



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

**CPT205 Computer Graphics**

**General Introduction**

**Hardware and Software**

**Week 01**

**2021-22**

**Yong Yue**

# What is Computer graphics?

‘Computer Graphics’ is concerned with all aspects of producing pictures or images using a computer. There are three closely related meanings, each representing a different perspective on the same thing.

- the images that you see on the computer screen
- the computer code that is used to create the images
- a mathematical model of the real-world (which is sometimes called the virtual world)

When working at the most advanced levels of Computer Graphics, the computer graphics specialist will

- create a virtual world
- implement the virtual world in computer code
- run the code to see life-like images on the computer screen

# How do the individual subjects relate?

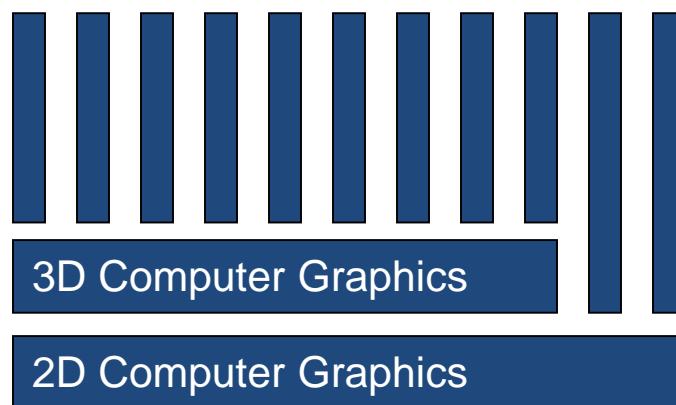
There are two subjects that are at the heart of computer graphics.

These are:

- 2D Computer Graphics, and
- 3D Computer Graphics

These provide a foundation of knowledge and skills that is essential for working in any area related to graphics.

The application areas



# What are the application areas?

The ‘application areas’ are the different types of computer application that can be produced using computer graphics technology. They are also the technical areas within which a student may seek to work when he or she leaves university with a relevant degree.

Example application areas:

- Display of information
- Design
- Simulation and animation
- User interfaces

# Why learn computer graphics?

There are numerous good reasons. These include:

- the application areas are exciting;
- the subject matter is intellectually stimulating and the knowledge that you will get on the course is relatively rare;
- computer graphics is a major facet of computer science (e.g. try to think of a computer program that does not involve some kind of graphics);
- the concepts associated with computer graphics are time independent (i.e. what you learn now will still be valid in the future, unlike some programming languages which disappear with passing time);
- computer programming is the type of job in computing that is in great demand, and the computer graphics course involves a good deal of “hands on” programming.

# Basic concepts of computer graphics

- Graphics hardware and software
- Fundamental mathematics
- Objects / geometric primitives – point, curve, surface and solids
- Modelling and representation schemes
- Geometric transformations
- Viewing and projections
- Clipping
- Removal of hidden curves and surfaces
- Lighting and materials
- Texture mapping
- Animation
- Programming and applications

# This module

Introduces a wide range of topics in computer graphics and its applications, providing you with both fundamental theory and hands-on experience through lab-based practice and assessment. It follows a standard textbook with additional materials used for contemporary developments and applications.

In terms of learning outcomes, you will be able to:

- demonstrate a good understanding of topics and applications in computer graphics covered in the module;
- demonstrate an in-depth knowledge of geometric creation and transformation, projection, clipping and hidden geometry removal, lighting and materials, and texture mapping;
- apply relevant techniques / algorithms covered in the module to specific scenarios;
- write programming code in conjunction with a popular graphics platform (e.g. OpenGL).

# Delivery and assessment

**Delivery:** You will have a two-hour formal lecture followed by a two-hour lab weekly. It assumes knowledge of matrices and vectors and previous experience of computer programming in a high-level procedural language (e.g. Java or C). Sample programs will be provided during the lab sessions.

## Assessment:

- |                                     |                           |
|-------------------------------------|---------------------------|
| 1) Assessment 1 (2D project – 15%): | Out Week 4 and due Week 8 |
| 2) Assessment 2 (3D project – 15%): | Weeks 9 to 13             |
| 3) Assessment 3 (Final exam – 70%): | Early January 2022        |
| 4) Resit Assessment (Exam – 100%):  | Early August 2022         |

No resit is allowed for assessments 1 and 2. If a student does not obtain an overall grade of 40% or higher for Assessments 1-3, she/he will be required to take the resit exam which will have a weighting of 100% for the module (regardless of grades of assessments 1 and 2).

***Important:*** *Plagiarism is a serious academic offence and will not be tolerated. Copying from other sources without appropriate acknowledgement may result in plagiarism! If in doubt, consult relevant members of academic staff.*

# Marking scheme of coursework

Category	Requirements (each category builds on the requirements for the preceding category)
<b>First Class (≥70%)</b>	<p>Overall outstanding work. Very neat program implements effectively all the graphics techniques covered.</p> <p>Artefact produced with realistic / real-life content and visual effect.</p> <p>Well-structured and concise written report providing all the required information.</p>
<b>Second Upper (60 to 69%)</b>	<p>Comprehensive program that utilises effectively the full range of the graphics techniques covered to date. Good commenting and layout of the program.</p> <p>An impressive artefact produced with a good range of features achieved by calling appropriate OpenGL functions.</p> <p>A comprehensive and clear report containing all required information within the page limit.</p>
<b>Second Lower (50 to 59%)</b>	<p>Substantial working program implements a good range of graphics techniques.</p> <p>Nice layout and objects in the artefact.</p> <p>Written report contains all the information of the features and functions of the program including some screenshots.</p>
<b>Third (40 to 49%)</b>	<p>Working program that generates a recognisable artefact with some objects and a limited range of the graphics techniques utilised.</p> <p>Written report provides a good overview and describes all the basic information for the work completed.</p>
<b>Fail (0 to 39%)</b>	<p>Some code produced attempts to the use of some graphics techniques covered in the module.</p> <p>No or very limited artefact produced.</p> <p>Written report covers very limited number of the items required in the assignment brief, acknowledging properly sources used if any.</p>
<b>Non-submission</b>	A mark of 0 will be awarded.

# Schedule and Module Team

## □ Delivery Schedule

Lecture (All groups): Tuesday 09:00-11:00

Online (Weeks 1-4)

SD102 (Weeks 5-6 and 8-14)

Lab (Groups 1 & 3): Wednesday 09:00-11:00

Online (Weeks 1-4)

SC464 (Group 1) / SD446E (Group 3) (Weeks 5-6 and 8-14)

Lab (Group 2): Thursday 11:00-13:00

Online (Weeks 1-4)

SC464 (Weeks 5-6 and 8-14)

## □ Module Leader and Contact Details

Name: Yong Yue

Email: [yong.yue@xjtu.edu.cn](mailto:yong.yue@xjtu.edu.cn)

Office telephone number: (0512) 8816 1503

Room number: SD523

Office hours: 16:00-17:00 Tuesday and 17:00-18:00 Wednesday

Preferred means of contact: email

# Schedule and Module Team

- **Teaching Assistants (TAs)**

*Lei Chen ([lei.chen02@xjtu.edu.cn](mailto:lei.chen02@xjtu.edu.cn))*

*Shanliang Yao ([shanliang.yao19@student.xjtu.edu.cn](mailto:shanliang.yao19@student.xjtu.edu.cn))*

*Yijie Chu ([yijie.chu16@student.xjtu.edu.cn](mailto:yijie.chu16@student.xjtu.edu.cn))*

*Ruben Ng ([r.ng19@student.xjtu.edu.cn](mailto:r.ng19@student.xjtu.edu.cn))*

*Yuanzhe Liu ([yuanzhe.liu17@student.xjtu.edu.cn](mailto:yuanzhe.liu17@student.xjtu.edu.cn))*

*Xingshuo Li ([xingshuo.li17@student.xjtu.edu.cn](mailto:xingshuo.li17@student.xjtu.edu.cn))*

*Zimimg Li ([ziming.li17@student.xjtu.edu.cn](mailto:ziming.li17@student.xjtu.edu.cn))*

*Depending on the demand for support later on, students may be divided into groups to receive support from dedicated TAs.*

# Textbooks

- Edward Angel and Dave Shreiner,  
*Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*, Sixth Edition,  
ISBN: 978-7121177095, 1 July 2012,  
Electronic Industry Press, China.
  
- Dave Shreiner, Graham Sellers, John M. Kessenich and Bill M. Licea-Kane,  
*OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Versions 4.3,  
ISBN: 978-0321773036,  
20 March 2013, Addison Wesley.



# What's next?

**To do well on the course:**

- try to enjoy yourself
- attend all lecture and practical sessions
- work consistently



# Hardware and Software

- **Graphics Hardware**

- Input, Processing and Output Devices
- Framebuffers
- Pixels and Screen Resolution

- **Graphics Software**

- Techniques (Algorithms, Procedures)
- Programming Library / API (OpenGL, JOGL and so on)
- Not our focus: High level Interactive Systems (Maya, Studio Max, Unity, AutoCAD and so on)

# Thank about

- Basic
  - How will you define graphics hardware?
  - Does graphics hardware involve input, processing and output?
  - When you buy a computer how will you specify the graphics requirement?
- Intermediate
  - What is the graphics board on your home desktop computer?
  - How many lines/sec or triangles/sec can you draw on your computer?
  - Screen resolution: How many pixels do you see on your screen?
- Advanced
  - How do you represent a scene (a house) graphically in your computer?
  - How do you display the house on the computer screen (framebuffer)?
  - How does the graphics hardware transform your scene to the display?

# Graphics devices

- Input Devices
- Processing Devices
- Output Devices

How do we link the three?



# Output devices



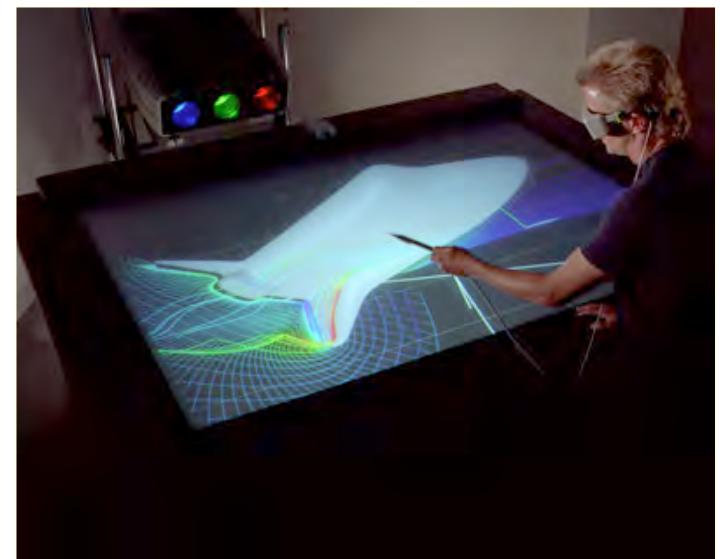
CRT



Laptop LCD



LCD

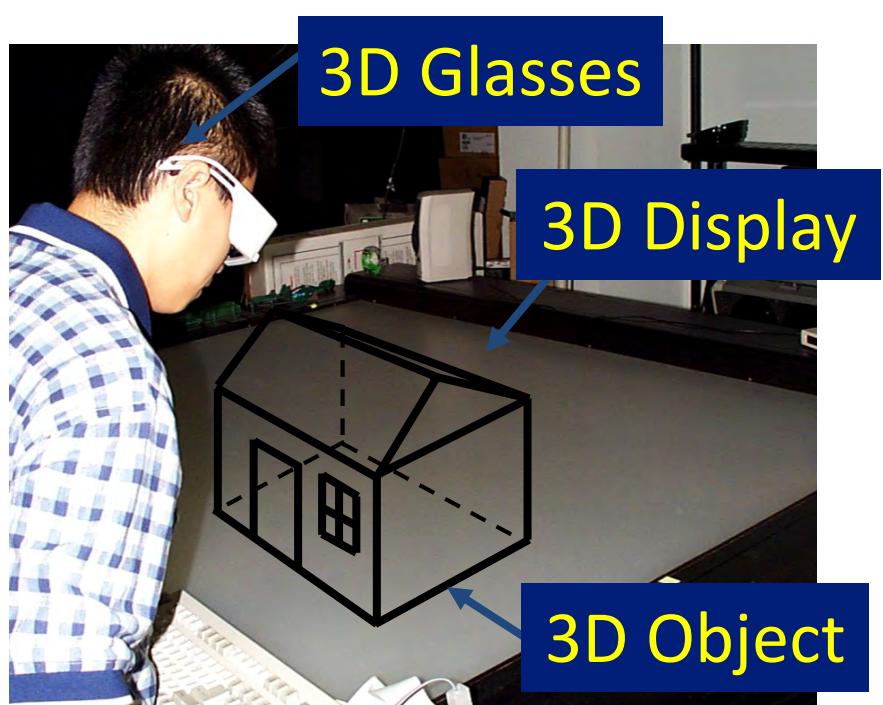


Projection Table

# Graphic display in virtual reality

- Head-Mounted Displays (HMDs)
  - The display and a position tracker are attached to the user's head
- Head-Tracked Displays (HTDs)
  - Display is stationary, tracker tracks the user's head relative to the display.
  - Example: CAVE, Workbench, Stereo monitor





# Graphic processing unit (GPU)

- Graphics Boards / GPUs sit inside



# Input devices

- Enables graphical interaction



# Input devices

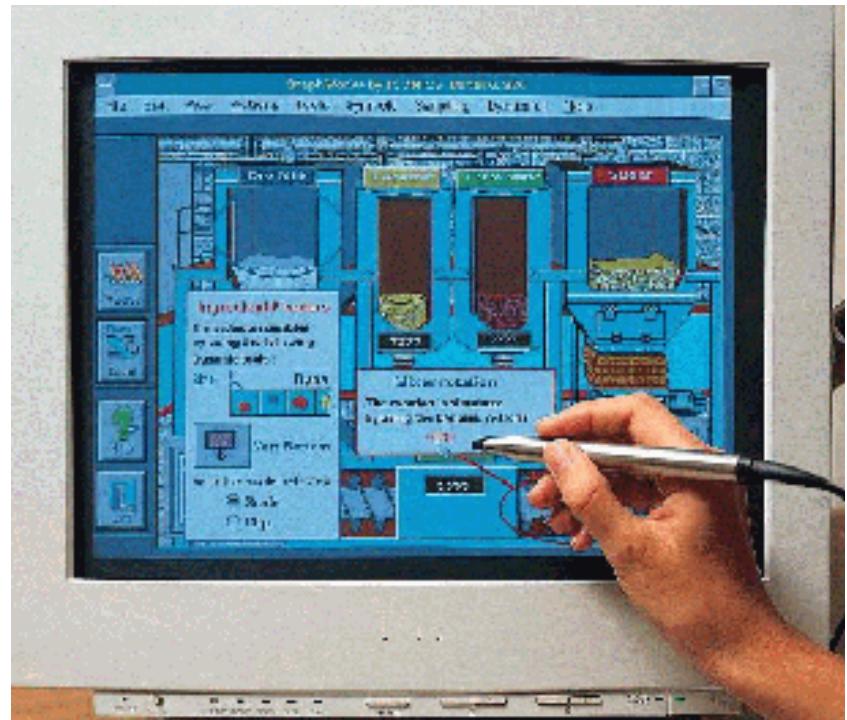
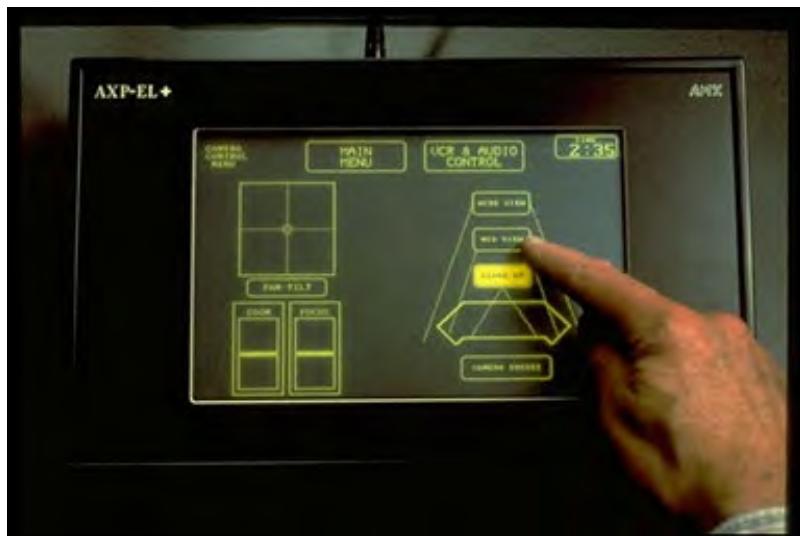
When queried, locator devices return a position and/or orientation

- Tablets
- Virtual Reality Trackers
  - Data Gloves
  - Digitisers



# Input devices

- Light Pens
- Voice Systems
- Touch Panels
- Camera/Vision based
- Which is best?



# Input devices for games

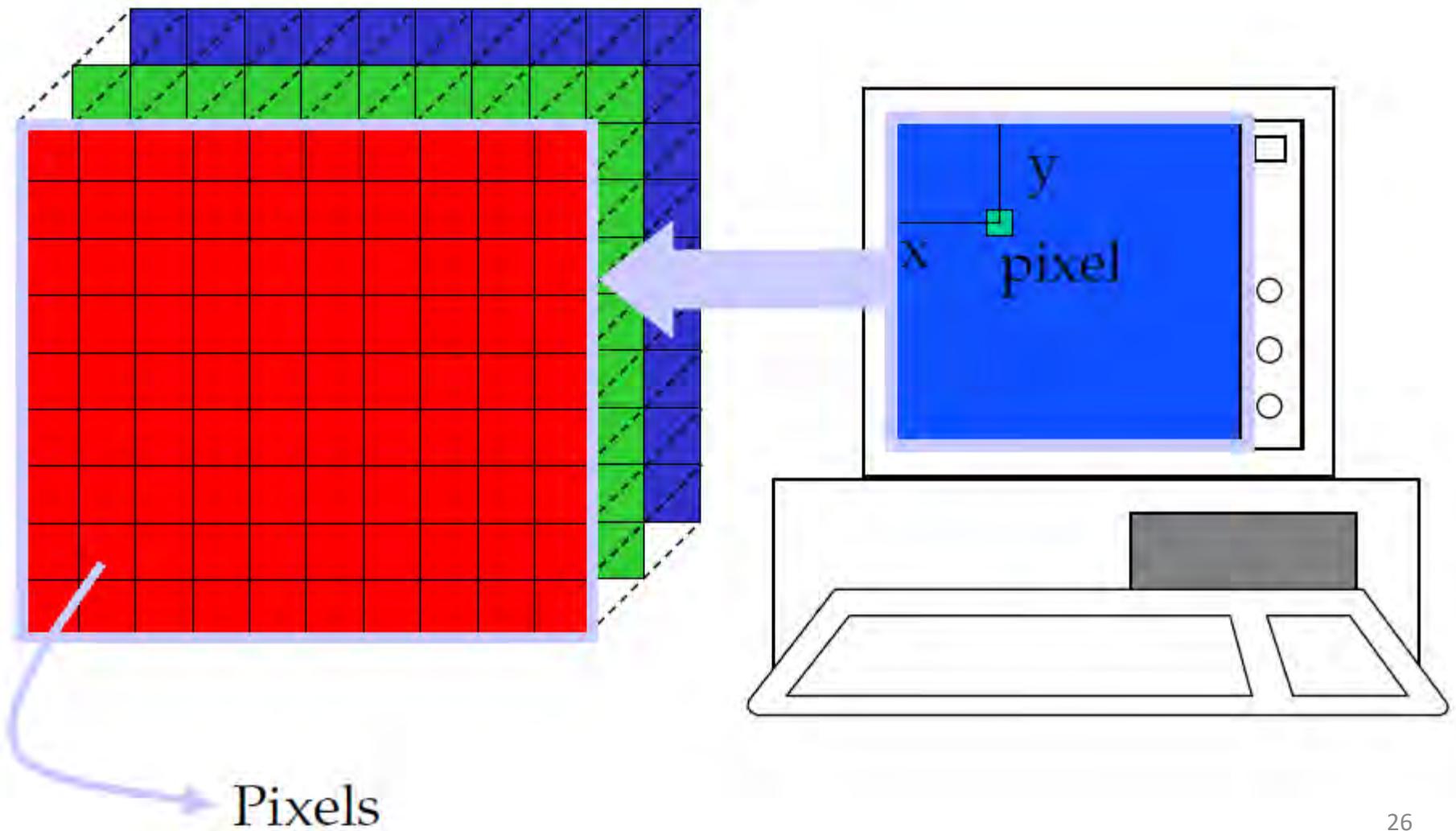


# Framebuffer

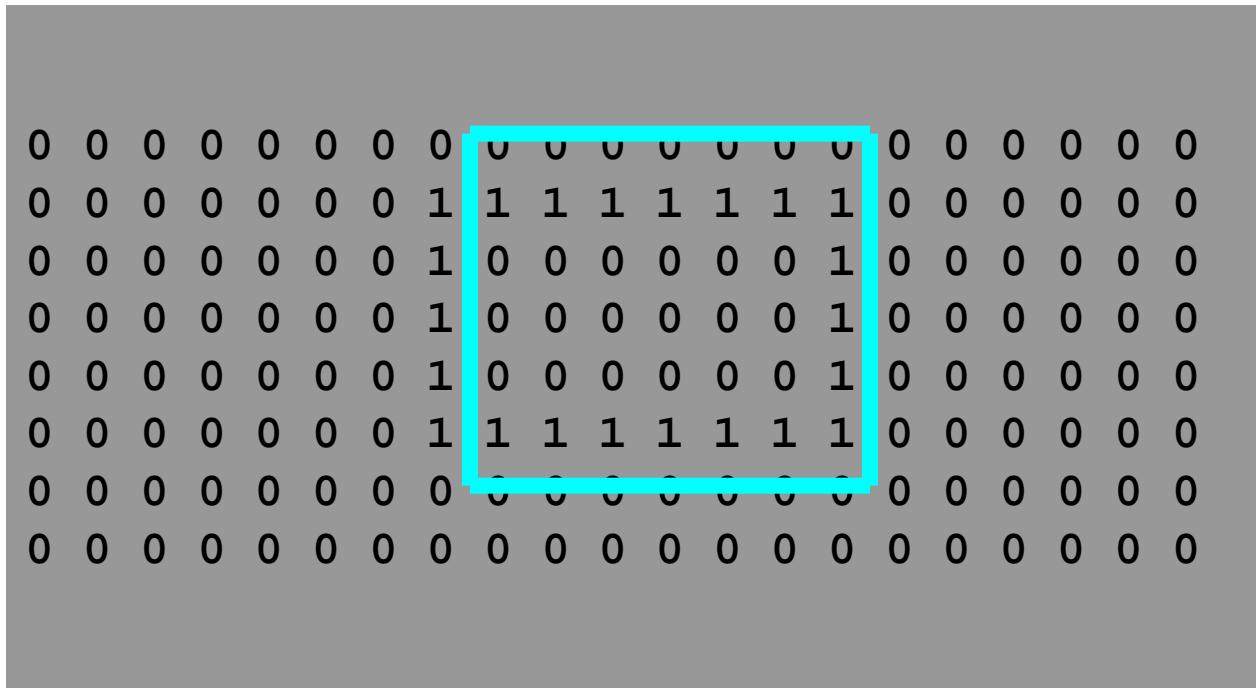
**Framebuffer** – A block of memory, dedicated to graphics output, that holds the contents of what will be displayed.

**Pixel** - an element of the framebuffer.

# Framebuffer



# Framebuffer



- Questions:**
- How many pixels are there?
  - What is the largest image you can display?
  - How big is the framebuffer?
  - How much memory do we need to allocate for the framebuffer?

# Framebuffer in memory

- If we want a framebuffer of 640 pixels by 480 pixels, we should allocate:

$$\text{framebuffer} = 640 * 480 \text{ bits}$$

- How many bits should we allocate?

Q: What do more bits get you?

A: More values to be stored at each pixel.

Why would you want to store something other than a 1 or 0?

# Framebuffer bit depths

- How many colours does 1 bit get you?
- How many colours do 8 bits get you?
  - Monochrome systems use this (green/grey scale)
- What bit depth would you want for your framebuffer?

**bit depth** - number of bits allocated per pixel in a buffer.

# Framebuffer bit depths

- Remember, we are asking “how much memory we allocate to store the colour at each pixel?”
- Common answer:
  - 32 bits RGBA



# Framebuffer bit depths

- 32 bits per pixel (true colour)
  - 8 bits for red, green, blue and alpha
  - potential for 256 reds, greens and blues
  - total colours: 16,777,216 (more than the eye can distinguish)
- Let's look at Display Control Panel

# Data type refresher

bit - a 0 or 1. Can represent 2 unique values

byte - 8 bits or 256 values

word - 32 bits or 4,294,967,296 values

int - 32 bits

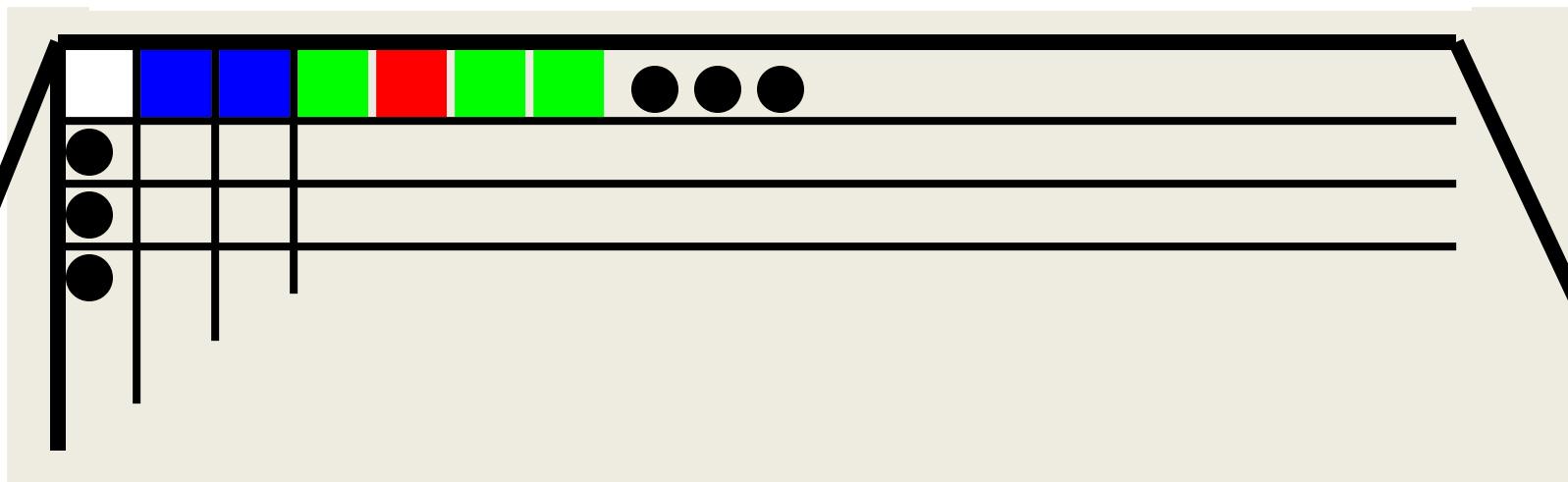
float - 32 bits

double - 64 bits

unsigned byte - 8 bits

# Framebuffer bit depths

unsigned byte framebuffer [640\*480\*3];



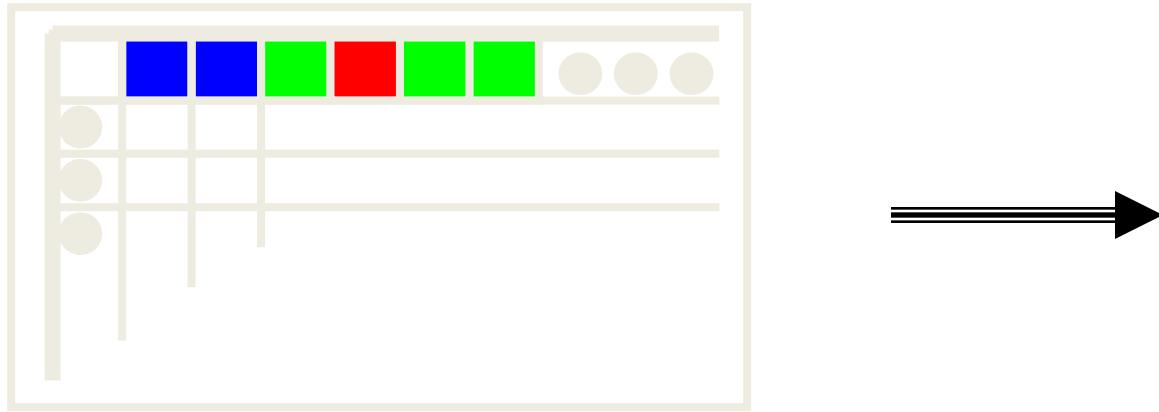
**framebuffer** =

```
[ 255 255 255 0 0 255 0 0 255 0 255 0 255 0 0  
 0 255 0 0 255 0 ... ]
```

# Graphic card memory

- How much memory is on our graphic card?
  - $640 * 480 * 32 \text{ bits} = 1,228,800 \text{ bytes}$
  - $1024 * 768 * 32 \text{ bits} = 3,145,728 \text{ bytes}$
  - $1600 * 1200 * 32 \text{ bits} = 7,680,000 \text{ bytes}$
- How much memory is on your graphics card?

# Framebuffer -> monitor



The values in the framebuffer are converted from a digital (1s and 0s representation, the bits) to an analog signal that goes out to the monitor. This is done automatically (not controlled by your code), and the conversion can be done while writing to the framebuffer.

# Image quality issues

- Screen resolution
- Colour
- Refresh rate
- Brightness
- Contrast
- Sensitivity of display to viewing angle

# Pixels

- Pixel - The most basic addressable image element in a screen
  - CRT - Colour triad (RGB phosphor dots)
  - LCD - Single colour element
- Screen Resolution - measure of number of pixels on a screen (m by n)
  - m - Horizontal screen resolution
  - n - Vertical screen resolution

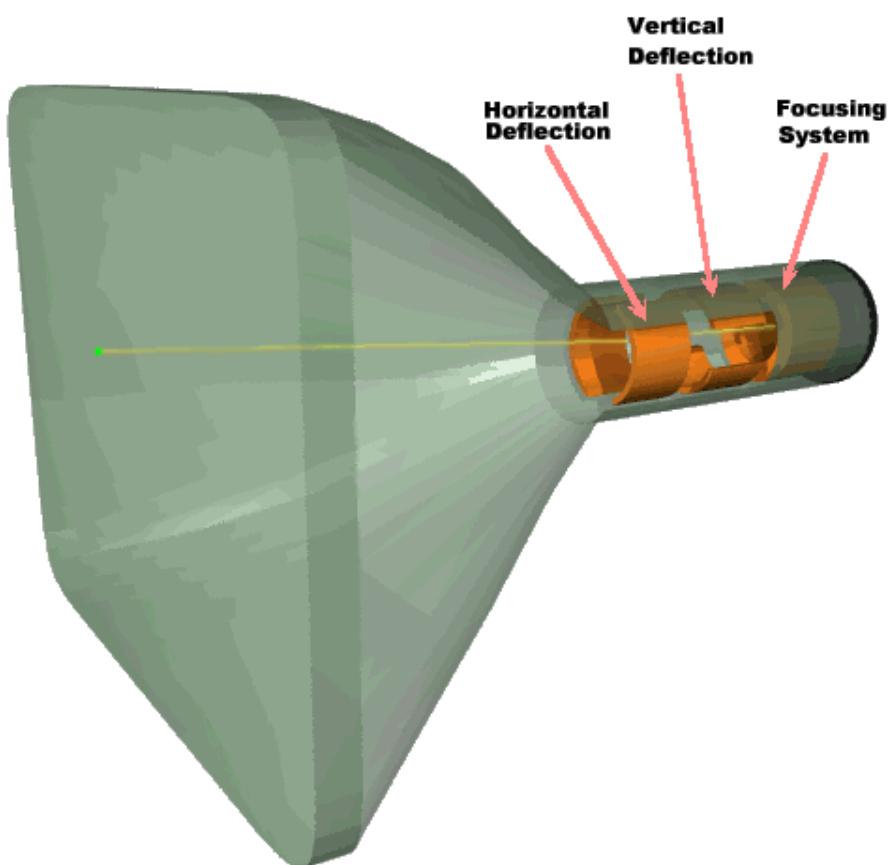
# Video formats

- NTSC - 525x480, 30f/s, interlaced
- PAL - 625x480, 25f/s, interlaced
- VGA - 640x480, 60f/s, non-interlaced
- SVGA - 800x600, 60f/s non-interlaced
- RGB - 3 independent video signals and synchronisation signal, vary in resolution and refresh rate
- Interlaced - scan every other line at a time, or scan odd and even lines alternatively; the even scan lines are drawn and then the odd scan lines are drawn on the screen to make up one video frame.

# Raster display

- Cathode Ray Tubes (CRTs), most “tube” monitors you might see. Used to be very common, but big and bulky.
- Liquid Crystal Displays (LCDs), there are two types transmissive (laptops, those snazzy new flat panel monitors) and reflective (wrist watches).

# Cathode ray tubes (CRTs)



Heating element on the yolk.

Phosphor coated screen

Electrons are boiled off the filament and drawn to the focusing system.

The electrons are focused into a beam and “shot” down the cylinder.

The deflection plates “aim” the electrons to a specific position on the screen.

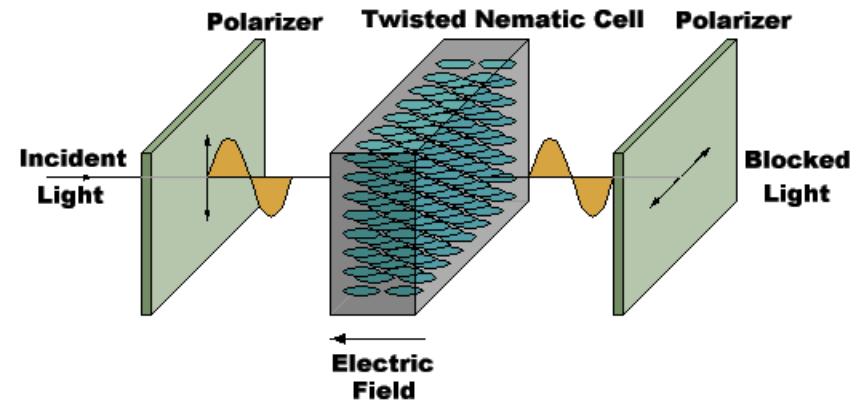
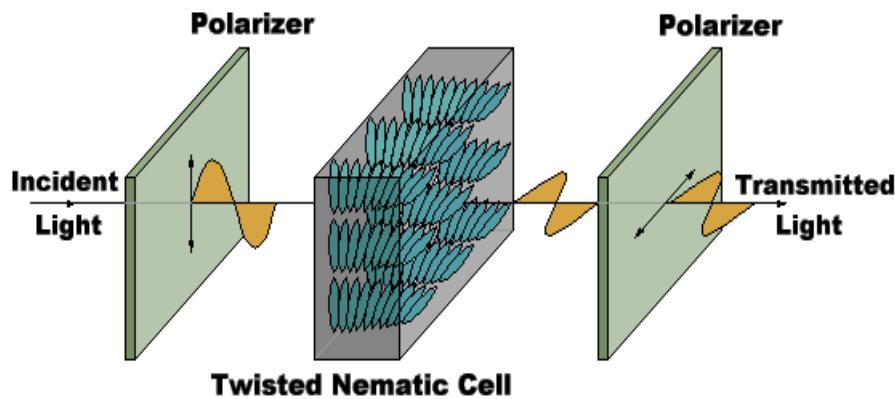
# Cathode ray tubes (CRTs)

- Strong electrical fields and high voltage
- Very good resolution
- Heavy, not flat



# Liquid crystal displays (LCDs)

- Also divided into pixels, but without an electron gun firing at a screen, LCDs have cells that either allow light to flow through, or block it.



# Liquid crystal displays (LCDs)

- Liquid crystal displays use small flat chips which change their transparency properties when a voltage is applied.
- LCD elements are arranged in an  $n \times m$  array called the LCD matrix.
- The level of voltage controls grey levels.
- LCDs elements do not emit light; use backlights behind the LCD matrix.
- Colour is obtained by placing filters in front of each LCD element.
- Image quality dependent on viewing angle.

# Advantages of LCDs

- Flat
- Light weight
- Low power consumption



# What is graphics software?

- Graphics drivers
- Graphics libraries
- Graphics editors
- Geometric modellers
- VR modellers
- Games
- Scientific visualisation packages
- ...

# Graphics software

- How to talk to the hardware?  
Algorithms, Procedures, Toolkits & Packages  
(Low Level  High Level)
- Programming API (**helps to program, for our labs**)
  - OpenGL (**our focus**)
  - JOGL (Open GL for Java)
  - OpenCV
  - Direct X, ...
- Special purpose software (**not our focus**)
  - Excel, Matlab, ...
  - AutoCAD, Studio Max, Unity, Maya, ...
  - Medical visualisation, modelling, ...

# OpenGL



- First introduced in 1992.
- The OpenGL graphics system is a software interface to graphics hardware (GL stands for Graphics Library).
- For interactive programs that produce colour images of moving three-dimensional objects.
- It consists of over 200 distinct commands that you can use to specify the objects and operations needed to produce interactive three-dimensional applications.



# OpenGL



- OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms.
- Similarly, OpenGL does not provide high-level commands for describing models of three-dimensional objects.
- With OpenGL, you must build up your desired model from a small set of geometric primitives - points, lines, and polygons.
- With OpenGL, you can control computer-graphics technology to produce realistic pictures or ones that depart from reality in imaginative ways.
- OpenGL has become the industry standard for graphics applications and games.

# Summary

- These are interesting times for computer graphics.
  - Commodity graphics cards are highly capable.
  - New algorithms, long-offline algorithms are becoming possible.
  - Hard to keep up, even for “experts”.
- What’s pushing the technology curve?
  - Games
  - Movies
  - Interactive graphics applications

# Topics covered today and ...

- Computer Graphics and its main topics
- Graphics Hardware
  - Input, Processing and Output Devices
  - Framebuffers
  - Pixels and Screen Resolution
- Graphics Software
  - Techniques (Algorithms, Procedures)
  - Programming Library / API (OpenGL, JOGL, OpenCV, DirectX)
  - **Not our focus:** High level Interactive Systems (Maya, Studio Max, Unity, AutoCAD, and so on)
- Think about ...
  - What are possible bottlenecks in system performance of a graphics system?
  - How can you achieve realism in a graphics system?
  - How do we combine the software and hardware to maximise performance?



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

## **CPT205 Computer Graphics**

# **Mathematics for Computer Graphics**

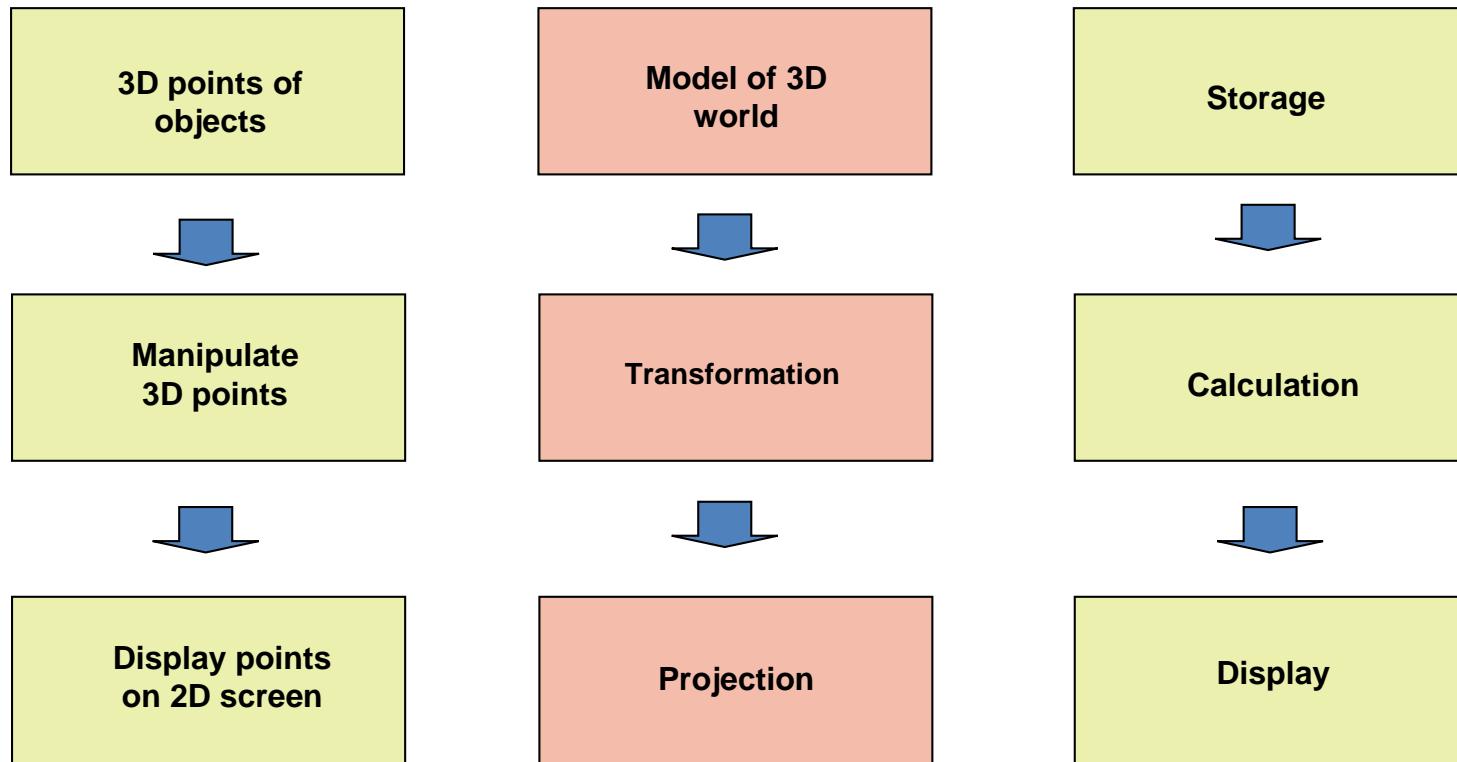
**Week 2  
2021-22**

**Yong Yue**

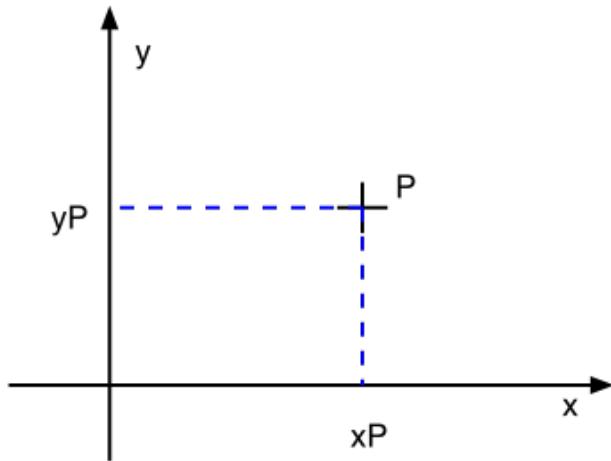
# Topics for today

- Computer representation of objects
- Cartesian co-ordinate system
- Points, lines and angles
- Trigonometry
- Vectors (unit vector) and vector calculations (addition, subtraction, scaling, dot product and cross product)
- Matrices (dimension, transpose, square/symmetric/identity and inverse) and matrix calculations (addition, subtraction and multiplication)

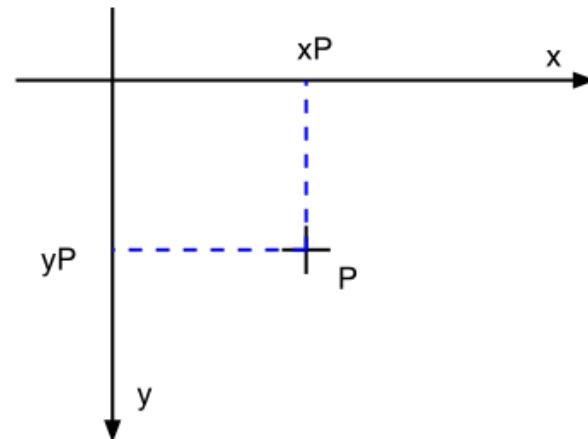
# Computer representation of objects



# Cartesian co-ordinate system

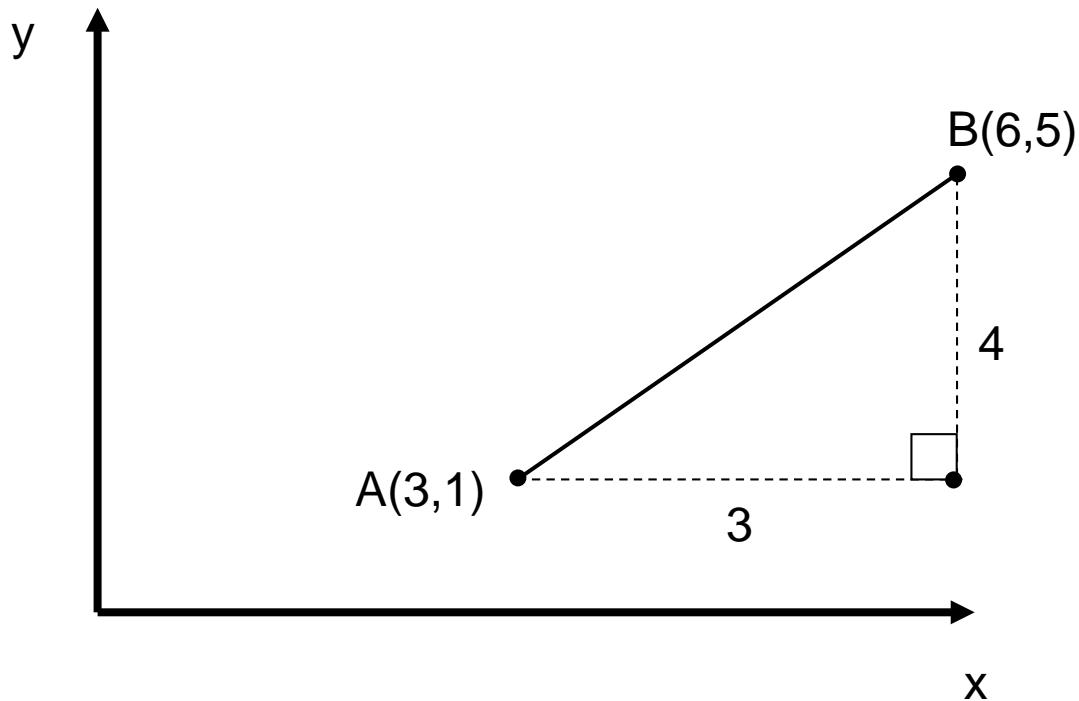


Representation of point  
 $P(x_p, y_p)$  in Cartesian co-ordinates



Representation of point  
 $P(x_p, y_p)$  on computer screen

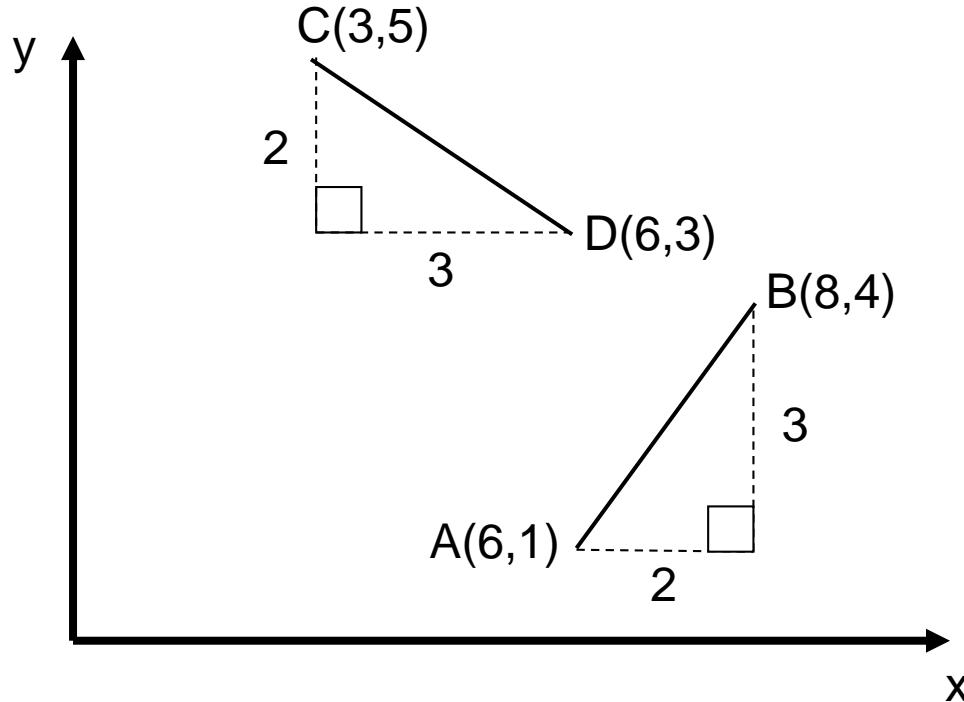
# Straight line



Using Pythagoras theorem:

$$AB = \sqrt{4^2 + 3^2} = 5$$

# Gradient of a line



$$\text{Gradient } AB = \Delta y / \Delta x = (4-1) / (8-6) = 3/2$$

$$\text{Gradient } CD = \Delta y / \Delta x = (3-5) / (6-3) = -(2/3)$$

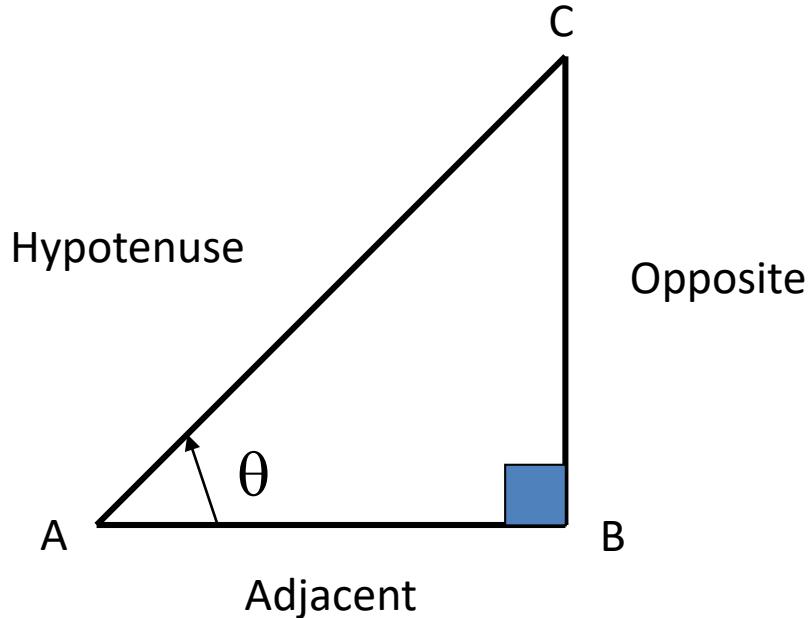
An uphill line (direction is ‘bottom left to top right’) has a **positive** gradient.

A downhill line (direction is ‘top left to bottom right’) has a **negative** gradient.

# Perpendicular lines

- Given that the gradient of AB= $3/2$  and gradient of CD= $-2/3$ , when the two gradients are multiplied together we have:  $(3/2) * (-2/3) = -1$ .
- Thus we, conclude that lines AB and CD are perpendicular.
- Prove this using graph paper.
- What can you say about lines with same gradient?

# Angles and trigonometry



$$\sin(\theta) = \text{Opposite} / \text{Hypotenuse} = BC / AC$$

$$\cos(\theta) = \text{Adjacent} / \text{Hypotenuse} = AB / AC$$

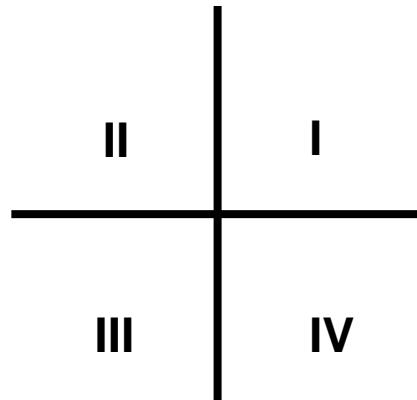
$$\tan(\theta) = \text{Opposite} / \text{Adjacent} = BC / AB$$

# Angles and trigonometry

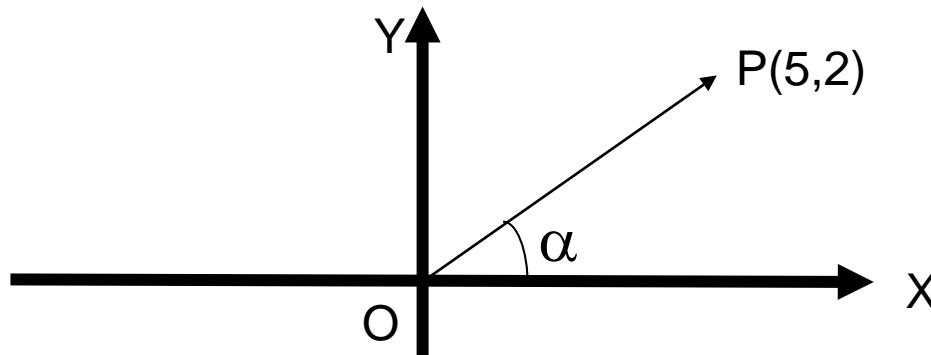
A complete revolution gives  $360^\circ$  or  $2\pi$  (rad).

The following diagram is used to find the values of the trigonometric ratios:

- All trigonometric ratios of angles in quadrant 1 have positive ratios.
- Only sine of angles in quadrant 2 have positive ratios.
- Only tangent of angles in quadrant 3 have positive ratios.
- Only cosine of angles in quadrant 4 have positive ratios.

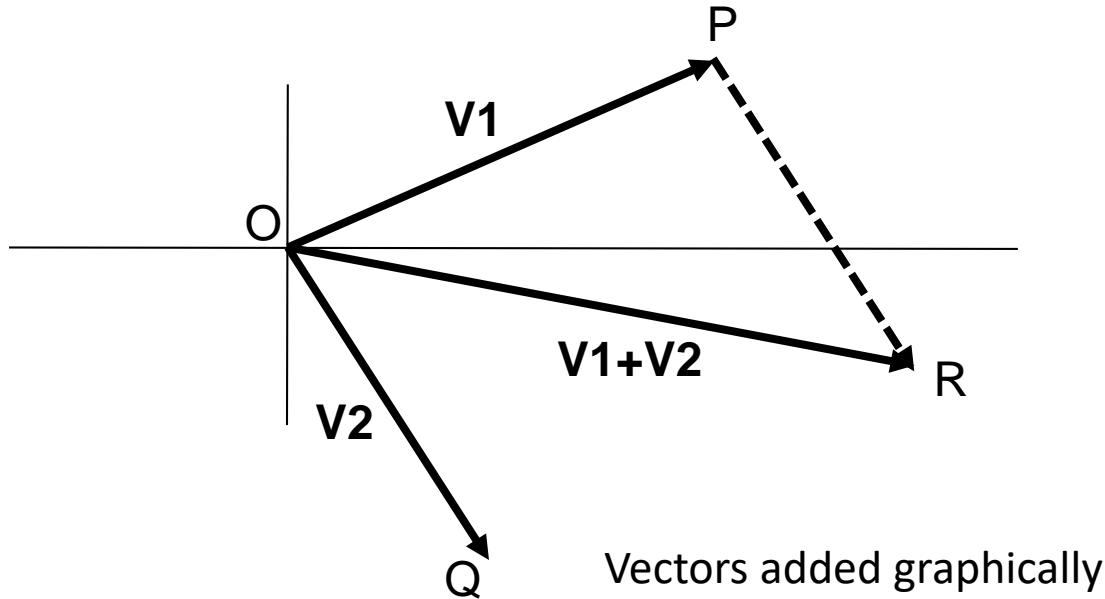


# Vectors



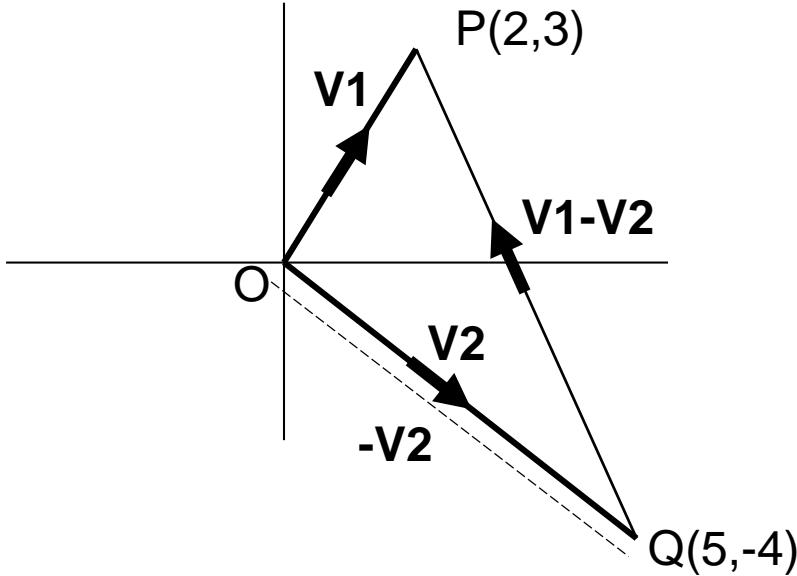
- $\mathbf{OP} = xi + yj$   
where **i** and **j** are **unit vectors** along the x- and y-axes, respectively.
- The magnitude or modulus of  $\mathbf{OP} = 5i + 2j$  is  
 $|\mathbf{OP}| = \sqrt{5^2 + 2^2} = 5.39$
- Unit vector of  $\mathbf{OP}$  is  
 $(\mathbf{OP}) = \mathbf{OP} / |\mathbf{OP}| = (5i + 2j) / 5.39 = 0.93i + 0.37j$
- $\sin(\alpha) = 2 / |\mathbf{OP}| = 2/5.39 = 0.37$   
 $\cos(\alpha) = 5 / |\mathbf{OP}| = 5/5.39 = 0.93$

# Vector addition



- For two vectors  $\mathbf{OP}$  and  $\mathbf{OQ}$  such as  
 $\mathbf{OP} = \mathbf{V1} = 5\mathbf{i} + 2\mathbf{j}$  and  $\mathbf{OQ} = \mathbf{V2} = 2\mathbf{i} - 4\mathbf{j}$
- A vector addition is the sum of vectors  $\mathbf{OP}$  and  $\mathbf{OQ}$   
$$\mathbf{V1} + \mathbf{V2} = (5\mathbf{i} + 2\mathbf{j}) + (2\mathbf{i} - 4\mathbf{j}) = 7\mathbf{i} - 2\mathbf{j}$$
- The direction of  $\mathbf{V1} + \mathbf{V2}$  with respect to the x-axis is  
$$\cos(\alpha) = 7 / |\mathbf{V1} + \mathbf{V2}| = 7 / \sqrt{7^2 + (-2)^2} = 0.962$$

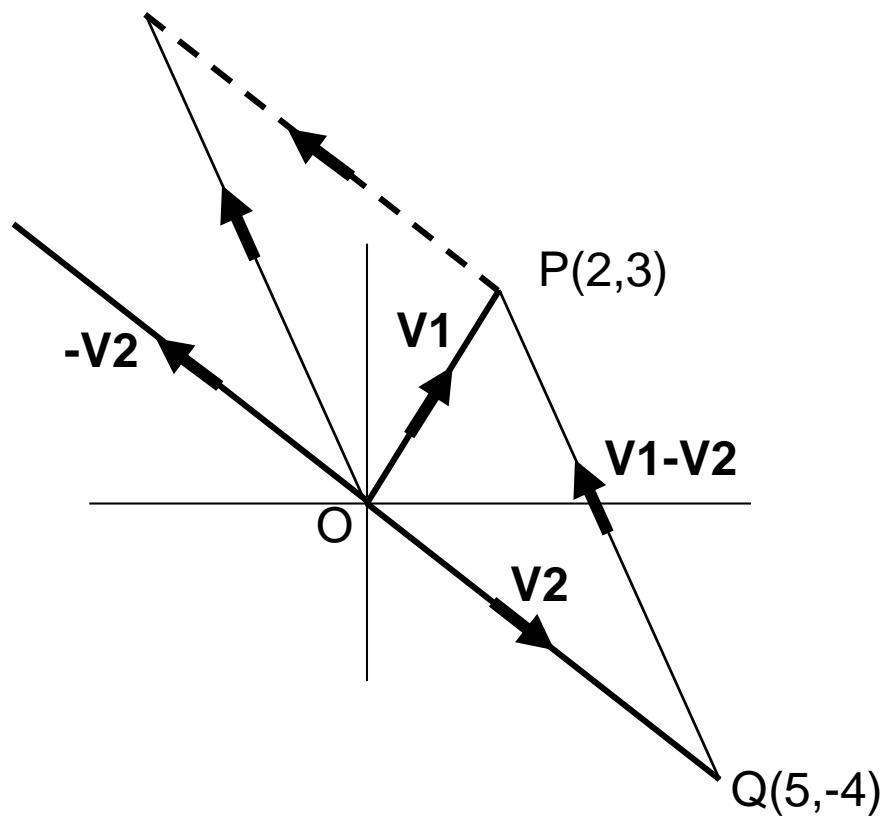
# Vector subtraction



Vectors subtracted  
graphically

- For two vectors  $\mathbf{OP}$  and  $\mathbf{OQ}$  such as  $\mathbf{OP} = \mathbf{V}_1 = 2\mathbf{i} + 3\mathbf{j}$  and  $\mathbf{OQ} = \mathbf{V}_2 = 5\mathbf{i} - 4\mathbf{j}$
- $\mathbf{V}_1 - \mathbf{V}_2 = (2\mathbf{i} + 3\mathbf{j}) - (5\mathbf{i} - 4\mathbf{j}) = -3\mathbf{i} + 7\mathbf{j}$
- The direction of  $\mathbf{V}_1 - \mathbf{V}_2$  with respect to the x-axis is  $\cos(\alpha) = -3 / |\mathbf{V}_1 + \mathbf{V}_2| = -3 / \sqrt{(-3)^2 + 7^2} = -0.394$

# Vector subtraction



Vectors subtracted graphically

# Vector scaling

A vector may be scaled up or down by multiplying it with a scalar number. Assume the following vector

$$\mathbf{V} = 4\mathbf{i} + 3\mathbf{j}$$

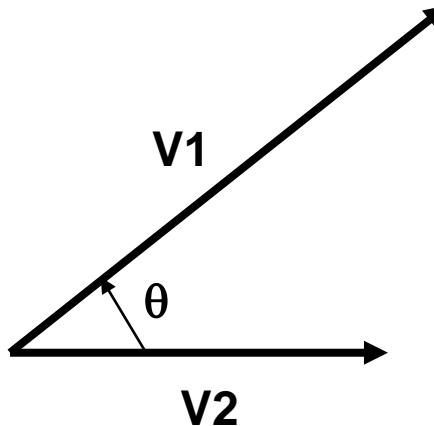
multiplying by 3, we have

$$3*\mathbf{V} = 3*(4\mathbf{i} + 3\mathbf{j}) = 12\mathbf{i} + 9\mathbf{j}$$

multiplying by 1/2, we have

$$(1/2)*\mathbf{V} = (1/2)*(4\mathbf{i} + 3\mathbf{j}) = 2\mathbf{i} + 1.5\mathbf{j}$$

# Dot product of two vectors



Given vectors  $\mathbf{V1}$  and  $\mathbf{V2}$ , their dot product is a scalar.

$$\mathbf{V1} \bullet \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \cos(\alpha) \quad \text{where } 0 \leq \alpha \leq 180^\circ$$

$$\cos(\alpha) = \mathbf{V1} \bullet \mathbf{V2} / (|\mathbf{V1}| |\mathbf{V2}|)$$

# Dot product of two vectors

- The product  $\mathbf{V1} \bullet \mathbf{V2}$  for  $\mathbf{V1} = x_1\mathbf{i} + y_1\mathbf{j}$  and  $\mathbf{V2} = x_2\mathbf{i} + y_2\mathbf{j}$  is

$$\begin{aligned}\mathbf{V1} \bullet \mathbf{V2} &= (x_1\mathbf{i}) \cdot (x_2\mathbf{i} + y_2\mathbf{j}) + (y_1\mathbf{j}) \cdot (x_2\mathbf{i} + y_2\mathbf{j}) \\ &= (x_1 * x_2) * \mathbf{i} * \mathbf{i} + (y_1 * y_2) * \mathbf{j} * \mathbf{j} + (x_1 * y_2) * \mathbf{i} * \mathbf{j} + (y_1 * x_2) * \mathbf{j} * \mathbf{i}\end{aligned}$$

- Because  $\mathbf{i} * \mathbf{i} = \mathbf{j} * \mathbf{j} = 1$  and  $\mathbf{i} * \mathbf{j} = \mathbf{j} * \mathbf{i} = 0$ , therefore

$$\mathbf{V1} \bullet \mathbf{V2} = x_1 * x_2 + y_1 * y_2$$

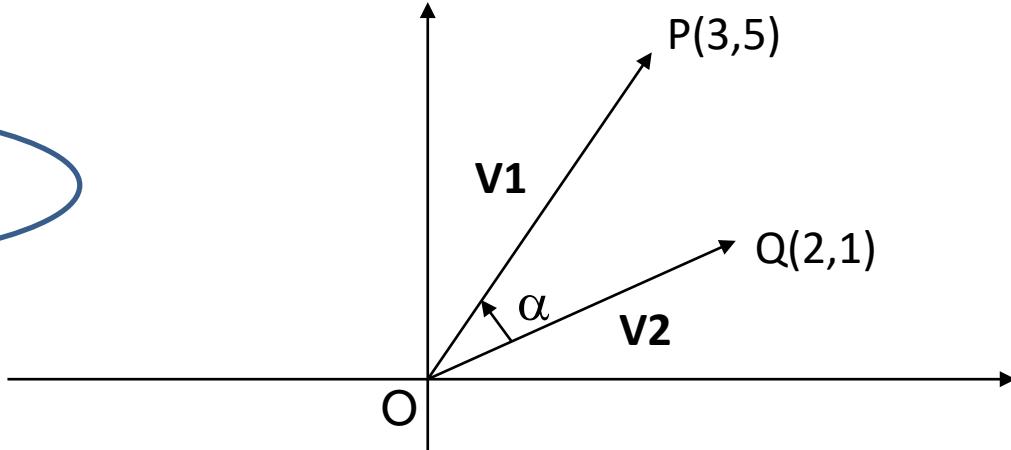
- The dot product is also expressed as

$$\mathbf{V1} \bullet \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \cos(\alpha)$$

$$\begin{aligned}\text{therefore } \cos(\alpha) &= \mathbf{V1} \bullet \mathbf{V2} / (|\mathbf{V1}| |\mathbf{V2}|) \\ &= (x_1 * x_2 + y_1 * y_2) / (|\mathbf{V1}| |\mathbf{V2}|)\end{aligned}$$

# Example use of dot product

Find angle  $\alpha$ ?



$$\mathbf{V1} = 3\mathbf{i} + 5\mathbf{j}$$

$$\mathbf{V2} = 2\mathbf{i} + \mathbf{j}$$

$$\mathbf{V1} \cdot \mathbf{V2} = 3 \cdot 2 + 5 \cdot 1 = 11$$

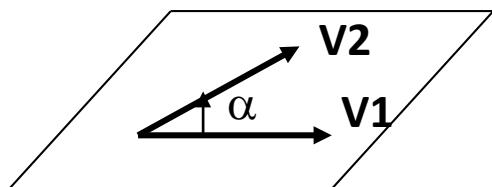
$$|\mathbf{V1}| = \sqrt{3^2 + 5^2} = \sqrt{34} = 5.831$$

$$|\mathbf{V2}| = \sqrt{2^2 + 1} = \sqrt{5} = 2.236$$

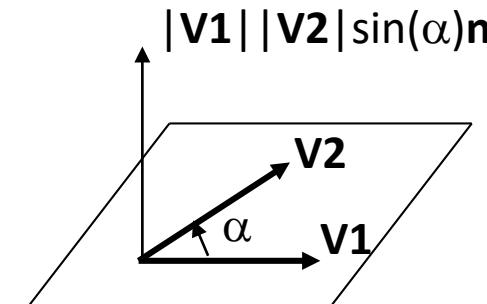
$$\cos(\alpha) = 11 / (5.831 \cdot 2.236) = 0.8437$$

$$\alpha = 32.47^\circ$$

# Cross product of two vectors



(i)



(ii)

For two vectors **V1** and **V2** lying on a plane (i), their cross product is another vector, which is perpendicular to the plane (ii).

The cross product is defined as

$$\mathbf{V1} \times \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \sin(\alpha) \mathbf{n}$$

where  $0 \leq \alpha < 180$  and **n** is a unit vector along the direction of the plane normal obeying the right-hand rule.

# Cross product of two vectors

- $\mathbf{V1} \times \mathbf{V2} = -\mathbf{V2} \times \mathbf{V1}$
- $\mathbf{V1} \times \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \sin(\alpha) \mathbf{n}$ , thus  $|\mathbf{V1} \times \mathbf{V2}| = |\mathbf{V1}| |\mathbf{V2}| \sin(\alpha)$
- When  $\alpha = 0$ ,  $\sin(\alpha) = 0$ . Hence  $\mathbf{i} \times \mathbf{i} = \mathbf{j} \times \mathbf{j} = \mathbf{k} \times \mathbf{k} = 0$   
where  $\mathbf{i}$ ,  $\mathbf{j}$  and  $\mathbf{k}$  are unit vectors along the x, y and z axes,  
respectively.
- When  $\alpha = 90$ ,  $\sin(\alpha) = 1$ . Hence  $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ ,  $\mathbf{j} \times \mathbf{k} = \mathbf{i}$  and  $\mathbf{k} \times \mathbf{i} = \mathbf{j}$
- From the identity in 1 above, the reverse 3 is true,  
i.e.  $\mathbf{j} \times \mathbf{i} = -\mathbf{k}$ ,  $\mathbf{k} \times \mathbf{j} = -\mathbf{i}$  and  $\mathbf{i} \times \mathbf{k} = -\mathbf{j}$

# Matrices

- Matrices are techniques for applying transformations.
- A matrix is simply a set of numbers arranged in a rectangular format.
- Each number is known as an element.
- Capital letters are used to represent matrices.
- Bold letters when printed (**M**), or underlined when written (M).
- A matrix has dimensions that refer to the number of rows and the number of columns it has.

# Dimensions of matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 6 & 3 \\ -1 & 2 & 4 \end{bmatrix}$$

Col 1 Col 2 Col 3  
Row 1 Row 2

The dimensions of A are  $(2 \times 3)$

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 1 & 0 \\ 1 & -4 & 6 \\ 0 & 2 & 1 \end{bmatrix}$$

Col 1 Col 2 Col 3  
Row 1 Row 2 Row 3 Row 4

The dimensions of B are  $(4 \times 3)$

$$\mathbf{C} = \begin{bmatrix} 1 & 6 \\ 2 & 9 \\ -3 & -2 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Col 1 Col 2  
Row 1 Row 2 Row 3 Row 4 Row 5

The dimensions of C are  $(5 \times 2)$

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0.5 \\ 0.3 & 0 & 0.2 \end{bmatrix}$$

Col 1 Col 2 Col 3  
Row 1 Row 2 Row 3

The dimensions of D are  $(3 \times 3)$

# Transpose matrix

- When a matrix is rewritten so that its rows and columns are interchanged, then the resulting matrix is called the transpose of the original.

$$\mathbf{A} = \begin{bmatrix} 1 & 6 & 3 \\ -1 & 2 & 4 \end{bmatrix}$$

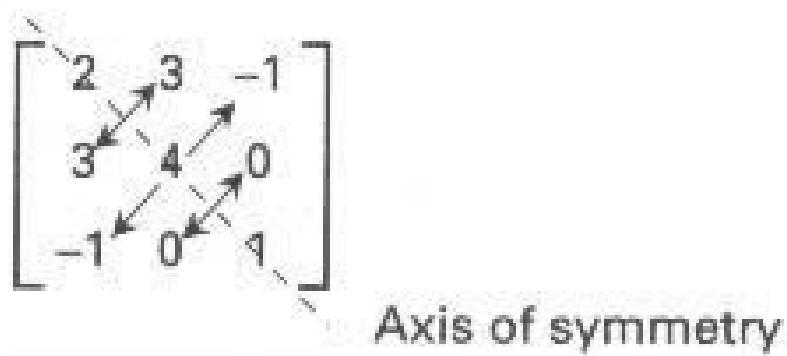
The dimensions of A are  $(2 \times 3)$

$$\mathbf{A}' = \begin{bmatrix} 1 & -1 \\ 6 & 2 \\ 3 & 4 \end{bmatrix}$$

The dimensions of A' are  $(3 \times 2)$

# Square and symmetric matrices

- A **square matrix** is a matrix where the number of rows equals the number of columns (e.g. Matrix D in slide 21).
- A **symmetric matrix** is a **square matrix** where the rows and columns are such that its transpose is the same as the original matrix, i.e. elements  $a_{ij} = a_{ji}$  where  $i \neq j$ .



# Identity matrices

- An **identity matrix**,  $I$  is a square/symmetric matrix with zeros everywhere except its diagonal elements which have a value of 1.
- Examples of 2x2, 3x3, and 4x4 matrices are

$$I_{(2 \times 2)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; I_{(3 \times 3)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; I_{(4 \times 4)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Adding matrices

- Matrices **A** and **B** may be added if they have the same dimensions.
- That is, the corresponding elements may be added to yield a resulting matrix.
- The sum is **commutative**, i.e.  $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$

$$\begin{bmatrix} 2 & 1 & 3 \\ 4 & 1 & -1 \end{bmatrix} \xrightarrow{+} \begin{bmatrix} 3 & 2 & 0 \\ 0 & 1 & 0 \end{bmatrix} \xrightarrow{=} \begin{bmatrix} 5 & 3 & 3 \\ 4 & 2 & -1 \end{bmatrix}$$

# Subtracting matrices

- Matrix **B** may be subtracted from matrix **A** if they have the same dimensions, i.e. the corresponding elements of **B** may be subtracted from those of **A** to yield a resulting matrix.

$$\begin{bmatrix} 2 & 1 \\ 6 & 5 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 3 \\ 2 & 2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ 4 & 3 \\ -1 & 1 \end{bmatrix}$$

- The result is **not commutative**. Reversing the order of the matrices yields different results, i.e.  $\mathbf{A} - \mathbf{B} \neq \mathbf{B} - \mathbf{A}$

$$\begin{bmatrix} 6 & 2 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 3 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & -1 \\ 2 & -1 \end{bmatrix}$$

Reversing the operation  $\begin{bmatrix} 1 & 3 \\ -1 & 1 \end{bmatrix} - \begin{bmatrix} 6 & 2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -5 & 1 \\ -2 & 1 \end{bmatrix}$  Different result

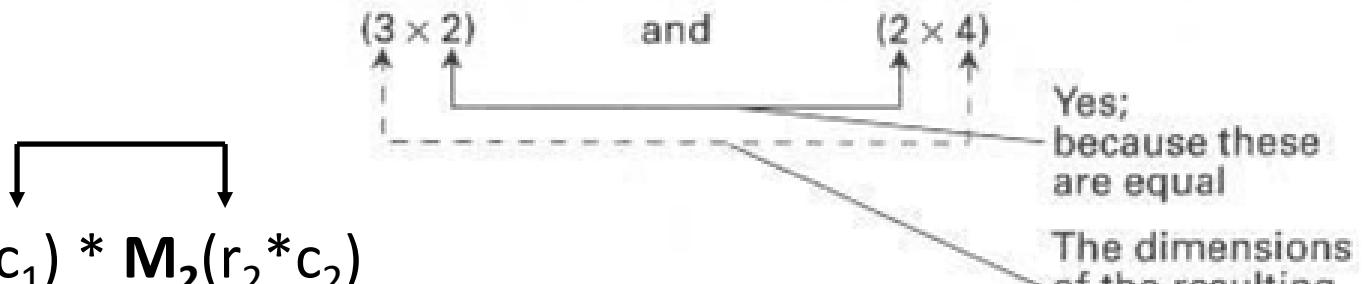
# Multiplying matrices

- By a constant

$$3 \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 6 & 3 \end{bmatrix}$$

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

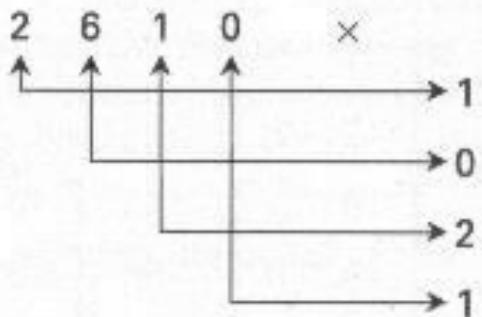
- By a matrix - The rule for multiplying one matrix to another is simple: if the **number of columns** in the first matrix is the **same as the number of rows** in the second matrix, the multiplication can be done.



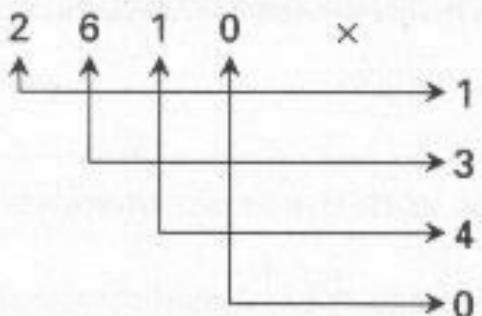
So  $M_1(r_1 * c_1) * M_2(r_2 * c_2)$   
=  $M_3(r_1 * c_2)$  where  $c_1 = r_2$

# Multiplying matrices

$$\begin{array}{c} \text{Dimension } 2 \times 4 \\ \left[ \begin{array}{cccc} 2 & 6 & 1 & 0 \\ 3 & 2 & 4 & 2 \end{array} \right] \end{array} \times \begin{array}{c} \text{Dimension } 4 \times 3 \\ \left[ \begin{array}{ccc} 1 & 1 & 2 \\ 0 & 3 & 5 \\ 2 & 4 & 1 \\ 1 & 0 & 3 \end{array} \right] \end{array} = \begin{array}{c} \text{Dimension } 2 \times 3 \\ \left[ \begin{array}{ccc} \triangle * & \text{hexagon}* & * \\ * & * & \text{circle}* \end{array} \right] \end{array}$$

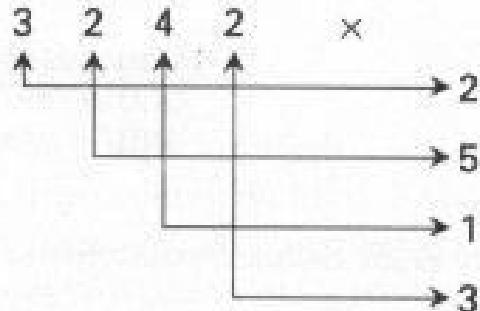


$$\begin{aligned} \text{gives } & (2 \times 1) + (6 \times 0) \\ & + (1 \times 2) + (0 \times 1) \\ & = 2 + 0 + 2 + 0 \\ & = \triangle 4 \end{aligned}$$



$$\begin{aligned} \text{gives } & (2 \times 1) + (6 \times 3) \\ & + (1 \times 4) + (0 \times 0) \\ & = 2 + 18 + 4 + 0 \\ & = \text{hexagon } 24 \end{aligned}$$

# Multiplying matrices – example



$$\begin{aligned} &\text{gives } (3 \times 2) + (2 \times 5) \\ &+ (4 \times 1) + (2 \times 3) \\ &= 6 + 10 + 4 + 6 \\ &= 26 \end{aligned}$$

$$\begin{bmatrix} 2 & 6 & 1 & 0 \\ 3 & 2 & 4 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 5 \\ 2 & 4 & 1 \\ 1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 4 & 24 & 35 \\ 13 & 25 & 26 \end{bmatrix}$$

The overall result

# Non-commutative property of matrix multiplication

Matrix multiplication is not **commutative**.

$$\begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \\ 2 & 6 \end{bmatrix} = \begin{bmatrix} 4 & 14 \\ 7 & 24 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 \\ 2 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 14 & 9 \\ 22 & 14 \end{bmatrix}$$

Reversing the order of the matrices yields different results.

# Non-commutative property of matrix multiplication

Reversing the order of the matrices yields different results (e.g. Slide 30) or the condition for matrix multiplication will not be satisfied (e.g. Slide 28).

Further example:

$$\mathbf{A} = [1 \ 2 \ 3], \quad \mathbf{B} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Will the following multiplications be possible?

$$\mathbf{A}^* \mathbf{B}$$

$$\mathbf{B}^* \mathbf{A}$$

# Inverse matrices

If two matrices **A** and **B**, when multiplied together, results in an identity matrix **I**, then matrix **A** is the inverse of matrix **B** and vice versa, i.e.

$$\mathbf{A} \times \mathbf{B} = \mathbf{B} \times \mathbf{A} = \mathbf{I}$$

$$\mathbf{A} = \mathbf{B}^{-1} \text{ and } \mathbf{B} = \mathbf{A}^{-1}$$

e.g.

$$\begin{bmatrix} 7 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -3 \\ -2 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Therefore

$$\mathbf{A}^{-1} = \begin{bmatrix} 1 & -3 \\ -2 & 7 \end{bmatrix}$$

# Topics covered today

- Computer representation of objects
- Cartesian co-ordinate system
- Points, lines and angles
- Trigonometry
- Vectors (unit vector) and vector calculations (addition, subtraction, scaling, dot product and cross product)
- Matrices (dimension, transpose, square/symmetric/identity and inverse) and matrix calculations (addition, subtraction and multiplication)



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Geometric Primitives**

**Week 03**  
**2021-22**

**Yong Yue**

# Topics for today

## ➤ **Graphics Primitives**

- Points
- Lines
- Polygons

## ➤ **Line Algorithms**

- Digital Differential Analyser (DDA)
- Bresenham Algorithm
- Circles
- Antialiasing

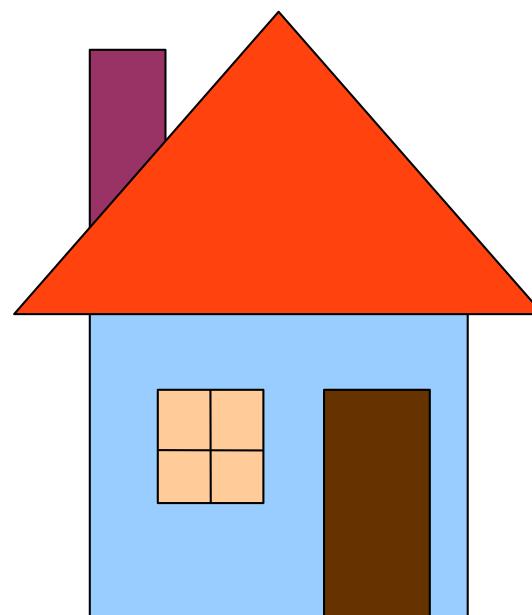
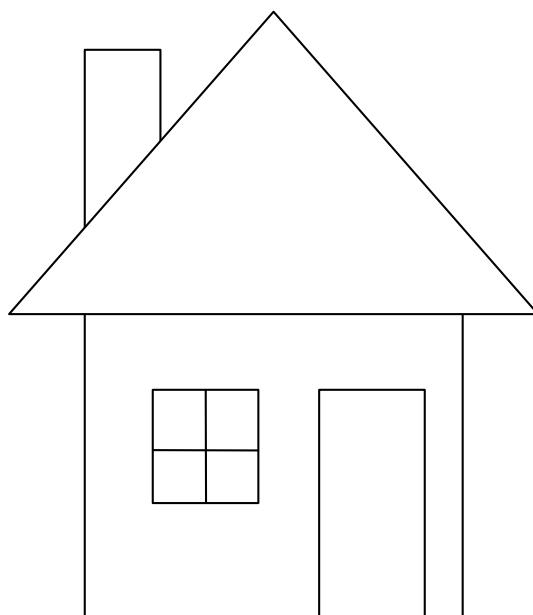
## ➤ **Polygon Fill**

## ➤ **Graphics Primitives with OpenGL**

- `glBegin(GL_POINTS); glEnd(GL_LINES)`
- `glBegin(GL_POLYGON); glEnd(GL_QUAD)`
- ...

# Question

How would you draw / model this house?



# Quick answer

## ➤ Applications Packages Tools

- Powerpoint
- Word
- Paint
- 3DS Max
- Maya

## ➤ API Library

- OpenGL
- DirectX
- Java2D

## ➤ Algorithms Techniques

- DDA
- Midpoint
- Bresenham
- Floodfill
- Antialiasing

# Quick answer

## ➤ Applications Packages Tools

- Powerpoint
- Word
- Paint
- 3DS Max
- Maya

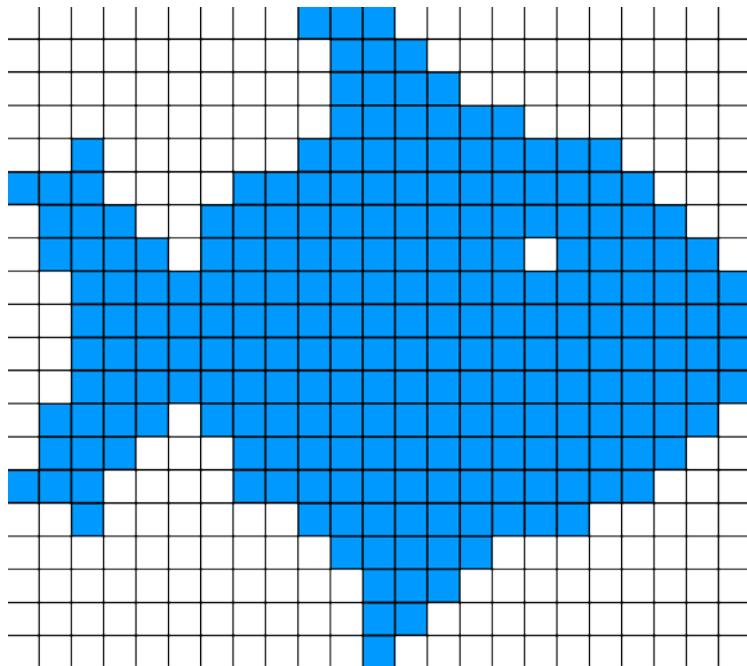
## ➤ API Library



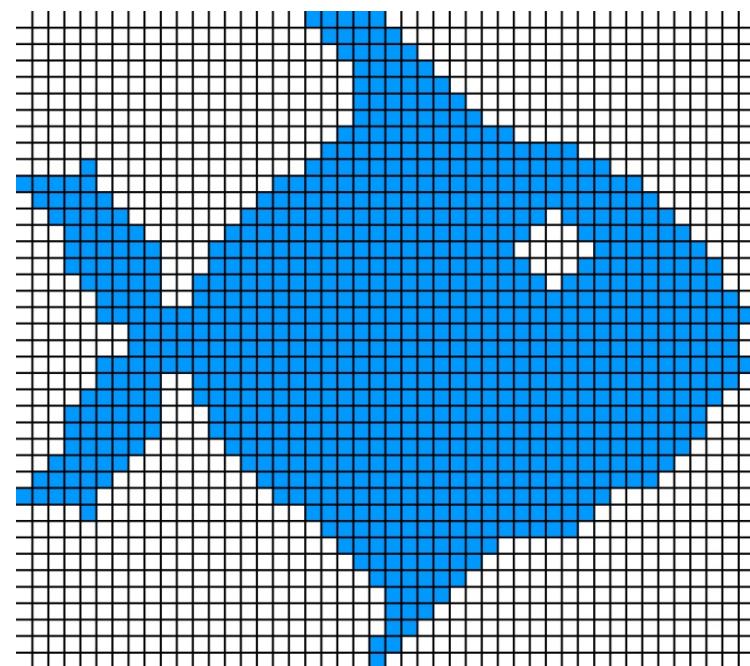
## ➤ Algorithms Techniques

- DDA
- Midpoint
- Bresenham
- Floodfill
- Antialiasing

# Image with 2D primitives

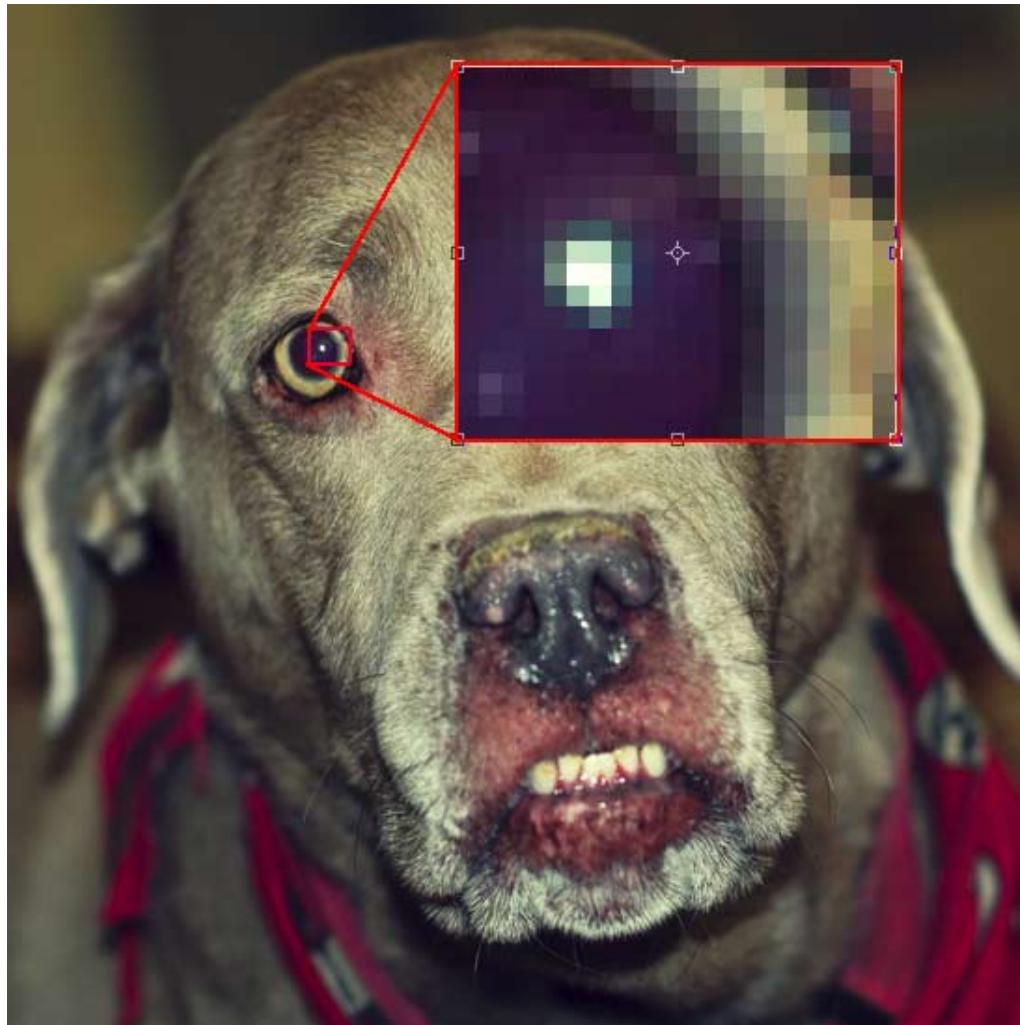


Low resolution



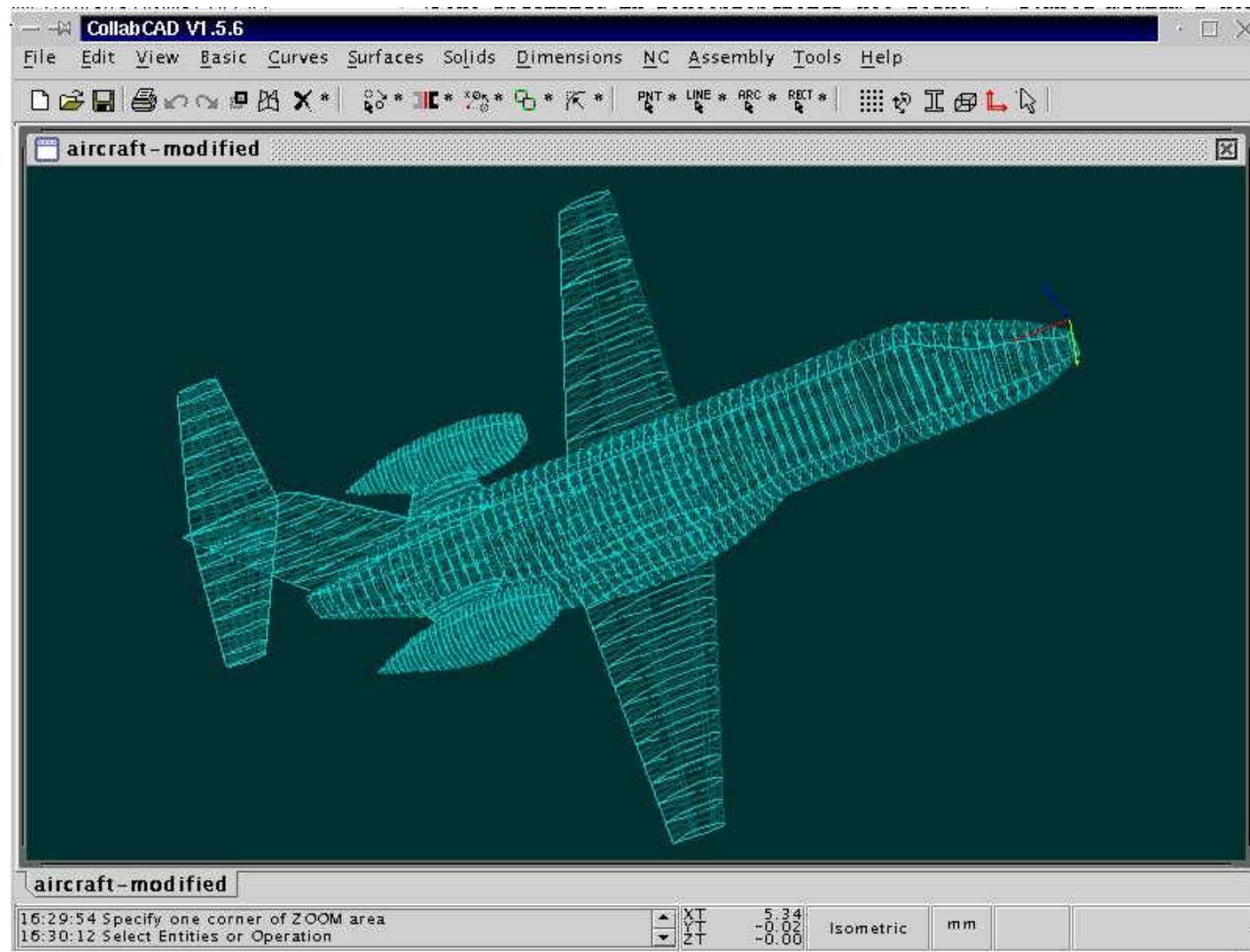
High resolution

# Example of points – photograph

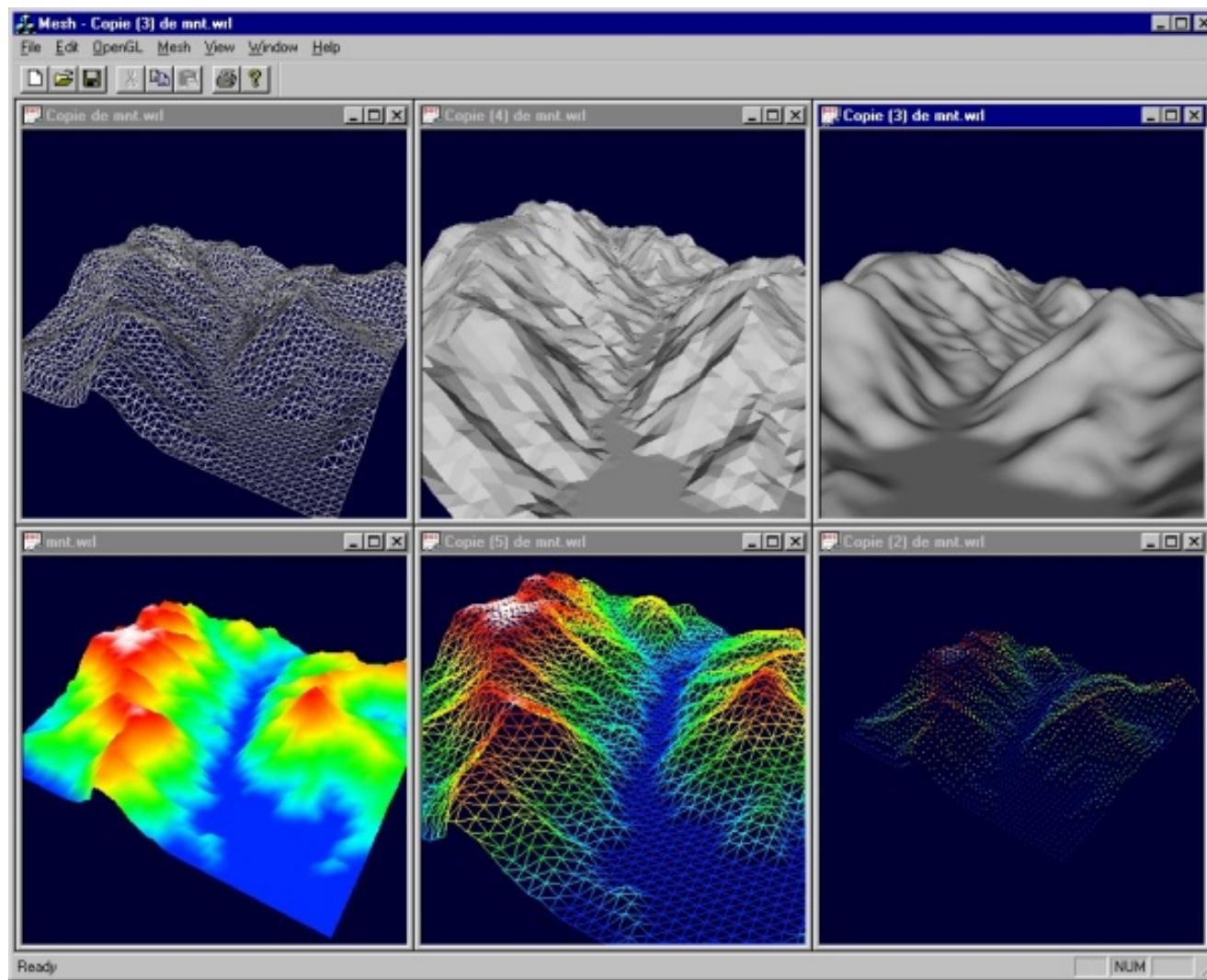


High-resolution points can be seen when we zoom in.

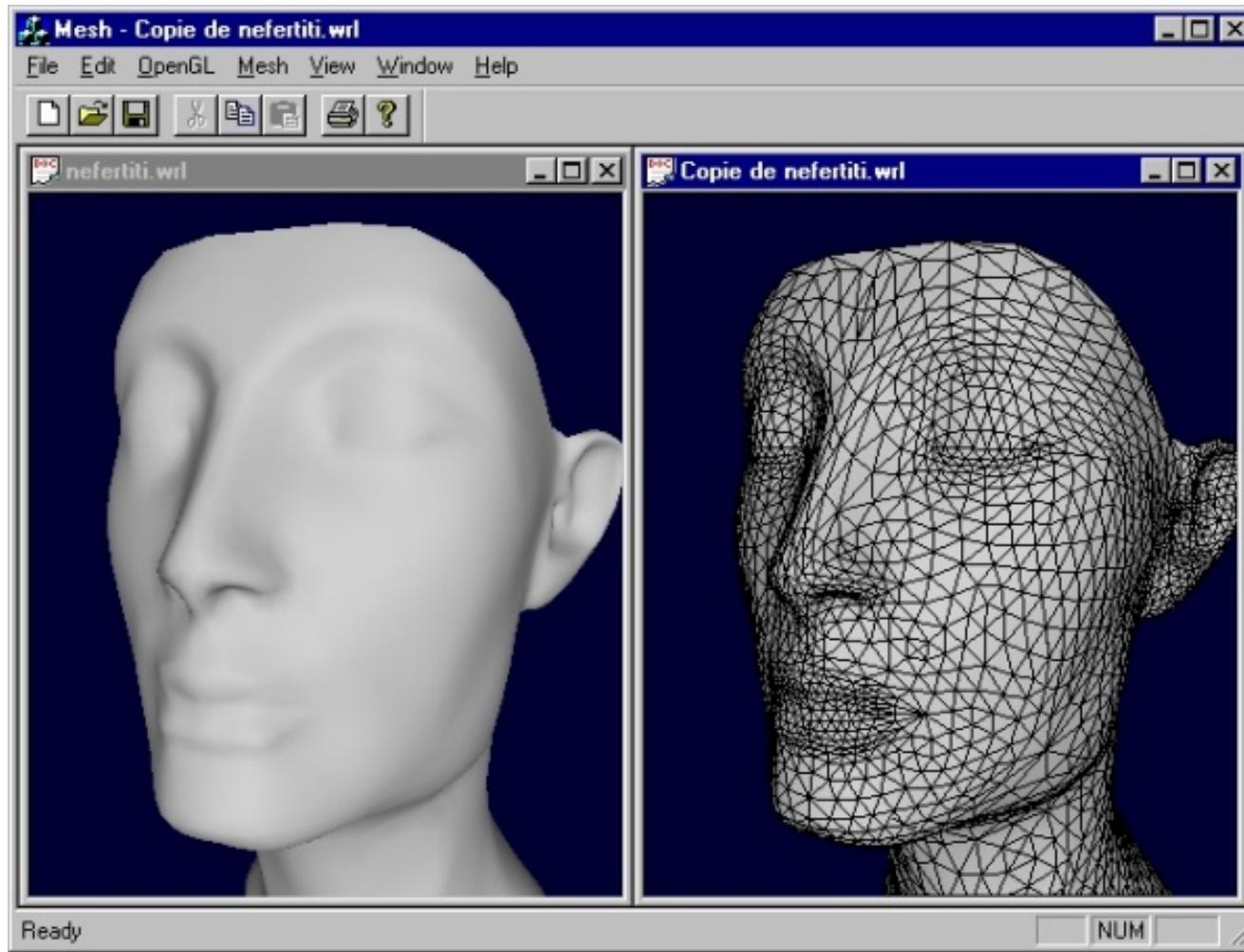
# Example of lines – wireframe



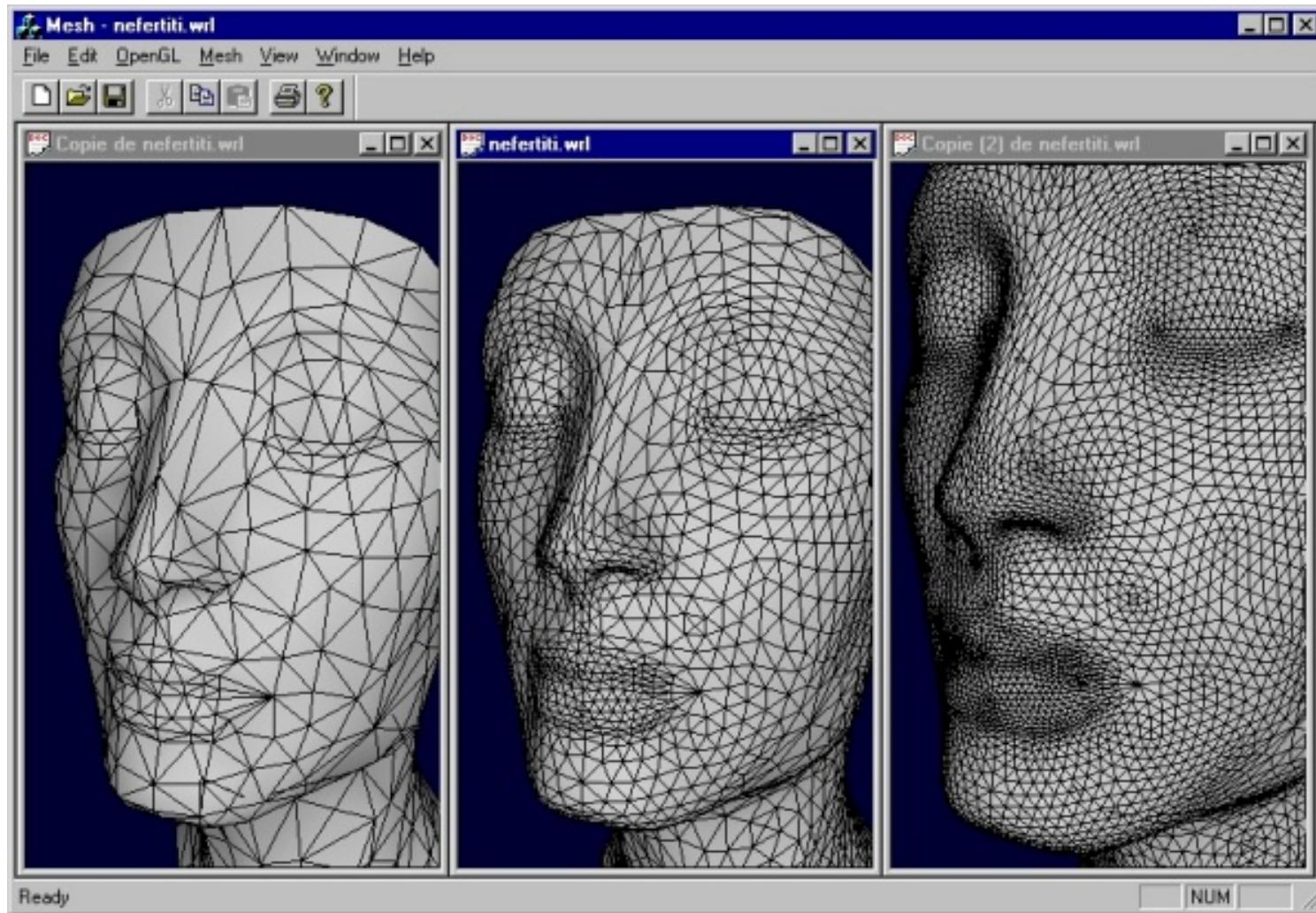
# Example of lines – sculptured mesh



# Example of lines – face mesh



# Mesh: level of detail (low, medium and high)



# Line characterisations

- Implicit

$$y = mx + b$$

or

$$F(x, y) = ax + by + c = 0$$

# Line characterisations

- Parametric     $P(t) = (1-t)P_0 + tP_1$   
(explicit)  
where               $P(0) = P_0 ; \quad P(1) = P_1$
  
- Intersection of 2 planes
  
- Shortest path between 2 points
  
- *Convex hull* of 2 discrete points

# “Good” discrete lines

- No gaps in adjacent pixels
- Pixels close to ideal line
- Consistent choices; same pixels in same situations
- Smooth looking
- Even brightness in all orientations
- Same line for  $P_0 P_1$  as for  $P_1 P_0$
- Double pixels stacked up?

# Line algorithms

- Drawing a horizontal line from  $(x_1, y)$  to  $(x_2, y)$  are easy ... just increment along x

while loop

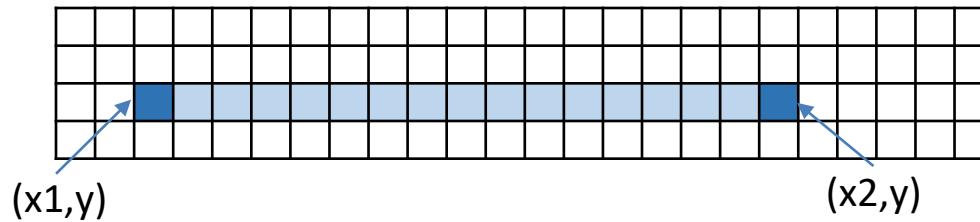
```
x = x1 while x <= x2
do {
    DrawPoint(x,y)
    x = x + 1
}
```

"for"-loop

```
for x = x1 to x2
Do {
    DrawPoint(x,y)
}
```

generator

```
DrawLine(x1 to x2, y)
```



- How do we draw lines that are not aligned to the X or Y axis?

# DDA – Digital Differential Algorithm

$$y = mx + b$$

$$m = (y_2 - y_1) / (x_2 - x_1) = \Delta y / \Delta x$$

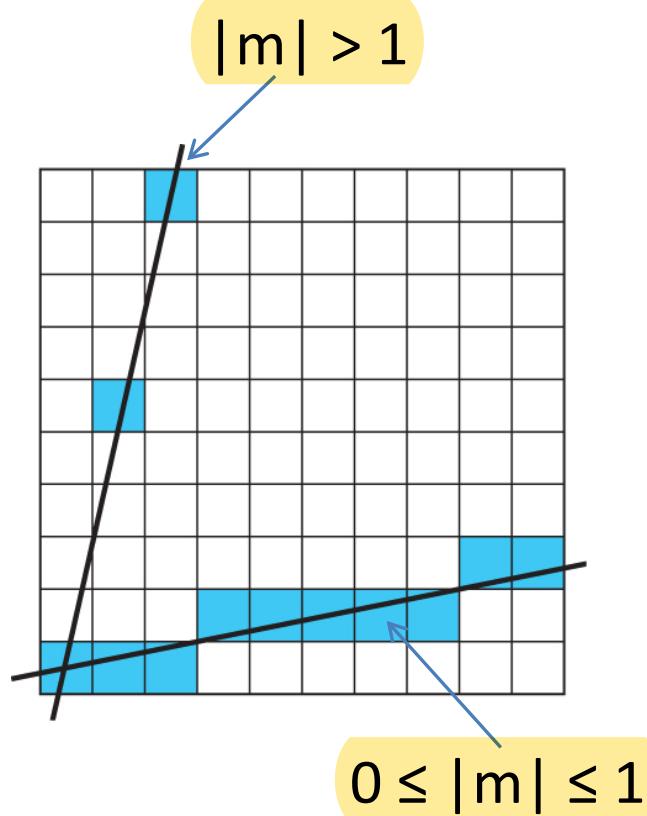
$$\Delta y = m \Delta x$$

As we move along  $x$  by incrementing  $x$ ,  $\Delta x = 1$ , so  $\Delta y = m$

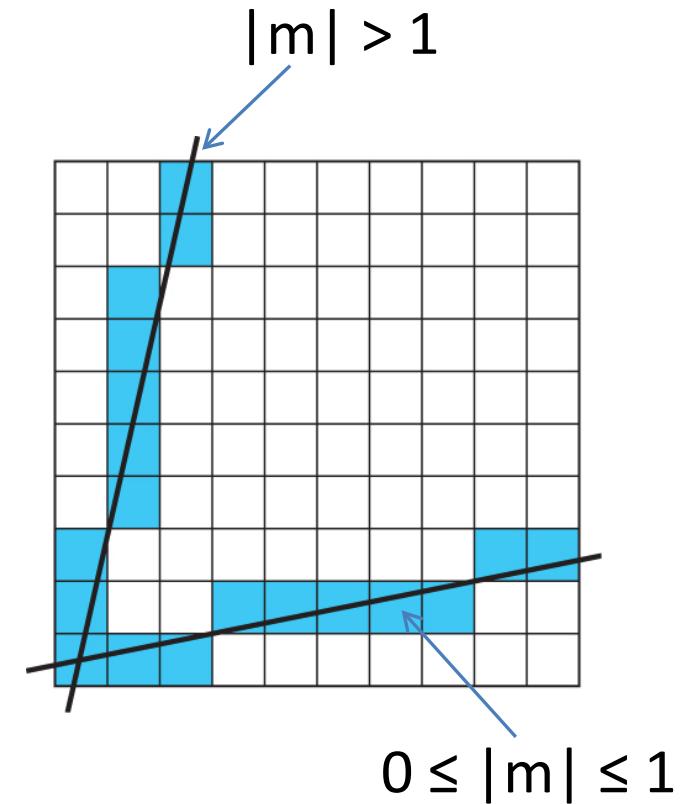
When  $0 \leq |m| \leq 1$

```
int x;  
float y=y1;  
for(x=x1; x<=x2; x++){  
    write_pixel(x, round(y), line_color);  
    y+=m;  
}
```

# DDA – Digital Differential Algorithm



Sampling along x-axis for both lines



Sampling along x-axis ( $0 \leq |m| \leq 1$ )  
Sampling along y-axis ( $|m| > 1$ )

# DDA – Digital Differential Algorithm

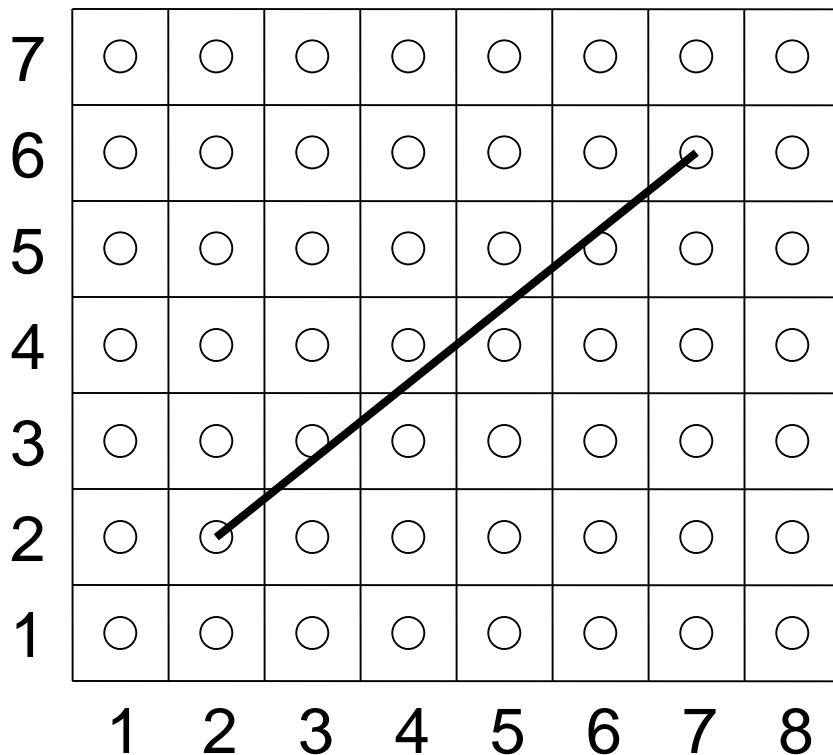
When  $|m| > 1$ , we swap the roles of  $x$  and  $y$   
( $\Delta y = m\Delta x$  so  $\Delta x = \Delta y/m = 1/m$ ).

```
int y;  
float x=x1;  
for(y=y1; y<=y2; y++){  
    write_pixel(y, round(x), line_color);  
    x+=1/m;  
}
```

Questions:

- 1) Can you combine the two in one pseudo-code?
- 2) Can you derive the parts of the algorithm for negative slopes?

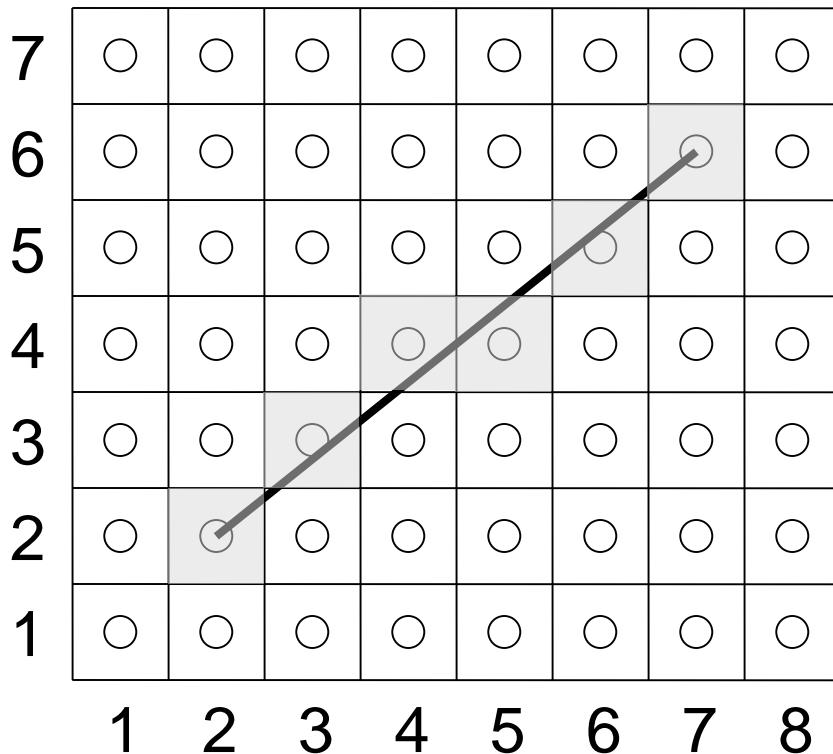
# Example: line from (2,2) to (7,6)



$$m = \Delta y / \Delta x = 0.8$$

xi	yf	yi

# Example: line from (2,2) to (7,6)



$$m = \Delta y / \Delta x = 0.8$$

$$y = x + m$$

xi	yf	yi
2	2.0	2
3	2.8	3
4	3.6	4
5	4.4	4
6	5.2	5
7	6.0	6

# The Bresenham line algorithm

- The Bresenham algorithm is another incremental scan conversion algorithm.
- It is accurate and efficient.
- Its big advantage is that it uses only integer calculations (unlike DDA which requires float-point additions).
- The calculation of each successive pixel requires only an addition and a sign test. It is so efficient that it has been incorporated as a single instruction on graphics chips.



Jack Elton Bresenham worked for 27 years at IBM before entering academia. Bresenham developed his famous algorithms at IBM in the early 1960s.

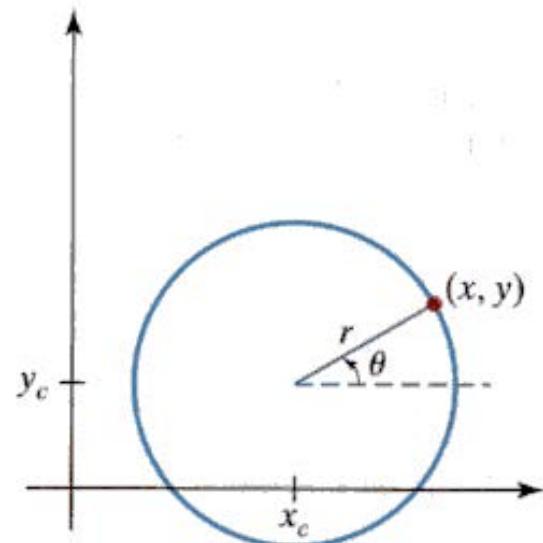
# Generation of circles

In Cartesian co-ordinates

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

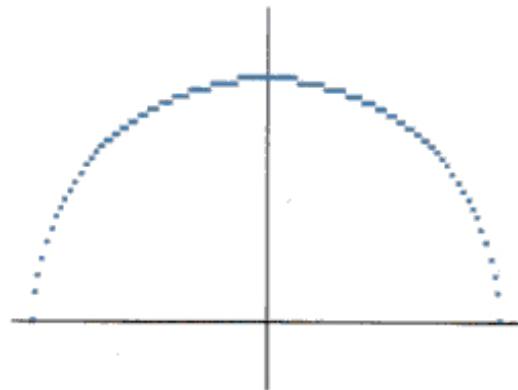
The position of points on the circle circumference can be calculated by stepping along the  $x$  axis in unit steps from  $x_c - r$  to  $x_c + r$  and calculating the corresponding  $y$  value at each position as

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$



# Generation of circles - problems

- Considerable amount of computation;
- The spacing between plotted pixel positions is not uniform;
  - This could be adjusted by interchanging  $x$  and  $y$  (stepping through  $y$  values and calculating the  $x$  values) whenever the absolute value of the slope of the circle is greater than 1.
  - However this simply increases the computation and processing required by the algorithm.



# Generation of circles – polar co-ordinates

In polar co-ordinates

$$\begin{cases} x = x_c + r \cos \theta \\ y = y_c + r \sin \theta \end{cases}$$

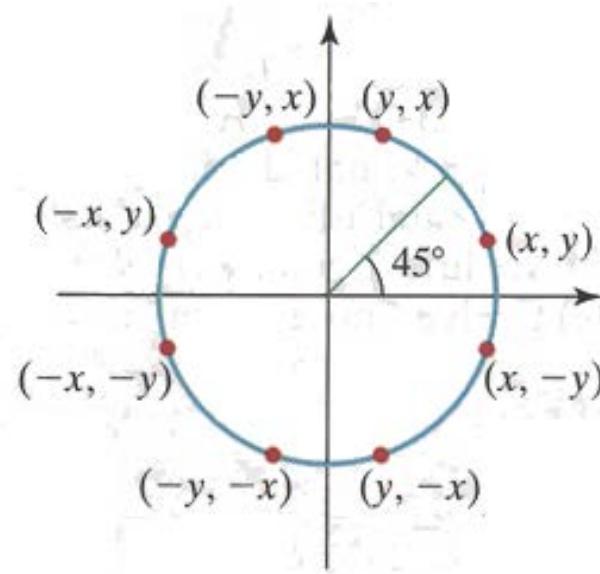
- When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference.
- To reduce calculations, a large angular separation can be used between points along the circumference and connect the points with straight-line segments to approximate the circle path.
- For a more continuous boundary on a raster display, the angular step size can be set at  $1/r$ . This plots pixel positions that are approximately one unit apart.

# Generation of circles - symmetry

- Computations can be reduced by considering the symmetry of the circle.
- If the curve positions in the first quadrant are determined, the circle section in the second quadrant can be generated by noting that the two circle sections are symmetric with respect to the  $y$  axis.
- Circle sections in the third and fourth quadrants can be obtained from the sections in the first and second quadrants by considering the symmetry about the  $x$  axis.
- Taking this one step further, it can be noted that there is also symmetry between octants. Circle sections in adjacent octants within one quadrant are symmetric with respect to the  $45^\circ$  line dividing the two octants.

# Generation of circles - symmetry

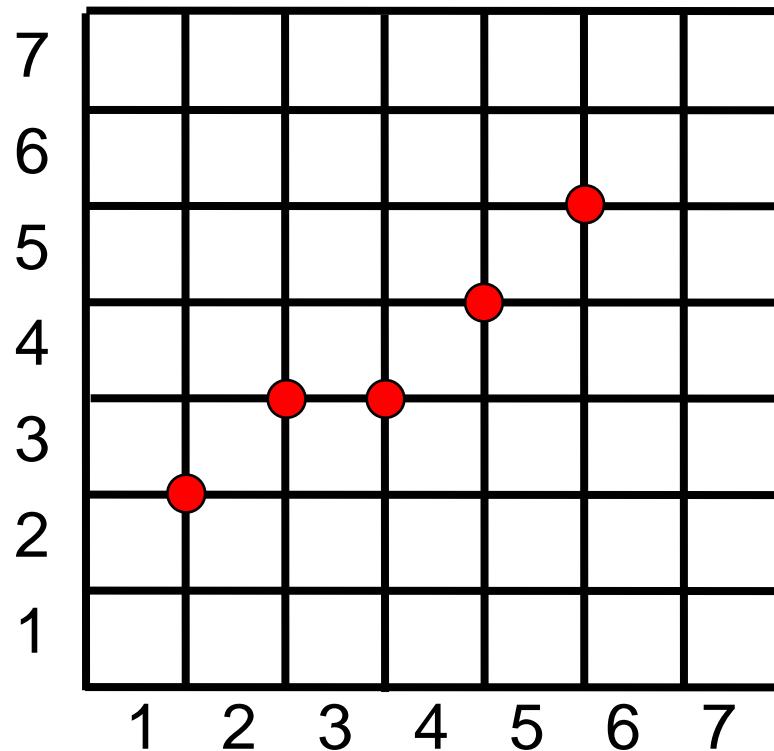
- Considering symmetry conditions between octants, a point at position  $(x, y)$  on a one-eighth circle sector is mapped into the seven circle points in the other octants of the  $x$ - $y$  plane.
- Taking the advantage of the circle symmetry in this way, all pixels around a circle can be generated by calculating only the points within the sector from  $x = r$  to  $x = y$ .
- The slope of the curve in this octant has an absolute magnitude equal to or larger than 1.



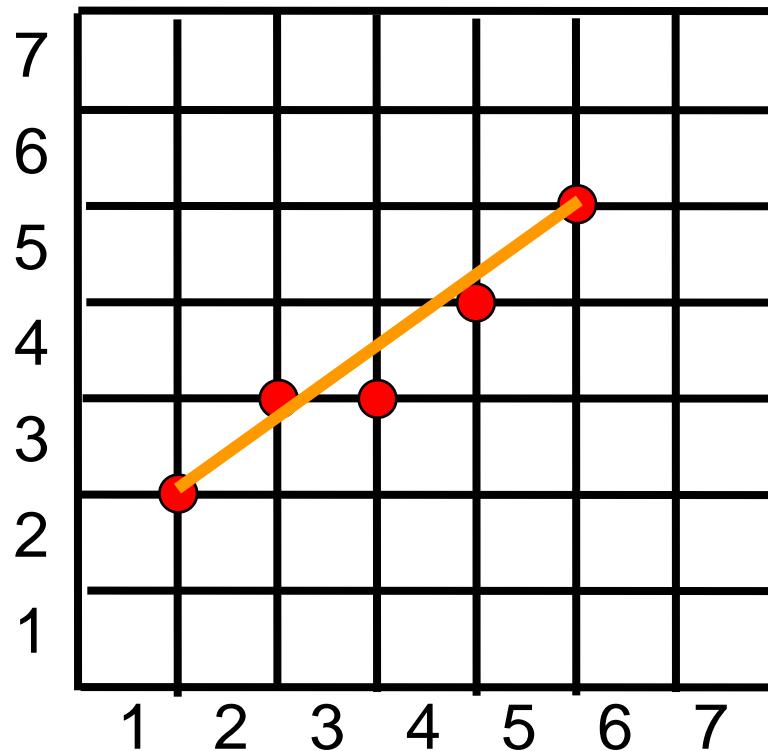
# Generation of circles - Efficiency

- Determining pixel positions along a circle circumference using symmetry and the equation either in Cartesian or polar co-ordinates, still requires a good deal of computation.
- The Cartesian equation involves multiplication and square root calculations.
- The parametric equations contain multiplications and trigonometric calculations.
- More efficient circle algorithms are based on incremental calculation of decision parameters, which involves only simple integer operations.

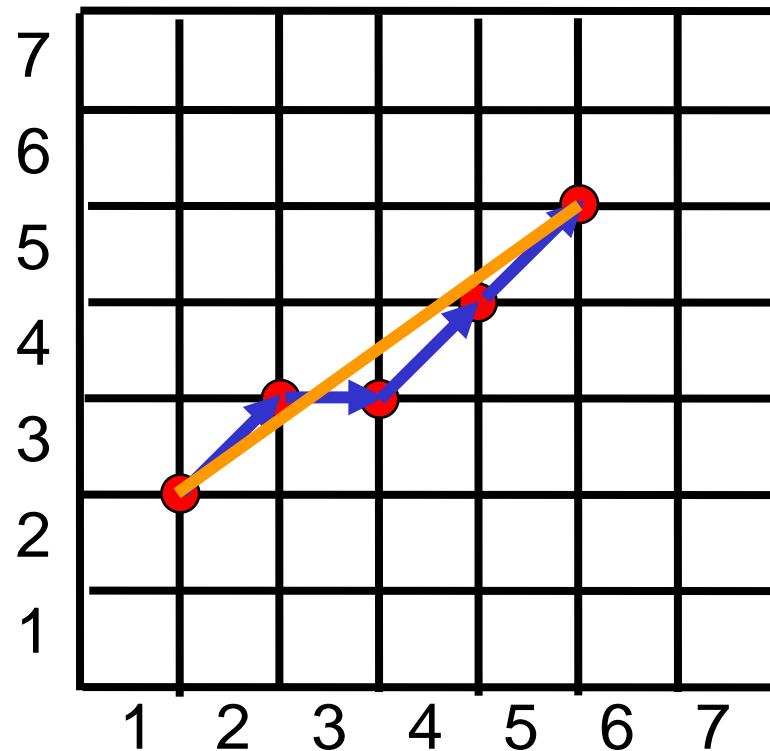
# Line: raster points



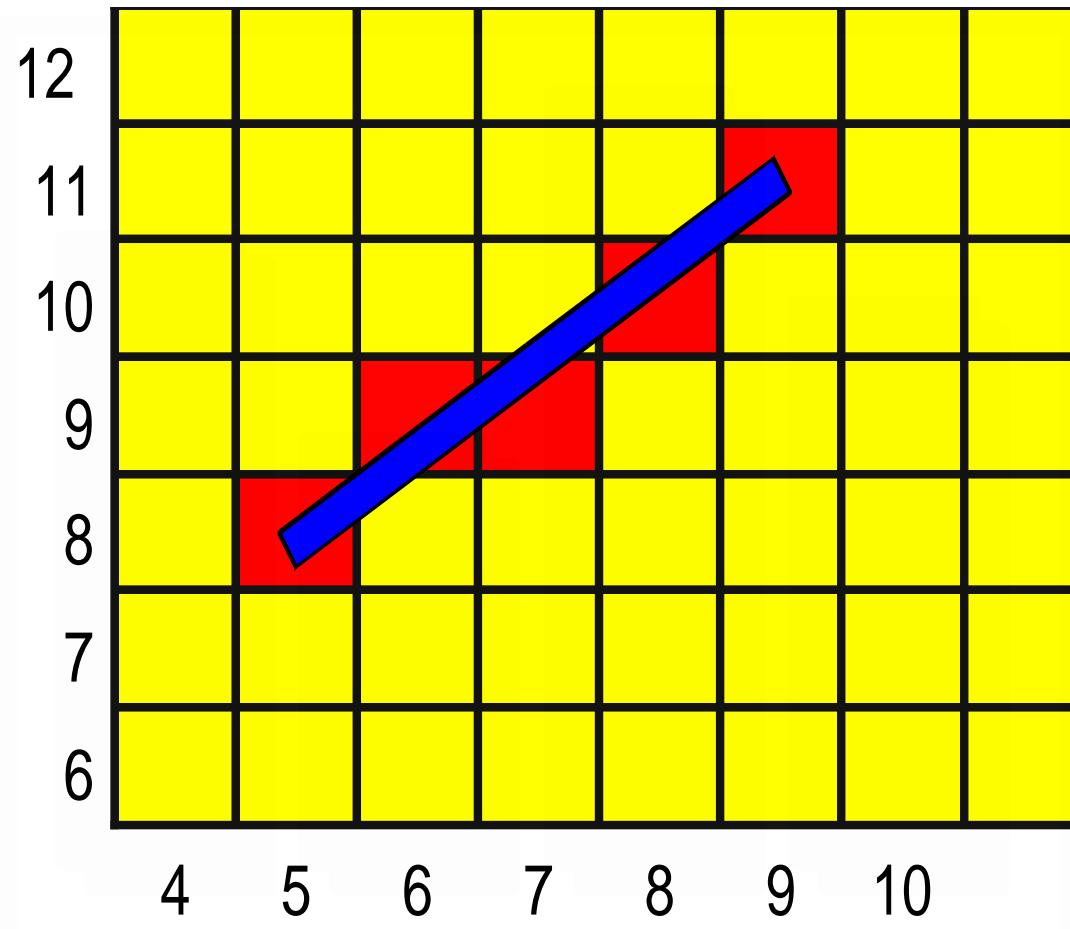
# Lines vs points



# Jaggies



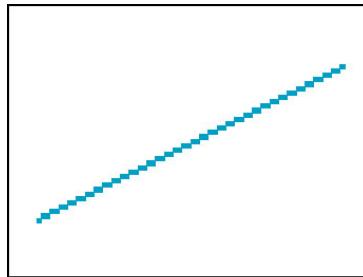
# Pixel space



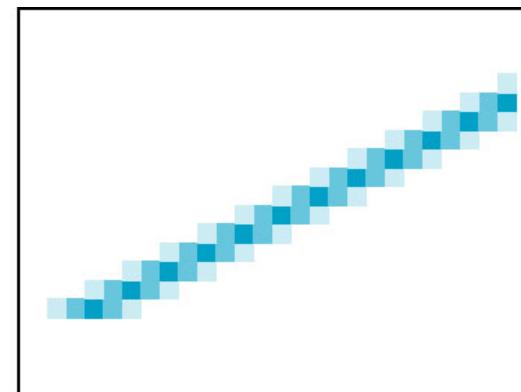
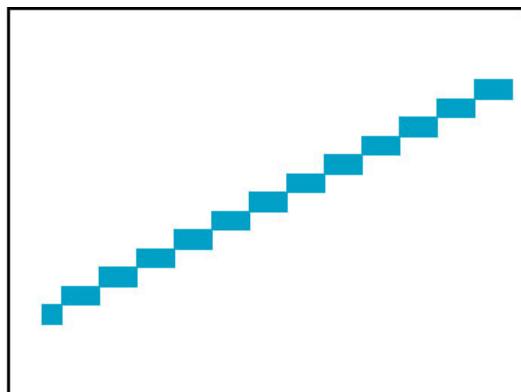
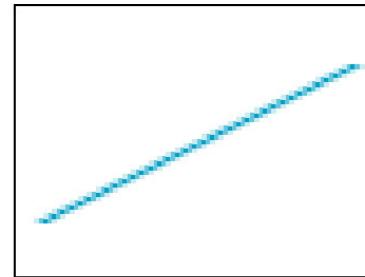
# Antialiasing by area averaging

Colour multiple pixels for each x depending on coverage by ideal line

Original



Antialiased



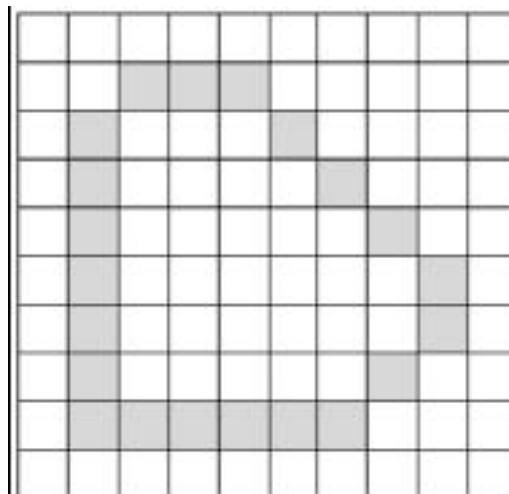
Magnified

# Geometric primitives – polygons and triangles

- The basic graphics primitives are points, lines and polygons
  - A polygon can be defined by an ordered set of vertices
- Graphics hardware is optimised for processing points and flat polygons
- Complex objects are eventually divided into triangular polygons (a process called tessellation)
  - Because triangular polygons are always flat

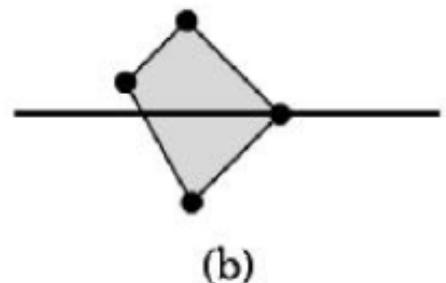
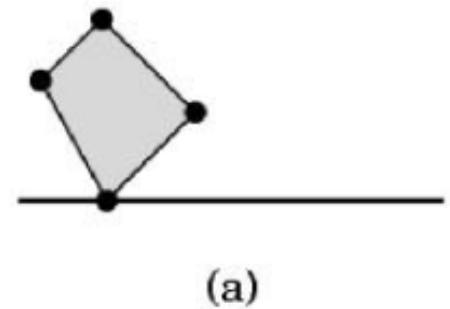
# Scan conversion

- Also called rasterization.
- The 3D to 2D projection gives us 2D vertices (points).
- We need to fill in the interior.



# Polygon fill

- Rasterize edges into framebuffer.
- Find a seed pixel inside the polygon.
- Visit neighbours recursively and colour if they are not edge pixels.
- When vertices lie on the scanlines, cases (a) and (b) must be treated differently when using odd-even fill definition
  - Case (a): zero or two crossings
  - Case (b): one edge crossing

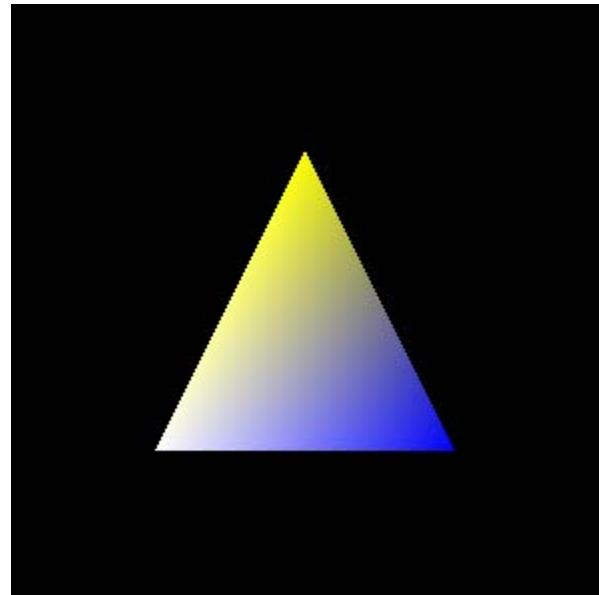
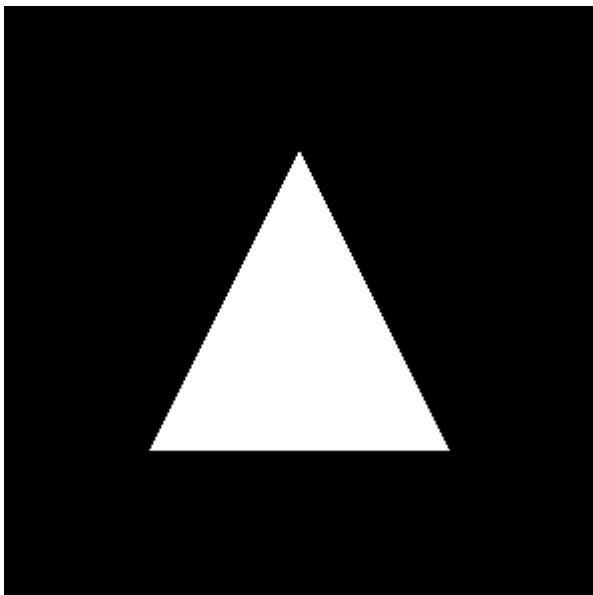


# Polygon fill

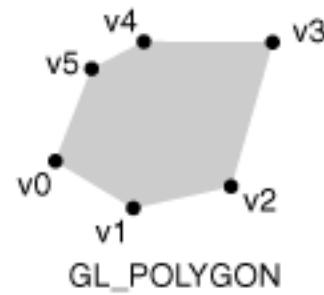
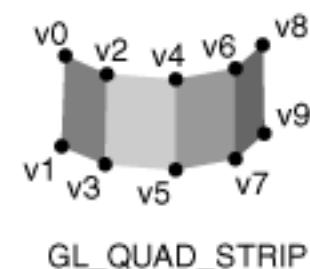
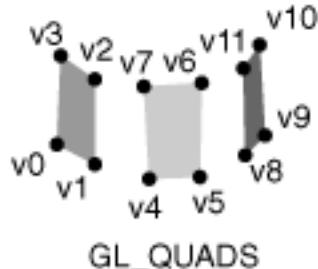
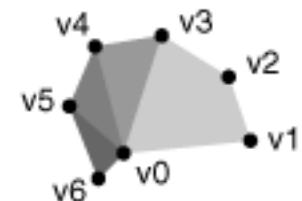
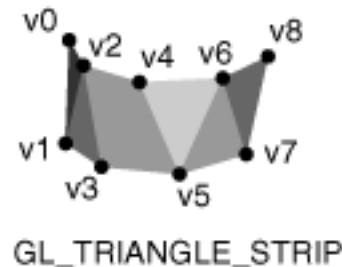
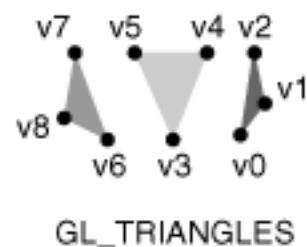
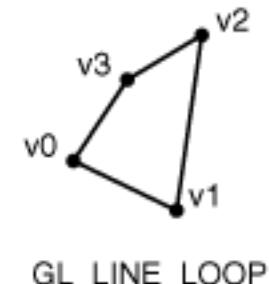
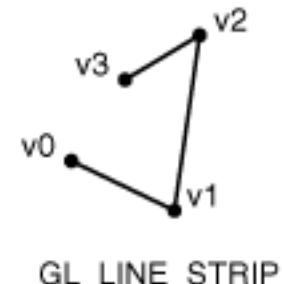
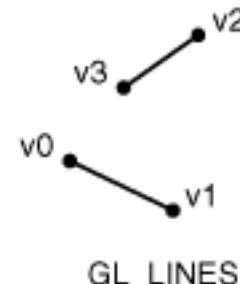
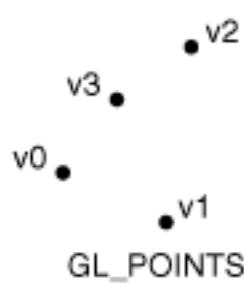
Flat shading

vs

Smooth shading



# Geometric primitives in OpenGL



# glBegin(prametres)

- GL\_POINTS: individual points
- GL\_LINES: pairs of vertices interpreted as individual line segments
- GL\_LINE\_STRIP: series of connected line segments
- GL\_LINE\_LOOP: same as above, with a segment added between last and first vertices
- GL\_TRIANGLES: triples of vertices interpreted as triangles
- GL\_TRIANGLE\_STRIP: linked strip of triangles
- GL\_TRIANGLE\_FAN: linked fan of triangles
- GL\_QUADS: quadruples of vertices interpreted as four-sided polygons
- GL\_QUAD\_STRIP: linked strip of quadrilaterals
- GL\_POLYGON: boundary of a simple, convex polygon

## **GL\_POINTS**

```
// this code will draw a point located at (100, 100)
```

```
glBegin(GL_POINTS);
```

```
glVertex2f(100.0f, 100.0f);
```

```
...
```

```
// add more points if required
```

```
glEnd();
```

# **GL\_LINES**

```
// this code will draw a line at starting and ending  
// coordinates specified by glVertex2f
```

```
glBegin(GL_LINES);  
    glVertex2f(100.0f, 100.0f); // origin of line  
    glVertex2f(200.0f, 140.0f); // end point of line  
glEnd();
```

# How to make lines efficient?

```
// this code will draw two lines "at a time" to save  
// the time it takes to call glBegin and glEnd
```

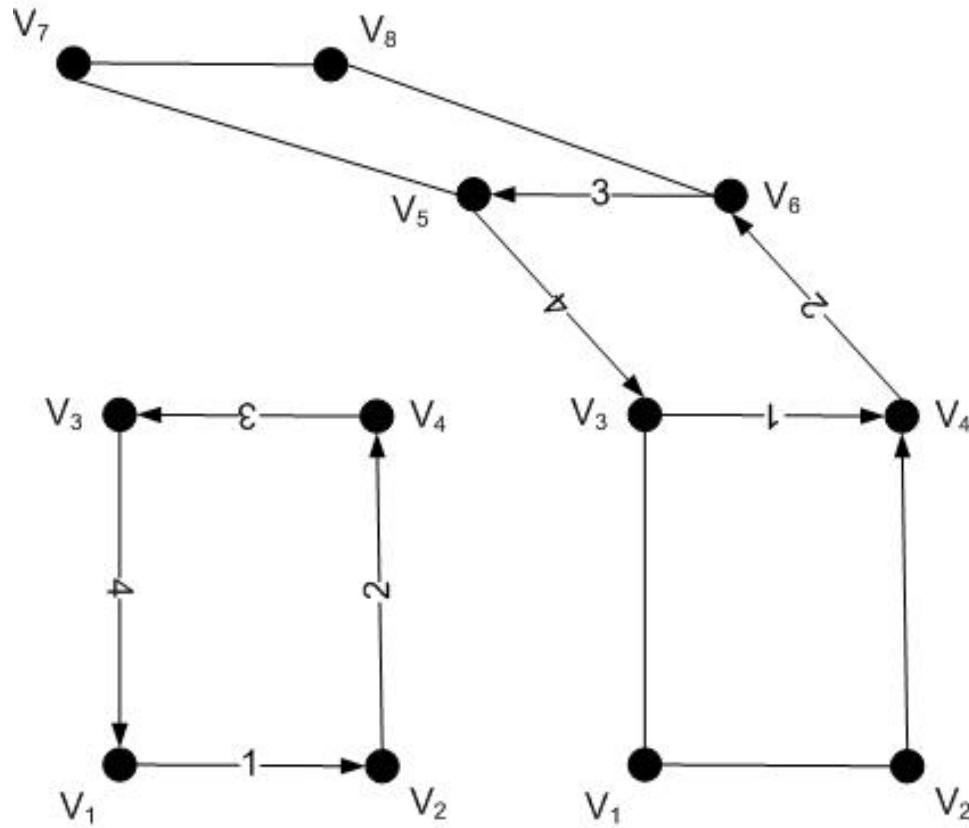
```
glBegin(GL_LINES);  
  
    glVertex2f(100.0f, 100.0f); // origin of the FIRST line  
    glVertex2f(200.0f, 140.0f); // end point of the FIRST line  
  
    glVertex2f(120.0f, 170.0f); // origin of the SECOND line  
    glVertex2f(240.0f, 120.0f); // end point of the SECOND line  
  
glEnd( );
```

# Triangle in OpenGL

```
glBegin(GL_TRIANGLES);  
    glVertex2f(-0.5,-0.5);  
    glVertex2f(0.5,0.0);  
    glVertex2f(0.0,0.5);  
glEnd();
```

# glQuad\_STRIP

Ordering of coordinates very important



$$|V| = 4$$

$$|V| = 8$$

# Summary

## ➤ **Graphics Primitives**

- Points
- Lines
- Polygons

## ➤ **Line Algorithms**

- Digital Differential Analyser (DDA)
- Bresenham Algorithm
- Circles
- Antialiasing

## ➤ **Polygon Fill**

## ➤ **Graphics Primitives with OpenGL**

- `glBegin(GL_POINTS); glEnd(GL_LINES)`
- `glBegin(GL_POLYGON); glEnd(GL_QUAD)`
- ...



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

**CPT205 Computer Graphics**

**Transformation Pipeline and  
Geometric Transformations**

**Week 04**  
**2021-22**

**Yong Yue**

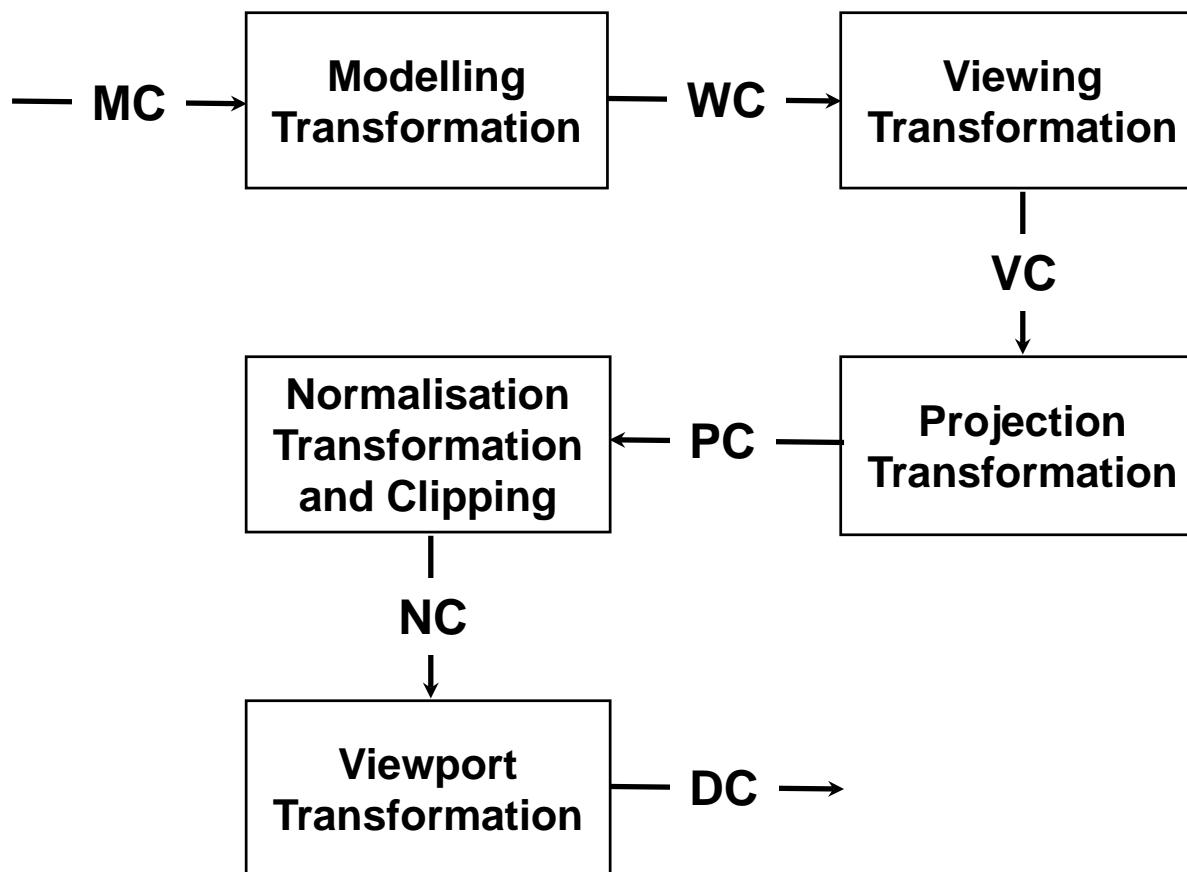
# Topics for today

- Transformation pipeline
- Standard transformations
  - Translation
  - Rotation
  - Scaling
  - Reflection
  - Shearing
- Homogeneous co-ordinate transformation matrices
- Composite (arbitrary) transformation matrices from simple transformations
- OpenGL functions for transformations

# Transformation pipeline (1)

- The Transformation Pipeline is the series of transformations (alterations) that must be applied to an object before it can be properly displayed on the screen.
- The transformations can be thought of as a set of processing stages. If a stage is omitted, very often the object will not look correct. For example if the *projection* stage is skipped then the object will not appear to have any depth to it.
- Once an object has passed through the pipeline it is ready to be displayed as either a wire-frame item or as a solid item.

# Transformation pipeline (2)



# Transformation pipeline (3)

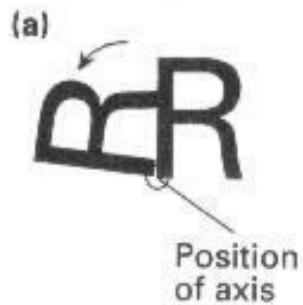
- Modelling Transformation - to place an object into the Virtual World.
- Viewing Transformation - to view the object from a different vantage point in the virtual world.
- Projection Transformation - to see depth in the object.
- Viewport Transformation - to temporarily map the volume defined by the "window of interest" plus the front and rear clipping planes into a unit cube. When this is the case, certain other operations are easier to perform.
- Device Transformation - to map the user defined "window of interest" (in the virtual world) to the dimensions of the display area.

# Transformation pipeline (4)

- We start when the object is loaded from a file and is ready to be processed.
- We finish when the object is ready to be displayed on the computer screen.
- You should be able to draw a picture of a simple object, say a cuboid, and show visually what happens to it as it passes through each pipeline stage.

# Types of geometric transformation

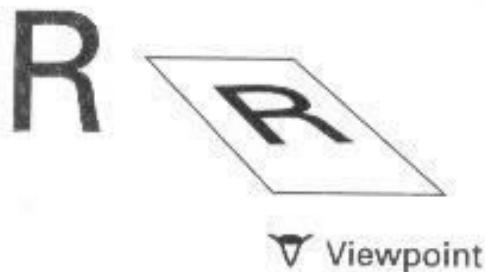
Rotation



Scaling



(e)



(b)



Translation

(d)



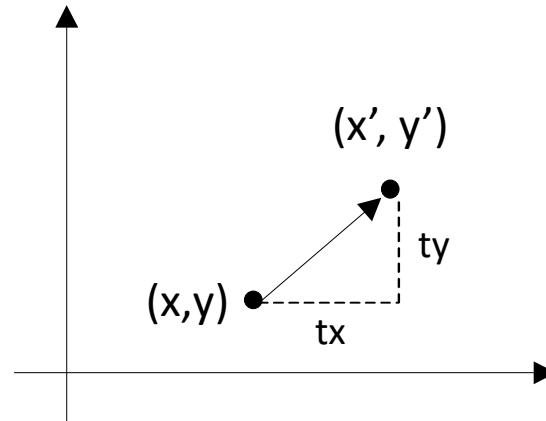
Reflection

Shearing

# 2D translation (1)

- Translating a point from  $P(x, y)$  to  $P'(x', y')$  along vector  $T$
- Importance in computer graphics – we need to only transform the two endpoints of a line segment and let the implementation draw the line segment between the transformed endpoints

$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$



# 2D translation (2)

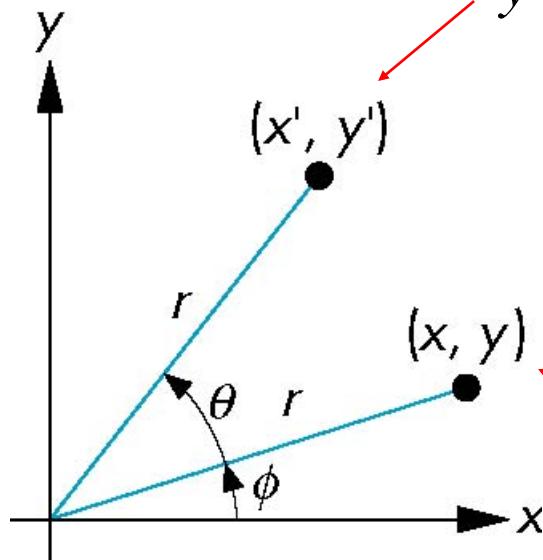
$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

where  $\mathbf{P}(x, y)$  and  $\mathbf{P}'(x', y')$  are the original and new positions, and  $\mathbf{T}$  is the distance translated.

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \quad \text{or} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

# 2D rotation (1)

Rotating a point from  $P(x, y)$  to  $P'(x', y')$  about the origin by angle  $\theta$ - radius stays the same, and angle increases by  $\theta$ .



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

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

$$x = r\cos\phi$$
$$y = r\sin\phi$$

# 2D rotation (2)

$$\begin{cases} x' = r \cos(\phi + \theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta \\ y' = r \sin(\phi + \theta) = r \cos \phi \sin \theta + r \sin \phi \cos \theta \end{cases}$$

$$\begin{cases} x = r \cos \phi \\ y = r \sin \phi \end{cases}$$

$$\begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$

# 2D rotation (3)

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P}$$

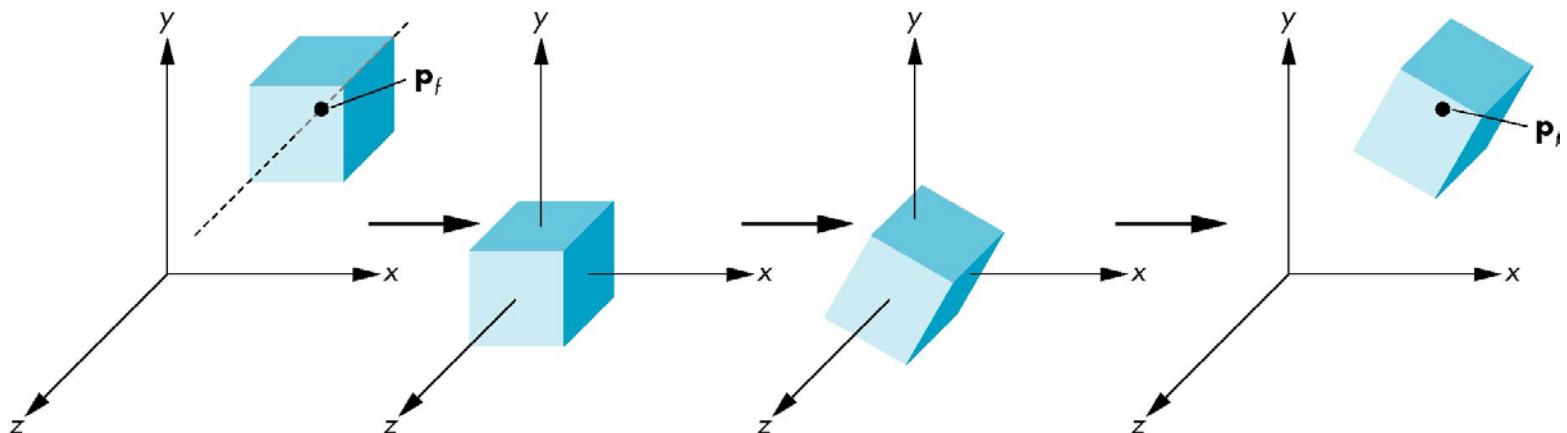
$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad \text{so} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where  $\theta$  is the rotation angle and  $\phi$  is the angle between the  $x$ -axis and the line from the origin to  $(x, y)$ .

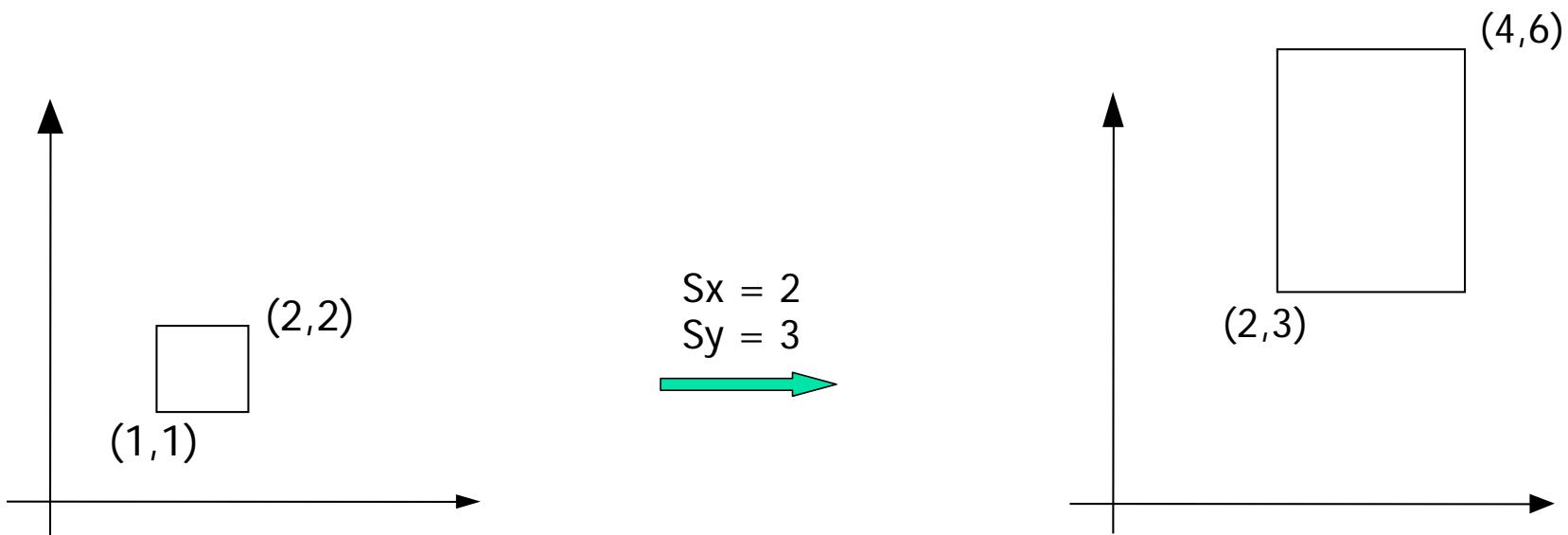
Notice that the rotation point (or pivot point) is the coordinate origin.

# Rotation about a fixed point rather than the origin

- Move the fixed point to the origin
- Rotate the object
- Move the fixed point back to its initial position
- $\mathbf{M} = \mathbf{T}(p_f) \mathbf{R}(\theta) \mathbf{T}(-p_f)$



# 2D scaling (1)



When an object is scaled, both the size and location change.

# 2D scaling (2)

$$\begin{cases} x' = x \cdot s_x \\ y' = y \cdot s_y \end{cases}$$

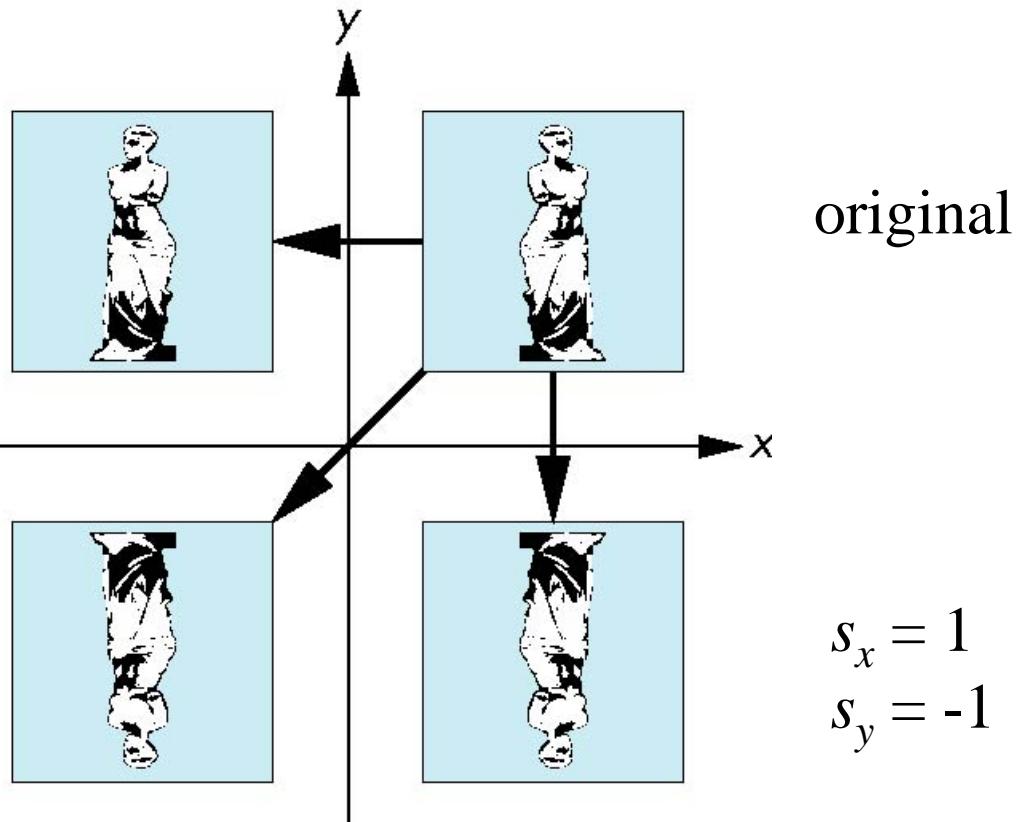
$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad \text{so} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

where  $\mathbf{P}$ , and  $\mathbf{P}'$  are the original and new positions, and  $s_x$  and  $s_y$  are the scaling factors along the  $x$ - and  $y$ -axes.

# 2D reflection

Special case of scaling - corresponding to negative scale factors.

$$s_x = -1$$
$$s_y = 1$$

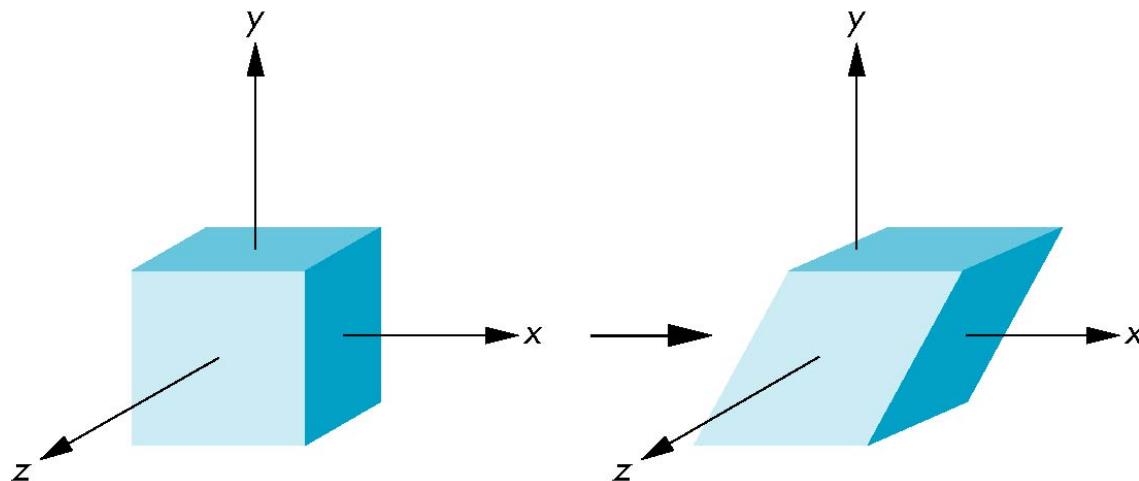


$$s_x = -1$$
$$s_y = -1$$

$$s_x = 1$$
$$s_y = -1$$

# 2D shearing (1)

Equivalent to pulling faces in opposite directions.



# 2D shearing (2)

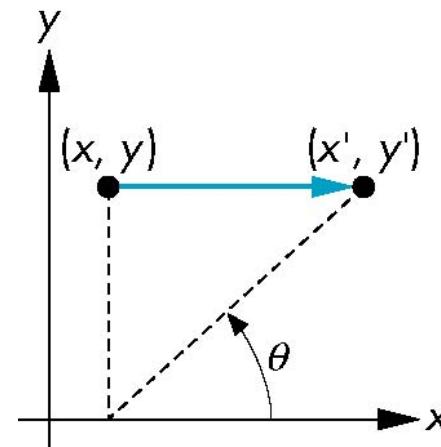
Consider simple shearing along the  $x$  axis.

$$x' = x + y \cot \theta$$

$$y' = y$$

$$z' = z$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \cot \theta \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$



# 2D homogeneous co-ordinates

- By expanding 2x2 matrices to 3x3 matrices, homogeneous co-ordinates are used.
- A homogeneous parameter is applied so that each 2D position is represented with homogeneous co-ordinates ( $h \cdot x, h \cdot y, h$ ).
- The homogeneous parameter has a non-zero value, and can be set to 1 for convenient use.

# 2D homogeneous matrices (1)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2D \text{ translation})$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2D \text{ rotation})$$

# 2D homogeneous matrices (2)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2D \text{ scaling})$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \cot \theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2D \text{ shearing})$$

# 2D composite transformation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where elements  $rs$  are the multiplicative rotation-scaling terms in the transformation (which involve only rotation angles and scaling factors);

elements  $trs$  are the translation terms, containing combination of translation distances, pivot-point and fixed-point co-ordinates, rotation angles and scaling parameters.

# 3D translation

3D translations and scaling can be simply extended from the corresponding 2D methods.

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

# 3D co-ordinate axis rotations (1)

- The extension from 2D rotation methods to 3D rotation is less straightforward (because this is about an arbitrary axis instead of an arbitrary point).
- Equivalent to rotation in two dimensions in planes of constant  $z$  (i.e. about the origin).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (\text{About } z\text{-axis})$$

# 3D co-ordinate axis rotations (2)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (\text{About } y\text{-axis})$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (\text{About } x\text{-axis})$$

# General rotation about the origin

A rotation by  $q$  about an arbitrary axis can be decomposed into the concatenation of rotations about the  $x$ ,  $y$ , and  $z$  axes.

$$\mathbf{R}(q) = \mathbf{R}_z(q_z) \mathbf{R}_y(q_y) \mathbf{R}_x(q_x)$$

where  $q_x$ ,  $q_y$  and  $q_z$  are called the Euler angles.

Note that rotations do not commute though we can use rotations in another order but with different angles.

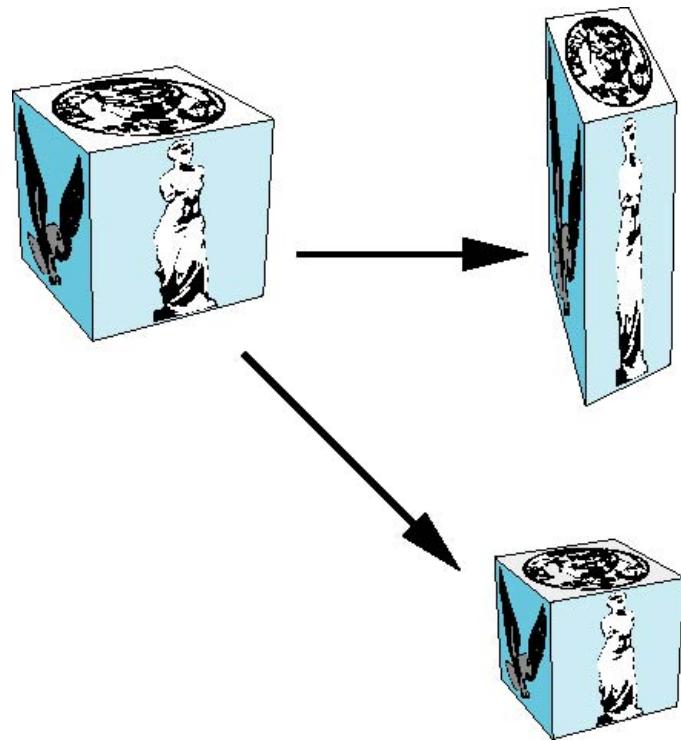
# 3D scaling

$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



# 3D composite transformation (1)

- As with 2D transformation, a composite 3D transformation can be formed by multiplying the matrix representations for the individual operations in the transformation sequence.
- There are other forms of transformation, namely reflection and shearing which can be implemented with the other three transformations.
- Translation, scaling, rotation, reflection and shearing are all affine transformations in that transformed point  $P'(x',y',z')$  is a **linear combination** of the original point  $P(x,y,z)$ .

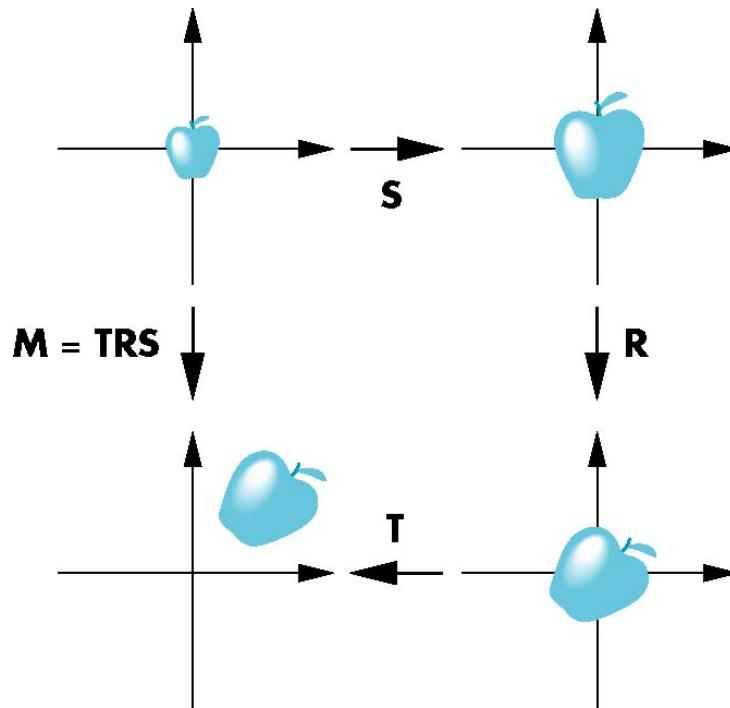
# 3D composite transformation (2)

- Matrix multiplication is associative
- $M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1)$
  
- Transformation products are  
not always commutative

$$A \cdot B \neq B \cdot A$$

# Instancing

- In modelling, we often start with a simple object centred at the origin, oriented with an axis, and of a standard size.
- We apply an *instance transformation* to its vertices to
  - Scale
  - Orient
  - Locate



# OpenGL matrices

- In OpenGL matrices are part of the state.
- Multiple types
  - Model-View (**GL\_MODELVIEW**)
  - Projection (**GL\_PROJECTION**)
  - Texture (**GL\_TEXTURE**)
  - Color (**GL\_COLOR**)
- Single set of functions for manipulation
- Select which to be manipulated by
  - **glMatrixMode(GL\_MODELVIEW);**
  - **glMatrixMode(GL\_PROJECTION);**

# Current transformation matrix

- Conceptually there is a 4x4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline.
- The CTM is defined in the user program and loaded into a transformation unit.
- The CTM can be altered either by loading a new CTM or by postmultiplication.
- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM.
- The CTM can manipulate each by first setting the correct matrix mode.

# CTM operations

The CTM can be altered either by loading a new CTM or by postmultiplication

Load an identity matrix:  $\mathbf{C} \leftarrow \mathbf{I}$

Load an arbitrary matrix:  $\mathbf{C} \leftarrow \mathbf{M}$

Load a translation matrix:  $\mathbf{C} \leftarrow \mathbf{T}$

Load a rotation matrix:  $\mathbf{C} \leftarrow \mathbf{R}$

Load a scaling matrix:  $\mathbf{C} \leftarrow \mathbf{S}$

Postmultiply by an arbitrary matrix:  $\mathbf{C} \leftarrow \mathbf{CM}$

Postmultiply by a translation matrix:  $\mathbf{C} \leftarrow \mathbf{CT}$

Postmultiply by a rotation matrix:  $\mathbf{C} \leftarrow \mathbf{CR}$

Postmultiply by a scaling matrix:  $\mathbf{C} \leftarrow \mathbf{CS}$

# Arbitrary Matrices

- We can load and multiply by matrices defined in the application program
  - `glLoadMatrixf(m)`
  - `glMultMatrixf(m)`
- Matrix `m` is a one-dimension array of 16 elements which are the components of the desired  $4 \times 4$  matrix stored by columns.
- In `glMultMatrixf`, `m` multiplies the existing matrix on the right.

# Matrix stacks

- CTM is not just one matrix but a matrix stack with the “current” at top.
- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures
  - Avoiding state changes when executing display lists
- Pre 3.1 OpenGL maintains stacks for each type of matrix
  - Access present type (as set by `glMatrixMode`) by  
`glPushMatrix()`  
`glPopMatrix()`
- Right now just 1-level CTM.

# Matrix stacks

- We can also access matrices (and other parts of the state) with *query* functions
  - `glGetIntegerv`
  - `glGetFloatv`
  - `glGetBooleanv`
  - `glGetDoublev`
  - `glIsEnabled`
- For matrices, we use as
  - `double m[16];`
  - `glGetFloatv(GL_MODELVIEW, m);`

# OpenGL transformation functions (1)

- **glTranslate\***

Specify translation parameters

- **glRotate\***

Specify rotation parameters for rotation about any axis through the origin

- **glScale\***

Specify scaling parameters with respect to the co-ordinate origin

- **glMatrixMode**

Specify current matrix for geometric-viewing, projection, texture or colour transformations

- **glLoadIdentity**

Set current matrix to identity

- **glLoadMatrix\* (elems)**

Set elements of current matrix

# OpenGL transformation functions (2)

- **glMultMatrix\*(elems)**  
Post-multiply the current matrix by the specified matrix
- **glGetIntegerv**  
Get max stack depth or current number of matrices in the stack for selected matrix mode
- **glPushMatrix**  
Copy the top matrix in the stack and store copy in the second stack position
- **glPopMatrix**  
Erase top matrix in stack and move second matrix to top stack
- **glPixelZoom**  
Specify 2D scaling parameters for raster operations

# Example of rotation

- Rotation about the  $z$  axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(1.0, 2.0, 3.0);  
glRotatef(30.0, 0.0, 0.0, 1.0);  
glTranslatef(-1.0, -2.0, -3.0);
```

- Note that the last matrix specified in the program is the first applied.

# Summary

- Transformation pipeline
- Standard transformations
  - Rotation
  - Translation
  - Scaling
  - Reflection
  - Shearing
- Homogeneous co-ordinate transformation matrices
- Composite (arbitrary) transformation matrices from simple transformations
- OpenGL functions for transformations



Xi'an Jiaotong-Liverpool University  
西交利物浦大学

# CPT205 Computer Graphics

# Viewing and Projection

Week 05  
2021-22

Yong Yue

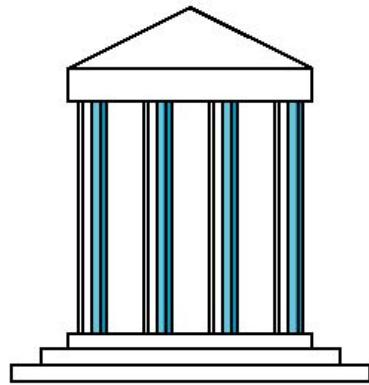
# Topics for today

- Concepts of viewing and projections
- Types and advantages/disadvantages of projection
- 3D viewing co-ordinate parameters
- Orthogonal projection
- Frustum perspective projection
- OpenGL functions
- Sample code

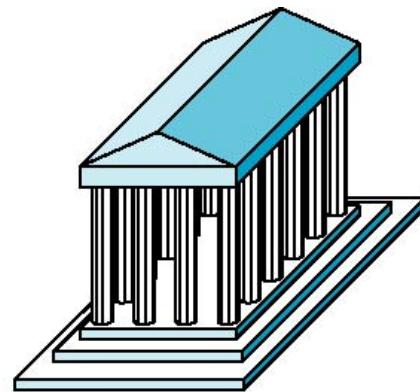
# Classic viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients the object in a way that it is to be seen.
- Each object is constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects, etc.

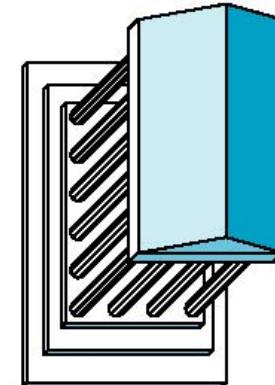
# Classic projection



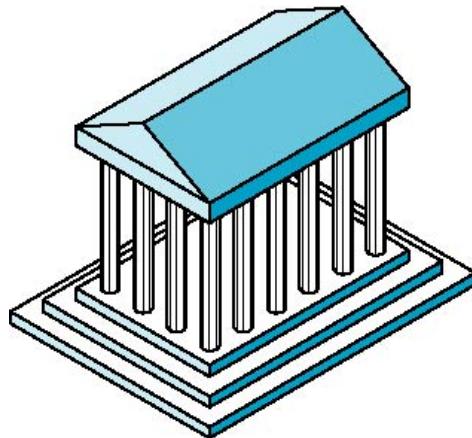
Front elevation



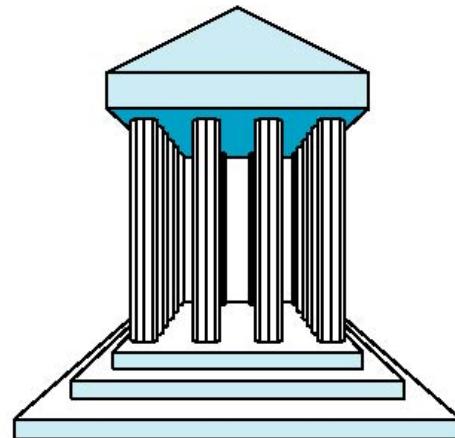
Elevation oblique



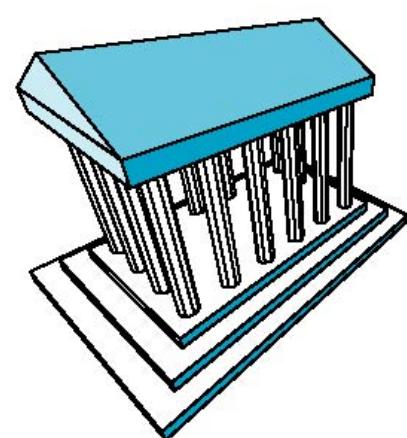
Plan oblique



Isometric



One-point perspective

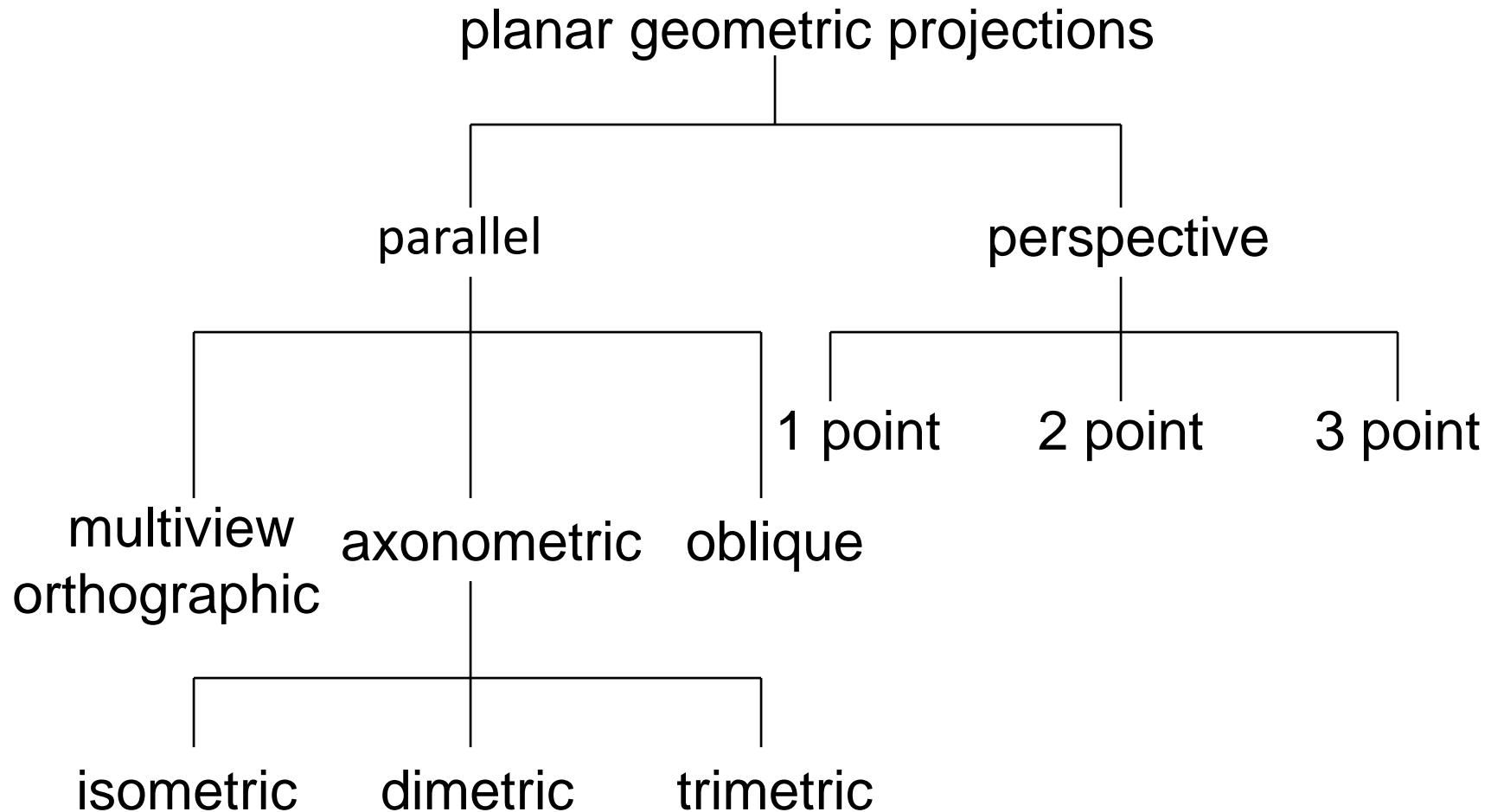


Three-point perspective

# Planar geometric projection

- Standard projections project onto a plane.
- Projectors are lines that either converge at a centre of projection or are parallel.
- Such projections preserve lines but not necessarily angles.
- Non-planar projections are needed for applications such as map construction.

# Taxonomy of planar geometric projection



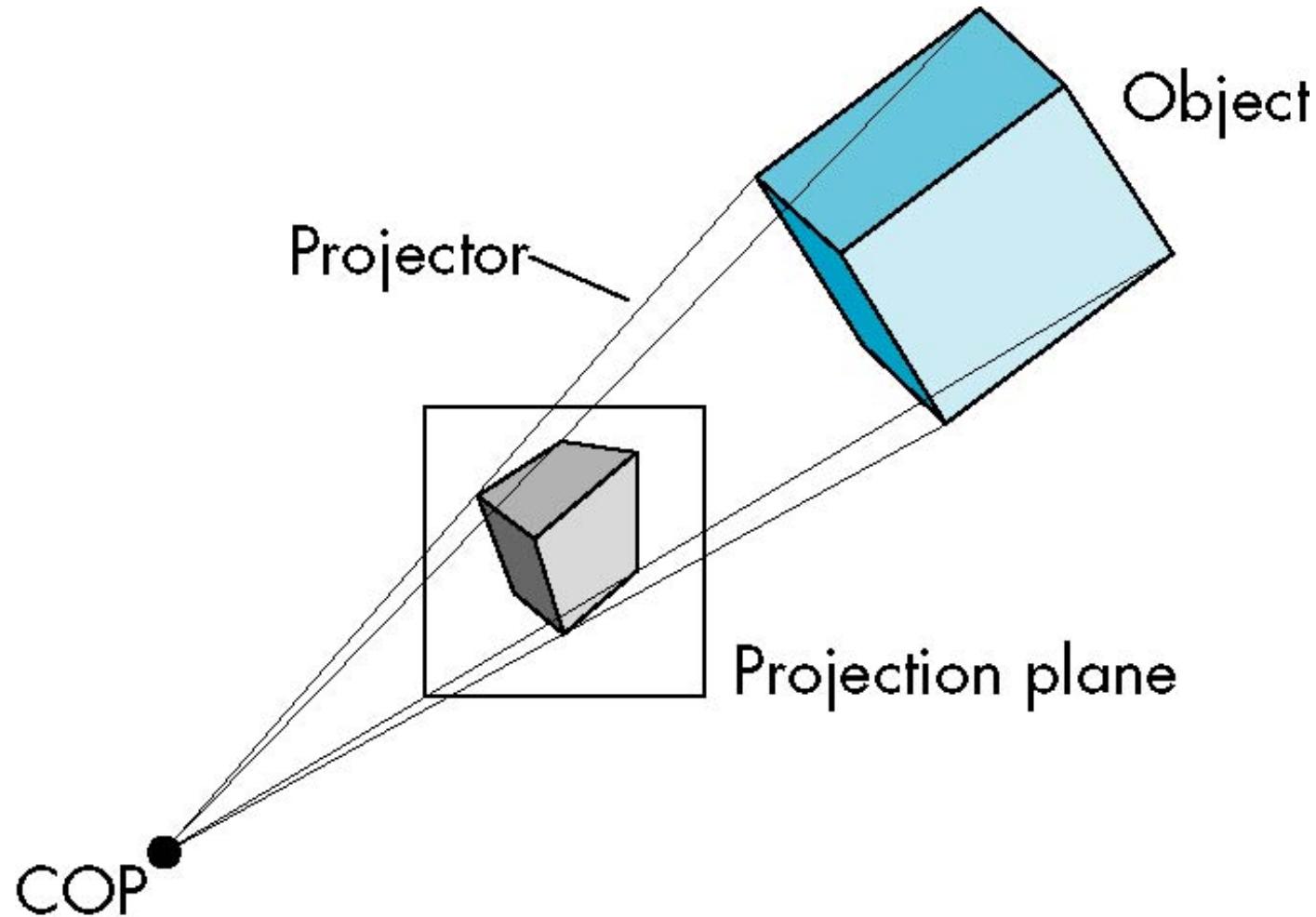
# Perspective vs parallel

- Computer graphics treats all projections in the same way and implements them with a single pipeline.
- Classical viewing has developed different techniques for drawing each type of projection.
- The fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing.

# Perspective projection

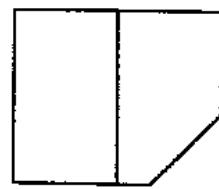
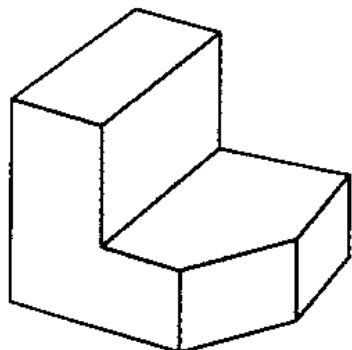
- Perspective projection generates a view of 3-dimensional scene by projecting points to the view plane along converging paths, causing the objects farther from the viewing position to be displayed smaller than the objects of the same size that are nearer to the viewing position.
- A scene generated using perspective projection appears more realistic since this is the way that human eyes and cameras form images.

# Perspective projection

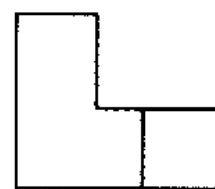


# Parallel projection

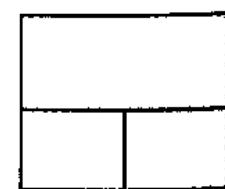
- This method projects points on the object surface along parallel lines.
- It is usually used in engineering and architecture drawings to represent an object with a set of views showing accurate dimensions.



Top

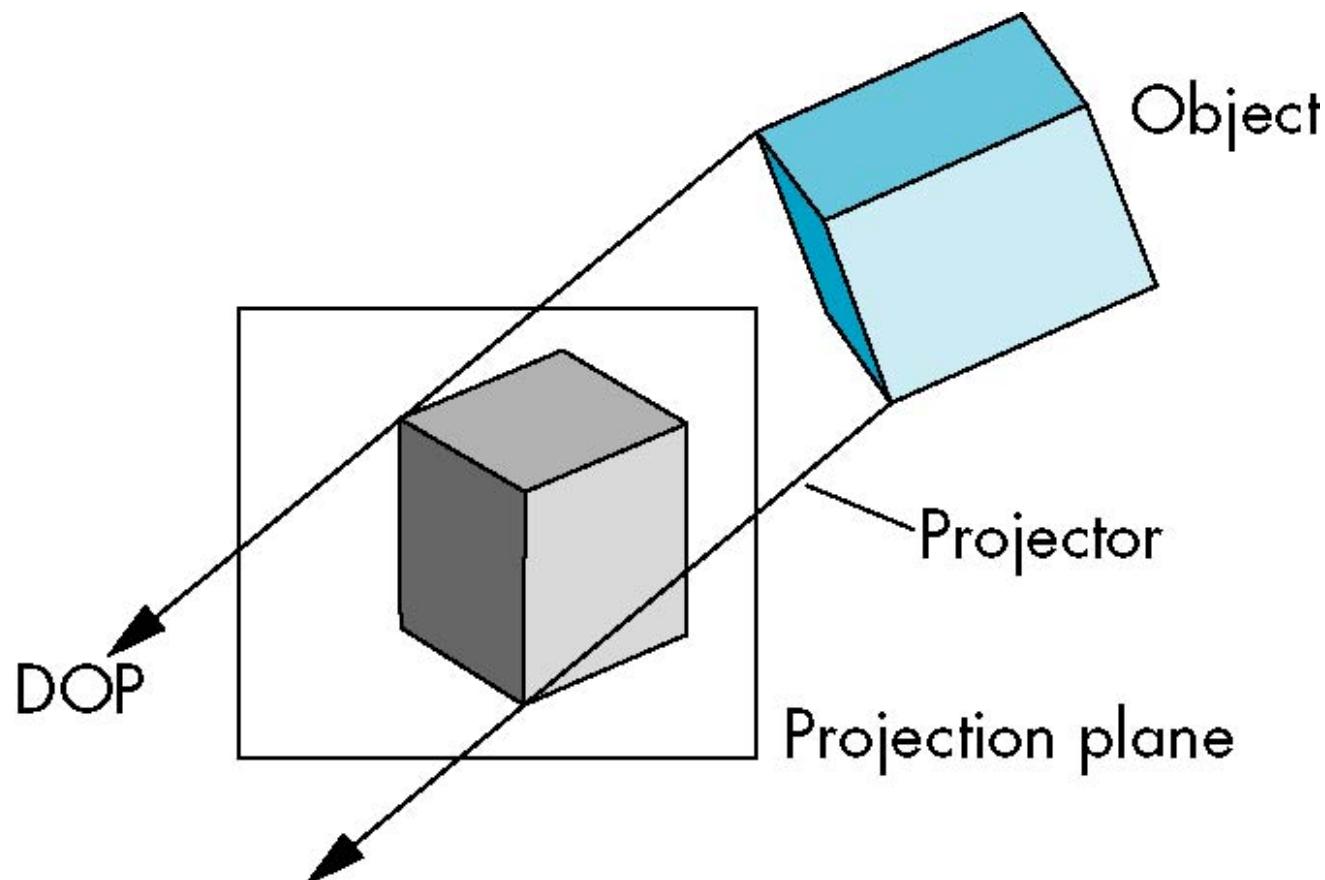


Side



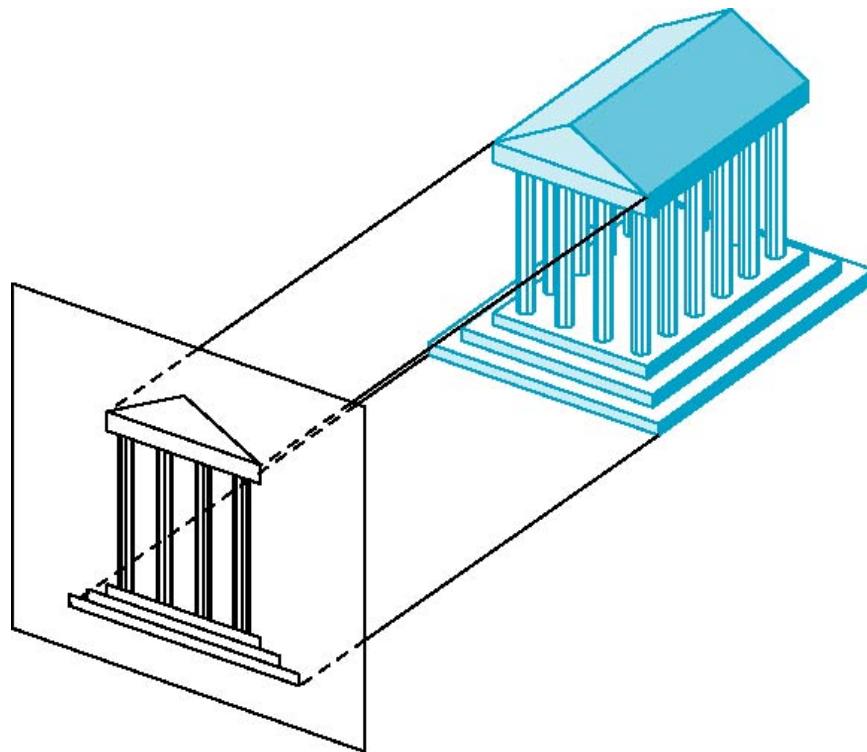
Front

# Parallel projection



# Orthographic projection

The projectors are orthogonal to projection surface.



# Multiview orthographic projections

- The projection plane is parallel to the principal face.
- Usually form front, top and side views.

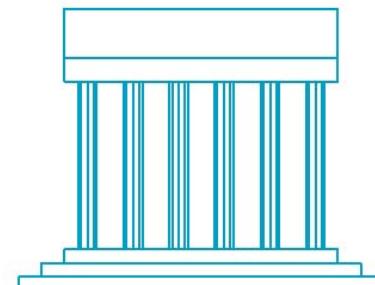
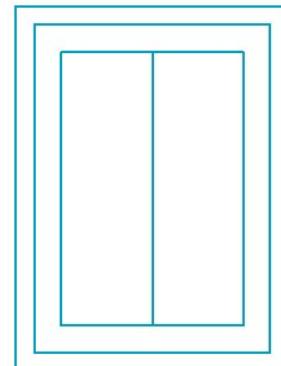
Isometric (not multiview orthographic view)



Front

In CAD and architecture, we often display three multiviews plus isometric

Top



Side

# Advantages and disadvantages

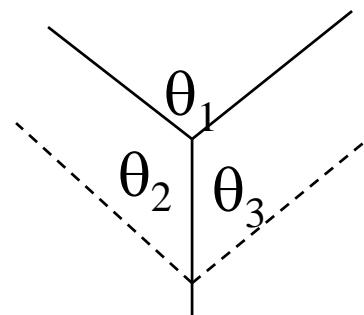
- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces are hidden from the view
  - Often the isometric view is added

# Axonometric projections

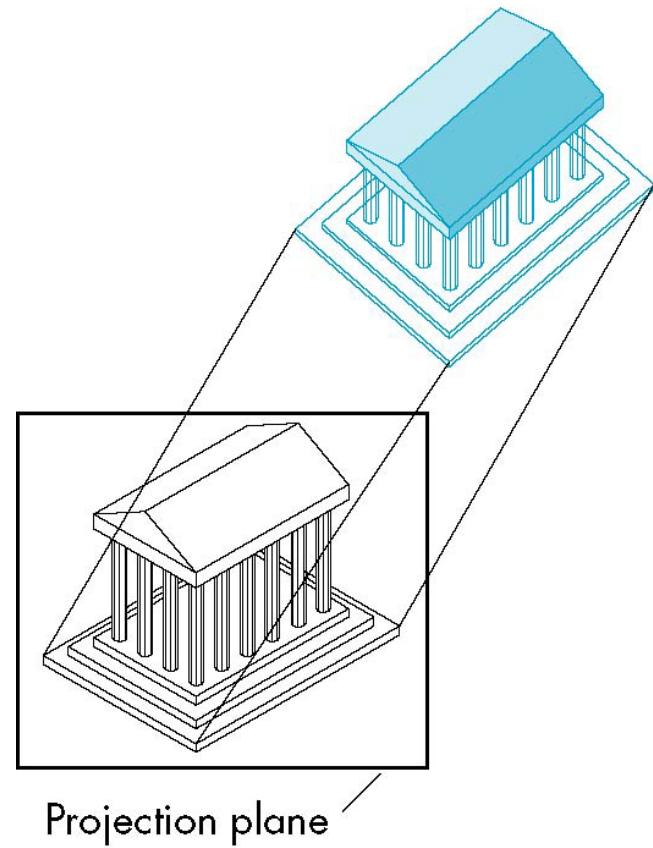
Allow projection plane to move relative to the object.

Classified by the number of angles of a corner of a projected cube

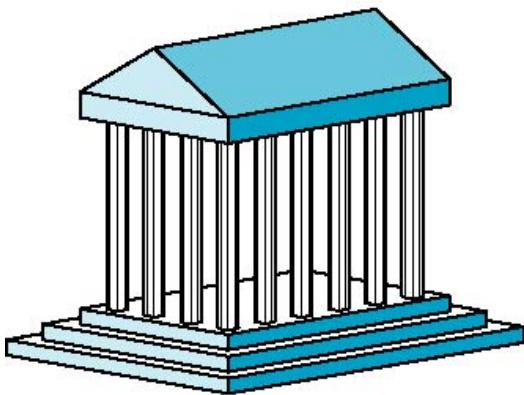
- One: trimetric
- Two: dimetric
- Three: isometric



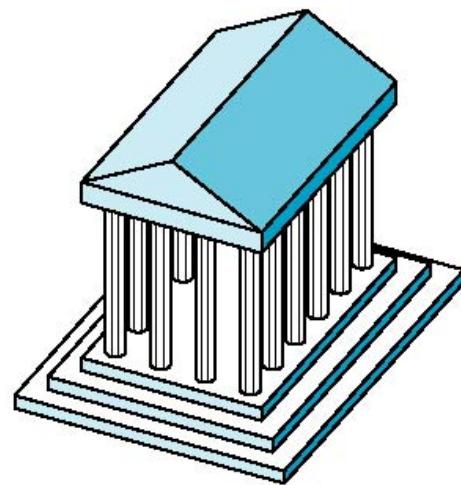
Refer the textbook for more detail.



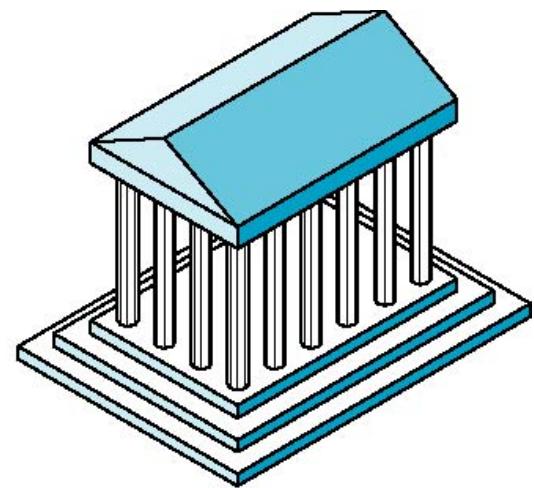
# Axonometric projections



Dimetric



Trimetric



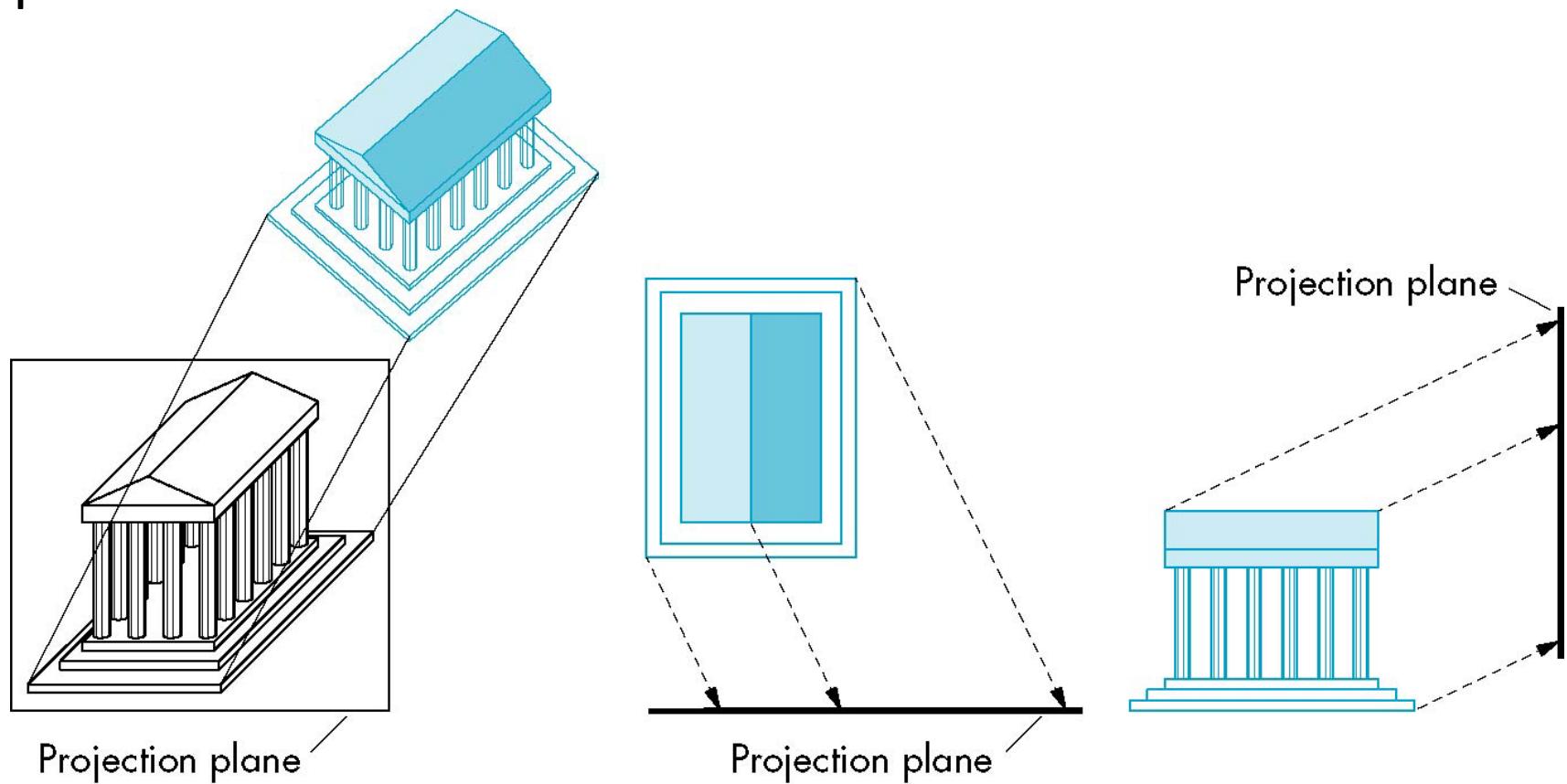
Isometric

# Advantages and disadvantages

- Lines are scaled (*foreshortened*) but can find scaling factors
- Lines preserved but angles are not
  - Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Can see three principal faces of a box-like object
- Some optical illusions possible
  - Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

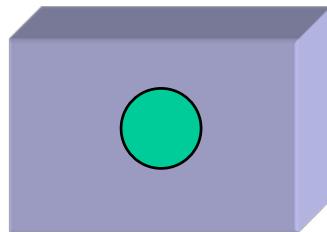
# Oblique projection

Arbitrary relationship between projectors and projection plane.



# Advantages and disadvantages

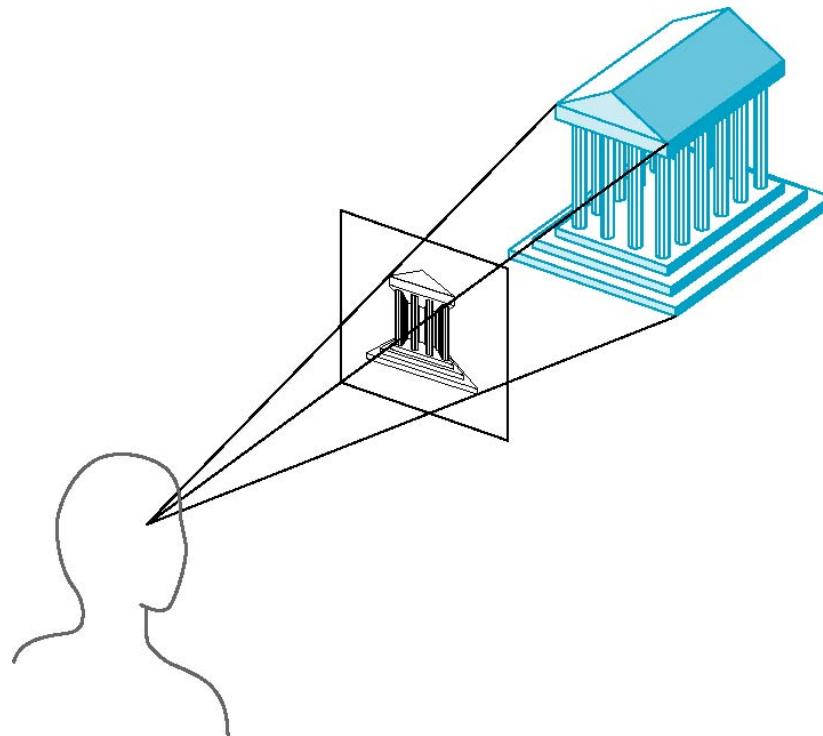
- Can pick the angles to emphasise a particular face
  - Architecture: plan oblique, elevation oblique
- Angles in faces parallel to the projection plane are preserved while we can still see “around” side.



- In the physical world, we cannot create oblique projections with a simple camera; possible with bellows camera or special lens (architectural).

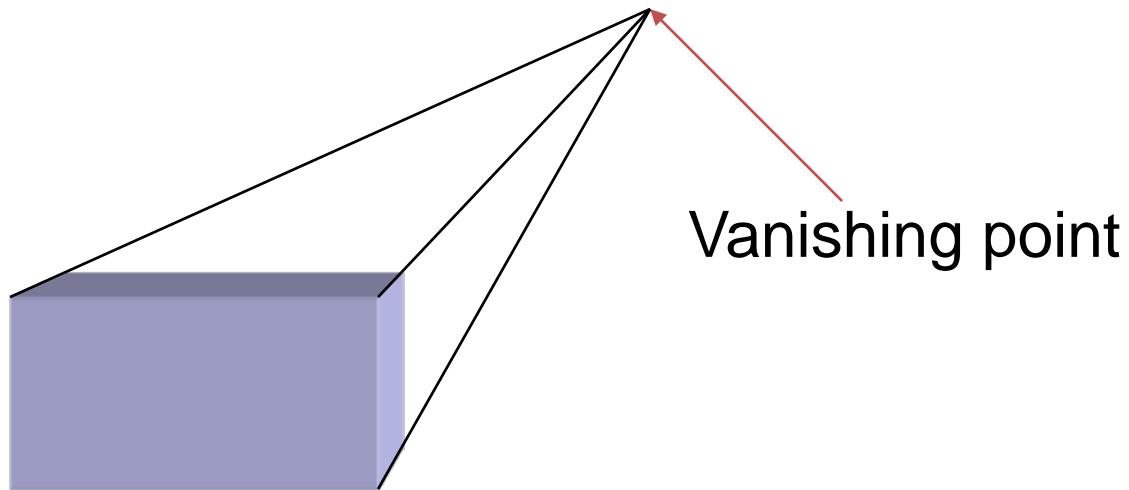
# Perspective projection

Projectors converge at the centre of projection (CoP).



# Vanishing points

- Parallel lines (not parallel to the projection plane) on the object converge at a single point in the projection (the *vanishing point*).
- Drawing simple perspectives by hand uses the vanishing point(s).



# Three-point perspective

- No principal face parallel to the projection plane
  - Three vanishing points for the cube



# Two-point perspective

- One principal direction parallel to the projection plane
- Two vanishing points for the cube



# One-point perspective

- One principal face parallel to the projection plane
- One vanishing point for the cube



# Advantages and disadvantages

- Objects further from the viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*non-uniform foreshortening*).
- Angles preserved only in planes parallel to the projection plane.
- More difficult to construct by hand than parallel projections (but not more difficult by computer).

# Computer viewing and projection

There are three aspects of the viewing process, all of which are implemented in the pipeline:

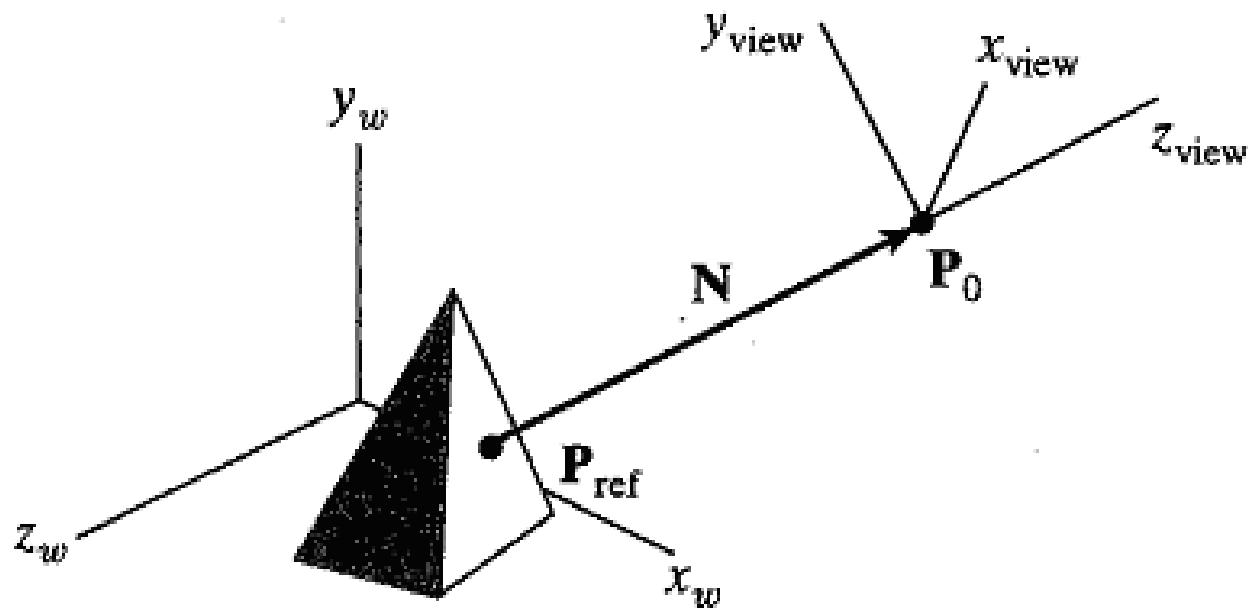
- Positioning the camera
  - Setting the model-view matrix
- Selecting a lens
  - Setting the projection matrix
- Clipping
  - Setting the view volume

# 3D viewing co-ordinate parameters (1)

- A world co-ordinate position  $P_0(x_0, y_0, z_0)$  is selected as the viewing origin (called view point, viewing position, eye position or camera position).
- The view plane can then be defined with a normal vector, which is the viewing direction, usually the negative  $z_{view}$  axis.
- The viewing plane is thus parallel to the  $x_{view}$ - $y_{view}$  plane.

# 3D viewing co-ordinate parameters (2)

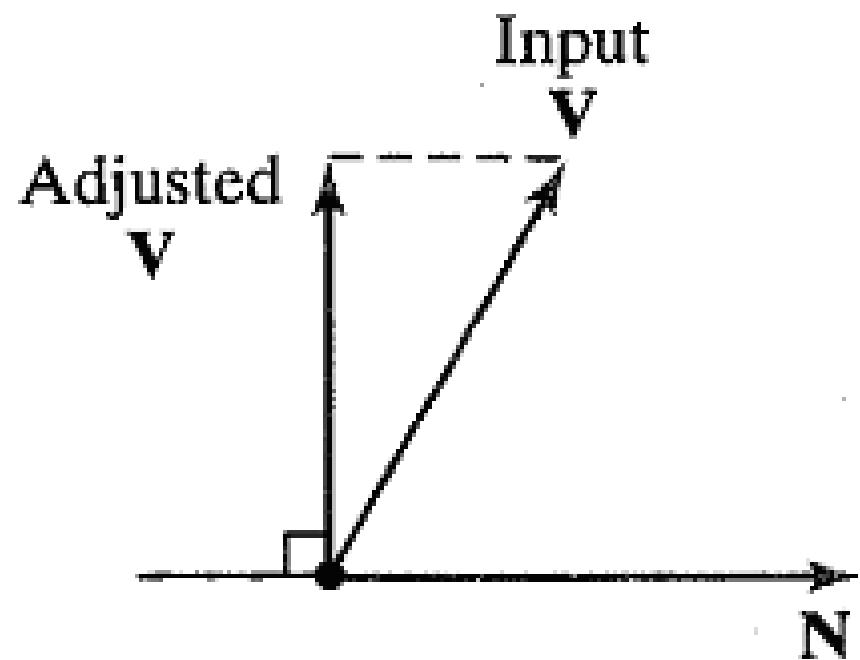
- The direction from a reference point to the viewing point can be taken as the viewing direction (vector), and the reference point ( $x_{\text{ref}}$ ,  $y_{\text{ref}}$ ,  $z_{\text{ref}}$ ) is termed the look-at point.



# 3D viewing co-ordinate parameters (3)

- Once the viewing plane normal vector  $\mathbf{N}$  is defined, the direction for view-up vector  $\mathbf{V}$  can be set to establish the positive  $x_{view}$  axis. Since  $\mathbf{N}$  defines the direction for  $z_{view}$ ,  $\mathbf{V}$  must be perpendicular to  $\mathbf{N}$  (i.e. parallel to the  $x_{view}$ - $y_{view}$  plane).
- But it is generally difficult to determine  $\mathbf{V}$  precisely, and viewing routines typically adjust the user defined value of  $\mathbf{V}$  (e.g. projecting it onto a plane perpendicular to  $\mathbf{N}$ ).
- Any direction can be used for  $\mathbf{V}$  as long as it is not parallel to  $\mathbf{N}$ .

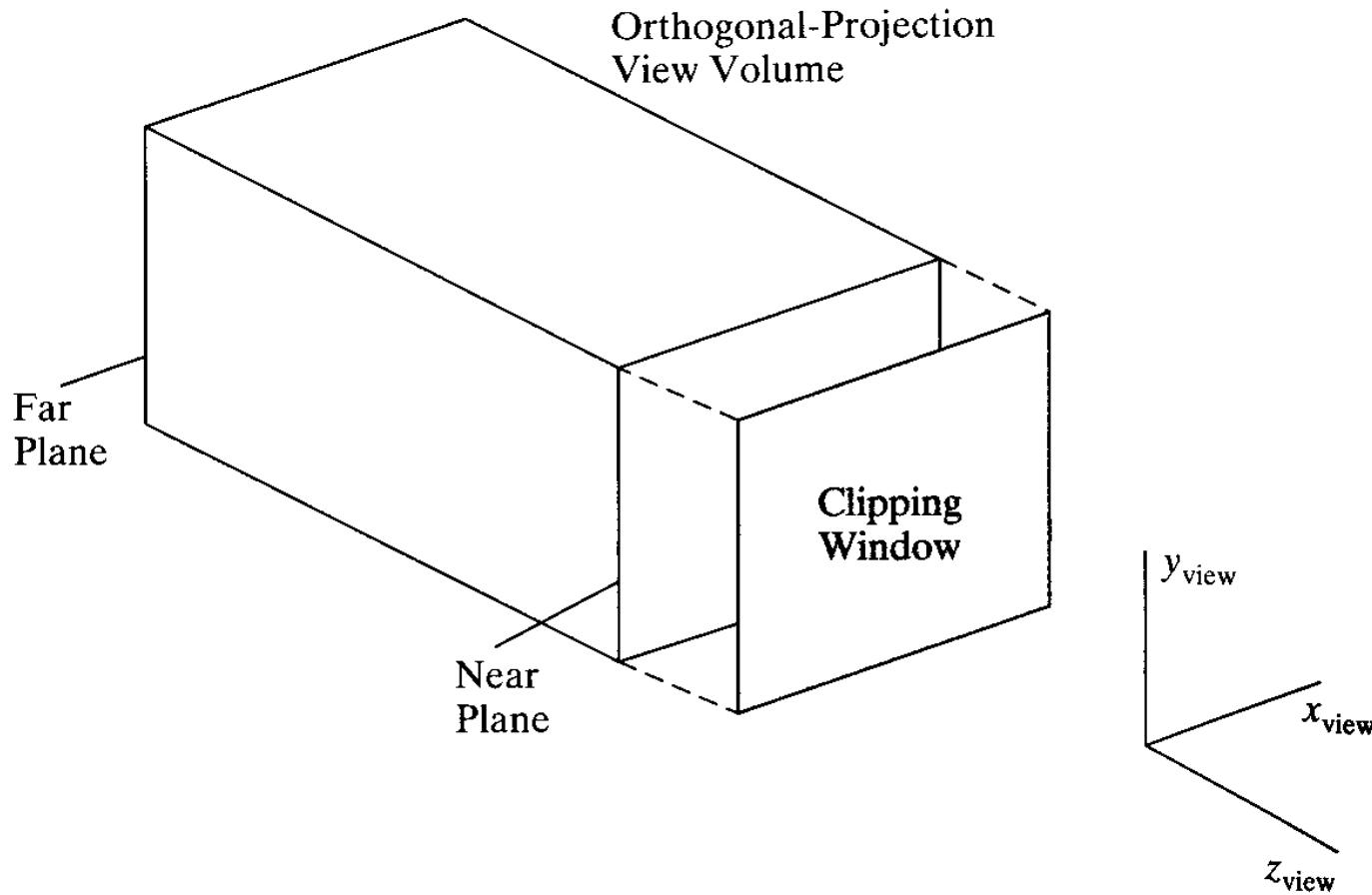
# 3D viewing co-ordinate parameters (4)



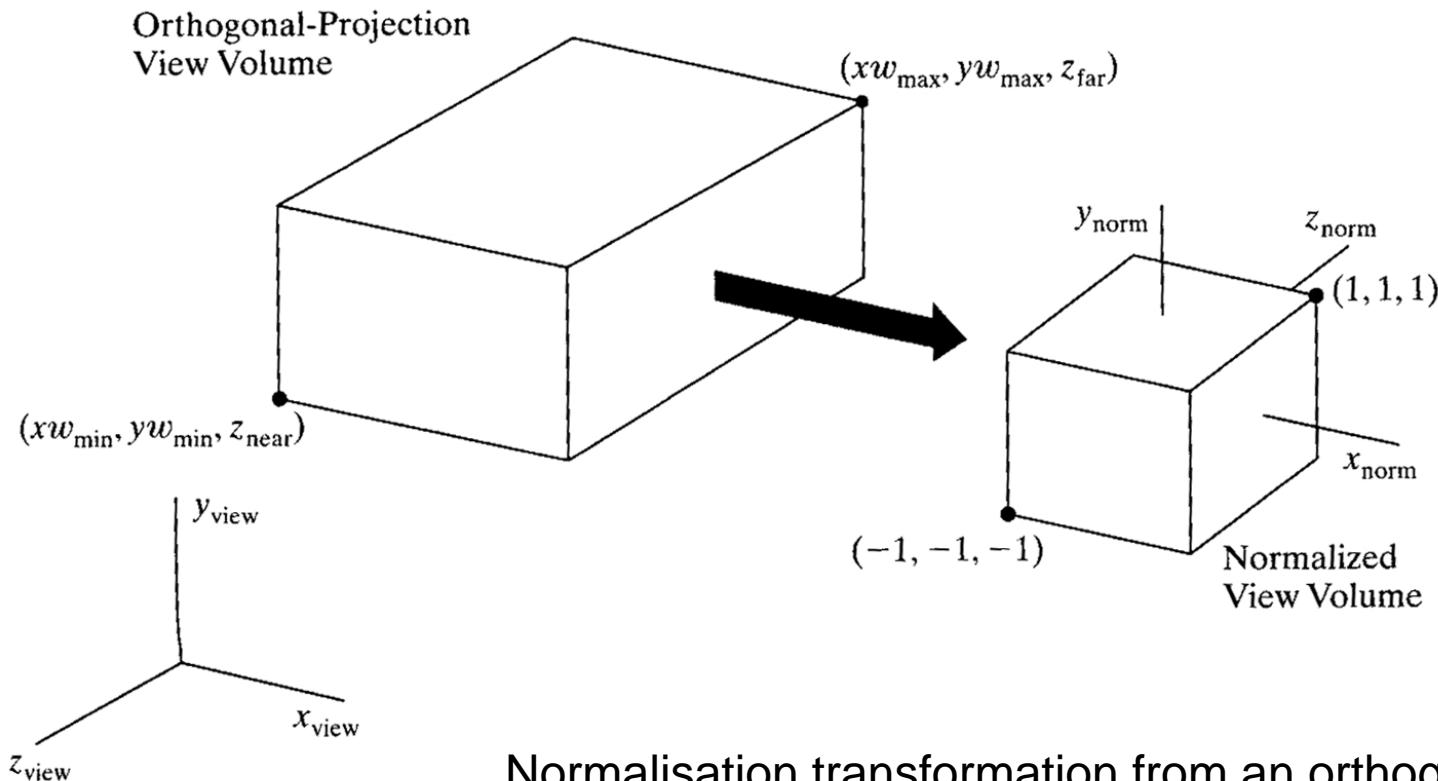
# Orthogonal projection (1)

- Orthogonal (or parallel) projection is a transformation of object descriptions to a view plane along lines parallel to the view-plane normal vector **N**.
- It is often used to produce the front, side and top views of an object.
- Engineering and architectural drawings commonly employ these orthographic projections since the lengths and angles are accurately depicted and can be measured from the drawings.

# Orthogonal projection (2)

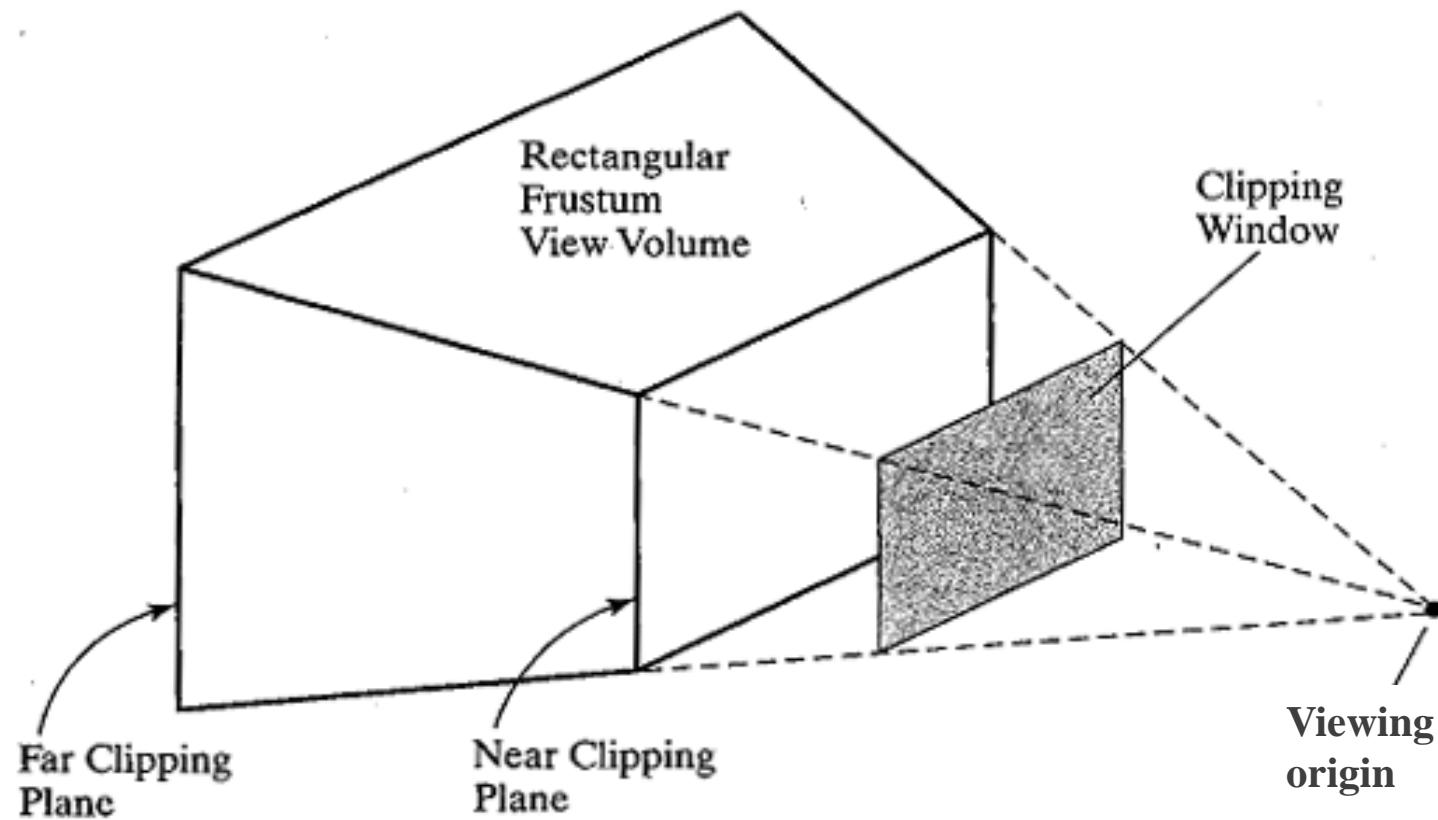


# Orthogonal projection (3)



Normalisation transformation from an orthogonal-projection volume to the systematic normalisation cube within a reference frame

# Frustum perspective projection (1)



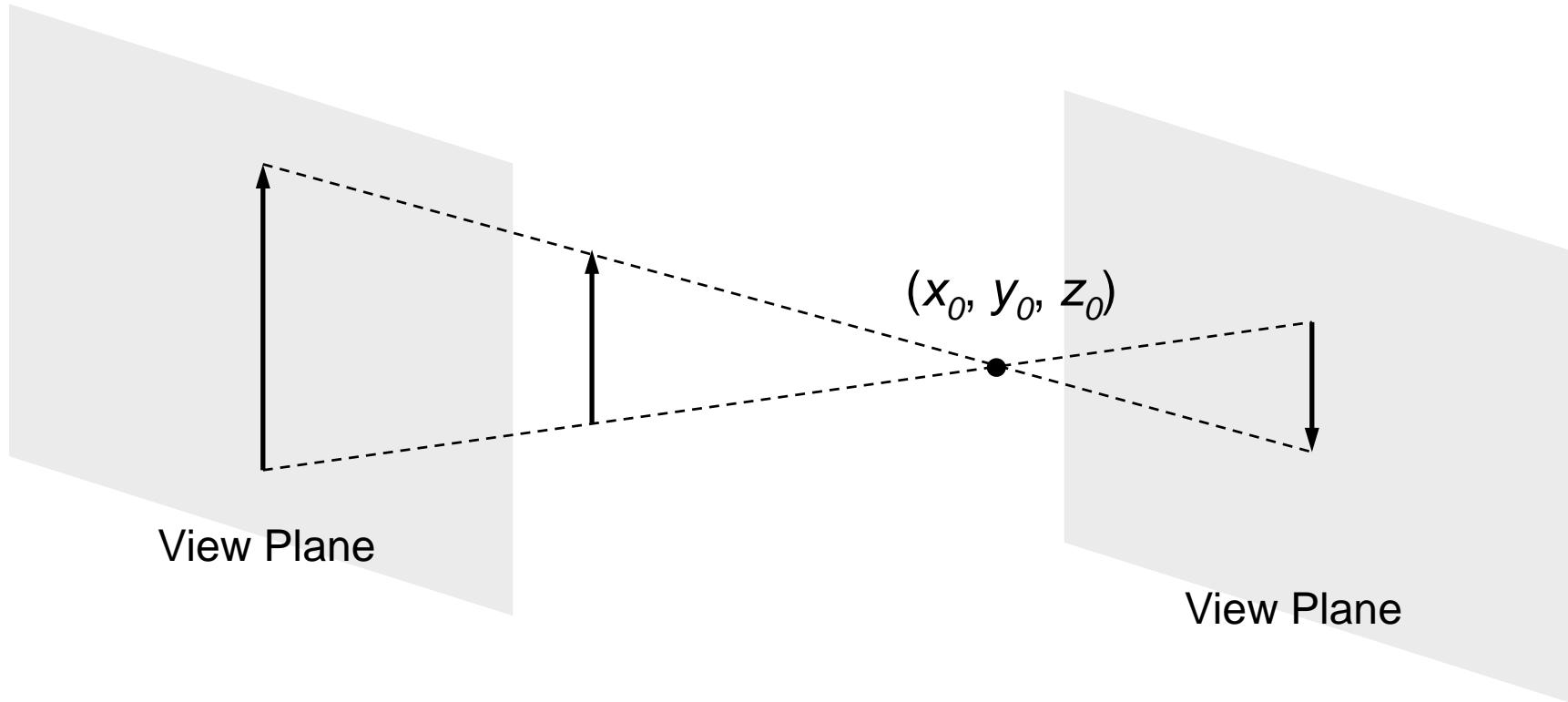
# Frustum perspective projection (2)

- By adding near and far clipping planes that are parallel to the viewing plane, parts of the infinite perspective view volume are chopped off to form a truncated pyramid or frustum. These clipping planes can be optional for some systems.
- The near and far clipping planes can be used simply to enclose objects to be displayed. The near clipping plane can be used to take out large objects close to the viewing plane, which could be projected into unrecognisable shapes in the clipping window. Likewise, the far clipping plane can cut out objects that may be projected to small blots.

# Frustum perspective projection (3)

- Some systems restrict the placement of the viewing plane relative to the near and far planes, and other systems allow it to be placed anywhere except at the position of the viewing origin (view point, viewing position, eye position or camera position).
- If the viewing plane is behind the projection reference point, objects are inverted on the view plane.
- If the viewing plane is behind the objects, the objects are simply enlarged as they are projected away.
- When the projection reference point is very far away from the view plane, a perspective projection approaches to a parallel projection.

# Frustum perspective projection (4)

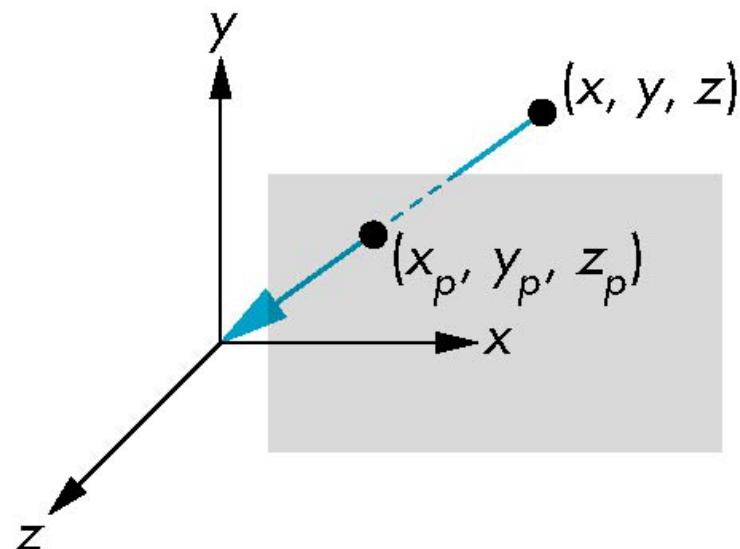


Objects are enlarged if the viewing plane is behind the objects.

Objects are inverted if the viewing plane is behind the projection reference point.

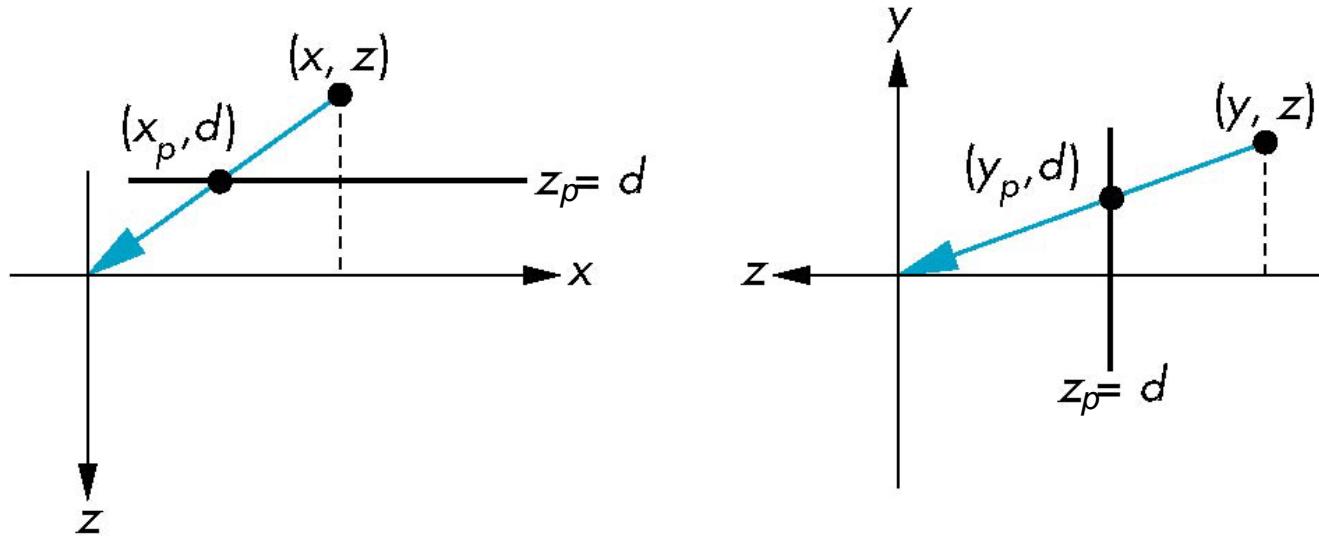
# Simple perspective projection (1)

- Centre of projection at the origin
- Projection plane  $z = d$ ,  $d < 0$



# Simple perspective projection (2)

Consider top and side views



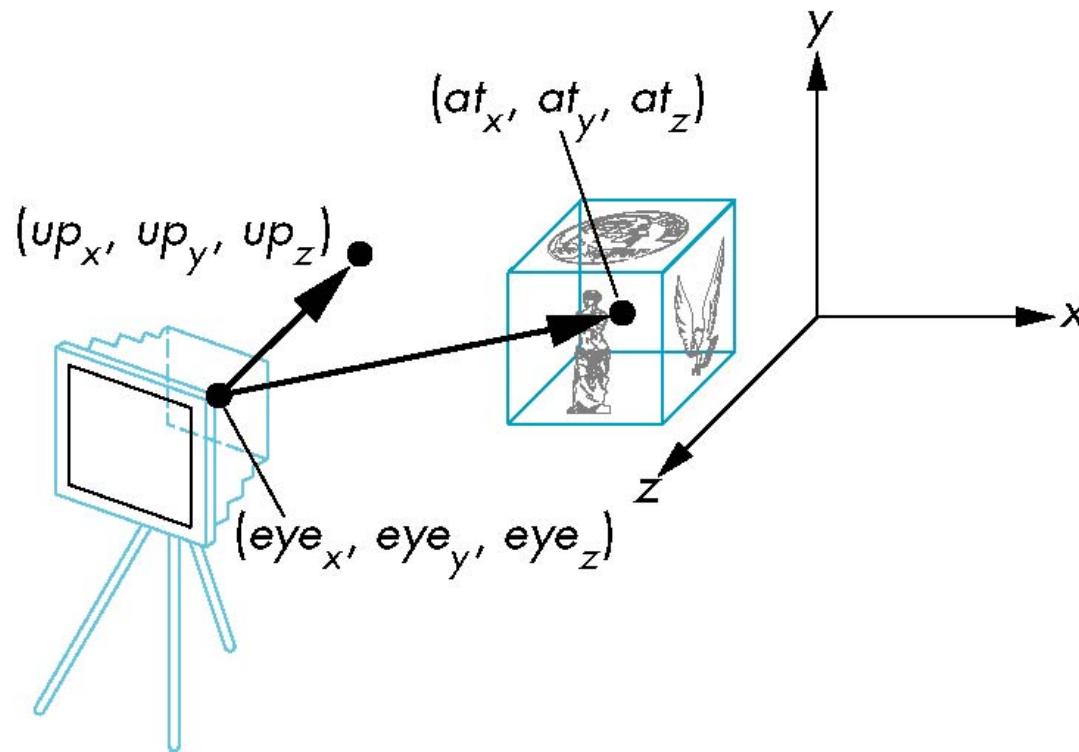
$$x_p = \frac{x}{z/d}$$

$$y_p = \frac{y}{z/d}$$

$$z_p = d$$

# OpenGL functions (1)

- `gluLookAt(eye_position, look_at, look_up)`  
Specify three-dimensional viewing parameters.



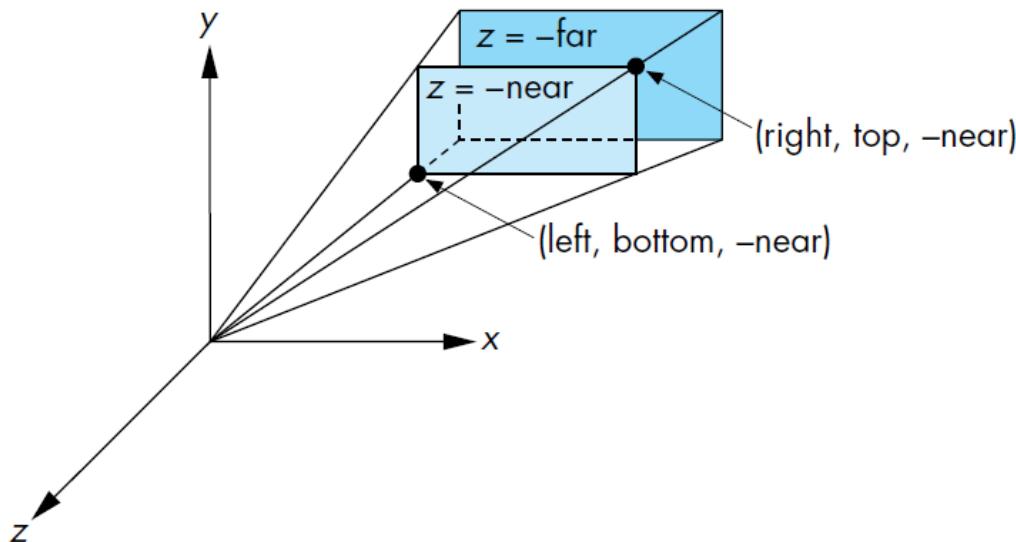
# OpenGL functions (2)

- `glOrtho(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far)`  
Specify parameters for a clipping window and the near and far clipping planes for an orthogonal projection.

# OpenGL functions (3)

- **glFrustum(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far)**

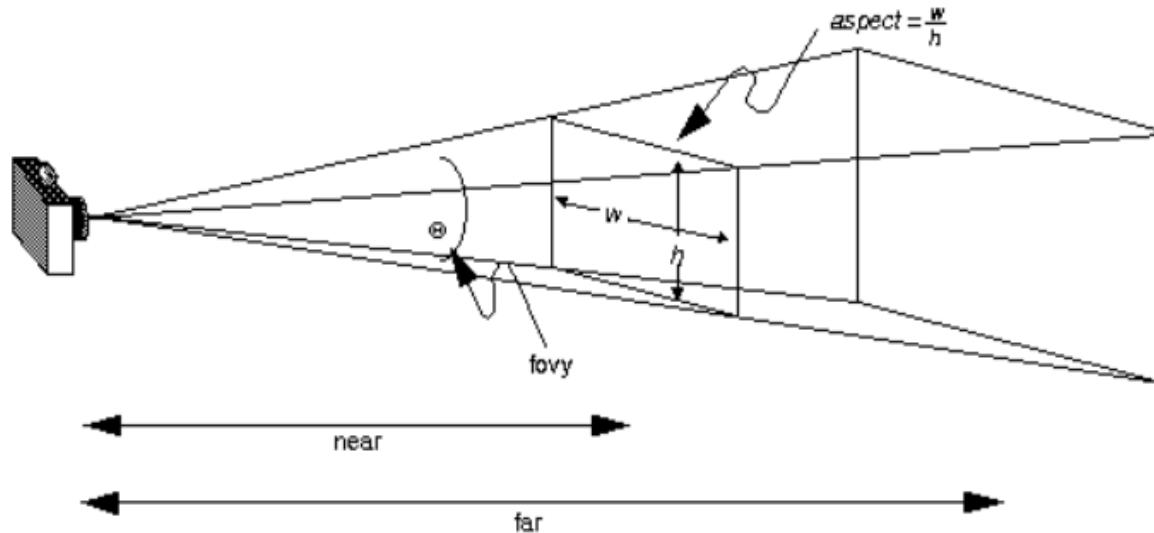
Specify parameters for a clipping window and the near and far clipping planes for a perspective projection (symmetric or oblique).



# OpenGL functions (4)

- **gluPerspective(GLfloat fov, GLfloat aspect, GLfloat near, GLfloat far)**

Specify field-of-view angle (**fov** which is a matrix) in the y-direction, the **aspect** ratio of **near** and **far** planes; It is less often used than **glFrustum()**.



# An example of OpenGL program (1)

```
#define FREEGLUT_STATIC
#include <gl/freeglut.h>

GLint winWidth = 600, winHeight = 600;           // initial display window size

GLfloat x0 = 100.0, y0 = 50.0, z0 = 50.0;         // viewing co-ordinate origin
GLfloat xref = 50.0, yref = 50.0, zref = 0.0;       // look-at point
GLfloat Vx = 0.0, Vy = 1.0, Vz = 0.0;             // view-up vector

// co-ordinate limits for clipping window
GLfloat xwMin = -40.0, ywMin = -60.0, xwMax = 40.0, ywMax = 60.0;

// positions for near and far clipping planes
GLfloat dnear = 25.0, dfar = 125.0;

void init(void) {
    glClearColor(1.0, 1.0, 1.0, 0.0);

    // decides which matrix is to be affected by subsequent transform functions
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

    glMatrixMode(GL_PROJECTION);
    glFrustum(xwMin, xwMax, ywMin, ywMax, dnear, dfar);
}
```

# An example of OpenGL program (2)

```
void displayFcn(void) {
    glClear(GL_COLOR_BUFFER_BIT);

    // parameters for a square fill area
    glColor3f(1.0, 0.0, 0.0);
    glPolygonMode(GL_FRONT, GL_FILL);
    glPolygonMode(GL_BACK, GL_LINE);

    // square in the x-y plane
    glBegin(GL_QUADS);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(100.0, 0.0, 0.0);
    glVertex3f(100.0, 100.0, 0.0);
    glVertex3f(0.0, 100.0, 0.0);
    glEnd();

    glFlush();
}

void reshapeFcn(GLint newWidth, GLint newHeight) {
    glViewport(0, 0, newWidth, newHeight); // Set viewport to window dimensions
}
```

# An example of OpenGL program (3)

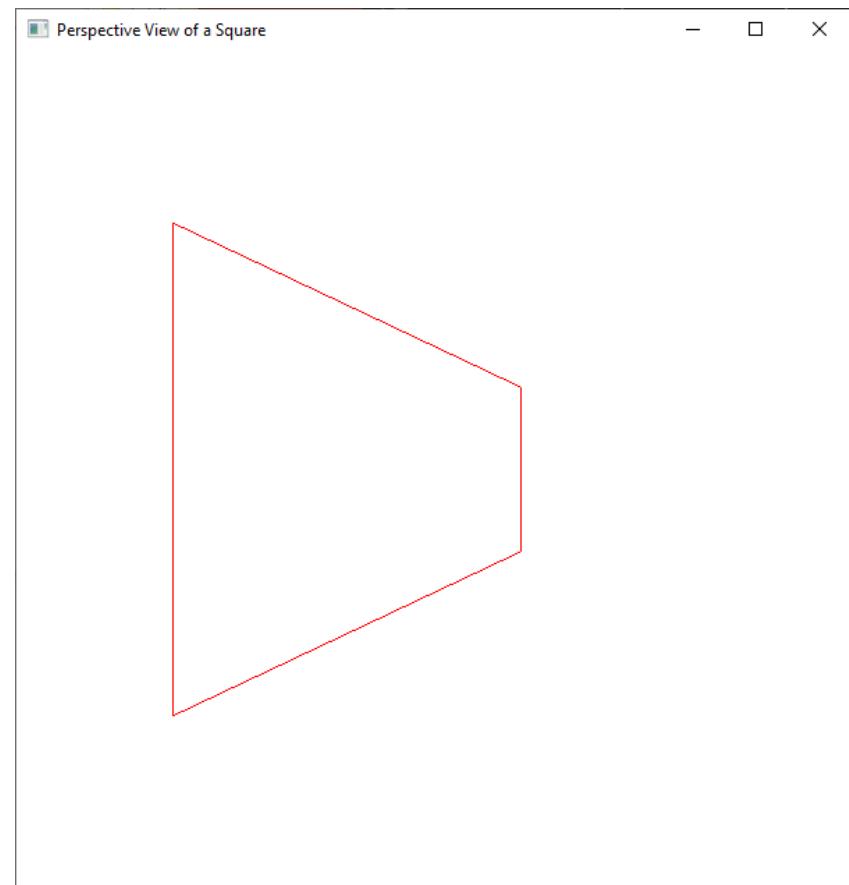
```
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(50, 50);
    glutInitWindowSize(winWidth, winHeight);
    glutCreateWindow("Perspective View of a Square");

    init();
    glutDisplayFunc(displayFcn);
    glutReshapeFunc(reshapeFcn);
    glutMainLoop();
}
```

# An example of OpenGL program (4)

- This program displays a perspective view of a square, which is defined in the x-y plane.
- A viewing co-ordinate origin is selected to view the front face at an angle. The perspective view is obtained using the `glFrustum()` function with the centre of the square as the look-at point.
- If the viewing origin is moved around to the other side of the square (e.g. `z0` is initialised to -50), the back face will be displayed in wireframe (since the back face is constructed without filling in the area).

# An example of OpenGL program (5)



# Exercise

- Create an object of your choice using OpenGL geometric primitive functions;
- Display the object in the following views
  - Isometric
  - Side
  - Top
- Add more effects by applying continuous viewing and projection transformations (similar to last week's lab but not geometric transformations)



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Parametric Curves and Surfaces**

**Week 06**  
**2021-22**

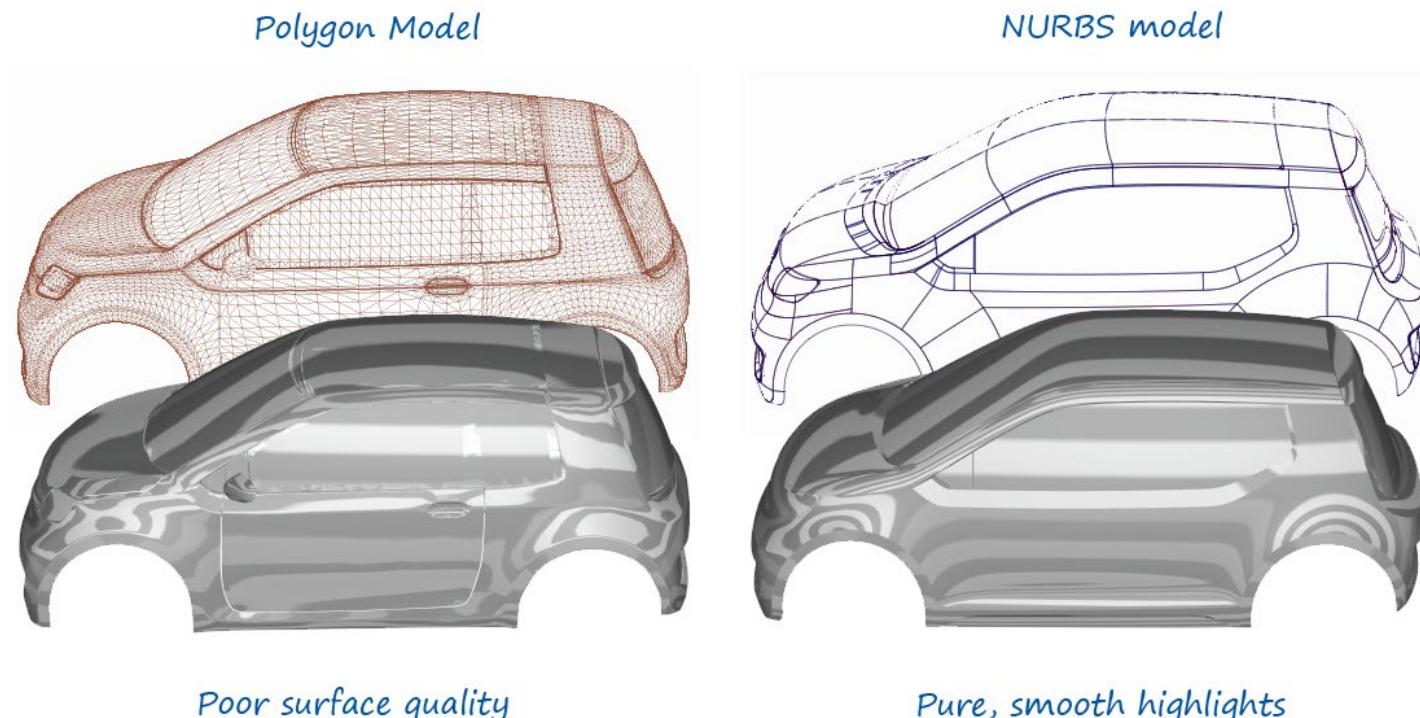
**Yong Yue**

# Topics for today

- Why parametric
- Parametric curves
- Splines
- Revolved, extruded and swept surfaces
- Tensor product surfaces
- Summary

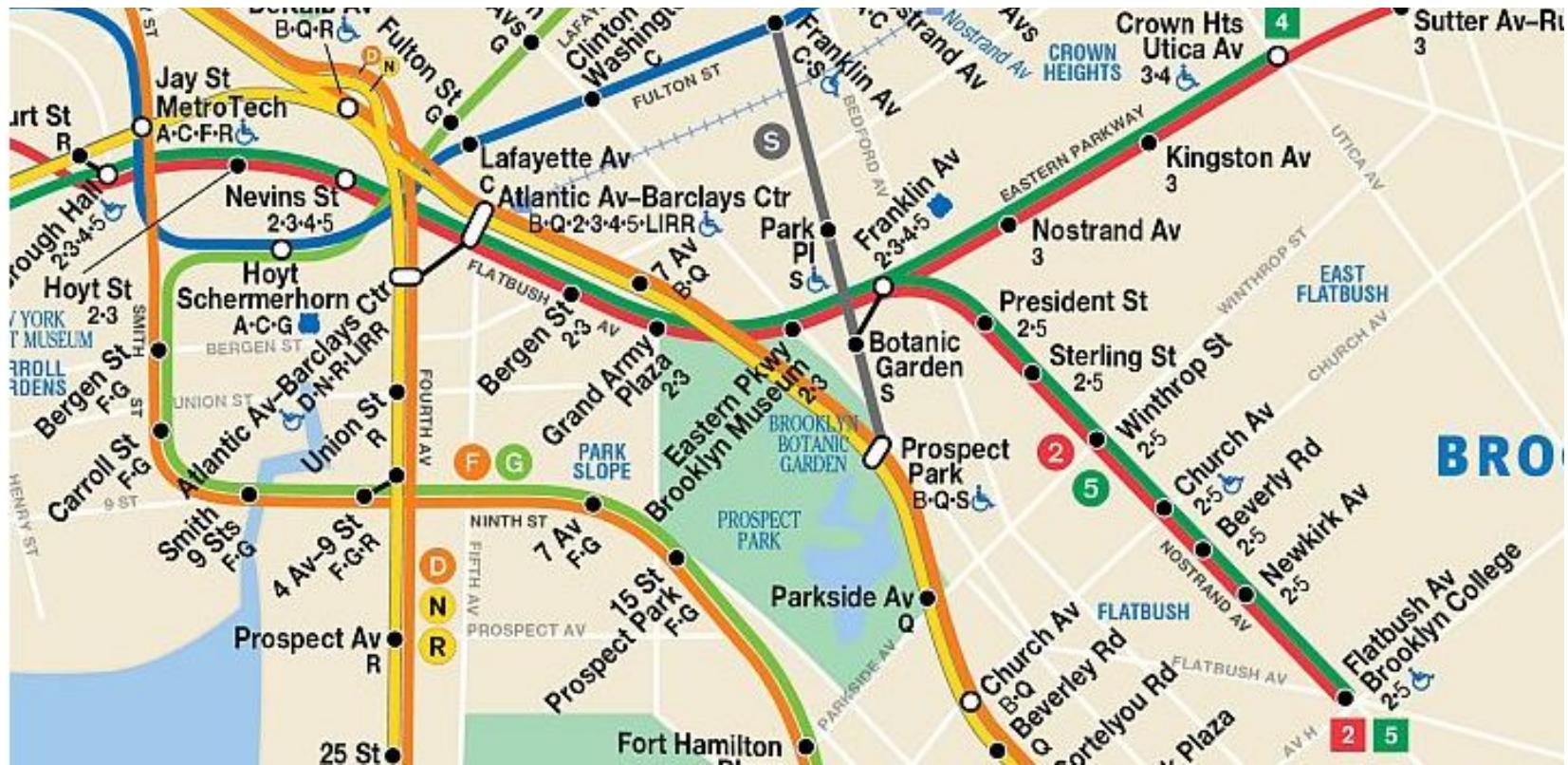
# Motivation

- More realistic 3D rendering of naturally curved objects



# Motivation

- Easier to follow than poly-lines



# Why parametric?

- Parametric surfaces are surfaces that are usually parameterised by two independent variables.
- By parameterisation, it is relatively easy to represent surfaces that are self-intersecting or non-orientable.
- It is impossible to represent many of these surfaces by using implicit functions.
- Even where implicit functions exist for these surfaces, the tessellated representation is often incorrect.

# Parametric curves

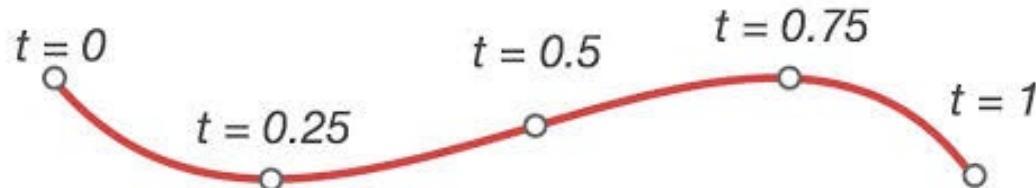
- A curve in a 2D  $(x, y)$  surface is defined as:

$$x = x(t)$$

$$y = y(t)$$

where  $t$  is a parameter in  $[0, 1]$ .

- In this way the curve is well defined, each value of  $t$  in  $[0, 1]$  defining one and only one point.
- The curve description will not change when rotation occurs.



# Parametric equation of a straight line

Implicit representation:  $y = a_0 + a_1x$

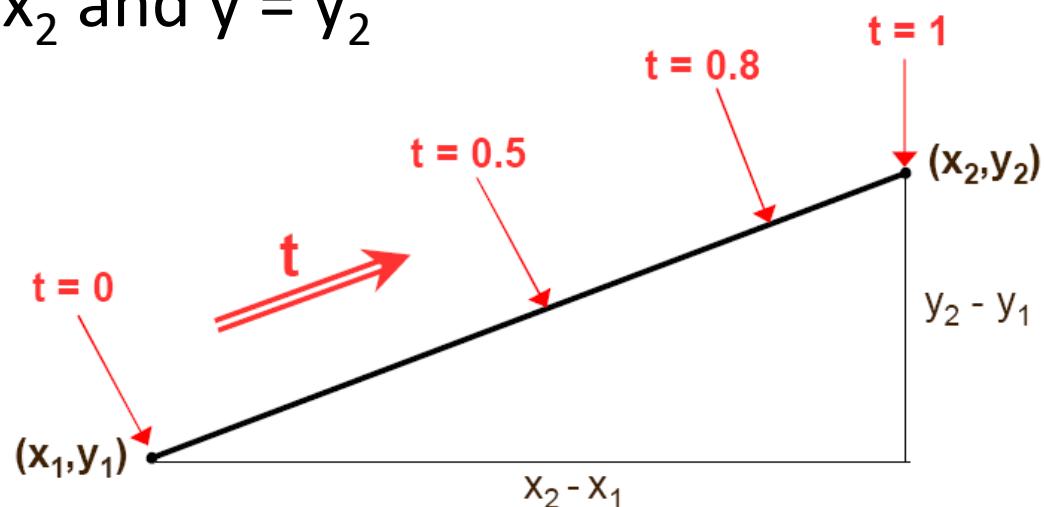
Parametric (explicit) representation:

$$x = x_1 + t(x_2 - x_1) \quad (0 \leq t \leq 1)$$

$$y = y_1 + t(y_2 - y_1)$$

when  $t = 0$ ,  $x = x_1$  and  $y = y_1$

when  $t = 1$ ,  $x = x_2$  and  $y = y_2$



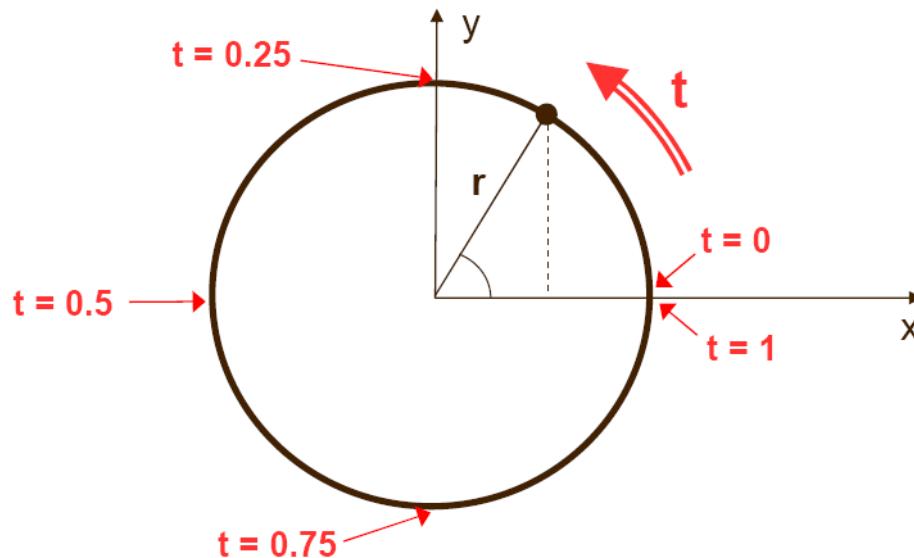
# Parametric equation of a circle

Implicit representation:

$$x^2 + y^2 = r^2 \quad (r = \text{radius})$$

Parametric equation:

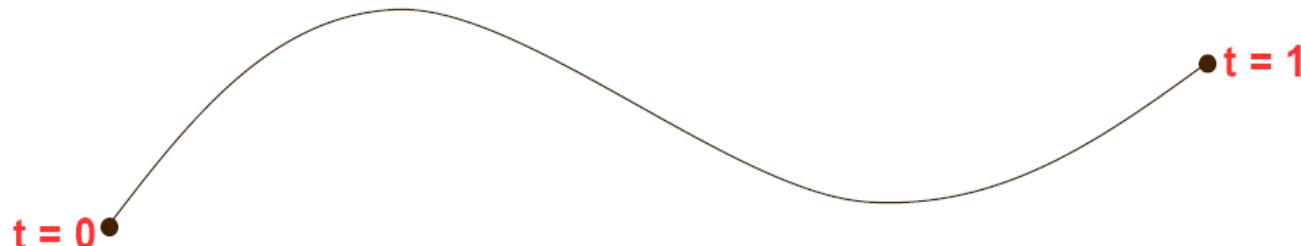
$$x = r \cos(360t), \quad y = r \sin(360t), \quad (0 \leq t \leq 1)$$



# Parametric equation of a cubic curve

$$\begin{aligned}x(t) &= a_0 + a_1 t + a_2 t^2 + a_3 t^3 & (0 \leq t \leq 1) \\y(t) &= b_0 + b_1 t + b_2 t^2 + b_3 t^3\end{aligned}$$

where  $a_i$  and  $b_i$  terms are constants that vary from curve to curve.

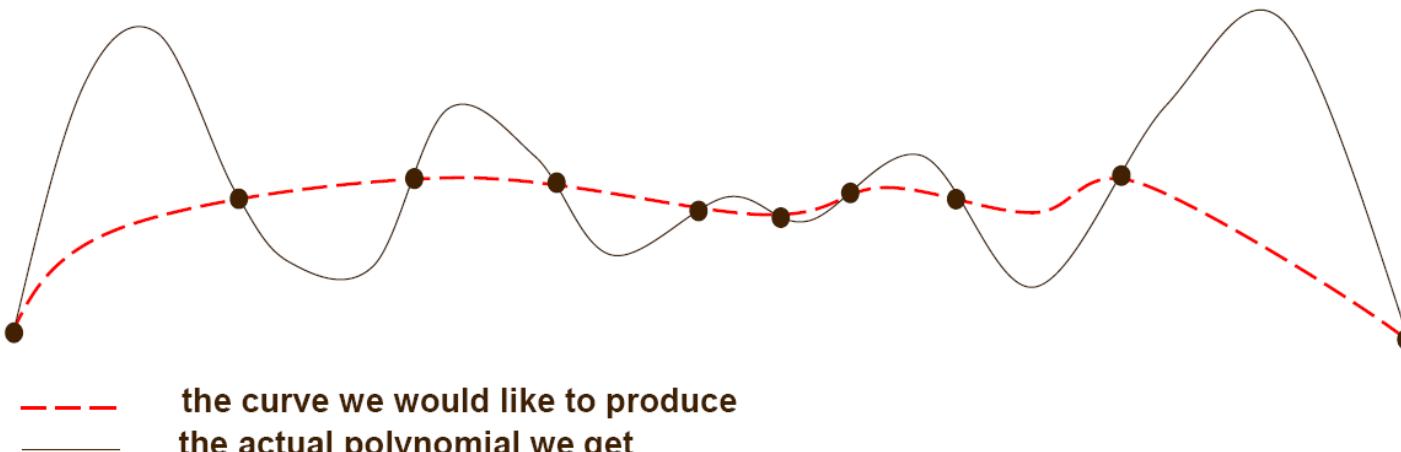


# What type of curve to use?

- A curve description should be used, which allows rapid computation (i.e. functions such as sin, cos, exp, log, and so on should be avoided).
- A polynomial can therefore be used
$$x(t) = a_0 + a_1t + a_2t^2 + a_3t^3, + \dots + a_nt^n$$
- For interpolation, if there are k points, then  $n = k - 1$  must be chosen, in order to find the correct values for  $a_i$ .

# Interpolation through $k$ points

- When  $k = 2$ , i.e.  $n = k - 1 = 1$ ; a straight line can be fitted.
- When  $k = 3$ , i.e.  $n = k - 1 = 2$ ; a parabola can be fitted.
- When  $k$  is large,  $n$  must be large, too; high-degree polynomials (i.e. with a large  $n$ ) oscillate wildly, particularly near the ends of the line, and are not suitable.



# Low-degree polynomial curves

- Polynomials have to be used for efficiency.
- High-degree polynomials are not suitable because of their behaviour.
- For curves of a large number of points
  - they can be broken into small sets (e.g. 4 points in each set).
  - a low-degree polynomial is put through each set (a cubic for 4 points).
  - these individual curves (cubics) are joined up smoothly.
- This is the basis of splines.

# Splines

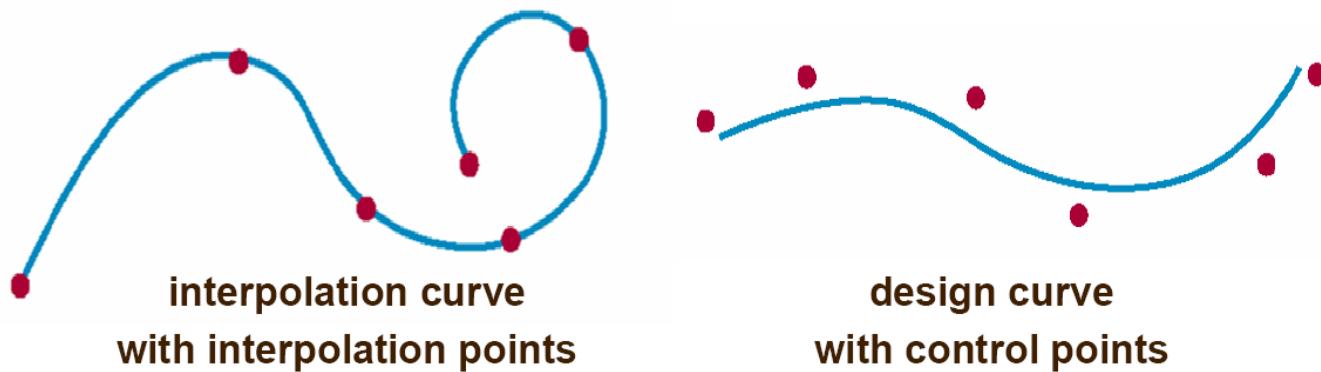
- A spline curve consists of individual components, joined together smoothly, looking like a continuous smooth curve.
- Different types of continuity exist and the following are generally required
  - continuity of the curve (no breaks)
  - continuity of tangent (no sharp corners)
  - continuity of curvature (not essential but avoids some artefact from lighting)
- Each component is a low-degree polynomial, and for these continuities, cubic polynomials are generally needed.

# Interpolation and design curves (1)

- An interpolation curve defines the exact position (point) that the curve must pass through, e.g. in a keyframe animation, an object must be at a particular point at a particular time.
- A design curve defines the general behaviour of the curve, e.g. what the curve should look like, and tuning the shape is often needed. The method is often used by designers.

# Interpolation and design curves (2)

- The shape of the interpolation curve depends on the data points provided.
- The shape of the design curve depends on the control points, which do not lie on the curve, but allow adjustment of the shape by moving the points.



# Design curves and local control

- The same approach is used in design curves.
- Each consists of separate, but joined parts.
- An important feature is local control.
- When a curve is designed, if one part is done, it would be preferred to keep its current shape when another part of the curve is adjusted.
- So the adjustment should influence only a small / local part of the curve – this is local control.

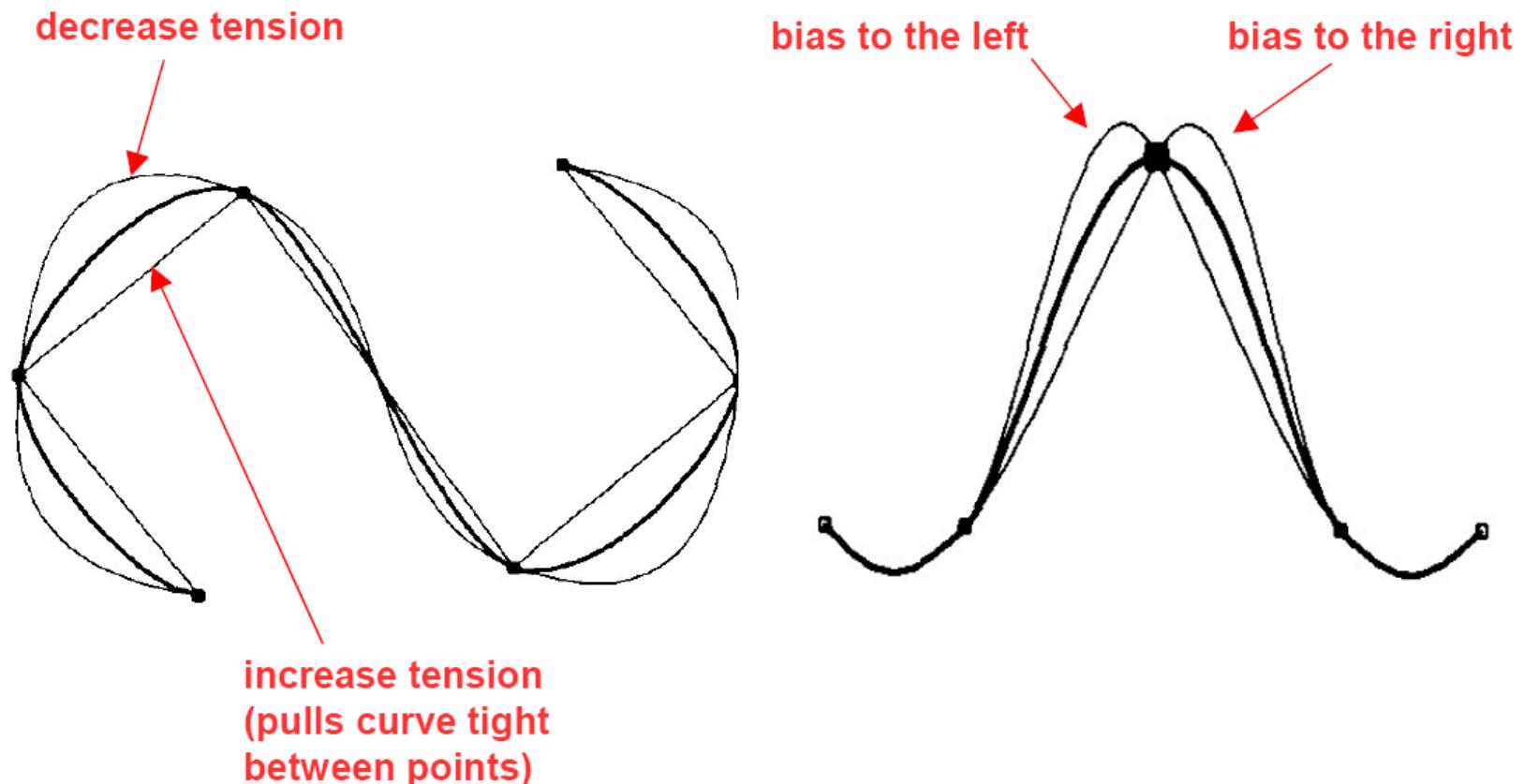
# Local control

- Curves without local control
  - Natural splines
  - Bezier curves (if continuity enforced)
- Curves with local control
  - B-Splines
  - NURBS (Non-Uniform Rational B-Splines)
- A cubic curve with local control
  - Normally influenced by only 4 control points
  - which are the control points most local to it

# Forms of local control

- So far we have considered controlling the design curve by only moving the control points.
- Some types of curve provide further parameters to allow some control while keeping the control points fixed – important ones include tension and bias.
- Such control can apply to both interpolation and design curves.

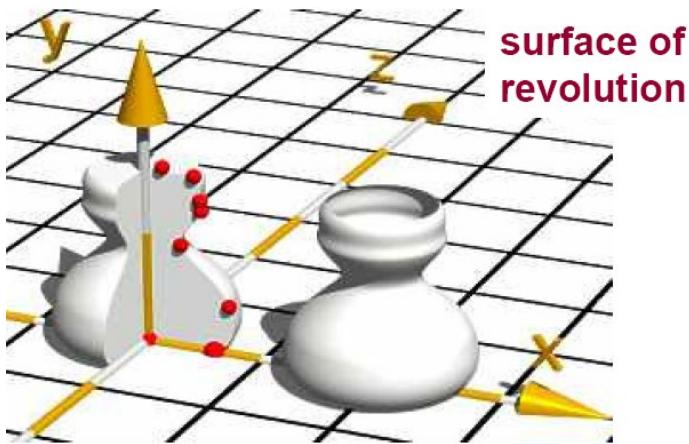
# Tension and bias



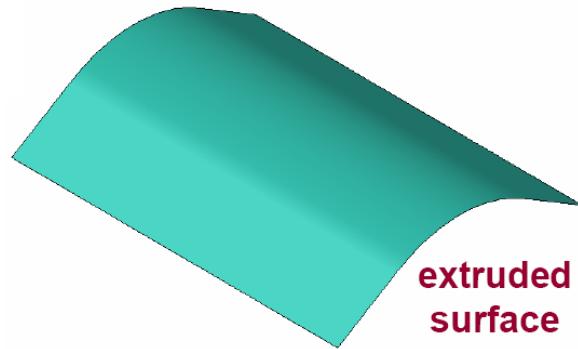
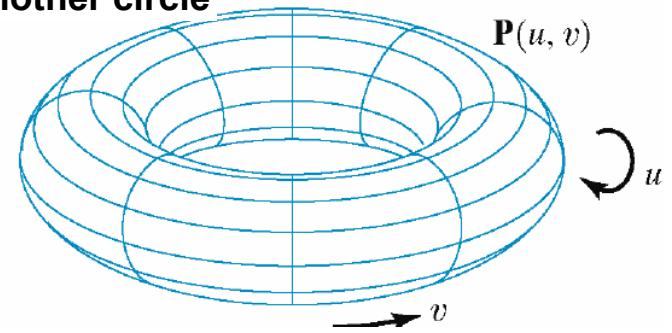
# Types of parametric surface

- Revolved surface – a 2D curve is revolved around an axis, and the parameter is the rotation angle.
- Extruded surface – a 2D curve is moved perpendicular to its own plane, and the parameter is the straight-line depth.
- Swept surface – a 2D curve is passed along a 3D path (which can be a curve), and the parameter is the path definition.

# Examples of surface

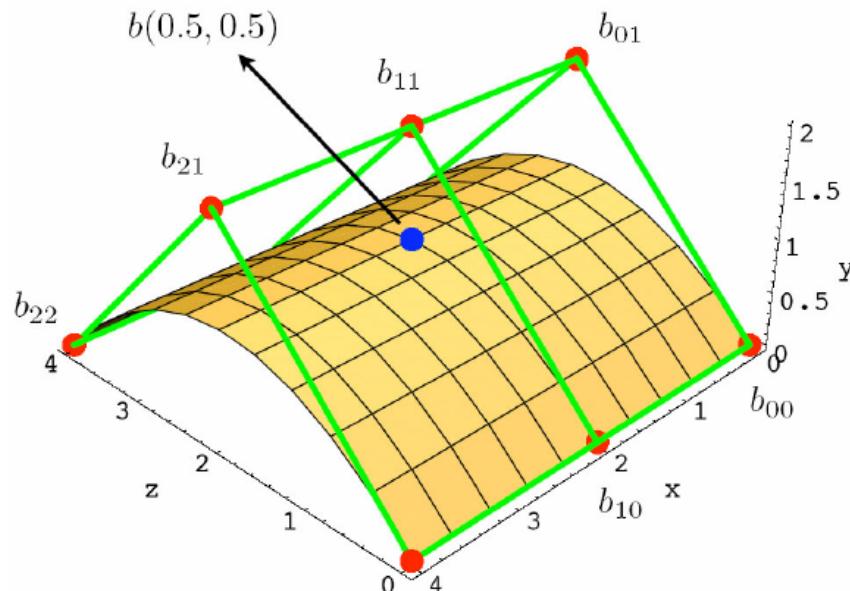


sweep surface produced by sweeping a circle along another circle



# Tensor product surfaces

- They are the most widely used parametric surfaces.
- A tensor product surface combines two parametric curves (curves of curves such as the examples in the previous slide). Essentially these work in perpendicular directions.
- Parametric curves depend on 1 parameter ( $t$ ) while parametric surfaces depend on 2 parameters ( $u, v$ ).

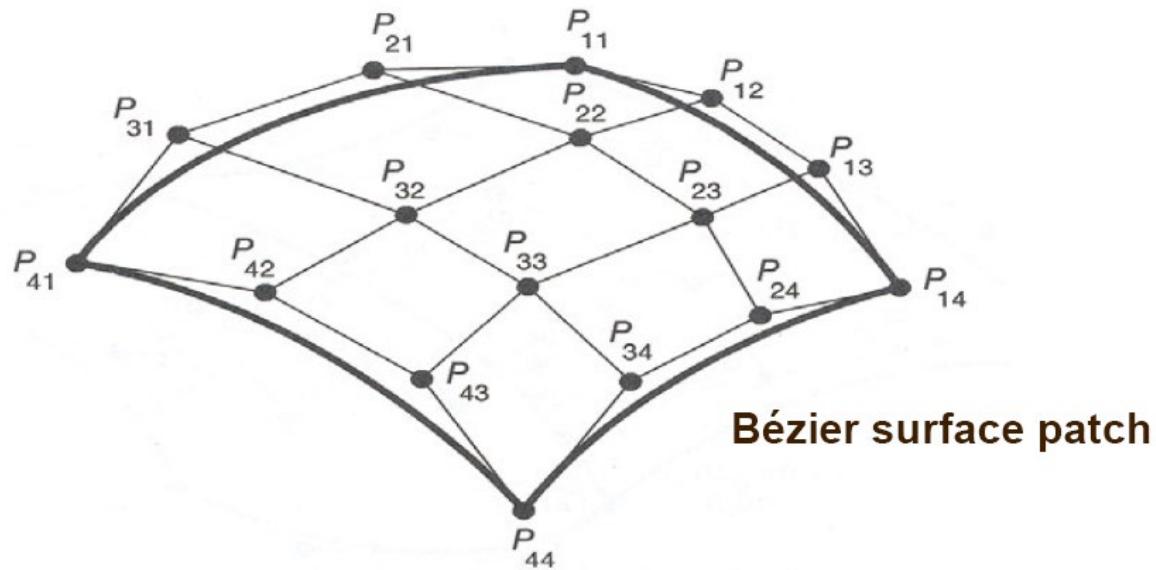


# Interpolation and design surfaces

- As for curves, there are interpolation and design surfaces, and the design form is more common.
- There is a control grid, normally a rectangular array of control points.
- As the curve is broken into smaller curves, the surface is broken into surface patches.
- Local control becomes even more important.

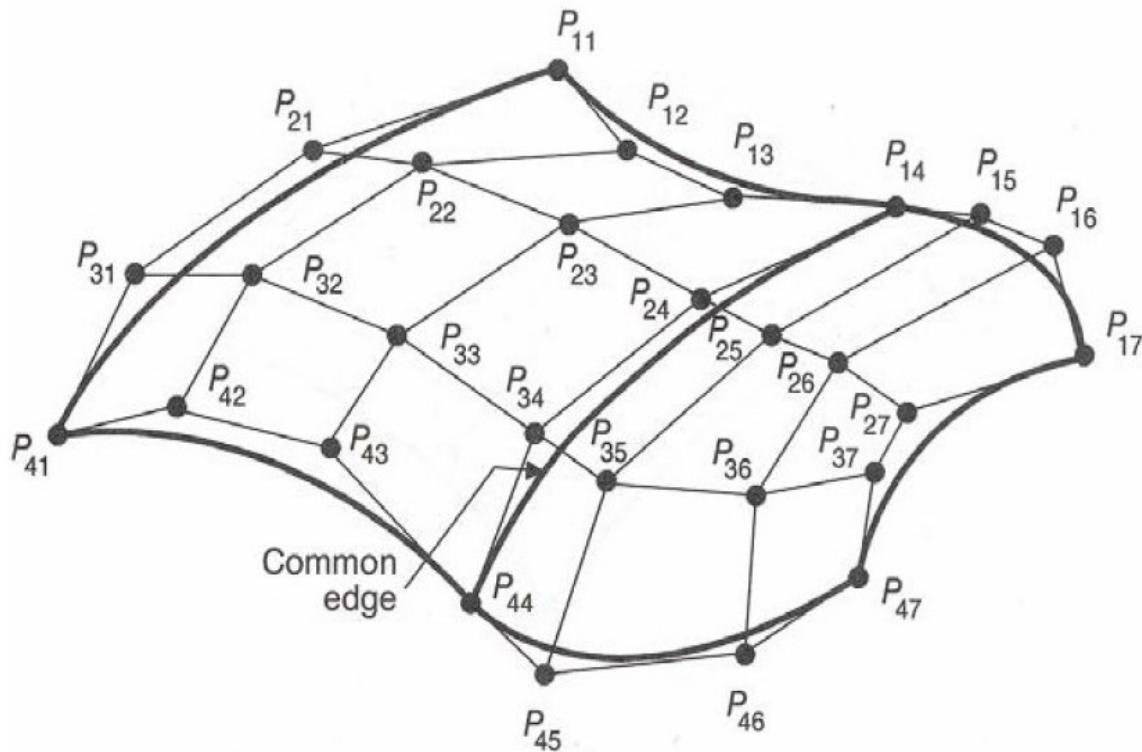
# Control grid for a surface patch

- In a cubic curve with local control, a curve segment is normally affected by only 4 control points.
- In a cubic surface with local control, a surface patch is affected by 16 control points, these being in a 4x4 grid.



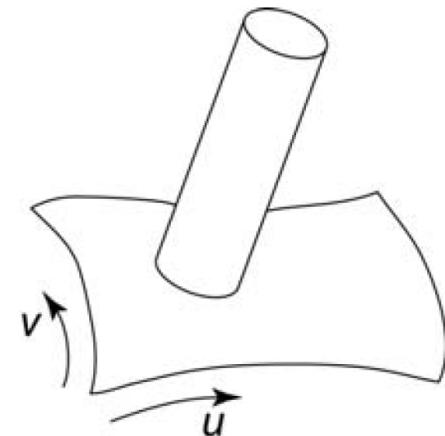
# Joining patches

- The patches are joined, which is not always straightforward.
- Appropriate continuity at the boundaries must be ensured.



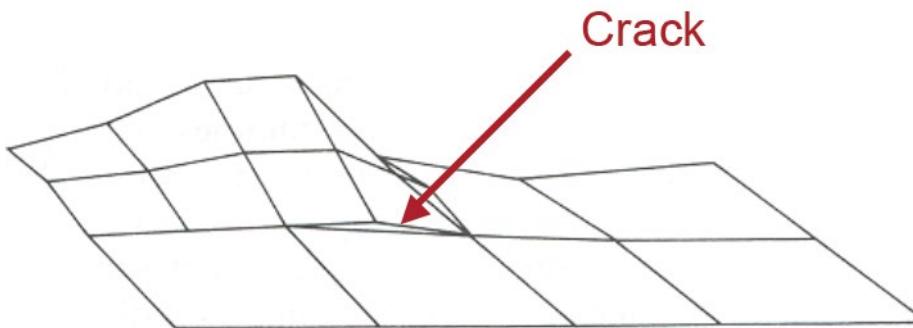
# Trimming and control

- The control of the surface is performed as for curves
  - the points in the control grid can be moved.
  - some types of surface have tension and other parameters to use without having to move grid points.
  
- If not all of the surface is to be displayed
  - a trim line cutting the surface can be defined.
  - all parts of the surface on one side of this line are removed.



# Improving resolution

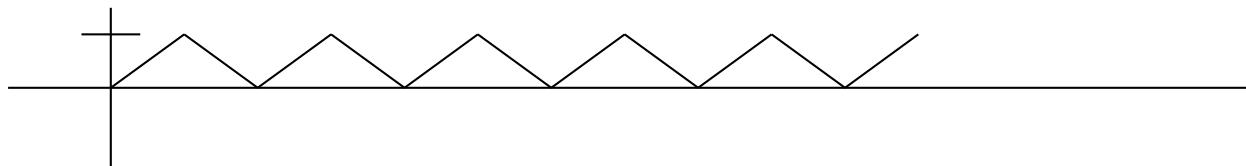
- To obtain finer detail, the number of patches can be increased.
- It is possible to do this adaptively, i.e. only increase the number of patches where extra detail is needed.
- This can lead to cracks in the surface unless care is taken.



# Linear interpolation

Linear interpolation is useful (e.g. for animation) where  $t$  varies from 0 to 1 (and sometimes back again) over time.

```
float t = 0;  
float dt = 0.01;  
void onIdle(void)  
{  
    t = t+dt;  
    if (t>1) {t = 1; dt = -0.01;}  
    if (t<0) {t = 0; dt = 0.01;}  
}
```



# Parametric functions

```
// Draw a sinewave

int i;
float x, y;
float d = 100.0;

glBegin(GL_POINTS);
for(i=0; i<=360; i=i+5)
{
    x = (float)i;
    y = d*sin(i*(3.1416/180.0));
    glVertex2f(x,y);
}
glEnd();
```

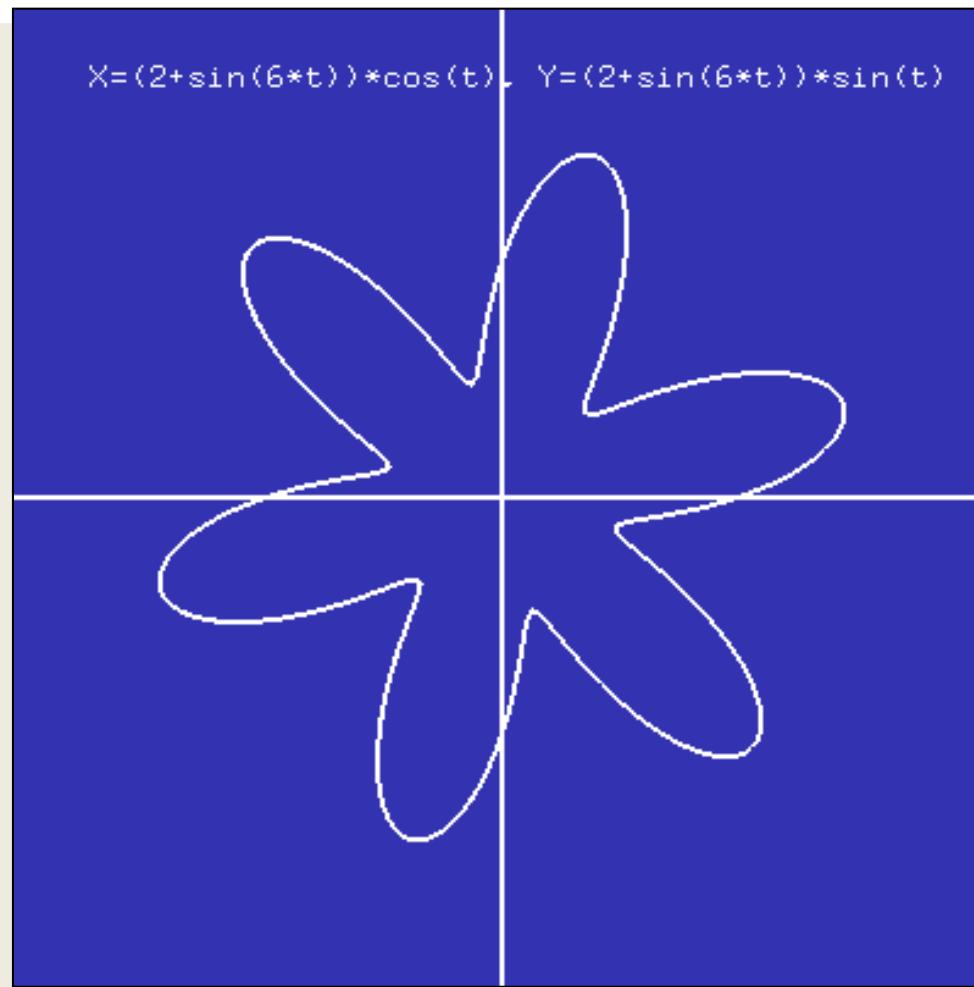
# Parametric functions

```
// Draw a circle

double x, y;
double t;
double r = 100;

glBegin(GL_LINE_STRIP);
for(t=0; t<=360; t+=1)
{
    x = r*cos(t*3.1416/180);
    y = r*sin(t*3.1416/180);
    glVertex3f(x,y,0);
}
glEnd();
```

# Parametric functions



# Summary

- Parametric (i.e. tensor product) surfaces provide a flexible modelling tool.
- The number of patches required for a model is far fewer than that of polygons for a similar model.
- Some modelling systems are based on such surfaces (NURBS being the most popular).
- Using such models produces an additional computational load on rendering the image (hidden-surface removal, shading calculations, collision detection and so on).



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **3D Modelling**

**Week 08**  
**2021-22**

**Yong Yue**

# Topics for today

## ➤ **3D modelling techniques**

- Wireframe
- Surface
- Solid

## ➤ **Constructive solid geometry (CSG)**

- CSG tree
- Characteristics and drawbacks

## ➤ **Boundary representation (B-Rep)**

- Geometry and topology
- Types of B-Rep models – manifold and nonmanifold
- Validity of B-Rep models – Euler's law
- Implementation of B-Rep models

# Wireframe modelling (1)

- It is the oldest and simplest approach.
- The model consists of a finite set of points and curves.
- Parametric representation of a space curve  
$$x = x(t), \quad y = y(t), \quad z = z(t)$$
- Implicit representation of a space curve  
$$s1(x,y,z) = 0, \quad s2(x,y,z) = 0$$

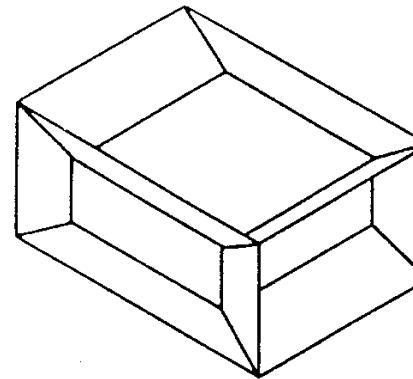
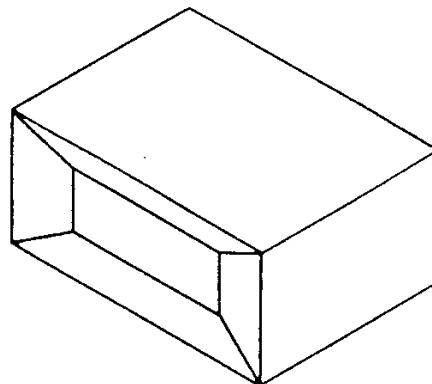
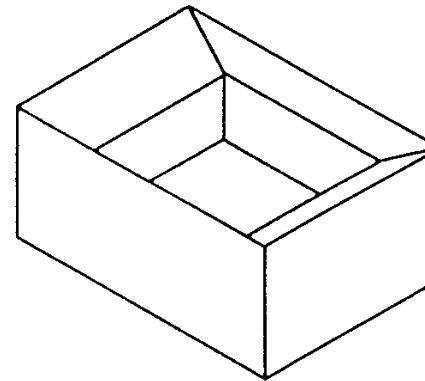
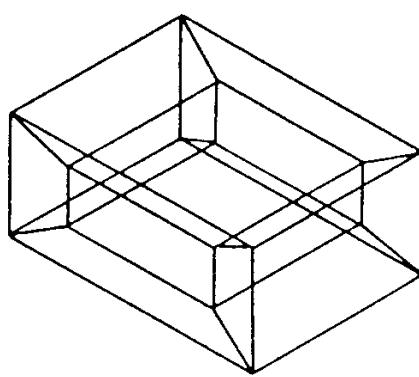
# Wireframe modelling (2)

	Parametric representation	Implicit representation
Straight line	$x = t + 1$ $y = 2t + 1$	$y = 2x - 1$
Circle	$x = r\cos\theta + a$ $y = r\sin\theta + b$	$(x - a)^2 + (y - b)^2 = r^2$

# Wireframe modelling (3)

- Combined use of curves can represent 3D objects in the space.
- Its disadvantages are the ambiguity of the model and the severe difficulty in validating the model.
- Furthermore, it does not provide surface and volume-related information.

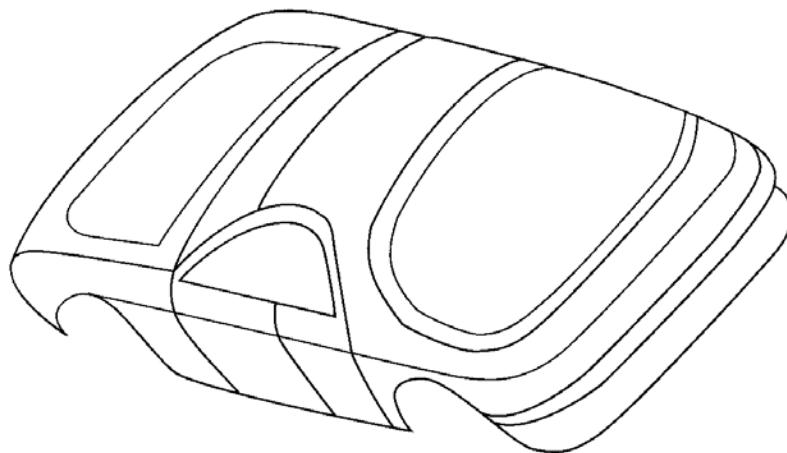
# Wireframe modelling (4)



Ambiguous wireframe models

# Surface modelling (1)

- It generates objects with a more complete and less ambiguous representation than its wireframe model.
- It is obvious that surface models are suitable for more applications, for example, design and representation of car bodies.

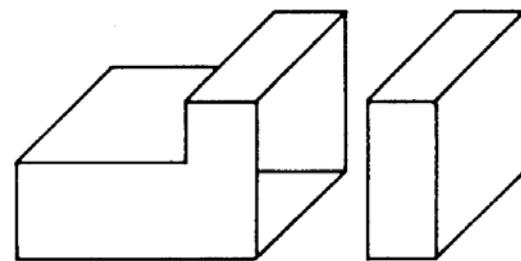
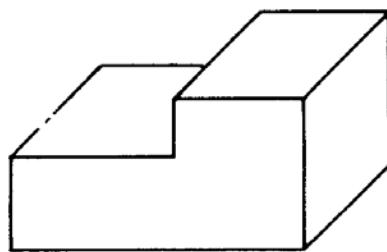


# Surface modelling (2)

- Surfaces can be 2D and 3D represented by a closed loop of curves with skin on it, the simplest form being a plane. Surfaces are built from points and curves.
- They are very important in modelling objects, and in many situations, used to represent 3D models to a large variety of satisfaction.
- Modelling packages usually provide a range of useful surface creation functions, some of which are similar to those for curves (but the geometric characteristics are different).

# Surface modelling (3)

- Despite their similar appearance, there are differences between surface and solid models.
- Apart from the lack of volume-related information, surface models normally define the geometry of their corresponding objects.



Surface model - hollow model:  
volume, mass, centroid?

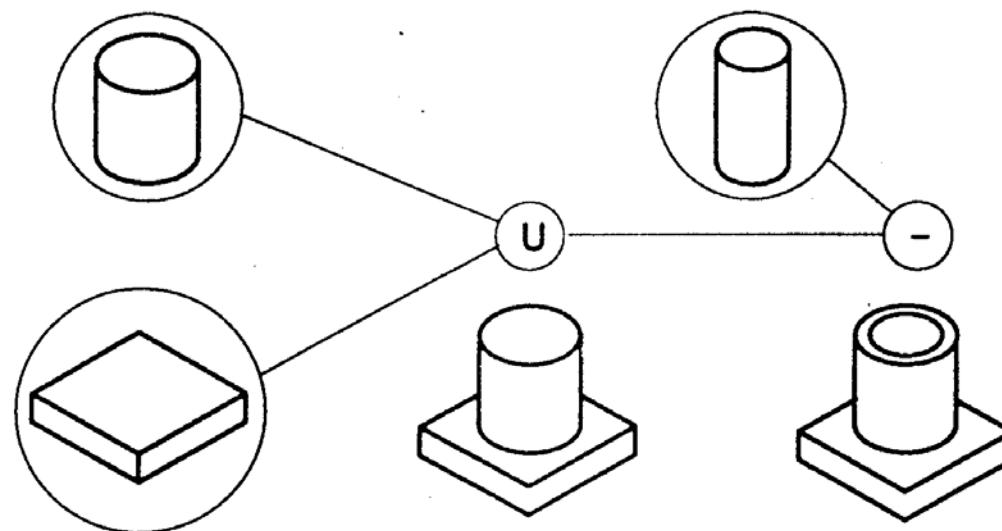
# Solid modelling

- Solid modelling represents both the geometric properties (e.g. points, curves, surfaces, volume, centre of shape) and physical properties (e.g. mass, centre of gravity and inertia) of solid objects.
- There is a number of schemes, namely primitive instancing, cell decomposition, constructive solid geometry (CSG) and boundary representation (B-Rep).
- CSG and B-Rep are the most popular.

# Constructive solid geometry (1)

- The CSG model is an ordered binary tree where the non-terminal nodes represent the operators and the terminal nodes are the primitives or transformation leaves.
- The operators may be rigid motions or regular Boolean operations.
- The primitive leaf is a primitive solid in the modelling space while the transformation leaf defines the arguments of rigid motion.

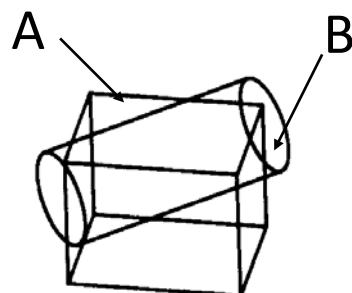
# Constructive solid geometry (2)



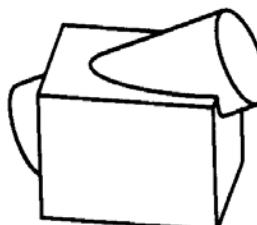
CSG tree

# Constructive solid geometry (3)

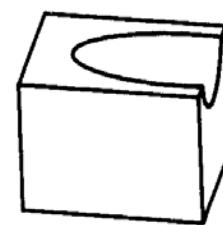
- Boolean operations include Boolean Union, Boolean Difference and Boolean Intersection.
- It should be noticed that the resultant solid of a Boolean operation depends not only on the solids but also on their location and orientation.



Objects



Union  
 $A \cup B$



Subtraction  
 $A - B$



Intersection  
 $A \cap B$

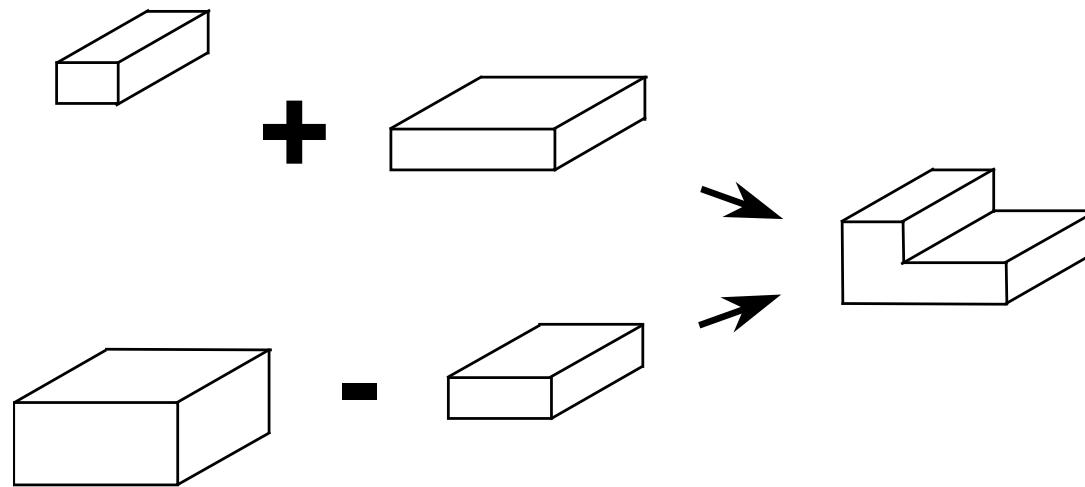
# Constructive solid geometry (4)

- Each solid usually has its local co-ordinate frame specified relative to the world co-ordinate frame.
- Before a Boolean operation is performed, it may be necessary to translate and/or rotate the solids in order to obtain the required relative location and orientation relationship between them.

# Constructive solid geometry (5)

- If an object can be represented by a unique set of data, the representation is said to be unique.
- The representation scheme for some applications (e.g. geometric reasoning) should ideally be unambiguous and unique.
- Solid representations are usually unambiguous but few of them can be unique, and it is not feasible to make CSG representation unique.

# Constructive solid geometry (6)

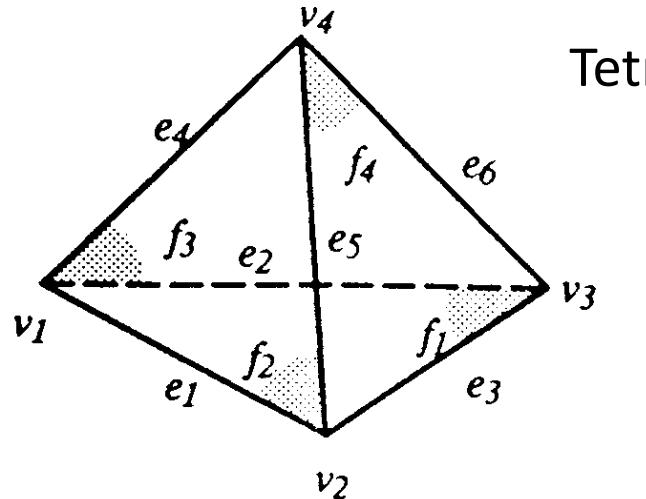


Nonuniqueness of CSG model

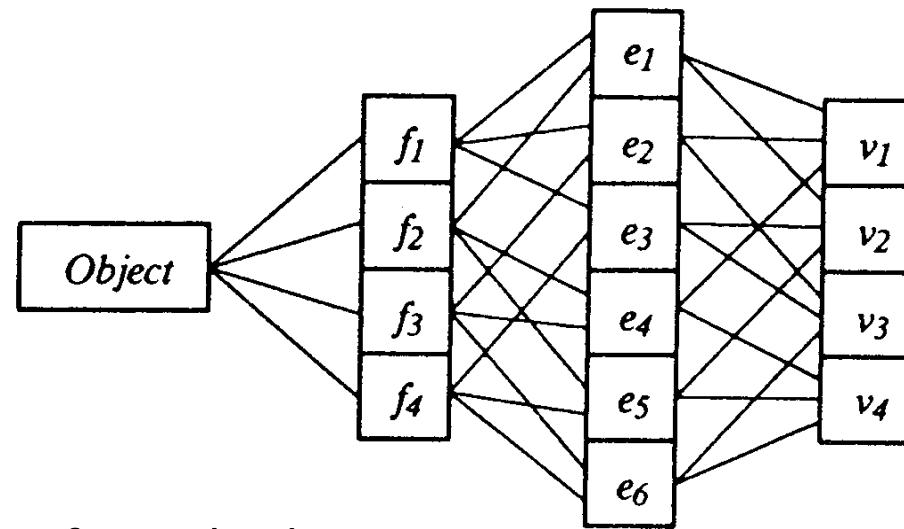
# Boundary representation (1)

- The boundary representation (B-Rep) model represents a solid by segmenting its boundary into a finite number of bounded subsets.
- It is basically a topologically explicit representation. Both geometric and topological information is stored in the data structure.
- The **geometry** is about the shape and size of the boundary entities called *points*, *curves* and *surfaces* while the topology keeps the **connectivity** of the boundary entities referred as *vertices*, *edges* and *faces* (corresponding to points, curves and surfaces).

# Boundary representation (2)



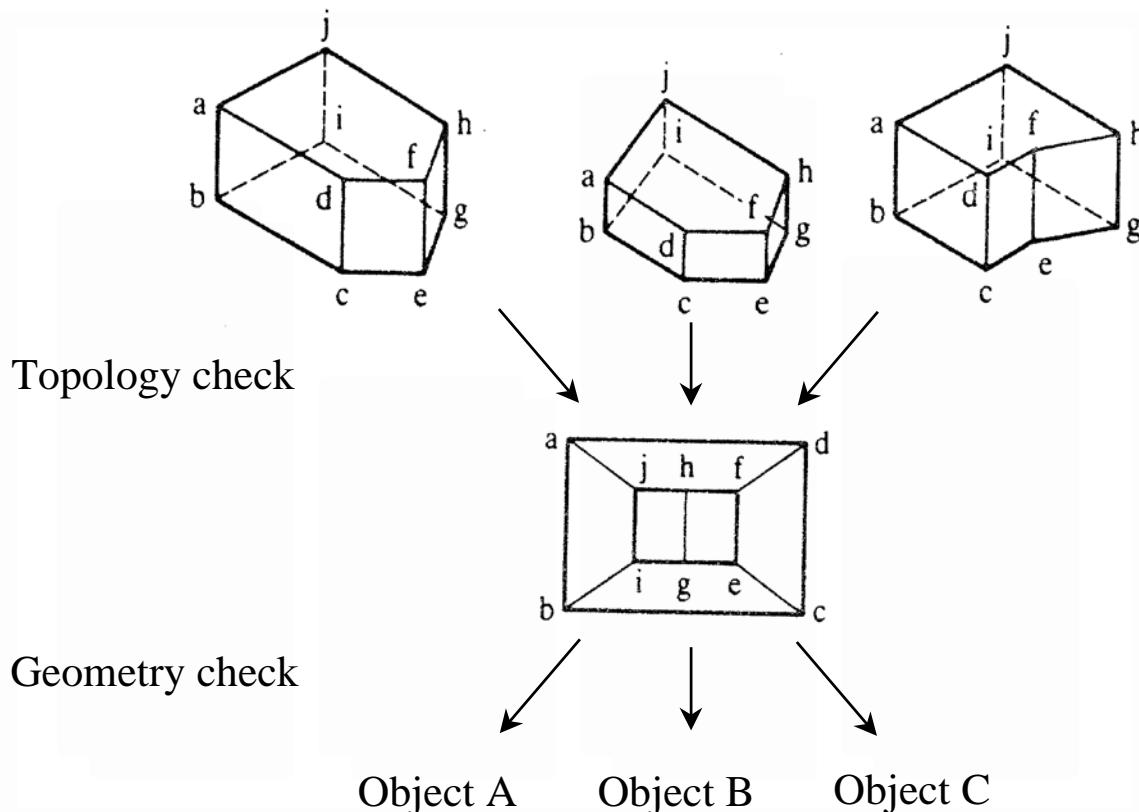
Tetrahedron



Topology of tetrahedron

# Boundary representation (3)

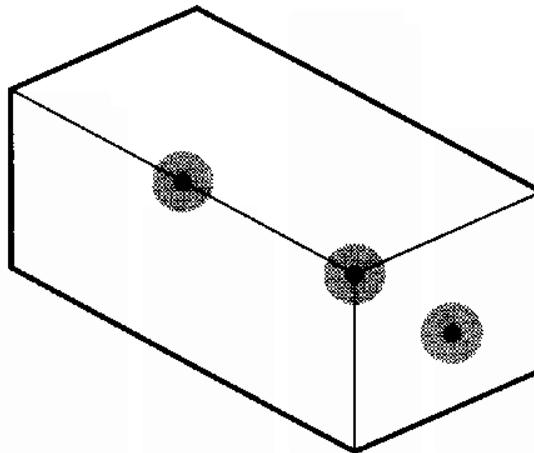
The same topology may represent different geometric shapes and therefore both topological and geometric data is necessary to fully and uniquely define an object.



# Types of B-Rep

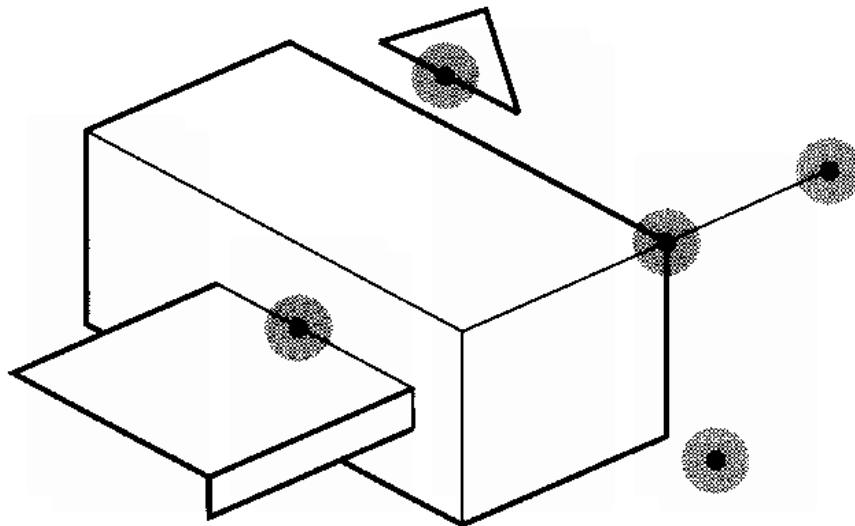
- B-Rep models can be divided into two types: *manifold* and *nonmanifold*.
- In a manifold model, an edge is connected exactly by two faces and at least three edges meet at a vertex.
- A nonmanifold model may have dangling faces, edges and vertices, and therefore represent a nonrealistic object.

# Manifold B-Rep model



Two faces meet exactly at one edge,  
and at least three edges meet at a vertex

# Nonmanifold B-Rep model



Dangling faces, edge and vertices

# Euler's law for manifold B-Rep (1)

To ensure the topological validity for a solid (i.e. manifold model), a manifold model must satisfy the following Euler (Leonhard Euler, 1707-1783) formula,

$$V - E + F - R + 2H = 2S$$

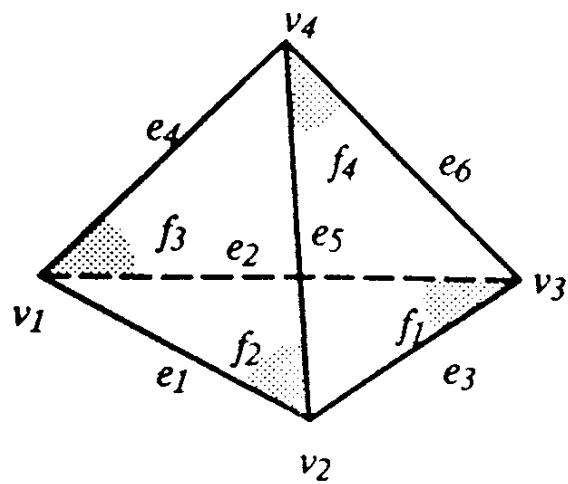
where  $V, E, F, R, H$  and  $S$  are the number of vertices, edges, faces, rings (inner loops on faces), passages/holes (genus) and shells (disjoint bodies), respectively.

# Euler's law for manifold B-Rep (2)

The Euler's law in its simplest form is

$$V - E + F = 2$$

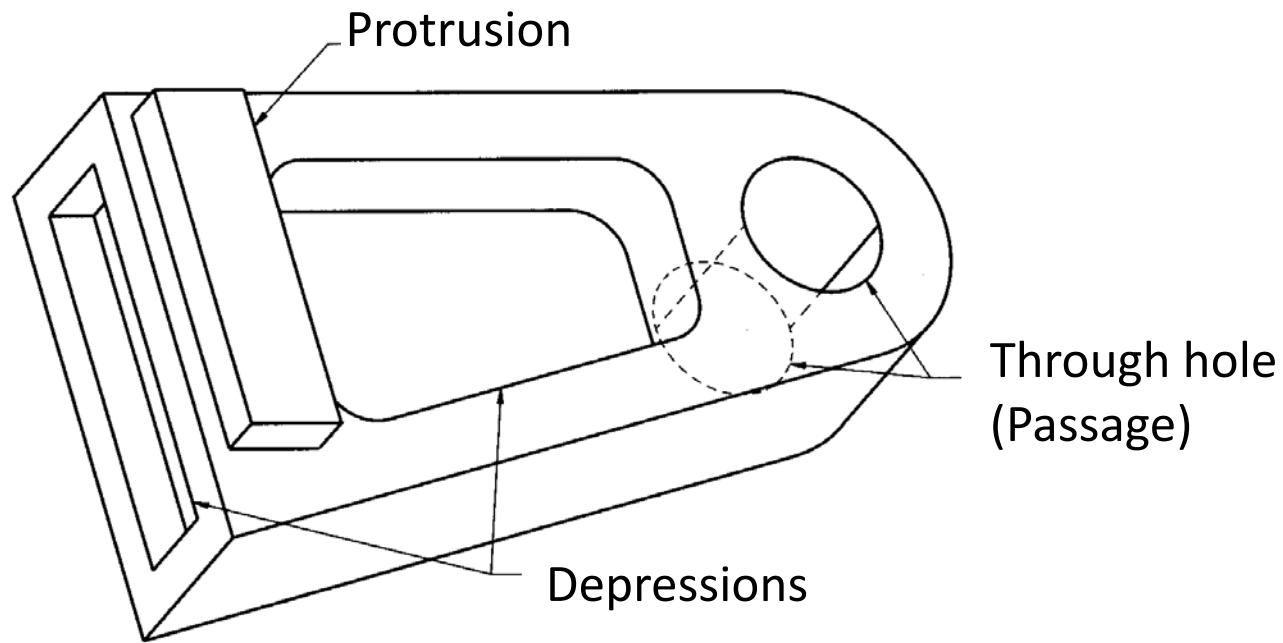
which can be applied to simple polyhedra (i.e. objects without inner loops of edges and passages).



For the tetrahedron

$$4 - 6 + 4 = 2$$

# Euler's law for manifold B-Rep (3)



# Euler's law for manifold B-Rep (4)

	V	E	F	R	H	S
Basic shape	8	12	6			1
Protrusion	8	12	5	1		
Sharp corner depression	8	12	5	1		
Round corner depression	16	24	9	1		
Hole	4	6	2	2	1	
Total	44	66	27	5	1	1

$$V - E + F - R + 2H = 2S$$

$$44 - 66 + 27 - 5 + 2 \times 1 = 2 \times 1$$

# Euler's law for manifold B-Rep (5)

- The cylindrical hole in the previous slide is represented by two edges along its axis, resulting in 4 vertices, 6 edges and 2 faces.
- Hence, a cylinder body can be represented by 4 vertices, 6 edges and 4 faces, satisfying the Euler's law.
- It is worth noting that the Euler's law also applies to curved objects represented by patches, curve segments and vertices.

# Implementation of B-Rep (1)

- B-Rep models can be conveniently implemented on computers by representing the topology as pointers and the geometry as numerical information in the data structure for extraction and manipulation using object-oriented programming techniques (e.g. C++).
- The latest B-Rep modellers also provide facilities to tag attributes (such as colour, tolerance and surface finish) on the boundary elements.

# Implementation of B-Rep (2)

Faces

Face	Edge
1	1
	2 •
	3
	4
2	.
	.
	.

Edges

Edge	Vertex
1	1
	2
• 2	5 •
	6
3	.
	.
	.

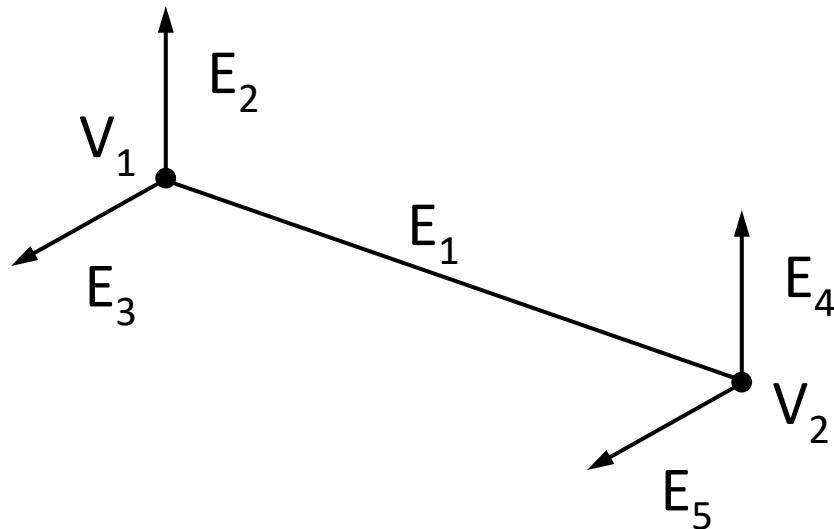
Vertices

Vertex	X	Y	Z
1	2	1	3
2	4	2	5
3	2	3	6
4	.	4	.
• 5	.	.	.
.	.	.	.
.	.	.	.

The three table-structure

# Implementation of B-Rep (3)

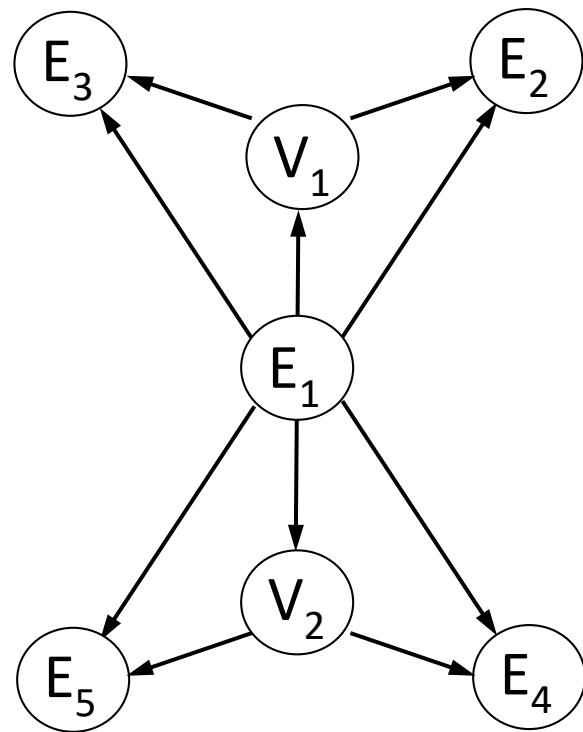
- The figure shows a single edge, ending in two vertices, which then each has two other edges leading off from them.
- The edge might, for example, be an edge of a cuboid. This is how the edge might be represented in the computer.



# Implementation of B-Rep (4)

- The edge has *pointers* to the vertices at its ends, and to the next edges. A pointer is essentially the address in the computer's memory where something is stored.
- The vertices have pointers to their coordinates ( $x, y, z$ ) and so on.
- This structure is called *Baugmart's winged edge* data structure named after its inventor. The 'winged edge' phrase refers to the edge and its adjoining faces, which look like a dihedral wing.

# Implementation of B-Rep (5)

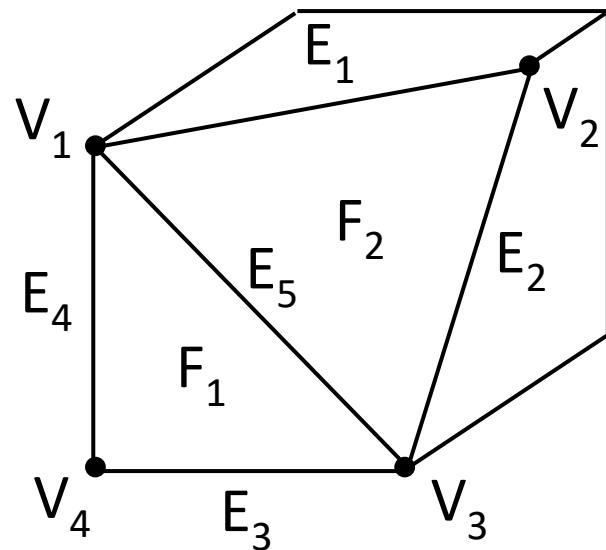
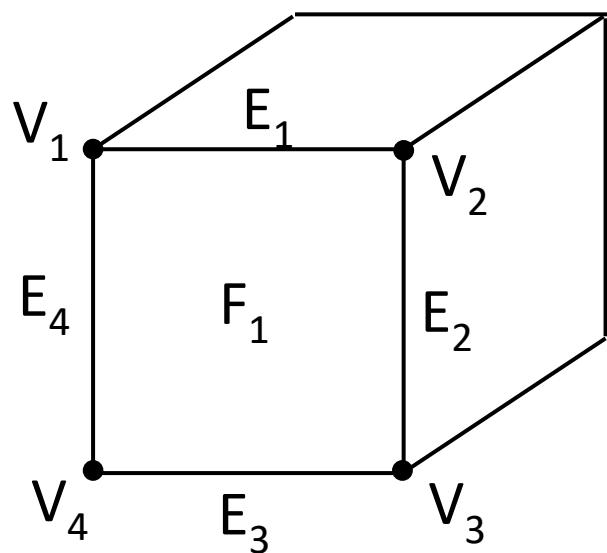


*Baugmart's winged edge data structure*

# Implementation of B-Rep (6)

- Many B-Rep modelling systems have procedures called Euler-operators.
- These modify the face-edge-vertex pointer structure in such a way that the Euler formula is always kept true.
- An example is: *make\_edge\_and\_face(f1, v1, v3).*
  - This would take the cuboid on the left, and split the face into two along its diagonal, making a new object.
  - Note that in the original cuboid, v1, v2, v3 and v4 must all lie in the same plane, but in the new object v2 is free to move along the top-right edge.

# Implementation of B-Rep (7)



# Summary

## ➤ **3D modelling techniques**

- Wireframe
- Surface
- Solid

## ➤ **Constructive solid geometry (CSG)**

- CSG tree
- Characteristics and drawbacks

## ➤ **Boundary representation (B-Rep)**

- Geometry and topology
- Types of B-Rep models – manifold and nonmanifold
- Validity of B-Rep models – Euler's law
- Implementation of B-Rep models



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Hierarchical Modelling**

**Week 9**  
**2021-22**

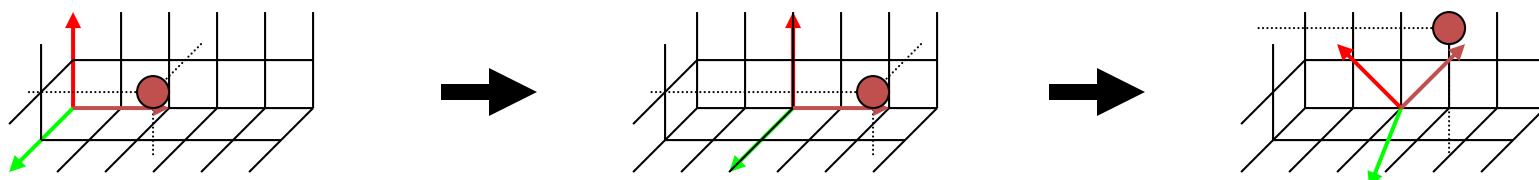
**Yong Yue**

# Topics for today

- Local and world co-ordinate frames of reference
- Object transformations
- Linear modelling
  - Symbols
  - Instances
- Hierarchical modelling
  - Hierarchical trees
  - Articulated models
- Examples and code

# Local and world frames of reference (1)

- We are used to defining points in space as  $(x,y,z)$ . But what does that actually mean? Where is  $(0,0,0)$ ?
- The actual truth is that there is no  $(0,0,0)$  in the real world. Objects are always defined *relative* to each other.
- We can *move*  $(0,0,0)$  and thus move all the points defined relative to that origin.



# Local and world frames of reference (2)

- The following terms are used interchangeably
  - *Local basis*
  - *Local transformation*
  - *Local / model frame of reference*
- Each of these refers to the location, in the greater world, of the (0,0,0) we are working with
  - They also include the concept of the current *local frame*, which is about the x, y, z directions.
  - By rotating the *local frame* of a coordinate system, we can rotate the world it describes.

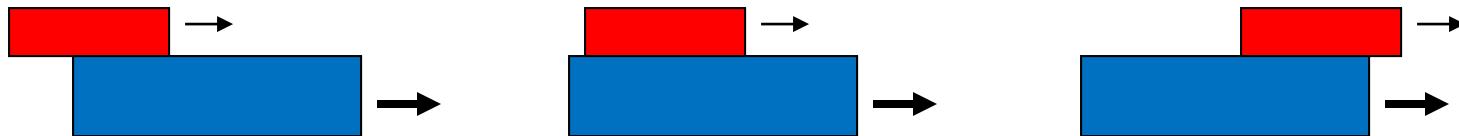
# What does the centre of the world mean?

- A *world frame of reference* is defined for a scene of objects.
- Each object has a *local frame of reference* which is relevant to the world frame.



# Relative motion

- Relative motion - a motion takes place relative to a local origin.

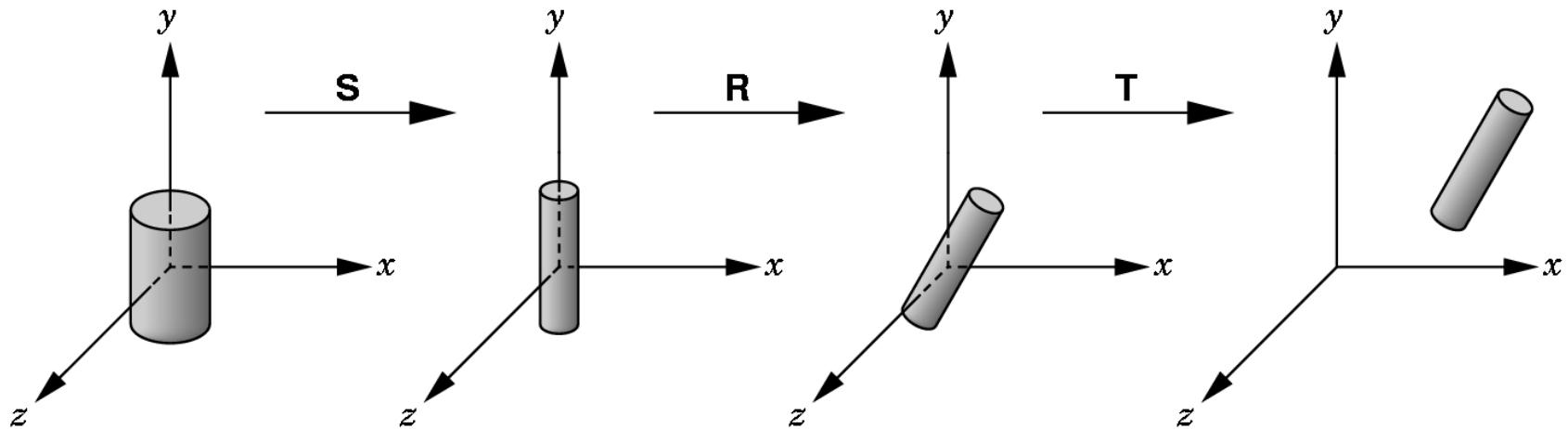


e.g. throwing a ball to a friend as you both ride in a train.

- The term *local origin* refers to the  $(0,0,0)$  that is chosen to measure the motion from.
- The local origin may be moving relative to some greater frame of reference.

# Linear modelling (1)

- Start with a *symbol* (prototype)
- Each appearance of the object in the scene is an *instance*
  - We must scale, orient and position it to define the instance transformation
  - $M = T \cdot R \cdot S$



# Linear modelling (2)

In OpenGL

- Set up appropriate transformations from the model frame (frame of symbols) to the world frame
- Apply it to the MODELVIEW matrix before executing the code

```
glMatrixMode(GL_MODELVIEW); // M = T·R·S  
glLoadIdentity();  
glTranslatef();  
glRotatef();  
glScalef();  
	glutSolidCylinder() // or other symbol
```

# Linear modelling (3)

Example: generating a cylinder

```
glBegin(GL_QUADS);
    For each A = Angles
    {
        glVertex3f(R*cos(A), R*sin(A), 0);
        glVertex3f(R*cos(A+DA), R*sin(A+DA), 0);
        glVertex3f(R*cos(A+DA), R*sin(A+DA), H);
        glVertex3f(R*cos(A), R*sin(A), H);
    }
glEnd();

// Make Polygons for Top/Bottom of cylinder
```

# Linear modelling (4)

- **Symbols (Primitives)**

Cone, Sphere, GeoSphere, Teapot, Box, Tube, Cylinder, Torus, etc.

- **Copy**

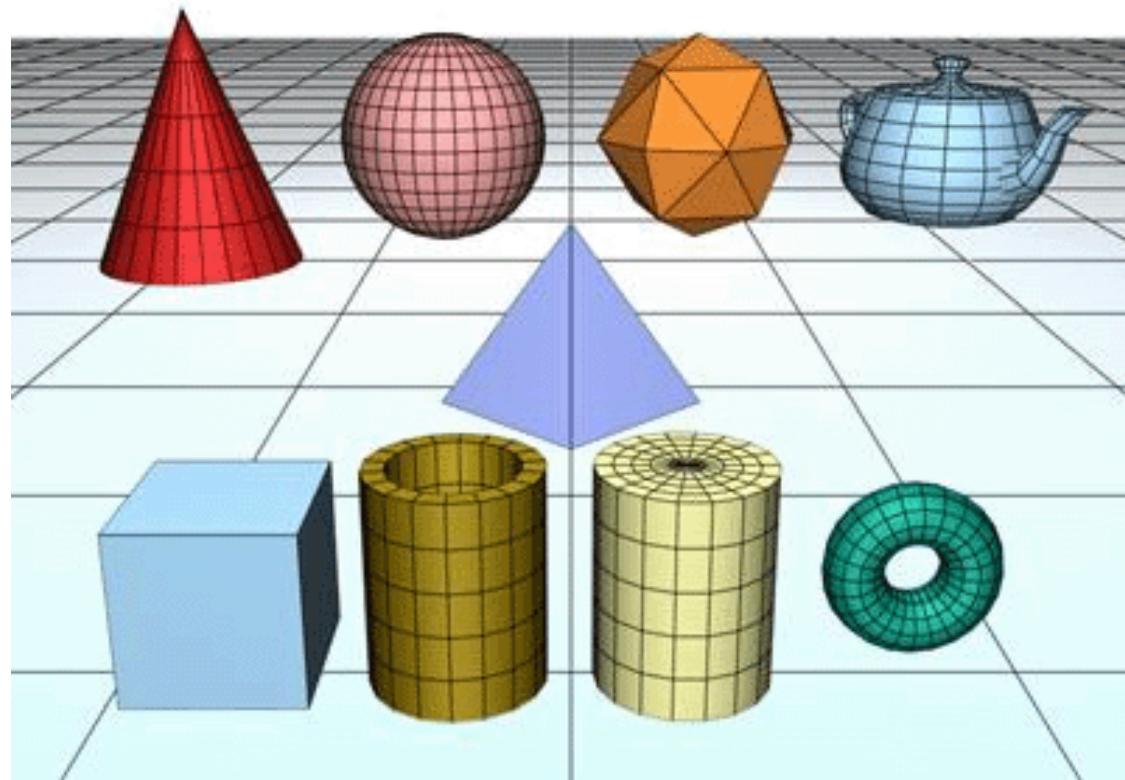
Creates a completely separate clone from the original.

Modifying one has no effect on the other.

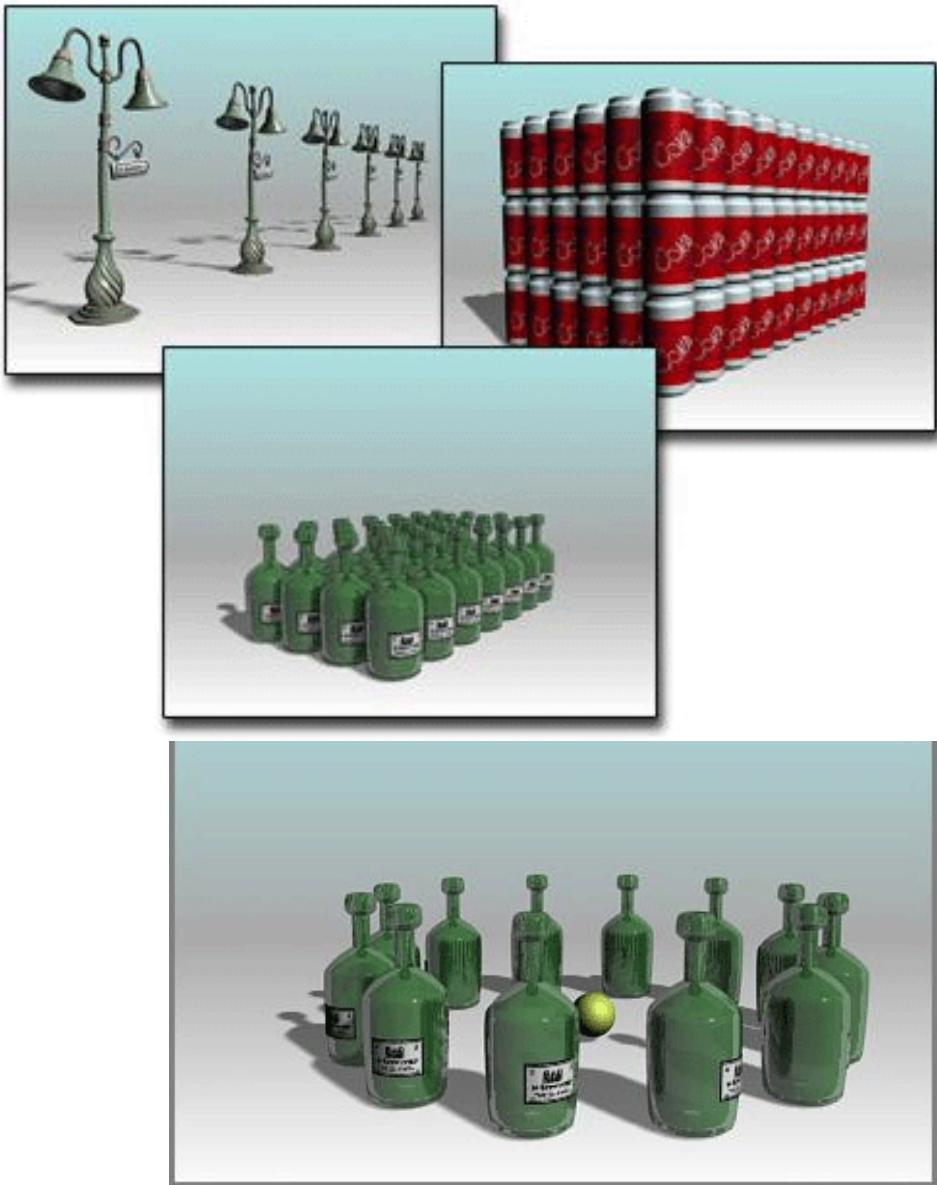
- **Instance**

Creates a completely interchangeable clone of the original.

Modifying an instanced object is the same as modifying the original.



# Linear modelling (5)



- Array: series of clones
  - Linear
    - Select object
    - Define axis
    - Define distance
    - Define number
  - Radial
    - Select object
    - Define axis
    - Define radius
    - Define number

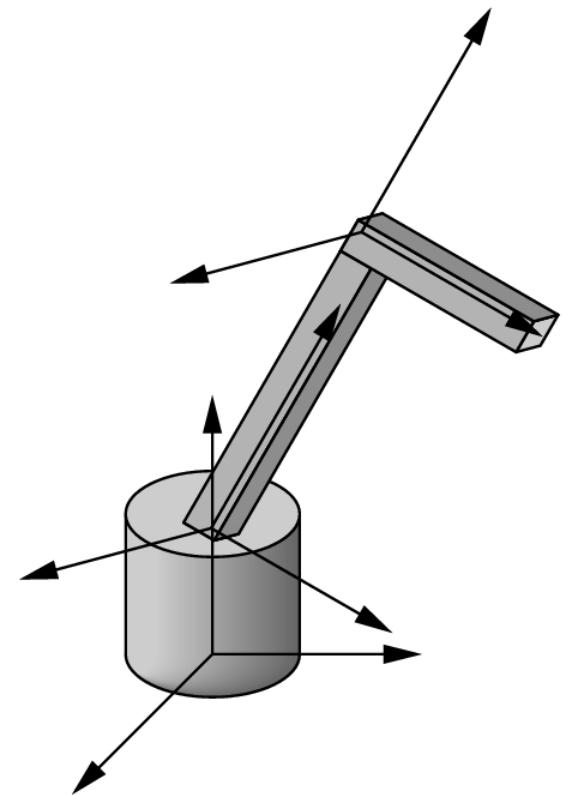
# Linear modelling (6)

- Model stored in a table by
  - assigning a number to each symbol and
  - storing the parameters for the instance transformation
- Contains flat information  
but no information on the actual structure
- How to represent complex structures with constraints?
- Each part has its own model frame of co-ordinate system  
but no information of relationships
- How to manipulate with substructures?

# Linear modelling (7)

Symbol	Scale	Rotate	Translate
1	$s_x, s_y, s_z$	$u_x, u_y, u_z$	$d_x, d_y, d_z$
2			
3			
1			
1			
.			
.			

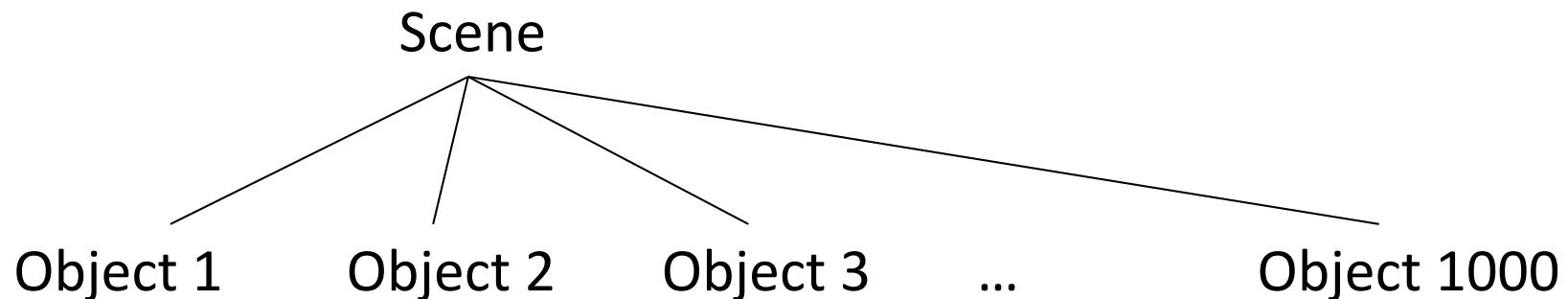
Linear model table



Model with constraints

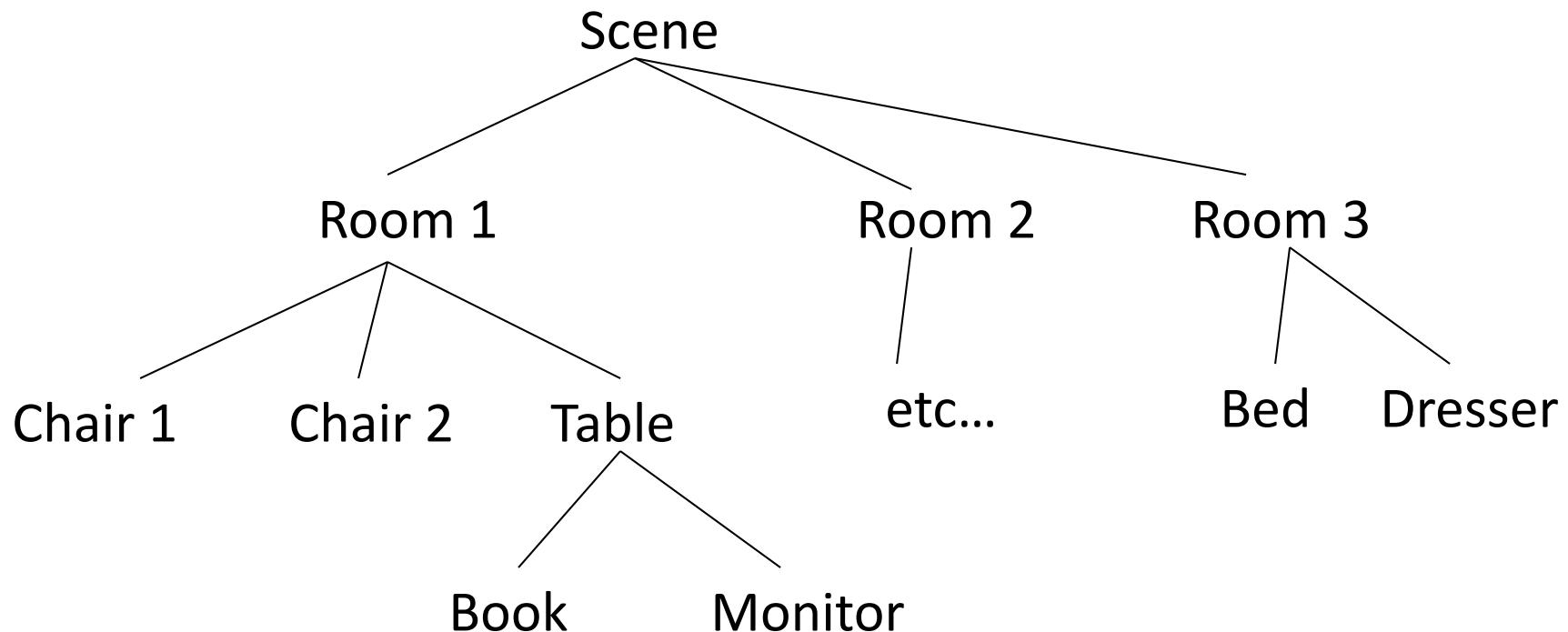
# Scene hierarchy (1)

If a scene contains 1000 objects, we might think of a simple organisation like this.



# Scene hierarchy (2)

We could also have a hierarchical grouping like this.

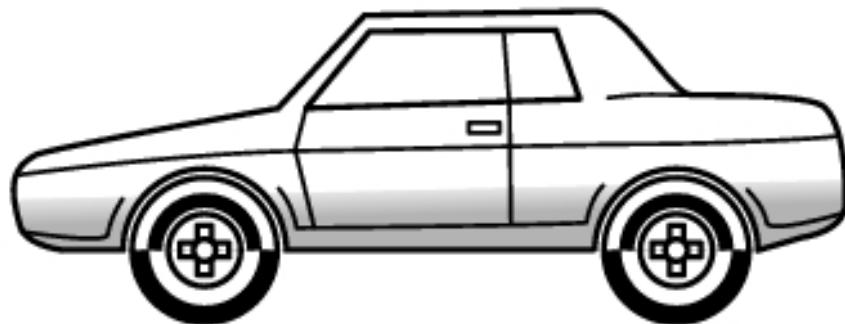


# Scene hierarchy (3)

- In a scene, some objects may be grouped together in some way. For example, an *articulated* figure may contain several rigid components connected together in a specified fashion.
  - several objects sitting on a tray that is being carried around
  - a bunch of moons and planets orbiting around in a solar system
  - a hotel with 200 rooms, each room containing a bed, table, chairs, etc.
- In each of these cases, the placement of objects is described more easily when we consider their locations relative to each other.

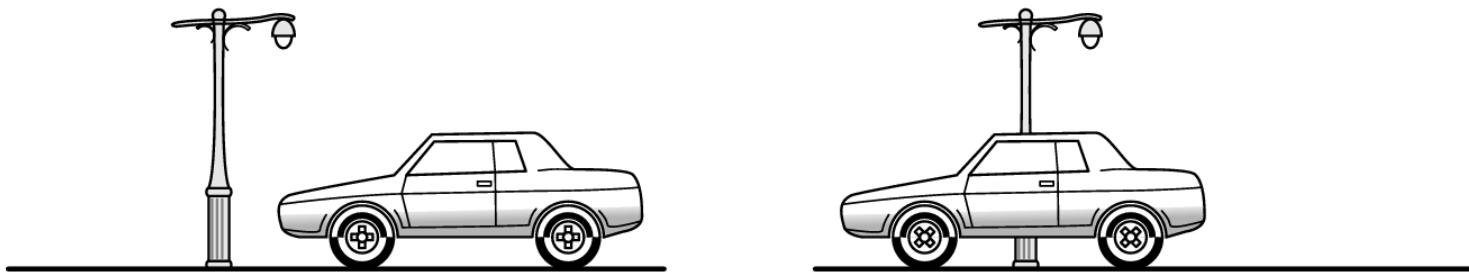
# Hierarchical models – a car (1)

- Consider the model of a car
  - Chassis + 4 identical wheels
  - Two symbols
- Speed of the car is actually determined by the rotational speed of wheels or vice versa.



# Hierarchical models – a car (2)

```
void main ( );
{   float s = ...;           // speed
    float d[3] = {...};      // direction
    draw_right_front_wheel(s,d);
    draw_left_front_wheel(s,d);
    draw_right_rear_wheel(s,d);
    draw_left_rear_wheel(s,d);
    draw_chassis(s,d);
} // WE DO NOT WANT THIS!
```



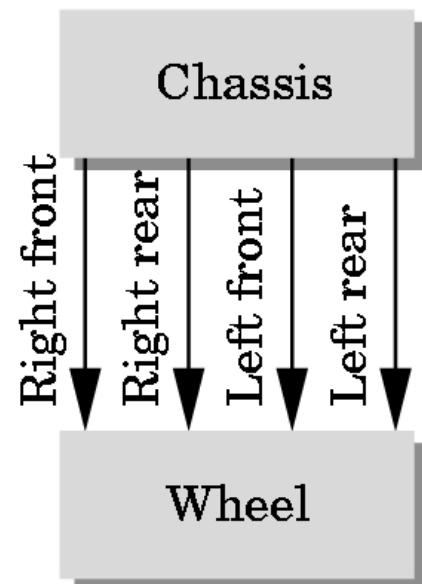
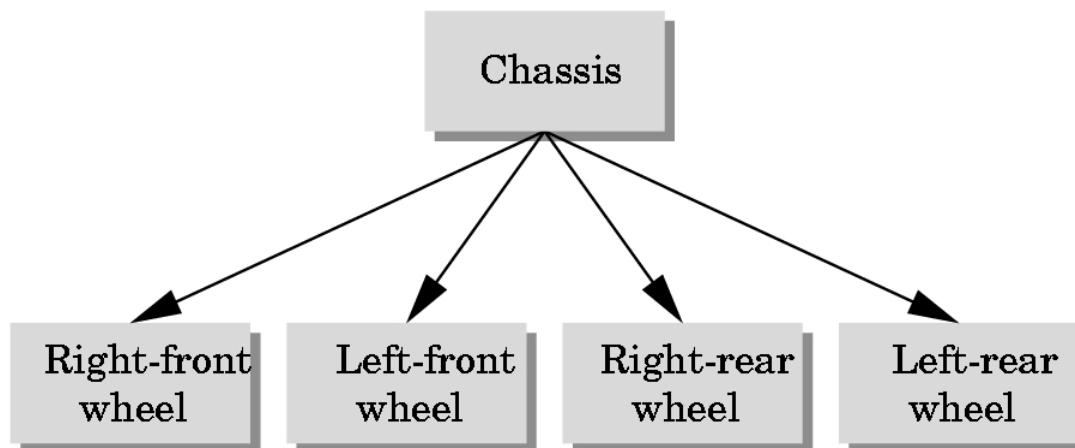
Two frames of reference for animation

# Hierarchical tree (1)

- It is very common in computer graphics to define a complex scene in some sort of hierarchical fashion.
- The individual objects are grouped into a hierarchy that is represented by a tree structure (upside down tree).
  - Each moving part is a single *node* in the tree.
  - The node at the top is the *root node*.
  - Each node (except the root) has exactly one *parent* node which is directly above it.
  - A node may have multiple *children* below it.
  - Nodes with the same parent are called *siblings*.
  - Nodes at the bottom of the tree with no children are called *leaf nodes*.

# Hierarchical tree (2)

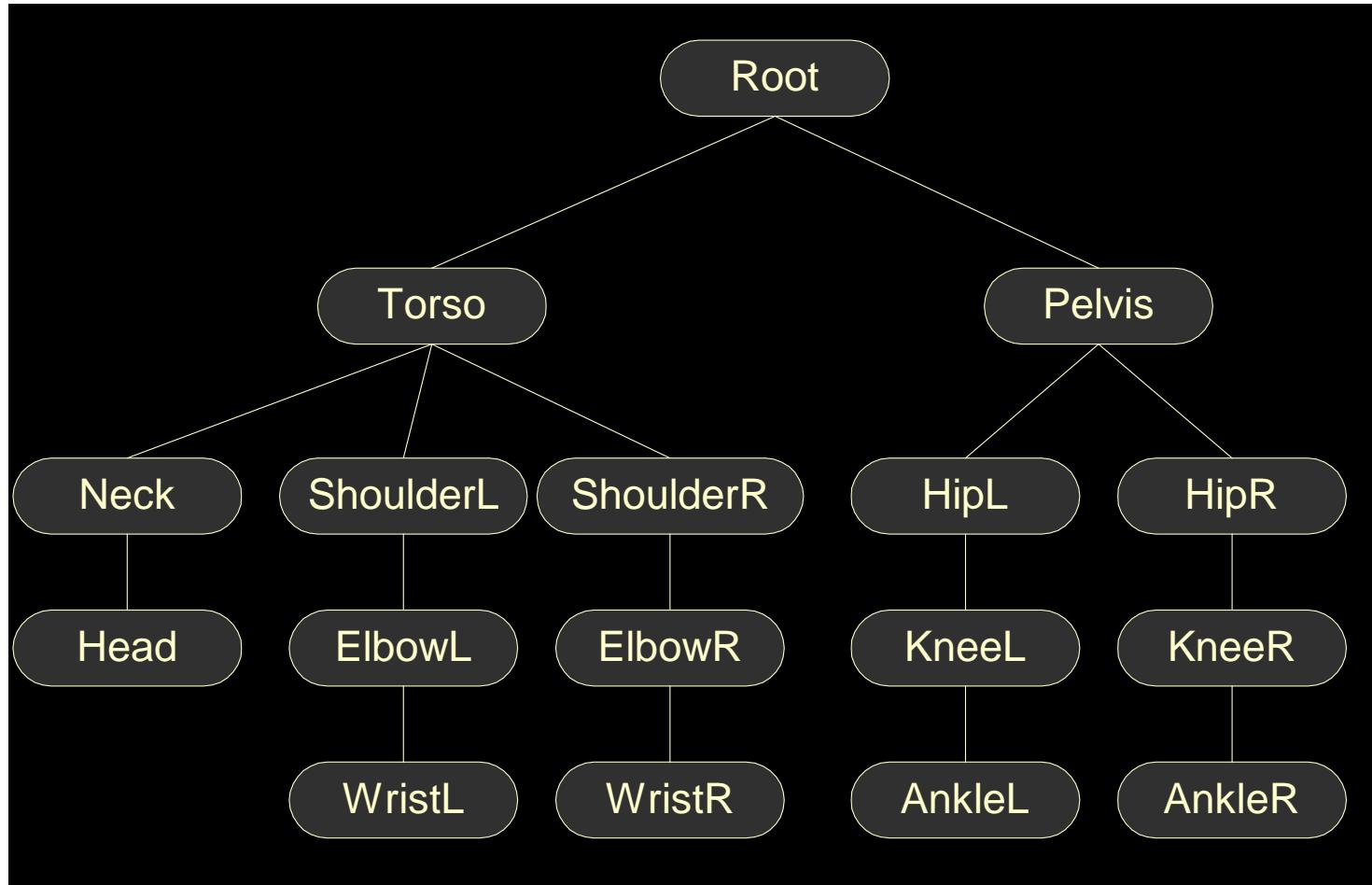
- Direct Acyclic Graph (DAG) stores a position of each wheel.
- Trees and DAGs – hierarchical methods express the relationships.



# Articulated model

- An articulated model is an example of a hierarchical model consisting of rigid parts and connecting joints.
- The moving parts can be arranged into a tree data structure if we choose some particular piece as the ‘root’.
- For an articulated model (like a biped character), we usually choose the root to be somewhere near the centre of the torso.
- Each joint in the figure has specific allowable *degrees of freedom (DOFs)* that define the range of possible poses for the model.

# Articulated model – biped character



# Hierarchical transformations

- Each *node* in the tree represents an object that has a matrix describing its location and a model describing its geometry.
- When a node up in the tree moves its matrix,
  - it takes its children with it (in other words, rotating a character's shoulder joint will cause the elbow, wrist, and fingers to move as well).
  - so child nodes inherit transformations from their parent node.
- Each node in the tree stores a *local matrix* which is its transformation *relative to its parent*.
- To compute a node's *world space matrix*, we need to concatenate its local matrix with its parent's world matrix:  
$$\mathbf{M}_{\text{world}} = \mathbf{M}_{\text{parent}} \cdot \mathbf{M}_{\text{local}}$$

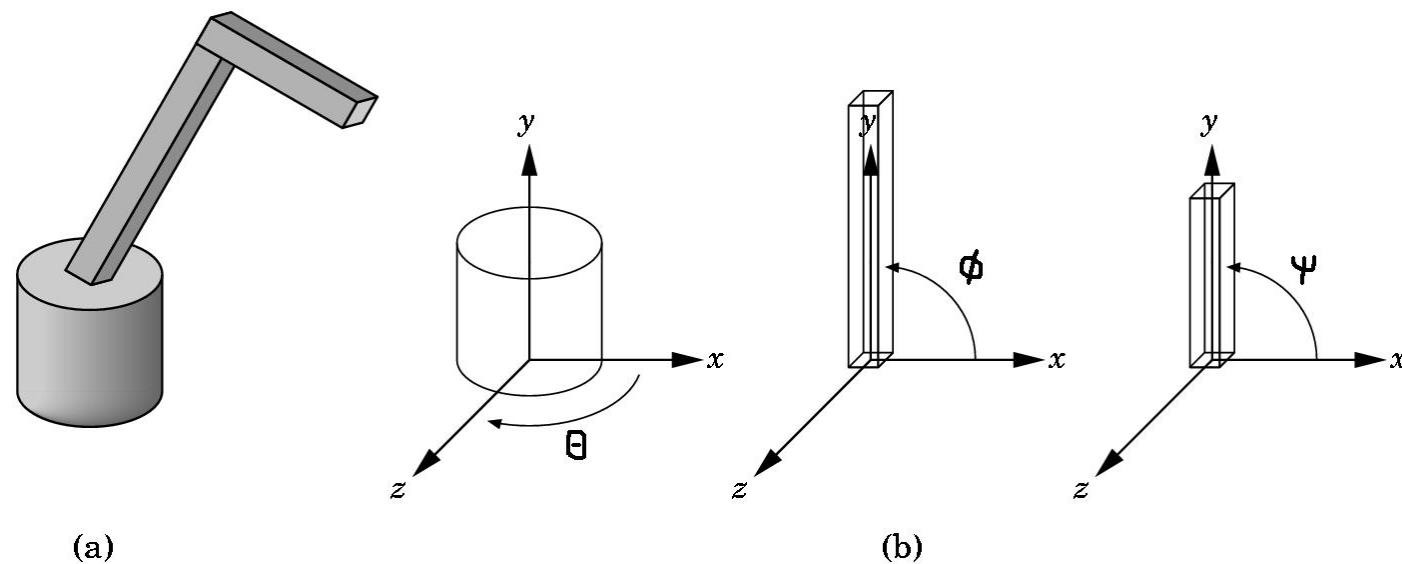
# Recursive traversal and OpenGL matrix stacks

- To compute all of the world matrices in the scene, we can traverse the tree in a *depth-first traversal*.
- As each node is traversed, its world space matrix is computed.
- By the time a node is traversed, it is guaranteed that its parent's world matrix is available.
- The GL matrix stack is set up to facilitate the rendering of hierarchical scenes.
- While traversing the tree, we can call `glPushMatrix()` when going down a level, and `glPopMatrix()` when coming back up.

# Articulated model – robot arm (1)

The robot arm is another example of articulated model.

- Parts are connected at joints.
- We can specify state of model by specifying all joint angles.



Robot arm

Parts in their own frames of reference

# Articulated model – robot arm (2)

- Base rotates independently
  - Single angle determines position
- Lower arm attached to the base
  - Its position depends on the rotation of the base
  - It must also translate relative to the base and rotate around the connecting joint
- Upper arm attached to lower arm
  - Its position depends on both the base and lower arm
  - It must translate relative to the lower arm and rotate around the joint connecting to the lower arm

# Articulated model – robot arm (3)

- Rotate the base:  $R_b$   
Apply  $M_{b-w} = R_b$  to the base
- Translate the lower arm relative to the base:  $T_{la}$
- Rotate the lower arm around the joint:  $R_{la}$   
Apply  $M_{la-w} = R_b \cdot T_{la} \cdot R_{la}$  to the lower arm
- Translate the upper arm relative to the lower arm:  
 $T_{ua}$
- Rotate the upper arm around the joint:  $R_{ua}$   
Apply  $M_{ua-w} = R_b \cdot T_{la} \cdot R_{la} \cdot T_{ua} \cdot R_{ua}$  to the upper arm

# Articulated model – robot arm (4)

- Each of the 3 parts has 1 degree of freedom – described by a joint angle between them.

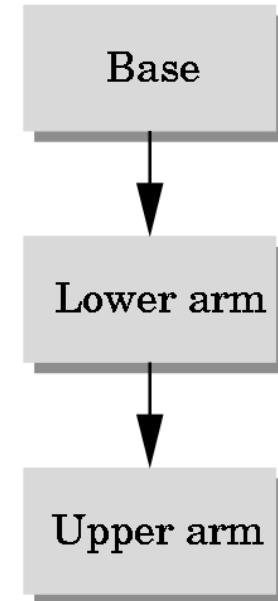
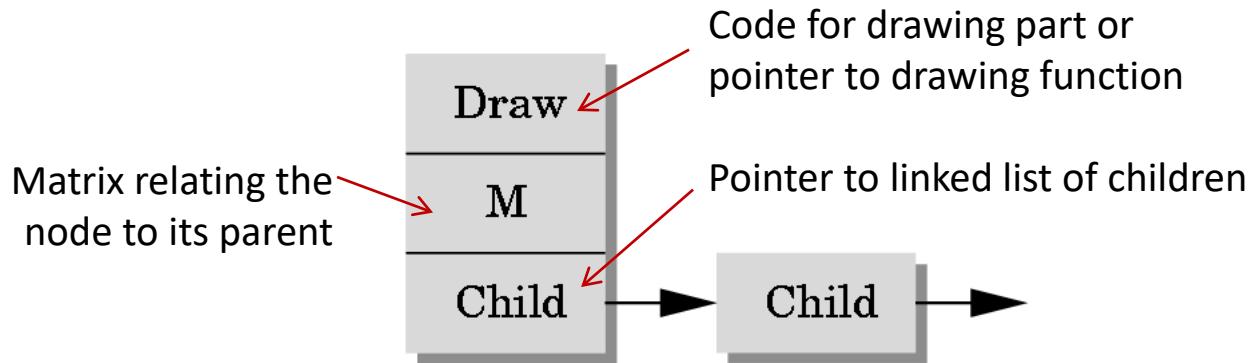
```
void display()
{
    glRotatef(theta, 0.0, 1.0, 0.0);
    base();
    glTranslatef(0.0, h1, 0.0);
    glRotatef(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glTranslatef(0.0, h2, 0.0);
    glRotatef(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
```

- The code shows relationships between the parts of the model. The appearance can change easily without altering the relationships.
- The MODELVIEW matrix for the upper arm is  
$$M_{ua-w} = R_b(\theta) \cdot T_{la}(h1) \cdot R_{la}(\phi) \cdot T_{ua}(h2) \cdot R_{ua}(\psi)$$

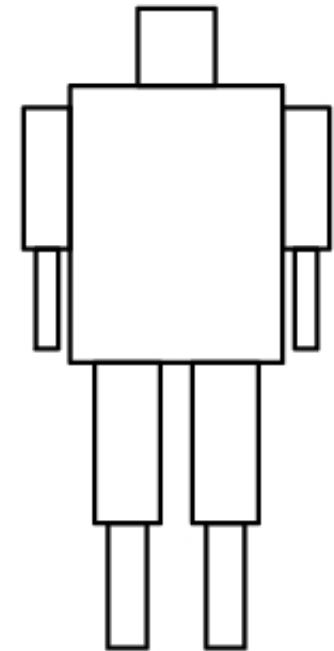
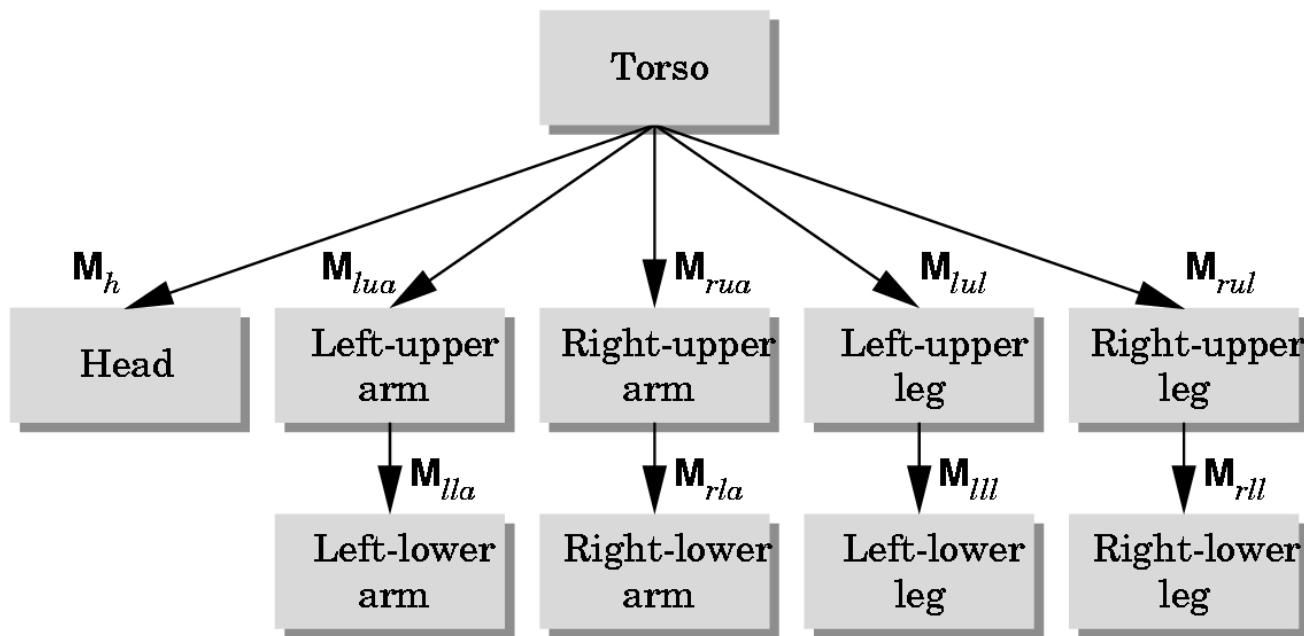
# Articulated model – robot arm (5)

If information is stored in the nodes (not in edges), each node must store at least:

- A pointer to a function that draws the object represented by the node.
- A matrix that positions, orients and scales the object of the node relative to the node's parent (including its children).
- A pointer to its children.



# A humanoid model



# A humanoid model – building the model

- We can build a simple implementation using quadrics: ellipsoids and cylinders.
- Access parts through functions such as
  - `torso()`
  - `left_upper_arm()`
- Matrices describe the position of a node with respect to its parent.
  - e.g.  $M_{lla}$  positions left lower arm with respect to left upper arm.

# A humanoid model – traversal and display

- The position of the figure is determined by 11 joint angles.
- Display of the tree can be thought of as a graph traversal.
  - Visit each node once.
  - Execute the display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation.

# A humanoid model – transformation matrices

There are 10 relevant matrices.

- $M_t$  positions and orients the entire figure through the torso which is the root node.
- $M_h$  positions the head with respect to the torso.
- $M_{l_{ua}}$ ,  $M_{r_{ua}}$ ,  $M_{l_{ul}}$ ,  $M_{r_{ul}}$  position the arms and legs with respect to the torso.
- $M_{l_{ll_a}}$ ,  $M_{r_{ll_a}}$ ,  $M_{l_{ll}}$ ,  $M_{r_{ll}}$  position the lower parts of the limbs with respect to the corresponding upper limbs (parents).

# A humanoid model – tree and traversal

- All matrices are incremental and any traversal algorithm can be used (depth-first or breadth-first). We can traverse from the left to right and depth first.
- Explicit traversal in the code is performed, using stacks to store required matrices and attributes.
- Recursive traversal code is simpler, and the storage of matrices and attributes is made implicitly.

# A humanoid model – stack-based traversal

- Set model-view matrix  $\mathbf{M}$  to  $\mathbf{M}_t$  and draw the torso.
- Set model-view matrix  $\mathbf{M}$  to  $\mathbf{M}_t \cdot \mathbf{M}_h$  and draw the head.
- For the left-upper arm, we need  $\mathbf{M}_t \cdot \mathbf{M}_{luu}$  and so on.
- Rather than re-computing  $\mathbf{M}_t \cdot \mathbf{M}_{luu}$  from scratch or using an inverse matrix, we can use the matrix stack to store  $\mathbf{M}_t \cdot \mathbf{M}_{luu}$  and other matrices as we traverse the tree.

Note that the model-view matrix for the left lower arm is

$$\mathbf{M}_{lla-w} = \mathbf{M}_t \cdot \mathbf{M}_{luu} \cdot \mathbf{M}_{lla}$$

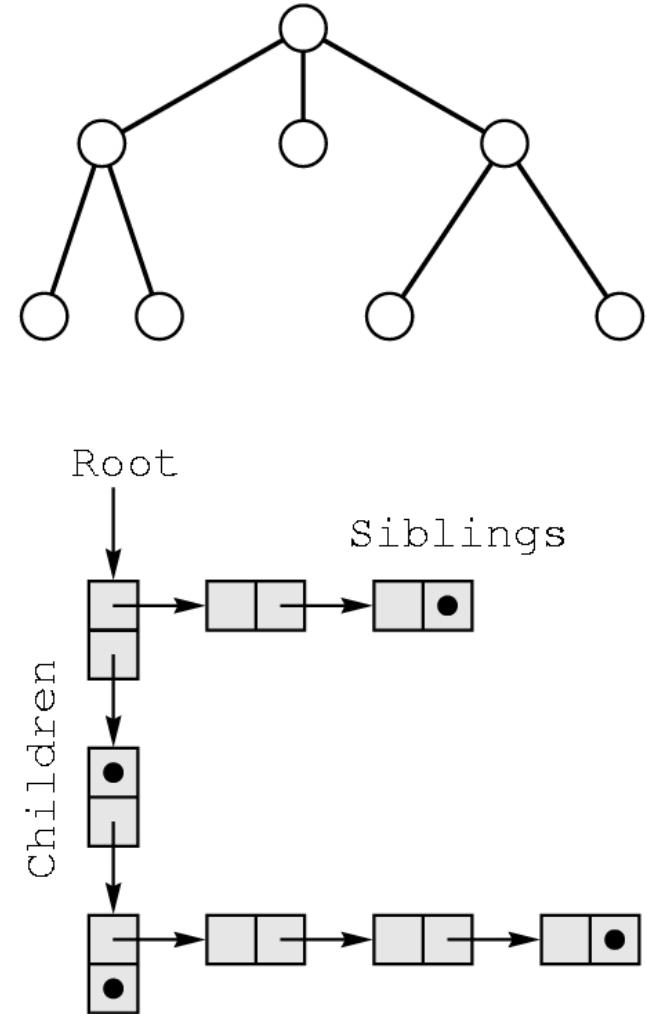
# A humanoid model – traversal code

```
void figure() {
    glPushMatrix();      // save present MODELVIEW matrix
    torso();
    glTranslatef();     // update MODELVIEW matrix for the head
    glRotate3();
    head();
    glPopMatrix();      // recover MODELVIEW matrix for the
    glPushMatrix();      // torso and save the state
    glTranslatef();     // update MODELVIEW matrix for
    glRotate3();         // the left upper leg
    left_upper_leg();
    glTranslatef();
    glRotate3();
    left_lower_leg();  // incremental change
    glPopMatrix();      // recent state recovery
    glPushMatrix();
    ...;
}
```

# A humanoid model – tree data structure (1)

```
typedef struct treenode
{
    GLfloat m[16];
    void(*f)();
    struct treenode *sibling;
    struct treenode *child;
} treenode;

...
treenode torso_node,
        head_node,
        ...;
```



# A humanoid model – tree data structure (2)

```
// for the torso
glLoadIdentity();
glRotatef(theta[0], 0.0, 1.0, 0.0);
glGetFloatv(GL_MODELVIEW_MATRIX, torso_node.m);
// matrix elements copied to the M of the node

// the torso node has no sibling; and
// the leftmost child is the head node

// rest of the code for the torso node
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child = &head_node;
```

# A humanoid model – tree data structure (3)

```
// for the upper-arm node
glLoadIdentity();
glTranslatef(-(TORSO_RADIUS+UPPER_ARM_RADIUS),
              0.9*TORSO_HEIGHT, 0.0)
glRotatef(theta[3], 1.0, 0.0, 0.0);
glGetFloatv(GL_MODELVIEW_MATRIX, lua_node.m);
// matrix elements copied to the m of the node

lua_node.f = left_upper_arm;
lua_node.sibling = &rua_node;
lua_node.child = &lla_node;
```

# A humanoid model – tree data structure (4)

```
// assumption MODELVIEW state
void traverse(treenode* root);
{
    if(root==NULL) return;
    glPushMatrix();
    glMultMatrixf(root->m);
    root->f();
    if(root->child!=NULL) traverse(root->child);
    glPopMatrix();
    if(root->sibling!=NULL) traverse(root->sibling);
} // traversal method is independent of the
// particular tree!
```

# A humanoid model – tree data structure (5)

- We must save model-view matrix (`glPushMatrix`) before multiplying it by the node matrix.
- Updated matrix applies to the children of the node.
- But not to its siblings which contain their own matrices; hence we must return to the previous state (`glPopMatrix`) before traversing the siblings.
- If we are changing attributes within nodes, we can either push (`glPushAttrib`) and pop (`glPopAttrib`) attributes within the rendering functions, or push the attributes when we push the model-view matrix.

# A humanoid model – tree data structure (6)

```
// generic display callback function
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    traverse(&torso_node);
    glutSwapBuffers();
}
```

Animation can then be implemented by controlling the joint angles (i.e. incremented or decremented) via the mouse or keyboard.

# Summary

- Linear modelling does not provide effective ways to retain relationships among the objects of a model.
- Complex models for real world applications can be created and manipulated with hierarchical modelling.
- Hierarchical structure trees are used to implement hierarchical models in computer graphics.
- The code for the humanoid figure is in C (using Struct) and it would be more efficient to write it in C++ (using Class).



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Lighting and Materials**

**Week 10**  
**2021-22**

**Yong Yue**

# Topics for today

- Understanding how real-world lighting works
- Lighting sources
- Lighting effects: ambient, diffuse and specular
- Lighting model and shading
- Rendering illuminated objects by defining the desired light sources
- Material properties of objects being illuminated
- OpenGL functions

# Background

- Realistic displays of a scene are obtained by generating perspective projections of objects and applying natural lighting effects to the visible surfaces.
- Photorealism in computer graphics also involves accurate representation of surface properties and good physical description of the lighting effects in the scene (such as light reflection, transparency, surface texture and shadows).

# Lighting sources

- A light source can be defined with a number of properties, such as position, colour, direction, shape and reflectivity.
- A single value can be assigned to each of the RGB colour components to describe the *amount* or *intensity* of the colour component.

# Lighting sources: Point light

- This is the simplest model for an object, which emits radiant energy with a single colour specified by the three RGB components.
- Large sources can be treated as point emitters if they are not too close to the scene.
- The light rays are generated along radially diverging paths from the light source position.

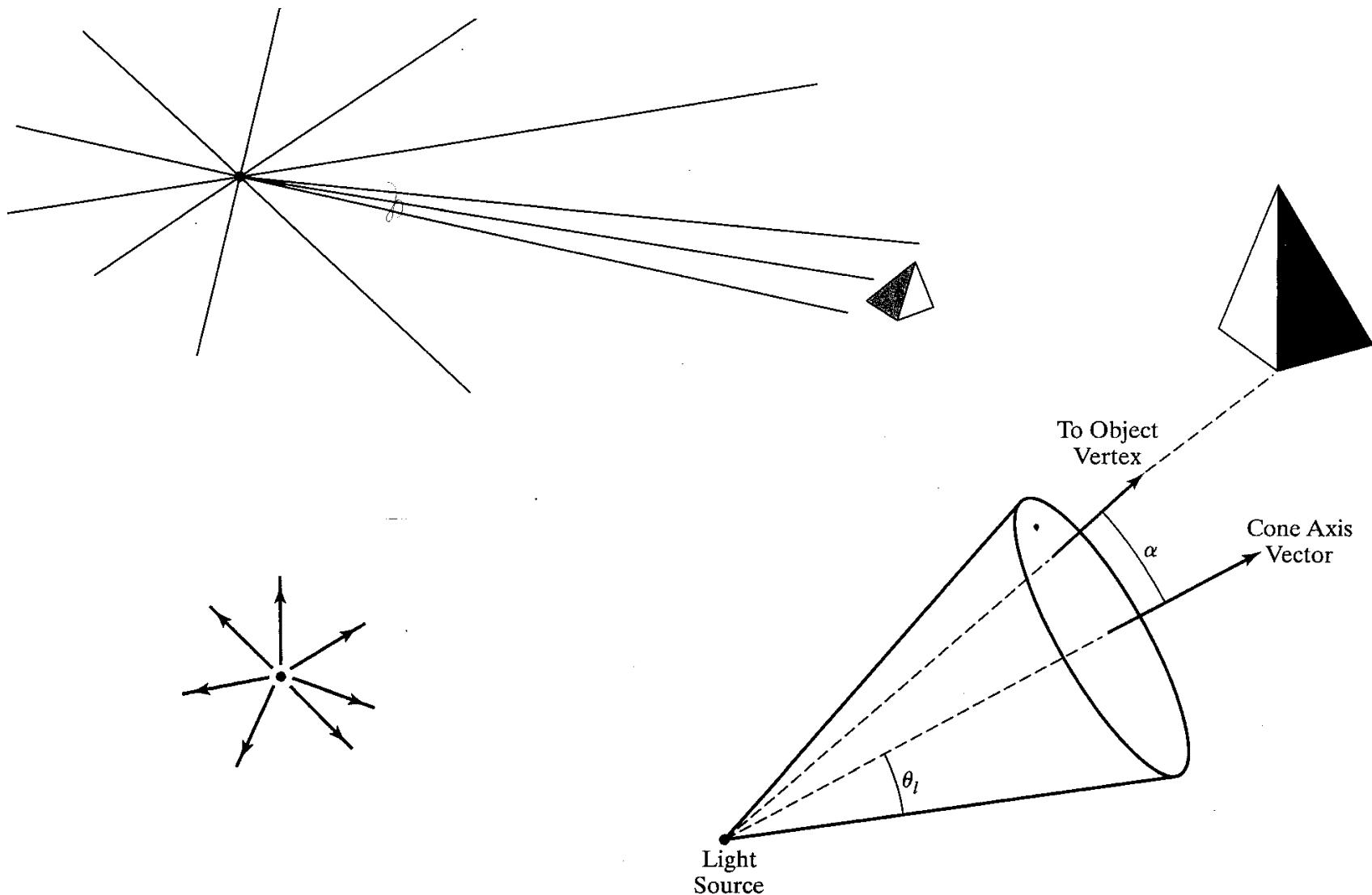
# Lighting sources: Directional light

- A large light source (e.g. the sun), which is very far away from the scene, can also be approximated as a point emitter, but there is little variation in its directional effects. This is called an infinitely distant / directional light.

# Lighting sources: Spot lights

- A local light source can easily be modified to produce a spotlight beam of light.
- If an object is outside the directional limits of the light source, it is excluded for illumination by that light source.
- A spot light source can be set up by assigning a vector direction and an angular limit  $\theta$ , measured from the vector direction along with the position and colour.

# Lighting sources: Illustrated



# Surface lighting effects

- Although a light source delivers a single distribution of frequencies, the ambient, diffuse and specular components might be different.
- For example, if you have a white light in a room with red walls, the scattered light tends to be red, although the light directly striking objects is white.

# Surface lighting effects: Ambient light (1)

The effect is a general background non-directed illumination. Each object is displayed using an intensity intrinsic to it, i.e. a world of non-reflective, self-luminous object.

Consequently each object appears as a monochromatic silhouette, unless its constituent parts are given different shades when the object is created.

# Surface lighting effects: Ambient light (2)

- The ambient component is the lighting effect produced by the reflected light from various surfaces in the scene.
- The light has been scattered so much by the environment that it is impossible to determine its direction - it seems to come from all directions. Back lighting in a room has a large ambient component, since most of the light that reaches the eye has been bounced off many surfaces first.
- A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since it is outdoors, very little of the light reaches the eye after bouncing off other objects. When ambient light strikes a surface, it is scattered equally in all directions.

# Surface lighting effects: Diffuse reflectance (1)

Each object is considered to present a dull, matte surface and all objects are illuminated by a point light source whose rays emanate uniformly in all directions.

The factors which affect the illumination are the point light source intensity, the material's diffuse reflection coefficient, and the angle of incidence of the light.

# Surface lighting effects: Diffuse reflectance (2)

- Diffuse light comes from one direction, so it is brighter if it comes squarely down on a surface than if it barely glances off the surface.
- Once it hits a surface, however, it is scattered equally in all directions, so it appears equally bright, no matter where the eye is located.
- Any light coming from a particular position or direction probably has a diffuse component.
- Rough or grainy surfaces tend to scatter the reflected light in all directions (called diffuse reflection).

# Surface lighting effects: Specular reflectance (1)

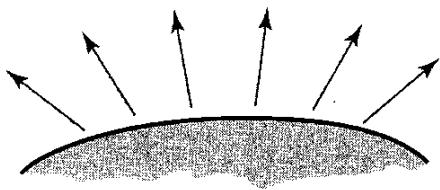
This effect can be seen on any shiny surface and the light reflected off the surface tends to have the same colour as the light source (a white spot from a white light reflected off a red apple). The reflectance tends to fall off rapidly as the viewpoint direction veers away from the direction of reflection (which is equal to the direction to the light source mirrored about the surface normal); for perfect reflectors (i.e. perfect mirrors), specular light appears only in the direction of reflection.

The factors that affect the amount of light reflected are the material's specular-reflection coefficient, the angle of viewing, relative to the direction of reflection, and the light source intensity.

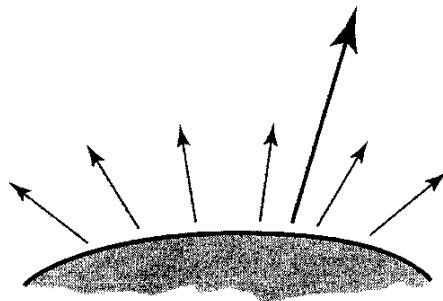
# Surface lighting effects: Specular reflectance (2)

- Specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction.
- A well-collimated laser beam bouncing off a high-quality mirror produces almost 100% specular reflection.
- Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. Specularity can be considered as shininess.

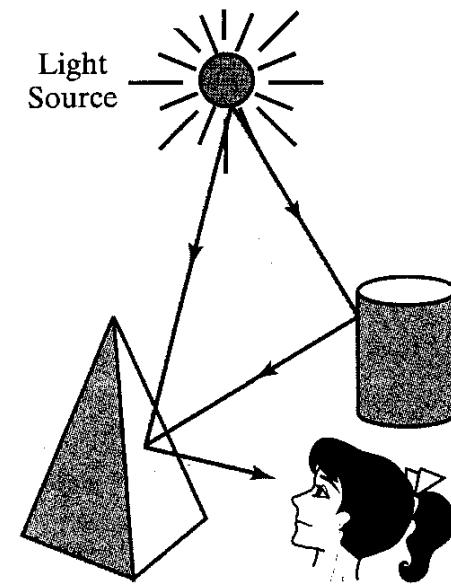
# Surface lighting effects: Illustrated



Diffuse reflection  
from a surface



Specular reflection  
superimposed on  
diffuse reflection vectors



Combined illumination  
from light sources  
and reflection from  
other surface

# Attenuation

For positional/point light sources, we consider the attenuation of light received due to the distance from the source. Attenuation is disabled for directional lights as they are infinitely far away, and it does not make sense to attenuate their intensity over distance.

Although for an ideal source the attenuation is inversely proportional to the square of the distance  $d$ , we can gain more flexibility by using the following distance-attenuation model,

$$f(d) = \frac{1}{k_c + k_l d + k_q d^2}$$

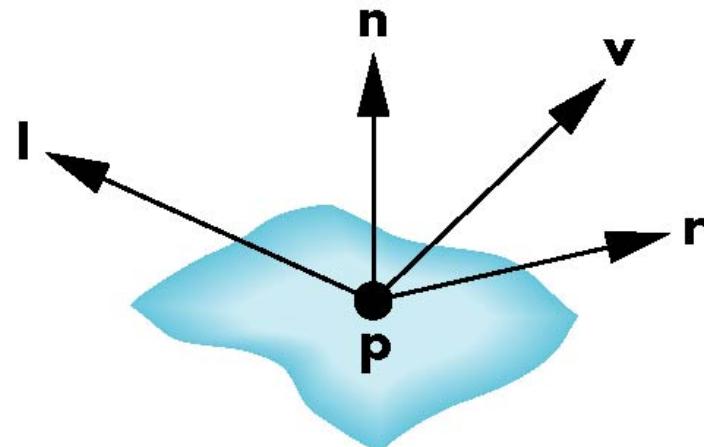
which contains constant, linear and quadratic terms. Three floats for these values,  $k_c$ ,  $k_l$  and  $k_q$  can be used for these terms.

# Lighting model

- A *lighting model* (also called *illumination* or *shading* model) is used to calculate the colour of an illuminated position on the surface of an object.
- The lighting model computes the lighting effects for a surface using various optical properties, which have been assigned to the surface, such as the degree of transparency, colour reflectance coefficients and texture parameters.
- The lighting model can be applied to every projection position, or the surface rendering can be accomplished by interpolating colours on the surface using a small set of lighting-model calculations.
- A *surface rendering method* uses the colour calculation from a lighting model to determine the pixel colours for all projected positions in a scene.

# Phong model

- A simple model that can be computed rapidly
- Three components considered
  - Diffuse
  - Specular
  - Ambient
- Four vectors used
  - To light source  $\mathbf{l}$
  - To viewer  $\mathbf{v}$
  - Face normal  $\mathbf{n}$
  - Perfect reflector  $\mathbf{r}$



# Phong model

- For each point light source, there are 9 coefficients
  - $I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab}$
- Material properties match light source properties
  - 9 absorption coefficients  
 $k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}$
  - Shininess coefficient  $\alpha$
- For each light source and each colour component, the Phong model can be written (without the distance terms) as
  - $I = k_d I_d \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^\alpha + k_a I_a$
- For each colour component, we add contributions from all sources

# Polygonal Shading

- In vertex shading, shading calculations are done for each vertex
  - Vertex colours become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade with a single shade (flat shading)

# Flat (constant) shading

- If the three vectors ( $\mathbf{l}$ ,  $\mathbf{v}$ ,  $\mathbf{n}$ ) are constant, then the shading calculation needs to be carried out only once for each polygon, and each point on the polygon is assigned the same shade.
- The polygonal mesh is usually designed to model a smooth surface, and flat shading will almost always be disappointing because we can see even small differences in shading between adjacent polygons.

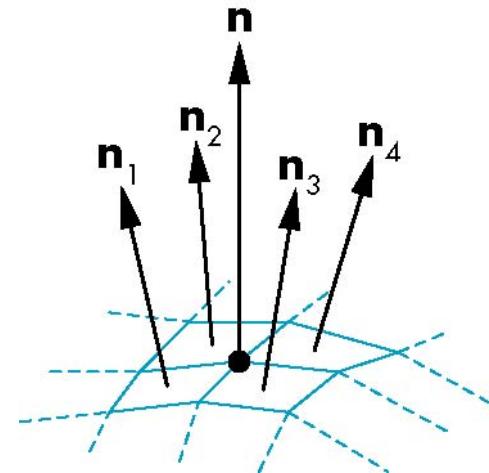


# Smooth (interpolative) shading

- The rasteriser interpolates colours assigned to vertices across a polygon.
- Suppose that the lighting calculation is made at each vertex using the material properties and vectors  $\mathbf{n}$ ,  $\mathbf{v}$ , and  $\mathbf{l}$  computed for each vertex. Thus, each vertex will have its own colour that the rasteriser can use to interpolate a shade for each fragment.
- Note that if the light source is distant, and either the viewer is distant or there are no specular reflections, then smooth (or interpolative) shading shades a polygon in a constant colour.

# Gouraud shading

- The vertex normals are determined (average of the normals around a mesh vertex:  $\mathbf{n} = (\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4) / |\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|$ ) for a polygon and used to calculate the pixel intensities at each vertex, using whatever lighting model.
- Then the intensities of all points on the edges of the polygon are calculated by a process of weighted averaging (linear interpolation) of the vertex values.
- The vertex intensities are linearly interpolated over the surface of the polygon.



# Material colours

- A lighting model may approximate a material colour depending on the percentages of the incoming red, green, and blue light reflected.
- For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light striking it.
- If the ball is lit in white light (composed of equal amounts of red, green and blue light), all the red is reflected, and a red ball is seen. If the ball is viewed in pure red light, it also appears to be red.
- If, however, the ball is viewed in pure green light, it appears black (all the green light is absorbed, and there is no incoming red, so no light is reflected).

# Material colours: Ambient, diffuse and specular

- Like lights, materials have different ambient, diffuse and specular colours, which determine the ambient, diffuse and specular reflectances of the material.
- The ambient reflectance of a material is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance component.
- Ambient and diffuse reflectances define the material colour and are typically similar if not identical.
- Specular reflectance is usually white or grey, so that specular highlights become the colour of the specular intensity of the light source. If a white light shines on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white.

# RGB values for lights

- The colour components specified for lights mean something different from those for materials.
- For a light, the numbers correspond to a percentage of full intensity for each colour. If the R, G and B values for a light colour are all 1.0, the light is the brightest white.
- If the values are 0.5, the colour is still white, but only at half intensity, so it appears grey. If R=G=1 and B=0 (full red and green with no blue), the light appears yellow.

# RGB values for lights and materials

- For materials, the numbers correspond to the reflected proportions of those colours. So if  $R=1$ ,  $G=0.5$  and  $B=0$  for a material, the material reflects all the incoming red light, half the incoming green and none of the incoming blue light.
- In other words, if a light has components  $(R_L, G_L, B_L)$ , and a material has corresponding components  $(R_M, G_M, B_M)$ ; then, ignoring all other reflectivity effects, the light that arrives at the eye is given by  $(R_L * R_M, G_L * G_M, B_L * B_M)$ .

# RGB values for two lights

- Similarly, if two lights,  $(R_1, G_1, B_1)$  and  $(R_2, G_2, B_2)$  are sent to the eye, these components are added up, giving  $(R_1+R_2, G_1+G_2, B_1+B_2)$ .
- If any of the sums are greater than 1 (corresponding to a colour brighter than the equipment can display), the component is clamped to 1.

# OpenGL: Light sources and colour (1)

- Light sources have a number of properties, such as colour, position and direction.
- The command used to specify all properties of lights is **glLight\***( ).
- It takes three arguments: the identity of the light, the property and the desired value for that property.

# OpenGL: Light sources and colour (2)

- `void glLight{if}[v](GLenum light, GLenum pname, TYPEparam)` creates the light specified by *light*, which can be GL\_LIGHT0, GL\_LIGHT1, ..., GL\_LIGHT7.
- The characteristic of the light being set is defined by *pname*, which specifies a named parameter (see table on next slide).
- The *TYPEparam* argument indicates the values to which the *pname* characteristic is set; it is a pointer to a group of values if the vector version is used, or the value itself if the non-vector version is used. The non-vector version can be used to set only single-valued light characteristics.

# OpenGL: Light sources and colour (3)

Parameter Name	Default Value	Meaning
<code>GL_AMBIENT</code>	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
<code>GL_DIFFUSE</code>	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
<code>GL_SPECULAR</code>	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
<code>GL_POSITION</code>	(0.0, 0.0, 1.0, 0.0)	( $x, y, z, w$ ) position of light
<code>GL_SPOT_DIRECTION</code>	(0.0, 0.0, -1.0)	( $x, y, z$ ) direction of spotlight
<code>GL_SPOT_EXPONENT</code>	0.0	spotlight exponent
<code>GL_SPOT_CUTOFF</code>	180.0	spotlight cut-off angle
<code>GL_CONSTANT_ATTENUATION</code>	1.0	constant attenuation factor
<code>GL_LINEAR_ATTENUATION</code>	0.0	linear attenuation factor
<code>GL_QUADRATIC_ATTENUATION</code>	0.0	quadratic attenuation factor

# OpenGL: Light sources and colour (4)

The default values listed for GL\_DIFFUSE and GL\_SPECULAR in the table apply only to GL\_LIGHT0. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL\_DIFFUSE and GL\_SPECULAR. Below is an example of using `glLight*`( ) :

```
GLfloat light_ambient[] = {0.0, 0.0, 0.0, 1.0};  
GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};  
  
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);  
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);  
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);  
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

# OpenGL: Light sources and colour (5)

- Arrays are defined for the parameter values, and `glLightfv()` is called repeatedly to set the various parameters. In this example, the first three calls to `glLightfv()` are superfluous, since they are used to specify the default values for the `GL_AMBIENT`, `GL_DIFFUSE`, and `GL_SPECULAR` parameters.
- Remember to turn on each light with  `glEnable()`;
- All the parameters for `glLight*`( ) and their possible values are explained shortly. These parameters interact with those that define the overall lighting model for a particular scene and the object's material properties.

# OpenGL: Light sources and colour (6)

- As you can see, OpenGL allows you to associate three different colour-related parameters (`GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR`) with any particular light.
- The `GL_AMBIENT` parameter refers to the RGBA intensity of the ambient light that a particular light source adds to the scene.
- The `GL_DIFFUSE` parameter probably most closely correlates with what you naturally think of as “the colour of a light”. It defines the RGBA colour of the diffuse light that a particular light source adds to a scene.

# OpenGL: Light sources and colour (7)

- The GL\_SPECULAR parameter affects the colour of the specular highlight on an object.
- Typically, a real-world object such as a glass bottle has a specular highlight that is the colour of the light shining on it (often white).
- Therefore, if you want to create a realistic effect, set the GL\_SPECULAR parameter to the same value as the GL\_DIFFUSE parameter.

# OpenGL: Light position and direction (1)

- A light source can be treated as though it is located infinitely far away from the scene or close to the scene.
- The first type is referred to as a *directional* light source; the effect of an infinite location is that the rays of light can be considered parallel.
- The second type is called a *positional* light source, since its exact position within the scene determines its effect on the scene and, specifically, the direction from which the light rays come. A desk lamp is an example of a positional light source. The light specified below is directional.

```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};  
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

# OpenGL: Light position and direction (2)

- As shown, a vector of four values ( $x, y, z, w$ ) is specified for GL\_POSITION. If the last value,  $w$ , is 0.0, the corresponding light source is a directional one, and the  $(x, y, z)$  values describe its direction. This direction is transformed by the model-view matrix. By default, GL\_POSITION is  $(0, 0, 1, 0)$ , which defines a directional light that points along the negative z-axis.
- If the  $w$  value is nonzero, the light is positional, and the  $(x, y, z)$  values specify the location of the light in homogeneous object co-ordinates. This location is transformed by the model-view matrix and stored in eye co-ordinates.

# OpenGL: Light position and direction (3)

- OpenGL treats the position and direction of a light source just as it treats the position of a geometric primitive. In other words, a light source is subject to the same matrix transformations as a primitive.
- More specifically, when `glLight*`( ) is called to specify the position or the direction of a light source, the position or direction is transformed by the current model-view matrix. This means you can manipulate the position or direction of a light source by changing the contents of the model-view matrix stack.
- Note that the projection matrix has no effect on light position or direction.

# OpenGL: Light position and direction (4)

The following three effects can be implemented by changing the point in the program at which the light position is set, relative to modelling or viewing transformations:

- A light position that remains fixed
- A light that moves around a stationary object
- A light that moves along with the viewpoint

# OpenGL: Attenuation

- By default,  $k_c$  is 1.0 and both  $k_l$  and  $k_q$  are zero, but you can give these parameters different values

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);  
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);  
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

- Note that the ambient, diffuse and specular contributions are all attenuated. Only the emission and global ambient values are not attenuated.

# OpenGL: Spotlight (1)

- Note that no light is emitted beyond the edges of the cone.
- By default, the spotlight feature is disabled because the GL\_SPOT\_CUTOFF parameter is 180.0. This value means that light is emitted in all directions (the angle at the cone's apex is 360 degrees, so it is not a cone at all).
- The value for GL\_SPOT\_CUTOFF is restricted to being within the range [0.0,90.0] (unless it has the special value 180.0).
- The following line sets the cut-off parameter to 45 degrees:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

# OpenGL: Spotlight (2)

- You also need to specify a spotlight direction, which determines the axis of the cone:

```
GLfloat spot_direction[] = {-1.0, -1.0, 0.0};  
  
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

- The direction is specified in homogeneous object coordinates, the default direction is (0.0, 0.0, -1.0), i.e. the light points down the negative z-axis.
- Also, keep in mind that a spotlight direction is transformed by the model-view matrix just as though it were a normal vector.

# OpenGL: Spotlight (3)

- In addition to the spotlight cut-off angle and direction, you can control the intensity distribution of the light within the cone, in two ways.
- First, you can set the attenuation factor described earlier.
- You can also set the `GL_SPOT_EXPONENT` parameter, to control how concentrated the light is.
- The light intensity is the highest in the centre of the cone. It is attenuated towards the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the vertex lit exponentially. Thus, a higher spot exponent results in a more focused light source.

# OpenGL: Multiple lights (1)

- As aforementioned, you can have up to 8 lights in your scene (possibly more, depending on your OpenGL implementation).
- Since OpenGL needs to perform calculations to determine how much light each vertex receives from each light source, increasing the number of lights adversely affects performance.
- The following lines of code define a point light and a spotlight:

# OpenGL: Multiple lights (2)

```
GLfloat light1_ambient[] = {0.2, 0.2, 0.2, 1.0};  
GLfloat light1_diffuse[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light1_specular[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light1_position[] = {-2.0, 2.0, 1.0, 1.0};  
GLfloat spot_direction[] = {-1.0, -1.0, 0.0};  
  
glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);  
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);  
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);  
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);  
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);  
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);  
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);  
  
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);  
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);  
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2.0);  
  
 glEnable(GL_LIGHT1);  
 glEnable(GL_LIGHT0)
```

# OpenGL: Selecting a lighting model (1)

- ***Global Ambient Light.*** Each light source can contribute ambient light to a scene. In addition, there can be other ambient light that is not from any particular source. The GL\_LIGHT\_MODEL\_AMBIENT parameter is used to specify the RGBA intensity of such global ambient light

```
GLfloat lmodel_ambient[] = {0.2, 0.2, 0.2, 1.0};  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

- In the example, the values used for *lmodel\_ambient* are the default values for GL\_LIGHT\_MODEL\_AMBIENT. Since these numbers yield a small amount of white ambient light, even if you do not add a specific light source to your scene, you can still see the objects in the scene.

# OpenGL: Selecting a lighting model (2)

- ***Local or Infinite Viewpoint.*** The location of the viewpoint affects the calculations for highlights produced by specular reflectance.
- More specifically, the intensity of the highlight at a particular vertex depends on the normal at that vertex, the direction from the vertex to the light source, and the direction from the vertex to the viewpoint.
- Keep in mind that the viewpoint is not actually moved by calls to lighting commands (you need to change the projection transformation); instead, different assumptions are made for the lighting calculations as if the viewpoint were moved.

# OpenGL: Selecting a lighting model (3)

- With an infinite viewpoint, the direction between it and any vertex in the scene remains constant. A local viewpoint tends to yield more realistic results, but since the direction has to be calculated for each vertex, overall performance is decreased.
- By default, an infinite viewpoint is assumed. Here is how to change to a local viewpoint

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

- This call places the viewpoint at (0, 0, 0). To switch back to an infinite viewpoint, pass in GL\_FALSE as the argument.

# OpenGL: Selecting a lighting model (4)

- ***Two-sided Lighting.*** Lighting calculations are performed for all polygons, whether they are front-facing or back-facing.
- You usually set up lighting conditions with the front-facing polygons in mind, but the back-facing polygons typically are not correctly illuminated.
- If the object is a sphere, only the front faces are ever seen, since they are the ones on the outside of the sphere. So, in this case, it does not matter what the back-facing polygons look like. If the sphere is cut away so that its inside surface would be visible, however, you may want to have the inside surface fully lit according to the lighting conditions defined; you may also want to supply a different material description for the back faces.

# OpenGL: Defining material properties (1)

- Most of the material properties are conceptually similar to those used to create light sources. The mechanism for setting them is similar, except that the command used is called **glMaterial\***( ).

```
void glMaterial{if}[v](GLenum face, GLenum pname,  
                      TYPEparam)
```

- The code above specifies a current material property for lighting calculations.
- The face parameter can be GL\_FRONT, GL\_BACK, or GL\_FRONT\_AND\_BACK to indicate which face of the object the material should be applied to.

# OpenGL: Defining material properties (2)

- The particular material property being set is identified by *pname* and the desired values for that property are given by *TYPEparam*, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the non-vector version is used).
- The non-vector version works only for setting GL\_SHININESS.
- The possible values for *pname* are shown in the table on the next slide. Note that GL\_AMBIENT\_AND\_DIFFUSE allows you to set both the ambient and diffuse material colours simultaneously to the same RGBA values.

# OpenGL: Defining material properties (3)

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient colour of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse colour of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse colour of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular colour of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive colour of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse and specular colour indices

# OpenGL: Defining material properties (4)

- You can choose to have lighting calculations performed differently for the front- and back-facing polygons of objects. If the back faces may be seen, you can supply different material properties for the front and the back surfaces by using the *face* parameter of **glMaterial\***( ).
- Note that most of the material properties set with **glMaterial\***( ) are RGBA colours. Regardless of what alpha values are supplied for other parameters, the alpha value at any particular vertex is the diffuse-material alpha value, that is, the alpha value given to GL\_DIFFUSE with the **glMaterial\***( ).
- Also, none of the RGBA material properties apply in colour-index mode.

# OpenGL: Defining material properties (5)

- Diffuse reflectance plays the most important role in determining what you perceive the colour of an object.
- Ambient reflectance affects the overall colour of the object. Because ambient reflectance is the brightest where an object is directly illuminated, it is the most noticeable where an object receives no direct illumination.
- The total ambient reflectance for an object is affected by the global ambient light and ambient light from individual light sources.
- Like diffuse reflectance, ambient reflectance is not affected by the position of the viewpoint.

# OpenGL: Defining material properties (6)

- Specular reflection from an object produces highlights. Unlike ambient and diffuse reflections, the amount of specular reflection seen by a viewer depends on the location of the viewpoint – it is brightest along the direct angle of reflection.
- By specifying an RGBA colour for GL\_EMISSION, you can make an object appear to be giving off light of that colour. Since most real-world objects (except lights) do not emit light, this feature is probably used to simulate lamps and other light sources in a scene.

# Summary

- Lighting and materials together have important effects on the graphics rendering.
- There is a range of factors to consider: lighting sources (positional, spot and directional) and effects (ambient, diffuse and specular), lighting models (e.g. Phong model) and shading methods.
- Global ambient light and multiple light sources are combined with material properties.
- Approximations (e.g. ignoring angles between light rays for a light source at a relatively large distance, or angles between the viewer and any vertex in a scene for an infinite viewpoint) are applied in order to reduce the expensive computational work.



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Texture Mapping**

**Week 11**  
**2020-21**

**Yong Yue**

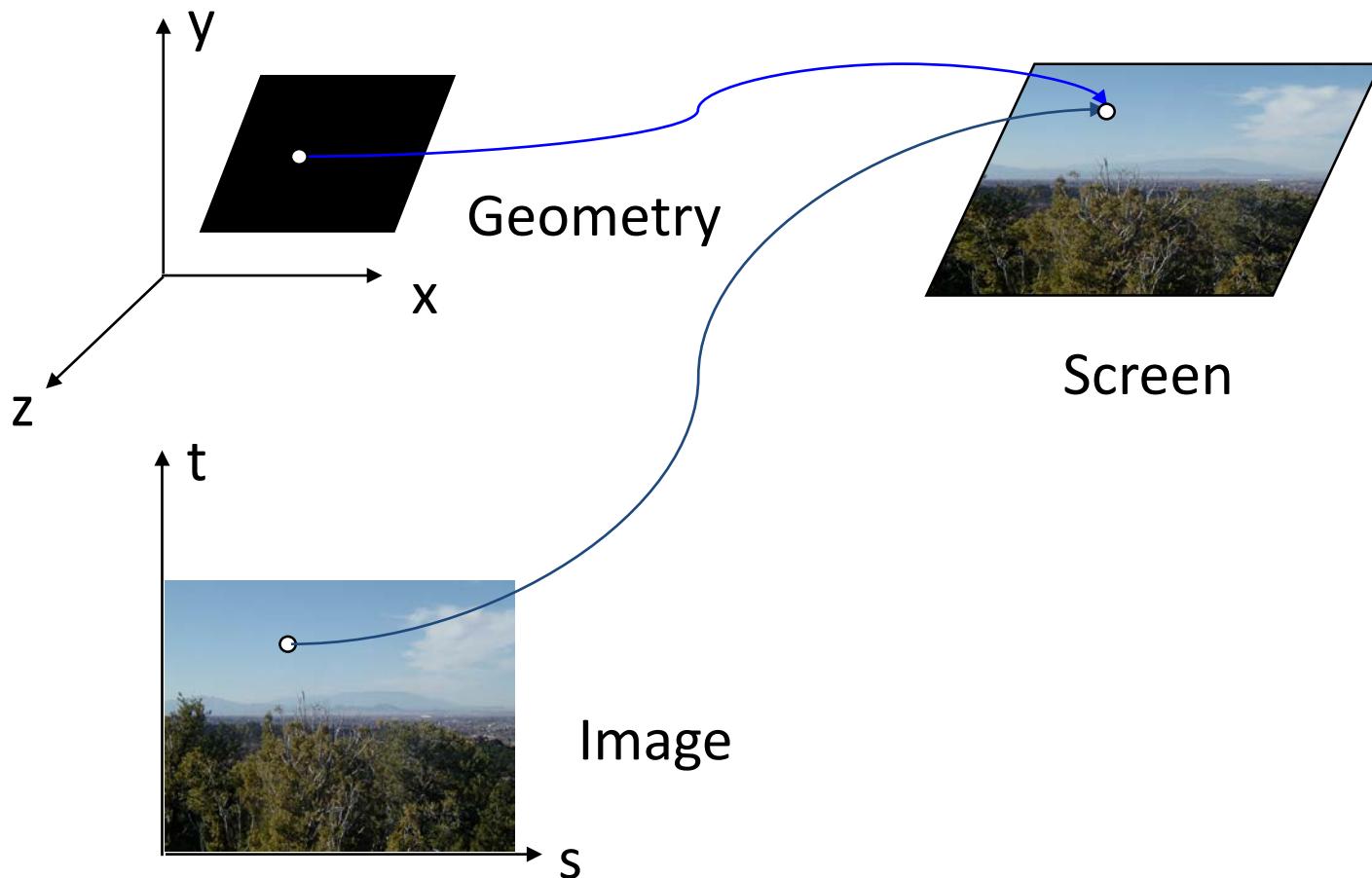
# Topics for today

- Concepts
- Types of texture mapping
- Specifying the texture
- Magnification and minification
- Multiple levels of detail
- Modes, filtering and wrapping
- Steps in texture mapping
- OpenGL functions

# Concepts of texture mapping (1)

- Although graphics cards can render over 10 million polygons per second, this may be insufficient or there could be alternative way to process phenomena, e.g. clouds, grass, terrain and skin.
- A common method for adding detail to an object is to map patterns onto the geometric description of the object.  
The method for incorporating object detail into a scene is called texture mapping or pattern mapping.
- What makes texture mapping tricky is that a rectangular texture can be mapped to nonrectangular regions, and this must be done in a reasonable way.
- When a texture is mapped, its display on the screen might be distorted due to various transformations applied – rotations, translations, scaling and projections.

# Concepts of texture mapping (2)



# Types of texture mapping (1)

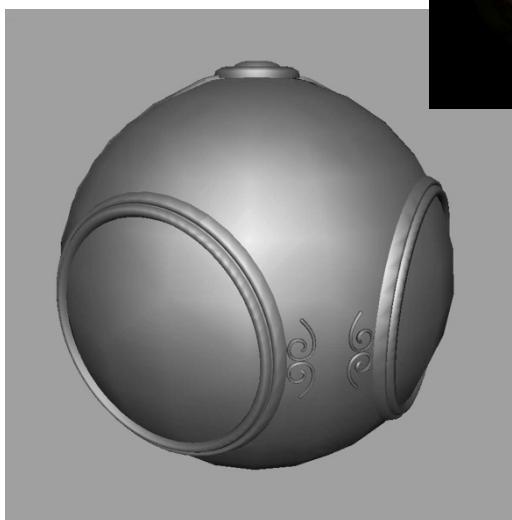
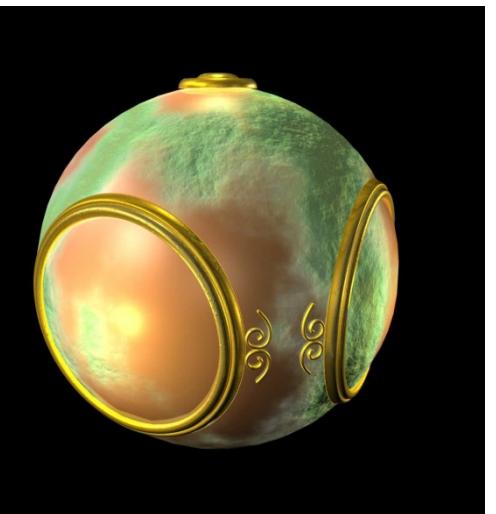
- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection) mapping
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process

# Types of texture mapping (2)

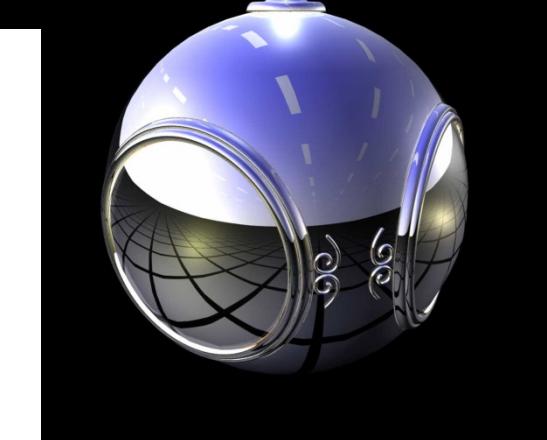
Texture mapped



Bump mapped



Geometric model



Environment mapped

# Specifying the texture (1)

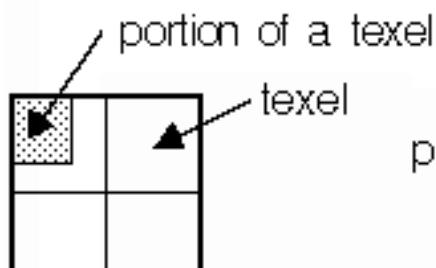
- There are two common types of texture:
  - **image**: a 2D image is used as the texture.
  - **procedural**: a program or procedure generates the texture.
- The texture can be defined as one-dimensional, two-dimensional or three-dimensional patterns.

# Specifying the texture (2)

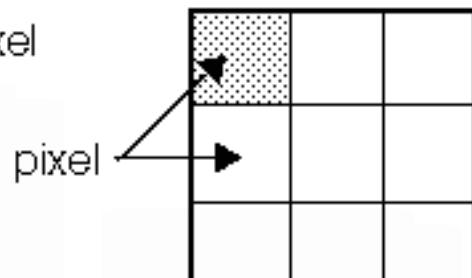
- Texture functions in a graphics package often allow the number of colour components for each position in a pattern to be specified as an option.
- For example each colour definition in a texture pattern could consist of four RGBA components, three RGB components, a single intensity value for a shade of blue, an index into a colour table, or a single luminance value (a weighted average of the RGB components of a colour).
- The individual values in a texture array are often called *texels* (texture elements).

# Magnification and minification (1)

- Depending on the texture size, the distortion and the size of the screen image, a texel may be mapped to more than one pixel (called *magnification*), and a pixel may be covered by multiple texels (called *minification*).

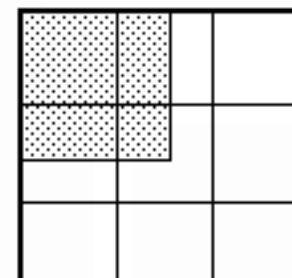


Texture

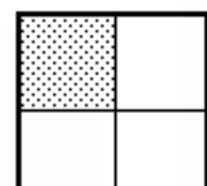


Polygon

Magnification



Texture



Polygon

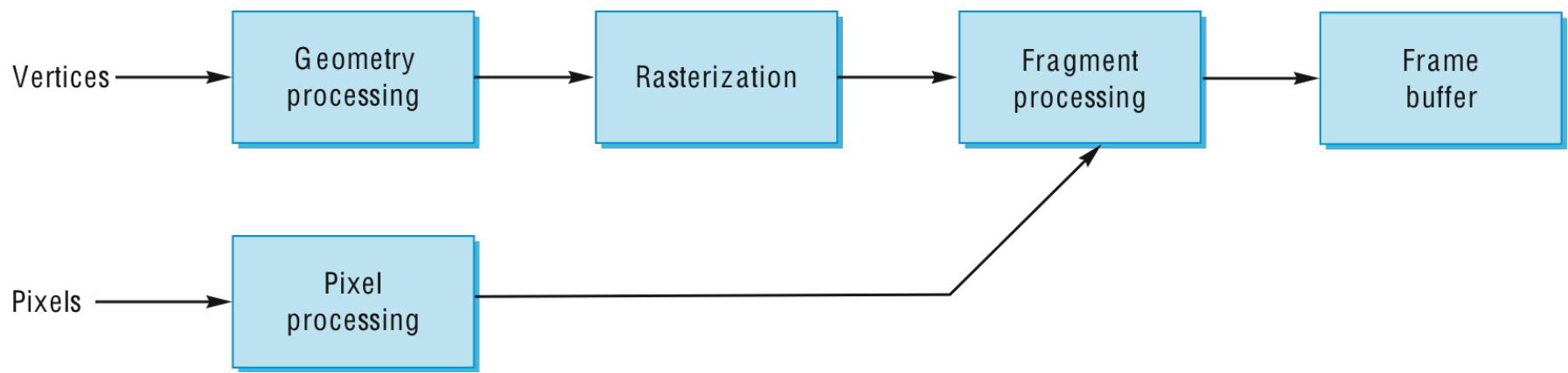
Minification

# Magnification and minification (2)

- Since the texture is made up of discrete texels, filtering operations must be performed to map texels to pixels.
- For example, if many texels correspond to a pixel, they are averaged down to fit; if texel boundaries fall across pixel boundaries, a weighted average of the applicable texels is performed.
- Because of these calculations, texture mapping can be computationally expensive, which is why many specialised graphics systems include hardware support for texture mapping.

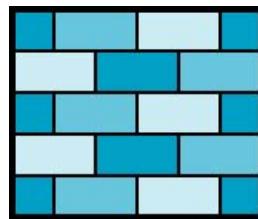
# Where does it take place?

- Texture mapping techniques are implemented at the end of the rendering pipeline.
- It is very efficient because a few polygons make it past the clipper.

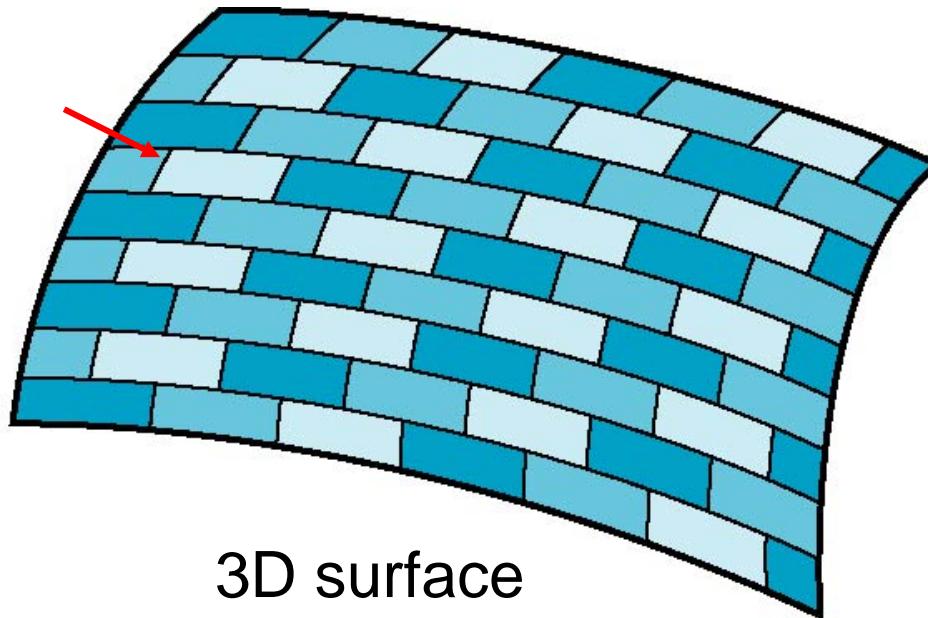


# Is it simple?

Although the idea is simple, mapping an image to a surface, 3 or 4 coordinate systems are involved.



2D image

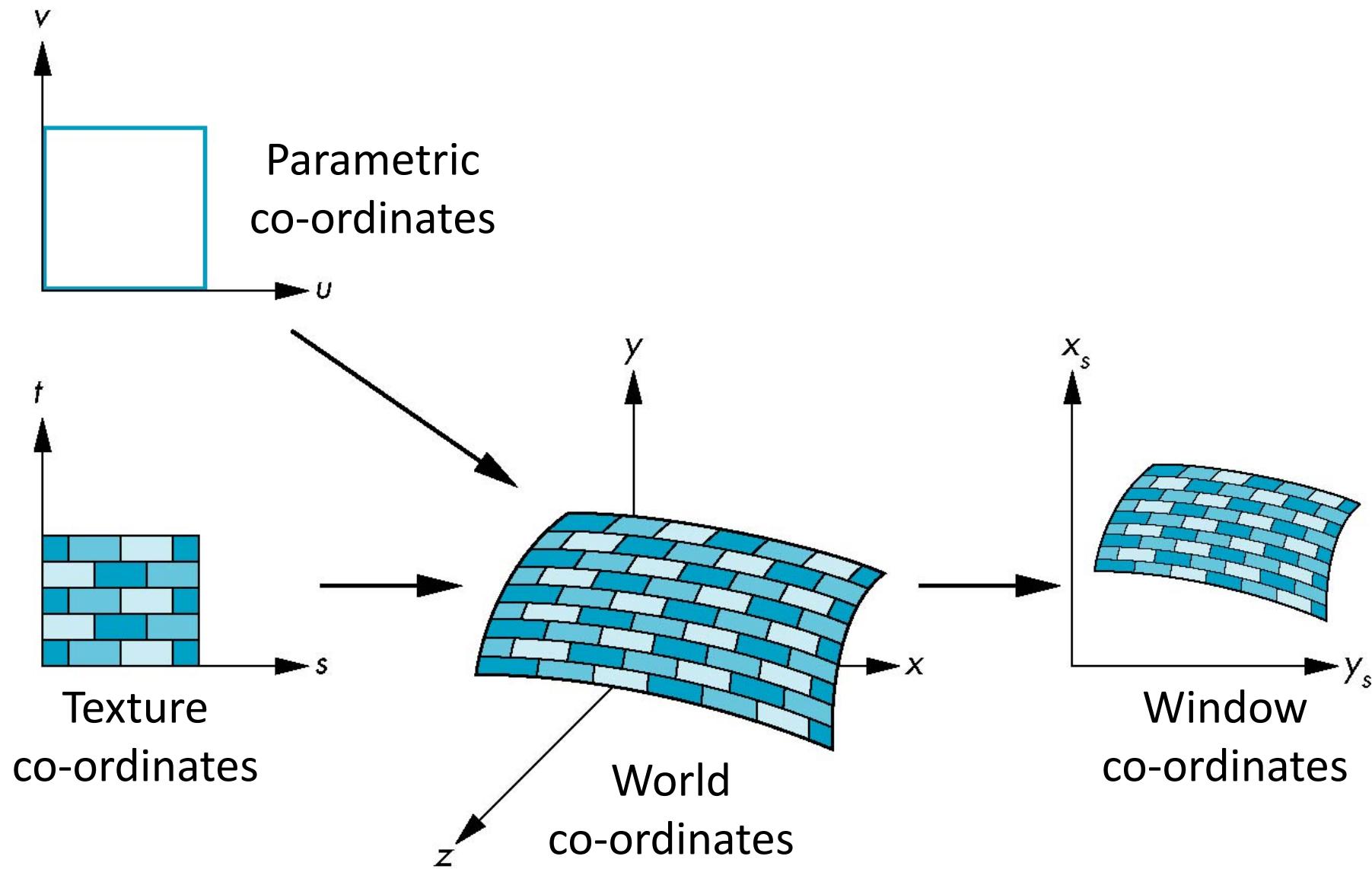


3D surface

# Co-ordinate systems for texture mapping

- Parametric co-ordinates  
may be used to model curves and surfaces
- Texture co-ordinates  
used to identify points in the image to be mapped
- Object or world co-ordinates  
conceptually, where the mapping takes place
- Window co-ordinates  
where the final image is finally produced

# Co-ordinate systems for texture mapping



# Co-ordinate systems for texture mapping

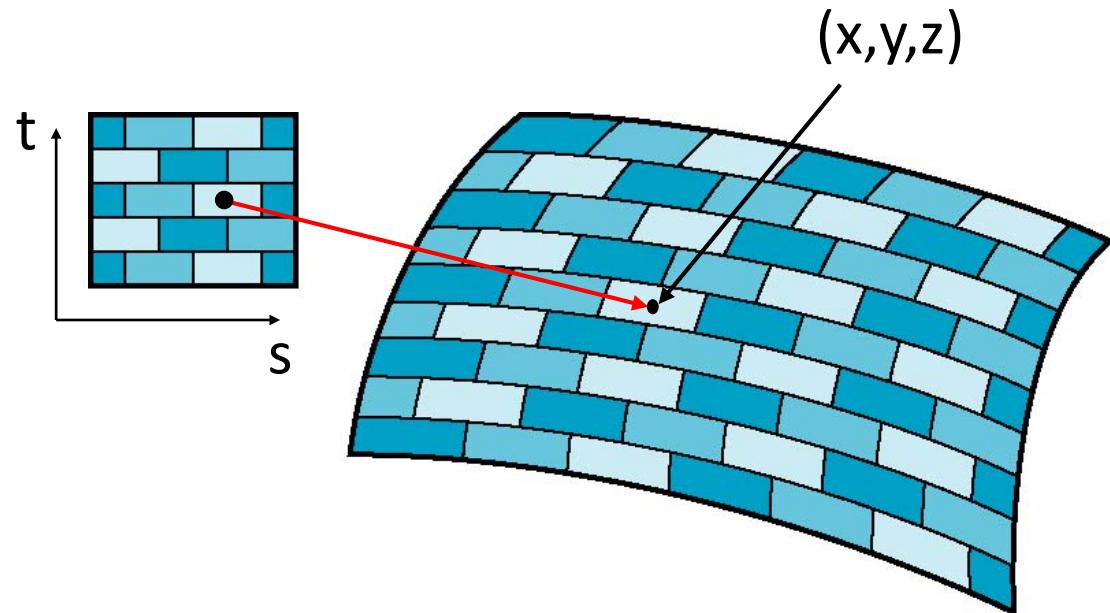
- The basic problem is how to find the maps.
- Consider mapping from texture co-ordinates to a point on a surface.
- Apparently three functions are needed

$$x = x(s,t)$$

$$y = y(s,t)$$

$$z = z(s,t)$$

- But we really want to go the other way.

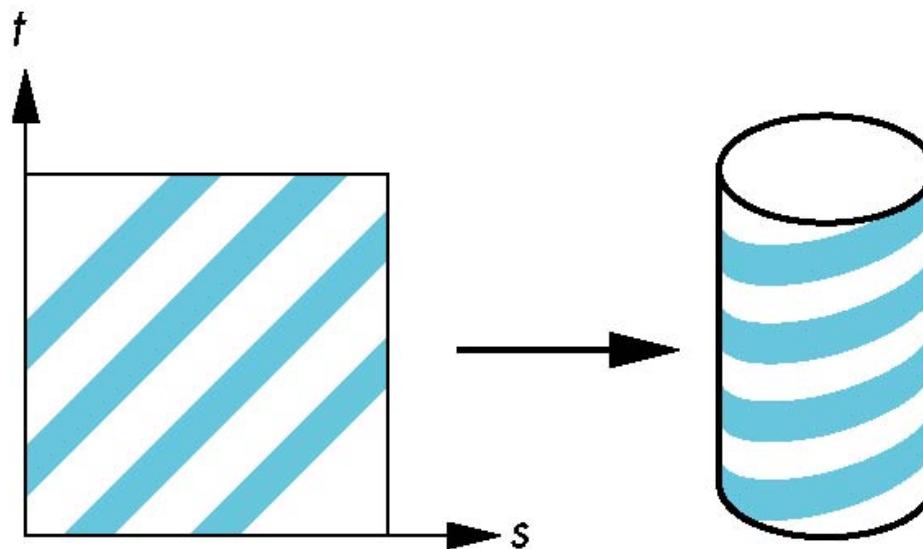


# Backward mapping

- We really want to go backwards:
  - 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.
- Need a map of the form:
$$s = s(x, y, z)$$
$$t = t(x, y, z)$$
- Such functions are difficult to find in general.

# Two-part mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface.
- Example: mapping to cylinder.



# Cylindrical mapping

A parametric cylinder

$$x = r \cdot \cos(2\pi \cdot u)$$

$$y = r \cdot \sin(2\pi \cdot u)$$

$$z = v \cdot h$$

maps a rectangle in the  $(u,v)$  space to the cylinder of radius  $r$  and height  $h$  in the world co-ordinates

$$s = u$$

$$t = v$$

maps from the texture space.

# Spherical mapping

We can map a parametric sphere

$$x = r \cdot \cos(2\pi \cdot u)$$

$$y = r \cdot \sin(2\pi \cdot u) \cdot \cos(2\pi \cdot v)$$

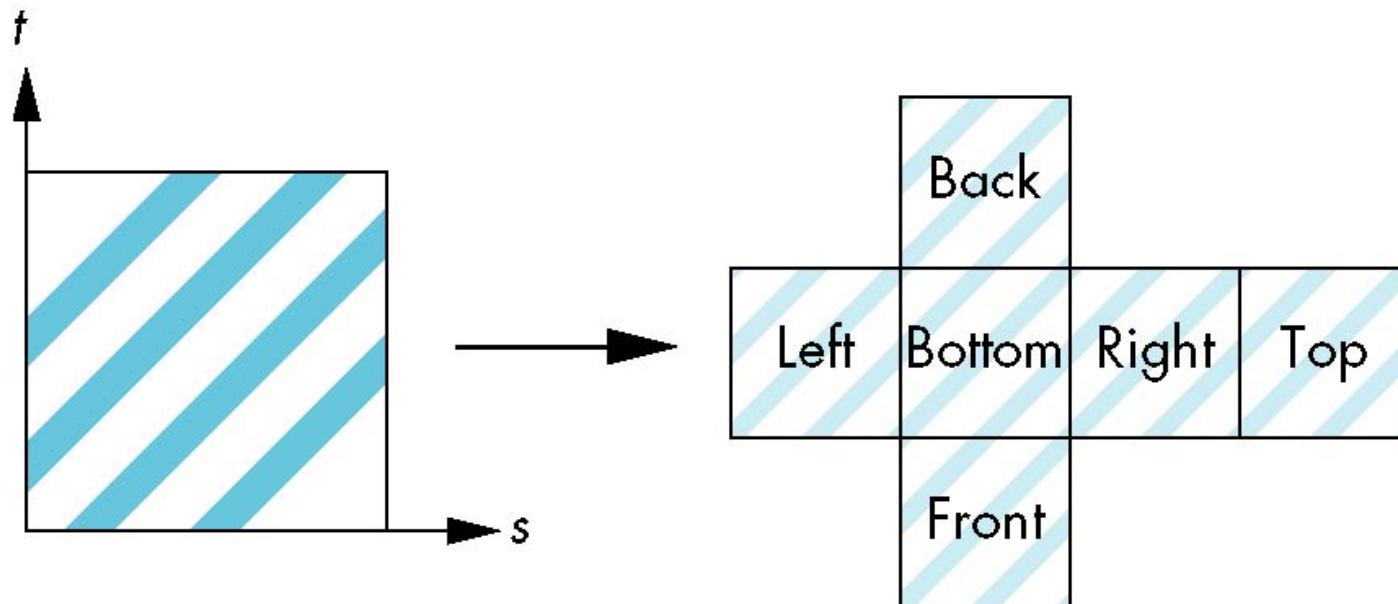
$$z = r \cdot \sin(2\pi \cdot u) \cdot \sin(2\pi \cdot v)$$

in a similar manner to the cylinder but have to decide where to put the distortion.

Spheres are used in environmental maps.

# Box mapping

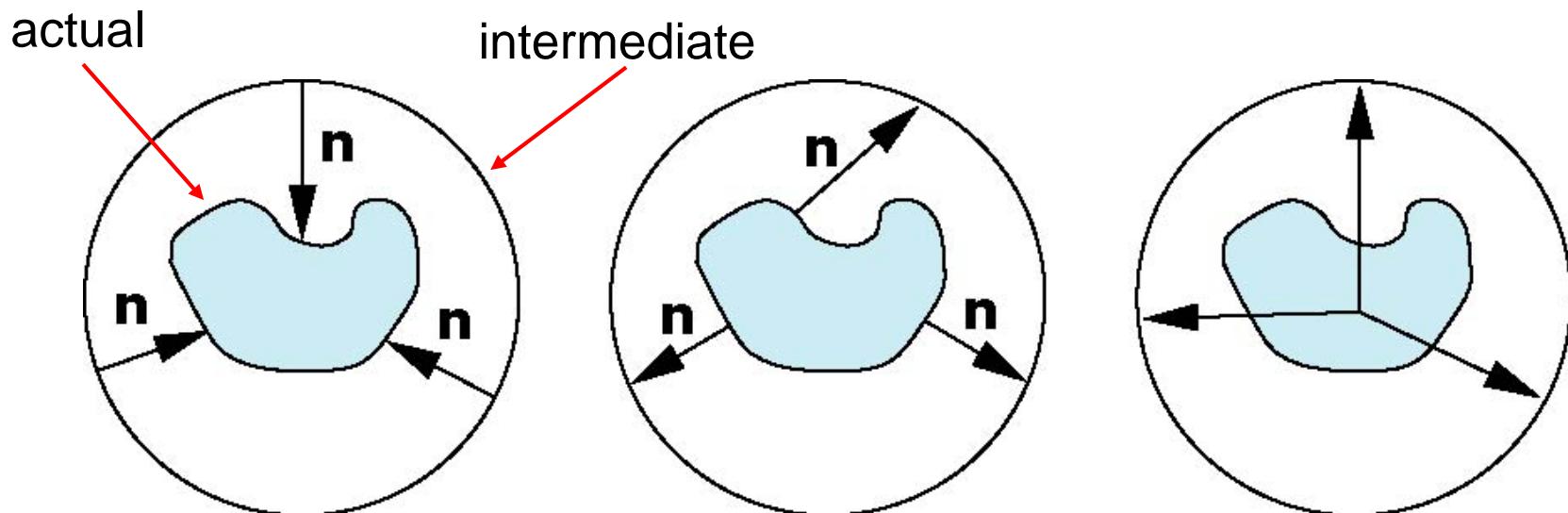
- Easy to use with simple orthographic projection
- Also used in environmental maps



# Second mapping

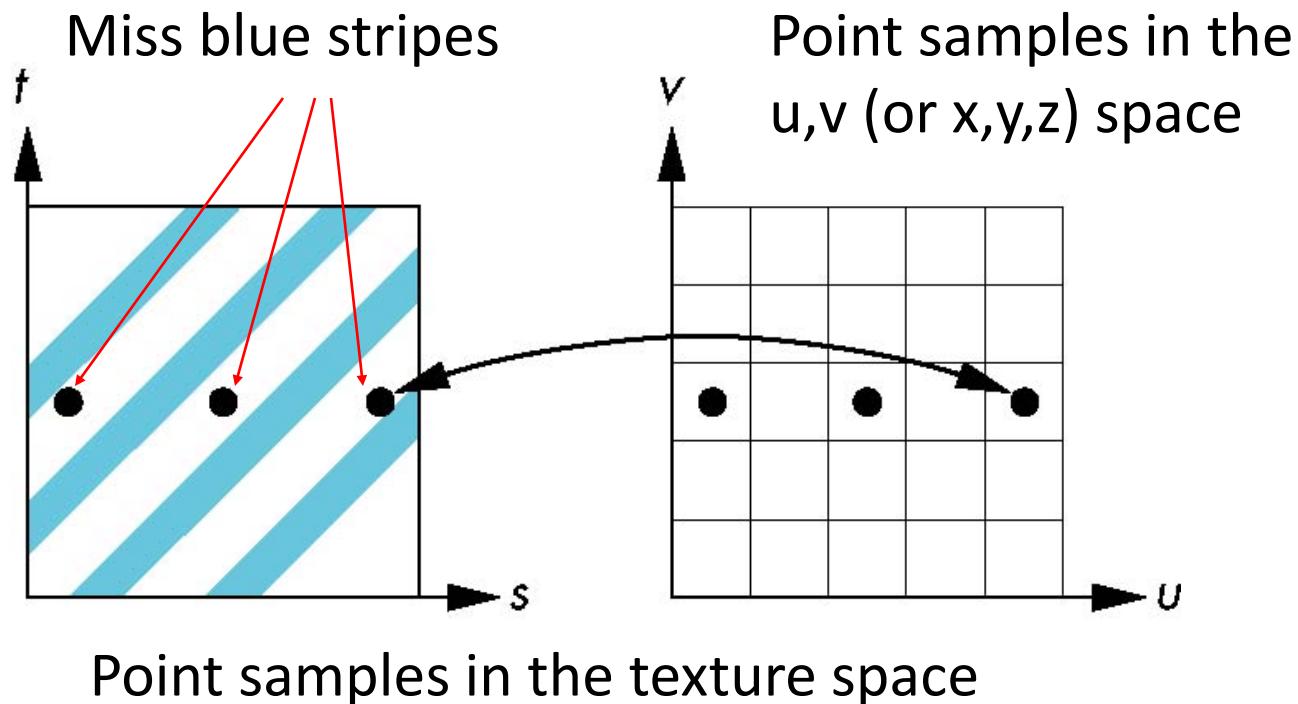
Mapping from an intermediate object to an actual object

- Normals from the intermediate to actual objects
- Normals from the actual to intermediate objects
- Vectors from the centre of the intermediate object



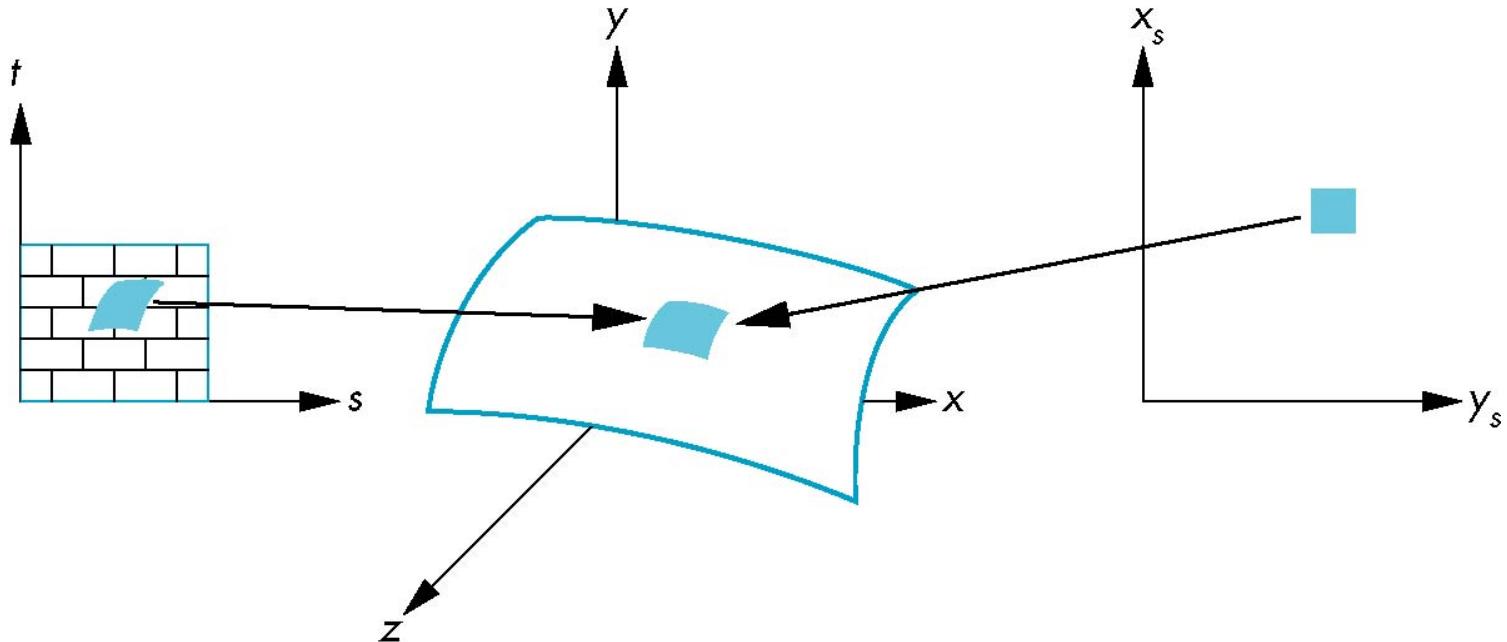
# Aliasing

- Point sampling of the texture can lead to aliasing errors



# Area averaging

A better but slower option is to use *area averaging*.



Note that the *pre-image* of the pixel is curved.

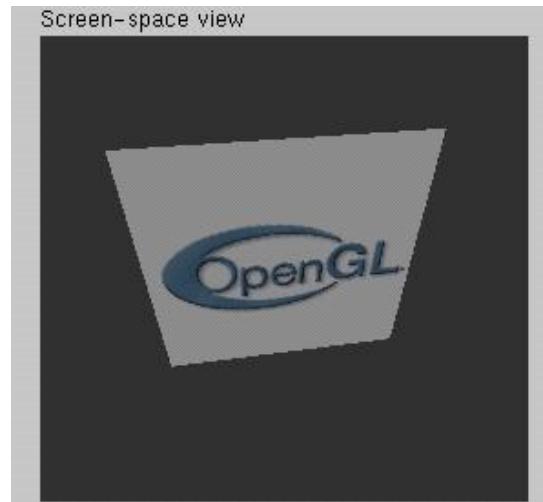
# OpenGL steps in texture mapping

Three steps to applying a texture

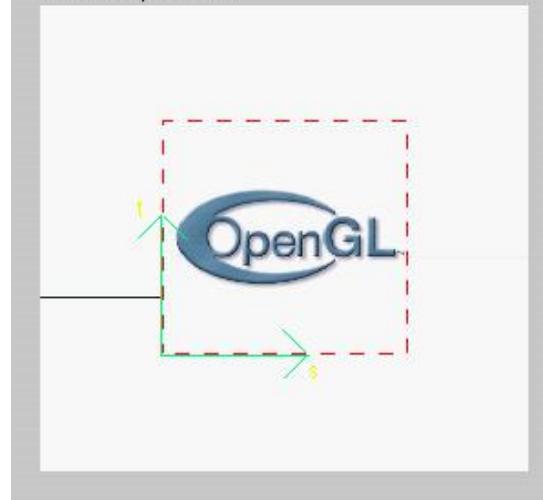
1. Specify the texture
  - read or generate image
  - assign to texture
  - enable texturing
2. Assign texture coordinates to vertices  
Proper mapping function is left to application
3. Specify texture parameters
  - mode
  - filtering
  - wrapping

# Texture example

The texture is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.

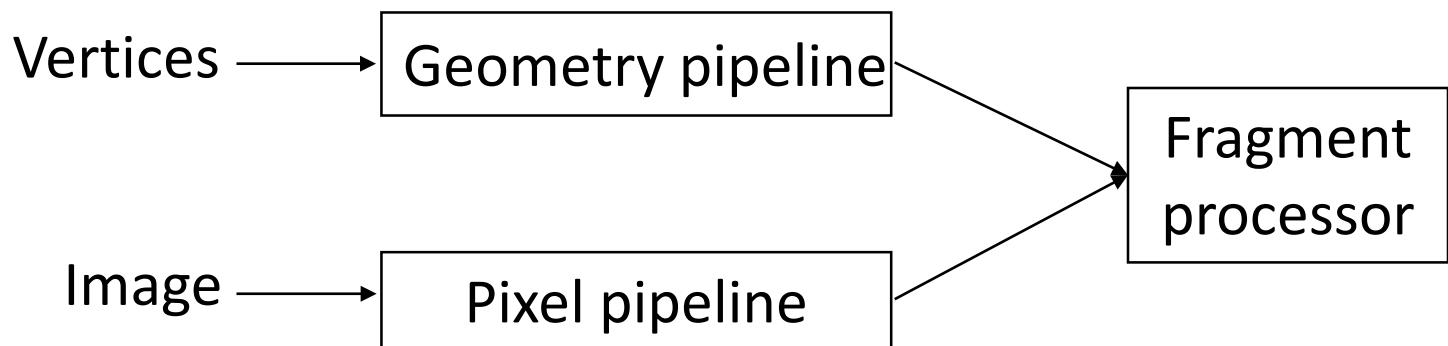


Screen-space view



# Texture mapping and the OpenGL pipeline

- The images and geometry flow through separate pipelines that join during fragment processing.
- “Complex” textures do not affect geometric complexity.



# Specifying a texture image

- Define a texture image from an array of *texels* (texture elements) in CPU memory
  - `Glubyte my_texels[512][512][4];`
- Define a texture as any other pixel map
  - Scanned image
  - Generate by an application program
- Enable texture mapping
  - `gLEnable(GL_TEXTURE_2D)`
  - OpenGL supports 1-4 dimensional texture maps

# Defining image as a texture (1)

```
glTexImage2D(target, level, components,  
             w, h, border, format, type, texels);
```

**target:** type of texture, e.g. `GL_TEXTURE_2D`

**level:** used for mipmapping (0 for only one/top resolution – to be discussed shortly)

**components:** colour elements per texel - an integer from 1 to 4 indicating which of the R, G, B and A components are selected for modulating or blending. A value of 1 selects the R component, 2 selects the R and A components, 3 selects R, G and B, and 4 selects R, B, G and A.

# Defining image as a texture (2)

**w** and **h**: width and height of the image in pixels.

**border**: used for smoothing (discussed shortly), which is usually 0.

Both **w** and **h** must have the form  $2^m + 2b$ , where **m** is an integer and **b** is the value of **border** though they can have different values. The maximum size of a texture map depends on the implementation of OpenGL, but it must be at least  $64 \times 64$  (or  $66 \times 66$  with borders).

# Defining image as a texture (3)

**format** and **type**: describe the format and data type of the texture image data. They have the same meaning with `glDrawPixels()`. In fact, texture data has the same format as the data used by `glDrawPixels()`.

The **format** parameter can be `GL_COLOR_INDEX`, `GL_RGB`, `GL_RGBA`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA` – i.e., the same formats available for `glDrawPixels()` with the exceptions of `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT`.

Similarly, the **type** parameter can be `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, or `GL_BITMAP`.

# Defining image as a texture (4)

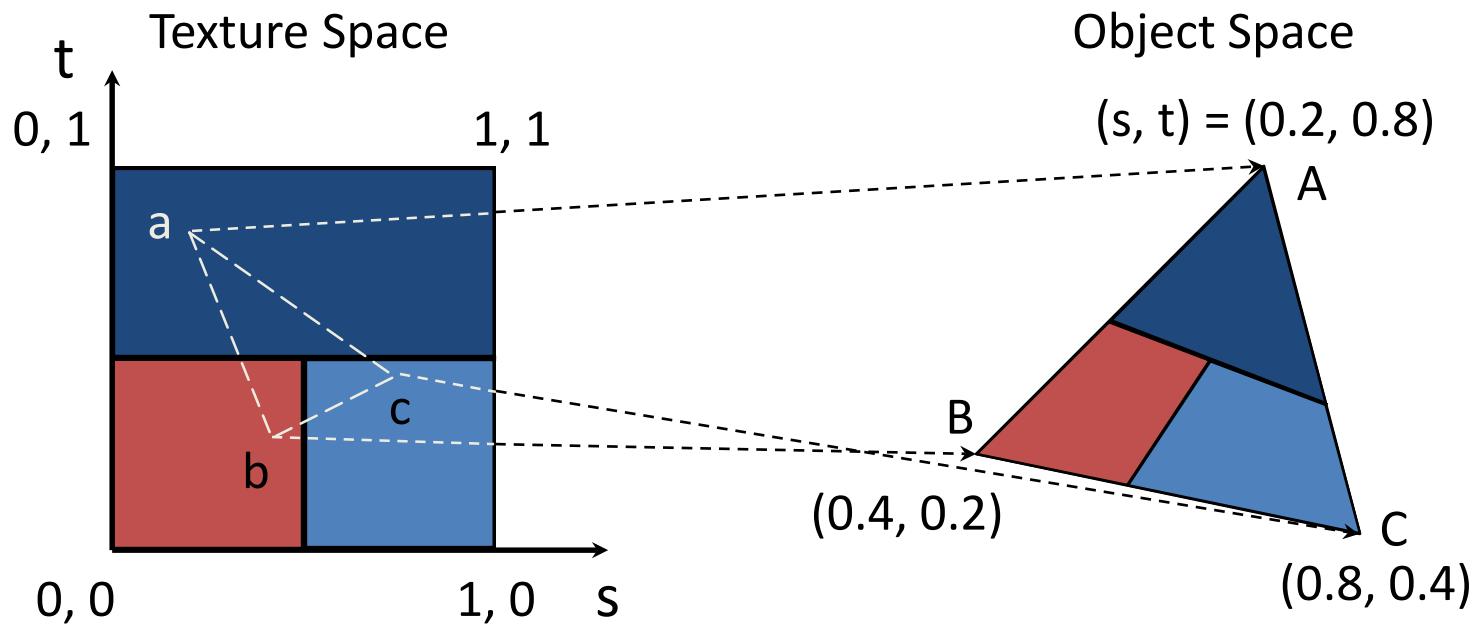
**texels**: pointer to the texel array which contains the texture-image data. This data describes the texture image itself as well as its border.

For example:

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 514, 514, 1,  
 GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

# Mapping a texture

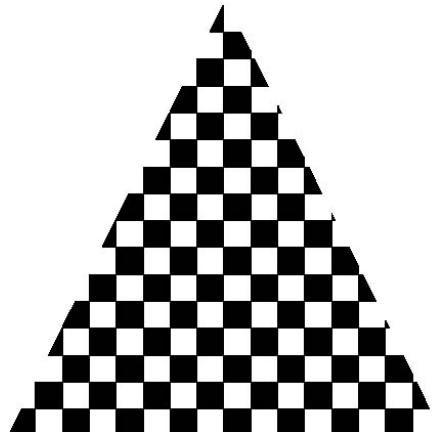
Based on parametric texture co-ordinates,  
`glTexCoord*`( ) is specified at each vertex.



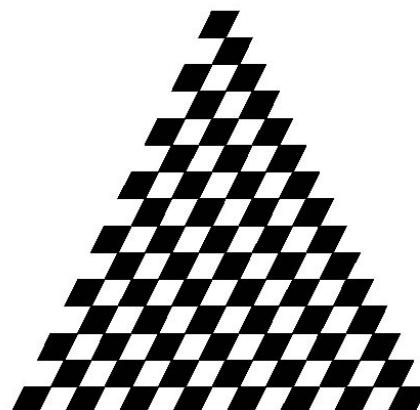
# Interpolation

- OpenGL uses interpolation to find proper texels from specified texture coordinates.
- There can be distortions.

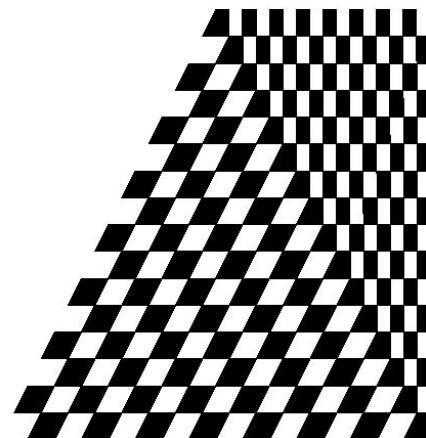
Good selection  
of texture co-ordinates



Poor selection  
of texture co-ordinates



Texture stretched  
over trapezoid  
showing effects of  
bilinear interpolation



# Texture parameters

OpenGL has a variety of parameters that determine how texture is applied:

- *Wrapping parameters* determine what happens if s and t are outside the [0,1] range.
- *Filter modes* allow us to use area averaging instead of point samples.
- *Mipmapping* (discussed shortly) allows us to use textures of multiple resolutions.
- *Mode/environment parameters* determine how texture mapping interacts with shading.

# Repeating and clamping textures (1)

- We can assign texture co-ordinates outside the range  $[0,1]$  and have them either clamp or repeat in the texture map.
- With repeating textures, if we have a large plane with texture co-ordinates running from 0.0 to 10.0 in both directions, for example, we will get 100 copies of the texture tiled together on the screen. During repeating, the integer part of texture co-ordinates is ignored, and copies of the texture map tile the surface.
- For most applications of repeating, the texels at the top of the texture should match those at the bottom, and similarly for the left and right edges.

# Repeating and clamping textures (2)

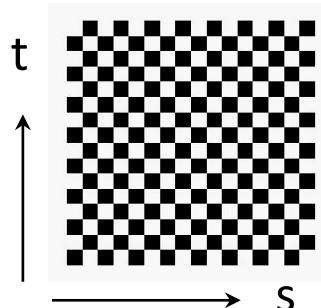
- The other possibility is to clamp the texture coordinates: any values greater than 1.0 are set to 1.0, and any values less than 0.0 are set to 0.0.
- Clamping is useful when a single copy of the texture is to appear on a large surface. If the surface-texture coordinates range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower corner of the surface. The rest of the surface is painted with the texture border colours as needed.

# Repeating and clamping textures (4)

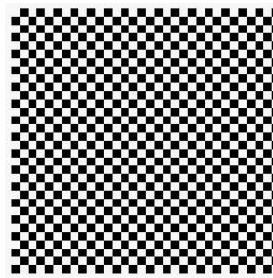
Clamping: if  $s, t > 1$  use 1, if  $s, t < 0$  use 0

Wrapping: use  $s, t$  modulo 1

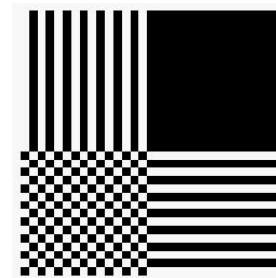
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  
                GL_CLAMP)  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  
                GL_REPEAT)
```



Texture



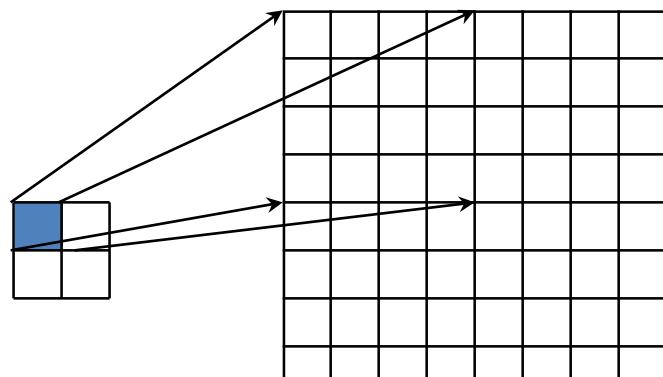
GL\_REPEAT  
wrapping



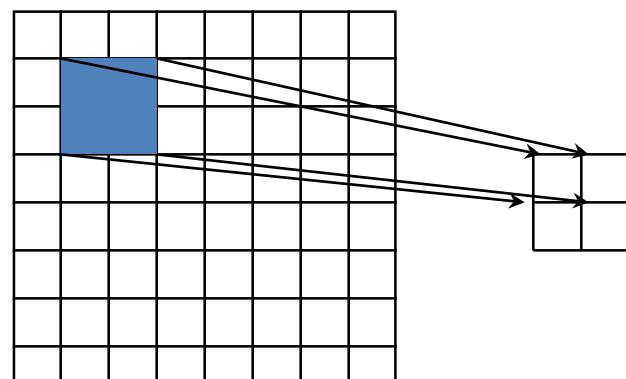
GL\_CLAMP  
wrapping

# Minification and magnification

- More than one pixel can cover a texel (*magnification*) or more than one texel can cover a pixel (minification).
- We can use point sampling (nearest texel) or linear filtering to obtain texture values.



Texture                          Polygon  
**Magnification**



Texture                          Polygon  
**Minification**

# Controlling filtering (1)

- OpenGL allows us to specify any of several filtering options to determine the calculations. The options provide different trade-offs between speed and image quality. We can specify the filtering methods for magnification and minification independently.
- OpenGL can make a choice between magnification and minification that in most cases gives the best result possible. The following lines are examples of how to use `glTexParameter*` to specify the magnification and minification filtering methods:

```
glTexParameteri(GL_TEXTURE_2D,  
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D,  
                 GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

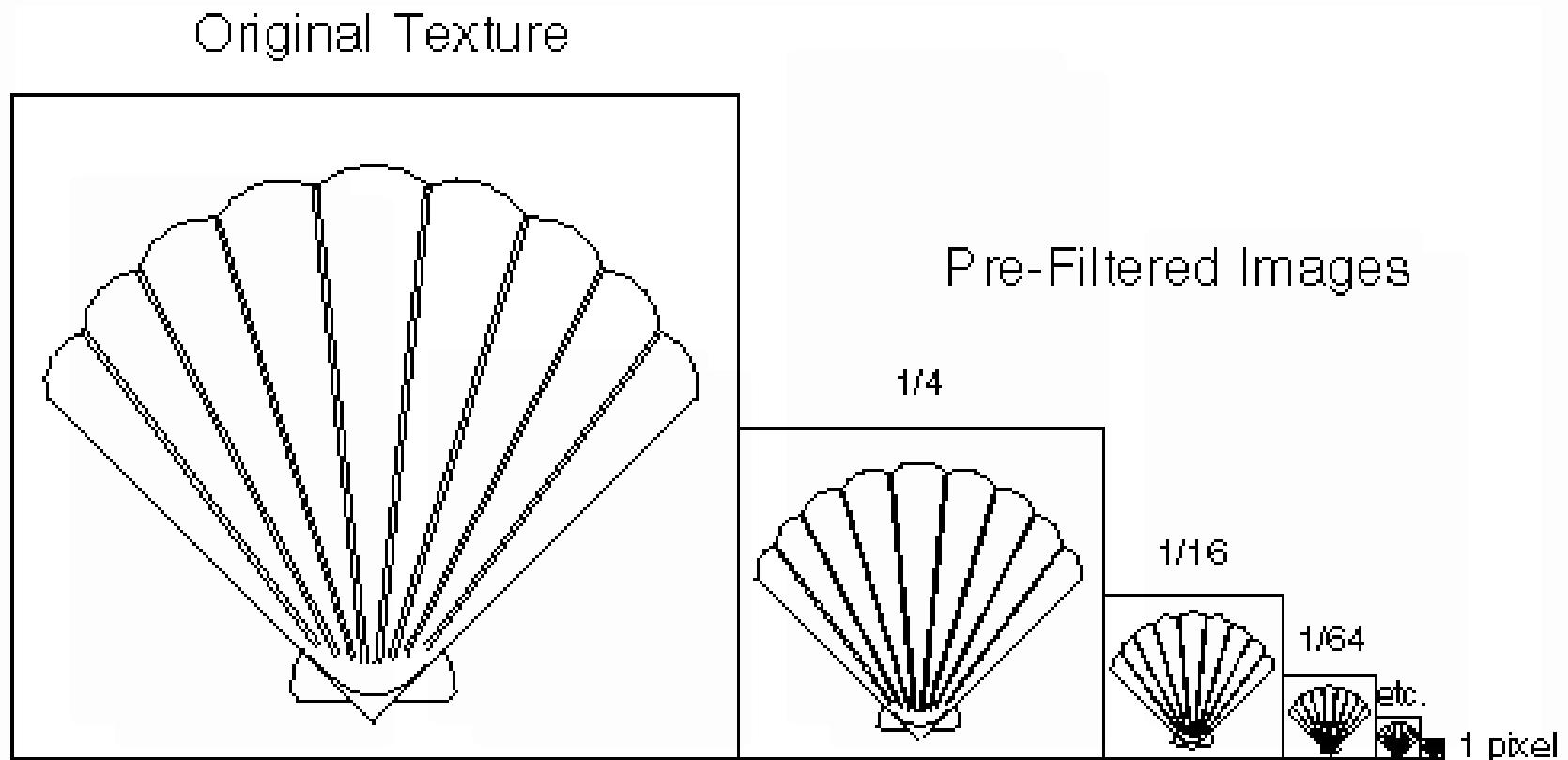
# Controlling filtering (2)

- The second argument is either `GL_TEXTURE_MAG_FILTER` or `GL_TEXTURE_MIN_FILTER` to indicate whether we are specifying the filtering method for magnification or minification. The third argument specifies the filtering method.
- Note that linear filtering requires a border of an extra texel for filtering at edges (`border = 1`).

# Multiple levels of detail (1)

- Textured objects can be viewed, like any other objects in a scene, at different distances from the viewpoint. In a dynamic scene, as a textured object moves further from the viewpoint, the texture map must decrease in size along with the size of the projected image.
- To accomplish this, OpenGL has to filter the texture map down to an appropriate size for mapping onto the object, without introducing visually disturbing artefact.
- To avoid such artefact, we can specify a series of pre-filtered texture maps of decreasing resolutions, called *mipmaps*, as shown on the next slide.

# Multiple levels of detail (2)



# Multiple levels of detail (3)

- OpenGL automatically determines which texture map to use based on the size (in pixels) of the object being mapped.
- With this approach, the level of detail in the texture map is appropriate for the image that is drawn on the screen - as the image of the object gets smaller, the size of the texture map decreases.
- Mipmapping requires some extra computation to reduce interpolation errors, but, when it is not used, textures that are mapped onto smaller objects might shimmer and flash as the objects move.

# Multiple levels of detail (4)

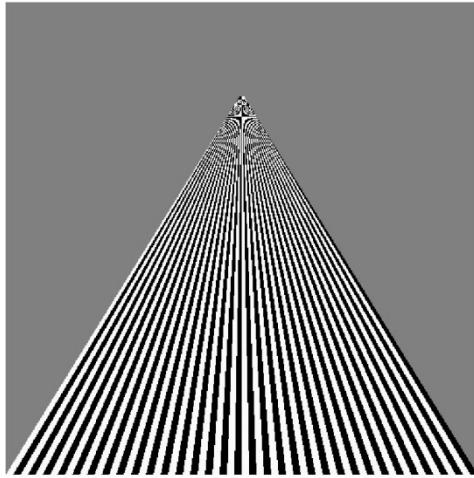
- To use mipmapping, we provide all sizes of the texture in powers of 2 between the largest size and a  $1 \times 1$  map.
- For example, if the highest-resolution map is  $64 \times 16$ , we must also provide maps of size  $32 \times 8$ ,  $16 \times 4$ ,  $8 \times 2$ ,  $4 \times 1$ ,  $2 \times 1$  and  $1 \times 1$ .
- The smaller maps are typically filtered and averaged-down versions of the largest map in which each texel in a smaller texture is an average of the corresponding four texels in the larger texture.
- OpenGL does not require any particular method for calculating the smaller maps, so the differently sized textures could be totally unrelated.

# Multiple levels of detail (5)

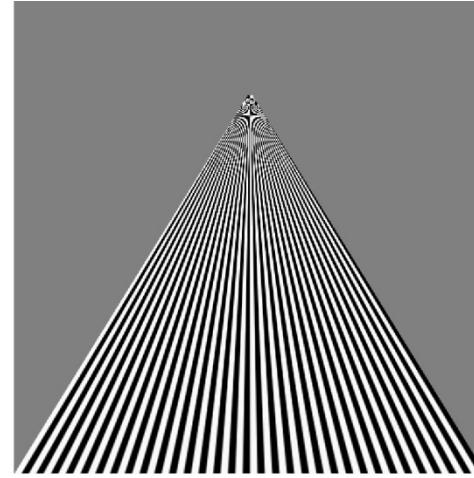
- Mipmapping level is declared during texture definition by calling `glTexImage2D(GL_TEXTURE_*D, level, ...)` once for each resolution of the texture map, with different values for the `level`, `width`, `height` and `image` parameters.
- Starting with zero, `level` identifies which texture in the series is specified; with the previous example, the largest texture of size  $64 \times 16$  would be declared with `level = 0`, the  $32 \times 8$  texture with `level = 1`, and so on.
- In addition, for the mipmapped textures to take effect, we need to choose one of the appropriate filtering methods.

# Mipmapped texture - example

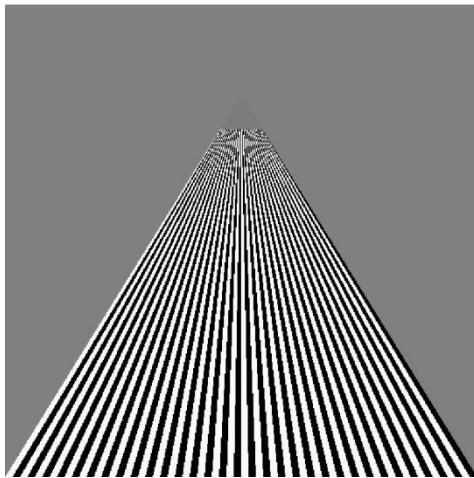
Point sampling



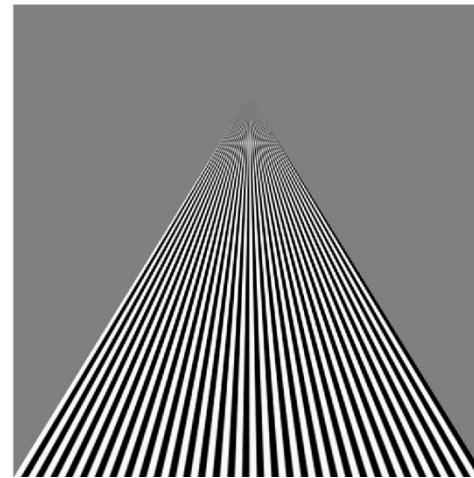
Linear filtering



Mipmapped point sampling



Mipmapped linear filtering



# Texture mapping functions

- Control how texture is applied

```
glTexEnv{fi}[v](GL_TEXTURE_ENV, prop, param)
```

- **GL\_TEXTURE\_ENV\_MODE** modes

**GL\_MODULATE:** modulates with computed shade

**GL\_BLEND:** blends with an environmental colour

**GL\_REPLACE:** use only texture colour

- Set blend colour with **GL\_TEXTURE\_ENV\_COLOR**

# Modulating and blending (1)

We can choose one of three possible functions for computing the final RGBA value from the fragment colour and the texture-image data.

- simply use the texture colour as the final colour, called the *decal* mode, in which the texture is painted on top of the fragment, just as a decal would be applied.
- use the texture to *modulate*, or scale, the fragment colour, useful for combining the effects of lighting with texturing.
- a constant colour can be blended with the fragment colour, based on the texture value.

# Modulating and blending (2)

- The values in the texture map can be used directly as colours to be painted on the surface being rendered. We can also use the values in the texture map to modulate the colour that the surface would be painted without texturing, or to blend the colour in the texture map with the non-textured colour of the surface.
- We can choose one of these three texturing functions by supplying the appropriate arguments to `glTexEnv{if}{v}(GLenum target, GLenum pname, TYPEparam)`, which sets the current texturing function.

# Modulating and blending (3)

- The **target** must be **GL\_TEXTURE\_ENV**. If **pname** is **GL\_TEXTURE\_ENV\_MODE**, **TYPEparam** can be **GL\_DECAL**, **GL\_MODULATE**, or **GL\_BLEND**, to specify how texture values are to be combined with the colour values of the fragment being processed. In the decal mode and with a three-component texture, the texture colours replace the fragment colours.
- With either of the other two modes or with a four-component texture, the final colour is a combination of the texture and the fragment values. If **pname** is **GL\_TEXTURE\_ENV\_COLOR**, **TYPEparam** is an array of four floating-point values representing the R, G, B and A components. These values are used only if the **GL\_BLEND** texture function is specified as well.

# Perspective correction hint

- Texture co-ordinate and colour interpolation
  - either linearly in the screen space
  - or using depth/perspective values (slower)
- Noticeable for polygons “on edge”

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, hint)
```

where **hint** is one of

**GL\_DONT\_CARE**

**GL\_NICEST**

**GL\_FASTEST**

# Generating texture co-ordinates (1)

- Texture co-ordinates are usually referred to as the  $s$ ,  $t$ ,  $r$  and  $q$  co-ordinates to distinguish them from object co-ordinates ( $x$ ,  $y$ ,  $z$  and  $w$ ). For 2D textures,  $s$  and  $t$  are used. Currently, the  $r$  coordinate is ignored (although it might have meaning in the future). The  $q$  coordinate, like  $w$ , is typically given the value 1 and can be used to create homogeneous co-ordinates.
- We need to indicate how the texture should be aligned relative to the fragments to which it is to be applied. We can specify both texture co-ordinates and geometric co-ordinates as we specify the objects in the scene.
- The command to specify texture co-ordinates, `glTexCoord*`( ), is similar to `glVertex*`( ), `glColor*`( ) and `glNormal*`( ) - it comes in similar variations and is used in the same way between `glBegin()` and `glEnd()` pairs.

# Generating texture co-ordinates (2)

- Usually, texture co-ordinate values range between 0 and 1.
- `void glTexCoord{1234}{sifd}[v](TYPEcoords)` sets the current texture co-ordinates ( $s, t, r, q$ ).
- Subsequent calls to `glvertex*`( ) result in those vertices being assigned the current texture co-ordinates.
- Using `glTexCoord2*`( ) allows us to specify  $s$  and  $t$ ;  $r$  and  $q$  are set to 0 and 1, respectively. We can supply the co-ordinates individually, or we can use the vector version of command to supply them in a single array. Texture co-ordinates are multiplied by the 4x4 texture matrix before any texture mapping occurs.

# Generating texture co-ordinates (3)

- OpenGL can generate texture co-ordinates automatically

`glTexGen{ifd}[v]()`

- Specify a plane  
generates texture coordinates based on the distance from the plane

- Generation modes

`GL_OBJECT_LINEAR`

`GL_EYE_LINEAR`

`GL_SPHERE_MAP` (used for environmental maps)

# Texture objects

- Texture is part of the OpenGL state
  - If we have different textures for different objects, OpenGL will be moving large amounts of data from the processor memory to the texture memory.
- Recent versions of OpenGL have *texture objects*
  - One image per texture object.
  - The texture memory can hold multiple texture objects.

# Applying textures

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex (coordinates can also be generated)

# OpenGL functions

- **glTexImage2D( )**: Defining image as texture (type, level and colour)
- **glTexParameter\***( ): Specifying filtering and wrapping
- **glTexEnv{fi }[v ]( )**: Specifying modulating, blending or decal
- **glHint( )**: Defining perspective correction hint
- **glBindTexture( )**: Binding a named texture to a texturing target
- **glTexCoord{1234}{sifd}{v}( )**: Specifying texture coordinates s and t while r is set to 0 and q to 1
- **glTexGen{ifd}[v]( )**: Automatic generation of texture coordinates by OpenGL

# Summary

- Why texture mapping is needed and how it works
- Techniques for texture mapping
  - Specifying the texture
  - Magnification and minification
  - Multiple levels of detail (mipmapping)
  - Modes and filtering
  - Wrapping
- Steps for texture mapping
- OpenGL functions



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Clipping**

**Week 12**  
**2021-22**

**Yong Yue**

# Topics for today

## ➤ Concepts

What is clipping?

Why is clipping important?

Clipping of points, lines and polygons

## ➤ Line clipping algorithms

Brute Force Simultaneous Equations

Brute Force Similar Triangles

Cohen-Sutherland

Liang-Barsky

## ➤ Polygon clipping

## ➤ OpenGL functions

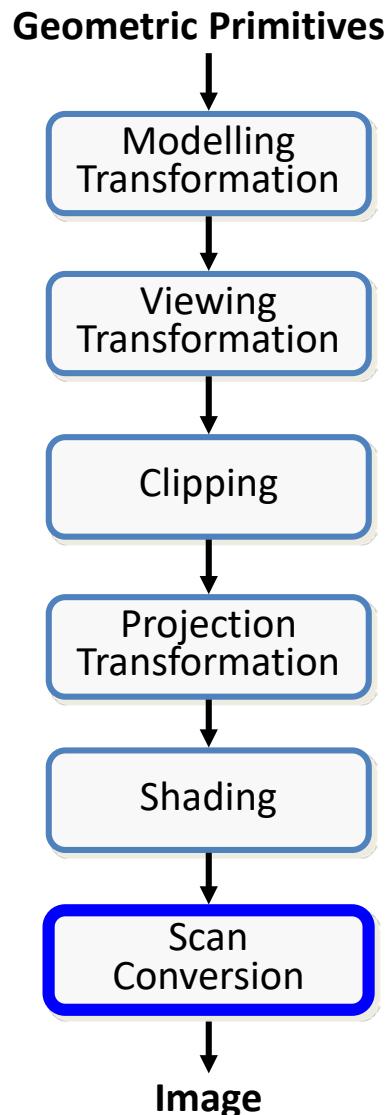
# Clipping and rasterization

- **Clipping** – Remove objects or parts of objects that are outside the clipping window.
- **Rasterization (scan conversion)** – Convert high level object descriptions to pixel colours in the framebuffer.
- Graphics systems (e.g. OpenGL) do these for us – no explicit OpenGL functions needed for doing clipping and rasterization.

# Why clipping

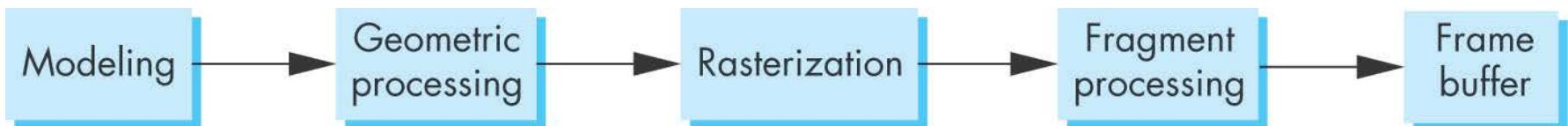
- Rasterization is very expensive
  - Approximately linear with the number of fragments created
  - Math and logic *per pixel*
- If we only rasterize what is actually viewable, we can save a lot of expensive computation
  - A few operations now can save many later

# Rendering pipeline



# Required tasks

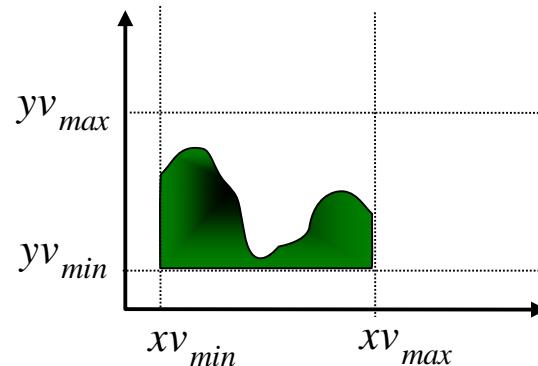
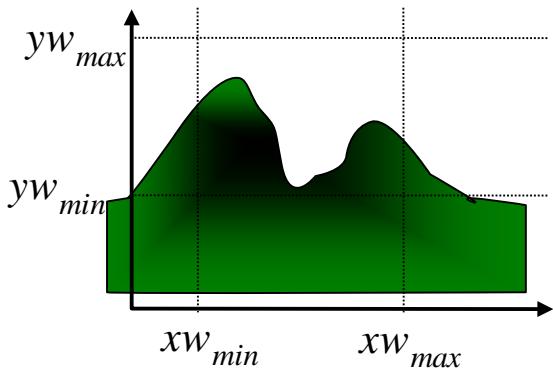
- Clipping
- Transformations
- Rasterization (scan conversion)
- Some tasks deferred until fragment processing
  - Hidden surface removal
  - Antialiasing



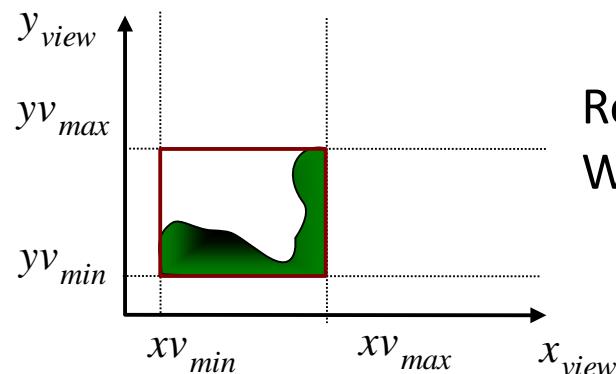
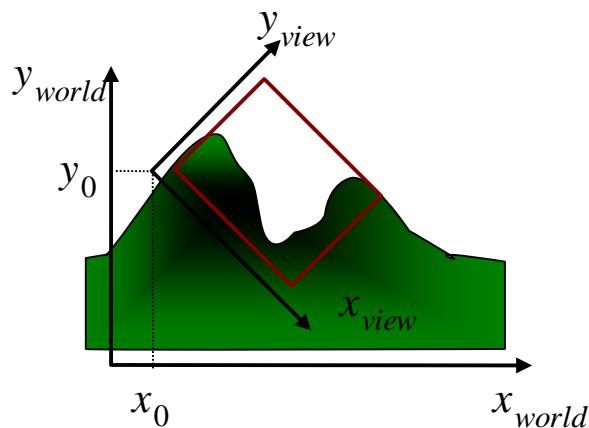
# Rasterization meta algorithms

- Consider two approaches to rendering a scene with opaque objects
- For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel
  - Ray tracing paradigm
- For every object, determine which pixels it covers and shade these pixels
  - Pipeline approach
  - Must keep track of depths

# The clipping window



Rectangular Window



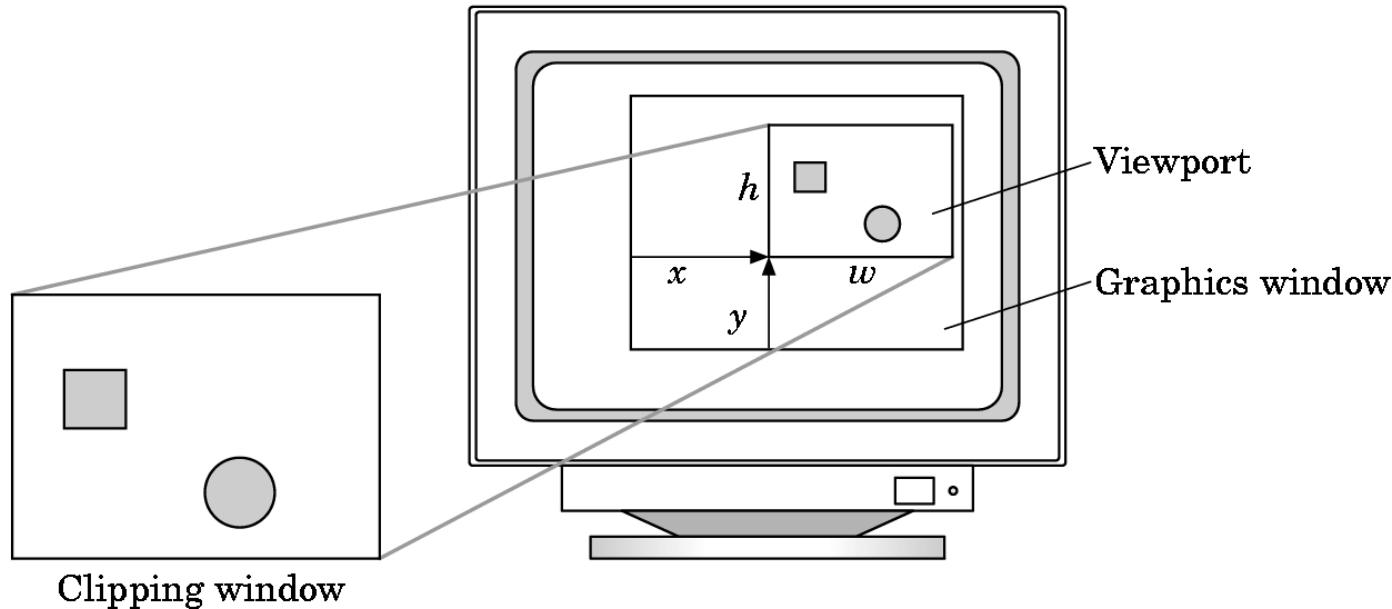
Rotated Window

# Clipping window vs viewport

- The clipping window selects *what* we want to see in our virtual 2D world.
- The *viewport* indicates where it is to be viewed on the output device (or within the display window).
- By default the *viewport* has the same location and dimensions of the GLUT display window we create (see next slide).

# Clipping window and viewport

```
void glviewport(GLint x, GLint y, GLsizei w, GLsizei h)
```

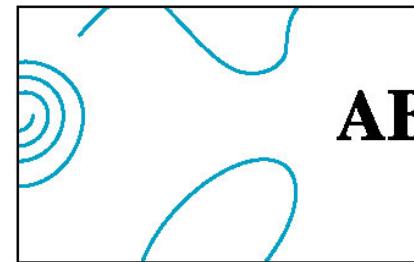
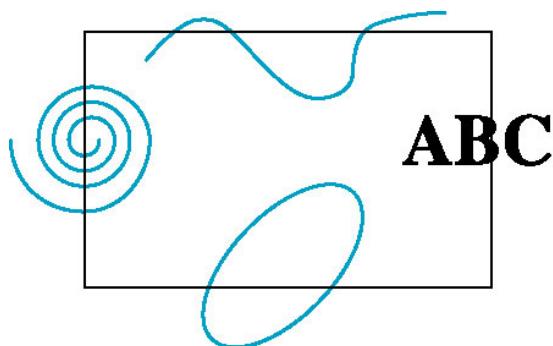


The viewport is part of the state

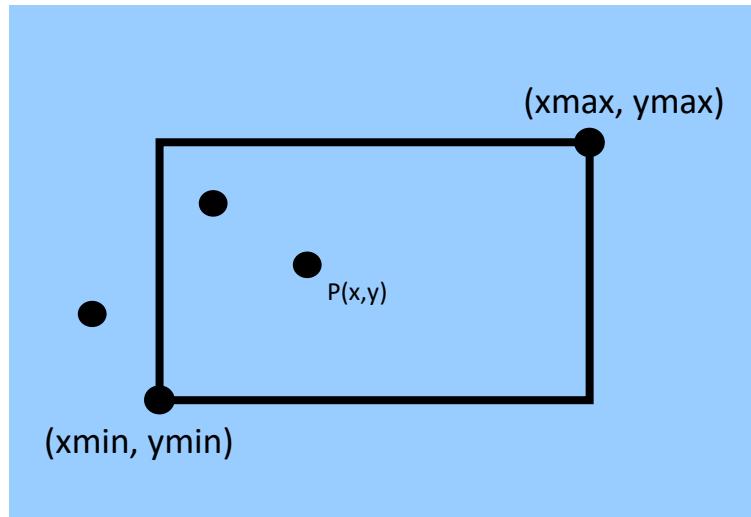
- changes between rendering objects or redisplay
- different window-viewport transformations used to make the scene

# Clipping primitives

- Different primitives can be handled in different ways
  - Points
  - Lines
  - Polygons
- 2D against clipping window
- 3D against clipping volume
- Easy for line segments and polygons
- Hard for curves and text - converted to lines and polygons first



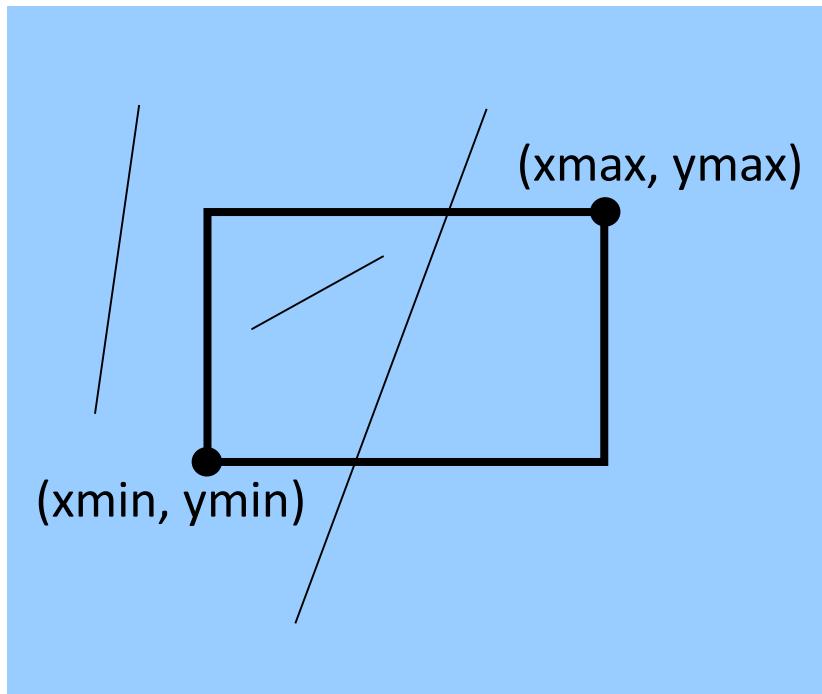
# 2D point clipping



Determine whether a point  $(x, y)$  is inside or outside of the window.

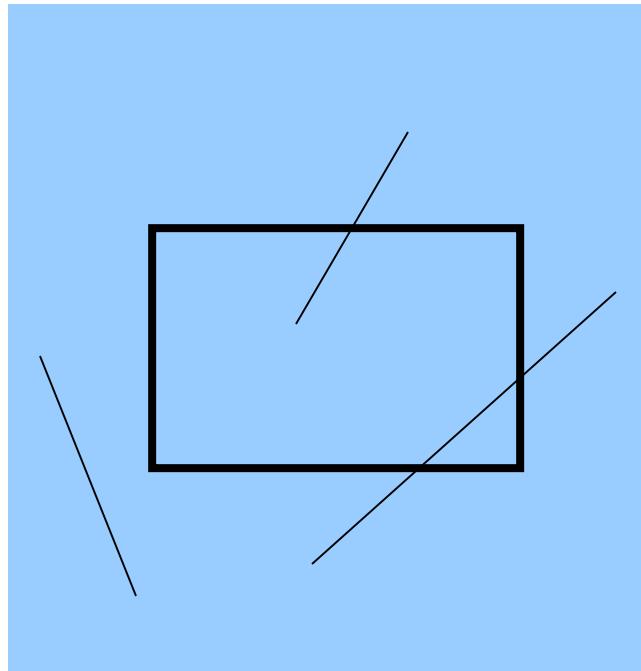
If  $(x_{\min} \leq x \leq x_{\max})$  and  $(y_{\min} \leq y \leq y_{\max})$   
the point  $(x, y)$  is inside  
else  
the point  $(x, y)$  is outside

# 2D line clipping



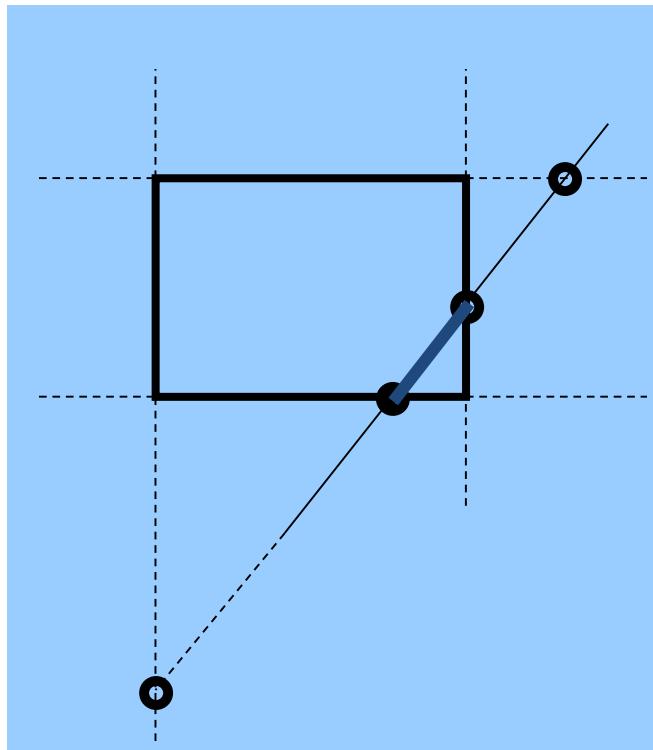
- Determine whether a line is inside, outside or partially inside the window.
- If a line is partially inside, we need to display the inside segment.

# 2D line clipping – non-trivial cases



- Lines that cannot be trivially rejected (i.e. entirely outside one of the 4 clipping window edges) or trivially accepted (i.e. entirely within the clipping window):
  - One point inside, and one point outside;
  - Both points are outside, but not “trivially” outside.
- Need to find the line segments that are inside.

# 2D line clipping – non-trivial clipping

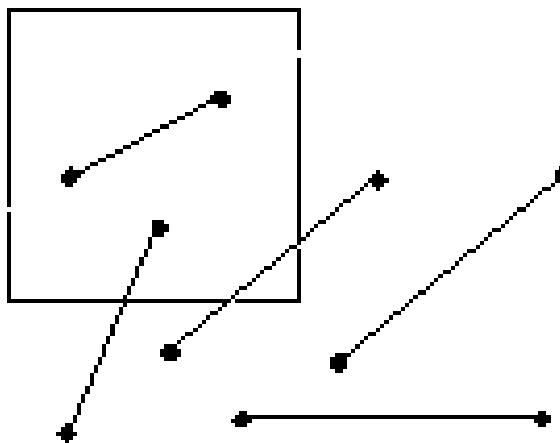


- Compute the line-window boundary edge intersection.
- There will be four intersections, but only one or two are on the window edges.
- These two points are the end points of the desired line segment.

# Brute force clipping of lines (Simultaneous equations)

Brute force clipping: **solve simultaneous equations** using  $y = mx + b$  for line and four clip edges

- Slope-intercept formula handles infinite lines only
- Does not handle vertical lines



# Brute force clipping of lines (Similar triangles)

Clip a line against 1 edge of the window, e.g. the top edge, so we need to find out  $x'$  and  $y'$  (which is  $y_{\max}$ ).

Similar triangles

$$A / B = C / D$$

$$A = (x_2 - x_1)$$

$$B = (y_2 - y_1)$$

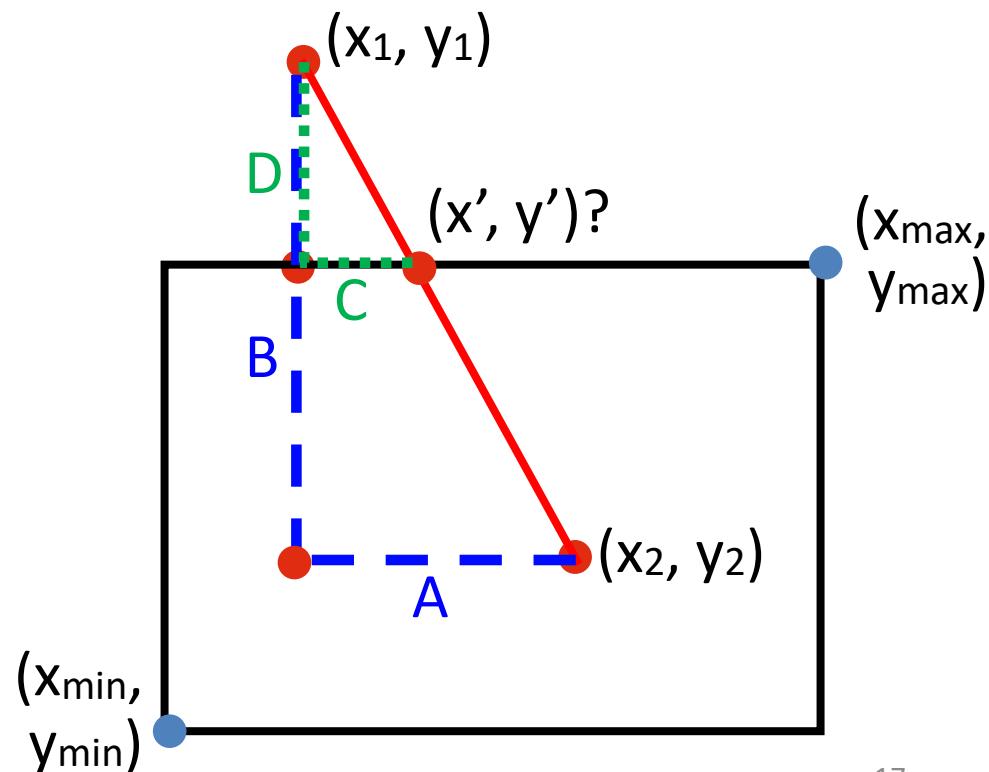
$$C = (x' - x_1)$$

$$D = (y' - y_1) = (y_{\max} - y_1)$$

$$\rightarrow C = A \bullet D / B$$

$$\rightarrow x' = x_1 + C$$

$$\rightarrow (x', y') = (x_1 + C, y_{\max})$$



# Brute force clipping of lines (Similar triangles)

➤ The problem?

Too expensive (the numbers below are for 2D)!

- 4 floating point subtractions
- 1 floating point multiplication
- 1 floating point division
- Repeat 4 times (once for each edge)

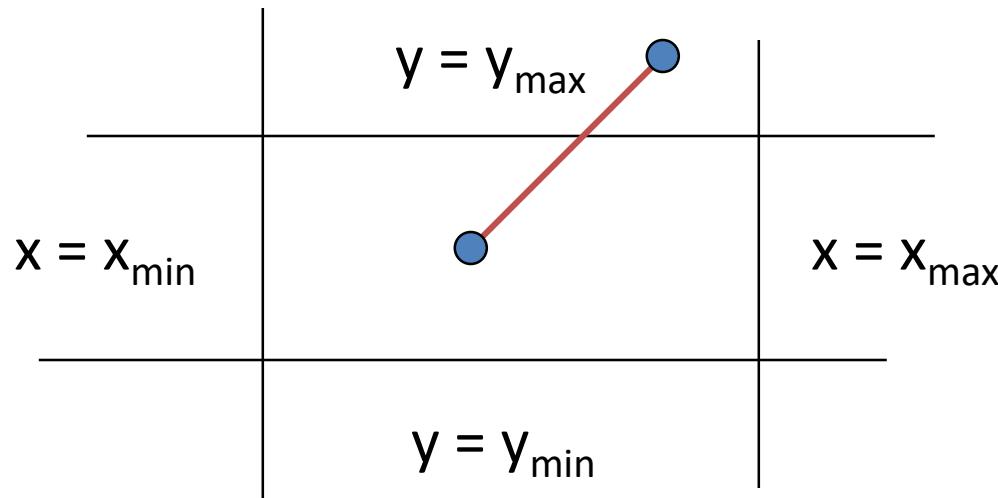
➤ We need to do better!

# Cohen-Sutherland 2D line clipping (1)

- Cohen-Sutherland
  - organised
  - efficient
  - computes new end points for the lines that must be clipped
- How it works?
  - Creates an outcode for each end point that contains location information of the point with respect to the clipping window.
  - Uses outcodes for both end points to determine the configuration of the line with respect to the clipping window.
  - Computes new end points if necessary.
  - Extends easily to 3D (using 6 faces instead of 4 edges).

# Cohen-Sutherland 2D line clipping (2)

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



# Cohen-Sutherland 2D line clipping (3)

- Case 1: both endpoints of line segment inside all four lines

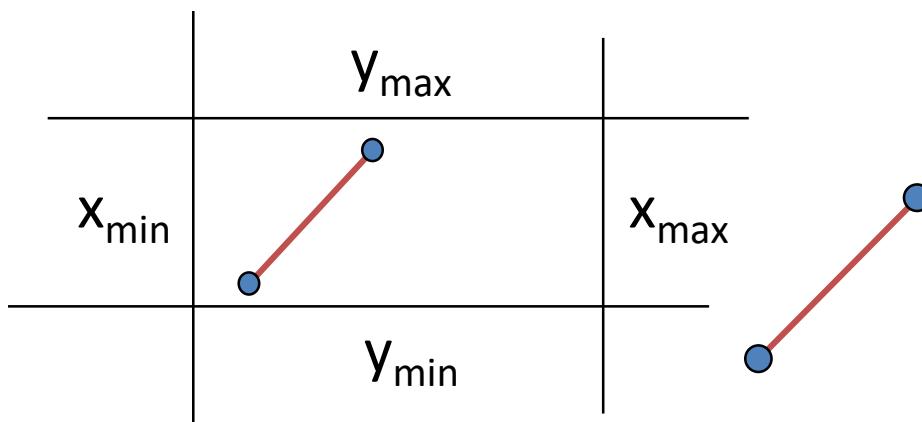
- Draw (trivial accept) line segment as is

Xmin  $\leq$  x1 and x2  $\leq$  Xmax  
Ymin  $\leq$  y1 and y2  $\leq$  Ymax

- Case 2: both endpoints outside all lines and on same side of a line

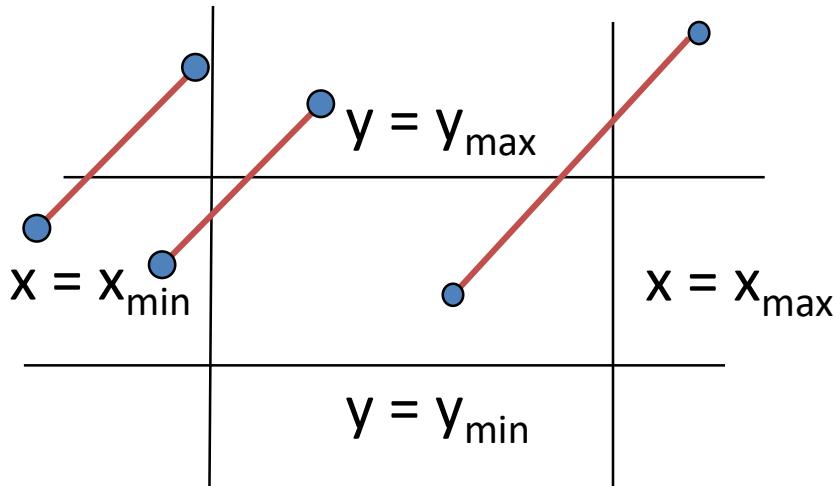
- Discard (trivial reject) the line segment

▪ x1 and x2  $<$  Xmin OR  
▪ x1 and x2  $>$  Xmax OR  
▪ y1 and y2  $<$  Ymin OR  
▪ y1 and y2  $>$  Ymax



# Cohen-Sutherland 2D line clipping (4)

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection



# Cohen-Sutherland 2D line clipping – defining outcodes (1)

- Split plane into 9 regions.
- Assign each a 4-bit outcode (above, below, right and left).

$b_0 = 1$  if  $x > x_{\min}$ , 0 otherwise

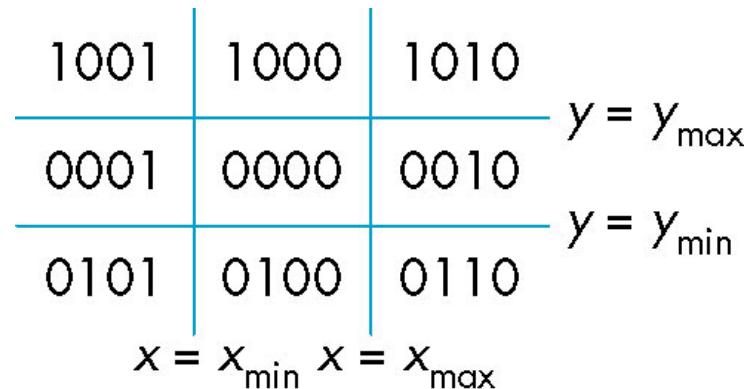
$b_1 = 1$  if  $x < x_{\max}$ , 0 otherwise

$b_2 = 1$  if  $y > y_{\min}$ , 0 otherwise

$b_3 = 1$  if  $y < y_{\max}$ , 0 otherwise

- Assign each endpoint an outcode.

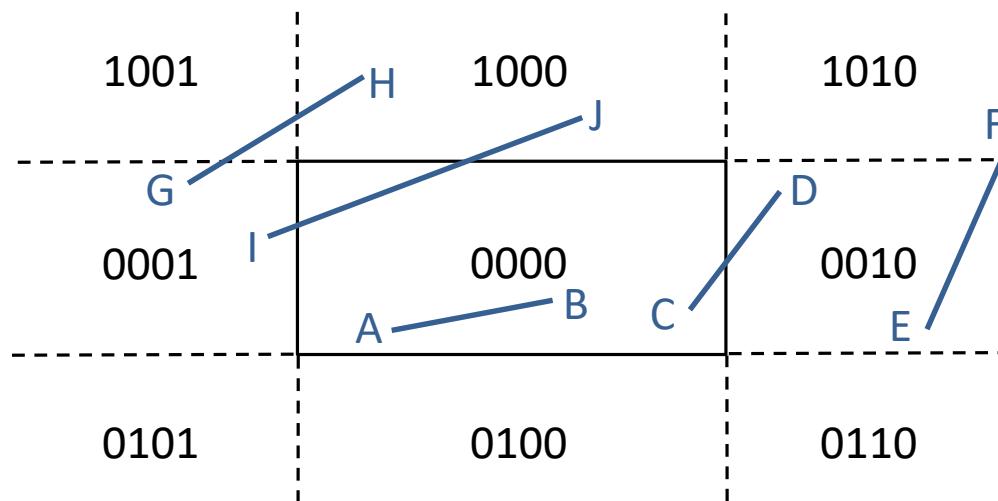
- Computation of outcode requires at most 4 subtractions.



# Cohen-Sutherland 2D line clipping – defining outcodes (2)

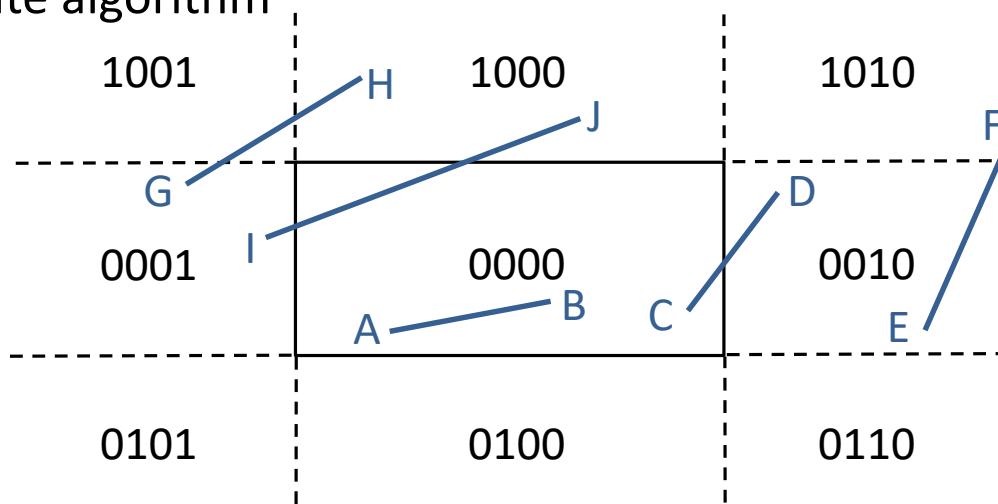
Consider the 5 cases below

- AB:  $\text{outcode}(A) = \text{outcode}(B) = 0000$ 
  - Accept line segment
- CD:  $\text{outcode}(C) = 0000$ ,  $\text{outcode}(D) = 0010 \neq 0000$ 
  - Compute intersection
  - Location of 1 in  $\text{outcode}(D)$  determines which edge to intersect with
  - Note if there were a segment from C to a point in a region with 2 ones in  $\text{outcode}$ , we might have to do two intersections



# Cohen-Sutherland 2D line clipping – defining outcodes (3)

- EF: outcode(E) logically ANDed with outcode(F) (bitwise)  $\neq 0$ 
  - Both outcodes (0010, 1010) have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - Trivial reject
- GH and IJ
  - Same outcodes (0001, 1000), neither 0 but logical AND yields 0
  - Shorten line segment by intersecting with one of window sides
  - Compute outcode of intersection (new endpoint of shortened line)
  - Re-execute algorithm

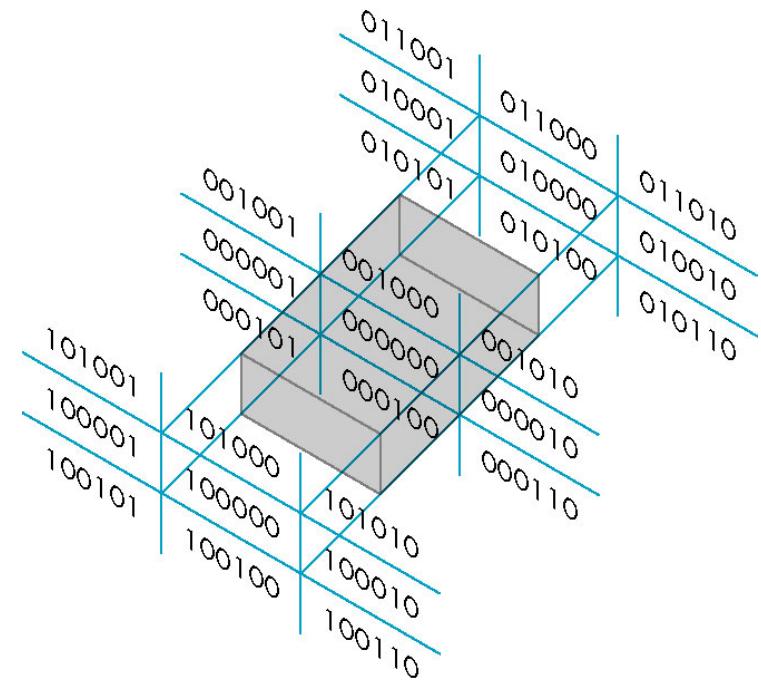
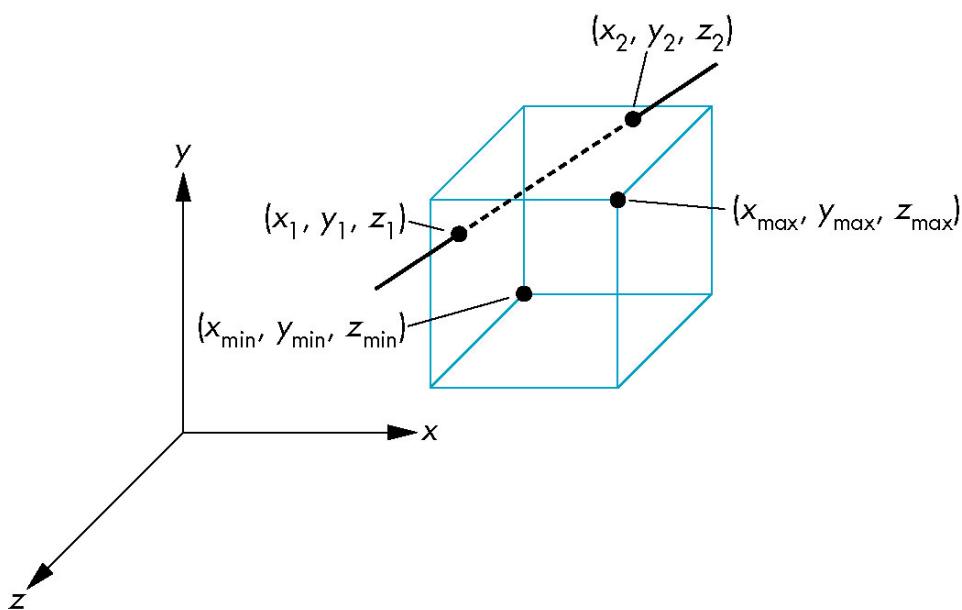


# Cohen-Sutherland 2D line clipping: Efficiency

- In many applications, the clipping window is small relative to the size of the entire database.
- Most line segments are outside one or more sides of the window and can be eliminated based on their outcodes.
- Inefficient when code has to be re-executed for line segments that must be shortened in more than one step, because two intersections have to be calculated anyway, in addition to calculating the outcode (extra computing in such a case).

# Cohen-Sutherland 3D line clipping

- Use 6-bit outcodes
- When needed, clip line segment against planes



# Liang-Barsky (Parametric line formulation)

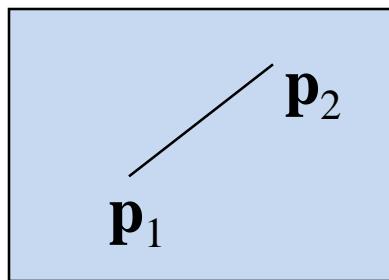
Consider the parametric form of a line segment

$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2, \quad 0 \leq \alpha \leq 1$$

or two scale expressions

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$

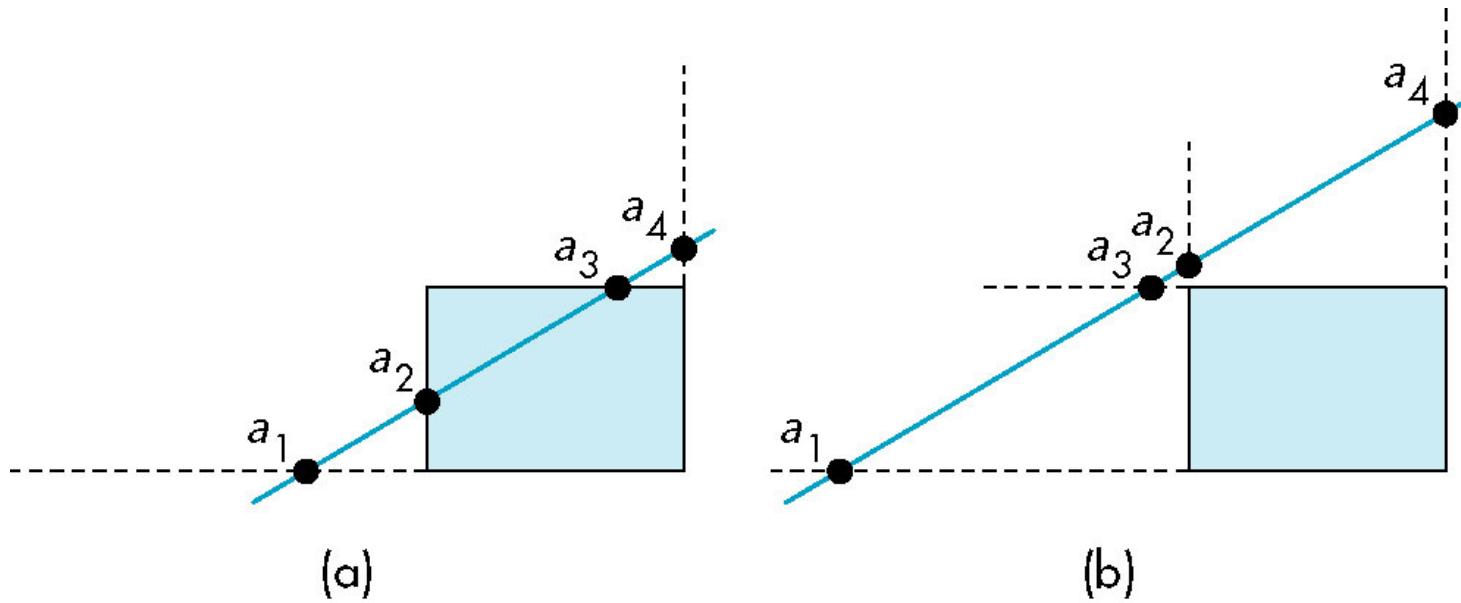
$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$



We can distinguish the cases by looking at the ordering of the  $\alpha$  values where the line is determined by the line segment crossing the lines that determine the window.

# Liang-Barsky (Parametric line formulation)

- In (a):  $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$   
Intersect right, top, left and bottom: shorten
- In (b):  $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$   
Intersect right, left, top and bottom: reject



# Liang-Barsky (Parametric line formulation)

Advantages:

- Can accept/reject as easily as with Cohen-Sutherland;
- Using values of  $\alpha$ , we do not have to use the algorithm recursively as with the Cohen-Sutherland method;
- Extends to 3D.

# Plane-line intersections

If we write the line and plane equations in matrix form (where  $\mathbf{n}$  is the normal to the plane and  $\mathbf{p}_0$  is a point on the plane), we must solve the equations

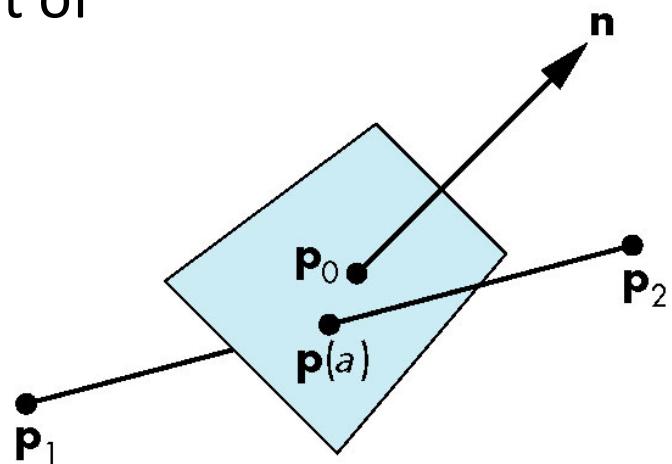
$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2, \quad 0 \leq \alpha \leq 1$$

$$\mathbf{n} \cdot (\mathbf{p}(\alpha) - \mathbf{p}_0) = 0$$

For the  $\alpha$  corresponding to the point of intersection. This value is

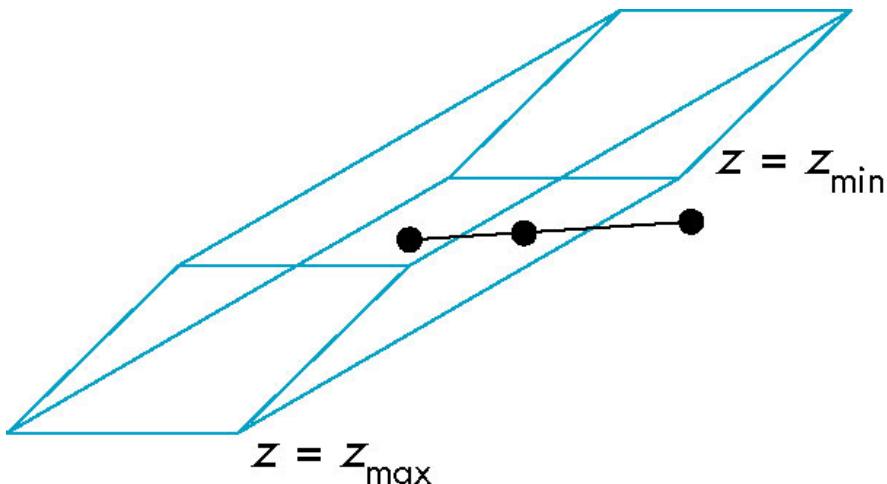
$$\alpha = \frac{\mathbf{n} \cdot (\mathbf{p}_o - \mathbf{p}_1)}{\mathbf{n} \cdot (\mathbf{p}_2 - \mathbf{p}_1)}$$

The intersection requires 6 multiplications and 1 division.



# General 3D clipping

- General clipping in 3D requires intersection of line segments against arbitrary plane.
- Example: oblique view

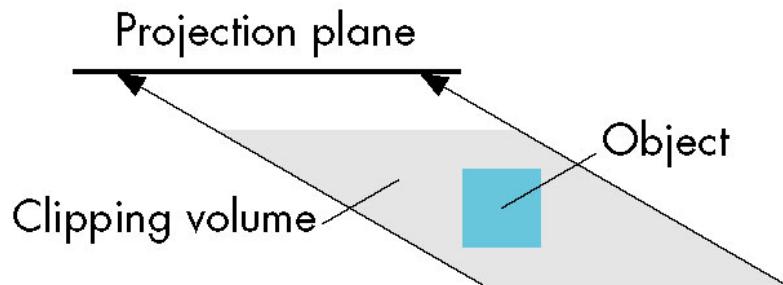


# Normalised form

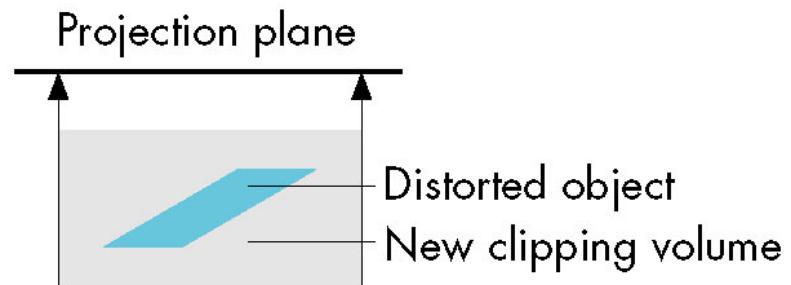
Normalisation is part of viewing (pre clipping) but after normalisation, we clip against sides of right parallelepiped.

Typical intersection calculation now requires only a floating point subtraction, e.g. is  $x > x_{\max}$  ?

Top view



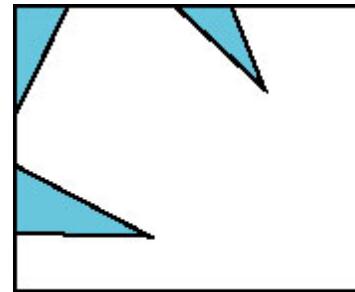
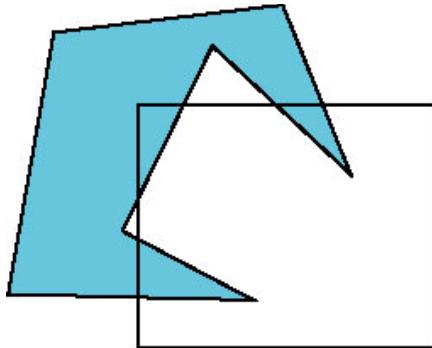
Before normalisation



After normalisation

# Polygon clipping

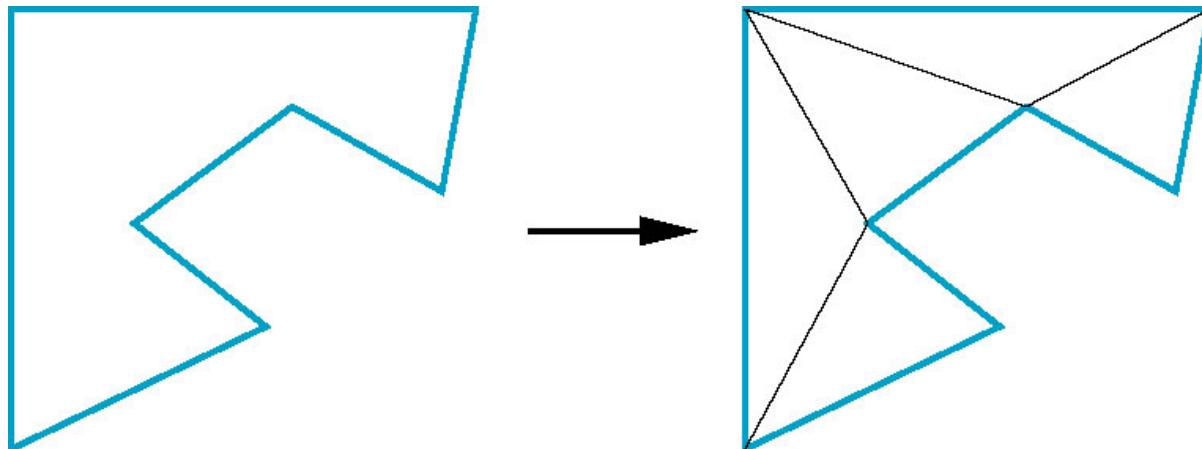
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment;
  - Clipping a polygon can yield multiple polygons.



- However, clipping a convex polygon can yield at most one other polygon.

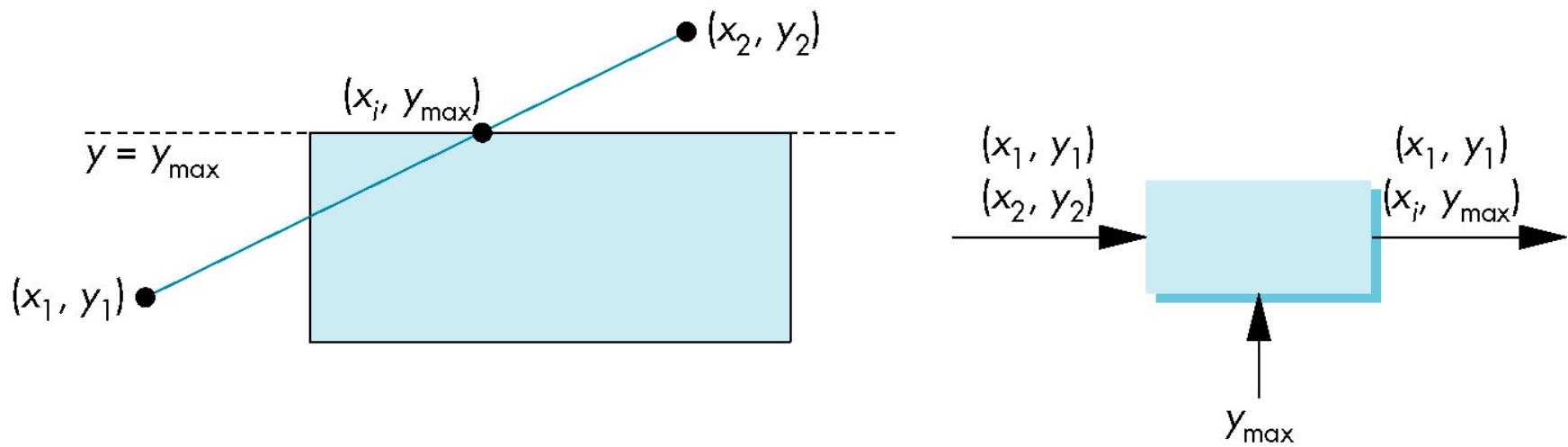
# Convexity and tessellation

- One strategy is to replace non-convex (*concave*) polygons with a set of triangular polygons (a *tessellation*).
- Also makes fill easier.
- Tessellation code in GLU library.



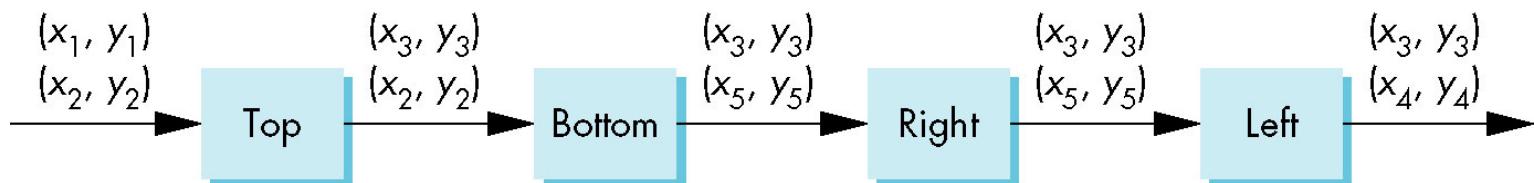
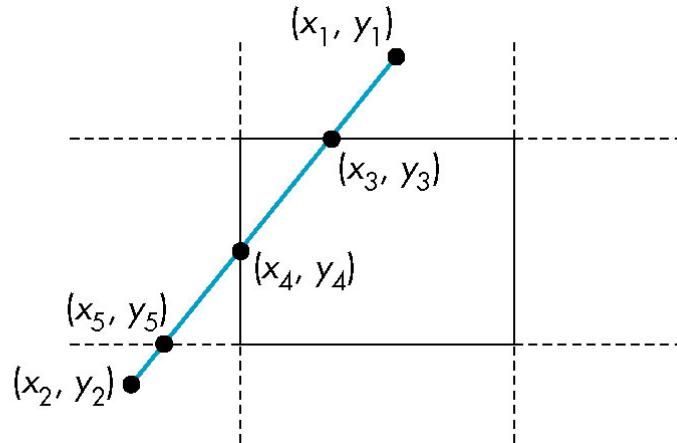
# Clipping as a black box

We can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment.



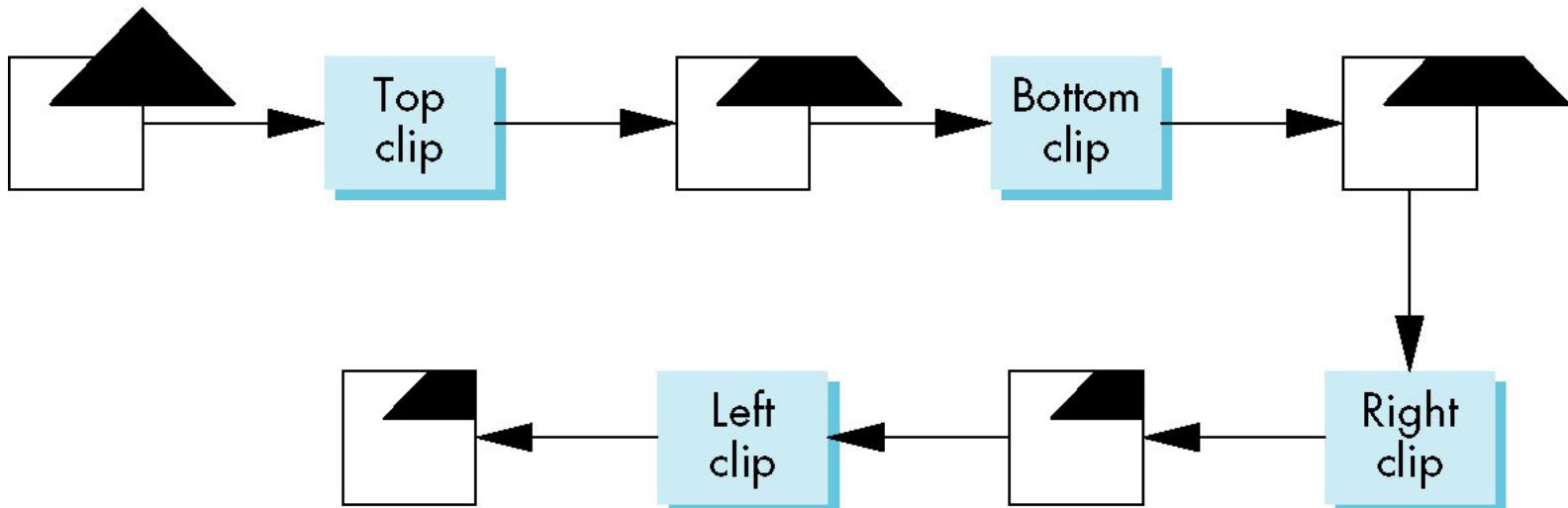
# Pipeline clipping of line segments

Clipping against each side of window is independent of other sides so we can use four independent clippers in a pipeline.



# Pipeline clipping of polygons

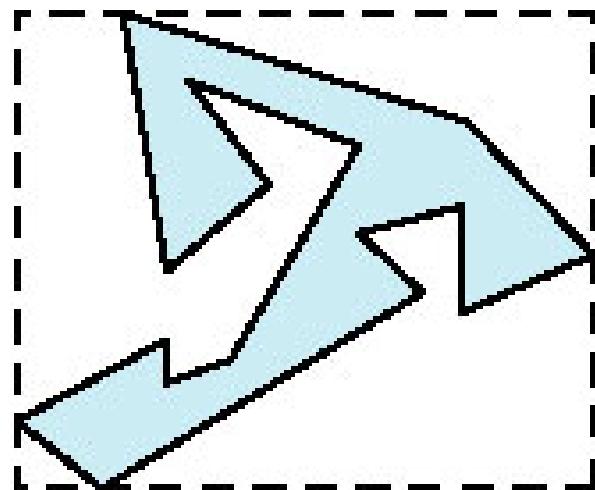
- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency



# Bounding boxes (1)

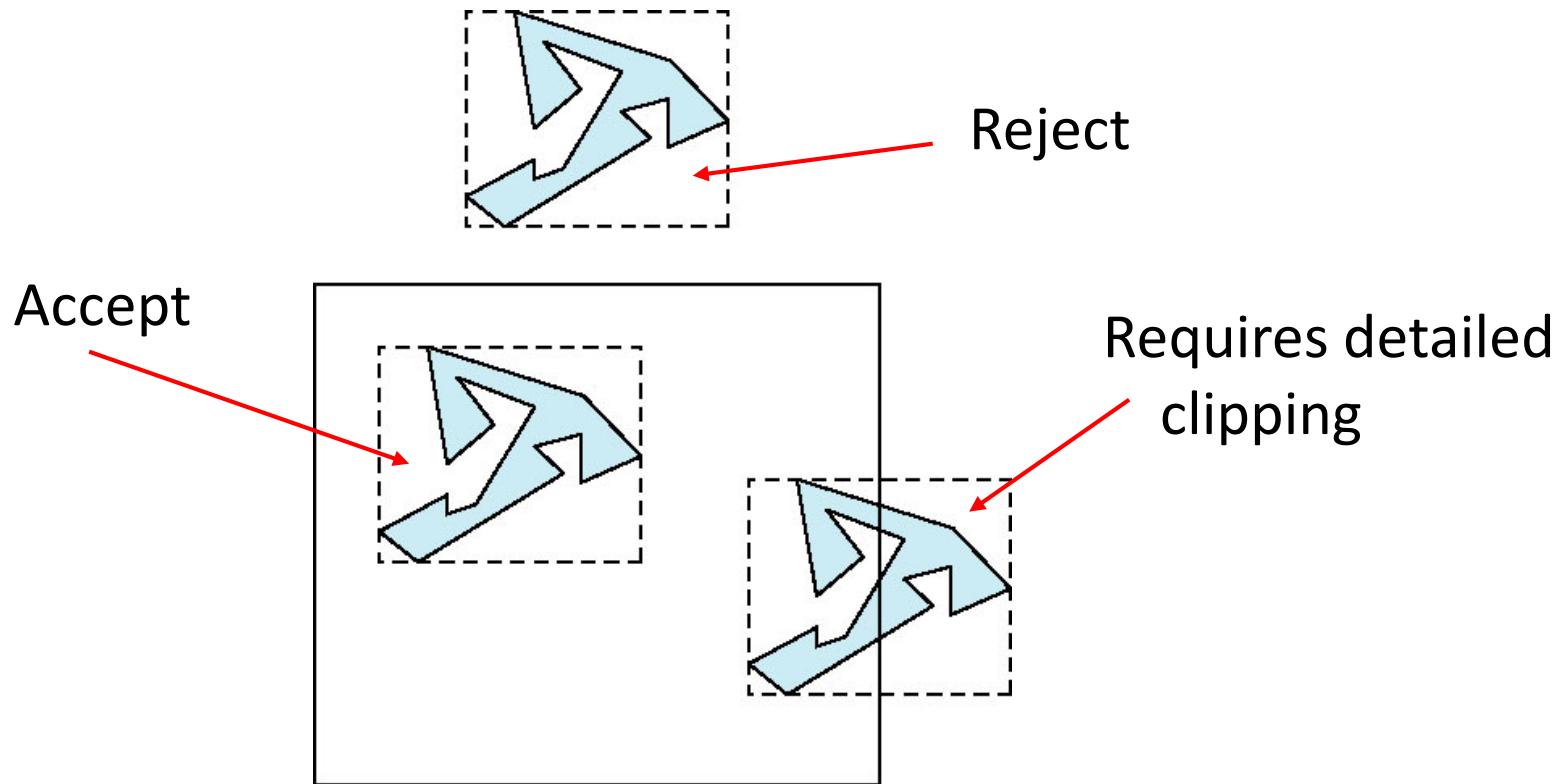
Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*:

- Smallest rectangle aligned with axes that encloses the polygon
- Simple to compute: max and min of x and y



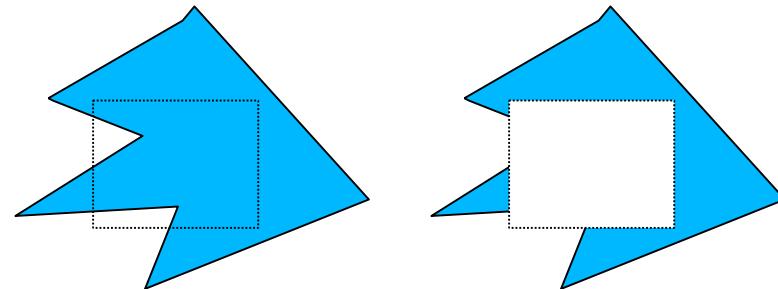
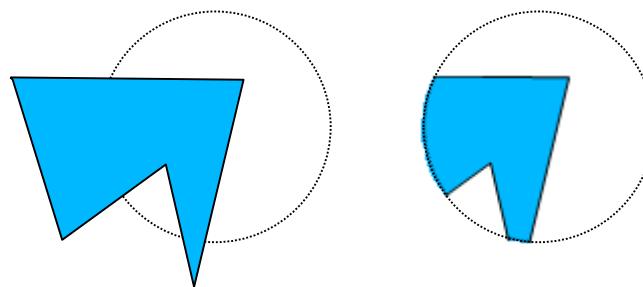
# Bounding boxes (2)

We can usually determine accept/reject based only on bounding box.



# Other issues in clipping

- Clipping other shapes:  
Circle, Ellipse, Curves
  
- Clipping a shape against  
another shape.
  
- Clipping the interior



# Setting up a 2D viewport in OpenGL

`glviewport(xvmin, yvmin, vpWidth, vpHeight);`

All the parameters are given in integer screen coordinates relative to the lower-left corner of the display window.

If we do not invoke this function, by default, a viewport with the same size and position of the display window is used (i.e., all of the GLUT window is used for OpenGL display).

# Setting up a clipping-window in OpenGL

```
glMatrixMode(GL_PROJECTION)
```

```
glLoadIdentity(); // reset, so that new viewing parameters  
// are not combined with old ones (if any)
```

```
gluOrtho2D(xwmin, xwmax, ywmin, ywmax);  
glOrtho(xwmin, xwmax, ywmin, ywmax, near, far);  
gluPerspective(vof, aspectratio, near, far);  
glFrustum(xwmin, xwmax, ywmin, ywmax, near, far);
```

Objects within the clipping window are transformed to normalised coordinates (-1,1).

# Summary

## ➤ Concepts of clipping

- What is clipping and why is it important?
- Points, lines and polygons

## ➤ Line clipping algorithms

- Brute Force Simultaneous Equations
- Brute Force Similar Triangles
- Cohen-Sutherland
- Liang-Barsky

## ➤ Polygon clipping in brief

## ➤ OpenGL functions

- `glViewport()`
- `glMatrixMode()` / `glLoadIdentity()`
- `glOrtho()` / `gluOrtho2D()`  
`gluPerspective` / `glFrustum()`



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

# **Hidden-Surface Removal**

**Week 13**  
**2021-22**

**Yong Yue**

# Topics for today

## ➤ Concepts

- What is hidden-surface removal?
- Why is hidden-surface removal important?

## ➤ Hidden-surface removal algorithms

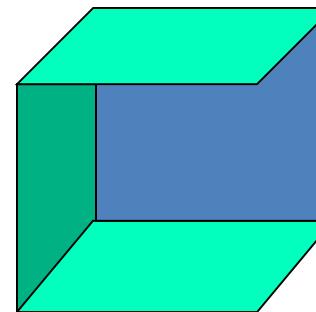
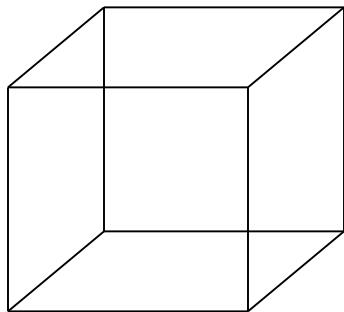
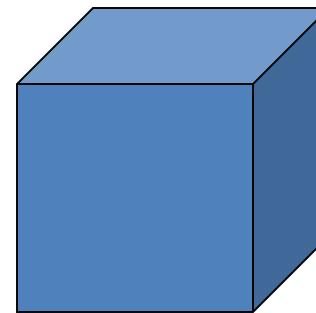
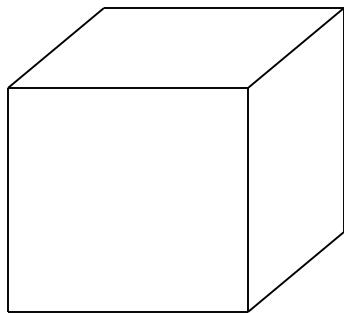
- Object space vs image space
- Painter's algorithm
- Back-face culling
- Z-buffer
- BSP tree

## ➤ Hidden-surface removal in OpenGL

# Clipping and hidden-surface removal

- Clipping has much in common with hidden-surface removal.
- In both cases, we try to remove objects that are not visible to the camera.
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline.

# Concepts (1)

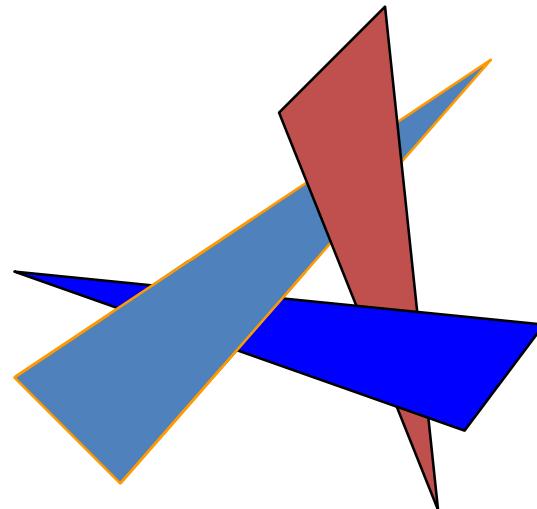


# Concepts (2)

- Display all visible surfaces, and do not display any occluded surfaces.
- Other names
  - Visible-surface detection
  - Hidden-surface elimination
- Determine which surfaces are visible and which are not.
  - Z-buffer is just one of hidden-surface removal algorithms.
- We can categorise into
  - Object-space methods
  - Image-space methods

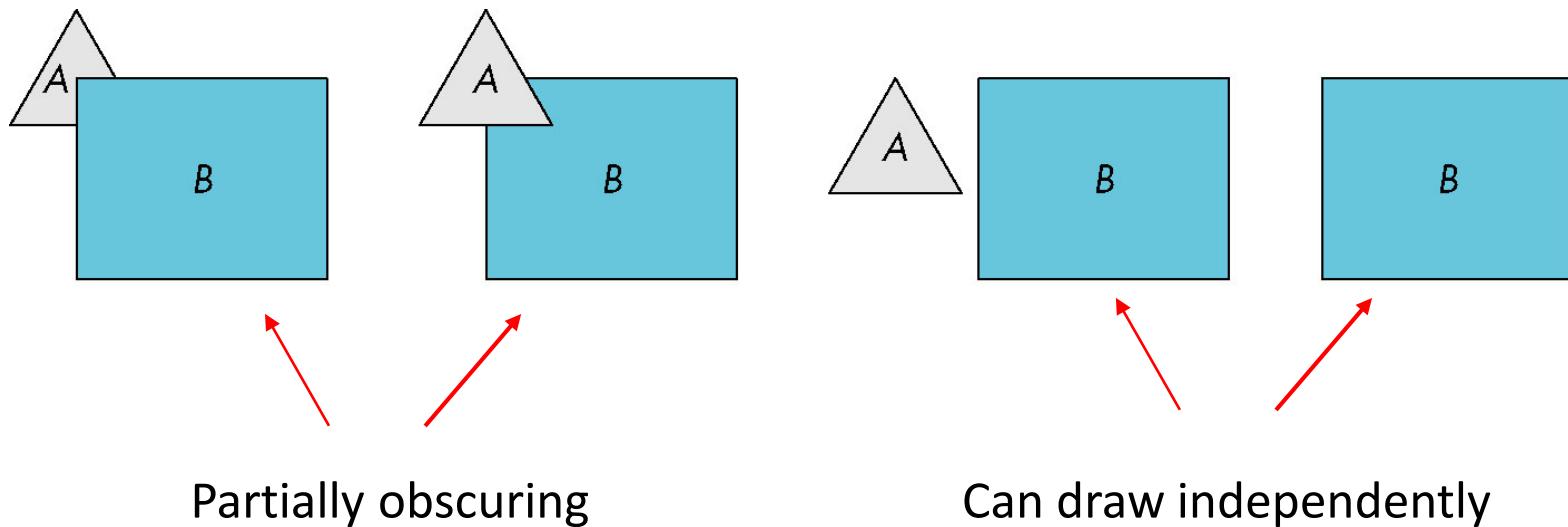
# Concepts (3)

- Object space algorithms: determine which objects are in front of others
  - Resize does not require recalculation;
  - Works for static scenes;
  - May be difficult / impossible to determine.
- Image space algorithms: determine which object is visible at each pixel
  - Resize requires recalculation;
  - Works for dynamic scenes.



# Object-space approach

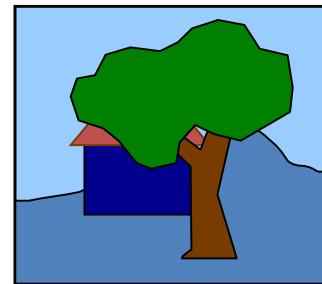
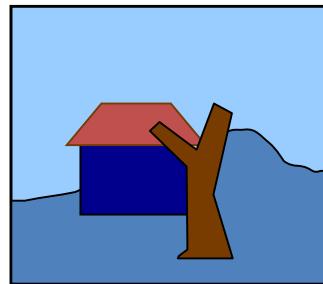
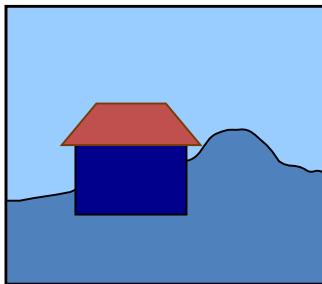
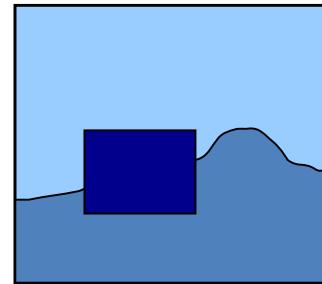
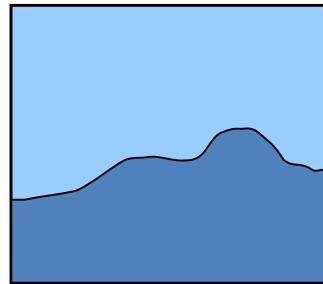
- It uses pairwise testing between polygons (objects).



- Worst case complexity  $O(n^2)$  for n polygons.

# Painter's algorithm

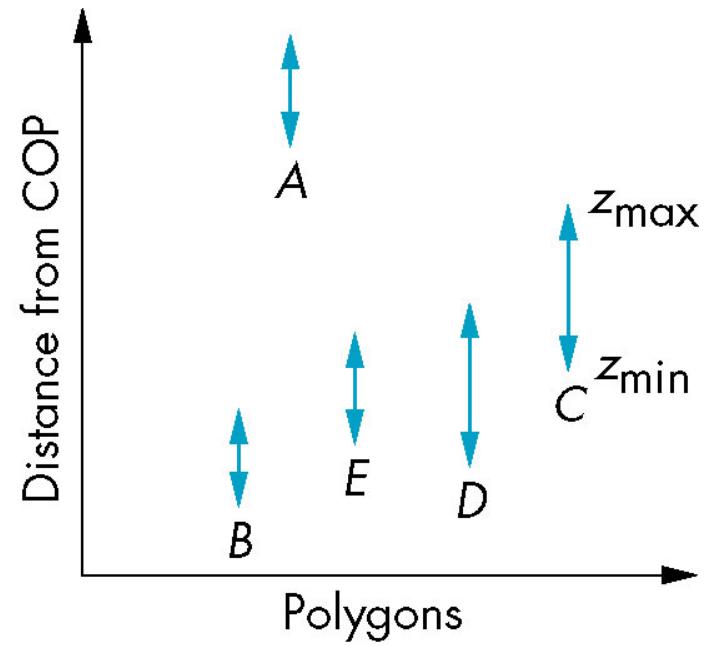
- It is an object-space algorithm.
- Render polygons in the back to front order so that polygons behind others are simply painted over.
- Sort surfaces/polygons by their depth (z value).



# Painter's algorithm: Depth sort

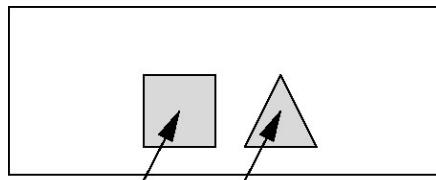
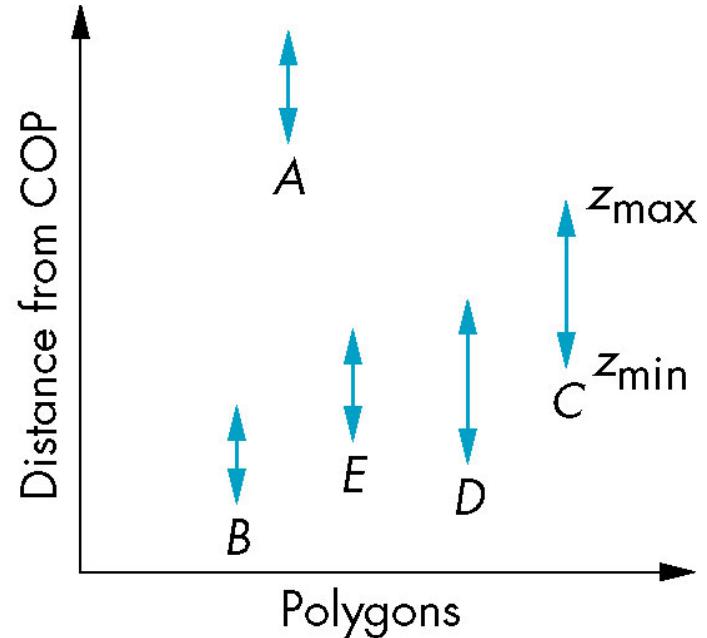
- Requires ordering of polygons first
  - $O(n \log n)$  calculations for ordering
  - Not every polygon is either in front or behind all other polygons
- Orders polygons and deals with easy cases first and harder later.

Polygons sorted by distance from Centre Of Projection (COP)

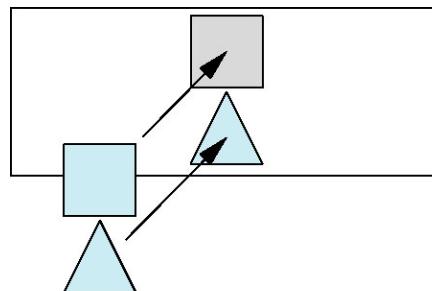


# Painter's algorithm: Easy cases

- Polygon A lies entirely behind all other polygons, and can be painted first.
- Polygons which overlap in z but not in either x or y, can be painted independently.

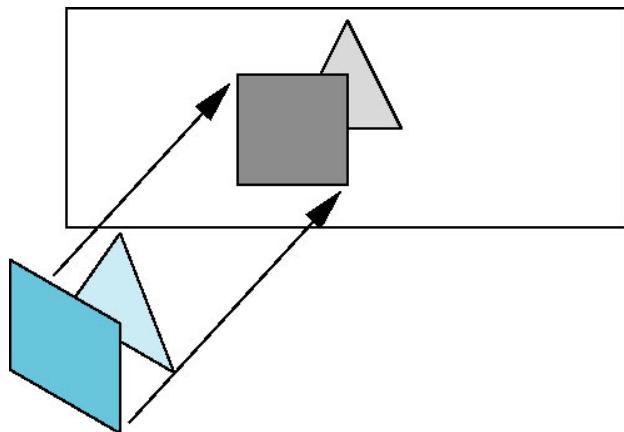


Test of overlap in x

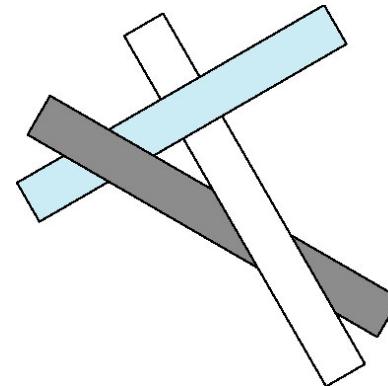


Test of overlap in y

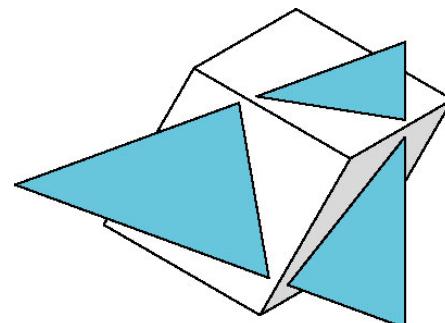
# Painter's algorithm: Hard cases



Overlap in all directions  
(polygons with  
overlapping projections)



Cyclic overlap



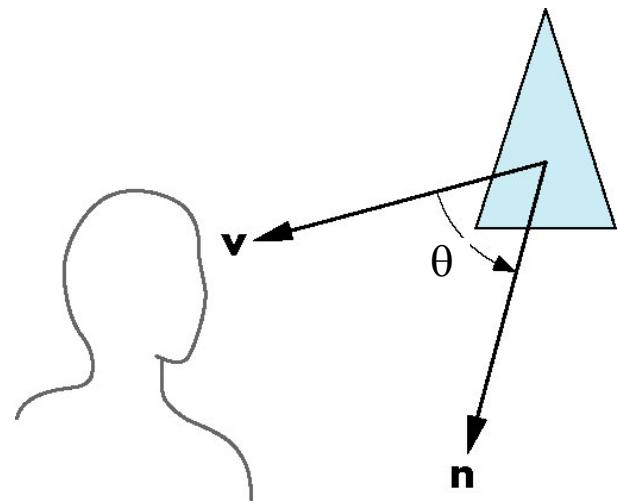
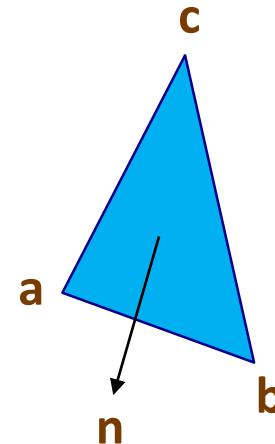
Penetration

# Back-face culling (1)

- Back-face culling is the process of comparing the position and orientation of polygons against the viewing direction  $\mathbf{v}$ , with polygons facing away from the camera eliminated.
- This elimination minimises the amount of computational overhead involved in hidden-surface removal.
- Back-face culling is basically a test for determining the visibility of a polygon, and based on this test the polygon can be removed if not visible – also called back-surface removal.

# Back-face culling (2)

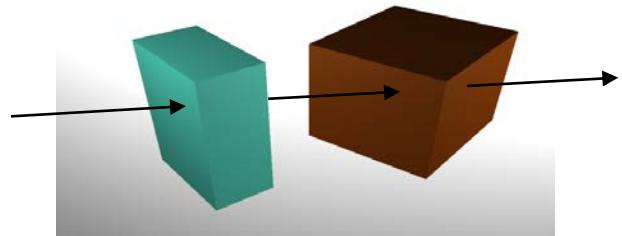
- Assumes the object is a solid polyhedron.
- Compute polygon normal  $\mathbf{n}$ :
  - Assume a counter-clockwise vertex order
  - For a triangle  $(abc)$ :  $\mathbf{n} = (\mathbf{ab}) \times (\mathbf{bc})$
- If  $90^\circ \geq \theta \geq -90^\circ$ , i.e.  $\mathbf{v} \bullet \mathbf{n} \geq 0$ , the polygon is a visible face.
- Furthermore, if the object descriptions are converted to projection co-ordinates and the viewing direction is parallel to the viewing  $z_v$  axis, i.e.  $\mathbf{n} = (0,0,1,1)^T$ , only the sign of the  $z$  component of  $\mathbf{n}$  is tested.



# Back-face culling (3)

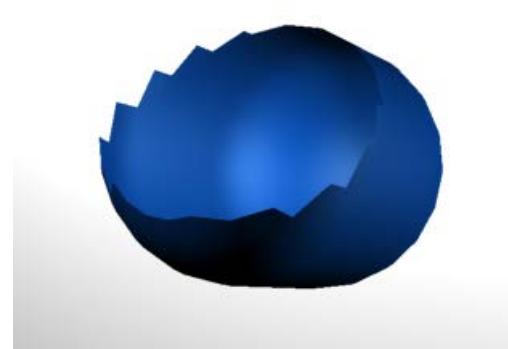
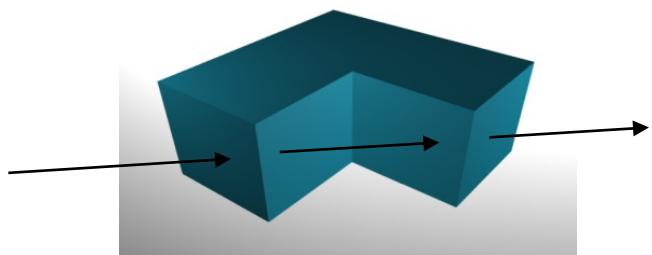
For each polygon  $P_i$

- Find polygon normal  $\mathbf{n}$
- Find viewer direction  $\mathbf{v}$
- IF  $\mathbf{v} \cdot \mathbf{n} < 0$   
Then CULL  $P_i$

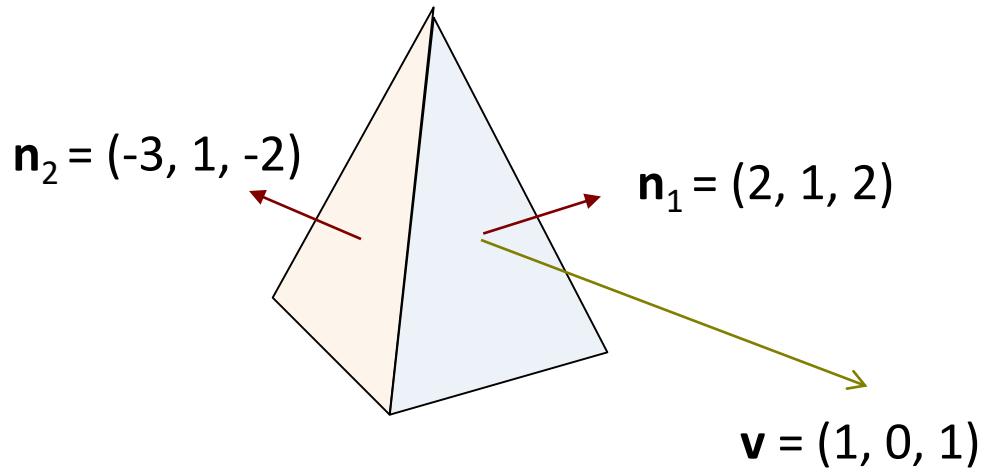


Back-face culling does not work for:

- Overlapping front faces due to
  - Multiple objects
  - Concave objects
- Non-polygonal models
- Non-closed Objects



# Back-face culling: Example



$$\begin{aligned}\mathbf{n}_1 \cdot \mathbf{v} &= (2, 1, 2) \cdot (1, 0, 1) \\ &= 2 + 0 + 2 = 4,\end{aligned}$$

i.e.  $\mathbf{n}_1 \cdot \mathbf{v} > 0$

so face<sub>1</sub> is a front facing polygon.

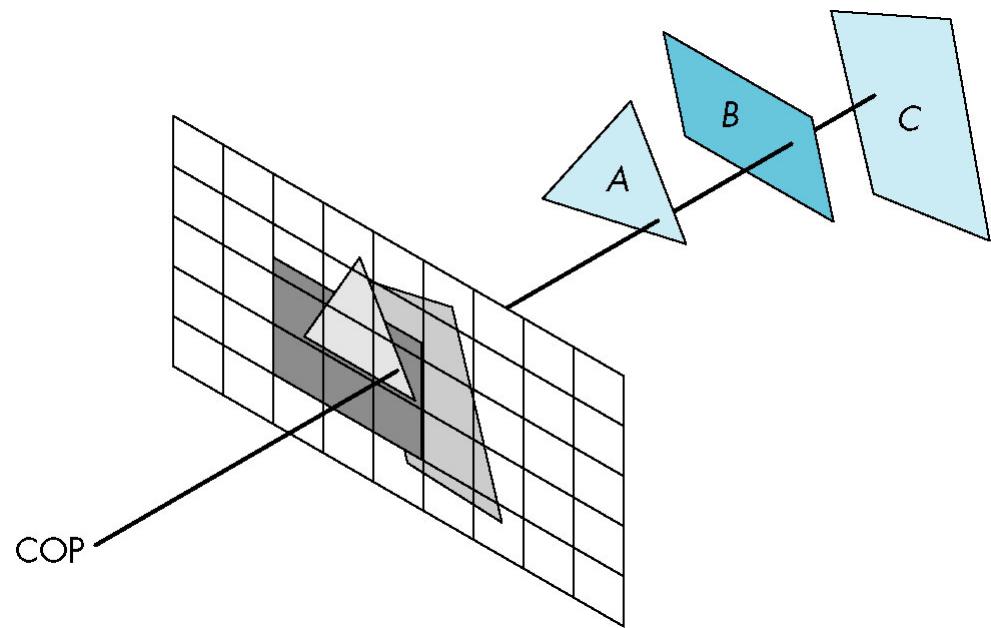
$$\begin{aligned}\mathbf{n}_2 \cdot \mathbf{v} &= (-3, 1, -2) \cdot (1, 0, 1) \\ &= -3 + 0 - 2 = -5\end{aligned}$$

i.e.  $\mathbf{n}_2 \cdot \mathbf{v} < 0$

so face<sub>2</sub> is a back facing polygon.

# Image space approach

- Look at each projector ( $n*m$  projectors for an  $n*m$  framebuffer) and find the closest among  $k$  polygons
- Complexity  $O(nmk)$
- Ray tracing
- Z-buffer



# Z-buffer (1)

- We now know which pixels contain which objects; however since some pixels may contain two or more objects, we must calculate which of these objects are visible and which are hidden.
- In graphics hardware, hidden-surface removal is generally accomplished using the Z-buffer algorithm.

# Z-buffer (2)

- In this algorithm, we set aside a two-dimensional array of memory (the Z-buffer) of the same size as the screen (number of rows \* number of columns).
- This is in addition to the colour buffer (frame buffer) which we will use to store the colour values of pixels to be displayed.
- The Z-buffer will hold values which are depths (quite often z-values).
- The Z-buffer is initialised so that each element has the value of the far clipping plane (the largest possible z-value after clipping is performed).
- The colour buffer is initialised so that each element contains a value which is the background colour.

# Z-buffer (3)

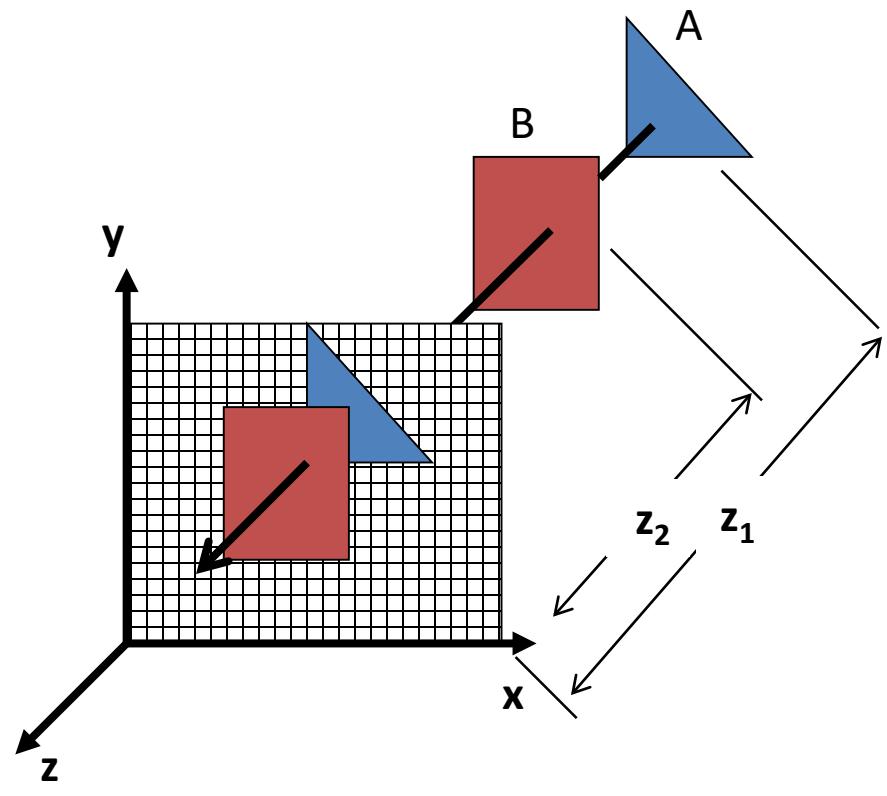
- Now for each polygon, we have a set of pixel values which the polygon covers.
- For each of the pixels, we compare its depth (z-value) with the value of the corresponding element already stored in the Z-buffer:
  - If this value is less than the previously stored value, the pixel is nearer the viewer than the previously encountered pixel;
  - Replace the value of the Z-buffer with the z value of the current pixel, and replace the value of the colour buffer with the value of the current pixel.
- Repeat for all polygons in the image.
- The implementation is typically done in normalised co-ordinates so that depth values range from 0 at the near clipping plane to 1.0 at the far clipping plane.

# Z-buffer (4)

1. Initialise all  $depth(x,y)$  to 1.0 and  $refresh(x,y)$  to background colour

2. For each pixel  
Get current value for  $depth(x,y)$   
Evaluate depth value  $z$

```
If(z > depth(x,y))  
{  
    depth(x,y) = z  
    refresh(x,y) =  $I_s(x,y)$   
}
```



Calculate this using relevant algorithms  
/illumination/fill colour/texturing

# Z-buffer (5)

## Advantages

- The most widely used hidden-surface removal algorithm;
- Relatively easy to implement in hardware or software;
- An image-space algorithm which traverses scene and operates per polygon rather than per pixel;
- We rasterize polygon by polygon and determine which (part of) polygons get drawn on the screen.

## Memory requirements

- Relies on a secondary buffer called the Z-buffer or depth buffer;
- Z-buffer has the same width and height as the framebuffer;
- Each cell contains the z-value (distance from viewer) of the object at that pixel position.

# Scan-line

If we work scan line by scan line as we move across a scan line, the depth changes satisfy  $a\Delta x + b\Delta y + c\Delta z = 0$ , noticing that the plane which contains the polygon is represented by

$$ax + by + cz + d = 0$$

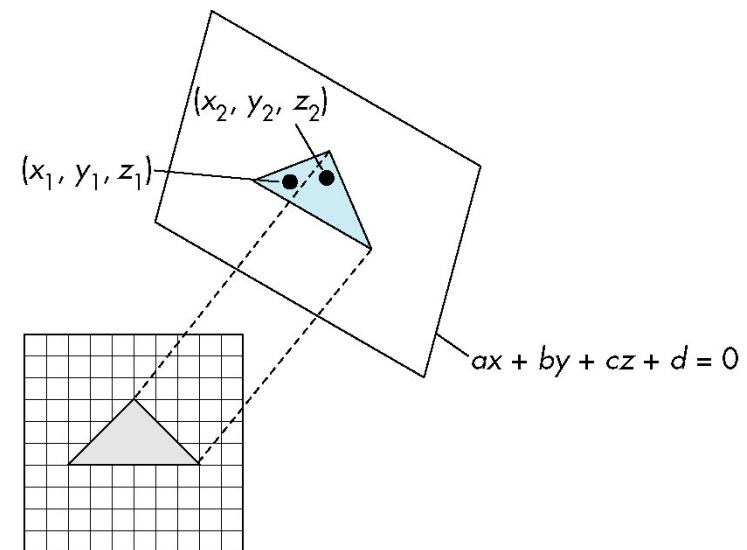
Along scan line

$$\Delta y = 0$$

$$\Delta z = -(a/c) * \Delta x$$

In screen space  $\Delta x = 1$

Only computed once per polygon.



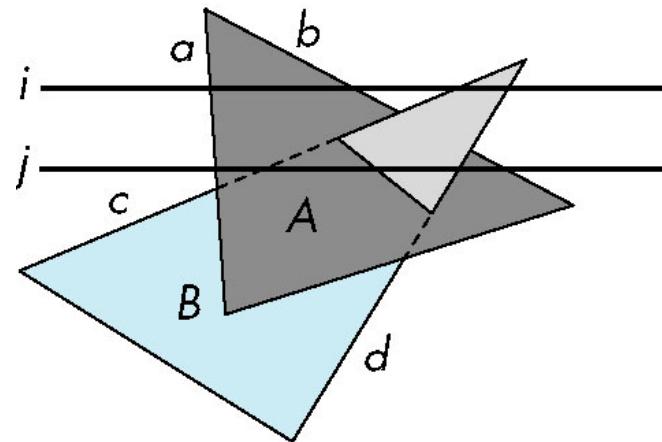
- Worst-case performance of an image-space algorithm is proportional to the number of primitives;
- Performance of z-buffer is proportional to the number of fragments generated by rasterization, which depends on the area of the rasterized polygons.

# Scan-line algorithm

We can combine shading and hidden-surface removal through scan line algorithm.

Scan line i: no need for depth information, can only be in no or one polygon.

Scan line j: need depth information only when in more than one polygon.



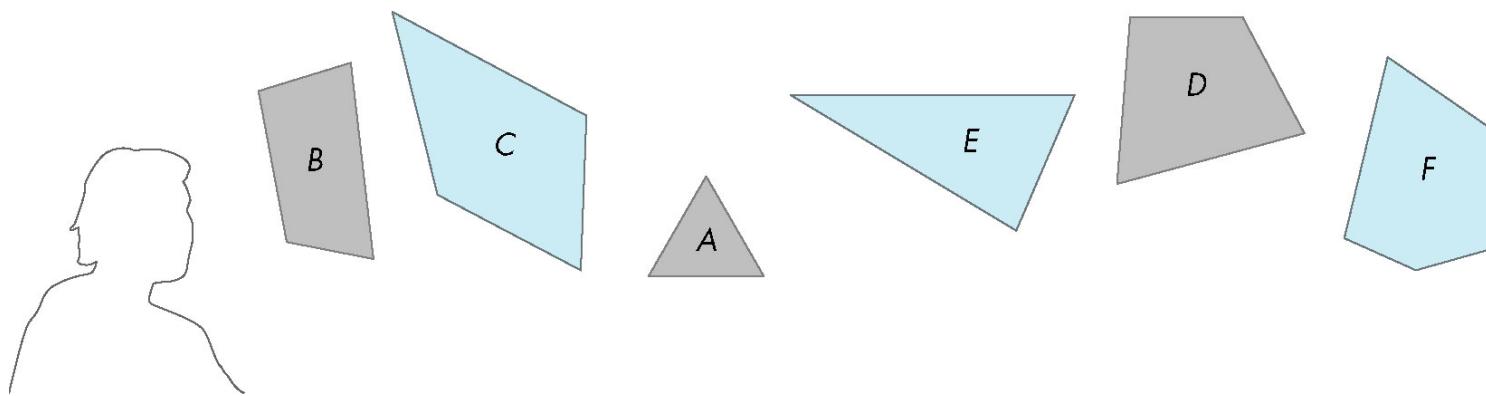
# Scan-line algorithm: implementation

Need a data structure to store

- Flag for each polygon (inside/outside)
- Incremental structure for scan lines that stores which edges are encountered
- Parameters for planes

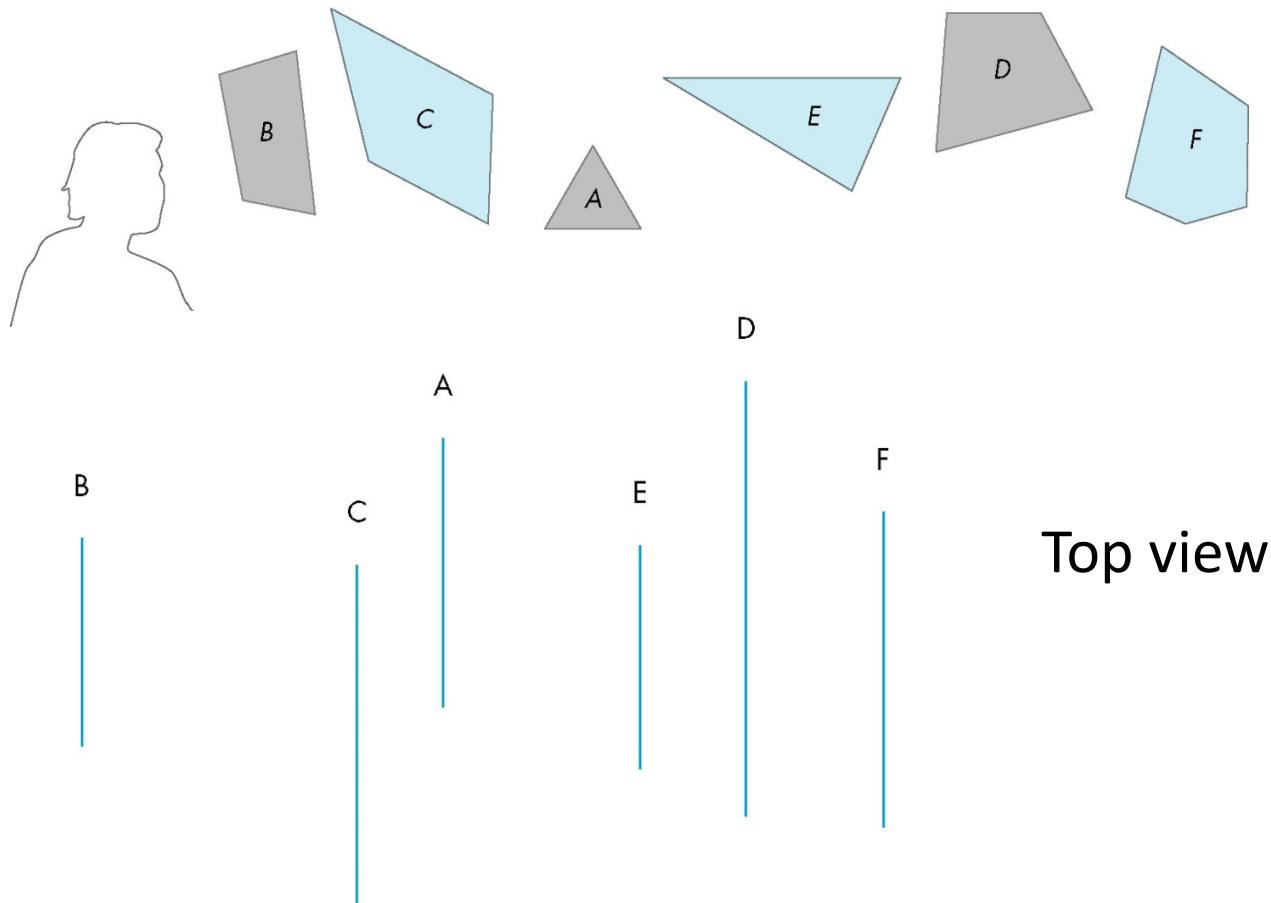
# BSP tree (1)

- In many real-time applications, such as games, we want to eliminate as many objects as possible within the application so that we can
  - reduce burden on pipeline
  - reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



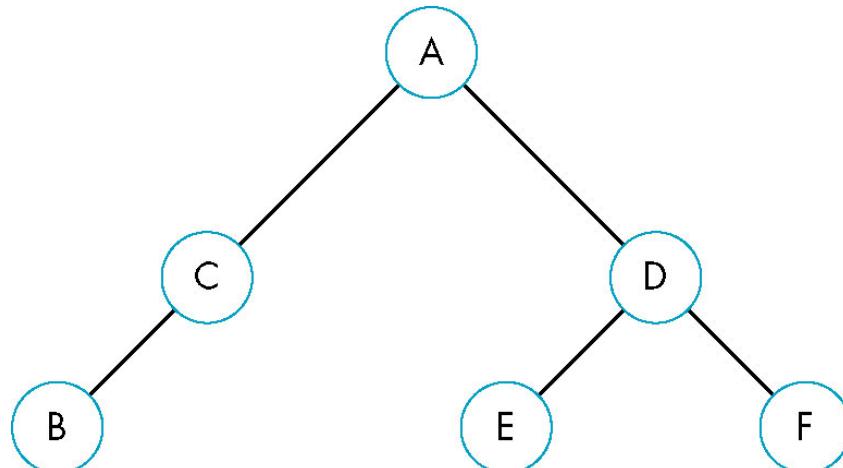
# BSP tree (2)

Consider 6 parallel planes. Plane A separates planes B and C from planes D, E and F.



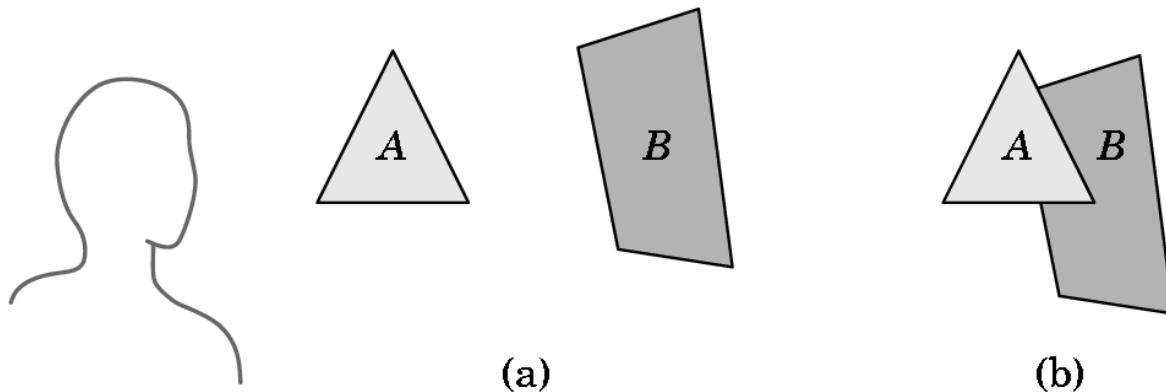
# BSP tree (3)

- We can continue recursively
  - plane C separates B from A
  - plane D separates E and F
- We can put this information in a BSP tree
  - use the BSP tree for visibility and occlusion testing



# BSP tree

- The *painter's algorithm* for hidden-surface removal works by drawing all faces, from back to front.
- How can we get a listing of the faces in the back-to-front order?
- Put them into a binary tree and traverse the tree, but in what order?



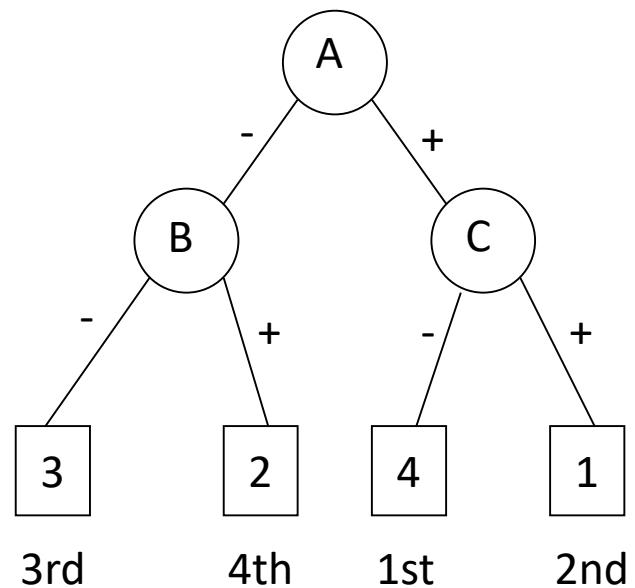
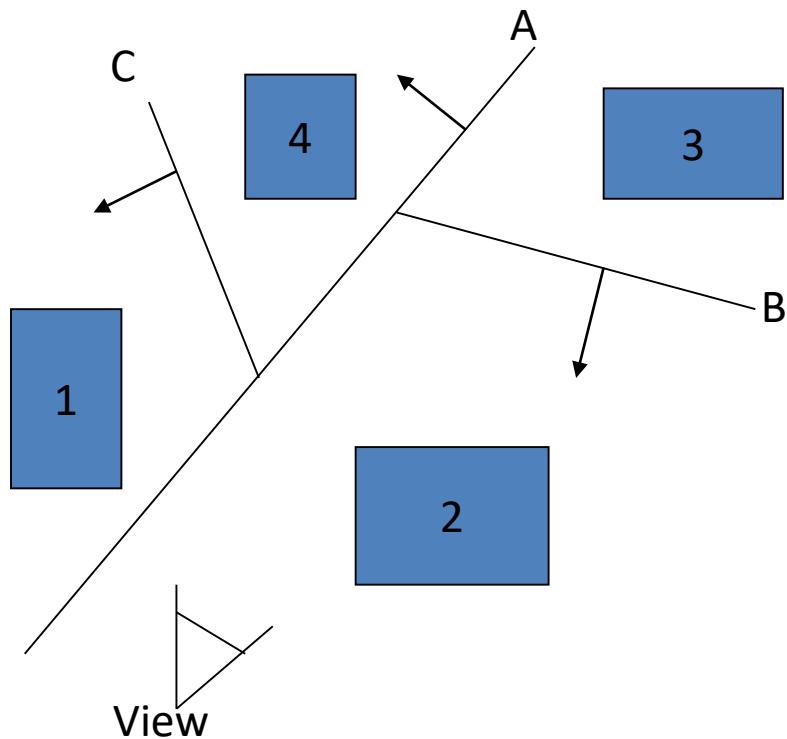
# BSP tree

- Choose polygon (arbitrary)
- Split the cell using the plane on which the polygon lies
  - May have to chop polygons in two (clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - BSP trees are not unique – there can be a number of alternatives for the same set of polygons
  - Optimal means minimum number of polygon fragments in a balanced tree

# BSP tree: Rendering

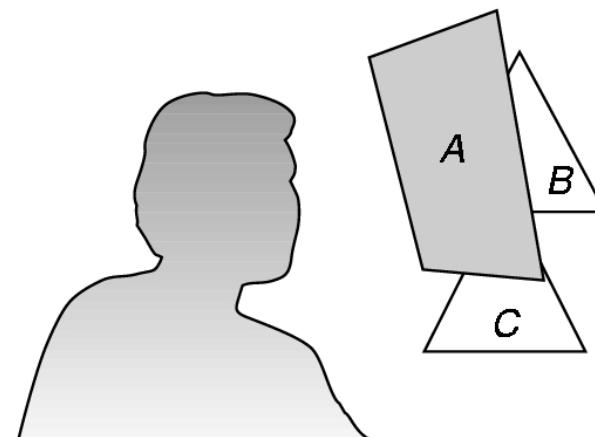
- Observation: things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint.
- The rendering is recursive descent of the BSP tree.
- At each node (for back to front rendering):
  - Recurse down the side of the sub-tree that does not contain the viewpoint
    - Test viewpoint against the split plane to decide the polygon
    - Draw the polygon in the splitting plane
      - Paint over whatever has already been drawn
  - Recurse down the side of the tree containing the viewpoint

# BSP tree: Rendering example



# OpenGL functions for hidden-surface removal

- How is hidden-surface removal done in OpenGL?
- OpenGL uses the z-buffer algorithm that saves depth information as objects/triangles are rendered so that only the front objects appear in the image.



# OpenGL Z-buffer functions

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline.
- It must be
  - requested in `main.c`  
`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`
  - enabled in `init.c`  
 `glEnable(GL_DEPTH_TEST)`
  - cleared in the display callback  
 `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

# OpenGL functions (1)

- OpenGL basic library provides functions for back-face removal and depth-buffer visibility testing.
- In addition, there are hidden-line removal functions for wireframe display, and scenes can be displayed with depth cueing.

# OpenGL functions (2)

**Face-culling functions** - Back-face removal (or back-face culling) is accomplished with the functions

```
glEnable(GL_CULL_FACE);  
glCullFace(GLenum);
```

- The parameter mode includes GL\_BACK, GL\_FRONT and GL\_FRONT\_AND\_BACK. So front faces can be removed instead of back faces, or both front and back faces can be removed.
- The default value for `glCullFace()` is GL\_BACK. Therefore, if `glEnable(GL_CULL_FACE)` is called to enable face culling without explicitly calling `glCullFace()`, back faces in the scene will be removed.

# OpenGL functions (3)

## Depth-buffer functions

- The first function is about the GLUT initialisation

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |  
GLUT_DEPTH);
```

- Depth buffer can then be initialised

```
glClear(GL_DEPTH_BUFFER_BIT);
```

- The depth-buffer visibility detection routines are activated / deactivated with

```
 glEnable(GL_DEPTH_TEST);  
 glDisable(GL_DEPTH_TEST);
```

# OpenGL functions (4)

- The depth values are normalised in the range [0.0, 1.0].  
But to speed up the surface processing, a maximum depth value, maxDepth, can be specified between 0.0 and 1.0

```
glClearDepth(maxDepth);
```

- Projection co-ordinates are normalised in the range [-1.0, 1.0], and the depth values between the near and far clipping planes are further normalised in the range [0.0, 1.0] where 0.0 and 1.0 correspond to the near and far clipping planes respectively.

# OpenGL functions (5)

- The function below allows adjustment of the clipping planes

```
glDepthRange(nearNormDepth, farNormDepth);
```

where the default values for the two parameters are 0.0 and 1.0 respectively, and they can have a value in the range of [0.0, 1.0].

- Another function for extra options is

```
glDepthFunc(testCondition);
```

*testCondition* can have the following values: GL\_LESS (default), GL\_GREATER, GL\_EQUAL, GL\_NOTEQUAL, GL\_LEQUAL, GL\_GEQUAL, GL\_NEVER (no points are processed), and GL\_ALWAYS (all points are processed), to reduce calculations based on the application.

# OpenGL functions (6)

- The status of the depth buffer can be set to read-only or read-write

```
glDepthMask(writeStatus);
```

- When `writeStatus = GL_FALSE`, the write mode in the depth buffer is disabled and only retrieval of values for comparison is possible.
- It is useful when a complex background is used with display of different foreground objects. After storing the background in the depth buffer, the write mode is disabled for processing the foreground, allowing the generation of a series of frames with different foreground objects or with one object in different positions for an animated sequence. It is also useful in displaying transparent objects.

# OpenGL functions (7)

**Wireframe surface-visibility functions** - A wireframe display of an object can be generated with

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
    // In this case, both visible and hidden edges  
    // are displayed.
```

**Depth-cueing functions** - The brightness of an object is varied as a function of its distance to the viewing position

```
glEnable(GL_FOG);  
glFog*(GL_FOG_MODE, GL_LINEAR);  
    // linear depth function for colour in [0.0, 1.0])  
glFog*(GL_FOG_START, minDepth);  
    // specifies a different value for  $d_{min}$ )  
glFog*(GL_FOG_END, maxDepth);  
    // specifies a different value for  $d_{max}$ )
```

# Summary

- Concepts of hidden-surface removal
- Relationships between clipping and hidden-surface removal
- Important techniques for hidden-surface removal
  - Object-space and image-space approaches
  - Painter's algorithms
  - Back-face culling
  - Z-buffering
  - BSP tree
- OpenGL functions

```
glutInitDisplayMode(), glClear(), glClearDepth(maxDepth),  
glDepthRange(nearNormDepth,farNormDepth);  
glDepthFunc(testCondition), glDepthMask(writeStatus),  
glPolygonMode(), glEnable(GL_DEPTH_TEST), glEnable(GL_FOG),  
glCullFace()
```



**Xi'an Jiaotong-Liverpool University**  
**西交利物浦大学**

# **CPT205 Computer Graphics**

## **Revision**

**Week 14**  
**2021-22**

**Yong Yue**

# Topics covered

Lecture	Topics
01	Introduction / Hardware and software
02	Mathematics for computer graphics
03	Graphics primitives
04	Geometric transformations
05	Viewing and projection
06	Parametric curves and surfaces
07	3D modelling
08	Hierarchical modelling
09	Lighting and materials
10	Texture mapping
11	Clipping
12	Hidden-surface removal

# Graphics hardware and software

## ➤ Graphics Hardware

- Input (light, sound and vision), processing (GPU) and output devices (Data gloves, Google glass, etc.)
- Frame buffers
- Pixels and screen resolution

## ➤ Graphics Software

- Techniques (low-level, e.g. algorithms and procedures)
- Programming library / API (OpenGL, JOGL, etc.)
- Not our focus: high level interactive systems (Maya, Studio Max, AutoCAD, etc.)

# Mathematics for computer graphics

- Computer representation of objects
- Cartesian co-ordinate system
- Points, lines and angles
- Trigonometry
- Vectors, unit vectors and vector calculations
  - Magnitude and direction
  - Form of  $\mathbf{v} = v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$
  - Dot product ( $\mathbf{V1} \bullet \mathbf{V2} = x_1 * x_2 + y_1 * y_2$  and  $\mathbf{V1} \bullet \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \cos(\alpha)$ , so  $\cos(\alpha) = ?$ )
  - Cross product ( $\mathbf{V1} \times \mathbf{V2} = |\mathbf{V1}| |\mathbf{V2}| \sin(\alpha)\mathbf{n}$ , thus  $|\mathbf{V1} \times \mathbf{V2}| = |\mathbf{V1}| |\mathbf{V2}| \sin(\alpha)$ )
- Matrices and matrix calculations
  - Dimensions (rows x columns)
  - Square, symmetric, identity and inverse matrices (if  $\mathbf{A} \times \mathbf{B} = \mathbf{B} \times \mathbf{A} = \mathbf{I}$ , then  $\mathbf{A} = \mathbf{B}^{-1}$  and  $\mathbf{B} = \mathbf{A}^{-1}$ )
  - Multiplication: condition and resultant matrix ( $\mathbf{M}_1(r_1, c_1) \times \mathbf{M}_2(r_2, c_2) = \mathbf{M}_3(r_1, c_2)$  where  $c_1 = r_2$ )

# Geometric primitives

- Transformation pipeline
- Primitives: points, lines and polygons
- Algorithms
  - DDA (Digital Differential Analyser) for line generation
  - The Bresenham line algorithm
  - Use of symmetry to reduce computation for circle generation
- Polygons and triangles
  - Polygon as an ordered set of vertices
  - Graphics hardware is optimised for processing points and flat polygons
  - Complex objects are decomposed into triangles as they are always flat
  - Polygon fill

# Transformation pipeline and geometric transformations

- The transformation pipeline  
(Modelling-Viewing-Projection-Normalisation-Device)
- Types of transformation
  - Translation
  - Rotation (about origin in 2D and an axis in 3D)
  - Scaling
  - Reflection (about an axis in 2D and a plane in 3D)
  - Shearing
- Homogeneous co-ordinate transformation matrices (4x4 in 3D)
  - For single transformations
  - Composite matrices (e.g. T·R·S)
- OpenGL functions

`glMatrixMode(), glLoadIdentity(), glTranslate(), glRotate(),  
glScale(), glPushMatrix(), glPopMatrix()`,

# Viewing and projection

- Types of projection
  - Planar geometric projection
  - Parallel (multiview orthographic, axonometric and oblique)
  - Perspective (1, 2 and 3 vanishing points)
  - Advantages and disadvantages
- Orthogonal projection
- Frustum / Perspective projection
- Parameters for 3D viewing and projection
  - Camera position, look-at point, view-up vector for  $x_{\text{view}}$ -axis
  - Viewing volume, near and far clipping planes
  - Viewing plane / clipping window
- OpenGL functions

```
glMatrixMode(), glFrustum(), gluLookAt(), glOrtho(), ...
```

# Parametric curves and surfaces

➤ Why parametric (explicit) representation?

➤ Parametric curves

- In 2D:  $x = x(t)$ ,  $y = y(t)$ ,  $(0 \leq t \leq 1)$

Line:  $x = x_1 + t(x_2 - x_1)$ ,  $y = y_1 + t(y_2 - y_1)$ ,  $(0 \leq t \leq 1)$

Circle:  $x = r \cos(360t)$ ,  $y = r \sin(360t)$ ,  $(0 \leq t \leq 1)$

- In 3D:  $x = x(t)$ ,  $y = y(t)$ ,  $z = z(t)$ ,  $(0 \leq t \leq 1)$

➤ Cubic curves and splines

- Control points
- Interpolation curves and design curves
- Local control: tension and bias

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3, \quad (0 \leq t \leq 1)$$
$$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3$$

➤ Parametric surfaces

- Revolved, extruded and swept surfaces
- Tensor product surfaces
  - Interpolation surfaces and design surfaces
  - Controls

# 3D modelling

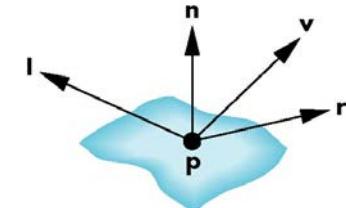
- Basic techniques: wireframes, surface models and solids; their advantages and disadvantages
- Constructive Solid Geometry (CSG)
  - CSG tree
  - Non-uniqueness
- Boundary Representation (B-Rep)
  - Boundary elements (geometry – points, curves and surfaces)
  - Relationships between boundary elements (topology / connectivity – vertices, edges and faces)
  - Types of B-Rep models – manifold and non-manifold  
(vertex connecting at least 3 edges and edge connecting exactly 2 faces)
  - Validity of B-Rep models – Euler's law ( $V - E + F - R + 2H = 2S$ )
  - Implementation of B-Rep models with OO techniques (e.g. C++)

# Hierarchical modelling

- Important concepts
  - Local and world co-ordinate frames of reference
  - Object transformations
- Linear modelling
  - Symbols (primitives): box, cone, cylinder, sphere, torus, etc.
  - Instances: copy, array (linear and radial)
  - Flat structure and no information of relationships between the parts
- Hierarchical modelling
  - Hierarchical trees and articulated models
  - Child inherits transformations from its parent ( $\mathbf{M} = \mathbf{M}_{\text{parent}} \cdot \mathbf{M}_{\text{local}}$ )
  - While traversing the tree, `glPushMatrix()` is called when going down a level, and `glPopMatrix()` when going up
- Examples: car, robot, torso (humanoid figure), solar system and train track

# Lighting and materials

- Lighting sources and properties
  - Infinite distant and positional
  - Position, colour, direction, shape and reflectivity
- Lighting and material effects: ambient, diffuse and specular; and combined effects
- Attenuation for positional light sources
  - Idea situation: inversely proportional to the square of distance
  - Flexible implementation to include constant, linear and quadratic terms: 
$$f(d) = \frac{1}{k_c + k_id + k_qd^2}$$
- Lighting model and shading
  - Phong model: 3 components, 4 vectors, 9 intensity and 9 absorption factors
$$I = k_d I_d \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^a + k_a I_a$$
  - Polygon shading: constant (flat) and interpolative (smooth)
  - Material properties of objects being illuminated
  - Combine effects: multiple lights, and light and materials
- OpenGL functions: `glLight{if}[v]()`, `glMaterial{if}[v]()`



# Texture mapping

- Concepts and types of texture mapping
  - Texture mapping, environment mapping and bump mapping
  - Takes place at the end of the rendering pipeline
- Types of texture
  - 1D, 2D and 3D
  - Defined by an image file or a procedure
- Co-ordinate systems for texture mapping
- Secondary mapping
- Control
  - Modes: decal, modulating and blending
  - Filtering: magnification and minification, and nearest or linear interpolation
  - Wrapping: repeating and clamping
  - Multiple levels of detail: mipmapping
- OpenGL functions

```
glTexImage2D(target, level, components, w, h, border, format, type, texels)
```

# Clipping

## ➤ Concepts

- Clipping window vs viewport
- Clipping primitives: points, lines, polygons, curves and text

## ➤ Line clipping algorithms

- Brute Force Simultaneous Equations: slope-intercept formula does not handle vertical lines
- Brute Force Similar Triangles: use of similar triangles to work out co-ordinate of intersect points with clipping window edges – very expensive!
- Cohen-Sutherland
  - Outcodes for 9 regions of clipping plane
  - Logic OR and AND of outcodes for the two end points of the line
    - Trivial accept and trivial reject of a line
    - Calculate intersections when necessary
- Liang-Barsky (parametric lines): value range of parameters for the 4 intersections determine if lines are to be accepted / rejected

## ➤ Polygon clipping: bounding box used for trivial accept / reject

# Hidden-surface removal

- Concepts: clipping vs hidden-surface removal
- Object-space and image-space approaches
- Algorithms
  - Painter's algorithm: depth sort, overlap test in x and y directions, hard cases, ...
  - Back-face culling
    - If  $\mathbf{v} \cdot \mathbf{n} \geq 0$  (viewing direction and face normal), polygon is visible
    - Limitations: single and convex object
  - Z-buffer
    - A depth buffer is used in addition to frame buffer
    - For each pixel position, the polygon closest to the viewport is determined, and corresponding colour used
  - Binary Spatial Partition (BSP) tree
    - Structure and creation of BSP
    - Rendering of BSP
- OpenGL functions: `glutInitDisplayMode()`, `glClear()`,  
`glClearDepth()`, `glDepthRange();` `glDepthFunc()`, `glDepthMask()`,  
`glPolygonMode()`,  `glEnable(GL_DEPTH_TEST)`,  `glEnable(GL_FOG)`,  
 `glCullFace()`

# About final exam

## ➤ Format of exam

- 5 Questions, each worth 20 marks
- Question 1 has 10 short-answer questions
- Questions 2-4 cover main topics of the module, and each has several sub-questions

## ➤ Study of a past exam paper

- during this revision session

## ➤ Answer of general questions

- During office hours and lab session times in office SD523