22. Computer Graphics

Computer Graphics algorithms create and render ***virtual worlds*** stored in the computer memory. The virtual world model may contain ***shapes*** (points, line segments, surfaces, solid objects etc.), which are represented by digital numbers. ***Rendering*** computes the displayed ***image*** of the virtual world from a given virtual camera. The image consists of small rectangles, called ***pixels***. A pixel has a unique colour, thus it is sufficient to solve the rendering problem for a single point in each pixel. This point is usually the centre of the pixel. Rendering finds that shape which is visible through this point and writes its visible colour into the pixel. In this chapter we discuss the creation of virtual worlds and the determination of the visible shapes.

22.1. Fundamentals of analytic geometry

The base set of our examination is the Euclidean ***space*** In computer algorithms the elements of this space should be described by numbers. The branch of geometry describing the elements of space by numbers is the ***analytic geometry***. The basic concepts of analytic geometry are the vector and the coordinate system.

Definition 22.1 *A **vector** is a **translation** that is defined by its direction and length. A vector is denoted by $\vec{v}$.*

The length of the vector is also called its ***absolute value***, and is denoted by $|\vec{v}|$. Vectors can be added, resulting in a new vector that corresponds to subsequent translations. Addition is denoted by $\vec{v}_1 + \vec{v}_2 = \vec{v}$. Vectors can be multiplied by scalar values, resulting also in a vector ($\lambda \cdot \vec{v}_1 = \vec{v}$), which translates at the same direction as $\vec{v}_1$, but the length of translation is scaled by λ.

The ***dot product*** of two vectors is a *scalar* that is equal to the product of the lengths of the two vectors and the cosine of their angle:

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \cos\alpha, \qquad \text{where } \alpha \text{ is the angle between } \vec{v}_1 \text{ and } \vec{v}_2 \ .$$

Two vectors are said to be ***orthogonal*** if their dot product is zero.

On the other hand, the ***cross product*** of two vectors is a *vector* that is orthogonal to the plane of the two vectors and its length is equal to the product of the

lengths of the two vectors and the sine of their angle:

$$\vec{v}_1 \times \vec{v}_2 = \vec{v}, \qquad \text{where } \vec{v} \text{ is orthogonal to } \vec{v}_1 \text{ and } \vec{v}_2, \text{ and } |\vec{v}| = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \sin\alpha\ .$$

There are two possible orthogonal vectors, from which that alternative is selected where our middle finger of the right hand would point if our thumb were pointing to the first and our forefinger to the second vector (***right hand rule***). Two vectors are said to be ***parallel*** if their cross product is zero.

22.1.1. Cartesian coordinate system

Any vector $\vec{v}$ of a plane can be expressed as the linear combination of two, non-parallel vectors $\vec{i}$, $\vec{j}$ in this plane, that is

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j}\ .$$

Similarly, any vector $\vec{v}$ in the three-dimensional space can be unambiguously defined by the linear combination of three, not coplanar vectors:

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j} + z \cdot \vec{k}\ .$$

Vectors $\vec{i}$, $\vec{j}$, $\vec{k}$ are called ***basis vectors***, while scalars x, y, z are referred to as ***coordinates***. We shall assume that the basis vectors have unit length and they are orthogonal to each other. Having defined the basis vectors any other vector can unambiguously be expressed by three scalars, i.e. by its coordinates.

A ***point*** is specified by that vector which translates the reference point, called ***origin***, to the given point. In this case the translating vector is the ***place vector*** of the given point.

The origin and the basis vectors constitute the ***Cartesian coordinate system***, which is the basic tool to describe the points of the Euclidean plane or space by numbers.

The Cartesian coordinate system is the algebraic basis of the Euclidean geometry, which means that scalar triplets of Cartesian coordinates can be paired with the points of the space, and having made a correspondence between algebraic and geometric concepts, the theorems of the Euclidean geometry can be proven by algebraic means.

Exercises

22.1-1 Prove that there is a one-to-one mapping between Cartesian coordinate triplets and points of the three-dimensional space.

22.1-2 Prove that if the basis vectors have unit length and are orthogonal to each other, then $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = x_1x_2 + y_1y_2 + z_1z_2$.

22.2. Description of point sets with equations

Coordinate systems provide means to specify points by numbers. Conditions on these numbers, on the other hand, may define sets of points. Conditions are formulated by

solid	$f(x,y,z)$ implicit function
sphere of radius R	$R^2 - x^2 - y^2 - z^2$
block of size $2a, 2b, 2c$	$\min\{a - \lvert x\rvert, b - \lvert y\rvert, c - \lvert z\rvert\}$
torus of axis z, radii r (tube) and R (hole)	$r^2 - z^2 - (R - \sqrt{x^2 + y^2})^2$

Figure 22.1 Functions defining the sphere, the block, and the torus.

equations. The coordinates found as the solution of these equations define the point set.

Let us now consider how these equations can be established.

22.2.1. Solids

A ***solid*** is a subset of the three-dimensional Euclidean space. To define this subset, continuous function f is used which maps the coordinates of points onto the set of real numbers. We say that a point belongs to the solid if the coordinates of the point satisfy the following implicit inequality:

$$f(x,y,z) \geq 0 \ .$$

Points satisfying inequality $f(x,y,z) > 0$ are the ***internal points,*** while points defined by $f(x,y,z) < 0$ are the ***external points.*** Because of the continuity of function f, points satisfying equality $f(x,y,z) = 0$ are between external and internal points and are called the ***boundary surface*** of the solid. Intuitively, function f describes the signed distance between a point and the boundary surface.

We note that we usually do not consider any point set as a solid, but also require that the point set does not have lower dimensional degeneration (e.g. hanging lines or surfaces), i.e. that arbitrarily small neighborhoods of each point of the boundary surface contain internal points.

Figure 22.1 lists the defining functions of the sphere, the box, and the torus.

22.2.2. Surfaces

Points having coordinates that satisfy equation $f(x,y,z) = 0$ are on the boundary ***surface.*** Surfaces can thus be defined by this ***implicit equation.*** Since points can also be given by the place vectors, the implicit equation can be formulated for the place vectors as well:

$$f(\vec{r}) = 0 \ .$$

A surface may have many different equations. For example, equations $f(x,y,z) = 0$, $f^2(x,y,z) = 0$, and $2 \cdot f^3(x,y,z) = 0$ are algebraically different, but they have the same roots and thus define the same set of points.

A plane of normal vector $\vec{n}$ and place vector $\vec{r}_0$ contains those points for which vector $\vec{r} - \vec{r}_0$ is perpendicular to the normal, thus their dot product is zero. Based on this, the points of a plane are defined by the following vector or scalar equations:

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} = 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \ , \tag{22.1}$$

solid	$x(u,v)$	$y(u,v)$	$z(u,v)$
sphere of radius R	$R \cdot \cos 2\pi u \cdot \sin \pi v$	$R \cdot \sin 2\pi u \cdot \sin \pi v$	$R \cdot \cos \pi v$
cylinder of radius R, axis z, and of height h	$R \cdot \cos 2\pi u$	$R \cdot \sin 2\pi u$	$h \cdot v$
cone of radius R, axis z, and of height h	$R \cdot (1-v) \cdot \cos 2\pi u$	$R \cdot (1-v) \cdot \sin 2\pi u$	$h \cdot v$

Figure 22.2 Parametric forms of the sphere, the cylinder, and the cone, where $u, v \in [0,1]$.

where n_x, n_y, n_z are the coordinates of the normal and $d = -\vec{r}_0 \cdot \vec{n}$. If the normal vector has unit length, then d expresses the signed distance between the plane and the origin of the coordinate system. Two planes are said to be ***parallel*** if their normals are parallel.

In addition to using implicit equations, surfaces can also be defined by ***parametric forms.*** In this case, the Cartesian coordinates of surface points are functions of two independent variables. Denoting these free parameters by u and v, the parametric equations of the surface are:

$$x = x(u,v), \qquad y = y(u,v), \qquad z = z(u,v), \qquad u \in [u_{\min}, u_{\max}], \quad v \in [v_{\min}, v_{\max}] \ .$$

The implicit equation of a surface can be obtained from the parametric equations by eliminating free parameters u, v. Figure 22.2 includes the parametric forms of the sphere, the cylinder and the cone.

Parametric forms can also be defined directly for the place vectors:

$$\vec{r} = \vec{r}(u,v) \ .$$

Points of a ***triangle*** are the ***convex combinations*** of points $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$, that is

$$\vec{r}(\alpha,\beta,\gamma) = \alpha \cdot \vec{p}_1 + \beta \cdot \vec{p}_2 + \gamma \cdot \vec{p}_3, \qquad \text{where} \quad \alpha,\beta,\gamma \geq 0 \quad \text{and} \quad \alpha+\beta+\gamma = 1 \ .$$

From this definition we can obtain the usual two-variate parametric form of a triangle substituting α by u, β by v, and γ by $(1-u-v)$:

$$\vec{r}(u,v) = u \cdot \vec{p}_1 + v \cdot \vec{p}_2 + (1-u-v) \cdot \vec{p}_3, \qquad \text{where} \quad u,v \geq 0 \quad \text{and} \quad u+v \leq 1 \ .$$

22.2.3. Curves

By intersecting two surfaces, we obtain a ***curve*** that may be defined formally by the implicit equations of the two intersecting surfaces

$$f_1(x,y,z) = f_2(x,y,z) = 0 \ ,$$

but this is needlessly complicated. Instead, let us consider the parametric forms of the two surfaces, given as $\vec{r}_1(u_1,v_1)$ and $\vec{r}_2(u_2,v_2)$, respectively. The points of the intersection satisfy vector equation $\vec{r}_1(u_1,v_1) = \vec{r}_2(u_2,v_2)$, which corresponds to three scalar equations, one for each coordinate of the three-dimensional space. Thus we can eliminate three from the four unknowns (u_1,v_1,u_2,v_2), and obtain a one-variate parametric equation for the coordinates of the curve points:

test	$x(u,v)$	$y(u,v)$	$z(u,v)$
ellipse of main axes $2a$, $2b$ on plane $z=0$	$a \cdot \cos 2\pi t$	$b \cdot \sin 2\pi t$	0
helix of radius R, axis z, and elevation h	$R \cdot \cos 2\pi t$	$R \cdot \sin 2\pi t$	$h \cdot t$
line segment between points (x_1,y_1,z_1) and (x_2,y_2,z_2)	$x_1(1-t)+x_2 t$	$y_1(1-t)+y_2 t$	$z_1(1-t)+z_2 t$

Figure 22.3 Parametric forms of the ellipse, the helix, and the line segment, where $t \in [0,1]$.

$$x = x(t), \qquad y = y(t), \qquad z = z(t), \qquad t \in [t_{\min}, t_{\max}] \ .$$

Similarly, we can use the vector form:

$$\vec{r} = \vec{r}(t), \qquad t \in [t_{\min}, t_{\max}] \ .$$

Figure 22.3 includes the parametric equations of the ellipse, the helix, and the line segment.

Note that we can define curves on a surface by fixing one of free parameters u, v. For example, by fixing v the parametric form of the resulting curve is $\vec{r}_v(u) = \vec{r}(u,v)$. These curves are called ***iso-parametric curves.***

Two points define a ***line.*** Let us select one point and call the place vector of this point the ***place vector of the line.*** On the other hand, the vector between the two points is the ***direction vector.*** Any other point of the line can be obtained by a translation of the point of the place vector parallel to the direction vector. Denoting the place vector by $\vec{r}_0$ and the direction vector by $\vec{v}$, the equation of the ***line*** is:

$$\vec{r}(t) = r_0 + \vec{v} \cdot t, \qquad t \in (-\infty, \infty) \ . \tag{22.2}$$

Two lines are said to be ***parallel*** if their direction vectors are parallel.

Instead of the complete line, we can also specify the points of a line segment if parameter t is restricted to an interval. For example, the ***equation of the line segment*** between points $\vec{r}_1$ and $\vec{r}_2$ is:

$$\vec{r}(t) = \vec{r}_1 + (\vec{r}_2 - \vec{r}_1) \cdot t = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t, \qquad t \in [0,1] \ . \tag{22.3}$$

According to this definition, the points of a line segment are the ***convex combinations*** of the endpoints.

22.2.4. Normal vectors

In computer graphics we often need the normal vectors of the surfaces (i.e. the normal vector of the tangent plane of the surface). Let us take an example. A mirror reflects light in a way that the incident direction, the normal vector, and the reflection direction are in the same plane, and the angle between the normal and the incident direction equals to the angle between the normal and the reflection direction. To carry out such and similar computations, we need methods to obtain the normal of the surface.

The equation of the tangent plane is obtained as the first order Taylor approximation of the implicit equation around point (x_0, y_0, z_0):

$$f(x,y,z) = f(x_0 + (x-x_0), y_0 + (y-y_0), z_0 + (z-z_0)) \approx$$

$$f(x_0, y_0, z_0) + \frac{\partial f}{\partial x} \cdot (x - x_0) + \frac{\partial f}{\partial y} \cdot (y - y_0) + \frac{\partial f}{\partial z} \cdot (z - z_0) \ .$$

Points (x_0, y_0, z_0) and (x, y, z) are on the surface, thus $f(x_0, y_0, z_0) = 0$ and $f(x, y, z) = 0$, resulting in the following equation of the ***tangent plane:***

$$\frac{\partial f}{\partial x} \cdot (x - x_0) + \frac{\partial f}{\partial y} \cdot (y - y_0) + \frac{\partial f}{\partial z} \cdot (z - z_0) = 0 \ .$$

Comparing this equation to equation (22.1), we can realize that the normal vector of the tangent plane is

$$\vec{n} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) = \mathrm{grad} f \ . \tag{22.4}$$

The normal vector of parametric surfaces can be obtained by examining the iso-parametric curves. The tangent of curve $\vec{r}_v(u)$ defined by fixing parameter v is obtained by the first-order Taylor approximation:

$$\vec{r}_v(u) = \vec{r}_v(u_0 + (u - u_0)) \approx \vec{r}_v(u_0) + \frac{d\vec{r}_v}{du} \cdot (u - u_0) = \vec{r}_v(u_0) + \frac{\partial \vec{r}}{\partial u} \cdot (u - u_0) \ .$$

Comparing this approximation to equation (22.2) describing a line, we conclude that the direction vector of the tangent line is $\partial\vec{r}/\partial u$. The tangent lines of the curves running on a surface are in the tangent plane of the surface, making the normal vector perpendicular to the direction vectors of these lines. In order to find the normal vector, both the tangent line of curve $\vec{r}_v(u)$ and the tangent line of curve $\vec{r}_u(v)$ are computed, and their cross product is evaluated since the result of the cross product is perpendicular to the multiplied vectors. The normal of surface $\vec{r}(u, v)$ is then

$$\vec{n} = \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v} \ . \tag{22.5}$$

22.2.5. Curve modelling

Parametric and implicit equations trace back the geometric design of the virtual world to the establishment of these equations. However, these equations are often not intuitive enough, thus they cannot be used directly during design. It would not be reasonable to expect the designer working on a human face or on a car to directly specify the equations of these objects. Clearly, indirect methods are needed which require intuitive data from the designer and define these equations automatically. One category of these indirect approaches apply ***control points.*** Another category of methods work with elementary building blocks (box, sphere, cone, etc.) and with set operations.

Let us discuss first how the method based on control points can define curves. Suppose that the designer specified points $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_m$, and that parametric curve of equation $\vec{r} = \vec{r}(t)$ should be found which "follows" these points. For the time being, the curve is not required to go through these control points.

We use the analogy of the centre of mass of mechanical systems to construct our

curve. Assume that we have sand of unit mass, which is distributed at the control points. If a control point has most of the sand, then the centre of mass is close to this point. Controlling the distribution of the sand as a function of parameter t to give the main influence to different control points one after the other, the centre of mass will travel through a curve running close to the control points.

Let us put weights $B_0(t), B_1(t), \ldots, B_m(t)$ at control points at parameter t. These weighting functions are also called the ***basis functions*** of the curve. Since unit weight is distributed, we require that for each t the following identity holds:

$$\sum_{i=0}^{m} B_i(t) = 1 \ .$$

For some t, the respective point of the curve curve is the centre of mass of this mechanical system:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i}{\sum_{i=0}^{m} B_i(t)} = \sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i \ .$$

Note that the reason of distributing sand of unit mass is that this decision makes the denominator of the fraction equal to 1. To make the analogy complete, the basis functions cannot be negative since the mass is always non-negative. The centre of mass of a point system is always in the ***convex hull***[1] of the participating points, thus if the basis functions are non-negative, then the curve remains in the convex hull of the control points.

The properties of the curves are determined by the basis functions. Let us now discuss two popular basis function systems, namely the basis functions of the Bézier-curves and the B-spline curves.

Bézier-curve. Pierre Bézier, a designer working at Renault, proposed the ***Bernstein polynomials*** as basis functions. Bernstein polynomials can be obtained as the expansion of $1^m = (t + (1-t))^m$ according to binomial theorem:

$$(t + (1-t))^m = \sum_{i=0}^{m} \binom{m}{i} \cdot t^i \cdot (1-t)^{m-i} \ .$$

The basis functions of ***Bézier curves*** are the terms of this sum ($i = 0, 1, \ldots, m$):

$$B_{i,m}^{\text{Bezier}}(t) = \binom{m}{i} \cdot t^i \cdot (1-t)^{m-i} \ . \tag{22.6}$$

According to the introduction of Bernstein polynomials, it is obvious that they really meet condition $\sum_{i=0}^{m} B_i(t) = 1$ and $B_i(t) \geq 0$ in $t \in [0, 1]$, which guarantees that Bézier curves are always in the convex hulls of their control points. The basis functions and the shape of the Bézier curve are shown in Figure 22.4. At parameter value $t = 0$ the first basis function is 1, while the others are zero, therefore the curve

[1] The convex hull of a point system is by definition the minimal convex set containing the point system.

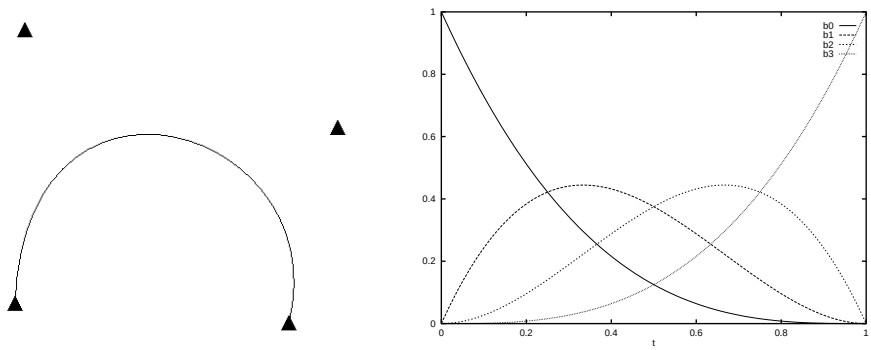


Figure 22.4 A Bézier curve defined by four control points and the respective basis functions ($m = 3$).

starts at the first control point. Similarly, at parameter value $t = 1$ the curve arrives at the last control point. At other parameter values, all basis functions are positive, thus they simultaneously affect the curve. Consequently, the curve usually does not go through the other control points.

B-spline. The basis functions of the ***B-spline*** can be constructed applying a sequence of linear blending. A B-spline weights the $m + 1$ number of control points by $(k - 1)$-degree polynomials. Value k is called the ***order*** of the curve. Let us take a non-decreasing series of $m + k + 1$ parameter values, called the ***knot vector***:

$$\mathbf{t} = [t_0, t_1, \ldots, t_{m+k}], \quad t_0 \leq t_1 \leq \cdots \leq t_{m+k} \ .$$

By definition, the ith first order basis function is 1 in the ith interval, and zero elsewhere (Figure 22.5):

$$B_{i,1}^{\mathrm{BS}}(t) = \begin{cases} 1, & \text{if } t_i \leq t < t_{i+1} \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Using this definition, $m + k$ number of first order basis functions are established, which are non-negative zero-degree polynomials that sum up to 1 for all $t \in [t_0, t_{m+k})$ parameters. These basis functions have too low degree since the centre of mass is not even a curve, but jumps from control point to control point.

The order of basis functions, as well as the smoothness of the curve, can be increased by blending two consecutive basis functions with linear weighting (Figure 22.5). The first basis function is weighted by linearly increasing factor $(t - t_i)/(t_{i+1} - t_i)$ in domain $t_i \leq t < t_{i+1}$, where the basis function is non-zero. The next basis function, on the other hand, is scaled by linearly decreasing factor $(t_{i+2} - t)/(t_{i+2} - t_{i+1})$ in its domain $t_{i+1} \leq t < t_{i+2}$ where it is non zero. The two weighted basis functions are added to obtain the tent-like second order basis functions. Note that

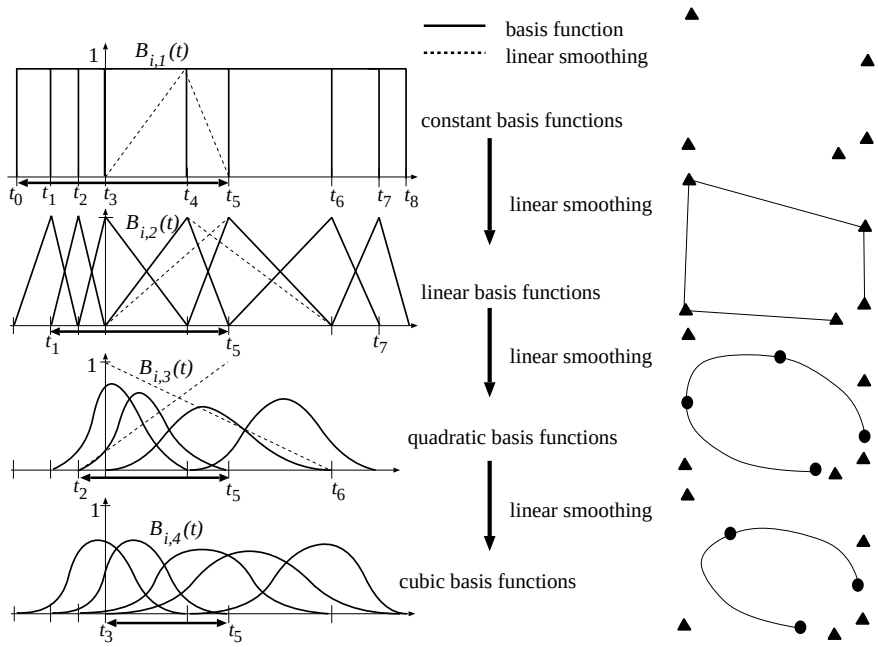


Figure 22.5 Construction of B-spline basis functions. A higher order basis function is obtained by blending two consecutive basis functions on the previous level using a linearly increasing and a linearly decreasing weighting, respectively. Here the number of control points is 5, i.e. $m = 4$. Arrows indicate useful interval $[t_{k-1}, t_{m+1}]$ where we can find $m + 1$ number of basis functions that add up to 1. The right side of the figure depicts control points with triangles and curve points corresponding to the knot values by circles.

while a first order basis function is non-zero in a single interval, the second order basis functions expand to two intervals. Since the construction makes a new basis function from every pair of consecutive lower order basis functions, the number of new basis functions is one less than that of the original ones. We have just $m+k-1$ second order basis functions. Except for the first and the last first order basis functions, all of them are used once with linearly increasing and once with linearly decreasing weighting, thus with the exception of the first and the last intervals, i.e. in $[t_1, t_{m+k-1}]$, the new basis functions also sum up to 1.

The second order basis functions are first degree polynomials. The degree of basis functions, i.e. the order of the curve, can be arbitrarily increased by the recursive application of the presented blending method. The dependence of the next order basis functions on the previous order ones is as follows:

$$B^{\mathrm{BS}}_{i,k}(t) = \frac{(t - t_i)B^{\mathrm{BS}}_{i,k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)B^{\mathrm{BS}}_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}, \qquad \text{if } k > 1 \ .$$

Note that we always take two consecutive basis functions and weight them in their non-zero domain (i.e. in the interval where they are non-zero) with linearly increasing factor $(t-t_i)/(t_{i+k-1}-t_i)$ and with linearly decreasing factor $(t_{i+k}-t)/(t_{i+k}-t_{i+1})$,

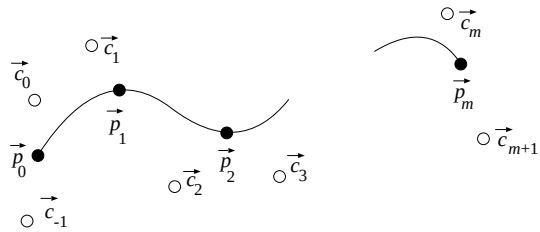


Figure 22.6 A B-spline interpolation. Based on points $\vec{p}_0, \ldots, \vec{p}_m$ to be interpolated, control points $\vec{c}_{-1}, \ldots, \vec{c}_{m+1}$ are computed to make the start and end points of the segments equal to the interpolated points.

respectively. The two weighted functions are summed to obtain the higher order, and therefore smoother basis function. Repeating this operation $(k-1)$ times, k-order basis functions are generated, which sum up to 1 in interval $[t_{k-1}, t_{m+1}]$. The knot vector may have elements that are the same, thus the length of the intervals may be zero. Such intervals result in 0/0 like fractions, which must be replaced by value 1 in the implementation of the construction.

The value of the ith k-order basis function at parameter t can be computed with the following ***Cox-deBoor-Mansfield recursion:***

B-SPLINE$(i, k, t, \mathbf{t})$

```
1  if k = 1                                                    ▷ Trivial case.
2     then if t_i ≤ t < t_{i+1}
3             then return 1
4             else return 0
5  if t_{i+k-1} − t_i > 0
6     then b_1 ← (t − t_i)/(t_{i+k-1} − t_i)   ▷ Previous with linearly increasing weight.
7     else  b_1 ← 1                                             ▷ Here: 0/0 = 1.
8  if t_{i+k} − t_{i+1} > 0
9     then b_2 ← (t_{i+k} − t)/(t_{i+k} − t_{i+1})   ▷ Next with linearly decreasing weight.
10    else  b_2 ← 1                                             ▷ Here: 0/0 = 1.
11 B ← b_1 · B-SPLINE(i, k − 1, t, t) + b_2 · B-SPLINE(i + 1, k − 1, t, t)   ▷ Recursion.
12 return B
```

In practice, we usually use fourth-order basis functions ($k = 4$), which are third-degree polynomials, and define curves that can be continuously differentiated twice. The reason is that bent rods and motion paths following the Newton laws also have this property.

While the number of control points is greater than the order of the curve, the basis functions are non-zero only in a part of the valid parameter set. This means that a control point affects just a part of the curve. Moving this control point, the change of the curve is ***local.*** Local control is a very important property since the designer can adjust the shape of the curve without destroying its general form.

A fourth-order B-spline usually does not go through its control points. If we wish to use it for interpolation, the control points should be calculated from the points to

be interpolated. Suppose that we need a curve which visits points $\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_m$ at parameter values $t_0 = 0, t_1 = 1, \ldots, t_m = m$, respectively (Figure 22.6). To find such a curve, control points $[\vec{c}_{-1}, \vec{c}_0, \vec{c}_1, \ldots, \vec{c}_{m+1}]$ should be found to meet the following interpolation criteria:

$$\vec{r}(t_j) = \sum_{i=-1}^{m+1} \vec{c}_i \cdot B_{i,4}^{\mathrm{BS}}(t_j) = \vec{p}_j, \qquad j = 0, 1, \ldots, m \ .$$

These criteria can be formalized as $m + 1$ linear equations with $m + 3$ unknowns, thus the solution is ambiguous. To make the solution unambiguous, two additional conditions should be imposed. For example, we can set the derivatives (for motion paths, the speed) at the start and end points.

B-spline curves can be further generalized by defining the influence of the ith control point as the product of B-spline basis function $B_i(t)$ and additional weight w_i of the control point. The curve obtained this way is called the Non-Uniform Rational B-Spline, abbreviated as ***NURBS,*** which is very popular in commercial geometric modelling systems.

Using the mechanical analogy again, the mass put at the ith control point is $w_i B_i(t)$, thus the centre of mass is:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} w_i B_i^{\mathrm{BS}}(t) \cdot \vec{r}_i}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} = \sum_{i=0}^{m} B_i^{\mathrm{NURBS}}(t) \cdot \vec{r}_i \ .$$

The correspondence between B-spline and NURBS basis functions is as follows:

$$B_i^{\mathrm{NURBS}}(t) = \frac{w_i B_i^{\mathrm{BS}}(t)}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} \ .$$

Since B-spline basis functions are piece-wise polynomial functions, NURBS basis functions are piece-wise rational functions. NURBS can describe quadratic curves (e.g. circle, ellipse, etc.) without any approximation error.

22.2.6. Surface modelling

Parametric surfaces are defined by two variate functions $\vec{r}(u, v)$. Instead of specifying this function directly, we can take finite number of control points $\vec{r}_{ij}$ which are weighted with the basis functions to obtain the parametric function:

$$\vec{r}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \vec{r}_{ij} \cdot B_{ij}(u, v) \ . \tag{22.7}$$

Similarly to curves, basis functions are expected to sum up to 1, i.e. $\sum_{i=0}^{n} \sum_{j=0}^{m} B_{ij}(u, v) = 1$ everywhere. If this requirement is met, we can imagine that the control points have masses $B_{ij}(u, v)$ depending on parameters u, v, and the centre of mass is the surface point corresponding to parameter pair u, v.

Basis functions $B_{ij}(u, v)$ are similar to those of curves. Let us fix parameter v. Changing parameter u, curve $\vec{r}_v(u)$ is obtained on the surface. This curve can be

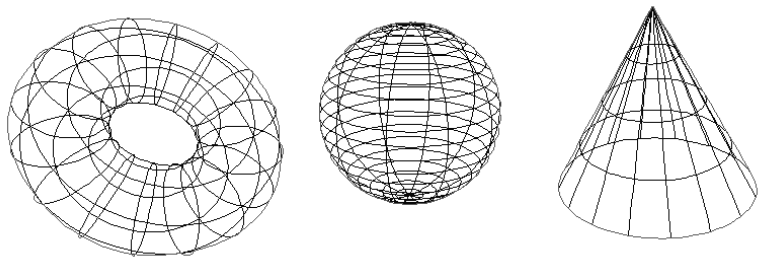


Figure 22.7 Iso-parametric curves of surface.

defined by the discussed curve definition methods:

$$\vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \cdot \vec{r}_i \ , \tag{22.8}$$

where $B_i(u)$ is the basis function of the selected curve type.

Of course, fixing v differently we obtain another curve of the surface. Since a curve of a given type is unambiguously defined by the control points, control points $\vec{r}_i$ must depend on the fixed v value. As parameter v changes, control point $\vec{r}_i = \vec{r}_i(v)$ also runs on a curve, which can be defined by control points $\vec{r}_{i,0}, \vec{r}_{i,2}, \ldots, \vec{r}_{i,m}$:

$$\vec{r}_i(v) = \sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \ .$$

Substituting this into equation (22.8), the parametric equation of the surface is:

$$\vec{r}(u,v) = \vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \left(\sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \right) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) \cdot \vec{r}_{ij} \ .$$

Unlike curves, the control points of a surface form a two-dimensional grid. The two-dimensional basis functions are obtained as the product of one-variate basis functions parameterized by u and v, respectively.

22.2.7. Solid modelling with blobs

Free form solids – similarly to parametric curves and surfaces – can also be specified by finite number of control points. For each control point $\vec{r}_i$, let us assign influence function $h(R_i)$, which expresses the influence of this control point at distance $R_i = |\vec{r} - \vec{r}_i|$. By definition, the solid contains those points where the total influence of the control points is not smaller than threshold T (Figure 22.8):

$$f(\vec{r}) = \sum_{i=0}^{m} h_i(R_i) - T \geq 0, \qquad \text{where } R_i = |\vec{r} - \vec{r}_i| \ .$$

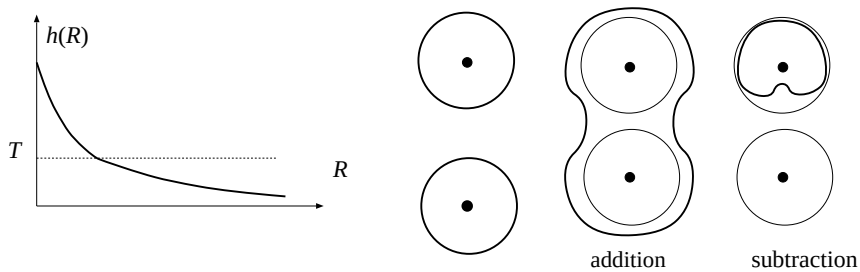


Figure 22.8 The influence decreases with the distance. Spheres of influence of similar signs increase, of different signs decrease each other.

With a single control point a sphere can be modeled. Spheres of multiple control points are combined together to result in an object having smooth surface (Figure 22.8). The influence of a single point can be defined by an arbitrary decreasing function that converges to zero at infinity. For example, Blinn proposed the

$$h_i(R) = a_i \cdot e^{-b_i R^2}$$

influence functions for his ***blob method.***

22.2.8. Constructive solid geometry

Another type of solid modelling is ***constructive solid geometry*** (***CSG*** for short), which builds complex solids from primitive solids applying set operations (e.g. union, intersection, difference, complement, etc.) (Figures 22.9 and 22.10). Primitives usually include the box, the sphere, the cone, the cylinder, the half-space, etc. whose functions are known.

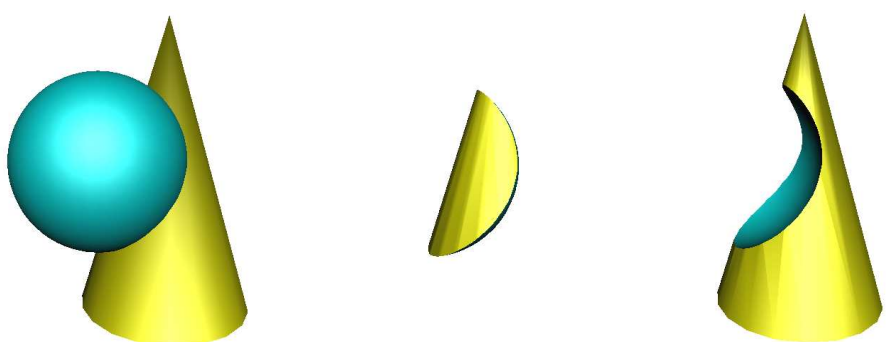


Figure 22.9 The operations of constructive solid geometry for a cone of implicit function f and for a sphere of implicit function g: union ($\max(f, g)$), intersection ($\min(f, g)$), and difference ($\min(f, -g)$).

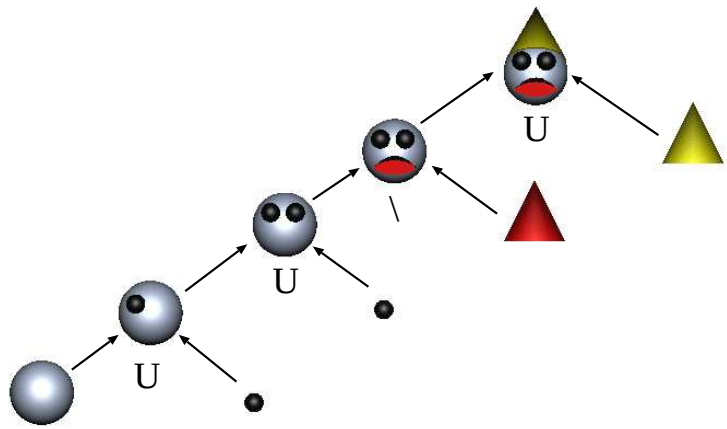


Figure 22.10 Constructing a complex solid by set operations. The root and the leaf of the ***CSG tree*** represents the complex solid, and the primitives, respectively. Other nodes define the set operations (U: union, \: difference).

The results of the set operations can be obtained from the implicit functions of the solids taking part of this operation:

- intersection of f and g: $\min(f, g)$;
- union of f and g: $\max(f, g)$.
- complement of f: $-f$.
- difference of f and g: $\min(f, -g)$.

Implicit functions also allow to ***morph*** between two solids. Suppose that two objects, for example, a box of implicit function f_1 and a sphere of implicit function f_2 need to be morphed. To define a new object, which is similar to the first object with percentage t and to the second object with percentage $(1 - t)$, the two implicit equations are weighted appropriately:

$$f^{morph}(x, y, z) = t \cdot f_1(x, y, z) + (1 - t) \cdot f_2(x, y, z) \ .$$

Exercises

22.2-1 Find the parametric equation of a torus.

22.2-2 Prove that the fourth-order B-spline with knot-vector [0,0,0,0,1,1,1,1] is a Bézier curve.

22.2-3 Give the equations for the surface points and the normals of the waving flag and waving water disturbed in a single point.

22.2-4 Prove that the tangents of a Bézier curve at the start and the end are the lines connecting the first two and the last two control points, respectively.

22.2-5 Give the algebraic forms of the basis functions of the second, the third, and the fourth-order B-splines.

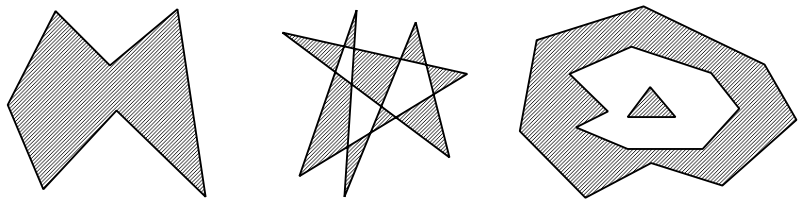


Figure 22.11 Types of polygons. **(a)** simple; **(b)** complex, single connected; **(c)** multiply connected.

22.2-6 Develop an algorithm computing the path length of a Bézier curve and a B-spline. Based on the path length computation move a point along the curve with uniform speed.

22.3. Geometry processing and tessellation algorithms

In Section 22.2 we met free-form surface and curve definition methods. During image synthesis, however, line segments and polygons play important roles. In this section we present methods that bridge the gap between these two types of representations. These methods convert geometric models to lines and polygons, or further process line and polygon models. Line segments connected to each other in a way that the end point of a line segment is the start point of the next one are called ***polylines.*** Polygons connected at edges, on the other hand, are called ***meshes. Vectorization*** methods approximate free-form curves by polylines. A polyline is defined by its vertices. ***Tessellation*** algorithms, on the other hand, approximate free-form surfaces by meshes. For illumination computation, we often need the normal vector of the original surface, which is usually stored with the vertices. Consequently, a mesh contains a list of polygons, where each polygon is given by its vertices and the normal of the original surface at these vertices. Methods processing meshes use other topology information as well, for example, which polygons share an edge or a vertex.

22.3.1. Polygon and polyhedron

Definition 22.2 *A **polygon** is a bounded part of the plane, i.e. it does not contain a line, and is bordered by line segments. A polygon is defined by the vertices of the bordering polylines.*

Definition 22.3 *A polygon is **single connected** if its border is a single closed polyline (Figure 22.11).*

Definition 22.4 *A polygon is **simple** if it is single connected and the bordering polyline does not intersect itself (Figure 22.11(a)).*

For a point of the plane, we can detect whether or not this point is inside the polygon by starting a half-line from this point and counting the number of

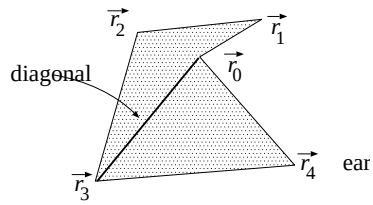


Figure 22.12 Diagonal and ear of a polygon.

intersections with the boundary. If the number of intersections is an odd number, then the point is inside, otherwise it is outside.

In the three-dimensional space we can form meshes, where different polygons are in different planes. In this case, two polygons are said to be neighboring if they share an edge.

Definition 22.5 *A **polyhedron** is a bounded part of the space, which is bordered by polygons.*

Similarly to polygons, a point can be tested for polyhedron inclusion by casting a half line from this point and counting the number of intersections with the face polygons. If the number of intersections is odd, then the point is inside the polyhedron, otherwise it is outside.

22.3.2. Vectorization of parametric curves

Parametric functions map interval $[t_{\min}, t_{\max}]$ onto the points of the curve. During vectorization the parameter interval is discretized. The simplest discretization scheme generates $N+1$ evenly spaced parameter values $t_i = t_{\min} + (t_{\max} - t_{\min}) \cdot i/N$ $(i = 0, 1, \ldots, N)$, and defines the approximating polyline by the points obtained by substituting these parameter values into parametric equation $\vec{r}(t_i)$.

22.3.3. Tessellation of simple polygons

Let us first consider the conversion of simple polygons to triangles. This is easy if the polygon is convex since we can select an arbitrary vertex and connect it with all other vertices, which decomposes the polygon to triangles in linear time. Unfortunately, this approach does not work for concave polygons since in this case the line segment connecting two vertices may go outside the polygon, thus cannot be the edge of one decomposing triangle.

Let us start the discussion of triangle conversion algorithms with two definitions:

Definition 22.6 *The **diagonal** of a polygon is a line segment connecting two vertices and is completely contained by the polygon (line segment $\vec{r}_0$ and $\vec{r}_3$ of Figure 22.12).*

The diagonal property can be checked for a line segment connecting two vertices by trying to intersect the line segment with all edges and showing that intersection is possible only at the endpoints, and additionally showing that one internal point of

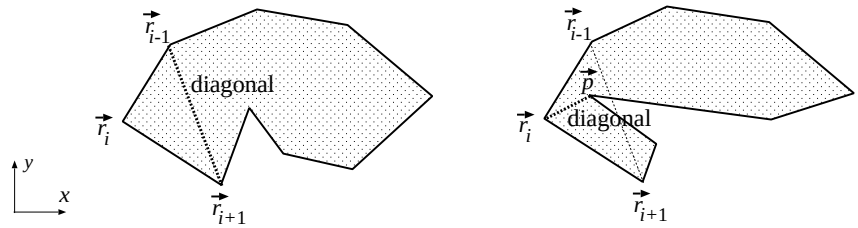


Figure 22.13 The proof of the existence of a diagonal for simple polygons.

the candidate is inside the polygon. For example, this test point can be the midpoint of the line segment.

Definition 22.7 *A vertex of the polygon is an* ***ear*** *if the line segment between the previous and the next vertices is a diagonal (vertex $\vec{r}_4$ of Figure 22.12).*

Clearly, only those vertices may be ears where the inner angle is not greater than 180 degrees. Such vertices are called ***convex vertices.***

For simple polygons the following theorems hold:

Theorem 22.8 *A simple polygon always has a diagonal.*

Proof Let the vertex standing at the left end (having the minimal x coordinate) be $\vec{r}_i$, and its two neighboring vertices be $\vec{r}_{i-1}$ and $\vec{r}_{i+1}$, respectively (Figure 22.13). Since $\vec{r}_i$ is standing at the left end, it is surely a convex vertex. If $\vec{r}_i$ is an ear, then line segment $(\vec{r}_{i-1}, \vec{r}_{i+1})$ is a diagonal (left of Figure 22.13), thus the theorem is proven for this case. Since $\vec{r}_i$ is a convex vertex, it is not an ear only if triangle $\vec{r}_{i-1}$, $\vec{r}_i$, $\vec{r}_{i+1}$ contains at least one polygon vertex (right of Figure 22.13). Let us select from the contained vertices that vertex $\vec{p}$ which is the farthest from the line defined by points $\vec{r}_{i-1}, \vec{r}_{i+1}$. Since there are no contained points which are farther from line $(\vec{r}_{i-1}, \vec{r}_{i+1})$ than $\vec{p}$, no edge can be between points $\vec{p}$ and $\vec{r}_i$, thus $(\vec{p}, \vec{r}_i)$ must be a diagonal. ■

Theorem 22.9 *A simple polygon can always be decomposed to triangles with its diagonals. If the number of vertices is n, then the number of triangles is $n-2$.*

Proof This theorem is proven by induction. The theorem is obviously true when $n = 3$, i.e. when the polygon is a triangle. Let us assume that the statement is also true for polygons having m $(m = 3, \ldots, n-1)$ number of vertices, and consider a polygon with n vertices. According to Theorem 22.8, this polygon of n vertices has a diagonal, thus we can subdivide this polygon into a polygon of n_1 vertices and a polygon of n_2 vertices, where $n_1, n_2 < n$, and $n_1 + n_2 = n + 2$ since the vertices at the ends of the diagonal participate in both polygons. According to the assumption of the induction, these two polygons can be separately decomposed to triangles. Joining the two sets of triangles, we can obtain the triangle decomposition of the original polygon. The number of triangles is $n_1 - 2 + n_2 - 2 = n - 2$. ■

The discussed proof is constructive thus it inspires a subdivision algorithm: let us find a diagonal, subdivide the polygon along this diagonal, and continue the same operation for the two new polygons.

Unfortunately the running time of such an algorithm is in $\Theta(n^3)$ since the number of diagonal candidates is $\Theta(n^2)$, and the time needed by checking whether or not a line segment is a diagonal is in $\Theta(n)$.

We also present a better algorithm, which decomposes a convex or concave polygon defined by vertices $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_n$. This algorithm is called ***ear cutting.*** The algorithm looks for ear triangles and cuts them until the polygon gets simplified to a single triangle. The algorithm starts at vertex $\vec{r}_2$. When a vertex is processed, it is first checked whether or not the previous vertex is an ear. If it is not an ear, then the next vertex is chosen. If the previous vertex is an ear, then the current vertex together with the two previous ones form a triangle that can be cut, and the previous vertex is deleted. If after deletion the new previous vertex has index 0, then the next vertex is selected as the current vertex.

The presented algorithm keeps cutting triangles until no more ears are left. The termination of the algorithm is guaranteed by the following ***two ears theorem:***

Theorem 22.10 *A simple polygon having at least four vertices always has at least two not neighboring ears that can be cut independently.*

Proof The proof presented here has been given by Joseph O'Rourke. According to theorem 22.9, every simple polygon can be subdivided to triangles such that the edges of these triangles are either the edges or the diagonals of the polygon. Let us make a correspondence between the triangles and the nodes of a graph where two nodes are connected if and only if the two triangles corresponding to these nodes share an edge. The resulting graph is connected and cannot contain circles. Graphs of these properties are trees. The name of this tree graph is the ***dual tree.*** Since the polygon has at least four vertices, the number of nodes in this tree is at least two. Any tree containing at least two nodes has at least two leaves[2]. Leaves of this tree, on the other hand, correspond to triangles having an ear vertex. ■

According to the two ears theorem, the presented algorithm finds an ear in $O(n)$ steps. Cutting an ear the number of vertices is reduced by one, thus the algorithm terminates in $O(n^2)$ steps.

22.3.4. Tessellation of parametric surfaces

Parametric forms of surfaces map parameter rectangle $[u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$ onto the points of the surface.

In order to tessellate the surface, first the parameter rectangle is subdivided to triangles. Then applying the parametric equations for the vertices of the parameter triangles, the approximating triangle mesh can be obtained. The simplest subdivision of the parametric rectangle decomposes the domain of parameter u to N parts, and

[2] A leaf is a node connected by exactly one edge.

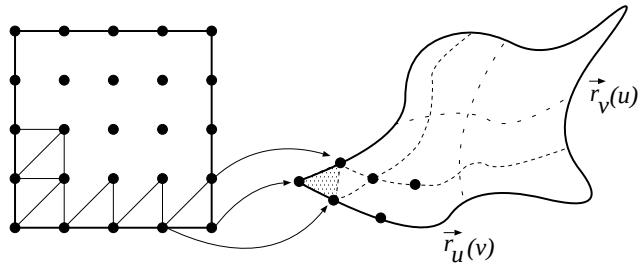


Figure 22.14 Tessellation of parametric surfaces.

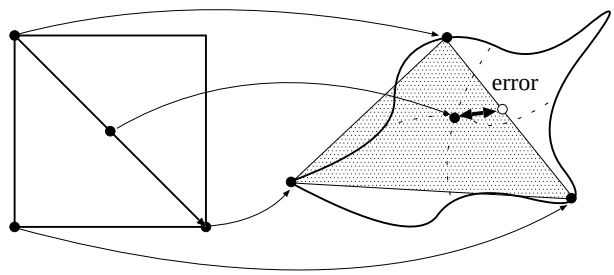


Figure 22.15 Estimation of the tessellation error.

the domain of parameter v to M intervals, resulting in the following parameter pairs:

$$[u_i, v_j] = \left[u_{\min} + (u_{\max} - u_{\min})\frac{i}{N},\ v_{\min} + (v_{\max} - v_{\min})\frac{j}{M} \right].$$

Taking these parameter pairs and substituting them into the parametric equations, point triplets $\vec{r}(u_i, v_j)$, $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_i, v_{j+1})$, and point triplets $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_{i+1}, v_{j+1})$, $\vec{r}(u_i, v_{j+1})$ are used to define triangles.

The tessellation process can be made ***adaptive*** as well, which uses small triangles only where the high curvature of the surface justifies them. Let us start with the parameter rectangle and subdivide it to two triangles. In order to check the accuracy of the resulting triangle mesh, surface points corresponding to the edge midpoints of the parameter triangles are compared to the edge midpoints of the approximating triangles. Formally the following distance is computed (Figure 22.15):

$$\left| \vec{r}\left(\frac{u_1 + u_2}{2}, \frac{v_1 + v_2}{2} \right) - \frac{\vec{r}(u_1, v_1) + \vec{r}(u_2, v_2)}{2} \right|,$$

where (u_1, v_1) and (u_2, v_2) are the parameters of the two endpoints of the edge.

A large distance value indicates that the triangle mesh poorly approximates the parametric surface, thus triangles must be subdivided further. This subdivision can

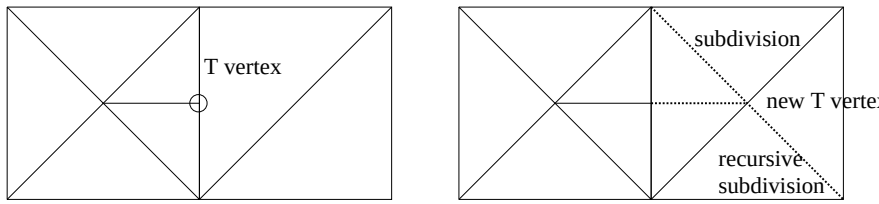


Figure 22.16 T vertices and their elimination with forced subdivision.

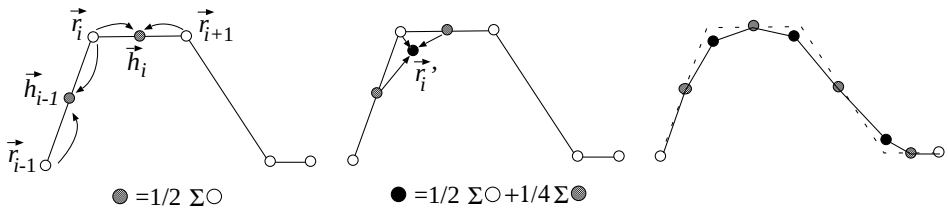


Figure 22.17 Construction of a subdivision curve: at each step midpoints are obtained, then the original vertices are moved to the weighted average of neighbouring midpoints and of the original vertex.

be executed by cutting the triangle to two triangles by a line connecting the midpoint of the edge of the largest error and the opposing vertex. Alternatively, a triangle can be subdivided to four triangles with its halving lines. The adaptive tessellation is not necessarily robust since it can happen that the distance at the midpoint is small, but at other points is still quite large.

When the adaptive tessellation is executed, it may happen that one triangle is subdivided while its neighbour is not, which results in holes. Such problematic midpoints are called ***T vertices*** (Figure 22.16).

If the subdivision criterion is based only on edge properties, then T vertices cannot show up. However, if other properties are also taken into account, then T vertices may appear. In such cases, T vertices can be eliminated by recursively forcing the subdivision also for those neighbouring triangles that share subdivided edges.

22.3.5. Subdivision curves and meshes

This section presents algorithms that smooth polyline and mesh models.

Let us consider a polyline of vertices $\vec{r}_0, \ldots, \vec{r}_m$. A smoother polyline is generated by the following vertex doubling approach (Figure 22.17). Every line segment of the polyline is halved, and midpoints $\vec{h}_0, \ldots, \vec{h}_{m-1}$ are added to the polyline as new vertices. Then the old vertices are moved taking into account their old position and the positions of the two enclosing midpoints, applying the following weighting:

$$\vec{r}_i{}' = \frac{1}{2}\vec{r}_i + \frac{1}{4}\vec{h}_{i-1} + \frac{1}{4}\vec{h}_i = \frac{3}{4}\vec{r}_i + \frac{1}{8}\vec{r}_{i-1} + \frac{1}{8}\vec{r}_{i+1} .$$

The new polyline looks much smoother. If we should not be satisfied with the smoothness yet, the same procedure can be repeated recursively. As can be shown, the result

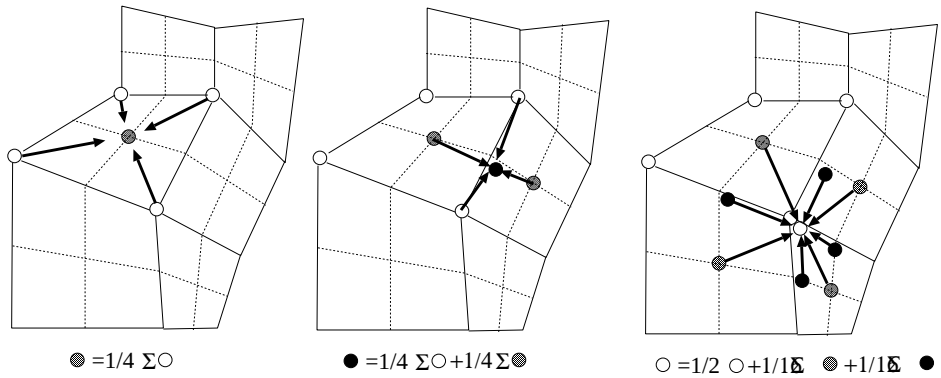


Figure 22.18 One smoothing step of the Catmull-Clark subdivision. First the face points are obtained, then the edge midpoints are moved, and finally the original vertices are refined according to the weighted sum of its neighbouring edge and face points.

of the recursive process converges to the B-spline curve.

The polyline subdivision approach can also be extended for smoothing three-dimensional meshes. This method is called ***Catmull-Clark subdivision algorithm.*** Let us consider a three-dimensional quadrilateral mesh (Figure 22.18). In the first step the midpoints of the edges are obtained, which are called ***edge points.*** Then ***face points*** are generated as the average of the vertices of each face polygon. Connecting the edge points with the face points, we still have the original surface, but now defined by four times more quadrilaterals. The smoothing step modifies first the edge points setting them to the average of the vertices at the ends of the edge and of the face points of those quads that share this edge. Then the original vertices are moved to the weighted average of the face points of those faces that share this vertex, and of edge points of those edges that are connected to this vertex. The weight of the original vertex is $1/2$, the weights of edge and face points are $1/16$. Again, this operation may be repeated until the surface looks smooth enough (Figure 22.19).

If we do not want to smooth the mesh at an edge or around a vertex, then the averaging operation ignores the vertices on the other side of the edge to be preserved.

The Catmull-Clark subdivision surface usually does not interpolate the original vertices. This drawback is eliminated by the ***butterfly subdivision,*** which works on triangle meshes. First the butterfly algorithm puts new edge points close to the midpoints of the original edges, then the original triangle is replaced by four triangles defined by the original vertices and the new edge points (Figure 22.20). The position of the new edge points depend on the vertices of those two triangles incident to this edge, and on those four triangles which share edges with these two. The arrangement of the triangles affecting the edge point resembles a butterfly, hence the name of this algorithm. The edge point coordinates are obtained as a weighted sum of the edge endpoints multiplied by $1/2$, the third vertices of the triangles sharing this edge using weight $1/8 + 2w$, and finally of the other vertices of the additional triangles with weight $-1/16-w$. Parameter w can control the curvature of the resulting mesh.

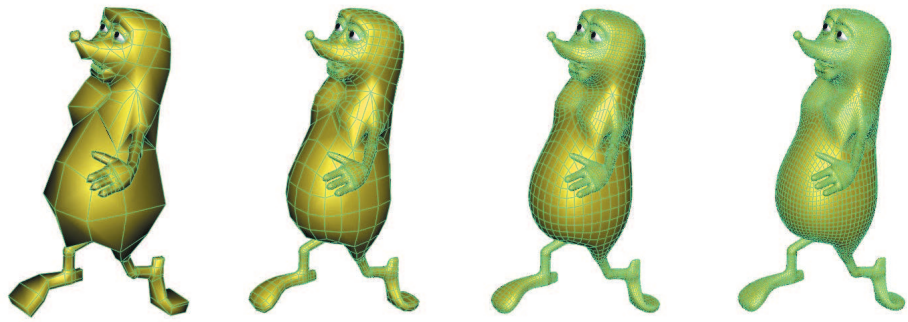


Figure 22.19 Original mesh and its subdivision applying the smoothing step once, twice and three times, respectively.

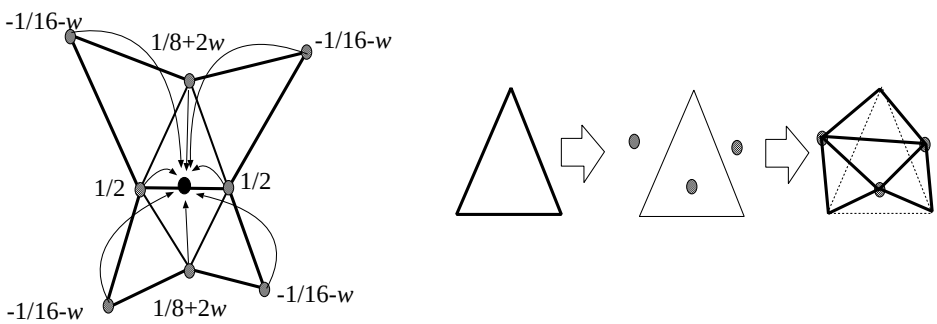


Figure 22.20 Generation of the new edge point with butterfly subdivision.

Setting $w = -1/16$, the mesh keeps its original faceted look, while $w = 0$ results in strong rounding.

22.3.6. Tessellation of implicit surfaces

A surface defined by implicit equation $f(x, y, z) = 0$ can be converted to a triangle mesh by finding points on the surface densely, i.e. generating points satisfying $f(x, y, z) \approx 0$, then assuming the close points to be vertices of the triangles.

First function f is evaluated at the grid points of the Cartesian coordinate system and the results are stored in a three-dimensional array, called ***voxel array.*** Let us call two grid points as neighbours if two of their coordinates are identical and the difference in their third coordinate is 1. The function is evaluated at the grid points and is assumed to be linear between them. The normal vectors needed for shading are obtained as the gradient of function f (equation 22.4), which are also interpolated between the grid points.

When we work with the voxel array, original function f is replaced by its ***tri-***

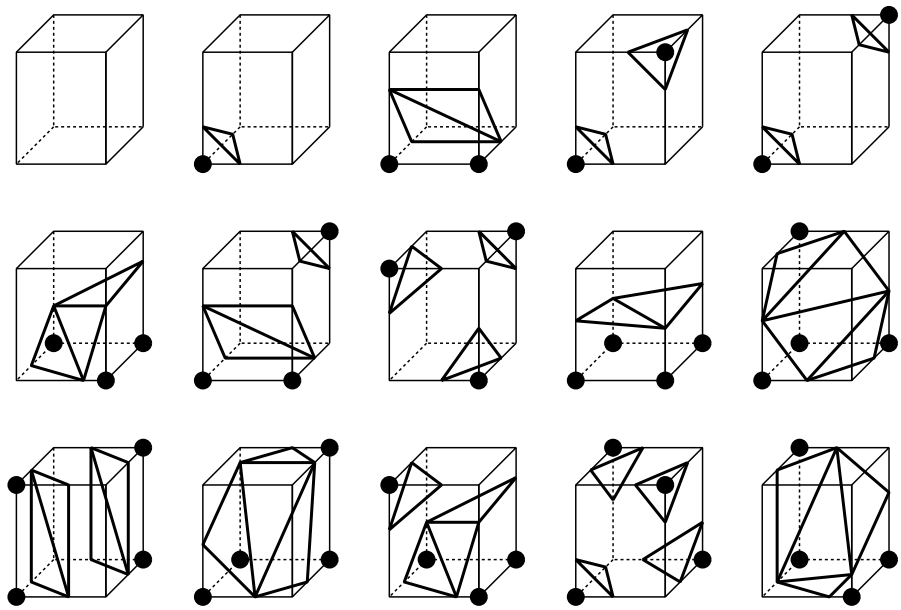


Figure 22.21 Possible intersections of the per-voxel tri-linear implicit surface and the voxel edges. From the possible 256 cases, these 15 topologically different cases can be identified, from which the others can be obtained by rotations. Grid points where the implicit function has the same sign are depicted by circles.

linear approximation (tri-linear means that fixing any two coordinates the function is linear with respect to the third coordinate). Due to the linear approximation an edge connecting two neighbouring grid points can intersect the surface at most once since linear equations may have at most one root. The density of the grid points should reflect this observation, then we have to define them so densely not to miss roots, that is, not to change the topology of the surface.

The method approximating the surface by a triangle mesh is called ***marching cubes algorithm.*** This algorithm first decides whether a grid point is inside or outside of the solid by checking the sign of function f. If two neighbouring grid points are of different type, the surface must go between them. The intersection of the surface and the edge between the neighbouring points, as well as the normal vector at the intersection are determined by linear interpolation. If one grid point is at $\vec{r}_1$, the other is at $\vec{r}_2$, and function f has different signs at these points, then the intersection of the tri-linear surface and line segment $(\vec{r}_1, \vec{r}_2)$ is:

$$\vec{r}_i = \vec{r}_1 \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \vec{r}_2 \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} \ .$$

The normal vector here is:

$$\vec{n}_i = \mathrm{grad} f(\vec{r}_1) \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \mathrm{grad} f(\vec{r}_2) \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} \ .$$

Having found the intersection points, triangles are defined using these points

as vertices. When defining these triangles, we have to take into account that a trilinear surface may intersect the voxel edges at most once. Such intersection occurs if function f has different signs at the two grid points. The number of possible variations of positive/negative signs at the 8 vertices of a cube is 256, from which 15 topologically different cases can be identified (Figure 22.21).

The algorithm inspects grid points one by one and assigns the sign of the function to them encoding negative sign by 0 and non-negative sign by 1. The resulting 8 bit code is a number in 0–255 which identifies the current case of intersection. If the code is 0, all voxel vertices are outside the solid, thus no voxel surface intersection is possible. Similarly, if the code is 255, the solid is completely inside, making the intersections impossible. To handle other codes, a table can be built which describes where the intersections show up and how they form triangles.

Exercises

22.3-1 Prove the two ears theorem by induction.
22.3-2 Develop an adaptive curve tessellation algorithm.
22.3-3 Prove that the Catmull-Clark subdivision curve and surface converge to a B-spline curve and surface, respectively.
22.3-4 Build a table to control the marching cubes algorithm, which describes where the intersections show up and how they form triangles.
22.3-5 Propose a marching cubes algorithm that does not require the gradients of the function, but estimates these gradients from its values.

22.4. Containment algorithms

When geometric models are processed, we often have to determine whether or not one object contains points belonging to the other object. If only yes/no answer is needed, we have a ***containment test*** problem. However, if the contained part also needs to be obtained, the applicable algorithm is called ***clipping.***

Containment test is also known as discrete time ***collision detection*** since if one object contains points from the other, then the two objects must have been collided before. Of course, checking collisions just at discrete time instances may miss certain collisions. To handle the collision problem robustly, continuous time collision detection is needed which also computes the time of the collision. Continuous time collision detection may use ray tracing (Section 22.6). In this section we only deal with the discrete time collision detection and the clipping of simple objects.

22.4.1. Point containment test

A solid defined by function f contains those (x, y, z) points which satisfy inequality $f(x, y, z) \geq 0$. It means that point containment test requires the evaluation of function f and the inspection of the sign of the result.

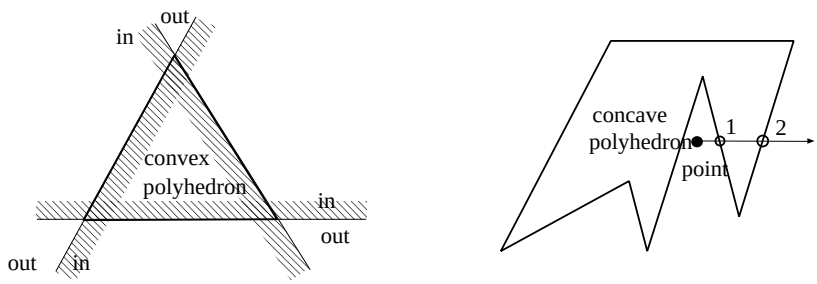


Figure 22.22 Polyhedron-point containment test. A convex polyhedron contains a point if the point is on that side of each face plane where the polyhedron is. To test a concave polyhedron, a half line is cast from the point and the number of intersections is counted. If the result is an odd number, then the point is inside, otherwise it is outside.

Half space. Based on equation (22.1), points belonging to a half space are identified by inequality

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0 \ , \tag{22.9}$$

where the normal vector is supposed to point inward.

Convex polyhedron. Any convex polyhedron can be constructed as the intersection of halfspaces (left of Figure 22.22). The plane of each face subdivides the space into two parts, to an inner part where the polyhedron can be found, and to an outer part. Let us test the point against the planes of the faces. If the point is in the inner part with respect to all planes, then the point is inside the polyhedron. However, if the point is in the outer part with respect to at least one plane, then the point is outside of the polyhedron.

Concave polyhedron. As shown in Figure 22.22, let us cast a half line from the tested point and count the number of intersections with the faces of the polyhedron (the calculation of these intersections is discussed in Section 22.6). If the result is an odd number, then the point is inside, otherwise it is outside. Because of numerical inaccuracies we might have difficulties to count the number of intersections when the half line is close to the edges. In such cases, the simplest solution is to find another half line and carry out the test with that.

Polygon. The methods proposed to test the point in polyhedron can also be used for polygons limiting the space to the two-dimensional plane. For example, a point is in a general polygon if the half line originating at this point and lying in the plane of the polygon intersects the edges of the polygon odd times.

In addition to those methods, containment in convex polygons can be tested by adding the angles subtended by the edges from the point. If the sum is 360 degrees, then the point is inside, otherwise it is outside. For convex polygons, we can also test whether the point is on the same side of the edges as the polygon itself. This

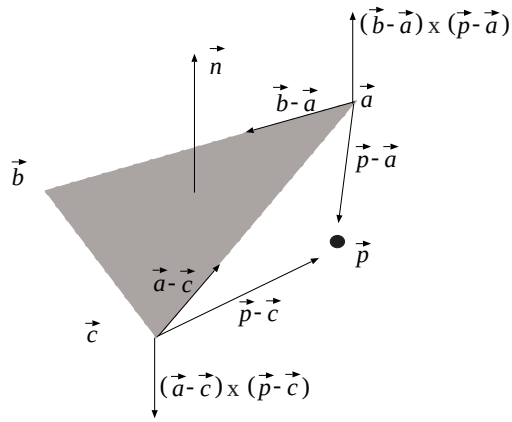


Figure 22.23 Point in triangle containment test. The figure shows that case when point $\vec{p}$ is on the left of oriented lines $\vec{ab}$ and $\vec{bc}$, and on the right of line $\vec{ca}$, that is, when it is not inside the triangle.

algorithm is examined in details for a particularly important special case, when the polygon is a triangle.

Triangle. Let us consider a triangle of vertices $\vec{a}, \vec{b}$ and $\vec{c}$, and point $\vec{p}$ lying in the plane of the triangle. The point is inside the triangle if and only if it is on the same side of the boundary lines as the third vertex. Note that cross product $(\vec{b}-\vec{a})\times(\vec{p}-\vec{a})$ has a different direction for point $\vec{p}$ lying on the different sides of oriented line $\vec{ab}$, thus the direction of this vector can be used to classify points (should point $\vec{p}$ be on line $\vec{ab}$, the result of the cross product is zero). During classification the direction of $(\vec{b}-\vec{a})\times(\vec{p}-\vec{a})$ is compared to the direction of vector $\vec{n}=(\vec{b}-\vec{a})\times(\vec{c}-\vec{a})$ where tested point $\vec{p}$ is replaced by third vertex $\vec{c}$. Note that vector $\vec{n}$ happens to be the normal vector of the triangle plane (Figure 22.23).

We can determine whether two vectors have the same direction (their angle is zero) or they have opposite directions (their angle is 180 degrees) by computing their scalar product and looking at the sign of the result. The scalar product of vectors of similar directions is positive. Thus if scalar product $((\vec{b}-\vec{a})\times(\vec{p}-\vec{a}))\cdot\vec{n}$ is positive, then point $\vec{p}$ is on the same side of oriented line $\vec{ab}$ as $\vec{c}$. On the other hand, if this scalar product is negative, then $\vec{p}$ and $\vec{c}$ are on the opposite sides. Finally, if the result is zero, then point $\vec{p}$ is on line $\vec{ab}$. Point $\vec{p}$ is inside the triangle if and only if all the following three conditions are met:

$$\begin{aligned} ((\vec{b}-\vec{a})\times(\vec{p}-\vec{a}))\cdot\vec{n} &\geq 0\,, \\ ((\vec{c}-\vec{b})\times(\vec{p}-\vec{b}))\cdot\vec{n} &\geq 0\,, \\ ((\vec{a}-\vec{c})\times(\vec{p}-\vec{c}))\cdot\vec{n} &\geq 0\,. \end{aligned} \tag{22.10}$$

This test is robust since it gives correct result even if – due to numerical precision problems – point $\vec{p}$ is not exactly in the plane of the triangle as long as point $\vec{p}$ is in the prism obtained by perpendicularly extruding the triangle from the plane.

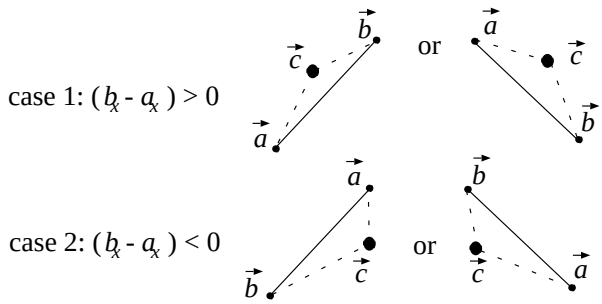


Figure 22.24 Point in triangle containment test on coordinate plane xy. Third vertex $\vec{c}$ can be either on the left or on the right side of oriented line $\vec{ab}$, which can always be traced back to the case of being on the left side by exchanging the vertices.

The evaluation of the test can be speeded up if we work in a two-dimensional projection plane instead of the three-dimensional space. Let us project point $\vec{p}$ as well as the triangle onto one of the coordinate planes. In order to increase numerical precision, that coordinate plane should be selected on which the area of the projected triangle is maximal. Let us denote the Cartesian coordinates of the normal vector by (n_x, n_y, n_z). If n_z has the maximum absolute value, then the projection of the maximum area is on coordinate plane xy. If n_x or n_y had the maximum absolute value, then planes yz or xz would be the right choice. Here only the case of maximum n_z is discussed.

First the order of vertices are changed in a way that when travelling from vertex $\vec{a}$ to vertex $\vec{b}$, vertex $\vec{c}$ is on the left side. Let us examine the equation of line $\vec{ab}$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (x - b_x) + b_y = y \ .$$

According to Figure 22.24 point $\vec{c}$ is on the left of the line if c_y is above the line at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y < c_y \ .$$

Multiplying both sides by $(b_x - a_x)$, we get:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

In the second case the denominator of the slope of the line is negative. Point $\vec{c}$ is on the left of the line if c_y is below the line at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y > c_y \ .$$

When the inequality is multiplied with negative denominator $(b_x - a_x)$, the relation is inverted:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

Note that in both cases we obtained the same condition. If this condition is not met,

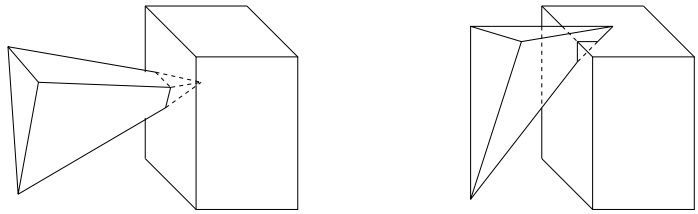


Figure 22.25 Polyhedron-polyhedron collision detection. Only a part of collision cases can be recognized by testing the containment of the vertices of one object with respect to the other object. Collision can also occur when only edges meet, but vertices do not penetrate to the other object.

then point $\vec{c}$ is not on the left of line $\vec{ab}$, but is on the right. Exchanging vertices $\vec{a}$ and $\vec{b}$ in this case, we can guarantee that $\vec{c}$ will be on the left of the new line $\vec{ab}$. It is also important to note that consequently point $\vec{a}$ will be on the left of line $\vec{bc}$ and point $\vec{b}$ will be on the left of line $\vec{ca}$.

In the second step the algorithm tests whether point $\vec{p}$ is on the left with respect to all three boundary lines since this is the necessary and sufficient condition of being inside the triangle:

$$\begin{aligned}(b_y - a_y) \cdot (p_x - b_x) &\leq (p_y - b_y) \cdot (b_x - a_x) \ , \\ (c_y - b_y) \cdot (p_x - c_x) &\leq (p_y - c_y) \cdot (c_x - b_x) \ , \\ (a_y - c_y) \cdot (p_x - a_x) &\leq (p_y - a_y) \cdot (a_x - c_x) \ .\end{aligned} \tag{22.11}$$

22.4.2. Polyhedron-polyhedron collision detection

Two polyhedra collide when a vertex of one of them meets a face of the other, and if they are not bounced off, the vertex goes into the internal part of the other object (Figure 22.25). This case can be recognized with the discussed containment test. All vertices of one polyhedron is tested for containment against the other polyhedron. Then the roles of the two polyhedra are exchanged.

Apart from the collision between vertices and faces, two edges may also meet without vertex penetration (Figure 22.25). In order to recognize this edge penetration case, all edges of one polyhedron are tested against all faces of the other polyhedron. The test for an edge and a face is started by checking whether or not the two endpoints of the edge are on opposite sides of the plane, using inequality (22.9). If they are, then the intersection of the edge and the plane is calculated, and finally it is decided whether the face contains the intersection point.

Polyhedra collision detection tests each edge of one polyhedron against each face of the other polyhedron, which results in an algorithm of quadratic time complexity with respect to the number of vertices of the polyhedra. Fortunately, the algorithm can be speeded up applying bounding volumes (Subsection 22.6.2). Let us assign a simple bounding object to each polyhedron. Popular choices for bounding volumes are the sphere and the box. During testing the collision of two objects, first their bounding volumes are examined. If the two bounding volumes do not collide, then neither can the contained polyhedra collide. If the bounding volumes penetrate each

other, then one polyhedra is tested against the other bounding volume. If this test is also positive, then finally the two polyhedra are tested. However, this last test is rarely required, and most of the collision cases can be solved by bounding volumes.

22.4.3. Clipping algorithms

Clipping takes an object defining the clipping region and removes those points from another object which are outside the clipping region. Clipping may alter the type of the object, which cannot be specified by a similar equation after clipping. To avoid this, we allow only those kinds of clipping regions and objects where the object type is not changed by clipping. Let us assume that the clipping region is a half space or a polyhedron, while the object to be clipped is a point, a line segment or a polygon.

If the object to be clipped is a point, then containment can be tested with the algorithms of the previous subsection. Based on the result of the containment test, the point is either removed or preserved.

Clipping a line segment onto a half space. Let us consider a line segment of endpoints $\vec{r}_1$ and $\vec{r}_2$, and of equation $\vec{r}(t) = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t$, $(t \in [0,1])$, and a half plane defined by the following equation derived from equation (22.1):

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0 \; .$$

Three cases need to be distinguished:

1. If both endpoints of the line segment are in the half space, then all points of the line segment are inside, thus the whole segment is preserved.

2. If both endpoints are out of the half space, then all points of the line segment are out, thus the line segment should be completely removed.

3. If one of the endpoints is out, while the other is in, then the endpoint being out should be replaced by the intersection point of the line segment and the boundary plane of the half space. The intersection point can be calculated by substituting the equation of the line segment into the equation of the boundary plane and solving the resulting equation for the unknown parameter:

$$(\vec{r}_1 \cdot (1-t_i) + \vec{r}_2 \cdot t_i - \vec{r}_0) \cdot \vec{n} = 0, \quad \Longrightarrow \quad t_i = \frac{(\vec{r}_0 - \vec{r}_1) \cdot \vec{n}}{(\vec{r}_2 - \vec{r}_1) \cdot \vec{n}} \; .$$

Substituting parameter t_i into the equation of the line segment, the coordinates of the intersection point can also be obtained.

Clipping a polygon onto a half space. This clipping algorithm tests first whether a vertex is inside or not. If the vertex is in, then it is also the vertex of the resulting polygon. However, if it is out, it can be ignored. On the other hand, the resulting polygon may have vertices other than the vertices of the original polygon. These new vertices are the intersections of the edges and the boundary plane of the

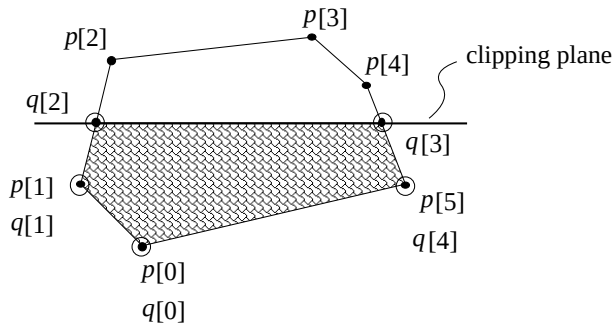


Figure 22.26 Clipping of simple convex polygon $\vec{p}[0], \ldots, \vec{p}[5]$ results in polygon $\vec{q}[0], \ldots, \vec{q}[4]$. The vertices of the resulting polygon are the inner vertices of the original polygon and the intersections of the edges and the boundary plane.

half space. Such intersection occurs when one endpoint is in, but the other is out. While we are testing the vertices one by one, we should also check whether or not the next vertex is on the same side as the current vertex (Figure 22.26).

Suppose that the vertices of the polygon to be clipped are given in array $\mathbf{p} = \langle \vec{p}[0], \ldots, \vec{p}[n-1] \rangle$, and the vertices of the clipped polygon is expected in array $\mathbf{q} = \langle \vec{q}[0], \ldots, \vec{q}[m-1] \rangle$. The number of the vertices of the resulting polygon is stored in variable m. Note that the vertex followed by the ith vertex has usually index $(i+1)$, but not in the case of the last, $(n-1)$th vertex, which is followed by vertex 0. Handling the last vertex as a special case is often inconvenient. This can be eliminated by extending input array $\mathbf{p}$ by new element $\vec{p}[n] = \vec{p}[0]$, which holds the element of index 0 once again.

Using these assumptions, the ***Sutherland-Hodgeman polygon clipping algorithm*** is:

SUTHERLAND-HODGEMAN-POLYGON-CLIPPING($\mathbf{p}$)

```
1  m ← 0
2  for i ← 0 to n − 1
3      do if p⃗[i] is inside
4           then q⃗[m] ← p⃗[i]                    ▷ The ith vertex is the vertex
                                                  ▷ of the resulting polygon.
5                m ← m + 1
6                if p⃗[i + 1] is outside
7                  then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
8                       m ← m + 1
9           else if p⃗[i + 1] is inside
10                 then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
11                      m ← m + 1
12 return q
```

Let us apply this algorithm for such a concave polygon which is expected to fall

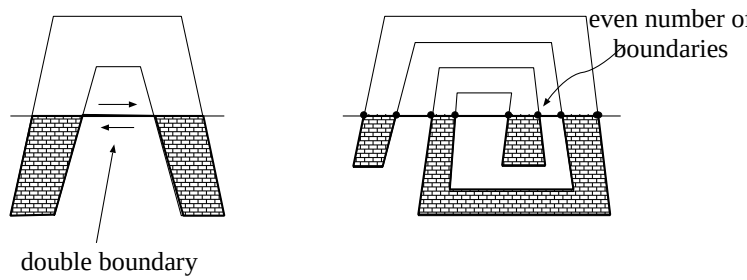


Figure 22.27 When concave polygons are clipped, the parts that should fall apart are connected by even number of edges.

to several pieces during clipping (Figure 22.27). The algorithm storing the polygon in a single array is not able to separate the pieces and introduces even number of edges at parts where no edge could show up.

These even number of extra edges, however, pose no problems if the interior of the polygon is defined as follows: a point is inside the polygon if and only if starting a half line from here, the boundary polyline is intersected by odd number of times.

The presented algorithm is also suitable for clipping multiple connected polygons if the algorithm is executed separately for each closed polyline of the boundary.

Clipping line segments and polygons on a convex polyhedron. As stated, a convex polyhedron can be obtained as the intersection of the half spaces defined by the planes of the polyhedron faces (left of Figure 22.22). It means that clipping on a convex polyhedron can be traced back to a series of clipping steps on half spaces. The result of one clipping step on a half plane is the input of clipping on the next half space. The final result is the output of the clipping on the last half space.

Clipping a line segment on an AABB. Axis aligned bounding boxes, abbreviated as AABBs, play an important role in image synthesis.

Definition 22.11 *A box aligned parallel to the coordinate axes is called* ***AABB****. An AABB is specified with the minimum and maximum Cartesian coordinates:* $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$.

Although when an object is clipped on an AABB, the general algorithms that clip on a convex polyhedron could also be used, the importance of AABBs is acknowledged by developing algorithms specially tuned for this case.

When a line segment is clipped to a polyhedron, the algorithm would test the line segment with the plane of each face, and the calculated intersection points may turn out to be unnecessary later. We should thus find an appropriate order of planes which makes the number of unnecessary intersection calculations minimal. A simple method that implements this idea is the ***Cohen-Sutherland line clipping algorithm.***

Let us assign code bit 1 to a point that is outside with respect to a clipping plane, and code bit 0 if the point is inside with respect to this plane. Since an AABB has

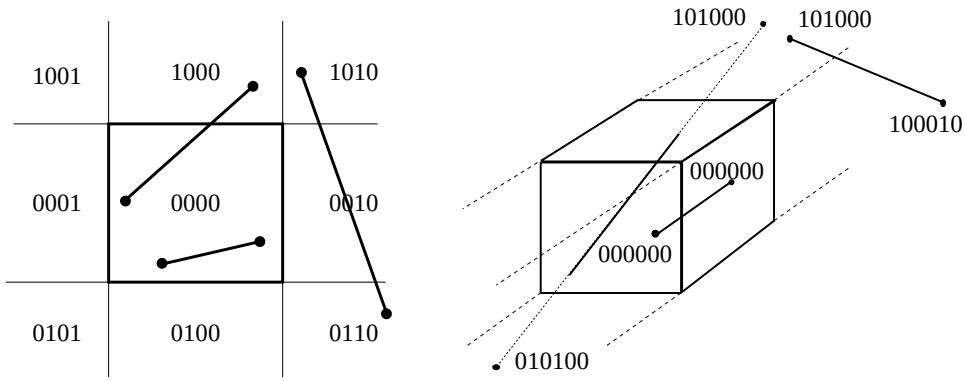


Figure 22.28 The 4-bit codes of the points in a plane and the 6-bit codes of the points in space.

6 sides, we get 6 bits forming a 6-bit code word (Figure 22.28). The interpretation of code bits $C[0], \ldots, C[5]$ is the following:

$$C[0] = \begin{cases} 1, & x \le x_{min}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[1] = \begin{cases} 1, & x \ge x_{max}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[2] = \begin{cases} 1, & y \le y_{min}\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

$$C[3] = \begin{cases} 1, & y \ge y_{max}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[4] = \begin{cases} 1, & z \le z_{min}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[5] = \begin{cases} 1, & z \ge z_{max}\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

Points of code word 000000 are obviously inside, points of other code words are outside (Figure 22.28). Let the code words of the two endpoints of the line segment be C_1 and C_2, respectively. If both of them are zero, then both endpoints are inside, thus the line segment is completely inside (trivial accept). If the two code words contain bit 1 at the same location, then none of the endpoints are inside with respect to the plane associated with this code bit. This means that the complete line segment is outside with respect to this plane, and can be rejected (trivial reject). This examination can be executed by applying the bitwise AND operation on code words C_1 and C_2 (with the notations of the C programming language C_1 & C_2), and checking whether or not the result is zero. If it is not zero, there is a bit where both code words have value 1.

Finally, if none of the two trivial cases hold, then there must be a bit which is 0 in one code word and 1 in the other. This means that one endpoint is inside and the other is outside with respect to the plane corresponding to this bit. The line segment should be clipped on this plane. Then the same procedure should be repeated starting with the evaluation of the code bits. The procedure is terminated when the conditions of either the trivial accept or the trivial reject are met.

The Cohen-Sutherland line clipping algorithm returns the endpoints of the clipped line by modifying the original vertices and indicates with TRUE return value if the line is not completely rejected:

```
COHEN-SUTHERLAND-LINE-CLIPPING(r⃗1, r⃗2)
 1  C1 ← codeword of r⃗1                    ▷ Code bits by checking the inequalities.
 2  C2 ← codeword of r⃗2
 3  while TRUE
 4        do if C1 = 0 AND C2 = 0
 5              then return TRUE             ▷ Trivial accept: inner line segment exists.
 6           if C1 & C2 ≠ 0
 7              then return FALSE            ▷ Trivial reject: no inner line segment exists.
 8           f ← index of the first bit where C1 and C2 differ
 9           r⃗i ← intersection of line segment (r⃗1, r⃗2) and the plane of index f
10           Ci ← codeword of r⃗i
11           if C1[f] = 1
12              then r⃗1 ← r⃗i
13                   C1 ← Ci                 ▷ r⃗1 is outside w.r.t. plane f.
14              else r⃗2 ← r⃗i
15                   C2 ← Ci                 ▷ r⃗2 is outside w.r.t. plane f.
```

Exercises

22.4-1 Propose approaches to reduce the quadratic complexity of polyhedron-polyhedron collision detection.

22.4-2 Develop a containment test to check whether a point is in a CSG-tree.

22.4-3 Develop an algorithm clipping one polygon onto a concave polygon.

22.4-4 Find an algorithm computing the bounding sphere and the bounding AABB of a polyhedron.

22.4-5 Develop an algorithm that tests the collision of two triangles in the plane.

22.4-6 Generalize the Cohen-Sutherland line clipping algorithm to convex polyhedron clipping region.

22.4-7 Propose a method for clipping a line segment on a sphere.

22.5. Translation, distortion, geometric transformations

Objects in the virtual world may move, get distorted, grow or shrink, that is, their equations may also depend on time. To describe dynamic geometry, we usually apply two functions. The first function selects those points of space, which belong to the object in its reference state. The second function maps these points onto points defining the object in an arbitrary time instance. Functions mapping the space onto itself are called ***transformations.*** A transformation maps point $\vec{r}$ to point $\vec{r}\,' = \mathcal{T}(\vec{r})$. If the transformation is invertible, we can also find the original for some transformed point $\vec{r}\,'$ using inverse transformation $\mathcal{T}^{-1}(\vec{r}\,')$.

If the object is defined in its reference state by inequality $f(\vec{r}) \geq 0$, then the points of the transformed object are

$$\{\vec{r}\,' : f(\mathcal{T}^{-1}(\vec{r}\,')) \geq 0\}\,, \tag{22.12}$$

since the originals belong to the set of points of the reference state.

Parametric equations define the Cartesian coordinates of the points directly. Thus the transformation of parametric surface $\vec{r} = \vec{r}(u, v)$ requires the transformations of its points

$$\vec{r}\,'(u, v) = \mathcal{T}(\vec{r}(u, v)) \ . \tag{22.13}$$

Similarly, the transformation of curve $\vec{r} = \vec{r}(t)$ is:

$$\vec{r}\,'(t) = \mathcal{T}(\vec{r}(t)) \ . \tag{22.14}$$

Transformation $\mathcal{T}$ may change the type of object in the general case. It can happen, for example, that a simple triangle or a sphere becomes a complicated shape, which are hard to describe and handle. Thus it is worth limiting the set of allowed transformations. Transformations mapping planes onto planes, lines onto lines and points onto points are particularly important. In the next subsection we consider the class of ***homogeneous linear transformations,*** which meet this requirement.

22.5.1. Projective geometry and homogeneous coordinates

So far the construction of the virtual world has been discussed using the means of the Euclidean geometry, which gave us many important concepts such as distance, parallelism, angle, etc. However, when the transformations are discussed in details, many of these concepts are unimportant, and can cause confusion. For example, parallelism is a relationship of two lines which can lead to singularities when the intersection of two lines is considered. Therefore, transformations are discussed in the context of another framework, called projective geometry.

The axioms of ***projective geometry*** turn around the problem of parallel lines by ignoring the concept of parallelism altogether, and state that two different lines always have an intersection. To cope with this requirement, every line is extended by a "point at infinity" such that two lines have the same extra point if and only if the two lines are parallel. The extra point is called the ***ideal point.*** The ***projective space*** contains the points of the Euclidean space (these are the so called ***affine points***) and the ideal points. An ideal point "glues" the "ends" of an Euclidean line, making it topologically similar to a circle. Projective geometry preserves that axiom of the Euclidean geometry which states that two points define a line. In order to make it valid for ideal points as well, the set of lines of the Euclidean space is extended by a new line containing the ideal points. This new line is called the ***ideal line.*** Since the ideal points of two lines are the same if and only if the two lines are parallel, the ideal lines of two planes are the same if and only if the two planes are parallel. Ideal lines are on the ***ideal plane,*** which is added to the set of planes of the Euclidean space. Having made these extensions, no distinction is needed between the affine and ideal points. They are equal members of the projective space.

Introducing analytic geometry we noted that everything should be described by numbers in computer graphics. Cartesian coordinates used so far are in one to one relationship with the points of Euclidean space, thus they are inappropriate to describe the points of the projective space. For the projective plane and space, we need a different algebraic base.

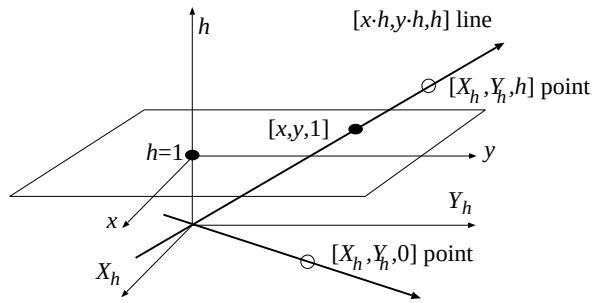


Figure 22.29 The embedded model of the projective plane: the projective plane is embedded into a three-dimensional Euclidean space, and a correspondence is established between points of the projective plane and lines of the embedding three-dimensional Euclidean space by fitting the line to the origin of the three-dimensional space and the given point.

Projective plane. Let us consider first the projective plane and find a method to describe its points by numbers. To start, a Cartesian coordinate system x, y is set up in this plane. Simultaneously, another Cartesian system X_h, Y_h, h is established in the three-dimensional space embedding the plane in a way that axes X_h, Y_h are parallel to axes x, y, the plane is perpendicular to axis h, the origin of the Cartesian system of the plane is in point $(0, 0, 1)$ of the three-dimensional space, and the points of the plane satisfy equation $h = 1$. The projective plane is thus embedded into a three-dimensional Euclidean space where points are defined by Descartes-coordinates (Figure 22.29). To describe a point of the projective plane by numbers, a correspondence is found between the points of the projective plane and the points of the embedding Euclidean space. An appropriate correspondence assigns that line of the Euclidean space to either affine or ideal point P of the projective plane, which is defined by the origin of the coordinate system of the space and point P.

Points of an Euclidean line that crosses the origin can be defined by parametric equation $[t \cdot X_h, t \cdot Y_h, t \cdot h]$ where t is a free real parameter. If point P is an affine point of the projective plane, then the corresponding line is not parallel with plane $h = 1$ (i.e. h is not constant zero). Such line intersects the plane of equation $h = 1$ at point $[X_h/h, Y_h/h, 1]$, thus the Cartesian coordinates of point P in planar coordinate system x, y are $(X_h/h, Y_h/h)$. On the other hand, if point P is ideal, then the corresponding line is parallel to the plane of equation $h = 1$ (i.e. $h = 0$). The direction of the ideal point is given by vector (X_h, Y_h).

The presented approach assigns three dimensional lines crossing the origin and eventually $[X_h, Y_h, h]$ triplets to both the affine and the ideal points of the projective plane. These triplets are called the ***homogeneous coordinates*** of a point in the projective plane. Homogeneous coordinates are enclosed by brackets to distinguish them from Cartesian coordinates.

A three-dimensional line crossing the origin and describing a point of the projective plane can be defined by its arbitrary point except the origin. Consequently, all three homogeneous coordinates cannot be simultaneously zero, and homogeneous coordinates can be freely multiplied by the same non-zero scalar without changing the described point. This property justifies the name "homogeneous".

It is often convenient to select that triplet from the homogeneous coordinates of

an affine point, where the third homogeneous coordinate is 1 since in this case the first two homogeneous coordinates are identical to the Cartesian coordinates:

$$X_h = x, \qquad Y_h = y, \qquad h = 1 \ . \tag{22.15}$$

>From another point of view, Cartesian coordinates of an affine point can be converted to homogeneous coordinates by extending the pair by a third element of value 1.

The embedded model also provides means to define the equations of the lines and line segments of the projective space. Let us select two different points on the projective plane and specify their homogeneous coordinates. The two points are different if homogeneous coordinates $[X_h^1, Y_h^1, h^1]$ of the first point cannot be obtained as a scalar multiple of homogeneous coordinates $[X_h^2, Y_h^2, h^2]$ of the other point. In the embedding space, triplet $[X_h, Y_h, h]$ can be regarded as Cartesian coordinates, thus the ***equation of the line*** fitted to points $[X_h^1, Y_h^1, h^1]$ and $[X_h^2, Y_h^2, h^2]$ is:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \ , \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \ , \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \ . \end{aligned} \tag{22.16}$$

If $h(t) \neq 0$, then the affine points of the projective plane can be obtained by projecting the three-dimensional space onto the plane of equation $h = 1$. Requiring the two points be different, we excluded the case when the line would be projected to a single point. Hence projection maps lines to lines. Thus the presented equation really identifies the homogeneous coordinates defining the points of the line. If $h(t) = 0$, then the equation expresses the ideal point of the line.

If parameter t has an arbitrary real value, then the points of a line are defined. If parameter t is restricted to interval $[0, 1]$, then we obtain the line segment defined by the two endpoints.

Projective space. We could apply the same method to introduce homogeneous coordinates of the projective space as we used to define the homogeneous coordinates of the projective plane, but this approach would require the embedding of the three-dimensional projective space into a four-dimensional Euclidean space, which is not intuitive. We would rather discuss another construction, which works in arbitrary dimensions. In this construction, a point is described as the centre of mass of a mechanical system. To identify a point, let us place weight X_h at reference point $\vec{p}_1$, weight Y_h at reference point $\vec{p}_2$, weight Z_h at reference point $\vec{p}_3$, and weight w at reference point $\vec{p}_4$. The centre of mass of this mechanical system is:

$$\vec{r} = \frac{X_h \cdot \vec{p}_1 + Y_h \cdot \vec{p}_2 + Z_h \cdot \vec{p}_3 + w \cdot \vec{p}_4}{X_h + Y_h + Z_h + w} \ .$$

Let us denote the total weight by $h = X_h + Y_h + Z_h + w$. By definition, elements of quadruple $[X_h, Y_h, Z_h, h]$ are the ***homogeneous coordinates*** of the centre of mass.

To find the correspondence between homogeneous and Cartesian coordinates, the relationship of the two coordinate systems (the relationship of the basis vectors

and the origin of the Cartesian coordinate system and of the reference points of the homogeneous coordinate system) must be established. Let us assume, for example, that the reference points of the homogeneous coordinate system are in points (1,0,0), (0,1,0), (0,0,1), and (0,0,0) of the Cartesian coordinate system. The centre of mass (assuming that total weight h is not zero) is expressed in Cartesian coordinates as follows:

$$\vec{r}[X_h, Y_h, Z_h, h] = \frac{1}{h} \cdot (X_h \cdot (1,0,0) + Y_h \cdot (0,1,0) + Z_h \cdot (0,0,1) + w \cdot (0,0,0)) = \left(\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h} \right).$$

Hence the correspondence between homogeneous coordinates $[X_h, Y_h, Z_h, h]$ and Cartesian coordinates (x, y, z) is ($h \neq 0$):

$$x = \frac{X_h}{h}, \qquad y = \frac{Y_h}{h}, \qquad z = \frac{Z_h}{h} \ . \tag{22.17}$$

The equations of ***lines in the projective space*** can be obtained either deriving them from the embedding four-dimensional Cartesian space, or using the centre of mass analogy:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \ , \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \ , \\ Z_h(t) &= Z_h^1 \cdot (1-t) + Z_h^2 \cdot t \ , \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \ . \end{aligned} \tag{22.18}$$

If parameter t is restricted to interval $[0, 1]$, then we obtain the equation of the ***projective line segment.***

To find the equation of the ***projective plane,*** the equation of the Euclidean plane is considered (equation 22.1). The Cartesian coordinates of the points on an Euclidean plane satisfy the following implicit equation

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \ .$$

Using the correspondence between the Cartesian and homogeneous coordinates (equation 22.17) we still describe the points of the Euclidean plane but now with homogeneous coordinates:

$$n_x \cdot \frac{X_h}{h} + n_y \cdot \frac{Y_h}{h} + n_z \cdot \frac{Z_h}{h} + d = 0 \ .$$

Let us multiply both sides of this equation by h, and add those points to the plane which have $h = 0$ coordinate and satisfy this equation. With this step the set of points of the Euclidean plane is extended with the ideal points, that is, we obtained the set of points belonging to the projective plane. Hence the equation of the projective plane is a homogeneous linear equation:

$$n_x \cdot X_h + n_y \cdot Y_h + n_z \cdot Z_h + d \cdot h = 0 \ , \tag{22.19}$$

or in matrix form:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \ . \tag{22.20}$$

Note that points and planes are described by row and column vectors, respectively. Both the quadruples of points and the quadruples of planes have the homogeneous property, that is, they can be multiplied by non-zero scalars without altering the solutions of the equation.

22.5.2. Homogeneous linear transformations

Transformations defined as the multiplication of the homogeneous coordinate vector of a point by a constant 4×4 $\mathbf{T}$ matrix are called ***homogeneous linear transformations:***

$$[X'_h, Y'_h, Z'_h, h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T} \ . \tag{22.21}$$

Theorem 22.12 *Homogeneous linear transformations map points to points.*

Proof A point can be defined by homogeneous coordinates in form $\lambda \cdot [X_h, Y_h, Z_h, h]$, where λ is an arbitrary, non-zero constant. The transformation results in $\lambda \cdot [X'_h, Y'_h, Z'_h, h'] = \lambda \cdot [X_h, Y_h, Z_h, h] \cdot \mathbf{T}$ when a point is transformed, which are the λ-multiples of the same vector, thus the result is a single point in homogeneous coordinates. ∎

Note that due to the homogeneous property, homogeneous transformation matrix $\mathbf{T}$ is not unambiguous, but can be freely multiplied by non-zero scalars without modifying the realized mapping.

Theorem 22.13 *Invertible homogeneous linear transformations map lines to lines.*

Proof Let us consider the parametric equation of a line:

$$[X_h(t), Y_h(t), Z_h(t), h(t)] = [X_h^1, Y_h^1, Z_h^1, h^1] \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot t, \quad t = (-\infty, \infty) \ ,$$

and transform the points of this line by multiplying the quadruples with the transformation matrix:

$$\begin{aligned} [X'_h(t), Y'_h(t), Z'_h(t), h'(t)] &= [X_h(t), Y_h(t), Z_h(t), h(t)] \cdot \mathbf{T} \\ &= [X_h^1, Y_h^1, Z_h^1, h^1] \cdot \mathbf{T} \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot \mathbf{T} \cdot t \\ &= [X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}] \cdot (1-t) + [X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}] \cdot t \ , \end{aligned}$$

where $[X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}]$ and $[X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}]$ are the transformations of $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$, respectively. Since the transformation is invertible, the two points are different. The resulting equation is the equation of a line fitted to the transformed points. ∎

We note that if we had not required the invertibility of the the transformation, then it could have happened that the transformation would have mapped the two points to the same point, thus the line would have degenerated to single point.

If parameter t is limited to interval $[0,1]$, then we obtain the equation of the projective line segment, thus we can also state that a homogeneous linear transformation maps a line segment to a line segment. Even more generally, a homogeneous linear transformation maps convex combinations to convex combinations. For example, triangles are also mapped to triangles.

However, we have to be careful when we try to apply this theorem in the Euclidean plane or space. Let us consider a line segment as an example. If coordinate h has different sign at the two endpoints, then the line segment contains an ideal point. Such projective line segment can be intuitively imagined as two half lines and an ideal point sticking the "endpoints" of these half lines at infinity, that is, such line segment is the complement of the line segment we are accustomed to. It may happen that before the transformation, coordinates h of the endpoints have similar sign, that is, the line segment meets our intuitive image about Euclidean line segments, but after the transformation, coordinates h of the endpoints will have different sign. Thus the transformation wraps around our line segment.

Theorem 22.14 *Invertible homogeneous linear transformations map planes to planes.*

Proof The originals of transformed points $[X'_h, Y'_h, Z'_h, h']$ defined by $[X_h, Y_h, Z_h, h] = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1}$ are on a plane, thus satisfy the original equation of the plane:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \ .$$

Due to the associativity of matrix multiplication, the transformed points also satisfy equation

$$[X'_h, Y'_h, Z'_h, h'] \cdot \begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = 0 \ ,$$

which is also a plane equation, where

$$\begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} \ .$$

This result can be used to obtain the normal vector of a transformed plane. ■

An important subclass of homogeneous linear transformations is the set of ***affine transformations,*** where the Cartesian coordinates of the transformed point are linear functions of the original Cartesian coordinates:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A} + [p_x, p_y, p_z] \ , \tag{22.22}$$

where vector $\vec{p}$ describes translation, $\mathbf{A}$ is a matrix of size 3×3 and expresses rotation, scaling, mirroring, etc., and their arbitrary combination. For example, the rotation around axis (t_x, t_y, t_z), $(|(t_x, t_y, t_z)| = 1)$ by angle ϕ is given by the following matrix

$$\mathbf{A} = \begin{bmatrix} (1-t_x^2)\cos\phi + t_x^2 & t_x t_y(1-\cos\phi) + t_z \sin\phi & t_x t_z(1-\cos\phi) - t_y \sin\phi \\ t_y t_x(1-\cos\phi) - t_z \sin\phi & (1-t_y^2)\cos\phi + t_y^2 & t_x t_z(1-\cos\phi) + t_x \sin\phi \\ t_z t_x(1-\cos\phi) + t_y \sin\phi & t_z t_y(1-\cos\phi) - t_x \sin\phi & (1-t_z^2)\cos\phi + t_z^2 \end{bmatrix} .$$

This expression is known as the ***Rodrigues-formula.***

Affine transformations map the Euclidean space onto itself, and transform parallel lines to parallel lines. Affine transformations are also homogeneous linear transformations since equation (22.22) can also be given as a 4×4 matrix operation, having changed the Cartesian coordinates to homogeneous coordinates by adding a fourth coordinate of value 1:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix} = [x, y, z, 1] \cdot \mathbf{T} \ . \tag{22.23}$$

A further specialization of affine transformations is the set of ***congruence transformations*** (isometries) which are distance and angle preserving.

Theorem 22.15 *In a congruence transformation the rows of matrix* $\mathbf{A}$ *have unit length and are orthogonal to each other.*

Proof Let us use the property that a congruence is distance and angle preserving for the case when the origin and the basis vectors of the Cartesian system are transformed. The transformation assigns point (p_x, p_y, p_z) to the origin and points $(A_{11} + p_x, A_{12} + p_y, A_{13} + p_z)$, $(A_{21} + p_x, A_{22} + p_y, A_{23} + p_z)$, and $(A_{31} + p_x, A_{32} + p_y, A_{33} + p_z)$ to points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. Because the distance is preserved, the distances between the new points and the new origin are still 1, thus $|(A_{11}, A_{12}, A_{13})| = 1$, $|(A_{21}, A_{22}, A_{23})| = 1$, and $|(A_{31}, A_{32}, A_{33})| = 1$. On the other hand, because the angle is also preserved, vectors (A_{11}, A_{12}, A_{13}), (A_{21}, A_{22}, A_{23}), and (A_{31}, A_{32}, A_{33}) are also perpendicular to each other. ■

Exercises

22.5-1 Using the Cartesian coordinate system as an algebraic basis, prove the axioms of the Euclidean geometry, for example, that two points define a line, and that

two different lines may intersect each other at most at one point.
22.5-2 Using the homogeneous coordinates as an algebraic basis, prove an axiom of the projective geometry stating that two different lines intersect each other in exactly one point.
22.5-3 Prove that homogeneous linear transformations map line segments to line segments using the centre of mass analogy.
22.5-4 How does an affine transformation modify the volume of an object?
22.5-5 Give the matrix of that homogeneous linear transformation which translates by vector $\vec{p}$.
22.5-6 Prove the Rodrigues-formula.
22.5-7 A solid defined by inequality $f(\vec{r}) \geq 0$ in time $t = 0$ moves with uniform constant velocity $\vec{v}$. Let us find the inequality of the solid at an arbitrary time instance t.
22.5-8 Prove that if the rows of matrix $\mathbf{A}$ are of unit length and are perpendicular to each other, then the affine transformation is a congruence. Show that for such matrices $\mathbf{A}^{-1} = \mathbf{A}^T$.
22.5-9 Give that homogeneous linear transformation which projects the space from point $\vec{c}$ onto a plane of normal $\vec{n}$ and place vector $\vec{r}_0$.
22.5-10 Show that five point correspondences unambiguously identify a homogeneous linear transformation if no four points are co-planar.

22.6. Rendering with ray tracing

When a virtual world is rendered, we have to identify the surfaces visible in different directions from the virtual eye. The set of possible directions is defined by a rectangle shaped window which is decomposed to a grid corresponding to the pixels of the screen (Figure 22.30). Since a pixel has a unique colour, it is enough to solve the visibility problem in a single point of each pixel, for example, in the points corresponding to pixel centres.

The surface visible at a direction from the eye can be identified by casting a half line, called ***ray,*** and identifying its intersection closest to the eye position. This operation is called ***ray tracing.*** Ray tracing has many applications. For example, ***shadow*** computation tests whether or not a point is occluded from the light source, which requires a ray to be sent from the point at the direction of the light source and the determination whether this ray intersects any surface closer than the light source. Ray tracing is also used by ***collision detection*** since a point moving with constant and uniform speed collides that surface which is first intersected by the ray describing the motion of the point.

A ray is defined by the following equation:

$$r\vec{a}y(t) = \vec{s} + \vec{v} \cdot t, \qquad (t > 0)\ , \tag{22.24}$$

where $\vec{s}$ is the place vector of the ***ray origin,*** $\vec{v}$ is the ***direction of the ray,*** and ***ray parameter*** t characterizes the distance from the origin. Let us suppose that direction vector $\vec{v}$ has unit length. In this case parameter t is the real distance,

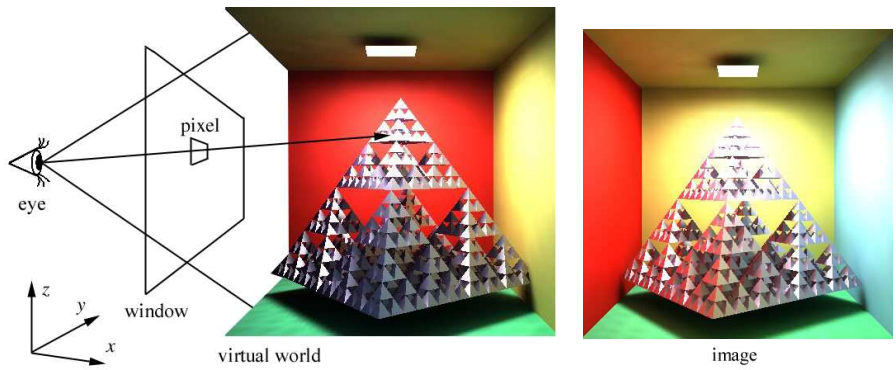


Figure 22.30 Ray tracing.

otherwise it would only be proportional to the distance[3]. If parameter t is negative, then the point is behind the eye and is obviously not visible. The identification of the closest intersection with the ray means the determination of the intersection point having the smallest, positive ray parameter. In order to find the closest intersection, the intersection calculation is tried with each surface, and the closest is retained. This algorithm obtaining the first intersection is:

RAY-FIRST-INTERSECTION$(\vec{s}, \vec{v})$

```
1  t ← t_max                          ▷ Initialization to the maximum size in the virtual world.
2  for each object o
3      do t_o ← RAY-SURFACE-INTERSECTION(s, v)
                                      ▷ Negative if no intersection exists.
4          if 0 ≤ t_o < t             ▷ Is the new intersection closer?
5            then t ← t_o             ▷ Ray parameter of the closest intersection so far.
6                 o_visible ← o       ▷ Closest object so far.
7  if t < t_max then                  ▷ Has been intersection at all?
8    then x ← s + v · t               ▷ Intersection point using the ray equation.
9         return t, x, o_visible
10   else return "no intersection"    ▷ No intersection.
```

This algorithm inputs the ray defined by origin $\vec{s}$ and direction $\vec{v}$, and outputs the ray parameter of the intersection in variable t, the intersection point in $\vec{x}$, and the visible object in $o_{visible}$. The algorithm calls function RAY-SURFACE-INTERSECTION for each object, which determines the intersection of the ray and the given object, and indicates with a negative return value if no intersection exists. Function RAY-SURFACE-INTERSECTION should be implemented separately for each surface type.

[3] In collision detection $\vec{v}$ is not a unit vector, but the velocity of the moving point since this makes ray parameter t express the collision time.

22.6.1. Ray surface intersection calculation

The identification of the intersection between a ray and a surface requires the solution of an equation. The intersection point is both on the ray and on the surface, thus it can be obtained by inserting the ray equation into the equation of the surface and solving the resulting equation for the unknown ray parameter.

Intersection calculation for implicit surfaces. For implicit surfaces of equation $f(\vec{r}) = 0$, the intersection can be calculated by solving the following scalar equation for t: $f(\vec{s} + \vec{v} \cdot t) = 0$.

Let us take the example of ***quadrics*** that include the sphere, the ellipsoid, the cylinder, the cone, the paraboloid, etc. The implicit equation of a general quadric contains a quadratic form:

$$[x, y, z, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \ ,$$

where $\mathbf{Q}$ is a 4×4 matrix. Substituting the ray equation into the equation of the surface, we obtain

$$[s_x + v_x \cdot t, s_y + v_y \cdot t, s_z + v_z \cdot t, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} s_x + v_x \cdot t \\ s_y + v_y \cdot t \\ s_z + v_z \cdot t \\ 1 \end{bmatrix} = 0 \ .$$

Rearranging the terms, we get a second order equation for unknown parameter t:

$$t^2 \cdot (\mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{v}^T) + t \cdot (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{v}^T + \mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{s}^T) + (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{s}^T) = 0 \ ,$$

where $\mathbf{v} = [v_x, v_y, v_z, 0]$ and $\mathbf{s} = [s_x, s_y, s_z, 1]$.

This equation can be solved using the solution formula of second order equations. Now we are interested in only the real and positive roots. If two such roots exist, then the smaller one corresponds to the intersection closer to the origin of the ray.

Intersection calculation for parametric surfaces. The intersection of parametric surface $\vec{r} = \vec{r}(u, v)$ and the ray is calculated by first solving the following equation for unknown parameters u, v, t

$$\vec{r}(u, v) = \vec{s} + t \cdot \vec{v} \ ,$$

then checking whether or not t is positive and parameters u, v are inside the allowed parameter range of the surface.

Roots of non-linear equations are usually found by numeric methods. On the other hand, the surface can also be approximated by a triangle mesh, which is intersected by the ray. Having obtained the intersection on the coarse mesh, the mesh around this point is refined, and the intersection calculation is repeated with the refined mesh.

Intersection calculation for a triangle. To compute the ray intersection for a ***triangle*** of vertices $\vec{a}$, $\vec{b}$, and $\vec{c}$, first the ray intersection with the plane of the triangle is found. Then it is decided whether or not the intersection point with the plane is inside the triangle. The normal and a place vector of the triangle plane are $\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$, and $\vec{a}$, respectively, thus points $\vec{r}$ of the plane satisfy the following equation:

$$\vec{n} \cdot (\vec{r} - \vec{a}) = 0 \; . \tag{22.25}$$

The intersection of the ray and this plane is obtained by substituting the ray equation (equation (22.24)) into this plane equation, and solving it for unknown parameter t. If root t^* is positive, then it is inserted into the ray equation to get the intersection point with the plane. However, if the root is negative, then the intersection is behind the origin of the ray, thus is invalid. Having a valid intersection with the plane of the triangle, we check whether this point is inside the triangle. This is a containment problem, which is discussed in Subsection 22.4.1.

Intersection calculation for an AABB. The surface of an AABB, that is an axis aligned block, can be subdivided to 6 rectangular faces, or alternatively to 12 triangles, thus its intersection can be solved by the algorithms discussed in the previous subsections. However, realizing that in this special case the three coordinates can be handled separately, we can develop more efficient approaches. In fact, an AABB is the intersection of an x-stratum defined by inequality $x_{min} \leq x \leq x_{max}$, a y-stratum defined by $y_{min} \leq y \leq y_{max}$ and a z-stratum of inequality $z_{min} \leq z \leq z_{max}$. For example, the ray parameters of the intersections with the x-stratum are:

$$t_x^1 = \frac{x_{min} - s_x}{v_x}, \qquad t_x^2 = \frac{x_{max} - s_x}{v_x} \; .$$

The smaller of the two parameter values corresponds to the entry at the stratum, while the greater to the exit. Let us denote the ray parameter of the entry by t_{in}, and the ray parameter of the exit by t_{out}. The ray is inside the x-stratum while the ray parameter is in $[t_{in}, t_{out}]$. Repeating the same calculation for the y and z-strata as well, three ray parameter intervals are obtained. The intersection of these intervals determine when the ray is inside the AABB. If parameter t_{out} obtained as the result of intersecting the strata is negative, then the AABB is behind the eye, thus no ray–AABB intersection is possible. If only t_{in} is negative, then the ray starts at an internal point of the AABB, and the first intersection is at t_{out}. Finally, if t_{in} is positive, then the ray enters the AABB from outside at parameter t_{in}.

The computation of the unnecessary intersection points can be reduced by applying the Cohen – Sutherland line clipping algorithm (subsection 22.4.3). First, the ray is replaced by a line segment where one endpoint is the origin of the ray, and the other endpoint is an arbitrary point on the ray which is farther from the origin than any object of the virtual world. Then this line segment is tried to be clipped by the AABB. If the Cohen – Sutherland algorithm reports that the line segment has no internal part, then the ray has no intersection with the AABB.

22.6.2. Speeding up the intersection calculation

A naive ray tracing algorithm tests each object for a ray to find the closest intersection. If there are N objects in the space, the running time of the algorithm is $\Theta(N)$ both in the average and in the worst case. The storage requirement is also linear in terms of the number of objects.

The method would be speeded up if we could exclude certain objects from the intersection test without testing them one by one. The reasons of such exclusion include that these objects are "behind" the ray or "not in the direction of the ray". Additionally, the speed is also expected to improve if we can terminate the search having found an intersection supposing that even if other intersections exist, they are surely farther than the just found intersection point. To make such decisions safely, we need to know the arrangement of objects in the virtual world. This information is gathered during the pre-processing phase. Of course, pre-processing has its own computational cost, which is worth spending if we have to trace a lot of rays.

Bounding volumes. One of the simplest ray tracing acceleration technique uses ***bounding volumes,*** The bounding volume is a shape of simple geometry, typically a sphere or an AABB, which completely contains a complex object. When a ray is traced, first the bounding volume is tried to be intersected. If there is no intersection with the bounding volume, then neither can the contained object be intersected, thus the computation time of the ray intersection with the complex object is saved. The bounding volume should be selected in a way that the ray intersection is computationally cheap, and it is a tight container of the complex object.

The application of bounding volumes does not alter the linear time complexity of the naive ray tracing. However, it can increase the speed by a scalar factor.

On the other hand, bounding volumes can also be organized in a hierarchy putting bounding volumes inside bigger bounding volumes recursively. In this case the ray tracing algorithm traverses this hierarchy, which is possible in sub-linear time.

Space subdivision with uniform grids. Let us find the AABB of the complete virtual world and subdivide it by an axis aligned uniform grid of cell sizes (c_x, c_y, c_z) (Figure 22.31).

In the preprocessing phase, for each cell we identify those objects that are at least partially contained by the cell. The test of an object against a cell can be performed using a clipping algorithm (subsection 22.4.3), or simply checking whether the cell and the AABB of the object overlap.

UNIFORM-GRID-CONSTRUCTION()

```
1  Compute the minimum corner of the AABB (x_min, y_min, z_min)
   and cell sizes (c_x, c_y, c_z)
2  for each cell c
3      do object list of cell c ← empty
4          for each object o                    ▷ Register objects overlapping
                                                ▷ with this cell.
```

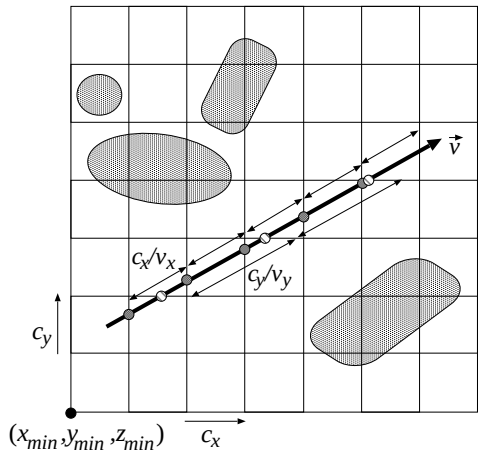


Figure 22.31 Partitioning the virtual world by a uniform grid. The intersections of the ray and the coordinate planes of the grid are at regular distances c_x/v_x,c_y/v_y, and c_z/v_z, respectively.

```
5        do if cell c and the AABB of object o overlap
6           then add object o to object list of cell c
```

During ray tracing, cells intersected by the ray are visited in the order of their distance from the ray origin. When a cell is processed, only those objects need to be tested for intersection which overlap with this cell, that is, which are registered in this cell. On the other hand, if an intersection is found in the cell, then intersections belonging to other cells cannot be closer to the ray origin than the found intersection. Thus the cell marching can be terminated. Note that when an object registered in a cell is intersected by the ray, we should also check whether the intersection point is also in this cell.

We might meet an object again in other cells. The number of ray–surface intersection can be reduced if the results of ray–surface intersections are stored with the objects and are reused when needed again.

As long as no ray–surface intersection is found, the algorithm traverses those cells which are intersected by the ray. Indices X, Y, Z of the first cell are computed from ray origin $\vec{s}$, minimum corner $(x_{min}, y_{min}, z_{min})$ of the grid, and sizes (c_x, c_y, c_z) of the cells:

```
UNIFORM-GRID-ENCLOSING-CELL(s⃗)
1  X ← INTEGER((s_x − x_min)/c_x)
2  Y ← INTEGER((s_y − y_min)/c_y)
3  Z ← INTEGER((s_z − z_min)/c_z)
4  return X, Y, Z
```

The presented algorithm assumes that the origin of the ray is inside the subspace covered by the grid. Should this condition not be met, then the intersection of the

ray and the scene AABB is computed, and the ray origin is moved to this point.

The initial values of ray parameters t_x, t_y, t_z are computed as the intersection of the ray and the coordinate planes by the UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION algorithm:

UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION$(\vec{s}, \vec{v}, X, Y, Z)$

```
1  if v_x > 0
2     then t_x ← (x_min + (X + 1) · c_x − s_x)/v_x
3     else if v_x < 0
4             then t_x ← (x_min + X · c_x − s_x)/v_x
5             else  t_x ← t_max                          ▷ The maximum distance.
6  if v_y > 0
7     then   t_y ← (y_min + (Y + 1) · c_y − s_y)/v_y
8     else if v_y < 0
9             then t_y ← (y_min + Y · c_y − s_y)/v_y
10            else  t_y ← t_max
11 if v_z > 0
12    then   t_z ← (z_min + (Z + 1) · c_z − s_z)/v_z
13    else if v_z < 0
14            then t_z ← (z_min + Z · c_z − s_z)/v_z
15            else  t_z ← t_max
16 return t_x, t_y, t_z
```

The next cell of the sequence of the visited cells is determined by the ***3D line drawing algorithm (3DDDA algorithm).*** This algorithm exploits the fact that the ray parameters of the intersection points with planes perpendicular to axis x (and similarly to axes y and z) are regularly placed at distance c_x/v_x (c_y/v_y, and c_z/v_z, respectively), thus the ray parameter of the next intersection can be obtained with a single addition (Figure 22.31). Ray parameters t_x, t_y, and t_z are stored in global variables, and are incremented by constant values. The smallest from the three ray parameters of the coordinate planes identifies the next intersection with the cell.

The following algorithm computes indices X, Y, Z of the next intersected cell, and updates ray parameters t_x, t_y, t_z:

UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)

```
1  if t_x = min(t_x, t_y, t_z) ▷ Next intersection is on the plane perpendicular to axis x.
2     then X ← X + sgn(v_x)                    ▷ Function sgn(x) returns the sign.
3          t_x ← t_x + c_x/|v_x|
4  else if t_y = min(t_x, t_y, t_z)
                  ▷ Next intersection is on the plane perpendicular to axis y.
5          then Y ← Y + sgn(v_y)
6               t_y ← t_y + c_y/|v_y|
```

```
7  else if t_z = min(t_x, t_y, t_z)
                          ▷ Next intersection is on the plane perpendicular to axis z.
8        then Z ← Z + sgn(v_z)
9             t_z ← t_z + c_z/|v_z|
```

To summarize, a complete ray tracing algorithm is presented, which exploits the uniform grid generated during preprocessing and computes the ray-surface intersection closest to the ray origin. The minimum of ray parameters (t_x, t_y, t_z) assigned to the coordinate planes, i.e. variable t_{out}, determines the distance as far as the ray is inside the cell. This parameter is used to decide whether or not a ray-surface intersection is really inside the cell.

RAY-FIRST-INTERSECTION-WITH-UNIFORM-GRID$(\vec{s}, \vec{v})$

```
1  (X, Y, Z) ← UNIFORM-GRID-ENCLOSING-CELL(s⃗)
2  (t_x, t_y, t_z) ← UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION(s⃗, v⃗, X, Y, Z)
3  while X, Y, Z are inside the grid
4        do t_out ← min(t_x, t_y, t_z)              ▷ Here is the exit from the cell.
5           t ← t_out                               ▷ Initialization: no intersection yet.
6           for each object o registered in cell (X, Y, Z)
7               do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗, o)
                                                    ▷ Negative: no intersection.
8                  if 0 ≤ t_o < t                   ▷ Is the new intersection closer?
9                  then t ← t_o
                                ▷ The ray parameter of the closest intersection so far.
10                      o_visible ← o               ▷ The first intersected object.
11             if t < t_out                         ▷ Was intersection in the cell?
12               then x⃗ ← s⃗ + v⃗ · t                ▷ The position of the intersection.
13                    return t, x⃗, o_visible        ▷ Termination.
14          UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)        ▷ 3DDDA.
15 return "no intersection"
```

Time and storage complexity of the uniform grid algorithm. The preprocessing phase of the uniform grid algorithm tests each object with each cell, thus runs in $\Theta(N \cdot C)$ time where N and C are the numbers of objects and cells, respectively. In practice, the resolution of the grid is set to make C proportional to N since in this case, the average number of objects per cell becomes independent of the total number of objects. Such resolution makes the preprocessing time quadratic, that is $\Theta(N^2)$. We note that sorting objects before testing them against cells may reduce this complexity, but this optimization is not crucial since not the preprocessing but the ray tracing time is critical. Since in the worst case all objects may overlap with each cell, the storage space is also in $O(N^2)$.

The ray tracing time can be expressed by the following equation:

$$T = T_o + N_I \cdot T_I + N_S \cdot T_S \ , \tag{22.26}$$

where T_o is the time needed to identify the cell containing the origin of the ray, N_I is the number of ray–surface intersection tests until the first intersection is found, T_I is the time required by a single ray–surface intersection test, N_S is the number of visited cells, and T_S is the time needed to step onto the next cell.

To find the first cell, the coordinates of the ray origin should be divided by the cell sizes, and the cell indices are obtained by rounding the results. This step thus runs in constant time. A single ray–surface intersection test also requires constant time. The next cell is determined by the 3DDDA algorithm in constant time as well. Thus the complexity of the algorithm depends only on the number of intersection tests and the number of the visited cells.

Considering a worst case scenario, a cell may contain all objects, requiring $O(N)$ intersection test with N objects. In the worst case the ray tracing has linear complexity. This means that the uniform grid algorithm needs quadratic preprocessing time and storage, but solves the ray tracing problem still in linear time as the naive algorithm, which is quite disappointing. However, uniform grids are still worth using since worst case scenarios are very unlikely. The fact is that classic complexity measures describing the worst case characteristics are not appropriate to compare the naive algorithm and the uniform grid based ray tracing. For a reasonable comparison, the probabilistic analysis of the algorithms is needed.

Probabilistic model of the virtual world. To carry out the average case analysis, the scene model, i.e. the probability distribution of the possible virtual world models must be known. In practical situations, this probability distribution is not available, therefore it must be estimated. If the model of the virtual world were too complicated, we would not be able to analytically determine the average, i.e. the expected running time of the ray tracing algorithm. A simple, but also justifiable model is the following: *Objects are spheres of the same radius r, and sphere centres are uniformly distributed in space.*

Since we are interested in the asymptotic behavior when the number of objects is really high, uniform distribution in a finite space would not be feasible. On the other hand, the boundary of the space would pose problems. Thus, instead of dealing with a finite object space, the space should also be expanded as the number of objects grows to sustain constant average spatial object density. This is a classical method in probability theory, and its known result is the Poisson point process.

Definition 22.16 *A **Poisson point process** $N(A)$ counts the number of points in subset A of space in a way that*

- *$N(A)$ is a Poisson distribution of parameter $\rho V(A)$, where ρ is a positive constant called "intensity" and $V(A)$ is the volume of A, thus the probability that A contains exactly k points is*

$$\Pr\{N(A) = k\} = \frac{(\rho V(A))^k}{k!} \cdot e^{-\rho V(A)} \ ,$$

and the expected number of points in volume $V(A)$ is $\rho V(A)$;

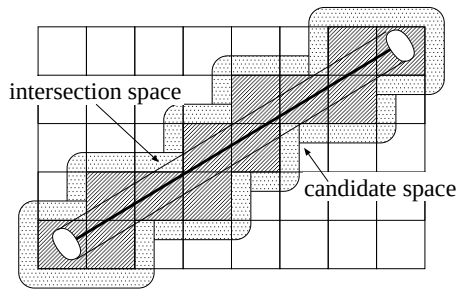


Figure 22.32 Encapsulation of the intersection space by the cells of the data structure in a uniform subdivision scheme. The intersection space is a cylinder of radius r. The candidate space is the union of those spheres that may overlap a cell intersected by the ray.

- *for disjoint $A_1, A_2, \ldots, A_n$ sets random variables $N(A_1), N(A_2), \ldots, N(A_n)$ are independent.*

Using the Poisson point process, the probabilistic model of the virtual world is:

1. The object space consists of spheres of the same radius r.
2. The sphere centres are the realizations of a Poisson point process of intensity ρ.

Having constructed a probabilistic virtual world model, we can start the analysis of the candidate algorithms assuming that the rays are uniformly distributed in space.

Calculation of the expected number of intersections. Looking at Figure 22.32 we can see a ray that passes through certain cells of the space partitioning data structure. The collection of those sphere centres where the sphere would have an intersection with a cell is called the ***candidate space*** associated with this cell.

Only those spheres of radius r can have intersection with the ray whose centres are in a cylinder of radius r around the ray. This cylinder is called the ***intersection space*** (Figure 22.32). More precisely, the intersection space also includes two half spheres at the bottom and at the top of the cylinder, but these will be ignored.

As the ray tracing algorithm traverses the data structure, it examines each cell that is intersected by the ray. If the cell is empty, then the algorithm does nothing. If the cell is not empty, then it contains, at least partially, a sphere which is tried to be intersected. This intersection succeeds if the centre of the sphere is inside the intersection space and fails if it is outside.

The algorithm should try to intersect objects that are in the candidate space, but this intersection will be successful only if the object is also contained by the intersection space. The probability of the success s is the ratio of the projected areas of the intersection space and the candidate space associated with this cell.

>From the probability of the successful intersection in a non-empty cell, the probability that the intersection is found in the first, second, etc. cells can also be

computed. Assuming statistical independence, the probabilities that the first, second, third, etc. intersection is the first successful intersection are s, $(1-s)s$, $(1-s)^2 s$, etc., respectively. This is a geometric distribution with expected value $1/s$. Consequently, the expected number of the ray–object intersection tests is:

$$E[N_I] = \frac{1}{s} \, . \tag{22.27}$$

If the ray is parallel to one of the sides, then the projected size of the candidate space is $c^2+4cr+r^2\pi$ where c is the edge size of a cell and r is the radius of the spheres. The other extreme case happens when the ray is parallel to the diagonal of the cubic cell, where the projection is a rounded hexagon having area $\sqrt{3}c^2 + 6cr + r^2\pi$. The success probability is then:

$$\frac{r^2\pi}{\sqrt{3}c^2 + 6cr + r^2\pi} \leq s \leq \frac{r^2\pi}{c^2 + 4cr + r^2\pi} \, .$$

According to equation (22.27), the average number of intersection calculations is the reciprocal of this probability:

$$\frac{1}{\pi}\left(\frac{c}{r}\right)^2 + \frac{4}{\pi}\frac{c}{r} + 1 \leq \mathrm{E}\left[N_I\right] \leq \frac{\sqrt{3}}{\pi}\left(\frac{c}{r}\right)^2 + \frac{6}{\pi}\frac{c}{r} + 1 \, . \tag{22.28}$$

Note that if the size of the cell is equal to the diameter of the sphere ($c = 2r$), then

$$3.54 < \mathrm{E}\left[N_I\right] < 7.03 \, .$$

This result has been obtained assuming that the number of objects converges to infinity. The expected number of intersection tests, however, remains finite and relatively small.

Calculation of the expected number of cell steps. In the following analysis the conditional expected value theorem will be used. An appropriate condition is the length of the ray segment between its origin and the closest intersection. Using its probability density $p_{t^*}(t)$ as a condition, the expected number of visited cells N_S can be written in the following form:

$$E[N_S] = \int_0^\infty E[N_S | t^* = t] \cdot p_{t^*}(t) \; dt \, ,$$

where t^* is the length of the ray and p_{t^*} is its probability density.

Since the intersection space is a cylinder if we ignore the half spheres around the beginning and the end, its total volume is $r^2\pi t$. Thus the probability that intersection occurs before t is:

$$\Pr\left\{t^* < t\right\} = 1 - e^{-\rho r^2 \pi t} \, .$$

Note that this function is the cumulative probability distribution function of t^*. The probability density can be computed as its derivative, thus we obtain:

$$p_{t^*}(t) = \rho r^2 \pi \cdot e^{-\rho r^2 \pi t} \, .$$

The expected length of the ray is then:

$$E[t^*] = \int_0^\infty t \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t} \, dt = \frac{1}{\rho r^2 \pi} \, . \tag{22.29}$$

In order to simplify the analysis, we shall assume that the ray is parallel to one of the coordinate axes. Since all cells have the same edge size c, the number of cells intersected by a ray of length t can be estimated as $E[N_S | t^* = t] \approx t/c + 1$. This estimation is quite accurate. If the the ray is parallel to one of the coordinate axes, then the error is at most 1. In other cases the real value can be at most $\sqrt{3}$ times the given estimation. The estimated expected number of visited cells is then:

$$\mathrm{E}\left[N_S\right] \approx \int_0^\infty \left(\frac{t}{c} + 1\right) \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t} \, dt = \frac{1}{c \rho r^2 \pi} + 1 \, . \tag{22.30}$$

For example, if the cell size is similar to the object size ($c = 2r$), and the expected number of sphere centres in a cell is 0.1, then $\mathrm{E}\left[N_S\right] \approx 14$. Note that the expected number of visited cells is also constant even for infinite number of objects.

Expected running time and storage space. We concluded that the expected numbers of required intersection tests and visited cells are asymptotically constant, thus the expected time complexity of the uniform grid based ray tracing algorithm is constant after quadratic preprocessing time. The value of the running time can be controlled by cell size c according to equations (22.28) and (22.30). Smaller cell sizes reduce the average number of intersection tests, but increase the number of visited cells.

According to the probabilistic model, the average number of objects overlapping with a cell is also constant, thus the storage is proportional to the number of cells. Since the number of cells is set proportional to the number of objects, the expected storage complexity is also linear unlike the quadratic worst-case complexity.

The expected constant running time means that asymptotically the running time is independent of the number of objects, which explains the popularity of the uniform grid based ray tracing algorithm, and also the popularity of the algorithms presented in the next subsections.

Octree. Uniform grids require many unnecessary cell steps. For example, the empty spaces are not worth partitioning into cells, and two cells are worth separating only if they contain different objects. Adaptive space partitioning schemes are based on these recognitions. The space can be partitioned adaptively following a recursive approach. This results in a hierarchical data structure, which is usually a tree. The type of this tree is the base of the classification of such algorithms.

The adaptive scheme discussed in this subsection uses an octal tree (octree for short), where non-empty nodes have 8 children. An octree is constructed by the following algorithm:

- For each object, an AABB is found, and object AABBs are enclosed by a scene AABB. The scene AABB is the cell corresponding to the root of the octree.

- If the number of objects overlapping with the current cell exceeds a predefined threshold, then the cell is subdivided to 8 cells of the same size by halving the original cell along each coordinate axis. The 8 new cells are the children of the node corresponding to the original cell. The algorithm is recursively repeated for the child cells.
- The recursive tree building procedure terminates if the depth of the tree becomes too big, or when the number of objects overlapping with a cell is smaller than the threshold.

The result of this construction is an ***octree*** (Figure 22.33). Overlapping objects are registered in the leaves of this tree.

When a ray is traced, those leaves of the tree should be traversed which are intersected by the ray, and ray–surface intersection test should be executed for objects registered in these leaves:

RAY-FIRST-INTERSECTION-WITH-OCTREE($\vec{s}, \vec{v}$)

```
1  q⃗ ← intersection of the ray and the scene AABB
2  while q⃗ is inside of the scene AABB                    ▷ Traversal of the tree.
3        cell ← OCTREE-CELL-SEARCH(octree root, q⃗)
4        t_out ← ray parameter of the intersection of the cell and the ray
5        t ← t_out                    ▷ Initialization: no ray–surface intersection yet.
6        for each object o registered in cell
7            do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗)
                                                  ▷ Negative if no intersection exists.
8               if 0 ≤ t_o < t                    ▷ Is the new intersection closer?
9                  then t ← t_o ▷ Ray parameter of the closest intersection so far.
10                      o_visible ← o             ▷ First intersected object so far.
11       if t < t_out                             ▷ Has been intersection at all ?
12          then x⃗ ← s⃗ + v⃗ · t                    ▷ Position of the intersection.
13               return t, x⃗, o_visible
14       q⃗ ← s⃗ + v⃗ · (t_out + ε)                  ▷ A point in the next cell.
15 return "no intersection"
```

The identification of the next cell intersected by the ray is more complicated for octrees than for uniform grids. The OCTREE-CELL-SEARCH algorithm determines that leaf cell which contains a given point. At each level of the tree, the coordinates of the point are compared to the coordinates of the centre of the cell. The results of these comparisons determine which child contains the point. Repeating this test recursively, we arrive at a leaf sooner or later.

In order to identify the next cell intersected by the ray, the intersection point of the ray and the current cell is computed. Then, ray parameter t_{out} of this intersection point is increased "a little" (this little value is denoted by ε in algorithm RAY-FIRST-INTERSECTION-WITH-OCTREE). The increased ray parameter is substituted into the ray equation, resulting in point $\vec{q}$ that is already in the next cell. The cell containing this point can be identified with OCTREE-CELL-SEARCH.

Cells of the octree may be larger than the allowed minimal cell, therefore the

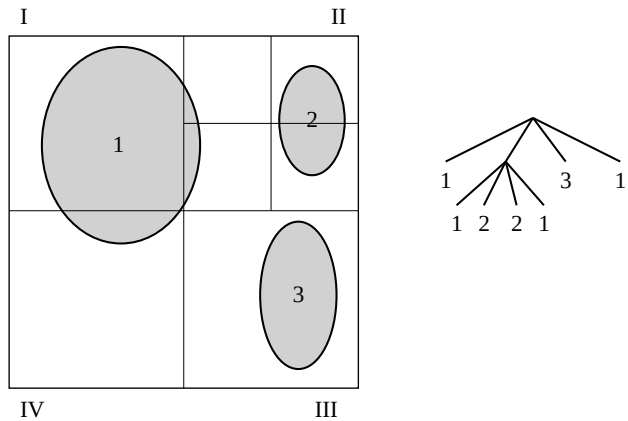


Figure 22.33 A quadtree partitioning the plane, whose three-dimensional version is the octree. The tree is constructed by halving the cells along all coordinate axes until a cell contains "just a few" objects, or the cell sizes gets smaller than a threshold. Objects are registered in the leaves of the tree.

octree algorithm requires less number of cell steps than the uniform grid algorithm working on the minimal cells. However, larger cells reduce the probability of the successful intersection tests since in a large cell it is less likely that a random ray intersecting the cell also intersects a contained object. Smaller successful intersection probability, on the other hand, results in greater expected number of intersection tests, which affects the performance negatively. It also means that non-empty octree cells are worth subdividing until the minimum cell size is reached even if the cell contains just a single object. Following this strategy, the size of the non-empty cells are similar, thus the results of the complexity analysis made for the uniform grid remain to be applicable to the octree as well. Since the probability of the successful intersection depends on the size of the non-empty cells, the expected number of needed intersection tests is still given by inequality (22.28). It also means that when the minimal cell size of an octree equals to the cell size of a uniform grid, then the expected number of intersection tests is equal in the two algorithms.

The advantage of the ocree is the ability to skip empty spaces, which reduces the number of cell steps. Its disadvantage is, however, that the time of the next cell identification is not constant. This identification requires the traversal of the tree. If the tree construction is terminated when a cell contains small number of objects, then the number of leaf cells is proportional to the number of objects. The depth of the tree is in $O(\lg N)$, so is the time needed to step onto the next cell.

kd-tree. An octree adapts to the distribution of the objects. However, the partitioning strategy of octrees always halves the cells without taking into account where the objects are, thus the adaptation is not perfect. Let us consider a partitioning scheme which splits a cell into two cells to make the tree balanced. Such method builds a binary tree which is called ***binary space partitioning tree,*** abbreviated as ***BSP-tree.***. If the ***separating plane*** is always perpendicular to one of the coor-

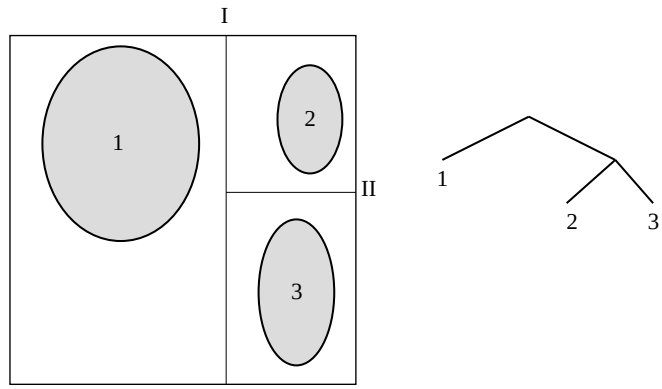


Figure 22.34 A kd-tree. A cell containing "many" objects are recursively subdivided to two cells with a plane that is perpendicular to one of the coordinate axes.

dinate axes, then the tree is called ***kd-tree***.

The separating plane of a kd-tree node can be placed in many different ways:

- the ***spatial median method*** halves the cell into two congruent cells.
- the ***object median method*** finds the separating plane to have the same number of objects in the two child cells.
- the ***cost driven method*** estimates the average computation time needed when a cell is processed during ray tracing, and minimizes this value by placing the separating plane. An appropriate cost model suggests to separate the cell to make the probabilities of the ray–surface intersection of the two cells similar.

The probability of the ray–surface intersection can be computed using a fundamental theorem of the ***integral geometry:***

Theorem 22.17 *If convex solid A contains another convex solid B, then the probability that a uniformly distributed line intersects solid B provided that the line intersected A equals to the ratio of the surface areas of objects B and A.*

According to this theorem the cost driven method finds the separating plane to equalize the surface areas in the two children.

Let us now present a general kd-tree construction algorithm. Parameter *cell* identifies the current cell, *depth* is the current depth of recursion, and *coordinate* stores the orientation of the current separating plane. A *cell* is associated with its two children (*cell.right* and *cell.left*), and its left-lower-closer and right-upper-farther corners (*cell.min* and *cell.max*). Cells also store the list of those objects which overlap with the cell. The orientation of the separation plane is determined by a round-robin scheme implemented by function ROUND-ROBIN providing a sequence like $(x, y, z, x, y, z, x, \ldots)$. When the following recursive algorithm is called first, it gets the scene AABB in variable *cell* and the value of variable *depth* is zero:

```
KD-TREE-CONSTRUCTION(cell, depth, coordinate)
 1  if the number of objects overlapping with cell is small or depth is large
 2    then return
 3  AABB of cell.left and AABB of cell.right ← AABB of cell
 4  if coordinate = x
 5    then cell.right.min.x ← x perpendicular separating plane of cell
 6         cell.left.max.x ← x perpendicular separating plane of cell
 7    else if coordinate = y
 8           then cell.right.min.y ← y perpendicular separating plane of cell
 9                cell.left.max.y ← y perpendicular separating plane of cell
10    else if coordinate = z
11           then cell.right.min.z ← z perpendicular separating plane of cell
12                cell.left.max.z ← z perpendicular separating plane of cell
13  for each object o of cell
14      do if object o is in the AABB of cell.left
15           then assign object o to the list of cell.left
16         if object o is in the AABB of cell.right
17           then assign object o to the list of cell.right
18  KD-TREE-CONSTRUCTION(cell.left, depth + 1, ROUND-ROBIN(coordinate))
19  KD-TREE-CONSTRUCTION(cell.right, depth + 1, ROUND-ROBIN(coordinate))
```

Now we discuss an algorithm that traverses the constructed kd-tree and finds the visible object. First we have to test whether the origin of the ray is inside the scene AABB. If it is not, the intersection of the ray and the scene AABB is computed, and the origin of the ray is moved there. The identification of the cell containing the ray origin requires the traversal of the tree. During the traversal the coordinates of the point are compared to the coordinates of the separating plane. This comparison determines which child should be processed recursively until a leaf node is reached. If the leaf cell is not empty, then objects overlapping with the cell are intersected with the ray, and the intersection closest to the origin is retained. The closest intersection is tested to see whether or not it is inside the cell (since an object may overlap more than one cells, it can also happen that the intersection is in another cell). If the intersection is in the current cell, then the needed intersection has been found, and the algorithm can be terminated. If the cell is empty, or no intersection is found in the cell, then the algorithm should proceed with the next cell. To identify the next cell, the ray is intersected with the current cell identifying the ray parameter of the exit point. Then the ray parameter is increased "a little" to make sure that the increased ray parameter corresponds to a point in the next cell. The algorithm keeps repeating these steps as it processes the cells of the tree.

This method has the disadvantage that the cell search always starts at the root, which results in the repetitive traversals of the same nodes of the tree.

This disadvantage can be eliminated by putting the cells to be visited into a stack, and backtracking only to the point where a new branch should be followed. When the ray arrives at a node having two children, the algorithm decides the order of processing the two child nodes. Child nodes are classified as "near" and "far"

depending on whether or not the child cell is on the same side of the separating plane as the origin of the ray. If the ray intersects only the "near" child, then the algorithm processes only that subtree which originates at this child. If the ray intersects both children, then the algorithm pushes the "far" node onto the stack and starts processing the "near" node. If no intersection exists in the "near" node, then the stack is popped to obtain the next node to be processed.

The notations of the ray tracing algorithm based on kd-tree traversal are shown by Figure 22.35. The algorithm is the following:

RAY-FIRST-INTERSECTION-WITH-KD-TREE($root, \vec{s}, \vec{v}$)

```
1  (t_in, t_out) ← RAY-AABB-INTERSECTION(s, v, root)
                                          ▷ Intersection with the scene AABB.
2  if no intersection
3     then return "no intersection"
4  PUSH(root, t_in, t_out)
5  while the stack is not empty                                  ▷ Visit all nodes.
6        do POP(cell, t_in, t_out)
7           while cell is not a leaf
8                 do coordinate ← orientation of the separating plane of the cell
9                    d ← cell.right.min[coordinate] − s[coordinate]
10                   t ← d/v[coordinate] ▷ Ray parameter of the separating plane.
11                   if d > 0      ▷ Is s on the left side of the separating plane?
12                      then (near, far) ← (cell.left, cell.right)          ▷ Left.
13                      else (near, far) ← (cell.right, cell.left)          ▷ Right.
14                   if t > t_out or t < 0
15                      then cell ← near ▷ The ray intersects only the near cell.
16                      else if t < t_in
17                              then cell ← far ▷ The ray intersects only the far cell.
18                              else PUSH(far, t, t_out) ▷ The ray intersects both cells.
19                                   cell ← near       ▷ First near is intersected.
20                                   t_out ← t ▷ The ray exists at t from the near cell.
                                                        ▷ If the current cell is a leaf.
21          t ← t_out                          ▷ Maximum ray parameter in this cell.
22          for each object o of cell
23             do t_o ←RAY-SURFACE-INTERSECTION(s, v)
                                          ▷ Negative if no intersection exists.
24                if t_in ≤ t_o < t ▷ Is the new intersection closer to the ray origin?
25                then t ← t_o
                        ▷ The ray parameter of the closest intersection so far.
26                     o_visible ← o
                              ▷ The object intersected closest to the ray origin.
```

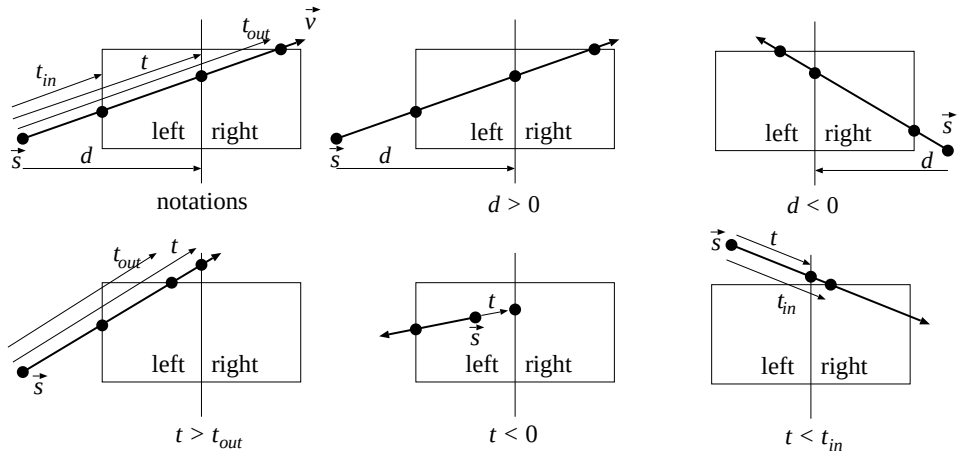


Figure 22.35 Notations and cases of algorithm RAY-FIRST-INTERSECTION-WITH-KD-TREE. t_{in}, t_{out}, and t are the ray parameters of the entry, exit, and the separating plane, respectively. d is the signed distance between the ray origin and the separating plane.

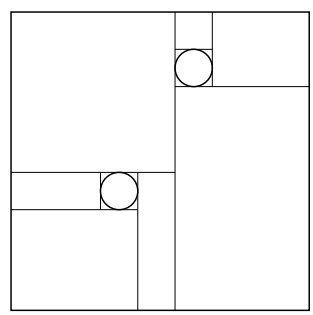


Figure 22.36 Kd-tree based space partitioning with empty space cutting.

```
27        if t < t_out                    ▷ Has been intersection at all in the cell?
28          then x⃗ ← s⃗ + v⃗ · t             ▷ The intersection point.
29               return t, x⃗, o_visible     ▷ Intersection has been found.
30  return "no intersection"              ▷ No intersection.
```

Similarly to the octree algorithm, the likelihood of successful intersections can be increased by continuing the tree building process until all empty spaces are cut (Figure 22.36).

Our probabilistic world model contains spheres of same radius r, thus the non-empty cells are cubes of edge size $c = 2r$. Unlike in uniform grids or octrees, the separating planes of kd-trees are not independent of the objects. Kd-tree splitting planes are rather tangents of the objects. This means that we do not have to be concerned with partially overlapping spheres since a sphere is completely contained

by a cell in a kd-tree. The probability of the successful intersection is obtained applying Theorem 22.17. In the current case, the containing convex solid is a cube of edge size $2r$, the contained solid is a sphere of radius r, thus the intersection probability is:

$$s = \frac{4r^2\pi}{6a^2} = \frac{\pi}{6} .$$

The expected number of intersection tests is then:

$$\mathrm{E}[N_I] = \frac{6}{\pi} \approx 1.91 .$$

We can conclude that the kd-tree algorithm requires the smallest number of ray-surface intersection tests according to the probabilistic model.

Exercises

22.6-1 Prove that the expected number of intersection tests is constant in all those ray tracing algorithms which process objects in the order of their distance from the ray origin.

22.6-2 Propose a ray intersection algorithm for subdivision surfaces.

22.6-3 Develop a ray intersection method for B-spline surfaces.

22.6-4 Develop a ray intersection algorithm for CSG models assuming that the ray–primitive intersection tests are already available.

22.6-5 Propose a ray intersection algorithm for transformed objects assuming that the algorithm computing the intersection with the non-transformed objects is available (hints: transform the ray).

22.7. Incremental rendering

Rendering requires the identification of those surface points that are visible through the pixels of the virtual camera. Ray tracing solves this visibility problem for each pixel independently, thus it does not reuse visibility information gathered at other pixels. The algorithms of this section, however, exploit such information using the following simple techniques:

1. They simultaneously attack the visibility problem for all pixels, and handle larger parts of the scene at once.
2. Where feasible, they exploit the ***incremental principle*** which is based on the recognition that the visibility problem becomes simpler to solve if the solution at the neighbouring pixel is taken into account.
3. They solve each task in that coordinate system which makes the solution easier. The scene is transformed from one coordinate system to the other by homogeneous linear transformations.
4. They minimize unnecessary computations, therefore remove those objects by *clipping* in an early stage of rendering which cannot be projected onto the window of the camera.

Homogeneous linear transformations and clipping may change the type of the surface except for points, line segments and polygons [4]. Therefore, before rendering is started, each shape is approximated by points, line segments, and meshes (Subsection 22.3).

Steps of incremental rendering are shown in Figure 22.37. Objects are defined in their reference state, approximated by meshes, and are transformed to the virtual world. The time dependence of this transformation is responsible for object animation. The image is taken from the camera about the virtual world, which requires the identification of those surface points that are visible from the camera, and their projection onto the window plane. The visibility and projection problems could be solved in the virtual world as happens in ray tracing, but this would require the intersection calculations of general lines and polygons. Visibility and projection algorithms can be simplified if the scene is transformed to a coordinate system, where the X, Y coordinates of a point equal to the coordinates of that pixel onto which this point is projected, and the Z coordinate can be used to decide which point is closer if more than one surfaces are projected onto the same pixel. Such coordinate system is called the ***screen coordinate system.*** In screen coordinates the units of axes X and Y are equal to the pixel size. Since it is usually not worth computing the image on higher accuracy than the pixel size, coordinates X, Y are integers. Because of performance reasons, coordinate Z is also often integer. Screen coordinates are denoted by capital letters.

The transformation taking to the screen coordinate system is defined by a sequence of transformations, and the elements of this sequence are discussed separately. However, this transformation is executed as a single multiplication with a 4×4 transformation matrix obtained as the product of elementary transformation matrices.

22.7.1. Camera transformation

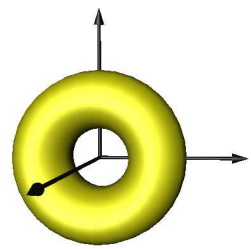
Rendering is expected to generate an image from a camera defined by ***eye position*** ($\vec{eye}$) (the focal point of the camera), looking target ($\vec{lookat}$) where the camera looks at, and by vertical direction $\vec{up}$ (Figure 22.38).

Camera parameter *fov* defines the vertical field of view, *aspect* is the ratio of the width and the height of the window, f_p and b_p are the distances of the front and back clipping planes from the eye, respectively. These clipping planes allow to remove those objects that are behind, too close to, or too far from the eye.

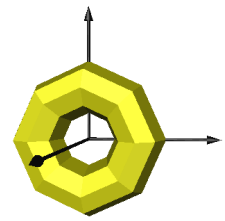
We assign a coordinate system, i.e. three orthogonal unit basis vectors to the camera. Horizontal basis vector $\vec{u} = (u_x, u_y, u_z)$, vertical basis vector $\vec{v} = (v_x, v_y, v_z)$, and basis vector $\vec{w} = (w_x, w_y, w_z)$ pointing to the looking direction are obtained as follows:

$$\vec{w} = \frac{\vec{eye} - \vec{lookat}}{|\vec{eye} - \vec{lookat}|}, \qquad \vec{u} = \frac{\vec{up} \times \vec{w}}{|\vec{up} \times \vec{w}|}, \qquad \vec{v} = \vec{w} \times \vec{u}\ .$$

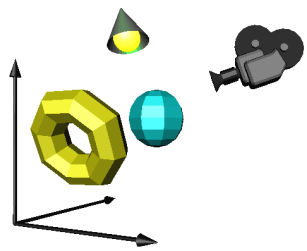
[4] Although Bézier and B-Spline curves and surfaces are invariant to affine transformations, and NURBS is invariant even to homogeneous linear transformations, but clipping changes these object types as well.



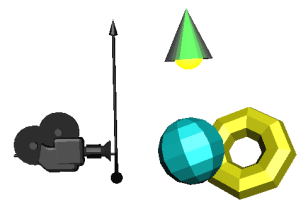
(a) Modelling



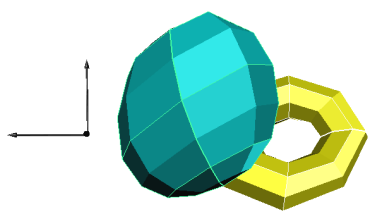
(b) Tessellation



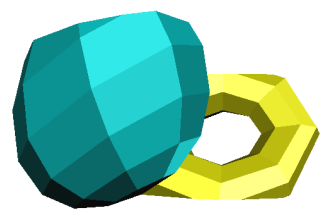
(c) Modelling transformation



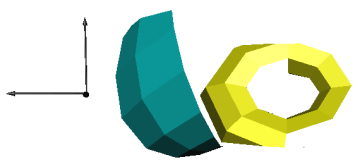
(d) Camera transformation



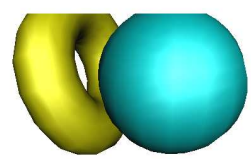
(e) Perspective transformation



(f) Clipping



(g) Hidden surface elimination



(h) Projection and shading

Figure 22.37 Steps of incremental rendering. **(a)** Modelling defines objects in their reference state. **(b)** Shapes are tessellated to prepare for further processing. **(c)** Modelling transformation places the object in the world coordinate system. **(d)** Camera transformation translates and rotates the scene to get the eye to be at the origin and to look parallel with axis $-z$. **(e)** Perspective transformation converts projection lines meeting at the origin to parallel lines, that is, it maps the eye position onto an ideal point. **(f)** Clipping removes those shapes and shape parts, which cannot be projected onto the window. **(g)** Hidden surface elimination removes those surface parts that are occluded by other shapes. **(h)** Finally, the visible polygons are projected and their projections are filled with their visible colours.

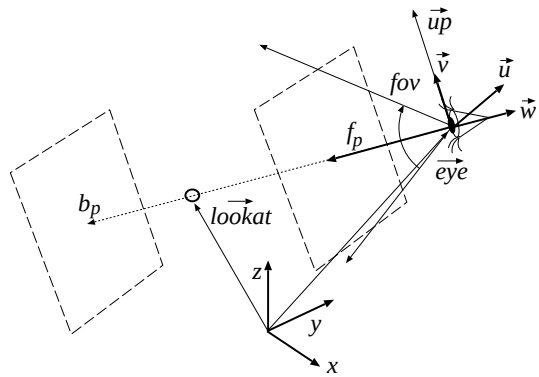


Figure 22.38 Parameters of the virtual camera: eye position $\vec{eye}$, target $\vec{lookat}$, and vertical direction $\vec{up}$, from which camera basis vectors $\vec{u}, \vec{v}, \vec{w}$ are obtained, front f_p and back b_p clipping planes, and vertical field of view *fov* (the horizontal field of view is computed from aspect ratio *aspect*).

The ***camera transformation*** translates and rotates the space of the virtual world in order to get the camera to move to the origin, to look at direction axis $-z$, and to have vertical direction parallel to axis y, that is, this transformation maps unit vectors $\vec{u}, \vec{v}, \vec{w}$ to the basis vectors of the coordinate system. Transformation matrix $\mathbf{T}_{camera}$ can be expressed as the product of a matrix translating the eye to the origin and a matrix rotating basis vectors $\vec{u}, \vec{v}, \vec{w}$ of the camera to the basis vectors of the coordinate system:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \mathbf{T}_{camera} = [x, y, z, 1] \cdot \mathbf{T}_{translation} \cdot \mathbf{T}_{rotation} , \qquad (22.31)$$

where

$$\mathbf{T}_{translation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_x & -eye_y & -eye_z & 1 \end{bmatrix}, \quad \mathbf{T}_{rotation} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Let us note that the columns of the rotation matrix are vectors $\vec{u}, \vec{v}, \vec{w}$. Since these vectors are orthogonal, it is easy to see that this rotation maps them to coordinate axes x, y, z. For example, the rotation of vector $\vec{u}$ is:

$$[u_x, u_y, u_z, 1] \cdot \mathbf{T}_{rotation} = [\vec{u} \cdot \vec{u}, \vec{u} \cdot \vec{v}, \vec{u} \cdot \vec{w}, 1] = [1, 0, 0, 1] .$$

22.7.2. Normalizing transformation

In the next step the viewing pyramid containing those points which can be projected onto the window is normalized making the field of view equal to 90 degrees (Figure 22.39).

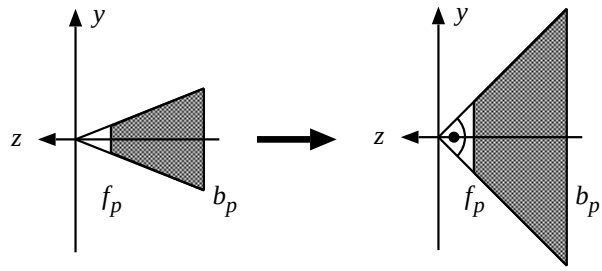


Figure 22.39 The normalizing transformation sets the field of view to 90 degrees.

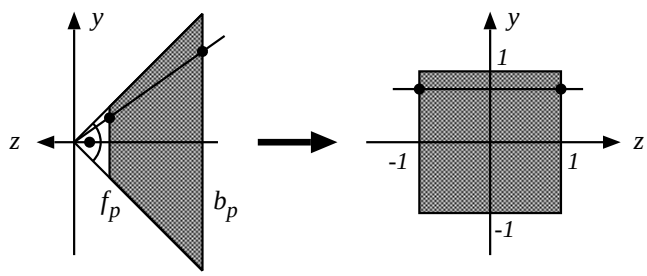


Figure 22.40 The perspective transformation maps the finite frustum of pyramid defined by the front and back clipping planes, and the edges of the window onto an axis aligned, origin centred cube of edge size 2.

Normalization is a simple scaling transformation:

$$\mathbf{T}_{norm} = \begin{bmatrix} 1/(\tan(fov/2) \cdot aspect) & 0 & 0 & 0 \\ 0 & 1/\tan(fov/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The main reason of this transformation is to simplify the formulae of the next transformation step, called perspective transformation.

22.7.3. Perspective transformation

The perspective transformation distorts the virtual world to allow the replacement of the perspective projection by parallel projection during rendering.

After the normalizing transformation, the potentially visible points are inside a symmetrical finite frustum of pyramid of 90 degree apex angle (Figure 22.39). The perspective transformation maps this frustum onto a cube, converting projection lines crossing the origin to lines parallel to axis z (Figure 22.40).

Perspective transformation is expected to map point to point, line to line, but to map the eye position to infinity. It means that perspective transformation cannot be a linear transformation of Cartesian coordinates. Fortunately, homogeneous linear

transforms also map point to point, line to line, and are able to handle points at infinity with finite coordinates. Let us thus try to find the perspective transformation in the form of a homogeneous linear transformation defined by a 4×4 matrix:

$$\mathbf{T}_{persp} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix} .$$

Figure 22.40 shows a line (projection ray) and its transform. Let m_x and m_y be the x/z and the y/z slopes of the line, respectively. This line is defined by equation $[-m_x \cdot z, -m_y \cdot z, z]$ in the normalized camera space. The perspective transformation maps this line to a "horizontal" line crossing point $[m_x, m_y, 0]$ and being parallel to axis z. Let us examine the intersection points of this line with the front and back clipping planes, that is, let us substitute $(-f_p)$ and $(-b_p)$ into parameter z of the line equation. The transformation should map these points to $[m_x, m_y, -1]$ and $[m_x, m_y, 1]$, respectively.

The perspective transformation of the point on the first clipping plane is:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x, m_y, -1, 1] \cdot \lambda ,$$

where λ is an arbitrary, non-zero scalar since the point defined by homogeneous coordinates does not change if the homogeneous coordinates are simultaneously multiplied by a non-zero scalar. Setting λ to f_p, we get:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x \cdot f_p, m_y \cdot f_p, -f_p, f_p] \ . \tag{22.32}$$

Note that the first coordinate of the transformed point equals to the first coordinate of the original point on the clipping plane for arbitrary m_x, m_y, and f_p values. This is possible only if the first column of matrix $\mathbf{T}_{persp}$ is $[1, 0, 0, 0]^T$. Using the same argument for the second coordinate, we can conclude that the second column of the matrix is $[0, 1, 0, 0]^T$. Furthermore, in equation (22.32) the third and the fourth homogeneous coordinates of the transformed point are not affected by the first and the second coordinates of the original point, requiring $t_{13} = t_{14} = t_{23} = t_{24} = 0$. The conditions on the third and the fourth homogeneous coordinates can be formalized by the following equations:

$$-f_p \cdot t_{33} + t_{43} = -f_p, \qquad -f_p \cdot t_{34} + t_{44} = f_p \ .$$

Applying the same procedure for the intersection point of the projection line and the back clipping plane, we can obtain other two equations:

$$-b_p \cdot t_{33} + t_{43} = b_p, \qquad -b_p \cdot t_{34} + t_{44} = b_p \ .$$

Solving this system of linear equations, the matrix of the perspective transformation can be expressed as:

$$\mathbf{T}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(f_p + b_p)/(b_p - f_p) & -1 \\ 0 & 0 & -2 \cdot f_p \cdot b_p/(b_p - f_p) & 0 \end{bmatrix} .$$

Since perspective transformation is not affine, the fourth homogeneous coordinate of the transformed point is usually not 1. If we wish to express the coordinates of the transformed point in Cartesian coordinates, the first three homogeneous coordinates should be divided by the fourth coordinate. Homogeneous linear transforms map line segment to line segment and triangle to triangle, but it may happen that the resulting line segment or triangle contains ideal points (Subsection 22.5.2). The intuition behind the homogeneous division is a traveling from the projective space to the Euclidean space, which converts a line segment containing an ideal point to two half lines. If just the two endpoints of the line segment is transformed, then it is not unambiguous whether the two transformed points need to be connected by a line segment or the complement of this line segment should be considered as the result of the transformation. This ambiguity is called the ***wrap around problem***.

The wrap around problem does not occur if we can somehow make sure that the original shape does not contain points that might be mapped onto ideal points. Examining the matrix of the perspective transformation we can conclude that the fourth homogeneous coordinate of the transformed point will be equal to the $-z$ coordinate of the original point. Ideal points having zero fourth homogeneous coordinate ($h = 0$) may thus be obtained transforming the points of plane $z = 0$, i.e. the plane crossing the origin and parallel to the window. However, if the shapes are clipped onto a first clipping plane being in front of the eye, then these points are removed. Thus the solution of the wrap around problem is the execution of the clipping step before the homogeneous division.

22.7.4. Clipping in homogeneous coordinates

The purpose of ***clipping*** is to remove all shapes that either cannot be projected onto the window or are not between the front and back clipping planes. To solve the ***wrap around problem***, clipping should be executed before the homogeneous division. The clipping boundaries in homogeneous coordinates can be obtained by transforming the screen coordinate AABB back to homogeneous coordinates. In screen coordinates, i.e. after homogeneous division, the points to be preserved by clipping meet the following inequalities:

$$-1 \le X = X_h/h \le 1, \qquad -1 \le Y = Y_h/h \le 1, \qquad -1 \le Z = Z_h/h \le 1\ . \tag{22.33}$$

On the other hand, points that are in front of the eye after camera transformation have negative z coordinates, and the perspective transformation makes the fourth homogeneous coordinate h equal to $-z$ in normalized camera space. Thus the fourth homogeneous coordinate of points in front of the eye is always positive. Let us thus add condition $h > 0$ to the set of conditions of inequalities (22.33). If h is positive, then inequalities (22.33) can be multiplied by h, resulting in the definition of the clipping region in homogeneous coordinates:

$$-h \le X_h \le h, \qquad -h \le Y_h \le h, \qquad -h \le Z_h \le h\ . \tag{22.34}$$

Points can be clipped easily, since we should only test whether or not the conditions of inequalities (22.34) are met. Clipping line segments and polygons, on the

other hand, requires the computation of the intersection points with the faces of the clipping boundary, and only those parts should be preserved which meet inequalities (22.34).

Clipping algorithms using Cartesian coordinates were discussed in Subsection 22.4.3. Those methods can also be applied in homogeneous coordinates with two exceptions. Firstly, for homogeneous coordinates, inequalities (22.34) define whether a point is in or out. Secondly, intersections should be computed using the homogeneous coordinate equations of the line segments and the planes.

Let us consider a line segment with endpoints $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$. This line segment can be an independent shape or an edge of a polygon. Here we discuss the clipping on half space of equation $X_h \leq h$ (clipping methods on other half spaces are very similar). Three cases need to be distinquished:

1. If both endpoints of the line segment are inside, that is $X_h^1 \leq h^1$ and $X_h^2 \leq h^2$, then the complete line segment is in, thus is preserved.

2. If both endpoints are outside, that is $X_h^1 > h^1$ and $X_h^2 > h^2$, then all points of the line segment are out, thus it is completely eliminated by clipping.

3. If one endpoint is outside, while the other is in, then the intersection of the line segment and the clipping plane should be obtained. Then the endpoint being out is replaced by the intersection point. Since the points of a line segment satisfy equation (22.19), while the points of the clipping plane satisfy equation $X_h = h$, parameter t_i of the intersection point is computed as:

$$X_h(t_i) = h(t_i) \quad \Longrightarrow \quad X_h^1 \cdot (1 - t_i) + X_h^2 \cdot t_i = h^1 \cdot (1 - t_i) + h^2 \cdot t_i \quad \Longrightarrow$$

$$\Longrightarrow t_i = \frac{X_h^1 - h^1}{X_h^1 - X_h^2 + h^2 - h^1} \cdot$$

Substituting parameter t_i into the equation of the line segment, homogeneous coordinates $[X_h^i, Y_h^i, Z_h^i, h^i]$ of the intersection point are obtained.

Clipping may introduce new vertices. When the vertices have some additional features, for example, the surface colour or normal vector at these vertices, then these additional features should be calculated for the new vertices as well. We can use linear interpolation. If the values of a feature at the two endpoints are I^1 and I^2, then the feature value at new vertex $[X_h(t_i), Y_h(t_i), Z_h(t_i), h(t_i)]$ generated by clipping is $I^1 \cdot (1 - t_i) + I^2 \cdot t_i$.

22.7.5. Viewport transformation

Having executed the perspective transformation, the Cartesian coordinates of the visible points are in $[-1, 1]$. These normalized device coordinates should be further scaled and translated according to the resolution of the screen and the position of the viewport where the image is expected. Denoting the left-bottom corner pixel of the screen viewport by (X_{min}, Y_{min}), the right-top corner by (X_{max}, Y_{max}), and Z coordinates expressing the distance from the eye are expected in (Z_{min}, Z_{max}), the

matrix of the viewport transformation is:

$$\mathbf{T}_{viewport} = \begin{bmatrix} (X_{max} - X_{min})/2 & 0 & 0 & 0 \\ 0 & (Y_{max} - Y_{min})/2 & 0 & 0 \\ 0 & 0 & (Z_{max} - Z_{min})/2 & 0 \\ (X_{max} + X_{min})/2 & (Y_{max} + Y_{min})/2 & (Z_{max} + Z_{min})/2 & 1 \end{bmatrix} .$$

Coordinate systems after the perspective transformation are ***left handed,*** unlike the coordinate systems of the virtual world and the camera, which are ***right handed.*** Left handed coordinate systems seem to be unusual, but they meet our natural expectation that the screen coordinate X grows from left to right, the Y coordinate from bottom to top and, the Z coordinate grows in the direction of the camera target.

22.7.6. Rasterization algorithms

After clipping, homogeneous division, and viewport transformation, shapes are in the screen coordinate system where a point of coordinates (X, Y, Z) can be assigned to a pixel by extracting the first two Cartesian coordinates (X, Y).

Rasterization works in the screen coordinate system and identifies those pixels which have to be coloured to approximate the projected shape. Since even simple shapes can cover many pixels, rasterization algorithms should be very fast, and should be appropriate for hardware implementation.

Line drawing. Let the endpoints of a line segment be (X_1, Y_1) and (X_2, Y_2) in screen coordinates. Let us further assume that while we are going from the first endpoint towards the second, both coordinates are growing, and X is the faster changing coordinate, that is,

$$\Delta X = X_2 - X_1 \geq \Delta Y = Y_2 - Y_1 \geq 0 .$$

In this case the line segment is moderately ascending. We discuss only this case, other cases can be handled by exchanging the X, Y coordinates and replacing additions by substractions.

Line drawing algorithms are expected to find pixels that approximate a line in a way that there are no holes and the approximation is not fatter than necessary. In case of moderately ascending line segments this means that in each pixel column exactly one pixel should be filled with the colour of the line. This coloured pixel is the one closest to the line in this column. Using the following equation of the line

$$y = m \cdot X + b, \qquad \text{where} \quad m = \frac{Y_2 - Y_1}{X_2 - X_1}, \quad \text{and} \quad b = Y_1 - X_1 \cdot \frac{Y_2 - Y_1}{X_2 - X_1} , \tag{22.35}$$

in pixel column of coordinate X, the pixel closest to the line has Y coordinate that is equal to the rounding of $m \cdot x + b$. Unfortunately, the determination of Y requires a floating point multiplication, addition, and a rounding operation, which are too slow.

In order to speed up line drawing, we apply a fundamental trick of computer

graphics, the ***incremental principle.*** The incremental principle is based on the recognition that it is usually simpler to evaluate a function $y(X+1)$ using value $y(X)$ than computing it from X. Since during line drawing the columns are visited one by one, when column $(X+1)$ is processed, value $y(X)$ is already available. In case of a line segment we can write:

$$y(X+1) = m \cdot (X+1) + b = m \cdot X + b + m = y(X) + m \ .$$

Note that the evaluation of this formula requires just a single floating point addition (m is less than 1). This fact is exploited in ***digital differential analyzator algorithms*** (DDA-algorithms). The ***DDA line drawing algorithm*** is then:

DDA-LINE-DRAWING$(X_1, Y_1, X_2, Y_2, \mathit{colour})$

```
1  m ← (Y2 − Y1)/(X2 − X1)
2  y ← Y1
3  for X ← X1 to X2
4      do Y ← ROUND(y)
5         PIXEL-WRITE(X, Y, colour)
6         y ← y + m
```

Further speedups can be obtained using ***fixed point number representation.*** This means that the product of the number and 2^T is stored in an integer variable, where T is the number of fractional bits. The number of fractional bits should be set to exclude cases when the rounding errors accumulate to an incorrect result during long iteration sequences. If the longest line segment covers L columns, then the minimum number of fractional bits guaranteeing that the accumulated error is less than 1 is $\log_2 L$. Thanks to clipping only lines fitting to the screen are rasterized, thus L is equal to the maximum screen resolution.

The performance and simplicity of the DDA line drawing algorithm can still be improved. On the one hand, the software implementation of the DDA algorithm requires shift operations to realize truncation and rounding operations. On the other hand – once for every line segment – the computation of slope m involves a division which is computationally expensive. Both problems are solved in the ***Bresenham line drawing algorithm.***

Let us denote the vertical, signed distance of the line segment and the closest pixel centre by s, and the vertical distance of the line segment and the pixel centre just above the closest pixel by t (Figure 22.41). As the algorithm steps onto the next pixel column, values s and t change and should be recomputed. While the new s and t values satisfy inequality $s < t$, that is, while the lower pixel is still closer to the line segment, the shaded pixel of the next column is in the same row as in the previous column. Introducing error variable $e = s - t$, the row of the shaded pixel remains the same until this error variable is negative ($e < 0$). As the pixel column is incremented, variables s, t, e are updated using the incremental formulae ($\Delta X = X_2 - X_1$, $\Delta Y = Y_2 - Y_1$):

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X}, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} \quad \Longrightarrow \quad e(X+1) = e(X) + 2\frac{\Delta Y}{\Delta X} \ .$$

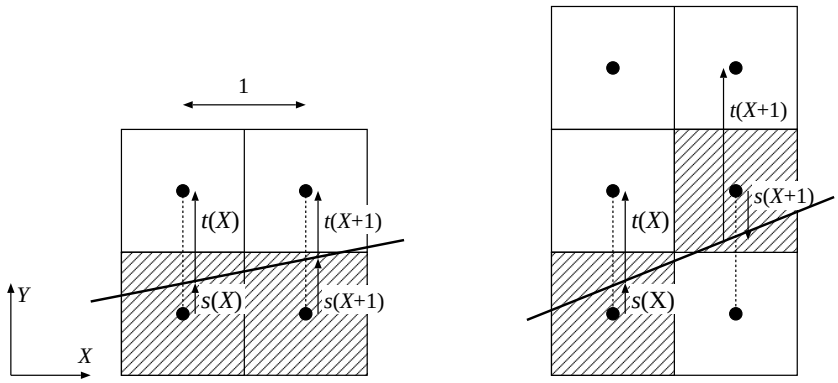


Figure 22.41 Notations of the Bresenham algorithm: s is the signed distance between the closest pixel centre and the line segment along axis Y, which is positive if the line segment is above the pixel centre. t is the distance along axis Y between the pixel centre just above the closest pixel and the line segment.

These formulae are valid if the closest pixel in column $(X+1)$ is in the same row as in column X. If stepping to the next column, the upper pixel gets closer to the line segment (error variable e becomes positive), then variables s, t, e should be recomputed for the new closest row and for the pixel just above it. The formulae describing this case are as follows:

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X} - 1, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} + 1 \Longrightarrow$$

$$\Longrightarrow e(X+1) = e(X) + 2\left(\frac{\Delta Y}{\Delta X} - 1\right) .$$

Note that s is a signed distance which is negative if the line segment is below the closest pixel centre, and positive otherwise. We can assume that the line starts at a pixel centre, thus the initial values of the control variables are:

$$s(X_1) = 0, \qquad t(X_1) = 1 \quad \Longrightarrow \quad e(X_1) = s(X_1) - t(X_1) = -1 .$$

This algorithm keeps updating error variable e and steps onto the next pixel row when the error variable becomes positive. In this case, the error variable is decreased to have a negative value again. The update of the error variable requires a non-integer addition and the computation of its increment involves a division, similarly to the DDA algorithm. It seems that this approach is not better than the DDA.

Let us note, however, that the sign changes of the error variable can also be recognized if we examine the product of the error variable and a positive number. Multiplying the error variable by ΔX we obtain ***decision variable*** $E = e \cdot \Delta X$. In case of moderately ascending lines the decision and error variables change their sign simultaneously. The incremental update formulae of the decision variable can

be obtained by multiplying the update formulae of error variable by ΔX:

$$E(X+1) = \begin{cases} E(X) + 2\Delta Y, & \text{if } Y \text{ is not incremented} \\ E(X) + 2(\Delta Y - \Delta X), & \text{if } Y \text{ needs to be incremented} \end{cases} .$$

The initial value of the decision variable is $E(X_1) = e(X_1) \cdot \Delta X = -\Delta X$.

The decision variable starts at an integer value and is incremented by integers in each step, thus it remains to be an integer and does not require fractional numbers at all. The computation of the increments need only integer additions or subtractions and multiplications by 2.

The complete ***Bresenham line drawing algorithm*** is:

BRESENHAM-LINE-DRAWING$(X_1, Y_1, X_2, Y_2, \mathit{colour})$

```
1  ΔX ← X2 − X1
2  ΔY ← Y2 − Y1
3  (dE+, dE−) ← (2(ΔY − ΔX), 2ΔY)
4  E ← −ΔX
5  Y ← Y1
6  for X ← X1 to X2
7      do if E ≤ 0
8           then E ← E + dE−          ▷ The line stays in the current pixel row.
9           else E ← E + dE+          ▷ The line steps onto the next pixel row.
10               Y ← Y + 1
11          PIXEL-WRITE(X, Y, colour)
```

The fundamental idea of the Bresenham algorithm was the replacement of the fractional error variable by an integer decision variable in a way that the conditions used by the algorithm remained equivalent. This approach is also called the ***method of invariants,*** which is useful in many rasterization algorithms.

Polygon fill. The input of an algorithm filling single connected polygons is the array of vertices $\vec{q}[0], \ldots, \vec{q}[m-1]$ (this array is usually the output of the polygon clipping algorithm). Edge e of the polygon connects vertices $\vec{q}[e]$ and $\vec{q}[e+1]$. The last vertex needs not be treated in a special way if the first vertex is put again after the last vertex in the array. Multiply connected polygons are defined by more than one closed polylines, thus are specified by more than one vertex arrays.

The filling is executed by processing a horizontal pixel row called ***scan line*** at a time. For a single scan line, the pixels belonging to the interior of the polygon can be found by the following steps. First the intersections of the polygon edges and the scan line are calculated. Then the intersection points are sorted in the ascending order of their X coordinates. Finally, pixels between the first and the second intersection points, and between the third and the fourth intersection points, or generally between the $(2i+1)$th and the $(2i+2)$th intersection points are set to the colour of the polygon (Figure 22.42). This algorithm fills those pixels which can be reached from infinity by crossing the polygon boundary odd number of times.

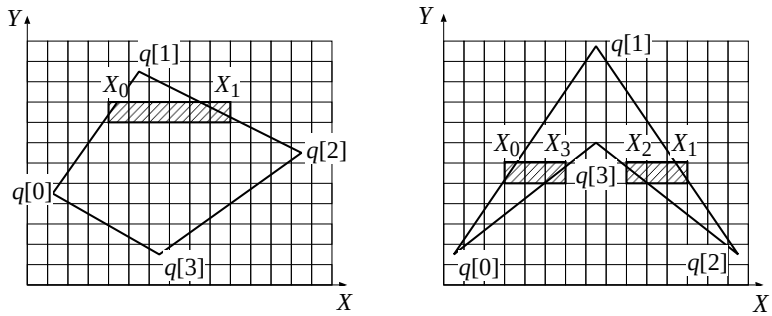


Figure 22.42 Polygon fill. Pixels inside the polygon are identified scan line by scan line.

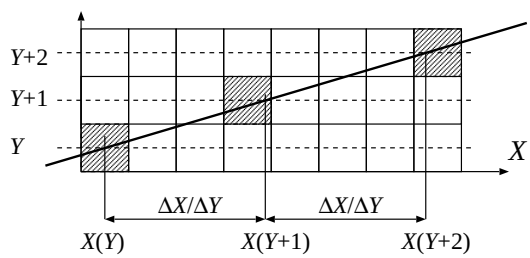


Figure 22.43 Incremental computation of the intersections between the scan lines and the edges. Coordinate X always increases with the reciprocal of the slope of the line.

The computation of the intersections between scan lines and polygon edges can be speeded up using the following observations:

1. An edge and a scan line can have intersection only if coordinate Y of the scan line is between the minimum and maximum Y coordinates of the edge. Such edges are the ***active edges.*** When implementing this idea, an ***active edge table*** (***AET*** for short) is needed which stores the currently active edges.

2. The computation of the intersection point of a line segment and the scan line requires floating point multiplication, division, and addition, thus it is time consuming. Applying the ***incremental principle,*** however, we can also obtain the intersection point of the edge and a scan line from the intersection point with the previous scan line using a single, fixed-point addition (Figure 22.43).

When the incremental principle is exploited, we realize that coordinate X of the intersection with an edge always increases by the same amount when scan line Y is incremented. If the edge endpoint having the larger Y coordinate is (X_{max}, Y_{max}) and the endpoint having the smaller Y coordinate is (X_{min}, Y_{min}), then the increment of the X coordinate of the intersection is $\Delta X/\Delta Y$, where $\Delta X = X_{max} - X_{min}$ and $\Delta Y = Y_{max} - Y_{min}$. This increment is usually not an integer, hence increment $\Delta X/\Delta Y$ and intersection coordinate X should be stored in non-integer, preferably

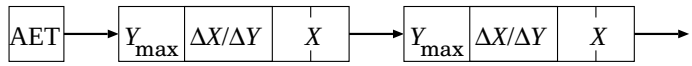


Figure 22.44 The structure of the active edge table.

fixed-point variables. An active edge is thus represented by a fixed-point increment $\Delta X/\Delta Y$, the fixed-point coordinate value of intersection X, and the maximum vertical coordinate of the edge (Y_{max}). The maximum vertical coordinate is needed to recognize when the edge becomes inactive.

Scan lines are processed one after the other. First, the algorithm determines which edges become active for this scan line, that is, which edges have minimum Y coordinate being equal to the scan line coordinate. These edges are inserted into the active edge table. The active edge table is also traversed and those edges whose maximum Y coordinate equals to the scan line coordinate are removed (note that this way the lower end of an edge is supposed to belong to the edge, but the upper edge is not). Then the active edge table is sorted according to the X coordinates of the edges, and the pixels between each pair of edges are filled. Finally, the X coordinates of the intersections in the edges of the active edge table are prepared for the next scan line by incrementing them by the reciprocal of the slope $\Delta X/\Delta Y$.

```
POLYGON-FILL(polygon, colour)
 1  for Y ← 0 to Y_max
 2      do for each edge of polygon              ▷ Put activated edges into the AET.
 3             do if edge.ymin = Y
 4                   then PUT-AET(edge)
 5         for each edge of the AET   ▷ Remove deactivated edges from the AET.
 6             do if edge.ymax ≤ Y
 7                   then DELETE-FROM-AET(edge)
 8         SORT-AET                                  ▷ Sort according to X.
 9         for each pair of edges (edge1, edge2) of the AET
10             do for X ← ROUND(edge1.x) to ROUND(edge2.x)
11                    do PIXEL-WRITE(X, Y, colour)
11         for each edge in the AET                  ▷ Incremental principle.
12             do edge.x ← edge.x + edge.ΔX/ΔY
```

The algorithm works scan line by scan line and first puts the activated edges ($edge.ymin = Y$) to the active edge table. The active edge table is maintained by three operations. Operation PUT-AET($edge$) computes variables ($Y_{max}, \Delta X/\Delta Y, X$) of an edge and inserts this structure into the table. Operation DELETE-FROM-AET removes an item from the table when the edge is not active any more ($edge.ymax \leq Y$). Operation SORT-AET sorts the table in the ascending order of the X value of the items. Having sorted the lists, every two consecutive items form a pair, and the pixels between the endpoints of each of these pairs are filled. Finally, the X coordinates of the items are updated according to the incremental principle.

22.7.7. Incremental visibility algorithms

The three-dimensional ***visibility problem*** is solved in the screen coordinate system. We can assume that the surfaces are given as triangle meshes.

Z-buffer algorithm. The ***z-buffer algorithm*** finds that surface for each pixel, where the Z coordinate of the visible point is minimal. For each pixel we allocate a memory to store the minimum Z coordinate of those surfaces which have been processed so far. This memory is called the ***z-buffer*** or the ***depth-buffer.***

When a triangle of the surface is rendered, all those pixels are identified which fall into the interior of the projection of the triangle by a triangle filling algorithm. As the filling algorithm processes a pixel, the Z coordinate of the triangle point visible in this pixel is obtained. If this Z value is larger than the value already stored in the z-buffer, then there exists an already processed triangle that is closer than the current triangle in this given pixel. Thus the current triangle is obscured in this pixel and its colour should not be written into the raster memory. However, if the new Z value is smaller than the value stored in the z-buffer, then the current triangle is the closest so far, and its colour and Z coordinate should be written into the pixel and the z-buffer, respectively.

The z-buffer algorithm is then:

```
Z-BUFFER()
1  for each pixel p                                          ▷ Clear screen.
2      do PIXEL-WRITE(p, background-colour)
3         z-buffer[p] ← maximum value after clipping
4  for each triangle o                                       ▷ Rendering.
5      do for each pixel p of triangle o
6         do Z ← coordinate Z of that point o which projects onto pixel p
7            if Z < z-buffer[p]
8               then PIXEL-WRITE(p, colour of triangle o in this point)
9                    z-buffer[p] ← Z
```

When the triangle is filled, the general polygon filling algorithm of the previous section could be used. However, it is worth exploiting the special features of the triangle. Let us sort the triangle vertices according to their Y coordinates and assign index 1 to the vertex of the smallest Y coordinate and index 3 to the vertex of the largest Y coordinate. The third vertex gets index 2. Then let us cut the triangle into two pieces with scan line Y_2. After cutting we obtain a "lower" triangle and an "upper" triangle. Let us realize that in such triangles the first (left) and the second (right) intersections of the scan lines are always on the same edges, thus the administration of the polygon filling algorithm can be significantly simplified. In fact, the active edge table management is not needed anymore, only the incremental intersection calculation should be implemented. The classification of left and right intersections depend on whether (X_2, Y_2) is on the right or on the left side of the oriented line segment from (X_1, Y_1) to (X_3, Y_3). If (X_2, Y_2) is on the left side, the

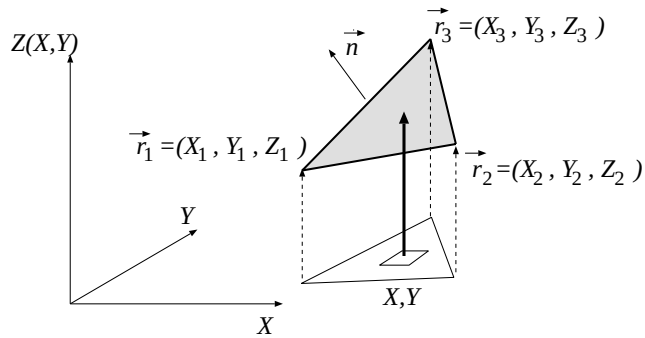


Figure 22.45 A triangle in the screen coordinate system. Pixels inside the projection of the triangle on plane XY need to be found. The Z coordinates of the triangle in these pixels are computed using the equation of the plane of the triangle.

projected triangle is called ***left oriented,*** and ***right oriented*** otherwise.

When the details of the algorithm is introduced, we assume that the already re-indexed triangle vertices are

$$\vec{r}_1 = [X_1, Y_1, Z_1], \qquad \vec{r}_2 = [X_2, Y_2, Z_2], \qquad \vec{r}_3 = [X_3, Y_3, Z_3].$$

The rasterization algorithm is expected to fill the projection of this triangle and also to compute the Z coordinate of the triangle in every pixel (Figure 22.45).

The Z coordinate of the triangle point visible in pixel X, Y is computed using the equation of the plane of the triangle (equation (22.1)):

$$n_X \cdot X + n_Y \cdot Y + n_Z \cdot Z + d = 0 \ , \tag{22.36}$$

where $\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1)$ and $d = -\vec{n} \cdot \vec{r}_1$. Whether the triangle is left oriented or right oriented depends on the sign of the Z coordinate of the normal vector of the plane. If n_Z is negative, then the triangle is left oriented. If it is negative, then the triangle is right oriented. Finally, when n_Z is zero, then the projection maps the triangle onto a line segment, which can be ignored during filling.

Using the equation of the plane, function $Z(X, Y)$ expressing the Z coordinate corresponding to pixel X, Y is:

$$Z(X, Y) = -\frac{n_X \cdot X + n_Y \cdot Y + d}{n_Z} \ . \tag{22.37}$$

According to the incremental principle, the evaluation the Z coordinate can take advantage of the value of the previous pixel:

$$Z(X+1, Y) = Z(X, Y) - \frac{n_X}{n_Z} = Z(X, Y) + \delta Z_X \ . \tag{22.38}$$

Since increment δZ_X is constant for the whole triangle, it needs to be computed only once. Thus the calculation of the Z coordinate in a scan line requires just a single addition per pixel. The Z coordinate values along the edges can also be obtained

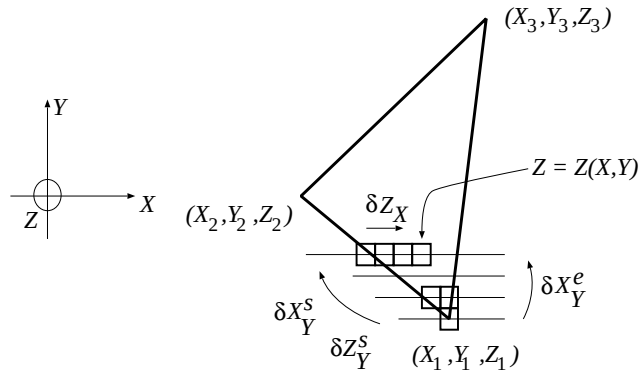


Figure 22.46 Incremental Z coordinate computation for a left oriented triangle.

incrementally from the respective values at the previous scan line (Figure 22.46). The complete incremental algorithm which renders a lower left oriented triangle is as follows (the other cases are very similar):

Z-BUFFER-LOWER-TRIANGLE$(X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3, \mathit{colour})$

```
1  n⃗ ← ((X2, Y2, Z2) − (X1, Y1, Z1)) × ((X3, Y3, Z3) − (X1, Y1, Z1)) ▷ Normal vector.
2  δZX ← −nX/nZ                                                      ▷ Z increment.
3  (δXsY, δZsY, δXeY) ← ((X2 − X1)/(Y2 − Y1), (Z2 − Z1)/(Y2 − Y1), (X3 − X1)/(Y3 − Y1))
4  (Xleft, Xright, Zleft) ← (X1, X1, Z1)
5  for Y ← Y1 to Y2
6      do Z ← Zleft
7          for X ← ROUND(Xleft) to ROUND(Xright)                     ▷ One scan line.
8              do if Z < z-buffer[X, Y]                              ▷ Visibility test.
9                    then PIXEL-WRITE(X, Y, colour)
10                        z-buffer[X, Y] ← Z
11                   Z ← Z + δZX
12         (Xleft, Xright, Zleft) ← (Xleft + δXsY, Xright + δXeY, Zleft + δZsY)
                                                                     ▷Next scan line.
```

This algorithm simultaneously identifies the pixels to be filled and computes the Z coordinates with linear interpolation. Linear interpolation requires just a single addition when a pixel is processed. This idea can also be used for other features as well. For example, if the colour of the triangle vertices are available, the colour of the internal points can be set to provide smooth transitions applying linear interpolation. Note also that the addition to compute the feature value can also be implemented by a special purpose hardware. Graphics cards have a great number of such interpolation units.

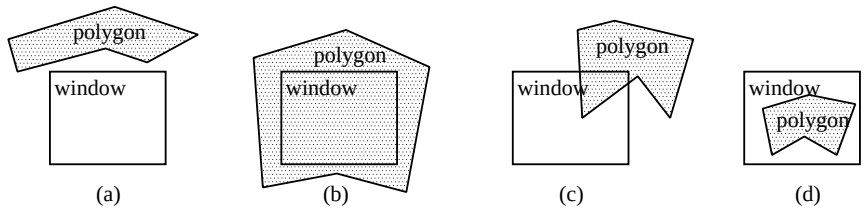


Figure 22.47 Polygon-window relations:: **(a)** distinct; **(b)** surrounding ; **(c)** intersecting; **(d)** contained.

Warnock algorithm. If a pixel of the image corresponds to a given object, then its neighbours usually correspond to the same object, that is, visible parts of objects appear as connected territories on the screen. This is a consequence of object coherence and is called ***image coherence.***

If the situation is so fortunate—from a labor saving point of view—that a polygon in the object scene obscures all the others and its projection onto the image plane covers the image window completely, then we have to do no more than simply fill the image with the colour of the polygon. If no polygon edge falls into the window, then either there is no visible polygon, or some polygon covers it completely (Figure 22.47). The window is filled with the background colour in the first case, and with the colour of the closest polygon in the second case. If at least one polygon edge falls into the window, then the solution is not so simple. In this case, using a divide-and-conquer approach, the window is subdivided into four quarters, and each subwindow is searched recursively for a simple solution.

The basic form of the algorithm called ***Warnock-algorithm*** rendering a rectangular window with screen coordinates X_1, Y_1 (lower left corner) and X_2, Y_2 (upper right corner) is this:

```
WARNOCK(X1, Y1, X2, Y2)
1  if X1 ≠ X2 or Y1 ≠ Y2                          ▷ Is the window larger than a pixel?
2    then if at least one edge projects onto the window
3           then                                  ▷ Non-trivial case: Subdivision and recursion.
4                WARNOCK(X1, Y1, (X1 + X2)/2, (Y1 + Y2)/2)
5                WARNOCK(X1, (Y1 + Y2)/2, (X1 + X2)/2, Y2)
6                WARNOCK((X1 + X2)/2, Y1, X2, (Y1 + Y2)/2)
7                WARNOCK((X1 + X2)/2, (Y1 + Y2)/2, X2, Y2)
8           else          ▷ Trivial case: window (X1, Y1, X2, Y2) is homogeneous.
9                polygon ← the polygon visible in pixel ((X1 + X2)/2, (Y1 + Y2)/2)
10               if no visible polygon
11                 then fill rectangle (X1, Y1, X2, Y2) with the background colour
12                 else fill rectangle (X1, Y1, X2, Y2) with the colour of polygon
```

Note that the algorithm can handle non-intersecting polygons only. The algorithm can be accelerated by filtering out those distinct polygons which can definitely

not be seen in a given subwindow at a given step. Furthermore, if a surrounding polygon appears at a given stage, then all the others behind it can be discarded, that is all those which fall onto the opposite side of it from the eye. Finally, if there is only one contained or intersecting polygon, then the window does not have to be subdivided further, but the polygon (or rather the clipped part of it) is simply drawn. The price of saving further recurrence is the use of a scan-conversion algorithm to fill the polygon.

Painter's algorithm. If we simply scan convert polygons into pixels and draw the pixels onto the screen without any examination of distances from the eye, then each pixel will contain the colour of the last polygon falling onto that pixel. If the polygons were ordered by their distance from the eye, and we took the farthest one first and the closest one last, then the final picture would be correct. Closer polygons would obscure farther ones — just as if they were painted an opaque colour. This method is known as the ***painter's algorithm.***

The only problem is that the order of the polygons necessary for performing the painter's algorithm is not always simple to compute. We say that a polygon P *does not obscure* another polygon Q if none of the points of Q is obscured by P. To have this relation, one of the following conditions should hold

1. Polygons P and Q do not overlap in Z range, and the minimum Z coordinate of polygon P is greater than the maximum Z coordinate of polygon Q.
2. The bounding rectangle of P on the XY plane does not overlap with that of Q.
3. Each vertex of P is farther from the viewpoint than the plane containing Q.
4. Each vertex of Q is closer to the viewpoint than the plane containing P.
5. The projections of P and Q do not overlap on the XY plane.

All these conditions are sufficient. The difficulty of their test increases, thus it is worth testing the conditions in the above order until one of them proves to be true. The first step is the calculation of an ***initial depth order.*** This is done by sorting the polygons according to their maximal Z value into a list. Let us first take the polygon P which is the last item on the resulting list. If the Z range of P does not overlap with any of the preceding polygons, then P is correctly positioned, and the polygon preceding P can be taken instead of P for a similar examination. Otherwise P overlaps a set $\{Q_1, \ldots, Q_m\}$ of polygons. The next step is to try to check whether P *does not* obscure any of the polygons in $\{Q_1, \ldots, Q_m\}$, that is, that P is at its right position despite the overlapping. If it turns out that P obscures Q for a polygon in the set $\{Q_1, \ldots, Q_m\}$, then Q has to be moved behind P in the list, and the algorithm continues stepping back to Q. Unfortunately, this algorithm can run into an infinite loop in case of cyclic overlapping. Cycles can be resolved by cutting. In order to accomplish this, whenever a polygon is moved to another position in the list, we mark it. If a marked polygon Q is about to be moved again, then — assuming that Q is a part of a cycle — Q is cut into two pieces Q_1, Q_2 by

the plane of P, so that Q_1 does not obscure P and P does not obscure Q_2, and only Q_1 is moved behind P.

BSP-tree. Binary space partitioning divides first the space into two halfspaces, the second plane divides the first halfspace, the third plane divides the second halfspace, further planes split the resulting volumes, etc. The subdivision can well be represented by a binary tree, the so-called ***BSP-tree*** illustrated in Figure 22.48. The kd-tree discussed in Subsection 22.6.2 is also a special version of BSP-trees where the splitting planes are parallel with the coordinate planes. The BSP-tree of this subsection, however, uses general planes.

The first splitting plane is associated with the root node of the BSP-tree, the second and third planes are associated with the two children of the root, etc. For our application, not so much the planes, but rather the polygons defining them, will be assigned to the nodes of the tree, and the set of polygons contained by the volume is also necessarily associated with each node. Each leaf node will then contain either no polygon or one polygon in the associated set.

The BSP-Tree-Construction algorithm for creating the BSP-tree for a set S of polygons uses the following notations. A node of the binary tree is denoted by *node*, the polygon associated with the node by *node.polygon*, and the two child nodes by *node.left* and *node.right*, respectively. Let us consider a splitting plane of normal $\vec{n}$ and place vector $\vec{r}_0$. Point $\vec{r}$ belongs to the positive (right) subspace of this plane if the sign of scalar product $\vec{n} \cdot (\vec{r} - \vec{r}_0)$ is positive, otherwise it is in the negative (left) subspace. The BSP construction algorithm is:

BSP-Tree-Construction(S)

```
1  Create a new node
2  if S is empty or contains just a single polygon
3     then node.polygon ← S
4          node.left ← null
5          node.right ← null
6     else node.polygon ← one polygon from list S
7          Remove polygon node.polygon from list S
8          S⁺ ← polygons of S which overlap with the positive subspace
                        of node.polygon
9          S⁻ ← polygons of S which overlap with the negative subspace
                        of node.polygon
10         node.right ← BSP-Tree-Construction(S⁺)
11         node.left ← BSP-Tree-Construction(S⁻)
12 return node
```

The size of the BSP-tree, i.e. the number of polygons stored in it, is on the one hand highly dependent on the nature of the object scene, and on the other hand on the "choice strategy" used when one polygon from list S is selected.

Having constructed the BSP-tree the visibility problem can be solved by traversing the tree in the order that if a polygon obscures another than it is processed later.

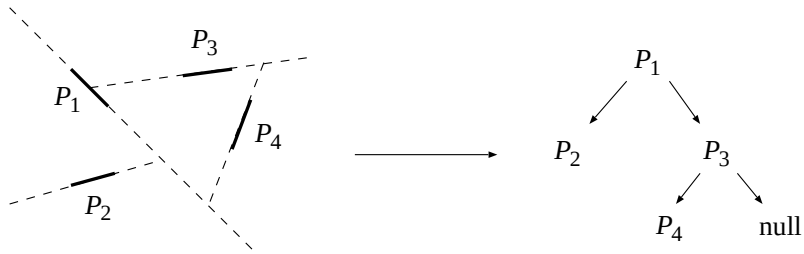


Figure 22.48 A BSP-tree. The space is subdivided by the planes of the contained polygons.

During such a traversal, we determine whether the eye is at the left or right subspace at each node, and continue the traversal in the child *not* containing the eye. Having processed the child not containing the eye, the polygon of the node is drawn and finally the child containing the eye is traversed recursively.

Exercises

22.7-1 Implement the complete Bresenham algorithm that can handle not only moderately ascending but arbitrary line segments.

22.7-2 The presented polygon filling algorithm tests each edges at a scan line whether it becomes active here. Modify the algorithm in a way that such tests are not executed at each scan line, but only once.

22.7-3 Implement the complete z-buffer algorithm that renders left/righ oriented, upper/lower triangles.

22.7-4 Improve the presented Warnock-algorithm and eliminate further recursions when only one edge is projected onto the subwindow.

22.7-5 Apply the BSP-tree for discrete time collision detection.

22.7-6 Apply the BSP-tree as a space partitioning structure for ray tracing.

Problems

22-1 Ray tracing renderer

Implement a rendering system applying the ray tracing algorithm. Objects are defined by triangle meshes and quadratic surfaces, and are associated with diffuse reflectivities. The virtual world also contains point light sources. The visible colour of a point is proportional to the diffuse reflectivity, the intensity of the light source, the cosine of the angle between the surface normal and the illumination direction (Lambert's law), and inversely proportional with the distance of the point and the light source. To detect whether or not a light source is not occluded from a point, use the ray tracing algorithm as well.

22-2 Continuous time collision detection with ray tracing

Using ray tracing develop a continuous time collision detection algorithm which computes the time of collision between a moving and rotating polyhedron and a still half space. Approximate the motion of a polygon vertex by a uniform, constant velocity

motion in small intervals dt.

22-3 Incremental rendering system

Implement a three-dimensional renderer based on incremental rendering. The modelling and camera transforms can be set by the user. The objects are given as triangle meshes, where each vertex has colour information as well. Having transformed and clipped the objects, the z-buffer algorithm should be used for hidden surface removal. The colour at the internal points is obtained by linear interpolation from the vertex colours.

Chapter Notes

The elements of Euclidean, analytic and projective geometry are discussed in the books of Maxwell [19, 20] and Coxeter [6]. The application of projective geometry in computer graphics is presented in Herman's dissertation [14] and Krammer's paper [17]. Curve and surface modelling is the main focus of computer aided geometric design (CAD, CAGD), which is discussed by Gerald Farin [9], and Rogers and Adams [23]. Geometric models can also be obtained measuring real objects, as proposed by *reverse engineering methods* [32]. Implicit surfaces can be studied by reading Bloomenthal's work [2]. Solid modelling with implicit equations is also booming thanks to the emergence of *functional representation methods (F-Rep)*, which are surveyed at http://cis.k.hosei.ac.jp/~F-rep. Blobs have been first proposed by Blinn [1]. Later the exponential influence function has been replaced by polynomials [33], which are more appropriate when roots have to be found in ray tracing.

Geometric algorithms give solutions to geometric problems such as the creation of convex hulls, clipping, containment test, tessellation, point location, etc. This field is discussed in the books of Preparata and Shamos [22] and of Marc de Berg [**?**, 7]. The triangulation of general polygons is still a difficult topic despite to a lot of research efforts. Practical triangulation algorithms run in $O(n \lg n)$ [7, 25, 34], but Chazelle [5] proposed an optimal algorithm having linear time complexity. The presented proof of the *two ears theorem* has originally been given by Joseph O'Rourke [21]. Subdivision surfaces have been proposed and discussed by Catmull and Clark [4], Warren and Weimer [**?**], and by Brian Sharp [27, 26]. The butterfly subdivision approach has been published by Dyn et al. [8]. The *Sutherland–Hodgeman polygon clipping* algorithm is taken from [28].

Collision detection is one of the most critical problems in computer games since it prevents objects to fly through walls and it is used to decide whether a bullet hits an enemy or not. Collision detection algorithms are reviewed by Jiménez, Thomas and Torras [15].

Glassner's book [13] presents many aspects of ray tracing algorithms. The *3D DDA algorithm* has been proposed by Fujimoto et al. [12]. Many papers examined the complexity of ray tracing algorithms. It has been proven that for N objects, ray tracing can be solved in $O(\lg N)$ time [**?**, 30], but this is theoretical rather than practical result since it requires $\Omega(N^4)$ memory and preprocessing time, which is practically unacceptable. In practice, the discussed heuristic schemes are preferred, which are better than the naive approach only in the average case. Heuristic meth-

ods have been analyzed by probabilistic tools by Márton [30], who also proposed the probabilistic scene model used in this chapter as well. We can read about heuristic algorithms, especially about the efficient implementation of the kd-tree based ray tracing in Havran's dissertation [**?**]. A particularly efficient solution is given in Szécsi's paper [**?**].

The probabilistic tools, such as the Poisson point process can be found in the books of Karlin and Taylor [16] and Lamperti [18]. The cited fundamental law of integral geometry can be found in the book of Santaló [24].

The geoinformatics application of quadtrees and octrees are also discussed in chapter 16 of this book.

The algorithms of incremental image synthesis are discussed in many computer graphics textbooks [11]. Visibility algorithms have been compared in [29, 31]. The *painter's algorithm* has been proposed by Newell et al. [**?**]. Fuchs examined the construction of minimal depth BSP-trees [**?**]. The source of the Bresenham algorithm is [3].

Graphics cards implement the algorithms of incremental image synthesis, including transformations, clipping, z-buffer algorithm, which are accessible through graphics libraries *(OpenGL, DirectX)*. Current graphics hardware includes two programmable processors, which enables the user to modify the basic rendering pipeline. Furthermore, this flexibility allows non graphics problems to be solved on the graphics hardware. The reason of using the graphics hardware for non graphics problems is that graphics cards have much higher computational power than CPUs. We can read about such algorithms in the ShaderX or in the GPU Gems [10] series or visiting the http://www.gpgpu.org web page.

Bibliography

[1] J. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):135–256, 1982. 1095

[2] J. Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, 1997. 1095

[3] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. 1096

[4] E. Catmull, J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, 1978. 1095

[5] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):353–363, 1991. 1095

[6] H. S. M. Coxeter. *Projective Geometry*. University of Toronto Press, 1974 (2nd edition). 1095

[7] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000. 1095

[8] N. Dyn, J. Gregory, D. Levin. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9:160–169, 1990. 1095

[9] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Morgan Kaufmann Publishers, 2002 (2nd revised edition). 1095

[10] R. Fernando. *GPUGems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004. 1096

[11] J. D. Fooley, A., S. K. Feiner, J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. 1096

[12] A. Fujimoto, T. Takayuki, I. Kansey. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986. 1095

[13] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989. 1095

[14] I. Herman. *The Use of Projective Geometry in Computer Graphics*. Springer-Verlag, 1991. 1095

[15] P. Jiménez, F. Thomas, C. Torras. 3D collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001. 1095

[16] S. Karlin, M. T. Taylor. *A First Course in Stochastic Processes*. Academic Press, 1975. 1096

[17] G. Krammer. Notes on the mathematics of the PHIGS output pipeline. *Computer Graphics Forum*, 8(8):219–226, 1989. 1095

[18] J. Lamperti. *Stochastic Processes*. Springer-Verlag, 1972. 1096

[19] E. A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogenous Coordinates*. Cambridge University Press, 1946. 1095

[20] E. A. Maxwell. *General Homogenous Coordinates in Space of Three Dimensions*. Cambridge University Press, 1951. 1095

[21] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987. 1095

[22] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 1095

[23] D. F. Rogers, J. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill Book Co., 1989. 1095

[24] L. A. Santaló. *Integral Geometry and Geometric Probability*. Addison-Wesley, 1976. 1096

[25] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991. 1095

[26] B. Sharp. Implementing subdivision theory. *Game Developer*, 7(2):40–45, 2000. 1095

[27] B. Sharp. Subdivision Surface theory. *Game Developer*, 7(1):34–42, 2000. 1095

[28] I. Sutherland, G. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974. 1095

[29] I. E. Sutherland, R. Sproull, R. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974. 1096

[30] L. Szirmay-Kalos, G. Márton. Worst-case versus average-case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998. 1095, 1096

[31] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémiai Kiadó, 1995. 1096

[32] T. Várady, R. R. Martin, J. Cox. Reverse engineering of geometric models - an introduction. *Computer-Aided Design*, 29(4):255–269, 1997. 1095

[33] G. Wyvill, C. McPheeters, B. Wyvill. Data structure for soft objects. *The Visual Computer*, 4(2):227–234, 1986. 1095

[34] B. Zalik, G. Clapworthy. A universal trapezoidation algorithms for planar polygons. *Computers and Graphics*, 23(3):353–363, 1999. 1095

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *exe* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.