LULEÅ TEKNISKA UNIVERSITET

2004:15

C/D-UPPSATS

COMPUTER GRAPHICS

The mathematics behind the scenes

LENNART JORDANSSON

Department of Mathematics
Supervisor: Thomas Gunnarsson

2004:15 • ISSN: 1402–1781 • ISRN: LTU-C/DUPP--04/15--SE

Abstract

This work describes computer graphics with focus on the mathematics behind the scenes. Representations of three-dimensional objects, such as curves, surfaces, and solids are described. Integral and rational Bézier curves, B-splines and NURBS are used for the modeling of curves and surfaces. The objects can be visualized as projections, and a useful model for defining the view of a scene, the *synthetic-camera model*, is described. Basic algorithms for rendering of lines and circles are discussed. Examples of interactive programs in Java which demonstrate rendering of lines, circles, ellipses, and Bézier curves, are given.

Sammanfattning

Denna C/D-uppsats handlar om matematiken bakom datorgrafik. Matematisk beskrivning av tredimensionella objekt, såsom kurvor, ytor och kroppar, behandlas. För modellering av kurvor och ytor används Bézierkurvor, B-splines och NURBS. Objekten kan visualiseras med hjälp av projektioner. En användbar modell för att definiera vyer förklaras. Grundläggande algoritmer för rastrering av linjer och cirklar beskrivs. Interaktiva program skrivna i Java demonstrerar rastrering av linjer, cirklar, ellipser och Bézierkurvor.

Preface

This work is an extended essay in mathematics and concludes the courses MAM603/604 at Luleå University of Technology. The reader is assumed to have a basic knowledge of mathematics at a university level, like calculus, linear algebra and geometry. The paper was written using LaTeX and most of the figures were produced by the mathematics program Maple.

I would like to thank my supervisor *Thomas Gunnarsson* at the *Department of Mathematics* for help and valuable opinions during the work, both regarding the mathematics, and report writing.

Luleå, May 27, 2004
Lennart Jordansson

Contents

List of Figures

Introduction

The paper starts by describing how graphical three-dimensional objects can be represented in a computer. Chapter 1 describes curves, surfaces and solids. The curves and surfaces are modeled as parametric functions. Bézier curves and surfaces are described, as well as B-spline/NURBS curves and surfaces. Chapter 2 discusses how the objects can be visualized as projections, and how to determine which objects in a scene should be seen. A useful model for determining a view, the *synthetic-camera model* is described. In Chapter 3, we show some basic algorithms for rendering lines and circles. Finally, in Chapter 4, some program examples written in Java are shown. The programs are interactive, and let the user explore how the algorithms for rendering work.

Chapter 1

Mathematical Representation of 3D Objects

1.1 Introduction

This chapter describes how graphical 3-dimensional objects can be represented in a computer. The representations of curves are described in Section 1.2, and the representations of surfaces and solids in Section 1.3 and Section 1.4 respectively.

1.2 Representation of 3D Curves

This section shows how curves in 3 dimensions can be expressed by parametric equations. Common special cases, as parametric cubic (pc) curves, Bézier curves, and B-spline curves are described.

1.2.1 Explicit and Implicit Equations

For a point (x, y, z) on a curve, the coordinates y and z can be expressed with two real-valued functions of x as

$$\begin{aligned} y &= f(x) \\ z &= g(x) \end{aligned} \tag{1.1}$$

where $a \leq x \leq b$. This formulation with explicit equations has the drawback that a curve with, for example, multiple values of y for the same value of x has to be broken into multiple curve segments, such that for each segment, we get a single value of y and z respectively, for each value of x. For example, a circle

(in the $z = 0$ plane) with radius r, centered at the origin, can be divided into two half circles C_1 and C_2 as:

$$\begin{array}{rrcc} C_1: & y & = & \sqrt{r^2 - x^2} \\ & z & = & 0 \\ C_2: & y & = & -\sqrt{r^2 - x^2} \\ & z & = & 0 \end{array} \quad , \quad -r \leq x \leq r \tag{1.2}$$

A planar curve can be represented by an implicit equation of the form

$$f(x, y) = 0 \tag{1.3}$$

In this way, the circle from the example above can be expressed as the solutions to the equation

$$x^2 + y^2 - r^2 = 0 \tag{1.4}$$

To express only the half circle where $y \geq 0$, this constraint must be added to the equation above.

As another example, the general second-degree implicit equation

$$Ax^2 + 2Bxy + Cy^2 + Dx + Ey + F = 0 \tag{1.5}$$

can represent lines and conics (ellipses, parabolas, and hyperbolas), by choosing the coefficients A, B, C, D, E, and F appropriately. If A, B, and C are all 0, and at least one of D and E is non-zero, the equation represents a line. If at least one of A, B, and C is non-zero, we have the following cases:

$AC - B^2 > 0$ An ellipse, a point, or the empty set.

$AC - B^2 < 0$ A hyperbola, or two crossing lines.

$AC - B^2 = 0$ A parabola, two parallel lines, a single line, or the empty set.

As an example of the first case, consider the equation

$$\begin{aligned} \frac{7}{48}x^2 - \frac{5\sqrt{3}}{72}xy + \frac{31}{144}y^2 + \left(\frac{5\sqrt{3}}{24} - \frac{7}{6}\right)x \\ + \left(\frac{5\sqrt{3}}{18} - \frac{31}{24}\right)y + \frac{157}{48} - \frac{5\sqrt{3}}{6} &= 0 \end{aligned} \tag{1.6}$$

Comparing this equation with Eq. 1.5 gives A, B, and C as

$$\begin{array}{rcl} A & = & \frac{7}{48} = \frac{21}{144} \\ B & = & -\frac{5\sqrt{3}}{144} \\ C & = & \frac{31}{144} \end{array} \tag{1.7}$$

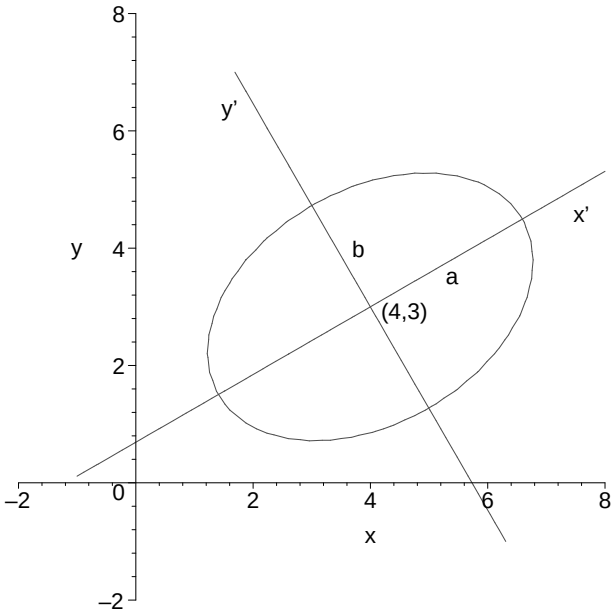

Figure 1.1: Rotated and translated ellipse

and $AC - B^2 > 0$. By the coordinate transformation

$$\begin{aligned} x' &= \frac{\sqrt{3}}{2}(x-4) + \frac{1}{2}(y-3) \\ y' &= -\frac{1}{2}(x-4) + \frac{\sqrt{3}}{2}(y-3) \end{aligned} \tag{1.8}$$

which represents a counterclockwise rotation by $\pi/6$ and a translation of the origin to $(4, 3)$, Eq. 1.6 can be written in the new coordinates x' and y' as

$$\frac{x'^2}{a^2} + \frac{y'^2}{b^2} - 1 = 0 \tag{1.9}$$

where $a = 3$ and $b = 2$. This is the equation for an ellipse. See Figure 1.1. The original and transformed coordinate systems are shown.

1.2.2 Parametric Cubic Curves

Parametric equations use an independent parameter u, and express points on the curve by expressing the coordinates of the points as functions of the parameter. For example, a three-dimensional curve is expressed with three equations

$$\begin{aligned} x &= x(u) \\ y &= y(u) \\ z &= z(u) \end{aligned} \tag{1.10}$$

A point on the three-dimensional curve is given by the row vector

$$\mathbf{p} = \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} \tag{1.11}$$

This form gives more freedom for controlling the shape of a curve, compared to a nonparametric form. It is also easier to handle infinite slopes and transforming the curve into another coordinate system.

Algebraic Form

A parametric cubic (pc) curve segment is expressed by the three polynomials

$$\begin{aligned} x(u) &= a_{3x}u^3 + a_{2x}u^2 + a_{1x}u + a_{0x} \\ y(u) &= a_{3y}u^3 + a_{2y}u^2 + a_{1y}u + a_{0y} \\ z(u) &= a_{3z}u^3 + a_{2z}u^2 + a_{1z}u + a_{0z} \end{aligned} \tag{1.12}$$

where $u \in [0, 1]$. The restriction of u makes the curve segment bounded, since the polynomials given are continuous functions in a closed bounded interval. This choice of the interval for u simplifies creation of composite curves. A curve parametrized within another interval can always be reparametrized to the interval $u \in [0, 1]$, see page 11. The 12 coefficients in Eq. 1.12 are called *algebraic coefficients*. The equation can be written more compactly in vector notation

$$\mathbf{p}(u) = \mathbf{a}_3u^3 + \mathbf{a}_2u^2 + \mathbf{a}_1u + \mathbf{a}_0 \tag{1.13}$$

and by introducing a 1×4 matrix U and a 4×3 matrix A, where $\mathbf{a_0}$, $\mathbf{a_1}$, $\mathbf{a_2}$, and $\mathbf{a_3}$ are row vectors.

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \tag{1.14}$$

$$A = \begin{bmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \\ \mathbf{a}_1 \\ \mathbf{a}_0 \end{bmatrix} = \begin{bmatrix} a_{3x} & a_{3y} & a_{3z} \\ a_{2x} & a_{2y} & a_{2z} \\ a_{1x} & a_{1y} & a_{1z} \\ a_{0x} & a_{0y} & a_{0z} \end{bmatrix} \tag{1.15}$$

the equation for a point on the curve can now be written

$$\mathbf{p} = UA \tag{1.16}$$

As an example, consider the pc curve segment specified by the matrix A, and parametrized in the interval $u \in [0, 1]$

$$A = \begin{bmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \\ \mathbf{a}_1 \\ \mathbf{a}_0 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -12 & -30 \\ 12 & 15 & 30 \\ 3 & 4 & 0 \end{bmatrix} \tag{1.17}$$

The curve is plotted in Figure 1.2 on page 7.

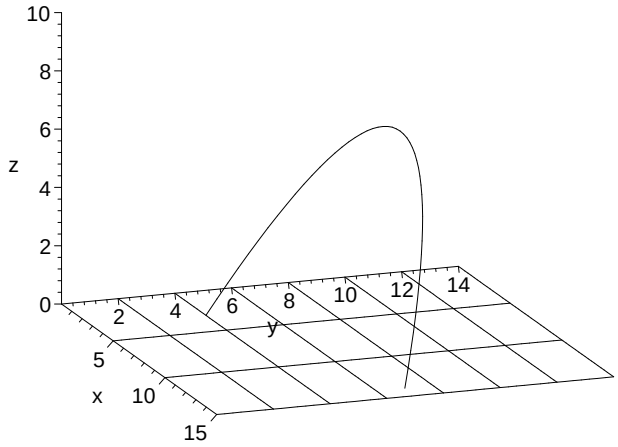

Figure 1.2: Parametric cubic curve

Geometric Form

The algebraic form has the drawback that it is not easy to get an intuitive sense of the shape of the curve when choosing the coefficients. The pc curve can also be defined in terms of the coordinates and tangent vectors at the end points. A cubic curve defined in this way is called a *Hermite curve*. Let $\mathbf{p}_u(u)$ denote the derivative of $\mathbf{p}(u)$ with respect to u

$$\mathbf{p}_u(u) = \frac{d\mathbf{p}(u)}{du} \tag{1.18}$$

By using Eq. 1.13 we get a set of four equations

$$\begin{aligned} \mathbf{p}(0) &= \mathbf{a}_0 \\ \mathbf{p}(1) &= \mathbf{a}_3 + \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0 \\ \mathbf{p}_u(0) &= \mathbf{a}_1 \\ \mathbf{p}_u(1) &= 3\mathbf{a}_3 + 2\mathbf{a}_2 + \mathbf{a}_1 \end{aligned} \tag{1.19}$$

which has the solution

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{p}(0) \\ \mathbf{a}_1 &= \mathbf{p}_u(0) \\ \mathbf{a}_2 &= -3\mathbf{p}(0) + 3\mathbf{p}(1) - 2\mathbf{p}_u(0) - \mathbf{p}_u(1) \\ \mathbf{a}_3 &= 2\mathbf{p}(0) - 2\mathbf{p}(1) + \mathbf{p}_u(0) + \mathbf{p}_u(1) \end{aligned} \tag{1.20}$$

These coefficients are put into Eq. 1.13, and after rearranging the terms we get

$$\begin{aligned} \mathbf{p}(u) &= (2u^3 - 3u^2 + 1)\mathbf{p}(0) + (-2u^3 + 3u^2)\mathbf{p}(1) \\ &\quad + (u^3 - 2u^2 + u)\mathbf{p}_u(0) + (u^3 - u^2)\mathbf{p}_u(1) \end{aligned} \tag{1.21}$$

The equation can be written in the *geometric form*

$$\mathbf{p}(u) = F_1(u)\mathbf{p}(0) + F_2(u)\mathbf{p}(1) + F_3(u)\mathbf{p}_u(0) + F_4(u)\mathbf{p}_u(1) \tag{1.22}$$

where $\mathbf{p}(0)$, $\mathbf{p}(1)$, $\mathbf{p}_u(0)$, and $\mathbf{p}_u(1)$ are called the *geometric coefficients*. The functions F_1, F_2, F_3, and F_4 are called *blending functions*, and are given by

$$\begin{aligned} F_1(u) &= 2u^3 - 3u^2 + 1 \\ F_2(u) &= -2u^3 + 3u^2 \\ F_3(u) &= u^3 - 2u^2 + u \\ F_4(u) &= u^3 - u^2 \end{aligned} \tag{1.23}$$

The blending functions F_1 and F_2 are plotted in Figure 1.3, and F_3 and F_4 in Figure 1.4 on page 9. Eq. 1.22 represents a point on the curve as a weighted sum of the geometric coefficients, with the blending functions acting as weights. For $u = 0$, all blending functions are 0, except F_1, which is 1, and we get $\mathbf{p}(0) = 1 \cdot \mathbf{p}(0)$ as expected. As u increases, the other geometric coefficients start to contribute to the weighted sum. When $u = 1$, $F_2 = 1$ and $F_1 = F_3 = F_4 = 0$, and we get $\mathbf{p}(1) = 1 \cdot \mathbf{p}(1)$. Differentiating Eq. 1.22 with respect to u gives

$$\mathbf{p}_u(u) = F_{1u}(u)\mathbf{p}(0) + F_{2u}(u)\mathbf{p}(1) + F_{3u}(u)\mathbf{p}_u(0) + F_{4u}(u)\mathbf{p}_u(1) \tag{1.24}$$

where $F_{1u}(u)$, $F_{2u}(u)$, $F_{3u}(u)$, and $F_{4u}(u)$ are given by

$$\begin{aligned} F_{1u}(u) &= 6u^2 - 6u \\ F_{2u}(u) &= -6u^2 + 6u \\ F_{3u}(u) &= 3u^2 - 4u + 1 \\ F_{4u}(u) &= 3u^2 - 2u \end{aligned} \tag{1.25}$$

Eq. 1.24 represents the tangent vector of a point on the curve as a weighted sum of the geometric coefficients, where the weights are given by Eq. 1.25. For $u = 0$, $F_{3u} = 1$, and $F_{1u} = F_{2u} = F_{4u} = 0$, and we get $\mathbf{p}_u(0) = 1 \cdot \mathbf{p}_u(0)$. In the same way, for $u = 1$, we get $F_{4u} = 1$, and $F_{1u} = F_{2u} = F_{3u} = 0$, and $\mathbf{p}_u(1) = 1 \cdot \mathbf{p}_u(1)$. This is the reason for the name blending functions, the functions blend the contributions of the geometric coefficients. Eq. 1.22 can be expressed as

$$\mathbf{p} = FB \tag{1.26}$$

where F is the 1×4 matrix of blending functions, and B denotes the 4×3 matrix of geometric coefficients

$$F = \begin{bmatrix} F_1 & F_2 & F_3 & F_4 \end{bmatrix} \tag{1.27}$$

$$B = \begin{bmatrix} \mathbf{p}(0) \\ \mathbf{p}(1) \\ \mathbf{p}_u(0) \\ \mathbf{p}_u(1) \end{bmatrix} = \begin{bmatrix} x(0) & y(0) & z(0) \\ x(1) & y(1) & z(1) \\ x_u(0) & y_u(0) & z_u(0) \\ x_u(1) & y_u(1) & z_u(1) \end{bmatrix} \tag{1.28}$$

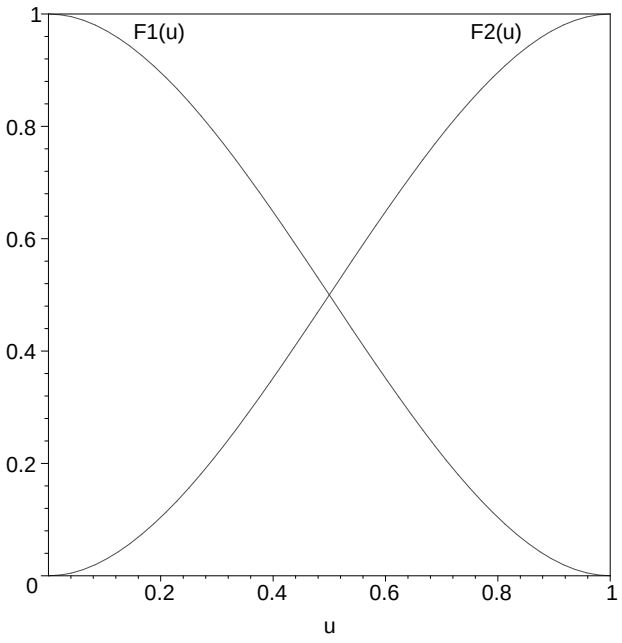

Figure 1.3: Blending functions F_1 and F_2

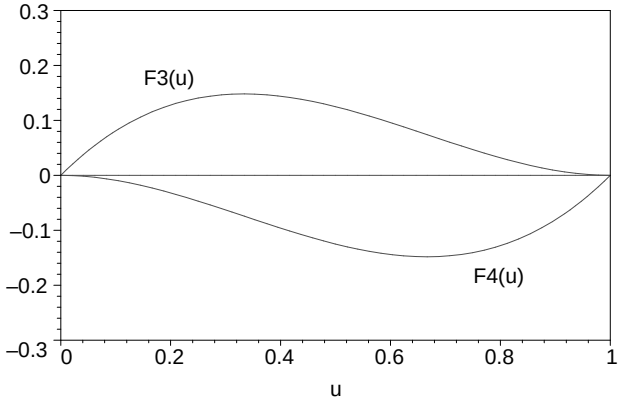

Figure 1.4: Blending functions F_3 and F_4

The matrix of blending functions, F, can be written as a product of the U matrix given by Eq. 1.14 and a 4×4 matrix M

$$F = UM = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{1.29}$$

and using this expression for F in Eq. 1.26 gives

$$\mathbf{p} = FB = UMB \tag{1.30}$$

Comparing this expression with Eq. 1.16 gives

$$\mathbf{p} = UA = UMB \tag{1.31}$$

This is a matrix expression for three equations, one equation for each of $x(u)$, $y(u)$, and $z(u)$. The equations hold for every u, and by choosing three linear independent vectors U_i for $i = 1, 2, 3$, it is possible to write a formula for converting from the geometric form to the algebraic form

$$A = MB \tag{1.32}$$

The inverse of M exists, since the determinant of M is non-zero, (-1), and is given by

$$M^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \tag{1.33}$$

Conversion from the algebraic form to the geometric form is given by

$$B = M^{-1}A \tag{1.34}$$

This equation of the geometric coefficients expressed as a linear combination of the algebraic coefficients is the same as Eq. 1.19, as expected.

The equation expressing the tangent vector of a point on a curve, Eq. 1.24, can also be written in a compact form

$$\mathbf{p}_u = UM_uB = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}(0) \\ \mathbf{p}(1) \\ \mathbf{p}_u(0) \\ \mathbf{p}_u(1) \end{bmatrix} \tag{1.35}$$

The matrices U, F, M, and M_u are the same for all parametric cubic curves, parametrized by $u \in [0, 1]$. The position and shape of the curve is determined completely by either the A or the B matrix, together with $u \in [0, 1]$. As an example of a pc curve specified by the geometric form, consider the curve segment specified by the matrix B, and parametrized in the interval $u \in [0, 1]$

$$B = \begin{bmatrix} \mathbf{p}(0) \\ \mathbf{p}(1) \\ \mathbf{p}_u(0) \\ \mathbf{p}_u(1) \end{bmatrix} = \begin{bmatrix} 3 & 3 & 0 \\ 13 & 6 & 5 \\ 12 & 18 & 6 \\ 12 & 16 & 16 \end{bmatrix} \tag{1.36}$$

The curve is plotted in Figure 1.5 on page 11.

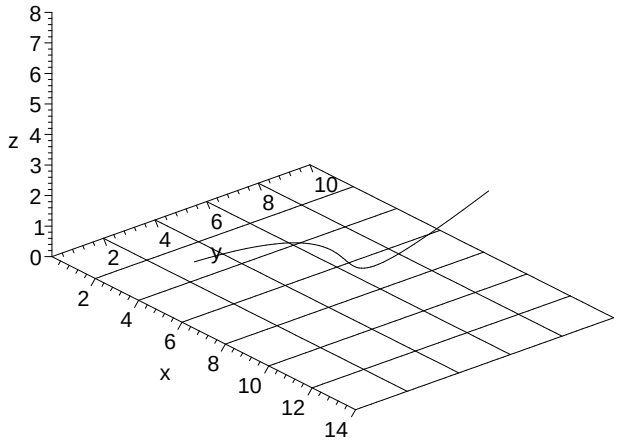

Figure 1.5: Parametric cubic curve

Reparametrization

Reparametrization refers to a change in the parametric interval of a given parametrized curve, while maintaining the position and shape of the curve. In the previous text, the interval of the parameter for a pc curve was $u \in [0, 1]$. Now, consider a curve originally parametrized in the interval $u \in [u_i, u_j]$, and specified in the geometric form with coordinates $\mathbf{p}(u_i)$, $\mathbf{p}(u_j)$ and tangent vectors $\mathbf{p}_u(u_i)$, $\mathbf{p}_u(u_j)$ at the end points. Let the new parameter v be expressed as a linear function of u

$$v = au + b \tag{1.37}$$

The linear relationship between the parameters u and v is needed in order to preserve the cubic form of the curve. The new parameter interval is $v \in [v_i, v_j]$, where v_i and v_j are given by

$$v_i = au_i + b \tag{1.38}$$

$$v_j = au_j + b \tag{1.39}$$

The above equations give a and b as

$$a = \frac{v_j - v_i}{u_j - u_i} \tag{1.40}$$

$$b = v_i - \frac{v_j - v_i}{u_j - u_i} u_i \tag{1.41}$$

Denote the geometric coefficients of the curve parametrized by v as $\mathbf{q}(v_i)$, $\mathbf{q}(v_j)$, $\mathbf{q}_v(v_i)$, and $\mathbf{q}_v(v_j)$. See Figure 1.6 on page 12. A tangent vector for a point on the curve parametrized by v is

$$\mathbf{q}_v = \frac{d\mathbf{q}(v)}{dv} = \frac{d\mathbf{q}(u)}{du} \cdot \frac{du}{dv} = \mathbf{q}_u \cdot \frac{1}{a} = \mathbf{q}_u \cdot \frac{u_j - u_i}{v_j - v_i} \tag{1.42}$$

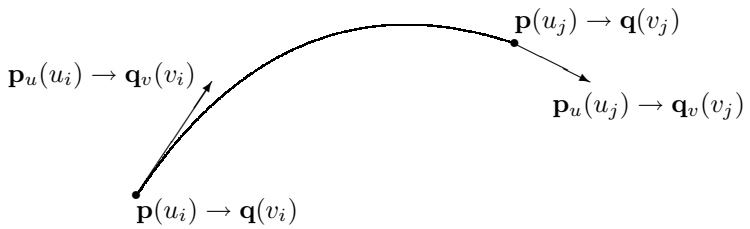

Figure 1.6: Reparametrization

where we have used the chain rule for differentiation, and Eq. 1.37 and Eq. 1.40. The expressions for the geometric coefficients of the curve parametrized by v becomes

$$\begin{aligned} \mathbf{q}(v_i) &= \mathbf{p}(u_i) \\ \mathbf{q}(v_j) &= \mathbf{p}(u_j) \\ \mathbf{q}_v(v_i) &= \frac{u_j - u_i}{v_j - v_i}\mathbf{p}_u(u_i) \\ \mathbf{q}_v(v_j) &= \frac{u_j - u_i}{v_j - v_i}\mathbf{p}_u(u_j) \end{aligned} \tag{1.43}$$

As an example, the reparametrization from $u \in [0,1]$ to $v \in [v_i, v_j]$ is given by

$$\begin{aligned} v &= v_i + (v_j - v_i)u \\ \mathbf{q}(v_i) &= \mathbf{p}(0) \\ \mathbf{q}(v_j) &= \mathbf{p}(1) \\ \mathbf{q}_v(v_i) &= \frac{1}{v_j - v_i}\mathbf{p}_u(0) \\ \mathbf{q}_v(v_j) &= \frac{1}{v_j - v_i}\mathbf{p}_u(1) \end{aligned} \tag{1.44}$$

Composite Parametric Cubic Curves

Consider the following problem: two disjoint pc curves $\mathbf{p}_1$ and $\mathbf{p}_3$ are given, and we wish to define a pc curve $\mathbf{p}_2$, connecting the given curves. See Figure 1.7 on page 13. $\mathbf{p}_1$ and $\mathbf{p}_3$ are specified with the geometric coefficients B_1 and B_3 respectively.

$$B_1 = \begin{bmatrix} \mathbf{p}_1(0) \\ \mathbf{p}_1(1) \\ \mathbf{p}_{1u}(0) \\ \mathbf{p}_{1u}(1) \end{bmatrix} \tag{1.45}$$

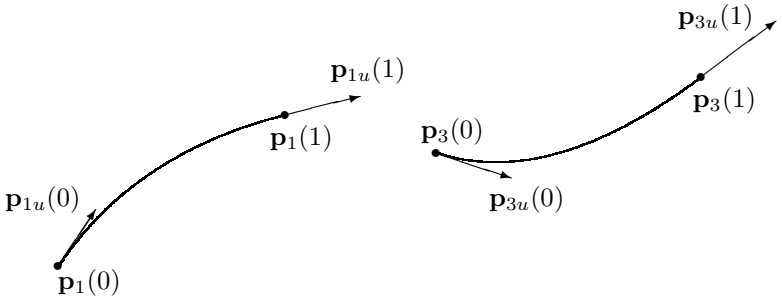

Figure 1.7: Composite curves

$$B_3 = \begin{bmatrix} \mathbf{p}_3(0) \\ \mathbf{p}_3(1) \\ \mathbf{p}_{3u}(0) \\ \mathbf{p}_{3u}(1) \end{bmatrix} \tag{1.46}$$

Before connecting the curve $\mathbf{p}_2$ in between $\mathbf{p}_1$ and $\mathbf{p}_3$, we have to decide the degree of continuity we want. If we are only interested of connecting the curves without gaps, it is sufficient to state that

$$\mathbf{p}_2(0) = \mathbf{p}_1(1) \tag{1.47}$$
$$\mathbf{p}_2(1) = \mathbf{p}_3(0) \tag{1.48}$$

this is called C^0 continuity. If we also state that the tangent vectors at the end points of $\mathbf{p}_2$ should be equal to the tangent vectors at $\mathbf{p}_1(1)$ and $\mathbf{p}_3(0)$ respectively, we get C^1 continuity, that is, we have the additional requirement

$$\mathbf{p}_{2u}(0) = \mathbf{p}_{1u}(1) \tag{1.49}$$
$$\mathbf{p}_{2u}(1) = \mathbf{p}_{3u}(0) \tag{1.50}$$

Now, the pc curve $\mathbf{p}_2$ is completely specified, in summary

$$B_2 = \begin{bmatrix} \mathbf{p}_1(1) \\ \mathbf{p}_3(0) \\ \mathbf{p}_{1u}(1) \\ \mathbf{p}_{3u}(0) \end{bmatrix} \tag{1.51}$$

We can not in general achieve C^2 continuity at a connection of two parametric cubic curves. In order to get C^2 continuity, one has to use a curve with a higher degree.

Sometimes, it is sufficient to state C^0 continuity, and that the tangent vectors at the connecting point should have the same direction, the magnitudes of the

tangent vectors need not to be same. This is called *geometric continuity*, and is denoted G^1. For the curve $\mathbf{p}_2$, the condition can be written

$$\mathbf{p}_{2u}(0) = k_1 \frac{\mathbf{p}_{1u}(1)}{|\mathbf{p}_{1u}(1)|} \tag{1.52}$$

$$\mathbf{p}_{2u}(1) = k_2 \frac{\mathbf{p}_{3u}(0)}{|\mathbf{p}_{3u}(0)|} \tag{1.53}$$

In this way, there is some freedom of varying the shape of $\mathbf{p}_2$ by varying k_1 and k_2.

1.2.3 Bézier Curves

A Bézier curve of degree n is defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \quad , \quad u \in [0,1] \tag{1.54}$$

where $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ are given *control points*, and the blending functions $B_{i,n}(u)$, also called *Bernstein polynomials*, defined by

$$B_{i,n}(u) = \begin{cases} \binom{n}{i}(1-u)^{n-i}u^i & \text{, if } 0 \le i \le n \\ 0 & \text{, otherwise} \end{cases} \tag{1.55}$$

where the *binomial coefficients* $\binom{n}{i}$ are defined by

$$\binom{n}{i} = \frac{n!}{(n-i)!i!} \qquad \text{, for } 0 \le i \le n \tag{1.56}$$

The binomial coefficients arise from the binomial theorem, which states that

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i \tag{1.57}$$

for any real numbers x and y, and for any natural number n.

The polygon with the control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ in this order as vertices, with an edge from $\mathbf{p}_n$ to $\mathbf{p}_0$ is called the *control polygon* of the Bézier curve. As an example, consider a planar cubic Bézier curve with control points

$$\begin{aligned} \mathbf{p}_0 &= (1,1) \\ \mathbf{p}_1 &= (2,3) \\ \mathbf{p}_2 &= (4,5) \\ \mathbf{p}_3 &= (6,2) \end{aligned}$$

Using Eq. 1.54 with $n = 3$ gives

$$\mathbf{p}(u) = (1-u)^3\mathbf{p}_0 + 3(1-u)^2u\mathbf{p}_1 + 3(1-u)u^2\mathbf{p}_2 + u^3\mathbf{p}_3 \tag{1.58}$$

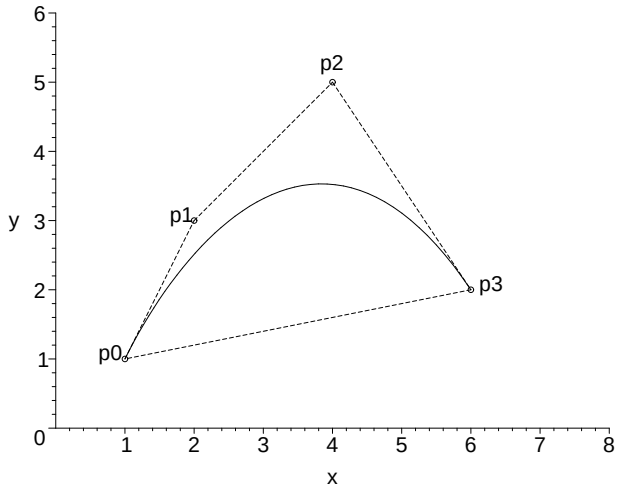

Figure 1.8: Planar cubic Bézier curve

and after expanding this equation and using the given control points, we can write the equations for each coordinate for a point on the curve as

$$x(u) = -u^3 + 3u^2 + 3u + 1 \tag{1.59}$$
$$y(u) = -5u^3 + 6u + 1 \tag{1.60}$$

where $u \in [0, 1]$. The curve is shown i Figure 1.8, which also shows the control points and the control polygon. The corresponding Bernstein polynomials of degree 3, $B_{i,3}(u)$ for $i = 0, \ldots, 3$ is plotted in Figure 1.9 on page 16.

Properties of the Bernstein Polynomials

The Bernstein polynomials $B_{i,n}(u)$ have some useful properties, which will be used for proving properties of Bézier curves.

Theorem 1.1 Partition of Unity
The Bernstein polynomials of degree n sum to 1, that is

$$\sum_{i=0}^{n} B_{i,n}(u) = 1 \quad , \quad u \in [0, 1] \tag{1.61}$$

Proof:
We can write 1 as $(1 - u) + u$ and use the binomial theorem

$$1 = 1^n = ((1 - u) + u)^n = \sum_{i=0}^{n} \binom{n}{i} (1 - u)^{n-i} u^i = \sum_{i=0}^{n} B_{i,n}(u) \tag{1.62}$$

which proofs the theorem.
□

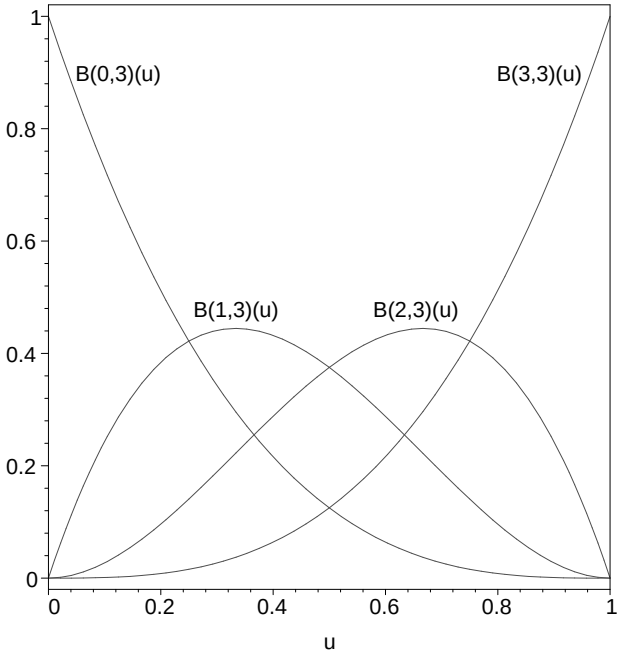

Figure 1.9: Bernstein polynomials of degree 3

Theorem 1.2 Positivity
The Bernstein polynomials are non-negative on the interval $[0, 1]$

$$B_{i,n}(u) \geq 0 \quad , \quad u \in [0, 1] \tag{1.63}$$

Proof:
By writing $B_{i,n}(u)$ using Eq. 1.55 and Eq. 1.56 we get

$$B_{i,n}(u) = \binom{n}{i}(1-u)^{n-i}u^i = \frac{n!}{(n-i)!i!}(1-u)^{n-i}u^i \tag{1.64}$$

The binomial coefficient $\binom{n}{i} > 0$, and since $u \in [0, 1]$, $(1 - u) \geq 0$ and

$$(1-u)^{n-i} \geq 0 \tag{1.65}$$
$$u^i \geq 0 \tag{1.66}$$

which proofs the theorem.
□

The partition of unity and positivity properties are used later for proving the convex hull property of a Bézier curve, and that an affine transformation of a Bézier curve is also a Bézier curve. See page 23.

Theorem 1.3 Symmetry
The Bernstein polynomials have the following symmetry

$$B_{n-i,n}(u) = B_{i,n}(1-u) \quad , \quad \textit{for } 0 \leq i \leq n \tag{1.67}$$

Proof:
The left-hand side of Eq. 1.67 can be written

$$\begin{aligned} B_{n-i,n}(u) &= \frac{n!}{(n-(n-i))!(n-i)!}(1-u)^{n-(n-i)}u^{n-i} \\ &= \frac{n!}{i!(n-i)!}(1-u)^{i}u^{n-i} \end{aligned} \tag{1.68}$$

The right-hand side of Eq. 1.67 becomes

$$\begin{aligned} B_{i,n}(1-u) &= \frac{n!}{(n-i)!i!}(1-(1-u))^{n-i}(1-u)^{i} \\ &= \frac{n!}{(n-i)!i!}u^{n-i}(1-u)^{i} \end{aligned} \tag{1.69}$$

which proofs the theorem.
□

The positivity and symmetry properties can be seen in Figure 1.9 on page 16, for $n = 3$. The symmetry property implies that a symmetric control polygon gives a symmetric Bézier curve.

Lemma 1.1 *The following expression holds for the binomial coefficients*

$$\binom{n}{i} + \binom{n}{i+1} = \binom{n+1}{i+1} \tag{1.70}$$

Proof:

$$\begin{aligned} &\binom{n}{i} + \binom{n}{i+1} = \frac{n!}{(n-i)!i!} + \frac{n!}{(n-(i+1))!(i+1)!} = \\ &\frac{n!(i+1)}{(n-i)!i!(i+1)} + \frac{(n-i)n!}{(n-i)(n-i-1)!(i+1)!} = \\ &\frac{n!(i+1)}{(n-i)!(i+1)!} + \frac{(n-i)n!}{(n-i)!(i+1)!} = \\ &\frac{n!i+n!+nn!-in!}{(n-i)!(i+1)!} = \frac{n!(n+1)}{(n-i)!(i+1)!} = \\ &\frac{(n+1)!}{((n+1)-(i+1))!(i+1)!} = \binom{n+1}{i+1} \end{aligned} \tag{1.71}$$

□

Theorem 1.4 Recursion
The Bernstein polynomials of degree n can be expressed in terms of the polynomials of degree $n-1$

$$B_{i,n}(u) = (1-u)B_{i,n-1}(u) + uB_{i-1,n-1}(u) \quad , \quad \textit{for } 0 \leq i \leq n \tag{1.72}$$

where, according to the definition of the Bernstein polynomials, Eq. 1.55,

$$B_{-1,n-1}(u) = 0 \tag{1.73}$$
$$B_{n,n-1}(u) = 0 \tag{1.74}$$

Proof:
For $i = 0$, we get by Eq. 1.55

$$B_{0,n}(u) = \binom{n}{0}(1-u)^n u^0 = (1-u)^n \tag{1.75}$$
$$B_{0,n-1}(u) = (1-u)^{n-1} \tag{1.76}$$

By using Eq. 1.73 and the equations above, we can write

$$B_{0,n}(u) = (1-u)B_{0,n-1}(u) + uB_{-1,n-1}(u) \tag{1.77}$$

For $i = n$, we get by Eq. 1.55

$$B_{n,n}(u) = \binom{n}{n}(1-u)^0 u^n = u^n \tag{1.78}$$
$$B_{n-1,n-1}(u) = u^{n-1} \tag{1.79}$$

By using Eq. 1.74 and the equations above, we can write

$$B_{n,n}(u) = (1-u)B_{n,n-1}(u) + uB_{n-1,n-1}(u) \tag{1.80}$$

For $1 \leq i \leq n-1$ we write the right-hand side of Eq. 1.72 using Eq. 1.55, and Lemma 1.1

$$\begin{aligned}
&(1-u)B_{i,n-1}(u) + uB_{i-1,n-1}(u) = \\
&\binom{n-1}{i}(1-u)^{n-1-i+1}u^i + \binom{n-1}{i-1}(1-u)^{n-1-(i-1)}u^{i-1+1} = \\
&\left[\binom{n-1}{i} + \binom{n-1}{i-1}\right](1-u)^{n-i}u^i = \binom{n}{i}(1-u)^{n-i}u^i = \\
&B_{i,n}(u)
\end{aligned} \tag{1.81}$$

We thus get for $0 \leq i \leq n$

$$B_{i,n}(u) = (1-u)B_{i,n-1}(u) + uB_{i-1,n-1}(u) \tag{1.82}$$

and the proof is complete.
□

The recursion property is used later for proving the de Casteljau algorithm. See page 26.

Theorem 1.5 First Derivative
The first derivative of the Bernstein polynomials $B_{i,n}(u)$ satisfies

$$\frac{d}{du}B_{i,n}(u) = n(B_{i-1,n-1}(u) - B_{i,n-1}(u)) \tag{1.83}$$

Proof:
Differentiating $B_{i,n}(u)$ by the product rule gives

$$\begin{aligned}
&\frac{d}{du}B_{i,n}(u) = \frac{n!}{(n-i)!i!}\frac{d}{du}((1-u)^{n-i}u^i) = \\
&\frac{n!}{(n-i)!i!}(-(n-i)(1-u)^{n-i-1}u^i + (1-u)^{n-i}iu^{i-1}) = \\
&\frac{-(n-i)n!}{(n-i)!i!}(1-u)^{n-i-1}u^i + \frac{n!i}{(n-i)!i!}(1-u)^{n-i}u^{i-1} = \\
&\frac{-n(n-1)!}{(n-i-1)!i!}(1-u)^{n-1-i}u^i + \\
&+ \frac{n(n-1)!}{((n-1)-(i-1))!(i-1)!}(1-u)^{(n-1)-(i-1)}u^{i-1} = \\
&-nB_{i,n-1}(u) + nB_{i-1,n-1}(u) = \\
&n(B_{i-1,n-1}(u) - B_{i,n-1}(u))
\end{aligned} \tag{1.84}$$

which proofs the theorem.
□

Convex Hull

Let $\mathbf{X}$ be a given set of points, where the points $\mathbf{x}_i$ are represented as vectors.

$$\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n\} \tag{1.85}$$

The *convex hull* of $\mathbf{X}$, denoted by $CH\{\mathbf{X}\}$, is defined to be the following set of points

$$CH\{\mathbf{X}\} = \left\{ a_0\mathbf{x}_0 + a_1\mathbf{x}_1 + \cdots + a_n\mathbf{x}_n \,\middle|\, \sum_{i=0}^{n} a_i = 1, a_i \geq 0 \right\} \tag{1.86}$$

For points in a plane, one can visualize the convex hull of the set of points by placing an elastic band around the set of points. The band is shrunk to form a convex polygon, with vertices which are a subset of the original set of points. The region bounded by the polygon is the convex hull of the points. For points in space, the elastic band is replaced by an elastic balloon, which is shrunk around the points to form a convex polyhedron. The convex hull is the region bounded by the polyhedron. See Figure 1.10 on page 20.

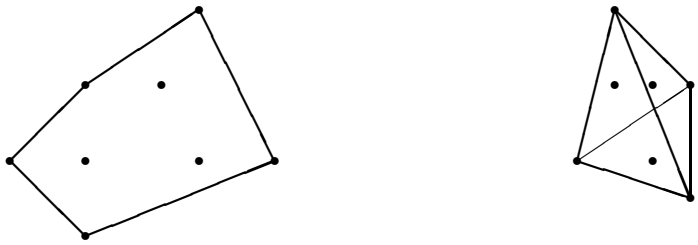

Figure 1.10: Convex hull of points in a plane and in space

Properties of Bézier Curves

Bézier curves have some properties which are useful for modeling, since it is easy to get an intuitive sense of the shape of the curve.

Theorem 1.6 End Point Interpolation Property
A Bézier curve $\mathbf{p}(u)$ *of degree* n *with control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ *satisfies*

$$\mathbf{p}(0) = \mathbf{p}_0 \tag{1.87}$$
$$\mathbf{p}(1) = \mathbf{p}_n \tag{1.88}$$

Proof:
Using the definition of the Bézier curve, Eq. 1.54, and writing the terms in the sum, we get

$$\begin{aligned}\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) = \sum_{i=0}^{n} \mathbf{p}_i \binom{n}{i} (1-u)^{n-i} u^i = \\ \mathbf{p}_0 \binom{n}{0} (1-u)^n + \mathbf{p}_1 \binom{n}{1} (1-u)^{n-1} u + \cdots \\ \cdots + \mathbf{p}_{n-1} \binom{n}{n-1} (1-u) u^{n-1} + \mathbf{p}_n \binom{n}{n} u^n\end{aligned} \tag{1.89}$$

For $u = 0$, all terms in the sum are 0 except the first, so

$$\mathbf{p}(0) = \mathbf{p}_0 \binom{n}{0} (1-0)^n = \mathbf{p}_0 \cdot 1 \cdot 1 = \mathbf{p}_0 \tag{1.90}$$

For $u = 1$, all terms are 0 except the last, and we get

$$\mathbf{p}(1) = \mathbf{p}_n \binom{n}{n} 1^n = \mathbf{p}_n \cdot 1 \cdot 1 = \mathbf{p}_n \tag{1.91}$$

and the proof is complete.
□

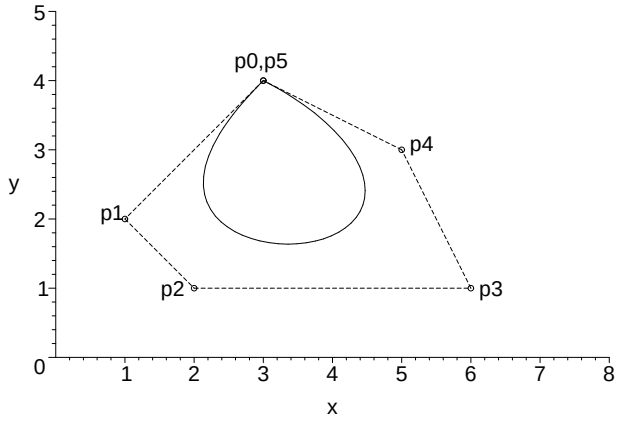

Figure 1.11: Closed planar Bézier curve of degree 5

The end point interpolation property can be seen in Figure 1.8 on page 15. As a consequence of this property, we can get a closed Bézier curve by setting $\mathbf{p}_0 = \mathbf{p}_n$. For example, consider the Bézier curve of degree 5, defined by the control points

$$\begin{aligned} \mathbf{p}_0 &= (3,4) \\ \mathbf{p}_1 &= (1,2) \\ \mathbf{p}_2 &= (2,1) \\ \mathbf{p}_3 &= (6,1) \\ \mathbf{p}_4 &= (5,3) \\ \mathbf{p}_5 &= (3,4) \end{aligned}$$

The coordinates for a point $\mathbf{p}(u)$ on the curve is given by

$$x(u) = 20u^5 - 40u^4 + 30u^2 - 10u + 3 \tag{1.92}$$
$$y(u) = -5u^5 + 5u^4 + 10u^2 - 10u + 4 \tag{1.93}$$

where $u \in [0,1]$. The curve is shown i Figure 1.11, which also shows the control points and the control polygon.

Theorem 1.7 First Derivative
The first derivative of a Bézier curve $\mathbf{p}(u)$ *of degree* n *with control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ *can be written as a Bézier curve of degree* $n-1$

$$\mathbf{p}_u(u) = \frac{d}{du}\mathbf{p}(u) = \sum_{i=0}^{n-1} \mathbf{q}_i B_{i,n-1}(u) \tag{1.94}$$

where the control points $\mathbf{q}_i$ *is given by*

$$\mathbf{q}_i = n(\mathbf{p}_{i+1} - \mathbf{p}_i) \tag{1.95}$$

Proof:
A Bézier curve of degree n is given by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \tag{1.96}$$

The first derivative of $\mathbf{p}(u)$ is

$$\mathbf{p}_u(u) = \sum_{i=0}^{n} \mathbf{p}_i \frac{d}{du} B_{i,n}(u) \tag{1.97}$$

Using Theorem 1.5 we can write

$$\begin{aligned} \mathbf{p}_u(u) &= \sum_{i=0}^{n} \mathbf{p}_i n(B_{i-1,n-1}(u) - B_{i,n-1}(u)) \\ &= \sum_{i=0}^{n} \mathbf{p}_i n B_{i-1,n-1}(u) - \sum_{i=0}^{n} \mathbf{p}_i n B_{i,n-1}(u) \end{aligned} \tag{1.98}$$

Since $B_{-1,n-1}(u) = 0$ and $B_{n,n-1}(u) = 0$, we can remove these terms from the above equation, which gives

$$\mathbf{p}_u(u) = \sum_{i=1}^{n} \mathbf{p}_i n B_{i-1,n-1}(u) - \sum_{i=0}^{n-1} \mathbf{p}_i n B_{i,n-1}(u) \tag{1.99}$$

Renumbering the first sum by replacing i with $i+1$ gives

$$\mathbf{p}_u(u) = \sum_{i=0}^{n-1} \mathbf{p}_{i+1} n B_{i,n-1}(u) - \sum_{i=0}^{n-1} \mathbf{p}_i n B_{i,n-1}(u) \tag{1.100}$$

The two sums are then written as a single sum

$$\begin{aligned} \mathbf{p}_u(u) &= \sum_{i=0}^{n-1} n(\mathbf{p}_{i+1} - \mathbf{p}_i) B_{i,n-1}(u) \\ &= \sum_{i=0}^{n-1} \mathbf{q}_i B_{i,n-1}(u) \end{aligned} \tag{1.101}$$

where $\mathbf{q}_i = n(\mathbf{p}_{i+1} - \mathbf{p}_i)$. This expresses a Bézier curve of degree $n-1$, with control points $\mathbf{q}_i$, and the proof is complete.
□

Theorem 1.8 End Point Tangent Property
The tangent vectors at the end points of a Bézier curve $\mathbf{p}(u)$ *of degree* n *with control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ *can be written*

$$\mathbf{p}_u(0) = n(\mathbf{p}_1 - \mathbf{p}_0) \tag{1.102}$$

$$\mathbf{p}_u(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1}) \tag{1.103}$$

Proof:
By Theorem 1.7 we can write

$$\mathbf{p}_u(u) = \mathbf{q}(u) = \sum_{i=0}^{n-1} \mathbf{q}_i B_{i,n-1}(u) \tag{1.104}$$

where

$$\mathbf{q}_i = n(\mathbf{p}_{i+1} - \mathbf{p}_i) \tag{1.105}$$

Using Theorem 1.6, where the degree of $\mathbf{q}(u)$ is $n-1$, we get

$$\mathbf{q}(0) = \mathbf{q}_0 \tag{1.106}$$
$$\mathbf{q}(1) = \mathbf{q}_{n-1} \tag{1.107}$$

Using Eq. 1.105 we can write

$$\mathbf{p}_u(0) = \mathbf{q}(0) = \mathbf{q}_0 = n(\mathbf{p}_1 - \mathbf{p}_0) \tag{1.108}$$
$$\mathbf{p}_u(1) = \mathbf{q}(1) = \mathbf{q}_{n-1} = n(\mathbf{p}_n - \mathbf{p}_{n-1}) \tag{1.109}$$

and the proof is complete.
□

Theorem 1.9 Convex Hull Property
Every point on a Bézier curve lies inside the convex hull of its control points. That is

$$\mathbf{p}(u) \in CH\{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n\} \ , \ \textit{for } u \in [0,1] \tag{1.110}$$

Proof:
By the definition of convex hull, Eq. 1.86, it is sufficient to show that each point $\mathbf{p}(u)$ on a Bézier curve can be written

$$\mathbf{p}(u) = a_0\mathbf{p}_0 + a_1\mathbf{p}_1 + \cdots + a_n\mathbf{p}_n \tag{1.111}$$

for some constants $a_i \geq 0$ satisfying

$$\sum_{i=0}^{n} a_i = 1 \tag{1.112}$$

Let $a_i = B_{i,n}(u)$. By the properties positivity and partition of unity of the Bernstein polynomials, the conditions of a_i are satisfied which proofs the theorem.
□

The convex hull property of a Bézier curve can be seen in Figure 1.11 on page 21.

Theorem 1.10 Invariance under Affine Transformations
Let T be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the Bézier curve

$\mathbf{p}(u)$ *can be written as a Bézier curve where the control points are transformed by* T*, that is*

$$T\left(\sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u)\right) = \sum_{i=0}^{n} T(\mathbf{p}_i) B_{i,n}(u) \tag{1.113}$$

Proof:
Let the affine transformation T be given by the matrix expression

$$T(\mathbf{p}) = \mathbf{p}A + \mathbf{b} \tag{1.114}$$

The transformation of a point $\mathbf{p}(u)$ on the curve is

$$T(\mathbf{p}(u)) = \left[\sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u)\right] A + \mathbf{b} \tag{1.115}$$

By the partition of unity property of $B_{i,n}(u)$ we can write this as

$$T(\mathbf{p}(u)) = \left[\sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u)\right] A + \left[\sum_{i=0}^{n} B_{i,n}(u)\right] \mathbf{b} \tag{1.116}$$

and then write as a single sum

$$T(\mathbf{p}(u)) = \sum_{i=0}^{n} (\mathbf{p}_i A + \mathbf{b}) B_{i,n}(u) = \sum_{i=0}^{n} T(\mathbf{p}_i) B_{i,n}(u) \tag{1.117}$$

and the proof is complete.
□

This means that instead of transforming each point $\mathbf{p}(u)$ of a Bézier curve, it is sufficient to transform only the control points.

Changing a Control Point

If a control point is changed, the entire curve is affected. This is called *global control*. This can be seen by studying the Bernstein polynomials, which are all non-zero for $u \in (0, 1)$.

For example, consider the cubic Bézier curve in Figure 1.12 on page 25, when changing $\mathbf{p}_1$ from $(2, 3)$ to $(3, 5)$. The new control point $\mathbf{p}_1 = (3, 5)$ lies on the line through $\mathbf{p}_0$ and the original $\mathbf{p}_1$. By the end point tangent property, the direction of the curve at $u = 0$ is thus unchanged.

As another example, if $\mathbf{p}_1$ is changed to $(1, 4)$, the direction of the curve at $u = 0$ is changed, since $(1, 4)$ does not lie on the line through $\mathbf{p}_0$ and the original $\mathbf{p}_1$. See Figure 1.13 on page 25, which shows both the old and the new curve.

Control Point with Multiplicity

If we choose k consecutive control points to be equal, we get a control point with *multiplicity* k. This has the effect of pulling the curve closer to that point.

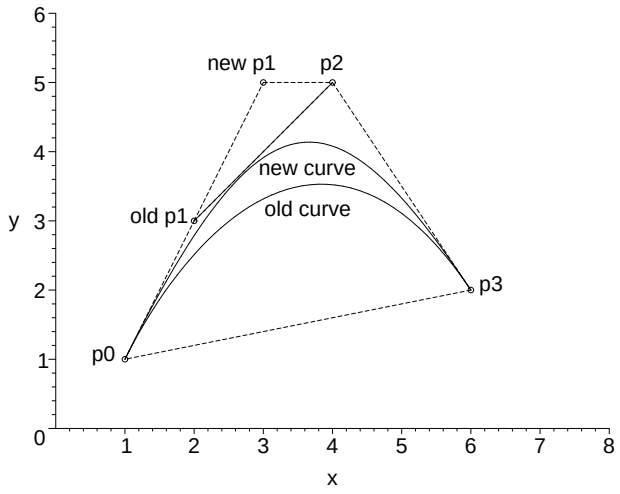

Figure 1.12: Planar cubic Bézier curves: $\mathbf{p}_1$ is changed, direction at $u = 0$ is not changed

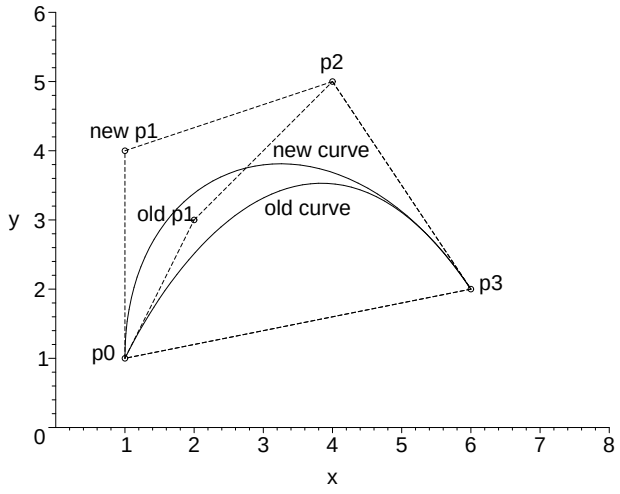

Figure 1.13: Planar cubic Bézier curves: $\mathbf{p}_1$ is changed, direction at $u = 0$ is changed

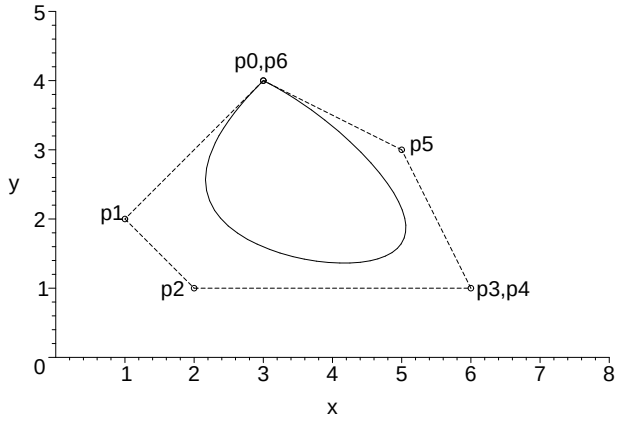

Figure 1.14: Closed planar Bézier curve of degree 6 with a multiple point at $(6, 1)$

For example, consider the Bézier curve in Figure 1.11 on page 21. By choosing a point with multiplicity 2 at $(6, 1)$ we get the curve shown in Figure 1.14. Since we have added an extra control point, the degree of the curve is increased to 6.

The de Casteljau Algorithm

The de Casteljau algorithm is a method for computing a point $\mathbf{p}(u)$ on a Bézier curve, for a given value of the parameter $u \in [0, 1]$. The algorithm can also be used for division of a Bézier curve into two Bézier curves, see page 29.

Theorem 1.11 The de Casteljau Algorithm
Let a Bézier curve $\mathbf{p}(u)$ of degree n be defined by control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$, and let $u \in [0, 1]$ be any given parameter value. Then $\mathbf{p}(u) = \mathbf{p}_0^n$ where

$$\begin{aligned} \mathbf{p}_i^0 &= \mathbf{p}_i & (1.118) \\ \mathbf{p}_i^j &= \mathbf{p}_i^{j-1}(1-u) + \mathbf{p}_{i+1}^{j-1}u & (1.119) \end{aligned}$$

where i and j are indices with $j = 1, \ldots, n$ and $i = 0, \ldots, n - j$.
Proof:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \tag{1.120}$$

By the recursion property of the Bernstein polynomials, Eq. 1.72, we can write

$$\begin{aligned} \mathbf{p}(u) &= \sum_{i=0}^{n} \mathbf{p}_i \left[(1-u)B_{i,n-1}(u) + uB_{i-1,n-1}(u)\right] \\ &= \sum_{i=0}^{n} \mathbf{p}_i (1-u)B_{i,n-1}(u) + \sum_{i=0}^{n} \mathbf{p}_i uB_{i-1,n-1}(u) \qquad (1.121) \end{aligned}$$

Since $B_{-1,n-1}(u) = 0$ and $B_{n,n-1}(u) = 0$, we can remove these terms from the above equation, which gives

$$\mathbf{p}(u) = \sum_{i=0}^{n-1} \mathbf{p}_i(1-u)B_{i,n-1}(u) + \sum_{i=1}^{n} \mathbf{p}_i u B_{i-1,n-1}(u) \tag{1.122}$$

Renumbering the second sum by replacing i with $i+1$ gives

$$\begin{aligned} \mathbf{p}(u) &= \sum_{i=0}^{n-1} \mathbf{p}_i(1-u)B_{i,n-1}(u) + \sum_{i=0}^{n-1} \mathbf{p}_{i+1} u B_{i,n-1}(u) \\ &= \sum_{i=0}^{n-1} \left[\mathbf{p}_i(1-u) + \mathbf{p}_{i+1}u\right] B_{i,n-1}(u) \end{aligned} \tag{1.123}$$

Set $\mathbf{p}_i^1 = \mathbf{p}_i(1-u) + \mathbf{p}_{i+1}u$ and $\mathbf{p}_i^0 = \mathbf{p}_i$. Thus, for $i = 0, \ldots, n-1$ we have

$$\mathbf{p}_i^1 = \mathbf{p}_i^0(1-u) + \mathbf{p}_{i+1}^0 u \tag{1.124}$$

$$\mathbf{p}(u) = \sum_{i=0}^{n-1} \mathbf{p}_i^1 B_{i,n-1}(u) \tag{1.125}$$

This equation expresses $\mathbf{p}(u)$ as a Bézier curve of degree $n-1$ with control points $\mathbf{p}_0^1, \ldots, \mathbf{p}_{n-1}^1$. Applying a similar argument to Eq. 1.125 gives for $i = 0, \ldots, n-2$

$$\mathbf{p}_i^2 = \mathbf{p}_i^1(1-u) + \mathbf{p}_{i+1}^1 u \tag{1.126}$$

$$\mathbf{p}(u) = \sum_{i=0}^{n-2} \mathbf{p}_i^2 B_{i,n-2}(u) \tag{1.127}$$

and in general, for $i = 0, \ldots, n-j$

$$\mathbf{p}_i^j = \mathbf{p}_i^{j-1}(1-u) + \mathbf{p}_{i+1}^{j-1} u \tag{1.128}$$

$$\mathbf{p}(u) = \sum_{i=0}^{n-j} \mathbf{p}_i^j B_{i,n-j}(u) \tag{1.129}$$

In particular, $j = n$ gives

$$\mathbf{p}(u) = \sum_{i=0}^{0} \mathbf{p}_i^n B_{i,0}(u) = \mathbf{p}_0^n B_{0,0}(u) = \mathbf{p}_0^n \tag{1.130}$$

and the proof is complete.
□

The de Casteljau algorithm for a cubic Bézier curve can be depicted with the following diagram

$$
\begin{array}{lllllll}
 & \mathbf{p}_0^0 & & \mathbf{p}_1^0 & & \mathbf{p}_2^0 & \mathbf{p}_3^0 \\
 & \downarrow (1-u) \quad \swarrow u & & \downarrow (1-u) \quad \swarrow u & & \downarrow (1-u) \quad \swarrow u & \\
j=1 & \mathbf{p}_0^1 & & \mathbf{p}_1^1 & & \mathbf{p}_2^1 & \\
 & \downarrow (1-u) \quad \swarrow u & & \downarrow (1-u) \quad \swarrow u & & & \\
j=2 & \mathbf{p}_0^2 & & \mathbf{p}_1^2 & & & \\
 & \downarrow (1-u) \quad \swarrow u & & & & & \\
j=3 & \mathbf{p}_0^3 & & & & &
\end{array}
$$

For example, $\mathbf{p}_0^1$ is computed as $\mathbf{p}_0^1 = \mathbf{p}_0^0(1-u) + \mathbf{p}_1^0 u$, for a given value of the parameter u. As a numeric example, consider the cubic Bézier curve given in Figure 1.8 on page 15, with the control points

$$
\begin{aligned}
\mathbf{p}_0 &= (1,1) \\
\mathbf{p}_1 &= (2,3) \\
\mathbf{p}_2 &= (4,5) \\
\mathbf{p}_3 &= (6,2)
\end{aligned}
$$

To compute the point on the curve $\mathbf{p}(u)$ at $u = 0.5$ we get

$$
\begin{array}{llllll}
(1,1) & & (2,3) & & (4,5) & (6,2) \\
\downarrow 0.5 & \swarrow 0.5 & \downarrow 0.5 & \swarrow 0.5 & \downarrow 0.5 \quad \swarrow 0.5 & \\
(1.5,2) & & (3,4) & & (5,3.5) & \\
\downarrow 0.5 & \swarrow 0.5 & \downarrow 0.5 & \swarrow 0.5 & & \\
(2.25,3) & & (4,3.75) & & & \\
\downarrow 0.5 & \swarrow 0.5 & & & & \\
(3.125,3.375) & & & & &
\end{array}
$$

By computing $\mathbf{p}(u)$ for $u = 0.5$ using Eq. 1.59 and Eq. 1.60 for $x(u)$ and $y(u)$ respectively, we also get $(3.125, 3.375)$, as expected.

Subdivision of a Bézier Curve

A Bézier curve $\mathbf{p}(u)$ is in general defined on the interval $u \in [0,1]$ and given by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \tag{1.131}$$

Suppose that the curve is cut at $u = \alpha$ to give two curve segments, denoted $\mathbf{p}_{left}(u)$ and $\mathbf{p}_{right}(u)$ defined on the intervals $u \in [0,\alpha]$ and $u \in [\alpha,1]$ respectively. Theorem 1.12 on page 32 gives expressions for the two curve segments. The following two lemmas are used in the proof of the theorem.

Lemma 1.2 *The $\mathbf{p}_k^j$ computed for $u = \alpha$ in the de Casteljau algorithm satisfy*

$$\mathbf{p}_k^j = \sum_{i=0}^{j} \mathbf{p}_{i+k} B_{i,j}(\alpha) \tag{1.132}$$

Proof:
The lemma is proved by induction over j. In the de Casteljau algorithm $\mathbf{p}_k^j$ are given by

$$\mathbf{p}_k^j = \mathbf{p}_k^{j-1}(1-\alpha) + \mathbf{p}_{k+1}^{j-1}\alpha \tag{1.133}$$

for $j = 1,\ldots,n$ and $k = 0,\ldots,n-j$. $\mathbf{p}_k^0 = \mathbf{p}_k$ for $k = 0,\ldots,n$.
For $j = 1$, the left-hand side of Eq. 1.132 is

$$\mathbf{p}_k^1 = \mathbf{p}_k^0(1-\alpha) + \mathbf{p}_{k+1}^0\alpha = \mathbf{p}_k(1-\alpha) + \mathbf{p}_{k+1}\alpha \tag{1.134}$$

and the right-hand side is

$$\begin{aligned}
\sum_{i=0}^{1} \mathbf{p}_{i+k} B_{i,1}(\alpha) &= \mathbf{p}_k B_{0,1}(\alpha) + \mathbf{p}_{k+1} B_{1,1}(\alpha) \\
&= \mathbf{p}_k \binom{1}{0}(1-\alpha)^1\alpha^0 + \mathbf{p}_{k+1}\binom{1}{1}(1-\alpha)^0\alpha^1 \\
&= \mathbf{p}_k(1-\alpha) + \mathbf{p}_{k+1}\alpha
\end{aligned} \tag{1.135}$$

That is, Eq. 1.132 is true for $j = 1$.
The induction hypothesis is that Eq. 1.132 is true for $j = p < n$

$$\mathbf{p}_k^p = \sum_{i=0}^{p} \mathbf{p}_{i+k} B_{i,p}(\alpha) \tag{1.136}$$

We want to show that it is also true for $j = p+1$. By the de Casteljau algorithm with $j = p+1$ we get

$$\mathbf{p}_k^{p+1} = \mathbf{p}_k^p(1-\alpha) + \mathbf{p}_{k+1}^p\alpha \tag{1.137}$$

Using the induction hypothesis, Eq. 1.136, we can write the above equation as

$$\mathbf{p}_k^{p+1} = \sum_{i=0}^{p} \mathbf{p}_{i+k} B_{i,p}(\alpha)(1-\alpha) + \sum_{i=0}^{p} \mathbf{p}_{i+k+1} B_{i,p}(\alpha)\alpha \tag{1.138}$$

The second sum is renumbered by replacing $i+1$ by i

$$\mathbf{p}_k^{p+1} = \sum_{i=0}^{p} \mathbf{p}_{i+k} B_{i,p}(\alpha)(1-\alpha) + \sum_{i=1}^{p+1} \mathbf{p}_{i+k} B_{i-1,p}(\alpha)\alpha \tag{1.139}$$

Since $B_{p+1,p}(\alpha) = 0$ and $B_{-1,p}(\alpha) = 0$, we can write the sums as

$$\begin{aligned} \mathbf{p}_k^{p+1} &= \sum_{i=0}^{p+1} \mathbf{p}_{i+k} B_{i,p}(\alpha)(1-\alpha) + \sum_{i=0}^{p+1} \mathbf{p}_{i+k} B_{i-1,p}(\alpha)\alpha \\ &= \sum_{i=0}^{p+1} \mathbf{p}_{i+k} (B_{i,p}(\alpha)(1-\alpha) + B_{i-1,p}(\alpha)\alpha) \end{aligned} \tag{1.140}$$

Using the recursion property of the Bernstein polynomials, the equation can be written

$$\mathbf{p}_k^{p+1} = \sum_{i=0}^{p+1} \mathbf{p}_{i+k} B_{i,p+1}(\alpha) \tag{1.141}$$

By the induction axiom, it follows that

$$\mathbf{p}_k^{j} = \sum_{i=0}^{j} \mathbf{p}_{i+k} B_{i,j}(\alpha) \tag{1.142}$$

for $j = 1, \ldots, n$ and the proof is complete.
□

Lemma 1.3 *The following property holds for the Bernstein polynomials*

$$B_{i,n}(\alpha u) = \sum_{j=0}^{n} B_{i,j}(\alpha) B_{j,n}(u) \quad , \; 0 \le i \le n \tag{1.143}$$

where $u \in [0,1]$ *and* $\alpha \in [0,1]$.
Proof:
The lemma is proved by induction over n.
For $n = 0$ and $0 \le i \le n$ the left-hand side of Eq. 1.143 is

$$B_{0,0}(\alpha u) = \binom{0}{0} (1-\alpha u)^0 (\alpha u)^0 = 1 \tag{1.144}$$

and the right-hand side is

$$\sum_{j=0}^{0} B_{0,j}(\alpha) B_{j,0}(u) = B_{0,0}(\alpha) B_{0,0}(u) = 1 \cdot 1 = 1 \tag{1.145}$$

That is, Eq. 1.143 is true for $n = 0$.
The induction hypothesis is that Eq. 1.143 is true for $n = p$

$$B_{i,p}(\alpha u) = \sum_{j=0}^{p} B_{i,j}(\alpha) B_{j,p}(u) \quad , \; 0 \le i \le p \tag{1.146}$$

We want to show that it is also true for $n = p+1$. Using the recursion property of Bernstein polynomials we can write

$$B_{i,p+1}(\alpha u) = (1-\alpha u)B_{i,p}(\alpha u) + \alpha u B_{i-1,p}(\alpha u) \tag{1.147}$$

Using the induction hypothesis, Eq. 1.146, we can write the above equation as

$$B_{i,p+1}(\alpha u) = (1-\alpha u)\sum_{j=0}^{p} B_{i,j}(\alpha)B_{j,p}(u) + \alpha u \sum_{j=0}^{p} B_{i-1,j}(\alpha)B_{j,p}(u) \tag{1.148}$$

and then write as a single sum

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p} \left[(1-\alpha u)B_{i,j}(\alpha) + \alpha u B_{i-1,j}(\alpha)\right] B_{j,p}(u) \tag{1.149}$$

Adding $-uB_{i,j}(\alpha) + uB_{i,j}(\alpha) = 0$ into the sum gives

$$\begin{aligned} B_{i,p+1}(\alpha u) &= \sum_{j=0}^{p} [B_{i,j}(\alpha) - uB_{i,j}(\alpha) \\ &+ uB_{i,j}(\alpha) - \alpha u B_{i,j}(\alpha) + \alpha u B_{i-1,j}(\alpha)] B_{j,p}(u) \end{aligned} \tag{1.150}$$

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p} \left[B_{i,j}(\alpha)(1-u) + u((1-\alpha)B_{i,j}(\alpha) + \alpha B_{i-1,j}(\alpha))\right] B_{j,p}(u) \tag{1.151}$$

Using the recursion property of the Bernstein polynomials gives

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p} \left[B_{i,j}(\alpha)(1-u) + uB_{i,j+1}(\alpha)\right] B_{j,p}(u) \tag{1.152}$$

Writing the sum as two sums gives

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p} B_{i,j}(\alpha)(1-u)B_{j,p}(u) + \sum_{j=0}^{p} B_{i,j+1}(\alpha)uB_{j,p}(u) \tag{1.153}$$

Renumbering the second sum by replacing j by $j-1$ gives

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p} B_{i,j}(\alpha)(1-u)B_{j,p}(u) + \sum_{j=1}^{p+1} B_{i,j}(\alpha)uB_{j-1,p}(u) \tag{1.154}$$

Since $B_{p+1,p}(u) = 0$ and $B_{-1,p}(u) = 0$ we can write the sums as

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p+1} B_{i,j}(\alpha)(1-u)B_{j,p}(u) + \sum_{j=0}^{p+1} B_{i,j}(\alpha)uB_{j-1,p}(u) \tag{1.155}$$

and then write as a single sum

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p+1} B_{i,j}(\alpha)\left[(1-u)B_{j,p}(u) + uB_{j-1,p}(u)\right] \tag{1.156}$$

Using the recursion property of the Bernstein polynomials gives

$$B_{i,p+1}(\alpha u) = \sum_{j=0}^{p+1} B_{i,j}(\alpha) B_{j,p+1}(u) \tag{1.157}$$

By the induction axiom, it follows that

$$B_{i,n}(\alpha u) = \sum_{j=0}^{n} B_{i,j}(\alpha) B_{j,n}(u) \tag{1.158}$$

and the proof is complete.
□

Theorem 1.12 Subdivision of a Bézier Curve
A general Bézier curve of degree n, with control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \tag{1.159}$$

which is cut at $u = \alpha$ *gives two Bézier curves* $\mathbf{p}_{left}(u)$ *and* $\mathbf{p}_{right}(u)$*, both defined on the interval* $u \in [0, 1]$

$$\mathbf{p}_{left}(u) = \sum_{j=0}^{n} \mathbf{p}_0^j B_{j,n}(u) \tag{1.160}$$

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_i^{n-i} B_{i,n}(u) \tag{1.161}$$

where the $\mathbf{p}_i^j$ *are the points computed for* $u = \alpha$ *in the de Casteljau algorithm.*
Proof:
Suppose $\mathbf{p}(u)$ is divided at $u = \alpha$. Define $\mathbf{p}_{left}(u)$ by

$$\mathbf{p}_{left}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \ , \ u \in [0, \alpha] \tag{1.162}$$

The curve can be reparametrized as

$$\mathbf{p}_{left}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(\alpha u) \ , \ u \in [0, 1] \tag{1.163}$$

By Lemma 1.3, $B_{i,n}(\alpha u)$ is

$$B_{i,n}(\alpha u) = \sum_{j=0}^{n} B_{i,j}(\alpha) B_{j,n}(u) \tag{1.164}$$

Using this expression in the equation for $\mathbf{p}_{left}(u)$ gives

$$\mathbf{p}_{left}(u) = \sum_{i=0}^{n} \mathbf{p}_i \sum_{j=0}^{n} B_{i,j}(\alpha) B_{j,n}(u) \tag{1.165}$$

Changing the order of the sums gives

$$\mathbf{p}_{left}(u) = \sum_{j=0}^{n} \sum_{i=0}^{n} \mathbf{p}_i B_{i,j}(\alpha) B_{j,n}(u) \tag{1.166}$$

When $i > j$, $B_{i,j}(\alpha) = 0$, so we can write

$$\mathbf{p}_{left}(u) = \sum_{j=0}^{n} \sum_{i=0}^{j} \mathbf{p}_i B_{i,j}(\alpha) B_{j,n}(u) \tag{1.167}$$

Lemma 1.2 states that

$$\mathbf{p}_k^j = \sum_{i=0}^{j} \mathbf{p}_{i+k} B_{i,j}(\alpha) \tag{1.168}$$

Using this expression with $k = 0$ in the equation for $\mathbf{p}_{left}(u)$ gives

$$\mathbf{p}_{left}(u) = \sum_{j=0}^{n} \mathbf{p}_0^j B_{j,n}(u) \tag{1.169}$$

which proofs the theorem for $\mathbf{p}_{left}(u)$.
Define $\mathbf{p}_{right}(u)$ by

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \ , \ u \in [\alpha, 1] \tag{1.170}$$

We can reparametrize by substituting u for $1 - u$

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(1-u) \ , \ u \in [0, 1-\alpha] \tag{1.171}$$

By the symmetry property of the Bernstein polynomials we can write

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{n-i,n}(u) \ , \ u \in [0, 1-\alpha] \tag{1.172}$$

Renumbering the sum by replacing i by $n - i$ gives

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_{n-i} B_{i,n}(u) \ , \ u \in [0, 1-\alpha] \tag{1.173}$$

We reparametrize for $u \in [0, 1]$

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n} \mathbf{p}_{n-i} B_{i,n}((1-\alpha)u) \ , \ u \in [0, 1] \tag{1.174}$$

By Lemma 1.3

$$B_{i,n}((1-\alpha)u) = \sum_{j=0}^{n} B_{i,j}(1-\alpha) B_{j,n}(u) \tag{1.175}$$

Using this expression in the equation for $\mathbf{p}_{right}(u)$ gives

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n}\sum_{j=0}^{n} \mathbf{p}_{n-i} B_{i,j}(1-\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.176}$$

We reparametrize by substituting u for $1-u$

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n}\sum_{j=0}^{n} \mathbf{p}_{n-i} B_{i,j}(1-\alpha) B_{j,n}(1-u) \ , \ u \in [0,1] \tag{1.177}$$

By the symmetry property of the Bernstein polynomials we can write

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n}\sum_{j=0}^{n} \mathbf{p}_{n-i} B_{i,j}(1-\alpha) B_{n-j,n}(u) \ , \ u \in [0,1] \tag{1.178}$$

We renumber the inner sum by replacing j by $n-j$

$$\mathbf{p}_{right}(u) = \sum_{i=0}^{n}\sum_{j=0}^{n} \mathbf{p}_{n-i} B_{i,n-j}(1-\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.179}$$

Changing the order of the sums gives

$$\mathbf{p}_{right}(u) = \sum_{j=0}^{n}\sum_{i=0}^{n} \mathbf{p}_{n-i} B_{i,n-j}(1-\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.180}$$

When $i > n-j$, $B_{i,n-j}(1-\alpha) = 0$, so we can write

$$\mathbf{p}_{right}(u) = \sum_{j=0}^{n}\sum_{i=0}^{n-j} \mathbf{p}_{n-i} B_{i,n-j}(1-\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.181}$$

By the symmetry property of the Bernstein polynomials we can write

$$\mathbf{p}_{right}(u) = \sum_{j=0}^{n}\sum_{i=0}^{n-j} \mathbf{p}_{n-i} B_{n-j-i,n-j}(\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.182}$$

We renumber the inner sum by replacing i by $n-j-i$

$$\mathbf{p}_{right}(u) = \sum_{j=0}^{n}\sum_{i=0}^{n-j} \mathbf{p}_{i+j} B_{i,n-j}(\alpha) B_{j,n}(u) \ , \ u \in [0,1] \tag{1.183}$$

By Lemma 1.2

$$\mathbf{p}_k^j = \sum_{i=0}^{j} \mathbf{p}_{i+k} B_{i,j}(\alpha) \tag{1.184}$$

With j for k, and $n-j$ for j we can write this as

$$\mathbf{p}_j^{n-j} = \sum_{i=0}^{n-j} \mathbf{p}_{i+j} B_{i,n-j}(\alpha) \tag{1.185}$$

Using this expression in the equation for $\mathbf{p}_{right}(u)$ gives

$$\mathbf{p}_{right}(u) = \sum_{j=0}^{n} \mathbf{p}_j^{n-j} B_{j,n}(u) \ , \ u \in [0,1] \tag{1.186}$$

which proofs the theorem for $\mathbf{p}_{right}(u)$, and the proof is complete.
□

As an example of subdivision of a cubic Bézier curve, consider the the Bézier curve defined by the control points

$$\begin{aligned} \mathbf{p}_0 &= (1,1) \\ \mathbf{p}_1 &= (2,3) \\ \mathbf{p}_2 &= (4,5) \\ \mathbf{p}_3 &= (6,2) \end{aligned}$$

These control points were also used in Figure 1.8 on page 15. The curve is divided at $u = 0.5$. The two curves, $\mathbf{p}_{left}(u)$ and $\mathbf{p}_{right}(u)$ are defined by the points computed in the de Casteljau algorithm for $u = 0.5$. See the diagram of the example on page 28. $\mathbf{p}_{left}(u)$ is defined by the control points $\mathbf{q}_i = \mathbf{p}_0^i$, for $i = 0, \ldots, 3$

$$\begin{aligned} \mathbf{q}_0 = \mathbf{p}_0^0 &= (1,1) \\ \mathbf{q}_1 = \mathbf{p}_0^1 &= (1.5,2) \\ \mathbf{q}_2 = \mathbf{p}_0^2 &= (2.25,3) \\ \mathbf{q}_3 = \mathbf{p}_0^3 &= (3.125,3.375) \end{aligned}$$

and $\mathbf{p}_{right}(u)$ is defined by the control points $\mathbf{r}_i = \mathbf{p}_i^{3-i}$, for $i = 0, \ldots, 3$

$$\begin{aligned} \mathbf{r}_0 = \mathbf{p}_0^3 &= (3.125,3.375) \\ \mathbf{r}_1 = \mathbf{p}_1^2 &= (4,3.75) \\ \mathbf{r}_2 = \mathbf{p}_2^1 &= (5,3.5) \\ \mathbf{r}_3 = \mathbf{p}_3^0 &= (6,2) \end{aligned}$$

The two curves are shown in Figure 1.15 on page 36. The original control polygon and the control polygons for $\mathbf{p}_{left}(u)$ and $\mathbf{p}_{right}(u)$ are also shown. Compare this with Figure 1.8 on page 15.

1.2.4 Rational Bézier Curves

A *rational* Bézier curve of degree n is defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i R_{i,n}(u) \ , \ u \in [0,1] \tag{1.187}$$

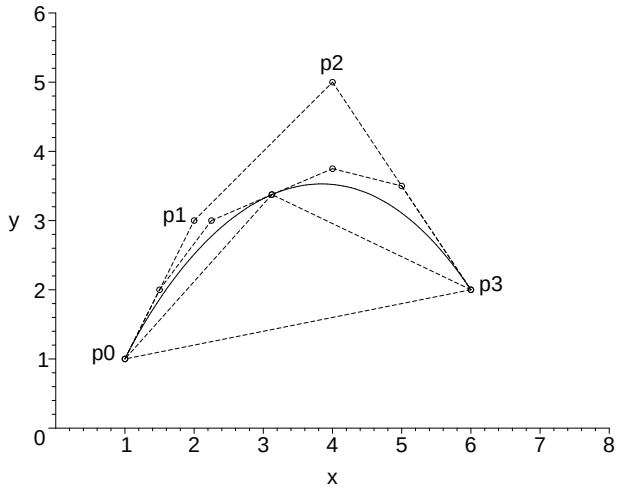

Figure 1.15: Subdivided cubic Bézier curve

where $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ are given control points, and the blending functions $R_{i,n}(u)$ are defined by

$$R_{i,n}(u) = \begin{cases} \frac{w_i B_{i,n}(u)}{\sum_{j=0}^{n} w_j B_{j,n}(u)} & , \quad \text{if } w_i \neq 0 \\ \frac{B_{i,n}(u)}{\sum_{j=0}^{n} w_j B_{j,n}(u)} & , \quad \text{if } w_i = 0 \end{cases} \tag{1.188}$$

where w_i is a scalar weight for control point $\mathbf{p}_i$. At least one of the weights $w_0, w_1, \ldots, w_n$ is non-zero. If we choose all the weights to be equal and non-zero, that is, $w = w_0 = w_1 = \cdots = w_n$, we can factor out w from the sum in the denominator of the first expression in Eq. 1.188, and then use the partition of unity property of $B_{i,n}(u)$ to get

$$R_{i,n}(u) = \frac{wB_{i,n}(u)}{\sum_{j=0}^{n} wB_{j,n}(u)} = \frac{wB_{i,n}(u)}{w\sum_{j=0}^{n} B_{j,n}(u)} = B_{i,n}(u) \tag{1.189}$$

that is, by setting all weights w_i equal and non-zero, we get the Bézier curves described in Section 1.2.3 (often called *integral* Bézier curves), as special cases of rational Bézier curves.

As an example of a planar quadratic rational Bézier curve, consider the curve defined by the control points

$$\begin{aligned} \mathbf{p}_0 &= (1, 0) \\ \mathbf{p}_1 &= (1, 1) \\ \mathbf{p}_2 &= (0, 1) \end{aligned}$$

and by the weights

$$w_0 = 1$$

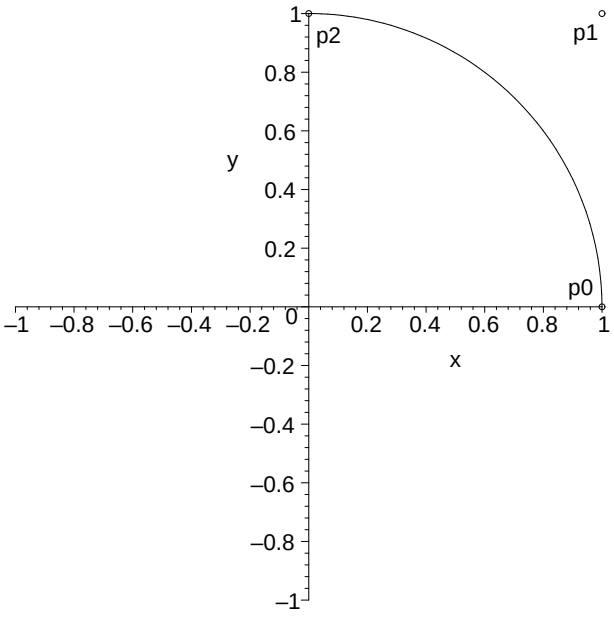

Figure 1.16: Planar quadratic rational Bézier curve

$$\begin{aligned} w_1 &= 1 \\ w_2 &= 2 \end{aligned}$$

Using Eq. 1.187 and Eq. 1.188 we can write the expression for a point of the curve as

$$\mathbf{p}(u) = \frac{w_0(1-u)^2\mathbf{p}_0 + 2w_1(1-u)u\mathbf{p}_1 + w_2u^2\mathbf{p}_2}{w_0(1-u)^2 + 2w_1(1-u)u + w_2u^2} \tag{1.190}$$

By using the given control points and weights, and simplifying, we can write the expressions for the coordinates of the point on the curve as

$$x(u) = \frac{1-u^2}{1+u^2} \tag{1.191}$$

$$y(u) = \frac{2u}{1+u^2} \tag{1.192}$$

This is a parametrization of a unit circle, and for $u \in [0,1]$, we get the part of the circle which lies in the first quadrant. See Figure 1.16.

Properties of Rational Bézier Curves

Rational Bézier curves have similar properties as integral Bézier curves.
Theorems 1.13–1.16 can be proved by first proving that $R_{i,n}(u)$ satisfy the positivity property and the partition of unity property, provided that all the weights satisfy $w_i > 0$.

Theorem 1.13 End Point Interpolation Property
A rational Bézier curve $\mathbf{p}(u)$ *of degree* n *with control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$*, and weights* $w_0, w_1, \ldots, w_n$*, where* $w_0 \neq 0$ *and* $w_n \neq 0$*, satisfies*

$$\mathbf{p}(0) = \mathbf{p}_0 \tag{1.193}$$

$$\mathbf{p}(1) = \mathbf{p}_n \tag{1.194}$$

Theorem 1.14 End Point Tangent Property
The tangent vectors at the end points of a rational Bézier curve $\mathbf{p}(u)$ *of degree* n *with control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$*, and weights* $w_0, w_1, \ldots, w_n$*, where* $w_0 \neq 0$ *and* $w_n \neq 0$*, can be written*

$$\mathbf{p}_u(0) = n\frac{w_1}{w_0}(\mathbf{p}_1 - \mathbf{p}_0) \tag{1.195}$$

$$\mathbf{p}_u(1) = n\frac{w_{n-1}}{w_n}(\mathbf{p}_n - \mathbf{p}_{n-1}) \tag{1.196}$$

Theorem 1.15 Convex Hull Property
Every point on a rational Bézier curve lies inside the convex hull of its control points, provided that all weights $w_i > 0$*. That is*

$$\mathbf{p}(u) \in CH\{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n\} \ , \ \textit{for } u \in [0, 1] \tag{1.197}$$

Theorem 1.16 Invariance under Affine Transformations
Let T *be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the rational Bézier curve* $\mathbf{p}(u)$ *can be written as a rational Bézier curve where the control points are transformed by* T*, that is*

$$T\left(\sum_{i=0}^{n} \mathbf{p}_i R_{i,n}(u)\right) = \sum_{i=0}^{n} T(\mathbf{p}_i) R_{i,n}(u) \tag{1.198}$$

This means that instead of transforming each point $\mathbf{p}(u)$ of a rational Bézier curve, it is sufficient to transform only the control points.

1.2.5 B-Spline Curves

A B-spline curve of degree d is a piecewise polynomial curve defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i N_{i,d}(u) \quad , \quad u \in [a, b] \tag{1.199}$$

where $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ are given control points. The *B-spline basis functions* of degree d, $N_{i,d}(u)$, for $i = 0, 1, \ldots, n$, defined by the *knot sequence* $u_0, u_1, \ldots, u_m$, are defined recursively as

$$N_{i,0}(u) = \begin{cases} 1 & , \text{ if } u \in [u_i, u_{i+1}) \\ 0 & , \text{ otherwise} \end{cases} \tag{1.200}$$

$$N_{i,d}(u) = \frac{u - u_i}{u_{i+d} - u_i} N_{i,d-1}(u) + \frac{u_{i+d+1} - u}{u_{i+d+1} - u_{i+1}} N_{i+1,d-1}(u) \tag{1.201}$$

The knots in the knot sequence satisfy $u_i \leq u_{i+1}$, for $i = 0, \ldots, m-1$, and $u_d = a$, $u_{m-d} = b$. The knots $u_0, u_1, \ldots, u_d$ and $u_{m-d}, u_{m-d+1}, \ldots, u_m$ are called *end knots*, and the knots $u_{d+1}, u_{d+2}, \ldots, u_{m-d-1}$ are called *interior knots*. The knot sequence can contain repeated knots. The number of times a knot value occurs is called the *multiplicity* of the knot. We can define a new sequence $v_0, v_1, \ldots, v_r$, called the *breakpoints*, satisfying $v_0 < v_1 < \cdots < v_r$ and consisting of the distinct values of the interior knots. In the equations defining the B-spline basis functions, Eq. 1.200 and Eq. 1.201, we will get a division of the form $0/0$ for some i, if we use repeated knot values. When this occurs, we replace this division by 0.

As an example of a quadratic planar B-spline curve, consider the curve defined by the knot sequence

$$\begin{aligned} u_0 &= 2 \\ u_1 &= 4 \\ u_2 &= 5 \\ u_3 &= 7 \\ u_4 &= 8 \\ u_5 &= 10 \\ u_6 &= 11 \end{aligned}$$

and by the control points

$$\begin{aligned} \mathbf{p}_0 &= (1, 2) \\ \mathbf{p}_1 &= (3, 5) \\ \mathbf{p}_2 &= (6, 2) \\ \mathbf{p}_3 &= (9, 4) \end{aligned}$$

The B-spline curve is shown in Figure 1.17 on page 40, together with the four control points. The points on the curve corresponding to the end knots and the single breakpoint are also shown. Note that this B-spline curve does not interpolate the first and last control points. The curve is defined on the interval $[u_2, u_4] = [5, 8]$ and consists of two curve segments. We can write the curve as the piecewise polynomial

$$\mathbf{p}(u) = \begin{cases} \frac{1}{6}(7-u)^2\mathbf{p}_0 + \frac{1}{3}(-u^2+12u-34)\mathbf{p}_1 + \frac{1}{6}(u-5)^2\mathbf{p}_2 & , \quad 5 \leq u < 7 \\ \\ \frac{1}{3}(8-u)^2\mathbf{p}_1 + \frac{2}{3}(-u^2+15u-55)\mathbf{p}_2 + \frac{1}{3}(u-7)^2\mathbf{p}_3 & , \quad 7 \leq u \leq 8 \end{cases}$$

As we have seen before, a Bézier curve of degree n has exactly $n+1$ control points. A B-spline curve of degree d can have any number of control points, provided that we specify a sufficient number of knots. In this way, we can get more freedom when defining a B-spline curve, by increasing the number of control points and keep the degree fixed. As can be seen from Eq. 1.201, each B-spline basis function $N_{i,d}(u)$ is defined by the $d+2$ knots $u_i, u_{i+1}, \ldots, u_{i+d+1}$. If we have specified $n+1$ control points $(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n)$ we need for $N_{0,d}(u)$

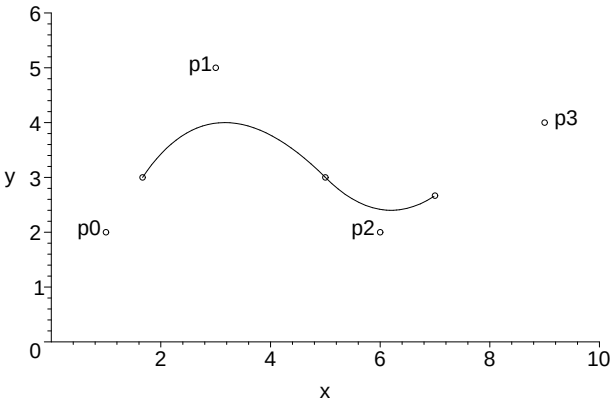

Figure 1.17: Quadratic planar B-spline curve

the knots $u_0, u_1, \ldots, u_{d+1}$. For $N_{n,d}(u)$ we need the knots $u_n, u_{n+1}, \ldots, u_{n+d+1}$. Thus, in total, we need the knots $u_0, u_1, \ldots, u_{n+d+1}$, that is, $n + d + 2$ knots. Since the number of knots is $m + 1$ we get the identity

$$m = n + d + 1 \tag{1.202}$$

Properties of the B-Spline Basis Functions

The B-spline basis functions $N_{i,d}(u)$ satisfy some properties, which in turn give useful properties of the B-spline curves. For the proofs of Theorems 1.17–1.21, see [3].

Theorem 1.17 Positivity

$$N_{i,d}(u) > 0 \ , \ \textit{for } u \in (u_i, u_{i+d+1}) \tag{1.203}$$

Theorem 1.18 Local Support

$$N_{i,d}(u) = 0 \ , \ \textit{for } u \notin (u_i, u_{i+d+1}) \tag{1.204}$$

Theorem 1.19 Piecewise Polynomial
$N_{i,d}(u)$ are piecewise polynomial functions of degree d.

Theorem 1.20 Partition of Unity

$$\sum_{j=r-d}^{r} N_{j,d}(u) = 1 \ , \ \textit{for } u \in [u_r, u_{r+1}) \tag{1.205}$$

Theorem 1.21 Continuity
If the interior knot u_i has multiplicity p_i, then $N_{i,d}(u)$ is C^{d-p_i} at $u = u_i$. $N_{i,d}(u)$ is C^∞ elsewhere.

Properties of B-Spline Curves

B-spline curves have some properties, which make them useful for modeling. For the proofs of Theorems 1.22–1.25, see [3].

Theorem 1.22 Local Control Property
Let a B-spline curve of degree d, defined by the knot sequence $u_0, u_1, \ldots, u_m$, *and with the control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$, *be given by*

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i N_{i,d}(u) \tag{1.206}$$

Then each segment of the curve is determined by $d+1$ *control points.*
If $u \in [u_r, u_{r+1})$ *where* $d \leq r \leq m-d-1$, *then*

$$\mathbf{p}(u) = \sum_{i=r-d}^{r} \mathbf{p}_i N_{i,d}(u) \tag{1.207}$$

That is, to evaluate $\mathbf{p}(u)$ *it is sufficient to evaluate the* $d+1$ *B-spline basis functions* $N_{r-d,d}(u), \ldots, N_{r,d}(u)$, *instead of all* $n+1$ *basis functions.*

Each of the control points can affect at most $d+1$ segments. For example, the control point $\mathbf{p}_0$ affects the first segment only, and $\mathbf{p}_1$ affects the first two segment of the curve. Compare this with a Bézier curve, where the entire curve is affected by each of its control points, which is called global control.

Theorem 1.23 Convex Hull Property
Let a B-spline curve of degree d, defined by the knot sequence $u_0, u_1, \ldots, u_m$, *and with the control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$, *be given by*

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i N_{i,d}(u) \tag{1.208}$$

If $u \in [u_r, u_{r+1})$ *where* $d \leq r \leq m-d-1$, *then*

$$\mathbf{p}(u) \in CH\{\mathbf{p}_{r-d}, \mathbf{p}_{r-d+1}, \ldots, \mathbf{p}_r\} \tag{1.209}$$

That is, each segment of the curve lies in the convex hull of the control points which determine the segment.

As an example of the properties local control and convex hull, consider the planar cubic B-spline curve defined by the knot sequence

$$u_i = \begin{cases} 0 & , \text{ for } i = 0,1,2,3 \\ (i-3) & , \text{ for } i = 4,5,6,7 \\ 5 & , \text{ for } i = 8,9,10,11 \end{cases} \tag{1.210}$$

and by the control points

$$\mathbf{p}_0 \quad = \quad (2,2)$$

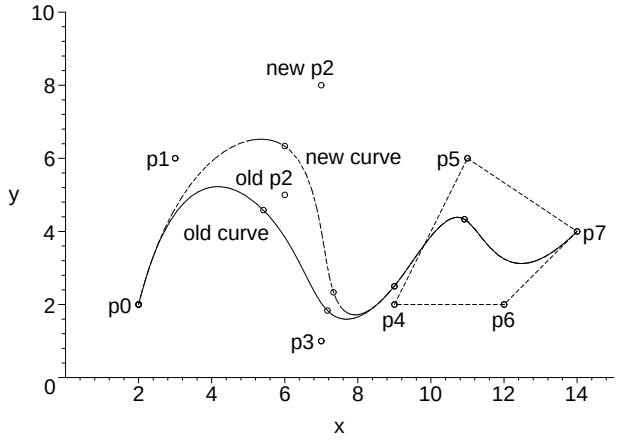

Figure 1.18: Cubic planar B-Spline curves, $\mathbf{p}_2$ is changed

$$\begin{aligned}
\mathbf{p}_1 &= (3,6) \\
\mathbf{p}_2 &= (6,5) \\
\mathbf{p}_3 &= (7,1) \\
\mathbf{p}_4 &= (9,2) \\
\mathbf{p}_5 &= (11,6) \\
\mathbf{p}_6 &= (12,2) \\
\mathbf{p}_7 &= (14,4)
\end{aligned}$$

The B-spline curve is shown in Figure 1.18, together with the eight control points. The effect of changing a control point is also shown. $\mathbf{p}_2$ is changed to $(7,8)$, which gives a new curve. Both curves consist of five segments. The points on the curves corresponding to the breakpoints define the boundaries of the segments. Each segment is determined by $d+1$ control points, where the degree, d, of the curves is 3. For example, the first segment is determined by $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$, and $\mathbf{p}_3$, and the second by $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, and $\mathbf{p}_4$. This means that the change of control point $\mathbf{p}_2$ affects the first three segments, while the last two segments are unaffected. Figure 1.18 also shows the convex hull property for the last segment, that is, the last segment lies in the convex hull of control points $\mathbf{p}_4, \mathbf{p}_5, \mathbf{p}_6$, and $\mathbf{p}_7$.

Theorem 1.24 Continuity
If p_i is the multiplicity of the breakpoint v_i, then $\mathbf{p}(u)$ is C^{d-p_i} at $u = v_i$, and C^∞ elsewhere.

Theorem 1.25 Invariance under Affine Transformations
Let T be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the B-spline curve $\mathbf{p}(u)$ can be written as a B-spline curve where the control points are transformed

by T, that is

$$T\left(\sum_{i=0}^{n} \mathbf{p}_i N_{i,d}(u)\right) = \sum_{i=0}^{n} T(\mathbf{p}_i) N_{i,d}(u) \tag{1.211}$$

This means that instead of transforming each point $\mathbf{p}(u)$ of a B-spline curve, it is sufficient to transform only the control points.

1.2.6 B-Spline Curve Types

By choosing the knot sequence and the control points in certain ways, we can get various types of B-spline curves.

Open B-Spline Curves

An *open B-spline curve* is a B-spline curve where the end knots satisfy

$$u_0 = u_1 = \cdots = u_d$$
$$u_{m-d} = u_{m-d+1} = \cdots = u_m$$

It can be shown that for this choice of the knot sequence, we get a B-spline curve which interpolates the end control points, that is

$$\mathbf{p}(u_d) = \mathbf{p}_0 \tag{1.212}$$
$$\mathbf{p}(u_{m-d}) = \mathbf{p}_n \tag{1.213}$$

The B-spline curves in Figure 1.18 on page 42 are open B-spline curves. Note that the end control points are interpolated.

Uniform B-Spline Curves

A B-spline curve is said to be *uniform* whenever its knots are equally spaced, and *non-uniform* otherwise.

Open Uniform B-Spline Curves

An *open uniform B-spline curve* is an open B-spline curve where the interior knots are equally spaced. The B-spline curves in Figure 1.18 on page 42 are open uniform B-spline curves.

Periodic B-Spline Curves

A *periodic B-spline curve* of degree d with $n+1$ control points is a B-spline curve where the first $n+1$ knots, $u_0 \leq u_1 \leq \cdots \leq u_n$, are chosen arbitrarily. The remaining $d+1$ knots are given by

$$u_{n+i} = u_{n+i-1} + (u_i - u_{i-1}) \ , \text{ for } i = 1, \ldots, d+1 \tag{1.214}$$

The knot sequence of this form is called a *periodic knot sequence*. A uniform B-spline curve is a special case of a periodic B-spline curve.

Closed Periodic B-Spline Curves

A *closed periodic B-spline curve* of degree d with $n+1$ control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$ is obtained by choosing a periodic knot sequence with $n + 2d + 2$ knots, and then introducing d additional control points given by

$$\mathbf{p}_{n+i} = \mathbf{p}_{i-1} \ , \ \text{for } i = 1, 2, \ldots, d \tag{1.215}$$

As an example of a closed periodic B-spline curve, consider the planar cubic curve defined by the periodic (and uniform) knot sequence

$$u_i = i \ , \ \text{for } i = 0, 1, \ldots, 12 \tag{1.216}$$

and by the control points

$$\begin{aligned} \mathbf{p}_0 &= (1, 2) \\ \mathbf{p}_1 &= (2, 4) \\ \mathbf{p}_2 &= (4, 5) \\ \mathbf{p}_3 &= (5, 4) \\ \mathbf{p}_4 &= (5, 2) \\ \mathbf{p}_5 &= (4, 1) \end{aligned}$$

We add $d = 3$ control points, $\mathbf{p}_6 = \mathbf{p}_0$, $\mathbf{p}_7 = \mathbf{p}_1$, and $\mathbf{p}_8 = \mathbf{p}_2$. The curve is shown in Figure 1.19 on page 45, together with the original control points. Note that the curve is closed, and does not interpolate any of the control points.

1.2.7 NURBS Curves

The B-spline curves described in Section 1.2.5 (often called *integral* B-spline curves) can be generalized to get *NURBS* curves. NURBS is an abbreviation for *Non-Uniform Rational B-Spline*. A NURBS curve of degree d is a piecewise rational curve defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i R_{i,d}(u) \ , \ u \in [a, b] \tag{1.217}$$

where $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ are given control points. The rational B-spline basis functions $R_{i,d}(u)$ are defined by

$$R_{i,d}(u) = \begin{cases} \frac{w_i N_{i,d}(u)}{\sum_{j=0}^{n} w_j N_{j,d}(u)} & , \ \text{if } w_i \neq 0 \\ \frac{N_{i,d}(u)}{\sum_{j=0}^{n} w_j N_{j,d}(u)} & , \ \text{if } w_i = 0 \end{cases} \tag{1.218}$$

where $N_{i,d}(u)$ are the basis functions for B-splines, defined by Eq. 1.200 and Eq. 1.201, and w_i is a scalar weight for control point $\mathbf{p}_i$. At least one of the weights $w_0, w_1, \ldots, w_n$ is non-zero. As for integral B-spline curves, we also need a knot sequence with $m + 1$ knots $(u_0, u_1, \ldots, u_m)$. If all weights are chosen

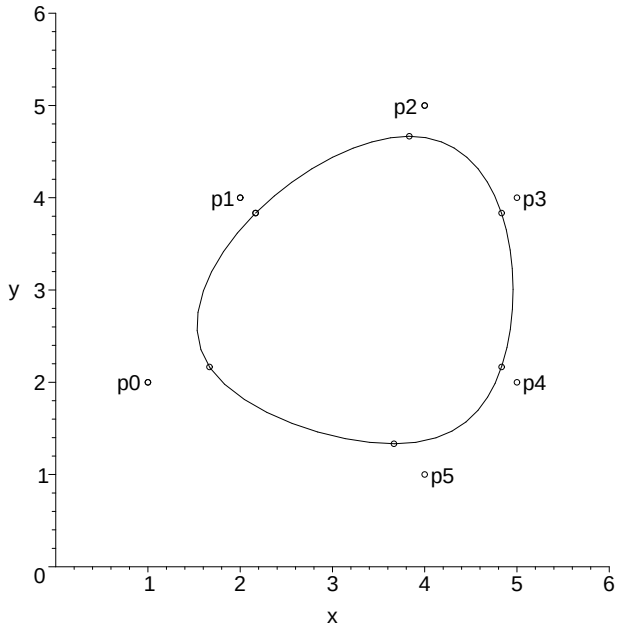

Figure 1.19: Closed periodic cubic planar B-spline curve

equal and non-zero, we get integral B-spline curves as special cases of NURBS curves.

As an example of a quadratic NURBS curve, consider the curve defined by the control points

$$\begin{aligned}
\mathbf{p}_0 &= (1,0) \\
\mathbf{p}_1 &= (1,1) \\
\mathbf{p}_2 &= (-1,1) \\
\mathbf{p}_3 &= (-1,0) \\
\mathbf{p}_4 &= (-1,-1) \\
\mathbf{p}_5 &= (1,-1) \\
\mathbf{p}_6 &= (1,0)
\end{aligned}$$

and by the weights

$$w_0, \ldots, w_6 = 1, \frac{1}{2}, \frac{1}{2}, 1, \frac{1}{2}, \frac{1}{2}, 1$$

and by the knot sequence

$$u_0, \ldots, u_9 = 0, 0, 0, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, 1, 1, 1$$

The NURBS curve is defined on the interval $[u_2, u_7] = [0, 1]$. As shown in Figure 1.20 on page 46, the curve is the unit circle. This NURBS curve is useful when creating surfaces of revolution, described in Section 1.3.4.

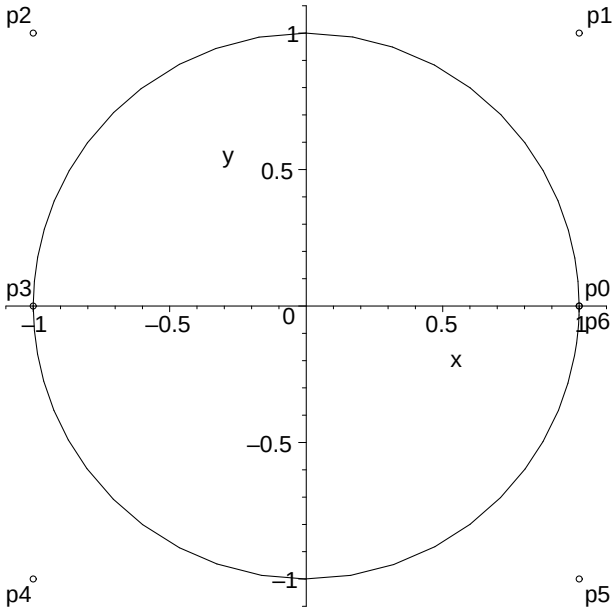

Figure 1.20: Unit circle as a quadratic planar NURBS curve

Properties of NURBS Curves

Theorems 1.26–1.29 state some useful properties of NURBS curves.

Theorem 1.26 Local Control Property
Let a NURBS curve of degree d, defined by the knot sequence $u_0, u_1, \ldots, u_m$, and with the control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$, with the corresponding weights w_i, be given by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i R_{i,d}(u) \tag{1.219}$$

Then each segment of the curve is determined by $d+1$ control points.
If $u \in [u_r, u_{r+1})$ where $d \leq r \leq m-d-1$, then

$$\mathbf{p}(u) = \sum_{i=r-d}^{r} \mathbf{p}_i R_{i,d}(u) \tag{1.220}$$

That is, to evaluate $\mathbf{p}(u)$ it is sufficient to evaluate the $d+1$ rational B-spline basis functions $R_{r-d,d}(u), \ldots, R_{r,d}(u)$, instead of all $n+1$ basis functions.

Theorem 1.27 Convex Hull Property
Let a NURBS curve of degree d, defined by the knot sequence $u_0, u_1, \ldots, u_m$, and with the control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$, with the corresponding weights w_i, be given by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i R_{i,d}(u) \tag{1.221}$$

If $u \in [u_r, u_{r+1})$ where $d \leq r \leq m-d-1$, then

$$\mathbf{p}(u) \in CH\{\mathbf{p}_{r-d}, \mathbf{p}_{r-d+1}, \ldots, \mathbf{p}_r\} \tag{1.222}$$

That is, each segment of the curve lies in the convex hull of the control points which determine the segment.

Theorem 1.28 Continuity
If p_i is the multiplicity of the breakpoint v_i, then $\mathbf{p}(u)$ is C^{d-p_i} at $u = v_i$, and C^∞ elsewhere.

Theorem 1.29 Invariance under Affine Transformations
Let T be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the NURBS curve $\mathbf{p}(u)$ can be written as a NURBS curve where the control points are transformed by T, that is

$$T\left(\sum_{i=0}^{n} \mathbf{p}_i R_{i,d}(u)\right) = \sum_{i=0}^{n} T(\mathbf{p}_i) R_{i,d}(u) \tag{1.223}$$

This means that instead of transforming each point $\mathbf{p}(u)$ of a NURBS curve, it is sufficient to transform only the control points.

1.2.8 Linear Approximations of Curves

We can approximate a curve by piecewise linear segments. For example, suppose a Bézier curve of degree n is given by

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(u) \ , \ u \in [0,1] \tag{1.224}$$

We can choose $N+1$ values of the parameter u as

$$u_i = \frac{i}{N} \ , \ i = 0, 1, \ldots, N \tag{1.225}$$

and evaluate $\mathbf{p}(u)$ at $u = u_i$. We can then connect the points $\mathbf{p}(u_i)$ and $\mathbf{p}(u_{i+1})$ for $i = 0, 1, \ldots, N-1$ by line segments. For $N+1$ points on the curve we get N line segments, and each line segment L_i can be parametrized as

$$L_i(t) = \mathbf{p}(u_i)(1-t) + \mathbf{p}(u_{i+1})t \ , \ t \in [0,1] \tag{1.226}$$

As an example, consider the cubic Bézier curve in Figure 1.8 on page 15. If we choose $N = 5$, we get u_i as

$$u_i = 0, \frac{1}{5}, \frac{2}{5}, \ldots, 1 \tag{1.227}$$

Figure 1.21 on page 48 shows the original curve and the approximation of the curve by 5 piecewise linear segments. In order to get a good approximation of the curve, we have to choose N sufficiently large.

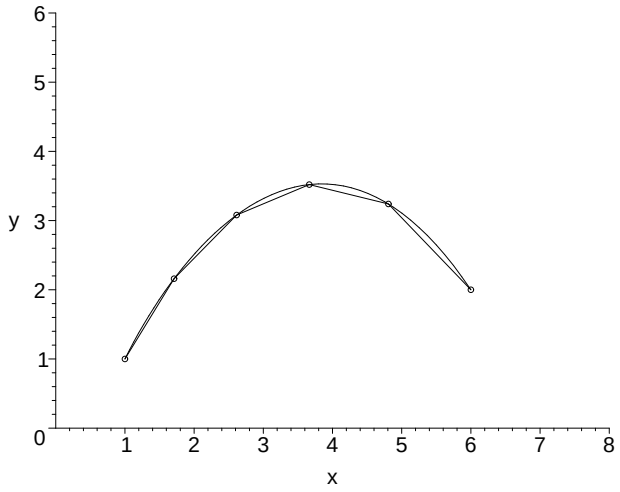

Figure 1.21: Linear approximation of a cubic Bézier curve

We could also use a non-uniform spacing between the parameter values u_i. For example, the curvature of the curve could be used as determining the spacing in such a way that when the curvature is small, we use a larger distance between successive values of u_i, and when the curvature is large, we use a smaller distance.

Linear approximation of a curve is often used when the curve is *rendered*, that is, plotted on a computer screen, for example. Rendering of a Bézier curve is described in more detail in Section 3.2.3.

1.3 Representation of 3D Surfaces

This section shows how surfaces in 3 dimensions can be expressed by parametric equations. Common special cases, as parametric bicubic surfaces, Bézier surfaces, and B-spline surfaces are described. Approximating a surface by a mesh of polygons will also be discussed.

1.3.1 Explicit and Implicit Equations

For a point (x, y, z) on a surface, we can express the z coordinate as a real-valued function of x and y as

$$z = f(x, y) \tag{1.228}$$

where $a \leq x \leq b$ and $c \leq y \leq d$. This formulation has the drawback that a surface with multiple values of z for the same value of the pair of x and y coordinates, has to be broken into multiple surfaces, such that for each surface we get a single value for z for each pair of the x and y coordinates. For example,

a sphere with radius r centered at the origin can be divided into two semi spheres S_1 and S_2 as:

$$\begin{array}{rrcll} S_1: & z & = & \sqrt{r^2-(x^2+y^2)} & , \quad x^2+y^2 \le r^2 \\ S_2: & z & = & -\sqrt{r^2-(x^2+y^2)} & , \quad x^2+y^2 \le r^2 \end{array} \tag{1.229}$$

Note that we need the constraint $x^2 + y^2 \le r^2$.

As another example, the general second-degree implicit equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + Px + Qy + Rz + T = 0 \tag{1.230}$$

can represent planes, cones, cylinders, ellipsoids, hyperboloids, and paraboloids, depending on the coefficients. For the classification, see for example [5]. As an example, consider the equation

$$\begin{aligned} \frac{7}{48}x^2 + \frac{31}{144}y^2 + \frac{1}{4}z^2 - \frac{5\sqrt{3}}{72}xy + \left(\frac{5\sqrt{3}}{24} - \frac{7}{6}\right)x \\ + \left(\frac{5\sqrt{3}}{18} - \frac{31}{24}\right)y - z + \frac{205}{48} - \frac{5\sqrt{3}}{6} &= 0 \end{aligned} \tag{1.231}$$

By the coordinate transformation

$$\begin{aligned} x' &= \frac{\sqrt{3}}{2}(x-4) + \frac{1}{2}(y-3) \\ y' &= -\frac{1}{2}(x-4) + \frac{\sqrt{3}}{2}(y-3) \\ z' &= z-2 \end{aligned} \tag{1.232}$$

which represents a counterclockwise rotation by $\pi/6$ around the z-axis and a translation of the origin to $(4, 3, 2)$, Eq. 1.231 can be written in the new coordinates x', y', and z' as

$$\frac{x'^2}{a^2} + \frac{y'^2}{b^2} + \frac{z'^2}{c^2} - 1 = 0 \tag{1.233}$$

where $a = 3$, $b = 2$, and $c = 2$. This is the equation for an ellipsoid. See Figure 1.22 on page 50.

1.3.2 Parametric Bicubic Surfaces

The coordinates of a point on a surface can be expressed by three parametric functions, one for each of the coordinates x, y, and z. Each function takes two parameters, u and w.

$$\begin{aligned} x &= x(u,w) \\ y &= y(u,w) \\ z &= z(u,w) \end{aligned} \tag{1.234}$$

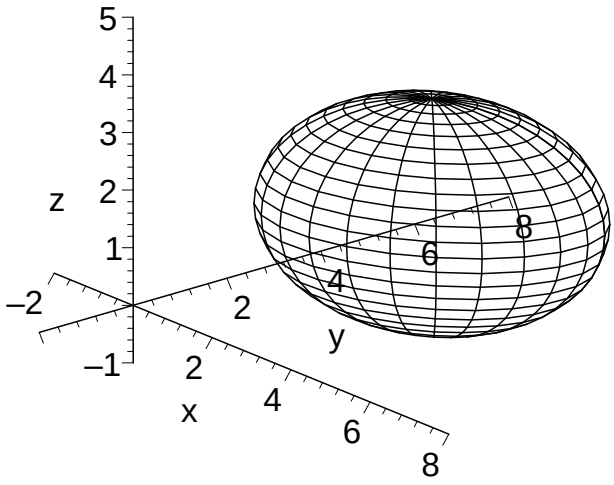

Figure 1.22: Rotated and translated ellipsoid

A point on the surface is given by the row vector

$$\mathbf{p} = \begin{bmatrix} x(u,w) & y(u,w) & z(u,w) \end{bmatrix} \tag{1.235}$$

A *patch* is a curve-bounded surface where the functions are continuous and the parameters u and w are constrained. Usually, u and w are constrained to the interval $[0,1]$. If one of the parameters, for example w, is fixed, we get a curve lying in the patch when we vary u. By repeating the process with different values of w, and then doing the same for u, we get a net of two one-parameter families of curves, such that one curve of each family passes through a point on the patch.

Algebraic Form

A *bicubic patch* is a patch where the coordinate functions are cubic polynomials in u and w. The algebraic form of the bicubic patch is given by

$$\mathbf{p}(u,w) = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{a}_{ij} u^i w^j \tag{1.236}$$

where $u, w \in [0,1]$ and the algebraic coefficients $\mathbf{a}_{ij}$ are given in vector form as

$$\mathbf{a}_{ij} = \begin{bmatrix} a_{ij_x} & a_{ij_y} & a_{ij_z} \end{bmatrix} \tag{1.237}$$

For a bicubic patch, there are 16 algebraic coefficients which gives 48 degrees of freedom. The x coordinate of a point on the patch can be written

$$x = U A_x W^T \tag{1.238}$$

where U and W are 1×4 matrices, and A_x is a 4×4 matrix consisting of the x components of the algebraic coefficients

$$\begin{aligned} U &= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \\ W &= \begin{bmatrix} w^3 & w^2 & w & 1 \end{bmatrix} \\ A_x &= \begin{bmatrix} a_{33_x} & a_{32_x} & a_{31_x} & a_{30_x} \\ a_{23_x} & a_{22_x} & a_{21_x} & a_{20_x} \\ a_{13_x} & a_{12_x} & a_{11_x} & a_{10_x} \\ a_{03_x} & a_{02_x} & a_{01_x} & a_{00_x} \end{bmatrix} \end{aligned} \tag{1.239}$$

Similar expressions can be written for the coordinates y and z.

1.3.3 Rational Bézier Surfaces

A rational Bézier surface with control points $\mathbf{p}_{i,j}$, and corresponding weights $w_{i,j}$, where $0 \leq i \leq n$ and $0 \leq j \leq p$, is for $(u,t) \in [0,1] \times [0,1]$ defined by

$$\mathbf{p}(u,t) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} \mathbf{p}_{i,j} B_{i,n}(u) B_{j,p}(t)}{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} B_{i,n}(u) B_{j,p}(t)} \tag{1.240}$$

where $B_{i,n}(u)$ and $B_{j,p}(t)$ are the Bernstein polynomials of degrees n and p, in the variables u and t, respectively. If $w_{i,j} = 0$, we replace $w_{i,j}\mathbf{p}_{i,j}$ by $\mathbf{p}_{i,j}$. The $(n+1) \cdot (p+1)$ control points form a *control point polyhedron.* If all weights $w_{i,j}$ are equal and non-zero, we get an integral Bézier surface as a special case of a rational Bézier surface, and Eq.1.240 simplifies to

$$\mathbf{p}(u,t) = \sum_{i=0}^{n} \sum_{j=0}^{p} \mathbf{p}_{i,j} B_{i,n}(u) B_{j,p}(t) \tag{1.241}$$

The coordinate curves of the surface are rational Bézier curves, that is, if we for example let $u = u_k$ be fixed, we get the rational Bézier curve $\mathbf{p}(t) = \mathbf{p}(u_k, t)$, where $t \in [0,1]$. Figure 1.23 on page 52 shows an example of a rational bicubic Bézier surface.

Properties of Rational Bézier Surfaces

Rational Bézier surfaces have some useful properties, similar to the properties of rational Bézier curves. The properties are stated in Theorems 1.30–1.32.

Theorem 1.30 End Point Interpolation Property
A rational Bézier surface $\mathbf{p}(u,t)$ *with control points* $\mathbf{p}_{0,0}, \ldots, \mathbf{p}_{n,p}$*, and weights* $w_{0,0}, \ldots, w_{n,p}$*, where* $w_{0,0}, w_{n,0}, w_{0,p}, w_{n,p}$*, are non-zero, satisfies*

$$\begin{aligned} \mathbf{p}(0,0) &= \mathbf{p}_{0,0} \\ \mathbf{p}(1,0) &= \mathbf{p}_{n,0} \\ \mathbf{p}(0,1) &= \mathbf{p}_{0,p} \\ \mathbf{p}(1,1) &= \mathbf{p}_{n,p} \end{aligned} \tag{1.242}$$

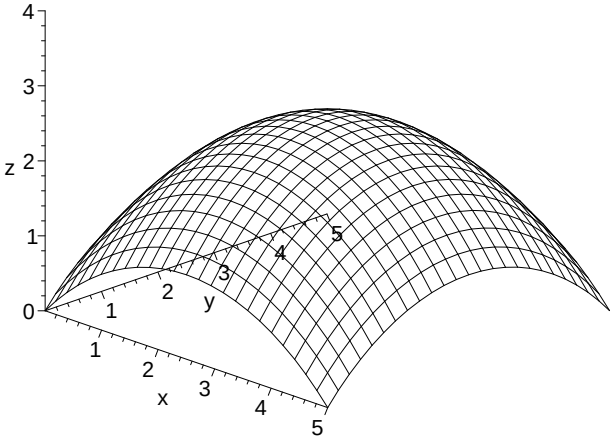

Figure 1.23: Rational bicubic Bézier surface

Theorem 1.31 Convex Hull Property
Every point on a rational Bézier surface lies inside the convex hull of its control points, provided that all weights $w_{i,j} > 0$. That is

$$\mathbf{p}(u,t) \in CH\{\mathbf{p}_{0,0}, \ldots, \mathbf{p}_{n,p}\} \ , \ \textit{for } (u,t) \in [0,1] \times [0,1] \tag{1.243}$$

Theorem 1.32 Invariance under Affine Transformations
Let T be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the rational Bézier surface $\mathbf{p}(u,t)$ can be written as a rational Bézier surface where the control points are transformed by T, that is

$$T(\mathbf{p}(u,t)) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} T(\mathbf{p}_{i,j}) B_{i,n}(u) B_{j,p}(t)}{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} B_{i,n}(u) B_{j,p}(t)} \tag{1.244}$$

This means that instead of transforming each point $\mathbf{p}(u,t)$ of a rational Bézier surface, it is sufficient to transform only the control points.

1.3.4 NURBS Surfaces

A NURBS surface with control points $\mathbf{p}_{i,j}$, and corresponding weights $w_{i,j}$, where $0 \le i \le n$ and $0 \le j \le p$, is for $(u,t) \in [u_d, u_{m-d}] \times [t_e, t_{q-e}]$ defined by

$$\mathbf{p}(u,t) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} \mathbf{p}_{i,j} N_{i,d}(u) N_{j,e}(t)}{\sum_{i=0}^{n} \sum_{j=0}^{p} w_{i,j} N_{i,d}(u) N_{j,e}(t)} \tag{1.245}$$

where $N_{i,d}(u)$ and $N_{j,e}(t)$ are the B-spline basis functions of degrees d and e, with the knot sequences $u_0, \ldots, u_m$ and $t_0, \ldots, t_q$, respectively. If $w_{i,j} = 0$, we replace $w_{i,j}\mathbf{p}_{i,j}$ by $\mathbf{p}_{i,j}$. The $(n+1) \cdot (p+1)$ control points form the control point polyhedron. If all weights $w_{i,j}$ are equal and non-zero, we get an integral

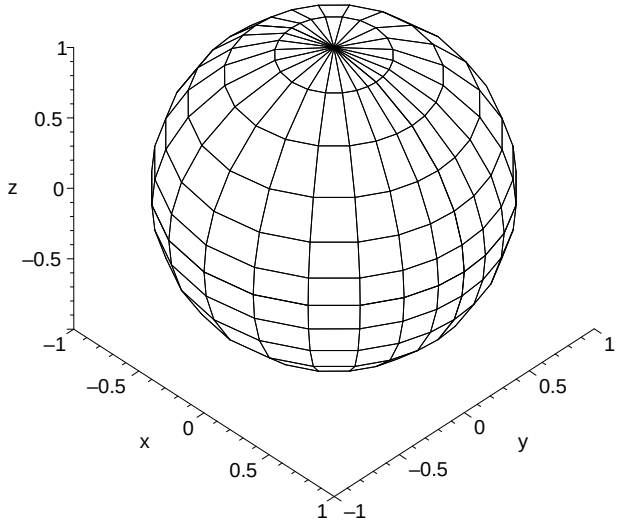

Figure 1.24: Unit sphere created as a NURBS surface

B-spline surface as a special case of a NURBS surface, and Eq.1.245 simplifies to

$$\mathbf{p}(u,t) = \sum_{i=0}^{n} \sum_{j=0}^{p} \mathbf{p}_{i,j} N_{i,d}(u) N_{j,e}(t) \tag{1.246}$$

The coordinate curves of the surface are NURBS curves, that is, if we for example let $u = u_k$ be fixed, we get the NURBS curve $\mathbf{p}(t) = \mathbf{p}(u_k, t)$, where $t \in [t_e, t_{q-e}]$.

Figure 1.24 shows an example of a NURBS surface, created as a surface of revolution. A unit circle is first created as a NURBS curve with 7 control points in the xy-plane (as in Figure 1.20 on page 46). A second unit circle is then created as a NURBS curve in the xz-plane. The control points of these two curves are then combined into a mesh of control points used for the NURBS surface. The curve in the xz-plane is thus rotated about the z-axis. As can be seen in Figure 1.24, we get the unit sphere centered at the origin.

Figure 1.25 on page 54 shows another example of a NURBS surface, also created as a surface of revolution. A unit circle is first created as a NURBS curve in the xz-plane. The curve is then scaled to a circle of radius $\frac{1}{4}$ and then translated $\frac{1}{2}$ units in the x-direction. The circle is then rotated about the z-axis, and the resulting surface is a torus.

Properties of NURBS Surfaces

NURBS surfaces have some useful properties, stated in Theorems 1.33–1.35.

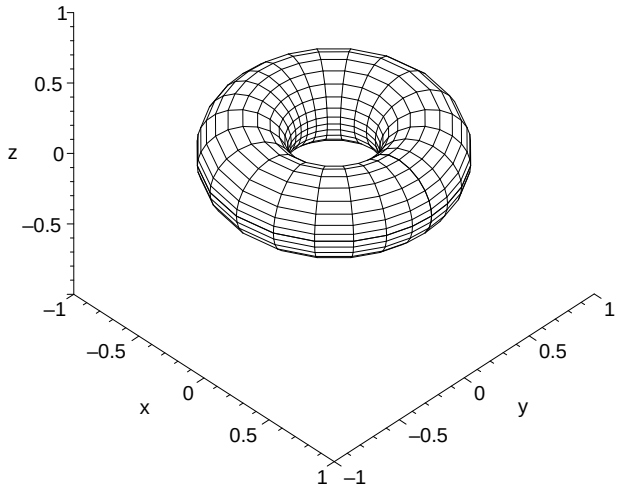

Figure 1.25: Torus created as a NURBS surface

Theorem 1.33 Local Control Property
Let a NURBS surface be given by

$$\mathbf{p}(u,t) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}\mathbf{p}_{i,j}N_{i,d}(u)N_{j,e}(t)}{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}N_{i,d}(u)N_{j,e}(t)} \tag{1.247}$$

Then each segment of the surface is determined by a mesh of $(d+1)\cdot(e+1)$ *control points. If* $u \in [u_r, u_{r+1})$ *and* $t \in [t_s, t_{s+1})$*, for* $d \le r \le m-d-1$ *and* $e \le s \le q-e-1$*, then*

$$\mathbf{p}(u,t) = \frac{\sum_{i=r-d}^{r}\sum_{j=s-e}^{s} w_{i,j}\mathbf{p}_{i,j}N_{i,d}(u)N_{j,e}(t)}{\sum_{i=r-d}^{r}\sum_{j=s-e}^{s} w_{i,j}N_{i,d}(u)N_{j,e}(t)} \tag{1.248}$$

That is, to evaluate $\mathbf{p}(u,t)$ *it is sufficient to evaluate the* $(d+1)$ *basis functions* $N_{r-d,d}(u),\ldots,N_{r,d}(u)$*, and the* $(e+1)$ *basis functions* $N_{s-e,e}(t),\ldots,N_{s,e}(t)$ *instead of all* $(n+1)\cdot(p+1)$ *basis functions.*

Theorem 1.34 Convex Hull Property
Let a NURBS surface be given by

$$\mathbf{p}(u,t) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}\mathbf{p}_{i,j}N_{i,d}(u)N_{j,e}(t)}{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}N_{i,d}(u)N_{j,e}(t)} \tag{1.249}$$

If $u \in [u_r, u_{r+1})$ *and* $t \in [t_s, t_{s+1})$*, for* $d \le r \le m-d-1$ *and* $e \le s \le q-e-1$*, and all weights* $w_{i,j} > 0$*, then*

$$\mathbf{p}(u,t) \in CH\{\mathbf{p}_{r-d,s-e},\ldots,\mathbf{p}_{r,s}\} \tag{1.250}$$

That is, each segment of the surface lies in the convex hull of the control points which determine the segment.

Theorem 1.35 Invariance under Affine Transformations
Let T be an affine transformation, for example a rotation, translation, scaling, reflection, or a combination of these. The transformation of the NURBS surface $\mathbf{p}(u,t)$ can be written as a NURBS surface where the control points are transformed by T, that is

$$T(\mathbf{p}(u,t)) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}T(\mathbf{p}_{i,j})N_{i,d}(u)N_{j,e}(t)}{\sum_{i=0}^{n}\sum_{j=0}^{p} w_{i,j}N_{i,d}(u)N_{j,e}(t)} \tag{1.251}$$

This means that instead of transforming each point $\mathbf{p}(u,t)$ of a NURBS surface, it is sufficient to transform only the control points.

1.3.5 Approximating Surfaces by Polygon Meshes

In Section 1.2.8 on page 47 we approximated a curve by piecewise linear segments. In a similar way, we can approximate a surface by *polygon meshes*. Suppose we have defined the surface as a parametric surface, $\mathbf{p}(u,t)$, in the parameters u and t. For example, let the surface be given as the Bézier surface

$$\mathbf{p}(u,t) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{p}_{i,j}B_{i,n}(u)B_{j,m}(t) \ , \ (u,t) \in [0,1]\times[0,1] \tag{1.252}$$

We choose $N+1$ values of parameter u, and $M+1$ values of parameter t as

$$\begin{aligned} u_i &= \frac{i}{N} \ , \ i = 0,1,\ldots,N \\ t_j &= \frac{j}{M} \ , \ j = 0,1,\ldots,M \end{aligned} \tag{1.253}$$

and evaluate $\mathbf{p}(u,t)$ at $u = u_i$ and $t = t_j$. We can then form $N \cdot M$ polygons, where each polygon $P_{i,j}$ is defined by the four points $\mathbf{p}(u_i,t_j)$, $\mathbf{p}(u_{i+1},t_j)$, $\mathbf{p}(u_{i+1},t_{j+1})$, and $\mathbf{p}(u_i,t_{j+1})$, for $i = 0,1,\ldots,N-1$ and $j = 0,1,\ldots,M-1$. Figure 1.26 on page 56 shows a surface we want to approximate. We have chosen $N = M = 3$, and we have evaluated $\mathbf{p}(u,t)$ at $(N+1)\cdot(M+1) = 16$ points. This gives us $N \cdot M = 9$ polygons. The polygon mesh is shown in Figure 1.27 on page 56. The polygons in the mesh is in general not planar. A simple way to get planar polygons is to divide each polygon into triangles. We can do this by computing a new point $\mathbf{q}_{i,j}$ (on the original surface) for polygon $P_{i,j}$ as

$$\mathbf{q}_{i,j} = \mathbf{p}(\frac{u_i + u_{i+1}}{2}, \frac{t_j + t_{j+1}}{2}) \tag{1.254}$$

Then we create four triangles with this point and the four points of the polygon, as in Figure 1.28 on page 57, which shows that polygon $P_{1,1}$ has been triangulated. The advantage of using triangles as polygons is that a triangle is planar and convex. Since it is planar, we can compute a normal vector to the plane defined by the triangle. This can be used for hidden surface removal, described in Section 2.5. It can also be used for adding realism to the image of the surface, by computing a color for each triangle based on the orientation of the triangle

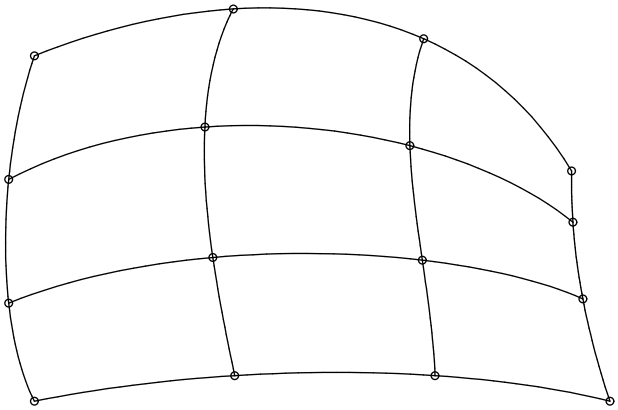

Figure 1.26: Surface to be approximated

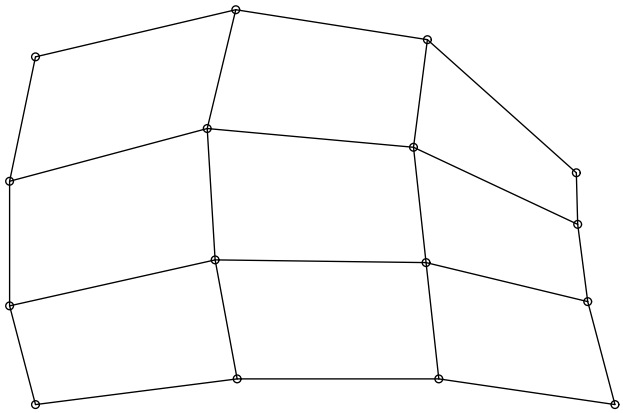

Figure 1.27: Surface approximated by a polygon mesh

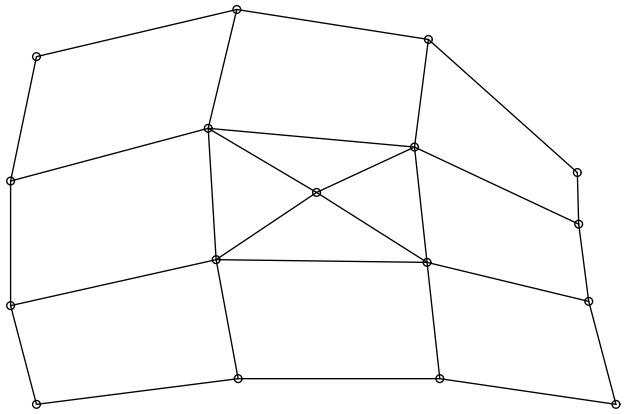

Figure 1.28: Polygon mesh with polygon $P_{1,1}$ triangulated

in relation to various light sources. A convex polygon is desirable when rendering, because there exist efficient algorithms for rendering convex polygons. The rendering of non-convex polygons is more complicated.

1.4 Representation of 3D Solids

This section briefly describes how solids in 3 dimensions can be represented. Parametric tricubic solids can be used to model the interior of a solid object, and is described in Section 1.4.1. Usually, in computer graphics, we are only interested of the boundary surface of a solid object, since this is what we would like to show as an image.

1.4.1 Parametric Tricubic Solids

The coordinates for a point in a solid can be expressed by three parametric functions, one for each of the coordinates x, y, and z. Each function takes three parameters, u, v, and w.

$$\begin{aligned} x &= x(u,v,w) \\ y &= y(u,v,w) \\ z &= z(u,v,w) \end{aligned} \tag{1.255}$$

A point in the solid is given by the row vector

$$\mathbf{p} = \begin{bmatrix} x(u,v,w) & y(u,v,w) & z(u,v,w) \end{bmatrix} \tag{1.256}$$

Algebraic Form

A tricubic solid is given in algebraic form as

$$\mathbf{p}(u,v,w) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} \mathbf{a}_{ijk} u^i v^j w^k \ , \ u,v,w \in [0,1] \tag{1.257}$$

where the algebraic coefficients $\mathbf{a}_{ijk}$ are given in vector form as

$$\mathbf{a}_{ijk} = \begin{bmatrix} a_{ijk_x} & a_{ijk_y} & a_{ijk_z} \end{bmatrix} \tag{1.258}$$

For a tricubic solid, there are $4^3 = 64$ algebraic coefficients which gives $64 \cdot 3 = 192$ degrees of freedom. If one of the parameters, for example w, is fixed, we get a surface lying in the solid when we vary u and v. The tricubic solid is bounded by bicubic patches. Tricubic solids are described in more detail in [4].

1.4.2 Solids Bounded by Surfaces

In the previous sections, we have seen surfaces which we can interpret as solids. The ellipsoid in Figure 1.22, the sphere in Figure 1.24, and the torus in Figure 1.25 are some examples.

Chapter 2

Visualizing 3D Objects as 2D Projections

2.1 Introduction

This chapter describes the mathematics involved when creating a 2-dimensional image from 3-dimensional objects. Transformations in 3D are described in Section 2.2, followed by projections in Section 2.3. Section 2.4 describes methods for specifying what in the 3D space should be seen.

2.2 Transformations

Transformations can be used for changing position, orientation, or shape of an object. They are also useful for projections, described in Section 2.3. The term *rigid-body transformation* is often used for a transformation which does not change the shape of the object.

2.2.1 Translation

A translation is a transformation which moves every point of an object a given distance in a given direction. The translation can be written by the matrix expression

$$\mathbf{p}' = \mathbf{p} + \mathbf{t} \tag{2.1}$$

where $\mathbf{p}'$ is the translated point, $\mathbf{p}$ is the point to be translated, and $\mathbf{t}$ the vector with the given distance and direction.

2.2.2 Rotation about a Coordinate Axis

Before deriving the expressions for rotations, we have to define the direction of positive rotation. We define the direction of positive rotation about a coordinate axis according to Table 2.1 on page 60. For a right-handed coordinate

axis of rotation	direction of positive rotation
x	$y \rightarrow z$
y	$z \rightarrow x$
z	$x \rightarrow y$

Table 2.1: Direction of positive rotation

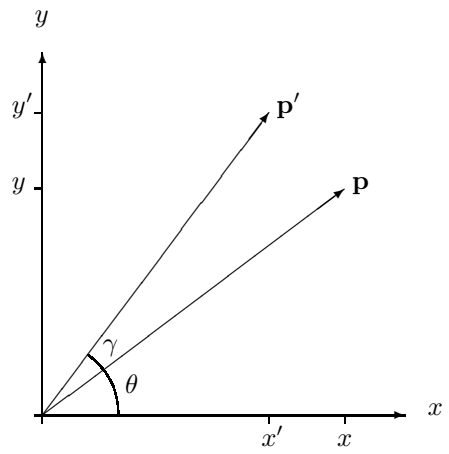

Figure 2.1: Positive rotation about the z-axis

system, the direction of positive rotation corresponds to a counterclockwise rotation when looking at the origin, along the axis of rotation. For a left-handed coordinate system, positive rotation corresponds to a clockwise rotation. The rotation of a point about the z-axis can be derived as follows. See Figure 2.1. The z-axis is directed out of the page for a right-handed system, and directed into the page for a left-handed system. In Figure 2.1 on the rotation is positive, and when looking at the origin along the z-axis, the rotation is counterclockwise for a right-handed system, and clockwise for a left-handed system. We get the following expressions for the coordinates of $\mathbf{p}$ and $\mathbf{p}'$

$$x = |\mathbf{p}| \cos\theta \tag{2.2}$$
$$y = |\mathbf{p}| \sin\theta \tag{2.3}$$
$$x' = |\mathbf{p}'| \cos(\theta + \gamma) \tag{2.4}$$
$$y' = |\mathbf{p}'| \sin(\theta + \gamma) \tag{2.5}$$

But $|\mathbf{p}| = |\mathbf{p}'|$, and using the addition rules of the sine and cosine functions we can write Eq. 2.4 and Eq. 2.5 as

$$x' = |\mathbf{p}|(\cos\theta\cos\gamma - \sin\theta\sin\gamma) \tag{2.6}$$

$$y' = |\mathbf{p}|(\sin\theta\cos\gamma + \cos\theta\sin\gamma) \tag{2.7}$$

Using Eq. 2.2 and Eq. 2.3 in the above equations gives

$$x' = x\cos\gamma - y\sin\gamma \tag{2.8}$$
$$y' = y\cos\gamma + x\sin\gamma \tag{2.9}$$

Since we rotate about the z-axis in a plane parallel to the xy-plane, $z' = z$, and we can write the coordinates of $\mathbf{p}'$ with the matrix expression

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

or more compact as

$$\mathbf{p}' = \mathbf{p}Rot_{z,\gamma} \tag{2.11}$$

In a similar way, we can derive the matrices $Rot_{x,\alpha}$ and $Rot_{y,\beta}$ for rotations about the x-axis and the y-axis respectively, as

$$Rot_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \tag{2.12}$$

$$Rot_{y,\beta} = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \tag{2.13}$$

2.2.3 Scaling about the Origin

An object can be scaled by multiplying the coordinates of every point by the scale factors s_x, s_y, and s_z, all non-zero. The scaling transformation can be written

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{2.14}$$

If $s_x = s_y = s_z$, we get *uniform scaling*.

2.2.4 Reflections

By using the scaling transformation and choosing combinations of ± 1 for s_x, s_y, and s_z, we can reflect a point $\mathbf{p}$ in a coordinate plane, in a coordinate axis, or in the origin. See Figure 2.2 on page 62. $\mathbf{p}'$ is the reflection of $\mathbf{p}$ in the xy-plane, $\mathbf{p}''$ is the reflection of $\mathbf{p}$ in the z-axis, and $\mathbf{p}'''$ is the reflection of $\mathbf{p}$ in the origin. The reflected points $\mathbf{p}'$, $\mathbf{p}''$, and $\mathbf{p}'''$ are given by the matrix expressions

$$\mathbf{p}' = \mathbf{p}R_{xy} = \mathbf{p}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{2.15}$$

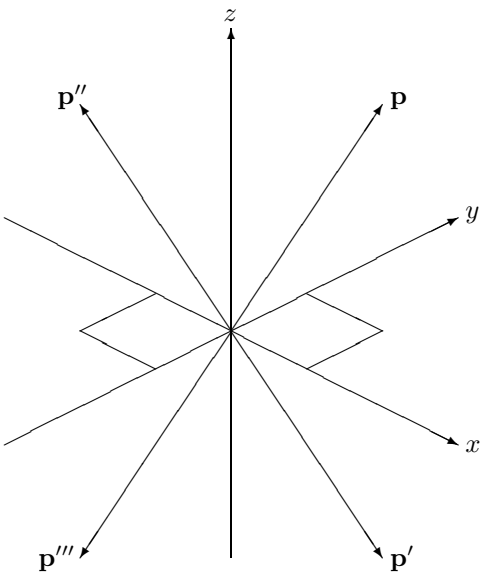

Figure 2.2: Reflections in the xy-plane, in the z-axis, and in the origin

$$\mathbf{p}'' = \mathbf{p}R_z = \mathbf{p}\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

$$\mathbf{p}''' = \mathbf{p}R_o = \mathbf{p}\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{2.17}$$

We can write similar equations as Eq. 2.15 and Eq. 2.16 for reflections in the xz-plane, yz-plane, x-axis, and y-axis respectively. In total we get 8 combinations (including the identity matrix, which gives the point $\mathbf{p}$ itself).

2.2.5 Homogeneous Transformations

The transformations rotation, scaling, and reflection can all be expressed using matrix multiplication. The composition of any sequence of these transformations can therefore be expressed in a convenient way by a single matrix which is the product of the matrices for the various transformations. However, translation is expressed by a matrix addition, and a sequence of translation, rotation, scaling, and reflection can not be written by a matrix multiplication. By using *homogeneous coordinates* and *homogeneous transformations*, we can express all the transformations above by matrix multiplications. Projective transformations, described in Section 2.3, can also be expressed easily by homogeneous transformations.

Homogeneous Coordinates

In homogeneous coordinates, a point in three dimensions is represented by a vector with four components. A point $\mathbf{p}$ with coordinates x, y, and z, represented by the vector

$$\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix} \tag{2.18}$$

is represented by the homogeneous coordinates hx, hy, hz, and h as

$$\hat{\mathbf{p}} = \begin{bmatrix} hx & hy & hz & h \end{bmatrix} \tag{2.19}$$

where $h \neq 0$. The coordinates x, y, and z are related to the homogeneous coordinates by

$$\begin{aligned} x &= \frac{hx}{h} \\ y &= \frac{hy}{h} \\ z &= \frac{hz}{h} \end{aligned} \tag{2.20}$$

The representation in homogeneous coordinates is not unique, since we can choose any real number h, except 0. For example, the point $(1, 2, 3)$ can be represented by $\hat{\mathbf{p}}_1$ with $h = 1$ as

$$\hat{\mathbf{p}}_1 = \begin{bmatrix} 1 & 2 & 3 & 1 \end{bmatrix} \tag{2.21}$$

or by $\hat{\mathbf{p}}_2$ using $h = 5$ as

$$\hat{\mathbf{p}}_2 = \begin{bmatrix} 5 & 10 & 15 & 5 \end{bmatrix} \tag{2.22}$$

To convert the homogeneous coordinates back to the ordinary coordinates, we divide each component by $h = 5$, and remove the last component to get

$$\mathbf{p} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \tag{2.23}$$

Usually, we choose $h = 1$, to simplify computation.

Transformations of points in homogeneous coordinates are expressed by 4×4 matrices. For translation, we can write

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \tag{2.24}$$

Computing this matrix expression gives

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x + t_x & y + t_y & z + t_z & 1 \end{bmatrix} \tag{2.25}$$

Rotation about the x-axis by α can be expressed as

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \widehat{Rot}_{x,\alpha} \tag{2.26}$$

where $\widehat{Rot}_{x,\alpha}$ is

$$\widehat{Rot}_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.27}$$

In a similar way, we can write the 4×4 matrices $\widehat{Rot}_{y,\beta}$ and $\widehat{Rot}_{z,\gamma}$ for rotating about the y-axis and the z-axis respectively

$$\widehat{Rot}_{y,\beta} = \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.28}$$

$$\widehat{Rot}_{z,\gamma} = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.29}$$

The matrix for scaling, $\hat{S}$ can be written

$$\hat{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.30}$$

A composite transformation can be expressed by a single matrix $\hat{M}$

$$\hat{M} = \left[\begin{array}{ccc|c} a & b & c & p \\ d & e & f & q \\ g & i & j & r \\ \hline k & l & m & h \end{array}\right] \tag{2.31}$$

The upper 3×3 submatrix corresponds to the total effect of rotation and scaling, and the lower 1×3 submatrix corresponds to the total effect of translation. The 3×1 submatrix can be used for projections, described in Section 2.3.

2.2.6 Rotation about an Arbitrary Axis

Suppose we want to rotate a point $\mathbf{p}$ by the angle α about an arbitrary axis directed from $\mathbf{q}$ to $\mathbf{r}$, that is, in the direction $\mathbf{r} - \mathbf{q}$, where $\mathbf{r} \neq \mathbf{q}$. See Figure 2.3 on page 65. The coordinate system is right-handed, and the direction of positive rotation is defined as counterclockwise when looking in the direction from $\mathbf{r}$ to $\mathbf{q}$. This definition is consistent with the definition of the direction of positive rotation about a coordinate axis given in Section 2.2.2, if we let $\mathbf{q}$ be the origin, and $\mathbf{r}$ a point on a positive coordinate axis. Equivalently, the direction of positive rotation is the same direction that a right screw has to be turned in order to be moved from $\mathbf{q}$ to $\mathbf{r}$.

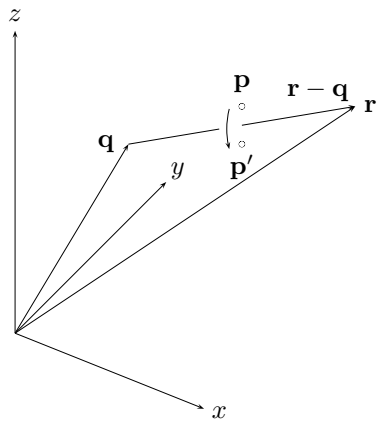

Figure 2.3: Rotation about an arbitrary axis

We can derive an expression for the rotation by α about $\mathbf{r}-\mathbf{q}$ as a sequence of simpler transformations. The idea is to transform the axis of rotation such that it is aligned with one of the coordinate axes, then rotate about that axis by α, and then transform the axis back to its original position. We can apply the following 7 transformations:

1. Translate by $-\mathbf{q}$. The translation of the points $\mathbf{q}$ and $\mathbf{r}$ gives the points $\mathbf{q}'$ and $\mathbf{r}'$, such that $\mathbf{q}'$ is at the origin. See Figure 2.4 on page 66.

The point $\mathbf{r}'$ can be expressed by the spherical coordinates

$$\begin{aligned} R &= \sqrt{r_x'^2 + r_y'^2 + r_z'^2} \\ \theta &= \arctan(r_y', r_x') \\ \phi &= \cos^{-1}\frac{r_z'}{R} \end{aligned} \tag{2.32}$$

where arctan is the two-argument form of $\tan^{-1}$, and is given by

$$\arctan(y,x) = \begin{cases} \tan^{-1}\frac{y}{x}, & \text{if } x > 0 \\ \pi + \tan^{-1}\frac{y}{x}, & \text{if } x < 0 \\ \frac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y < 0 \end{cases} \tag{2.33}$$

The angle θ, called *azimuth*, is the angle between the xz-plane and the plane through $\mathbf{r}'$ and the z-axis. The angle ϕ, called *colatitude*, is the angle between $\mathbf{r}'$ and the z-axis.

If $r_x' = r_y' = 0$, then $\mathbf{r}'$ lies on the z-axis, and since we assumed that $\mathbf{r} \neq \mathbf{q}$, we get $\mathbf{r}' \neq \mathbf{q}'$, and $\mathbf{r}' \neq \mathbf{0}$, which gives $r_z' \neq 0$ and $R = |r_z'|$, and thus $\phi = 0$ or $\phi = \pi$. We can then set $\theta = 0$ in step 2, since the axis given by $\mathbf{r}' - \mathbf{q}'$ already lies in the xz-plane.

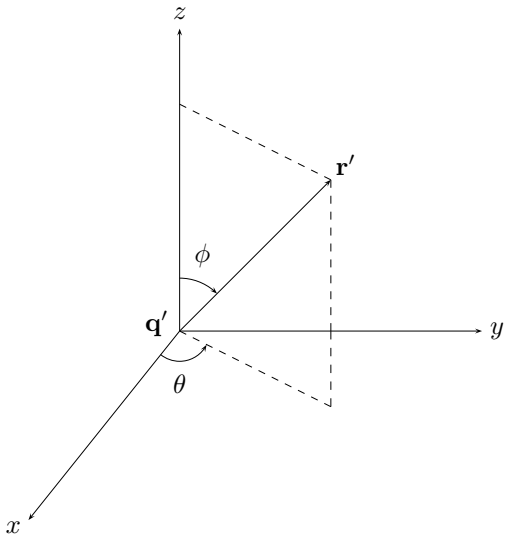

Figure 2.4: Axis of rotation translated

2. Rotate by $-\theta$ about the z-axis. The rotation of $\mathbf{r}'$ is $\mathbf{r}''$, which lies in the xz-plane.

3. Rotate by $-\phi$ about the y-axis. The rotation of $\mathbf{r}''$ is $\mathbf{r}'''$, which lies on the positive z-axis.

Now the axis of rotation $(\mathbf{r}''' - \mathbf{q}''')$ is aligned with the z-axis.

4. Rotate by α about the z-axis.

Now we have to transform the axis of rotation back to the original position. This is done by steps 5–7 below. Note that the transformations is the inverses of the transformations given in step 1–3, in the reverse order.

5. Rotate by ϕ about the y-axis.

6. Rotate by θ about the z-axis.

7. Translate by $\mathbf{q}$.

By using homogeneous transformations, we can combine the 7 transformations into a single transformation, given by the matrix expression

$$\widehat{Rot}_{\hat{\mathbf{q}},\hat{\mathbf{r}},\alpha} = \hat{T}_{-\hat{\mathbf{q}}}\widehat{Rot}_{z,-\theta}\widehat{Rot}_{y,-\phi}\widehat{Rot}_{z,\alpha}\widehat{Rot}_{y,\phi}\widehat{Rot}_{z,\theta}\hat{T}_{\hat{\mathbf{q}}} \tag{2.34}$$

By computing this matrix for the given points $\mathbf{q}$ and $\mathbf{r}$, and the given angle α, we can save a lot of computations when we want to rotate more than one point

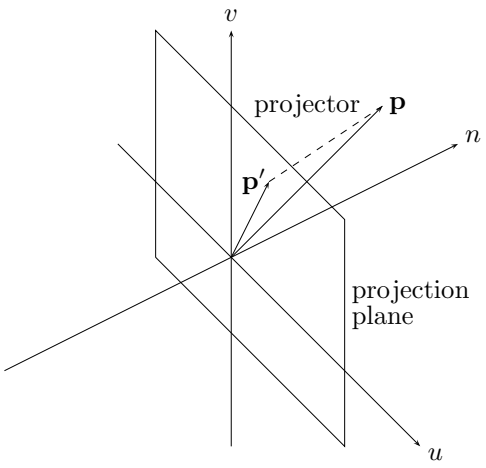

Figure 2.5: Planar geometric projection

p. The rotation of **p** about the axis $\mathbf{r} - \mathbf{q}$ by α is given by the expression (in homogeneous coordinates)

$$\hat{\mathbf{p}}' = \hat{\mathbf{p}}\widehat{Rot}_{\hat{\mathbf{q}},\hat{\mathbf{r}},\alpha} \tag{2.35}$$

2.3 Planar Geometric Projections

This section describes projections from 3 dimensions into 2 dimensions. The projections are *planar*, that is, the projected images reside on a plane rather than a curved surface. The projections are also *geometric*, which means that each *projector* is a straight line, rather than a general curve. For each point **p** to be projected, there is a projector, which is a ray from **p** intersecting the projection plane. The intersection point, $\mathbf{p}'$, is the projection of **p**. See Figure 2.5. We define the projection plane as the uv-plane. The coordinate system given by $\mathbf{u}, \mathbf{v}, \mathbf{n}$, is a left-handed coordinate system. A left-handed system is often more intuitive to use, since we can think of **u** and **v** as the directions right and up, respectively, and a point **p** in front of us increases its n coordinate as it moves away from us.

It is convenient to represent 3D objects in a fixed right-handed coordinate system, often called the *world* coordinate system, and then transform each object into a *view* coordinate system given by $\mathbf{u}, \mathbf{v}, \mathbf{n}$. The position and orientation of the view system relative to the world system can then be varied, in order to create different views of the objects in the world. In this section we will assume that we have transformed all objects into the view coordinate system. Section 2.4 describes a useful model to transform between the two systems.

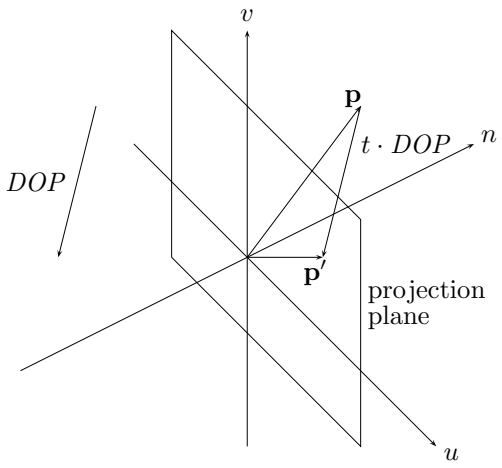

Figure 2.6: Parallel projection

The projections can be classified into parallel projections and perspective projections, described in Section 2.3.1 and Section 2.3.2, respectively.

2.3.1 Parallel Projections

For parallel projections, all projectors are parallel, and we can denote their direction DOP, *direction of projection*. See Figure 2.6. Parallel projections can be classified into two types, depending on the relation between DOP and the projection plane. If DOP is orthogonal to the projection plane, the projection is called *orthographic*, otherwise it is called an *oblique* projection. We assume that DOP is not parallel to the projection plane, that is, we assume that $DOP_n \neq 0$. We can write an expression for $\mathbf{p}'$, the projection of $\mathbf{p}$ as

$$\mathbf{p}' = \mathbf{p} + t \cdot DOP \tag{2.36}$$

The point $\mathbf{p}'$ lies in the projection plane, so its n coordinate is 0. We can write Eq. 2.36 as three equations

$$u' = u + t \cdot DOP_u \tag{2.37}$$
$$v' = v + t \cdot DOP_v \tag{2.38}$$
$$0 = n + t \cdot DOP_n \tag{2.39}$$

We can solve for t in Eq. 2.39 which gives

$$t = -\frac{n}{DOP_n} \tag{2.40}$$

and then use this t in Eq. 2.37 and Eq. 2.38 which gives

$$u' = u - n\frac{DOP_u}{DOP_n} \tag{2.41}$$

$$v' = v - n\frac{DOP_v}{DOP_n} \tag{2.42}$$

We can write this as a matrix expression, and use homogeneous coordinates

$$\hat{\mathbf{p}}' = \hat{\mathbf{p}}\hat{M}_{par} \tag{2.43}$$

that is

$$\begin{bmatrix} u' & v' & 0 & 1 \end{bmatrix} = \begin{bmatrix} u & v & n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{DOP_u}{DOP_n} & -\frac{DOP_v}{DOP_n} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.44}$$

If $DOP_u = DOP_v = 0$, we get orthogonal projection, and $\hat{M}_{par}$ is simplified to

$$\hat{M}_{par} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.45}$$

2.3.2 Perspective Projections

For perspective projections, all projectors converge into a point called COP, *center of projection*. See Figure 2.7 on page 70, which shows the projection of $\mathbf{p}$ into $\mathbf{p}'$ on the projection plane (uv-plane). Let $\mathbf{p}$ and COP be given by the row vectors

$$\mathbf{p} = \begin{bmatrix} u & v & n \end{bmatrix} \tag{2.46}$$

$$COP = \begin{bmatrix} COP_u & COP_v & COP_n \end{bmatrix} \tag{2.47}$$

We assume that $COP_n < 0$ and $n > COP_n$. The projector can be parametrized as

$$\mathbf{r}(t) = COP(1-t) + \mathbf{p}t \tag{2.48}$$

The projector intersects the projection plane at $t = t'$, and the intersection point, $\mathbf{p}'$, is the projection of $\mathbf{p}$, that is

$$\mathbf{p}' = \mathbf{r}(t') = COP(1-t') + \mathbf{p}t' \tag{2.49}$$

We can write Eq. 2.49 as three equations

$$u' = COP_u(1-t') + ut' \tag{2.50}$$

$$v' = COP_v(1-t') + vt' \tag{2.51}$$

$$0 = COP_n(1-t') + nt' \tag{2.52}$$

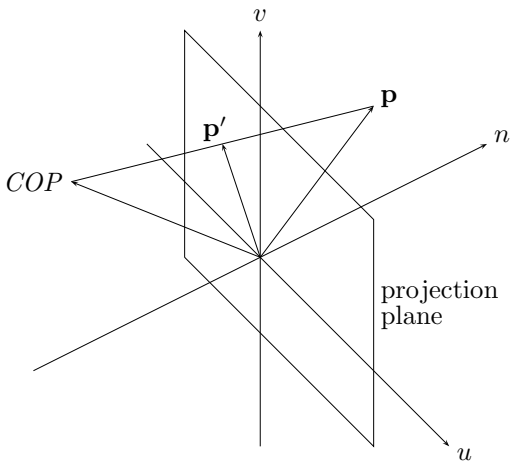

Figure 2.7: Perspective projection

Solving for t' in Eq. 2.52 gives

$$t' = \frac{COP_n}{COP_n - n} \ , \ COP_n \neq n \tag{2.53}$$

Using this in Eq. 2.50 and Eq. 2.51, and simplifying, gives

$$u' = \frac{uCOP_n - nCOP_u}{COP_n - n} \tag{2.54}$$

$$v' = \frac{vCOP_n - nCOP_v}{COP_n - n} \tag{2.55}$$

Consider the matrix expression in homogeneous coordinates

$$\begin{bmatrix} hu' & hv' & 0 & h \end{bmatrix} = \begin{bmatrix} u & v & n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{COP_u}{COP_n} & -\frac{COP_v}{COP_n} & 0 & -\frac{1}{COP_n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Computing the matrix expression gives

$$hu' = u - n\frac{COP_u}{COP_n} = \frac{uCOP_n - nCOP_u}{COP_n} \tag{2.56}$$

$$hv' = v - n\frac{COP_v}{COP_n} = \frac{vCOP_n - nCOP_v}{COP_n} \tag{2.57}$$

$$h = 1 - \frac{n}{COP_n} = \frac{COP_n - n}{COP_n} \tag{2.58}$$

Dividing hu' and hv' by h gives

$$u' = \frac{uCOP_n - nCOP_u}{COP_n - n} \tag{2.59}$$

$$v' = \frac{vCOP_n - nCOP_v}{COP_n - n} \tag{2.60}$$

That is, we can express the perspective projection by a matrix multiplication using homogeneous coordinates.

Often we let the center of projection be a point on the negative n-axis, that is, $COP_u = COP_v = 0$. Then we can simplify the expressions for u' and v' to

$$u' = u\frac{COP_n}{COP_n - n} \tag{2.61}$$

$$v' = v\frac{COP_n}{COP_n - n} \tag{2.62}$$

The expression $\frac{COP_n}{COP_n - n}$ is called the *foreshortening factor*. Since we have chosen $COP_n < 0$ we can write

$$\frac{COP_n}{COP_n - n} = \frac{-|COP_n|}{-|COP_n| - n} = \frac{|COP_n|}{|COP_n| + n} \tag{2.63}$$

and for $n > 0$ we get

$$0 < \frac{|COP_n|}{|COP_n| + n} < 1 \tag{2.64}$$

and as an object moves farther away, its projected image becomes smaller. For $n < 0$ we can write Eq. 2.63 as

$$\frac{COP_n}{COP_n - n} = \frac{|COP_n|}{|COP_n| - |n|} \tag{2.65}$$

and since we assumed that $COP_n < n$, we get $0 < |COP_n| - |n| < |COP_n|$ and thus

$$\frac{|COP_n|}{|COP_n| - |n|} > 1 \tag{2.66}$$

that is, for an object between the projection plane and COP, the projected image becomes magnified.

Pseudodepth

As we have seen before, if we choose COP as a point on the negative n-axis, we get the coordinates of the projected point $\mathbf{p}'$ as

$$u' = u\frac{COP_n}{COP_n - n} = \frac{u}{1 - n/COP_n} \tag{2.67}$$

$$v' = v\frac{COP_n}{COP_n - n} = \frac{v}{1 - n/COP_n} \tag{2.68}$$

$$n' = 0 \tag{2.69}$$

which can be written as a matrix multiplication using homogeneous coordinates

$$\hat{\mathbf{p}}' = \hat{\mathbf{p}}\hat{M}_{persp} \tag{2.70}$$

where $\hat{\mathbf{p}}'$, $\hat{\mathbf{p}}$, and $\hat{M}_{persp}$ are given by

$$\hat{\mathbf{p}}' = \begin{bmatrix} hu' & hv' & hn' & h \end{bmatrix} \tag{2.71}$$

$$\hat{\mathbf{p}} = \begin{bmatrix} u & v & n & 1 \end{bmatrix} \tag{2.72}$$

$$\hat{M}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{COP_n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.73}$$

The perspective projection in some way destroys information, since we have lost the distance from the projection plane to the point $\mathbf{p}$. For example, consider two points, $\mathbf{p}_1$ and $\mathbf{p}_2$, lying on the same projector, that is, they both project to the same point $\mathbf{p}'$, and we do not know which of the two points that was projected into $\mathbf{p}'$. This can also be seen by the fact that the matrix $\hat{M}_{persp}$ is singular.

It is useful to represent a measure of the distance from the projection plane to the point $\mathbf{p}$. A natural choice of the measure is the expression

$$n'' = n\frac{COP_n}{COP_n - n} = \frac{n}{1 - n/COP_n} \tag{2.74}$$

which is called the *pseudodepth*. Since we have chosen $COP_n < 0$, we can write

$$n'' = \frac{n}{1 + n/|COP_n|} \tag{2.75}$$

and as n increases, n'' increases monotonically. We can then write Eq. 2.70 as

$$\hat{\mathbf{p}}' = \hat{\mathbf{p}}\hat{M}_p\hat{M}_{proj} \tag{2.76}$$

where the matrix $\hat{M}_p$ represents a *perspective transformation*, and is given by

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{COP_n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.77}$$

and $\hat{M}_{proj}$ is a projection matrix given by

$$\hat{M}_{proj} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.78}$$

The matrix $\hat{M}_{proj}$ is the same as the orthogonal projection matrix $\hat{M}_{par}$ given by Eq. 2.45 on page 69. We can therefore treat a perspective projection as a perspective transformation followed by an orthogonal projection. This fact will be useful in the clipping process, described in Section 2.4.3. The pseudodepth is useful in for example hidden surface removal, described in Section 2.5.

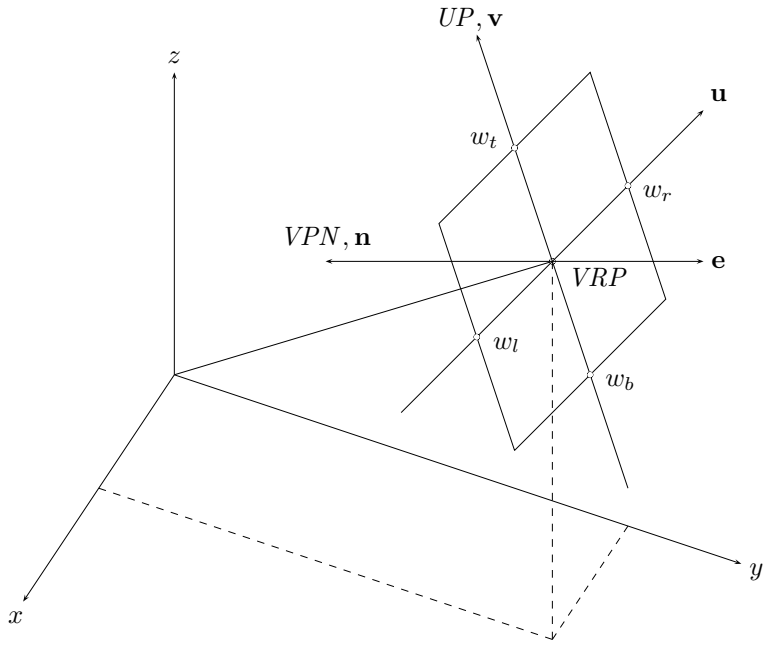

Figure 2.8: Synthetic-camera

2.4 Viewing in 3D

This section describes a model which is useful for determining what should be seen of the 3D objects. The model is often called *the synthetic-camera model*. We specify a *view volume* which determines which objects should be seen. The process of determining which objects that reside in the view volume is called *clipping*. The synthetic-camera model, the view volume, and clipping are described in Section 2.4.1, Section 2.4.2, and Section 2.4.3, respectively.

2.4.1 The Synthetic-Camera Model

We can imagine a camera attached to the view coordinate system, taking snapshots of objects. The camera can be positioned and oriented by specifying the position and orientation of the view coordinate system relative the world coordinate system. See Figure 2.8. The position and orientation of the view coordinate system are specified by

1. *VRP*: view reference point, which is the origin of the view coordinate system.

2. *VPN*: viewplane normal, this direction becomes the direction of **n** in the view coordinate system.

3. *UP*: a vector which defines the up direction, that is, defining $\mathbf{v}$ in the view coordinate system. *UP* is perpendicular to *VPN*.

VRP, *VPN*, and *UP* are specified in the world coordinate system. We can compute $\mathbf{u}$ as the vector product of $\mathbf{n}$ and $\mathbf{v}$, that is

$$\mathbf{u} = \mathbf{n} \times \mathbf{v} \tag{2.79}$$

This gives a left-handed coordinate system with the coordinates u, v, n. In the viewplane, we define a *window* which determines the size of the snapshots taken by the camera. The window borders are specified in the view coordinate system by w_l, w_r, w_t, and w_b. $u = w_l$ and $u = w_r$ define the left and right borders of the window, and $v = w_t$ and $v = w_b$ define the top and bottom borders. We also define an eye position, $\mathbf{e}$, in the view coordinate system. For a perspective projection, $\mathbf{e}$ is the center of projection, *COP*. For a parallel projection, we can define the direction of projection, *DOP*, as the direction of a vector from the origin of the view coordinate system to the point $\mathbf{e}$.

In an interactive computer graphics application, the following approach is useful for specifying the position and orientation of the view coordinate system relative the world coordinate system.

1. The user specifies the *VRP* which becomes the origin of the view coordinate system.

2. The user specifies *VPN*, the viewplane normal, or alternatively, a point S which the user wants to look at. *VPN* is then computed as $VPN = \mathbf{s} - \mathbf{r}$, where $\mathbf{s}$ and $\mathbf{r}$ are the position vectors of S and *VRP* respectively.

The unit vector $\mathbf{n}$ is then computed as

$$\mathbf{n} = \frac{VPN}{|VPN|} \tag{2.80}$$

3. The user specifies an approximate up direction, UP', which does not have to be orthogonal to *VPN*.

The actual up direction is then computed as

$$UP = UP' - (UP' \cdot \mathbf{n})\mathbf{n} \tag{2.81}$$

In this way, *UP* will be orthogonal to $\mathbf{n}$, provided that UP' was not parallel to *VPN*. Then the unit vector $\mathbf{v}$ is formed by

$$\mathbf{v} = \frac{UP}{|UP|} \tag{2.82}$$

and finally, $\mathbf{u}$ is computed as

$$\mathbf{u} = \mathbf{n} \times \mathbf{v} \tag{2.83}$$

This approach gives the user an intuitive way to specify the view.

World to View Coordinate System Transformation

The vectors **u**, **v**, and **n** are expressed in the world coordinate system, and are given by the row vectors

$$\begin{aligned} \mathbf{u} &= \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} \\ \mathbf{v} &= \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \\ \mathbf{n} &= \begin{bmatrix} n_x & n_y & n_z \end{bmatrix} \end{aligned} \tag{2.84}$$

We would like to compute a transformation matrix, $\hat{A}_{wv}$, which transforms the coordinates of a point **p** in the world coordinate system into coordinates in the view coordinate system. Let the view reference point, *VRP*, be represented by the row vector **r** in the world coordinate system. We can write the transformation matrix $\hat{A}_{wv}$ in homogeneous coordinates as

$$\hat{A}_{wv} = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ r'_x & r'_y & r'_z & 1 \end{bmatrix} \tag{2.85}$$

where r'_x, r'_y, and r'_z are given by

$$\begin{aligned} r'_x &= -\mathbf{r} \cdot \mathbf{u} \\ r'_y &= -\mathbf{r} \cdot \mathbf{v} \\ r'_z &= -\mathbf{r} \cdot \mathbf{n} \end{aligned} \tag{2.86}$$

for the details, see [2].

2.4.2 View Volume

The window together with either *DOP* (for parallel projection), or *COP* (for perspective projection), define the view volume. For a parallel projection, we get a parallelepiped, see Figure 2.9 on page 76. For simplicity, the figure shows an orthogonal projection, that is, *DOP* is orthogonal to the viewplane, in the direction of the negative n-axis. Figure 2.10 shows the view volume for a perspective projection. For simplicity, we put the *COP* on the negative n-axis. In Figure 2.9 and Figure 2.10 we have also introduced a *front plane* at $n = F$ and a *back plane* at $n = B$. The front and back planes are used for restricting the view volume, and objects outside the view volume will not be projected. In general, we assume that $e_n < F < B$.

2.4.3 Clipping

Clipping is the process of rejecting objects outside the view volume, and preserving objects which reside inside the volume. We can first test which objects are either completely inside, or completely outside the view volume. An object partly inside and partly outside the view volume has to be cut into two pieces, such that one piece is inside, and the other piece is outside the volume.

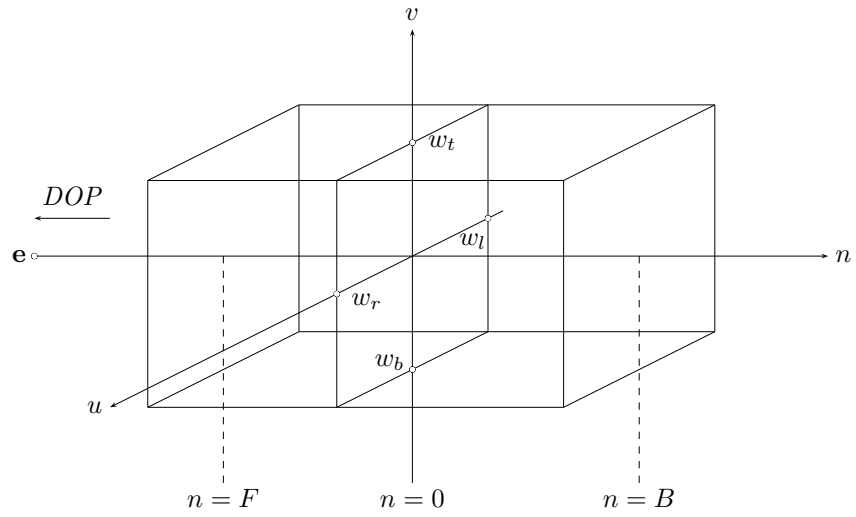

Figure 2.9: View volume for orthogonal projection

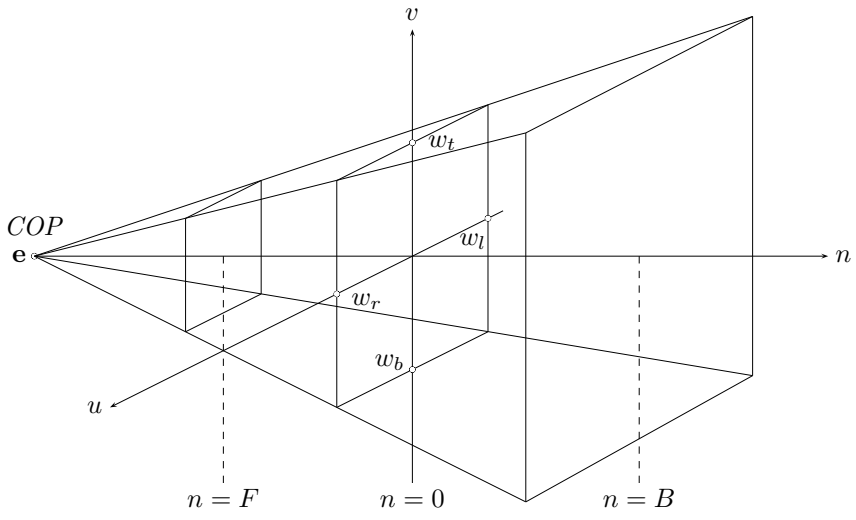

Figure 2.10: View volume for perspective projection

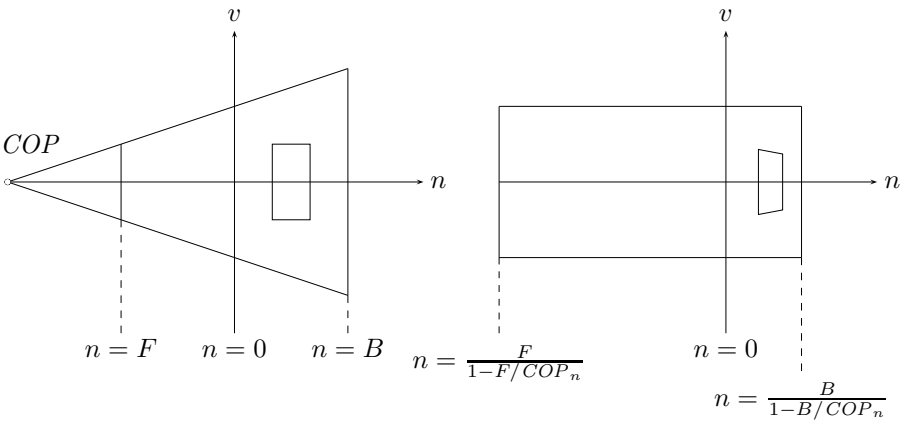

Figure 2.11: Prewarping

For an orthogonal projection, the view volume was shown in Figure 2.9 on page 76. To test if a point $\mathbf{p}$ (given in the view coordinate system), is inside the view volume, we can use the following inequalities, which all must hold

$$\begin{aligned} w_l &< u < w_r \\ w_b &< v < w_t \\ F &< n < B \end{aligned} \tag{2.87}$$

For a perspective projection, the view volume was shown in Figure 2.10 on page 76. In this case, we can not test each coordinate of $\mathbf{p}$ with three independent inequalities as was the case with the orthogonal projection. However, as we saw in Section 2.3.2, we can express a perspective projection as a perspective transformation followed by an orthogonal projection. The idea is to transform the point $\mathbf{p}$ into a point $\mathbf{q}$ (in homogeneous coordinates) as

$$\hat{\mathbf{q}} = \hat{\mathbf{p}}\hat{M}_p \tag{2.88}$$

where $\hat{M}_p$ was given by Eq. 2.77 on page 72. This transformation is often called *prewarping*. See Figure 2.11, which shows the original perspective view volume to the left, and the prewarped view volume to the right. In each part of the figure, an object is also shown between the viewplane and the back plane. In the right part of Figure 2.11, we can see the perspective foreshortening. See [2] for the details of how the volume to the left will be transformed into the volume to the right. Now we can test if a point $\mathbf{p}$ is inside the original view volume by testing if the prewarped point $\mathbf{q}$ is inside the transformed view volume, using the following inequalities

$$\begin{aligned} w_l &< u' < w_r \\ w_b &< v' < w_t \end{aligned} \tag{2.89}$$

$$\frac{F}{1 - F/COP_n} < n' < \frac{B}{1 - B/COP_n}$$

where n' is the pseudodepth of the point $\mathbf{p}$.

Two common clipping algorithms are the Cohen-Sutherland line clipping algorithm, and the Liang-Barsky polygon clipping algorithm. They are described in [1].

2.5 Hidden Surface Removal

In order to get a realistic view of a 3D object, we need to remove hidden surfaces. In this section we mention some common methods. For more details, see [1] and [2].

2.5.1 Back Face Culling

For a planar polygon, we can define a unique normal vector. It is common to specify the vertices of a polygon in a counterclockwise order, and define the normal vector as pointing towards us. When we have set up the viewing transform and transformed each vertex of each polygon, we can check the direction of the normal vectors. If a polygon has a normal vector pointing away from us, the polygon can be culled out (removed) since it can not be seen. However, this method is not guaranteed to work correctly for a nonconvex object.

2.5.2 Depth Buffer Algorithm

The depth buffer algorithm is used after projection, during the rendering of a polygon. The pseudodepth is stored for each vertex in the polygon. The idea is to render the polygon without showing it on the screen directly, but to store the color of each point of the polygon into an intermediate buffer and to store the pseudodepth of the point in a depth buffer if the current value in the depth buffer corresponds to a point of a polygon which are more distant. If the current value in the depth buffer corresponds to a point of a previously rendered polygon with a smaller pseudodepth, the intermediate buffer and the depth buffer are not modified. When all polygons have been rendered into the intermediate buffer, the contents of the buffer is shown on the screen.

Chapter 3

Rendering the Projected Objects

3.1 Introduction

This chapter describes how the images of the projected objects are *rendered*, that is, plotted on a computer screen. Nowadays, *raster displays* are most common. A raster display consists of a rectangular grid of *pixels*. Pixel is an abbreviation for picture element. Rendering of curves is described in Section 3.2 and rendering of surfaces in Section 3.3.

3.2 Rendering Curves

In this section, we will look at the basic algorithms for rendering curves efficiently. Section 3.2.1 and Section 3.2.2 describes rendering of lines and circles respectively. Rendering of Bézier curves is described briefly in Section 3.2.3.

3.2.1 Rendering Lines

In this section we derive an algorithm for rendering lines, the *midpoint line algorithm*. The algorithm generates the pixels which approximate a line between two end points, given by integer coordinates.

The Midpoint Line Algorithm

Before we derive the algorithm, consider the line between the points $\mathbf{p} = (0, 0)$ and $\mathbf{q} = (7, 3)$. Figure 3.1 on page 80 shows the ideal line, and the pixels which approximate the line. Note, in this case, we have plotted exactly one pixel in each column of the grid. For this particular line, the algorithm starts by plotting the pixel at $(0, 0)$, increases the x coordinate in steps of 1, and plots the pixel closest to the ideal line. This works because the slope of the line, which is $\frac{3}{7}$, is

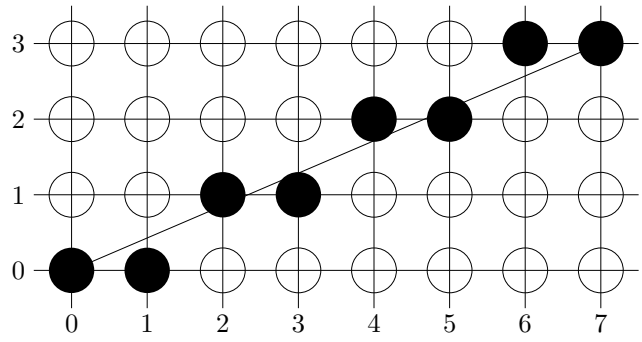

Figure 3.1: Midpoint line algorithm

less than 1. If the slope was greater than 1, we would get more than one pixel in some of the columns.

To derive the midpoint line algorithm for a line between the points (x_0, y_0) and (x_n, y_n), we start with the following assumptions

$$x_0 < x_n \tag{3.1}$$

$$0 \leq \frac{y_n - y_0}{x_n - x_0} = \frac{\Delta y}{\Delta x} \leq 1 \tag{3.2}$$

The algorithm can be extended to handle general lines, this which will be discussed briefly later.

Consider Figure 3.2 on page 81, which shows an intermediate step in the algorithm. We have plotted a pixel at (x_p, y_p), denoted the previous pixel, and we want to choose the current pixel to be plotted. For the current pixel we get two choices, denoted by E (east), and NE (northeast). Let Q be the point of intersection between the ideal line and the grid line $x = x_p + 1$. Let M be the midpoint between the pixels E and NE, that is, $M = (x_p + 1, y_p + 1/2)$. To choose between E and NE, we check which side of the line M lies. If M is below the line, as is the case in Figure 3.2, we choose the current pixel as NE. If M is above the line, we choose E. If M is exactly on the line, we can choose either E or NE, so we choose E arbitrarily. We can represent the line with an implicit function as

$$F(x, y) = ax + by + c = 0 \tag{3.3}$$

and also in explicit form as

$$y = \frac{\Delta y}{\Delta x} x + B \tag{3.4}$$

which can be written

$$\frac{\Delta y}{\Delta x} x - y + B = 0 \tag{3.5}$$

multiplying by Δx gives

$$\Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot B = 0 \tag{3.6}$$

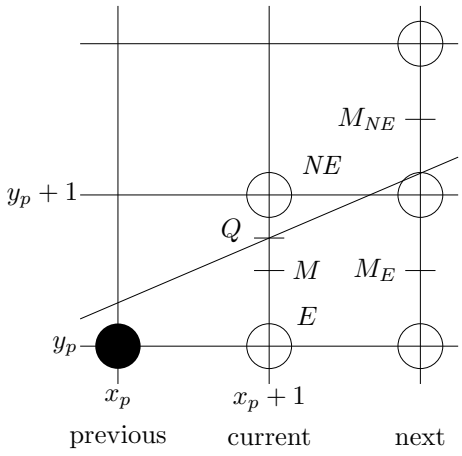

Figure 3.2: Choices for current and next pixels

comparing with Eq. 3.3 gives

$$\begin{aligned} a &= \Delta y \\ b &= -\Delta x \\ c &= \Delta x \cdot B \end{aligned} \tag{3.7}$$

We note that $F(x, y) < 0$ for a point above the line, $F(x, y) > 0$ for a point below the line, and $F(x, y) = 0$ for a point on the line. We check if the midpoint $M = (x_p + 1, y_p + 1/2)$ is above or below the line by introducing a decision variable d, defined by

$$d = F(x_p + 1, y_p + 1/2) = a(x_p + 1) + b(y_p + 1/2) + c \tag{3.8}$$

If $d > 0$, M is below the line, and we choose the current pixel as NE. If $d < 0$, M is above the line, and we choose the pixel E. If $d = 0$, M is on the line, so we choose E.

When we have chosen the current pixel, we update the decision variable. The new value depends on which pixel we have chosen, and we get two cases:

1. If we have chosen NE, we move the midpoint M one step in both the x and the y direction, that is, to M_{NE}. Thus, we get the new decision variable, d_{new}, as

   $$d_{new} = F(x_p + 2, y_p + 3/2) = a(x_p + 2) + b(y_p + 3/2) + c \tag{3.9}$$

   which can be written

   $$d_{new} = a(x_p + 1) + b(y_p + 1/2) + c + a + b \tag{3.10}$$

comparing with Eq. 3.8, denoting the old value of d as d_{old}, and using Eq. 3.7, gives

$$d_{new} = d_{old} + a + b = d_{old} + \Delta y - \Delta x \tag{3.11}$$

We denote the difference between d_{new} and d_{old} when NE is chosen as Δ_{NE}, given by

$$\Delta_{NE} = \Delta y - \Delta x \tag{3.12}$$

2. If E was chosen, the midpoint M is moved one step in the x direction, that is, to M_E, and we get d_{new} as

$$d_{new} = F(x_p + 2, y_p + 1/2) = a(x_p + 2) + b(y_p + 1/2) + c \tag{3.13}$$

which is written

$$d_{new} = a(x_p + 1) + b(y_p + 1/2) + c + a \tag{3.14}$$

comparing with Eq. 3.8, and using Eq. 3.7, gives

$$d_{new} = d_{old} + a = d_{old} + \Delta y \tag{3.15}$$

and the difference between d_{new} and d_{old} when E was chosen is denoted Δ_E, given by

$$\Delta_E = \Delta y \tag{3.16}$$

Finally, we need an expression for the initial value of d, denoted d_{init}, computed at the midpoint M when the first point at (x_0, y_0) has been plotted

$$d_{init} = F(x_0 + 1, y_0 + 1/2) = a(x_0 + 1) + b(y_0 + 1/2) + c \tag{3.17}$$

which is written

$$d_{init} = ax_0 + by_0 + c + a + \frac{b}{2} = F(x_0, y_0) + a + \frac{b}{2} \tag{3.18}$$

but $F(x_0, y_0) = 0$, since the point (x_0, y_0) lies on the line, and we get

$$d_{init} = a + \frac{b}{2} = \Delta y - \frac{\Delta x}{2} \tag{3.19}$$

We can refine the algorithm to work with integers only by defining a new decision variable d', as

$$d' = 2d = 2F(x_p + 1, y_p + 1/2) \tag{3.20}$$

Since we only use the checks $d < 0$, $d > 0$, and $d = 0$, the same checks will apply to the new decision variable d'. The initial value of d' is then

$$d'_{init} = 2\Delta y - \Delta x \tag{3.21}$$

and the increments of d' when NE and E are chosen, respectively, are given by

$$\begin{aligned} \Delta'_{NE} &= 2(\Delta y - \Delta x) \\ \Delta'_E &= 2\Delta y \end{aligned} \tag{3.22}$$

The advantage is that the algorithm now only uses integer addition and subtraction, and multiplications by 2, which can be implemented efficiently in hardware.

We have only treated the special case when the slope of the line is between 0 and 1, and we assumed that $x_0 < x_n$. We can extend the algorithm to handle more general cases, for example, if the slope is > 1, we can exchange the role of x and y in the algorithm, such that we increase y in steps of 1, and choose between pixels with different x coordinates, that is, we choose between the pixels N (north) and NE (northeast).

For cases where $x_0 > x_n$, we can decrease x, and adjust the the algorithm to choose between pixels W (west) and NW (northwest) or SW (southwest) respectively. See Section 4.2 for a program that handles general lines.

3.2.2 Rendering Circles

In this section we derive an algorithm for rendering circles, the *midpoint circle algorithm*. A similar technique for choosing between pixels is used as was used in the midpoint line algorithm.

The Midpoint Circle Algorithm

We assume that the circle has an integer radius, R, and is centered at the origin. For a circle not centered at the origin, we can translate each pixel the algorithm generates by (x_c, y_c), where (x_c, y_c) is the integer coordinates of the center of the circle.

The algorithm basically generates pixels in the second octant (here, by octant we mean a quadrant divided into two parts), and uses symmetry to generate the pixels in the remaining 7 octants. Before we derive the algorithm, consider the circle with radius 15 centered at the origin. Figure 3.3 on page 84 shows the part of the ideal circle and the pixels generated by the algorithm in the first quadrant. The algorithm starts by plotting the pixel at $(x, y) = (0, R)$, then chooses between the pixels $(1, R)$ and $(1, R-1)$, and plots the pixel closest to the ideal circle. It repeats this process as long as $y > x$. For each pixel generated, 8 pixels are plotted, one for each octant.

Consider Figure 3.4 on page 84, which shows a magnified view of an intermediate step in the algorithm. We have plotted a pixel at (x_p, y_p), denoted the previous pixel, and we want to choose the current pixel to be plotted. For the current pixel we get two choices, denoted by E (east), and SE (southeast). To choose between these pixels, we check whether the midpoint $M = (x_p + 1, y_p - 1/2)$ is inside or outside the ideal circle. If M is outside, as in Figure 3.4, the pixel SE is chosen, since it is closest to the circle. If M is inside, pixel E is chosen, and if M is exactly on the circle, we can choose either E or SE, so we choose for example SE. To check whether M is inside or outside the circle, we can use the implicit equation of a circle, given by

$$F(x, y) = x^2 + y^2 - R^2 = 0 \tag{3.23}$$

For a point inside the circle, $F(x, y) < 0$, and for a point outside the circle, $F(x, y) > 0$. If the point lies on the circle, $F(x, y) = 0$. We define a decision

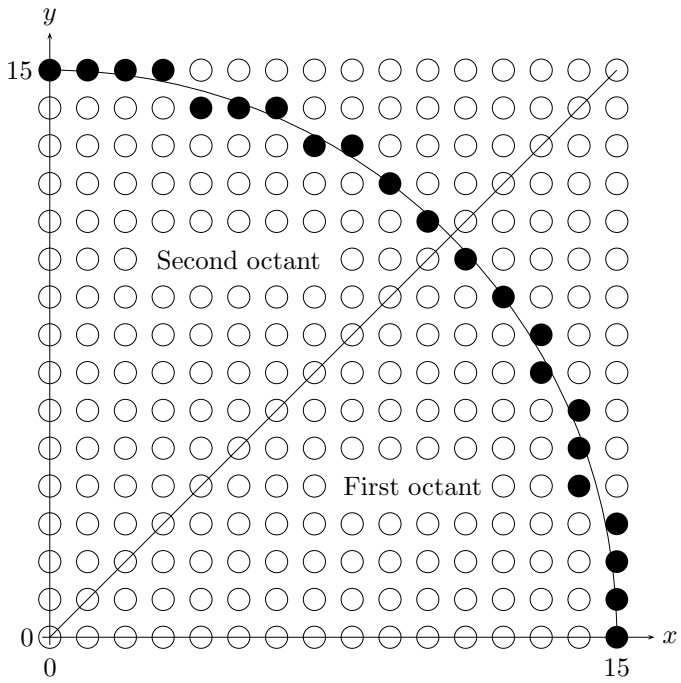

Figure 3.3: Midpoint circle algorithm

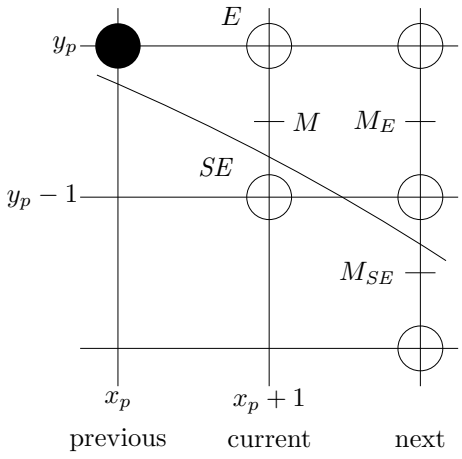

Figure 3.4: Choices for current and next pixels

variable, d, corresponding to the midpoint M, as

$$d = F(x_p + 1, y_p - 1/2) = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2 \tag{3.24}$$

If $d < 0$, M is inside the circle, so we choose pixel E. If $d \geq 0$, M is outside or on the circle, so we choose pixel SE. When we have chosen the current pixel, we update the decision variable to correspond to one of the midpoints M_E and M_{SE} respectively. In Figure 3.4 we choose the current pixel as SE, so the next midpoint will be M_{SE}. We denote the old decision variable given by Eq. 3.24 as d_{old}. Depending on the pixel chosen as the current we get two cases:

1. If E is chosen, we get the new decision variable d_{new}, as

   $$d_{new} = F(x_p + 2, y_p - 1/2) = (x_p + 2)^2 + (y_p - 1/2)^2 - R^2 \tag{3.25}$$

   which can be written

   $$d_{new} = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2 + 2x_p + 3 \tag{3.26}$$

   which is then written

   $$d_{new} = d_{old} + 2x_p + 3 \tag{3.27}$$

   The increment of d when E is chosen is denoted Δ_E, and is given by

   $$\Delta_E = 2x_p + 3 \tag{3.28}$$

2. If SE is chosen, we get the new decision variable d_{new}, as

   $$d_{new} = F(x_p + 2, y_p - 3/2) = (x_p + 2)^2 + (y_p - 3/2)^2 - R^2 \tag{3.29}$$

   which can be written

   $$d_{new} = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2 + 2x_p + 3 - 2y_p + \frac{8}{4} \tag{3.30}$$

   and thus

   $$d_{new} = d_{old} + 2x_p - 2y_p + 5 \tag{3.31}$$

   The increment of d when SE is chosen is denoted Δ_{SE}, and is given by

   $$\Delta_{SE} = 2x_p - 2y_p + 5 \tag{3.32}$$

Finally, we need an initial value of d, denoted d_{init}, corresponding to the first midpoint $M = (1, R - 1/2)$

$$d_{init} = F(1, R - 1/2) = 1^2 + (R - 1/2)^2 - R^2 = \frac{5}{4} - R \tag{3.33}$$

We can refine the algorithm to use integers only by defining a new decision variable, d', given by

$$d' = d - \frac{1}{4} \tag{3.34}$$

The initial value of d', denoted d'_{init}, is then

$$d'_{init} = 1 - R \tag{3.35}$$

The original comparisons $d < 0$ and $d \geq 0$ then becomes $d' < -\frac{1}{4}$ and $d \geq -\frac{1}{4}$, but since d' is initialized to d'_{init}, which is an integer, and the increments Δ_E and Δ_{SE} are both integers, d' will always be an integer, so we can use the comparisons $d' < 0$ and $d' \geq 0$ instead.

See Section 4.2 for a program that implements the midpoint circle algorithm.

3.2.3 Rendering Bézier Curves

In this section we describe how Bézier curves can be rendered. One way to do this is to approximate the Bézier curve by piecewise linear segments, as in Section 1.2.8 on page 47. However, it can be difficult to choose the spacing between the parameter values u_i in order to get a good approximation, and at the same time get a fast rendering.

In Section 1.2.3 on page 29 subdivision of a Bézier curve was described. We used the de Casteljau algorithm to compute the control points for two Bézier curves created by subdividing a given Bézier curve. For example, Figure 1.15 on page 36 showed a cubic Bézier curve divided into two Bézier curves. Note that the convex hull for each of the two parts of the original curve was shrinked compared to the convex hull for the original curve. We can use the technique of subdivision to render a Bézier curve by recursively subdivide the curve, until the convex hull for each part of the curve is sufficiently small. See Section 4.3 for a program implementing this idea.

3.3 Rendering Surfaces

A parametric surface can be rendered by rendering the coordinate curves of the surface. For example, in Figure 1.23 on page 52, a rational Bézier surface was shown. By choosing the number of coordinate curves sufficiently large, we can get the impression of a surface. Figure 1.24 and Figure 1.25 on pages 53 and 54, respectively, are other examples.

In Section 1.3.5 on page 55 approximation of surfaces by polygon meshes was described. The polygons in the mesh were triangulated. In this way, the polygons are both convex and planar, and there exist efficient algorithms for rendering them. Figure 3.5 on page 87 shows an approximation of a sphere created by a polyhedron consisting of 320 polygons. The polyhedron was created by triangulization of an icosahedron. An icosahedron is a regular polyhedron with 20 triangular faces. Each face was divided into 4 triangles, yielding a polyhedron with 80 triangular faces. Each of these faces was then further divided into 4 triangles, thus giving 320 triangles. Finally, each vertex of the polyhedron was translated to get a unit distance to the origin.

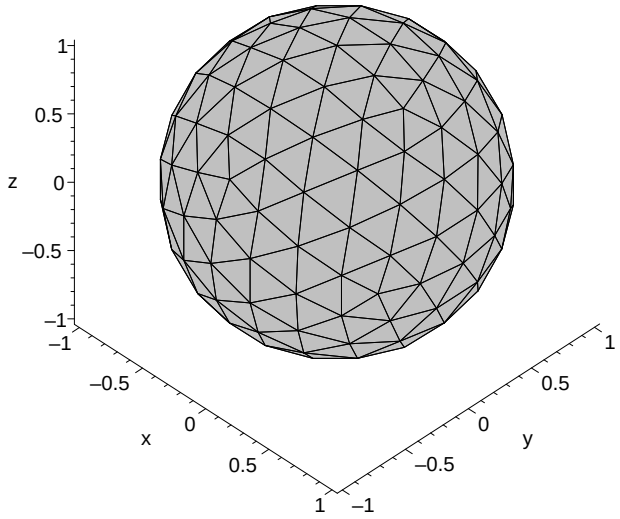

Figure 3.5: Approximation of a sphere by polygons

Chapter 4

Programming Examples

4.1 Introduction

This chapter shows some program examples written in Java. Section 4.2 demonstrates the midpoint algorithms for lines, circles and for ellipses. In order to see the pixels chosen by the algorithms, we simulate the pixels as rather large circles. Section 4.3 lists a program for rendering a Bézier curve, using recursive subdivision. In this program, we use the line drawing routines supported in Java. Note, these programs are meant to demonstrate the basic ideas only. For real graphics applications, we should use graphics libraries such as OpenGL (`http://www.opengl.org`), for example.

4.2 Midpoint Algorithm for Lines, Circles and Ellipses

4.2.1 Midpoint Line Algorithm

```
//
// Demonstration of the Midpoint Line Algorithm
//

import java.awt.*;
import java.awt.event.*;

public class MidpointLine extends Frame {

    public static void main(String[] args) {

        new MidpointLine();
    }

    public MidpointLine() {
```

```
    super("Demo of the Midpoint Line Algorithm");
    addWindowListener(new WindowAdapter()
        {public void windowClosing(WindowEvent e){System.exit(0);}});
    setSize(500,510);
    add("Center", new MyCanvas());
    setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
    show();
}

public class MyCanvas extends Canvas implements MouseListener {

    final int GRID_SPACE = 15;
    final int PIXEL_RADIUS = 6;
    int maxX, maxY;
    int gridX0, gridY0;
    int P1x, P1y;
    int P2x, P2y;
    int nPoints = 0;

    public MyCanvas() {

        addMouseListener(this);
    }

    public void initCanvas() {

        Dimension d = getSize();
        maxX = d.width - 1;
        maxY = d.height - 1;
        gridX0 = GRID_SPACE;
        gridY0 = GRID_SPACE*(maxY/GRID_SPACE);
    }

    public void drawGrid(Graphics g) {

        int x, y;
        g.setColor(Color.black);
        for (x = GRID_SPACE; x < maxX; x += GRID_SPACE) {
            g.drawLine(x, GRID_SPACE,
                       x, GRID_SPACE*(maxY/GRID_SPACE));
        }
        for (y = GRID_SPACE; y < maxY; y += GRID_SPACE) {
            g.drawLine(GRID_SPACE, y,
                       GRID_SPACE*(maxX/GRID_SPACE), y);
        }
    }

    public void drawPixel(Graphics g, int x, int y) {

        // Draw the simulated pixel as a filled circle on the grid
        // Convert from grid coordinates to Canvas coordinates
```

```
    int xc = gridX0 + x * GRID_SPACE;
    int yc = gridY0 - y * GRID_SPACE;
    g.setColor(Color.black);
    g.fillOval(xc - PIXEL_RADIUS, yc - PIXEL_RADIUS,
               2 * PIXEL_RADIUS, 2 * PIXEL_RADIUS);
}

public void drawMidpointLine(Graphics g, int x1, int y1,
                             int x2, int y2) {
    int dx = Math.abs(x2 - x1);
    int dy = Math.abs(y2 - y1);
    int d;
    int deltaE = 2 * dy;
    int deltaN = 2 * dx;
    int deltaNE;
    if (dx > dy) {
        d = 2 * dy - dx;
        deltaNE = 2 * (dy - dx);
    } else {
        d = 2 * dx - dy;
        deltaNE = 2 * (dx - dy);
    }
    if (x1 > x2) {
        int t = x1; x1 = x2; x2 = t;
        t = y1; y1 = y2; y2 = t;
    }
    int x = x1;
    int y = y1;
    int yInc = y2 < y1 ? -1 : 1;
    drawPixel(g, x, y);
    if (dx > dy) {
        while (x != x2) {
            if (d <= 0) {
                d += deltaE;
            } else {
                d += deltaNE;
                y+=yInc;
            }
            x++;
            drawPixel(g, x, y);
        }
    } else {
        while (y != y2) {
            if (d <= 0) {
                d += deltaN;
            } else {
                d += deltaNE;
                x++;
            }
            y+=yInc;
            drawPixel(g, x, y);
```

```
                }
            }
        }

        public void paint(Graphics g) {

            initCanvas();
            drawGrid(g);
            if (nPoints == 1) {
                drawPixel(g, P1x, P1y);
            } else if (nPoints == 2) {
                // Draw ideal line
                int xIdeal1 = gridX0 + P1x * GRID_SPACE;
                int yIdeal1 = gridY0 - P1y * GRID_SPACE;
                int xIdeal2 = gridX0 + P2x * GRID_SPACE;
                int yIdeal2 = gridY0 - P2y * GRID_SPACE;
                g.setColor(Color.red);
                g.drawLine(xIdeal1, yIdeal1, xIdeal2, yIdeal2);
                drawMidpointLine(g, P1x, P1y, P2x, P2y);
            }
        }

        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {

            if (nPoints == 2) {
                nPoints = 0;
            }
            if (nPoints == 0) {
                // Get first end point of the line
                // Convert to grid coordinates
                P1x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
                P1y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
                nPoints++;
                repaint();
            } else if (nPoints == 1) {
                // Get second end point of the line
                // Convert to grid coordinates
                P2x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
                P2y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
                nPoints++;
                repaint();
            }
        }
    }
}
```

4.2.2 Midpoint Circle Algorithm

```
//
// Demonstration of the Midpoint Circle Algorithm
//

import java.awt.*;
import java.awt.event.*;

public class MidpointCircle extends Frame {

    public static void main(String[] args) {

        new MidpointCircle();
    }

    public MidpointCircle() {

        super("Demo of the Midpoint Circle Algorithm");
        addWindowListener(new WindowAdapter()
            {public void windowClosing(WindowEvent e){System.exit(0);}});
        setSize(500,510);
        add("Center", new MyCanvas());
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
        show();
    }

    public class MyCanvas extends Canvas implements MouseListener {

        final int GRID_SPACE = 15;
        final int PIXEL_RADIUS = 6;
        int maxX, maxY;
        int gridX0, gridY0;
        int P1x, P1y;
        int P2x, P2y;
        int R;
        int nPoints = 0;

        public MyCanvas() {

            addMouseListener(this);
        }

        public void initCanvas() {

            Dimension d = getSize();
            maxX = d.width - 1;
            maxY = d.height - 1;
            gridX0 = GRID_SPACE;
            gridY0 = GRID_SPACE*(maxY/GRID_SPACE);
        }
```

```
public void drawGrid(Graphics g) {

    int x, y;
    g.setColor(Color.black);
    for (x = GRID_SPACE; x < maxX; x += GRID_SPACE) {
        g.drawLine(x, GRID_SPACE,
                   x, GRID_SPACE*(maxY/GRID_SPACE));
    }
    for (y = GRID_SPACE; y < maxY; y += GRID_SPACE) {
        g.drawLine(GRID_SPACE, y,
                   GRID_SPACE*(maxX/GRID_SPACE), y);
    }
}

public void drawPixel(Graphics g, int x, int y) {

    // Draw the simulated pixel as a filled circle on the grid
    // Convert from grid coordinates to Canvas coordinates
    int xc = gridX0 + x * GRID_SPACE;
    int yc = gridY0 - y * GRID_SPACE;
    g.setColor(Color.black);
    g.fillOval(xc - PIXEL_RADIUS, yc - PIXEL_RADIUS,
               2 * PIXEL_RADIUS, 2 * PIXEL_RADIUS);
}

public void drawCirclePixels(Graphics g, int x, int y,
                             int xc, int yc) {

    drawPixel(g, xc + y, yc + x); // Octant 1
    drawPixel(g, xc + x, yc + y); // Octant 2
    drawPixel(g, xc - x, yc + y); // Octant 3
    drawPixel(g, xc - y, yc + x); // Octant 4
    drawPixel(g, xc - y, yc - x); // Octant 5
    drawPixel(g, xc - x, yc - y); // Octant 6
    drawPixel(g, xc + x, yc - y); // Octant 7
    drawPixel(g, xc + y, yc - x); // Octant 8
}

public void drawMidpointCircle(Graphics g,
                               int xc, int yc, int R) {

    int x = 0;
    int y = R;
    int d = 1 - R;
    drawCirclePixels(g, x, y, xc, yc);
    while (y > x) {
        if (d < 0) {
            d += 2 * x + 3;
            x++;
        } else {
```

```
            d += 2 * (x - y) + 5;
            x++;
            y--;
        }
        drawCirclePixels(g, x, y, xc, yc);
    }
}

public void paint(Graphics g) {

    initCanvas();
    drawGrid(g);
    if (nPoints == 1) {
        drawPixel(g, P1x, P1y);
    } else if (nPoints == 2) {
        // Draw ideal circle
        int xIdeal = gridX0 + P1x * GRID_SPACE;
        int yIdeal = gridY0 - P1y * GRID_SPACE;
        int rIdeal = R * GRID_SPACE;
        g.setColor(Color.red);
        g.drawOval(xIdeal - rIdeal, yIdeal - rIdeal,
                   2 * rIdeal, 2 * rIdeal);
        drawPixel(g, P1x, P1y);
        drawMidpointCircle(g, P1x, P1y, R);
    }
}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mousePressed(MouseEvent e) {

    if (nPoints == 2) {
        nPoints = 0;
    }
    if (nPoints == 0) {
        // Get center of circle
        // Convert to grid coordinates
        P1x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
        P1y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
        nPoints++;
        repaint();
    } else if (nPoints == 1) {
        // Get a point, compute integer radius
        // Convert to grid coordinates
        P2x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
        P2y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
        float dx = P2x - P1x;
        float dy = P2y - P1y;
        R = Math.round((float)Math.sqrt(dx * dx + dy * dy));
```

```
                nPoints++;
                repaint();
            }
        }
    }
}
```

4.2.3 Midpoint Ellipse Algorithm

```
//
// Demonstration of the Midpoint Ellipse Algorithm
//

import java.awt.*;
import java.awt.event.*;

public class MidpointEllipse extends Frame {

    public static void main(String[] args) {

        new MidpointEllipse();
    }

    public MidpointEllipse() {

        super("Demo of the Midpoint Ellipse Algorithm");
        addWindowListener(new WindowAdapter()
            {public void windowClosing(WindowEvent e){System.exit(0);}});
        setSize(500,510);
        add("Center", new MyCanvas());
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
        show();
    }

    public class MyCanvas extends Canvas implements MouseListener {

        final int GRID_SPACE = 15;
        final int PIXEL_RADIUS = 6;
        int maxX, maxY;
        int gridX0, gridY0;
        int P1x, P1y;
        int P2x, P2y;
        int a, b;
        int nPoints = 0;

        public MyCanvas() {

            addMouseListener(this);
        }

        public void initCanvas() {
```

```
    Dimension d = getSize();
    maxX = d.width - 1;
    maxY = d.height - 1;
    gridX0 = GRID_SPACE;
    gridY0 = GRID_SPACE*(maxY/GRID_SPACE);
}

public void drawGrid(Graphics g) {

    int x, y;
    g.setColor(Color.black);
    for (x = GRID_SPACE; x < maxX; x += GRID_SPACE) {
        g.drawLine(x, GRID_SPACE,
                   x, GRID_SPACE*(maxY/GRID_SPACE));
    }
    for (y = GRID_SPACE; y < maxY; y += GRID_SPACE) {
        g.drawLine(GRID_SPACE, y,
                   GRID_SPACE*(maxX/GRID_SPACE), y);
    }
}

public void drawPixel(Graphics g, int x, int y) {

    // Draw the simulated pixel as a filled circle on the grid
    // Convert from grid coordinates to Canvas coordinates
    int xc = gridX0 + x * GRID_SPACE;
    int yc = gridY0 - y * GRID_SPACE;
    g.setColor(Color.black);
    g.fillOval(xc - PIXEL_RADIUS, yc - PIXEL_RADIUS,
               2 * PIXEL_RADIUS, 2 * PIXEL_RADIUS);
}

public void drawEllipsePixels(Graphics g, int x, int y,
                              int xc, int yc) {

    drawPixel(g, xc + x, yc + y); // Quadrant 1
    drawPixel(g, xc - x, yc + y); // Quadrant 2
    drawPixel(g, xc - x, yc - y); // Quadrant 3
    drawPixel(g, xc + x, yc - y); // Quadrant 4
}

public void drawMidpointEllipse(Graphics g, int xc, int yc,
                                int a, int b) {

    int x = 0;
    int y = b;
    int a2 = a * a;
    int b2 = b * b;
    double d1 = b2 - a2*b + 0.25*a2;
    drawEllipsePixels(g, x, y, xc, yc);
```

```
    while (a2*(y-0.5) > b2*(x+1)) {
        if (d1 < 0) {
            d1 += b2*(2*x+3);
            x++;
        } else {
            d1 += b2*(2*x+3)+a2*(2-2*y);
            x++;
            y--;
        }
        drawEllipsePixels(g, x, y, xc, yc);
    }
    double d2 = b2*(x+0.5)*(x+0.5) + a2*(y-1)*(y-1) - a2*b2;
    while(y > 0) {
        if (d2 < 0) {
            d2 += b2*(2*x+2)+a2*(3-2*y);
            x++;
            y--;
        } else {
            d2 += a2*(3-2*y);
            y--;
        }
        drawEllipsePixels(g, x, y, xc, yc);
    }
}

public void paint(Graphics g) {

    initCanvas();
    drawGrid(g);
    if (nPoints == 1) {
        drawPixel(g, P1x, P1y);
    } else if (nPoints == 2) {
        // Draw ideal ellipse
        int xIdeal = gridX0 + P1x * GRID_SPACE;
        int yIdeal = gridY0 - P1y * GRID_SPACE;
        int aIdeal = Math.abs(P1x - P2x) * GRID_SPACE;
        int bIdeal = Math.abs(P1y - P2y) * GRID_SPACE;
        g.setColor(Color.red);
        g.drawOval(xIdeal-aIdeal, yIdeal-bIdeal,
                   2*aIdeal, 2*bIdeal);

        int a = Math.abs(P1x - P2x);
        int b = Math.abs(P1y - P2y);
        drawMidpointEllipse(g, P1x, P1y, a, b);
    }
}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
```

```
        public void mousePressed(MouseEvent e) {

            if (nPoints == 2) {
                nPoints = 0;
            }
            if (nPoints == 0) {
                // Get coordinates of the center
                // Convert to grid coordinates
                P1x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
                P1y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
                nPoints++;
                repaint();
            } else if (nPoints == 1) {
                // Get coordinates of a point
                // of the bounding box
                // Convert to grid coordinates
                P2x = Math.round((e.getX()-gridX0)/((float)GRID_SPACE));
                P2y = -Math.round((e.getY()-gridY0)/((float)GRID_SPACE));
                nPoints++;
                repaint();
            }
        }
    }
}
```

4.3 Algorithm for Bézier Curve

```
//
// Representation of a 2D Point
//

public class Point2D {

    float x;
    float y;

    public Point2D(float x, float y) {

        this.x = x;
        this.y = y;
    }
}
```

```
//
// Representation of a Bezier Curve
//

import java.awt.*;

public class Bezier {

    private Point2D[] ctrlPoints;
    private float xExtent, yExtent, pixelSize;
    private int centerX, centerY, nPoints;
    private Graphics g;
    private Bezier left, right;

    public Bezier(Graphics g, Point2D[] ctrlPoints,
                  float pixelSize, int centerX, int centerY) {

        this.ctrlPoints = ctrlPoints;
        this.nPoints = ctrlPoints.length;
        this.g = g;
        this.pixelSize = pixelSize;
        this.centerX = centerX;
        this.centerY = centerY;
        float xMin, xMax, yMin, yMax;
        Point2D P = ctrlPoints[0];
        xMin = xMax = P.x;
        yMin = yMax = P.y;
        for (int i = 1; i < nPoints; i++) {
            P = ctrlPoints[i];
            xMin = P.x < xMin ? P.x : xMin;
            yMin = P.y < yMin ? P.y : yMin;
            xMax = P.x > xMax ? P.x : xMax;
            yMax = P.y > yMax ? P.y : yMax;
        }
        this.xExtent = Math.abs(xMax - xMin);
        this.yExtent = Math.abs(yMax - yMin);
    }

    public void draw(float limit) {

        Point2D P1 = ctrlPoints[0];
        Point2D P2 = ctrlPoints[nPoints-1];
        if (nPoints == 2 || xExtent < limit && yExtent < limit) {
            g.drawLine(intX(P1.x),
                       intY(P1.y),
                       intX(P2.x),
                       intY(P2.y));
        } else {
            subdivide();
            left.draw(limit);
            right.draw(limit);
```

```
        }
    }

    private int intX(float x) {
        return Math.round(centerX + x/pixelSize);
    }

    private int intY(float y) {
        return Math.round(centerY - y/pixelSize);
    }

    private void subdivide() {

        int i, j;
        Point2D[] leftPoints = new Point2D[nPoints];
        Point2D[] rightPoints = new Point2D[nPoints];
        Point2D[] tempPoints = new Point2D[nPoints];
        Point2D P, P1, P2;
        for (i = 0; i < nPoints; i++) {
            P = ctrlPoints[i];
            tempPoints[i] = new Point2D(P.x, P.y);
        }
        P = tempPoints[0];
        leftPoints[0] = new Point2D(P.x, P.y);
        P = tempPoints[nPoints-1];
        rightPoints[nPoints-1] = new Point2D(P.x, P.y);
        for (j = 1; j < nPoints; j++) {
            for (i = 0; i < nPoints - j; i++) {
                tempPoints[i].x += tempPoints[i+1].x;
                tempPoints[i].x *= 0.5;
                tempPoints[i].y += tempPoints[i+1].y;
                tempPoints[i].y *= 0.5;
            }
            P1 = tempPoints[0];
            leftPoints[j] = new Point2D(P1.x, P1.y);
            P2 = tempPoints[nPoints-1-j];
            rightPoints[nPoints-1-j] = new Point2D(P2.x, P2.y);
        }
        left = new Bezier(g, leftPoints, pixelSize, centerX, centerY);
        right = new Bezier(g, rightPoints, pixelSize, centerX, centerY);
    }
}
```

```
//
// Demonstration of Rendering Bezier Curves
//

import java.awt.*;
import java.awt.event.*;

public class BezierDemo extends Frame {

    private Bezier curve;

    public static void main(String[] args) {

        new BezierDemo();
    }

    public BezierDemo() {

        super("Demo of Bezier Curves");
        addWindowListener(new WindowAdapter()
            {public void windowClosing(WindowEvent e){System.exit(0);}});
        setSize(500,500);
        add("Center", new MyCanvas());
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
        show();
    }

    public class MyCanvas extends Canvas implements MouseListener {

        final static float REAL_X = 10.0f;
        final static float REAL_Y = 10.0f;
        final static int MAX_POINTS = 20;
        float pixelSize;
        int maxX, maxY;
        int centerX, centerY;
        Point2D[] ctrlPoints = new Point2D[MAX_POINTS];
        int nPoints = 0;
        boolean lastPoint = false;

        public MyCanvas() {

            addMouseListener(this);
        }

        public void initCanvas() {

            Dimension d = getSize();
            maxX = d.width - 1;
            maxY = d.height - 1;
            centerX = maxX / 2;
            centerY = maxY / 2;
```

```
    pixelSize = Math.max(REAL_X/maxX, REAL_Y/maxY);
}

public void paint(Graphics g) {

    int i;
    Point2D P;
    initCanvas();
    g.setColor(Color.black);
    for (i = 0; i < nPoints; i++) {
        P = ctrlPoints[i];
        g.drawRect(intX(P.x)-2,
                   intY(P.y)-2,4,4);
        if (i > 0) {
            g.drawLine(intX(ctrlPoints[i-1].x),
                       intY(ctrlPoints[i-1].y),
                       intX(ctrlPoints[i].x),
                       intY(ctrlPoints[i].y));
        }
    }
    if (nPoints > 1 && lastPoint) {
        g.setColor(Color.red);
        Point2D[] points = new Point2D[nPoints];
        for (i = 0; i < nPoints; i++)
            points[i] = ctrlPoints[i];
        curve = new Bezier(g, points, pixelSize,
                           centerX, centerY);
        curve.draw(pixelSize*2);
    }
}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mousePressed(MouseEvent e) {

    if (lastPoint) {
        lastPoint = false;
        nPoints = 0;
    }
    if (nPoints<MAX_POINTS) {
        Point2D P = new Point2D(floatX(e.getX()),
                                floatY(e.getY()));
        ctrlPoints[nPoints] = P;
        nPoints++;
    }
    if (e.getButton() == MouseEvent.BUTTON2) {
        lastPoint = true;
    }
    repaint();
```

```
        }

        public int intX(float x) {
            return Math.round(centerX + x/pixelSize);
        }

        public int intY(float y) {
            return Math.round(centerY - y/pixelSize);
        }

        public float floatX(int x) {
            return (x - centerX) * pixelSize;
        }

        public float floatY(int y) {
            return -(y - centerY) * pixelSize;
        }
    }
}
```

Conclusion

Computer graphics is a broad subject, and during the work with this paper I have only scratched the surface. I have focused on the mathematics used for representing and transforming three-dimensional objects. There is a lot to write about computer graphics algorithms, which are also very interesting. The developments in this area are rapid, especially of computer graphics hardware. Computer graphics are used in many different areas, such as CAD systems, computer games, movies, and flight simulators.

During the work I have learned a lot, but I have also realized that there is a lot more to learn about computer graphics. There are many good books about the subject. The books listed in the bibliography on page 105 serve as a good starting point.

Bibliography

[1] Foley, James D et al, *Computer Graphics: Principles and Practice*. 2nd ed. in C. Addison-Wesley. 2000.

[2] Hill, Francis S., *Computer Graphics*. Macmillan. 1990.

[3] Marsh, Duncan, *Applied Geometry for Computer Graphics and CAD*. Springer. 1999.

[4] Mortenson, Michael E., *Geometric modeling*. 2nd ed. John Wiley & sons. 1997.

[5] Roe, John, *Elementary Geometry*. Oxford. 1995.

[1] is a very good book, containing various aspects of computer graphics. It is useful both as an introduction, but also as an advanced text. It covers representation of graphical objects, rendering algorithms, design of user interfaces, and the history of computer graphics. [2] is also very good. It is not as complete as [1], but perphaps easier to understand. [3] is a good introduction to Bézier curves and B-splines. [4] describes parametric curves, surfaces, and solids in general.

Biography

My name is Lennart Jordansson, and I grew up in Säffle in Värmland. I received the degree of M. Sc. (civilingenjör) in computer science and electrical engineering at Luleå University of Technology in 1999. I have worked as a hardware and software engineer in Karlstad for about four years. My interests include electronics, computer science, mathematics, and computer graphics. I went back to Luleå to study mathematics in 2002, and this work is a result of combining some of my interests. It has been a great opportunity to learn more about computer graphics and mathematics.