NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE CODE :CIT 371

COURSE TITLE:

INTRODUCTION TO COMPUTER GRAPHICS AND ANIMATION

COURSE GUIDE

CIT 371
INTRODUCTION TO COMPUTER GRAPHICS AND ANIMATION

Course Team	Mr. F. E. Ekpenyong (Writer) – NDA
Course Editor	
Programme Leader	
Course Coordinator	

NATIONAL OPEN UNIVERSITY OF NIGERIA
National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
No. 5 Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

Printed 2009

ISBN:

CONTENTS **PAGE**

Introduction

Computer graphics is concerned with producing images and animations (or sequences of images) using a computer. This includes the hardware and software systems used to make these images. The task of producing photo-realistic images is an extremely complex one, but this is a field that is in great demand because of the nearly limitless variety of applications. The field of computer graphics has grown enormously over the past 10–20 years, and many software systems have been developed for generating computer graphics of various sorts. This can include systems for producing 3-dimensional models of the scene to be drawn, the rendering software for drawing the images, and the associated user-interface software and hardware.

Our focus in this course will *not* be on how to use these systems to produce these images, but rather in understanding how these systems are constructed, and the underlying mathematics, physics, algorithms, and data structures needed in the construction of these systems.

The field of computer graphics dates back to the early 1960's with Ivan Sutherland, one of the pioneers of the field. This began with the development of the (by current standards) very simple software for performing the necessary mathematical transformations to produce simple line-drawings of 2- and 3-dimensional scenes.

As time went on, and the capacity and speed of computer technology improved, successively greater degrees of realism were achievable. Today it is possible to produce images that are practically indistinguishable from photographic images

What you will Learn in this Course

The course consists of units and course guide. This course guide tells you briefly what the course is all about, what course materials you will be using and how you can work with these materials. In addition, it advocates some general guidelines for the amount of time you are likely to spend on each unit of the course in order to complete it successfully.
It gives you guidance in respect of your tutor Marked assignment which will be made available in the assignment file. There will be regular tutorial classes that are related to the course. It is advisable for you to

attend these tutorial sessions. The course will prepare you for the challenges you will meet in the field of Computer graphics.

Course Aims

The aim of this course is to provide an introduction to the theory and practice of computer graphics. By introducing topics that deal with computer graphics rendering of primitive objects, polygon clipping algorithms, two-dimensional transformations, three-dimensional transformations, viewing camera rendering and projections, object representations, three-dimensional curve and surface rendering algorithms, and line and surface removal algorithms.

Objectives

In order to achieve the laid down goals, the course has a set of objectives. Each unit is designed with specific objectives at the beginning. The students are advised to read these objectives very carefully before embarking on the study unit. You may also wish to refer to them during your study in order to measure your progress. You are also advised to look at the unit objectives after completion of each unit. By so doing, you would have followed the instruction of the unit.
Below are the comprehensive listings of the overall objective s of the course. By meeting these objectives, the said aims of the course must have been achieved.

Thus the after going through this course you should be able to:

- Explain the overall workflow and techniques involved in computer animation production.
- To understand the fundamental computer graphics topics including graphics pipeline architecture, transformations, modeling, viewing, shading, and texture mapping.
- To study basic mathematical backgrounds related to computer graphics including linear algebra and geometry.
- Understand vividly, those computer graphic algorithms (such as object transformation, geometric representation, shading and illumination model, anti-aliasing and Ray tracing).
- Study 3-D curve and surface algorithms, in rendering, surface and line removal algorithms
- Explain the abstract mathematical model describing the way colors can be represented

- Understand the methods of Computing a digital image of what the virtual camera sees

Working through this Course

To complete this course, you are required to read each study units, read the textbooks and other materials which may be provided by the National Open University of Nigeria. Each unit contains self-assessment and at certain points in the course you would be required to submit assignment for assessment purposes. At the end of the course there is a final examination. The course should take you about 16 weeks to complete. Below you will find listed all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course in time and successfully.
This course entails you spend a lot of time to read. I would advise that you avail yourself the opportunity of attending the tutorial sessions where you have the opportunity of comparing your knowledge with that of others.

Course Materials

The main components of the course are:

1. The course guide
2. Study Units
3. References/Further Readings
4. Assignments
5. Presentation Schedule

Study Units

The study Units of this course are:

Module 1	**Definition and Concepts of Computer Graphics**
Unit1	Computer Graphics and Applications
Unit2	Hardware, Software and Display Devices
Unit3	Graphics Data Structure
Unit4	Colour Theory
Unit5	Image Representation
Module 2	**Geometric Modelling**
Unit1	Basic Line drawing
Unit 2	Mathematics of CG
Unit 3	Curve and Surface Design

Description

The course is made up of three modules viz; module one Definition and Concepts of computer graphics, module two modeling, module three 3D Graphics Rendering.

Each module contains five units as follows:

Unit one focuses on the definition, history and application areas of computer graphics. The second unit deals with basic graphics i/o devices and display devices. Graphics data structures is detailed in unit three while unit four treats the different colour representation in the graphics system. How digital images are represented in a computer is discusses in unit five. This 'mini'-topic explores different forms of frame-buffer for storing images, and also different ways of representing colour and key issues that arise in colour.

Geometric Transformation How to use linear algebra, e.g. matrix transformations, to manipulate points in space Geometric Modelling is treated in module two it also explores how points can be "joined up" to form curves and surfaces, unit three and Ray tracing techniques and texture Mapping units four and five of the second module.

The third module discuses manipulating/positioning of points in 3D space, this module treats the modeling of objects and their trajectories 3D transformation principles and Animation is seen in the first unit of this module, Scan conversion or Rasterisation, 3D viewing animation techniques and Hidden surface elimination are explained in units three, four and five of module three respectively.

Each unit consist of about two or three weeks work and include an introduction, objectives, reading materials, exercises and conclusion, summary and Tutor marked Assignments(TMAs), references and other resources. The units directs you to work on exercises related to the requires readings. In general, these exercises test you on the materials you have just covered or require you to apply it in some way and thereby assist you to evaluate your progress and to reinforce your comprehension of the material. Together with TMAs, these exercises will help you in achieving the stated learning objectives of the individual units and of the course as a whole.

Presentation Schedule

Your course materials have important dates for the early and timely submission of your TMAs and attending tutorials. You should remember that you are required to submit all your assignments by the stipulated time and date. You should also guard against falling behind your work.

Assessment

There are three aspects of assessment of the course. First is made up of self assessment exercise, second consists of tutor marked assignments and the third is the written examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you gathered during the course. The assignments must be submitted to your facilitator for formal assessment in accordance with the deadlines stated in the presentation schedule and the assignment file. The work you submit to your tutor for assessment will count for 30% of your total course work. At the end of the course, you will need to sit for a final or end of course examination of about three hours duration. This examination will count for 70% of your total course mark.

Tutor-Marked Assignment

The Tutor Marked Assignment (TMA) is a continuous assessment component of your course. It accounts for 30% of the total score. You will be given four (4) TMAs to answer. Three of these must be answered before you are allowed to sit for the end of year examination. The TMA would be given to you by your facilitator and returned after you have done the assignment. Assignment questions for the units in this course are contained in the assignment file. You will be able to complete your assignment from the information and materials contained in your reading, reference and study units. However, it is desirable in all degree level of education to demonstrate that you have read and researched more into your references, which will give you a wider view point and may provide you with a deeper understanding of the subject.
Make sure that each assignment reaches your facilitator on or before the deadline given in the presentation schedule and assignment file. If for

any reason you cannot complete your work on time, contact your facilitator before the assignment is due to discuss the possibility of an extension. Extension will not be granted after the due date unless there are exceptional circumstances.

Final Examination and Grading

The end of course examination for Introduction to Computer Graphics and Animation will be about 3 hours and it has a value of 70% of the total course work. The examination will consist of questions which will reflect the type of self testing, practice exercise and tutor marked assignment problems you have previously encountered. All areas of the course will be assessed.

Use the time between finishing the last unit and sitting for the examination to revise the whole course. You might find it useful to review your self-test, TMAs and comments on them before the examination. The on course examination covers all parts of the course.

Course Marking Scheme

Assignment	Marks
Assignment1-4	Four assignments, best three marks of the four count at 10% each-30% of course marks
End of course examination	70% of overall course marks
Total	100% of course materials

Facilitators/Tutors and Tutorials

There are 15 hours of tutorials provided for in support this course. You will be notified of the dates, times and location of these tutorials as well as the name and phone number of your facilitator as soon as you are allocated a tutorial group.

Your facilitator will mark and comment on you assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail your Tutor Marked Assignment to your facilitator before the schedule date. (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not delay to contact your facilitator by telephone or e-mail if you need assistance.

The following might be circumstances which you find assistance necessary, hence you would have to contact your facilitator if:

- You do not understand any part of the study or assigned readings.
- You have difficulty with the self-tests.
- You have a question or problem with an assignment or with the grading of an assignment.

You should endeavour to attend the tutorials. This is the only chance to have face to face contact with your course facilitator and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study.

Summary

This course provides intermediate background in computer graphics for graduate and undergraduate students.

To give students a solid understanding of the principles of modeling, viewing, and rendering operations in 3-dimensional computer graphics systems. Special attention is given to the variety of methods for modeling 3D virtual worlds, and to advanced techniques for realism in rendering images of such models. Included are detailed treatments of polygonally-based modeling methods, plus substantial investigation of the basic methods of parametric surface representation.

Details of 3D viewing methods are reviewed and enhanced, and techniques for hidden-surface removal, shadow generation, highlighting, and texture in images are covered. Also included is an introduction to the animation principles.

MAIN COURSE

Course code	CIT 371
Course Title	Introduction to Computer Graphics and Animation.
Course Team	Mr. F. E. Ekpenyong (Writer) – NDA
Course Editor	
Programme Leader	
Course Coordinator	

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island

Lagos

Abuja Office
No. 5 Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

Printed 2009

ISBN:

CONTENTS **PAGE**

Unit2	Hardware, Software and Display Devices
Unit3	Graphics Data Structure
Unit4	Colour Theory
Unit5	Image Representation

Module 2 Geometric Modelling

Unit1	Basic Line drawing
Unit 2	Mathematics of CG
Unit 3	Curve and Surface Design
Unit 4	Ray Tracing
Unit 5	Texture Mapping

Module 3 3D Graphics Rendering

Unit 1	Geometric Transformation
Unit 2	Scan Conversion
Unit 3	Three-dimensional Viewing
Unit 4	3D Transform and Animation
Unit 5	Hidden Surface Elimination

MODULE 1: DEFINITION AND CONCEPTS OF COMPUTER GRAPHICS

UNIT 1 COMPUTER GRAPHICS AND APPLICATIONS

CONTENTS

1.0 Introduction

Given the state of current technology, it would be possible to design an entire university major to cover everything (important) that is known about computer graphics. In this introductory course, we will attempt to cover only the merest *fundamentals* upon which the field is based. Nonetheless, with these fundamentals, you will have a remarkably good insight into historical development of Computer graphics, the various application areas and the graphics rendering pipeline.

2.0 Objectives

Upon successful completion of this unit, students will:

- have an understanding of the history of computer graphics
- have gained an appreciation for the art of computer graphics
- Understand the basic Graphic rendering pipeline.

3.0 MAIN CONTENT

3.1 Definition of Computer Graphics.

Computer graphics generally means creation, storage and manipulation of models and images. Such models come from diverse and expanding set of fields including physical, mathematical, artistic, biological, and even conceptual (abstract) structures.

“Perhaps the best way to define computer graphics is to find out what it is not. It is not a machine. It is not a computer, nor a group of computer programs. It is not the know-how of a graphic designer, a programmer, a writer, a motion picture specialist, or a reproduction specialist.

Computer graphics is all these –a consciously managed and documented technology directed toward communicating information accurately and descriptively.

3.2 History

3.1.1 The Age of Sutherland

In the early 1960's IBM, Sperry-Rand, Burroughs and a few other computer companies existed. The computers of the day had a few kilobytes of memory, no operating systems to speak of and no graphical display monitors. The peripherals were Hollerith punch cards, line printers, and roll-paper plotters. The only programming languages supported were assembler, FORTRAN, and Algol. Function graphs and “Snoopy" calendars were about the only graphics done.

In 1963 Ivan Sutherland presented his paper Sketchpad at the Summer Joint Computer Conference. Sketchpad allowed interactive design on a vector graphics display monitor with a light pen input device. Most people mark this event as the origins of computer graphics.

3.1.2 The Middle to Late '60's

Software and Algorithms

Jack Bresenham taught us how to draw lines on a raster device. He later extended this to circles. Anti-aliased lines and curve drawing is a major topic in computer graphics. Larry Roberts pointed out the usefulness of homogeneous coordinates, matrices and hidden line detection algorithms. Steve Coons introduced parametric surfaces and developed early computer aided geometric design concepts. The earlier work of Pierre Bézier on parametric curves and surfaces also became public. Author Appel at IBM developed hidden surface and shadow algorithms that were pre-cursors to ray tracing. The fast Fourier transform was discovered by Cooley and Tukey. This algorithm allow us to better understand signals and is fundamental for developing antialiasing techniques. It is also a precursor to wavelets.

Hardware and Technology

Doug Englebart invented the mouse at Xerox PARC. The Evans & Sutherland Corporation and General Electric started building flight simulators with real-time raster graphics. The floppy disk was invented at IBM and the microprocessor was invented at Intel. The concept of a research network, the ARPANET, was developed.

3.1.3 The Early '70's

The state of the art in computing was an IBM 360 computer with about 64 KB of memory, a Tektronix 4014 storage tube, or a vector display with a light pen (but these were very expensive).

Software and Algorithms

Rendering (shading) were discovered by Gouraud and Phong at the University of Utah. Phong also introduced a reflection model that included specular highlights. Keyframe based animation for 3-D graphics was demonstrated. Xerox PARC developed a ``paint'' program. Ed Catmull introduced parametric patch rendering, the z-buffer algorithm, and texture mapping. BASIC, C, and Unix were developed at Dartmouth and Bell Labs.

Hardware and Technology

An Evans & Sutherland Picture System was the high-end graphics computer. It was a vector display with hardware support for clipping and perspective. Xerox PARC introduced the Altos personal computer, and an 8 bit computer was invented at Intel.

3.1.4 The Middle to Late '70's

Software and Algorithms

Turned Whitted developed recursive ray tracing and it became the standard for photorealism, living in a pristine world. Pascal was the programming language everyone learned.

Hardware and Technology

The Apple I and II computers became the first commercial successes for personal computing. The DEC VAX computer was the mainframe (mini) computer of choice. Arcade games such as Pong and Pac Mac became popular. Laser printers were invented at Xerox PARC.

3.1.5 The Early '80's

Hardware and Technology
The IBM PC was marketed in 1981 The Apple MacIntosh started production in 1984, and microprocessors began to take off, with the Intel x86 chipset, but these were still toys. Computers with a mouse, bitmapped (raster) display, and Ethernet became the standard in academic and science and engineering settings.

3.1.6 The Middle to Late '80's

Software and Algorithms

Jim Blinn introduces blobby models and texture mapping concepts. Binary space partitioning (BSP) trees were introduced as a data structure, but not many realized how useful they would become. Loren Carpenter starting exploring fractals in computer graphics. Postscript was developed by John Warnock and Adobe was formed. Steve Cook introduced stochastic sampling to ray tracing. Paul Heckbert taught us to ray trace Jello(this is a joke;) Character animation became the goal for animators. Radiosity was introduced by the Greenberg and folks at Cornell. Photoshop was marketed by Adobe. Video arcade games took off, many people/organizations started publishing on the desktop. Unix and X windows were the platforms of choice with programming in C and C++, but MS-DOS was starting to rise.

Hardware and Technology

Sun workstations, with the Motorola 680x0 chipset became popular as advanced workstation a in the mid 80's. The Video Graphics Array (VGA) card was invented at IBM. Silicon Graphics (SGI) workstations that supported real-time raster line drawing and later polygons became the computer graphicists desired. The data glove, a precursor to virtual reality, was invented at NASA. VLSI for special purpose graphics processors and parallel processing became hot research areas.

3.1.7 The Early '90's

The computer to have now was an SGI workstation with at least 16 MB of memory, at 24-bit raster display with hardware support for Gouraud shading and z-buffering for hidden surface removal. Laser printers and single frame video recorders were standard. Unix, X and Silicon Graphics GL were the operating systems, window system and application programming interface (API) that graphicist used. Shaded raster graphics were starting to be introduced in motion pictures. PCs started to get decent, but still they could not support 3-D graphics, so most programmer's wrote software for scan conversion (rasterization) used the painter's algorithm for hidden surface removal, and developed “tricks”" for real-time animation.

Software and Algorithms

Mosaic, the first graphical Internet browser was written by xxx at the University of Illinois, National Center for Scientific Applications (NCSA). MPEG standards for compressed video began to be promulgated. Dynamical systems (physically based modeling) that allowed animation with collisions, gravity, friction, and cause and effects were introduced. In 1992 OpenGL became the standard for graphics APIs In 1993, the World Wide Web took off. Surface subdivision algorithms were rediscovered. Wavelets begin to be used in computer graphics.

Hardware and Technology

Hand-held computers were invented at Hewlett-Packard about 1991. Zip drives were invented at Iomega. The Intel 486 chipset allowed PC to get reasonable floating point performance. In 1994, Silicon Graphics produced the Reality Engine: It had hardware for real-time texture mapping. The Ninetendo 64 game console hit the market providing Reality Engine-like graphics for the masses of games players. Scanners were introduced.

3.1.8 The Middle to Late '90's

The PC market erupts and supercomputers begin to wane. Microsoft grows, Apple collapses, but begins to come back, SGI collapses, and lots of new startups enter the graphics field.

Software and Algorithms

Image based rendering became the area for research in photo-realistic graphics. Linux and open source software become popular.

Hardware and Technology

PC graphics cards, for example 3dfx and Nvidia, were introduced. Laptops were introduced to the market. The Pentium chipset makes PCs almost as powerful as workstations. Motion capture, begun with the data glove, becomes a primary method for generating animation sequences. 3-D video games become very popular: DOOM (which uses BSP trees), Quake, Mario Brothers, etc. Graphics effects in movies becomes pervasive: Terminator 2, Jurassic Park, Toy Story, Titanic, Star Wars I. Virtual reality and the Virtual Reality Meta (Markup) Language (VRML) become hot areas for research. PDA's, the Palm Pilot, and flat panel displays hit the market.

3.1.9 The '00's

Today most graphicist want an Intel PC with at least 256 MB of memory and a 10 GB hard drive. Their display should have graphics board that supports real-time texture mapping. A flatbed scanner, color laser printer, digital video camera, DVD, and MPEG encoder/decoder are the peripherals one wants. The environment for program development is most likely Windows and Linux, with Direct 3D and OpenGL, but Java 3D might become more important. Programs would typically be written in C++ or Java.

What will happen in the near future -- difficult to say, but high definition TV (HDTV) is poised to take off (after years of hype). Ubiquitous, untethered, wireless computing should become widespread, and audio and gestural input devices should replace some of the functionality of the keyboard and mouse.

You should expect 3-D modeling and video editing for the masses, computer vision for robotic devices and capture facial expressions, and realistic rendering of difficult things like a human face, hair, and water. With any luck C++ will fall out of favor.

3.1.10 Ethical Issues

Graphics has had a tremendous affect on society. Things that affect society often lead to ethical and legal issues. For example, graphics are used in battles and their simulation, medical diagnosis, crime re-enactment, cartoons and films. The ethical role played by a computer graphic is in the use of graphics programs that may be used for these and other purposes is discussed and analyzed in the notes on Ethics.

3.2 Application of Computer Graphics

3.2.1 Medical Imaging

There are few endeavors more noble than the preservation of life. Today, it can honestly be said that computer graphics plays an significant role in saving lives. The range of application spans from tools for teaching and diagnosis, all the way to treatment. Computer graphics is tool in medical applications rather than an a mere artifact. No cheating or tricks allowed.

3.2.2 Scientific Visualization

Computer graphics makes vast quantities of data accessible. Numerical simulations frequently produce millions of data values. Similarly, satellite-based sensors amass data at rates beyond our abilities to interpret them by any other means than visually. Mathematicians use computer graphics to explore abstract and high-dimensional functions and spaces. Physicists can use computer graphics to transcend the limits of scale. With it they can explore both microscopic and macroscopic world

3.2.3 Computer Aided Design

Computer graphics has had a dramatic impact on the design process. Today, most mechanical and electronic designs are executed entirely on computer. Increasingly, architectural and product designs are also migrating to the computer. Automated tools are also available that verify tolerances and design constraints directly from CAD designs. CAD designs also play a key role in a wide range of processes from the design of tooling fixtures to manufacturing.

3.2.4 Graphical User Interfaces (GUIs)

Computer graphics is an integral part of everyday computing. Nowhere is this fact more evident than the modern computer interface design. Graphical elements such as windows, cursors, menus, and icons are so common place it is difficult to imagine computing without them. Once graphics programming was considered a speciality. Today, nearly all professional programmers must have an understanding of graphics in order to accept input and present output to users.

3.2.5 Games

Games are an important driving force in computer graphics. In this class we are going to talk about games. We'll discuss on how they work. We'll also question how they get so much done with so little to work with.

3.2.6 Entertainment

If you can imagine it, it can be done with computer graphics. Obviously, Hollywood has caught on to this. Each summer, we are amazed by state-of-the-art special effects. Computer graphics is now as much a part of the entertainment industry as stunt men and makeup. The entertainment industry plays many other important roles in the field of computer graphics.

3.3 What is Interactive Computer Graphics?

User controls contents, structure, and appearance of objects and their displayed images via rapid visual feedback.

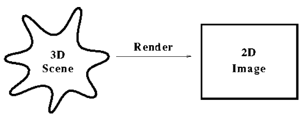
Basic components of an interactive graphics system input (e.g., mouse, tablet and stylus, force feedback device, scanner, live video streams…), processing (and storage), display/output (e.g., screen, paper-based printer, video recorder, non-linear editor.

3.4 What do we need in computer graphics?

In computer graphics we work with points and vectors defined in terms of some coordinate frame (a positioned coordinate system). We also need to change coordinate representation of points and vectors, hence to transform between different coordinate frames. Hence a mathematical background of geometry and algebra is very essential and also a knowledge of basic programming in C language.

3.5 The Graphics Rendering Pipeline

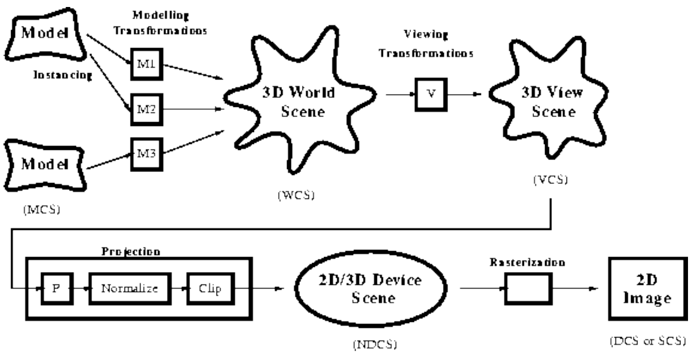
Rendering is the conversion of a scene into an image:



The scene composed of models in three space.
Models, composed of primitives, supported by the rendering system. Models entered by hand or created by a program.

For our purposes today, models already generated. The image drawn on monitor, printed on laser printer, or written to a raster in memory or a file. These different possibilities require us to consider device independence.

Classically, “model” to “scene”' to “image” conversion broken into finer steps, called the graphics pipeline. Commonly implemented in graphics hardware to get interactive speeds. At a high level, the graphics pipeline usually looks like



Each stage refines the scene, converting primitives in modelling space to primitives in device space, where they are converted to pixels (rasterized).

A number of coordinate systems are used:

MCS: Modeling Coordinate System.
WCS: World Coordinate System.
VCS: Viewer Coordinate System.
NDCS: Normalized Device Coordinate System.
DCS or SCS: Device Coordinate System or equivalently the Screen Coordinate System.

Keeping these straight is the key to understanding a rendering system. Transformation between two coordinate systems represented with matrix. Derived information may be added (lighting and shading) and primitives may be removed (hidden surface removal) or modified (clipping).

4.0 Conclusion

Computer graphics (CG) is the field of visual computing, where one utilizes computers both to generate visual images synthetically and to integrate or alter visual and spatial information sampled from the real world.

5.0 Summary

In this unit, we have learnt

i. the evolution of computer graphics from the early days to the recent times, the technological innovations
ii. The various application areas of computer graphics
iii. What is needed in computer graphics
iv. Graphics rendering pipeline.

6.0 Tutor Marked Assignment

1a. What is Computer Graphics all about?
 b. Write on the history of Computer graphics under the following:
  i) the age of Sutherland
  ii) the Early '70's
  iii) the '00's
2a. What are the application areas of computer graphics?
 b. Briefly describe the basic graphics rendering pipeline.

7.0 References/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and

Practice, 2nd Ed. in C, Addison-Wesley, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

UNIT 2 HARDWARE, SOFTWARE AND DISPLAY DEVICES

CONTENTS

1.0 Introduction

Computer graphics don't work in isolation. They require Hardware- the physical component that houses the software. The software tools to create graphics applications are also needed and display devices for effective output are not left. These and their functionalities, we shall discuss in this unit.

2.0 Objectives

At the completion of this unit, the students are expected to get acquainted with the following:

· Basics of graphics hardware and software
· Graphics display devices
· Hard copy technologies
· Display technologies
· Raster and random scan display systems

3.0 Main Content

Types of Input Devices

Devices can be described either by
- Physical properties
    · Mouse
    · Keyboard

· Trackball

- Logical Properties

· What is returned to program via API

A position

An object identifier

Input Devices are also categorized as follows

String: produces string of characters. (e.g keyboard)

Valuator: generates real number between 0 and 1.0 (e.g.knob)

Locator: User points to position on display (e.g. mouse)

Pick: User selects location on screen (e.g. touch screen in restaurant, ATM)

3.1 Graphics Software

Graphics software (that is, the software tool needed to create graphics applications) has taken the form of subprogram libraries. The libraries contain functions to do things like: draw points, lines, polygons apply transformations fill areas with color handle user interactions. An important goal has been the development of standard hardware-independent libraries such as:

CORE GKS (Graphical Kernel Standard)

PHIGS (Programmer's Hierarchical Interactive Graphics System)

X Windows OpenGL

Hardware vendors may implement some of the OpenGL primitives in hardware for speed.

3.2 OpenGL:

gl: basic graphics operations

glu: utility package containing some higher-level modeling capabilities (curves, splines)

glut: toolkit. adds platform-independent functions for window management, mouse and keyboard interaction, pull-down menus

glui: adds support for GUI tools like buttons, sliders, etc.

Open Inventor. An object-oriented API built on top of OpenGL.

VRML. Virtual Reality Modeling Language. Allows creation of a model which can then be rendered by a browser plug-in. Java3d. Has hierarchical modeling features similar to VRML.

POVray. A ray-tracing renderer

3.3 Hardware

"Vector graphics" Early graphic devices were line-oriented. For example, a "pen plotter" from H-P. Primitive operation is line drawing. "Raster graphics" Today's standard. A raster is a 2-dimensional grid of pixels (picture elements). Each pixel may be addressed and illuminated independently. So the primitive operation is to draw a point; that is, assign a color to a pixel. Everything else is built upon that. There are a variety of raster devices, both hardcopy and display.

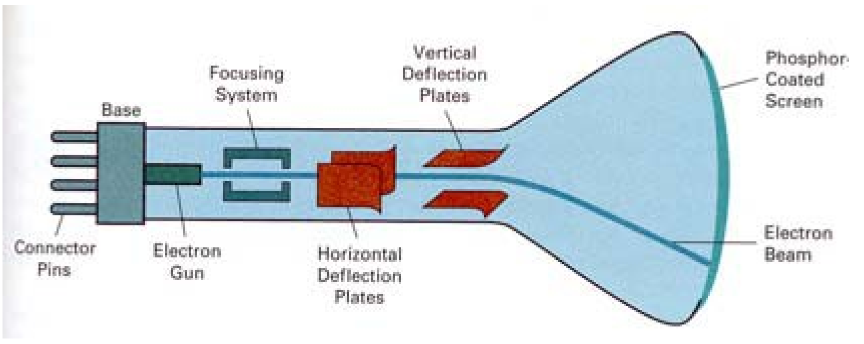
Hardcopy:
Laserprinter
Ink-jet printer
Film recorder
Electrostatic printer
Pen plotter

3.4 Display Hardware

An important component is the "refresh buffer" or "frame buffer" which is a random-access memory containing one or more values per pixel, used to drive the display. The video controller translates the contents of the frame buffer into signals used by the CRT to illuminate the screen. It works as follows:

1. The display screen is coated with "phospors" which emit light when excited by an electron beam. (There are three types of phospor, emitting red, green, and blue light.) They are arranged in rows, with three phospor dots (R, G, and B) for each pixel.
2. The energy exciting the phosphors dissipates quickly, so the entire screen must be refreshed 60 times per second.
3. An electron gun scans the screen, line by line, mapping out a scan pattern. On each scan of the screen, each pixel is passed over once. Using the contents of the frame buffer, the controller controls the intensity of the beam hitting each pixel, producing a certain color.

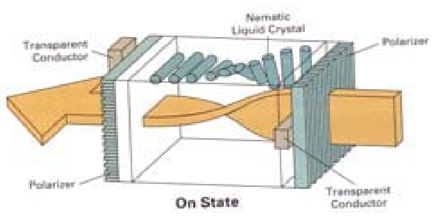
3.4.1 Cathode Ray Tube (CRT)

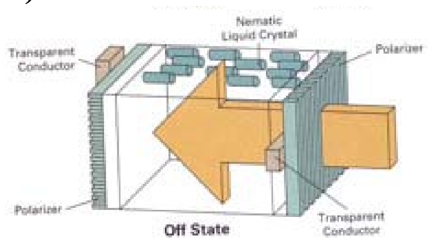


Electron gun sends beam aimed (deflected) at a particular point on the screen, Traces out a path on the screen, hitting each pixel once per cycle. "scan lines" · Phosphors emit light (phosphoresence); output decays rapidly (exponentially - 10 to 60 microseconds) · As a result of this decay, the entire screen must be redrawn (refreshed) at least 60 times per second. This is called the refresh rate . If the refresh rate is too

slow, we will see a noticeable flicker on the screen. CFF (Critical Fusion Frequency) is the minimum refresh rate needed to avoid flicker. This depends to some degree on the human observer. Also depends on the persistence of the phosphors; that is, how long it takes for their output to decay. · The horizontal scan rate is defined as the number of scan lines traced out per second. · The most common form of CRT is the shadow-mask CRT. Each pixel consists of a group of three phosphor dots (one each for red, green, and blue), arranged in a triangular form called a triad. The shadow mask is a layer with one hole per pixel. To excite one pixel, the electron gun (actually three guns, one for each of red, green, and blue) fires its electron stream through the hole in the mask to hit that pixel. · The dot pitch is the distance between the centers of two triads. It is used to measure the resolution of the screen.
(Note: On a vector display, a scan is in the form of a list of lines to be drawn, so the time to refresh is dependent on the length of the display list.)

3.4.2 Liquid Crystal Display (LCD)





A liquid crystal display consists of 6 layers, arranged in the following order (back-to-front):

A reflective layer which acts as a mirror
A horizontal polarizer, which acts as a filter, allowing only the horizontal component of light to pass through
A layer of horizontal grid wires used to address individual pixels

The liquid crystal layer

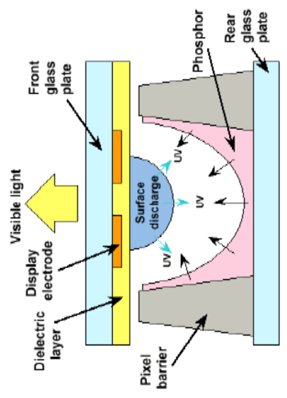
A layer of vertical grid wires used to address individual pixels
A vertical polarizer, which acts as a filter, allowing only the vertical component of light to pass through

How it works:

The liquid crystal rotates the polarity of incoming light by 90 degrees.
Ambient light is captured, vertically polarized, rotated to horizontal polarity by the liquid crystal layer, passes through the horizontal filter, is reflected by the reflective layer, and passes back through all the layers, giving an appearance of lightness. However, if the liquid crystal

molecules are charged, they become aligned and no longer change the polarity of light passing through them. If this occurs, no light can pass through the horizontal filter, so the screen appears dark.
The principle of the display is to apply this charge selectively to points in the liquid crystal layer, thus lighting or not lighting points on the screen. Crystals can be dyed to provide color. An LCD may be backlit, so as not to be dependent on ambient light.
TFT (thin film transistor) is most popular LCD technology today.
Plasma Display Panels

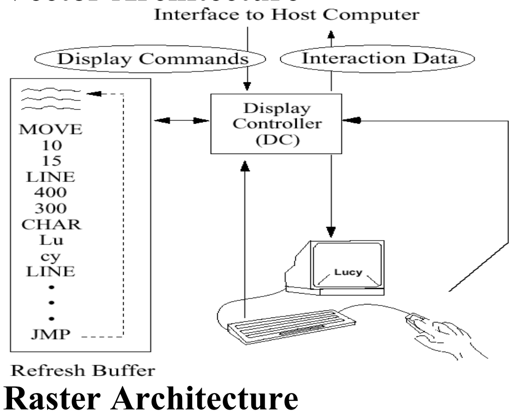
Promising for large format displays
Basically fluorescent tubes
High-voltage discharge excites gas mixture (He, Xe)
Upon relaxation UV light is emitted
UV light excites phosphors
Large viewing angle



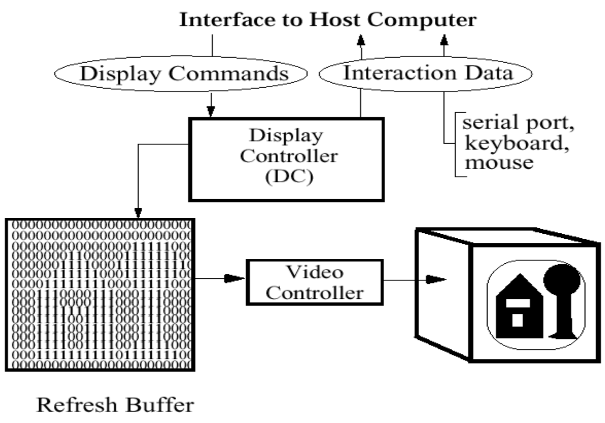
3.4.3 Vector Displays

Oscilloscopes were some of the 1st computer displays
Used by both analog and digital computers
Computation results used to drive the vertical and horizontal axis (X-Y)
Intensity could also be controlled (Z-axis)
Used mostly for line drawings
Called vector, calligraphic or affectionately stroker displays
Display list had to be constantly updated
(except for storage tubes)

Vector Architecture

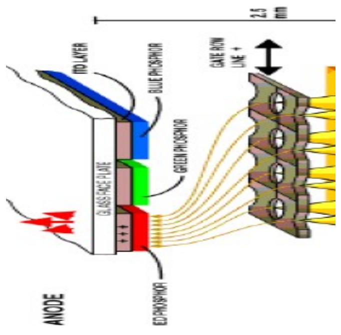


Raster Architecture



Raster display stores bitmap/pixmap in refresh buffer, also known as bitmap, frame buffer; be in separate hardware (VRAM) or in CPU's main memory (DRAM) Video controller draws all scan-lines at Consistent >60 Hz; separates update rate of the frame buffer and refresh rate of the CRT

Field Emission Devices (FEDs)



Works like a CRT with multiple electron guns at each pixel uses modest voltages applied to sharp points to produce strong E fields
Reliable electrodes proven difficult to produce
Limited in size
Thin, and requires a vacuum

Interfacing between the CPU and the Display

A typical video interface card contains a display processor, a frame buffer, and a video controller. The frame buffer is a random access memory containing some memory (at least one bit) for each pixel, indicating how the pixel is supposed to be illuminated. The depth of the frame buffer measures the number of bits per pixel. A video controller then reads from the frame buffer and sends control signals to the monitor, driving the scan and refresh process. The display processor processes software instructions to load the frame buffer with data.

(Note: In early PCs, there was no display processor. The frame buffer was part of the physical address space addressable by the CPU. The CPU was responsible for all display functions.)
Some Typical Examples of Frame Buffer Structures:

1. For a simple monochrome monitor, just use one bit per pixel.
2. A gray-scale monitor displays only one color, but allows for a range of intensity levels at each pixel. A typical example would be to use 6-8 bits per pixel, giving 64-256 intensity levels. For a color monitor, we

need a range of intensity levels for each of red, green, and blue. There are two ways to arrange this.
3. A color monitor may use a color lookup table (LUT). For example, we could have a LUT with 256 entries. Each entry contains a color represented by red, green, and blue values. We then could use a frame buffer with depth of 8. For each pixel, the frame buffer contains an index into the LUT, thus choosing one of the 256 possible colors. This approach saves memory, but limits the number of colors visible at any one time.
4. A frame buffer with a depth of 24 has 8 bits for each color, thus 256 intensity levels for each color. 224 colors may be displayed. Any pixel can have any color at any time. For a 1024x1024
monitor we would need 3 megabytes of memory for this type of frame buffer. The display processor can handle some medium-level functions like scan conversion (drawing lines, filling polygons), not just turn pixels on and off. Other functions: bit block transfer, display list storage. Use of the display processor reduces CPU involvement and bus traffic resulting in a faster processor. Graphics processors have been increasing in power faster than CPUs, a new generation every 6-9 months. example: 10 3E. NVIDIA GeForce FX
· 125 million transistors (GeForce4: 63 million)
· 128MB RAM
· 128-bit floating point pipeline
One of the advantages of a hardware-independent API like OpenGL is that it can be used with a wide range of CPU-display combinations, from software-only to hardware-only. It also means that a fast video card may run slowly if it does not have a good implementation of OpenGL.

4.0 Conclusion
Computer graphics ultimately involves creation, storage and manipulation of models and images. This however is made realistic through effective hardware, software and display devices.

5.0 Summary
In this module, we have leant that:
i. Input devices are of various types and can further be categorized.
ii. Computer Graphics software are the software require to run computer graphics.
iii. There are different display hardware such as the LCD, CRT, etc. Also learnt about their architecture and functionality

6.0 Tutor Marked Assignment

1. Explain about the display technologies?
2. Explain various display devices?
3. What are the different hardware and software of graphics?

5. List five graphic hard copy devices for each one briefly explain?

- How it works.
- Its advantages and limitations.
- The circumstances when it would be more useful.

7.0 References/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

UNIT 3 GRAPHICS DATA STRUCTURES

Contents **Page**

1.0 Introduction

In recent years, methods from computational geometry have been widely adopted by the computer graphics community. Many solutions draw their elegance and efficiency from the mutually enriching combination of such geometrical data structures with computer graphics algorithms. This course imparts a working knowledge of a number of essential geometric data structures and explains their elegant use in several representatives, diverse, and current areas of research in computer graphics such as terrain visualization, texture synthesis, modelling, and tesselation. Our selection of data structures and algorithms consists of well known concepts, which are both powerful and easy to implement, ranging from quadtrees to BSP trees and bounding volume trees.

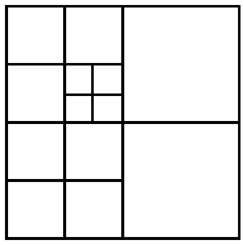
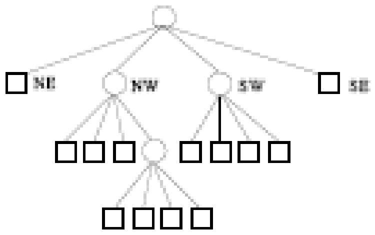
2.0 Objectives

The major objective of this unit is to present each geometric data structure, familiarize students with some very versatile and ubiquitous geometric data structures, enable them to readily recognize geometric problems during their work, modify the algorithms to their needs, and hopefully make them familiar with the basic principles of graphics data and the type of problems in the area.

3.0 Main Content

3.1 Quadtrees

A *quadtree* is a rooted tree so that every internal node has four children. Every node in the tree corresponds to a square. If a node *v* has children, their corresponding squares are the four quadrants, as shown

Quadtrees can store many kinds of data. We will describe the variant that stores a set of points and suggest a recursive definition. A simple recursive splitting of squares is continued until there is only one point in a square. Let *P* be a set of points. The definition of a quadtree for a set of points in a square $Q = [x1_Q : x2_Q]$ £ $[y1_Q : y2_Q]$ is as follows:

• If *jPj* · 1 then the quadtree is a single leaf where *Q* and *P* are stored.
• Otherwise let Q_{NE}, Q_{NW}, Q_{SW} and Q_{SE} denote the four quadrants. Let x_{mid} $:= (x1_Q + x2_Q)/2$
and $y_{mid} := (y1_Q + y2_Q)/2$, and define
$P_{NE} := \{p \in P : p_x > x_{mid} \wedge py > ymid\}$
$P_{NW} := \{p \in P : p_x \cdot x_{mid} \wedge py > ymid\}$
$P_{SW} := \{p \in P : p_x \cdot x_{mid} \wedge py \cdot ymid\}$
$P_{SE} := \{p \in P : p_x > x_{mid} \wedge py \cdot ymid\}$

The quadtree consists of a root node *v*, *Q* is stored at *v*. In the following, let *Q*(*v*) denote the square stored at *v*. Furthermore *v* has four children: The X-child is the root of the quadtree of the set P_X, where *X* is an element of the set *{NE*, *NW*, *SW*, *SE}*.

3.1.1 Uses

Quadtrees are used to partition 2-D space, while octrees are for 3-D. The two concepts are nearly identical, and I think it is unfortunate that they are given different names.
Handling Observer-Object Interactions:

Subdivide the quadtree/octree until each leaf's region intersects only a small number of objects.
Each leaf holds a list of pointers to objects that intersect its region.
Find out which leaf the observer is in. We only need to test for interactions with the objects pointed to by that leaf.

- Inside/Outside Tests for Odd Shapes

The root node represent a square containing the shape.

If a node's region lies entirely inside or entirely outside the shape, do not subdivide it.
Otherwise, do subdivide (unless a predefined depth limit has been exceeded).
Then the quadtree or octree contains information allowing us to check quickly whether a given point is inside the shape.

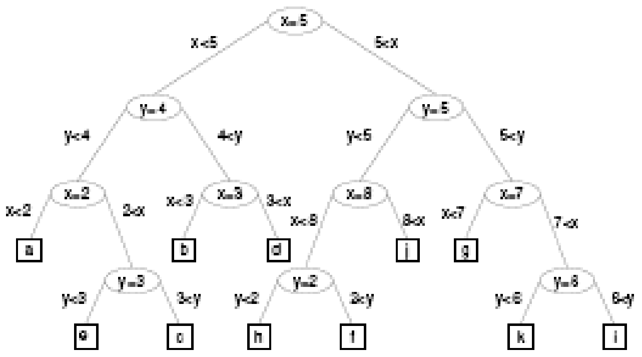
- Sparse Arrays of Spatially-Organized Data

Store array data in the quadtree or octree.
Only subdivide if that region of space contains interesting data.
This is how an octree is used in the BLUIsculpt program.

3.2 K-d-Trees

The *k-d*-tree is a natural generalization of the one dimensional search tree.
Let *D* be a set of *n* points in ℝ*k*. For convenience let *k* = 2 and let us assume that all *X*- and
Y-coordinates are different.

a



First, we search for split-value s of the X-coordinates. Then we split D by the split-line X = s into subsets:

D<s = {(x, y) ϵD; x < s} = D ∩{X < s}

D>s = {(x, y) ϵ D; x > s} = D ∩ {X > s}.
For both sets we proceed with the Y-coordinate and split-lines Y = t_1 and Y = t_2. We repeat the process recursively with the constructed subsets. Thus, we obtain a binary tree, namely the 2-tree of the point set D, as shown in the Figure above. Each internal node of the tree corresponds to a split-line.

For every node v of the 2-d-tree we define the rectangle R(v), which is the intersection of halfplanes corresponding to the path from the root to v. For the root r, R(r) is the plane itself; for the sons of r, say left and right, we produce to halfplanes R(left) and R(right) and so on. The set of rectangles {R(l) : l is a leaf}gives a partition of the plane into rectangles. Every R(l) has exactly one point of D inside.

This structure supports range queries of axis parallel rectangles. For example, if Q is an axis-parallel rectangle, the set of sites v ϵ D with v ϵ

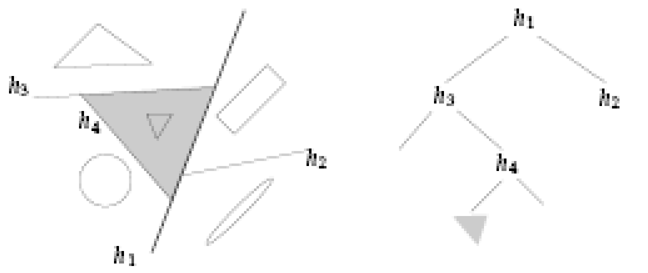
Q can be computed efficiently. We simply have to compute all nodes v with:

$R(v) \cap \neq \square$

Additionally we have to test whether the points inside the subtree of v are inside Q.

3.3 BSP Trees

BSP trees (short for binary space partitioning trees) can be viewed as a generalization of *k*-d trees. Like *k*-d trees, BSP trees are binary trees, but now the orientation and position of a splitting plane can be chosen arbitrarily. The figure below depicts the feeling of a BSP tree.



A *Binary Space Partition* tree (BSP tree) is a very different way to represent a scene, Nodes hold facets, the structure of the tree encodes spatial information about the scene. It is useful for HSR and related applications

3.2.1 Characteristics of BSP Tree

A BSP tree is a binary tree.
Nodes can have 0, 1, or two children.
Order of child nodes matters, and if a node has just 1 child, it matters whether this is its left or right child.

- Each node holds a facet.
    - This may be only part of a facet from the original scene.
    - When constructing a BSP tree, we may need to split facets.
- Organization:
    - Each facet lies in a unique plane.In 2-D, a unique line.
    - For each facet, we choose one side of its plane to be the “outside”. (The other direction is “inside”.)
    - This can be the side the normal vector points toward.
    - Rule: For each node,
        - Its left descendant subtree holds only facets “inside” it.

- Its right descendant subtree holds only facets "outside" it.

3.2.2 Construction

- To construct a BSP tree, we need:

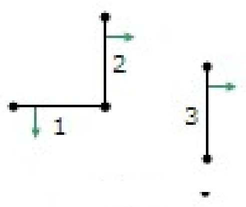
  A list of facets (with vertices).
  An "outside" direction for each.
- Procedure:

  Begin with an empty tree. Iterate through the facets, adding a new node to the tree for each new facet. The first facet goes in the root node.
  For each subsequent facet, descend through the tree, going left or right depending on whether the facet lies inside or outside the facet stored in the relevant node.
  - If a facet lies partially inside & partially outside, split it along the plane [line] of the facet.
  - The facet becomes two "partial" facets. Each inherits its "outside" direction from the original facet.
  - Continue descending through the tree with each partial facet separately.

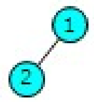
  Finally, the (partial) facet is added to the current tree as a leaf.

3.2.3 Simple Example

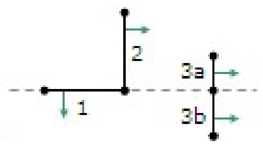
- Suppose we are given the following (2-D) facets and "outside" directions:



- We iterate through the facets in numerical order.
  Facet 1 becomes the root. Facet 2 is inside of 1.
  Thus, after facet 2, we have the following BSP tree:

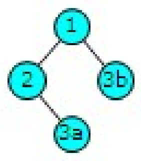


- Facet 3 is partially inside facet 1 and partially outside.
  We split facet 3 along the line containing facet 1.
  The resulting facets are 3a and 3b. They inherit their
  "outside" directions from facet 3.



- We place facets 3a and 3b separately.
  Facet 3a is inside facet 1 and outside facet 2.
  Facet 3b is outside facet 1.

The final BSP tree looks like this:

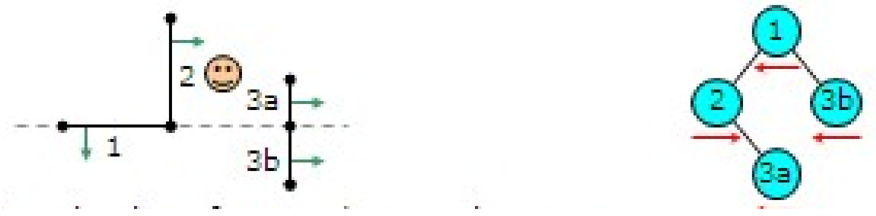


3.2.4 Traversing

An important use of BSP trees is to provide a back-to-front (or front-to-back) ordering of the facets in a scene, from the point of view of an observer.

When we say "back-to-front" ordering, we mean that no facet comes before something that appears directly behind it. This still allows nearby facets to precede those farther away.
Key idea: All the descendants on one side of a facet can come before the facet, which can come before all descendants on the other side.

- Procedure:
  For **each facet**, determine on which side of it the observer lies.
  Back-to-front ordering: Do an in-order traversal of the tree in which the subtree opposite from the observer comes before the subtree on the same side as the observer.
- Procedure:
  For **each facet**, determine on which side of it the observer lies.
  Back-to-front ordering: Do an in-order traversal of the tree in which the subtree opposite from the observer comes before the subtree on the same side as the observer.
- Our observer is inside 1, outside 2, inside 3a, outside 3b.



- Resulting back-to-front ordering: 3b, 1, 2, 3a.
- Is this really back-to-front?

Uses

- BSP trees are primarily useful when a back-to-front or front-to-back ordering is desired:

For HSR. For translucency via blending.

- Since it can take some time to construct a BSP tree, they are useful primarily for:

Static scenes. Some dynamic objects are acceptable.

- BSP-tree techniques are generally a waste of effort for small scenes. We use them on:

Large, complex scenes.

3.4 Bounding Volume Hierarchies

Like the previous hierarchical data structures, bounding volume hierarchies (BVHs) are mostly used to prevent performing an operation exhaustively on all objects. Like with previously discussed hierarchical data structures, one can improve a huge range of applications and queries using BVHs, such as ray shooting, point location queries, nearest neighbor search, view frustum and occlusion culling, geographical data bases, and collision detection (the latter will be discussed in more detail below). Often times, bounding volume (BV) hierarchies are described as the opposite of spatial partitioning schemes, such as quadtrees or BSP trees: instead of partitioning space, the idea is to partition the set of objects recursively until some leaf criterion is met. Here, objects can be anything from points to complete graphical objects. With BV hierarchies, almost all queries, which can be implemented with space partitioning schemes, can also be answered, too. Example queries and operations are ray shooting, frustum culling, occlusion culling, point location, nearest neighbor, collision detection.

3.4.1 Construction of BV Hierarchies

Essentially, there are 3 strategies to build BV trees:
• bottom-up,
• top-down,
• insertion

From a theoretical point of view, one could pursue a simple top-down strategy, which just splits the set of objects into two equally sized parts, where the objects are assigned randomly to either subset. Asymptotically, this usually yields the same query time as any other strategy. However, in practice, the query times offered by such a BV hierarchy are by a large factor worse.

During construction of a BV hierarchy, it is convenient to forget about the graphical objects or primitives, and instead deal with their BVs and consider those as the atoms. Sometimes, another simplification is to just approximate each object by its center (baryenter or bounding box center), and then deal only with sets of points during the construction. Of course, when the BVs are finally computed for the nodes, then the true extents of the objects must be considered.

In the following we will describe algorithms for each construction strategy.

3.4.2 Bottom-up

In this class, we will actually describe two algorithms. Let *B* be the set of BVs on the top-most level of the BV hierarchy that has been constructed so far. For each *bi 2 B* find the nearest neighbor *b0i 2 B*; let *di* be the distance between *bi* and *b0i*. Sort *B* with respect to *di*. Then, combine the first *k* nodes in *B* under a common father; do the same with the next
k elements from *B*, etc. This yields a new set *B0*, and the process is repeated.
Note that this strategy does not necessarily produce BVs with a small “dead space”:

the strategy would choose to combine the left pair (distance = 0), while choosing the right pair would result in much less dead space.
The second strategy is less greedy in that it computes a tiling for each level. We will describe it first in 2D. Again, let B be the set of BVs on the top-most level so far constructed, with $|B| = n$.

4.0 Conclusion

Presented in this unit are two data structures for multi-dimensional data: Quadtrees and kd-Trees . Quadtrees are better in some cases as they have less overhead for keeping track of split lines, Have faster arithmetic compared to kd-Trees, especially for graphics

5.0 Summary

The idea of the binary space partition is one with good general applicability. Some variation of it is used in a number of different structures.

BSP trees
- Split along planes containing facets.

  Quadtrees & octrees
- Split along pre-defined planes.

  kd-trees
- Split along planes parallel to coordinate axes, so as to split up the objects nicely.
  - Quadtrees are used to partition 2-D space, while octrees are for 3-D.

The two concepts are nearly identical.

6.0 Tutor Marked Assignment
1. Why do we want to build a hierarchical data structure such a bounding box hierarchy?
2. Describe a technique for constructing a bounding box hierarchy. What data structure will you
   use to store this hierarchy?

3. Describe the main difference between the following two approaches: (1) construct a bounding volume hierarchy (e.g. bounding boxes or bounding spheres), (2) construct a hierarchy of splitting planes (e.g. KD trees or BSP trees).

4. An alternative to the bounding box or bounding sphere hierarchy is to use splitting planes to divide space. Octrees, KD trees, and BSP trees are all splitting plane algorithms.
   Describe the differences between these approaches.

7.0 References/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

UNIT 4 COLOUR THEORY

Contents **Page**

1.0 Introduction:

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

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

2.0 Objectives:

Upon successful completion of this module students will be able to:

- describe the nature of colour and its numerical description
- Know the basic principles of colour mixing.

3.0 MAIN CONTENT

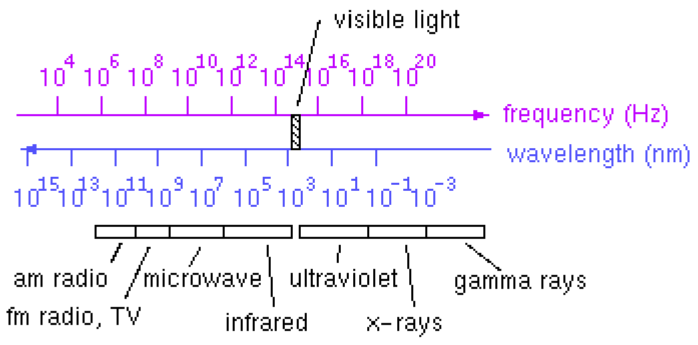
3.1 Color space

A color model is an abstract mathematical model describing the way colors can be represented as tuples of numbers, typically as three or four values or color components (e.g. RGB and CMYK are color models). However, a color model with no associated mapping function to an absolute color space is a more or less arbitrary color system with little connection to the requirements of any given application.
Adding a certain mapping function between the color model and a certain reference color space results in a definite "footprint" within the reference color space. This "footprint" is known as a gamut, and, in combination with the color model, defines a new color space. For example, Adobe
RGB and SRGB are two different absolute color spaces, both based on the RGB model.

3.1 Light

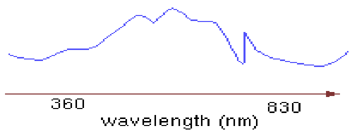
Light as we perceive it is electromagnetic radiation from a narrow band of the complete spectrum of electromagnetic radiation called the visible spectrum. The physical nature of light has elements that are like particle (when we discuss photons) and as a wave. Recall that wave can be described either in terms of its frequency, measured say in cycles per second, or the inverse quantity of wavelength. The electro-magnetic spectrum ranges from very low frequency (high wavelength) radio waves (greater than 10 centimeter in wavelength) to microwaves, infrared, visible light, ultraviolet and x-rays and high frequency (low wavelength) gamma rays (less than 0.01 nm in wavelength). Visible light lies in the range of wavelengths from around 400 to 700 nm, where nm denotes a nanometer, or 10−9 of a meter.
Physically, the light energy that we perceive as color can be described in terms of a function of wavelength λ, called the spectral distribution function or simply spectral function, $f(\lambda)$. As we walk along the wavelength axis (from long to short wavelengths), the associated colors that we perceive varying along the colors of the rainbow red, orange, yellow, green, blue, indigo, violet. (Remember the “Roy G. Biv” mnemonic.) Of course, these color names are human interpretations, and not physical divisions.

3.2 The electromagetic spectrum

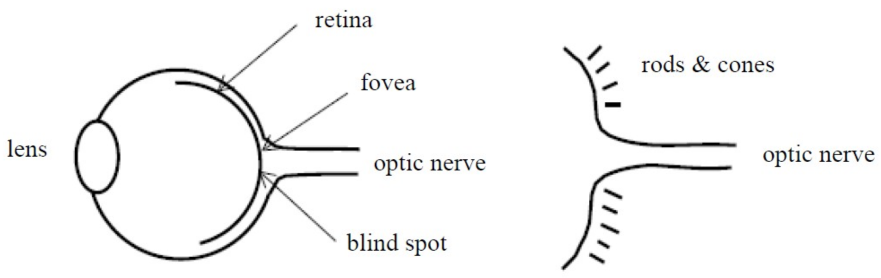


3.2.1 ROYGBIV acronym

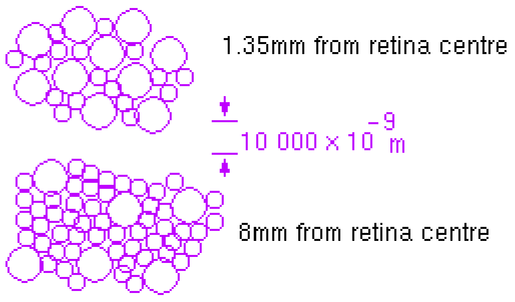
The following is a potential spectral energy distribution of light reflecting from a green wall.



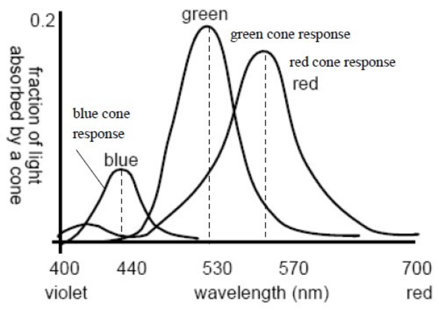
3.3 The Retina

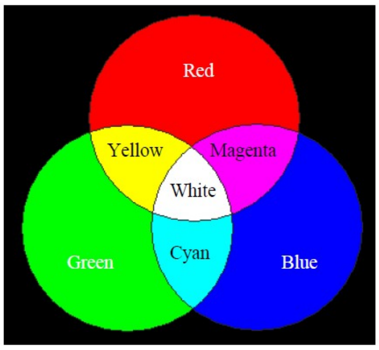


The retina has both rods and cones, as shown below. It is the cones which are responsible for colour perception.



There are three types of cones, referred to either as B, G, and R, or equivalently as S, M, and L, respectively. Their peak sensitivities are located at approximately 430nm, 560nm, and 610nm for the "average" observer. Animals exist with both fewer and more types of cones. The photopigments in rods and cones are stimulated by absorbed light, yielding a change in the cell membrane potential. The different types of cells have different spectral sensitivities:





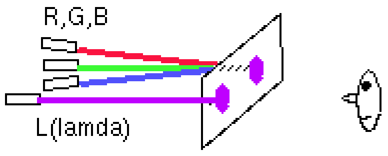
3.4 Mapping from Reality to Perception

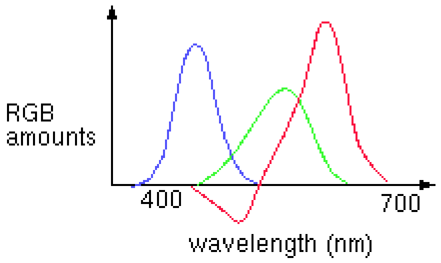
- metamers: different spectra that are perceptually identical
- colour perception: stimulation of 3 types of cones
- surround effects, adaptation

Different spectra can be perceptually identical to the eye. Such spectra are called metamers. Our perception of colour is related only to the stimulation of three types of cones. If two different spectra stimulate the three cone types in the same way, they will be perceptually indistinguishable.

3.5 Colour Matching

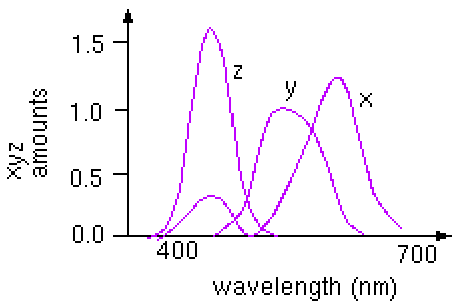
In order to define the perceptual 3D space in a "standard" way, a set of experiments can (and have been) carried by having observers try and match colour of a given wavelength, lambda, by mixing three other pure wavelengths, such as R=700nm, G=546nm, and B=436nm in the following example. Note that the phosphours of colour TVs and other CRTs do not emit pure red, green, or blue light of a single wavelength, as is the case for this experiment.





The above scheme can tell us what mix of R,G,B is needed to reproduce the perceptual equivalent of any wavelength. A problem exists, however, because sometimes the red light needs to be added to the target before a match can be achieved. This is shown on the graph by having its intensity, R, take on a negative value.

In order to achieve a representation which uses only positive mixing coefficients, he CIE ("Commission Internationale d'Eclairage") defined three new hypothetical light sources, x, y, and z, which yield positive matching curves:
 If we are given a spectrum and wish to find the corresponding X, Y, and Z quantities, we can do so by integrating the product of the spectral power and each of the three matching curves over all wavelengths. The weights X,Y,Z form the three-dimensional CIE XYZ space, as shown below.

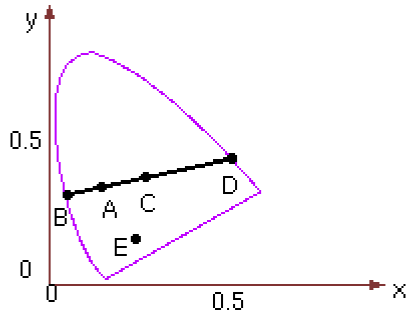


Often it is convenient to work in a 2D colour space. This is commonly done by projecting the 3D colour space onto the plane X+Y+Z=1, yielding a CIE chromaticity diagram. The projection is defined as given below, and produces the following chromaticity diagram:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

or

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$



A few definitions:

3.6 spectroradiometer

A device to measure the spectral energy distribution. It can therefore also provide the CIE xyz tristimulus values. illuminant C A standard for white light that approximates sunlight. It is defined by a colour temperature of 6774 K.

3.7 Complementary colours

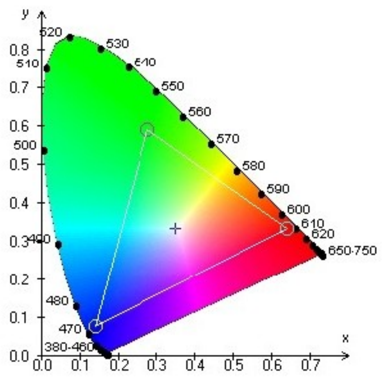
Colours which can be mixed together to yield white light. For example, colours on segment CD are complementary to the colours on segment CB.

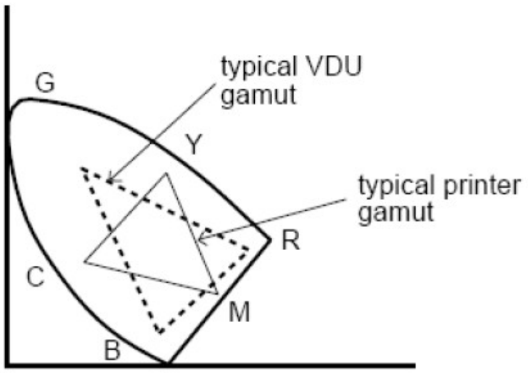
3.8 Dominant wavelength

The spectral colour which can be mixed with white light in order to reproduce the desired colour. Colour B in the above figure is the dominant wavelength for colour A.

3.9 non-spectral colours

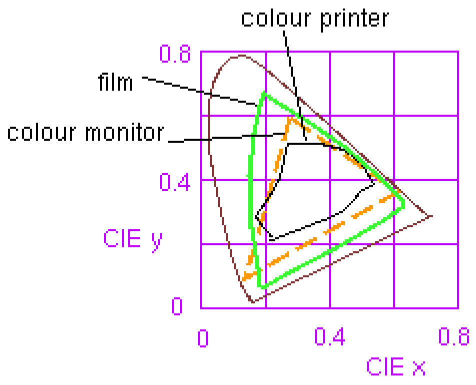
Colours not having a dominant wavelength. For example, colour E in the above figure.   perceptually uniform colour space.
A colour space in which the distance between two colours is always proportional to the perceived distance. The CIE XYZ colour space and the CIE chromaticity diagram are not perceptually uniform, as the following figure illustrates. The CIE LUV colour space is designed with perceptual uniformity in mind.





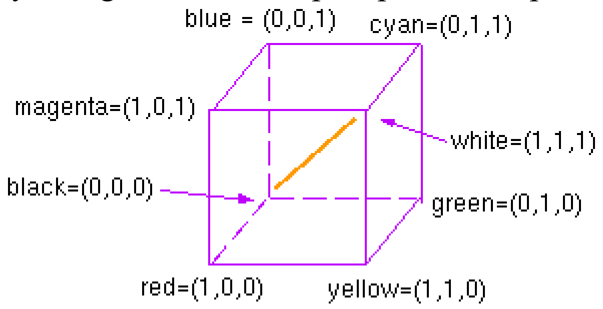
3.10 Colour Gamuts

The chromaticity diagram can be used to compare the "gamuts" of various possible output devices (i.e., monitors and printers). Note that a colour printer cannot reproduce all the colours visible on a colour monitor.

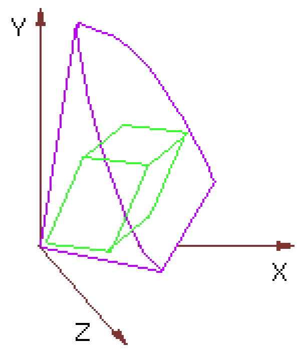


3.11 The RGB Colour Cube

The additive colour model used for computer graphics is represented by the RGB colour cube, where R, G, and B represent the colours produced by red, green and blue phosphours, respectively.



The colour cube sits within the CIE XYZ colour space as follows.



3.12 Colour Printing

Green paper is green because it reflects green and absorbs other wavelengths. The following table summarizes the properties of the four primary types of printing ink.

dye colour	absorbs	reflects
Cyan	red	blue and green
Magenta	green	blue and red
yellow	blue	red and green
Black	all	none

To produce blue, one would mix cyan and magenta inks, as they both reflect blue while each absorbing one of green and red. Unfortunately, inks also interact in non-linear ways. This makes the process of converting a given monitor colour to an equivalent printer colour a challenging problem.

Black ink is used to ensure that a high quality black can always be printed, and is often referred to as to K. Printers thus use a CMYK colour model.

3.13 Colour Conversion

Monitors are not all manufactured with identical phosphours. To convert from one colour gamut to another is a relatively simple procedure (with the exception of a few complicating factors!). Each phosphour colour can be represented by a combination of the CIE XYZ primaries, yielding the following transformation from RGB to CIE XYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

or

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The transformation $C_2 = M_2^{-1} M_1 C_1$ yields the colour on monitor 2 which is equivalent to a given colour on monitor 1. Quality conversion to-and-from printer gamuts is difficult. A first approximation is shown on the left. A fourth colour, K, can be used to replace equal amounts of CMY, as shown on the right.

K = min(C,M,Y)

C = 1 - R C' = C - K

M = 1 - G M' = M - K

Y = 1 - B Y' = Y – K

3.14 Other Colour Systems

Several other colour models also exist. Models such as HSV (hue, saturation, value) and HLS (hue, luminosity, saturation) are designed for intuitive understanding. Using these colour models, the user of a paint program would quickly be able to select a desired colour.

4.0 Conclusion

Colour is one of the most important features in a visual presentation. The perceived colour of an object is influenced by the colour of the surroundings referred to as colour interaction, colour induction, or colour assimilation.

5.0 Summary

Color is an immensely complex subject, one that draws on concepts and results from physics, physiology, psychology,art, and graphic design.
Perceived color depends on:

- Object color
- Color of lights illuminating the object
- Color of surrounding objects
- Human visual system
- Object’s absorptive, reflective, emissive, and transmissive properties

6.0 Tutor Marked Assignment

1. Draw the RGB color cube, labeling the corners of the cube and the color axes. Also show where gray colors appear in the cube.
2. Draw the CMY color cube, labeling the corners of the cube and the color axes. Also show where gray colors appear in the cube.
4. Show that if the wavelength of the hue is the same as the strength of the white light, then the purity is 0.
3. Draw a sample spectra for Blue Light.
4. What is the meaning of the pre-multiplied (R, G, B, A) tuple (0.0, 0.25, 0.25, 0.5)?
5. Representing colors as pre-multiplied (R, G, B, A) tuples, what color do we get if we composite (0.0, 0.25, 0.25, 0.5) OVER (1.0, 0.0, 1.0, 1.0)?

7.0 References/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.
J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.
P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 1: Definition and Concepts of computer graphics
Unit 5 Image Representation

Table of Contents
Title
Page

4.0 Introduction:

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

5.0 Objectives:

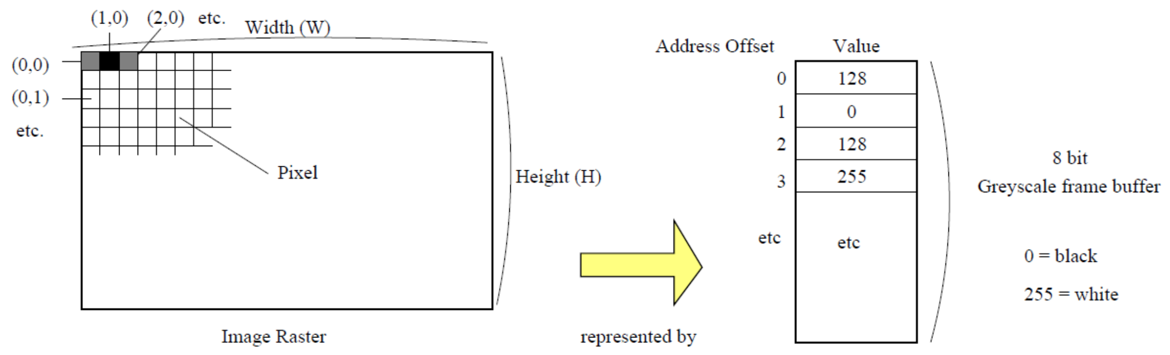
The main objective is to study how digital images are represented in a computer. This unit also explores different forms of frame-buffer for storing images, and also different ways of representing colour and key issues that arise in colour.

6.0 The Digital Image

Virtually all computing devices in use today are digital; data is represented in a discrete form using patterns of binary digits (bits) that can encode numbers within finite ranges and with limited precision. By contrast, the images we perceive in our environment are analogue. They are formed by complex interactions between light and physical objects, resulting in continuous variations in light wavelength and intensity. Some of this light is reflected in to the retina of the eye, where cells convert light into nerve impulses that we interpret as a visual stimulus.
Suppose we wish to 'capture' an image and represent it on a computer e.g. with a scanner or camera (the machine equivalent of an eye). Since we do not have infinite storage (bits), it follows that we must convert that analogue signal into a more limited digital form. We call this conversion process sampling. Sampling theory is an important part of Computer Graphics, underpinning the theory behind both image capture and manipulation.

3.1 Raster Image Representation

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



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

The pixel is the atomic unit of the image; it is coloured uniformly, its single colour represents a discrete sample of light e.g. from a captured image.

In most implementations, rasters take the form of a rectilinear grid often containing many thousands of pixels. The raster provides an orthogonal two-dimensional basis with which to specify pixel coordinates. By convention, pixels coordinates are zero-indexed and so the origin is located at the top-left of the image. Therefore pixel (W − 1,H − 1) is located at the bottom-right corner of a raster of width W pixels and height H pixels. As a note, some Graphics applications make use of hexagonal pixels instead 1, however we will not consider these on the course.

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

3.2 Hardware Frame Buffers

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

Note that the term 'frame buffer' as originally defined, strictly refers to the area of memory reserved for direct manipulation of the currently displayed image. In the early days of Graphics, special hardware was needed to store enough data to represent just that single image. However we may now manipulate hundreds of images in memory simultaneously and the term 'frame buffer' has fallen into informal use to describe any piece of storage that represents an image.

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

3.2.1 Greyscale Frame Buffer

Arguably the simplest form of frame buffer is the greyscale frame buffer; often mistakenly called 'black and white' or 'monochrome' frame buffers. Greyscale buffers encodes pixels using various shades of grey. In common implementations, pixels are encoded as an unsigned integer using 8 bits (1 byte) and so can represent 28 = 256 different shades of grey. Usually black is represented by value 0, and white by value 255. A mid-intensity grey pixel has value 128. Consequently an image of width W pixels and height H pixels requires W ×H bytes of memory for its frame buffer.

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

A = x +Wy (2.1) i.e. A bytes from the start of the frame buffer. Sometimes we use the term scan line to refer to a full row of pixels. A scan-line is therefore W pixels wide.

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

3.2.3 Pseudo-colour Frame Buffer

The pseudo-colour frame buffer allows representation of colour images. The storage scheme is identical to the greyscale frame buffer. However the pixel values do not represent shades of grey. Instead each possible value (0 − 255) represents a particular colour; more specifically, an index into a list of 256 different colours maintained by the video hardware.

The colours themselves are stored in a "Colour Lookup Table" (CLUT) which is essentially a map < colourindex, colour > i.e. a table indexed with an integer key (0−255) storing a value that represents colour. In alternative terminology the CLUT is sometimes called a palette. As we

will discuss in greater detail shortly, many common colours can be produced by adding together (mixing) varying quantities of Red, Green and Blue light.
For example, Red and Green light mix to produce Yellow light. Therefore the value stored in the CLUT for each colour is a triple (R, G,B) denoting the quantity (intensity) of Red, Green and Blue light in the mix. Each element of the triple is 8 bit i.e. has range (0 − 255) in common implementations.
The earliest colour displays employed pseudo-colour frame buffers. This is because memory was expensive and colour images could be represented at identical cost to grayscale images (plus a small storage overhead for the CLUT). The obvious disadvantage of a pseudocolour frame buffer is that only a limited number of colours may be displayed at any one time (i.e. 256 colours). However the colour range (we say gamut) of the display is 28 × 28 × 28 = 224 = 16, 777, 216 colours.
Pseudo-colour frame buffers can still be found in many common platforms e.g. both MS and X Windows (for convenience, backward compatibility etc.) and in resource constrained computing domains (e.g. low-spec games consoles, mobiles). Some low-budget (in terms of CPU cycles) animation effects can be produced using pseudo-colour frame buffers. Consider an image filled with an expanses of colour index 1 (we might set CLUT < 1,Blue >, to create a blue ocean). We could sprinkle consecutive runs of pixels with index '2,3,4,5' sporadically throughout the image. The CLUT could be set to increasing, lighter shades of Blue at those indices. This might give the appearance of waves. The colour values in the CLUT at indices 2,3,4,5 could be rotated successively, so changing the displayed colours and causing the waves to animate/ripple (but without the CPU overhead of writing to multiple locations in the frame buffer). Effects like this were regularly used in many '80s and early '90s computer games, where computational expense prohibited updating the frame buffer directly for incidental animations.

3.2.5 True-Colour Frame Buffer

The true-colour frame-buffer also represents colour images, but does not use a CLUT. The RGB colour value for each pixel is stored directly within the frame buffer. So, if we use 8 bits to represent each Red, Green and Blue component, we will require 24 bits (3 bytes) of storage per pixel.
As with the other types of frame buffer, pixels are stored in left-right, then top-bottom order. So in our 24 bit colour example, pixel (0, 0) would be stored at buffer locations 0, 1 and 2.Pixel (1, 0) at 3, 4, and 5; and so on. Pixel (x, y) would be stored at offset A where:
S = 3W
A = 3x + Sy where S is sometimes referred to as the stride of the display.
The advantages of the true-colour buffer complement the disadvantages of the pseudo-colour buffer We can represent all 16 million colours at

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

3.2.6 Alternative forms of true-colour buffer

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

7.0 Conclusion

Image generation remains an integral part of computer graphics hence how these digital images are represented in a computer cannot be overemphasized in this subject area.

8.0 Summary

In this unit, we have learnt

v. What digital image is.
vi. The concept of Raster Image Representation
vii. About the three common frame buffer formats

9.0 Tutor Marked Assignment

1. Distinguish between hardware frame buffer and Greyscale frame buffer.
2. Describe briefly using relevant diagrams the forms of frame buffers

10.0 References/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.
J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.
D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.
P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 2: Geometric Modeling
Unit 1 Basic Line drawing

Table of Contents

11.0 Introduction:

The most fundamental graphics primitive you can draw, other than a point (dot), is a line. As you will see in this unit, lines are very versatile and form the basis of a number of other graphics primitives such as polylines and simple geometrical shapes.

12.0 Objectives:

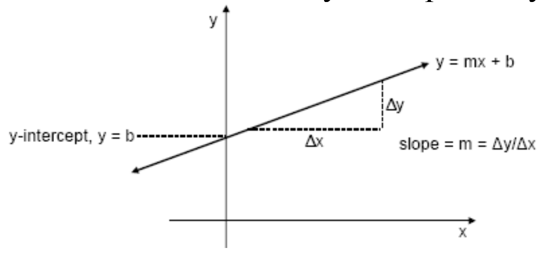
By the end of this unit, you should be able to:

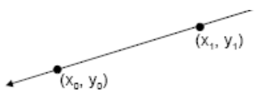
1. Know the basic representation of line and line segments.
2. Learn end understand different line drawing algorithms.
3. Learn the mathematics of circle and its use in graphic designs.

13.0 Representing Straight Lines

A line can be defined by its slope, m, and its y intercept b

Points on the line satisfy the equation $y = mx + b$





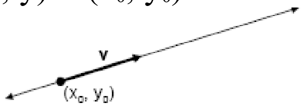
A line can be defined by two points

• e.g., (x_0, y_0), (x_1, y_1)

• To convert to the previous representation, substitute the two points into $y = mx + b$ and determine m and b

$$(1): y_0 = mx_0 + b$$
$$(2): y_1 = mx_1 + b$$
$$(2) – (1): y_1 – y_0 = m(x_1 – x_0)$$
$$m = (y_1 – y_0) / (x_1 – x_0) \text{ if } x_1 \neq x_0$$
$$b = y_0 – (y_1 – y_0)\, x_0 / (x_1 – x_0)$$
$$= (x_1y_0 – x_0y_0 – x_0y_1 + x_0y_0) / (x_1 – x_0)$$
$$= (x_1y_0 – x_0y_1) / (x_1 – x_0)$$

A line can be defined by a point and a vector. e.g., $(x, y) = (x_0, y_0) + k\mathbf{v}$ is on the line, where k is a scalar value



If **v** is the vector from (x_0, y_0) to (x_1, y_1) and $k \in [0, 1]$, this is a line segment from (x_0, y_0) to (x1, y1)

If the vector **v** has unit length and $k \in [0, L]$, this is a line segment of length L in the direction of **v**

3.1 Explicit, Implicit and Parametric Forms.

3.1.1 Explicit form:

In an explicit form, one variable is expressed as a function of the other variable(s)

In 2D, y is expressed as an explicit function of x

e.g., line: $y = mx + b$. e.g., circle: $y = \sqrt{(R^2 - x^2)}$

3.1.2 Implicit form

In an implicit form, points on the shape satisfy an implicit function

In 2D, a point (x, y) is on the shape if $f(x, y) = 0$. e.g., line: $f(x,y) = mx + b - y$

e.g., circle: $f(x,y) = R^2 - x^2 - y^2$

3.1.3 Parametric form

In a parametric form, points on the shape are expressed in terms of a separate parameter (not x or y).

Line segment:

$$x = (1\text{-}t)\, x_0 + t\, x_1 \quad t \in [0, 1],$$

$$y = (1\text{-}t)\, y_0 + t\, y_1$$

Points on the line are a linear combination of the endpoint positions (x0, y0) and (x1, y1)

circle:

$$x = R \cos\Theta \qquad \Theta \in [0, 2\pi]$$

$$y = R \sin\Theta$$

Points on the circle are swept out as Θ ranges from 0 to 2π

3.2 Drawing Straight Lines

A line segment is continuous

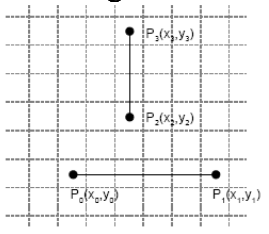
All points between the two endpoints belong to the line; a digital image is discrete

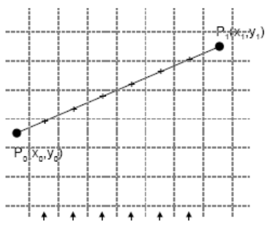
We can only set pixels at discrete locations

Pixels have a finite size

How do we know what pixels between the two endpoints should be set?

Drawing vertical and horizontal lines is straightforward



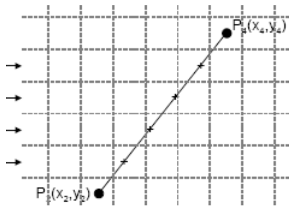


Set pixels nearest the line endpoints and all the pixels in between

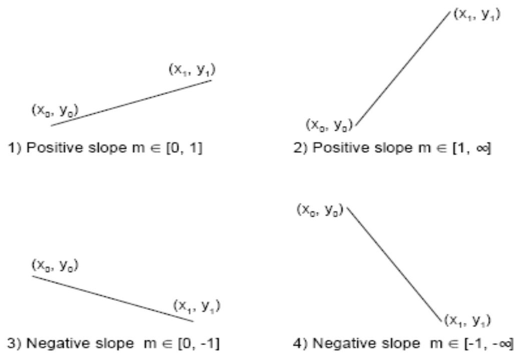
3.3 Line Drawing Algorithms:

3.3.1 Digital Differential Analyzer (DDA)

Draw the thinnest possible line with no gaps
Sample the line at equal intervals
Determine the optimal intervals from the slope of the line



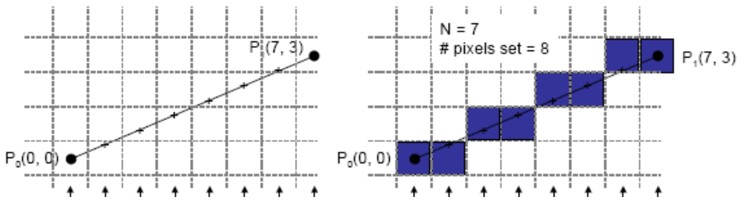
Guarantees the thinnest possible line with no gaps
For lines that are more horizontal than vertical (P0P1): sample once each time for each column of pixels between the endpoints
For lines that are more horizontal than vertical (P0P1): sample once each time for each column of pixels between the endpoints
There are 4 cases to consider


(x1, y1)
(x0, y0)
1) Positive slope m ∈ [0, 1]
(x1, y1)
(x0, y0)
2) Positive slope m ∈ [1, ∞]
(x0, y0)
(x1, y1)
3) Negative slope m ∈ [0, -1]
(x0, y0)
(x1, y1)
4) Negative slope m ∈ [-1, -∞]

We will consider the first case (the others are treated similarly):

Positive slope, m ∈ [0, 1]

For now, consider integer endpoints (x_0, y_0) and (x_1, y_1). For convenience, order the endpoints from left to right m ∈ [0,1] → These lines are more horizontal than vertical
– Choose the interval between sample points to sample once for each column between the endpoints→ Produces the thinnest possible line without gaps.
If the endpoints are at integer grid locations, the number of pixels to be set along the line (including the endpoints) is N + 1, where $N = x_1 - x_0$



Recall the parametric form of the line

$x = (1-t)\, x_0 + t\, x_1 \quad t \in [0, 1]$

$y = (1-t)\, y_0 + t\, y_1$

Sample the line at t = 0, 1/N, 2/N, … 1

$x(i) = x_0 + i$

$y(i) = (1 - t)\, y_0 + t\, y_1$

Basic algorithm:

```
N = (x_1 – x_0)
for(i = 0; i <= N; i++) {
t = i / N
setPixel (x_0 + i, round((1 – t) y_0 + t y_1))
}
```

Each new pixel requires 1 divide, 3 additions, 2 multiplications and a round

3.3.2 More efficient algorithm

Increment y between sample points

$yi = (1 - t_i)\, y0 + t_i\, y1$

$yi+1 = (1 - t_i+1)\, y_0 + t_i+1 y_1$

$y_i+1 - y_i = (-t_i+1 + t_i)\, y_0 + (t_i+1 - t_i)\, y1$

$= (-(i + 1)/N + i/N)\, y_0 + (((i + 1)/N - i/N)\, y_1$

$= (y_1 - y_0) / N$

$= (y_1 - y_0) / (x_1 - x_0)$

$= m$

For each sample, x increases by 1 and y increases by m

More efficient algorithm

```
i = x0
y = y0
m = (y1 – y0) / (x1 – x0)
(while i <= x1) {
setPixel (i, round(y))
i = i + 1
y = y + m
}
```

Each new pixel requires 2 additions and a round

To adapt the algorithm for floating point endpoints, simply change the initial conditions

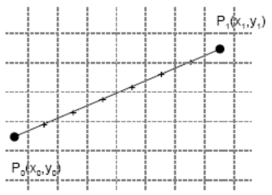
The main loop stays the same

```
i = round(x0)
m = (y1 – y0) / (x1 – x0)
y = y0 + m * (i – x0)
(while i <= x1) {
setPixel (i, round(y))
i = i + 1
```

```
        $y = y + m$
        }
```
Advantages
Eliminates the multiplications and division required for each pixel

Limitations
Time consuming round() is still required y and m are both floats – this is problematic for fixed point processors (e.g., modern PDAs and cell phones).

3.3.3 Midpoint Line Algorithm
Extends the DDA to avoid floating point calculation
Each new pixel requires one or two integer adds and a sign test



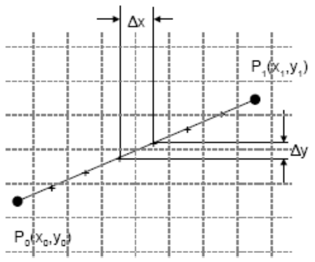
Consider the case where m ∈ [0,1]

Order the endpoints from left to right
These lines are more horizontal than vertical

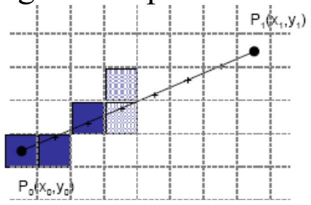
m = Δy/Δx ∈ [0,1] so Δy < Δx between each sample Point

m ∈ [0,1] → the line goes up and to the right x increases by 1 for each

sample along the line
The next pixel will be in the next column y increases along the line, but more slowly than x
For each unit change in x, y will change by less than one
• Sometimes the next pixel will be on the same row (to the right)
• Sometimes it will be on the row above (to the right and up)



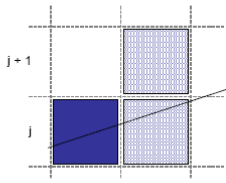
Approach
Sample once along the line at every column
At each sample point, determine if the next pixel is to the right or to the right and up from the current pixel.



Basic algorithm:

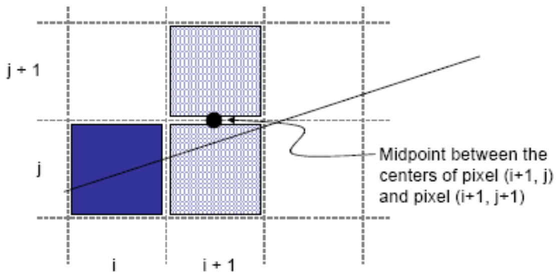
```
i = x0
j = y0
while (i <= x1) {
setPixel (i, j)
i = i + 1
if (Condition) j = j+1
}
```

Condition() determines if the next pixel is to the right or to the right and up. The key is to determine an efficient Condition()



Consider a line with slope m ∈ [0, 1]

Assume that for column i, the pixel (i, j) was set. We need to determine if the next pixel is pixel (i+1, j) or (i+1, j+1)


j + 1
j
i
i + 1
Midpoint between the centers of pixel (i+1, j) and pixel (i+1, j+1)

Consider the midpoint between these two pixel centers: the point (i+1, j+½)
If the midpoint is above the line, the pixel (i +1, j) is closer to the line
If the midpoint is below the line, the pixel (i +1, j+1) is closer to the line.

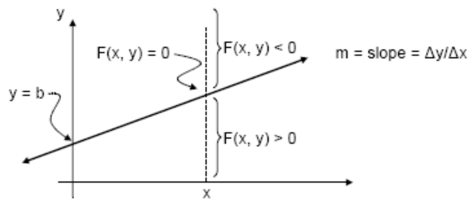
- Advance to the next sample point

Recall the basic algorithm

```
i = x0
j = y0
while (i <= x1) {
setPixel (i, j)
i = i + 1
if (Condition) j = j+1
}
```

Condition() tests whether the midpoint, (i+1, j+½), is above or below the line
We need a mathematical expression for Condition()
Use the implicit representation for the line

$$F(x,y) = mx + b - y$$



Verify that for positive m, at a given x,

If y is on the line, then $F(x, y) = 0$
If y is above the line, then $F(x,y) < 0$
If y is below the line, then $F(x,y) > 0$

To determine if the next pixel is (i+1, j) or (i+1, j+1):
Evaluate $F(i+1, j + ½)$

$$F(i+1, j + ½) = m(i + 1) + b - (j + ½)$$

The midpoint test, i.e., the desired Condition (i, j) is

$F(i+1, j + ½) > 0$

3.4 Bresenham's Algorithm

An efficient form of the midpoint algorithm uses integer math that is more amenable for fixed point processors assumes both endpoints are at integer pixel locations uses incremental arithmetic for each new pixel.

Midpoint test

$F(i, j+ ½) > 0$

$m*i + b – (j+ ½) > 0$

• Recall

$m = (y_1 – y_0) / (x_1 – x_0)$

$b = (x_1y_0 – x_0y_1) / (x_1 – x_0)$

• Multiply through by $2(x_1 – x_0)$ to get rid of all Fractions

1. ***Use integer math***

Multiply both sides of the midpoint test by 2(x1 – x0) to get rid of all fractions

• As long as x1 > x0, the modified midpoint test is equivalent to the midpoint test

Modified midpoint test

$2(y_1 – y_0)i + 2(x_1y_0 – x_0y_1) - 2(x_1 – x_0)j - (x_1 – x_0) > 0$

2. ***Use incremental arithmetic***

$F(i, j) = 2(y_1 – y_0)i + 2(x_1y_0 – x_0y_1) - 2(x_1 – x_0)j - (x_1– x_0)$

if F <= 0, the next pixel is (i+1, j)

$F(i+1, j) = 2(y_1 – y_0)(i+1) + 2(x_1y_0 – x_0y_1) - 2(x_1 – x_0)j - (x_1 – x_0)$

$= F(i,j) + 2(y_1 – y_0)$

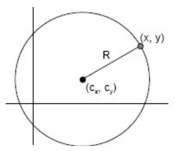
else (if F > 0), the next pixel is (i+1, j+1)

$F(i+1, j+1) = 2(y_1 – y_0)(i+1) + 2(x_1y_0 – x_0y_1) - 2(x_1 – x_0)(j+1) - (x_1 – x_0)$

$= F(i,j) + 2(y_1 – y_0) - 2(x_1 – x_0)$

3.5 Drawing Circles

A circle is the set of all points that are a distance R from a center point (cx, cy)



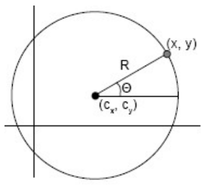
Explicit equation

y = cy + √(R2 - (x – cx)2)

Implicit equations

F(x, y) = R2 - (x – cx)2 - (y – cy)2

Parametric equation



$x = cx + R \cos Θ$ $\qquad Θ \in [0, 2π]$

$y = cy + R \sin Θ$

3.6 Symmetry

If the circle is centered at the origin, we can exploit 8-way symmetry
• If (x, y) is on the circle, then (y, x), (y, -x), (x, -y), (-x, y), (-y, x), (-y, -x) and (-x, -y) are all on the circle as well. If the circle is centered at an integer point (i, j)
• Shift the circle to the origin
• Determine which points to set using eight-way symmetry
• Shift the determined pixels back to (i, j)

Polygonal Approximation

• Approximate the circle with straight lines
Generate points at equal angular increments around the circle and draw straight lines between the points
• This general strategy is often used for more complex curves
• Tradeoff between accuracy and speed (more lines → more accurate)

Uniform Angular Sampling

• 2) Sample circle at equal angular increments
Set pixels beneath the sample points
How to sample?
• Use the parametric expression of the circle and sample at equal intervals of Θ

$$x = c_x + R \cos \Theta \quad \Theta \in [0, 2\pi]$$

$$y = c_y + R \sin \Theta$$

What increments of Θ should be used?
If increments are too big the circle may contain gaps
If increments are too small the circle may be chunky and blending artifacts may occur
Thickness may be uneven

Midpoint Circle Algorithm

The circle can be divided into some regions where x changes faster than y … and regions where y changes faster than x. We will consider one of these regions
The others can be determined by symmetry
• We will assume the circle is centered at the origin. More general circles are left as an exercise
Consider the explicit equation for this region of the circle:
$y = \sqrt{(R^2 - x^2)}, x < y$
• Note that y changes more slowly than x in this region
Approach:
Set one pixel for each column that the section intersects
Assuming the current pixel is (i, j), then the pixel in the next column will either be pixel (i+1, j) or (i+1, j-1)
Approach:

Similar to the midpoint line algorithm, we need a test to determine whether the pixel following (i, j) should be

(i+1, j) or (i+1, j-1)

Use the implicit form of a circle to determine which point to choose

$F(x, y) = R^2 – x^2 – y^2$

Verify that

$F(x, y) = 0$, for points on the circle

$F(x, y) < 0$, for points outside the circle

$F(x, y) > 0$, for points inside the circle

The midpoint test is $F(i, j–½) < 0$, where $F(x, y) = R^2 – x^2 – y^2$

If the test is false, the midpoint is inside the circle and the next pixel is (i +1, j)

If the test is true, the midpoint is outside the circle and the next pixel is (i+1, j-1)

• Basic algorithm

```
i = 0
j = round (R)
while (i <= j) {
setPixel (i, j)
i = i + 1
F(i, j-½) = R2 – i2 – (j+ ½)2
if (F(i, j+ ½) < 0) j = j-1
}
```

14.0 Conclusion

Lines are very versatile and form the basis of a number of other graphics primitives such as polylines and simple geometrical shapes. Therefore, lines are the foundation in graphics because it takes line to form other shapes.

15.0 Summary

Lines are fundamental in computer graphics other than a point (dot). Lines are represented in the form of straight, midpoint and circle. Algorithms such as Bresenham's, Digital Differential Analyzer (DDA) amongst others were discussed. It's clear that mathematics is void while discussing line drawing.

16.0 Tutor Marked Assignment

1. Find the distance between the points (5,2) and (7,3).
2. What is the normal vector to a plane containing the points (1,0,2), (2,3,0) and (1,2,4)
3. Write the equation for the line that passes through (1,2) and (3,–2).
4. Show that $w.w = |w|^2$

5. Draw the following line segments
   (1, 15) to (8, 15)
   (3.5, 13.5) to (11.5, 13.5)
   (1, 6) to (6, 11)
   (1, 5) to (8, 7)
   (1, 0) to (7, 4)

17.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 2: Geometric Modeling
Unit 2 Mathematics of Computer Graphics

Table of Contents

18.0 Introduction:

This unit is drawn upon mathematical background in linear algebra. We briefly revise some of the basics here. It describes how important vectors are in computer graphics. It includes:

- a review of vector arithmetic relating geometrical shapes to appropriate algebraic expressions.
- the development of tools for working with objects in 2D and 3D space

19.0 Objectives:

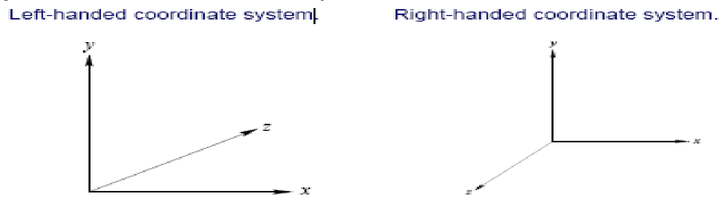
To study basic mathematical backgrounds related to computer graphics including linear algebra and geometry.

20.0 What do we need in computer graphics?

In computer graphics we work with points and vectors defined in terms of some coordinate frame (a positioned coordinate system).
We also need to change coordinate representation of points and vectors, hence to transform between different coordinate frames.

3.0.1 Definition of Cartesian coordinate system:

A Cartesian coordinate system is an orthogonal coordinate system with lines as coordinate axes.
Definition of Cartesian coordinate frame:
A Cartesian coordinate frame is a Cartesian coordinate system positioned in space.
Left-handed coordinate system


Left-handed coordinate system.
y
z
x
Right-handed coordinate system.
y
x
z

3.1 Vectors

A vector u; v;w is a directed line segment (no concept of position).
Vectors are represented in a coordinate system by a n-tuple v = (v_1; : : : ; v_n).
The dimension of a vector is dim(v) = n.
Length jvj and direction of a vector is invariant with respect to choice of Coordinate system.

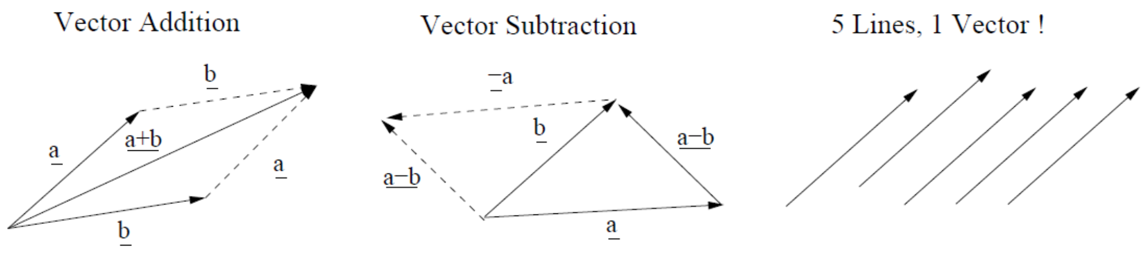
Points, Vectors and Notation

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

The superscript $[...]^T$ denotes transposition of a vector, so points p and q are column vectors (coordinates stacked on top of one another vertically). This is the convention used by most researchers with a Computer Vision background, and is the convention used throughout this course. By contrast, many Computer Graphics researchers use row vectors to represent points. For this reason you will find row vectors in many Graphics textbooks including Foley et al, one of the course texts. Bear in mind that you can convert equations between the two forms using transposition. Suppose we have a 2×2 matrix M acting on the 2D point represented by column vector p. We would write this as Mp.
If p was transposed into a row vector $p' = p^T$, we could write the above transformation $p'M^T$. So to convert between the forms (e.g. from row to column form when reading the course-texts), remember that: $Mp = (p^T M^T)^T$

3.1.1 Basic Vector Algebra

Just as we can perform basic operations such as addition, multiplication etc. on scalar values, so we can generalize such operations to vectors. The figure below summarizes some of these operations in diagrammatic form.



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

3.1.2 Vector Addition

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

3.1.3 Vector Subtraction

Vector subtraction is identical to the addition operation with a sign change, since when we negate a vector we simply flip the sign on its elements.
$-b = [-s,-t]^T$
$a - b = a + (-b) = [u - s, v - t]^T$

3.1.4 Vector Scaling

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

$\lambda_a = [\lambda_u, \lambda_v]^T$

3.1.5 Vector Magnitude

We write the length of magnitude of a vector s as |s|. We use Pythagoras' theorem to compute the magnitude:

$|\underline{a}| = \sqrt{u^2 + v^2}$ the figure shows this to be valid, since u and v are distances along the principal axes (x and y) of the space, and so the distance of a from the origin is the hypotenuse of a right-angled triangle. If we have an n-dimensional vector q = [q_1, q_2, q_3, q..., q_n] then the definition of vector magnitude generalises to:

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

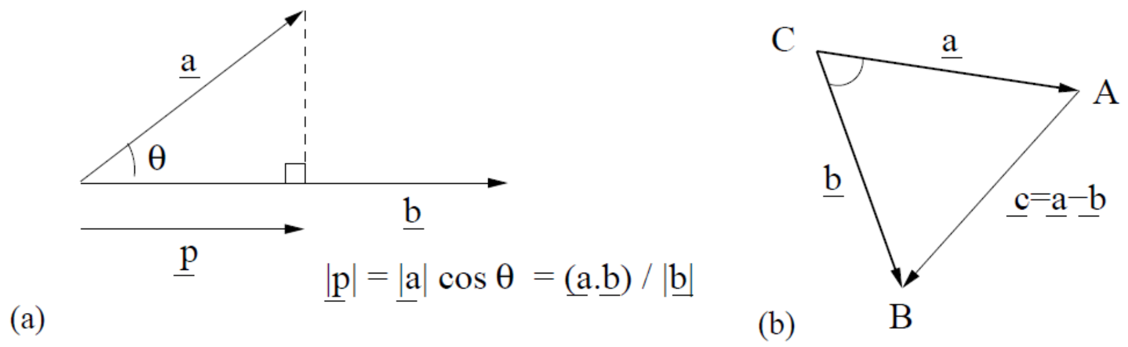

a
θ
b
p
$|\underline{p}| = |\underline{a}| \cos\theta = (\underline{a}.\underline{b}) / |\underline{b}|$
(a)
C
a
A
b
c=a−b
B
(b)

Figure (a) Demonstrating how the dot product can be used to measure the component of one vector in the direction of another (i.e. a projection, shown here as p). (b) The geometry used to prove a ∘ b = |a||b|cosθ via the Law of Cosines

3.1.6 Vector Normalisation

We can normalise a vector a by scaling it by the reciprocal of its magnitude:

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

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

3.1.7 Vector Multiplication

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

3.1.8 Dot Product

The dot product sums the products of corresponding elements over a pair of vectors. Given vectors
a = [a1, a2, a3, a..., an]^T and b = [b1, b2, b3, b..., bn]^T , the dot product is defined as:

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

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

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

Second, we can more generally compute a ◦ b, the magnitude of one vector a in the direction of another b, i.e. projecting one vector onto another. Third, we can use the dot product to compute the angle θ between two vectors (if we normalize them first). This relationship can be used to define the concept of an angle between vectors in n −dimensional spaces. It is also fundamental to most lighting calculations in Graphics, enabling us to determine the angle of a surface (normal) to a light source.
a ◦ b = |a||b|cosθ
A proof follows from the "law of cosines", a general form of Pythagoras' theorem. Consider triangle ABC in Figure b, with respect to the equation. Side C~A is analogous to vector a, and side C~B analogous to vector b. θ is the angle between C~A and C~B, and so also vectors a and b.
|c|2 = |a|2 + |b|2 − 2|a||b|cosθ …
c ◦ c = a ◦ a + b ◦ b − 2|a||b| cos θ
Now consider that c = a − b
(a − b) ◦ (a − b) = a ◦ a + b ◦ b − 2|a||b| cos θ
a ◦ a − 2(a ◦ b) + b ◦ b = a ◦ a + b ◦ b − 2|a||b| cos θ
−2(a ◦ b) = −2|a||b| cos θ
a ◦ b = |a||b| cos θ
Another useful result is that we can quickly test for the orthogonality of two vectors by checking if their dot product is zero.

3.1.9 Cross Product

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

$$\underline{a} \times \underline{b} = \begin{bmatrix} a_yb_z - a_zb_y \\ a_zb_x - a_xb_z \\ a_xb_y - a_yb_x \end{bmatrix}$$

This is often remembered using the mnemonic ‘xyzzy’. In this course we only consider the definition of the cross product in 3D. An important Computer Graphics application of the cross product is to determine a vector that is orthogonal to its two inputs. This vector is said to be normal to those inputs, and is written n in the following relationship (care: note the normalisation):
a × b = |a||b| sin θn
A proof is beyond the requirements of this course.

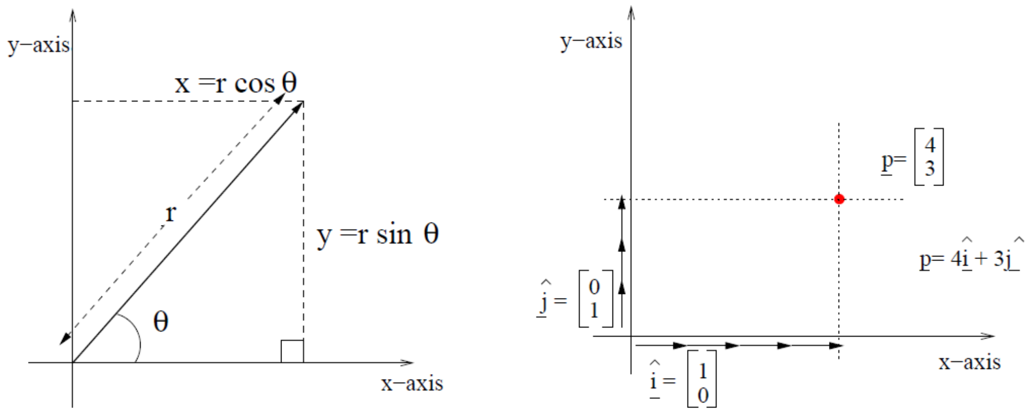


Figure3: Left: Converting between Cartesian (x, y) and radial-polar (r, θ) form. We treat the system as a right-angled triangle and apply trigonometry. Right: Cartesian coordinates are defined with respect to a reference frame. The reference frame is defined by basis vectors (one per axis) that specify how ‘far’ and in what direction the units of each coordinate will take us.

3.2 Matrix Algebra

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

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

Observe that the notation for addressing an individual element of a matrix is xrow,column.

3.2.1 Matrix Addition

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

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

This is identical to vector addition.

3.2.2 Matrix Scaling

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

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

3.2.3 Matrix Multiplication

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

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

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

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

Not all matrices are compatible for multiplication. In the above system, A must have as many columns as B has rows. Furthermore, matrix multiplication is non-commutative, which means that BA 6= AB, in general. Given equation 1.27 you might like to write out the multiplication for BA to satisfy yourself of this.
Finally, matrix multiplication is associative i.e.:
ABC = (AB)C = A(BC)
If the matrices being multiplied are of different (but compatible) sizes, then the complexity of evaluating such an expression varies according to the order of multiplication1.

3.3.4 Matrix Inverse and the Identity

The identity matrix I is a special matrix that behaves like the number 1 when multiplying scalars (i.e. it has no numerical effect):
IA = A
The identity matrix has zeroes everywhere except the leading diagonal which is set to 1,
e.g. the (2 × 2) identity matrix is:

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

1Finding an optimal permutation of multiplication order is a (solved) interesting optimization problem, but falls outside the scope of this course.

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

$AA^{+}-1 = A^{-1}A = I$

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

1.7.5 Matrix Transposition

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

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

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

21.0 Conclusion

The study of linear algebra and geometry (vector and matrices) reveals that computer graphics work with points and vectors defined in terms of some coordinate frame.

We also need to change coordinate representation of points and vectors, hence to transform between different coordinate frames.

22.0 Summary

This unit is has discussed the mathematical background in linear algebra. It has also explained how important vectors are in computer graphics.

We have learnt various vector arithmetic relating geometrical shapes to appropriate algebraic expressions. Amongst them are vector addition, dot product vector multiplication, mathematics of matrix like addition, multiplication, transpose, identity and others that are useful in graphics design.

23.0 Tutor Marked Assignment

1. Perform the following operations on the given vector:
   Subtract $-b = [-s,-t]^T$
   Add vectors $a = [u, v]^T$ and $b = [s, t]^T$

Define the Cartesian Coordinate system

2. Proof that a ◦ b = |a||b|cosθ
3. Consider the matrix A= $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and B= $\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
   a. Compute the inverse of A
   b. Compute the Transpose of B
   c. Perform multiplication AxB

24.0

25.0 **Refrences/Further reading:**

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Textbook Homepage:
http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPH CS/FOURTH_EDITION/

Module 2: Geometric Modeling
Unit 3 Curve and surface design

Table of Contents

Title
Page

26.0 Introduction:

Until now we have worked with flat entities such as lines and flat polygons. Though it fits well with graphics hardware and of course it is mathematically simple. But the world is not composed of flat entities as such we need curves and curved surfaces
It may only have need at the application level. This unit explores the different types of curves used in graphics design.

27.0 Objectives:

At the completion of this unit, we must have dealt with the following:
Introduce types of curves and surfaces

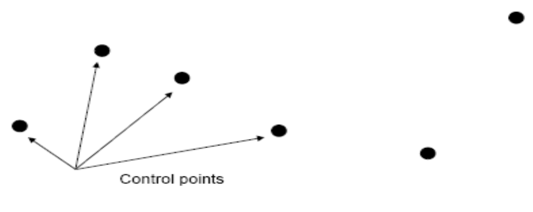
- Explicit
- Implicit
- Parametric
- Strengths and weaknesses

• Discuss surfaces and its usefulness.

28.0 Curves and Surfaces

3.1 Spline Curves

A spline curve is a smooth curve that is specified succinctly in terms of a few control points
A spline interpolates a control point if it passes through the control point



A set of control points

3.2 Representing Curves

Consider 3 basic representation of shape
Explicit functions
Implicit functions
Parametric representation

3.2.1 Explicit Representation

y = f(x)

Advantages

•Simple
•Easy to find points on the curve
•Simple to subdivide (consider equal intervals of x)

Disadvantages

•Impossible to get multiple y values for a single x value; can't represent circles and other closed forms with a single function
•Not rotationally invariant (rotation may require breaking the curve into multiple segments)

3.2.1 Implicit Representation

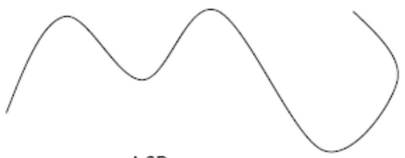
f(x,y,z) = 0

Advantages

•Can represent some shapes very simply
e.g., circle x2 + y2 = R2
•Affine invariant
i.e., rotation, translation scale invariant

Disadvantages

•Hard to define implicit functions for general shapes
•Hard to put constraints on the implicit functions
e.g., hard to require that the tangent vectors of two component curves match where they join up
Parametric Representation
Points on the curve are functions of a parameter t


A 2D curve

P(t) = f(t), t ∈ [0, 1], where

P(t) is a point (2D, 3D, etc.) on the curve
f(t) is a parametric function that represents the curve

3.2.3 Parametric representation

Note that there are many parameterizations for a given curve

e.g., if P(t) = f(t), t ∈ [0, 1] is one parameterization, then

P(q) = f(q), q = 2t, q ∈ [0, 0.5],

P(r) = f(r), r = 1 – t, r ∈ [0, 1],

P(s) = f(s), s = t2, s ∈ [0, 1],

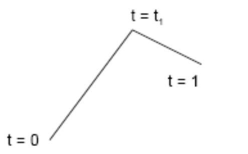
etc., are also parameterizations of the same curve

3.3 Piecewise parametric representation

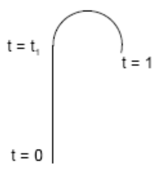
Complex curves can be subdivided into simpler pieces

$$P(t) = \begin{cases} \mathbf{f}_1(t),\ t \in [0, t_1] \\ \mathbf{f}_2(t),\ t \in [t_1, 1] \end{cases}$$

P(t) = f1(t), t ∈ [0, t1]



A piecewise curve made up of two connected line segments



A piecewise curve made up of a line segment and a circular arc

f2(t), t ∈ [t1, 1]

A piecewise polynomial curve

Has a parametric representation that is a polynomial function of t, i.e.,

$$P(t) = \sum_{i=0}^{N} w_i t^i,\ t \in [0, 1], \text{ where N is the degree of the polynomial curve}$$

Example

•A line segment is a piecewise linear curve, where N = 1

•A parametric representation of the line segment with endpoints P0 and P1 is

$P(t) = (1 – t)P_0 + tP_1$

•In polynomial form

$P(t) = P_0 + t (P_1 – P_0)$

Example

•Quadratic polynomials have N = 2 •The polynomial form of a quadratic polynomial is

$P(t) = at^2 + b_t + c$, where a, b, and c are the vectors

$a = (a_x, a_y, a_z)$

$b = (b_x, b_y, b_z)$

$c = (c_x, c_y, c_z)$

•i.e., the polynomial form of the point $P(t) = (x(t),y(t),z(t))$ is

$x(t) = a_x t^2 + b_x t + c_x$

$y(t) = a_y t^2 + b_y t + c_y$

$z(t) = a_z t^2 + bzt + c_z$

3.4 Tangent vector

The parametric tangent vector is the derivative of the parametric form with respect to the parameter t

•e.g., for quadratic polynomials $P(t) = at^2 + bt + c$

$X' = dx(t)/dt = 2a_x t + b_x$

$y' = dy(t)/dt = 2a_y t + b_y$

$z' = dz(t)/dt = 2a_z t + b_z$

•Using simplified notation $P'(t) = dP(t)/dt = 2at + b$

e.g., for cubic polynomials

$P(t) = at^3 + bt^2 + ct + d$

$x' = dx(t)/dt = 3a_x t^2 + 2b_x t + c_x$

$y' = dy(t)/dt = 3ayt^2 + 2b_y t + c_y$

$z' = dz(t)/dt = 3a_z t^2 + 2b_z t + c_z$

•Using simplified notation $P'(t) = dP(t)/dt = 3at^2 + 2bt + c$

3.5 Curve Continuity

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

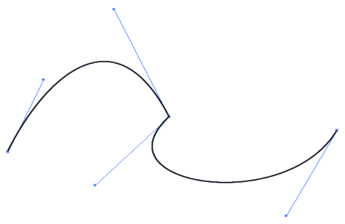
The continuity of a piecewise polynomial curve is determined by how the individual curves are joined.

Each polynomial curve is smooth (infinitely differentiable) along its length

The quality of a curve is characterized by the continuity at points where the piecewise polynomial curves are joined together

3.5.1 C^0, G^0 continuity

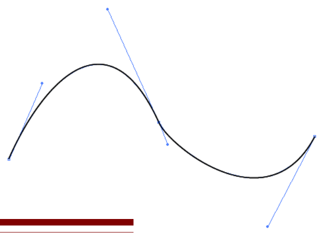
–A curve is C^0 and G^0 continuous if adjacent segments join at a common endpoint.



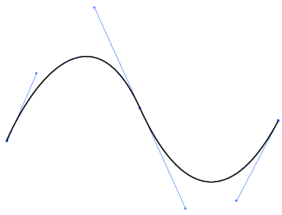
3.5.2 G^1 continuity

–A curve is G^1 continuous if the parametric first derivative is continuous across its joints

•i.e., the tangent vectors of adjacent segments are collinear (i.e., the tangent vectors are on the same line) at the shared endpoint



A curve is C^1 continuous if the spatial first derivative is continuous across joints
•i.e., tangent vectors of adjacent segments are collinear and have the same magnitude at their shared endpoint

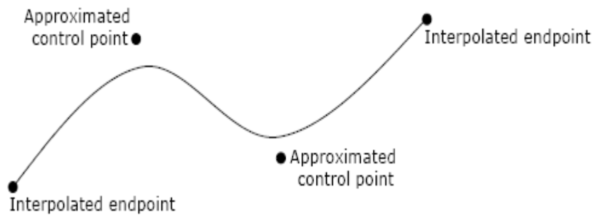


3.5.3 C^N continuity

The curve is C^N continuous if the nth derivatives of adjacent segments are collinear and have the same magnitude at their shared endpoint
Curve continuity has a significant impact on the quality of a curve or surface
Different industries have different standards
•Computer Graphics often requires G^1 continuity
'Good enough' for animations and games
•The automotive industry often requires G^2 continuity
Visually appealing surface reflections off of car bodies
•Aircraft and race cars may require G^4 or G^5 continuity
Avoid turbulence when air flows over the surface of the vehicle

3.4 Bezier Curves

Constraints for Bezier curves
Bezier curves are constrained by a set of control points
•One more control point than the order of the curve
•Interpolate the control points specifying the endpoints of the curve
•Approximate the other control points



A 3rd order Bezier curve has 4 control points. The 2 endpoints are interpolated.
Constraints for Bezier curves
Bezier curves are constructed by determining blending functions Bi(t) that satisfy the constraints
•Similar to Hermite curves

$$P(t) = \sum_{i=0}^{N} B_i(t)P_i$$

The constraints are the positions of the control points
Blending functions

$$P(t) = \sum^{N} B_i(t)P_i$$

To satisfy the constraints, the blending functions must force the curve to interpolate the endpoints.
•At $t = 0$, $B_0(t) = 1$ and all other $B_i(t) = 0$
•At $t = 1$, $B_N(t) = 1$ and all other $B_i(t) = 0$
The other two constraints will control how much the curve wiggles between the endpoints

29.0 Conclusion

Smooth curves and surfaces must be generated in many computer graphics applications.
Many real-world objects are inherently smooth, and much of computer graphics involves
modelling the real world.

30.0 Summary

In this unit, we have learnt the following:
Definitions and properties of polynomial curves, splines, B-splines, surfaces of revolution. Definitions of C and G continuity and recognize differences visually. Derived analytic expressions for polynomial curves and spline from constraints indicating locations, tangents, and continuity. Evaluated Bezier and B-splines with geometric construction. Displayed polynomial curves and splines using line segments.

31.0 Tutor Marked Assignment

1. What are the meanings of each of the following forms of continuity:C0:, C1:, C2:,C∞.
2. Ahead of each type of point, mark what kind of continuity it has (none, C0, C1, C2, C3, C∞).

An interior point of a Bezier curve.
A non-joint point on a uniform cubic B-spline.
A joint of a uniform cubic B-spline.
A joint of a cubic Catmull-Rom spline.

3. State two advantages of splines (piecewise low-order Beziers) over high-order Bezier curves.

32.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.
J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.
D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.
P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 2: Geometric Modeling
Unit 4 Ray Tracing

Table of Contents

1.0 Introduction

Ray tracing is a rendering technique that is, in large part, a simulation of geometric optics. It does this by first creating a mathematical representation of all the objects, materials, and lights in a scene, and then shoots infinitesimally thin rays of light from the viewpoint through the image plane into the scene. The rays are then tested against all objects in the scene to determine their intersections. In a simple ray tracer, a single ray is shot through each pixel on the view plane, and the nearest object that the ray hits is determined. The pixel is then shaded

according to the way the object's surface reflects the light. If the ray does not hit any object, the pixel is shaded with the background colour.
Ray Tracing was developed as one approach to modeling the properties of global illumination.
The basic idea is as follows:
For each pixel:
Cast a ray from the eye of the camera through the pixel, and find the first surface hit by the ray.
Determine the surface radiance at the surface intersection with a combination of local and global models.
To estimate the global component, cast rays from the surface point to possible incident directions to determine how much light comes from each direction. This leads to a recursive form for tracing paths of light backwards from the surface to the light sources.
Computational Issues
• Form rays.
• Find ray intersections with objects.
• Find closest object intersections.
• Find surface normals at object intersection.
• Evaluate reflectance models at the intersection.
What is the best way to map the scene to the display?

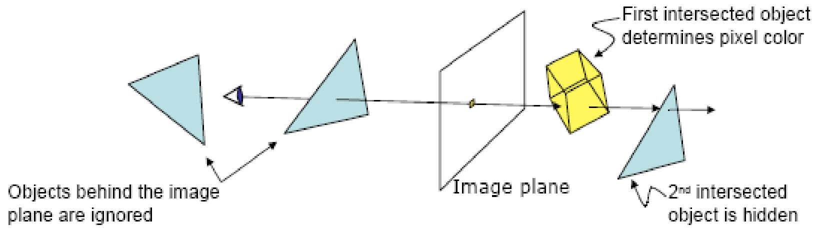
2.0 Objectives:

This unit discuses about the concept of ray tracing by showing various cases of ray-object intersection, shadow rays, reflected rays and transmitted rays.

3.0 Ray casting

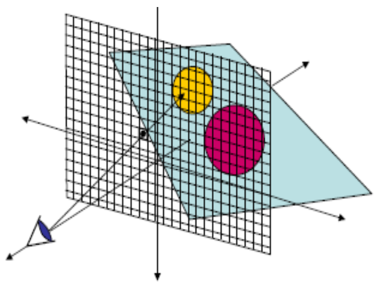
Consider each element of the view window one at a time (i.e., each image pixel)
Test all of the objects in the scene to determine which one affects the image pixel
Determine the color of the image pixel from the appropriate object
The goal of ray casting is to determine the color of each pixel in the view window by considering all of the objects in the scene
•What part of the scene affects a single pixel?
•For a single pixel, see a finite volume of the scene
Limited by the pixel boundaries
Difficult to compute contribution over the finite volume
•For a single pixel, see a finite volume of the scene
Approximate the total contribution by sampling a single view direction through the center of the pixel
•To approximate the color of a single image pixel
Sample the scene by shooting a ray through the center of the pixel
•If the ray intersects at least one object
Set the pixel color from the first object intersected by the ray
•If the ray does not intersect anything

Set the pixel color from the background color
•Ignore
Objects behind the camera
Objects between the camera and the image plane


First intersected object determines pixel color
Objects behind the image plane are ignored
Image plane
2nd intersected object is hidden

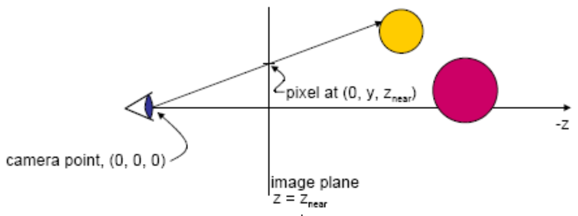
Basic ray casting algorithm

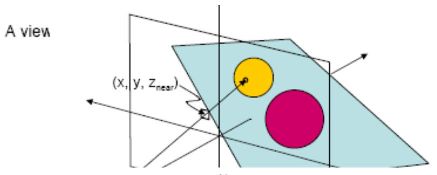
```
For each image pixel (i, j)
{
Determine the viewing ray from the eye through the pixel
Find the first object beyond the image intersected by the ray
Set the pixel color from the object color
}
```


The viewing ray for pixel (i, j) passes through the center of (i, j)

Determining the viewing ray
The viewing ray of a given pixel starts at the camera point and passes through the pixel center into the scene

3.1 Determining the viewing ray in camera space





The camera is at the origin and looks down the negative z-axis
First, consider a ray in the y-z plane (i.e., x = 0)
•The ray passes through the two points (0, 0, 0) and $(0, y, z_{near})$
The ray passes through (0, 0, 0) and $(0, y, z_{near})$
•In parametric form, the ray is
$(x(t), y(t), z(t)) = (1-t)\,(0, 0, 0) + t\,(0, y, z_{near})$
$= t\,(0, y, z_{near})$, where $t > 1$ beyond the image
A general ray passes through (0, 0, 0) and (x, y, z_{near})
•In parametric form, the ray is
$(x(t), y(t), z(t)) = (1-t)\,(0, 0, 0) + t\,(x, y, z_{near})$
$= t\,(x, y, z_{near})$, where $t > 1$ beyond the image
Given a viewport ranging from (x_{left}, y_{bot}) to (x_{right}, y_{top})
Render the viewport into an image of size W x H

3.2 Mapping image coordinates to the viewport

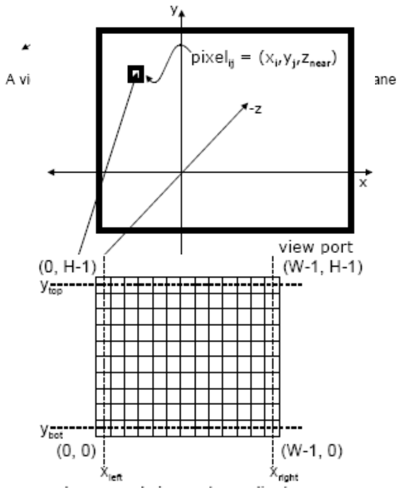


Image and viewport coordinates

Map the image boundaries to the viewport boundaries
$(0, 0) \rightarrow (x_{left}, y_{bot})$ and
$(W-1, H-1) \rightarrow (x_{right}, y_{top})$,
Thus,
$x_i = x_{left} + i(x_{right} - x_{left}) / (W-1)$
$y_j = y_{bot} + j(y_{top} - y_{bot}) / (H-1)$

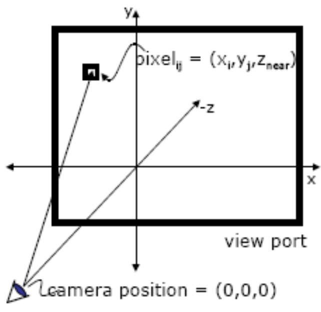
3.3 Determining the viewing ray in camera space

Given a viewport from (x_{left}, y_{bot}) to (x_{right}, y_{top}) and an image of size W x H

•Pixel (i, j) corresponds to viewport position (xi, yj, z_{near}), where

xi = $x_{left} + i(x_{righ}t - x_{left}) / (W-1)$

yj = $y_{bot} + j(y_{top} - y_{bot}) / (H-1)$


y
pixel$_{ij}$ = (x_i,y_j,z_{near})
-z
x
view port
camera position = (0,0,0)

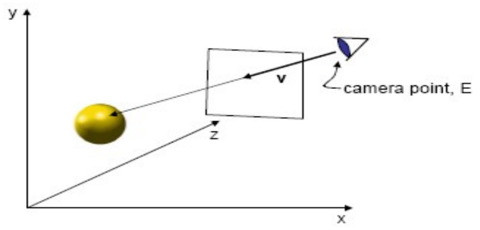
3.4 Determining the viewing ray in world coordinates

Viewing rays are particularly easy to compute in camera space

$(x(t), y(t), z(t)) = t * (x_i, y_j, z_{near})$

However, the approach requires that we represent the scene in camera coordinates rather than world coordinates. This is inconvenient if we want to move the camera about the scene.

One solution is to determine the viewing ray in world coordinates



Given a camera is specified in world coordinates, we want to determine a parametric expression for the ray from the eye through a given pixel

The ray has the form

(x(t), y(t), z(t)) = E + tv, where

E is the camera position (i.e., the eye point) in world coordinates v is the vector from E to the image point (i, j) transformed into world coordinates. Use the look-at point camera specification for defining E and v in world coordinates E is the camera position (px, py, pz). v can be expressed in terms of the camera space axis vectors ux', uy', and uz'

$u'_z = (p_x - l_x, p_y - ly, p_z - l_z) / ((p_x - lx)^2 + (p_y - ly)^2 + (p_z - l_z)2)^{½}$

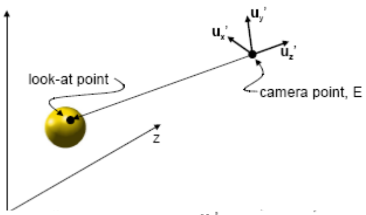
$u'_x = ((v_x, v_y, v_z) \times u'_z) / \|(v_x, v_y, v_z) \times u'_z\|$

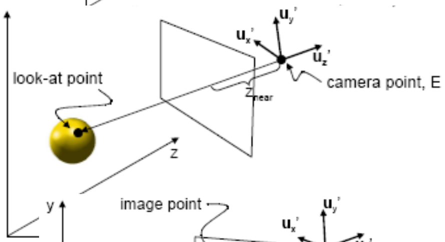
$u'_y = u'_z \times u'_x$

(x(t), y(t), z(t)) = E + tv, where

E is the camera position (p_x, p_y, p_z)

v can be expressed in terms of the u_x', u_y', and u_z'





Note that

•The camera looks in the direction of -u_z

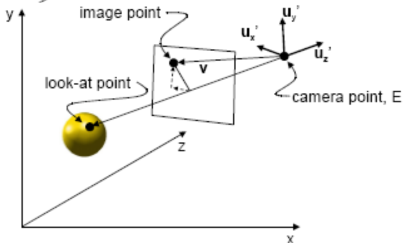
•The image plane is perpendicular to u_z

•The image plane is defined by u_x', u_y' and z_{near}

•Rows are parallel to ux', columns are parallel to u_y'

Using projective geometry, $v = x_i u_x' + y_j u_y' + z_{near} u_z'$, with

$x_i = x_{left} + i(x_{right} - x_{left}) / (W-1)$



$y_j = y_{bottom} + j(y_{top} - y_{bottom}) / (H-1)$

3.5 Finding objects intersected by the ray

Given a (general) ray E + tv, we want to find the first intersection between the ray and any object where t > 1. Need to be able to find intersections between an arbitrary ray and some common geometric shapes•e.g., sphere, plane, triangle.

Intersection tests

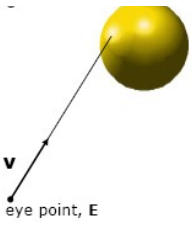
Specify the ray using a parametric expression, e.g., E + tv

Specify an implicit representation of the object, e.g., for a sphere,

$f(x, y, z) = (x - xc)^2 + (y - c)^2 + (z - zc)^2 - R^2$

Solve for the positive values of t so that E + tv satisfies the implicit equation f(x, y, z) = 0

Intersection between a ray and a sphere



Ray

•Specified by the parametric expression E + tv = E + tv

Sphere centered at (x_c, y_c, z_c)

•Specify the sphere by the implicit equation

$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$

Equations

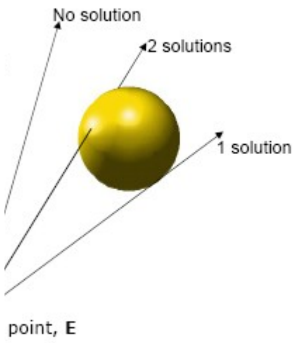
View ray: $E + tv = (e_x, e_y, e_z) + t\,(v_x, v_y, v_z)$

Implicit sphere: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$

Substitute the expression for the ray into the implicit equation

$(e_x + tv_x - x_c)^2 + (e_y + tv_y - y_c)^2 + (e_z + tv_z - z_c)^2 - R^2 = 0$

Expand and express as a function in t



$t^2(vx^2+vy^2+vz^2) + 2t(vx(ex-xc) + vy(ey-yc) + vz(ez-zc)) +$
$(e_x-x_c)^2 + (e_y-y_c)^2 + (e_z-z_c)^2 - R^2 = 0$

After expanding and rearranging terms, This is a quadratic expression in t of the form

$at^2 + bt + c = 0$

Intersection test yields a quadratic expression in t

$at^2 + bt + c = 0$

•This expression can have 0, 1, or 2 solutions

$t = (-b +/- \sqrt{(b^2 - 4ac)}) / 2a$

No solution if $b^2 - 4ac < 0 \rightarrow$ no intersection

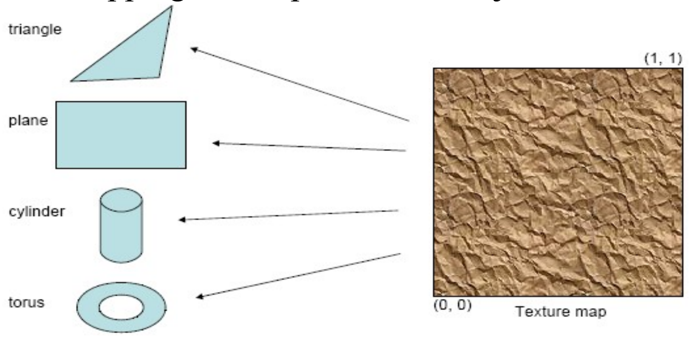
One solution if $b^2 - 4ac = 0 \rightarrow$ray grazes sphere

Two solutions if $b^2 - 4ac > 0 \rightarrow$ ray enters and n exits the sphere, choose the smallest t which is greater than 1

3.6 Texture

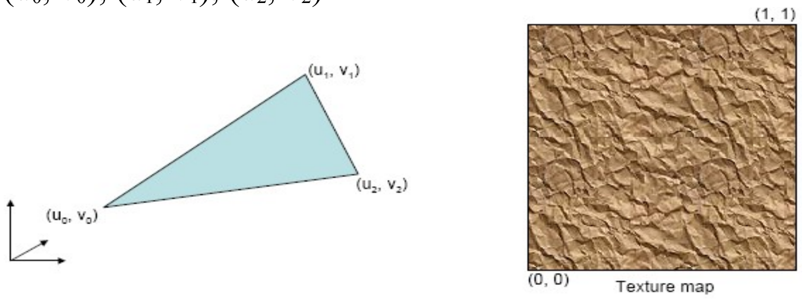
Texture can be used to modulate diffuse and ambient reflection coefficients, as with Gouraud shading. We simply need a way to map each point on the surface to a point in texture space, e.g. given an intersection point p(λ*), convert into parametric form s(α, β) and use (α, β) to find texture coordinates (μ, ν). Unlike Gouraud shading, we don't need to interpolate (μ, ν) over polygons. We get a new (μ, ν) for each intersection point.

The mapping will depend on the object

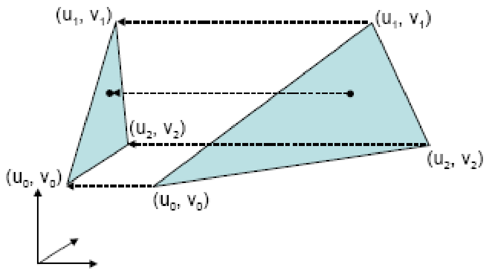


3D triangle

Specify the texture coordinates at the triangle's 3 vertices
(u_0, v_0), (u_1, v_1), (u_2, v_2)



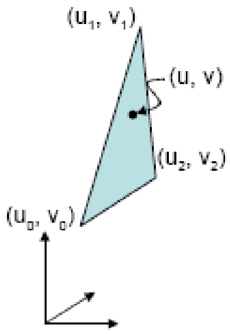
Project the triangle vertices and the intersection point onto a convenient 2D plane using orthographic projection. e.g., project onto the xy-plane



by dropping the z-coordinate.

Interpolate the texture coordinates at the ray-triangle intersection point in the projected triangle

e.g., use Pineda's method to perform both the inside/outside test to see if the ray intersects the triangle and to interpolate the texture coordinate



3.7 Inter-Object Illumination

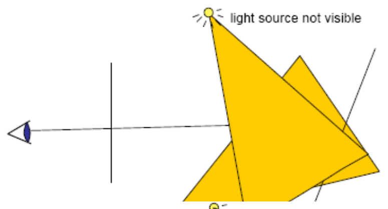
In reality, the color at the intersection point depends on

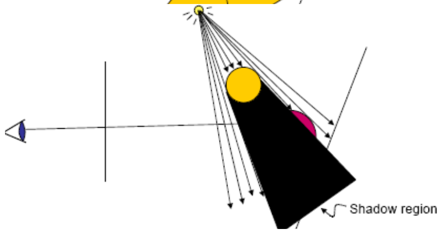
•The object's color or texture color
•Illumination from light sources (Phong illumination)
•Objects occluding the light sources
•Light from other objects that is reflected in the direction of the eye
•Light that is transmitted through the object in the direction of the eye

3.8 Shadows

A simple way to include some global effects with minimal work is to turn off local reflection when the surface point p cannot see light sources, i.e. when p is in shadow.

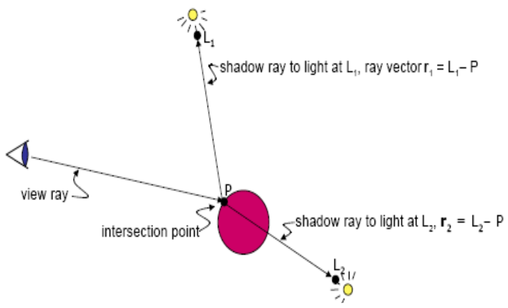
A light source only contributes to the color if it is visible from the intersection point





How do we know if an object is in shadow?
A point is in shadow if there is no direct path to the light source
Trace a ray from the intersection point to each light, if the ray does not intersect an object, the light is visible

To determine if a light is visible, test to see if the shadow ray from the intersection point to the light intersects any object in the scene


L1
shadow ray to light at L1, ray vector r1 = L1 – P
view ray
P
intersection point
shadow ray to light at L2, r2 = L2 – P
L2

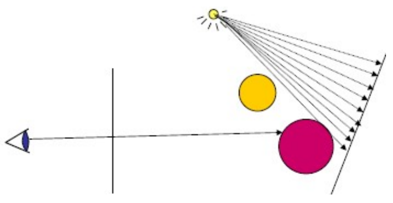
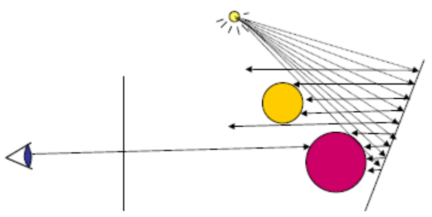
Augment the Phong illumination model:

$$I = k_a I_a + \square_i (f_{att} \underbrace{S_i}_{\text{Shadow factor for light } L_i} (k_d I_{L_i} (\mathbf{n} \cdot \mathbf{l}) + k_s I_{L_i} (\mathbf{r} \cdot \mathbf{v})^s)), \text{ where}$$

Si = 0 if the light Li is occluded by an object in the scene
Si = 1 if the light Li is not occluded
Si | (0, 1) if there is a transparent object between the intersection point and the light L1
Total illumination
The color of a particular point is determined by
The base color/texture of the object
Illumination directly from light sources (Phong illumination model)
Illumination from other objects in the scene (direct reflection).

3.9 Reflection

•How do we model illumination due to light reflected off of other surfaces?

(A)Light leaves the light source, (B)Light leaves the light source, is reflected off the back wall

How do we model illumination due to light reflected off of other surfaces?

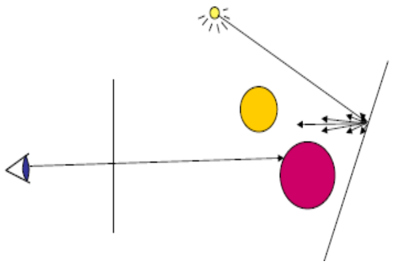
•Tracing rays from the light source is expensive as Lots of the reflected rays do not affect the image

Direct reflection

•Recall that reflection off most surfaces is not perfect

Light is reflected in multiple directions.

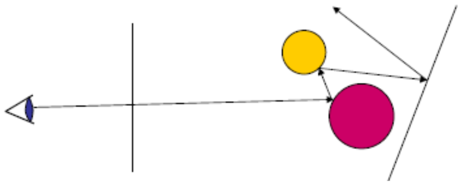
Distribution depends on the material and the cosine of the angle from the perfect reflection direction; $I_r = k_s (r \cdot v)^s$



For most surfaces, light is reflected in multiple directions.

Assume that the amount of secondary light reaching the intersection point after reflection off of another object is small unless was reflected in the perfect reflection direction

Determine the contribution from other objects by looking backwards in the direction of perfect reflection. Trace a ray backwards from the intersection point in this direction to see what might contribute to the reflected light.



Contributions from other objects in the scene are determined using reflection rays

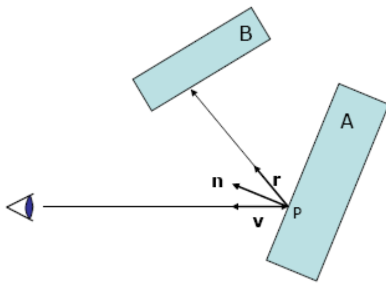
The local color at P is determined using Phong illumination

The reflected color at P is light from other objects (e.g., B) that is reflected in the direction of the eye. The color from B is determined by the local color at P’ and light from other objects reflected in the direction of P

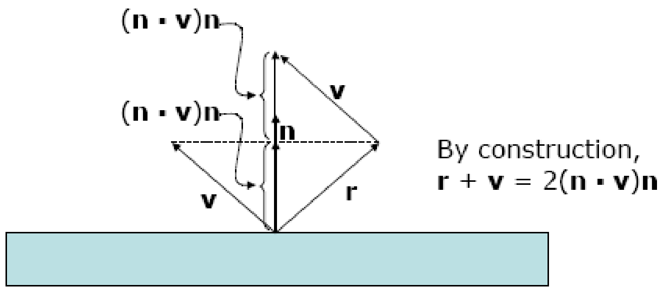
Note that the reflected color is usually much smaller than the local color

To compute the reflection contribution at P, trace ray r backwards until it intersects another object

•The reflection ray is in the direction of perfect reflection, r



By construction, r = 2(n ▪ v)n – v, where, r, n, v are all unit vectors with directions as shown below



n, r, and v are the unit surface normal, reflection vector, and vector to the eye respectively.

Ray Tracing – Refraction

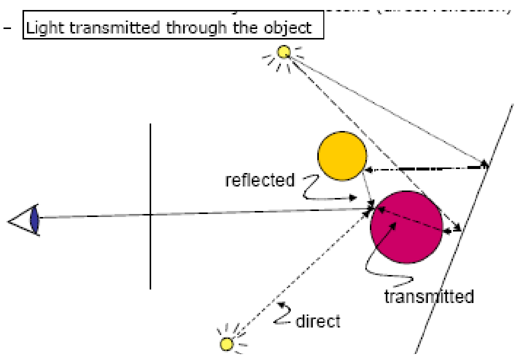
3.10 Total illumination

•The color of a particular point is determined by

i. The base color/texture of the object

ii. llumination directly from light sources (Phong illumination model)

iii. Illumination from other objects in the scene (direct reflection)

– 

3.11 Refraction

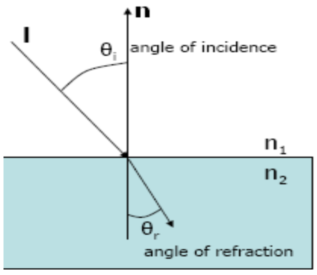
Light travels at different speeds through different media

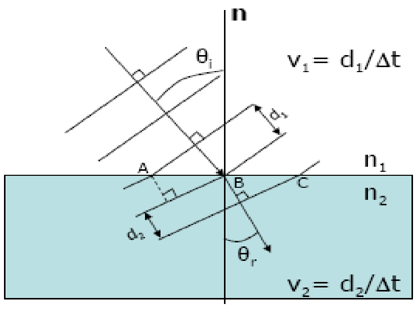
The speed of light in a material is inversely proportional to the material's index of refraction, n

n = (speed of light in vacuum) / (speed of light in material)

For example n_{air}= 1.003, n_{water} = 1.33, n_{glass}= 1.5, $n_{diamond}$= 2.5

When light crosses a surface between materials that have different indices of refraction, it changes direction





Light travels at different speeds in different materials

Light changes speed when the wave front hits a new material

•When it hits at an angle, the direction of the wave front changes

Transmission – angle of refraction

Snell's Law:

$\sin\theta_r = (n_1/n_2) \sin\theta_i$

4.0 Conclusion

This unit illustrates the concept of ray tracing by showing various cases of ray-object intersection, shadow rays, reflected rays and transmitted rays.

5.0 Summary

In this unit, we see how rays follow (trace) the path of a ray of light and model how it interacts with the scene. When a ray intersects an object, send off secondary rays (reflection, shadow, transmission) and determine how they interact with the scene.

We also discovered that Ray tracing can handle shadows, specular reflection, texture mapping, depth of field and other optical effects. The rays do not have to stop when they hit an object they can be reflected from the surfaces of reflective objects, and transmitted through transparent objects. A recursive ray tracer can accurately model and render optical effects such as reflections, transparency, translucency, caustics, dispersion, global illumination, and other effects.

6.0 Tutor Marked Assignment

1. Write and explain the process of mapping image coordinates to the viewport
2. Briefly describe the Basic ray casting algorithms that you know.
3. What do you understand by Ray casting
4. Why do rays bend? What are the factors responsible for this?

7.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 2: Geometric Modeling

Unit 5 Texture Mapping

Table of Contents

Title

Page

33.0 Introduction:

A texture is a digital image in that has to be mapped onto the surface of a polygon as it is rendered into the colour buffer. There are several issues involved:

Is the format of the texture the same as the colour buffer?

If no, convert image to the colour buffers format before use

Are the dimensions of the texture the same as the colour buffer?

No, polygons are 2D objects, textures can have 1, 2, 3 or 4 dimensions

How is the image stored?

The image is a bitmap, consisting of texels

How is the image mapped onto the polygon? Most of this lecture is devoted to that topic

34.0 Objectives:

The major objective is to introduce Mapping Methods

Texture Mapping

- Environmental Mapping
- Bump Mapping

And also to Consider basic strategies

- Forward vs backward mapping
- Point sampling vs area averaging

35.0 Mapping Functions

Basic problem is how to find the maps

Consider mapping from texture coordinates to a point a surface

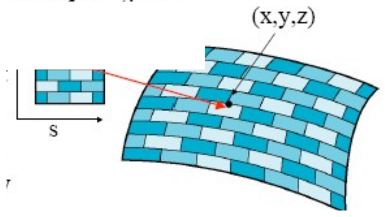
Appear to need three functions

x = x(s,t)

y = y(s,t)

z = z(s,t)

But we really want to go the other way



3.1 Backward Mapping

Given a pixel, we want to know to which point on an object it corresponds

Given a point on an object, we want to know to which point in the texture it corresponds

As such we need a map of the form

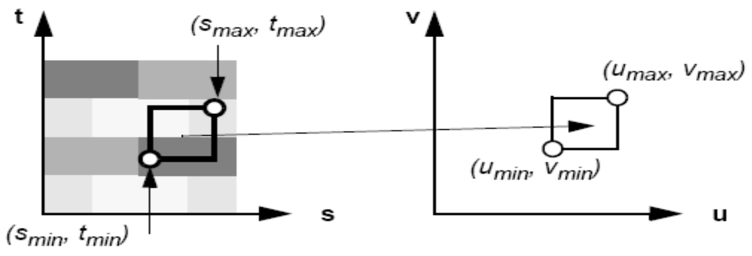
s = s(x,y,z)

t = t(x,y,z)

Such functions are difficult to find in general.

Texture space is indexed by coordinates (s,t) which are normalised to the range $0 \le s \le 1$ and $0 \le t \le 1$

Consider a rectangular polygon in model space, indexed by (u,v):



The mapping between the two spaces is defined parametrically in terms of the maximum and minimum coordinates:

$$u = u_{min} + \frac{s - s_{min}}{s_{max} - s_{min}}(u_{max} - u_{min})$$

3.2 Two-part Mapping

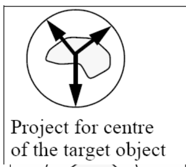
An alternative is to texture a whole object a one go using a projection from a simpler intermediate object

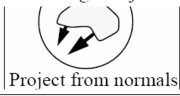
This is a ***2 part mapping***:

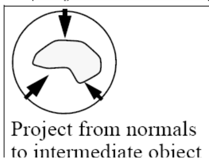
1. Map texture onto regular surface, e.g.:Sphere, Cylinder,Box
2. Project texture onto object:

Normal from intermediate, Normal to intermediate, Projector from centre of object

All we have to do is define the mappings onto the intermediate object:

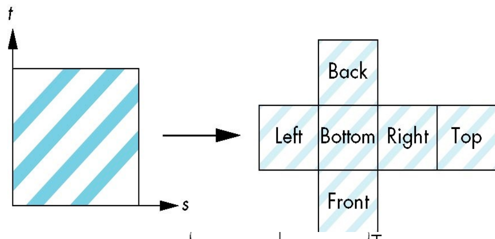

Project for centre of the target object


Project from normals


Project from normals to intermediate object

3.3 Box Mapping

Map to each face: Easy to use with simple orthographic projection. Also used in environmental maps

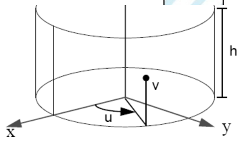


Cylinder: of radius r and height h:

x = r cos($2\pi v$)

y = r sin($2\pi v$)

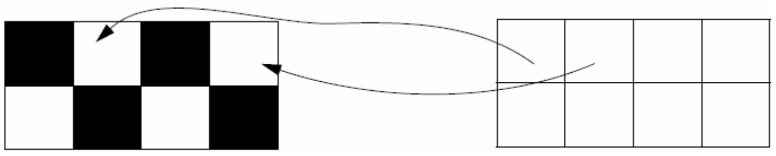
z = vh



Sphere: use any one of the many cartographic projections used by mapmakers – there will be distortions of the texture near the poles.

3.4 Resolution Issues

One potential issue in texture mapping is the relationship between the resolution of the texture and sampling frequency



This can be exacerbated by perspective effects. The solution is to use ***mipmapping***

This controls the level of detail by generating a pyramid of textures with each level having half the resolution of the level below and using them where appropriate

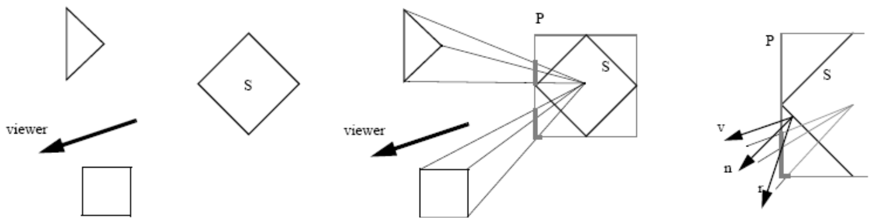
3.5 Illumination of Textured Surfaces

We have a final decision to make:
During scan conversion, how should a texel be treated with regards to illumination
Two options:
1. If the polygon is being shaded should the illumination model be applied to the texture?
The texture will not appear the same from all angles under all lights, but rather the texture will be shaded using the same calculations as would be applied to material properties of the polygon
2. Should the texture replace whatever underlying colour the polygons material properties have defined or be blended with it?
This decides whether we see only texture or a mixture of texture and material

3.6 Environment Mapping

We can simulate the appearance of a reflective object in a complex environment without using ray tracing. This is called ***environment or reflection mapping***
We position a viewpoint within the object looking out, then use the resulting image as a texture to apply to the object

- Replace reflective object S by projection surface P
- Compute image of environment on P and project image from P to S



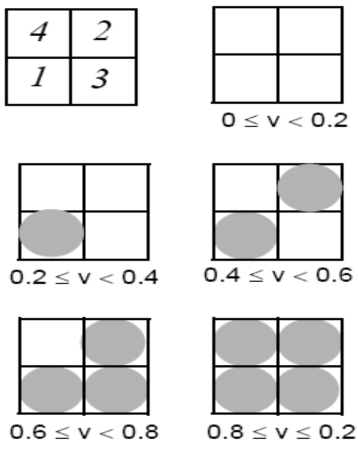
Pixel Operations

Besides texturing, there are other pixel operations available:
Fog: blend pixels with fog colour with blending governed by Z coordinate
Antialiasing: replace pixels by the average of their own and their nearest neighbours colours
Colour balancing: modify colours as they are written into the colour buffer.
Direct manipulation: copy or replace pixels As well as the colour and depth buffers OpenGL provides:
A ***stencil buffer*** used for masking areas of other buffers
An ***accumulation buffer*** used for whatever you want
A ***bitwise XOR operator*** is provided usually hardwired in the graphics chip.

Halftones

The idea of ***halftoning*** is to increase the apparent number of available intensities

The trade-off is a loss of spatial resolution. Rectangular pixel regions are called halftone patterns:



- n2 pixel grid gives n2 intensities
- so 4 by 4 block has 17 shades from white to black
- level k → turn on pixels numbered ≤ k
- generalises to colour

Pattern generation is not trivial:

- Sub-grid patterns become evident
- Visual effects, e.g. contouring, to be avoided
- Isolated pixels not effective on some devices

Half toning is used by newspapers to print shaded photographs using only black ink:

3.6 Dithering

Dithering uses the same principle that halftoning uses in printing, but the output medium is pixels of a fixed size. Each "pixel" is represented by a block of pixels – the dither matrix

3.7 Bump Maps

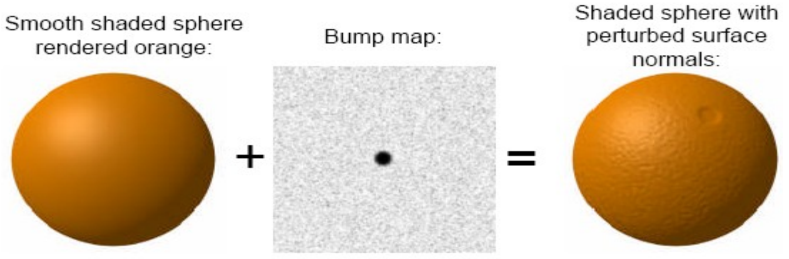
Bump Maps are used to capture fine-scale surface detail or roughness:

- Apply perturbation function to surface normal
- Use perturbed normal in lighting calculations

Elements from the bump map are mapped to a polygon in exactly the same way as a surface texture, but they are interpreted as a perturbation to the surface normal, which in turn affects the rendered intensity. The bump map may contain:

- Random patterns
- Regular patterns
- Surface detail

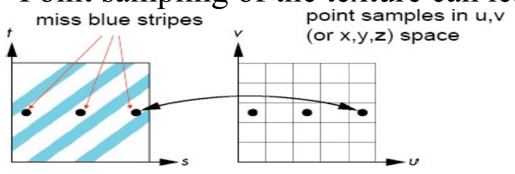
Example



Smooth shaded sphere rendered orange:
Bump map:
Shaded sphere with perturbed surface normals:
+
=

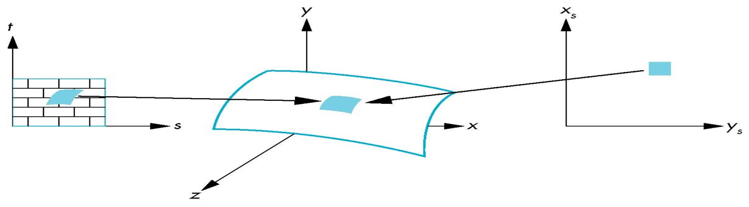
3.8 Aliasing

• Point sampling of the texture can lead to aliasing errors



3.9 Area Averaging

A better but slower option is to use area averaging



36.0 Conclusion

Texture is one important feature to visualize a surface. Textures have been studied by researchers in computer graphics, computer vision and cognitive psychology. Sometimes the term pexel is used for perceptual texture to emphasize the perceptual aspects of textures.

37.0 Summary

In this unit, we have learnt seen texture a digital image in that has to be mapped onto the surface of a polygon as it is rendered into the colour buffer alongside issues involved in the mapping process. We also discussed the various Mapping Methods ; environmental mapping and the Bump Mapping. And also to Consider basic strategies such as forward and backward mapping,
Point sampling and area averaging.

38.0 Tutor Marked Assignment

1. When warping an image, we use either forward mapping or reverse mapping.
   Describe two problems with forward mapping
2. What is aliasing?
3. Why is the sinc filter useful for avoiding aliasing in image processing applications?
4. Why don't we use the sinc filter in practice in image processing applications?

39.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 3: 3D Graphics Rendering
Unit 1 Transformation

Table of Contents

Title **Page**

42.0 Introduction:

In Computer Graphics we most commonly model objects using points, i.e. locations in 2D or
3D space. For example, we can model a 2D shape as a polygon whose vertices are points. By manipulating the points, we can define the shape of an object, or move it around in space.
In 3D too, we can model a shape using points. Points might define the locations (perhaps the corners) of surfaces in space. In this unit, we will describe how to manipulate models of objects and display them on the screen.

43.0 Objectives:

On completing this unit, we will determine how a scene is mapped to a particular computer screen. We will also learn about Projection transform that:

- Specifies how a 3D scene is mapped to the 2D screen
- Specifies the shape of the view volume Viewport (i.e., image) transformation

We will also apply appropriate transformation matrices to primitive graphics shape data structures:

- rotate 2D and 3D shapes
- translate 2D and 3D shapes
- scale 2D and 3D shapes.

3.0 Transformation

Transformations are often considered to be one of the hardest concepts in elementary computer graphics. But transformations are straightforward, as long as you

•Have a clear representation of the geometry
•Understand the underlying mathematics
•Are systematic about concatenating transformations

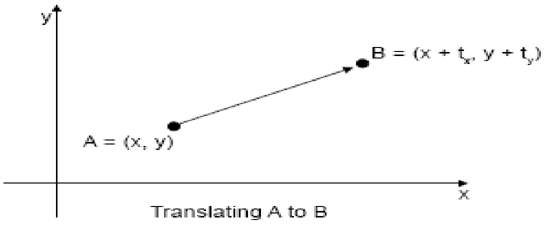
Given a point cloud, polygon, or sampled parametric curve, we can use transformations for several purposes:

1. Change coordinate frames (world, window, viewport, device, etc).
2. Compose objects of simple parts with local scale/position/orientation of one part defined with regard to other parts. For example, for articulated objects.
3. Use deformation to create new shapes.

3.1 Translation

Suppose we want to move a point from A to B e.g, the vertex of a polygon. This operation is called a translation

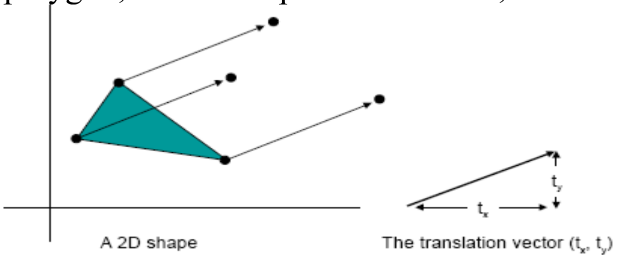
To translate point A by (t_x, t_y), we add (t_x, t_y) to A's coordinates



Translating A to B

To translate a 2D shape by (t_x, t_y) …

•Translate each point that defines the shape e.g, each vertex of a polygon, the center point of a circle, the control points of a curve

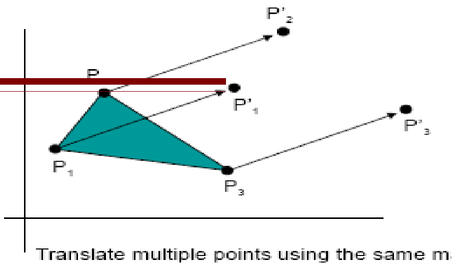


A 2D shape The translation vector (t_x, t_y)

Translation by (t_x, t_y) moves each object point by (t_x, t_y)

$(x, y) \rightarrow (x + t_x, y + t_y)$

Translation is a linear operation. The new coordinates are a linear combination of the previous coordinates, the new coordinates are determined from a linear system



Translate multiple points using the same matrix

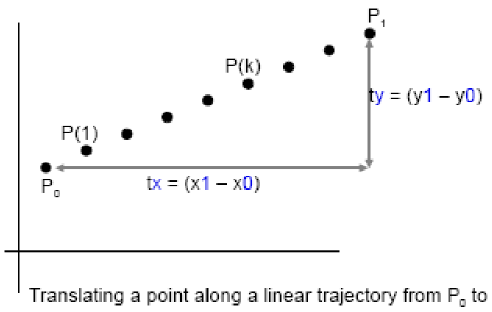
$x' = x + t_x$
$y' = y + t_y$
Hence, translations can be expressed using matrix notation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

for $(x, y) = P_1$, P_2, and P_3

Using matrices to represent transformations is convenient e.g., if the transformation is part of an animation

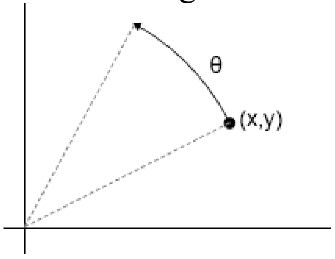
$$\begin{pmatrix} x(k) \\ y(k) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + (k/N) \begin{pmatrix} t_x \\ t_y \end{pmatrix} \text{, for } k = 0, 1, 2, \ldots N$$



Translating a point along a linear trajectory from P_0 to P_1

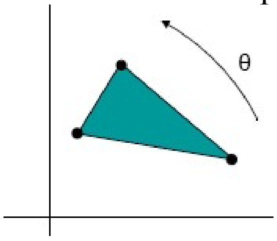
3.2 Rotation

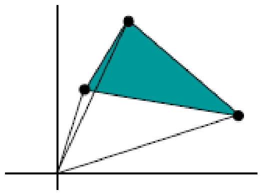
Suppose we want to rotate a point about the origin



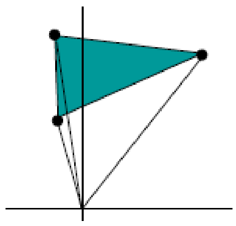
Rotate (x, y) about the origin by θ

To rotate a 2D shape defined by a set of points





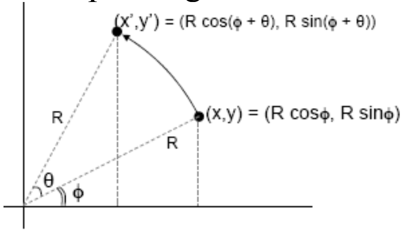
Rotate each point



and redraw the shape.

3.2.1 How do we define a 2D rotation?

First, represent (x, y) and (x’, y’) in polar coordinates and compute the corresponding cartesian coordinates



Represent the point and its destination in polar coordinates

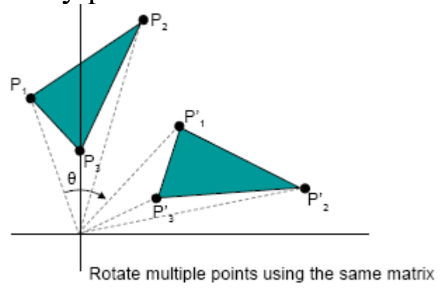
This is a Cartesian coordinates as functions of polar coordinates
(x, y) = (R cosφ,□R sinφ), and
(x’, y’) = (R cos(φ□+□θ),□R sin(φ□+□θ))
Use trig identities

$\cos(\varphi\square+\square\theta) = \cos\varphi\square\cos\theta\square-\square\sin\varphi\square\sin\theta$
$\sin(\varphi\square+\square\theta) = \cos\varphi\square\sin\theta\square+\square\sin\varphi\square\cos\theta$
Thus,
$x' = x \cos\theta\square-\square y \sin\theta$
$y' = x \cos\theta\square+\square y \sin\theta$
Rotation by θ □moves each object point according to $(x, y) \rightarrow (x \cos\theta\square-\square y \sin\theta,\ x \cos\theta\square+\square y \sin\theta)$
Rotation is a linear operation (like translation). The new coordinates are a linear combination of the previous coordinates and the new coordinates are determined from a linear system
$x' = x \cos\theta\square-\square y \sin\theta$
$y' = x \cos\theta\square+\square y \sin\theta$
Hence, rotations can be expressed using matrix notation

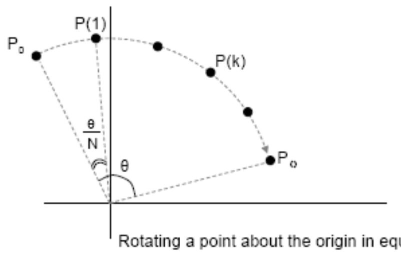
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(k\theta/N) & -\sin(k\theta/N) \\ \sin(k\theta/N) & \cos(k\theta/N) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \text{ for } k = 0, 1, 2, \dots N$$

Using matrices to represent transformations is convenient if you have many points to transform.



Rotate multiple points using the same matrix

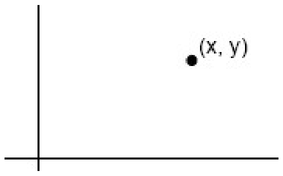


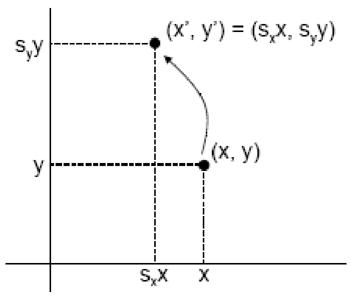
Rotating a point about the origin in equal angular increments

Or if the transformation is part of an animation

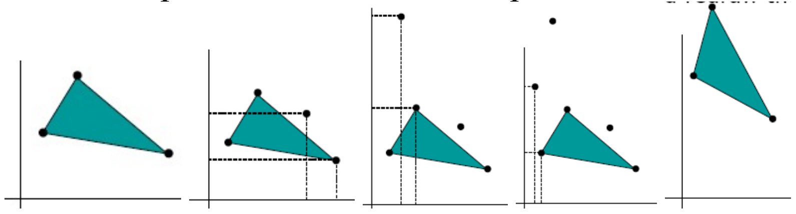
3.3 Scaling

Scale by (s_x, s_y) about the origin
–Scale the coordinates (x,y) by (s_x, s_y)





Otherwise, the scale is non-uniform.
To scale an object about the origin by (s_x, s_y), Scale each point, and redraw the shape.



Scaling by (s_x, s_y) scales each coordinate point
$(x, y) \rightarrow (s_x x, s_y y)$
Scaling is a linear operation. The new coordinates are a linear combination of the previous coordinates. The new coordinates are determined from a linear system

$x' = s_x\, x$

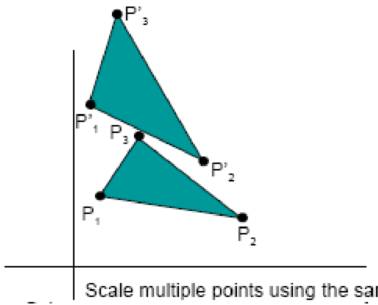
$y' = s_y\, y$

Hence, scaling can be expressed using matrix notation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

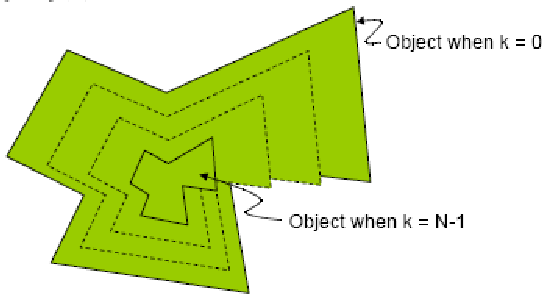
Using matrices to represent transformations is convenient e.g., (a)if you have many points to transform or (b) if the transformation is part of an animation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \text{ for } (x, y) = P_1, P_2, \text{ and } P_3$$



Scale multiple points using the same matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = ((N-k)/N) \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \text{ for } k = 0, 1, 2, \ldots N-1$$



The changing scale as a function of time makes this object appear to shrink

3.4 Linear transformations

The coordinate transformations for translation, rotation, and scaling can all be described by linear systems

Translation

$x' = x + t_x$

$y' = y + t_y$

Rotation

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

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

Scale

$x' = s_x x$

$y' = s_x y$

3.4.1 Affine Transformations

Affine transformations are coordinate transforms that can be described by linear systems

i.e., affine transformations can be written in the form

$x' = a_x x + b_x y + c_x$

$y' = a_y x + b_y y + c_y$

Translation, rotation, and scale are special examples of affine transformations

Affine transformations can be expressed in matrix notation

Linear system

$x' = a_x x + b_x y + c_x$

$y' = a_y x + b_y y + c_y$

Matrix notation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_x & b_x \\ a_y & b_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix}$$

3.5 Homogeneous Coordinates

Homogeneous coordinates are another way to represent points to simplify the way in which we express affine transformations

Conventional matrix notation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_x & b_x \\ a_y & b_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix}$$

Homogeneous coordinates are a more convenient notation for 2D transformations

•An equivalent representation

•Require a single matrix to represent general affine transformations

•Can be used to represent perspective transformations (later)

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

An affine transform using homogeneous coordinates

3.6 Matrix Transformations

Translation by (t_x, t_y)

Inverse of translation by (t_x, t_y) is translation by ($-t_x$, $-t_y$)

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & -t_x \\ 0 & 1 & t \end{pmatrix}$$

Rotation by θ about the origin

$$\mathbf{R} = \begin{pmatrix} \cos() & -\sin() & 0 \\ \sin() & \cos() & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \sin\theta & 0 \\ \cos\theta & 0 \\ 0 & 1 \end{pmatrix}$$

Inverse of rotation by θ is rotation by –θ

Recall cos(-θ) = cosθ and sin(-θ) = -sinθ

Scale by (sx, sy) about the origin

Inverse of scale by (s_x, s_y) is scale by ($1/s_x$,$1/s_y$)

$$\mathbf{S}^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

4.0 Conclusion

Simple transformations such as translation, rotation and scaling of objects and vectors can be performed using matrices. Firstly we look at these 3 transformations as applied to a 2D co-ordinate system.

5.0 Summary

In computer graphics, we use combinations of the following types of transformations (the affine transformations):

- Translation
- Rotation
- Scaling
- Shear

The order of transformations matters in some cases. We also use homogeneous coordinates because they allow us to write the affine transformations purely in terms of matrix multiplications:

6.0 Tutor Marked Assignment

1. Consider a rectangle whose corners are (1, 1), (3, 1), (3, 2), (1, 2).

(a) Describe the transformations which would rotate this rectangle by 90^o around its center

2. Consider a rectangle whose corners are (1, 1), (3, 1), (3, 2), (1, 2).

(a) Describe the transformations which would shear the rectangle so that its vertical sides are tilted 30◦ clockwise but the base of the rectangle is unchanged.

3. For each of the following cases, write a single 4x4 matrix that applies the given transformation to any arbitrary 3D point (x, y, z, 1). Assume that points to be transformed are represented as a column vectors and they are left-multiplied by the matrix. If it is impossible to define a matrix for the given transformation, say so and explain why.

(a) Transform (x, y, z, 1) to (x+3, -2y, 2z - 4, 1).

(b) Transform (x, y, z, 1) to (x+y/2, y, z, 1).

Also, what is the name for this kind of transformation?

(c) Rotate (x, y, z, 1) by Θ degrees around the Z axis (clockwise when viewed in the Z direction).

(d) Transform (x, y, z, 1) to (x, yz, 0, 1).

(e) Scale (x, y, z, 1) by a factor S around the point C = (1, 2, -3, 1) (you may leave your answer as a product of matrices):

7.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 3: 3D Graphics Rendering
Unit 2 Scan conversion

Table of Contents

Title
Page

44.0 Introduction:

No matter how complex the rendering process (2D/3D, orthographic/perspective, flat/smooth shading etc), ultimately all graphics comes down to:
write_pixel(x,y,colour)
We are likely to have to perform this operation many times for every pixel.
If we wish to draw frames fast enough for smooth 3D graphics, we clearly need to be able to turn our graphical primitives into write_pixel operations extremely quickly. It turns out even division operations need to be avoided. This process is called ***scan conversion or rasterisation***

45.0 Objectives:

In this unit, we are expected to look in some detail at two scan conversion algorithms for lines:

- DDA Algorithm
- Bresenham's Algorithm

Plus issues related to line intensity.
We will then look at methods for polygons:

- Filled polygons
- Edge tables

46.0 Scan Conversion for Lines

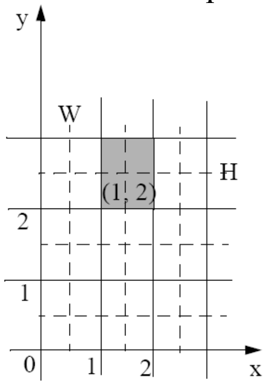
In order to scan convert lines, we must tackle the following problem:
Input
We have a line defined on the plane of real numbers (x, y). We have a discrete model of the (x, y) plane consisting of a regular array of rectangles called pixels, which can be coloured.
What we wish to do
Map the line onto the pixel array, while satisfying or optimising the following constraints:
Maintain constant brightness
Differing pen styles or thicknesses may be required
Shape of endpoints if line thicker than one pixel.

3.2 Pixel Space

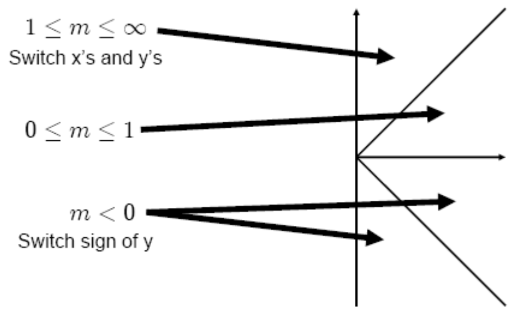
The pixel space is a rectangle on the x y plane, bounded by:
$0 \leq x \leq x_{max}$ and $0 \leq y \leq y_{max}$
Each axis is divided into an integer number of pixels, N_x and N_y. The pixels there for have width and height:

$$W = \frac{x_{max}}{N_x} \qquad H = \frac{y_{max}}{N_y}$$

Pixels are referred to using integer coordinates that either refers to the location of their lower left hand corners or their centres. Knowing the W and H values allows the pixel aspect ratio to be defined. Assume that W and H are equal so pixels are square.

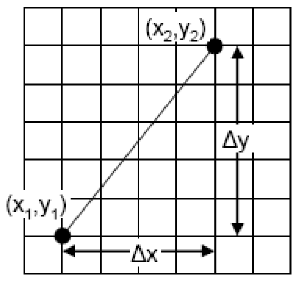


Assumptions on Gradient
Both of the algorithms we will look at make the assumption that the gradient of the line satisfies $0 \leq m \leq 1$
However, through symmetry we can handle all other cases



3.3 Digital Differential Analyser (DDA) Algorithm

A line segment is to be drawn from (x1,y1) to (x2,y2)



Generating a line segment is equivalent to solving a simple differential equation:

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

Starting from (x_1,y_1), for any change in x of size Δx, the corresponding change in y is:

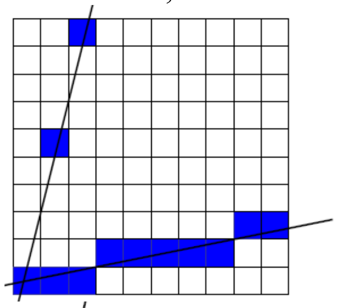
$\Delta y = m\Delta x$

The DDA algorithm uses a function round(x) to convert floating point, real valued coordinates to integer pixel coordinates:

```
line_start = round(x1);
line_end = round(x2);
colour_pixel(round(x1), round(y1));
for(i = line_start + 1; i<=line_end; i++{
   y1 = y1 + m; ←——————————————————— Floating point addition
   colour_pixel(i, round(y1));
}
```

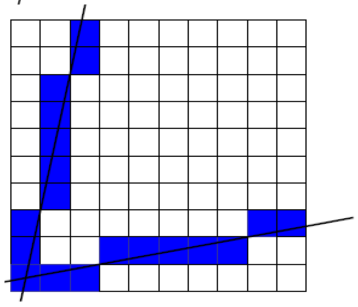
Actual pixel coloured uses rounded version of current y value

There is no scaling involved in moving from projection to viewport coordinates, so we can use these directly.



If we use this algorithm when m>1, the line is drawn incorrectly. The algorithm says: "for each x, find best y" but for large m, gap between pixels is large.

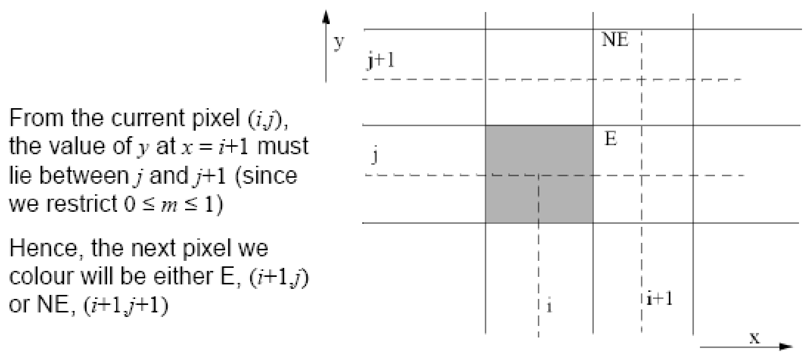
Solution, when m>1, switch roles of x and y. Say "for each y, find best x". Using this modified approach we get…



Correctly drawn lines using modified DDA algorithm that checks gradient and exploits symmetry as appropriate.

3.4 Bresenham's Algorithm

The DDA algorithm requires a floating point addition and rounding operation for every iteration Bresenham's algorithm operates only on integers requiring only that the start and end points of a line segment are rounded



From the current pixel (i,j), the value of y at $x = i+1$ must lie between j and $j+1$ (since we restrict $0 \le m \le 1$)

Hence, the next pixel we colour will be either E, $(i+1,j)$ or NE, $(i+1,j+1)$

The key idea of Bresenham's algorithm is to reduce line drawing to the decision problem of choosing between N and NE for the next pixel

It turns out that we can design a decision variable on which to base this choice which we can compute without division. Rewrite equation of a straight line:

$$y = mx + c$$

$$mx - y + c = 0$$

$$\frac{\Delta y}{\Delta x}x - y + c = 0$$

Multiplying through by Δx gives us our decision variable:

$$F(x, y) = x\Delta y - y\Delta x + c\Delta x = x\Delta y - y\Delta x + B = 0$$

The useful property of this function is that:

$F(x,y) < 0$ for points above the line

$F(x,y) > 0$ for points below the line

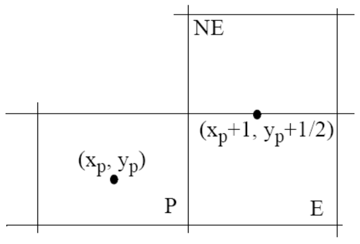
We can now use the decision variable to derive our line drawing algorithm. This involves some rational numbers in the derivation, but we cancel for these at the end. Consider an arbitrary pixel (x_p, y_p) that is on the line segment. We need to design a test to decide which of the next 2 pixels to colour. The point to test is $(x_p+1, y_p+1/2)$:

This point will be on the boundary between pixels E and NE and at the horizontal midpoint

So:

If $F(x_p+1, y_p+1/2) \le 0$ then next pixel is E

Otherwise next pixel is NE



The decision for our next step is dependent on the decision we have just made:

If E was chosen then the next test will be $F(x_p+2, y_p+1/2)$
If NE was chosen the next test will be $F(x_p+2, y_p+3/2)$
We can obtain an iterative form of this decision variable which accounts for these two cases
Let dn = $F(x_p+1, y_p+1/2)$
If we substitute (x+2, y+1/2) and (x+2, y+3/2) back into the definition of F we end up with:
If E was chosen, the next test will be $d_{n+1} = d_n + \Delta_y$
If NE was chosen, the next test will be $d_{n+1} = d_n + (\Delta_y-\Delta_x)$
To initialise the process we need a formula to compute d_1:

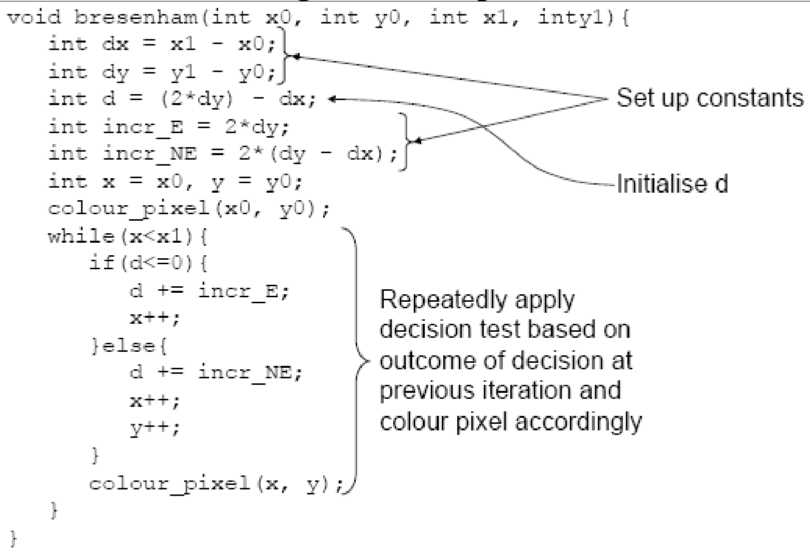
$$d_1 = F(x_0+1, y_0+1/2) = \Delta y(x_0+1) - \Delta x(y_0+1/2) + B$$
$$d_1 = (x_0\Delta y - y_0\Delta x + B) + \Delta y - \Delta x/2$$
$$d_1 = F(x_0, y_0) + \Delta y - \Delta x/2$$
$$d_1 = d_0 + \Delta y - \Delta x/2$$

Since we know that the starting point is on the line, d0 = 0 and hence:

$$d_1 = \Delta y - \Delta x/2$$

This still has a factor of 1/2 in it and we wanted only integers
Solution: simply multiply everything by 2

3.4.1 Bresenham's Algorithm: Implementation

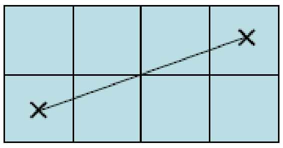


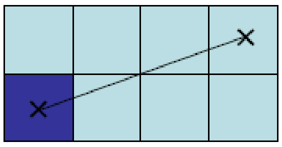
```
void bresenham(int x0, int y0, int x1, inty1){
   int dx = x1 - x0;
   int dy = y1 - y0;
   int d = (2*dy) - dx;
   int incr_E = 2*dy;
   int incr_NE = 2*(dy - dx);
   int x = x0, y = y0;
   colour_pixel(x0, y0);
   while(x<x1){
      if(d<=0){
         d += incr_E;
         x++;
      }else{
         d += incr_NE;
         x++;
         y++;
      }
      colour_pixel(x, y);
   }
}
```

Set up constants
Initialise d
Repeatedly apply decision test based on outcome of decision at previous iteration and colour pixel accordingly

Bresenham’s Algorithm: Worked Example

We would like to draw a line from (1,1) to (4,2):





We can begin by colouring pixel (1,1)

and computing some values that we will need:

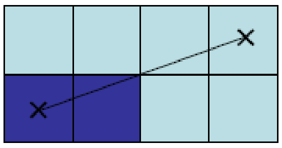
$\Delta x = 4\text{-}1 = 3 \qquad 2\Delta y = 2$

$\Delta y = 2\text{-}1 = 1 \qquad 2(\Delta y\text{-}\Delta x) = 2^*(1\text{-}3) = \text{-}4$

We initialise $d_0 = 0$ and hence:

$d_1 = 2\Delta y - \Delta x = 2\text{-}3 = \text{-}1$

When d<=0 we move East, so we can colour the next pixel:



$\Delta x = 4\text{-}1 = 3$	$2\Delta y = 2$
$\Delta y = 2\text{-}1 = 1$	$2(\Delta y\text{-}\Delta x) = 2^*(1\text{-}3) = \text{-}4$

Now we simply need to iteratively apply the rule:

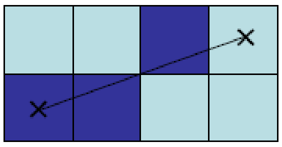
$$d_{n+1} = d_n + \begin{cases} 2\Delta y & \text{if } d_n \le 0 \\ 2(\Delta y - \Delta x) & \text{otherwise} \end{cases}$$

d_1 was less than 0,

so we use:

$d_2 = d_1 + 2\Delta y$

$= \text{-}1 + 2 = 1$



This time, d is greater than 1, so we move North-East and hence know which pixel to colour.

$\Delta x = 4\text{-}1 = 3$	$2\Delta y = 2$
$\Delta y = 2\text{-}1 = 1$	$2(\Delta y\text{-}\Delta x) = 2^*(1\text{-}3) = \text{-}4$

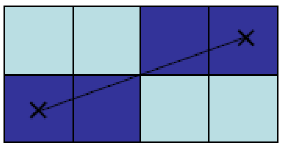
Again, apply the iterative rule:

$$d_{n+1} = d_n + \begin{cases} 2\Delta y & \text{if } d_n \le 0 \\ 2(\Delta y - \Delta x) & \text{otherwise} \end{cases}$$

d_2 was >0, so we use:

$d_3 = d_2 + 2(\Delta y\text{-}\ \Delta x)$

$= 1 - 4 = \text{-}3$

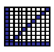


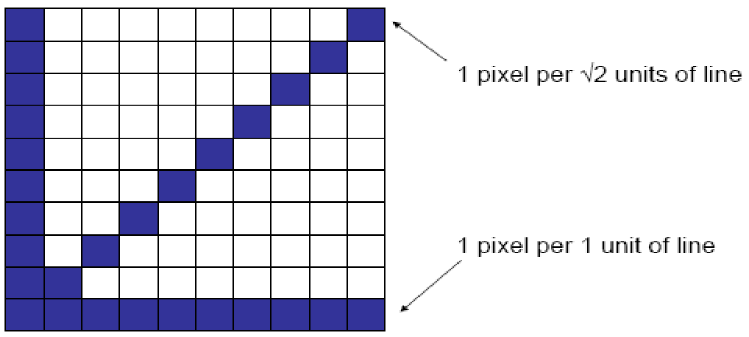
This time, d is less than 0, so we finish by moving East and colouring the last pixel.

$\Delta x = 4\text{-}1 = 3$	$2\Delta y = 2$
$\Delta y = 2\text{-}1 = 1$	$2(\Delta y\text{-}\Delta x) = 2^*(1\text{-}3) = \text{-}4$

3.5 Line Intensity

Simple line drawing algorithms such as DDA and Bresenham’s algorithm do not give lines that appear with the correct intensity:
Apparent line intensity is a function of the gradient
Lines with different slopes have a different number of pixels per unit length
To draw two such lines with the same intensity must make pixelintensity a function of the gradient (or use antialiasing)

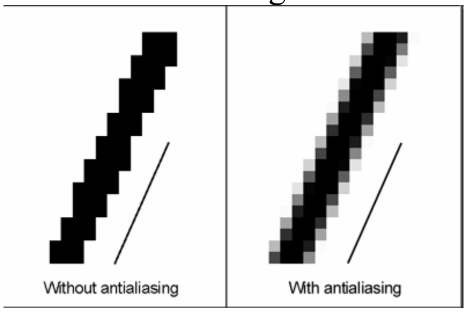


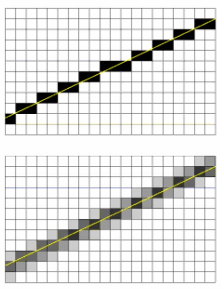


Pixels per unit line length is a function of gradient (densest for horizontal and vertical lines, sparsest for 45 degree lines). To make lines appear the same brightness, must vary intensity according to gradient.

Anti-aliasing

The best solution to the problem of maintaining consistent line intensity is to use antialiasing:

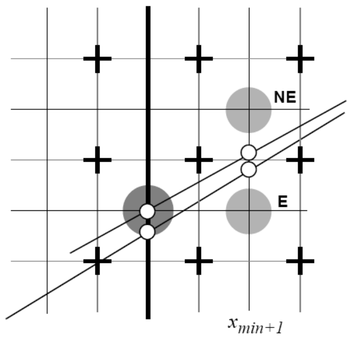




If lines are assigned a thickness, we can calculate what proportion of each pixel is covered by the line and colour accordingly. Pixels only partially covered should be blended with whatever is behind

3.6 Rasterisation and Clipping

If a line segment has been clipped, we need to be careful how the clipped end is treated:



At the clipping boundary, one coordinate will be integer, one real

Pixel at boundary will be correctly drawn

Subsequent pixels may not be as gradient of clipped line is different

The solution is:

Draw edge pixel

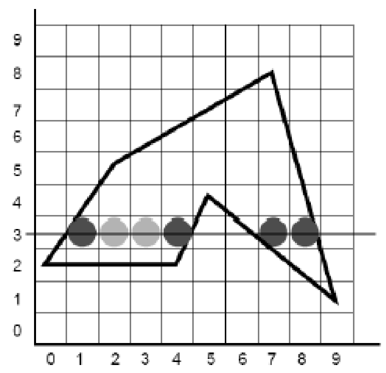
Initialise F(…) for next column over

Use original (not clipped gradient)

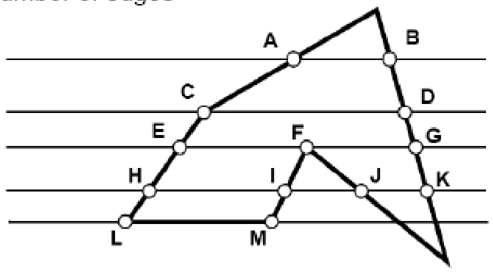
Filled Polygons

The basic idea:

```
for each scan line
   calculate list of edge intersections
   sort list in increasing x
   for each pair (p1, p2) in list
   fill span from p1 to p2
```



Line through a polygon will cross an even number of edges



47.0 Conclusion

Displaying graphical content of any kind, whether 2D or 3D, orthographic or perspective, flat/smooth shading etc, eventually it comes down to drawing individual pixels with individual colours to the screen. This is the fundamental operation in graphics.

48.0 Summary

Pixels are discrete elements of a grid defined over the same plane. For a given primitive, we tried to determine which pixels should be coloured. We also discussed the basic line drawing algorithms:

- DDA Algorithm: Conceptually simple, but requires floating point arithmetic at every iteration
- Bresenham's Algorithm: Draws the same lines, but avoids all floating point operation. Based on evaluating a decision variable at every iteration which
- Rasterizing polygons is in some ways easier. We just need to keep track of when we enter and exit a polygon as we move along a scanline, either colouring pixels or not. Need to be careful with abutting polygons etc.

49.0 Tutor Marked Assignment

1. Describe a way to test whether a point is inside a triangle
2. Can this "inside triangle test" be applied to arbitrary polygons? (Explain in one short sentence.)
3. What modifications must be made to the **sweep-line** algorithm for scan converting triangles in order to apply it to arbitrary polygons?
4. What approach is normally implemented in modern graphics systems for scan-converting arbitrary polygons?

50.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 3: 3D Graphics Rendering
Unit 3 Three-Dimensional Viewing

Table of Contents

Title
Page

51.0 Introduction:

This unit will introduce you to the concept of 3D viewing. The concepts are the same as
2D viewing except there is a new dimension to work with depth.
In this unit you will gain the skills to design more complex 3D scenes and to manipulate a viewing camera.

52.0 Objectives:

- Upon successful completion of this module students will be able to:
- Learn the basic ways of projecting a tree dimensional scene
- Develop 3D tools for use in controlling the camera in a scene
- Understand the transformations used in 3D animation.

53.0 3D Camera Transforms

Camera Analogy

The rendering pipeline mimics photography;

The **viewing transform:** performs the same operation as mounting a camera on a tripod to view a scene.

The **model transform:** places objects in the scene, just as objects are placed in front of the camera.

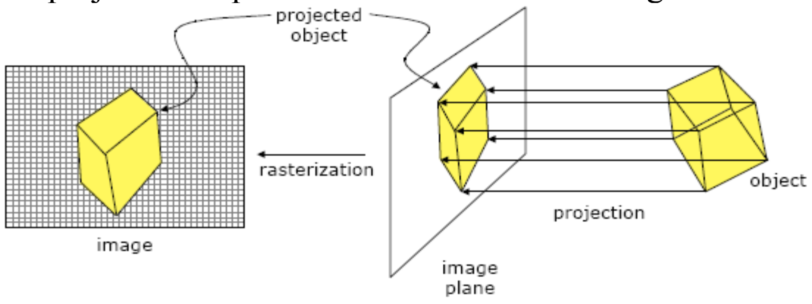
The **projection** transform: defines the shape of the viewing volume, just as the camera's lens determines what the camera sees, e.g., a wide angle lens sees a wide viewing volume while a telephoto lens sees a narrow viewing volume.

The image is rendered to the **viewport** just as the photograph is rendered on film.

3.1 Camera Projection

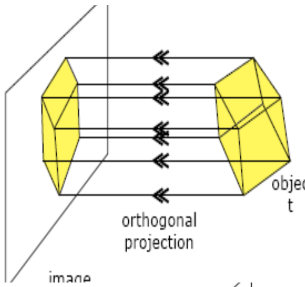
Projecting camera space onto the image plane

3D objects are projected onto the 2D image plane (a.k.a. viewport). The 2D projected shapes are rasterized into the image.



There are two basic forms of projection

3.1.1 Orthogonal projection

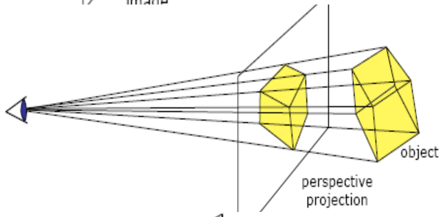


• Parallel lines remain parallel

• Relative distances and angles are better preserved

Used for CAD/CAM, architectural drawings, medical applications

3.1.2 Perspective projection



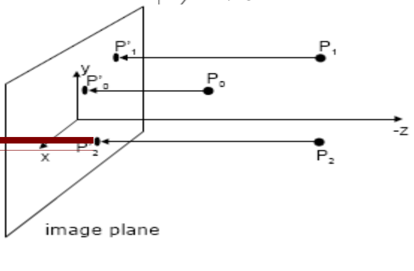
• Parallel lines don't remain parallel, rendered object size decreases with distance from the image plane

• More realistic, provides a sense of being in the scene

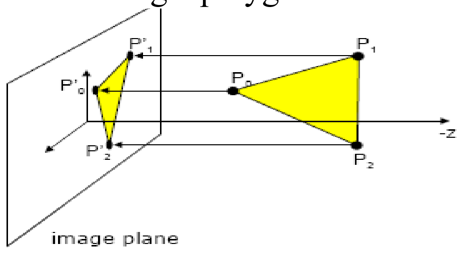
Used for immersive environments

3.1.3 Orthographic Projection

Parallel projection

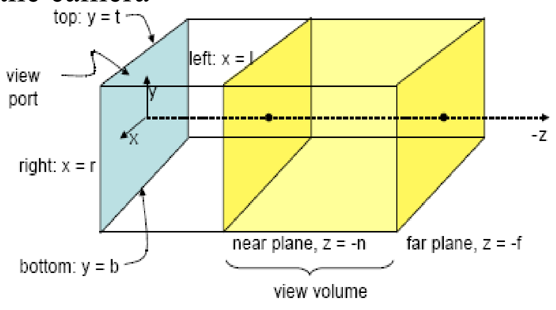


Points are projected from 3D camera space onto the 2D screen in a direction parallel to the view direction;
Points map to points, Straight lines map to straight lines. 3D polygons are mapped to 2D polygons in the 2D image plane. The 2D polygons can be rendered using a polygon fill method



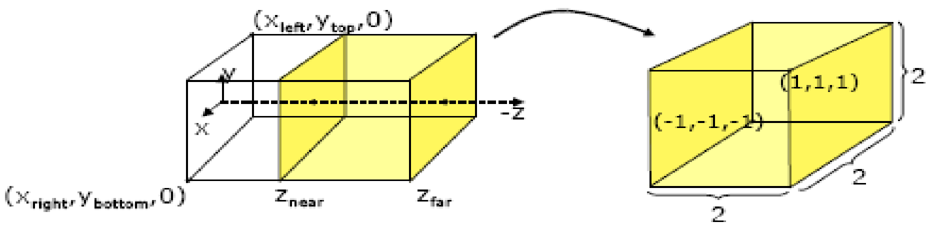
3.2 View port

What is seen depends on the window, or view port, that you look through
• A large view port sees more of camera space
• A small view port sees less of camera space
The view port corresponds to what is recorded on the image. The view port is specified by the left, right, top, bottom, near and far planes
$l <= x <= r$
$b <= y <= t$
$n <= -z <= f$, where n and f are specified as positive values. These limits define a rectangular prism that is perpendicular to viewing plane for orthographic projection.
The view port defines the view volume – the volume of space visible to the camera



3.3 Performing the orthographic projection

It is convenient for lots of algorithms to transform points in the view volume so that the x, y, and z boundaries map to +/-1. The orthographic projection scales x, y, and z so that the rectangular prism is in from -1 to 1 in x, y, and z. This is a cube with sides of length 2, centered on the camera axis.



To translate the camera axis to the origin

$t_x = -(r+l)/(r-l)$, $t_y = -(t+b)/(t-b)$, $t_z = -(f+n)/(f-n)$

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & -(x_{right}+x_{left})/2 \\ 0 & 1 & 0 & -(y_{top}+ y_{bottom})/2 \\ 0 & 0 & 1 & -(z_{far}+z_{near})/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} 2/(x_{right}-x_{left}) & 0 & 0 & 0 \\ 0 & 2/(y_{top}-y_{bottom}) & 0 & 0 \\ 0 & 0 & 2/(z_{near}-z_{far}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To scale to a 2x2x2 cube

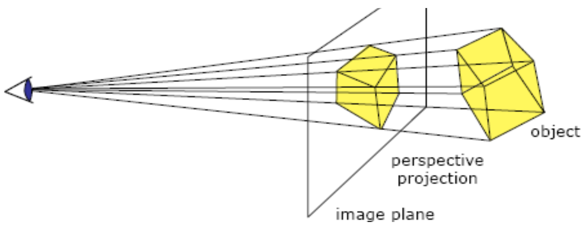
$s_x = 2/(r-l)$, $s_y = (2/(t-b)$, $s_z = (2/f-n)$

Translate the camera axis to the origin

Scale to a 2x2x2 cube

3.4 Perspective Projection

Models the optics of the eye. Objects are projected towards an eye point situated behind the camera plane. Perspective projection tends to look more realistic than orthographic projection

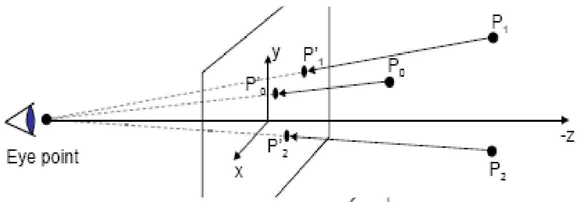


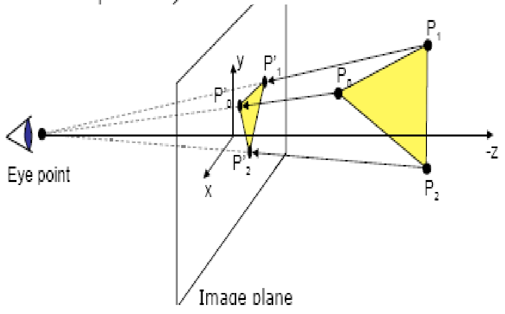
3.5 Non-parallel projection

The Points are projected from 3D camera space onto the 2D screen along rays that converge to the eye point

Properties:

- Points map to points
- Straight lines map to straight lines
- 3D polygons are mapped to 2D polygons in the 2D image plane
- The 2D polygons can be rendered using a polygon fill method





What is seen depends on the view port:

A large view port sees more of camera space

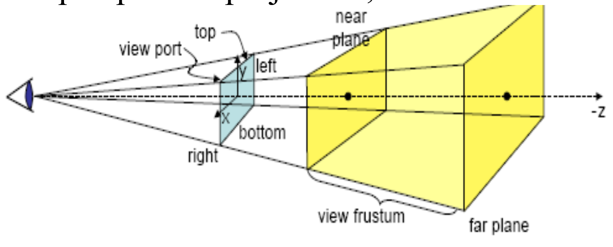
A small view port sees less of camera space

And its location relative to the eye point

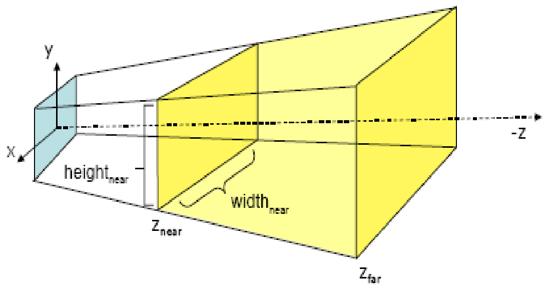
The eye point and the location of the view port determine the field of view

3.6 View frustum

The view port and its location relative to the eye point define the view volume. The view frustum is the volume of space visible to the camera. For perspective projection, the view frustum is a truncated pyramid.



View frustum for perspective projection



The axially symmetric view frustrum is specified by $width_{near}$, $height_{near}$, z_{near} and x_{far}

3.6.1 Specifying the view frustum

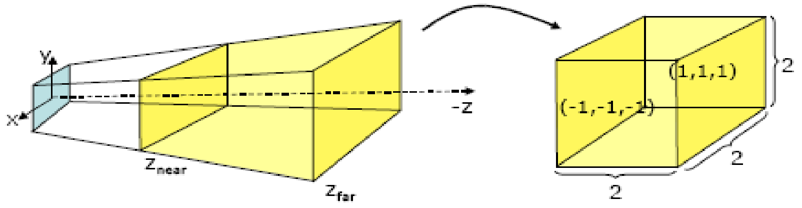
Assume the camera looks down the negative z-axis and that the eye point lies at the origin, then the view frustum is symmetric about the z-axis. The view frustum is fully specified by the width and height at the near plane and z_{near} and z_{far}

3.7 Performing the perspective projection

Like orthographic projection, it is convenient (e.g., for clipping) to transform the view frustum so its boundaries map to +/-1

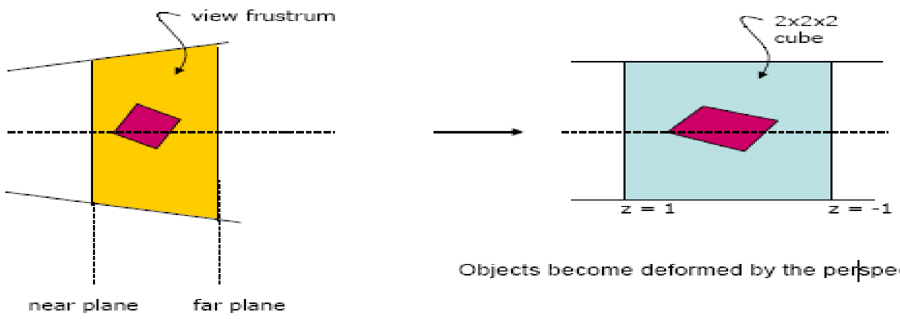
• Map the truncated pyramid to a cube with sides of length 2, centered on the camera axis

• How do we determine the mapping?



Note: Objects are deformed by perspective projection

Mapping the truncated pyramid to the 2x2x2 cube deforms the camera space



Objects become deformed by the perspective transform

Objects farther from the view port become smaller

Assumptions about the view frustum

• The camera looks down the negative z-axis

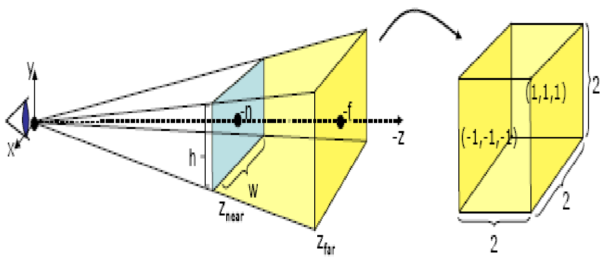
• The view port is symmetric about the z-axis

It has width w and height h

• The eye point lies at the origin

• The camera plane lies at the near plane, z_{near} = -n

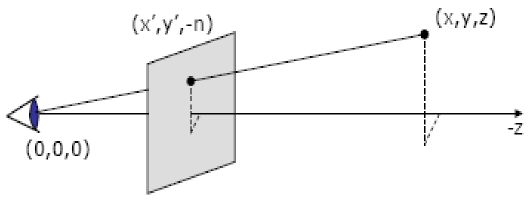
• The far plane is z_{far} = -f

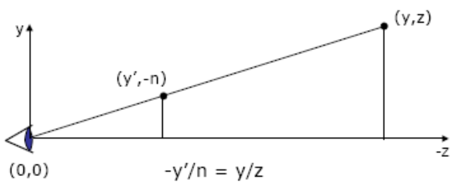


For convenience, place the camera plane at the near plane of the view frustrum

3.7.1 Specifying perspective projection

What happens to a point (x,y,z) when it is projected onto the view window?



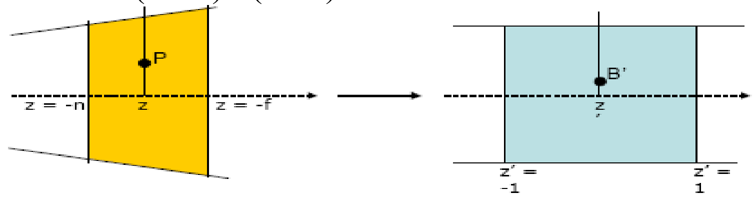


(x, y, z) projects onto the view port along the line from (x,y,z) to the origin

Use similar triangles to verify that

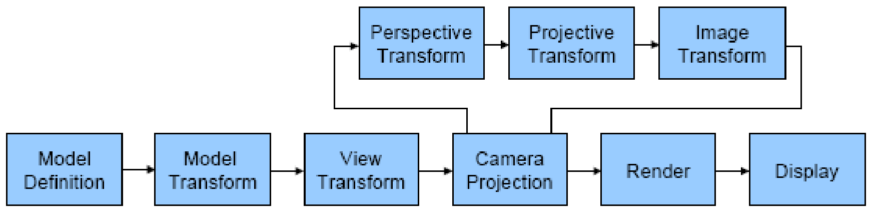
x' = xn /z
y' = yn /z
Note that as z gets larger, x' and y' get smaller
Objects look smaller when they are farther away
To map (x, y, z) into the 2x2x2 box
• Scale x and y by -n/z to model converging line of sight
• Scale x and y so that x' and y' range from -1 to 1 instead of –w/2 to w/2 and –h/2 to h/2
x' = -(xn/z) / (w/2) = -(x/z)(2n/w)
y' = -(yn/z) / (h/2) = -(y/z)(2n/h)

How do we map z into the 2x2x2 box?
One possibility would be to map z between n and f to values between -1 and 1 using a linear function, i.e.
z' = -1+ 2(z – n) / (f – n)



The function z' = -1+ 2(z – n) / (f – n) is a linear function that maps points between z = -n and z = -f to points from -1 to 1

3.8 Graphics Pipeline Transformations



Model transform
Place the object in the world coordinates
The triangle is originally specified in the coordinates of the model (perhaps in a hierarchy). The model is then transformed into world coordinates to place it in the scene
$(x,y,z)_{\text{Object}} \rightarrow (x,y,z)_{\text{WorldCoordinates}}$

3.9 View transform

Determine what is seen from the camera
• Perform the coordinate transform (rotation and translation) to camera space
$(x,y,z)_{\text{WorldCoordinates}} \rightarrow (x,y,z)_{\text{CameraSpace}}$

3.9.1 Orthographic or perspective transform

Transform what is seen by the camera into the standard 2x2x2 view volume

• Perform the orthographic or perspective projection into the view volume

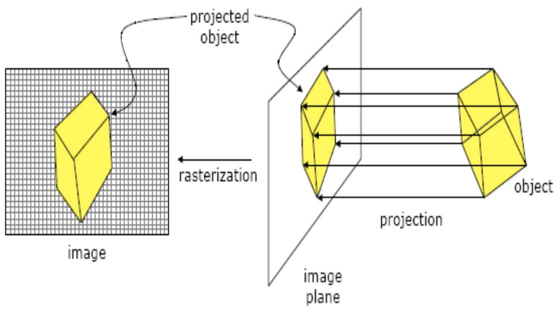
$(x,y,z)_{CameraSpace} \rightarrow (x,y,z)_{ViewVolume}$

Projecting camera space onto the image plane

3D objects are **projected** onto the 2D view port

The viewport is mapped to the image

The 2D projected shapes are rasterized into the image



3.10 Projective transform

Project each point in the view volume onto the view port

• Points in the view volume are projected onto the viewport by discarding the z component of view volume coordinates $(x,y,z)_{ViewVolume} \rightarrow (x,y)_{Viewport}$

• Use the z component of the view volume;

To determine which object is in front if two objects overlap

To modify the pixel color if depth-based shading is used

For special effects (e.g., depth-based fog)

3.11 Image transform

Transform viewport coordinates into image coordinates;

• Maps $(-1,1) \times (-1x1)_{Viewport} \rightarrow (0,W-1) \times (0,H-1)_{Image}$

• Requires (another) scale and translation

(1) Scale the view port to the size of the image $s_x = W/2$, $s_y = H/2$

(2) Translate the origin of the view port to the center of the image $t_x = W/2$, $t_y = H/2$

$$\mathbf{S} = \begin{pmatrix} W/2 & 0 & 0 & 0 \\ 0 & H/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & W/2 \\ 0 & 1 & 0 & H/2 \\ 0 & 0 & 1 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Image Transform} = \mathbf{TS} = \begin{pmatrix} W/2 & 0 & 0 & W/2 \\ 0 & H/2 & 0 & H/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

54.0 Conclusion

Great visual realism can be achieved using complex texturing and lighting effects. However even before applying these to a 3D scene the computer graphics programmer needs to understand how issues of perspective can best be addressed.

55.0 Summary

This introduces us to the basic camera concepts. It explains how you can create perspective views of 3D scenes and how to position and move the camera.

You should be familiar with the concepts of:

- eye
- view volume

- view angle
- near plane
- far plane
- aspect ratio
- viewplane.

56.0 Tutor Marked Assignment

1. Draw four pictures of a rectangular box. From left to right, the drawings should be in
   (1) one-point, (2) two-point, and (3) three-point perspective, and (4) orthographic projection
2. **2.** What is the difference between orthographic and oblique projection?
3. Write out the 4x4 perspective projection matrix that projects from 3D camera coordinates into screen coordinates:

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ w_s \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

57.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.

P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.

Module 3: 3D Graphics Rendering

Unit 4 3D transform and Animation

Table of Contents

58.0 Introduction:

In this unit, we will address 3D transformation. This allows us to represent translations, scalings, and rotations as multiplication of a vector by a matrix in a three dimensional scene. And also take a brief look at computer animation principles.

59.0 Objectives:

To give explore the different ways of transforming 3D objects
To provide a comprehensive introduction to computer animation
To study the difference between Traditional and Computer Animation

60.0 3D Transformations

We can represent 3D transformations by 4 × 4 matrices in the same way as we represent
2D transformations by 3 × 3 matrices.
•Much of computer graphics deals with 3D spaces
3D transformations are similar to 2D transforms, Some important transformations are:
•Translation
•Rotation
•Scaling
•Reflection
•Shears

3.0.1 3D Translation

$(x, y, z) \rightarrow (x + t_x, y + t_y, z + t_z)$

It is similar to 2D transformations, a 3D translation can be written as a set of linear equations in x, y, and z

$x' = 1x + 0y + 0z + t_x$

$y' = 0x + 1y + 0z + t_y$

$z' = 0x + 0y + 1z + t_z$

3.1 In matrix notation

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

Using homogenous coordinates

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

3.1.1 3D Scale about the origin

$(x, y, z) \rightarrow (a_x, b_y, c_z)$

–In matrix notation

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Using homogenous coordinates

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3.1.2 3D Scaling about an arbitrary point (p_x, p_y, p_z)

1.Translate (p_x, p_y, p_z) to the origin
2.Scale about the origin
3.Translate the origin back to (p_x, p_y, p_z)

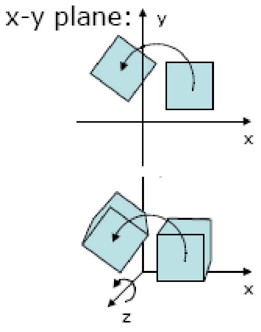
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3.2 3D Rotation

We can rotate about any axis but will start with rotations about the coordinate (x, y, and z) axes

Rotation about the z-axis:

This is related to a 2D rotation in the x-y plane:



x’ = xcosθ – ysinθ
y’ = xsinθ + ycosθ
z’ = z

In (homogeneous) matrix form:

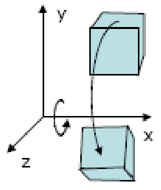
Notes about the direction of rotation

By convention, the rotation angle is positive if it is in a counter-clockwise direction when looking in the negative direction down the axis of rotation.

3.2.1 Rotation about the x-axis:

x’ = x
y’ = ycosθ – zsinθ
z’ = ysinθ + zcosθ
In (homogeneous) matrix form:

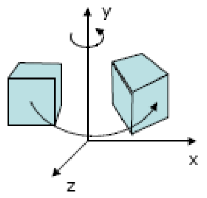
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$



3.2.2 Rotation about the y-axis:

x’ = xcosθ + zsinθ
y’ = y
z’ = -xsinθ + zcosθ
In (homogeneous) matrix form:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

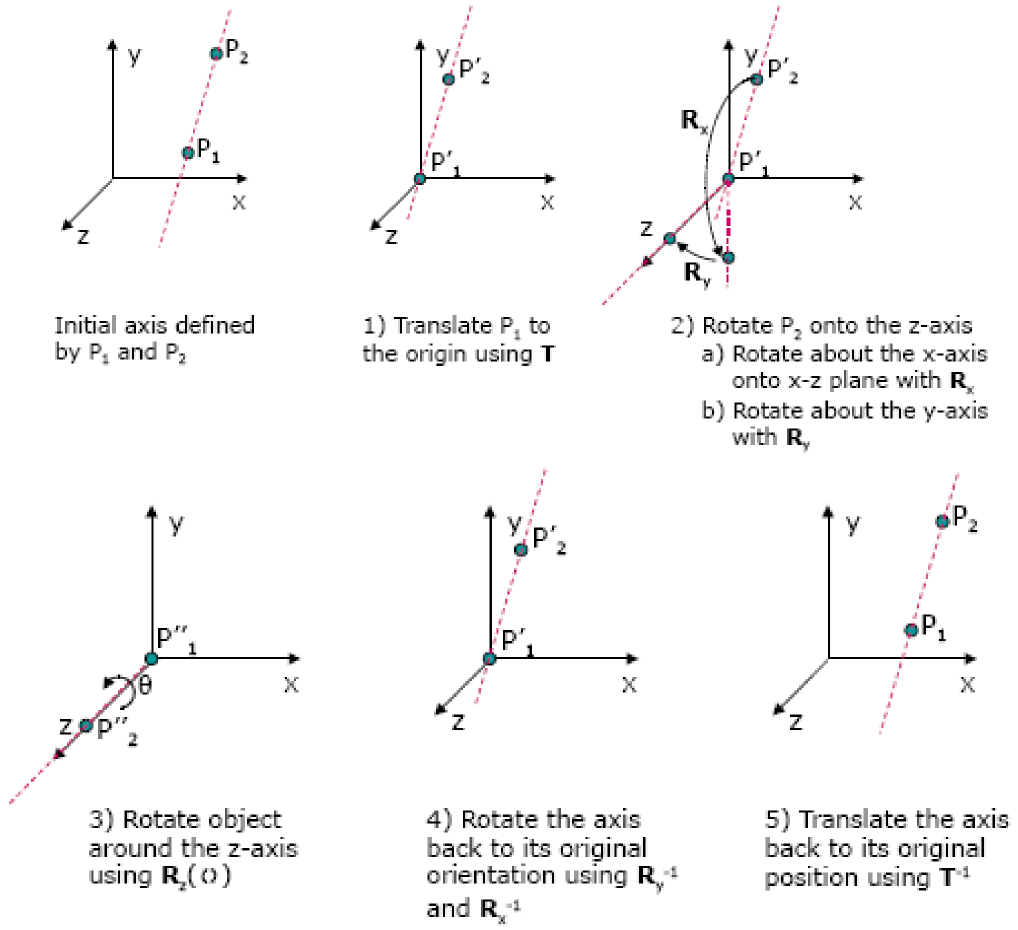


3.2.3 General 3D Rotation

To rotate about an arbitrary axis

•Translate the object so the rotation axis passes through the origin
•Rotate the object so that the rotation axis coincides with one of the coordinate (x, y, or z) axes
•Perform the specified rotation about the selected coordinate axis
•Apply the inverse rotations to bring the coordinate axis back to its original orientation
•Apply the inverse translation to bring the rotation axis back to its original spatial location

To rotate about an arbitrary axis



y
P_2
P_1
x
z
Initial axis defined by P_1 and P_2
y
P'_2
P'_1
x
z
1) Translate P_1 to the origin using **T**
$\mathbf{R}_x$
y
P'_2
P'_1
x
z
$\mathbf{R}_y$
2) Rotate P_2 onto the z-axis
a) Rotate about the x-axis onto x-z plane with $\mathbf{R}_x$
b) Rotate about the y-axis with $\mathbf{R}_y$
y
P''_1
x
θ
z
P''_2
3) Rotate object around the z-axis using $\mathbf{R}_z(\theta)$
y
P'_2
P'_1
x
z
4) Rotate the axis back to its original orientation using $\mathbf{R}_y^{-1}$ and $\mathbf{R}_x^{-1}$
y
P_2
P_1
x
z
5) Translate the axis back to its original position using $\mathbf{T}^{-1}$

$\mathbf{R}(\theta) = \mathbf{T}^{-1}\mathbf{R}_x^{-1}\mathbf{R}_y\text{-}^{1}\mathbf{R}_z\mathbf{R}_y\mathbf{R}_x\mathbf{T}$

These matrices are derived from the direction vector of the axis of rotation and a point on the axis.

3.3 Traditional Animation

At the early day of the history of animation, it took a lot of effort to make an animation, even the shortest ones. In film, every second requires 24 picture frames for the movement to be so smooth that humans cannot recognise discrete changes between frames. Before the appearance of cameras and computers, animations were produced by hand. Artists had to draw every single frame and then combined them as one animation.

It is worth mentioning about some of the techniques that were used to produce animations in the early days that are still being employed in computer-based animations:

3.1.1 ***Key frames***: this technique is used to sub divide the whole animation into key points between which a lot of actions happen. For example, to specify an action of raising a hand, at this stage the manager only specifies the start and finish positions of the hand without having to worry about the image sequence in between. It is then the artist's job to draw images in between the start and finish positions of the hand, a process called in-betweening. Using this technique, many people can be involved in producing one animation and hence it helps reduce the amount of time to get the product done. In today's computer animation packages, key frame technique is used as a powerful tool for designing. Here, the software does the in-betweening.

3.3.2 Cel animation: this is also a very powerful technique in producing animations. It is common that only a few objects will change in the animation. It is time consuming to draw the whole image background for every single frame. When using Cel animation method, moving objects and background are drawn on separate pictures and they will be laid on top of each other when merging. This technique significantly reduces production time by reducing the work and allowing many people can work independently at the same time.

Motion can bring the simplest of characters to life. Even simple polygonal shapes can convey a number of human qualities when animated: identity, character, gender, mood, intention, emotion, and so on. Animation make objects change over time according to scripted actions.

In general, animation may be achieved by specifying a model with n parameters that identify degrees of freedom that an animator may be interested in such as

• polygon vertices,
• spline control,
• joint angles,
• muscle contraction,
• camera parameters, or
• color.

With n parameters, this results in a vector ~q in n-dimensional state space. Parameters may be varied to generate animation. A model's motion is a trajectory through its state space or a set of motion curves for each parameter over time, i.e. ~q(t), where t is the time of the current frame.

Every animation technique reduces to specifying the state space trajectory. The basic animation algorithm is then: for t = t1 to tend: render(~q(t)).

Modeling and animation are loosely coupled. Modeling describes control values and their actions.

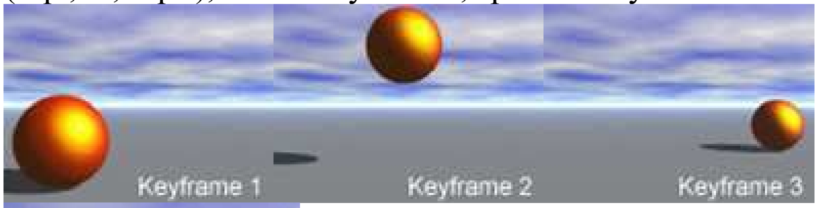
Animation describes how to vary the control values. There are a number of animation techniques, including the following:

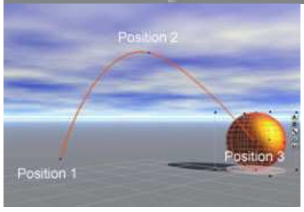
• User driven animation
    Keyframing
    Motion capture
• Procedural animation
    Physical simulation
    Particle systems

Crowd behaviors
• Data-driven animation

3.4 Keyframing

Keyframing is an animation technique where motion curves are interpolated through states at times,
(~q1, ..., ~qT), called keyframes, specified by a user.

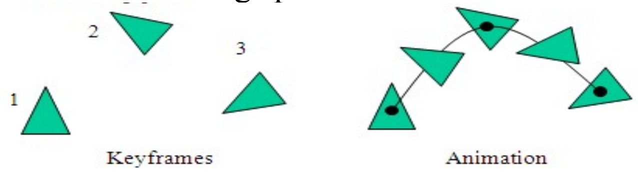




Underlying technique is *interpolation*
The in-between frames are interpolated from the keyframes. Originally done by armies of underpaid animators but now done with computers

3.5 Interpolation

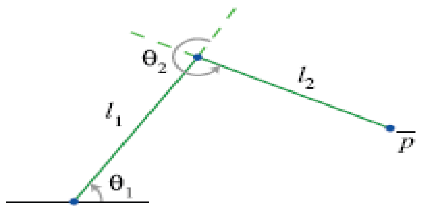
- Interpolating splines are smooth curves that interpolate their control points. it is perfect for keyframe animation
- Typically, time is directly associated with the parameter value, controlling speed



Anything can be keyframed and interpolated. Position, Orientation, Scale, Deformation, Patch Control Points (facial animation), Color, Surface normals
Special interpolation schemes for things like rotations. Use *quaternions* to represent rotation and do spherical interpolation, Control of parameterization controls speed of animation

3.6 Kinematics

Kinematics describe the properties of shape and motion independent of physical forces that cause motion. Kinematic techniques are used often in keyframing, with an animator either setting joint parameters explicitly with forward kinematics or specifying a few key joint orientations and having the rest computed automatically with inverse kinematics.
Forward Kinematics
With forward kinematics, a point ¯p is positioned by ¯p = f(θ) whereθ is a state vector (θ1, θ2, ...θn) specifying the position, orientation, and rotation of all joints.



For the above example, $\bar{p} = (l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2), l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2))$.

Inverse Kinematics

With inverse kinematics, a user specifies the position of the end effector, p, and the algorithm has to evaluate the required θ give p. That is, $\theta = f^{-1}(p)$. Usually, numerical methods are used to solve this problem, as it is often nonlinear and either underdetermined or over determined. A system is underdetermined when there is not a unique solution, such as when there are more equations than unknowns. A system is overdetermined when it is inconsistent and has no solutions.

Extra constraints are necessary to obtain unique and stable solutions. For example, constraints may be placed on the range of joint motion and the solution may be required to minimize the kinetic energy of the system.

3.7 Motion Capture

In motion capture, an actor has a number of small, round markers attached to his or her body that reflect light in frequency ranges that motion capture cameras are specifically designed to pick up.

With enough cameras, it is possible to reconstruct the position of the markers accurately in 3D.

In practice, this is a laborious process. Markers tend to be hidden from cameras and 3D reconstructions fail, requiring a user to manually fix such drop outs. The resulting motion curves are often noisy, requiring yet more effort to clean up the motion data to more accurately match what an animator wants.

Despite the labor involved, motion capture has become a popular technique in the movie and game industries, as it allows fairly accurate animations to be created from the motion of actors. However, this is limited by the density of markers that can be placed on a single actor. Faces, for example, are still very difficult to convincingly reconstruct.

Motion capture is one of the primary animation techniques for computer games

Gather lots of snippets of motion capture e.g.: Several ways to dunk, dribble, pass, and arrange them so that they can be pieced together smoothly. At run time, figure out which pieces to play to have the character do the desired thing

Problems:

Once the data is captured, it's hard to modify for a different purpose

• Uses:

Character animation

Medicine, such as kinesiology and biomechanics

3.8 Procedural Animation

Animation is generated by writing a program that outputs the position/shape/whatever of the scene over time

Generally:

Program some rules for how the system will behave

Choose some initial conditions for the world

Run the program, maybe with user input to guide what happens

Advantage: Once you have the program, you can get lots of motion

Disadvantage: The animation is generally hard to control, which makes it hard to tell a story with purely procedural means

61.0 Conclusion

3D objects are transformed the same as 2D except the we have to consider the depth in 3D

Animation produces the illusion of movement which are displayed in a series of frames with small differences between them done in rapid succession, eye blends to get motion.

62.0 Summary

In 3D computer graphics, we also use combinations of the following types of transformations (the affine transformations):

- Translation
- Rotation
- Scaling
- Shear

Computer animation became available when computers and animation creating software packages became available. With the processing power of computers and the utilities offered by many drawing software packages, making animations has become more efficient and less time consuming.

63.0 Tutor Marked Assignment

1 One principle of traditional animation is called "squash and stretch." Name and describe three more principles.

2 Describe a problem with using linear interpolation between keyframes.

3 Describe a problem with using interpolating splines between keyframes

4 What is the basic difference between dynamics and kinematics?

5 What is the basic difference between forward kinematics and inverse kinematics?

6 Where did the word *cel* in "cel animation" come from?

64.0 Refrences/Further reading:

Rick Parent. "Computer Animation: Algorithms and Techniques": Morgan-Kaufmann.

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.

J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.

Module 3: 3D Graphics Rendering
Unit 5 Hidden Surface Elimination

Table of Contents

65.0 Introduction:

We consider algorithmic approaches to an important problem in computer graphics, *hidden surface removal*. We are given a collection of objects in 3-space, represented, say, by a set of polygons, and a viewing situation, and we want to render only the visible surfaces. This unit will guide us through the various techniques of rendering only the visible surfaces.

66.0 Objectives:

- To determine which of the various objects that project to the same pixel is closest to the viewer and hence is displayed.
- To determine which objects are visible to the eye and what colors to use to paint the pixels

The unit will address the surfaces we cannot see and their elimination methods:

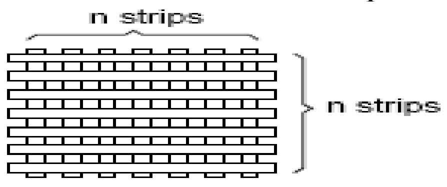
- Occluded surfaces: hidden surface removal (visibility).
- Back faces: back face culling
- Faces outside view volume: viewing frustrum culling

67.0 Algorithm Types

3.0.1 Object precision: The algorithm computes its results to machine precision (the precision used to represent object coordinates). The resulting image may be enlarged many times without significant loss of accuracy. The output is a set of visible object faces, and the portions of faces that are only partially visible.

3.0.2 Image precision: The algorithm computes its results to the precision of a pixel of the image. Thus, once the image is generated, any attempt to enlarge some portion of the image will result in reduced resolution.

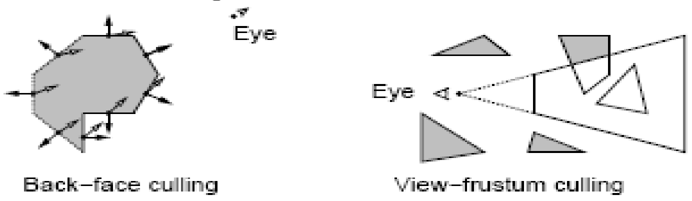
Although image precision approaches have the obvious drawback that they cannot be enlarged without loss of resolution, the fastest and simplest algorithms usually operate by this approach.
The hidden-surface elimination problem for object precision is interesting from the perspective of algorithm design, because it is an example of a problem that is rather hard to solve in the worst-case, and yet there exists a number of fast algorithms that work well in practice. As an example of this, consider a patch-work of *n* thin horizontal strips in front of *n* thin vertical strips.



Worst-case example for hidden-surface elimination.

3.1 Culling:

Before performing a general hidden surface removal algorithm, it is common to first apply heuristics to remove objects that are obviously not visible. This process is called *culling*. There are three common forms of culling.



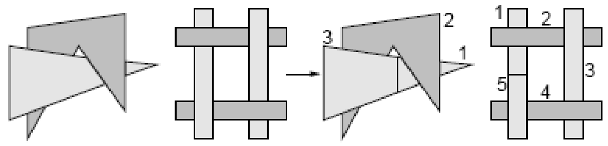
3.1.1 Back-face Culling: This is a simple trick, which can eliminate roughly half of the faces from consideration. Assuming that the viewer is never inside any of the objects of the scene, then the back sides of objects are never visible to the viewer, and hence they can be eliminated from consideration.
For each polygonal face, we assume an outward pointing normal has been computed. If this normal is directed *away* from the viewpoint, that is, if its dot product with a vector directed towards the viewer is negative, then the face can be immediately discarded from consideration. 3.1.2 **View Frustum Culling:** If a polygon does not lie within the view frustum (recall from the lecture on perspective), that is, the region that is visible to the viewer, then it does not need to be rendered. This automatically eliminates polygons that lie behind the viewer.
This amounts to clipping a 2-dimensional polygon against a 3-dimensional frustum. The Liang-Barsky clipping algorithm can be generalized to do this.

3.1.3 Visibility Culling: Sometimes a polygon can be culled because it is "known" that the polygon cannot be visible, based on knowledge of the domain. For example, if you are rendering a room of a building, then it is reasonable to infer that furniture on other floors or in distant rooms on the same floor are not visible. This is the hardest type of culling, because it relies on knowledge of the environment. This information is typically precomputed, based on expert knowledge or complex analysis of the environment.

3.2 Depth-Sort Algorithm: A fairly simple hidden-surface algorithm is based on the principle of painting objects from back to front, so that more distant polygons are overwritten by closer polygons. This is called the *depthsort algorithm*. This suggests the following algorithm: sort all the polygons according to increasing distance from the viewpoint, and then scan convert them in reverse order (back to front). This is sometimes called the*painter's algorithm* because it mimics the way that oil painters usually work (painting the background before the foreground). The painting process involves setting pixels, so the algorithm is an image precision algorithm.

There is a very quick-and-dirty technique for sorting polygons, which unfortunately does not generally work.

Compute a *representative point* on each polygon (e.g. the centroid or the farthest point to the viewer). Sort the objects by decreasing order of distance from the viewer to the representative point (or using the pseudodepth which we discussed in discussing perspective) and draw the polygons in this order. Unfortunately, just because the representative points are ordered, it does not imply that the entire polygons are ordered. Worse yet, it may be *impossible* to order polygons so that this type of algorithm will work. The Fig. blow shows such an example, in which the polygons overlap one another cyclically.



Hard cases to depth-sort.

In these cases we may need to *cut* one or more of the polygons into smaller polygons so that the depth order can be uniquely assigned. Also observe that if two polygons do not overlap in *x; y* space, then it does not matter what order they are drawn in.

Here is a snapshot of one step of the depth-sort algorithm. Given any object, define its *z-extents* to be an interval along the *z*-axis defined by the object's minimum and maximum *z*-coordinates. We begin by sorting the polygons by depth using farthest point as the representative point, as described above. Let's consider the polygon *P* that is currently at the end of the list. Consider all polygons *Q* whose *z*-extents overlaps *P*'s. This can be done by walking towards the head of the list until finding the first polygon whose maximum *z*-coordinate is less than *P*'s minimum *z*-coordinate. Before drawing *P* we apply the following tests to each of these polygons *Q*. If any answers is "yes", then we can safely draw *P* before *Q*.

(1) Are the *x*-extents of *P* and *Q* disjoint?

(2) Are the *y*-extents of *P* and *Q* disjoint?

(3) Consider the plane containing *Q*. Does *P* lie entirely on the opposite side of this plane from the viewer?

(4) Consider the plane containing *P*. Does *Q* lie entirely on the same side of this plane from the viewer?

(5) Are the projections of the polygons onto the view window disjoint?

In the cases of (1) and (2), the order of drawing is arbitrary. In cases (3) and (4) observe that if there is any plane with the property that *P* lies to one side and *Q* and the viewer lie to the other side, then *P* may be drawn before *Q*. The plane containing *P* and the plane containing *Q* are just two convenient planes to test. Observe that tests (1) and (2) are very fast, (3) and (4) are pretty fast, and that (5) can be pretty slow, especially if the polygons are nonconvex.

If all tests fail, then the only way to resolve the situation may be to split one or both of the polygons. Before doing this, we first see whether this can be avoided by putting *Q* at the end of the list, and then applying the process on *Q*. To avoid going into infinite loops, we mark each polygon once it is moved to the back of the list.

Once marked, a polygon is never moved to the back again. If a marked polygon fails all the tests, then we need to split. To do this, we use *P*'s plane like a knife to split *Q*. We then take the resulting pieces of *Q*, compute the farthest point for each and put them back into the depth sorted list.

In theory this partitioning could generate *O*(*n*2) individual polygons, but in practice the number of polygons is much smaller. The depth-sort algorithm needs no storage other than the frame buffer and a linked list for storing the polygons (and their fragments). However, it suffers from the deficiency that each pixel is written as many times as there are overlapping polygons.

3.3 Depth-buffer Algorithm: The *depth-buffer algorithm* is one of the simplest and fastest hidden-surface algorithms.

Its main drawbacks are that it requires a lot of memory, and that it only produces a result that is accurate to pixel resolution and the resolution of the depth buffer. Thus the result cannot be scaled easily and edges appear jagged (unless some effort is made to remove these effects called "aliasing"). It is also called the *z-buffer algorithm* because the *z*-coordinate is used to represent depth. This algorithm assumes that for each pixel we store two pieces of information, (1) the color of the pixel (as usual), and (2) the depth of the object that gave rise to this color. The depth-buffer values are initially set to the maximum possible depth value.

Suppose that we have a *k*-bit depth buffer, implying that we can store integer depths ranging from 0 to $D = 2k - 1$. After applying the perspective-with-depth transformation (recall Lecture 12), we know that all depth values have been scaled to the range $[-1; 1]$. We scale the

depth value to the range of the depth-buffer and convert this to an integer, e.g. $b(z + 1)=(2D)c$. If this depth is less than or equal to the depth at this point of the buffer, then we store its RGB value in the color buffer. Otherwise we do nothing.
This algorithm is favored for hardware implementations because it is so simple and essentially reuses the same algorithms needed for basic scan conversion.

Z-Buffering: Algorithm

```
allocate z-buffer; // Allocate depth buffer →Same size as viewport.
for each pixel (x,y) // For each pixel in viewport.
writePixel(x,y,backgrnd); // Initialize color.
writeDepth(x,y,farPlane); // Initialize depth (z) buffer.
for each polygon // Draw each polygon (in any order).
for each pixel (x,y) in polygon // Rasterize polygon.
pz = polygon's z-value at (x,y); // Interpolate z-value at (x, y).
if (pz < z-buffer(x,y)) // If new depth is closer:
writePixel(x,y,color); // Write new (polygon) color.
writeDepth(x,y,pz); // Write new depth.
Note: This assumes' you've negated the z values!right edges.
```

Advantages:

Easy to implement in hardware (and software!)
Fast with hardware support Fast depth buffer memory
Hardware supported
Process polygons in arbitrary order
Handles polygon interpenetration trivially

Disadvantages:

Lots of memory for z-buffer:
Integer depth values
Scan-line algorithm
Prone to aliasing
Super-sampling
Overhead in z-checking: requires fast memory

68.0 Conclusion

There are several approaches to drawing only the things that should be visible in a 3D scene. The "painter's algorithm" says to sort the objects by distance from the camera, and draw the farther things first, and the nearer ones on top ("painting over") the farther ones. This approach may be too inefficient We need an approach that removes the hidden parts mathematically before submitting primitives for rendering.
z-buffers, or depth buffers, which are extra memory buffers, to track different objects' relative depths.

69.0 Summary

Elements that are closer to the camera obscure more distant ones. We needed to determine which surfaces are visible to the eye, those which are not, and what colors to use to paint the pixels.
The following techniques were applied to eliminate the back surfaces.

- The Painter's algorithm; to draw visible surfaces from back (farthest) to front (nearest) by applying back-face culling, depth-sort, BSP tree.
- Image precision algorithms; to determine which object is visible at each pixel by applying z-buffering, ray tracing

70.0 Tutor Marked Assignment

1. Specify whether the following visibility algorithms operate with object-space precision or image-space (pixel) precision by writing the letter 'O' or 'I' next to each.
(a) back-face culling:
(b) depth-sort:
(c) z-buffer:
(d) ray casting:
2. what do you understand by the term culling? Write on the various culling methods.

71.0 Refrences/Further reading:

A. Watt, 3D Computer Graphics, 3rd Ed., Addison-Wesley, 2000.
J.D. Foley, A. Van Dam, et al., Computer Graphics: Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.
D. Hearn, M.P. Baker, Computer Graphics, 2nd Ed. in C, Prentice-Hall, 1996.
P.A. Egerton, W.S. Hall, Computer Graphics - Mathematical First Steps, Prentice Hall, 1998.