Fundamentals of Computer Graphics - CM20219

Lecture Notes

Dr John Collomosse

University of Bath, UK

SUMMARY

Welcome to CM20219 — Fundamentals of Computer Graphics. This course aims to provide both the theoretical and practical skills to begin developing Computer Graphics software.

These notes are a supplementary resource to the lectures and lab sessions. They should not be regarded as a complete account of the course content. In particular Chapter 4 provides only very introductory material to OpenGL and further resources (e.g. Powerpoint slides) handed out in lectures should be referred to. Furthermore it is not possible to pass the course simply by learning these notes; you must be able to apply the techniques within to produce viable Computer Graphics solutions.

All Computer Graphics demands a working knowledge of linear algebra (matrix manipulation, linear systems, etc.). A suitable background is provided by first year unit CM10197, and Chapter 1 provides a brief revision on some key mathematics. This course also requires a working knowledge of the C language, such as that provided in CM20214 (runs concurrently with this course).

The course and notes focus on four key topics:

1. **Image Representation**
   How digital images are represented in a computer. This 'mini'-topic explores different forms of frame-buffer for storing images, and also different ways of representing colour and key issues that arise in colour (Chapter 2)

2. **Geometric Transformation**
   How to use linear algebra, e.g. matrix transformations, to manipulate points in space (Chapter 3). This work focuses heavily on the concept of reference frames and their central role in Computer Graphics. Also on this theme, eigenvalue decomposition is discussed and a number of applications relating to visual computing are explored (Chapter 5).

3. **OpenGL Programming**
   Discusses how the mathematics on this course can be implemented directly in the C programming language using the OpenGL library (Chapter 4). Note much of this content is covered in Powerpoint handouts rather than these notes.

4. **Geometric Modelling**
   Whereas Chapters 3,5 focus on manipulating/positioning of points in 3D space, this Chapter explores how these points can be "joined up" to form curves and surfaces. This allows the modelling of objects and their trajectories (Chapter 6).

This course and the accompanying notes, Moodle pages, lab sheets, exams, and coursework were developed by John Collomosse in 2008/9. If you notice any errors please contact `jpc@cs.bath.ac.uk`. Please refer to Moodle for an up to date list of supporting texts for this course.

Contents

1 Mathematical Background **1**
1.1 Introduction 1
1.2 Points, Vectors and Notation 1
1.3 Basic Vector Algebra 1
1.3.1 Vector Addition 2
1.3.2 Vector Subtraction 2
1.3.3 Vector Scaling 2
1.3.4 Vector Magnitude 2
1.3.5 Vector Normalisation 3
1.4 Vector Multiplication 3
1.4.1 Dot Product 3
1.4.2 Cross Product 4
1.5 Reference Frames 5
1.6 Cartesian vs. Radial-Polar Form 6
1.7 Matrix Algebra 6
1.7.1 Matrix Addition 6
1.7.2 Matrix Scaling 7
1.7.3 Matrix Multiplication 7
1.7.4 Matrix Inverse and the Identity 7
1.7.5 Matrix Transposition 8

2 Image Representation **9**
2.1 Introduction 9
2.2 The Digital Image 9
2.2.1 Raster Image Representation 9
2.2.2 Hardware Frame Buffers 10
2.2.3 Greyscale Frame Buffer 11
2.2.4 Pseudo-colour Frame Buffer 11
2.2.5 True-Colour Frame Buffer 12
2.3 Representation of Colour 13
2.3.1 Additive vs. Subtractive Primaries 14
2.3.2 RGB and CMYK colour spaces 15
2.3.3 Greyscale Conversion 16
2.3.4 Can any colour be represented in RGB space? 18
2.3.5 CIE colour space 18
2.3.6 Hue, Saturation, Value (HSV) colour space 19
2.3.7 Choosing an appropriate colour space 21

3 Geometric Transformation **23**
3.1 Introduction 23
3.2 2D Rigid Body Transformations 23
3.2.1 Scaling 24
3.2.2 Shearing (Skewing) 24
3.2.3 Rotation 25
3.2.4 Active vs. Passive Interpretation 26
3.2.5 Transforming between basis sets 27
3.2.6 Translation and Homogeneous Coordinates 28
3.2.7 Compound Matrix Transformations 30
3.2.8 Animation Hierarchies 32
3.3 3D Rigid Body Transformations 34
3.3.1 Rotation in 3D — Euler Angles 35
3.3.2 Rotation about an arbitrary axis in 3D 36
3.3.3 Problems with Euler Angles 38
3.4 Image Formation – 3D on a 2D display 40
3.4.1 Perspective Projection 40
3.4.2 Orthographic Projection 43
3.5 Homography 44
3.5.1 Applications to Image Stitching 45
3.6 Digital Image Warping 46

4 OpenGL Programming **50**
4.1 Introduction 50
4.1.1 The GLUT Library 50
4.2 An Illustrative Example - Teapot 51
4.2.1 Double Buffering and Flushing 53
4.2.2 Why doesn't it look 3D? 53
4.3 Modelling and Matrices in OpenGL 53
4.3.1 The Matrix Stack 54
4.4 A Simple Animation - Spinning Teapot 55
4.5 Powerpoint 57

5 Eigenvalue Decomposition and its Applications in Computer Graphics **58**
5.1 Introduction 58
5.1.1 What is EVD? 58
5.2 How to compute EVD of a matrix 59
5.2.1 Characteristic Polynomial: Solving for the eigenvalues 59
5.2.2 Solving for the eigenvectors 60
5.3 How is EVD useful? 61
5.3.1 EVD Application 1: Matrix Diagonalisation 61
5.3.2 EVD Application 2: Principle Component Analysis (PCA) 62
5.4 An Introduction to Pattern Recognition 63
5.4.1 A Simple Colour based Classifier 63
5.4.2 Feature Spaces 65
5.4.3 Distance Metrics 66
5.4.4 Nearest-Mean Classification 66
5.4.5 Eigenmodel Classification 69

5.5 Principle Component Analysis (PCA) 72
5.5.1 Recap: Computing PCA 73
5.5.2 PCA for Visualisation 73
5.5.3 Decomposing non-square matrices (SVD) 75

6 Geometric Modelling **77**
6.1 Lines and Curves 77
6.1.1 Explicit, Implicit and Parametric forms 77
6.1.2 Parametric Space Curves 79
6.2 Families of Curve 81
6.2.1 Hermite Curves 81
6.2.2 Bézier Curve 83
6.2.3 Catmull-Rom spline 85
6.2.4 β-spline 87
6.3 Curve Parameterisation 87
6.3.1 Frenet Frame 88
6.4 Surfaces 90
6.4.1 Planar Surfaces 91
6.4.2 Ray Tracing with Implicit Planes 92
6.4.3 Curved surfaces 94
6.4.4 Bi-cubic surface patches 95

Chapter 1

Mathematical Background

1.1 Introduction

The taught material in this course draws upon a mathematical background in linear algebra. We briefly revise some of the basics here, before beginning the course material in Chapter 2.

1.2 Points, Vectors and Notation

Much of Computer Graphics involves discussion of points in 2D or 3D. Usually we write such points as **Cartesian Coordinates** e.g. $\underline{p} = [x, y]^T$ or $\underline{q} = [x, y, z]^T$. Point coordinates are therefore **vector** quantities, as opposed to a single number e.g. 3 which we call a **scalar** quantity. In these notes we write vectors in bold and underlined once. Matrices are written in bold, double-underlined.

The superscript $[...]^T$ denotes transposition of a vector, so points $\underline{p}$ and $\underline{q}$ are **column vectors** (coordinates stacked on top of one another vertically). This is the convention used by most researchers with a Computer Vision background, and is the convention used throughout this course. By contrast, many Computer Graphics researchers use **row vectors** to represent points. For this reason you will find row vectors in many Graphics textbooks including Foley *et al*, one of the course texts. Bear in mind that you can convert equations between the two forms using transposition. Suppose we have a 2×2 matrix $\underline{\underline{M}}$ acting on the 2D point represented by column vector $\underline{p}$. We would write this as $\underline{\underline{M}}\underline{p}$.

If $\underline{p}$ was transposed into a row vector $\underline{p'} = \underline{p}^T$, we could write the above transformation $\underline{p'}\underline{\underline{M}}^T$. So to convert between the forms (e.g. from row to column form when reading the course-texts), remember that:

$$\underline{\underline{M}}\underline{p} = (\underline{p}^T \underline{\underline{M}}^T)^T \tag{1.1}$$

For a reminder on matrix transposition please see subsection 1.7.5.

1.3 Basic Vector Algebra

Just as we can perform basic operations such as addition, multiplication etc. on scalar values, so we can generalise such operations to vectors. Figure 1.1 summarises some of these operations in diagrammatic form.

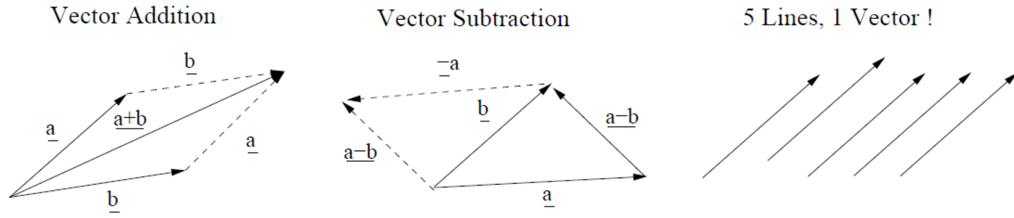


Figure 1.1: Illustrating vector addition (left) and subtraction (middle). Right: Vectors have direction and magnitude; lines (sometimes called 'rays') are vectors plus a starting point.

1.3.1 Vector Addition

When we add two vectors, we simply sum their elements at corresponding positions. So for a pair of 2D vectors $\underline{a} = [u, v]^T$ and $\underline{b} = [s, t]^T$ we have:

$$\underline{a} + \underline{b} = [u + s, v + t]^T \tag{1.2}$$

1.3.2 Vector Subtraction

Vector subtraction is identical to the addition operation with a sign change, since when we negate a vector we simply flip the sign on its elements.

$$\begin{aligned} -\underline{b} &= [-s, -t]^T \\ \underline{a} - \underline{b} &= \underline{a} + (-\underline{b}) = [u - s, v - t]^T \end{aligned} \tag{1.3}$$

1.3.3 Vector Scaling

If we wish to increase or reduce a vector quantity by a scale factor λ then we multiply each element in the vector by λ.

$$\lambda\underline{a} = [\lambda u, \lambda v]^T \tag{1.4}$$

1.3.4 Vector Magnitude

We write the length of **magnitude** of a vector $\underline{s}$ as $|s|$. We use Pythagoras' theorem to compute the magnitude:

$$|\underline{a}| = \sqrt{u^2 + v^2} \tag{1.5}$$

Figure 1.3 shows this to be valid, since u and v are distances along the principal axes (x and y) of the space, and so the distance of $\underline{a}$ from the origin is the hypotenuse of a right-angled triangle. If we have an n-dimensional vector $\underline{q} = [q_1, q_2, q_3, q_{...}, q_n]$ then the definition of vector magnitude generalises to:

$$|\underline{q}| = \sqrt{q_1^2 + q_2^2 + q_{...}^2 + q_n^2} = \sqrt{\sum_{i=1}^{n} q_i^2} \tag{1.6}$$

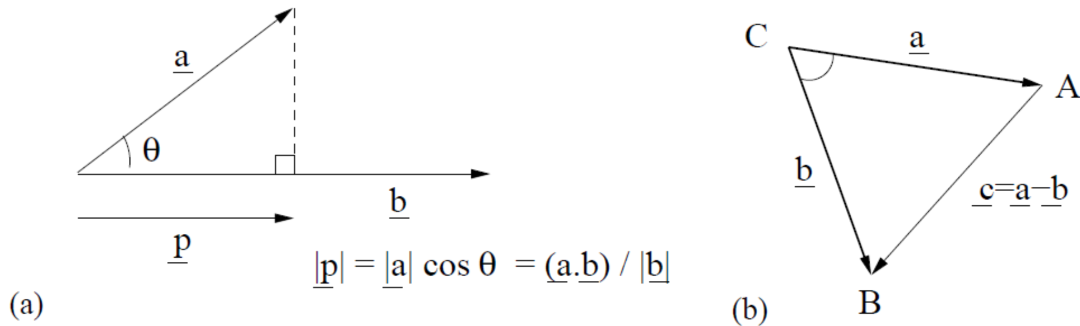


Figure 1.2: (a) Demonstrating how the dot product can be used to measure the component of one vector in the direction of another (i.e. a projection, shown here as $\underline{p}$). (b) The geometry used to prove $\underline{a} \circ \underline{b} = |\underline{a}||\underline{b}|cos\theta$ via the Law of Cosines in equation 1.11.

1.3.5 Vector Normalisation

We can normalise a vector $\underline{a}$ by scaling it by the reciprocal of its magnitude:

$$\hat{\underline{a}} = \frac{\underline{a}}{|\underline{a}|} \tag{1.7}$$

This produces a **normalised vector** pointing in the same direction as the original (un-normalised) vector, but with **unit length** (i.e. length of 1). We use the superscript 'hat' notation to indicate that a vector is normalised e.g. $\hat{\underline{a}}$.

1.4 Vector Multiplication

We can define multiplication of a pair of vectors in two ways: the **dot product** (sometimes called the 'inner product', analogous to matrix multiplication), and the **cross product** (which is sometimes referred to by the unfortunately ambiguous term 'vector product').

1.4.1 Dot Product

The **dot product** sums the products of corresponding elements over a pair of vectors. Given vectors $\underline{a} = [a_1, a_2, a_3, a_{...}, a_n]^T$ and $\underline{b} = [b_1, b_2, b_3, b_{...}, b_n]^T$, the dot product is defined as:

$$\begin{aligned} \underline{a} \circ \underline{b} &= a_1b_1 + a_2b_2 + a_3b_3 + ... + a_nb_n \\ &= \sum_{i=1}^{n} a_ib_i \end{aligned} \tag{1.8}$$

The dot product is both symmetric and positive definite. It gives us a *scalar value* that has three important uses in Computer Graphics and related fields. First, we can compute the square of the magnitude of a vector by taking the dot product of that vector and itself:

$$\begin{aligned} \underline{a} \circ \underline{a} &= a_1a_1 + a_2a_2 + ... + a_na_n \\ &= \sum_{i=1}^{n} a_i^2 \\ &= |\underline{a}|^2 \end{aligned} \tag{1.9}$$

Second, we can more generally compute $\underline{a} \circ \underline{b}$, the magnitude of one vector $\underline{a}$ in the direction of another $\underline{b}$, i.e. projecting one vector onto another. Figure 1.2a illustrates how a simple rearrangement of equation 1.10 can achieve this.

Third, we can use the dot product to compute the angle θ between two vectors (if we normalise them first). This relationship can be used to define the concept of an angle between vectors in $n-$dimensional spaces. It is also fundamental to most lighting calculations in Graphics, enabling us to determine the angle of a surface (normal) to a light source.

$$\underline{a} \circ \underline{b} = |\underline{a}||\underline{b}|cos\theta \tag{1.10}$$

A proof follows from the "law of cosines", a general form of Pythagoras' theorem. Consider triangle ABC in Figure 1.2b, with respect to equation 1.10. Side $\vec{CA}$ is analogous to vector $\underline{a}$, and side $\vec{CB}$ analogous to vector $\underline{b}$. θ is the angle between $\vec{CA}$ and $\vec{CB}$, and so also vectors $\underline{a}$ and $\underline{b}$.

$$|\underline{c}|^2 = |\underline{a}|^2 + |\underline{b}|^2 - 2|\underline{a}||\underline{b}|cos\theta \tag{1.11}$$

$$\underline{c} \circ \underline{c} = \underline{a} \circ \underline{a} + \underline{b} \circ \underline{b} - 2|\underline{a}||\underline{b}| \cos\theta \tag{1.12}$$

Now consider that $\underline{c} = \underline{a} - \underline{b}$ (refer back to Figure 1.1):

$$(\underline{a} - \underline{b}) \circ (\underline{a} - \underline{b}) = \underline{a} \circ \underline{a} + \underline{b} \circ \underline{b} - 2|\underline{a}||\underline{b}| \cos\theta \tag{1.13}$$

$$\underline{a} \circ \underline{a} - 2(\underline{a} \circ \underline{b}) + \underline{b} \circ \underline{b} = \underline{a} \circ \underline{a} + \underline{b} \circ \underline{b} - 2|\underline{a}||\underline{b}| \cos\theta \tag{1.14}$$

$$-2(\underline{a} \circ \underline{b}) = -2|\underline{a}||\underline{b}| \cos\theta \tag{1.15}$$

$$\underline{a} \circ \underline{b} = |\underline{a}||\underline{b}| \cos\theta \tag{1.16}$$

Another useful result is that we can quickly test for the orthogonality of two vectors by checking if their dot product is zero.

1.4.2 Cross Product

Taking the **cross product** (or "**vector product**") of two vectors returns us a vector orthogonal to those two vectors. Given two vectors $\underline{a} = [a_x, a_y, a_z]^T$ and $\underline{B} = [b_x, b_y, b_z]^T$, the cross product $\underline{a} \times \underline{b}$ is defined as:

$$\underline{a} \times \underline{b} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix} \tag{1.17}$$

This is often remembered using the mnemonic '**xyzzy**'. In this course we only consider the definition of the cross product in 3D. An important Computer Graphics application of the cross product is to determine a vector that is orthogonal to its two inputs. This vector is said to be **normal** to those inputs, and is written $\underline{n}$ in the following relationship (care: note the normalisation):

$$\underline{a} \times \underline{b} = |\underline{a}||\underline{b}| \sin\theta \hat{\underline{n}} \tag{1.18}$$

A proof is beyond the requirements of this course.

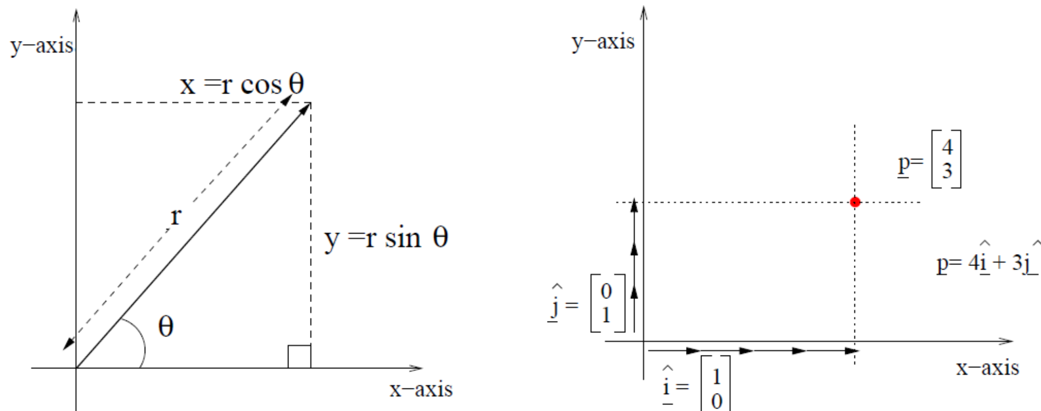


Figure 1.3: Left: Converting between Cartesian (x, y) and radial-polar (r, θ) form. We treat the system as a right-angled triangle and apply trigonometry. Right: Cartesian coordinates are defined with respect to a reference frame. The reference frame is defined by basis vectors (one per axis) that specify how 'far' and in what direction the units of each coordinate will take us.

1.5 Reference Frames

When we write down a point in Cartesian coordinates, for example $\underline{p} = [3, 2]^T$, we interpret that notation as "the point $\underline{p}$ is 3 units from the origin travelling in the positive direction of the x axis, and 2 units from the origin travelling in the positive direction of the y axis". We can write this more generally and succinctly as:

$$\underline{p} = x\hat{\underline{i}} + y\hat{\underline{j}} \tag{1.19}$$

where $\hat{\underline{i}} = [1, 0]^T$ and $\hat{\underline{j}} = [0, 1]^T$. We call $\hat{\underline{i}}$ and $\hat{\underline{j}}$ the **basis vectors** of the Cartesian space, and together they form the **basis set** of that space. Sometimes we use the term **reference frame** to refer to the coordinate space, and we say that the basis set $(\hat{\underline{i}}, \hat{\underline{j}})$ therefore defines the reference frame (Figure 1.3, right).

Commonly when working with 2D Cartesian coordinates we work in the reference frame defined by $\hat{\underline{i}} = [1, 0]^T$, $\hat{\underline{j}} = [0, 1]^T$. However other choices of basis vector are equally valid, so long as the basis vectors are neither parallel not **anti-parallel** (do not point in the same direction). We refer to our 'standard' reference frame ($\hat{\underline{i}} = [1, 0]^T, \hat{\underline{j}} = [0, 1]^T$) as the **root reference frame**, because we define the basis vectors of 'non-standard' reference frames with respect to it.

For example a point $[2, 3]^T$ defined in reference frame ($\underline{i} = [2, 0]^T, \underline{j} = [0, 2]^T$) would have coordinates $[4, 6]^T$ in our root reference frame. We will return to the matter of converting between reference frames in Chapter 3, as the concept underpins a complete understanding of geometric transformations.

1.6 Cartesian vs. Radial-Polar Form

We have so far recapped on Cartesian coordinate systems. These describe vectors in terms of distance along each of the **principal axes** (e.g. x,y) of the space. This **Cartesian form** is by far the most common way to represent vector quantities, like the location of points in space.

Sometimes it is preferable to define vectors in terms of length, and their orientation. This is called **radial-polar form** (often simply abbreviated to 'polar form'). In the case of 2D point locations, we describe the point in terms of: (a) its distance from the origin (r), and (b) the angle (θ) between a vertical line (pointing in the positive direction of the y axis), and the line subtended from the point to the origin (Figure 1.3).

To convert from Cartesian form $[x, y]^T$ to polar form (r, θ) we consider a right-angled triangle of side x and y (Figure 1.3, left). We can use Pythagoras' theorem to determine the length of hypotenuse r, and some basic trigonometry to reveal that $\theta = tan(y/x)$ and so:

$$r = \sqrt{x^2 + y^2} \tag{1.20}$$

$$\theta = atan(y/x) \tag{1.21}$$

To convert from polar form to Cartesian form we again apply some trigonometry (Figure 1.3, left):

$$x = r\cos\theta \tag{1.22}$$

$$y = r\sin\theta \tag{1.23}$$

1.7 Matrix Algebra

A matrix is a rectangular array of numbers. Both vectors and scalars are degenerate forms of matrices. By convention we say that an $(n \times m)$ matrix has n rows and m columns; i.e. we write ($height \times width$). In this subsection we will use two 2×2 matrices for our examples:

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

$$\underline{\underline{B}} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{1.24}$$

Observe that the notation for addressing an individual element of a matrix is $x_{row,column}$.

1.7.1 Matrix Addition

Matrices can be added, if they are of the same size. This is achieved by summing the elements in one matrix with corresponding elements in the other matrix:

$$\underline{\underline{A}} + \underline{\underline{B}} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} (a_{11} + b_{11}) & (a_{12} + b_{12}) \\ (a_{21} + b_{21}) & (a_{22} + b_{22}) \end{bmatrix} \tag{1.25}$$

This is identical to vector addition.

1.7.2 Matrix Scaling

Matrices can also be scaled by multiplying each element in the matrix by a scale factor. Again, this is identical to vector scaling.

$$s\underline{\underline{A}} = \begin{bmatrix} sa_{11} & sa_{12} \\ sa_{21} & sa_{22} \end{bmatrix} \tag{1.26}$$

1.7.3 Matrix Multiplication

As we will see in Chapter 3, matrix multiplication is a cornerstone of many useful geometric transformations in Computer Graphics. You should ensure that you are familiar with this operation.

$$\begin{aligned} \underline{\underline{AB}} &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ &= \begin{bmatrix} (a_{11}b_{11} + a_{12}b_{21}) & (a_{11}b_{12} + a_{12}b_{22}) \\ (a_{21}b_{11} + a_{22}b_{21}) & (a_{21}b_{21} + a_{22}b_{22}) \end{bmatrix} \end{aligned} \tag{1.27}$$

In general each element c_{ij} of the matrix $\underline{\underline{C}} = \underline{\underline{AB}}$, where $\underline{\underline{A}}$ is of size $(n \times P)$ and $\underline{\underline{B}}$ is of size $(P \times m)$ has the form:

$$c_{ij} = \sum_{k=1}^{P} a_{ik} b_{kj} \tag{1.28}$$

Not all matrices are compatible for multiplication. In the above system, $\underline{\underline{A}}$ must have as many columns as $\underline{\underline{B}}$ has rows. Furthermore, matrix multiplication is **non-commutative**, which means that $\underline{\underline{BA}} \neq \underline{\underline{AB}}$, in general. Given equation 1.27 you might like to write out the multiplication for $\underline{\underline{BA}}$ to satisfy yourself of this.

Finally, matrix multiplication is **associative** i.e.:

$$\underline{\underline{ABC}} = (\underline{\underline{AB}})\underline{\underline{C}} = \underline{\underline{A}}(\underline{\underline{BC}}) \tag{1.29}$$

If the matrices being multiplied are of different (but compatible) sizes, then the complexity of evaluating such an expression varies according to the order of multiplication[1].

1.7.4 Matrix Inverse and the Identity

The **identity matrix** $\underline{\underline{I}}$ is a special matrix that behaves like the number 1 when multiplying scalars (i.e. it has no numerical effect):

$$\underline{\underline{IA}} = \underline{\underline{A}} \tag{1.30}$$

The identity matrix has zeroes everywhere except the **leading diagonal** which is set to 1, e.g. the (2×2) identity matrix is:

$$\underline{\underline{I}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{1.31}$$

[1] Finding an optimal permutation of multiplication order is a (solved) interesting optimization problem, but falls outside the scope of this course (see Rivest *et al.* text on Algorithms)

The identity matrix leads us to a definition of the **inverse of a matrix**, which we write $\underline{\underline{A}}^{-1}$. The inverse of a matrix, when pre- or post-multiplied by its original matrix, gives the identity:

$$\underline{\underline{A}}\underline{\underline{A}}^{+}-1 = \underline{\underline{A}}^{-1}\underline{\underline{A}} = \underline{\underline{I}} \tag{1.32}$$

As we will see in Chapter 3, this gives rise to the notion of reversing a geometric transformation. Some geometric transformations (and matrices) cannot be inverted. Specifically, a matrix must be square and have a non-zero **determinant** in order to be inverted by conventional means.

1.7.5 Matrix Transposition

Matrix transposition, just like vector transposition, is simply a matter of swapping the rows and columns of a matrix. As such, every matrix has a transpose. The transpose of $\underline{\underline{A}}$ is written $\underline{\underline{A}}^T$:

$$\underline{\underline{A}}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix} \tag{1.33}$$

For some matrices (the **orthonormal** matrices), the transpose actually gives us the inverse of the matrix. We decide if a matrix is orthonormal by inspecting the vectors that make up the matrix's columns, e.g. $[a_{11}, a_{21}]^T$ and $[a_{12}, a_{22}]^T$. These are sometimes called **column vectors** of the matrix. If the magnitudes of all these vectors are one, and if the vectors are orthogonal (perpendicular) to each other, then the matrix is orthonormal. Examples of orthonormal matrices are the identity matrix, and the rotation matrix that we will meet in Chapter 3.

Chapter 2

Image Representation

2.1 Introduction

Computer Graphics is principally concerned with the generation of images, with wide ranging applications from entertainment to scientific visualisation. In this chapter we begin our exploration of Computer Graphics by introducing the fundamental data structures used to represent images on modern computers. We describe the various formats for storing and working with image data, and for representing colour on modern machines.

2.2 The Digital Image

Virtually all computing devices in use today are digital; data is represented in a discrete form using patterns of binary digits (**bits**) that can encode numbers within finite ranges and with limited precision. By contrast, the images we perceive in our environment are analogue. They are formed by complex interactions between light and physical objects, resulting in continuous variations in light wavelength and intensity. Some of this light is reflected in to the retina of the eye, where cells convert light into nerve impulses that we interpret as a visual stimulus.

Suppose we wish to 'capture' an image and represent it on a computer e.g. with a scanner or camera (the machine equivalent of an eye). Since we do not have infinite storage (bits), it follows that we must convert that analogue signal into a more limited digital form. We call this conversion process **sampling**. Sampling theory is an important part of Computer Graphics, underpinning the theory behind both image capture and manipulation — we return to the topic briefly in Chapter 4 (and in detail next semester in CM20220).

For now we simply observe that a digital image can not encode arbitrarily fine levels of detail, nor arbitrarily wide (we say 'dynamic') colour ranges. Rather, we must compromise on accuracy, by choosing an appropriate method to sample and store (i.e. represent) our continuous image in a digital form.

2.2.1 Raster Image Representation

The Computer Graphics solution to the problem of **image representation** is to break the image (picture) up into a regular grid that we call a '**raster**'. Each grid cell is a 'picture cell', a term often contracted to **pixel**. The pixel is the atomic unit of the image; it is coloured uni-

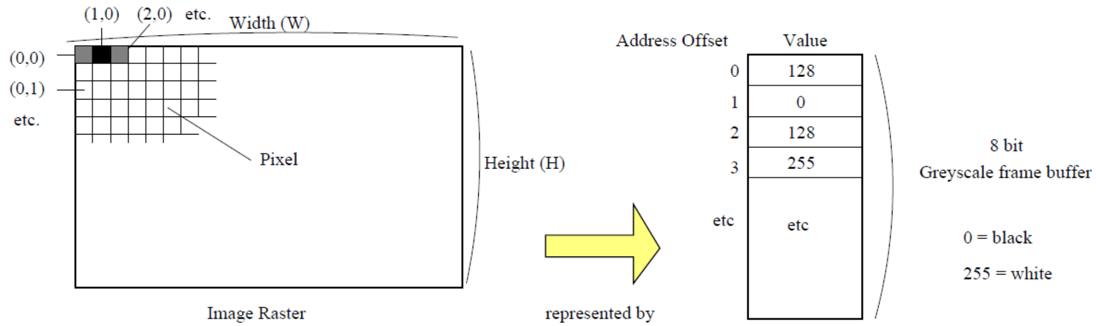


Figure 2.1: Rasters are used to represent digital images. Modern displays use a rectangular raster, comprised of $W \times H$ pixels. The raster illustrated here contains a greyscale image; its contents are represented in memory by a greyscale frame buffer. The values stored in the frame buffer record the intensities of the pixels on a discrete scale (0=black, 255=white).

formly — its single colour representing a discrete sample of light e.g. from a captured image. In most implementations, rasters take the form of a rectilinear grid often containing many thousands of pixels (Figure 2.1). The raster provides an orthogonal two-dimensional basis with which to specify pixel coordinates. By convention, pixels coordinates are zero-indexed and so the origin is located at the **top-left** of the image. Therefore pixel $(W-1, H-1)$ is located at the bottom-right corner of a raster of width W pixels and height H pixels. As a note, some Graphics applications make use of hexagonal pixels instead [1], however we will not consider these on the course.

The number of pixels in an image is referred to as the image's **resolution**. Modern desktop displays are capable of visualising images with resolutions around 1024×768 pixels (i.e. a million pixels or one **mega-pixel**). Even inexpensive modern cameras and scanners are now capable of capturing images at resolutions of several mega-pixels. In general, the greater the resolution, the greater the level of spatial detail an image can represent.

2.2.2 Hardware Frame Buffers

We represent an image by storing values for the colour of each pixel in a structured way. Since the earliest computer Visual Display Units (VDUs) of the 1960s, it has become common practice to reserve a large, **contiguous** block of memory specifically to manipulate the image currently shown on the computer's display. This piece of memory is referred to as a **frame buffer**. By reading or writing to this region of memory, we can read or write the colour values of pixels at particular positions on the display[2].

Note that the term 'frame buffer' as originally defined, strictly refers to the area of memory reserved for direct manipulation of the currently displayed image. In the early days of

[1]Hexagonal displays are interesting because all pixels are equidistant, whereas on a rectilinear raster neighbouring pixels on the diagonal are $\sqrt(2)$ times further apart than neighbours on the horizontal or vertical.

[2]Usually the frame buffer is not located on the same physical chip as the main system memory, but on separate graphics hardware. The buffer 'shadows' (overlaps) a portion of the logical address space of the machine, to enable fast and easy access to the display through the same mechanism that one might access any 'standard' memory location.

Graphics, special hardware was needed to store enough data to represent just that single image. However we may now manipulate hundreds of images in memory simultaneously, and the term 'frame buffer' has fallen into informal use to describe any piece of storage that represents an image.

There are a number of popular formats (i.e. ways of encoding pixels) within a frame buffer. This is partly because each format has its own advantages, and partly for reasons of backward compatibility with older systems (especially on the PC). Often video hardware can be switched between different **video modes**, each of which encodes the frame buffer in a different way. We will describe three common frame buffer formats in the subsequent sections; the greyscale, pseudo-colour, and true-colour formats. If you do Graphics, Vision or mainstream Windows GUI programming then you will likely encounter all three in your work at some stage.

2.2.3 Greyscale Frame Buffer

Arguably the simplest form of frame buffer is the greyscale frame buffer; often mistakenly called 'black and white' or 'monochrome' frame buffers. Greyscale buffers encodes pixels using various shades of grey. In common implementations, pixels are encoded as an unsigned integer using 8 bits (1 byte) and so can represent $2^8 = 256$ different shades of grey. Usually black is represented by value 0, and white by value 255. A mid-intensity grey pixel has value 128. Consequently an image of width W pixels and height H pixels requires $W \times H$ bytes of memory for its frame buffer.

The frame buffer is arranged so that the first byte of memory corresponds to the pixel at coordinates $(0, 0)$. Recall that this is the top-left corner of the image. Addressing then proceeds in a left-right, then top-down manner (see Figure 2.1). So, the value (grey level) of pixel $(1, 0)$ is stored in the second byte of memory, pixel $(0, 1)$ is stored in the $(W + 1)th$ byte, and so on. Pixel (x, y) would be stored at buffer offset A where:

$$A = x + Wy \tag{2.1}$$

i.e. A bytes from the start of the frame buffer. Sometimes we use the term **scan-line** to refer to a full row of pixels. A scan-line is therefore W pixels wide.

Old machines, such as the ZX Spectrum, required more CPU time to iterate through each location in the frame buffer than it took for the video hardware to refresh the screen. In an animation, this would cause undesirable flicker due to partially drawn frames. To compensate, byte range $[0, (W - 1)]$ in the buffer wrote to the first scan-line, as usual. However bytes $[2W, (3W - 1)]$ wrote to a scan-line one third of the way down the display, and $[3W, (4W - 1)]$ to a scan-line two thirds down. This interleaving did complicate Graphics programming, but prevented visual artifacts that would otherwise occur due to slow memory access speeds.

2.2.4 Pseudo-colour Frame Buffer

The **pseudo-colour frame buffer** allows representation of colour images. The storage scheme is identical to the greyscale frame buffer. However the pixel values do not represent shades of grey. Instead each possible value $(0 - 255)$ represents a particular colour; more

specifically, an index into a list of 256 different colours maintained by the video hardware.

The colours themselves are stored in a "**Colour Lookup Table**" (**CLUT**) which is essentially a map $< colourindex, colour >$ i.e. a table indexed with an integer key $(0-255)$ storing a value that represents colour. In alternative terminology the CLUT is sometimes called a **palette**. As we will discuss in greater detail shortly (Section 2.3), many common colours can be produced by adding together (mixing) varying quantities of Red, Green and Blue light. For example, Red and Green light mix to produce Yellow light. Therefore the value stored in the CLUT for each colour is a triple (R, G, B) denoting the quantity (**intensity**) of Red, Green and Blue light in the mix. Each element of the triple is 8 bit i.e. has range $(0-255)$ in common implementations.

The earliest colour displays employed pseudo-colour frame buffers. This is because memory was expensive and colour images could be represented at identical cost to greyscale images (plus a small storage overhead for the CLUT). The obvious disadvantage of a pseudo-colour frame buffer is that only a limited number of colours may be displayed at any one time (i.e. 256 colours). However the colour range (we say **gamut**) of the display is $2^8 \times 2^8 \times 2^8 = 2^24 = 16,777,216$ colours.

Pseudo-colour frame buffers can still be found in many common platforms e.g. both MS and X Windows (for convenience, backward compatibility etc.) and in resource constrained computing domains (e.g. low-spec games consoles, mobiles). Some low-budget (in terms of CPU cycles) animation effects can be produced using pseudo-colour frame buffers. Consider an image filled with an expanses of colour index 1 (we might set CLUT $< 1, Blue >$, to create a blue ocean). We could sprinkle consecutive runs of pixels with index '2,3,4,5' sporadically throughout the image. The CLUT could be set to increasing, lighter shades of Blue at those indices. This might give the appearance of waves. The colour values in the CLUT at indices 2,3,4,5 could be rotated successively, so changing the displayed colours and causing the waves to animate/ripple (but without the CPU overhead of writing to multiple locations in the frame buffer). Effects like this were regularly used in many '80s and early '90s computer games, where computational expense prohibited updating the frame buffer directly for incidental animations.

2.2.5 True-Colour Frame Buffer

The true-colour frame-buffer also represents colour images, but does not use a CLUT. The RGB colour value for each pixel is stored directly within the frame buffer. So, if we use 8 bits to represent each Red, Green and Blue component, we will require 24 bits (3 bytes) of storage per pixel.

As with the other types of frame buffer, pixels are stored in left-right, then top-bottom order. So in our 24 bit colour example, pixel $(0, 0)$ would be stored at buffer locations 0, 1 and 2. Pixel $(1, 0)$ at 3, 4, and 5; and so on. Pixel (x, y) would be stored at offset A where:

$$\begin{aligned} S &= 3W \\ A &= 3x + Sy \end{aligned} \tag{2.2}$$

where S is sometimes referred to as the **stride** of the display.

The advantages of the true-colour buffer complement the disadvantages of the pseudo-colour buffer We can represent all 16 million colours at once in an image (given a large enough image!), but our image takes 3 times as much storage as the pseudo-colour buffer. The image would also take longer to update (3 times as many memory writes) which should be taken under consideration on resource constrained platforms (e.g. if writing a video codec on a mobile phone).

Alternative forms of true-colour buffer

The true colour buffer, as described, uses 24 bits to represent RGB colour. The usual convention is to write the R, G, and B values in order for each pixel. Sometime image formats (e.g. Windows Bitmap) write colours in order B, G, R. This is primarily due to the little-endian hardware architecture of PCs, which run Windows. These formats are sometimes referred to as RGB888 or BGR888 respectively.

2.3 Representation of Colour

Our eyes work by focussing light through an elastic lens, onto a patch at the back of our eye called the **retina**. The retina contains light sensitive **rod** and **cone** cells that are sensitive to light, and send electrical impulses to our brain that we interpret as a visual stimulus (Figure 2.2).

Cone cells are responsible for colour vision. There are three types of cone; each type has evolved to be optimally sensitive to a particular wavelength of light. Visible light has wavelength 700-400nm (red to violet). Figure 2.3 (left) sketches the response of the three cone types to wavelengths in this band. The peaks are located at colours that we have come to call "Red", "Green" and "Blue"[3]. Note that in most cases the response of the cones decays monotonically with distance from the optimal response wavelength. But interestingly, the Red cone violates this observation, having a slightly raised secondary response to the Blue wavelengths.

Given this biological apparatus, we can simulate the presence of many colours by shining Red, Green and Blue light into the human eye with carefully chosen intensities. This is the basis on which all colour display technologies (CRTs, LCDs, TFTs, Plasma, Data projectors) operate. Inside our machine (TV, Computer, Projector) we represent pixel colours using values for Red, Green and Blue (RGB triples) and the video hardware uses these values to generate the appropriate amount of Red, Green and Blue light (subsection 2.2.2).

Red, Green and Blue are called the "**additive primaries**" because we obtain other, non-primary colours by blending (adding) together different quantities of Red, Green and Blue light. We can make the additive **secondary colours** by mixing pairs of primaries: Red and Green make Yellow; Green and Blue make Cyan (light blue); Blue and Red make Magenta (light purple). If we mix all three additive primaries we get **White**. If we don't mix any amount of the additive primaries we generate zero light, and so get **Black** (the absence of colour).

[3]Other animals have cones that peak at different colours; for example bees can see ultra-violet as they have three cones; one of which peaks at ultra-violet. The centres of flowers are often marked with ultra-violet patterns invisible to humans.

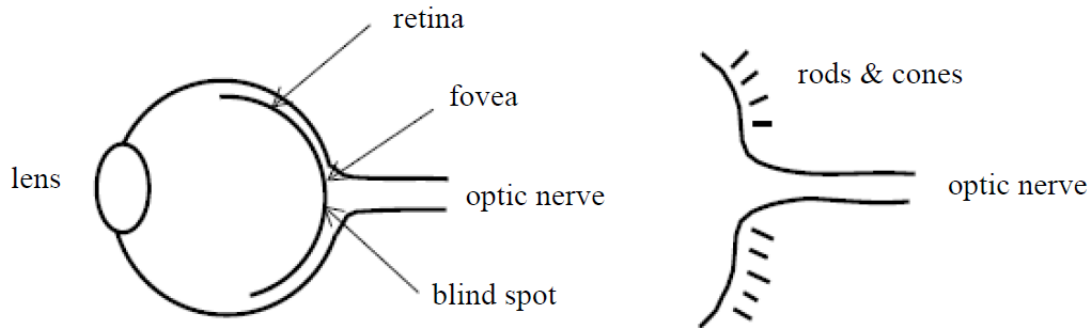


Figure 2.2: In the human eye, light is focused through an elastic lens onto a light sensitive patch (the retine). Special cells (rods and cones) in the retina convert light into electrical impulses that travel to the brain via the optic nerve. The site where the nerve exits the eye contains no such cells, and is termed the "blind spot". The cone cells in particular are responsible for our *colour* vision.

2.3.1 Additive vs. Subtractive Primaries

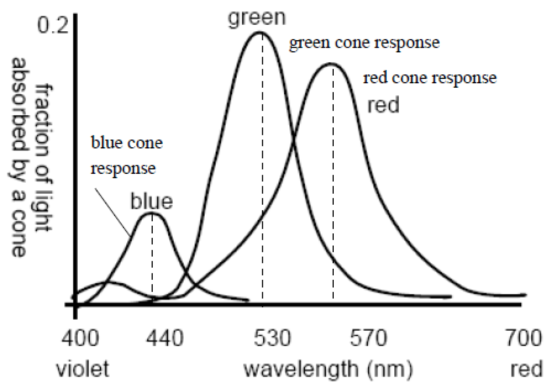
You may recall mixing paints as a child; being taught (and experimentally verifying) that Red, Yellow and Blue were the primary colours and could be mixed to obtain the other colours. So how do we resolve this discrepancy; are the primary colours Red, Green, and Blue; or are they Red, Yellow, and Blue?

When we view a Yellow object, we say that it is Yellow because light of a narrow band of wavelengths we have come to call "Yellow" enters our eye, stimulating the Red and Green cones. More specifically, the Yellow object reflects ambient light of the Yellow wavelength and absorbs all other light wavelengths.

We can think of Yellow paint as reflecting a band wavelengths spanning the Red-Green part of the spectrum, and absorbing everything else. Similarly, Cyan paint reflects the Green-Blue part of the spectrum, and absorbs everything else. Mixing Yellow and Cyan paint causes the paint particles to absorb all but the Green light. So we see that mixing Yellow and Cyan paint gives us Green. This allies with our experience mixing paints as a child; Yellow and Blue make Green. Figure 2.3 (right) illustrates this diagrammatically.

In this case adding a new paint (Cyan) to a Yellow mix, caused our resultant mix to become more restrictive in the wavelengths it reflected. We earlier referred to Cyan, Magenta and Yellow as the additive secondary colours. But these colours are more often called the "**subtractive primaries**". We see that each subtractive primary we contribute in to the paint mix "subtracts" i.e. absorbs a band of wavelengths. Ultimately if we mix all three primaries; Cyan, Yellow and Magenta together we get Black because all visible light is absorbed, and none reflected.

So to recap; RGB are the additive primaries and CMY (Cyan, Magenta, Yellow) the subtractive primaries. They are used respectively to mix light (e.g. on displays), and to mix ink/paint (e.g. when printing). You may be aware that colour printer cartridges are sold



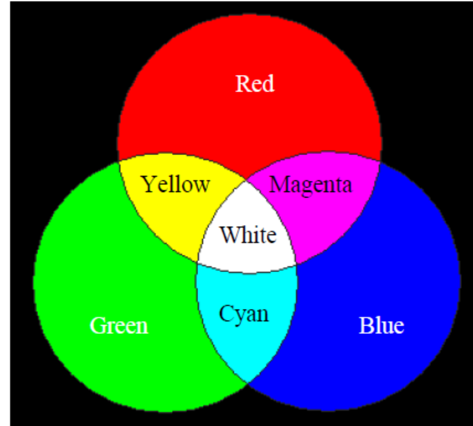


Figure 2.3: Left: Sketching the response of each of the three cones to differing wavelengths of light. Peaks are observed around the colours we refer to as Red, Green and Blue. Observe the secondary response of the Red cone to Blue light. Right: The RGB additive primary colour wheel, showing how light is mixed to produce different colours. All 3 primaries (Red, Green, Blue) combine to make white.

containing Cyan, Magenta and Yellow (CMY) ink; the subtractive primaries. Returning to our original observation, CMY are the colours approximated by the child when mixing Red, Yellow and Blue paint (blue is easier to teach than 'cyan'; similarly magenta).

2.3.2 RGB and CMYK colour spaces

We have seen that displays represent colour using a triple (R,G,B). We can interpret each colour as a point in a three dimensional space (with axes Red, Green, Blue). This is one example of a **colour space** — the RGB colour space. The RGB colour space is cube-shaped and is sometimes called the **RGB colour cube**. We can think of picking colours as picking points in the cube (Figure 2.4, left). Black is located at $(0, 0, 0)$ and White is located at $(255, 255, 255)$. Shades of grey are located at (n, n, n) i.e. on the diagonal between Black and White.

We have also seen that painting processes that deposit pigment (i.e. printing) are more appropriately described using colours in CMY space. This space is also cube shaped.

We observed earlier that to print Black requires mixing of all three subtractive primaries (CMY). Printer ink can be expensive, and Black is a common colour in printed documents. It is therefore inefficient to deposit three quantities of ink onto a page each time we need to print something in Black. Therefore printer cartridges often contain four colours: Cyan, Magenta, Yellow, and a pre-mixed Black. This is written CMYK, and is a modified form of the CMY colour space.

CMYK can help us print non-black colours more efficiently too. If we wish to print a colour (c, y, m) in CMY space, we can find the amount of Black in that colour (written k for 'key')

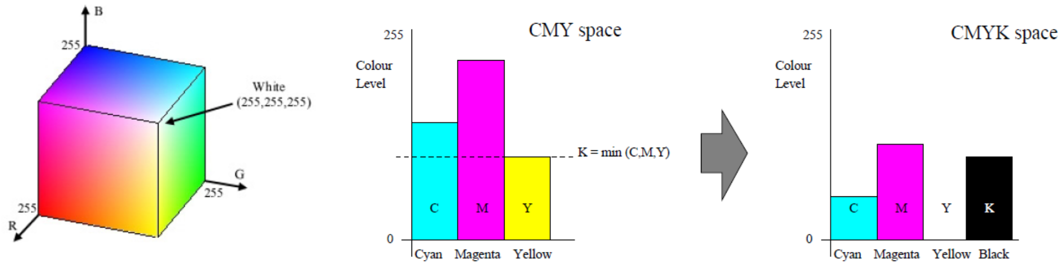


Figure 2.4: Left: The RGB colour cube; colours are represented as points in a 3D space each axis of which is an additive primary. Right: CMYK; the quantity of black (C, M, and Y ink mix) contained within a colour is substituted with pure black (K) ink. This saves ink and improves colour tone, since a perfectly dark black is difficult to mix using C, M, and Y inks.

by computing:

$$k = \min(c, y, m) \tag{2.3}$$

We then compute $c = c - k$, $y = y - k$, $m = m - k$ (one or more of which will be zero) and so represent our original CMY colour (c, y, m) as a CMYK colour (c, m, y, k). Figure 2.4 (right) illustrates this process. It is clear that a lower volume of ink will be required due to economies made in the Black portion of the colour, by substituting CMY ink for K ink.

You may be wondering if an "RGB plus White" space might exist for the additive primaries. Indeed some devices such as data projectors do feature a fourth colour channel for White (in addition to RGB) that works in a manner analogous to Black in CMYK space. The amount of white in a colour is linked to the definition of **colour saturation**, which we will return to in subsection 2.3.6.

2.3.3 Greyscale Conversion

Sometimes we wish to convert a colour image into a greyscale image; i.e. an image where colours are represented using shades of grey rather than different hues. It is often desirable to do this in Computer Vision applications, because many common operations (edge detection, etc) require a scalar value for each pixel, rather than a vector quantity such as an RGB triple. Sometimes there are aesthetic motivations behind such transformations also.

Referring back to Figure 2.3 we see that each type of cones responds with varying strength to each wavelength of light. If we combine (equation 2.4) the overall response of the three cones for each wavelength we would obtain a curve such as Figure 2.5 (left). The curve indicates the *perceived* **brightness** (the Graphics term is **luminosity**) of each light wavelength (colour), given a light source of *constant brightness* output.

By experimenting with the human visual system, researchers have derived the following equation to model this response, and so obtain the luminosity (l) for a given RGB colour (r, g, b) as:

$$l(r, g, b) = 0.30r + 0.59g + 0.11b \tag{2.4}$$

Figure 2.5: Greyscale conversion. Left: The human eye's combined response (weighted sum of cone responses) to light of increasing wavelength but constant intensity. The perceived luminosity of the light changes as a function of wavelength. Right: A painting (Impressionist Sunrise by Monet) in colour and greyscale; note the isoluminant colours used to paint the sun against the sky.

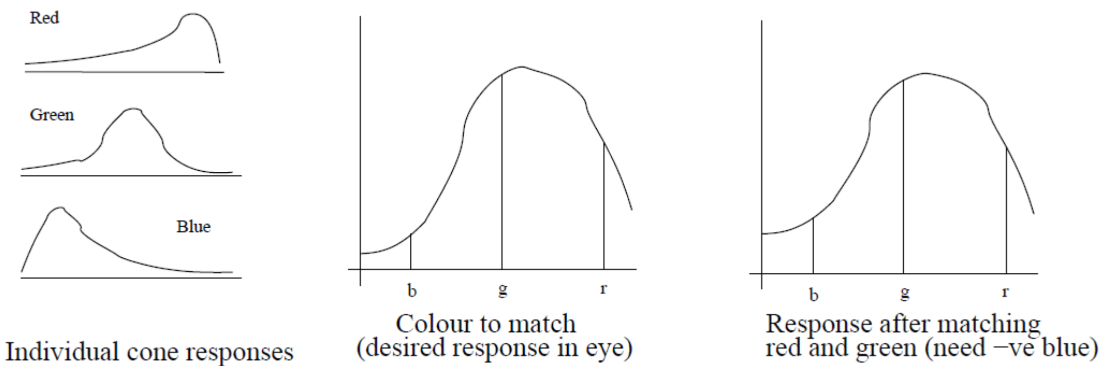


Figure 2.6: Tri-stimulus experiments attempt to match real-world colours with a mix of red, green and blue (narrow bandwidth) primary light sources. Certain colours cannot be matched because cones have a wide bandwidth response e.g. the blue cone responds a little to red and green light (see subsection 2.3.4).

As a note, usually we denote luminosity using Y — do not confuse this with the Y in CMYK.

You can write Matlab code to convert a colour JPEG image into a greyscale image as follows:

```
img=double(imread('image.jpg'))./255;
r=img(:,:,1);
g=img(:,:,2);
b=img(:,:,3);
y=0.3.*r + 0.69.*g + 0.11.*b;
imshow(y);
```

This code was applied to the source image in Figure 2.5. This Figure gives clear illustration that visually distinct colours (e.g. of the sun and sky in the image) can map to similar greyscale values. Such colours are called "isoluminant" (i.e. of same luminance).

2.3.4 Can any colour be represented in RGB space?

No. This can be shown using the **tri-stimulus experiment**; a manual colour matching exercise performed under controlled conditions as follows:

A human participant is seated in a darkened room, with two pieces of card before him/her. One piece of card is coloured uniformly with a target colour, and illuminated with white light. The other piece of card is white, and illuminated simultaneously by a Red, Green and Blue light source. The human is given control over the intensities of the Red, Green and Blue primary sources and asked to generate a colour on the white card that matches the coloured card.

To illustrate why we cannot match some colours (and thus represent those colours in RGB space), consider the following. The participant attempts to match a colour by starting with all three primary sources at zero (off), and increasing the red and green primaries to match the wavelength distribution of the original colour (see Figure 2.6). Although the red end of the spectrum is well matched, the tail ends of the red and green cone responses cause us to perceive a higher blue component in the matched colour than there actually is in the target colour. We would like to "take some blue out" of the matched colour, but the blue light is already completely off.

Thus we could match any colour if we allowed negative values of the RGB primaries, but this is not physically possible as our primaries are additive by definition. The best we could do to create a match is cheat by shining a blue light on the target card, which is equivalent mathematically to a negative primary but of course is not a useful solution to representing the target (white illuminated) card in RGB space.

2.3.5 CIE colour space

We can represent all physically realisable colours in an alternative colour space named CIE after Commission Internationale de L'Eclairage who developed the space in 1976 in an attempt to "standardise" the representation of colour.

The CIE colour space is sometimes known as the CIEXYZ model, because it represents colours as a weighted sum of three **ideal primaries** that are written **X**, **Y** and **Z**. These primaries are offset from Red, Green and Blue in such a way that we need only positive contributions from those primaries to produce all visible colours. For the reasons outlined in subsection 2.3.4 the ideal primaries are not physically realisable light sources. Rather they are mathematical constants in the CIEXYZ model.

Thus a colour C is written:

$$C = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z} \tag{2.5}$$

The triple (X, Y, Z) is known as the **chromaticity coordinates** of the colour C. We more commonly work with **normalised chromaticity coordinates** (x, y, z) as follows:

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \tag{2.6}$$

Since $x + y + z = 1$, we need only note down two of the three normalised coordinates to represent chromaticity. The loss of a dimension is caused by the normalisation, which has divided out intensity (brightness) information. To fully represent a colour, we therefore need one of the non-normalised coordinates and two normalised coordinates. Say, for example we have Y then we recover (X, Y, Z) as:

$$\begin{aligned} X &= \frac{x}{y} Y \\ Y &= Y \\ Z &= \frac{1 - x - y}{y} Y \end{aligned} \tag{2.7}$$

To recap; we need just two normalised coordinates e.g. (x, y) to represent the colour's chromaticity (e.g. whether it is red, yellow, blue, etc.) and an extra non-normalised coordinate e.g. Y to recover the colour's luminance (its brightness).

Figure 2.7 (left) shows how colours map on to a plot of (x, y) i.e. the chromaticity space of CIEXYZ. White is located somewhere close to $x = y = \frac{1}{3}$; as points (colours) in the space approach this "white point" they become de-saturated (washed out) producing pastel shades. Colours of the spectrum fall on the curved part of the boundary. All colours in the chromaticity space are of unit intensity due to the normalisation process, thus there are no intensity related colours e.g. browns.

We do not need to explore the CIEXYZ model in greater depth for the purposes of this course. There are some small modifications that can be made to the CIEXYZ model to transform the space into one where Euclidean distance between points in the space corresponds to perceptual distance between colours. This can be useful for colour classification (Chapter 5). For further details look up the CIELAB colour space in Foley *et al.*

However there is one final, important observation arising from the CIE model to discuss. We can see that Red, Green and Blue are points in the space. Any linear combination of those **additive** primaries must therefore fall within a triangle on the CIE (x, y) space. The colours in that triangle are those that can be realised using three physical primaries (such as phosphors in a VDU). We can also see that Cyan, Magenta and Yellow are points in the space forming a second triangle. Similarly, the printable colours (from these **subtractive** primaries) are all those within this second triangle. Some VDUs and Printers have the CIEXYZ coordinates of their primaries stamped on their machine casings.

It is very likely that the triangle created by your printer's primaries does not exactly overlap the triangle created by your VDU's primaries (and vice versa) (Figure 2.7). Thus it is difficult to ensure consistency of colour between screen and print media. Your computer's printer driver software manages the transformation of colours from one region (triangle) in CIEXYZ space to another. With the driver, various (often proprietary) algorithms exist to transform between the two triangles in CIEXYZ space without introducing perceptually unexpected artifacts.

2.3.6 Hue, Saturation, Value (HSV) colour space

The RGB, CMYK and CIE models are not very intuitive for non-expert users. People tend to think of colour in terms of hue, rather than primaries. The **Hue, Saturation, Value**

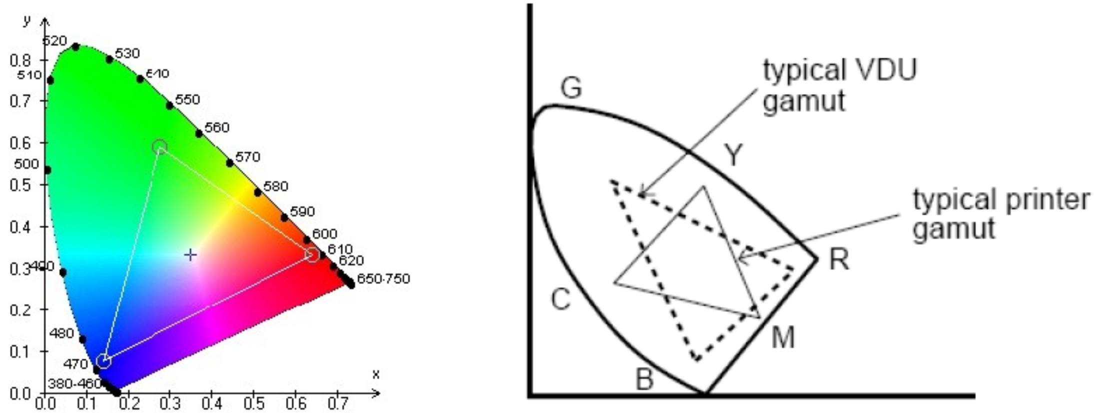


Figure 2.7: Left: The (x, y) normalised chromaticity space of the CIEXYZ model. Spectral colours appear around the curved edge of the space (wavelengths noted here in nanometres). The white point is roughly in the centre of the space. Right: Colours obtainable via the RGB (additive) primaries and CMY (subtractive) primaries create separate triangular regions in the normalised chromaticity space. The area of overlap shows which VDU colours can be faithfully reproduced on a printer, and vice versa.

(HSV) model tries to model this. It is used extensively in GUIs for colour picker interfaces.

Here, we define HSV by describing conversion of a colour from RGB. Throughout this section, we will assume colours in RGB space have been normalised to range $(0-1)$ instead of the more common $(0-255)$.

As discussed in subsection 2.3.1, we can compute the amount of White (W) in an RGB colour just as we can compute the amount of Black in a CMY colour — by taking the minimum of the three colour components.

$$W = \min(R, G, B) \tag{2.8}$$

The **saturation** (S) of a colour is defined as $S = 1 - W$. Thus a de-saturated colour has a lot of white in it; such colours resemble pastel shades. A highly saturated colour would be a rich red, yellow, green etc. as might be seen on an advertisement/poster. The saturation S forms one of the three values (H, S and V) of the HSV colour.

Equally simple to compute is the 'V' component or **value**. In HSV space this is computed as the maximum of the R, G and B components.

$$V = \max(R, G, B) \tag{2.9}$$

This is a quick (to compute) approximation to luminance. In the related **Hue, Saturation, Luminance (HSL)** colour model, equation 2.4 is used instead of equation 2.9. The two models are functionally near-identical.

Finally, hue is computed by subtracting the white component W from each of the colour components R, G and B. For example, RGB colour (0.3, 0.6, 0.2) would yield triple $(0.1, 0.4, 0.0)$

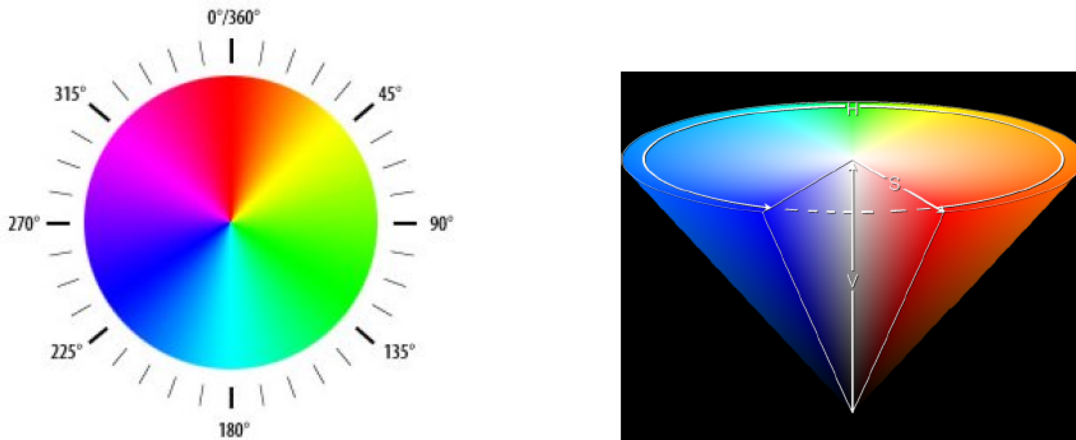


Figure 2.8: Left: The Hue colour wheel; hue is represented as degrees around a circle. Right: The HSV colour space can be visualised as a cone; the apex (Black) having any Hue or Saturation, but Value=0. Similarly, colours at the centre of the cone are white (completely desaturated). The above images are reproduced under Creative Commons/Wikimedia licence.

after subtracting the white component ($W = 0.2$) from the colour. Because one of the values in the white-subtracted triple is always zero (just as in CMYK — subsection 2.3.1) we only have two numbers to represent. The convention is to represent hue as degrees around a circle (see Figure 2.8).

Continuing our example, the two non-zero components of the white-subtracted colour are used to compute the hue as follows. We observe that the colour $0.4/(0.4+0.1)$ i.e. $4/5$ of the way along the arc between Red and Green. Expressing hue as an angle, this is $4/5 \times 120 = 96$ around the hue circle.

Hue is computed similarly for colours with 0.0 in the Red, or Green positions within the white-subtracted RGB triple. However a constant (120° or 240°) is added to shift the hue value into the appropriate range. To illustrate, consider an example where RGB=$(0.4, 0.3, 0.6)$. We compute $V = 0.6$ and $S = 0.7$. The white-subtracted RGB triple is $(0.1, 0.0, 0.3)$. The hue is therefore $0.3/(0.1 + 0.3) = 3/4$ of the way along the arc between Blue and Red, i.e. 90°. The total angle around the circle (our final value for Hue) is $240° + 90° = 330°$.

2.3.7 Choosing an appropriate colour space

We have discussed four colour spaces (RGB, CMYK, CIELAB and HSV), and summarised the advantages of each. Although RGB is by far the most commonly used space, the space you should choose for your work is a function of your requirements. For example, HSV is often more intuitive for non-expert users to understand and so is commonly used for GUI colour pickers. However the fact that many points in HSV space e.g. $(h, s, 0)$ map to Black can be problematic if using HSV for colour classification in Computer Vision; perhaps RGB would be more suitable as each point in the RGB colour cube identifies a unique colour.

Another consideration in your choice should be interpolation. Suppose we have two colours,

Bright Red and Bright Green that we wish to mix in equal parts e.g. in a photo editing package. In RGB space these colours are $(255, 0, 0)$ and $(0, 255, 0)$ respectively. In HSV space these colours are $(0^\circ, 1, 1)$ and $(120^\circ, 1, 1)$ respectively. Using linear interpolation, a colour halfway between Red and Green is therefore $(128, 128, 0)$ and $(60\circ, 1, 1)$ in RGB and HSV space respectively (i.e. the mean average of Red and Green).

However $(128, 128, 0)$ in RGB space is a dull yellow, whereas $(60, 1, 1)$ in HSV space is a bright yellow. Clearly the choice of colour space affects the results of blending (i.e. interpolating) colours.

Chapter 3

Geometric Transformation

3.1 Introduction

In this chapter we will describe how to manipulate models of objects and display them on the screen.

In Computer Graphics we most commonly model objects using points, i.e. locations in 2D or 3D space. For example, we can model a 2D shape as a polygon whose vertices are points. By manipulating the points, we can define the shape of an object, or move it around in space. In 3D too, we can model a shape using points. Points might define the locations (perhaps the corners) of surfaces in space.

Later in Chapter 6 we will consider various object modelling techniques in 2D and 3D. For now, we need concern ourselves only with points and the ways in which we may manipulate their locations in space.

3.2 2D Rigid Body Transformations

Consider a shape, such as a square, modelled as a polygon in 2D space. We define this shape as a collection of points $\underline{\underline{p}} = [\underline{p_1}\ \underline{p_2}\ \underline{p_3}\ \underline{p_4}]$, where $\underline{p_i}$ are the corners of the square. Commonly useful operations might be to enlarge or shrink the square, perhaps rotate it, or move it around in space. All such operations can be achieved using a matrix transformation of the form:

$$\underline{p'} = \underline{\underline{M}}\underline{p} \tag{3.1}$$

where $\underline{p}$ are the original locations of the points, and $\underline{p'}$ are the new locations of the points. $\underline{\underline{M}}$ is the transformation matrix, a 2×2 matrix that we fill appropriate values to achieve particular transformations.

These "matrix transformations" are sometimes called **rigid body transformations** because all points $\underline{p}$ under-go the same transformation. Although many useful operations can be performed with rigid body transformations, not all can be. For example we couldn't squash the sides of a square inward to produce a pin-cushion distortion as in Figure 3.1 using a matrix transformation; pin-cushion distortion is an example of a more general geometric

Figure 3.1: Various geometric transformations applied to an image. The results were achieved using two rigid body transformations (reflection, and a combined rotation and scaling) and a non-rigid transformation (pin-cushion distortion). Note that only rigid body transformations can be realised using matrix multiplication.

transformation. Rigid body transformations do however form a useful subset of geometric transformations, and we will explore some of these now.

3.2.1 Scaling

We can scale (enlarge or shrink) a 2D polygon using the following **scaling matrix**:

$$\underline{\underline{M}} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \tag{3.2}$$

where S_x is the **scale factor** by which we wish to enlarge the object in the direction of the x-axis similarly S_y for the y-axis. So a point $(x, y)^T$ will be transformed to location $(S_x x, S_y y)^T$.

For example, setting $S_x = S_y = 2$ will double the size of the object, and $S_x = S_y = \frac{1}{2}$ will halve the size of the object. The centre of projection for the scaling is the origin, so the centre of the shape will become farther or nearer to the origin respectively as a result of the transformation. Figure 3.2 (red) shows an example of scaling where $S_x = 2, S_y = 1$, resulting in square $\underline{p}$ being stretched in th direction of the x-axis to become a rectangle.

You may note that setting the scale factor to 1 resulting in the identity matrix, i.e. with no effect.

A negative scale factor will reflect points. For example, setting $S_x = -1 and S_y = 1$ will reflect points in the y-axis.

3.2.2 Shearing (Skewing)

We can shear (skew) a 2D polygon using the following **shearing matrix**:

$$\underline{\underline{M}} = \begin{bmatrix} 1 & q \\ 0 & 1 \end{bmatrix} \tag{3.3}$$

So a point $(x, y)^T$ will be transformed to location $(x + qy, y)^T$, so shifting the x component of the point by an amount proportional to its y component. Parameter q is the constant of proportionality and results, for example, in the vertical lines of the polygon “tipping” into diagonal lines with slope $1/q$. Figure 3.2 (green) illustrates this.

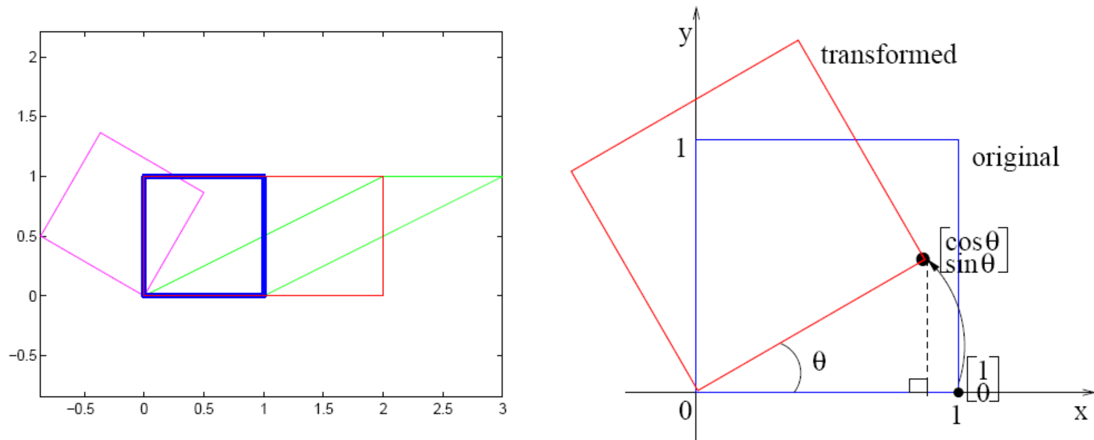


Figure 3.2: Illustrating various linear transformations. Left: a blue unit square, with one corner at the origin. A scaled square in red ($S_x = 2, S_y = 1$), a sheared square in green ($q = 2$), a rotated square in magenta ($\theta = \pi/3$ radians). Right: A diagram illustrating rotation θ degrees anti-clockwise about the origin. A corner at point $(1, 0)^T$ moves to points $(\cos\theta, \sin\theta)^T$ under the rotation.

3.2.3 Rotation

We can rotate a 2D polygon θ degrees anti-clockwise about the origin using the following rotation matrix:

$$\underline{\underline{M}} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.4}$$

Figure 3.2 (magenta) demonstrates operation of this matrix upon a square. Consider the corner $(1, 0)^T$ transformed by the above matrix. The resulting point has location $(\cos\theta, \sin\theta)^T$. We can see from the construction in Figure 3.2 that the point is indeed rotated θ degrees anti-clockwise about the origin. Now consider a rotation of the point $(\cos\theta, \sin\theta)^T$ a further β degrees about the origin; we would expect the new location of the point to be $(cos(\theta + \beta), sin(\theta + \beta))^T$. As an exercise, perform this matrix multiplication and show that the resulting point has location $(\cos\theta\cos\beta - \sin\theta\sin\beta, \sin\theta\cos\beta + \cos\theta sin\beta)^T$. You may recognise these as the **double-angle formulae** you learnt in trigonometry:

$$\begin{aligned} \cos(\theta + \beta) &= \cos\theta\cos\beta - \sin\theta\sin\beta \\ \sin(\theta + \beta) &= \sin\theta\cos\beta + \cos\theta\sin\beta \end{aligned} \tag{3.5}$$

The rotation matrix is orthonormal, so the inverse of the matrix can obtained simply by transposing it (exchanging the signs on the $\sin\theta$ elements). **In general, when we multiply by the inverse of a transformation matrix, it has the opposite effect**. So, the inverse of the rotation matrix defined in eq.(3.4) will rotate in a clockwise rather than anti-clockwise direction (i.e. by $-\theta$ rather than θ). The inverse of a scaling matrix of factor two (i.e. making objects 2 times larger), results in an scaling matrix of factor 0.5 (i.e. making objects 2 times smaller).

3.2.4 Active vs. Passive Interpretation

We can interpret the actions of matrix transformations in two complementary ways (these are different ways of thinking, and do not alter the mathematics). So far we have thought of matrices as acting upon points to move them to new locations. This is called the **active interpretation** of the transformation.

We could instead think of the point coordinates remaining the same, but the space in which the points exist being warped by the matrix transformation. That is, the reference frame of the space changes but the points do not. This is the **passive interpretation** of the transformation. We illustrate this concept with two examples, a scaling and a rotation. You may wish to refer to the discussion of basis sets and reference frames in the "Revision of Mathematical Background" section of Chapter 1.

Example 1: Scaling Transformation

Recall eq.(3.2) which describes a scaling transformation. Setting $S_x = S_y = 2$ will scale points up by a factor of 2. Consider the transformation acting on point $\underline{p} = (1, 1)^T$, i.e.:

$$\underline{\underline{M}}\underline{p} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{3.6}$$

The **active interpretation** of the transformation is that the x and y coordinates of the point will be modified to twice their original value, moving the point away from the origin to location $(2, 2)^T$. This all occurs within root reference frame in which $\underline{p}$ is defined, i.e. where $\hat{\underline{i}} = [1\ 0]^T$ and $\hat{\underline{j}} = [0\ 1]^T$.

In the **passive interpretation** of the transformation, we imagine $\underline{p}$ to stay "fixed", whilst the space it is defined in "slips beneath" $\underline{p}$, contracting to half its size. The tick marks on the axes of the space shrink towards the origin. $\underline{p}$ ends up aligned with coordinates $(2, 2)^T$ with respect to the shrunken space, rather than $(1, 1)^T$ as was originally the case. Consequently the space in which the point is defined as $(1, 1)^T$ is twice as large as the space in which the point is defined as $(2, 2)^T$; that is, one step in the former space will take us twice as far as one step in the latter space.

This explanation is made clearer with some mathematics. We interpret the transformation matrix $\underline{\underline{M}}$ as defining a new basis set with basis vectors $(\underline{i_M}, \underline{j_M})$ defined by the columns of $\underline{\underline{M}}$, i.e. $\underline{i_M} = [2\ 0]^T$ and $\underline{j_M} = [0\ 2]^T$. In the passive interpretation, we interpret point $(1, 1)^T$ as existing within this new basis set $(\underline{i_M}, \underline{j_M})$ rather than the root reference frame. However, we wish to convert from this reference frame to discover the coordinates of the point within the root reference frame in which we usually work, i.e. where $\hat{\underline{i}} = [1\ 0]^T$ and $\hat{\underline{j}} = [0\ 1]^T$.

The x coordinate of the point within the $[\underline{i_M}, \underline{j_M}]$ reference frame is 1, and the y coordinate is also 1. So these coordinates contribute $(1 \times 2)\hat{\underline{i}} + (1 \times 0)\hat{\underline{j}} = 2\hat{\underline{i}}$ to the point's x coordinate in the root frame (the 2 and the 0 come from $\underline{i_M} = [2\ 0]^T$). The y coordinate in the root frame is contributed to by $(1 \times 0)\hat{\underline{i}} + (1 \times 2)\hat{\underline{j}} = 2\hat{\underline{j}}$. So the point's coordinates in the root reference frame is $(2, 2)^T$.

Thus to think in the **passive interpretation** is to affect our transformation by thinking of the original coordinates $(1,1)^T$ as existing within an arbitrary reference frame, as defined by $\underline{\underline{M}}$. Multiplying the point's coordinates in that frame by $\underline{\underline{M}}$ takes the point out of the arbitrary reference frame and into the root reference frame.

Example 2: Rotation Transformation

Recall eq.(3.4) which describes a rotation operation. Setting $\theta = 45°$ (or $\pi/2$ radians) will rotate the points anti-clockwise about the origin by that amount. E.g. a point $(1,0)^T$ would under-go the following to end up at $(\cos\theta, \sin\theta)^T$:

$$\underline{\underline{M}}p = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} \quad (3.7)$$

The **active interpretation** of this transformation is simply that the point has moved (rotated) about the origin 45 degrees *anti-clockwise*, within the root reference frame i.e. where $\hat{\underline{i}} = [1\ 0]^T$ and $\hat{\underline{j}} = [0\ 1]^T$. So the point moves from $(1,0)^T$ to $(0.707, 0.707)^T$. Figure 3.2 (right) illustrated this.

As with the previous example, the passive interpretation is that the point remains "fixed" but the space warps via the inverse transformation. In this case the space "slips beneath" the point, rotating θ degrees in a *clockwise* direction. Thus the point becomes aligned with coordinates $(\cos\theta, \sin\theta)$, in the slipped space, rather than $(1,0)^T$ as it was in the space before the transformation.

Again it may be clearer to look at a mathematical explanation. We view the coordinates $(1,0)^T$ as being defined in a new reference frame with basis vectors taken from the columns of $\underline{\underline{M}}$, i.e. $\underline{i_M} = [\cos\theta\ \ \sin\theta]^T$ and $\underline{j_M} = [-\sin\theta\ \ \cos\theta]^T$. When we multiply by $\underline{\underline{M}}$ we are computing the coordinates of that point within our root reference frame, i.e. where $\hat{\underline{i}} = [1\ 0]^T$ and $\hat{\underline{j}} = [0\ 1]^T$. The x coordinate of the point within the $[\underline{i_M}, \underline{j_M}]$ reference frame is 1, and the y coordinate within that frame is 0. So these coordinates contribute $(1 \times \cos\theta)\hat{\underline{i}} + (0 \times \sin\theta)\hat{\underline{j}}$ to the point's x coordinate in the root frame (the $\cos\theta$ and the $\sin\theta$ come from $\underline{i_M} = [\cos\theta\ \ \sin\theta]$). Similarly the point's y coordinate within the root frame is contributed to by $(1 \times -\sin\theta)\hat{\underline{i}} + (0 \times \cos\theta)\hat{\underline{j}}$. So the point's coordinates in the root reference frame are $(\cos\theta, \sin\theta) = (0.707, 0.707)^T$.

3.2.5 Transforming between basis sets

As we saw in both Chapter 1 and the previous subsection, we can talk about a single point as having many different sets of coordinates each defined their own reference frame. In this subsection we introduce a subscript notation $\underline{p_F}$ to denote coordinates of a point $\underline{p}$ defined in a reference frame $\underline{\underline{F}}$ with basis vectors $\underline{i_F}$ and $\underline{j_F}$.

We have seen that a point with coordinates $\underline{p_D}$, defined in an arbitrary reference frame $(\underline{i_D}, \underline{j_D})$ can be represented by coordinates $\underline{p_R}$ in the root reference frame by:

$$\begin{aligned} \underline{p_R} &= \underline{\underline{D}}\underline{p_D} \\ \underline{p_R} &= \left[\underline{i_D}\ \underline{j_D}\right] \underline{p_D} \end{aligned} \quad (3.8)$$

Conversely , if we have a point $\underline{p_R}$ defined in the root reference frame, we can convert those coordinates into those any other reference frame e.g. $\underline{\underline{D}}$ by multiplying by the inverse of of $\underline{\underline{D}}$:

$$\begin{aligned} \underline{p_D} &= \underline{\underline{D}}^{-1}\underline{p_R} \\ \underline{p_D} &= \left[\underline{i_D}\ \underline{j_D}\right]^{-1}\underline{p_R} \end{aligned} \tag{3.9}$$

Suppose we had $\underline{p_R} = (1,1)^T$ and a reference frame defined by basis $\underline{i_D} = [3\ 0]^T$ and $\underline{j_D} = [0\ 3]^T$. The coordinates $\underline{p_D}$ following eq.(3.9) would be $(0.33, 0.33)^T$ — which is as we might expect because $\underline{\underline{D}}$ is 'three times as big' as the root reference frame (we need to move three steps in the root reference frame for every one we taken in frame $\underline{\underline{D}}$).

So we have seen how to transform coordinates to and from an arbitrary reference frame, but how can we convert directly between two arbitrary frames without having to convert to the root reference frame?

Transforming directly between arbitrary frames

Given a point $\underline{p_D}$ in frame $\underline{\underline{D}}$, we can obtain that point's coordinates in frame $\underline{\underline{E}}$ i.e. $\underline{p_E}$ as follows.

First observe that points $\underline{p_D}$ and $\underline{p_E}$ can be expressed as coordinates in the root frame ($\underline{p_R}$) by simply multiplying by the matrices defining the frames $\underline{\underline{D}}$ and $\underline{\underline{E}}$:

$$\begin{aligned} \underline{p_R} &= \underline{\underline{D}}\underline{p_D} \\ \underline{p_R} &= \underline{\underline{E}}\underline{p_E} \end{aligned} \tag{3.10}$$

By a simple rearrangement then we obtain a direct transformation from frame $\underline{\underline{D}}$ to frame $\underline{\underline{E}}$:

$$\underline{\underline{E}}\underline{p_E} = \underline{\underline{D}}\underline{p_D}$$

$$\underline{p_E} = \underline{\underline{E}}^{-1}\underline{\underline{D}}\underline{p_D} \tag{3.11}$$

$$\underline{p_D} = \underline{\underline{D}}^{-1}\underline{\underline{E}}\underline{p_E} \tag{3.12}$$

3.2.6 Translation and Homogeneous Coordinates

So far we have seen that many useful transformations (scaling, shearing, rotation) can be achieved by multiplying 2D points by a 2×2 matrix. Mathematicians refer to these as **linear transformations**, because each output coordinate is a summation over every input coordinate weighted by a particular factor. If you do not see why this is so, refer to Chapter 1 and the section on Matrix Multiplication.

There are a number of other useful **rigid body transformations** that cannot be achieved using linear transformations. One example is **translation** or "shifting" of points – say we have a set of points describing a shape, and we wish to move that shape 2 units in the positive direction of the x-axis. We can achieve this by simply adding 2 to each point's x-coordinate. However it is impossible to directly add a constant to a particular coordinate within the

framework of **linear transformations**, i.e. in general we resort to the messy form of a 'multiplication and an addition' for translations:

$$\underline{p'} = \underline{\underline{M}}\underline{p} + \underline{t} \tag{3.13}$$

Fortunately there is a solution; we can write translations as a matrix multiplications using **homogeneous coordinates**.

When we work with homogeneous coordinates, we represent an n-dimensional point in a $(n+1)$-dimensional space i.e.

$$\begin{bmatrix} x \\ y \end{bmatrix} \sim \begin{bmatrix} \alpha x \\ \alpha y \\ \alpha \end{bmatrix} \tag{3.14}$$

By convention we usually set $\alpha = 1$ so a point $(2,3)^T$ is written $(2,3,1)^T$ in homogeneous coordinates. We can manipulate such 2D points using 3×3 matrices instead of the 2×2 framework we have used so far. For example translation can be written as follows — where T_x and T_y are the amount to shift (translate) the point in the x and y directions respectively:

$$\underline{p'} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \underline{p} \tag{3.15}$$

We can accommodate all of our previously discussed linear transformations in this 3×3 framework, by setting the matrix to the identity and overwriting the top-left 2×2 section of the matrix by the original 2×2 linear transformation matrix. For example, rotation degrees anticlockwise about the origin would be:

$$\underline{p'} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \underline{p} \tag{3.16}$$

Following the transformation we end up with a point in the form $\underline{p} = [x, y, \alpha]^T$. We divide by α (the 'homogeneous coordinate') to obtain the true locations of the resultant point's x and y coordinates in the 2D space.

Classes of transformation enabled by homogeneous coordinates

We have seen that homogeneous coordinates allow us to effect linear transformations with matrix multiplications. Not only this, but they enable translation too. Translation is one example from a super-set of transformations called the **affine transformations** that incorporate all the linear transformations, and more besides (see next section for additional examples).

Whereas the top-left 2×2 elements in the matrix are manipulated to perform linear transformations, the top 2×3 elements of the matrix are manipulated to perform affine transformations. In both cases the bottom row of the matrix remains [0 0 1] and so the homogeneous coordinate (the α) of the resulting point is always 1. However there exists an even broader class of transformations (the **projective transformations**) that are available by manipulating the bottom row of the matrix. These may change the homogeneous coordinate to a

value other than 1. It is therefore important that we form the habit of **dividing by the homogeneous coordinate, and not simply discarding it**, following the result of a matrix transformation.

In summary, homogeneous coordinates are a way of performing *up to* projective transformations using linear operators in a higher dimensional space. This is facilitated by mapping points in regular (e.g. 2D) space to lines in the homogeneous space (e.g. 3D), and then performing linear transforms on the line in that higher dimensional (homogeneous) space.

Advantages of homogeneous coordinates

An important consequence of homogeneous coordinates is that we can represent several important classes of transformation (linear, affine, projective) in a single framework - multiplication by a 3×3 matrix. But why is this useful?

First, for software engineering reasons it is useful to have a common data-type by which we can represent general transformations e.g. for passing as arguments between programmed functions. Second, and more significantly, we are able to multiply 3×3 matrices together to form more sophisticated **compound matrix transformations** that are representable using a single 3×3 matrix. As we will show in Section 3.2.7, we can, for example, multiply rotation and translation matrices (eqs. 3.15–3.16) to derive a 3×3 matrix for rotation about an arbitrary point — we are not restricted to rotation about the origin as before. Representing these more complex operations as a single matrix multiplication (that can be pre-computed prior to, say, running an animation), rather than applying several matrix multiplications inside an animation loop, can yield substantial improvements in efficiency.

3.2.7 Compound Matrix Transformations

We can combine the basic building blocks of translation, rotation about the origin, scaling about the origin, etc. to create a wider range of transformations. This is achieved by multiplying together the 3×3 matrices of these basic building blocks, to create a single ”compound” 3 3 matrix.
Suppose we have a matrix $\underline{\underline{S}}$ that scales points up by a factor of 2, and matrix $\underline{\underline{T}}$ that translates points 5 units to the left. We wish to translate points, then scale them up. Given a point $\underline{p}$ we could write the translation step as:

$$\underline{p'} = \underline{\underline{T}}\underline{p} \tag{3.17}$$

and the subsequent scaling step as:

$$\underline{p''} = \underline{\underline{S}}\underline{p'} \tag{3.18}$$

where $\underline{p''}$ is the resultant location of our point. However we could equally have written:

$$\underline{p''} = \underline{\underline{S}}(\underline{\underline{T}}\underline{p}) \tag{3.19}$$

and because matrix multiplication is associative, we can write:

$$\begin{aligned} \underline{p''} &= \underline{\underline{S}}\underline{\underline{T}}\underline{p} \\ \underline{p''} &= \underline{\underline{M}}\underline{p} \end{aligned} \tag{3.20}$$

where $\underline{\underline{M}} = \underline{\underline{S}}\underline{\underline{T}}$. Thus we combine $\underline{\underline{S}}$ and $\underline{\underline{T}}$ to produce a 3×3 matrix $\underline{\underline{M}}$ that has exactly the same effect as multiplying the point by $\underline{\underline{T}}$ and then by $\underline{\underline{S}}$.

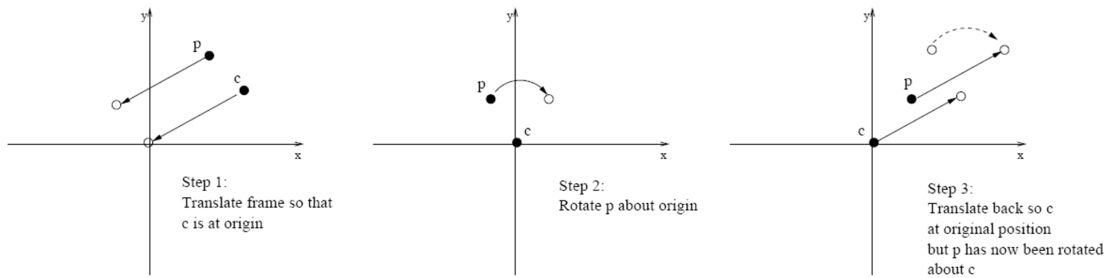


Figure 3.3: Illustrating the three steps (matrix transformations) required to rotate a 2D point about an arbitrary centre of rotation. These three operations are combined in a compound matrix transformation able to perform this 'complex' operation using a single 3×3 matrix.

Order of composition

Recall that matrix operation is non-commutative (Chapter 1) this means that $\underline{\underline{ST}} \neq \underline{\underline{TS}}$. i.e. a translation followed by a scaling about the origin is not the same as a scaling about the origin followed by a translation. You may like to think through an example of this to see why it is not so.

Also note the order of composition - because we are representing our points as column vectors, the order of composition is counter-intuitive. Operations we want to perform first (the translation in this case) are at the right-most end of the chain of multiplications. Operations we perform last are at the left-most end — see eq.(3.20).

2D Rotation about an arbitrary point

We will now look in detail at a common use of compound transformations; generalising the rotation operation to having an arbitrary centre of rotation, rather than being restricted to the origin.

Suppose we wish to rotate a point $\underline{p}$ by θ degrees anti-clockwise about centre of rotation $\underline{c}$. This can be achieved using three matrix multiplications. First, we translate the reference frame in which $\underline{p}$ is defined, so that $\underline{c}$ coincides with the origin (Figure 3.3). If we write $\underline{c} = [c_x \ c_y]^T$ then this translation operation is simply:

$$\underline{\underline{T}} = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.21}$$

Now that the centre of rotation is at the origin, we can apply our basic rotation matrix to rotate the point the user-specified number of degrees θ.

$$\underline{\underline{R(\theta)}} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.22}$$

Finally, we translate the reference frame so that $\underline{c}$ moves back from the origin to its original position. The point $\underline{p}$ has been rotated θ degrees about $\underline{c}$. Figure 3.3 illustrates.

Our operation was therefore first a translation, then rotation, then another translation (the reverse of the first translation). This is written:

$$\begin{aligned} \underline{p}' &= \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \underline{p} \\ \underline{p}' &= \underline{\underline{T}}^{-1} \underline{\underline{R}}(\theta) \underline{\underline{T}} \underline{p} \\ \underline{p}' &= \underline{\underline{M}} \underline{p} \end{aligned} \tag{3.23}$$

where $\underline{\underline{M}}$ is the compound matrix transformation $\underline{\underline{T}}^{-1} \underline{\underline{R}}(\theta) \underline{\underline{T}}$.

An important observation is that matrix transformations down-stream from a particular transformation (e.g. the first translation) operate in the reference frame output by that transformation. This is how we were able to use our "standard" rotate-about-the-origin-anticlockwise matrix to create a rotation about an arbitrary point. This observation is important in the next section and also when we discuss 3D rotation.

We could use this compound transformation to create an animation of a point orbiting another, by successively applying $\underline{\underline{M}}$ at each time-step to rotate point $\underline{p}$ a further θ degrees about $\underline{c}$. After one time-step the position of the point would be $\underline{\underline{M}}\underline{p}$, after the second time-step the position would be $\underline{\underline{M}}\underline{\underline{M}}\underline{p}$ and so on until the nth time-step where the point is at $\underline{\underline{M}}^n \underline{p}$. Pre-computing $\underline{\underline{M}}$ saves us a lot of computational cost at each iteration of the animation.

We can also use the same compound approach to scale about an arbitrary point or in an arbitrary direction, or perform any number of other affine transformations.

3.2.8 Animation Hierarchies

We can use the principles of compound matrix transformation to create more complex animations; where objects move around other objects, that are themselves moving. We will look at two illustrative examples; the Moon orbiting the Earth, and a person walking.

Earth and Moon

Suppose we have a set of points $\underline{\underline{e}} = \begin{bmatrix} \underline{e_1} & \underline{e_2} & ... & \underline{e_n} \end{bmatrix}$ that describe vertices of a polygon representing the Earth, and a set of points $\underline{\underline{m}}$ that similarly approximate the Moon. Both polygons are squares centred at the origin; the moon's square is smaller than the earth. Figure 3.4 (left) illustrates. We wish to animate both the Earth and Moon rotating in space, and the Moon also orbiting the Earth.

Modelling the Earth and Moon rotating in space is simple enough; they are already centred at the origin, so we can apply the standard rotation matrix to each of them:

$$\underline{\underline{R}}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.24}$$

We increase θ as time progresses throughout the animation, to cause the polygons to spin in-place. We could even use two variables θ_e and θ_m, for the Earth and Moon respectively,

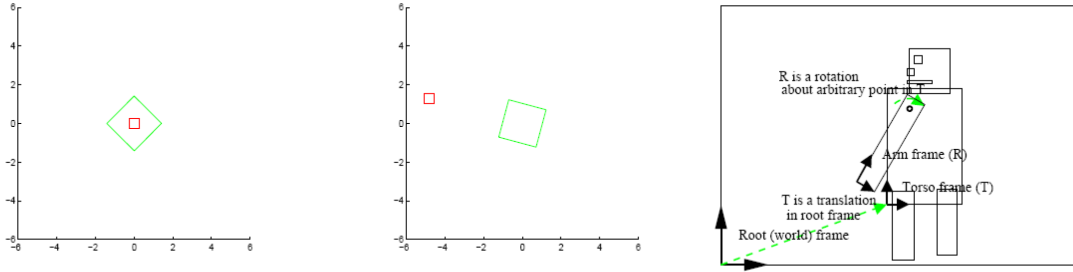


Figure 3.4: Illustrating animation hierarchies. Left, middle: Steps toward animation of a spinning Moon orbiting a spinning Earth (see subsection 3.2.8). Right: A hierarchy for an articulated body such as a walking person. The torso's frame moves within the root reference frame, and limbs directly attached to the torso move within the reference frame defined by the torso.

to enable the rotation speeds to differ e.g. if $\theta_m = 5\theta_e$ then the Moon would rotate 5 times faster than the Earth.

$$
\begin{aligned}
\underline{e}' &= \underline{\underline{R}}(\theta_e)\underline{e} \\
\underline{m}' &= \underline{\underline{R}}(\theta_m)\underline{m}
\end{aligned} \quad (3.25)
$$

Currently the Moon is spinning 'inside' the Earth, and so should be moved a constant distance away with a translation. The translation should occur after the rotation, because we need to rotate while the Moon is centred at the origin:

$$
\begin{aligned}
\underline{\underline{T}} &= \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
\underline{m}' &= \underline{\underline{T}}\,\underline{\underline{R}}(\theta_m)\underline{m}
\end{aligned} \quad (3.26)
$$

At this stage the Earth is spinning on its centre at the origin, and the Moon is spinning on its centre to the right of the Earth. We would like to 'connect' the two, so that the Moon not only spins on its axis but also rotates with the Earth. We must therefore **place the Moon within the Earth's reference frame**. We do this by **pre-multiplying** eq.(3.26) with $\underline{\underline{R}}(\theta_e)$.

$$
\underline{m}' = \underline{\underline{R}}(\theta_e)\underline{\underline{T}}\,\underline{\underline{R}}(\theta_m)\underline{m} \quad (3.27)
$$

This results in the correct animation (Figure 3.4, middle). Recall our discussion of the passive interpretation of transformations. This final multiplication by $\underline{\underline{R}}(\theta_e)$ is treating the Moon's coordinates as being written with respect to a reference frame rotating with the Earth. Multiplying by $\underline{\underline{R}}(\theta_e)$ yields the Moon's coordinates in the root reference frame, and so tells us where to draw the Moon on our display.

Walking person

In the above example we saw the Moon rotating within the reference frame of another object, the Earth. The Earth was itself rotating within the root reference frame. This idea generalises to the concept of a hierarchy (specifically a n-ary tree) of reference frames, in which

nodes represent reference frames that are linked to (move with) their parent reference frames in the tree.

A person's body structure consists of limbs that can pivot on other limbs. As such, the position of a limb (e.g. an arm) is not only a function of the angle it makes about its pivot (e.g. shoulder) but also a function of the position of the torso. Such structures (comprising rigid, pivoting limbs) are called articulated bodies and are amenable to modelling using a hierarchy of reference frames. Figure 3.2.8 (right) illustrates.

For example, consider a person's torso moving in the world. Its location in the world can be described by some affine transformation given by a matrix $\underline{\underline{T_t}}$ that transforms the torso from a pre-specified constant reference point to its current position in the world. It may also be subject to some rotation $\underline{\underline{R_t}}$. We can think of the torso being specified about the origin of its own local "torso" reference frame, defined by $\underline{\underline{T_t R_t}}$. So if we consider vertices of the torso polygon $\underline{t}$, defined in that local frame, we would actually draw vertices $\underline{\underline{T_t R_t}}\underline{t}$ i.e. the coordinates of $\underline{t}$ in the root (world) reference frame.

Now consider an upper-arm attached to the torso; its position is specified relative to the torso. Just like the torso, the upper-arm is defined about its own origin; that origin is offset from the torso's origin by some translation $\underline{\underline{T_u}}$, and may rotate about its own origin (i.e. the shoulder) using a matrix $\underline{\underline{R_s}}$. If we consider the upper-arm polygon $\underline{s}$ defined within its local reference frame, then its coordinates within the torso reference frame are $\underline{\underline{T_s R_s}}\underline{s}$. And its coordinates within the root (world) reference frame are:

$$\underline{\underline{T_t R_t T_s R_s}}\underline{s} \tag{3.28}$$

So, just as with the Earth, we pre-multiplied by the matrix ($\underline{\underline{T_t R_t}}$) describing the torso's reference frame. Other, independent, matrices exist to describe the other upper-arm, legs and so on. The lower-arm $\underline{q}$'s position about elbow might be given by $\underline{\underline{T_q R_q}}\underline{q}$, and its absolute position therefore by $\underline{\underline{T_t R_t T_s R_s T_q R_q}}\underline{q}$, and so on.

Thus we can use compound matrix transformations to control animation of an articulated body. We must be careful to design our hierarchy to be as broad and shallow as possible; matrix multiplications are usually performed in floating point which is an imprecise representation system for Real numbers. The inaccuracies in representation are compounded with each multiplication we perform, and can quickly become noticeable after 5 or 6 matrix multiplications in practice. For example, if we chose the root of our hierarchical model as the right foot, we would have to perform more matrix multiplications to animate the head, say, than if we chose the torso as the root of our tree. Articulated bodies are very common in Computer Graphics; it is not always obvious how to design a animation hierarchy to avoid this problem of multiplicative error and often requires some experimentation for bodies with large numbers of moving parts.

3.3 3D Rigid Body Transformations

So far we have described only 2D affine transformations using homogeneous coordinates. However the concept generalises to any number of dimensions; in this section we explore 3D

transformations. In 3D a point $(x, y, z)^T$ is written $(\alpha x, \alpha y, \alpha z, \alpha)^T$ in homogeneous form. Rigid body transformations therefore take the form of 4×4 matrices.

We can perform 3D translation ($\underline{\underline{T}}$) and scaling ($\underline{\underline{S}}$) using the following matrices:

$$\underline{\underline{T}} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.29}$$

$$\underline{\underline{S}} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.30}$$

where S_x, S_y, S_z are the scale factors in the x, y and z directions respectively, and T_x, T_y, T_z are similarly the shifts in the x, y and z directions. Rotation in 3D is slightly more complicated.

3.3.1 Rotation in 3D — Euler Angles

There is no concept of rotation about a point in 3D, as there is in 2D. The analogous concept is rotation about an axis. Whereas there was one fundamental 'building block' matrix to describe rotation in 2D, there are 3 such matrices in 3D. These matrices enable us to rotate about the x-axis, y-axis and z-axis respectively. These rotation operations are sometimes given special names, respectively "**roll**", "**pitch**" and "**yaw**". The following three matrices will perform a rotation of θ degrees clockwise in each of these directions respectively. Note that our definition of "clockwise" here assumes we are "looking along" along the principal axis in the positive direction:

$$\underline{\underline{R_x}}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.31}$$

$$\underline{\underline{R_y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.32}$$

$$\underline{\underline{R_z}}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.33}$$

We can effect any rotation we desire in 3D using some angular quantity of roll, pitch and yaw. This would be performed using some sequence of $\underline{\underline{R_x}}$, $\underline{\underline{R_y}}$, and $\underline{\underline{R_z}}$ i.e. a compound matrix transformation. We will see in subsection 3.3.3 that the order of multiplication does matter, because particular orderings prohibit particular rotations due to a mathematical phenomenon known as "Gimbal lock". This system of rotation is collectively referred to as **Euler angles**.

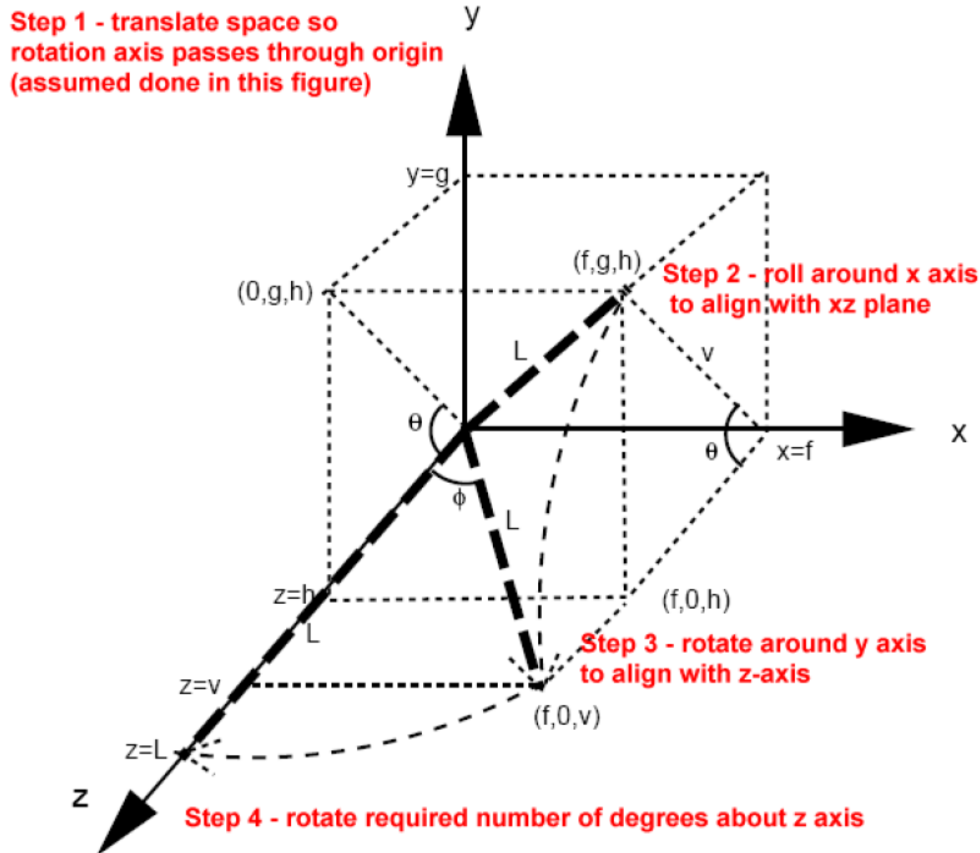


Figure 3.5: Illustrating the rotation of a point about an arbitrary axis in 3D – to be read in conjunction with the description in subsection 3.3.2. The figure assumes the axis of rotation (L) already passes through the origin.

3.3.2 Rotation about an arbitrary axis in 3D

In subsection 3.2.7 we saw that rotation about an arbitrary point could be performed using a compound matrix transformation. Rotation degrees about an arbitrary axis can be achieved using a generalisation of that process (see also Figure 3.5). The seven steps of this process are:

- Given an arbitrary axis to rotate about, we first translate the 3D space using a 4×4 translation matrix $\underline{\underline{T}}$ so that the axis passes through the origin.
- The axis is then rotated so that it lies in one of the principal planes of the 3D space. For example we could roll around the x-axis using $\underline{\underline{R_x}}$ so that the arbitrary axis lies in the xz-plane.
- We then perform a rotation about another axis e.g. the y-axis (via $\underline{\underline{R_y}}$ to align the axis in the xz-plane with one of the principal axes e.g. the z-axis.
- Now that the axis of rotation lies along one of the principal axes (i.e. the z-axis, we can apply $\underline{\underline{R_z}}$ to rotate the user-specified number of degrees θ.

- Transform by the inverse of $\underline{\underline{R_y}}$.
- Transform by the inverse of $\underline{\underline{R_x}}$.
- Transform by the inverse of $\underline{\underline{T}}$.

So the complete transformation is $\underline{p'} = \underline{\underline{T}}^{-1}\underline{\underline{R_x}}^{-1}\underline{\underline{R_y}}^{-1}\underline{\underline{R_z}}\,\underline{\underline{R_y}}\,\underline{\underline{R_x}}\,\underline{\underline{T}}\underline{p}$. Note that we could have constructed this expression using a different ordering of the rotation matrices — the decision to compose the transformation in this particular ordering was somewhat arbitrary.

Inspection of this transformation reveals that 2D rotation about a point is a special case of 3D rotation, when one considers that rotation about a point on the xy-plane is equivalent to rotation about the z-axis (consider the positive z-axis pointing 'out of' the page). In this 2D case, steps 2-3 (and so also 5-6) are redundant and equal to the identity matrix; because the axis of rotation is already pointing down one of the principle axes (the z-axis). Only the translations (steps 1 and 7) and the actual rotation by the user requested number of degrees (step 3) have effect.

Determining values for the $\underline{\underline{T}}$ and $\underline{\underline{R}}$ matrices

Clearly the value of $\underline{\underline{R_z}}$ is determined by the user who wishes to rotate the model, and specifies a value of degrees for the rotation. However the values of $\underline{\underline{T}}$, $\underline{\underline{R_x}}$, $\underline{\underline{R_y}}$ are determined by the equation of the line (axis) we wish to rotate around.

Let us suppose the axis of rotation (written $\underline{L}(s)$) has the following parametric equation (see Chapter 6 for parametric equation of a line):

$$\underline{L}(s) = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + s \begin{bmatrix} f \\ g \\ h \end{bmatrix} \tag{3.34}$$

Then a translation matrix that ensures the line passes through the origin is:

$$\underline{\underline{T}} \;=\; \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.35}$$

With the rotation axis L passing through the origin (i.e. Step 1 of the 7 completed) we have the setup shown in Figure 3.5. To obtain a value for $\underline{\underline{R_x}}$ we need to know angle , and for $\underline{\underline{R_y}}$ we need to know angle . Fortunately the rotation matrices require only the sines and cosines of these angles, and so we can use ratios in the geometry of Figure 3.5 to determine these matrices.

Writing $v = \sqrt{g^2 + h^2}$ we can immediately see that $\sin\alpha = g/v$ and $\cos\alpha = h/v$. These are the values plugged into the "roll" or x-axis rotation matrix for $\underline{\underline{R_x}}(\alpha)$.

The length of line L in the construction is given by Pythagoras i.e. $l = \sqrt{f^2 + g^2 + h^2}$. This leads to the following expressions for the β angle: $\sin\beta = f/l$ and $\cos\beta = v/l$, which are plugged into the "pitch" or y-rotation matrix for $\underline{\underline{R_y}}$.

3.3.3 Problems with Euler Angles

Recall the discussion of compound matrix transformations in subsection 3.2.7. We observed that each matrix in the sequence of transformations operates in the reference frame of the previous matrix. We use this observation to offer a geometric explanation of Euler angles (Figure 3.6), which in turn reveals some of the shortfallings of this system.

Consider a plate upon the surface of which points are placed. This is analogous to the 3D reference frame in which points are defined. The plate can pivot within a surrounding mechanical frame, allowing the plate to tilt to and fro. This is analogous to a rotation by one of the Euler angle matrices – e.g. $\underline{\underline{R_x}}$. Now consider a mechanical frame surrounding the aforementioned frame, allowing that frame to rotate in an orthogonal axis — this is analogous to rotation by another Euler angle matrix, e.g. $\underline{\underline{R_y}}$. Finally consider a further mechanical frame surrounding the frame of $\underline{\underline{R_y}}$, allowing that frame to pivot in a direction orthogonal to both the two inner frames – this is analogous to rotation by the final Euler angle matrix i.e. $\underline{\underline{R_z}}$. Figure 3.6 illustrates this mechanical setup, which is called a **gimbal**.

It is clear that the points on the plate are acted upon by $\underline{\underline{R_x}}$, which is in turn acted upon by $\underline{\underline{R_y}}$, which is in turn acted upon by $\underline{\underline{R_z}}$. If we write the points on the plate as $\underline{p}$ then we have:

$$\underline{\underline{R_z}}\,\underline{\underline{R_y}}\,\underline{\underline{R_x}}\,\underline{p} \tag{3.36}$$

Note that we could have configured the system to use any ordering of the frames e.g. $\underline{\underline{R_x}}\,\underline{\underline{R_z}}\,\underline{\underline{R_y}}$, and so on. But we must choose an ordering for our system.

Now consider what happens when we set θ in the middle frame i.e. $\underline{\underline{R_y}}$ to 90 . The axis of rotation of $\underline{\underline{R_x}}$ is made to line up with the axis of rotation of $\underline{\underline{R_z}}$. We are no longer able to move in the direction that $\underline{\underline{R_x}}$ previously enabled us too; we have lost a degree of freedom. An Euler angle system in this state is said to be in **gimbal lock**.

This illustrates one of the major problems with the Euler angle parameterisation of rotation. Consideration of rotation as a roll, pitch and yaw component is quite intuitive, and be useful in a graphics package interface for animators. But we see that without careful planning we can manouver our system into a configuration that causes us to lose a degree of freedom. Gimbal lock can be 'broken' by changing the rotation parameter on the matrix that has caused the lock (e.g. $\underline{\underline{R_y}}$) or by adding a new reference frame outside the system, e.g.

$$\underline{\underline{R'_x}}\,\underline{\underline{R_z}}\,\underline{\underline{R_y}}\,\underline{\underline{R_x}} \tag{3.37}$$

However adding a new reference frame is inadvisable; we may quickly find ourselves in Gimbal lock again, and feel motivated to add a further frame, and so on – each time the animator gains a new parameter to twiddle on the rotation control for his model, and this quickly becomes non-intuitive and impractical.

The best solution to avoid Gimbal lock is not to use Euler Angles at all, but a slightly more sophisticated form of rotation construct called a **quaternion** (which can also be expressed as a 4×4 matrix transformation). However quaternions are beyond the scope of this introductory course; you might like to refer to Alan Watt's "Advanced Animation and Rendering Techniques" text for more information.

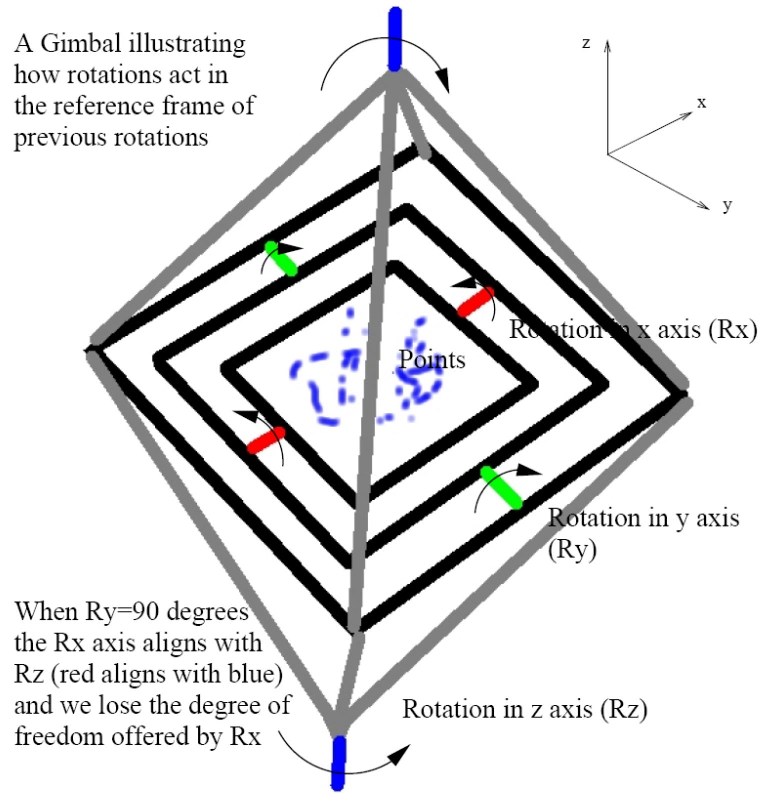


Figure 3.6: A mechanical gimbal, showing how points (defined in a space at the centre of the Gimbal) are acted upon by rotation operations along the x, y and z axes. Setting the middle Gimbal at 90 degrees causes Gimbal lock; the axes of rotation of the inner and outer frames become aligned.

Impact of Gimbal Lock on Articulated Motion

We have seen that each Euler angle rotation matrix operates in the reference frame of the previous matrix in the chain. Similarly, each matrix transformation in the matrix hierarchy of an articulated structure operates within the frame of the previous. Suppose we have an articulated structure, such as a walking person (subsection 3.2.8), with hierarchy spanning three or more nodes from root to leaf of the tree. Referring back to our previous example, we might have a torso, upper-arm and low-arm. The points of the lower-arm $\underline{l}$ have the following location in the root reference frame:

$$\underline{\underline{T_t}}\,\underline{\underline{R_t}}\,\underline{\underline{T_u}}\,\underline{\underline{R_u}}\,\underline{\underline{T_l}}\,\underline{\underline{R_l}}\,\underline{l} \tag{3.38}$$

Suppose the $\underline{\underline{R}}$ in the above equation are specified by Euler angles (e.g. $\underline{\underline{R}} = \underline{\underline{R_z}}\,\underline{\underline{R_y}}\,\underline{\underline{R_x}}$). Certain combinations of poses (e.g. where $\underline{\underline{R_u}}$ is a $90°$ rotation about the y-axis) may cause parts of the model to fall into Gimbal lock, and so lose a degree of freedom. This would cause animators difficulty when attempting to pose parts of the model (e.g. $\underline{l}$ correctly).

The solution is to introduce a manually designed local reference frame for each limb, and specify the Euler Angle rotation *inside* that frame. For example, instead of writing $\underline{\underline{T_l}}\,\underline{\underline{R_l}}$ we

write $\underline{\underline{T_l}}(\underline{\underline{K_l}}^{-1}\underline{\underline{R_l K_l}})$ or maybe even $\underline{\underline{K_l}}^{-1}(\underline{\underline{T_l R_l}})\underline{\underline{K_l}}$.[1]. The final chain might end up as:

$$\underline{\underline{T_t}}(\underline{\underline{K_t}}^{-1}\underline{\underline{R_t K_t}})\underline{\underline{T_u}}(\underline{\underline{K_u}}^{-1}\underline{\underline{R_u K_u}})\underline{\underline{T_l}}(\underline{\underline{K_l}}^{-1}\underline{\underline{R_l K_l}})\underline{l} \tag{3.39}$$

Suppose we had an articulated body where $\underline{\underline{R_u}}$ was likely to be a rotation in the y-axis 90° due to the nature of the motion being modelled. We might specify a $\underline{\underline{K_u}}$ that rotated the y-axis to align with another axis, to prevent this problematic transformation being introduced into the matrix chain for later reference frames (i.e. $\underline{\underline{R_l}}$).

Non-sudden nature of Gimbal Lock

Recall Figure 3.6 where the system is in Gimbal lock when $\underline{\underline{R_y}}$ is at 90°. It is true that the system is in Gimbal lock only at 90° (i.e. we completely lose or "lock out" one degree of freedom). However our system is still very hard to control in that degree of freedom at 89°. The problems inherent to Gimbal lock (i.e. loss of control) become more troublesome as we approach 90°; they are not entirely absent before then.

3.4 Image Formation – 3D on a 2D display

We have talked about 3D points and how they may be manipulated in space using matrices. Those 3D points might be the vertices of a cube, which we might rotate using our Euler angle rotation matrices to create an animation of a spinning cube. They might form more sophisticated objects still, and be acted upon by complex compound matrix transformations. Regardless of this, for any modelled graphic we must ultimately create a 2D image of the object in order to display it.

Moving from a higher dimension (3D) to a lower dimension (2D) is achieved via a **projection** operation; a lossy operation that too can be expressed in our 4×4 matrix framework acting upon homogeneous 3D points. Common types of projection are **perspective projection** and **orthographic projection**. We will cover both in this Section.

3.4.1 Perspective Projection

You may already be familiar with the concept of perspective projection from your visual experiences in the real world. Objects in the distance appear smaller that those close-by. One consequence is that parallel lines in the real world do not map to parallel lines in an image under perspective projection. Consider parallel train tracks (tracks separated by a constant distance) running into the distance. That distance of separation appears to shrink the further away the tracks are from the viewer. Eventually the tracks appear to converge at a **vanishing point** some way in to the distance.

Sometimes illustrators and draftsmen talk of "1-point perspective" or "2-point perspective". They are referring to the number of vanishing points present in their drawings. For example, it is possible to draw a cube in 1 or even 3 point perspective — simply by varying the point of view from which it is drawn. This terminology is largely redundant for purposes of Computer Graphics; whether a rendering is characterised as having n-point perspective has no bearing

[1]The bracketing here is redundant and for illustration only

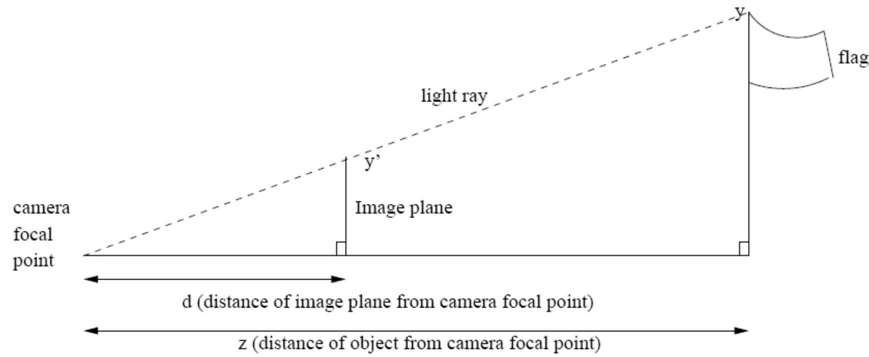


Figure 3.7: Schematic illustrating the perspective projection of a 3D flagpole to create a 3D image. Light rays travel in straight lines from points on the flagpole to a eye/camera focal point, passing through a planar surface representing the image to be rendered.

on the underlying mathematics that we use to model perspective.

We will now derive equations for modelling perspective projection in a computer graphics system. Consider a flag pole in a courtyard, which we observe from within a building through a window. Rays of light (travelling in straight lines) reflect from visible points on the flagpole and into our eye via the window. Figure 3.7 illustrates this system, considering for clarity only the y-axis and z-axis of the 3D setup. In this setup the tip of the flagpole is z distance from us, and y high. The image of the flagpole is y high on a window d distance from us.

The geometry of the scene is a system of similar triangles:

$$\frac{z}{d} = \frac{y}{y'} \tag{3.40}$$

which we can rearrange to get an expression for y' (height of the flagpole on the window) in terms of the flagpole's height y and distance z from us, and the distance between us and the window d:

$$y' = \frac{dy}{z} \tag{3.41}$$

We see from this ratio that increasing the flagpole's height creates a larger image on the window. Increasing the flagpole's distance from us decreases the size of the flagpole's image on the window. Increasing the distance d of the window from our eye also increases the size of the flagpole's image (in the limit when $d = z$ this will equal the actual height of the flagpole). Exactly the same mathematics apply to the x component of the scene, i.e.

$$x' = \frac{dx}{z} \tag{3.42}$$

Thus the essence of perspective projection is division by the z coordinate; which makes sense as size of image should be inversely proportional to distance of object.

Cameras work in a similar manner. Rays of light enter the camera and are focused by a lens to a point (analogous to the location of our eye in the flagpole example). The rays of light

'cross' at the **focal point** and fall upon a planar imaging sensor analogous to the window in our flagpole example. Due to the cross-over of rays at the focal point, the image is upside down on the image sensor and is inverted by the camera software. In this example we have assumed a "**pin-hole camera**" (camera obscura), i.e. without a curved lens. These were the earliest form of camera, developed during the Renaissance, and most modern cameras use curved glass lenses leading to more complicated projection models as light is bent along its path from object to sensor. However the pin-hole camera assumption is acceptable for most Graphics applications.

The distance d between the camera focal point and the image plane is called the **focal length**. Larger focal lengths create images contain less of the scene (narrower **field of view**) but larger images of the objects in the scene. It is similar to a zoom or **telephoto lens**. Small focal lengths accommodate larger fields of view, e.g. a **wide-angle lens**.

Matrix transformation for perspective projection

We can devise a matrix transformation $\underline{\underline{P}}$ that encapsulates the above mathematics. Given a 3D point in homogeneous form i.e. $(x, y, z, 1)^T$ we write:

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix} \tag{3.43}$$

Note that the homogeneous coordinate is no longer unchanged, it is z. Therefore we must normalise the point by dividing by the homogeneous coordinate. This leads to the point $(dx/z, dy/z, dz/z = d)^T$. The transformation has projected 3D points onto a 2D plane located at $z = d$ within the 3D space. Because we are usually unconcerned with the z coordinate at this stage, we sometimes write the transformation omitting the third row of the perspective projection matrix:

$$\underline{\underline{P}} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.44}$$

leading to:

$$\underline{\underline{P}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ z \end{bmatrix} \tag{3.45}$$

A matrix with equivalent functionality is:

$$\underline{\underline{P}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \tag{3.46}$$

Finally, note that any points "behind" the image plane will be projected onto the image plane upside down. This is usually undesirable, and so we must inhibit rendering of such points — a process known as point "culling" — prior to applying the projection transformation.

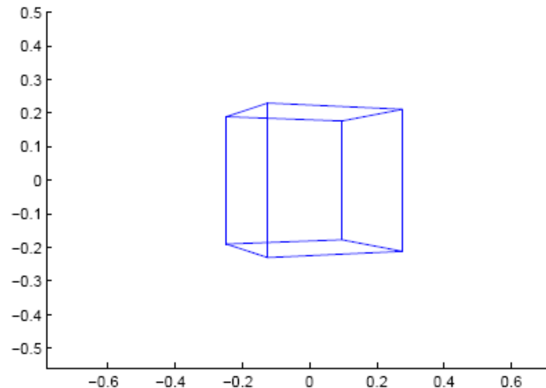


Figure 3.8: A spinning 3D cube is animated by manipulating its position in space using 3D rigid body transformations (4×4 matrix transforms), and projected on the 2D screen using the perspective projection matrix.

Putting it all together

With knowledge of perspective projection, we are finally in a position to create a full 3D animation. Imagine that we wish to create a single frame of animation for a spinning wire-frame cube.

First, we model the cube as a set of points corresponding to the cube's vertices. We define associations between vertices that represent edges of the cube.

Second, we apply various 3D matrix transformations (say a translation $\underline{\underline{T}}$ followed by a rotation $\underline{\underline{R}}$) to position our cube in 3D space at a given instant in time.

Third, we create a 2D image of the cube by applying the perspective matrix $\underline{\underline{P}}$.

The compound matrix transformation would be $\underline{\underline{PRT}}$, which we apply to the vertices of the cube. Note that the perspective transformation is applied last. Once the locations of the vertices are known in 2D, we can join associated vertices by drawing lines between them in the image. Figure 3.8 illustrates.

3.4.2 Orthographic Projection

A less common form of projection is **orthographic projection** (sometimes referred to as orthogonal projection). Put simply we obtain 2D points from a set of 3D points by just dropping the z coordinate; no division involved. This results in near and distant objects projecting to the image as the same size, which is sometimes desirable for engineering or

architectural applications. This is achievable using the matrix:

$$\underline{\underline{P}} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.47}$$

3.5 Homography

The final matrix transformation we shall discuss is the **homography**. The homography is the general 3×3 matrix transformation that maps four 2D points (a quadrilateral) to a further set of four 2D points (another quadrilateral); the points being in homogeneous form. It has several major applications in Computer Vision and Graphics, one of which is stitching together digital images to create panoramas or "mosaics". We will elaborate on this application in a moment, but first show how the homography may be computed between two sets of corresponding homogeneous 2D points $\underline{\underline{p}} = [\underline{p_1}\ \underline{p_2}\ ...\ \underline{p_n}]$ and $\underline{\underline{q}} = [\underline{q_1}\ \underline{q_2}\ ...\ \underline{q_n}]$, where for now $n = 4$. For this section, we introduce the notation $\underline{p^i_x}$ to indicate, for example, "the x component of the ith point $\underline{p}$".

First we observe that '**computing the homography**' means finding the matrix $\underline{\underline{H}}$ such that $\underline{\underline{H}}\underline{\underline{p}} = \underline{\underline{q}}$. This matrix is a 3×3 rigid body transformation, and we expand the equation $\underline{\underline{H}}\underline{\underline{p}} = \underline{\underline{q}}$ to get:

$$\begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} p^i_x \\ p^i_y \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha q^i_x \\ \alpha q^i_y \\ \alpha \end{bmatrix} \tag{3.48}$$

Our task is to find $h_{1..9}$ to satisfy all $i = [1, n]$ points. We can rewrite eq.(3.48) out as individual linear equations:

$$h_1 p^i_x + h_2 p^i_y + h_3 = \alpha q^i_x \tag{3.49}$$

$$h_4 p^i_x + h_5 p^i_y + h_6 = \alpha q^i_y \tag{3.50}$$

$$h_7 p^i_x + h_8 p^i_y + h_9 = \alpha \tag{3.51}$$

And substitute eq.(3.51) into eq.(3.49) and eq.(3.50) to get:

$$h_1 p^i_x + h_2 p^i_y + h_3 = h_7 p^i_x q^i_x + h_8 p^i_y q^i_x + h_9 q^i_x \tag{3.52}$$

$$h_4 p^i_x + h_5 p^i_y + h_6 = h_7 p^i_x q^i_y + h_8 p^i_y q^i_y + h_9 q^i_y \tag{3.53}$$

Rearranging these two equations we get:

$$h_1 p^i_x + h_2 p^i_y + h_3 - h_7 p^i_x q^i_x - h_8 p^i_y q^i_x - h_9 q^i_x = 0 \tag{3.54}$$

$$h_4 p^i_x + h_5 p^i_y + h_6 - h_7 p^i_x q^i_y - h_8 p^i_y q^i_y - h_9 q^i_y = 0 \tag{3.55}$$

We can then write these two equations as a homogeneous linear system (in the form $\underline{\underline{A}}\underline{x} = 0$) which we duplicate for each ith point:

Figure 3.9: Demonstrating an image stitching operation facilitated by homography estimation. Four corresponding points in each image were identified. The homography between those points was computed. The homography was then used to warp the image from one 'point of view' to another; i.e. move all pixels in the image according to the homography (Digital image warping is covered in the next section). The two images were then merged together.

$$\begin{bmatrix} p_x^1 & p_y^1 & 1 & 0 & 0 & 0 & -p_x^1 q_x^1 & -p_y^1 q_x^1 & -q_x^1 \\ 0 & 0 & 0 & p_x^1 & p_y^1 & 1 & -p_x^1 q_y^1 & -p_y^1 q_y^1 & -q_y^1 \\ p_x^2 & p_y^2 & 1 & 0 & 0 & 0 & -p_x^2 q_x^2 & -p_y^2 q_x^2 & -q_x^2 \\ 0 & 0 & 0 & p_x^2 & p_y^2 & 1 & -p_x^2 q_y^2 & -p_y^2 q_y^2 & -q_y^2 \\ \ldots \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \underline{0} \tag{3.56}$$

We call $\underline{\underline{A}}$ the **design matrix**, and is completely defined by the point correspondences. Vector $\underline{x}$ is our solution vector. We can use standard techniques such as **SVD (Singular Value Decomposition)** to solve this system for $\underline{x}$ and so recover $h_{1..9}$ i.e. matrix $\underline{\underline{H}}$. In brief the SVD algorithm 'decomposes' $\underline{\underline{A}}$ into three matrices such that $\underline{\underline{A}} = \underline{\underline{U}}\underline{\underline{S}}\underline{\underline{V}}^T$. We will discuss SVD and matrix decomposition in Chapter 5. In the context of this problem, we will treat SVD as a "black box"; the columns making up output $\underline{\underline{V}}$ are all possible solutions for $\underline{x}$. The value in $S_{i,i}$ is the error for the solution in the ith column of $\underline{\underline{V}}$. For number of points $n = 4$ there will always be an exact solution. If $n > 4$ then the $\underline{\underline{H}}$ computed will be a "best fit" mapping between all n point correspondences, and may not have zero error. The system is under-determined if $n < 4$. Further details are beyond the scope of this course.

3.5.1 Applications to Image Stitching

We can apply the homography transformation to perform a popular Computer Graphics task; "stitching together" two overlapping photographs taken of an object e.g. two fragments of a panoramic mountain view. By "overlapping photographs" we mean two photographs that image a common part of the object. The motivation for doing this might be that the object is too large to fit in a single image. Figure 3.9 illustrates this application.

Given two images I_1 and I_2, we manually pick 4 points in I_1 and four corresponding points in I_2. These can be any four points that are not co-linear; e.g. 4 corners of a doorway that is visible in both images. Writing these four points as $\underline{p}$ and $\underline{q}$ we can apply the mathematics of the previous section to compute the homography $\underline{\underline{H}}$ between those points. Thus the homography describes an exact mapping from the coordinates of the doorway in I_1 (i.e. $\underline{p}$) to the coordinates of the doorway in I_2 (i.e. $\underline{q}$).

Now, if the subject matter of the photograph is flat i.e. such that all the imaged points lie on a plane in 3D, then the homography also gives us a mapping from any point in I_1 to any point in I_2. Thus, if we had taken a couple of photographs e.g. of a painting on a flat wall, the homography would be a perfect mapping between all points in I_1 to all corresponding points in I_2 . A proof is beyond the scope of this course, but you may like to refer to the text "Multiple View Geometry" by Hartley and Zisserman for more details.

It turns out that very distant objects (e.g. mountains/landscapes) can be approximated as lying on a plane, and in general small violations of the planar constraint do not create large inaccuracies in the mapping described by $\underline{\underline{H}}$. This is why the homography is generally useful for stitching together everyday outdoor images into panoramas, but less useful in stitching together indoor images where multiple objects exist that are unlikely to be co-planar in a scene.

With the homography obtained we can warp pixels to I_1 to new positions, which should match up with the content in I_2 . As the homography is simply a 3×3 matrix transformation, we can effect this operation using standard image warping techniques — as discussed in Section 3.6. The result is that the image content of I_1 is transformed to the point of view from which I_2 was taken. The warped I_1 and original I_2 images may then be overlayed to create a final panorama (or trivial colour blending operations applied to mitigate any inaccuracies in the mapping; e.g. caused by violation of the plane-plane mapping assumption of the homography).

It is possible to obtain the 4 point correspondences ($\underline{p}$ and $\underline{q}$) automatically, i.e. as most photo stitching programs do without requiring manual identification of the matching points. We can use Computer Vision algorithms to identify stable "interest points" in images; that is, points that we can repeatably identify regardless of the point of view from which an image is taken. Often we use simple "corner detectors", such as the Harris/Stephens Corner Detector (Alvey 1988). The problem of finding 4 corresponding points is then reduced to finding 4 interest points in one image, and a matching 4 interest points in the other image. This search is usually performed stochastically via RANSAC or a similar algorithm; the details are beyond the scope of this course.

3.6 Digital Image Warping

Throughout this Chapter we have discussed points and how matrices may act upon those points to modify their locations under rigid body transformation. A common application of these techniques is to apply a matrix transformation to a 2D image, e.g. to rotate, skew, or scale a "source" (input) image to create a modified "target" (output) image. This process is known as **digital image warping** and was the motivating example given back in Figure 3.1.

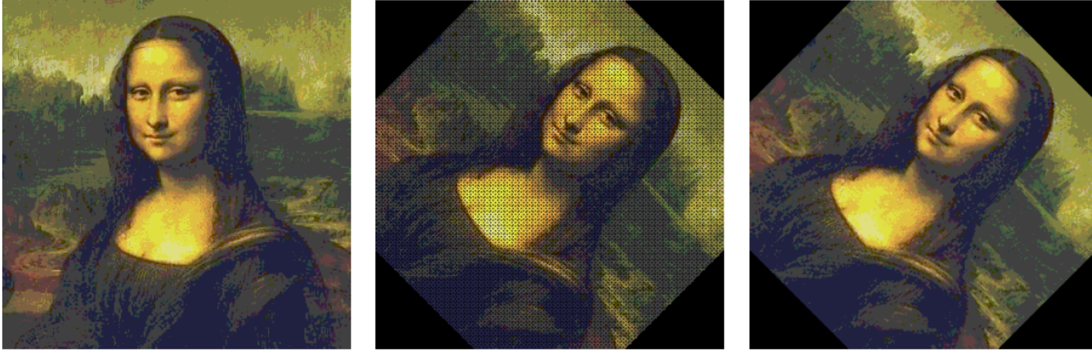


Figure 3.10: An image (left) subjected to a digital image warp using forward mapping (middle) and backward mapping (right). Substantial improvements in quality are achieved using backward mapping.

The most straightforward and intuitive process for performing image warping is to iterate over the source image, treating each pixel as a 2D point $\underline{p} = (x, y)^T$. We apply a matrix transformation $\underline{p}' = \underline{\underline{M}}\underline{p}$ to those coordinates to determine where each pixel "ends up" in our target image, i.e. $\underline{p}' = (x', y')^T$. We then colour in the target image pixel at $\underline{p}$ with the colour of pixel $\underline{p}$ in the source image. This process is called **forward mapping** and example Matlab code for it is given in Figure 3.11.

Unfortunately there are a number of problems with forward mapping. First, the resultant target coordinates $\underline{p}$ are real valued; however pixels are addressed by integer coordinates and so some sort of correction is usually made e.g. rounding $\underline{p}$ to the nearest integer, so that we know which pixel to colour in. This creates aesthetically poor artifacts in the image. Second, many transformations will result in pixels being "missed out" by the warping operation and thus not receiving any colour. Consider a scaling of factor 2. Point $(0,0)^T$ maps to $(0,0)^T$ on the target, point $(1,0)^T$ maps to $(2,0)^T$ on the target, $(2,0)^T$ to $(4,0)^T$ and so on. However pixels with odd coordinates in the target are not coloured in by the algorithm. This leads to 'holes' appearing in the target image (Figure 3.10, middle).

A better solution is **backward mapping** – Figure 3.12 contains code. In backward mapping we iterate over the target image, rather than the source image. For each pixel we obtain integer coordinates $\underline{p}' = (x, y)^T$ which we multiply by $\underline{\underline{M}}^{-1}$ to obtain the corresponding pixel in the source image $\underline{p} = \underline{\underline{M}}^{-1}\underline{p}$. We then colour the target pixel with the colour of the source pixel. This approach does not suffer from the 'holes' of forward mapping, because we are guaranteed to visit and colour each pixel in the target image. The approach is still complicated by the fact that the coordinates $\underline{p} = (x, y)^T$ may not be integer valued. We could simply round the pixel to the nearest integer; again this can create artifacts in the image. Nevertheless the technique produce substantially improved results over forward mapping (Figure 3.10, right).

In practice we can use **pixel interpolation** to improve on the strategy of rounding pixel coordinates to integer values. Interpolation is quite natural to implement in the framework of backward mapping; we try to blend together colours from neighbouring pixels in the source image to come up with an estimate of the colour at real-valued coordinates $\underline{p}$. There are var-

```
source=double(imread('c:\mona.jpg'))./255;

target=zeros(size(source));

T=[1 0 -size(source,2)/2 ; 0 1 -size(source,1)/2; 0 0 1]
t=pi/4;
R=[cos(t) -sin(t) 0 ; sin(t) cos(t) 0 ; 0 0 1]

M=inv(T)*R*T;   % the warping transformation (rotation about arbitrary point)

% the forward mapping loop
for y=1:size(source,1)
    for x=1:size(source,2)

        p=[x ; y ; 1];
        q=M*p;

        u=round(q(1)/q(3));
        v=round(q(2)/q(3));

        if (u>0 & v>0 & u<=size(target,2) & v<=size(target,1))
            target(v,u,:)=source(y,x,:);
        end

    end
end

imshow([source target]);
```

Figure 3.11: Matlab code to warp an image using forward mapping.

ious strategies for this (e.g. bi-linear or bi-cubic interpolation); these are beyond the scope of this course.

However note that the ease with which interpolation may be integrated with backward mapping is another advantage to the approach. In the framework of forward mapping, interpolation is very tricky to implement. We must project the quadrilateral of each pixel forward onto the target image and maintain records of how much (and which) colour has contributed to every pixel in the target image. However forward mapping can give good results if implemented properly, and is the only solution if working with a non-invertible transformation.

```
source=double(imread('c:\mona.jpg'))./255;

target=zeros(size(source));

T=[1 0 -size(source,2)/2 ; 0 1 -size(source,1)/2; 0 0 1]
t=pi/4;
R=[cos(t) -sin(t) 0 ; sin(t) cos(t) 0 ; 0 0 1]

M=inv(T)*R*T;   % the warping transformation (rotation about arbitrary point)

M=inv(M); % note, we invert the matrix because we are backward mapping

% the backward mapping loop
for u=1:size(target,2)
    for v=1:size(target,1)

        q=[u ; v ; 1];
        p=M*q;

        x=round(p(1)/p(3));
        y=round(p(2)/p(3));

        if (x>0 & y>0 & x<=size(source,2) & y<=size(source,1))
            target(v,u,:)=source(y,x,:);
        end

    end
end

imshow([source target]);
```

Figure 3.12: Matlab code to warp an image using backward mapping.

Chapter 4

OpenGL Programming

4.1 Introduction

This chapter describes features of the **OpenGL library** - a 3D graphics programming library that enables you to create high performance graphics applications using the mathematics covered on this course.

Like all software libraries, OpenGL consists of a series of function calls (an **Application Programmers Interface** or 'API') that can be invoked from your own programs. OpenGL has been ported to a number of platforms and languages, but this course will focus on the C programming language. In typical use cases OpenGL is invoked via C or C++; most graphics applications demand efficient and fast-executing code - and this requirement is most often satisfied by C or C++. However it is possible to use OpenGL with other languages such as Java (via the JOGL API).

OpenGL was developed by **Silicon Graphics (SGI)**; a company that pioneered many early movie special effects and graphics techniques. OpenGL was derived from SGI's properietary graphics running on their IRIX operating system. The 'Open' in OpenGL indicates that SGI opened the library up to other operating systems - it does not indicate Open Source. Indeed, there is often little code in an OpenGL library - the library often acts simply as an interface to the graphics hardware in a machine e.g. a PC. However, in cases where a machine has no specialised graphics hardware, an OpenGL library will often emulate its presence. This is the case in Microsoft Windows - if your PC has no specialised graphics hardware, OpenGL will imitate the hardware using software only. The results will be visually (near) identical, but execute much more slowly.

4.1.1 The GLUT Library

If one were to use OpenGL by itself, graphics application programming would not be as simple as might be hoped. It would be necessary to manage the keyboard, mouse, screen/window re-paint requests, etc. manually. This would result in substantial amounts of boilerplate code for even the simplest applications. For this reason the **GLUT library** was developed. GLUT is an additional set of function calls that augment the OpenGL library to manage windowing and event handling for your application. With GLUT it is possible to write a basic OpenGL program in tens of lines of source code, rather than hundreds. We will be using GLUT on this course, to enable us to focus on the graphics issues. Using GLUT also enables us to be

platform independent; the code in this chapter will run with minimal changes on Windows, Mac and Linux platforms.

4.2 An Illustrative Example - Teapot

Any programming course starts with an illustrative example; in graphics courses it is traditional to draw a Teapot. Here is the OpenGL (plus GLUT) code to draw a teapot.

```
#include "GL/gl.h"
#include "GL/glut.h"

void keyboard(unsigned char key, int x, int y);
void reshape(int w, int h);
void display(void);
void idle(void);

void init(void);


int main (int argc, char** argv) {

   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (600, 600);
   glutCreateWindow ("Hello Teapot");

   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutIdleFunc (idle);

   glutMainLoop();
}

void keyboard(unsigned char key, int x, int y) {

   switch (key) {
  case 0x1b:
  exit (0);
  break;
   }
}

void idle()  { }
```

```
void reshape(int w, int h) {

   glViewport(0, 0, w, h);


   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60,(GLfloat)w/(GLfloat)h,2,40.0);
}



void display(void) {

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // glTranslatef(0,0,-10);  // example translation

  glutSolidTeapot(1.0);

  glFlush();
  glutSwapBuffers();
}
```

There are 4 functions in this code. The entry point to the program main(...) creates a window of size 600×600 pixels and sets its title to ”Hello Teapot”. The function sets up a ”callback” for the keyboard. Callbacks are functions without our own code that GLUT will invoke when certain ”events” happen; in this case GLUT will invoke function keyboard(...) when a key is pressed. A callback is also set up for ”displaying”; any time GLUT wants to refresh the graphics on screen the display(...) function will be called. We therefore place our code for drawing graphics inside the display(...) function. Finally a callback is set up for ”reshaping”. This callback is invoked when we resize the window (and also when the window is first drawn).

The most crucial part of the main(...) function is the last line - the glutMainLoop() call. This passes all control in our program to GLUT. GLUT will never return from this call, so it is pointless writing code in main(...) beyond this invocation. We will only gain control again when GLUT invokes any of our callbacks in response to an event. If we call the C function exit(int) during a callback, then our program will end - this is the only way (other than writing faulty code) that we can terminate our program. This inelegant control structure is the price we pay for the simplicity of using GLUT.

The keyboard callback function handles key presses. GLUT passes our keyboard function a "key code"; the ASCII code for the key that has just been pressed. The ASCII code for Escape is 27 (0x1b in hexadecimal). If Escape is pressed we call the aforementioned system exit(...) function.

The reshape callback function sets up the perspective transformation necessary to render our 3D graphics (the teapot) to our 2D display. Refer to Chapter 3 for the mathematics of perspective - and later in this Chapter for how that mathematics is realised by OpenGL.

The display function clears the screen and draws our teapot. First there are a couple of lines referring to matrices - OpenGL uses matrices exactly as we did in Chapter 3 to manipulate objects using rigid body transformations. In this case the two lines tell OpenGL that no such matrix transformations will occur, in this simple example. Then we have the command to draw a Teapot - OpenGL/GLUT may be the only library in the world to contain a function for drawing teapots. This function has limited utility in real world applications, but serves for demonstrations. The final two commands in display(...) tell OpenGL to commit to the screen any graphics drawn (i.e. the teapot). We discuss these command in more detail next.

4.2.1 Double Buffering and Flushing

It is possible to run your GLUT managed display in single or double buffered mode. In our opening lines of main(...) we specified that the display would run in double buffered mode (the **GLUT_DOUBLE** flag was passed to glutInit). In single buffered mode, anything we draw goes straight to the screen - i.e. is written to the frame buffer (recall Chapter 2) linked to the screen. In double buffered mode we still have a frame buffer for the screen, but also a "spare" buffer. We draw onto a "spare" buffer in our display(...) function, and then swap the screen and spare frame-buffers with a call to glutSwapBuffers(). We would usually prefer double-buffering when creating an animation in OpenGL. This is because drawing direct to the screen (i.e. using single-buffering) can cause flickering.

The call to glFlush() prior to the glutSwapBuffers() is for compatibility with slower graphics hardware. Slower hardware will queue up the various OpenGL library calls and execute them in batches. Usually we want to ensure all our queued OpenGL calls are executed before we swap frame-buffers (the calls to swap buffers are not queued). Therefore we 'flush' the queue for good practice prior to glutSwapBuffers().

4.2.2 Why doesn't it look 3D?

The teapot has been modelled as 3D object, but has been rendered with the OpenGL default scene lighting settings which give it a flat appearance. Later we will describe how to set up the OpenGL lighting properly. For now, we can write glEnable(**GL_LIGHTING**) before the glutSolidTeapot(..) line to improve aesthetics.

4.3 Modelling and Matrices in OpenGL

The 3D matrix transformation theory described in Chapter 3 can be put into practice using OpenGL, to create scaling, rotation, perspective projection effects and similar.

Recall from Chapter 3 that we create 3D graphics by first modelling objects in 3D space. These models comprise a series of vertices that may be linked together to form surfaces (see Chapter 6) and so form 3D objects. We can then manipulate our modelled objects by applying matrix transformations to the vertices in the model (for example applying translations $\underline{\underline{T}}$, or rotations $\underline{\underline{R}}$). The vertices move, and thus so do the surfaces anchored to them. Finally we apply a perspective transformation (for example $\underline{\underline{P}}$) to project our vertices from 3D to 2D, and draw the 2D points on the screen. For example, for some vertices $\underline{\underline{p}}$:

$$\underline{\underline{p'}} = \underline{\underline{PRTp}} \tag{4.1}$$

In OpenGL the projection matrix (i.e. $\underline{\underline{P}}$) is stored in a 'variable' called the PROJECTION matrix. The rigid body transformations $\underline{\underline{R}}$, $\underline{\underline{T}}$ (and any similar) are stored as a compound matrix transformation in a 'variable' called the MODELVIEW matrix.

We can write to these variables by first 'selecting' them using a glMatrixMode(**GL_PROJECTION**) or glMatrixMode(**GL_MODELVIEW**) call. We then use the various helper functions in OpenGL to post-multiply the existing contents of the variable with another matrix. For example, the call to gluPerspective(...) in Section 4.2 creates a perspective matrix $\underline{\underline{P}}$ and post-multiplies it with the current contents of the PROJECTION matrix. In that example we had used another helper function glLoadIdentity() to load the identity matrix ($\underline{\underline{I}}$) into the PROJECTION matrix. So the effect of the code in reshape(...) was to load $\underline{\underline{I}}$ into PROJECTION and then set projection to $\underline{\underline{IP}}$ where $\underline{\underline{P}}$ is a new perspective matrix configured with the various parameters passed to gluPerspective(...).

Similarly inside the display(...) we first overwrite the MODELVIEW matrix with the identity matrix $\underline{\underline{I}}$. We could then use other OpenGL helper functions to post-multiply $\underline{\underline{I}}$ with rotations, translations, or similar - to manipulate our model as we see fit. The most commonly used helper functions are:-

```
glTranslatef(x,y,z)   // Translation matrix with shift T_x=x, T_y=y, T_z=z
glRotatef(theta,x,y,z)// Rotation matrix theta degrees clockwise about
                      // axis pointing along vector (x,y,z)
glScalef(x,y,z)     // Scale matrix with scale factors S_x=x, S_y=y, S_z=z.
```

Note that unlike most C calls the rotation angle is specified in degrees rather than radians.

We could ignore the distinction between PROJECTION and MODELVIEW matrices, and just write our perspective, translations, scalings etc. all into a single variable e.g. MODELVIEW and leave the other matrix as the identity. However this is bad practice as typically we would want to set the PROJECTION matrix up once e.g. during window reshape(...), and then manipulate the MODELVIEW several times during display(..).

4.3.1 The Matrix Stack

At any time we can call glPushMatrix() to save the contents of the currently selected matrix variable. The contents can later be restored with a call to glPopMatrix(). As the name implies, the variables are saved onto a LIFO (stack) structure. This is very useful when

drawing articulated bodies which are often represented as a hierarchy (tree) of reference frames, and traversed recursively (recall Chapter 3).

4.4 A Simple Animation - Spinning Teapot

We can create animations in OpenGL by introducing a further callback - the idle(...) callback. This function is called repeatedly by GLUT when the system is idling (doing nothing). We can use this opportunity to increment a counter and redraw our graphics. For example, we might maintain a counter that increments from 0 to 359 - indicating the number of degrees to rotate a Teapot. The code is near identical to our previous example:

```
#include "GL/gl.h"
#include "GL/glut.h"

void keyboard(unsigned char key, int x, int y);
void reshape(int w, int h);
void display(void);
void idle(void);

void init(void);

static int gRotAngle = 0; // global variable for rotation

int main (int argc, char** argv) {

   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (600, 600);
   glutCreateWindow ("Animated Teapot");

   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutIdleFunc (idle);

   glutMainLoop();
}

void keyboard(unsigned char key, int x, int y) {

   switch (key) {
  case 0x1b:
  exit (0);
  break;
   }
```

```
}

void idle()  {

   gRotAngle = (gRotAngle + 1) % 360;
   glutPostRedisplay(); // trigger callback to display(..)

}

void reshape(int w, int h) {

   glViewport(0, 0, w, h);


   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60,(GLfloat)w/(GLfloat)h,2,40.0);
}



void display(void) {

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // glTranslatef(0,0,-10);  // example translation
glRotate(gRotAngle,0,0,1);

  glutSolidTeapot(1.0);

  glFlush();
  glutSwapBuffers();
}
```

Key features of this code are:-

- The use of a global variable to store the counter. This is messy but necessary as GLUT doesn't let us pass state around as parameters to its callbacks.
- The use of the idle callback to increment the counter.
- The call to glutPostRedisplay() within the idle callback, which triggers a display event

inside GLUT - and thus triggers a callback to display(...) - which draws the teapot at a new angle.

In this case the counter is simply used as an input to glRotate to create a spinning teapot, but more complex examples could be imagined. Note that the teapot will spin at the maximum speed possible given the graphics hardware. This will vary greatly from machine to machine. We could use operating system specific calls to govern the animation's speed. For example, under Windows, we could insert a Sleep(milliseconds) system call. Or we could repeatedly read the system clock and loop (block) until a constant number of milliseconds has elapsed. Again, an OS-specific system call is used to access the system clock.

4.5 Powerpoint

The remainder of the OpenGL content on this course was delivered via Powerpoint and you should refer to the sets of Powerpoint handouts on Moodle.

Chapter 5

Eigenvalue Decomposition and its Applications in Computer Graphics

5.1 Introduction

Linear algebra is essential to Computer Graphics. We have already seen (Chapter 3) that matrix transformations can be applied to points to perform a wide range of essential graphics operations, such as scaling, rotation, perspective projection etc. In this Chapter we will explore a further aspect of linear algebra; **eigenvalue decomposition (EVD)** and the many useful applications it has in both Computer Graphics and Pattern Recognition.

5.1.1 What is EVD?

EVD is a method for "factorising" (in linear algebra terminology we say **decomposing**) a square matrix $\underline{\underline{M}}$ into matrices $\underline{\underline{U}}$ and $\underline{\underline{V}}$ such that:

$$\underline{\underline{M}} = \underline{\underline{U V U}}^T \tag{5.1}$$

i.e. the EVD process outputs $\underline{\underline{U}}$ and $\underline{\underline{V}}$ for a given input matrix $\underline{\underline{M}}$. If $\underline{\underline{M}}$ is an $n \times n$ matrix, then $\underline{\underline{U}}$ and $\underline{\underline{V}}$ will also be $n \times n$ matrices. They have the form:

$$\begin{aligned} \underline{\underline{U}} &= \left[\underline{u_1}\ \underline{u_2}\ ...\ \underline{u_n}\right] \\ \underline{\underline{V}} &= \begin{bmatrix} \lambda_1 & 0 & ... & 0 \\ 0 & \lambda_2 & ... & 0 \\ & & ... & \\ 0 & 0 & ... & \lambda_n \end{bmatrix} \end{aligned} \tag{5.2}$$

i.e. $\underline{\underline{V}}$ is a **diagonalised matrix**. The n-dimensional vectors $\underline{u_i}$ are called the **eigenvectors** and the scalar values λ_i are called the **eigenvalues** (where $i = 1...n$). Thus for every eigenvector there is an associated eigenvalue — and there are n pairs of eigenvectors/values for an $n \times n$ matrix $\underline{\underline{M}}$ (e.g. there are two pairs of eigenvectors/values for a 2×2 matrix).

Eigenvectors and eigenvalues also satisfy this interesting property (in fact this property is often used to formally define what an eigenvector/value is):

$$\underline{\underline{M}}\underline{u_i} = \lambda_i \underline{u_i} \tag{5.3}$$

To understand this, consider $\underline{\underline{M}}$ as a matrix transformation acting on a point $\underline{p}$, (i.e. $\underline{\underline{M}}\underline{p}$). Points are simply vectors from the origin in a reference frame (i.e. $(3,2)^T$ means 3 units along, 2 units up from the origin). Eigenvectors ($\underline{u_i}$) are specially chosen vectors such that when $\underline{\underline{M}}$ acts upon them (i.e. $\underline{\underline{M}}\underline{u_i}$), the effect is identical to simply scaling vector $\underline{u_i}$ by value λ_i (i.e. $\lambda_i \underline{u_i}$).

'Eigen' is the German stem word meaning "same" or "characteristic". An eigenvector is a vector $\underline{u_i}$ that, when scaled by its corresponding eigenvalue λ_i, gives the *same* result as multiplying that eigenvector by $\underline{\underline{M}}$.

5.2 How to compute EVD of a matrix

Before we discuss why this is useful — i.e. the applications of EVD, we will briefly describe how to perform EVD. Most scientific computing packages have an EVD function built-in e.g. in Matlab:

```
[U V]=eig(M)
```

performs EVD on matrix $\underline{\underline{M}}$ to yield eigenvectors $\underline{\underline{U}}$ and eigenvalues $\underline{\underline{V}}$ as defined in eq.(5.2). However there is value in knowing how to perform the mathematics manually.

Given a matrix $\underline{\underline{M}}$ we first compute the eigenvalues λ_i by solving an equation known as the **characteristic polynomial** (subsection 5.2.1). Once we have the eigenvalues we can then solve a linear system to get the eigenvectors (subsection 5.2.2).

5.2.1 Characteristic Polynomial: Solving for the eigenvalues

Recall that, for an $n \times n$ matrix $\underline{\underline{M}}$, there are n eigenvector/value pairs. Any eigenvector $\underline{u}$ and eigenvalue λ of $\underline{\underline{M}}$ will satisfy:

$$\underline{\underline{M}}\underline{u} = \lambda \underline{u} \tag{5.4}$$

Re-arranging this relationship we get:

$$\underline{\underline{M}}\underline{u} - \lambda \underline{u} = 0 \tag{5.5}$$

And factoring out $\underline{u}$ we get:

$$(\underline{\underline{M}} - \lambda \underline{\underline{I}})\underline{u} = 0 \tag{5.6}$$

where $\underline{\underline{I}}$ is the $n \times n$ identity matrix. There are solutions for this equation only when the determinant of $\underline{\underline{M}} - \lambda \underline{\underline{I}}$ is zero. Writing the determinant as $|...|$ we obtain solutions for λ by solving:

$$|\underline{\underline{M}} - \lambda \underline{\underline{I}}| = 0 \tag{5.7}$$

If we expand out the determinant of the above equation we will obtain a polynomial in λ of order n; i.e. with n roots. This is called the **characteristic polynomial**. The roots of the characteristic polynomial are our eigenvalues. It follows that not all of the roots may be real. **In fact we are guaranteed real roots (i.e. real eigenvalues) only when $\underline{\underline{M}}$ is a *symmetric matrix*).**

An example: Eigenvalues of a 2×2 matrix

We will illustrate the process of solving for eigenvalues using a simple 2×2 example matrix:

$$\underline{\underline{M}} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{5.8}$$

Substituting $\underline{\underline{M}}$ into eq.(5.7) we need to solve:

$$\begin{aligned} \left| \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \right| &= 0 \\ \left| \begin{bmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{bmatrix} \right| &= 0 \end{aligned} \tag{5.9}$$

Recall that the determinant of a 2×2 matrix is defined as:

$$\left| \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right| = ad - bc \tag{5.10}$$

So writing out the determinant of eq.(5.9) we get:

$$\begin{aligned} (2-\lambda)^2 - 1 &= 0 \\ \lambda^2 - 4\lambda + 3 &= 0 \end{aligned} \tag{5.11}$$

Solving this quadratic is straightforward, taking $a = 1, b = -4, c = 3$ we have:

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.12}$$

So the two roots of the characteristic polynomial (the eigenvalues) are $\lambda = 3$ and $\lambda = 1$.

5.2.2 Solving for the eigenvectors

Once we know the eigenvalues λ we need to find corresponding eigenvectors $\underline{u}$. We do this by substituting each eigenvalue solution λ into:

$$(\underline{\underline{M}} - \lambda \underline{\underline{I}})\underline{u} = 0 \tag{5.13}$$

and solving the resulting linear system to find $\underline{u}$. So, continuing our previous 2×2 example:

$$\underline{\underline{M}} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{5.14}$$

with eigenvalues $\lambda_1 = 3$ and $\lambda_2 = 1$ we must find $\underline{u_1}$ and $\underline{u_2}$. Let's look at $\underline{u_1}$ first:

$$\begin{aligned} (\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix})\underline{u_1} &= 0 \\ \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \underline{u_1} &= 0 \end{aligned} \tag{5.15}$$

Writing $\underline{u_1} = [x_1 \; y_1]^T$ we have simultaneous equations:

$$\begin{aligned} -x_1 + y_1 &= 0 \\ x_1 - y_1 &= 0 \end{aligned} \tag{5.16}$$

i.e. $x_1 = y_1$. This illustrates an ambiguity with eigenvectors; there are many possible vector solutions for a particular eigenvector — all pointing in the same direction but with different magnitudes. i.e. we can only solve up to a scale factor using these equations. But to satisfy our original definition of eigenvectors/values:

$$\underline{\underline{M}}\underline{u} = \lambda \underline{u} \tag{5.17}$$

the eigenvectors must be **unit length**, so in this case a solution is $\underline{u_1} = [0.707, 0.707]^T$ although $\underline{u_1} = [-0.707, -0.707]^T$ is also perfectly valid for the first eigenvector.

The second eigenvector equation yields:

$$\begin{aligned} x_2 + y_2 &= 0 \\ x_2 + y_2 &= 0 \end{aligned} \tag{5.18}$$

So $x_2 = -y_2$ and a unit length vector satisfying this is $\underline{u_2} = [0.707, -0.707]^T$. For EVD of larger matrices it is sensible to use some alternative technique to solve the linear system and thus find the eigenvectors; for example Gauss-Jordan elimination (covered in CM10197).

Orthonormal nature of eigenvectors

We will not prove it here, but eigenvectors are not only unit length but are also **mutually orthogonal** to one another. Thus the matrix $\underline{\underline{U}}$ formed of eigenvectors (subsection 5.1.1) is an **orthonormal matrix**. This is a useful property to remember when we come to invert $\underline{\underline{U}}$ later on in the Chapter (since the inverse of an orthonormal matrix is simply its transpose).

5.3 How is EVD useful?

There are two main uses for EVD in Computer Graphics, and we will discuss these in the remainder of this Chapter.

5.3.1 EVD Application 1: Matrix Diagonalisation

Consider an origin centred polygon $\underline{\underline{p}} = [\underline{p_1}\ \underline{p_2}\ ...\ \underline{p_n}]$ acted upon by a transformation $\underline{\underline{M}}$:

$$\underline{\underline{M}} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} f & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{5.19}$$

The resulting polygon $\underline{\underline{M}}\underline{\underline{p}}$ will be stretched by a factor f along an axis passing through the origin, with direction $[\cos\theta, \sin\theta]^T$. Figure 5.1 illustrates.

Suppose we did not know the matrix chain initially used to create compound matrix $\underline{\underline{M}}$. By inspecting the elements of compound matrix $\underline{\underline{M}}$, it is not obvious what its operation is; we would have to apply the matrix to a polygon to observe its effect. However, we could use EVD to decompose $\underline{\underline{M}}$ into an orthonormal matrix $\underline{\underline{U}}$ and a diagonal matrix $\underline{\underline{V}}$ (sometimes this process is also referred to as **matrix diagonalisation**):

$$\underline{\underline{M}} = \underline{\underline{UVU^T}} \tag{5.20}$$

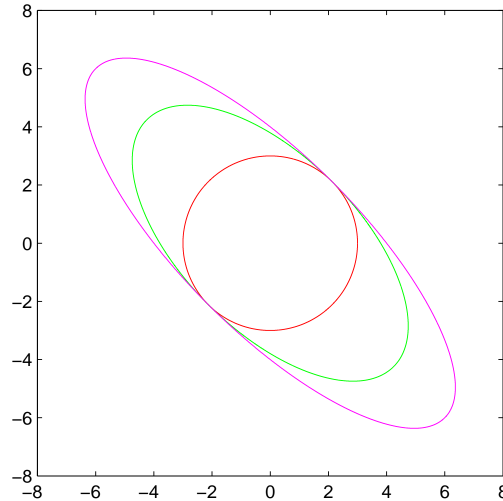


Figure 5.1: Polygon approximating a circle (red), stretched using the compound matrix transformation ($f = 2$) outlined in Section 5.3.1. The green circle using the transformation raised to power 1, the magenta circle using the transformation raise to power 1.5 using the diagonalisation technique to raise matrices to an arbitrary power.

Because the rotation matrix is orthonormal, and the scaling matrix is diagonal, we have recovered the rotation matrix in $\underline{\underline{U}}$ and the scaling in $\underline{\underline{V}}$.

Further uses for this operation are to raise a matrix to a power. For example, if we wanted to produce double the effect of transform $\underline{\underline{M}}$ we could multiply the points $\underline{p}$ by $\underline{\underline{M}}$ *twice* i.e. $\underline{\underline{M}}\,\underline{\underline{M}}\,\underline{p}$, or $\underline{\underline{M}}^2$. It is initially unclear how one might raise $\underline{\underline{M}}$ to a non-integer or even a negative power. However eq.(5.20) provides a solution; we can raise the diagonal elements of $\underline{\underline{V}}$ to any power we choose and recombine $\underline{\underline{U}}\,\underline{\underline{V}}\,\underline{\underline{U}}^T$ to create a compound matrix $\underline{\underline{M}}$ raised to the required power. Note we could also raise $\underline{\underline{M}}$ to power -1 in this way to create the inverse transformation, or we can even define the concept of a **matrix square root**.

Note that EVD matrix diagonalisation will only work for symmetric, square matrices; thus this technique is restricted to only particular forms of compound matrix transformation. For more general diagonalisations we can use SVD (Section 5.5.3).

5.3.2 EVD Application 2: Principle Component Analysis (PCA)

Consider a collection of n points $\underline{\underline{p}} = [\underline{p_1}\ \underline{p_2}\ ...\ \underline{p_n}]$; the space might be 1D, 2D, 3D or even higher dimensional. We can perform a procedure "**principal component analysis (PCA)**" to learn about the distribution of those points within the space. More specifically, PCA gives us a reference frame embedded within the space known as a "data dependent reference frame" or Eigenmodel. This reference frame is useful for a large number of applications, as we will see later in this Chapter. EVD is one of the key steps in computing PCA.

Rather than introduce the mathematics of PCA directly, we introduce it within the context

of **pattern recognition** (Section 5.4); a very common application of PCA. However, as we will see later, PCA can be used for data visualization and collision detection too, to name but a few alternative applications.

5.4 An Introduction to Pattern Recognition

Pattern Recognition underpins much of Computer Vision; a complementary field to Computer Graphics that deals with images as input (rather than output) data, and make decisions based on that data. It is within that decision making process that Pattern Recognition techniques are applied. As an example, consider the London Congestion Charging system. This is a Computer Vision system, consisting of numerous cameras, that can read vehicle licence plates and bill commuters for driving into Central London. Here, Pattern Recognition algorithms are applied to extract a alphanumeric string (the vehicle licence plate) from pixels in an image taken by one such camera.

Many Pattern Recognition problems are also **classification problems**. Given some data, we 'classify' that data by making a decision on whether the data fits into **category A**, **B**, **C**, etc. or perhaps does not fit well into any category (the **null category**). For example, given an image of a character on a vehicle number plate, we could classify that image as being a letter 'A', 'B', 'C', or not a letter at all. We usually would also like to quantify the confidence we have in our classification.

Many classification problems rely on the system designer **training** the classifier with example data for each category. These data are called **exemplar data** or the **training set**. In the above example, the system might be shown a training set containing several examples of each letter to learn an 'understanding' (we call this 'learning a **model**') of each category. These problems are called **supervised classification** problems. By contrast, unsupervised classification problems take a set of data and automatically learn a set of categories for classification without training. We will only look at supervised classification problems here. On this course we have time only to introduce the main concepts of classification (feature spaces and distance metrics), and to discuss two simple classification algorithms (nearest mean classifiers and Eigenmodel based classifiers). The latter example is where we find our application of PCA.

5.4.1 A Simple Colour based Classifier

Suppose we have an image, such as Figure 5.2 and wish to identify the sky region in the image. This is a classification problem; we wish to classify each pixel in the image as being in the 'sky' category, or not. There are many ways we could approach this problem. Here we will assume that sky pixels are of a distinctive colour, and that we can make a decision independently for each pixel based solely on its colour. For this example, we will deal with a greyscale image — so for now, by colour we mean pixel intensity.

We could create a **training set** of sky pixels by manually picking out regions of sky from the image. We could then take a mean average of those pixels to create a **model** of the sky category. In our case this would be a scalar value representing the average intensity (I_{sky}) of pixels in our training set (sky pixels).

Figure 5.2: Illustrating a simple intensity classification approach to detecting sky. Left: Source greyscale image; pixels used to train model marked in green (mean intensity was 175). Middle: Classification at $T = 1$ (white is sky, black is non-sky). Right: Poorer classification at $T = 10$; more sky pixels have been categorised as non-sky. In both cases pixels on the camera and man have been marked as sky. Clearly intensity is not an ideal discriminating feature for detecting sky.

We could then test each pixel in the image, and measure how similar that test pixel's intensity (I_{test}) is to our model (I_{sky}). A simple way to do this would be to take the absolute value of their difference:

$$d(I_{test}, I_{sky}) = |I_{test} - I_{sky}| \tag{5.21}$$

We call $d(.)$ a **distance metric**; we use it to measure how far from a given category model our test sample is.

We could state that all pixels with $d(.)$ above a certain value T are too different from our model to be in the sky category, otherwise they are similar enough to be sky. We call T a **threshold**. Figure 5.2 shows the result of defining different threshold levels (assuming pixel intensities range between 0 and 255). If we set T too low, then we incorrectly mark sky pixels as non-sky. If we set T too high, then we incorrectly mark non-sky pixels as sky.

Caveats to this approach

We have produced a basic classifier capable of discriminating sky pixels from non-sky, although not with great accuracy. In this example much of the data (pixels) we tested our classifier on were same data (pixels) used to train the model. Usually this is inadvisable, because:

1. we have already manually classified the training data (i.e. marked where the sky is). There is little point in running a classifier over data we already know the classification of!

2. running your classifier over its training data usually results in a better classification performance than running it over new 'unseen' **test data**. You therefore gain a false impression of the capabilities of your classifier.

A better impression of the classifier's performance would be gained by training it on examples of sky from several training images, and then testing it on a set of images distinct from that

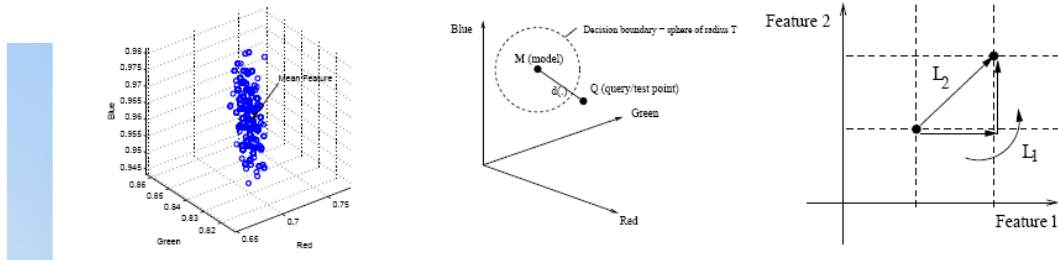


Figure 5.3: Left: A 3D feature space used to classify pixels on their colour; here an enlarged portion of that space is shown, each pixel in a photograph of sky (above) is plotted in the feature space. Middle: Euclidean distance metric between two points; an 'ideal' (averaged) feature for sky (the model), and a pixel being tested for membership of the sky category. Right: Diagrams of the L_2 norm (Euclidean distance), and the L_1 norm ("City Block" or Manhattan distance). Here shown in a 2D feature space.

training set. We will not discuss methodologies for evaluating classifier performance on this course. You may like to search Google for the term "Precision Recall" to get a feeling for common approaches to this problem.

5.4.2 Feature Spaces

We now consider an RGB colour version of Figure 5.2 (left) and how we might go about performing the same classification on a colour image. We wish to classify pixel as being sky or non-sky on the basis of its RGB colour, which is now vector valued, i.e. $[r, g, b]^T$ rather than a scalar.

In Pattern Recognition terms we say that the colour of the pixel is the **feature** that we are measuring, in order to make a classification decision (i.e. whether the pixel is sky or non-sky). Features are not always measurements of colour – they can be anything measurable about the classification problem we are trying to solve that give us enough information to discriminate between categories (see Section 5.4.4 for more examples). However in this case the feature is a vector with 3 elements, representing Red, Green and Blue. In the previous example (Section 5.4.1) the feature was a vector with 1 element (i.e. a scalar) representing pixel intensity.

We can imagine the features forming a distribution of points in a n-dimensional space, each point representing a different pixel (i.e. feature measurement). We call this the **feature space**. In our case the feature space is $3-$dimensional, in fact it is an RGB colour cube identical to that discussed in Chapter 2. Figure 5.3 (left) gives an example of an image, and corresponding features in our feature space. Pixels of similar colour will occupy similar regions of the space.

We train our system as before, by manually identifying sky pixels to form a training set. Features from the training pixels form a distribution in the feature space. We produce a model for sky by computing an average feature (vector) from all points in this distribution (Figure 5.3, left).

Now, given a test pixel we wish to use this model to decide whether that pixel is sky or not. We first compute our feature vector from the test pixel, which we write $\underline{Q} = [Q_r, Q_g, Q_b]^T$. We then measure the proximity of $\underline{Q}$ to the model, using a **distance metric**. Writing the model as $\underline{M} = [M_r, M_g, M_b]^T$, the distance $d(\underline{Q}, \underline{M})$ is simply the magnitude of the vector between those two points in the feature space.

$$\begin{aligned} d(\underline{Q}, \underline{M}) &= \left|\underline{Q} - \underline{M}\right| \\ &= \sqrt{(Q_r - M_r)^2 + (Q_g - M_g)^2 + (Q_b - M_b)^2} \end{aligned} \quad (5.22)$$

We are then able to threshold distance $d(.)$ to decide whether the test pixel is sky or not. We can imagine a sphere in the feature space, centered at $\underline{M}$, of radius T, within which lie all features corresponding to sky pixels (Figure 5.3, middle). Features outside of the sphere are those arising from non-sky pixels. The perimeter of this sphere is said to describe the **decision boundary** of the classifier.

5.4.3 Distance Metrics

We have seen that classification of sky in intensity and colour images was achieved in a similar manner. We just need to choose (1) an appropriate feature space (we might have chosen any of the colour models from Chapter 2), and (2) an appropriate distance metric.

The distance metric used in eq.(5.22) is called the **Euclidean distance**. It is sometimes called an L_2 (or order 2) **norm**. An order α norm is defined as:

$$L_\alpha = \left(\sum_{i=1}^{n} |p_i - q_i|^\alpha \right)^{\frac{1}{\alpha}} \quad (5.23)$$

where p_i and q_i are the ith elements of two n-dimensional element vectors being compared. Sometimes you will encounter the L_1 norm – sometimes called the City Block or **Manhattan distance**. This is the sum of the absolute difference between corresponding elements of the vector. Given 2D points $(0,0)^T$ and $(1,1)^T$, the L_2 norm is $(\sqrt{2})$ but the L_1 norm is 2 (see Figure 5.3, right). The distance used in our first example eq.(5.21) was written as an L_1 norm – although in 1D the L_1 and L_2 norms are equivalent.

There are many other definitions of distance metric. *Choosing an appropriate* **feature space***, and an appropriate* **distance metric** *are the key issues to address when solving a classification problem.*

5.4.4 Nearest-Mean Classification

So far we have explored single category classification problems. We trained up a model of a category (e.g. sky) and computed the similarity of our test data to that model. We created a decision boundary to decide that, within a certain level of similarity, test data is in the category (sky) otherwise it is in the null category (non-sky).

In this subsection we will explore the more interesting problem of multiple categories. Our opening example (Section 5.4) of recognising the characters on a vehicle licence plate was a multiple category problem. The simplest approach is to model each of your categories as

points in a feature space. Then, given a datum to classify, you work out its point in the feature space and see which category point it is nearest to. This is called **Nearest Mean classification**. We will look at two examples: a movie recommender system, and a weather classification system.

Movie Recommender System

Consider an online DVD movie store with a facility to recommend movies to customers. Let's assume that when customers buy a movie, they rate it on a scale from 0 (bad) to 1 (good). Consider a customer, **X**. Based on the ratings collected from the store's customers, we would like to recommend a movie to **X**. One solution is to look for customers who have bought the same movies as **X**. If we compare their ratings with **X's** we should be able to find the customer with the closest ratings to **X**. Then perhaps some of the other movies that 'closest' customer purchased, and liked, will be suitable recommendations for **X**.

Let's suppose **X** has purchased "The Terminator" and "The Sound of Music", giving them scores of 0.8 and 0.2 respectively. We query the customer database of our store, obtaining a list of customers who also bought "The Terminator" and "The Sound of Music". We find that 4 other users **A-D** with those films in common, and plot their ratings in a 2D **feature space** with the movie ratings as the axes (Figure 5.4, left). Each user becomes a point in the feature space; their ratings for the movies are the feature vectors.

If we measure the Euclidean distance (i.e. compute a distance metric) between **X** and all other users, we obtain a measure of similarity in movies tastes between **X** and those users. We can see that **C** has the most similar tastes to **X**, having near identical ratings of "The Terminator" and "The Sound of Music". We can now look at **C**'s purchasing history and recommend items in it to **X**.

This is an example of a multiple category classifier. We have used the feature space to work out which of the categories (in this case users **A-D**) a query point (**X**) is nearest to. No distance threshold is used this time because we are interested in choosing the category nearest to **X**. There is no "null category" because there will always be a "nearest" category to **X**. We might want to modify the problem so that users greater than a threshold distance T from **X** have tastes that are too dissimilar, and so no recommendations will be made. This effectively introduces a "null category"; we say that **X** has similar tastes to one of **A, B, C, D**, or none of the above.

In our description, we used a 2D feature space because **X** had 2 films in common with users **A-D**. However in general the space would be n-dimensional if the users had n films in common. Feature spaces often have more than 2 or 3 dimensions. Figure 5.4 (left) illustrates this, although of course we cannot easily visualize spaces with > 3 dimensions.

Weather Classification System

Suppose we are given some measurements of ice-cream sales, and humidity, taken over 100 days. 50 of those days were sunny, and 50 were rainy. We would like to train a classifier over that data, so that if we are given data for ice-cream sales and humidity for a previously unseen day, we can determine if it is a sunny or rainy day. This is an example of a two

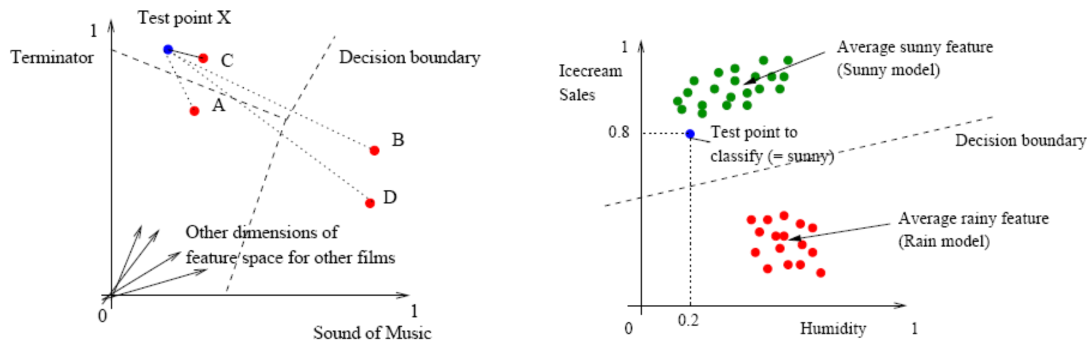


Figure 5.4: Two examples of multi-category classification problems. Left: We wish to categorise X's tastes as being closest to either A, B, C or D. The feature space represents movie tastes, each axis a rating for a movie that users have in common. The decision boundary here is quite complex, resembling a Voronoi diagram. Right: A further example in which we wish to categorise a day as being 'sunny' or 'rainy'. The point (0.8, 0.2) is closer to the sunny model, and so the day is classified as sunny. This type of classifier (two categories, no null category) is called a **dichotomiser**.

category classification problem. We need to train a classifier to tell us if a given sample is in one category (sunny) or another (rainy). There is no null category.

First we identify our feature space. The features we are measuring are level of ice-cream sales and humidity, so we have a 2D feature space with those measurements as the axes. We can then plot our training data sets (the 50 sunny, and 50 rainy days) on the axes (Figure 5.4, right).

Now we build our model. We need one model per category. As before we will just take an average to obtain points in the space that describe 'sunny' or 'rainy' days. We *average the sunny and rainy samples independently.* This results in two points, modelling the sunny and rainy day training data respectively.

Now, given some data (a feature, e.g. ice cream sales of 0.8, humidity of 0.2) of a previously unseen day, we measure the distance from that feature to each of the models. We could use any distance metric for this but we will use Euclidean Distance. We see that the "sunny" model is the closest. As before, we have the concept of a decision boundary based on distance from the models (Figure 5.4, right). We can therefore state that the day was sunny. We could even take a ratio of the distances between the point and the sunny and rainy model to give us an estimate of confidence in our decision.

This approach extends trivially to any number of categories (e.g. add in a snow day category). We just compute more models for more independent training sets, and include them in our distance comparison. The decision boundary becomes more complex; in this case it becomes a Voronoi diagram with regions seeded at each model's point in the feature space.

5.4.5 Eigenmodel Classification

So far all of our classification examples have used a mean average to compute the model over a training set. This approach is unsatisfactory in general, because the mean alone captures no information about the distribution of the training data in the feature space.

In Figure 5.4 (right), the sunny day training data formed an elongated distribution in the space – shaped rather like a Rugby ball. As we are working in a 2D feature space, the locus of features equidistant from the distribution's mean can be described by a circle. It is trivial to construct a circle where some points on the circumference fall significantly outside the distribution, yet others fall inside the distribution. This tells us that the combination of the model as a mean average, and the distance metric as "Euclidean distance from the mean" is insufficient to determine membership of a category when the training set is not **uniformly distributed**.

Most real data is not uniformly distributed. Thus we would like to use a more sophisticated model and distance metric that takes into account not only the mean, but also shape of the training set's distribution about the mean. One way to do this is to construct an **Eigenmodel** as our model, and use a new distance metric – the **Mahalanobis distance** – which measures the distance of a feature from an Eigenmodel.

Constructing an Eigenmodel from a Training Set

An Eigenmodel is derived from the mean and **covariance** of a point distribution (i.e. features in a training set). You will be familiar with the concepts of mean and variance, as defined for 1D features (scalars). The mean generalises easily to n-dimensional vectors; you simply take the mean average across each dimension of the vector independently. Covariance is the generalisation of variance to n-dimensional vectors; it is represented as an $n \times n$ symmetric matrix. Each element $c_{i,j}$ describes the variance of values in the ith dimension with respect to the jth dimension of the vector space.

Writing our training set as a collection of n features $\underline{\underline{p}} = [\underline{p_1}, \underline{p_2}, \underline{p_3}, ..., \underline{p_n}]$ within an d-dimensional feature space, we compute the mean $\underline{\mu}$ as:

$$\underline{\mu} = \frac{1}{n} \sum_{i=1}^{n} \underline{p_i} \tag{5.24}$$

We are then able to compute the covariance as:

$$\underline{\underline{C}} = \frac{1}{n} (\underline{\underline{p}} - \underline{\mu})(\underline{\underline{p}} - \underline{\mu})^T \tag{5.25}$$

An Eigenmodel consists of the mean $\underline{\mu}$, and a eigenvalue decomposition (EVD) of the covariance matrix $\underline{\underline{C}}$ into two matrices containing the eigenvectors ($\underline{\underline{U}}$) and eigenvalues $\underline{\underline{V}}$ of $\underline{\underline{C}}$, satisfying:

$$\underline{\underline{C}} = \underline{\underline{UVU}}^T \tag{5.26}$$

The process of forming the Eigenmodel from the set of points $\underline{\underline{p}}$ is called 'Principal Component Analysis' (PCA). We will return to PCA in more detail in Section 5.3.2.

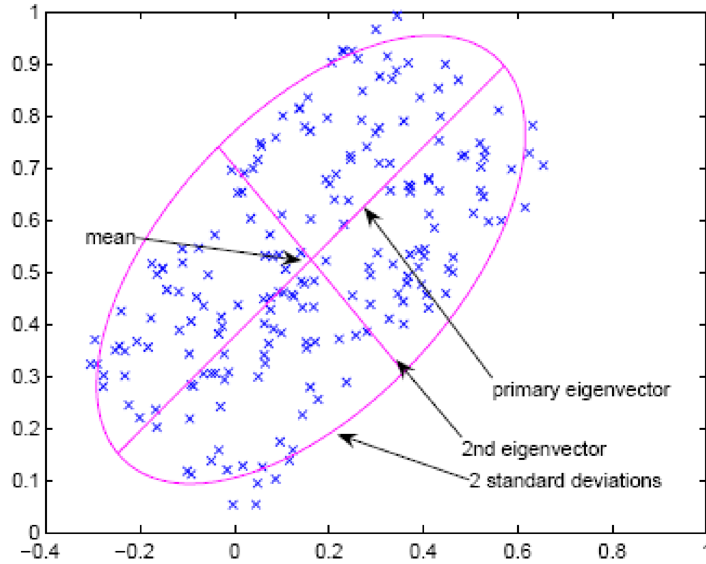


Figure 5.5: A 2D distribution of 100 points, and the corresponding Eigenmodel. The mean and eigenvectors are indicated. Note how the principal eigenvector lies across the direction of greatest variance. A contour has been drawn at a constant 2 standard deviations from the mean.

Interpreting the Eigenmodel

The three components of the Eigenmodel $\{\underline{\mu}, \underline{\underline{U}}, \underline{\underline{V}}\}$ define a reference frame with respect to the training data; Figure 5.5 illustrates.

The mean $\underline{\mu}$ describes the origin of the reference frame.

The basis vectors are described by the columns in the $d \times d$ matrix ($\underline{\underline{U}}$); each column is an eigenvector specifying a direction through the d-dimensional feature space. Recall from Section 5.1.1 that there are as many eigenvectors as there are dimensions in the space. So the 2D example of Figure 5.5 ($d = 2$) we have $\underline{\underline{U}} = [\underline{u_1}\ \underline{u_2}]$. $\underline{\underline{U}}$ is orthonormal; that is, the eigenvectors are mutually orthogonal and have unit length (magnitude).

$\underline{\underline{V}}$ is a diagonal matrix containing the eigenvalues; each eigenvalue is associated with an eigenvector:

$$\underline{\underline{V}} = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix} \tag{5.27}$$

Here, i is the ith eigenvalue, and is associated with the ith eigenvector $\underline{u_i}$. Most eigendecomposition routines sort $\underline{\underline{U}}$ and $\underline{\underline{V}}$ such that $\sigma_1^2 \geq \sigma_2^2 \geq ...\sigma_d^2$.

The eigenvalues represent the **variance** (σ^2) of data along the corresponding eigenvectors. So the first eigenvector ($\underline{u_1}$) is the direction along which there is the most variation (we call this the **principal eigenvector**), the second eigenvector has the second greatest variation,

and so on. Figure 5.5 illustrates.

Thus we see that the Eigenmodel not only represents the average of the training data, but also the principal directions in which it varies and how much variation there is along each of those directions. This is far more descriptive than the mean alone.

Classifying with the Mahalanobis Distance

The **Mahalanobis distance** is another distance metric for use in classification. Given a point $\underline{x}$ in the feature space, the Mahalanobis distance computes the distance between x and a model, in units of **standard deviation from the model**. Figure 5.5 illustrates why standard deviation is a better unit of distance than Euclidean distance for non-uniform distributions. It overcomes the problem discussed at the start of this subsection.

We compute Mahalanobis distance using the 3 components of an Eigenmodel fitted to the training data; i.e. we must have first modelled our training set using an Eigenmodel (obtained via PCA) in order to use the Mahalanobis distance.

To compute the Mahalanobis distance between x and an Eigenmodel $\{\underline{\mu}, \underline{\underline{U}}, \underline{\underline{V}}\}$ we must represent x in the reference frame defined by that Eigenmodel. We do this by subtracting the mean from x to align the origin of the space, and then multiply by the inverse of $\underline{\underline{U}}$ to get the coordinates of x in terms of the basis set $\underline{\underline{U}}$. You may find it useful to refer back to Chapter 3 and our discussion of basis sets to see why this is so.

$$\underline{x'} = \underline{\underline{U}}^T(\underline{x} - \underline{\mu}) \tag{5.28}$$

Note we have written $\underline{\underline{U}}^T$ rather than $\underline{\underline{U}}^{-1}$; the two are equivalent because $\underline{\underline{U}}$ is orthonormal.

Point $\underline{x'}$ is now defined in terms of distance along each of the eigenvectors. Computing $|\underline{x'}|$ at this point would simply give us the Euclidean distance from the mean. Instead we apply a scaling transformation, to warp the space such that the effective distance along each eigenvector is reduced in proportion to the variation of the data set along that eigenvector. Recall that this variation is encoded by the eigenvalues, and since these are already conveniently in a diagonalised (i.e. scaling) matrix $\underline{\underline{V}}$, we can multiply by $\underline{\underline{V}}^{-1}$. If you are unsure why this scales the space refer back to Chapter 3 and our discussion of geometric transformations and the scaling matrix.

$$\underline{x''} = \underline{\underline{V}}^{-1}\underline{\underline{U}}^T(\underline{x} - \underline{\mu}) \tag{5.29}$$

Now $|\underline{x''}|$ is the distance of $\underline{x}$ from $\underline{\mu}$ in units of variance. Thus the Mahalanobis distance $M(.)$ is $|\underline{x''}|$. We often write this in the convenient form:

$$M(\underline{x}; \{\underline{\mu}, \underline{\underline{U}}, \underline{\underline{V}}\})^2 = (\underline{x} - \underline{\mu})^T \underline{\underline{UV}}^{-1}\underline{\underline{U}}^T(\underline{x} - \underline{\mu}) \tag{5.30}$$

In some texts you will see this written as:

$$M(\underline{x}; \{\underline{\mu}, \underline{\underline{U}}, \underline{\underline{V}}\})^2 = (\underline{x} - \underline{\mu})^T \underline{\underline{C}}^{-1}(\underline{x} - \underline{\mu}) \tag{5.31}$$

Because the inverse of the covariance matrix $\underline{\underline{C}}$ is:

$$\underline{\underline{C}}^{-1} = \underline{\underline{UV}}^{-1}\underline{\underline{U}}^T \tag{5.32}$$

Refer back to subsection 5.3.1 to see why this is the inverse (arising from the diagonalisation of $\underline{\underline{C}}^{-1}$). In fact if we use this formulation of $M(.)$ we do not need to perform PCA to compute the Mahalanobis distance at all; we need only the mean and (inverse of) the covariance of the point distribution.

Use of Eigenmodels in Classification

When classifying, instead of computing Euclidean (absolute) distance from the mean of a training set, we can compute Mahalanobis (number of standard deviations) from the mean of a training set. The Mahalanobis distance takes into account the shape of the training set distribution, and so generally gives more accurate results. See, for example Figure 5.5which plots a contour at a constant number of standard deviations (2) from the mean. Computing the Mahalanobis distance requires that we model our training data using an Eigenmodel instead of just a mean. The rest of the classification process remains as previously discussed; we have simply redefined our distance metric and the way in which we model the classification categories.

5.5 Principle Component Analysis (PCA)

We have just seen that PCA can be applied to a set of points (e.g. features in a feature space) to produce a reference frame local to those points. The reference frame has equal dimension to the source points, and each of the basis vectors in the frame are mutually orthogonal and of unit length[1]. However the reference frame need not be oriented so as to be aligned with the principal axes of the space; indeed it usually is not. Furthermore the reference frame need not share the origin of the space; its origin is located at the mean of the point distribution.

However PCA is *not* limited to use in classification problems. We can apply PCA to any distribution of data points in any n-dimensional space, to gain an Eigenmodel (i.e. a reference frame that is a function of the data).

Consider, for example, a collision detection problem. Say we have modelled an object in 3D using many surface patches (defined by many vertices i.e. points in 3D). We want to know how far a test point $\underline{t}$ is from the model — say, how far a player's position is from an object in a computer game. One very (computationally) expensive way to compute this is to measure the Euclidean distance between $\underline{t}$ and every vertex in the model. This is certainly impractical for any real-time system with moderately detailed objects. One better, but approximate, approach would be to compute the mean position of the vertices and measure Euclidean distance between $\underline{t}$ and the mean. This would work well if the object was roughly spherical.

However a better estimate would be yielded by performing PCA on the vertices (3D points) and obtaining a data dependent reference frame (Eigenmodel) for the points. We then measure the **Mahalanobis distance** from $\underline{t}$ to the Eigenmodel. Although this is still an approximate solution, it is more accurate that a simple mean, and is also statistically grounded. After all, we are measuring how many standard deviations away from the Eigenmodel our

[1]Recall that PCA also gives us an eigenvalue (variance) along each basis vector, telling us how broadly the data is scattered in that direction through the space. However the eigenvalue is not a component of the reference frame which is simply the mean (origin)and eigenvalues (basis vectors)

point $\underline{t}$ is. The mathematics of PCA assume that points are **Gaussian distributed**[2]. From the normal distribution we can say, for example, that a point ≤ 3 standard deviations from the model is 97% probable to be a member of that distribution.

5.5.1 Recap: Computing PCA

To recap, and make explicit the separation of PCA from applications such as classification, we now restate the mathematics necessary to compute PCA of points $\underline{\underline{p}} = [\underline{p_1}\ \underline{p_2}\ ...\ \underline{p_n}]$ in a d-dimensional space.

First we compute the mean of $\underline{\underline{p}}$ i.e.:

$$\underline{\mu} = \frac{1}{n}\sum_{i=1}^{n} \underline{p_i} \tag{5.33}$$

Then we subtract $\underline{\mu}$ from each point in $\underline{\underline{p}}$ and compute a symmetric matrix as follows:

$$\underline{\underline{G}} = (\underline{\underline{p}} - \underline{\mu})(\underline{\underline{p}} - \underline{\mu})^T \tag{5.34}$$

This $d \times d$ matrix $\underline{\underline{G}}$ is known as a **Grammian matrix**; each element g_{ij} is the dot product of the ith row of $(\underline{\underline{p}} - \underline{\mu})$ and the jth row of $(\underline{\underline{p}} - \underline{\mu})^T$ i.e. the ith dimension of the points with the jth dimension. We can think (informally) of this dot product as a projection of the variance within one dimension of the space onto another. Thus this gives rise to the **covariance matrix** $\underline{\underline{C}}$ when divided by a normalising factor (the number of points):

$$\begin{aligned} \underline{\underline{C}} &= \frac{1}{n}\underline{\underline{G}} \\ \underline{\underline{C}} &= \frac{1}{n}(\underline{\underline{p}} - \underline{\mu})(\underline{\underline{p}} - \underline{\mu})^T \end{aligned} \tag{5.35}$$

We then apply EVD to $\underline{\underline{C}}$ to yield two $d \times d$ eigenvectors $\underline{\underline{U}}$ and eigenvalues $\underline{\underline{V}}$. These d eigenvector/values and the mean $\underline{\mu}$ form the Eigenmodel, that is the data dependent reference frame defined by point distribution $\underline{\underline{p}}$.

5.5.2 PCA for Visualisation

Often in Computer Graphics, or Computer Vision, we work with data in so called "high dimensional spaces" i.e. a n-dimensional space where $n > 3$. Consider our classification examples; it would be trivial to conceive of a real-world problem requiring measurement of more than 3 features i.e. a high dimensional feature space. The main problem with working in a high dimensional space is that we cannot plot (visualise) our data points very easily.

Usually when we plot data points we wish to visually inspect how those points are distributed i.e. how they vary from one another. We can apply PCA to a high dimensional point distribution to obtain a data dependent reference frame for those points, and also a measure of

[2] A Gaussian is a normal distribution with any mean and standard deviation (normal distribution has zero mean and standard deviation one)

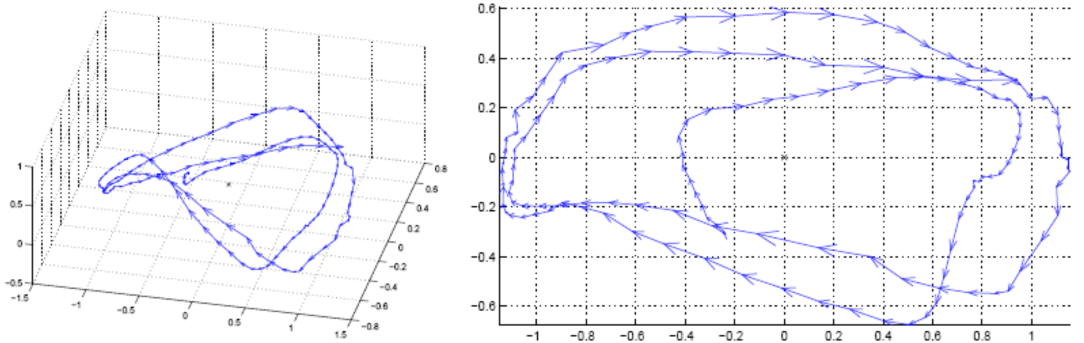


Figure 5.6: Visualisation of joint angles in a motion capture sequence of a human walking. The arrows indicate the flow of time. Note the cyclic nature of the trajectory in the pose space. The original dimensionality of the space was 36 — the data was projected down to 2D and 3D for visualization purposes. The projection was performed via the method outlined in Section 5.3.2.

variance of the data along each basis vector in that reference frame. We can then pick 2 or 3 basis vectors that exhibit high variance (i.e. eigenvectors that have the largest eigenvalues) and project all of our n-dimensional data points onto that lower dimensional basis set. We can then plot the resulting projected points in 2D or 3D — the resulting plot shows the modes of greatest variation in the data.

Another way to think of this is to imagine an object in 3D modeled by a set of points (vertices). Imagine we could not plot in 3D, but could only conceive of and interpret objects in 2D. We could use the perspective projection (Chapter 3) to project the "inconceivable" 3D object onto the 2D image plane, and thus get an impression of the distribution of those points. The PCA operation can be thought of translating and rotating the 3D points to the angle in which we could observe the greatest distribution of the points (i.e. the points are spread out as wide as possible in the image). The process outlined in the previous paragraph is doing just that, but projecting $> 3D$ points (which we cannot imagine/conceive an image of) to 2D or 3D.

As an example, consider an articulated body e.g. a walking human. If the human's joints are parameterised solely by angles, then we can describe a particular position of the human using only a list of angles. If we treated each angle as a dimension in a high dimensional space, then we could describe the pose of the human as a point in that space. As the human moved its limbs, so the point representing the human's pose would move around the high dimensional space. In fact in periodic motion (e.g. walking) we might imagine the points to trace out a cyclic trajectory in the high-dimensional space.

Such a model might contain a great many joints, resulting in points within a high dimensional space that we cannot plot. Yet, by performing PCA on the points (gathered over all time instants) we can identify the directions through the space in which most variation occurs, and project the points into that subspace. Figure 5.6 illustrates.

Such a projection is trivial to perform once we have the Eigenmodel from PCA. Suppose

we have a 20 dimensional space; we perform PCA on 20-dimensional points $\underline{\underline{p}}$ and obtain a mean $\underline{\mu}$ and 20×20 matrices $\underline{\underline{U}}$ and $\underline{\underline{V}}$ (eigenvectors and eigenvalues). We identify that, say, eigenvalues 14, 18 and 20 are the largest. We then form a 20×3 matrix $\underline{\underline{U'}} = [\underline{u_{14}}\ \underline{u_{18}}\ \underline{u_{20}}]$. This defines the 3D basis set we wish to project into (the subspace in which there was most variation). We can then subtract the mean $\underline{\mu}$ from our points $\underline{\underline{p}}$ (shifting their origin to the mean) and multiply by the inverse of $\underline{\underline{U'}}$ to project the mean subtracted points into the 3D space:

$$\underline{\underline{p'}} = \underline{\underline{U'}}^T (\underline{\underline{p}} - \underline{\mu}) \tag{5.36}$$

The resulting 3D points $\underline{\underline{p'}}$ can now be plotted. Note that we inverted matrix $\underline{\underline{U'}}$ by simply taking its transpose; this is possible because $\underline{\underline{U'}}$ is an orthonormal matrix i.e. eigenvectors are always mutually orthogonal and of unit length. This is convenient since $\underline{\underline{U'}}$ was not square and traditional matrix inversion techniques would therefore not be applicable.

5.5.3 Decomposing non-square matrices (SVD)

EVD operates only on square matrices, and will yield real eigenvalues only in the case of symmetric matrices. Although the covariance matrix will always satisfy this constraint (and so we can always perform PCA using EVD), there are many situations in which we may wish to compute eigenvectors and eigenvalues for non-square i.e. **rectangular matrices** or non-symmetric matrices. For this, there is a generalised form of EVD called **singular value decomposition (SVD)**. SVD decomposes a matrix $\underline{\underline{M}}$ into three matrices $\underline{\underline{U}}, \underline{\underline{S}}, \underline{\underline{V}}$ such that:

$$\underline{\underline{M}} = \underline{\underline{USV}}^T \tag{5.37}$$

If $\underline{\underline{M}}$ is an $m \times n$ matrix, then $\underline{\underline{U}}$ is $n \times n$, $\underline{\underline{S}}$ is $n \times m$ and $\underline{\underline{V}}$ is $m \times m$. The matrix $\underline{\underline{S}}$ is diagonal and contains the eigenvalues, and the matrix $\underline{\underline{V}}$ is square and contains the eigenvectors. We will not discuss SVD in any detail on this course; you may refer to Press *et al.*'s "Numerical Recipes in C" for a good introductory explanation and also C source code. Note that SVD works with square matrices also, and so can be used for most applications that require EVD. Thus if you wanted to perform EVD in your own programs, you might wish to make use of this code (readily available on the internet). The Intel OpenCV (Open Source Computer Vision) library also has a couple of SVD algorithms implemented.

One useful application of SVD is the **pseudo-inverse (Moore-Penrose inverse)** of any matrix $\underline{\underline{M}}$, including rectangular matrices. This is a generalised matrix inverse enabling us to derive a matrix $\underline{\underline{A}}$ such that $\underline{\underline{MA}} = \underline{\underline{I}}$ where $\underline{\underline{I}}$ is the identity; i.e. $\underline{\underline{A}}$ is the inverse of $\underline{\underline{M}}$. This is given by:

$$\underline{\underline{A}} = \underline{\underline{VS^+U}}^T \tag{5.38}$$

where $\underline{\underline{S^+}}$ refers to the diagonal matrix $\underline{\underline{S}}$ with its diagonal entries replaced with their reciprocal values.

Finally (for this course), we can also use SVD to solve homogeneous linear systems. That is, to find a non-trivial (i.e. non-zero) vector $\underline{x}$ to satisfy $\underline{\underline{Mx}} = 0$. We saw the need to solve such a system in Chapter 3, when deriving an expression for the homography. In this case

the eigenvectors (columns in $\underline{\underline{V}}$) give us potential solutions i.e. $\underline{x}$ whilst the corresponding eigenvalues give us a measure of error in the solution. If we consider the defining equation for eigenvectors/values:

$$\underline{\underline{M}}\underline{x} = \lambda \underline{x} \tag{5.39}$$

A perfect solution would have no error i.e. $\lambda = 0$, and that is is the $\underline{x}$ that annihilates $\underline{\underline{M}}\underline{x}$. A deeper justification for this and a proof are beyond this course's scope; here we note SVD only for optional further reading (e.g. Press *et al.*) and for its useful applications in both Computer Graphics and Computer Vision.

Chapter 6

Geometric Modelling

This Chapter is about geometric modelling. By "**modelling**" we mean "forming a mathematical representation of the world". The output of the process might be a representation (**model**) of anything from an object's shape, to the trajectory of an object through space. In Chapter 3 we learnt how to manipulate points in space using matrix transformations. Points are important because they form the basic building blocks for representing models in Computer Graphics. We can connect points with lines, or curves, to form 2D shapes or trajectories. Or we can connect 3D points with lines, curves or even surfaces in 3D.

6.1 Lines and Curves

We start our discussion of modelling by looking at **space curves**. Space curves are nothing more than trajectories through space; e.g. in 2D or 3D. A line is also a space curve, but a special case of one that happens to be straight. For avoidance of doubt, when we talk about curves in this Chapter we are referring to space curves in general, which include straight lines.

6.1.1 Explicit, Implicit and Parametric forms

We are all familiar with the equation $y = mx + c$ for specifying a 2D straight line with gradient m and y-axis intersection at $y = c$. However there is a major problem with using this equation to represent lines in general; vertical lines cannot be represented.

In general we can represent a 2D space curve using $y = f(x)$, i.e. a function that returns a value of y for any given value of x. We saw $f(x) = mx + c$ was a concrete example of this abstract definition; a straight line. However we are unable to represent all space curves in the form $y = f(x)$. Specifically, we cannot represent curves that cross a line $x = a$, where a is a constant, more than once. Figure 6.1.1 (left) gives an example of a curve we could not model using the $y = f(x)$ form. This is the more general explanation behind our observation that straight lines cannot be represented by $y = f(x)$.

We call the form $y = f(x)$ the explicit form of representation for space curves. It provides coordinate in one dimension (here, y) in terms of the others (here, x). It is clearly limited in the variety of curves it can represent, but it does have some advantages which we discuss later.

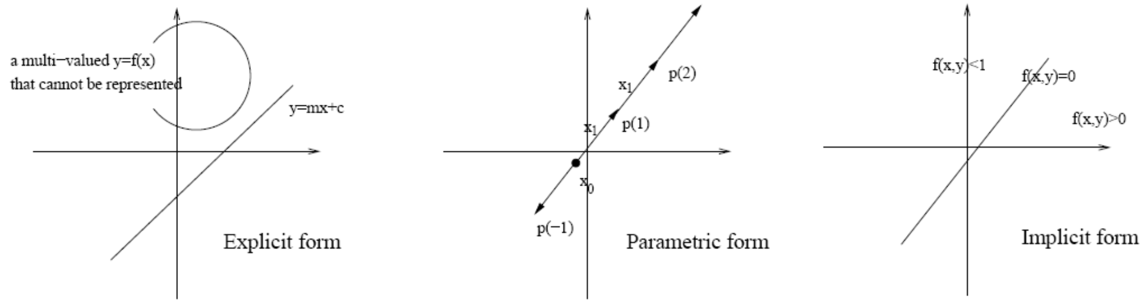


Figure 6.1: The explicit, parametric and implicit forms of a line.

So suppose we want to model a line, *any line*, in our Computer Graphics system. An alternative way to do this is to using the *parametric form*:

$$\underline{p}(s) = \underline{x_0} + s\underline{x_1} \tag{6.1}$$

Here vector $\underline{x_0}$ indicates the start point of the line, and $\underline{x_1}$ a vector in the positive direction of the line. We have introduced the dummy parameter s as a mechanism for iterating alone the line; given an s we obtain a point $\underline{p}$(s) some distance along the line. When s = 0 we are at the start of the line; positive values of s move us forward along the line and negative values of s move us backward (Figure 6.1, middle).

By convention we use $\underline{p}(0)$ to denote the start of a space curve, and $\underline{p}(1)$ to denote the end point of the curve. That is, we increase s from 0 to 1 to move along the entire, finite length of the curve.

We can generalise this concept to produce curved trajectories. Consider $\underline{p}(s)$ as a the position of a particle flying through space, and s as analogous to time. Then $\underline{x_0}$ is the start position of the particle, and $\underline{x_1}$ is the velocity of the particle. We can add a term $\underline{x_2}$ for acceleration as follows:

$$\underline{p}(s) = \underline{x_0} + s\underline{x_1} + s^2\underline{x_2} \tag{6.2}$$

to yield a quadratic curve. We can continue adding terms to any order n:

$$\underline{p}(s) = \sum_{i=1}^{n} s^i \underline{x_i} \tag{6.3}$$

although in practice it is unusual in Computer Graphics to use anything above $n = 3$ (cubic curves), for reasons we later explain (Section 6.1.2). For completeness, we note that other parametric forms of curve are of course available. For example a more convenient parameterisation of a circle might be:

$$\underline{p}(\theta) = \begin{pmatrix} r\cos\theta \\ r\sin\theta \end{pmatrix} \tag{6.4}$$

The final representation of space curve that we will look at is the *implicit form*. In implicit form we express the equation of a curve as a function of all of its coordinates equal to a

constant (often zero), e.g. in 2D we have $f(x, y) = 0$. We can reach the implicit form for a 2D line easily from the parametric form, by writing out the x and y components of the equation. Here we write $\underline{x_0} = [\underline{x_0}\ \underline{y_0}]^T$ and $\underline{x_1} = [\underline{u}\ \underline{v}]^T$:

$$
\begin{aligned}
x &= x_0 + su \\
y &= y + 0 + sv
\end{aligned}
\tag{6.5}
$$

Rearranging for s we get:

$$\frac{x - x_0}{u} = s = \frac{y - y_0}{v} \tag{6.6}$$

$$(x - x_0)v = (y - y_0)u \tag{6.7}$$

$$(x - x_0)v - (y - y_0)u = 0 \tag{6.8}$$

where x_0, y_0, u, and v are constants defining the line. We insert values for a given point (x, y) and if the equation is satisfied (i.e. the left hand side is zero) we are at a point on the line. Furthermore the term on the left hand side is positive if we are off the line but on one side of it, and negative if we are on the other side. This is because the left hand side evaluates the **signed distance** from the point (x, y) to the nearest point on the line (Figure 6.1, right).

The **explicit**, **parametric** and **implicit** forms of a space curve have now been described with the concrete example of a straight line given for each form.

Which form to use?

Each of the three forms have particular advantages over the others; the decision of which to use depends on your application. The implicit form is very useful when we need to know if a given point (x, y) is on a line, or we need to know how far or on which side of a line that point lies. This is useful in Computer Games applications e.g. "clipping" when we need to determine whether a player is on one side of a wall or another, or if they have walked into a wall (collision detection). The other forms can not be used to easily determine this. The parametric form allows us to easily iterate along the path of a space curve; this is useful in a very wide range of applications e.g. ray tracing or modelling of object trajectories. Again, it is hard to iterate along space curves using the other two forms. Finally, the explicit form is useful when we need to know one coordinate of a curve in terms of the others. This is less commonly useful, but one example is Bresenham's algorithm; an integer-only algorithm for plotting straight lines on raster displays. You can look this up in most raster Computer Graphics textbooks (discussion is beyond the scope of this course).

6.1.2 Parametric Space Curves

Parametric space curves are by far the most common form of curve in Computer Graphics. This is because: (a) they generalise to any dimension; the $\underline{x_i}$ can be vectors in 2D, 3D, etc.; and (b) we usually need to iterate along shape boundaries or trajectories modelled by such curves. It is also trivial to differentiate curves in this form with respect to s, and so obtain tangents to the curve or even higher order derivatives (see discussion of the Frenet Frame in Section 6.3.1).

Let us suppose that we want to model the outline of a teapot in 2D. We could attempt to do so using a parametric cubic curve, but the curve would need to turn several times to produce the complex outline of the teapot, implying a very high order curve. Such a curve perhaps be of order $n = 20$ or more, requiring the same number of $\underline{x_i}$ vectors to control its shape (representing position, velocity, acceleration, rate of change of acceleration, rate of change of rate of change of acceleration, and so on). Clearly such a curve is very difficult to control (fit) to the required shape, and in practice using such curves in manageable; we tend to over-fit the curve. The result is usually that the curve approximates the correct shape, but undulates (wobbles) along that path. There are an infinite set of curve trajectories passing through a pre-defined set of points, and finding acceptable fits (usually the fits in which the curve smoothly passes through all points without undulation) becomes harder as the number of points on the path (and so order of the polynomial) increases.

Therefore we usually model the complex shape by breaking it up into simpler curves and fitting each of these in turn. It is common practice in Computer Graphics to use cubic curves, to give a compromise between ease of control and expressiveness of shape:

$$\underline{p}(s) = \underline{x_0} + s\underline{x_1} + s^2\underline{x_2} + s^3\underline{x_3} \tag{6.9}$$

More commonly we write cubic curves as inner products of the form:

$$\begin{aligned} \underline{p}(s) &= \begin{bmatrix} \underline{x_3} & \underline{x_2} & \underline{x_1} & \underline{x_0} \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \\ \underline{p}(s) &= \underline{\underline{C}}\,\underline{\underline{Q}}(s) \end{aligned} \tag{6.10}$$

where $\underline{\underline{C}}$ is a matrix containing the xi vectors that control the shape of the curve, and $\underline{\underline{Q}}(s)$ contains the parameterisation that moves us along the curve.

Consideration of Continuity

When modelling a shape in piecewise fashion, i.e. as a collection of joined-together (concatenated) curves, we need to consider the nature of the join. We might like to join the curves together in a "smooth" way so that no kinks are visually evident; after all, the appearance of the shape to the user should ideally be independent of the technique we have used to model it. The question arises then, what do we mean by a "smooth" join? In computer graphics we use C^n and G^n notation to talk about the smoothness or **continuity** of the join between piecewise curves.

If two curves join so that the end point of the first curve ($\underline{p_1}(1)$) is the starting point of the second curve ($\underline{p_2}(0)$), we say that the curves have zero-th order or C^0 continuity.

If the two curves have C^0 continuity and the tangent at the end of the first curve $\underline{p_1}'(0)$ matches the tangent at the start of the second curve $\underline{p_2}'(1)$, then we say the curves have first order or C^1 continuity. This is because both their zero-th differentials (i.e. position) and first order differentials (i.e. tangents) match at the join. Another way to think about this returns to the idea of a particle moving along the first curve, across the join and then along the second curve. If the curves meet, and the particle's velocity does not change across the

join, then the curves have C 1 continuity.

The idea extends trivially to higher orders of continuity although in practice greater than C^2 continuity is rarely useful. G^n refers to "geometric continuity". In brief a curve is G^1 continuous if the tangent vectors have the same direction (but they need not have the same magnitude). Thus C^n implies G^n but not vice versa; G^n is a weaker definition of continuity. Clearly certain applications require us to define piecewise curves with particular orders of continuity. If we were using piecewise cubic curves to model the trajectory of a roller-coaster in an animation, we would want at least C^1 continuity to preserve velocity over the joins in the model. We would not want the roller-coaster to alter its velocity arbitrarily at join points as this would disrupt the quality of the animation. Ideally the decision to model a curve piecewise, and the method chosen to do so, should not affect the application i.e. be transparent to the user.

6.2 Families of Curve

Eq. (6.10) defined a curve using $\underline{p}(s) = \underline{\underline{C}}\underline{\underline{Q}}(s)$, where $\underline{c}$ comprises fours vectors that determine the shape of the curve. Recall that using the physical analogy of a particle $\underline{p}$ moving along a trajectory as s increases, these vectors represent initial position, velocity, acceleration, and rate of change of acceleration. However it is very hard to control the shape of a curve by manipulating these vector quantities. It is also difficult to choose appropriate values to ensure a particular order of continuity between curve joins when modelling curves in piecewise fashion.

To make curves easier to control, we separate $\underline{\underline{C}}$ into two matrices $\underline{\underline{G}}$ and $\underline{\underline{M}}$ as follows:

$$\underline{p}(s) = \underline{\underline{G}}\underline{\underline{M}}\underline{\underline{Q}}(s) \tag{6.11}$$

We call $\underline{\underline{M}}$ the **blending matrix** and $\underline{\underline{G}}$ the **geometry matrix**. By introducing a particular 4×4 matrix $\underline{\underline{M}}$ we can change the meaning of the vectors in $\underline{\underline{G}}$, making curve control more intuitive, as we shall see in a moment. Note that when $\underline{\underline{M}}$ is the identity, then the vectors in $\underline{\underline{G}}$ have the same function as in our $\underline{\underline{C}}\underline{\underline{Q}}(s)$ formulation. Informally, we say that $\underline{\underline{M}}$ defines the family of curves we are working with. We will now look at four curve families, each of which uses a different $\underline{\underline{M}}$.

6.2.1 Hermite Curves

The Hermite curve has the form (relate this to eq. 6.11):

$$\underline{p}(s) = \begin{bmatrix} \underline{p}(0) & \underline{p}(1) & \underline{p}'(0) & \underline{p}'(1) \end{bmatrix} \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.12}$$

Here we can see $\underline{\underline{M}}$ has been set to a constant 4×4 matrix that indicates we are working with the Hermite family of curves. This changes the meaning of the vectors comprising the geometry matrix $\underline{\underline{G}}$. The vectors are now (from left to right), the start point of the curve, the end point of the curve, the start *tangent* of the curve, the end *tangent* of the curve (Figure 6.2).

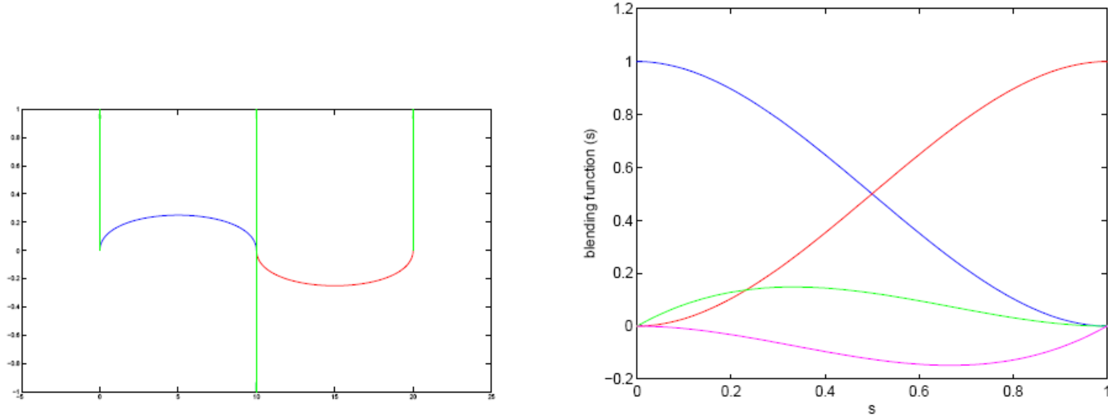


Figure 6.2: Example of two Hermite curves (red, blue) joined with C^1 continuity; tangents indicated with green arrows. A plot of the blending functions ($\underline{\underline{M}}\underline{\underline{Q}}(s)$) for the Hermite curve.

It is now trivial to model complex trajectories in piecewise fashion with either C^0 or C^1 continuity. We simply ensure that the curves start and end with the same positions (C^0) and tangents (C^1). Note that every Hermite curve has the same, constant, $\underline{\underline{M}}$ as specified above. We change only $\underline{\underline{G}}$ to create curves of different shapes. As before, matrix $\underline{\underline{Q}}(s)$ controls where we are along the curve's trajectory; i.e. we substitute in a particular value s between 0 and 1 to evaluate our position along the curve $\underline{p}(s)$.

Derivation of $\underline{\underline{M}}$ for Hermite curves

We have yet to explain the values of the 16 scalar constants comprising $\underline{\underline{M}}$. We will now derive these, but first make the observation that we can easily compute the tangent at any point on a curve by differentiating eq. (6.11) with respect to s. Only matrix $\underline{\underline{Q}}(s)$ has terms dependent on s, and so is the only matrix that changes:

$$\underline{p}(s) = \underline{\underline{G}}\,\underline{\underline{M}} \begin{bmatrix} 3s^2 \\ 2s \\ 1 \\ 0 \end{bmatrix} \tag{6.13}$$

We also make the observation that we can compute the coordinates/tangent of more than point on the curve by adding extra columns to $\underline{\underline{Q}}(s)$ e.g.:

$$\begin{bmatrix} \underline{p}(s) & \underline{p}'(s) \end{bmatrix} = \underline{\underline{G}}\,\underline{\underline{M}} \begin{bmatrix} s^3 & 3s^2 \\ s^2 & 2s \\ s & 1 \\ 1 & 0 \end{bmatrix} \tag{6.14}$$

Now, to derive $\underline{\underline{M}}$ for the Hermite curve we setup an arbitrary $\underline{\underline{G}}$ and evaluate the coordinates and tangents of the curve at the start/end respectively. Since those are exactly the vectors used to define $\underline{\underline{G}}$ in a Hermite curve, the matrix $\underline{\underline{G}}$ and the left-hand side of the equation are identical:

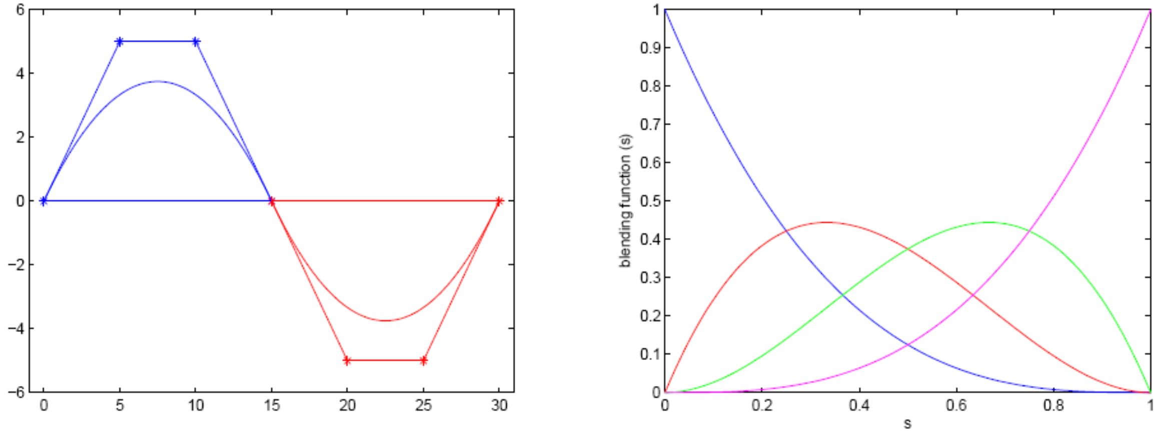


Figure 6.3: Diagram of two Bézier curves (red, blue) joined with C^1 continuity (control polygons also indicated). Note the two interpolated and two approximated points on each curve, and the co-linearity of points at the join to generate the C^1 continuity. A plot of the blending functions ($\underline{\underline{M}}\underline{\underline{Q}}(s)$) for the Bézier curve.

$$
\begin{aligned}
\begin{bmatrix} \underline{p}(0) & \underline{p}(1) & \underline{p}'(0) & \underline{p}'(1) \end{bmatrix} &= \begin{bmatrix} \underline{p}(0) & \underline{p}(1) & \underline{p}'(0) & \underline{p}'(1) \end{bmatrix} \underline{\underline{M}} \begin{bmatrix} s^3 & s^3 & 3s^2 & 3s^2 \\ s^2 & s^2 & 2s & 2s \\ s & s & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \\
\begin{bmatrix} \underline{p}(0) & \underline{p}(1) & \underline{p}'(0) & \underline{p}'(1) \end{bmatrix} &= \begin{bmatrix} \underline{p}(0) & \underline{p}(1) & \underline{p}'(0) & \underline{p}'(1) \end{bmatrix} \underline{\underline{M}} \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}
\end{aligned}
\qquad (6.15)
$$

Cancelling the left-hand side and $\underline{\underline{G}}$, then rearranging yields our $\underline{\underline{M}}$:

$$
\underline{\underline{I}} = \underline{\underline{M}} \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \qquad (6.16)
$$

$$
\underline{\underline{M}} = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}^{-1} \qquad (6.17)
$$

Which when evaluated produces the 4×4 matrix $\underline{\underline{M}}$ used in eq. (6.12).

The Hermite curve is an example of an **interpolating curve**, because it passes through all of the points used to specify it (in $\underline{\underline{G}}$).

6.2.2 Bézier Curve

The Bézier curve is an example of an **approximating curve**. It is specified using coordinates of four points (known as **knots** or, more commonly, as **control points**). The curves

passes through (interpolates) two of these points, and in general approximates (passes close to) the two other points.

Specifying a curve using spatial positions (rather than derivatives) makes the Bézier curve a very intuitive modelling technique. In fact the curve models the physical techniques used in manufacturing to shape thin strips of metal, called **splines** (Pierre Bézier was working for engineering firm Renault when he developed the curve in the 1960s). The metal spline is nailed e.g. to a workbench at each end, and masses suspended beneath it. The nailed points are analogous to the interpolated points, and the masses analogous to the approximated points. For this reason the term **spline** is also used in Computer Graphics to describe any curve we can shape to our needs by specifying spatial control points; however the term is used particularly frequently in piecewise modelling. The formulation of the Bézier curve is:

$$\underline{p}(s) = \begin{bmatrix} \underline{p}(0) & \underline{a_0} & \underline{a_1} & \underline{p}(1) \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.18}$$

Points $\underline{p}(0)$ and $\underline{p}(1)$ are the start and end points of the curve respectively. Points $\underline{a_0}$ and $\underline{a_1}$ are the approximated points (Figure 6.3). The **convex hull** (i.e. convex polygon) formed by those four points are guaranteed to enclose the complete trajectory of the curve. The special case of a straight line (when all four points are co-linear) is the only case is which $\underline{a_0}$ and $\underline{a_1}$ are interpolated.

It is therefore trivial to achieve C^0 continuity for Bézier curves; as with the Hermite curve we ensure that the end and start points of the two respective curves are identical. Less obvious is that we can easily enforce C^1 continuity with Bézier curves. We do this by ensuring $\underline{a_1}$ on the first curve, and $\underline{a_0}$ on the second curve are co-linear and equidistant from the join point $\underline{p}(1)$ / $\underline{p}(0)$ (Figure 6.3).

The Blending matrix $\underline{\underline{M}}$ for Bézier curves

Although we have outlined only cubic Bézier curves here, the Bézier curve can be of any degree n and the matrix formulation of eq. 6.18 is a rearrangement of the Bernstein polynomials:

$$\underline{p}(s) = \sum_{i=0}^{n} \underline{x_i} \frac{n!}{i!(n-1)!} s^i (1-s)^{n-i} \tag{6.19}$$

where, in our cubic curve, $n = 3$ and $\underline{x_{1..4}}$ are the four vectors comprising the $\underline{\underline{G}}$ matrix. Further discussion of the polynomials is beyond the scope of this course. However we will now discuss how they operate to determine the behaviour of the curve.

Recall that we refer to $\underline{\underline{M}}$ as the blending matrix. We can see that, during matrix multiplication, the values in the matrix weights (i.e. blends) the contribution of each control point

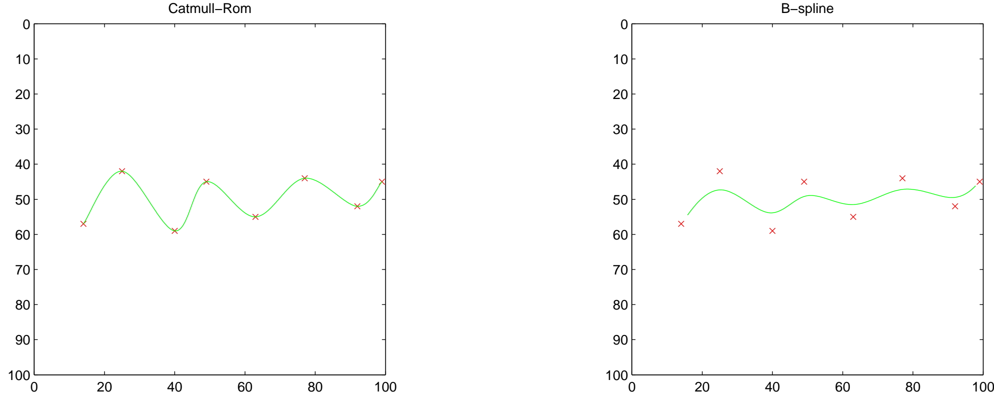


Figure 6.4: Eight points used to define the geometry of a Catmull-Rom piecewise spline (left) and a piecewise B-spline (right). Note that the Catmull-Rom spline interpolates all points, whereas the B-spline approximates all points.

in $\underline{\underline{G}}$ to the summation producing $\underline{p}(s)$ i.e.:

$$\underline{\underline{M}}\underline{Q}(s) = \begin{bmatrix} -s^3 + 3s^2 - 3s + 1 \\ 3s^3 - 6s^2 + 3s \\ -3s^2 + 3s^2 \\ s^3 \end{bmatrix} \tag{6.20}$$

$$\underline{p}(s) = \underline{\underline{G}}\,\underline{\underline{M}}\underline{Q}(s) = \begin{bmatrix} \underline{p}(0) & \underline{a_0} & \underline{a_1} & \underline{p}(1) \end{bmatrix} \begin{bmatrix} -s^3 + 3s^2 - 3s + 1 \\ 3s^3 - 6s^2 + 3s \\ -3s^2 + 3s^2 \\ s^3 \end{bmatrix} \tag{6.21}$$

Figure 6.3 (right) plots the blending function in each of the 4 rows of $\underline{\underline{M}}\underline{Q}(s)$ as s increases from 0 to 1. We see that when $s = 0$, the weighting is entirely towards point $\underline{p}(0)$. As s increases, the weighting towards that point decreases whilst the weight towards $\underline{a_0}$ increases (although this term never completely dominates the contribution to $\underline{p}(s)$). As s increases further still, greater weighting is attributed to $\underline{a_1}$, and finally all weighting is towards $\underline{p}(1)$.

6.2.3 Catmull-Rom spline

The **Catmull-Rom spline** allows us to specify piecewise trajectories with C^1 continuity that interpolate (pass through) all curve control points. This is particularly convenient when we wish to model a smooth curve (or surface) by simply specifying points it should pass through rather than approximate.

The Catmull Rom spline is specified through the framework of eq. (6.11) as follows:

$$\underline{p}(s) = \begin{bmatrix} \underline{a} & \underline{p}(0) & \underline{p}(1) & \underline{b} \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.22}$$

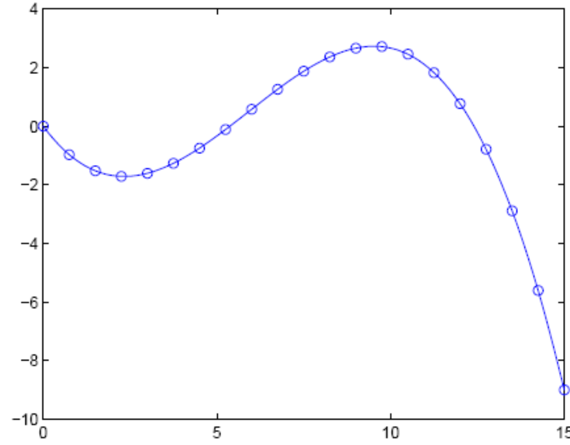


Figure 6.5: A cubic curve $\underline{p}(s) = \underline{\underline{G}}\,\underline{\underline{M}}\,\underline{Q}(s)$. Increasing s in constant increments does not move us a constant distance along the curve at each point. The distance we move is a function of the curve's geometry; ideally we would like the distance moved to be independent of the geometry.

where $\underline{a}$, $\underline{p}(0)$, $\underline{p}(1)$, $\underline{b}$ are points we wish the curve to pass through.

In fact by evaluating eq.(6.22) from $s = 0$ to $s = 1$ we will obtain points only for the section of curve between the second and third vectors in $\underline{\underline{G}}$ i.e. the points indicated $\underline{p}(0)$ and $\underline{p}(1)$. The points $\underline{a}$ and $\underline{b}$ only help to shape the path of the curve. We must use several Catmull-Rom curves to trace out a piecewise trajectory in full.

An an example, consider six points we wish to interpolate with a piecewise Catmull-Rom cubic curve. We write these points $\underline{p_i}$ where $i = [1, 6]$. We can find the path of a curve through all the points by evaluating several $\underline{p}(s) = \underline{\underline{G_j}}\,\underline{\underline{M}}\,\underline{Q}(s)$ equations with the following $\underline{\underline{G_j}}$ where $j = [1, 5]$:

$$
\begin{aligned}
\underline{\underline{G_1}} &= \begin{bmatrix} \underline{p_1} & \underline{p_1} & \underline{p_2} & \underline{p_3} \end{bmatrix} \\
\underline{\underline{G_2}} &= \begin{bmatrix} \underline{p_1} & \underline{p_2} & \underline{p_3} & \underline{p_4} \end{bmatrix} \\
\underline{\underline{G_3}} &= \begin{bmatrix} \underline{p_2} & \underline{p_3} & \underline{p_4} & \underline{p_5} \end{bmatrix} \\
\underline{\underline{G_4}} &= \begin{bmatrix} \underline{p_3} & \underline{p_4} & \underline{p_5} & \underline{p_6} \end{bmatrix} \\
\underline{\underline{G_5}} &= \begin{bmatrix} \underline{p_4} & \underline{p_5} & \underline{p_6} & \underline{p_6} \end{bmatrix}
\end{aligned}
\tag{6.23}
$$

So if we plot the $\underline{p}(s)$ generated geometry matrix $\underline{\underline{G_j}}$ plugged into eq.(6.22) we will interpolate between points $\underline{p_j}$ and $\underline{\underline{p_{j+1}}}$. Note how points are allocated to $\underline{\underline{G_j}}$ in a "rolling" manner, and also duplicated at the beginning and end of the piecewise curve. Figure 6.4 illustrates.

6.2.4 β-spline

The β-spline (or **B-spline**) is very similar in function to the Catmull-Rom spline. It is also specified using a "rolling" form of matrix $\underline{\underline{G_j}}$ but instead of interpolating all four points with C^1 continuity, it **approximates** all four with C^1 continuity. The piecewise curve is simply formed with an alternative form of blending matrix:

$$\underline{p}(s) = \begin{bmatrix} \underline{a} & \underline{p}(0) & \underline{p}(1) & \underline{b} \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.24}$$

This can be useful when fitting a curve to noisy data, for example.

Unlike the Hermite and Bézier curves, we will not explain the functionality of the Catmull-Rom and B-spline blending matrices on this course, but interested readers may referred to "Computer Graphics" (Foley *et al.*) for further details on these and other types of spline.

6.3 Curve Parameterisation

So far we have discussed iteration along a parametric curve by varying continuous parameter s, but have not discussed the nature of this parameter (other than, by convention, s varies between 0 and 1).

In many applications we would like s to be a measure of distance (**arc length**) along the curve. That is, increasing s in constant increments will take us a constant distance along the curve. Unfortunately the 'natural' i.e. **algebraic parameterisation** of $\underline{p}(s) = \underline{\underline{GMQ}}(s)$ does not behave this way in general, i.e. not for orders of curve greater than 1 (straight lines). Figure 6.5 illustrates; increasing s by equal increments causes the resulting points $\underline{p}(s)$ to bunch together in high curvature regions of the trajectory.

Therefore for many applications we must re-parameterise the curve to be in terms of arc-length (an **arc-length parameterisation**). We do this by writing $\underline{p}(c(s))$ where we define $c(s)$ as a **transfer function** accepting an arc-length parameter as input, and outputting an algebraic parameterisation that takes us to the correct position on the curve. For example, $s = 0.5$ should take us halfway along the curve, but the algebraic parameter we actually need to feed into $\underline{\underline{GMQ}}(.)$ may be something different – obtained via $c(0.5)$. Unfortunately we cannot write down an equation (analytic solution) for $c(s)$ for a cubic curve; it is impossible to solve this arc-length re-parameterisation task in closed form for orders of curve greater than 1 (this order of 'curve' is arc-length parameterised anyway). Instead we resort to a numerical approximation using the following algorithm.

Arc-length re-parameterisation via piecewise linear approximation

We construct function $c(s)$ by creating a lookup table mapping between s (which is now the input arc-length parameter) and $c(s)$ which is the algebraic parameter. A simple algorithm for creating such a lookup table follows:

1. Let `alg=0`. Let `arclen=0`.
2. Let d be some very small length, e.g. 0.1% of estimated total curve length.
3. Loop while (`algiter` < 1):
   a Add entry to lookup table mapping $c(s)$=`arclen` to s=`alg`.
   b Let `algiter` = `alg`.
   c Increase `algiter` by d while $||\underline{p}(\texttt{algiter}) - \underline{p}(\texttt{alg})| - d| < 0$.
   d Set `arclen`=`arclen` + $|\underline{p}(\texttt{algiter}) - \underline{p}(\texttt{alg})|$
   e Set `alg`=`algiter`
4. Normalise the s column in the lookup table so it ranges 0 to 1 (i.e. divide by the largest value in the column).

Other more complicated algorithms based on recursive subdivision exist. The result is a normalised arc-length parameter $s = [0, 1]$ mapped to a normalised algebraic parameter $c(s) = [0, 1]$. Since the values in the lookup table are a function of the curve's shape, if we change $\underline{\underline{G}}$ in eq.(6.11) we must re-run the algorithm.

In summary, we approximate the curve by a series of very small lines of approximate length d (a user selectable value, but usually chosen as a compromise between visual smoothness of approximation and speed).

At each iteration we incremented the algebraic parameter of the curve by very small amounts until we travelled distance d. We keep a count of how far we have travelled so far in `arclen`. Thus after moving along the curve d distance we have a pair mapping algebraic distance to arc-length distance, which we write into the lookup table. Clearly this lookup table can now be used to translate arc-length parameterisation to algebraic parameterisation; i.e. to implement $c(s)$. If a value for an arc-length parameterisation is not present in the table, then we interpolate it from surround values. Since we approximated the curve with small lines, it is valid to use linear interpolation. E.g. if a table contains two values s_1 and s_2, then for a value s where $s_1 < s < s_2$ we compute:

$$
\begin{aligned}
\alpha &= (s - s_1)/(s_2 - s_1) \\
c(s) &= c(s_1) + \alpha(c(s_2) - c(s_1))
\end{aligned}
\tag{6.25}
$$

6.3.1 Frenet Frame

We have already seen that the parametric cubic space curve is a very useful modelling tool, allowing us to define trajectories and iterate (move) along them to obtain a position vector $\underline{p}(s) = \underline{\underline{GMQ}}(s)$. We have also already seen that it is possible to obtain more than simply a position at each point; we can obtain a tangent vector $\underline{p}(s)$ by differentiating $\underline{\underline{GMQ}}(s)$ once with respect to s.
It transpires that the tangent is just one vector component of the Frenet Frame; a natural reference frame commonly used with 3D parametric space curves. The components of the Frenet Frame are summarised in Figure 6.6. On this course we do not concern ourselves with how the Frenet Frame is derived; simply how its various components are computed and what

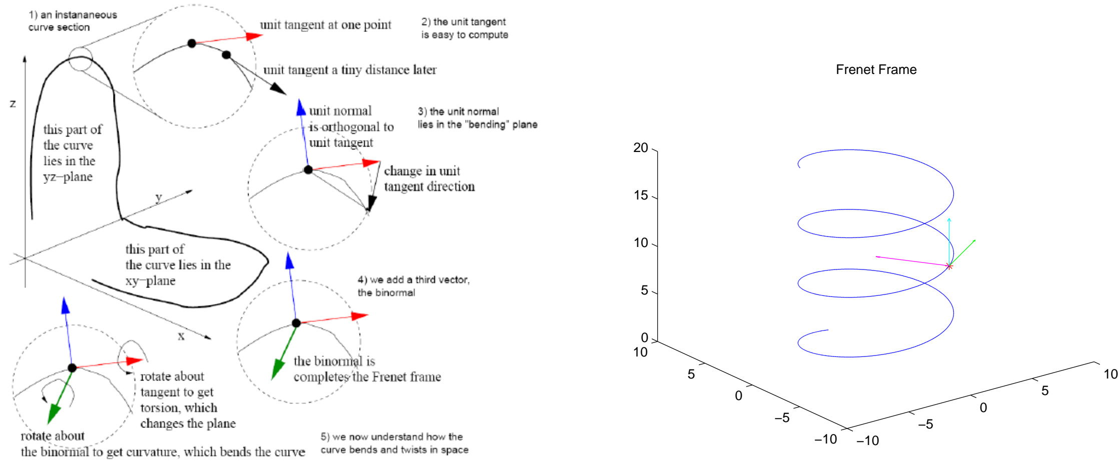


Figure 6.6: Left: The development of the Frenet frame – a natural reference frame for any parametric curve providing all derivatives defined in the frame exist. Right: Visualisation of the Frenet frame computed over a parametric 3D space curve; a spiral. Tangent in green, normal in purple (recall this and the tangent define the bending plane; note the direction of bending), and bi-normal in cyan.

their applications are.

To recap, the tangent to the space curve is:

$$\underline{p}'(s) = \frac{\delta \underline{p}(s)}{\delta s} \tag{6.26}$$

i.e. the unit tangent $\hat{\underline{p}'(2)}$ points along the *instantaneous direction* of the curve. This unit tangent is the first component of the Frenet Frame.

The **normal** to the curve is another vector component of the Frenet Frame. It is orthogonal to the tangent, and so at 90° to the tangent (and so, the curve). However this statement alone is insufficient to define the normal; we could imagine many possible normal vectors pointing away from the curve orthogonal to the tangent at any point. We need to fix its direction. To do so we consider that, given an infinitesimally small increment of s, the curve bends (i.e. the tangent changes) only in one plane (the **blending plane**). The amount of 'bend' is exactly the rate at which the unit tangent changes direction i.e.

$$\underline{p}''(s) = \frac{\delta \underline{p}'(s)}{\delta s} \tag{6.27}$$

This second derivative thus yields the normal, fixed in a consistent direction, and is the second component of the Frenet Frame. The magnitude $\rho = |\underline{p}(s)|$ is the **curvature** of the space curve at s. At that point, the curve can be considered to bend around the arc of a circle radius $1/\rho$.

The third and final component of the Frenet Frame is the **bi-normal**. The bi-normal is simply the cross product of the unit tangent and unit normal; i.e. it is orthogonal to both.

$$\underline{b}(s) = \underline{p}'(s) \times \underline{p}''(s) \tag{6.28}$$

The three vector components of the Frenet Frame can be connected in another way:

$$\frac{\delta \underline{p}''(s)}{\delta s} = -\rho \underline{p}'(s) + \tau \underline{b}(s) \tag{6.29}$$

i.e. the rate of change of the normal (i.e. the change in curvature) is due to two components: twisting about the bi-normal (the curvature) and twisting about the tangent (called the **torsion**), which is what we might intuitively expect.

As an application of the Frenet Frame, consider a roller-coaster moving along the trajectory of a parametric space curve. We wish to mount a camera on the front of the roller-coaster looking forward. This requires us to define a stable reference frame on the front of the roller-coaster; the Frenet Frame gives us this. We might point the camera along the tangent, with an image plane aligned such that the y-axis of the image plane points along the normal, and the x-axis of the plane points along the bi-normal.

Example of Frenet Frame calculation

Consider a parametric space curve that traces out a spiral in 3D space. Such a curve might be generated by a 2D parametric equation for a circle ($\underline{p}(\theta) = [-\sin\theta \;\; \cos\theta]^T$), as the x-axis and y-axis coordinates, with the addition of θ as a distance along the z-axis:

$$\underline{p}(\theta) = \begin{bmatrix} -\cos\theta \\ -\sin\theta \\ \theta \end{bmatrix} \tag{6.30}$$

We can easily compute the first two components of the Frenet frame (tangent and normal) by computing the first two derivatives of the curve:

$$\underline{p}'(\theta) = \begin{bmatrix} \sin\theta \\ -\cos\theta \\ 0 \end{bmatrix} \tag{6.31}$$

$$\underline{p}''(\theta) = \begin{bmatrix} -\cos\theta \\ -\sin\theta \\ 0 \end{bmatrix} \tag{6.32}$$

And the bi-normal is the cross product of the tangent and normal: $\underline{p}'(\theta) \times \underline{p}''(\theta)$. Figure 6.6 (right) plots the curve and the three Frenet Frame vectors at a given value of θ.

6.4 Surfaces

In Section 6.1.1 we saw how space curves can be modelled in explicit, implicit, and parametric forms. This concept generalises to mathematical descriptions of surfaces, which are very commonly used in Computer Graphics to represent objects – again, usually in a piecewise fashion.

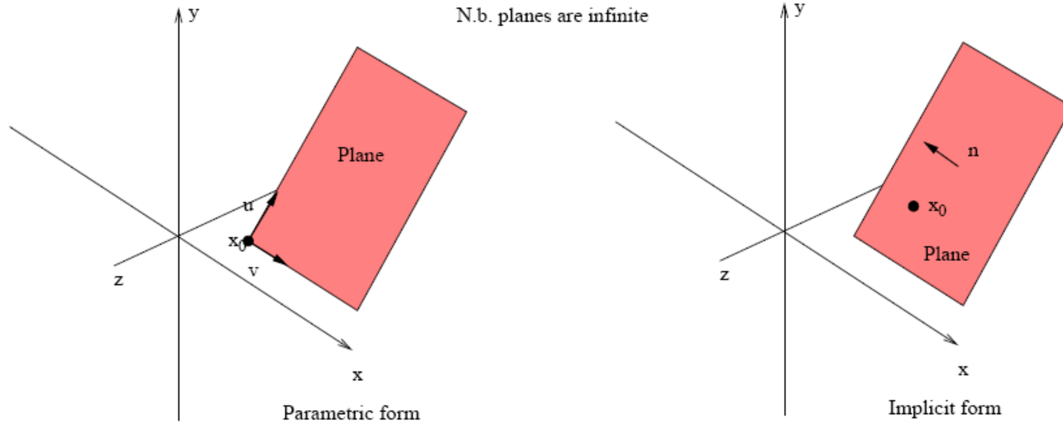


Figure 6.7: Illustrating the parametric and implicit forms of a plane.

6.4.1 Planar Surfaces

We can generalise our parametric equation for a line to obtain an equation for an infinite plane. We simply add another term; the product of a new parameter and a new vector:

$$\underline{p}(s,r) = \underline{x_0} + s\underline{u} + r\underline{v} \tag{6.33}$$

Figure 6.7 illustrates this; we have a 2D parameter space (s,r) that enable us to iterate over a 3D surface. The origin of that parameter space is at $\underline{x_0}$. Vectors $\underline{u}$ and $\underline{v}$ orient the plane; they effectively define a set of basis vectors that specify a reference frame with respect to the geometry of the surface. We can bound this infinite plane by bounding acceptable values of s and r. Later we will return to the parametric equation for a plane when we discuss texture mapping.

Infinite planes can be defined in implicit form too; using only an origin (a point $\underline{c}$ on the plane) and a vector normal to the plane ($\underline{\hat{n}}$). The vector between $\underline{c}$ and any point on the plane must (by definition) be orthogonal to $\underline{\hat{n}}$ and so have:

$$(\underline{p} - \underline{c}) \circ \underline{\hat{n}} = 0 \tag{6.34}$$

Here we have written the normal as unit length (normalised) but this needn't be the case generally (it is the convention to do so however). If we write $\underline{c} = [c_x \; c_y \; c_z]^T$, $\underline{\hat{n}} = [n_x \; n_y \; n_z]^T$, and $\underline{p} = [x \; y \; z]^T$ then:

$$n_x(x - c_x) + n_y(y - c_y) + n_z(z - c_z) = 0 \tag{6.35}$$

i.e. we have our familiar implicit form of model $f(x,y,z) = 0$; a function evaluating all coordinates in the space, equal to 0 on the model. We can use the left-hand side of equation to determine if a point is above or below the plane by checking its sign for a given (x,y,z).

For completeness we note that infinite planes may also be defined in an explicit form:

$$z = ax + bx + c \tag{6.36}$$

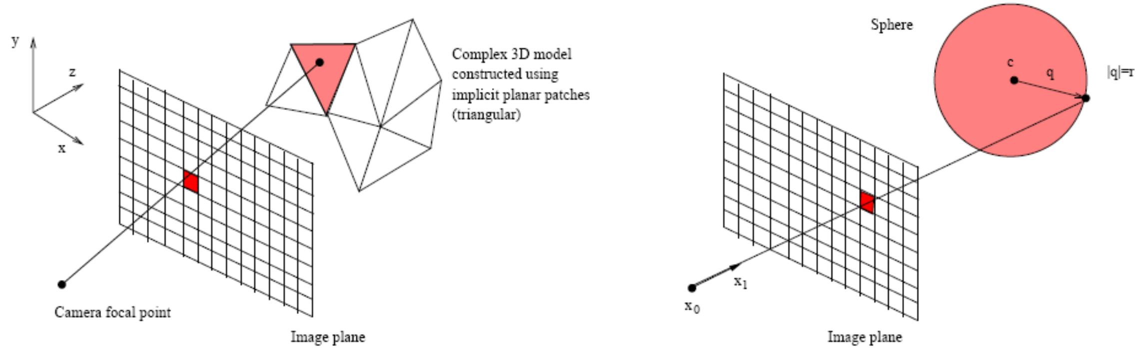


Figure 6.8: Illustrating the Ray Tracing of a model constructed piecewise using triangular planar patches, and a sphere. A camera is modelled in 3D using a focal point and a plane (identical to the perspective construction in Chapter 3). However instead of computing perspective projection of points in the model, we model the interaction of light with the surface "in reverse". We shoot rays out through each of the pixels in the image plane, and test for intersection with the surface; if the ray intersects, we shade the pixel with the appropriate colour.

i.e. one coordinate in terms of the others: $z = f(x, y)$. Although the explicit form is rarely used to represent planes in Computer Graphics, we used it regularly to represent images when performing image processing operations in Computer Vision. There, the height of the the image at (x, y) is analogous to its intensity $I(x, y)$.

6.4.2 Ray Tracing with Implicit Planes

Ray tracing (sometimes referred to as ray casting[1]) is a commonly used technique to produce 3D Computer Graphics images. As discussed in Chapter 2, we see objects because light reflects off their exterior surfaces and into our eye. Ray tracing models this process in reverse. We trace the path of light from our eye/camera back to the surfaces from which it would reflect. Figure 6.8 illustrates the process; we model the camera's imaging plane as a planar surface in the same 3D space as the 3D object we wish to visualise. We shoot a ray from the camera's focal point (also modelled in 3D) through each pixel modelled in the plane. We then test to see whether the ray intersects with the model of the object or not. If it does, we shade the pixel with an appropriate colour — this is determined as a function of the colour of the surface (refer to earlier discussions of the Phong Illumination Model in OpenGL).

We commonly model objects in a piecewise fashion using small triangular (i.e. planar) surface patches. Typically these are represented using the implicit form of a plane. This is because it is easy to intersect the light ray (modelled as a parametric line) with an implicit plane. Consider a ray modelled using $\underline{p}(s) = \underline{x_0} + s\underline{x_1}$. The plane is modelled with $(\underline{p} - \underline{c}) \circ \underline{\hat{n}} = 0$. We can find the intersection of the light ray and planar surface patch by observing that at the point of intersection:

$$((\underline{x_0} + s\underline{x_1}) - \underline{c}) \circ \underline{\hat{n}} = 0 \tag{6.37}$$

[1]There is a difference between ray casting and ray tracing (ray tracers trace reflected rays off objects), but at the level of detail described in this course there is no distinction.

which, observing that the dot product distributes, we can rearrange for s:

$$s = \frac{(\underline{c} - \underline{x_0}) \circ \underline{\hat{n}}}{\underline{x_1} \circ \underline{\hat{n}}} \tag{6.38}$$

which we evaluate, and then substitute for s into $\underline{p}(s) = \underline{x_0} + s\underline{x_1}$ yielding the point of intersection. Note that if the plane is aligned exactly side-on to the ray direction $\underline{x_1}$ then $\underline{x_1} \circ \underline{\hat{n}} = 0$ and the point of intersection is undefined.

Finally, observe that we have intersected the ray with an infinite plane; when really we are only interested if we have intersected with the finite, triangular region of the plane used to model the patch of the object. We need to perform a containment test, to see if the point on the plane lies within the triangular region.

If we write the 3 corners of the triangular patch as 3D points $\underline{P_1}, \underline{P_2}, \underline{P_3}$ then we can derive vectors between those points: $\underline{V_1} = \underline{P_2} - \underline{P_1}, \underline{V_2} = \underline{P_3} - \underline{P_2}, \underline{V_3} = \underline{P_1} - \underline{P_3}$. If our point of intersection with the surface is $\underline{q}$ then that point will be inside the triangle if the following equation holds:

$$\frac{\underline{q} - \underline{P_1}}{|\underline{q} - \underline{P_1}|} \times \frac{\underline{V_1}}{|\underline{V_1}|} = \frac{\underline{q} - \underline{P_2}}{|\underline{q} - \underline{P_2}|} \times \frac{\underline{V_2}}{|\underline{V_2}|} = \frac{\underline{q} - \underline{P_3}}{|\underline{q} - \underline{P_3}|} \times \frac{\underline{V_3}}{|\underline{V_3}|} \tag{6.39}$$

This works up to $\underline{P_i}$ for any i-sided convex polygon.

Texture mapping

During our discussion of OpenGL we explored texture mapping, and how planes may be textured with images by defining a 2D coordinate system (parameterisation) over the plane.

We can set up a 2D parameterisation (u, v) over the patch surface (perhaps using two basis vectors $\underline{V_1}$ and $\underline{V_2}$ defined by the triangle's sides, as above). By projecting our point of intersecting into this basis, we obtain 2D coordinates of the point of intersection which can be used to reference pixels in an image (i.e. a frame-buffer). When we colour the pixel on the image plane during ray tracing, we can use the colour in the image at pixel (u, v). This gives the impression of the image being affixed to the 3D surface we are ray tracing. Full details are beyond the scope of this course, but this demonstrates both the implicit and parametric forms of plane being applied to different aspects of the ray tracing process.

Ray tracing/casting vs. Perspective matrix

In Chapter 3 we saw that 3D shapes could be visualised by modelling a set of 3D points. Those 3D points could be manipulated in space and projected to 2D via matrix transformations. Associated points (e.g. vertices of a cube) could then be joined with 2D lines to create a "wire-frame" visualisation. In Chapter 4, and accompanying lecture slides, we saw how OpenGL implements this approach to create graphics.

In this Chapter we have seen an alternative way of producing visualisations of seemingly 'solid' 3D objects. Objects are modelled using 3D implicit descriptions of surfaces (the locations of which may still be manipulated via matrix transformations on the points that define them). These surfaces are intersected with simulated rays of light in a "ray tracing"

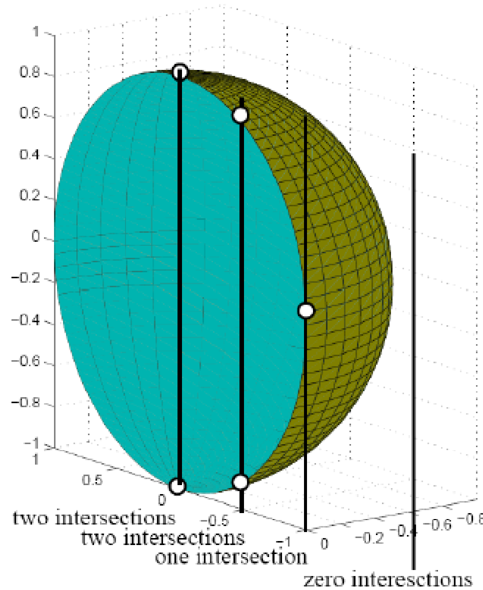


Figure 6.9: Rearranging the implicit form of a sphere into an explicit form prevents modelling of the entire spherical surface; the positive root for $z^2 = x^2 + y^2$ produces a half-sphere in $z > 0$.

process. There is no perspective matrix transformation involved. However we can see the geometry of perspective projection at work through the modelling of a camera using a focal point and plane in 3D – and the intersection of light rays with objects.

6.4.3 Curved surfaces

As you might suspect, many curved surfaces can also be modelled in explicit, parametric and implicit forms. The simplest curved surface is arguably a sphere. An implicit equation for a sphere is $x^2 + y^2 + z^2 = r^2$. Or, to be consistent with the form $f(x, y, z) = 0$:

$$x^2 + y^2 + z^2 - r^2 = 0 \tag{6.40}$$

Additionally, the left-hand side of the equation is positive if a point is outside the sphere, and negative if it is inside the shell of the sphere's perimeter. Introducing an (inconsequential) sign change $x^2 + y^2 - z^2 - r^2 = 0$ we might suspect that a sphere could be modelled explicitly as $z = f(x, y)$ via:

$$z = \sqrt{x^2 + y^2 - r^2} \tag{6.41}$$

However this would give us only half a sphere; again we see the weakness of the explicit form as being unable to represent multi-valued functions i.e. $z = f(x, y)$. In this case, by convention, we take the positive root and so model a half-sphere in the positive z domain (Figure 6.9).

We can also form a parametric representation of a sphere. We commonly use such a representation everyday e.g. in GPS navigation when we talk of longitude (east-west, θ) and

latitude (north-south, ϕ) on the Earth. We specify a 3D point on a unit sphere's surface via the 2D coordinate space (θ, ϕ):

$$\begin{aligned} x(\theta,\phi) &= \cos\theta\cos\phi \\ y(\theta,\phi) &= \sin\theta\cos\phi \\ z(\theta,\phi) &= \sin\phi \end{aligned} \tag{6.42}$$

If we were able to ray trace such a sphere, we could use such a coordinate system to texture map a 2D image (say a spherical projection of a World map) onto a sphere using a similar approach to planar texture mapping (where the plane was parameterised by (u, v) we are instead parameterised in 2D via θ, ϕ).

Intersecting a ray with a sphere

Finding the intersection of a ray with a sphere is slightly more complex than an infinite plane. As before we need to find the s at which we hit the surface. As the sphere is finite in extent, there may be no solutions. If we do intersect it then there we would typically two solutions as the ray enters and exits the sphere; although it is possible that the ray just grazes the sphere producing one solution. Given the previous implicit equation for a sphere (eq.6.40) clearly we are solving a quadratic with up to 2 real solutions.

First we define the sphere using an implicit representation; points on the sphere are described by $|\underline{c}+\underline{q}|-r=0$ where $\underline{c}$ is the centre of the sphere, $\underline{c}+\underline{q}$ is any point on the sphere's surface, and r is the radius of the sphere. As before we define the light ray as $\underline{p}(s) = \underline{x_0} + s\underline{x_1}$; here ensuring that x1 is a unit vector (this does not affect the derivation but simplifies it very slightly). Figure 6.8 (right) illustrates this geometry. At the point of intersection:

$$\underline{c} + \underline{q} = \underline{x_0} + s\underline{x_1} \tag{6.43}$$

which we rearrange to:

$$\underline{q} = (\underline{x_0} + \underline{c}) + s\underline{x_1} \tag{6.44}$$

and take the dot product of each side with itself (noting that $|\underline{q}| = r$, and that $\underline{x_1} \circ \underline{x_1} = 1$):

$$r^2 = (\underline{x_0} - \underline{c}) \circ (\underline{x_0} - \underline{c}) + 2s\underline{x_1} \circ (\underline{x_0} - \underline{c}) + s^2 \tag{6.45}$$

So we can solve the following quadratic for s in the standard way; any real roots are intersections with the sphere:

$$s^2 + s(2\underline{x_1} \circ (\underline{x_0} + \underline{c})) + (\underline{x_0} - \underline{c}) \circ (\underline{x_0} - \underline{c}) - r^2 = 0 \tag{6.46}$$

6.4.4 Bi-cubic surface patches

In subsection 6.1.2 we saw how cubic curves can be used to model shapes in piecewise fashion. The idea generalises to model small pieces of surface referred to as bi-cubic surface patches. We can model complex objects by joining together such patches, rather like a three dimensional jigsaw puzzle. Recall that a curve can be specified using:

$$\underline{p}(s) = \left[\ \underline{g_1} \quad \underline{g_2} \quad \underline{g_3} \quad \underline{g_4}\ \right] \underline{\underline{M}}\,\underline{Q}(s) \tag{6.47}$$

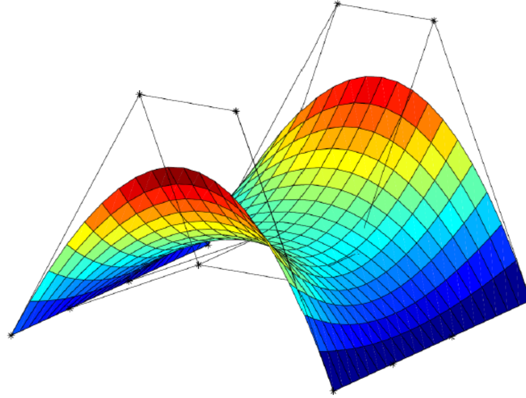


Figure 6.10: Illustration of a Bézier bi-cubic surface patch. The 16 control points $\underline{h_{i,j}}$ of the surface are indicated in black and determine the geometry of the surface.

where $\underline{g_{1..4}}$ are the geometry vectors that define the shape of the curve. But now consider that each of those four vectors could itself be a point on four independent parametric cubic curves respectively. That is, for $i = [1, 4]$:

$$\underline{g_i}(r) = \underline{\underline{H_i}}\,\underline{\underline{M}}\,\underline{\underline{Q}}(r) \tag{6.48}$$

where $\underline{\underline{H_i}} = [\underline{h_{i,1}}\ \underline{h_{i,2}}\ \underline{h_{i,3}}\ \underline{h_{i,4}}]$ — introducing the notation $\underline{h_{i,j}}$ to denote the jth control point on the $i^t h$ curve. So the control points for eq.(6.47) are defined by four independent curves, commonly parameterised by r. A position on the surface is thus defined in 2D parameter space (s, r); hence the term **bi-cubic surface patch**. Recall that these parameters are typically defined in range $s, r = [0, 1]$.

We can combine eq.(6.47) and eq.(6.48) to express the surface in a single equation. We first rewrite the equation as:

$$\underline{g_i}(r)^T = \underline{\underline{Q}}^T \underline{\underline{M}}^T \left[\begin{array}{cccc} \underline{h_{i,1}} & \underline{h_{i,2}} & \underline{h_{i,3}} & \underline{h_{i,4}} \end{array} \right]^T \tag{6.49}$$

And then with some minor abuse of notation (to avoid introducing tensor notation) we can write:

$$\underline{p}(s, r) = \underline{\underline{Q}}(r)^T \underline{\underline{M}}^T \left[\begin{array}{cccc} \underline{h_{1,1}} & \underline{h_{2,1}} & \underline{h_{3,1}} & \underline{h_{4,1}} \\ \underline{h_{1,2}} & \underline{h_{2,2}} & \underline{h_{3,2}} & \underline{h_{4,2}} \\ \underline{h_{1,3}} & \underline{h_{2,3}} & \underline{h_{3,3}} & \underline{h_{4,3}} \\ \underline{h_{1,4}} & \underline{h_{2,4}} & \underline{h_{3,4}} & \underline{h_{4,4}} \end{array} \right] \underline{\underline{M}}\,\underline{\underline{Q}}(s) \tag{6.50}$$

where each element of the 4×4 matrix $\underline{h_{i,j}}$ is the jth geometry vector of $\underline{g_i}$ from eq.(6.48). If $\underline{\underline{M}}$ was the blending matrix for the Bézier curve family then each $\underline{h_{i,j}}$ would be a point in space. The volume bounded by the convex hull of those points would contain the bi-cubic surface patch (Figure 6.10). This hull can be used to quickly check for potential ray intersections with the patch when ray tracing (although more detailed intersection through solution of a cubic is necessary to find the exact point of intersection; in practice a numerical rather than a closed form approach is used for this).

Consideration of continuity

We can join patches together with C^n continuity, much as with curves. For example we can define adjacent Bézier bi-cubic patches to share control points, to enforce C^0 or C^1 continuity. We can even use the 2D parameterisation over the surface (s, r) to perform texture mapping, as we discussed with planar surfaces. However we would need to take care in reparameterising the surface appropriately beforehand (consider the discussion of arc-length vs. algebraic parameterisation in subsection 6.3; what implications might this have for a texture mapped bi-cubic surface?)

It is possible to use the Hermite blending matrix to define Hermite bi-cubic surface patches; however it is usually impractical to control surfaces in this way. There is little benefit over Bézier patches, through which it is already possible to ensure C^0 and C^1 continuity over surfaces.

It transpires that C^1 continuity is very important when modelling objects. The lighting calculations used to determine the colour of a surface when ray tracing (subsection 6.4.2) is a function of the surface's normal. If the normal changes suddenly at the surface interface (e.g. due to non-conformance to C^1 continuity) then the colour of the surface will also change. This effect is exacerbated by the human eye's sensitivity to sudden intensity change (recall the **Mach Banding effect** discussed during OpenGL lectures) to create highly noticeable and undesirable edge artifacts. This effect can be also observed when modelling objects with triangular patches (planes) that are too large.

Index

active interpretation, 26, 27
additive primaries, 13
affine transformations, 29
algebraic parameterisation, 87
anti-parallel, 5
Application Programmers Interface, 50
approximating curve, 83
arc length, 87
arc-length parameterisation, 87
associative, 7

B-spline, 87
backward mapping, 47
basis set, 5
basis vectors, 5
bi-cubic surface patch, 96
bi-normal, 89
bits, 9
Black, 13
blending matrix, 81
blending plane, 89
brightness, 16

Cartesian Coordinates, 1
Cartesian form, 6
category, 63
Catmull-Rom spline, 85
characteristic polynomial, 59
chromaticity coordinates, 18
classification problems, 63
CLUT, 12
Colour Lookup Table, 12
colour saturation, 16
colour space, 15
column vectors, 1, 8
compound matrix transformations, 30
computing the homography, 44
cone, 13
continuity, 80
control points, 83
convex hull, 84
covariance, 69
covariance matrix, 73
cross product, 3, 4
curvature, 89

decision boundary, 66
decomposing, 58
design matrix, 45
determinant, 8
diagonalised matrix, 58
dichotomiser, 68
digital image warping, 46
distance metric, 64, 66
dot product, 3
double-angle formulae, 25

Eigenmodel, 69
eigenvalue decomposition (EVD), 58
eigenvalues, 58
eigenvectors, 58
Euclidean distance, 66
Euler angles, 35
exemplar data, 63
explicit, 79

feature, 65
feature space, 65–67
field of view, 42
focal length, 42
focal point, 42
forward mapping, 47
frame buffer, 10

gamut, 12
Gaussian distributed, 73
geometry matrix, 81
gimbal, 38
gimbal lock, 38
GLUT library, 50

Grammian matrix, 73

homogeneous coordinates, 29
homography, 44
Hue, Saturation, Luminance (HSL), 20
Hue, Saturation, Value (HSV), 19

ideal primaries, 18
identity matrix, 7
image representation, 9
implicit, 79
intensity, 12
interpolating curve, 83
inverse of a matrix, 8

knots, 83

leading diagonal, 7
linear transformations, 28, 29
luminosity, 16

Mach Banding effect, 97
magnitude, 2
Mahalanobis distance, 69, 71
Manhattan distance, 66
matrix diagonalisation, 61
matrix square root, 62
mega-pixel, 10
model, 63, 77
modelling, 77

Nearest Mean classification, 67
non-commutative, 7
norm, 66
normal, 4, 89
normalised chromaticity coordinates, 18
normalised vector, 3
null category, 63

OpenGL library, 50
orthographic projection, 40, 43
orthonormal, 8

palette, 12
parametric, 79
passive interpretation, 26, 27
Pattern Recognition, 63
pattern recognition, 63
perspective projection, 40
pin-hole camera, 42
pitch, 35
pixel, 9
pixel interpolation, 47
principal axes, 6
principal component analysis (PCA), 62
principal eigenvector, 70
projection, 40
projective transformations, 29
pseudo-colour frame buffer, 11
pseudo-inverse (Moore-Penrose inverse), 75

quaternion, 38

radial-polar form, 6
raster, 9
rectangular matrices, 75
reference frame, 5
resolution, 10
retina, 13
RGB colour cube, 15
rigid body transformations, 23, 28
rod, 13
roll, 35
root reference frame, 5
row vectors, 1

sampling, 9
saturation, 20
scalar, 1
scale factor, 24
scaling matrix, 24
scan-line, 11
secondary colours, 13
shearing matrix, 24
signed distance, 79
Silicon Graphics (SGI), 50
singular value decomposition (SVD), 75
space curves, 77
spline, 84
splines, 84
stride, 12
subtractive primaries, 14
supervised classification, 63
SVD (Singular Value Decomposition), 45

telephoto lens, 42
test data, 64
threshold, 64
torsion, 90

training, 63
training set, 63
transfer function, 87
translation, 28
tri-stimulus experiment, 18

uniformly distributed, 69
unit length, 3

value, 20
vanishing point, 40
variance, 70
vector, 1
vector product, 4
video modes, 11

White, 13
wide-angle lens, 42

xyzzy, 4

yaw, 35