Computer Science
Aston University
Birmingham B4 7ET
http://www.cs.aston.ac.uk/

Computer Graphics

Dan Cornford

CS2150 September 20, 2004

1 Course Outline

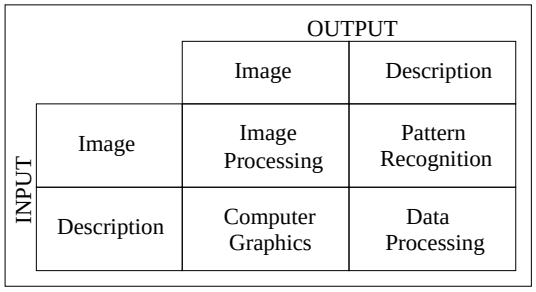


Figure 1.1: What is *computer graphics*?

This course is meant as an introduction to *computer graphics*, which covers a large body of work. The intention is to give a solid grounding in basic 2D *computer graphics* and introduce the concepts and some techniques required to implement 3D graphics. In order to achieve as complete an understanding as possible, we investigate the low level methods used in *computer graphics* while using OpenGL, a high level *computer graphics* Application Programmer Interface (API) to code up and use many of the methods discussed. Image processing is also briefly introduced, but in a more descriptive manner in the last part of the course.

Computer graphics sits at the interface between the **Image** and the **Description** (of that image) as shown in Figure 1.1. The most simple explanation of *computer graphics*, is the complete processes of taking a **Description** of an object or scene, and producing an **Image** of that object or scene using a computer. Image processing, on the other hand takes in an **Image** and alters it in some way to produce another **Image**.

Note that this module embraces the left hand column, the right hand column being essentially the concern of statistics and pattern recognition.

1.1 Course philosophy

The course is loosely based on Foley *et al.* (1993) and thus this is the recommended reading for the course, although most will probably find an OpenGL book more useful. The first part of the course will concentrate on the knowledge necessary to understand *computer graphics* using methods and algorithms for 2D graphics as examples. The second part develops the concepts into the 3D graphics domain with an *introduction* to some of the methods used to generate realistic representations of real world objects. In the third part of the module the hardware background and some related algorithms are reviewed. In the final part of the course a *brief* introduction to image processing and associated techniques is given. This should give you both a solid foundation in *computer graphics* and an appreciation of the breadth and complexity of the modern subject.

The lab classes introduce some standard programming approaches to *computer graphics*, using OpenGL. OpenGL is probably the most widely used cross platform graphics API. Although Direct 3D (written by Microsoft) is also very popular on PC systems, OpenGL has the advantage of being supported on almost all platforms, with bindings to almost all of the commonly used languages. Thus the lab classes support the lecturing material, often implementing what is covered in the lectures. We will use C in the labs, and thus the first few labs will involve teaching you basic C. I will not spend too much time on this, however, since this is not a programming module. Note that the syntax of C is almost identical to Java, so you are not being made to learn a completely new language.

1.2 Course composition

The course consists of the following main sections:

- why study *computer graphics*;
- 2D geometrical transformations and clipping;
- 3D *computer graphics*;
- curves and surfaces.
- visible surface problem, illumination and shading;
- conceptual models for *computer graphics*;
- *computer graphics* hardware issues;
- raster algorithms for 2D *computer graphics*;
- image formats and compression;
- image processing.

1.3 Recommended reading

These lecture notes contain enough detail to pass the CS215 exam, however to get a good grade you will need to attend the lectures and practicals **and** do some independent reading.

There are two good books which cover almost all the material in the course: Foley *et al.* (1993) and Hearn and Baker (1994). There is really very little to choose between them - of the two Hearn and Baker (1994) has a *little* more mathematics - although really they are very similar in content. Neither book covers OpenGL which is used in the lab classes, so an alternative might be Hill (2000), however this book is rather difficult to read and uses rather a lot of C++– check it out before you buy it. If you are really interested in *computer graphics* and you have funds available then Foley *et al.* (1996) is a very complete volume covering all aspects of *computer graphics*, and will serve as a useful future reference. If you are more interested in the programming side and want to learn more about OpenGL, then Woo *et al.* (1999) is a very well written book, if a little expensive. A cheaper alternative is Angel (2002) which is cheaper, and covers quite a bit of OpenGL, but no graphics theory.

The library has copies of all the above books, and many more (check out the class-marks 006.6 in the library). Short loan should hold at least one copy of the key books. The library also takes a number of journals which cover computer graphics matters, on the first floor.

References

Angel, E. 2002. *OpenGL: A primer*. London: Addison-Wesley.

Foley, J., A. van Dam, S. K. Feiner, and J. F. Hughes 1996. *Computer Graphics: Principles and Practice* (Second Edition in C ed.). Reading, Massachusetts: Addison-Wesley.

Foley, J., A. van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips 1993. *Introduction to Computer Graphics*. Reading, Massachusetts: Addison-Wesley.

Hearn, D. and P. M. Baker 1994. *Computer Graphics* (Second Edition ed.). London: Prentice Hall.

Hill, F. S. 2000. *Computer Graphics using OpenGL* (Second Edition ed.). London: Prentice Hall.

Woo, M., J. Neider, T. Davis, and D. Shreiner 1999. *OpenGL Programming Guide* (Third Edition ed.). Reading, Massachusetts: Addison-Wesley.

Contents

2 Why Study Computer Graphics?

Computer Graphics is an essential part of the Computer Science curriculum. It is the primary method of presentation of information from computer to human. As such it is a core component of any computer system, with *computer graphics* playing a major role in:

- ➢ Entertainment - computer animation;
- ➢ User interfaces;
- ➢ Interactive visualisation - business and science;
- ➢ Cartography;
- ➢ Medicine;
- ➢ Computer aided design;
- ➢ Multimedia systems;
- ➢ Computer games;
- ➢ Image processing.

2.1 A *very* brief history

In the early days of *computer graphics* developments were strongly constrained by hardware developments. In 1950 the Whirlwind Computer, at the Massachusetts Institute of Technology had a Cathode Ray Tube (CRT) display for output. By the mid sixties Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) systems where being used, for instance in the motor industry, and graphics played a large role in these systems. However these applications were still rather batch mode based.

At this time most of the display devices were vector based displays. CRT displays work by firing an electron beam at a phosphor coating, and in vector systems the lines are created by firing the electron beam along the path of the line. In order to maintain the phosphorescence the display had to refresh the screen 30 times per second (that is a refresh rate of $30Hz$).

In the early 1970's the development of television technology meant that cheap raster displays (based on arrays of pixels) became available. Raster displays store the display primitives in a refresh buffer (see later) in terms of the primitives' component pixels. At the same time colour systems, with three beams for the red, green and blue primary colours became more common. Some early home computers, such as the ZX Spectrum, had display converters which allowed output through television sets.

In the early 1980's the advent of the personal computer, with built in raster display capabilities (i.e. the Apple Mac and the IBM PC) led to the widespread adoption of bitmap graphics, and interactive graphics. The explosion of computer use in all areas of society meant there were potentially millions of users, many of whom had limited computer skills. The development of Graphical User Interfaces (GUIs) allowed novice users to access a large variety of applications.

The computer screen became the electronic 'desktop' and window managers handled multiple applications. Direct manipulation of objects (via 'point and click') allowed intuitive control and released the user from lengthy command line instructions. This has been parallelled by the development of input technology, from the light pen to the mouse and beyond.

As the hardware has developed, software has also changed. Gone are the low level proprietary systems of the early years in computing. The first graphics specification to receive an official standard (in 1985) was the Graphics Kernel System (GKS) which provided a high level 2D graphics standard. This was complemented (in 1988) by GKS-3D, which unsurprisingly adds 3D capabilities to GKS.

Also in 1988 the Programmer's Hierarchical Interactive Graphics System (PHIGS - pronounced *figs*), which allows a nested hierarchical grouping of 3D sub-primitives called structures, was recognised. The primitives and structures in PHIGS could be geometrically transformed and clipped, to allow dynamic graphics. In 1992 an extension PHIGS PLUS included pseudo-realistic rendering of graphics.

There are now several standards, such as PHIGS, OpenGL (Silicon Graphics), X Windows System, PostScript (Adobe) and Direct 3D (Microsoft). Many of the functions in these graphics specifications are supported by

hardware on modern graphics cards. These graphics libraries are used at almost all levels and in all branches of computer graphics and free the programmer from the job of re-inventing the wheel (i.e. implementing low level graphics functions) while freeing the Central Processor Unit (CPU) from performing a lot of the processing since this can be implemented in hardware on a chip on the graphics display hardware.

3 2D Geometrical Transformations

This section is perhaps the most mathematical section of the course, so maybe you are wondering why it comes first – get the pain over with quickly. However the maths is relatively simple and it is necessary to understand this in order to progress, since these transformations form the basis of the mapping from the application model to the graphics system.

We begin with a brief mathematical review of vector and matrix algebra. This is not scary, algebra simply means we are looking at arithmetic, but generalising it using variables, just as we do in computer programs. It is important that you understand this section, and unfortunately with things mathematical there is generally only one way to do this – that is through practice.

3.1 Vectors

A vector is an n-tuple of numbers, for instance in 2D a point can be written as a 2 component vector. Vectors are denoted by lower case bold letters[1], for instance:

$$\boldsymbol{r} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} ,$$

is a vector in 2D (also represents a point). Note vectors are usually represented as columns.

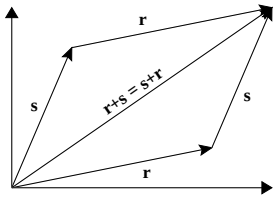


Figure 3.1: Vector addition.

The rules for the addition;

$$\boldsymbol{r} + \boldsymbol{s} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} + \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} r_1 + s_1 \\ r_2 + s_2 \\ r_3 + s_3 \end{bmatrix} ,$$

and scalar multiplication;

$$a\boldsymbol{r} = a \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} ar_1 \\ ar_2 \\ ar_3 \end{bmatrix} ,$$

of vectors are very simple. It is easy to see that vector addition is commutative (that is the order is not important) with $\boldsymbol{r} + \boldsymbol{s} = \boldsymbol{s} + \boldsymbol{r}$. This is illustrated in Figure 3.1.

Given two vectors $\boldsymbol{r}$ and $\boldsymbol{s}$ we define their **dot product** (also sometimes called their **inner product**) to be:

$$\boldsymbol{r} \cdot \boldsymbol{s} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} \cdot \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = r_1 s_1 + r_2 s_2 + r_3 s_3 .$$

The **length** of a vector $\boldsymbol{r}$, denoted $\|\boldsymbol{r}\|$, is given by the square root of the dot product of the vector with itself: $\sqrt{\boldsymbol{r} \cdot \boldsymbol{r}}$. The dot product can be used to generate unit length vectors using $\boldsymbol{r}/\|\boldsymbol{r}\|$ and compute the angle between two vectors:

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{r} \cdot \boldsymbol{s}}{\|\boldsymbol{r}\|\|\boldsymbol{s}\|}\right) .$$

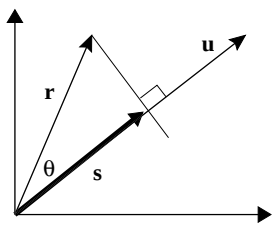


Figure 3.2: Vector projection.

[1] There is no general convention for vector notation, Foley *et al.* (1993) use italicised lower case letters, while Hearn and Baker (1994) use bold upper case letters. We adhere to the more standard mathematical notation.

We can also **project** one vector, $\boldsymbol{r}$, onto another unit vector, $\boldsymbol{u}$. This is shown in Figure 3.2, the length of $\boldsymbol{s}$ will be given by:

$$\|\boldsymbol{s}\| = \|\boldsymbol{r}\| \cos(\theta) = \|\boldsymbol{r}\| \left(\frac{\boldsymbol{r} \cdot \boldsymbol{u}}{\|\boldsymbol{r}\|\|\boldsymbol{u}\|} \right) = \boldsymbol{r} \cdot \boldsymbol{u} \ .$$

This means the dot product can be interpreted as the length of the projection of $\boldsymbol{r}$ onto $\boldsymbol{u}$ whenever $\boldsymbol{u}$ is a unit vector.

3.2 Matrices

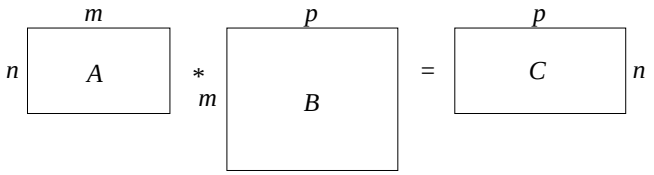


Figure 3.3: Matrix multiplication.

A matrix is a rectangular array of numbers (which can be viewed as a row of vectors) which is extensively used in computer graphics since it gives us very compact notations. A general matrix will be represented by an upper case letter:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \ .$$

The element on the ith row and jth column is denoted by a_{ij}. Note that we start indexing at 1, whereas C indexes arrays from 0 - beware! A matrix is said to be of dimension n by m (written $n \times m$) if it has n rows and m columns. Matrix multiplication is more complex. Given two matrices, A and B if we want to multiply B by A (that is form AB) then if A is $(n \times m)$, B must be $(m \times p)$. This produces a result, $C = AB$, which is $(n \times p)$, with elements $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$, that is the i, jth element of C is the ith row of A dot producted with the jth column of B. Note that matrix multiplication is **not** commutative, indeed in this case we cannot multiply BA, since the sizes are wrong.

```
/* Define the structure to hold the elements of a 3 by 3 matrices. */
 typedef struct Matrix3struct { double el[3][3]} Matrix3;
/* Declare the function  to multiply C = AB */
 Matrix3 *MatMult3(a,b,c) /* We return a pointer to a matrix. */
 Matrix3 *a, *b, *c;       /* These are pointers too. */
{
    int i,j,k;
   /* Do the loop to multiply the matrices - recall the zero indexing */
    for (i = 0; i < 3; i++) {
       for (j = 0; j < 3; j++) {
         /* Recall we use the -> operator to access an element of a  */
         /* structure when we have only a pointer to that structure. */
         c->el[i][j] = 0.0;      /* Make sure C is initialised. */
          for (k = 0; k < 3; k++) {
            c->el[i][j] += a->el[i][k]*b->el[k][j];
         }
       }
   }
 return (c);
}
```

Listing 1: C code to multiply two 3×3 matrices.

A simple C program to perform matrix multiplication of two 3×3 matrices is shown in Listing 1. Matrix multiplication distributes over addition, that is $A(B + C) = AB + AC$, and there is an identity matrix for multiplication, denoted I, which is square and has ones on the diagonal with zeros everywhere else.

The transpose of a matrix, A, which is either denoted A^T or A' is obtained by swapping the rows and columns of the matrix. Thus:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \quad \Rightarrow \quad A' = \begin{bmatrix} a_{1,1} & a_{2,1} \\ a_{1,2} & a_{2,2} \\ a_{1,3} & a_{2,3} \end{bmatrix} \ .$$

If we consider a $n \times 1$ matrix (that is a column vector, $\boldsymbol{s}$) then it transpose $\boldsymbol{s}'$ is a $1 \times n$ matrix (which we would call a row vector).

There are many other vector and matrix operators that have not been introduced here, however we shall deal with these as they are required.

3.3 2D transformations

This section considers some of the transformations that are applied to 2D graphics primitives and objects. This section uses many of the results that were shown above for vectors and matrices.

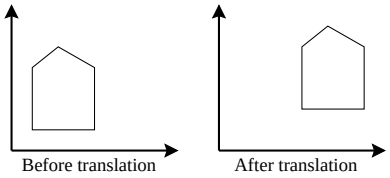


Figure 3.4: 2D translation.

Translations are simple movements or offsets from the existing position of the object. If we have a point at $\boldsymbol{r}$ (a 2 by 1 column vector) and we translate it by an amount $\boldsymbol{t} = (t_1, t_2)'$ (another 2 by 1 column vector) then the new location will be $\boldsymbol{r}^* = \boldsymbol{r} + \boldsymbol{t}$. This is shown in Figure 3.4 for a rather more complex object. This is translated by applying the same translation to all the points that define the object.

Scalings are simple stretchings of the object, generally about the origin. Given a point $\boldsymbol{r}$ and a scaling matrix S, where:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} ,$$

where s_x is the x-axis scaling and s_y is the y-axis scaling the location of the new point can be written $\boldsymbol{r}^* = S\boldsymbol{r}$. If $s_x = s_y = s$ the scaling is said to be **uniform** and $\boldsymbol{r}^* = s\boldsymbol{r}$, otherwise the scaling is called **differential**. An example is shown in Figure 3.5.

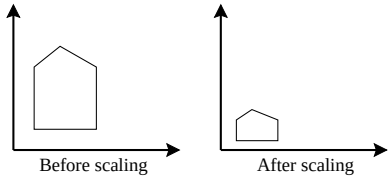


Figure 3.5: 2D scaling.

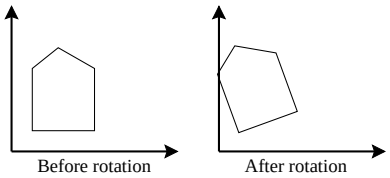


Figure 3.6: 2D rotation.

Rotations about the origin by an angle θ are defined by the rotation matrix R which is given by:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} ,$$

and the rotated point, $\boldsymbol{r}^* = R\boldsymbol{r}$. Note that positive θ implies an anti-clockwise rotation. It is a simple exercise to show, using simple trigonometry, that R is indeed a matrix which rotates points by θ.

3.4 Homogeneous coordinates

Representing 2D coordinates in terms of vectors with 2 components turns out to be rather awkward when it come to the sort of manipulation that needs to be carried out for *computer graphics*. **Homogeneous coordinates** allow us to treat all transformations in the same way, as matrix multiplications. The consequence is that our 2-vectors become extended to 3-vectors, with a resulting increase in storage and processing.

Homogeneous coordinates mean that we represent a point (x, y) by the extended triple (x, y, w). In general w should be non-zero. The normalised homogeneous coordinates are given by $(x/w, y/w, 1)$ where $(x/w, y/w)$ are the Cartesian coordinates of the point. Note in homogeneous coordinates (x, y, w) is the same as $(x/w, y/w, 1)$ as is (ax, ay, aw) where a can be any real number. Points with $w = 0$ are called points at infinity, and are not frequently used.

Vector triples usually represent points in 3D space, however here we are using them to represent points in 2D space, so what is going on. Well we are using a bit of mathematical trickery to make life easy for ourselves (really!). If you like then you can think of 2D space corresponding to plane $w = 1$.

Now in homogeneous coordinates the transformations can be given as: translation:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = T\boldsymbol{r} \ ;$$

scaling:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = S\boldsymbol{r} \ ;$$

and rotation:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = R\boldsymbol{r} \ .$$

This is a very useful description of arbitrary translations. There are several different type of transformations. **Rigid body transformations** preserve length and angles (e.g. translation or rotation), while **affine transformations** preserve parallelism in lines (e.g. translation, rotation, scaling and shearing). A shear transformation is given by:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H\boldsymbol{r} \ ,$$

where h_x and h_y represent the amount of shear along the x and y axes respectively.

3.5 Composition of transformations

One of the big advantages of homogeneous coordinates is that transformations can be very easily combined. All that is required is multiplication of the transformation matrices. This makes otherwise complex transformations very easy to compute. For instance if we wanted to rotate an object about some point, $\boldsymbol{p}$, on (or in) that object, this can easily be achieved by:

- ➢ translate object by $-\boldsymbol{p}$,
- ➢ rotate object by angle θ,
- ➢ translate object by $\boldsymbol{p}$.

This can be written as:

$$\begin{aligned} T(\boldsymbol{p})R(\theta)T(-\boldsymbol{p}) &= \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & p_x(1-\cos\theta) + p_y\sin\theta \\ \sin\theta & \cos\theta & p_y(1-\cos\theta) - p_x\sin\theta \\ 0 & 0 & 1 \end{bmatrix} . \end{aligned}$$

Note that matrix multiplication proceeds from right to left. Thus to apply the composite transformation we only need apply the matrix on the right hand side to all points in the object. For those interested, MATLAB provides an excellent platform for investigating these sort of transformations, since its natural matrix format makes things very easy to code.

3.6 Window to viewport transformation

In general the objects and primitives represented in the application model will be stored in **world coordinates**, that is their size, shape, position, etc. will be given in terms of logical units for whatever the object represent (e.g. *mm*, *cm*, *m*, *km* or light years). Thus to display the appropriate images on the screen (or other device) it is necessary to map from **world coordinates** to **screen** or **device coordinates**. This transformation is

known as the **window to viewport transformation**, the mapping from the *world coordinate* **window**[2] to the **viewport** (which is given in screen coordinates).

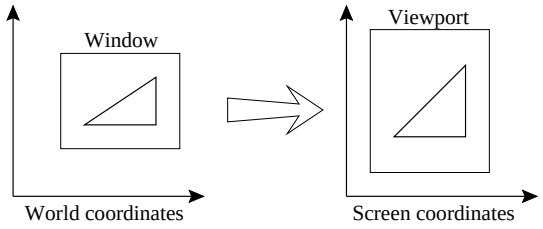


Figure 3.7: The window to viewport transformation.

In general the **window to viewport transformation** will involve a scaling and translation as in Figure 3.7 where the scaling in non-uniform (the vertical axis has been stretched). Non uniform scalings result from the world coordinate window and viewport having different aspect ratios. In Figure 3.7 the screen window (that is the viewport) covers only part of the screen. The transformation is generally achieved by a translation in world coordinates, a scaling to viewport coordinates and another translation in viewport coordinates, which are generally composed to give a single transformation matrix. Often the clipping of visible elements will be carried out at the same time as the transformation is applied. Typically the region will be clipped in world coordinates and then transformed.

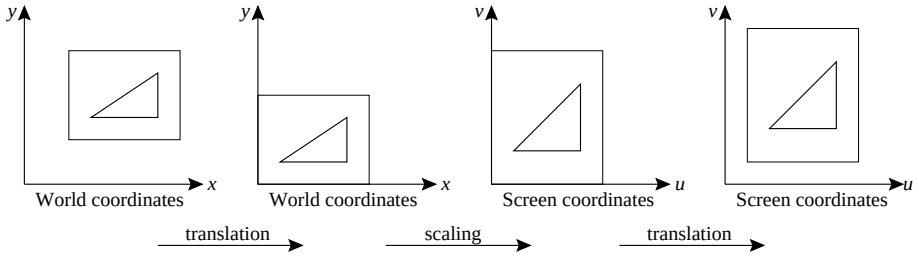


Figure 3.8: The procedure used to transform from world coordinate window to viewport.

The required transformations are shown in Figure 3.8. If the world coordinate window has dimensions $(x_{\min}, y_{\min}) \rightarrow (x_{\max}, y_{\max})$ and the viewport (or screen coordinate window) has dimensions $(u_{\min}, v_{\min}) \rightarrow (u_{\max}, v_{\max})$, then the transformation will be given by: a translation;

$$T_x = \begin{bmatrix} 1 & 0 & -x_{\min} \\ 0 & 1 & -y_{\min} \\ 0 & 0 & 1 \end{bmatrix} ;$$

a scaling:

$$S_{xu} = \begin{bmatrix} \frac{u_{\max}-u_{\min}}{x_{\max}-x_{\min}} & 0 & 0 \\ 0 & \frac{v_{\max}-v_{\min}}{y_{\max}-y_{\min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} ;$$

and finally another translation;

$$T_u = \begin{bmatrix} 1 & 0 & u_{\min} \\ 0 & 1 & v_{\min} \\ 0 & 0 & 1 \end{bmatrix} .$$

These can be combined to yield the transformation matrix $M_{xu} = T_u S_{xu} T_x =$

$$M_{xu} = \begin{bmatrix} \frac{u_{\max}-u_{\min}}{x_{\max}-x_{\min}} & 0 & -x_{\min}\frac{u_{\max}-u_{\min}}{x_{\max}-x_{\min}} + u_{\min} \\ 0 & \frac{v_{\max}-v_{\min}}{y_{\max}-y_{\min}} & -y_{\min}\frac{v_{\max}-v_{\min}}{y_{\max}-y_{\min}} + v_{\min} \\ 0 & 0 & 1 \end{bmatrix} .$$

[2]This is not the usual meaning of window as in window manager, it merely refers to the fact that we may only be taking a small window on the world, not necessarily considering infinite space.

3.6.1 Efficiency

The composition of transformations involving translation, scaling and rotation lead to transformation matrices M of the general form:

$$M = \begin{bmatrix} r_{1,1} & r_{1,2} & t_x \\ r_{2,1} & r_{2,2} & t_y \\ 0 & 0 & 1 \end{bmatrix} .$$

To achieve such a transformation efficiently it should be recognised that a straight matrix multiplication of the matrix times a column vector, $\boldsymbol{r} = [x, y, 1]'$, is not efficient because M is sparse. This matrix multiplication would take nine multiplies and six adds. Using the fixed structure in the final row of the matrix, the actual transformation can be written:

$$x^* = r_{1,1}x + r_{1,2}y + t_x ,$$
$$y^* = r_{2,2}y + r_{2,1}x + t_y ,$$

which uses four multiplies and four adds. Since this transformation must be applied to several hundred, possibly millions, of points for each image, this can mean a big saving in computational time. However some hardware matrix multipliers (such as are present on many graphics systems) have parallel adders and multipliers which effectively removes this concern. In any case the use of homogeneous coordinates still makes sense because of the simplicity it brings to composing transformations.

3.6.2 Inverse transformations

If a general transformation (which may be composite) is given by the 3×3 matrix M, then the inverse transformation, which maps the new image back to the original, untransformed one is given by M^{-1}, that is M inverse. Since M is a matrix representing transformations this inverse matrix will exist (because a general transformation matrix is non-singular). The matrix inverse is defined so that $MM^{-1} = I$ where I is the 3×3 identity matrix,

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

For most transformations is is quite obvious, from first principles what the inverse transformations are, for instance the translation matrix has inverse:

$$T^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} ,$$

while the scaling matrix has inverse:

$$S^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

If you doubt this then you can easily verify the inverse transformation matrix in MATLAB. In general MATLAB is a very useful tool for exploring the use 2D (and 3D as we shall see) transformations, since it was designed to be used for matrices and has extensive platform independent graphics capabilities, from drawing primitives and rendering and illumination models. If you have time I would advise you 'play around' with MATLAB and reproduce some of the methods discussed in the notes, although this isn't assessed.

4 Viewing in 3D

Most of the objects that will be stored in the application model will naturally exist in either 2D (plans, cross-sections, simple graphs) or more likely 3D (real world objects, more complex graphs) space. Since the viewport is currently a 2D representation of whatever is in the application model, 3D coordinate systems call for a little extra work, defining the 2D projection of the 3D objects. First we look at defining and manipulating objects in 3D.

4.1 Geometric modelling

Modelling is a very familiar concept to computer scientists. We use models to represent objects, processes and abstract ideas in a way which makes understanding more simple. Graphics have a use in all modelling exercises, particularly in displaying results and conclusions. More directly, computer graphics might be concerned with different types of models such as:

- ➢ **organisation models** - hierarchies, flowcharts; directed graph representations.
- ➢ **quantitative models** - graphs, maps.
- ➢ **geometric models** - engineering and architectural structures, chemicals.

In this section we confine ourselves to geometric models. As the name suggests geometric models describe the geometry of the objects which they represent. This includes:

- ➢ spatial layout and shape of the component parts of the object (geometry),
- ➢ connectivity of the component parts (topology),
- ➢ attributes (which affect the appearance),
- ➢ attributes (which pertain to the object but do not affect appearance).

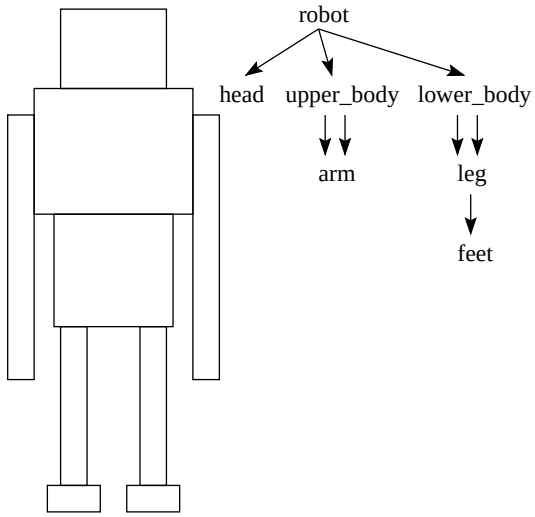


Figure 4.1: A hierarchical geometric robot model and its **directed acyclic graph**.

We commonly use hierarchical constructs to help store geometric models, because the objects we want to represent can be easily stored as a series of connected common components. The hierarchy will start with base components and combine these to form successive levels of objects. In general each object will appear in the hierarchy more than once, and the hierarchy can be symbolised using a **Directed Acyclic Graph** (DAG). A 2D example of a robot and its DAG can be seen in Figure 4.1. In some DAGs the arrows are omitted, since the ordering is given by the position on the page. If each object appeared only once in the hierarchy the resulting data structure would be a **tree**. The DAG may give details of the topology such as specifying where the objects are attached (or equivalently about which axes the objects can rotate or translate).

Object hierarchies are useful because:

- ➢ complex models can be constructed in a simple modular fashion,
- ➢ stored efficiently and
- ➢ updated simply (the updating of one level in the hierarchy automatically updates elements below it).

Figure 4.2 shows a zoomed view on the definition of the application model. We can see that the application program is composed of several subsystems which have variable degrees of access to the application model. This figure shows the logical organisation of the modules, however the actual code may be very differently organised. In many industrial application the 80/20 rule is generally true: 80% of the program deals with modelling objects (the database) and only 20% deals with producing the pictures.

This view of the graphics process leads to retained mode graphics packages such as PHIGS. PHIGS is a **retained mode** graphics package, that is it keeps a record of all the primitives in the application model which allows automatic updating of the screen and simple editing of the primitives. Basic OpenGL, on the other hand, is

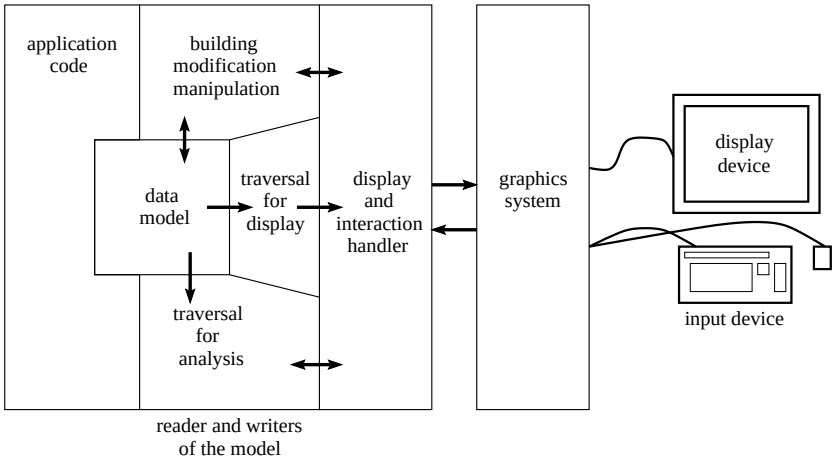


Figure 4.2: A closer view of the application model in the conceptual model.

an **immediate mode** graphics package, where only the effects on the screen are stored, not the generating primitives.

PHIGS stores its information in the Central Structural Storage (CSS), a database for graphical structures. A **structure** in PHIGS is a sequence of **elements**, which are themselves primitives, appearance attributes, transformation matrices or subordinate structures. The structures define coherent geometric entities. Thus PHIGS is essentially a device independent hierarchical display list package. The CSS duplicates information in the application model, which allows fast display traversal, which is the main benefit of using a separate CSS (especially when a co-processor handles the rendering). The CSS also allows automatic pick correlation and is useful for editing the structure contained within, to produce dynamic motion (animation). This can be done by applying time varying dynamics to position, rotate or scale sub-objects within the parent objects. For instance if our parent object were a 3D robot we could use a rotation applied to a substructure (e.g. the arm) to represent joints, and dynamically rotate the arm by editing a single rotation matrix.

Note that the CSS is not necessary (the application program could implement the display traversal and pick correlation - albeit at great expense) or sufficient (most real application programs will need additional data stored in the application model) for most graphical applications, it is simply an efficient way of implementing a graphics library. Thus not all graphical libraries use the retained mode principle because the overheads of duplicating the application model data and keeping the two consistent. If things change very rapidly between successive images then there will be very little benefit in using a retained mode graphics package. Thus OpenGL offers an immediate mode for such situations, and retained mode like structures (called display lists) where a CSS is more appropriate. This is covered in more detail in the lab classes.

4.2 3D coordinate systems

Much of what was discussed in the section on 2D graphics also applies here. The homogeneous coordinate system (which this time means points (or vectors) are represented by four numbers) is still very useful for manipulating transformations, but we have to take care with our coordinate system.

Axis of rotation	Direction of positive rotation
x	$y \rightarrow z$
y	$z \rightarrow x$
z	$x \rightarrow y$

Table 1: Tabular definition of the right handed coordinate system.

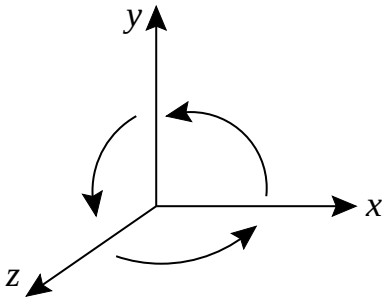


Figure 4.3: Right handed coordinate system.

In this text we assume a right handed coordinate system, shown in Figure 4.3, where the z axis comes out of the page. This is simply a convention, and is maybe a little counter intuitive since it means that objects further away from you have a smaller z value. Beware, the z axis is taken to mean the depth (that is the depth in terms of the negative distance from the viewer) not the height as is taken is usual scientific applications. The right hand coordinate system means that a 90° anti-clockwise rotation about a given axis rotates one positive axis onto another:

4.3 Transformations in 3D

In 3D coordinates a general point $\boldsymbol{r} = [x, y, z]'$ will be represented in homogeneous coordinates by $\boldsymbol{r} = [x/w, y, w, z/w, 1]'$, just like the 2D case. Just like the 2D case we can define transformation matrices, which will this time be 4×4 matrices. The transformations are: translation:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = T\boldsymbol{r} \ ;$$

scaling:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = S\boldsymbol{r} \ .$$

Rotation is a little more tricky, since we need to be careful what axis we are rotating about. A rotation about the z axis (which is basically the 2D rotation of earlier is given by:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = R_z\boldsymbol{r} \ ,$$

a rotation about the x axis is given by:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = R_x\boldsymbol{r} \ ,$$

and a rotation about the y axis is given by:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = R_y\boldsymbol{r} \ .$$

All of the 3×3 upper left sub-matrices of the rotation matrices are special orthogonal which means that they preserve distances and angles. Any arbitrary sequence of rotation matrices is also special orthogonal. What is more all the above transformation matrices have inverses. For rotations, the 3×3 upper left sub-matrices are special orthogonal and their inverse is given by their transpose, $R^{-1} = R'$.

We can again combine the transformations to give a general transformation matrix $M =$

$$\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R^* & \boldsymbol{t}^* \\ \boldsymbol{0} & 1 \end{bmatrix} \ ,$$

where R^* is the 3×3 matrix that represents the combined effect of rotations and scalings and $\boldsymbol{t}^*$ is the 3×1 column vector representing the combined effect of all translations and $\mathbf{0}$ is a 1×3 row vector of zeros. Rather like in 2D efficiency can be improved by recognising that $\boldsymbol{r}^* = M\boldsymbol{r}$ can be computed using:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = R^* \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \boldsymbol{t}^* \ .$$

A shear in (x, y) is given by:

$$\boldsymbol{r}^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = H_{x,y}\boldsymbol{r} \ ,$$

where h_x and h_y represent the amount of shear on the x and y axes as a function of the z value.

4.4 Transforming lines and planes

In the discussion so far we have considered the transformation of points. Lines are generally transformed by transforming the two end points separately. Planes, however, may be transformed differently. Three points define a plane and we can transform planes, much like lines, by transforming these three points. However, planes are usually defined by the (implicit) equation for a plane, $f(x, y, z) = ax + by + cz + d = 0$. If we define a column vector, $\boldsymbol{a} = [a, b, c, d]'$ then writing an arbitrary point as $\boldsymbol{p} = [x, y, z, 1]'$ points on the plane satisfy $\boldsymbol{a} \cdot \boldsymbol{p} = 0$ or $\boldsymbol{a}'\boldsymbol{p} = 0$.

Now if we transform all points $\boldsymbol{p}$, with a transformation matrix M, this is equivalent to transforming $\boldsymbol{a}$ so that the condition $\boldsymbol{a}_n'\boldsymbol{p} = 0$ defines the transformed plane, where $\boldsymbol{a}_n = Q\boldsymbol{a}$ and Q is the transformation matrix for $\boldsymbol{a}$. Now:

$$(Q\boldsymbol{a})'(M\boldsymbol{p}) = 0 \ ,$$

and we can now use the identity $(AB)' = B'A'$ to write $\boldsymbol{a}'Q'M\boldsymbol{p} = 0$. This can only be satisfied if $Q'M = \alpha I$. Assuming $\alpha = 1$ leads us to:

$$Q = (M^{-1})' \ ,$$

so that the vector of coefficients, $\boldsymbol{a}$, must be multiplied by the transpose of the inverse transformation matrix of $\boldsymbol{p}$. Some care must be taken because there is no guarantee that M^{-1} will exist, particularly if the transformation represents a projection, as we will see later.

In general the transformation of points, objects, planes, etc. can be most easily accomplished through combining elementary transformations using matrix multiplication. There is, however, a rather more abstract method, based on a geometrical understanding of vector operations, particularly the vector, or dot, product and the cross product. Recall that the dot product of a vector $\boldsymbol{r}$ with a unit vector $\boldsymbol{u}$, gave the length of the projection of $\boldsymbol{r}$ onto $\boldsymbol{u}$. Thus the projection of the vector $\boldsymbol{r}$ onto $\boldsymbol{u}$ is given by $(\boldsymbol{r} \cdot \boldsymbol{u})\boldsymbol{u}$.

The vector cross product, denoted as $\boldsymbol{r} \times \boldsymbol{s}$ is given by:

$$\begin{aligned} \boldsymbol{t} = \boldsymbol{r} \times \boldsymbol{s} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} \times \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} &= \det \begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ r_1 & r_2 & r_3 \\ s_1 & s_2 & s_3 \end{bmatrix} \\ &= (r_2s_3 - r_3s_2)\boldsymbol{i} + (r_3s_1 - r_1s_3)\boldsymbol{j} + (r_1s_2 - r_2s_1)\boldsymbol{k} = \begin{bmatrix} r_2s_3 - r_3s_2 \\ r_3s_1 - r_1s_3 \\ r_1s_2 - r_2s_1 \end{bmatrix} \ , \end{aligned}$$

where det means find the determinant of the matrix, as shown and $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$ are unit vectors along the x, y and z axes respectively. This vector, $\boldsymbol{t}$, is perpendicular to the plane defined by $\boldsymbol{r}$ and $\boldsymbol{s}$ and has length $\|\boldsymbol{r}\|\|\boldsymbol{s}\| \cos\theta$, where θ is the angle between $\boldsymbol{r}$ and $\boldsymbol{s}$. Using these geometrical concepts it is possible to define the transformation matrices from first principles, although it can be rather unintuitive. For this reason we shall stick to composing elementary transformations.

4.5 Transformations are a change of coordinate systems

So far we have discussed transformations of objects which all lie in the same coordinate system. When this is the case we could equally as well say that the objects stayed the same, but the coordinate system was transformed - it is just another way of saying the same thing. Often this view makes more sense because the real world objects do not change position as we rotate or translate them (to reflect our motion for example), although our view of them does. However if we are stationary and the object is moving then, the former view is more logical.

Consider a point $\boldsymbol{p}^{(i)}$ in the ith coordinate system, then define the transformation between from the jth to the ith coordinate system to be $M_{i\leftarrow j}$. Now:

$$\boldsymbol{p}^{(i)} = M_{i\leftarrow j}\boldsymbol{p}^{(j)} \ ,$$

and $\boldsymbol{p}^{(i)}$ and $\boldsymbol{p}^{(j)}$ are the same point in different coordinate systems. The coordinate system transformation matrices can be combined just like the earlier transformation matrices.

One useful coordinate system transformation is given by:

$$M_{R\leftarrow L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = M_{L\leftarrow R} \ ,$$

which transforms from a left handed to a right handed coordinate systems (and is its own inverse).

The view of objects having their own coordinate systems can make a lot more sense, these being relative to some world coordinate system. In general the transformation between coordinate systems in which a point is represented will be equal to the inverse of the corresponding transformation of the set of points in a fixed coordinate system. When using multiple transformations we can use the matrix identity $(AB)^{-1} = B^{-1}A^{-1}$ to work out the inverse transformation when the transformation is composite.

Viewing objects as having their own frame of reference can be particularly useful when dealing with sub-objects. Considering the tricycle shown in Figure 4.4, we see that the tricycle exists in a world coordinate system, (x, y, z). The main frame the tricycle has its own coordinate system, (x_t, y_t, z_t) which is fixed along the tricycle but not in space. The front wheel of the tricycle also has its own coordinate system (x_w, y_w, z_w), which allows the wheel to rotate, and the handle bars to turn.

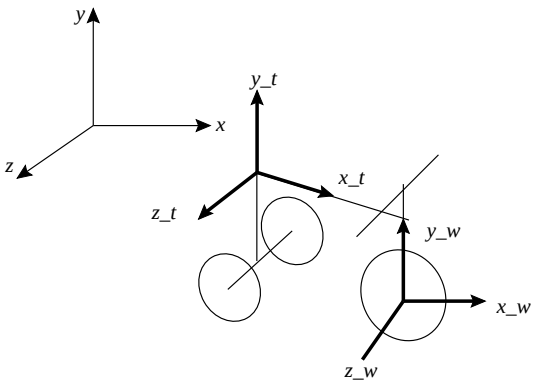


Figure 4.4: A simple tricycle.

If the tricycle moves by rotating the front wheel about the z_w axis, then we need to know what do with rest of the tricycle. As the front wheel moves, both the tricycle and wheel reference frame move by a translation in x and z and a rotation about y. The tricycle coordinate system is also tied to the wheel coordinate system as the handlebars are turned. Thus to produce a full time varying model of the tricycle would be rather complex.

The situation is simplified if the tricycle and wheel coordinates are parallel to the world coordinate axes, and that the wheel moves straight parallel to the x axis. As the front wheel rotates by an angle θ, a point, $\boldsymbol{p}$, on the wheel moves by an amount θr, where r is the radius of the wheel, and thus the tricycle moves forward θr units. Thus the point on the wheel has rotated about the wheel z axis by an angle θ and moved forward a distance θr. The coordinates of the new point in the original wheel coordinate system are:

$$\boldsymbol{p}^{*(w)} = T(\theta r, 0, 0)R_z(\theta)\boldsymbol{p}^{(w)} \ ,$$

while the coordinates in the new, translated, wheel coordinate system are:

$$\boldsymbol{p}^{*(w')} = R_z(\theta)\boldsymbol{p}^{(w)} \ .$$

To find the point in world coordinates we need to transform from wheel coordinates to world coordinates. Thus:

$$\boldsymbol{p}^{*} = M_{.\leftarrow w}\boldsymbol{p}^{(w)} = M_{.\leftarrow t}M_{t\leftarrow w}\boldsymbol{p}^{(w)} \ ,$$

that is we can convert to the world coordinate system by going from wheel to tricycle and then tricycle to world coordinate systems. Both these transformations are given by translations given by the initial position of the tricycle. Thus the new point in the world coordinate system is given by:

$$\boldsymbol{p}^* = M_{\cdot \leftarrow w} T(\theta r, 0, 0) R_z(\theta) \boldsymbol{p}^{(w)} \ .$$

There are other ways to represent this. In general if we were to represent the tricycle we would use the equations of motion of the tricycle to update $M_{\cdot \leftarrow w'}$ and $M_{t' \leftarrow w'}$ and then use these to determine the world coordinates of the updated points on the tricycle expressed in local coordinates.

4.6 Projection

This is possibly the most difficult section of these notes. However since many of the objects which exist in the application model will naturally be represented in a 3D coordinate system, and the screen is 2D, projection is a necessary part of any graphics module.

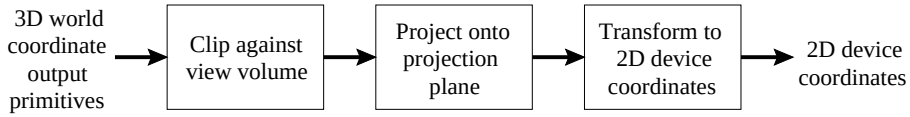


Figure 4.5: The procedure used to transform from 3D world coordinate window to the 2D viewport.

In Figure 4.5 we see that projection is only one part of the chain necessary for displaying a 3D application model on a 2D screen. The first step involves clipping against a 3D **view volume**, to remove those objects that are behind us or too distant to be visible. We then need to define the projection we wish to use and the viewing parameters, and apply this to the clipped part of the application model. Finally we transform to screen coordinates, in the manner we have already addressed.

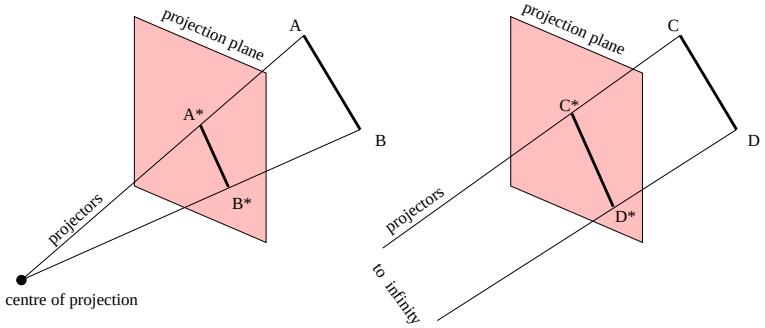


Figure 4.6: A perspective projection (left) and a parallel projection (right).

Projections are general mappings that transform a nD coordinate system into a mD coordinate system. Almost all useful projections involve $m < n$, and in computer graphics $n = 3$ and $m = 2$ most if the time[3]. The projection of a 3D object is described by straight 'projection rays' (called **projectors**), which emanate from the centre of the projection, pass through all points in the object and intersect the **projection plane** to produce the 'image'. The **centre of projection** is generally at a finite distance from the projection plane (giving **perspective projections**), but is sometimes defined at infinity (giving **parallel projections** – see Figure 4.6).

We consider **planar geometric projections** since the surface we are projecting the objects onto is a flat plane. Perspective projections require the definition of the centre of projection, which will be a point, $\boldsymbol{p}$, with homogeneous coordinates $[x, y, z, 1]'$. A parallel projection is defined by a direction (which can be expressed as the difference of two vectors; $[x, y, z, 1]' - [x^*, y^*, z^*, 1]' = [a, b, c, 0]$. Recall that points expressed in homogeneous coordinates with $w = 0$ are called **points at infinity**, and also correspond to directions. When we, as humans, view the world we experience something known as **perspective foreshortening**, which is very similar to the effect produce by a perspective projection. Perspective foreshortening means that the size of the project object varies as one over the distance to the object. What is more perspective projections are not very useful for representing exact shapes, since angles for all objects other than those parallel to the plane of projection are distorted, distances cannot be measured from the projection, and parallel lines are not preserved. The view is,

[3] In scientific visualisation it is not uncommon for n to be equal to 100, for instance, but we shall not consider these problems here.

however, more visually realistic. Parallel projections also distort angles, but maintain parallelism and distances. Parallel projections look less realistic.

4.6.1 Perspective projections

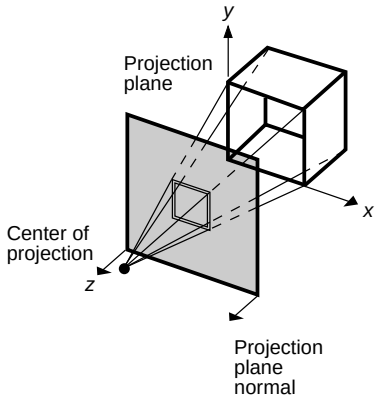


Figure 4.7: One point perspective projection of a cube (from Foley *et al.* (1993)).

In a perspective projection all parallel lines that are not parallel to the projection plane appear to go to a **vanishing point**, which will only be reached at infinity. If the set of parallel lines is also parallel to one of the principal axes, the point at which the lines meet is called an **axis vanishing point**. The number of axis vanishing points will depend on the number of axes cut by the projection plane. The number of axis vanishing points defines the type of the perspective projection. Figure 4.7 shows the one (vanishing axis) point projection of a cube onto a plane that only cuts the z axis.

Figure 4.8 shows a two (vanishing axis) point projection of the cube, where the projection plane cuts x and z axis but is parallel to the y axis. Note the two vanishing points for the x and z axes as well as the centre of the project, which in this case (and generally) will be different. Three point projections are used rather less regularly, since two point projections are generally sufficiently realistic.

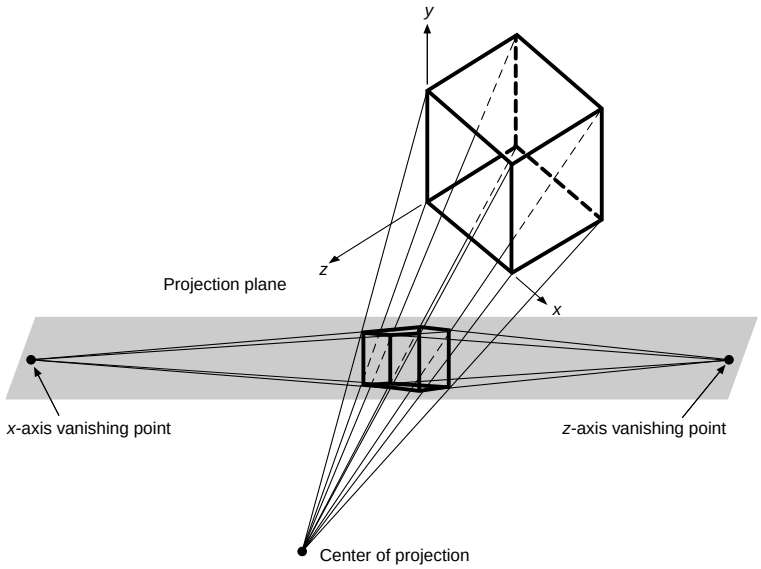
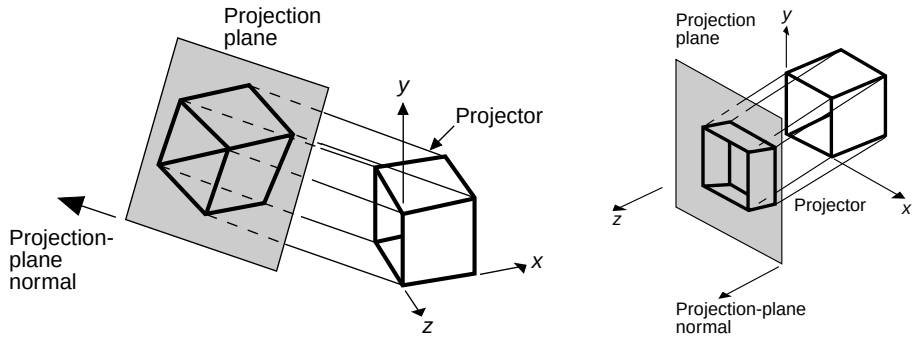


Figure 4.8: A two point perspective projection of a cube (from Foley *et al.* (1993)).

4.6.2 Parallel projections



(a) (b)

Figure 4.9: (a) A isometric projection of a cube, (b) an oblique projection of a cube (both from Foley *et al.* (1993)).

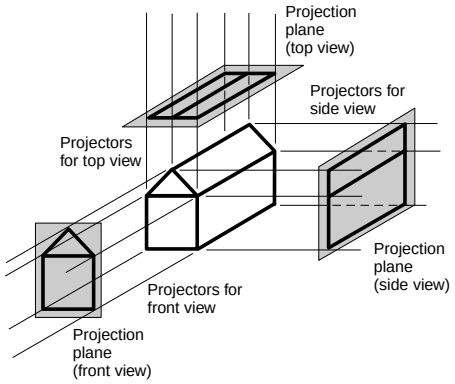


Figure 4.10: Three orthographic projections of a 'house ' (from Foley *et al.* (1993)).

Parallel projections can be divided into two categories: **orthographic** and **oblique**. Parallel **orthographic** projections have the direction of the projection parallel to the normal to the projection plane, as shown in Figure 4.10. These projections are frequently found in architecture and engineering drawings where three views are recognised: **front elevation**, **plan elevation** and **side elevation**. In these cases the projection plane is parallel to a principal axis.

Axonometric orthographic projections are parallel projections that use projection planes that are not normal to the principal axes. This projection preserves parallel lines but not angles and distances. An **isometric** projection is a frequently used **axonometric orthographic** projection in which the projection plane normal makes equal angles with the principal axes. There are only eight possible directions for isometric projects, an example being shown in Figure 4.9(a).

A different form of parallel projection is the so called **oblique** projection. In these the projection plane normal and direction of projection differ. The projection plane is still normal to the principal axes, and thus for objects parallel to the projection plane distances and angles are preserved. They are frequently used to illustrate 3D objects in texts (e.g. largely used in Foley *et al.* (1993)). There are many types of oblique projection but we do not consider them here.

4.6.3 Specification of 3D views

Recall that Figure 4.5 showed two stages to the 3D to 2D transformation, first a clipping, then a projection. The clipping is achieved by defining a **view volume** (the volume that can be seen). This section considers the specification of a view volume.

The projection plane (which is called the **view plane** in the graphics literature - a term we shall henceforth adopt), is defined by the **View Reference Point** (VRP), a point on the view plane, and the **View Plane Normal** (VPN), a normal to the view plane. Note that the view plane can be anywhere with respect to the objects in the application model - behind, in front of or even cutting through. Given we have the view plane (which defines an infinite 2D region) we now need to define a view window, so that objects outside the window are not displayed.

Defining a view window requires us to define two principal axes in the view plane and then the maximum and minimum coordinates of the view window. The axes are an integral part of the **Viewing Reference**

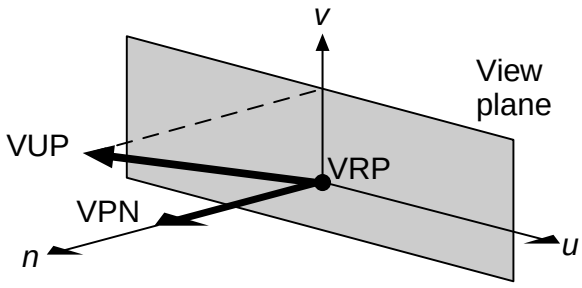


Figure 4.11: Definition of the view plane (from Foley *et al.* (1993)).

Coordinate (VRC) system. The VRC has the VRP as it's origin, with the n axis defined by the VPN. The v axis of the VRC is defined by the **View Up Vector** (VUP), and thus the u axis follows from the assumption of a right handed coordinate system, as shown in Figure 4.11. The VRP (point) and VPN and VUP (directions) are specified in a right handed world (application model) coordinate system. The view window is then defined by $(u_{\min}, v_{\min})$, $(u_{\max}, v_{\max})$.

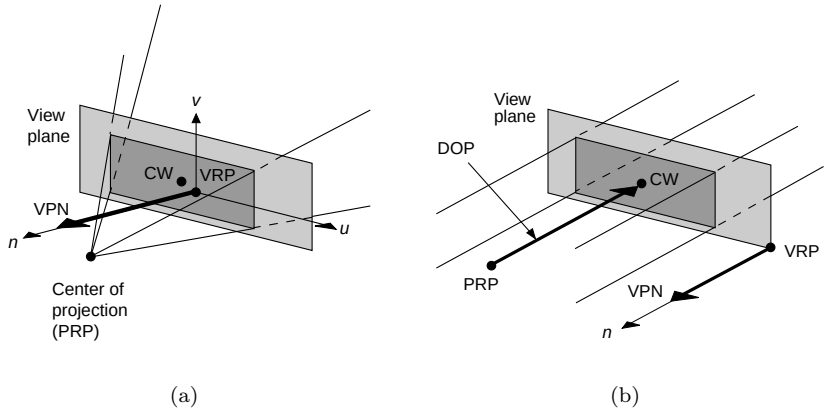


Figure 4.12: Definition of the infinite view volume from Foley *et al.* (1993): (a) perspective projection, (b) parallel projection.

The centre of projection and Direction Of Projection (DOP) are defined by a **Projection Reference Point** (PRP) and an indicator of the projection type (see Figure 4.12(a)). If the projection is a perspective projection the the PRP is is the centre of projection. If the projection is parallel, the the DOP is defined by the direction between the PRP and the centre of the projection window (CW in Figure 4.12(a)). The coordinates of the PRP are defined in the VRC, not world coordinates, which means the PRP does not change (relative to the VRP) as the VUP and VRP are changed. This means that it is simpler for the programmer to change the direction of projection required, at the expense of extra complexity if the PRP is moved (i.e. to get different views of the object).

The view volume of a perspective projection is the infinite volume defined by the combination of the PRP and the view window (Figure 4.12(a)), which defines a pyramid. This can be contrasted to the view volume for a parallel projection which gives an infinite parallelepiped ((Figure 4.12(b)). In general we will not want the view volume to be infinite, rather we will want to limit the number of output primitives which are displayed.

Figure 4.13 shows how finite volumes are defined by selecting the signed quantities F and B which define the locations of the **front** and **back clipping planes** (also called the **hither** and **yon planes**). Both these planes are parallel to the view plane, thus F and B are defined along the VPN. By changing the distances F and B it is possible to give a good feeling for the 3D shape of an object, almost as if we could slice through it.

To display the contents of the view volume the objects mapped into a unit cube whose axes are aligned with the VRC system (called the normalised projection coordinates), giving us the **3D viewport**, which is contained within a unit cube. The $z = 1$ face of this unit cube is then mapped to the largest square that can be displayed (in display coordinates). To create a wire-frame display of the contents of the 3D viewport we can simply drop

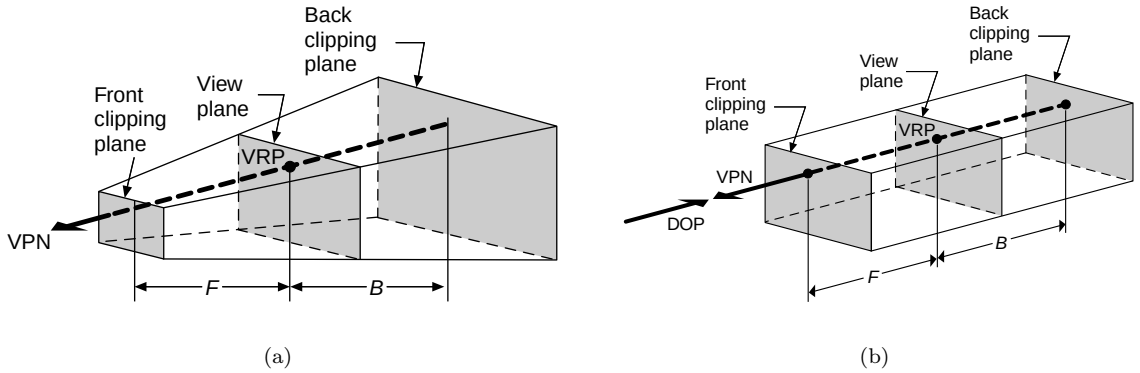


Figure 4.13: Definition of the finite view volume (from Foley *et al.* (1993)).(a) perspective projection, (b) parallel projection.

the z coordinate. For more complex displaying methods, such as applying hidden line removal algorithms we must retain the z coordinate.

In general two matrices are used to represent the complete view specifications; the **view orientation matrix** and the **view mapping** or view projection matrix. The view orientation matrix gives information about the position of the viewer relative to the object and combines the VRP, VPN and VUP. The 4×4 matrix then transforms positions represented in world coordinates to positions represented in the VRC, that is we map the u, v, n axes into the x, y, z axes respectively (note the coordinate change is reversed).

The view mapping matrix uses the PRP, view window and the front and back distances to transform points in the VRC to points in normalised projection coordinates, as we shall see next.

4.7 Implementing 3D→2D projections

This section contains the details necessary to implement (and understand) understand planar geometric projections. Rather than get involve with a mathematical derivation this section gives details of the mathematics required to implement the methods. By working back from this, those students that wish to can gain a better understanding of the mathematics as well.

The overall aim here is to define **normalising transformations** N_{par} (parallel) and N_{per} (perspective) that transform the points in world coordinates within the view volume to points in the normalised projection coordinates. From these we can apply clipping algorithms and 3D to 2D projections in a simple manner. This essentially adds a preprocessing step to Figure 4.5. We treat the parallel and perspective cases separately, although as you will see there are a lot of similarities.

4.7.1 Parallel projections

In this case our canonical view volume (created by the mapping into normalised projection coordinates) is the unit parallelepiped defined by the planes, $x = -1$, $x = 1$, $y = -1$, $y = 1$, $z = 0$ and $z = -1$. Here we derive the transformation matrix N_{par} for very general parallel projections, which may be oblique. The steps necessary to achieve a canonical view volume are:

1. Translate the VRP to the origin.

2. Rotate the VRC such that it is aligned with the world coordinate system.

3. Shear (in x and y) so that the direction of projection is aligned with the z axis.

4. Translate and scale into the canonical view volume (parallelepiped).

To implement these transformations all at one go would require a lot of mathematical analysis, however using homogeneous coordinates we can easily create the necessary individual steps 1 to 4 and combine the transformation matrices. Steps 1 and 2 combined produce the view orientation matrix, while steps 3 and 4 give the view mapping matrix.

Step 1: Translate the VRP (that is the point at which we are 'looking') to the origin. The VRP is specified in world coordinates, so the translation is simply:

$$T(-VRP) = \begin{bmatrix} 1 & 0 & 0 & -vrp_x \\ 0 & 1 & 0 & -vrp_y \\ 0 & 0 & 1 & -vrp_z \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Step 2: Rotate the VRC such that it is aligned with the world coordinate system. The row vectors of the rotation matrix to perform this transformation are the unit vectors that are rotated by R into the x, y and z axes. The VPN is rotated into the z axis, so

$$R_z = \frac{\text{VPN}}{\|\text{VPN}\|} .$$

Similarly (recalling that the vector cross product gives vectors that are orthogonal to both input vectors) we can rotate u to x giving

$$R_x = \frac{\text{VUP} \times R_z}{\|\text{VUP} \times R_z\|} .$$

Finally the v axis is rotated into the y axis:

$$R_y = R_z \times R_x ,$$

where the result is already normalised since both R_x and R_z are. The final rotation matrix is given by:

$$R = \begin{bmatrix} r_{1,x} & r_{2,x} & r_{3,x} & 0 \\ r_{1,y} & r_{2,y} & r_{3,y} & 0 \\ r_{1,z} & r_{2,z} & r_{3,z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

The derivation is rather tricky, but the concept of rotating the two coordinate systems (world and view reference) so that they are equivalent is logical.

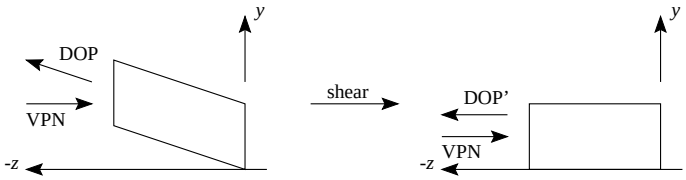


Figure 4.14: Shearing the view volume so that the direction of projection (DOP) is parallel to the z axis (side view).

Step 3: Shear (in x and y) so that the direction of projection is aligned with the z axis. The details of the derivation of the correct shear parameters can be found in Foley *et al.* (1993, p. 219). The direction of projection is the vector from the PRP to the centre of the view window. Thus the vector can be written:

$$DOP = \begin{bmatrix} dop_x \\ dop_y \\ dop_z \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{u_{\max} - u_{\min}}{2} - prp_u \\ \frac{v_{\max} - v_{\min}}{2} - prp_v \\ -prp_n \\ 0 \end{bmatrix} ,$$

noting that previous transformations mean that the world coordinates and VRC are the same thing. The shear matrix required is given by:

$$H_{par} = \begin{bmatrix} 1 & 0 & -\frac{dop_x}{dop_z} & 0 \\ 0 & 1 & -\frac{dop_y}{dop_z} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} ,$$

Step 4: The transformation to canonical coordinates. This step is not necessary for display, however most clipping algorithms require a canonical volume element, and it makes projection easier. Essentially we need to centre the view volume at the origin and then scale so that it has the correct dimensions. The matrix to achieve the translation is:

$$T_{par} = \begin{bmatrix} 1 & 0 & 0 & -\frac{u_{\max}+u_{\min}}{2} \\ 0 & 1 & 0 & -\frac{v_{\max}+v_{\min}}{2} \\ 0 & 0 & 1 & -F \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and the scaling is:

$$S_{par} = \begin{bmatrix} \frac{2}{u_{\max}+u_{\min}} & 0 & 0 & 0 \\ 0 & \frac{2}{v_{\max}+v_{\min}} & 0 & 0 \\ 0 & 0 & \frac{1}{F-B} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can now write the matrix for performing arbitrary parallel projections as $N_{par} = S_{par}T_{par}H_{par}RT(-VRP)$, that is the composition of several simple individual transformations.

4.7.2 Perspective projections

For perspective transformations a slightly different combination of transformations is needed to obtain a canonical view volume. These are:

1. Translate the VRP to the origin.
2. Rotate the VRC such that it is aligned with the world coordinate system.
3. Translate so that the centre of projection (i.e. the PRP) is at the origin.
4. Shear (in x and y) so that the direction of projection is aligned with the z axis.
5. Scale into the canonical view volume (truncated pyramid).

Steps 1 and 2: same as those in the parallel case.

Step 3: the translation of the centre of projection (i.e. the PRP) to the origin. This step is given by:

$$T(-PRP) = \begin{bmatrix} 1 & 0 & 0 & -prp_u \\ 0 & 1 & 0 & -prp_v \\ 0 & 0 & 1 & -prp_n \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Step 4: same as step 3 in the parallel case.

Step 5: Scale into the canonical view volume (truncated pyramid). The scaling matrix is given by:

$$S_{per} = \begin{bmatrix} \frac{-2prp_n}{(u_{\max}-u_{\min})(B-prp_n)} & 0 & 0 & 0 \\ 0 & \frac{-2prp_n}{(v_{\max}-v_{\min})(B-prp_n)} & 0 & 0 \\ 0 & 0 & \frac{1}{prp_n-B} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The reasoning behind the structure of S_{per} can be found in Foley *et al.* (1993, p. 225–226).

The final composite transformation matrix is given by $N_{per} = S_{per}H_{par}T(-PRP)RT(-VRP)$. Note that both N_{par} and N_{per} will not affect the homogeneous coordinate w since they are combinations of translations, rotations, scalings and shearings, so division by w is not normally necessary to return to 3D coordinates.

It is often useful to add an additional step to the computation of N_{per} transforming the truncated pyramid to a parallelepiped so that the same clipping algorithms can be used for parallel and perspective projections. This transformation matrix is given by:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

where $z_{min} = (prp_n - F)/(B - prp_n)$. Using M we can create a new perspective transformation matrix which maps the objects into the rectangular canonical view volume: $N^*_{per} = MN_{per}$.

4.7.3 Clipping

Once the application model has been projected into a canonical view volume the clipping of objects can be undertaken in two ways, either in 3D coordinates or homogeneous coordinates. We do not have time to deal fully with this subject, details can be found in Foley *et al.* (1993).

4.7.4 Projection

After the objects have been clipped we can project the remaining elements from the 3D canonical volume into 2D coordinates. This is achieved by using an appropriate transformation matrix. For the parallel projection case, the orthographic projection matrix is:

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

For a perspective projection the transformation matrix is given by:

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{bmatrix},$$

where $d = -1$ having performed the transformation using N_{per}.

4.7.5 Viewport transformation

The objects transformed by the projection will then be transformed into the viewport coordinates using the following matrices:

$$\begin{bmatrix} 1 & 0 & 0 & x_{v\min} \\ 0 & 1 & 0 & y_{v\min} \\ 0 & 0 & 0 & z_{v\min} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x_{v\max}-x_{v\min}}{2} & 0 & 0 & 0 \\ 0 & \frac{y_{v\max}-y_{v\min}}{2} & 0 & 0 \\ 0 & 0 & \frac{z_{v\max}-z_{v\min}}{1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This moves the origin of the canonical view volume to $(0, 0, 0)$, scales by the viewport dimension and then translates to the viewport origin. To plot the resulting 2D object we divide by w and then simply ignore the z coordinate and plot the x and y coordinates.

4.7.6 Overview

There was a lot of material in this section. To summarise the process of 2D viewing of 3D objects is given by:

- ➢ 3D → homogeneous,
- ➢ apply N_{par} or N_{per},
- ➢ homogeneous → 3D,
- ➢ clip,
- ➢ 3D → homogeneous,
- ➢ project using M_{ort} or M_{per},
- ➢ transform into device coordinates,
- ➢ homogeneous → 2D.

It is not always necessary to apply all these steps, for instance in some case clipping may not be very important (for instance when the application model only contains a small amount of information), and the steps related to clipping may be omitted. The series of projections outlined in this section are those which are used across all systems, however you should be aware that different terms are used for some of the coordinate systems, in different graphics libraries.

5 Surface Modelling

So far in this course, simple wire-frame graphics have been the focus. In this section we extend the wire-frame model to allow for surfaces and thus the illusion of solid objects. In this section we consider only surfaces composed of planar objects, in the next section we briefly develop methods for curved surfaces. The area is related to the concept of solid modelling, which involves the representation of volumes. Surface models are often used to specify the bounding elements of the solid object.

5.1 Polygonal Meshes

A **polygonal mesh** is a set of connected planar polygons (often triangles) which are used to represent the surface of an object. Because we are restricted to planar geometry these work best on naturally planar (typically man made) objects. However by specifying a sufficiently large number of very small polygons it is possible to approximate curved objects to arbitrary accuracy, at the expense of greatly increased data and processing cost.

Within a polygonal mesh each polygon edge is shared by only two polygons. An edge always connects two vertices, and the polygon is a closed sequence of edges (that is the start and end vertices are the same). Also every edge must be part of some polygon.

There are several possible representations for a polygonal mesh, the most appropriate one being determined by the specific part of the application for which it is to be used. Often several representations will be used within the same program, one for external storage, one for internal use and possibly one for interactive mesh definition.

As with most software problems there will be a space-time tradeoff for each representation. In general the more information on the structure of the mesh (that is the relations between the edges, vertices and polygons) the greater the space requirements, but the faster the processing of queries such as finding all the edges of a given polygon, finding both polygons of a given edge, or finding the vertices connected to a given edge.

The **explicit representation** defines each polygon P as:

$$P = \{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\} ,$$

where the vertices are stored in the order in which they would be encountered were we to move around the edge of the polygon. This is fine (space wise) for one polygon, however if we have more than one polygon they will have at least one shared edge, so there is unnecessary duplication. What is more there is no topological information here, that is there is no explicit information on shared edges or vertices. Also if we wanted to display the mesh as filled or outline polygons we would need to transform and clip each vertex and edge of each polygon. Then when the mesh was displayed each edge would be drawn twice.

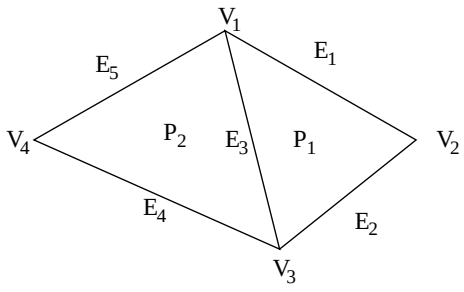


Figure 5.1: A polygon mesh.

An alternative method, known as the **pointers to a vertex list** method uses a technique whereby each vertex in the polygon mesh is stored just once. This creates a vertex list:

$$V = \{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\} ,$$

where n might be very large (this contains the vertices for all polygons). Each polygon is then represented as a series of pointers to vertices in the vertex list, $P = \{V_1^*, \ldots, V_n^*\}$. While this structure has reduced the storage requirements, it is still not very efficient for finding polygons that share edges and the edges will be drawn twice if all polygons are drawn.

The solution is to use a **pointers to an edge list**, which keeps the vertex list V, but adds an edge list E which stores all the edges using pointers to the two end vertices and to the polygons which the edge is part of. We also store the polygons as pointers to members of the edge list. An example is shown in Figure 5.1 where the vertex list $V = \{V_1, V_2, V_3, V_4\} = \{(x_1, y_1, z_1), \ldots, (x_4, y_4, z_4)\}$ and the edge list is $E_1 = \{V_1^*, V_2^*, P_1, O\}$, $E_2 = \{V_2^*, V_3^*, P_1, O\}$, $E_3 = \{V_3^*, V_1^*, P_1, P_2\}$, $E_4 = \{V_3^*, V_4^*, P_2, O\}$ and $E_5 = \{V_4^*, V_1^*, P_2, O\}$ where O represents the null or external polygon. Finally the polygon list would be $P_1 = \{E_1^*, E_2^*, E_3^*\}$ and $P_2 = \{E_3^*, E_4^*, E_5^*\}$. We have

used * to denote the fact we store a pointer to the objects, rather than the objects themselves. When displaying the polygon mesh the edges are drawn, which means there is no repetition of edges. The transformations will be applied to the vertex list, while clipping and scan conversion can be done using the edges. However it is still difficult to determine which edges are connected to a given vertex - all edge lists must still be examined - unless we add extra information explicitly.

5.2 Solid modelling

This section gives a very brief introduction to the concepts and methods used in modelling solid objects. This is important in CAD/CAM applications and the generation of photo-realistic images. The details of the implementations will not given but the interested reader can refer to Foley *et al.* (1993) or Hearn and Baker (1994). Further detail can be found in Foley *et al.* (1996). The essential difference between solid modelling and surface modelling is that we have to know the difference between the inside and the outside of a solid model.

So what capabilities should a solid model have? It should cover a **domain of representation** that is large enough to embody those objects which we might reasonably want to model. The representation should be **unambiguous** or **complete** in that given a particular representation there is only one solid that corresponds to it. We would also like it to be **unique** so that a given solid can only be represented one way. It might also be desirable that the representation is **accurate** so that the model is an exact representation of the object. Some types of model only allow straight lines, others can be used with curved surfaces. If the representation can ensure it is impossible to create an invalid object it will save us time having to check for validity. It should allow easy construction of the model. The representation should be **closed** under under rotation, translation and other operations, so that a transformed solid is still a solid. The representation should also be **compact** and **efficient**. It should be clear from this long list of requirements that no single system will satisfy all requirements simultaneously, thus there are trade-offs with every representation.

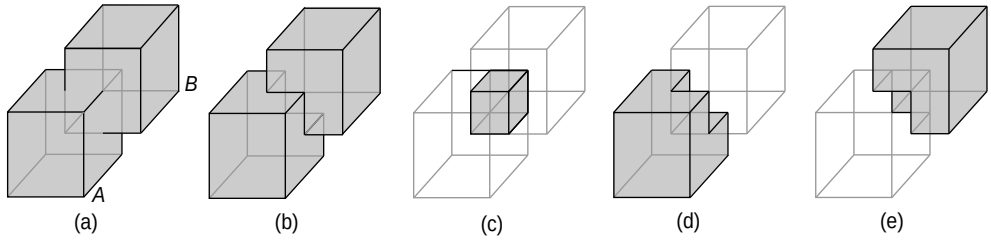


Figure 5.2: Boolean operation on solids. (a) Objects A and B, (b) $A \cup B$, (c) $A \cap B$, (d) $A - B$, (e) $B - A$ (from Foley *et al.* (1993)).

Just like the 2D case, we can define the standard Boolean set operators for 3D solids, as shown in Figure 5.2. Some care has to be taken since the application of a Boolean operator to two solids may not produce a solid. The **regularised Boolean operators** are defined in such a way as to ensure that their application always yields solids. For example the regularisation of the intersection of two cubes which only meet at a vertex is the null object.

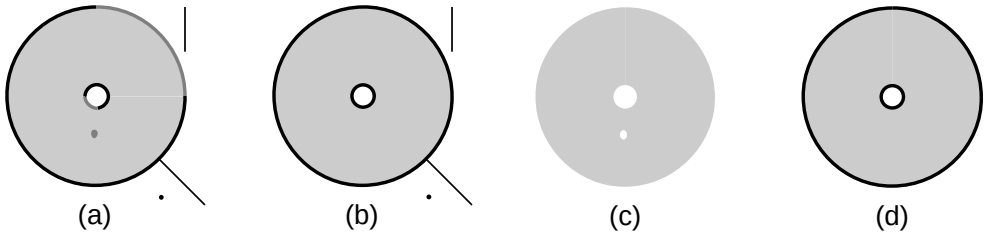


Figure 5.3: The regularisation of an object. (a) The original object, (b) the closure of the object, (c) the interior of the object and (d) the regularisation of (a). (from Foley *et al.* (1993)).

The concept of regularisation can be seen in Figure 5.3. The **boundary points** of an object are those which have zero distance from the object and it's complement. A **closed set** contains the boundary points, while an **open set** does not. The union of the set and it's boundary is called the **closure**. The **regularisation** of a set is the closure of the set's interior points.

We will now briefly review some of the more common solid representations.

5.2.1 Primitive instancing

As the name suggests, when using **primitive instancing** a set of primitives is defined by the model, which are directly relevant to the objects which the application should represent. For instance if the object is an engine, then the primitives might consist of pistons, rods, chains and cogs. This implies a rather high level specification, the primitives themselves may be rather complex to define.

5.2.2 Sweep representations

Sweep representations use a 2D planar template (cross-section) to define the profile of the object. This is then swept along a path, normal to the plane of the 2D template, to generate the 3D element. In **translational sweeps** or **extrusion** the sweep path is a straight line, and a 3D extension of the 2D template is produced. This can be modified by applying transformations to the 2D template as the template moves along the line that defines the translational sweep.

Often the sweep path is circular, generating a **rotational sweep**, which can be used to produce objects with some form of rotational symmetry about at least one of their axes.

The method can be generalised to sweep 3D objects. Note that the set of solids produced by even simple translational sweeps is not close under the Boolean set operation of union, since the union of 2 sweeps is in general not a sweep itself (because the symmetry will be broken unless the lines of sweep of both objects are parallel).

5.2.3 Boundary representations

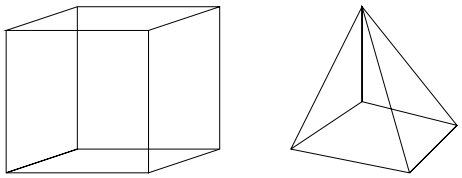


Figure 5.4: Simple polyhedra.

Boundary representations or **b-reps** describe the solid object in terms of its boundaries, that is the vertices, edges and faces. We will only consider planar b-reps, however curved surfaces can be represented within the same framework, although it is common to represent curved surfaces as a series of planar faces (c.f. finite elements). This uses representation methods that are very similar to those discussed in the Section 5.1.

Most b-rep systems can only represent shapes that have boundaries that are **2-manifolds**. This means that an edge can be shared by at most 2 faces. A **polyhedron** (which belongs to the set of objects having 2-manifolds) is a solid which is bounded by a set of polygons whose edges belong to only one other polygon. A **simple polyhedron** has no holes in it - that is it can be transformed into a sphere without breaking any of the connectivity. **Euler's formula** tells us that for a simple polyhedron with V vertices, E edges and F faces, $V - E + F = 2$ (see Figure 5.4). This gives us necessary conditions for an object to be a simple polygon, but not sufficient conditions since there exist objects for which Euler's formula is satisfied, but which are not simple polyhedra. Sufficient conditions are given by ensuring that each edge only connects two vertices and is shared by only two faces, faces do not inter-penetrate and at least three faces meet at each vertex.

One advantage of the b-rep system is that b-reps can be combined using the regularised Boolean set operators, such as union, to produce new b-reps, although this can be rather complicated to implement.

5.2.4 Spatial partitioning representations

Spatial partitioning representations divide the solid to be represented into a number of primitive solid objects, which are more simple than the object. One of the most general forms, known as **cell decomposition** represents the object as a number of small, simple cells - which will often be parametrically represented and curved. Combining these cells in a bottom-up fashion allows us to define more complex objects. Combining the cells, sometimes called **gluing**, is like a union, but must ensure that the objects do not intersect.

Spatial occupancy enumeration is a special case of cell decomposition where only one type of cell (typically a cube) is used on a fixed, regular grid. The cells, often referred to as **voxels** (volume elements), represent a discrete version of the object and have the problems that we met earlier in the transformation from 2D objects to pixels. There is also the problem of storage space since the amount of information need grows as n^3 if n is the number of voxels along each axis. The storage requirements can be reduced by using **octrees** which applies a binary subdivision in a divide-and-conquer algorithm. In general the number of nodes in an octree (which gives the amount of storage needed) will be proportional to the objects surface area, which is of order n^2 rather than n^3. There has been a lot of work done on the processing (by Boolean set operations) of octrees and efficient algorithms exist.

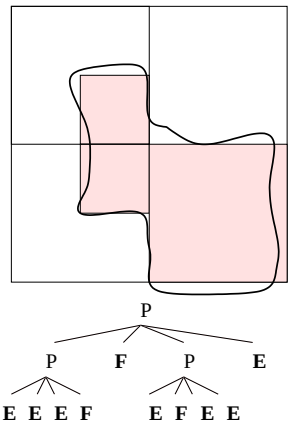


Figure 5.5: A simple quadtree.

To visualise the implementation of an octree, it is easier to understand it's 2D equivalent, the quadtree (Figure 5.5). In a quadtree the cells are squares, with each square being divided into four smaller squares at each level of the tree. Figure 5.5 shows the use of a 3 level quadtree to represent a planar object shown by the thick black line. A node in the quadtree can either be partially covered by the object (P), fully covered (**F**) or not covered (**E**). By defining the first leaf to be the lower left hand square of the division of the node above, the full information can be stored by just 3 values at each node. The leaf nodes will always be **F** or **E**. Since the position of the nodes is fixed, there are some problems with using these sort of models (which are also used widely in image processing where they are referred to as **tree** based models.)

Binary Space-Partition (BSP) trees divide space into two using arbitrary planes. BSP trees were originally developed to aid visible surface determination. Each internal node of the BSP tree is associated with a plane and has two children, inside and outside (for each side of the plane). If the normal points out of the object then the left child is behind the plane (inside) and the right child is in front of the plane (outside). The construction of BSP trees is rather complex and we do not address it further here. More information can be found in Foley *et al.* (1993, p. 386).

5.2.5 Constructive solid geometry

In **constructive solid geometry** the solid is defined using the regularised Boolean operations on some simple primitives such as shown in Figure 5.6. The means by which the primitives in the representation are specified varies across implementations. It is clear that cell decomposition and spatial occupancy enumeration can be viewed as special cases of the constructive solid geometry model.

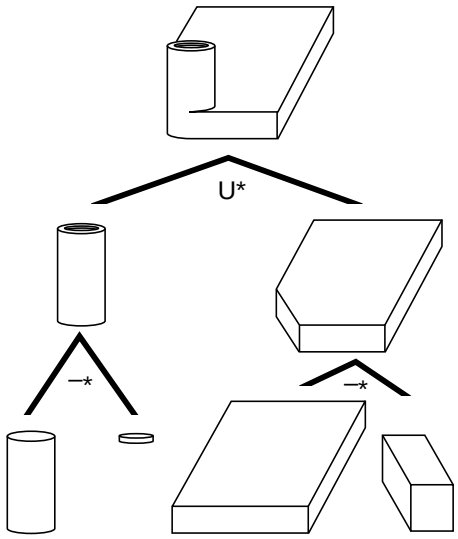


Figure 5.6: The use of constructive solid geometry. (from Foley *et al.* (1993)).

5.2.6 Which model to use?

Clearly there are several models which one could use to model a given solid. Which model is best will depend upon many considerations, as outlined at the start of this section. For instance spatial partitioning and b-rep (with polygonal planar boundaries) will not be very accurate for curved objects (although if we take smaller and smaller divisions we can minimise this at the cost of increased storage and computation). Foley *et al.* (1993, p. 390) gives a more complete discussion of the merits of the various approaches.

Another issue in solid modelling is that of the definition of the solids in the first place. This can be done mentally for simple objects, however for more complex tasks, such as the display of a room, automated or computer assisted methods will be necessary.

6 Curves

So far in the course we have dealt only with models which use straight lines or planar surfaces. Natural objects have a variety of possible shapes, some curved and some very irregular (but often repetitive at the fine scale - fractals). In the first part of this section we deal with curves in 2D. The second part outlines the specification of smooth surfaces in 3D.

6.1 2D curves

Most of the curves used in *computer graphics* are parametric curves, that is the position of the curve in (x, y) space is given by some **parameterised** function.

The explicit form of the functions, such as $y = f(x)$, are generally not appropriate because:

- ➢ it is impossible to get multiple y values for a given x value,
- ➢ the form is not rotationally invariant and,
- ➢ you cannot describe curves with a vertical tangent.

The implicit form of a function, such as $f(x, y) = 0$, is not very suitable for representing curves either because:

- ➢ the given equation may have more solutions than we want,
- ➢ to restrict the solution to one branch we need extra constraints,
- ➢ joining curves can be a problem.

Thus a third form of curve, know as a **parameterised** function is used to represent general 2D curves. Here we let $x = x(t)$ and $y = y(t)$ where t is some index which you can think of as time or distance along the curve. The most commonly used type of parametric curve is the **piecewise cubic polynomial curve**. This is much like the polylines (piecewise continuous linear objects) used in earlier sections, except the individual elements are now cubic functions of t. Cubic polynomials are often used because they are sufficiently flexible to approximate general curves but do not have too many coefficients (parameters).

6.1.1 Piecewise cubic polynomial curves

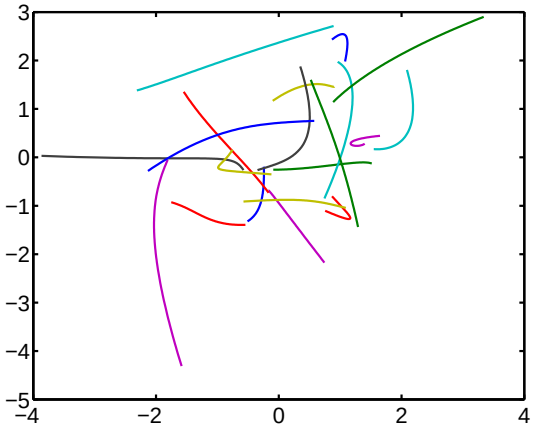


Figure 6.1: A sample of cubic pieces.

The general form of a curve segment is given by $q(t) = (x(t), y(t))'$ where:

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \ , \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \ . \end{aligned}$$

We restrict $t \in [0, 1]$. Some examples are shown in Figure 6.1. Using matrix and vector notation we can write:

$$\boldsymbol{t} = \begin{bmatrix} t^3 & t^3 \\ t^2 & t^2 \\ t & t \\ 1 & 1 \end{bmatrix} \ , \ C = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \end{bmatrix} \ ,$$

so that $q(t) = C\boldsymbol{t}$, giving a compact notation. It is clear that this representation can be used for a single curve segment, but how do we join curve segments? Well ideally we would like the curve segments to join (that is we would like continuity) and we would like the curves to have the same slope when they join (smoothness, also called continuity of the derivative).

We ensure continuity and smoothness by matching the derivatives (or slopes) of the curves at the joining points. Thus we compute:

$$\frac{\partial q(t)}{\partial t} = \left(\frac{\partial x(t)}{\partial t}, \frac{\partial y(t)}{\partial t} \right) = \frac{\partial (C\boldsymbol{t})}{\partial t} = C \frac{\partial \boldsymbol{t}}{\partial t} \ ,$$

where:

$$\frac{\partial \boldsymbol{t}}{\partial t} = \begin{bmatrix} 3t^2 & 3t^2 \\ 2t & 2t \\ 1 & 1 \\ 0 & 0 \end{bmatrix} .$$

It is possible to define many types of continuity:

- G^0 geometric continuity - the curves join
- G^1 geometric continuity - the curves join with equal tangent directions.
- C^1 continuity - the curves join with equal tangent directions and magnitude (first derivatives equal).
- C^n continuity - the curves join with equal n'th derivatives.

If we consider t to be time, then C^1 continuity implies that the velocity of an object moving along the curve is continuous. It will often also be useful to enforce C^2 continuity on the curve so that the acceleration is also continuous. For instance if the curve were to represent the path of an artificial camera it might look rather odd if the acceleration suddenly changed across the piecewise segments.

So how do we join the segments? There are four unknowns for each of x and y thus we require four constraints to determine the system of equations at each segment. The constraints we have are:

- constraints on end points,
- constraints on tangent vectors (derivatives),
- and constraints on continuity between segment end points.

In this section we will consider three main types of curve:

- **Hermite** – defined by two end points and the tangent vectors at the end points,
- **Bézier** – defined by two end points and two control points which determine the tangent vectors at the end points,
- **B-splines** – defined by four control points, with C^2 continuity.

6.1.2 Hermite curves

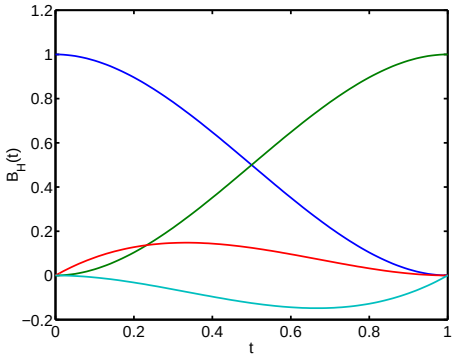


Figure 6.2: The Hermite basis functions.

A Hermite polynomial form is specified by the definition of two end points $\boldsymbol{p}_1$, $\boldsymbol{p}_4$ and two end tangent vectors, $\boldsymbol{r}_1$, $\boldsymbol{r}_4$. The question is how do we determine the coefficients of the polynomials, that is the matrix C given these constraints? The answer is complex but when followed in a step by step fashion it should be comprehensible.

The essential step is that we expand C into two matrices. Let us consider the row of C that corresponds to the x coefficients, C_x. We can write:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x = C_x \boldsymbol{t} = G_x M_H \boldsymbol{t} ,$$

where G_x is the x-component of the so called **geometry matrix**, which is given by the constraints and M_H is the **Hermite basis matrix** which defines the basis functions used to represent the function $x(t)$. We know what G_x is - it is simply $[p_{1,x} p_{4,x} r_{1,x} r_{4,x}]$ from the constraints (this is the definition of G). Thus we can determine M_H using the above equations as outlined in Foley *et al.* (1993, p. 333). This yields:

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

We can now write $x(t)$ in two ways; either as a function $a_x t^3 + b_x t^2 + c_x t + d_x$ or as a linear combination on finite support non-linear basis functions $G_x B_H(t)$ where $B_H(t) = M_H \boldsymbol{t}$ are the **Hermite blending functions**. The basis (or blending) functions for Hermite polynomials are shown in Figure 6.2. Note this is the only time you might see the curves plotted as a function of t, since in real applications x is plotted against y.

Now given the geometry matrix, G, and given that we know the Hermite basis matrix, M_H, we can compute $C = GM_H$. To draw the curves we simply evaluate $x(t)$ and $y(t)$ at successive values of t, where the step size is kept small. Hermite curves are often used in graphics packages for specifying smooth curves.

6.1.3 Bézier curves

Bézier curves are very similar to Hermite curves except that we use four control points, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$ and $\boldsymbol{p}_4$. The intermediate control points give the starting and ending tangent vectors, so that $\boldsymbol{r}_1 = 3(\boldsymbol{p}_2 - \boldsymbol{p}_1)$ and $\boldsymbol{r}_4 = 3(\boldsymbol{p}_4 - \boldsymbol{p}_3)$. Thus the Bézier **geometry matrix** is G_x is - it is simply $[p_{1,x}, p_{2,x}, p_{3,x}, p_{4,x}]$. This means that the **Bézier basis matrix** will be:

$$M_B = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & 6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

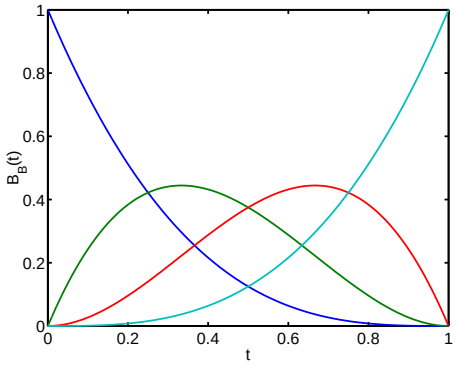


Figure 6.3: The Bézier basis functions.

Again we can compute the basis (or blending) functions which are shown in Figure 6.3. These basis functions are known as Bernstein polynomials. Note that the Bézier curve is completely contained within the convex hull defined by the four control points.

If we need to join two Bézier curves, with control points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$, $\boldsymbol{p}_4$ (in both), $\boldsymbol{p}_5$, $\boldsymbol{p}_6$, $\boldsymbol{p}_7$, then we will need an extra condition to ensure G^1 or C^1 continuity. Having $\boldsymbol{p}_4$ in both curves ensures G^0 or C^0 continuity, however we have:

- ➢ G^1 continuity if $\boldsymbol{p}_3 - \boldsymbol{p}_4 = k(\boldsymbol{p}_4 - \boldsymbol{p}_5)$ where $k > 0$,
- ➢ C^1 continuity if $\boldsymbol{p}_3 - \boldsymbol{p}_4 = \boldsymbol{p}_4 - \boldsymbol{p}_5$.

This gives us the recipe for constructing arbitrarily long curves with the desired properties.

6.1.4 B-splines

Splines have a long history in computer graphics and before that in technical drawing. The name comes from the once common practice of using long thin strips of metal or wood, together with carefully chosen weights to create curves used to define the shapes of boats, cars and aeroplanes. The **natural cubic spline** is a mathematical model for such a thin strip, which passes exactly through all its control points while minimising the bending energy. The **natural cubic spline** has C^2 (and therefore C^0, C^1) continuity and is thus smoother than the Hermite and Bézier curves discussed above.

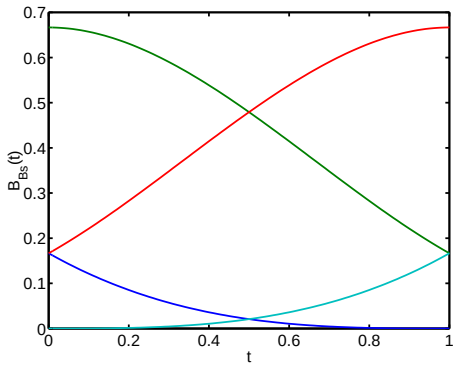


Figure 6.4: The B-spline basis functions.

We focus on **B-splines** in this section, that is splines that depend only on a few **local** control points, the B standing for Basis. These are more efficiently computed than **natural cubic splines**, however they do not interpolate their control points. In this section we consider a curve specified by a number of control points, not just a curve segment.

Cubic B-splines approximate a series of $m + 1$ control points, $\boldsymbol{p}_i$, $i = 0, ..., m + 1$, where m is at least 3. The approximating curve consists of $m - 2$ segments Q_j, $j = 3, ..., m$. For each $j \geq 4$ there is a **knot** (or join) between Q_{j-1} and Q_j at the parameter value t_j. The initial and final points t_3 and t_{m+1} are also **knots**, giving $m - 1$ knots, see Figure 6.5.

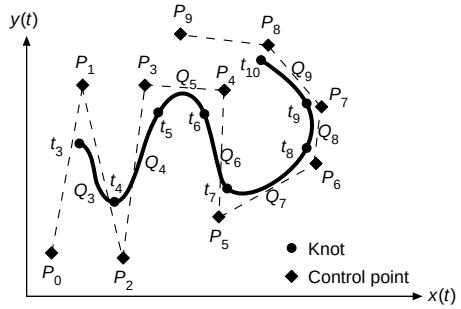


Figure 6.5: An example of uniform non-rational B-splines (from Foley *et al.* (1993)).

For now we consider the case that the knots t_i are uniformly spaced along the curve (**uniform non-rational B-splines**). The non-rational refers to the fact that the splines cannot be represented as the ratio of two cubic polynomials. Since these are B-splines we can represent them as a linear combination of fixed non-linear basis functions (which is not true for natural cubic splines).

We simply show how to implement uniform non-rational B-splines in this section and do not derive the results. Like the Hermite and Bézier curves above we can define a geometry matrix $G_{Bs,j} = [\boldsymbol{p}_{j-3}\boldsymbol{p}_{j-2}\boldsymbol{p}_{j-1}\boldsymbol{p}_j]$ for each curve segment Q_j, each time assuming t varies from $j-3$ to $j-2$ over the segment. Now $\boldsymbol{t}$ remains the same and the B-spline basis matrix is given by:

$$M_{Bs} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} .$$

We can again derive the basis (blending) functions $B_{Bs}(t)$, which are shown in Figure 6.4.

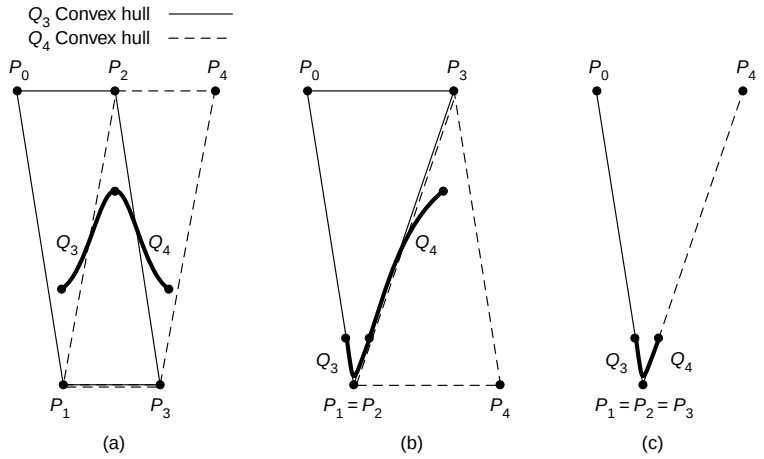


Figure 6.6: Using uniform non-rational B-splines (from Foley *et al.* (1993)).

Non-uniform non-rational B-splines can also be defined where, as the name suggests, the spacing between knot points is no longer uniform. Thus there are no longer fixed basis (blending) functions, however by adding control points on top of each other we can change the continuity of the curve at those points as shown in Figure 6.6. This gives enhanced flexibility and the ability to represent both smooth and sharp changes in direction using a small number of control points. This ability is ideally suited for describing the shapes of fonts (such as the quadratic B-splines used to define True Type fonts).

Rational curves are generated using the ratios of (e.g. cubic) polynomials such as:

$$x(t) = \frac{X(t)}{W(t)}, \qquad y(t) = \frac{Y(t)}{W(t)} ,$$

where $X(t)$, $Y(t)$ and $W(t)$ are cubic polynomial curves. The use of rational polynomials is very similar to the use of homogeneous coordinates, since we can write the curve segment $Q(t) = [X(t), Y(t), W(t)]$. Rational polynomial curves are invariant under translation, rotation, scaling and (unlike non-rational curves) perspective projections. Thus a perspective projection of a rational polynomial curve can be obtained by applying the projection operators to the control points. Rational curves can also exactly define any conic section, which is not the case for non-rational curves. In many CAD applications, where conic sections are common, the use of rational curves is much more efficient. The most common form of rational curve used is the so called **NURBS** curve, that is the Non-Uniform Rational B-Spline.

6.1.5 Comparison of curves

In the above three sections we have seen several varieties of piecewise cubic curves (some of which have different flavours). The question remains which one is the best to use for a given application? Well there are several

criteria by which we can select a method including:

- ➢ degree of continuity of the complete curve,
- ➢ speed of computation to generate the curve,
- ➢ ease of definition of the curve,
- ➢ ability of the curve to represent desired objects.

Actually, to some extent the choice can be avoided since any of the (uniform non-rational) curves can be rewritten as any other (uniform non-rational) form. Thus a user might define the objects using Bézier curves, while the application program stores them in its internal representation as B-splines. For instance Postscript fonts are stored as Bézier curves internally, however font designers may use packages that allow font definition using Hermite curves. Conversion from NURBS curves can be achieved using algorithms mentioned in Foley *et al.* (1996). Table 2, below, can be used when determining which method to use.

Table 2: The relative merits of different piecewise cubic curve definitions

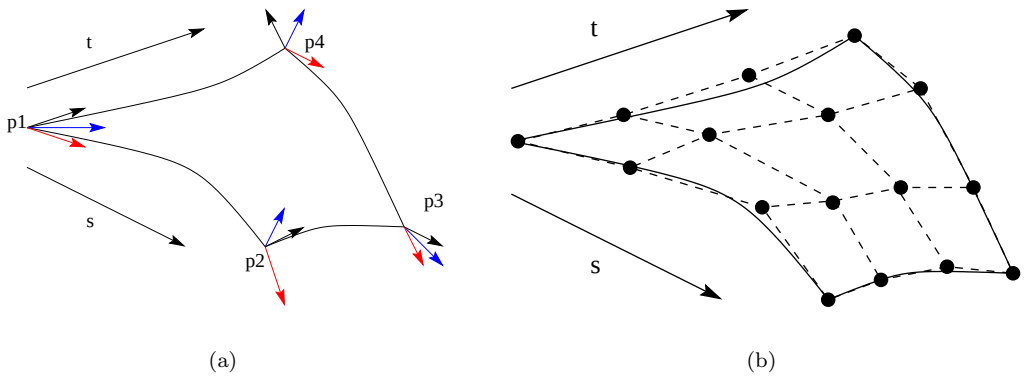
	Hermite	Bézier	Uniform B-spline	Non-uniform B-spline
Convex hull?	N/A	Y	Y	Y
Interpolates some?	Y	Y	N	N
Interpolates all?	Y	N	N	N
Ease of subdivision	good	best	average	worst
C Continuity	0	0	2	2
G Continuity	0	0	2	2
C Continuity possible	1	1	2	2
G Continuity possible	1	1	2	2
Number of parameters	4	4	4	5

The ease of subdivision is related to how easy the curve is to actually draw.

6.2 3D curves

The extension of the 2D curves to 3D curves is very straightforward. We simple add an extra polynomial $z(t)$ to give us the behaviour in the third dimension. Nothing else changes since our control points and vectors are now in 3 space, we can proceed exactly as we have done above. In 3D the advantage of using NURBS curves becomes more apparent since we will often project the curved objects using a perspective transform.

6.3 Surfaces



(a) (b)

Figure 6.7: How we define 2D curved surfaces using bicubic methods (a) Hermite representation, (b) Bézier representation.

We can also extend the above methods to define 2D curved surfaces in 3D. This has several applications, particularly in modelling naturally curved objects. The mathematics get a bit complicated at this point so we will not develop them in any great depth here.

6.3.1 Bicubic surfaces

Parametric bicubic surfaces are so called because rather than having just one cubic in t to represent them, here we use two parameters s and t to parametrise the curves. The way that this works is sketched in Figure 6.7. Essentially, s and t define a 2D manifold in the 3D space and we can use the same concepts of the geometry and basis matrices to write the functions down (see Foley *et al.* (1993, p. 351) for details). We can define the same curves as we used in the previous section on 2D/3D curves, but this time we need to supply more geometry information.

6.3.2 Hermite surfaces

As before the surface is specified by defining the end points - in this case we need four to define the edges of the four sided polygon, together with the tangents along the curves in the t and s directions and the **twist** of the surface (given by the second partial derivative of the cubic functions with respect to t and s) at the four control points. Thus we require 16 conditions to specify a single cubic patch (see Figure 6.7(a)).

6.3.3 Bézier surfaces

Like Bézier curves, Bézier surfaces are specified by control points rather than tangents. Thus we use 16 control points to specify the patch, as shown in Figure 6.7(b). While we require 16 control points to define a single patch, if we join patches we can see that the number of control points per patch will reduce, since neighbouring patches will use the same edge control points. Just as in the previous section we can use B-splines or any of the other curves to create bicubic surfaces.

6.3.4 Normals to the surfaces

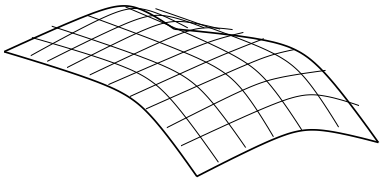


Figure 6.8: Evaluating a surface.

If we represent the parametric surface as $q(s,t)$ then the tangents to the surface in the s and t directions are given by:

$$\frac{\partial q(s,t)}{\partial s} \quad \text{and} \quad \frac{\partial q(s,t)}{\partial t} \ ,$$

respectively. Thus a normal to the surface is given by:

$$\frac{\partial q(s,t)}{\partial s} \times \frac{\partial q(s,t)}{\partial t} \ ,$$

the vector cross product of the two tangents.

6.3.5 Displaying the surfaces

Just as in the case with curves, we display the surfaces by approximating them with straight lines over very small increments. This time we fix either s or t and increment over the other variable. We do this for a number of fixed values of s and t to produce the mesh plots, such as shown in Figure 6.8.

7 The Visible Surface Problem

Almost all 3D objects we see are solid or sufficiently opaque that we can only see the 'front 'sides. Thus to realistically represent these objects on the computer screen we need to be able to calculate what parts of an object we can see and which are hidden. The methods used to perform these calculations are known as:

- ➢ **visible line determination** – 'wire-frame' models,
- ➢ **visible surface determination** – solid models,
- ➢ **hidden line removal** or **elimination** – 'wire-frame',

➢ **hidden surface removal** or **elimination** – solid.

These terms are all synonymous, but refer to different forms of the application model.

The fundamental concept of **visible surface determination** is simple - find those surfaces which we can see and draw them. The implementation is less simple. There are two brute force approaches to the problem, which are very different in terms of their implementation. We can either check for each pixel which object is closest, or check for each object which is closest and also what parts are visible.

The pseudo-code for the pixel approach is:

```
for (each pixel in the image) {
   determine the object closest to the viewer that is
      pierced by the projector through the pixel.
   draw the pixel in the appropriate colour.
}
```

To determine the object that is closest to the viewer we could compute, for each pixel the z distance (having performed the projection to 2D, but retained the z coordinate. Thus if we have an image with p pixels and n objects the cost of the algorithm would be of order np. Note that if the screen is 1024×768, $p = 786432$. This approach is generally referred to as the **image-precision** approach.

The pseudo-code for the object approach is:

```
for (each object in the image) {
   determine the parts of the object which are not
      obstructed by itself or other objects.
   draw those parts in the appropriate colour.
}
```

Since this routine only needs to compare all n objects with all other objects the cost will be approximately n^2. This approach is generally referred to as the **object-precision** approach. In general the number of objects will be much smaller than the number of pixels, so the object-precision scheme might seem attractive, however the individual steps in the procedure are generally a lot more complex than those in the image-precision approach, so that the image-precision approach is often quicker.

Object-precision calculations have an advantage if we need to change the resolution of the display or zoom in to a specific region. The visibility of the objects in the application model is determined in world coordinates, thus while the objects will require scan conversion, there is no need to recompute the depth ordering. Object precision systems were first designed for vector display systems, where their application is natural.

The optimal efficiency can be obtained by combining the benefits of both methods. We either need to determine, for a given projector, which objects are intersected and at what distance from the viewer, or for all the objects, whether they intersect and if so where. To minimise the time required to render a picture we need to perform these steps as efficiently as possible.

7.1 Coherence

We have already mentioned that **coherence** can be a very useful property. Coherence implies that there is a considerable degree of similarity in parts of the picture that are close to each other (in space and time). From a computer vision perspective (and probably human vision too) this coherence is vital to allow us to distinguish objects. The sharp changes across objects are much rarer and therefore easily detected (think about how camouflage works).

We exploit the coherence in the image by writing routines that reuse the calculations from one part of the image in nearby regions. There are several types of coherence which we can utilise.

➢ **Object coherence** – if objects are totally separate, then only the objects need to be compared not their component parts.

- **Face coherence** – the attributes will probably not vary very quickly (or at all) across a face. Some models may not permit faces to inter-penetrate.
- **Edge coherence** – edges will only change appearance where they cross behind a visible face or penetrate an object.
- **Implied edge coherence** – if two planar objects intersect, that intersection will be a line, which can be defined using two points of intersection.
- **Scan-line coherence** – we have already met this. The scan line of a typical image is very similar to the one above and below.
- **Area coherence** – the pixels in a small area tend to be similar because a group of pixels is covered by the same face.
- **Span coherence** – the pixels within a face on a scan line tend to be similar.
- **Depth coherence** – adjacent parts of the same object have similar depths, while different objects have less similar depths.
- **Frame coherence** – even with animated pictures, since a frame is displayed every 1/30th of a second, there will be considerable similarities across frames.

This list makes it clear that there are a large number of coherence properties that we could utilise to improve the speed with which we render 3D scenes. Their use is required to achieve reasonable frame rates.

7.2 Perspective projections

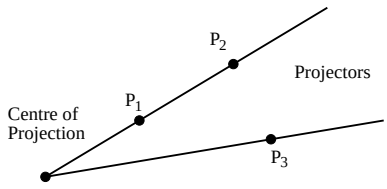


Figure 7.1: Occlusion in perspective projections.

In order to determine the visibility of 3D objects, we need to know their distance from the viewer along a projector. In parallel projections this is simple to define (since these are parallel to the z axis), however perspective projections are a little more complex to deal with. This can be seen in Figure 7.1 where we show how we can determine whether one point obscures another. We simply need to know whether both points lie on the same projector, and then which one is closer.

This will typically be done after the projection normalising transformation has been applied but before the final projection matrix is used. Thus the projectors will be all parallel to the z axis. Thus for perspective projections we use the following projection matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

where $z_{min} = (prp_n - F)/(B - prp_n)$ as before. This ensures that the objects are mapped into the rectangular canonical view volume: $N'_{per} = MN_{per}$ (see Section 4.7).

Having used this transformation matrix then we can check for occlusion of two points $p_1 = [x_1, y_1, z_1]'$ and $p_2 = [x_2, y_2, z_2]'$ by checking whether $x_1 = x_2$ and $y_1 = y_2$. To determine the closer point we simply find the larger of the two (usually negative) z values. It is important to remember whether we are using left or right handed viewing systems. It may seem more natural that large positive z values mean objects that are farther away, and many graphics texts and packages adopt this convention. We do not. Unless otherwise stated we shall assume that objects we are considering have been projected into their canonical view volumes in the rest of this section.

7.3 Extents and bounding volumes

It is quite simple to determine the rectangular extent or bounding parallelepiped for each object. We simply need to find the maximum and minimum values along each coordinate axis. This is shown for 2D and 3D objects in Figure 7.2. Given bounding elements are created, then to determine whether two objects overlap we can simply test the bounding elements. This simply involves a **minimax** test to determine whether $x_{max,1} < x_{min,2}$ and

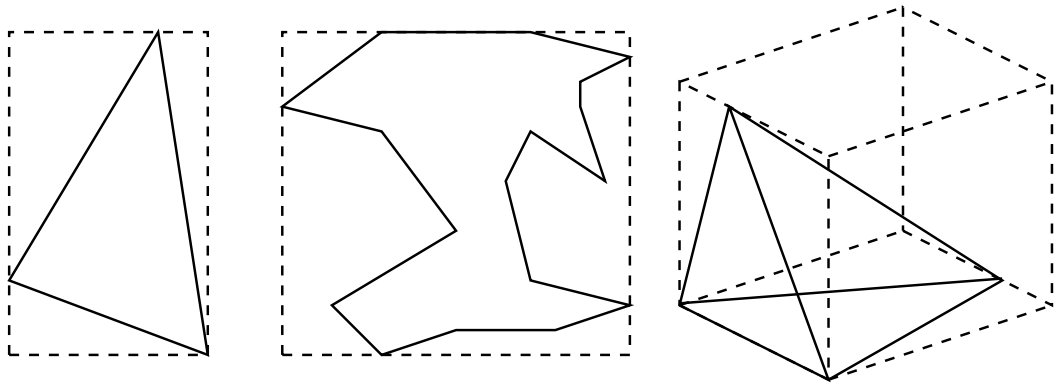


Figure 7.2: Bounding elements and volumes.

$x_{min,1} > x_{max,1}$ for each axis. This is clearly quicker than testing each point in the object. A similar approach, called the trivial reject is also used in clipping.

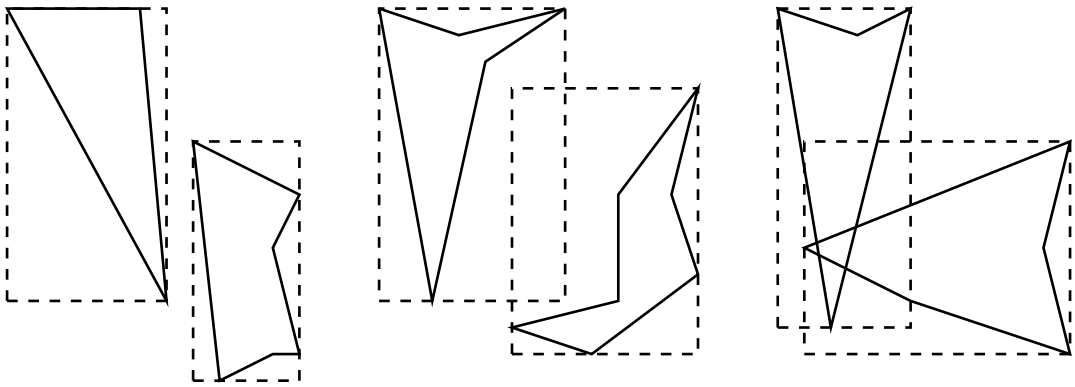


Figure 7.3: Bounding elements and their possible intersections.

There are three possible cases (illustrated in 2D in Figure 7.3). Only when the two bounding elements are disjoint can we obtain any advantage using this method. We could, or course, define a tighter bounding element, and therefore probably find more disjoint cases, but at the expense of a greater cost to compute the bounding element and test for occlusion. The optimal solution will depend on the objects being modelled.

7.4 Back Face Culling

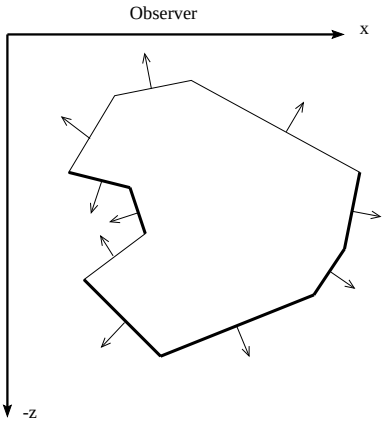


Figure 7.4: Back face culling.

If we are dealing will a solid object (and in particular if that solid object is approximated by a solid polyhedron) then all of its faces will completely enclose its volume. We have defined that each facet of the polyhedron has its normal facing outside, so that as long as we cannot see inside the object at any point (for instance because the front clipping plane intersected the object) then all facets with normals pointing away from the observer cannot be seen. This is illustrated in Figure 7.4, which shows a 2D cross-section through a polyhedron.

Since the direction of projection will be parallel to the z axis, taking the dot product of the surface normal with the direction of projection corresponds to:

$$\begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} ,$$

which effectively means that you need to test the sign of the z component of the surface normal. If n_z is negative then we know the facet is facing backwards and we can remove this facet from further consideration. If we only have one convex polyhedron this is all we need to do.

Since a projector passing through a closed polyhedron passes as many back facets as front facets, using this method will halve the number of computations which need to be undertaken in an image-precision method. On average it will also halve the number of polygon which need to be considered using a object-precision method, although this will depend on the geometry of the objects.

7.5 Spatial partitioning

By dividing the volume considered into a number of disjoint regions (such as used in quadtree and octree schemes) we can readily reduce the number of object comparisons (in both image- and object- precision methods) that need to be made since we only need consider objects within each region. If the objects are unevenly distributed in space, adaptive partitioning schemes may increase efficiency.

7.6 Hierarchical models

It may often be the case that the bounding volume of the top level in the hierarchy will define the bounding volume of all the components in the hierarchy. If this is the case then a null result from a test for whether a projector intersects an object at the top level means that objects further down the hierarchy need not be considered. This is related to object coherence.

7.7 The z-buffer algorithm

So far we have considered methods than can be used to reduce the number of polygons that we need to consider when performing visible surface determination. We now look at a specific algorithm, which is the most widely used and often implemented in hardware.

In addition to a frame buffer, which stores the 8 to 32 bits of colour information we also need a z-buffer which typically stores 16 to 32 bits of depth information for the visible polygons. This at least doubles the memory requirements but the simplicity and generality of the method means it is very widely used.

In essence the polygons that make up the scene are scan converted in arbitrary order. The z-buffer is initialised to zero (which represents the depth at the back clipping plane) and the frame buffer is initialised to the background

colour. Then for each polygon the depth of each pixel is calculated. If this is closer than the existing value in the z-buffer, then the z-buffer is updated and the correct colour is written to the frame buffer. The largest z value (which depends on the number of bits used in the z-buffer) is allotted to the front clipping plane. Pseudo-code is shown in Listing 2.

```
 void zBuffer ()
{
    int pz; /* Polygons z at pixel (x,y) */
    for (y = ymin; y <= YMAX; y++) {
       for (x = xmin; x <= XMAX; x++) {
         WritePixel(x,y,BACKGROUND_VALUE);
         WriteZ(x,y,0);
       }
   }
    for (each polygon) {
       for (each pixel in the polygons projection) {
         pz = polygons z value at (x,y);
          if (pz >= ReadZ(x,y)) {
            WritePixel(x,y,polygon colour);
            WriteZ(x,y,pz);
         }
       }
   }
}
```

Listing 2: Pseudocode for the z-buffer algorithm.

To apply this algorithm we do not need to pre-sort the polygons or compare objects, however if the computation of the polygon colour is expensive, then some pre-sorting of the polygons will produce a speed up. The use of the z-buffer algorithm combines scan conversion and visible surface determination. Like scan conversion there are problems due to aliasing, so techniques such as the A-buffer have been developed to reduce this effect.

We can also use depth coherence to our benefit by using iterative methods to compute the depth of pixels which belong to the same object along a given scan line. If the polygon is planar we can write its equation as $ax + by + cz + d = 0$. We can solve this equation for z:

$$z = -\frac{ax + by + d}{c} ,$$

but computing this is expensive. However we can use first order differences. Evaluating z_1 at (x_1, y_1) we can compute z_2 at $(x_2, y_1) = (x_1 + \Delta x, y_1)$:

$$z_2 = z_1 - \frac{a}{c}\Delta x .$$

Since $\Delta x = 1$ and a/c is a constant this is quick to evaluate.

Note that one of the most persuasive arguments for using the z-buffer algorithm is that the image need not be composed of polygons. The only requirement is that we can compute the depth of the object at any point.

7.8 Scan-line algorithms

In a manner similar to the methods used for scan converting polygons we can use active edge tables to determine which polygons are visible along a given scan line. In addition we will need a polygon table which contains the equation of the polygon, colour information and direction (in or out - a Boolean flag). Details of this method can be found in Foley *et al.* (1993, p. 455).

7.9 The depth sort algorithm

Another popular method for visible surface determination is the depth sort algorithm. The algorithm is:

- ➢ sort all polygons by their z coordinate;
- ➢ resolve any ambiguities by splitting polygons that inter-penetrate;
- ➢ scan convert the polygons in order, from the back to the front.

A simplified form, known as **painter's algorithm**, assigns a unique z value to each polygon. This resolves any inter-penetration by not allowing it. This algorithm is best implemented in 2.5D situations, where objects are confined to a number of planes, such as window management, or silicon chip design. The name comes from the fact that this is the method many artists use to paint their pictures.

7.10 Alternative methods

Binary space partition trees use ideas similar to those used in the depth sort algorithm and work at object precision. However when using binary space partitions, we choose and arbitrary root polygon, and then assign all other polygons into two half planes; in front and behind the root polygon (as defined by the polygons surface normal). Any polygons that are in both half planes are split so that they are correctly assigned. One child is chosen from both half planes and the division is repeated. This repeated until there is only one polygon at each node. This tree can be used to determine visibility from any view angle. The polygons are then scan converted from the back to the front. This may not be very useful because pixels that are subsequently obscured will be coloured (which may be expensive). It is possible to start from the front and work back, scan converting only those pixels that have not already been set. Most of the expense of this routine occurs whenever the picture changes, thus this method is most useful when the scene is fixed and only the view changes.

Visible surface ray tracing uses imaginary rays projected from the eyes of the viewer into the scene to determine what parts of the scene are visible. This is an image precision method, since a ray has to be 'fired' through every pixel in the scene. Ray tracing can do much more than simple visible surface determination, but for this section it is sufficient to note than it can be used.

Area subdivision algorithms are closely related to the divide-and-conquer spatial partitioning algorithms. Essentially a scene is partitioned, and one then looks at whether it is easy to decide on the visibility of the objects within the partition. If this is not the case then this region is further partitioned until it is clear. At its limit this method reduces to a simple pixel based image precision method. One variant known as **Warnock's algorithm** is based on the division of space into square regions and testing for the type of inclusion of the polygon in the square. More details are in Foley *et al.* (1993, p. 469).

8 Illumination and Shading

Removing those parts of the image we cannot see is key to producing realistic pictures, but that is only one component of photo-realistic rendering. We also need to model the interaction of light with the objects being modelled. This section considers the treatment of light in computer graphics.

8.1 Light

Some aspects of light are still a mystery. Most people are comfortable with the idea of light as waves, however the quantum revolution has led us to question this model. The alternative is to regard light as a stream of particles, called photons. The quantum paradox means that both views are consistent with what is observed in experiments.

For the purposes of this course, however, we can consider light to be a wave. The wavelength determines the colour of the light, the amplitude the intensity. The visible region of the electro-magnetic spectrum is very small, but it is the part that our eyes are sensitive to. It is no coincidence that it is this part of the spectrum in which the sun emits most energy. The visible spectrum ranges from 400 nm (violet) to 700 nm (red), with the remaining spectrum being filled by the 'colours of the rainbow'.

The way we perceive light is determined by the response function of the rods and cones in the human eye. There is a whole literature on the response of the human eye, and this has bearing on the way we represent and display colour in computer graphics Foley *et al.* (1993, p. 402–).

When light hits a surface it may either be:

- ➢ absorbed (and often remitted at another frequency),
- ➢ reflected (bounced straight back from whence it came),
- ➢ scattered (bounced off the object in many directions),
- ➢ refracted (transmitted through the object, but bending the direction),
- ➢ transmitted (passes right through the object unaltered).

It is not possible for computer graphic applications to accurately model all these processes for all material types, thus the computer graphics treatment of light is characterised by a lot of ad-hoc schemes, which approximate a number of these properties.

Global illumination methods try and account, in a physically based manner, for the interchange of light between all surfaces in the application model, however producing images using such methods may take several hours on some of the faster computers around. **Ray tracing** takes the image precision methods of hidden line removal further by allowing reflection, refraction and approximate scattering of the rays used to determine the surface visibility. **Radiosity** methods attempt to calculate the equilibrium lighting for a given scene, and can thus be viewed from any view point without requiring re-computation.

In this section we briefly review the methods that can be used to simulate the effects of lighting. More information can be found in Chapter 14 of Foley *et al.* (1993). We assume that the light can be represented by an intensity, which does not vary with wavelength.

8.2 Illumination models

The simplest form of illumination is to give each object an illumination property of its own, which does not depend on the angle of the viewer relative to the light source. This is equivalent to assuming no external light source, with the light coming from the objects themselves and with no interaction. This is clearly very unrealistic. We could write this model as:

$$I = k_i \ ,$$

where I is the intensity of the observed light, and k_i is the objects intrinsic emission intensity.

If we now assume an external source of light, but this time a light that has no preferred direction (possibly as a

result of multiple reflections and scattering) we can generate an **ambient light** model. This can be written as:

$$I = I_a k_a \ ,$$

where I_a is the intensity of the ambient light, and k_a is the objects **ambient reflection coefficient**, which takes a value between zero and one.

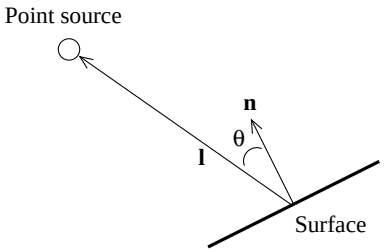


Figure 8.1: Diffuse reflection.

Now let us consider a more realistic situation, that of a point light source. This will produce rays which emanate uniformly from the source. Now the brightness of the object will vary according to its distance from and angle to the light source.

For mat surfaces which scatter the incoming light in all directions (a process which is also called **diffuse reflection** or **Lambertain reflection**) the intensity depends only on the angle the surface normal makes with the light source. It does not depend on the viewing angle. Thus the intensity can be written:

$$I = I_p k_d \cos(\theta) \ ,$$

where I_p is the intensity of the **point light source**, k_d is the objects **diffuse reflection coefficient** (0–1) and θ is the angle between the incident light beam ($\boldsymbol{l}$) and the surface normal ($\boldsymbol{n}$), as shown in Figure 8.1. Note that the angle between the viewer and object does not matter because, as this angle is increased the amount of light reflected decreases, but the area of the surface seen per unit area normal to the viewer increases. For this equation to have any real meaning θ must be between zero and ninety degrees, otherwise the light is coming from behind the object, and unless the object is transparent or translucent we should not illuminate it. Thus we often check that this criterion is met. If the surface normals, $\boldsymbol{l}$ and $\boldsymbol{n}$, have been normalised to have unit length then we can also write:

$$I = I_p k_d (\boldsymbol{n} \cdot \boldsymbol{l}) \ .$$

Note that the lighting equation must be undertaken in world coordinates, prior to any projections.

If the point light source is a long distance from the objects being illuminated (i.e. like the sun) then $\boldsymbol{l}$ will be constant, and the source is termed a **directional light source** and can be represented in homogeneous coordinates as a point at infinity ($w = 0$).

Combining the effect of ambient lighting and diffuse reflection yields:

$$I = I_a k_a + I_p k_d (\boldsymbol{n} \cdot \boldsymbol{l}) \ .$$

If we are trying to model light sources accurately then we should account for the affects of light attenuation. This allows us to account for the fact that a light source or an illuminated object looks dimmer the further they are from the viewer. This can be modelled using an attenuation factor, f_{att}, to give:

$$I = I_a k_a + f_{att} I_p k_d (\boldsymbol{n} \cdot \boldsymbol{l}) \ ,$$

and we can write:

$$f_{att} = \frac{1}{d_l^2} \ ,$$

where d_l is the distance to the light source. Although this is a good model for attenuation, it can be improved by using a more complex function to represent the decrease in brightness as objects get further from the light source Foley *et al.* (1993, p. 482).

We might also like to include the effect of the distance of the object from the viewer, called **atmospheric attenuation** or **depth cueing**, so that objects further from the viewer (for which we can use the z coordinate in the projected canonical view volume as an approximator) appear dimmer.

8.3 Coloured light

When we move on to coloured light the most simple thing we can do is to representing the final colour as a combination of intensities of each base colour (e.g. the RGB colour model), with objects having different reflective properties for each base colour. This approximates the true situation.

8.4 Specular reflection

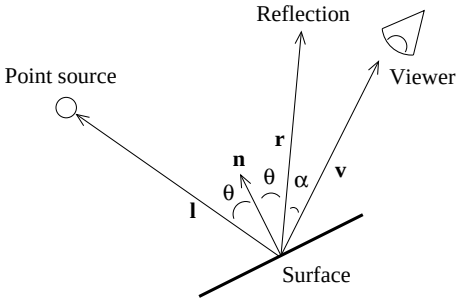


Figure 8.2: Specular reflection.

Specular reflection is what is observed when a bright light is incident on a 'shiny' surface. The colour of the reflection is governed by the colour of the incident light, thus these are most often seen as the white (bright) patches on objects. If the object were a perfect mirror then we would only see the reflection of light coming in at an angle equal to the viewing angle (Figure 8.2). Thus specular reflection varies with the viewing angle.

A commonly used model, called the **Phong illumination model**, applies to imperfect mirrors (which covers most objects we would model) which reflect not just at the reflection vector $\boldsymbol{r}$, but also within a small angle, α, about that reflection vector, as shown in Figure 8.2. In this figure $\boldsymbol{v}$ is used to shown the direction of the viewer. The Phong model says that the amount of light reflected is equal to $\cos^n(\alpha)$. This can be added to the equation for the intensity:

$$I = I_a k_a + f_{att} I_p \left(k_d (\boldsymbol{n} \cdot \boldsymbol{l}) + k_s (\boldsymbol{r} \cdot \boldsymbol{v})^n \right) \ ,$$

where k_s, the specular reflection coefficient is generally set using aesthetics to produce pleasing results. Choosing the power n is also arbitrary and can be set between 3 and 200. This is a classic example of an empirical model, rather than a physical one.

Coping with multiple light sources is relatively simple, we simply need sum the diffuse and specular terms for each light source.

8.5 Shading polygons and polygon meshes

When shading a polygon we could simply compute the surface normal for each polygon and shade the polygon appropriately. This ignores the effect of changes in distance from the light source and viewer on the polygon shading, and is known as the **constant shading** or **flat shading** model. This is generally rather unrealistic, so this model is not widely used. It will produce very flat looking surfaces which have constant shading on any one surface.

Using **interpolated shading** the shading of a polygon is computed at each of it's vertices and the shading for any pixel within the polygon is then computed as an interpolation of the vertex shadings. This means that the shading can vary realistically across the face of the polygon at the expense of computing a weighted average (interpolant) of the vertex shadings.

When shading a polygon mesh we need to take into account not just the intensity within the polygon but also across polygons, particularly if we are using a polygon mesh to represent a curved object. **Gouraud shading** extends interpolated shading to interpolate across neighbouring polygons, providing a smooth change in intensity at polygon boundaries. Essentially a normal is computed at each vertex of the polygon mesh (generally by averaging the normals of the polygons that share the vertex) and the shading is computed using these normals and then interpolated across the object. This shading algorithm is often efficiently integrated with a visible surface algorithm.

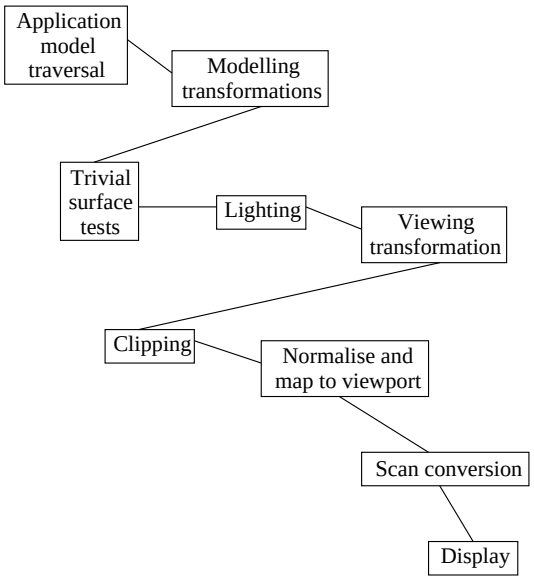


Figure 8.3: The rendering pipe line using Gouraud shading.

Phong shading is similar to Gouraud shading but this time interpolates the surface normals (which are then used in the shading calculations) rather than the intensities themselves. This model is more accurate but also more expensive than Gouraud shading.

8.6 Rendering pipelines

The illumination models we have discussed form part of the rendering pipeline, such as was shown earlier in the notes. Two different versions of the full rendering pipeline are shown in Figure 8.3 and Figure 8.4. Figure 8.3 shows the rendering pipeline that would be used with z-buffer hidden surface removal and Gouraud shading. In this case the illumination is computed at each vertex in world coordinates. The rasterisation step (based on the z-buffer algorithm) includes the implementation of scan conversion, the interpolation of the vertex intensities and possibly any depth cueing.

Figure 8.4 shows how the rendering pipeline is implemented for Phong shading. Since the lighting is based on interpolated normals, and this is carried out in the rasterisation process (again implemented by the z-buffer algorithm), we need to be able to map the vertices (and normals) back into world coordinate space so that when they are interpolated the lighting model is computed correctly. This is clearly a more expensive scheme to implement.

This is as complex as the graphics we consider in this course will get. The final rendering step brings together pretty much everything we have discussed in these notes.

8.7 Limitations

So far we have discussed only point light sources. Methods exist to accommodate directional light sources (see Foley *et al.* (1993, p. 487) for example) but we shall not consider them here. We have also not considered physically based lighting models, since these are beyond the scope of this course. We have not considered shadows either, so it is clear there is a lot more complexity to realistic rendering. This is discussed in the next section.

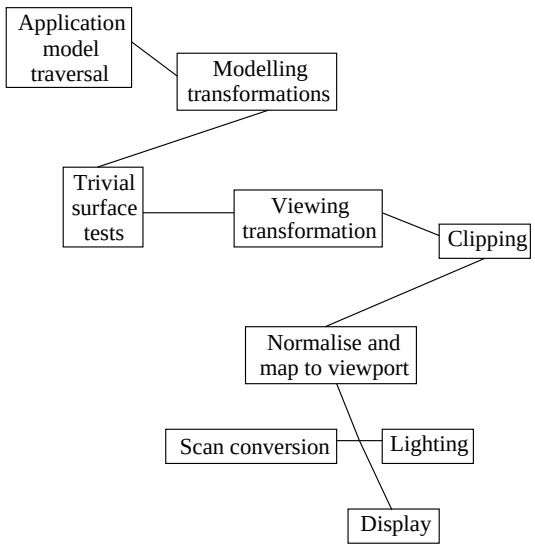


Figure 8.4: The rendering pipe line using Phong shading.

9 Visual Realism

In the previous sections we have dealt with the basic methods we can use to represent 2D and 3D graphical objects, but in most of the examples we have used rather straight forward models, such as lines, polyhedra or simple curves. To generate realistic images, such as those used in special effects requires a number of extra 'tricks' to fool us into believing we are looking at the real objects.

Of course we should also ask ourselves why we might want to produce this visual realism.

The most obvious trick that is used is to make the scene consist of a very large number of polygons, so that the sheer complexity overwhelms us. This works reasonably well for non-natural objects, but other methods must be used for natural objects.

Using visible surface algorithms (typically implemented as z-buffer type methods) and good illumination algorithms (particularly ray tracing) can give very realistic pictures.

There are however a number of problems with rendering a visually realistic picture:

- ➢ textures – skin, brick-work, grass;
- ➢ subtle colour variations – trees, brick-work, the sky;
- ➢ shadows;
- ➢ reflections – metallic objects, glass;
- ➢ irregularities – grass, desk tops, clouds.

To store all of this for a graphics program to reproduce would take too much resource, but without them the scene we produce still looks artificial.

Other methods we can use to improve the visual realism include:

- ➢ anti-aliasing – this is so important in producing realistic pictures that it is implemented automatically in almost all graphics cards and libraries.
- ➢ perspective projections;
- ➢ **texture mapping** – this can get very sophisticated - we essentially map bitmapped textures onto the visible surfaces;
- ➢ **bump mapping** – often used with texture mapping, bump mapping takes a step further by mapping irregularities (not just bitmaps) onto polygons so that their appearance changes with lighting angle;
- ➢ colour – for instance objects in the distance tend to look more blue;
- ➢ material properties – the amount and type of light reflected, refracted and transmitted by objects can be represented at various levels of sophistication;
- ➢ the use of curves and surfaces – where appropriate these can look a whole lot better than a approximate mesh representation;
- ➢ better light source modelling – for instance strip lighting does not produce sharp shadows;
- ➢ improved camera (or optical system) models – both our eyes and photographs do not have an infinite depth of field. We can simulate parts being out of focus, and the motion blur that is due to the camera having a finite shutter time;
- ➢ **depth cueing** – lowering the intensity of the image, as a function of distance from the viewer helps to enhance the impression of depth. The reason this works is that we are using a very simple model for atmospheric attenuation.

A major change we can foresee in computer graphics is the use of a technique known as **stereopsis** or **stereo vision**. Rather than projecting one 2D version of a 3D object or scene, we project two 2D versions, one for each eye, accounting for the different positions of the eyes. This technology is key in immersive Virtual Reality (VR) applications.

The images for the left and right eyes may be displayed in different colours and then a filter (coloured plastic!) used to ensure each eye only sees the relevant information (remember those old red and green 3D pictures?). This precludes the use of colour. A better solution is to present separate colour images to each eye using for instance VR goggles. This requires a very small, high resolution display which should be very light weight. A method that may well become popular is the direct projection of the two images into the eye. Although technically difficult (and potentially dangerous) such systems already exist and are being further developed. A good match between both images is crucial to tricking the brain into seeing the 3D images (remember those 3D pictures, and how difficult they were to see).

We now go back to the very low level and take a look at how hardware is used the render the images we have created using the methods we have studied so far.

10 Conceptual Models for Computer Graphics

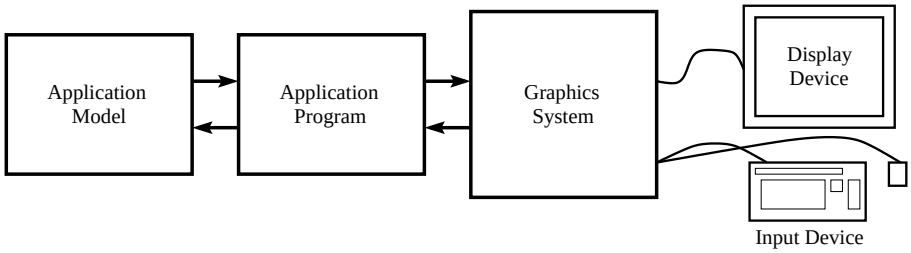


Figure 10.1: A conceptual framework for (interactive) computer graphics.

The model shown in Figure 10.1 can be used to describe pretty much any graphics system. The hardware part of the system is shown symbolically, and essentially involves the reception of input from interactive devices and the output of the images to the display device.

The software part has three components: the application program handles the exchange of data between the application model and the graphics system. The application model represents the data or objects to be visualised on the display device. The graphics system produces the output to drive the display device, given a series of graphics output commands which give geometrical information on what is to be viewed and attributes describing how the object is to be viewed.

As well as producing this output transformation from objects in the application model to objects on the display device, the graphics system performs the inverse operation for the input devices. Thus the graphics system also handles the input transformations from user actions to the application program, which consequently changes the application model. This is the essence of interactive graphics.

The design of interactive graphics application programs centres around the definition of the data items and objects which can be represented on the display device and how the application program can interact with these items and objects to modify the application model (and thus what is seen on the display device). Thus the greatest effort in programming involves defining and implementing the application model and handling user interaction. The graphics system will usually already be implemented via some form of API, so creating the output on the display system is of lesser concern, although it is important for you to understand how this is done.

10.1 Application models

The form that the application model takes will depend on the aim of the application program. For drawing programs, the application model is typically the displayed bitmap, although the structure in Figure 10.1 remains, it is the application programs job to handle user interaction. In more general programs the application model will depend on the type of program. For instance a spreadsheet will store the application model in the application database - which will typically be a particular array structure. The application program will then have at least two graphical aspects: the display of the primary data (i.e. the spreadsheet itself) and graph based display of the data therein - which will typically have its own additional application model. The application model should ideally be independent of the actually display device.

The application models we shall consider, are largely those which store graphics primitives, such as points, lines, curves, polygons (2D or 3D) and polyhedra and surfaces (3D) which define the geometric properties of the objects, together with attributes, such as colour, style and texture. The connectivity relations might also be stored, which define how the primitives fit together (i.e. their topology).

10.2 Displaying the application model

It is the role of the application program to convert the data in the application model to commands used in the graphics system to produce a view of the application model. Typically the application program will also create

the application model interactively, with the user specifying the particulars of the model (e.g. the generation of a graph in a spreadsheet). In some domains, particularly scientific computing the application model may result from many hours of prior, predefined computation.

The application program must translate between the internal representation of the application model (which is not necessarily a geometrical representation of the view), to something that can be understood by the graphics system. It may either do this by creating and storing a geometrical representation, or do it on the fly as it is needed.

This conversion (or translation) process will typically have two phases. First the application program queries the application database to extract those parts of the application model required for the desired view. This data is then converted into a geometrical description (if necessary) and sent to the graphics system. Typically this means the data is sent in the form of graphics primitives.

The job of the graphics system is to convert these output primitives from the application model to commands that drive the output display device. Thus the graphics system typically consists of a graphics subroutine library or API which can be called from some high level language such as C or Ada. The functions within the library generally consist of commands to draw the various graphics primitives, and these will generally follow some standard specification, such as OpenGL or Direct 3D. To shield the developer from the intricacies of the graphics and input hardware the graphics system creates what is known as a logical display device.

The input devices are typically event driven, that is an interrupt is generated when an input device is employed by the user, rather than sample driven - where the application continually checks the device.

10.3 Graphics systems

The primary job of the graphics system is to manage input and output between the user and the application program. From a *computer graphics* point of the view the most important role is the display of the data in the application model. Before we discuss the architecture of the graphics system, we take a brief look at display technology, since this impacts many of the design decisions in the graphics system.

11 Graphics Hardware Issues

In order to program graphical routines sensibly it is necessary to have a firm understanding of the hardware issues involved. This section gives a brief introduction to the simple computer graphics hardware.

11.1 Video display devices

The Cathode Ray Tube (CRT) is the standard mechanism for the display of computer information. It is the same technology that is present in standard television sets.

The CRT tube works by emitting electrodes from the electron gun and accelerating and focusing these onto a phosphor coated screen (see Figure 11.1). The electrons are continually 'boiled off' the cathode by the heating filament. Inside the vacuum within the CRT envelope the electrons (which are negatively charged) are accelerated and focused using anodes (that is positively charged objects). Actual details of the exact implementations vary across systems, this gives the general idea.

The horizontal and vertical deflection plates generate an electric field which is used to deflect the focused electron beam onto a particular phosphor dot[4]. In simple systems a phosphor dot is either on or off, however most modern systems can control the intensity of the electron beam using the control grid, which gives the possibility of grey-scale images.

When the focused electron beam hits the phosphor coated screen, the energy excites the phosphor atoms. This

[4]This applies to electrostatic deflection systems, which are in common use. In a magnetic deflection system, which can give better focusing at a higher cost, magnetic fields are used.

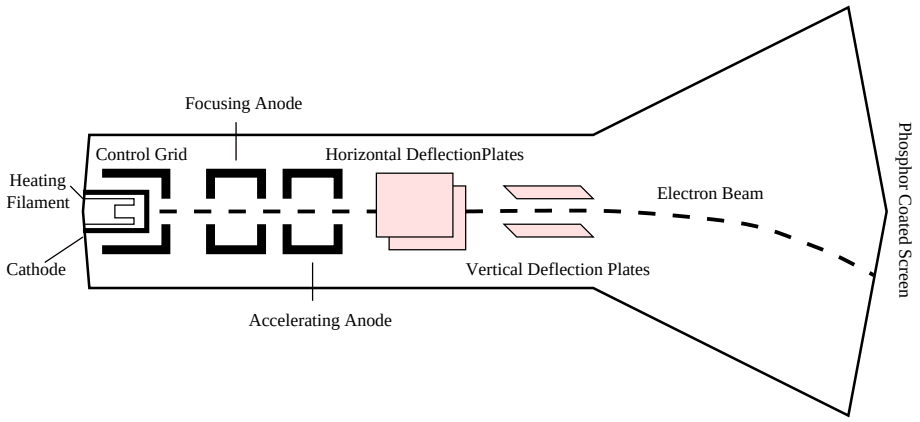


Figure 11.1: A simplified view of a CRT system.

raises the energy levels of the phosphor atom electrons, which emit light energy (photons) as they return to their base states. The length of time it takes for the light energy to fall to one tenth of the original intensity is known as the **persistence**. Phosphors with low persistence are useful for animation, but require higher refresh rates for flicker free display. Although phosphors exist with persistence times of the order of a second, most commercial systems use phosphors with a persistence time of between 10 and 60 μs. A μs, pronounced microsecond, is a thousandth of a second.

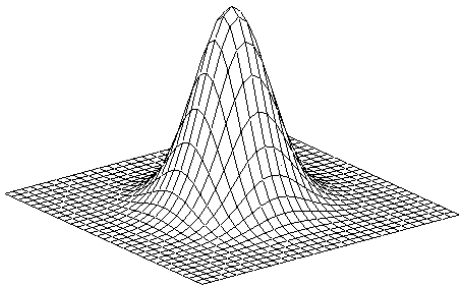


Figure 11.2: The distribution of light intensity at an individual phosphor dot.

The light emitted from a phosphor dot, has a certain spatial support (that is it is not truly a point). This support is Gaussian in shape as shown in Figure 11.2, thus the dots will appear to be separate if the centres of the dots are separated by approximately two times the distance at which the intensity drops to 60%. The maximum number of points that can be displayed without overlap is termed the **resolution** of the CRT. The resolution of the CRT changes with intensity of the dot, and depends on the type of phosphor used, the intensity to be displayed and focusing and deflection systems.

Also important is the aspect ratio of the screen, that is for a given length of line on the screen what is the required ratio of vertical : horizontal pixels. Thus an aspect ratio of 3/4 implies that to represent a given length of line three vertical pixels are need as opposed to four pixels along the horizontal.

11.1.1 Colour

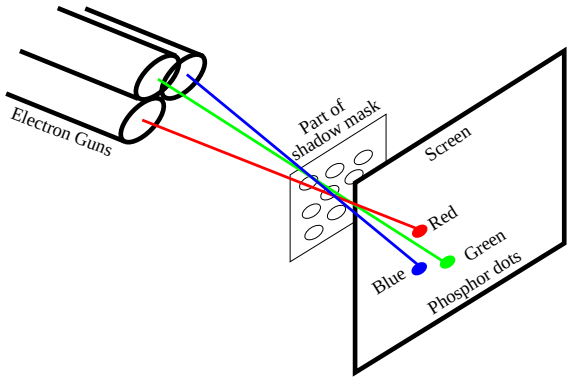


Figure 11.3: The simplified idea of using three electron beams to generate a colour display, using a delta-delta, shadow mask CRT.

In video display devices the **additive** colour model is used, because coloured light combines additively – for instance $red + green = yellow$, whereas in the **subtractive** colour model (such as when mixing paints) we get $red + green = brown$. On a standard CRT, colour is obtained by having 3 separate electron beams which pass through a **shadow mask** which ensures each beam is focused on one phosphor dot of the correct colour (Figure 11.3). Phosphors that emit at Red, Green and Blue (RGB) wavelengths are used, with the dots either grouped in triangular clusters (delta-delta) of the three colours, or in lines (inline). More details can be found in Foley *et al.* (1993, p. 137) or Hearn and Baker (1994, p. 43).

High quality monitors usually use a delta-delta, shadow mask, with each RGB electron beam being able to be set at one of 256 intensity levels. Thus with 8 bits of colour depth per beam, a total colour depth of 24 bits per pixel is available, which is referred to as a **true colour** system. To be of any use this must also apply to the graphics system hardware.

Some manufacturers are now producing flat-panel CRT's where the electrons are accelerated parallel to the display screen and then turned 90° to hit the phosphor. These are difficult to build but offer the prospect of much lighter and thinner CRT monitors.

11.1.2 Liquid crystal displays

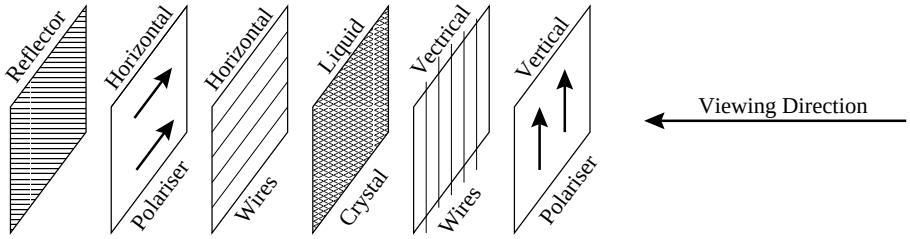


Figure 11.4: The liquid crystal display.

The increased demand for mobile computing drove the development of Liquid Crystal Displays (LCDs). These work by having a material (the liquid crystal) whose properties, with respect to the transmission of polarised light, can be changed by applying an electrical potential. Today, concerns about the impact of CRT screens on users eyesight, and desk space requirements mean LCD displays are coming onto the desktop too.

Older LCD displays worked by using polarised light, which is vertically polarised at the front of the display, is rotated by 90° by the long crystalline molecules in the liquid crystal, so that it is now horizontally polarised, thus the light passes through the rear horizontal polariser (see Figure 11.4). This is reflected by some mirror and passes back through the system. However when a charge is applied to the liquid crystals they straighten and line up. This means they no longer rotate the polarisation of the light, and thus no light is transmitted through the horizontal polariser, causing a black spot.

Rather like CRTs, LCDs need to be refreshed, when the older wire technology (unlike CRTs, liquid crystals have a rather longer persistence of several hundred milliseconds) is used. The sheer number of pixels that needed to be refreshed meant that there was an increasing lack of contrast as the resolution increased. Due to the polarised nature of the reflected light, viewing could be awkward - this can be improved by using some form of back-lighting, such as is generally done today, using flat electro-luminescent strips.

Most modern LCDs use **active matrix panels**, where Thin Film Transistor (TFT) technology is used to create tiny transistors at each pixel locations and these can be used to determine the degree of alignment of the liquid crystals. Thus these devices allow gray scale images, and by adding dye, it is possible to create colour displays. Since the transistors can easily be controlled independently these can act as a memory and thus the screen no longer needs to be refreshed (unless the picture changes) and a brighter image results. TFT displays have traditionally been expensive, because each pixel actually requires three transistor switches – one to activate each primary colour – which makes the LCD panels very complex to manufacture.

One problem with LCD displays is that they have a native resolution, which is defined by the number of pixels in the matrix. Working at lower resolutions than in the matrix results in a loss of clarity and image distortion. although some manufacturers have developed hardware workarounds.

11.1.3 Alternative devices

A number of alternative devices have been devised for video display, such as plasma panels, and projector devices, however these will not be dealt with here. More details can be found in most graphics text books.

11.2 Hard-copy devices

Video devices are great for transient images (such as animation, video clips, etc.) but are not suited to the long term display of images. For these we require hard-copy devices, that should accurately and efficiently reproduce the look of the object on the screen (where it will typically have been created) on paper (or some other substance).

For black and white images such as text or line drawings, many possible technologies can be used to produce the output. Since the output devices work very similarly to raster video devices, similar problems must be addressed, particularly questions of resolution and anti-aliasing. For more information on graphical hard-copy devices consult a standard graphics textbook.

For colour devices there is an additional problem, that of colour matching. While video displays use the additive RGB colour model, hard-copy devices tend to use the subtractive Cyan, Magenta, Yellow (CMY) colour model. Conversion between these in true colour (24 bit) mode is non-trivial.

11.3 SI units

This is just a brief reminder of the definition of SI units, just in case you can recall your micro's from your milli's.

Unit	Numerically	English (US)	Standard symbol
tera	10^{12}	trillion	T
giga	10^{9}	billion	G
mega	10^{6}	million	M
kilo	10^{3}	thousand	K
mili	10^{-3}	thousandth	m
micro	10^{-6}	millionth	μ
nano	10^{-9}	billionth	n
pico	10^{-12}	trillionth	p

Beware some people use a trillion to mean 10^{18} which is the old English meaning.

11.4 Graphics system hardware

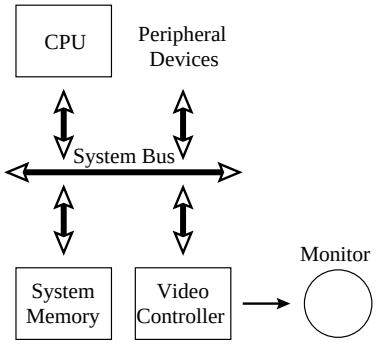


Figure 11.5: A simple graphics system.

This section discusses the various elements of a raster graphics system. Most raster displays will have a **graphics display processor** (often also referred to as the **graphics controller** and **graphics processing unit**) to aid the conversion of output primitives into a pixmap and also for block operations on the pixmap (e.g. moving, copying, etc.). The greatest difference between different display systems relates to the amount of processing done by the **graphics display processor** as opposed to processing performed on the main CPU running a graphics subroutine library.

The simplest type of graphics system, which has no dedicated graphics display processor, and no internal memory is shown in Figure 11.5. The system memory is used to store the pixmap, and the system CPU is used (with some graphics subroutine library) to generate the pixmap from the graphics primitives. Typically to speed up transfers between system memory and the video controller, a dedicated bus from a portion of the system memory where the pixmap is stored, called the **frame buffer**, to the video controller is used.

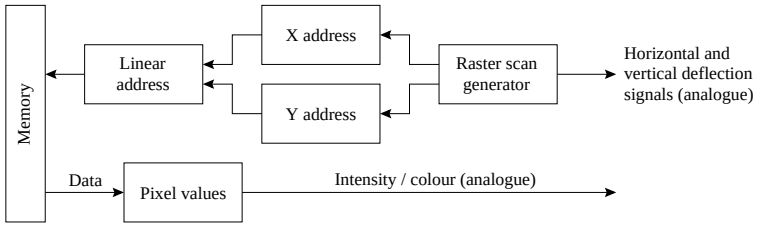


Figure 11.6: The layout of a typical video controller.

A typical video controller is shown in Figure 11.6. The operation is relatively simple. The raster scan generator is activated each time the screen is refreshed. It does two things: first it controls the X and Y addresses registers (which give the x and y coordinates of the pixel) and, secondly, it generates the correct vertical and horizontal deflection signals. The X and Y address register results are combined by the linear address calculator, which then accesses the relevant part of memory, for the given pixel. Data on the intensity or colour of the pixel is then converted to the value needed by the (analogue) display system.

In order to refresh a display of 1024×768 pixels at a moderate refresh rate of 60 Hz requires a memory access every $1/(1024 * 768 * 60)$ seconds, or 21 ns (see Table 11.3 for information on SI units). Since most older system RAM had access times of the order of 70 ns, multiple bits were retrieved at the same time. This would still put a great deal of extra strain on the system bus, meaning that the CPU could not access system memory every cycle.

For general colour systems each pixel may have between 8 and 32 bits of information, depending on the colour depth needed for the particular application. In order to flexibly convert the n bits of colour information to the correct settings for the red, green and blue colour components the video controller often uses a **Look Up Table** (LUT). This ascribes the correct colour information to a given bit pattern in the pixmap.

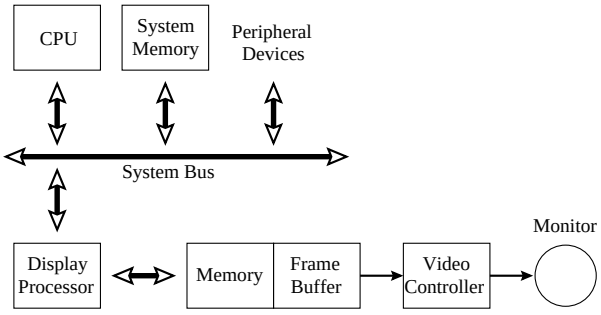


Figure 11.7: A more complex raster display system with a dedicated display processor, known as a **peripheral display processor**.

The simple model of the graphics system (Figure 11.5) is no longer widely used. Instead most graphics systems now possess dedicated graphics display processors. These cut down the load on the CPU and the system memory since such designs also often incorporate their own graphics processor memory (as shown). This memory is often Video RAM (VRAM) where a complete scan line of pixels is read at each refresh cycle, or is some other form of fast memory.

The graphics processor generally performs scan line conversions and raster operations (such as pixel copy, move etc.). Many of the new generation of graphics cards also have hardware support for the common graphics libraries such as OpenGL and Direct 3D. The CPU and graphics processor typically communicate via a queueing system, which allows the CPU to get on with other jobs while the graphics processor handles the display, particularly when the graphics hardware has a large amount of memory in which it can store display primitives, textures and other data.

There is a danger in loading ever more work onto the graphics processor, possibly allowing it to handle interaction between the screen and a pointer device, which is known as the **wheel of reincarnation**. As early as 1968 it was recognised that there is a tradeoff between specialisation and general purpose functionality. Specialisation means the hardware can do the job more quickly than a general purpose design, however specialised designs cost more and cannot be used for other purposes. This is as true in computer graphics systems as anywhere. The means by which we can confront this issue is using widely agreed standards.

The dedicated graphics processor may also store details of the image, in terms of a hierarchical data structure, known as a **display storage list**. This list will consist of primitives or combinations of primitives (segments), which have unclipped integer coordinates and are stored locally in the display processor memory. These segments can then be redrawn, zoomed, scrolled, dragged, etc. without the need for any action by the CPU.

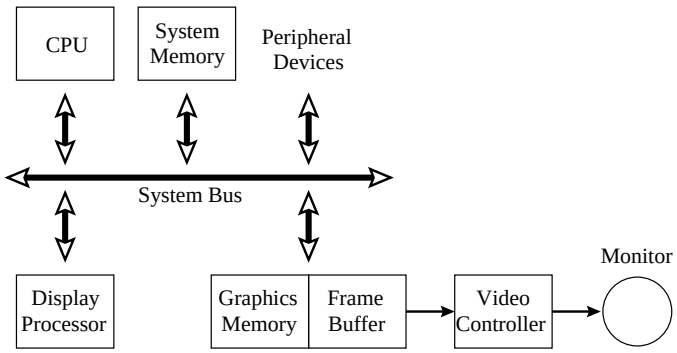


Figure 11.8: A raster display system with a dedicated display processor, known as an **integrated display processor**.

The final widely used alternative to the **peripheral display processor** architecture is the **integrated display processor**, also known as the **Single Address Space** (SAS) display system architecture (Figure 11.8). Here the CPU, the display processor and the video controller are on the system bus and can access either the system or graphics memory. In some systems there is no distinction between the system and graphics memory, and in any case the two are addressed identically.

There are advantages and disadvantages to both peripheral and integrated systems. Integrated systems may be slower due to the relatively low speeds of the system bus and the greater demands placed on it, however from a programming point of view they are much neater, since all the memory can be addressed from both processors. This avoids conflicts which might occur in having different transfers between system and graphics memory.

11.5 Input devices

This course does not consider input devices in any great detail. There are a wide range of devices available, from the common mouse, through scanners and graphics tables to laser imaging devices for 3D objects. Both Foley *et al.* (1993) and Hearn and Baker (1994) cover input devices.

12 Basic Raster Algorithms for 2D Graphics

A raster graphics package approximates the graphical primitives (points, lines and polygons) described in a 2D Cartesian coordinate system, by setting pixels in the pixmap to the appropriate intensities or colours. The conversion from primitives to pixmap is generally known as scan conversion. This section deals with the fundamental algorithms that are used in performing such operations. In general such algorithms will be implemented on the display processor, although some older systems, without a display processor may still use software on the CPU.

12.1 Scan converting lines

The most simple (non-trivial) scan conversion task, is that of scan converting a line. The scan conversion routine should compute those pixels which lie on or near to an ideal straight line imposed on the pixmap (a raster grid), as in Figure 12.1. Theoretically, any scan conversion algorithm should produce a line with constant brightness (independent of orientation and length) and do it quickly. It should also be able to cope with lines greater than one pixel in width and be able to display different styles and attributes. The end points may need to be bevelled (i.e. come to a point), rounded or mitred (asymmetric bevel). It is also necessary to minimise the jaggies on the line, by using anti-aliasing methods to set pixel intensities.

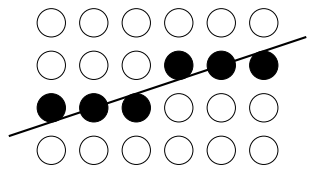


Figure 12.1: Scan converting a line.

Assume that pixels are disjoint circles (representing the phosphor dot on the screen) on an integer, (x, y) grid, and that the line starts and ends at integer coordinates (x_0, y_0) and (x_e, y_e) respectively. The simplest algorithm to scan convert the line is an incremental one. First, compute the slope $m = \Delta y/\Delta x$, where $\Delta y = y_1 - y_e$. Now start at the leftmost point and increment x by 1 each time calculating $y_i = mx_i + c$, where c is now the offset (the value of y when $x = 0$). Set the intensity at the pixel value $(x_i, \text{Round}(y_i))$, where $\text{Round}(y_i) = \text{Floor}(y_i + 0.5)$. This selects the pixel that is closest to the true line, but is very inefficient because each pixel needs a floating point multiply, an addition and a call to Floor. However noting that $y_{i+1} = mx_{i+1} + c = m(x_i + \Delta x) + c = y_i + m\Delta x$ and that Δx in this case will be 1, gives:

$$x_{i+1} = x_i + 1 \,,$$
$$y_{i+1} = y_i + m \,.$$

This is a more efficient algorithm, which works so long as $|m| < 1$. If this is not the case, it is necessary to swap x and y, which will give a slope of $1/m$. It is also necessary to check for the special conditions of horizontal, vertical and diagonal lines.

```
 void Line ( int x0,  int xe,  int y0,  int ye,  int value)
{          /* Assumes -1 <= m <= 1, x0 < xe */
    int x;
    float y,dx,dy,m;

   dx = xe - x0;
   dy = ye - y0;
   m = dy / dx;
   y = y0;

    for (x = x0; x <= xe; x++){
      WritePixel(x,(int) floor(y + 0.5),value);
      y += m;
   }
}
```

Listing 3: Pseudocode for the incremental line scan conversion algorithm.

The simple pseudocode for the algorithm is shown in Listing 3. The algorithm is often referred to as the **digital differential analyser** due to it's similarity to finite difference techniques used to solve discretised differential

equations. One potential problem with the method arises because the float m has a finite precision. Repeatedly adding m builds up the error due to precision, but this is not generally a problem for short lines, such as those displayed on a CRT.

12.1.1 Midpoint Line Algorithm

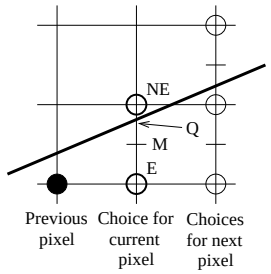


Figure 12.2: The midpoint schema.

A more advanced algorithm is the **midpoint line algorithm**, which does not use floating point computation (which is computationally expensive). The midpoint line algorithm is a generalisation of Bresenham's well known incremental technique, and uses only integer arithmetic. Furthermore, the algorithm works only for $0 \leq m \leq 1$, other slopes being catered for by reflection about the principal axes of the 2D plane. Now (x_0, y_0) will be the lower left endpoint and (x_e, y_e) the upper right endpoint.

The method is incremental, like the last on, but this time, due to the restrictions on the slope of the line it is known that if we are at some pixel (x_p, y_p), we now need to choose between only two pixels (see Figure 12.2). Either the east pixel, E, or the northeast pixel, NE, is chosen depending upon which side of the midpoint, M, the crossing point, Q, lies. If we represent the line by an implicit function $f(x, y) = ax + by + \gamma = 0$, then recalling that the slope-intercept explicit form is:

$$y = mx + c = \frac{\Delta y}{\Delta x} x + c \; ,$$

we can write:

$$f(x, y) = \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot c \; ,$$

and thus $a = \Delta y$, $b = -\Delta x$ and $\gamma = \Delta x \cdot c$. For any point on the line, $f(x, y)$ is zero, any point above the line will have $f(x, y)$ negative and any point below has $f(x, y)$ positive. Applying the midpoint criterion means computing $d = f(\mathrm{M}) = f(x_p + 1, y_p + 1/2) = a(x_p + 1) + b(y_p + 1/2) + \gamma$. If d is positive we choose NE, otherwise we pick E (this defines the convention, for the case when $d = 0$).

So what happens to the location of the next midpoint, M_{new}? This depends on whether E or NE was chosen. If E is chosen then the new d_{new} will be:

$$d_{new} = f(\mathrm{M}_{new}) = f(x_p + 2, y_p + 1/2) = a(x_p + 2) + b(y_p + 1/2) + \gamma \; .$$

Comparing this with the old d shows that $d_{new} = d_{old} + a$. Thus a is the increment to add after E is chosen, denoted Δ_E. If NE is chosen then both x and y are incremented, and it is left as a simple exercise to show that $\Delta_{NE} = a + b$. This is repeated incrementally.

Starting from (x_0, y_0), the first d is:

$$d = f(x_0 + 1, y_0 + 1/2) = a(x_0 + 1) + b(y_0 + 1/2) + \gamma = f(x_0, y_0) + a + \frac{b}{2} \; .$$

Since $f(x_0, y_0)$ is on the line, this will be zero and $d = a + b/2 = \Delta y - \Delta x/2$. We already know the increments to add, but we still have the rather annoying factor of $b/2$. We can remove this simply by using $d = 2f(x, y)$, which will not affect the sign of the decision variable and keep everything integer. This means the algorithm uses one addition and a test of a single bit (positive or negative), which is much faster than the multiplication required in the previous incremental algorithm.

The pseudocode for the midpoint algorithm is shown in Listing 4, and can be seen to be very simple and use only integer arithmetic. There are several improvements that could be envisaged to the midpoint algorithm. These are outlined in Foley *et al.* (1993, p. 76), and can give significant speed ups. One method involves looking ahead two pixels at a time (so called double-step algorithm) and another uses the symmetry about the midpoint of the whole line, which allows both ends to be scan converted simultaneously.

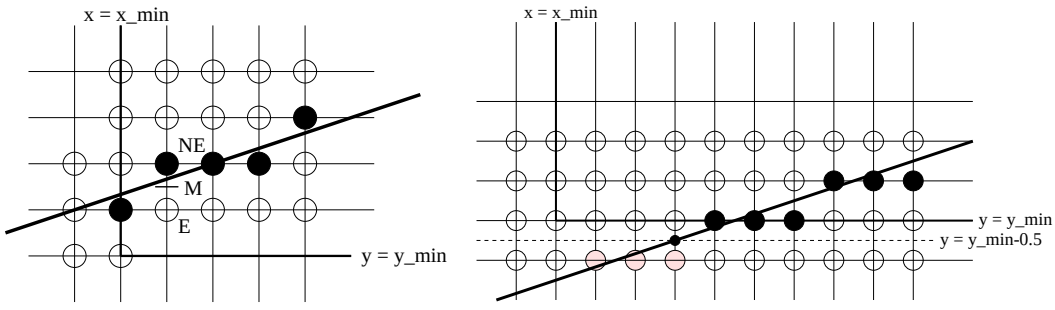
Additional issues arise in the scan conversion of lines. One is that lines should look the same regardless of whether they were described as lines from p_0 to p_e or p_e to p_0, that is their end point order. This will only affect the midpoint algorithm during the part where the algorithm defines that E is chosen when Q = M (Figure 12.2). In this case, making the algorithm choose SW rather than W (etc.) in the inverted version will ensure independence of end point order. Extra care needs to be taken when also using line styles (Foley *et al.*, 1993).

```
void MidpointLine ( int x0, int xe, int y0, int ye, int value)
{          /* Assumes 0 <= m <= 1, x0 < xe, y0 < ye */
    int x,y,dx,dy,d,incE,incNE;

   dx = xe - x0;
   dy = ye - y0;
   d = 2*dy - dx;
   incE = 2*dy;
   incNE = 2*(dy-dx);
   x = x0;
   y = y0;
   WritePixel(x,y,value);
    while (x < xe) {
       if (d <= 0) {
          d += incE;
          x++;
      }  else {
          d += incNE;
          x++;
          y++;
      }
      WritePixel(x,y,value);
   }
}
```

Listing 4: Pseudocode for the midpoint line scan conversion algorithm.

12.1.2 Line clipping



(a) Vertical (b) Horizontal

Figure 12.3: Illustration of the problems of clipping lines with rectangular regions.

It is also common to clip a line by a bounding rectangle (often the virtual or real screen boundaries). To do this properly it is necessary to consider two cases, that when the line intersects a vertical of the clipping rectangle and a horizontal of the clipping rectangle.

Assume the bounding rectangle has coordinates, (x_{min}, y_{min}), (x_{max}, y_{max}). If the line intersects the left hand vertical edge, $x = x_{min}$, the situation is as shown in Figure 12.3(a). The intersection point of the line with the boundary is $(x_{min}, (m \cdot x_{min} + c))$, however this is not at a pixel, hence we need to start the line from $(x_{min}, \text{Round}(m \cdot x_{min} + c))$, which is the same pixel which would have been selected by the incremental line algorithm (or the midpoint algorithm). Note that we round down to ensure that E is chosen rather than NE, for the case that the intersection falls exactly on the midpoint. Then the standard midpoint algorithm, starting from the pixel $(x_{min}, \text{Round}(m \cdot x_{min} + c))$ can be used. The algorithm must be applied this way, it would not be correct to scan convert the clipped line from $(x_{min}, \text{Round}(m \cdot x_{min} + c))$ to (x_e, y_e).

If the line crosses the lower horizontal boundary, a different strategy is required. We assume that any of the lines pixels falling on or inside the clip region are drawn. Thus it is necessary to determine the point at which the first pixel on the boundary should be set. This is not at the point $((y_{min} - c)/m, y_{min})$ where the line crosses the bounding line, nor always at $(\text{Round}((y_{min} - c)/m), y_{min})$. Rather we determine the leftmost pixel, by computing the pixel just above and and to the right of the intersection of the line with $y = y_{min} - 0.5$, thus the first pixel is:

$$\left(\text{Round}\left(\frac{(y_{min} - 0.5 - c)}{m}\right), y_{min}\right) .$$

The standard midpoint line algorithm can be used, from that starting point, taking note that negative values always means E is chosen.

12.1.3 Additional issues for lines

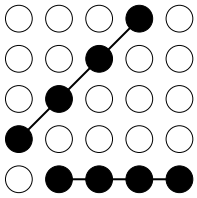


Figure 12.4: Line intensity problems.

Lines of different slopes often will have different intensities on the display, unless care is taken. Figure 12.4 shows two different lines, both made up of four pixels, but the diagonal line is $\sqrt{2}$ times as long as the horizontal line. The intensity per unit length is therefore different and this will be detected by the eye. Thus on systems where there are n bits per pixel the intensity can be set as a function of the line slope.

Knowing how to scan convert straight lines is not the whole story. Polylines (lines made up of several straight line segments) require special attention to the end and start points. If the object is being written in **xor** mode then setting the pixel twice (at the end and start of joining lines) will cause the wrong colour (intensity) to be displayed. This may also be true for shared pixels of lines that are close or cross.

12.2 Scan converting circles

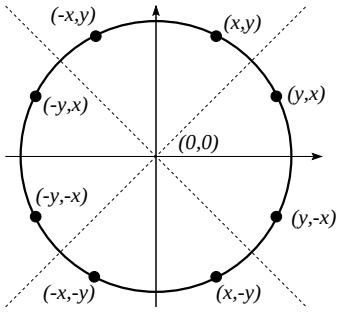


Figure 12.5: Eight fold symmetry of a circle.

Just like lines, efficient algorithms for computing those pixels to be set to draw a specific circle are available. The general equation for a circle is $x^2 + y^2 = r^2$, where r is the radius of the circle. This assumes the circle is centred at the origin, $(x = 0, y = 0)$, although this can easily be corrected for by adding the appropriate offset to all pixel values. It is again possible to solve the problem using a brute force approach, for instance by plotting $y = \pm\sqrt{r^2 - x^2}$ for a quadrant and using symmetry, or more sensibly $(r\cos\theta, r\sin\theta)$ where θ goes from 0° to 90°, again using symmetry.

These routines require complex functions (the square root or sine / cosine) to be evaluated for each pixel in the circle (or arc), and are thus very inefficient. Progress can initially be made by noting the eight fold symmetry of all circles. This can be seen in Figure 12.5, where given a point (x, y) which lies on the circle, there are seven other points which are easily found, which also lie on the circle. To simplify the plotting of a circle a small function, such as the `WriteCirclePixels` function will embody this symmetry (Listing 5), although adding a quick check for $x = y$ will speed things up because in this case only four pixels need to be set.

```
 void WriteCirclePixels ( int x,  int y,  int value)
{          /* Assumes we are plotting integer points */
   WritePixel(x,y,value);
   WritePixel(y,x,value);
   WritePixel(y,-x,value);
   WritePixel(x,-y,value);
   WritePixel(-x,-y,value);
   WritePixel(-y,-x,value);
   WritePixel(-y,x,value);
   WritePixel(-x,y,value);
}
```

Listing 5: Pseudocode for using the eight way symmetry of a circle.

12.2.1 Midpoint circle algorithm

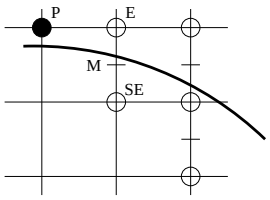


Figure 12.6: Midpoint circle algorithm.

Rather like the midpoint line algorithm, there exists a midpoint circle algorithm also developed by Bresenham, which also uses the eight way symmetry. Working on the second octant, where x goes from 0 to $r/\sqrt{2}$, while y goes from r to $r/\sqrt{2}$ and using the `WriteCirclePixels` function we use similar arguments to the midpoint line algorithm. Figure 12.6 shows an example where the current point P $= (x_p, y_p)$ has been selected. We can now either select the point E or SE, depending on where the line falls relative to the midpoint, M. We then proceed in a similar manner of the next pixel. Of course, this is not the clever bit of the algorithm, that comes in the part where we compute the decision variable, d that allows us to choose E or SE, in an efficient manner.

The implicit equation for a circle is $f(x, y) = x^2 + y^2 - r^2 = 0$. Inside the circle the function will be negative, outside positive. If the midpoint, M, is inside the circle then we will choose E, if outside, SE. So creating the appropriate decision variable:

$$d_{\text{old}} = f(x_p + 1, y_p - 0.5) = (x_p + 1)^2 + (y_p - 0.5)^2 - r^2 \ ,$$

we chose E if $d < 0$, otherwise SE. If we choose E, then the new decision variable will be:

$$d_{\text{new}} = f(x_p + 2, y_p - 0.5) = (x_p + 2)^2 + (y_p - 0.5)^2 - r^2 \ ,$$

that is, $d_{\text{new}} = d_{\text{old}} + 2x_p + 3$ giving the increment $\Delta_{\text{E}} = 2x_p + 3$. For the case when we choose SE:

$$d_{\text{new}} = f(x_p + 2, y_p - 1.5) = (x_p + 2)^2 + (y_p - 1.5)^2 - r^2 \ ,$$

that is, $d_{\text{new}} = d_{\text{old}} + 2x_p - 2y_p + 5$ giving the increment $\Delta_{\text{SE}} = 2(x_p - y_p) + 5$.

This looks very similar to the midpoint line algorithm, although in this case the increments depend upon the current position. This makes the algorithm more expensive since we need to evaluate these linear functions. To implement the algorithm we need to know what to do at the start point (an initial condition). In the second quadrant, assuming an integer radius, the first point lies at $(0, r)$. The next midpoint lies at $(1, r - 0.5)$ where $f((1, r - 0.5)) = 1.25 - r$, which gives us our first value of d. This is not readily amenable to integer arithmetic, so we use a slightly different decision variable h. Now if $h = d - 0.25$ then everything will be integer, in particular at the first step, $h = 1 - r$. Since h is an integer value (as r is) and the increments are all integer, testing $h < -0.25$ (which is $d < 0$) is the same as testing $h < 0$.

Listing 6 shows the pseudocode of the integer midpoint circle scan conversion algorithm, which is similar to the midpoint line algorithm but this time uses linear equations to update the decision variable h.

It is possible to improve the algorithm further by recognising that any second order polynomial can be written using second order differences. Without considering the mathematical derivation (which is quite straightforward) observe that in moving from P to E, the increment Δ_{Eold} at $(x_p, y_p) = 2x_p + 3$. If from the new point we again choose E we find that Δ_{Enew} at $(x_p + 1, y_p) = 2(x_p + 1) + 3 = \Delta_{\text{Eold}} + 2$. This is because the Δ's are evaluated

```
 void MidpointCircle ( int radius,  int value)
{          /* Assumes integer radius. */
    int x,y,h

   x = 0;
   y = radius;
   h = 1 - radius;
   WriteCirclePixels(x,y,value);
    while (y > x) {
       if (h < 0) {
          h += 2*x + 3;
          x++;
       }  else {
          h += 2*(x - y) + 5;
          x++;
          y--;
       }
       WriteCirclePixels(x,y,value);
   }
}
```

Listing 6: Pseudocode for the midpoint circle scan conversion algorithm.

using linear functions, whose difference will be a constant. Even if we move to E, we need to update Δ_{SEnew} which is easily shown to be $\Delta_{\text{SEold}} + 2$.

Similarly if we choose SE we must update $\Delta_{\text{Enew}} = \Delta_{\text{Eold}} + 2$ and $\Delta_{\text{SEnew}} = \Delta_{\text{SEold}} + 4$. If we implement this second order scheme in the algorithm, we eliminate all multiplications (this is equivalent to efficient routines used to evaluate polynomials in numerical methods). The pseudocode for such an algorithm is shown in Listing 7.

12.3 Scan converting area primitives

Scan converting objects with area is more complex than scan converting linear objects, because the boundaries cause problems. Even a simple rectangle can give problems if it shares a common edge with another rectangle particularly if `xor` mode is being used. This is true for all area containing primitives. It seems logical that those pixels in the interior of a polygon should belong to that polygon, but what about those on the boundaries? The solution is to impose some rules that clarify the situation.

The most simple algorithm for scan converting axis aligned rectangles would be the one shown in Listing 8. However, this routine will fill the shared pixels of two rectangles which share an edge twice.

A rule that is commonly used to decide what to do with edge pixels is as follows. A boundary pixel is not considered part of the primitive if the half-plane defined by the edge and containing the primitives lies below a non-vertical edge or to the left of a vertical edge. Thus given this definition the top and right hand rows and columns of a rectangle are not drawn as part of the object. Thus rectangles sharing a common boundary would be correctly represented. There are several important points concerning this scheme:

- ➢ it applies equally well to polygons;
- ➢ it works for unfilled polygons, and should be used for consistency;
- ➢ each span misses its right- and top- most pixels.

It should be clear that there is no panacea here, any set of rules will have its own drawbacks and advantages. Implementors consider this scheme to be visually less disturbing than other competing methods.

```
void MidpointCircle ( int radius, int value)
{          /* Assumes integer radius. */
    int x,y,h,dE,dSE

   x = 0;
   y = radius;
   h = 1 - radius;
   dE = 3;
   dSE = 5 - 2*radius;
   WriteCirclePixels(x,y,value);
    while (y > x) {
       if (h < 0) {
          h += dE;
          dE += 2;
          dSE += 2;
          x++;
       } else {
          h += dSE;
          dE += 2;
          dSE += 4;
          x++;
          y--;
       }
       WriteCirclePixels(x,y,value);
   }
}
```

Listing 7: Pseudocode for the midpoint circle scan conversion algorithm using second order differences to ensure only integer addition is used.

```
void FillRectangle ( int xmin, int xmax, int ymin, int ymax, int value)
{
    int x,y;

    for (y = ymin; y <= ymax; y++) {
       for (x = xmin; x <= xmax; x++) {
          WritePixel(x,y,value);
       }
   }
}
```

Listing 8: Pseudocode for a very simple rectangle filling algorithm.

12.3.1 Filling polygons

Routines for scan converting polygons are covered in text books such as Foley *et al.* (1993, p. 87) and Hearn and Baker (1994, p. 117). We do not have time to cover such methods in depth in the course and the interested reader is invited to follow up the results in those books. In any case these scan conversion routines are pretty low level and will generally already be hardware or software encoded in almost all applications. Any given algorithm for polygon filling will have drawbacks, such as not properly rendering sliver polygons (that is very thin polygons which can be less than one pixel wide).

Most algorithms work as follows:

- ➢ find the intersections of the scan line with all polygon edges;
- ➢ sort the intersections;
- ➢ fill those points which are interior.

The first step involves the use of a scan-line algorithm that takes advantage of **edge coherence** to produce a data structure called an **active-edge table**, details in Foley *et al.* (1993, p. 91). **Edge coherence** simply means that if an edge is intersected in scan line i, it will probably be intersected in scan line $i + 1$.

12.3.2 Pattern filling

Patterns will typically be defined by some form of pixmap pattern. In this case the pattern is assumed to fill the entire screen, then `and`ed with the filled region of the primitive, determining where the pattern can 'show through'. This means the pattern is anchored to the screen origin and thus will change as the primitive is moved. In the other commonly used alternative method it is necessary to decide where in the primitive the pattern is anchored. Once this is done the pattern is replicated at those places determined using the scan-line algorithm for filled primitives. This type of operation is now widely used in many of the more modern graphics cards, where it is referred to as texture mapping.

12.4 Thick primitives

Primitives that are 1 pixel wide are often rather difficult to see or do not reproduce well. Thus it is often necessary to scan convert thick primitives, which can be done in several efficient ways. This is often done by placing a thick pixel 'brush' at each pixel chosen on the primitive by the scan conversion algorithm. However this causes problems with `xor` mode operations and is inefficient, since several pixels will be set several times. It is also necessary to define what occurs at the ends and edges of the primitives and how the 'brush' behaves as the primitives change direction. These issues are briefly dealt with in Foley *et al.* (1993, p. 97).

12.5 Clipping

Clipping is important because it allows us to efficiently scan convert specific regions. This can speed up the user view of the application model, and the faster this is in general the better. It is often convenient to combine scan conversion with clipping, called **scissoring**, in integer graphics packages such as SRGP, although lines and polygons, which are analytically clipped rather efficiently may be clipped first. This is very memory efficient and the whole activity can be performed in the CPU instruction cache or the display controllers microcode memory. Floating point graphics, such as are implemented in OpenGL, are most efficiently implemented by performing analytical clipping in the floating point coordinate system and then scan convert the clipped region. Details of clipping can be found in Foley *et al.* (1993, p. 101) and Hearn and Baker (1994, p. 224).

12.6 Text

Most high quality and WYSIWYG systems do not use raster definitions of their fonts, rather a curve based representation is used which will be discussed later. This representation is then scan converted to produce the output. However some systems still use rectangular bitmaps, stored in a **font cache**, to represent textual characters. The character is then `copyPixel`ed to the required position on the screen. A separate font cache is need for each font size and style. However the curve based representations may require multiple sizes to be differently defined for aesthetic reasons, so the space saving of curve based fonts is not that great, and they require a great deal more effort in scan conversion.

12.7 Anti-aliasing

All raster primitives outlined so far have a common problem, that of **jaggies**. These result from the discrete nature of the pixelated display canvas. **Jaggies** are a particular instance of **aliasing**. Techniques that attempt to minimise the visual impact of jaggies are known as **anti-aliasing**. The term alias originates from signal processing where it can be simplified to imply the effect of a signal which cannot be resolved at the sampling frequencies.

Since our application models, which will typically be based on 2D or 3D graphics primitives are generally defined in continuous space, the scan conversion process will necessarily introduce some aliasing problems. In the limit, as the pixel size shrinks to an infinitely small dot, these problems will be minimised, thus one solution is to increase the screen resolution. This will indeed reduce the aliasing problems, but at the expense of the requirement for extremely high resolution screens[5], which are not currently available. Thus at the moment

[5]Note that doubling screen resolution will quadruple the memory requirements and the scan conversion time.

another solution must be found.

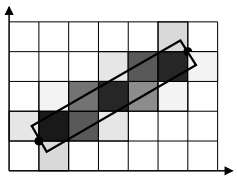


Figure 12.7: Simple anti-aliasing.

One solution to the problem involves recognising that primitives, such as lines are really areas in the raster world. Even a horizontal or vertical line will have width at least one pixel, thus can be represented as a rectangle rather than a line. This leads to the concept of **unweighted area sampling** where the intensity of the pixel is set according to how much of its area is overlapped by the primitive (as in Figure 12.7). Here we have assumed that the pixels can be represented as square areas with constant intensity. Of course doing these calculations is computationally more expensive than straight scan conversion, however the result, particularly from a distance, is greatly improved appearance.

This is one of the most simple anti-aliasing methods. More complex methods involve weighted area sampling. A good reason for using weighted area sampling is that the square, constant intensity model of a pixel is incorrect for most display devices (see Figure 11.2). Furthermore the primitive will not affect the pixel intensity unless there is some overlap. which may produce slightly worse results than a more gradual transition.

In **weighted area sampling** we assume a realistic model for pixel intensity, which involves pixels having a disc like support which is larger than the actual pixel size, so that adjacent pixels overlap. The method proceeds by placing a **weighting function** at the pixel centre and then computing the the percentage of the volume of the weighting function that is contained in the primitives area (Figure 12.8). Using a sensible weighting function, such as a cone or Gaussian function, will result in a smoother anti-aliasing, but at the price of even greater computational burden. For CRT based display devices this type of anti-aliasing can produce much better display results, however for square pixel based LCD systems Gaussian anti-aliasing is less important. However some LCD's now use anti-aliasing methods to allow non-native resolutions to be supported on fixed resolution LCD devices.

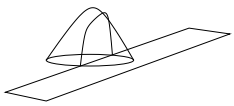


Figure 12.8: Weighted anti-aliasing.

Most of the anti-aliasing methods discussed so far have been concerned with single objects such as lines. What can we do about aliasing across the screen for all objects? One solution, called full screen anti-aliasing, is to produce several versions of the same resulting bitmap, each shifted by a small amount. When these are added together (possibly in a weighted combination) the effect will be anti-aliasing of all objects on the screen. The down side is the necessity to compute several version of nearly the same scene. In some very recent hardware developments 4 separate rendering pipelines are provided just for this functionality.

13 Basic Image Processing

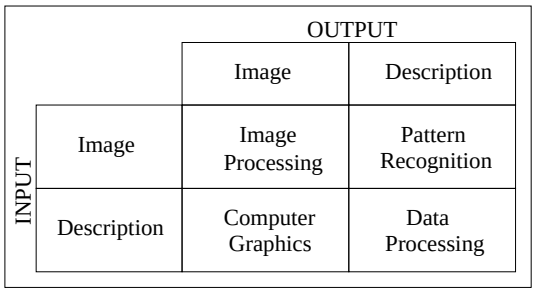


Figure 13.1: Computer graphics and image processing

Image processing is related to, but distinct from, *computer graphics*. This section of the course is designed to give you a basic insight into what image processing involves and methods that can be used for image compression. Figure 13.1 recalls that image processing involves taking an existing image and changing it in some way to produce another image. In general the aim will be to enhance certain features of the image, to improve its clarity, aesthetics or extract some particular information. By its very nature image processing is highly mathematical, but this introduction is largely descriptive.

14 Image compression

We are routinely storing more and more images, from scanners, digital cameras, generated by computer graphics (screen shots) and now digital video and television. Storing these images has not really been a problem as hard disk sizes have risen and prices have fallen. If we consider a 1024×768 pixel image in true colour (24 bit), then this would take about 2.4 Mb to store. When we have disks of the order of 30 Gb this is not a huge problem however the growth of the world wide web and its popular uptake has meant that more and more images are transmitted over a finite bandwidth. Thus to minimise the impact of graphics on the web, to optimise storage efficiency and allow us to fit images onto floppy disks we have to develop some form of data compression.

Data compression attempts to use the structure present in the data to minimise the amount of space required to store the given information. There are very many different forms of data compression, but they can be broadly divided into two types. **Lossless compression** uses the structure in the data to compress the information in such a way that it can be exactly reproduced. Lossless compression is used in the `zip` and `gzip` programs because we clearly do not want the compression / uncompression process to change our data. In image compression we may also use **lossy compression**. Lossy compression may alter our data somewhat, but retains the important information. Because we can throw away some of the data we generally get much better compression rates when using lossy compression.

Images are almost always stored as bitmaps (that is an array of pixel colour / intensity) The most basic image format is the bitmap (`.bmp`) file. This stores the intensity values for each pixel as a n bit number. Thus to store a true colour (24 bit) image of size 1024×768 would take 2.4 Mb without any image compression. There are several other image formats which are widely used (e.g. `.gif` and `.jpg`) which use image compression methods to reduce the storage requirements.

14.1 Lossless image compression

In this section we will briefly look at the methods which can be used to compress information (in this case images, but the techniques are also used elsewhere). When compressing colour images it is often useful to convert from RGB space into HSV (Hue Saturation and Value or Brightness) space since the most important information that our eyes are sensitive to is the brightness, and it is this which tends to vary most in the image. This means that in RGB images all channels vary strongly while in HSV images typically only the V channel has a large variability.

Of course the amount of compression we can achieve for any given image will depend on the method used and the complexity of the image. When assessing the complexity of an image one measure that is widely used is the image entropy. Entropy is a measure of disorder or randomness. An image composed of independent randomly distributed pixel values has maximum entropy (and will be very difficult to compress) while an image that

consists of just one colour will have minimum entropy. This concept links image compression with the world of statistics and provides us with a rigorous framework for developing the theory behind image compression (which we do not exploit further here!).

14.1.1 Run length encoding

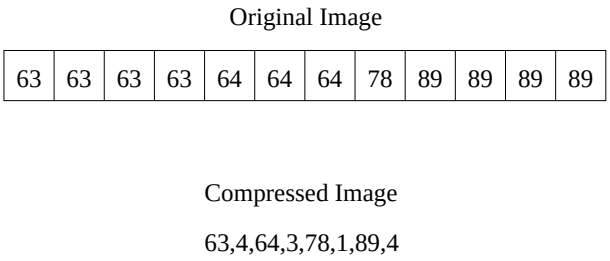


Figure 14.1: Run length encoding of a line of pixel brightnesses from some image.

As the name suggest **run length encoding** attempts to compress an image by coding the number of bits taking the same value along a given scan line. We are exploiting the scan line coherence that can be used in visible surface determination. The method is illustrated in Figure 14.1. The method works particularly well on binary images, since there is no need to store the actual values, only the length of the runs of zeros and ones. We can also use **bit-plane run length encoding** on non-binary images by considering each bit of the, say 8 bit, image one at a time.

On typical images these compression schemes produce compression rates of 1.5:1 (gray-scale / colour images), 4:1 (binary images) and 2:1 (bit-plane compression on gray-scale / colour images). However if the intensity values of an image change every pixel this scheme will produce a **data explosion**, thus when implementing this type of scheme the algorithm should check that such a problem does not occur. This scheme is also sensitive to channel errors since a corruption near the start of the image will propagate through the entire image. This can be alleviated by and end of line marker.

14.1.2 Huffman coding

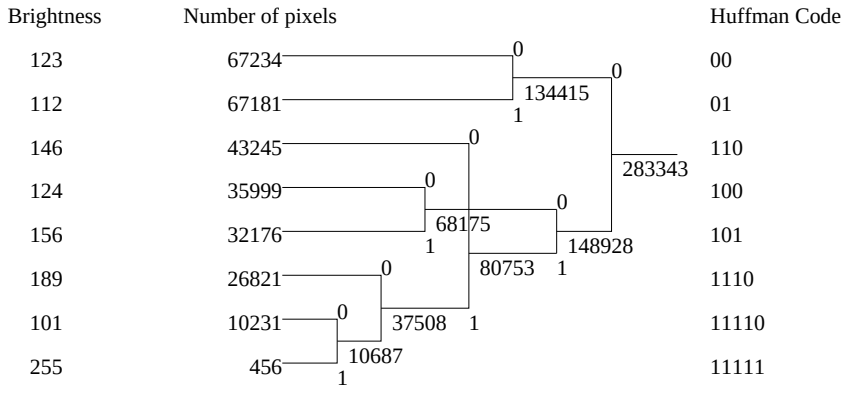


Figure 14.2: Huffman coding of a 3 bit gray-scale image.

Huffman coding works on the image brightness histogram. It finds the most commonly occuring brightness patterns and uses the shortest codes to represent these. Progressively longer codes are used for the less frequently occuring patterns. This a tree based method, which is probably most easy to understand by reference to Figure 14.2. On typical images Huffman coding gives compression rates of 1.5 – 2:1. Huffman coding may also be used after run length coding to give further compression.

14.1.3 Predictive coding

Predictive coding again exploits scan line coherence, storing each line using its initial pixel brightness and then the differences between successive pixels brightnesses (which we hope can be stored in fewer bits). This is shown in Figure 14.3 where it can be seen that the method relies on the image having smooth changes in brightness. The method is also referred to **differential pulse code modulation**. At edges in the image we may well need to allow for overload patterns (that is the ability to flag the fact that the brightness change is greater than we can represent with the number of bits we have allowed ourselves). The routine may also be applied iteratively, since the second order differences are typically smaller than the first order differences (this

Original Image

63	63	63	63	64	64	64	78	89	89	89	89

Compressed Image

63,0,0,0,1,0,0,14,11,0,0,0

Figure 14.3: Predictive coding of a line of pixel brightnesses from some image.

is related to the method used to scan convert a circle). Typically the application of predictive coding gives up to 2:1 image compression rates.

14.1.4 Block coding

Block coding looks for frequently occuring block patterns in the image, rather than simply looking along the scan lines. This is more computationally expensive than the other methods we have considered so far, but gives slightly better compression rates. As well as sending the block codes for the image we also need to send the codebook (or look up table) to decode the image. Despite this compression rates are generally 2 – 3:1.

Lemple-Ziv-Welch coding uses a variant of the block method. First the number of blocks to be used is fixed, and these are filled with the longest, most commonly occuring block patterns. Overload codes may be used if we do not have a sufficient number of block patterns. The coded image is then further compressed using Huffman coding. This forms the basis of the `.gif` file format, with some modifications.

14.1.5 Problems of lossless compression

So far we have only considered lossless image compression. For some applications it may be very important to retain **all** information in the original image, however if some of this information is noise this may not be desirable. The fact that we retain everything in the image means that our compression rates on real images are still only of the order of 3:1 at best, which means that our 2.4 Mb image is now down to 0.8 Mb. This is still rather large. If we throw away some of the irrelevant information in the image we can reduce this by quite a large factor. The price to pay is more computation.

14.2 Lossy image compression

When performing lossy compression we accept a small loss of information from the image to achieve a better compression rate. It is possible to get compression rates of 10:1 without noticeable degradation of the image quality and rates of 100:1 with only slight effects. The quality of the TV image is very poor but we still find it acceptable, thus some loss of quality will often go un-noticed.

14.2.1 Truncation coding

Truncation coding is the simplest lossy compression method. We remove the least significant bits from the image or remove successive pixel rows and columns. If we remove every other row and column from an image we will reduce the size by a factor of four giving a compression rate of 4:1. Of course this will produce a smaller image, so we will need to use some interpolation method to reproduce the approximate original image. We can also truncate the number of bits used to store the image, representing say an 8 bit image using only 4 bits of brightness information, giving a compression ratio of 2:1. To reconstruct the approximate original image we need to add extra brightness depth to avoid **posterising effects** (big blocks of the same brightness) by adding 4 bits of **dither noise** back onto the image. Truncation coding can also be applied to sequences of images, simply by dropping every other frame for example.

14.2.2 Lossy predictive coding

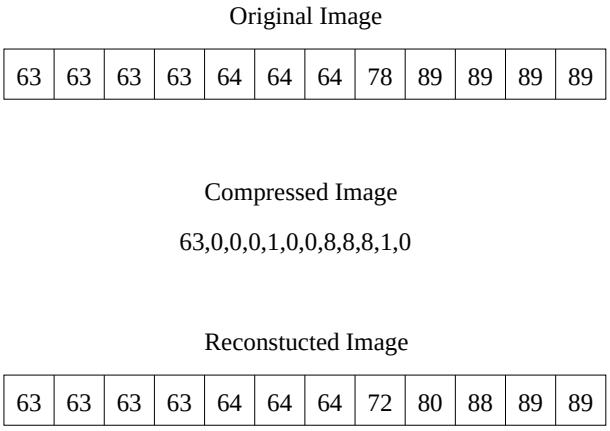


Figure 14.4: Lossy predictive coding of a line of pixel brightnesses from some image.

Lossy predictive coding is rather like the lossless case without the need for a code overload option. Thus if the algorithm cannot represent the change in one step it is successively corrected in the following pixels (see Figure 14.4). This will act to smooth sharp changes in the image but will give increased compression ratios of the order of 3:1. Selecting an appropriate code length for an image can reduce the severity of the distortions introduced.

Delta modulation is a special case of lossy predictive coding, where only one bit is use to indicate brightness difference. This will introduce significant smearing of the image, but will give a compression ratio of 8:1 for an 8 bit gray-scale image.

14.2.3 Lossy block coding

Lossy block coding, like its lossless counterpart use codebook blocks to encode an image, although this time not necessarily exactly. The difference between the original and reconstructed image – the error image – is minimised. The criterion used to determine the error (the error metric) is determined by the application for which the compressed image is to be used. A common error measure, the sum of the squared differences in the pixel brightnesses, is used because this is what the human eye is most sensitive to.

Fractal compression methods also use basic codebook blocks, but the image can be described in terms of translations, rotations and scalings of these base images. Thus the self-similarity of natural objects is embodied in the compression scheme.

Most lossy block coding schemes can produce compression ratios of 10:1 and better without noticeable distortion of the original image.

14.2.4 Transform coding

Transform coding is like block encoding but this time the blocks used are predefined, thus there is no requirement to search for and store them. The most common method is to transform the image from the spatial domain to the frequency domain. This transform is based on the 2D Fourier transform (or in practice the discrete cosine transform), for which a fast, efficient algorithm exists (the fast Fourier transform). However one problem with the Fourier transform is that it treats the image as being periodic (which means that the edges wrap around onto each other) which is unrealistic. To overcome this windowing techniques are used, whereby blocks of the images (typically of size 4×4 or 8×8 pixels) are coded in frequency space, using the discrete cosine transform.

Another method, growing in popularity is the **wavelet transform** which uses slightly different basis functions to encode the image. It is possible to tune these basis functions to the types of images that we want to represent. In general transform coding will compress images by rates of 10:1 and better, again with very little distortion. The price to pay is the extra computational effort. The `.jpg` file format uses the discrete cosine transform followed by lossless predictive coding of the retained frequency components and then Huffman coding of the predictive image. The quality (and size) of the compressed image can be determined by changing the number of frequency components that are retained.

15 Image Processing

In this section we consider the basic methods used in image processing and their applications. The field of image processing was principally initiated by NASA when processing Ranger 7 pictures of the surface of the moon to assess possible landing sites for the Apollo missions. Thus the field is relatively new, but has undergone rapid development over the last fifty years. A good source of reference is *Baxes* (1994) which gives a non-mathematical introduction to image processing, or for the more Java inclined amongst you (and also a fair bit cheaper) *Efford* (2000).

Baxes, G. A., 1994. *Digital Image Processing: Principles and Applications*, John Wiley & Sons, New York.

Efford, N., 2000. *Digital image processing, a practical introduction using Java*, Addison-Wesley, Harlow, UK.

15.1 Applications

Image processing is now used in many applications. These include:

- ➢ remote sensing;
- ➢ medical applications;
- ➢ intelligence gathering / law enforcement;
- ➢ document processing;
- ➢ 'smart' weapons;
- ➢ artistically;
- ➢ car guidance and
- ➢ optical correction (Hubble telescope).

Of course this is just a short list of some applications, there are many more and many of these applications are one step in a larger processing chain. For instance most face recognition algorithms start with some form of image processing.

15.2 High level overview

We can broadly divide image processing into two main classes; **image enhancement** and **image restoration**. **Image compression**, which we have just dealt with, is also a discipline of image processing, while we might also include **image analysis** and **image synthesis**. Image analysis and image synthesis are really in the realms of statistical pattern analysis, but often have components which would be considered as image processing techniques.

Image restoration is concerned with the correction of camera induced errors (**photogrammetric correction**) and viewing induced errors (**geometric correction**). Inverse filtering is also sometimes considered part of image restoration.

15.3 Image enhancement

This is the only part of image processing that we have time to deal with in any depth. We will consider two central techniques: **contrast enhancement** and **image filtering**. Both can be viewed in a simple framework. If the original image is denoted $I(x, y)$, then all image processing operations can be represented as:

$$O(x, y) = M[I(x, y)] \ ,$$

where M is some operation applied to the original image (e.g. a filter) and $O(x, y)$ is the resulting (new) image.

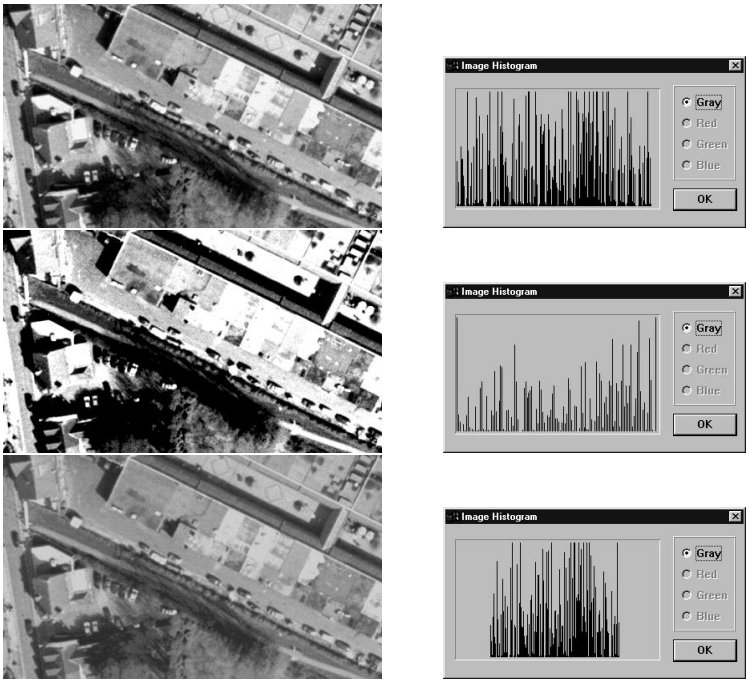


Figure 15.1: Top, the original image and its histogram, middle the high contrast image and histogram and bottom the low contrast image and histogram.

15.3.1 Contrast enhancement

The aim of contrast enhancement is exactly what the name suggests, to enhance the colour or brightness contrast of an image. Unlike the filtering techniques we will discuss in the next section this operation is calculated on the image histogram rather than individual pixels. The image histogram is simply the histogram of the image brightness values. For colour images this is the histogram for each channel (RGB or HSV).

We can either enhance the contrast of an image by moving the histogram mass to the edges of the histogram, or we can reduce the contrast of the image by bringing the mass into the middle of the histogram. This is illustrated in Figure 15.1, which shows the original image and its contrast enhanced and reduced version. The histograms are also shown.

In general contrast enhancement is used to improve the visual look of the image, although at it's limit (a **binary contrast enhancement**) we will only produce black and white colours, generally in equal measure. We could achieve the same effect by setting a threshold at a certain brightness value and allocating everything below this value to black and above to white. This is called **binary thresholding** and will not necessarily ensure a balanced histogram.

In some cases it might be necessary to define the thresholds or the image histograms to be local (in space) and thus changeable (in space). If we change the threshold or neighbourhood as we process the image then we are performing what is known as an **adaptive operation**. In many cases the threshold will be automatically determined using only the surrounding portion of the image, which is to be processed. We do not have time to develop these powerful methods.

15.3.2 Filtering

Filtering is almost always a local operation (that is the filter is applied over a finite neighbourhood). The main exception to this we have already met in the section on image compression. This is the transformation of the image from space to frequency which can be written as a filtering operation (particularly when certain frequency components are removed). In this section we only consider filtering in space rather than frequency.

The most simple filters are designed to smooth or sharpen the image, around a local neighbourhood. They can be written in term of the weight the is used to multiply the values of the surrounding pixels before adding the result to give the central pixel value.

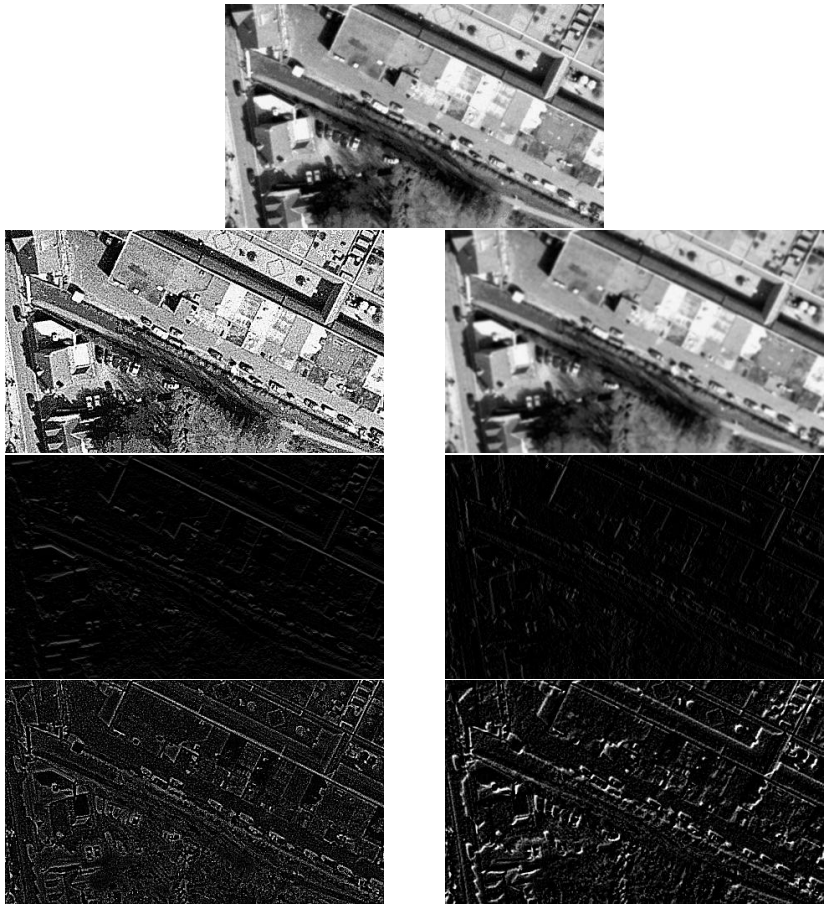


Figure 15.2: Top, the original image, next a high pass filtered version (left) and a low pass filtered (right), next horizontal (left) and vertical (right) edge filters and bottom the Laplacian (left) and Prewitt north-west (right) filters.

The easiest filter to understand is the smoothing filter (often also called the **blur** filter). This can be written as:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}.$$

If this is applied on each 3×3 window of the image we will compute a smoother version of the original image. Since the sum of all the filter values is 1, the overall brightness is unchanged. This sort of filter is also often referred to as a **low pass filter** since it will tend to remove the high frequency components from an image (and pass the low frequency components, hence the name). Since noise is generally assumed to be high frequency this is also often called a **noise filter**.

Another noise filter, this time more robust, but also more smoothing is the median filter. This assigns the median value of surround $n \times n$ pixels to the central pixel. This is very effective for removing speckle noise. Note that although the filters here are written as operating over a 3×3 square window there is no reason why they cannot be applied over bigger windows and non-square windows.

If rather than blurring the detail, we actually wish to bring the detail out, then we can apply a **high pass filter** which emphasises the high frequency components. The **sharpening** filter can be written:

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 9 & -1\\ -1 & -1 & -1\end{bmatrix}.$$

Notice that again the sum is 1, so this filter is also **unbiased** in that the overall brightness is conserved. This filter can be used to enhance the look of a digital photograph for instance.

Another set of commonly used filters are edge filters. These are frequently used in many image processing

applications, for instance prior to automatic image segmentation. The most simple edge filters are:

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} ,$$

for horizontal and vertical edge enhancement respectively. Note that these filters sum to 0, thus the average brightness value will be zero for homogeneous images. The largest values will occur where there is the greatest change in brightness across pixels, hence their ability to enhance edges. A more effective, but still directional filter is the Prewitt gradient filter:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{bmatrix} ,$$

which in this case picks out edges aligned north-west – south-east. This filter again has zero sum. A non-directional edge filter is given by the so called **Laplacian** filter which is given by:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} ,$$

which is also zero sum. The results of the application of these filters to an aerial photograph is shown in Figure 15.2.

All of these filters can be applied to images in any order, and iteratively if necessary. It is possible to develop many other filters, although we have covered to most commonly occuring ones. The filtering is generally a precursor to more complex manipulation of the image, for instance segmentation, character recognition or visual inspection. We do not have time to cover this fascinating area.

16 Summary

We have now reached the end of the course. You should now have a pretty decent appreciation of the methods that are used to generate the computer graphics which grace our screens on a daily basis. Right from the hardware necessary, through the basic methods for scan conversion of the final pictures, to the transformations used in 2D and 3D to generate animated graphics, the methods used to represent solid objects and render them almost realistically on the screen. We have also covered other areas such as using curves, image compression and really basic image processing. In addition the lab element has introduced you to OpenGL.

I hope you have enjoyed the course (a bit!) and if you have any comments on what could be done to improve the course for future years send them to `d.cornford@aston.ac.uk`.