UNIVERSITY
OF LONDON

Advanced graphics and animation

K. Devlin and M. Gillies

CO3355

2014

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

Kate Devlin, Department of Computing, Goldsmiths, University of London

Marco Gillies, Department of Computing, Goldsmiths, University of London.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the authors are unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Preface

About this half unit

CO3355 Advanced graphics and animation, is a half unit Level 3 option for the BSc/Diploma in Computing and Information Systems (CIS) and the BSc/Diploma in Creative Computing. It combines an overview of key topics in computer graphics and animation, including major contemporary graphics and animation techniques. These theoretical aspects are taught in the context of their practical use, with hands-on experience of current software tools to exemplify and explore the written theory. We will briefly introduce some industry standard graphics software tools but the main focus will be on programming the graphical software.

This course has no specific or formal prerequisites. Some of the material covered in this subject guide will be familiar to those who have taken Creative Computing I and Creative Computing II, although this unit will not cover those topics in such depth. There will be some mathematics and programming involved and this is introduced in this subject guide, though previous familiarity with topics such as trigonometry, vectors and matrices would be advantageous.

Assessment

The course is assessed via an unseen written examination and practical coursework. A sample examination paper is provided in an Appendix at the end of this subject guide, with some guidelines on how to answer the questions. You will be required to answer three questions out of a possible five. The examination is designed to assess your understanding of key concepts from the course. Sample examination questions are provided at the end of each chapter.

The coursework will focus on the practical elements. You will be expected to provide electronic copies of your coursework. It is very important that any material that is not original to you should be properly attributed and placed in quotation marks, with a full list of references.

Important: the information and advice given here are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check both the current Regulations for relevant information about the assessment, and the VLE where you should be advised of any forthcoming changes. You should also carefully check the rubric/instructions on the examination paper you actually sit and follow those instructions.

Remember, it is important to check the VLE for:

- up-to-date information on examination and assessment arrangements for this course
- where available, past examination papers and Examiners commentaries for the course which give advice on how each question might best be answered.

Aims of this course

This course will cover advanced methods used in current state of the art 2D and 3D graphics and animation systems. This course will cover the mathematical foundations, computational techniques and their use in creative practice. Students will be given the mathematical foundations of the subject as well as other theoretical foundations such as perceptual theories.

Learning outcomes

On successful completion of this course, including Essential and Recommended readings, exercises and activities, you should be able to:

- understand and explain the mathematical and theoretical principles of computer graphics
- understand and explain many computer graphics techniques used in contemporary graphical software
- write basic but complete graphics software systems
- analyse and evaluate the use of computer graphics methods in practical applications
- apply computer graphics techniques in order to create aesthetic effects
- plan and implement a small piece of software taking into account technical and aesthetic considerations.

The subject guide and other learning resources

This subject guide is not intended as a self-contained textbook. It is designed to help you learn by setting out specific topics for study in the CO3355 half unit. It will identify issues that are important and highlight particular areas of study. There are recommended textbooks and a number of other readings are listed at appropriate places. There are also links to websites providing useful resources. It will not be possible to pass this half unit by reading only the subject guide. Please refer to the Computing VLE for other resources, which should be used as an aid to your learning. Please note that colour versions of some of the diagrams in the subject guide are available in the electronic version; you may find them easier to read in this format.

Suggested study time

The Computing Handbook states the following: 'To be able to gain the most benefit from the programme, it is likely that you will have to spend at least 300 hours studying for each full unit, though you are likely to benefit from spending up to twice this time'. Note that this subject is a half unit.

The course is designed to be delivered over a ten-week term, and there are ten chapters to the subject guide. It is suggested that a chapter be read in detail each

week, while also accessing the recommended associated reading materials and resources for that chapter. The practical exercises and examples should also be completed alongside the relevant chapter. Revision should take place over a number of weeks before the examination.

Reading lists and other learning resources

This is a list of textbooks and other resources which will be useful for all or most parts of the course. Additional readings will be given at the start of each chapter. See the bibliography for a full list of books and articles referred to, including ISBNs.

Essential reading

- Angel, E. and D. Shreiner *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL.* (USA: Addison-Wesley Publishing Company, 2011) sixth edition [ISBN 9780273752264(pbk); 9780132545235].

  This is a good general graphics textbook. The maths is explained in a very thorough and rigorous way.

- Shirley, P. and S. Marschner *Fundamentals of Computer Graphics.* (Natick, MA, USA: A.K. Peters, Ltd, 2009) third edition [ISBN 9781568814698].

  This is a more advanced text, covering relevant topics above and beyond those in Angel and Shreiner (above).

Additional reading

There are a wide variety of computer graphics textbooks available. From the point of view of computer graphics theory, most graphics textbooks should cover the basic concepts and algorithms. We recommend the following widely-used texts:

- Glassner, A. *Principles of Digital Image Synthesis.* (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994) [ISBN 9781558602762].

  Graphics pioneer Andrew Glassner wrote these two volumes as a guide to the human visual system, digital signal processing, and the interaction of matter and light. This is an excellent resource written in a very accessible manner and is now available free in PDF format (currently downloadable from `http://www.realtimerendering.com/Principles_of_Digital_Image_Synthesis_v1.0.1.pdf`, available May 2014). Much of it is beyond the scope of this course but it does have a good discussion of advanced rendering methods and is full of fascinating information.

- Hughes, J., A. van Dam, M. McGuire, D. Sklar, J. Foley, S. Feiner, and K. Akeley *Computer graphics: principles and practice.* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2013) third edition [ISBN 9780321399526].

  This is the latest version of the graphics textbook known colloquially as 'Foley and van Dam'. Both this and the earlier editions cover much of the necessary theory in good detail.

- Watt, A. *3D Computer Graphics with Cdrom.* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999) third edition [ISBN 9780201398557].

  Despite this book being in its second decade, the theoretical basis of computer graphics is well-explained and fundamental concepts are described clearly.

Software requirements

This course will use the *Processing* language, available at `http://www.processing.org/`. The course was designed for *Processing* version 2.0; it should work on later versions of the language, but if there are problems with the latest version the *Processing* website contains an archive of previous versions. This should work on most currently active versions of Windows, Mac OSX and Linux; see the current support page: `http://wiki.processing.org/w/Supported_Platforms` (available May 2014) for more details of support for older systems. The 3D graphics component of *Processing* requires OpenGL 2.0 which should be supported by almost all current graphics cards. Chapter 9 requires the use of an additional physics library for *Processing*; we suggest BRigid, but there are many other suitable options.

Sources of further information

A very wide range of computer graphics and animation material is available online, including helpful tutorials and worked examples of key concepts. Graphics-related search terms will provide a multitude of potential sources, some more reliable than others. We have listed selected sources in each chapter but such lists are not exhaustive and are, of course, time-sensitive.

Structure of the guide

This section gives a brief summary of each chapter. The learning outcomes are listed at the beginning of each main chapter.

Chapter 1, An introduction to computer graphics describes computer graphics and animation and gives a brief history of the field, indicating the main application areas.

Chapter 2, Mathematics for graphics and animation summarises the key mathematics required for computer graphics.

Chapter 3, From model to screen describes the process of rasterisation – the process by which three-dimensional scene geometry is displayed on a two-dimensional computer screen.

Chapter 4, Graphics programming introduces the basic concepts involved in graphics programming.

Chapter 5, Lighting 1 – shading discusses light and colour and how polygons are assigned a value and shaded.

Chapter 6, Textures shows how texturing provides a shortcut way of adding realism to images without increasing the polygon count.

Chapter 7, Lighting 2 – representing the real world discusses the approaches to creating realistic images that represent the real world, and provides examples of applications from current research.

Chapter 8, Animation explores the basic concepts of computer animation.

Chapter 9, Physics simulation introduces physics simulation for animation and interactive graphics.

Chapter 10, Display explores what happens to the graphics we have created when we display them on a screen, and they are perceived by a human observer.

Sample examination paper.

Bibliography.

Chapter 1

An introduction to computer graphics

1.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 1.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapter 1.

1.2 Recommended reading

Bailey, M. and A. Glassner 'Introduction to computer graphics.' (New York, NY, USA: ACM, 2004) ACM SIGGRAPH 2004 Course Notes. A version of this can be accessed online at:
`http://www.inf.ed.ac.uk/teaching/courses/cg/Web/intro_graphics.pdf`
(available May 2014).

Durand, F. 'A short introduction to computer graphics: MIT laboratory for computer science' ; `people.csail.mit.edu/fredo/Depiction/1_Introduction/reviewGraphics.pdf`, (accessed May 2014), publication date unknown.

1.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- give an historical account of the field of computer graphics
- demonstrate knowledge of the terminology of computer graphics systems
- demonstrate awareness of the graphics pipeline.

1.4 Introduction

Computer graphics is a term that describes the use of computers to create or manipulate images. All of these computer-generated images are made from a number of very simple primitives: points, lines and polygons. These describe the geometry of an object; we can then build on that geometry by adding information about materials, textures, colours and light in order to produce a resulting image that looks 'real'. Three-dimensional (3D) computer graphics have a number of applications, with the four main categories being computer-aided design (CAD), scientific visualisation, the entertainment industries (that is, film and television) and computer gaming. The underlying algorithms that form the basis of these applications are all the same.

1.5 Why study computer graphics?

Computer graphics is a well-established field of computer science. It has a huge range of applications, from entertainment to cutting-edge science, and due to its many subdisciplines it is a field that is suited to both technical and artistic people alike. It uses clever algorithms and cutting-edge technology. It is the primary method to deliver information from computer to human, and this course teaches the fundamental concepts and algorithms behind creating computer-generated images and delivering them to the screen.

1.6 History

The term computer graphics was first used in 1960, where early work was focused on visualisation for scientists and engineers. In the decades prior to this there had been some early work in creating computer-generated images, but it was 1961 when the first popular computer graphics game, Spacewar, was written and 1963 when Ivan Sutherland's pioneering thesis, *Sketchpad*, was presented; both of these shaped the future of the discipline. Sketchpad allowed the user to draw simple lines and curves directly on the screen. It is still regarded today as one of the most important developments in human-computer interaction, and as one of the earliest examples of object-oriented programming.

From these beginnings, development of graphical hardware and algorithms commenced. In 1965, Bresenham presented his algorithm for drawing lines on a screen. This was followed by a move away from vector displays (where it was only possible to draw lines) to raster displays formed of a grid of pixels which allowed the on-screen graphics to be so much more than mere lines.

Shading was introduced in the 1970s, which allowed images to look solid. The 1970s were also the time when special effects houses were founded to explore the power of computer graphics in movies, including George Lucas' Lucasfilm, creators of the *Star Wars* films.

The 1980s saw the mainstream launch of the **graphical user interface** (GUI) when Apple released the first Macintosh computer. Also in that decade was the development of raytracing – a method for simulating light in a scene, resulting in very realistic images. In 1983, computer graphics pioneer Jim Blinn created pre-encounter animated simulations of NASA's Voyager space probe, for which he was awarded the NASA Exceptional Service medal[1]. Although his animations may seem a little clunky and basic to us today, this was a key moment when people realised that computer graphics had the potential to generate images and animations to rival photographs.

In the 1990s, computer animation took a lead role, most notably in the Pixar feature *Toy Story*, which was the first commercially successful computer animated full length film. CGI special effects were also key to films such as *Terminator 2* and *Jurassic Park*. The 1990s also marked breakthroughs in the gaming industry. Real-time 3D graphics became increasingly common in computer games, which led to an

[1] Blinn's animations can be viewed on YouTube – for example, the Voyager 2 Flyby of Saturn (1981) Official NASA Animation: `http://www.youtube.com/watch?v=SQk7AFe13CY` (available May 2014).

increasing demand for hardware-accelerated 3D graphics, as seen in games such as Quake, Half-Life and Unreal.

Learning activity

Explore the history of computer graphics and animation in more detail at:
`design.osu.edu/carlson/history/lessons.html` (Carlson (2003), available May 2014).

1.7 A graphics system

There are six major elements to a graphics system:

1. Input devices
2. Central processing unit (CPU)
3. Graphics processing unit (GPU)
4. Memory
5. Framebuffer
6. Output devices.

An **input device** is any device that allows information from outside the computer to be communicated to the computer. A variety of input devices are currently available but the most commonly encountered in everyday use are the computer mouse and keyboard, with touchscreen having recently gained prominence due to the increased use of mobile devices. In terms of graphics for computer gaming, a controller is often the main source of input. This has, in recent years, been extended to using the body itself as an input device, such as Microsoft's Kinect (interaction is covered in detail in subject guide **CO3348 Interaction design**).

Input devices take a signal from the user and send this signal to the **central processing unit** (CPU) – the hardware in a computer that carries out a sequence of stored instructions kept in computer memory. Such a sequence of instructions is known as a computer program. Increasingly, the power of the CPU is enhanced by another part of the computer – the **graphics processing unit** (GPU). The GPU is a dedicated, highly-parallel, processor that is ideal for executing mathematically-intensive tasks such as 3D graphics instructions quickly and efficiently. The GPU is covered in more detail in Chapter 4 of this subject guide.

Computer **memory** stores computer programs. This is directly accessible to the processing unit. Dedicated video memory is located on and only accessible by the graphics card. The more video memory, the more capable the computer will be at handling complex graphics at a faster rate.

A **framebuffer** (also known as a framestore) is a portion of memory representing a frame that holds the properties such as the colours of that frame and sends it to an **output device**, most typically a screen.

1.8 The graphics pipeline

Computers have a graphics pipeline: the sequence of steps used to create a 2D representation of a 3D scene. Figure 1.1 gives an overview. Each of the stages in this pipeline can be further subdivided and have their own pipelines too, but at this level the sequence of steps encompasses the very beginning of the modelling process to the final output – the **rendered** image – seen by the user.

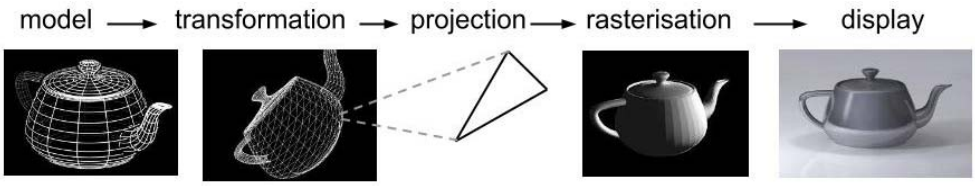


Figure 1.1: The graphics pipeline: the stages involved in going from geometric models to images.

This subject guide has been written to take you through this sequence of steps, from the initial geometric model to the completed image appearing on screen, explaining the key concepts and algorithms behind each stage.

1.9 Subfields of computer graphics

As mentioned previously, the main steps involved in moving from geometric model to rendered image are made up of many different research areas. An example of these follows[2]:

Mathematical structures: spaces, points, vectors, dusts, curves, surfaces, solids.

Modelling: ie. the description of objects and their attributes, including: primitives (pixels, polygons), intrinsic geometry, attributes (colour, texture), dynamics (motion, morphing).Techniques for object modelling, including: polygon meshes, patches, solid geometry, fractals, particle systems.

User interfaces: human factors, input/output devices, colour theory, workstations, interactive techniques, dialogue design, animation, metaphors for object manipulation, virtual reality.

Graphics software: graphics APIs; paint, draw, CAD and animation software; modelling and image databases; iconic operating systems; software standards.

Graphics hardware: input/output devices, specialised chips, specialised architectures.

Viewing: abstract to device coordinate transformations, the synthetic camera, windows, viewports, clipping.

Rendering: realism, physical modelling, ray tracing, radiosity, visible surface determination, transparency, translucency, reflection, refraction, shadows, shading, surface and texture mapping.

[2]Categories are modified from Ray Toal's homepage at: `cs.lmu.edu/~ray/` (accessed May 2014).

Image processing: image description, image storage, image transformations, image filtering, image enhancement, pattern recognition, edge detection, object reconstruction.

1.10 Summary

This chapter has introduced the framework for this subject guide: the steps involved in the graphics pipeline. The following chapters will work through the key theory involved, reinforced by practical examples and exercises.

1.11 Exercises

Find out about the computer-generated special effects used in films today.

1.12 Sample examination questions

1. What is the purpose of an input device? Give examples of two types of input device.
2. Describe the development and key moments in computer graphics from the 1960s to the present day.

Chapter 2

Mathematics for graphics and animation

2.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 3.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapters 2, 5 and 6.

2.2 Recommended reading

The topics covered in this chapter will be included in most high school or introductory university level mathematics textbooks and any will be sufficient. If you would like a text that is specific to 3D graphics we recommend the following text. One benefit of this book is that it covers homogeneous coordinates, which are not often covered in standard mathematics texts.

Dunn, F. and I. Parberry *3D Math Primer for Graphics and Game Development.* An A.K. Peters book (Taylor & Francis, 2011) second edition [ISBN 9781568817231].

2.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- perform calculations using trigonometry and vectors
- perform calculations on 3x3 and 4x4 affine transform matrices.

2.4 Introduction

This chapter will summarise the basic mathematics needed for 3D graphics. The mathematics requirements do not go beyond the A-level syllabus or other high school mathematics courses so we shall summarise it rapidly on the assumption that most of it will be familiar to you. If you are not familiar with any of the material please consult the Essential and Recommended reading above.

2.5 Trigonometry

This course will require use of basic trigonometry. You should ensure that you are familiar with the basic trigonometric fuctions **sine** (abbreviated to sin), **cosine** (cos)

and **tangent** (tan) as well as their inverse functions **arcsine** ($\sin^{-1}$ or asin), **arccosine** ($\cos^{-1}$ or acos) and **arctangent** ($\tan^{-1}$ or atan). If you are not familiar with the definition and use of these functions, please consult an introductory mathematics textbook.

2.6 Vectors

This course also requires the use of vectors to represent positions and directions in 3D space. You should ensure that you are familiar with the following:

- The definition of a vector in 3D space:

$$\mathbf{v} = (x, y, z)$$

- Position and direction vectors
- The length of a vector:

$$|\mathbf{v}| = \sqrt{x^2 + y^2 + z^2}$$

- Normalisation of a vector:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$$

- Multiplication by a scalar:

$$a\mathbf{v} = (ax, ay, az)$$

- Negating a vector:

$$-\mathbf{v} = (-x, -y, -z)$$

- Vector addition:

$$\mathbf{v} + \mathbf{u} = (v_x + v_x, v_y + v_y, v_z + v_z)$$

- Vector subtraction:

$$\mathbf{v} - \mathbf{u} = (v_x - v_x, v_y - v_y, v_z - v_z)$$

- Dot (scalar) product:

$$\mathbf{v} \cdot \mathbf{u} = v_x u_x + v_y v_y + v_z v_z$$

You should be familiar with all of these operations, both in terms of performing calculations using them and how they affect points in spaces. For example, you should be comfortable using vector subtraction to obtain a vector between two points or using the dot product to calculate the angle between two vectors. If you do not feel comfortable with any of the following exercises please revise these topics in an introductory mathematics textbook.

Learning activity

Attempt the following problems. If you are unable to answer a significant number of them we recommend you consult a textbook as described in the Essential and Recommended reading sections of this chapter.

1. Evaluate the following vector expressions.

(a) $5 * \begin{pmatrix} 4 \\ 3 \end{pmatrix}$

(b) $- \begin{pmatrix} 4 \\ 3 \end{pmatrix}$

(c) $2 * \begin{pmatrix} 1.3 \\ 5.6 \end{pmatrix}$

(d) $\begin{pmatrix} 8 \\ 3 \end{pmatrix} - \begin{pmatrix} 3 \\ 5 \end{pmatrix}$

(e) $\left\| \begin{matrix} 1.3 \\ 5.6 \end{matrix} \right\|$

(f) $\begin{pmatrix} 8 \\ 3 \end{pmatrix} + \begin{pmatrix} 3 \\ 5 \end{pmatrix}$

(g) $\begin{pmatrix} 8 \\ 3 \end{pmatrix} - 4 \begin{pmatrix} 2 \\ 1 \end{pmatrix}$

(h) $4 * \left[\begin{pmatrix} 2 \\ 7 \end{pmatrix} + \begin{pmatrix} 4 \\ -3 \end{pmatrix} \right]$

2. What is the length of the following vectors?

(a) $\begin{pmatrix} 4 \\ 3 \end{pmatrix}$

(b) $\begin{pmatrix} 2 \\ -3 \end{pmatrix}$

(c) $5 * \begin{pmatrix} 1 \\ 3 \end{pmatrix}$

(d) $\left[\begin{pmatrix} 3 \\ 2 \end{pmatrix} + \begin{pmatrix} 2 \\ -1 \end{pmatrix} \right]$

3. Normalise the following vectors.

   (a) $\begin{pmatrix} 4 \\ 3 \end{pmatrix}$

   (b) $\begin{pmatrix} 7 \\ 3 \end{pmatrix}$

   (c) $\begin{pmatrix} -4 \\ 8 \end{pmatrix}$

   (d) $5 * \begin{pmatrix} 2 \\ -3 \end{pmatrix}$

4. What is the vector between $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 3 \\ 7 \end{pmatrix}$?

5. How far is position $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$ from $\begin{pmatrix} 2 \\ 8 \end{pmatrix}$?

6. If you start at a position $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$ and travel with a velocity $\begin{pmatrix} 0.2 \\ 0.1 \end{pmatrix}$ for 5 seconds, what is your final position?

7. If my position is $\begin{pmatrix} 8 \\ 5 \end{pmatrix}$ what direction is the vector $\begin{pmatrix} 14 \\ 3 \end{pmatrix}$ from me?

8. If I start at $\begin{pmatrix} 3 \\ 7 \end{pmatrix}$ and walk towards $\begin{pmatrix} 23 \\ 18 \end{pmatrix}$ at a constant speed of 1.5, what is my velocity?

9. Calculate the following dot products. What is the angle between each set of vectors?

   (a) $\begin{pmatrix}4\\1\end{pmatrix} \cdot \begin{pmatrix}1\\2\end{pmatrix}$

   (b) $\begin{pmatrix}5\\9\end{pmatrix} \cdot \begin{pmatrix}0\\6\end{pmatrix}$

   (c) $\begin{pmatrix}2\\4\end{pmatrix} \cdot \begin{pmatrix}2\\4\end{pmatrix}$

   (d) $\begin{pmatrix}1\\2\\8\end{pmatrix} \cdot \begin{pmatrix}3\\6\\4\end{pmatrix}$

10. *Processing*, the programming language we are using in this course, has a class PVector for representing vectors. Look up the documentation and see how you can perform the operations we have described so far.

2.7 Matrices

Matrices are fundamental to computer graphics as they are used to perform transformations on objects. You should ensure that you are familiar with the following matrix calculations. The next section will describe how to represent transformations as matrices.

- Definition of a matrix:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix}$$

- Transpose:

$$m_{ij}^T = m_{ji}$$

- Multiplication by a scalar:

$$a \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} am_{11} & am_{12} \\ am_{21} & am_{22} \end{bmatrix}$$

- Multiplication of two matrices

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} n_{11} & n_{12} \\ n_{21} & n_{22} \end{bmatrix} = \begin{bmatrix} m_{11}n_{11} + m_{12}n_{21} & m_{11}n_{12} + m_{12}n_{22} \\ m_{21}n_{11} + m_{22}n_{21} & m_{21}n_{12} + m_{22}n_{22} \end{bmatrix}$$

- Multiplication of a vector by a matrix:

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{11}v_1 + m_{12}v_2 \\ m_{21}v_1 + m_{22}v_2 \end{bmatrix}$$

- Associativity of matrix multiplication:

$$\mathbf{L(MN)} = \mathbf{(LM)N}$$

- Non-commutativity of matrix multiplication:

$$\mathbf{MN} \neq \mathbf{MN}$$

You should be familiar with all of these operations. If you do not feel comfortable with any of the following exercises please revise these topics in an introductory mathematics textbook.

2.8 Transforms

Matrices can be used to define transforms. In this section we describe how to define a number of different transforms.

Identity

The simplest transform is the one that leaves a vector unchanged, called the **identity matrix**. This is a matrix with 1s along the main diagonal and 0s elsewhere:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z \\ 0x + 1y + 0z \\ 0x + 0y + 1z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Scale

If, instead of 1s along the diagonal, we have a different scalar a we get a matrix that is equivalent to multiplying the vector by a:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + 0y + 0z \\ 0x + ay + 0z \\ 0x + 0y + az \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az \end{bmatrix}$$

This will **scale** the vector by a, it will become larger if a is greater than 1 and smaller if a is less than 1.

If we have different values for each element on the diagonal, we have a **nonuniform scale**: namely, a scale in which each direction is scaled differently resulting in a squashing or stretching, not just a change in size:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + 0y + 0z \\ 0x + by + 0z \\ 0x + 0y + cz \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \end{bmatrix}$$

Rotation

To derive a rotation matrix, we can begin in 2D and by rotating a vector with only an x component $(x, 0)$.

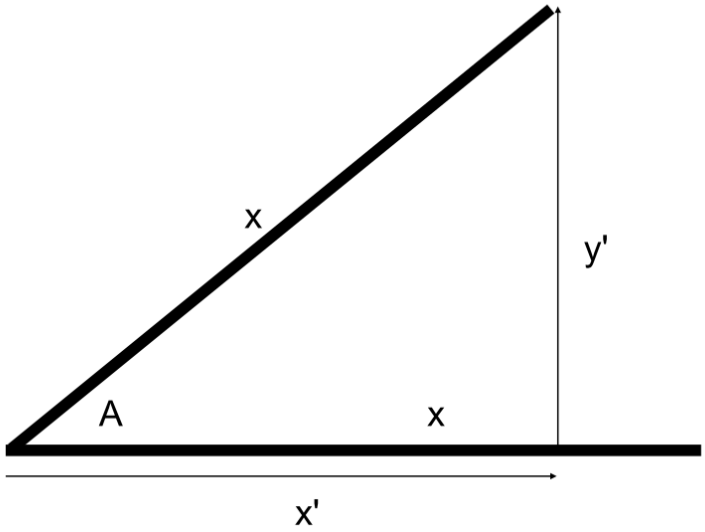


Figure 2.1: Rotating a vector $(x, 0)$.

As we can see from Figure 2.1, when the vector $(x, 0)$ is rotated by θ it still has length x but it has new coordinates (x', y'). x, x' and y' form a right angled triangle, so:

$$\cos(\theta) = \frac{x'}{x}$$

This allows us to calculate x':

$$x' = x\cos(\theta)$$

Similarly for y'

$$\sin(\theta) = \frac{-y'}{x}$$

The minus sign is there because we are using a coordinate system in which y points down, but y' points up. This gives us:

$$y' = -x\sin(\theta)$$

Thus the transformed point is $(x\cos(\theta), -x\sin(\theta))$

We can do the same for a point $(0, y)$ as in Figure 2.2.

$$\cos(\theta) = \frac{y'}{y}$$

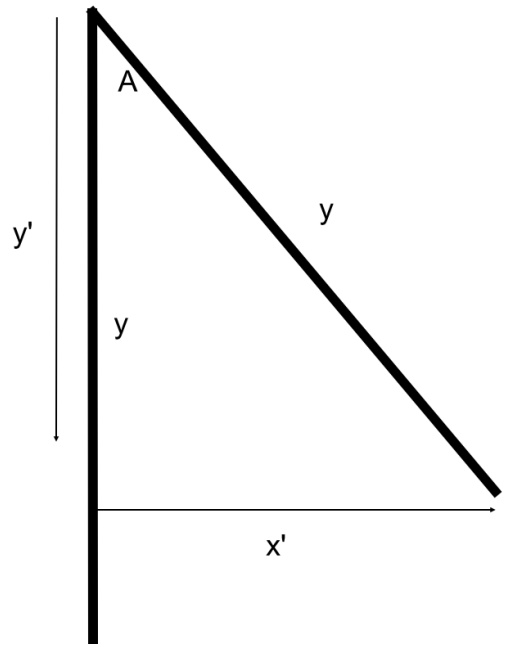


Figure 2.2: Rotating a vector $(0, y)$.

$$\sin(\theta) = \frac{x'}{y}$$

giving:

$$x' = y \sin(\theta)$$

$$y' = y \cos(\theta)$$

To rotate a vector with components about both axes (x, y), we can combine these equations:

$$x' = x \cos(\theta) + y \sin(\theta)$$

$$y' = y \cos(\theta) - x \sin(\theta)$$

The above equations give us the relationship between the untransformed values position vector (x, y) and the transformed vector (x', y'). This is a general linear transform that can be used to rotate any 2D vector. As (x', y') is just a linear combination of the untransformed coordinates x and y we can write it as a matrix:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos(\theta) + y\sin(\theta) \\ -x\sin(\theta) + y\cos(\theta) \end{bmatrix}$$

Moving to 3D we can define a rotation about the z axis that has the same effect on the x and y coordinates as a 2D rotation and we can use a similar matrix:

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

Similarly, we can define rotations about the x and y axes:

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

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

Any 3D rotation can be created out of a combination of rotations about the three axes. As the combination of rotations can be quite complex (particularly given that matrix multiplication is non-commutative) most graphics engines restrict rotations to three rotations about each of the axes x, y, z. In some cases this representation can also be problematic, particularly for animation, and we use a representation called *Quaternions*; these will be mentioned briefly in a learning activity in Chapter 8, but are beyond the scope of this subject guide.

Translation

Translation is equivalent to adding a vector (t_x, t_y, t_z) to a position vector **p** to create a new position $\mathbf{p}'$:

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

A 3D translation cannot be performed by a 3x3 transform matrix acting on a 3D vector. However, we can do a translation by using a trick. We add a fourth component to our vector:

$$\mathbf{v} = (x, y, z, 1)$$

This fourth component is, in most cases, 1. This new 4D coordinate system is called **homogeneous coordinates**.

Now we can create a translation matrix for translation **t** by using the 4x4 identity matrix but adding the components (t_x, t_y, t_z) to the fourth column:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can see that this has the correct effect by performing the multiplication:

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

Translations should only apply to position vectors because direction vectors are independent of position. We can easily implement this in homogeneous coordinates by making the fourth component of direction vectors 0. This means that translation matrices will not affect them.

Combining transforms

Transform matrices can be combined together by multiplying them together (remembering that order matters in matrix multiplication). For example, we can perform a scale, rotation and translation on a vector like this (all matrices are now in homogeneous coordinates):

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax\cos(\theta) + ay\sin(\theta) + t_x \\ -ax\sin(\theta) + ay\cos(\theta) + t_y \\ az + t_z \\ 1 \end{bmatrix}$$

The transform nearest the vector is applied first: that is, the rightmost one. In this case it is the scale. Transforms can therefore be read right to left, in the opposite order we would expect from reading English. Transforms in programming languages have a similar backwards effect with the last one to be called applied first to any object.

The order is important as we can see by swapping the translation and rotation.

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} (ax + t_x)\cos(\theta) + (ay + t_y)\sin(\theta) \\ -(ax + t_x)\sin(\theta) + (ay + t_y)\cos(\theta) \\ az + t_z \\ 1 \end{bmatrix}$$

The rotation is now applied to the translated coordinates. This could have a radically different end result.

Learning activity

Attempt the following problems. If you are unable to answer a significant number of them we recommend you consult a textbook as described in the Essential and Recommended reading sections of this chapter.

1. Calculate the dimension of the following matrices.

   (a) $\begin{bmatrix} 4 & 2 \\ 3 & 4 \end{bmatrix}$

   (b) $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$

   (c) $\begin{bmatrix} 2 & -3 \end{bmatrix}$

   (d) $\begin{bmatrix} 1 & 3 \\ 3 & 7 \\ 1 & 4 \end{bmatrix}$

   (e) $\begin{bmatrix} 1 & 3 & 4 & 9 \\ 3 & 7 & 4 & 1 \end{bmatrix}$

2. Evaluate the following matrix expressions.

   (a) $\begin{bmatrix} 4 & 2 \\ 3 & 4 \end{bmatrix}^T$

   (b) $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 3 \end{bmatrix}^T$

   (c) $2 * \begin{bmatrix} 4 & 1 & 1 \\ 3 & 1 & 2 \end{bmatrix}$

   (d) $3 * - \begin{bmatrix} 1 & 3 \\ 3 & 1 \\ 0 & 2 \end{bmatrix}^T$

3. Calculate the result of the following matrix multiplications.

(a) $\begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

(b) $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

(c) $\begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$

(d) $\begin{bmatrix} 1 & 0 & 2 \\ 4 & 5 & 4 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 0 & 2 & 1 \end{bmatrix}$

4. What transforms do these matrices perform?

(a) $\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(b) $\begin{bmatrix} 0.7 & 0.7 & 0 & 0 \\ 0.7 & 0.7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(c) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(d) $\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(e) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(f) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(g) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

5. Give matrices for the following transformations.

   (a) A 2D rotation of 30°

   (b) A 3D rotation of 45° about the z axis

   (c) A 3D rotation of 60° about the y axis

   (d) A 3D rotation of 180° about the x axis followed by a uniform scale of 1.5.

2.9 Summary

This chapter has summarised the key mathematics that is needed for computer graphics. If you are not familiar with the concepts covered, please revise them before continuing with the course.

2.10 Sample examination questions

1. State which of these is a rotation matrix, a scale matrix and a translation matrix?
   (a)
   $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
   (b)
   $$\begin{bmatrix} 0.93 & 0 & -0.36 & 0 \\ 0 & 1 & 0 & 0 \\ 0.36 & 0 & 0.93 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
   (c)
   $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
2. The shoulder, elbow and wrist joint of a skeleton have the following transform matrices given below. What is the position of the wrist?
   Shoulder:
   $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
   Elbow:
   $$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Wrist:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.707 & 0.707 & 0 \\ 0 & 0.707 & -0.707 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Calculate a matrix that performs a translation by the vector $(3, 2, 4)$ followed by a rotation of 60° about the z axis.

Chapter 3

From model to screen

3.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapters 3 and 4.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapters 3 and 7.

3.2 Recommended reading

Hughes, J. et al. *Computer graphics: principles and practice.* (2013), Chapter 6 and Chapter 15.

Rath, E. 'Coordinate systems in OpenGL' ; `www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl` (accessed May 2014), 2014.

3.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the key components of the rasterisation process
- identify different coordinate systems
- explain the concept of projection
- describe and explain the main algorithms involved in creating a 2D image from a 3D scene.

3.4 Introduction

Everything you see on your computer's screen, from text to pictures, is simply a two-dimensional grid of **pixels**, a term which comes from the words '**pic**ture **el**ement'. Every pixel on your screen has a particular colour (colour is discussed in more detail in Chapter 5 of this subject guide). This two-dimensional array of pixels is an image: a **raster** of pixels. The raster is organised into rows and each row holds a number of pixels. The quality of an image is related to the accuracy of the pixel colour value and the number of pixels in the raster (Figure 3.1).

In order to display an image on screen we simply need to choose what colour each pixel will be. The more accurate our colour choice, the more faithful the image. Displays usually index pixels in an ordered pair (x, y) indicating the row and column

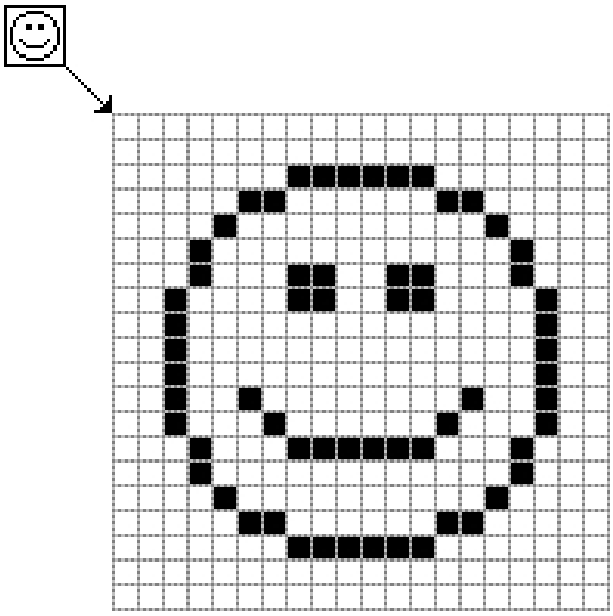


Figure 3.1: Example of a raster image. When enlarged, individual pixels appear as squares.

number of the pixels. If a display has n_x and n_y rows of pixels, the bottom-left pixel is (0,0) and the top-right is $(n_x - 1, n_y - 1)$. Figure 3.2 shows this. (Note: in some cases the top-left pixel has the coordinates (0,0) as it is the convention for television transmission.)

Objects are the 3D models that exist (mathematically-described in 3D space) in their own coordinate systems; images are the 2D realisations of objects which are displayed on screen. Converting 3D information for display on a 2D screen involves a process known as **rasterisation**. Rasterisation algorithms take a 3D scene described as polygons, and render it as discrete 2D primitives – using shaded pixels that convince you that the scene you are looking at is still a 3D world rather than a 2D image. It is the process of computing the mapping from scene geometry to pixels. The term rasterisation is derived from the fact that an image described in a vector graphics format (shapes) is converted into a raster image (pixels or dots).

3.5 Geometric models

A computer-generated image begins with a 3D geometric model. 3D models are widely used anywhere in 3D graphics. The geometric models themselves may have been created in a modelling package, or may be the automatically generated output from a data capture device such as a laser scanner, or from techniques such as 3D scene reconstruction from images. These models describe 3D objects using mathematical primitives such as spheres, cubes, cones and polygons. The most common type is a triangle or quadrilateral mesh composed of polygons with shared vertices. Hardware-based rasterisers use triangles because all of the lines of a triangle are guaranteed to be in the same plane, which makes the process less complicated. The polygon mesh defines the outer surface of the object. A model or scene made in this manner is known as a **wireframe**. Figure 3.3 shows an object created from triangle mesh.

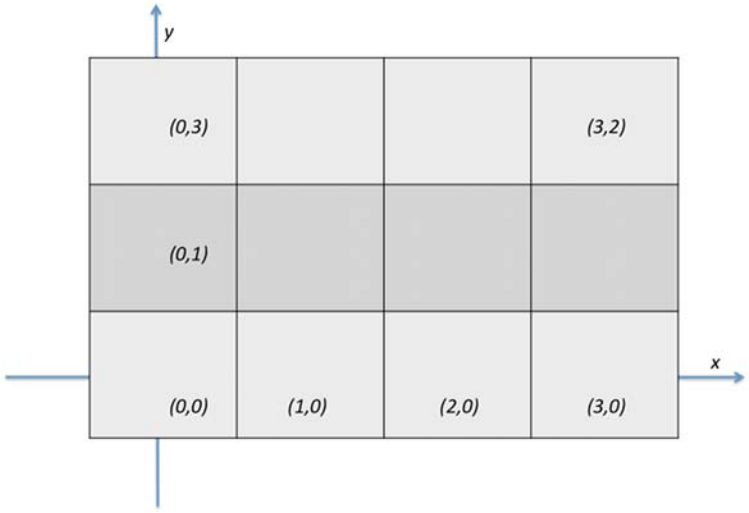


Figure 3.2: An example of a 4x3 pixel screen.

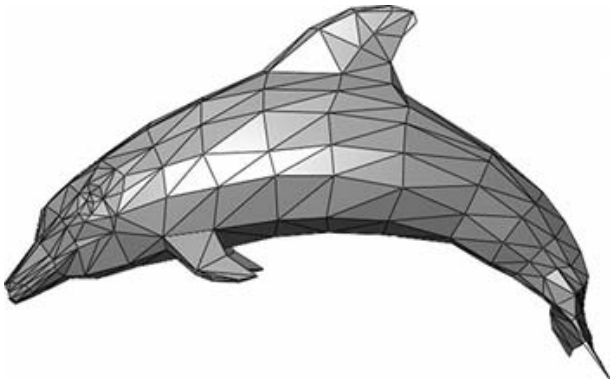


Figure 3.3: Example of a triangle mesh representing a dolphin.
Source: By Chrschn, from `http://en.wikipedia.org/wiki/Triangle_mesh#mediaviewer/File:Dolphin_triangle_mesh.png`

Chapter 2 dealt with the mathematics involved in transforms – manipulations such as translate, rotate and scale that are performed on objects within a modelled scene or environment. As mentioned, we have objects in an arbitrary 3D space which then need to be mapped onto a 2D screen. We need **coordinate systems** for vectors and transforms to make sense.

Learning activity

Computer graphics researchers use test models as a basis for evaluating graphics algorithms and techniques. These are 3D geometric models that are regarded as common, standard tests, as a baseline for evaluation. Explore these at `en.wikipedia.org/wiki/List_of_common_3D_test_models` (available May 2014). Why do you think an object such as the Utah teapot makes a good test model?

3.6 Coordinate systems

3D computer graphics rely on a number of coordinate systems. These include:

- World
- Object
- Local
- Viewpoint
- Screen.

The primary frame is the **world coordinate system** (WCS), also known as the **universe** or **global** or sometimes **model** coordinate system. This is the base reference system for the overall model (generally in 3D), to which all other model coordinates relate. Usually a model is built in its own **object coordinate system** (OCS), and later this model is placed into a scene in the WCS.

When each object is created in a modelling program, a point must be picked to be the origin of that particular object, and the orientation of the object to a set of model axes. Vertices are defined relative to the specific object. For example, when modelling a car, the modeller might choose a point in the centre of the car for the origin, or the point in the back of the car, or the front left wheel. When this object is moved to a point in the WCS, it is really the origin of the object (in the OCS) that is moved to the new world coordinates, and all other points in the model are moved by an equal amount. Figure 3.4 shows the WCS and OCS.

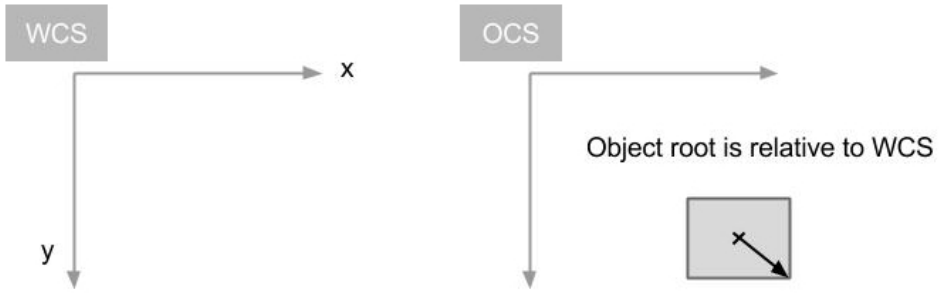


Figure 3.4: Left: the World Coordinate System; Right: The Object Coordinate System. Note that when the object moves, it does so relative to the WCS.

Local coordinate systems (LCS) are a further subdivision. For example,when modelling a car the centre of each wheel can be described with respect to the car's coordinate system, but each wheel can be specified in terms of a local coordinate system (Figures 3.5–3.6). This way, the information describing each wheel can be simply duplicated four times.

Sometimes objects in a scene are arranged in a hierarchy, so that the 'position' of one object in the hierarchy is relative to its parent in the hierarchy scheme, rather than to the world coordinate system. This would be useful in the case of the car model as

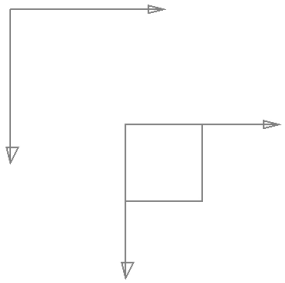


Figure 3.5: (a) In a local coordinate system, a 'forward' vector is always the same relative to the object.

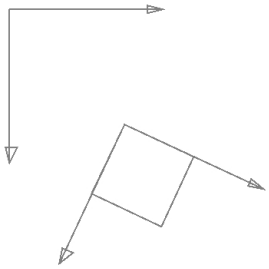


Figure 3.6: (b) This allows for rotations and is still able to apply relative translations and have the object moving 'forwards'.

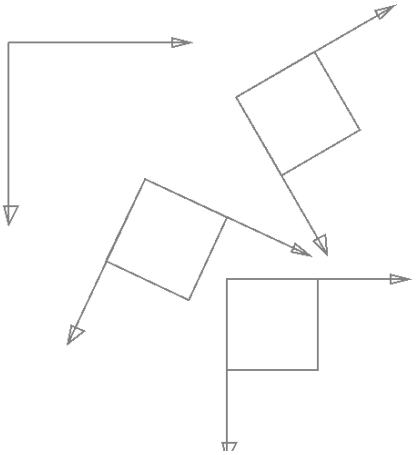


Figure 3.7: (c) An LCS can have many objects, each with their own coordinate system.

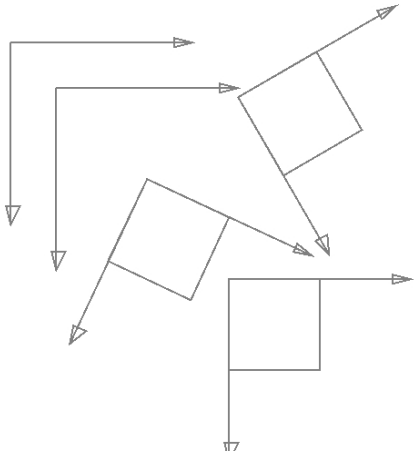


Figure 3.8: (d) These may be grouped, defining a hierarchy of relative coordinate systems.

independent transformations (for example, steering rotation) could be carried out. Figures 3.7–3.8 demonstrate the hierarchical aspects of the coordinate systems.

In addition to the coordinate systems that place an object within a scene, there is also the **viewpoint coordinate system**, also known as the **camera coordinate system**. This is based upon the viewpoint of the observer, and changes as they change their view. Moving an object 'forward' in this coordinate system moves it along the direction that the viewer happens to be looking at the time.

Finally, there is the **screen coordinate system**.This is a 2D coordinate system which refers to the physical coordinates of the pixels on the computer screen, based on current screen resolution (for example, 1024x768). It is, essentially, the grid of pixels that we see.

3.7 Projections

A display device (such as a screen) is generally 2D, so how do we map 3D objects to 2D space? This is a question that artists and engineers alike over the centuries have had to contemplate. The answer is that we need to transform 3D objects on to a 2D plane using **projections**.

The two basic types of projections – **perspective** and **parallel** – were designed to solve two mutually exclusive demands: showing an object as it looks in real life, and preserving its true size and shape.

Perspective projection: The visual effect of perspective projection is similar to the human visual system: it has perspective foreshortening, in that the size of an object varies inversely with distance. Even wireframe objects look 'real' if perspective is used – that is, distant objects look smaller. A feature of perspective drawings is that sets of parallel lines appear to meet at a vanishing point – like train tracks running into the distance.

Parallel projection: Parallel projection is a less realistic view as there is no foreshortening; however, parallel lines remain parallel. Parallel projection methods are used by architects and engineers to create working drawings that preserve scale and shape, such as orthographic projections where all the projection lines are orthogonal to the projection plane. This is useful because angle and distance measurements can be made.

Basic principles

In this course we are most interested in perspective projection as this is what is required to make our modelled scenes look 'real', enhancing realism by providing a type of **depth cue**. Projection from 3D to 2D is defined by straight projection rays (projectors) emanating from the **centre of projection**, passing through each point of the object, and intersecting the **projection plane** to form a projection (Figure 3.9).

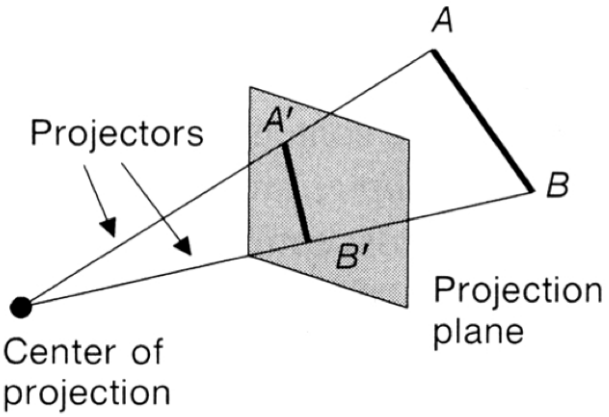


Figure 3.9: Projectors intersect the projection plane to form a projection.

The actual part of the scene in world coordinates that is to be displayed is called a **window**. On a screen, the picture inside this window is mapped onto the **viewport** – the available display area (Figure 3.10).

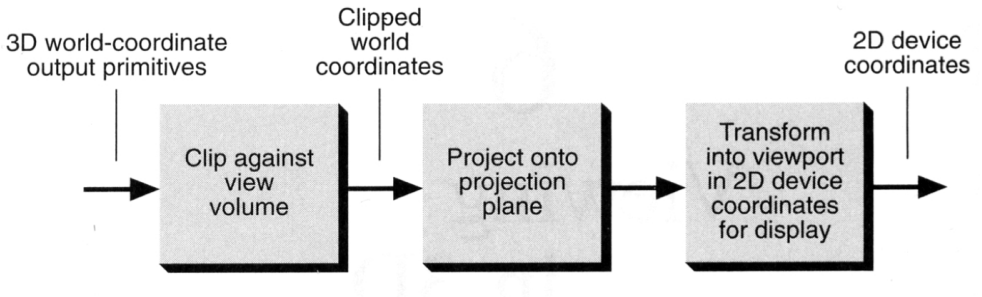


Figure 3.10: The world coordinates need to be clipped to the viewing window and ultimately mapped to the viewport and converted into device coordinates.

Learning activity

Read through the Art Institute of Chicago's summary of the lecture by James Elkins 'What is perspective?' and watch the accompanying video:
`www.artic.edu/aic/education/sciarttech/2di.html` (accessed May 2014).

3.8 Basic drawing actions

All drawing functions simply place pixel values into memory; that is, they set a pixel on the screen to a specific colour:

```
SetPixel{x,y,R,G,B}
```

with x and y representing the coordinates of the pixel and R, G and B, the colour value (see Chapter 5 of this subject guide).

From this simple concept springs the basis of computer graphics: colour the correct pixels in the correct manner to achieve the desired image.

Line drawing

Line drawing involves taking two endpoints in screen coordinates and drawing a line between them. For general screen coordinates (x_0, y_0) and (x_1, y_1) the algorithm should draw a set of pixels that approximates a line between them. Drawing vertical and horizontal lines is straightforward: it is simply a matter of setting the pixels nearest the line endpoints and all the pixels in between. However, when the line is non-vertical or non-horizontal, it becomes more problematic. Pixels are not points so we have to find the pixel that is closest to the actual point (Figure 3.11).

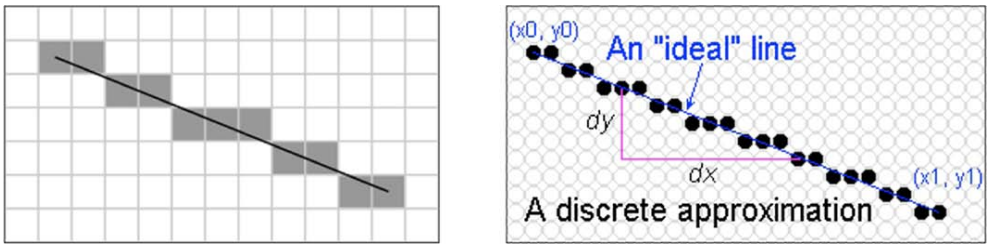


Figure 3.11: Points do not equate to pixels. To give the impression of a continuous line on a screen, the pixels that most closely approximate the mathematical line should be set to the colour of that line.

Line drawing needs to be fast and as simple as possible, giving a continuous appearance. Commonly used procedures are the **Bresenham** algorithm and the **midpoint** algorithm. These differ slightly in their methods but produce the same lines; Bresenham's algorithm can be considered an efficient form of the midpoint algorithm. Both of these algorithms extend and simplify the **Digital Differential Analyzer (DDA) algorithm** (which uses floating points and is slower, less efficient and less accurate). The algorithm works incrementally from a starting point (x_0, y_0) and the first step is to identify in which one of eight octants the direction of the end point lies (Figure 3.12). Bresenham's algorithm has been the basis for line generation since its inception in 1965. It is scaled so that we work with integers as this is much faster and more efficient.

In the case of the 1st octant, the line is drawn by stepping one horizontal pixel at a time from x_0 to x_1 and at each step making the decision whether or not to step +1 pixel in the y direction.

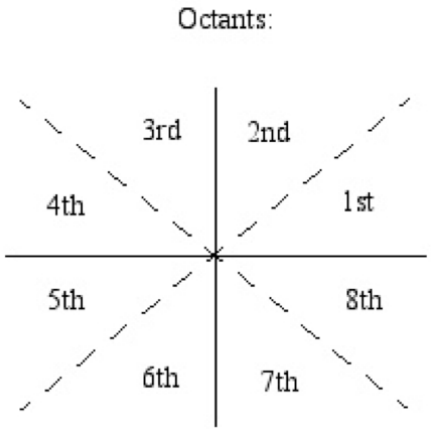


Figure 3.12: Octants.

The process is as follows:

- Draw the pixel at (x, y) making sure it is the closest pixel to the point that belongs on the line.
- Move across the x-axis and decide at each step what will be the next point on the y-axis to draw.

- Since the point on the line may not be in the centre of the pixel we must find the second best point – the one where the distance from the line is the smallest possible.
- Decision: do we draw $(x + 1, y)$ or do we draw $(x + 1, y + 1)$?
  $d1$ is smaller than $d2$ so the next point after (x, y) that should be drawn is $(x + 1, y + 1)$.
- Every iteration we increment the x coordinate and calculate whether or not we should increment the y coordinate.

This works for simple positive line slopes where the slopes are less than 45°, which corresponds to the first octant. For other octants, it is simply a matter of using the appropriate increment/decrement.

Circles

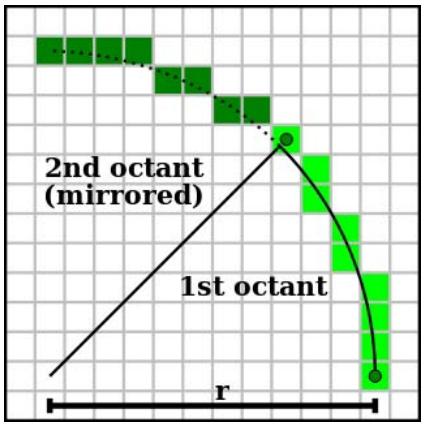


Figure 3.13: Rasterisation of a circle using the Bresenham algorithm.
Source: By Perey reproduced under Creative Commons licence CC BY-SA 3.0 from `http://en.wikipedia.org/wiki/Midpoint_circle_algorithm#mediaviewer/File:Bresenham_circle.svg`

Bresenham's circle algorithm – derived from the midpoint circle algorithm – takes advantage of a circle's symmetry to plot eight points, reflecting each calculated point around each 45° axis. Figure 3.13 shows the rasterisation of a circle beginning with a point in the first octant and proceeding anticlockwise. As with the line-drawing algorithm, the procedure is incremental and at each step one of two points are chosen by means of a decision variable d, where the best approximation of a true circle is described by the pixels that fall the least distance from the true circle. Given the calculation of the arc in the first octant, those pixels can be reflected to get the whole circle. This is computed by specifying the coordinates as follows:

1st octant: x, y

2nd octant: y, x

3rd octant: $-y, x$

4th octant: $-x, y$

5th octant: $-x, -y$

6th octant: $-y, -x$

7th octant: $y, -x$

8th octant: $x, -y$

The above algorithm draws a complete circle. For a circular arc, the coordinates of the two ends of the arc, together with the radius, are required. The algorithm is run over the complete octant or circle and sets the pixels only if they fall into the desired interval.

Curves

A curve is made up of a number of points that are related by some function. Any point on the curve has two neighbours, except for the endpoints which have only one neighbour. (Some curves can be infinite, or are closed, in which case they have no endpoints.) In computer graphics terminology, a **spline** is a curve that connects two or more specific points: the term comes from mechanical drafting where a flexible strip was used to trace a curve. In rasterizing a curve, the challenge is to construct a spline from some given points in the plane. A natural way to construct a curve from a set of given points is to force the curve to pass through the points, or interpolate the points. A spline curve is usually approximated by a set of short line segments. It is rare that a single function can be constructed to pass smoothly through all given points; therefore, it is generally necessary to use a series of curves end-to-end.

In 1962, Pierre Bézier came up with a way of constructing a curve based on a number of control points. A Bézier curve is a polynomial curve that **approximates** its control points (often known as **knots**), coming close to each of the points in a principled and smooth fashion. It is one of the most-used methods for free-form curves in computer graphics as the mathematical descriptions are compact and easy to compute and they can be chained together to represent many different shapes.

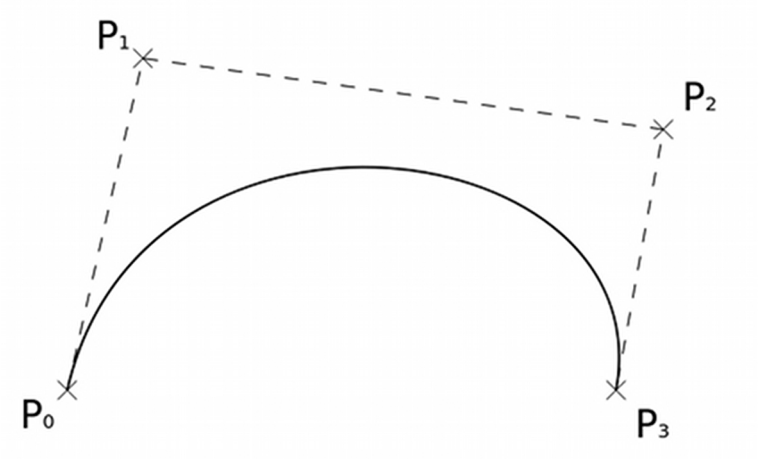


Figure 3.14: A cubic Bézier curve has four control points.

In theory, Bézier curves can be constructed for any number of points, but four control points (a cubic Bézier) are commonly used because increasing the control points leads to polynomials of a very high order, making things computationally inefficient. The curve is defined by four points: the initial position, P_0, and the terminating position, P_3, and two separate middle points, P_1 and P_2: the curve itself

only passes through the first and last points (Figure 3.14). The first and last control points interpolate the curve; the rest approximate the curve. The shape of a Bézier curve can be altered by moving the middle points.

We can describe a Bézier curve by a function which takes a parameter t as a value. The value t ranges from 0 to 1, where 0 corresponds to the start point of the curve and 1 corresponds to the endpoint of the curve. We can think of t as the time taken to draw a curve on a piece of paper with a pen from start to finish: $t = 0$ when the pen first touches the paper, and $t = 1$ when we have drawn the curve. Values in between correspond to other points on the curve. The formula for cubic Bézier curves is:

$$[x, y] = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3$$

The direction of the curve as it leaves P_0 is towards the second point (P_1) and the direction the curve approaches the end point is from the penultimate point (P_2).

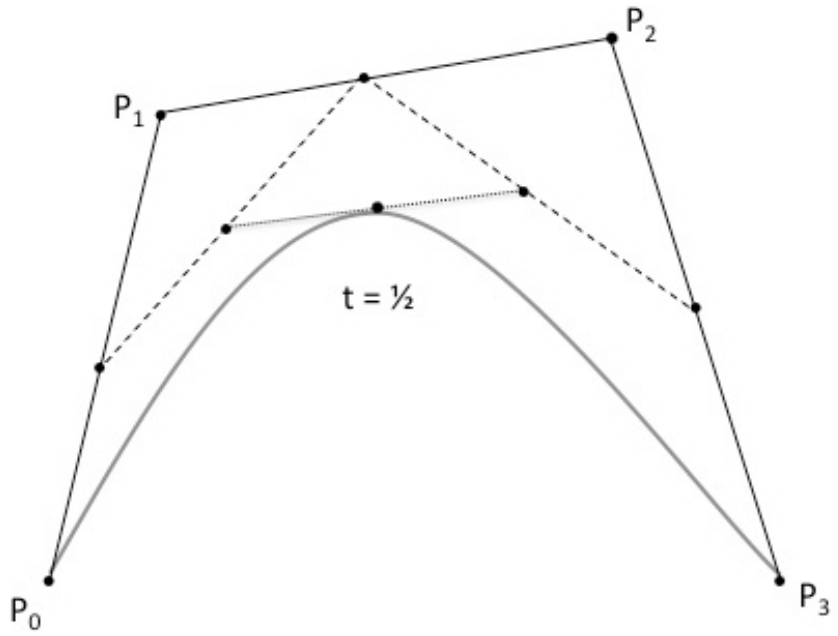


Figure 3.15: An illustration of the recursive de Casteljau algorithm for a cubic Bézier showing the construction for $t = 0.5$.

A convenient method for drawing Bézier curves is to use a recursive procedure via the **de Casteljau** algorithm (Figure 3.15), using a sequence of linear interpolations to compute the positions along the curve. The steps are as follows:

- Start at the beginning and end points, P_0 and P_3.
- Calculate the halfway point (in other words, at the first pass the halfway point will be t=0.5).
- If the angle between, formed by the two line segments, is smaller than a threshold value, then add that point as a drawing point. Now recursively repeat

with each half of the curve. Stop the algorithm when no more division is possible, or the line segments reach a minimal length.

Bézier curves are also used in animation as a tool to control motion, as discussed in Chapter 8 of this subject guide.

Learning activity

Jason Davies' animated curves web page allows you to drag the control points to modify the curves – try it out at `http://www.jasondavies.com/animated-bezier/` (accessed May 2014).

3.9 2D polygons

Once lines and curves have been drawn on the screen there are a sequence of operations that take place in order to portray polygons on the screen. These operations involve getting rid of back-facing polygons (**culling**) that the viewer will not see, removing polygons outside of the viewing volume (**clipping**), and applying **hidden surface removal** so that only those forward-facing surfaces that are visible to the viewer will be drawn. Once the visible faces are determined, the polygon must be **filled** in some way; that is, the pixels within the visible faces need to be set to a given colour.

Culling and clipping

When displaying images on the screen, we want to restrict polygons to what will actually appear within the field of view and the confines of the screen. The default clipping rectangle is the full canvas (the screen), and it is obvious that we cannot see any graphics primitives that lie outside the screen. It is rarely necessary to display the whole of a scene in its entirety.

Culling refers to discarding any complete polygons that lie outside the clip rectangle. It is a relatively quick way of deciding whether to draw a triangle or not. It involves removing any polygons behind the projection plane, removing any polygons projected to lie outside the clip rectangle, and culling any back-faces (that is, any polygon that faces away from the viewport). A polygon is back-facing if its normal, N, is facing away from the viewing direction, V. If a polygon is facing away from the camera and is part of a solid object, then it can't be seen. Consider a cube: at any one time three of the sides of the cube will face away from the user and therefore will not be visible. Even if these faces were drawn they would be obscured by the three 'forward' facing sides. Back-face culling therefore reduces the number of faces drawn from twelve to six. Removing unseen polygons can give a big increase in speed for complex scenes. Processing supports back-face culling internally, so it can be enabled to get this effect.

If a polygon lies partially outside the viewport then there is no need for the program to spend any CPU time doing any of the calculations for the part that is not going to be seen. **Clipping** is generally used to mean avoiding drawing anything outside the camera's field of view. To clip a line we only need to consider its endpoints. If both endpoints of a line lie inside the clip rectangle, the entire line lies inside the clip

rectangle and no clipping is required. If one endpoint lies inside and one outside, we must compute the intersection point. If both endpoints are outside the clip rectangle, the line may or may not intersect with the clip rectangle.

There are many methods that can be used to clip a line. The most widely-used algorithm for clipping is the **Cohen-Sutherland line-clipping algorithm**. It is fast, reliable, and easy to understand. It uses what is termed 'inside-outside window codes'. To determine whether endpoints are inside or outside a window, the algorithm sets up a half-space code for each endpoint. Each edge of the window defines an infinite line that divides the whole space into two half-spaces, the inside half-space and the outside half-space.

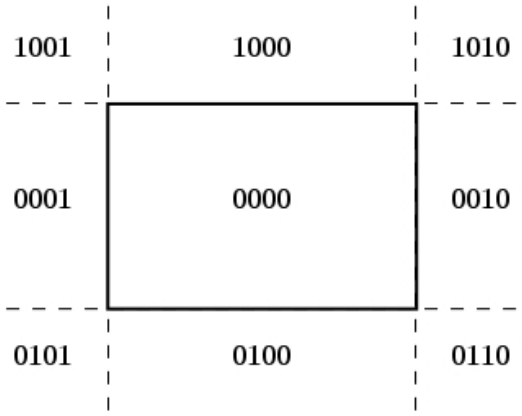


Figure 3.16: Cohen-Sutherland 'inside-outside window codes'.

As you proceed around the window, nine regions are created – the eight outside regions and the one inside region (Figure 3.16). Each of the nine regions associated with the window is assigned a 4-bit code to identify the region. Each bit in the code is set to either a 1(true) or a 0(false):
If the region is to the left of the window, the first bit of the code is set to 1.
If the region is to the right of the window, the second bit of the code is set to 1.
If to the bottom, the third bit is set, and if to the top, the fourth bit is set. The 4 bits in the code then identify each of the nine regions.

For any endpoint (x, y) of a line, the code can be determined that identifies the region in which the endpoint lies. The code's bits are set according to the following conditions:

- First bit set **1**: point lies to the **left** of the window, $x < x_{min}$
- Second bit set **1**: point lies to the **right** of the window, $x > x_{max}$
- Third bit set **1**: point lies below (**bottom**) of the window, $y < y_{min}$
- Fourth bit set **1**: point lies above (**top**) of the window, $y > y_{max}$

The sequence for reading the codes' bits is LRBT (Left, Right, Bottom, Top).

Once the codes for each endpoint of a line are determined, the logical AND operation of the codes determines if the line is completely outside of the window. If the logical AND of the endpoint codes is not zero, the line can be trivially rejected. For example, if an endpoint had a code of 1001 and the other endpoint had a code of 1010, the logical AND would be 1000 indicating the line segment lies outside of

the window. If endpoints had codes of 1001 and 0110, the logical AND would be 0000, and the line could not be trivially rejected.

The logical OR of the endpoint codes determines if the line is completely inside the window. If the logical OR is zero, the line can be trivially accepted. For example, if the endpoint codes are 0000 and 0000, the logical OR is 0000 – the line can be trivially accepted. If the endpoint codes are 0000 and 0110, the logical OR is 0110 and the line cannot be trivially accepted.

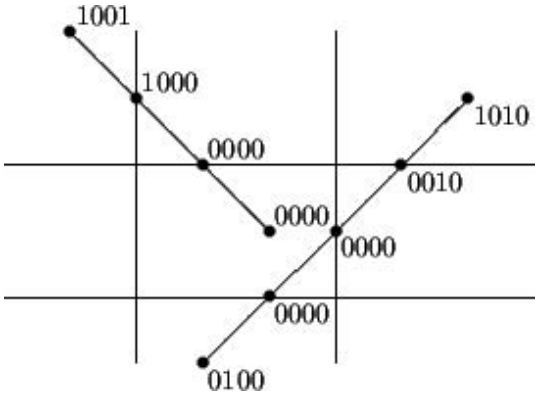


Figure 3.17: An example of the Cohen-Sutherland Line-Clipping Algorithm.

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivially accepted or rejected (Figure 3.17). If the line cannot be trivially accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted.

Hidden surfaces

When you want to make a wireframe more realistic you need to take away the edges you cannot see; that is, you need to make use of hidden line and hidden surface algorithms. The facets in a model determine what is and what is not visible.

In a wireframe drawing, all the edges of the polygons making up the model are drawn. A hidden surface drawing procedure attempts to colour in the drawing so that joins between polygons and polygons that are out of sight (hidden by others) are not shown. For example, if you look at a solid cube you can only see three of its facets at any one time, or from any one viewpoint.

To construct a hidden surface view, each polygon is projected onto the viewing plane. Instead of drawing the edges, pixels lying inside the boundary formed by the projected edges of a polygon are given an appropriate colour. This is not a trivial task. Given a typically high polygon count in a scene, any procedure to implement the filling task must be very efficient.

The painter's algorithm

This is the simplest of all the hidden surface rendering techniques, although it is not a 'true' hidden surface as it cannot handle polygons which intersect or overlap in certain ways. It relies on the observation that if you paint on a surface, you paint over anything that had previously been there, thus hiding it; namely, paint from furthest to nearest polygons.

```
sort the list of polygons by distance from viewpoint (furthest away at start of list)
repeat the following for all polygons in the ordered list:
      draw projected polygon
```

The advantage of the painter's algorithm is that it is very easy to implement for simple cases (but not so good for more complex surface topologies such as certain overlapping polygons or the presence of holes). However, it's not very efficient – all polygons are rendered, even when they become invisible.

The z-buffer algorithm

It is very impractical to carry out a depth sort among every polygon for every pixel. The basic idea of the z-buffer algorithm is to test the z-depth of each surface to determine the closest (visible) surface. The z-buffer stores values $[0..ZMAX]$ corresponding to the depth of each surface. If the new surface is closer than one in the buffers, it will replace the buffered values:

```
Declare an array z_buffer(x, y) with one entry for each pixel position.
Initialise the array to the maximum depth.
for each polygon P
     for each pixel (x, y) in P
          compute z_depth at x, y
               if z_depth < z_buffer (x, y) then
                    set_pixel (x, y, colour)
                    z_buffer (x, y) <= z_depth
```

The advantage of the z-buffer algorithm is that it is easy to implement, particularly in hardware (it is a standard in many graphics packages such as Open GL). Also, there is no need to sort the objects and no need to calculate object-object intersections. However, it is problematic as there is some inefficiency as pixels in polygons nearer the viewer will be drawn over polygons at greater depth.

BSP trees

The painter's algorithm is quite limited as time is wasted drawing objects that will be overdrawn later. However, the z-buffer algorithm also has its drawbacks as it is expensive in terms of memory use. **Binary Space Partitioning (BSP) trees** provide an elegant, efficient method for sorting polygons by building a structure that captures some relative depth information between objects. It involves partitioning a space into two (binary) parts using a separating plane. Polygons on the same side of that plane as the eye can obscure – but can **not** be obscured by – polygons on the other side. This is then repeated for both resulting subspaces, giving a BSP tree. BSP trees split up objects so that the painter's algorithm will draw them correctly without

need of a Z-buffer, and eliminates the need to sort the objects as a simple tree traversal will yield them in the correct order.

Filling

A wireframe drawing usually needs to be filled in some way and there are several ways of achieving this. The following methods consider the simple problem of filling a closed polygonal region with a single colour (as opposed to shading, which is dealt with in Chapter 5 of this subject guide).

Recursive seed fill

This is a simple but slow method. A pixel lying within the region to be filled is taken as a seed and set to record the chosen colour. The seed's nearest neighbours are found and, with each in turn, this procedure is carried out recursively. The process continues until the boundary is reached.

Ordered seed fill

A seed pixel is chosen inside the region to be filled. Each pixel on that row, to the left and right of the seed, is filled pixel by pixel until a boundary is encountered. Extra seeds are placed on the rows above and below and are processed in the same manner. This is also recursive but the number of recursive calls is dramatically reduced.

Scanline fill

The polygon is filled by stroking across each row and colouring any pixels on that row if they lie within the polygon:

```
remove any horizontal edges from the polygons
find the max and min y values of all polygon edges.
make a list of scanlines that intersect the polygon
for each scanline repeat
     for each polygon edge list the pixels where the edge intersects the scan line
          order them in ascending x value (p0, p1, p2, p3)
          step along the scanline filling pairwise (p0 --> p1, p2 --> p3)
```

This is the most useful filling procedure for a 3D polygon as it works very efficiently for simple convex shapes.

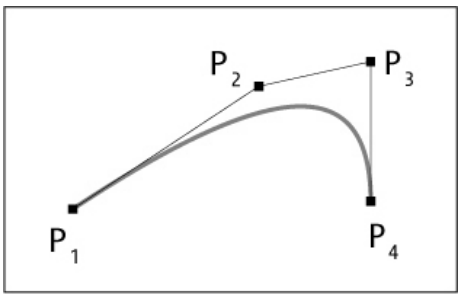
3.10 Exercises

1. Joshua Scott from the University of Canterbury has created an exercise with a practical walkthrough of linedrawing. Complete this exercise: `csunplugged.org/sites/default/files/activity_pdfs_full/Lines.pdf` (available May 2014).

2. Mike Kamermans has written a wonderfully detailed primer on Bézier curves using *Processing* examples. Explore this and work with his code, which is freely available to copy and manipulate: `http://pomax.github.io/bezierinfo/` (accessed May 2014).

3.11 Sample examination questions

1. Explain what is meant by the term **viewing coordinates**.
2. How many control points are required to specify a Bézier curve of degree d?



3. In the figure above, sketch how you can find the centre ($t = 0.5$) of the mapped Bézier curve using the de Casteljau algorithm.
4. What is a depth buffer and how is it used in hidden surface removal?
5. The z-buffer method requires a z-value to be stored for every pixel in the entire screen. In some situations, this memory requirement is prohibitive. Propose a method in which the z-buffer approach is used, but memory is allocated for only part of the screen. What additional problems arise?
6. Sketch a configuration of polygons for which the painter's algorithm fails. How can it be fixed?
7. Indicate why 3D clipping is needed in a practical renderer. Why is this especially important for video games?

Chapter 4

Graphics programming

4.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapters 2 and 8.

This chapter covers programming environments, which often develop rapidly. We will be using *Processing* for this course. The most up to date resources will be on the *Processing* website:

`www.processing.org/`

The site contains a number of useful tutorials, including one on shader programming. However, since these may change rapidly over time, we have not included direct urls, and recommend you check the site yourself.

4.2 Recommended reading

If you are not familiar with *Processing*, there are a number of good introductory textbooks. We would particularly recommend this one:

Shiffman, D. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (The Morgan Kaufmann Series in Interactive 3D Technology)*. (Elsevier Science, 2009) [ISBN 9780080920061].

GPU shader programming is a large topic and there are many excellent online resources on it. If you would like a book to deepen your understanding we would recommend:

Rost, R., B. Licea-Kane, D. Ginsburg, J. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen *OpenGL Shading Language.* (Addison-Wesley Professional., 2009) [ISBN 9780321637635].

4.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- write moderately complex graphics programs in *Processing*
- write programs using immediate mode graphics
- make use of high level graphics objects on top of immediate mode
- explain the principles of GPU shader programming
- write simple shader programs.

4.4 Introduction

This chapter will cover the basics of programming 3D graphics software. The principles taught in this chapter are common to most graphics programming environments; however, in order to do practical work we must choose one. In this course we will choose *Processing*, a dialect of Java that has excellent graphics support. *Processing* will be familiar to most students on the Creative Computing programmes. If you have not used *Processing* before, please download it and familiarise yourself with it. *Learning Processing* by Daniel Shiffman is an excellent introduction for those of you who have not used *Processing* before. The 3D graphics API in *Processing* is closely modelled on the most popular graphics standard OpenGL. Most of what is covered in this chapter applies to OpenGL which in most cases only differs in terms of syntax. The exception is section 4.9 which covers higher level representations of graphical objects very similar to those found in game or graphics engines such as 'Unity3D', 'OGRE' or the 'Unreal Engine'.

4.5 3D Graphics in *Processing*

Below is a basic two dimensional program in *Processing* that draws a rectangle in grey on a white background:

```
void setup(){
    size(640, 480);
}
void draw(){
    background(255);

    fill(100);

    rect(100, 100, 100, 200);
}
```

If you do not understand this program, please ensure you practise basic *Processing* programming.

Turning this program into a 3D one is fairly straightforward, but requires some changes, which are marked with comments below:

```
void setup(){
    size(640, 480, P3D); // use the P3D renderer
}

void draw(){
    background(255);
    lights(); // add lighting

    fill(100);

    box(100, 100, 100); // 3D primitive
}
```

The first change is to use a 3D rather than a 2D renderer (the software that creates images from your graphics commands). This is done by passing a third argument, `P3D`, to the size command.

Secondly, to view a 3D scene effectively requires lighting. We will cover lighting in detail in later chapters, but for the moment we will use the `lights` command which provides a standard set of lights.

Finally, rather than using a 2D primitive object such as a rectangle, we can use 3D primitives; in this case a box. The `box` command takes three parameters. These are the three dimensions of the box: width, depth and height, or to describe them in standard mathematical notation x, y, and z. This is probably the most obvious difference of writing 3D graphics; we must work with three rather than two dimensions. In most cases where 2D code would require us to specify an x and a y we must also specify a z for depth. Another difference is that the `box` command, unlike `rect` does not allow us to specify the position of the box. To move it around we need transformations, which are covered later in this chapter.

Learning activity

One major problem with the program listed above is that we cannot move around and view the box from different positions and directions, which is particularly problematic as it is in the top left corner of the screen and we cannot move it yet.

The course will teach you techniques that allow you to move around the scene, but the simplest way is to use a pre-existing camera control library to change your viewpoint. There are several camera libraries for *Processing*; at the time of writing, the easiest to use is called PeasyCam. Install PeasyCam or another 3D camera control library. Read the documentation and look at the examples so you understand how to use it. Then adapt the above example so that it uses your chosen library to control the camera.

4.6 Vertex shapes

In 3D graphics solid shapes are actually represented simply by their outer surface and in most cases this surface is represented as a number of polygons. A polygon is a flat shape made up of a finite number of straight sides. We can create a polygon by simply specifying the positions in 3D space of each of the corners where its sides meet. In 3D graphics we call these corner points **vertices** (singular **vertex**). Vertices are the basic building blocks of our 3D graphics shapes. In *Processing* we can make a shape out of vertices using the commands `beginShape`, `vertex` and `endShape`:

```
beginShape();
{
    vertex(100, 100, -100);
    vertex(200, 100, -150);
    vertex(200, 200, -200);
    vertex(150, 200, -150);
    vertex(100, 150, -100);
}
endShape();
```

Each call to `vertex` adds a new vertex to the shape, with its x, y and z position given by the three parameters.

This draws an open shape: the start and end point do not join up. To close the shape we can pass an argument, `CLOSE`, to `endShape`:

```
beginShape();
{
    vertex(100, 100, -100);
    vertex(200, 100, -150);
    vertex(200, 200, -200);
    vertex(150, 200, -150);
    vertex(100, 150, -100);
}
endShape(CLOSE);
```

Types of shape

The commands given above create a single polygon out of all of the vertices provided. However, most 3D shapes are made up of several polygons. Normally these polygons would only be triangles.

Triangles

Triangles are the simplest possible polygon having only three sides. That means that they cannot have some of the more complex configurations that other polygons can have. Firstly, they cannot be **concave**. Secondly, they are always **planar**, meaning they are always flat. All three of their vertices are always in the same plane. More complex shapes can be bent, with vertices in different planes. These two features can greatly simplify the graphics calculations required to render triangles and current graphics hardware is designed to work with triangles.

In *Processing* it is possible to make a shape out of multiple triangles by passing a parameter `TRIANGLES` to the `beginShape` command. If this is done, every set of three vertices is formed into a triangle as shown below:

```
beginShape(TRIANGLES);
{
    // triangle 1
    vertex(100, 100, -100);
    vertex(200, 100, -150);
    vertex(200, 200, -200);

    // triangle 2
    vertex(150, 200, -150);
    vertex(100, 150, -100);
    vertex(100, 200, -100);
}
endShape();
```

This will create a set of disconnected triangles, but in most cases we want a single surface made of joined triangles. Using the above commands this would require

considerable effort to make sure the triangles are joined together. In particular, we would need to ensure that each triangle shared vertices with the previous one.

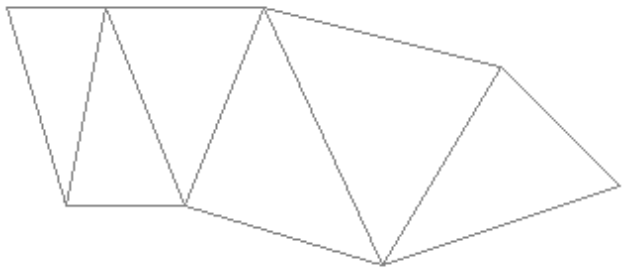


Figure 4.1: A triangle strip.

Triangle strips

A **triangle strip** is a type of shape that makes it more efficient to create shapes out of connected triangles, by ensuring that each triangle is automatically joined to the previous one. The first three vertices of a triangle strip are joined into a triangle, but after that each triangle is formed by taking the next vertex and joining it to the last two vertices of the previous triangle (see Figure 4.1).

```
beginShape(TRIANGLE_STRIP);
{
    vertex(100, 100, -100);  // triangle 1
    vertex(200, 100, -150);  // triangle 1 and 2
    vertex(200, 200, -200);  // triangle 1, 2 and 3
    vertex(150, 200, -150);  // triangle 2, 3 and 4
    vertex(100, 150, -100);  // triangle 3 and 4
    vertex(100, 200, -100);  // triangle 4
}
endShape();
```

Triangle fans

A **triangle fan** is similar to a triangle strip but uses the vertices in a different order. The first vertex is part of every single triangle. Every two vertices in turn are used to create a triangle together with the first vertex, with each vertex being joined to the last vertex of the previous triangle and the first vertex of the overall shape (see Figure 4.2).

Quads and quad strips

As well as triangles it is possible to create shapes out of quadrilaterals (**quads**), four sided shapes, and there is a type of shape called a **quad strip** which is analagous to a triangle strip. The quads are automatically split into triangles before rendering.

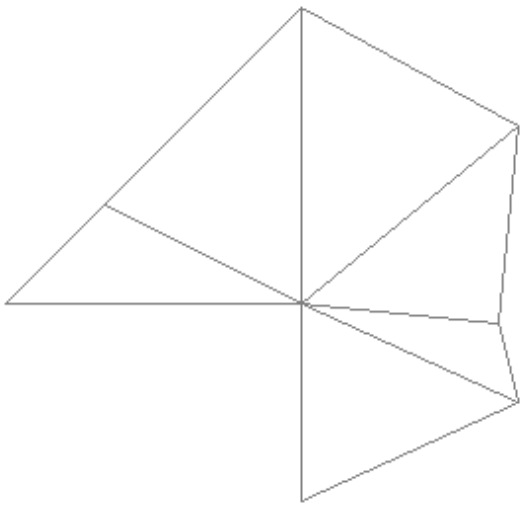


Figure 4.2: A triangle fan.

Learning activity

Write a program that uses triangles or quads to draw a cube.

4.7 Graphics state

We can change the style of a shape using commands such as `fill` which changes the colour of the body of a shape; `stroke` which changes the colour of the edges, and `strokeWeight` which changes the width of the edges:

```
fill(255, 0, 0); // make the shape red
strokeWeight(6); // make the edges thick
beginShape(TRIANGLE_STRIP);
{
    vertex(-100, -100, 50);
    vertex(100, -100, 0);
    vertex(100, 100, -50);
    vertex(50, 100, 0);
    vertex(-100, 50, 50);
}
endShape();
```

These commands function on the graphics state, which is the combination of all the current graphics settings that control drawing, including colours and line drawing settings. That means that the commands do not change the colour of particular shapes, but change the state of the renderer. Calling `fill(255, 0, 0)` will put the renderer into a state where it will draw shapes in red. That means that any shape drawn after that command, until `fill` is called again, will be drawn in red.

4.8 Transformations

Objects in 3D can be moved and positioned using **Transformations**. There are three types of transformation that are commonly used:

Translation

Changes the position of an object by adding a vector (x, y, z) to that position. In *Processing* the `translate` command implements translation.

Rotation

Rotates shapes by a certain angle around a certain axis. In *Processing* the most common way of using rotation is to use the `rotateX`, `rotateY` and `rotateZ` commands which rotate by angle (in radians) around each of the x, y and z axes.

Scale

Changes the size of the shape. This can be a uniform scale in which the size is changed equally in all directions according to a number. Numbers less than 1 reduce the size and more than 1 increase it. A scale can also be non-uniform meaning the x, y and z directions are scaled differently. In *Processing*, the `scale` command can either take one parameter for a uniform scale or three for a non-uniform scale.

The following code shows an example of using transforms to position and rotate an object:

```
translate(20, 20, 50);
rotateY(radians(30));
rotateX(radians(45));
beginShape(TRIANGLE_STRIP);
{
    vertex(-100, -100, 50);
    vertex(100, -100, 0);
    vertex(100, 100, -50);
    vertex(50, 100, 0);
    vertex(-100, 50, 50);
}
endShape();
```

Tranforms can also be used to animate objects. The following code rotates based on a variable `angle` which is increased slightly with each frame resulting in a spinning object.

```
rotateY(angle);
angle += 0.05;
```

Transforms are implemented as matrices. Each type of transform corresponds to a particular form of matrix as described in Chapter 2 of this subject guide. The transform system works as a state machine in the same way as the style system. The renderer maintains a current matrix which represents the current transform state. Each time a `translate`, `rotate` or `scale` commmand is called this current matrix is multiplied by the relevant transform matrix. Whenever a vertex is rendered it is transformed by the current transform matrix.

This implies two things. Firstly, each call to a transform is applied on top of the last. A call to `translate` does not replace the previous call to `translate`; the two accumulate to produce a combined transform. This can result in some quite complex transformations. For example, an existing rotation will apply to a new translation command. For example:

```
rotateZ(PI/2.0);
translate(100, 0, 0);
```

In this code any object will appear to move by 100 units along the y-axis, not the x-axis because the translation has been affected by the rotation. Similarly, this code will result in a movement of 200 units, not 100:

```
scale(2.0);
translate(100, 0, 0);
```

This interference between tranforms can be confusing. In general, the interference can be avoided by always applying transforms in the following order:

```
translate(x,y,z);
rotate(a);
scale(s);
```

The other implication of the way transforms work is that any transform will apply to all objects that are drawn after it has been applied:

```
translate(100, 0, 0);
// both boxes are translated by 100 units:
box(10, 10, 10);
box(10, 10, 10);
```

What if we want to move two objects independently? We can use two commands `pushMatrix` and `popMatrix`. The above description was a slight simplification. The renderer does not maintain a single current matrix, it maintains a **stack** of matrices. A stack is a list of objects, where objects can be added and removed. The last object to be added is always the first one to be removed. The transform stack contains all of the matrices that affect the current objects being drawn. The `pushMatrix` command adds a new matrix to the stack. Any transforms after the call to `pushMatrix` are applied to this new matrix. When `popMatrix` is called the last matrix to be added to the stack is removed. This has the effect of cancelling any transform that was called after the previous call to `pushMatrix`, but still keeping any matrices that were active before the call to `pushMatrix`. This makes it simple to move two objects independently.

```
pushMatrix();
    translate(100, 0, 0);
    box(10, 10, 10);
popMatrix();

pushMatrix();
    translate(0, 100, 0);
    box(10, 10, 10);
popMatrix();
```

We can think of a transform matrix as providing a new coordinate system, as described in Chapter 3 of this subject guide. The matrix transforms coordinates in one coordinate system into another coordinate system. For example, it could convert from the coordinate system of an object into a coordinate system of the world. A transform stack allows us to combine multiple transforms and therefore multiple coordinate systems. For example, the top of the stack could be the world coordinate system, below that would be the coordinate system of an object (a table) and below that the coordinate system of sub-parts of the object (table legs). This allows us to create the type of hierarchical coordinate system shown in Figure 3.8 in chapter 3 of this subject guide. The commands `pushMatrix` and `popMatrix` allow us to move from one coordinate system to another.

Learning activity

Change the colour of the cube you created in the last activity.

Use transforms to move the cube to a different position and animate it spinning slowly.

Draw a second cube in a different position and colour.

4.9 Graphics objects

The process described above is called Immediate Mode graphics and is a low level approach to graphics that closely corresponds to how the graphics card works. Transforms and styles are maintained in a state machine. Shapes are drawn by specifying vertices, which are immediately sent for drawing on the graphics card. Understanding immediate mode is key to understanding how graphics hardware works. However, it has some flaws.

Firstly, it does not correspond well to how we normally think about graphics. We do not normally think in terms of graphics states and vertices. We think in terms of objects. Vertices are not ephemeral things that are re-created in each frame; they form stable objects that exist from frame to frame (for example, a teapot, or a table). Colours and positions are not graphics states, they are properties of objects. In the real world there is no abstract ‘red’ state; there are red teapots and red tables.

The second problem is efficiency. In immediate mode, each vertex is sent to the graphics card as soon as the vertex command is called. Transfers to the graphics card require a lot of overhead and transferring vertices one at a time can be expensive.

Happily, these two problems can have a common solution. Most high level graphics engines include the concept of a graphics object. A graphics object contains the vertices required to draw it (called the **Geometry**) as well as colour and style properties (called the **Material**). In most cases it will also include the transformations applied to the geometry. These properties are stored in memory and stay the same over time, so the same geometry, material and transformations are used every time an object is drawn. A graphics object corresponds much more closely to our understanding of a real world object. It can also be more efficient. All of the vertices can be set to the graphics card at once, thus reducing transfer overheads. In fact, it is even possible to optimise further by storing the geometry and materials directly on the graphics card almost eliminating transfer costs.

In *Processing* a graphics object is represented by a class called a PShape. This is a built in class and you can create a variable of type PShape without using any libraries, like this:

```
// declare a variable to hold our shape
PShape myShape;
```

A PShape object has methods `beginShape`, `endShape` and `vertex` that act much like their immediate mode equivalents. However, they do not draw the vertices directly, they simply add the vertices to the shape. This only needs to be done once, when we create the shape, rather than whenever we draw it:

```
// most of the work is done in setup,
// which creates the shape object
void setup(){
    size(640, 480, P3D);

    // create the shape object
    // it starts off empty, with
    // no vertices
    myShape = createShape();

    // add vertices to the shape using
    // beginShape and the vertex method
    // this works just like immediate mode
    myShape.beginShape(TRIANGLE_STRIP);
    {
        myShape.vertex(-100, -100, 50);
        myShape.vertex(100, -100, 0);
        myShape.vertex(100, 100, -50);
        myShape.vertex(50, 100, 0);
        myShape.vertex(-100, 50, 50);
    }
    myShape.endShape(CLOSE);

}
```

Drawing a shape is as simple as calling a command `shape` and passing your shape object as a parameter:

```
// draw simply has to draw the shape that has been defined.
void draw(){
    background(255);
    lights();

    // draw the shape
    shape(myShape);
}
```

As well as creating shapes out of vertices, it is possible to create PShape objects based on primitives.

```
float shapeParameters [] = new float []{100, 100, 100};
myShape = createShape(BOX, shapeParameters);
```

It is also possible to load objects from files using the 'OBJ' format. This is useful as it allows you to create 3D objects in an external modelling tool such as 'Blender' or 'Autodesk Maya' and use them in your program:

```
myShape = loadShape("Sphere.obj");
```

Figure 4.3: An OBJ model loaded into *Processing*.

The above code shows how to create PShapes out of vertices, but it is also possible to edit the vertices of an existing object. This makes it possible to animate the shape of an object:

```
for (int i = 0; i < myShape.getVertexCount(); i++) {
    PVector v = myShape.getVertex(i);
    v.x += random(-1, 1);
    v.y += random(-1, 1);
    myShape.setVertex(i, v);
}
```

A shape also has style properties. These are changed using commands such as `fill` and `stroke`, just like in immediate mode. (Note that, unlike immediate mode, these commands must be called between `beginShape` and `endShape`):

```
myShape.beginShape(TRIANGLE_STRIP);
{
        myShape.fill(100);
        myShape.noStroke();

        myShape.vertex(-100, -100, 50);
        myShape.vertex(100, -100, 0);
        myShape.vertex(100, 100, -50);
        myShape.vertex(50, 100, 0);
        myShape.vertex(-100, 50, 50);
}
myShape.endShape();
```

Shapes also have transform properties:

```
myShape.translate(-50, -50, -50);
myShape.rotateY(radians(30));
```

Just as in immediate mode, these transforms accumulate rather than replacing the existing transform. That means that calling `myShape.rotateY(0.1)` in draw will result in a spinning shape not a static rotation. This accumulation of transforms rather goes against the idea of using a graphics object, as it is more intuitive to think in terms of an object having a single position, rotation and scale than applying a complex set of transforms to them. In fact, most graphics engines will directly store a position, rotation and scale to their graphics objects and use those as the main method of controlling movement. Luckily it is straightforward to wrap a PShape in a custom class that has this functionality:

```
class GraphicsObject
{
  PVector position;
  PVector rotation;
  PVector scale;

  PShape shape;

  GraphicsObject(String filename)
  {
    shape = loadShape(filename);
    position = new PVector(0, 0, 0);
    rotation = new PVector(0, 0, 0);
    scale = new PVector(1, 1, 1);
  }

  void display()
  {
    pushMatrix();
        translate(position.x,position.y,position.z);
        rotateX(rotation.x);
        rotateY(rotation.y);
        rotateZ(rotation.z);

        scale(scale.x,scale.y,scale.z);

        shape(shape);
    popMatrix();
  }

}
```

Note that we are applying standard transforms before drawing the shape, rather than applying the transforms to the shape itself; this saves the problems of transforms accumulating on PShapes.

Learning activity

Rewrite your program from the previous activity so the cubes are represented as PShape objects.

Hierarchical graphics objects

It is often useful to make graphics objects out of a collection of other objects. For example, a table is a top and four legs. Most graphics engines enable you to do this by allowing graphics objects to have other graphics objects as children: a table would have its top and legs as children. The children inherit the transforms of their parents, so moving the parent will move all of the children together.

The PShape class is able to handle shapes which contain other shapes as children, but this feature is also easy to implement using our custom GraphicsObject class by adding an array of children:

```
class GraphicsObject
{
  PVector position;
  PVector rotation;
  PVector scale;

  PShape shape;

  GraphicsObject [] children;

  GraphicsObject(String filename)
  {
    shape = loadShape(filename);
    position = new PVector(0, 0, 0);
    rotation = new PVector(0, 0, 0);
    scale = new PVector(1, 1, 1);
    children = new GraphicsObject[0];
  }
  void addChild(GraphicsObject obj)
  {
        children = (GraphicsObject[])append(children, obj);
  }
  void display()
  {
    pushMatrix();
        translate(position.x,position.y,position.z);
        rotateX(rotation.x);
        rotateY(rotation.y);
        rotateZ(rotation.z);

        scale(scale.x,scale.y,scale.z);

        shape(shape);

        for (int i = 0; i < children.length; i++)
        {
            children[i].display();
        }
    popMatrix();
  }

}
```

Note, we are calling `display` on each of the children inside the `pushMatrix` `popMatrix` of the parent; this means that the children will inherit the transforms of the parent (but can still have their own transforms relative to the parent). Since all of the children can also have their own children, we can create arbitrarily complex hierarchies.

Learning activity

Create a hierarchical object using our GraphicsObject class; for example, a table.

4.10 Graphics Processing Units

The rapid improvement in the quality of 3D computer graphics over the last two decades has largely been down to the increasing use and power of dedicated graphics hardware in modern computers. These are custom chips called **Graphics Processing Units** (GPUs), that are designed specifically for creating images from 3D shapes represented as a collection of polygons made out of vertices. GPUs are often part of dedicated graphics cards, but can also be integrated on a single chip with the CPU. Early GPUs simply implemented a fixed set of graphics functionality in hardware, but modern GPUs are programmable: you can run short programs to be run directly on the GPU hardware that will change the way your objects are drawn. These programs are traditionally called **Shaders**.

GPU shader programming is an integral part of modern graphics programming, but it is very different from normal CPU programming because GPUs and CPUs rely on a very different programming model. CPUs are sequential, they perform one operation at a time, in rapid succession and programs are generally made up of a sequence of instructions. GPUs on the other hand are massively parallel and are able to perform many simple instructions simultaneously. Modern CPUs are somewhat parallel because they are multi-core, they might be made up of, for example, four cores, each of which is essentially an independent CPU. However, the model of parallelism used for CPU programming is generally multithreading, multiple independent programs running at the same time with possibly a small amount of interaction between them. GPUs work differently, they are designed to run many copies of the same program, each of which runs on different data and each of which must run independently of the other. This can make GPU programs extremely fast. However, it means that programming a GPU means thinking in a different way about programming, using different techniques and different programming languages that are designed specifically for GPUs. Luckily graphics programs are very well suited to parallelisation: they perform the same action independently on large numbers of vertices or pixels. (GPUs can also be used for non-graphics tasks, but this is often much more challenging and we will not cover it here.)

Shaders allow you to create a vast range of effects that are not possible with the fixed functionality; for example:

- complex materials: rough stone, translucent skin, shiny metals;
- natural phenomena: fire, smoke, water;
- advanced lighting effects: ambient occlusion, image based lighting;
- non-photorealistic rendering: cartoon or sketch effects;

- animation: skinning, particle systems.

The graphics pipeline

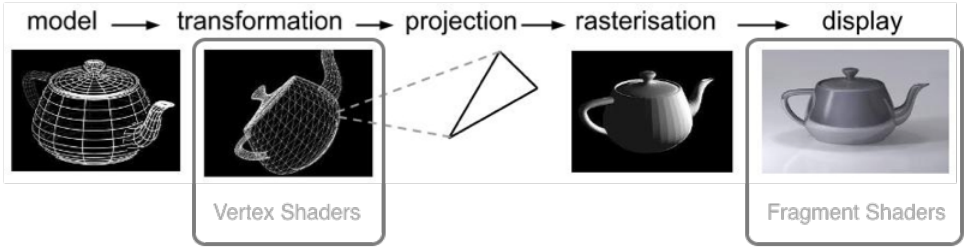


Figure 4.4: The graphics pipeline: the stages involved in going from geometric models to images.

In Chapter 1 of this subject guide we introduced the graphics pipeline, which is shown again in Figure 4.4. It consists of a number of stages, most of which occur on the GPU and some of which are programmable.

1. **Model.** Objects in the world are modelled out of vertices by the CPU program. These vertices are sent to the GPU.
2. **Transform.** All vertices are transformed into view coordinates by the ModelView matrix. This stage can also include other operations such as lighting vertices.
3. **Projection.** The vertices are projected into screen space using the Projection matrix (perspective or orthographic).
4. **Rasterisation.** The vertices are assembled into primitives (normally polygons but they could also be lines or dots). These polygons are then rasterised: that is they are drawn as a set of individual pixels (or fragments).
5. **Display.** The appearance of the fragments is determined based on the properties of the polygons. The colour of the vertices is often interpolated to get the fragment colour. Alternatively, the colour could be determined by a new operation on the individual fragment, such as applying a texture. The final step is to draw pixels to a frame buffer, a piece of memory that is mapped to the screen.

What we have described is a fixed functionality that is implemented in hardware as standard, but we can also program elements of the pipeline with small programs called shaders that run at certain stages of the pipeline. There are different types of shader that happen at different stages in the pipeline.

Vertex shaders. These act on vertices and take the place of the standard **transform** stage of the pipeline.

Fragment shaders. These act on fragments (pixels) and take the place of standard display, after the rasterisation stage and prior to writing pixels to the frame buffer.

Shader programming

You therefore have two (or more) programs in your graphics software: your main (CPU) program and one or more shader programs. The CPU program is an ordinary program written in a standard programming language such as *Processing*, Java or C++. This will define and prepare all of the vertex and texture data. It loads the shader to the GPU then sends all the vertex/texture data. The GPU programs are written in a specialist shader programming language. In our examples we will use GL Shading Language (GLSL), which is part of OpenGL but other languages include HLSL developed by Microsoft and Cg developed by nVidia.

The following sections will describe shader programming, but you should also refer to other reference material.

Below is an example of a (very simple) vertex shader program in GLSL. It transforms the vertices by the transform matrix and then passes the colour of the vertices to the fragment shader (this is the colour set using the fill command in *Processing*). We will describe it in more detail in the following sections.

```
#define PROCESSING_COLOR_SHADER // Note the US spelling

uniform mat4 transform;

attribute vec4 vertex;
attribute vec4 color;   // Note the US spelling

varying vec4 col;

void main(){
    gl_Position = transform*vertex;
    col = color;
}
```

This is the corresponding fragment shader. It simply colours the pixel according to the vertex colour.

```
#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec4 col;

void main() {
  gl_FragColor = col; // Note US spelling
}
```

To use a shader in *Processing* we must use a PShader object which represents a shader program:

```
PShader myShader;
```

We can then load in the fragment and vertex shaders from file:

```
myShader = loadShader("frag.glsl", "vert.glsl");
```

To use the shader, we need to call the `shader` command before any drawing that we want to use the shader for:

```
shader(myShader);
noLights();

fill(100);
noStroke();

translate(20, 20, 50);
beginShape(TRIANGLE_STRIP);
{
    vertex(-100, -100, 50);
    vertex(100, -100, 0);
    vertex(100, 100, -50);
    vertex(50, 100, 0);
    vertex(-100, 50, 50);
}
endShape();
```

Because our shader has no lighting we have to turn off all lighting with the `noLights` command. When using shaders in *Processing* it is important to define the type of shader, which determines what data *Processing* sends to the shader. This shader was defined as a colour shader using this line in the vertex shader.

```
#define PROCESSING_COLOR_SHADER  //Note US spelling
```

That means it only uses colour, not lighting or textures. In the following chapters we will see how to add lighting and textures.

Learning activity

Save the vertex and fragment shader defined above to separate files `vert.glsl` and `frag.glsl`. Write a *Processing* program to use those two shader programs (remember not to use any lights or textures).

Variables in shader programs

Variables in a CPU program are pieces of memory that can be read from or written to freely at any time. This is because the programs are sequential: as only one instruction can happen at a time we can be sure that there will not be two instructions changing a variable at the same time. Since GPU programs are massively parallel they rely on strict rules to ensure that variables are not written to simultaneously. These rules affect how different variables can be read or written to. Shader variables can be one of five different types, each of which can be written to in different ways:

- `uniform`: These are variables that have a single value passed per object or render. They are set by the CPU program and do not change until the CPU program starts a new shader. They can be read in any shader but not written to. Examples include light positions and transform matrices.

- `attribute:` These variables have a different value for each vertex. These are sent from the CPU to the shader as part of the vertex data (typically in an array). They can be read in a vertex shader but not written to. Examples include vertex position and normal.
- `varying:` are values that are passed from the vertex shader to the fragment shader. The vertex shader computes their values for each vertex and the rasteriser interpolates them to get a value for each fragment, which is passed to the fragment shader. They can be written to in the vertex shader and read in the fragment shader.
- `const:` variables are compile-time constant, they cannot be changed for a particular shader.
- `local:` these behave like standard local variables in a CPU program. They can both be written to and read from in a shader, but only exist within the scope of a single run of a single shader.

Shader variables have similar built in types to a CPU language like *Processing*: `float`, `int`. However, they do not have some types like strings and have some built in types that do not typically exist in other languages.

These are the types available in GLSL:

- **Integers:** `int`, `short` (a small integer) and `long` (a large integer).
- **Floating point:** `float`, `double` and `half` (which have, respectively, more or less precision than a `float`).
- **Vectors:** a single type representing multiple values, for example a `vec4` is a vector of 4 floats and ivec3 is a vector of 3 integers.
- **Matrices:** a 2 dimensional matrix of values, for example a `mat4` represents a 4 by 4 matrix of floats.
- **Texture samplers:** represent a texture, for example `sampler2D`.
- **Structures:** a set of values that are grouped together (like a class without methods). For example, this is a structure with an integer and a vector of floats:

```
struct MyStruct
{
    int a;
    vec4 b;
};
```

In a GLSL shader a uniform variable is defined within the shader using similar syntax to Java, with a type and a name but also the modifier `uniform`:

```
uniform mat4 transform;
```

`transform` is a built in uniform variable that is defined by *Processing* for the current transform matrix. We have used it in the above example. We could also define our own custom uniform variables; for example, a colour to use to tint our shapes (it is a `vec3` for r, g, b colour):

```
uniform vec3 tint;
```

If we are using custom uniform variables we need to set them from within our CPU program using the `set` command of `PShader`:

```
myShader.set("tint", 1.0, 0.0, 0.0);
```

This sets the three components of the tint variable. The set command can take a variable number of parameters for different types of uniform variables. For example, a float variable can be set with one parameter:

```
myShader.set("intensity", 1.0);
```

Vertex attributes use the `attribute` modifier. Custom vertex attributes are less common and not easily supported in *Processing*. We will therefore normally only use the default attributes that are automatically passed into the shader by *Processing*; for example, the position (`vertex`) and colour:

```
attribute vec4 vertex;
attribute vec4 color;  // Note US spelling
```

Varying variables are also defined in a similar way to standard variables, but use the `varying` modifier.

```
varying vec4 col;
```

This is a colour value that should be passed from the vertex shader to the fragment shader. It needs to be defined, with the same name and type in both vertex and fragment shader.

Vertex shaders

Vertex shaders act on individual vertices. They act on the basic vertex data to produce values that are interpolated by the rasteriser and then sent to the fragment shader. They perform a number of functions:

- vertex and normal transformations
- texture coordinate calculation
- lighting
- material application.

They take two types of input **uniform variables** such as the transform matrix or parameters of light sources and **vertex attributes** such as the position, normal or colour of a vertex. They then calculate **varying** values as output; for example, the colour of the vertex after lighting or the transformed (world space) position. These are interpolated and passed to the fragment shader.

Our example vertex shader starts with a definition, which tells us it only uses colour, not lighting or texture (see above):

```
#define PROCESSING_COLOR_SHADER // Note US spelling
```

It then defines the uniform, attribute and varying variables:

```
uniform mat4 transform;

attribute vec4 vertex;
attribute vec4 color; // Note US spelling
```

```
varying vec4 col;
```

The code itself is defined in the `main` function. It calculates the screen space position of the vertex by multiplying it by the transform matrix. It puts the value into `gl_Position`, which is a built in varying variable which will be automatically passed to the fragment shader. It also passes the vertex colour to our custom varying variable `col`:

```
void main(){
    gl_Position = transform*vertex;
    col = color; // Note US spelling
}
```

We can do many things with a fragment shader. One example is changing the vertex position. A simple example is to apply a sine function to the vertex position:

```
gl_Position.x *= (sin(0.04*gl_Position.y) + 2.0)/2.0;
```

This will create a wavy effect on the shape.

Fragment shaders

Fragment shaders act on individual fragments. Fragments are the elements of a polygon that will be drawn to specific pixels on screen. Fragments are created by the rasteriser, by interpolating the outputs of the vertex shader. Examples of what they do include:

- texture mapping
- bump/normal mapping
- generic texture operations
- fog.

They take three types of input, uniform variables, in the same way as vertex shader, varying values passed in from the vertex shader (e.g. transformed position, colour and texture coordinates) and texture maps. They output colour and depth values that are used to draw the fragments to the framebuffer.

Our example fragment shader starts with some code that ensures compatibility between different versions of OpenGL (do not worry too much about this but remember to include it in your code):

```
#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif
```

It then defines the same varying variable that we defined in our vertex shader:

```
varying vec4 col;
```

As with the vertex shader, the code is contained in the `main` function:

```
void main() {
  gl_FragColor = col;  // Note US spelling
}
```

It simply sets the fragment colour to be equal to the variable `col`. `gl_FragColor` is a built in variable that is used to set the fragment (pixel) colour.

This example is very simple, but we can do many more things in the fragment shader. We will cover some of these later in the course, but here we will give an example of tinting the colour. We can define a custom uniform variable as defined above (this should go in the fragment shader):

```
uniform vec3 tint;
```

We can then use this to calculate the fragment colour, for example:

```
gl_FragColor = vec4(col.xyz*tint, 1);
```

The syntax `col.xyz` means the x, y and z components of a vector.

Shader programs can achieve many more complex effects, such as lighting and textures, but these will be covered in future chapters of this subject guide.

Learning activity

Adapt one of the example shader programs you wrote in the previous activity to achieve a different visual effect (for example, drawing stripes on an object).

4.11 Summary

This chapter has introduced the basic concepts involved in graphics programming. We have used *Processing* as an example platform but most concepts translate straightforwardly to other platforms such as OpenGL. Graphics programming consists of two complementary elements: CPU programs that set up and control the graphics and GPU programs that do the actual rendering of the polygons. CPU graphics programming should be relatively familiar to experienced programmers, but GPU programming requires a different way of thinking. We recommend you continue to practise your graphics programming by applying it to the concepts taught in the following chapters of this subject guide.

4.12 Sample examination questions

1. Explain the difference between a vertex shader and a fragment shader.
2. Why are triangle strips and triangle fans often better to use than individual triangles? Give one example in which you would use a triangle strip and one in which you would use a triangle fan.
3. What is a uniform variable in a GPU program? Give three examples of how you might use uniform variables in a graphics program.

Chapter 5

Lighting 1 – shading

5.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 5.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapter 9.

5.2 Recommended reading

Hughes, J. et al. *Computer graphics: principles and practice.* (2013), Chapters 13 and 16.

Glassner, A. *Principles of Digital Image Synthesis.* (1994), Chapter 15.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapters 19 and 20.

Watt, A. *3D Computer Graphics.* (1999), Chapter 4.

5.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe and explain different types of light sources
- demonstrate awareness of surface properties: diffuse and specular
- explain the difference between local and global illumination
- explain the key concepts and algorithms behind shading.

5.4 Introduction

The light that we see is a mixture of all kinds of different sources. Incoming light interacts with a surface and may be absorbed, reflected, and/or transmitted (Figure 5.1). Each different colour is simply energy which can be represented by a wavelength: colour is a wavelength visible to the eye. In simple terms, the colour that a material reflects is observed as that material's colour. For example, if an object absorbed all the light except for red then the object would appear to be red when viewed in a white light. Also, the more light the material reflects the shinier it will appear to the viewer.

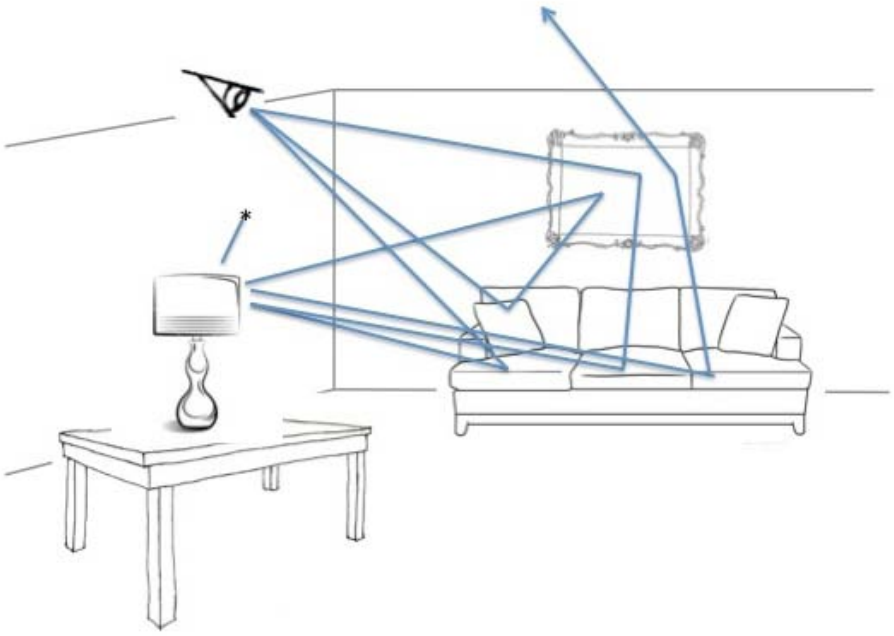


Figure 5.1: The illumination of a scene depends upon the light source, the path of the light, how that light interacts with objects, what the properties of those objects are, and the viewpoint for the scene.

In the real world, the interaction of light on surfaces gives shading, which humans use as an important depth cue. When creating a 3D computer generated image we can model the appearance of real-world lighting. As light-material interactions cause each point to have a different colour, we therefore need to know a number of different properties, namely:

- light sources
- material properties
- location of viewer
- surface orientation.

When applied to 3D graphics, adding this information is often collectively referred to as **shading**. Note the distinction in the following terms that you will encounter on this course: **Illumination** is the calculation of light intensity at a particular point on a surface. **Shading** uses these calculated intensities to shade the whole surface or the whole scene.

5.5 Local versus global illumination

In general, light leaves a light source, is reflected from many surfaces and then finally reflected to our eyes, or through an image plane of a camera. Illumination models are able to model the interaction of light with the surface and range from simple to very complex.

The light that goes directly from the light source and is reflected from the surface is called a **local illumination model** and the shading of any surface is independent from the shading of all other surfaces. Only the interaction between the light source and the point on the surface being shaded is considered. Light that takes an indirect path to the surface is not considered. This means that each object is lit individually, regardless of what objects surround it. Most real-time graphics rendering systems use local illumination.

A **global illumination model** adds to the local model the light that is reflected from other surfaces to the current surface. A global illumination model is more comprehensive, more physically correct, and produces more realistic images. It is also a great deal more computationally expensive. Global illumination is covered in Chapter 7 of this subject guide.

5.6 RGB colour

Light gives us colour, and we need to reproduce that colour in our virtual image by giving each pixel on the screen a certain colour value. The colour of a pixel is usually recorded as three values: red, green and blue (RGB). Mixing these three primary lights is sufficient to give the impression that a full rainbow spectrum is reproducible. This is known as the **RGB colour model**. This is additive colour mixing; it differs from subtractive mixing used in colour photography and printing where cyan, magenta and yellow are the primaries.

A colour in the RGB colour model is described by indicating how much of each of the red, green, and blue is included. The colour is expressed as an RGB triplet (red value, green value, blue value); each component of which can vary from zero to a defined maximum value – on a normalised scale $[0.0 \ldots 1.0]$ or actual byte values in the range $[0 \ldots 255]$. If all the components are at zero the result is black; if all are at maximum, the result is the brightest representable white.

An example of additive colour mixing:
red (1, 0, 0) + green (0, 1, 0) = yellow (1, 1, 0)
green (0, 1, 0) + blue (0, 0, 1) = cyan (0, 1, 1)
blue (0, 0, 1) + red (1, 0, 0) = magenta (1, 0, 1)
red (1, 0, 0) + green (0, 1, 0) + blue (0, 0, 1) = white (1, 1, 1)

RGB is a device-dependent colour model: different devices will reproduce a given RGB value differently, so an RGB value is not guaranteed to look the same across all devices without some kind of colour management being used.

24-bit colour

Actual RGB levels are often specified with an integer for each component. This is most commonly a one byte integer, so each of the three RGB components is an integer between 0 and 255. The three integers together take up three bytes, which is 24 bits: thus a system with 24 bit colour has 256 possible levels for each of the three primary colours. This is defined as ‘true colour’ or ‘millions of colours’ – a total of at least 16,777 216 colour variations, allowing for perceptually smooth transitions

between colours. Many modern desktop systems have options for 24-bit true colour with an additional 8 bits for an alpha (transparency) channel, which is referred to as '32-bit colour' – an **RGBA colour space**.

5.7 Lighting

The way in which light interacts with a scene is the most significant effect that we can simulate to provide visual realism. Light can be sent into a 3D computer model of a scene in a number of ways:

- as a **directional** or **parallel** light source that shines in a particular direction but does not emanate from any particular location
- as a **point** source that illuminates in all directions and diminishes with distance
- as a **spotlight** that is limited to a small cone-shaped region of the scene
- as **ambient** light, a constant which is everywhere in the scene.

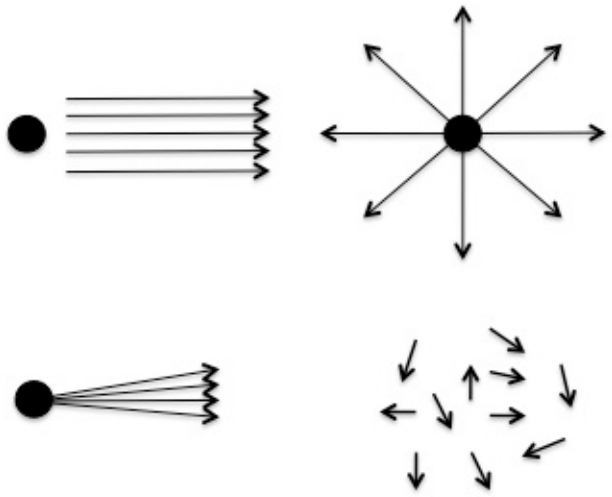


Figure 5.2: Lighting types:
Top left: directional. Top right: point.
Bottom left: spot. Bottom right: ambient.

Directional light

All of the rays from a directional light source have a common direction and no point of origin (Figure 5.2). It is as if the light source was infinitely far away from the surface that it is illuminating. For outdoor scenes, the sun is so far away that its illumination is simulated as a directional light source with all rays arriving at the scene in a parallel direction.

Point light

The point light source emits rays in radial directions from its source (Figure 5.2). A point light source is a fair approximation of a local light source such as a light bulb. For many scenes a point light gives the best approximation to lighting conditions.

Spot light

A spot light is a point source whose intensity falls off away from a given direction (Figure 5.2). The beam of the spotlight is normally assumed to have a graduated edge so that the illumination is at its maximum inside a cone, falling to zero intensity outside a cone.

Ambient light

Even though an object in a scene is not directly lit it will still be visible (Figure 5.2). This is because light is reflected indirectly from nearby objects. Ambient light does not model a 'true' light source; it is a workaround for 3D modelling – a way of faking things – consisting of a constant value that mimics indirect lighting.

Learning activity

In Chapter 4 of this subject guide we used the `lights` command to provide some basic lighting, but this gives very little control. *Processing* includes more advanced lighting commands such as `directionalLight` and `pointLight`; look these up in the *Processing* reference (`http://www.processing.org/reference/`, available May 2014) to understand how they work and then rewrite one of your programs from Chapter 4 in order to use more sophisticated lighting.

5.8 Reflection

When light hits an opaque surface some is absorbed and the rest is reflected, and occasionally transmitted/scattered too (Figure 5.3). The reflected light is what reaches our eyes: it is what we see. Calculating reflection is not simple and varies with material – surfaces' microstructure variations produce anything from bright reflection, such as a mirror, to a dull matte finish, like cardboard.

Specular reflection

Specular reflection is the direct reflection of light by a surface. Most light is reflected in a narrow range of angles. Shiny surfaces reflect almost all incident light and therefore have bright **specular highlights** or 'hot spots'. For a perfect mirror the angle of reflection is equal to the angle of incidence.

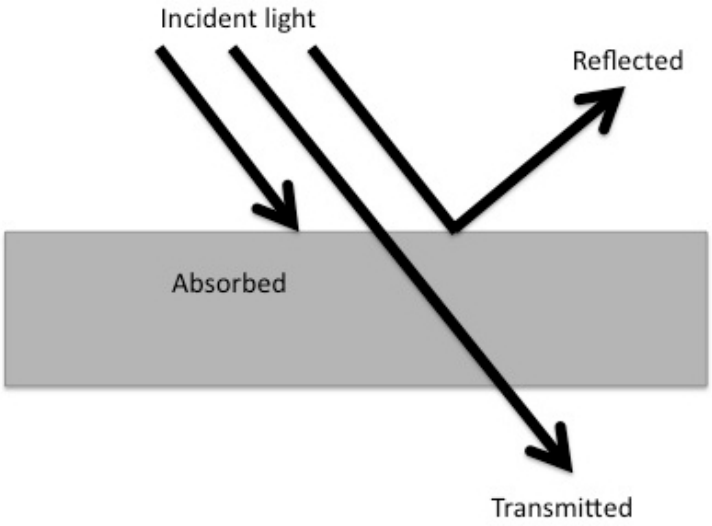


Figure 5.3: Light incident at a surface = light reflected + light scattered + light absorbed + light transmitted.

Phong model for specular reflection

The **Phong reflection model** (also called **Phong illumination model** or **Phong lighting model**) is widely used for real-time computer graphics to approximate specular reflection. This is an empirical model, which is not based on physics, but on physical observation; that is, it fits empirical observations but with no particular theoretical justification. Phong Bui-Tuong observed that for very shiny surfaces the specular highlight was small and the intensity fell off rapidly, while for duller surfaces it was larger and fell off more slowly. The model consists of three reflection components: the diffuse component, the specular component and the ambient component.

Diffuse reflection

Diffuse lighting is the most significant component of an illumination model. Light reflected from a diffuse surface is scattered in all directions. A material that is perfectly diffuse follows Lambert's Cosine Law, and so the surface looks the same from all directions; that is, the reflected energy from a small surface area in a particular direction is proportional to the cosine of the angle between that direction and the surface normal. An ideal diffuse surface is, at the microscopic level, a very rough surface (for example, chalk or cardboard). Because of the microscopic variations in the surface, an incoming ray of light is equally likely to be reflected in any direction. To model the effect we assume that a polygon is most brightly illuminated when the incident light strikes the surface at right angles. Illumination falls to zero when the beam of light is parallel to the surface.

Ambient reflection

When there are no lights in a scene the picture will be blank. By including a small fraction of the surface colour we can simulate the effect of light reflected from around the scene. Ambient reflection is a gross approximation of multiple reflections from indirect light sources. By itself, ambient reflection produces very little realism.

Depth cueing

Light is attenuated as it travels away from its source. In theory, light intensity should be attenuated using an inverse square law. In practice, a linear fall-off looks much more realistic. Fade distances can be set; for example, OpenGL uses the attenuation coefficient.

5.9 Shading

Human vision uses shading as a cue to form, position and depth.

Modelling the path of light in a scene is complex and computationally expensive, but shading models can be used to give us a good approximation of what would 'really' happen, much less expensively. This means that light can be modelled in a scene, but this light will be average and approximate rather than accurate and physically correct.

In terms of local illumination, to shade an image we need to set each pixel in a scene to a certain colour. To determine that colour we need to combine the effects of the lights with the surface properties of the polygon visible in that pixel.

The process for this is as follows:

$C_R = I_a s_R + I_c(I_s + I_d s_R) l_R$
$C_G = I_a s_G + I_c(I_s + I_d s_G) l_G$
$C_B = I_a s_B + I_c(I_s + I_d s_B) l_B$

where
C is the value for surface colour and illumination
s_R , s_G , s_B represents the surface colour
I_R , I_G , I_B represents the colour of the light
I_a , I_c , I_d , I_s refer to the ambient reflection, depth cueing, diffuse reflection and specular reflection.

To model a sphere so that it looks smooth by increasing or decreasing the number of polygons is very impractical. Although the outlines of these spheres in Figure 5.4 do not look particularly circular, they differ in appearance yet all have the same number of facets. We can fool the eye by showing continuity in shading.

There are two general ways in which shading can be applied in polygon based systems: **flat shading** and **interpolation shading**. They provide increasing realism at a higher computational cost.

Figure 5.4: A model of a sphere with (from left to right): no shading, flat shading, interpolation shading applied.

Flat shading

The simplest shading model for a polygon is flat shading, also known as **constant** or **faceted** shading. Each polygon is shaded uniformly over its surface. Illumination is calculated at a single point for each polygon. Usually only diffuse and ambient components are used.

Interpolation shading

A more accurate way to render a shaded polygon is to compute an independent shade value at each point. This is done quickly by interpolation:

1. Compute a shade value at each vertex.
2. Interpolate to find the shade value at the boundary.
3. Interpolate to find the shade values in the middle.

There are two main types of interpolation shading: **Gouraud** and **Phong**.

Gouraud

Gouraud shading was invented by Gouraud in 1971. The method simulates smooth shading across a polygon by interpolating the light intensity (that is, the colour) across polygons. It is a fast method and is supported by graphics accelerator cards. The downside is that it cannot model specular components accurately, since we do not have the normal vector at each point on a polygon.

The procedure is first to calculate the intensity of light at each vertex. Next, interpolate the RGB values between the vertical vertices. This gives us the RGB components for the left and right edges of each scan line (pixel row). We then display each row of pixels by horizontally interpolating the RGB values between that row's left and right edges.

Phong

Phong shading was introduced by Phong in 1975 and it is used by OpenGL. (Note: it is **not** the same as the Phong reflection model, described above.) It linearly interpolates a normal vector across the surface of the polygon from the polygon's vertex normals. The surface normal is interpolated and normalised at each pixel and then used to obtain the final pixel colour. Phong shading is more computationally expensive than Gouraud shading since the reflection model must be computed at

each pixel instead of at each vertex. It is slower but provides more accurate modelling of specular highlights.

The procedure is to compute a normal for each vertex of the polygon using bi-linear interpolation, then to compute a normal for each pixel. For each pixel normal, compute an intensity for each pixel of the polygon. Paint the pixel to the corresponding shade.

5.10 Lighting in a GPU shader

In order to perform lighting in a shader we must first tell *Processing* we will be using lights by adding the following definition to our vertex shader (instead of PROCESSING_COLOR_SHADER):

```
#define PROCESSING_LIGHT_SHADER
```

This gives us access to a new uniform variable `lightPosition`, which we must declare in our vertex shader before we can use it:

```
uniform vec4 lightPosition;
```

It also gives us access to vertex normals via the `normal` vertex attribute:

```
attribute vec3 normal;
```

Instead of transforming the normal by the model view matrix, we must transform it by a special normal matrix:

```
uniform mat3 normalMatrix;
```

Lighting calculations are often performed in the vertex shader. Code example 5.1 shows an example of a shader program with lighting.

```
void main(){
    gl_Position = transform*vertex;
    vec3 vertexCamera = vec3(modelview * vertex);
    vec3 transformedNormal = normalize(normalMatrix * normal);

    vec3 dir = normalize(lightPosition.xyz - vertexCamera);
    float light = max(0.0, dot(dir, transformedNormal));
    col = vec4(light, light, light, 1) * color;
}
```

Code example 5.1: Lighting in a vertex shader.

This code implements the diffuse lighting equation **XXXX**. First it calculates the vertex position in camera coordinates by transforming by the model view matrix:

```
vec3 vertexCamera = vec3(modelview * vertex);
```

Then we transform the normal:

```
vec3 transformedNormal = normalize(normalMatrix * normal);
```

We calculate the direction from the vertex to the light:

```
vec3 dir = normalize(lightPosition.xyz - vertexCamera);
```

And then calculate the dot product of the light direction and the vertex normal (equivalent to calculating the cosine of the angle between them):

```
float light = max(0.0, dot(direction, transformedNormal));
```

The result is the light intensity which can then be multiplied by the vertex colour (it has to be converted to a vector first) to get the lit colour of the vertex:

```
col = vec4(light, light, light, 1) * color;
```

Learning activity

Adapt your shader programs from Chapter 4 of this subject guide to include lighting.

The above example only uses one light, but we can access more lights by using an array uniform for the lights (see the *Processing* shader tutorials on processing.org for a full list of the light variables that *Processing* supports in shaders):

```
uniform int lightCount;
uniform vec4 lightPosition[8];
```

`lightCount` holds the number of active lights, so we can just loop over the active lights and perform the lighting calculation on them:

```
vec3 vertexCamera = vec3(modelview * vertex);
vec3 transformedNormal = normalize(normalMatrix * normal);

float light = 0.0;
for (int i = 0; i < lightCount; i++)
{
    vec3 dir = normalize(lightPosition[i].xyz - vertexCamera);
    light += max(0.0, dot(dir, transformedNormal));
}
col = vec4(light, light, light, 1) * color;
```

Learning activity

Our previous example of performing lighting in a shader did the light calculations in the vertex shader, which means they were calculated for each vertex (Gouraud shading). To perform Phong shading we must perform the lighting calculations in a fragment shader. Instead of calculating the light values in the vertex shader and passing them as a varying variable, we calculate the world space vertex position and normals and pass them to the fragment shader which calculates the lighting. Write a shader to perform Phong (per pixel) shading; there are numerous guides online on how to do this, for example: `http://www.processing.org/tutorials/pshader/` (accessed May 2014).

5.11 Summary

In terms of real-time graphics, the primary objective of shading is for efficiency of computation rather than for accurate physical simulation. As Phong himself stated:

'We do not expect to be able to display the object exactly as it would appear in reality, with texture, overcast shadows, etc. We hope only to display an image that approximates the real object closely enough to provide a certain degree of realism.'

Phong (1975)

5.12 Exercises

1. Investigate why specular highlights on plastic objects usually look white, while those on gold metal look gold.
2. Local illumination is kept quite simple and uses workarounds such as a constant value for ambient light to avoid expensive computation. While this works in many instances, can you think of scenes where such shortcuts might generate noticeable problems?

5.13 Sample examination questions

1. Explain how a colour value is described in the RGB colour model. Give examples.
2. The Phong model for specular reflection consists of an ambient, a diffuse and a specular component. Explain the purpose of Phong's model, what each of the three components are, and what real effect each is trying to model.
3. Discuss the reasons for any visual differences between Gouraud shading and Phong shading.
4. Describe the following types of reflection, using diagrams where appropriate:
   (a) diffuse reflection
   (b) specular reflection
   (c) ambient reflection.

Chapter 6

Textures

6.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 7.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapter 11.

6.2 Recommended reading

Hughes, J. et al. *Computer graphics: principles and practice.* (2013), Chapters 13 and 16.

Watt, A. *3D Computer Graphics.* (1999), Chapter 4.

6.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- understand the basics of applying textures to models
- describe and explain the commonly-used techniques
- implement texturing using *Processing*.

6.4 Introduction

Shading provides the first step towards realistic images but even the best synthesised images lack the richness of a real surface. Irregularities, wear-and-tear and imperfections are difficult to represent with polygons – to include that level of detail would require a huge number of facets and vertices. Because of this, it is necessary to find techniques that allow for realistic detail to be added without increasing the polygon count. Fortunately there is a conceptually simple and effective solution: texturing.

Texturing can be divided into two categories: **texture mapping** (also called **image mapping**) and **procedural texturing**. Both add realistic details to 3D models without increasing the number of polygons (Figure 6.1). Texture mapping places images on top of the polygons and thus can make simple models look incredibly realistic and is the easiest way to add fine detail. Procedural textures are calculated mathematically to provide realism at any resolution.

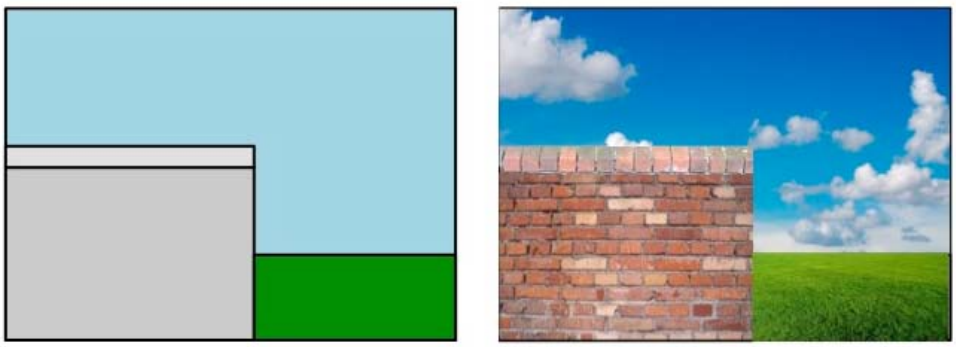


Figure 6.1: The image on the right portrays a brick wall, a lawn and the sky. In actuality the wall was modelled as a rectangular solid, and the lawn and the sky were created from rectangles. The entire image contains eight polygons.
(After Rosalee Wolfe, ©1997. Reproduced by kind permission from:
`http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf`)

6.5 Texture mapping

Texture mapping uses an **image map**, which is literally just an image; that is, a 2D array of intensities. Any image may be used as the source for the image map. Digital photographs or 2D artwork are the usual sources. It is not limited to actual pictures; surface textures like skin and cloth can also be added using this method. These images can then be 'painted' onto polygons – it is like taking some giftwrap and sticking it onto the polygons.

The key step in texturing is mapping from coordinates in the object space to particular points in the image. For each pixel in an object, we have to ask where to look in the texture map to find the colour. To be able to answer this, we need to consider two things: **map shape** and **map entity**.

Textures repeat themselves, just like giftwrap with a repeating motif. In general, mapping a 2D image to a polygon is just a 2D transformation. This is where knowledge of coordinate systems is necessary (see Chapter 3 of the subject guide for a reminder of this). In the OCS the coordinates are fixed relative to the object. Most mapping techniques will therefore use object coordinates to keep the texture in the correct place when the object moves. If the texture was mapped using the WCS then the pattern would shift as the object moves.

Map shape

If we are using a map shape that is planar, we take an (x, y, z) value from the object and project (that is, discard) one of the components. This results in a two-dimensional (planar) coordinate which can be used to look up the colour from the texture map. Figure 6.2 shows some textured-mapped objects that have a planar map shape. No rotation has occurred. It is the z-coordinate – namely, the depth – that has been discarded. You can work out which component has been projected by looking for colour changes in coordinate directions. In this case, movement along

the z-axis does not produce a change in colour, which is how you can tell that the z-component was eliminated.

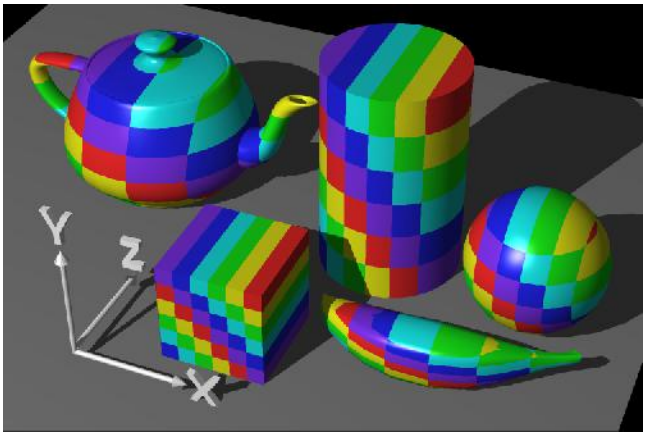


Figure 6.2: This image shows several textured-mapped objects that have a planar map shape. None of the objects have been rotated. Movement along the z-axis does not produce a change in colour. This is how you can tell that the z-component was projected.
Reproduced by kind permission of Rosalee Wolfe:
`http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf`

Learning activity

Work out what the objects in Figure 6.2 would look like if the x or the y components were projected.

Some surfaces are more difficult to texture map, like the problems encountered when giftwrapping a cylinder or a spherical object. In this case, an (x, y, z) value is converted to cylindrical coordinates (radius, theta, height). The radius is not used. To find the required colour in the 2D texture map, theta is converted into an x-coordinate and height is converted into a y-coordinate, meaning the texture map is wrapped around the object (Figure 6.3).

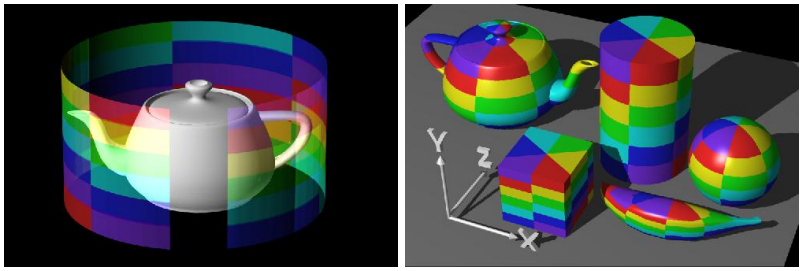


Figure 6.3: The texture-mapped objects in this image have a cylindrical map shape, and the cylinder's axis is parallel to the z-axis.
Reproduced by kind permission of Rosalee Wolfe:
`http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf`

With a sphere, the (x, y, z) value of a point is converted into spherical coordinates to gain the latitude and the longitude information. The latitude is then converted into

an x-coordinate and the longitude is converted into a y-coordinate. As might be expected, the objects 'North Pole' and 'South Pole' shows the the texture map squeezed into pie-wedge shapes (Figure 6.4).

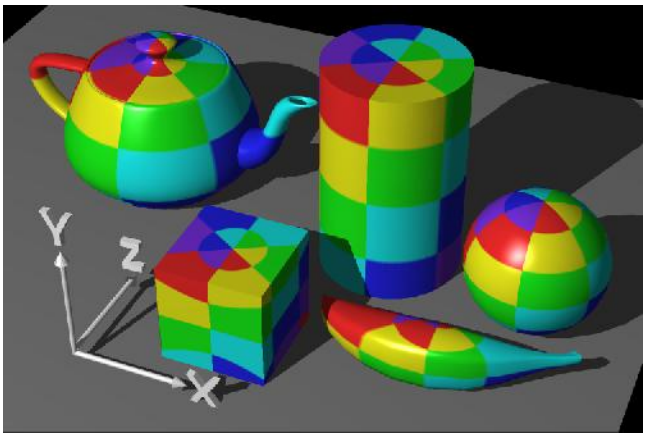


Figure 6.4: The objects have a map shape of a sphere, and the poles of the sphere are parallel to the y-axis. Compare this image to Figure 6.3 which has a map shape of a cylinder. Both map shapes have the pie-wedge shapes at the poles, but there is a subtle difference at the objects' 'equator'. The spherical mapping stretches the squares in the texture map near the equator, and squeezes the squares as the longitude reaches a pole.
Reproduced by kind permission of Rosalee Wolfe:
`http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf`

Map entity

The map entity determines what we use as the (x, y, z) value. It could be a point on the object, the surface normal, a vector running from the object's centroid through the point, or perhaps the reflection vector at the current point (Figure 6.5).

Learning activity

Some combinations of map shape and map entity produce more useful results than others. Explore combinations of map shapes and map entities using Rosalee Wolfe's "Teaching Texture Mapping Visually" at: `http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf` (available May 2014).

6.6 Texturing in *Processing*

Images in *Processing* are represented using the `PImage` class and loaded from file using the `loadImage` command:

```
PImage myTexture = loadImage("texture.jpg");
```

To texture a shape in *Processing* we can pass a `PImage` object into the `texture` command, which must be called inside `beginShape` and `endShape`:

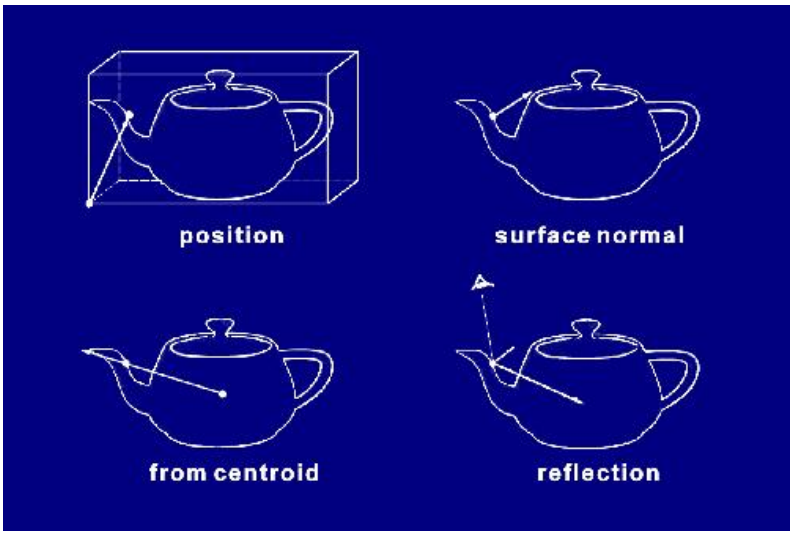


Figure 6.5: Commonly-used map entities.
Reproduced by kind permission of Rosalee Wolfe:
`http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe_mapping.pdf`

```
beginShape();
    texture (myTexture);
    vertex(100, 100, -100, 0, 0);
    vertex(200, 100, -150, 1, 0);
    vertex(200, 200, -200, 1, 1);
    vertex(150, 200, -150, 0.5, 1);
    vertex(100, 150, -100, 0, 0.5);
endShape();
```

In the above example we are passing an extra two parameters to `vertex`; these are the texture coordinates. There are two modes for handling texture coordinates in *Processing* (set using the `textureMode` command): in normal mode the coordinates represent pixel in the image, while in the normalised mode (used in the example) the texture coordinates are between 0 (top or left of the image) and 1 (the bottom or right of the image).

Learning activity

Rewrite one of your programs from Chapter 4 of this subject guide to include textures.

Texturing in a shader

In order to perform texturing in a shader we must first tell *Processing* we will be using textures by adding the following definition to our vertex shader (instead of PROCESSING_COLOR_SHADER):

```
#define PROCESSING_TEXTURE_SHADER
```

This will give us access to a new vertex attribute, `texCoord`, which we should declare in our vertex shader (*Processing* will pass it to our shader automatically):

```
attribute vec2 texCoord;
```

It will also provide a new uniform variable, `texMatrix`, which is the matrix that is used to transform the texture coordinates:

```
uniform mat4 texMatrix;
```

We also need to provide a varying variable to pass the transformed texture coordinate to the fragment shader:

```
varying vec4 outputTexCoord;
```

The vertex shader has to transform the texture coordinates of the vertex (we must first convert them to a vec4):

```
void main(){
    gl_Position = transform*vertex;
    outputTexCoord.xy = texCoord;
    outputTexCoord.zw = vec2(1.0, 1.0);
    outputTexCoord = texMatrix*texCoord;
    col = color;
}
```

The texturing itself is done in the fragment shader. We must declare a uniform variable for the texture. This is of type `samples2D`. It is called a sampler because it samples from a texture (in this case a 2D image texture):

```
uniform sampler2D texture;
```

We can pass a texture from the CPU to GPU by using the `set` command of `PShader`, passing in a `PImage` object:

```
myShader.set("texture", myTexture);
```

The fragment shader uses the transformed texture coordinates to sample the texture:

```
gl_FragColor = texture2D(texture, outpuTexCoord.xy);
```

Texturing can be combined with lighting. To use both in a shader we need the following definition in order to have all the variables available:

```
#define PROCESSING_TEXLIGHT_SHADER
```

Lighting and texturing can be combined by multiplying the texture colour by the calculated light colour:

```
gl_FragColor = lightColour*texture2D(texture, outpuTexCoord.xy);
```

Learning activity

Adapt your shader programs from Chapter 4 of this subject guide to include textures.

6.7 Procedural texturing

Texture maps are a good solution for adding realism but they have one major drawback: a fixed amount of detail. They cannot be scaled larger without losing resolution. Procedural textures take an entirely different approach, creating the texture itself. Procedural textures are textures that are defined mathematically. You provide a formula and the computer is able to create the texture at any scale, in any orientation. Instead of using an intermediate map shape, the (x, y, z) coordinate is used to compute the colour directly – equivalent to carving an object out of a solid substance. Rather than storing a value for each coordinate, 3D texture functions use a mathematical procedure to compute a value based on the coordinate, hence the name 'procedural textures'.

There are two general types of procedural texture:

1. Those that use regular geometric patterns.
2. Those that use random patterns.

Combining these two types can give enhanced realism, for example, making paving slabs look irregular, or making brickwork appear weathered).

Usually, the natural look of the rendered result is achieved by the use of fractal noise (such as Perlin noise, see below) and turbulence functions. These functions are used as a numerical representation of the 'randomness' found in nature. Without such randomness, computer generated scenes can look too orderly, uniform and neat – and thus unconvincing.

Perlin noise was invented by Ken Perlin in the 1980s and is a technique used to generate noise that is more natural in appearance. For this, the `noise( )` method in *Processing* is used, rather than the more familiar `random( )` method. This value simply determines where in a pre-computed function to look up the noise value. Small jumps create small shifts in random values, and larger jumps create larger jumps in the random values.

Learning activity

Read through Shea McCombs tutorial on procedural textures. It can be found online at: `www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures` (accessed May 2014)

6.8 Bump mapping

Bump mapping is a lot like texture mapping. However, where texture mapping adds colour to a polygon, bump mapping adds, what appears to be surface roughness. The surface of a 2D texture is flat and therefore the surface normals go straight up and the surface appears flat. Bump mapping evaluates the current light intensity at any given pixel on the texture. Rather than altering the geometry, it adds 'fake' depth by modifying the surface normals. The technique uses the colour from an image map to change the direction of the surface normal (Figure 6.6).

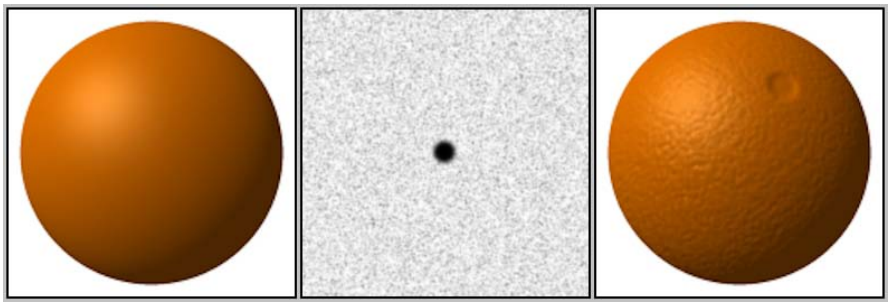


Figure 6.6: A sphere without bump mapping (left). A bump map to be applied to the sphere (middle). The sphere with the bump map applied (right) appears to have a mottled surface resembling an orange. Bump maps achieve this effect by changing how an illuminated surface reacts to light without actually modifying the size or shape of the surface.
Source: By Brion VIBBER and GDallimore; reproduced under Creative Commons licence CY BY-SA 3.0 from:
`http://en.wikipedia.org/wiki/Bump_mapping#mediaviewer/File:Bump-map-demo-full.png`

The most common way to represent bumps is by the heightfield method. A greyscale texture map is used, where the brightness of each pixel represents how much it sticks out from the surface (black is minimum height, white is maximum height). Bump mapping changes the brightness of the pixels on the surface in response to the heightmap that is specified for each surface.

Learning activity

If you completed the activity from Chapter 5 of this subject guide which asked you to implement Phong shading in a GPU program you can extend this program to include bump mapping. You need to combine the normal values passed from the vertex shader with normal values sampled from the normal map. There are many tutorials on GPU based normal mapping online; for example:
`http://fabiensanglard.net/bumpMapping/index.php/` (available May 2014)

6.9 Displacement mapping

Bump mapping does not alter an object's geometry. Because of this, a bump-mapped object will cast shadows that show the underlying geometry; that is, shadows will have smooth edges, not bumpy. However, a similar technique known as **displacement mapping** does actually alter an object's geometry. Displacement mapping changes the position of points over the textured surface, often along the local surface normal, according to the value from the texture map. It moves the render faces, not the physical mesh faces, so the polygon count stays the same but the surface is altered and therefore has a profile and shadows that show this (Figure 6.7).

Figure 6.7: Compare the silhouettes of these three objects. Left: original object. Middle: with bump mapping applied. Right: using displacement mapping.
Source: By Paul at Chromesphere. Reproduced by kind permission from: `www.chromesphere.com`

6.10 Environment mapping

Creating true specular highlights and reflections in a generated image is discussed in Chapter 7 of this subject guide. It is time consuming and computationally expensive; therefore, a shortcut is needed for realtime graphics. An **environment map**, also known as a **reflection map** can satisfy this requirement.

Environment mapping is the process of reflecting the surrounding environment in a shiny object – it is a cheap way to create reflections. When you look at a shiny object, what you see is not the object itself but how the object reflects its environment. What you see when you look at a reflection is not the surface itself but what the environment looks like in the direction of the reflected ray. An environment can be viewed as an infinity of light sources and a map can represent any arbitrary geometry of light sources. For example, in the environment map, striplights are just rectangles of high-intensity white values.

First, the environment itself has to be created or captured – either computer generated or captured using a probe, such as a chrome sphere, which captures real world reflections. The object is then surrounded by a closed three dimensional surface onto which the environment is projected. Imagine the mapping process as one where the reflected surface is located at the centre of an infinitely large hollow cube, with the image map painted on the inside, where each inner face of the cube is a 2D texture map representing a view of the environment. The map is actually projected onto the surface of the object from the point of view of the observer. Reflected rays are traced from the object, hit the surface and are then indexed onto the map (Figure 6.8).

6.11 Summary

Texturing provides a shortcut way of adding realism to images without increasing the polygon count. This can include not only adding images to polygons, but also the appearance of changes to the surface of the model.

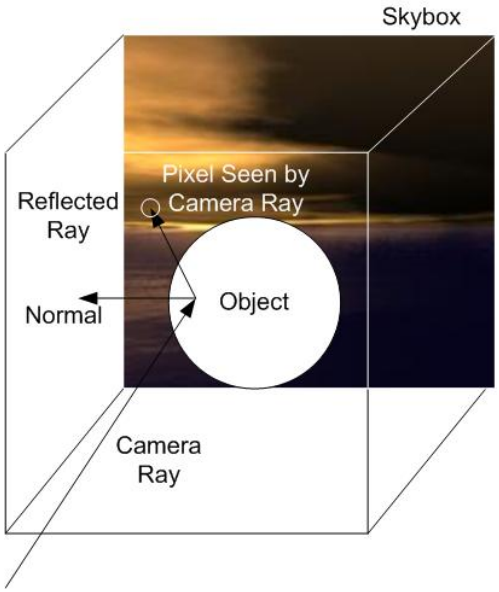


Figure 6.8: A diagram depicting an apparent reflection being provided by cube mapped reflection.
Source: By TopherTG; reproduced under Creative Commons licence CC BY-SA-3.0 from: `http://en.wikipedia.org/wiki/Reflection_mapping#mediaviewer/File:Cube_mapped_reflection_example.jpg`

6.12 Exercises

1. The `noise( )` method in *Processing* can take 1, 2 or 3 parameters. What do these parameters represent?

6.13 Sample examination questions

1. What shapes are the following texture coordinate functions being projected onto?
   (a) $u = \text{atan2}(x, z)$; $v = y$
   (b) $u = x; v = y$
   (c) $u = \text{atan2}(x, z)$; $v = \text{atan2}(y, \sqrt{x2 + z2}\,)$.
2. There are several techniques that can be used to make surfaces look more realistic.
   (a) Describe the technique of bump mapping.
   (b) How is it performed?
   (c) How does bump mapping differ from displacement mapping?

Chapter 7

Lighting 2 – representing the real world

7.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 11.

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapter 10.

7.2 Recommended reading

Glassner, A. *Principles of Digital Image Synthesis.* (1994), Chapters 18 and 19.

7.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- demonstrate awareness of the need for realistic images
- describe and explain the techniques involved in global illumination
- evaluate the strengths and weaknesses of these techniques
- give examples of applications requiring physically accurate images.

7.4 Introduction

There has long been philosophical debate about what it means to say that an image looks 'realistic'. In terms of computer-generated imagery, one of the aims is often to achieve **photorealism**; that is, images that look as convincing as a photograph of a real scene. Creating realistic images is not just a goal for the entertainment industry, where special effects rely on looking as convincing as possible. Increasingly, there are many applications where it is important that an image not only looks real but also simulates a real scene physically and accurately. This chapter discusses the approaches to creating realistic images that represent the real world, and provides examples of applications from current research.

In Chapter 5 of this subject guide it was mentioned that local illumination only considers direct light; that is, light coming directly from light sources. However, much of the light we see is indirect light. In local illumination models we can fake this using an ambient light value. However, global illumination models take into account direct and indirect illumination – we do not need to use an ambient value. This chapter describes the techniques behind global illumination.

7.5 BRDFs

From everyday observation we know that objects look different when viewed from different angles, and when lit from different directions. This is what is known as a **Bi-directional Distribution Function** or **BRDF**. A BRDF is essentially the description of how a surface reflects. It simply describes how much light is reflected when light makes contact with a certain material.

If a material is opaque then the majority of incident light is transformed into reflected light and absorbed light, and so what an observer sees when its surface is illuminated is the reflected light. The degree to which light is reflected (or transmitted) depends on the viewer and light position relative to the surface normal and tangent. A BRDF is therefore a function of the incoming light direction and the outgoing direction (the direction of the viewer). As well as that, since light interacting with a surface absorbs, reflects, and transmits wavelengths depending upon the physical properties of the material, this means that a BRDF is also a function of wavelength. It can be considered as an impulse-response of a linear system.

There are different ways to determine the value of a BRDF. They can be measured directly from real objects using specially calibrated cameras and light sources, or they can be models derived from empirical measurements of real-world surfaces.

Learning activity

Collections of BRDFs for materials exist, such as the Mitsubishi Electric Research Laboratories (MERL) database: `www.merl.com/brdf/` and Cornell University's reflectance data repository: `www.graphics.cornell.edu/online/measurements/reflectance/`. Walt Disney Animation Studios have developed software that allows you to explore such databases using their BRDF Explorer (`www.disneyanimation.com/technology/brdf.html`) which is available to download and try out (all available May 2014).

7.6 The rendering equation

Using the definition of a BRDF we can describe surface radiance in terms of incoming radiance from all different directions. The **rendering equation**, introduced independently by D. Immel et al. and J. T. Kajiya in 1986, is a way of describing how light moves through an environment.

Simply put, the rendering equation defines how much light leaves from any point in a 3D scene. Mathematically-speaking, the light exiting any point in a particular direction is the sum of the amount of light it emits in that direction and the amount of light it reflects in that direction from any incoming light. The rendering equation describes the total amount of light emitted from a point along a particular viewing direction, given a function for incoming light and a BRDF, as shown in Figure 7.3. Global illumination algorithms aim to solve approximations of the rendering equation.

The full equation is as follows:

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n) d\omega_i$$

where:

- λ is a particular wavelength of light (without this, everything would be greyscale)
- t is time (for ease, let us assume it is constant for now)
- x is the location in space
- ω_o is the direction of the outgoing light
- ω_i is the negative direction of the incoming light
- $L_o(x, \omega_o, \lambda, t)$ is the total spectral radiance of wavelength λ directed outward along direction ω_o at time t, from a particular position x. **In other words, the rendering equation is a function which gives you the outgoing light in a particular direction ω_o from a point x on a surface.**
- $L_e(x, \omega_o, \lambda, t)$ is emitted spectral radiance (and since most surfaces tend not to emit light there is not usually any contribution here)
- Ω is the unit hemisphere (see Figure 7.3) containing all possible values for ω_i
- $\int_\Omega \dots \, d\omega_i$ is an integral over Ω. The enclosed functions need to be integrated over all directions ω_i in the hemisphere above x. The orientation of the hemisphere is determined by the normal, n
- $f_r(x, \omega_i, \omega_o, \lambda, t)$ is the bidirectional reflectance distribution function (BRDF) – the proportion of light reflected from ω_i to ω_o at position x, time t, and at wavelength λ
- $L_i(x, \omega_i, \lambda, t)$ is spectral radiance of wavelength λ coming inward towards x from direction ω_i at time t. The incoming light does not have to come from a direct light source – it may be indirect, having been reflected or refracted from another point in the scene.
- $\omega_i \cdot n$ is the weakening factor of inward irradiance due to incident angle, as the light flux is smeared across a surface whose area is larger than the projected area perpendicular to the ray. This attenuates the incoming light.

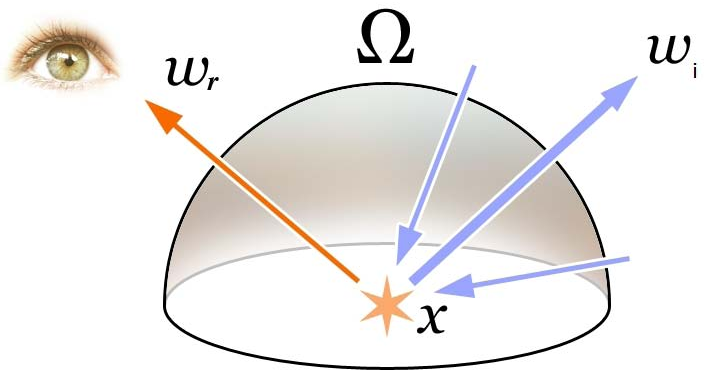


Figure 7.1: The rendering equation describes the total amount of light emitted from a point x along a particular viewing direction, given a function for incoming light and a BRDF. Source: By Timrb; reproduced under Creative Commons licence CC BY-SA 3.0 from: `http://en.wikipedia.org/wiki/Rendering_equation#mediaviewer/File:Rendering_eq.png`

Solving the rendering equation for a given scene is the main challenge in physically-based rendering, where we try to model the way in which light behaves in the real world. This chapter of the subject guide is a starting point for learning about the rendering equation and its applications. It plays a vitally important role in the sections we describe below on ray-tracing and radiosity. It is not able to capture every form of lighting (for example, it cannot handle aspects such as subsurface scattering, transmission, or fluorescence) but it is paramount in creating realistic graphics.

7.7 Ray tracing

Making an image appear realistic involves setting pixels to the correct colour, depending on the material properties and how those materials are lit. Put simply, if you can follow the paths that light takes around a scene then you can simulate real world lighting and generate a very realistic image indeed. In practice, this would be a hopeless task as it would be computationally impossible to calculate every interaction between light and multiple surfaces as it travels around a scene. There is, however, a simpler method. **Ray-tracing** simulates the path of light in a scene, but it does so in reverse. A ray of light is traced backwards through the scene, starting from what the eye or camera sees. When it intersects with objects in the scene its reflection, refraction, or absorption is calculated.

Ray tracing is the most complete simulation of an illumination-reflection model in computer graphics. Not only does it incorporate direct illumination – light that travels directly from a light source to a surface – but it can also handle light that originates from within the scene environment – the indirect light that is present due to light bouncing off other surfaces and reaching other objects. Its big advantage is that it combines hidden surface removal with shading due to direct illumination, shading due to global illumination, and shadow computation within a single model. The downside of ray tracing is that although it generates incredibly realistic images, it is computationally expensive with extremely high processing overheads – scenes can take minutes, hours or days to render.

A ray in a 3D scene generally uses a 3D vector for the origin and a normalised 3D vector for the direction. We begin by shooting rays from the camera out into the scene (Figure 7.2). The pixels can be rendered in any order (even randomly), but it is easiest to go from top to bottom, and left to right. We generate an initial primary ray (also called a camera ray or eye ray) and loop over all of the pixels. The ray origin is simply the camera's position in world space.

The initial camera ray is tested for intersection with the 3D scene geometry. If the ray does not hit anything, then we can colour the pixel to some specified 'background' colour. Otherwise, we want to know the first thing that the ray hits – it is possible that the ray will hit several surfaces, but we only care about the closest one to the camera. For the intersection, we need to know the position, normal, colour, texture coordinate, material, and any other relevant information about that exact location. If we hit somewhere in the centre of a polygon, for example, then this information would get computed by interpolating the vertex data.

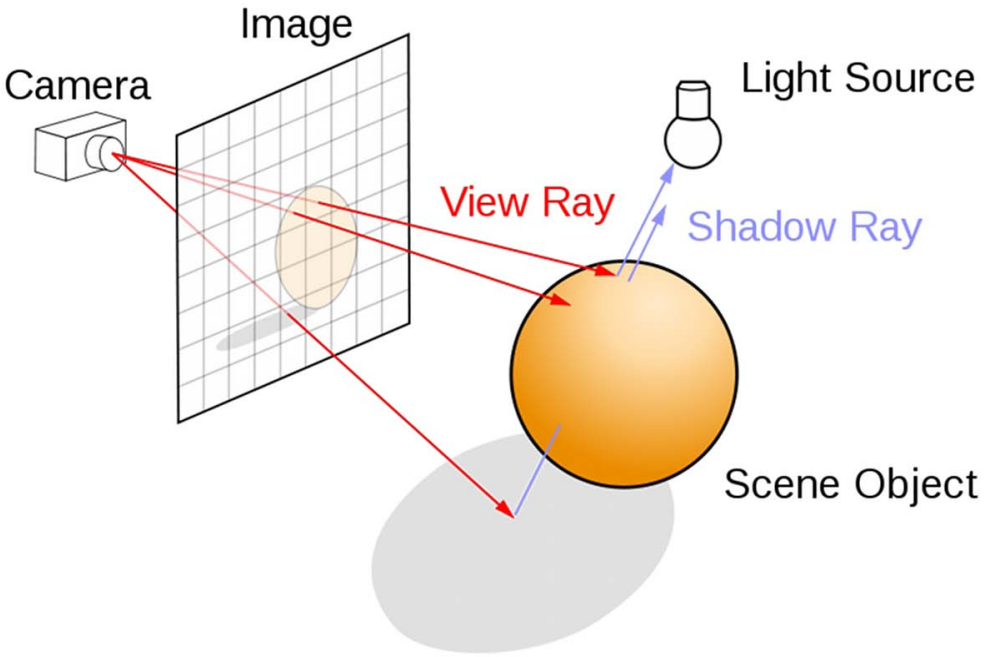


Figure 7.2: The ray tracing algorithm builds an image by extending rays into a scene. Source: By Henrik; reproduced under Creative Commons Licence CC BY-SA 3.0 from: `http://en.wikipedia.org/wiki/Ray_tracing_(graphics)#mediaviewer/File:Ray_trace_diagram.svg`

Once we have the key intersection information (position, normal, colour, texture coordinates, and so on) we can apply any lighting model we want. This can include procedural shaders, lighting computations, texture lookups, texture combining, bump mapping, and more. However, the most interesting forms of lighting involve spawning off additional rays and tracing them recursively. The nearest object hit spawns secondary rays that also intersect with every object in the scene (except the current one), and this continues recursively until the end result is reached: a value that is used to set the pixel colour.

When a ray hits a surface, it can generate reflection, refraction, and shadow rays. A reflection ray is traced in the mirror-reflection direction and the object it intersects is what will be seen in the reflection. Refraction rays work similarly. For clear

reflections and refractions only a single ray is traced. For shadows, a shadow ray is traced toward each light. If the ray intersects with a light then the point is illuminated based on the light's settings. If it does intersect with another object, that object casts a shadow on it.

The quality of a ray traced scene depends on the number of 'bounces' – the more bounces (that is, the deeper the recursion), the better the quality. Ray traced images have a tendency to appear very clean and sharp with hard shadows. In fact, they can look so clean that they appear unrealistic. Introducing randomness can counter this. Distributed ray tracing, a refinement of ray tracing that generates 'softer' images, is one way of doing this.

Real-time ray tracing

The first implementation of a 'real-time' ray-tracer occurred in 2005. It was a parallel implementation limited to just a few frames per second. Since then there has been much focus on achieving faster real-time ray tracing but it is limited by the computational power available. It is not yet practical but moves are being made in this direction and as the hardware and software improves, so too does the likelihood of usable, practicable real-time ray tracing within the next few years.

Learning activity

Want to try out physically-based rendering? For highly accurate lighting simulations we recommend Radiance, a free-of-charge open source ray tracing software system that runs on UNIX: `radsite.lbl.gov/radiance/`. It is not user-friendly but its accuracy has been rigorously validated. Popular favourite POV-Ray (the Persistence of Vision ray tracer), available at: `www.povray.org/`, is a free ray tracer for Windows, Linux and Mac. (Both available May 2014.)

7.8 Radiosity

Ray tracing follows all rays from the eye of the viewer back to the light sources. It is entirely dependent on what the viewer can see from their given position. If you want to change the position of the viewer it involves recalculating the lighting by re-ray tracing the entire scene to update it to the new position.

There is an alternative to calculating viewpoint-dependent illumination interactions. **Radiosity** simulates the diffuse propagation of light starting at the light sources. It is independent from what we see – it calculates 3D data rather than working on the pixels in an image plane projection, so the solution will be the same regardless of the viewpoint – something that would be very time consuming with ray tracing where for each change in viewing position the scene needs to be recalculated.

The main idea of the method is to store illumination values on the surfaces of the objects. It has its basis in the field of thermal heat transfer: the radiosity of a surface is the rate at which energy leaves that surface, and this includes energy emitted by the surface as well as energy reflected from other surfaces. In other words, the light that contributes to the scene comes not only from the light source itself but also from the surfaces that receive and then reflect light. It uses the finite element method to

solve an approximation of the rendering equation.

The surfaces of the scene to be rendered are each divided up into one or more smaller surfaces known as patches. A **form factor** (also known as a **view factor**) – a coefficient describing how well the patches can see each other – is computed for each patch. The form factors can be calculated in a number of ways. The early way of doing this was the **hemicube method** – a way to represent a 180° view from a surface or point in space. This, however, is quite computationally expensive, so a more efficient approach is to use a BSP tree to reduce the amount of time spent.

The radiosity for each patch is then calculated. The radiosity equation is a matrix equation or set of simultaneous linear equations derived by approximations to the rendering equation. This gives a single value for each patch. Gouraud shading is then used to interpolate these values across all patches. Now that some parts of the scene are lit, they themselves have become sources of light, and they could possibly cast light onto other parts of the scene. The process is therefore repeated a second time to take into account those patches that are now lit. They, in turn, will light other patches, and so on, and so on, and the process is repeated until a given number of **passes** or **bounces** are achieved – with each pass, the process converges on a stable image. Because these calculations are pre-processed the results can be presented interactively as they are view-independent.

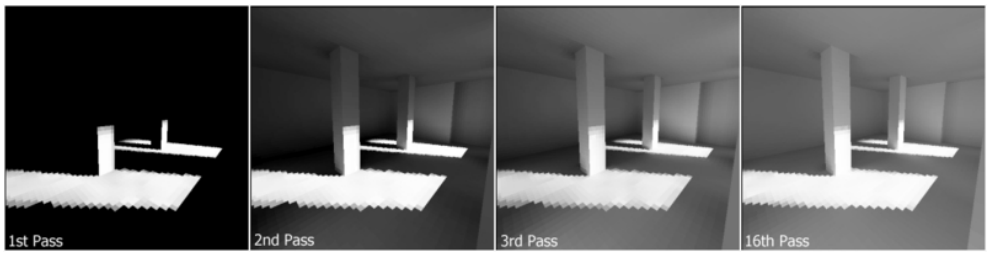


Figure 7.3: As the algorithm iterates, light can be seen to flow into the scene, as multiple passes are computed. Individual patches are visible as squares on the walls and floor. Source: By Hugo Elias; reproduced under Creative Commons licence CC BY-SA 3.0 from: `http://en.wikipedia.org/wiki/Radiosity_(computer_graphics)#mediaviewer/File:Radiosity_Progress.png`

The advantages of radiosity are that it gives good results for effects such as colour bleeding. Interactivity is achieved since the scene calculations are pre-computed and only need to be done once. It is good for scenes (especially indoor scenes) where most lighting is indirect and where ray tracing copes badly. However, it is not very good for scenes involving transparency and non-diffuse reflection as it cannot handle specular objects and mirrors.

Learning activity

Hugo Elias has written an excellent explanation of a radiosity renderer from the point of view of a patch. Read through it and study his examples:
`http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm` (accessed May 2014).

7.9 Image-based lighting

Image-based lighting (IBL) is the process of illuminating scenes and objects (real or synthetic) with images of light from the real world. This increases the level of realism of a scene and is a technique that is becoming more commonplace given its potential for applications such as motion-picture visual effects, where light from a CG scene can be integrated with light from a real world scene. Environment mapping is a forerunner of this technique, where an image of an environment is directly texture-mapped on to an object surface. However, environment mapping does not take into account interreflections. IBL provides a more faithful link between light in the real world and light in the virtual world.

The processes behind IBL are those of global illumination and also high dynamic range imaging (HDRI), which is discussed in Chapter 10 of this subject guide.

The steps involved are as follows:

- Real-world illumination is captured as an omni-directional light probe image. In essence, a video camera captures a series of images of reflections in a mirrored sphere.
- The computer generated scene geometry is created and objects are assigned material values.
- The captured images are mapped onto a representation of the environment (for example, as a sphere encompassing the modelled scene).
- The light in that environment is simulated using ray tracing with the light probe image as the light source.

Learning activity

Fiat Lux – The Movie is a short film demonstrating IBL by the originator of this technique, Paul Debevec. Watch it online at `http://www.pauldebevec.com/FiatLux/movie/` (accessed May 2014).

7.10 Applications for realistic graphics

There are a number of areas of research where it is necessary for images to appear as realistic as possible. Architectural visualisations, for example, rely on selling a plausible vision of the finished building. For applications that involve simulation (for example, training environments for the aviation industry), or precise data interpretation (such as forensic reconstructions), the user must be confident that the image they are viewing is faithful to the real world — they require perceptual fidelity.

Predictive lighting for archaeology

Predictive lighting has been used to great advantage in the representation of archaeological sites and artefacts, with the aim of depicting the environment as it would have looked to an observer situated within it.

An often neglected facet of reconstructions is the lighting. Standard three-dimensional modelling packages do not allow precise control over lighting values, and as a result, many simulations show the archaeological site under inappropriate or false lighting conditions such as the bright, steady light of the present. Experimental archaeology and realistic lighting simulation allow us to recreate the original lighting of an archaeological site and show how it might have looked to those who built and used it. Predictive lighting also opens up new avenues of exploring how past environments may have been perceived, allowing us to investigate on a computer new hypotheses about architecture, art and artefacts in a safe, non-invasive manner.

The method uses information about the geometry of an environment, the material properties of the surfaces in that environment and the spectral properties of the light sources to generate a mathematical simulation that we can, to some degree, consider accurate. This enables us to explore new hypotheses and examine the way in which things were viewed and understood in the past by manipulating variables and working with virtual objects in a way that is not possible with a real site or artefact.

Having simulated a scene where one can have confidence in its fidelity, the resulting images can be used to generate and test new hypotheses regarding perception of past environments and our overall impression of the scene – the lights, the colours, the shadows, the features that are visible and not visible – should appear the same to us as they would to those who originally occupied that space. We are working with a simulation of their surroundings that can be said to be physically valid. This provides us with the option of changing and manipulating variables in a manner that is not possible in real life, such as introducing dynamic flames and the effects of smoke, or adding artefacts or furnishings to interiors, and we can do this with a good degree of confidence because we have had control over the input to the process.

Perceptual consistency is desirable in cultural heritage applications in the capture and creation of images that can be used as perceptually equivalent representations of an original. Virtual reality and visualisation methods can provide highly detailed models of sites and artefacts. Advances in scanning and digital photography have led to the widespread use of this technology to digitally preserve original text and art. For digital archiving to be used as a technique for representation or preservation, the integrity of an image must be affirmed.

Case study for lighting simulation: Pompeii frescoes

For highly-decorative interiors, predictive lighting can be useful in testing how a room may have been laid out or used by the original inhabitants. The UNESCO World Heritage site of the Archaeological Areas of Pompeii, Ercolano and Torre Annunziata contain fine examples of Roman frescoes. The House of the Vettii in Pompeii was chosen for the study, with the work focusing on a reception room off the colonnaded sculpture garden. This room is lavishly decorated in the IV Style (Figure 7.4) and was chosen due to the rich colours, good state of preservation, and artistic effects such as *trompe l'oeil*, a painting technique that deceives the eye into viewing a two-dimensional image as having a three-dimensional structure.

The frescoes were recorded photographically, with the use of a colour chart for calibration purposes and to identify illumination levels. A three-dimensional model was generated from a scale plan. The most readily available fuel type for this area was deemed to be olive oil, so the spectral profile of the olive oil lamps was used to illuminate the scene. Also, a technique for including real flame captured from video

Figure 7.4: The room in the House of the Vettii as it appears today.

footage and inserted in the virtual scene gave a realistic appearance to the lamps without having to model the actual flame. Therefore, the virtual scene contained the correct illumination levels for a scene lit by olive oil lamps, with a real flame incorporated (Figure 7.7).

In the resulting images it is plainly demonstrable how the scenes vary depending on how they are illuminated. Under modern lighting conditions (Figure 7.5) such as we might see today, the colours are not as vibrant as they appear under lamp light (Figure 7.6). When viewed under olive oil lamps, the red and yellow paint of the frescoes is particularly well-emphasised. Also, the *trompe l'oeil* artwork resembling mock windows and external architecture actually takes on the appearance of a real view to the exterior as the three-dimensional depth cues are increased.

Figure 7.5: Simulation viewed under modern lighting.

Figure 7.6: Simulation viewed under olive oil lamps.

By changing the number and the positions of the light sources in the room, various effects can be achieved.

It is possible to test how lighting may have been distributed in order to highlight the artwork in the most effective manner. Such positioning of lighting may have determined the arrangement of furniture in a room. Again, such manipulations are possible when working with a virtual version of the scene.

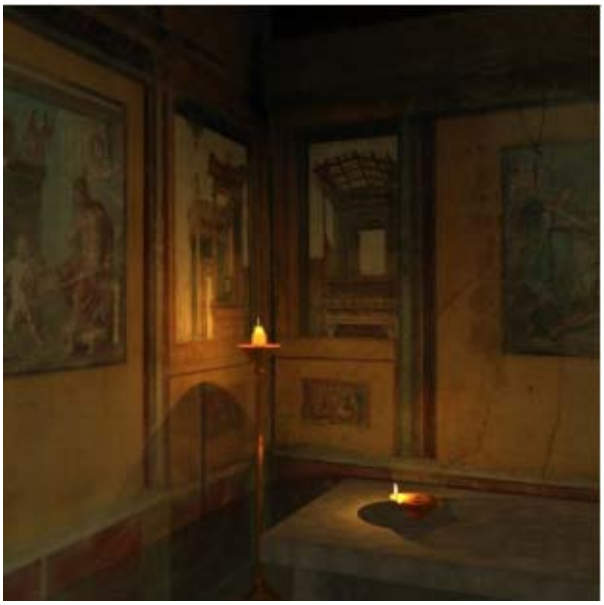


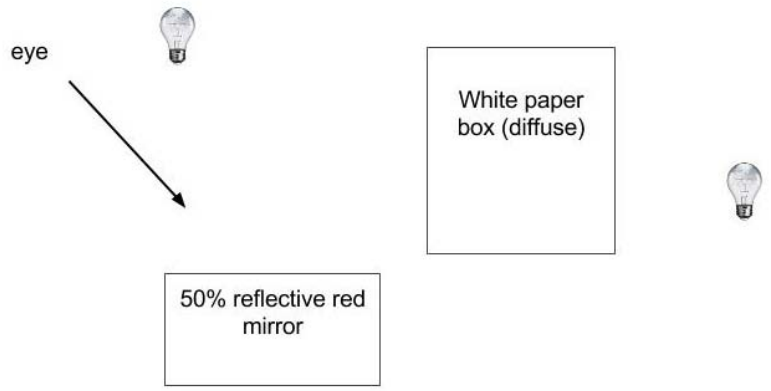
Figure 7.7: Simulation viewed under olive oil lamps, with furniture to show shadow effects.

7.11 Summary

This chapter has explained the techniques involved in global illumination, indicating the strengths and weaknesses of these techniques. It has discussed the need for realistic images for certain applications. The case study on virtual heritage has provided an example as to why applications may require physically accurate images.

7.12 Exercises

1. The diagram below has a mirror, a white paper box, and two lights. Starting from the eye ray drawn, draw all additional reflection, refraction and shadow rays needed to compute the colour of the eye ray.



7.13 Sample examination questions

1. What are the differences between real-time graphics and offline (photorealistic) computer graphics? In this context, also explain why graphics hardware (for example, graphics cards) are useful for computer graphics.
2. Describe how shadows are determined in a recursive ray tracer. Extend your answer to consider polygonal area light sources.
3. Describe the process of image-based lighting.
4. Explain, step-by-step, the basic principles in the radiosity approach to producing computer generated images. State two advantages and two disadvantages of this technique.

Chapter 8

Animation

8.1 Essential reading

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapter 17.

8.2 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- explain the basic concepts of computer animation including keyframe animation
- implement a basic keyframe animation system in a computer program
- explain a number of character animation techniques including skeletal animation and morph targets
- select an appropriate animation technique for a particular application
- implement basic aspects of character animation.

8.3 Introduction

Computer animation is about 'making things move'; a key part of computer graphics. In film, animations are created off-line, possibly taking days of animator time per scene. However, in games everything must be done in real time. In practice, this generally means creating a number of animations off line, either animating them by hand or using motion capture, which are then triggered in real time based on game events. Animation can also be created procedurally using a physics engine or a particle system. This part of the course will primarily cover playback of pre-created animation (called keyframe animation). This is mostly applied to character animation, so that will be the focus of the course.

Note: **Creative computing II** covers the basics of computer animation. As not all students of this course will have studied **Creative computing II** this chapter will recap material from that course before moving on to more advanced material.

8.4 Traditional animation

Before coming to computer animation let us talk briefly about traditional animation. The most basic form of animation is the flip book. It presents a sequence of images in quick succession, each of which is a page of the book. Each image changes slightly from the previous one. If you flip through the pages fast enough the images are presented one by one and the small changes no longer seem like a sequence of

individual images but a continuous sequence. In film, this becomes a sequence of frames, which are also images, but they are shown automatically on a file projector.

Cell animation works on the same principle. A sequence of images is displayed at 25 frames per second (the minimum to make it appear like smooth motion). For the purpose of creating animation these images are arranged in a 'time line'. In traditional animation this is a set of images with frame numbers drawn by the side. Each frame is an image. Traditionally each image had to be hand drawn individually. This potentially required vast amounts of work from a highly skilled animator. What is more, in order to make the final animation look consistent each character should always be drawn by the same animator. Disney and other animation houses developed techniques to make the process more efficient. Without these methods full length films like *Snow White* would not have been possible.

One technique is **layering**. There is a background image that does not move and you put foreground images on a transparent slide in front of it. You only have to animate bits that move. Next time you watch an animation, notice that the background is always more detailed than the characters. Asian animation often uses camera pans across static images.

Possibly the most important method is **keyframing**. You only need a few images to get the entire feel of a sequence of animation, but you would need many more to make the final animation look smooth. The head animator for a particular character draws the most important frames (keyframes). An assistant draws the in-between frames (inbetweening).

Other methods are aimed at making the animation look more realistic and expressive. 'Squash and stretch' is about changing the shape of an object to emphasise its motion. In particular, stretching it along the direction of movement when going fast and then squashing when changing direction. 'Slow in slow out' is about controlling the speed of an animation to make it seem smoother. Start slow, speed up in the middle and slow to a stop.

Stop motion animation is a very different process. It involves taking many still photographs of real objects instead of drawing images. The object is moved very slightly after each photograph to get the appearances of movement. More work is put into creating characters. You can have characters with a lot of detail and character creators will spend a lot of effort making characters easy to move. For example, Aardman animations use metal skeletons underneath their clay models, so that the characters can be moved easily and robustly without breaking. Each individual movement is then less work (though still a lot).

8.5 Computer animation

Computer animation (both 2D and 3D) is quite a lot like Stop Motion Animation. You put a lot of effort into creating a (virtual) model of a character and then when animating it you move it frame by frame. You will spend a lot of time making easy-to-use controls for a character, a process called **rigging**. For basic objects these will just be transforms like translations and rotation but human characters will have complex skeletons, analogous to the metal skeletons Aardman Animations use (more on this later). Once you have completed this set up effectively, animation becomes much simpler.

Keyframing

Keyframing can reduce this effort even more. The animator only needs to define the 'key frames' of a movement (which will be values for transforms). The computer does the inbetweening automatically. Keyframes are a tuple consisting of the time at which the keyframe occurs and the value of the transforms. These will be set up in a timeline, which is just an array of keyframes.

As an example we could define a basic keyframe class that had keyframes on position and would look something like this:

```
public class Keyframe
{

  PVector position;
  float time;

  public Keyframe (float t, float x, float y, float z) {
    time = t;
    position = new PVector (x,y,z);
  }
}
```

Code example 8.1: A keyframe class.

A timeline would essentially be an array of these keyframe objects:

```
Keyframe [] timeline;
```

In a full program this would be bundled into a complete timeline class, but for a basic demo we can just use the array directly by adding keyframes to it:

```
void setup(){
  size(640, 480);

  timeline = new Keyframe [5];
  timeline[0] = new Keyframe(0, 0, 0, 0);
  timeline[1] = new Keyframe(2, 0, 100, 0);
  timeline[2] = new Keyframe(4, 100, 100, 0);
  timeline[3] = new Keyframe(6, 200, 200, 0);
  timeline[4] = new Keyframe(10, 0, 0, 0);
}
```

Code example 8.2: Creating keyframes.

Note that we are adding keyframes in the correct time order. We will use the ordering later when we have to find the current keyframe.

Now we have the timeline we can get hold of the positions at a certain key frame and use them to translate our object. This is an example of how to get keyframe 0:

To play an animation back effectively we need to be able to find the current keyframe based on time. We can use the millis command in *Processing* to get the current time

```
PVector pos = timeline[0].position;
pushMatrix();
  translate(pos.x, pos.y);
  ellipse(0, 0, 20, 20);
popMatrix();
```

Code example 8.3: Applying a keyframe to an object transform.

in milliseconds. We can now search for the current keyframe. We need to find the keyframe just before the current time. We can do that by finding the position in the timeline where the keyframe is less than t but the next keyframe is more than t:

```
float t = float(millis())/1000.0; // convert time from milliseconds to second
int currentKeyframe;
for (currentKeyframe = 0;
     currentKeyframe < timeline.length-1;
     currentKeyframe++)
{
  if(timeline[currentKeyframe].time < t
    && timeline[currentKeyframe+1].time > t)
    break;
}
PVector pos = timeline[currentKeyframe].position;
```

Code example 8.4: Finding the current keyframe.

This is quite a complex loop with a number of unusual features. Firstly, we are defining the loop variable `currentKeyframe` outside the loop. That is because we want to use it later in the program. Secondly, we are not going to the end or the array but to the position before the end. This is because we are checking both the current keyframe and the next keyframe. At the end of the array there is no next keyframe. Finally, we are using the break statement to break out of the loop when we have found the right keyframe. Because of the way loops work, if we never break out of the loop `currentKeyframeM` will be set to the last index of the array, which is what we want, because it means that the current time is after the last keyframe.

Once we have found the current keyframe we can use it to get a position:

```
PVector pos = timeline[currentKeyframe].position;
```

Learning activity

Implement a *Processing* sketch that animates an object using the keyframe code above.

Implement a complete timeline class and use it to animate game objects as we defined in Chapter 4 of this subject guide.

The example above implements keyframes but the animation is not at all smooth. The object instantly jumps from one keyframe position to the next, rather than gradually moving between the keyframes. The frames between the keyframes have

to be filled in (interpolated). For example, if you have the following positions of the ball.

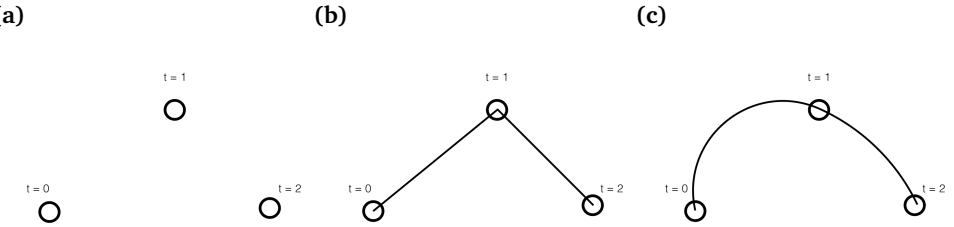


Figure 8.1: Keyframe animation. (a) Three keyframes for a ball. (b) Linear interpolation. (c) Spline interpolation.

The simplest approach is to interpolate them in straight lines between the keyframes (Figure 8.1(b)). The position is interpolated linearly between keyframes using the following equation:

$$P(t) = tP(t_k) + (1 - t)P(t_{k-1}) \tag{8.1}$$

Where t is the time parameter. The equation interpolates between keyframe $P(tk - 1)$ and keyframe $P(t)$ as t goes from 0 to 1. This simple equation assumes that the keyframe times are 0 and 1. The following equation takes different keyframe values and normalises them so they are between 0 and 1.

$$P(t) = \frac{t - t_{k-1}}{t_k - t_{k-1}} P(t_k) + \left(1 - \frac{t - t_{k-1}}{t_k - t_{k-1}}\right) P(t_{k-1}) \tag{8.2}$$

To implement this equation we can use the code in Code example 8.5. The code itself is explained in the comments.

The animation can be jerky, as the object changes direction of movement rapidly at each keyframe. To improve this we can do **spline interpolation** which uses smooth curves to interpolate positions (Figure 8.1(b)). Bézier curves (Figure 8.2(a)) would be an obvious choice of curve to use as they are smooth but they do not go through all the control points; we need to go through all the keyframes. As we need to go through the keyframes we use **Hermite curves** instead (Figure 8.2(b)). These are equivalent to Bézier curves, but rather than specifying four control points specify two end points and tangents at these end points. In the case of interpolating positions the tangents are velocities.

The formula for a Hermite curve is:

$$\begin{aligned} P(t) =& (-2s^3 + 3s^2)P(t_k) + (s^3 - s^2)T(t_k) \\ &+ (2s^3 - 3s^2 + 1)P(t_{k-1}) + (s^3 - 2s^2 + s)T(t_{k-1}) \end{aligned}$$

Where s is the time t, rescaled to be between 0 and 1:

```
PVector pos;

// first we check whether we have reached the last keyframe
if(currentKeyframe == timeline.length-1)
{
  // if we have reached the last keyframe,
  // use that keyframe as the position (no interpolation)
  pos = timeline[currentKeyframe].position;
}
else
{
  // This part does interpolation for all keyframes before
  // the last one

  // get the position and time of the keyframe before
  // and after the current time
  PVector p1 = timeline[currentKeyframe].position;
  PVector p2 = timeline[currentKeyframe+1].position;
  float t1 = timeline[currentKeyframe].time;;
  float t2 = timeline[currentKeyframe+1].time;

  // multiply each position by the interpolation
  // factors as given in the linear interpolation
  // equation
  p1 = PVector.mult(p1, 1.0-(t-t1)/(t2-t1));
  p2 = PVector.mult(p2, (t-t1)/(t2-t1));

  // add the results together to get the
  // interpolated position.
  pos = PVector.add(p1, p2);
}
```

Code example 8.5: Interpolating keyframes.

$$s = \frac{t - t_{k-1}}{t_k - t_{k-1}}$$

Where do we get the tangents (velocities) from? We could directly set them; they act as an extra control on the behaviour. However, often we want to generate them automatically. We can base the tangent at a keyframe on the previous and next keyframe. They can be calculated as the average of the distance from the previous keyframe and to the next one. This distance is divided by the time between the previous and next keyframe in order to get the correct speed of movement:

$$T(t_k) = \frac{P(t_{k+1}) - P(t_{k-1})}{t_{k+1} - t_{k-1}} \tag{8.3}$$

If tangents are calculated in this way the curves are called **Catmull-Rom splines**. If you set the tangents at the first and last frame to zero you get ‘slow in slow out’.

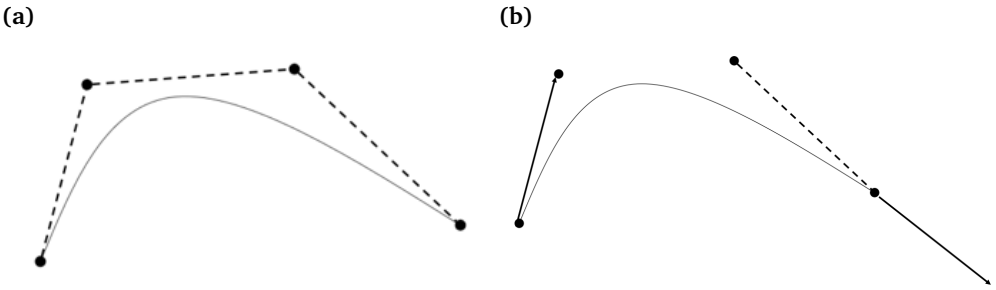


Figure 8.2: Spline interpolation. (a) Bézier curve. (b) Hermite curve.

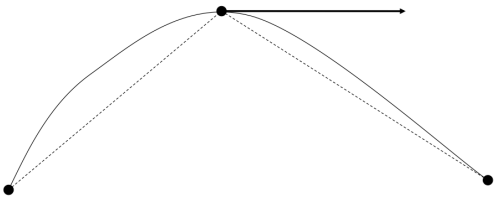


Figure 8.3: Calculating tangents.

Learning activity

Adapt the time line code developed in the the previous activity to include keyframe interpolation as shown in Code example 8.5. As an optional extension adapt the code to work with hermite spline interpolation (you will need to add tangents to the keyframe class).

Adapt the keyframe animation code to also work with rotations. You can represent a 3D rotation as a separate rotation about x, y and z. In practice, this representation can lead to some problems in animation. Most 3D animation systems use an alternative representation of rotations called Quaternions. An optional extension of this work would be to read about quaternions and how they apply to 3D animation (they are covered in Shirley et al., Chapter 17).

8.6 Human character animation

Character animation is normally divided into **Body animation** and **Facial animation** each of which uses different techniques. Within each there are a number of important topics:

- Body animation
  - skeletal animation
  - skinning
  - motion capture
- Facial animation
  - morph targets
  - facial bones
  - facial motion capture.

8.7 Skeletal animation

The fundamental aspect of human body motion is the motion of the skeleton. To a first approximation this is the motion of rigid **bones** linked by rotational **joints** (the terms joints and bones are often used interchangeably in animation; although, of course, they mean something different, the same data structure is normally used to represent both). We will discuss other elements of body motion such as muscle and fat briefly later.

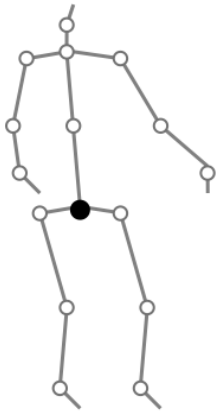


Figure 8.4: A character skeleton.

Circles are rotational joints, lines are rigid links (bones). Joints are represented as rotations. The black circle is the root, the position and rotation offset from the origin. The root is (normally) the only element of the skeleton that has a translation. The character is animated by rotating joints and translating and rotating the root.

The position of a bone is calculated by concatenating rotations and offsets; this process is called **forward kinematics** (FK). First choose a position on a bone (the end point). This position is rotated by the rotation of the joint above the bone. Translate by the length (offset) of the parent bone and then rotate by its joint. Go up its parent and iterate until you get to the root. Rotate and translate by the root position.

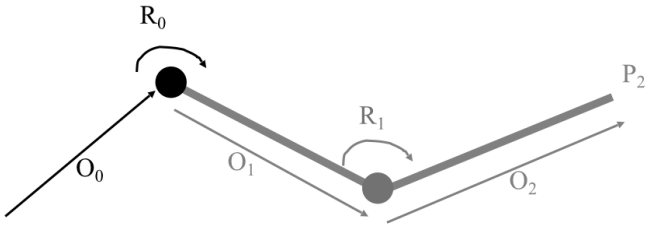


Figure 8.5: Forward kinematics.

$$P_2 = R_0(R_1(O_2) + O_1) + O_0 \tag{8.4}$$

Joints are represented as transforms. As in most graphics systems transforms are hierarchical; FK can easily be implemented using the existing functionality of the engine. You can build a skeleton as a hierarchy of graphics objects of the kind described in Chapter 4 of this subject guide. Each joint is represented as a graphics object and its child joints will be child graphics objects (for example, the elbow will be a child of the shoulder, the left and right thighs will both be children of the hips). Bones have geometry attached in some way. The most basic way would be to have a separate piece of geometry attached to each graphics object representing a joint. Below we will discuss a more sophisticated approach called **skinning**.

Joints are generally represented as full three degrees of freedom rotations but human joints cannot handle that range. Either you build rotation limits into the animation system or you can rely on the methods generating joint angles to give reasonable values (as motion capture normally will).

Learning activity

Create a simple character as a hierarchy of graphics objects, each of which represents a bone or joint. The geometry can be very simple; for example, a simple box for each bone, but you should be able to move the character by rotating the joints.

Adapt your keyframe animation code so that you can set keyframes for all of the joints of your character and therefore animate the character.

Inverse kinematics

Forward kinematics is a simple and powerful system. However, it can be fiddly to animate with. Making sure that a hand is in contact with an object can be difficult. For example, when animating you often want to do things like make a character's hand touch a door handle. Trying to get the joint angles right so as to accurately place the hand can be a long and difficult process.

Inverse kinematics is a way of doing this automatically so that you can animate in terms of hand and foot positions rather than joint angles. Given a desired position for a part of the body (end effector) inverse kinematics is the process of calculating the required joint angles to achieve that position (in the above diagram, given P_t IK will calculate R_0 and R_1).

We could try to work out an exact (analytic) formula, but this would be specific to a given number of links. It would also be underconstrained for more than two links; that is, there is more than one solution (you can hold your shoulder and wrist still but still rotate your elbow into different positions). There are a number of approaches to performing inverse kinematics:

- Matrix (Jacobean) methods.
- **Cyclic Coordinate Descent** (CCD).
- Specific analytic methods for the human body.

Here we will only look at CCD as it is the simplest.

Cyclic Coordinate Descent

This is an iterative geometric method. You start with the final link and rotate it towards the target. Then go to the next link up and rotate it so that the end effector points towards the target; you then move up to the next joint. Once you reach the top of the hierarchy you need to go back down to the bottom and iterate the whole procedure again until you hit the correct end effector position.

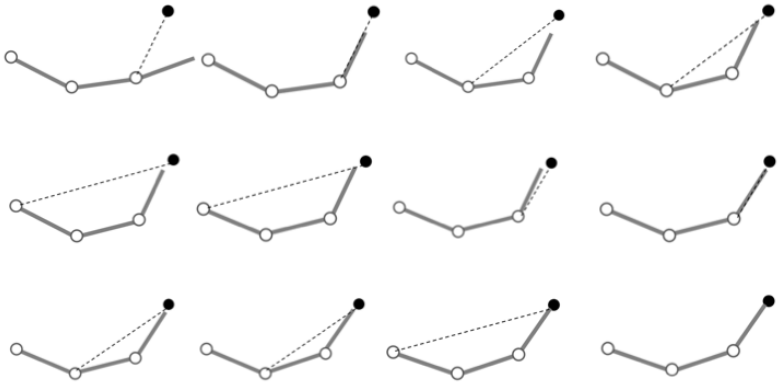


Figure 8.6: The stages in Cyclic Coordinate Descent.

CCD is very general and powerful; it can work for any number and combinations of bones. However, there are problems. It does not know anything about the human body. It can put you in unrealistic or impossible configurations (for example, elbow bent backwards). To avoid this we need to introduce joint constraints. CCD makes this easy; you constrain the joints after each step.

It is also very computationally expensive; it is an iterative algorithm and often takes a long time to converge. This means that it is not suitable for real time use.

Skinning

A skeleton is a great way of animating a character but it does not necessarily look very realistic when rendered. We need to add a graphical 'skin' around the character. The simplest way is to make each bone a transform and hang a separate piece of geometry off each bone. This works but the body is broken up (how most games worked 15 years ago). We want to represent a character as a single smooth mesh (a 'skin'). This should deform smoothly based on the motion of the skeleton.

The mesh is handled on a vertex by vertex basis. Each vertex can be associated with more than one bone. The effect on each vertex is a combination of the transformations of the different bones. The effect of a bone on a vertex is specified by a weight, a number between 0 and 1. All weights on a vertex sum to 1.

The vertices are transformed individually by their associated bones. The resulting position is a weighted sum of the individual joint transforms.

$$T(v_i) = \sum_{j \in joints} wijR_j(v_i) \tag{8.5}$$

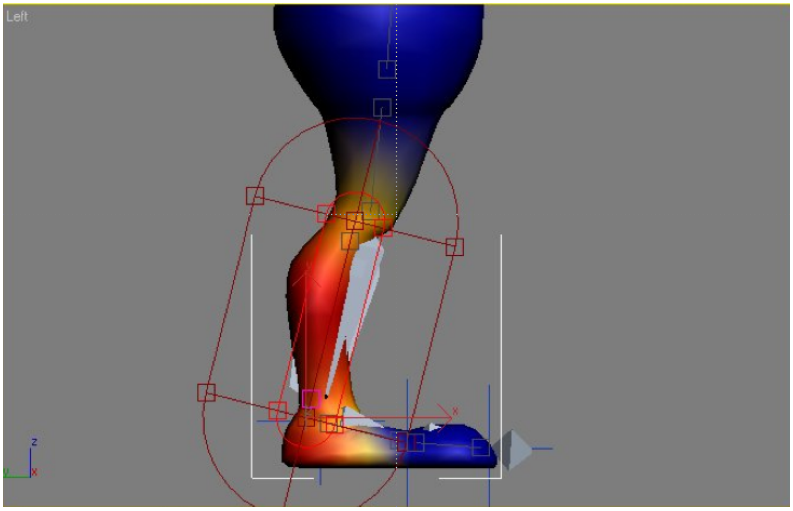


Figure 8.7: Skinning weights.

A character is generally rigged with the skeleton in a default pose, called the **bind pose**, but not necessarily zero rotation on each bone. If the skeleton is in the bind pose the mesh should be in its default location. If the bind pose is not zero rotation you need to subtract the inverse bind pose from the current pose.

Skinning is well suited to implementing in a shader. The bone weights are passed in as vertex attributes and an array of joint transforms is passed in as a uniform variable. The shader transforms each vertex by each bone transform in turn and then adds together the results multiplied by the weights. In order to limit the number of vertex attributes we normally have a limit of four bones per vertex and use a vec4 to represent the bone weights. This means you also need to know which bones correspond to which weights, so you also have a second vec4 attribute specifying bone indices.

The deformation of a human body does not just depend on the motion of the skeleton. The movement of muscle and fat also affect the appearance. These soft tissues need different techniques from rigid bones. More advanced character animation systems use multiple layers to model skeleton, muscle and fat. These could use geometric methods (for example, free form deformations based on NURBS); or simulation methods (model physical properties of fat and muscle using physics engines). Hair is also normally modelled using physics simulation.

8.8 Facial animation

The face does not have a common underlying structure like a skeleton. Faces are generally animated as meshes of vertices, either by moving individual vertices or by using a number of types of rig.

Morph targets are one of the most popular methods. Each facial expression is represented by a separate mesh. Each of these meshes must have the same number of vertices as the original mesh but with different positions. New facial expressions are created from these base expressions (called **Morph targets**) by smoothly

```
// Here are the variables:

// use a uniform variable for the bone transformations
uniform mat4 Transforms[58];

// use an attribute for the bone weights for each vertex
// this is a vec4 so you can only have 4 bone weights
attribute vec4 Weights;

// you need another attribute to tell you which bones
// the weights correspond to
attribute vec4 MatrixIndices;

// here is the code to do the transformation:

// build up a transform matrix by adding up the transform
// matrices for each bone multiplied by the weights
mat4 transform    = Weights.x*Transforms[int(round(MatrixIndices.x))];
transform += Weights.y*Transforms[int(round(MatrixIndices.y))];
transform += Weights.z*Transforms[int(round(MatrixIndices.x))];
transform += Weights.w*Transforms[int(round(MatrixIndices.w))];

// multiply the vertex position by this transform
gl_Position = transform *gl_Vertex;
// do the same for your normals
normal = vec3(transform*vec4(gl_Normal,0.0));
```

Code example 8.6: An example of a GLSL shader to do skinning.

blending between them.

Each target is given a weight between 0 and 1 and a weighted sum is performed on all of the vertices in all of the targets to get the output mesh:

$$v_i = \sum_{t \in \text{morph targets}} w_t v_{ti}; \text{where} \sum w_t = 1 \tag{8.6}$$

To implement morph targets we need a vertex shape that we want to animate:

```
PShape base;
```

We also need an array of vertex shapes to represent the morph targets:

```
PShape [] morphs;
```

These need to have an exactly identical structure to each other and to the base shape. That means they need to have exactly the same number of child shapes and each child shape must have exactly the same number of vertices. It is generally a good idea to start with one basic shape (morph[0]) and edit it to create the other morphs. The same shape can be initially loaded into base and morph[0] (but they must be loaded separately, not simply two variables pointing to the same shape otherwise editing base will also change morph[0]).

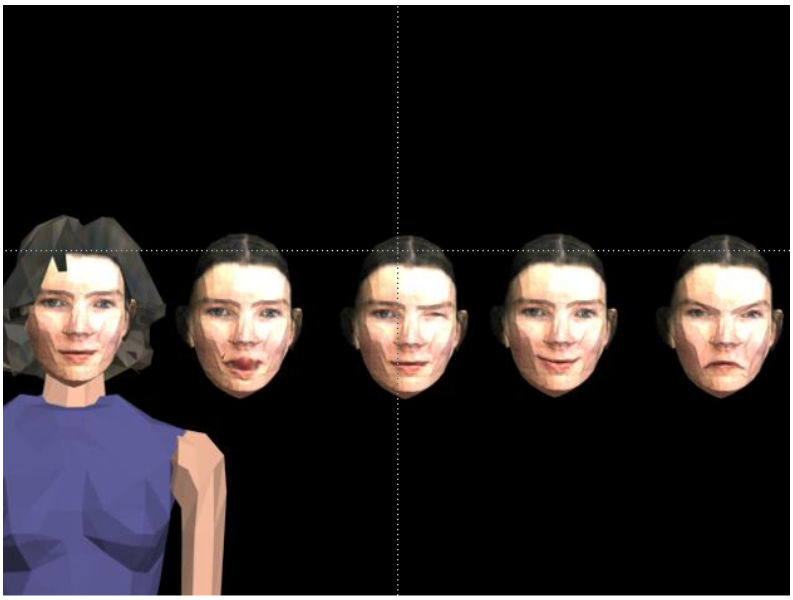


Figure 8.8: Morph targets.

We also need an array of weights:

float [] weights;

Once we have all of these in place we can modify the base shape, by iterating through all the vertices and calculating a new vertex position using Equation 8.6, above. This implementation assumes that the shape is composed of a number of child shapes (this is often the case when a shape is loaded from an obj file).

In the previous formula the weights have to sum to 1. If they do not, the size of the overall mesh will increase (if the weights sum to more than 1) or decrease (if they sum to less than 1). Subtracting a neutral mesh from all the targets allows us to lift the restriction because we are adding together differences not absolute positions. This can allow more extreme versions of a target or the use of two complete morph targets simultaneously (for example, a smile together with closed eyes).

$$v_i = v_{0i} + \sum_{t \in \text{morph targets}} w_t(v_{ti} - v_{oi}) \tag{8.7}$$

Learning activity

Create a 3D object, either in a modelling tool like *Blender* or using code based on `createShape`. Create a number of modified versions of that object to act as morph targets. Create a program based on the code in Code example 8.7 to animate the object using the morph targets.

Modify the code to use the difference based morphs method given in Equation 8.7 above.

Modify your keyframe animation code to create animations of the morph targets.

```
// iterate over all children of the base shape
for (int i = 0; i < base.getChildCount(); i++)
{
  // iterate over all vertices of the current child
  for (int j = 0; j < base.getChild(i).getVertexCount(); j++)
  {
    // create a PVector to represent the new
    // vertex position
    PVector vert = new PVector(0, 0, 0);
    // iterate over all the morph targets
    for (int morph  = 0; morph < morphs.length; morph++)
    {
      // get the corresponding vertex in the morph target
      // i.e. the same child and vertex number
      PVector v = morphs[morph].getChild(i).getVertex(j);
      // multiply the morph vertex by the morph weight
      // and add it to our new vertex position
      vert.add(PVector.mult(v, weights[morph]));
    }
    // set the vertex position of the base object
    // to the newly calculated vertex position
    base.getChild(i).setVertex(j, vert);
  }
}
```

Code example 8.7: Implementing morph targets.

Using morph targets

Morph targets are a good low level animation technique. To use them effectively we need ways of choosing morph targets. We could let the animator choose (nothing wrong with that) but there are also more principled ways.

Psychologist Paul Ekman defined a set of six universal human emotions (joy, sadness, surprise, anger, disgust, fear), which he called the **basic emotions**. All are independent of culture and each has a distinctive facial expression. They are a common basis for morph targets but can be very unsubtle.

An important problem is how to animate people talking. In particular, how to animate appropriate mouth shapes for what is being said (Lip-sync). Each sound (phoneme) has a distinctive mouth shape; we can create a morph target for each sound (visemes). Analyse the speech or text into phonemes (automatically done by text to speech engine), match phonemes to visemes and generate morph target weights.

Other approaches to facial animation

There is plenty more to facial animation than morph targets, often related to body animation techniques:

- facial bones

- muscle models
- facial action parameters
- facial motion capture.

Facial bones are similar to bones in body animation. A set of underlying objects that can be moved to control the mesh. Each bone affects a number of vertices with weights in a similar way to smooth skinning for body animation. Facial bones essentially use the same mechanisms as skeletal animation and skinning. The main difference is that facial bones do not correspond to any real structures. Facial bones are normally animated by translation rather than rotation as it is easier. This is very useful as it means you only have one method in your animation code (one shader). Morphs are very convenient from an animator point of view, but bones are easier in the engine.

A more principled approach is to model each of the muscles of the face. In fact this can be done using one of the techniques just discussed. Each muscle could have a morph target, or a bone, or there could be a more complex physical simulation as mentioned for multi-layered body animation.

Paul Ekman invented a system of classifying facial expressions called Facial Action Parameters (FAPs) which is used for psychologists to observe expressions in people. It consists of a number of parameters each representing a minimal degree of freedom of the face. These parameters can be used for animation. FAPs really correspond to the underlying muscles so they are basically a standardised muscle model. Again they can be implemented as morph targets or bones.

8.9 Motion capture

Keyframe animation is based on sets of movement data which can come from one of two sources: hand animation or **motion capture**. Hand animation tends to be very expressive but has a less realistic more cartoon-like style. It is easy to get exactly what you want with hand animated data and to tweak it to your requirements. Motion capture can give you potentially very realistic motion but is often ruined by noise, bad handling, etc. It can also be very tricky to work with. Whether you choose motion capture or hand animation depends on what you want out of your animation: a computer graphics character that is to be inserted into a live action film is likely to need the realism of motion capture, while a children's animation might require the more stylised movement of hand animation.

Motion capture is a general term for technologies that can record the full 3D motion of the human body. There are many different types of motion capture.

Magnetic

This involves puting magnetic transmitters on the body. The positions of these transmitters are tracked by a base station. These methods are very accurate but expensive for large numbers of markers. The markers also tend to be relatively heavy. They are generally used for tracking small numbers of body parts rather than whole body capture.

Mechanical

This involves putting strain gauges or mechanical sensors on the body. These are self contained and do not require cameras or a base station making them less

constrained in terms of the space in which you do it. They also have the benefit of directly outputting joint angles rather than marker positions. They can be bulky, particularly the cheap systems. It can be uncomfortable and constraining to wear resulting in less realistic motion. Lighter-weight systems have recently been developed but they can be expensive.

Optical

This is the most commonly used system in the film and games industries. Coloured or reflective balls (markers) are put on the body and the positions of these balls are tracked by multiple cameras. The cameras use infra-red to avoid problems of colour. Problems of occlusion (markers being blocked from the cameras by other parts of the body) are partly solved by using many cameras spread around a room. The markers themselves are lightweight and cheap although the cameras can be expensive and require a large area.

Markerless optical

These systems use advanced computer vision techniques to track the body without needing to attach markers. These have the potential to provide an ideal tracking enivornment that is completely unconstrained. However, it involves very difficult computer vision techniques. The Microsoft Kinect has recently made markerless motion capture more feasible by using a depth camera, but it is still less reliable and accurate than marker based capture. In particular, the Kinect only tends to be able to capture a relatively constrained range of movements (reasonably front on and standing up or sitting down).

Motion capture can also be used for facial movement. Again the movement of an actor can be captured and applied to a character in a similar way to body motion capture. Body motion capture relies on putting markers on the body and recording their movement but generally these are too bulky to use on the face. The only real option is to use optical methods. Markerless systems tend to work better for facial capture as there are less problems of occlusion and there are more obvious features on the face (eyes, mouth, nose). They do, however, have problems if you move or rotate your head too much.

8.10 Summary

Animation is a vital part of interactive 3D graphics and computer graphics films. It is one which does a lot to create character and appeal. There are many animation techniques used in modern computer graphics, most of which aim at simplifying the animation process, by not requiring animators to move every fine detail of an object in every frame. Examples include keyframing (not moving an object every frame) and skeletal animation (not moving every detail of a character). These approaches give animators a lot of control over the movement of a character while minimising their workload. However, other approaches to animation are not based on the work of an animator but instead rely on simulating the movement of objects; we will cover this type of simulation in the next chapter of this subject guide.

8.11 Sample examination questions

1. What are morph targets? What are the benefits and drawbacks of using morph targets for facial animation as opposed to facial bones?
2. Early 3D games such as *Quake* created animated models by directly setting keyframes on the vertices of a mesh. Why do you think this approach was abandoned in favour of using skeletal animation? Can you see any reason why a modern animation studio might return to this approach?
3. Describe one context in which you would use linear interpolation of keyframes and one in which you would use spline interpolation. Explain the reasons for your choice in each case.

Chapter 9

Physics simulation

9.1 Essential reading

Angel, E. and Shreiner, D. *Interactive Computer Graphics.* (2011), Chapter 9.

Chapters 1, 2 and 5 of the following book cover physics simulation in *Processing*:

Shiffman, D. *The Nature of Code.* (USA, 2012) [ISBN 9780985930806].

The book is also available as a free online book (in fact the online version is recommended as it includes interactive examples):

`http://natureofcode.com/book/` (accessed April 2014).

9.2 Recommended reading

The following book is considerably more advanced than the level we expect in this course, but, if you would like to go into greater depth into the workings of a physics engine it explains them in detail.

Millington, I. *Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game.* (Elsevier; Morgan Kaufmann, 2010) [ISBN 9780123819765].

9.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- explain the basic principles of physics simulation for computer animation
- explain the role of rigid bodies, forces and constraints in a physics simulation
- demonstrate the use of a physics engine to create a basic simulation
- create a simulated scene using rigid bodies and manipulate the properties of those rigid bodies to create appropriate effects
- demonstrate the use of forces and constraints to control the behaviour of rigid bodies.

9.4 Introduction

In the previous chapter we discussed how objects in the world can be animated using keyframes. That is one approach to making things move in a graphical world.

Another approach is to move objects based on a mathematical simulation of how the world works. A particularly popular approach is to simulate the laws of physics to get realistic movement and interaction between objects. This has become a fairly standard feature of modern video games and other real time graphics systems. This popularity has been driven by the availability of high quality physics engines, software libraries for performing physics simulations. Engines are very important as programming high quality simulations is extremely difficult so engines make simulation available much more widely. For that reason, after a brief explanation of how an engine works, we will mostly talk about using physics engines, rather than writing simulations from scratch.

9.5 Simulating physics

A physics engine is a piece of software that simulates the laws of physics (at least within a certain domain). This means a mathematical simulation of the equations of physics. The most important equation is Newton's second law:

$$\mathbf{f} = m\mathbf{a} \tag{9.1}$$

To state it in words Force (**f**) equals mass (m) times acceleration (**a**). A force acts on an object to create a change of movement (acceleration). The acceleration depends both on the force and on the mass of the object. Force and acceleration are written in bold because they are both vectors having both a magnitude and a direction.

Re-arranging the equation we can see that the total acceleration of an object is the sum of all the forces acting on the object divided by its mass:

$$\mathbf{a} = \frac{1}{m}\sum_i \mathbf{f}_i \tag{9.2}$$

The acceleration is the rate of change of velocity (**v**), in mathematical terms:

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} \tag{9.3}$$

where velocity is the rate of change of position (**v**):

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{9.4}$$

From this equation we can see that the basic function of a physics engine is to evaluate all of the forces on an object, sum them to calculate the acceleration and then use the acceleration to update the velocity and position.

In terms of code a basic simulation would look something like this. The acceleration is calculated as in Code example 9.1.

The code assumes you have defined a class to represent a force.

The simplest way to update velocity and position is simply to add on the current acceleration, like this:

```
PVector acceleration = new PVector(0,0,0);
for (int i = 0; i < forces.length; i++)
{
    acceleration.add(forces[i].calculate());
}
acceleration.div(mass);

velocity.add(PVector.mult(acceleration, deltaTime));
position.add(PVector.mult(velocity, deltaTime));
```

Code example 9.1: Calculating acceleration.

```
velocity.add(PVector.mult(acceleration, deltaTime));
position.add(PVector.mult(velocity, deltaTime));
```

Where deltaTime is the time since the last frame.

Learning activity

Write a simple program to simulate a ball falling under gravity using the approach just described. Gravity is a constant force proportional to the object's mass (assuming we are on the earth's surface).

Is this a valid thing to do? The equations given above are continuous relationships where position and velocity are varying continuously over time. In the code shown above time is split into discrete frames and velocities and positions are updated only at those time steps. Will this introduce errors?

One way of looking at this is through the Taylor series. This states that a small change (δt) in a function can be represented as an infinite series like this:

$$y(t + \delta t) = y(t) + \delta t \frac{dy}{dt}(t) + \frac{(\delta t)^2}{2!}\frac{d^2y}{dt^2}(t) + \ldots \tag{9.5}$$

For small values of δt the later terms become smaller and smaller so the function can be approximated with the first few terms.

The code above implements the following equation for position (and the equivalent for velocity):

$$\mathbf{x}(t + \delta t) = \mathbf{x}(t) + \delta t \frac{d\mathbf{x}}{dt}(t) \tag{9.6}$$

Comparing these equations we can see that our code corresponds to the first two terms of the Taylor series. This means that it is a valid approximation, because for small (δt) the later terms of the Taylor series become smaller. However, it is just an approximation, and it is only a valid approximation for small values of δt. That means it can lead to noticeable errors if the rate at which we update the simulation is slow compared to our objects' velocities or accelerations. More accurate simulations can be created by including higher order derivatives of the function and through other sophisticated techniques. Implementing these techniques is beyond

the scope of this subject guide; however, they are built in to many readily available physics engines, which we will be using for the rest of this chapter of the subject guide to create interesting physics simulations.

9.6 Physics engines

A physics engine is a piece of software for simulating physics in an interactive 3D graphics environment. It will perform the simulation behind the scenes and make it easy to set up complex simulations.

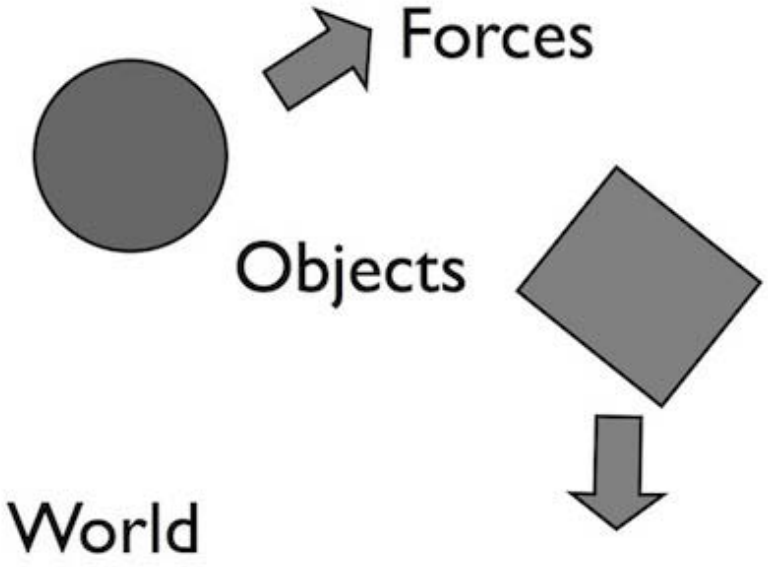


Figure 9.1: A physics simulation.

As shown in the image above a physics simulation consists of a **World** that contains a number of **Objects** which can have a number of **Forces** acting on them (including forces that are due to the interaction between objects). In addition there can be a number of **Constraints** that restrict the movement of objects. Each of these elements will be covered in the following sections.

Typically use of a physics engine consists primarily of setting up the elements of a simulation then letting it run, with maybe some elements of interaction.

There are many good physics engines available; we will use an engine for *Processing*, `called` BRigid which is based on the jBullet engine, which is itself based on the C++ engine Bullet.

Learning activity

Download a physics engine for *Processing*. At the time of writing, a good example is `BRigid`. The engine is likely to come with some documentation and examples. Run the example code and try to understand how it works using the documentation.

World

A Physics World is a structure that defines properties of the whole simulation. This typically includes the size of the volume to be simulated as well as other parameters such as gravity. Most physics engines require you to create a world before setting up any other element of the simulation, and to explicitly add objects to that world.

In BRigid, creating a world requires you to set the extents of the world; that is, the minimum and maximum values for x, y and z. These are used to create a BPhysics object which represents the world as shown in Code example 9.2.

```
//extents of physics world
Vector3f min = new Vector3f(-120, -250, -120);
Vector3f max = new Vector3f(120, 250, 120);
//create a rigid physics engine with a bounding box
physics = new BPhysics(min, max);
```

Code example 9.2: Creating a physics world.

The BPhysics object is used to simulate the world. In *Processing*'s `draw` function you must update the physics object so that the simulation runs:

```
physics.update();
```

Objects

The most visible elements of a simulation are the objects that move and interact with each other. There are a number of different types of objects:

Particles are the simplest type of object. They have a position, velocity and mass but zero size and no shape (at least from the point of view of the simulation). They can move, but they do not have a rotation. They are typically used for very small objects.

Rigid bodies are slightly more complex. They have a size and shape. They can move around and rotate but they cannot change their shape or deform in any way; they are rigid. This makes them relatively easy to simulate and means that they are the most commonly used type of object in most physics engines.

Compound bodies are objects that are made out of a number of rigid bodies linked together by joints or other methods. They are a way of creating objects with more complex movement while maintaining the simplicity of rigid body simulation. We will describe a number of ways of joining rigid bodies below.

Soft bodies and cloth are much more complex as they can change their shape as well as moving. Many modern physics engines are starting to include soft as well as rigid bodies, but they are out of the scope of this subject guide.

For the rest of the chapter we will use rigid bodies, and will also discuss ways of combining them into compound objects.

Collision shape

One of the most important properties of a rigid body is its shape. From the point of view of a physics engine the shape controls how it collides and interacts with other objects. An object typically has a different shape representation for physics than it has for graphics. This is because physics shapes need to be fairly simple so that the collision calculations can be done efficiently; while graphics shapes are typically much more complex so the object looks good.

A physics shape is designed to approximate the graphics shape while still remaining simple. There are a number of different types of physics shape that are typically used.

Simple primitives. Often objects are represented as simple primitive shapes for which simple collision equations are defined; for example, boxes or spheres. These are typically only rough approximations of the appearance of the object but are very efficient and are often close enough that the differences in an object's movement are not noticeable.

BRigid has a number of primitive collision shapes:

- Boxes (BBox)
- Spheres (BSphere)
- Planes (BPlane)

Compound shapes. If an object cannot be represented as a single primitive object, it may be possible to represent it as a number of primitive objects joined together: a compound shape.

In BRigid the BCompound class is used to create compound shapes.

Meshes. If the shape of an object is too complex to represent out of primitive objects it is possible to represent its physics shape as a polygon mesh, in the same way as a graphics object. Simulating meshes is much more expensive than simulating primitives, so the meshes must be simple. They are usually a different, and much lower resolution, mesh from the one used to render the graphics. They are typically created by starting with the graphics mesh and greatly reducing the number of polygons.

Creating a rigid body

In BRigid, creating a rigid body involves a number of steps. Firstly, you need to create a shape:

```
box = new BBox(this, 1, 50, 50, 50);
```

Then you need to assign a mass and initial position to the object. Positions are represented as Vector3f objects (a different representation of a vector from a *Processing* `PVector`):

```
float mass = 100;
Vector3f pos = new Vector3f(random(30), -150, random(1));
```

The shape, mass and position are used to create a BObject, which contains the rigid body:

```
BObject physicsShape = new BObject(this, mass, box, pos, true);
```

The BObject has a member variable `rigidBody` which represents the rigid body.

Finally, you add the body to the physics world so it will be simulated.

```
physics.addBody(physicsShape);
```

Learning activity

Use your chosen physics engine to write a program that creates a rigid body and simulates its behaviour.

Forces

In a physics simulation objects are affected by forces that change their movement in a number of ways. These can be forces that act on objects from the world (for example, gravity and drag); that act between objects (for example, collisions and friction); or that are triggered on objects from scripts (for example, impulses).

The following sections describe a number of different types of force.

Gravity

Figure 9.2: Gravity.

Gravity is a force that acts to pull objects towards the ground. It is proportional to the mass of an object. The mass term in gravitational force cancels out the mass term in Newton's second law of motion (Equation 9.1) so as to produce a constant downward acceleration. Gravity is typically a global parameter of the physics world.

In BRigid you set gravity on the physics world like this:

```
physics.world.setGravity(new Vector3f(0, 500, 0));
```

Note that gravity in physics engines typically just models gravity on the surface of the earth (or other planet) where the pull of the planet is the only significant gravitational force and gravity is constant. For simulations in outer space that include multiple planets and orbits you would need to implement your own gravitational force using a custom force script and Newton's law of gravity.

Learning activity

Use BRigid or another physics engine to create a simple simulation of a single rigid body falling under gravity.

Drag

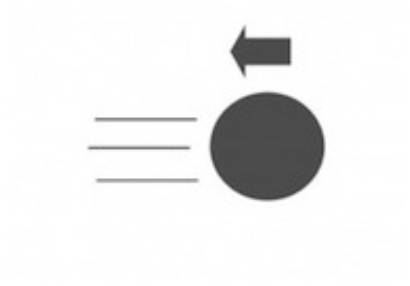


Figure 9.3: Drag.

Drag is a force that models air resistance, which slows down moving objects. It is proportional to the speed of an object and in the opposite direction so it will always act to reduce the speed.

In BRigid, drag is included in a general damping parameter which includes all damping forces; there is both a linear damping which reduces linear (positional) velocity and an angular damping which reduces angular (rotational) velocity. The damping applied to an object can be set using the `setDamping` method of a `RigidBody`:

```
body.rigidBody.setDamping(linearDamping, angularDamping);
```

Collision

When two objects collide they produce forces on each other to stop them penetrating each other and to separate them.

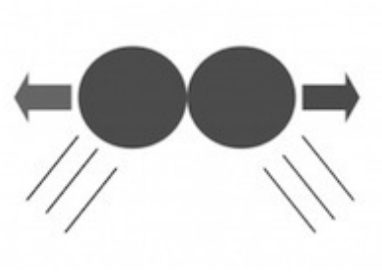


Figure 9.4: Collision.

In collision, momentum is conserved, so the sum of the velocities of the two objects stays the same. However, this does not tell us anything about what the two individual objects do. They might join together and move with a velocity that is the result of combining their momentum or they might bounce back from each other perfectly, without losing much velocity at all. What exactly happens depends on a number called the coefficient of restitution. If the restitution is 1, the objects will bounce back with the same speed and if it is 0 they will remain stuck together. The restitution is a combined property of the two objects. In most physics engines each object has its own restitution and they are combined to get the restitution of the collision. In BRigid you can set the restitution on a rigid body:

```
body.rigidBody.setRestitution(restitutionCoefficient);
```

A physics engine will typically handle all collisions without you needing to write any code for them. However, it is often useful to be able to tell when a collision has happened; for example, to increase a score when a ball hits a goal or to take damage when a weapon hits an enemy. Code example 9.3 gives an example of how to detect collisions in BRigid.

Friction

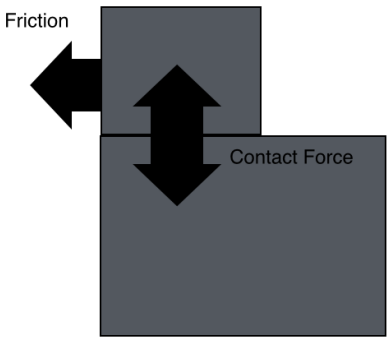


Figure 9.5: Friction.

```
//check if bodies are intersecting
int numManifolds = physics.world.getDispatcher().getNumManifolds();

for (int i = 0; i < numManifolds; i++) {
    PersistentManifold contactManifold
         = physics.world.getDispatcher().getManifoldByIndexInternal(i);
    int numCon = contactManifold.getNumContacts();
    //change and use this number
    if (numCon > 0) {
      RigidBody rA = (RigidBody) contactManifold.getBody0();
      RigidBody rB = (RigidBody) contactManifold.getBody1();
      if(rA == droid.physicsObject.rigidBody)
      {
        for (int j = 0; j < crates.length; j++)
        {
          if(rB == crates[j].physicsObject.rigidBody)
          {
            score+= 1;
          }
        }
      }
      if(rB == droid.physicsObject.rigidBody)
      {
        for (int j = 0; j < crates.length; j++)
        {
          if(rA == crates[j].physicsObject.rigidBody)
          {
            score+= 1;
          }
        }
      }
    }
}
```

Code example 9.3: Responding to a collision.

Friction is a force that acts on two bodies that are in contact with each other. Friction depends on the surface texture of the objects involved. Slippery surfaces like ice have very low friction, while rough surfaces like sandpaper have very high friction. The differences between these surfaces is represented by a number called the coefficient of friction. Friction also depends on the contact force between two objects; that is, the force that is keeping them together. For one object lying on top of another this contact force would be the gravity acting on the top object as shown in Figure 9.5; this is why heavy objects have more friction than light objects.

There are two types of friction: static and dynamic friction.

Static friction

This happens when two objects are in contact with each other but not moving relative to each other. It acts to stop objects starting to move across each other and is equal and opposite to the forces parallel to the plane of contact of the objects. It has

a maximum value that is proportional to the contact reaction force of the objects and a coefficient of friction.

Dynamic friction

This happens when two objects are in contact and are already moving relative to each other. It acts against the relative velocity of the objects and is in the opposite direction to that velocity, so it tends to slow the objects down (like drag). It is proportional to the velocity, the contact reaction force and a coefficient of friction.

The coefficients of friction are different in the two cases. Both depend on the two materials involved in a complex way. Most physics engines do not deal with these complexities; each object has only a coefficient of friction. The coefficients of the two objects are multiplied to get the coefficient of their interaction. Some physics engines have separate parameters for static and dynamic friction coefficients, but BRigid has only one, which can be set using the `setFriction` method:

```
body.rigidBody.setFriction(frictionCoefficient);
```

Impulse

The forces listed above are all typically included as standard in a physics engine, but sometimes you will need to apply a force that is not included. Most physics engines allow you to apply a force directly to an object through code. This will take the form either of a constant force that acts over time (such as gravity); or of an impulse which is a force that happens at one instant of time and then stops (such as a collision).

In the following example code we are applying an impulse to simulate a catapult. The player can drag the object about with the mouse and when the mouse is released, an impulse is applied to it that is proportional to the distance to the catapult. The impulse is calculated as the vector from the object to the catapult and is then scaled by a factor `forceScale`. The result is applied to the rigid body using the `applyCentralImpulse` command (there is also a command `applyImpulse` which can apply an impulse away from the centre of the object).

```
void mouseReleased()
{
  PVector impulse = new PVector();
  impulse.set(startPoint);
  impulse.sub(droid.getPosition());
  impulse.mult(forceScale);
  droid.physicsObject.rigidBody.applyCentralImpulse(
        new Vector3f(impulse.x, impulse.y, impulse.z));
}
```

Code example 9.4: Applying an impulse.

Learning activity

Based on Code example 9.4 implement a simple catapult game in which a rigid body is launched with an impulse and then can collide with other rigid bodies to knock them over.

Joints and constraints

Constraints are a way of limiting how objects can move. These can be absolute restrictions on the movement of an object (for example, the object can only move along the y-axis), or they can restrict the movement of an object relative to another object (for example, creating a hinge between two objects). The latter type of constraint are often called joints. Joints are ways of attaching objects to each other in a more flexible way than simply clamping them together into a single object.

Rigid bodies have six Degrees Of Freedom (DOF) of movement (in this context a DOF is a dimension in which an object can move). Objects have three positional DOFs (x,y,z) and three rotational DOFs (rotation about the x,y,z axes). Constraints typically limit the number of degrees of freedom of an object while joints restrict the DOFs relative to another object.

Learning activity

Implement a chain by using a number of rigid bodies connected by hinge joints.

9.7 Summary

This chapter has introduced the basic concept involved in a physics simulation for animation and interactive graphics. A simulation is composed of a number of objects (normally rigid bodies) acting under the effect of forces and interacting with each other. Writing a fully functional physics simulation from scratch is highly challenging and beyond the scope of this course, but the topics covered in this section should be enough to effectively use a physics engine to create a moderately complex interactive simulation.

9.8 Sample examination questions

1. Define the term *rigid body*. How are rigid bodies used in physics simulations?
2. You have been given a complex graphics model of a space ship for use in a physics based game. Explain how you would go about creating a physics shape for this model and how the shape should relate to the graphics model.
3. How would you create appropriate physical properties for the following objects:
   - a highly polished ball bearing
   - a patch of damp grass
   - a rubber ball?

Chapter 10

Display

10.1 Essential reading

Shirley, P. et al. *Fundamentals of Computer Graphics.* (2009), Chapters 21 and 22.

10.2 Recommended reading

Glassner, A. *Principles of Digital Image Synthesis.* (1994), Chapter 20.

Reinhard, E., G. Ward, S. Pattanaik, and P. Debevec *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics)*. (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005) second edition [ISBN 9780123749147].

10.3 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- demonstrate awareness of visual perception as it relates to computer generated imagery
- describe and explain methods of tone reproduction
- explain the processes involved in high dynamic range imaging.

10.4 Introduction

By now you have learnt the theory and the practice behind generating computer graphics. We have covered the intrinsic techniques for creating 3D images and animations. In this chapter, it is time to explore what happens to the graphics we have created when we display them on a real device, and they are perceived by a human observer.

10.5 Visual perception: an overview

Perception is the process that enables humans to make sense of the stimuli that surround them. Visual perception deals with the information that reaches the brain through the eyes. It links the physical environment with the physiological and

psychological properties of the brain, transforming sensory input into meaningful information.

In recent years, visual perception has increased in importance in computer graphics, predominantly due to the demand for realistic computer generated images. The goal of perceptually-based rendering is to produce imagery that evokes the same responses as an observer would have when viewing a real-world equivalent. To this end, work has been carried out on exploiting the behaviour of the human visual system (HVS). For this information to be measured quantitatively, a branch of perception known as **psychophysics** is employed, where quantitative relations between physical stimuli and psychological events can be established.

Psychophysical experiments are a way of measuring psychological responses in a quantitative way so that they correspond to actual physical values. It is a branch of experimental psychology that examines the relationship between the physical world and peoples' reactions and experience of that world. Psychophysical experiments can be used to determine responses such as sensitivity to a stimulus. In the field of computer graphics, this information can then be used to design systems that are finely attuned to the perceptual attributes of the visual system.

To make an assessment of the effects of reflected ambient light on the perception of electronically displayed images, it is necessary to understand several perceptual phenomena that may play a part in the process.

The human visual system

The human visual system receives and processes electromagnetic energy in the form of light waves reaching the eye. This starts with the path of light through the **pupil** (Figure 10.1), which changes in size to control the amount of light reaching the back of the eye. Light then passes through the **lens**, which provides focusing adjustments, before reaching the photoreceptors in the **retina** at the back of the eye. These receptors in the retina consist of about 120 million **rods** and 8 million **cones**. Rods are highly sensitive to light and provide low intensity vision in low light levels, but they cannot detect colour. They are located primarily in the periphery of the visual field. In contrast to this, high-acuity colour vision is provided through three types of cones: **L**, which are sensitive to long wavelengths; **M**, which are sensitive to medium wavelengths; and **S**, which are short wavelength sensitive. Finally, the **photopigments** in the rods and cones transform this light into electrical impulses that are passed to neuronal cells and transmitted to the brain via the **optic nerve**.

Lightness and colour constancy

The ability to judge a surface's reflectance properties despite any changes in illumination is known as **colour constancy**. **Lightness constancy** is the term used to describe the phenomena whereby a surface appears to look the same regardless of any differences in the illumination. For example, white paper with black text maintains its appearance when viewed indoors in a dark environment or outdoors in bright sunlight, even if the black ink on a page viewed outdoors actually reflects more light than the white paper viewed indoors. **Chromatic colour constancy** extends this to colour: a plant seems as green when it is outside in the sun as it does if it is taken indoors under artificial light.

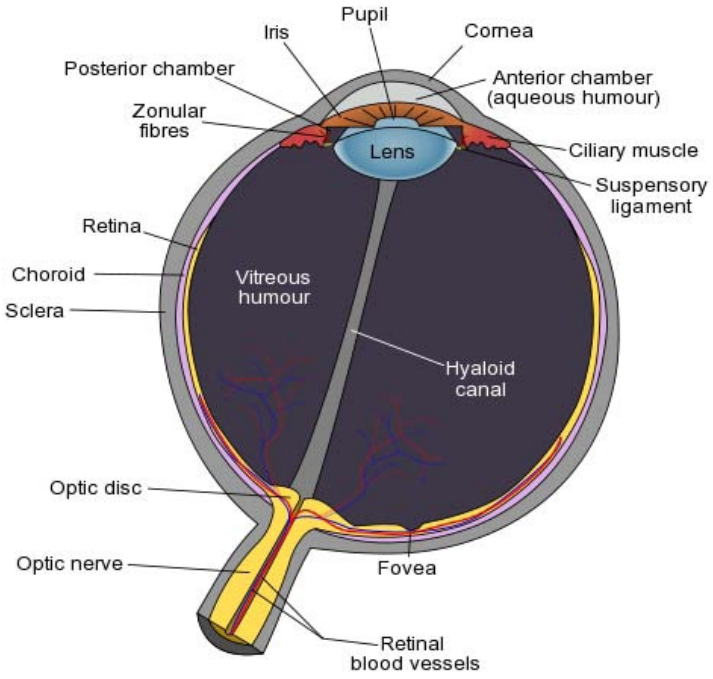


Figure 10.1: A schematic section through the human eye.
Source: By Silversmith; reproduced under Creative Commons licence CC BY-SA 3.0 from:
`http://en.wikipedia/wiki/File:`
`Schematic_diagram_of_the_human_eye_with_English_annotations.svg`

A number of theories have been put forward regarding constancy. Early explanations involved adaptational theories, suggesting that the visual system adjusts in sensitivity to accommodate changes. However, this would require a longer time than is needed for lightness constancy to occur, and adaptational mechanisms cannot account for shadow effects. Other proposed theories include unconscious inference (where the visual system 'knows' the relationship between reflectance and illumination and discounts it); relational theories (where perceived lightness depends upon the relative luminance – the contrast – between neighbouring regions); and anchoring (where the region with the highest luminance is regarded as being white and all other regions are scaled relative to it).

10.6 High dynamic range imaging

The ultimate aim of realistic graphics is the creation of images that provoke the same responses that a viewer would have to a real scene. This requires significant effort to achieve, and one of the key properties of this problem is that the overall performance of a photorealistic rendering system is only as good as its worst component. In the field of computer graphics, the actual image synthesis algorithms – from scanline techniques to global illumination methods – are constantly being reviewed and improved, but weaknesses in a system in both areas can make any improvements in the underlying rendering algorithm insignificant. Consequently, good care has to be

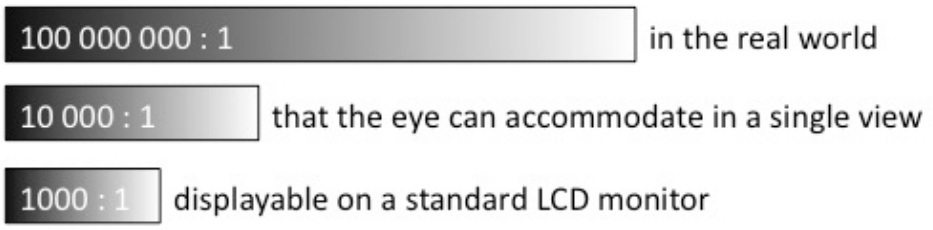


Figure 10.2: A comparative view of dynamic range.

taken when designing a system for image synthesis to strike a good balance between the capabilities of the various stages in the rendering pipeline.

While research into ways of rendering images provides us with better and faster methods, we do not necessarily see their full effect due to limitations of the display hardware. To ensure that the scene as it was created closely resembles the scene as it is displayed, it is necessary to be aware of any factors that might adversely influence the display medium.

One major problem is that computer screens are limited in the range of luminance they can display. Most are not yet capable of producing anywhere near the range of light in the real world. This means the realistic images we have carefully created are not being properly displayed.

One goal of realistic computer graphics is such that if a virtual scene is viewed under the same conditions as a corresponding real-world scene, the two images should have the same luminance levels, or **tones**. However, due to the limitations of current technology, this is rarely the case. The ratio between the darkest and the lightest values in a scene is known as the **dynamic range**. The dynamic range in the real world is far greater than the range that can be produced on most electronic displays. Luminous intensity – the power of a light source – is measured in candelas per square metre (cd/m^2). The human visual system can accommodate a wide range of luminance in a single view, around 10 000:1 cd/m^2, and our eye adapts to our surroundings, changing what we see over time.

An image displayed on a standard LCD screen is greatly restricted in terms of tonality, perhaps achieving at most 1000:1 cd/m^2. It is therefore necessary that the image be altered in some way, usually through some form of scaling, to fit a display device that is only capable of outputting a low dynamic range.

Tone mapping

To ensure a true representation of tonal values, some form of scaling or mapping is required to convey the range of a light in a real-world scene on a display with limited capabilities.

Tone mapping (also known as **tone reproduction**) provides a method of scaling (or mapping) luminance values in the real world to a displayable range. Although it is tempting to use a straightforward linear scaling, this is not an adequate solution as many details can be lost (Figure 10.3). The mapping must be tailored in some

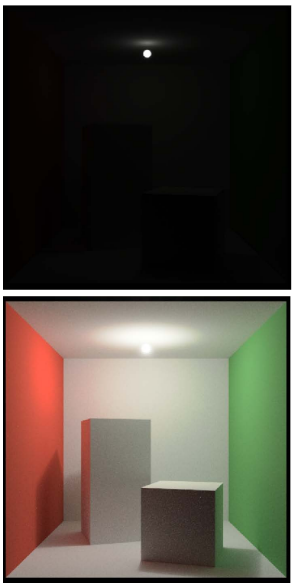


Figure 10.3: Linear scaling of HDR data will cause almost all details to be lost, as the top image shows. Here, the light bulb is mapped to a few white pixels and the remainder of the image is black. Tone reproduction operators attempt to solve this issue; in this case recovering detail in both light and dark areas as well as all areas in between (bottom image).

non-linear way. Instead, the mapping must be specifically tailored in a non-linear manner, permitting the luminance to be compressed in an appropriate way. Algorithmic solutions, known as tone mapping operators, or tone reproduction operators, have been devised to compress certain features of an image and produce a result with a reduced dynamic range that appears plausible or appealing on a computer monitor.

Some tone mapping operators focus on preserving aspects such as detail or brightness, some concentrate on producing a subjectively pleasing image, while others focus on providing a perceptually-accurate representation of the real-world equivalent. In addition to compressing the range of luminance, it can be used to mimic perceptual qualities, resulting in an image which provokes the same responses as someone would have when viewing the scene in the real world. For example, a tone reproduction operator may try to preserve aspects of an image such as contrast, brightness or fine detail – aspects that might be lost through compression.

Learning activity

Compare the different types of tone mapping operators on Bernhard Vogl's website: `dativ.at/logmap/index.html#overview` (accessed May 2014).

Two types of tone reproduction operators can be used: **spatially uniform** – also known as **single-scale** or **global**, and **spatially varying** – also known as **multi-scale** or **local**. Global operators apply the same transformation to every pixel. It may depend upon the contents of the image as a whole, but the same transformation is applied to every pixel. All pixels of the image are processed in a similar way,

regardless of whether they are located in a bright or dark area. This often results in a tone mapped image that looks 'flat', having lost its local details. Conversely, local operators apply a different scale to different parts of an image. Local tone mapping operators consider pixel neighbourhood information for each individual pixel, which simulates the adaptive and local properties of the human vision system. This can lead to better results but this will take longer for the computer to process.

At present, it is a matter of choosing the best tool for the job, although the development of high dynamic range displays means operators can be compared more accurately. Where required, for the purposes of realistic image generation, perceptually-accurate operators are the best choice.

Related effects

Replication of visual effects that are related to the area of tone reproduction include the modelling of glare. Psychophysically-based algorithms have been produced that will add glare to digital images, simulating the flare and bloom seen around very bright objects. Psychophysical tests have demonstrated that these effects increase the apparent brightness of a light source in an image. While highly effective, glare simulation is computationally expensive.

HDR capture and storage

Current state-of-the-art image capturing techniques allow much of the luminance values to be recorded in **high dynamic range (HDR)** images. This is desirable, because high dynamic range display devices are being developed that will allow this data to be displayed directly. By capturing and storing as much of the real scene as possible, and only reducing the data to a displayable form just before display, the image becomes future-proof. HDR images store a depiction of the scene in a range of intensities commensurate with the real-world scene. These images may be rendered images or photographs.

Digital photographs are often encoded in a camera's raw image format, because 8 bit JPEG encoding does not offer enough values to allow fine transitions (and introduces undesirable effects due to the lossy compression). HDR images do not use integer values to represent the single colour channels (for example, $[0 \ldots 255]$ in an 8 bit per pixel interval for R, G and B) but instead use a floating point representation. Three of the most common file formats are as follows:

Format	Extension	Bits per Pixel	Advantages/Disadvantages
Radiance RGBE	.hdr	32	Superlative dynamic range; sacrifices some colour precision but results in smaller file size.
OpenEXR	.exr	48	High colour precision at the expense of some dynamic range; can be compressed.
Floating point TIFF/PSD	.tiff .psd	96	Very accurate with large dynamic range but results in huge file sizes and wasted internal data space.

Non-HDR cameras take pictures at one exposure level with a limited contrast range.

Figure 10.4: The top images show the extent of the dynamic range. The bottom image is tonemapped for display on a computer monitor. (Rendering of Kalabsha temple courtesy of Veronica Sundstedt, ©2003.)

This results in the loss of detail in bright or dark areas of a picture, depending on whether the camera had a low or high exposure setting. HDR compensates for this loss of detail by taking multiple pictures at different exposure levels and intelligently stitching them together to produce a picture that is representative in both dark and bright areas. Figure 10.4 demonstrates how varying levels of exposure reveal different details. By combining the various exposure levels and tone mapping them, a better overall image can be achieved.

Learning activity

Digital photography has meant that anyone can try their hand at creating HDR images. You can see examples at `www.flickr.com/groups/hdr/pool/` (available May 2014).

10.7 Colour reproduction

While the previous section deals with the range of image intensities that can be displayed, devices are also limited in the range of colours that may be shown. Tone mapping compresses luminance values rather than colour values.

Gamut mapping

The term **gamut** is used to indicate the range of colours that the human visual system can detect, or that display devices can reproduce.

Even with 24-bit colour, although indicated as 'millions of colours' or 'true colour',

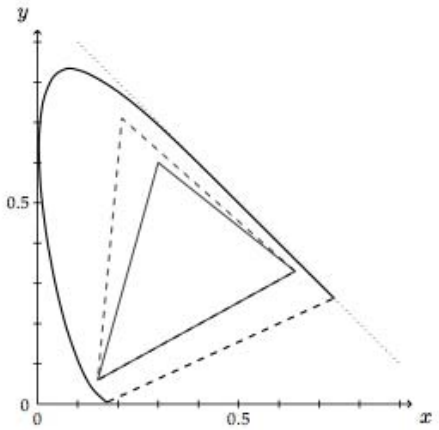


Figure 10.5: A representation of sRGB (solid triangle) and Adobe RGB (dashed triangle) colour space gamuts, relative to the CIE standard observer chromaticity gamut.
Source: Reproduced from the subject guide for CO2227 Creative Computing II: interactive multimedia, Volume 2, by Christophe Rhodes and Sarah Rauchas.

there are many colours within the visible spectrum that screens cannot reproduce. To show the extent of this limitation for particular display devices, chromaticity diagrams are often used. Here, the Yxy colour space is used, where Y is a luminance channel (which ranges from black to white via all greys), and x and y are two chromatic channels representing all colours. Figure 10.5 shows a chromaticity diagram indicating the gamut of colours visible to humans, and two restricted gamuts, one for a typical screen and one for a printing device. Given that the triangle indicating screen capability is completely contained with the shape of all visible colours, there are many visible colours that cannot be reproduced on a screen.

Assuming that some of the colours to be displayed in an image are outside a screen's gamut, the image's colours may be remapped to bring all its colours within displayable range. This process is referred to as gamut mapping. A simple mapping would only map out-of-range colours directly inward towards the screen triangular gamut. Such a 'colorimetric' correction produces visible artefacts. A better solution is to re-map the whole gamut of an image to the screen's gamut, thus remapping all colours in an image. This 'perceptual' or 'photometric' correction may avoid the above artefacts, but conversely there are many different ways in which such remapping may be accomplished. As such, there is no standard way to map one gamut into another more constrained gamut.

More detailed information on colour spaces and profiles can be found in Chapter 1 of the subject guide for **Creative computing II: interactive multimedia** (Vol 2).

10.8 Summary

The study of human perception is inseparable from the presentation of images in computer graphics. Both at low-level visual responses and higher cognitive processing, the attributes of the human visual system play an important role in how we perceive all that we are presented with on-screen. Research into this is still at an exploratory stage – there is a wealth of information still to be investigated, and much to learn from related fields, such as vision, psychology and neuroscience. With

every advance in computer graphics techniques new challenges are thrown open, and we find ourselves looking beyond the traditional boundaries of the field.

10.9 Exercises

Try out one of the many online HDR tutorials, such as: `digital-photography-school.com/how-to-hdr-photography` (available May 2014.)

10.10 Sample examination questions

1. Give an overview of the concept of tone mapping.
2. Describe some of the characteristic effects that can appear in tone-mapped images. Why do these occur?

Sample examination paper

Duration: 2 hours 15 minutes.

There are FIVE questions in this paper. Each question is worth 25 marks. Candidates should answer THREE questions. All questions carry equal marks and full marks can be obtained for complete answers to THREE questions.

Question 1: Transformations, projections and rasterisation

1. All modern computers have a graphics pipeline – the sequence of steps used to create a 2D representation of a 3D scene. Draw a diagram of this and explain each stage in the pipeline. (10 marks)
2. In a typical graphics program we may need to deal with a number of different coordinate systems. Describe three of these. (15 marks)

> **Tip:** It is always fine to include a diagram in an examination answer. Where you are explicitly asked for a diagram, make sure you have clearly labelled and annotated it.

Question 2: Surfaces and shading

1. What is 'hidden surface removal' and why is it necessary? (3 marks)
2. Describe two methods of achieving hidden surface removal. (10 marks)
3. Gouraud shading computes an intensity for each vertex and then interpolates the computed intensities across the polygons. Briefly describe the steps involved in this. (4 marks)
4. Why does Gouraud shading not handle specular highlights very well? Use diagrams to illustrate your answer where appropriate. (8 marks)

> **Tip:** The Examiners want you to demonstrate understanding of the algorithms involved. An answer in the form of pseudocode is acceptable, but make sure that this is also explained in more detail.

Question 3: Textures

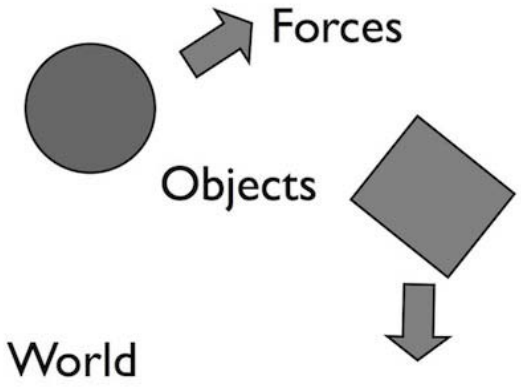
1. Briefly explain the following terms:
   (a) texel (2 marks)
   (b) texture coordinates (2 marks)
   (c) bounding box (2 marks)
   (d) procedural textures (2 marks)
   (e) height field. (2 marks)

2. Generally speaking, there are two types of procedural texture. What are they and why might they be used together? (4 marks)
3. Describe the concept of environment (reflection) mapping. (5 marks)
4. Describe the process of environment mapping. (6 marks)

Tip: Where there are two marks available for (part of) a question, a couple of succinct sentences will suffice. Where more marks are available, a longer answer is expected and you should go into much more detail, perhaps also giving examples and drawing diagrams.

Question 4: Representing the real world

1. Describe THREE ways lighting can be sent into a scene, giving an example of each. (6 marks)
2. Describe the difference between local and global illumination. Explain the advantages and disadvantages of each. (8 marks)
3. What is the bidirectional reflectance distribution function (BRDF)? Explain, using a diagram, and state why BRDFs are important in computer graphics. (4 marks)
4. A physics engine is a piece of software for simulating physics in an interactive 3D graphics environment. The image below shows the components of a physics simulation.



(a) Describe the three components shown in the diagram above. (3 marks)

(b) Describe two types of force that can be simulated in a physics simulation. (4 marks)

Tip: For a question looking for advantages and disadvantages, first explain what the technique is, then give the advantages and disadvantages, saying why these are good/bad. If you have, for example, two advantages, try also to supply two disadvantages.

Question 5: Post-processing and display

1. Give three examples of applications that use electronic display devices and which require images to be faithful to the original. (3 marks)
2. Explain what is meant by the dynamic range of an image and of a display. Give an example. (6 marks)
3. What is tone mapping? Why is it used? (6 marks)
4. Why is a linear scaling not appropriate for accurate tone reproduction? (6 marks)
5. How might the viewing conditions affect our perception of an image? Explain the principles behind this. (4 marks)

Tip: Always check the marks available for a question and divide your time appropriately. Although you are expected to answer all parts of a question in sequence, the questions themselves do not have to be answered in order. For example, you could answer Question 4, then Question 1, then Question 2.

Bibliography

Angel, E. and D. Shreiner *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL.* (USA: Addison-Wesley Publishing Company, 2011) sixth edition [ISBN 9780273752264(pbk); 9780132545235].

Bailey, M. and A. Glassner 'Introduction to computer graphics.' (New York, NY, USA: ACM, 2004) ACM SIGGRAPH 2004 Course Notes.

Carlson, W. 'A critical history of computer graphics and animation;' `http://design.osu.edu/carlson/history/lessons.html`, (accessed May 2014), 2003.

Dunn, F. and I. Parberry *3D Math Primer for Graphics and Game Development.* An A.K. Peters book (Taylor & Francis, 2011) second edition [ISBN 9781568817231].

Durand, F. 'A short introduction to computer graphics: MIT laboratory for computer science' ; `people.csail.mit.edu/fredo/Depiction/1_Introduction/reviewGraphics.pdf`, (accessed May 2014), publication date unknown.

Glassner, A. *Principles of Digital Image Synthesis.* (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994) [ISBN 9781558602762].

Hughes, J., A. van Dam, M. McGuire, D. Sklar, J. Foley, S. Feiner, and K. Akeley *Computer graphics: principles and practice.* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2013) third edition [ISBN 9780321399526].

Millington, I. *Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game.* (Elsevier; Morgan Kaufmann, 2010) [ISBN 9780123819765].

Phong, B. T. 'Illumination for computer generated pictures.' *Commun. ACM* 18(6), 1975, pp. 311–317 [0001-0782].

Rath, E. 'Coordinate systems in OpenGL' ; `www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl` (accessed May 2014), 2014.

Reinhard, E., G. Ward, S. Pattanaik, and P. Debevec *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics).* (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005) second edition [ISBN 9780123749147].

Rost, R., B. Licea-Kane, D. Ginsburg, J. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen *OpenGL Shading Language.* (Addison-Wesley Professional., 2009) [ISBN 9780321637635].

Shiffman, D. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (The Morgan Kaufmann Series in Interactive 3D Technology).* (Elsevier Science, 2009) [ISBN 9780080920061].

Shiffman, D. *The Nature of Code.* (USA, 2012) [ISBN 9780985930806].

Shirley, P. and S. Marschner *Fundamentals of Computer Graphics.* (Natick, MA, USA: A.K. Peters, Ltd, 2009) third edition [ISBN 9781568814698].

Watt, A. *3D Computer Graphics with Cdrom.* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999) third edition [ISBN 9780201398557].

Notes

Notes

Notes

Comment form

We welcome any comments you may have on the materials which are sent to you as part of your study pack. Such feedback from students helps us in our effort to improve the materials produced for the University of London.
If you have any comments about this guide, either general or specific (including corrections, non-availability of Essential readings, etc.), please take the time to complete and return this form.

Title of this subject guide: .....

.....

Name .....

Address .....

.....

Email .....

Student number .....

For which qualification are you studying? .....

Comments

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Please continue on additional sheets if necessary.

Date: .....

Please send your completed form (or a photocopy of it) to:
Publishing Manager, Publications Office, University of London, Stewart House, 32 Russell Square, London WC1B 5DN, UK.