— CM20219 —

Fundamentals of Visual Computing

Course Notes
(preliminary version)

Lecturer:
Dr Christian Richardt

Last updated: 2017-10-06

Summary

Welcome to CM20219 – Fundamentals of Visual Computing.

This is an introductory unit that teaches the essential mathematical and computational foundations for visual computing (which includes computer graphics, image processing and computer vision among other areas), as well as the practical skills to begin developing visual computing software in practice with MATLAB and JavaScript/WebGL programming.

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 5 provides only very introductory material to OpenGL (which will be replaced with WebGL for 2017/2018), and further resources such as online documentation and tutorials or books 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 described within to produce viable visual computing solutions.

Please note:
These notes will be updated throughout the semester, so please remember to check the Moodle page regularly for updated versions of these notes as well as other course material such as slides.

Visual computing demands a working knowledge of linear algebra (matrix manipulation, linear systems, etc.). A suitable background is provided by the first-year unit CM10197 (Analytical mathematics for applications), and Chapter 1 provides a brief revision on some key mathematics. This course also requires a basic working knowledge of the MATLAB and JavaScript languages, which will be introduced in the labs.

The course and notes focus on four key topics:

1. **Images and colours (Chapter 2)** How digital images are represented in a computer. This 'mini'-topic explores different formats for storing images, different ways of representing colours, and key issues that arise.
2. **The Fourier transform (Chapter 3)** — will be completed for 2017/2018
   How converting signals to a different domain – the Fourier domain – enables advanced filtering and analysis operations.
3. **Geometric transformations (Chapter 4)** How to use linear algebra, such as matrix transformations, to manipulate points in space. Focuses heavily on the concept of reference frames and their central role in visual computing.
4. **OpenGL programming (Chapter 5)** — to be replaced with WebGL for 2017/2018
   Discusses how the mathematics on this course can be implemented directly in the C programming language using the OpenGL library.
5. **Geometric modelling (Chapter 6)** This chapter explores how these points can be "joined up" to form curves and surfaces. This allows the modelling of objects and their trajectories.

Acknowledgements This course and the accompanying notes, lab sheets, exams and coursework were originally developed by John Collomosse in 2008/9. The course content and material has since been refined and extended by Matt Brown, Mac Yang, Andrew Chinery and Daniel Beale.

Contents

Chapter 1

Mathematical background

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

1.1 Notation of points and vectors

Much of visual computing involves discussion of points in 2D or 3D. Usually, we write such points as **Cartesian coordinates**, e.g. $\mathbf{p} = [x, y]^\top$ or $\mathbf{q} = [x, y, z]^\top$. Point coordinates are therefore **vector** quantities, as opposed to a single number such as '3', which we call a **scalar** quantity. In these notes, we write vectors with bold lower-case letters ($\mathbf{a}$), and matrices as bold upper-case letters ($\mathbf{A}$).

The superscript $[...]^\top$ denotes the **transposition** of a vector, so points $\mathbf{p}$ and $\mathbf{q}$ 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 [Foley et al., 1990; Hughes et al., 2013]. Bear in mind that you can convert equations between the two forms using transposition. Suppose we have a 2×2 matrix $\mathbf{M}$ acting on the 2D point represented by column vector $\mathbf{p}$. We would write this as $\mathbf{Mp}$.

If $\mathbf{p}$ was transposed into a row vector $\mathbf{p}' = \mathbf{p}^\top$, we could write the above transformation $\mathbf{p}'\mathbf{M}^\top$. So to convert between the forms (e.g. from row to column form when reading the course texts), remember that:

$$\mathbf{Mp} = (\mathbf{p}^\top \mathbf{M}^\top)^\top. \tag{1.1}$$

For a reminder on the matrix transposition please see Section 1.6.5.

1.2 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.

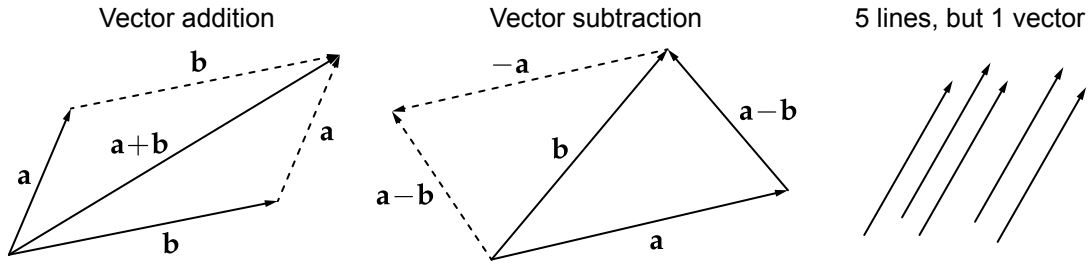

Figure 1.1: Illustrating vector addition (**left**) and subtraction (**middle**). **Right:** Vectors have direction and magnitude; lines (sometimes called 'rays') are vectors plus a starting point.

1.2.1 Vector addition

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

$$\mathbf{a} + \mathbf{b} = [u, v]^\top + [s, t]^\top = [u + s, v + t]^\top. \tag{1.2}$$

1.2.2 Vector subtraction

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

$$-\mathbf{b} = [-s, -t]^\top \tag{1.3}$$

$$\mathbf{a} - \mathbf{b} = \mathbf{a} + (-\mathbf{b}) = [u - s, v - t]^\top. \tag{1.4}$$

1.2.3 Vector scaling

If we wish to increase or reduce a vector quantity by a scale factor s, then we multiply each element in the vector by s:

$$s\mathbf{a} = [su, sv]^\top. \tag{1.5}$$

1.2.4 Vector magnitude

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

$$\|\mathbf{a}\| = \sqrt{u^2 + v^2}. \tag{1.6}$$

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 $\mathbf{a}$ from the origin is the hypotenuse of a right-angled triangle. If we have an n-dimensional vector $\mathbf{q} = [q_1, q_2, q_3, \ldots, q_n]^\top$, then the definition of vector magnitude generalises to

$$\|\mathbf{q}\| = \sqrt{q_1^2 + q_2^2 + q_3^2 + \ldots + q_n^2} = \sqrt{\sum_{i=1}^{n} q_i^2}. \tag{1.7}$$

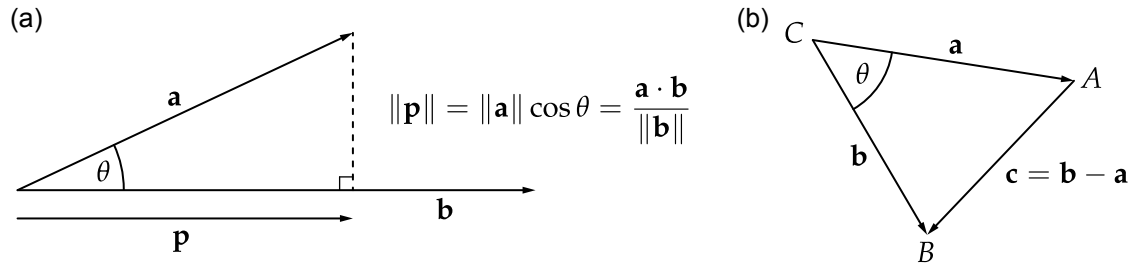

Figure 1.2: **(a)** Demonstrating how the dot product can be used to measure the component of one vector in the direction of another (i.e. a projection, shown here as $\mathbf{p}$). **(b)** The geometry used to prove $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$ via the Law of Cosines in Equation 1.15.

1.2.5 Vector normalisation

We can normalise a vector $\mathbf{a}$ by dividing it by its magnitude $\|\mathbf{a}\|$:

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}. \tag{1.8}$$

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

1.3 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').

1.3.1 Dot product

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

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 + \ldots + a_n b_n \tag{1.9}$$

$$= \sum_{i=1}^{n} a_i b_i. \tag{1.10}$$

The dot product is both symmetric and positive definite. It gives us a *scalar value* that has three important uses:

1. We can compute the square of the magnitude of a vector by taking the dot product of that vector and itself:

$$\mathbf{a} \cdot \mathbf{a} = a_1 a_1 + a_2 a_2 + ... + a_n a_n \tag{1.11}$$

$$= \sum_{i=1}^{n} a_i^2 \tag{1.12}$$

$$= \|\mathbf{a}\|^2 \tag{1.13}$$

2. We can more generally compute $\mathbf{a} \cdot \mathbf{b}$, the magnitude of one vector $\mathbf{a}$ in the direction of another $\mathbf{b}$, i.e. projecting one vector onto another. Figure 1.2(a) illustrates how a simple rearrangement of Equation 1.14 can achieve this.
3. 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:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta. \tag{1.14}$$

A proof follows from the "law of cosines" – a general form of Pythagoras' theorem. Consider triangle ABC in Figure 1.2(b) with respect to Equation 1.14. Side $\overrightarrow{CA}$ is analogous to vector $\mathbf{a}$, and side $\overrightarrow{CB}$ analogous to vector $\mathbf{b}$. θ is the angle between $\overrightarrow{CA}$ and $\overrightarrow{CB}$, and so also vectors $\mathbf{a}$ and $\mathbf{b}$. The law of cosines states that

$$\|\mathbf{c}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2\|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.15}$$

$$\mathbf{c} \cdot \mathbf{c} = \mathbf{a} \cdot \mathbf{a} + \mathbf{b} \cdot \mathbf{b} - 2\|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta. \tag{1.16}$$

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

$$(\mathbf{a} - \mathbf{b}) \cdot (\mathbf{a} - \mathbf{b}) = \mathbf{a} \cdot \mathbf{a} + \mathbf{b} \cdot \mathbf{b} - 2\|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.17}$$

$$\mathbf{a} \cdot \mathbf{a} - 2(\mathbf{a} \cdot \mathbf{b}) + \mathbf{b} \cdot \mathbf{b} = \mathbf{a} \cdot \mathbf{a} + \mathbf{b} \cdot \mathbf{b} - 2\|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.18}$$

$$-2(\mathbf{a} \cdot \mathbf{b}) = -2\|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.19}$$

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.20}$$

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

1.3.2 Cross product

Taking the **cross product** (or "**vector product**") of two 3D vectors returns us a vector orthogonal to those two vectors (that respects the right-hand rule). Given two vectors $\mathbf{a} = [a_x, a_y, a_z]^\top$ and $\mathbf{B} = [b_x, b_y, b_z]^\top$, the cross product $\mathbf{a} \times \mathbf{b}$ is defined as:

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}. \tag{1.21}$$

An important 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 $\mathbf{n}$ in the following relationship (care: note the normalisation):

$$\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin\theta \, \hat{\mathbf{n}}. \tag{1.22}$$

A proof is beyond the requirements of this course.

1.4 Reference frames

When we write down a point in Cartesian coordinates, for example $\mathbf{p} = [3, 2]^\top$, we interpret that notation as "the point $\mathbf{p}$ is 3 units from the origin travelling in the positive direction of the x-axis,

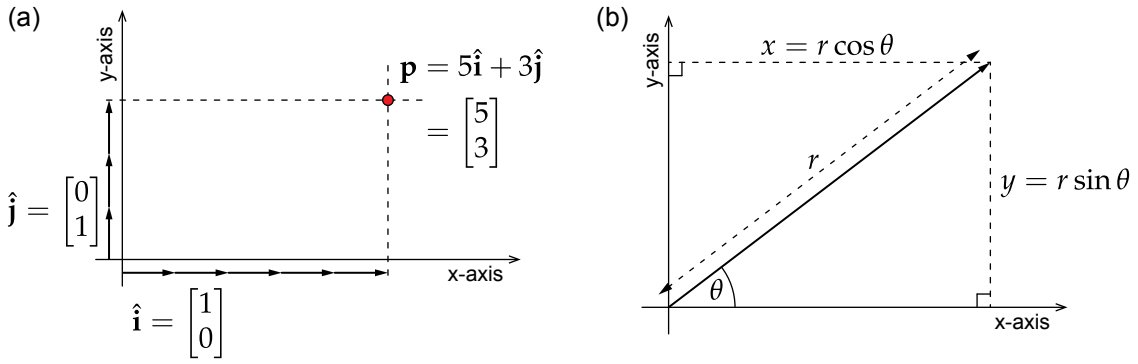

Figure 1.3: **(a)** Cartesian coordinates are defined with respect to a reference frame. The reference frame is defined by basis vectors $\hat{\mathbf{i}}$ and $\hat{\mathbf{j}}$ (one per axis) that specify how 'far' and in what direction the units of each coordinate will take us. **(b)** Converting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

and 2 units from the origin travelling in the positive direction of the y-axis". We can write this more generally and succinctly as:

$$\mathbf{p} = x\hat{\mathbf{i}} + y\hat{\mathbf{j}}, \tag{1.23}$$

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

Commonly when working with 2D Cartesian coordinates, we work in the reference frame defined by $\hat{\mathbf{i}} = [1, 0]^\top$, $\hat{\mathbf{j}} = [0, 1]^\top$. However, other choices of basis vectors are equally valid, so long as the basis vectors are neither parallel nor **anti-parallel** (i.e. do not point in the same or opposite direction). We refer to our 'standard' reference frame ($\hat{\mathbf{i}} = [1, 0]^\top$, $\hat{\mathbf{j}} = [0, 1]^\top$) as the **root reference frame**, because we define the basis vectors of 'non-standard' reference frames with respect to it.

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

1.5 Cartesian vs. polar coordinates

We have so far recapped about Cartesian coordinate systems. These describe vectors in terms of distance along each of the **principal axes** (e.g. x and y) of the space. This **Cartesian coordinate system** 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 a **polar coordinate system**. In the case of 2D point locations, we describe the point in terms of: (a) its distance, or radius, r from the origin, and (b) the angle θ between the positive direction of the x-axis, and the line from the origin to the point (see Figure 1.3b).

To convert from Cartesian coordinates $[x, y]^\top$ to polar coordinates (r, φ), we consider a right-angled triangle with sides x and y (Figure 1.3b). We can use Pythagoras' theorem to determine the length

of the hypotenuse r, and some basic trigonometry to reveal that $\tan\theta = y/x$, and thus:

$$r = \sqrt{x^2 + y^2} \tag{1.24}$$

$$\theta = \arctan\left(\frac{y}{x}\right). \tag{1.25}$$

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

$$x = r\cos\theta \tag{1.26}$$

$$y = r\sin\theta. \tag{1.27}$$

1.6 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×width". In this subsection, we will use two 2×2 matrices for our examples:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \tag{1.28}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}. \tag{1.29}$$

Observe that the notation for addressing an individual element of a matrix $\mathbf{X}$ is $x_{\text{row,column}}$ (and the comma is often omitted).

1.6.1 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:

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} (a_{11} + b_{11}) & (a_{12} + b_{12}) \\ (a_{21} + b_{21}) & (a_{22} + b_{22}) \end{bmatrix}. \tag{1.30}$$

This is identical to vector addition.

1.6.2 Matrix scaling

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

$$s\mathbf{A} = \begin{bmatrix} sa_{11} & sa_{12} \\ sa_{21} & sa_{22} \end{bmatrix}. \tag{1.31}$$

1.6.3 Matrix multiplication

As we will see in Chapter 4, matrix multiplication is a cornerstone of many useful geometric transformations. You should ensure that you are familiar with this operation.

$$\mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} (a_{11}b_{11} + a_{12}b_{21}) & (a_{11}b_{12} + a_{12}b_{22}) \\ (a_{21}b_{11} + a_{22}b_{21}) & (a_{21}b_{12} + a_{22}b_{22}) \end{bmatrix}. \tag{1.32}$$

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

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{1.33}$$

Not all matrices are compatible for multiplication. In the above system, $\mathbf{A}$ must have as many columns (n) as $\mathbf{B}$ has rows. The resulting matrix $\mathbf{C}$ then is of size $m \times o$.

Furthermore, matrix multiplication is **non-commutative**, which means that $\mathbf{BA} \neq \mathbf{AB}$, in general. Given Equation 1.32, you might like to write out the multiplication for $\mathbf{BA}$ to satisfy yourself of this.

Finally, matrix multiplication is **associative**, that is

$$\mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}). \tag{1.34}$$

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].

1.6.4 Matrix inverse and the identity matrix

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

$$\mathbf{IA} = \mathbf{A}. \tag{1.35}$$

The identity matrix has zeroes everywhere except on the **diagonal**, which is set to 1. So, for example, the 2×2 identity matrix is:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{1.36}$$

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

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}. \tag{1.37}$$

As we will see in Chapter 4, this gives rise to the notion of reversing a geometric transformation. However, some geometric transformations (and matrices) cannot be inverted. Specifically, a matrix must be square and non-singular, i.e. have a non-zero **determinant**, in order to be inverted by conventional means.

1.6.5 Matrix transposition

Matrix transposition, just like vector transposition, is simply a matter of swapping the rows and columns of a matrix (reflection along the diagonal). As such, every matrix has a transpose. The transpose of $\mathbf{A}$ is written $\mathbf{A}^\top$:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^\top = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}. \tag{1.38}$$

[1] Finding an optimal permutation of multiplication order is a (solved) interesting optimization problem, but falls outside the scope of this course (see Cormen et al.'s *Introduction to Algorithms* [2009])

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. $[a_{11}, a_{21}]^\top$ and $[a_{12}, a_{22}]^\top$. These are sometimes called **column vectors** of the matrix. If the magnitudes of all these vectors are one, and if each pair of vectors is orthogonal to each other (i.e. their dot product is zero), then the matrix is orthonormal. Examples of orthonormal matrices are the identity matrix, and the rotation matrix that we will meet in Chapter 4.

Chapter 2

Images and colours

Visual computing is principally concerned with the generation and understanding of images, with wide-ranging applications from entertainment to scientific visualisation. In this chapter, we begin our exploration of visual computing 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.

2.1 Digital images

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 and continuous. 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 into 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 a camera (the machine equivalent of an eye). Since we do not have infinite storage (bits), it follows that we must convert that continuous analogue signal into a more limited digital form. We call this conversion process **sampling**. Sampling theory is an important part of computer science, underpinning the theory behind both image capture and manipulation – we return to the topic briefly in Chapter 5 (and in detail in a later lecture).

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

2.1.1 Raster image representations

Digital images are represented using a regular grid that we call a 'raster', and digital images are thus also sometimes called '**raster images**'. Each grid cell is a **pixel** (short for '*picture element*'). The pixel is the atomic unit of the image; it has a single colour representing a discrete sample of light, for example from a captured image, but is usually visualised as a solidly coloured square[1].

[1] Pixels don't need to be square, they can also be rectangular, which is used for some television video formats such as PAL and NTSC. For more detail, see https://en.wikipedia.org/wiki/Pixel_aspect_ratio.

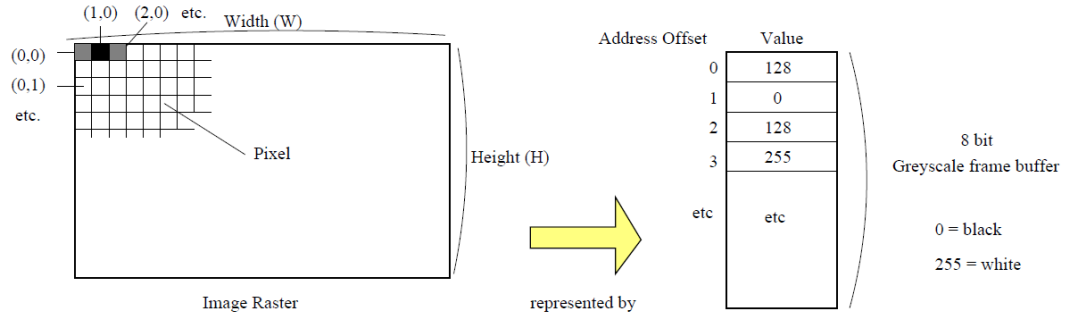

Figure 2.1: Rasters are used to represent digital images. Modern displays use a rectangular raster, comprised of $W \times H$ pixels. The raster illustrated here contains a greyscale image; its contents are represented in memory by a greyscale framebuffer. The values stored in the framebuffer record the intensities of the pixels on a discrete scale, where 0 is black and 255 is white for 8-bit images.

In most implementations, rasters take the form of a rectilinear grid, often containing thousands or even millions of pixels (see Figure 2.1). The raster provides an orthogonal two-dimensional basis with which to specify pixel coordinates. By convention, pixel coordinates are zero-indexed and the origin is located at the **top-left** of the image. Therefore, the pixel $(w-1, h-1)$ is located at the bottom-right corner of a raster of width w pixels and height h pixels. As an aside, some applications make use of hexagonal pixels instead[2], however we will not consider these on the course.

The dimensions of an image (in pixels) is referred to as the image's **spatial resolution**. Most modern desktop displays are capable of natively displaying images with a resolutions of 1920×1080 pixels (also known as **Full HD** or **1080p**), which is about 2 million pixels or 2 **megapixels**. Even inexpensive modern cameras and scanners are now capable of capturing images at resolutions of several megapixels. In general, the greater the resolution, the greater the level of spatial detail an image can represent.

2.1.2 Hardware framebuffers

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 **framebuffer**. 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[3].

Note that the term 'framebuffer', as originally defined, strictly refers to the area of memory reserved for direct manipulation of the currently displayed image. In the early days of computer 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 'framebuffer' has fallen into informal use to describe any piece of storage that represents an image, particularly when residing on a **graphics card**.

There are a number of popular formats (i.e. ways of encoding pixels) within a framebuffer. This is partly because each format has its own advantages, and partly for reasons of backward compatibility

[2] 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.

[3] The framebuffer is usually 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.

with older systems (especially on the PC). Older video hardware can be switched between different **video modes**, each of which encodes the framebuffer in a different way. We will describe three common framebuffer formats in the subsequent sections: the greyscale, indexed and true-colour formats. You will likely encounter all three in your work at some stage.

2.1.3 Greyscale framebuffers

Arguably the simplest form of framebuffer is the greyscale framebuffer, which is often mistakenly called 'black and white' or 'monochrome'. Greyscale buffers encode pixels using various discrete 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. Consequently, an image of size $w \times h$ pixels requires $w \times h$ bytes of memory for its framebuffer. Black is usually represented by the value 0, white by 255, and a mid-level grey pixel has the value 128; the brightness of a pixel is thus directly correlated with its value.

The framebuffer 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 from left to right within a row of pixels (also known as a **scanline**), and then top to bottom across the rows (see Figure 2.1), which is known as **scanline order**. The value (grey level) of pixel (1, 0) is thus stored in the second byte of memory, and so on for the subsequent pixels in that row. The first pixel in row y, i.e. the pixel at (0, y), is then stored at offset wy. The pixel (x, y) would therefore be stored at buffer offset $wy + x$ from the start of the framebuffer.

2.1.4 Indexed framebuffers

The **indexed framebuffer** allows very efficient representation of colour images. The storage scheme is in principle identical to the greyscale framebuffer. 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**) or **palette**, which is essentially a map (index $\mapsto$ colour), i.e. a table indexed with an integer key $[0, 255]$ storing a value that represents colour. As we will discuss in greater detail shortly (Section 2.2), 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 triplet (r, g, b) denoting the quantity (**intensity**) of red, green and blue light in the mix. Each element of the triplet has 8 bits and thus a has range of $[0, 255]$ in common implementations.

The earliest colour displays employed indexed framebuffers. 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 an indexed framebuffer is that only a limited number of colours may be displayed at any one time (i.e. 256 colours), even though the colour range (we say **gamut**) of the display supports $2^8 \times 2^8 \times 2^8 = 2^{24} =$ 16,777,216 different colours.

Indexed framebuffers can still be found in many common platforms, for example on Windows and X Windows (for convenience, backward compatibility etc.), and in resource-constrained computing domains such as low-spec games consoles. Perhaps the most common usage of indexed framebuffers nowadays are images stored in the **Graphics Interchange Format** (**GIF**), where each frame of an animation can have a different 256-colour palette.

Some low-budget (in terms of CPU cycles) animation effects can be produced using indexed framebuffers. Consider an image filled with an expanses of colour index 1 (we might set this index to

'blue', to create a blue ocean). We could sprinkle consecutive runs of pixels with indeces '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, hence changing the displayed colours and causing the waves to animate/ripple (but without the CPU and memory overhead of rewriting the entire framebuffer). Effects like this were regularly used in many 1980s and early 1990s computer games, where computational expense prohibited updating the framebuffer directly for incidental animations.

2.1.5 Colour framebuffers

The **colour framebuffer** also represents colour images, but does not use a CLUT. The RGB colour value for each pixel is stored directly within the framebuffer. So, if we use 8 bits each to represent the red, green and blue components, we will require 24 bits (or 3 bytes) of storage per pixel.

As with the other types of framebuffers, 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 locations 3, 4 and 5, and so on. In general, pixel (x, y) would be stored at offset

$$s = 3w, \tag{2.1}$$

$$a = sy + 3x, \tag{2.2}$$

where s is referred to as the **stride** of the image.

The advantages of the colour buffer complement the disadvantages of the indexed 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 indexed buffer. The image also take longer to update (3 times as many memory writes), which should be taken into consideration on resource-constrained platforms (e.g. if writing a video codec on a mobile phone).

Alternative image formats

The colour framebuffer described above uses 24 bits to represent RGB colours. The usual convention is to write the R, G and B values in order for each pixel. However, many image formats, such as Windows Bitmaps, write colours in the 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 'RGB24' (24-bit RGB) or 'RGB888' (8-bit RGB colour channels), and 'BGR24' or 'BGR888', respectively.

Remember that a key characteristic of a specific processor design is its **word size** – the number of bits a processor stores and processes natively (e.g. in a register), for example 32 bits for x86 or 64 bits for x64. Directly modifying a single byte in memory, for example to modify the colour of a pixel, is often not possible, so a processor would need to load the whole word containing the byte to be changed, modify the byte, and write back the whole word. This can be a huge overhead, depending on a particular application. The RGB32 (or BGR32) image format avoids this by storing each RGB colour in a separate word (and usually leaving the last 8 bits empty). In this way, each pixel can be read or written directly without interfering with neighbouring pixels. The remaining 8 bits can also be used for the alpha channel to encode opacity, which gives rise to format names like 'RGBA32' or 'RGBA8888' and so on, depending on the order of colour channels.

The word size also affects the stride of an image. The minimum stride for an 8-bit RGB image is $3w$ bytes, but this might lead to a scanline starting in the middle of a word, which is a bad idea as discussed above. For this reason, the stride is often rounded up to the nearest multiple of the word size, leaving some padding at the end of each scanline, but ensuring that each scanline starts at the

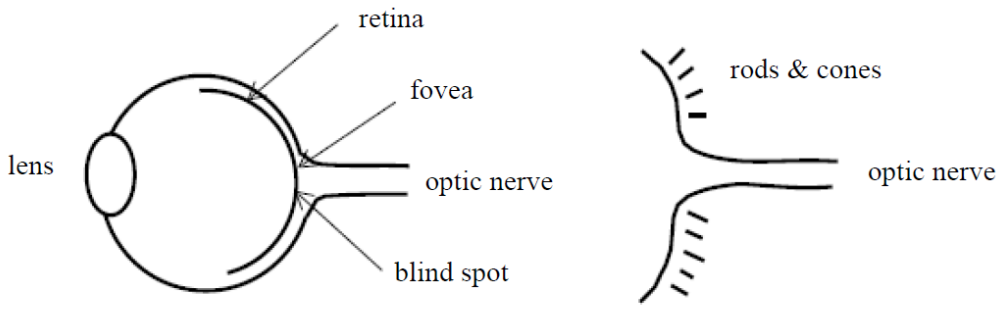

Figure 2.2: In the human eye, light is focused through an elastic lens onto a light sensitive patch (the retina). Special receptor cells (rods and cones) in the retina convert light into electrical impulses that travel to the brain via the optic nerve. The site where the nerve exits the eye contains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

beginning of a word. Even images of the same resolution and colour depth can thus have different strides, which can be a source of confusion.

2.2 Colours

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 cells called **rods** and **cones** that are sensitive to light and send electrical impulses to our brain, which 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 wavelengths of 400–700 nm (violet to red). Figure 2.3 (left) sketches the response of the three cone types to wavelengths in this band. The peaks are located close to the colours red, green and blue[4]. 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 '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, plasma, projectors) operate. Computers and displays we represent pixel colours using values for red, green and blue (RGB triplets), and the video hardware uses these values to generate the appropriate amount of red, green and blue light (Section 2.1.5).

Red, green and blue are called "**additive primaries**", because we can 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); and 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).

[4]Other animals and some humans have cones that peak at different colours. Bees, for example can see in ultra-violet as one of their cone cells peaks at ultra-violet. This allows bees to easily spot the centres of flowers, which are often marked with ultra-violet patterns invisible to humans.

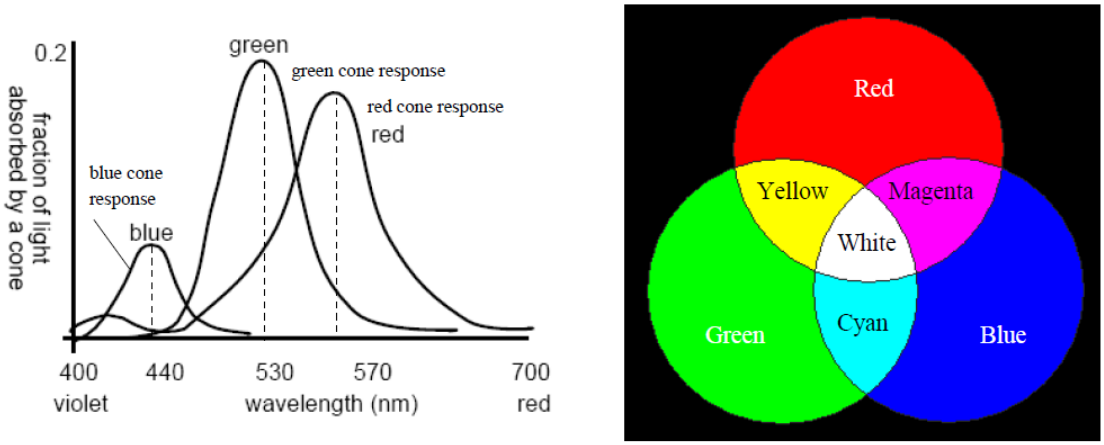

Figure 2.3: Left: Sketching the response of each of the three cones to differing wavelengths of light. Peaks are observed around the colours we refer to as red, green and blue. Observe the secondary response of the red cone to blue light. Right: The RGB additive primary colour wheel, showing how light is mixed to produce different colours. The combination of all 3 primaries results in white.

2.2.1 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 of 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 aligns with our experience mixing paints as a child: yellow and cyan make green. Figure 2.3 (right) illustrates this diagrammatically. [**TODO:** Add CMY colour mixing figure]

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. This is how all printing technology works: paper reflects light across all wavelengths, resulting in a **white** colour, and the appropriate mixture of the subtractive primaries then subtract parts of the colour spectrum to achieve a desired colour.

Recap **RGB** (red, green, blue) are the additive primaries, and **CMY** (cyan, magenta, yellow) the subtractive primaries. They are used, respectively, to mix light (in displays and projectors), and to mix ink or paint (when printing). Returning to our original observation, CMY are the colours approximated by the child when mixing red, yellow and blue paints (‘blue’ is easier to teach than ‘cyan’, similarly

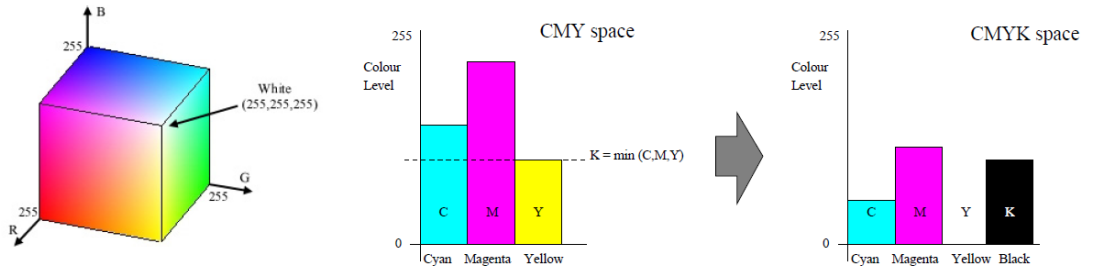

Figure 2.4: Left: The RGB colour cube; colours are represented as points in a 3D space, each axis of which is an additive primary. Right: Conversion from CMY to CMYK. The quantity of black (C, M and Y ink mix) contained within a colour is substituted with pure black (K) ink. This saves ink and improves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

'red' for 'magenta').

2.2.2 The RGB and CMYK colour spaces

We have seen that displays represent colour using an RGB triplet. 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 $(0, 0, 0)$, white is located at $(255, 255, 255)$, and shades of grey are located at (n, n, n), i.e. on the diagonal between black and white.

We have also seen that processes that deposit pigments, such as printing, are more appropriately described using colours in the CMY space. This space is also shaped like a cube.

We observed earlier that to print black requires mixing of all three subtractive primaries (CMY). Printer ink can be expensive, and black is a very common colour in printed documents (think of all the text). 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** ('K' for 'key') 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') using

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

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 K ink for multiple CMY inks.

You may be wondering if an "RGB plus white" space might exist for the additive primaries. Indeed, some devices such as projectors do feature a fourth colour channel for white (in addition to RGB) that works in a manner analogous to black in the CMYK space. The amount of white in a colour is linked to the definition of **saturation**, which we will return to in Section 2.2.6.

Figure 2.5: Greyscale conversion. Left: The human eye's combined response (weighted sum of cone responses) to light of different wavelengths but constant intensity (*luminous efficiency function*). The perceived brightness of the light changes as a function of wavelength. Right: A painting ("Impression, Sunrise" by Claude Monet, 1872) in colour and greyscale. Note the isoluminant colours used to paint the sun against the sky, which make the sun disappear in greyscale.

2.2.3 Greyscale conversion

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

Referring back to Figure 2.3, we see that each type of cone responds with varying strength to each wavelength of light. If we combine the responses of the three types of cones, we would obtain a curve such as Figure 2.5 (left). The curve indicates the *perceived* **brightness** of each light wavelength (monochromatic colour), given a light source of constant energy output.

By experimenting with the human visual system, researchers have derived the following equation to *approximate* this response, which computes the **luma** l for a given RGB colour (r, g, b) using

$$l(r, g, b) = 0.2126r + 0.7152g + 0.0722b. \tag{2.4}$$

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

```
img = im2double(imread('image.jpg'));
r = img(:,:,1);
g = img(:,:,2);
b = img(:,:,3);
y = 0.2126 * r + 0.7152 * g + 0.0722 * b;
imshow(y);
```

This code was applied to the source image in Figure 2.5. This figure gives clear illustration that visually distinct colours, such as the sun and sky in the painting, can map to similar greyscale values. Such colours are called "**isoluminant**" as they have the same brightness (or *luminance*).

2.2.4 Can any colour be represented in RGB space?

Yes and no. Any visible colour can be represented as an RGB triplet, but not all colours can be created by mixing RGB light sources. The reason is that representing some colours would require a negative amount of red, green or blue light, which is not physically possible. This can be shown using the **tristimulus experiment** – a manual colour-matching exercise performed under controlled conditions as follows.

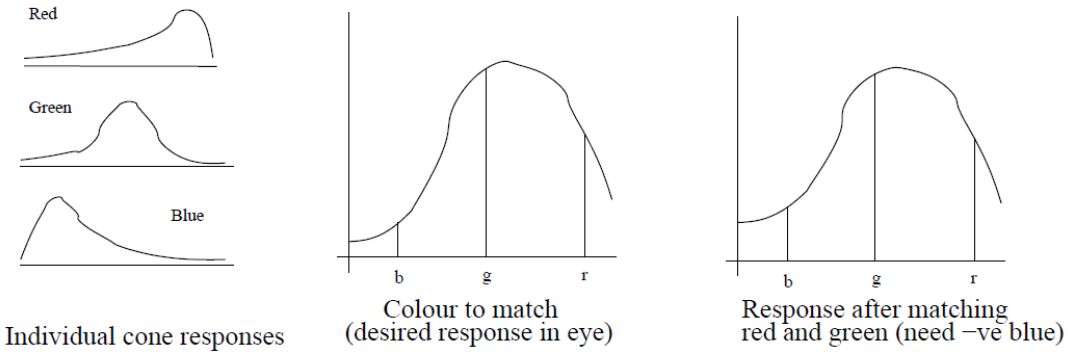

Figure 2.6: Tri-stimulus experiments attempt to match real-world colours with a mix of monochromatic (narrow bandwidth) red, green and blue primary light sources. Certain colours cannot be matched because cones have a wide bandwidth response, e.g. the blue cone responds a little to red and green light (see Section 2.2.4). [**TODO:** redraw cleaner]

A human participant is seated in a darkened room, with two pieces of card before them. 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 monochromatic red, green and blue light sources. The human is given control over the intensities of the red, green and blue primary sources using a dial, 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. [**TODO:** Technically true, but red goes much more negative, so would be more illustrative.]

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.

2.2.5 The CIEXYZ colour space

We can represent all physically realisable colours in the **CIEXYZ colour space**, which was created by the International Commission on Illumination (CIE) in 1931 to standardise the representation of colour.

The CIEXYZ colour space represents colours as a weighted sum of three **ideal primaries** that are written X, Y and Z. These primaries are offset from the human cone responses for 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 Section 2.2.4, the ideal primaries are not physically realisable light sources; rather they are mathematical definitions with useful properties. However, the Y primary (also called **luminance**; do not confuse with the yellow component in CMY) was specifically designed to match the luminous efficiency function of the human eye (Figure 2.5, left).

The triplet (X, Y, Z) is known as the **tristimulus values**. We more commonly work with the **chro-**

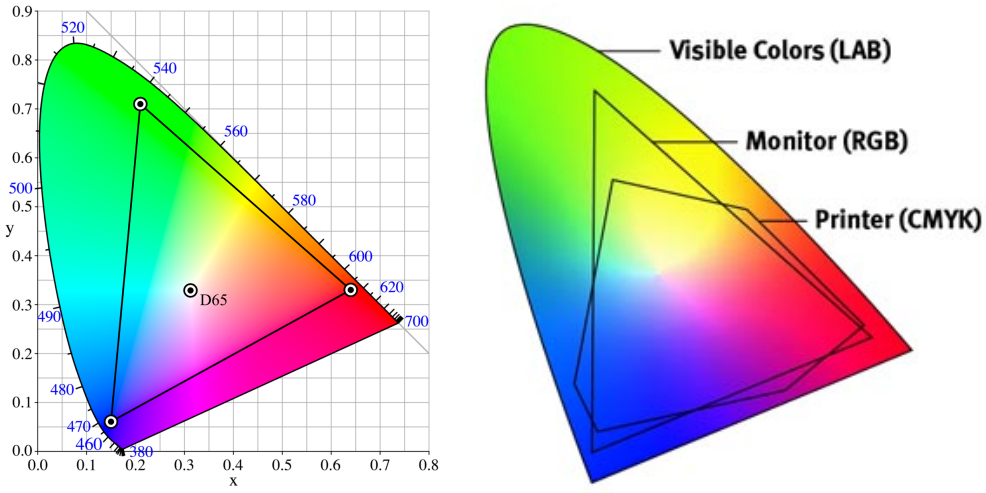

Figure 2.7: **Left:** The CIE (x, y)-chromaticity diagram. Spectral colours appear around the curved edge of the space (wavelengths noted here in nanometres). The triangle describes all colours that can be represented in a particular RGB colour space, here: Adobe RGB (1998). The white point (D65) is roughly in the centre of the space. [Wikimedia Commons / CC BY-SA 3.0] **Right:** Colours obtainable via the RGB (additive) primaries and CMY (subtractive) primaries cover different regions in the chromaticity diagram. The area of overlap shows which display colours can be faithfully reproduced on a printer, and vice versa. [© 2016 Oki Data Americas, Inc.]

maticity coordinates (x, y, z):

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z} \quad \text{and} \quad z = \frac{Z}{X+Y+Z}. \tag{2.5}$$

Since $x+y+z=1$, we need only two of the three normalised coordinates to represent chromaticity. The loss of a dimension is caused by the normalisation.

We need just two coordinates to represent a colour's chromaticity (e.g. whether it is red, yellow, blue, etc.), and an extra non-normalised coordinate to recover the colour's luminance (brightness). Using the x and y chromaticity coordinates, and the Y tristimulus value, results in the **CIE xyY colour space** which is widely used in practice for specifying colours. We can easily recover the (X, Y, Z) tristimulus values from (x, y, Y) using

$$X = \frac{x}{y}Y, \quad Y = Y \quad \text{and} \quad Z = \frac{1-x-y}{y}Y. \tag{2.6}$$

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) and produce pastel shades. Monochromatic (single-wavelength) 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 such as browns.

We do not need to explore the CIEXYZ colour space in greater depth for the purposes of this course[5]. There are some small modifications that can be made to the CIEXYZ model to transform the space into one where the Euclidean distance between points in the space corresponds more closely to the perceptual distance between colours. For further details, look up the CIELAB colour space.

[5]The interested reader is encouraged to read the excellent article at https://en.wikipedia.org/wiki/CIE_1931_color_space.

However, there is one final, important observation arising from the CIE xyY colour space to discuss. We can see that red, green and blue are points in the CIE (x, y) diagram. Any linear combination of those **additive** primaries must therefore fall within a triangle in the CIE (x, y) diagram. The colours in that triangle are those that can be realised using three physical primaries. 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.

As illustrated in Figure 2.7, it is very likely that the region created by your printer's primaries does not exactly overlap the triangle created by your display's primaries (and vice versa). 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 in CIEXYZ space to another. With the driver, various (often proprietary) algorithms exist to transform between the two regions in CIEXYZ space without introducing perceptually unexpected artefacts. This is in addition to differences in brightness reproduction; printers cannot make a colour appear brighter than an empty sheet of paper, while display tend to achieve much higher brightness levels.

2.2.6 The HSV colour space

The RGB, CMY and CIEXYZ colour spaces are not very intuitive for most users, even experts. People tend to think of colour in terms of hue, rather than primaries. The **HSV colour space** (hue, saturation, value) tries to model this. It is used extensively in graphical interfaces, for example for colour pickers.

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 $[0, 255]$. The **value** component v is the easiest to compute:

$$v = \max(r, g, b). \tag{2.7}$$

This is a fast approximation to luminance. In the related **HSL colour space** (hue, saturation, lightness), the **lightness** l is computed using

$$l = \frac{\max(r, g, b) + \min(r, g, b)}{2}. \tag{2.8}$$

The two models are functionally near-identical.

For the remaining two components, we first need to introduce the maximum M and minimum m of the RGB colour channels:

$$M = \max(R, G, B) \quad \text{and} \quad m = \min(R, G, B). \tag{2.9}$$

The **saturation** component s of a colour is then defined slightly differently for the HSV and HSL colour spaces:

$$s_{\text{HSV}} = \begin{cases} 0, & \text{if } M = m \\ \frac{M-m}{v}, & \text{otherwise} \end{cases} \tag{2.10}$$

$$s_{\text{HSL}} = \begin{cases} 0, & \text{if } M = m \\ \frac{M-m}{1-|2l-1|}, & \text{otherwise} \end{cases} \tag{2.11}$$

In both colour spaces, a greyscale colour (x, x, x), where all colour channels are the same, would satisfies $M = m$ and thus has zero saturation ($s = 0$). Colours that are close to greys are *desaturated*. The more the minimum and maximum of the colour channels are apart (i.e. the larger $M - m$), the more vivid or *saturated* a colour is.

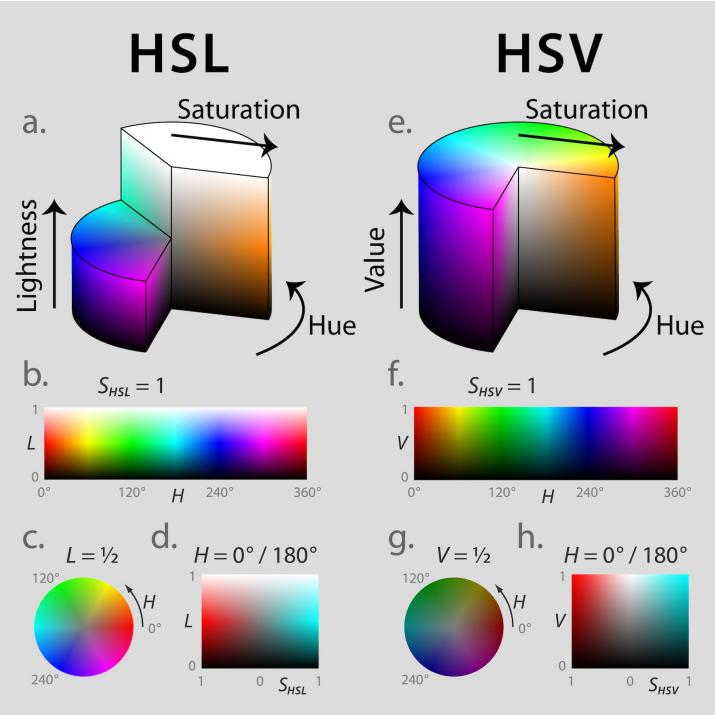

Figure 2.8: Visualisations of the HSL (a–d) and HSV (e–h) colour spaces. **Top:** (a, e) cut-away 3D models of each. **Below:** 2D visualisations showing each pair of coordinates while keeping the other constant. (b, f) The outer cylinder shells at full saturation. (c, g) A horizontal slice through the cylinders at half lightness/value. (d, h) A vertical slice through the cylinders at constant hue (0°/180° for red–cyan). [Jacob Rus / CC BY-SA 3.0]

Finally, the **hue** component h describes the angle along the 'colour circle' in Figure 2.8(c,g):

$$h = 60° \cdot \begin{cases} \text{undefined}, & \text{if } M = m \\ \frac{g-b}{M-m} \bmod 6, & \text{if } M = r \\ \frac{b-r}{M-m} + 2, & \text{if } M = g \\ \frac{r-g}{M-m} + 4, & \text{if } M = b \end{cases} \tag{2.12}$$

This definition considers different cases depending on the maximum colour channel. First, if $M = m$, then all channels are equal, which means the colour is some shade of grey. In this case, the hue is undefined, although $h = 0$ is usually used for convenience. In the second case, red is the maximum channel, and the amount of green and blue relative to saturation determines the hue angle. The remaining two cases work similarly, just with shifts of 120° (green) and 240° (blue) along the colour circle.

Example Let's convert the colour $(r, g, b) = (0.3, 0.6, 0.2)$ to the HSV and HSL colour spaces. Both colour spaces share the same hue component (case $M = g$ in Equation 2.12):

$$h = 60° \cdot \left(\frac{b-r}{M-m} + 2 \right) = 60° \cdot \left(\frac{0.2 - 0.3}{0.6 - 0.2} + 2 \right) = 60° \cdot 1.75 = 105°. \tag{2.13}$$

We next compute the value and lightness:

$$v = \max(r, g, b) = \max(0.3, 0.6, 0.2) = 0.6, \tag{2.14}$$

$$l = \frac{\max(r, g, b) + \min(r, g, b)}{2} = \frac{0.6 + 0.2}{2} = 0.4, \tag{2.15}$$

followed by the different saturations for the HSV/HSL colour spaces:

$$s_{\text{HSV}} = \begin{cases} 0, & \text{if } M = m \\ \frac{M-m}{v}, & \text{otherwise} \end{cases} \Bigg| = \frac{M-m}{v} = \frac{0.6-0.2}{0.6} = \frac{2}{3}, \tag{2.16}$$

$$s_{\text{HSL}} = \begin{cases} 0, & \text{if } M = m \\ \frac{M-m}{1-|2l-1|}, & \text{otherwise} \end{cases} \Bigg| = \frac{M-m}{1-|2l-1|} = \frac{0.6-0.2}{1-|2\cdot 0.4-1|} = \frac{0.4}{0.8} = 0.5. \tag{2.17}$$

2.2.7 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 colour space, the colour space you should choose for your work depends on your requirements. For example, HSV is often more intuitive for non-expert users to understand, and so is commonly used for colour pickers, for example. However, the fact that many points in HSV space map to black ($(h, s, 0)$), 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, for example in a photo editing software. In RGB space, these colours are $(255, 0, 0)$ and $(0, 255, 0)$, respectively. In HSV space, these colours are $(0°, 1, 1)$ and $(120°, 1, 1)$, respectively. Using linear interpolation, a colour halfway between these two is therefore $(128, 128, 0)$ in RGB, and $(60°, 1, 1)$ HSV. 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 or interpolating colours.

Chapter 3

The Fourier transform

Note: This chapter will be completed soon.

3.1 Complex numbers

A complex number is a number that can be written as $a+bi$, where a and b are two real numbers, and i is the **imaginary unit** which satisfies $i^2 = -1$ (or "$i = \sqrt{-1}$"). Complex numbers extend the real one-dimensional number line to the two-dimensional **complex plane**, as illustrated to the right. A complex number $z = a + bi \in \mathbb{C}$ has two parts: the **real part** $\mathrm{Re}(z) = a$ and the **imaginary part** $\mathrm{Im}(z) = b$, both of which are real numbers ($a, b \in \mathbb{R}$).

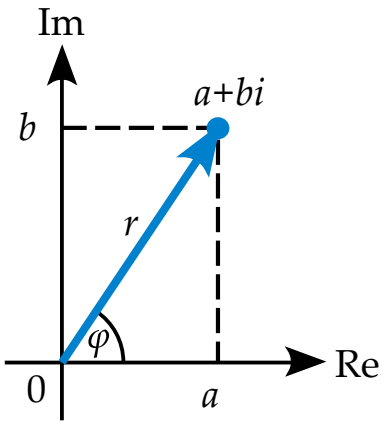

Illustration of a complex number $a + bi$ in the 2D plane. [Wolfkeeper / Wikipedia / CC BY-SA 3.0]

Any real number a is also a complex number $a + 0i$ with an imaginary part that is zero. A purely imaginary number bi has a real part of zero, i.e. $0 + bi$. However, the shorter versions are generally preferred. When the imaginary part is negative, it is common to write $a - bi$ with $b > 0$ instead of $a + (-b)i$, e.g. $2 - 7i$ rather than $2 + (-7)i$.

An operation unique to complex numbers is the **complex conjugate** $\overline{z}$ of a number z:

$$\overline{z} = \mathrm{Re}(z) - \mathrm{Im}(z) \cdot i = a - bi. \tag{3.1}$$

The complex conjugate negates the imaginary part of the number while keeping the real part the same. Geometrically speaking, the result is that the number z is reflected about the real axis. Double conjugation returns to the original value: $\overline{\overline{z}} = z$.

Sometimes it is convenient to express complex numbers in **polar form**:

$$z = r \cdot \mathrm{e}^{-i\varphi}, \tag{3.2}$$

using the radius $r \in \mathbb{R}$ and argument (angle) $\varphi \in \mathbb{R}$. The angle is measured (in radians, i.e. $\pi \leftrightarrow 180°$) counter-clockwise from the real axis. The radius is also known as the **magnitude** or **absolute value**:

$$|z| = \sqrt{z\overline{z}} = \sqrt{(a+bi)(a-bi)} = \sqrt{a^2 - b^2 i^2} = \sqrt{a^2 + b^2}. \tag{3.3}$$

3.1.1 Arithmetic with complex numbers

Complex numbers can be manipulated like any other mathematical expressions, except that they can sometimes be simplified using $i^2 = -1$, as we shall see. Addition and subtraction are performed per component:

$$(a + bi) + (c + di) = (a - c) + (c - d) \cdot i, \tag{3.4}$$
$$(a + bi) - (c + di) = (a - c) + (c - d) \cdot i. \tag{3.5}$$

To multiply two complex numbers, we first multiply out their expressions, and then collect real and imaginary terms (using $i^2 = -1$):

$$(a + bi) \cdot (c + di) = ac + adi + bci + bdi^2 \tag{3.6}$$
$$= (ac - bd) + (ad + bc) \cdot i. \tag{3.7}$$

Multiplying two complex numbers in polar form (Equation 3.2) is particularly easy:

$$(r_1 e^{i\varphi_1}) \cdot (r_2 e^{i\varphi_2}) = r_1 r_2 e^{i(\varphi_1 + \varphi_2)}, \tag{3.8}$$

as their radii r_1 and r_2 are multiplied, and angles φ_1 and φ_2 are simply added.

Dividing two complex numbers is less straightforward, and requires a trick:

$$\frac{a + bi}{c + di} = \frac{(a + bi) \cdot (c - di)}{(c + di) \cdot (c - di)} = \frac{ac - adi + cbi - bdi^2}{c^2 - di^2} \tag{3.9}$$
$$= \frac{ac - adi + cbi + bd}{c^2 + d} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i. \tag{3.10}$$

Notice that both sides of the fraction are multiplied by $c - di$, the complex conjugate of $c + di$ (Equation 3.1), to eliminate the imaginary part of the denominator (below the line). Finally, individual terms are rearranged to collect real and imaginary parts.

3.1.2 Euler's formula

Euler's formula establishes a relationship between the complex exponential function (e^x) and trigonometric functions ($\sin x$, $\cos x$). For any real number (angle) φ,

$$e^{i\varphi} = \cos\varphi + i \sin\varphi, \tag{3.11}$$

where $e \approx 2.71828$ is Euler's number[1] and the argument φ is given in radians.

Euler's formula describes the complex numbers on the unit circle, i.e. with a radius of $r = 1$, but angle φ anticlockwise from the real axis. We can verify this by computing the magnitude of $e^{i\varphi}$ using Equation 3.3:

$$\left|e^{i\varphi}\right| = \sqrt{e^{i\varphi} \cdot \overline{e^{i\varphi}}} \tag{3.12}$$
$$= \sqrt{(\cos\varphi + i\sin\varphi) \cdot \overline{(\cos\varphi + i\sin\varphi)}} \tag{3.13}$$
$$= \sqrt{(\cos\varphi + i\sin\varphi) \cdot (\cos\varphi - i\sin\varphi)} \tag{3.14}$$
$$= \sqrt{\cos^2\varphi + i\sin\varphi\cos\varphi - i\sin\varphi\cos\varphi - i^2\sin^2\varphi} \tag{3.15}$$
$$= \sqrt{\cos^2\varphi + \sin^2\varphi} = 1. \tag{3.16}$$

[1] https://en.wikipedia.org/wiki/E_(mathematical_constant)

We can use Euler's formula and the trigonometric identities

$$\sin(-\varphi) = -\sin\varphi, \tag{3.17}$$
$$\cos(-\varphi) = \cos\varphi \tag{3.18}$$

to express the functions $\sin$ and $\cos$ in terms of Euler's number e and the imaginary unit i:

$$\cos\varphi = \frac{\mathrm{e}^{i\varphi} + \mathrm{e}^{-i\varphi}}{2}, \tag{3.19}$$
$$\sin\varphi = \frac{\mathrm{e}^{i\varphi} - \mathrm{e}^{-i\varphi}}{2i}. \tag{3.20}$$

We can verify these expressions by plugging in Euler's formula (Equation 3.11), and simplifying the terms:

$$\cos\varphi = \frac{\mathrm{e}^{i\varphi} + \mathrm{e}^{-i\varphi}}{2} = \frac{\mathrm{e}^{i\varphi} + \mathrm{e}^{i(-\varphi)}}{2} = \frac{(\cos\varphi + i\sin\varphi) + (\cos(-\varphi) + \sin(-\varphi))}{2} \tag{3.21}$$
$$= \frac{\cos\varphi + i\sin\varphi + \cos\varphi - i\sin\varphi}{2} = \frac{2\cos\varphi}{2} = \cos\varphi, \tag{3.22}$$
$$\sin\varphi = \frac{\mathrm{e}^{i\varphi} - \mathrm{e}^{-i\varphi}}{2i} = \frac{\mathrm{e}^{i\varphi} - \mathrm{e}^{i(-\varphi)}}{2i} = \frac{(\cos\varphi + i\sin\varphi) - (\cos(-\varphi) + i\sin(-\varphi))}{2i} \tag{3.23}$$
$$= \frac{\cos\varphi + i\sin\varphi - \cos\varphi - i\sin\varphi}{2i} = \frac{2i\sin\varphi}{2i} = \sin\varphi. \tag{3.24}$$

3.2 Even and odd functions

A function $f(x)$ is **even**, if $f(-x) = f(x)$, i.e. if the function f is symmetric about the y-axis. For example, the function $f(x) = |x|$ is even, as $f(-x) = |-x| = |x| = f(x)$. Another even function is $\cos x$, as $\cos(-x) = \cos x$ (see Equation 3.18).

A function $f(x)$ is **odd**, if $f(-x) = -f(x)$, i.e. if the function f is anti-symmetric about the y-axis. For exmaple, the function $f(x) = x$ is odd, as $f(-x) = -x = -f(x)$. Another odd function is $\sin x$ is even, as $\sin(-x) = -\sin x$ (see Equation 3.17).

If functions $f(x)$ and $g(x)$ are even, then their product $h(x) = f(x)g(x)$ is even.

If functions $f(x)$ and $g(x)$ are odd, then their product $h(x) = f(x)g(x)$ is even.

If $f(x)$ is even and $g(x)$ is odd, then their product $h(x) = f(x)g(x)$ is odd.

If a function $f(x)$ is odd, then its integral $\int_{-a}^{a} f(x)\,dx = 0$ for any $a > 0$.

If a function $f(x)$ is even, then $\int_{-a}^{a} f(x)\,dx = 2\int_{0}^{a} f(x)\,dx$ for any $a > 0$.

Any function $f(x)$ is the sum of an even and an off function:

$$f(x) = f_{\text{even}}(x) + f_{\text{odd}}(x), \tag{3.25}$$

which can be obtained using

$$f_{\text{even}}(x) = \frac{f(x) + f(-x)}{2}, \tag{3.26}$$
$$f_{\text{odd}}(x) = \frac{f(x) - f(-x)}{2}. \tag{3.27}$$

3.3 Definition of the Fourier transform

The **Fourier transform** $\mathcal{F}[f](\omega)$ of a function $f(x)$ is defined by:

$$F(\omega) = \mathcal{F}[f](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) \mathrm{e}^{-i\omega x} dx, \tag{3.28}$$

where ω is the frequency.

The **inverse Fourier transform** restores the function f from F:

$$f(x) = \mathcal{F}^{-1}[F](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(\omega) \mathrm{e}^{i\omega x} d\omega. \tag{3.29}$$

Note the similarity between the Fourier transform and its inverse. We use $F(\omega)$ and $\mathcal{F}[f](\omega)$ interchangeably to refer to the representation of $f(x)$ in the frequency (or Fourier) domain.

3.3.1 Example: Fourier transform of the box function

Let's compute the Fourier transform of the **box function**:

$$\mathrm{box}(x) = \begin{cases} 1 & \text{if } -1 < x < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{3.30}$$

Short derivation

$$\mathcal{F}[\mathrm{box}](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{box}(x)\, \mathrm{e}^{-i\omega x}\, dx = \frac{1}{\sqrt{2\pi}} \int_{-1}^{1} \mathrm{e}^{-i\omega x}\, dx = \frac{1}{\sqrt{2\pi}} \left[\frac{\mathrm{e}^{-i\omega x}}{-i\omega} \right]_{-1}^{1} \tag{3.31}$$

$$= \frac{1}{\sqrt{2\pi}} \frac{\mathrm{e}^{i\omega} - \mathrm{e}^{-i\omega}}{i\omega} = \frac{1}{\sqrt{2\pi}} \frac{2}{\omega} \frac{\mathrm{e}^{i\omega} - \mathrm{e}^{-i\omega}}{2i} = \sqrt{\frac{2}{\pi}} \frac{\sin\omega}{\omega} = \sqrt{\frac{2}{\pi}} \mathrm{sinc}(\omega). \tag{3.32}$$

Detailed derivation

Starting from the definition of the Fourier transform $\mathcal{F}[\text{box}](\omega)$ of the function $\text{box}(x)$:

$$\mathcal{F}[\text{box}](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \text{box}(x)\, \mathrm{e}^{-i\omega x}\, dx \quad \leftarrow \text{this is just Equation 3.28} \tag{3.33}$$

$$\downarrow \quad \text{split integration domain into 3 parts: } x<-1,\ -1<x<1,\ x>1$$

$$= \frac{1}{\sqrt{2\pi}} \left(\int_{-\infty}^{-1} \text{box}(x)\, \mathrm{e}^{-i\omega x}\, dx + \int_{-1}^{1} \text{box}(x)\, \mathrm{e}^{-i\omega x}\, dx + \int_{1}^{\infty} \text{box}(x)\, \mathrm{e}^{-i\omega x}\, dx \right) \tag{3.34}$$

$$\downarrow \quad \text{plug in the definition of the box function (Equation 3.30)}$$

$$= \frac{1}{\sqrt{2\pi}} \left(\int_{-\infty}^{-1} 0 \cdot \mathrm{e}^{-i\omega x}\, dx + \int_{-1}^{1} 1 \cdot \mathrm{e}^{-i\omega x}\, dx + \int_{1}^{\infty} 0 \cdot\ \mathrm{e}^{-i\omega x}\, dx \right) \tag{3.35}$$

$$= \frac{1}{\sqrt{2\pi}} \left(\int_{-\infty}^{-1} 0\, dx + \int_{-1}^{1} \mathrm{e}^{-i\omega x}\, dx + \int_{1}^{\infty} 0\, dx \right) \tag{3.36}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-1}^{1} \mathrm{e}^{-i\omega x}\, dx \tag{3.37}$$

$$\downarrow \quad \text{integrate expression } \mathrm{e}^{-i\omega x}$$

$$= \frac{1}{\sqrt{2\pi}} \left[\frac{\mathrm{e}^{-i\omega x}}{-i\omega} \right]_{-1}^{1} = \frac{1}{\sqrt{2\pi}} \left(\frac{\mathrm{e}^{-i\omega}}{-i\omega} - \frac{\mathrm{e}^{i\omega}}{-i\omega} \right) = \frac{1}{\sqrt{2\pi}} \left(\frac{\mathrm{e}^{i\omega} - \mathrm{e}^{-i\omega}}{i\omega} \right) \tag{3.38}$$

$$\downarrow \quad \text{rearrange to match Euler's formula expression } \sin(\omega) = \frac{\mathrm{e}^{i\omega} - \mathrm{e}^{-i\omega}}{2i} \text{ (see 3.20)}$$

$$= \frac{2}{\sqrt{2\pi}\omega} \left(\frac{\mathrm{e}^{i\omega} - \mathrm{e}^{-i\omega}}{2i} \right) = \sqrt{\frac{2}{\pi}} \frac{\sin\omega}{\omega} \quad \leftarrow \text{It's okay to stop here.} \tag{3.39}$$

$$= \sqrt{\frac{2}{\pi}} \text{sinc}(\omega) \quad \leftarrow \text{This line is just FYI.} \tag{3.40}$$

3.4 Properties of the Fourier transform

If $f(x)$ is even and real-valued, then $F(\omega)$ is even and real-valued.

If $f(x)$ is odd and real-valued, then $F(\omega)$ is odd and purely imaginary.

3.4.1 The linearity property

For constants a and b, the Fourier transform of a function $h(x) = a \cdot f(x) + b \cdot g(x)$ is given by:

$$\mathcal{F}[h](\omega) = a \cdot \mathcal{F}[f](\omega) + b \cdot \mathcal{F}[g](\omega). \tag{3.41}$$

Proof
Starting from the definition of the Fourier transform $\mathcal{F}[h](\omega)$ of the function $h(x)$:

$$\mathcal{F}[h](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} h(x)\, \mathrm{e}^{-i\omega x}\, dx \quad \leftarrow \text{this is just Equation 3.28} \tag{3.42}$$

$$\downarrow \quad \text{substitute } h(x) \text{ with } a\cdot f(x) + b\cdot g(x)$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} (a\cdot f(x) + b\cdot g(x))\, \mathrm{e}^{-i\omega x}\, dx \tag{3.43}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} a\cdot f(x)\, \mathrm{e}^{-i\omega x}\, dx + \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} b\cdot g(x)\, \mathrm{e}^{-i\omega x}\, dx \tag{3.44}$$

$$= a\cdot \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, \mathrm{e}^{-i\omega x}\, dx + b\cdot \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x)\, \mathrm{e}^{-i\omega x}\, dx \tag{3.45}$$

$$\downarrow \quad \text{match with definition of the Fourier transforms of } f \text{ and } g$$

$$= a\cdot \mathcal{F}[f](\omega) + b\cdot \mathcal{F}[g](\omega). \tag{3.46}$$

3.4.2 The shifting property

For a constant a, the Fourier transform of a function $g(x) = f(x-a)$ is given by:

$$\mathcal{F}[g](\omega) = \mathrm{e}^{-ia\omega}\mathcal{F}[f](\omega). \tag{3.47}$$

Proof
Starting from the definition of the Fourier transform $\mathcal{F}[g](\omega)$ of the function $g(x)$:

$$\mathcal{F}[g](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x)\, \mathrm{e}^{-i\omega x}\, dx \quad \leftarrow \text{this is just Equation 3.28} \tag{3.48}$$

$$\downarrow \quad \text{substitute } g(x) \text{ with } f(x-a)$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x-a)\, \mathrm{e}^{-i\omega x}\, dx \tag{3.49}$$

$$\downarrow \quad \text{integration by substitution using } y = x-a$$

$$= \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\infty} f(y)\, \mathrm{e}^{-i\omega(y+a)}\, dy \tag{3.50}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\infty} f(y)\, \mathrm{e}^{-i\omega y}\mathrm{e}^{-ia\omega}\, dy \tag{3.51}$$

$$= \mathrm{e}^{-ia\omega} \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\infty} f(y)\, \mathrm{e}^{-i\omega y}\, dy \tag{3.52}$$

$$\downarrow \quad \text{match with definition of the Fourier transform of } f$$

$$= \mathrm{e}^{-ia\omega}\mathcal{F}[f](\omega). \tag{3.53}$$

3.4.3 The modulation property

For a constant a, the Fourier transform of a function $g(x) = \mathrm{e}^{iax} f(x)$ is given by:

$$\mathcal{F}[g](\omega) = \mathcal{F}[f](\omega - a). \tag{3.54}$$

Proof
Starting from the definition of the Fourier transform $\mathcal{F}[g](\omega)$ of the function $g(x)$:

$$\mathcal{F}[g](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x)\, \mathrm{e}^{-i\omega x}\, dx \quad \leftarrow \text{this is just Equation 3.28} \tag{3.55}$$

$$\downarrow \quad \text{substitute } g(x) \text{ with } \mathrm{e}^{iax} f(x)$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{e}^{iax} f(x)\, \mathrm{e}^{-i\omega x}\, dx \tag{3.56}$$

$$\downarrow \quad \text{rearrange using } \mathrm{e}^{iax}\mathrm{e}^{-i\omega x} = \mathrm{e}^{iax - i\omega x} = \mathrm{e}^{-i(\omega - a)x}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, \mathrm{e}^{-i(\omega - a)x}\, dx \tag{3.57}$$

$$\downarrow \quad \text{match with definition of the Fourier transform of } f$$

$$= \mathcal{F}[f](\omega - a). \tag{3.58}$$

3.4.4 The scaling property

For a constant a, the Fourier transform of a function $g(x) = f(ax)$ is given by:

$$\mathcal{F}[g](\omega) = \frac{1}{|a|} \mathcal{F}[f]\left(\frac{\omega}{a}\right). \tag{3.59}$$

Proof
Starting from the definition of the Fourier transform $\mathcal{F}[g](\omega)$ of the function $g(x)$:

$$\mathcal{F}[g](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x)\, \mathrm{e}^{-i\omega x}\, dx \quad \leftarrow \text{this is just Equation 3.28} \tag{3.60}$$

$$\downarrow \quad \text{substitute } g(x) \text{ with } f(ax)$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(ax)\, \mathrm{e}^{-i\omega x}\, dx \tag{3.61}$$

$$\downarrow \quad \text{integration by substitution using } y = ax$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-a\cdot\infty}^{a\cdot\infty} f(y)\, \mathrm{e}^{-i\omega y/a} \frac{1}{a}\, dy \tag{3.62}$$

$$\downarrow \quad \text{need to do a case split, as } a < 0 \text{ swaps integration limits}$$

Case $a > 0$: $$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(y)\, \mathrm{e}^{-i\omega y/a} \frac{1}{a}\, dy \tag{3.63}$$

$$= \frac{1}{|a|} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(y)\, \mathrm{e}^{-i(\omega/a)y}\, dy \right] \tag{3.64}$$

$$\downarrow \quad \text{match with definition of the Fourier transform of } f$$

$$= \frac{1}{|a|} \mathcal{F}[f]\left(\frac{\omega}{a}\right), \tag{3.65}$$

Case $a < 0$: $$= \frac{1}{\sqrt{2\pi}} \int_{\infty}^{-\infty} f(y)\, \mathrm{e}^{-i\omega y/a} \frac{1}{a}\, dy = -\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(y)\, \mathrm{e}^{-i\omega y/a} \frac{1}{a}\, dy \tag{3.66}$$

$$= \frac{1}{|a|} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(y)\, \mathrm{e}^{-i(\omega/a)y}\, dy \right] \tag{3.67}$$

$$\downarrow \quad \text{match with definition of the Fourier transform of } f$$

$$= \frac{1}{|a|} \mathcal{F}[f]\left(\frac{\omega}{a}\right), \tag{3.68}$$

Case $a = 0$: undefined (division by zero). (3.69)

3.4.5 The differentiation property

The Fourier transform for the derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$ of a function $f(x)$ is given by:

$$\mathcal{F}\left[\frac{\mathrm{d}f}{\mathrm{d}x}\right](\omega) = i\omega\mathcal{F}[f](\omega). \tag{3.70}$$

Taking the derivative in the spatial domain multiplies the Fourier transform with the frequency.

Proof
Starting from the inverse Fourier transform of the Fourier transform of $f(x)$:

$$f(x) = \mathcal{F}^{-1}\left[\mathcal{F}[f]\right](x) \tag{3.71}$$

$$\downarrow \quad \text{using the definition of the inverse Fourier transform}$$

$$= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\mathcal{F}[f](\omega)\,\mathrm{e}^{i\omega x}\,d\omega \tag{3.72}$$

$$\downarrow \quad \text{differentiate both sides of the equation with respect to } x$$

$$\frac{\mathrm{d}}{\mathrm{d}x}f(x) = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\mathcal{F}[f](\omega)\,\mathrm{e}^{i\omega x}\,d\omega\right) \tag{3.73}$$

$$= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\mathcal{F}[f](\omega)\,\mathrm{e}^{i\omega x}\,i\omega\,d\omega \tag{3.74}$$

$$= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\left[i\omega\mathcal{F}[f](\omega)\right]\mathrm{e}^{i\omega x}\,d\omega \tag{3.75}$$

$$\downarrow \quad \text{match with definition of the inverse Fourier transform}$$

$$= \mathcal{F}^{-1}\left[i\omega\mathcal{F}[f]\right](x) \tag{3.76}$$

$$\downarrow \quad \text{apply Fourier transform to both sides of the equation}$$

$$\mathcal{F}\left[\frac{\mathrm{d}f}{\mathrm{d}x}\right](\omega) = i\omega\mathcal{F}[f]. \tag{3.77}$$

Corollary
The n^{th}-order derivative $\frac{\mathrm{d}^n f}{\mathrm{d}x^n}$ of a function $f(x)$ is given by:

$$\mathcal{F}\left[\frac{\mathrm{d}^n f}{\mathrm{d}x^n}\right](\omega) = (i\omega)^n\mathcal{F}[f](\omega). \tag{3.78}$$

The higher the order n of differentiation, the stronger the attenuation of low frequencies ω, and the stronger the amplification of high frequencies.

3.4.6 The Convolution theorem

The **convolution** $f * g$ of two functions f and g is given by:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-y)g(y)\,dy. \tag{3.79}$$

The **Convolution theorem** states that:

$$\mathcal{F}[f * g](\omega) = \sqrt{2\pi}\mathcal{F}[f](\omega)\cdot\mathcal{F}[g](\omega), \tag{3.80}$$

i.e. convolution in the spatial domain becomes multiplication in the Fourier domain. Afterwards, transform the result back to spatial domain. As we will see, this can save a lot of time for convolutions with large kernels

The **reciprocal convolution theorem** states that:

$$\mathcal{F}[f \cdot g](\omega) = \sqrt{2\pi}\mathcal{F}[f](\omega) * \mathcal{F}[g](\omega). \tag{3.81}$$

Chapter 4

Geometric transformations

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

In visual computing, 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, which 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.

4.1 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 $\mathbf{p} = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4]$, where $\mathbf{p}_i$ 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

$$\mathbf{p}' = \mathbf{M}\,\mathbf{p}, \tag{4.1}$$

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

These "matrix transformations" are sometimes called **rigid body transformations** because all points $\mathbf{p}$ undergo 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 distortion as in Figure 4.1 using a matrix transformation; this type of distortion is an example of a more general geometric transformation. Rigid body transformations do however form a useful subset of geometric transformations, and we will explore some of these now.

4.1.1 Scaling

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

$$\mathbf{M} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \tag{4.2}$$

Figure 4.1: 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. Note that only rigid body transformations can be realised using matrix multiplication.

where s_x is the **scale factor** by which we wish to enlarge the object in the direction of the x-axis, and s_y similarly scales along the y-axis. So a point $[x, y]^\top$ will be transformed to location $[s_x \cdot x, s_y \cdot y]^\top$.

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 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 4.2 (red) shows an example of scaling where $s_x = 2$, $s_y = 1$, resulting in the square being stretched in the direction of the x-axis to become a rectangle.

You may note that setting both scale factors to 1 resulting in the identity matrix, i.e. it has no effect.

A negative scale factor will reflect points. For example, setting $s_x = -1$ and $s_y = 1$ will reflect points along the y-axis.

4.1.2 Shearing (or skewing)

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

$$\mathbf{M} = \begin{bmatrix} 1 & q \\ 0 & 1 \end{bmatrix}. \tag{4.3}$$

This matrix transforms a point $[x, y]^\top$ to the location $[x + qy, y]^\top$, which shifts the x-component of the point by an amount proportional to its y-component. The parameter q is the constant of proportionality and results, for example, in the vertical lines of the polygon "tipping" into diagonal lines with slope $1/q$. See the green polygon in Figure 4.2 for an example.

4.1.3 Rotation

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

$$\mathbf{M} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{4.4}$$

Figure 4.2 demonstrates the result of this matrix in magenta. Consider the corner $[1, 0]^\top$ transformed by the above matrix. The resulting point has location $[\cos\theta, \sin\theta]^\top$. We can see from the construction in Figure 4.2 that the point is indeed rotated θ degrees anti-clockwise about the origin. Now consider a rotation of the point $[\cos\theta, \sin\theta]^\top$ by a further angle β about the origin; we would expect the new location of the point to be $[cos(\theta + \beta), sin(\theta + \beta)]^\top$. As an exercise, perform this matrix

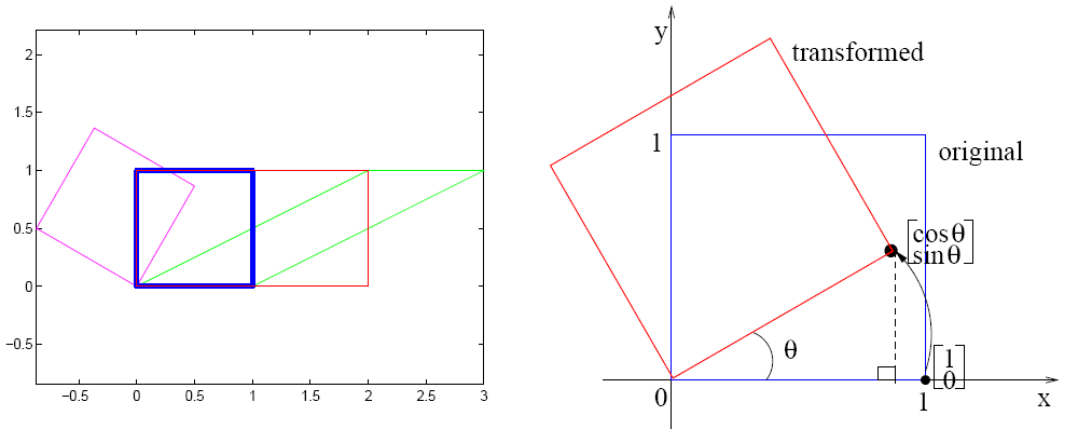

Figure 4.2: Illustration of various linear transformations. **Left:** a blue unit square, with one corner at the origin, is scaled (in red, using $s_x = 2$, $s_y = 1$ in Equation 4.2), sheared (in green, using $q = 2$ in Equation 4.3), and rotated (in magenta, using $\theta = \pi/3$ radians in Equation 4.4). **Right:** A diagram illustrating rotation by θ degrees anti-clockwise about the origin. A corner at location $[1, 0]^\top$ moves to $[\cos\theta, \sin\theta]^\top$ under the rotation.

multiplication and show that the resulting point has location $[\cos\theta\cos\beta - \sin\theta\sin\beta, \sin\theta\cos\beta + \cos\theta sin\beta]^\top$. You may recognise these as the **double-angle formulae** you learnt in trigonometry:

$$\cos(\theta + \beta) = \cos\theta\cos\beta - \sin\theta\sin\beta, \tag{4.5}$$

$$\sin(\theta + \beta) = \sin\theta\cos\beta + \cos\theta\sin\beta. \tag{4.6}$$

The rotation matrix is orthonormal, so its inverse matrix can obtained simply by transposing it (which simplifies to 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 Equation 4.4 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).

4.1.4 Active versus 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 reference frames and basis vectors in Section 1.4.

Example 1: scaling transformation

Recall the scaling matrix in Equation 4.2, where setting $s_x = s_y = 2$ will scale points up by a factor of 2. Consider this transformation acting on a point $\mathbf{p} = [1, 1]^\top$:

$$\mathbf{M}\,\mathbf{p} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}. \tag{4.7}$$

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 $[2, 2]^\top$. This all occurs within the root reference frame in which $\mathbf{p}$ is defined, i.e. with the basis vectors $\hat{\mathbf{i}} = [1, 0]^\top$ and $\hat{\mathbf{j}} = [0, 1]^\top$.

In the **passive interpretation** of the transformation, we imagine $\mathbf{p}$ to stay "fixed", whilst the space it is defined in transforms "beneath" $\mathbf{p}$, contracting to half its size. The tick marks on the axes of the space shrink towards the origin. $\mathbf{p}$ ends up aligned with coordinates $[2, 2]^\top$ with respect to the shrunken space, rather than $[1, 1]^\top$ as was originally the case. Consequently, the space in which the point is defined as $[1, 1]^\top$ is twice as large as the space in which the point is defined as $[2, 2]^\top$, i.e. 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 $\mathbf{M}$ as defining a new basis with basis vectors $(\mathbf{i_M}, \mathbf{j_M})$ defined by the columns of $\mathbf{M}$, i.e. $\mathbf{i_M} = [2, 0]^\top$ and $\mathbf{j_M} = [0, 2]^\top$. In the passive interpretation, we interpret point $[1, 1]^\top$ as existing within this new basis $(\mathbf{i_M}, \mathbf{j_M})$ rather than the root reference frame's basis $(\hat{\mathbf{i}}, \hat{\mathbf{j}})$. 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 $\hat{\mathbf{i}} = [1, 0]^\top$ and $\hat{\mathbf{j}} = [0, 1]^\top$.

The x-coordinate of the point $\mathbf{p}$ within the $(\mathbf{i_M}, \mathbf{j_M})$ reference frame is 1, and the y-coordinate is also 1. So these coordinates contribute $(1 \times 2)\hat{\mathbf{i}} + (1 \times 0)\hat{\mathbf{j}} = 2\hat{\mathbf{i}}$ to the point's x-coordinate in the root frame (the 1s come from the point's coordinates, which are both 1, and the 2 and the 0 come from $\mathbf{i_M} = [2, 0]^\top$). The y-coordinate in the root frame is contributed to by $(1 \times 0)\hat{\mathbf{i}} + (1 \times 2)\hat{\mathbf{j}} = 2\hat{\mathbf{j}}$. So the point's coordinates in the root reference frame is $[2, 2]^\top$.

Thus, to think in the **passive interpretation** is to affect our transformation by thinking of the original coordinates $[1, 1]^\top$ as existing within an arbitrary reference frame, as defined by $\mathbf{M}$. Multiplying the point's coordinates in that frame by $\mathbf{M}$ takes the point out of the arbitrary reference frame and into the root reference frame.

Example 2: rotation transformation

Equation 4.4 describes a rotation operation. Setting $\theta = 45°$ (or $\pi/4$ radians) will rotate the points anti-clockwise about the origin by that amount. For example, a point $[1, 0]^\top$ would undergo the following to end up at $[\cos\theta, \sin\theta]^\top$:

$$\mathbf{M}\,\mathbf{p} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}. \tag{4.8}$$

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 with basis vectors $\hat{\mathbf{i}} = [1, 0]^\top$ and $\hat{\mathbf{j}} = [0, 1]^\top$. So the point moves from $[1, 0]^\top$ to about $[0.707, 0.707]^\top$. This is illustrated in Figure 4.2 (right).

As with the previous example, the passive interpretation is that the point remains "fixed" in terms of coordinates, but the space warps via the inverse transformation. In this case, the space "slips

beneath" the point, rotating by the angle θ in *clockwise* direction. Thus, the point becomes aligned with coordinates $[\cos\theta, \sin\theta]^\top$, in the slipped space, rather than $[1,0]^\top$ as it was in the space before the transformation.

Again, it may be clearer to look at a mathematical explanation. We view the coordinates $[1,0]^\top$ as being defined in a new reference frame with basis vectors taken from the columns of $\mathbf{M}$, i.e. $\mathbf{i_M} = [\cos\theta, \sin\theta]^\top$ and $\mathbf{j_M} = [-\sin\theta, \cos\theta]^\top$. When we multiply by $\mathbf{M}$ we are computing the coordinates of that point within our root reference frame, i.e. where $\hat{\mathbf{i}} = [1,0]^\top$ and $\hat{\mathbf{j}} = [0,1]^\top$. The x-coordinate of the point within the $[\mathbf{i_M}, \mathbf{j_M}]$ reference frame is 1, and the y-coordinate within that frame is 0. So these coordinates contribute $(1 \times \cos\theta)\hat{\mathbf{i}} + (0 \times \sin\theta)\hat{\mathbf{j}}$ to the point's x-coordinate in the root frame (the '$\cos\theta$' and the '$\sin\theta$' come from $\mathbf{i_M} = [\cos\theta, \sin\theta]$). Similarly, the point's y-coordinate within the root frame is contributed to by $(1 \times -\sin\theta)\hat{\mathbf{i}} + (0 \times \cos\theta)\hat{\mathbf{j}}$. So the point's coordinates in the root reference frame are $[\cos\theta, \sin\theta] \approx [0.707, 0.707]^\top$.

4.1.5 Transforming between bases

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

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

$$\mathbf{p_R} = \mathbf{D}\,\mathbf{p_D} \tag{4.9}$$

$$= \begin{bmatrix} \mathbf{i_D} & \mathbf{j_D} \end{bmatrix} \mathbf{p_D}. \tag{4.10}$$

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

$$\mathbf{p_D} = \mathbf{D}^{-1}\,\mathbf{p_R} \tag{4.11}$$

$$= \begin{bmatrix} \mathbf{i_D} & \mathbf{j_D} \end{bmatrix}^{-1} \mathbf{p_R}. \tag{4.12}$$

Suppose we had $\mathbf{p_R} = [1,1]^\top$ and a reference frame defined by the basis vectors $\mathbf{i_D} = [3,0]^\top$ and $\mathbf{j_D} = [0,3]^\top$. The coordinates $\mathbf{p_D}$ following Equation 4.11 would be $[1/3, 1/3]^\top$ – which is as we might expect, because $\mathbf{D}$ 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 $\mathbf{D}$).

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 reference frames

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

First observe that points $\mathbf{p_D}$ and $\mathbf{p_E}$ can be expressed as coordinates $\mathbf{p_R}$ in the root reference frame ($\mathbf{R}$) by simply multiplying by the matrices defining the frames $\mathbf{D}$ and $\mathbf{E}$ (Equation 4.9):

$$\mathbf{p_R} = \mathbf{D}\,\mathbf{p_D}, \tag{4.13}$$

$$\mathbf{p_R} = \mathbf{E}\,\mathbf{p_E}. \tag{4.14}$$

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

$$\mathbf{E}\,\mathbf{p}_{\mathbf{E}} = \mathbf{D}\,\mathbf{p}_{\mathbf{D}} \tag{4.15}$$

$$\mathbf{p}_{\mathbf{E}} = \mathbf{E}^{-1}\,\mathbf{D}\,\mathbf{p}_{\mathbf{D}} \tag{4.16}$$

$$\mathbf{p}_{\mathbf{D}} = \mathbf{D}^{-1}\,\mathbf{E}\,\mathbf{p}_{\mathbf{E}}. \tag{4.17}$$

4.1.6 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 Section 1.6.3 on matrix multiplication.

There are a number of other useful **rigid body transformations** that cannot be achieved using purely 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**. In general, we resort to the messy form of a 'multiplication and an addition' for translations:

$$\mathbf{p}' = \mathbf{M}\,\mathbf{p} + \mathbf{t}. \tag{4.18}$$

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 an $(n+1)$-dimensional space. For example, a homogeneous 2D point 'lives' in a 3D space:

$$\begin{bmatrix} x \\ y \end{bmatrix} \sim \begin{bmatrix} wx \\ wy \\ w \end{bmatrix}. \tag{4.19}$$

By convention, we usually set $w=1$, so a 2D point $[2,3]^\top$ is written as $[2,3,1]^\top$ in homogeneous coordinates. We can manipulate such 2D points using 3×3 matrices instead of the 2×2 matrices 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-direction, respectively:

$$\mathbf{p}' = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}. \tag{4.20}$$

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, anti-clockwise rotation about the origin (Equation 4.4) would be:

$$\mathbf{p}' = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}. \tag{4.21}$$

Following the transformation, we end up with a point in the form $\mathbf{p} = [x, y, w]^\top$. We divide by the 'homogeneous coordinate' w to obtain the true location 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 apply linear transformations with matrix multiplications. Not only this, but they enable translation, too. Translation is one example from a superset of transformations called the **affine transformations** that incorporate all the linear transformations.

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 w) 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. Note that the homogeneous points $[wx, wy, w]^\top$ for $w \neq 0$ all lie on a line through the origin, but in fact represent the same point $[x, y]^\top$ in 2D 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, for example when passing them as arguments between functions. Second, having transformations represented in matrix form makes computing their inverse information as easy as inverting the matrix. Third, and most 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 the next subsection, we can, for example, multiply rotation and translation matrices (Equations 4.20 and 4.21) 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 precomputed prior to, say, running an animation), rather than applying several matrix multiplications inside an animation loop, can yield substantial improvements in efficiency.

4.1.7 Compound matrix transformations

We can combine the basic building blocks of translation, rotation about the origin, scaling about the origin and so on, 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 $\mathbf{S}$ that scales points up by a factor of 2, and a matrix $\mathbf{T}$ that translates points 5 units to the left. We wish to translate points, then scale them up. Given a point $\mathbf{p}$, we could write the translation step as:

$$\mathbf{p}' = \mathbf{T}\,\mathbf{p} \tag{4.22}$$

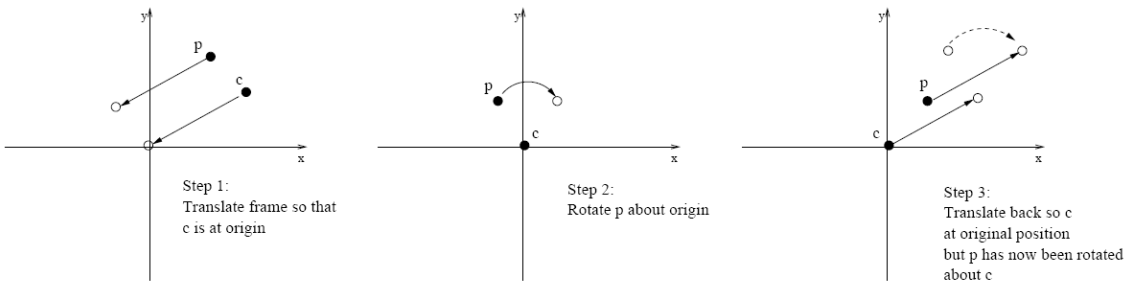

Figure 4.3: Illustrating the three steps (matrix transformations) required to rotate a 2D point about an arbitrary centre of rotation. These three operations are combined in a compound matrix transformation able to perform this 'complex' operation using a single 3×3 matrix.

and the subsequent scaling step as

$$\mathbf{p}'' = \mathbf{S}\,\mathbf{p}', \tag{4.23}$$

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

$$\mathbf{p}'' = \mathbf{S}(\mathbf{T}\,\mathbf{p}), \tag{4.24}$$

and because matrix multiplication is associative, we can write

$$\mathbf{p}'' = \mathbf{S}\,\mathbf{T}\,\mathbf{p}, \tag{4.25}$$

$$\mathbf{p}'' = \mathbf{M}\,\mathbf{p}, \tag{4.26}$$

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

Order of composition

Recall that matrix multiplication is non-commutative (Section 1.6.3). This means that $\mathbf{S}\,\mathbf{T} \neq \mathbf{T}\,\mathbf{S}$, 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 (closest to $\mathbf{p}$). Operations we perform last are at the left-most end – see Equation 4.25.

2D Rotation about an arbitrary point

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

Suppose we wish to rotate a point $\mathbf{p}$ by θ degrees anti-clockwise about centre of rotation $\mathbf{c}$. This can be achieved using three matrix multiplications, which are illustrated in Figure 4.3. First, we translate the reference frame in which $\mathbf{p}$ is defined, so that $\mathbf{c}$ coincides with the origin. If we write $\mathbf{c} = [c_x, c_y]^\top$, then this translation operation is simply

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.27}$$

Now that the centre of rotation is at the origin, we can apply our basic rotation matrix to rotate the point by the user-specified angle θ:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.28}$$

Finally, we translate the reference frame so that $\mathbf{c}$ moves back from the origin to its original position. The result is that the point $\mathbf{p}$ has been rotated by the angle θ about $\mathbf{c}$.

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

$$\mathbf{p}' = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p} \tag{4.29}$$

$$\mathbf{p}' = \mathbf{T}^{-1}\,\mathbf{R}(\theta)\,\mathbf{T}\,\mathbf{p} \tag{4.30}$$

$$\mathbf{p}' = \mathbf{M}\,\mathbf{p}, \tag{4.31}$$

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

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-anti-clockwise 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 $\mathbf{M}$ at each time-step to rotate point $\mathbf{p}$ a further θ degrees about $\mathbf{c}$. After one time step, the position of the point would be $\mathbf{M}\,\mathbf{p}$, after the second time step, the position would be $\mathbf{M}\,\mathbf{M}\,\mathbf{p}$ and so on until the n-th time step, where the point is at $\mathbf{M}^n\,\mathbf{p}$. Precomputing $\mathbf{M}$ 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.

4.1.8 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 $\mathbf{e} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n]$ that describe vertices of a polygon representing the Earth, and a set of points $\mathbf{m}$ that similarly approximate the Moon. Both polygons are squares centred at the origin; the Moon's square is smaller than the Earth (see Figure 4.4, left). 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:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.32}$$

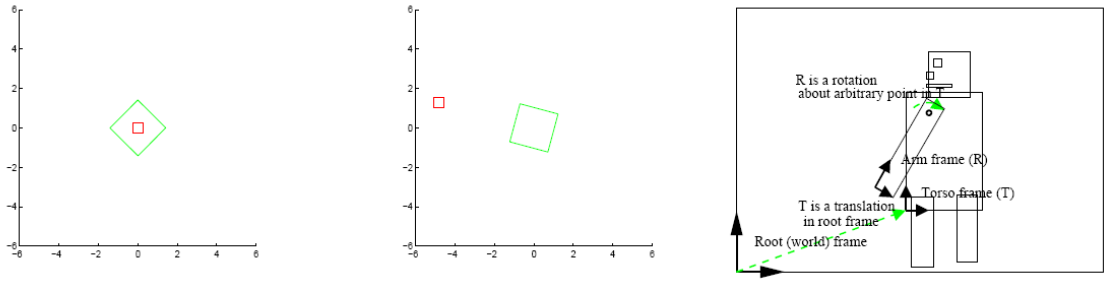

Figure 4.4: Illustration of animation hierarchies. **Left, middle:** Steps toward animation of a spinning Moon orbiting a spinning Earth (see Section 4.1.8). **Right:** A hierarchy for an articulated body such as a walking person. The torso's frame moves within the root reference frame, and limbs directly attached to the torso move within the reference frame defined by the torso.

We increase the angle 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, to enable the rotation speeds to differ, for example if $\theta_m = 5\theta_e$, then the Moon would rotate 5 times faster than the Earth.

$$\mathbf{e}' = \mathbf{R}(\theta_e)\,\mathbf{e}, \tag{4.33}$$

$$\mathbf{m}' = \mathbf{R}(\theta_m)\,\mathbf{m}. \tag{4.34}$$

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:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.35}$$

$$\mathbf{m}' = \mathbf{T}\,\mathbf{R}(\theta_m)\,\mathbf{m}. \tag{4.36}$$

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** Equation 4.35 with $\mathbf{R}(\theta_e)$:

$$\mathbf{m}' = \mathbf{R}(\theta_e)\,\mathbf{T}\,\mathbf{R}(\theta_m)\,\mathbf{m}. \tag{4.37}$$

This results in the correct animation (Figure 4.4, middle). Recall our discussion of the passive interpretation of transformations. This final multiplication by $\mathbf{R}(\theta_e)$ is treating the Moon's coordinates as being written with respect to a reference frame rotating with the Earth. Multiplying by $\mathbf{R}(\theta_e)$ 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, such as an arm, is not only a function of the angle it makes about its pivot (e.g. the 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. This is illustrated by Section 4.1.8 (right).

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 $\mathbf{T}_t$ 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 $\mathbf{R}_t$. We can think of the torso being specified about the origin of its own local "torso" reference frame, defined by $\mathbf{T}_t\,\mathbf{R}_t$. So if we consider vertices of the torso polygon $\mathbf{t}$, defined in that local frame, we would actually draw vertices $\mathbf{T}_t\,\mathbf{R}_t\,\mathbf{t}$, i.e. the coordinates of $\mathbf{t}$ 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 $\mathbf{T}_s$, and may rotate about its own origin (i.e. the shoulder) using a matrix $\mathbf{R}_s$. If we consider the upper-arm polygon $\mathbf{s}$ defined within its local reference frame, then its coordinates within the torso reference frame are $\mathbf{T}_s\,\mathbf{R}_s\,\mathbf{s}$. And its coordinates within the root (world) reference frame are $\mathbf{T}_t\,\mathbf{R}_t\,\mathbf{T}_s\,\mathbf{R}_s\,\mathbf{s}$.

So, just as with the Earth, we pre-multiplied by the matrix ($\mathbf{T}_t\,\mathbf{R}_t$) describing the torso's reference frame. Other, independent, matrices exist to describe the other upper-arm, legs and so on. The lower-arm $\mathbf{q}$'s position about elbow might be given by $\mathbf{T}_q\,\mathbf{R}_q\,\mathbf{q}$, and its absolute position therefore by $\mathbf{T}_t\,\mathbf{R}_t\,\mathbf{T}_s\,\mathbf{R}_s\,\mathbf{T}_q\,\mathbf{R}_q\,\mathbf{q}$, 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 in practice quickly become noticeable after 5 or 6 matrix multiplications. 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 visual computing; 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.

4.2 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. A 3D point $[x, y, z]^\top$ is written $[wx, wy, wz, w]^\top$ in homogeneous coordinates. Rigid body transformations therefore take the form of 4×4 matrices.

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

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.38}$$

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.39}$$

where s_x, s_y, s_z are the scale factors in the x-, y- and z-direction, respectively, and t_x, t_y, t_z are similarly the shifts in the x-, y- and z-direction, respectively. Rotation in 3D is slightly more complicated.

4.2.1 Rotation in 3D with 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 now 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 the special names "**roll**", "**pitch**" and "**yaw**", respectively. The following three matrices will perform a rotation by the angle θ clockwise in each of these directions respectively. Note that our definition of "clockwise" here assumes we are "looking along" the principal axis in the positive direction:

$$\mathbf{R}_{\mathrm{x}}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.40}$$

$$\mathbf{R}_{\mathrm{y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.41}$$

$$\mathbf{R}_{\mathrm{z}}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.42}$$

We can perform any rotation we desire in 3D using some angular quantity of roll, pitch and yaw. This would be performed using some sequence of $\mathbf{R}_{\mathrm{x}}$, $\mathbf{R}_{\mathrm{y}}$ and $\mathbf{R}_{\mathrm{z}}$, in the form of a compound matrix transformation. We will see in Section 4.2.3 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**.

4.2.2 Rotation about an arbitrary axis in 3D

In Section 4.1.7, we saw that rotation about an arbitrary point could be performed using a compound matrix transformation. Rotation about an arbitrary axis can be achieved using a generalisation of that process (see also Figure 4.5). The seven steps of this process are:

1. Given an arbitrary axis to rotate about, we first translate the 3D space using a 4×4 translation matrix $\mathbf{T}$ so that the axis passes through the origin.
2. 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 $\mathbf{R}_{\mathrm{x}}$ so that the arbitrary axis lies in the xz-plane.
3. We then perform a rotation about another axis, e.g. the y-axis (via $\mathbf{R}_{\mathrm{y}}$ to align the axis in the xz-plane with one of the principal axes such as the z-axis.
4. Now that the axis of rotation lies along one of the principal axes (e.g. the z-axis), we can apply $\mathbf{R}_{\mathrm{z}}$ to rotate the user-specified number of degrees θ.
5. Transform by the inverse of $\mathbf{R}_{\mathrm{y}}$.

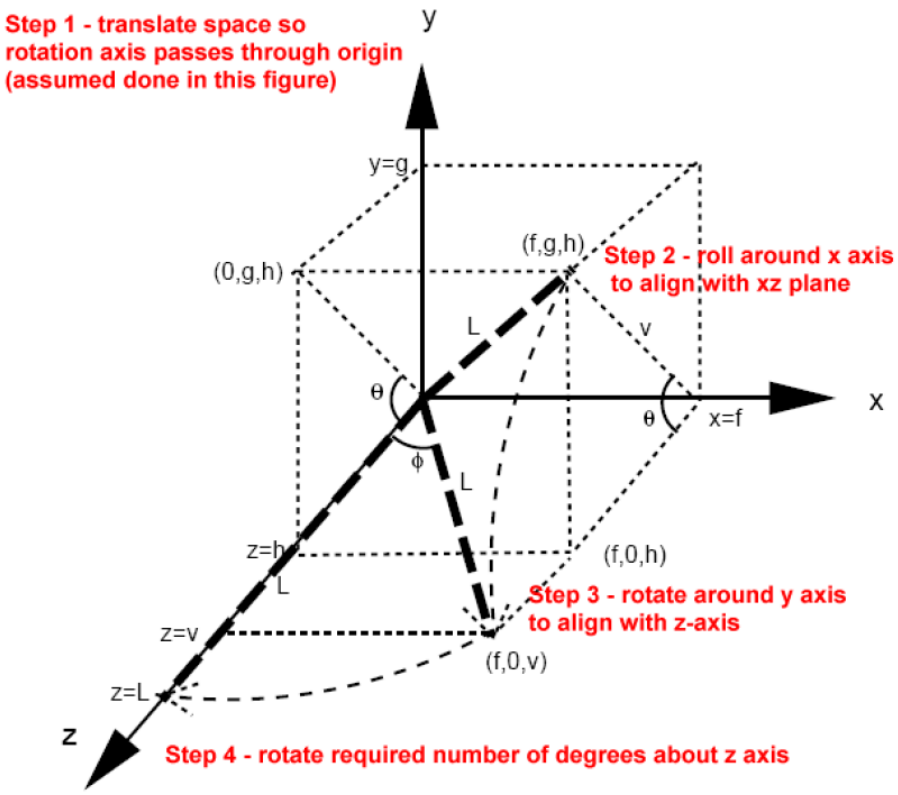

Figure 4.5: Illustrating the rotation of a point about an arbitrary axis in 3D – to be read in conjunction with the description in Section 4.2.2. The figure assumes the axis of rotation (L) already passes through the origin.

6. Transform by the inverse of $\mathbf{R}_x$.
7. Transform by the inverse of $\mathbf{T}$.

So the complete transformation is $\mathbf{p}' = \mathbf{T}^{-1}\,\mathbf{R}_x^{-1}\,\mathbf{R}_y^{-1}\,\mathbf{R}_z\,\mathbf{R}_y\,\mathbf{R}_x\,\mathbf{T}\,\mathbf{p}$. Note that we could also 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 principal axes (the z-axis). Only the translations (steps 1 and 7), and the actual rotation by the user–requested angle (step 4) have an effect.

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

Clearly the value of $\mathbf{R}_z$ is determined by the user, who wishes to rotate the model, and specifies a value of angle for the rotation. However, the values of $\mathbf{T}$, $\mathbf{R}_x$, $\mathbf{R}_y$ are determined by the equation of the line (axis) we wish to rotate around.

Let us suppose the axis of rotation (written $\mathbf{L}(s)$) has the following parametric equation (see

Equation 6.1 for the parametric equation of a line):

$$\mathbf{L}(s) = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + s \begin{bmatrix} f \\ g \\ h \end{bmatrix}. \tag{4.43}$$

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

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.44}$$

With the rotation axis $\mathbf{L}$ passing through the origin (i.e. step 1 of the 7 completed), we have the setup shown in Figure 4.5. To obtain a value for $\mathbf{R}_x$, we need to know the angle of rotation α, and for $\mathbf{R}_y$ we need to know the angle of rotation β. Fortunately, the rotation matrices require only the sines and cosines of these angles, and so we can use ratios in the geometry of Figure 4.5 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 $\mathbf{R}_x(\alpha)$.

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 $\mathbf{R}_y$.

4.2.3 Problems with Euler angles

Recall the discussion of compound matrix transformations in Section 4.1.7. 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 4.6), which in turn reveals some of the shortcomings 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. $\mathbf{R}_x$. 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. $\mathbf{R}_y$. Finally, consider a further mechanical frame surrounding the frame of $\mathbf{R}_y$, 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. $\mathbf{R}_z$. Figure 4.6 illustrates this mechanical setup, which is called a **gimbal**.

It is clear that the points on the plate are acted upon by $\mathbf{R}_x$, which is in turn acted upon by $\mathbf{R}_y$, which is in turn acted upon by $\mathbf{R}_z$. If we write the points on the plate as $\mathbf{p}$ then we have:

$$\mathbf{R_z}\,\mathbf{R_y}\,\mathbf{R_x}\,\mathbf{p}. \tag{4.45}$$

Note that we could have configured the system to use any ordering of the frames, e.g. $\mathbf{R_x}\,\mathbf{R_z}\,\mathbf{R_y}$, 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. $\mathbf{R}_y$ to 90°. The axis of rotation of $\mathbf{R}_x$ is made to line up with the axis of rotation of $\mathbf{R}_z$. We are no longer able to move in the direction that $\mathbf{R}_x$ 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**.

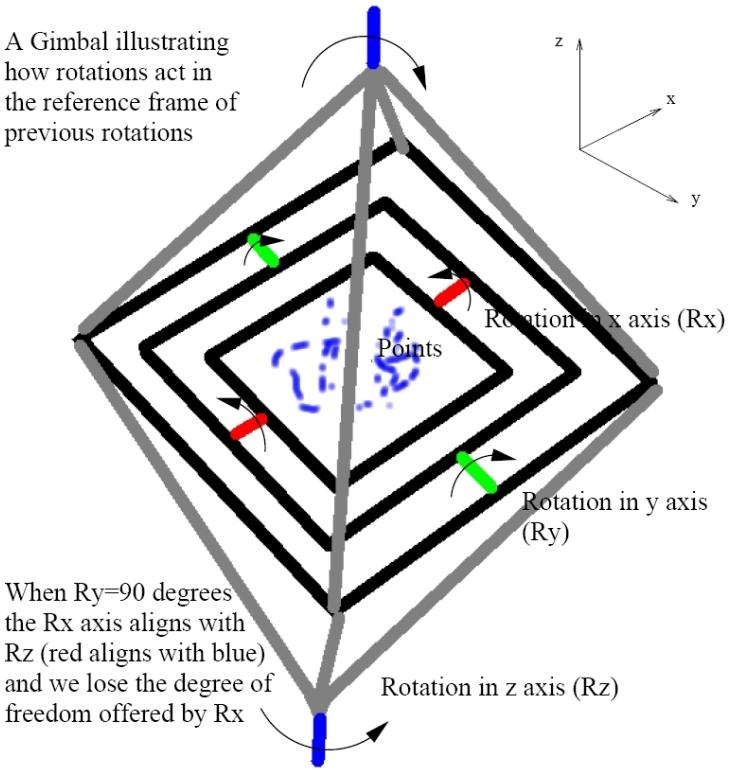

Figure 4.6: A mechanical gimbal, showing how points (defined in a space at the centre of the gimbal) are acted upon by rotation operations along the x-, y- and z-axes. Setting the middle gimbal at 90 degrees causes gimbal lock; the axes of rotation of the inner and outer frames become aligned.

This illustrates one of the major problems with the Euler angle parametrisation of rotation. Consideration of rotation as a roll, pitch and yaw component is quite intuitive, and can be useful in a graphics package interface for animators. But we see that without careful planning, we can steer 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. $\mathbf{R}_y$) or by adding a new reference frame outside the system, for example

$$\mathbf{R}_x{}' \, \mathbf{R}_z \, \mathbf{R}_y \, \mathbf{R}_x. \tag{4.46}$$

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 a textbook or other reference for more information.

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 (Section 4.1.8), 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 lower arm. The points of the lower arm $\mathbf{l}$ have the following location in the root reference frame:

$$\mathbf{T}_t \mathbf{R}_t \mathbf{T}_u \mathbf{R}_u \mathbf{T}_l \mathbf{R}_l \mathbf{l}. \tag{4.47}$$

Suppose the $\mathbf{R}$ in the above equation are specified by Euler angles (e.g. $\mathbf{R} = \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x$). Certain combinations of poses (e.g. where $\mathbf{R}_u$ 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. $\mathbf{l}$ 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 $\mathbf{T}_l \mathbf{R}_l$ we write $\mathbf{T}_l(\mathbf{K}_l^{-1} \mathbf{R}_l \mathbf{K}_l)$ or maybe even $\mathbf{K}_l^{-1}(\mathbf{T}_l \mathbf{R}_l)\mathbf{K}_l$ (note that the bracketing here is redundant and for illustration only). The final chain might end up as:

$$\mathbf{T}_t(\mathbf{K}_t^{-1} \mathbf{R}_t \mathbf{K}_t)\mathbf{T}_u (\mathbf{K}_u^{-1} \mathbf{R}_u \mathbf{K}_u)\mathbf{T}_l(\mathbf{K}_l^{-1} \mathbf{R}_l \mathbf{K}_l)\mathbf{l} \tag{4.48}$$

Suppose we had an articulated body where $\mathbf{R}_u$ was likely to be a rotation in the y-axis 90° due to the nature of the motion being modelled. We might specify a $\mathbf{K}_u$ 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. $\mathbf{R}_l$).

Non-sudden nature of gimbal lock

Recall Figure 4.6, where the system is in gimbal lock when $\mathbf{R}_y$ 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.

4.3 Projection – 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. This is a lossy operation that can also 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.

4.3.1 Perspective projection

You may already be familiar with the concept of perspective projection from your own visual experiences in the real world. Objects in the distance appear smaller that those nearby. One consequence is that parallel lines in the real world do not map to parallel lines in an image under perspective projection. Consider for example parallel train tracks (tracks separated by a constant distance)

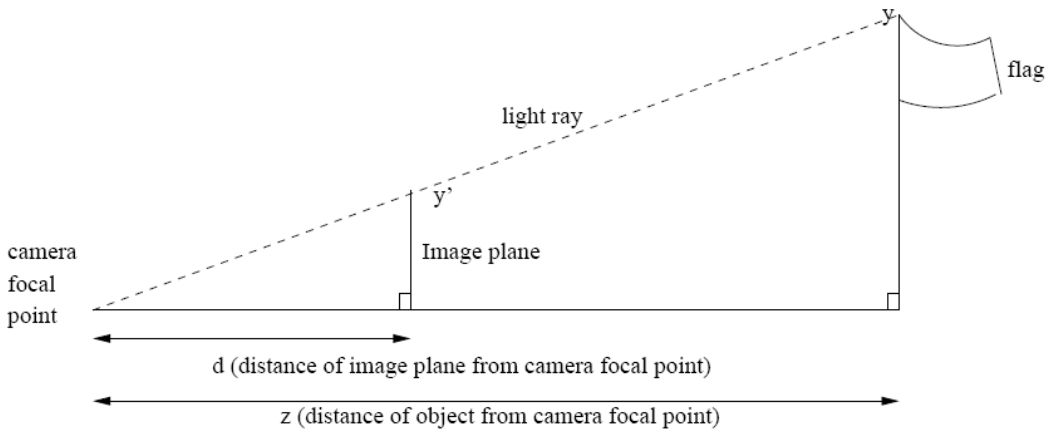

Figure 4.7: Diagram illustrating the perspective projection of a 3D flagpole to create a 3D image. Light rays travel in straight lines from points on the flagpole to a camera focal point, passing through a planar surface representing the image to be rendered.

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 visual computing, however. Whether a rendering is characterised as having n-point perspective has no bearing on the underlying mathematics that we use to model perspective.

We will now derive equations for modelling perspective projection. 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 4.7 illustrates this system, considering for clarity only the y-axis and z-axis of the 3D set-up. In this set-up, the tip of the flagpole is at distance 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:

$$\frac{z}{d} = \frac{y}{y'}, \tag{4.49}$$

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

$$y' = \frac{dy}{z}. \tag{4.50}$$

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.

$$x' = \frac{dx}{z}. \tag{4.51}$$

Thus the essence of perspective projection is division by the z-coordinate. This makes sense as the size of an image should be inversely proportional to the distance of and 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 "**pinhole camera**", which is an idealised camera similar to the *camera obscura* without any lens. These were the earliest form of camera, developed during the Renaissance, and most modern cameras use glass or plastic lenses leading to more complicated projection models as light is bent along its path from object to sensor. However, the pinhole camera assumption is acceptable for most visual computing applications.

The distance d between the camera focal point and the image plane is called the **focal length**. Larger focal lengths create images that 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, for example found in a **wide-angle lens**, accommodate larger fields of view.

Matrix transformation for perspective projection

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

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix} . \tag{4.52}$$

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 $[dx/z, dy/z, dz/z = d]^\top$. 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:

$$\mathbf{P} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} , \tag{4.53}$$

which leads to

$$\mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ z \end{bmatrix} . \tag{4.54}$$

A matrix with equivalent functionality is:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} . \tag{4.55}$$

Finally, note that any points "behind" the image plane will also be projected onto the image plane, and objects behind the camera will appear 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.

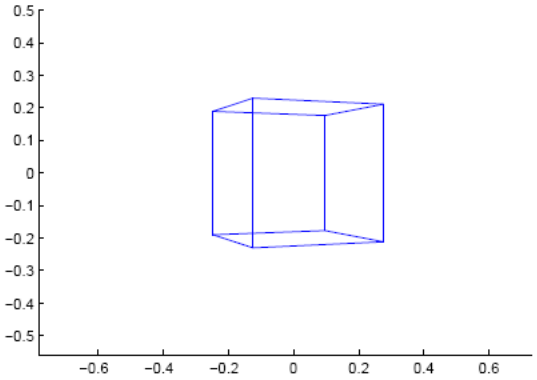

Figure 4.8: A spinning 3D cube is animated by manipulating its position in space using 3D rigid body transformations (4×4 matrices), and projected on the 2D screen using the perspective projection matrix.

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 $\mathbf{T}$ followed by a rotation $\mathbf{R}$) 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 $\mathbf{P}$.

The compound matrix transformation would be $\mathbf{P}\,\mathbf{R}\,\mathbf{T}$, 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. This is illustrated in Figure 4.8.

4.3.2 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 one of the coordinates, such as 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 (for 'top', 'front' and 'side' views). This is achievable using the matrix:

$$\mathbf{P} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.56}$$

4.4 Homographies

The final matrix transformation we shall discuss is the **homography**. A 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), where all points are in homogeneous form. It has several major applications in visual computing, 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 a homography may be computed between two sets of corresponding homogeneous 2D points $\mathbf{p}=[\mathbf{p}_1,\mathbf{p}_2,\ldots,\mathbf{p}_n]$ and $\mathbf{q}=[\mathbf{q}_1,\mathbf{q}_2,\ldots,\mathbf{q}_n]$, where for now $n=4$. For this section, we introduce the notation p_x^i to indicate the x-component of $\mathbf{p}_i$ (the i-th point in $\mathbf{p}$).

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

$$\begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} p_x^i \\ p_y^i \\ 1 \end{bmatrix} = \begin{bmatrix} wq_x^i \\ wq_y^i \\ w \end{bmatrix}. \tag{4.57}$$

Our task is to find all h_k ($k=1,\ldots,9$) to satisfy points $i=1,2,3,4$. We can rewrite Equation 4.57 out as individual linear equations:

$$h_1p_x^i+h_2p_y^i+h_3=wq_x^i, \tag{4.58}$$
$$h_4p_x^i+h_5p_y^i+h_6=wq_y^i, \tag{4.59}$$
$$h_7p_x^i+h_8p_y^i+h_9=w, \tag{4.60}$$

and substitute Equation 4.60 into Equations 4.58 and 4.59 to get

$$h_1p_x^i+h_2p_y^i+h_3=h_7p_x^iq_x^i+h_8p_y^iq_x^i+h_9q_x^i \tag{4.61}$$
$$h_4p_x^i+h_5p_y^i+h_6=h_7p_x^iq_y^i+h_8p_y^iq_y^i+h_9q_y^i \tag{4.62}$$

Rearranging these two equations, we get

$$h_1p_x^i+h_2p_y^i+h_3-h_7p_x^iq_x^i-h_8p_y^iq_x^i-h_9q_x^i=0 \tag{4.63}$$
$$h_4p_x^i+h_5p_y^i+h_6-h_7p_x^iq_y^i-h_8p_y^iq_y^i-h_9q_y^i=0 \tag{4.64}$$

We can then write these two equations as a homogeneous linear system (in the form $\mathbf{A}\,\mathbf{h}=\mathbf{0}$), which we repeat for every point $\mathbf{p}_i$:

$$\begin{bmatrix} p_x^1 & p_y^1 & 1 & 0 & 0 & 0 & -p_x^1q_x^1 & -p_y^1q_x^1 & -q_x^1 \\ 0 & 0 & 0 & p_x^1 & p_y^1 & 1 & -p_x^1q_y^1 & -p_y^1q_y^1 & -q_y^1 \\ p_x^2 & p_y^2 & 1 & 0 & 0 & 0 & -p_x^2q_x^2 & -p_y^2q_x^2 & -q_x^2 \\ 0 & 0 & 0 & p_x^2 & p_y^2 & 1 & -p_x^2q_y^2 & -p_y^2q_y^2 & -q_y^2 \\ & & & & & \vdots & & & \\ p_x^n & p_y^n & 1 & 0 & 0 & 0 & -p_x^nq_x^n & -p_y^nq_x^n & -q_x^n \\ 0 & 0 & 0 & p_x^n & p_y^n & 1 & -p_x^nq_y^n & -p_y^nq_y^n & -q_y^n \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \mathbf{0}. \tag{4.65}$$

We call $\mathbf{A}$ the **design matrix**, and it is completely defined by the point correspondences. The vector $\mathbf{h}$ is our solution vector (a vectorised version of the homography $\mathbf{H}$). We can use standard techniques such as **singular value decomposition** (SVD) to solve this system for $\mathbf{h}$, and so recover $h_{1\ldots9}$, i.e. the matrix $\mathbf{H}$. In brief, the SVD algorithm 'decomposes' $\mathbf{A}$ into three matrices such that

Figure 4.9: 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 Section 4.5). The two images were then merged together.

$\mathbf{A} = \mathbf{U}\,\mathbf{S}\,\mathbf{V}^\top$. In the context of this problem, we will treat SVD as a "black box"; the columns making up output $\mathbf{V}$ are all possible solutions for $\mathbf{h}$. The value in $S_{i,i}$ is the error for the solution in the i-th column of $\mathbf{V}$. For $n=4$ points, there will always be an exact solution. If $n>4$, then the homography $\mathbf{H}$ 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.

4.4.1 Applications to image stitching

We can apply the homography transformation to perform the popular task of "stitching" together two overlapping photographs of an object, such as two fragments of a panoramic mountain view. By "overlapping photographs" we mean two photographs that capture 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. This application is illustrated in Figure 4.9.

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, for example 4 corners of a doorway that is visible in both images. Writing these four points as $\mathbf{p}$ and $\mathbf{q}$, we can apply the mathematics of the previous section to compute the homography $\mathbf{H}$ between those points. Thus the homography describes an exact mapping from the coordinates of the doorway in I_1 (i.e. $\mathbf{p}$) to the coordinates of the doorway in I_2 (i.e. $\mathbf{q}$).

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 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 book "Multiple View Geometry" by Hartley and Zisserman [2004] for more details.

It turns out that very distant objects, such as mountains or 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 the homography $\mathbf{H}$. 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 perform

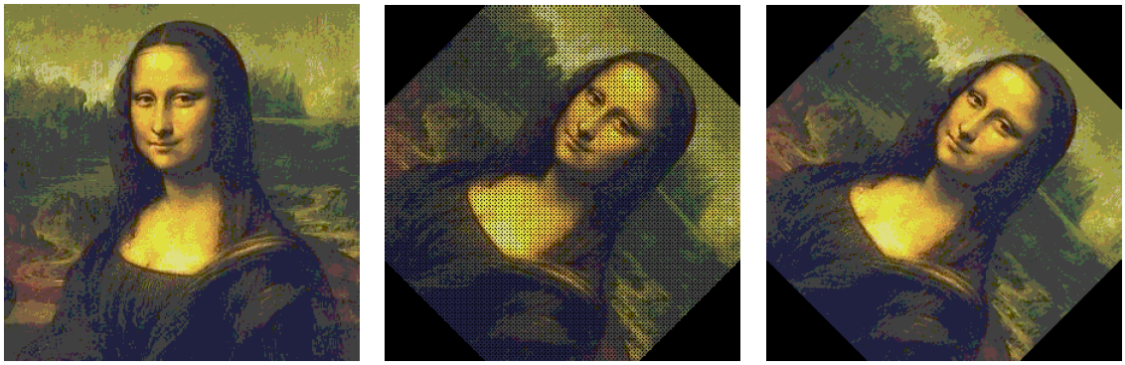

Figure 4.10: An image (left) is warped using forward mapping (middle) and backward mapping (right). Backward mapping substantially improves the quality compared to forward mapping.

this operation using standard image warping techniques – as discussed in Section 4.5. 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 overlaid to create a final panorama (or trivial colour blending operations applied to mitigate any inaccuracies in the mapping, which may for example be caused by violation of the plane-plane mapping assumption of the homography).

It is possible to obtain the 4 point correspondences ($\mathbf{p}$ and $\mathbf{q}$) automatically; most photo-stitching programs do not require manual identification of the matching points. We can use computer vision algorithms to identify stable "interest points" in images, i.e. points that we can reliably identify regardless of the point of view from which an image is taken. Often we use simple "corner detectors", such as the Harris corner detector [Harris and Stephens, 1988], or more robust features such as SIFT [Lowe, 2004]. The problem of finding 4 corresponding points is then reduced to finding 4 interest points in one image, and the matching 4 interest points in the other image. This search is usually performed stochastically via RANSAC [Fischler and Bolles, 1981] or a similar algorithm; the details are beyond the scope of this course.

4.5 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, for example 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 4.1.

The most straightforward and intuitive process for performing image warping is to iterate over the source image, treating each pixel as a 2D point $\mathbf{p} = [x, y]^\top$. We apply a matrix transformation $\mathbf{p}' = \mathbf{M}\,\mathbf{p}$ to those coordinates to determine where each pixel "ends up" in our target image, i.e. $p' = [x', y']^\top$. We then colour in the target image pixel at $\mathbf{p}'$ with the colour of pixel $\mathbf{p}$ in the source image. This process is called **forward mapping**, and example MATLAB code for it is given in Figure 4.11.

Unfortunately, there are a number of problems with forward mapping. First, the resultant target coordinates $\mathbf{p}$ are real-valued; however, pixels are addressed by integer coordinates, and so some sort of correction is usually made, such as rounding $\mathbf{p}$ to the nearest integer, so that we know which pixel to colour in. This creates aesthetically poor artefacts 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 $[0, 0]^\top$ maps to $[0, 0]^\top$ on the target,

```
source = im2double(imread('mona.jpg'));
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];
S = [2 0 0; 0 2 0; 0 0 1];

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

% The forward mapping loop: iterate over every source pixel.
for y = 1:size(source, 1)
    for x = 1:size(source, 2)

        % Transform source pixel location (round to pixel grid).
        p = [x; y; 1];
        q = M * p;
        u = round(q(1) / q(3));
        v = round(q(2) / q(3));

        % Check if target pixel falls inside the image domain.
        if (u > 0 && v > 0 && u <= size(target, 2) && v <= size(target, 1))
            % Sample the target pixel colour from the source pixel.
            target(v, u, :) = source(y, x, :);
        end

    end
end

imshow([source target]);
```

Figure 4.11: MATLAB code to warp an image using forward mapping.

point $[1,0]^\top$ maps to $[2,0]^\top$ on the target, $[2,0]^\top$ to $[4,0]^\top$ 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 4.10, middle).

A better solution is **backward mapping**. In backward mapping, we iterate over the target image, rather than the source image. For each pixel, we obtain integer coordinates $\mathbf{p}' = [x,y]^\top$, which we multiply by $\mathbf{M}^{-1}$ to obtain the corresponding pixel in the source image $\mathbf{p} = \mathbf{M}^{-1}\,\mathbf{p}$. 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 $\mathbf{p} = [x,y]^\top$ may not be integer-valued. We could simply round the pixel to the nearest integer; again this can create artefacts in the image. Nevertheless, the technique produces substantially improved results over forward mapping (Figure 4.10, compare right and middle).

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 $\mathbf{p}$. There are various strategies for this, such as bi-linear or bi-cubic interpolation.

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 may be the only solution if working with a non-invertible transformation.

Chapter 5

Basics of OpenGL programming

5.1 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 '**API**' or application programming interface), which can be invoked from your own programs. OpenGL has been ported to most platforms (from your computer to your smartphone) and programming languages, but this course will focus on the C programming language, the original OpenGL API. 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 most other languages, such as Java (using JOGL[1]) or Python (using PyOpenGL[2]).

OpenGL was originally developed by **Silicon Graphics (SGI)**, a company that pioneered many early movie special effects and computer 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 code. Indeed, there is often little code in an OpenGL library – the library often acts simply as an interface to the graphics driver in a machine. However, in cases where a machine has no specialised graphics hardware, an OpenGL library will often emulate its presence. This is for example the case in Microsoft Windows: if your computer has no specialised graphics hardware, OpenGL will imitate the hardware in software. The result will be visually (nearly) identical, but it may execute much more slowly. Another cue is the version number: Windows ships with OpenGL version 1.1 (released in 1997!), while the latest version is 4.5 (as of November 2016).

Note: This chapter describes OpenGL 1 functionality that has been discontinued since OpenGL 3. This is for demonstration purposes and to simplify the coursework compared to modern OpenGL.

5.1.1 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 requires substantial amounts of boilerplate code for even the simplest

[1] http://jogamp.org/jogl/www/
[2] http://pyopengl.sourceforge.net/

applications. For this reason, the **GLUT** library (OpenGL utility toolkit) 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.

5.2 Illustrative example: drawing a teapot

Any programming course starts with an illustrative example; in graphics courses, it is traditional to draw a teapot, specifically the Utah teapot[3]. Here is the OpenGL and GLUT code to draw a teapot:

```
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenGL/gl.h>
#include <GLUT/glut.h>
#else
#ifdef _WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glut.h>
#endif

void keyboard(unsigned char key, int x, int y)
{
    switch(key) {
        case 27: exit(0);
    }
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    //glTranslatef(1, 0, 0);  // example translation

    glutSolidTeapot(.5);

    glFlush();
    glutSwapBuffers();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);

    glutInitWindowSize(600, 600);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

    glutCreateWindow("Hello, teapot!");
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    glutMainLoop();
}
```

[3] https://en.wikipedia.org/wiki/Utah_teapot

There are three functions in this piece of 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 within our own code that GLUT will invoke when certain events happen. In this case, GLUT will invoke function the 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. There are also additional callbacks for 'reshaping' a window (which is invoked whenever a window is resized), and for mouse interaction (invoked by any button press/release as well as mouse motion), but we are not using these here.

The most crucial part of the '`main(...)`' function is the last line: the call to '`glutMainLoop()`'. This passes all control in our program to GLUT. GLUT will never return from this call, so frankly it is pointless to write 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 the 'Esc' key is 27; if it is pressed, we call the previously mentioned system '`exit(...)`' function to halt our program.

The '`display(...)`' function clears the screen and draws our teapot. There are a couple of lines referring to matrices – OpenGL uses matrices exactly as we did in Chapter 4 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 – GLUT may be the only library in the world to contain such a function for drawing a teapot. 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 commands in more detail in the following section.

5.2.1 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 passed to '`glutInit`'). In single-buffered mode, anything we draw goes straight to the screen, i.e. it is written directly to the framebuffer (recall Section 2.1.2) linked to the screen. In double-buffered mode, we still have a framebuffer for the screen, but also a 'spare' buffer. We draw onto the 'spare' buffer in our '`display(...)`' function, and then swap the screen and spare framebuffers with a call to '`glutSwapBuffers()`'. We would usually prefer double-buffering when creating an animation in OpenGL. This is because drawing directly to the screen (i.e. using single-buffering) can cause flickering, for instance when moving or resizing a window.

The call to '`glFlush()`' prior to '`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 framebuffers (the calls to swap buffers are not queued). Therefore we 'flush' the queue for good practice prior to '`glutSwapBuffers()`'.

5.3 Modelling and matrices in OpenGL

The 3D matrix transformation theory described in Chapter 4 can be put into practice using OpenGL, to create scaling, rotation and perspective projection effects.

Recall from Chapter 4 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 $\mathbf{T}$ or rotations $\mathbf{R}$). The vertices move, and thus so do the surfaces anchored to them. Finally, we apply a perspective transformation (for example $\mathbf{P}$) to project our vertices from 3D to 2D, and draw the 2D points on the screen. For example, to project some homogeneous 3D vertices $\mathbf{p}$ into homogeneous 2D image locations $\mathbf{p}'$:

$$\mathbf{p}' = \mathbf{P}\,\mathbf{R}\,\mathbf{T}\,\mathbf{p}. \tag{5.1}$$

In OpenGL, the projection matrix (i.e. $\mathbf{P}$) is stored in a 'variable' called the '`PROJECTION`' matrix. The rigid-body transformations $\mathbf{R}$, $\mathbf{T}$ (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. In our example above, we used another helper function '`glLoadIdentity()`' to load the identity matrix ($\mathbf{I}$) into the '`PROJECTION`' matrix. A subsequent call to '`gluPerspective(...)`' would for example create a perspective matrix $\mathbf{P}$, and post-multiply it with the current contents of the '`PROJECTION`' matrix.

Similarly, inside the '`display(...)`' function, we first overwrite the '`MODELVIEW`' matrix with the identity matrix $\mathbf{I}$. We could then use other OpenGL helper functions to post-multiply $\mathbf{I}$ 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 (x, y, z).
glRotatef(theta, x, y, z) // Rotation matrix rotating theta degrees
                          // clockwise about vector (x, y, z).
glScalef(x, y, z)         // Scale matrix with scale factors (x, y, 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(...)`'.

5.3.1 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 last-in-first-out (LIFO) stack data structure. This is very useful when drawing articulated bodies, which are often represented as a hierarchy (tree) of reference frames, and traversed recursively (see Section 4.1.8 on animation

5.4 Simple animation: a 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 <stdlib.h>

#ifdef __APPLE__
#include <OpenGL/gl.h>
#include <GLUT/glut.h>
#else
#ifdef _WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glut.h>
#endif

// Global variable for teapot rotation.
static int g_angle = 0;

void keyboard(unsigned char key, int x, int y)
{
    switch(key) {
        case 27: exit(0);
    }
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glRotatef(g_angle, 0, 0, 1);

    glutSolidTeapot(.5);

    glFlush();
    glutSwapBuffers();
}

void idle()
{
    g_angle = (g_angle + 1) % 360;
    glutPostRedisplay(); // trigger call to display(...)
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);

    glutInitWindowSize(600, 600);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

    glutCreateWindow("Hello, teapot!");
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutIdleFunc(idle);

    glutMainLoop();
}
```

The key features of this code are:

- The use of the global variable '`g_angle`' to store the current rotation angle. 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 call to '`display(...)`' – which draws the teapot at the current angle.

In this case, the counter is simply used as an input to '`glRotatef`' ('`f`' for using '`float`' numbers) 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, which will vary greatly from machine to machine. We could use OS-specific calls to govern the animation's speed. For example, under Windows, we could insert a '`Sleep(milliseconds)`' system call.

Chapter 6

Geometric modelling

This chapter is about geometric modelling, and by '**modelling**' we mean "forming a mathematical representation of the world". The output of this process might be a representation (or **model**) of anything from an object's shape, to the trajectory of an object through space. In Chapter 4, we learned how to manipulate points in space using matrix transformations. Points are important because they form the basic building blocks for representing models in visual computing. 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 **curves**. Curves are nothing more than trajectories through space, for example in 2D or in 3D. Technically, a line is also a curve, but a special case that happens to be straight. For avoidance of doubt, when we talk about curves in this chapter, we are referring to curves in general, which also includes straight lines.

6.1.1 Explicit, implicit and parametric forms

We are all familiar with the equation $y = mx + c$ for specifying a straight 2D 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 curve using $y = f(x)$, as 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 cannot represent all curves in the form $y = f(x)$. Specifically, we cannot represent curves that cross a vertical line $x = a$, where a is a constant, more than once. Figure 6.1 (left) shows an example of a curve which we could not model using the $y = f(x)$ form. This is the more general explanation behind our observation that not all straight lines can be represented by $y = f(x)$.

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

So suppose we want to model a line, *any line*. An alternative way to do this is using the **parametric**

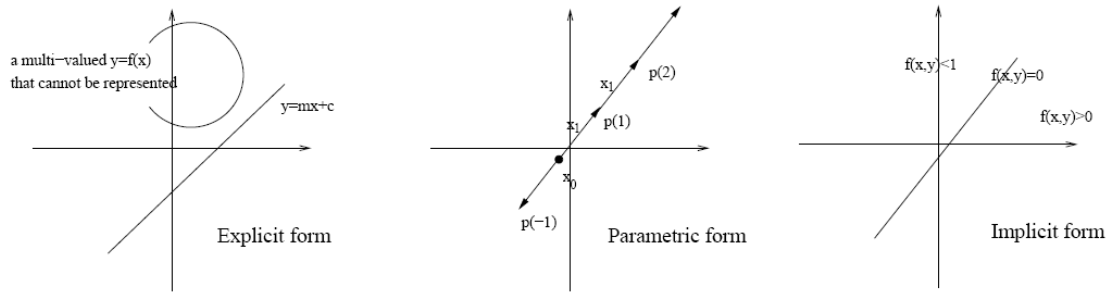

Figure 6.1: The explicit, parametric and implicit forms of a line.

form:

$$\mathbf{p}(s) = \mathbf{x}_0 + s\,\mathbf{x}_1. \tag{6.1}$$

Here, the vector $\mathbf{x}_0$ indicates the starting point of the line, and $\mathbf{x}_1$ is a vector representing the positive direction of the line. We have introduced the parameter s as a mechanism for tracing alone the line: given a value s, we obtain a point $\mathbf{p}(s)$ 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 $\mathbf{p}(0)$ to denote the start of a curve, and $\mathbf{p}(1)$ 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 $\mathbf{p}(s)$ as the position of a particle flying through space, and s as analogous to time. Then $\mathbf{x}_0$ is the start position of the particle, and $\mathbf{x}_1$ is the velocity of the particle. We can even add a term $\mathbf{x}_2$ for acceleration as follows:

$$\mathbf{p}(s) = \mathbf{x}_0 + s\,\mathbf{x}_1 + s^2\,\mathbf{x}_2 \tag{6.2}$$

to yield a *quadratic* curve. We can continue adding terms to obtain a polynomial curve of degree n:

$$\mathbf{p}(s) = \sum_{i=1}^{n} s^i\,\mathbf{x}_i, \tag{6.3}$$

although 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. To give another example of a curve represented in parametric form, a more convenient parametrisation of a circle might be:

$$\mathbf{p}(\theta) = \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}. \tag{6.4}$$

The final representation of 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, which is equated to a constant (often zero). For example, in 2D, implicit forms generally look like $f(x,y)=0$. 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 $\mathbf{x}_0=[x_0,y_0]^\top$ and $\mathbf{x}_1=[u,v]^\top$:

$$x = x_0 + s\,u, \tag{6.5}$$
$$y = y_0 + s\,v. \tag{6.6}$$

After rearranging both equations for s, we get

$$\frac{x - x_0}{u} = s = \frac{y - y_0}{v}, \tag{6.7}$$

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

$$(x - x_0)v - (y - y_0)u = 0, \tag{6.9}$$

where x_0, y_0, u and v are the parameters 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 (see Figure 6.1, right).

The explicit, parametric and implicit forms of a 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 for example useful in computer games to perform '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 cannot be used to easily determine this.

The parametric form allows us to easily trace along the path of a curve; this is useful in a very wide range of applications, such as ray tracing or modelling of object trajectories. Again, it is hard to trace along 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 line drawing algorithm, an integer-only algorithm for drawing straight lines on raster displays. You can look this up in the course textbook [Shirley et al., 2009, Section 8.1.1] or on Wikipedia[1], but this discussion is beyond the scope of this course.

6.1.2 Parametric curves

Parametric curves are by far the most common form of curve in computer graphics. This is because: (a) they generalise to any dimension as the $\mathbf{x}_i$ 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.

Suppose 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 many times to produce the complex outline of the teapot, implying a very high order curve. Such a curve might perhaps be of degree $n = 20$ or more, and hence requiring the same number of $\mathbf{x}_i$ 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 even though though such curves are manageable in practice, we tend to over-fit the curve.

[1] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

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 the degree of the polynomial) increases.

Therefore, we usually model complex shapes by breaking them up into simpler curves, and fitting each of these in turn. It is common practice in modelling to use cubic curves, which provide a compromise between ease of control and expressiveness of shape:

$$\mathbf{p}(s) = \mathbf{x}_0 + s\,\mathbf{x}_1 + s^2\,\mathbf{x}_2 + s^3\,\mathbf{x}_3. \tag{6.10}$$

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

$$\mathbf{p}(s) = \begin{bmatrix}\mathbf{x}_3 & \mathbf{x}_2 & \mathbf{x}_1 & \mathbf{x}_0\end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.11}$$

$$= \mathbf{C}\,\mathbf{Q}(s), \tag{6.12}$$

where $\mathbf{C}$ is a matrix containing the $\mathbf{x}_i$ vectors that control the shape of the curve, and $\mathbf{Q}(s)$ contains the parametrisation 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 carefully consider the nature of the join. We might like to join the curves together in a 'smooth' way so that no kinks are visible; 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 modelling, 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 endpoint of the first curve – $\mathbf{p}_1(1)$ – is the starting point of the second curve – $\mathbf{p}_2(0)$ –, 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 $\mathbf{p}_1'(0)$ matches the tangent at the start of the second curve $\mathbf{p}_2'(1)$, then we say the curves have first order or C^1 continuity. This is because both their zero-th derivatives (i.e. position) and first-order derivatives (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.

This idea of continuity extends trivially to higher orders (C^n), although greater than C^2 continuity is rarely useful in practice.

G^n refers to 'geometric continuity', a weaker definition of 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. it should be entirely transparent to the user.

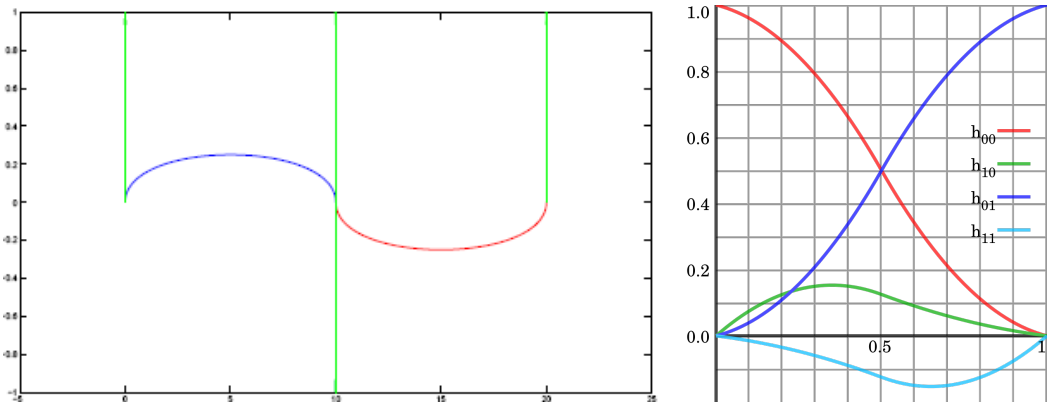

Figure 6.2: Left: Example of two cubic Hermite curves (in blue and red) joining with C^1 continuity; tangents indicated with long green arrows. Right: A plot of the blending functions ($\mathbf{M}\,\mathbf{Q}(s) = [h_{00}(s), h_{01}(s), h_{10}(s), h_{11}(s)]^\top$) for Hermite curves (source: Berland, ElectroKid/Wikipedia).

6.2 Families of curves

Equation 6.12 defined a curve using the form $\mathbf{p}(s) = \mathbf{C}\,\mathbf{Q}(s)$, where the matrix $\mathbf{C}$ comprises four vectors (columns) that determine the shape of the curve. Recall that using the physical analogy of a particle $\mathbf{p}$ 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 directly. 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 the matrix $\mathbf{C}$ into two matrices $\mathbf{G}$ and $\mathbf{M}$ as follows:

$$\mathbf{p}(s) = \mathbf{G}\,\mathbf{M}\,\mathbf{Q}(s). \tag{6.13}$$

We call $\mathbf{M}$ the **blending matrix** and $\mathbf{G}$ the **geometry matrix**. By introducing a particular 4×4 matrix $\mathbf{M}$, we can change the meaning of the vectors in $\mathbf{G}$, making curve control more intuitive, as we shall see in a moment. Note that when $\mathbf{M}$ is the identity, then the vectors in $\mathbf{G}$ have the same function as in our original $\mathbf{C}\,\mathbf{Q}(s)$ formulation (Equation 6.12). Informally, we say that the matrix $\mathbf{M}$ defines the family of curves we are working with. We will now look at four curve families, each of which uses a different matrix $\mathbf{M}$.

6.2.1 Hermite curves

The cubic Hermite curve has the form (relate this to Equation 6.13):

$$\mathbf{p}(s) = \begin{bmatrix}\mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1)\end{bmatrix} \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.14}$$

Here we can see that the blending matrix $\mathbf{M}$ 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 $\mathbf{G}$. 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 (see Figure 6.2).

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, blending matrix $\mathbf{M}$ as specified above. We change only the geometry matrix $\mathbf{G}$ to create curves of different shapes. As before, the matrix $\mathbf{Q}(s)$ controls where we are along the curve's trajectory, i.e. we substitute in a particular parameter value s between 0 and 1 to evaluate our position along the curve $\mathbf{p}(s)$.

Derivation of the blending matrix $\mathbf{M}$ for Hermite curves

We have yet to explain the values of the 16 scalar constants comprising $\mathbf{M}$. 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 Equation 6.13 with respect to s. Only matrix $\mathbf{Q}(s)$ has terms dependent on s, and so is the only matrix that changes:

$$\mathbf{p}'(s) = \mathbf{G}\,\mathbf{M}\begin{bmatrix}3s^2\\2s\\1\\0\end{bmatrix}. \tag{6.15}$$

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

$$\begin{bmatrix}\mathbf{p}(s) & \mathbf{p}'(s)\end{bmatrix} = \mathbf{G}\,\mathbf{M}\begin{bmatrix}s^3 & 3s^2\\ s^2 & 2s\\ s & 1\\ 1 & 0\end{bmatrix}. \tag{6.16}$$

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

$$\begin{bmatrix}\mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1)\end{bmatrix} = \begin{bmatrix}\mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1)\end{bmatrix}\mathbf{M}\begin{bmatrix}s^3 & s^3 & 3s^2 & 3s^2\\ s^2 & s^2 & 2s & 2s\\ s & s & 1 & 1\\ 1 & 1 & 0 & 0\end{bmatrix} \tag{6.17}$$

$$\begin{bmatrix}\mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1)\end{bmatrix} = \begin{bmatrix}\mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1)\end{bmatrix}\mathbf{M}\begin{bmatrix}0 & 1 & 0 & 3\\ 0 & 1 & 0 & 2\\ 0 & 1 & 1 & 1\\ 1 & 1 & 0 & 0\end{bmatrix}. \tag{6.18}$$

Note that Equation 6.18 is obtained from Equation 6.17 by setting $s=0$ (for the start point of the curve) in columns 1 and 3, and $s=1$ (for the end point of the curve) in columns 2 and 4 in the right-most matrix.

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

$$\mathbf{I} = \mathbf{M} \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \tag{6.19}$$

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}^{-1}, \tag{6.20}$$

which, when evaluated, produces the 4×4 blending matrix $\mathbf{M}$ used in Equation 6.14.

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

6.2.2 Bézier curves

The Bézier curve is an example of an **approximating curve**. It is specified using coordinates of four **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, for example, to a workbench at each end, and masses are 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 sometimes used in modelling to describe any curve we can shape to our needs by specifying spatial control points; however, the term 'spline' is most frequently used in piecewise modelling to describe pieced-together functions or curves. The formulation of the Bézier curve is:

$$\mathbf{p}(s) = \begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{6.21}$$

The points $\mathbf{p}_0$ and $\mathbf{p}_3$ are the start and end points of the curve, respectively. The points $\mathbf{p}_1$ and $\mathbf{p}_2$ are the approximated points (see Figure 6.3). The **convex hull** (i.e. convex polygon) formed by those four points is 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 in which $\mathbf{p}_1$ and $\mathbf{p}_2$ 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 just as easily enforce C^1 continuity with Bézier curves. We do this by ensuring $\mathbf{p}_2$ on the first curve, and $\mathbf{p}_1$ on the second curve are co-linear and equidistant from the join point $\mathbf{p}_3$ / $\mathbf{p}_0$ (see Figure 6.3).

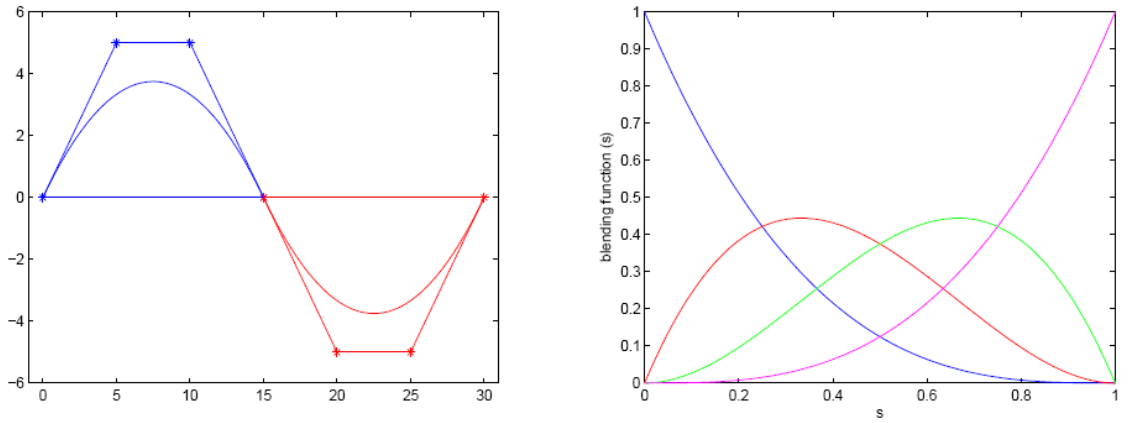

Figure 6.3: Left: Diagram of two Bézier curves (in blue and red), joined with C^1 continuity (control polygons also indicated). Note the two interpolated and two approximated points on each curve, and the co-linearity of points at the join to generate the C^1 continuity. Right: A plot of the blending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

The blending matrix $\mathbf{M}$ 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 Equation 6.21 is a rearrangement of the Bernstein polynomials:

$$\mathbf{p}(s) = \sum_{i=0}^{n} \mathbf{p}_i \frac{n!}{i!(n-1)!} s^i (1-s)^{n-i}, \tag{6.22}$$

where, in our cubic curve, $n = 3$ and $\mathbf{p}_{0\ldots3}$ are the four vectors comprising the geometry matrix $\mathbf{G}$. 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 $\mathbf{M}$ as the blending matrix. We can see that, during matrix multiplication, the values in the matrix blend the contribution of each control point in $\mathbf{G}$ to the summation producing $\mathbf{p}(s)$, i.e.:

$$\mathbf{M}\,\mathbf{Q}(s) = \begin{bmatrix} 1s^0(1-s)^3 \\ 3s^1(1-s)^2 \\ 3s^2(1-s)^1 \\ 1s^3(1-s)^0 \end{bmatrix} = \begin{bmatrix} (1-s)^3 \\ 3s\ (1-s)^2 \\ 3s^2(1-s) \\ s^3 \end{bmatrix} = \begin{bmatrix} -s^3+3s^2-3s+1 \\ 3s^3-6s^2+3s \\ -3s^3+3s^2 \\ s^3 \end{bmatrix} \tag{6.23}$$

$$\mathbf{p}(s) = \mathbf{G}\,\mathbf{M}\,\mathbf{Q}(s) = \begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{bmatrix} \begin{bmatrix} -s^3+3s^2-3s+1 \\ 3s^3-6s^2+3s \\ -3s^3+3s^2 \\ s^3 \end{bmatrix} \tag{6.24}$$

Figure 6.3 (right) plots the blending function in each of the four rows of $\mathbf{M}\,\mathbf{Q}(s)$ as s increases from 0 to 1. We see that when $s=0$, the weighting is entirely towards point $\mathbf{p}_0$. As s increases, the weighting towards that point decreases whilst the weight towards $\mathbf{p}_1$ increases (although this term never completely dominates the contribution to $\mathbf{p}(s)$). As s increases further still, greater weighting is attributed to $\mathbf{p}_2$, and finally all weighting is towards $\mathbf{p}_3$.

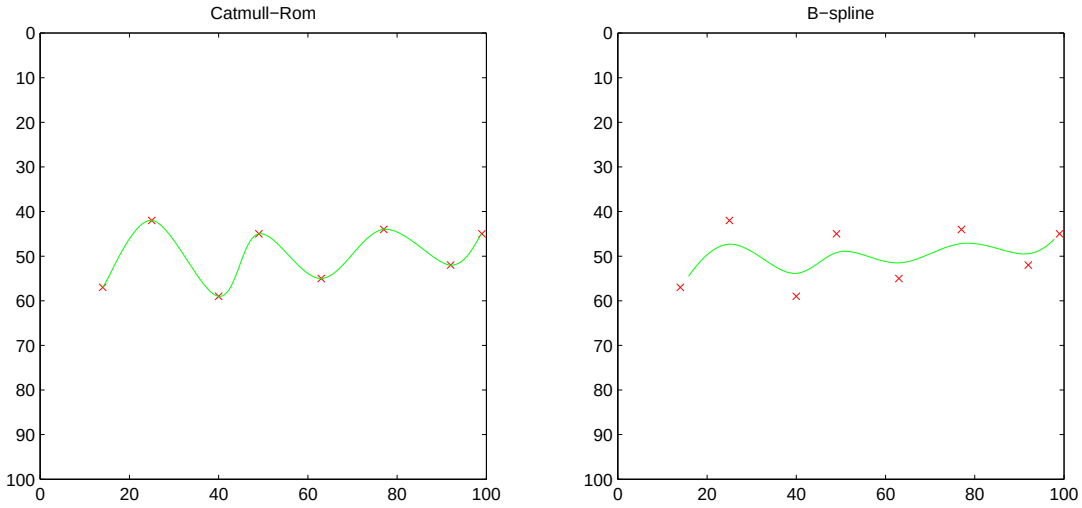

Figure 6.4: Eight points used to define the geometry of a Catmull–Rom spline (left) and a B-spline (right). Note that the Catmull-Rom spline interpolates all points, whereas the B-spline approximates all points.

6.2.3 Catmull–Rom splines

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 the points it should pass through, rather than those to approximate.

The Catmull–Rom spline is specified through the framework of Equation 6.13 as follows:

$$\mathbf{p}(s) = \begin{bmatrix}\mathbf{a} & \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{b}\end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix}, \tag{6.25}$$

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

In fact, by evaluating Equation 6.25 from $s=0$ to $s=1$, we will obtain points only for the section of curve between the second and third vectors in $\mathbf{G}$ i.e. the points indicated $\mathbf{p}(0)$ and $\mathbf{p}(1)$. The points $\mathbf{a}$ and $\mathbf{b}$ only help to shape the path of the curve. We must use several Catmull–Rom curves to trace out a piecewise trajectory (spline) in full.

An an example, consider six points we wish to interpolate with a piecewise Catmull-Rom cubic curve. We write these points $\mathbf{p}_i$ where $i=1\ldots6$. We can find the path of a curve through all the points by evaluating several curve segments $\mathbf{p}_j(s) = \mathbf{G}_j \mathbf{M} \mathbf{Q}(s)$ equations with the following geometry matrices $\mathbf{G}_j$ where $j=1\ldots5$:

$$\mathbf{G}_1 = \begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3\end{bmatrix}, \tag{6.26}$$
$$\mathbf{G}_2 = \begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 & \mathbf{p}_4\end{bmatrix}, \tag{6.27}$$
$$\mathbf{G}_3 = \begin{bmatrix}\mathbf{p}_2 & \mathbf{p}_3 & \mathbf{p}_4 & \mathbf{p}_5\end{bmatrix}, \tag{6.28}$$
$$\mathbf{G}_4 = \begin{bmatrix}\mathbf{p}_3 & \mathbf{p}_4 & \mathbf{p}_5 & \mathbf{p}_6\end{bmatrix}, \tag{6.29}$$
$$\mathbf{G}_5 = \begin{bmatrix}\mathbf{p}_4 & \mathbf{p}_5 & \mathbf{p}_6 & \mathbf{p}_6\end{bmatrix}. \tag{6.30}$$

So if we plot the $\mathbf{p}_j(s)$ generated by geometry matrix $\mathbf{G}_j$ plugged into Equation 6.25, we will

interpolate between points $\mathbf{p}_j$ and $\mathbf{p}_{j+1}$. Note how points are allocated to $\mathbf{G}_j$ in a 'rolling' manner, and also duplicated at the beginning and end of the piecewise curve. See also Figure 6.4.

6.2.4 B-spline

The **B-spline** is very similar in function to the Catmull–Rom spline. It is also specified using a 'rolling' form of geometry matrix $\mathbf{G}_j$, 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:

$$\mathbf{p}(s) = \begin{bmatrix}\mathbf{a} & \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{b}\end{bmatrix} \frac{1}{6} \begin{bmatrix}-1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0\end{bmatrix} \begin{bmatrix}s^3 \\ s^2 \\ s \\ 1\end{bmatrix}. \tag{6.31}$$

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 in this course, but interested readers may refer to textbooks [Hughes et al., 2013; Shirley et al., 2009] for further details on these and other types of spline.

6.3 Surfaces

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

6.3.1 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:

$$\mathbf{p}(s,t) = \mathbf{x}_0 + s\,\mathbf{u} + t\,\mathbf{v}. \tag{6.32}$$

Figure 6.5 illustrates this; we have a 2D parameter space (s,t) that enables us to span a 3D surface. The origin of that parameter space is at $\mathbf{x}_0$. The vectors $\mathbf{u}$ and $\mathbf{v}$ 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 the parameters s and t. Later we will return to the parametric equation of a plane when we discuss texture mapping.

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

$$(\mathbf{p} - \mathbf{c}) \cdot \hat{\mathbf{n}} = 0. \tag{6.33}$$

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

$$n_x(x - c_x) + n_y(y - c_y) + n_z(z - c_z) = 0, \tag{6.34}$$

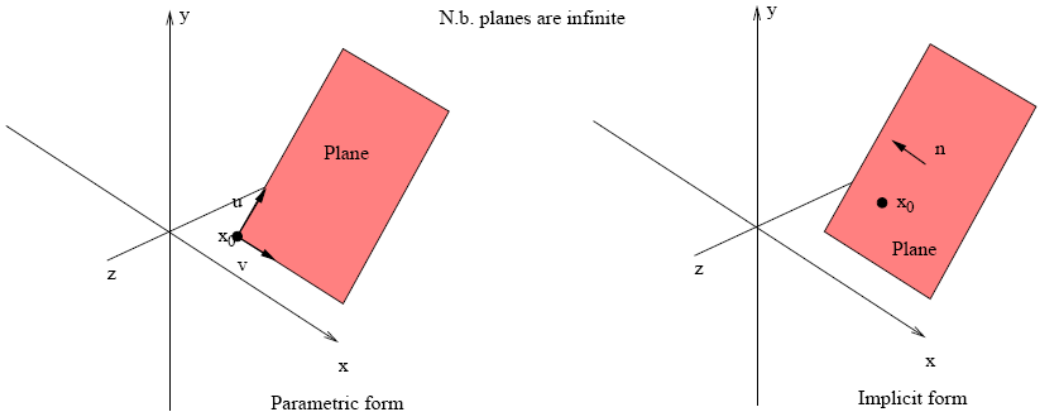

Figure 6.5: Illustrating the parametric and implicit forms of a plane.

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

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

$$z = ax + bx + c, \tag{6.35}$$

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

6.3.2 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 $f(x, y, z) = 0$,

$$x^2 + y^2 + z^2 - r^2 = 0. \tag{6.36}$$

Notice that the left-hand side of the equation is positive if a point is outside the sphere, and negative if it is inside. To obtain an explicit form of the 3D sphere, we can solve for z:

$$z = \sqrt{r^2 - x^2 - y^2}. \tag{6.37}$$

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, we only obtain a half-sphere in the positive z-domain (Figure 6.6).

We can also form a parametric representation of a sphere. We commonly use such a representation every day, for example in satellite 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):

$$x(\theta, \phi) = \cos\theta \cos\phi, \tag{6.38}$$
$$y(\theta, \phi) = \sin\theta \cos\phi, \tag{6.39}$$
$$z(\theta, \phi) = \sin\phi. \tag{6.40}$$

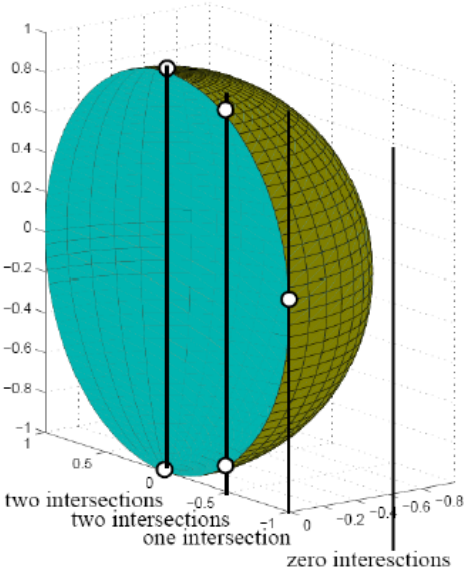

Figure 6.6: Rearranging the implicit form of a sphere into an explicit form prevents modelling of the entire spherical surface; the positive root for $z^2 = x^2 + y^2$ produces a half-sphere in $z > 0$.

6.3.3 Bi-cubic surface patches

Earlier, in Section 6.1.2, we saw how cubic splines can be used to model shapes in piecewise fashion. This idea generalises to 3D 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:

$$\mathbf{p}(s) = \begin{bmatrix}\mathbf{g}_1 & \mathbf{g}_2 & \mathbf{g}_3 & \mathbf{g}_4\end{bmatrix} \mathbf{M}\,\mathbf{Q}(s), \tag{6.41}$$

where $\mathbf{g}_{1\ldots4}$ 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 $i = 1 \ldots 4$,

$$\mathbf{g}_i(t) = \mathbf{H}_i\,\mathbf{M}\,\mathbf{Q}(t), \tag{6.42}$$

where $\mathbf{H_i} = [\mathbf{h}_{i,1}, \mathbf{h}_{i,2}, \mathbf{h}_{i,3}, \mathbf{h}_{i,4}]$ – introducing the notation $\mathbf{h}_{i,j}$ to denote the j^{th} control point on the i^{th} curve. So the control points for Equation 6.41 are defined by four independent curves, commonly parameterised by t. A position on the surface is thus defined in 2D parameter space (s, t); hence the term **bi-cubic surface patch**. Recall that these parameters are typically defined in range $s, t \in [0, 1]$.

We can combine Equations 6.41 and 6.42 to express the surface in a single equation. We first rewrite the equation as

$$\mathbf{g}_i(t)^\top = \mathbf{Q}(t)^\top\,\mathbf{M}^\top \begin{bmatrix}\mathbf{h}_{i,1} & \mathbf{h}_{i,2} & \mathbf{h}_{i,3} & \mathbf{h}_{i,4}\end{bmatrix}^\top. \tag{6.43}$$

And then, with some minor abuse of notation (to avoid introducing tensor notation), we can write:

$$\mathbf{p}(s,t) = \mathbf{Q}(t)^\top\,\mathbf{M}^\top \begin{bmatrix}\mathbf{h}_{1,1} & \mathbf{h}_{2,1} & \mathbf{h}_{3,1} & \mathbf{h}_{4,1} \\ \mathbf{h}_{1,2} & \mathbf{h}_{2,2} & \mathbf{h}_{3,2} & \mathbf{h}_{4,2} \\ \mathbf{h}_{1,3} & \mathbf{h}_{2,3} & \mathbf{h}_{3,3} & \mathbf{h}_{4,3} \\ \mathbf{h}_{1,4} & \mathbf{h}_{2,4} & \mathbf{h}_{3,4} & \mathbf{h}_{4,4}\end{bmatrix} \mathbf{M}\,\mathbf{Q}(s), \tag{6.44}$$

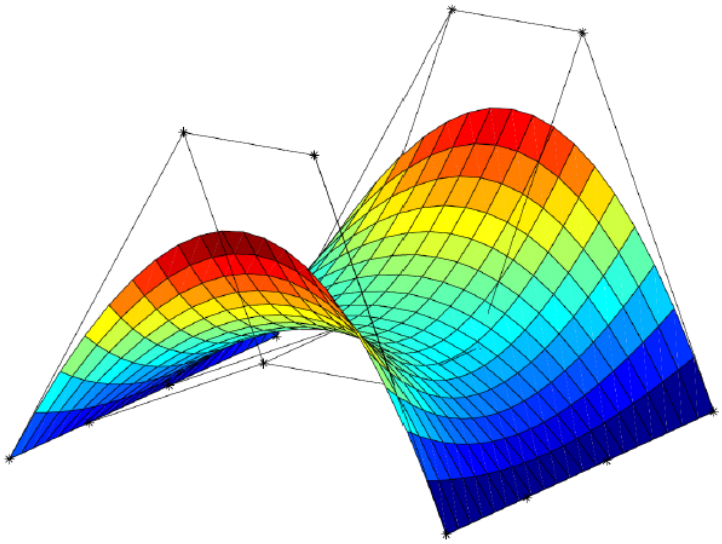

Figure 6.7: Illustration of a bi-cubic Bézier surface patch. The 16 control points $\mathbf{h}_{i,j}$ of the surface are indicated in black, and determine the geometry of the surface.

where each element of the 4×4 matrix $\mathbf{h}_{i,j}$ is the j^{th} geometry vector of $\mathbf{g}_i$ from Equation 6.42.

If $\mathbf{M}$ was the blending matrix for the Bézier curve family, then each $\mathbf{h}_{i,j}$ would be a point in space. The volume bounded by the convex hull of those points would contain the bi-cubic Bézier surface patch (Figure 6.7).

Consideration of continuity

We can join patches together with C^n continuity, much as with curves. For example, we can define adjacent bi-cubic Bézier patches to share control points, to enforce C^0 or C^1 continuity.

It is possible to use the Hermite blending matrix to define bi-cubic Hermite 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 turns out that C^1 continuity is very important when modelling objects. The lighting calculations used to determine the colour of a surface is generally a function of the surface's normal. If the normal changes suddenly at the surface interface, for example 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, leading to highly noticeable and undesirable edge artefacts. This effect can be also observed when modelling objects with triangular patches (planes) that are too large.

Index

Bibliography

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24 (6):381–395, 1981. ISSN 0001-0782. doi: 10.1145/358669.358692.

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1990. ISBN 0201121107.

Chris Harris and Mike Stephens. A combined corner and edge detector. In *Proceedings of the Alvey Vision Conference*, pages 147–151, 1988.

Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004. ISBN 0521540518. URL http://www.robots.ox.ac.uk/~vgg/hzbook/.

John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley, 3rd edition, July 2013. ISBN 0321399528.

David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

Peter Shirley, Stephen Marschner, Michael Ashikhmin, Michael Gleicher, Naty Hoffman, Tamara Munzner, Erik Reinhard, Kelvin Sung, William B. Thompson, Peter Willemsen, and Brian Wyvill. *Fundamentals of Computer Graphics*. CRC Press, 3rd edition, 2009. ISBN 1568814690. URL https://www.cs.cornell.edu/~srm/fcg3/.