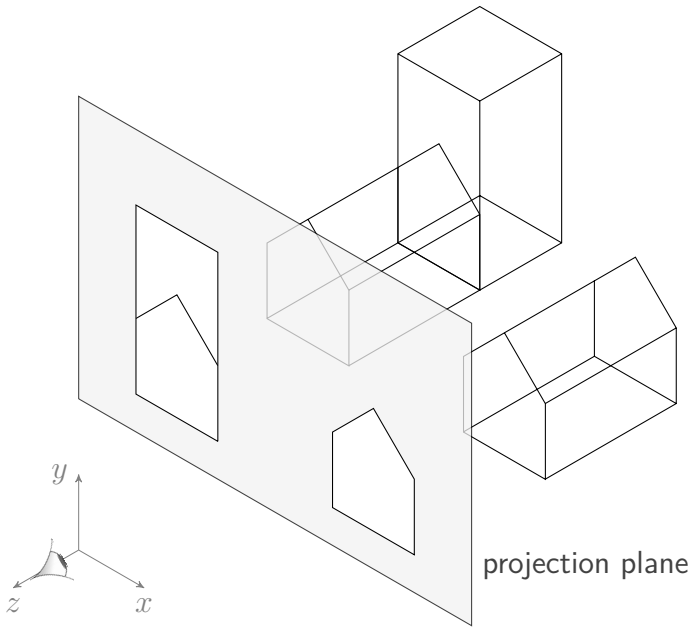
Manchester Metropolitan University

Computational Mathematics

Computer Graphics Lecture Notes

Dr Jon Shiach & Dr Killian O'Brien

Spring 2022



Contents

This version compiled on April 25, 2022.

Chapter 0

Preliminaries

0.1 Learning and Teaching

0.1.1 Lecture notes

These are the lecture notes that accompany the computer graphics part of the unit Computational Mathematics with the other part covering computing mathematics. Students are provided with a printed copy of the lecture notes so that they can focus on what is being covered in the lecture without having to worrying about making their own notes. There is also an electronic copy in PDF format which is available on the moodle area for this unit. These notes are quite comprehensive and are written specifically for this unit so should serve as your main point of reference. However, it is always advisable to seek out other sources of information either on the internet or better still textbooks from the library. Mathematical notation can differ between authors and these notes have been written to use notation that is most commonly found online.

These notes use a tried and tested format of definition (what is it that we are studying?) $\rightarrow$ explanation (why do we use it?) $\rightarrow$ examples (how is it used?) $\rightarrow$ exercises (now you try it). The examples in the printed version of the notes are left empty for students to complete in class. This is done because by writing out the steps used in a method it helps students to better understand that method. The PDF version of the notes contains the full solutions to the examples so if you do happen to miss an example you can complete the example by looking it up in the PDF version.

You should read through the lecture notes prior to attending the lectures that focus on that particular chapter. Don't worry about trying to understand everything when you first read through them. Reading mathematics is not like reading your favourite novel, it often requires repeated reading of a passage before you fully grasp the concepts that are being explained. In the lectures we will explain the various topics and provide more insight than what is written in the notes.

The PDF version of the notes contains lots of hyperlinks for easy navigation around the document. Hyperlinks show up as blue text and can save lots of time scrolling up and down the pages.

0.1.2 Unit timetable

The timetable for the computing mathematics part of the unit is shown below (this can also be accessed via MyMMU). The material will be covered in one 2-hour lecture and one 2-hour lab.

- Lecture: Mondays 12:00 - 14:00 in JD E249
- Lab: Mondays 15:00 - 17:00 in JD C2.04

Students are expected to devote at least 30 hours per week to their studies and should complete all reading and tutorial exercises during this time.

Note that due to the 2nd May being a bank holiday monday, the lecture and lab will be moved to the following times on Wednesday 4th May:

- Lecture: 09:00 – 11:00 in E145
- Lab: 11:00 – 13:00 in C3.03

0.1.3 Teaching Schedule

The computing mathematics material will be covered over the 6 weeks of the teaching block as outlined in table 1 so you should be aware of what is being covered and when. There may be times when we have to deviate from this slightly and you will be informed of this in the lecture.

Table 1: Computer graphics teaching schedule

Week	Date (w/c)	Material
1	14/03/2022	Chapter 1 Vector Geometry: co-ordinate systems; vectors; equations of points, line and planes; distance calculations.
2	21/03/2022	Chapter 2 Translation, Rotation and Scaling Transformations: linear transformations; translation, scaling and rotation transformations; transformation matrices. **Coursework assignment handed out**
3	28/03/2022	Chapter 3 Virtual Environments: the viewing pipeline; defining objects; building the world space; aligning to the camera space; 3D to 2D projections; projecting onto the screen space.
4	04/04/2022	Chapter 4 Clipping: Sutherland-Hodgman algorithm; intersection between lines and planes.
		Easter Break
5	25/04/2021	Chapter 4 Hidden Surface Removal: back face culling; painters algorithm; binary space partitioning.
6	02/05/2021	Revision. **Coursework assignment deadline – 6th May 2022**
7	02/05/2022	Assessment week
		Examination – 9th May to 10th May 2022

0.2 Assessment

The assessment for this unit takes the form of two coursework assignments and a take home examination.

- **Coursework (20%) – Computing Mathematics**
  - Released to students on **Monday 21st March 2022**;
  - Deadline is **9pm Friday 6th April 2022**;
- **Coursework (20%) – Computer Graphics**
  - Released to students on **Monday 21st March 2022**;
  - Deadline is **9pm Friday 6th May 2022**;
- **Examination (60%)**
  - Released to students at **09:00 on Tuesday 10th May 2022**;
  - Deadline is **12:30 on 9pm Friday 6th May 2022**;

  - Students will have access to the lecture notes, unit materials on moodle and any other sources of information available;
  - Total of 4 questions, 2 questions on computer graphics and 2 questions on computing mathematics.

You will receive a percentage mark for each assessment component and your overall mark for the unit will be calculated using the simple equation

$$\text{unit mark} = 0.2 \times \text{coursework 1 mark} + 0.2 \times \text{coursework 2 mark} + 0.6 \times \text{examination mark}.$$

To successfully pass the unit you will need a unit mark of at least 40%.

0.3 Advice to students

Some general advice to students:

- **Attend all of the classes** – the key to successfully passing the unit is to attend all of your classes. Mathematics is not a subject that can be learned easily in isolation just by reading the lecture notes. You will get much more out of the unit by attending, and more importantly, actively engaging in the classes.
- **Complete all of the exercises** – you would not expect an athlete to get faster or stronger without exercising and the same applies to studying mathematics. The tutorial exercises are designed to give you practice at using the various techniques and to help you to fully grasp the content. Try to make sure that you complete all of the exercises before the following week's lectures. Full worked solutions are provided in **??** at the end of these notes but do try to be disciplined and avoid looking up the answers before you have attempted the questions.
- **Catch up on missed work** – for whatever reason there may be a day when you cannot attend your classes, if this happens make sure you make the effort to catch up on missed work. Read through the appropriate chapter in the notes (as specified in the teaching schedule), complete the examples and attempt the tutorial exercises. It is very easy to start falling behind and the longer you leave it the more difficult it will be to catch up.
- **Start the coursework as soon as you are able** – the coursework is released to students early in the teaching block so you are not expected to be able to answer all of the questions right away. As we progress through the unit you will be told which questions you should now be able to answer. Try to start these questions as soon as you can and not leave it to the last minute.
- **Ask questions!** – perhaps the most important piece of advice here, there will be times when you are not quite sure about a concept, application or question. You can ask for help from the teaching staff and your fellow students. Mathematics is a hierarchical subject which means it requires full understanding at a fundamental level before moving onto more advanced topics, so if there are any gaps in your knowledge don't be afraid to ask questions.

Chapter 1

Vector Geometry

1.1 Co-ordinate systems

A **co-ordinate system** is a system that uses a sequence of numbers to determine the position of an object in Euclidean space. The numbers are known as **co-ordinates** and are usually represented in an ordered set of numbers enclosed in parenthesis known as a **tuple**. The most common example of a co-ordinate system is the **Cartesian co-ordinate system** (named after René Descartes) which uses perpendicular number lines to define points in the space. Consider fig. 1.1 where three number lines called **axes** (singular: axis) are labelled x, y and z. The position of a point in this space is defined by the co-ordinates (x, y, z) where x, y and z are the distances along the axes from zero. Since these values are real numbers a three-dimensional space is often denoted by $\mathbb{R}^3$.

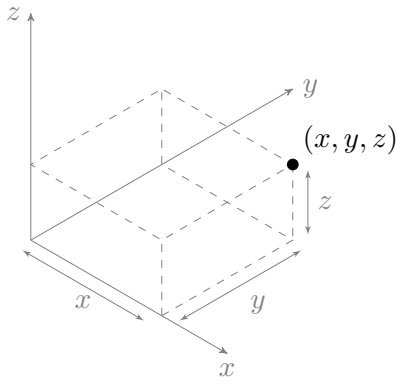


Figure 1.1: The three-dimensional Cartesian co-ordinate system.

1.1.1 Homogeneous co-ordinates

Homogeneous co-ordinates are a system of co-ordinates that include an additional value in the tuple. The use of homogeneous co-ordinates is standard practice in computer graphics since they allow us to apply operations such as translation, scaling, rotation and projection using matrix multiplication. The Cartesian co-ordinates (x, y, z) are represented using homogeneous co-ordinates as (wx, wy, wz, w) where w is some scalar quantity. Note that when $w = 1$ the homogeneous co-ordinates are $(x, y, z, 1)$.

1.2 Vectors

A vector is an object that has length and direction. In mathematical notation vectors are denoted in print using a boldface character, e.g., $\mathbf{a}$ or as an arrow over a character, e.g., $\vec{a}$ or underlined when handwritten, e.g., $\underline{a}$. A vector is defined by the signed distance along each axis by a tuple. For example, let $\mathbf{a}$ be a vector

in $\mathbb{R}^3$ defined by the tuple $\mathbf{a} = (a_x, a_y, a_z)$ where $a_x, a_y, a_z \in \mathbb{R}$, then $\mathbf{a}$ can be represented geometrically as the

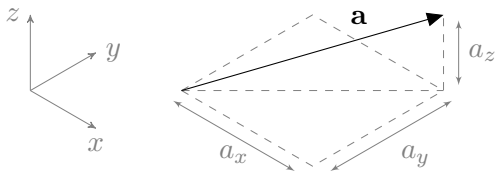


Figure 1.2: The vector $\mathbf{a} = (a_x, a_y, a_z)$.

1.2.1 Matrix representation

The tuple representing a vector can be written as either a matrix consisting of a single row or a single column. So a vector in $\mathbb{R}^3$ can be represented as

$$\mathbf{a} = (a_x, a_y, a_z) = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}.$$

These representations are called **row vector** and **column vector** respectively.

1.2.2 Vector magnitude

The length of a vector $\mathbf{a}$ is known as the **magnitude** and is denoted using $|\mathbf{a}|$.

Definition 1.1: Vector magnitude

The vector magnitude of a vector in $\mathbb{R}^n$, $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ is calculated using.

$$|\mathbf{a}| = \sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}. \tag{1.1}$$

Note that $|\mathbf{a}| > 0$.

Example 1.1

Calculate the magnitude of the vector $\mathbf{a} = (3, 4, 0)$.

Solution:

$$|\mathbf{a}| = \sqrt{3^2 + 4^2 + 0} = \sqrt{25} = 5.$$

1.2.3 Unit vectors

A **unit vector** is denoted by $\hat{\mathbf{a}}$ (referred to as '$\mathbf{a}$ hat') is a vector parallel to $\mathbf{a}$ with a magnitude of 1.

Definition 1.2: Normalising a vector

The unit vector $\hat{\mathbf{a}}$ that is parallel to the vector $\mathbf{a}$ can be calculated using

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|}, \tag{1.2}$$

this is known as **normalising a vector**.

Example 1.2

Calculate a unit vector that is parallel to $\mathbf{a} = (3, 4, 0)$.

Solution:

$$\hat{\mathbf{a}} = \frac{(3,4,0)}{5} = \left(\frac{3}{5}, \frac{4}{5}, 0\right).$$

Checking that $|\hat{\mathbf{a}}| = 1$

$$|\hat{\mathbf{a}}| = \sqrt{\left(\frac{3}{5}\right)^2 + \left(\frac{4}{5}\right)^2 + 0^2} = \sqrt{\frac{9}{25} + \frac{16}{25}} = \sqrt{1} = 1.$$

1.2.4 Vector addition and subtraction

The addition of two vectors is achieved by adding the correspond elements in the tuples. Let $\mathbf{a} = (a_1, a_2, a_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ then the sum $\mathbf{a} + \mathbf{b}$ is calculated by

$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}.$$

Similarly the subtraction of two vectors is achieved by subtracting the corresponding element in the tuples, i.e.,

$$\mathbf{a} - \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{bmatrix}.$$

In geometry, the vector addition $\mathbf{a} + \mathbf{b}$ is achieved by placing the tail of $\mathbf{b}$ at the head of $\mathbf{a}$. The resulting vector points from the tail of $\mathbf{a}$ to the head of $\mathbf{b}$. The vector subtraction $\mathbf{a} - \mathbf{b}$ is achieved by reversing the direction of $\mathbf{b}$ and placing the tail at the head of $\mathbf{a}$.

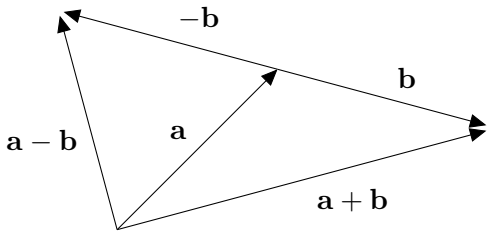


Figure 1.3: The addition and subtraction of two vectors.

1.2.5 Multiplying a vector by a scalar

The scalar multiple of a vector $\mathbf{a} = (a_1, a_2, a_3)$ by the scalar k is defined by

$$k\mathbf{a} = \begin{bmatrix} ka_1 \\ ka_2 \\ ka_3 \end{bmatrix}.$$

The effect of multiplying a vector by a scalar is that the magnitude of the vector is scaled by the value of the scalar. If $k > 0$ then the direction of the vector $k\mathbf{a}$ is the same as $\mathbf{a}$ whereas if $k < 0$ then the vector $k\mathbf{a}$ points in the opposite direction to $\mathbf{a}$ (fig. 1.4).

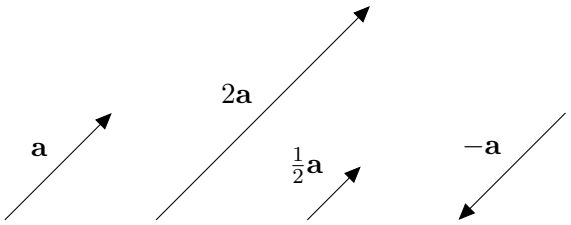


Figure 1.4: Scalar multiples of the vector $\mathbf{a}$.

1.2.6 The dot product

The product of two vectors can be calculated in two ways: the **dot product** and the **cross product**. The dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} \cdot \mathbf{b}$ and returns a scalar quantity and the dot product is often referred to as the **scalar** product.

Definition 1.3: Geometric definition of the dot product

The geometric definition of the dot product of two vectors, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, is

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\theta), \tag{1.3}$$

where θ is the angle between the two vectors (fig 1.5).

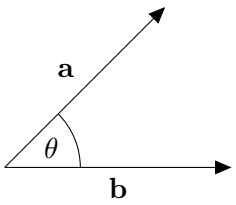


Figure 1.5: The two vectors $\mathbf{a}$, $\mathbf{b}$ and the angle between them θ is related by the dot product.

The value of a dot product can be computed using the algebraic definition of the dot product

Definition 1.4: Algebraic definition of the dot product

The dot product of two vectors $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ in $\mathbb{R}^n$ is defined by

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n. \tag{1.4}$$

For vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c} \in \mathbb{R}^n$ the dot product has the following properties

- commutative: $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$;
- distributive: $\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$;
- orthogonal: the non-zero vectors $\mathbf{a}$ and $\mathbf{b}$ are orthogonal (perpendicular) if $\mathbf{a} \cdot \mathbf{b} = 0$.

Example 1.3

(i) Calculate the dot product of the two vectors $\mathbf{a} = (3, 4, 0)$ and $\mathbf{b} = (5, 12, 0)$.

(ii) Calculate the angle between the two vectors $\mathbf{a} = (3, 4, 0)$ and $\mathbf{b} = (5, 12, 0)$.

(iii) Two orthogonal vectors are $\mathbf{a} = (1, 2, 3)$ and $\mathbf{b} = (4, x, 6)$. Determine the value of x.

Solution:

(i)

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 12 \\ 0 \end{bmatrix} = 3 \times 5 + 4 \times 12 + 0 \times 0 = 15 + 48 = 63.$$

(ii)

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\theta)$$
$$\therefore \theta = \cos^{-1}\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right) = \cos^{-1}\left(\frac{63}{5 \times 13}\right) = 0.2487.$$

(iii)

$$0 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ x \\ 6 \end{bmatrix} = 4 + 2x + 18,$$
$$\therefore x = -11.$$

1.2.7 The cross product

The **cross product** of two vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} \times \mathbf{b}$ and returns a vector that is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$ (fig. 1.6).

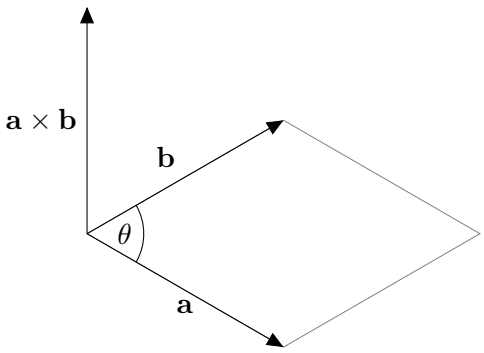


Figure 1.6: The cross product of the two vectors $\mathbf{a}$ and $\mathbf{b}$ produces a vector that is perpendicular to the plan that $\mathbf{a}$ and $\mathbf{b}$ lie on.

Definition 1.5: Geometric definition of the cross product

The cross product of two vectors, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$, is defined by

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}| \sin(\theta)\hat{\mathbf{n}}, \tag{1.5}$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$ and $\hat{\mathbf{n}}$ is a unit vector perpendicular to both $\mathbf{a}$ and $\mathbf{b}$.

The cross product of two vectors in $\mathbb{R}^3$ can be computed using the determinant formula.

Definition 1.6: Determinant formula for computing a cross product

The cross product of two vectors, $\mathbf{a} = (a_x, a_y, a_z)$ and $\mathbf{b} = (b_x, b_y, b_z)$, is computed using

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_z \end{bmatrix}. \tag{1.6}$$

For vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c} \in \mathbb{R}^3$ and $k \in \mathbb{R}$ the cross product has the following properties:

- $\mathbf{a} \times \mathbf{a} = \mathbf{0}$;
- $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$;
- not commutative: $\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$;
- distributive: $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$;
- scalar multiplication: $k\mathbf{a} \times \mathbf{b} = \mathbf{a} \times k\mathbf{b} = k(\mathbf{a} \times \mathbf{b})$.

Example 1.4

Calculate the cross product of the two vectors $\mathbf{a} = (3, 4, 0)$ and $\mathbf{b} = (1, 2, 3)$.

Solution:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 3 & 4 & 0 \\ 1 & 2 & 3 \end{vmatrix} = (4 \times 3 - 0 \times 2)\mathbf{i} - (3 \times 3 - 0 \times 1)\mathbf{j} + (3 \times 2 - 4 \times 1)\mathbf{k}$$
$$= \begin{bmatrix} 12 \\ -9 \\ 2 \end{bmatrix}.$$

We can check that this vector is perpendicular to $\mathbf{a}$ and $\mathbf{b}$ using the dot product, e.g.,

$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{a} = \begin{bmatrix} 12 \\ -9 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} = 36 - 36 + 0 = 0,$$
$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{b} = \begin{bmatrix} 12 \\ -9 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 12 - 18 + 6 = 0.$$

1.2.8 Vector basis

A **basis** is of a vector space V is the set of vectors $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$ which are linearly independent (i.e., no vector in the basis can be represented as a linear combination of the other vectors in the basis) and span V. Every other vector $\mathbf{v}$ in V can be expressed as a unique linear combination of the vectors in the basis, i.e.,

$$\mathbf{v} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \cdots + a_n\mathbf{e}_n.$$

where a_i are scalars. The **dimension** of a vector space V is the number of vectors in the basis. For example, the basis for the Euclidean space $\mathbb{R}^3$ is commonly denoted as $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ where

$$\mathbf{i} = (1, 0, 0), \qquad \mathbf{j} = (0, 1, 0), \qquad \mathbf{k} = (0, 0, 1),$$

so $\mathbb{R}^3$ is known as a three-dimensional Euclidean space. If $U = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ and $W = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n\}$ are two different bases of the same vector space V and $[\mathbf{u}]_W$ denotes that the vector $\mathbf{u}$ is expressed with respect to the basis W then

$$[\mathbf{u}_1]_W = a_{11}\mathbf{w}_1 + a_{21}\mathbf{w}_2 + \cdots + a_{n1}\mathbf{w}_n \tag{1.7}$$
$$[\mathbf{u}_2]_W = a_{12}\mathbf{w}_1 + a_{22}\mathbf{w}_2 + \cdots + a_{n2}\mathbf{w}_n \tag{1.8}$$
$$\vdots$$
$$[\mathbf{u}_n]_W = a_{1n}\mathbf{w}_1 + a_{2n}\mathbf{w}_2 + \cdots + a_{nn}\mathbf{w}_n, \tag{1.9}$$

where a_{ij} are scalars. If $\mathbf{v} \in V$ where $[\mathbf{v}]_U = (v_1, v_2, \ldots, v_n)$ then

$$\begin{aligned}
[\mathbf{v}]_W &= v_1[\mathbf{u}_1]_W + v_2[\mathbf{u}_2]_W + \cdots + v_n[\mathbf{u}_n]_W \\
&= v_1(a_{11}\mathbf{w}_1 + a_{21}\mathbf{w}_2 + \cdots a_{n1} + \mathbf{w}_n) + v_2(a_{12}\mathbf{w}_1 + a_{22}\mathbf{w}_2 + \cdots + a_{n2}\mathbf{w}_n) + \cdots \\
&\quad v_n(a_{1n}\mathbf{w}_1 + a_{2n}\mathbf{w}_2 + \cdots + a_{nn}\mathbf{w}_n) \\
&= (a_{11}v_1 + a_{12}v_2 + \cdots + a_{1n}v_n)\mathbf{w}_1 + (a_{21}v_1 + a_{22}v_2 + \cdots + a_{2n}v_n)\mathbf{w}_2 + \cdots \\
&\quad (a_{n1}v_1 + a_{n2}v_2 + \cdots + a_{nn}v_n)\mathbf{w}_n.
\end{aligned}$$

We can express this using a matrix equation $[\mathbf{v}]_W = A_{U\to W}[\mathbf{v}]_U$ where

$$[\mathbf{v}]_W = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

and $A_{U\to W}$ is square matrix here is known as the **change of basis matrix**. To determine $A_{U\to W}$ we need to solve eqs. (1.7) to (1.9) which can be done by performing Gaussian elimination on the augmented matrix

$$[\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n | \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n].$$

Note that its easy to show that $A_{W\to U} = A^{-1}_{U\to W}$.

Example 1.5

U and W are two basis of a vector space V given by

$$U = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}, \qquad W = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\},$$

and $\mathbf{v} = (3, 2)$ is a vector in V.

(i) Show that U and W are valid basis;

(ii) determine $[\mathbf{v}]_U$ and $[\mathbf{v}]_W$;

(iii) calculate the change of basis matrix $A_{U\to W}$;

(iv) use the change of basis matrix to express $[\mathbf{v}]_W$ with respect to basis W.

Solution:

(i) To show a basis is valid we need to show that the basis vectors are linearly independent, i.e., we need to show that the following linear system does not have non-zero solutions

$$a_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + a_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \mathbf{0}.$$

Using Gaussian elimination

$$\left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & 0 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right],$$

so the solution is $a_1 = a_2 = 0$ and U is a valid basis. Doing similar for W

$$\left[\begin{array}{cc|c} 0 & -1 & 0 \\ 1 & 1 & 0 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & -1 & 0 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & 0 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right],$$

so W is also a valid basis.

(ii) For $[\mathbf{v}]_U$ we need to solve

$$\alpha_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \mathbf{v}.$$

Using Gaussian elimination

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 0 & 1 & 2 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 2 \end{array}\right],$$

so $[\mathbf{v}]_U = (1, 2)$. Doing similar for $[\mathbf{v}]_W$

$$\left[\begin{array}{cc|c} 0 & -1 & 3 \\ 1 & 1 & 2 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & -1 & 3 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & -3 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & -3 \end{array}\right],$$

so $[\mathbf{v}]_W = (5, -3)$.

(iii) We need to solve

$$\begin{aligned} a_{11}\mathbf{w}_1 + a_{21}\mathbf{w}_2 &= \mathbf{u}_1, \\ a_{12}\mathbf{w}_1 + a_{22}\mathbf{w}_2 &= \mathbf{u}_2. \end{aligned}$$

Using Gaussian elimination

$$\left[\begin{array}{cc|cc} 0 & -1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{array}\right] \longrightarrow \left[\begin{array}{cc|cc} 1 & 1 & 0 & 1 \\ 0 & -1 & 1 & 1 \end{array}\right] \longrightarrow \left[\begin{array}{cc|cc} 1 & 1 & 0 & 1 \\ 0 & 1 & -1 & -1 \end{array}\right]$$
$$\longrightarrow \left[\begin{array}{cc|cc} 1 & 0 & 1 & 2 \\ 0 & 1 & -1 & -1 \end{array}\right],$$

therefore

$$A_{U \to W} = \begin{bmatrix} 1 & 2 \\ -1 & -1 \end{bmatrix}.$$

(iv) use the change of basis matrix to determine $[\mathbf{v}]_W$.

$$[\mathbf{v}]_W = A_{U\to W} \cdot [\mathbf{v}]_U = \begin{bmatrix} 1 & 2 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ -3 \end{bmatrix}.$$

which is the same as $[\mathbf{u}]_W$ from part (ii).

1.3 Points, lines and planes

Computer graphics uses points, line and planes to describe virtual environments. Euclid defined these in his works *The Elements* around 300 BCE which is considered the most important mathematical works ever written. It introduced mathematical concepts of Euclidean geometry, mathematical proofs, logic and number theory. Euclid defined a point to as "that which has not part", a line to be of "breadthless length" and a plane "that which has length and breadth only".

Definition 1.7: Point

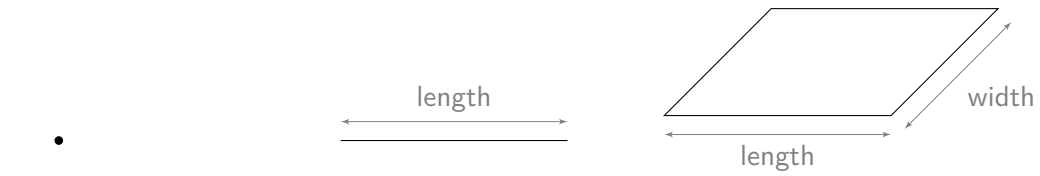
A **point** in Euclidean space is an object of dimension 0 that has no length, width or thickness.

Definition 1.8: Line

A **line** in Euclidean space is an object of dimension 1 that has a length but no width or thickness.

Definition 1.9: Plane

A **plane** in Euclidean space is an object of dimension 2 that has a length and width but no thickness.



(a) point (b) line (c) plane

Figure 1.7: A point, line and plane in Euclidean geometry.

1.3.1 Points

Individual points in a virtual environment can be described by their **position vectors** which is a vector pointing from the origin of the co-ordinate system to the point (fig. 1.8). Sometimes we abuse the language and refer to a point with position vector $\mathbf{p}$ as simply *the point* $\mathbf{p}$. But we should still be mindful of the difference between a point, a single dimensionless position in a space, and the position vector, which is a vector quantity corresponding to the direction and distance from the origin to the point in question.

1.3.2 Lines

We can define a line, L, in using a single point and a direction vector by the vector equation of a line.

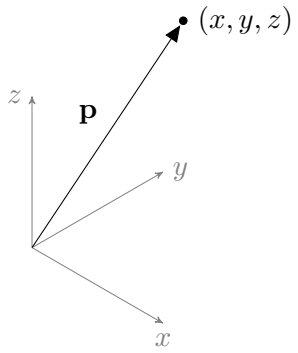


Figure 1.8: A position $\mathbf{p}$ vector points from the origin to the point with co-ordinates (x, y, z).

Definition 1.10: Vector equation of a line

The **vector equation of a line**, L, is an expression of the form

$$\mathbf{r} = \mathbf{p} + t\mathbf{d}, \tag{1.10}$$

where $\mathbf{p}$ is a position vector of a point on L, $\mathbf{d}$ is a vector pointing in a direction parallel to L and t is some scalar.

Often a line L will be defined as the unique line passing through a pair of points with position vectors $\mathbf{p}_1$ and $\mathbf{p}_2$ (fig. 1.9). Then we may use one of the points, $\mathbf{p}_1$ say, as the reference point and the vector $\mathbf{d} = \mathbf{p}_2 - \mathbf{p}_1$ as the direction vector. The vector equation of L is then

$$\mathbf{r} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1). \tag{1.11}$$

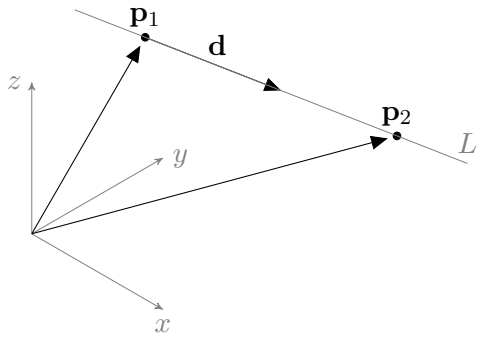


Figure 1.9: The line L can be described using the equation $\mathbf{r} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$.

This has the nice aspect of showing that points on L are points with position vectors that are this particular linear combination of $\mathbf{p}_1$ and $\mathbf{p}_2$. Moreover, values of the scalar satisfying $0 < t < 1$ will correspond to points on L lying between $\mathbf{p}_1$ and $\mathbf{p}_2$, and as t varies continuously from 0 to 1, the point $\mathbf{r}$ varies continuously along the line L from $\mathbf{p}_1$ to $\mathbf{p}_2$.

Example 1.6

(i) The line L passes through two points with position vectors $\mathbf{p}_1 = (5, 4, 1)$ and $\mathbf{p}_2 = (6, -2, 3)$. Calculate the position vector of the point that is a quarter of the way along L between $\mathbf{p}_1$ and $\mathbf{p}_2$.

(ii) The line L passes through the point with position vector $\mathbf{p} = (2, 0, 1)$ and is parallel to $\mathbf{d} = (2, 2, 1)$. Does the point with position vector $\mathbf{q} = (4, 2, 1)$ also lie on L?

Solution:

(i) Using eq. (1.11)

$$\mathbf{r} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) = \begin{bmatrix} 5 \\ 4 \\ 1 \end{bmatrix} + \frac{1}{4}\left(\begin{bmatrix} 6 \\ -2 \\ 3 \end{bmatrix} - \begin{bmatrix} 5 \\ 4 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 5 \\ 4 \\ 1 \end{bmatrix} + \begin{bmatrix} \frac{1}{4} \\ \frac{3}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 2\frac{1}{4} \\ \frac{5}{2} \\ \frac{3}{2} \end{bmatrix}.$$

(ii) Using the vector equation of a line eq. (1.10)

$$\mathbf{r} = \mathbf{p} + t\mathbf{d}$$
$$\begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} + t \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}.$$

So we have the system

$$\begin{aligned} 4 &= 2 + 2t, \\ 2 &= 2t, \\ 1 &= 1 + t. \end{aligned}$$

The second equation gives $t = 1$ which substituted into the third equation gives $1 = 2$ which is a contradiction so $\mathbf{q}$ does not lie on L.

1.3.3 Planes

We can define a plane, P, in three dimensions by specifying three distinct points in $\mathbb{R}^3$. A set of three such points will define a unique plane passing through the points. Let the points have positions vectors $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. If we focus on the first point as our reference point on the plane then the two vectors $\mathbf{p}_2 - \mathbf{p}_1$ and $\mathbf{p}_3 - \mathbf{p}_1$ will be two linearly independent vectors parallel to the plane P (fig. 1.10). As such, any point on P, will have position vector $\mathbf{r}$ of the form

$$\mathbf{r} = \mathbf{p}_1 + a(\mathbf{p}_2 - \mathbf{p}_1) + b(\mathbf{p}_3 - \mathbf{p}_1), \tag{1.12}$$

where $a, b \in \mathbb{R}$ are some scalars.

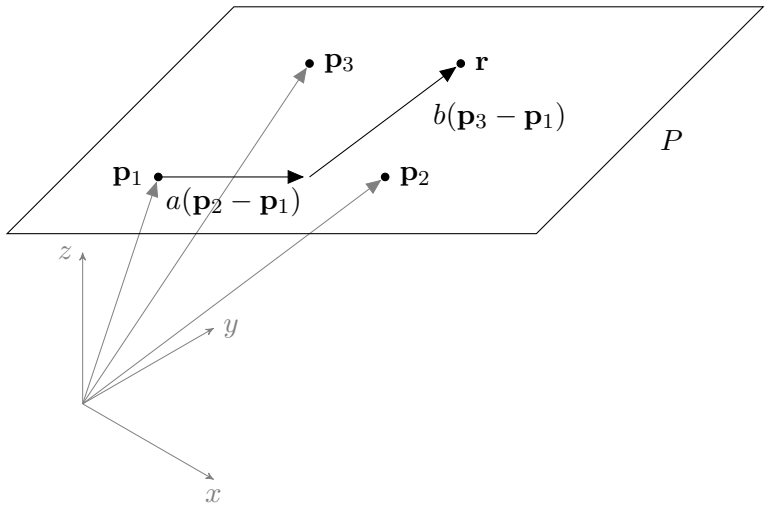


Figure 1.10: A point on a plane $\mathbf{r}$ can be determined by a linear combination of the vectors $\mathbf{p}_2 - \mathbf{p}_1$ and $\mathbf{p}_3 - \mathbf{p}_1$.

The algebra dealing with planes can be greatly simplified by considering the normal vector to a plane.

Definition 1.11: Normal vector

The **normal vector** to a plane is a vector that is perpendicular to the plane. If the points with position vectors $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ lie on a plane P then the normal vector to P is (fig. 1.11)

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1). \tag{1.13}$$

Note that the normal vector to a plane can point in two opposite directions.

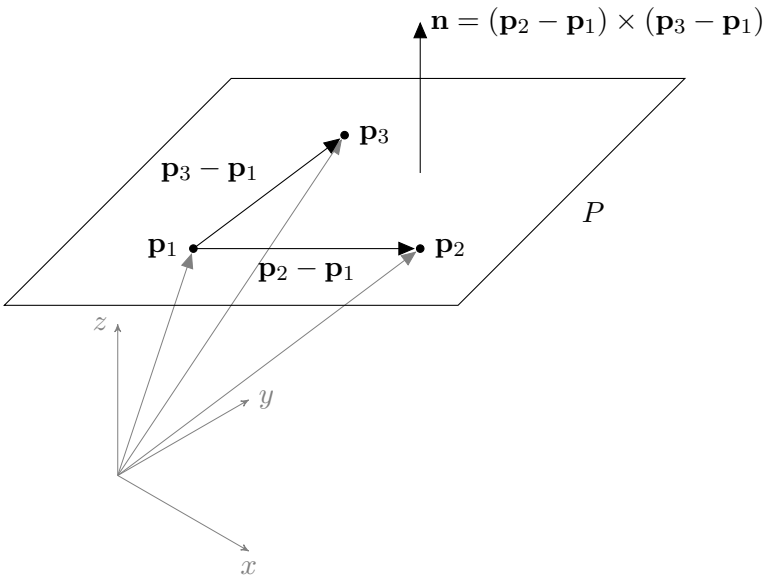


Figure 1.11: The normal vector to a plane is calculated using $\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$.

Suppose now that $\mathbf{r}$ is the position vector of a point on the plane P. Then the vector $\mathbf{r} - \mathbf{p}$ will be a vector with direction parallel to P. As such it will satisfy

$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{p}_1) = 0,$$

which using the distributivity property of the dot product gives the vector equation of a plane.

Definition 1.12: Vector equation of a plane

The position vector of a point, $\mathbf{r}$, that lies on the plane, P, with normal vector, $\mathbf{n}$, and the position vector a point, $\mathbf{p}$, known to lie on P can be calculated using the **vector equation of a plane**

$$\mathbf{n} \cdot \mathbf{r} = \mathbf{n} \cdot \mathbf{p}. \tag{1.14}$$

Note that the right-hand side does not depend on the particular point $\mathbf{r}$. So the quantity $\mathbf{n} \cdot \mathbf{p}$ is a fixed constant associated to the plane P and the choice of $\mathbf{n}$. Let us denote it as $s = \mathbf{n} \cdot \mathbf{p}$ then the vector equation of a plane can be expressed as

$$\mathbf{n} \cdot \mathbf{r} = s.$$

So any point $\mathbf{r}$ that satisfies this equation will lie on the plane P.

Example 1.7

(i) The plane P passes through the three points $\mathbf{p}_1 = (1, 0, 3)$, $\mathbf{p}_2 = (2, 1, 1)$ and $\mathbf{p}_3 = (0, 1, 3)$. Determine the vector equation that describes P.

(ii) Does the point with position vector $\mathbf{p}_4 = (1, 2, 5)$ lie on the plane from part (i)?

Solution:

(i) Calculating the normal vector

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 1 & 1 & -2 \\ -1 & 1 & 0 \end{vmatrix} = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}.$$

Calculate the scalar $s = \mathbf{n} \cdot \mathbf{p}_1$

$$s = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix} = 2 + 0 + 6 = 8,$$

therefore the vector equation of the plane is

$$\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \cdot \mathbf{r} = 8.$$

(ii)

$$\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 5 \end{bmatrix} = 2 + 4 + 10 = 16,$$

since $\mathbf{r} \cdot \mathbf{n} = 16 \neq 8$ so $\mathbf{p}_4$ does not lie on the plane P.

1.4 Distance calculations

By using vectors to describe points, lines and planes is makes calculating distances between a point and a line easier than using Cartesian co-ordinates.

1.4.1 Distance from a point to a line

Consider the diagram in fig. 1.12 which shows a line which passes through the point at position $\mathbf{p}$ and is parallel to the vector $\mathbf{d}$. Another point, $\mathbf{q}$, lies somewhere off the line and we wish to know the distance, d, from $\mathbf{q}$ to the line. Depending where on the line we measure the distance we will get a different values for d, however, this will be at a minimum where the vector joining some point on the line, $\mathbf{r}$, to $\mathbf{q}$ is perpendicular to $\mathbf{r}$.

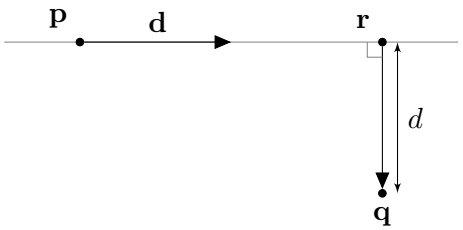


Figure 1.12: The distance between a point and a line

The position of $\mathbf{r}$ is given by the vector equation of a line

$$\mathbf{r} = \mathbf{p} + t\mathbf{v},$$

and we know that the dot product between two perpendicular vectors is zero

$$\mathbf{v} \cdot (\mathbf{q} - \mathbf{r}) = 0,$$

so

$$\mathbf{v} \cdot (\mathbf{q} - \mathbf{p} - t\mathbf{v}) = 0$$

which can be rearranged to solve for t

$$t = \frac{\mathbf{v} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{v} \cdot \mathbf{v}}.$$

The distance from $\mathbf{q}$ to $\mathbf{r}$ is the magnitude of the vector $\mathbf{q} - \mathbf{r}$, i.e.,

$$d = |\mathbf{q} - \mathbf{r}|.$$

Example 1.8

A line, L, passes through the point $\mathbf{p} = (1, 0, 2)$ and is parallel to the vector $\mathbf{v} = (2, -1, 1)$. Another point is positioned at $\mathbf{q} = (6, 4, 5)$, calculate the shortest distance between the point and the line.

Solution:

$$t = \frac{\mathbf{v} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{v} \cdot \mathbf{v}} = \frac{\begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \cdot \left(\begin{bmatrix} 6 \\ 4 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} \right)}{\begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}} = \frac{3}{2}.$$

So the point on the line that is closest to $\mathbf{q}$ is

$$\mathbf{r} = \mathbf{p} + t\mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + \frac{3}{2} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ -\frac{3}{2} \\ 7/2 \end{bmatrix},$$

and the distance between the point and the line is

$$d = |\mathbf{q} - \mathbf{r}| = \left| \begin{bmatrix} 6 \\ 4 \\ 5 \end{bmatrix} - \begin{bmatrix} 4 \\ -\frac{3}{2} \\ 7/2 \end{bmatrix} \right| = \left| \begin{bmatrix} 2 \\ 11/2 \\ 3/2 \end{bmatrix} \right| = 6.0415.$$

1.4.2 Distance from a point to a plane

Consider the diagram in fig. 1.13 which shows a plane, P, defined by the point $\mathbf{p}$ and the normal vector $\mathbf{n}$. Another point which lies off the plane is $\mathbf{q}$ and we wish to find the distance, d, from $\mathbf{q}$ to the plane P. Similar to the point-line distance problem, the shortest distance of $\mathbf{q}$ from P is the perpendicular distance.

The geometric definition of the dot product between the vectors $\mathbf{q} - \mathbf{p}$ and $\mathbf{n}$ is

$$(\mathbf{q} - \mathbf{p}) \cdot \mathbf{n} = |\mathbf{q} - \mathbf{p}||\mathbf{n}| \cos(\theta)$$

where θ is the angle between $\mathbf{n}$ and $\mathbf{q} - \mathbf{p}$. If we consider the right-angled triangle where the hypotenuse is the line $\mathbf{p} \to \mathbf{q}$ and the adjacent side is parallel to $\mathbf{n}$ then

$$\cos(\theta) = \frac{d}{|\mathbf{q} - \mathbf{p}|}.$$

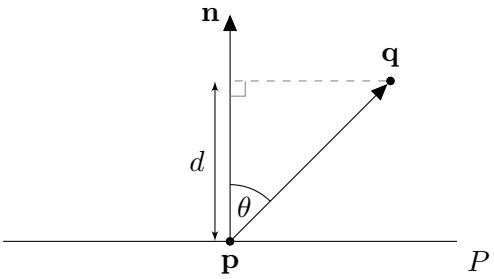


Figure 1.13: The distance between a point and a plane.

Substituting this into the geometric definition of the dot product we have

$$(\mathbf{q}-\mathbf{p})\cdot\mathbf{n} = d|\mathbf{n}|$$

so

$$d = \frac{(\mathbf{q}-\mathbf{p})\cdot\mathbf{n}}{|\mathbf{n}|} = (\mathbf{q}-\mathbf{p})\cdot\hat{\mathbf{n}}. \tag{1.15}$$

Example 1.9

A plane, P, is defined by the point $\mathbf{p} = (1,1,3)$ and the normal vector $\mathbf{n} = (1,-1,1)$. Calculate the distance of the point at position $\mathbf{q} = (4,-3,2)$ to P.

Solution:

$$d = \frac{(\mathbf{q}-\mathbf{p})\cdot\mathbf{n}}{|\mathbf{n}|} = \frac{\left(\begin{bmatrix}4\\-3\\2\end{bmatrix} - \begin{bmatrix}1\\1\\3\end{bmatrix}\right)\cdot\begin{bmatrix}1\\-1\\1\end{bmatrix}}{\left\|\begin{bmatrix}1\\-1\\1\end{bmatrix}\right\|} = \frac{6}{\sqrt{3}} = 2\sqrt{3} = 3.4641.$$

1.5 Lab exercises

Use MATLAB to calculate the solutions to examples 1.1 to 1.9.

The solutions are given on page 79.

Chapter 2

Translation, Rotation and Scaling Transformations

2.1 Linear Transformations

Definition 2.1: Linear transformation

A **linear transformation** between two vector spaces, $\mathbb{R}^m$ and $\mathbb{R}^n$, is a map $T : \mathbb{R}^m \to \mathbb{R}^n$ such that for $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$ and some scalar α the following hold

(i) $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$;

(ii) $T(\alpha\mathbf{u}) = \alpha T(\mathbf{u})$.

Note that if we can show that $T(\mathbf{u} + \alpha\mathbf{v}) = T(\mathbf{u}) + \alpha T(\mathbf{v})$ then conditions (i) and (ii) are satisfied.

2.1.1 Matrix representation of linear transformations

In computer graphics it is convenient to use matrices to represent linear transformations. Let $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ be a basis for $\mathbb{R}^3$ then every vector in $\mathbb{R}^3$ can be uniquely determined by the coefficients x, y and z by

$$\mathbf{u} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}.$$

If $T : \mathbb{R}^3 \to \mathbb{R}^3$ is a linear transformation then by section 2.1 $T(\mathbf{u})$ can be written as

$$T(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) = xT(\mathbf{i}) + yT(\mathbf{j}) + zT(\mathbf{k}),$$

which means that $T(\mathbf{u})$ is determined by the vectors $T(\mathbf{i})$, $T(\mathbf{j})$ and $T(\mathbf{k})$. Suppose these vectors are

$$\begin{aligned} T(\mathbf{i}) &= a\mathbf{i} + d\mathbf{j} + g\mathbf{k}, \\ T(\mathbf{j}) &= b\mathbf{i} + e\mathbf{j} + h\mathbf{k}, \\ T(\mathbf{k}) &= c\mathbf{i} + f\mathbf{j} + i\mathbf{k}, \end{aligned}$$

where $a, b, \ldots, i$ are scalars then $T(\mathbf{u})$ is

$$\begin{aligned} T(\mathbf{u}) &= (a\mathbf{i} + b\mathbf{j} + c\mathbf{k})x + (b\mathbf{i} + e\mathbf{j} + h\mathbf{k})y + (c\mathbf{i} + f\mathbf{j} + i\mathbf{k})z \\ &= (ax + by + cz)\mathbf{i} + (dx + ey + fz)\mathbf{j} + (gx + hy + iz)\mathbf{k} \\ &= \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \\ &= A\mathbf{u}. \end{aligned}$$

So the matrix A represents the linear transformation T with respect to the basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$. This representation provides an exact correspondence between linear transformations and matrices. Each linear transformation is represented by a unique matrix, and each matrix is the representation of a unique linear transformation.

The same linear transformation can be applied to multiple vectors in the space. Let A be the matrix form of the linear transformation T and $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n \in \mathbb{R}^n$ are column vectors then

$$T(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) = A \cdot \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \end{bmatrix}$$

We can form a **co-ordinate matrix**, P, which contains $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$, e.g., if $\mathbf{u}_i = (x_i, y_i, z_i)$ then

$$P = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ x_1 & y_2 & \cdots & z_n \end{bmatrix}$$

and applying the linear transformation have

$$\begin{aligned} A \cdot P &= \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ x_1 & y_2 & \cdots & z_n \end{bmatrix} \\ &= \begin{bmatrix} ax_1 + dy_1 + gz_1 & ax_2 + dy_2 + gz_2 & \cdots & ax_n + dy_n + gz_n \\ dx_1 + ey_1 + fz_1 & dx_2 + ey_2 + fz_2 & \cdots & dx_n + ey_n + fz_n \\ gx_1 + hy_1 + iz_1 & gx_2 + hy_2 + iz_2 & \cdots & gx_n + hy_n + iz_n \end{bmatrix}. \end{aligned}$$

Note how the same transformation has been applied to all columns of P. This is incredibly useful in computer graphics because it allows us to apply the same linear transformations to thousands of points using a single matrix multiplication.

Example 2.1

A transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ is defined by

$$T \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2x \\ 3y \end{bmatrix}$$

(i) Show that T is a linear transformation;

(ii) determine the matrix that represents T;

(iii) use the matrix from part (ii) to calculate $T(\mathbf{u}_1)$, $T(\mathbf{u}_2)$ and $T(\mathbf{u}_3)$ where $\mathbf{u}_1 = (1, 2)$, $\mathbf{u}_2 = (0, 3)$ and $\mathbf{u}_3 = (-1, 4)$.

Solution:

(i) Let $\mathbf{u} = (u_1, u_2)$, $\mathbf{v} = (v_1, v_2)$ where $u_i, v_i \in \mathbb{R}$ and α is any scalar

$$\begin{aligned} T(\mathbf{u} + \alpha\mathbf{v}) &= T\left(\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \alpha \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right) = T \begin{bmatrix} u_1 + \alpha v_1 \\ u_2 + \alpha v_2 \end{bmatrix} \\ &= \begin{bmatrix} 2u_1 + 2\alpha v_1 \\ 3u_2 + 3\alpha v_2 \end{bmatrix} = \begin{bmatrix} 2u_1 \\ 3u_2 \end{bmatrix} + \begin{bmatrix} 2\alpha v_1 \\ 3\alpha v_2 \end{bmatrix} = T(\mathbf{u}) + \alpha T(\mathbf{v}). \end{aligned}$$

So T is a linear transformation.

(ii) We need to find the matrix that satisfies

$$\begin{bmatrix} 2x \\ 3y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

therefore

$$\begin{bmatrix} 2x \\ 3y \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

so the matrix that represents T is

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

(iii) The co-ordinate matrix is

$$P = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 3 & 4 \end{bmatrix},$$

and applying the transformation

$$A \cdot P = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 0 & -2 \\ 6 & 9 & 12 \end{bmatrix}.$$

so $T(\mathbf{u}_1) = (2, 6)$, $T(\mathbf{u}_2) = (0, 9)$ and $T(\mathbf{u}) = (-2, 12)$.

2.1.2 Inverse transformations

Definition 2.2: Inverse linear transformation

If $T : \mathbb{R}^n \to \mathbb{R}^n$ is a linear transformation that is one-to-one map and $\mathbf{v} = T(\mathbf{u})$ for all $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, then the inverse transformation is denoted by $T^{-1} : \mathbb{R}^n \to \mathbb{R}^n$ and defined by

$$T^{-1}(\mathbf{v}) = \mathbf{u}.$$

If A is the transformation matrix for the one-to-one linear transformation $T(\mathbf{u})$ then

$$T^{-1}(\mathbf{u}) = A^{-1}\mathbf{u},$$

where A^{-1} is the matrix inverse of A. This is useful as it allows us to undo the effects of a linear transformation.

Example 2.2

A transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ is defined by

$$T \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2x \\ 3y \end{bmatrix}$$

(i) determine the matrix representation of T^{-1};

(ii) hence determine the inverse transformation $T^{-1}(\mathbf{u})$.

Solution:

(i) The matrix that represents T^{-1} is the inverse of the matrix from example 2.1 part (ii). Using $A^{-1} = \text{adj}(A)/\det(A)$

$$T = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix},$$

$$T^{-1} = \frac{\text{adj}\left(\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}\right)}{\det\left(\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}\right)} = \frac{\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}}{6} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{3} \end{bmatrix}.$$

(ii)

$$T^{-1}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/3 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x/2 \\ y/3 \end{bmatrix}.$$

2.1.3 Composite transformations

Definition 2.3: Composite transformations

The **composite transformations** of two linear transformations $S : \mathbb{R}^n \to \mathbb{R}^p$ and $T : \mathbb{R}^m \to \mathbb{R}^p$ is denoted by $S \circ T$ and is the transformation $S \circ T : \mathbb{R}^m \to \mathbb{R}^n$ defined by

$$(S \circ T)(\mathbf{u}) = S(T(\mathbf{u})),$$

for all $\mathbf{u} \in \mathbb{R}^m$.

The composite transformation $(S \circ T)(\mathbf{u})$ is the same as the result of first applying the transformation $T(\mathbf{u})$ before applying the transformation S to the result, i.e., the order of which the transformations are applied is read from right-to-left.

If S and T are the transformation matrices for the transformations $S(\mathbf{u})$ and $T(\mathbf{u})$, then

$$(S \circ T)(\mathbf{u}) = S \cdot T\mathbf{u}.$$

This is another incredibly useful result as it means we can apply multiply linear transformations using matrix multiplication. For example, let $T_1, T_2, \ldots, T_n$ be n linear transformations applied to $\mathbf{u}$ with equivalent matrix forms then

$$(T_n \circ T_{n-1} \circ \cdots \circ T_2 \circ T_1)(\mathbf{u}) = T_n \cdot T_{n-1} \cdots T_2 \cdot T_1\mathbf{u}$$

In practice, it is often advantageous to pre-calculate the product of the transformation matrices to give a **composite transformation matrix**, this means a composite transformation can be applied with a single matrix multiplication

$$A = T_n \cdot T_{n-1} \cdots T_2 \cdot T_1.$$

The composite transformation is then applied to $\mathbf{u}$ using

$$(T_n \circ T_{n-1} \circ \cdots \circ T_2 \circ T_1)(\mathbf{u}) = A\mathbf{u}.$$

Example 2.3

The two linear transformations $S : \mathbb{R}^2 \to \mathbb{R}^2$ and $T : \mathbb{R}^2 \to \mathbb{R}^2$ are defined by

$$S\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2y \\ x \end{bmatrix}, \qquad T\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}.$$

(i) Without using matrices, calculate $(S \circ T)\begin{bmatrix} 3 \\ 1 \end{bmatrix}$;

(ii) Determine the matrices A and B such that $S(\mathbf{u}) = A\mathbf{u}$ and $T(\mathbf{u}) = B\mathbf{u}$;

(iii) use the matrices from part (ii) to calculate $(S \circ T)\begin{bmatrix} 3 \\ 1 \end{bmatrix}$;

(iv) calculate $(T^{-1} \circ S \circ T)\begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

Solution:

(i)

$$(S \circ T)\begin{bmatrix} 3 \\ 1 \end{bmatrix} = S\left(T\begin{bmatrix} 3 \\ 1 \end{bmatrix}\right) = S\begin{bmatrix} 9 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}$$

(ii) The matrix equation that represents $S(\mathbf{u})$ is

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & b_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2y \\ x \end{bmatrix},$$

therefore $a_{11} = 0$, $a_{12} = 2$, $a_{21} = 1$ and $a_{22} = 0$ so

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

The matrix equation that represents $T(\mathbf{u})$ is

$$\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix},$$

therefore $b_{11} = 3$, $b_{12} = 0$, $b_{21} = 0$ and $b_{22} = 2$ so

$$B = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}.$$

(iii)

$$(S \circ T)\begin{bmatrix} 3 \\ 1 \end{bmatrix} = A \cdot B \cdot \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 4 \\ 3 & 0 \end{bmatrix}\begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}.$$

(iv)

$$(T^{-1} \circ S \circ T)\begin{bmatrix} 1 \\ 2 \end{bmatrix} = B^{-1} \cdot A \cdot B \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}\begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & \frac{4}{3} \\ \frac{3}{2} & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{8}{3} \\ \frac{3}{2} \end{bmatrix}.$$

2.2 Translation

A **translation** is a linear transformation that moves a set of points by the same distance in a given direction. For example, the diagram in fig. 2.1 shows the translation of the point a position $\mathbf{v}$ by the translation vector $\mathbf{p}$.

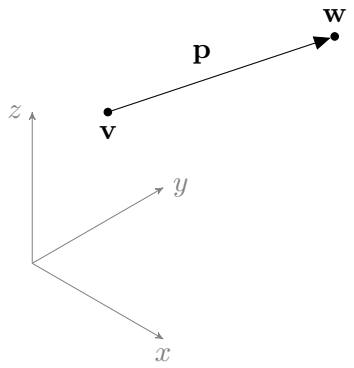


Figure 2.1: The translation of point $\mathbf{v}$ to $\mathbf{w}$ by the translation vector $\mathbf{p}$.

Using vector addition it is easy to see that

$$\mathbf{w} = \mathbf{v} + \mathbf{p}.$$

To represent translation using a single transformation matrix we need to use homogeneous co-ordinates. Let $\mathbf{v} = (x, y, z, 1)$ and $\mathbf{p} = (p_x, p_y, p_z, 1)$ be the homogeneous co-ordinates in $\mathbb{R}^3$ then translation can be written as the matrix equation $\mathbf{w} = A\mathbf{v}$, i.e.,

$$\mathbf{w} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + p_z \\ y + p_y \\ z + p_z \\ 1 \end{bmatrix},$$

Considering the first row of A we the values of a_{11}, a_{12}, a_{13} and a_{14} to satisfy

$$a_{11}x + a_{12}y + a_{13}z + a_{14} = x + p_x,$$

so $a_{11} = 1$, $a_{12} = 0$, $a_{13} = 0$ and $a_{14} = p_x$. Doing similar for the other three rows of A we have

$$A = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.1}$$

which is known as the **translation matrix** and usually denoted by T.

Example 2.4

A triangle is defined by three points with position vectors $\mathbf{v}_1 = (1, 0, 1)$, $\mathbf{v}_2 = (3, 0, 1)$ and $\mathbf{v}_3 = (2, 0, 3)$. The triangle is translated by the translation vector $\mathbf{p} = (3, 0, 1)$. Calculate the positions of the three points after the translation has been applied.

Solution:

The translation matrix is

$$T = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and the homogeneous co-ordinate matrix is

$$P = \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix}.$$

Applying the translation

$$T \cdot P = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 5 \\ 0 & 0 & 0 \\ 2 & 2 & 4 \\ 1 & 1 & 1 \end{bmatrix}.$$

So $\mathbf{w}_1 = (5, 0, 2)$, $\mathbf{w}_2 = (7, 0, 2)$ and $\mathbf{w}_3 = (6, 0, 4)$. The result of the translation can be seen in fig. 2.2. Note that the shape of the triangle has been preserved because all three points have been translated by the same translation vector.

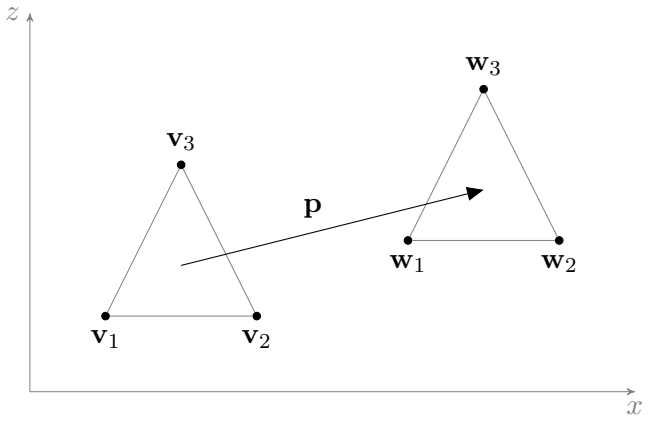


Figure 2.2: Translation of the three points $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ by the translation vector $\mathbf{p} = (3, 0, 1)$.

2.2.1 Inverse translation

Definition 2.4: Inverse linear transformation

If $f : \mathbb{R}^n \to \mathbb{R}^n$ is a linear transformation that is one-to-one that maps any vector $\mathbf{v}$ to $\mathbf{w}$ then the inverse transformation $f^{-1} : \mathbb{R}^n \to \mathbb{R}^n$ maps $\mathbf{w}$ to $\mathbf{v}$.

Translation is a one-to-one map in $\mathbb{R}^3$ so it has an inverse transformation. Translation is simply the addition of the vector $\mathbf{p}$, i.e., $\mathbf{w} = \mathbf{v} + \mathbf{p}$, so the inverse of translation is $\mathbf{v} = \mathbf{w} - \mathbf{p}$, therefore

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.2}$$

2.3 Scaling

Scaling is a linear transformation that moves a point closer or further away from the origin by a certain **scaling vector**. Consider the diagram shown in fig. 2.3 where the point at position $\mathbf{v} = (v_x, v_y, v_z)$ has been scaled by the scaling vector $\mathbf{s} = (s_x, s_y, s_z)$ to change its position to $\mathbf{w}$.

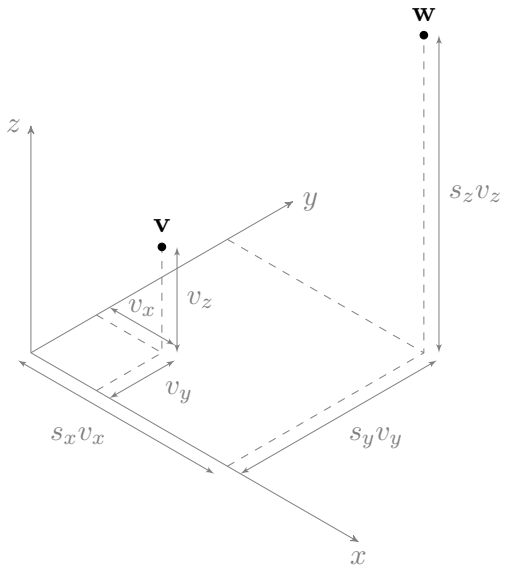


Figure 2.3: The scaling of point $\mathbf{v} = (v_x, v_y, v_z)$ by the scaling vector $\mathbf{s} = (s_x, s_y, s_z)$.

The matrix representation of the scaling transformation is determined using the same method as seen in section 2.2 which results in the following **scaling matrix**. Note that we could have represented scaling in $\mathbb{R}^3$ using a 3×3 matrix and Cartesian co-ordinates but since we require homogeneous co-ordinates for the matrix representation of translation and we want to be able to combine transformations then we use homogeneous co-ordinate for scaling (and any other transformation).

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.3}$$

A scaling vector $\mathbf{s} = (1, 1, 1)$ will result in $S = I$ and the position of any point that is scaled will be unchanged.

Example 2.5

A triangle is defined by three points with position vectors $\mathbf{v}_1 = (1, 0, 1)$, $\mathbf{v}_2 = (3, 0, 1)$ and $\mathbf{v}_3 = (2, 0, 3)$. The triangle is scaled by a the scaling vector $\mathbf{s} = (3, 1, 2)$. Calculate the positions of the three points after the scaling has been applied.

Solution:
The scaling matrix is

$$S = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and the homogeneous co-ordinate matrix is

$$P = \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix}.$$

Applying the scaling

$$S \cdot P = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 9 & 6 \\ 0 & 0 & 0 \\ 2 & 2 & 6 \\ 1 & 1 & 1 \end{bmatrix}.$$

So $\mathbf{w}_1 = (3, 0, 2)$, $\mathbf{w}_2 = (9, 0, 2)$ and $\mathbf{w}_3 = (6, 0, 6)$. The result of the scaling can be seen in fig. 2.4. Note that the shape of the triangle has changed since the scaling is not the same along the x and z axes.

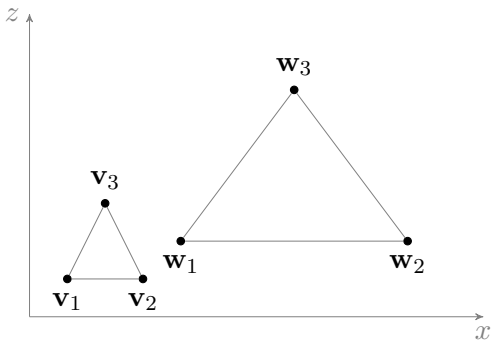


Figure 2.4: Scaling of the three points $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ by the scaling vector $\mathbf{s} = (3, 1, 2)$.

2.3.1 Inverse scaling

Scaling the position vector $\mathbf{v} = (v_x, v_y, v_z, 1)$ by the scaling vector $\mathbf{s} = (s_x, s_y, s_z, 1)$ results in $\mathbf{w} = (s_x v_x, s_y v_y, s_z v_z, 1)$, so the inverse of scaling is $\mathbf{v} = (w_x/s_x, w_y/s_y, w_z/s_z, 1)$, therefore

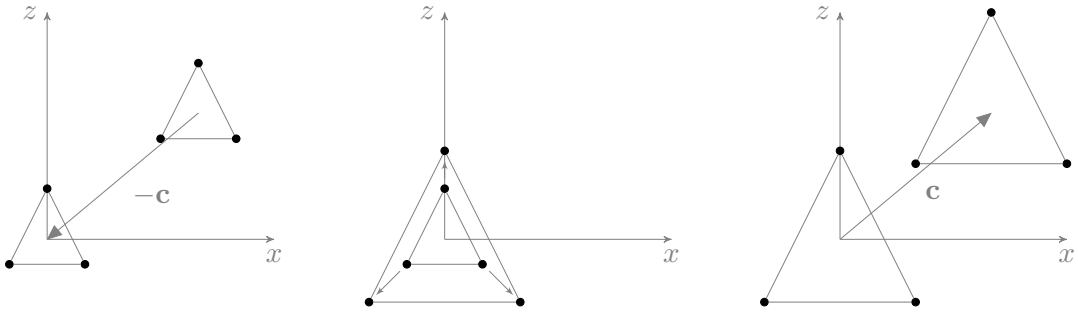
$$S^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.4}$$

2.3.2 Scaling about the centre of a shape

We have seen in example 2.5 that scaling the triangle resulted in the centre of the triangle shifting position. This was because scaling was applied to a shape whose centre was not at the original. To preserve the position of the centre of a shape when scaling we first need to translate the points that define the shape to the origin, which means that the translation vector if $-\mathbf{c}$ where $\mathbf{c}$ is the centre of the shape. Then we can perform the scaling before using the inverse translation so that the centre of the shape is back at $\mathbf{c}$ (fig. 2.5).

So we have the following composite transformation matrix

$$A = T^{-1} \cdot S \cdot T$$



(a) translate by $-\mathbf{c}$ (b) scale points (c) translate by $\mathbf{c}$

Figure 2.5: Scaling a shape about its centre.

Example 2.6

A triangle is defined by three points with position vectors $\mathbf{v}_1 = (2, 0, 2)$, $\mathbf{v}_2 = (4, 0, 2)$ and $\mathbf{v}_3 = (3, 0, 4)$. The triangle is scaled by the scaling vector $\mathbf{s} = (2, 1, 2)$ about its centre. Calculate the positions of the three points after the scaling has been applied.

Solution:
The centre of the triangle is at

$$\mathbf{c} = \frac{1}{3}(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3) = \frac{1}{3}\left(\begin{bmatrix}2\\0\\2\end{bmatrix} + \begin{bmatrix}4\\0\\2\end{bmatrix} + \begin{bmatrix}3\\0\\4\end{bmatrix}\right) = \begin{bmatrix}3\\0\\\frac{8}{3}\end{bmatrix}.$$

so the composite transformation matrix is

$$A = T^{-1} \cdot S \cdot T = \begin{bmatrix}1&0&0&3\\0&1&0&0\\0&0&0&\frac{8}{3}\\0&0&0&1\end{bmatrix}\begin{bmatrix}2&0&0&0\\0&1&0&0\\0&0&2&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&3\\0&1&0&0\\0&0&0&\frac{8}{3}\\0&0&0&1\end{bmatrix} = \begin{bmatrix}2&0&0&-3\\0&1&0&0\\0&0&2&-\frac{8}{3}\\0&0&0&1\end{bmatrix}.$$

and the homogeneous co-ordinate matrix is

$$P = \begin{bmatrix}2&4&3\\0&0&0\\2&2&4\\1&1&1\end{bmatrix}.$$

Applying the composite transformation

$$A \cdot P = \begin{bmatrix}2&0&0&-3\\0&1&0&0\\0&0&2&-\frac{8}{3}\\0&0&0&1\end{bmatrix}\begin{bmatrix}2&4&3\\0&0&0\\2&2&4\\1&1&1\end{bmatrix} = \begin{bmatrix}1&5&3\\0&0&0\\\frac{4}{3}&\frac{4}{3}&\frac{16}{3}\\1&1&1\end{bmatrix}.$$

So $\mathbf{w}_1 = (1, 0, \frac{4}{3})$, $\mathbf{w}_2 = (5, 0, \frac{4}{3})$ and $\mathbf{w}_3 = (3, 0, \frac{16}{3})$. The result of the scaling can be seen in fig. 2.6. Note that the shape of the triangle has remained the same.

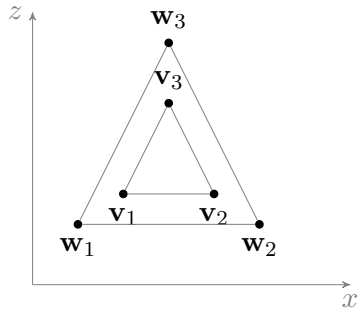


Figure 2.6: Scaling of the three points $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ by the scaling vector $\mathbf{s} = (2, 1, 2)$ about centre of the three points.

2.4 Rotation

Rotation is a linear transformation that rotates a set of points by some angle about one of the axes that define the frame of the space. In $\mathbb{R}^3$ we can rotate around the x, y and z axes and we assume that the direction of rotation is in the anti-clockwise when looking along the axis towards the origin (fig. 2.7).

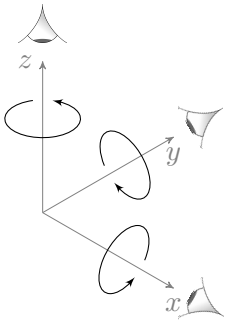


Figure 2.7: Rotation is assumed to be defined as anti-clockwise around the axis when viewed looking down towards the origin.

To determine the rotation transformation we shall consider the rotation about the x axis. Consider the diagram in fig. 2.8 which is fig. 2.7 view looking down the x axis where the point with position vector $\mathbf{v}$ is rotated by θ about the x axis to the point with position vector $\mathbf{w}$.

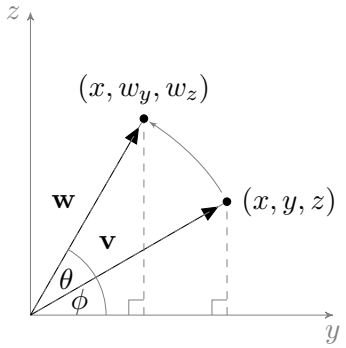


Figure 2.8: Rotation about the x axis.

Let $\mathbf{v} = (x, y, z)$ and $\mathbf{w} = (x, w_y, w_z)$ (note the x co-ordinate is unchanged since we are rotating around the x axis), then using trigonometry the position vectors $\mathbf{v}$ and $\mathbf{w}$ are

$$\mathbf{v} = (x, r\cos(\phi), r\sin(\phi)),$$
$$\mathbf{w} = (x, r\cos(\phi + \theta), r\sin(\phi + \theta)),$$

where $r = |\mathbf{v}| = |\mathbf{w}|$. Using compound angle formulae

$$\cos(\phi + \theta) = \cos(\phi)\cos(\theta) + \sin(\phi)\sin(\theta),$$
$$\sin(\phi + \theta) = \sin(\phi)\cos(\theta) - \cos(\phi)\sin(\theta),$$

then we can write $\mathbf{w}$ using

$$w_y = r\cos(\phi)\cos(\theta) + r\sin(\phi)\sin(\theta),$$
$$w_z = r\sin(\phi)\cos(\theta) - r\cos(\phi)\sin(\theta),$$

and since $y = r\cos(\phi)$ and $z = r\sin(\phi)$ these simplify to

$$w_y = y\cos(\theta) + z\sin(\theta),$$
$$w_z = -y\sin(\theta) + z\cos(\theta),$$

Using homogeneous co-ordinates to be consistent with translation we can write this as the matrix equation

$$\begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

So the rotation by angle θ about the x axis in $\mathbb{R}^3$ can be represented by the following **rotation matrix**

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.5}$$

Doing similar for the rotation about the y and z axes gives

$$R_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.6}$$

$$R_z = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.7}$$

Example 2.7

A triangle is defined by three points with position vectors $\mathbf{v}_1 = (4, 0, 1)$, $\mathbf{v}_2 = (6, 0, 1)$ and $\mathbf{v}_3 = (5, 0, 3)$. The triangle is rotated by angle $\theta = \pi/4$ anti-clockwise about the y axis. Calculate the positions of the three points after the scaling has been applied.

Solution:
The rotation matrix is

$$R_y = \begin{bmatrix} \cos(\pi/4) & 0 & -\sin(\pi/4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\pi/4) & 0 & \cos(\pi/4) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 & 0 & -\sqrt{2}/2 & 0 \\ 0 & 1 & 0 & 0 \\ \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the homogeneous co-ordinate matrix is

$$P = \begin{bmatrix} 4 & 6 & 5 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix}.$$

Applying the rotation

$$R_y \cdot P = \begin{bmatrix} \sqrt{2}/2 & 0 & -\sqrt{2}/2 & 0 \\ 0 & 1 & 0 & 0 \\ \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 6 & 5 \\ 0 & 0 & 0 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3\sqrt{2}/2 & 5\sqrt{2}/2 & \sqrt{2} \\ 0 & 0 & 0 \\ 5\sqrt{2}/2 & 7\sqrt{2}/2 & 4\sqrt{2} \\ 1 & 1 & 1 \end{bmatrix}.$$

So $\mathbf{w}_1 = (3\sqrt{2}/2, 0, 5\sqrt{2}/2)$, $\mathbf{w}_2 = (5\sqrt{2}/2, 0, 7\sqrt{2}/2)$, and $\mathbf{w}_3 = (\sqrt{2}, 0, 4\sqrt{2})$. The result of the rotation can be seen in fig. 2.9. Note that the triangle has been rotated about the origin as such the position of its centre has changed.

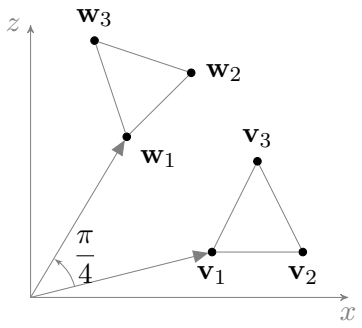
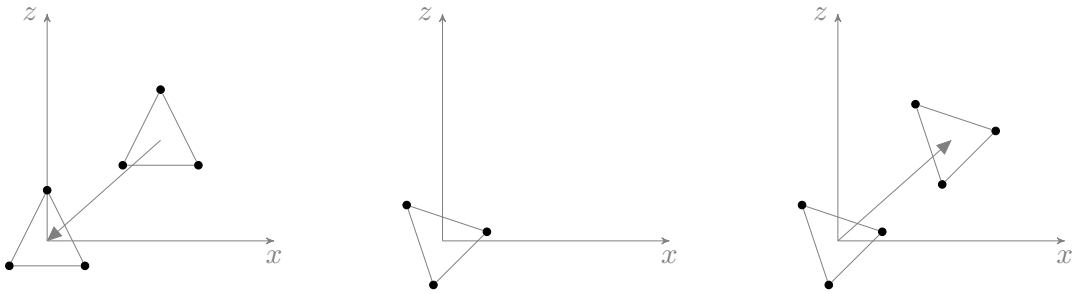


Figure 2.9: Rotation of the three points $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ by angle $\theta = \pi/4$ anti-clockwise about the y axis.

2.4.1 Rotating about the centre of a shape

We have seen in example 2.7 the centre of triangle shifted position because we rotate around the origin. So, similar to scaling about the centre of a shape, we need to translate the points by $-\mathbf{c}$ where $\mathbf{c}$ is the centre of the shape, perform the rotation before translating back by $\mathbf{c}$.



(a) translate by $-\mathbf{c}$ (b) rotate points (c) translate by $\mathbf{c}$

Figure 2.10: Rotating a shape about its centre.

So we have the following composite transformation matrix

$$A = T^{-1} \cdot R_y \cdot T$$

2.4.2 Inverse rotation

The inverse of rotating a position vector by angle θ is to rotate it by angle $-\theta$. Since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$ then the inverse rotation matrices are

$$R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.8}$$

$$R_y^{-1} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.9}$$

$$R_z^{-1} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.10}$$

2.4.3 Rotations around other axes

Suppose we wish to rotate $\mathbb{R}^3$ by an angle θ about a general line, L, where L does not pass through the original and is not parallel to any of the three axes. To achieve this we first need to apply translation and rotation so that L along one of the three axes, rotate about this axis by θ before reversing the rotation and translation operations. Each of the individual transformations can be represented by a matrix so the composite transformation that achieves the rotation about L is a product of the individual matrices.

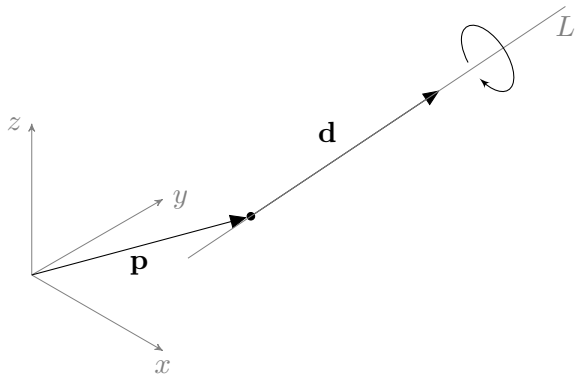


Figure 2.11: Rotation about the line $L : \mathbf{r} = \mathbf{p} + t\mathbf{d}$.

For example, consider the diagram in fig. 2.11 where we wish to rotate about the line $L : \mathbf{r} = \mathbf{p} + t\mathbf{d}$. The steps required to do this are as follows:

1. First we need to translate by $-\mathbf{p}$ so that L passes through the origin, so the first transformation matrix is

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_x \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   where $\mathbf{p} = (p_x, p_y, p_z)$. This results in the diagram shown in fig. 2.12(a).

2. Now consider the direction vector $\mathbf{d} = (d_x, d_y, d_z)$ of L as shown in fig. 2.12(a). We will use v to denote the magnitude of the projection of $\mathbf{d}$ onto the xy plane. We wish to rotate $\mathbb{R}^3$ about the z axis so that $\mathbf{d}$ lies on the yz plane so that $d_x = 0$. If ϕ is the angle between side v and the y axis

then the matrix for achieving this rotation is

$$R_z = \begin{bmatrix} \cos(\phi) & \sin(\phi) & 0 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d_y/v & d_x/v & 0 & 0 \\ -d_x/v & d_y/v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Performing this rotation gives the vector $\mathbf{d}$ in fig. 2.12(b).

3. Now consider $\mathbf{d}$ in fig. 2.12(b) and we will use d to denote the magnitude of $\mathbf{d}$. We rotate $\mathbb{R}^3$ **clockwise** around the x axis so that $\mathbf{d}$ is brought into alignment with the z axis so that $d_x = 0$ and $d_y = 0$. Suppose ψ is the angle between the sides $\mathbf{d}$ and v in fig. 2.12(b) then the matrix for achieving this rotation is (note that the signs of the $\sin$ functions are swapped because we are rotating in the clockwise direction)

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) & 0 \\ 0 & \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & v/d & -d_z/d & 0 \\ 0 & d_z/d & v/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

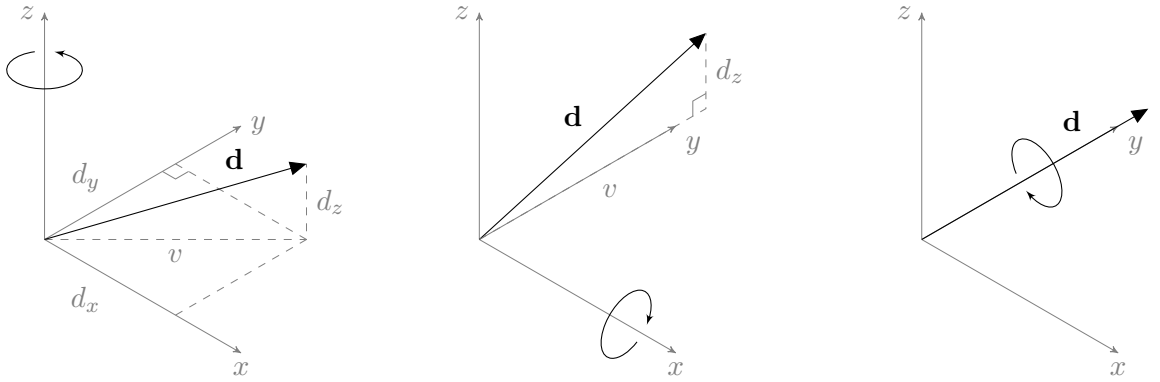
Performing this rotation gives the vector $\mathbf{d}$ in fig. 2.12(c)

4. The direction vector $\mathbf{d}$, and hence the line L, now lies on the y axis. Now we can perform the rotation of $\mathbb{R}^3$ by angle θ about the y axis. The rotation matrix for performing this rotation is

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

5. Rotating $\mathbf{d}$ so that it points in its original direction is achieved using R_x^{-1} and R_z^{-1}.

6. Translating L so that it passes through $\mathbf{p}$ is achieved using T^{-1}.



(a) rotate around z axis. (b) rotate around x axis (c) rotate around y axis

Figure 2.12: Rotation about the line $L : \mathbf{r} = \mathbf{p} + t\mathbf{d}$

Multiplying these matrices together in the correct order will gives the composite transformation matrix A representing rotation by an angle θ anti-clockwise around the line $L : \mathbf{r} = \mathbf{p} + t\mathbf{d}$

$$A = T^{-1} \cdot R_z^{-1} \cdot R_x^{-1} \cdot R_y \cdot R_x \cdot R_z \cdot T.$$

Note that we could have chosen other axes to rotate around to achieve the same goal. The choice of axes is arbitrary and may depend upon which octant $\mathbf{d}$ is pointing towards.

Example 2.8

A flight simulation program is simulating the flight of an aeroplane positioned with its centre of mass at $\mathbf{p} = (10, 5, 50)$ travelling in the direction given by the vector $\mathbf{d} = (2, -1, -3)$. The user performs a roll of the plane by rotating about $\mathbf{d}$ by angle $\theta = \pi/6$ in the anti-clockwise direction. Calculate the composite transformation matrix that performs this action.

Solution:

First we translate by $-\mathbf{p}$ so that the centre of the plane is at the origin (fig. 2.13(a))

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & -50 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Now we need to consider which rotations we apply so that $\mathbf{d}$ points along one of the axes. This decision is arbitrary but in this case since d_x is positive then rotating so that $\mathbf{d}$ points along the x axis would require the fewest rotations. Looking at fig. 2.13(b) this means can first rotate $\mathbb{R}^3$ anti-clockwise around the z axis so that $\mathbf{d}$ is in the xz plane. If v is the magnitude of the projection of $\mathbf{d}$ onto the xy plane and ϕ is the angle between side v and the x axis then the matrix for achieving this rotation is (note that $d_y < 0$)

$$R_z = \begin{bmatrix} \cos(\phi) & \sin(\phi) & 0 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d_x/v & d_y/v & 0 & 0 \\ -d_y/v & d_x/v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} & 0 & 0 \\ 1/\sqrt{5} & 2/\sqrt{5} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can check whether this rotation is correct by multiplying it by the homogeneous form of $\mathbf{d}$

$$Rz \cdot \mathbf{d} = \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} & 0 & 0 \\ 1/\sqrt{5} & 2/\sqrt{5} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ -3 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{5} \\ 0 \\ -3 \\ 1 \end{bmatrix}.$$

Since $d_y = 0$ then $\mathbf{d}$ is now in the xz plane. Now we need to rotate $\mathbb{R}^3$ clockwise around the y axis so that $\mathbf{d}$ points along the x axis (remember that the direction of rotation is based on the view looking down the axis towards the origin). If ψ is the angle between the x axis and $\mathbf{d}$ in fig. 2.13(c) then the matrix for achieving this rotation is (note that $d_z < 0$)

$$R_y = \begin{bmatrix} \cos(\psi) & 0 & \sin(\psi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\psi) & 0 & \cos(\psi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} v/|\mathbf{d}| & 0 & d_z/|\mathbf{d}| & 0 \\ 0 & 1 & 0 & 0 \\ -d_z/|\mathbf{d}| & 0 & v/|\mathbf{d}| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{5}/\sqrt{14} & 0 & 3/\sqrt{14} & 0 \\ 0 & 1 & 0 & 0 \\ -3/\sqrt{14} & 0 & \sqrt{5}/\sqrt{14} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Again, we can check whether this rotation is correct by multiplying it by $Rz \cdot \mathbf{d}$

$$R_y \cdot R_z \cdot \mathbf{d} = \begin{bmatrix} \sqrt{5}/\sqrt{14} & 0 & 3/\sqrt{14} & 0 \\ 0 & 1 & 0 & 0 \\ -3/\sqrt{14} & 0 & \sqrt{5}/\sqrt{14} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{5} \\ 0 \\ -3 \\ 1 \end{bmatrix} = \begin{bmatrix} 13/\sqrt{14} \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Since $d_y = 0$, $d_z = 0$ and $d_x > 0$ then $\mathbf{d}$ is now pointing along the x axis (fig. 2.8). Now we perform the rotation of $\mathbb{R}^3$ anti-clockwise around the x axis by angle $\theta = \pi/6$ and the matrix for achieving this rotation is

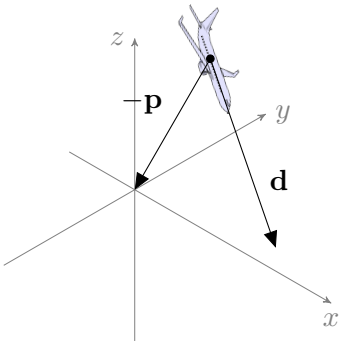
$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \sqrt{3}/2 & 1/2 & 0 \\ 0 & -1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The rotations about the y and z axis and the translation are reversed so the complete composite matrix is

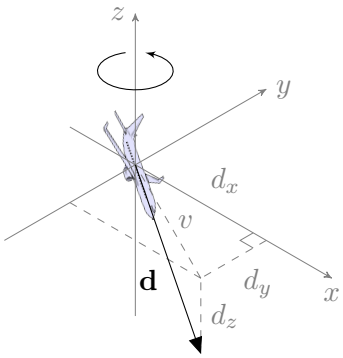
$$A = T^{-1} \cdot R_z^{-1} \cdot R_y^{-1} \cdot R_x \cdot R_y \cdot R_z \cdot T,$$

where

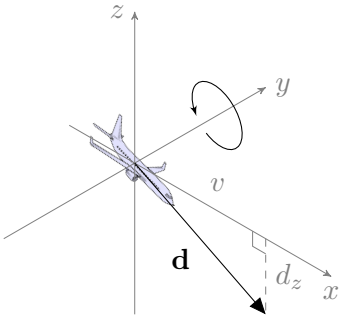
$$R_z^{-1} = \begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} & 0 & 0 \\ -1/\sqrt{5} & 2/\sqrt{5} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad R_y^{-1} = \begin{bmatrix} \sqrt{5}/\sqrt{14} & 0 & -3/\sqrt{14} & 0 \\ 0 & 1 & 0 & 0 \\ 3/\sqrt{14} & 0 & \sqrt{5}/\sqrt{14} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 50 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$



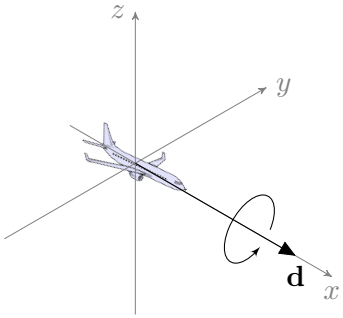
(a) translate by $-\mathbf{p}$



(b) rotating around z axis



(c) rotating around y axis



(d) rotating around x axis

Figure 2.13: The translations and rotations used in example 2.8.

2.5 Lab exercises

Use MATLAB to calculate the solutions to examples 2.1 to 2.9.

The solutions are given on page 81.

Chapter 3

Virtual Environments

3.1 The viewing pipeline

The steps required to construct a virtual environment and display it so it can be viewed by a computer user are summarised in a flow diagram known as **the viewing pipeline** which is shown in fig. 3.1.

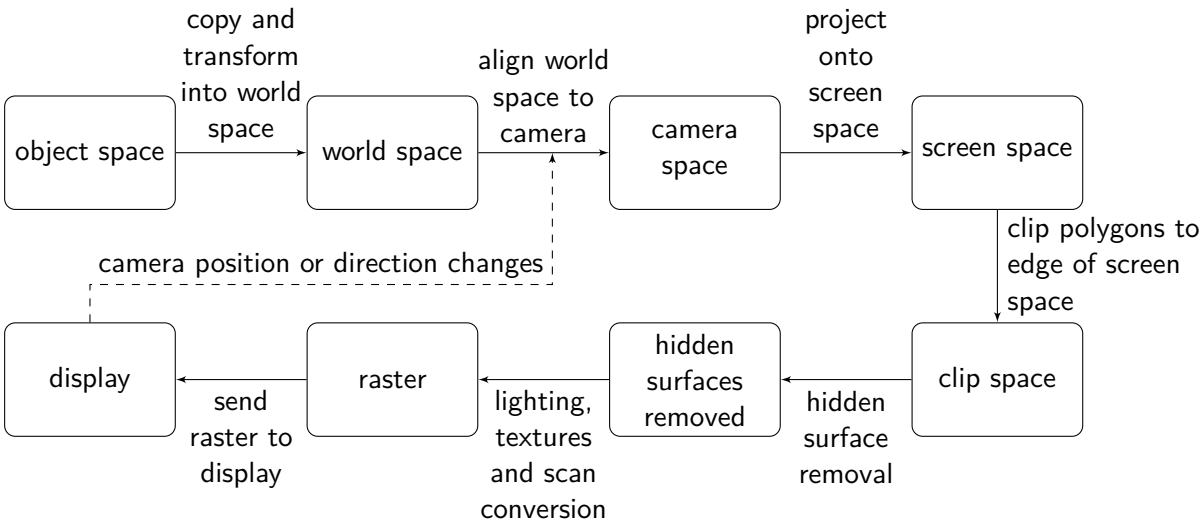


Figure 3.1: The viewing pipeline summarises the steps used to construct and display a virtual environment.

The steps are as follows

- **Object space** – Primitive objects that are used to build your virtual environment are defined within their own space such that the origin passes through the centre of the object (or sometimes the centre of the bases of the object). For example, a virtual environment that describes a street scene may have objects for buildings, cars, bus stops, street lights etc.
- **World space** – The virtual environment is constructed by copying the objects into the world space and applying scaling, rotation and translation transformations.
- **Camera space** – To view the virtual environment we place a virtual camera in the world space and use translation and rotation transformations so that the position of the camera is at the origin looking along the z axis in the negative direction.
- **Screen space** – The camera space is projected onto a two-dimensional projection plane using perspective projection so that the further objects are from the camera the smaller they appear giving the illusion of depth.

- **Clipped screen space** – Any objects in the screen space that are outside of the region that visible to the camera are removed and objects that lie partially outside are 'clipped' to the edges.
- **Hidden surfaces removed** – Any face of an object that is facing away from the camera are removed and a rendering order of the polygons is determined so that closer surfaces obstruct those further away.
- **Raster** – The colours of the individual pixels on a display screen are determined from the information obtained about the virtual environment up to this point and any lighting conditions used. This information is stored in a **raster array** where each element corresponds to a pixel on the display.
- **Display** – The raster array is sent to the display hardware.

If the position and/or direction of the virtual camera used to calculate the camera space changes, e.g., through the actions of a player or a computer game, the steps from the camera space to the display are repeated. This will typically be 30 or 60 times per second which highlights the computational demands placed on computer hardware and the need for all calculations to be as efficient as possible.

These notes will focus on the steps from the object space to the hidden surface removal stage. The remaining steps are outside the focus of this unit.

3.2 Defining objects

Each object in a virtual world is defined in its own space where the centre of mass of the object is at the origin so that scaling, translation and rotation operations can be easily applied (fig. 3.2).

(a) house (b) tower

Figure 3.2: Each object is defined in its own object space.

A three-dimensional object is constructed using polygons for the **faces** of the object where each face is defined by its **vertices**. Since multiple faces of on object can share the same vertex we use one array to list the vertex co-ordinates and another array to list the vertices that defines each face. Consider the cube object in fig. 3.3, if the sides are parallel to the co-ordinate axes with side lengths 2 then the homogeneous co-ordinates of the 8 vertices $\mathbf{v}_1$ to $\mathbf{v}_8$ are

$$\mathbf{v}_1 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix},$$

$$\mathbf{v}_5 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_6 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_7 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{v}_8 = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

We can form a single matrix containing these vertices known as the **vertex matrix**

$$V = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 & \mathbf{v}_5 & \mathbf{v}_6 & \mathbf{v}_7 & \mathbf{v}_8 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

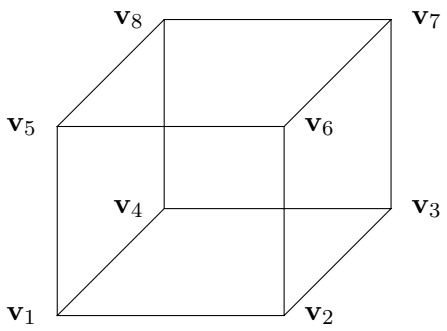
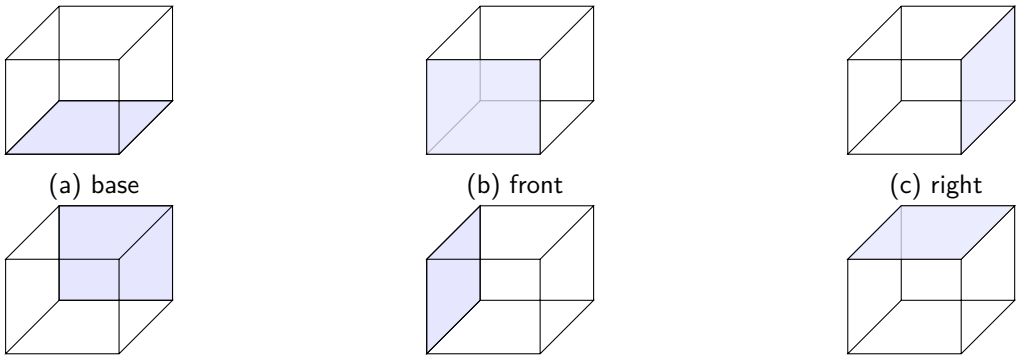


Figure 3.3: Cube object

The faces of the object are defined by a **face matrix** that links to the vertex matrix. The face matrix is an $m \times n$ matrix where m is the number of faces of the object and n is the number of sides for each face, so for the cube object the face matrix will be 6×4. The rows of the face matrix contain the column number of the vertex matrix corresponding to vertices of that face. For example, consider the faces of the cube object shown in fig. 3.4. The face that represents the base has vertices $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ and $\mathbf{v}_4$ so the row of the face matrix will contain the numbers 1, 2, 3 and 4.



(a) base (b) front (c) right

(d) back (e) left (f) top

Figure 3.4: The faces of the cube object.

We also need to consider the order that the vertices are listed for each face. Objects are defined so that the normal vectors for each face is pointing away from the centre of the object. We have seen in section 1.3.3 that the normal vector to a plane that passes through the 3 points with position vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ is calculated using

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1).$$

If $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are ordered anti-clockwise when viewed from one side of the plane then the above equation will result in a normal vector pointing towards the viewer. With this in mind the face matrix for the cube

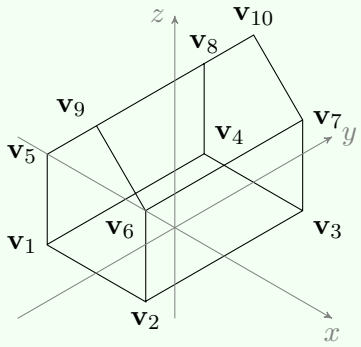
object is

$$F = \begin{matrix} \text{base} \\ \text{front} \\ \text{right} \\ \text{back} \\ \text{left} \\ \text{right} \end{matrix} \begin{bmatrix} 1 & 4 & 3 & 2 \\ 1 & 2 & 6 & 5 \\ 2 & 3 & 7 & 6 \\ 3 & 4 & 8 & 7 \\ 4 & 1 & 5 & 8 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Note that it does not matter which order the vertices are listing as columns in the world space vertex matrix or the faces listed in rows of the face matrix, as long as the face matrix correctly links to the vertex matrix.

Example 3.1

A simple three-dimensional object that resembles a house is shown below (modelled on the house piece from the *Monopoly* board game)



The house has length of 2, width of 1, wall height 1 and roof height 2 and defined so that the center of the base is at the origin. Define the vertex and face matrices for this house object.

Solution:
The homogeneous co-ordinates of the vertices $\mathbf{v}_1$ to $\mathbf{v}_{10}$ are:

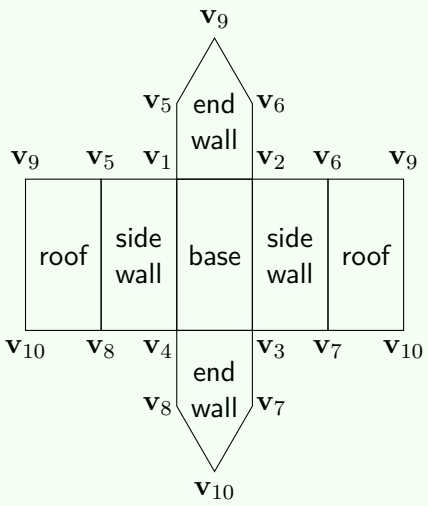
$$\mathbf{v}_1 = \begin{bmatrix} -\frac{1}{2} \\ -1 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{v}_2 = \begin{bmatrix} \frac{1}{2} \\ -1 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{v}_3 = \begin{bmatrix} \frac{1}{2} \\ 1 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{v}_4 = \begin{bmatrix} -\frac{1}{2} \\ 1 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{v}_5 = \begin{bmatrix} -\frac{1}{2} \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{v}_6 = \begin{bmatrix} \frac{1}{2} \\ -1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{v}_7 = \begin{bmatrix} \frac{1}{2} \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{v}_8 = \begin{bmatrix} -\frac{1}{2} \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{v}_9 = \begin{bmatrix} 0 \\ -1 \\ 2 \\ 1 \end{bmatrix} \quad \mathbf{v}_{10} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 1 \end{bmatrix},$$

so the vertex matrix is

$$V_{\text{house}} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

The house object has 7 faces: a base, 2 end walls, 2 side walls and 2 roofs. The base, side walls and roofs are 4-sided faces whereas the end walls are 5-sided faces. The face matrix for the house object that ensures the normal vectors are point away from the centre of the object is (vertices are listed in an anti-clockwise direction when looking towards the centre). Note that the last vertex of the 4-sided faces are repeated to ensure that each row of the face matrix has 5 elements.



$$F_{\text{house}} = \begin{matrix} \text{base} \\ \text{end wall} \\ \text{side wall} \\ \text{end wall} \\ \text{side wall} \\ \text{roof} \\ \text{roof} \end{matrix} \begin{bmatrix} 1 & 4 & 3 & 2 & 2 \\ 1 & 2 & 6 & 9 & 5 \\ 2 & 3 & 7 & 6 & 6 \\ 3 & 4 & 8 & 10 & 7 \\ 1 & 5 & 8 & 4 & 4 \\ 6 & 7 & 10 & 9 & 9 \\ 5 & 9 & 10 & 8 & 8 \end{bmatrix}$$

3.2.1 MATLAB code

The MATLAB code in listing 3.1 defines the vertex and face matrices for the house object from example 3.1 and plots the object space (fig. 3.5). The `patch` command can use the vertex and face matrices `Vhouse` and `Fhouse`. Note that we only use the first three rows of `Vhouse` and the array has been transposed because MATLAB assumes the co-ordinates are listed in rows. The `FaceAlpha` is set to zero so that the faces of the object are transparent.

Listing 3.1: MATLAB code to define the house object from example 3.1 and plot the object space.

```
% Define house object
Vhouse = [-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2, 0, 0 ;
         -1, -1, 1, 1, -1, -1, 1, 1, -1, 1 ;
         0, 0, 0, 0, 1, 1, 1, 1, 2, 2 ;
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
Fhouse = [1, 4, 3, 2, 2 ;
         1, 2, 6, 9, 5 ;
         2, 3, 7, 6, 6 ;
         3, 4, 8, 10, 7 ;
         1, 5, 8, 4, 4 ;
         6, 7, 10, 9, 9 ;
         5, 9, 10, 8, 8 ];

% Plot object space
patch('Vertices', Vhouse(1:3,:)', 'Faces', Fhouse, 'FaceAlpha', 0)
axis([-1, 1, -2, 2, 0, 2])
view(60, 30)
xlabel('x')
ylabel('y')
zlabel('z')
```

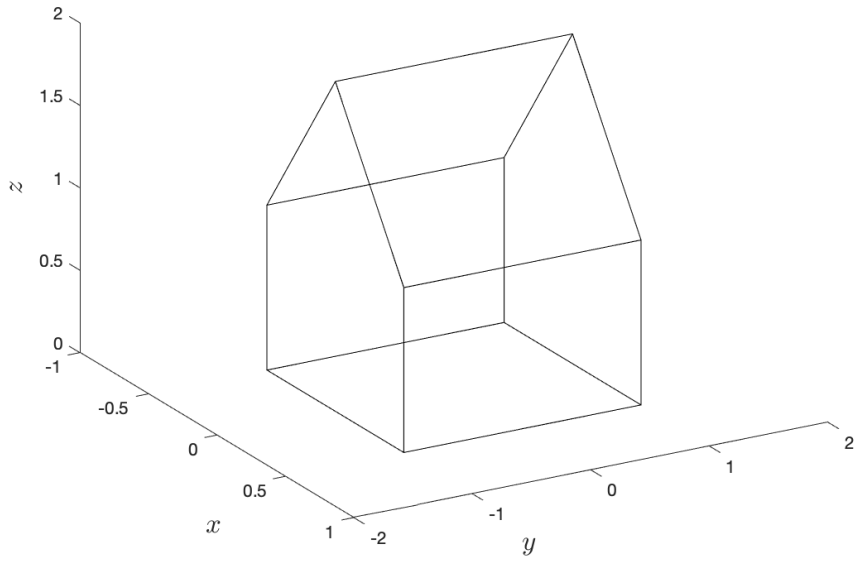


Figure 3.5: MATLAB plot of the house object from example 3.1.

3.3 Building a virtual environment

Once the vertex and face matrices for the objects have been define they can be used to build the virtual environment. The objects are copied to the world space and transformed using scaling, rotation and translation operations to construct the virtual world (fig. 3.6)

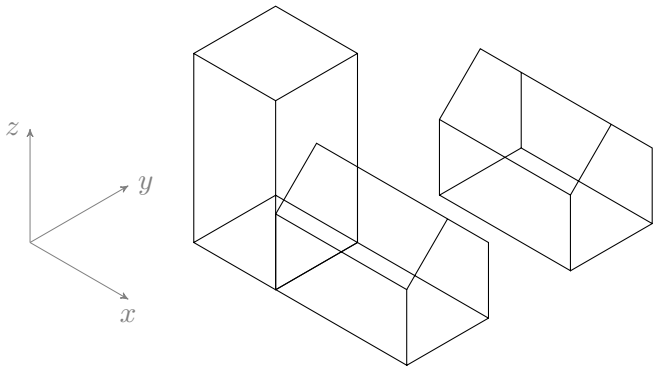


Figure 3.6: Objects copied and transformed into the world space create the virtual world.

The transformations applied to the objects is done in the order scaling $\rightarrow$ rotation $\rightarrow$ translation to preserve the shape of the object. So the world space vertices of each object vertices are calculated using

$$V_{\text{object}} = T \cdot R_z \cdot R_y \cdot R_x \cdot S \cdot V.$$

The object vertex co-ordinates are then appended to a vertex matrix containing all of the objects in the virtual environment, i.e.,

$$V_{\text{world}} = \begin{bmatrix} V_{\text{object 1}} & V_{\text{object 2}} & V_{\text{object 3}} & \cdots \end{bmatrix}.$$

So V_{world} is a $4 \times n$ matrix where n is the total number of vertices that define the virtual environment. The face array for each object is also appended to the bottom of a face matrix containing all faces for the

objects that make up the virtual environment, i.e.,

$$F_{\text{world}} = \begin{bmatrix} F_{\text{object 1}} \\ F_{\text{object 2}} \\ \vdots \end{bmatrix}.$$

So F_{world} is an $p \times q$ matrix where p is the total number of faces in the virtual environment and q is the maximum number of sides of the faces. Since F_{world} links to V_{world} then when adding a new face to F_{world} we need to add the number of columns currently in V_{world} to F_{object}.

Example 3.2

The virtual world shown in fig. 3.6 is constructed using 2 house objects from example 3.1 and a cube object. The house objects are rotated by angle $\theta = \pi/2$ about the z axis and translated so that the centre of the bases have position vectors $(3, 3.5, 0)$ and $(3, 1.5, 0)$. The cube object has side lengths 2 is scaled by a factor of 0.5 in the x and y directions and 1.5 in the z direction so that it resembles a tower and translated so that the centre of the base is at $(1.5, 1.5, 0)$. Determine the vertex and face matrices for the world space.

Solution:
Since $\cos(\pi/2) = 0$ and $\sin(\pi/2) = 1$ then the rotation and translation matrices for the first house object are

$$R_z = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad T = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying these transformations to V_{house} from example 3.1 gives

$$V_{\text{world}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 3.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} 2 & 2 & 4 & 4 & 2 & 2 & 4 & 4 & 2 & 4 \\ 4 & 3 & 3 & 4 & 4 & 3 & 3 & 4 & 3.5 & 3.5 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix},$$

and since this is the first object added to the world space then $F_{\text{world}} = F_{\text{house}}$. Doing similar for the second house object gives the transformed object co-ordinates

$$T \cdot S \cdot V_{\text{house}} = \begin{bmatrix} 2 & 2 & 4 & 4 & 2 & 2 & 4 & 4 & 2 & 4 \\ 2 & 1 & 1 & 2 & 2 & 1 & 1 & 2 & 1.5 & 1.5 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix},$$

which are appended to the end of the world space vertex matrix

$$V_{\text{world}} = \begin{bmatrix} \underbrace{2 & 2 & \cdots & 4}_{\text{first house object}} & \underbrace{2 & 2 & \cdots & 4}_{\text{second house object}} \\ 2 & 1 & \cdots & 3.5 & 2 & 1 & \cdots & 1.5 \\ 0 & 0 & \cdots & 2 & 0 & 0 & \cdots & 2 \\ 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \end{bmatrix}.$$

Since there were 10 columns in V_{world} prior to appending the second house object then we need to

add 10 to F_{house} and append it to F_{world}

$$F_{\text{world}} = \begin{bmatrix} 1 & 4 & 3 & 2 & 2 \\ 1 & 2 & 6 & 9 & 5 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 5 & 9 & 10 & 8 & 8 \\ 11 & 14 & 13 & 12 & 12 \\ 11 & 12 & 16 & 19 & 15 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 15 & 19 & 20 & 18 & 18 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}1\\1\\\vdots\\5\end{matrix}}\right\} \text{first house object} \\ \left.\vphantom{\begin{matrix}11\\11\\\vdots\\15\end{matrix}}\right\} \text{second house object} \end{matrix}$$

The scaling and translation matrices for the cube (tower) object are

$$S = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad T = \begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 1.5 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which gives the transformed vertex matrix

$$T \cdot S \cdot V_{\text{cube}} = \begin{bmatrix} 1 & 2 & 2 & 1 & 1 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Appending these to V_{world} gives

$$V_{\text{world}} = \begin{bmatrix} 2 & 2 & \cdots & 4 & 1 & 2 & \cdots & 1 \\ 2 & 1 & \cdots & 1.5 & 1 & 1 & \cdots & 2 \\ 0 & 0 & \cdots & 2 & 0 & 0 & \cdots & 3 \\ 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \end{bmatrix}.$$

house objects (columns 1–4) | cube object (columns 5–8)

There were 20 columns in V_{world} prior to appending the cube object vertices so we need to add 20 to F_{cube} and append it to F_{world}

$$\begin{bmatrix} 1 & 4 & 3 & 2 & 2 \\ 1 & 2 & 6 & 9 & 5 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 15 & 19 & 20 & 18 & 18 \\ 21 & 24 & 23 & 22 & 22 \\ 21 & 22 & 26 & 25 & 25 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 25 & 26 & 27 & 28 & 28 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}1\\1\\\vdots\\15\end{matrix}}\right\} \text{house objects} \\ \left.\vphantom{\begin{matrix}21\\21\\\vdots\\25\end{matrix}}\right\} \text{cube object} \end{matrix}$$

After the 3 objects have been added to the world space the virtual world is described by 20 polygons defined by 28 vertices.

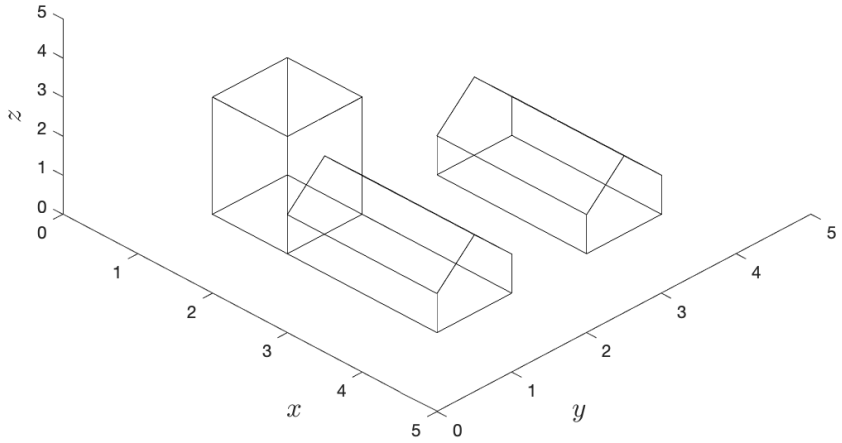


Figure 3.7: A plot of the virtual environment containing the cube and house objects from example 3.2.

3.4 Transforming to the camera space

The next step in the viewing pipeline (fig. 3.1) is to align the world space to the camera space. Imagine you are viewing a virtual environment through a camera positioned in the world space at $\mathbf{p}$ and pointed towards the point with position $\mathbf{c}$ known as the **centre of view** (fig. 3.8). We want to transform the world space so that $\mathbf{p}$ is at the origin of a new space with axes x^*, y^* and z^* where, from our point of view, x^* points to the right, y^* points up and z^* points towards us. The reason we want this new axes configuration is so that the x and y co-ordinates match the horizontal and vertical axes of the display.

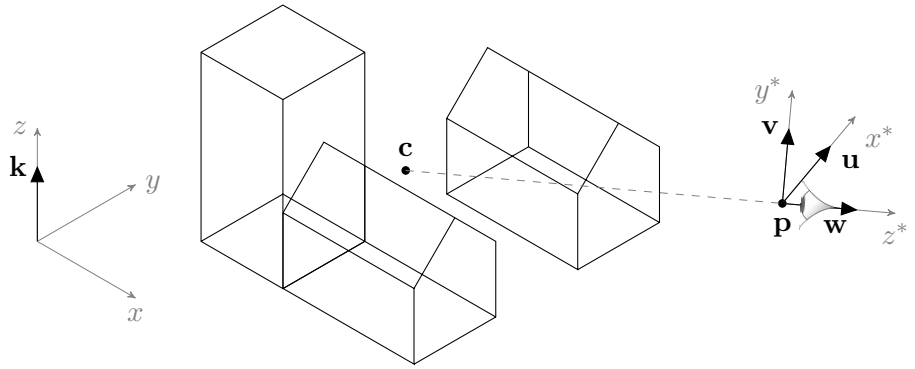


Figure 3.8: The virtual world is transformed so that the viewer position $\mathbf{p}$ is the origin of a new co-ordinate axes x^*, y^* and z^*.

The first transformation is to translate the world space by $-\mathbf{p}$ so that the viewer is positioned at the origin. The matrix that performs this translation is

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Next we need to transform from the (x, y, z) axes to the (x^*, y^*, z^*) axes. Let $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ be a basis for the new axes where $\mathbf{u} = (u_1, u_2, u_3)$, $\mathbf{v} = (v_1, v_2, v_3)$ and $\mathbf{w} = (w_1, w_2, w_3)$ are unit vectors pointing in the x^*, y^* and z^* directions respectively. The $\mathbf{w}$ vector is calculated using

$$\mathbf{w} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|}.$$

The $\mathbf{u}$ vector is perpendicular to the plane that $\mathbf{w}$ and $\mathbf{k}$ lie on so

$$\mathbf{u} = \frac{\mathbf{k} \times \mathbf{w}}{|\mathbf{k} \times \mathbf{w}|}.$$

The $\mathbf{v}$ vector is perpendicular to the plane that $\mathbf{u}$ and $\mathbf{w}$ lie on so

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

Note that the order of the vectors in the cross products when calculating $\mathbf{u}$ and $\mathbf{v}$ are important as the cross product is not commutative. The change of basis matrix for going from the basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ to the new basis $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ is

$$R = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_2 & 0 \\ w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Combining the translation and change of basis matrix gives

$$A = T \cdot R = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_2 & 0 \\ w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & u_3 & -p_x u_1 - p_y u_2 - p_z u_3 \\ v_1 & v_2 & v_2 & -p_x v_1 - p_y v_2 - p_z v_3 \\ w_1 & w_2 & w_3 & -p_x w_1 - p_y w_2 - p_z w_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} u_1 & u_2 & u_3 & -\mathbf{p} \cdot \mathbf{u} \\ v_1 & v_2 & v_2 & -\mathbf{p} \cdot \mathbf{v} \\ w_1 & w_2 & w_3 & -\mathbf{p} \cdot \mathbf{w} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

A is the **alignment transformation matrix** that aligns the world space to the camera space. This is applied to the world space vertex matrix to give the camera space vertex matrix, i.e.,

$$V_{\text{camera}} = A \cdot V_{\text{world}}$$

Note that every time the view position changes or the centre of view changes the alignment transformation matrix will need to be recalculated and applied to calculate the camera space vertices. In a computer game this typically happens 30 or 60 times a second as the player is moving the camera. The player traversing a virtual environment feels like they are moving through the world space, what actually happens is the player remains are the origin looking down the z^* axis and it is the world space that moves around them.

Example 3.3

The world space from example 3.2 is viewed from position $\mathbf{p} = (6, 5, 0.5)$ looking towards the centre of view at $\mathbf{c} = (2, 2, 1)$. Calculate the camera space co-ordinates of the virtual environment.

Solution:
Calculate the basis $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ for the camera space

$$\mathbf{w} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|} = \frac{(4, 3, -0.5)}{\sqrt{4^2 + 3^2 + (-0.5)^2}} = (0.7960, 0.5970, -0.0995),$$

$$\mathbf{u} = \frac{\mathbf{k} \times \mathbf{w}}{|\mathbf{k} \times \mathbf{w}|} = \frac{\det\left(\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0 & 0 & 1 \\ 0.7960 & 0.5970 & -0.0995 \end{bmatrix}\right)}{\left|\det\left(\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0 & 0 & 1 \\ 0.7960 & 0.5970 & -0.0995 \end{bmatrix}\right)\right|} = (-0.6, 0.8, 0),$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u} = \det\left(\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0.7960 & 0.5970 & -0.0995 \\ -0.6 & 0.8 & 0 \end{bmatrix}\right) = (0.0796, 0.0597, 0.9950).$$

So the change of basis matrix is

$$R = \begin{bmatrix} -0.6 & 0.8 & 0 & 0 \\ 0.0796 & 0.0597 & 0.9950 & 0 \\ 0.7960 & 0.5970 & -0.0995 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can check that R is correct by multiplying it by the homogeneous form of $\mathbf{p} - \mathbf{c}$ to see if it points along the positive z^* direction

$$R \cdot (\mathbf{p} - \mathbf{c}) = \begin{bmatrix} -0.6 & 0.8 & 0 & 0 \\ 0.0796 & 0.0597 & 0.9950 & 0 \\ 0.7960 & 0.5970 & -0.0995 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \\ -0.5 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 5.0249 \\ 1 \end{bmatrix}.$$

Since the x and y values are 0 and the z value is positive then R is correct. The alignment transformation matrix is

$$A = \begin{bmatrix} u_1 & u_2 & u_3 & -\mathbf{p} \cdot \mathbf{u} \\ v_1 & v_2 & v_2 & -\mathbf{p} \cdot \mathbf{v} \\ w_1 & w_2 & w_3 & -\mathbf{p} \cdot \mathbf{w} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.6000 & 0.8000 & 0.0000 & -0.4000 \\ 0.0796 & 0.0597 & 0.9950 & -1.2736 \\ 0.7960 & 0.5970 & -0.0995 & -7.7115 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Applying the alignment transformation to the world space co-ordinates

$$V_{\text{camera}} = A \cdot V_{\text{world}} = \begin{bmatrix} -0.6000 & 0.8000 & 0.0000 & -0.4000 \\ 0.0796 & 0.0597 & 0.9950 & -1.2736 \\ 0.7960 & 0.5970 & -0.0995 & -7.7115 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 4 & 4 & \cdots \\ 4 & 3 & 3 & 4 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ 1 & 1 & 1 & 1 & \cdots \end{bmatrix}$$

$$= \begin{bmatrix} 1.6000 & 0.8000 & -0.4000 & 0.4000 & \cdots \\ -0.8745 & -0.9353 & -0.7761 & -0.7164 & \cdots \\ -3.7314 & -4.3284 & -2.7364 & -2.1393 & \cdots \\ 1 & 1 & 1 & 1 & \cdots \end{bmatrix}.$$

A plot of the camera space is shown in fig. 3.9.

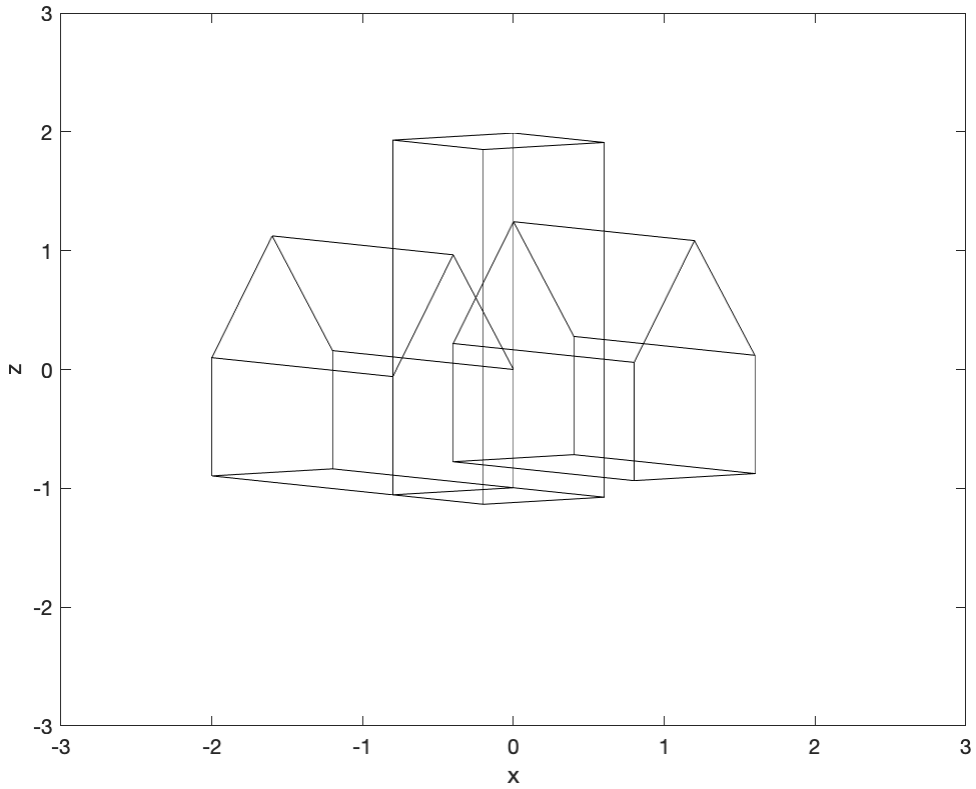


Figure 3.9: A plot of the camera space from the point of view of the viewer looking along the z axis.

3.5 Projecting onto the screen space

After aligning the world space to the viewer we need to transform the three-dimensional camera space to a two-dimensional space that we can represent on a computer display. This is achieved by defining a **projection plane** that is parallel to the x and y axes of the camera space (we have dropped the * suffix from the camera space co-ordinates from this point onwards) and positioned so that intersects with a negative value on the z axes fig. 3.10.

3.5.1 Orthographic projection

The simplest kind of projection is the **orthographic projection**. This is where we simply ignore the z co-ordinate of each point in the environment, and consider the positions of the points (or components of vectors) in the plane to be given by their x and y coordinates (or components).

An orthographic projection can often be carried out directly without the need for any real processing by simply neglecting the z co-ordinate of all the points. However to retain consistency with our previous matrix representations we could carry out this projection with the use of the transformation matrix (again using homogeneous coordinates) P given by

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

A point with a position given by the homogeneous camera space co-ordinates $\mathbf{x} = (x, y, z, 1)$ the orthographic screen space co-ordinates are calculated using

$$P \cdot \mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$

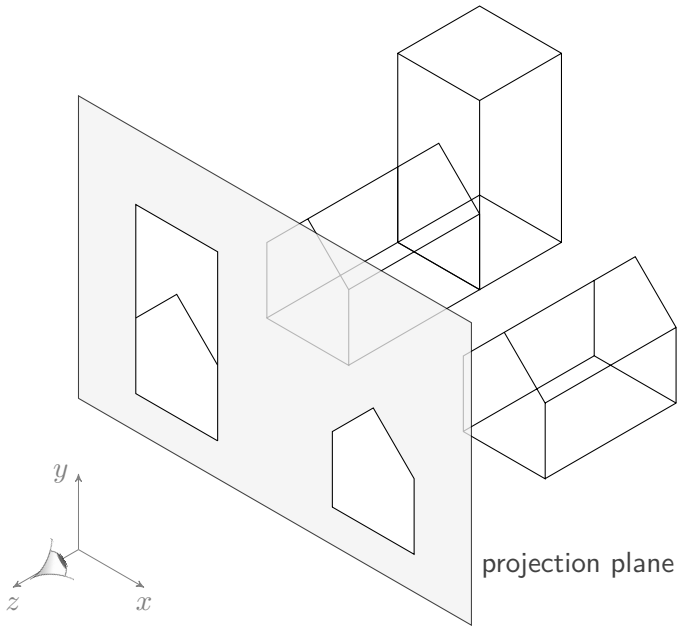


Figure 3.10: The camera space is projected onto a projection plane to give a two-dimensional representation of the virtual world.

This projection is depicted in fig. 3.11. It can be seen there that if each point in the aligned space is joined to its projection in the xy-plane this produces a collection of **projectors**. For orthographic projection these lines are all parallel to the z axis and so intersect the xy-plane orthogonally (i.e. at a right angle).

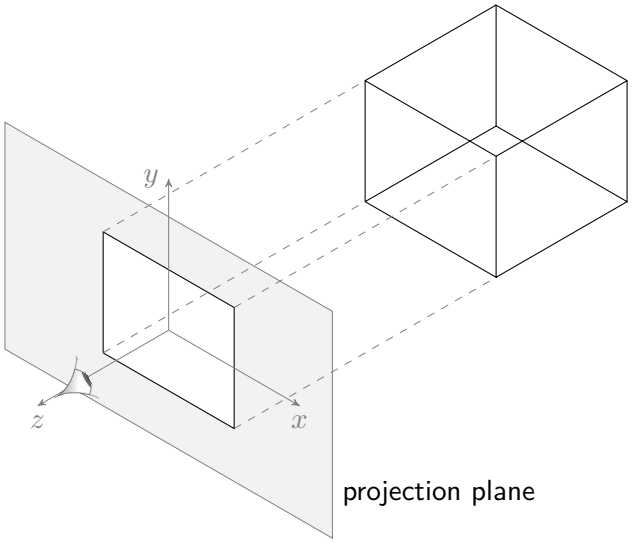


Figure 3.11: Orthographic projection.

3.5.2 Perspective projection

A major drawback of the orthographic projection is that we lose all the depth information and will not be able to tell which object are closer than others. **Perspective projection** does retain depth information by making objects further away from the viewpoint appear smaller in the projection than similar objects closer to the viewpoint. Perspective projection emulates the way the human eye, or a camera, perceives objects. Rays of light are reflected off of objects and travel in straight lines, converging on the eye, or lens, which for these purposes can be thought of as a single point, the viewpoint. Examples of the use of perspective projection can be found in art where artists such as Albrecht Durer have used it to improve the realism of their work (fig. 3.12).

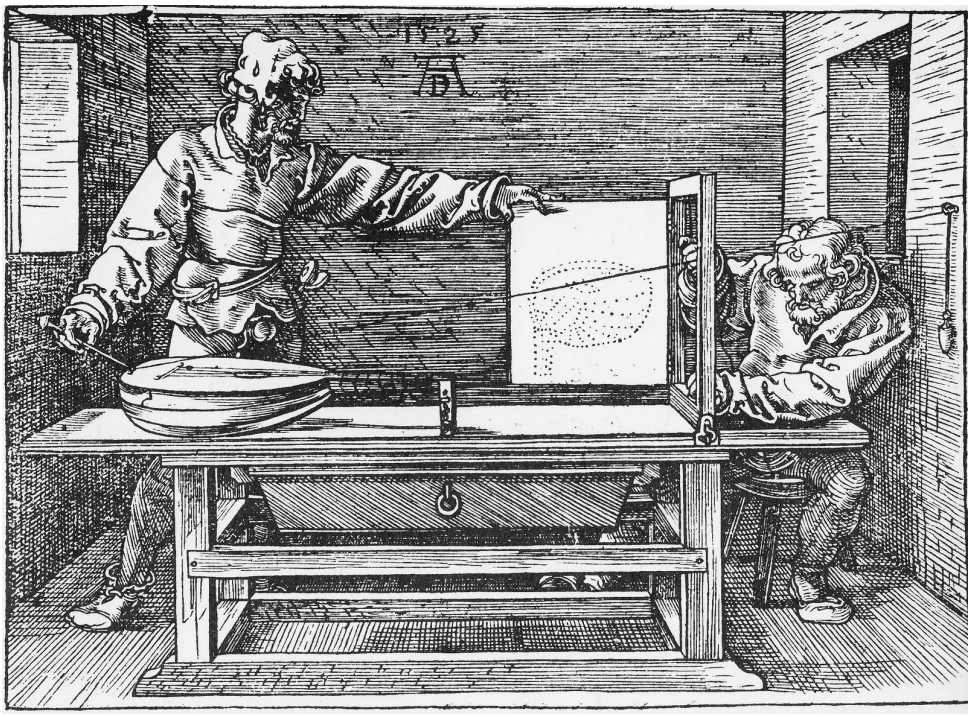


Figure 3.12: An engraving by German artist Albrecht Durer (1471 – 1528) showing himself using perspective projection to draw a lute.

In perspective projection the viewpoint is often called the centre of projection, it is the point where all the projectors meet. The projected image is formed on a projection plane, a plane parallel to the xy plane and positioned at a distance f from the centre of projection (which after alignment is of course positioned at the origin). The projection plane can be thought of as a glass pane held up between the eye and the scene being viewed (fig. 3.13). A projector line from a point in the world space intersects the projection plane in the projected point, and then goes through the centre of projection.

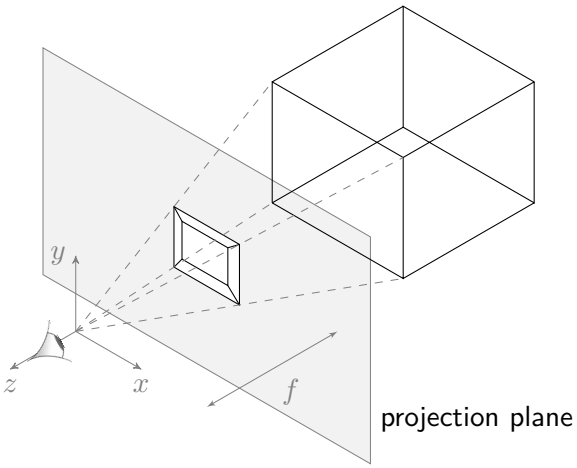


Figure 3.13: Perspective projection.

Consider the diagram shown in fig. 3.14 where the point with co-ordinates (x, y, z) is projected onto the projection plane located at $z = -f$ to give to the point with co-ordinates (x^*, y^*, z^*). The triangle with sides x, y and r is similar to the triangle x^*, y^* and r^* so

$$\frac{x^*}{f} = \frac{x}{z}, \qquad \frac{y^*}{f} = \frac{y}{z},$$

which gives the projected co-ordinates

$$x^* = \frac{fx}{z}, \qquad y^* = \frac{fy}{z}.$$

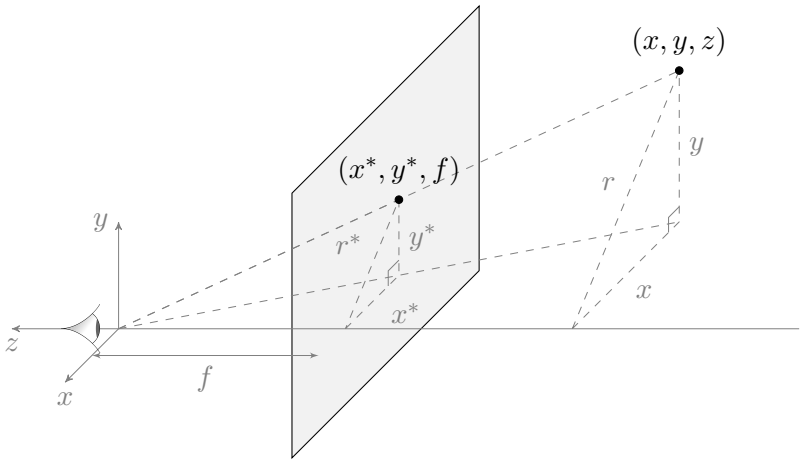


Figure 3.14: The perspective projection of the point with co-ordinates (x, y, z) on to the projection plane located at $z = f$.

Both x^* and y^* are divided by z/f so the homogeneous co-ordinates of the projected point is $(x, y, z, z/f)$ (remember that we divide by the fourth co-ordinates to convert to Cartesian co-ordinates). Therefore the transformation matrix for perspective projection is

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}.$$

The screen space co-ordinates of a point with the homogeneous camera space co-ordinates $\mathbf{x} = (x, y, z, 1)$ are calculated using

$$P \cdot \mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/f \end{bmatrix}$$

and dividing by the fourth co-ordinates gives $(fx/z, fy/z, f, 1)$ which are the projected co-ordinates derived above.

3.5.3 Viewing frustum

When we view a virtual world we will only be able to see objects that are located within a finite region known as the **viewing frustum**. Consider the diagram shown in fig. 3.15. The camera space is projected onto the near viewing plane and we view the virtual environment through the display screen which lies on the near projection plane. If we place another plane parallel to the projection plane further away from the origin then we have a volume which will contain the region of the camera space that should be visible to us. The location of the far viewing plane depends upon the computing power available, the further away it is the more of the camera space we will be able to see but this will of course require more computational resources.

The viewing frustum shown in fig. 3.15 is an awkward shape to deal with when it comes to clipping objects that lie partially outside so we transform it so that the viewing frustum is a cube with sides of lengths 2 parallel to the co-ordinate axes whilst still maintaining the perspective projection. Consider the diagram shown in fig. 3.16(a). The camera space co-ordinates of the vertices of the screen on the near projection plane are $(l, b, near)$, $(r, b, near)$, $(r, t, near)$ and $(l, t, near)$ where l, r, t and b denote left, right, top and bottom edges respectively. The values of these co-ordinates are determined by the position of the near viewing plane, the **field of view** angle fov which controls the horizontal peripheral vision of the viewer and the width-to-height ratio of the screen.

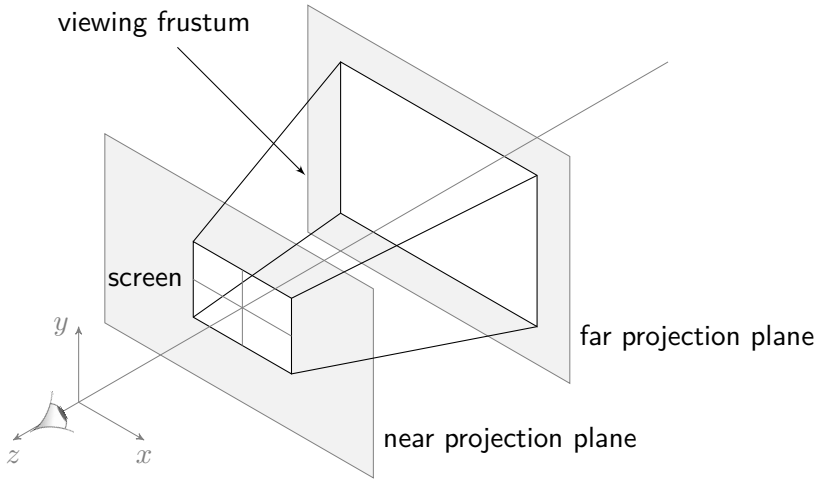
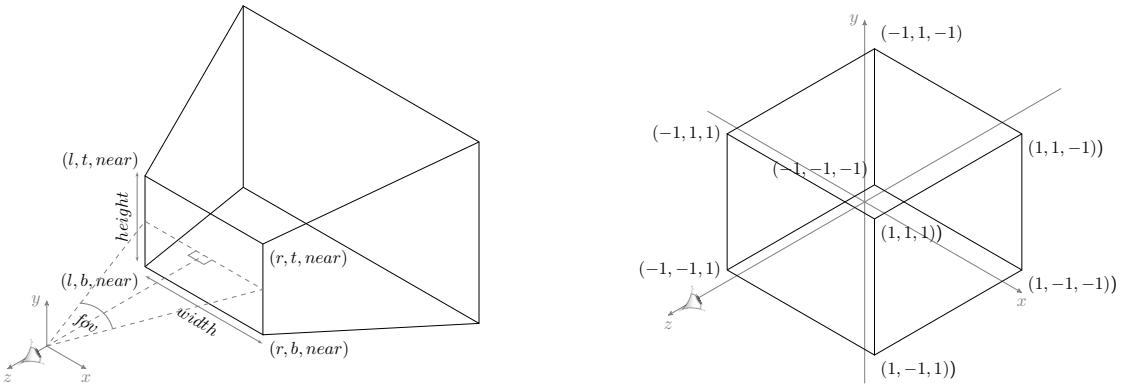


Figure 3.15: The viewing frustum.

We want to transform the viewing frustum so that its sides are parallel to the co-ordinate axes as shown in fig. 3.16(b). The co-ordinates of the left, right, top and bottom corners of the screen are transformed so they are either -1 or $+1$.



(a) viewing frustum

(b) transformed viewing frustum

Figure 3.16: The viewing frustum is transformed so that its edges are parallel to the co-ordinate axes.

The r co-ordinate is calculated using

$$r = |near| \cdot tan\left(\frac{fov}{2}\right),$$

and since the centre of the screen is on the z axis then $l = -r$. Note that the $near$ co-ordinate is negative so we use the absolute value to calculate r. The t co-ordinate is determined by the aspect ratio of the screen, i.e.,

$$aspect = \frac{width}{height},$$

so

$$aspect = \frac{r-l}{t-b} = \frac{r}{t}$$
$$\therefore t = \frac{r}{aspect} = \frac{r \cdot height}{width},$$

Common aspect ratios are 4/3 (for old televisions and computer monitors), 16/9 (modern televisions) and 2.35/1 (cinema screens).

To transform the viewing frustum we need to transform the camera space so that the points within the viewing frustum have x^*, y^* and z^* co-ordinates in the range $-1 \leq x^*, y^*, z^* \leq 1$ and projected using perspective projection. The x camera space co-ordinate for a point that is on the near projection plane and within our screen will be in the range $l \leq x \leq r$. Since $l = -r$ and dividing throughout by r we have

$$-1 \leq \frac{x}{r} \leq 1,$$

and using perspective projection $x^* = x/near$ then

$$-1 \leq \frac{near \cdot x}{rz} \leq 1,$$

so

$$x^* = \frac{near \cdot x}{rz}.$$

Doing similar for y^* gives

$$y^* = \frac{near \cdot y}{tz}.$$

Now x^* and y^* are perspective screen space co-ordinates that are in the range $-1 \leq x^*, y^* \leq 1$. The matrix that performs this transformation is

$$P = \begin{bmatrix} near/r & 0 & 0 & 0 \\ 0 & near/t & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where a and b are some scalars. A point with the homogeneous camera space co-ordinates $\mathbf{x} = (x, y, z, 1)$ the screen space co-ordinates are calculated using

$$P \cdot \mathbf{x} = \begin{bmatrix} near/r & 0 & 0 & 0 \\ 0 & near/t & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} near \cdot x/r \\ near \cdot y/t \\ az + b \\ z \end{bmatrix},$$

and dividing by the fourth co-ordinate to convert to Cartesian co-ordinates we have

$$\begin{bmatrix} near \cdot x/(rz) \\ near \cdot y/(tz) \\ (az + b)/z \\ 1 \end{bmatrix}.$$

The z camera space co-ordinate for a point within the viewing frustum is in the range $near \leq z \leq far$ so we need to transform $near \mapsto 1$ and $far \mapsto -1$. So the minimum and maximum z^* co-ordinates are

$$1 = \frac{a \cdot near + b}{near},$$
$$-1 = \frac{a \cdot far + b}{far}.$$

Solving for a and b gives $a = (near + far)/(near - far)$ and $b = -2 \cdot near \cdot far/(near - far)$ so the transformation matrix becomes

$$P = \begin{bmatrix} near/r & 0 & 0 & 0 \\ 0 & near/t & 0 & 0 \\ 0 & 0 & (near + far)/(near - far) & -2 \cdot near \cdot far/(near - far) \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The transformation matrix P combines perspective projection and transformation of the viewing frustum to a cube. The screen space co-ordinates are calculated using

$$V_{\text{screen}} = P \cdot V_{\text{camera}}.$$

Each column in V_{screen} is a homogeneous screen space co-ordinate where the fourth row contains the scaling factor. The Cartesian co-ordinates are calculated by dividing V_{screen} by the fourth row.

Example 3.4

The camera space from example 3.3 is projected onto the screen space defined by near and far projection plans located at $z_{near} = -2$ and $z_{far} = -10$, a field of view angle of $fov = 1$ and a screen aspect ratio of $aspect = 4/3$. Calculate the screen space co-ordinates of the virtual world.

Solution:
Calculate the r and t co-ordinates

$$r = |near| \cdot \tan\left(\frac{fov}{2}\right) = 2\tan(0.5) = 1.0926,$$
$$t = \frac{r \cdot height}{width} = \frac{1.0926(3)}{4} = 0.8195,$$

so the projection matrix is

$$P = \begin{bmatrix} near/r & 0 & 0 & 0 \\ 0 & near/t & 0 & 0 \\ 0 & 0 & (near+far)/(near-far) & -2 \cdot near \cdot far/(near-far) \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} -2/1.0926 & 0 & 0 & 0 \\ 0 & -2/0.8195 & 0 & 0 \\ 0 & 0 & (-2-10)/(-2+10) & -2(-2)(-10)/(-2+10) \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} -1.8305 & 0 & 0 & 0 \\ 0 & -2.4407 & 0 & 0 \\ 0 & 0 & -1.5 & 5 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Applying the perspective transformation matrix to the camera space co-ordinates from example 3.3

$$V_{\text{screen}} = P \cdot V_{\text{view}}$$
$$= \begin{bmatrix} -1.8305 & 0 & 0 & 0 \\ 0 & -2.4407 & 0 & 0 \\ 0 & 0 & -1.5 & 5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1.6000 & 0.8000 & -0.4000 & 0.4000 & \cdots \\ -0.8745 & -0.9353 & -0.7761 & -0.7164 & \cdots \\ -3.7314 & -4.3284 & -2.7364 & -2.1393 & \cdots \\ 1 & 1 & 1 & 1 & \cdots \end{bmatrix}$$
$$= \begin{bmatrix} -2.9288 & -1.4644 & 0.7322 & -0.7322 & \cdots \\ 2.1371 & 2.2828 & 1.8943 & 1.7485 & \cdots \\ 0.5971 & 1.4926 & -0.8955 & -1.7910 & \cdots \\ -3.7314 & -4.3284 & -2.7364 & -2.1393 & \cdots \end{bmatrix}.$$

Dividing V_{screen} by the fourth row to give the Cartesian screen space co-ordinates

$$V_{screen} = \begin{bmatrix} 0.7849 & 0.3383 & -0.2676 & 0.3423 & \cdots \\ -0.5727 & -0.5274 & -0.6923 & -0.8173 & \cdots \\ -0.1600 & -0.3448 & 0.3273 & 0.8372 & \cdots \\ 1 & 1 & 1 & 1 & \cdots \end{bmatrix}$$

A plot of the screen space is shown in fig. 3.17.

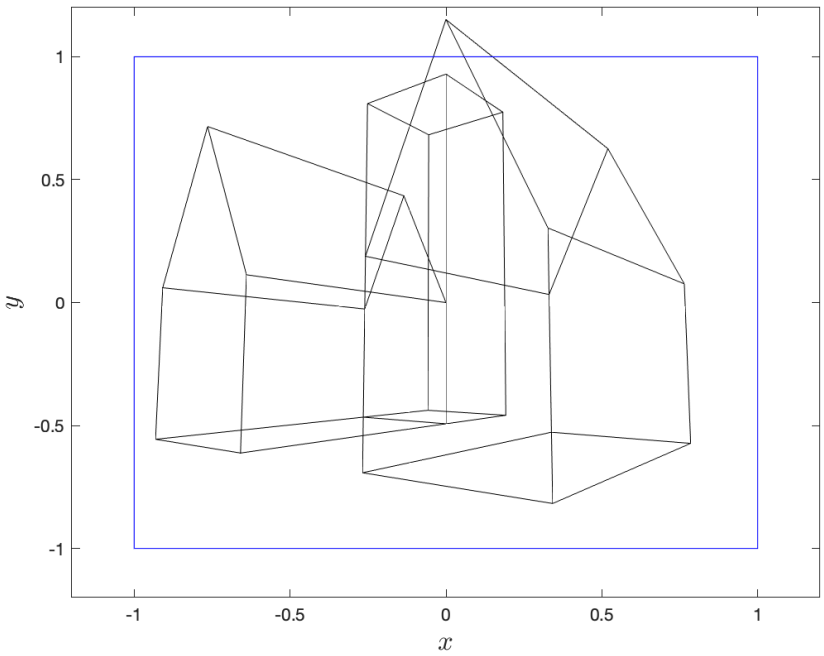


Figure 3.17: A plot of the screen space from the point of view of the viewer looking along the z axis.

3.6 Lab exercises

Use MATLAB to calculate the solutions to examples 3.1 to 3.4 and reproduce the plots shown in figs. 3.5, 3.7, 3.9 and 3.17.

The solutions are given on page 84.

Chapter 4

Clipping and Hidden Surface Removal

4.1 Clipping

We saw in fig. 3.17 on page 57 that after the camera space has been projected onto the screen space some polygons that construct the virtual world may lie wholly or partially outside of the screen space defined by $-1 < x, y, z < 1$. The next step in the viewing pipeline (fig. 3.1 on page 39) is to remove any polygons that are wholly outside of the screen space and to cut polygons that intersected by the planes that define the boundaries of the screen space. This process is known as **clipping**.

The screen space is bounded by a unit cube with vertices $(\pm 1, \pm 1, \pm 1)$ as shown in fig. 3.16(b) so there are 6 planes that need to be considered. A plane is defined by a normal vector $\mathbf{n}$ and the position $\mathbf{p}$ of a point which lies on the plane. The normal vectors for the edges of the screen space are assumed to point towards the interior of the screen space so

$$\mathbf{n}_{\text{bottom}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad \mathbf{n}_{\text{front}} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \qquad \mathbf{n}_{\text{right}} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix},$$

$$\mathbf{n}_{\text{back}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{n}_{\text{left}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{n}_{\text{top}} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix},$$

where the edges labelled bottom, front, right, back, left and top are from the point of view of the viewer looking down the negative direction of the z axis. The position vector $\mathbf{p}$ for the edges of the screen space can be any point on the plane so for simplicity we can use

$$\mathbf{p}_{\text{bottom}} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \qquad \mathbf{p}_{\text{front}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{p}_{\text{right}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},$$

$$\mathbf{p}_{\text{back}} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \qquad \mathbf{p}_{\text{left}} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{p}_{\text{top}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},$$

For each edge of the screen space we consider each polygon of the virtual world as defined by the world space face matrix, F_{world}, and apply an algorithm called the **Sutherland-Hodgman algorithm**. The algorithm works by considering each edge of the polygon joining sequential vertices $\mathbf{v}_i \rightarrow \mathbf{v}_{i+1}$. If $\mathbf{v}_i$ is in front of the screen edge (i.e., it is in the space which the polygon normal vector points towards) then $\mathbf{v}_i$ is added to a list of the vertices in the clipped polygon and if $\mathbf{v}_{i+1}$ is behind the edge then the line $\mathbf{v}_i \rightarrow \mathbf{v}_{i+1}$ intersects the screen edge so the co-ordinates of the intersection point are calculated and added to the clip polygon list. If however $\mathbf{v}_i$ is behind the screen edge and $\mathbf{v}_{i+1}$ is in front then the line $\mathbf{v}_i \rightarrow \mathbf{v}_{i+1}$ intersects the screen edge so the co-ordinates of the intersection point are calculated and added to the clip polygon list.

If not then both $\mathbf{v}_i$ and $\mathbf{v}_{i+1}$ are behind the screen edge so are removed from the polygon. This algorithm is presented in algorithm 1.

Algorithm 1 Sutherland-Hodgman algorithm

```
Initialise V_clip ← V_screen, F_clip ← V_world and n ← number of vertices in V_clip
for each edge of the screen space do
    for each polygon in F_clip do
        Clear list
        for each polygon edge v_i → v_{i+1} do
            if v_i is in front of the screen edge then
                Add i to list
                if v_{i+1} is behind the screen edge then
                    Calculate the intersection point v_{n+1} and append to V_clip
                    Update n ← n + 1 and add to list
                end if
            else if v_{j+1} is in front of the screen edge then
                Calculate the intersection point v_{n+1} and append to V_clip
                Update n ← n + 1 and add to list
            end if
        end for
        Replace the appropriate row of F_clip with list
    end for
end for
```

For the Sutherland-Hodgman algorithm we need to be able to determine whether a point is in front or behind the plane which forms the edges of the screen space. To do this we can use eq. (1.15) on page 19 to calculate the distance between a point and a plane and if this distance is positive then the point is in front of the plane and if it is negative the point is behind the plane. This calculation is made easy since we have transformed the screen space so that it is a unit cube and we just need to compare the co-ordinates to ± 1.

4.1.1 Calculating the intersection between a line and a plane

We also need to be able to calculate the co-ordinates of the intersection point between a line and plane. Consider the diagram in fig. 4.1 which shows a straight line between the two endpoints with positions $\mathbf{a}$ and $\mathbf{b}$ which intersects a plane passing through the point $\mathbf{p}$ with normal vector $\mathbf{n}$. The point at which the line intersects with the plane is at position $\mathbf{c}$.

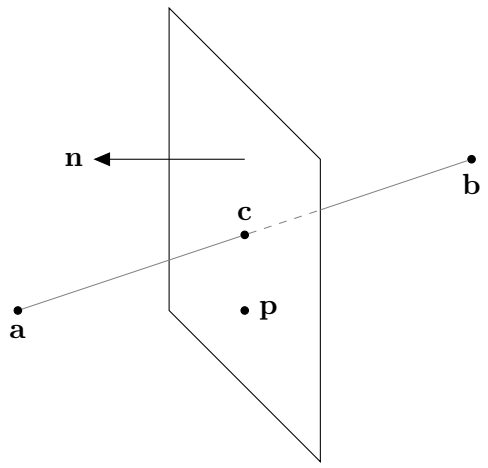


Figure 4.1: Intersection between a line and a plane.

Since $\mathbf{c}$ lies on the line then it must satisfy the vector equation of a line

$$\mathbf{c} = \mathbf{a} + t(\mathbf{b} - \mathbf{a}), \tag{4.1}$$

where $0 \leq t \leq 1$. Since $\mathbf{c}$ also lies on the plane then the vector $\mathbf{c} - \mathbf{p}$ and $\mathbf{n}$ are perpendicular so

$$(\mathbf{c} - \mathbf{p}) \cdot \mathbf{n} = 0, \tag{4.2}$$

Substituting eq. (4.1) into eq. (4.2) and solving for t

$$\begin{aligned}(\mathbf{a} + t(\mathbf{b} - \mathbf{a}) - \mathbf{p}) \cdot \mathbf{n} &= 0 \\ (\mathbf{a} - \mathbf{p}) \cdot \mathbf{n} + t(\mathbf{b} - \mathbf{a}) \cdot \mathbf{n} &= 0 \\ \therefore t &= \frac{(\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}}{(\mathbf{b} - \mathbf{a}) \cdot \mathbf{n}}.\end{aligned}$$

Note that this equation for t is only valid when $(\mathbf{b} - \mathbf{a}) \cdot \mathbf{n} \neq 0$. If $(\mathbf{b} - \mathbf{a}) \cdot \mathbf{n} = 0$ then the line is parallel to the plane and the never intersects the plane. Once the value of t has been determined the point that the line intersects with the plane is calculated using eq. (4.1).

Example 4.1

Apply the Sutherland-Hodgman polygon clipping algorithm to the fourth polygon in the screen space in example 3.4 where F_{clip} and V_{clip} are

$$F_{\text{clip}} = \begin{matrix}1\\ \\4\\ \\20\end{matrix}\begin{bmatrix}1 & 4 & 3 & 2 & 2\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ 3 & 4 & 8 & 10 & 7\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ 25 & 26 & 27 & 28 & 28\end{bmatrix},$$

$$V_{\text{clip}} = \begin{matrix}\\ \\ \\ \\ \end{matrix}\begin{matrix}\cdots & \mathbf{v}_3 & \cdots & \mathbf{v}_4 & \cdots & \mathbf{v}_7 & \cdots & \mathbf{v}_8 & \cdots & \mathbf{v}_{10} & \cdots\end{matrix}\begin{bmatrix}\cdots & -0.2676 & \cdots & 0.3423 & \cdots & -0.2582 & \cdots & 0.3270 & \cdots & 0.0000 & \cdots\\ \cdots & -0.6923 & \cdots & -0.8173 & \cdots & 0.1884 & \cdots & 0.3037 & \cdots & 1.1513 & \cdots\\ \cdots & 0.3273 & \cdots & 0.8372 & \cdots & 0.2631 & \cdots & 0.7333 & \cdots & 0.3962 & \cdots\\ \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots\end{bmatrix}.$$

Solution:
We begin by initialising the clip space vertex and face matrices $V_{\text{clip}} = V_{\text{screen}}$ and $F_{\text{clip}} = F_{\text{world}}$. By inspection all of the x and z co-ordinates are between -1 and 1 but one of the y co-ordinates is greater than 1 so we need to clip to the top edge of the screen. The first 3 vertices for this polygon are $\mathbf{v}_3$, $\mathbf{v}_4$ and $\mathbf{v}_8$ which all have y co-ordinates less that 1 so are in front and added to the list of the clipped polygon

$$list = \begin{bmatrix}3 & 4 & 8\end{bmatrix}.$$

The next vertex of the polygon is $\mathbf{v}_{10}$ which has y co-ordinate $y = 1.1513 > 1$ so is behind the top edge of the screen and we need to calculate the intersection point between the line $\mathbf{v}_8 \rightarrow \mathbf{v}_{10}$ and the top edge of the screen space. Calculating the new intersection point $\mathbf{v}_{29}$ (since there currently

28 vertices in V_{clip})

$$t = \frac{(\mathbf{v}_8 - \mathbf{p}_{\text{top}}) \cdot \mathbf{n}_{\text{top}}}{(\mathbf{v}_8 - \mathbf{v}_{10}) \cdot \mathbf{n}_{\text{top}}} = \frac{\left(\begin{bmatrix}0.3270\\0.3037\\0.7333\end{bmatrix} - \begin{bmatrix}0\\1\\0\end{bmatrix}\right) \cdot \begin{bmatrix}0\\-1\\0\end{bmatrix}}{\left(\begin{bmatrix}0.3270\\0.3037\\0.7333\end{bmatrix} - \begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix}\right) \cdot \begin{bmatrix}0\\-1\\0\end{bmatrix}} = \frac{0.6963}{0.8476} = 0.8215,$$

$$\mathbf{v}_{29} = \mathbf{v}_8 + t(\mathbf{v}_{10} - \mathbf{v}_8) = \begin{bmatrix}0.3270\\0.3037\\0.7333\end{bmatrix} + 0.8215\left(\begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix} - \begin{bmatrix}0.3270\\0.3037\\0.7333\end{bmatrix}\right) = \begin{bmatrix}0.0584\\1\\0.4564\end{bmatrix}.$$

The new vertex $\mathbf{v}_{29}$ is appended to the end of V_{clip} and added to the face list

$$list = \begin{bmatrix}3 & 4 & 8 & 29\end{bmatrix}.$$

The next vertex in the polygon is $\mathbf{v}_7$. We know that $\mathbf{v}_{10}$ is behind the top edge and $\mathbf{v}_7$ is in front so we need to calculate the intersection point $\mathbf{v}_{30}$ between the line $\mathbf{v}_{10} \rightarrow \mathbf{v}_7$ and the top edge of the screen space.

$$t = \frac{(\mathbf{v}_{10} - \mathbf{p}_{\text{top}}) \cdot \mathbf{n}_{\text{top}}}{(\mathbf{v}_{10} - \mathbf{v}_7) \cdot \mathbf{n}_{\text{top}}} = \frac{\left(\begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix} - \begin{bmatrix}0\\1\\0\end{bmatrix}\right) \cdot \begin{bmatrix}0\\-1\\0\end{bmatrix}}{\left(\begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix} - \begin{bmatrix}-0.2582\\0.1884\\0.2631\end{bmatrix}\right) \cdot \begin{bmatrix}0\\-1\\0\end{bmatrix}} = \frac{-0.1513}{-0.9629} = 0.1571,$$

$$\mathbf{v}_{30} = \mathbf{v}_{10} + t(\mathbf{v}_7 - \mathbf{v}_{10}) = \begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix} + 0.1571\left(\begin{bmatrix}-0.2582\\0.1884\\0.2631\end{bmatrix} - \begin{bmatrix}0.0000\\1.1513\\0.3962\end{bmatrix}\right) = \begin{bmatrix}-0.1011\\1\\0.3753\end{bmatrix}.$$

The new vertex $\mathbf{v}_{30}$ is appended to the end of V_{clip} and added to the face list

$$list = \begin{bmatrix}3 & 4 & 9 & 29 & 30\end{bmatrix}.$$

$\mathbf{v}_7$ is in front and added to the face list. The final face list is

$$list = \begin{bmatrix}3 & 4 & 8 & 29 & 30 & 7\end{bmatrix},$$

which replaces the fourth row in F_{clip}. By clipping this polygon to the top edge we now have a 6 sided polygon, so we need to extend F_{clip} so that it has 6 columns.

$$F_{\text{clip}} = \begin{matrix}1\\4\\20\end{matrix}\begin{bmatrix}1 & 4 & 3 & 2 & 2 & 2\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ 3 & 4 & 8 & 29 & 30 & 7\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ 25 & 26 & 27 & 28 & 28 & 28\end{bmatrix},$$

$$V_{\text{clip}} = \begin{matrix}\\\\\\\end{matrix}\begin{matrix}\mathbf{v}_3 & \mathbf{v}_4 & & \mathbf{v}_7 & \mathbf{v}_8 & & \mathbf{v}_{10} & & \mathbf{v}_{29} & \mathbf{v}_{30}\end{matrix}\begin{bmatrix}\cdots & -0.2676 & 0.3423 & \cdots & -0.2582 & 0.3270 & \cdots & 0.0000 & \cdots & 0.0584 & -0.1011\\ \cdots & -0.6923 & -0.8173 & \cdots & 0.1884 & 0.3037 & \cdots & 1.1513 & \cdots & 1.0000 & 1.0000\\ \cdots & 0.3273 & 0.8372 & \cdots & 0.2631 & 0.7333 & \cdots & 0.3962 & \cdots & 0.4564 & 0.3753\\ \cdots & 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & \cdots & 1 & 1\end{bmatrix}.$$

We do similar for all other polygons in F_{clip}. This means that the number of vertices in the clip space vertex matrix may increase and the number of polygons in the clip space face matrix may decrease as polygons that are wholly behind the edge of the screen space are discarded. A plot of the clip space is shown in fig. 4.2.

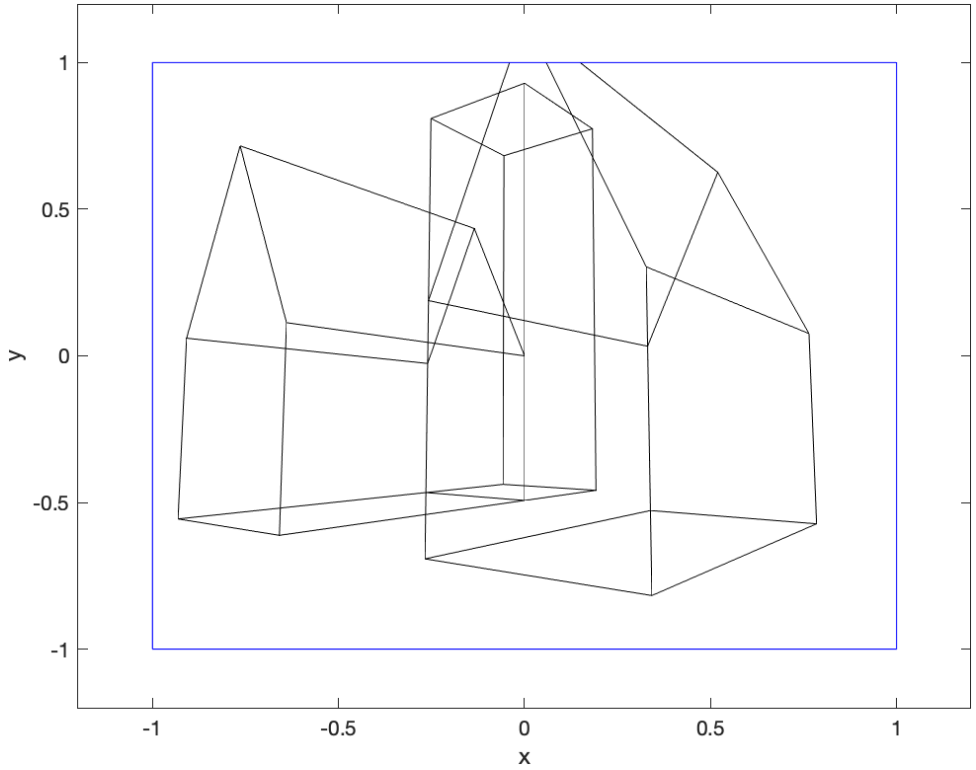


Figure 4.2: A plot of the clip space.

4.2 Hidden surface removal

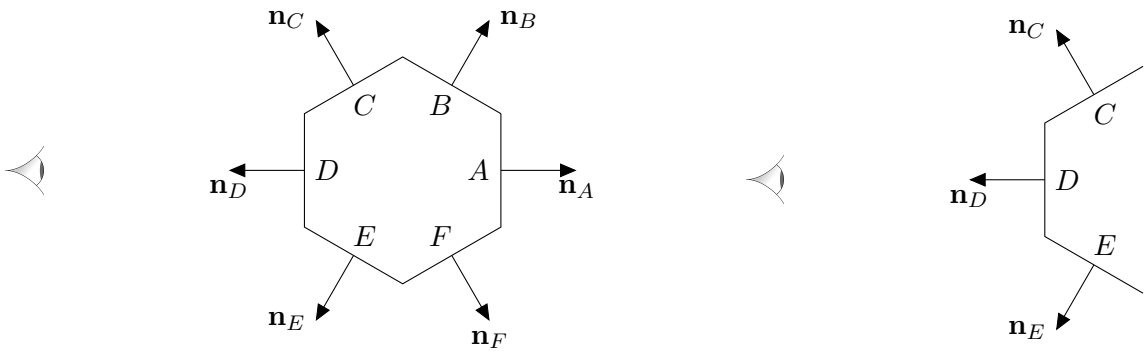
The next step in the viewing pipeline (fig. 3.1 on page 39) is to remove those polygons and parts of polygons that should not be visible to the viewer. The polygons in the plot of the clip space in fig. 4.2 are transparent so better show the virtual environment, if they were opaque the tower object would be mostly obscured by the house object that is closer to the viewer. We will look at three hidden surface removal techniques: back-face culling, painter's algorithm and binary space partitioning.

4.2.1 Back-face culling

The first hidden surface technique that is applied is **back-face culling** which as the name suggests, culls all polygons that are considered back-facing from the point of view of the camera from the screen space. In computer graphics, the normal vector for a polygon is considered unique so that we can distinguish between the two sides of the polygon (see section 1.3.3). The side of the polygon which the normal vector is pointing is known as the **front face** of the polygon and the side facing away from the normal vector is known as the **back face**.

Consider fig. 4.3 where a hexagonal object is defined by 6 polygons A to F. When viewed from the position on the left, polygons C, D and E are front facing and polygons A, B and F are back facing. Assuming that all polygons are opaque, removing (or culling) the back facing polygons from the object does not change the appearance of the object to the viewer.

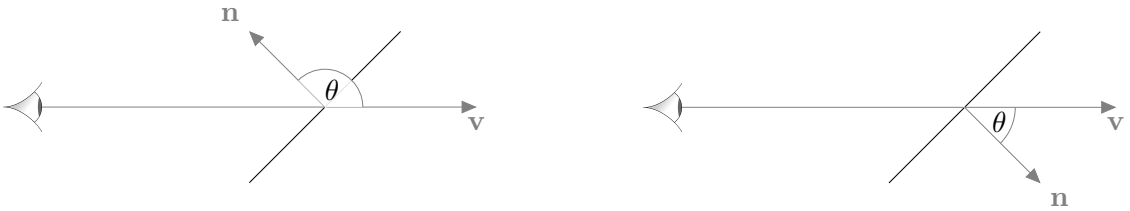
To determine whether a polygon is front or back facing we use the normal vector for the polygon and a **viewing vector** which is a vector pointing from the viewer to a point on the polygon. Consider the diagram



(a) all faces

(b) back-faces culled

Figure 4.3: Culling the back facing polygons does not change the appearance of the object to the viewer.



(a) font facing polygon

(b) back facing polygon

Figure 4.4: Front and back facing polygons

shown in fig. 4.4(a) that shows a front facing polygon. The angle, θ, between the viewing vector $\mathbf{v}$ and the polygon normal vector $\mathbf{n}$ is greater than $\pi/2$ whereas this angle for a back facing polygon shown in fig. 4.4(b) is less than $\pi/2$. Recall the geometric definition of the dot product

$$\mathbf{n} \cdot \mathbf{v} = |\mathbf{n}||\mathbf{v}| \cos(\theta),$$

when $\theta < \pi/2$, $\cos(\theta) > 0$ and when $\theta < \pi/2$, $\cos(\theta) < 0$, therefore a polygon is front facing if

$$\mathbf{n} \cdot \mathbf{v} < 0.$$

If we consider that the viewer is positioned on the z axis at $(0, 0, 1)$ and looking in the negative z direction then any polygon in the clip space with a normal vector $\mathbf{n} = (n_x, n_y, n_z)$ where $n_z > 0$ is front facing.

Algorithm 2 Back-face culling

```
for every polygon in F_clip do
    calculate the normal vector n = (n_x, n_y, n_z)
    if n_z > 0 then
        Add polygon to F_Front
    end if
end for
```

Example 4.2

Apply the back-face culling to the clip space from example 4.1 where the clip space vertex and face

matrices are

$$V_{\text{clip}} = \begin{bmatrix} 0.7849 & 0.3383 & -0.2676 & 0.3423 & \cdots \\ -0.5727 & -0.5274 & -0.6923 & -0.8173 & \cdots \\ -0.1600 & -0.3448 & 0.3273 & 0.8372 & \cdots \\ 1.0000 & 1.0000 & 1.0000 & 1.0000 & \cdots \end{bmatrix},$$

$$F_{\text{clip}} = \begin{bmatrix} 1 & 4 & 3 & 2 & 2 & 2 \\ 1 & 2 & 6 & 9 & 5 & 5 \\ 2 & 3 & 7 & 6 & 6 & 6 \\ 3 & 4 & 8 & 29 & 30 & 7 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Solution:
Calculating the normal vector for the first polygon

$$\begin{aligned} \mathbf{n} &= (\mathbf{v}_4 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_4) \\ &= \left(\begin{bmatrix} 0.3423 \\ -0.8173 \\ 0.8372 \end{bmatrix} - \begin{bmatrix} 0.7849 \\ -0.5727 \\ -0.1600 \end{bmatrix} \right) \times \left(\begin{bmatrix} -0.2676 \\ -0.6923 \\ 0.3273 \end{bmatrix} - \begin{bmatrix} 0.3432 \\ -0.8173 \\ 0.8372 \end{bmatrix} \right) \\ &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ -0.4426 & -0.2446 & 0.9972 \\ -0.6098 & 0.1251 & -0.5099 \end{vmatrix} = \begin{bmatrix} 0.0000 \\ -0.8338 \\ -0.2045 \end{bmatrix}. \end{aligned}$$

Since $n_z = -0.2045 < 0$ then this polygon is back facing and not added to F_{front}. This is repeated for the other polygons in the clip space and a plot of the front facing polygons is shown in fig. 4.5.

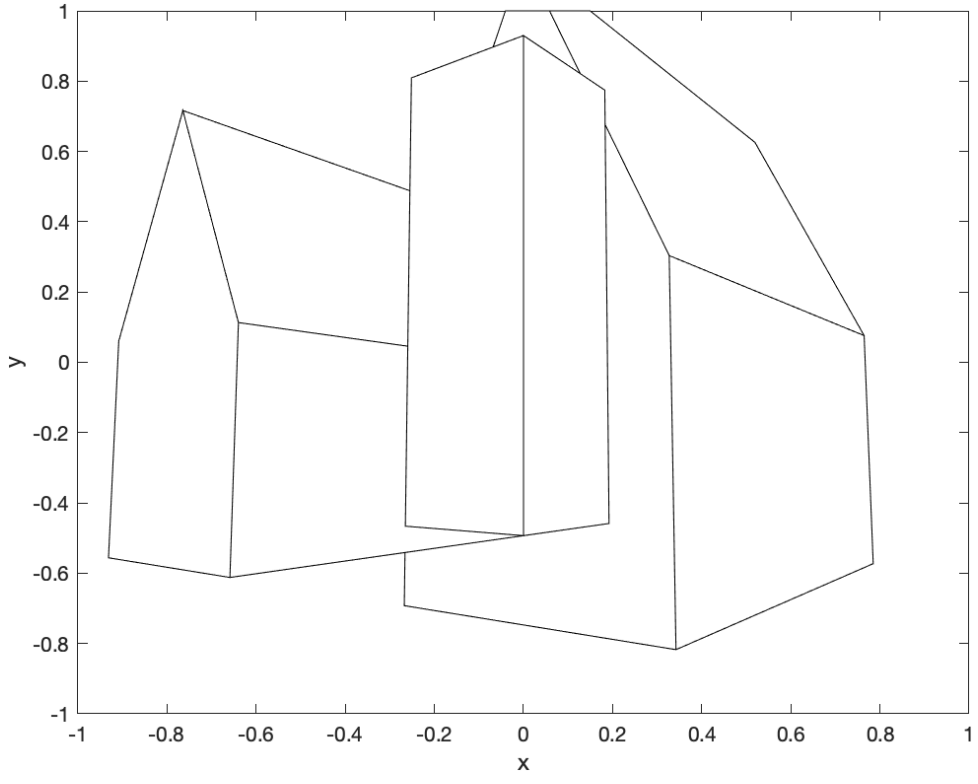


Figure 4.5: The clip space after back facing polygons have been removed (plotted using opaque polygons).

4.3 Painter's algorithm

We have seen in fig. 4.5 that simply culling the back facing polygons from the clip space does not result in a realistic view of the virtual world. This is because the polygons plotted first are obscured by those plotted afterwards. To overcome this we need to sort the polygons by order the distance from the viewer so the polygons further away are plotted first and the polygons closest to the viewer are plotted last. This method has been given the name the **painter's algorithm** because an oil painter painting a scene needs to begin with the background elements before painting the middle ground and foreground elements fig. 4.6.

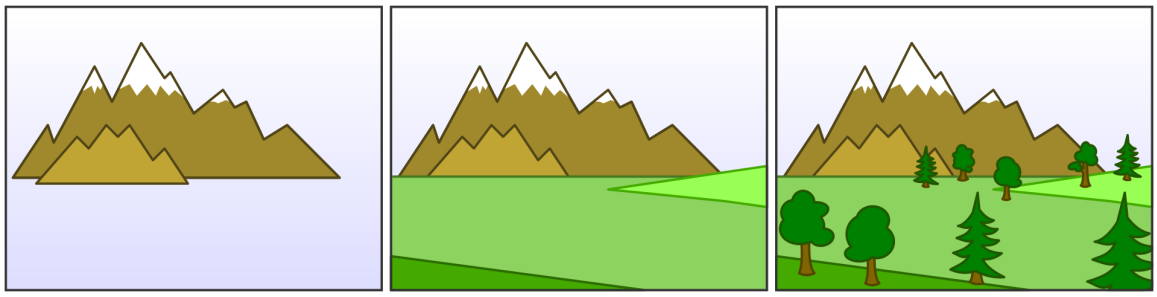


Figure 4.6: An oil painter painting a scene need to paint background elements first and foreground elements last.

Since the clip space is viewed from the position at $(0, 0, 1)$ looking in the negative z axis direction the polygons furthest away will have vertices with z co-ordinates close to -1 and those polygons that are closest to the viewer will have z values close to 1. Therefore we sort the polygons in ascending order of their z co-ordinates and plot the polygons in that order. Since polygons have multiple vertices with possibly different z co-ordinates we assume the polygon has a distance which is that of the vertex with the largest z co-ordinate.

Algorithm 3 Painter's algorithm

for each polygon i in F_{front} **do**
  $z_i \leftarrow -2$
  for each vertex $\mathbf{v} = (v_x, v_y, v_z)$ in the polygon **do**
    $z_i \leftarrow \max(z_i, v_z)$
  end for
end for
Plot polygons in ascending order of z value

The result of the painters algorithm when applied to the clip space from example 4.2 is shown in fig. 4.7. Here the hidden surfaces have been removed and we have a realistic view of the virtual world.

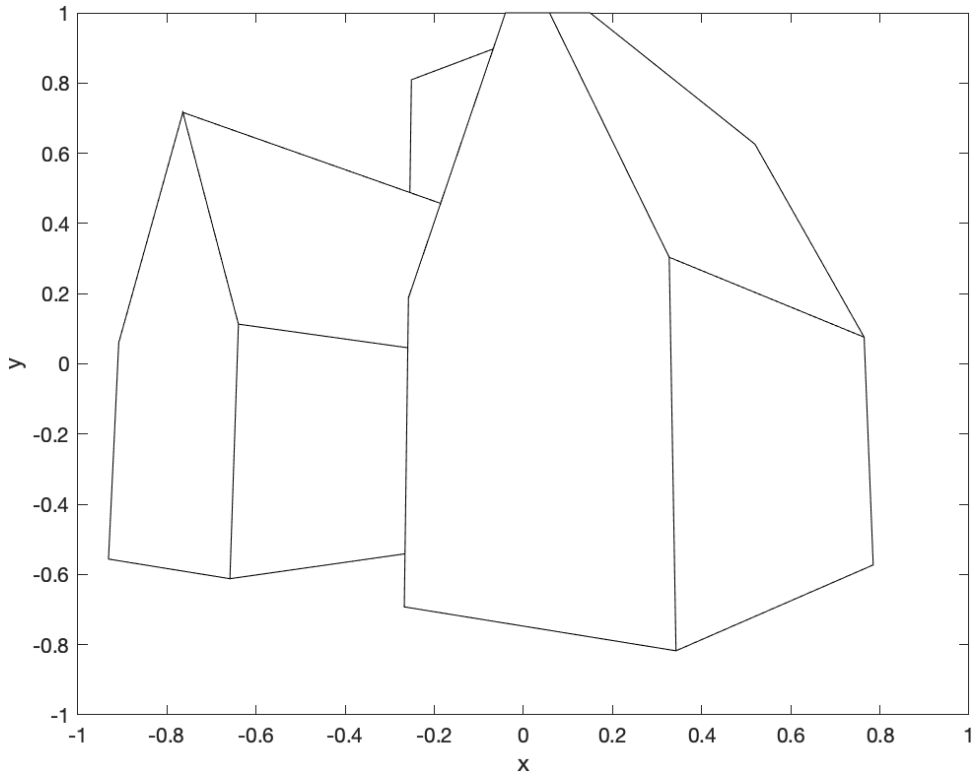


Figure 4.7: A plot of the clip space with back facing polygons culled and sorted in ascending order by their z co-ordinates.

4.4 Binary space partitioning

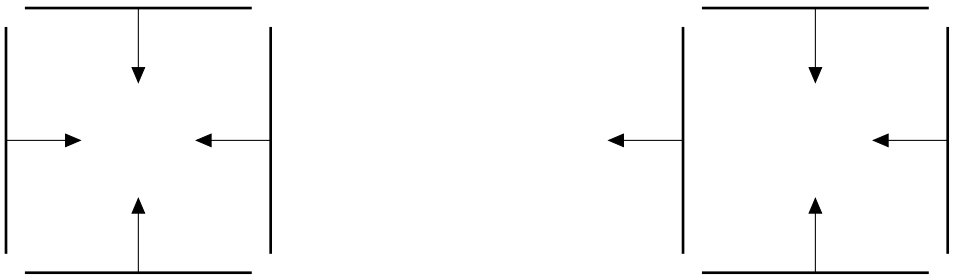
The disadvantage of the painter's algorithm is that every time the camera position or direction changes the z distances need to be recalculated. A method used in computer games that provides the correct rendering order for static polygons in a virtual environment is called Binary Space Partitioning (BSP). BSP was first described in the 1960s by Schumaker et al. (1969) to improve the rendering of three-dimensional scenes using computer graphics. However, it wasn't until the early 1990s that BSP became widely used in the computer game industry to improve the performance of rendering three-dimensional scenes. id Software's seminal game *Doom* (Carmack and Romero 1993) was the first game to use this method and all games with three-dimensional virtual worlds have since have used BSP.

Definition 4.1

A **convex set** is a set of polygons where every polygon is facing every other polygon. A polygon A is said to be facing another polygon B if the surface normal vector is pointing towards B.

Consider the two spaces in fig. 4.9. The set of polygons on the left is a convex set since every polygon in the set has its normal vector pointing to every other polygon in the set. The set of polygons on the right is not a convex set since one of the polygons has its normal vector pointing away from the other polygons in the set.

Figure 4.8: A screenshot from the game *Doom* (Carmack and Romero 1993) which was the first game to use binary space partitioning.



(a) convex set (b) non-convex set

Figure 4.9: The difference between a convex and non-convex set.

Definition 4.2

A **hyperplane** in n-dimensional space is an $(n-1)$-dimensional object that is used to bisect the space to form two new subspaces.

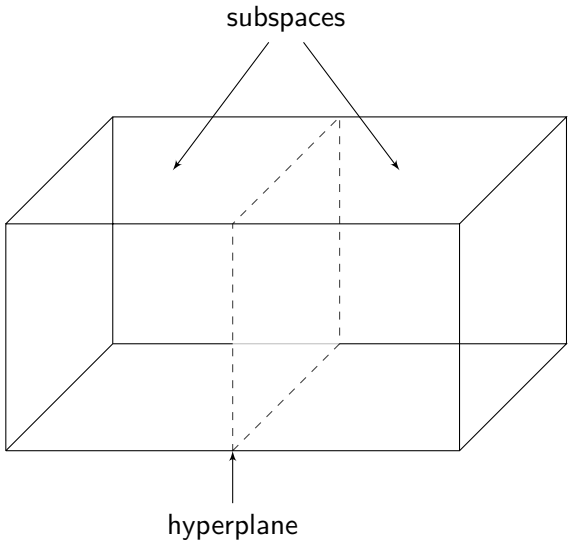


Figure 4.10: A hyperplane divides a space into two subspaces.

Definition 4.3

An **atomic subspace** is a subspace where all polygons contained within the subspace form a convex set.

The aim of binary space partitioning is to divide the world space into atomic subspaces by inserting hyperplanes along polygons. By definition, the polygons that form each atomic subspace do not obscure each other so can be rendered at the same time. The order that each atomic subspace is rendered is calculated to ensure that hidden surfaces are removed in a similar way to the painters algorithm.

Binary space partitioning is primarily used in the rendering of static polygons in the world space. For example, consider a computer game that requires the player to navigate a three-dimensional virtual environment. The polygons that construct the walls, floors, buildings etc. are known prior to the player navigating the map and remain fixed in place. So we can perform binary space partitioning before the scene needs to be rendered in order to save computational effort whilst the game is being played.

4.4.1 BSP trees

When using binary space partitioning we need to record each subdivision and the polygons that are contained within each subspace. To this a data structure known as a **binary tree** is used. A binary tree consists of **nodes** that are joined by **edges** where each node has a single input edge and at most two output edges (fig. 4.11). The node attached to the other end of the input edge is called the **parent node** and the node attached to the other end of the output edges are called **child nodes**. For example in fig. 4.11, A is the root node with two child nodes B and C. B is the parent of nodes D and E and C is the parent of nodes F and G. Nodes D, E, F and G have no child nodes so are therefore leaf nodes.

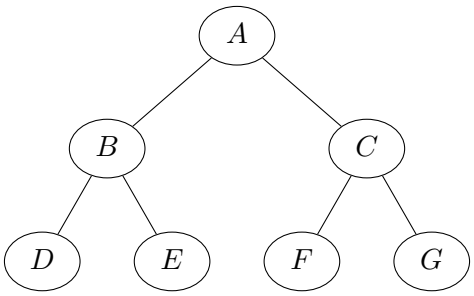


Figure 4.11: A simple binary tree

Consider the space containing the three polygons A, B and C in fig. 4.12. The set $\{A, B, C\}$ is not an atomic subspace since none of the three polygons face each other. If we subdivide the space by inserting a polygon on the same plane that polygon A lies on then, since the normal vector for polygon A points to the right, polygon B is in the front subspace and polygon C is in the back subspace. We represent this subdivision as a binary tree with polygon A contained in the root node, polygon B contained in the left child node and polygon C contained within the right child node fig. 4.12. The choice of which node to insert the hyperplane along is arbitrary and we can select any polygon in the set, however, there may be optimality considerations.

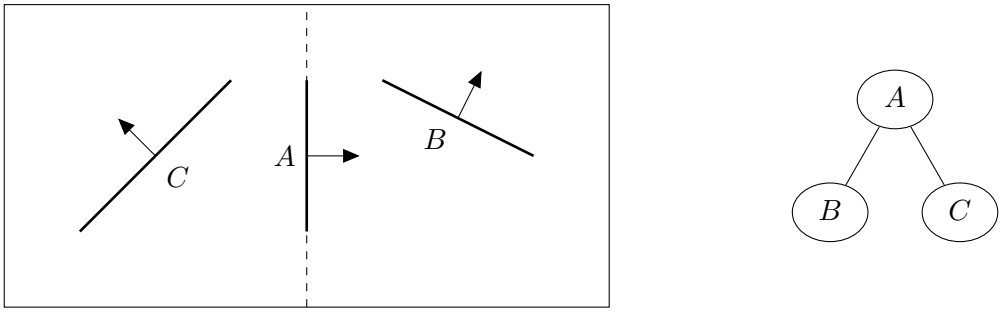


Figure 4.12: Polygons in the front subspace are listed in the left child node and polygons in the back subspace are listed in the right child node in a BSP tree.

Definition 4.4: Coincident polygons

If two or more polygons exist on the same plane and they are facing in the same direction then they are said to be **coincident**.

Coincident polygons can be treated as one polygon in the BSP process. It is advantageous to use coincident polygons as the ones which a hyperplane is inserted since it reduces the number of polygons we have to deal with. For example, consider fig. 4.13 where polygons A and B are coincident. Polygon C is in the front subspace and polygon D is in the back subspace.

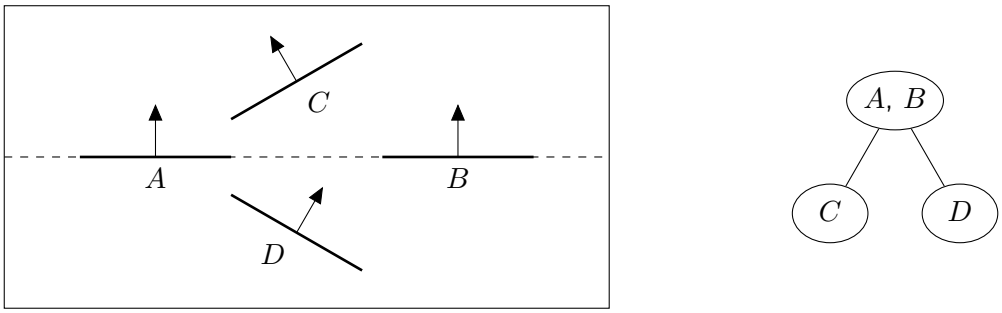


Figure 4.13: Coincident polygons are group into one node in the BSP tree.

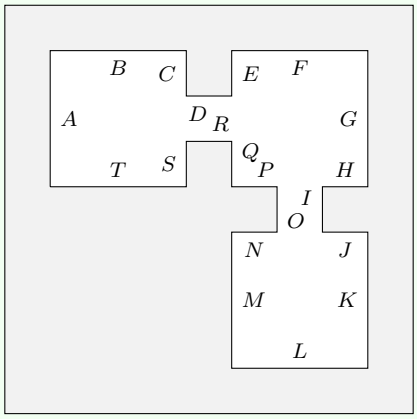
This procedure continues until all subspaces contain convex sets, i.e., all subspaces are atomic subspaces. The algorithm for the generation of a BSP tree is given below. Note that this is a recursive algorithm where the function calls itself.

Algorithm 4 BSP-tree generating algorithm

function BSPTREE(node)
  if node is a convex set **then**
    Exit function.
  else
    Choose a polygon from node (or coincident polygons) to be the root node.
    Split any polygons that are intersected by the hyperplane of the root node.
    List all of the polygons in the front subspace of the plane of the root node in the left child node.
    List all of the polygons in the back subspace of the root node in the right child node.
    Invoke BSPTREE(left child node).
    Invoke BSPTREE(right child node).
  end if
end function

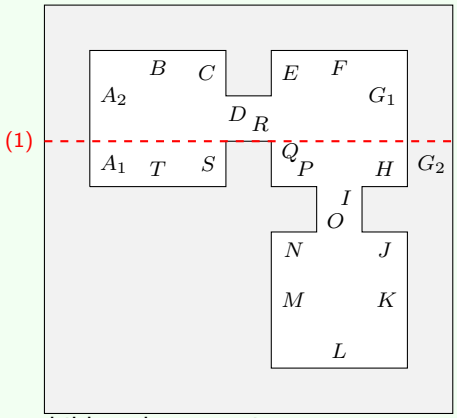
Example 4.3

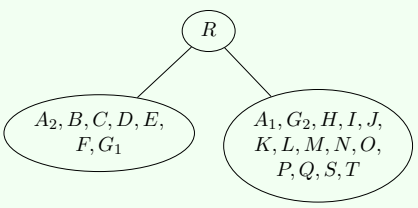
The diagram below shows a plan view of a virtual environment constructed using polygons labelled A to T with normal vectors pointing towards the interior. Construct a BSP tree for this virtual environment.



Solution:

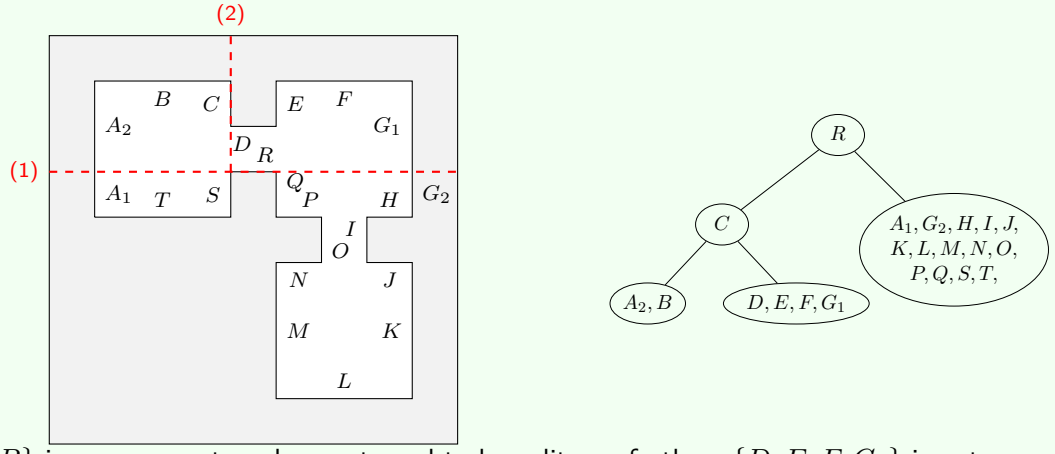
Let's choose the polygon R as the root node (note this is not a good choice to insert the hyperplane since it creates splits in the polygons, however, this is deliberate to show how splittings are dealt with). The hyperplane along polygon R splits polygons A and G so we label the split polygons as A_1, A_2, G_1 and G_2. Polygons $\{A_1, B, C, D, E, F, G_1\}$ are in the front subspace of R (since its normal vector is pointing upwards) so are listed in the left child node and polygons $\{A_1, G_2, H, I, J, K, L, M, N, O, P, Q, S, T\}$ are in the back subspace of T and are listed in the right child node.



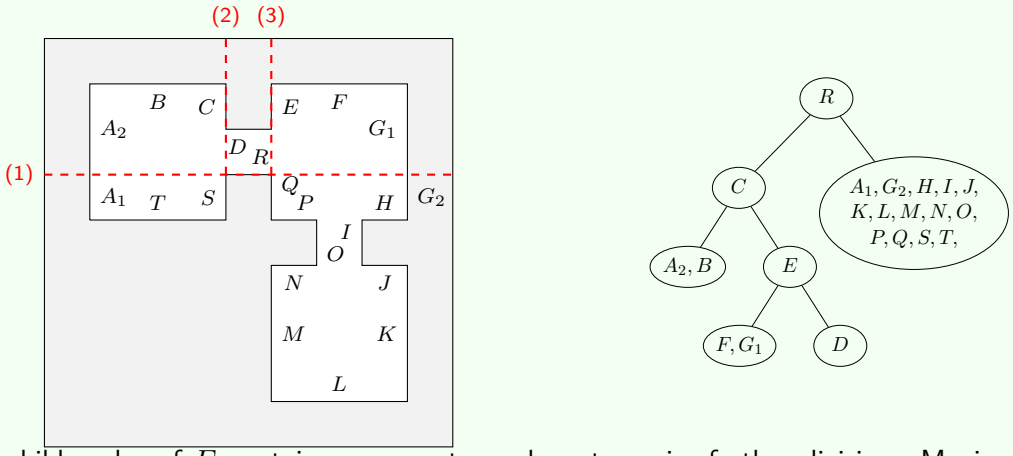


Neither child nodes contain convex sets so we need to subdivide each one. Focussing on the left

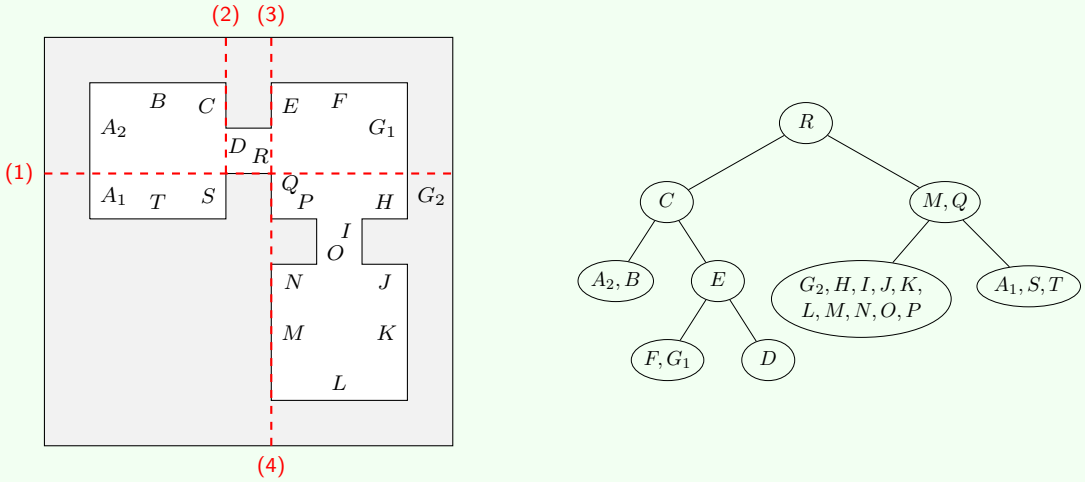
child node let's choose polygon C for our next node. Note that the hyperplane that is inserted along C extends only as for as the hyperplane along R. Polygons $\{A_2, B\}$ are in the front subspace of C so are listed in the left child node and polygons $\{D, E, F, G_1\}$ are in the back subspace of C and are listed in the right child node.



$\{A_2, B\}$ is a convex set so does not need to be split any further. $\{D, E, F, G_1\}$ is not a convex set so needs dividing further. Let's choose polygon E for the next node. Polygons $\{F, G_1\}$ are in the front subspace of E so are listed in the left child node and polygon D is in the back subspace of E and is listed in the right child node. The BSP tree is now

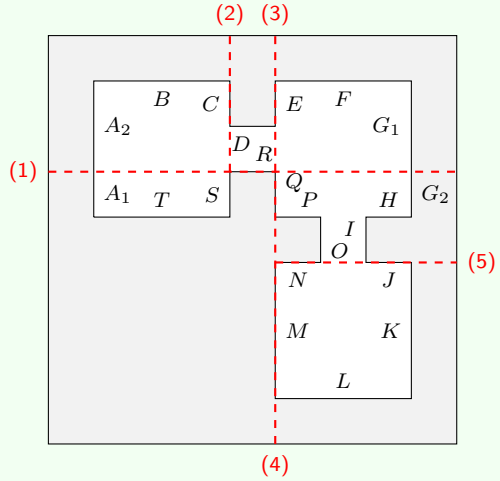


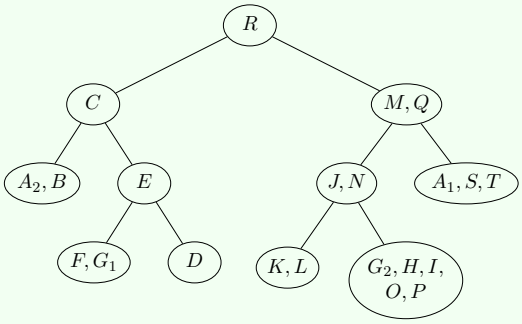
Both child nodes of E contain convex sets so do not require further division. Moving our focus to the right child node of R let's choose coincident polygons $\{M, Q\}$ for the next node. Polygons $\{H, I, J, K, L, N, O, P\}$ are in the front subspace of $\{M, Q\}$ so are listed in the left child node and polygons $\{A_1, S, T\}$ are in the back subspace of $\{M, Q\}$ so are listed in the right child node.



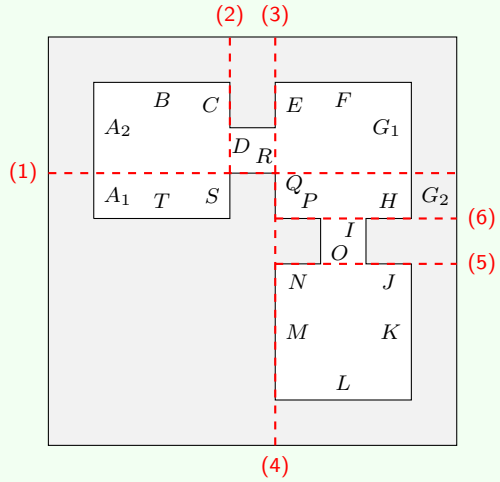
$\{A_1, S, T\}$ is a convex set so does not need further subdivision but $\{G_2, H, I, J, K, L, N, O, P\}$ is not a convex set so needs further subdivision. Let's choose coincident polygons $\{J, N\}$ for the next

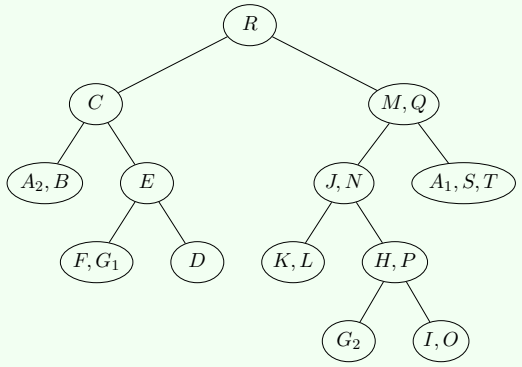
node. Polygons $\{K, L\}$ are in the front subspace of $\{J, N\}$ so are listed in the left child node and polygons $\{G_2, H, I, O, P\}$ are in the back subspace of $\{J, N\}$ so are listed in the right child node.





$\{K, L\}$ is a convex set so does not need further subdivision but $\{G_2, H, I, O, P\}$ is not a convex set so need further subdivision. Let's choose coincident polygons $\{H, P\}$ for the next node. Polygon G_2 is in the front subspace so is listed in the left child node and polygons $\{I, O\}$ are in the back subspace so are listed in the right child node.





4.4.2 Visibility ordering using BSP trees

So far we can construct a BSP tree by performing subdivision of a space and splitting the polygons where necessary and optimise it using various criteria. You might be forgiven in thinking how does BSP help when rendering three-dimensional scenes? If you consider what is actually happening with a BSP tree, we are determining which polygons are in front of the others. Therefore BSP trees lend themselves well to a technique similar to the painter's algorithm where we render the furthest polygons from the camera before the nearer ones. The problem with the painter's algorithm is that every time the camera position changes, the distances of each polygon have to be recalculated. Therefore this is not a suitable method to use in computer games where the camera position is commonly controlled by the player and changing often. BSP trees provide the visibility ordering for a scene depending on the camera position relative to the root node of the tree.

Consider the scene and the associated BSP tree shown in fig. 4.14 which is viewed from the viewpoint at $\mathbf{p}$. $\mathbf{p}$ is in the front subspace of the root node A so the polygons that are in the back subspace of A, in this case this is just polygon C, are further away from $\mathbf{p}$ than the polygons in the front subspace of A, which in this case is just polygon B. So the rendering order used by the painter's algorithm can be determined by drawing polygons in the opposite subspace to the viewing position first. Notice that polygon C is contained in the left sub-tree of A and polygon B is contained in the right sub-tree. So the rendering

order of polygons using the painter's algorithm can be determined using a BSP tree.

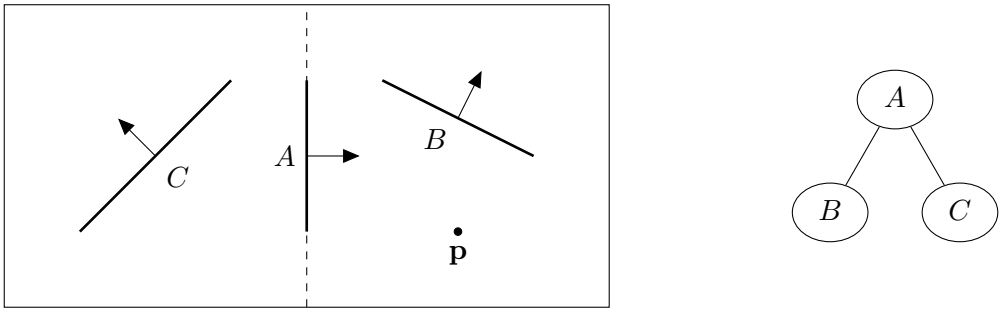


Figure 4.14: The world space is viewed from the viewpoint at $\mathbf{p}$.

The visibility ordering of polygons in a BSP tree are determined using an **in-order** tree walk presented in algorithm 5.

Algorithm 5 BSP-tree traversal

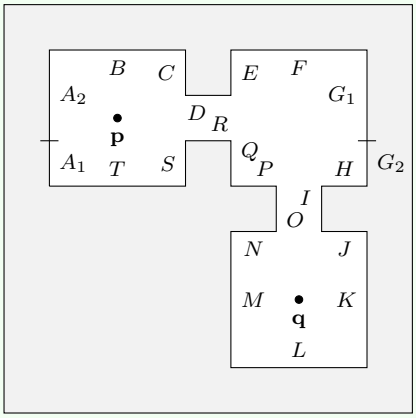
```
function BSPTreeTraversial(node)
    if node is a leaf node then
        Draw the polygons in node
    else if viewing position is in the front subspace of node then
        BSPtreetraversial(right child node)
        Draw the polygons in node
        BSPtreetraversial(left child node)
    else if viewing position is in the back subspace of node then
        BSPtreetraversial(left child node)
        Draw the polygons in node
        BSPtreetraversial(right child node)
    end if
end function
```

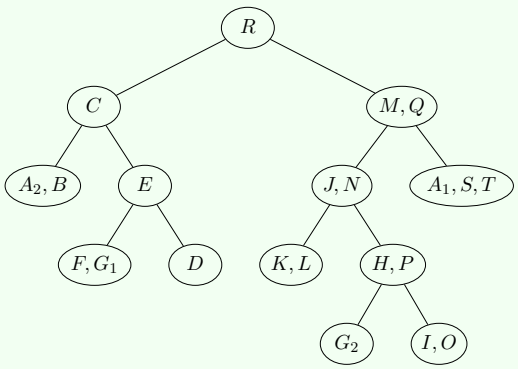
The advantage of using BSP trees is that once a tree has been generated for a virtual world it can be used to determine the rendering order for any position of the viewpoint.

Example 4.4

Using the BSP tree given below, determine the order that the polygons are drawn when the viewed from positions:

(i) $\mathbf{p}$; (ii) $\mathbf{q}$.





Solution:

(i) Starting at the root node R.

- $\mathbf{p}$ is in the front subspace of R so we move to the right child node $\{M, Q\}$.
- $\mathbf{p}$ is in the back subspace of $\{M, Q\}$ so we move to the left child node $\{J, N\}$.
- $\mathbf{p}$ is in the back subspace of $\{J, N\}$ so we move to the left child node $\{K, L\}$.
- $\{K, L\}$ is a leaf node so we draw the polygons $\underline{\{K, L\}}$ followed the parent node $\underline{\{J, N\}}$ and move to the right child node $\{H, P\}$.
- $\mathbf{p}$ is in the front subspace of $\{H, P\}$ so we move to the right child node $\{I, O\}$.
- $\{I, O\}$ is a leaf node so we draw the polygons $\underline{\{I, O\}}$ followed by the parent node $\underline{\{H, P\}}$ and move to the left child node G_2.
- G_2 is a left node so we draw the polygon $\underline{G_2}$ and move to the parent node $\{H, P\}$. $\{H, P\}$ has already been drawn so we move to its parent node $\{J, N\}$. $\{J, N\}$ has already been drawn so we move to its parent node $\{M, Q\}$ which has not yet been drawn so we draw polygons $\underline{\{M, Q\}}$ and move to the right child node $\{A_1, S, T\}$.
- $\{A_1, S, T\}$ is a leaf node so we draw polygons $\underline{\{A_1, S, T\}}$ and move to the parent node $\{M, Q\}$. $\{M, Q\}$ has already been drawn so we move to its parent node R which has not yet been drawn so we draw $\underline{R}$ and move to the left child node C.
- $\mathbf{p}$ is in the front subspace of C so we move to the right child node E.
- $\mathbf{p}$ is in the back subspace of E so we move to the left child node $\{F, G_1\}$.
- $\{F, G_1\}$ is a leaf node so we draw polygons $\underline{\{F, G_1\}}$ followed by the parent node $\underline{E}$ and move to the right child node D.
- D is a leaf node so we draw polygon $\underline{D}$ and move to the parent node E. E has already been drawn so we move to its parent node C which has not yet been drawn so we draw polygon $\underline{C}$ and move to the left child node $\{A_2, B\}$.
- $\{A_2, B\}$ is a leaf node so we draw polygons $\underline{\{A_2, B\}}$.
- All nodes have now been drawn so the order with which the polygons were draw is

$$\{K, L\}, \{J, N\}, \{I, O\}, \{H, P\}, \{M, Q\}, \{A_1, S, T\}, R, \{F, G_1\}, E, D, C, \{A_2, B\}.$$

(ii) Using the tree walk for position $\mathbf{q}$ the rendering order is

$$\{A_2, B\}, C, D, E, \{F, G_1\}, R, \{A_1, S, T\}, \{M, Q\}, G_2, \{H, P\}, \{I, O\}, \{J, N\}, \{K, L\}.$$

4.5 Lab exercises

Use the time in the lab to work on the coursework.

References

Carmack, J. and Romero, J. (1993). "Doom". In: *id software*.

Schumaker, R., Brand, B., Gilliland, M., and Sharp, W. (1969). *Study for applying computer generated images to visual simulation*. Tech. rep. General Electric Co.

Appendix A

Solutions to Lab Exercises

A.1 Vector geometry

These are the solutions to the lab exercises on page 20.

Solution 1.1.

```
a = [3, 4, 0];
norm(a)
```

Solution 1.2.

```
a = [3, 4, 0];
ahat = a / norm(a)

% Checking the magnitude
norm(ahat)
```

Solution 1.3.

```
a = [3, 4, 0];
b = [5, 12, 0];

% (i)
dot(a, b)

% (ii)
theta = acos(dot(a, b) / (norm(a) * norm(b)))

% (iii)
syms x
a = [1, 2, 3];
b = [4, x, 6];
x = solve(dot(a, b) == 0))
```

Solution 1.4.

```
% (i)
U = [1, 1 ; 0, 1];
W = [0, -1, ; 1, 1];
v = [3 ; 2];

rref(U)
rref(W)

% (ii)
v_U = U \ v
v_W = W \ v
```

```
% (iii)
UtoW = rref([W, U]);
UtoW(:, 1:2) = []

% (iv)
v_W = UtoW * v_U
```

Solution 1.5.

```
% (i)
p1 = [5, 4, 1];
p2 = [6, -2, 3];

r = p1 + 1 / 4 * (p2 - p1)

% (ii)
syms t
p = [2, 0, 1];
d = [2, 2, 1];
solve(r == p1 + t * d)
```

Solution 1.6.

```
% (i)
p1 = [1, 0, 3];
p2 = [2, 1, 1];
p3 = [0, 1, 3];

n = cross(p2 - p1, p3 - p1)
s = dot(n, p1)

% (ii)
p4 = [1, 2, 5]
dot(n, p4)
```

Solution 1.7.

```
p = [1, 0, 2];
v = [2, -1, 1];
q = [6, 4, 5];

t = dot(v, q - p) / dot(v, v)
r = p + t * v
d = norm(q - r)
```

Solution 1.8.

```
p = [1, 1, 3];
n = [1, -1, 1];
q = [4, -3, 2];

d = dot(q - p, n) / norm(n)
```

A.2 Translation, Rotation and Scaling Transformations

These are the solutions to the lab exercises on page 38.

Solution 2.1.

```
% (i)
% Define transformation
T = @(u) [2 * u(1) ; 3 * u(2)];

syms u1 u2 v1 v2 alpha
u = [u1, u2];
v = [v1, v2];

T(u + alpha * v)
T(u) + alpha * T(v)

% (ii)
A = [2, 0 ; 0, 3]

% (iii)
P = [1, 0, -1 ; 2, 3, 4];
A * P
```

Solution 2.2.

```
% (i)
T = [2, 0 ; 0, 3];
Tinv = inv(T)

% (ii)
syms x y
Tinv * [x ; y]
```

Solution 2.3.

```
% Define transformations
S = @(u) [2 * u(2) ; u(1)];
T = @(u) [3 * u(1) ; 2 * u(2)];

% (i)
S(T([3 ; 1]))

% (ii)
A = [0, 2 ; 1 , 0]
B = [3, 0 ; 0, 2]

% (iii)
A * B * [3 ; 1]

% (iv)
inv(B) * A * B * [1 ; 2]
```

Solution 2.4.

```
v1 = [1, 0, 1];
v2 = [3, 0, 1];
v3 = [2, 0, 3];
p = [3, 0, 1];

% Define translation function
T = @(p) [1, 0, 0, p(1) ; 0, 1, 0, p(2) ; 0, 0, 1, p(3) ; 0, 0, 0, 1];

% Co-ordinate matrix
P1 = [v1', v2', v3'];
P1(4, :) = ones(size(P1, 2), 1)
```

```
% Apply translation
P2 = T(p) * P1

% Plot triangles
clf
patch(P1(1, :), P1(3, :), 'w', 'EdgeColor', 'b', 'FaceAlpha', 0)
patch(P2(1, :), P2(3, :), 'w', 'EdgeColor', 'r', 'FaceAlpha', 0)
axis([0, 6, 0, 5])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 2.5.

```
v1 = [1, 0, 1];
v2 = [3, 0, 1];
v3 = [2, 0, 3];
scal = [3, 1, 2];

% Define scaling function
S = @(scal) [scal(1), 0, 0, 0 ; 0, scal(2), 0, 0 ; 0, 0, scal(3), 0 ; 0, 0, 0, 1];
% Co-ordinate matrix
P1 = [v1', v2', v3'];
P1(4, :) = ones(size(P1, 2), 1);

% Apply scaling
P2 = S(scal) * P1
% Plot triangles
clf
patch(P1(1, :), P1(3, :), 'w', 'EdgeColor', 'b', 'FaceAlpha', 0)
patch(P2(1, :), P2(3, :), 'w', 'EdgeColor', 'r', 'FaceAlpha', 0)
axis([0, 10, 0, 8])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 2.6.

```
v1 = [2, 0, 2];
v2 = [4, 0, 2];
v3 = [3, 0, 4];
scal = [2, 1, 2];

% Calculate centre of the triangle
c = 1 / 3 * (v1 + v2 + v3)

% Transformation matrix
A = inv(T(c)) * S(scal) * T(c)

% Co-ordinate matrix
P1 = [v1', v2', v3'];
P1(4, :) = ones(size(P1, 2), 1);

% Apply transformations
P2 = A * P1

% Plot triangles
clf
patch(P1(1, :), P1(3, :), 'w', 'EdgeColor', 'b', 'FaceAlpha', 0)
patch(P2(1, :), P2(3, :), 'w', 'EdgeColor', 'r', 'FaceAlpha', 0)
axis([0, 6, 0, 6])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 2.7.

```
v1 = [4, 0, 1];
v2 = [6, 0, 1];
v3 = [5, 0, 3];
theta = pi / 4;

% Define rotation transformation
Ry =@(t) [cos(t), 0, -sin(t), 0 ; 0, 1, 0, 0 ; sin(t), 0, cos(t), 0 ; 0, 0, 0, 1];
Ry(theta)

% Co-ordinate matrix
P1 = [v1', v2', v3'];
P1(4, :) = ones(size(P1, 2), 1);

% Apply rotation
P2 = Ry(theta) * P1

% Plot triangles
clf
patch(P1(1, :), P1(3, :), 'w', 'EdgeColor', 'b', 'FaceAlpha', 0)
patch(P2(1, :), P2(3, :), 'w', 'EdgeColor', 'r', 'FaceAlpha', 0)
axis([0, 6, 0, 6])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 2.8.

```
p = [10, 5, 50];
d = [2, -1, -3];
theta = pi / 6;

% Rotation around the z axis
v = sqrt(d(1) ^ 2 + d(2) ^ 2);
c = d(1) / v;
scal = d(2) / v;
Rz = [c, scal, 0, 0 ; -scal, c, 0, 0 ; 0, 0, 1, 0 ; 0, 0, 0, 1]

% Check rotation matrix
Rz * [d' ; 1]

% Rotation around the y axis
c = v / norm(d);
scal = - d(3) / norm(d);
Ry = [c, 0, -scal, 0 ; 0, 1, 0, 0 ; scal, 0, c, 0 ; 0, 0, 0, 1]

% Check rotation matrix
Ry * Rz * [d' ; 1]

% Rotation around x axis
Rx = @(t) [1, 0, 0, 0 ; 0, cos(t), sin(t), 0 ; 0, -sin(t), cos(t), 0 ; 0, 0, 0, 1];
Rx(theta)
```

A.3 Virtual Environments

These are the solutions to the lab exercises on page 58.

Solution 3.1.

```
clear

% Define house object
Vhouse = [-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2, 0, 0 ;
    -1, -1, 1, 1, -1, -1, 1, 1, -1, 1 ;
    0, 0, 0, 0, 1, 1, 1, 1, 2, 2 ;
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

Fhouse = [1, 4, 3, 2, 2 ;
    1, 2, 6, 9, 5 ;
    2, 3, 7, 6, 6, ;
    3, 4, 8, 10, 7 ;
    1, 5, 8, 4, 4 ;
    6, 7, 10, 9, 9 ;
    5, 9, 10, 8, 8 ];

% Plot object space
clf
patch('Vertices', Vhouse(1:3, :)', 'Faces', Fhouse, 'FaceAlpha', 0)
axis([-1, 1, -2, 2, 0, 2])
view(60, 30)
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$y$', 'FontSize', 16, 'Interpreter', 'latex')
zlabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 3.2.

```
theta = pi / 2;          % rotation angle for the houses
p1 = [3, 3.5, 0];        % position of the first house
p2 = [3, 1.5, 0];        % position of the second house
p3 = [1.5, 1.5, 0];      % position of the tower
scal = [0.5, 0.5, 1.5]; % scaling factors for the tower

% Define transformation functions
T = @(p) [1, 0, 0, p(1) ; 0, 1, 0, p(2) ; 0, 0, 1, p(3) ; 0, 0, 0, 1];
S = @(s) [s(1), 0, 0, 0 ; 0, s(2), 0, 0 ; 0, 0, s(3), 0 ; 0, 0, 0, 1];
Rz = @(t) [cos(t), sin(t), 0, 0 ; -sin(t), cos(t), 0, 0 ; 0, 0, 1, 0 ; 0, 0, 0, 1];

% Calculate world space co-ordinates for the first house
house1 = T(p1) * Rz(theta) * Vhouse

% Add house 1 to Fworld and Vworld
Fworld = Fhouse;
Vworld = house1;

% Calculate world space co-ordinates for the second house
house2 = T(p2) * Rz(theta) * Vhouse

% Add house2 to Fworld and Vworld
Fworld = [Fworld ; Fhouse + size(Vworld, 2)];
Vworld = [Vworld, house2];

% Calculate world space co-ordinates for the tower
Vtower = [-1, 1, 1, -1, -1, 1, 1, -1 ;
    -1, -1, 1, 1, -1, -1, 1, 1 ;
    0, 0, 0, 0, 2, 2, 2, 2 ;
    1, 1, 1, 1, 1, 1, 1, 1];
Ftower = [1, 4, 3, 2 ;
    1, 2, 6, 5 ;
    2, 3, 7, 6 ;
    3, 4, 8, 7 ;
```

```
    4, 1, 5, 8 ;
    5, 6, 7, 8 ];

tower = T(p3) * S(scal) * Vtower

% Add tower to Fworld and Vworld
Ftower(:, 5) = Ftower(:, 4);    % extend Ftower so that is has the same number of
    columns as Fworld
Fworld = [Fworld ; Ftower + size(Vworld, 2)]
Vworld = [Vworld, tower]

% Plot world space
clf
patch('Vertices', Vworld(1:3, :)', 'Faces', Fworld, 'FaceAlpha', 0)
axis([0, 5, 0, 5, 0, 5])
view(45, 55)
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$y$', 'FontSize', 16, 'Interpreter', 'latex')
zlabel('$z$', 'FontSize', 16, 'Interpreter', 'latex')
```

Solution 3.3.

```
p = [6, 5, 0.5];    % camera position
c = [2, 2, 1];      % centre of view
k = [0, 0, 1];      % up vector

% Calculate change of basis matrix
w = (p - c) / norm(p - c);
u = cross(k, w) / norm(cross(k, w));
v = cross(w, u);
R = [u, 0 ; v, 0 ; w, 0 ; 0, 0 ,0, 1]

% Check change of basis matrix
R * [(p - c)' ; 1]

% Calculate alignment matrix
A = [u, -dot(p, u) ; v, -dot(p, v) ; w, -dot(p, w) ; 0, 0, 0, 1]

% Align world space to camera space
Vcamera = A * Vworld

% Plot camera space
clf
patch('Vertices', Vcamera(1:2, :)', 'Faces', Fworld, 'FaceAlpha', 0)
axis([-3, 3, -3, 3])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$y$', 'FontSize', 16, 'Interpreter', 'latex')
box on
```

Solution 3.4.

```
near = -2;      % near viewing plane
far = -10;      % far viewing plane
fov = 1;        % field of view angle
aspect = 4/3;   % screen aspect ratio

% Calculate projection matrix
r = abs(near) * tan(fov / 2);
t = r / aspect;
P = [near/r, 0, 0, 0 ; 0, near/t, 0, 0 ; 0, 0, far/(far-near), -far*near/(far-near) ;
     0, 0, 1, 0]

% Project camera space co-ordinates onto the screen space
Vscreen = P * Vcamera
Vscreen = Vscreen ./ Vscreen(4, :)
```

```
% Plot screen space (with the border of the viewing frustum)
clf
patch('Vertices', Vscreen(1:2, :)', 'Faces', Fworld, 'FaceAlpha', 0)
patch([-1, 1, 1, -1], [-1, -1, 1, 1], 'b', 'FaceAlpha', 0, 'EdgeColor', 'b')
axis([-1.2 1.2, -1.2, 1.2])
xlabel('$x$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$y$', 'FontSize', 16, 'Interpreter', 'latex')
box on
```

Index