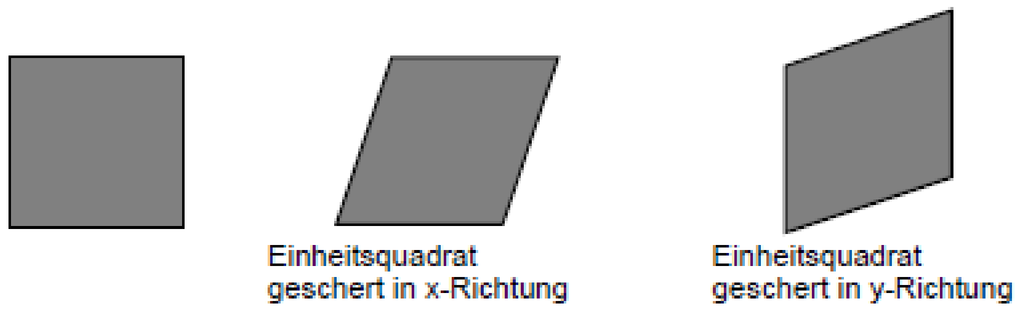
$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Similarly, the shear matrix in y is given by:

$$SH_y = \begin{bmatrix} 1 & 0a & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that $SH_x \times (x, y, 1)^T = (x + ay, y, 1)^T$ and $SH_y \times (x, y, 1)^T = (x, y + bx, 1)^T$ is.

Example: Shear in x and y direction:



The shear transformation is also linear.

3.2.6 Scaling and rotation in homogeneous coordinates

As with translation, the multiplication properties of scaling and rotation can be demonstrated.
The following relationships are given:

$$P' = S(s_{x_1}, s_{y_1}) \cdot P$$
$$P'' = S(s_{x_2}, s_{y_2}) \cdot P'$$

Substituting the first equation into the second, we get

$$P'' = S(s_{x_2}, s_{y_2}) \cdot (S(s_{x_1}, s_{y_1}) \cdot P) = (S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1})) \cdot P$$

In matrix notation follows

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

One recognizes the multiplication of the scaling parameters, where the *scaling matrix* is the form

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

assumes For the rotation matrix one finds

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

assumes For the *rotation matrix* one finds

$$R(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

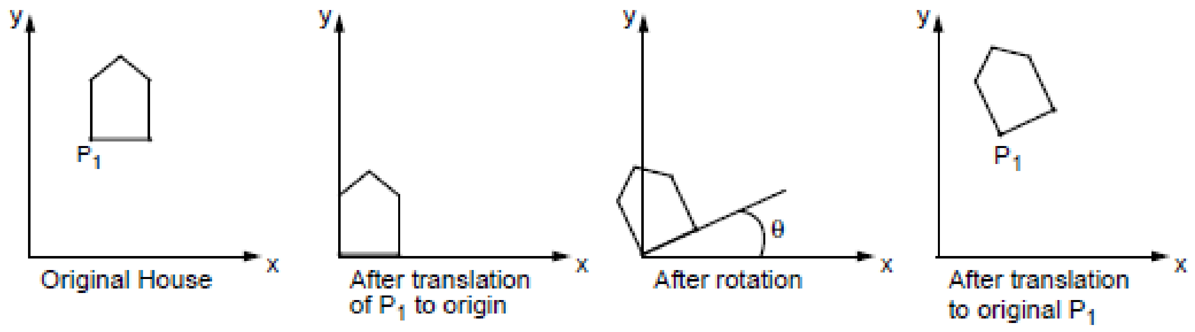
A $T \cdot R$ matrix of the form

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

describes a *rigid-body transformation* (rigid body), because the shape of an object (length and angle relationships) is preserved.

3.2.7 Concatenation of 2D transformations

To increase efficiency in computer-aided applications, it makes sense to chain elementary transformations. *Example:* Rotation of an object around the point $P_1(x_1, y_1)$

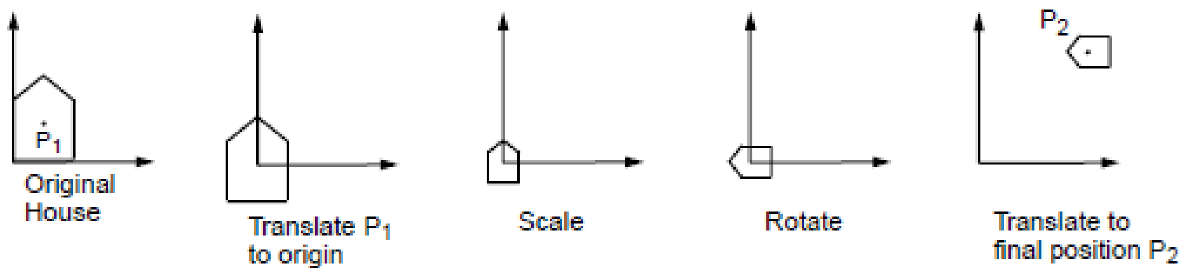

y
x
P1
Original House
After translation of P1 to origin
θ
After rotation
After translation to original P1

Successive execution of a translation from P_1 to the origin, a rotation and subsequent back transformation becomes in matrix form

$$T(x_1,y_1)\times R(\theta)\times T(-x_1,-y_1)=\begin{bmatrix}1&0&x_1\\0&1&y_1\\0&0&1\end{bmatrix}\cdot\begin{bmatrix}cos\theta&-sin\theta&0\\sin\theta&cos\theta&0\\0&0&1\end{bmatrix}\cdot\begin{bmatrix}1&0&-x_1\\0&1&-y_1\\0&0&1\end{bmatrix}$$

$$=\begin{bmatrix}cos\theta&-sin\theta&x_1(1-cos\theta)+y_1sin\theta\\sin\theta&cos\theta&y_1(1-cos\theta)-x_1are\theta\\0&0&1\end{bmatrix}$$

Example: Concatenation of translation, rotation and scaling



In this case, the following operations must be applied to each vector in homogeneous coordinates. The order of the operations must be observed.

$$T(x_2,y_2)\times R(\theta)\times S(s_x,s_y)\times T(-x_1,-y_1)$$

commutativity of two matrices M_1 and M_2 is only given if

M_1	M_2
Translation	Translation
Scaling	Scaling
Rotation	Rotation
Scaling	Rotation

Conclusion: Affine transformations (rotation, scaling, shearing, translation) can be concatenated as matrix products in homogeneous coordinates.

3.3 Coordinate systems

3.3.1 World, object and camera coordinates

- *world coordinates:* Coordinate system that describes the entire scene (world) in the units of measurement specified by the application. These can be given in meters, light years, but also seconds, amperes, etc. for visualization applications.
- *Object coordinates:* Coordinate system in which an individual object is described according to its construction. To place objects in the world, the object coordinates must be transformed into world coordinates.
- *Camera coordinates:* Coordinate system that has its origin in the projection center of the camera. It is of fundamental importance for image generation.

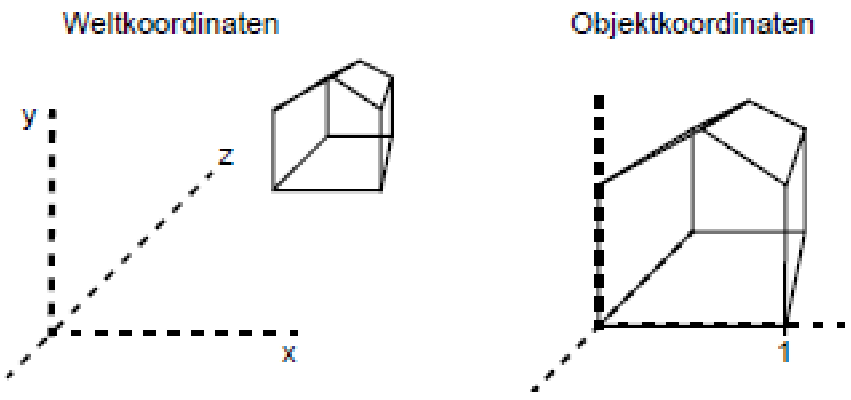


Figure 3.2: *World and object coordinates*

3.3.2 Windows and Viewports

To map a scene in 2D, a *window* is generally first defined in world coordinates and this is then mapped into the *viewport*, a coordinate system of the output device.

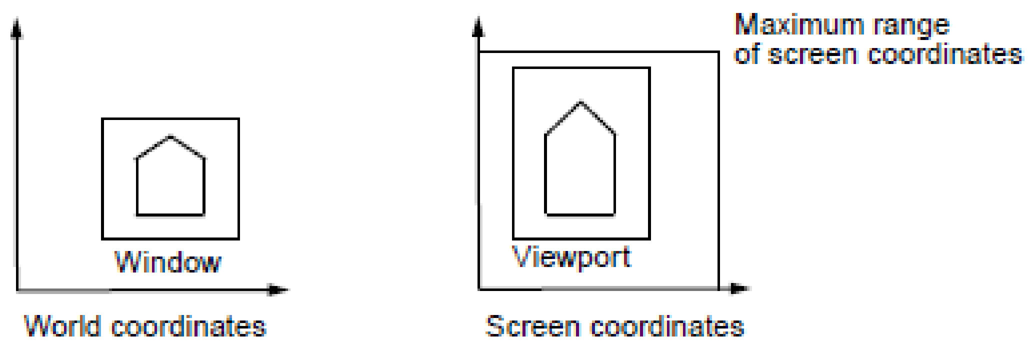


Figure 3.3: *Relationship between world and screen coordinates, windows and viewports*

Therefore, depending on the *aspect ratio* (ratio of height to width of the viewport), non-uniform scaling is performed. The same windows can also be transformed into different viewports (Fig. 3.4).

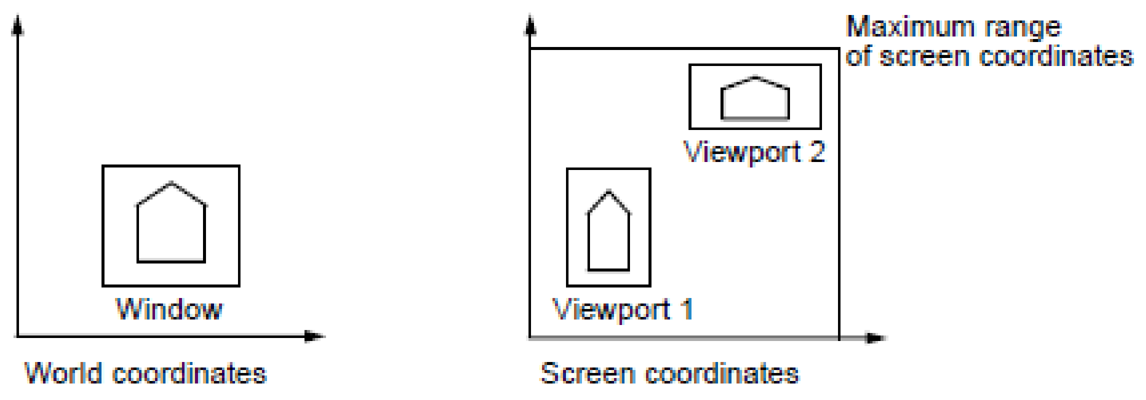


Figure 3.4: *Non-uniform scaling and multiple viewports for the same window*

3.3.3 Calculation bases

The transformation *Window* $\rightarrow$ *Viewport* is a 3-step process consisting of a translation, subsequent scaling and translation again.

The overall matrix of this figure in homogeneous coordinates results in:

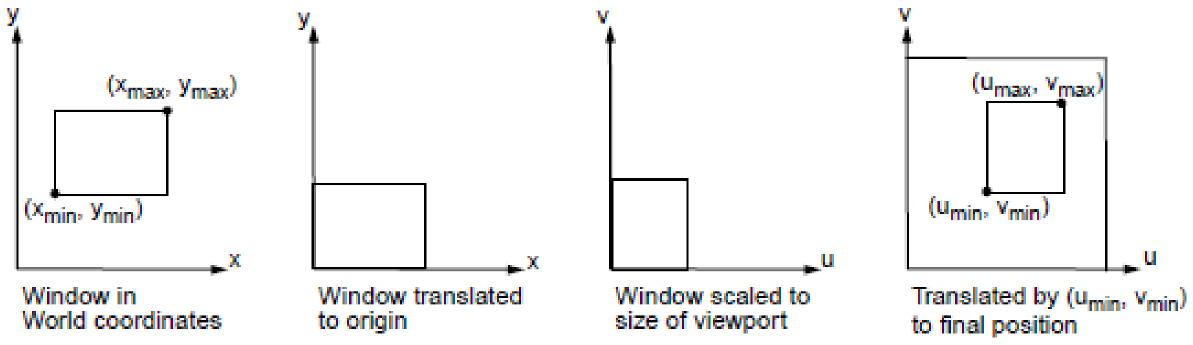

y
(x_{max}, y_{max})
(x_{min}, y_{min})
x
Window in World coordinates
y
x
Window translated to origin
v
u
Window scaled to size of viewport
v
(u_{max}, v_{max})
(u_{min}, v_{min})
u
Translated by (u_{min}, v_{min}) to final position

Figure 3.5: *mapping of a window into a viewport*

$$\begin{aligned}
M_{WV} &= T(u_{min}, v_{min}) \cdot S(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - x \setminus y_{min}}) \cdot T(-x_{min}, -y_{min}) \\
&= \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{u_{max}-u_{min}}{x_{max}-x_{min}} & 0 & 0 \\ 0 & \frac{v_{max}-v_{min}}{y_{max}-x\setminus y_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -x_{min} \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{u_{max}-u_{min}}{x_{max}-x_{min}} & 0 & -x_{min} \cdot \frac{u_{max}-u_{min}}{x_{max}-x_{min}} + u_{min} \\ 0 & \frac{v_{max}-v_{min}}{y_{max}-x\setminus y_{min}} & -y_{min} \cdot \frac{v_{max}-v_{min}}{y_{max}-x\setminus y_{min}} + v_{min} \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{3.13}$$

Each point $(x, y, 1)^T$ becomes on

$$P = \begin{bmatrix} (x - x_{min} \cdot \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min}(y - y_{min}) \cdot \frac{v_{max} - v_{min}}{y_{max} - x \setminus y_{min}} + v_{min} & 1 \end{bmatrix}$$

pictured. There is a need for so-called *clipping methods*, which test objects against the window or viewport limits (see Chapter 5).

Note: In practical applications, vector matrix multiplications can use matrices of the form

$$M = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

to

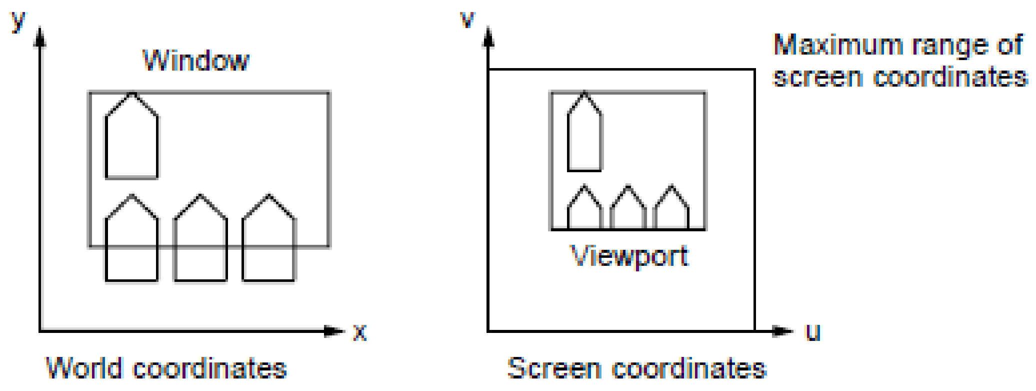


Figure 3.6: *Clipping at window and viewport*

$$x' = x \cdot r_{11} + y \cdot r_{12} + t_x$$
$$y' = y \cdot r_{12} + y \cdot r_{22} + t_y$$

be simplified. Four multiplications and four additions are sufficient.

3.4 3D transformations in homogeneous coordinates

Using homogeneous coordinates, each point (x, y, z) in 3D is described as a 4D vector of the form (x, y, z, W). The homogenization is done by dividing by W:

$$(\frac{x}{W}, \frac{y}{W}, \frac{z}{W}, 1)$$

Each point in three-dimensional space can be described as a straight line through the origin in 4D. $W = 1$ defines an affine 3D subspace (hyperplane) of the 4D space.

3.4.1 Right and left systems

Right-handed coordinate system

The axes run in the positive direction along the right hand. When looking along one axis in the direction of the origin, the other two axes merge into one another through rotation in the mathematically positive sense.

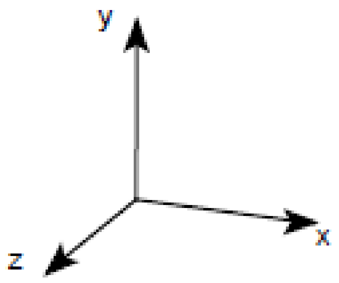


Figure 3.7: *Right-handed coordinate system*

Axis of rotation	positive direction
x	y → z
y	z → x
z	x → y

Left-handed coordinate system

A links system with the screen area in the xy plane is often used. This has the advantage that larger, positive z values correspond to a larger distance from the viewer (camera coordinate system).

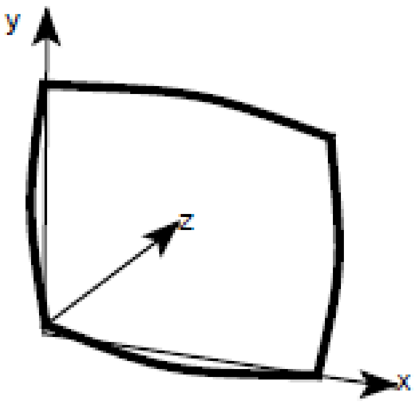


Figure 3.8: *Left-handed coordinate system*

3.4.2 Translation

The translation in 3D can be realized by the following matrix:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

3.4.3 Scaling

The scaling in 3D results in:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.15}$$

3.4.4 Rotation

The 2D rotation can be understood as a 3D rotation along the z-axis

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

To verify, rotate $(1, 0, 0, 1)^T$ by $\frac{\pi}{2}$. The rotation matrices around the x-axis and around the y-axis also result

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.18}$$

A matrix *M* composed of affine single operations has the general form

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.19}$$

The r_{ij} describe rotation and scaling, the t_i describe the translation.

3.4.5 Inversion

Due to the orthogonality of the rotations, a simple inversion is made possible by forming the transpose. This corresponds to a negation of the rotation angle. The inversion of the scaling S results from the reciprocation of s_x, s_y, s_z, that of the translation T results from the negation of t_x, t_y, t_z.

3.4.6 Shear

In 3D one defines three shear matrices: *xy-shear:*

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.20}$$

xz-shear:

$$SH_{xz}(sh_x, sh_z) = \begin{bmatrix} 1 & sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.21}$$

yz shear:

$$SH_{yz}(sh_y, sh_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y & 1 & 0 & 0 \\ sh_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.22}$$

3.4.7 Transformation of normal vectors

Let a plane over the normal n be implicitly defined by $n \cdot P = 0$ in Hessian form.

$$Ax + By + Cz + D = 0 \text{ with } n = (A, B, C, D)^T \text{ and } P = (x, y, z, 1)^T$$

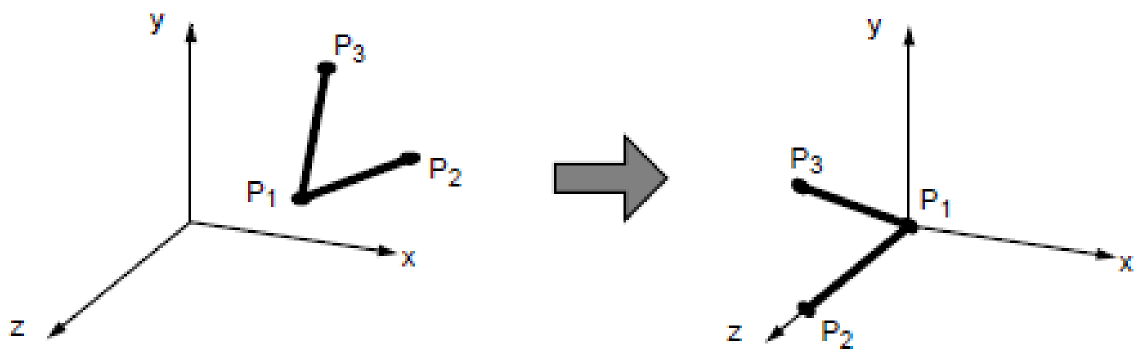
If all points P of the plane are to be transformed with M, the normal n' of the transformed plane results in:

$$n' = (M^{-1})^T n$$

Note: In homogeneous coordinates, all transformations defined as a mapping of $\mathfrak{R}^4 \rightarrow \mathfrak{R}^4$ are linear, but affine in the $\mathfrak{R}^3$ subspace.

3.4.8 Composite 3D Transforms

Example: The two line segments $\overline{P_1P_2}$ and $\overline{P_1P_2}$ are to be transformed as follows:



procedure:

1. Translation by P_1 to the origin
2. y-rotation, so that P_1P_2 in yz-plane
3. x-rotation so that P_1P_2 on z-axis
4. z-rotation, so that P_1P_3 in yz-plane

Step 1: Translation by P_1

$$T(-x_1, -y_1, -z_1) = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

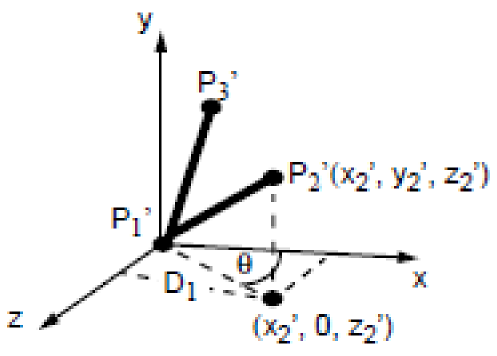
Apply T to all three points

$$P_1' = T(-x_1, -y_1, -z_1) \times P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$P_2' = T(-x_1, -y_1, -z_1) \times P_2 = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 1 \end{bmatrix}$$

$$P_3' = T(-x_1, -y_1, -z_1) \times P_3 = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \\ 1 \end{bmatrix}$$

Step 2: Rotation around the y-axis



The rotation angle results in $-(\pi/2 - \theta)$ and the values for R_y in

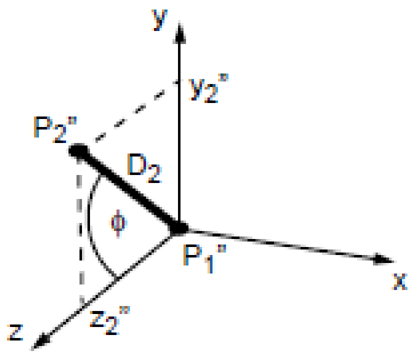
$$cos(\theta - \pi/2) = sin(\theta) = \frac{z_2'}{D_1} = \frac{z_2 - z_1}{D_1}$$
$$sin(\theta - \pi/2) = -cos(\theta) = \frac{x_2'}{D_1} = \frac{x_2 - x_1}{D_1}$$

whereby
$$D_1 = \sqrt{(z_2')^2 + (x_2')^2} = \sqrt{(z_2 - z_1)^2 + (x_2 - x_1)^2}$$

The new points P_2'' and P_3'' result in
$$P_2'' = R_y(\theta - \pi/2) \cdot P_2' = [0 \; y_2 - y_1 \; D_1 \; 1]^T$$
$$P_3'' = R_y(\theta - \pi/2) \cdot P_3'$$

As expected, the x component of P_2'' is 0.

Step 3: Rotation around the x-axis



With

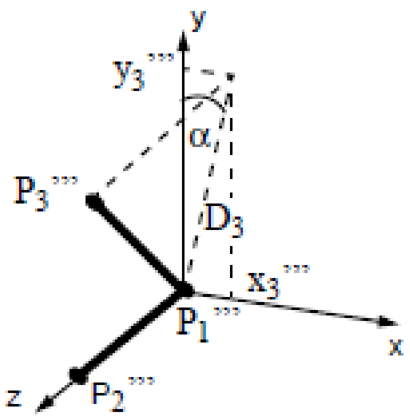
$$D_2 = ||P_1''P_2''|| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

the positive rotation angle ϕ results

$$cos\phi = \frac{z_2''}{D_2} \qquad sin\phi = \frac{y_2''}{D_2}$$

The new point P_2''' becomes

$$\begin{aligned} P_2''' &= R_x(\phi) \cdot P_2'' = R_x(\phi) \cdot R_y(\phi - \pi/2) \cdot P_2' \\ &= R_x(\phi) \cdot R_y(\phi - \pi/2) \cdot T \cdot P_2 \\ &= [0 \; 0 \; ||P_1P_2|| \; 1]^2 \end{aligned}$$



Step 4: rotation around the z-axis

The position of P_3''' results after step 3

$$\begin{aligned} P_3''' &= [x_3'''\ y_3'''\ z_3'''\ 1] \\ &= R_x(\phi) \cdot R_y(\phi - \pi/2) \cdot T(-x_1, -y_1, -z_1) \cdot P_3 \end{aligned}$$

The last positive rotation angle α can be calculated as follows:

$$cos\alpha = \frac{y_3'''}{D_3} \qquad sin\alpha = \frac{x_3'''}{D_3} \qquad D_3 = \sqrt{(x_3''')^2 + (y_3''')^2}$$

The entire transformation matrix is obtained from:

$$R_z(\alpha) \cdot R_x(\phi) \cdot R_y(\phi - \pi/2) \cdot T(-x_1, -y_1, -z_1) = R \cdot T$$

3.4.9 Example application flight simulation

An object, which is available in object coordinates (x_p, y_p, z_p), is to be transformed into a world coordinate system (x, y, z), with the point P and the vector DOF (direction of flight) given are.

Instead of performing the above steps successively, use can be made of the orthogonality of the 3x3 rotation matrices.

Is R through

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix}$$

given, the column vectors rotate
(r_{1x}, r_{1y}, r_{1z}) the x-axis,

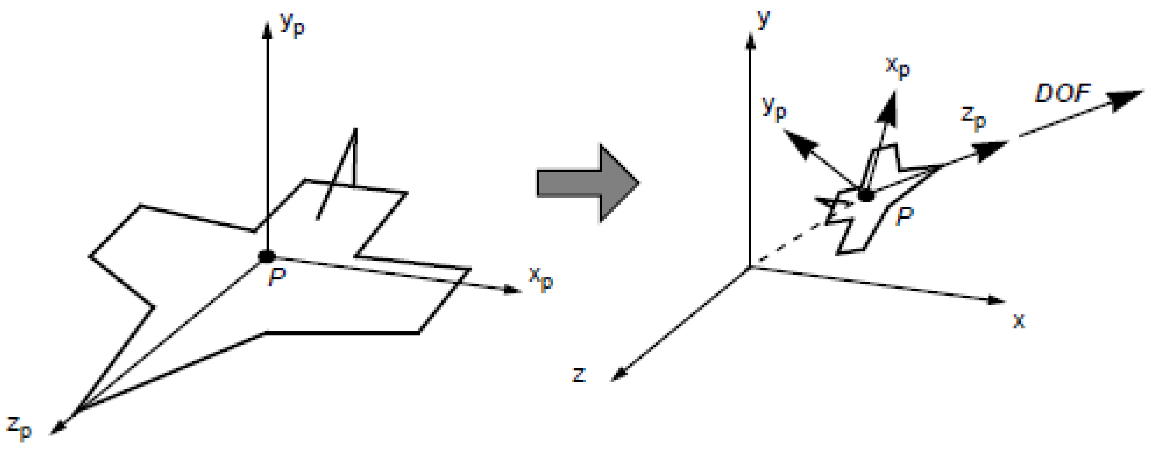


Figure 3.9: *Object and world coordinates for flight simulation*

(r_{2x}, r_{2y}, r_{2z})	the y-axis and
(r_{3x}, r_{3y}, r_{3z})	the z-axis

separated from the original coordinate system; i.e. the vector $(0,0,1)^T$ (the z-axis) is transferred in direction $(r_{3x}, r_{3y}, r_{3z})^T$. Likewise $(1,0,0)^T$ in $(r_{1x}, r_{1y}, r_{1z})^T$ and $(0,1,0)^T$ in $(r_{2x}, r_{2y}, r_{2z})^T$.

If, as in the example, the three orthogonal directions are given as follows and DOF normalized, the result is: $(r_{3x}, r_{3y}, r_{3z})^T = DOF$
$(r_{1x}, r_{1y}, r_{1z})^T = y \times DOF$ (in the left system: - yx DOF)
The vector x_p is perpendicular to y and DOF, which means the system is not tilted.
$(r_{2x}, r_{2y}, r_{2z})^T = DOF \times (y \times DOF)$
Vector y_p is perpendicular to z_p and x_p.
This defines the complete 3x3 rotation matrix.

Example: Rotation around any axis If one wants to describe a rotation around an angle ϕ via any axis $U = (u_x, u_y, u_z)$, the rotation matrix results in homogeneous coordinates

$$R = \begin{bmatrix} u_x^2 + cos\theta(1-u_x^2) & u_xu_y(1-cos\theta) - u_zsin\theta & u_zu_x(1-cos\theta) + u_ysin\theta & 0 \\ u_xu_y(1-cos\theta + u_zsin\theta & u_y^2 + cos\theta(1-u_y^2) & u_yu_z(1-cos\theta) + u_xsin\theta & 0 \\ u_zu_x(1-cos\theta) - u_ysin\theta & u_yu_z(1-cos\theta) + u_xsin\theta & u_z^2 + cos\theta(1-u_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.23}$$

The derivation of the above matrix can be done as an exercise.

It should be noted that transformations can also be understood as a change of coordinate system. The following applies to a right-left change:

$$M_{R\leftarrow L} = M_{L\leftarrow R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.24}$$

3.5 3D rotations and translations with quaternions

3.5.1 Definition and Properties

Another, quite elegant possibility for the definition of rotations and translations results with the help of the so-called quaternions. A major advantage is the compact display compared to a standard 4x4 matrix. A quaternion is a mathematical element defined as follows:

$$q = c + xi + yj + zk \tag{3.25}$$

where c, x, y, z are real numbers and i, j, k are imaginary numbers. In general, the following notation is also used:

$$q = c + u \tag{3.26}$$

c is the real part and $u = xi + yj + zk$ the *pure quaternion*. A quaternion can be understood as a *hypercomplex number* and formally represents an analogous extension of a complex number $c = a + bi$ to four dimensions. A number of operations can be defined on quaternions, resulting in characteristic properties:

- Addition

$$q + q' = (c + c') + (x + x')i + (y + y')j + (z + z')k \tag{3.27}$$

- Multiplication properties of the base $[1, i, j, k]$

$$\begin{aligned} &i^2 = j^2 = k^2 = -1 \\ &ij = k, ji = -k; \qquad jk = i, kj = -i; \qquad ki = j, ik = -j \end{aligned} \tag{3.28}$$

This results in the multiplication formula for two quaternions q and q':

- multiplication

$$qq' = (c + u)(c' + u') \tag{3.29}$$

$$= (cc' - u \cdot u') + (u \times u' + \langle cu' \rangle + \langle c'u \rangle) \tag{3.30}$$

where the operations inner product, $\langle\ \rangle$ and cross product are defined as follows:

$$u \cdot u' = xx' + yy' + zz' \tag{3.31}$$

$$\langle cu \rangle = cxi + cyj + czk \tag{3.32}$$

$$u \times u' = (yz' - zy')i + (zx' - xz')j + (xy' - yx')k \tag{3.33}$$

From this, some essential properties can be derived, which allow the set of quaternions to become a ring $(Q, +, \cdot)$ with the defined mathematical operations.

- one-items
  Addition (3.28) and multiplication (3.30) have the following unity elements:

$$0 = 0 + 0i + 0j + 0k \tag{3.34}$$
$$1 = 1 + 0i + 0j + 0k \tag{3.35}$$
$$\tag{3.36}$$

- Inverse elements
  The inverse element of the addition $-q$ is given by

$$-q = -c - xi - yj - zk \tag{3.37}$$

  The inverse element of the multiplication q^{-1} results from

$$q^{-1} = \frac{1}{||q||^2}\overline{q} \tag{3.38}$$

- Conjugate element $\overline{q}$

$$\overline{q} = c - u \qquad\qquad q = c + u \tag{3.39}$$

- absolute value formation of quaternions

$$q\overline{q} = (c^2 + u \cdot u) + (u \times u - \langle cu \rangle + \langle cu \rangle) \tag{3.40}$$
$$= c^2 + x^2 + y^2 + z^2 \tag{3.41}$$
$$= ||q||^2 \tag{3.42}$$
$$\tag{3.43}$$

`Multiplication is not commutative.`

3.5.2 Quaternions of length one

The quaternions that are important in connection with the transformations are the unit quaternions, i.e. those with absolute value 1. If you form the absolute value of a unit quaternion $||q||^2 = c^2 + u \cdot u = 1$, this relationship can be calculated using a Unit vector $N = [N_x, N_y, N_z]^T$ in $\mathfrak{R}^3$ and $\mathfrak{I} = [i, j, k]^T$, with

$$u = sn\text{with}n = N \cdot I$$
$$q = c + u$$

be rewritten as follows:

$$c^2 + s^2 = 1 \tag{3.44}$$

Thus each unit quaternion can be in the form

$$q = cos(\Theta) + sin(\Theta)n \tag{3.45}$$

being represented. The inner product of two unit quaternions $q4$ and q' results from the above relationships by summing the corresponding angles (trigonometric addition theorem).

$$qq' = cos(\Theta + \Phi) + sin(\Theta + \Phi)n \tag{3.46}$$

3.5.3 3D rotations using unit quaternions

The geometric meaning of the unit quaternions to describe the rotation will be explained below. For this purpose, the rotation of a point P to P' over an arbitrary axis N in 3D, as shown in Fig. 3.10, is first considered Using the following relationships we can express the rotation in 3D

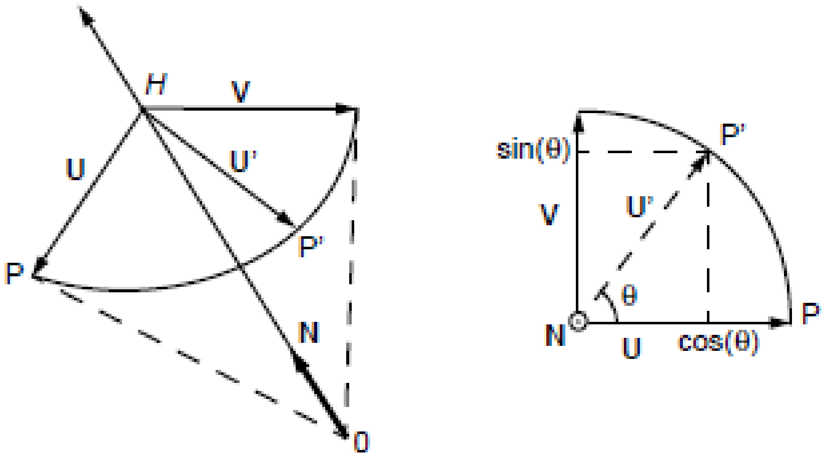


Figure 3.10: *Arrangement of characteristic vectors for 3D rotation*

in a conditional equation for P' depending on P and on N and Θ:

$$P' = cos(\Theta)P + (1 - cos(\Theta))N(N \times P) + sin(\Theta) \times P) \tag{3.47}$$

Let $U \perp V, V \perp N$. The derivation can be done as an exercise. The corresponding rotation matrix $R(\Theta, N)$ can now be derived from the above equation as follows:

$$R(\Theta, N) = cos(\Theta)I_3 + (1 - cos(\Theta))N^T N + sin(\Theta)A_N \tag{3.48}$$

With

$$N = [N_1 N_2 N_3], P = [P_1 P_2 P_3], I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, A_N = \begin{bmatrix} 0 & N_3 & -N_2 \\ -N_3 & 0 & N_1 \\ N_2 & -N_1 & 0 \end{bmatrix}$$

If the analogy between the point $P = [x, y, z]$ in three-dimensional space and the pure quaternion $p = 0 + v = xi + yj + zk$ is considered and further fixed a unit quaternion $q = c + u$ then the following operation can be defined:

$$R_q(p) = qp\overline{q} \tag{3.49}$$

After some rearranging we get

$$R_q(p) = \langle (c^2 - u \cdot u)v \rangle + \langle 2(v \cdot u)u \rangle + \langle 2c(u \times v) \rangle \tag{3.50}$$

Since q is a unit quaternion, we can write $q = cos(\Theta) + sin(\Theta)n$ and get:

$$R_q(p) = \langle cos(2\Theta)v \rangle + \langle (1 - cos(2\Theta))(n \cdot v)n \rangle + \langle sin(2\Theta)(n\ timesv) \rangle \tag{3.51}$$

The established relations are of fundamental importance:

- $R_q$describes the 3D rotation of any point P around the angle 2Θ over the axis N.
- Conversely, a rotation $R(\Theta)$ in 3D is completely determined by the quaternion q. The rotation of the point P simplifies to two multiplications with q and $\overline{q}$ with .

$$p' = qp\overline{q} \tag{3.52}$$
$$q = cos(\Theta/2) + sin(\Theta/2)n \tag{3.53}$$
$$\tag{3.54}$$

3.5.4 Translations and concatenations

The translation as well as concatenations of rotations and translations can also be described using quaternions. This is particularly interesting for animation applications. If p and t are pure quaternions, which describe the point P in space and a translation vector t, then the following relationship applies:

$$p' = p + t \tag{3.55}$$

Furthermore, if r is a unit quaternion that describes the rotation, translation and rotation can be concatenated in the form of the operator $M_{(t,r)}$.

$$p \to p' = M_{(t,r)}(p) = rp\overline{r} + t \tag{3.56}$$

$M_{(0,r)}$ describes the pure rotation and $M_{(t,1)}$ the pure translation. Fig. 3.11 illustrates the connection.
The resulting transformation can be composed by

$$M_{(t,r)} = M_{(0,r)} \cdot M_{(t,1)} \tag{3.57}$$

In a similar way, successive translations and rotations, such as those performed by the camera during an animation, can be linked. Let $M_{(t_0,r_0)}$ be an initial transformation from the origin of the world coordinate system. Each further transformation of the coordinate system $M_{(t_i,r_i)}$ leads to an overall transformation

$$M_{(t,r)} = M_{(t_0,r_0)} \cdot M_{(t_i,r_i)} \tag{3.58}$$

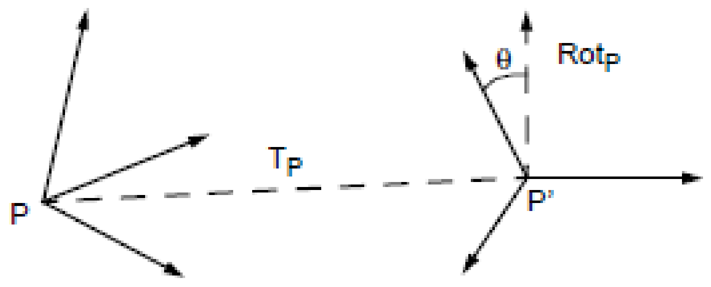


Figure 3.11: *concatenation of translation and rotation*

where $\cdot$ represents a multiplication on the element $M_{(t,r)}$. It will

$$M_{(t,r)} \cdot M_{(t',r')} = M_{(t+rt'\bar{r},r\bar{r})} \tag{3.59}$$

A camera work could look something like Fig. 3.12 and through

$$M_{(t,r)} = M_{(t_0,r_0)} \cdot \ldots \cdot M_{(t_i,r_i)} \cdot \ldots \cdot M_{(t_n,r_n)} \tag{3.60}$$

to be discribed. **The implementation of this type of transformation is**

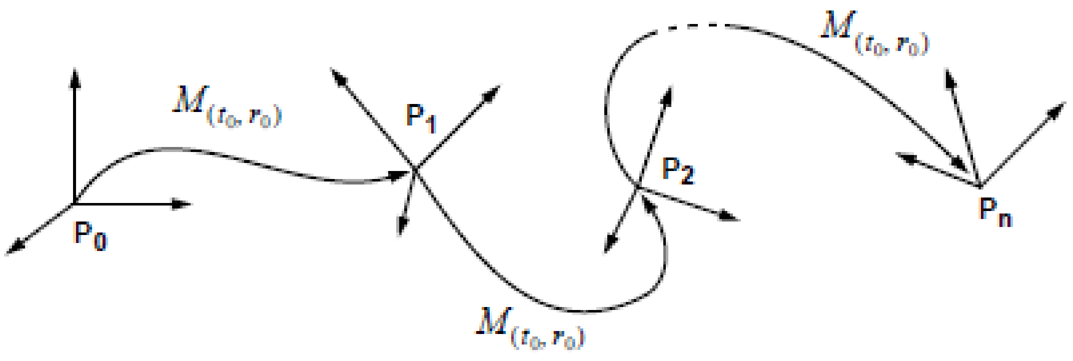


Figure 3.12: *Movement sequence of a camera coordinate system.*

very simple. Only routines for the elementary operations on quaternion defined in Section 3.5.1 have to be provided. The transformation is now fully described by t (3 components), N (3 components) and Θ (1 component) and no longer requires a 4x4 matrix.

Projections for 3D representation

4.1 Fundamentals of planar projections

Basically, planar projections can be divided into *parallel* and *perspective* projections. These two basic types of projection are of fundamental importance in graphic data processing.

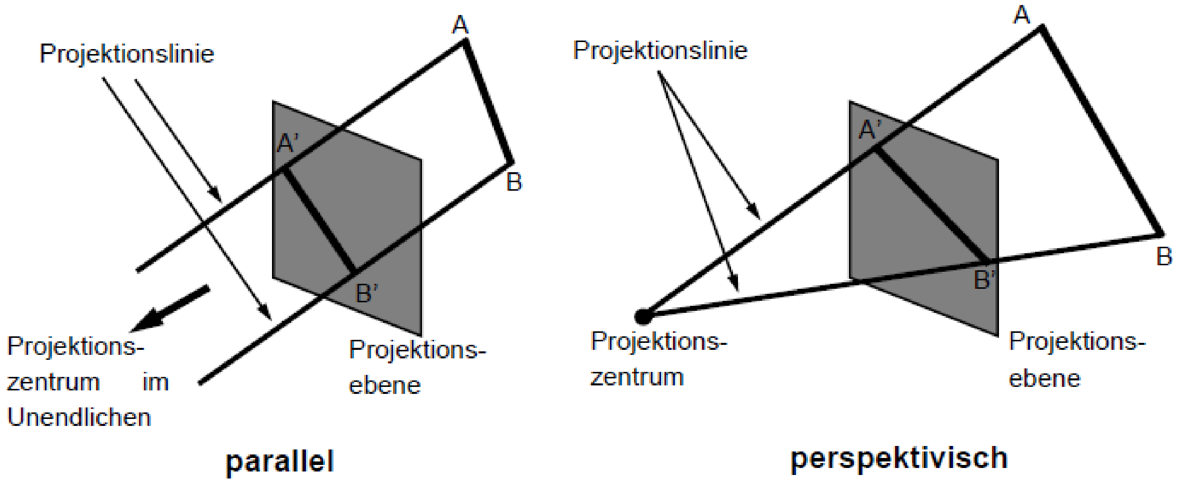


Figure 4.1: *Parallel and perspective projection*

The individual planar projection variants can be classified as follows:

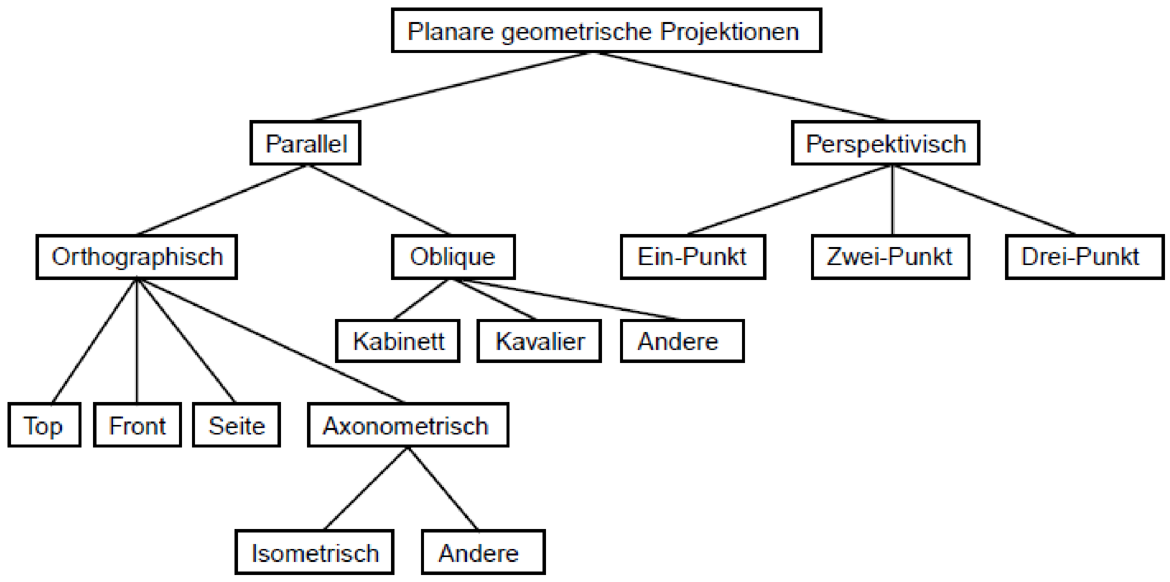


Figure 4.2: *Overview of the variants of the planar projection*

4.1.1 Parallel Projection

Parallel projections differentiate based on the position of the projection direction relative to the normal of the projection plane. When they align, it is termed as orthographic projections.

Orthographic Projections

- *Top, Front, and Side Views:*
  The projection plane is orthogonal to one of the principal axes.
- *Isometric Projection*
  The normal of the projection plane forms equal angles with all principal axes. If the normal is denoted as $\vec{n} = (n_x, n_y, n_z)$, then $n_x = n_y = n_z$ must hold.

Oblique Projections

Oblique projections are employed when the normals do not align with the projection direction. The projection plane is orthogonal to a principal axis. However, distance and angular relationships are generally not preserved.

- *Cavalier View*
  The projection angle significantly influences the appearance of the object. In the cavalier projection, lengths of lines parallel to the projection plane are preserved. The angle β between the projection direction and the projection plane is $45°$. The vertical offset in the projection, denoted by angle α, can also be varied. Adjusting the vertical offset angle will alter the perceived "height" or "depth" of the object in the 2D projection.

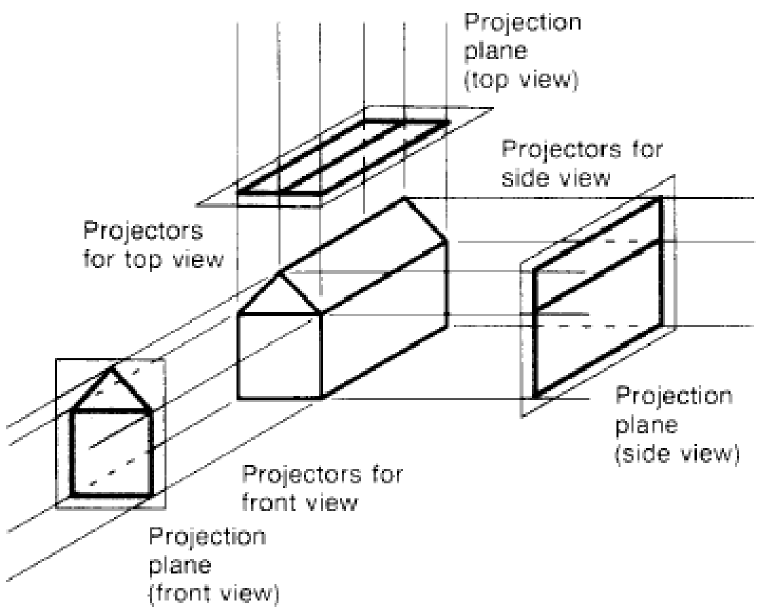

Projection plane (top view)
Projectors for side view
Projectors for top view
Projection plane (side view)
Projectors for front view
Projection plane (front view)

Figure 4.3: *Construction of Three Orthographic Projections*

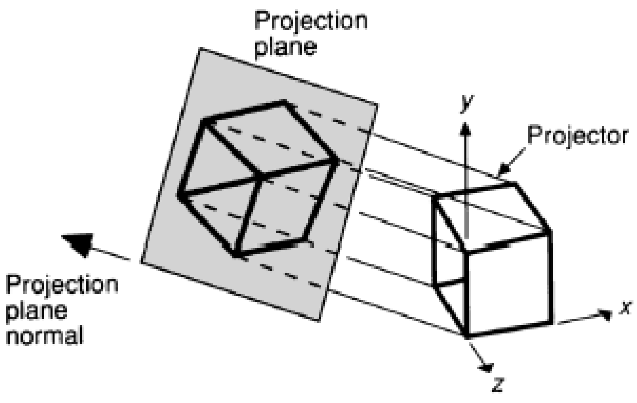


Figure 4.4: *Construction of an Isometric Projection of a Unit Cube*

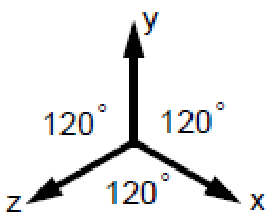


Figure 4.5: *Isometric Projection of the Unit Vectors with the Projection Direction* $(1, 1, 1)$

- *Cabinet View*
  In the cabinet projection, the angle $\beta = \arctan(2) = 63.43°$ is projected in one direction, thereby reducing lengths perpendicular to the projection plane by a factor of 2. The

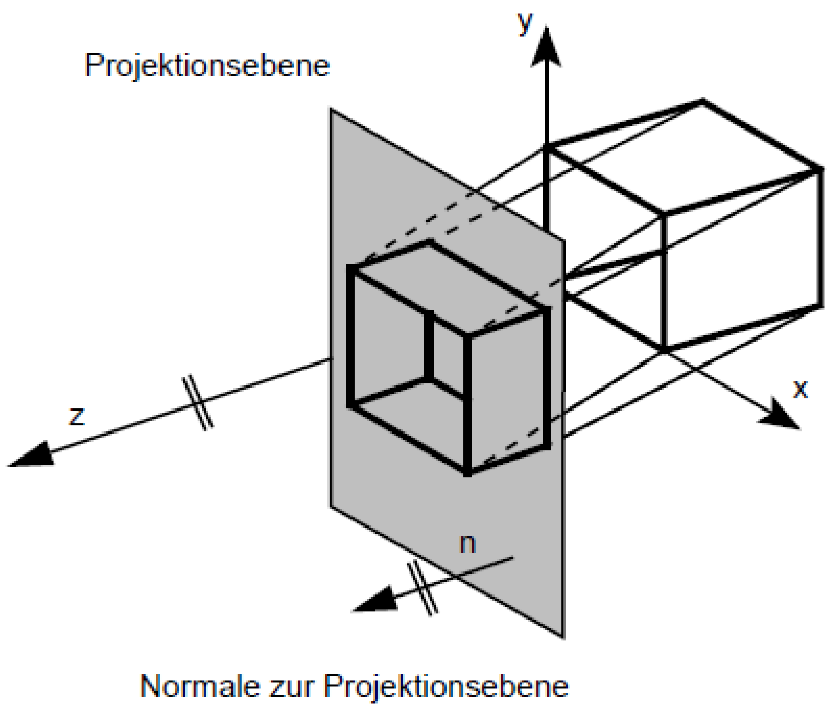


Figure 4.6: *Construction of an Oblique Projection*

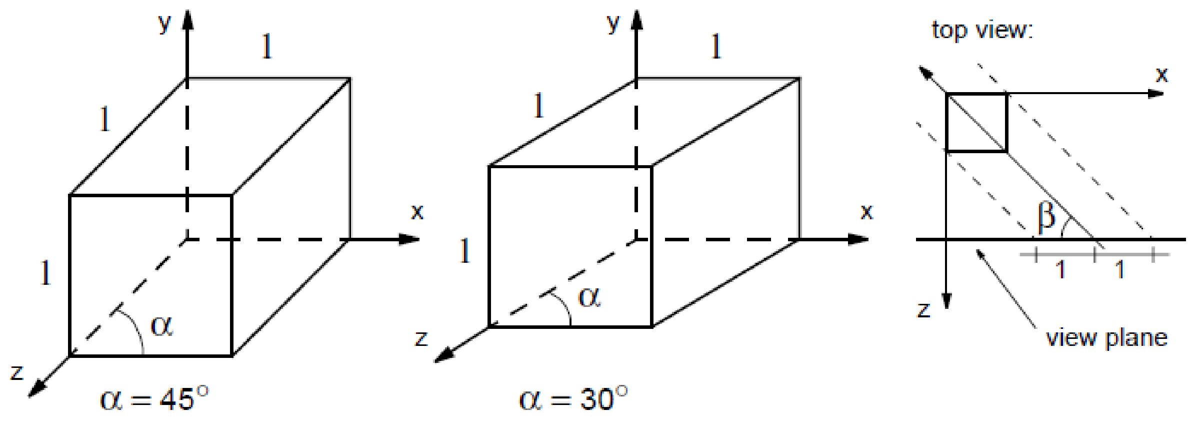


Figure 4.7: *Cavalier Projection of the Unit Cube*

point $\mathbf{P}$ is projected onto the projection plane (x, y) at $\mathbf{P}'$. β is the angle between the projection direction and the plane, while α describes the position relative to the x-axis, i.e., the vertical offset. Note that $\mathbf{P}$ and $\mathbf{P}'$ are perpendicular to each other. The point $\mathbf{P} = (0, 0, 1)$ is mapped to $\mathbf{P}' = (l \cdot \cos\alpha, l \cdot \sin\alpha)$. The direction of projection is $\mathbf{P}' - \mathbf{P} = (l \cdot \cos\alpha, l \cdot \sin\alpha, -1)$. The length of l is calculated as $l = \frac{|\mathbf{P}|}{\tan\beta}$.

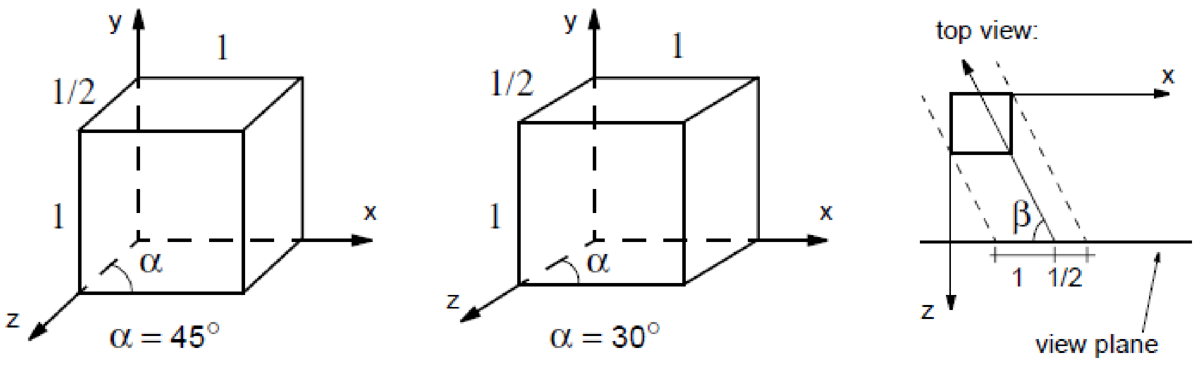


Figure 4.8: *Cabinet Projection of the Unit Cube*

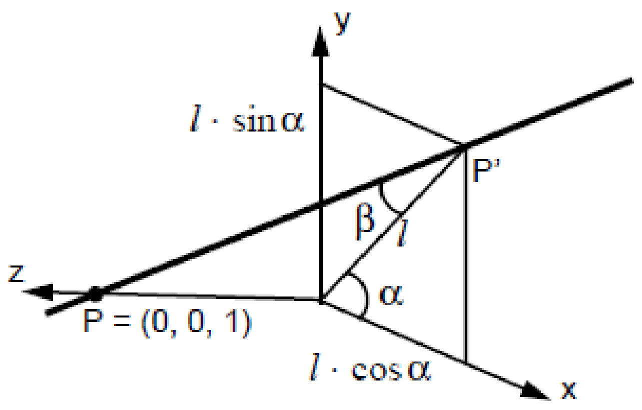


Figure 4.9: *Illustration of the Angles in Cavalier and Cabinet Projection*

4.1.2 Perspective projections

Perspective projections are characterized by one or more vanishing points. If a projection plane is perpendicular to the z-axis, the resulting vanishing point is in that direction as seen from the projection center. However, lines parallel to the x and y axes do not converge.

- *1 point projection*
  The number of vanishing points depends on the number of axes intersected by the projection plane. If, as in Figure 4.10, it is perpendicular to the z axis, a vanishing point results. It is different in Figure 4.11, where both the x and the z axis penetrate the projection plane, resulting in two vanishing points.
- *2 point projection*
  This is shown in Figure 4.11.
- *3 point projection*
  This is shown in Figure 4.12.

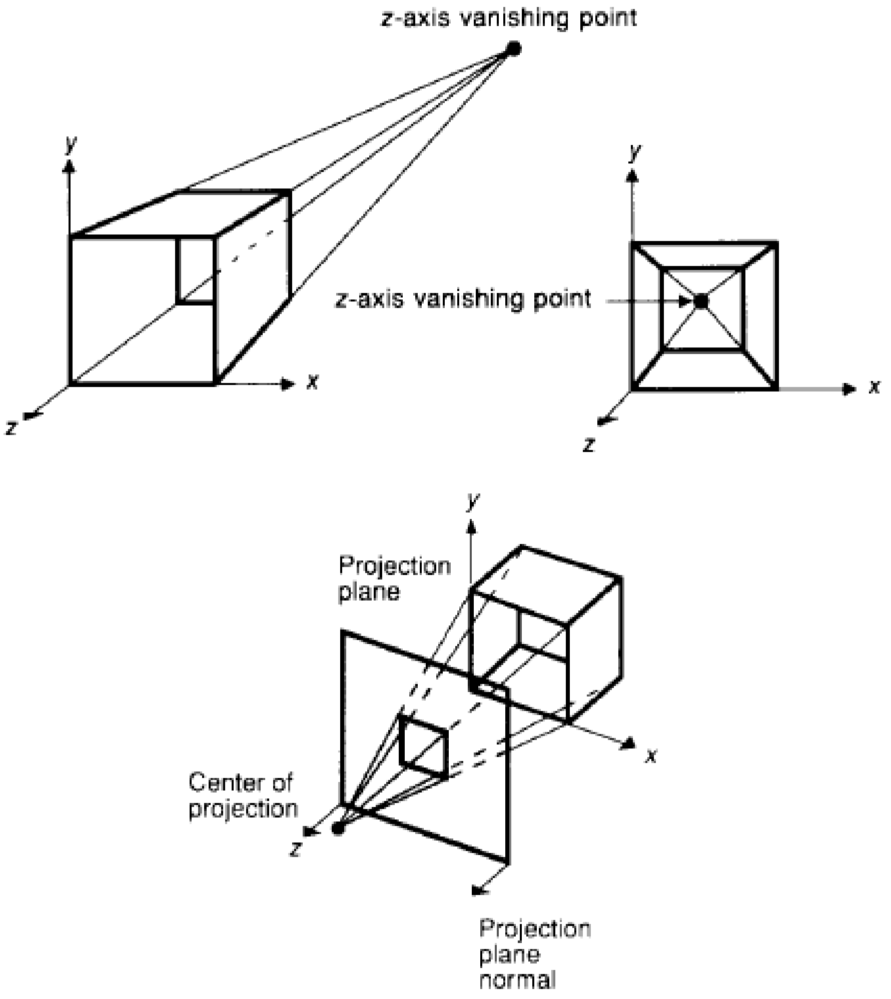


Figure 4.10: *Perspective projection of the unit cube with a vanishing point*

4.1.3 Coordinate systems and viewports

The projection plane and the camera coordinate system can be specified in different ways. Basically, a distinction is made between right and left systems.

Legal system

One way of specifying the camera coordinate system is to use three vectors:

- **VPN**: Normal on the projection plane (View Plane Normal). Also often referred to as $\mathbf{n}$ or $\mathbf{look} - \mathbf{at}$.
- **VRP**: Reference point in the projection plane (View Reference Point). It is used to define the image plane, but can also be defined by a second vector in the image plane, as in the links system described later.
- **VUP**: Defines the v direction in the projection plane (View Up Vector).

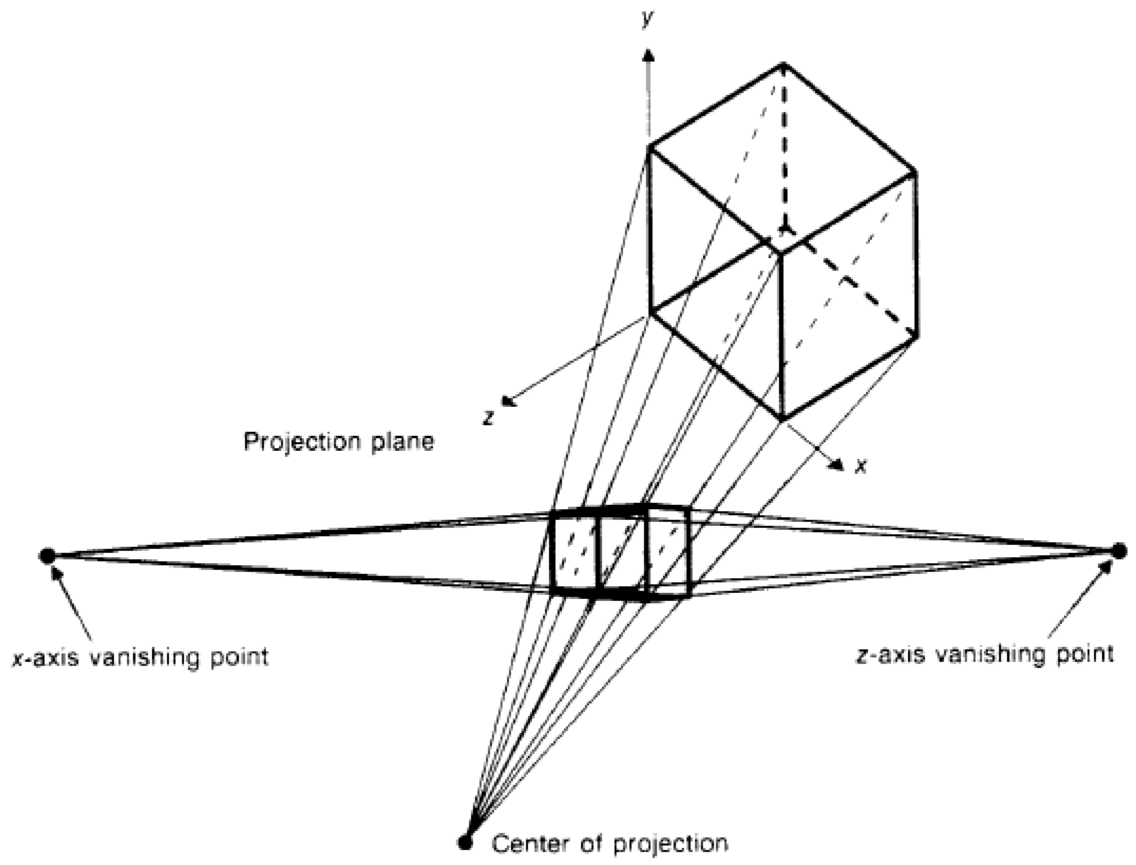


Figure 4.11: *Perspective 2-point projection of the unit cube (in this drawing, the projection plane corresponds to the drawing plane.*

The projection window can be defined with the window center (center of window: **CW**) on the projection plane. The limitations of the projection windows result in *viewing volumes*, which can have a different shape depending on the type of projection.

Link system

The advantage of a links system is the positive z-axis to describe the distance to the viewer. The following terms are commonly used in ray tracing programs. Here, too, the projection is clearly described by **K, R(look-at), Up** and **Right**.

4.1.4 Clipping Planes

As described in Chapter 5, object clipping is an important process within the graphics pipeline. In addition to the natural clipping planes along the viewing volume, *Front Planes* and *Back Planes* are often introduced.

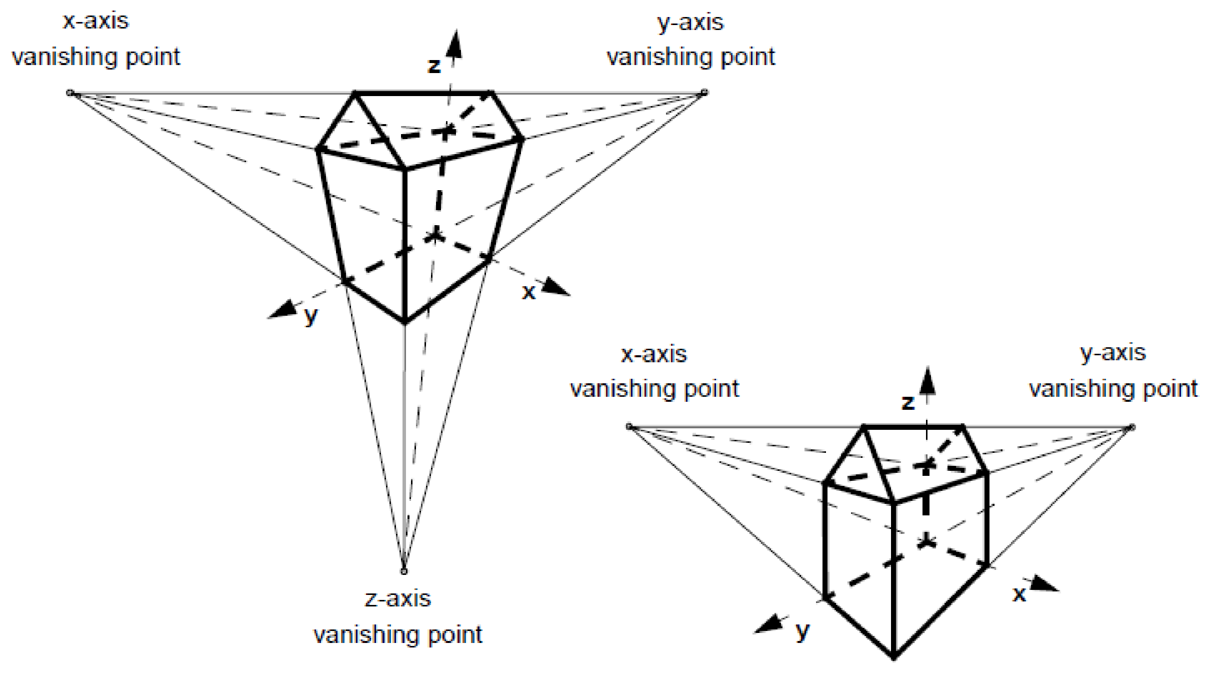


Figure 4.12: *Example of a 3 point projection (projection plane intersects all 3 coordinate axes) compared to the 2 point projection of the same object (only the x and y axes are intersected*

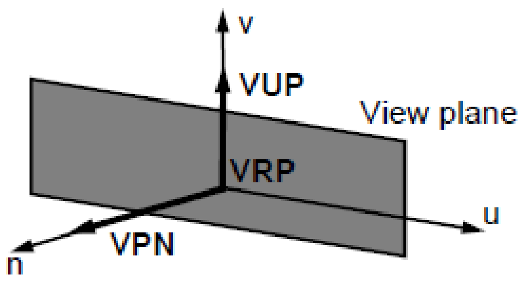


Figure 4.13: *Plane of projection in the legal system*

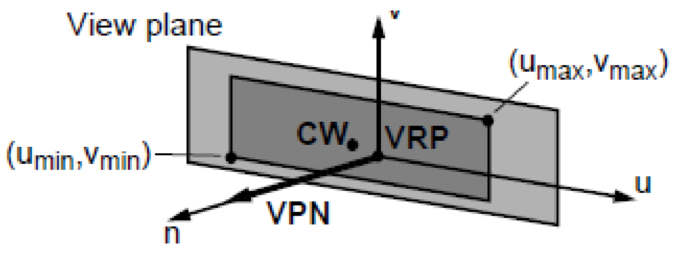


Figure 4.14: *Right-hand reference coordinate system given by the u,v, and n-axis (camera or viewing coordinate system)*

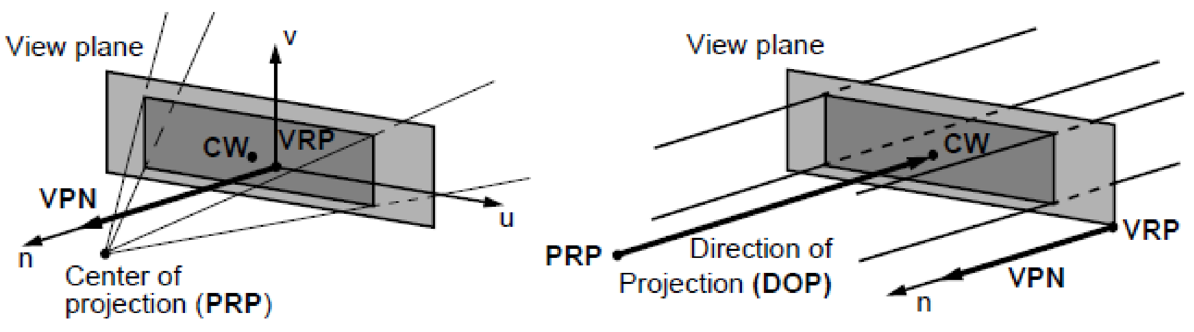


Figure 4.15: *Viewing volumes for parallel and perspective projection*

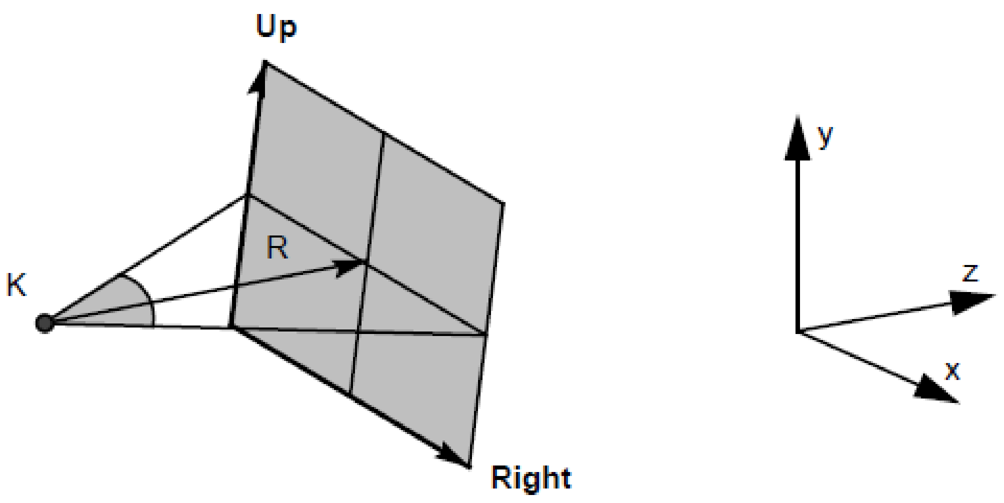


Figure 4.16: *Plane of projection in the left system*

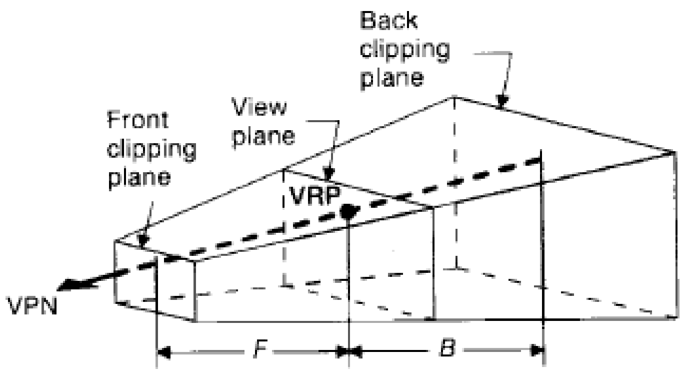


Figure 4.17: *Clipping planes with perspective projection*

4.2 Mathematics of projection types

For the perspective projection, it should apply in the following that the projection plane is perpendicular to the z axis at a distance d from the origin. In the case of parallel projection, the

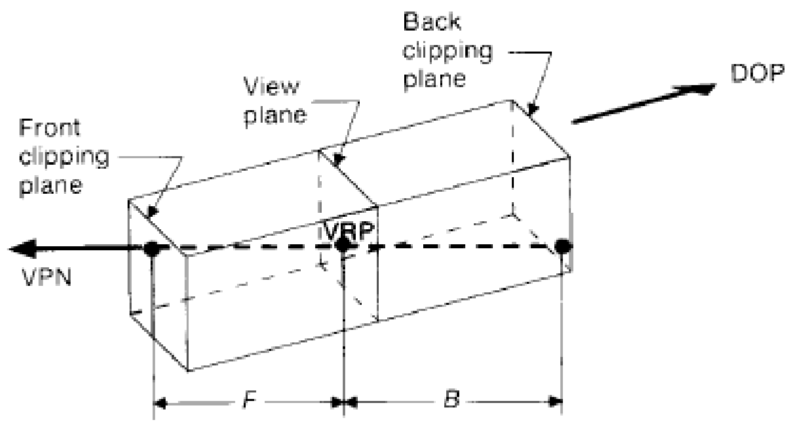


Figure 4.18: *Clipping planes in oblique parallel projection*

$z = 0$ plane should be imaged (camera coordinate system).

4.2.1 Perspective projection

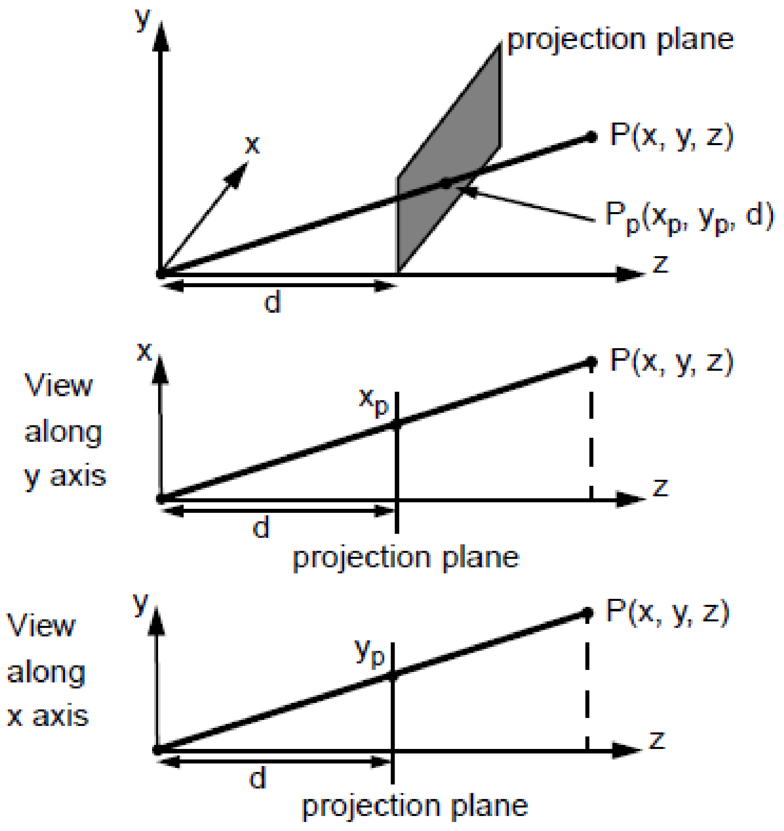


Figure 4.19: *Perspective projection*

In the perspective projection of Figure 4.19 the following applies:

$$\frac{x_p}{d} = \frac{x}{z} \qquad \frac{y_p}{d} = \frac{y}{z}$$

solved for x_p or y_p results

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d} \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}$$

for $z \neq 0$.

Using the homogeneous coordinates, the projection can be expressed by the following 4x4 matrix:

$$\mathbf{M_{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \tag{4.1}$$

In general, one obtains the point $\mathbf{P_p}$ in homogeneous coordinates $[X, Y, Z, W]^T$ by

$$\mathbf{P_p} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \mathbf{M_{per}} \cdot \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

or

$$P_p = [X\ Y\ Z\ W]^T = \left[x\ y\ z\ \frac{z}{d}\right]$$

The 3D coordinates transformed into the $(z = d)$ plane result from homogenization

$$\left[\frac{X}{W}, \frac{Y}{W} \frac{Z}{W}\right] = [x_p, y_p, z_p] = \left[\frac{x}{z/d}, \frac{y}{z/d}, d\right] \tag{4.2}$$

An alternative projection into the $(z = 0)$-plane with center $z = -d$ results

$$\frac{x_p}{d} = \frac{x}{z+d} \qquad \frac{y_p}{d} = \frac{y}{z+d}$$

or

$$x_p = \frac{d \cdot x}{z+d} = \frac{x}{(z/d)+1} \qquad y_p = \frac{d \cdot y}{z+d} = \frac{y}{(z/d)+1}$$

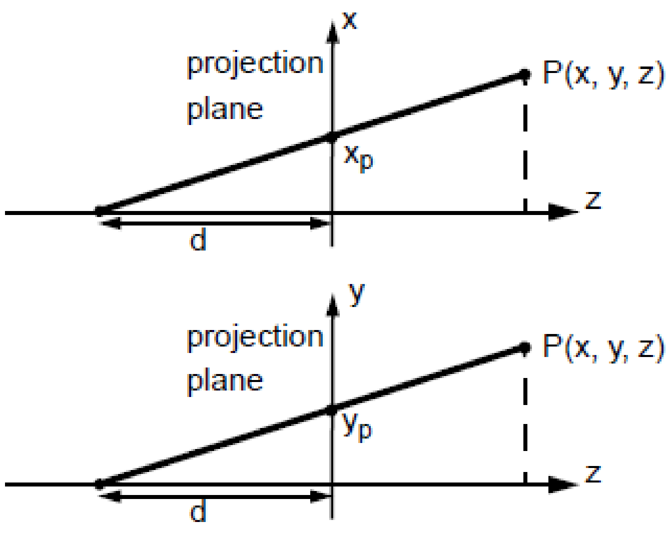


Figure 4.20: *Alternative perspective projection into the (z=0)-plane*

As a 4x4 matrix one obtains

$$\mathbf{M}'_{\mathbf{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \tag{4.3}$$

4.2.2 parallel projection

An advantage of the projection into the $(z = 0)$ plane is that for $d \rightarrow -\infty$ the perspective projection is transformed into the parallel projection. The above variables then assume the following values.

$$x_p = x \qquad y_p = y \qquad z_p = 0$$

and

$$\mathbf{M}_{\mathbf{place}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

4.2.3 General formulation

A uniform description of parallel and perspective projection is desirable, with any projection center **COP**, as shown in Figure 4.21. This also allows projections with multiple vanishing points to be calculated.

Given:

- **COP**
- Projection plane $(0, 0, z_p)$
- Point **P**(x,y,z)
- normalized direction vector (d_x, d_y, d_z) with distance Q

Searched:

- Projection coordinates (x_p, y_p, z_p)

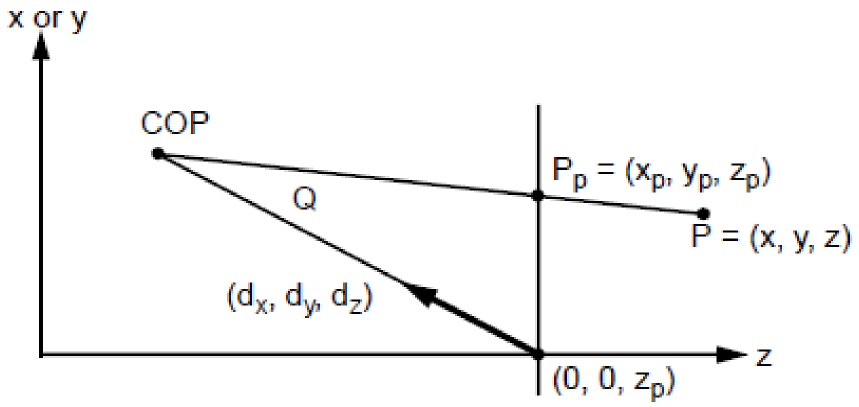


Figure 4.21: *Illustration for the derivation of a general formulation for the projection*

The straight line from **P** to **COP** results in parametric representation:

$$\mathbf{COP} + t(\mathbf{P} - \mathbf{COP}), \qquad 0 \leq t \leq 1$$

or for each point $\mathbf{P}'(x', y', z')to$

$$x' = Q \times d_x + (xQ \times d_x) \times ty' \quad = Q \times d_y + (yQ \times d_y) \times tz' = (z_p + Q \cdot d_z) + (z - (z_p + Q \cdot d_z)) \cdot t$$

The projection $\mathbf{P_p}$ of **P** results from the intersection of the straight line and the projection plane. Out of

$$z_p = (z_p + Q \cdot d_z) + (z - (z_p + Q \cdot d_z)) \cdot t$$

you get

$$t = \frac{z_p - (z_p + Q \cdot d_z)}{z + (z_p + Q \cdot d_z}$$

With that comes about

$$x_p = \frac{xz \cdot \frac{d_x}{d_z} + z_p \cdot \frac{d_x}{d_z}}{\frac{z_p - z}{Q \cdot d_z} + 1} y_p \qquad = \frac{yz \cdot \frac{d_y}{d_z} + z_p \cdot \frac{d_y}{d_z}}{\frac{z_p - z}{Q \cdot d_z} + 1}$$

By extension, z_p can be used as

$$z_p = z_p \cdot \frac{\frac{z_p - z}{Q \cdot d_z} + 1}{\frac{z_p - z}{Q \cdot d_z} + 1} = \frac{-z \cdot \frac{z_p}{Q \cdot d_z} + \frac{z_p^2 + z_p \cdot Q \cdot d_z}{Q \cdot d_z}}{\frac{z_p - z}{Q \cdot d_z} + 1}$$

express and one gets a general 4x4 matrix:

$$\mathbf{M_{general}} = \begin{bmatrix} 1 & 0 & -\frac{d_x}{d_z} & z_p \frac{d_x}{d_z} \\ 0 & 1 & -\frac{d_y}{d_z} & z_p - \frac{d_y}{d_z} \\ 0 & 0 & (-\frac{z_p}{Qd_z}) & \frac{z_p^2}{Qd_z} + z_p \\ 0 & 0 & -\frac{1}{Qd_z} & \frac{z_p}{Qd_z} + 1 \end{bmatrix} \tag{4.5}$$

The previous formulations then result from $\mathbf{M_{general}}$ as follows:

	z_p	Q	$[d_x\ d_y\ d_z]$
$\mathbf{M_{ort}}$	0	∞	$[0\ 01]$
$\mathbf{M_{per}}$	d	d	$[0\ 0\ -1]$
$\mathbf{M'_{per}}$	0	d	$[0\ 0\ -1]$

The cavalier and cabinet projections result in:

	z_p	Q	$[d_x\ d_y\ d_z]$
Cavalier	0	∞	$[cos\alpha\ sin\alpha 1]$
Cabinet	0	∞	$[\frac{cos\alpha}{2}\ \frac{sin\alpha}{2}\ -1]$

4.2.4 summary

In practical applications, the perspective mapping of a scene in world coordinates is achieved through the following steps:

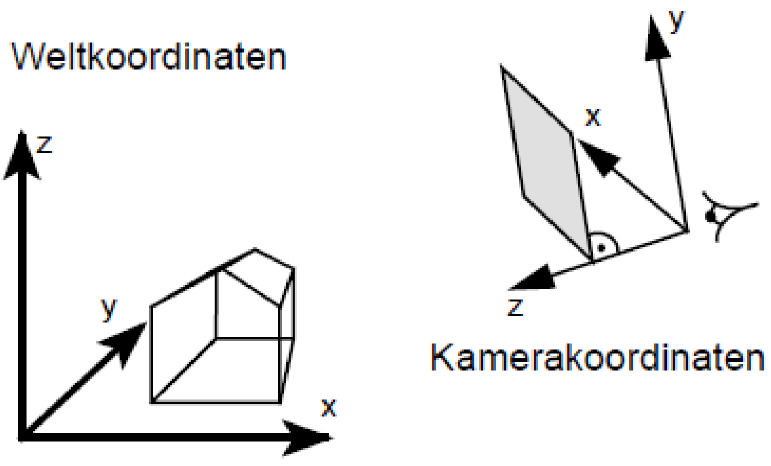


Figure 4.22: *Transition from world coordinates to camera coordinates*

1. The viewing transformation first leads to the camera coordinate system by means of an affine mapping (translation and rotation):

$$\begin{bmatrix} x_k \\ y_k \\ z_k \\ w_k \end{bmatrix} = \mathbf{M_k} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

2. In order for the z-axis to point in the positive direction, a left-hand system is used:

$$\begin{bmatrix} x'_k \\ y'_k \\ z'_k \\ w'_k \end{bmatrix} = \mathbf{M_{RL}} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

3. In the simplest case, the 2D projection is achieved by z-division

$$x_p = x'_k \cdot \frac{d}{z'_k} = u$$
$$y_p = y'_k \cdot \frac{d}{z'_k} = v$$
$$z_p = d$$

d creates a zoom effect. By additionally defining the window $(u_{min}, u_{max}, v_{min}, v_{max})$, a specified opening angle (e.g. $46°$ with a 50mm lens) can be achieved.

Remark: The model described here is just a pinhole model. Non-linearities and edge distortions are not taken into account.

Given:

- **UP** vector
- **Loot-At** vector

The mapping into the camera coordinate system was done by translation at its origin and by rotation, so that
x axis = $\mathbf{h_x}$ vector
y axis = $\mathbf{h_y}$ vector
z axis = $\mathbf{h_z}$ vector

The following applies:

$$\boldsymbol{h_x} = -\frac{\boldsymbol{UP} \times \boldsymbol{Look} - \boldsymbol{At}}{||\boldsymbol{UP} \times \boldsymbol{Look} - \boldsymbol{At}||}$$
$$\boldsymbol{h_y} = \frac{\boldsymbol{UP}}{||\boldsymbol{UP}||}$$
$$\boldsymbol{h_z} = -\frac{\boldsymbol{LookAt}}{||\boldsymbol{LookAt}||}$$

The orthogonality of the rotation matrix results in the resulting transformation as a concatenation of the individual operations.

$$\mathbf{M_{LR}} \cdot \mathbf{M_T} \cdot \mathbf{M_R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & -1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_{x_1} & h_{x_2} & h_{x_3} & 0 \\ h_{y_1} & h_{y_2} & h_{y_3} & 0 \\ h_{z_1} & h_{z_2} & h_{z_3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

$$= \begin{bmatrix} h_{x_1} & h_{x_2} & h_{x_3} & -P_x \\ h_{y_1} & h_{y_2} & h_{y_3} & -P_y \\ -h_{z_1} & -h_{z_2} & -h_{z_3} & -Pz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

Clipping

In contrast to the ray tracing method (see Chapter 10), which is still to be discussed, clipping, hidden surface calculations and scan conversion must be carried out for direct image generation by projecting primitives into the image plane.
The algorithms required for this are presented in the following chapters. You deal with elementary algorithms to solve these problems. This chapter covers 2D and 3D clipping, while Chapter 6 deals with scan conversion and Chapter 7 with hidden line and hidden surface problems.

5.1 Introduction

One of the basic tasks when processing geometry is the limitation of lines and polygons to a given rectangle (viewport, window) in 2D or to a viewing volume in 3D. This processing step is called clipping. It is used both to accelerate the display of graphic primitives and to avoid overflow-related artefacts (depending on the hardware). Basic algorithms are particularly important for lines and polygons, since more complex primitives are represented in most cases using linear or planar approximations. 2D clipping of lines to rectangles always results in line segments, but clipping of *concave* polygons can result in multiple polygons.

5.2 Line clipping in 2D

For line clipping on a rectangle, an inside-outside preselection of the line end points is first carried out. The following three cases must be distinguished.

I. If both endpoints are inside the rectangle, no clipping is necessary.

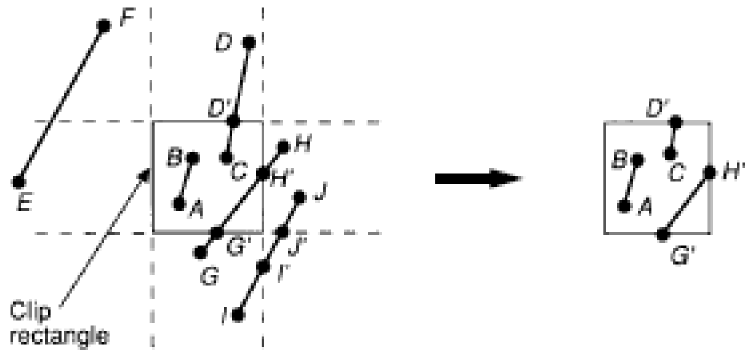


Figure 5.1: *Line clipping in 2D*

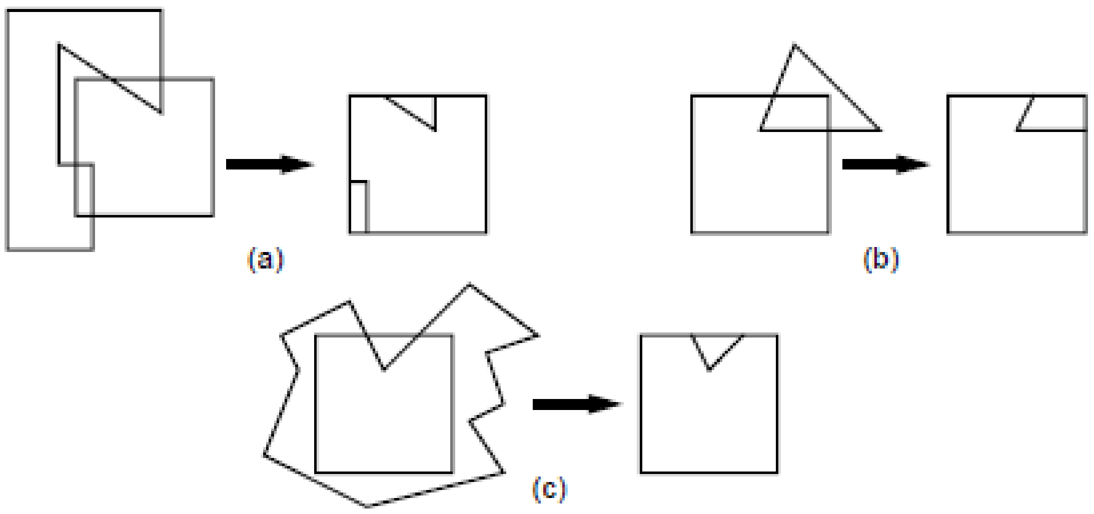


Figure 5.2: *Polygon clipping in 2D*

II. If one endpoint is inside and one is outside the rectangle, clipping is necessary. This requires intersection calculations.

III. If both points are outside, clipping may be necessary if the clipping rectangle is intersected by the lines. Further tests must be carried out.

A point (x, y) is inside the clipping rectangle $(x_{min}, x_{max}, y_{min}, y_{max})$ if

$$x_{min} \leq x \leq x_{max} \text{ and } y_{min} \leq y \leq y_{max}$$

5.2.1 brute force method

For all cases under II and III, calculate the intersections with the four boundary lines of the rectangle and use the intersection points to decide whether line clipping is necessary. For example, in Figure 5.1, G' and H' intersect inside, while I' and J' intersect outside. For this we set up the parametric form of a straight line, which can be expressed as:

$$x = x_0 + t(x_1 - x_0) \tag{5.1}$$
$$y = y_0 + t(y_1 - y_0) \tag{5.2}$$

These equations describe (x, y) on the line segment from (x_0, y_0) to (x_1, y_1) for the parameter t in the range $[0, 1]$. In order to intersect two straight lines, the equations for the rectangle edge with the parameter t_{edge} as well as for the line itself with the parameter t_{line} have to be set up and the resulting system of equations has to be solved. If both parameters are in the range $[0, 1]$, clipping is necessary. Furthermore, the special case of a line parallel to an edge of a rectangle must be dealt with before the above equations can be solved. Because of the computational effort involved in this approach, it is inefficient and not used in practice.

5.2.2 Cohn-Sutherland Algorithm

This is a traditional, but quite efficient algorithm, which is particularly suitable for hardware implementations.

To avoid computationally expensive intersection calculations, an attempt is made to accept or eliminate as many lines as possible through simple comparisons (as in step I, for example). If this does not succeed, the line is divided into two segments, one of which can be treated once. So you get a two-step process:

1. **step:** *sectioning of the plane*

   The basic idea for a quick preselection of lines is to check whether both end points are in certain sections with respect to the clipping rectangle. The following 4-bit code is used for this:

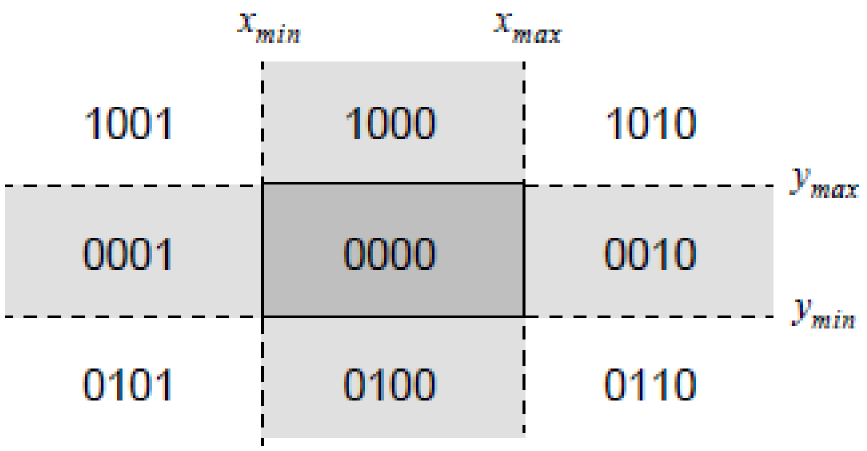


Figure 5.3: *4-bit codes of the different regions*

   The individual bits have the following meaning:

1. Bit: In the half-plane above the upper edge $y > y_{max}$
2. Bit: In the half-plane below the lower edge $y < y_{max}$
3. Bit: In the half-plane to the right of the right edge $x > x_{max}$
4. Bit: In the half-plane to the left of the left edge $x > x_{max}$

The bits resulted as signs from the following subtractions:

1. Bit: $y_{max} - y$
2. Bit: $y - y_{min}$
3. Bit: $x_{max} - x$
4. Bit: $x - x_{min}$

Such a code is calculated for each line end point. Then the codes of the two line end points are binary **AND**-linked. If $code_1 \wedge code_2 \neq 0$ the line can be removed because there is no intersection with the clipping rectangle. For $code1 \vee code2 = 0$ the line is entirely within the clipping rectangle.

For example, the following codes result for the lines in Fig. 5.1:

line	$\mathbf{code_1}$	$\mathbf{code_2}$	$\mathbf{code_1 \wedge code_2}$
$\mathbf{A \rightarrow B}$	0000	0000	0000
$\mathbf{C \rightarrow D}$	0000	1000	0000
$\mathbf{E \rightarrow F}$	0001	1001	0001
$\mathbf{G \rightarrow H}$	0100	0010	0000
$\mathbf{I \rightarrow J}$	0100	0010	0000

2. **Step:** *Iterative Subdivision*

Using the codes of the line endpoints and the tests performed in step 1, it can be determined which edges of the clipping rectangle will be intersected.

Point D in Fig. 5.4 has the code 1001, ie the top and left edges must be cut. A line that survived the tests in step 1 is now divided at one of its intersections with the rectangle edges. For this purpose, a fixed order is specified for the interpretation of the bits in the code (e.g. top $\rightarrow$ bottom $\rightarrow$ right $\rightarrow$ left). Now the following steps are carried out iteratively until the line is trivially accepted:

- Selection of one of the two points of the line. (The point must be in the outer half-plane of a rectangle edge)
- Split the line into two segments at its intersection with the highest priority rectangle edge
- Eliminate segment point $\rightarrow$ intersection

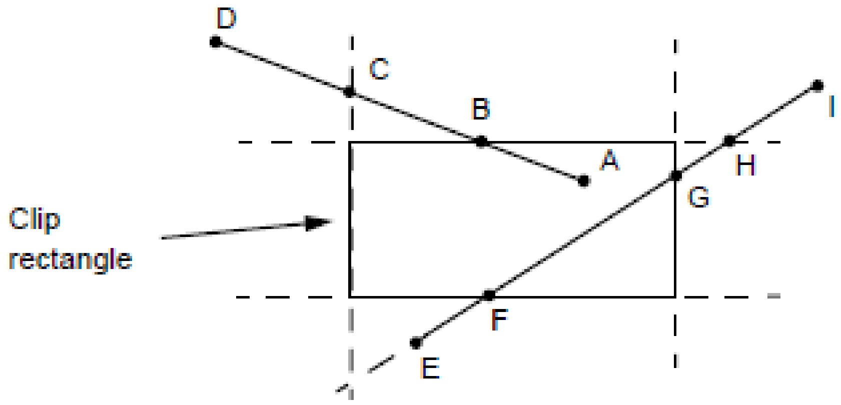


Figure 5.4: *Illustration of the Cohen-Sutherland clipping*

- Calculation of the code for the intersection
- Iteration until the remainder segment is trivially accepted

In Fig. 5.4, point D is first selected by the algorithm as the starting point and then the intersection point B is calculated because of the priority convention (up $\rightarrow$ down $\rightarrow$ right $\rightarrow$ left).$D \rightarrow B$ is eliminated and $B \rightarrow A$ is accepted. The line $E \rightarrow I$, on the other hand, has to be treated several times. The algorithm first chooses E(0100) and computes the intersection H(0010) and the remainder segment $E \rightarrow H$. For this, E is selected as the starting point and F is first calculated based on the priorities. Finally, the clipped lines are calculated from $F \rightarrow H$ by dividing them into G.

The algorithm is particularly suitable for very small and very large rectangles (eg picking). Another advantage is the easy expansion to 3D. A disadvantage are sometimes unnecessary intersection calculations like in the example of the intersection H).

The Cohen-Sutherland algorithm in C code:

```
#define NIL 0
#define LEFT 1
#define RIGHT 2
#define BOTTOM 4
#define TOP 8

/* clipping rectangle coordinates are global */
float xmin, xmax;
float ymin, ymax;
/* returns outcode for a point x,y */
short CompOutCode(float x, float y) {
        short outcode;
        outcode = NIL;
        if (y > ymax)
                outcode |= TOP;
        else if (y < ymin)
```

```
                outcode |= BOTTOM;
        if (x > xmax)
                outcode |= RIGHT;
        else if (x < xmin)
                outcode |= LEFT;
        return outcode;
}

/***********************************************
Cohen−Sutherland clipping algorithm
for the line from P0 to P1 against a
clipping rectangle.

−−> x0, y0: P0
−−> x1, y1: P1
(−−> xmin, xmax, ymin, ymax: clipping rectangle)
***********************************************/
void CohenSutherlandLineClipAndDraw(float x0, float y0, float x1, float y1) {
        int accept = FALSE;
        int done = FALSE;
        float x, y;
        float slope = (y1 − y0) / (x1 − x0);
        short outcode0, outcode1;
        short outcodeOut;

        outcode0 = CompOutCode(x0, y0);
        outcode1 = CompOutCode(x1, y1);
        do {
                if (!(outcode0 | outcode1)) {/* trivial inside */
                        accept = TRUE;
                        done = TRUE;
                }
                else if ((outcode0 & outcode1))/* trivial outside */
                        done = TRUE;
                else {
                /* If the previous two tests failed, the
                intersection are calculated. At least one endpoint
                the line to be clipped is outside −−> outcodeOut */
                        if (outcode0)
                                outcodeOut = outcode0;
                        else
                                outcodeOut = outcode1;

                        /* now find the intersection using the relations
                                y = y0 + slope*(x − x0),
                                x = x0 + 1/slope*(y − y0) */
```

```
                    if (outcodeOut & TOP) {
                            x = x0 + (x1 − x0)∗(ymax − y0)/(y1 − y0);
                            y = ymax;
                    }
                    else if (outcodeOut & BOTTOM) {
                            x = x0 + (x1 − x0)∗(ymin − y0)/(y1 − y0);
                            y = ymin;
                    }
                    else if (outcodeOut & RIGHT) {
                            y = y0 + (y1 − y0)∗(xmax − x0)/(x1 − x0);
                            x = xmax;
                    }
                    else if (outcodeOut & LEFT) {
                            y = y0 + (y1 − y0)∗(xmin − x0)/(x1 − x0);
                            x = xmin;
                    }

                    /∗ Move the outside endpoint
                    to the intersection and preparation of the next
                    loop iteration ∗/
                    if (outcodeOut == outcode0) {
                            x0 = x;
                            y0 = y;
                            outcode0 = CompOutCode(x0, y0);
                    }
                    else {
                            x1 = x;
                            y1 = y;
                            outcode1 = CompOutCode(x1, y1);
                    }
            }
        } while (!done);

        /∗ draw line ∗/
        if (accept)
                MidpointLineReal(x0, y0, x1, y1);
}
```

5.2.3 Parametric line clipping (Liang-Barsky / Cyrus-Beck)

The essential feature of this algorithm is the calculation of all points of intersection in the only one-dimensional parameter space of the straight line. The algorithm works as follows:

1. **step:** *calculation of the cut in the parameter space*

With the Cohen-Sutherland algorithm, the (x, y) coordinates or in 3D the (x, y, z) coordinates are calculated for each intersection point. The description of the intersection points in the 1D parameter space of the straight line is much more efficient. The following situation is assumed for this:

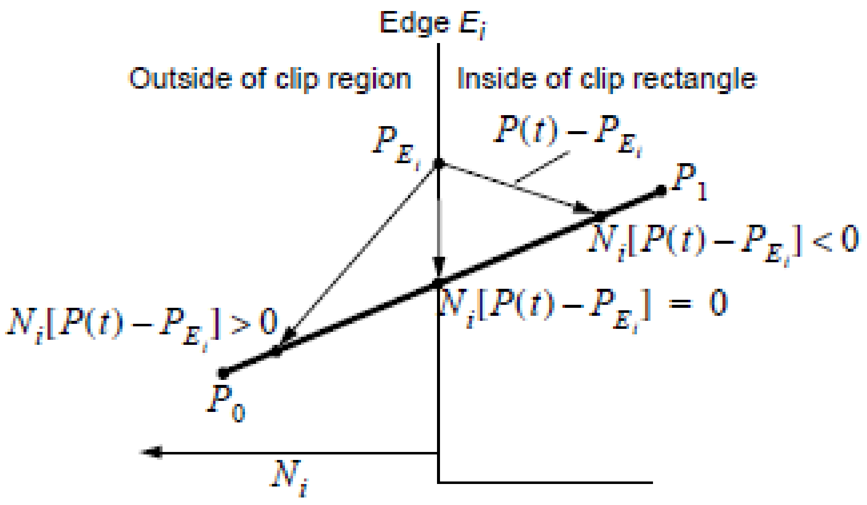


Figure 5.5: *Dot product for three points, one outside, one inside and one on the clipping rectangle*

The line from P_0 to P_1 is to be clipped at the edge E_i of the clipping rectangle with the normal N_i pointing outwards. The following parametric representation can be found for this line:

$$P(t) = P_0 + (P_1 - P_0)t \text{ with} t \in [0, 1] \tag{5.3}$$

For any point P_{E_i} on the edge E_i the scalar product $N_i \cdot (P(t) - P_{E_i})$ for all points $P(t)$ on the line is either positive, negative or equal to zero, depending on which half-plane $P(t)$ lies in relation to the edge. The required parameter of the point of intersection results from:

$$N_i \cdot [P(t) - P_{E_i}] = 0 \tag{5.4}$$

By inserting (5.1)

$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{E_i}] = 0$$

we find with $D = P_1 - P_0$

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D} \quad \text{for} \quad N_i \cdot D \neq 0 \tag{5.5}$$

The condition $N_i \cdot \neq 0$ is met if neither the normal nor the line itself have length 0 and the line and clipping edge are not parallel. Since the line generally intersects all edges of the clipping rectangle, one finds (by inserting the pairs $(P_{E_1}, N_1)...(P_{E_4}, N_4)$ into equation (5.3)) four values for the parameter t to describe the four intersections.

2. **Step:** *Determining the intersection points relevant for clipping*

First it is tested whether all $t \in [0, 1]$, otherwise the intersection is not inside the line segment $P_0 \rightarrow P_1$. The problem of determining the intersection points of the line with the clipping rectangle remains.

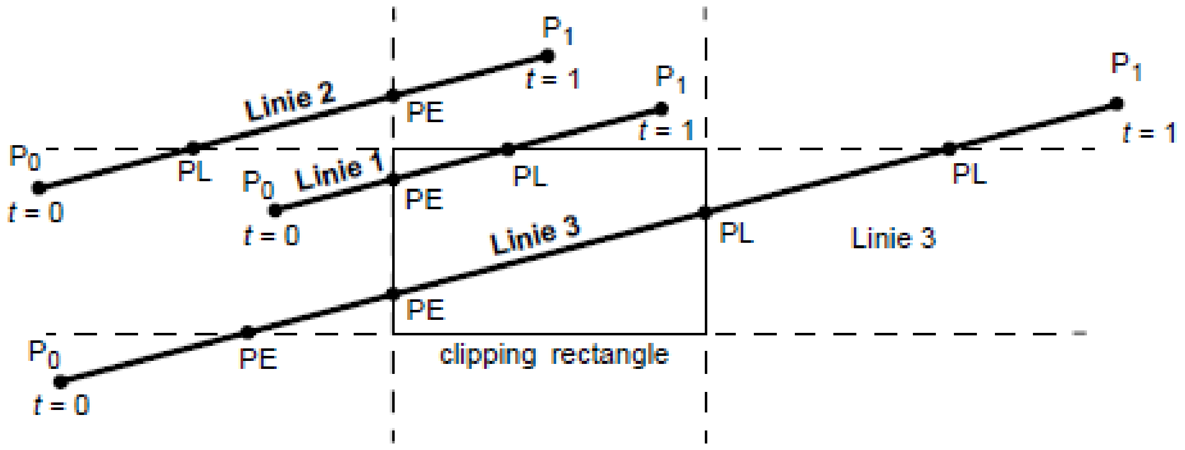


Figure 5.6: *Lines diagonal to the clipping rectangle*

In the example from Fig. 5.6, both intersection points are relevant for line 1, none for line 2 and only two of four for line 3. The box-cut test method is used to solve the problem. Each box edge defines two half-planes (inside and outside according to the definition of the normal). If you come across intersections from P_0 to P_1, which are traversed from outside to inside, then these are *Entry Points (PE)*, otherwise *Leaving Points (PL)*. The characterization of the intersection points according to PE or PL is based on the sign of the scalar product:

$$N_i \cdot D < 0 \Rightarrow PE \qquad \text{(Angle greater than } 90^\circ\text{)}$$
$$N_i \cdot D > 0 \Rightarrow PL \qquad \text{(Angle smaller than } 90^\circ\text{)}$$

After the intersection points have been characterized, it must still be determined which of the entry points and exit points lies on the boundary of the clipping rectangle. The valid entry point is the one with the largest parameter value t, the valid exit point is the one with the smallest t.

$$t_E = max\langle t_{E_i} | P_E \rangle$$
$$t_L = min\langle t_{L_i} | P_{L_i} \rangle$$

If $t_E > t_L$, there is no relevant intersection. Otherwise the clipped line is described by the interval $[t_E, t_L]$.

The algorithm according to *Cyrus-Beck* in pseudocode is as follows.

```
compute all $N_i$ and choose a $P_{E_l}$ for each edge;
/* Note that for the $P_{E_i}$ of each edge that component
is irrelevant, which is equal to 0 at the corresponding normal */

for (each line segment to clip){
    if ($P_1 == P_0$)
        Line is degenerate, so clip it as a point;
    else {
        $t_E = 0; t_L = 1;$
        **for** (every possible cut with a clip edge){
            /* ignore edges parallel to the line */
            **if** ( != 0){
                calculated;
                use the sign of Ni · D to
                to make the **case** distinction PE or PL;
                **if** (Ni · D < 0) tE = max(tE, t);/* PE */
                **if** (Ni · D > 0) tL = min(tL, t);/* PL */
            }
        }
        **if** ($t_E > t_L$)
            **return** NULL;
        **else**
            **return** P($t_E$) and P($t_L$);
    }
}
```

The calculations required for each edge of the clipping rectangle are illustrated in Fig. 5.7:

$Clip\ edge_i$	$Normal\ N_i$	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i}}{-N_i \cdot D}$
$left:\ x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
$right:\ x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{(x_0 - x_{max})}{-(x_1 - x_0)}$
$bottom:\ y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
$top:\ y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{(y_0 - y_{max})}{-(y_1 - y_0)}$

Table 5.1: *Table of calculations required for parametric clipping*

In each case one coordinate of the normal is 0. The parameter t thus results from the quotient of the distance to the respective edge and the gradient in this direction. The signs of the numerator and denominator must be preserved.

The exact algorithm according to *Liang-Barsky* is as follows:

```
/************************************************ ****************
CLIPt calculates a new value of tE or tL for an inner
Intersection of a line segment with an edge.
Parameter:
denom: −(Ni · D) (= dx or dy, for horizontal clipping rectangles),
        the sign determines whether the PL or PE case is present
num: Ni · (P0 − PEi) for a single edge−line combination,
        which, in the case of upright clipping rectangles, the directional high
```

```
            describes rizontal or vertical distance from P0 to an edge;
            the sign determines the visibility of P0 and decides
            the trivial case that lines lie parallel to the edges.
If a line segment is eliminated, FALSE is returned,
otherwise tE and tL for the part of the segment
mentes that lies inside the edge adjusted.
tE: t−value for intersection point from outside to inside
tL: t−value for intersection from inside to outside
************************************************** ***************/
boolean CLIPt (float denom, float num, float *tE, float *tL) {

            float t;
            boolean accept;

            accept = TRUE;
            if (denom > 0) {   /* PE cut */
                        t = num/denom;    /* t−value at intersection */
                        if (t > *tL)    /* tE and tL swapped */
                                    accept = FALSE;/* line is rejected */
                        else if (t > *tE)   /* a new tE was found */
                                    *tE = t;
            }
            else if (denom < 0) {/* PL cut */
                        t = num/denom;    /* t−value at intersection */
                        if (t < *tE)    /* tE and tL swapped */
                                    accept = FALSE;/* line is rejected */
                        else if (t < *tL)   /* a new tL was found */
                                    *tL = t;
            }
            else    /* line parallel to rectangle */
                        if (num > 0)    /* line outside the edge */
                                    accept = FALSE;
            return accept;
}

/************************************************** ****************
Clips a 2D line segment with endpoints x0, y0, x1, y1 against a clipping
rectangle with the corners at xmin, ymin, xmax and ymax,
which have been declared as global variables. The flag becomes visible
then set to TRUE if in the endpoint parameters a clipped
line segment is returned, otherwise FALSE.
************************************************** ***************/
void Clip2D (float *x0, float *y0, float *x1, float *y1, boolean *visible)
{
            float tE, tL;
            float dx,dy;
```

```
        dx = ∗x1 − ∗x0;
        dy = ∗y1 − ∗y0;
        ∗visible = FALSE;

        /∗ Test if line is degenerate to point; if this is the case
        is, clip point using min−max test in ClipPoint ∗/
        if (dx == 0 && dy == 0 && ClipPoint(∗x0, ∗y0))
                ∗visible = TRUE;
        else {
                tE = 0;
                tL = 1;

                if (CLIPt(dx, xmin − ∗x0, &tE, &tL))
                        if (CLIPt(−dx, ∗x0 − xmax, &tE, &tL))
                                if (CLIPt(dy, ymin − ∗y0, &tE, &tL))
                                        if (CLIPt(−dy, ∗y0−ymax, &tE, &tL)) {
                                                ∗visible = TRUE;
                                                if (tL < 1) {
                                                        /∗ Calculation of the new PL
                                                        intersection only if
                                                        really necessary ∗/
                                                        ∗x1 = ∗x0 + tL ∗ dx;
                                                        ∗y1 = ∗y0 + tL ∗dy;
                                                }
                                                if (tE > 0) {
                                                        /∗ Calculation of the new PE
                                                        intersection ∗/
                                                        ∗x0 = ∗x0 + tE ∗ dx;
                                                        ∗y0 = ∗y0 + tE ∗ dy;
                                                }
                                        }
        }
}
```

As an assessment, it can be said that parametric clipping is generally better if many lines cannot be eliminated by the trivial pre-sorting of the Cohen-Sutherland algorithm. Otherwise, both methods can be combined

5.3 Polygon clipping in 2D

As already shown in Fig. 5.2, different cases can occur when clipping polygons. In particular, concave polygons are more difficult to deal with because they can break up into several sub-polygons (Fig. 5.2a). Nevertheless, polygons can be viewed as sets of connected lines.

5.3.1 Identification of convex polygons

Definition: An n-sided polygon consisting of $\{P1, P2, ..., Pn\}$ is called *convex* if for $\forall i, j \in [1, n]$ all points on the distance P_iP_j lie within the polygon.

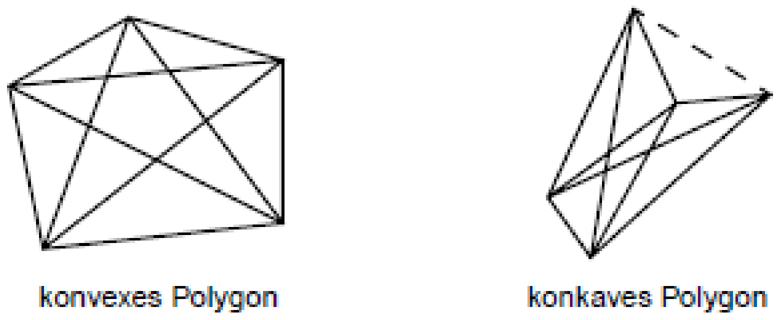


Figure 5.7: *difference between convex and concave polygon*

The check for convexity can be done via the cross products of neighboring edges:

- If all cross products are 0, then the polygon is collinear.
- If some of the cross products are greater than 0 and some are less than 0, then the polygon is concave.
- If all cross products are greater than 0 or all less than 0, the polygon is convex

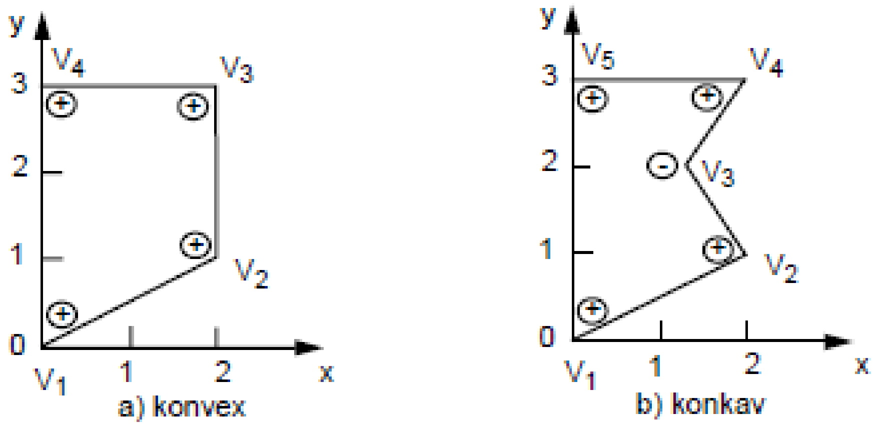


Figure 5.8: *Sign of the cross products for testing convex polygons*

Since the cross product is defined in $\Re^3$, the points have to be supplem

with a third dimension. The following applies: $\begin{bmatrix} P_{0x} \\ P_{0y} \\ 0 \end{bmatrix} \times \begin{bmatrix} P_{1x} \\ P_{1y} \\ 0 \end{bmatrix}$

5.3.2 Sutherland-Hodgeman algorithm

The following algorithm allows clipping of arbitrary polygons to convex polygons using *divide and conquer* strategies. In doing so, all edges of the convex polygon are successively clipped. Figure 5.10 shows the procedure for a rectangular clipping polygon.

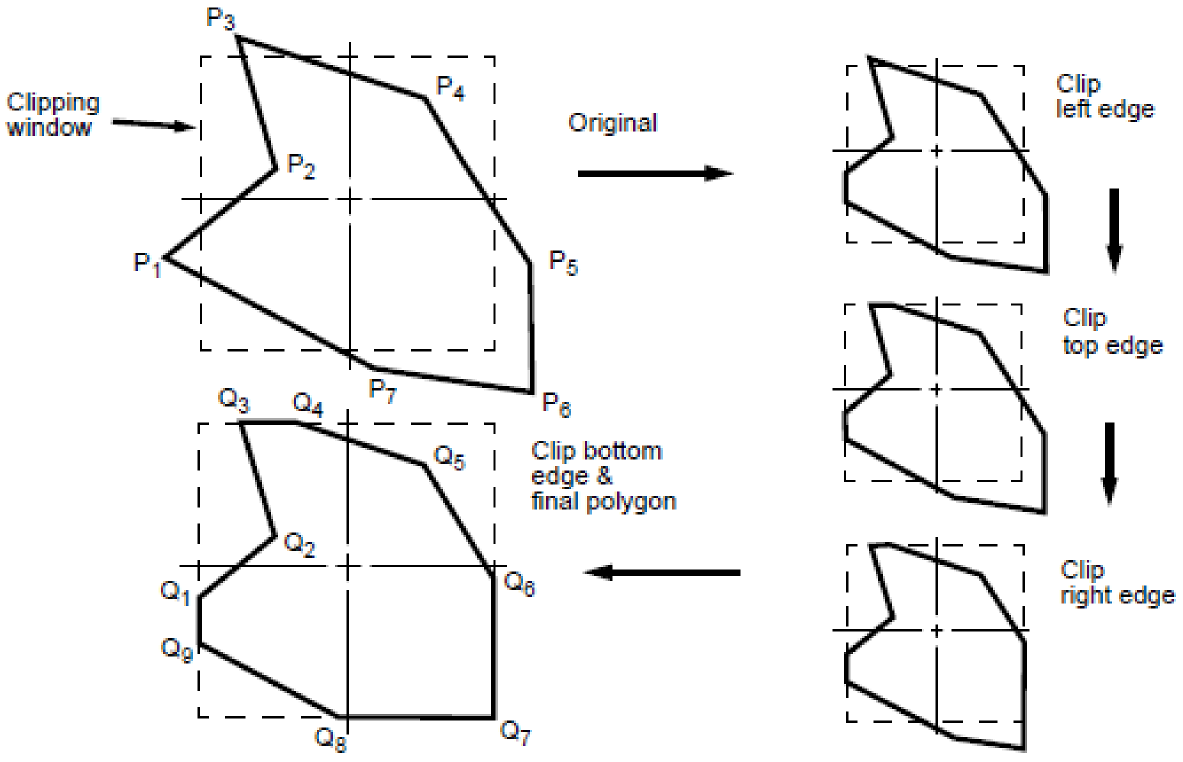


Figure 5.9: *Polygon clipping at a rectangular clipping window*

The polygon is described by a vertex list $P1, ..., P_n$ and by an edge list $\overline{P_1P_2}, \overline{P_2P_3}, ...\overline{P_nP_1}$. The successive clipping of the polygon at all edges creates new vertices and edges if necessary.

The clipping process is reduced to the calculation of the clipping of *individual* polygon edges (i.e. lines) on *individual* rectangular edges. By determining a direction of rotation, all edges receive a direction. If S is the starting point of a polygon edge and P is the end point, the four cases in Fig. 5.11 result.

During the tests, a new corner point list Q is generated, to which the output of the individual tests is added. For example, if the edge runs into the inner area of the rectangle, both the intersection point I and the end point P must be added. However, S is not included because it is output from a previous test. The list is initialized with a starting point F if this is within the rectangle. The edge P_nF is treated like any other edge.
To distinguish the four cases mentioned, a quick visibility test must be found that determines whether a point lies within the clipping polygon. This is done, for example, by testing the point against the clipping edge, as shown in the example in Fig. 5.12.

$\overline{P_1P_2}$ describes the edge, P_3 the point to be tested. The edges $\overline{P_1P_2}$ and $\overline{P_1P_3}$ span a plane.

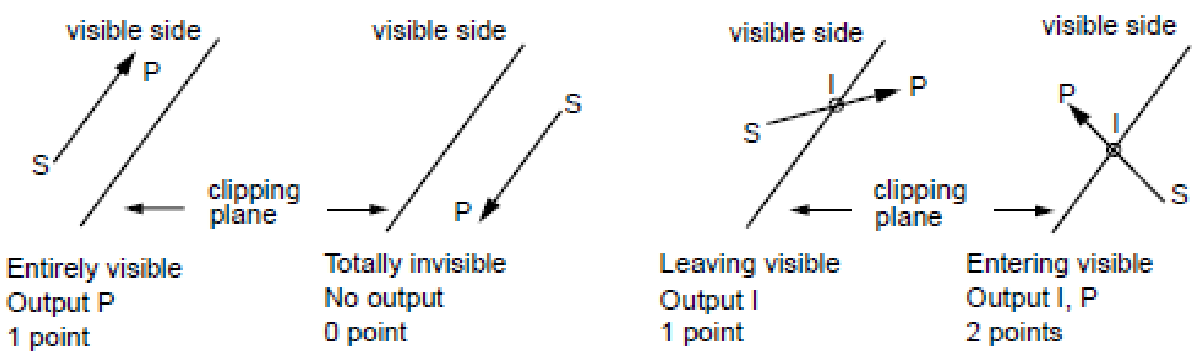


Figure 5.10: *Relationship when clipping edges on lines*

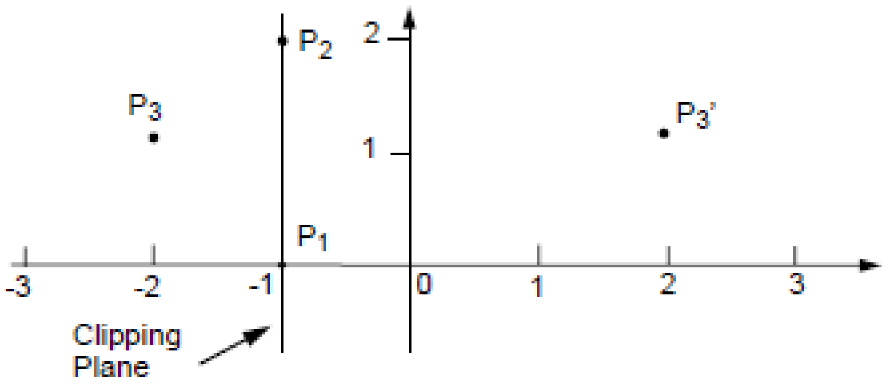


Figure 5.11: *visibility test*

If they lie in the xy plane, the cross product $\overline{P_1P_3} \times \overline{P_1P_2}$ has only one z component, which consists of $(x_3 - x_1)(y_2 - y_1) - (y_3 - y_1)(x_2 - x_1)$ results. Three cases can be distinguished:

- z component positive: The point is to the right of $\overline{P_1P_2}$
- z component equals 0: The point is on $\overline{P_1P_2}$
- z component negative: The point is to the left of $\overline{P_1P_2}$

Figure 5.13 shows the flowchart for the Sutherland-Hodgeman algorithm for each individual edge. Part a is executed for each point, part b only for the last one.

The algorithm can be implemented in hardware via pipelining. Known line algorithms that have already been presented are called up for the individual tests.

Example: Clipping a polygon at the unit square $(-1,-1),(1,1)$

The following table shows the results after clipping at the various boundary edges of the square:

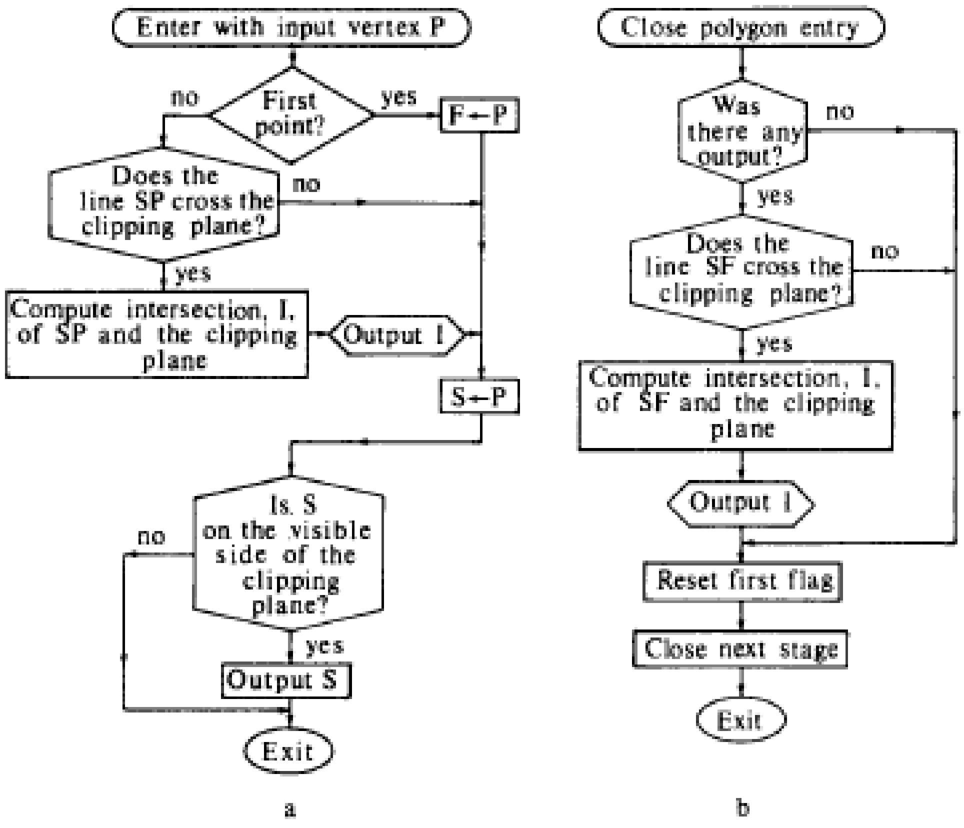
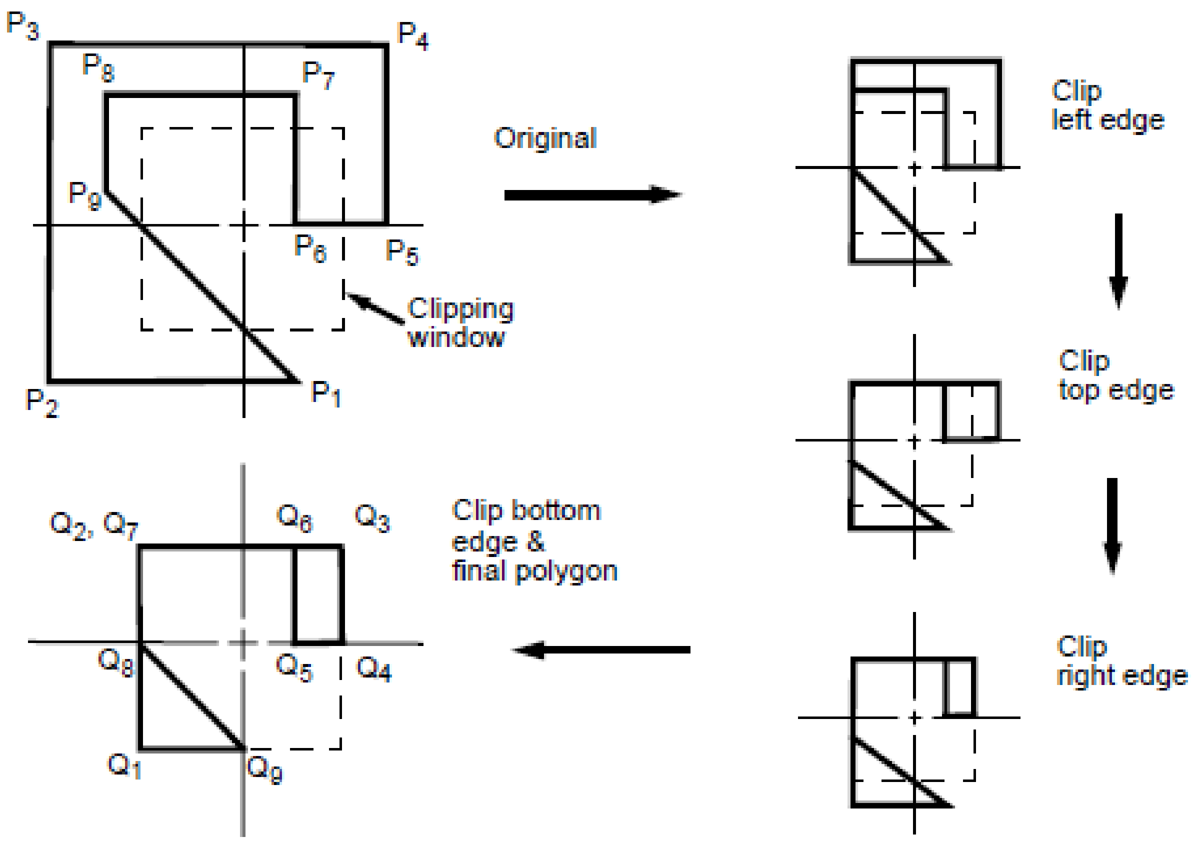

Enter with input vertex P
First point?
no
yes
F←P
Does the line SP cross the clipping plane?
no
yes
Compute intersection, I, of SP and the clipping plane
Output I
S←P
Is S on the visible side of the clipping plane?
no
yes
Output S
Exit
a
Close polygon entry
Was there any output?
no
yes
Does the line SF cross the clipping plane?
no
yes
Compute intersection, I, of SF and the clipping plane
Output I
Reset first flag
Close next stage
Exit
b

Figure 5.12: *Flowchart of the Sutherland-Hodgeman algorithm*

	Original polygon	Clipped against left edge	Clipped against top edge	Clipped against right edge	Final polgon
P_1	$(1/2, -3/2)$	$(1/2, -3/2)$	$(1/2, -3/2)$	$(1/2, -3/2)$	$(-1, 1)$
P_2	$(-2, -3/2)$	$(-1, -3/2)$	$(-1, -3/2)$	$(-1, -3/2)$	$(-1, 1)$
P_3	$(-2, -2)$	$(-1, 2)$	$(-1, -1)$	$(-1, -1)$	$(1, 1)$
P_4	$(3/2, 2)$	$(3/2, 2)$	$(3/2, 1)$	$(1, 1)$	$(1, 0)$
P_5	$(3/2, 0)$	$(3/2, 0)$	$(3/2, 0)$	$(1, 0)$	$(1/2, 0)$
P_6	$(1/2, 0)$	$(1/2, 0)$	$(1/2, 0)$	$(1/2, 0)$	$(1/2, 1)$
P_7	$(1/2, 3/2)$	$(1/2, 3/2)$	$(1/2, 1)$	$(1/2, 1)$	$(-1, 1)$
P_8	$(-3/2, 3/2)$	$(-1, 3/2)$	$(-1, 1)$	$(-1, 1)$	$(-1, 0)$
P_9	$(-3/2, 1/2)$	$(-1, 0)$	$(-1, 0)$	$(-1, 0)$	$(0, -1)$

Both degenerate vertices (Q_2, Q_7) and degenerate edges Q_7Q_6 are created.



5.3.3 Liang-Barsky polygon clipping

As with line clipping, the one-dimensional parameter space also forms the basis of calculation here. The following explanations refer to the case of window (rectangle) clipping. The algorithm uses a presectioning of the plane as in Fig. 5.14.

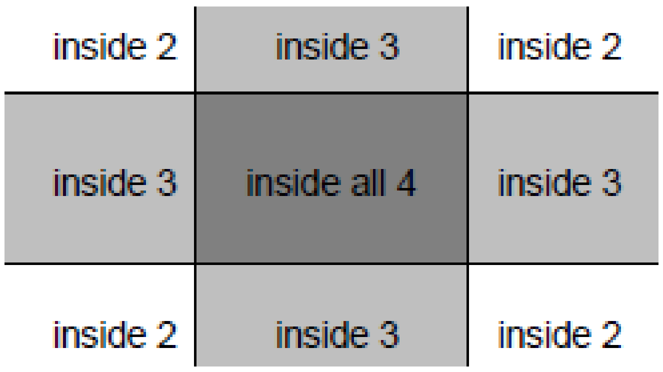


Figure 5.13: *Pre-sectioning of the plane*

Each edge of the clipping rectangle divides the plane into a half-plane containing the rectangle *(inside region)* and one not containing it *(outside region)*. The resulting nine sections are named

according to the number of inside regions they contain. The edges are also directed.
A polygon is given by a vertex list $P_1, ..P_n$ and by an edge list $\overline{P_1P_2}, ...\overline{P_nP_1}$. Each edge $\overline{P_iP_{i+1}}$ is parametrically described by

$$P(t) = (1-t)P_i + t \cdot P_{i+1} \qquad i = 1...(n-1) \qquad (5.6)$$
$$P(t) = (1-t)P_n + t \cdot P_1 \qquad i = n \qquad (5.7)$$

If the starting point of an edge to be examined is outside the rectangle, its potential intersection point depends on the position of the starting point within the described sections (Fig. 5.15).
The algorithm uses the following observation: If the starting point of a line is within the window and an end point is in the top left corner region, a subsequent edge can re-enter the window area either from above or from the right (inside 2, case b in Fig. 5.15). If the window is not entered again via the same edge that was used to leave it, the upper left window wake-up point is added as an additional corner point. In the other case, this additional corner point is superfluous, but does not interfere.

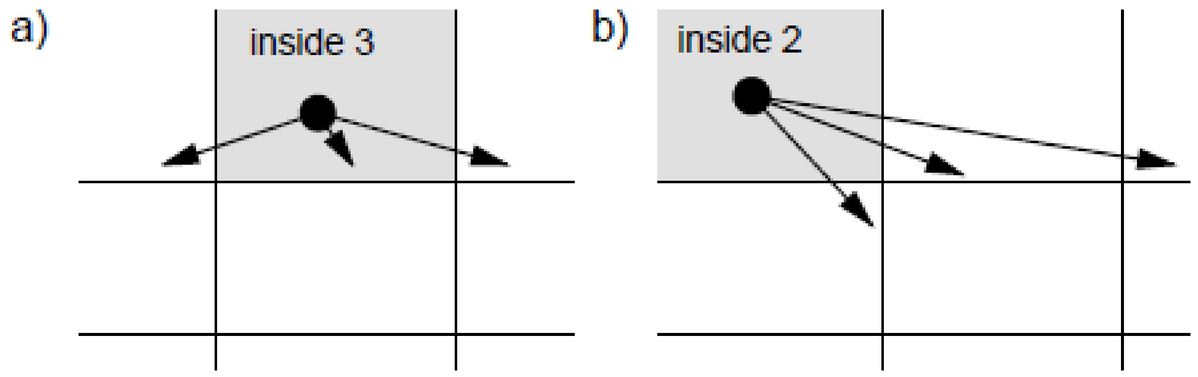


Figure 5.14: *Different positions of the starting point in the sections*

`Every time a polygon edge enters an (inside 2) region, the corresponding window corner (turning vertex) is included in the polygon list.`

The test to determine whether a window corner point is included is carried out via the parametric description of the line. As can be seen from Fig. 5.17, a line intersects the window edges at the four points $t_{in}1, t_{in}2, t_{out}1, t_{out}2$ each with two entry and two leaving -Points. $t_{in}1$ is always the smallest parameter value, while $t_{out}2$ represents the largest parameter value. The conditions for the visibility of a line in the window result in:

- $t_{in}2 > t_{out}1$ No cut in the visible area (Fig. 5.17b)
- $(t_{in}2 < t_{out}1) \wedge (0 < t_{out}1 \wedge 1 > t_{in}2)$ edges are partly within the window (Fig. 5.17a)

If the edge does not intersect the window, the descriptive straight line always begins and ends in an inside 2 section. In addition, it always runs through another *inside 2* section in the middle (Fig. 5.17). For each *inside 2* section that is run through within the edge ($t \in [0..1]$), the corresponding window corner must be included in the polygon list.

The decision about the sections that have been run through is made with the help of $t_{out}1$ and $t_{out}2$:

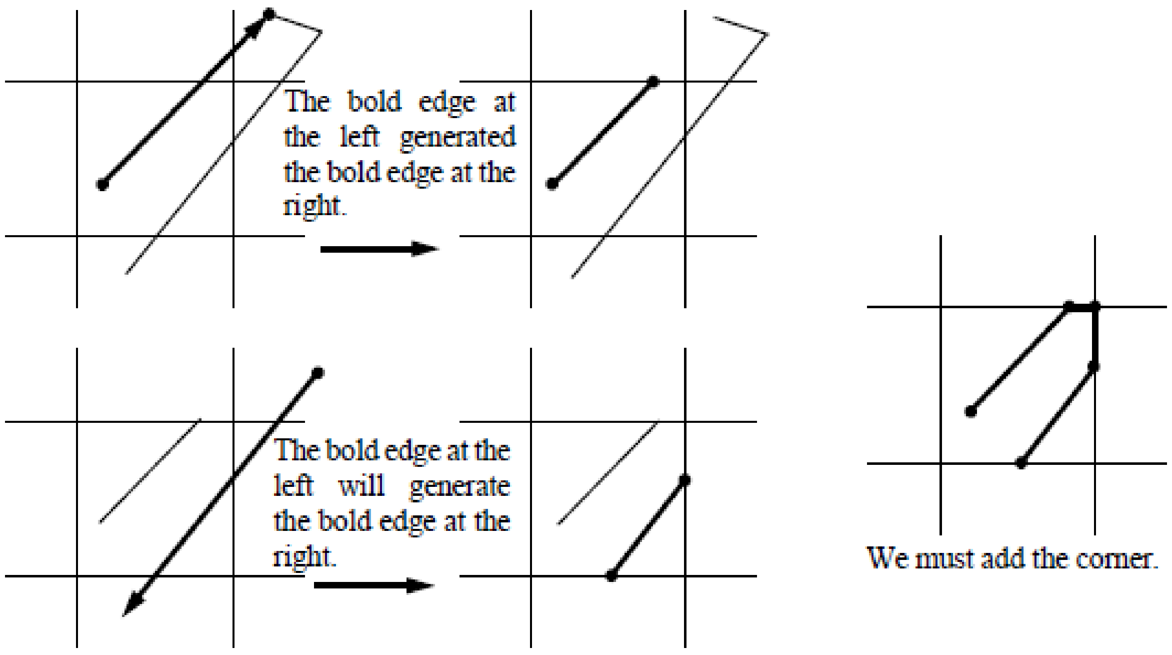

The bold edge at the left generated the bold edge at the right.
The bold edge at the left will generate the bold edge at the right.
We must add the corner.

Figure 5.15: *necessity of a point in window vertex*

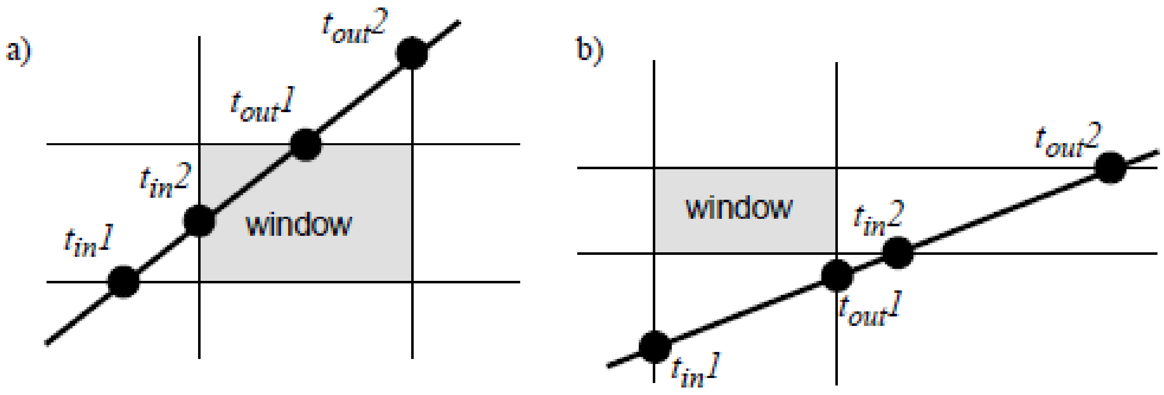


Figure 5.16: *The two possibilities of the intersection of a line with the window*

- Criterion for a traversed region: $0 < t_{out}1 <= 1$
- criterion for entering the target section: $0 < t_{out}2 <= 1$
  (also applies to lines that intersect the window)

For each polygon edge that meets the above criteria, the corresponding window corners must be included in the polygon list.
The algorithm in pseudocode is as follows:

```
for (every edge e) {
        determine the direction of the edge;
        use these to determine which boundary lines the
        clip regions that hit the straight line first;
        find t−values for exit points;
        if ($t_{out}2$ > 0)
                find t−value for second entry point;
        if ($t_{in}2$ > $t_{out}1$)
                /* no visible segment */
                if (0 < $t_{out}1$ <= 1) Output_vert(turning vertex);
        else
                if ((0 < $t_{out}1$) && (1 >= $t_{in}2$)) {
                /* visible segment available */
                        if (0 <= $t_{in}2$)
                                Output_vert(appropriate side intersection);
                        else
                                Output_vert(starting vertex);
                        if (1 >= $t_{out}1$)
                                Output_vert(appropriate side intersection);
                        else
                                Output_vert(ending vertex);
                }
        if (0 < $t_{out}2$ <= 1) then Output_vert(appropriate corner);
} /* for each edge */
```

The intersection parameters t are only calculated if necessary.

The procedure must treat vertical and horizontal lines separately. In these cases, a special value for an entry and a leaving point can be assigned based on a check $dx = 0$ or $dy = 0$.

The algorithm according to Liang−Barsky is on average twice as fast as that according to Sutherland−Hodgeman!

5.4 Line clipping in 3D

The canonical clipping volumes in 3D for parallel and perspective projections are given by the unit cube and the truncated pyramid, respectively (see also Chapter 4).

5.4.1 Cohn-Sutherland Algorithm

Similar to the 2D case, the space is divided into half-spaces using a code. A 6-bit code is used in 3D, whereby the individual bits have the following meaning for the parallel projection:

1. Bit: In the half-space above the viewing volume $y > 1$
2. Bit: In the half-space below the viewing volume $y < -1$
3. Bit: In the half-space to the right of the viewing volume $x > 1$
4. Bit: In the half-space to the left of the viewing volume $x < -1$

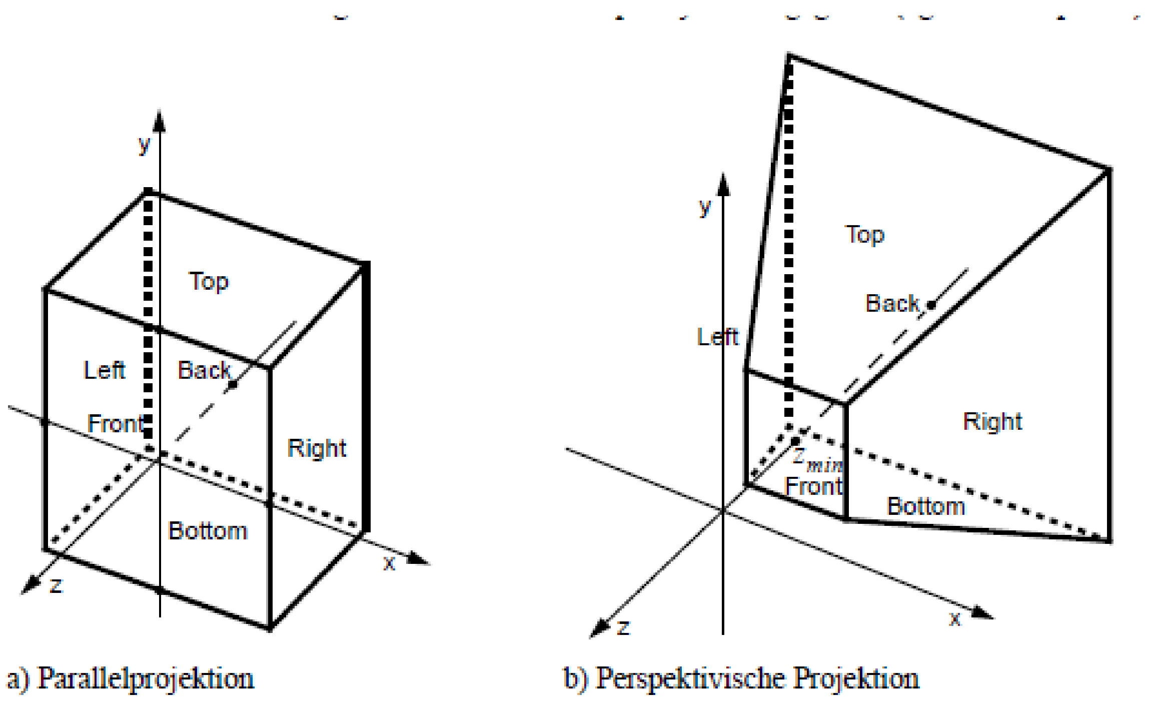


Figure 5.17: *Clipping volumes in 3D*

5. Bit: In the half-space behind the viewing volume $z < -1$
6. Bit: In the half-space in front of the viewing volume $z > 0$

The 3D lines are accepted if the code of both endpoints is equal to 000000 and eliminated if the bitwise AND of the two codes is not equal to 000000. Otherwise, the line is subdivided at the intersection with one of the bounding planes. The parametric form of the line is used for the calculation:

$$x = x_0 + t(x_i - x_0) \tag{5.8}$$
$$y = y_0 + t(y_i - y_0) \qquad 0 <= t <= 1 \tag{5.9}$$
$$z = z_0 + t(z_i - z_0) \tag{5.10}$$

Example: For an intersection with the $y = 1$ plane, $t = (1 - y_0)/(y_1 - y_0)$ if $t \in [0..1]$. Substituting into the equations (5.5) yields:

$$x = x_0 + \frac{(1 - y_0)(x_1 - x_0)}{y_1 - y_0}, z = z_0 + \frac{(1 - y_0)(z_1 - z_0)}{y_1 - y_0} \tag{5.11}$$

The code for perspective projection (using a right pyramid as the viewing volume) is:

1. Bit: In the half-space above the viewing volume $y > -z$
2. Bit: In the half-space below the viewing volume $y < z$
3. Bit: In the half-space to the right of the viewing volume $x > -z$
4. Bit: In the half-space to the left of the viewing volume $x < z$
5. Bit: In the half-space behind the viewing volume $z < -1$

6. Bit: In the half-space in front of the viewing volume $z > z_{min}$

The calculation of the intersection points with the boundary planes is just as easy as for the parallel projection.
Example: In the $y = z$ plane, $y_0 + t(y_1 - y_0) = z_0 + t(z_1 - z_0)$ is valid. One finds for t:

$$t = \frac{(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$$

and insertion into (5.5) yields for the remaining coordinates:

$$x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}, \qquad y = y_0 + \frac{(y_1 - y_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$$

5.4.2 Parametric Clipping

When clipping using the 3D variant of the Liang-Barsky algorithm, six different parameters t have to be calculated in the worst case. An optimized variant similar to the ray box test in ray tracing (see Chapter 10) is used for clipping on a cuboid (parallel projection). The transition to the viewing pyramid (perspective projection) results in new relationships for the variables $N_i, P_{E_i}, P_0 - P_{E_i}$. They are given as follows:

Clip edge	Outward normal N_i	Point on edge P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i}}{-N_i \cdot D}$
$right:\ x = -z$	$(1,0,1)$	$(x, y, -x)$	$(x_0 - x, y_0 - y, z_0 + z)$	$\frac{(x_0 - x) + (z_0 + x}{-(dx + dz)} = \frac{x_0 + z_0}{-dx - dz}$
$left:\ x = z$	$(-1,0,1)$	(x, y, x)	$(x_0 - x, y_0 - y, z_0 - z)$	$\frac{-(x_0 - x) + (z_0 - x}{(dx - dz)} = \frac{-x_0 + z_0}{dx - dz}$
$bottom:\ y = z$	$(0,-1,1)$	(x, y, y)	$(x_0 - x, y_0 - y, z_0 - y)$	$\frac{-(y_0 - y) + (z_0 - y}{(dy - dz)} = \frac{-y_0 + z_0}{dy - dz}$
$top:\ y = -z$	$(0,1,1)$	$(x, y, -y)$	$(x_0 - x, y_0 - y, z_0 + y)$	$\frac{(y_0 - y) + (z_0 + y}{-(dy - dz)} = \frac{y_0 + z_0}{-dy - dz}$
$front:\ z = z_{min}$	$(0,0,1)$	(x, y, z_{min})	$(x_0 - x, y_0 - y, z_0 + z_{min})$	$\frac{(z_0 - z_{min})}{-dz} = \frac{z_0 + z_{min}}{dz}$
$back:\ z = -1$	$(0,0,-1)$	$(x, y, -1)$	$(x_0 - x, y_0 - y, z_0 + 1)$	$\frac{-(z_0 + 1)}{dz} = \frac{z_0 - 1}{dz}$

Table 5.2: *Table of calculations needed for parametric 3D clipping*

The sign of the denominator $(N_i D)$ describes the type of intersection (PE entry point, PL leaving point). The following algorithm results. (The function $CLIPt$ is the same as in *Liang-Barsky*'s algorithm in chapter 5.2.3)

```
void Clip3D ( float *x0, float *y0, float *z0,
              float *x1, float *y1, float *z1,
              float zmin, boolean *accept )
{
        float tmin, tmax;
        float dx,dy,dz;
accept = FALSE;
t min = 0;
t max =1;
dx = ∗x1 − ∗x0;
dz = ∗z1 − ∗z0;
if (CLIPt(−dx−dz, ∗x0+∗z0, &tmin, &tmax))    /∗ right ∗/
        if (CLIPt(dx−dz, −∗x0+∗z0, &tmin, &tmax)) {   /∗ left ∗/
dy = ∗y1 − ∗y0;
                        if (CLIPt(dy−dz, −∗y0+∗z0, &tmin, &tmax))  /∗ bottom ∗/
                        if (CLIPt(−dy−dz, ∗y0+∗z0, &tmin, &tmax))/∗ top ∗/
                                if (CLIPt(−dz, ∗z0−zmin, &tmin, &tmax))/∗ front ∗/
                                        if (CLIPt(dz, −∗z0−1, &tmin, &tmax)) { /∗ back ∗/
                                                accept = TRUE;
                                                /∗ if end point (t==1) not visible,
                                                compute intersection ∗/
                                                if (t max < 1) {
                                                        ∗x1 = ∗x0 + tmax ∗ dx;
                                                        ∗y1 = ∗y0 + tmax ∗ dy;
                                                        ∗z1 = ∗z0 + tmax ∗ dz;
                                                        }
                                                /∗ if endpoint (t==0) not visible,
                                                compute intersection ∗/
                                                if (t min > 1) {
                                                        ∗x0 = ∗x0 + tmin ∗ dx;
                                                        ∗y0 = ∗y0 + tmin ∗ dy;
                                                        ∗z0 = ∗z0 + tmin ∗ dz;
                                                }
                                        }
        }
}
```

5.5 Clipping in homogeneous coordinates

The clipping algorithms are often performed in homogeneous coordinates. The perspective projection can be converted into a parallel projection using the matrix **M**.

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \qquad z_{min} \neq -1 \qquad (5.12)$$

The consequence of this is that the viewing volume degenerates into a cuboid and thus allows the use of a single (optimized) clipping procedure for the various projection types. The viewing cuboid in parallel projection is defined in homogeneous coordinates by (remember: $x = X/W, y = Y/W, z = Z/W$):

$$-1 <= \frac{X}{W} <= 1, \qquad -1 <= \frac{Y}{W} <= 1, \qquad -1 <= \frac{Z}{W} <= 0 \tag{5.13}$$

The corresponding plane equations become:

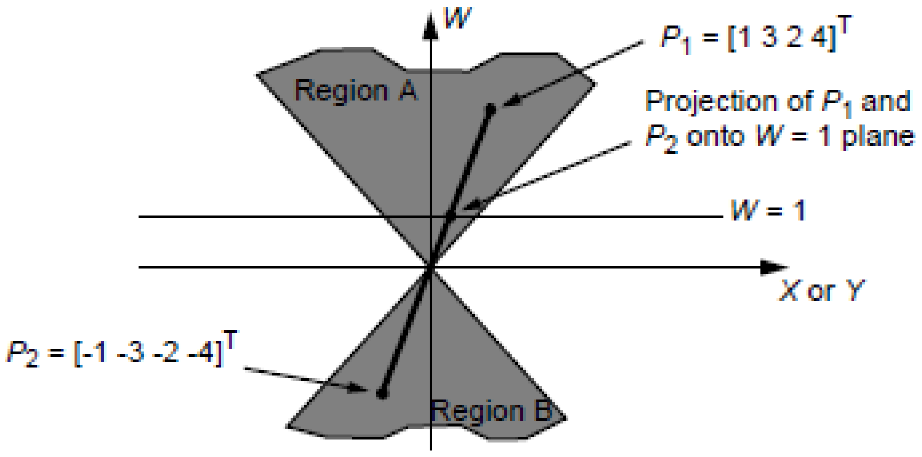
$$x = -W, \qquad X = W, \qquad Y = -W, \qquad Y = W, \qquad Z = -W, \qquad Z = 0 \tag{5.14}$$

The cases $W > 0$ and $W < 0$ must be considered separately:

$$W > 0 : \qquad -W <= X <= W, \qquad -W <= Y <= W, \qquad -W <= Z <= 0 \tag{5.15}$$

$$W < 0 : \qquad -W >= x >= W, \qquad -W >= Y >= W, \qquad -W >= Z >= 0 \tag{5.16}$$

Example in which the cases $W > 0$ and $W < 0$ are not treated separately: The points P_1 and P_2 in homogeneous coordinates have the same 3D coordinates. Clipping at Region A incorrectly eliminates P_2.


W
$P_1 = [1\ 3\ 2\ 4]^T$
Region A
Projection of P_1 and P_2 onto $W = 1$ plane
W = 1
X or Y
$P_2 = [-1\ -3\ -2\ -4]^T$
Region B

Points with any W<0 can arise in the parametric description of curves and surfaces, especially in the case of non-uniform rational B-splines (NURBS) and isolated Bézier splines (cf. Part II of the script).

Scan Conversion

Since all modern graphics systems work with raster graphics, there is a need for algorithms to convert the (transformed, illuminated and clipped) primitives into individual discrete pixels. This process is called scan conversion. Since millions of primitives per second usually have to be processed in real-time applications, computing efficiency is of the utmost importance. The pixels generally lie on a regular integer grid. For the sake of simplicity, the binary case is assumed first, i.e. a pixel can be either white or black.

6.1 Lines

The basic problem is to approximate an (infinitely) thin line as optimally as possible using a finite number of pixels of finite extent. This is trivial for line slopes $m = 0, -1, 1$ and ∞. But not with any m, as shown in Fig. 6.1.

6.1.1 First incremental algorithm (Digital differential analyzer)

The naive approach is as follows: Calculate m as $\Delta y/\Delta x$ for the line in the viewport's coordinate system (Windows) and start with the smaller x value of the two line endpoints. Let $x_0 = round(x_{min})$ be the minimum x value rounded to an integer pixel address. The pixel address is calculated by calculating the y value for an integer x_i and rounding it off. The result is the address $(x_i, round(y_i))$. Thereby x_{i+1} is calculated by incrementing x_i by δx and

$$y_{i+1} = m \cdot x_{i+1} + B = m(x_i + \delta x) + B = y_i + m \cdot \delta x$$
$$\text{with } B = y_0 - m \cdot x_0$$

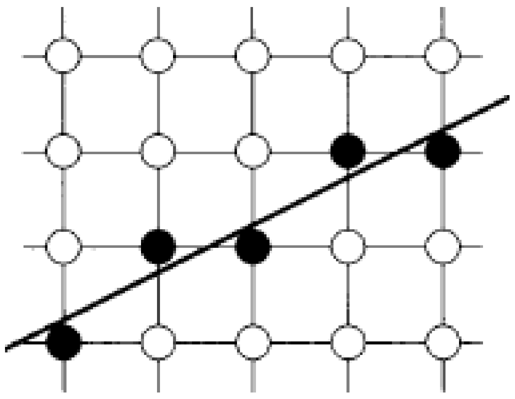


Figure 6.1: *scan-converted line*

Thus all points on the line are defined recursively, where for $\delta x = 1$ it follows:

$$
\begin{aligned}
x_{i+1} &= x_i + 1 \\
y_{i+1} &= round(y_i + m)
\end{aligned}
$$

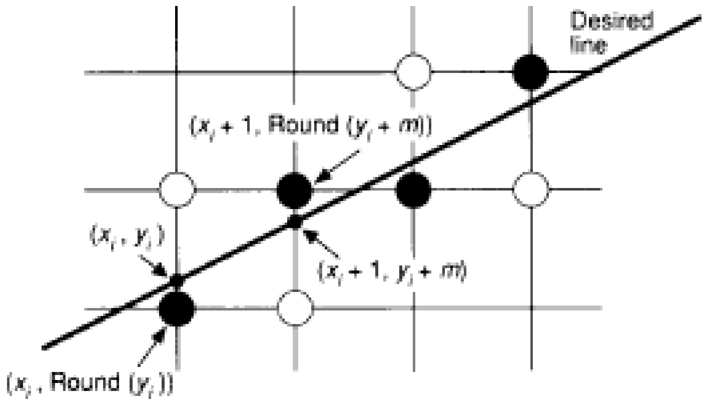

Desired line
$(x_i + 1, \text{Round}(y_i + m))$
(x_i, y_i)
$(x_i + 1, y_i + m)$
$(x_i, \text{Round}(y_i))$

Figure 6.2: *Incremental calculation of* (x_i, y_i)

The algorithm determines the pixels with the smallest distance to the straight line. For $|m| > 1$ the roles of x and y must be swapped: $dx = dy/m = 1/m$. Horizontal, vertical and diagonal lines are treated as special cases. The following algorithm results:

```
void Line(int x0, int y0, int x1, int y1) {

        int x;
        float dx,dy,y,m;

        /* Assumption: -1 <= m <= 1, x0 < x1 */
        dy = y1-y0;
        dx = x1-x0;
        m = dy/dx;
        y = y0;
        for (x = x0; x <= x1; x++) {
                WritePixel(x, round(y));
                y += m;
        }
}
```

The endpoints of the line are converted to integer addresses by rounding operations. The main disadvantage of this method is the time-consuming call of the $round()$ function, the use of float arithmetic, and a principle accumulation of errors.

6.1.2 Bresenham's algorithm

The basic idea of the Bresenham algorithm is to quickly decide for gradients $0 < m < 1$ and increments of $\Delta x = 1$ which of the two possible pixels in the next column (NE or E, see Fig. 6.3) must be set. This is done by determining the side on which the intersection Q lies with respect to the midpoint M. This always selects the pixel with the smallest distance to Q.

To calculate the midpoint criterion, the straight line is described in implicit form:

$$F(x,y) = ax + by + c = 0 \tag{6.1}$$

With the same straight line in explicit form

$$y = \frac{dy}{dx}x + B \tag{6.2}$$

is obtained with $a = dy, b = -dx$ and $c = dx \cdot B$

$$F(x,y) = dy \times x - dx \times y + dx \times B = 0 \tag{6.3}$$

In general, the implicit definition of nonlinear curves $f(x,y) = c$ or surfaces $f(x,y,z) = c$ is advantageous because it enables fast localizat tests.

To apply the midpoint criterion, given the point (x_p, y_p), the function F must be evaluated at the point M:

$$d = F(M) = F\left(x_p + 1, y_p + \frac{1}{2}\right) \tag{6.4}$$

With $d > 0$ the choice falls on the pixel NE, with $d < 0$ on E.

Since the algorithm uses the decision variable d to set the pixels, d should be able to be calculated as quickly as possible, i.e. incrementally. In the example, suppose the pixel E im has been

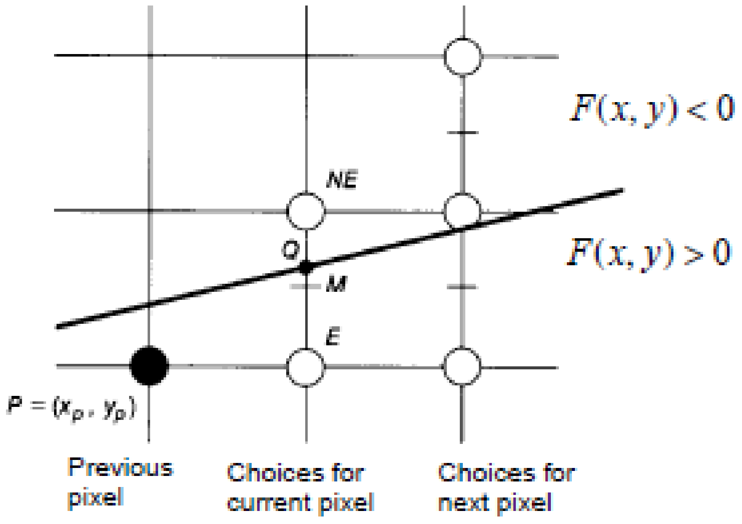


Figure 6.3: *Points M and Q, as well as pixels N and NE in the Bresenham algorithm*

selected. Then x is incremented by 1 and the midpoint criterion d_{new} for the new point results in:

$$d_{new} = F\left(x_p + 2, y_p + \frac{1}{2}\right) = a(x_p + 2) + b\left(y_p + \frac{1}{2}\right) + c \tag{6.5}$$

So d_new also results from:

$$d_{new} = d_{old} + a = a(x_p + 1) + \left(y_p + \frac{1}{2}\right) + c + a \tag{6.6}$$

From (6.5) and (6.6) the increment for d in this case becomes

$$\Delta_E = a = dy \tag{6.7}$$

An increment Δ_{NE} can be calculated in the same way:

$$d_{new} = F\left(x_p + 2, y_p + \frac{3}{2}\right) = a(x_p + 2) + b\left(y_p + \frac{3}{2}\right) + c \tag{6.8}$$

$$d_{new} = d_{old} + a + b \tag{6.9}$$

In this case, the increment results

$$\Delta_{NE} = a + b = dy - dx \tag{6.10}$$

In each step, the algorithm decides based on the sign of the decision variable d of the last step. Then one of the increments Δ_E or Δ_{NE} is added. Since the first point of the straight line is set to (x_0, y_0), the associated decision variable d is calculated

$$
\begin{aligned}
F\left(x_0+1, y_0+\frac{1}{2}\right) &= a(x_0+1)+b\left(y_0+\frac{1}{2}\right)+c \\
&= ax_0+by_0+c+\frac{b}{2} \\
&= F(x_0,y_0)+a+\frac{b}{2}
\end{aligned}
$$

This turns d_{start} into:

$$d_{start} = a + \frac{a}{2} = dy - \frac{dx}{2} \tag{6.11}$$

Multiply by a factor of 2 to avoid division

$$F(x,y) = 2(ax+by+c) \tag{6.12}$$

The *Bresenham algorithm* for lines is thus as follows:

```
void BresenhamLine(int x0, int y0, int x1, int y1) {

        int dx,dy,incE,incNE,d,x,y;

        dx = x1 − x0; dy = y1 − y0;
        d = 2∗dy − dx;
        incE = 2∗dy;
        incNE = 2∗(dy − dx);
        x = x0; y = y0;
        WritePixels(x, y);   /∗ write start pixels ∗/
        while (x < x1) {
                if (d <= 0)   /∗ choose E ∗/
                        d += incE;
                else {
                        d += incNE;   /∗ choose NE ∗/
                        y++;
                }
                x++;
                WritePixels(x, y);
        }
}
```

Fig. 6.4 illustrates the generalization for all quadrants. If $|m| > 1$ y must be incremented and a decision made on incrementing x using Bresenham's criterion. The signs of the increments differ depending on the quadrant.

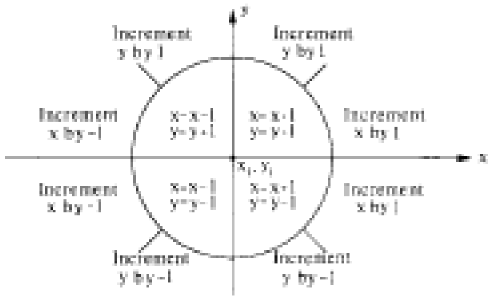


Figure 6.4: *increments in the Bresenham algorithm related to the respective quadrant*

6.2 Circles(Bresenham)

6.2.1 Derivation from implied circle equation

The circle equation can be simply described in implicit form:

$$F(x, y) = x^2 + y^2 - R^2 = 0 \tag{6.13}$$

In the following, only the second octant of a circle is considered: $x = 0$ to $x = y = R/\sqrt{2}$. The remaining decomposition of the circle follows from considerations of symmetry (see also Fig. 6.7).

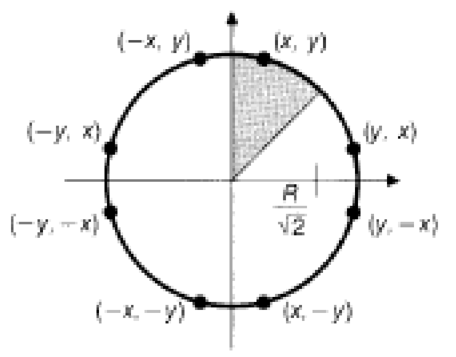


Figure 6.5: *Eight symmetrical points on the circle*

The procedure is analogous to the midpoint algorithm described by Bresenham for lines. If pixel $P(x_p, y_p)$ is found as a pixel, only E or SE can be the next pixel in the 2nd octant of a circle (see Fig. 6.6).

The implicit formulation is positive outside, zero on and negative inside the circle. If M is inside the circle, the algorithm chooses E. The increments for the decision variable d are derived as follows:

$$d_{old} = F\left(x_p + 1, y_p - \frac{1}{2}\right) = (x_p + 1)^2 + \left(y_p - \frac{1}{2})^2\right) - R^2 \tag{6.14}$$

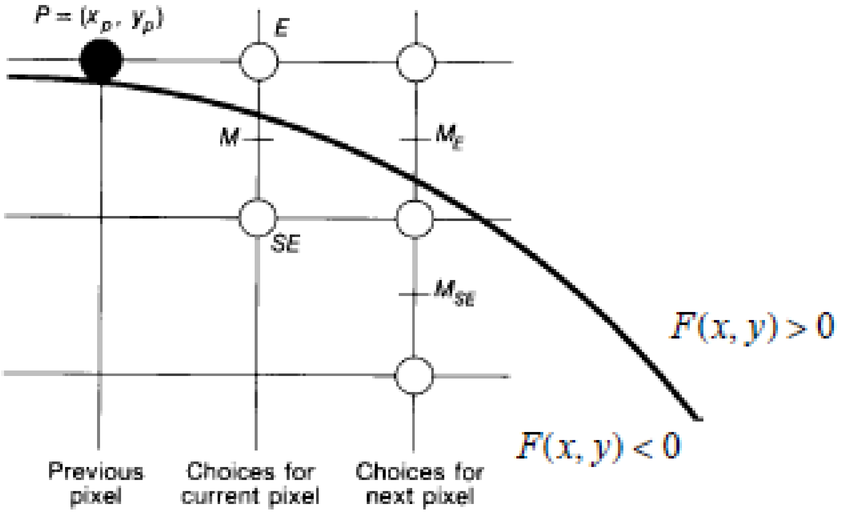


Figure 6.6: *Pixel grid with the points M, E and SE*

If $d_old < 0$, the decision goes to E and d_{new} becomes:

$$d_{new} = F\left(x_p + 2, y_p - \frac{1}{2}\right) = (x_p + 2)^2 + \left(y_p - \frac{1}{2})^2\right) - R^2 \tag{6.15}$$

From (6.14) and (6.15) one finds for the increment Δ_E:

$$\Delta_E = 2x_p + 3 \tag{6.16}$$

However, if $d_{old} > 0$, the decision falls on SE and one finds analog

$$d_{new} = F\left(x_p + 2, y_p - \frac{3}{2}\right) = (x_p + 2)^2 + \left(y_p - \frac{3}{2})^2\right) - R^2 \tag{6.17}$$

and from this the increment Δ_{SE}

$$\Delta_{SE} = 2x_p - 2y_p + 5 \tag{6.18}$$

Due to the quadratic circle equation, the increments are no longer constants but depend linearly on the previous pixel coordinates.

The procedure for drawing a circle is as follows:

1. *Step:* Draw pixels based on the previously calculated d.
2. *step:* Calculate new d according to the chosen pixel

The algorithm is initialized by a starting point located at $(0, R)$. The decision variable d is therefore initialized with (6.14) to the following value.

$$d_{start} = \frac{5}{4} - R \tag{6.19}$$

This gives the following algorithm:

```
void Circle (int radius) {

        int x,y;
        float d;

        /* assume center is in (0,0) */
        x = 0;
        y = radius;
        d = 5/4 – radius;
        CirclePoints(x,y);    /* write first circle pixels */
        while (y > x) {
                if (d < 0)    /* choose E */
                        d += 2*x + 3;
                else {     /* choose SE */
                        d += 2*(x – y) + 5;
                        y––;
                }
                x++;
                CirclePoints(x,y);
        }
}
```

6.2.2 Elimination of real arithmetic

The problem with the algorithm just described is the need for real arithmetic because the initialization value of d is not an integer. The broken expressions can be eliminated by a simple program transformation. A new decision variable h is introduced for this purpose

$$h = d - \frac{1}{4} \tag{6.20}$$

Everywhere in the code, $h + 1/4$ is now substituted for d. The initialization for h is now

$$h = 1 - R \tag{6.21}$$

and the test $d < 0$ becomes $h < -1/4$. Since h is initialized with an integer value and only incremented by integer values (Δ_E and Δ_{SE}), it can still be tested for $h < 0$. The algorithm now only works with integer values. Substituting d for h again, it reads as follows:

```
void Circle (int radius) {
        int x,y,d;

        /* assume center is in (0,0) */
        x = 0;
        y = radius;
        d = 1 − radius;
        CirclePoints(x,y);    /* write first circle pixels */
        while (y > x) {
                if (d < 0)    /* choose E */
                        d += 2*x + 3;
                else {    /* choose SE */
                        d += 2*(x − y) + 5;
                        y−−;
                }
                x++;
                CirclePoints(x,y);
        }
}
```

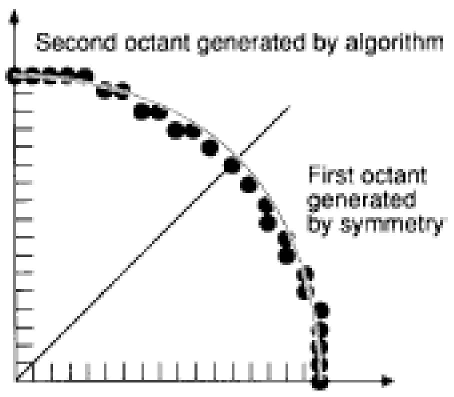

Second octant generated by algorithm
First octant generated by symmetry

Figure 6.7: *Example of creating a quadrant using symmetry*

6.2.3 Improvement by second-order partial differences

So far, the increments for the decision functions were each formed by subtracting the function values $F_{new} - F_{oled}$. This can be viewed as an approximation of the first derivative of this function. In the case of the circle, the first-order finite differences are linear functions. (x_p, y_p) can be used to calculate the second-order difference. If you choose the point E in the current iteration, the point moves from (x_p, y_p) to $(x_p + 1, y_p)$. The first order difference is calculated as before $\Delta_{E_{old}} = 2xp + 3$. At the new point it surrenders to

$$\Delta_{E_{new}} = 2(x_p + 1) + 3$$

So for the second-order difference one finds here

$$\Delta_{E_{new}} - \Delta_{E_{old}} = 2 \tag{6.22}$$

In the same way one finds with the corresponding first-order differences

$$\Delta_{SE_{old}} = 2x_p - 2y_p + 5$$
$$\Delta_{SE_{new}} = 2(x_p + 1) - 2y_p + 5$$

the second-order difference

$$\Delta_{SE_{new}} - \Delta_{SE_{old}} = 2 \tag{6.23}$$

If the point SE is chosen in the current iteration, the point moves from (x_p, y_p) to (x_p+1, y_p-1). Then arise analogously

$$\Delta_{SE_{new}} = 2(x_p + 1) + 3$$

and thus

$$\Delta_{E_{new}} - \Delta_{E_{old}} = 2 \tag{6.24}$$

such as

$$\Delta_{SE_{new}} = 2(x_p + 1) - 2(y_p - 1) + 5$$

and

$$\Delta_{SE_{new}} - \Delta_{SE_{old}} = 4 \tag{6.25}$$

The algorithm then undergoes the following changes:

1. *step:* Choose pixel E or SE based on the sign of dold.
2. *Step:* Increment d with Δ_E or Δ_{SE} (current values calculated during the previous iteration).
3. *step:* Increment the two increments Δ_E and Δ_{SE} with constants corresponding to the newly selected pixel.
4. *Step:* Draw the selected pixel

Δ_E and Δ_{SE} are initialized via the start pixel $(0, R)$.

```
void Circle (int radius) {

        int x, y, d, deltaE, deltaSE;

        /* assume center is in (0,0) */
        x = 0;
        y = radius;
        d = 1 − radius;
        delta E = 3;
        deltaSE = −2*radius + 5;
        CirclePoints(x, y);   /* write first circle pixels */
        while (y > x) {
                if (d < 0) {    /* choose E */
                        d += deltaE;
                        deltaE += 2;
                        deltaSE += 2;
                }
                else {    /* choose SE */
                        d += deltaSE;
                        deltaE += 2;
                        deltaSE += 4;
                        y−−;
                }
                x++;
                CirclePoints(x, y);
        }
}
```

6.3 Ellipses

Similarly, the Bresenham algorithm can also be generalized to ellipses. To do this, consider the implicit definition of an ellipse at the origin

$$F(a,b) = b^2x^2 + a^2y^2 - a^2b^2 = 0 \tag{6.26}$$

where 2a and 2b are the axis lengths (Fig. 6.8).

Due to the symmetry properties, only the 1st quadrant is considered. This is divided at a point of slope -1 (tangent at an angle of 45 degrees) (Fig. 6.9). For this, the gradient of the curve is used, which is perpendicular to the tangent. It can be represented by the unit vectors i and j as follows.

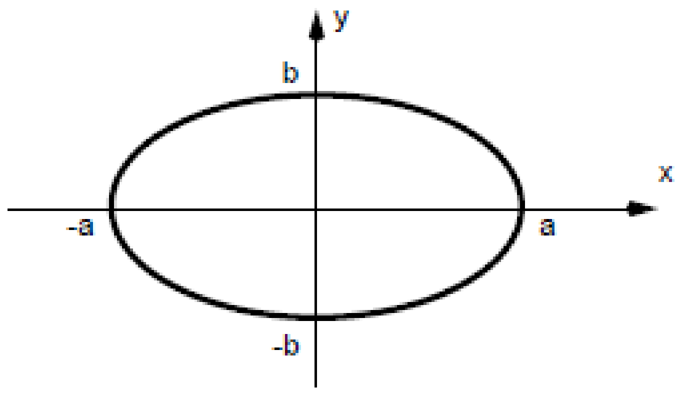


Figure 6.8: *Standard ellipse at origin*

$$\nabla F(x,y) = \frac{\delta F}{\delta x} i + \frac{\delta F}{\delta y} j = 2b^2 x i + 2a^2 y j$$

The gradient has the slope 1 in the searched point, which means that the two components i and j of the vector are equal (Fig. 6.9). This results in the following assignment of a point to a region:

$$a^2(y_p) > b^2(x_p) \leftarrow Region1$$
$$a^2(y_p) < b^2(x_p) \leftarrow Region2$$

To calculate the decision variable d_1 in region 1, F is evaluated at the point $(x_p + 1, y_p - 1/2)$ (midpoint between E and SE). The partial differences Δ are calculated as follows:

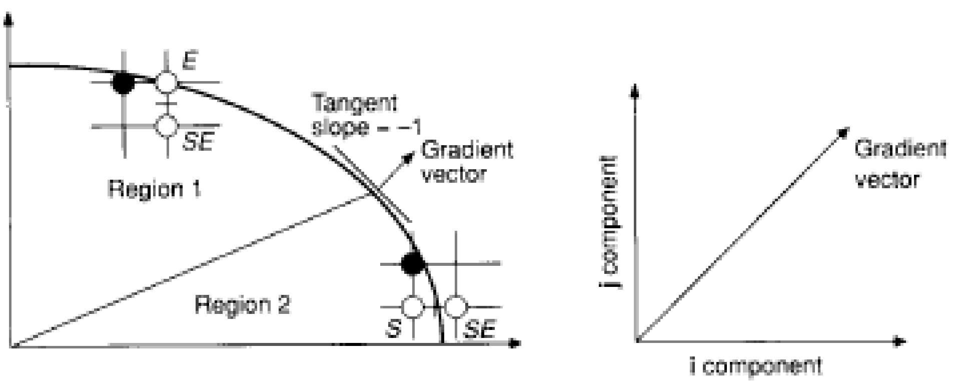


Figure 6.9: *Two regions of an ellipse defined by the* $45°$ *tangent*

Choosing E gives

$$d_{old} = F\left(x_p+1, y_p-\frac{1}{2}\right) = b^2(x_p+1)^2 + a^2\left(y_p-\frac{1}{2}\right)^2 - a^2b^2$$
$$d_{new} = F\left(x_p+2, y_p-\frac{1}{2}\right) = b^2(x_p+2)^2 + a^2\left(y_p-\frac{1}{2}\right)^2 - a^2b^2$$

and thus

$$\Delta_E = d_{new} - d_{old} = b^2(2x_p+3) \tag{6.27}$$

On the other hand, if SE is selected, it is found

$$d_{new} = F\left(x_p+2, y_p-\frac{3}{2}\right) = b^2(x_p+2)^2 + a^2\left(y_p-\frac{3}{2}\right)^2 - a^2b^2$$

and thus

$$\Delta_{SE} = b^2(2x_p+3) + a^2(-2y_p+2) \tag{6.28}$$

In region 2, the decision variable d_2 is calculated at the point $(x_p+1/2, y_p-1)$ (midpoint between S and SE). The increments are calculated in the same way as for region 1. The results can be found in the following program code.

The initialization is done assuming that a and b are integers and the starting point is given by $(0, b)$. The first center point to be calculated becomes $(1, b-1/2)$. The decision function takes the initial value there

$$F\left(1, b-\frac{1}{2}\right) = b^2 + a^2\left(b-\frac{1}{2}\right)^2 - a^2b^2 = b^2 + a^2\left(-b+\frac{1}{4}\right) \tag{6.29}$$

on.

In each iteration step it must be checked whether region 1 has been left. If so, the decision variable d_2 must be initialized with $(x_p+1/2, y_p-1)$, where (x_p, y_p) denotes the last point in region 1.

The algorithm ends at $y=0$.

The following program code shows the scan conversion algorithm for ellipses using real arithmetic and first-order differences. In the case of integer a and b, real arithmetic can again be circumvented by a program transformation similar to that for circles (see 6.2.2). Furthermore, instead of the direct calculation of the increments, a calculation using second-order differences could also take place.

```
void ellipse(int a, int b) {

    int x,y;
    float d1,d2;

    /* Assume the center of the ellipse is in (0,0) */
    x = 0; y = b;
    d1 = b*ba*a*b+a*a/4;
    EllipsePoints(x,y);   /* write first ellipse pixels */
    while (a*a*(y–0.5) > b*b*(x+1)) {   /* region 1 */
        if (d1 < 0)      /* choose E */
            d1 += b*b*(2*x + 3);
        else {    /* choose SE */
            d1 += b*b*(2*x + 3) + a*a*(–2*y + 2);
            y––;
        }
        x++;
        EllipsePoints(x, y);
    }

    d2 = b*b*(x + 0.5)*(x + 0.5) + a*a*(y − 1)*(y − 1) − a*a*b*b;
    while (y > 0) {     /* region 2 */
        if (d2 < 0) {      /* choose SE */
            d2 += b*b*(2*x + 2) + a*a*(–2*y + 3);
            x++;
        }
        else       /* choose S */
            d2 += a*a*(–2*y + 3);
        y––;
        EllipsePoints(x,y);
    }
}
```

6.4 Scan conversion of polygons

Scan conversion of polygons is done by filling the primitive with pixels as efficiently as possible. The simplest way is to determine for each pixel whether it is inside the polygon and set it if so (*inside tests*). Of course, this is too inefficient, so people are looking for faster methods. Scan conversion of polygons is an elementary functionality included in modern graphics hardware systems.

First the concept of the *span* is introduced and only the binary case is considered. A span is an area of set pixels within a *scan line* (pixel line). The spatial coherence of the scan line is exploited by spans. Transitions between set and unset pixels can only occur at the intersections of the polygon edge with the scan line. In the example in Fig. 6.10, the scan line 8 intersects the polygon four times in a, b, c and d. There are 2 spans from 2 to 4 and from 9 to 13.

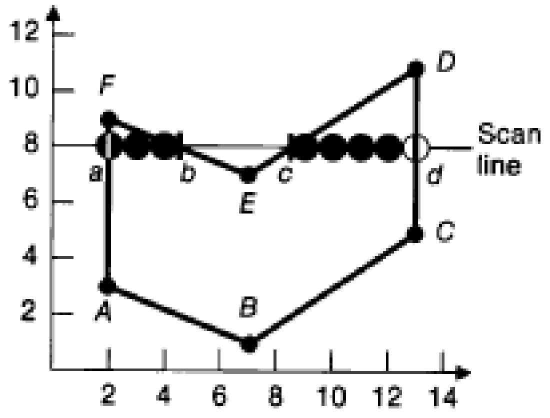


Figure 6.10: *polygon and scan line*

For this reason, the well-known scan conversion algorithms for lines (Bresenham) can be used to calculate the intersection points of each polygon edge with the scan line - i.e. the extrema of the resulting spans

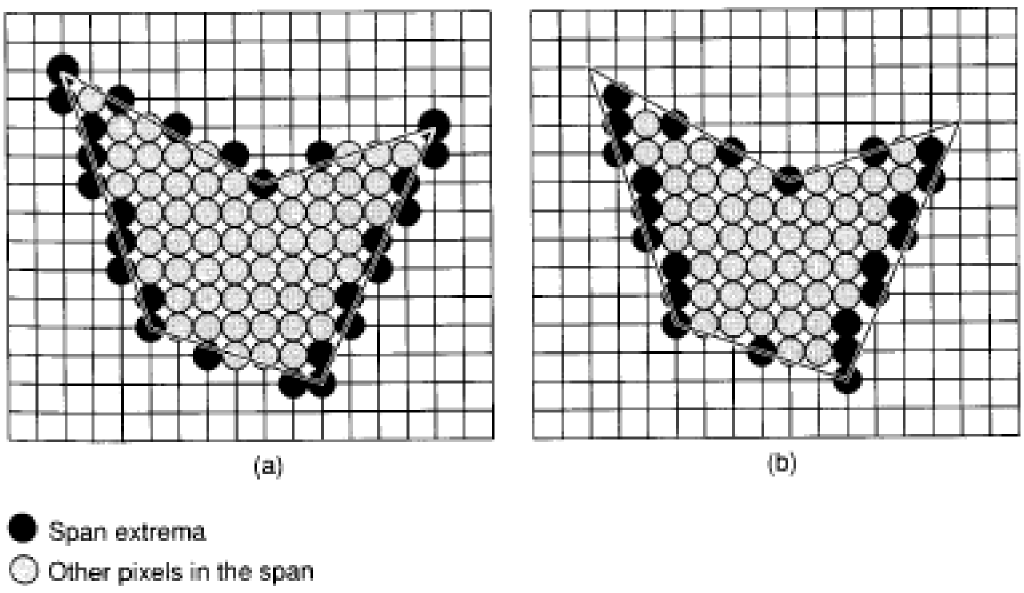


Figure 6.11: *Span of a polygon: extrema are shown in black, inner points are grey:*
a) Extrema calculated with midpoint algorithm (Bresenheam)
b) extrema within the polygon

However, this can lead to incorrect results at polygon corners, since the line algorithm selects the pixels that are closest to the line, regardless of which side of the line they are on (Fig. 6.11 (a)). The algorithm has to be modified to find the correct extrema *inside* the polygon as in figure (b). Otherwise, the current polygon's region would overlap with regions of immediately adjacent polygons.

6.4.1 A three-step algorithm

I. *Step:* Calculate the intersection points of all polygon edges with the scan line

II. *Step:* Sort the intersection points by increasing x-coordinates and number them starting at 0.

III. *Step:* Fill the spans between two consecutive intersections, provided the number of the intersection has parity *odd* (odd). Outside the polygon, the parity is *even* (even) and is inverted at each intersection.

This algorithm has the following problems:

a. Treatment of the edge pixels at any (real) intersection between two integer coordinates. Which pixels are inside the polygon?

b. Handling of integer intersections (intersections whose coordinates take integer values). Are they inside or outside the polygon?

c. Treatment of common vertices of two intersecting lines

d. Handling of horizontal edges in integer case

To solve the problems described, the following procedure is suggested:

a. If you approach an intersection from the left and are inside the polygon, you will be rounded off. If you are outside, round up.

b. If the first pixel from the left of a span is an integer intersection then the pixel is set, otherwise not.

c. Only the vertex with the smaller y-coordinate of an edge is included in the parity calculation. Thus the lines are open at their ymax endpoints. In the example in Fig. 6.10, this looks like this:

   - In scan-line 3, A counts only once because A is the ymax vertex of AB and the ymin vertex of FA;
   - In scan-line 1 vertex B counts twice because it is ymin vertex of AB and BC (zero span, only one pixel drawn)
   - in Scan-line 9, however, F is not set because F is twice the ymax vertex.

d. Pixels are placed on the lower horizontal edges, but not on the upper ones. This is done automatically if the vertices are not included in the parity calculation.

6.4.2 Horizontal edges

The vertices of horizontal edges are not counted in the integer case. With the y_{min} criterion for parity calculation, this means that lower edges are recorded, but upper ones are not.

That horizontal edges are treated correctly can be shown using the different cases in the example in Fig. 6.12:

Edge AB:	vertex A is for edge YES y_{min} vertex and AB is not counted. Therefore the parity is *odd*, and the edge is drawn as a span to B. This is the y_{min} vertex for BC, which makes the parity *even* again.
Edge CD:	The span starts at IJ and the vertices of CD are not counted. Therefore the parity remains *odd* and the edge is drawn to DE where the parity becomes *even* again.
Edge IH:	I is y_{max} vertex of IJ and IH is not counted. This leaves the parity *even* and the edge is not drawn.
Edge GF:	G is y_{max} vertex of HG and GF is not counted. This keeps the parity even and the edge is not drawn.

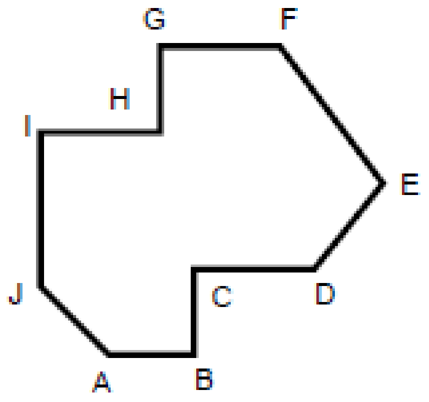


Figure 6.12: *Horizontal edges in a polygon*

6.4.3 Problems with long thin polygons (slivers)

By the above rules, only pixels inside or on lower left edges are drawn. This creates so-called *slivers*. These are places where the polygon area is so thin that the interior does not have a span on each scan line that results in a pixel being drawn. This can lead to aliasing effects. The gradation is clearly recognizable.

6.4.4 Edge coherence

Calculating the intersections of all edges with each scan line is very time consuming. Given the intersection x_i for an edge with the scan line i, the intersection with the line $i + 1$ can be calculated incrementally:

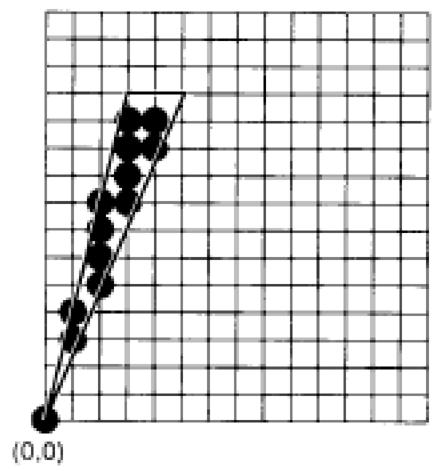


Figure 6.13: *Scan conversion of Silvers*

$$x_{i+1} = x_i + \frac{1}{m}, \qquad \text{where } m = \frac{(y_{max} - y_{min})}{(x_{max} - x_{min})} \tag{6.30}$$

Consider left edges with a slope $m > 1$:

- A pixel is set in the bottom left corner (x_{min}, y_{min}) (x-start).
- Incrementing y by 1 (new scanline) increases x-start for that scanline by a constant fraction of $1/m$. As a result, x now has an integer and a fractional part.
- As soon as the fractional part becomes > 1, x-Start for the current scan line can be incremented and the fractional part decremented by 1.

To avoid real arithmetic, m is divided into numerator and denominator (integer values). The fractional portion now overflows when the numerator is greater than the denominator. This gives the following algorithm for scan conversion of a left polygon edge.

```
void LeftEdgeScan(int xmin, int ymin, int xmax, int ymax) {

        int x, y, num, denom, inc;

        x = xmin;
        num = xmax – xmin;
        denom = ymax – ymin;
        inc = denom;

        for (y = ymin; y <= ymax; y++ ) {
                WritePixels(x,y);
                inc += number;
                if (inc > denom) { /* Overflow, so round up */
                        x++;
                inc −= denom;
            }
        }
}
```

The treatment of right edges and other slopes is done with similar, albeit slightly more tricky, arguments. Vertical edges are a special case. Horizontal edges are implicitly handled by the span rules, as shown above.

6.4.5 The Active Edge Table data structure (AET)

The AET contains all edges that intersect the scan line, sorted in ascending order according to the x coordinates of the intersection points. From this, the extrema defining a span can be easily determined and the span can be drawn. The AET is updated for each new scan line. To do this, the edges that no longer intersect the new scan line $(y = y_{max})$ are eliminated and new ones $(y = y_{min} + 1)$ are inserted. The x coordinates of the intersections of all remaining edges are updated.

In order to access and update the AET efficiently, a global *Edge Table (ET)* is created at the beginning, which contains all edges sorted in ascending order according to their smaller y coordinate (y_{min}) . It is constructed using *bucket sorting* (separate hashing), with the list containing as many buckets as scan lines.

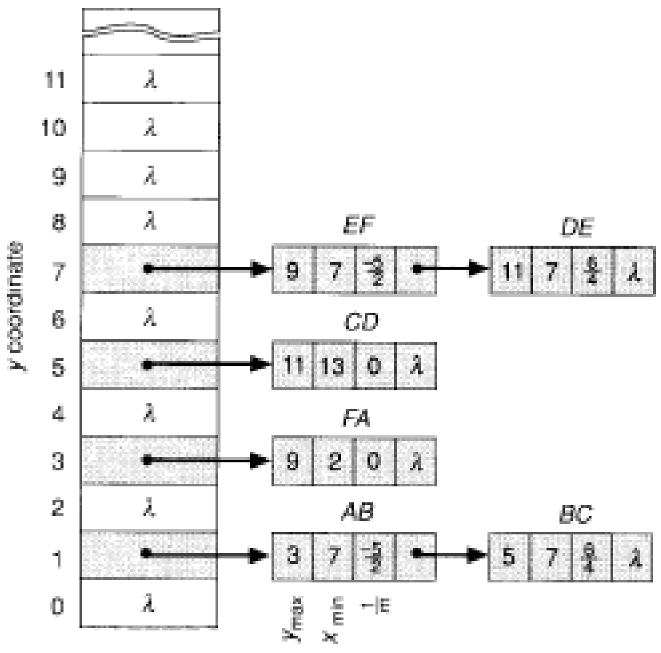


Figure 6.14: *ET for the example polygon from Fig. 6.10*

In each bucket are held all edges whose y_{min} coordinates correspond to the corresponding bucket, sorted in ascending order by their x_{min} coordinate (x coordinate of their lower endpoint). Each entry contains the y_{max} coordinate for the edge, the x_{min} coordinate and the increment of x for the transition to the next scan line $(1/m)$.

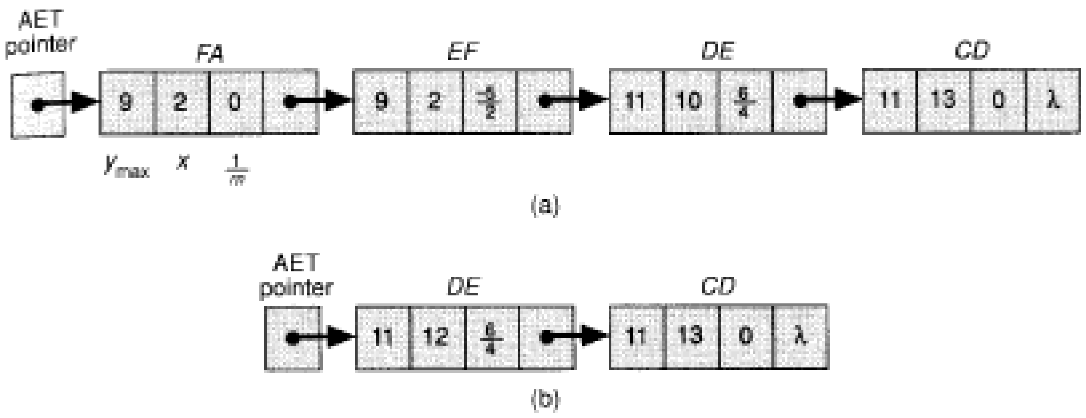


Figure 6.15: *AET for scan lines 9 and 10 from Fig. 6.10*

The algorithm can thus be described as follows:

I. Generate ET by sorting all edges by y_{min} coordinate and concatenating by increasing x_{min} coordinate

II. Search for first y value (scan line) containing items

III. Initialize AET to empty state

IV. Repeat until AET and ET are empty:

   .1 Move all items with $y = y_{min}$ from ET to AET, preserving the sorting by x_{min}

   .2 Eliminate from AET all items with $y = y_{max}$

   .3 Draw all spans of scan line y using x coordinate pairs from AET

   .4 Increase y by 1 (next scan-line)

   .5 Increment the x coordinate in the item for each non-vertical edge of the AET

   .6 Reorder the AET by increasing x coordinates

It should be noted that in step 4.6 the AET must be sorted again. However, since in general only little has changed compared to the last step, special sorting methods (insertion, bubble) of complexity $O(N)$ can be used in this case.

In the case of multiple polygons, the spans are overlaid (accumulated).

When using Gouraud shading (see Chapter 8) within a span, the intensities between two extremes are interpolated. In this case, the AET and ET can be supplemented accordingly with intensity increments.

Hidden Line and Hidden Surface Algorithms

The efficient calculation of hidden lines and areas is of essential importance in graphic data processing, because the order in which lines, polygons or polygon parts are drawn depends on the current overlap and the viewer's point of view. Only ray tracing (see Chapter 10) deals with this problem implicitly by setting the intersection point with the smallest distance in the z direction to the viewer for opaque objects (Z buffering).
In the following chapter, the most important algorithms will be examined in more detail.

7.1 Hidden Line Algorithms

7.1.1 Back Face Culling

In the case of polygonal object descriptions, the individual polygons enclose the entire object volume. If the normal is defined as outward, then the polygons whose normals point away from the viewer are in a non-visible area (back-facing polygons).

The test is carried out using the sign of the scalar product from the normal **N** and the projection direction **R**. For orthographic projections into the xy plane, $\mathbf{R} = (0, 0, -1)$, so the sign of the z coordinate of the normal is crucial. If only one object is to be drawn, this test is sufficient.

Since a projection ray in the case of polyhedron objects always cuts exactly as many front-facing as back-facing polygons, the number of polygons to be examined (converted) is reduced

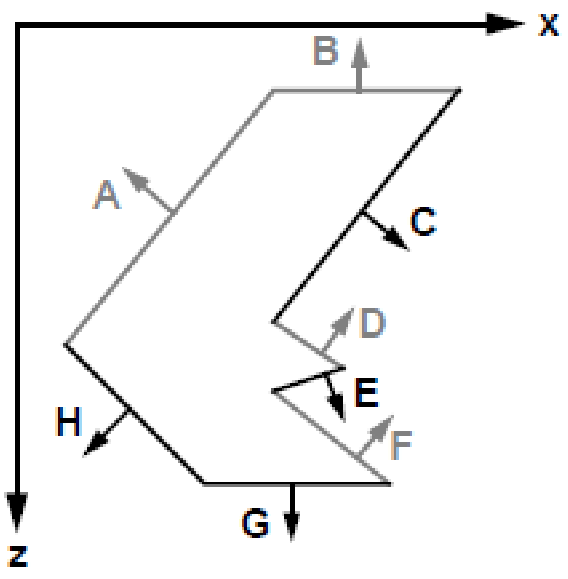


Figure 7.1: *Illustration of back face culling:*
grey: Back-facing polygons
black: Front-facing polygons

by a factor of 2 on average. This knowledge is also often used as a preprocessing step, which can be done with hardware support.

7.1.2 Appel's algorithm

A typical algorithm based on back-face culling is the Appel method. Only edges of front-facing polygons are considered. A prerequisite for the algorithm is that edges can be combined into polygons. The further propagation of visibility information of an edge at intersections with other edges (edge coherence) is helpful for the process.

A *quantitative invisibility* is introduced for points on a polygon edge. If a line goes behind a front-facing polygon, its quantitative invisibility is incremented, if it comes out again, it is decremented. Only points whose invisibility is 0 are visible. The line AB in Fig. 7.2 is divided into individual segments. As long as no penetrating polygons are allowed, the invisibility of a line only changes if it goes behind so-called *contour lines*. A contour line is either an edge shared by a front-facing and a back-facing polygon, or an edge of a front-facing polygon that is not part of a closed polyhedron. In Fig. 7.2, the edges $\overline{AB}$, $\overline{CD}$, $\overline{DF}$ and $\overline{KL}$ are contour lines, while $\overline{CE}$, $\overline{EF}$ and $\overline{JK}$ are none.

A contour line runs in front of an edge if it pierces the eyepoint-polygon edge triangle.

`The intersection of the contour line with the edge (projection onto the edge) can be calculated by clipping the edge on the plane of eye point and contour line.`

This results in the following algorithm:

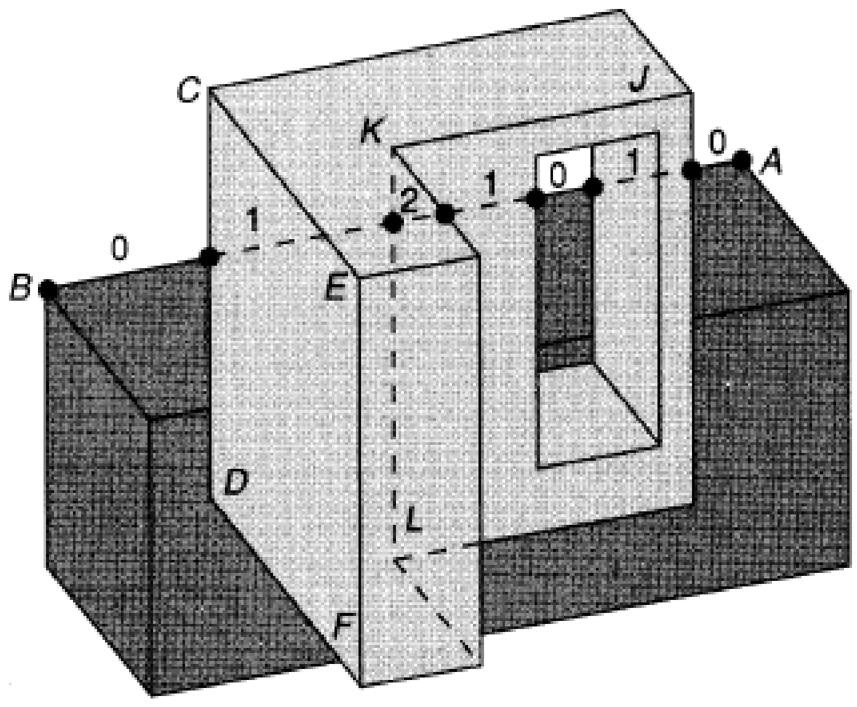


Figure 7.2: *Quantitative invisibility of lines*

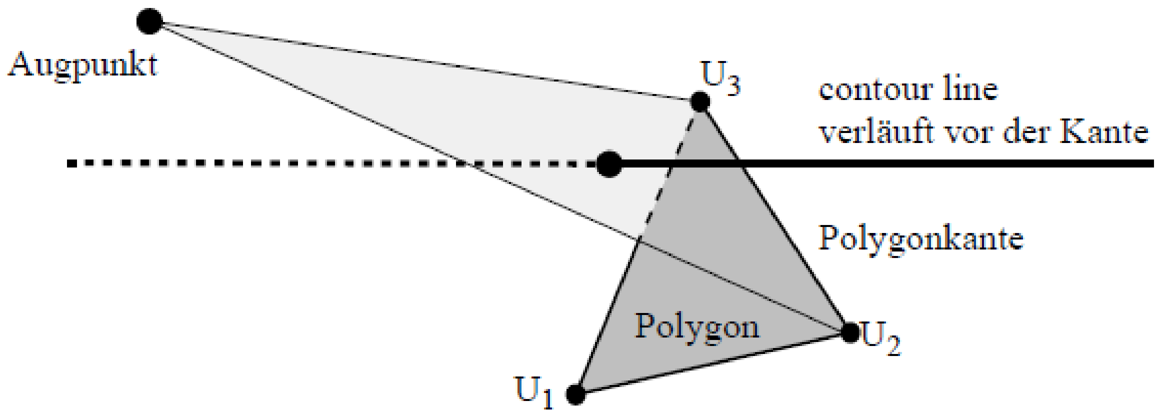


Figure 7.3: *Contour line runs in front of the polygon edge*

1. Pick an initial edge vertex on an arbitrary polygon (of an object), where the edge is sensibly a common edge of a front- and a back-facing polygon.
2. Determine the initial value for the invisibility by brute-force testing against all front-facing polygons in front of it.
3. This starting value is propagated for all edges that start from this vertex.
4. As soon as an edge passes a contour line, the invisibility value is incremented or decremented.

5. The areas of an edge that have the invisibility value 0 are drawn.
6. If the second vertex of an edge is reached, the current value of the invisibility is set as the starting value for all further edges that start from this point (propagation of the quantitative invisibility by exploiting the edge coherence).

When propagating invisibility at vertices intersected by a contour line, a special case needs to be considered: edges emanating from such vertices may be obscured by one or more front-facing polygons that share that vertex. For example, in Fig. 7.2 the edge $\overline{KL}$ has the invisibility 1 because it is covered by the upper object polygon. However, the corner point K is set to 0 by the edge $\overline{KJ}$. Therefore each new edge must be tested against all front-facing polygons containing such a vertex (modification of the initial value).

7.2 Hidden Surface Algorithms

7.2.1 Z buffering

The simplest, but at the same time the most important hidden surface algorithm is *Z-Buffering*. In addition to the *frame buffer* (image memory) for recording the color, transparency and texture of each pixel, there is the so-called *z-buffer*, in which the depth values (distance from the observer) of each pixel are stored during scan conversion. All that is needed is additional depth in the screen memory, for example 16 or 24 additional bits per pixel.

The algorithm is then as follows:

1. Set the initial z value of all pixels to the maximum distance value R – in the case of the links system with increasing z coordinate for increasing distance, this corresponds to the rear clipping plane – and the color to background.
2. Scan-convert all polygons of the scene in any order. When scan converting a polygon pixel, if the current z value is less than the current value in the z-buffer, replace the values (color and z value) with those of the current pixel.

In this way, the color value of the polygon closest to the viewer is kept in the frame buffer. The complexity of the algorithm is thus only linear to the number of polygons.

The method performs a depth sorting of the pixel values. Only one comparison is required per decision. The resolution of the depth values and thus the quality of the algorithm depends on the memory depth of the Z buffer.

Figure 7.4 shows an example of how the Z-buffer algorithm works. Color values of the polygons are represented by gray values, while numbers represent the corresponding depth values. In step (a) a polygon with constant z value is inserted into an empty frame and Z buffer. In step (b) another polygon is then drawn, which penetrates the first one.

The calculation of a z value can be simplified using depth coherence. Usually, to find the z

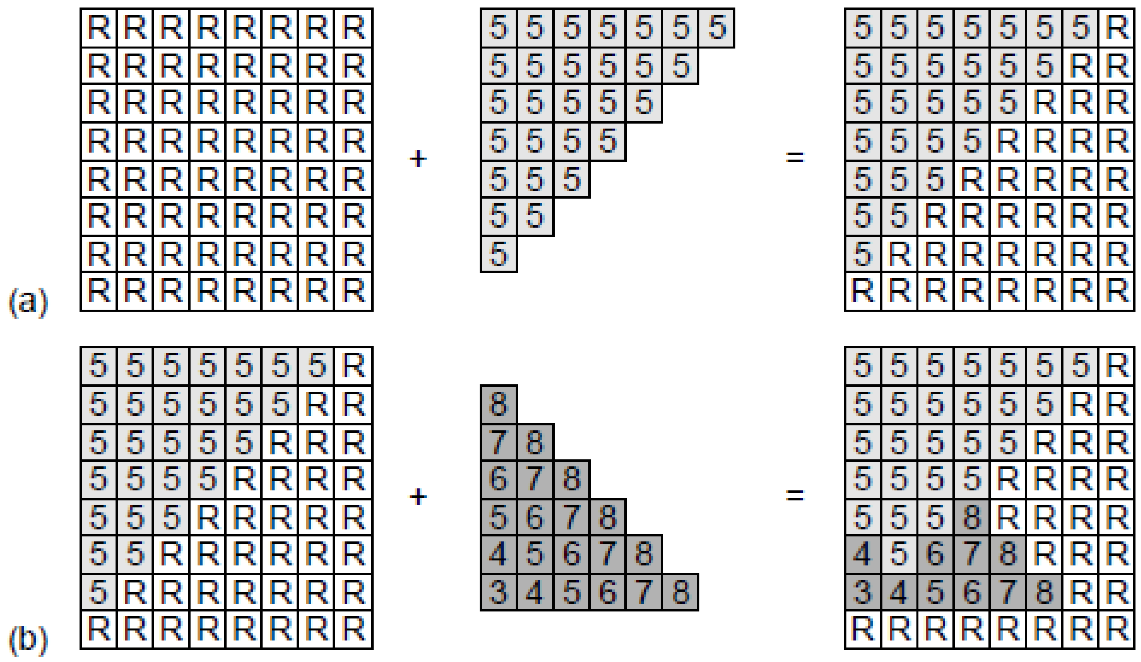


Figure 7.4: *Example of how the Z-buffer works*

value of the polygon at a location (x, y), the plane equation $Ax + By + Cz + D = 0$ has to be solved for z.

$$z = \frac{-D - Ax - By}{C} \tag{7.1}$$

Calculating the z value of each point on a scanline can now be simplified and done by increments, given that a polygon is planar. Suppose at the point (x, y) evaluate (7.1) to z_1. As a result, one now finds $(x + \Delta x, y)$ for z at the position

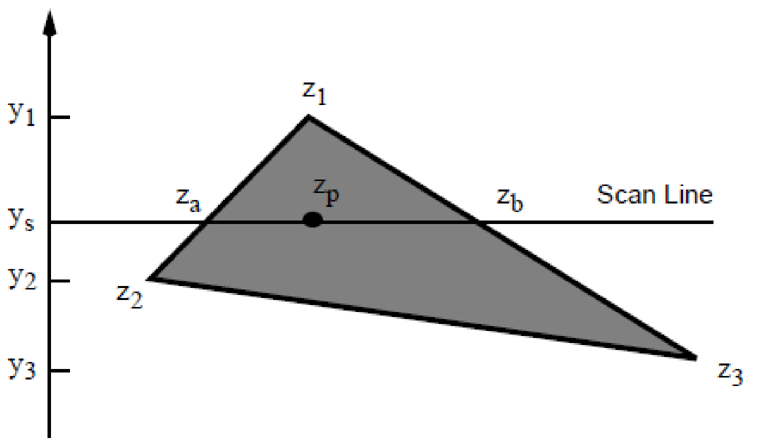
$$z = z_1 - \frac{A}{C}\Delta x \tag{7.2}$$

To calculate $(x+1, y)$ given $z(x, y)$ only one subtraction is necessary because the quotient A/C is constant and $\Delta x = 1$. Similarly, for the transition to the next scan line, the first z value is obtained by subtracting B/C for each Δy.

If the polygon is not planar or not defined, Gouraud interpolation (see Chapter 8) can also be used.

Shading only needs to be calculated for visible polygons.

Z-Buffering is the most important hidden surface algorithm. It is often implemented in hardware. The limited depth resolution (image precision) can lead to aliasing effects. Furthermore, clipping on the z front and back plane can also be handled with the z buffer.



$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_S}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_S}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Figure 7.5: *Interpolation of z values by incrementing*

7.2.2 Depth sorting (Newell-Newell-Sancha)

Depending on the complexity of the scene, a priority list can also suffice to draw the polygons in the correct order. If polygons only partially cover each other in the z direction, i.e. do not overlap each other, it is sufficient to apply a simple z sorting and then draw in this order (Fig. 7.6 image (a)).

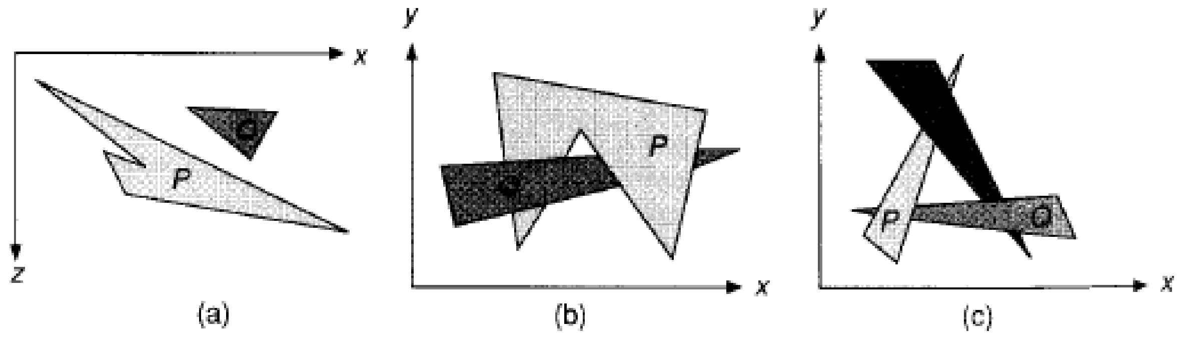


Figure 7.6: *Different situations for the spatial arrangement of polygons*

If the polygons interpenetrate or overlap cyclically, no clear sequence can be identified. (Fig. 7.6 (b) and (c)). In this case the polygon must be split.

Conceptually, the following three steps must be carried out:

- Sort all polygons by their smallest z coordinate.
- For each polygon: Solve all ambiguities arising from overlaps in z-direction.
- Scan-convert all polygons according to the updated list (i.e. back-to-front).

If the polygons are in $z = const.$ planes, there can be no overlap and the 2nd step can be ignored. This leads to a simplified form of depth sorting, the so-called *Painter's Algorithm*, which is mainly used in $2\frac{1}{2}D$ situations such as window managers.

But how can ambiguities be resolved? After pre-sorting the polygons, polygon P is at the end of the list and thus furthest from the viewer. Assuming a legal system, this polygon P has the smallest z_{min} coordinate. Before P can be drawn, all other polygons Q in the list must be tested to see if they are covered by P. If this is not the case, P can be drawn.

The following six tests of increasing complexity are therefore carried out for each polygon Q in the list. Once a test condition is met, Q is uncovered by P and the next polygon Q is tested. If all polygons Q pass the tests, P is drawn and the next polygon Q in the list becomes the new P.

For all polygons Q in the list:

1. Do the z extents of P and $Q(P_{z_{max}} > Q_{z_{min}}$ overlap? If not, **done**.
2. Do the extents in the x direction overlap? If not, **done**.
3. Do the extents in the y direction overlap? If not, **done**.
4. Is P entirely on the opposite side of the plane through Q from the viewer? If so, **done**.

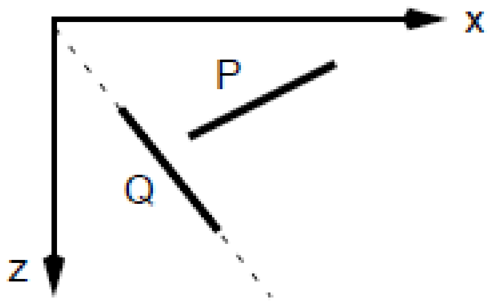


Figure 7.7: *Test 4 is fulfilled*

5. Is Q completely on the same side as the viewer with respect to the level through P ? If so, **done**.

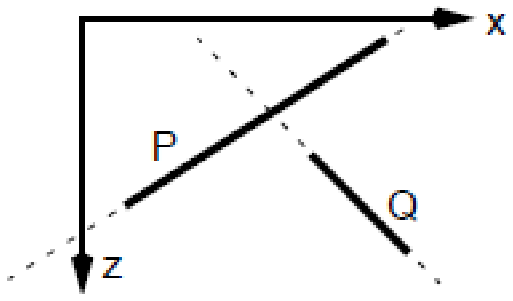


Figure 7.8: *Test 4 is not fulfilled, test 5 is fulfilled*

6. Do the projections of the polygons overlap in the xy plane (testing all edges against all)? If not, **done**.

If all tests fail, then Q is obscured by P and it must be checked whether Q can be scan-converted before P. To do this, swap P and Q and repeat tests 4 and 5:

4'. Is Q completely on the other side than the viewer with respect to the level through P ? If so, **done**.

4'. Is P completely on the same side as the viewer with respect to the level through Q ? If so, **done**.

If again an attempt is made to swap P and Q, then there is a cyclic overlap. If this is the case, follow steps 7 and 8:

7. Split P into two partial polygons at the plane through Q, delete P from the list and include the new polygons (can be done using Sutherland-Hodgeman clipping, for example).

7. Repeat the tests with the new list.

The procedure for tests 4 and 5 is explained using the example in Fig. 7.9. The point is to decide whether polygons Q_1 and Q_2 lie entirely on one side of the area defined by polygon P. The three orthographic projections in Figures (a) - (c) are not able to provide any information about this.

The polygon vertices are with

$$\begin{array}{cccc} P: & (1,1,1) & (4,5,2) & (5,2,5) \\ Q_1: & (2,2,0.5) & (3,3,1.75) & (6,1,0.5) \\ Q_2: & (0.5,2,5.5) & (2,5,3) & (4,4,5) \end{array}$$

given.

For the plane equation of the plane defined by the polygon P, one finds

$$15x - 8y - 13z + 6 = 0$$

The test function (implicit form) is then

$$v = 15x - 8y - 13z + 6$$

Inserting the 3 corners of Q_1 results in:

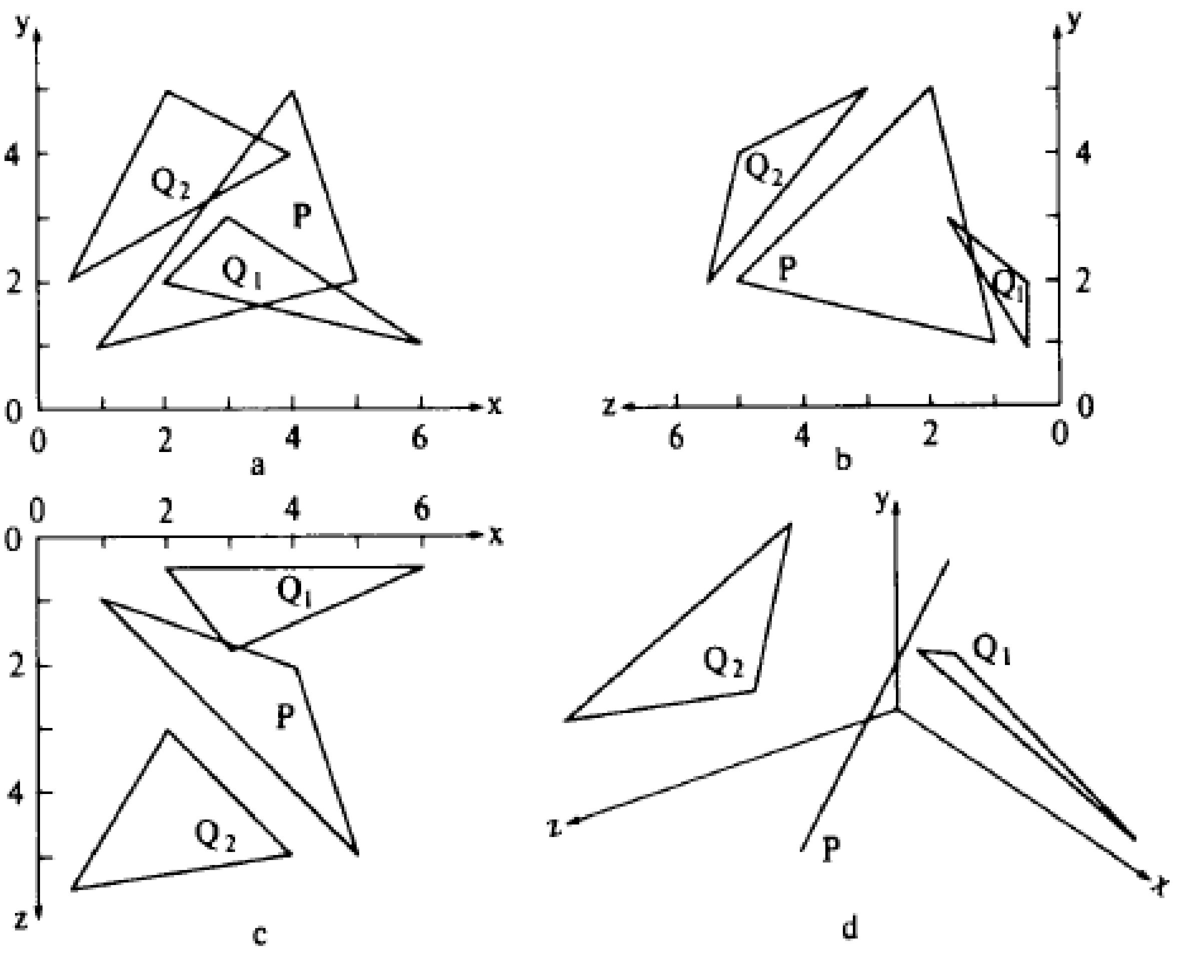


Figure 7.9: Example of deciding tests 4 and 5

$$v_1 = 15 \cdot 2 - 8 \cdot 2 - 13 \cdot 0.5 + 6 = 13.5 > 0$$
$$v_2 = 15 \times 3 - 8 \times 3 - 13 \times 1.75 + 6 = 4.25 > 0$$
$$v_2 = 15 \times 6 - 8 \times 1 - 13 \times 0.5 + 6 = 81.5 > 0$$

Since the signs are all positive, Q_1 lies entirely in one of the half-planes defined by P. For Q_2 one finds analogously:

$$v_4 = 15 \cdot 0.5 - 8 \cdot 2 - 13 \cdot 5.5 + 6 = -74 > 0$$
$$v_5 = 15 \cdot 2 - 8 \cdot 5 - 13 \cdot 3 + 6 = -43 > 0$$
$$v_6 = 15 \cdot 4 - 8 \cdot 4 - 13 \cdot 5 + 6 = -31 > 0$$

The consistently negative signs of the test function show that Q_2 is completely in the other half-plane. If the observer is far away on the positive z-axis in Fig. 7.9 (d), then Q_1 lies in the hemiplane facing away from the observer and is thus partially covered by P. It is also evident that Q_2 lies in the same half-plane as the observer and thus partially covers P.

7.2.3 BSP trees

In the sorting algorithm, the problem of visibility was previously solved dynamically and must therefore be recalculated for each change in the viewer's location, even in the case of a static scene. This is too expensive for real-time applications such as flight simulation. An elegant and efficient method that does not have this disadvantage is the *Binary Space Partitioning Tree*.

The idea of the BSP tree is to subdivide the scene into *polygon clusters* using dividing planes. If the observer is in the same half-plane as a cluster, the cluster can occlude others, but cannot occlude itself.

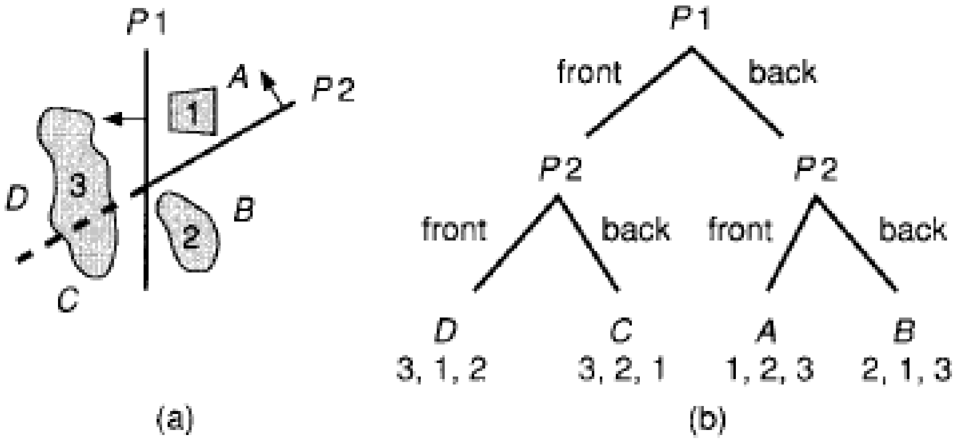


Figure 7.10: *Clustering of a polygon scene and associated BSP tree*

Each cluster is subdivided as long as dividing planes can be found. It is represented by a binary tree whose nodes contain the dividing planes and whose leaves represent the resulting regions in space.

The viewer's point of view is determined by a top-down traversal of the tree and comparisons with the plane equations in the respective nodes.

This scheme can be extended to the polygon level. First, a *root polygon* is selected whose plane splits the remaining polygons into two half-spaces. Polygons that intersect the parting plane are further split. New root polygons are selected in each of the two resulting half-spaces. These should cause as few *splittings* as possible. This is done recursively until there is only one polygon per sheet.

Fig. 7.11 (a)-(d) shows an example of building a BSP tree. Figure (a) shows a top view of the scene before the recursion is complete, with polygon 3 as the root polygon. Fig. (b) shows the BSP tree after recursion for the left subtree while Fig. (c) shows the complete tree. Finally, figure (d) shows an alternative tree with polygon 5 as the root polygon.

The tree is traversed to generate the correct order for scan conversion. With regard to a root polygon (node) in the tree, the following order applies:

1. Scanconvert the polygons that are in the opposite half-space from the viewer (because they may be covered by the current root polygon).
2. scanconvert the root polygon.
3. Scanconvert all polygons that are in the same half-space as the viewer.

Since the scheme can be applied recursively to each node of the tree, the polygon list sought is obtained by successively applying the above rule to all nodes in the tree.

Figure 7.12 shows an example of the scan conversion order of the same scene for different observer locations. The projection lines are represented by thin lines, while white numbers in black circles indicate the order of drawing the corresponding polygons.

The following code illustrates the construction and traversal of the tree:

```
typedef struct BSP_tree {
        polygon root;
        struct BSP_tree *backChild;
        struct BSP_tree *frontChild;
} BSP_Tree;

BSP_Tree* BSP_makeTree(listOfPolygons polyList); {

        polygon root;
        listOfPolygons backList, frontList;
        polygon p, backPart, frontPart;

        /* we assume each polygon is convex */
        if ("polyList␣is␣empty")
                return NIL;
        else {
                root = BSP_selectAndRemovePoly(polyList);
                backList = NIL; frontList = NIL;
                for ("each␣remaining␣polygon␣p␣in␣polyList") {
                        if ("polygon␣p␣in␣front␣of␣root")
                                BSP_addToList(p, frontList);
                        else
                                if ("polygon␣p␣in␣back␣of␣root")
                                        BSP_addToList(p, backList);
                else {
                    /* polygon p must be split */
                    BSP_splitPoly(p, root, &frontPart, &backPart);
                    BSP_addToList(frontPart, frontList);
                    BSP_addToList(backPart, backList);
                }
        }
        return BSP_combineTree(BSP_makeTree(frontList),
                                        root, BSP_makeTree(backList));
    }
}
BSP_Tree* BSP_displayTree(BSP_Tree *tree) {

    if ("tree␣is␣not␣empty")
        if ("viewer␣is␣in␣front␣of␣root") {
            BSP_displayTree(tree−>backChild);
            displayPolygon(tree−>root);
            BSP_displayTree(tree−>frontChild);
        }
        else {
            BSP_displayTree(tree−>frontChild);
            displayPolygon(tree−>root);
            BSP_displayTree(tree−>backChild);
        }
}
```

The tree only needs to be generated once for static scenes. The number of splitting operations, which depends on the choice of the root polygon, is decisive for the computational effort.

7.2.4 Warnock's Algorithm

The divide and conquer strategy of the BSP tree was carried out in object space. Similar approaches can also be formulated for the image plane. The image plane is broken down recursively into a quadtree, whereby the following 4 cases can occur in a cell created by the breakdown:

1. Polygon contains cell
2. Polygon intersects cell
3. cell contains polygon
4. Polygon outside the cell

Case 4 does not affect the cell. In case 2, the polygon can be split into sub-polygons, which can be treated using case 3 and case 4. In the following cases, the decision for a cell can be made directly, so that no further subdivision is necessary:

1. The cell does not contain a polygon $\rightarrow$ Background color
2. Just an intersecting or contained polygon $\rightarrow$ First fill cell with background, then scan-convert polygon
3. Just a bounding polygon (case 1) $\rightarrow$ Fill cell with polygon color
4. There are multiple polygons affecting the cell, but there is a bounding polygon that is closer to the viewer than all other $\rightarrow$ fill cells with the color of that bounding polygon

Cases 1-3 are easy to understand. Fig. 7.13 illustrates the fourth case. The test of whether a closer, bounding polygon exists is done by determining the z coordinates of the cell's vertices projected onto the polygon plane. If one finds a surrounding polygon whose four z coordinates determined in this way are larger (legal system) than the z coordinates of all other surrounding, intersecting or contained polygons, then the criterion of case 4 is fulfilled (Fig. 7.13 (a) fulfilled, in contrast to Fig. 7.13 (b)).

If the cell cannot be assigned to any of the four cases, it is subdivided into four cells for which the same tests are run again. The recursion is aborted as soon as the size of the cells corresponds to the pixel resolution. In such a case, the visibility is determined by z-buffering.

Fig. 7.14 shows the subdivision in the Warnock algorithm using a simple scene as an example. The number in the cells corresponds to the number of one of the cases 1 - 4. For cells without numbers, none of the cases apply.

As an alternative to subdivision into square cells, it can also be divided into rectangular areas. It is divided around a polygon corner point in order to avoid unnecessary subdivisions in the cell. Fig. 7.15 shows this case for the same scene as in Fig. 7.14. First it is divided around the corner point A, in the next step around the corner point B.

To handle case 2, polygon clipping algorithms are required for the rectangular cells. Preprocessing by backface culling and 3D clipping is also essential.

7.2.5 Weiler-Atherton Algorithm

Another variant is to carry out the subdivision along polygon boundaries, although a powerful clipping algorithm for concave polygons must be available. The algorithm first performs a Z sorting of the polygons. The polygon closest to the viewer is selected as the clipping polygon and all others are clipped to it. This results in a splitting of the polygons into an *inside* and an *outside part polygon*.

Two lists are built, an *Inside* list and an *Outside* list. The partial polygons in the inside list that are completely behind the clipping polygon can be deleted. The other polygons in the *Inside* list are processed recursively in the same way: The partial polygon in the *Inside* list that is closest to the viewer becomes the new clipping polygon to which all other polygons in the list are clipped. When the recursion is complete, the current *Inside* list is scan-converted and then the *Outside* list is treated in the same way.

The original polygon is always clipped because it is less expensive than clipping against one or more parts of it. It is therefore necessary to have references from the subpolygons back to the original.

The clipping polygons are managed within the recursion using a stack. The stack always contains polygons that are needed as clipping polygons but are not the current clipping polygon due to recursive subdivision. Cyclic overlaps can be detected using the stack: If a partial polygon is found that is in front of the current clipping polygon, the stack is searched for. If it is present, no further recursion is necessary because all polygon parts that lie inside and behind the clipping polygon have already been eliminated.

Fig. 7.16 illustrates the Weiler-Atherton algorithm. The current clipping polygon is identified by a bold outline. Polygons that have already been scan converted appear gray. The numbers in Fig. (a) denote the z-coordinates of the respective vertices.

The flow of the algorithm for this example is as follows:

1. First, A is selected as the clipping polygon: The resulting *Inside* list is $\{B_{in}A, D_{in}A, C_{in}A, A\}$, the outside list is $\{B_{out}A, D_{out}A, C_{out}A\}$.
2. $B_{in}A$ and $D_{in}A$ can be deleted from the list because $A$is closer to the viewer.
3. $C_{in}A$ is closer, so subdivide recursively and clip the remaining inside list against C. An *inside* list of $\{A_{in}C, C_{in}A\}$ and an *outside* list of $\{A_{out}C\}$ are created.
4. $A_{in}C$ is after C, so it will be deleted. It remains $C_{in}A$ to be drawn.
5. Before returning from the recursion, the element of the outside list $A_{out}C$ can be drawn, since it corresponds to the remaining inside element of the next higher level.
6. After the return, the *Outside* list is processed. B now becomes a clipping polygon and creates an *inside* list of $\{B_{out}A, C_{out}A_{in}B\}$ and an *outside* list of $\{C_{out}A_{out}B, D_{out}A\}$.
7. $C_{out}A_{in}B$ is deleted because it is behind B. So only $B_{out}A$ remains in the *Inside* list and is drawn.
8. After returning again, the remaining *Outside*-list is processed and $C_{out}A_{out}B$ and $D_{out}A$ are drawn.

The algorithm can be used in the same form as the Warnock algorithm.

The following pseudocode illustrates the recursion:

```
WA_visibleSurface(void)
{
        listOfPolygons *tempList=0;
        polyList = list of copies of all polygons;
        /∗ sort polyList by descending z−max of vertices ∗/
        sortPolyList(polyList);

        /∗ Delete stack ∗/
        deleteStack();

        /∗ process each remaining polygon region ∗/
        while (polyList)
                WA_subdivide(polyList−>firstElement, &polyList);
}

WA_subdivide(polygon clipPolygon, listOfPolygons ∗∗polyList) {

        listOfPolygons ∗inList=0;
        listOfPolygons ∗outList=0;
        listOfPolygons ∗tempList=0;
        polygon ∗poly;

        for (tempList=∗polyList; tempList; tempList=tempList−>next)
                /∗ Clip the polygon tempList to clipPolygon
                Hang pieces inside in the inList,
                Subpolygons outside in the outList. ∗/
                clipPolygonAndAppendToList(tempList, clipPolygon,
                        &inList, &outList);

        /∗ delete subpolygons behind clipPolygon
        insideList ∗/
        deletePiecesBehindClipPolygon(clipPolygon, &inList);

        /∗ recursively process the misordered ones
        fragments ∗/
        for (all polygons poly from inList that are not on the stack
                lie and are not part of the clipPolygon) {
                pushPolygonOnStack(clipPolygon);
                WA_subdivide(poly, &inList);
                popStack();
        }

        /∗ Display the rest of the polygons inside the clipPolygon ∗/
        for (all polygons poly in inList)
                displayPolygon(poly);

        ∗polyList = outList;
}
```

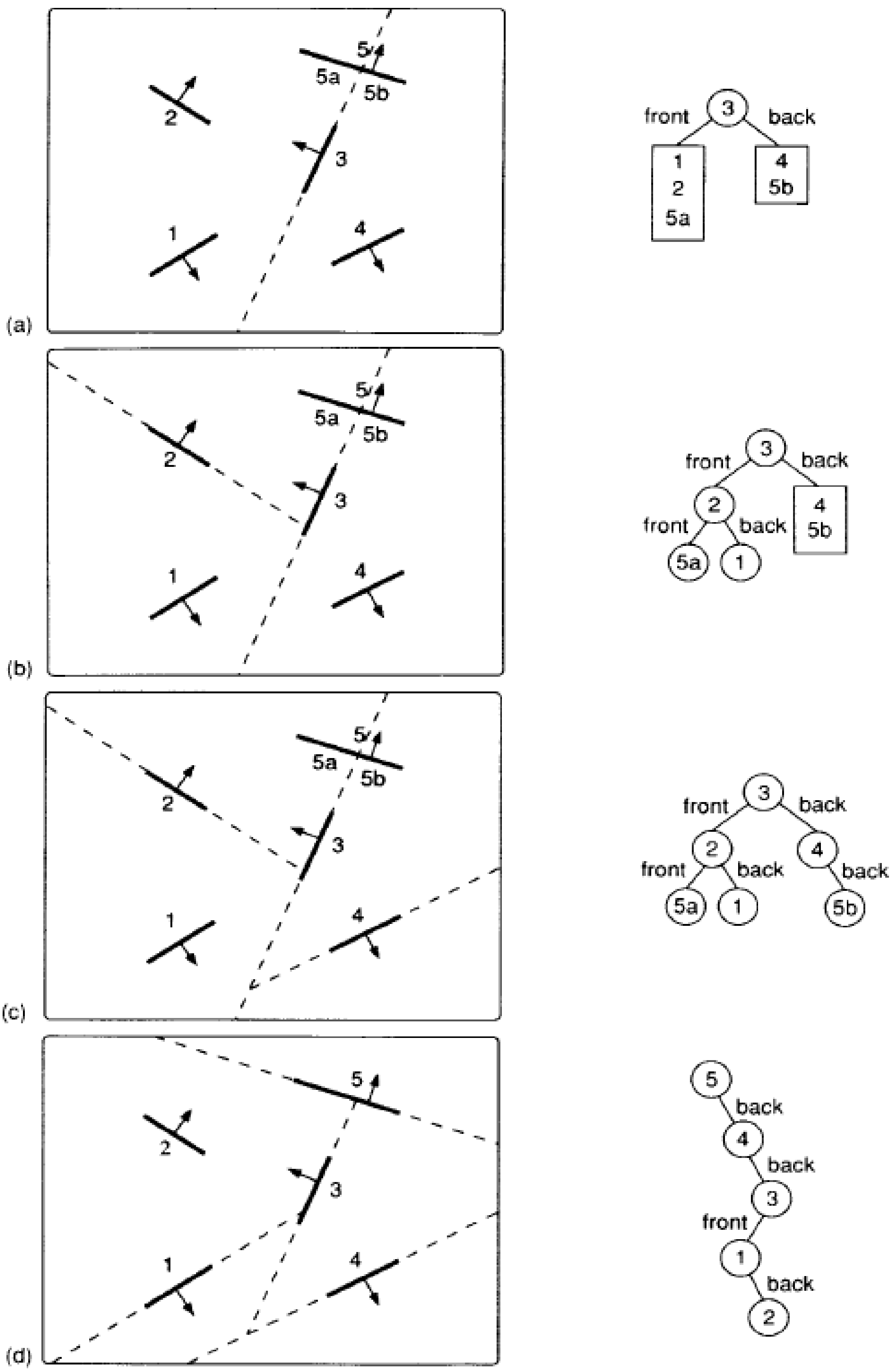


Figure 7.11: *Examples for building BSP trees*

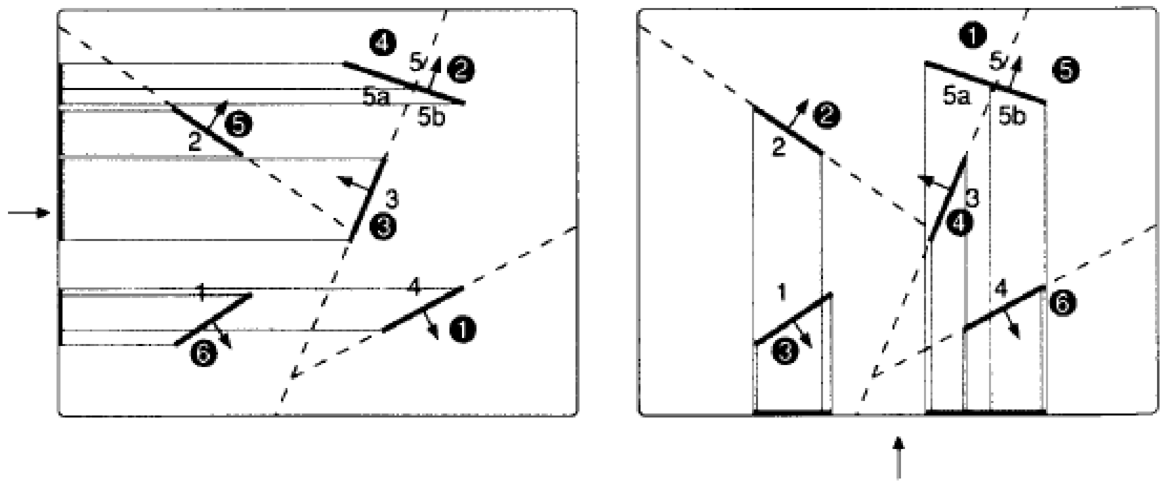


Figure 7.12: *BSP tree traversal and resulting scan conversion order for two different projections of the same scene*

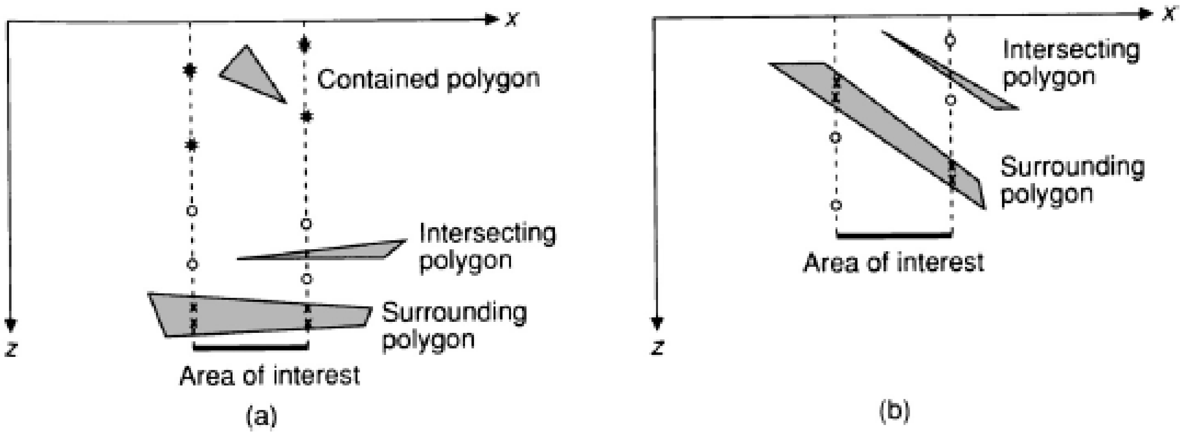


Figure 7.13: *Illustration of case 4 for deciding whether a further subdivision is necessary for quadtree methods*

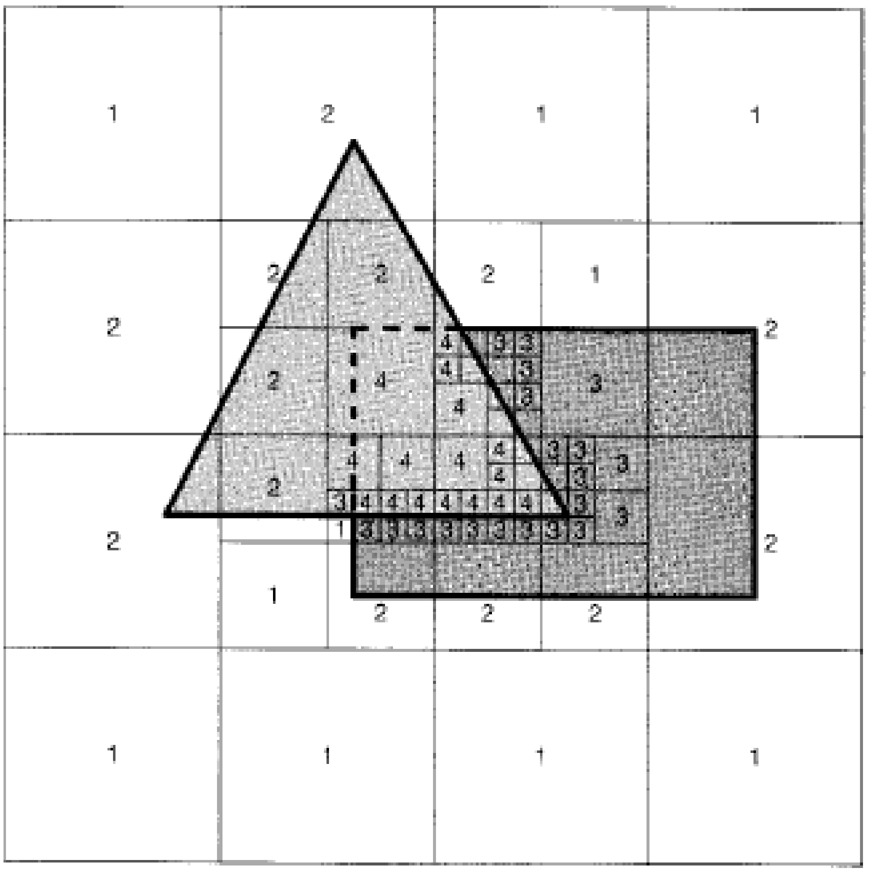


Figure 7.14: *Subdivision into square cells*

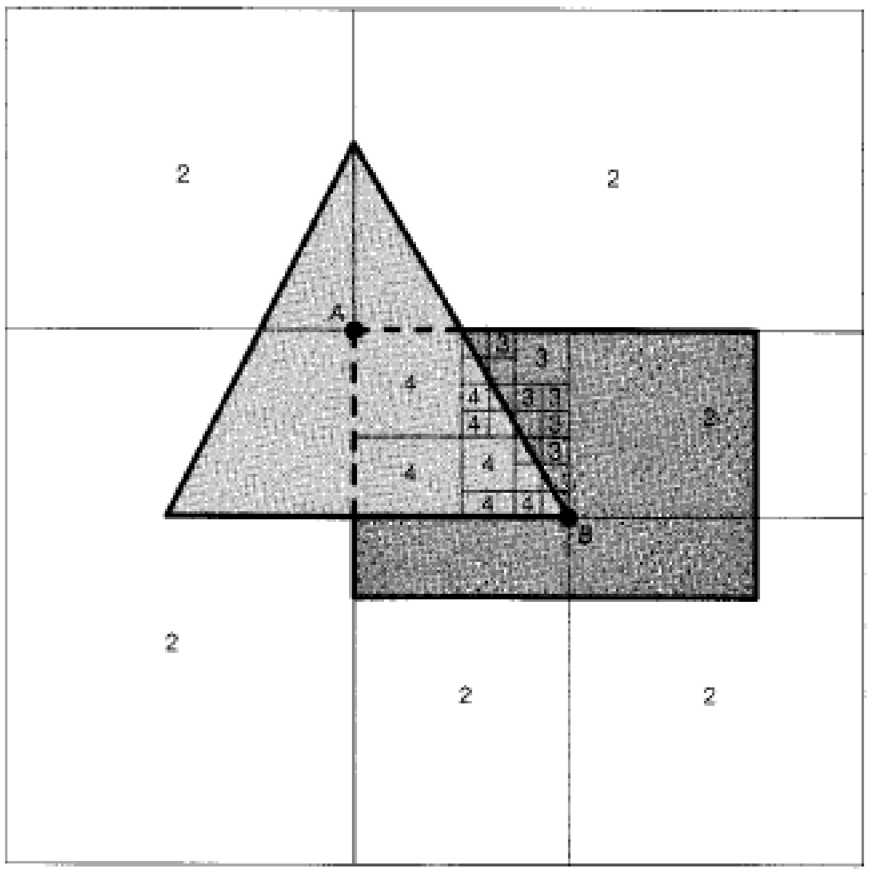


Figure 7.15: *Subdivision around the polygon vertices marked with circles*

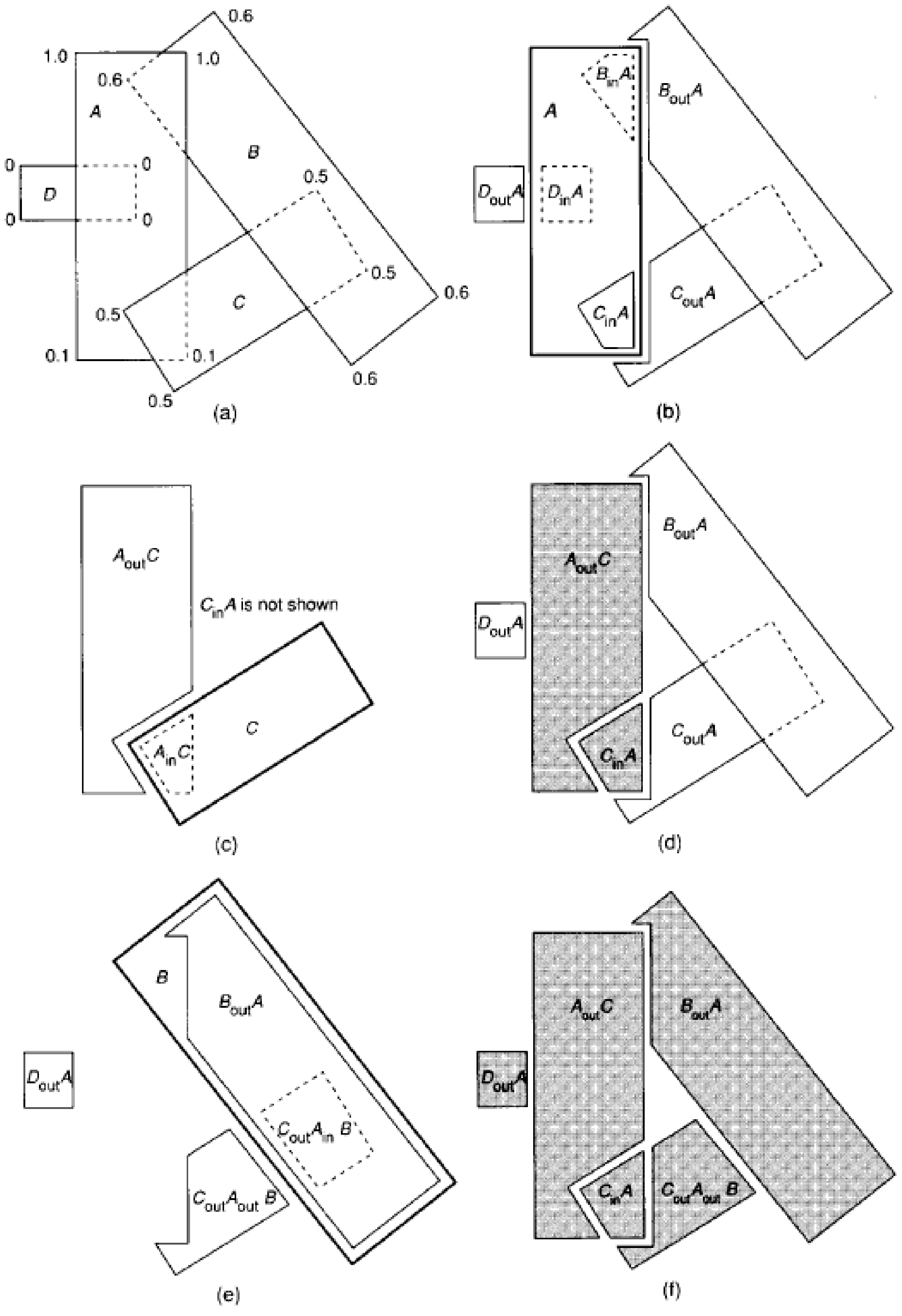


Figure 7.16: *Examples for the Weiler-Atherton algorithm*
(a) original scene
(b) Polygons clipped to A
(c) inside list of A clipped to C
(d) visible parts within A
(e) Polygons clipped to B
(f) All visible parts of the scene

Lighting, Shading and Texturing

8.1 Lighting

8.1.1 Ambient Light

Ambient light is created by the self-emissions of an object (h_i) or by the global illumination of a large number of diffuse sources in the area (I_a). In the monochrome case:

$$I = h_i + I_a k_a \tag{8.1}$$

h_i	own emission coefficient
k_a	ambient reflection coefficient
I_a	intensity of the ambient light
I	resulting intensity

The self-emission coefficient h_i is no longer included explicitly in all subsequent formulations of the lighting model. The self-emission share is assumed to be included in the ambient reflection coefficient.

8.1.2 Diffuse Reflection

Assuming a point light source that illuminates the object, *Lambert's law of reflection* is valid for matte surfaces:

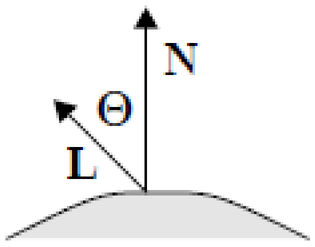


Figure 8.1: *Diffuse reflection*

Lambertian surfaces appear equally bright from any viewing angle because they reflect light with equal intensity in all directions. This means that the intensity only depends on the geometric relationships between the light source and the surface, but not on the observer's location. The following applies:

$$I = I_p k_d \cos\theta \tag{8.2}$$

I_p	light source intensity
k_d	diffuse reflection coefficient
θ	Angle between surface normal N and light source vector L with $\theta \in [0, \pi/2]$

If the vectors are normalized, the following notation can be used with the help of the scalar product:

$$I = I_p k_d (\mathbf{N} \cdot \mathbf{L}) \tag{8.3}$$

If the point of light is at infinity, the angle θ is constant for surfaces of the same normal, and a directed light source is obtained.

Combining the ambient (8.1) and the diffuse model (8.3) results in:

$$I = I_a k_a + I_p k_d (\mathbf{N} \cdot \mathbf{L}) \tag{8.4}$$

8.1.3 Attenuation

Due to the solid angle radiation of a light source, the luminous flux that arrives on a surface element dA is inversely proportional to the square of its distance from the light source: $1/d_L^2$.

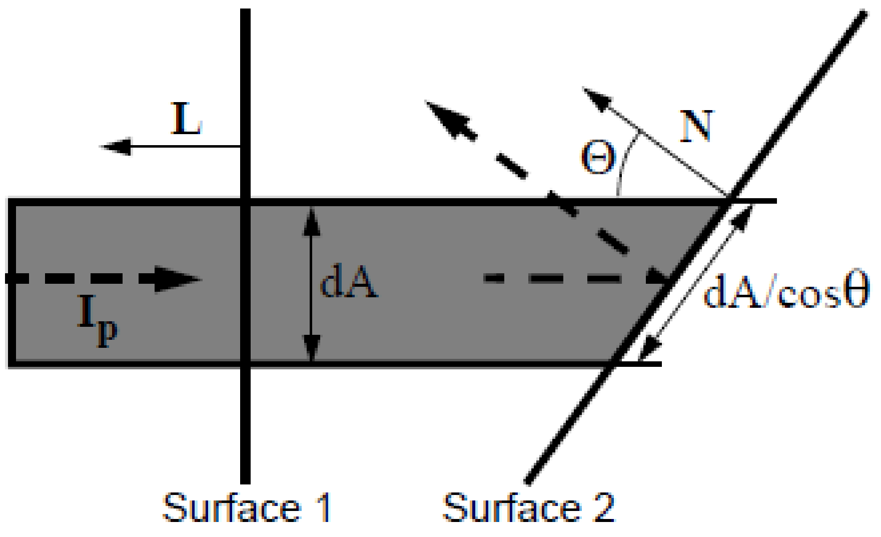


Figure 8.2: *Diffuse reflection: Ratio between incident and reflected light intensity depending on the size of the radiating comparison to the reflecting surface* $dA/(dA/\cos\theta)$

Therefore one can introduce an *attenuation factor* f_{att}.

$$f_{\text{att}} = \frac{1}{d_L^2} \tag{8.5}$$

Due to the rapid drop from $1/d_L^2$ for $d_L \to \infty$, f_{att} also often becomes

$$f_{\text{att}} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right) \tag{8.6}$$

where c_1, c_2 and c_3 are user-defined constants that describe the properties of a light source.

This expands the lighting model by the factor f_{att} to:

$$I = I_a k_a + f_{att} I_p k_d (\mathbf{N} \cdot \mathbf{L}) \tag{8.7}$$

8.1.4 Colors

So far, only monochrome relationships have been considered. It makes sense to consider equation (8.7) as a function of the wavelength λ.

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{\text{att}} I_{p\lambda} k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}) \tag{8.8}$$

Here, $O_{d\lambda}$ denotes the value of the spectrum of the object color at the point λ. Usually, the lighting equation is only solved at a few spectral points (e.g. for R, G, B).

`This restriction can lead to aliasing effects (see following chapter).`

8.1.5 Depth Cueing

The admixture of atmospheric white that occurs with increasing distance from the observer to the object can be modeled using a linearized approach. You get I_λ from the previous intensity by adding the atmosphere color $I_{dc\lambda}$ (depth cue):

$$I_\lambda = s_0 I_\lambda + (1 - s_0) I_{dc\lambda} \tag{8.9}$$

where s_0 is a scaling factor for interpolation between the original and depth cueing color, which is calculated from the z-coordinate of object z_0 as follows:

$$s_0 = s_b + \frac{(z_0 - z_b)(s_f - s_b)}{z_f - t_b} \qquad \text{for} \qquad z_b \leq z_0 \leq z_f \tag{8.10}$$

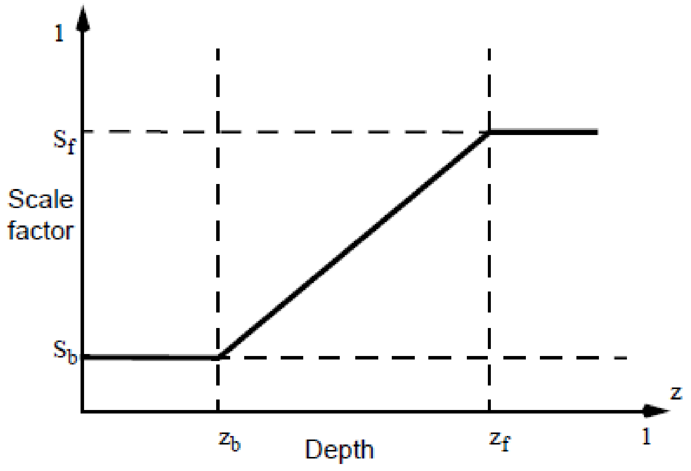


Figure 8.3: *Sigmoids for calculating the attenuation*

Fig. 8.3 illustrates the meaning of s_f and s_b (minimum and maximum addition of the atmosphere color) as well as z_f and z_b (linear interpolation interval).

In the further development of the illumination model, the attenuation terms are not included for the sake of clarity.

8.1.6 Specular Reflection (Directional Reflection)

In the ideal case of the reflective surface, the angle of reflection is also given by the law of reflection with the angle of incidence θ.

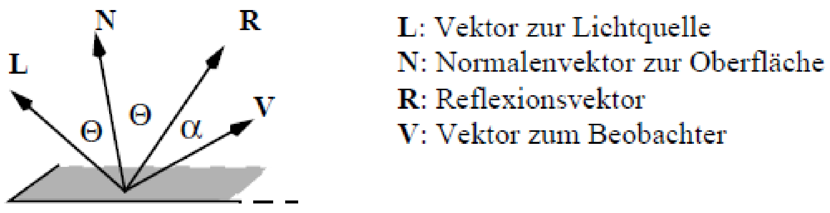


Figure 8.4: *Vectors and angles in specular reflection: vector to light source* L, *surface normal* N, *reflection vector* R, *vector to viewer* V

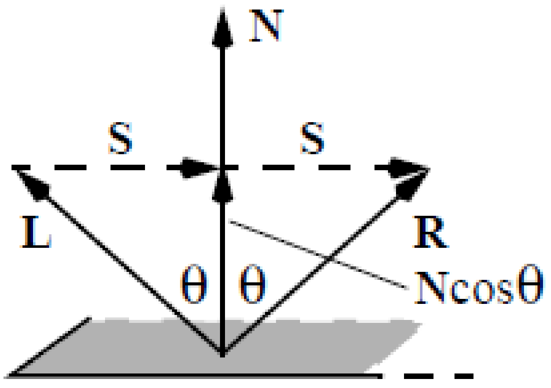


Figure 8.5: *Calculation of the reflection vector*

R is calculated by reflecting **L** on **N** (Fig. 8.5).

With normalized **N** and **L** the following applies:

$$\mathbf{R} = \mathbf{N} \cdot cos\theta + \mathbf{S} \tag{8.11}$$

With $|\mathbf{S}| = sin\theta$ and $\mathbf{S} = \mathbf{N} \cdot cos\theta - \mathbf{L}$ follows

$$\mathbf{R} = 2\mathbf{N} \cdot cos\theta - \mathbf{L} = 2\mathbf{N}(\mathbf{N} \cdot \mathbf{L}) - \mathbf{L} \tag{8.12}$$

The angle α between the reflection and observer vector results from

$$cos\alpha = \mathbf{R} \cdot \mathbf{V} = (2\mathbf{N}(\mathbf{N} \cdot \mathbf{L}) - \mathbf{L}) \cdot \mathbf{V} \tag{8.13}$$

8.1.7 Phong's lighting model

Typically, the ideally reflecting case occurs only very rarely in reality, so that the lighting equation must have a reflecting part that depends on the angle α in addition to all previous components. It is assumed that the maximum specular reflection occurs for $\alpha = 0$ (Fig. 8.4) and then decreases at different rates for increasing angles. This behavior can be modeled using the $cos^n(\alpha)$ function. This gives *Phong's illumination model:*

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}cos\theta + W(\theta)(cos\ alpha)^n] \tag{8.14}$$

Here, $W(\theta)$ denotes the fraction of the specularly reflected light and n denotes a material constant, the *specular reflection exponent.* The exponent n decides the shape of the highlight on the surface. A value of 1 produces a broad, tapering highlight, while higher values produce sharp, focused highlights (Fig. 8.6). For the ideal reflector, n would be infinite.

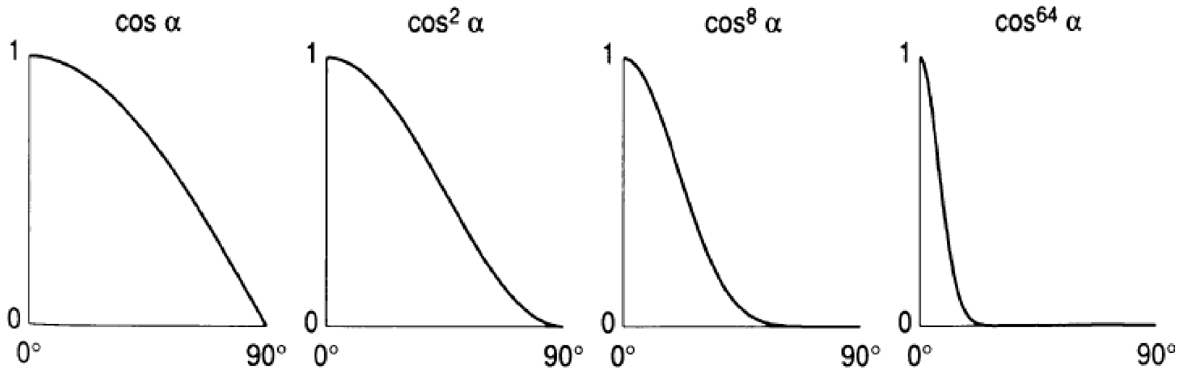


Figure 8.6: *Characteristics of the $cos^n\alpha$ function for different n*

A constant value between 0 and 1 is often assumed for $W(\theta)$, the so-called *specular reflection coefficient* k_s. With this, and replacing the cosine terms with the corresponding scalar products, we get

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}(N \cdot L) + k_s(\mathbf{R} \cdot \mathbf{V})^n] \tag{8.15}$$

The effects of the ratio of k_d/k_s on the surface propagation picture are shown for different n in the following picture.

In equation (8.15) the specular part is independent of material properties such as color. However, it turns out that the specular reflection is influenced by the surface properties. It generally has different specular reflection properties than diffuse reflection. Therefore, Phong's illumination model can be extended by the *specular color $O_{s\lambda}$* , and one gets:

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}(N \cdot L) + k_s(\mathbf{R} \cdot \mathbf{V})^n] \tag{8.16}$$

Phong's illumination model can also be formulated using the *halfway vector* **H** (Fig. 8.8.)

The halfway vector points midway between the direction of the light source and the viewing direction. If **H** and **N** point in the same direction, the viewer sees the brightest highlight because **R** and **V** point in the same direction. Using the halfway vector, the specular term can be expressed by

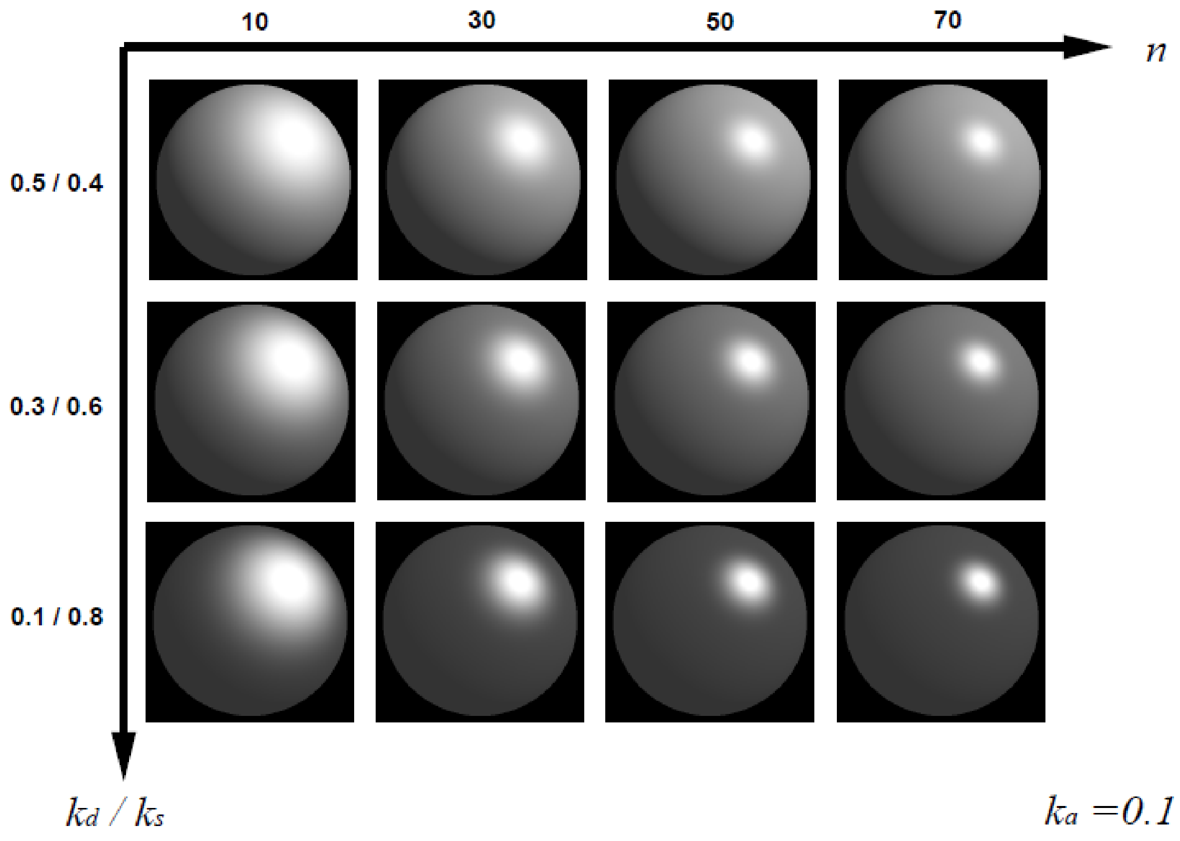


Figure 8.7: *Propagation map for different* n

$$(cos\beta)^n = (\mathbf{N} \cdot \mathbf{H})^n \tag{8.17}$$

With

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{8.18}$$

From Fig. 8.8 it can be seen that the angle β is not equal to the angle α. Therefore, the specular exponent n does not produce the same results as the original Phong model.

Multiple light sources with different characteristics and positions in the scene can pass through

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1<=i<=m} f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (N \cdot L_i) + O_{s\lambda} (\mathbf{R_i} \cdot \mathbf{V})^n] \tag{8.19}$$

be modeled. However, I_{lambda} must be checked for overflow.

8.1.8 Modeling light sources

In addition to the simple point light source, spotlights can also be approximated using simplified light intensity distributions. The *Warn illumination model* is used, in which a light source is modeled as a specular reflection distribution of a single point that is illuminated by a point light source (Fig. 8.9). A specular exponent p can now be defined for the reflector on which the point is located, which describes the opening angle of the spotlight.

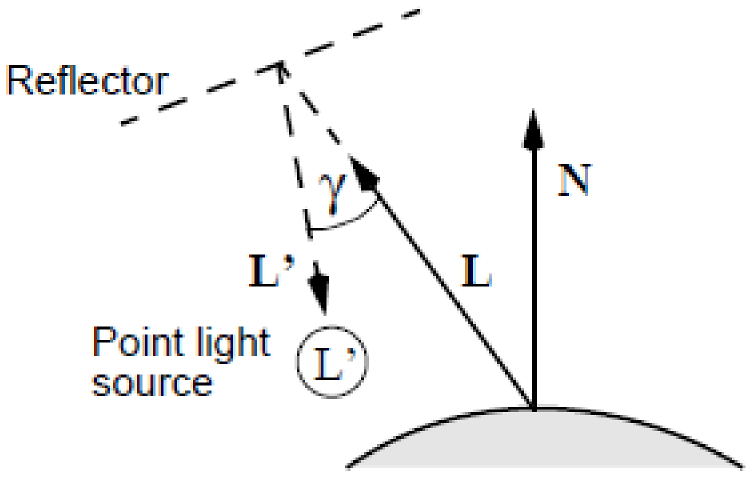


Figure 8.8: *Warn's lighting model and spotlight*

The light intensity at a point on the object thus increases

$$I_{L'\lambda}(cos\gamma)^n \tag{8.20}$$

where $I_{L'\lambda}$ denotes the intensity of the hypothetical light source. Replacing the cosine term with a scalar product gives

$$I_{L'\lambda}(-\mathbf{L}\cdot\mathbf{L}')^p \tag{8.21}$$

This expression can now be used for $I_{p\lambda}$ in (8.15) or (8.16).

Fig. 8.10 shows *goniometric diagrams* of the intensity distributions (luminous intensity distribution, cf. also chapter 2.2.2) of Warn's light sources with different specular exponents p compared with the intensity distribution of a point light source. The intensity is recorded as a function of the angle from the light source axis in polar coordinates. The arrow represents the direction vector L'.

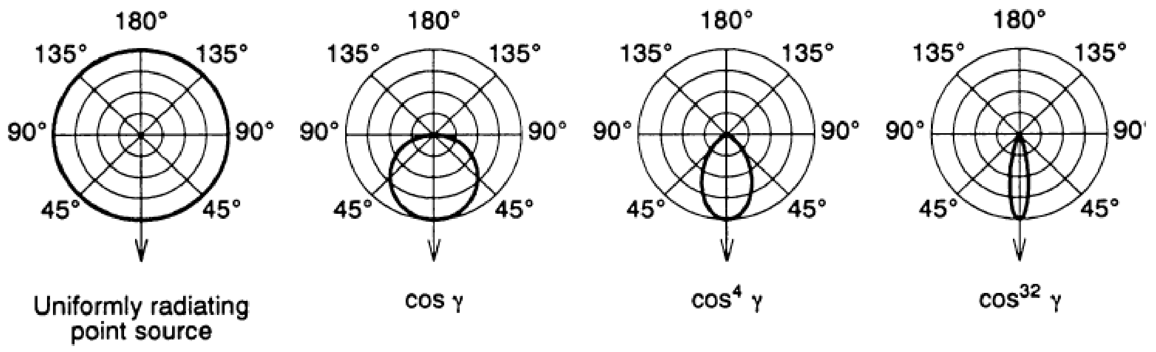


Figure 8.9: *Goniometric diagrams of Warn's light sources and a point light source*

8.2 Shading

Since the lighting model has to be evaluated at many different points in a scene for shading, the question of the frequency of this calculation arises, particularly in the case of polygonal representations of objects.

8.2.1 Constant shading

In the simplest case, the lighting equation only needs to be calculated once per polygon. This leads to a constant color value over the entire area. This type of shading is called *Flat Shading* or *Constant Shading*.

This is only ok if either the light source is at infinity or the polygon is very small.

8.2.2 Gouraud shading

For a convincing shading of polygonal bodies, the lighting model has to be evaluated at every point of the scan conversion, or one has to interpolate appropriately. With *Gouraud Shading* only corner point intensities are interpolated over the surface. This accounts for the shape and curvature of the approximated surface. To do this, the normals on all corner points of the polygons must be known. If the corner point normals are not given, they can be calculated from the surface normals (Fig. 8.11).

Gouraud shading then takes place in four steps:

1 Calculate the surface normals.

2 Calculate the normal to each vertex V as

$$N_v = \frac{\sum\limits_{i=1}^{N} N_i}{|\sum\limits_{i=1}^{N} N_i|}$$

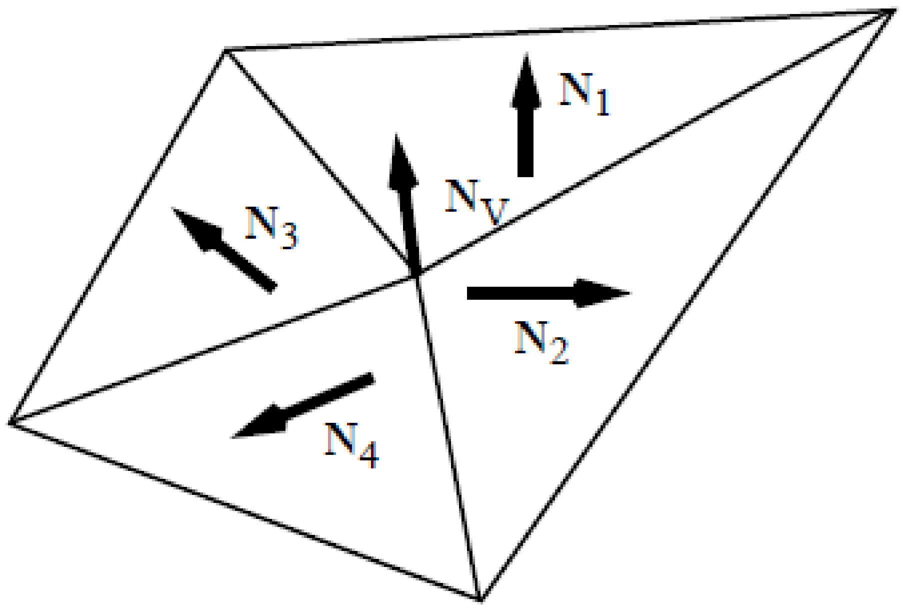
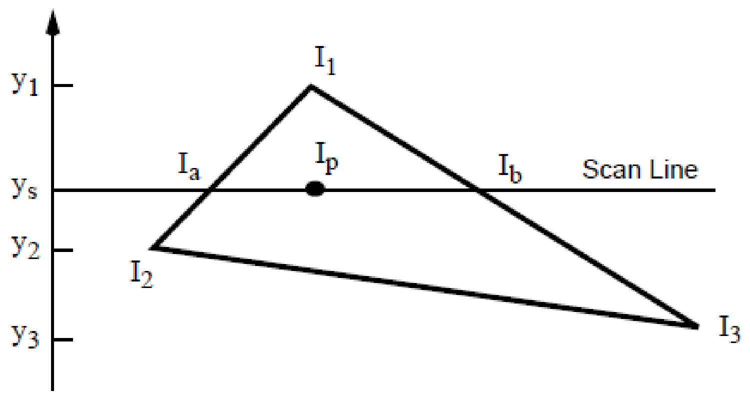


Figure 8.10: *vertex and surface normals*

3 Calculate the desired lighting model at each vertex.

4 Interpolate the intensity Ip for each discrete point **P** of the polygon according to Fig. 8.12 (bilinear interpolation).



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

$$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}$$

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

Figure 8.11: *Intensity interpolation along polygon edges and scan lines*

Gouraud shading is widespread and often implemented in hardware. Disadvantages of the process, however, are incorrect highlights, especially on the inside. The abrupt changes in intensity that occur with flat shading can also be weakened with gouraud shading, but not completely eliminated.

Note that the decomposition of a non-planar quadrilateral into two triangles is not unique. The corresponding bilinear interpolation surface is not planar.

For very large polygons relative to the pixel size, interpolation artifacts typically occur and specular effects cannot be represented. For small polygons relative to the pixel size, all the effects of the Phong model can be simulated.

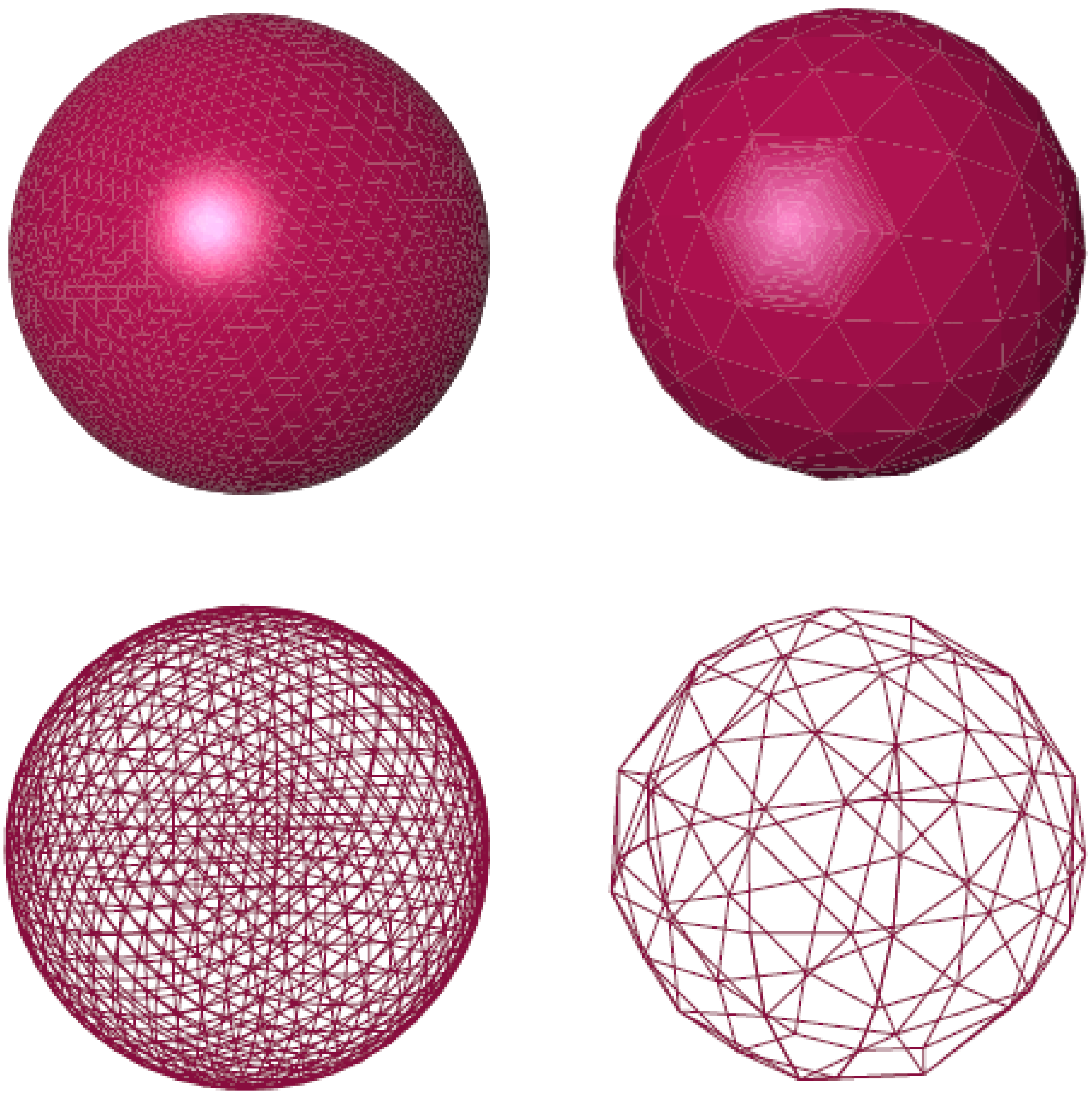


Figure 8.12: *Illuminated sphere approximated by different numbers of polygons as wireframe and as solid*

8.2.3 Phong shading

In contrast to the interpolation of intensities, the surface normals are interpolated with *Phong Shading*. For this purpose, the surface normal is interpolated from the corner points at each point P and the corresponding lighting model is thus evaluated. This gives a more accurate approximation of the surface. The interpolation is done on the span of a polygon on a scan line between the normal at the beginning and that at the end of the span. These normals are in turn interpolated between the corresponding vertex normals (Fig. 8.14).

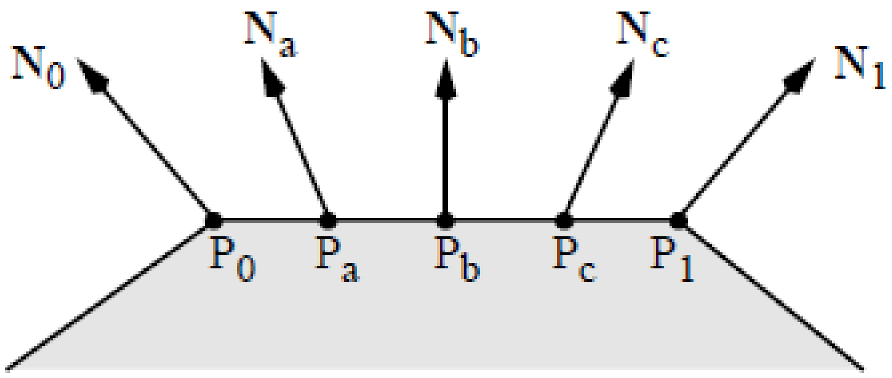


Figure 8.13: *Interpolation of the normal vector in Phong shading*

If the surface normals are not given explicitly, the corner point normals can be calculated by averaging the cross products of all adjacent edge vectors. For example, the normal at the corner point V_1 in Fig. 8.15 is calculated

$$\boldsymbol{N_{V_1}} = \frac{\overline{\boldsymbol{V_1V_2}} \times \overline{\boldsymbol{V_1V_4}} + \overline{\boldsymbol{V_1V_2}} + \overline{\boldsymbol{V_1V_4}} \times \overline{\boldsymbol{V_1V_5}}}{|\overline{\boldsymbol{V_1V_2}} \times \overline{\boldsymbol{V_1V_4}} + \overline{\boldsymbol{V_1V_2}} + \overline{\boldsymbol{V_1V_4}} \times \overline{\boldsymbol{V_1V_5}}|}$$

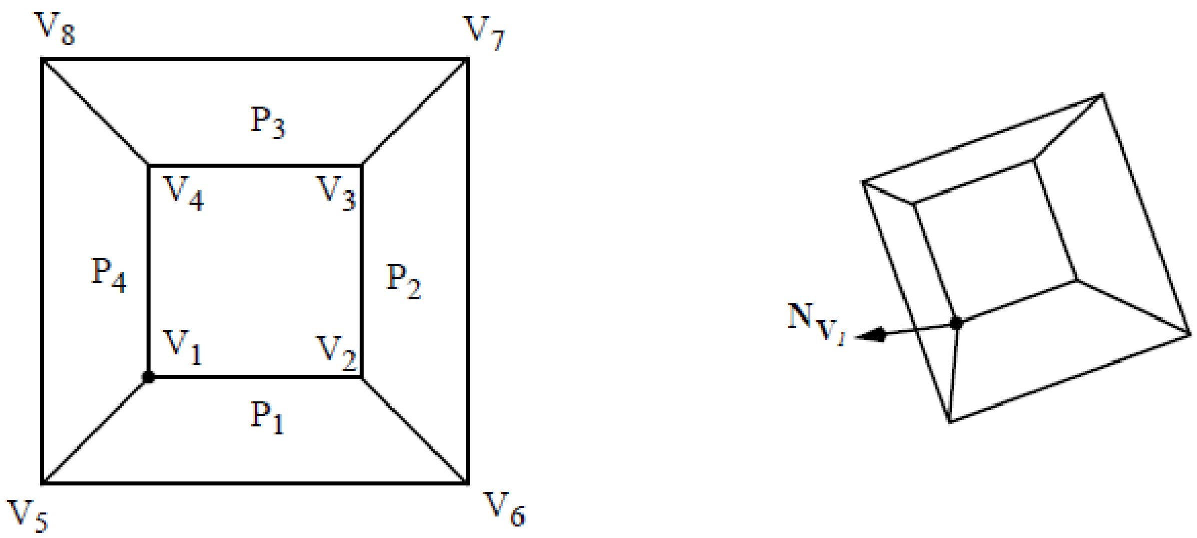


Figure 8.14: *Interpolation of the vertex normals*

Figure 8.16 shows the three shading methods *Flat Shading* (left), *Gouraud Shading* (middle) and *Phong Shading* (right) in direct comparison. The differences between the first two methods are obvious, while the differences between Gouraud and Phong shading are particularly recognizable in the case of highlights (shining points) - for example on the spout.

`Consider which shading operations can be performed in image space and which in object space.`
`What implications does this have for 3D graphics accelerators?`

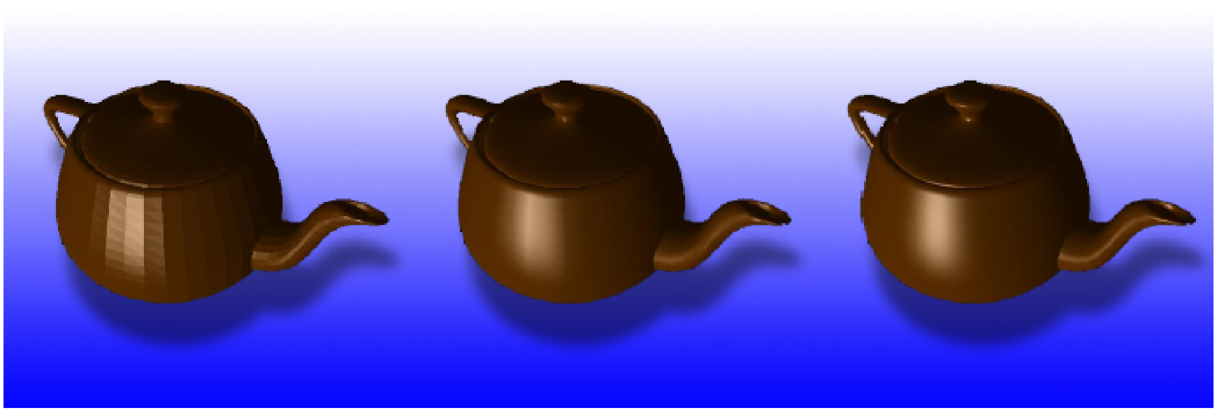


Figure 8.15: *Flat, Gouraud and Phong Shading in comparison*

8.3 Transparency and refraction

When passing into an optically thinner or thicker medium, light rays are refracted according to the *Huygens-Fresnel principle*.

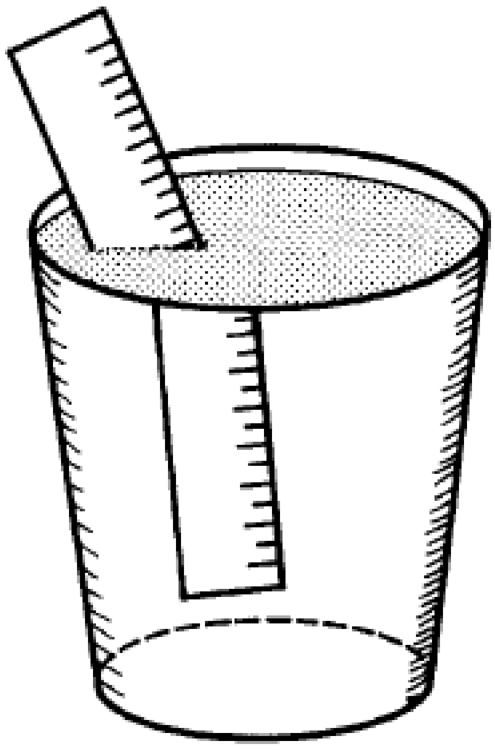


Figure 8.16: *Refraction in the water glass*

When passing from one medium to another, Snell's law of refraction applies (Fig. 8.18).

$$\frac{sin\theta_i}{sin\theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i} \tag{8.22}$$

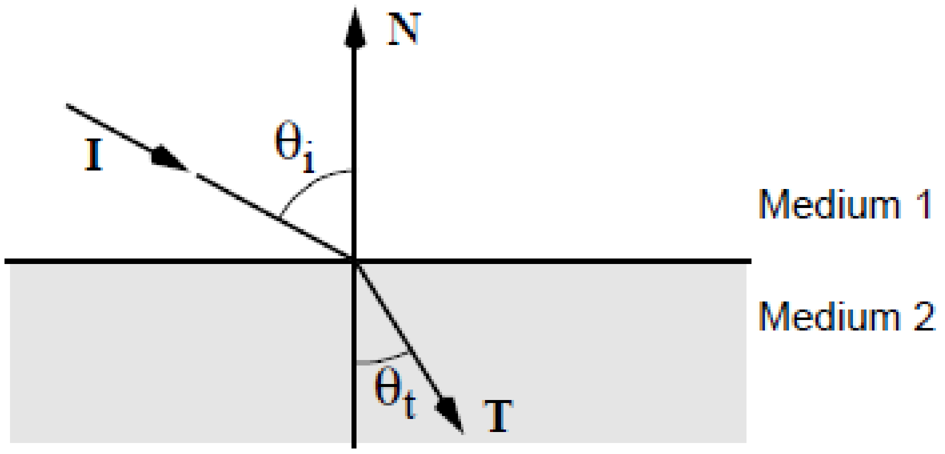


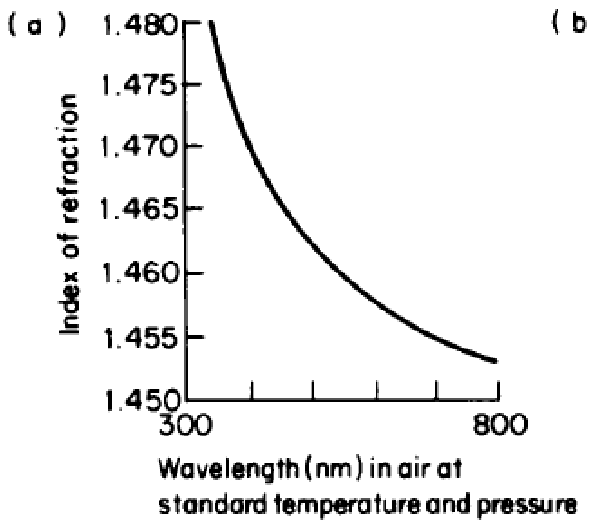
Figure 8.17: *Snell's law of refraction*

The reflected vector **T** results from a linear combination of **I** and **N**. A derivation is presented in the following Section 8.3.1.

$$\mathbf{T} = \alpha\mathbf{I} + \beta\mathbf{N} \tag{8.23}$$

The refractive index is a function of wavelength (Fig. 8.19 (a)).

Another phenomenon to consider is total internal reflection from optically thinner media.



(b)

Medium	Index of refraction*
Water	1.33
Ethyl alcohol	1.36
Carbon bisulfide	1.63
Air (1 atm, 20°C)	1.0003
Methylene iodide	1.74
Fused quartz (see left)	1.46
Glass, crown	1.52
Glass, dense flint	1.66
Sodium chloride	1.53

*Measured with respect to vacuum

Figure 8.18: *Refractive index as a function of wavelength (left) and some typical indices measured at a fixed wavelength (right).*

8.3.1 Derivation of the refraction vector

The vector **T** can be derived using Fig. 8.20.

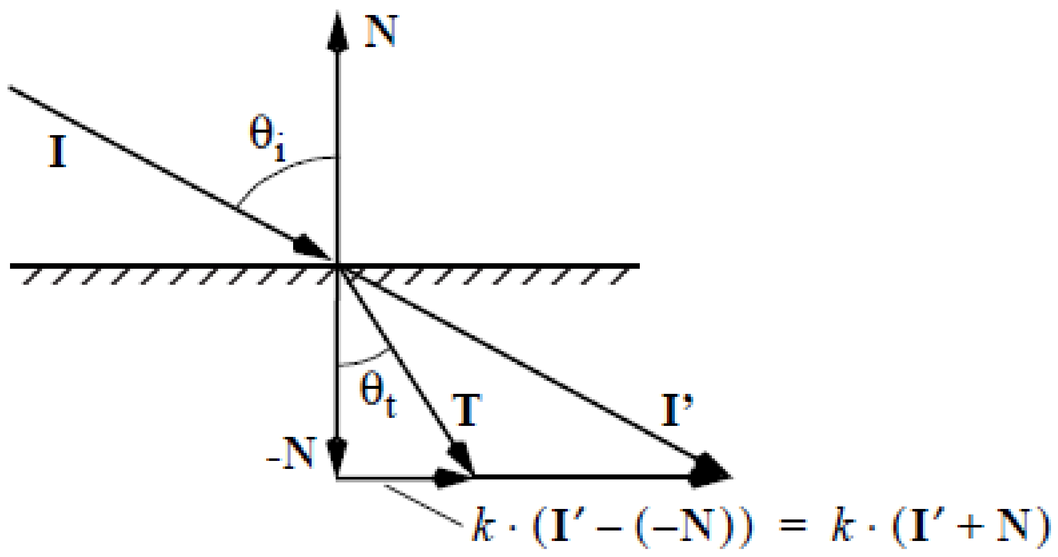

N
θi
I
θt
T
I'
-N
$k \cdot (\mathbf{I}' - (-\mathbf{N})) = k \cdot (\mathbf{I}' + \mathbf{N})$

Figure 8.19: *Arrangement for the derivation of the vector **T***

With $S_{t,i} = sin(\theta_{t,i})$ and $C_{t,i} = cos(\theta_{t,i}$ the following applies:

$$\frac{S_t}{S_i} = \eta_{it} \tag{8.24}$$

$$\mathbf{T} = \alpha\mathbf{I} + \beta\mathbf{N}$$

We are looking for the sizes α and β. The following notation is introduced for this purpose:

$$cos(\theta_i) = C_i \qquad = (\mathbf{N} \cdot (-\mathbf{I})) \tag{8.25}$$

$$cos(\theta_t) = C_t \qquad = (-\mathbf{N} \cdot \mathbf{T}) \tag{8.26}$$

By squaring the equation (8.24) one obtains

$$S_i^2 \eta_{it}^2 = S_t^2$$

or

$$(1 - C_i^2)\eta_{it}^2 = (1 - C_t^2)$$

This becomes

$$\begin{aligned}(1-C_i^2)\eta_{it}^2-1 &= -C_t^2\\ &= -[-\mathbf{N}\cdot\mathbf{T}]^2\\ &= -[-\mathbf{N}\cdot(\alpha\mathbf{I}+\beta\mathbf{N})]^2\\ &= -[\alpha(-\mathbf{N}\cdot\mathbf{I})+\beta(-\mathbf{N}\cdot\mathbf{N})]^2\\ &= -[\alpha C_i-\beta]^2\end{aligned}$$

because $\mathbf{N}\cdot\mathbf{N}=1$. Likewise:

$$\begin{aligned}1 &= \mathbf{T}\cdot\mathbf{T}\\ &= (\alpha\mathbf{I}+\beta\mathbf{N})\cdot(\alpha\mathbf{I}+\beta\mathbf{N})\\ &= \alpha^2(\mathbf{I}\cdot\mathbf{I})+2\alpha\beta(\mathbf{I}\cdot\mathbf{N})+\beta^2\mathbf{N}\cdot\mathbf{N})\\ &= \alpha^2-2\alpha\beta C_i+\beta^2\end{aligned}$$

Finally we get the system of equations:

$$\begin{aligned}(1-C_i^2)\eta_{it}^2-1 &= -[\alpha C_i-\beta]^2\\ 1 &= \alpha^2-2\alpha\beta C_i+\beta^2\end{aligned}$$

which only gives a concrete solution for α and β. Inserted into (8.23) we get

$$\mathbf{T}=\eta_{it}\mathbf{I}+(\eta_{it}C_i-\sqrt{1+\eta_{it}^2(C_i^2-1)})\cdot\mathbf{N}$$

This calculation is essential for calculating transparency in ray tracing (see Chapter 10).

8.3.2 Neglecting refraction (α *blending)*

Simple transparency can be achieved, for example, by so-called α-*blending*: If there is a transparent polygon P_1 between the viewer and polygon P_2, the following applies (Fig. 8.21):

Due to the filter effect of P_1, I_{λ_2} is weakened with

$$I'_{\lambda_2}=I_{\lambda_2}\cdot e^{-\alpha_1\delta t}$$

Linearizing and $\delta t=1$ results in

$$I'_{\lambda_2}=I_{\lambda_2}\times(1-\alpha_1)$$

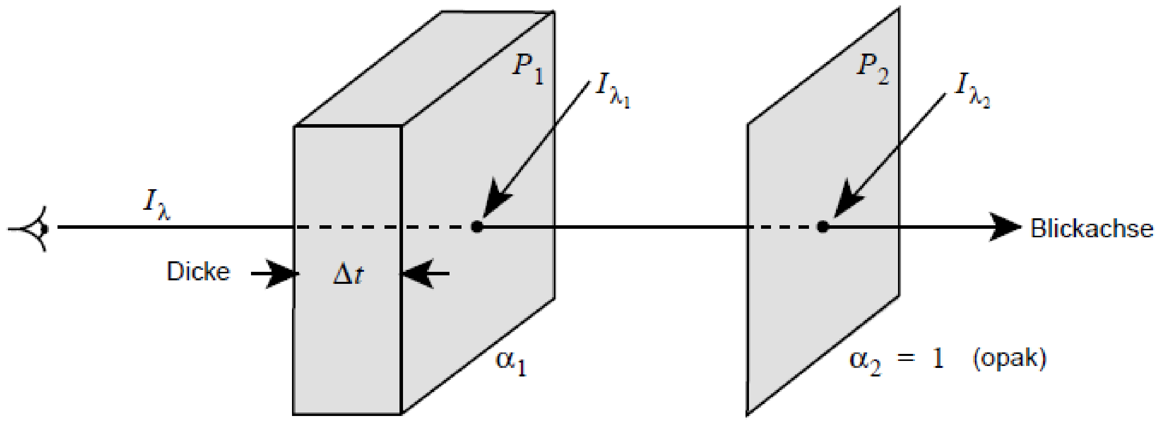


Figure 8.20: *Absorption factor in α blending*

The contribution of P_1 is

$$I'_{\lambda_1} = I_{\lambda_1} \cdot \alpha_1 \cdot \delta t = I_{\lambda_1} \cdot \alpha_1$$

so that

$$I_\lambda = I'_{\lambda_1} + I'_{\lambda_2} = I_{\lambda_1} \cdot \alpha_1 + I_{\lambda_2}(1 - \alpha_1)$$

where I_{λ_1} is the general volume intensity. This results in the filtered intensity I_λ for N polygons

$$I_\lambda = \sum_{i=1}^{N} \alpha_i I_{\lambda_i} \cdot \prod_{b=1}^{i-1} (1 - \alpha_b) \tag{8.27}$$

This method linearizes absorption effects and is often implemented in hardware.

8.4 cast shadow

Shadows contribute significantly to improving the realistic impression. They determine the points that are visible or invisible from a light source. If there are several light sources, each must be considered separately. Shadow calculation is therefore closely related to the Visible surface calculation problem. A visible surface algorithm calculates which surfaces are visible to the observer, while a shadow algorithm determines those surfaces which are "seen" by the light source. With extensive light sources and lighting effects, penumbra *(penumbra)* can occur. In what follows, however, we limit ourselves to point light sources.

The lighting equation is expanded as follows:

$$I_\lambda = I_{\alpha\lambda}O_{d\lambda} + \sum_{1<=i<=m} S_i f_{att_i} I_{p\lambda_i}[k_d O_{d\lambda}(mathbfN \cdot \mathbf{L_i}) + k_s O_{s\lambda}(\mathbf{R_i} \cdot \mathbf{V})^n] \tag{8.28}$$

$$S_i = \begin{cases} 0, & \text{if the light } i \text{ is blocked at this point} \\ 1, & \text{if the light } i \text{ reaches the object at this point} \end{cases}$$

8.4.1 Scan line shadow calculation

One way to calculate the shadow cast is to extend the scan line algorithm to include shadow projections. The edges of possibly shadow-casting polygons are projected onto all polygons in the direction of the light flux that are intersected by the current scan line (Fig. 8.22). This creates additional partial areas (partial polygons) on the existing polygons, which represent the shadow areas. As soon as such a projected line is crossed during scan conversion, the pixels lying in the shadow can be darkened accordingly.

A brute-force implementation of the procedure requires the calculation of all $n(n-1)$ projections of all polygons onto all other polygons. The cost is therefore of order $O(n^2)$.

8.4.2 Shadow Volumes

The area of influence of the shadow cast by a polygon can be understood as a volume. Such a *shadow volume* is defined by the light source and an object and is bounded by invisible *shadow polygons*. Each silhouette edge of the object relative to the light source creates a quadrilateral shadow polygon bounded on three sides by the silhouette edge itself and the rays of light emanating from the light source passing through the endpoints of the edge. The fourth side is formed by a scaled copy of the shadow-casting polygon that is far enough from the light source to be outside its sphere of influence (Fig. 8.23). The normals of the shadow polygons point outward, as do the normals of the original polygon and the scaled copy. So the normal of the copy is inverted.

The determination of whether an object point is in the shadow or not is achieved with the help of the shadow volumes. Objects lying behind a shadow polygon with the normal pointing towards the observer (*front-facing polygon*, polygon A and B in Fig. 8.23) are in the shadow. A shadow polygon with a normal pointing away from the observer (*back-facing polygon*, polygon C in Fig. 8.23) cancels the effect of a front-facing polygon again.

Imagining a vector from the observer V to a point on an object, the point is in shadow if the vector intersects more front-facing than back-facing shadow polygons. For this reason, points A and C in Fig. 8.24 (a) are shaded. As long as the observer himself is not in shadow, every other point is illuminated (point B). If the observer is in shadow himself, an additional rule must be observed: A point is also in shadow even if a corresponding back-facing polygon has not yet been cut for each shadow volume within which the observer lies (point B in Fig. 8.24 (b)).

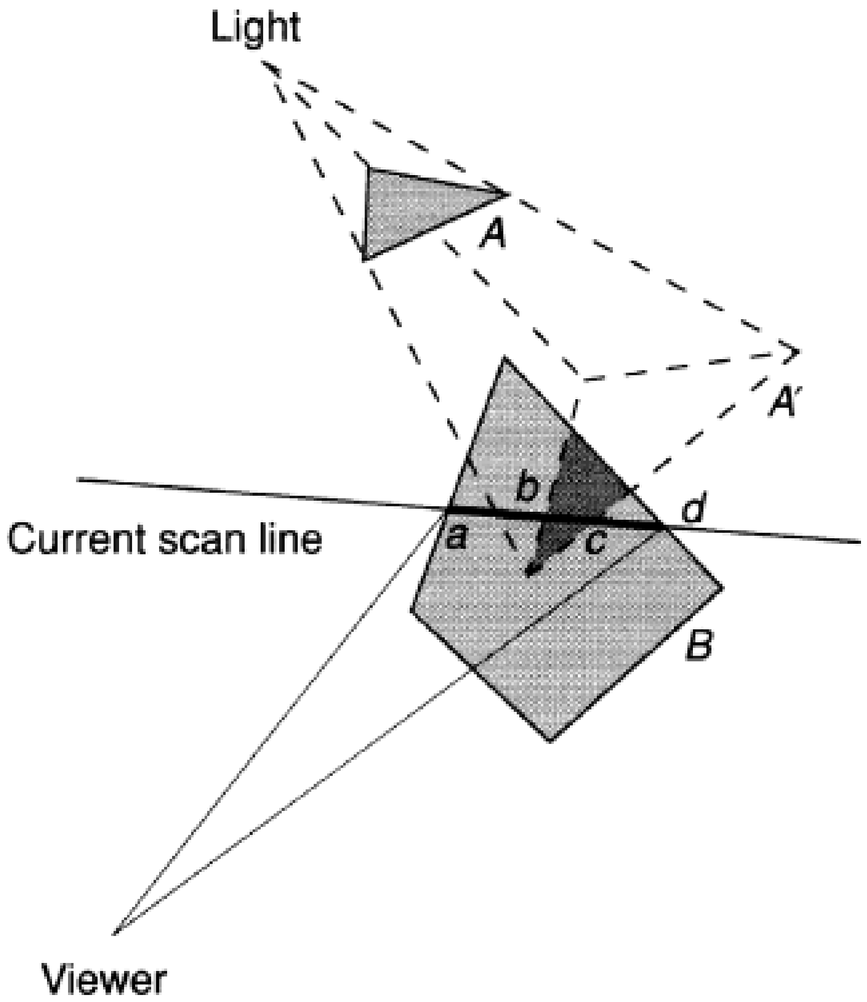


Figure 8.21: *Scan-Line Shadow Algorithm*

The algorithmic realization succeeds simply by assigning the value $+1$ to front-facing and -1 to back-facing polygons. A counter is initially initialized with the number of shadow volumes within which the observer is located and then incremented with the values of the shadow polygons located between the eye point and the point in question on the object. The point is in shadow if and only if the counter is positive ($>= 1$), otherwise it is illuminated. The number of shadow volumes within which the observer is located only has to be calculated once for each eye point. It can be determined by shooting a beam in any direction, starting from the eye point. For each back-facing polygon intersected, -1 is added, for each front-facing polygon, $+1$ is added to a counter previously initialized to zero. The value of the negated counter just corresponds to the number sought.

The shadow polygons and volumes are not calculated for each polygon, only for the object silhouette relative to the light source. Multiple light sources cause multiple shadow volumes per object. The complexity of the algorithm essentially depends on the additional number of shadow polygons.

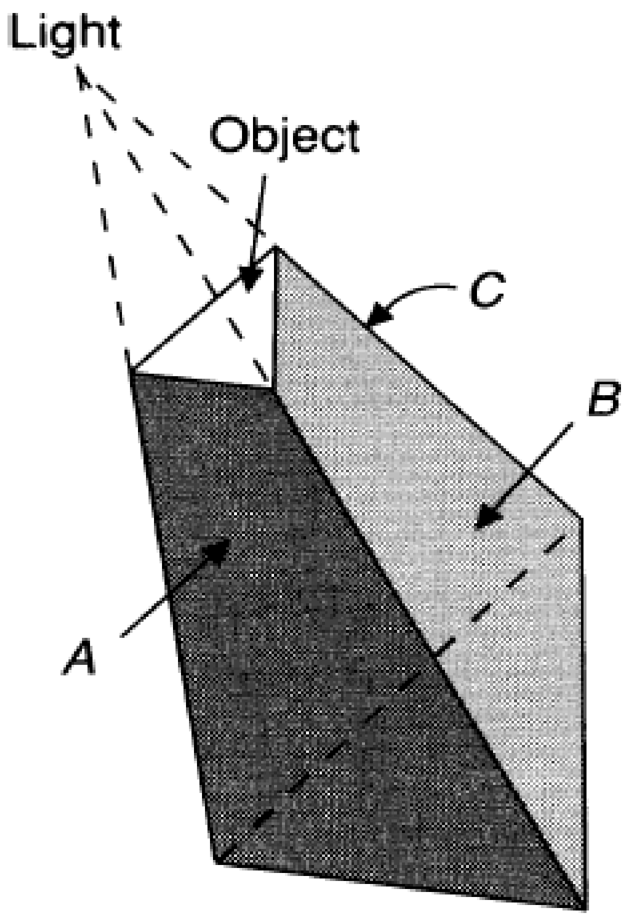


Figure 8.22: *Shadow volume defined by the light source and the shadow-casting polygon*

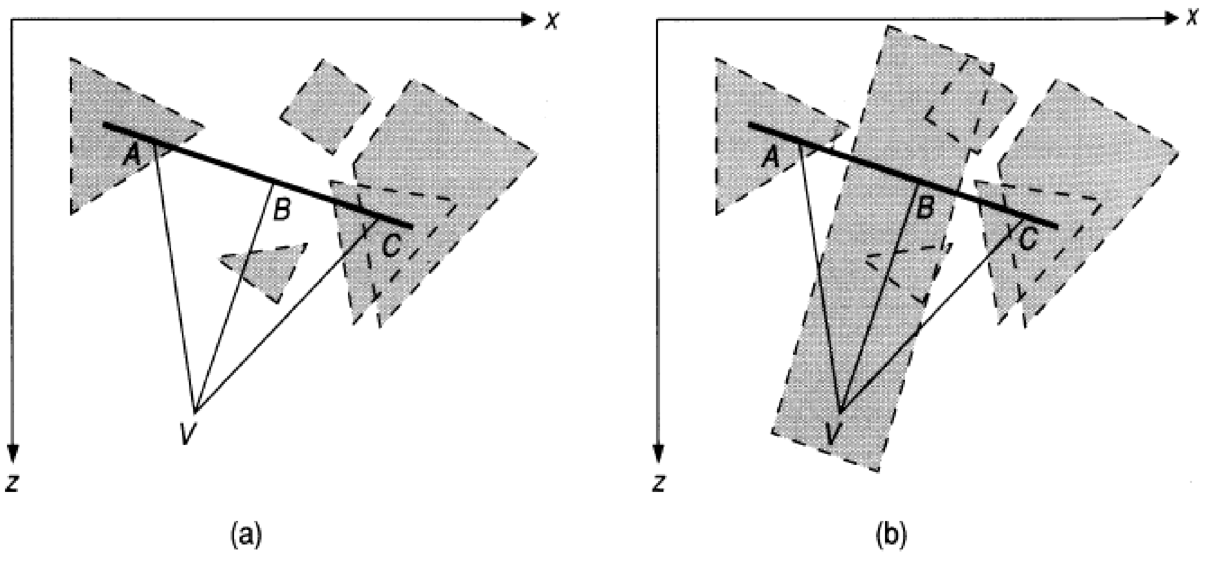


Figure 8.23: *Calculating the shadow cast with the help of shadow volumes*

8.5 Texture Mapping

An essential aspect of creating realistic images is the enrichment of surfaces with details. This can be both local modulations of color and intensity, as well as changes in the smoothness and surface texture of the object.

- The modulation of color and brightness is done by mapping a modulation function or an image onto the surface.
- The change in the surface texture is realized by a perturbation of the surface normal.

8.5.1 Mapping of brightness and color functions

This form of texture mapping can essentially be understood as a two-stage process according to Fig. 8.25.

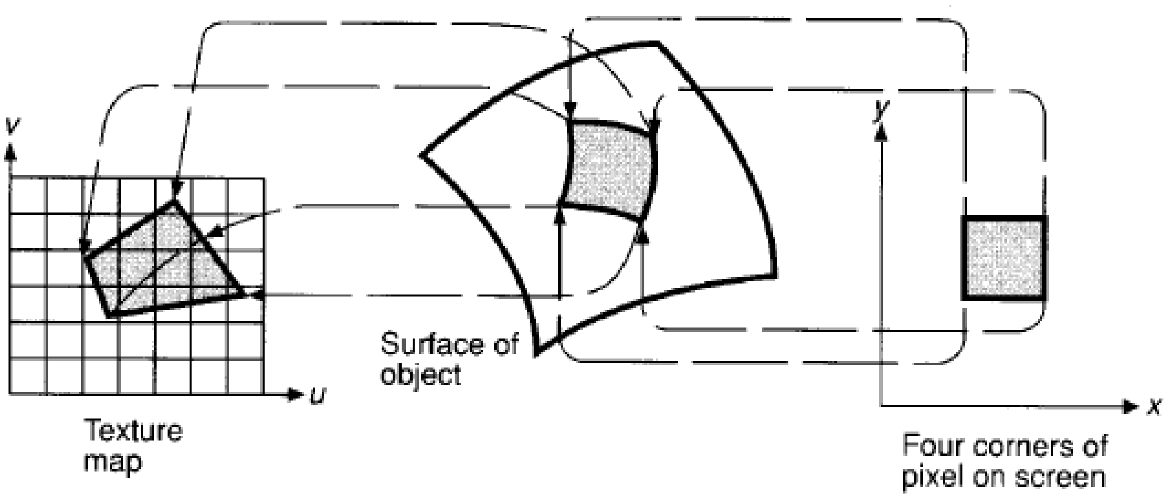


Figure 8.24: *Texture mapping from pixel to surface into texture map*

A mapping from the texture coordinate system (u, v) to the target area on the object surface is first carried out using a transformation rule. After that, object and surface are mapped into the screen coordinate system.

The texture space is given with the orthogonal coordinates (u, v) and with (θ, ϕ) the orthogonal coordinate system of the surface. We are looking for a mapping function of the form:

$$\theta = f(u, v) \qquad \phi = g(u, v) \tag{8.29}$$

or alternatively

$$u = r(\theta, \phi) \qquad v = s(\theta, \phi) \tag{8.30}$$

Usually, linear mapping functions of the kind

$$\theta = Au + B \qquad \phi = Cv + D \tag{8.31}$$

for use, where the constants A, B, C, D are given by the known corresponding points in both coordinate systems *(control points)*.

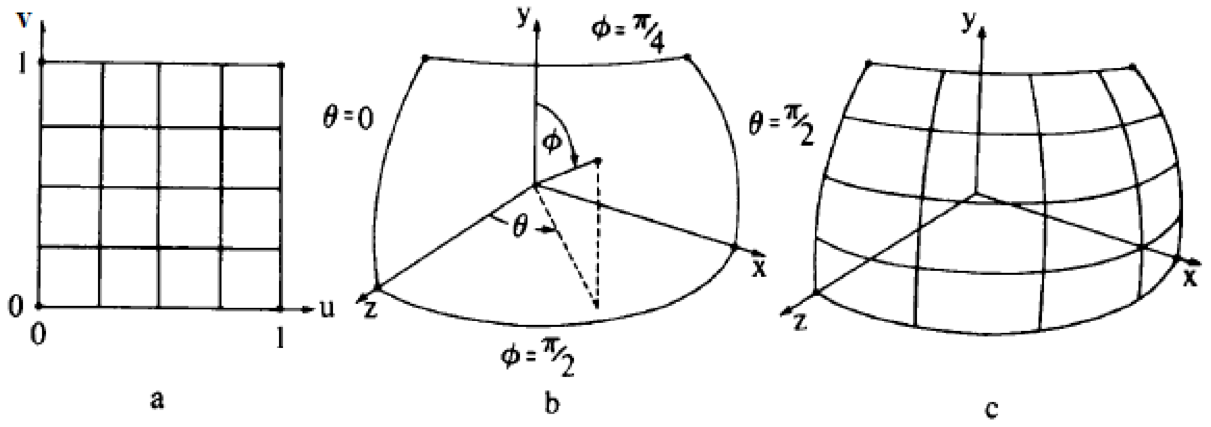


Figure 8.25: *Mapping a line texture to a spherical octant*

Example: *Texture mapping to a spherical octant*

The mapping function of a pattern given in the Cartesian coordinate system (u, v) to a spherical octant is to be calculated (Fig. 8.26). The parametric description of the sphere is:

$$\begin{aligned} x &= sin\theta sin\phi & & \\ y &= cos\phi & 0 &\le \theta \le \pi/2 \\ z &= cos\theta sin\phi & \pi/4 &\le \phi \le \pi/2 \end{aligned}$$

With the linear mapping function (8.31) and the control points

$u = 0, v = 0$	with $\theta = 0, \phi = \pi/2$
$u = 1, v = 0$	with $\theta = \pi/2, \phi = \pi/2$
$u = 0, v = 1$	with $\theta = 0, \phi = \pi/4$
$u = 1, v = 1$	with $\theta = \pi/2, \phi = \pi/4$

arises:

$$A = \pi/2 \qquad B = 0 \qquad\qquad C = -\pi/4 \qquad D = \pi/2$$

With this the mapping functions (8.31) become

$$\theta = \frac{\pi}{2}u \qquad \phi = \frac{\pi}{2} - \frac{\pi}{4}v$$

For the inverse functions one finds:

$$u = \frac{\theta}{\pi/2} \qquad v = \frac{\pi/2 - \phi}{\pi/4}$$

The table in Fig. 8.27 results for the mapping of the lines of the texture map in Fig. 8.26 (a). The lines shown are shown in Fig. 8.26 (c).

u	v	θ	ϕ	x	y	z
$1/4$	0	$\pi/2$	$\pi/2$	0.38	0	0.92
	$1/4$		$7/16\pi$	0.38	0.20	0.91
	$1/2$		$3/8\pi$	0.35	0.38	0.85
	$3/4$		$5/16\pi$	0.32	0.56	0.77
	1		$\pi/4$	0.27	0.71	0.65

Table 8.1: *Mapping of the lines from Fig. 8.26 (a) to the spherical octants in Fig. 8.26 (b)*

Example: *mapping for a cylinder*

Given is a cylinder of radius C_r and height C_h with:

$$X_C^2 + Y_C^2 = C_r^2 \qquad \text{with} \qquad 0 \leq Z_C \leq C_h$$

We are looking for the (u, v) coordinates for the (intersection) point $R_i(x_i, y_i, z_i)$, where $u \in [0..1]$ with the $+x$ axis starts and $v \in [0..1]$ runs along the cylinder wall parallel to the z axis. You get:

$$v = \frac{z_i}{C_h} u' = \frac{acos\frac{x_i}{C_r}}{2\pi}$$

If $y_i < 0$, set $u = 1 - u'$, else $u = u'$.

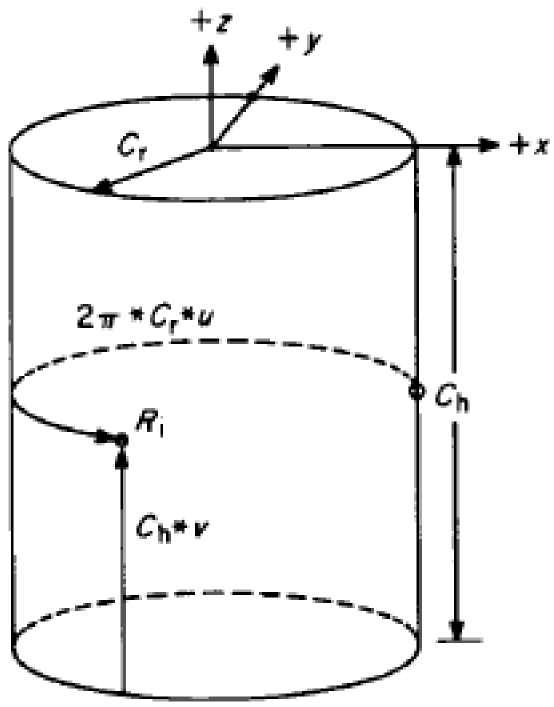


Figure 8.26: *Coordinate relationships in texture mapping to a cylinder*

8.5.2 Reflection Mapping

A special form of texture mapping is the so-called reflection mapping. A ray (vector) v_e to the viewer's eye is assumed for each image pixel. This is reflected via the surface normal n and results in the vector v_r. The texture to be mapped is assumed to be on a virtual sphere, as shown in Fig. 8.29. The color of the pixel then results from the intersection of the vector v_r with the virtual sphere and interpolation in the texture map.

Reflection mapping is a fast alternative to ray tracing, especially when simulating specular reflections.

8.5.3 Aliasing Effects in Texture Mapping

With the mapping methods described, aliasing effects can occur due to the finite screen resolution, with neighboring pixels only insufficiently representing the original resolution of the texture. This can be resolved by the following procedures:

1 *object space subdivision*
The object surface is subdivided until only one pixel is covered in the screen system. Suppose the textured spherical octant from Fig. 8.26 is to be rotated by $-45°$ around the y-axis and by $35°$ around the x-axis and in a screen resolution of 32 pixels with an orthographic projection (Fig. 8.31 (a)). The texture is given in a resolution of 6464, with the lines each being one pixel wide (Fig. 8.31 (b)).

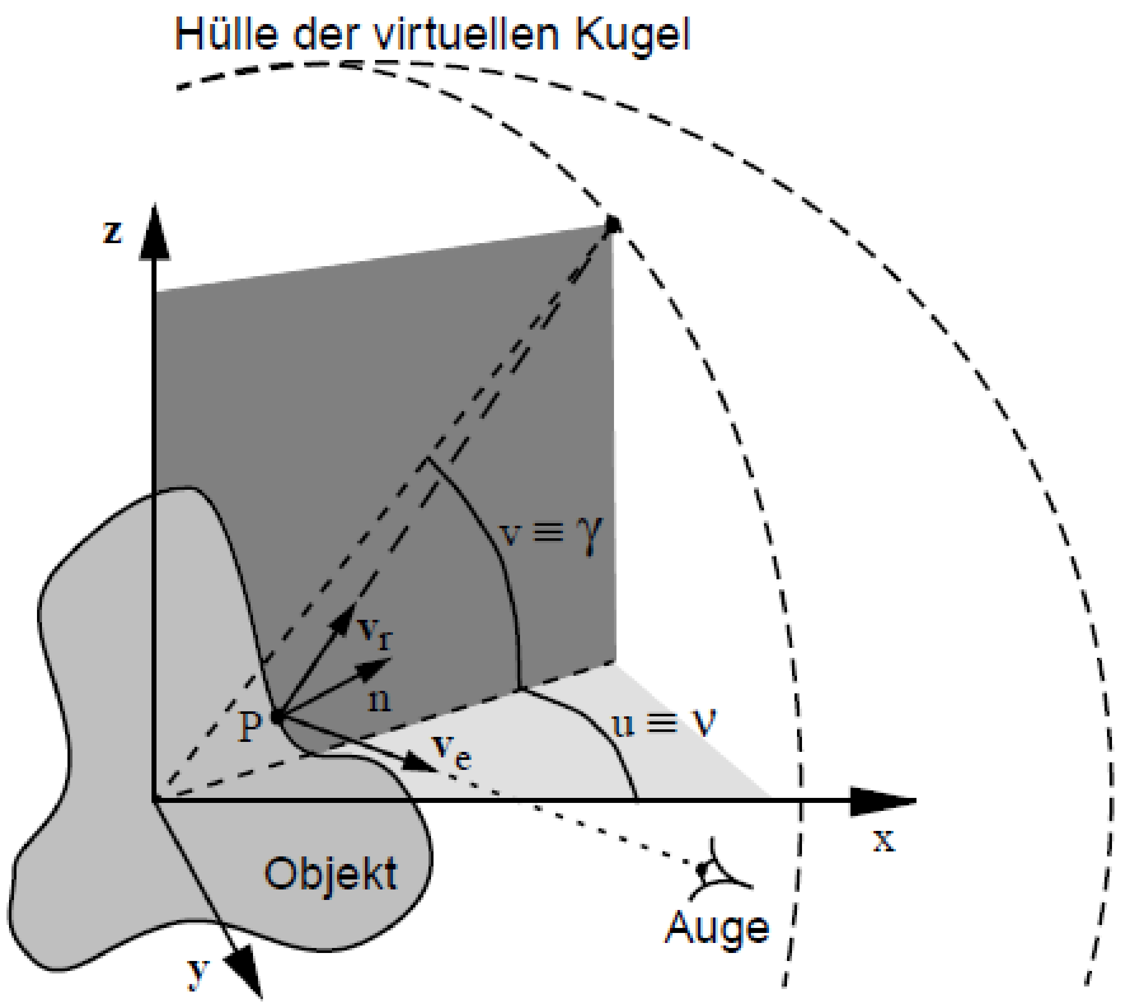


Figure 8.27: *Calculation of texture coordinates in reflection mapping*

First, the object surface is subdivided until only one pixel center is covered in screen coordinates (in the example 4 subdivisions). The resulting subpatch ranges from $0 <= \theta <= \frac{\pi}{32}, \frac{31}{64}\pi <= \phi <= \frac{\pi}{2}$ in object space . With the inverse mapping functions, the coordinates of the patch in texture space are also given:

$$\begin{array}{rlcll}
\theta = 0, & \phi = \pi/2 & \rightarrow & u = 0, & v = 0 \\
\theta = 0, & \phi = \dfrac{31}{64}\pi & \rightarrow & u = 0, & v = 1/16 \\
\theta = \pi/32, & \phi = \dfrac{31}{64}\pi & \rightarrow & u = 1/16, & v = 1/16 \\
\theta = \pi/32, & \phi = \pi/2 & \rightarrow & u = 1/16, & v = 0
\end{array}$$

In the example, exactly 4×4 texture elements, so-called *texels*, are covered by the pixel. The intensity of the texture can now be calculated by averaging the texel values or by other

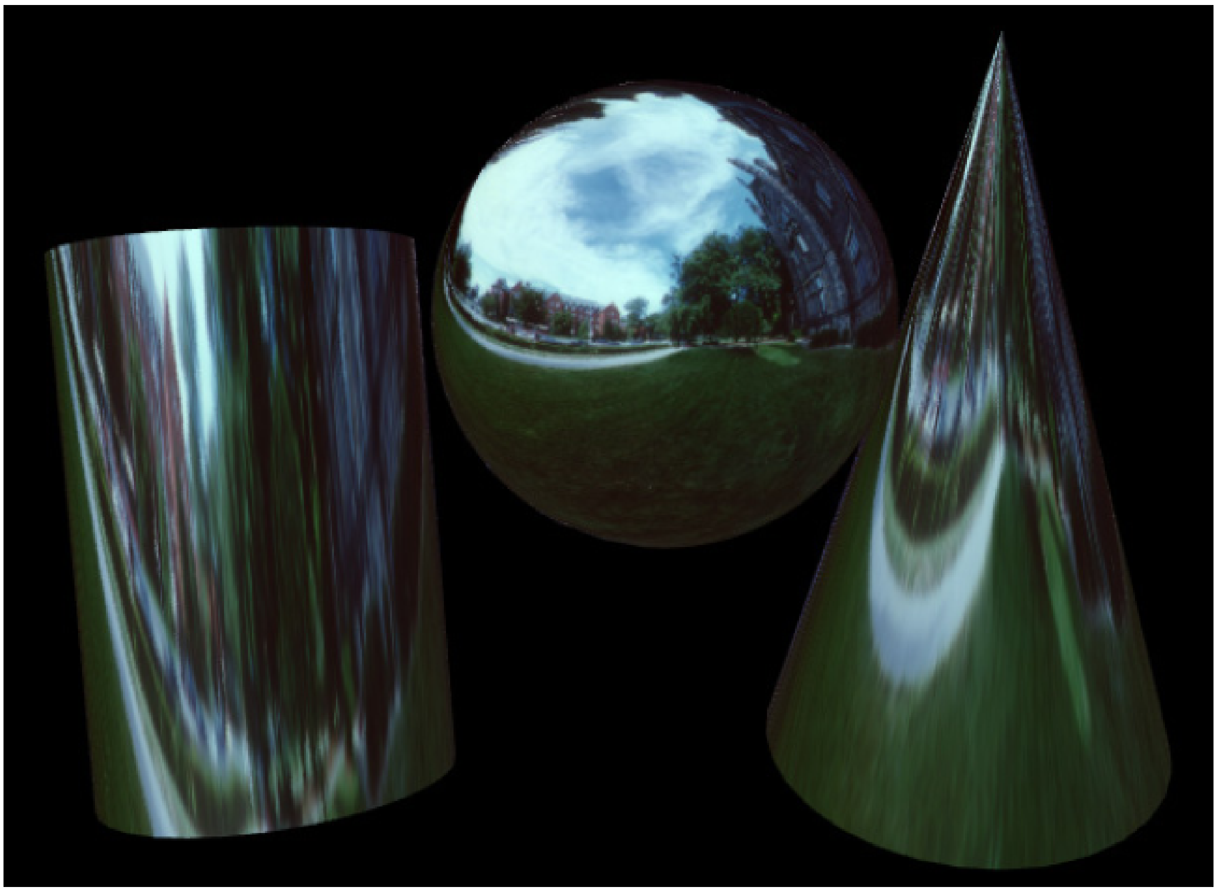
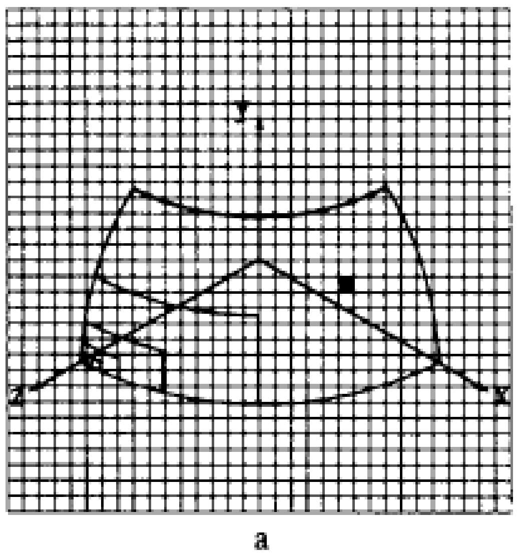


Figure 8.28: *Illustration of Reflection Mapping*



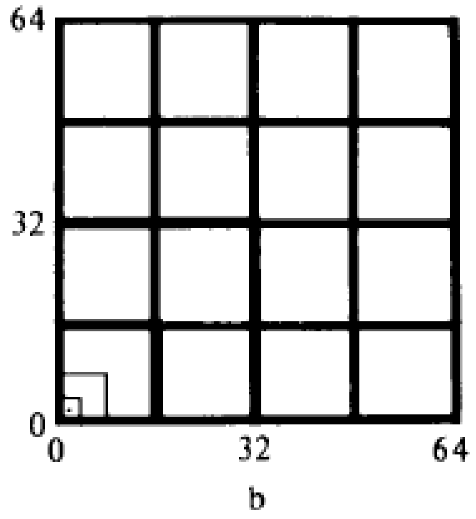


Figure 8.29: *Texture mapping by object space subdivision*

texture filters. In the opposite case, where a texel covers several pixels, interpolation (bilinear) between the texels has to be carried out. The diffuse reflection component of the lighting model is usually modulated with the value determined from texture mapping. The advantage of the method is that neither the inverse mapping function from screen to object space nor knowledge of the depth (z coordinate) of the subpatch is required.

2 *Inverse mapping*

Alternatively, the pixel area can be mapped from the screen coordinate system back to object space and then to texture space. To do this, the inverse transformations for the viewing transformation and the projection as well as the depth (z value, often known from a visible surface algorithm) must be known. The pixel intensity in the image space (diffuse reflection coefficient) then results from filtering the corresponding texels .

The procedure is again explained using the example of the rotated spherical octant. Consider the pixel 21 $<= P_x <=$ 22 and 15 $<= P_y <=$ 16 in Fig. 8.31. The normalized image space in orthographic projection is given by $-1 <= x' <= 1$ and $-1 <= y' <= 1$. The following applies:

$$x' = \frac{P_x}{16} - 1 y' = \frac{P_y}{16} - 1$$

By defining the unit sphere

$$z' = \sqrt{1 - (x'^2 + y'^2)}$$

with x', y', z' as coordinates in the camera coordinate system, the pixel corner points result as follows after the viewing transformation:

P_x	P_y	x'	y'	z'
21	15	0.3125	-0.0625	0.948
22	15	0.3750	-0.0625	0.925
22	16	0.3750	0	0.927
21	16	0.3125	0	0.950

The matrix of the camera transform and its inverse are:

$$T = \begin{bmatrix} 0.707 & -0.406 & 0.579 & 0 \\ 0 & 0.819 & 0.574 & 0 \\ -0.707 & -0.406 & 0.579 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T^{-1} = \begin{bmatrix} 0.707 & 0 & 0.707 & 0 \\ -0.406 & 0.819 & -0.406 & 0 \\ 0.579 & 0.574 & 0.579 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This results in the corner points of the pixel in the object space (world coordinates).

$$[xyz1] = [x'y'z'1] \cdot T^{-1}$$

to:

P_x	P_y	x	y	z
21	15	0.795	0.493	0.341
22	15	0.826	0.479	0.296
22	16	0.802	0.532	0.272
21	16	0.771	0.545	0.329

With the parametric description of the sphere

$$\begin{aligned} x &= sin\theta sin\phi \\ y &= cos\phi \\ z &= cos\theta sin\phi \end{aligned}$$

we find:

$$\phi = acos(y) \qquad \theta = asin\left(\frac{x}{sin\phi}\right)$$

Now the pixel vertex coordinates in texture space can be mapped from parameter space to texture space by the inverse mapping function (see example for subdivision)

$$u = \frac{\theta}{\pi/2} \qquad v = \frac{\pi/2 - \phi}{\pi/4}$$

be calculated:

This method is particularly useful for ray tracing (see Chapter 10).

P_x	P_y	ρ	ω	u	v
21	15	60.50°	66.04°	0.734	0.656
22	15	61.34°	70.30°	0.781	0.636
22	16	57.88°	71.28°	0.792	0.714
21	16	56.99°	66.88°	0.743	0.734

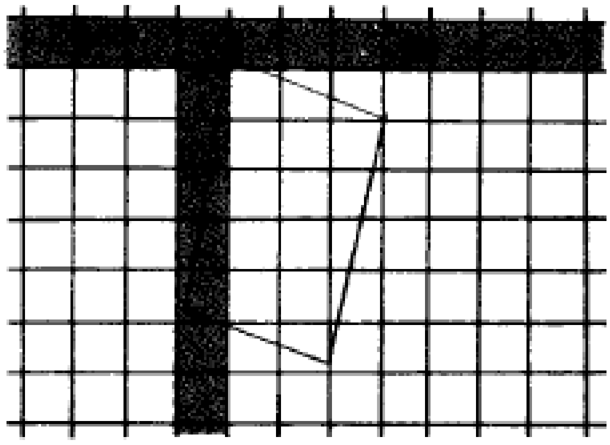


Figure 8.30: *projection of the pixel in image space onto the texel in texture space*

8.5.4 Bump mapping

So far, the surface has only been modulated in brightness and color, but not in its surface texture. To achieve this, the surface normal, which is used to calculate the lighting model, can be perturbated using a function or *bump map B(u,v)*.

In the general case let a point **P** of the surface with $\mathbf{P} = (x(s,t), y(s,t), z(s,t))$ in world coordinates (x, y, z) and in object parameter coordinates (s, t). The normal **N** is calculated with the help of partial derivatives:

$$N = \frac{\delta \mathbf{P}}{\delta s} \times \frac{\delta \mathbf{P}}{\delta t} = \mathbf{P_s} \times \mathbf{P_t} \tag{8.32}$$

In the case of polygonal surfaces, the calculation of the normal is often unnecessary because it is already given by Phong's interpolation.

$\mathbf{P}$ is now shifted to $\mathbf{P}'$ by any bump value B:

$$\mathbf{P}' = \mathbf{P} + \frac{B\mathbf{N}}{|\mathbf{N}|} \tag{8.33}$$

be calculated. The perturbed normal can be approximated by

$$\mathbf{N}' = \mathbf{N} + B_u(\mathbf{N} \times \mathbf{P_t}) + B_v(\mathbf{N} \times \mathbf{P_s}) \tag{8.34}$$

where B_u and B_v are the partial derivatives of the bump function $B(u, v)$ in texture coordinates. The derivation of this approximation can be found in the proceedings of SIGGRAPH '78 (pp. 286-292). The publication is entitled *"Simulation of Wrinkled Surfaces"* and is by James F. Blinn.
The mapping process is now analogous to standard texture mapping. Intermediate values can be generated by bilinear interpolation, while the partial derivatives are by finite differences of form

$$B_u = \frac{B(u_2, v_1) - B(u_1, v_1) + B(u_2, v_2) - B(u_1, v_2)}{2\Delta u} \tag{8.35}$$

$$B_v = \frac{B(u_1, v_2) - B(u_1, v_1) + B(u_2, v_2) - B(u_2, v_1)}{2\Delta v} \tag{8.36}$$

can be calculated (Fig. 8.33).

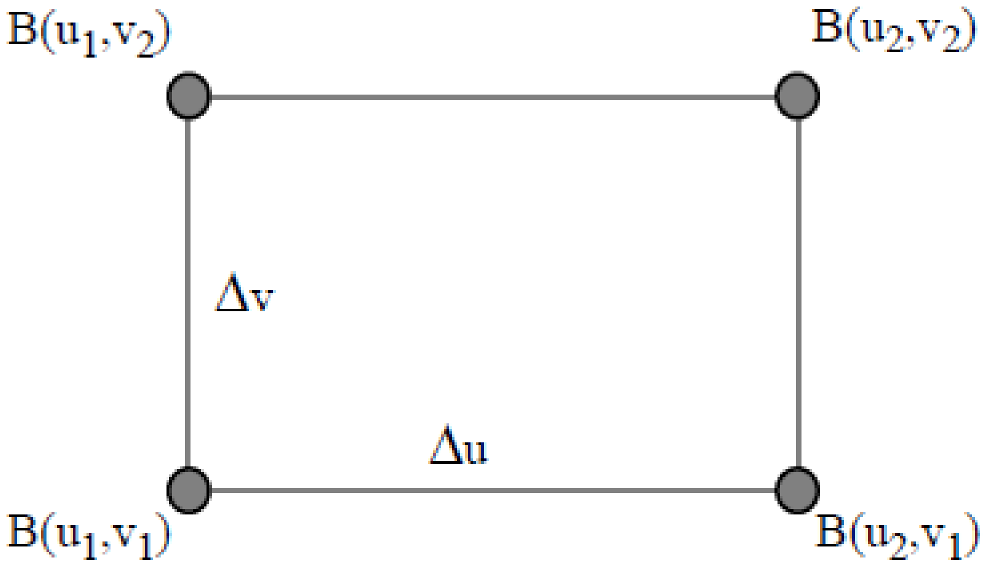


Figure 8.31: *Core points for calculating partial derivatives with finite differences*

`The type of interpolation is essential to avoid aliasing effects.`

Figure 8.34 shows four different texture mapping methods in comparison. While the example on the left shows an object that is only modulated in terms of brightness and color, the two images in the middle illustrate changes in the surface texture. You can clearly see the difference between bump mapping, in which the normals are changed but not the geometry itself, and displacement mapping, in which real changes are made to the geometry. The example on the right is another illustration of reflection mapping.

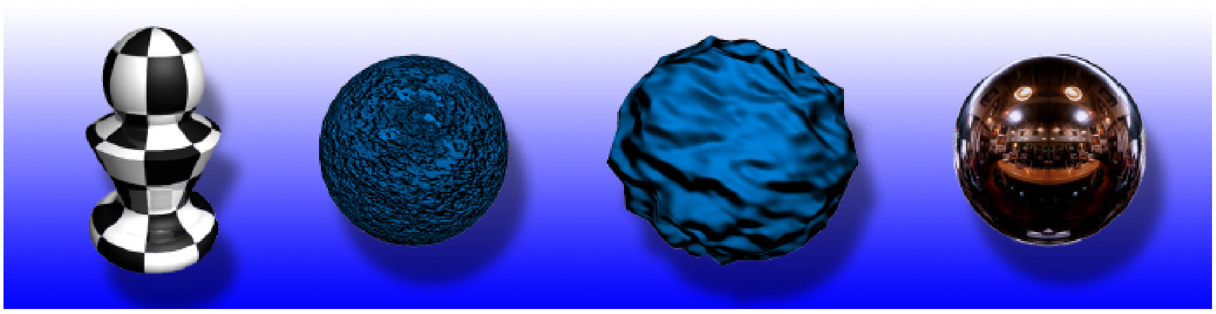


Figure 8.32: *Texture mapping methods in comparison (from left to right: modulation of colors - or brightness values, bump mapping, displacement mapping and reflection mapping)*

The Open Graphics Library (OpenGL)

9.1 Introduction

9.1.1 Graphics systems and standards

3D graphics functionality is increasingly required in many complex application scenarios as an essential part of a powerful software system. A basic problem is the different configuration of the underlying hardware and system software. Today, the palette ranges from simple PCs or Macs to powerful RISC workstations and mainframe computers. The operating systems and graphical interfaces are correspondingly different, ranging from Windows and MacOS to Windows/NT or X/Motif. Also, individual graphics capabilities and performance can vary greatly on the same platform. This is particularly the case in the PC sector.

software systems, proposals for the standardization of corresponding interfaces have therefore been discussed for a long time. Such standards enable the hardware-independent and thus cross-platform development of software and its exchange. A particular problem is the integration of 3D graphics functionality into existing 2D window systems.

Historically, for example, $Postscript$ has established itself as a description language for the electronic exchange of documents containing text and 2D graphics for 2D graphics. Another important step in the area of 2D graphics towards user-friendly user interfaces was the introduction of the "window" paradigm and its implementation, for example in the form of X-Windows, which became the standard for the entire UNIX world. In particular, X's client-server architecture allows integration into modern computer networks.

In 3D graphics, the situation was and is a bit more confused: Various standards have been proposed and partly also adopted internationally, but none (with the exception of OpenGL) has found sufficient industrial acceptance. The *Graphics Kernel System (GKS)* developed at the TH Darmstadt was originally standardized by ISO9905, although it was conceptually designed for vector graphics systems. Based on this, the *Programmer's Hierarchical Interactive Graphics System (PHIGS)* and corresponding extensions $(PHIGS++)$ were proposed, which were accepted as the standard by ANSI. The PHIGS concept is based on a $Display$ list, which contains the entire description of the object to be displayed and its attributes. The advantage of a display list is that objects only have to be described once, even if they are displayed many times. However, a disadvantage is the inflexibility regarding the interactive manipulation of objects and their features, because the display list has to be traversed each time. A *central structure memory* takes over the administration of the objects. This means that higher-level data structures and hierarchies are set up to describe scenes. However, the integration of powerful rendering functionality, such as e.g. B. Texture mapping. Finally, PEX should be mentioned at this point, which represents a 3D extension of the X server with regard to PHIGS features.

9.1.2 The OpenGL

The *Open Graphics Library (OpenGL)* was originally developed as IRIS GL in several versions by the *Silicon Graphics (SGI)* company since the early 1980s. It was primarily intended for efficient programming of the manufacturer's high-performance graphics workstations. The library was not based on a well-founded programming language concept. With the success of the SGI systems, however, GL gained more and more influence in the field of 3D graphics programming and was finally made available to other hardware and software manufacturers as OpenGL by SGI. In the meantime, many large and well-known companies are on the list of OpenGL licensees.

OpenGL is controlled by the *OpenGL Architectural Review Board (OpenGL ARB)*, an industry consortium of major hardware and software companies that must approve any language convention change. The following 10 companies are currently members of the ARB:

• 3D Labs	http://www.3dlabs.com/
• Apple	http://www.apple.com/
• ATI	http://www.atitech.com/
• Dell	http://www.dell.com/
• Evans & Sutherland	http://www.es.com/
• Hewlett-Packard	http://www.hp.com/
• IBM	http://www.austin.ibm.com/software/OpenGL/index.html
• Intel	http://www.Intel.com/
• Matrox	http://www.matrox.com/
• Microsoft	http://www.microsoft.com/hwdev/devdes/openglalt.htm
• nVidia	http://www.nvidia.com/
• sgi	http://www.sgi.com/software/opengl/
• SUN	http://www.sun.com

The current list of members of the OpenGL ARB and further information are available on the Internet at *http://www.opengl.org/developers/about/arb.html*.

OpenGL represents a software interface for 3D graphics programming. It is neither descriptive nor object-oriented. It provides a set of functions and procedures that provide high-quality rendering functionality. It is also independent of the window system used.

OpenGL writes graphic primitives, such as points, lines, polygons, pixels or bitmaps, into a $Framebuffer$. The way it is displayed depends on a number of selectable modes, which can be set individually using additional commands. The primitives are typically specified by vertices, and each point is processed independently in a pipeline. The OpenGL allows direct control over basic functions for 2D and 3D display, such as

- Simple definition of geometric objects
- Specification of transformation and projection matrices
- Compilation of display lists
- Definition of lighting and shading models
- Transparency and Fog

- Antialiasing
- Texture Mapping
- Framebuffer operations for composing images

However, structures for describing complex scenarios or objects are not provided. The application program (client) sends commands to the GL server, which interprets and executes them.

Since OpenGL performs all operations in or on the frame buffer, this also represents the interface to the window system used. The window system is therefore responsible for the allocation of the frame buffer resources and for their administration, including the output on the monitor .

9.1.3 OpenGL organizational principles

Fig. 9.1 schematically shows the basic data flow in OpenGL. First, commands are either executed directly or optionally saved in a *Display list*. An $Evaluator$ evaluates, among other things, polynomial functions for describing curves and surfaces, in order to then transform them into polygonal approximations. In the next step, all operations are carried out that work on the primitives (lines, points, polygons) defined via vertices: transformations, clipping, lighting, etc. A further step is the $rasterization$ $(scanconversion)$, which creates a set of framebuffer addresses. Another class of operations, such as *Depth buffering* or $Blending$, are performed on the generated $Fragments$ before the results are written to the framebuffer. Bitmaps are routed past the pipeline in a bypass and can either be taken into account as a texture during rasterization or written directly to the frame buffer as a fragment.

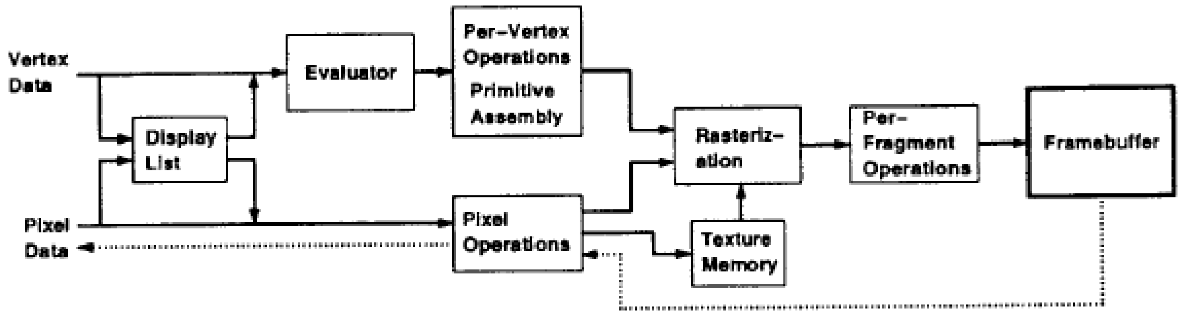


Figure 9.1: *Block diagram of data flow in OpenGL*

As already mentioned, in contrast to PHIGS, OpenGL does not support higher-order object descriptions. For example, no functions for rendering concave polygons or NURBS are provided. Such higher primitives must first be decomposed into convex polygons. The *OpenGL Utility Library* provides the appropriate routines for describing and decomposing meaningful objects, which, in addition to concave polygons, also knows NURBS curves and surfaces as well as spheres, cylinders and cones.

9.2 The OpenGL Pipeline

9.2.1 Graphic primitives

Most geometric primitives are described in OpenGL by simple constructs specifying vertices, normals, color, texture coordinates, etc., and are enclosed in the $glBegin$ and $glEnd$ commands. For example, a triangle with coordinates (0,0,0), (1,0,0), and (1,1,0) is specified as follows:

```
glBegin(GL_POLYGON);
glVertex3i(0,0,0);
glVertex3i(1,0,0);
glVertex3i(1,1,0);
glEnd();
```

A corresponding color could, for example, be set outside the construct by a command of the form

```
glColor3f(0.3,0.4,0.0);
```

be specified.

OpenGL is a state machine, ie all set commands, such as color definitio or transformation, are retained until they are changed.

Figure 9.2 shows the ten different types of graphic primitives that are defined in this way. As can be seen in the picture, all these primitives can be described by a simple list of vertices. Corner points are described by 2, 3 or 4 coordinates (homogeneously) and can be supplemented by normal ($glNormal()$), color ($glColor()$) or texture coordinates ($glTexCoord()$). Proper shading thus requires specifying object normals per vertex. The color can be specified in RGBa or as an index into a look-up table. The initialization takes place via the function

```
auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
```

which sets the so-called *display mode* of a Windows, which also contains the color model to be used.

The arguments of the functions and procedures described allow the attributes to be configured individually. There are several variants for most functions, which differ in their suffixes. Thus, by appending a 2, two-dimensional data can be specified, while a 3 requires three parameters for 3D data. There are also different types of arguments. Some functions can be appended with a v, indicating that the function expects a pointer to a vector (array) of values as an argument, rather than an argument list.

9.2.2 Transformations

Commands that do not specify vertices or their attributes must not be placed within the glBegin/glEnd construct. This allows efficient processing of all vertex-related commands. When a

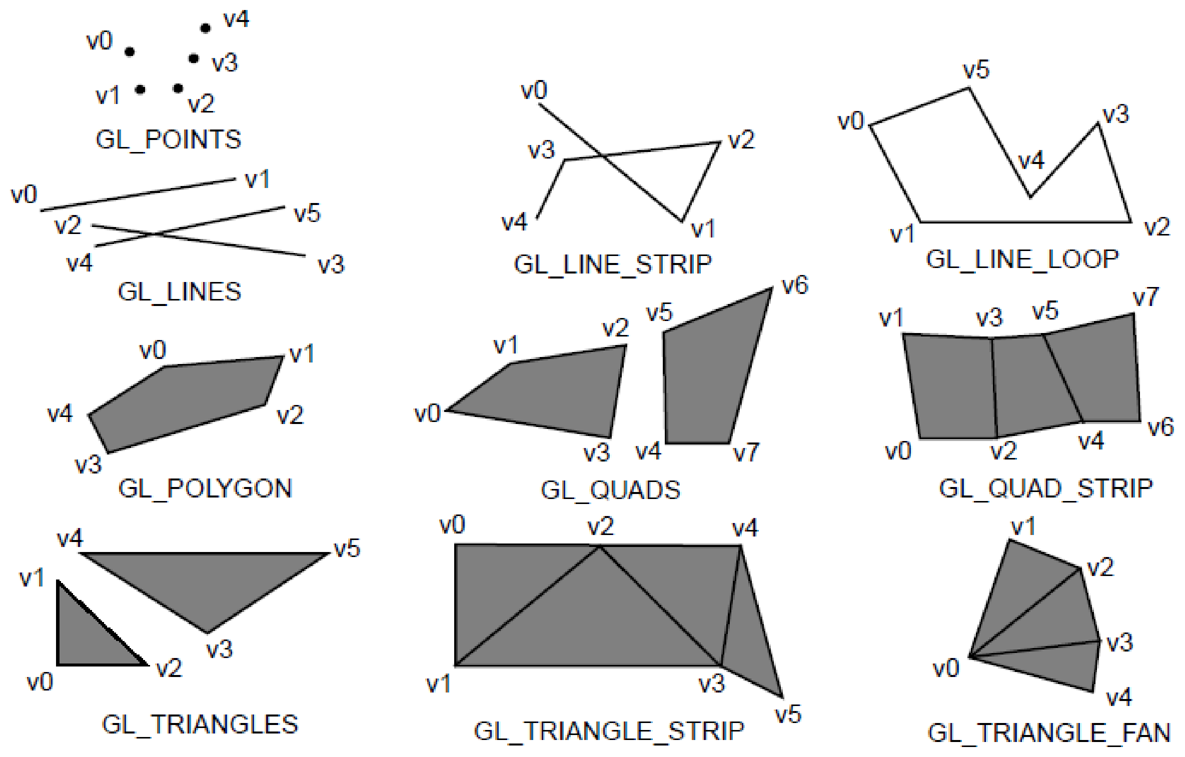


Figure 9.2: *Types of geometric primitives in OpenGL*

Suffix	**Data Type**	**Open GL type**
b	signed char	GLbyte
s	short	GLshort
i	long	GLint
f	float	GLfloat
d	doule	GLdouble
v	pointer to vector of values	

Examples:

glVertex2f(...);	float argument list (two float values)
glVertex2fv(...);	pointer to float arguments
glVertex3i(...);	long argument list (three long values)

vertex is specified, *current color, current normal, and current texture* coordinates are used to calculate the corresponding values of that vertex. Fig. 9.3 clarifies the processing.

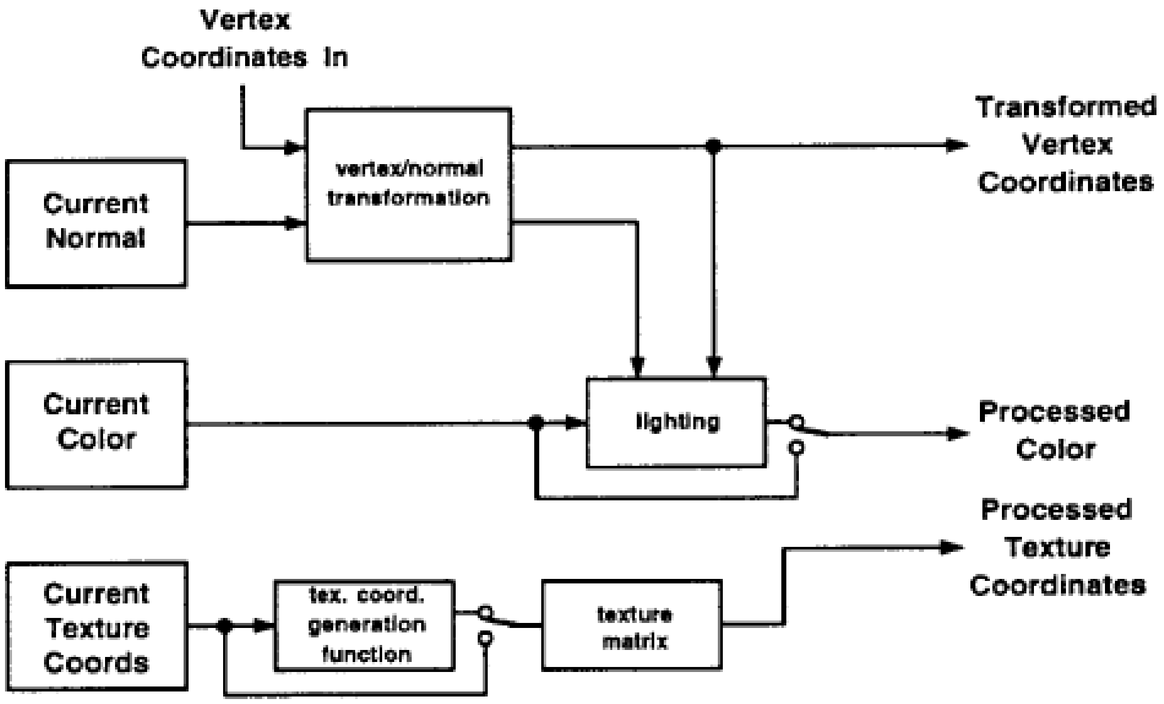

Vertex Coordinates In
Current Normal
vertex/normal transformation
Transformed Vertex Coordinates
Current Color
lighting
Processed Color
Processed Texture Coordinates
Current Texture Coords
tex. coord. generation function
texture matrix

Figure 9.3: *Relationship between coordinates and attributes of a corner point*

All vertices are transformed using a model-view matrix, a 4x4 matrix containing both linear and translational components. Likewise, the texture coordinates can be manipulated using linear operators. About the function

```
glMatrixMode(GL_MODELVIEW); /* relies on modelview transformation */
```

can be used to determine which *matrix stack* should be influenced by the following operations. There are four modes to choose from: Modelview, Projection, Texture and Color. The transformations set in one of the four modes are each managed on a separate stack.

Typical commands used in transformation operations are:

```
glLoadIdentity();
glTranslatef(x,y,z);
glRotate(f,x,y,z);  /* f:angle, x,y,z: axis */
```

In the following example, general matrix transformations, which are described using 4x4 matrices, are used. A corner point $\mathbf{v}$ is transformed to $\mathbf{v}' = \mathbf{LMNv}$.

Example:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);  /* apply transformation N */
glMultMatrixf(M);  /* apply transformation M */
```

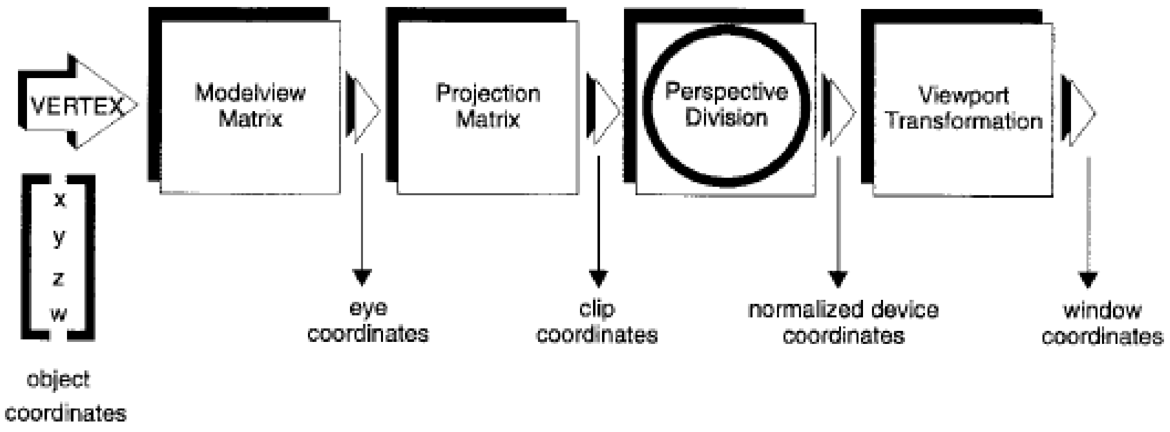


Figure 9.4: *sequence of vertex transformations to represent primitives*

```
glMultMatrixf(L);  /* apply transformation L */
glBegin(GL_POINTS);
glVertex3fv(v);  /* draw transformed vertex v */
glEnd();
```

9.2.3 Lighting Models

The color of each corner point can be calculated using an illumination model or can be predetermined individually for each corner point. The parameters of the lighting model are divided into three categories:

- Material properties to describe the physical object properties.

```
glMaterialfv(PARAMS);  /* material properties */
```

- Light Source Properties to describe the specific characteristics of a single light source.

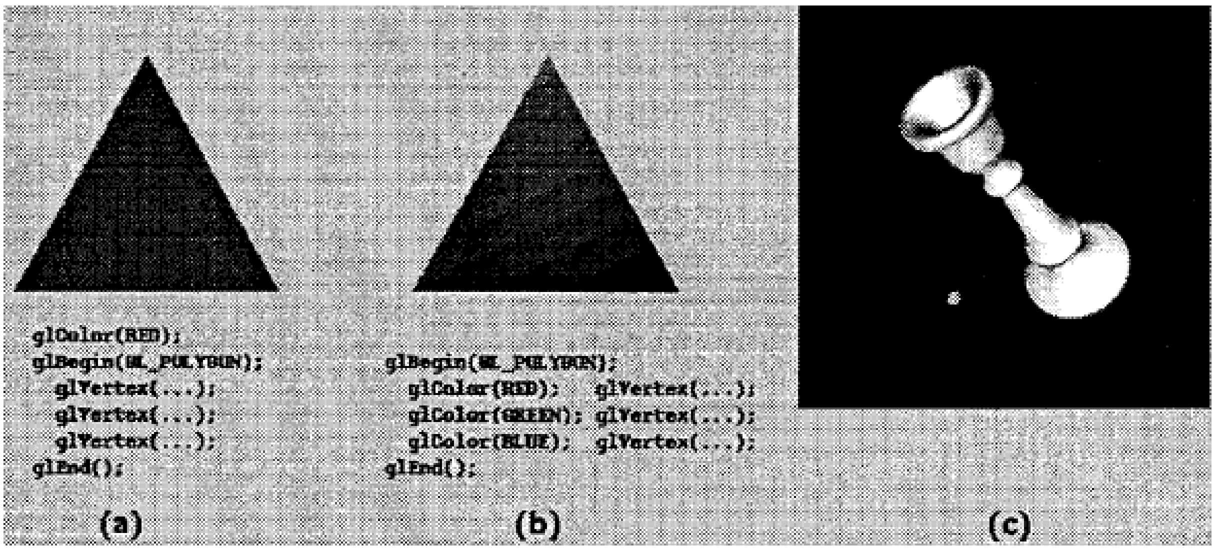
```
glLightfv(PARAMS);  /* Light source and its properties */
```

- Global properties of the lighting model to describe the approximation quality of the model used.

```
glLightModelfv(PARAMS);  /* Model and its properties
```

The lighting is always based on corner points. Fig. 9.5 shows three different images as an example, whereby the lighting is not activated on the left $(glDisable(GL_LIGHTING))$.

The variety of definitions of individual surface properties is illustrated in Fig. 9.6. Different properties are set for each teapot in order to simulate real materials.



```
glColor(RED);
glBegin(GL_POLYGON);
  glVertex(...);
  glVertex(...);
  glVertex(...);
glEnd();
```

```
glBegin(GL_POLYGON);
  glColor(RED);   glVertex(...);
  glColor(GREEN); glVertex(...);
  glColor(BLUE);  glVertex(...);
glEnd();
```

Figure 9.5: *a) Monochrome triangle*
b) Gouraud-shaded triangle
c) More complex shaded object

9.2.4 Texture Mapping

Textures can be treated in a similar way to the vertices of primitives. Basically, all textures are processed using a texture generation function and then transformed with a texture matrix in relation to the object. A texture is defined, for example, by

```
glTexImage2D(PARAMS);
```

The texture is generally an RGBa image.

Additional control over the type of mapping and filtering is provided by using

```
glTexParameterf(PARAMS);
```

For the actual referencing of the texture coordinates with regard to an object, there are commands of the form

```
glTexCoordf(PARAMS);
glTexGen(PARAMS); /* allows automatic generation of the
          Coordinates – useful in Fig.~9.8 b,c */
```

In this context, the support of MipMaps (Fig. 9.7) by OpenGL is of particular importance. It allows the definition of one and the same texture in different resolution levels. This allows more efficient management of textures in dynamic scenes. If the object is far from the viewer, a lower resolution instance of the texture is chosen or interpolated between two textures.

Automatic generation of mipmaps from the highest resolution image is enabled by

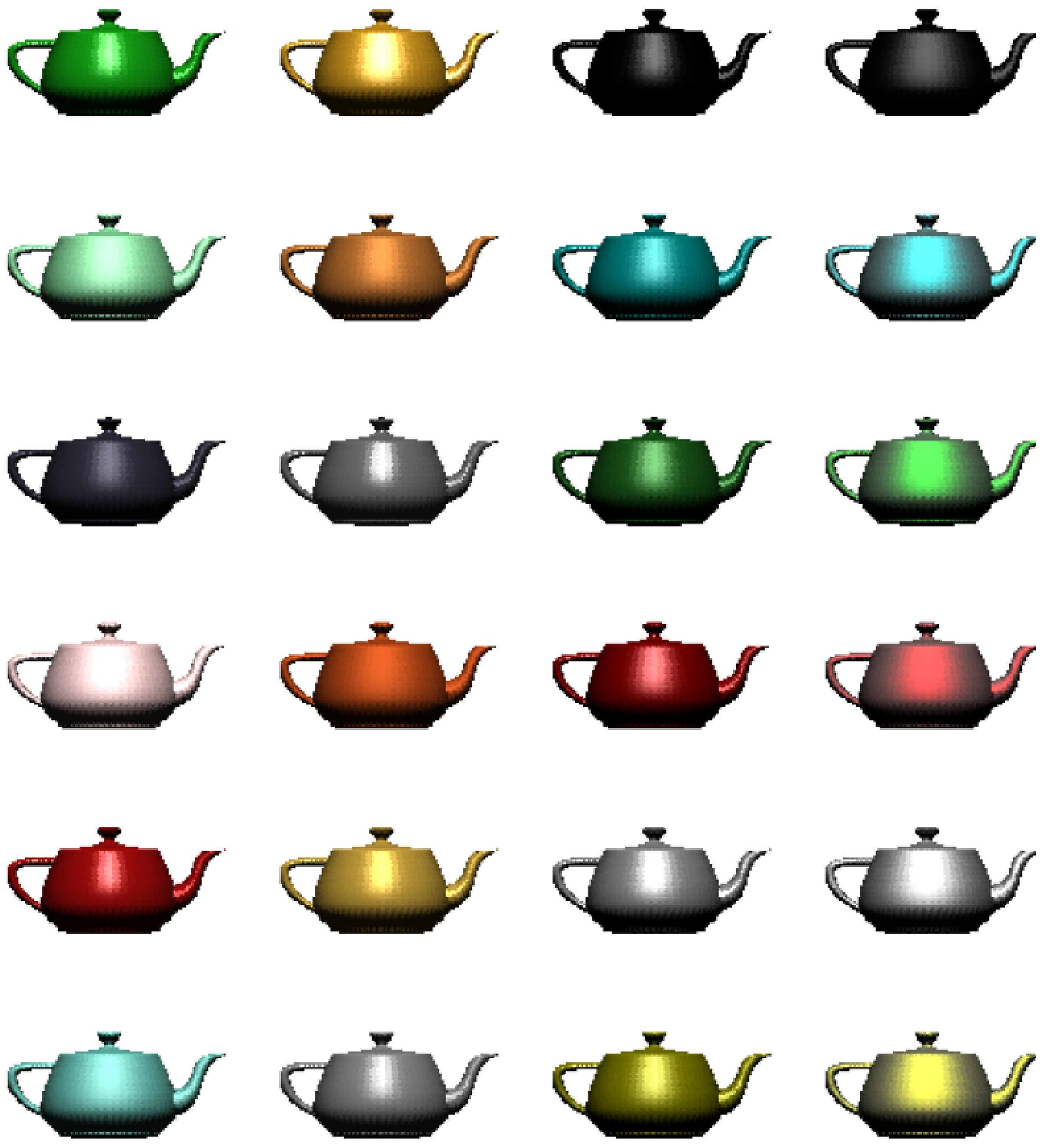


Figure 9.6: *Utah teapots with different surface properties*

```
gluBuild2DMipmaps(PARAMS);   /* from the GLU library */
```

supports. Figure 9.8 shows three examples of texture mapping with OpenGL.

Like textures, the creation of fog (fog) is also supported, which is calculated using the blending functions of the color attributes. A distinction is made between linear and exponential superposition

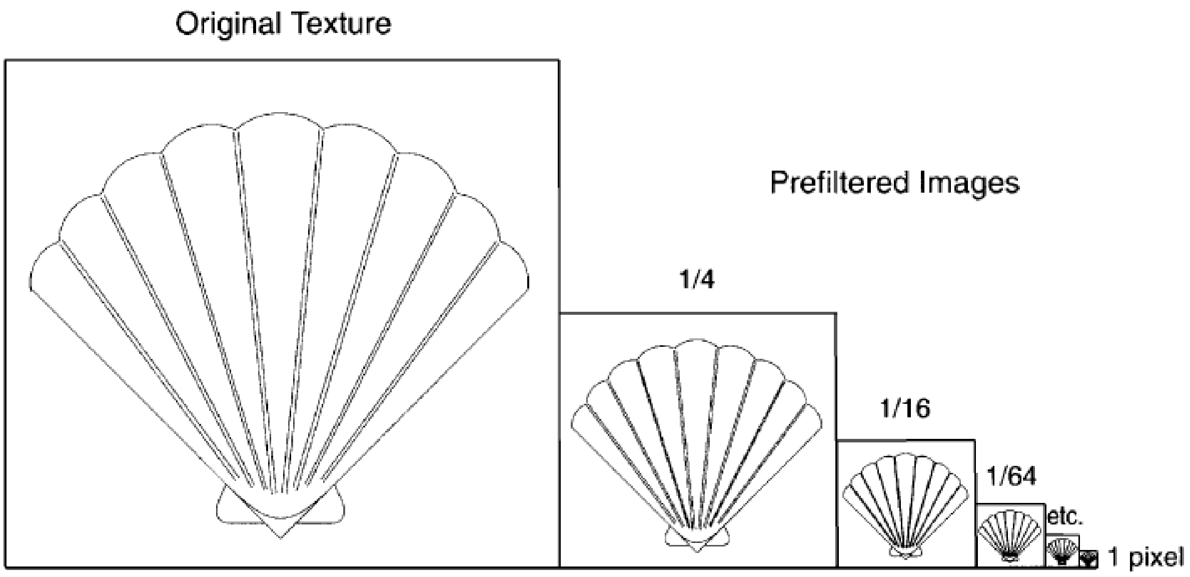


Figure 9.7: *mipmap representation of a texture*

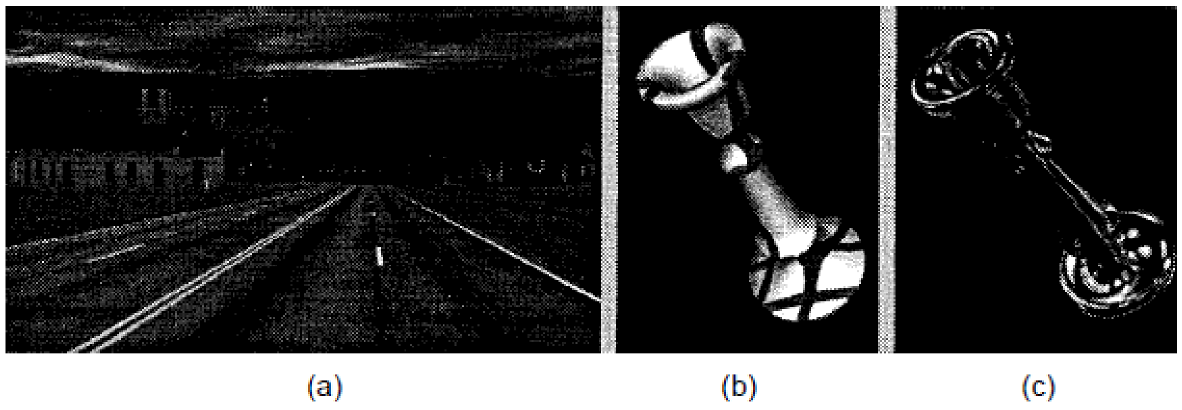


Figure 9.8: *a) Typical scene from many textured polygons (driving simulation)*
b) Contouring through texture mapping
c) reflection mapping

9.2.5 Clipping and projection

After the primitives are assembled and transformed, they must be clipped to the six canonical clipping planes. These are for this purpose

```
glClipPlane(PARAMS);
```

define. According to Chapter 5, the clipping method used depends on the type of primitive (point, line, convex polygon). If necessary, new vertex lists are generated, with the associated attributes having to be approximated by linear interpolation. The six half-spaces defining the clipping planes are $-w \le x \le w$, $-w \le y \le w$, and $-w \le z \le w$.

As is known, the projection is carried out by dividing the homogeneous coordinate: x/w, y/w, z/w, the resulting values being in [-1,1]. The viewport is controlled by the commands

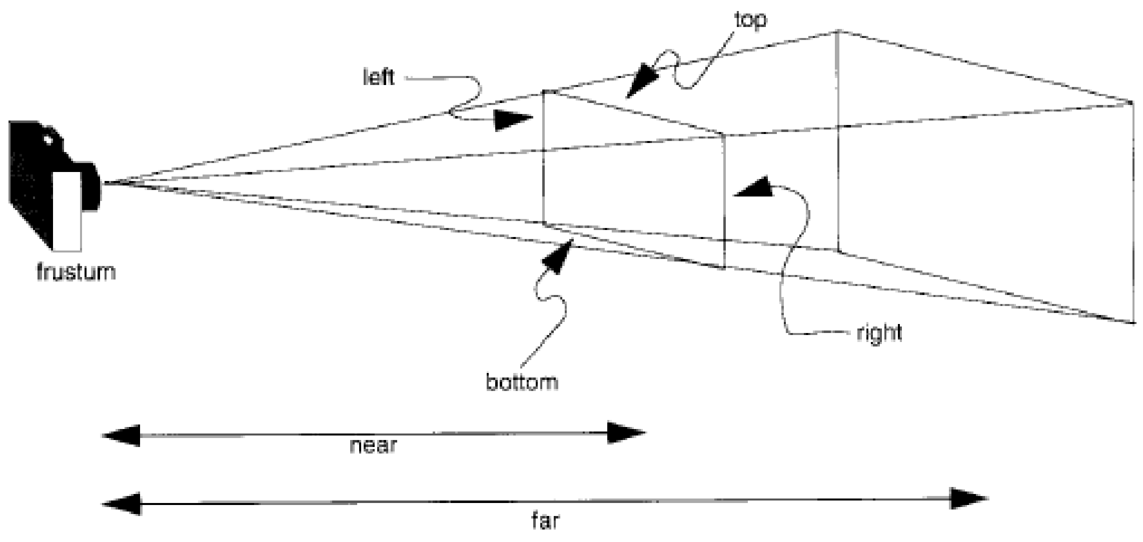


Figure 9.9: *Viewing Volume of Perspective Projection*

```
glViewPort(PARAMS);
glDepthRange(PARAMS);
```

controlled.

The function

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

produces the following matrix in homogeneous coordinates:

$$\mathbf{R} = \begin{bmatrix} \frac{2n}{rl} & 0 & \frac{r+l}{rl} & 0 \\ 0 & \frac{2n}{tb} & \frac{t+b}{tb} & 0 \\ 0 & 0 & \frac{-(f+n)}{fn} & \frac{-2fn}{fn} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{9.1}$$

$\mathbf{R}$ is defined as long as $l \neq r, t \neq b$ and $n \neq f$.

9.2.6 Raster scan conversion and antialiasing

The raster scan conversion breaks down the illuminated, transformed and clipped primitive into individual fragments. In addition to a pixel coordinate in the frame buffer, these also include color, texture coordinates and the depth, e.g. When scan-converting (Chapter 6) a line or polygon, the associated values are interpolated along the primitives to obtain values for each fragment. Both thickness and pattern of the line are generated here.

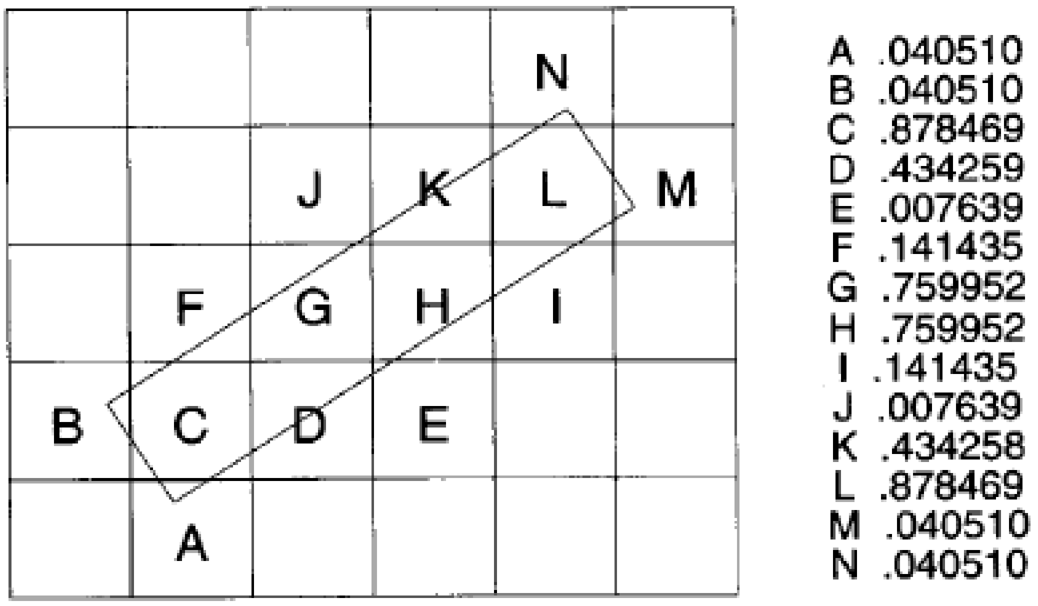
```
glLineStripple(PARAMS);   /* width of rasterized lines */
glLineWidth(PARAM);   /* line stipple pattern */
```

polygons. You will first be commanded by the command

```
glEnable(GL_LINE_SMOOTH); /* for lines */
```

activated.

A percentage coverage by the object is calculated for each pixel, as shown in Fig. 9.10, and set as the a-value for the corresponding pixel in RGBa mode.



A	.040510
B	.040510
C	.878469
D	.434259
E	.007639
F	.141435
G	.759952
H	.759952
I	.141435
J	.007639
K	.434258
L	.878469
M	.040510
N	.040510

Figure 9.10: *Determining the percentage coverage of a pixel*

Because antialiasing is a computationally expensive operation,

```
glHint(GL_NICEST);   /* as beautiful primitives as possible */
glHint(GL_FASTEST);   /* fastest possible variant */
glHint(GL_DONT_CARE);   /* no preference */
```

the system can also be given a note on execution.

9.2.7 Pixels and Bitmaps

Pixel quantities, such as images, are always routed past the entire geometry pipeline and broken down directly into fragments. There are a number of functions for manipulating pixel sets in the OpenGL library, such as

```
glDrawPixels(PARAMS);
```

which writes a block of pixels directly into the framebuffer. The arguments contain pointers to the corresponding pixmap or the values of the height and width of the rectangle. Additional parameters allow a more precise definition of the data format in which the pixmaps are available and can be used to implement reading routines for common formats such as TIFF, GIF, RGB etc.

You can also work with bitmaps. The function

```
glBitmap(PARAMS);
```

writes a binary image to a specified position in the framebuffer.

9.2.8 The framebuffer

As previously described, at the end of the processing pipeline is the framebuffer, which can be viewed as a rectangular area of pixels. The information of a pixel consists of a number of bit planes, as in Fig. 9.11, in which different pieces of information are stored. In OpenGL, a distinction is made between *color, depth, stencil* and *accumulation*. The stencil buffer can contain additional information, which can be manipulated each time new pixels are written according to defined comparison operators $(glStencilFunc())$. The accumulation buffer is extremely interesting for multipass applications. He's going through

```
glAccum(PARAMS);
```

controlled.

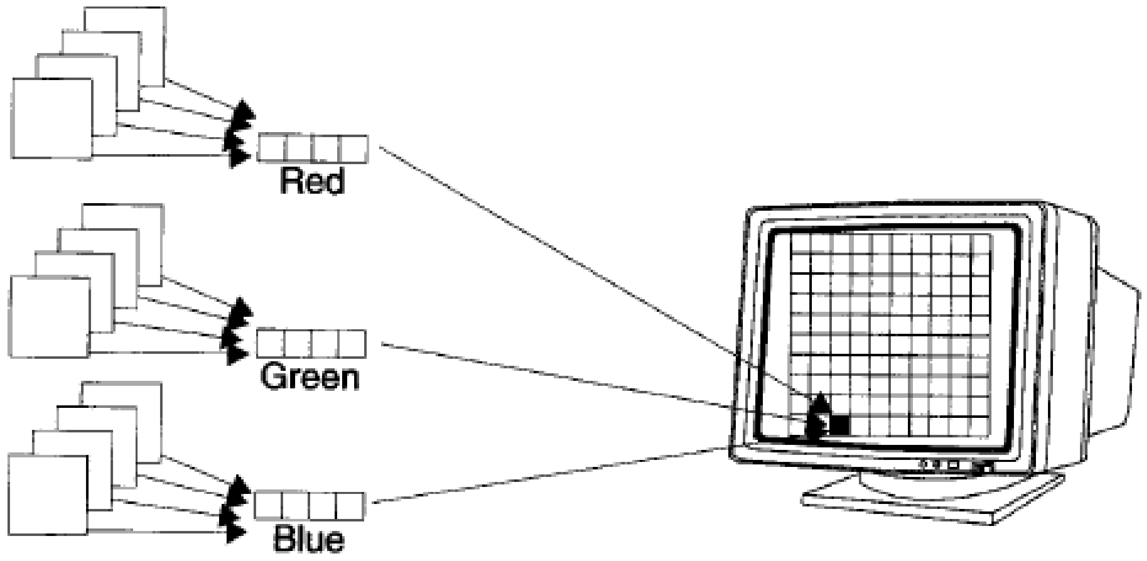


Figure 9.11: *Bitplanes of the framebuffer*

OpenGL supports multipass rendering algorithms. With this method, one and the same scene is rendered several times and the results are mixed in the accumulation buffer according to certain algorithms.

`An example of this is full-frame antialiasing, where the viewpoint is jittered (noisy) and the resulting images are accumulated. Likewise multipass methods allow the implementation of temporal antialiasing.`

OpenGL supports *Double-buffering* and *Stereo*, with a maximum of four framebuffers available (front-left, front-right, back-left and back-right). However, the support of these single stores depends on the implementation and the environment used. Minimum requirement is a buffer (front left).

A single fragment (pixel with attributes) is subjected to a series of tests, which can be controlled individually, before it is finally written into the framebuffer. The buffers used for this are initialized separately by

```
glClear∗(PARAMS);
```

where * can stand for *Color, Index, Depth, Stencil* or *Accum.*

9.2.9 Variable

Another important function for implementing high-quality object descriptions based on OpenGL are *Evaluators*. They are used to evaluate polynomial curves or surfaces and are based on a Bézier basis. On the one hand, the curve or surface can be evaluated point by point. In this case, the user takes care of defining a corresponding GL primitive:

```
glMap2f(PARAMS);
...
glBegin();
...
glEvalCoord2(PARAMS);
...
glEnd();
```

The second option is to use the command

```
glEvalMesh2(PARAMS);
```

which calculates a two-dimensional network of points or lines. A NURBS interface is provided in the GLUtilities.

Subroutines, which very often have to be executed in unchanged form, can be encapsulated in display lists. These are defined by a name and can be accessed via it.

```
glNewList(list, mode);  /∗ defines the beginning of the list and a
                           unique identifier: list ∗/
...
glColor3fv(color_vector);
...
```

```
glEndList();
...
glCallList(list);
```

Other important functions are $feedback$ and $selection$, which return the framebuffer information for a given pixel or describe which parts of a scene are in a certain area of the buffer. They are important in the context of 3D picking operations.

The selected objects are managed in a *Selection Stack*.

9.3 Integration into window systems

As described above, OpenGL provides an interface for 3D graphics programming, but does not have its own functions for managing the frame buffer resources. This is the task of the corresponding window system. In the following, the integration of OpenGL in X will be briefly discussed as an example.

The OpenGL is implemented in the form of an extension of X, the GLX. GLX is a set of functions that allows for a compact and generic wrapping in X. The extension defines a specific network protocol for the OpenGL rendering functions, which are encapsulated in the X byte stream.

Basically, the OpenGL needs a frame buffer, in which the graphic data can be written. Such a pixel array is a $Drawable$ in X. A $Window$ - a form of the drawable - additionally specifies a $Visual$, which describes the current frame buffer configuration of the window. However, the visuals provided by X are not sufficient for the GL requirements and must be extended with Depth, Accumulation, Stencil, etc. A second type of X visuals is also extended to meet GL requirements: the $Pixmap$, which allows off-screen rendering into a software framebuffer.

To use a GL-enabled drawable, the user must first generate an OpenGL context that targets the current drawable. A copy of an OpenGL renderer is initialized with the information about this drawable. The OpenGL renderer is conceptually considered as an X server extension, so that an X client (application program) can connect to it and send OpenGL commands. Fig. 9.12 illustrates the concept described.

This concept also allows the use of several GL contexts at the same time, whereby each GL-enabled drawable can also understand the complete X instruction set. The buffers not used by X are simply ignored. However, the synchronization of X and GL commands must be guaranteed by the user.

As can be seen in Fig. 9.12, there is also a bypass to the X server. This direct way of OpenGL rendering is primarily of interest for computers that have a hardware subsystem. In this case,

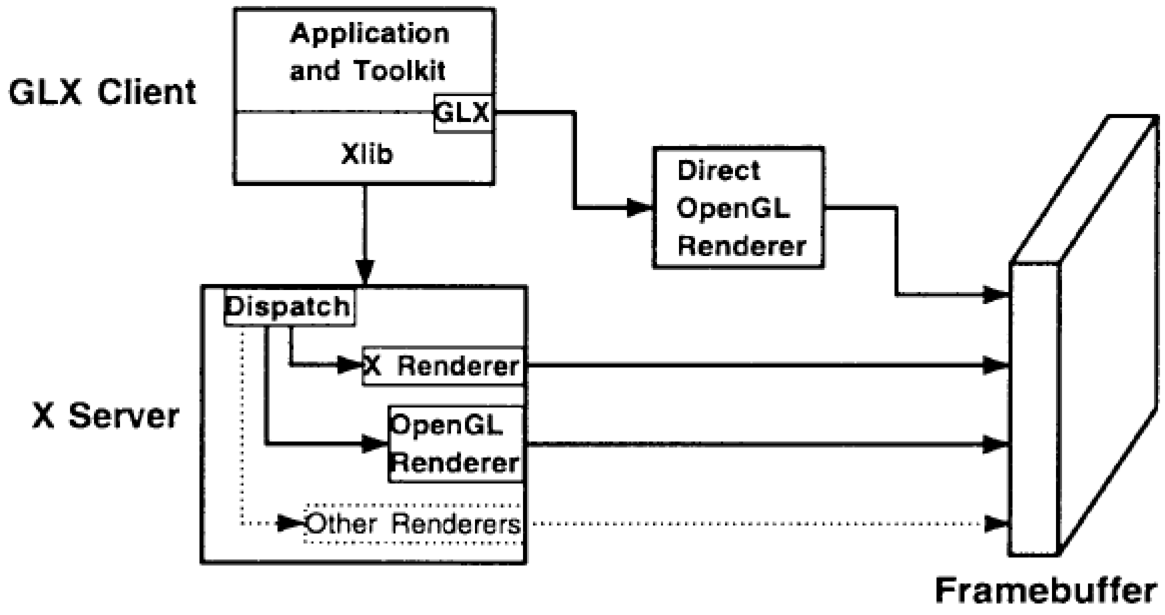


Figure 9.12: *GLX client, X server and OpenGL renderer*

the overhead of the X tokens would limit performance considerably (eg Silicon Graphics workstations).

Direct rendering is possible because a specific order of processing X and GL commands is generally not necessary. If it does, it must be guaranteed by the user through explicit synchronization.

Typical X resource management commands are

```
auxMainLoop(PARAM);   /* register display callback function
auxInitWindow(PARAM);   /* open window with specified characteristics
```

These functions are summarized in the auxiliary library and, in addition to window management, are used to recognize and handle simple user interactions (e.g. mouse and keyboard events). Since OpenGL itself is completely independent of the operating and window system, such an intermediate layer must be introduced in order to be able to write complete graphics programs at all. An alternative to the auxiliary library is *GLUT (OpenGL Utility Toolkit)*, which, in conjunction with the *MUI (Micro User Interface)*, allows for comfortable programming of more complex graphics programs.

Finally, a simple example should be used to illustrate the use of OpenGL and the wrapping in X:

```
/* (c) Copyright 1993, Silicon Graphics, Inc.
* ALL RIGHTS RESERVED */
#include <GL/gl.h>
```

```
#include <GL/glu.h>
#include <stdlib.h>
#include "glaux.h"
/* Initialize material property, light source, and lighting model
*/
void myinit(void) {
GLfloat mat_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
/* mat_specular and mat_shininess are NOT default values */
GLfloat mat_diffuse[] = { 0.4, 0.4, 0.4, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 15.0 };

GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_TEST);
}

void drawPlane(void) {
glBegin(GL_QUADS);
glNormal3f (0.0, 0.0, 1.0);
glVertex3f (-1.0, -1.0, 0.0);
glVertex3f (0.0, -1.0, 0.0);
glVertex3f (0.0, 0.0, 0.0);
glVertex3f (-1.0, 0.0, 0.0);

glNormal3f (0.0, 0.0, 1.0);
glVertex3f (0.0, -1.0, 0.0);
glVertex3f (1.0, -1.0, 0.0);
glVertex3f (1.0, 0.0, 0.0);
glVertex3f (0.0, 0.0, 0.0);
glNormal3f (0.0, 0.0, 1.0);
glVertex3f (0.0, 0.0, 0.0);
glVertex3f (1.0, 0.0, 0.0);
glVertex3f (1.0, 1.0, 0.0);
glVertex3f (0.0, 1.0, 0.0);

glNormal3f (0.0, 0.0, 1.0);
glVertex3f (0.0, 0.0, 0.0);
glVertex3f (0.0, 1.0, 0.0);
glVertex3f (−1.0, 1.0, 0.0);
glVertex3f (−1.0, 0.0, 0.0);
glEnd();
}
void display (void) {
GLfloat infinite_light[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat local_light[] = { 1.0, 1.0, 1.0, 1.0 };

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix();
glTranslatef (−1.5, 0.0, 0.0);
glLightfv(GL_LIGHT0, GL_POSITION, infinite_light);
drawPlane();
glPopMatrix();

glPushMatrix();
glTranslatef (1.5, 0.0, 0.0);
glLightfv(GL_LIGHT0, GL_POSITION, local_light);
drawPlane();
glPopMatrix();
glFlush();
}

void myReshape(int w, int h) {
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho (−1.5, 1.5, −1.5∗(GLdouble)h/(GLdouble)w,
1.5∗(GLdouble)h/(GLdouble)w, −10.0, 10.0);
else
glOrtho (−1.5∗(GLdouble)w/(GLdouble)h,
1.5∗(GLdouble)w/(GLdouble)h, −1.5, 1.5, −10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
}
```

```
/* Main loop
 * Open window with initial window size, RGB display mode
 * and simple input event handling.
 */
int main(int argc, char** argv) {
auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH);
auxInitPosition(0, 0, 500, 200);
auxInitWindow (argv[0]);
myinit();
auxReshapeFunc (myReshape);
auxMainLoop(display);
}
```

Output of the example program above

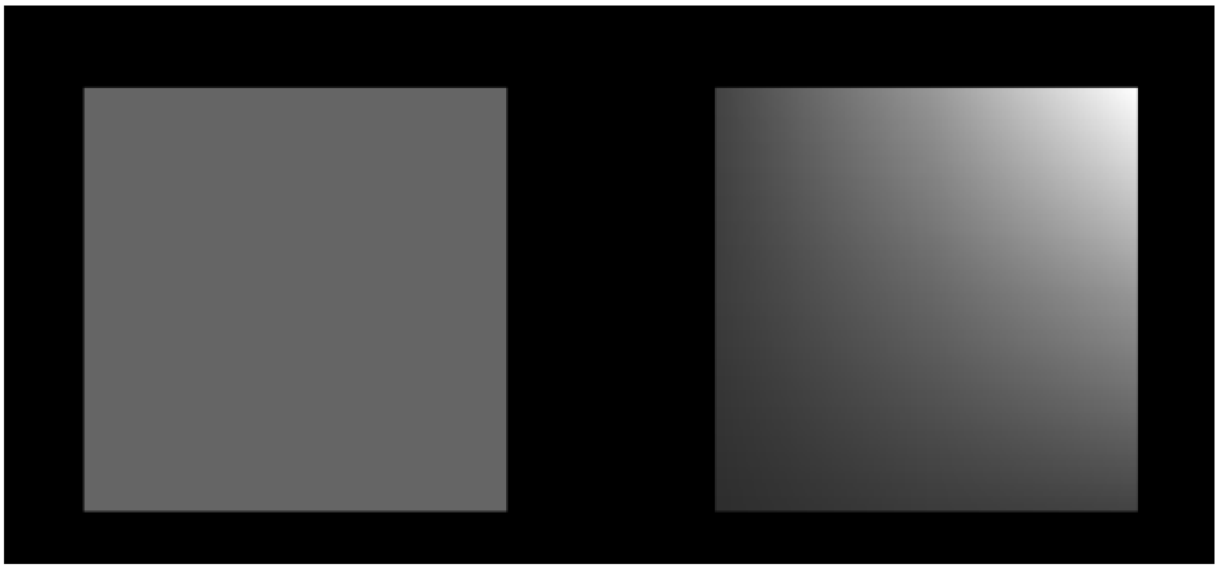


Figure 9.13: *Output of the sample program above*

Recursive Raytracing

10.1 Global Illumination

In the previous considerations, the illumination model was only evaluated locally for each pixel of the image plane to calculate the intensity of a point on the object surface. The algorithm typically used for this is called ray casting. A ray is projected from the origin of the camera coordinate system through the current screen pixel into the scene, all object intersections are calculated and the lighting model is evaluated in the intersection closest to the viewer (see Figure 10.1).

The method has the advantage that the clipping as well as the hidden surface calculations and the scan conversion are solved implicitly. However, global lighting effects such as multiple reflections are only approximated by the ambient term of the lighting model.

In fact, however, light can contribute to the illumination of a point in complex ways through reflections and refractions (Figure 10.2).

A *global illumination model* must therefore be sought that takes such effects into account.

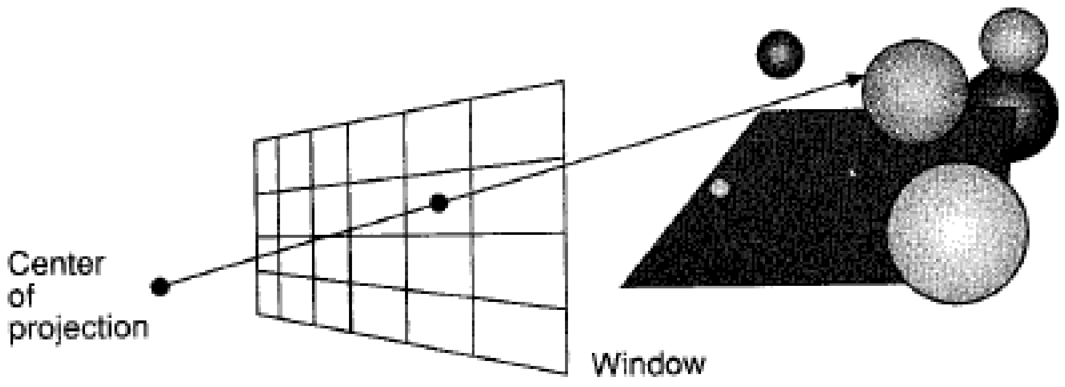


Figure 10.1: *Principle of ray casting*

10.2 Abstract description by Kajiya's rendering equation

The *Rendering-Equation* (10.1) describes the transport of light from a point x' to a point x in space under the influence of the reflection of light from all points x'' of the scene via x' to x.

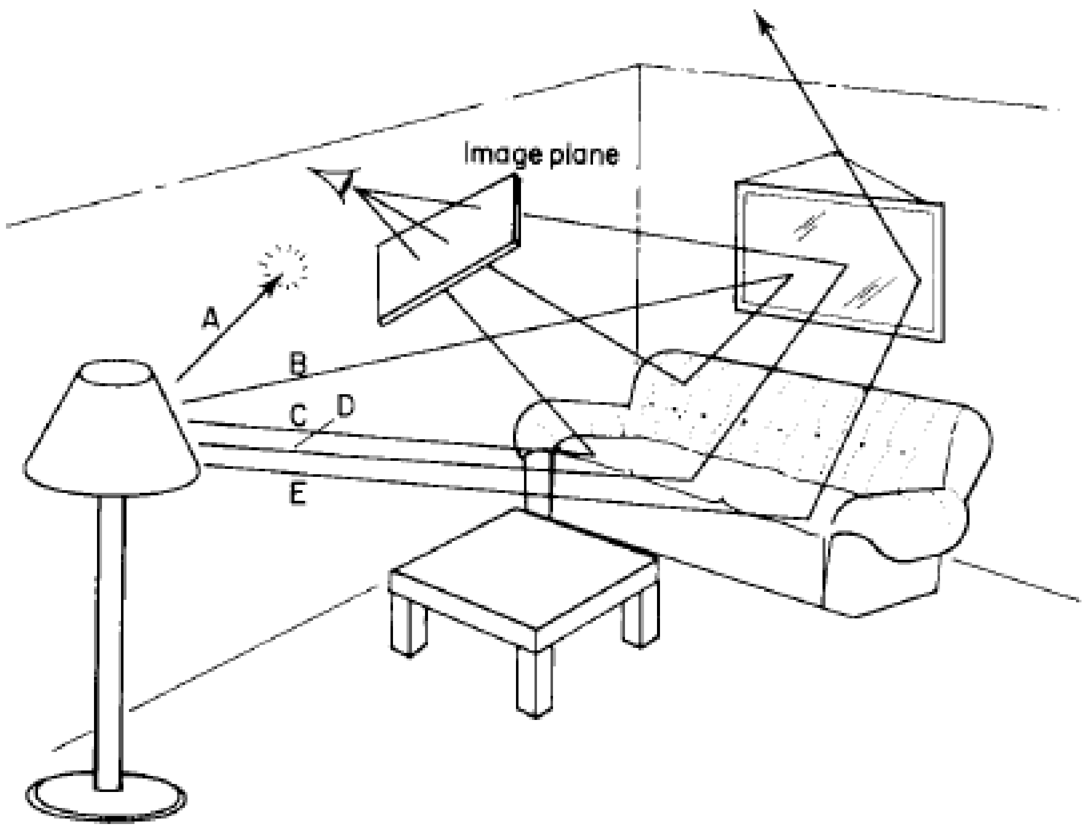


Figure 10.2: *Influence of reflection and refraction on lighting*

$$I(x,x') = g(x,x') \times \left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{10.1}$$

The terms that appear have the following meaning:

$I(x,x')$	Describes the intensity (light flux component) from x' to x
$g(x,x')$	geometry term: $g(x,x') = \begin{cases} 0 & : x \text{ not visible from } x' \\ 1/r^2 & : \text{else} \end{cases}$
$\epsilon(x,x')$	self-emission term
$\rho(x,x',x'')$:	Describes the bidirectional reflection function for specular and diffuse components (BRDF, *bidirectional reflection distribution function*)

The intensity in x results from self-emission from x' in the direction of x, as well as from scattered reflection of the intensities of all points x'' in the direction of x'.

The fundamental relation of Kajiya's Rendering Equation describes an ultimate lighting model to be approximated. A diffuse approximation is achieved using the *radiosity method*, while the recursive *raytracing method*, discussed in detail in the following chapter, approximates the specular effects.

10.3 Recursive Raytracing

To calculate the color and intensity of multiple perfectly reflected and transmitted light, the path from the light source to the eye must be traversed (Figure 10.3). The color of the light entering the eye depends on the object colors seen and on the intensity and color of the incident light.

This results in the following algorithm:

After the first intersection, trace the ray back into the scene to a given recursion depth and evaluate the lighting model at each intersection.

Through the recursion, the reflected and transmitted rays as well as the so-called *shadow rays* build up a binary tree structure (Figure 10.4).

Example Recursive raytracing using the example of the scene in Figure 10.4:

1. Send a ray from the eye through the current pixel in the image plane
2. Ray hits plane 3 $\rightarrow$ Split into a transmitted ray T_1 and a reflected ray R_1

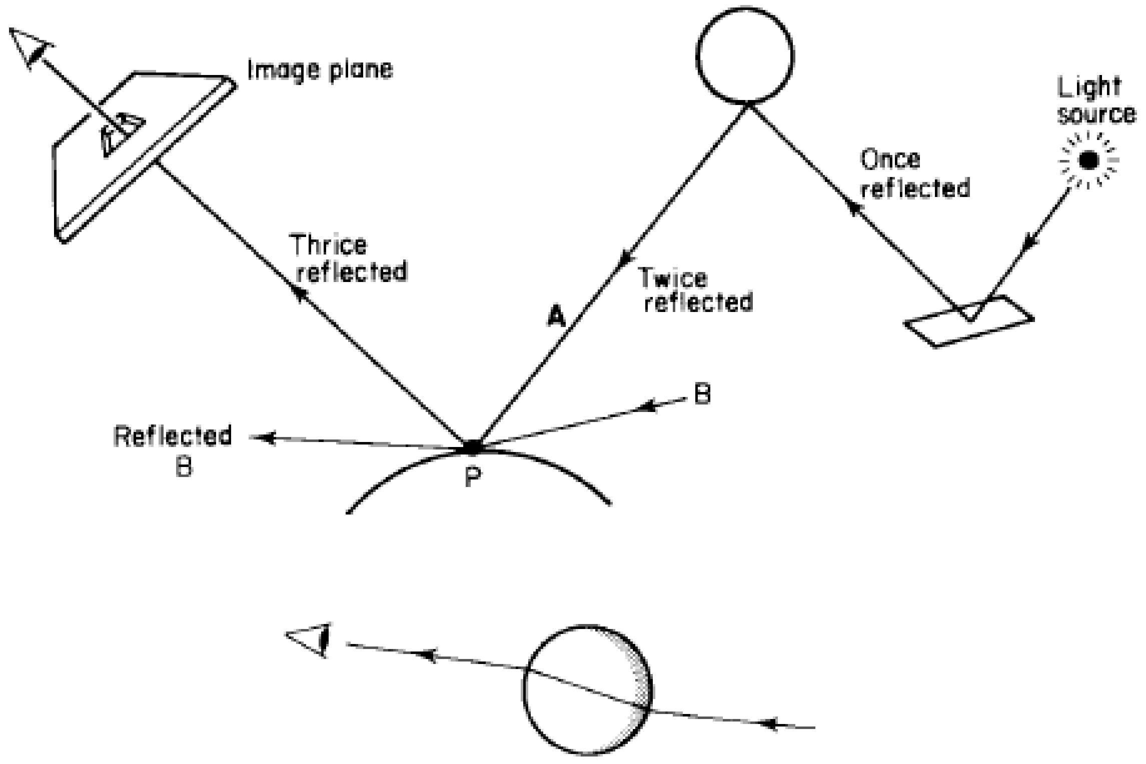


Figure 10.3: *REflection and transmission on the way eye-light source*

3. To calculate the level 3 contribution, a shadow ray is sent to each light source:
   S1 after $L_A \to$ visible
   S2 after $L_B \to$ not visible (section with sphere 4)
4. Evaluation of the local lighting model on level 3 for L_A
5. To calculate the total intensity, calculate the proportions of R_1 and T_1:
   5.1. Track T_1
      a) T_1 hits sphere 6 and is reflected $\to$ ray R_2
      b) Send shadow rays S_3 and S_4 to L_A or L_B (both visible)
      c) Evaluation of the local lighting model on sphere 6 for L_A, L_B
      d) Follow R_2 further $\to$ exits scene $\to$ background color
   5.2. Track R_1
      a) R_1 meets level 9 $\to$ rays R_3 and T_2
      b) Send shadow rays S_5 and S_6 to L_A or L_B (both visible)
      c) Evaluation of the local lighting model on level 9 for L_A, L_B
      d) Follow R_3, T_2 forward $\to$ exit scene $\to$ background color

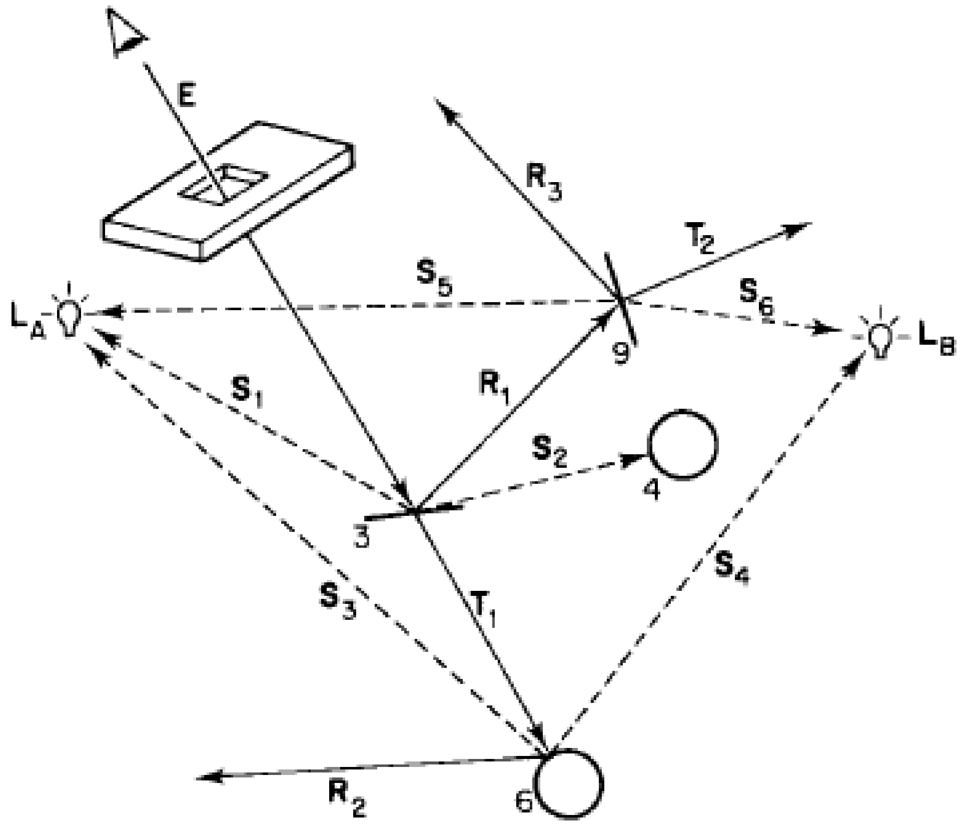


Figure 10.4: *Path of a line of sight in an example scene*

10.3.1 Schematization of the algorithm by ray tree

The nodes of the tree contain object intersections and the local evaluations of the lighting model, while the edges represent the rays. The left branch is the transmitted one, the right one the reflected one.

The illumination is calculated by a *bottom-up traversal* of the tree. The result is a two-stage process of beam generation (intersection calculation) and traversal (evaluation of the lighting model).

10.3.2 Recursive definition of the local lighting model

The traversal of the tree implies a recursive definition of the lighting model. Starting from Phong's illumination model (8.27) one finds the following illumination equation (specular term expressed by halfway vector):

$$I_\lambda^m = I_{a\lambda} k_a O_{d\lambda}^m + \sum s_i^m f_{att_i} I_{p\lambda_i} \left[k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L_i}) + k_s (\mathbf{N} \cdot \mathbf{H_i})^n \right] + k_S I_{r\lambda}^{m+1} + k_t I_{t\lambda}^{m+1} \quad (10.2)$$

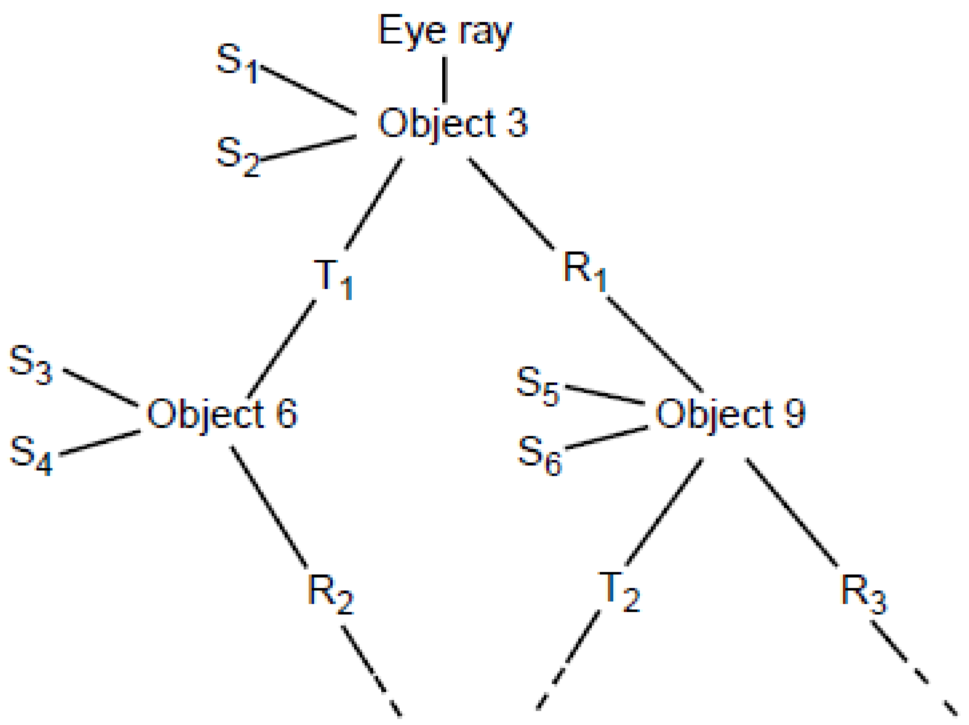


Figure 10.5: *Resulting binary ray tree for the example scene in Figure 10.4*

Intensity of a reflected or transmitted ray I_λ^m in the depth m of the tree results from the intensities of the two sons $I_{r\lambda}^{m+1}$ and $I_{t\lambda}^{m+1}$ weighted with the coefficients k_s and k_t as well as from the lighting model evaluated locally at the intersection. $S_i^{;}m$ describes the result of the shadow ray calculation (delta function: 1 $\rightarrow$ visible; 0 $\rightarrow$ invisible). In addition, the attenuation caused by the accumulated beam length should also be included in the equation (e.g. weighting of I with $1/dis$).

10.3.3 Typical ray tracing algorithm

If necessary, the algorithm from Figure 10.6 generates a reflected and a transmitted beam at each intersection point. It then first wanders through the subtree of the transmitted rays (the right subtree in Figure 10.7). If the end of the subtree is reached - either because there is no longer any transmission or because the maximum recursion depth has been reached - the lighting model is evaluated at this point. For this purpose, the corresponding left subtree is generated first.

A stack is used to manage the beams. Because the stack only ever has to contain a part of the entire ray tree, it never becomes larger than the maximum recursion depth. The generation of reflected or transmitted beams corresponds to a $push$ operation on the stack, while pop operations are required to evaluate the lighting model.

Figure 10.7 shows the course of the algorithm as a traversal of the ray tree, which takes place along the dashed line. Down arrows correspond to $push$ operations, while up arrows represent pop operations.

The following information is located on the stack for each ray:

Ray number:	Unique number for each ray
Ray type:	Beam type: **v:** Line of sight **r:** Reflected ray **p:** Transmitted ray
Ray source number:	Ray number of the ray that created the current ray
Ray source type:	**v,r** or **p** depending on the generating ray
Intersection flag:	1 if an intersection was found for this ray 0, otherwise
Object pointer:	pointer to the object properties of the intersected object in the object description list
Intersection values:	$x-, y-$ and $z-$coordinates of the intersection from which the current ray originates
Direction cosines:	direction of the ray
d:	distance between intersection of current ray and intersection of generating ray (Intersection values)
I_t:	intensity of the transmitted light along the beam
I_s:	Intensity of the specularly reflected light along the beam

The intensity (component *Calculate intensity I* in Figure 10.6 is calculated according to the following formula (like formula (10.2) for monochrome situations)

$$I = k_a I_a + k_d \sum_i S_i I_{I_i}(\mathbf{n} \cdot \mathbf{L_i}) + k_s \sum_i S_i I_{I_i}(\mathbf{S} \cdot \mathbf{R_i})^n + k_s I_s + k_t I_t \tag{10.3}$$

Here, k_a, k_d, and k_s denote the ambient, diffuse, and specular reflection coefficients, while k_t stands for the transmission coefficients. Contrary to earlier formulations, but in accordance with the diagram in Figure 10.8, $\mathbf{S}$ designates the direction of vision and $\mathbf{R_i}$ the reflection vector of the light source i with intensity I_{I_l} . Figure 10.8 explains the procedure.

10.3.4 Adaptive tree depth control

The efficiency of the algorithm can be increased by reducing the average depth of the ray tree and thus the number of intersection calculations. This can be achieved by only generating and inserting into the tree those rays that also have significant impact on the intensity calculations.

In general, it can be said that the contribution of individual rays decreases with increasing iteration depth - on the one hand due to the attenuation increasing with the path length, on the other hand due to the cumulative attenuation due to reflection and transmission. A threshold value for evaluating the significance of the ray contribution can be found by calculating the brightness at the first intersection of the line of sight with a surface, which occurs at the intersection point by evaluating the illumination model. All shadow effects are taken into account, but not reflection and transmission. With each further cut, only the lighting model is evaluated again, this time neglecting the shadow effects. The resulting intensity at the point of intersection is now weighted with the cumulative reflection and transmission coefficients and inversely proportional to the distance traveled. If the result falls below a threshold value, the branch of the ray tree does not need to be pursued any further.

Practical tests have shown that the method leads to an approximately eight-fold increase in speed. A flaw of this method, however, is that a significant contribution to the intensity can occur after ray tracing has already stopped.

10.3.5 Ray box intersection calculation

Since complex objects are often approximated by bounding volumes or kept in octrees to avoid unnecessary intersection tests, the efficient computation of ray box intersections is important.

A simple method for such intersection calculations is based on so-called slabs. A slab is the space between two parallel planes. Bounding volumes can be easily defined with three slabs. To test for a ray's intersection with the bounding volume, one successively computes the intersections with the two planes of each slab, noting the largest near and smallest far intersections. As soon as the largest near intersection is larger than the smallest far intersection, there is no intersection with the bounding volume (compare with the parametric clipping in chapter 5.2.3).

The section tests for slabs whose planes are perpendicular to the main axes are particularly simple (Figure 10.9).

If a box is defined by two points $\mathbf{B_l}$ and $\mathbf{B_h}$ with

$$\begin{aligned}\mathbf{B_l} &= [x_l, y_l, z_l]\\ \mathbf{B_h} &= [x_h, y_h, z_h]\end{aligned}$$

and the ray in a parametric representation by its support point $\mathbf{R_0}$ and a direction vector $\mathbf{R_d}$

$$
\begin{aligned}
\mathbf{R}_{origin} &\equiv \mathbf{R}_0 = [x_0, y_0, z_0] \\
\mathbf{R}_{direction} &\equiv \mathbf{R}_d = [x_d, y_d, z_d] \\
\mathbf{R}(t) &= \mathbf{R}_0 + \mathbf{R}_d \cdot t \qquad t \in \mathfrak{R}_0^+
\end{aligned}
$$

this results in the following algorithm:

```
t_near = – inf;
t_far = inf;

// Shown below for the slab normal to the x−axis

for (all slabs to the main axes x,y and z) {
    if (xd == x0) {
        //ie the ray is parallel to the planes
        if (x0 < x1 || x0 > xh)
            // Origin of ray outside
            return FALSE;
    }
    else {
        // Calculate Intersection Distances Of Planes
        t1 = (xl – x0) / xd;
        t2 = (xh – x0) / xd;

        if (t1 > t2) swap (t1, t2);// swap t1 and t2
        if (t1 > tnear) tnear = t1;
        if (t2 < tfar) tfar = t2;

        if (tnear > tfar)// no cut
            return FALSE

        if (tfar < 0)// box is behind ray, no cut
            return FALSE;
    }
}
return TRUE;
```

this algorithm is a modification of parametric clipping and is therefor very efficient. Only really necessary arithmetic operations are carried out

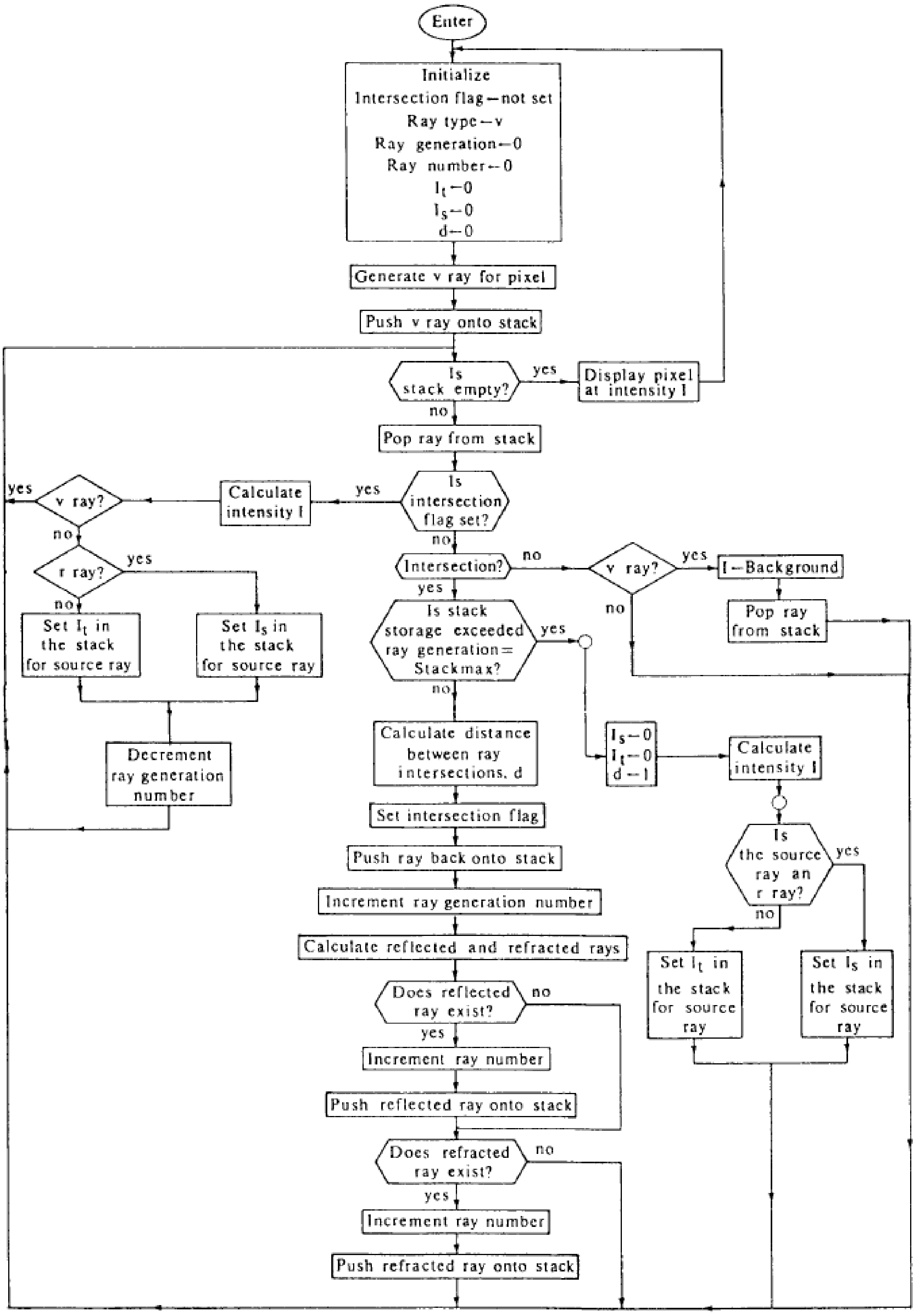

Enter
Initialize
Intersection flag←not set
Ray type←v
Ray generation←0
Ray number←0
I_t←0
I_s←0
d←0
Generate v ray for pixel
Push v ray onto stack
Is stack empty?
yes
Display pixel at intensity I
no
Pop ray from stack
Is intersection flag set?
yes
Calculate intensity I
v ray?
yes
no
r ray?
yes
no
Set I_t in the stack for source ray
Set I_s in the stack for source ray
Decrement ray generation number
no
Intersection?
no
v ray?
yes
I←Background
Pop ray from stack
no
yes
Is stack storage exceeded ray generation = Stackmax?
yes
no
Calculate distance between ray intersections, d
I_s←0
I_t←0
d←1
Calculate intensity I
Is the source ray an r ray?
yes
no
Set I_t in the stack for source ray
Set I_s in the stack for source ray
Set intersection flag
Push ray back onto stack
Increment ray generation number
Calculate reflected and refracted rays
Does reflected ray exist?
no
yes
Increment ray number
Push reflected ray onto stack
Does refracted ray exist?
no
yes
Increment ray number
Push refracted ray onto stack

Figure 10.6: *Flow chart for recursive ray tracing (for the meaning of the nomenclature see the following page)*

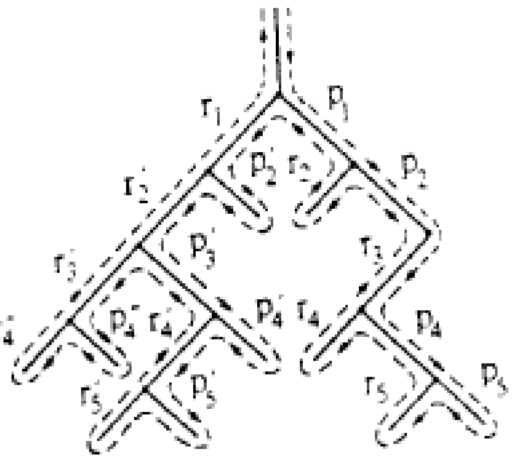


Figure 10.7: *Traversal of the ray tree in the algorithm from Figure 10.6*

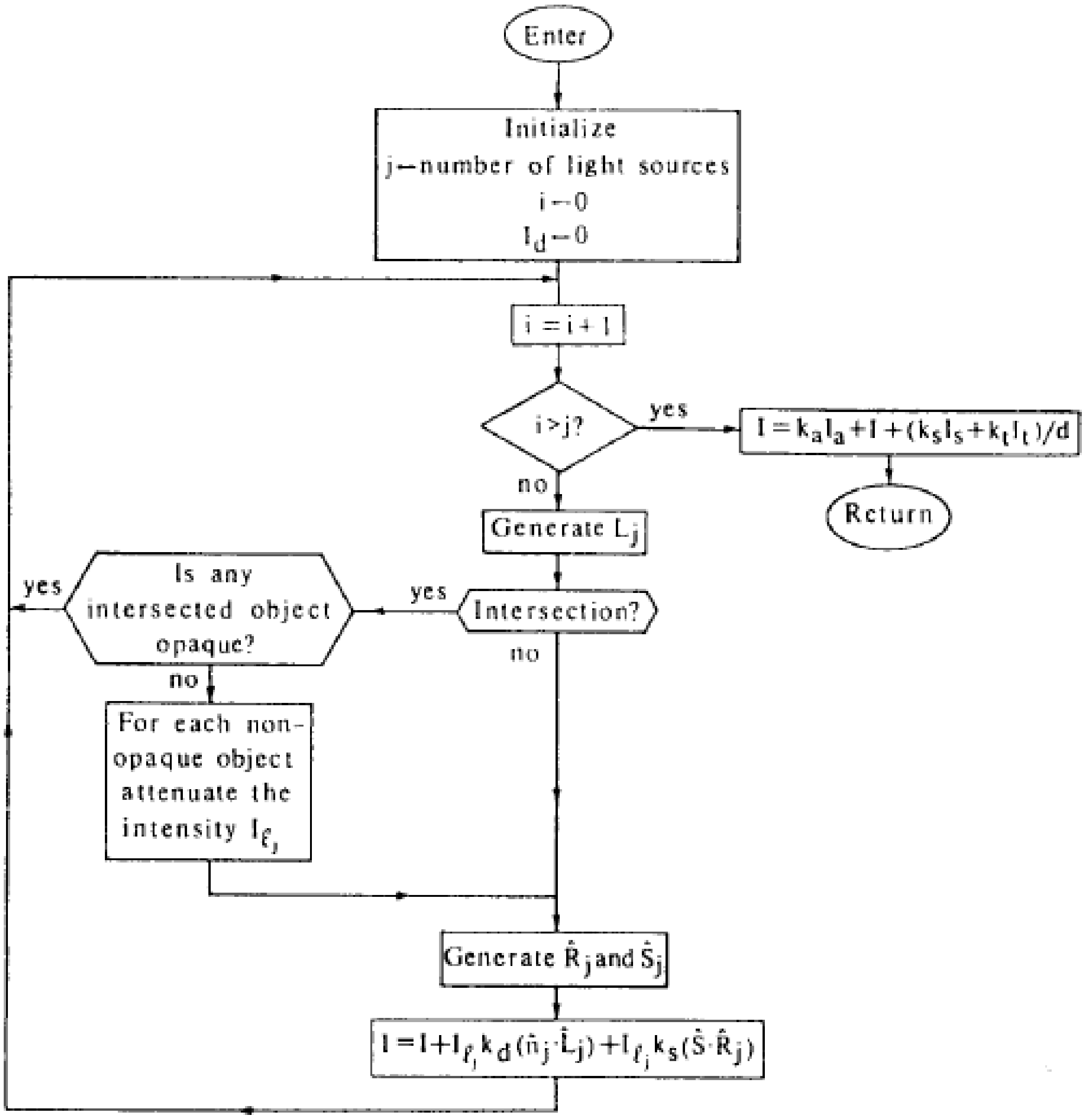

Enter
Initialize
j←number of light sources
i←0
I_d←0
i = i+1
i>j?
yes
$I = k_aI_a + I + (k_sI_s + k_tI_t)/d$
Return
no
Generate L_j
Intersection?
yes
Is any intersected object opaque?
yes
no
For each non-opaque object attenuate the intensity I_{ℓ_j}
no
Generate $\hat{R}_j$ and $\hat{S}_j$
$I = I + I_{\ell_j} k_d(\hat{n}_j \cdot \hat{L}_j) + I_{\ell_j} k_s(\hat{S} \cdot \hat{R}_j)$

Figure 10.8: *Calculation of the intensity at an intersection*

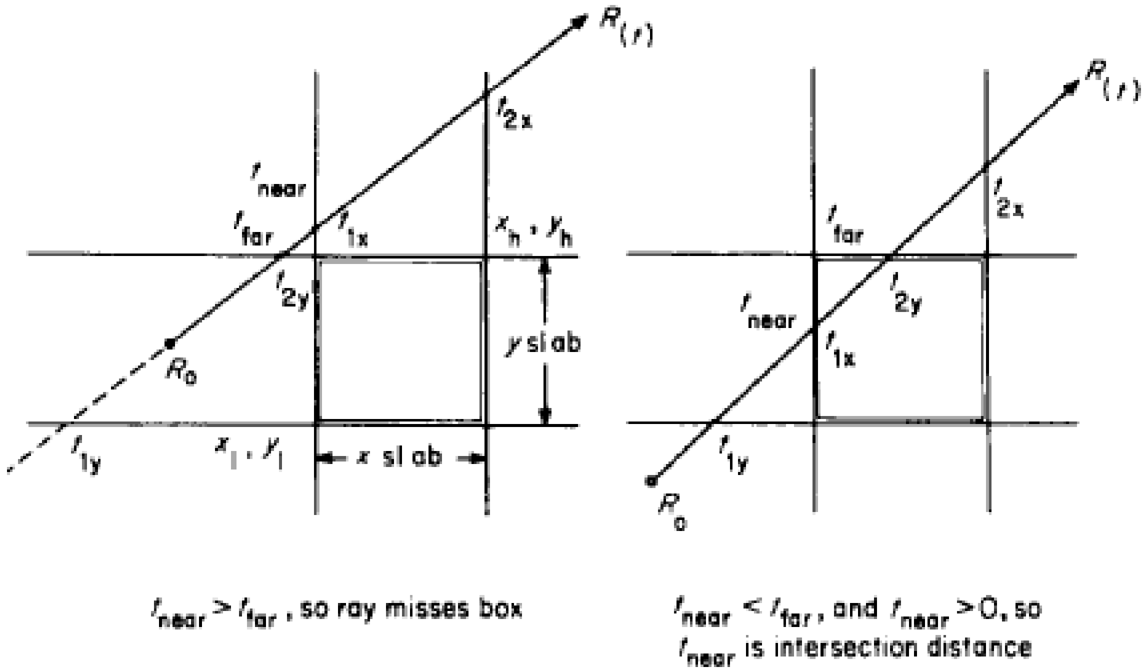


Figure 10.9: *Ray box intersection calculation for slabs perpendicular to the main axes*

Antialiasing

Since the resolution of computer screens is finite and thus all pixels are discrete, image generation is essentially a sampling problem as well. If objects are displayed on the screen whose spatial frequencies are greater than the corresponding Nyquist frequency, aliasing terms arise. These appear in the image in form of cracks at edges (jaggies), Moiré rings in textures, or false contours.

Example:
a) Line with and without antialiasing filter
b) Texture mapping with and without antialiasing filter

To compensate for these issues, methods for filtering high spatial frequencies are required. Before that, however, the essential basic terms and definitions of signal theory are to be recalled below.

11.1 Definitions

11.1.1 Folding

The convolution integral of two functions $f(x)$ and $g(x)$ is given by:

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\alpha) \cdot g(x - \alpha) d\alpha \tag{11.1}$$

In the discrete case for $x = 0, ..., M - 1$, the above equation changes to

$$f_e(x) * g_e(x) = \sum_{m=0}^{M-1} f_e(m) g_e(xm) \tag{11.2}$$

In the same way, the 2D convolution can be described as a separable extension of the one-dimensional case:

$$f(x,y) * g(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\alpha,\beta) g(x-\alpha, y-\beta) d\alpha d\beta \tag{11.3}$$

In the discrete case, $x = 0, .., M - 1$ and $y = 0, .., N - 1$:

$$f_e(x,y) * g_e(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_e(m,n) g_e(xm, yn) \tag{11.4}$$

11.1.2 Fourier Transform

Let $F(u)$ be the 1D Fourier transform

$$\mathfrak{F}\{f(x)\} = F(u) = \int_{-\infty}^{\infty} f(x) e^{-j2\pi ux} dx \tag{11.5}$$

with their inverse

$$\mathfrak{F}^{-1}\{F(u)\} = f(x) = \int_{-\infty}^{\infty} F(u) e^{j2\pi ux} du \tag{11.6}$$

where the Fourier transform is also complex for real functions $f(x)$.

In the discrete case it follows that

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{\frac{j2\pi ux}{N}} \tag{11.7}$$

with $u = 0, .., N - 1$ and the inverse

$$f(x) = \sum_{u=0}^{N-1} F(u) e^{\frac{j2\pi ux}{N}} \tag{11.8}$$

with $x = 0, .., N - 1$.

You also get the separable extensions for 2D

$$\mathfrak{F}\{f(x,y)\} = F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}dxdy \tag{11.9}$$

$$\mathfrak{F}^{-1}\{F(u,v)\} = f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{j2\pi(ux+vy)}dudv \tag{11.10}$$

Example:
2D Fourier transform of a block function

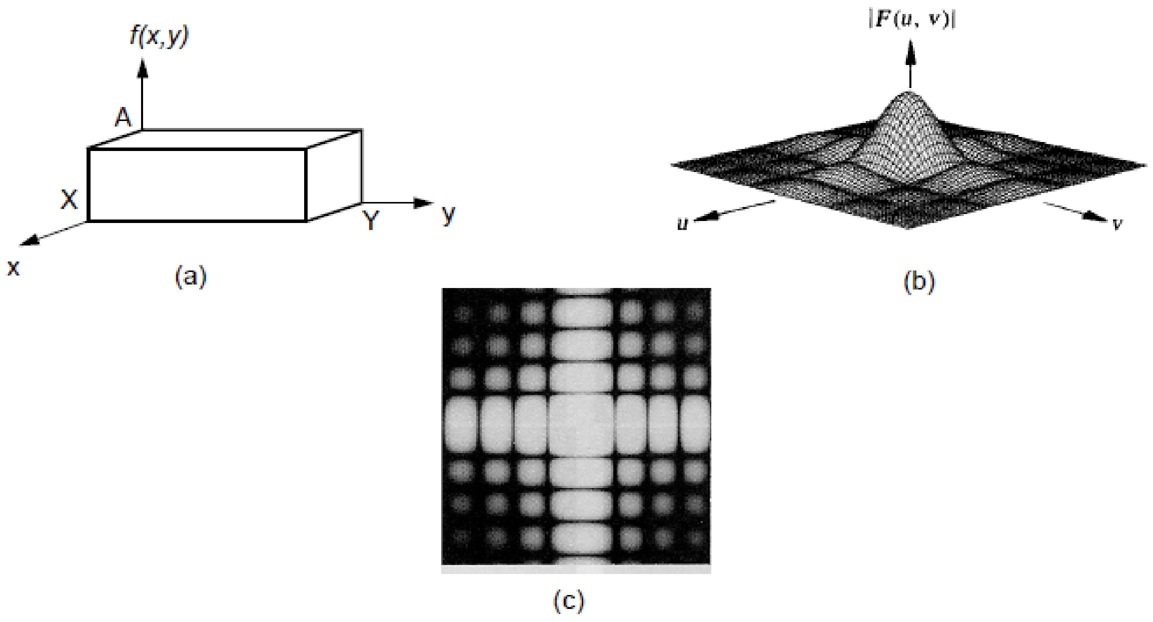


Figure 11.1: *(a) 2D block function*
(b) Corresponding Fourier spectrum
(c) Spectrum shown as intensity distribution (logarithmically scaled)

In the two-dimensional case, the discrete formulation becomes

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{v=0}^{N-1} f(x,y)e^{-j2\pi(\frac{ux}{M}+\frac{vy}{N})} \tag{11.11}$$

for $u = 0, .., M-1$ and $v = 0, .., M-1$ with the inverse

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(\frac{ux}{M}+\frac{vy}{N})} \tag{11.12}$$

Here, u and v are also called spatial frequencies.

11.1.3 Elementary Relations

The so-called *Parseval energy equivalent* exists between the two spaces:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}|f(x,y)|^2dxdy = \frac{1}{4\pi^2}\int_{-\ infty}^{\infty}\int_{-\infty}^{\infty}|F(u,v)|^2dudv \tag{11.13}$$

and the Heisenberg relation for the resolutions Δx and Δu in the spatial and frequency domain

$$\Delta x \cdot \Delta u \geq \frac{1}{4\pi} \tag{11.14}$$

According to the theory of linear, time-invariant systems (LTI), the filtering of a signal $f(x)$ with an impulse response $g(x)$ results from convolution in the spatial domain. This corresponds to a multiplication in the frequency domain:

$$f(x) * g(x) \equiv F(u)G(u) \tag{11.15}$$

Prerequisites for this are functions of finite energy, $f(x) \in L^2(\Re)$ or $f(x,y) \in L^2(\Re^2)$, i.e., *square-integrable functions* :

$$\int_{-\infty}^{\infty}|f(x)|^2dx < \infty \tag{11.16}$$

11.2 Sampling

11.2.1 Sampling of One-Dimensional Functions

The *sampling (sampling, discretization)* of a signal results from multiplication by an impulse comb, where the *delta distributions* $\delta(x - x_0)$ are defined as follows:

$$\int_{-\infty}^{\infty} f(x)\delta(x - x_0)dx = f(x_0) \tag{11.17}$$

The *sampling theorem* states that the maximum distance between two samples Δx for a band-limited function $f(x)$ with an upper limit frequency W is given by:

$$\Delta x \leq \frac{1}{2W} \tag{11.18}$$

The signal can only be perfectly interpolated using a reconstruction filter $G(u)$ if this condition is met.

The corresponding frequency

$$\frac{1}{\Delta x} = 2W \tag{11.19}$$

is called *Nyquist frequency* and denotes the minimum required sampling rate.

If the sampling rate $1/\Delta x$ is not high enough, overlaps occur in the spectrum. This prevents a perfect reconstruction, which is noticeable in the form of aliasing artefacts (Fig. 11.2).

Figures (a) to (f) in Fig. 11.3 correspond to those in Fig. 11.2 with the difference that the sampling rate is sufficiently high to avoid aliasing effects.

The previous results are based on functions of unlimited duration in the spatial domain. Since this also implies an unlimited sampling time, sampling finite signals is only considered in practice. Sampling in a finite interval can be represented mathematically by multiplying the sampled result (figure (e) in Fig. 11.3) by a so-called *window*. A window is the rectangle function

$$h(x) = \begin{cases} 1 & 0 \leq x \leq X \\ 0 & \text{else} \end{cases} \tag{11.20}$$

Figures (g) and (h) in Fig. 11.3 show the window function and its Fourier transform, (i) and (j) the results after multiplying or convolving the signal with the window.

The convolution of the function $S(u) * F(u)$ with $H(u)$ (Fig. 11.3 (j)) produces so-called *rippling effects*, which are due to the fact that $h(x)$ is a non-band-limited function. Therefore, even if the conditions of the sampling theorem are satisfied, it is impossible to reconstruct a function without error if it is only sampled over a limited interval.

So far, all results in the frequency domain have been of a continuous nature, regardless of whether discrete or continuous functions have been considered in the spatial domain. In order to obtain a discrete Fourier transform, the continuous transform must be "sampled" with a pulse comb whose pulses are Δu units apart. Fig. 11.4 illustrates the situation based on the results (i) and (j) from Fig. 11.3.

The sampling of the Fourier transform $F(u)$ corresponds to a multiplication with the impulse comb $S(u)$. This results in the discrete Fourier transform in Fig. 11.4 (f). The corresponding operation in the spatial domain is a convolution. The corresponding result is shown in figure (e). It is easy to see that this function has a period of $1/\Delta u$. The discrete Fourier transformation can therefore be understood as a periodization of the signal in the spatial domain.

If $f(x)$ and $F(u)$ are each sampled at N equidistant points, so that exactly one period is covered by the N points both in the spatial and in the frequency domain, then *applies in the spatial domain* $N\Delta x = X$ and in frequency space $N\Delta u = 1/\Delta x$. From this, the elementary relationship

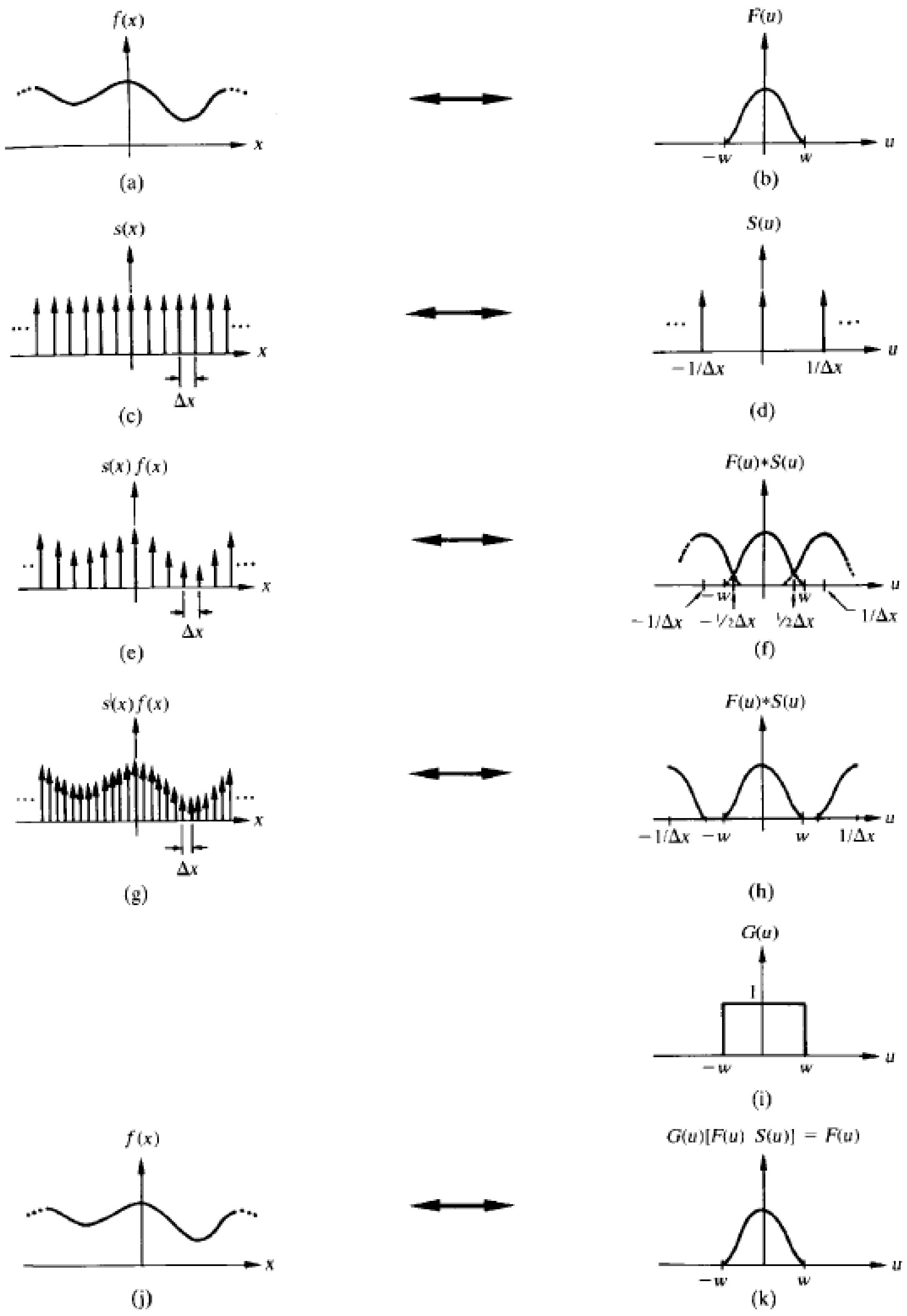


Figure 11.2: *Illustration of sampling and aliasing (from the complex-valued Fourier transform, the magnitude is plotted)*

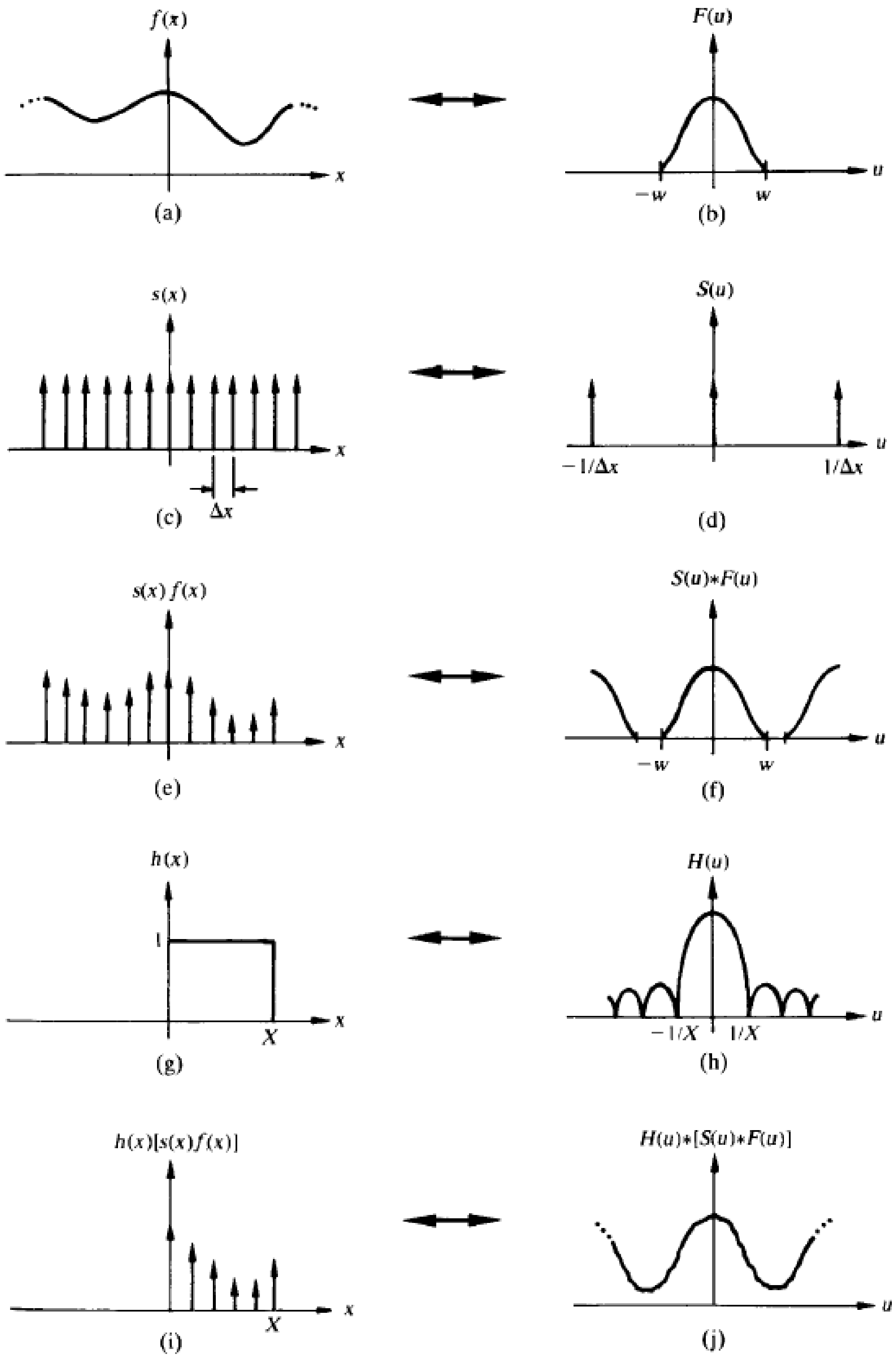


Figure 11.3: *Illustration of the sampling of time-limited signals*

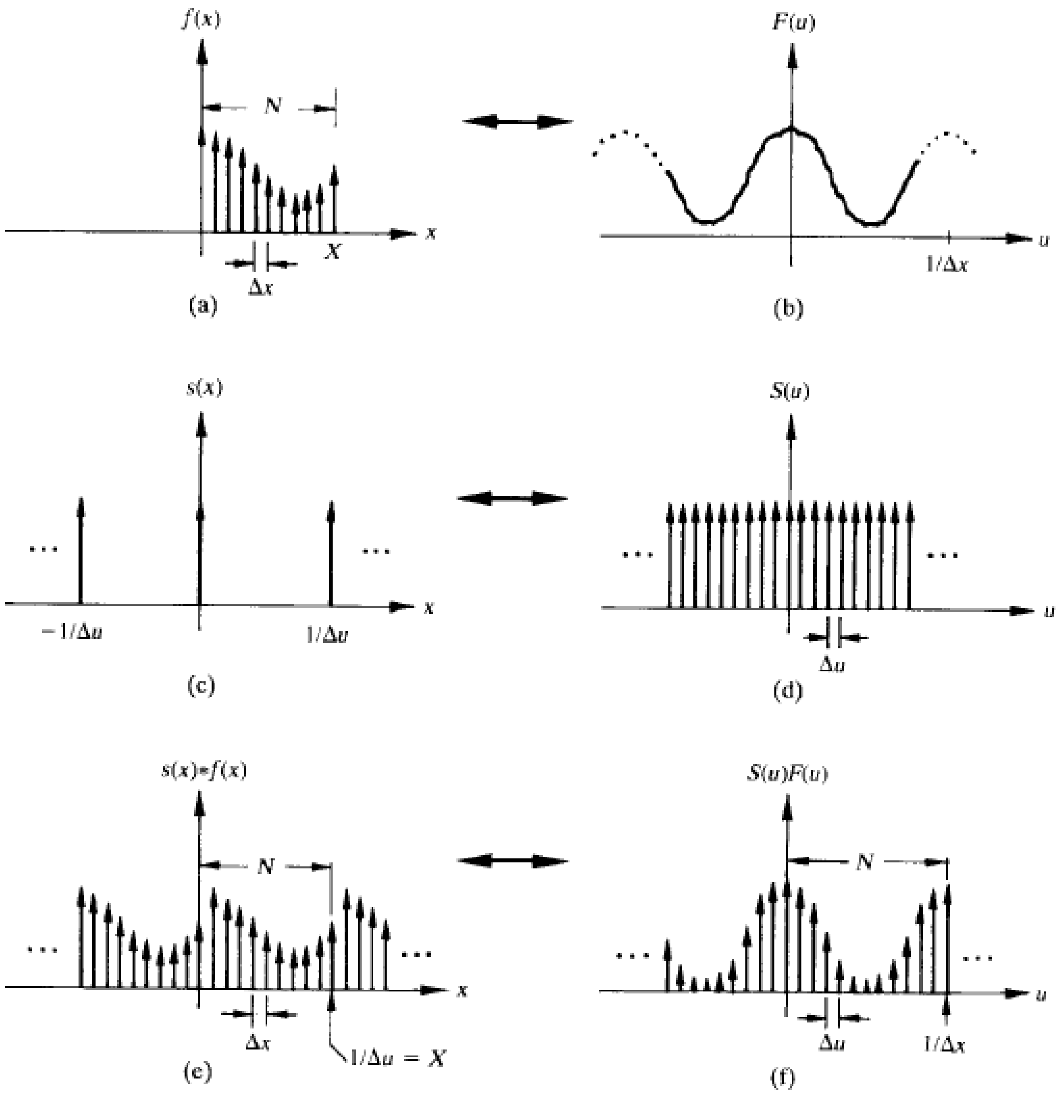


Figure 11.4: *Illustration of the Discrete Fourier Transform*

between the resolutions Δx and Δu follows directly:

$$\Delta u = \frac{1}{N \cdot \Delta x} \tag{11.21}$$

11.2.2 Sampling of Two-dimensional Functions

The 2D expansion of the δ distributions is given as follows:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\delta(x-x_0, y-y_0)dxdy = f(x_0,y_0) \tag{11.22}$$

The one-dimensional momentum combs are extended to momentum fields $s(x,y)$ with sampling distances Δx resp. Δy (Fig. 11.5).

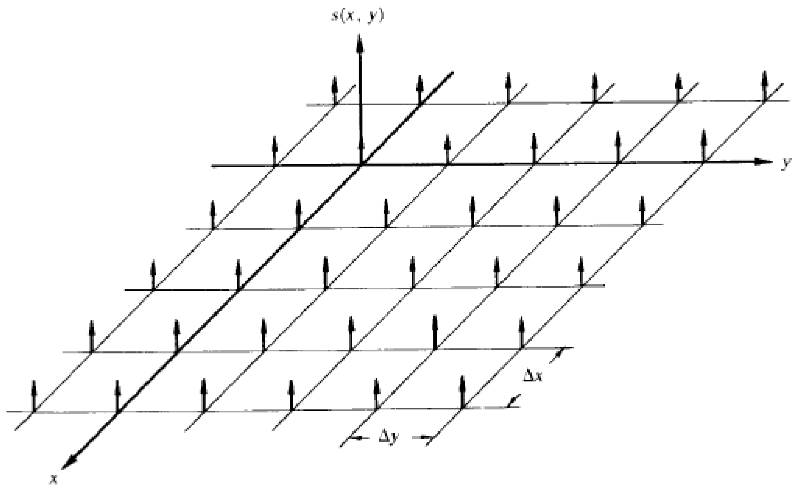


Figure 11.5: *2D momentum combs*

A function $f(x, y)$ is sampled by multiplying it with an impulse field $s(x, y) \cdot f(x, y)$. The corresponding operation in frequency space is the convolution $S(u, v) * F(u, v)$, where $S(u, v)$ is a momentum field with resolutions $1/\Delta x$ and $1/\Delta y$ in u- resp. v direction. For a band-limited function $f(x, y)$ the result of this convolution could look something like Fig. 11.6.

The function $f(x, y)$ can be reconstructed from $S(u, v) * F(u, v)$ by multiplying it with the interpolation filter (low pass) $G(u, v)$

$$f(x, y) = G(u, v)[S(u, v) * F(u, v)] \tag{11.23}$$

where

$$G(u, v) = \begin{cases} 1 & (u, v) \text{ inside the bounding box of R} \\ 0 & \text{else} \end{cases}$$

if the conditions $1/\Delta x > 2W_u$ and $1/\Delta y > 2W_v$ (no aliasing) are satisfied. Here, $2W_u$ and $2W_v$ denote the bandwidth in the u and v direction, respectively, and thus the localization of the function in the frequency domain.

From this, the 2D version of the sampling theorem follows:

$$\begin{aligned} \Delta x &<= \frac{1}{2W_u} \\ \Delta y &<= \frac{1}{2W_v} \end{aligned} \tag{11.24}$$

If the discrete 2D Fourier transformation $f(x, y)$ is spatially limited by a 2D window function $h(x, y)$ (analogous to the 1D window function $h(x)$ in Fig. 11.3), then the convolution

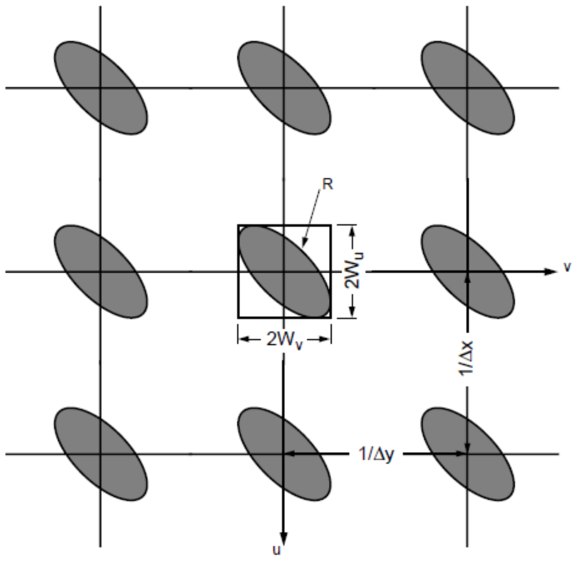


Figure 11.6: Representation of a sampled, band-limited 2D function in frequency space

$H(u, v) * [S(u, v) * F(u, v)]$ also produces distortions for the transformation of the sampled function. Analogous to the one-dimensional case, periodic functions are the exception.

Using methods similar to the 1D case, the following conditions are found on the sampling resolutions, with a complete 2D period being covered by $N\mu^1 \times N$ equidistant values in both the spatial and frequency domain:

$$\begin{aligned} \Delta u &<= \frac{1}{N\Delta x} \\ \Delta v &<= \frac{1}{N\Delta y} \end{aligned} \tag{11.25}$$

11.3 Antialiasing Methods

11.3.1 Band Limitation through Filtering

In the simplest case, the band of a discrete image can be limited by post-filtering using various filter functions. The image function is band-limited and the aliasing artefacts can be eliminated.

An important design criterion for a filter is its frequency response, i.e., linearity in the pass band (*pass band*), slope in the transition band (*transition band*) and attenuation in the stop band (*stop band*) (Fig. 11.7).

The simplest filters are so-called *B-spline filters*, whose impulse responses are generated from a box filter by repeated convolution with itself. You start with the box filter (B-spline filter of

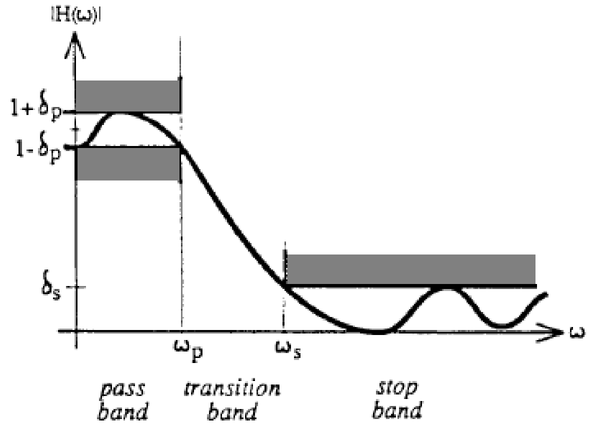


Figure 11.7: *Design criteria for interpolation filters*

the first order)

$$g_1(x) = \begin{cases} 1 & |x| \leq 1/2 \\ 0 & |x| > 1/2 \end{cases} \leftrightarrow \frac{\sin \omega/2}{\omega/2} = \frac{\sin \pi f}{\pi f} = \operatorname{sinc} f \tag{11.26}$$

with $f = \omega/2\pi$ as rotation frequency. The frequency response of this filter is a sinc function. The nth-order B-spline filters now result from $(n-1)$-fold successive convolution:

$$g_n(x) = g_1(x) * g_1(x) * ... * g_1(x) \leftrightarrow \operatorname{sinc}^n f \tag{11.27}$$

The higher the order of the filter, the faster it decays in the frequency domain. The Fourier transform of such filters always consists of a polynomial of sinc functions. The first two B-spline filters are shown in Fig. 11.8 (above).

Another popular filter is the *Gaussian filter* with the impulse response

$$g_{\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2} \leftrightarrow G_{\sigma^2}(\omega) = e^{-\sigma^2\omega^2/2} = \frac{\sqrt{2\pi}}{\sigma} g_{1/\sigma^2}(\omega) \tag{11.28}$$

where the B-spline filter for $n \to \infty$ converges to a Gaussian filter (Fig. 11.8, middle).

The so-called ideal low-pass filter consists of a first-order sinc *filter* of the form

$$\operatorname{sinc}\left(\frac{\omega_c x}{\pi}\right) = \frac{\sin(\omega_c x)}{\pi x} \leftrightarrow g_1\left(\frac{\omega}{2\omega_c}\right) = \begin{cases} 1 & |\omega| \leq \omega_C \\ 0 & |\omega| > \omega_c \end{cases} \tag{11.29}$$

The sinc filter is shown in Fig. 11.8 (below). Due to its infinite impulse response, this type of filter is also called *non-causal.*

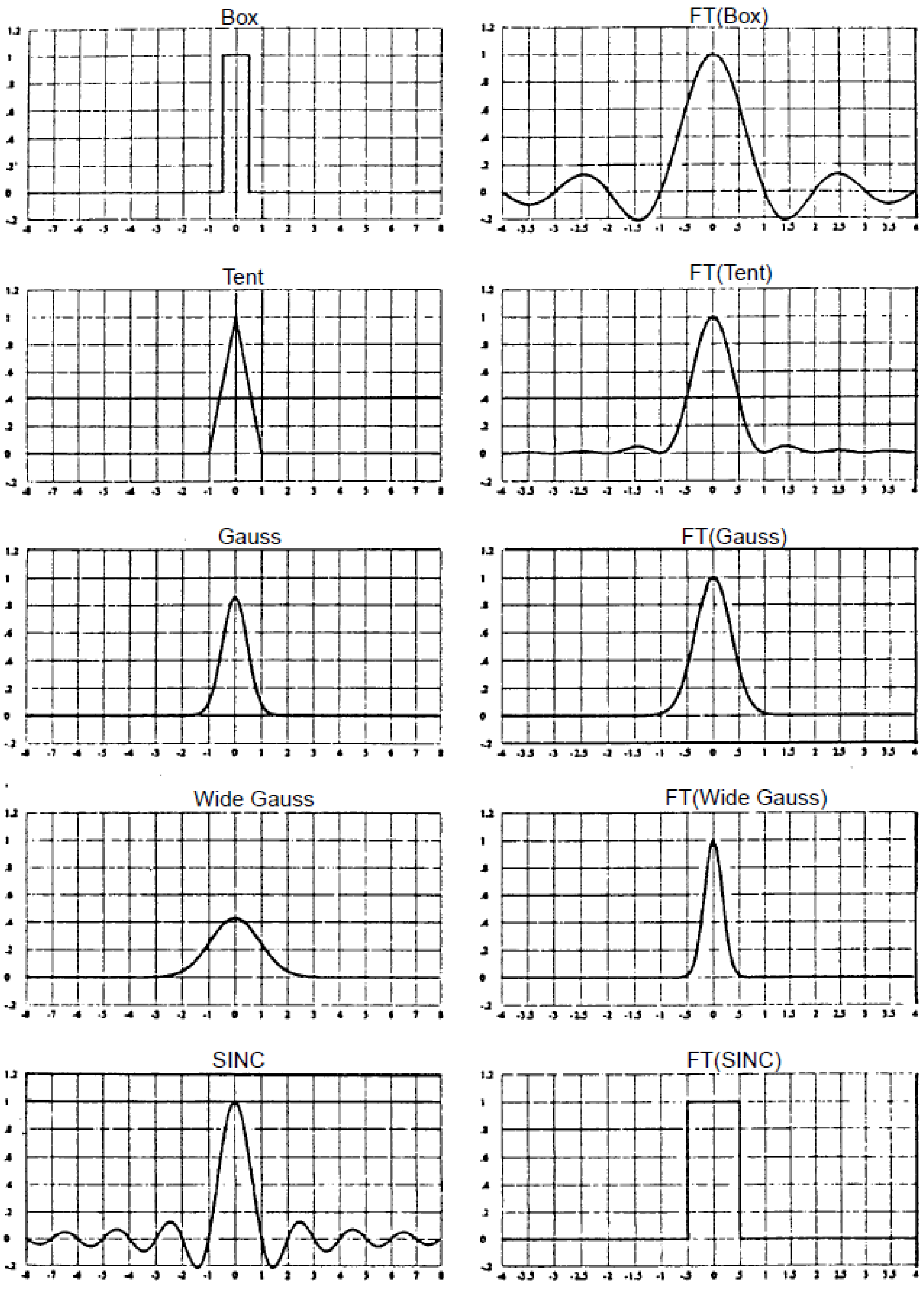


Figure 11.8: *Different 1D filters and their Fourier transforms*

The 2D extension of the impulse responses results from the tensor products of the convolution kernel

$$g(x,y) = g(x) \times g(y) \rightarrow g(n,m) = g(n) \times g(m) \tag{11.30}$$

Bilinear interpolation corresponds exactly to second-order 2D B-spline filtering.

Filters can always be viewed as interpolators for the reconstruction of discrete signals. In practice, a discrete 2D filter kernel is generated and convolved over the image.

Example:
Various discrete 2D filter cores:

$$\begin{array}{ccc}
\text{Gauss} & \text{Box} & \text{Bilinear} \\
\begin{bmatrix}
1 & 4 & 8 & 10 & 8 & 4 & 1 \\
4 & 12 & 25 & 29 & 25 & 12 & 4 \\
8 & 25 & 49 & 58 & 49 & 25 & 8 \\
10 & 29 & 58 & 67 & 58 & 29 & 10 \\
8 & 25 & 49 & 58 & 49 & 25 & 8 \\
4 & 12 & 25 & 29 & 25 & 12 & 4 \\
1 & 4 & 8 & 10 & 8 & 4 & 1
\end{bmatrix} &
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix} &
\begin{bmatrix}
1 & 2 & 3 & 4 & 3 & 2 & 1 \\
2 & 4 & 6 & 8 & 6 & 4 & 2 \\
3 & 6 & 9 & 12 & 9 & 6 & 3 \\
4 & 8 & 12 & 16 & 12 & 8 & 4 \\
3 & 6 & 9 & 12 & 9 & 6 & 3 \\
2 & 4 & 6 & 8 & 6 & 4 & 2 \\
1 & 2 & 3 & 4 & 3 & 2 & 1
\end{bmatrix}
\end{array}$$

One disadvantage of filter operations is the loss of image sharpness because high spatial frequencies (sharp edges) are removed.

11.3.2 Filtering of Textures

When using texture mapping, it becomes clear that interpolation is required to reconstruct the continuous texture signal. The filters described above can be used to achieve this. However, the following should be noted:

1 The entire theory is based on band-limited signals. However, this is not always the case with textures.
Example: checkerboard pattern in the plane, viewed in perspective.

2 The mapping process maps the screen's regular sampling grid to an irregular grid in the texture's parameter space.

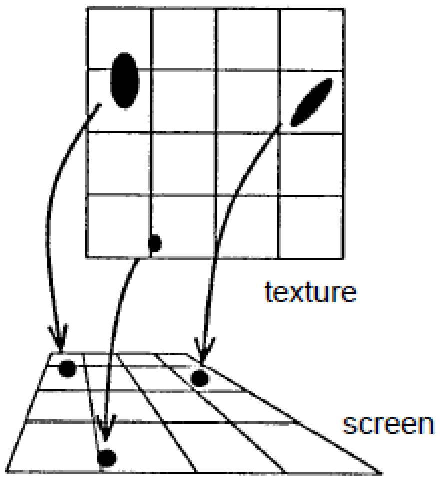


Figure 11.9: *Same shapes on screen correspond to different shapes in texture space*

In case 1, the aliasing can only be reduced at the expense of the resolution. In case 2, affine mappings into the texture space generate spatially variant filter kernels, which, for example, allow isotropic, circular filter kernels to degenerate into ellipses of different alignment and length (Fig. 11.9). Therefore, spatially invariant filtering in the texture space by bilinear interpolation is subject to error. This can be remedied by location-variant filters (Fig. 11.10).

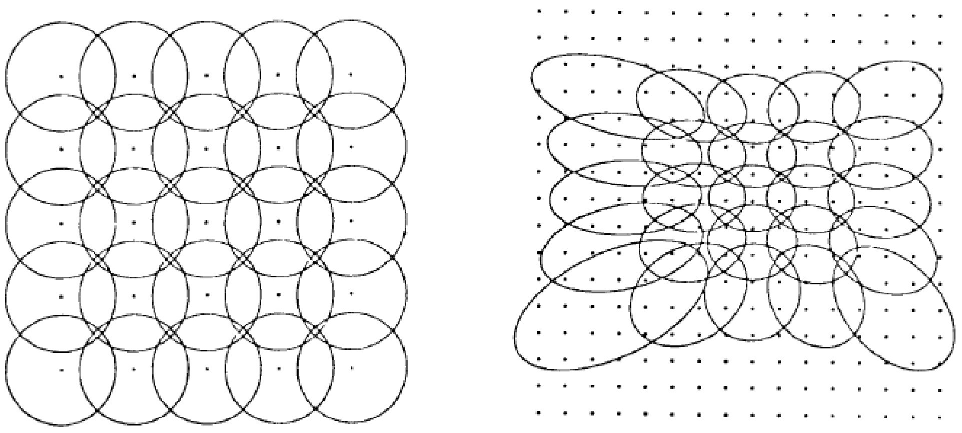


Figure 11.10: *Local variant filter: Circular region in screen space (left) are mapped to elliptical region in texture space. These regions differ in size, eccentricity and orientation. Dots mark pixel centers.*

11.3.3 Raytracing Supersampling Methods

Supersampling is an important method for reducing aliasing artifacts in raytracing. Several rays are shot into the scene per pixel and the pixel color is determined by averaging. This corresponds to sampling the signal at a higher sampling rate and band limiting it exactly to the resolution of the screen. Therefore, in contrast to post-filtering in 2D, the resolution of the image is not artificially reduced and the image is therefore not blurred.

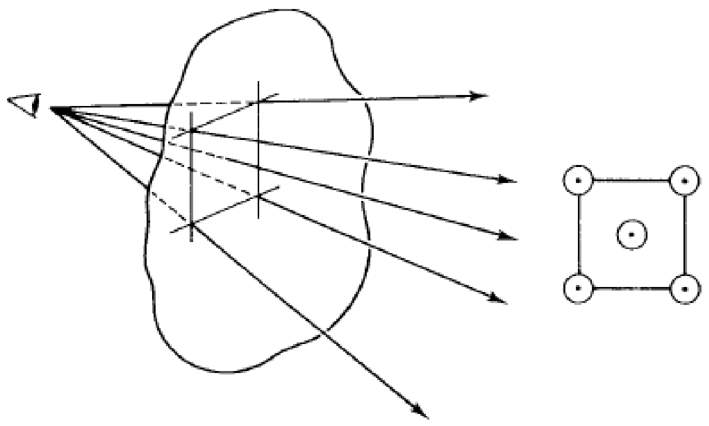


Figure 11.11: *Example of super sampling: Four corner rays and one center ray are traced for each pixel*

A disadvantage of this method is the enormous additional computation time. For this reason, adaptive supersampling methods have been developed.

11.3.4 Adaptive Supersampling

With *adaptive supersampling*, a total of five rays through the pixel corner points and through the pixel center are first calculated. If they differ sufficiently with regard to the calculated intensities, the four segments are further subdivided. This builds a quadtree for the pixel area (Fig. 11.12).

The final color is computed by bottom-up traversal of the quadtree. The color is recursively defined for each quadrant by averaging over the children.

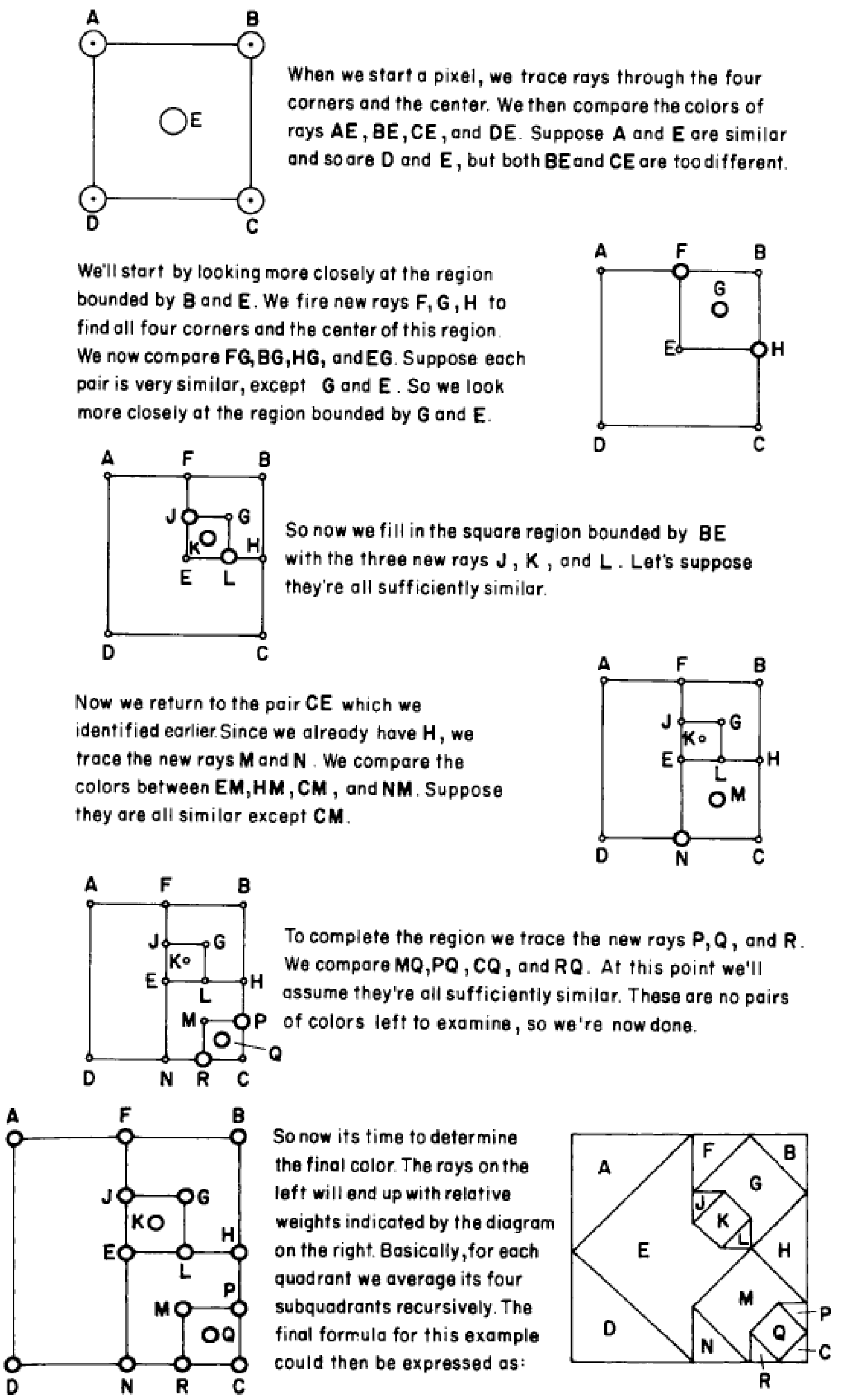
11.3.5 Stochastic Supersampling

A problem with regular sampling grids is the occurrence of *leakage effects*. For example, a sine function can be sampled exactly in the zeros.

One way of reducing aliasing even with a constant image resolution (sampling rate) is to add a stochastic sampling process. The cut-off frequency of the signal is reduced in favor of noise. Part of this noise can be filtered out again by subsequent filtering to the Nyquist frequency. The most important processes are Poisson sampling and jittering.

Poisson Sampling

In the one-dimensional case, the sampling can be in the form of a pulse comb



When we start a pixel, we trace rays through the four corners and the center. We then compare the colors of rays **AE**, **BE**, **CE**, and **DE**. Suppose **A** and **E** are similar and so are **D** and **E**, but both **BE** and **CE** are too different.

We'll start by looking more closely at the region bounded by **B** and **E**. We fire new rays **F**, **G**, **H** to find all four corners and the center of this region. We now compare **FG**, **BG**, **HG**, and **EG**. Suppose each pair is very similar, except **G** and **E**. So we look more closely at the region bounded by **G** and **E**.

So now we fill in the square region bounded by **BE** with the three new rays **J**, **K**, and **L**. Let's suppose they're all sufficiently similar.

Now we return to the pair **CE** which we identified earlier. Since we already have **H**, we trace the new rays **M** and **N**. We compare the colors between **EM**, **HM**, **CM**, and **NM**. Suppose they are all similar except **CM**.

To complete the region we trace the new rays **P**, **Q**, and **R**. We compare **MQ**, **PQ**, **CQ**, and **RQ**. At this point we'll assume they're all sufficiently similar. These are no pairs of colors left to examine, so we're now done.

So now its time to determine the final color. The rays on the left will end up with relative weights indicated by the diagram on the right. Basically, for each quadrant we average its four subquadrants recursively. The final formula for this example could then be expressed as:

$$\frac{1}{4}\left(\frac{A+E}{2}+\frac{D+E}{2}+\frac{1}{4}\left[\frac{F+G}{2}+\frac{B+G}{2}+\frac{H+G}{2}+\frac{1}{4}\left\{\frac{J+K}{2}+\frac{G+K}{2}+\frac{L+K}{2}+\frac{E+K}{2}\right\}\right]\right.$$

$$\left.+\frac{1}{4}\left[\frac{E+M}{2}+\frac{H+M}{2}+\frac{N+M}{2}+\frac{1}{4}\left\{\frac{M+Q}{2}+\frac{P+Q}{2}+\frac{C+Q}{2}+\frac{R+Q}{2}\right\}\right]\right)$$

Figure 11.12: *Example of adaptive super sampling*

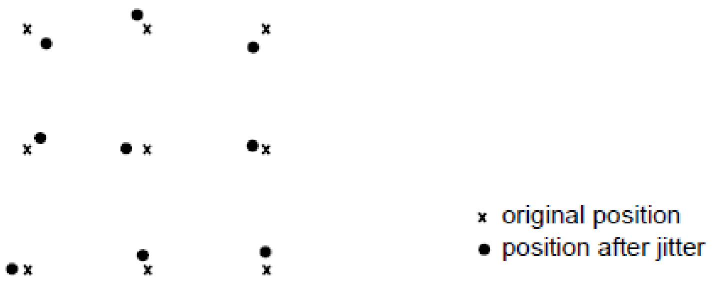


Figure 11.13: *Jittering: Each sampling point is "jiggled" by two uncorrelated random values in the x and y direction*

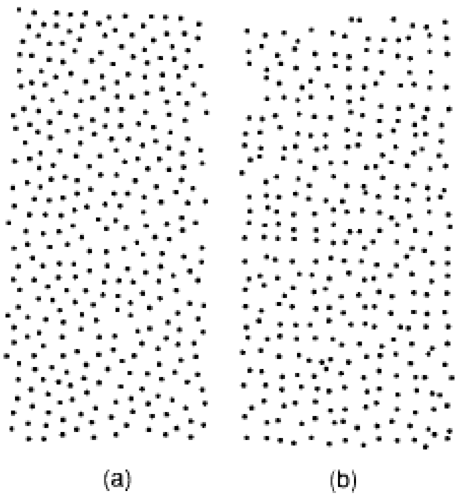


Figure 11.14: *Poisson samples (a) versus jittering (b)*

$$s(x) = \sum_{k=0}^{K-1} \delta(x - x_k) \tag{11.31}$$

The x_k have the distribution

$$p(x_k) = \begin{cases} 1/X & 0 < x_k < X \\ 0 & \text{else} \end{cases} \tag{11.32}$$

where $K = \beta \cdot X$, with the sampling rate $\beta \in [0, .., X]$, is the number of distributions in the sampling interval. The expected value of the square mean results in:

$$E\left[|S_X(u)|^2\right] = \begin{cases} K^2 = \beta^2 \cdot X^2 & u = 0 \\ K = \beta \cdot X & u \neq 0 \end{cases} \tag{11.33}$$

The 2D amplitude spectrum of the sampling process results in a single delta distribution surrounded by noise. The impulse comb is thus only transformed into a single impulse in the spectrum and aliasing can therefore no longer occur. The delta functions required for periodizing the spectrum are missing. However, a noise term is added.

The result is calculated by spectral estimation, where the *spectral power density distribution* of the sampling process is determined by

$$\Phi_s(u) = \lim_{X\to\infty} \frac{|S_X(u)|^2}{X} = \beta + 2\pi\beta^2\delta(u) \tag{11.34}$$

is given.

With a stationary 1D function $f(x)$, which is statistically independent of $s(x)$, the spectral power density of the sampled signal results in $g(x) = f(x) \cdot s(x)$ through

$$\Phi_g(u) = \Phi_f(u) * \Phi_s(u) = \beta \int_R \Phi_f(u)du + 2\pi\beta^2\Phi_f(u) \tag{11.35}$$

The first term describes a noise process, the second the scaled spectrum of the original image. This means that the image signal is reduced in favor of (white) broadband noise.

Another generalization is the *minimum distance* Poisson sampling with

$$x_{k+1} = x_k + l_k \tag{11.36}$$

and l_k exponentially distributed

$$l_k \sim p(l_k) = \begin{cases} \beta \cdot e^{-\beta_y(l_k - l_{k_0})} & l_k > l_{k_0} \\ 0 & \text{else} \end{cases} \tag{11.37}$$

The value l_{k_0} denotes the minimum distance between the samples. For $l_{k_0} = 0$, there is Poisson sampling with $\beta = \beta_y$. For $\beta_y \to \infty$, there is regular sampling with $\beta = 1/l_{k_0}$.

The power density spectrum of the sampled signal becomes

$$\Phi_s(u) = \begin{cases} \beta[1 - 2\beta_y sin(l_{k_0}u) + 2\beta_y^2 cos(l_{k_0}u) - 2\beta_y^2] & u \neq 0 \\ 2\pi\beta^2\delta(u) & u = 0 \end{cases} \tag{11.38}$$

where the spectral distribution of the noise can be controlled via the product $l_{k_0}\beta$. The expression for $u \neq 0$ is the so-called *Flat Field Response Noise Spectrum (FFRNS)*, which represents the noise component of the Flat Field Response scaled with the sampling rate β.

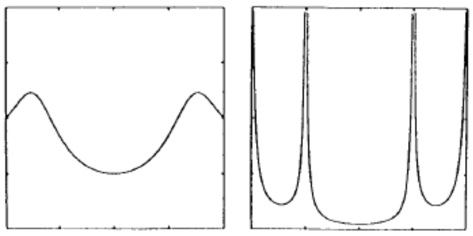


Figure 11.15: *FFRNS for* $l_{k_0}\beta = 0.5$ *and* $l_{k_0}\beta = 0.95$

Jittering

In contrast to Poisson sampling, with jittering only the positions of the samples on the regular grid are disturbed by adding a random variable j_k.

$$x_k = y_k + j_k \qquad y_k = \frac{k}{\beta} \qquad j_k \sim p(j_k) \qquad k = -\infty, ..., \infty \tag{11.39}$$

(11.40)

The power spectral density distribution of the sampled signal increases

$$\Phi_s(u) = \beta[1 - |\gamma(u)|^2] + 2\pi\beta^2|\gamma(u)|^2 \sum_{k=-\infty}^{infty} \delta(uk2\pi\beta) \tag{11.41}$$

where the first term contains broadband noise and the second term contains an aliasing-producing impulse comb now. This is weighted by $|\gamma(u)|^2$. If $p(j_k)$ is uniformly distributed over $[-1/2\beta, ..., 1/2\beta]$, the result is

$$\gamma(u) = \frac{\sin(u/2\beta}{u/2\beta} = \mathrm{sinc}(u/2\beta) \tag{11.42}$$

and

$$\Phi_s(u) = \beta[1 - sinc^2(u/2\beta)] + 2\pi\beta^2\delta(u) \tag{11.43}$$

which means that the delta functions that produce aliasing can disappear here as well.

If the jitter is distributed over a smaller interval, e.g., $[-\alpha/2\beta, ..., \alpha/2\beta]$, then delta functions immediately arise at the band limits.

In this approach, therefore, there is the possibility of suppressing the trade-off between aliasing and noise. It is important to analyze the *signal-to-noise ratio.*

In the 2D case $g(x, y)$ is the sampled image filtered with the reconstruction filter $r(x, y)$. The power spectral density is given by:

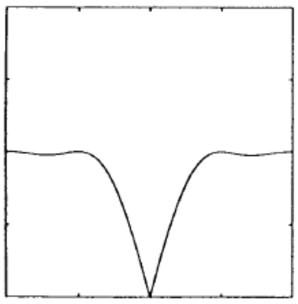


Figure 11.16: *FFRNS for uniformly distributed jitter*

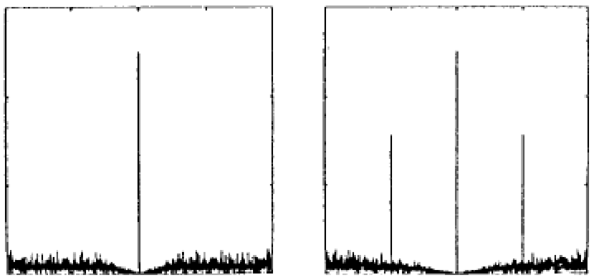


Figure 11.17: *FFRNS for* $\alpha = 1.0$ *and* $\alpha = 0.5$

$$\Phi_{g_r}(u,v) = [\Phi_f(u,v) * \Phi_s(u,v)] \cdot |R(u,v)|^2 \tag{11.44}$$

The spectra of the scanning processes are always of the form

$$\Phi_s(u,v) = 2\pi\beta^2\delta(u,v) + \beta\Phi_n(u,v) \tag{11.45}$$

If you plug this in, it follows

$$\Phi_{g_r}(u,v) = 2\pi\beta^2\Phi_f(u,v) \cdot |R(u,v)|^2 + \beta(\Phi_f(u,v) * \ Phi_n(u,v)) \cdot |R(u,v)|^2 \tag{11.46}$$

The *signal-to-noise ratio (SNR)* as an RMS measure is then obtained by integrating the two terms across the spectrum

$$\left|\left|\frac{S}{R}\right|\right|_{RMS} = (2\pi\beta)^{1/2}\left[\frac{\int\int_{R^s}\Phi_f \cdot |R|^2 dudv}{\int\int_{R^s}(\Phi_f * \Phi_n \cdot |R|^2 dudv}\right] \tag{11.47}$$

So the SNR only falls with the square root of the sampling rate. Normally, reasonable picture quality results from 40dB.

However, estimating the image spectrum is difficult with synthetic images. Therefore, the following approach can be used with Poisson sampling: Let β be the sampling rate and λ the size of the smallest detail to be represented. Since the samples with

$$P[d < \lambda] = 1 - e^{-\beta\lambda}$$

are distributed, a detail of size λ can be detected with a probability of P_0, i.e,

$$\beta = -\frac{ln(1 - P_0)}{\lambda}$$