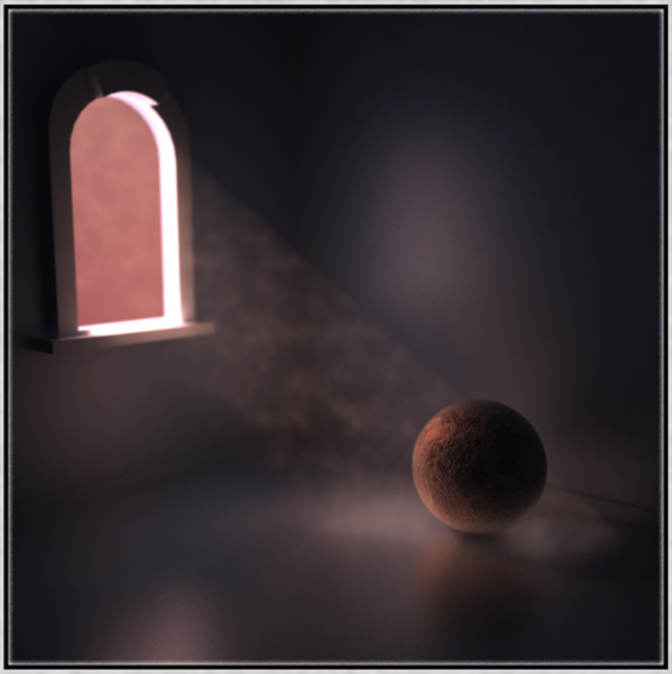
Wiesław Kotarski
Krzysztof Gdawiec
Grzegorz T. Machnik

Basics of Modelling and Visualization



Institute of Computer Science
University of Silesia
Katowice 2009

Basics of Modelling and Visualization

Wiesław Kotarski, Krzysztof Gdawiec,
Grzegorz T. Machnik

Basics of Modelling and Visualization

Katowice, 2009

Cover designed by Grzegorz T. Machnik

This project was completed with the support granted by Iceland, Liechtenstein and Norway by means of co-financing from the European Economic Area Financial Mechanism and the Norwegian Financial Mechanism as part of the Scholarship and Training Fund.

University of Silesia, Institute of Computer Science
Katowice, Poland 2009

Typeset by authors using the LaTeX 2ε
Printed in European Union

ISBN 978-83-929710-1-6

Contents

Preface

This textbook presents basic concepts related to modelling and visualization tasks. It consists of 6 chapters. Chapters 1-4 describe basic transformations in the plane and in the space and geometric forms of graphical objects such as curves, patches and fractals.

Namely, in Chapter 1 transformations such as translation, rotation, scaling, shears and projections are presented in a matrix form thanks to the usage of homogenous coordinates. Transformations are needed to perform required changes of graphical objects and their locations.

Chapter 2 discusses representations of curves used in computer graphics and computer aided design (CAD). Especially important are Bézier and subdivision curves. Bézier curves are free form curves whose shapes can be effectively modelled by changing the position of their control points. Subdivision is a very interesting and effective approach which makes it possible to generate smooth curves via pure geometrical and simple cutting corner algorithm.

In Chapter 3 patches are discussed – Bézier patches and subdivision patches. Bézier patches inherit nice properties from curves. Namely, their shapes can be easily and in a predictable way modelled with the help of control points. Moreover, Bézier patches are invariant under affine transformations and de Casteljau subdivision algorithm is also applicable to them. Also subdivision algorithms based on tensor product of subdivision matrices can be easily used to produce smooth patches.

Chapter 4 gives an introduction to graphical objects – fractals that cannot be presented with the help of functions. Fractals are very complicated objects that can be generated by a small amount of information gathered as coefficients of Iterated Function Systems, in short *IFS*. It is very interesting that fractals and Bézier curves and patches can be obtained using the same common approach. Namely, subdivision may produce both smooth curves and fractals.

Information presented in Chapters 1-4 is enough for the representation of geometry of graphical objects. But to obtain their visual photorealistic presentation we need to add to their geometry further factors such as lights, materials, textures and, of course, colours. Some information on that subject is presented in Chapter 5. In this chapter also problems related to cameras and rendering are described. Many photorealistic images are also presented.

In Chapter 6 a freeware software that can be used to visualize graphical objects is presented. POV Ray, MayaVi and Deep View are described. Using those software one can obtain photorealistic renderings. Among them there are nice example renderings of biological particles as hemoglobin that can be easily visualized with the help of Deep View and finally rendered in POV Ray.

This textbook was prepared for students of the specialization "Modelling and Visualization in Bioinformatics" but it should be helpful to anyone who is interested in computer graphics, modelling techniques and animation. Authors of this textbook believe that information presented in the book will be useful for students and will inspire their imagination in creation of photorealistic static $3D$ scenes and also will be helpful in creation of animations in an effective and professional way.

Wiesław Kotarski, Krzysztof Gdawiec, Grzegorz T. Machnik

Sosnowiec, June 2009

Chapter 1

Geometry

In this chapter we will present basic transformations in the plane and in the $3D$ space. All these transformations thanks to homogenous coordinates take the uniform matrix form. Homogenuos coordinates are used to present translation, for which a natural form is a vector one, in the matrix form. Presentation of translation in the matrix form is very useful because all the geometrical transformations such as: rotation, scaling, shears can be expressed in the matrix form. So, having matrix representation of geometrical transformations it is very easy to perform complex transformations by simply multiplying matrices. Here, it should be pointed out that the result of matrix multiplication depends on the order of matrices that is equivalent to the order of performed geometrical transformations. More information related to geometry can be found, e.g. in [14].

1.1 Transformations in the plane

Points in the plane, presented as row vectors $[x, y]$, will be considered in homogenous coordinates as $[x, y, 1]$.

Translation

Translation by a vector $[t_x, t_y]$ transforms a point $[x, y]$ into a new one $[x', y']$. This can be written in the following form:

$$\begin{cases} x' = x + t_x, \\ y' = y + t_y, \\ 1 = 1. \end{cases} \tag{1.1}$$

Applying identity $1 = 1$ enables matrix representation of translation:

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} = [x, y, 1] \cdot T(t_x, t_y), \tag{1.2}$$

where $T(t_x, t_y)$ is the translation operator with parameters t_x and t_y.

Rotation about the origin

A rotation of the point $[x, y]$ through an angle φ about the origin transforms it to the new point $[x', y']$. A rotation angle is assumed to be positive, if the rotation is carried out in the anticlockwise direction. Rotation can be written in the following matrix form:

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x, y, 1] \cdot R(\varphi), \tag{1.3}$$

where $R(\varphi)$ is the rotation operator through an angle φ about the origin.

Scaling

Scaling with factors s_x and s_y according to OX and OY axes, respectively, transforms the point $[x, y]$ into the new one $[x', y']$. It can be written in the following form:

$$\left[x', y', 1\right] = [x, y, 1] \cdot \begin{bmatrix} s_y & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x, y, 1] \cdot S(s_x, s_y), \tag{1.4}$$

where $S(s_x, s_y)$ is the scaling operator with factors s_x and s_y according to OX and OY axes, respectively.

Sheares

- Shear in the direction of OX axis

  A shear in the direction of OX axis with the factor r is described by the following formula:

$$\begin{cases} x' = x + ry, \\ y' = y, \\ 1 = 1, \end{cases} \tag{1.5}$$

that can be written in the equivalent matrix form:

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ r & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x, y, 1] \cdot Sh_x(r), \tag{1.6}$$

where $Sh_x(r)$ determines the shear operator with the factor r in the direction of OX axis.

- Shear in the direction of OY axis

  A shear in the direction of OY axis with the factor r is described by the following formula:

$$\begin{cases} x' = x, \\ y' = rx + y, \\ 1 = 1, \end{cases} \tag{1.7}$$

that can be written in the equivalent matrix form:

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} 1 & r & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x, y, 1] \cdot Sh_y(r), \tag{1.8}$$

where $Sh_y(r)$ determines the shear operator with the factor r in the direction of OY axis.

1.2 Transformations in the $3D$ space

Points in the $3D$ space are row vectors $[x, y, z]$ and they are considered in homogenous coordinates as $[x, y, z, 1]$.

Translation $3D$

Translation by a vector $[t_x, t_y, t_z]$ transforms a point $[x, y, z]$ to a new one $[x', y', z']$. This can be written in the following form:

$$\begin{cases} x' = x + t_x, \\ y' = y + t_y, \\ z' = z + t_z, \\ 1 = 1. \end{cases} \tag{1.9}$$

Applying identity $1 = 1$ enables matrix representation of translation $3D$:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = [x, y, z, 1] \cdot T(t_x, t_y, t_z), \tag{1.10}$$

where $T(t_x, t_y, t_z)$ is the translation operator with parameters t_x, t_y, t_z.

Rotations about coordinate axes

In the $3D$ space rotations about three axes can be defined. In Fig. 1.1 right-handed coordinate system is presented. The arrows show positive rotation directions about axes. Below matrices that define rotations according to axes OX, OY and OZ, respectively through angles φ_x, φ_y i φ_z are given.

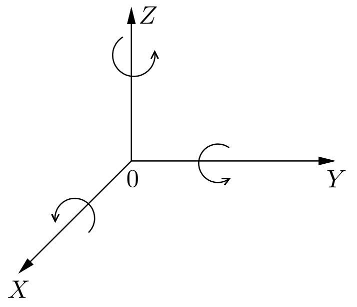


Fig. 1.1. Right-handed coordinate axes system.

- Rotation about OX axis

$$
\begin{aligned}
[x', y', z', 1] &= [x, y, z, 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi_x & \sin\varphi_x & 0 \\ 0 & -\sin\varphi_x & \cos\varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= [x, y, z, 1] \cdot R_x(\varphi_x),
\end{aligned} \tag{1.11}
$$

where $R_x(\varphi_x)$ is the rotation operator about OX axis through an angle φ_x.

- Rotation about OY axis

$$
\begin{aligned}
[x', y', z', 1] &= [x, y, z, 1] \cdot \begin{bmatrix} \cos\varphi_y & 0 & -\sin\varphi_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi_y & 0 & \cos\varphi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= [x, y, z, 1] \cdot R_y(\varphi_y),
\end{aligned} \tag{1.12}
$$

where $R_y(\varphi_y)$ is the rotation operator about OY axis through an angle φ_y.

- Rotation about OZ axis

$$
\begin{aligned}
[x', y', z', 1] &= [x, y, z, 1] \cdot \begin{bmatrix} \cos\varphi_z & \sin\varphi_z & 0 & 0 \\ -\sin\varphi_z & \cos\varphi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= [x, y, z, 1] \cdot R_z(\varphi_z),
\end{aligned} \tag{1.13}
$$

where $R_z(\varphi_z)$ is the rotation operator about OZ axis through an angle φ_z.

Scaling

Scaling in the $3D$ space is described by the following formula:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x, y, z, 1] \cdot S(s_x, s_y, s_z), \quad (1.14)$$

where $S(s_x, s_y, s_z)$ is the scaling operator with scaling factors s_x, s_y, s_z about OX, OY, OZ axes, respectively.

1.3 Projections

$3D$ graphical objects or the virtual world modelled using computer graphics or computer aided design software have to be projected to the plane to display them on the computer monitor. There are two main types of projections:

- parallel projection,
- perspective projection.

In both projections the depth of object is lost. Parallel projection preserves parallelism of lines, whereas perspective projection does not, and in that projection extensions of parallel lines may intersect. In Fig. 1.2 two kinds of projections of a cube are presented. Thanks to homogeneous coordinates both types of projections can be expressed with the help of one formula that uses natural information needed in computations.

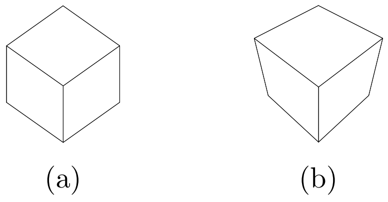


Fig. 1.2. Projection of a cube: (a) parallel, (b) perspective.

The formula for the projection, denoted by M, is as follows:

$$M = \mathbf{n}^T \cdot \mathbf{v} - (\mathbf{n} \circ \mathbf{v}) \cdot I_4, \quad (1.15)$$

where $\mathbf{n}$ is the vector perpendicular to the viewplane, $\mathbf{n}^T$ is its transposition, $\mathbf{v}$ denotes the position of an observer or camera in homogeneous coordinates, I_4 is the identity matrix of dimensions 4×4.

Note that "$\cdot$" denotes multiplication of matrices, "$\circ$" is the scalar product of vectors giving the matrix of dimensions 4×4 and the number as the result, respectively. The vector $\mathbf{v}$ takes the following forms $\mathbf{v} = [x, y, z, 1]$ for perspective projection (Fig. 1.3 a) and $\mathbf{v} = [x, y, z, 1]$ for parallel projection (Fig. 1.3 b), respectively. Notation $[x, y, z, 0]$ determines the point at infinity in the direction $[x, y, z]$.

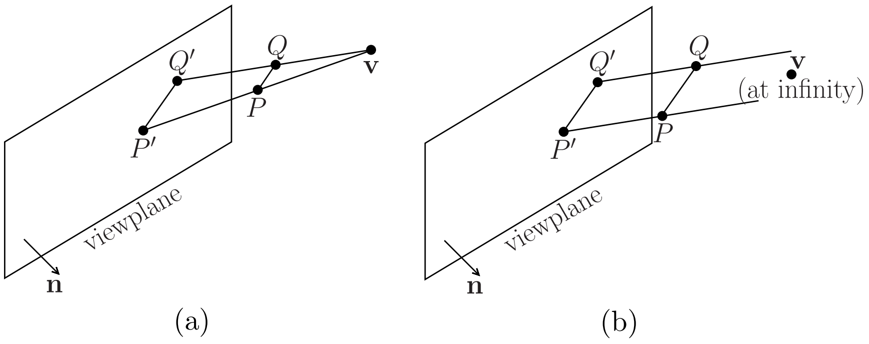


Fig. 1.3. 3D projections: (a) perspective, (b) parallel.

Chapter 2

Curves

Curves arise in many applications such as art, industrial design, mathematics, architecture and engineering. Many drawing applications and computer-aided design packages have been developed to facilitate the creation of curves. A good illustrative example is that of computer fonts which are defined by curves. Different font sizes are obtained by applying scaling transformations whereas special font effects can be obtained by using other transformations such as shears and rotations. Similarly, in other applications there is a need to perform various changes of curves such as modifying, analyzing and visualizing. To perform such operations, a mathematical representation for curves is needed. In this chapter different kinds of representation of curves will be presented: parametric, non-parametric explicit, using interpolation and approximation, implicit, Bézier and subdivision which also lead to curves. The representations of curves are naturally related to representations of surfaces. More information on curves and their representations can be found in [8, 12, 14].

2.1 Curves as functions

Curves can be defined using the following representations:

Parametric curves

The coordinates of points of a parametric curve are expressed as functions of a variable or a parameter denoted, e.g. by t. A curve in the plane has the form $C(t) = (x(t), y(t))$, and a curve in the space has the form $C(t) = (x(t), y(t), z(t))$. The functions $x(t)$, $y(t)$, $z(t)$ are called the coordinate functions. A parametric curve defined by polynomial coordinate functions is called a polynomial function.

Example 2.1. Parabola (t, t^2), for $t \in \mathbb{R}$ is a polynomial curve of degree 2.

Example 2.2. Unit circle $(\cos(t), \sin(t))$, for $t \in [0, 2\pi]$.

In Fig. 2.1 two examples of parametric curves are presented.

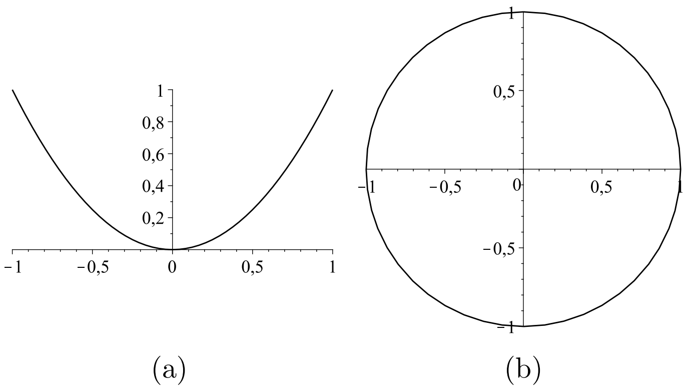


Fig. 2.1. Parametric curves: (a) parabola, (b) unit circle.

Non-parametric explicit curves

The coordinates (x, y) of points of a nonparametric explicit planar curve satisfy $y = f(x)$ or $x = g(y)$. Such curves have the parametric form $C(t) = (t, f(t))$ or $C(t) = (g(t), t)$.

Example 2.3. Parabola $y = x^2$, for $x \in \mathbb{R}$.

Example 2.4. Upper half of a unit circle $y = \sqrt{1 - x^2}$, for $x \in [-1, 1]$.

In Fig. 2.2 an example of an explicit curve is presented.

Implicit curves

The coordinates (x, y) of points of non-parametric implicit curve satisfy $F(x, y) = 0$, where F is some function. When F is polynomial in variables x and y then it is called an algebraic curve.

Example 2.5. Circle $x^2 + y^2 - 1 = 0$.

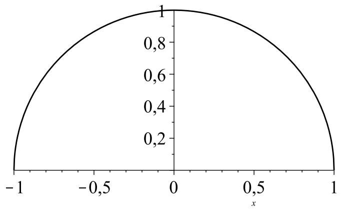


Fig. 2.2. Explicit curve: half of a unit circle.

2.2 Interpolation and approximation curves

Experiments usually deliver discrete data obtained by sampling some investigated continuous process. In two dimensional case that data represent a finite number of points in the plane. The task is to find in some sense an optimal function basing on those points. There are two approaches to solve the problem – interpolation and approximation.

Interpolation curves

Let $f(x)$ be an unknown function defined on an interval $[a, b]$ with values in $\mathbb{R}$. Assume that we know values of the function $f(x)$ at given points, the so-called interpolation knots $a \leq x_0 < x_1 < \ldots < x_n \leq b$. Namely, $f(x_0) = y_0$, $f(x_1) = y_1$,..., $f(x_n) = y_n$. We are looking for a polynomial $W_n(x)$ of the n-th order having the general form:

$$W_n(x) = a_0 + a_1 x + \ldots + a_n x^n \tag{2.1}$$

with unknown coefficients $a_0, a_1 \ldots, a_n$, such that

$$W_n(x_0) = f(x_0), \quad W_n(x_1) = f(x_1), \ldots, W_n(x_n) = f(x_n). \tag{2.2}$$

That problem known as Lagrange approximation has only one solution defined as follows:

$$W_n(x) = \sum_{k=0}^{n} y_k w_n^k(x), \tag{2.3}$$

where

$$w_n^k(x) = \prod_{\substack{j=0\\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}.$$

Observe that:

$$w_n^k(x_i) = \begin{cases} 0, & \text{for} \quad i \neq k, \\ 1, & \text{for} \quad i = k. \end{cases}$$

Example 2.6. Find the Lagrange interpolation polynomial that for points $1, 2, 3, 4$ take values $3, 1, -1, 2$, respectively. So then, we are looking for a polynomial of the third order $W_3(x)$. It takes the following form:

$$\begin{aligned} W_3(x) = {} & 3\frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} + 1\frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \\ & - 1\frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} + 2\frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)}. \end{aligned}$$

After easy simplifications $W_3(x) = \frac{5}{6}x^3 - 5x^2 + \frac{43}{6}$. In Fig. 2.3 that polynomial is presented together with the interpolation knots.

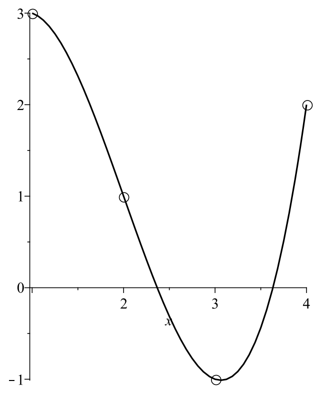


Fig. 2.3. Interpolation curve.

Approximation curves

Assuming that we have a sequence $x_0 < x_1 < \ldots < x_n$ for which an unknown function $f(x)$ take values $f(x_0), f(x_1), \ldots, f(x_n)$, respectively. We find an approximation polynomial $W_m(x)$ of degree $m \leq n$ i.e.

$$W_m(x) = a_0 + a_1 x + \ldots + a_m x^m,$$

such that

$$\sum_{k=0}^{n} [f(x_k) - W_m(x_k)]^2 \to \min . \tag{2.4}$$

The above optimization problem, known as the least square fitting method, has a unique solution being a set of coefficients $\{a_0, \ldots, a_m\}$ that can be found from the following equations:

$$\begin{cases} a_0 S_0 + a_1 S_1 + \cdots + a_m S_m = T_0 \\ a_0 S_1 + a_1 S_2 + \cdots + a_m S_{m+1} = T_1 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_0 S_m + a_1 S_{m+1} + \cdots + a_m S_{2m} = T_m \end{cases}, \tag{2.5}$$

where

$$S_k = \sum_{i=0}^{n} x_i^k, \quad k = 0, 1, \ldots, 2m,$$

$$T_k = \sum_{i=0}^{n} x_i^k f(x_i), \quad k = 0, 1, \ldots, m.$$

Example 2.7. Find the polynomial of the first order that approximates an unknown function taking the values $1.6, 2.0, 2.5, 3.5$ for the arguments $1, 2, 4, 7$, respectively.

We are looking for the polynomial of the form $W_1(x) = a_0 + a_1 x$. Necessary computations of coefficients S_0, S_1, S_2, T_0 and T_1 are given in the table below.

i	x_i^0	x_i^1	x_i^2	$f(x_i)$	$x_i f(x_i)$
0	1	1	1	1.6	1.6
1	1	2	4	2.0	4.0
2	1	4	16	2.5	10.0
3	1	7	49	3.5	24.5
	$S_0 = 4$	$S_1 = 14$	$S_2 = 70$	$T_0 = 9.6$	$T_1 = 40.1$

Next, coefficients a_0 and a_1 can be found from the following equations:

$$\begin{cases} 4a_0 + 14a_1 = 9.6, \\ 14a_0 + 70a_1 = 40.1. \end{cases} \tag{2.6}$$

The solution of equation (2.6) is the following: $a_0 = 1.3$ and $a_1 = 0.3$. So, the approximation polynomial, being a line, has the form: $W_1(x) = 1.3 + 0.3x$. In Fig 2.4 that line is presented together with the points it approximates.

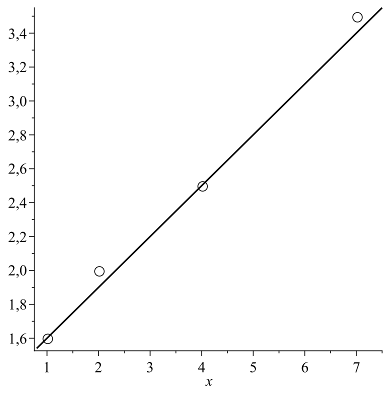


Fig. 2.4. Approximation line.

2.3 Bézier curves

In computer graphics and computer-aided design Bézier curves are widely used because of their flexibility and possibility to change predictably their shapes in an intuitive way. Bézier curves popularity is due to the fact that they possess a number of nice mathematical properties which enable their easy manipulation and which is very important – no mathematical knowledge is required to use those curves. They have been invented by two Frenchmen working in the automobile industry, Pierre Bézier at Renault and Paul de Casteljau at Citroën during the period between 1958-60. Bézier curves are the polynomial ones. Their shapes are fully determined by a finite number of control points. By changing the position of control points one can influence the shape of the curve. In practice only quadratic and cubic Bézier curves are applied. Quadratic Bézier curves are determined by three control points, whereas cubic Bézier curves for their description need four control points. Bézier curves with more than four control points can also be defined but they are not good in practice.

Quadratic Bézier curves

Let $P_0 = [x_0, y_0]$, $P_1 = [x_1, y_1]$, $P_2 = [x_2, y_2]$ be given three points, the so-called control points. Then the quadratic Bézier curve $Q(t)$ is defined by the formula:

$$Q(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2, \quad t \in [0,1]. \tag{2.7}$$

The starting point of the curve is $Q(0) = P_0$ and the ending one is $Q(1) = P_2$. So then, the curve $Q(t)$ interpolates the first and the last control point and approximates the point P_1. The curve $Q(t)$ can be expressed in the parametric form $(x(t), y(t))$, where

$$\begin{cases} x(t) = (1-t)^2 x_0 + 2(1-t)tx_1 + t^2 x_2, \\ y(t) = (1-t)^2 y_0 + 2(1-t)ty_1 + t^2 y_2, \end{cases}$$

for $t \in [0,1]$.

In Fig. 2.5 an example of a quadratic Bézier curve is presented. By joining the control points one obtains the so-called control polygon, here – a triangle. It is easy to observe that any quadratic Bézier curve lies within the triangle defined by its control points. The curve $Q(t)$ is tangent to segments $\overline{P_0P_1}$ and $\overline{P_1P_2}$, respectively. Moreover, the curve $Q(t)$ is invariant under affine transformation of its control points. It means that the result of any affine transformation applied to the curve $Q(t)$ is equivalent to using the same affine transformation restricted only to control points.

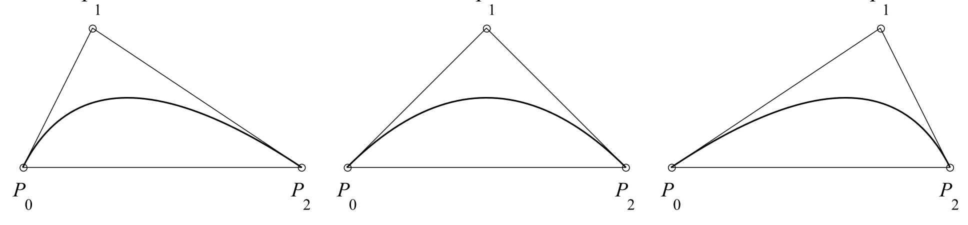


Fig. 2.5. Quadratic Bézier curve with different positions of the one control point.

Cubic Bézier curves

Let $P_0 = [x_0, y_0]$, $P_1 = [x_1, y_1]$, $P_2 = [x_2, y_2]$, $P_3 = [x_3, y_3]$ be given four control points. Then the cubic Bézier curve $Q(t)$ is defined by the formula:

$$Q(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, \quad t \in [0,1]. \tag{2.8}$$

The starting point of the curve is $Q(0) = P_0$ and the ending one is $Q(1) = P_3$. So then, the curve $Q(t)$ interpolates the first and the last control point and approximates the points P_1, P_2. The curve $Q(t)$ can be expressed in the parametric form $(x(t), y(t))$, where

$$\begin{cases} x(t) = (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3, \\ y(t) = (1-t)^3 y_0 + 3(1-t)^2 t y_1 + 3(1-t)t^2 y_2 + t^3 y_3, \end{cases}$$

for $t \in [0,1]$.
In Fig. 2.6 an example of a cubic Bézier curve is presented. Similarly as for quadratic Bézier curve also cubic Bézier curve lies within the control polygon defined by its control points, here – a tetragon. The curve $Q(t)$ is tangent to segments $\overline{P_0P_1}$ and $\overline{P_2P_3}$, respectively.

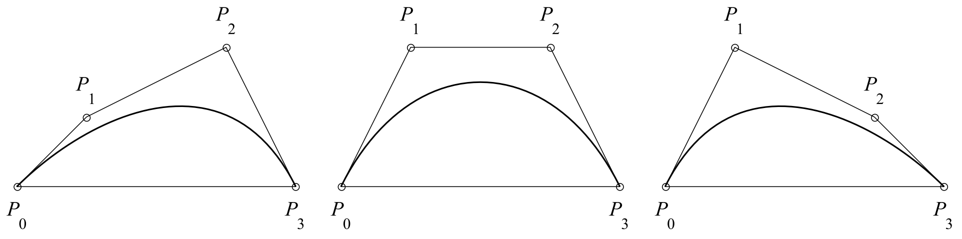


Fig. 2.6. Cubic Bézier curve with different positions of two control points.

2.4 Subdivision curves

Subdivision is a simple purely geometric method that when applied to a starting polygon often leads to a smooth curve. That method was discovered by Chaikin [3], in 1974. Using different subdivision strategies one can obtain smooth limit curves good for modelling or curves that are less regular, than smooth ones, and therefore not practically useful.

Chaikin's algorithm

According to Chaikin's paradigm, a smooth curve can be generated using the corner cutting scheme. It works in the following way. For a given control polygon $\{P_0, ..., P_n\}$, we create a new one by generating a sequence of the following control points $\{Q_0, R_0, Q_1, R_1, ..., Q_{n-1}, R_{n-1}\}$, where Q_i and R_i are computed according to the formulae:

$$\begin{cases} Q_i = \frac{3}{4}P_i + \frac{1}{4}P_{i+1}, \\ R_i = \frac{1}{4}P_i + \frac{3}{4}P_{i+1}. \end{cases} \tag{2.9}$$

In Fig. 2.7 the result of Chaikin's algorithm is presented. Observe that every segment of the polygon is divided into three parts in the ratio $\frac{1}{4} : \frac{1}{2} : \frac{1}{4}$. Two new points are placed on the segment and corner points are eliminated.

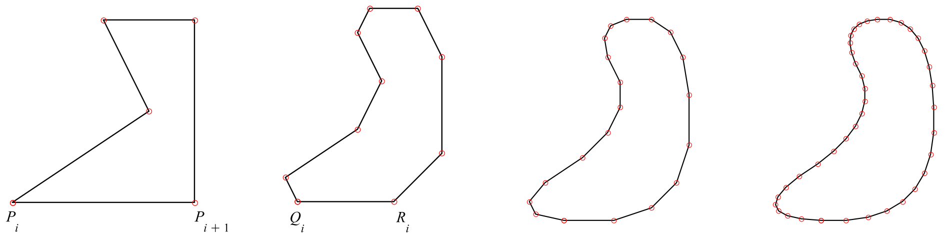


Fig. 2.7. Chaikin's cutting corner algorithm. Iterations: $0, 1, 2, 3$.

After a few iterations one obtains a visually smooth resulting curve. Chaikin's method provides a very simple and elegant drawing mechanism. It is interesting that using a subdivision with different parameters from coefficients $\frac{3}{4}$ and $\frac{1}{4}$ leads to limit curves that are not necessarily smooth. If we take in Chaikin's algorithm subdivision parameters $\frac{4}{7}$ and $\frac{3}{7}$ then we obtain a continuous, but not smooth, limit curve, as in Fig. 2.8.

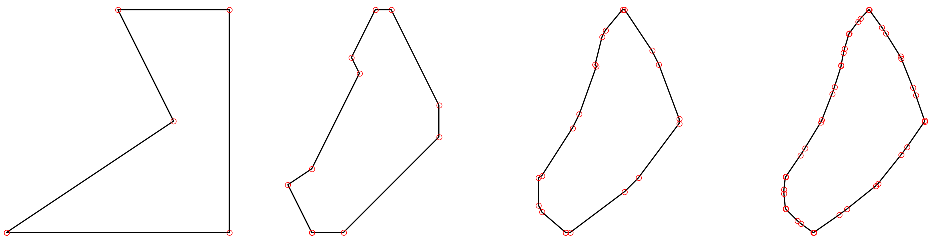


Fig. 2.8. Cutting corner algorithm with parameters $\frac{4}{7}$ and $\frac{3}{7}$. Iterations: $0, 1, 2, 3$.

The de Casteljau algorithm

We demonstrate the performance of the de Casteljau algorithm in the case of a quadratic Bézier curve. The algorithm enables us to obtain a quadratic Bézier curve in a pure geometrical way. So start with three given control points P_0, P_1, P_2 and perform the following construction, illustrated in Fig. 2.9:

- fix a parameter $t \in [0,1]$,
- divide the segment $\overline{P_0P_1}$ in the ratio $(1-t):t$ and put on the segment the point P_1^1 such that
$$P_1^1 = (1-t)P_0 + tP_1,$$
- divide the segment $\overline{P_1P_2}$ in the same ratio $(1-t):t$ and put on the segment the point P_2^1 such that
$$P_2^1 = (1-t)P_1 + tP_2,$$
- divide the segment $\overline{P_1^1P_2^1}$ in the same ratio $(1-t):t$ and put on the segment the point P_2^2 such that
$$P_2^2 = (1-t)P_1^1 + tP_2^1,$$
- define the point $Q(t) = P_2^2$.

After easy computations we get:

$$Q(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2, \quad t \in [0,1], \tag{2.10}$$

that means that we obtained a parabolic curve, a quadratic Bézier curve parametrized by $t \in [0,1]$. Observe, that using an analytical approach we have obtained earlier the same expression for $Q(t)$. Then, one can see that the point P_2^2 divides the curve $Q(t)$ into two parts: the left one with control points P_0, P_1^1, P_2^2 and the right one with control points P_2^2, P_2^1, P_2. Further, every part is a Bézier curve with its control points. The possibility to divide

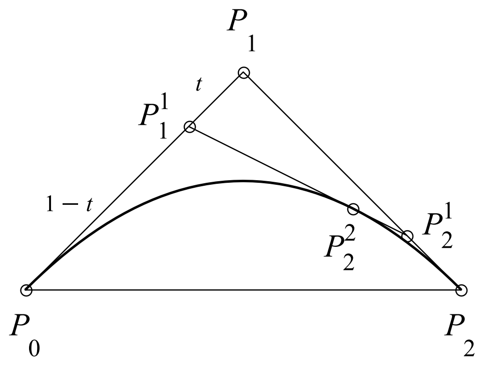


Fig. 2.9. The de Casteljau algorithm.

any Bézier curve into parts being also Bézier curves is very important in applications. That feature enables us to model effectively any $2D$ contour locally because changes can be done only on the parts of the contour where they are needed. Similar construction as above can be repeated in the case of a cubic Bézier curve. We shall leave that for the reader as an excercise.

Next, something should be said about joining Bézier curves presented in Fig. 2.10. It is clear that if the last control point of the first curve covers the first control point of the second curve, i.e. $P_2 = R_0$, then we obtain only continuous joining of the both curves. In such a situation a cusp can appear. To avoid a cusp and get smooth joining of the curves, some additional geometrical assumptions should be satisfied. Namely, three following points – preceeded the last, the last one of the first curve and the first, the second one of the second curve should be collinear. That means that the points P_1, $P_2 = R_0$, R_1 should form a linear segment.

All the nice properties mentioned above of Bézier curves are very useful in modelling of $2D$ contours. In Fig. 2.11 a heart and a flower modelled respectively with the help of 4 quadratic and 8 cubic Bézier curves are presented.

Now consider a special case of subdivision with $t = \frac{1}{2}$. That subdivision is called the midpoint strategy. The dependency between control points of every part of a quadratic Bézier curve and its control points P_0, P_1, P_2 can be expressed with the help of subdivision matrices L and R in the following way:

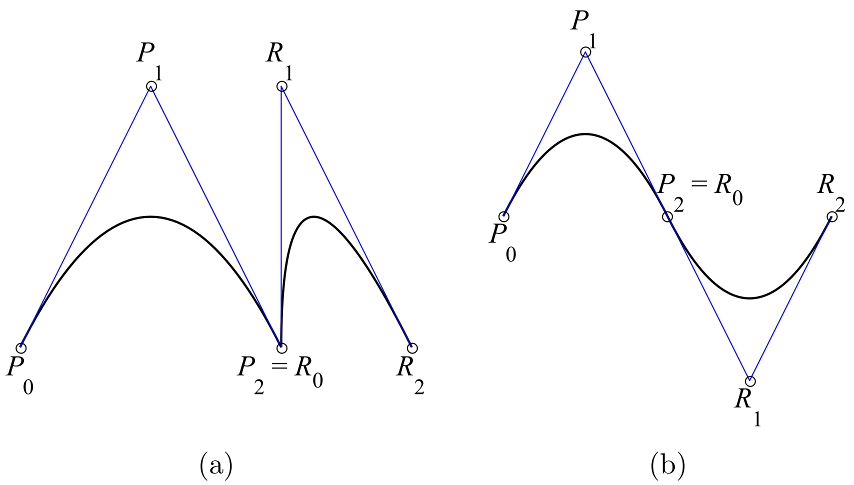


Fig. 2.10. Joining of two Bézier curves: (a) continuous, (b) smooth.

$$\begin{bmatrix} P_0 \\ P_1^1 \\ P_2^2 \end{bmatrix} = L \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix}, \quad \begin{bmatrix} P_2^2 \\ P_2^1 \\ P_2 \end{bmatrix} = R \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix},$$

where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}, \mathrm{R} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Treating points P_0, P_1^1, P_2^2 as the new input points P_0, P_1, P_2 and multiplying them by matrices L and R one can obtain new points of the new left and the right parts. The same can be applied to points P_2^2, P_2^1, P_2. Repeating the described process, after several iterations, one can obtain a good approximation of the quadratic Bézier curve, as presented in Fig. 2.12.

Similarly, as above, two subdivision matrices for a cubic Bézier curve can be expressed as follows:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1/4 & 1/2 & 1/4 & 0 \\ 1/8 & 3/8 & 3/8 & 1/8 \end{bmatrix}, R = \begin{bmatrix} 1/8 & 3/8 & 3/8 & 1/8 \\ 0 & 1/4 & 1/2 & 1/4 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

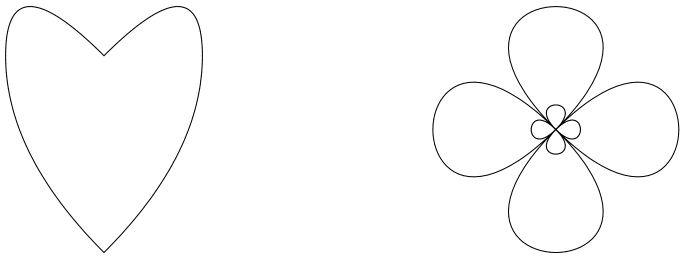


Fig. 2.11. A heart and a flower modelled with the help of Bézier curves.

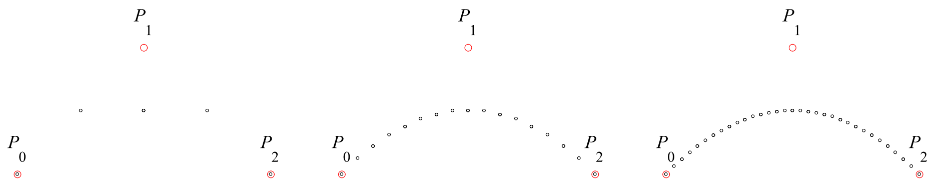


Fig. 2.12. Approximation of a quadratic Bézier curve obtained via subdivision. Iterations: $1, 3, 4$.

Observe, that using two matrices L and R one can generate iteratively Bézier curves via the subdivision method. But if we take into consideration the following subdivision matrices:

$$L = \begin{bmatrix} 3/4 & 1/4 & 0 \\ 1/4 & 3/4 & 0 \\ 0 & 3/4 & 1/4 \end{bmatrix}, R = \begin{bmatrix} 1/4 & 3/4 & 0 \\ 0 & 3/4 & 1/4 \\ 0 & 1/4 & 3/4 \end{bmatrix},$$

then one can generate iteratively Chaikin's curve that is equivalent to the so-called B-spline quadratic curve presented in Fig. 2.13. None of the control points lie on Chaikin's curve in opposite to Bézier curve which passes through P_0, P_2, i.e. the first and the last control points.

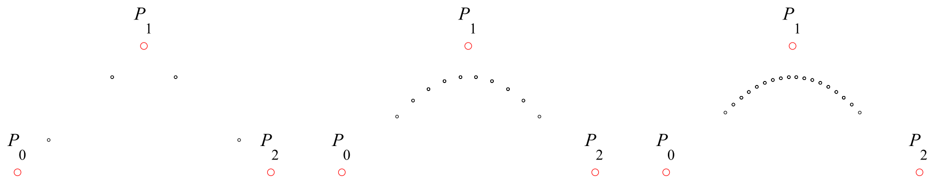


Fig. 2.13. Approximation of a B-spline curve obtained via subdivision. Iterations: $1, 3, 4$.

Chapter 3

Patches

Patches are used in computer graphics and computer aided design to model surfaces of $3D$ graphical objects. They are used to model cars, planes, buildings, virtual reality in computer games and many others. Patches are natural generalization of curves. Some techniques known for curves can be easily extended to patches. Especially Bézier patches and some patches obtained via subdivision techniques can be treated as a collection of curves lying on the patches. Those patches are easily obtained using the so-called tensor product of curves. Very useful surfaces such as extruded, ruled, swept and surfaces of revolution can be constructed with the help of the so-called generating functions. Further, more detailed, information about patches and their representations can be found in [8, 12, 27, 29].

3.1 Surfaces as functions

Surfaces can be defined explicitly as $z = f(x, y)$, where $f : \mathbb{R}^2 \to \mathbb{R}$ and $(x, y) \in D \subset \mathbb{R}^2$. If they are defined as a subset of $\mathbb{R}^3$ of the form $\{(x, y, z) : F(x, y, z) = 0\}$, where $F : \mathbb{R}^3 \to \mathbb{R}$ is some function, then they are called implicit surfaces. Further, parametric surfaces can be expressed as functions of two variables, for instance in the form $Q(u, v) = (x(u, v), y(u, v), z(u, v))$, for (u, v) belonging to some subset of $\mathbb{R}^2$. Below we will give some examples of surfaces.

Example 3.1. The explicit surface $z = \sin(x)\cos(y)$ for $(x, y) \in D$, where $D = \{(x, y) : -1 \le x \le 1, -1 \le y \le 1\}$ is presented in Fig. 3.1(a).

Example 3.2. The implicit surfaces of the form $ax+by+cz+d = 0$ for constants $a, b, c, d \in \mathbb{R}$ are planes.

Example 3.3. The implicit surface $x^2 + y^2 + z^2 - 1 = 0$ is the unit sphere with the centre at the origin.

Example 3.4. The parametric surface $Q(u, v) = (u - v, u + v, u^2 - v^2)$ for $(u, v) \in \mathbb{R}^2$, resembling a saddle, is illustrated in Fig. 3.1(b).

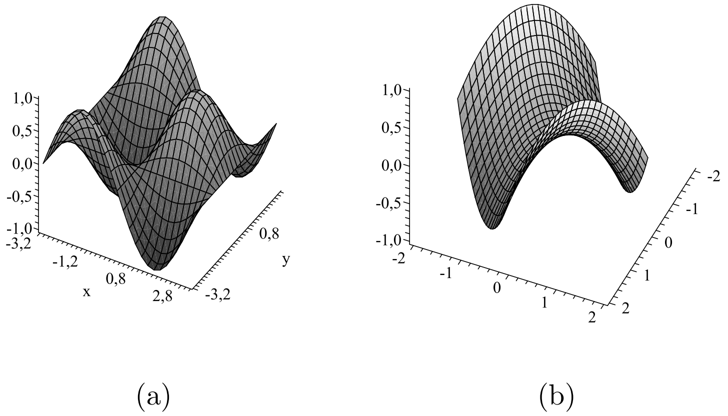


Fig. 3.1. Surfaces: (a) explicit, (b) parametric.

Further examples of many different surfaces can be found in the Internet at http://www.javaview.de/demo/index.html. There is a huge collection of surfaces that are presented as JavaView applets. Those surfaces can be rotated, translated, scaled and precisely investigated. Among them, one can find classical $3D$ objects as ellipsoid, hyperboloid, paraboloid, cylinder. Also strange graphical objects as one-sided surface – Moebius strip and zero volume Klein bottle are presented. Here, it should be mentioned that JavaView is a nice $3D$ geometry viewer and a mathematical visualization software that allows a display of $3D$ geometries and interactive geometry experiments embedded in any HTML document in the Internet. That software initialized by Konrad Polthier from TU-Berlin was selected for the presentation at the Third European Congress of Mathematicians in Barcelona in 2000.

3.2 Bézier patches

Bézier patches are obtained as a family of Bézier curves parametrized in two perpendicular directions denoted by u and v, respectively. The analytical formula for Bézier patches can be easily presented using the so-called Bernstein polynomials. So, let us define the following functions:

$$B_i^n(t) = \begin{cases} \frac{n!}{(n-i)!i!}(1-t)^{n-i}t^i, & \text{for } \ \mathrm{i} = 0,1,2,\ldots,\mathrm{n}, \\ 0, & \text{otherwise,} \end{cases}$$

for $t \in [0,1]$.
$B_i^n(t)$ for $i = 0,1,2,\ldots,n$ are called Bernstein polynomials or Bernstein basis functions of degree n. The most important in practice are two Bernstein basis: B_i^2 for $i = 0,1,2$ and B_i^3 for $i = 0,1,2,3$. The basis B_i^2 for $i = 0,1,2$ consists of three functions $(1-t)^2, 2(1-t)t, t^2$, whereas the basis B_i^3 for $i = 0,1,2,3$ contains four functions $(1-t)^3, 3(1-t)^2t, 3(1-t)t^2, t^3$. They are used to define bi-quadratic and bi-cubic Bézier patches, respectively. Here, it should be pointed out that many nice properties of Bézier curves have Bézier patches, too. Namely, convex hull and affine invariance property. Bézier patches can also be divided into subpatches using the de Casteljau algorithm. However, joining Bézier patches smoothly is much more complicated in comparison to curves.

Bi-quadratic Bézier patches

Bi-quadratic Bézier patches are characterized by 9 control points: P_{ij} for $i = 0,1,2, j = 0,1,2$ in $\mathbb{R}^3$. They are enumerated as shown in Fig. 3.2. The analytical formula for a bi-quadratic Bézier patch is as follows:

$$Q(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} P_{ij}B_i^2(u)B_i^2(v), \quad u,v \in [0,1]. \tag{3.1}$$

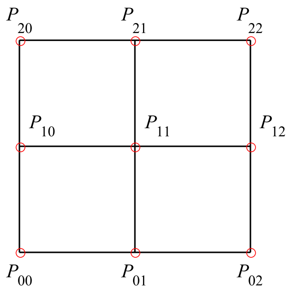


Fig. 3.2. Control points array of a quadratic patch.

In Fig. 3.3(a) an example of a bi-quadratic Bézier patch is presented.

Bi-cubic Bézier patches

Bi-cubic Bézier patches are characterized by 16 control points: P_{ij}, $i = 0, \ldots, 3, j = 0, \ldots, 3$ in $\mathbb{R}^3$. The analytical formula for a cubic Bézier patch is as follows:

$$Q(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} P_{ij} B_i^3(u) B_i^3(v), \quad u, v \in [0,1]. \tag{3.2}$$

In Fig. 3.3(b) an example of a bi-cubic Bézier patch is presented.

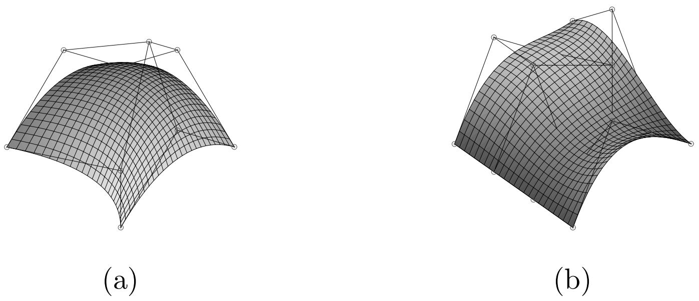


Fig. 3.3. Bézier patches: (a) bi-quadratic, (b) bi-cubic.

In Fig. 3.4 shapes of the well-known teapot from Utah and a rabbit modelled with the help of Bézier patches are presented.

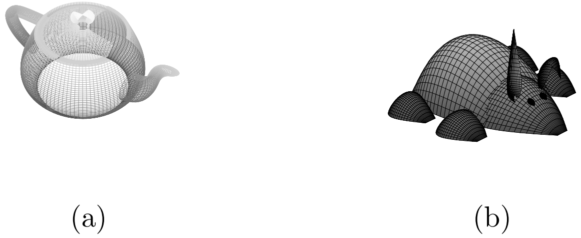


Fig. 3.4. Examples: (a) teapot, (b) rabbit.

3.3 Special surface constructions

In CAD systems surfaces that are obtained via some transformations applied to the so-called generating functions are widely used. Most often are used:

- extruded surfaces,
- ruled surfaces,
- swept surfaces,
- surfaces of revolution.

An extruded surface can be obtained when a spatial generating function $\mathbf{f}(s)$ is translated in the given direction. A ruled surface is formed by using two spatial curves $\mathbf{f}(s)$ and $\mathbf{g}(s)$ when points on each curve corresponding to the parameter s are joint by a line. Swept surfaces are obtained by translating a generating function $\mathbf{f}(s)$ along a trajectory curve $\mathbf{g}(s)$. Swept surfaces are more general than the extruded ones. Surfaces of revolution are obtained by rotating a generating curve $\mathbf{f}(s)$ about a fixed axis. It should be assumed that both the curve and the axis lie in one plane and the curve has no self-intersections. Those assumptions guarantee to obtain surfaces not having self-intersections. Surfaces of revolution are used to model symmetrical objects, e.g. vases, bottles. In Fig. 3.5 some examples of special surfaces are presented.

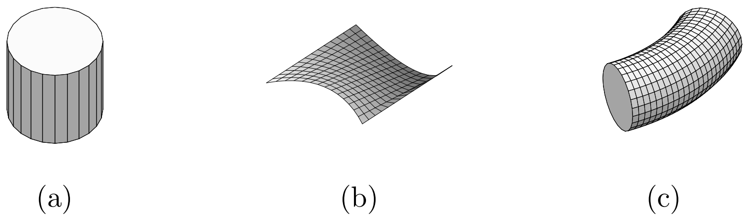


Fig. 3.5. Surfaces: (a) extruded, (b) ruled, (c) swept.

3.4 Subdivision patches

Subdivision is a very simple and efficient method of obtaining smooth surface via cutting corner algorithm applied to some initial mesh. A great number of subdivision algorithms generating patches is known, e.g. Doo-Sabin (1978), Catmull-Clark (1978), Loop (1987), Kobbelt (2000). Some of them are extended to meshes of arbitrary topology. Subdivision algorithms are widely described in [8, 27, 29]. While using subdivision, it is possible to generate complicated $3D$ graphical objects. That method was first used commercially by Pixar Studio in his 3.5 min animation "Geri's Game" which was awarded by an Oscar in the category of short computer animations in 1997. Next, that effective method was used by Disney/Pixar in productions of "Bug's Life" and "Toy Story 2" for the creation of almost all the elements of virtual film world. Subdivision is implemented in many $3D$ modelers such as; Maya, POV Ray, etc.

The de Casteljau algorithm for Bézier patches

The de Casteljau algorithm can be easily extended from curves to Bézier patches. In the case of Bézier patch $Q(u, v)$ the algorithm is applied first in the u direction and then in v direction, or conversely. It should be noticed that fixing the parameter $u = u_0$ or $v = v_0$ one obtains a Bézier curve $Q(u_0, v)$ or $Q(u, v_0)$ lying on the patch, respectively. So the de Casteljau algorithm can be applied to both curves separately. As a result one obtains the point $Q(u_0, v_0)$ that divides the patch into four Bézier subpatches.

Bézier patches as being tensor product Bézier curves, can be obtained using 4 subdivision matrices:

$$L \otimes L, L \otimes R, R \otimes L, R \otimes R, \tag{3.3}$$

where $\otimes$ denotes the so-called tensor product of matrices, that can be computed as in Example 3.5.

Example 3.5. Take the following matrices

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}.$$

Then the tensor product $A \otimes B$ is equal to:

$$A \otimes B = \begin{bmatrix} 1\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} & 0\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \\ \\ 1\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} & 2\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 1 & 4 & 2 \\ 0 & 1 & 0 & 2 \end{bmatrix}.$$

It is worth to mention that in general $A \otimes B \neq B \otimes A$.

Subdivision matrices for quadratic Bézier patches have dimensions 9×9, whereas for cubic Bézier ones 16×16. We leave the reader computing them as an easy exercise. Those matrices are also the Markov ones. So, their elements are summed up in rows to 1. Multiplication of control points matrix by subdivision matrices produces new control points characterizing four subpatches. Then the new control points of every subpatch are transformed iteratively by 4 subdivision matrices. After several iterations one can obtain a good approximation of a Bézier patch. In Fig. 3.6 an example of a bi-quadratic Bézier patch obtained via subdivision matrices is presented.

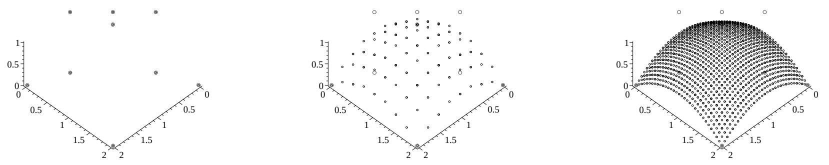


Fig. 3.6. Bi-quadratic Bézier patch obtained via subdivision. Iterations: $0, 2, 4$.

If we compute 4 tensor products $L \otimes L, L \otimes R, R \otimes L, R \otimes R$, where L and R are subdivision matrices for Chaikin's curve, then we obtain subdivision matrices for generation of a bi-quadratic Chaikin's patch (equivalent to a bi-quadratic B-spline patch) shown in Fig. 3.7. Observe, that none of control points lie on the patch.

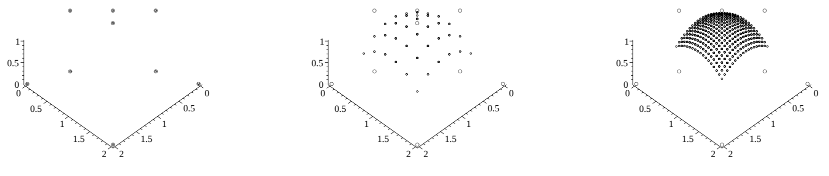


Fig. 3.7. Bi-quadratic Chaikin patch obtained via subdivision. Iterations: $0, 2, 4$.

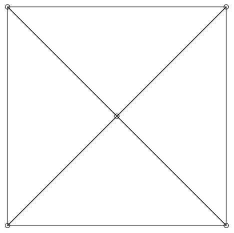
Catmul-Clark subdivision

Catmull-Clark subdivision operates on meshes of arbitrary topology. The refining algorithm works as follows.

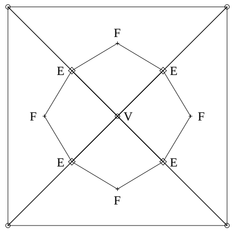
- Take a mesh of control points with an arbitrary topology. Basing on that mesh create three kinds of new points.
  - Face points denoted by F that are computed as the average of all the points defining a face.
  - Edge points denoted by E that are computed as the average of the original points defining the edge and the new face points for the faces that are adjacent to the edge.
  - Vertex points defined by V that are computed as the average of Q, $2R$ and $\frac{n-3}{3}S$, where Q is the average of the new face points adjacent to the original face point, R is the average of the midpoints of all original edges incident on the original vertex point, S is the original vertex point, and n is the number of points defining the face.
- Next, the mesh is reconnected as follows:
  - Each new face point is connected to the new edge points of the edges defining the original face.
  - Each new vertex point is connected to the new edge points of all the original edge midpoints incident on the original vertex point.

The above algorithm, for which a one round of performance illustrated on an example of a diamond-like mesh is presented in Fig. 3.8, produces refined mesh having four edges. It is known that the Catmull-Clark subdivision generates surfaces that are smooth almost everywhere excluding the so-called extraordinary points. In Fig. 3.9 the result of performing the Catmull-Clark algorithm starting from a box is presented. That algorithm is very efficient and quickly convergent producing a good approximation of the limit surface, usually after 4-5 iterations. Compare the result generated by the Catmull-Clark subdivision with a simple midpoint subdivision presented in Fig. 3.10. Midpoint subdivision does not produce smooth limit surface and its convergence is lower in comparison to the Catmull-Clark subdivision.

To create a more complicated subdivision surface, we construct a coarse polygon mesh that approximates the shape of the desired surface. Then the subdivision algorithm recursively refines the mesh producing a sequence of finer meshes which converges to a smooth limit surface. Subdivision techniques are widely used by movie studios, as Pixar, to produce high quality $3D$ elements of virtual worlds in their productions.



(a)



(b)

Fig. 3.8. Mesh: (a) initial, (b) after performing one round of Catmull-Clark subdivision.

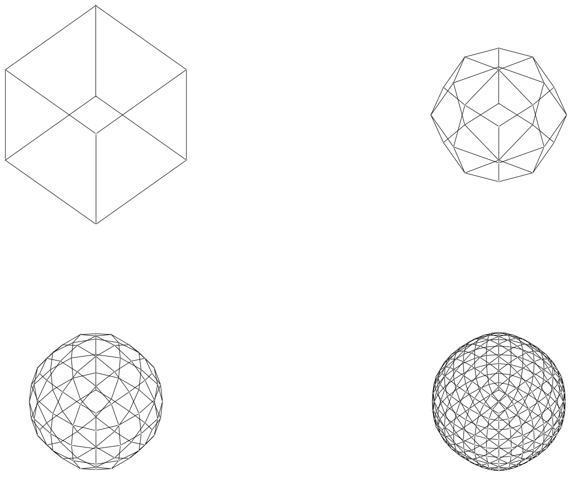


Fig. 3.9. Catmull-Clark subdivision applied to box. Iterations: $0, 1, 2, 3$.

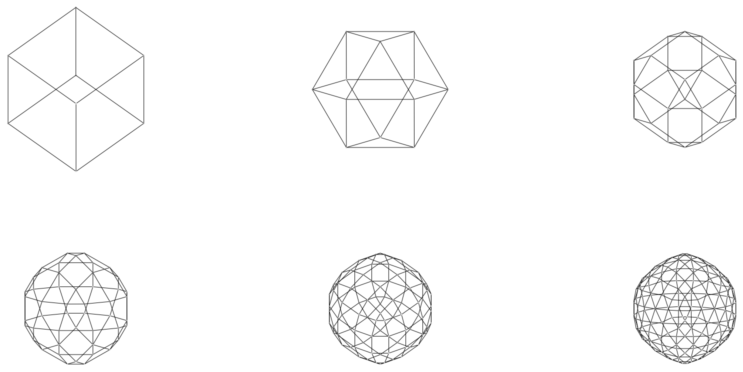


Fig. 3.10. Midpoint subdivision applied to box. Iterations, $0, 1, 2, 3, 4, 5$.

Chapter 4

Fractals

Fractals are geometrical objects that cannot be expressed with the help of functions because of their complexity. Such objects as clounds, mountains, seeshores cannot be desribed using the classical euklidean geometry. Fractals have been discovered by Mandelbrot in 1970s [13].

4.1 Properties of fractals

Fractals are characterized by the following properties:

- Self-similarity,
- Non-integer dimension,
- Can be generated recursively.

Self-similarity means that the whole object is similar to its parts. Non-integer dimension means that the dependency between the area of the planar object and its periphery are not proportional to power of 2 as in the case of classical objects, for example a square or a circle. Denote by P and l an area and a periphery of a given object in the plane, respectively. Then, for a square with a side lenght a we have:

$$\begin{cases} P = a^2, \\ l = 4a. \end{cases} \tag{4.1}$$

So, the dependency between P and l is the following:

$$P = (\frac{1}{16})l^2.$$

Similarly, for a circle with a radius r we have:

$$\begin{cases} P = \pi r^2, \\ l = 2\pi r, \end{cases} \tag{4.2}$$

and the dependency between P and l is expressed as:

$$P = (\frac{1}{4\pi})l^2.$$

From the above simple computations it is easily seen that graphical objects in the plane are two dimensional objects. Fractals behave in a different way. So, the notion of non-integer dimensionality has been needed to understand the difference between fractals and classical objects in the plane. In spite of very complicated fractal structures, they can be obtained with a help of a simple set of rules that are used repeteadly to produce fractals. So, recurrencies are typically used to generate fractals.

4.2 Fractal types

There are many different types of fractals considered in literature [1, 2, 7, 16], but one can distinguish seven main classes of fractals:

1. Classical fractals, which come from standard geometry. They can be generated with the help of iteration procedure applied to some initial set. They have been investigated at the end of XIX century and at the begining of XX century. The best known examples are: Cantor set (1883), Koch curve (1904), Sierpinski gasket (1915), Menger sponge (1926).

2. Fractals generated via iterations of polynomials with complex variables. The famous examples are: Julia set (1918) and Mandelbrot set (1980).

3. Fractals generated with the help of Iterated Function Systems *IFS*. They are obtained as the limit of iteration process. They were discovered by Barnsley (1988).

4. Fractals generated on the base of nonlinear *IFS*. Fractal flames are their good examples. They were discovered by Draves in 1992.

5. Fractals as strange attractors, which form complicated trajectories of dynamical systems. They are observed in many physical and numerical experiments. Heron's attractor (1969) is a good example of that class.

6. Fractals generated with the help of the so-called Brown's motions. As a result very interesting fractal structures are obtained that resemble clounds, fire, non-standard textures.

7. Fractals obtained with the help of L-systems, also called as Lindenmayer systems. They were discovered in 1968. L-system represents a formal grammar, that applied to some initial set gives as a result nice looking fractal plants.

In Fig. 4.1 many different types of fractals are presented. They can be obtained with the help of freeware software such as FractInt, Fractal Explorer, Apophysis, Chaosscope easily found in the Internet.

Futher, we limit our considerations only to fractals generated with the help of the so-called Iterated Functions Systems.

4.3 Iterated Functions System

A transformation $w : \mathbb{R}^2 \mapsto \mathbb{R}^2$ in the following form:

$$\begin{cases} x' = ax + by + e, \\ y' = cx + dy + f, \end{cases} \tag{4.3}$$

where $a, b, c, d, e, f \in \mathbb{R}$, is called the affine transformation. Parameters a, d are responsible for scalings according to axes OX, OY; b, c for rotations and shears; e, f for translations with respect to axes OX and OY, respectively.

The transformation w can be represented in the following equivalent matrix form:

$$[x', y'] = [x, y] \cdot \begin{bmatrix} a & c \\ b & d \end{bmatrix} + [e, f], \tag{4.4}$$

or by using homogeneous coordinates w can be further expressed as:

$$[x', y', 1] = [x, y, 1] \cdot F, \tag{4.5}$$

where

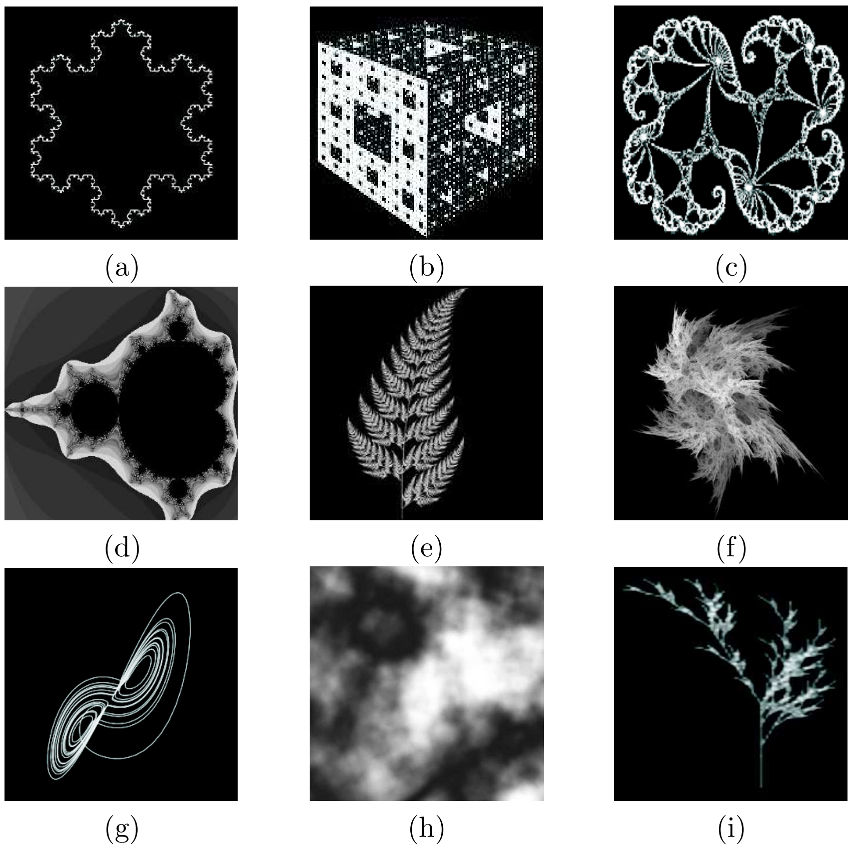


Fig. 4.1. Fractal examples: Koch snow-flake (a), Menger sponge (b), Julia set (c), Mandelbrot set (d), Barnsley fern (e), fractal flame (f), strange attractor (g), plasma fractal (h), plant created via L-system (i).

$$F = \begin{bmatrix} a & c & 0 \\ b & d & 0 \\ e & f & 1 \end{bmatrix}.$$

Of course, any affine transformation in $\mathbb{R}^2$ is chacterized uniquely by the vector of six parameters $\mathbf{v} = [a, b, c, d, e, f]$. In the similar way as in the plane, one can define the affine trasformation in the space. Any affine transformation in $\mathbb{R}^3$ is defined by the matrix of dimensions 4×4 or the equivalent vector of parameters containing 12 coefficients.

Especially important, in context of fractals, are affine transformations that diminish distance, i.e. the so-called contractions. A set of n affine contractive transformations $\{w_1, w_2, ..., w_n\}$ is called Iterated Functions System, in short *IFS*. *IFSs* are used in deterministic or random algorithms to generate fractals. It should be said that to measure distances between sets in the plane, classical Euclidean metric cannot be used. Instead of it, the so-called Hutchinson metric is applied. Having defined Hutchinson metric it is possible to use the well-known Fixed Point Banach theorem that ensures the existence of the limit when passing to infinity in an iteration process. That limit, called attractor or a fractal, is the unique one and is not dependent on a starting set. More detailed discussion about that subject can be found in [1, 2, 13].

4.4 Deterministic fractal algorithm

In Fig. 4.2 the performance of the deterministic fractal algorithm is presented. An arbitrary closed and constrained set $X \subset \mathbb{R}^2$ or in $\mathbb{R}^3$ is transformed via *IFS*, which creates reduced copies of the set X and places them according to the performance of every transformation from *IFS*. Note, that at every step of iteration in deterministic fractal algorithm all transformations are used. The obtained result is repeated over and over again in a loop transformed via *IFS*. Theoretically, that process is repeated an infinite number of times. As a result, an attractor of *IFS* is generated. In practice, the performance of several up to a dozen or so iterations is enough to obtain a good quality of the attractor approximation for the given *IFS*.

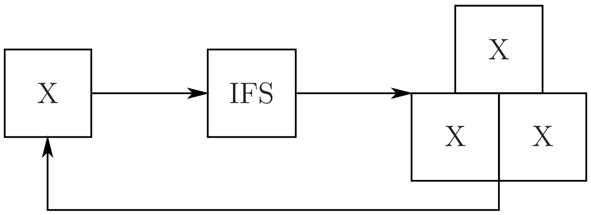


Fig. 4.2. Deterministic fractal generation algorithm.

4.5 Random fractal algorithm

Consider *IFS* consisting of n affine contractive transformations $w_1, w_2, \ldots, w_n$. In the case of deterministic algorithm at every iteration all transformations are used. It is possible to introduce a probabilistic model, in which at every iteration only one transformation is used. That transformation is drawn, as the one from *IFS*, with a fixed probability $0 < p_j < 1, j = 1, \dots, n$, such that $\sum_{j=1}^{n} p_j = 1$. Additionally, transformations in the probabilistic model transform points into points, not sets into sets as in the deterministic model. Usually, an equal probability is taken for all transformations from the *IFS* set, although a better solution is to choose probability according to the following formula:

$$p_i = \frac{|\det A_i|}{\sum_{i=1}^{n} |\det A_i|}, \tag{4.6}$$

where

$$A_i = \left[\begin{array}{cc} a_i & c_i \\ b_i & d_i \end{array} \right], \quad i = 1, ..., n.$$

In the matrix A_i, elements a_i, d_i are scaling parameters and b_i, c_i are parameters responsible for rotation and shears. Such a choice of probabilities assure uniform fractal coverage by points generated during the random process. The starting point can be an arbitrary one from the set X. If the starting point belongs to the fractal, then all further points generated during the random process belong to the fractal. In the opposite case, in which the starting point does not belong to the fractal, some finite number of points do not belong to the fractal, either. So then, usually several intial points from many thousands generated by random algorithm are thrown out to get better quality of generated fractals. In Fig. 4.3 the performance of the random fractal algorithm is presented.

For both algorithms, deterministic and random ones, from iteration to iteration one obtains better and better approximation of a generated fractal. That iteration process is stopped after a finite number of iterations. Usually, in the case of random algorithm several thousands of iterations are needed. The random algorithm is stopped when changes in the shape of the fractal are not notizable with respect to the finite resolution of computer devices used to its visualization and because of the limited recognition possibilities of details by human sight.

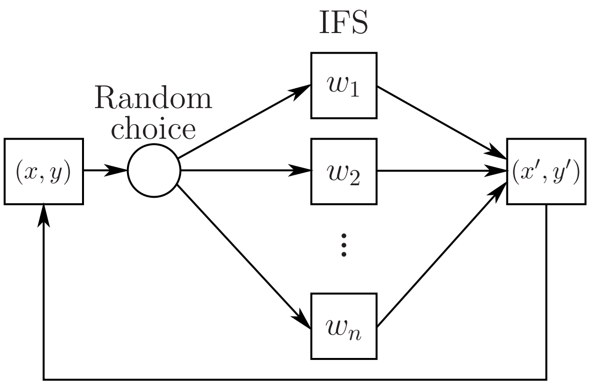


Fig. 4.3. Random fractal algorithm.

4.6 Examples of *IFS* generated fractals

Begin with the standard Sierpinski triangle. Observe, that initial triangle T shown in Fig. 4.4a can be covered by three smaller triangles $1, 2, 3$ that are similar to T. So, similarity of T to its parts can be described using three following transformations:

$$w_1(x,y) := (\tfrac{1}{2}x, \tfrac{1}{2}y), \quad w_2(x,y) := (\tfrac{1}{2}x + \tfrac{1}{2}, \tfrac{1}{2}y), \quad w_3(x,y) := (\tfrac{1}{2}x + \tfrac{1}{4}, \tfrac{1}{2}y + \tfrac{1}{2}). \tag{4.7}$$

These transformations scale an input by the factor $\frac{1}{2}$. Additionally, w_2 translates by the vector $\left[\frac{1}{2}, 0\right]$, whereas w_3 by the vector $\left[\frac{1}{4}, \frac{1}{2}\right]$. It is easily seen that those trasformations are contractive, so then they determine *IFS* that can be used to generate the fractal – Sierpinski triangle (Fig. 4.4b) – using deterministic or random fractal algorithm.

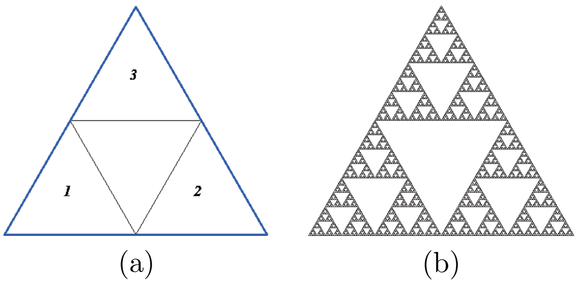


Fig. 4.4. Self-similarity of traingle T (a) and Sierpinski triangle (b).

The next fractal example – Koch curve is presented in Fig. 4.5. Koch curve arrises as the result of the performance of 4 affine contractive transformations. Each of them diminishes the initial segment by scale factor $\frac{1}{3}$ producing 4 smaller copies of that segment. Next, the first transformation places the first copy on the left-hand side of the initial segment, the second transformation translates the second copy by vector $[\frac{1}{3}, 0]$ to the right and rotates it by angle 60° in the anticlockwise direction, the third transformation translates the third copy by vector $[\frac{2}{3}, 0]$ to the right and rotates it by angle 120° in the anticlockwise direction, whereas the forth transformation places the fourth copy on the right-hand side of the initial segment. Formally, four transformations defining *IFS* have the following form:

$$w_1 = [1/3, 0, 0, 1/3, 0, 0]\,, \quad w_2 = \left[1/6, -\sqrt{3}/6, \sqrt{3}/6, 1/6, 1/3, 0\right],$$
$$w_3 = \left[-1/6, \sqrt{3}/6, \sqrt{3}/6, 1/6, 2/3, 0\right], \quad w_4 = [1/3, 0, 0, 1/3, 2/3, 0]$$

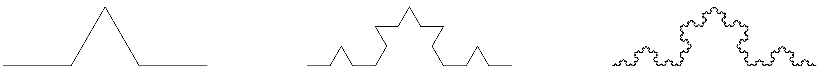


Fig. 4.5. Koch curve. Iterations: 1,2,5.

4.7 Subdivision and fractals

In this section we will show a suprising connection between subdivision and fractals. Namely, any curve or patch obtained via subdivision can be generated in a fractal way. For curves *IFS* consist of two transformations, whereas for patches consist of four transformations.

For example, *IFS* generating a quadratic Bézier curve has the following form:

$$IFS = \left\{P^{-1} \cdot L \cdot P, P^{-1} \cdot R \cdot P\right\},$$

where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}, R = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, P = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix},$$

L, R are subdivision matrices, P is the matrix of control points and P^{-1} is the inverse matrix to P.

In Fig. 4.6 a process of fractal generation of a quadratic Bézier curve is presented. The deterministic fractal algorithm starting from a triangle was used.

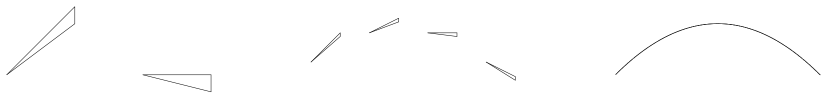


Fig. 4.6. Quadratic Bézier curve generated fractally. Iterations: 1,2,8.

For a bi-quadratic Bézier patch *IFS* has the form, as follows:

$$IFS = \left\{P^{-1} \cdot L \otimes L \cdot P, P^{-1} L \otimes \cdot R \cdot P, P^{-1} \cdot R \otimes L \cdot P, P^{-1} R \otimes \cdot R \cdot P\right\},$$

where L, R are subdivision matrices as above, $\otimes$ denotes tensor product, P is the matrix of control points completed by arbitrary elements to create some non-singular quadratic matrix of dimensions 9×9 and P^{-1} is the inverse matrix to P.

The fact that subdivision may produce smooth curves, patches and fractals is very important. That is why the limit graphical objects obtained via subdivision inherit advantages of both functional and fractal approaches. Fractals, in spite of their complexity, provide simple algorithms to generate them, whereas functions with control points provide a very easy way to maniputate their shapes. Those two features are required from methods used in computer graphics and computer aided design. So, without any doubt, popularity of subdivision methods will be steadily growing up in the future.

Chapter 5

Visual photorealism

In earlier chapters only aspects related to the geometry have been considered. But they are not sufficient to obtain realistic appearance of $3D$ objects and $3D$ scenes. Their severe geometry should be enriched by colours, lights and textures. Well rendered photorealistic $3D$ scene, as in Fig. 5.1, should be compared to a good photo. So, it is easier for us to obtain good photorealistic results if we know the basic rules of photography.

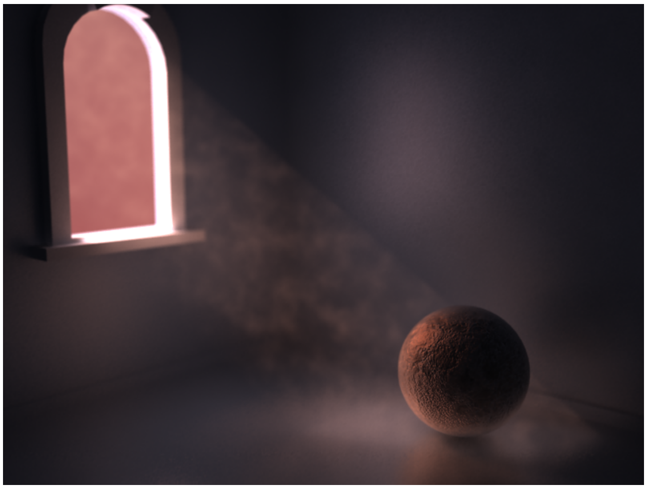


Fig. 5.1. Photorealistic rendering.

5.1 Light

Light in computer graphics is not only a light source, but also a shading model and a few other parameters which will be described in this section. The easy way to shade an object is by using a flat shading or Gouraund shading model which is replaced by a better and more realistic Phong lighting model. The Gouraund model assumes that $3D$ object is represented by polygons and the value of lighting is linked to verticles of all the object's polygons. In this model the object which should be smooth will be sharp, because the most lighting point can be only on one of the polygon's verticles. If the object is a sphere, the edges of polygons will be well marked. This model computes normals only for polygon's verticles and edges are shaded using a linear interpolation. Every edge creates a gradient if values of connected vertices will be different. This kind of problem solves the Phong lighting model, which is slower but creates much better effects for concave and convex objects. In this case, the difference between Gouraund and Phong models is that Phong computes all the needed normals for each polygon, not only polygon's vertices. By using this method the brightest point can also be placed in the middle or any other place of the polygon, not only on one of the vertices. The Phong lighting model creates a smooth transition between polygons so every sphere is smooth, too. The lighting model includes also reflection and refraction of light, but this kind of effects will be described in section 5.2 because they have a connection with materials. Nowadays, the best computer graphics programs use Phong or more complicated Blinn shader.

Next, if the lighting model is chosen it has to create a light source and specify shadows that it makes. Basic light types are Point Lights, which emit light in every direction like in Fig. 5.2. Often Spot Lights type are used to minimize the area of lighting. The Spot Light looks like a cone in Fig. 5.3, that is getting larger with the distance gained between the light source and the object (target). Cylindrical Light from Fig. 5.4 is one of the alternatives for the Spot Light, because the field of lighting is stationary, no matter how far the light source (from the object) is. Knowledge on the mentioned types of lights is sufficient to create a photorealistic scene. It should be remembered to set more than one, but not too many light sources, because rendering will take longer and the obtained effects will probably be worse. For the best results in the basic scene it two or three Point or Spot Lights should be used. The majority of light sources have such properties that enable their gradual disappearance with distance and smooth transition between the point with good lighting and the one badly lit. Shadows are created with lights, so the properties are set in lights. Shadows can have sharp or smooth contours like it is shown in Fig. 5.5, and they are dependend on the object's material which works as a filter. In this case a good example to describe is material like glass for which we should take reflection into account. That will be described in the next section. The whole scene will look differently depending on the light declaration, choices of the best shading, the light intensity and colour. More red and yellow lights in the scene create warm lighting, which is natural for a bulb and sun-light. More blue colour in light imitates a fluorescent lamp and photo-flash lamp as it is shown in Fig. 5.6.

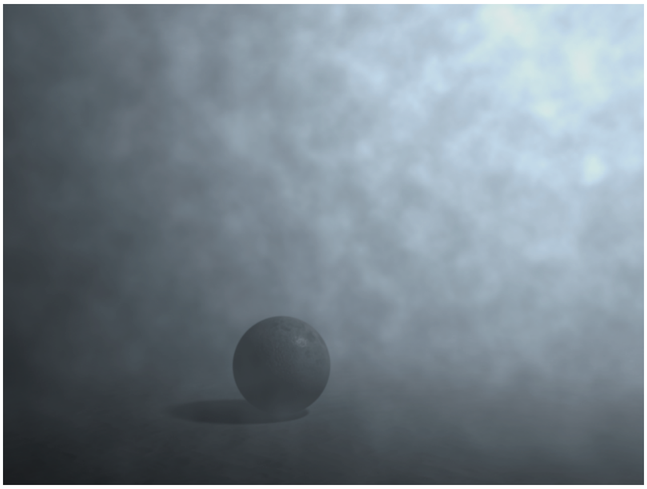


Fig. 5.2. Point Light.

The colour of light depends on the colour and material of the object in the scene. The effect of the so-called bouncing light will be described in the section about rendering. That effect depends on the kind of the renderer used (Mental Ray, V-Ray, etc.) and its properties.

5.2 Materials

Complex programs for computer graphics possess a wide range of possibilities to create any material for every object in the scene. At first, the type of lighting should be chosen, next one of few kinds of interactions with the object must be set. The most basic, but the least photorealistic is the Ambient Light (environment light). Ambient values should be small comparing

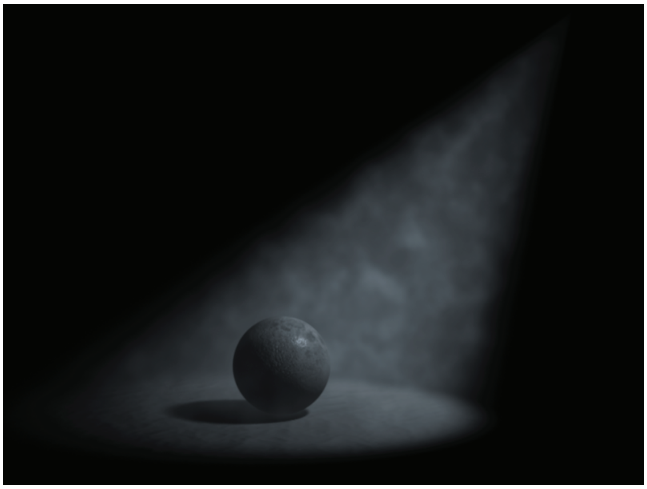


Fig. 5.3. Spot Light.

to other settings. The second parameter is Diffuse which creates the object lighting and the colour of material from which it is made of. The value should be always higher than Ambient colour, but if the object absorbs light, Diffuse value should be small and vice versa. Next, the parameter related to the distracted reflection is Specular. It fixes how large the reflection area on material should be. The area of reflection and its focus are getting smaller with the Specular value. For example, an orange fruit has a small Specular value because it has soft, porous and irregular skin. If the object should look like an apple then the Specular and Diffuse values are higher because this fruit is smooth and hard. Soft and hard materials are presented in Fig. 5.7. For glass and polished metals Diffuse and Specular values should be as large as possible. Other used option is Self Illumination which decides about self-contained light of the object. In this case the object becomes a source of light, but computer graphics programs decide if other objects treat that object as the light source.

If the object is a polished metal, then the material should have mirror specular (Reflection). This kind of object in reflection parameters should have two values. One of them represents perpendicular reflection and the second one border reflection. The second value should be smaller than the first one. If the object is characterized by a perfect reflection material, then the colour of material is invisible and the object imitates a mirror. Naturally, larger angle between the object and the observer gives less reflection. Glass and water are one of the most difficult materials to create. Glass is not a material with

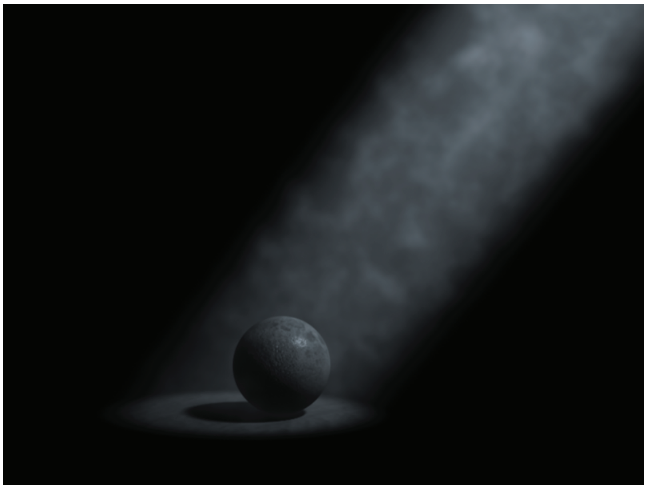


Fig. 5.4. Cylindrical Light.

only low opacity, it has the parameter called the Index of Refraction (IOR). It means how the object lying behind glass is deformed, physically it means how glass refracts a ray of light. Deformation depends on IOR, shapes and bumps of the material (there will be more informations about it in section 5.3). The IOR is the characteristic physical parameter e.g. for glass it is equal to 1.52, for crown (silicon-potassium glass) – 1.66, for flint (silicon-lead glass), for water – 1.33. The second parameter of real glass material is a reflection which is, presented according to the angle of look, exactly the same as the refraction is. Fig. 5.8 presents glass and chrome balls. The most showy effects in glass creation produce focus and diffuse rays of light which pass through the glass object. That effect is called Caustics and it depends on material, light and mostly on the renderer parameters. More details on that effect will be presented in section 5.5.

5.3 Textures

Section describing textures could be a part of the section 5.2, because textures also give a realistic appearance of material. Yet we decided to describe them in a separate section to show how complicated the problem is.

Textures can simulate surface quality of a material. The process called texturing places any $2D$ image on $3D$ object using shape information or user

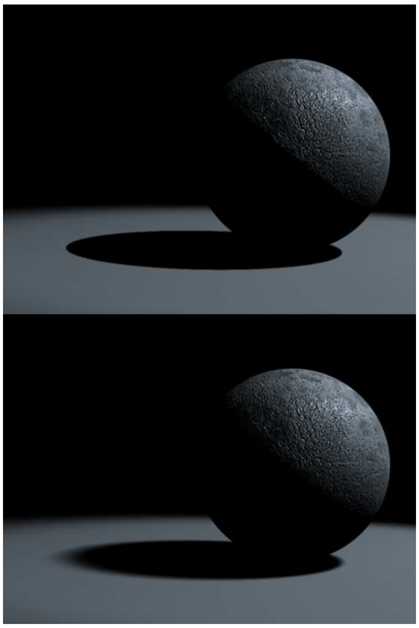


Fig. 5.5. Sharp and smooth shadows.

directives. By this technique a bicubic surface can simulate a part of the fabric and a sphere can imitate, for example a stone. Moreover, the textured object can be seen to be more complicated that it really is. Sometimes, for example some real parts of a church building textured by painting artificial shadows leads to exaggerated, fake effects. Using special textures, without any changes in geometry of the object, its surface can look as the one with depressions and grooves called Bumps, Wrinkles, etc. Texturing is much faster than geometry modification, but objects like a sphere with Bumps will still have rounder edges without any structure modification, as shown in Fig. 5.9. Technically, textures create Bumps by changing discreetly direction of normal vectors of surfaces. The process changes shading and lighting of the object, so human perception takes Bumps like real irregular shape geometry. In rendering some points of a shape are translated a little higher or lower in relation to their original position. That makes this process fast.

Textures can be created in many ways. One of them uses mathematical description. Fractal structures perfectly imitate clouds, Perlin noise can greatly simulate smoke. Other methods for realistic textures creation are based on bitmaps prepared in any $2D$ computer graphics program. Coloured bitmaps

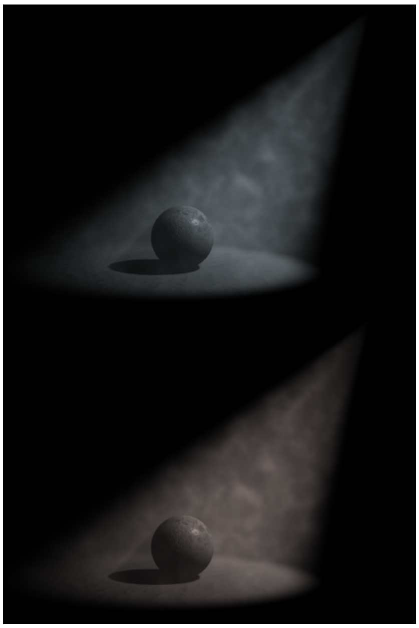


Fig. 5.6. Cold and warm light colours.

are used to cover objects. Similar bitmaps, but in grayscale with large contrast can be applied to create Bumps like grooves, scratches and grain coarse. A map of this kind is shown in Fig. 5.10. Parameters of Bumps tell us about the difference in depth between white and black colours. If the Bump bitmap has many white and black areas without smooth transition between them, then the obtained texture will by very irregular with distortions. Gentle transition between white and black regions makes textures smoother.

Usually, the best textures are photos where all their regions are correctly and uniformly lighted. When using an image in the loop, its right edge should be fitted to the left one, and the top edge should be matched to the bottom one. The image can be multiplied, but if the number of multiplications is too large, then a characteristic undesirable pattern appears. To avoid this effect, the texture could be composed of two various size images, from which the second should have low opacity and alpha channel to differentiate the first image multiplications.

Making good-looking textures take a lot of time. For example, to create a trousers texture it is needed to have two trousers photos, one taken from the front and the second one from the back side without background. Next, using

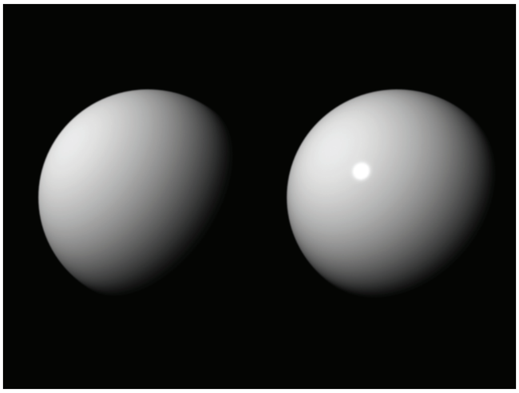


Fig. 5.7. Soft and hard material.

any $2D$ computer graphics program both photos should be joined creating one. At the end opposite edges of the image must be matched like in Fig. 5.11. Many textures can be created by this method. Computer graphics programs often offer a tool which can be used to match automatically textures to objects.

Textures can be composed from several images having different properties. This technique is used, e.g. in car texturing. In this case the texture consist of varnish, mud and scratch images, where the last one can be also a Bump map.

5.4 Camera

A camera in $3D$ graphics is not a visible object in the space. It declares only the observer's position. The camera is needed because it creates $2D$ images of objects from the abstract $3D$ scene. An image can be the $3D$ one only if it is projected from two projectors equiped with polarization filters similar to the one presented in Fig. 5.12. The $3D$ image is visible using special glasses with the same filters. This kind of image can be created using the same two cameras located at a small distance between them (like human eyes). One of the images is shown in Fig. 5.13. The observer to see the $3D$ image has to look at it with crossed eyes. However, every process of creating an image depends on hitting points on virtual screen located between objects and the observer. This process will be described in details in section 5.5.

Virtual camera has many parameters, some of them have origin in photography. We start with orthogonal projection and perspective view. In mod-

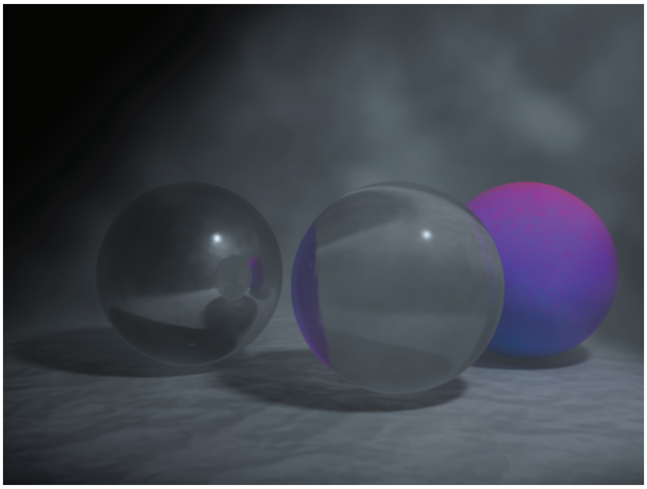


Fig. 5.8. Glass and metal balls.

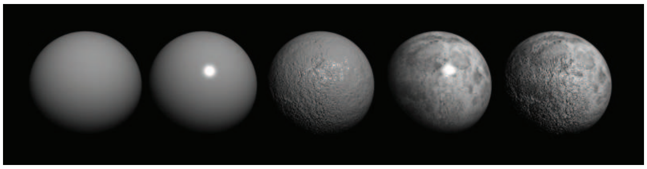


Fig. 5.9. Sphere with Moon texture.

elling, the most popular and comfortable is orthogonal projection shown in Fig. 5.14(a), because the same object situated at different distances from the observer has still the same size. This kind of view is used in draught and in geometry. It is useful in engineering projects but is completely not photorealistic. Human eyes see objects in perspective (presented in Fig. 5.14(b)), so then receive objects placed far as smaller than those which are closer to the camera. Similarly, a perspective view is used in rendering. So the distance between the observer and the objects situated in the scene is a very important parameter.

The first of the parameters that comes from photography is the angle of view. It is known that human angle of view is about 45° (which corresponds to the focal length 50mm) but photography often uses lenses with different angles. In photographic art standard lenses with human focal length are rarely used because it is not the aim to show the world exactly as a man can see it. Wide-

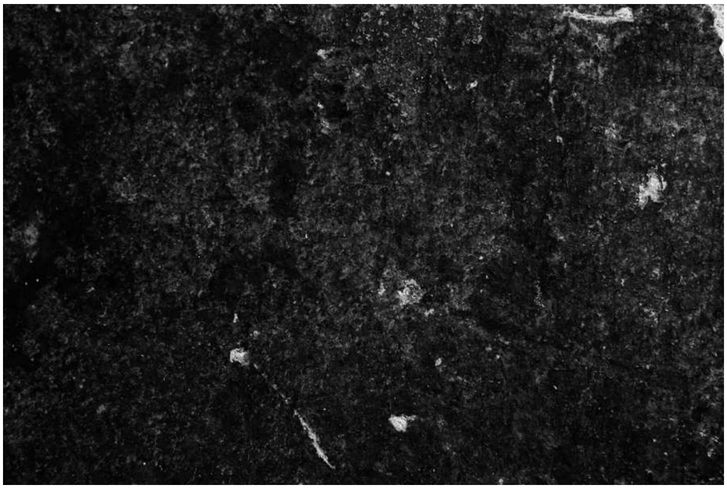


Fig. 5.10. Bump bitmap.

angled lenses are standard for compact cameras (28mm or less) which enable to catch a wide area in short distances. In computer graphics the problem with the distance between the camera and the object does not exists at all. By changing the focal length of the virtual camera one can increase rendering similarity to real photos. The use of various levels of focal length in static images or animations can create different visual effects. Wide-angled lenses break the geometry of huge objects. For example, the image of a building, presented in Fig. 5.15, has rounded walls, and this effect is getting larger with the increase of the lens angle. Extremal example delivers the fish-eye lens which has even 220° (6mm) angle of view. It can be simulated by computer graphics programs. Other example is a telephoto lens which has the small angle varying from 28° down to even 2° which corresponds to 80-1200mm focal length. Differences between two or more objects situated at large distances will be more evident. For standard lenses and telephoto ones the foreground is looking similar but the second plan looks differently – for standard lenses it is wider than for the telephoto. Moreover, using telephoto lenses one can take good quality photos from large distance.

In computer animation camera parameters can be changed, so physical camera is getting closer to the first plan objects and the angle lens is getting larger but animated objects look as still. In this action the background is changing because it is getting wider without changing the foreground. This kind of a trick can be used in computer graphics. The next effect creating

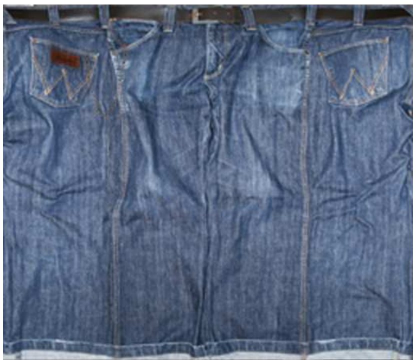


Fig. 5.11. Jeans texture.

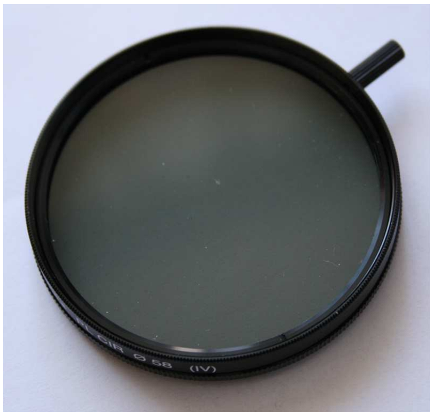


Fig. 5.12. Polarization circular filter.

more realistic renderings is the depth of the field. In real world a lens without this effect does not exist. Even human eye cannot focus on two distant objects simultaneously. This effect occurs when almost the same picture is rendered a few times, more than 10 passes. The image quality is getting better while the number of passes grows (12-36 is enough). Other optional effect creates coloured orbs on the rendered image. It is lens effect and it appears in a real picture when any strong source of light, like the sun or even large power headlight, is placed on the boundary of the visible area, exactly like in Fig. 5.16. Not always this effect makes the image more realistic. Usually, it is better to ignore the lens effect by keeping the observer focused on the scene elements. In computer graphics programs the parameter orbs can be enabled or reduced.

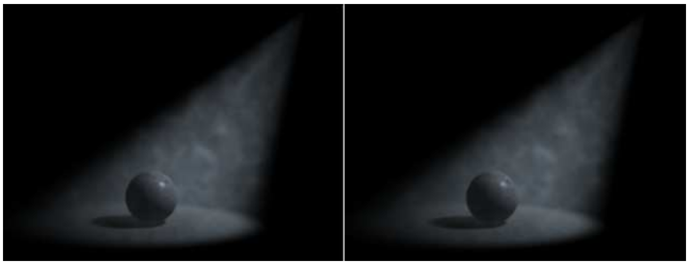


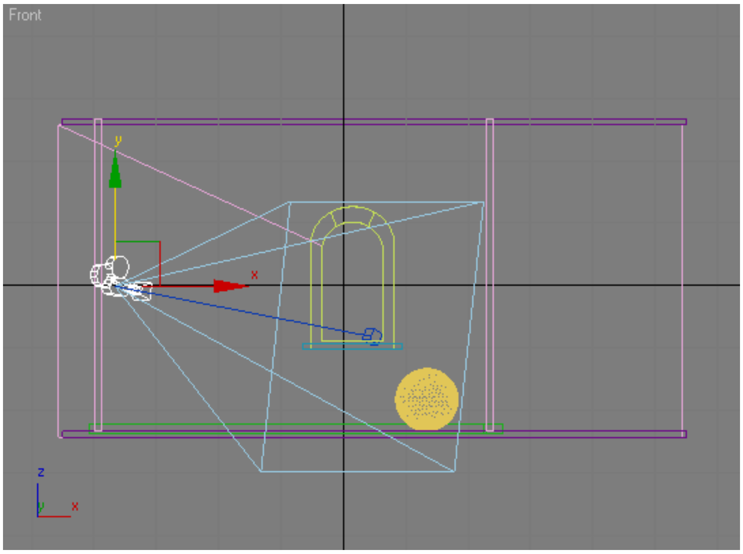
Fig. 5.13. Cross-eye rendering.

In computer graphics placing a camera in the scene is realized with the help of the special kind of objects. In some programs there are two different cameras. The first one is the free camera which can be easily rotated and translated but its dynamic focal and target points are difficult to set. The second one is the target camera with look-at point as the part of the object. The target point can be translated independently on the camera and this point can be used as the focal point. To translate this object, the camera and the camera target should be selected. That is the only weakness of this kind of the camera object.

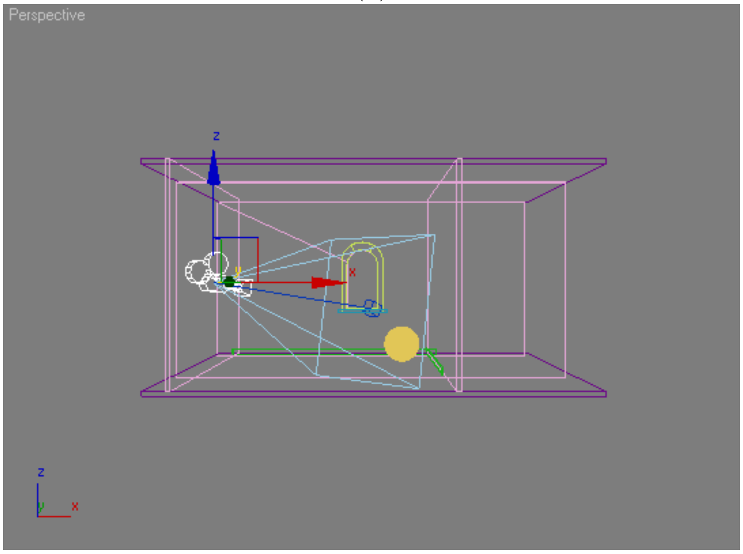
5.5 Rendering

Rendering is a complex process which creates $2D$ image of the $3D$ abstract space of the scene. To start this process, except for objects creation, the scene must contain at least one light source and a camera. The location of the camera decides what objects will appear on the rendered image. The location of the light and the type of object materials decide whether the object will be visible or not on the rendered image. There are two methods of rendering. The first and longer one with special effects, like reflection and refraction, is Raytracing. The second one, for a long time used in computer games, because of its speed, is Radiosity. Further we will describe those two methods of rendering.

Algorithm of Raytracing is simple but it requires much time-consuming computations. Nowadays, games are using this method because GPU on graphic cards are much more powerful than a few years ago. Even 3D Studio MAX 2010 allows rendering in real time on viewports using this method. Every pixel on a computer screen is tested by at least one ray, so time of rendering grows with resolution of the image. The camera is located at the point where the observer is placed and every ray starts its course from this



(a)



(b)

Fig. 5.14. Camera in: (a) orthogonal projection, (b) perspective projection.

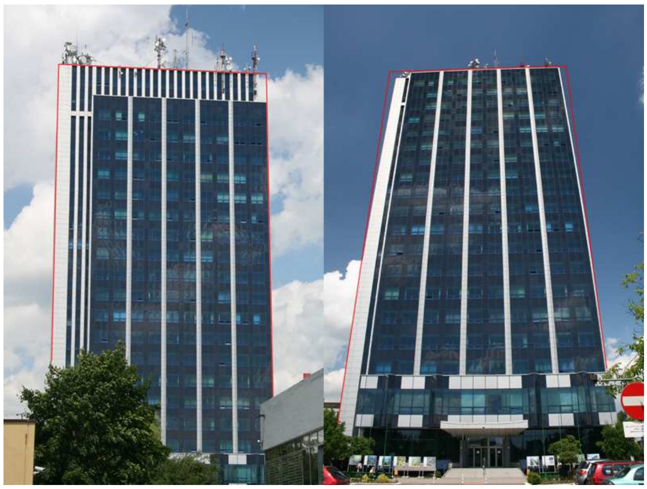


Fig. 5.15. Photos shot using 55 and 18mm focal length lens.

point. Next, every ray must pass through his own pixel on virtual plane on which the rendered image will be created.

Every pixel is passed by at least one ray. Next it is tested whether the ray intersects the surface of any object in the scene. If yes, then the renderer checks if the point of intersection is in light or in dark and computes the colour of the pixel on the virtual plane basing on object material and light colour. If the render is more complicated, then the ray can bounce from the surface, if it reflects (metal) or passes through it, if the surface refracts (glass). Raytracing can split computation between few CPUs or GPUs because every pixel is computed separately which is the disadvantage of this rendering method. Raytracing cannot render dispersion of light because it operates on single rays which are not connected. Also, rendering is performed only for one point of view, so if the observer changes his position, then all computations must be performed from the very beginning.

Radiosity is a technique which works in a different way in comparison to Raytracing. The disadvantage of Raytracing is the advantage of Radiosity. Renderer computes dispersion of light so it cannot use rays. Computations are performed for the whole scene, so it does not matter where the observer is. Every object is split into patches and every patch has energy which describes how much light it absorbs and how much light it diffuses. Images using this renderer are very fotorealistic because one light source can lighten objects which are even beyond the lighting area. The light is bouncing from

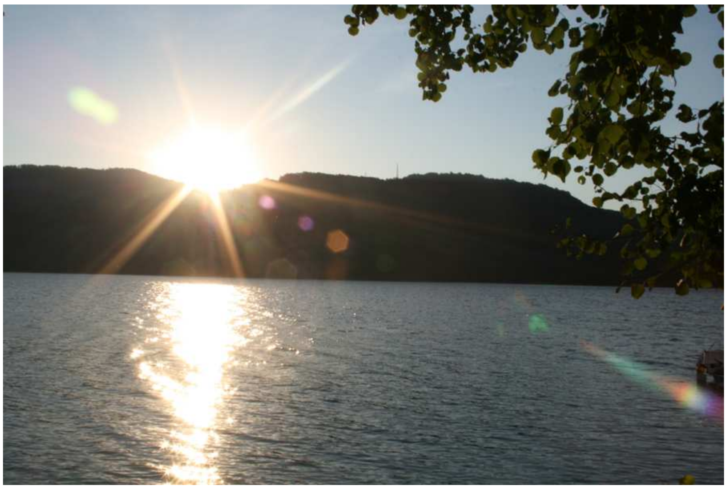


Fig. 5.16. Photo with natural lens effect.

the objects and is exchanging among them. If the object is red it absorbs green and blue lights and bounce red, so other white object will look like red too, if the object will be in the range of the reflex light. Not only material of the object determines colour of this object in the rendered image. Radiosity cannot render refraction and reflection because this kind of effects needs Raytracing. This is the difference between those two techniques, but most of computer graphics programs can use both these methods at the same time. So, Radiosity will compute realistic dispersion of light whereas Raytracing will create refraction and reflection surfaces of objects.

There are a few other effects which help to create very photorealistic images and animations. One of them is Caustics known also as Photon mapping, shown in Fig. 5.17. This technique creates photorealistic glass and similar material interactions with light. More complicated renders like Mental Ray or V-Ray can create this effect by passing through an object thousands or even millions photons to simulate refraction and dispersion of light. After long computations a glass object creates focus and diffuses visible light rays that are falling on other objects. This effect can perfectly imitate the focus light after passing the lens.

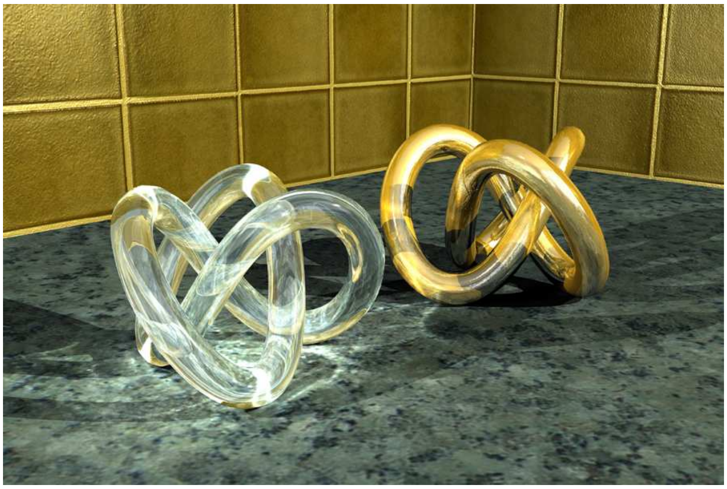


Fig. 5.17. Rendering with caustics.

5.6 Colour models

Computer graphics uses many colour models, but the most popular ones are RGB, CMY and HSV. The first two models are used as the techniques of colours representation, whereas the last one enables a user to choose precisely the required colour. RGB model consists of three colours Red, Green and Blue. This is the so-called an additive colour model which means that a sum of all three colours gives white as in Fig. 5.18(a). Also a sum of two of those colours gives cyan, magenta or yellow, respectively. RGB model was created for displaying colours by electronic devices. For example LCD monitor has those three colours which are masks made by liquid crystals to display a suitable mix of colours. If all colours are masked with the maximum value the LCD monitor is black, while all colours shine with the maximum value the monitor displays white colour. Most of digital cameras and image scanners use this colour model because CMOS images sensor has those three filters to separate intensity of every colour and add them together on a display device. Analogously, colourful digital photos are also based on the same colour model or similar (sRGB or Adobe RGB).

To represent an image on white paper there must be used a negative of RGB model which is subtractive one. It means that colours of this model added together give black. This is a CMY model whose name comes from the initials of Cyan, Magenta and Yellow. This model is shown in Fig. 5.18(b).

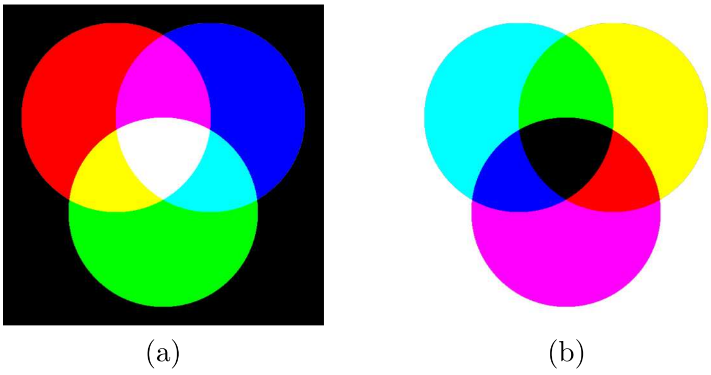


Fig. 5.18. RGB model (a) and CMY model (b).

Similarly as in the RGB model, two colours from the CMY model give one from the RGB model. This model is known also as CMYK where K is black because the letter B is reserved by Blue. Cyan, magenta and yellow, added together, theoretically should give black colour, but technically those three colours are not perfect so black created using CMY is not so deep as separated black. This model is used by printer devices and for example in CMOS of Canon family digital cameras. This technology allows to make photos with less light, so Canon cameras have less noise on dark photos.

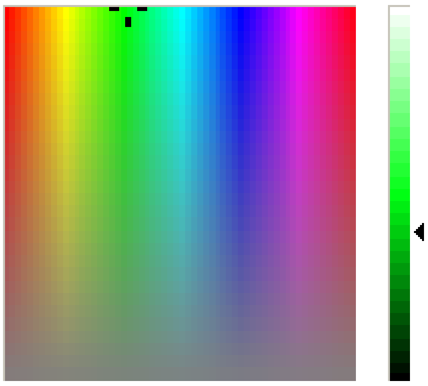


Fig. 5.19. HSV model.

The mentioned above two models were created for projecting and printing devices, but for the user it is easier to use the HSV model. HSV comes from the first letters of Hue, Saturation and Value, respectively. Hue is the clean colour without features like light or dark in Fig. 5.19 represented by the upper border of the square. Saturation describes the region between clean hue colour and gray colour. In Fig. 5.19 it is the region between the upper and lower border of the square. The value is the region between black and white shown on the bar on the right of the square in Fig. 5.19. Using those three variables the user can easily choose the colour which he exactly needs. It is not so simple in RGB or CMY model, but many classic painters use CMY model to mix colours together to get the required one. In computer graphics HSV model can simply make that green grass on a photo is greener by adding saturation and teeth of a photographed person are whiter when saturation of yellow and red are set smaller than normally.

Chapter 6

Software

In the previous chapters we have talked about the basic geometrical objects i.e. curves, patches, fractals, their mathematical background and about visual photorealism. In this chapter we will present software in which all the mentioned notions are used. First we will present a POV-Ray scripting language. With the help of this language we can render photorealistic images using ray tracing technique. Next, we will present a MayaVi package which is used for visualization of scientific data. Finally, we will present a program for visualization and analysis of molecules, proteins, etc. called Deep View or Swiss-PdbViewer. All the programs presented in this chapter are multiplatform applications and they are available for free.

6.1 POV Ray

POV-Ray is a free source graphic program without interaction modelling interface. It is not as popular and as simple as Maya or 3DS Max but it is free and good for start. POV-Ray delivers some kind of a script language called "Scene Description Language" which helps to form a user spatial imagination. Creation of a scene in POV-Ray is better to do in two steps. First it is good to create all objects and then, after placing them in the scene, describe the objects materials, lighting and camera effects, because only graphic preview is available after rendering. POV-Ray language is very simple but allows to use average effects and to obtain nice renderings like in Fig. 6.1. We will try to describe basics of this program.

Fig. 6.1. An image created with the help of POV-Ray.

6.1.1 Objects creation

Every object in POV-Ray is created in the virtual world described by three axes. Horizontal X-axis leads to the right, vertical Y-axis leads up and Z-axis leads into the screen. So then, the coordinate system used by POV-Ray is the left-handed one.

Every object creation starts up from a point in $3D$ space. If it is a sphere, this point will be the centre of the sphere, if it is a box the point will be one of the box corners. Codes for a sphere and a box covered by colours is given in listing 6.1.

Listing 6.1.

```
sphere { <-1.5, 0, 0>, 1
  pigment { color <0, 1, 1> }}

box {<.5, -1, -0.5>, <2.5, 1, .5>
  pigment { color <1, 0, 1> }}
```

Colour consists of three components <Red, Green, Blue> from which each takes values from the interval $[0, 1]$. When we have a colour with the values of components from a set $\{0, \ldots, 255\}$ then every component value should be divided by 255 to obtain a value from $[0, 1]$.

POV-Ray needs a light and a camera to render the scene. A simple light source consist of $3D$ point and colour. The camera has two parameters: the first one is camera location and the second is the target point. These two elements are presented in the listing 6.2. Using the light and the camera we obtain the rendering shown in Fig. 6.2.

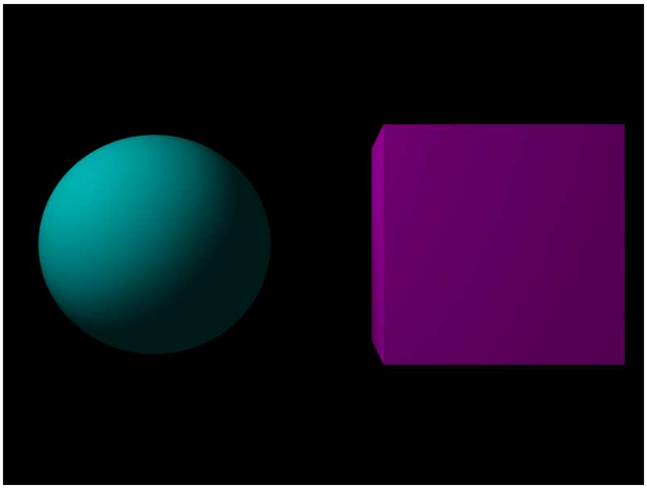


Fig. 6.2. Basic scene.

Listing 6.2.

```
camera { location <0, 0, −4.5>
         look_at  <0, 0,    0> }

light_source { <−3, 2, −3>
  color <1,1,1> }
```

Objects like a cylinder or a cone are created by setting two points as centres of two bases and the same two radiuses, in the case of a cylinder, or different in the case of a truncated cone. Other predefined objects are: a bicubic patch, objects obtained by rotating the points around an axis called Surface of Revolution or by rotating a spline called Lathe. In listing 6.3 an example of Surface of Revolution based on eight points is given and the obtained rendering is presented in Fig. 6.3.

Listing 6.3.

```
sor {8,
  <0, 0>
  <1.7, 0>
  <0.75, 0.75>
```

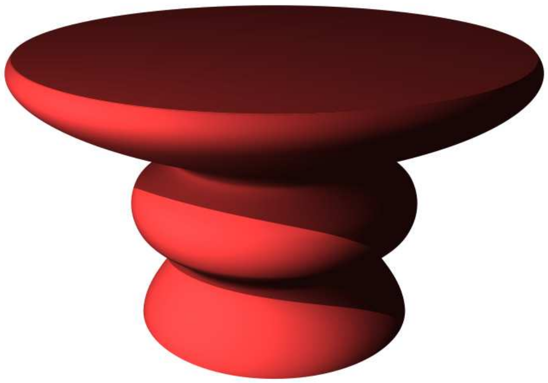


Fig. 6.3. Surface of Revolution.

```
    <1, 1>
    <1, 1.25>
    <0.75, 1.5>
    <1, 2>
    <0.8, 1.9>
    pigment { color <1, 0.25, 0.25> }

    translate <0, −1, 0>
    rotate <0, 0, 180> }
```

To all the listings the camera and light sources should be added. The object after creation can be modified by transformations like **translate** to move the object, **rotate** to rotate it about one or more axes and **scale** to make the object smaller or bigger.

More complex objects are created using boolean algebra. This technique is called Constructive Solid Geometry known as CSG in short. It allows to obtain more complex objects from two or more using the following keywords: **union**, **difference**, **intersection**, **inverse** and **merge**. Listing 6.4 explains how to cut a sphere from a box and the obtained shape is presented in Fig. 6.4.

Listing 6.4.

```
difference{

  box { <−1, −1, −1>, <1, 1, 1>
    pigment { color <1,0,1> }}
```

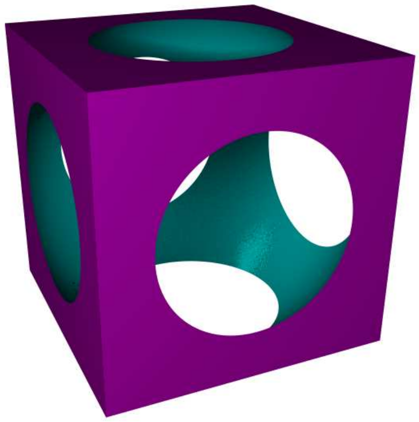


Fig. 6.4. Difference of a box and sphere.

```
    sphere { <0, 0, 0>, 1.2
      pigment { color <0,1,1> }}
}
```

All CSG operations are shown in Fig. 6.5.

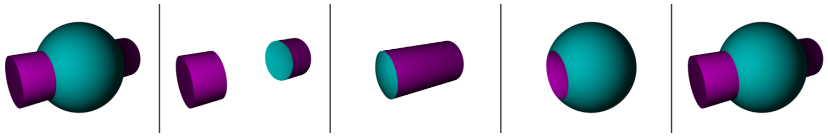


Fig. 6.5. **union**, **difference**, **intersection**, **inverse**, **merge**.

To automatize some objects creation one can use decision instruction such as **if** or loop like **while** or for. Listing 6.5 demonstrates how to use the **while** loop and the result of its rendering is shown in Fig. 6.6.

Listing 6.5.

```
#declare counter = 0;

#while (counter < 20)

  sphere { <counter , 0, 0>, 0.3
    pigment { color <0.25, 1, 0.25> }}
```

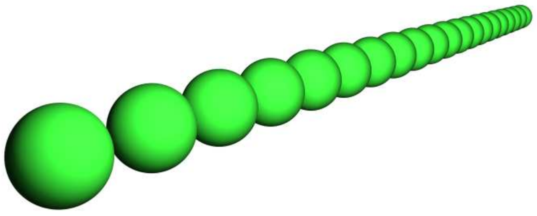


Fig. 6.6. 20 balls created with the help of **while** loop.

```
    #declare counter = counter + 1;

#end
```

6.1.2 Materials and textures

Materials in POV-Ray require a few parameters to set. But, if some of them are missing, then POV-Ray automatically sets them to the default values. For example, listing 6.6 with the added camera and light source produces a black box. In this case the material colour pigment is set as the black one. If we do not define the background or global ambient lighting then the rendered image will be completely black.

Listing 6.6.

```
box { <−1, −1, −1>, <1, 1, 1> }
```

So setting e.g. red colour of the box or any other object can be done with the help of listing 6.7 or 6.8.

Listing 6.7.

```
box { <−1, −1, −1>, <1, 1, 1>
    pigment { color <1,0,1> }}
```

Listing 6.8.

```
box { <−1, −1, −1>, <1, 1, 1>
    pigment { color red 1 blue 1 }}
```

Pigment colour in the above code is described by three parameters Red, Green and Blue. If not all of the colour values are defined, then the missing ones (as in the listing 6.8 Green parameter) are set by POV-Ray as default 0. Pigment colour has two additional parameters **filter** and **transmit**. So it has up to five parameters, as shown in listing 6.9.

Listing 6.9.

```
pigment { color rgbft <red, green, blue, filter, transmit> }
```

transmit is responsible for opacity and **filter** except opacity, deforms a little rays which pass through the surface of an object. Those parameters are not both necessary simultaneously so **color** can have the following syntax **rgb**, **rgbf**, **rgbt** or **rgbft**. The parameters described up to now are the basic ones and they do not assure photorealistic apperance of the object. To obtain photorealism we need an advanced material definition consisting of a few parts. The first one is the pigment which describes colour or opacity of material. The next is two-parameter **finish** (with ambient and diffuse parameters) which creates material as described in section 5.2. The **ambient** parameter sets interaction between material and ambient light. It should be very small or 0 because ambient light has no source, so it is everywhere with the same value. If the scene is lighted by ambient light, then every object looks flat and it is hard to see its real shape. The **diffuse** parameter sets the material reaction with light. It should be larger than ambient and high if the material is bright. Using **ambient** and **diffuse** parameters one can create bright materials but without shine. Shiness of the material can be obtained with the help of **phong** and **phong_size** parameters. The **phong** sets shining power, whereas **phong_size** decides how large area on the shape has glitter. Hard objects often have narrow area and bright shining, soft ones inversely. If the object should be metallic than it is characterized by the keyword **metallic** without any value. An example of metallic object is presented in Fig. 6.7. More realistic metallic appearance of the object can be achieved using **reflection**. If the **reflection** is constant on the surface of the object than one parameter is used, in the other case two values are needed.

Further posibilities of influence on apperance of the material can be obtained with the help of the keyword **normal**. It creates bumps on the surface without geometric modification. There are available a few kinds of **normal**

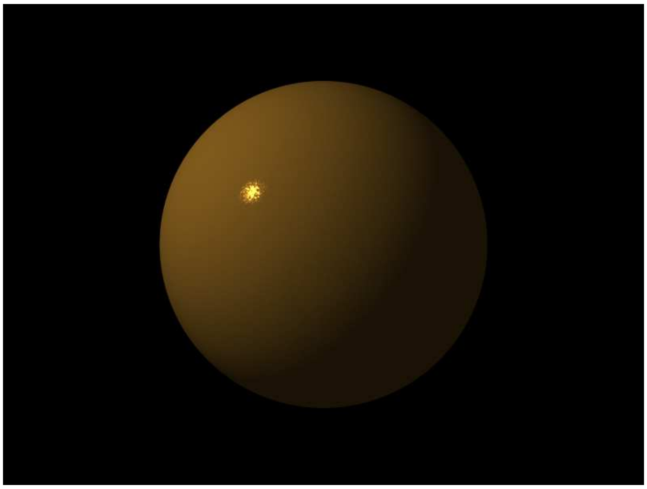


Fig. 6.7. Metallic (golden) ball with bumps.

modification with the help of **bumps**, **dents**, **ripples**, **waves**, etc. All of them can be scaled. Additionaly, for glass and water Index of Refraction that characterizes interior of the object can be defined. Using listing 6.10 the iced cube, as in Fig. 6.8, can be obtained.

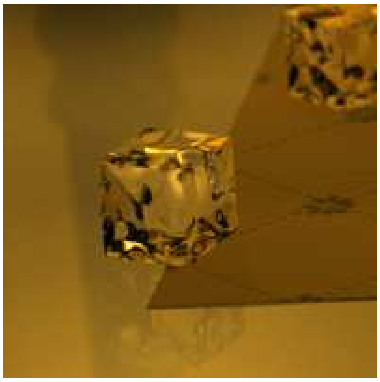


Fig. 6.8. Ice cube.

Listing 6.10.

```
box { <0, 0, 0>, <1, 1, 1>

  material { texture { pigment { rgbf <.98, .98, .98, .99> }
```

```
      finish  { ambient 0.5 diffuse 0.1
        reflection {0.1, 0.5}
        specular .9 roughness 0.0003 phong 1 phong_size 400 }

    normal { bumps 0.9 scale 0.05 }}

    interior{ ior 1.33 }}}
```

Default textures in POV-Ray have mathematical description. Most of them use **pigment_map** to create multicolour material. For example, the code from listing 6.11 creates the sky pigment shown in Fig. 6.9, whereas the code from listing 6.12 creates wooden texture without bumps.

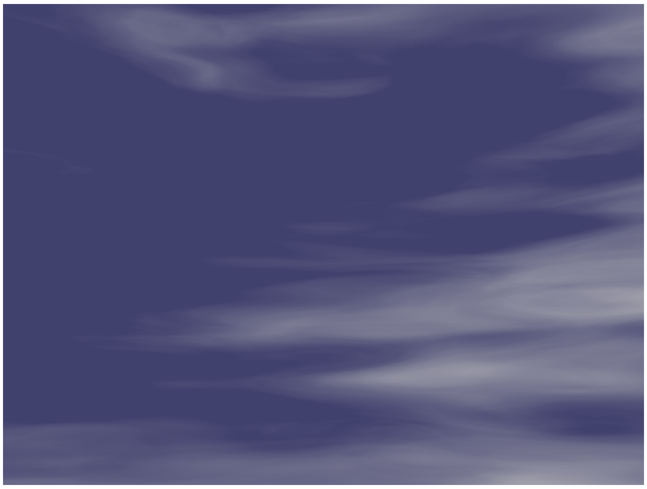


Fig. 6.9. Sky sphere.

Listing 6.11.

```
  pigment {
      wrinkles
      turbulence 0.65
      octaves 6
      omega 0.7
      lambda 2
      color_map {
          [0.0, 0.1   color red 0.85 green 0.85 blue 0.85
                      color red 0.75 green 0.75 blue 0.75]
          [0.1, 0.5   color red 0.75 green 0.75 blue 0.75
                      color rgb <0.258, 0.258, 0.435>  ]
          [0.5, 1.001 color rgb <0.258, 0.258, 0.435>
```

```
                        color rgb <0.258, 0.258, 0.435> ]
      }
      scale <6, 1, 6>
}
```

Listing 6.12.

```
color_map {
    [0.0, 0.3 color rgbt <0.25, 0.10, 0.10, 0.00>
              color rgbt <0.25, 0.10, 0.10, 0.40>]
    [0.3, 0.5 color rgbt <0.25, 0.10, 0.10, 0.40>
              color rgbt <0.60, 0.15, 0.10, 1.00>]
    [0.5, 0.7 color rgbt <0.60, 0.15, 0.10, 1.00>
              color rgbt <0.25, 0.10, 0.10, 0.40>]
    [0.7, 1.0 color rgbt <0.60, 0.15, 0.10, 0.40>
              color rgbt <0.25, 0.10, 0.10, 0.00>]
}
```

In POV-Ray effects like fire, smoke, etc. are created as special materials. Listing 6.13 defines for example a fire sphere, where **density_map** describes every layer of the fire object.

Listing 6.13.

```
sphere{ 0, 1 pigment { rgbt 1 } hollow
  interior{ media{ emission 15
  density{ spherical density_map {
    [0 rgb <0, 0, 0>]
    [0.2 rgb <0.2, 0, 0>]
    [0.5 rgb <0.75, 0.1, 0>]
    [0.75 rgb <1, 1, 0>]
    [1 rgb <1, 1, 0.1>]}}}}}
```

The object defined in listing 6.13 can be translated, rotated, scaled and also modified by using CSG technique. Fig. 6.10 presents an example of a candle flame created by this method.

Textures or bumps can be generated using any jpeg or png image with the help of code from listing 6.14.

Listing 6.14.

```
box { <-10, -10, 6>, <22, 16, 5>
  texture{pigment {image_map {jpeg "file.jpg" } scale 6}
  normal {bump_map {jpeg "file_b.jpg" bump_size 0.5} scale 6}
  finish {reflection {0.5} phong 0.75 phong_size 400}}
  rotate<0, 90, 0> translate<0.97,0,0.46>
}
```

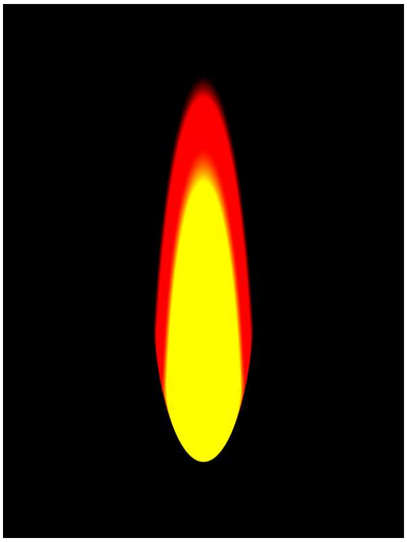


Fig. 6.10. Candle flame.

That code creates the textured box with the texture coming from two jpeg files – file.jpg and file_b.jpg. The first one is a colour image of the material and the second one presents the greyscale image of irregular surfaces with scratches. Images applied as textures can be scaled, translated and rotated in order to be precisely placed on any textured object. Here, it should be stressed that in some cases mathematical description creates better textures than the images.

6.1.3 Lights and camera

POV-Ray delivers different light sources. Every light source is defined by its location and some of them additionally by a target point. Basic lights create strong and dark shadow that can be smoothed using special parameters. A simple point light can be created by listing 6.15.

Listing 6.15.

```
light_source {
  <1, 2, -3>
  color <1, 1, 1> }
```

This type of light is analogical to point light from section 5.1. Often it is enough to create photorealistic scene but more spectacular effects can be obtained with a spot light from listing 6.16.

Listing 6.16.

```
light_source {
   <1, 2, -3>
   color rgb <1,1,1>
   spotlight
   point_at <0, 0, 0>
   radius 5
   tightness 50
   falloff 8 }
```

By some modification of the point light it becomes a spot light which can light up only a chosen area. Not having defined localization point, the spot light has a destination point which is called **point_at**. The destination point is the centre of the circular lighted area. Objects situated outside that area are not lighted. The lighted area is characterized by the following parameters: **tightness** and **falloff**. **tightness** describes how softly or hardly light becomes dark on the boundary of the circle. For low **tightness** value the boundary of the lighted area will be soft and looking more naturally. **falloff** is the intensity of falloff radius.

Every kind of light can have more than necessary settings. More advanced parameters are **fade_distance**, **fade_power** or **jitter**. First two describe fade, in other way it is the lighted area. Those parameters can make the rendering more photorealistic. The sun creates light with low fade and illuminates a large area, whereas bulb or candle light has medium and large fade, respectively. First setting is **fade_distance** which sets the area of lighting. Outside that area everything will be dark. The **fade_power** sets how much light wanes.

The above parameters set only lights but do not describe shadows. A light can be **shadowless** and it is useful while the scene contains many lights and some of the shadows can break the composition of the scene. To a many shadows divert the observer's sight from the objects. Other lights still have strong and dark shadows which are not photorealistic. To create a good looking light, naturally fading and creating smooth shadows, a few parameters as in listing 6.17 should be used.

Listing 6.17.

```
light_source {
   <0, 0.65, 0>
   color rgb <0.95, 0.7, 0.2>
   fade_distance 2
   fade_power 1
```

```
    area_light <3, 0, 0>, <0, 0, 3>, 9, 9
    adaptive 0.5
    jitter }
```

The **area_light** creates something like a few lights illuminating simultaneously the scene which gives smooth shadows and makes them less dark. **jitter** is the parameter which set rendering more realistic – fade of light is smoother. The **jitter** parameter causes that rendering will take much longer but the obtained effect will be much better.

POV-Ray has few types of cameras. Creation of any kind of camera looks similar, only parameters used are different. In section 5.4 we described in details camera settings which can also be used in POV-Ray. Usually, in photography a focal length is used, whereas in computer graphics programs **angle** is more comfortable to apply. Basic camera settings are **location** point and **look_at** point which is the target one. The code defining a simple camera is presented in listing 6.18.

Listing 6.18.

```
camera {
  location   <−2.2, 0.8, −3.75>
  look_at    <−0.4, −0.3,  0.0> }
```

The camera from the listing 6.18 has set the default angle that can be easily changed by appending its value. To create more realistic renderings camera parameters must include depth of the field which is described by three values, as presented in listing 6.19. The obtained rendering of this scene shows Fig. 6.11.

Listing 6.19.

```
camera {
  location   <−2.2, 0.8, −3.75>
  look_at    <−0.4, −0.3,  0.0>
  angle 46
  focal_point <0.5, 0, 0>
  aperture 0.2
  blur_samples 50
}
```

The keyword **focal_point** sets point with perfect focus and it can be different of **look_at** point. The focal point can be imagined not only as one point but as the plane created by points lying at the same distance from the camera, those with the perfect focus. Objects lying before and after the plane will be

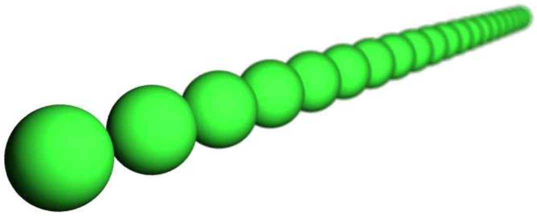


Fig. 6.11. Camera blur.

rendered with blur. The value of blur is set by **aperture** which is a characteristic feature of the lens.

Blur is larger if the value of lens aperture is smaller but in POV-Ray blur is stronger when the value of **aperture** is higher. The keyword **blur_samples** sets the quality of the render. A poor blur effect appears for blur values 3 or 4 but better results can be obtained for 25 or higher ones. If the value of **blur_samples** is getting higher then the time of rendering is getting longer.

POV-Ray has many types of cameras from a fish-eye to spherical camera. But good renderings can be obtained using simple camera with 45 angle and blur.

6.1.4 Rendering

Rendering in POV-Ray is controlled by *.ini file presented in listing 6.20.

Listing 6.20.

```
Input_File_Name="file.pov"
Width=800
Height=600
Antialias=On
Antialias_Depth=9
Quality=11
```

The first line sets a project file name which will be rendered, so it is not necessary to open this file in POV-Ray. Next parameters fix the image size, width (x) and height (y). The third line sets antialiasing on/off and the value from the next line determines the quality of the antialiasing. Rendering with antialiasing on is slow but the obtained quality of the rendered image is very good. The last line sets the quality to maximum. For example, quality fixed as 0 or 1 determine rendering with only ambient light and the object with only colours, 6 and 7 even compute photons and 9 to 11 produce renders with all defined effects like radiosity, etc. The above listed code is executed only once producing the rendered image.

for and **while** loops automatize project rendering but in every execution of the code project parameters are set to the default. To obtain renderings for animation the parameter called **clock** should be used in *.ini files.

Lines in the listing 6.21 from *.ini file allows creating 25 frames that can be used in animation.

Listing 6.21.

```
(...)
Quality=11
Initial_Frame=1
Final_Frame=25
Initial_Clock=1
Final_Clock=5
```

The sphere from listing 6.22 will be translated in every frame. When the rendering is over, bitmap files numbered from file001.bmp to file025.bmp can be found in the project directory. Those files can be further joined with the help of e.g. BMP to AVI movie creator or other programs to obtain *.avi animation files.

Listing 6.22.

```
sphere { <0, -0.5, 0>, 0.5
   (...)
   translate < 2 + clock, 0, 0> }
```

6.2 MayaVi

MayaVi is a general-purpose 3D scientific visualization package created by Prabhu Ramachandran from the Department of Aerospace Engineering at Indian Institute of Technology Bombay. It uses the Visualization Toolkit

(VTK) [26] and it is entirely written in Python. The main features of MayaVi are:

- visualization of scalar, vector and tensor data in 3D,
- easy scriptability using Python,
- easy extendability via custom modules, data filters, etc.,
- graphical interfaces and dialogs with a focus on usability,
- supporting of several file formats, e.g. VTK, plot3d, VRML2,
- saving of visualization,
- saving of scene in a variety of image formats, e.g. png, jpeg, VRML, eps,
- functionality for rapid scientific plotting via mlab.

Fig. 6.12 presents the user interface of MayaVi.

6.2.1 Installation

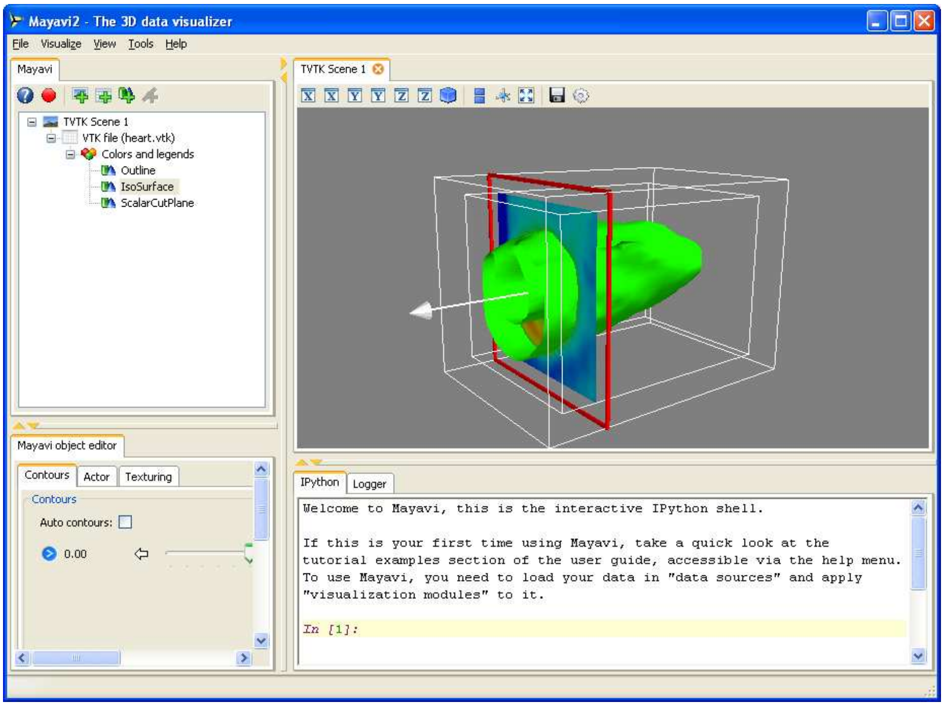
The easiest way to install the MayaVi package is to install a full Python distribution. Under Windows we can choose an Enthought Python Distribution (EPD) [6] or a Python(x,y) [17]. The EPD distribution is free only for educational purposes, for other purposes we need to purchase the subscription. The full Python distribution of EPD is also available for MacOSX and Red Hat Enterprise Linux 3 and 4. In Debian and Ubuntu MayaVi is packaged. The Python(x,y) distribution is also available for Ubuntu.

The second possibility to install the MayaVi is to install it from the Python Eggs. The instructions for this kind of installation can be found in the users manual of MayaVi [21].

6.2.2 Pipeline model

Fig. 6.13 presents the pipeline model of the MayaVi package.

The process of a visualization begins with raw data entering the filtering stage. Usually, filtering means selective removal, but here the term is used in more general meaning, namely to transform data from one form into another one, e.g. from $2D$ image data representing a heightmap into a triangle mesh representing a terrain.



```
Welcome to Mayavi, this is the interactive IPython shell.

If this is your first time using Mayavi, take a quick look at the
tutorial examples section of the user guide, accessible via the help menu.
To use Mayavi, you need to load your data in "data sources" and apply
"visualization modules" to it.

In [1]:
```

Fig. 6.12. MayaVi user interface.

$$\begin{tikzcd}
\text{Raw data} \arrow[r] & \boxed{\text{Filtering}} \arrow[r] & \boxed{\text{Mapping}} \arrow[r] & \boxed{\text{Rendering}} \arrow[r] & \text{Image}
\end{tikzcd}$$

Fig. 6.13. Pipeline model.

After filtering the mapping stage comes which produces an abstract visualization object. An abstract visualization object is an imaginary object with attributes such as size, colour, transparency and so on. Thus, for example the height might control the colour of an object, ranging from blue for very low points to red for very high points.

The final stage is to render the abstract visualization object into an image. Here the object might need to be rotated or scaled and it could be viewed with perspective or projection.

6.2.3 Filters

In MayaVi the filtering stage is available through the Filters (menu *Visualize → Filters*). The list of these filters and a brief description of them is following:

- *CellDerivatives* – computes derivatives of input point/vector data and outputs these as cell data,
- *CellToPointData* – converts cell data to point data for the active data,
- *Contour* – computes contour of the input dataset,
- *CutPlane* – slices the input dataset with a cut plane,
- *DecimatePro* – reduces the number of triangles in a triangular mesh by approximating the original mesh,
- *Delaunay2D* – executes a $2D$ Delaunay triangulation,
- *Delaunay3D* – executes a $3D$ Delaunay triangulation,
- *Elevation Filter* – creates scalar data corresponding to the elevation of the data points along a line,
- *Extract Edges* – the filter for extracting edges from any data,
- *Extract Grid* – allows to select a part of a structured grid,
- *Extract Tensor Components* – the filter for extracting components from a tensor field,
- *Extract Unstructured Grid* – allows to select a part of an unstructured grid,
- *Extract Vector Norm* – computes the Euclidean norm of the input vector data,
- *Extract Vector Components* – extracts vector components from vector data,
- *Gaussian Splatter* – the filter for splatting point into a volume with Gaussian distribution,

- *Greedy Terrain Decimation* – the filter for approximating a heightmap ($2D$ image data) with a triangle mesh with the minimum number of triangles,
- *Change ImageData information* – changes the origin, spacing and extents of an image dataset,
- *Probe data onto image data* – samples arbitrary datasets onto an image dataset,
- *Mask Points* – selectively passes the input points downstream. It is used to subsample the input points,
- *PointToCellData* – converts data located on the points to data located on the cells,
- *Compute Normals* – the filter for computing the normal vectors to the surface,
- *Quadric Decimation* – reduces the number of triangles in a mesh,
- *Select Output* – chooses the output of the source that should be used,
- *SetActiveAttribute* – sets the active attribute (scalar, vector, tensor) to use,
- *Transform Data* – performs translation, rotation and scaling to the input data,
- *Threshold* – the filter that thresholds the input data,
- *TriangleFilter* – converts input polygons and triangle strips to triangles,
- *Tube* – turns lines into tubes,
- *UserDefined* – creates a user defined filter,
- *Vorticity* – computes the vorticity (curl) of input vector field,
- *Warp Scalar* – the filter for warping the input data along particular direction with a scale specified by the local scalar value,
- *Warp Vector* – the filter for warping the input data along the point vector attribute scaled as per a scale factor.

6.2.4 Modules

The mapping stage of the MayaVi pipeline is done through the Modules (menu *Visualize* → *Modules*). The list of available modules and a brief description of them is following:

- *Axes* – draws axes,
- *ContourGridPlane* – this module allows to take a slice of the input grid data and view contours of the data,
- *CustomGridPlane* – similar to the ContourGridPlane but with more flexibility,
- *Glyph* – displays different types of glyphs at the input points,
- *GridPlane* – displays a simple grid plane,
- *HyperStreamline* – the module that integrates through a tensor field to generate a hyperstreamline,
- *ImageActor* – the module for viewing image data,
- *ImagePlaneWidget* – the module for viewing image data along a cut,
- *IsoSurface* – the module that allows a user to make contours (isosurfaces) of input volumetric data,
- *Labels* – displays labels for active dataset or active module,
- *Orientation Axes* – displays small axes which indicate the position of the co-ordinate axes,
- *Outline* – draws an outline (bounding box) for the given data,
- *Scalar Cut Plane* – creates a cut plane of any input data along an implicit plane and plots the data,
- *Slice Unstructured Grid* – the module creates a slice of the unstructured grid data and shows the cells that intersect or touch the slice,
- *StructuredGridOutline* – draws a grid outline for structured grids,
- *Streamline* – draws streamlines for given vector data,

- *Surface* – draws a surface for input data,
- *TensorGlyph* – displays tensor glyphs at the input points,
- *Text* – displays a given text on the scene,
- *VectorCutPlane* – creates a cut plane of the input data along an implicit cut plane and places glyphs according to the vector field data,
- *Vectors* – displays different types of glyphs oriented and coloured as per vector data at the input data,
- *Volume* – visualizes scalar fields using volumetric visualization techniques,
- *WarpVectorCutPlane* – creates a cut plane of the input data along an implicit cut plane and wraps it according to the vector field data.

6.2.5 Interaction with the scene

In MayaVi there are three different ways to interact with the scene:

- scene toolbar,
- mouse,
- keyboard.

Fig. 6.14 presents the scene toolbar. The first seven buttons on the toolbar are responsible for the view of the camera. The buttons with the letters X, Y or Z set the view along the $-/+$ X, Y or Z axes, respectively. The seventh button sets the isometric view of the scene. Beside the view buttons there are: button for toggling on/off the parallel projection, button for toggling on/off the axes indicator (this is the same as the *Orientation Axes* module), button for displaying the scene on the full screen, button for saving a snapshot of the scene. The last button is responsible for a configuration of the scene. Here we can change the foreground/background colour, change the magnification scale, set the quality of the jpeg image, change settings of the lighting of the scene (change the number of lights, add/remove a light from the scene, change the settings of each light, etc.).

There are two modes of a mouse interaction. The first one is a camera mode (the default mode). In this mode the mouse operates on the camera. In

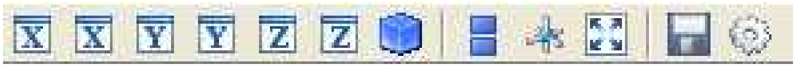


Fig. 6.14. Scene toolbar.

the second mode, the actor mode, mouse operates on the actor (object) the mouse is currently above. In each of the modes we can do several things with holding down a mouse button and dragging:

- dragging the mouse with the left mouse button will rotate the camera/actor; when we additionally hold down the Shift ⇧ key on the keyboard we will pan the scene, holding down the Ctrl key will rotate around the camera's focal point,
- dragging up/down the mouse with the right mouse button will zoom in/out,
- rotating the mouse wheel will zoom in/out.

To interact with the scene using a keyboard we can use following keys:

- a – switch to the actor mode,
- c – switch to the camera mode,
- e , q , Esc – exit the full screen mode and return to the window view,
- f – move camera's focal point to the current mouse position,
- l – lights configuration,
- p – pick the data at the current mouse point,
- r – reset the camera settings,
- s – save the scene to an image,
- = / + – zoom in,
- − – zoom out,
- ← , → , ↑ , ↓ – rotate the camera; when Shift ⇧ key is pressed the camera is panned.

6.2.6 Examples

Till this moment we have only talked about the features and possibilities of the MayaVi package. In this section we will show some examples of using MayaVi. Let us start with the examples provided with the package.

Example 6.1. When we want only to play with the program a good choice is to visualize one of the parametric surfaces given in the program. After the start of MayaVi we choose menu *File* → *Load Data* → *Create Parametric surface source*. After that we will see that a new node in the tree on the left appears – *ParametricSurface*. We click on the new node and in MayaVi object editor we can choose which surface we want to visualize. Let us choose the torus surface for which the parametric equations are following:

$$\begin{cases} x(u,v) = (c + a\cos v)\cos u, \\ y(u,v) = (c + a\cos v)\sin u, \\ z(u,v) = a\sin v, \end{cases} \tag{6.1}$$

where $u, v \in [0, 2\pi]$, $c > 0$ is the radius from the center to the middle of the ring (Ring radius) and $a > 0$ is the radius of the cross-section of the ring of the torus (Cross section radius). We can change the values of the ring radius, cross section radius and the minimal, maximal values for u, v parameters in the MayaVi object editor. These values will define the look of the torus. To see the torus we click on the node with *ParametricSurface* and then we choose menu *Visualize* → *Modules* → *Surface* or we can double click on the *Add module or filter* node below the *ParametricSurface* node and then double click on the *Surface* at the *Visualization modules* tab. Now we can interact with the scene as we described it in section 6.2.5. To change the colours of the torus we click on the *Colors and legends* node and in MayaVi object editor we click on the *Lut mode* button, then we choose one of the predefined modes also we can change the *Number of colors* used. When we look at the properties of the *Surface* module in the MayaVi object editor we have a tab called *Actor*. In this tab we can turn on/off the surface (*Visibility* checkbox), turn on/off the *Scalar visibility*, change the *Representation* of the surface to: points, wireframe, surface, change the *Line width*, etc.

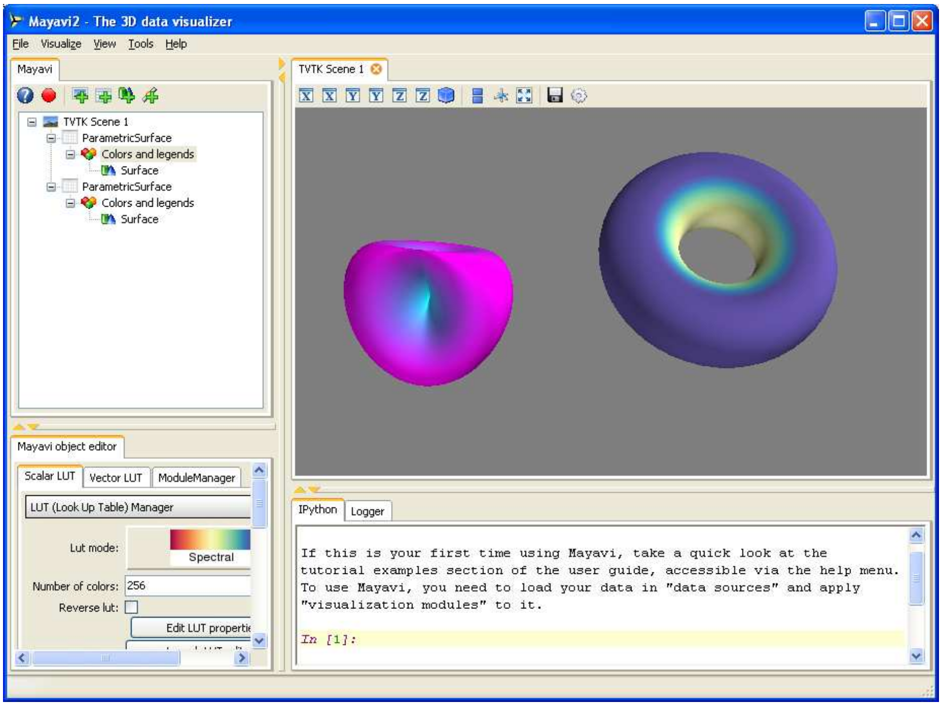
In the same scene there can be many visualizations of different data sources. Now we show how to add a second parametric surface to the same scene. To add the next data source we simply click on the root node (*TVTK Scene* node), then we go to the menu *File* → *Load Data* → *Create Parametric surface source*. After that in the tree appears a new *ParametricSurface* node. This time we change the surface to the cap. This is a cross-cap surface for which the parametric equations are following:

$$\begin{cases} x(u,v) = \cos u \sin 2v, \\ y(u,v) = \sin u \sin 2v, \\ z(u,v) = \cos^2 v - \cos^2 u \sin^2 v, \end{cases} \tag{6.2}$$

where $u, v \in [0, \pi]$. Then we add *Surface* module to the parametric surface in the same way as in the case of torus. We see that the torus and the cap overlap. To change the position of the torus or the cap we can use one of two methods: the *Transform Data* filter or the interaction with the scene. Let us see how the second method works. First we go to the actor mode by pressing the $\boxed{\text{a}}$, then we press and hold the $\boxed{\text{Shift} \Uparrow}$. Next we click and hold the left mouse button on the cap surface and then we drag the surface where we want. When this has been done we release the mouse button, $\boxed{\text{Shift} \Uparrow}$ and go to the camera mode by pressing the $\boxed{\text{c}}$. Figure 6.15 presents the visualization of the torus and the cross-cap surface.

Example 6.2. In this example we will see how to create visualization from Fig. 6.12. The data for this visualization is provided with the MayaVi package. To load the data we choose menu *File* → *Load data* → *Open file...*, then we search for the *heart.vtk* file. The vtk file format is the standard format used by MayaVi and we will talk about it in section 6.2.8. A new node *VTK file (heart.vtk)* should appear in the scene tree. Now we need to add some modules so we double click in the tree on *Add module or filter* node. Then we double click on the *Outline*, *IsoSurface* and *Scalar Cut Plane* modules. When we click on the *IsoSurface* module in the MayaVi object editor we see that we can change the value for contours. In the *Actor* tab, similar to the *Surface* module, we can turn on/off the *Visibility* of the IsoSurface, change the *Representation* of the IsoSurface, etc. When we click and hold the left mouse button on the cut plane in the scene and then move, we see that the cut plane moves and shows the actual cut of the IsoSurface. We can change the direction of the cut. To do this we click on the *ScalarCutPlane* node and in the MayaVi object editor we can choose *Normal to x/y/z* axis which makes the cut plane perpendicular to the x, y or z axis. Of course we can give the cut plane any angle by simply clicking and holding the left mouse button on the arrow in the scene and then moving.

Example 6.3. The last example will show how to visualize a terrain when we have a heightmap of it. So, let us assume that we have a heightmap of a



Lut mode: Spectral
Number of colors: 256
Reverse lut: ☐

```
If this is your first time using Mayavi, take a quick look at the
tutorial examples section of the user guide, accessible via the help menu.
To use Mayavi, you need to load your data in "data sources" and apply
"visualization modules" to it.

In [1]:
```

Fig. 6.15. Visualization of the torus and the cross-cap surface.

terrain as in Fig. 6.16. First we must load the data into MayaVi so we choose menu *File* → *Load data* → *Open file...* and find the image with heightmap. A new node with the name of the file should appear. Before the visualization of the terrain we must use some filters to prepare the data. First we need to change the flat image into a mesh. For this purpose we will use the *Greedy Terrain Decimation* filter so we double click on the *Add module or filter* node then change tab to the *Processing filters* and then double click on the *Greedy Terrain Decimation.* After that we need to change one option in the properties of this filter. We click on the node with *Greedy Terrain Decimation* and in MayaVi object editor we turn off the *Boundary vertex deletion.* Next we reduce the number of triangles in the mesh so double click on the *Add module or filter* node, change the tab to the *Processing filters* and then double click on the *Quadric Decimation* filter. At this stage we can visualize the terrain with the *Surface* module but the terrain will be rendered only with white colour. To add more colours we need to use the *Elevation Filter* so we double click on the *Add module or filter* node, change the tab to the *Processing filters* and double click on the *Elevation Filter*. Then we need to change parameters of this filter. We click on the node representing the *Elevation Filter* and in the MayaVi object editor we change the value of the $F2$ parameter in the *High point* to 255. To make the terrain look smoother we will compute normals. To do this we double click on the *Add module or filter* node, change tab to the *Processing filters* and double click on the *Compute Normals* filter. Finally, we can visualize our terrain with the *Surface* module. The visualization of the heightmap from Fig. 6.16 is shown in Fig. 6.17.

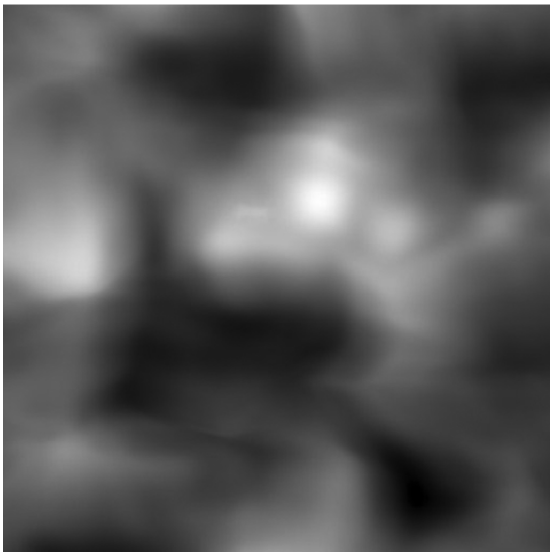


Fig. 6.16. Example heightmap.

6.2.7 Scripting with mlab

In some cases we have data which we want to visualize in a format that is not supported by MayaVi, e.g. other file format, data stored in a database, so we cannot use the MayaVi package to the visualization. The creator of MayaVi predicted such cases and he has written a Python package called mlab. With help of mlab and Python scripting language we can load the data we want and then visualize it using the functionality of MayaVi.

In this section we will show only some basic ideas of the scripting with mlab. Let us look at the listing 6.23 which presents an example of the script that has been written in Python and using the mlab package to visualize the

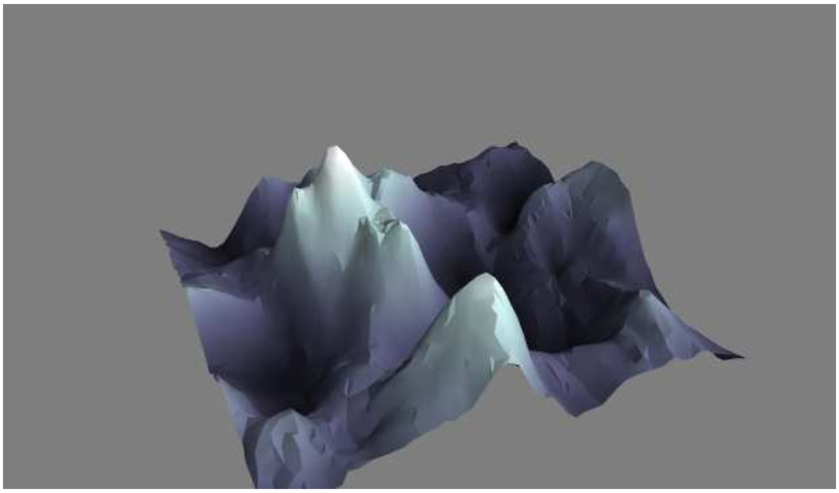


Fig. 6.17. Visualization of the heightmap from Fig. 6.16.

generated data. In the first line of the script we load a numpy package from which we will use the data structures to store the generated data. Next, in the second line we load the mlab package. The code in lines 4-16 is responsible for data generation. In this case it is data for visualization of a cone given by the parametric equations:

$$\begin{cases} x(\theta, v) = r(1 - v)\cos(\theta), \\ y(\theta, v) = r(1 - v)\sin(\theta), \\ z(\theta, v) = v \cdot h, \end{cases} \tag{6.3}$$

where $\theta \in [0, 2\pi]$, $v \in [0, 1]$, $r, h > 0$, r is the radius of the cone and h is the hight of it. In general case this part of code is responsible for loading or generating the data and storing this data in a numpy data structures.

Lines 17-21 are responsible for the visualization. This is very similar to the use of MayaVi. Simply, we execute appropriate functions which correspond to the filters and modules of MayaVi. In our example we use the mesh function (line 18) to visualize the cone, then we add to the scene the axes (axes function – line 19) and a title (title function – line 20) and finally we change the point of view (view function – line 21). The description of the used functions and the full mlab function reference can be found in the MayaVi2 User Guide [21]. To execute the script in MayaVi we write the script to disc, open MayaVi and then go to menu *File → Open Text File...* and search for the script. After this the content of the script should appear in the MayaVi window. Now we simply press Ctrl + r and after a few moments the visualization should appear in the scene. Fig. 6.18 presents the result of executing script from the listing 6.23.

Listing 6.23.

```
from numpy import *
from enthought.mayavi import mlab

# generating a data for visualization

# parameters of a cone
thetamax = 2*pi #
radius = 5 # radius of a cone
height = 5 # height of a cone

# generating a cone
[theta, v] = mgrid[0:thetamax+pi/50:pi/50, 0:1:0.05]
x = radius * (1-v) * cos(theta)
y = radius * (1-v) * sin(theta)
z = v * height

# visualization of the generated data
mlab.mesh(x, y, z, colormap='black-white')
mlab.axes()
mlab.title('Example', size=0.5)
mlab.view(45, 55, 30)
```

More examples and details about writing scripts in mlab can be found in the MayaVi2 User Guide [21].

6.2.8 VTK file format

As we mentioned earlier the VTK file format is a standard format used in MayaVi. In this section we will present some basic information about this file format.

There are two different styles of file formats in VTK:

1. legacy,
2. XML.

We will describe only the structure of the legacy style. The legacy style of the VTK file format consists of five parts:

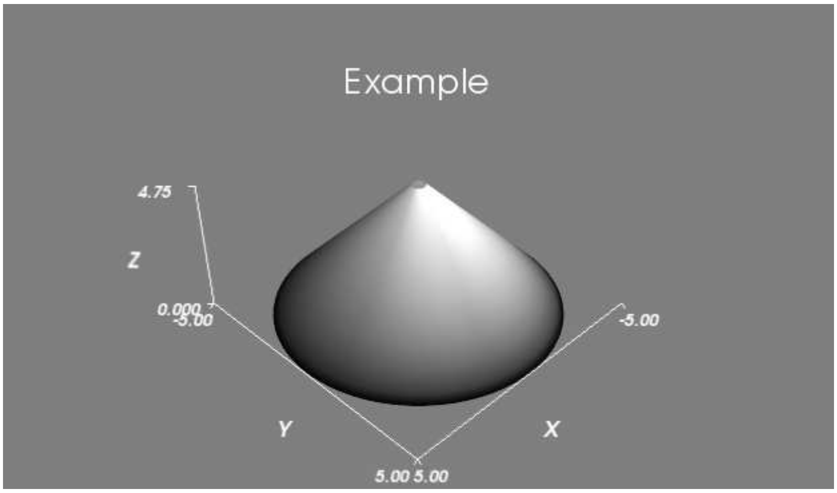


Fig. 6.18. Result of executing script from the listing 6.23.

1. File version and identifier. This part contains one single line
   #vtk DataFile Version x.x,
   where x.x is the number of version which varies with different releases of VTK.

2. Header. The header is used to describe the data and it is a character string (maximum 256 characters) terminated by the end-of-line character.

3. File format. This line describes the type of file. It can be either ASCII or BINARY.

4. Dataset structure. This is the geometry part describing the geometry and topology of the dataset and it begins with the line with keyword DATASET. There are five different dataset formats: structured points, structured grid, rectilinear grid, unstructured grid, polygonal data.

5. Dataset attributes. This part is used to give the data attribute values, i.e. scalars, vectors, tensors, normals, texture coordinates, field data. It begins with the keyword POINT_DATA or CELL_DATA followed by an integer number specifying the number of points or cells.

The first three parts are mandatory and the other two are optional. More details about the dataset structure and dataset attributes parts can be found in The VTK User's Guide [23] or on the VTK website [26].

Listing 6.24 presents an example of VTK file and Fig. 6.19 presents a visualization of the data included in this file.

Listing 6.24.

```
# vtk DataFile Version 2.0
Cube example
ASCII
DATASET POLYDATA
POINTS 8 float
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
0.0 1.0 1.0
POLYGONS 6 30
4 0 1 2 3
4 4 5 6 7
4 0 1 5 4
4 2 3 7 6
4 0 4 7 3
4 1 2 6 5
CELL_DATA 6
SCALARS cell_scalars int 1
LOOKUP_TABLE default
0
1
2
3
4
5
```

The XML style of the VTK file format is much more complicated than the legacy style but it supports many more features. Some features of the format include support for compression, portable binary encoding, random access, big endian and little endian byte order, multiple file representation of piece data and new file extensions for different VTK dataset types. The details about the XML style can be found in The VTK User's Guide [23] or on the VTK website [26].

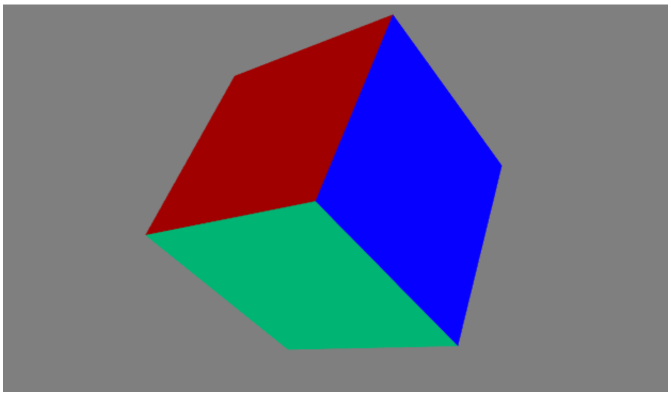


Fig. 6.19. Visualization of the data from a VTK file presented in listing 6.24.

6.3 Deep View / Swiss-PdbViewer

Deep View or Swiss-PdbViewer [22] is a multiplatform (Windows, Mac, Linux, Irix) application for interactive viewing and analyzing several protein and nucleic acid structures at the same time. It was created in 1994 by Nicolas Guex. The main features of Deep View are:

- superimposition – structural alignments and comparisons of their active sites or any other relevant,
- makes amino acid mutations,
- generates Hydrogen bonds,
- calculates angles and distances between atoms,
- tightly linked to Swiss-Model, an automated homology modelling server,
- threads a protein primary sequence onto a 3D template,
- builds missing loops and refines sidechain packing,
- reads electron density maps and builds into the density,
- performs energy minimization,
- POV-Ray scenes can be generated for stunning ray-traced quality images.

6.3.1 Basics of Deep View

When we start the Deep View application we will see a window which looks as in Fig. 6.20. Dependent on which operation will be executed there can appear other windows: Graphic window, Control Panel, Layers infos window, Alignment window, Ramachandran plot window, Surface and cavities window, Electron density map infos window, Text windows.

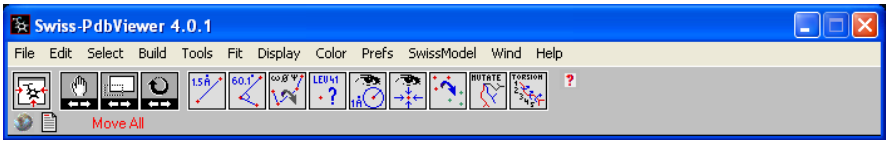

Swiss-PdbViewer 4.0.1
File Edit Select Build Tools Fit Display Color Prefs SwissModel Wind Help
Move All

Fig. 6.20. Main window of the Swiss-PdbViewer.

In Deep View we can open several types of files with molecule data – the default is PDB (Protein Data Bank) file. The specification of the PDB format can be found in [18]. We can open a PDB file or import it from a server. To import a file from the server we go to the menu *File* → *Import...* and then we see a window where we type a name of the molecule and in the *Grab from server* section we push the *PDB file* button. Let us try to import a protein called 1gdi. After importing a PDB file there should appear a Graphic window with a visualization of the molecule in wireframe representation. The default atoms colours are following:

- C → white,
- O → red,
- N → blue,
- S → yellow,
- P → orange,
- H → cyan,
- other → gray.

We can change the default colours of atoms in menu *Prefs* → *Colors...*. Besides, we can individually change the colour of any group using the *Control Panel*. To activate the *Control Panel* we go to *Wind* → *Control Panel*. In this panel we see all the groups from which the molecule consists (group column), also we can turn each of the groups on/off (show column), turn on/off the labels (labl column), surfaces (::v column), ribbons (ribn column) or change the colour of groups (col column). To change some settings for several groups at the same time we click on the first group and drag the mouse to the last group we want to change. The selected groups are indicated by the red colour and now we can change the settings using the +/- sign situated above the columns in *Control Panel*. Let us select all the groups and then turn off all the groups, turn on the ribbons. As we can see the default rendering of ribbons is a wireframe. To change the way in which they are rendered we go to the menu *Display* → *Render in* 3*D*.

Now, with the help of the mouse we can rotate, translate or scale the molecule. The rotation is done by dragging the mouse with the left mouse button, the translation is done in a similar way only we hold down the right mouse button and the scaling is done by dragging the mouse with the left and right mouse buttons.

If we are ready with modelling of the molecule, we can save the result as an image or as a POV-Ray file. To do this we go to menu *File* → *Save* → *Image...* for the image and to menu *File* → *Save* → *Pov-Ray Scene...* for the POV-Ray file. When we choose the POV-Ray file later we can change the look of the molecule using the POV-Ray language and then render it.

Because the aim of this book is only to show the basics of modelling and visualization, so we do not go deeper in the theme of modelling molecules. More details about this theme can be found in Deep View user guide [5] or on the Deep View website [22].

6.3.2 Examples

In this section we will present examples of images obtained with the help of Deep View and POV-Ray.

Example 6.4. Fig. 6.21 presents an example of 1gdi molecule and Fig. 6.22 presents the same molecule but all the groups are turned off and the ribbons are turned on.

Example 6.5. Figs. 6.23, 6.24 present an example of Hemoglobin modelled in Deep View and rendered by POV-Ray.

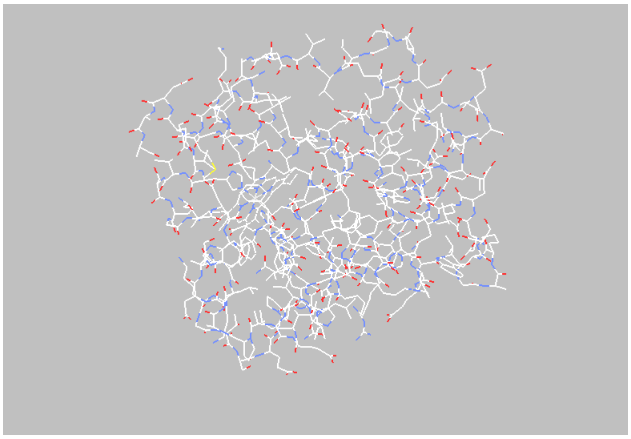


Fig. 6.21. Visualization of the 1gdi molecule.

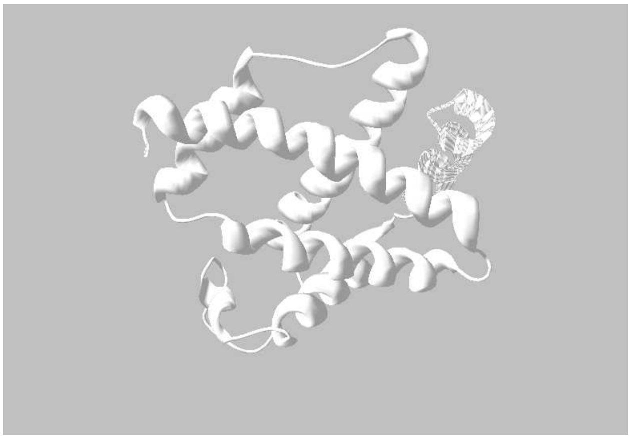


Fig. 6.22. Visualization of the 1gdi molecule ribbons.

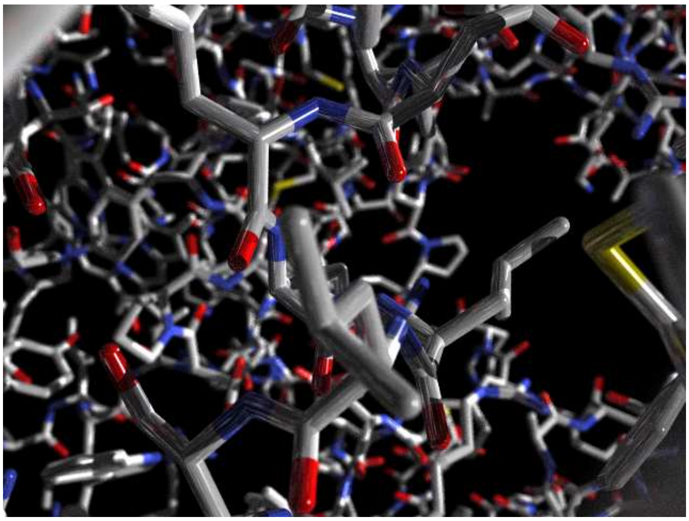


Fig. 6.23. Visualization of Hemoglobin rendered by POV-Ray.

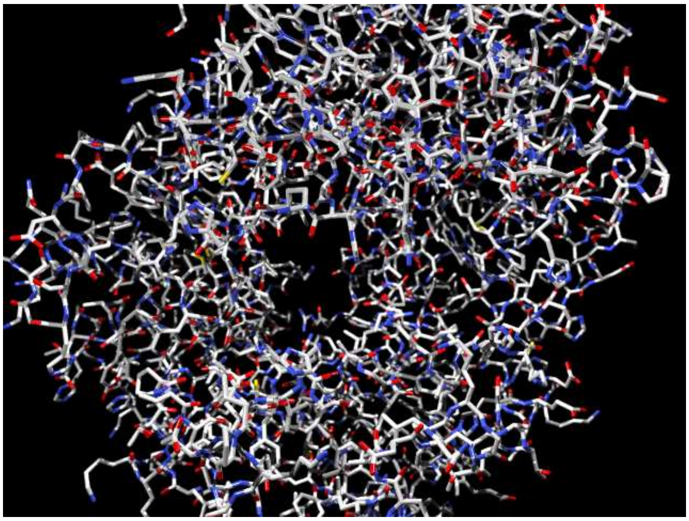


Fig. 6.24. Visualization of Hemoglobin rendered by POV-Ray.

Bibliography

[1] Barnsley M., *Fractals Everywhere*, Academic Press, New York, 1988.

[2] Barnsley M., *Superfractals*, Cambridge University Press, New York, Melbourne, 2006.

[3] Chaikin G., *An Algorithm for High Speed Curve Generation*, Computer Graphics and Image Processing, Vol. 3, 346-349, 1974.

[4] Cohen M.F., Wallace J.R., *Radiosity and Realistic Image Synthesis*, Academic Press, 1995.

[5] *Deep View – The Swiss-Pdb Viewer User Guide*, http://spdbv.vital-it.ch/Swiss-PdbViewerManualv3.7.pdf, Accessed September 2009.

[6] Enthought Python Distribution, http://www.enthought.com/products/epd.php, Accessed September 2009.

[7] Falconer K., *Fractal Geometry. Mathematical Foundations and Applications*, Second Edition, Wiley, 2003.

[8] Farin G., *Curves and Surfaces for CAD, A Practical Guide*, Academic Press 2002.

[9] Glassner A.S., *An Introduction to Ray Tracing*, Academic Press, 1991.

[10] Goldman R., *The Fractal Nature of Bézier Curves*, Proceedings of the Geometric Modeling and Processing, April 13-15, 2004, Beijing, China, 3-11.

[11] Jensen H.W., *Realistic Image Synthesis using Photon Mapping*, AK Peters, 2001.

[12] Joy K. I., *On-line Geometric Modeling Notes*, Dept. of Computer Science, University of California, Davis, 2000, http://graphics.cs.ucdavis.edu/education/CAGDNotes/homepage.html, Accessed September 2009.

[13] Mandelbrot B., *The Fractal Geometry of Nature*, Freeman and Company, San Francisco, 1983.

[14] Marsh D., *Applied Geometry for Computer Graphics and CAD*, Springer, 2000.

[15] POV-Ray, http://www.povray.org/, Accessed September 2009.

[16] Prusinkiewicz P., Lindenmayer A., *The Algorithmic Beauty of Plants*, Springer, New York, 1990.

[17] Python(x,y), http://www.pythonxy.com/, Accessed September 2009.

[18] *Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description*, http://www.wwpdb.org/documentation/format32/v3.2.html, Accessed September 2009.

[19] Ramachandran P., *MayaVi-2: The next generation*, EuroPython Conference Proceedings, Goteborg, Sweden June 2005.

[20] Ramachandran P., Varoquaux G., *Mayavi: Making 3D Data Visualization Reusable*, SciPy08: Proceedings of the 7th Python in Science Conference, Caltech, Pasadena, CA, 19-24 August, 2008.

[21] Ramachandran P., Varoquaux G., *Mayavi2 Users Guide*, http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/, Accessed September 2009.

[22] Swiss-PdbViewer (DeepView), http://spdbv.vital-it.ch/, Accessed September 2009.

[23] Schroeder W.J., *The VTK User's Guide*, Kitware Inc., 2001.

[24] Schroeder W.J., Martin K., Lorensen B., *The Visualization Toolkit. An Object-Oriented Approach to 3D Graphics*, 3rd Edition, Kitware Inc., 2002.

[25] Shirley P., Morley R.K., *Realistic Ray Tracing*, 2nd Edition, AK Peters, 2003.

[26] Visualization Toolkit, http://www.vtk.org/, Accessed September 2009.

[27] Warren J., Weimer H., *Subdivision Methods for Geometric Design: A Constructive Approach*, Morgan Kaufmann, San Francisco, 2001.

[28] Wright H., *Intorduction to Scientific Visualization*, Springer, 2007.

[29] Zorin D., Schröder P., Levin A., Kobbelt L., Sweldens W., DeRose T., *Subdivision for Modeling and Animation*, Course Notes, SIGGRAPH 2000.

List of Figures

About authors

Wiesław Kotarski is a professor at the Institute of Computer Science, University of Silesia in Sosnowiec. His main scientific interests are related to computer graphics (subdivision methods, fractals) and applied mathematics (optimal control theory). He is the author of several papers and the monograph entitled ,,Fractal modelling of shapes", published in 2008. He is a member of international societies: IEEE Computer Society, American and European Mathematical Societies (AMS, EMS).

Krzysztof Gdawiec is an assistant at the Institute of Computer Science, University of Silesia in Sosnowiec. Recently, he prepared his PhD thesis on local fractal methods of 2D shapes recognition. He is interested in computer graphics, fractal modelling and automatic 2D shape recognition.

Grzegorz T. Machnik is a PhD student at the Institute of Computer Science, University of Silesia in Sosnowiec. He is interested in computer graphics and application of genetic algorithms to natural evolution processes.

Description:

This textbook presents basic concepts related to modelling and visualization tasks. Chapters 1-4 describe transformations in the plane and in the space, and geometrical forms of graphical objects such as curves, patches and fractals. Chapter 5 is about lights, materials, textures, colours that all are needed to enrich a severe appearance of pure geometrical objects leading to their photorealistic visualizations. In Chapter 6 freeware software such as POV Ray, MayaVi and Deep View are described. Using those software one can obtain photorealistic renderings and visualize data.

The textbook was prepared for students of the specialization "Modelling and Visualization in Bioinformatics" but it should be helpful to anyone who is interested in computer graphics, modelling techniques, animation and visualization of data. Authors of this textbook believe that information presented in the book will be useful for students and will inspire their imagination in creation of photorealistic static 3D scenes and also will be helpful in creation of animations and visualization of data in an effective and professional way.

eea grants
iceland liechtenstein norway

norway grants

FSS
Fundusz Stypendialny i Szkoleniowy

frse
Fundacja Rozwoju Systemu Edukacji

This project was completed with the support granted by Iceland, Liechtenstein and Norway by means of co-financing from the European Economic Area Financial Mechanism and the Norwegian Financial Mechanism as part of the Scholarship and Training Fund