



Computer Graphics

Lecture 1 – Introduction, Overview

John Shearer

Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



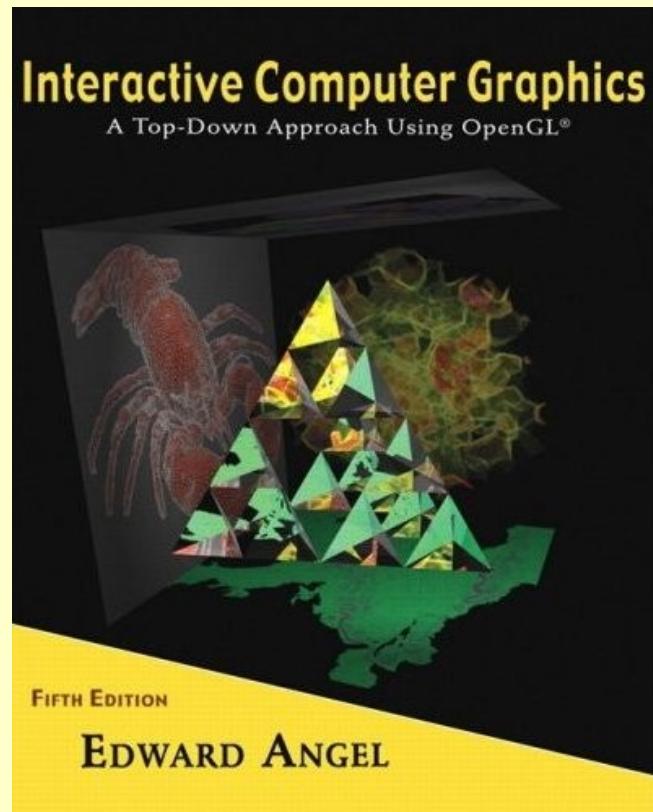
Overview

- Ed Angel, Interactive Computer Graphics, A Top-down Approach with OpenGL (Fifth Edition), Addison-Wesley – ISBN 0321535863
 - Previous editions should be ok.
 - Paperback edition recently published
- These lectures cover
 - Chapters 1-7



Acknowledgements

- Lecture course based on lecture slides by Ed Angel of the University of New Mexico to accompany Interactive Computer Graphics, A Top-down Approach with OpenGL





Objectives

- Broad introduction to Computer Graphics
 - Software
 - Hardware
 - Applications
- Top-down approach
- OpenGL



Prerequisites

- Programming skills in Java
- Basic Data Structures
 - Linked lists
 - Arrays
- Geometry
- Simple Linear Algebra



Course examination

- csc3201
- **80%** - 90 minute Written Examination end of Semester 1
- **20%** - coursework during Semester 1
 - One piece of work (*programming*)
 - Submission of working program, source code, (*short*) video, and documentation
 - *We're in the process of changing this to 50%-50%, but this is only provisional*



Resources

- Can run OpenGL on any system
 - Windows
 - Linux
 - Mac
- Using any programming language
 - C
 - C++
 - **Java (csc3201)** – with Lightweight Java Graphics Library (<http://lwjgl.org/>)
 - Python
 - ...



Resources (2)

- Handouts in practical sessions for getting a base system working
- Practicals start next week – twice a week



References

- Other helpful references
 - OpenGL: A Primer, Ed Angel, Addison-Wesley, (Third Edition), 2008
 - Designed for students who need more programming information
 - The OpenGL Programmer's Guide (the Redbook) and the OpenGL Reference Manual (The Blue book), Addison-Wesley,
 - **The definitive references**



Web Resources

- www.opengl.org
- <http://lwjgl.org/>
- <http://nehe.gamedev.net/>
- <http://stackoverflow.com/questions/62540/learning-opengl>
- <http://fly.cc.fer.hr/~unreal/theredbook/> Version 1.1
- <http://www.videotutorialsrock.com/>
- ...
- [google](#)



Outline: Part 1 - Introduction

- Text: Chapter 1
- Lectures 1-4 (angel 0-3)
 - What is Computer Graphics?
 - Applications Areas
 - History
 - Image formation
 - Basic Architecture



Outline: Part 2 - Basic OpenGL

- Text: Chapters 2-3
- Lectures 5-10 (angel 4-9)
 - Architecture
 - GLUT
 - Simple programs in two and three dimensions
 - Interaction



Outline: Part 3 - Three-Dimensional Graphics

- Text: Chapters 4-6
- Lectures 11-17 (angel 10-20)
 - Geometry – less than is in textbook
 - Transformations
 - Homogeneous Coordinates
 - Viewing
 - Shading



Outline: Part 4 – Implementation

- Text: Chapter 7
- Lectures: 18-22 (angel 21-23)
 - Approaches (object vs. image space)
 - Implementing the pipeline
 - Clipping
 - Line drawing
 - Polygon Fill
 - Display issues (color)



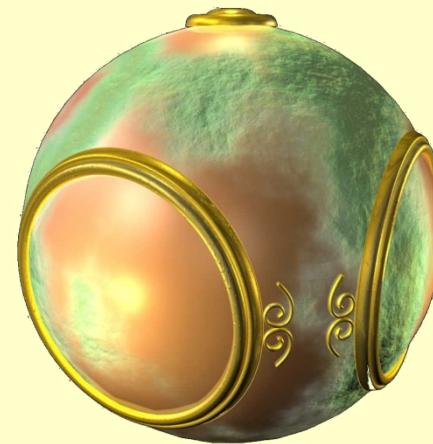
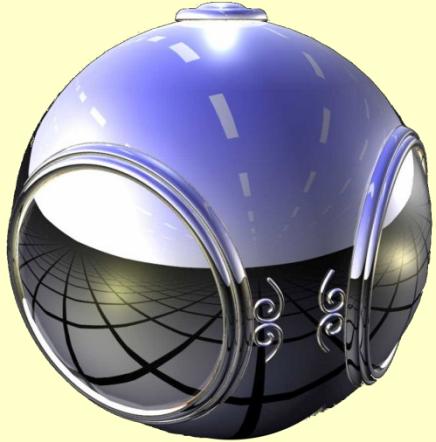
Outline: Review, revision

- Lectures: 23-24
 - Review
 - Revision
 - Questions
 - Etc.



WHY?

- Example computer graphics. Some state of the art





WHY? 2

- Example computer graphics. Some state of the art
- Motivation, etc etc





WHY? 3



- FarCry, Half-Life



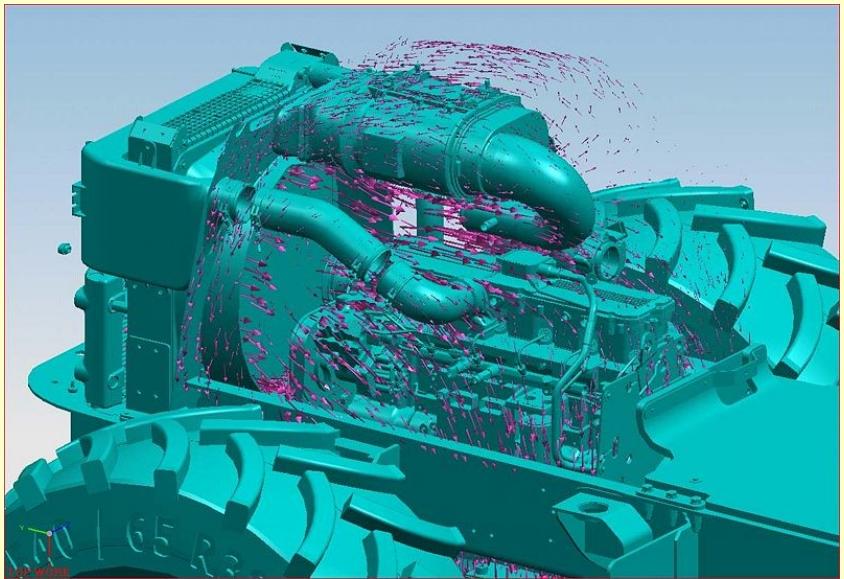
WHY? 4



- America's Army, BZflag



WHY? 5



- Unreal Tournament, CAD - airflow



WHY? 6 - The Future



Gilles Tran © 2000 www.oyondale.com



Extra if we have time

- Text: Chapter 8
 - Texture Mapping



Computer Graphics

Lecture 2 - What is Computer Graphics?

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- In this lecture, we explore what computer graphics is about and survey some application areas
- We start with a historical introduction



Computer Graphics

- *Computer graphics* deals with all aspects of creating images with a computer
 - Hardware
 - Software
 - Applications



Example

- Where did this image come from?



- What hardware/software did we need to produce it?

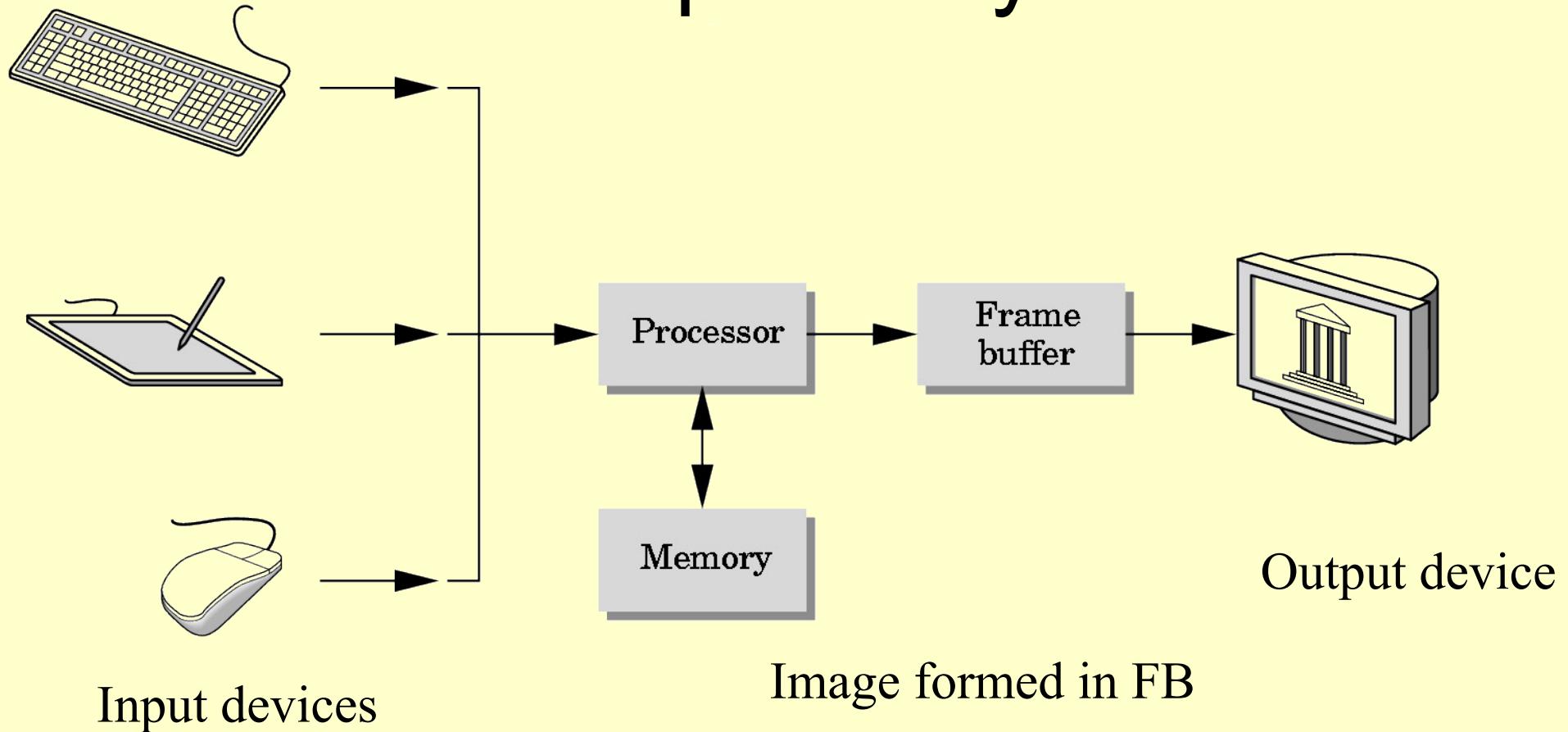


Preliminary Answer

- **Application:** The object is an artist's rendition of the sun for an animation to be shown in a domed environment (planetarium)
- **Software:** Maya for modeling and rendering but Maya is built on top of OpenGL
- **Hardware:** PC with graphics card for modeling and rendering

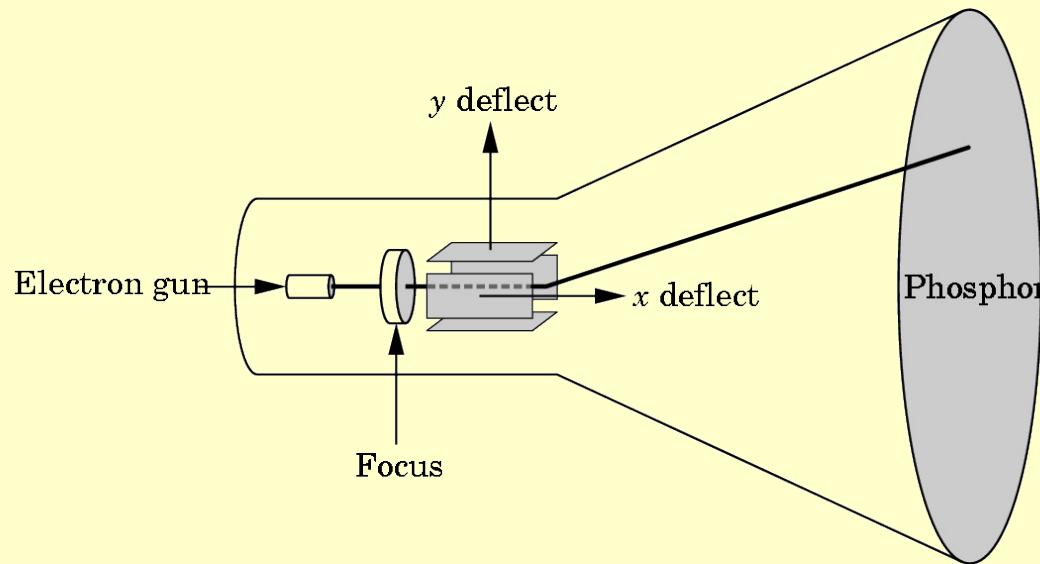


Basic Graphics System





CRT



Can be used either as a line-drawing device
(calligraphic) or to display contents of frame buffer
(raster mode)



Computer Graphics: 1950-1960

- Computer graphics goes back to the earliest days of computing
 - Strip charts
 - Pen plotters (e.g. HP 7470)
 - Simple displays using D/A converters to go from computer to calligraphic CRT
- Cost of refresh for CRT too high
 - Computers slow, expensive, unreliable



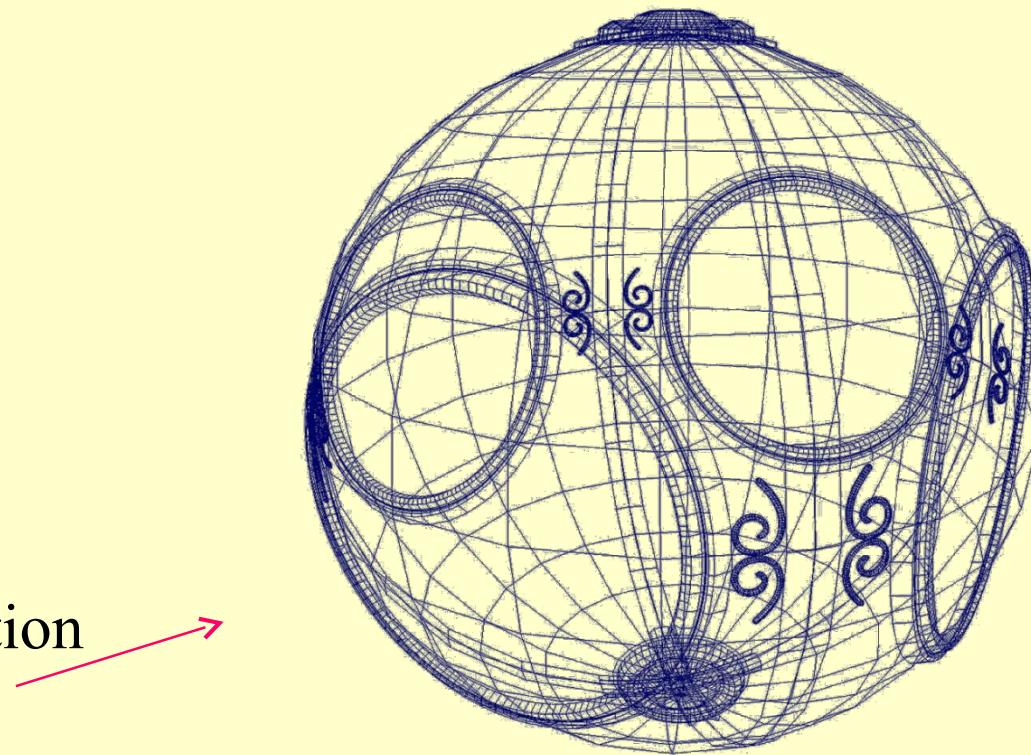
image ref: http://www.hpmuseum.net/display_item.php?hw=73



Computer Graphics: 1960-1970

- *Wireframe* graphics
 - Draw only lines
- Sketchpad
- Display Processors
- Storage tube

wireframe representation
of sun object





Sketchpad

- Ivan Sutherland's PhD thesis at MIT
 - <http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-574.pdf>
 - Recognized the potential of man-machine interaction
 - Loop
 - Display something
 - User moves light pen
 - Computer generates new display
 - Sutherland also created many of the now common algorithms for computer graphics

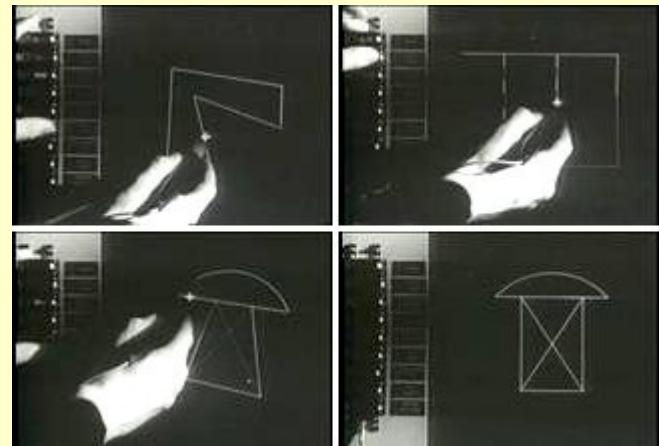
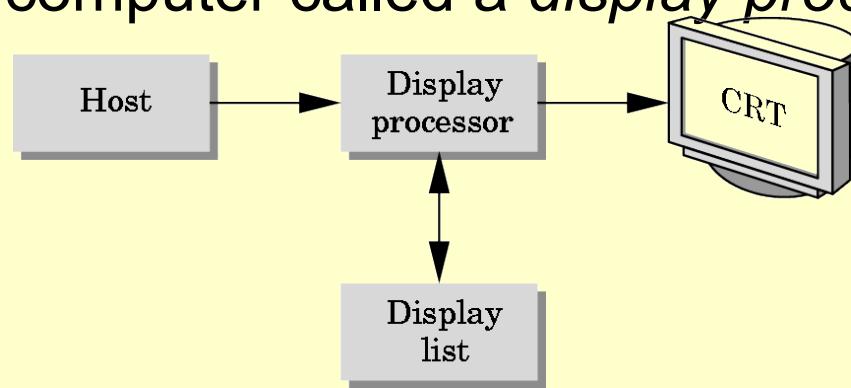


image ref: <http://en.wikipedia.org/wiki/File:Sketchpad-Apple.jpg>



Display Processor

- Rather than have the host computer try to refresh display use a special purpose computer called a *display processor* (DPU)



- Graphics stored in display list (display file) on display processor (note: this is not the same as the display lists we'll come across later)
- Host *compiles* display list and sends to DPU



Direct View Storage Tube

- Created by Tektronix
 - Did not require constant refresh
 - Standard interface to computers
 - Allowed for standard software
 - Plot3D in Fortran
 - Relatively inexpensive
 - Opened door to use of computer graphics for CAD community



image ref: http://en.wikipedia.org/wiki/File:Tektronix_4014.jpg



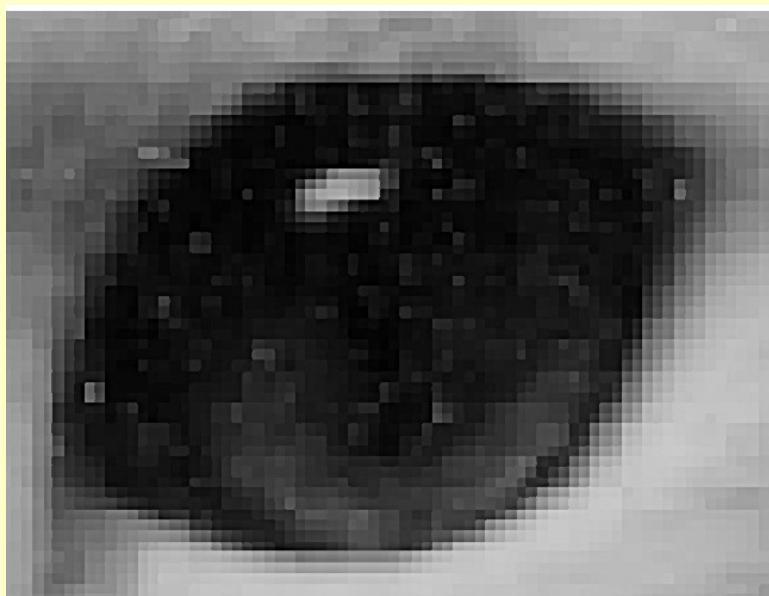
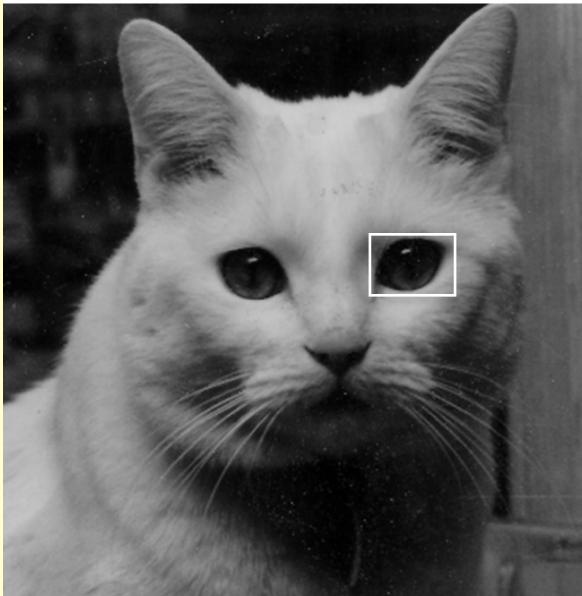
Computer Graphics: 1970-1980

- Raster Graphics
- Beginning of graphics standards
 - IFIPS
 - GKS: European effort
 - Becomes ISO 2D standard
 - Core: North American effort
 - 3D but fails to become ISO standard
- Workstations and PCs



Raster Graphics

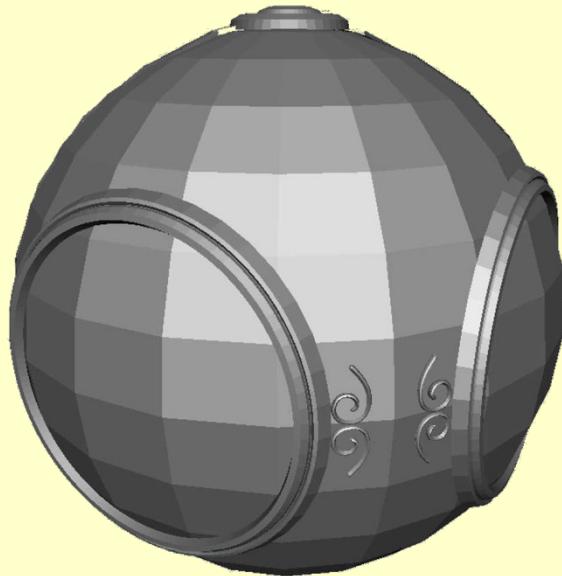
- Image produced as an array (the *raster*) of picture elements (*pixels*) in the *frame buffer*





Raster Graphics

- Allows us to go from lines and wire frame images to filled polygons





PCs and Workstations

- Although we no longer make the distinction between workstations and PCs, historically they evolved from different roots
 - Early workstations characterized by
 - Networked connection: client-server model
 - High-level of interactivity
 - Early PCs included frame buffer as part of user memory
 - Easy to change contents and create images

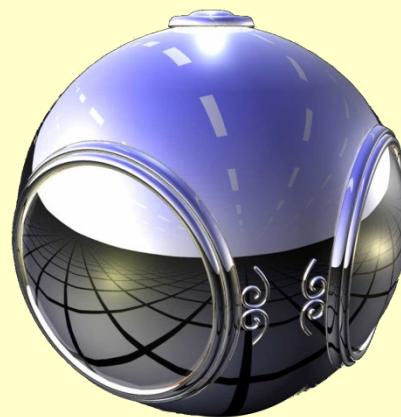


Computer Graphics: 1980-1990

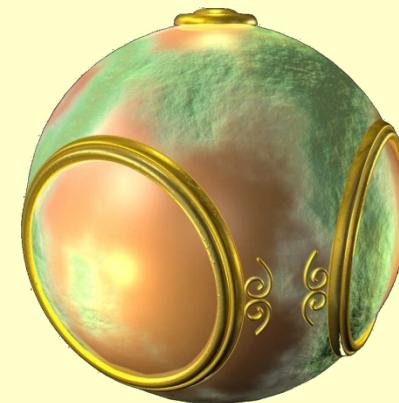
Realism comes to computer graphics



smooth shading



environment
mapping

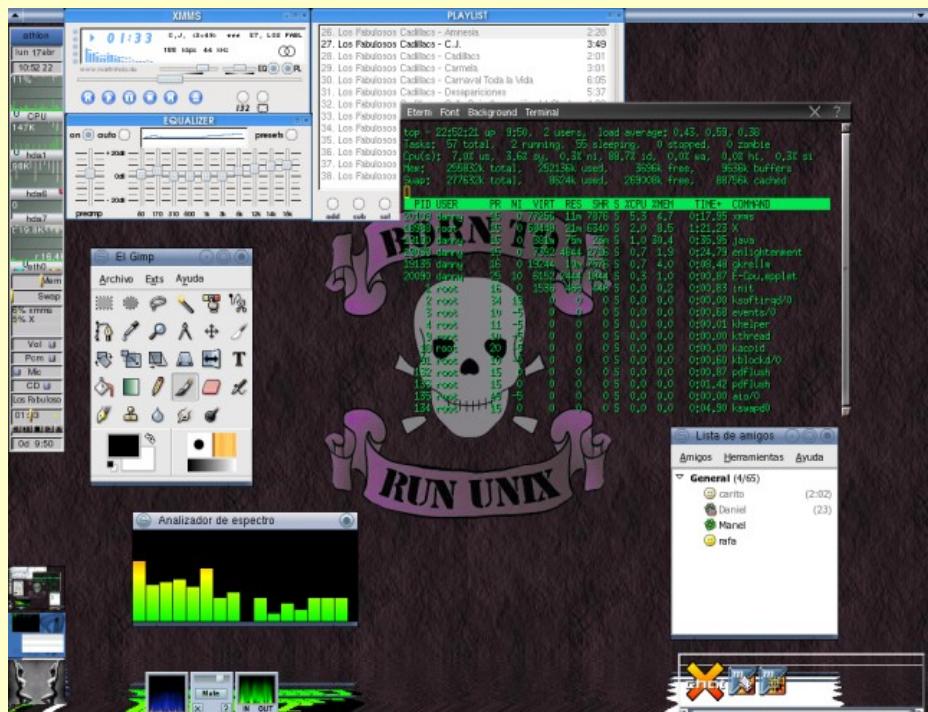


bump mapping



Computer Graphics: 1980-1990

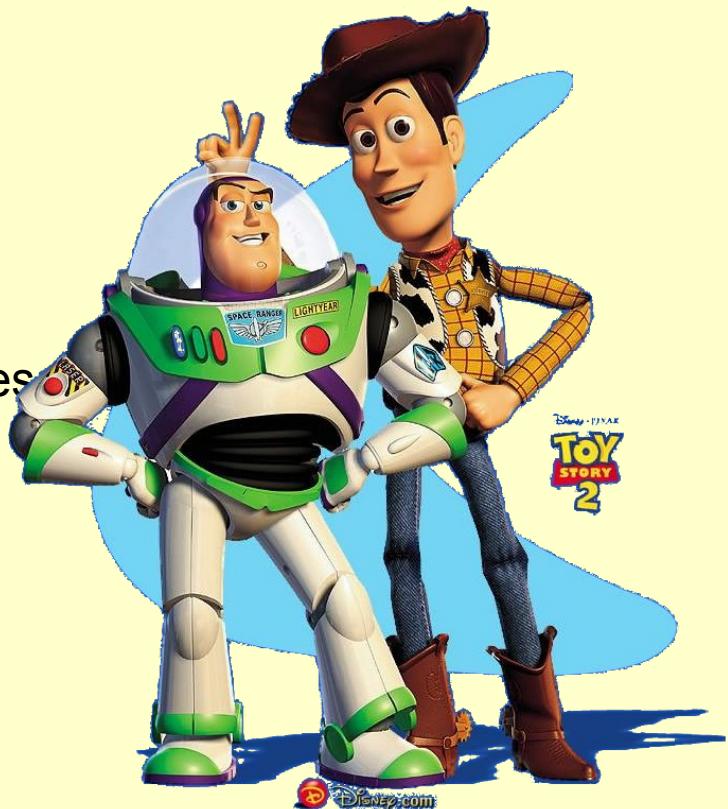
- Special purpose hardware
 - Silicon Graphics geometry engine
 - VLSI implementation of graphics pipeline
- Industry-based standards
 - PHIGS
 - RenderMan
- Networked graphics: X Window System
- Human-Computer Interface (HCI)





Computer Graphics: 1990-2000

- Completely computer-generated feature-length movies (Toy Story) are successful
- OpenGL API
- DirectX (September 1995 as the Windows Games SDK)
- New hardware capabilities
 - Texture mapping
 - Blending
 - Accumulation, stencil buffers





Computer Graphics: 2000-

- Photorealism (real-time)
- Graphics cards for PCs dominate market
 - Nvidia, ATI
- Game boxes and game players determine direction of market
- Computer graphics routine in movie industry: Maya, Lightwave
- Programmable pipelines





Computer Graphics

Lecture 3 - Image Formation

John Shearer

Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Book

Lecture 1 introduced the text for the course

Ed Angel, Interactive Computer Graphics, A Top-down Approach with OpenGL (Fifth Edition), Addison-Wesley – ISBN 0321535863

All lecture content for this module is based on the book
It is HIGHLY recommended that you get hold of a copy



Objectives

- Fundamental imaging notions
- Physical basis for image formation
 - ~~Light~~
 - ~~Color~~
 - ~~Perception~~
- Synthetic camera model
- Other models



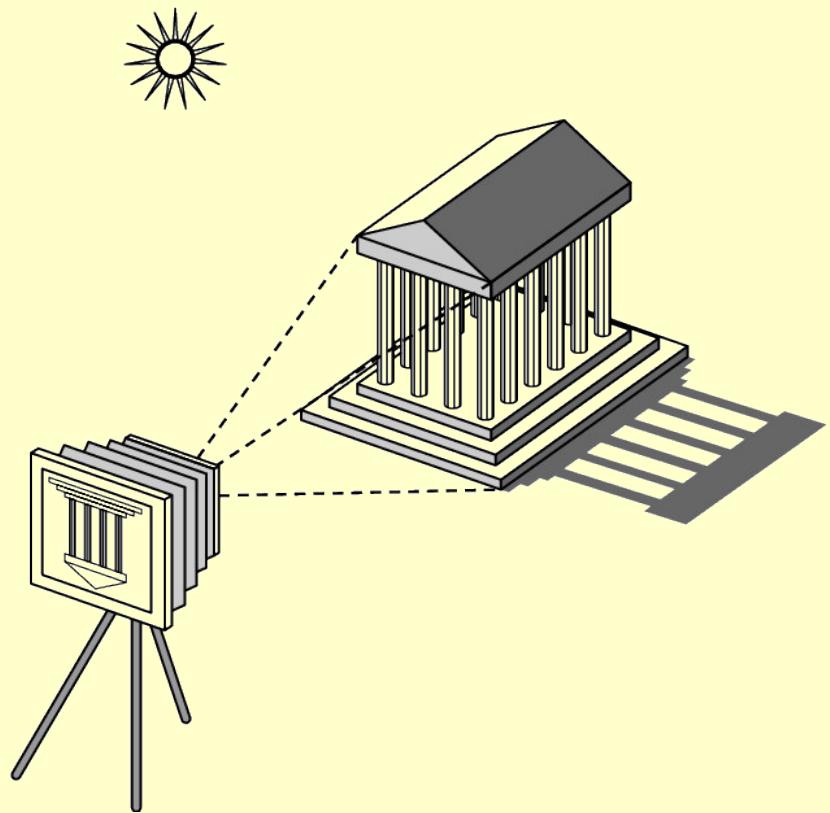
Image Formation

- In computer graphics, we form images which are generally two dimensional using a process analogous to how images are formed by physical imaging systems
 - Cameras
 - Microscopes
 - Telescopes
 - Human visual system



Elements of Image Formation

- Objects
- Viewer
- Light source(s)
- Attributes that govern how light interacts with the materials in the scene
- Note the independence of the objects, the viewer, and the light source(s)





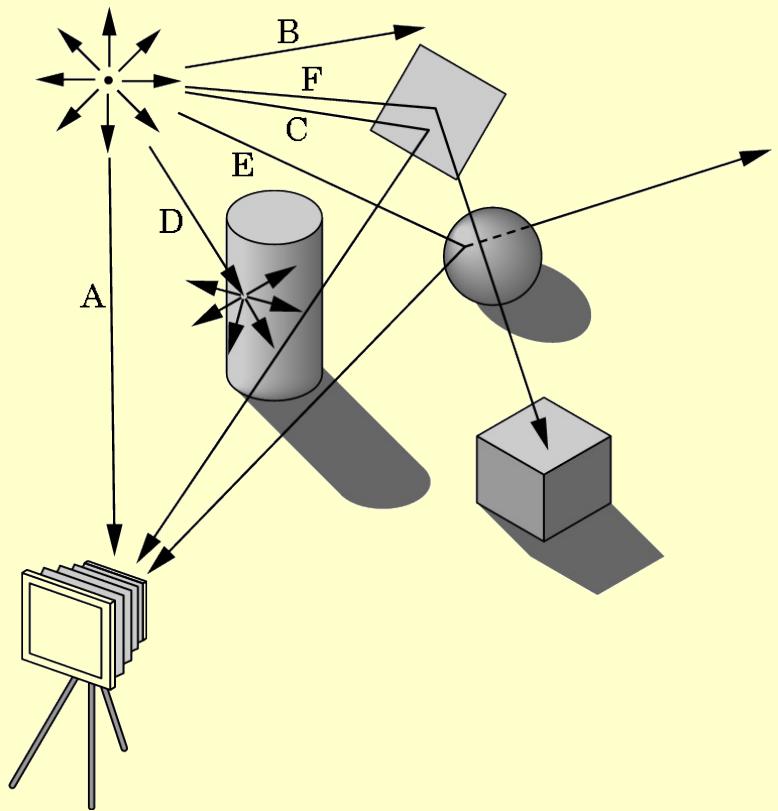
Light

- *Light* is the part of the electromagnetic spectrum that causes a reaction in our visual systems
- Generally these are wavelengths in the range of about 350-750 nm (nanometers)
- Long wavelengths appear as reds and short wavelengths as blues



Ray Tracing and Geometric Optics

- One way to form an image is to follow rays of light from a point source finding which rays enter the lens of the camera.
- However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.





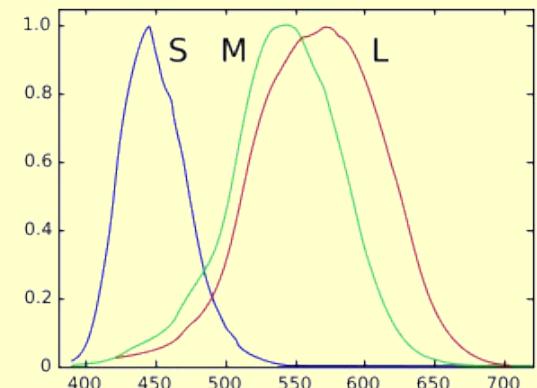
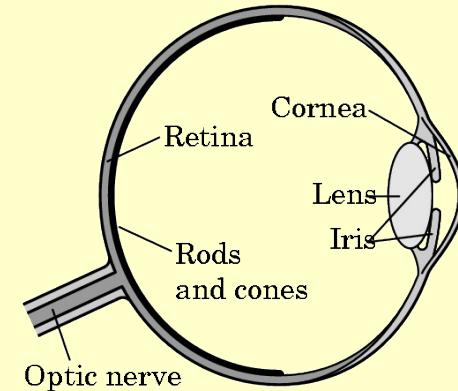
Luminance and Color Images

- Luminance Image
 - Monochromatic
 - Values are gray levels
 - Analogous to working with black and white film or television
- Color Image
 - Has perceptual attributes of hue, saturation, and lightness
 - Do we have to match every frequency in visible spectrum? No!



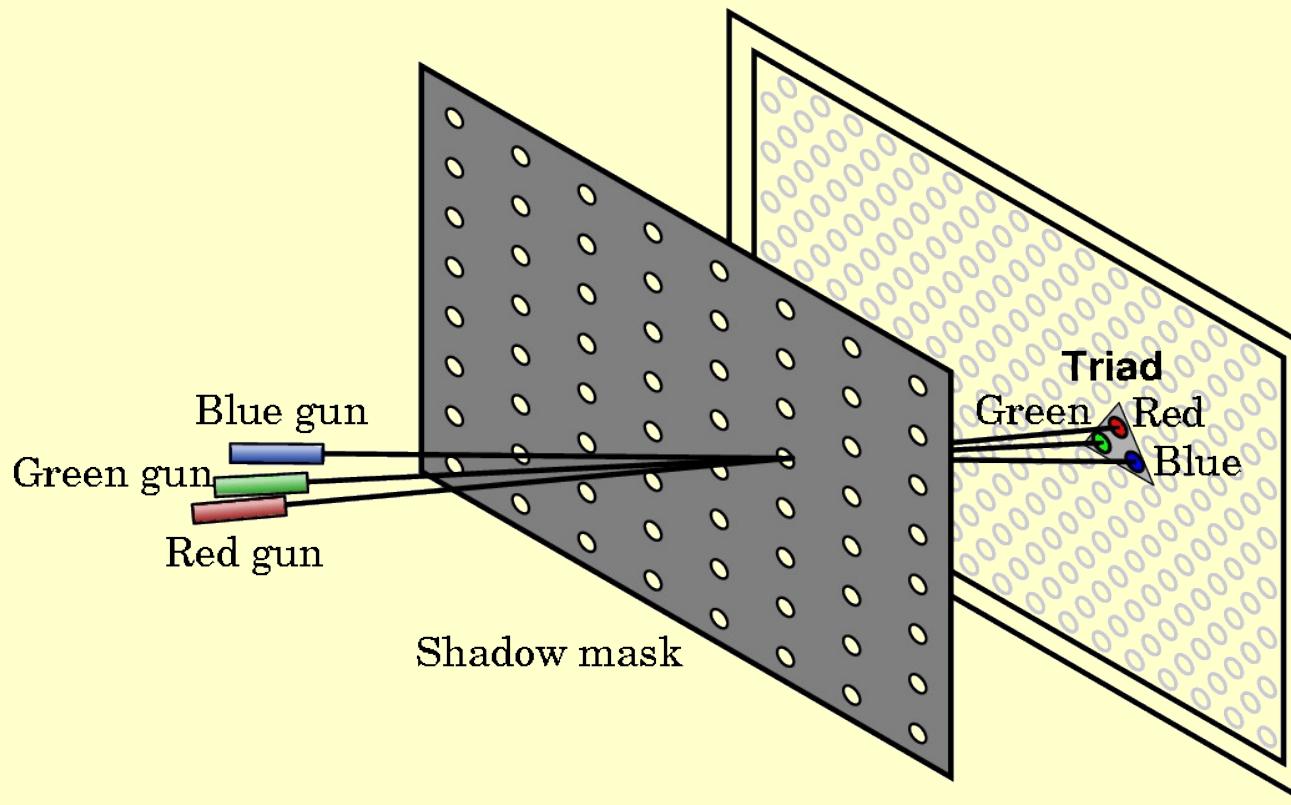
Three-Color Theory

- Human visual system has two types of sensors
 - Rods: monochromatic, night vision
 - Cones
- Color sensitive
- Three types of cones
- Only three values (the *tristimulus* values) are sent to the brain
- Need only match these three values
 - Need only three *primary colors*



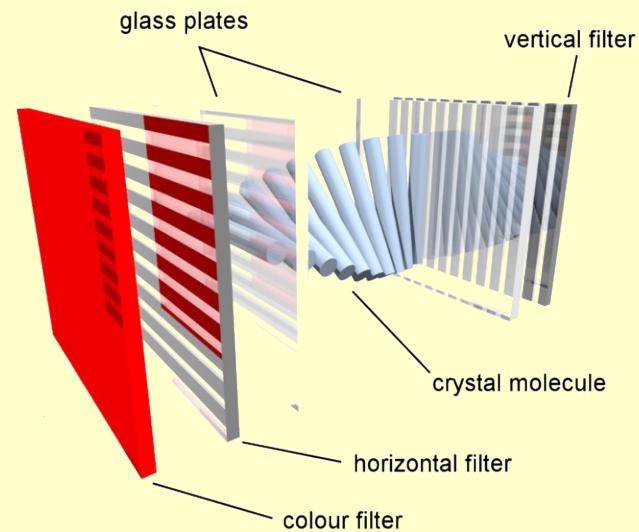


Shadow Mask CRT





Colour LCD





Additive and Subtractive Color

- Additive color
 - Form a color by adding amounts of three primaries
 - CRTs, projection systems, positive film
 - Primaries are Red (~~R~~) (#FF0000), Green (~~G~~) (#00FF00), Blue (~~B~~) (#0000FF)
- Subtractive color
 - Form a color by filtering white light with cyan (~~C~~) #00FFFF, Magenta (~~M~~) (#FF00FF), and Yellow (~~Y~~) (#FFFF00) filters
 - Light-material interactions
 - Printing
 - Negative film



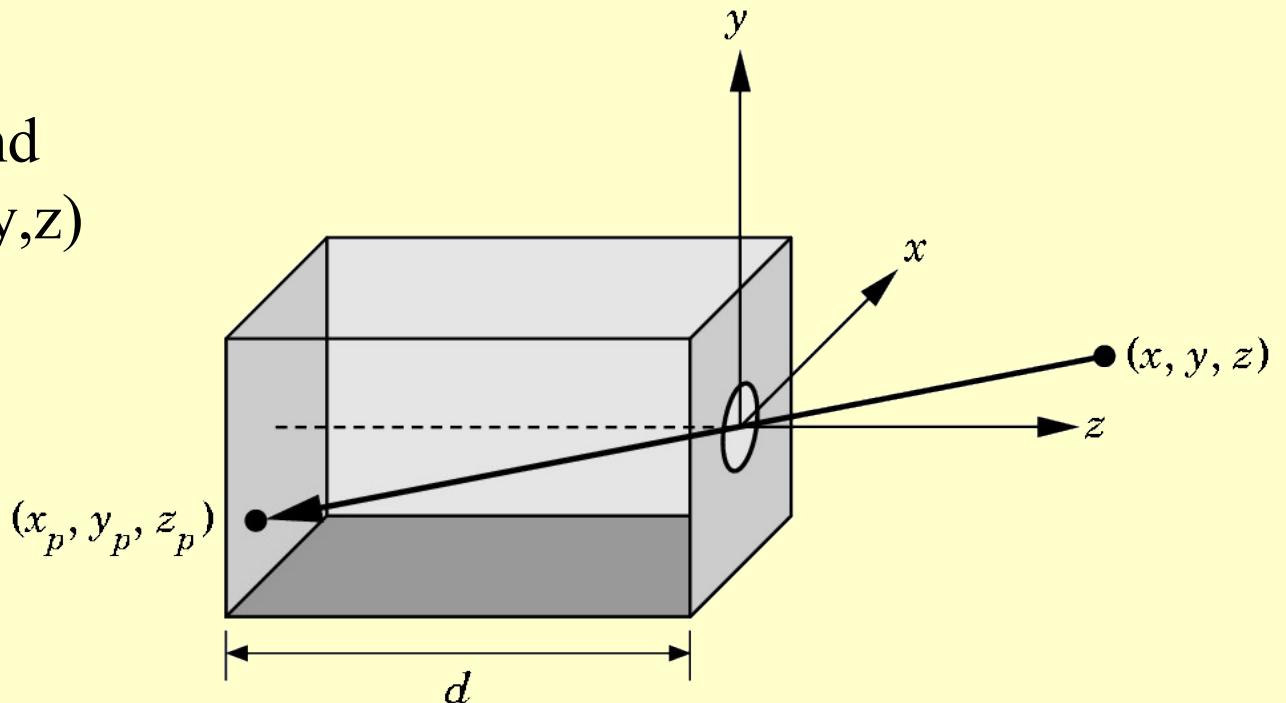
Pinhole Camera

Use trigonometry to find
projection of point at (x, y, z)

$$x_p = -x/z/d$$

$$y_p = -y/z/d$$

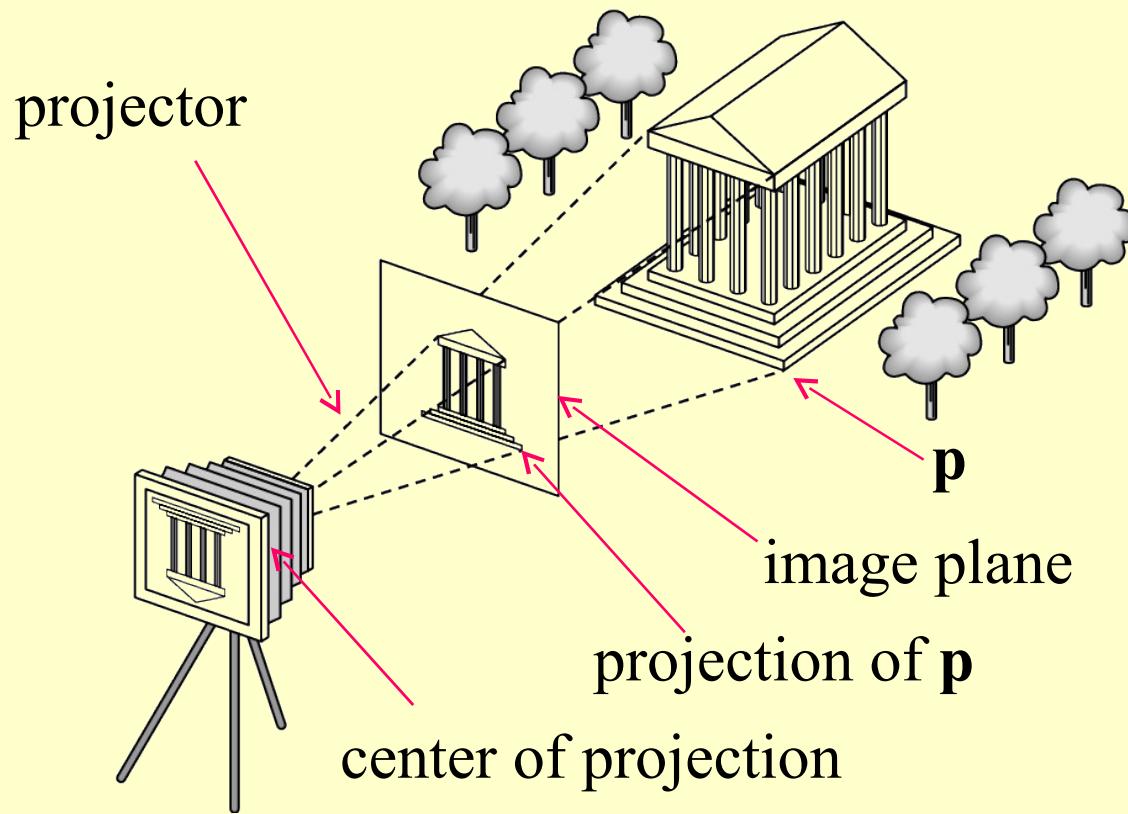
$$z_p = d$$



These are equations of simple perspective



Synthetic Camera Model





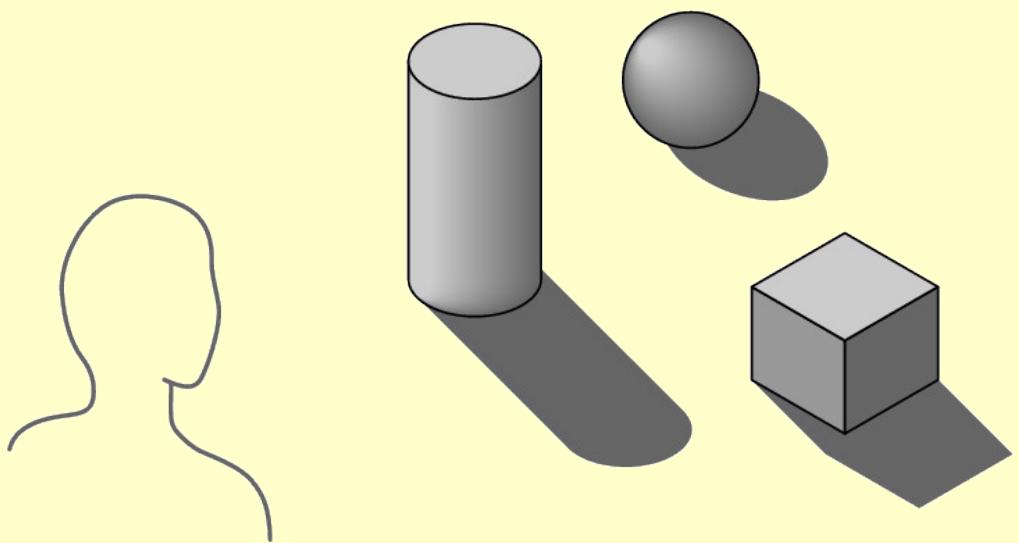
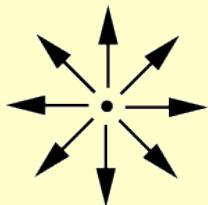
Advantages

- Separation of objects, viewer, light sources
- Two-dimensional graphics is a special case of three-dimensional graphics
- Leads to simple software API
 - ~~Specify objects, lights, camera, attributes~~
 - ~~Let implementation determine image~~
- Leads to fast hardware implementation



Global vs Local Lighting

- Cannot compute color or shade of each object independently
 - Some objects are blocked from light
 - Light can reflect from object to object
 - Some objects might be translucent





Why not ray tracing?

- Ray tracing seems more physically based so why don't we use it to design a graphics system?
- Possible and is actually simple for simple objects such as polygons and quadrics with simple point sources
- In principle, can produce global lighting effects such as shadows and multiple reflections but ray tracing is slow and not well-suited for interactive applications





Computer Graphics

Lecture 4 - Models and Architectures

John Shearer
Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Learn the basic design of a graphics system
- Introduce pipeline architecture
- Examine software components for an interactive graphics system



Image Formation Revisited

- Can we mimic the synthetic camera model to design graphics hardware software?
- Application Programmer Interface (API)
 - **Need only specify**
 - **Objects**
 - **Materials**
 - **Viewer**
 - **Lights**
- But how is the API implemented?



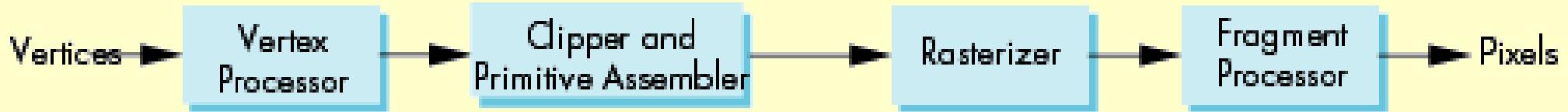
Physical Approaches

- Ray tracing: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
 - **Can handle global effects**
 - **Multiple reflections**
 - **Translucent objects**
 - **Slow**
 - **Must have whole data base available at all times**
- Radiosity: Energy based approach
 - **Very slow**



Practical Approach

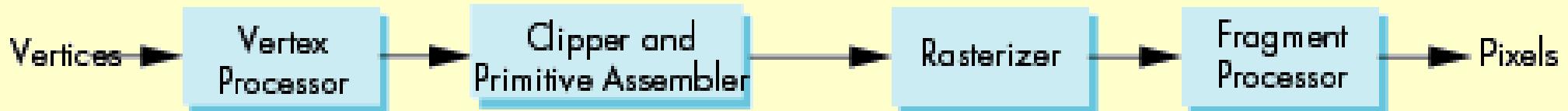
- Process objects one at a time in the order they are generated by the application
 - **Can consider only local lighting**
- Pipeline architecture
- All steps can be implemented in hardware on the graphics card





Vertex Processing

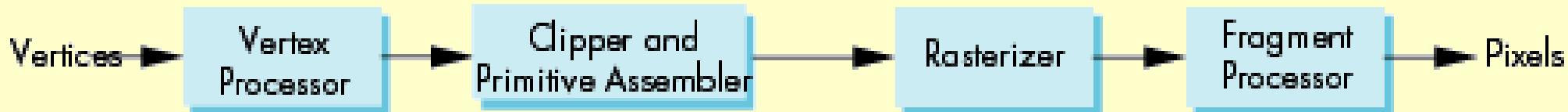
- Much of the work in the pipeline is in converting object representations from one coordinate system to another
 - **Object coordinates**
 - **Camera (eye) coordinates**
 - **Screen coordinates**
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors





Projection

- *Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image
 - **Perspective projections:** all projectors meet at the center of projection
 - **Parallel projection:** projectors are parallel, center of projection is replaced by a direction of projection





Primitive Assembly

Vertices must be collected into geometric objects before clipping and rasterization can take place

- Line segments
- Polygons
- Curves and surfaces

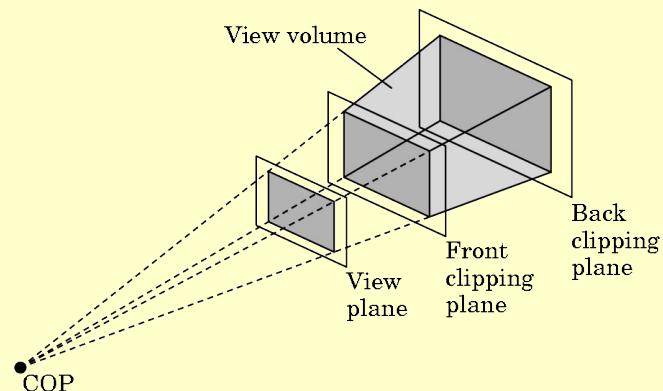
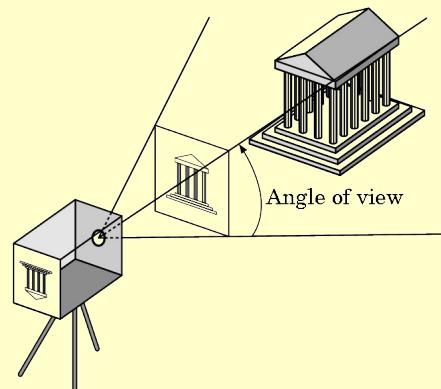




Clipping

Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space

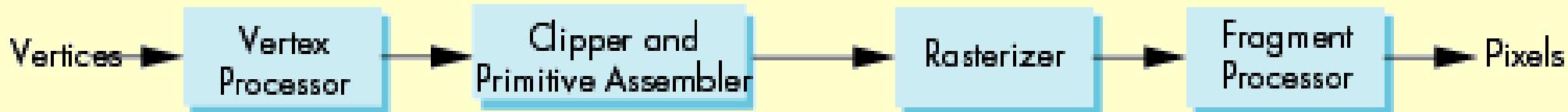
–**Objects that are not within this volume are said to be *clipped* out of the scene**





Rasterization

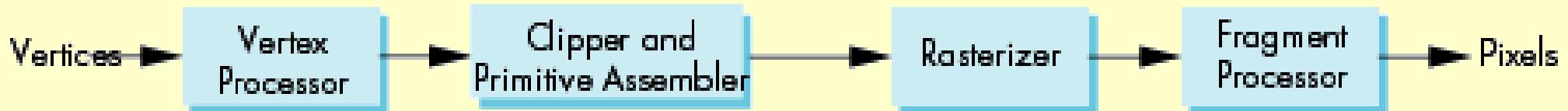
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
 - Fragments are “potential pixels”
 - **Have a location in frame buffer**
 - **Color and depth attributes**
 - Vertex attributes are interpolated over objects by the rasterizer





Fragment Processing

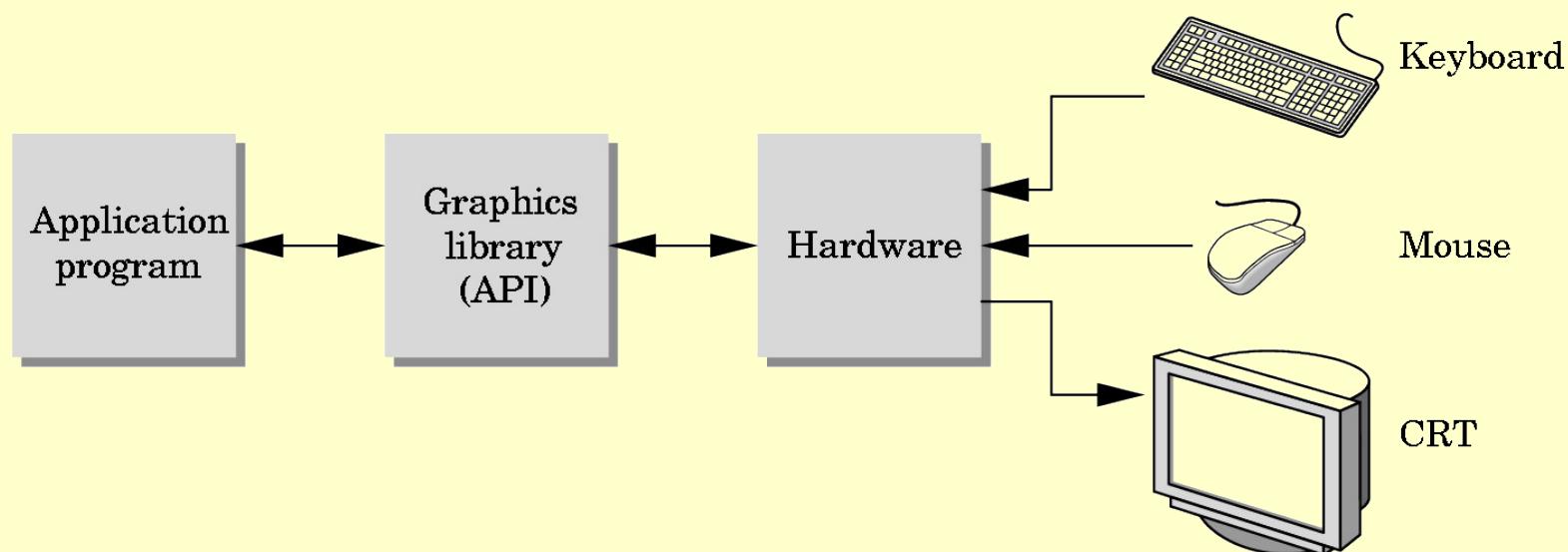
- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
 - **Hidden-surface removal**





The Programmer's Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)





API Contents

- Functions that specify what we need to form an image
 - **Objects**
 - **Viewer**
 - **Light Source(s)**
 - **Materials**
- Other information
 - **Input from devices such as mouse and keyboard**
 - **Capabilities of system**



Object Specification

- Most APIs support a limited set of primitives including
 - **Points (0D object)**
 - **Line segments (1D objects)**
 - **Polygons (2D objects)**
 - **Some curves and surfaces**
- **Quadratics**
- **Parametric polynomials**
- All are defined through locations in space or *vertices*



Example

type of object

location of vertex

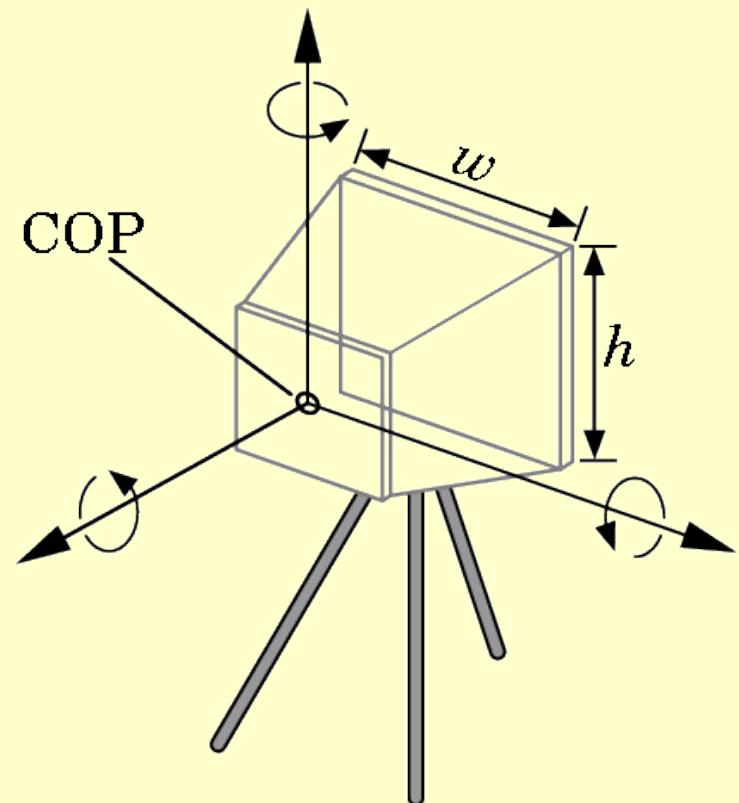
```
glBegin(GL_POLYGON);  
    glVertex3f(0.0, 0.0, 0.0);  
    glVertex3f(0.0, 1.0, 0.0);  
    glVertex3f(0.0, 0.0, 1.0);  
glEnd();
```

end of object definition



Camera Specification

- Six degrees of freedom
 - Position of center of lens
 - Orientation
- Lens
- Film size
- Orientation of film plane





Lights and Materials

- Types of lights
 - Point sources vs distributed sources
 - Spot lights
 - Near and far sources
 - Color properties
- Material properties
 - Absorption: color properties
 - Scattering
- Diffuse
- Specular





Additional reference

http://www.cs.dartmouth.edu/~fabio/teaching/cs52-winter08/lectures/12_Pipeline_Web.pdf



Computer Graphics

Lecture 5 - Programming with OpenGL

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Development of the OpenGL API
- OpenGL Architecture
 - OpenGL as a state machine
- Functions
 - Types
 - Formats
- Fundamental OpenGL primitives
- Attributes
- Simple program



Early History of APIs

- IFIPS (1973) formed two committees to come up with a standard graphics API
 - Graphical Kernel System (GKS)
 - 2D but contained good workstation model
 - Core
 - Both 2D and 3D
 - GKS adopted as ISO and later ANSI standard (1980s)
 - GKS not easily extended to 3D (GKS-3D)
 - Far behind hardware development



PHIGS and X

- Programmers Hierarchical Graphics System (PHIGS)
 - Arose from CAD community
 - Database model with retained graphics (structures)
- X Window System
 - DEC/MIT effort
 - Client-server architecture with graphics
- PEX combined the two
 - Not easy to use (all the defects of each)



SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To access the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications



OpenGL

The success of GL lead to OpenGL (1992), a platform-independent API that was

- Easy to use
- Close enough to the hardware to get excellent performance
- Focus on rendering
- Omitted windowing and input to avoid window system dependencies



OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
 - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,.....
 - Relatively stable
- Though recent changes have caused some disruption
- Evolution reflects new hardware capabilities
- **3D texture mapping and texture objects**
- **Vertex programs**
- Allows for platform specific features through extensions
- ARB replaced by Kronos



Kronos

- The Khronos Group was founded in January 2000 by a number of leading media-centric companies, including 3Dlabs, ATI, Discreet, Evans & Sutherland, Intel, NVIDIA, SGI and Sun Microsystems, dedicated to creating open standard APIs to enable the authoring and playback of rich media on a wide variety of platforms and devices.
- Standards include:
 - OpenGL (2D & 3D graphics)
 - COLLADA (digital asset exchange)
 - glFX (run-time effects API for OpenGL)
 - OpenGL ES (Embedded 3D graphics)
 - ...



OpenGL Libraries

- OpenGL core library
 - OpenGL32.dll on Windows
 - GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
 - Provides functionality in OpenGL core but avoids having to rewrite code
- Links with window system
 - GLX for X window systems
 - WGL for Windows
 - AGL for Macintosh

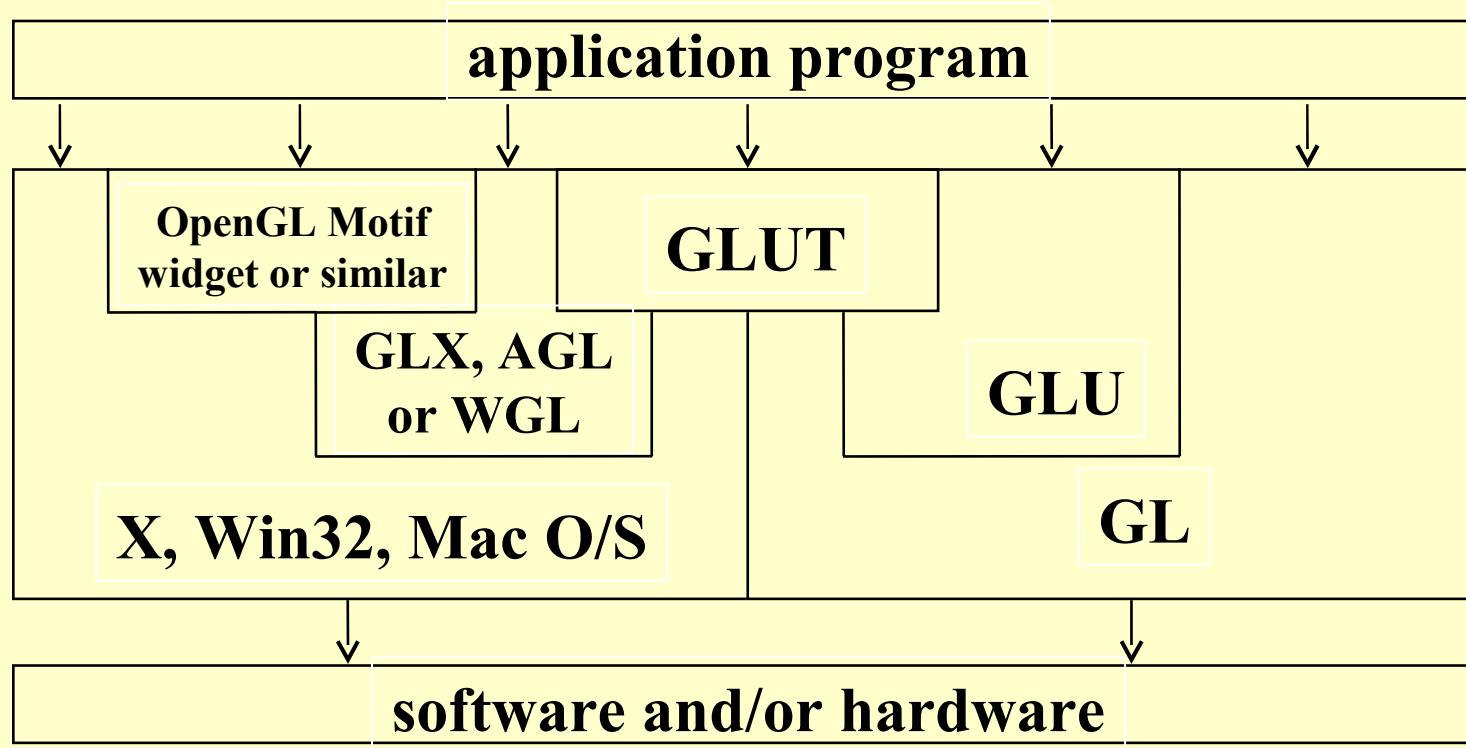


GLUT

- OpenGL Utility Toolkit (GLUT)
 - Provides functionality common to all window systems
 - Open a window
 - Get input from mouse and keyboard
 - Menus
 - Event-driven
 - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
 - No slide bars

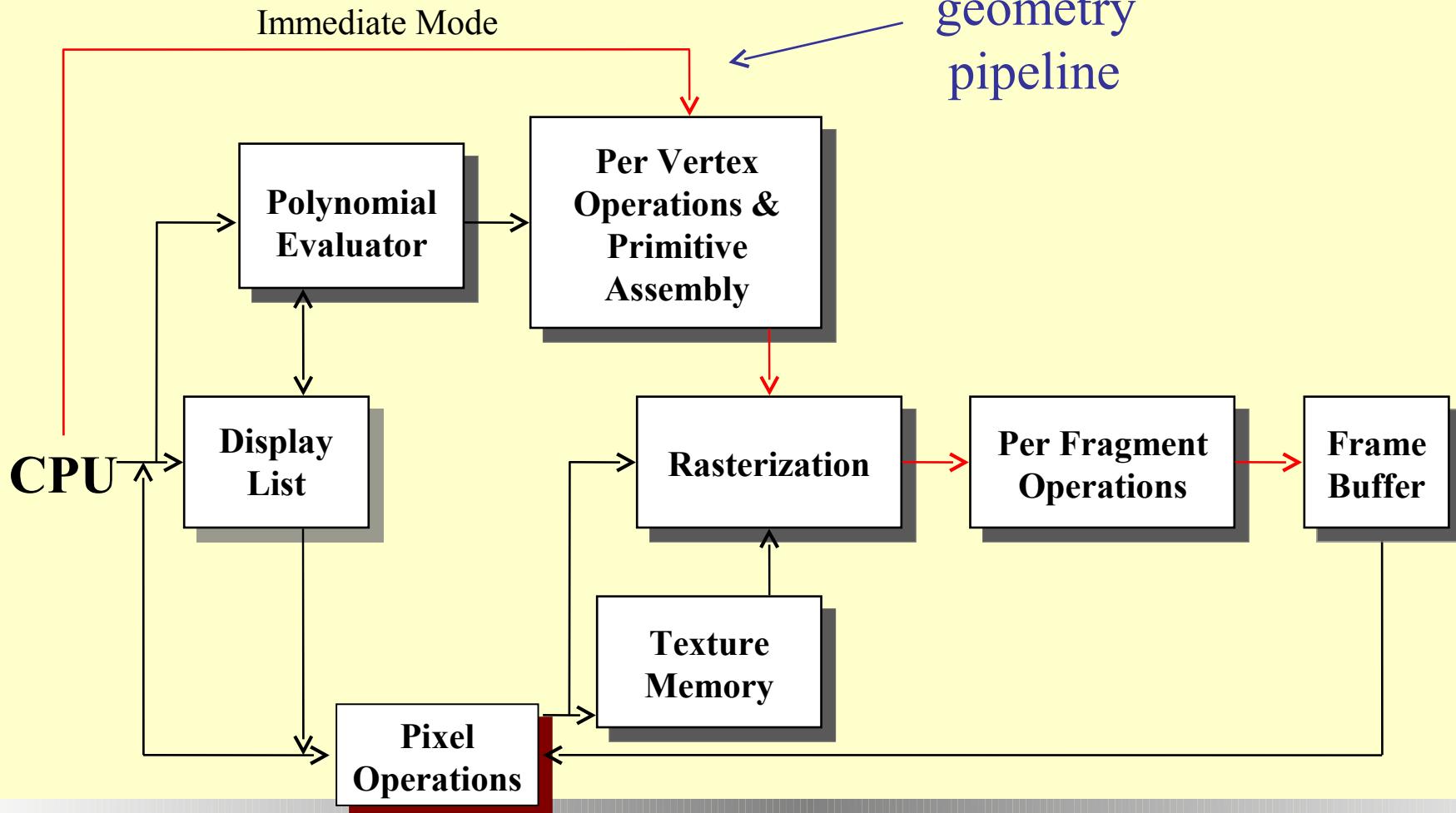


Software Organization





OpenGL Architecture





OpenGL Functions

- Primitives
 - Points
 - Line Segments
 - Polygons
- Attributes
- Transformations
 - Viewing
 - Modeling
- Control (GLUT / org.lwjgl.opengl)
- Input (GLUT / org.lwjgl.input)
- Query



OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
 - Primitive generating
 - Can cause output if primitive is visible
 - How vertices are processed and appearance of primitive are controlled by the state
 - State changing
 - Transformation functions
 - Attribute functions



Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function
 - `glVertex3f` – 3 FLOATS
 - `glVertex2i` – 2 INTEGERS
 - `glVertex3dv`
 - a vector (pointer to) 3 DOUBLES (in C/C++)
 - `glVertexPointer(int size, int stride, java.nio.FloatBuffer pointer)`
- Underlying storage mode is the same
- Easy to create overloaded functions but issue is efficiency
- The OpenGL API is a C API, but in practice most implementations (NVidia, AMD/ATI) are written in C++.



OpenGL function format

belongs to GL library

function name

dimensions

`glVertex3f(x, y, z)`

x, y, z are floats

`glVertex3fv(p)`

p is a pointer to an array



OpenGL defines

- Most constants are defined in the package **org.lwjgl.opengl.GL11**
- Examples
 - **glBegin(GL_POLYGON)**
 - **glClear(GL_COLOR_BUFFER_BIT)**
- include files or org.lwjgl.opengl.GL11 package in LWJGL, also define OpenGL data types:
GLfloat, **GLdouble**, ...



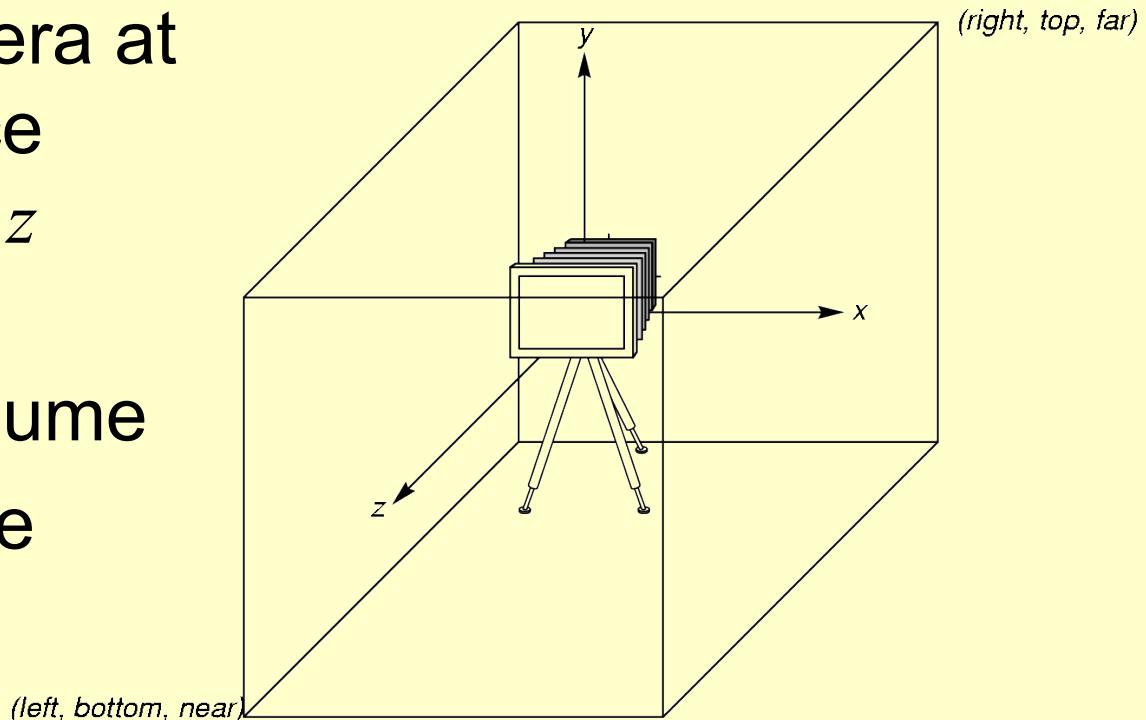
Coordinate Systems

- The units in `glVertex` are determined by the application and are called *object* or *problem coordinates*
- The viewing specifications are also in object coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to *camera (eye) coordinates* and later to *screen coordinates*
- OpenGL also uses some internal representations that usually are not visible to the application



OpenGL Camera

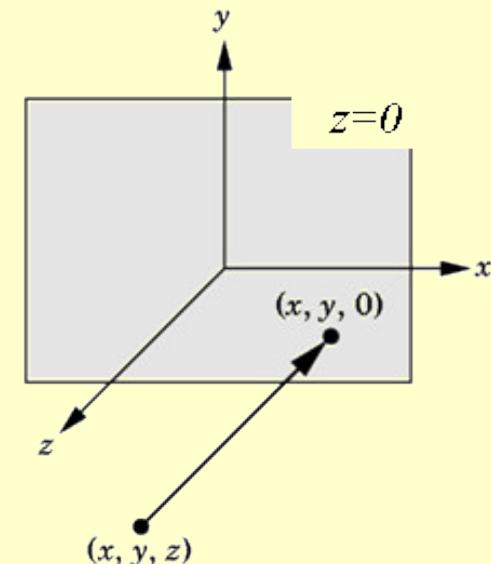
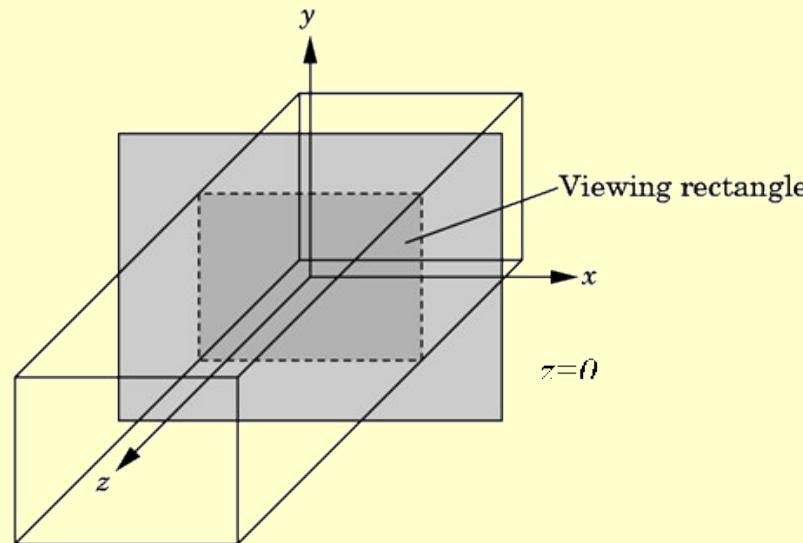
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2





Orthographic Viewing

- In the default orthographic view, points are
 - projected forward along the z axis onto the
 - plane $z=0$





Transformations and Viewing

- In OpenGL, projection is carried out by a projection matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first

```
glMatrixMode (GL_PROJECTION)
```

- Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume

```
glLoadIdentity();  
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```



Projection matrix & Modelview matrix

- OpenGL uses two matrices (projection & modelview) to make life simpler. A single matrix could be used (as two matrices can be multiplied together to create just one).
- In most scenarios the projection the user/programmer wants is rarely changed, so the projection matrix is used for this and isn't changed often
- The modelview matrix is used to transform the position, orientation, etc of objects in the real world and so is changed very frequently (when objects move, but more importantly, it is different for each object).



GL_PROJECTION

- `glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1, 1, -1, 1, -1.0, 1.0);
glTranslate(100, 100, 100);
glRotateF(45, 1, 0, 0);`
- Really means:
`GL_PROJECTION_MATRIX =
 IDENTITY
 * ORTHOGRAPHIC_MATRIX
 * TRANSLATION_MATRIX
 * ROTATION_MATRIX`



GL_MODELVIEW

- `glMatrixMode(GL_MODELVIEW);`
- `glLoadIdentity();`
`glTranslate(100, 100, 100);`
`glRotatef(45, 1, 0, 0);`
- `glPushMatrix();`
- `glTranslate(carx, cary, carz);`
// draw car here, by specifying various gl vertices
- `glPopMatrix();`
- `glPushMatrix();`
- `glTranslate(battleshipx, battleshipy, battleshipz);`
// draw battleship here, by specifying various gl vertices
- `glPopMatrix();`

- `car_vertex_matrix = projection_ortho_matrix * projection_translation * car_translation_matrix`
- `battleship_vertex_matrix = projection_ortho_matrix * projection_translation * battleship_translation_matrix`

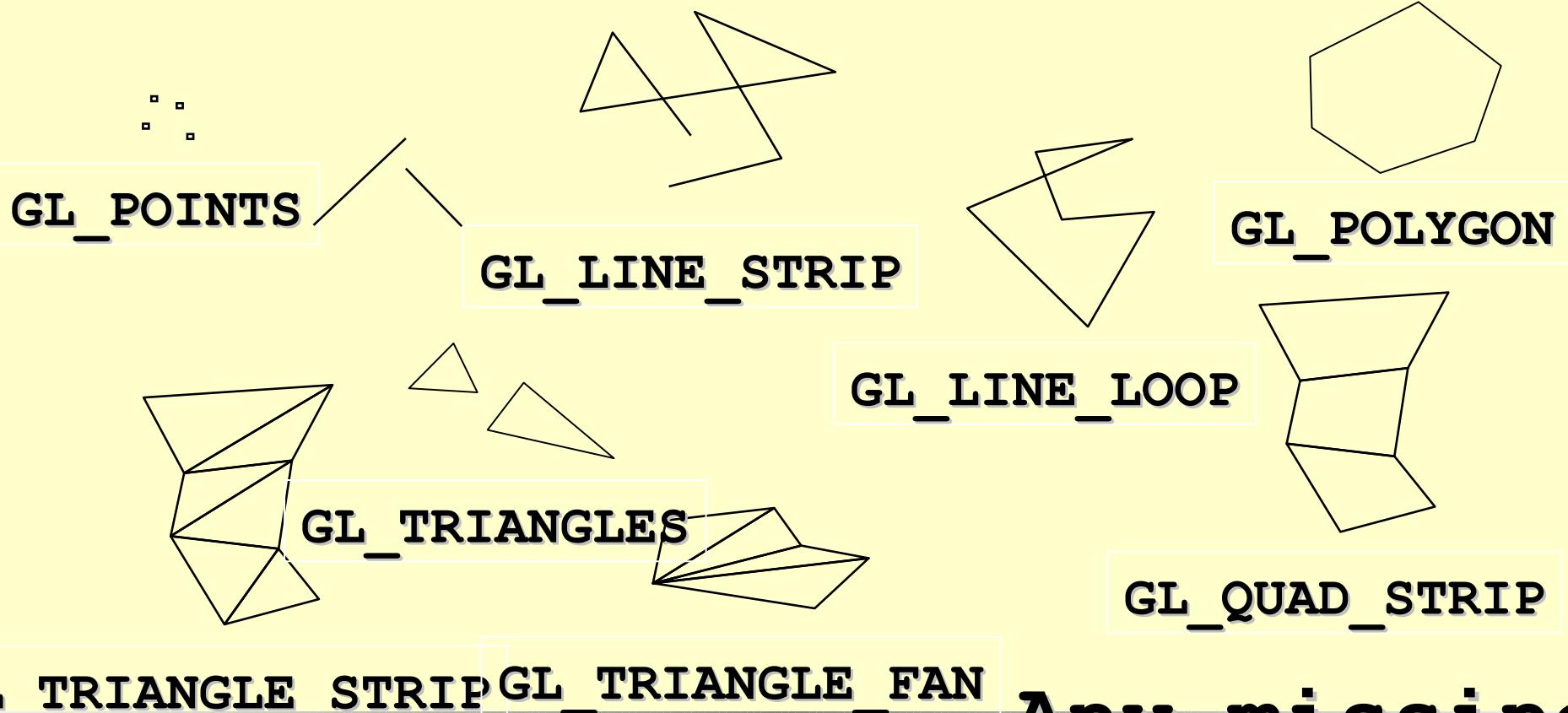


Two- and three-dimensional viewing

- In `glOrtho(left, right, bottom, top, near, far)` the near and far distances are measured from the camera
- Two-dimensional vertex commands place all vertices in the plane $z=0$
- If the application is in two dimensions, we can use the function
 - `gluOrtho2D(left, right, bottom, top)`
- In two dimensions, the view or clipping volume becomes a *clipping window*



OpenGL Primitives

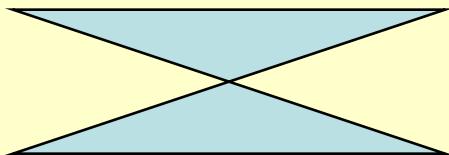


Any missing?

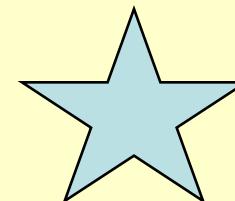


Polygon Issues

- OpenGL will only display polygons correctly that are
 - Simple: edges cannot cross
 - Convex: All points on line segment between two points in a polygon are also in the polygon
 - Flat: all vertices are in the same plane
- User program can check if above true
 - OpenGL will produce output if these conditions are violated but it may not be what is desired
- Triangles satisfy all conditions



nonsimple polygon



nonconvex polygon



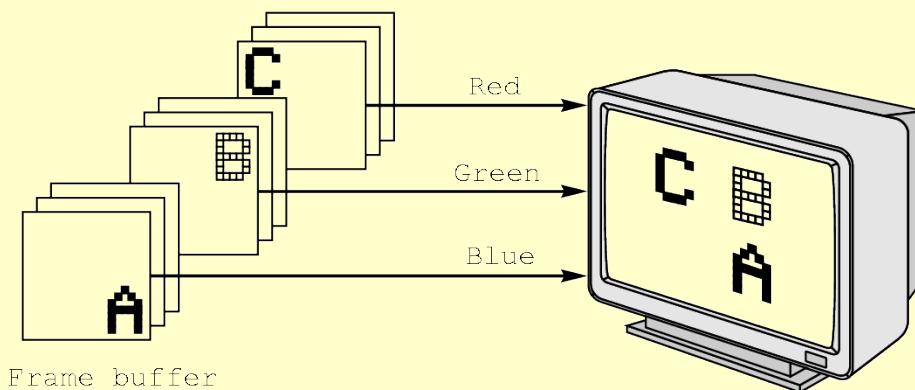
Attributes

- Attributes are part of the OpenGL state and determine the appearance of objects
 - Color (points, lines, polygons)
 - Size and width (points, lines)
 - Stipple pattern (lines, polygons)
 - Polygon mode
- Display as filled: solid color or stipple pattern
- Display edges
- Display vertices



RGB color

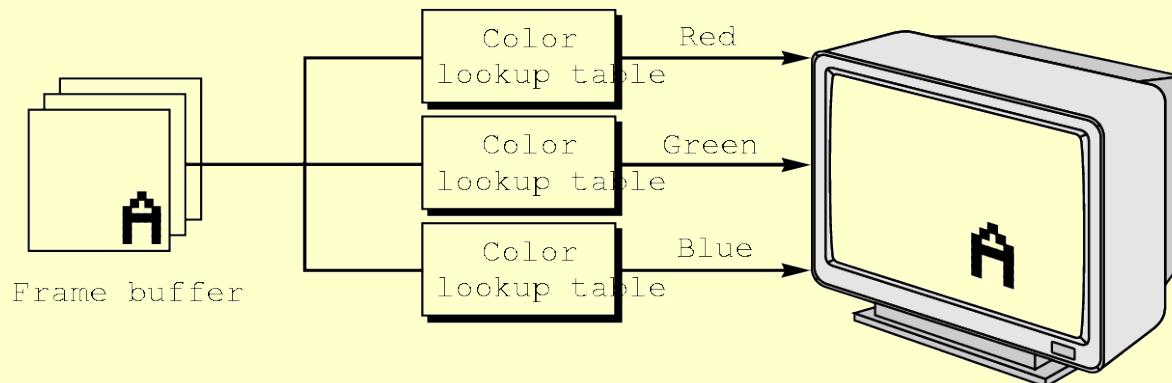
- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in `glColor3f` the color values range from 0.0 (none) to 1.0 (all), whereas in `glColor3ub` the values range from 0 to 255





Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
 - indices usually 8 bits
 - not as important now
- Memory inexpensive
- Need more colors for shading





Color and State

- The color as set by `glColor` becomes part of the state and will be used until changed
 - Colors and other attributes are not part of the object but are assigned when the object is rendered
- We can create conceptual *vertex colors* by code such as

`glColor`

`glVertex`

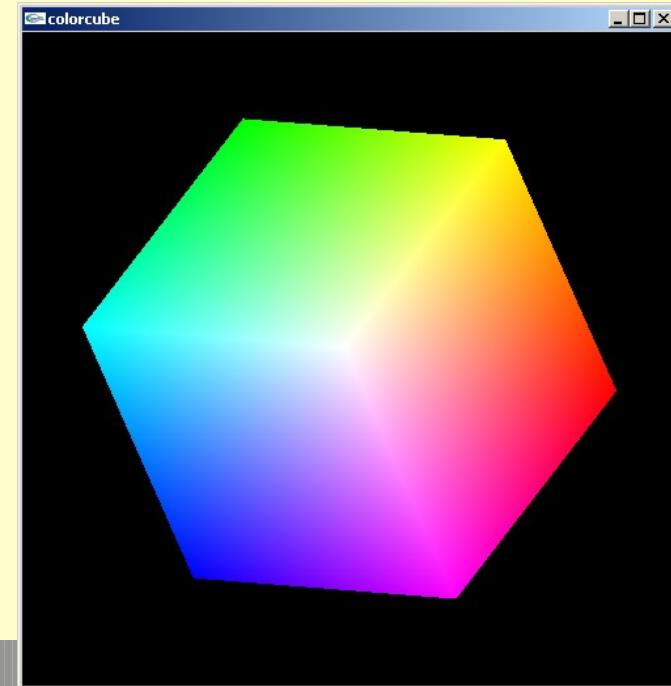
`glColor`

`glVertex`



Smooth Color

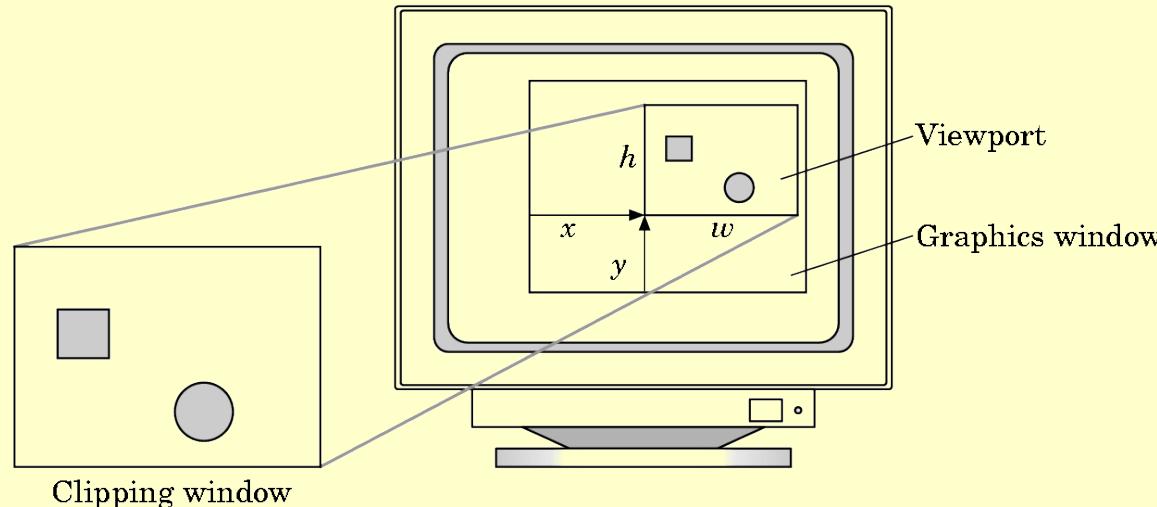
- Default is *smooth shading*
 - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
 - Color of first vertex determines fill color
- **glShadeModel**
(GL_SMOOTH)
or **GL_FLAT**





Viewports

- Do not have to use the entire window for the image:
glViewport(x, y, w, h)
- Values in pixels (screen coordinates)





Computer Graphics

Programming with OpenGL
Three Dimensions

Lecture 06

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Develop a more sophisticated example
 - ~~Sierpinski gasket: a fractal~~
- Think about extending that to 3D



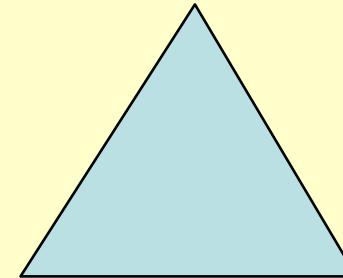
Three-dimensional Applications

- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
 - Not much changes
 - Use `glVertex3*` ()
 - Have to worry about the order in which polygons are drawn or use hidden-surface removal
 - Polygons should be simple, convex, flat

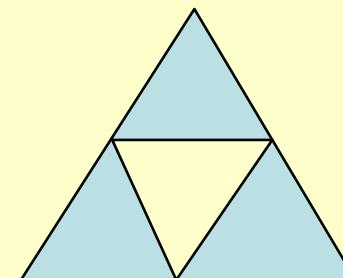


Sierpinski Gasket (2D)

- Start with a triangle



- Connect bisectors of sides and remove central triangle

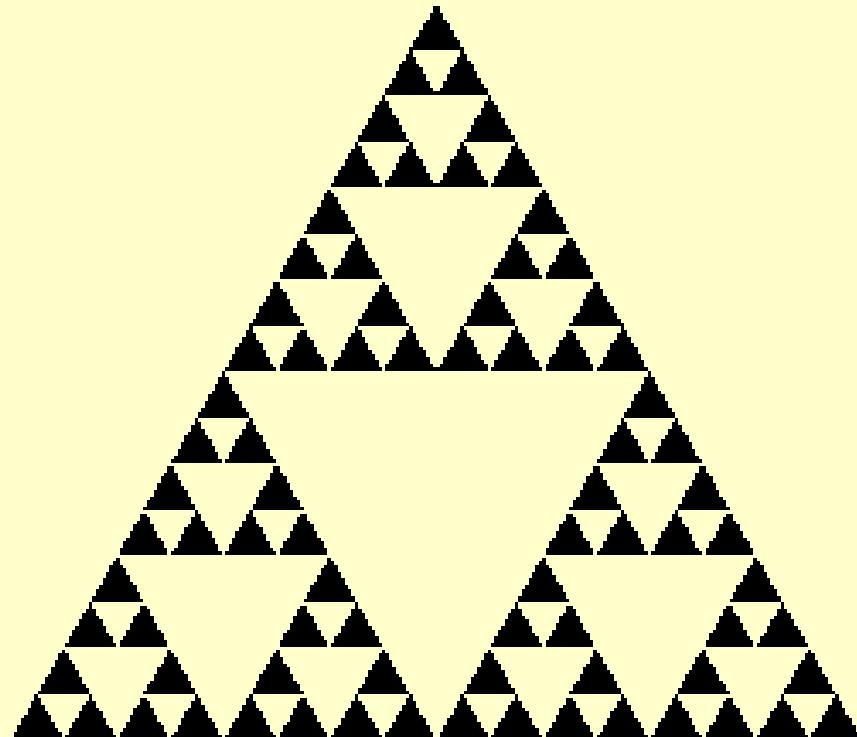


- Repeat



Example

- Five subdivisions





The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
 - the area goes to zero
 - but the perimeter goes to infinity
- This is not an ordinary geometric object
 - It is neither two nor three dimensional
- It is a *fractal* (fractional dimension) object



Gasket Program

- Can we write a program to draw Sierpinski Gasket (to a certain number of subdivisions)?
- Write a function to draw a triangle
 - ~~What parameters should it take?~~



Gasket Program

- First draw a triangle

```
private static void drawTriangle( float ax, float ay,
float bx, float by, float cx, float cy)
/* display one triangle */
{
    glBegin(GL_TRIANGLES);
        glVertex2f(ax, ay);
        glVertex2f(bx, by);
        glVertex2f(cx, cy);
    glEnd(); // Done Drawing
}
```



Gasket Program

- First draw a triangle

```
private static void drawTriangle( float ax, float ay, float bx, float  
by, float cx, float cy)  
/* display one triangle */  
{  
    GL11.glBegin(GL11.GL_TRIANGLES);  
        GL11 glVertex2f(ax, ay);  
        GL11 glVertex2f(bx, by);  
        GL11 glVertex2f(cx, cy);  
    GL11 glEnd(); // Done Drawing  
}
```

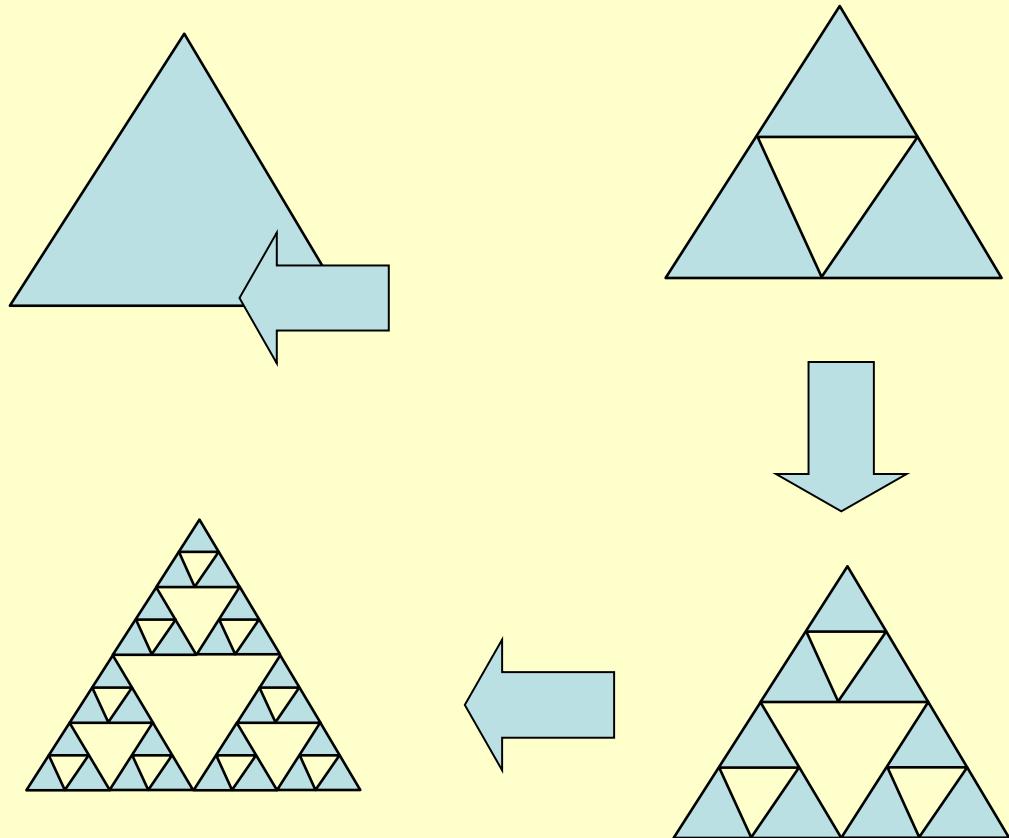
- We could pass in vectors for each to opengl rather than separate floats
 - How do you do this with LWJGL?



Triangle Subdivision 1

- pseudocode

- ...





Triangle Subdivision 2

pseudocode

```
divideTriangle(vector2f a,
   vector2f b, vector2f c, int
   m)
{
  // triangle subdivision
  using vertex numbers
  vector2f v0, v1, v2;
  if(m>0)
  {
    v0.x = (a.x + b.x) / 2;
    v0.y = (a.y + b.y) / 2;
    v1.x = (b.x + c.x) / 2;
    v1.y = (b.y + c.y) / 2;
    v2.x = (c.x + a.x) / 2;
    v2.y = (c.y + a.y) / 2;
    divide_triangle(a, v0, v1, m-1);
    divide_triangle(c, v1, v2, m-1);
    divide_triangle(b, v2, v0, m-1);
  }
  else(triangle(a,b,c));
  // draw triangle at end of recursion
}
```



Triangle Subdivision 3

pseudocode

```
main()
{
    create window etc.
    n=4;

    // setup projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-2.0, 2.0, -2.0,
    2.0);

    // prepare for drawing
    glMatrixMode(GL_MODELVIEW);
    glClearColor (1.0, 1.0,
    1.0,1.0)
    glColor3f(0.0,0.0,0.0);

    glClear(GL_COLOR_BUFFER_BIT)
    ;

    divideTriangle( a, b, c);
}
```



Efficiency Note 1

Having the `glBegin` and `glEnd` in the display callback rather than in the function `triangle` means we could call `glBegin` and `glEnd` only once for the entire gasket rather than once for each triangle



Moving to 3D

- Think about it for next lecture

~~How to we store the points?~~

~~What will we see using the above ?~~

~~gluOrtho2D(-2.0, 2.0, -2.0, 2.0)~~

~~Should we change this?~~



Computer Graphics

Programming with OpenGL
Three Dimensions – 2

Lecture 7

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Extend Sierpinski gasket to 3D
- Other ways of making Sierpinski gasket



Moving to 3D

- We can easily make the program three-dimensional by using
 - ~~glVertex3f~~
 - ~~glOrtho~~
- But that would not be very interesting
 - The gasket is still a 2D object
- Instead, we can start with a tetrahedron



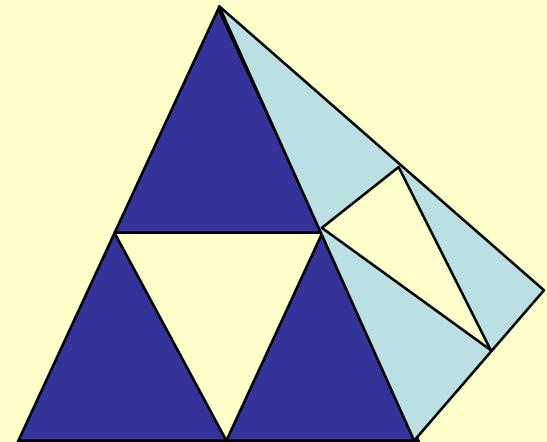
3D Sierpinski Gasket

- Tetrahedron / Triangle-based pyramid
 - Polyhedron composed of four triangular faces
 - Three of which meet at each vertex
- Either:
 - Slice through the tetrahedron with the points through the bisectors of each side
 - Subdivide each of the four faces



3D Sierpinski Gasket 2

- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra





divideTriangle

pseudocode

```
divideTriangle(vector2f a,
   vector2f b, vector2f c, int
   m)
{
    // triangle subdivision
    using vertex numbers
    vector2f v0, v1, v2;
    if(m>0)
    {
        v0.x = (a.x + b.x) / 2;
        v0.y = (a.y + b.y) / 2;
        v1.x = (b.x + c.x) / 2;
        v1.y = (b.y + c.y) / 2;
        v2.x = (c.x + a.x) / 2;
        v2.y = (c.y + a.y) / 2;    }

        divide_triangle(a, v0,
        v1, m-1);
        divide_triangle(c, v1,
        v2, m-1);
        divide_triangle(b, v2,
        v0, m-1);
    }
    else(triangle(a,b,c));
    // draw triangle at end of
    recursion
}
```



Subdividing triangular faces

- Use divideTriangle from 2D
- How to apply to a tetrahedron?



Subdividing triangular faces 2

- Use divideTriangle from 2D
- How to apply to a tetrahedron?

```
void tetrahedron( int m)
{
    glColor3f(1.0,0.0,0.0);
    divide_triangle(v[0], v[1], v[2], m);
    glColor3f(0.0,1.0,0.0);
    divide_triangle(v[3], v[2], v[1], m);
    glColor3f(0.0,0.0,1.0);
    divide_triangle(v[0], v[3], v[1], m);
    glColor3f(0.0,0.0,0.0);
    divide_triangle(v[0], v[2], v[3], m);
}
```

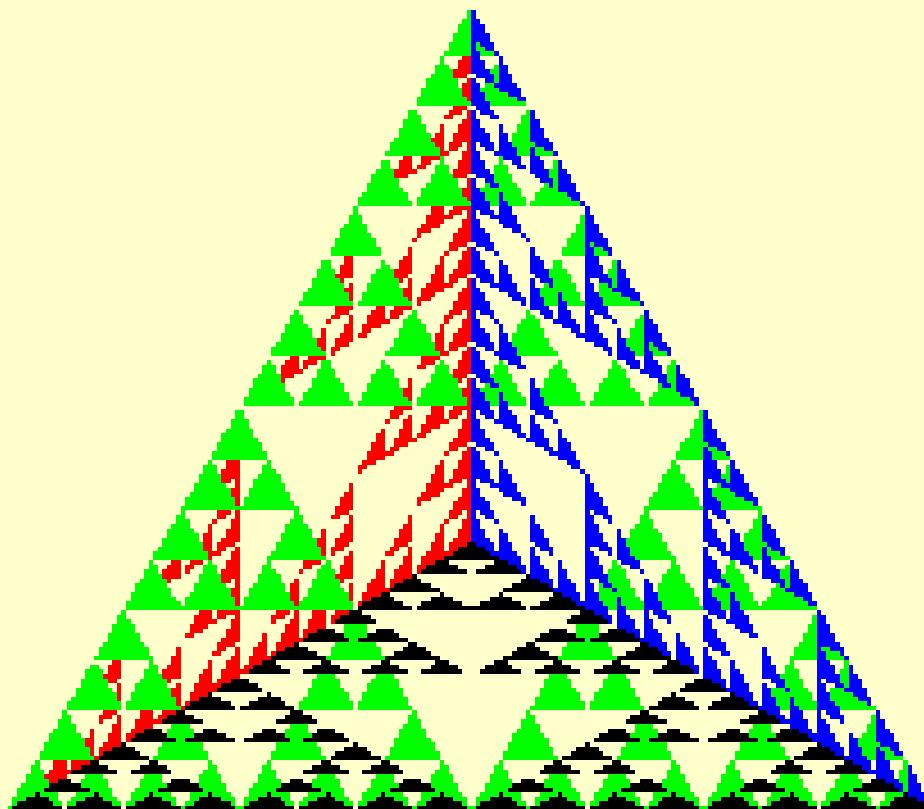


Almost Correct 1

- The triangles are drawn in the order they are defined in the program, so the front triangles are not always rendered in front of triangles behind them
- ...

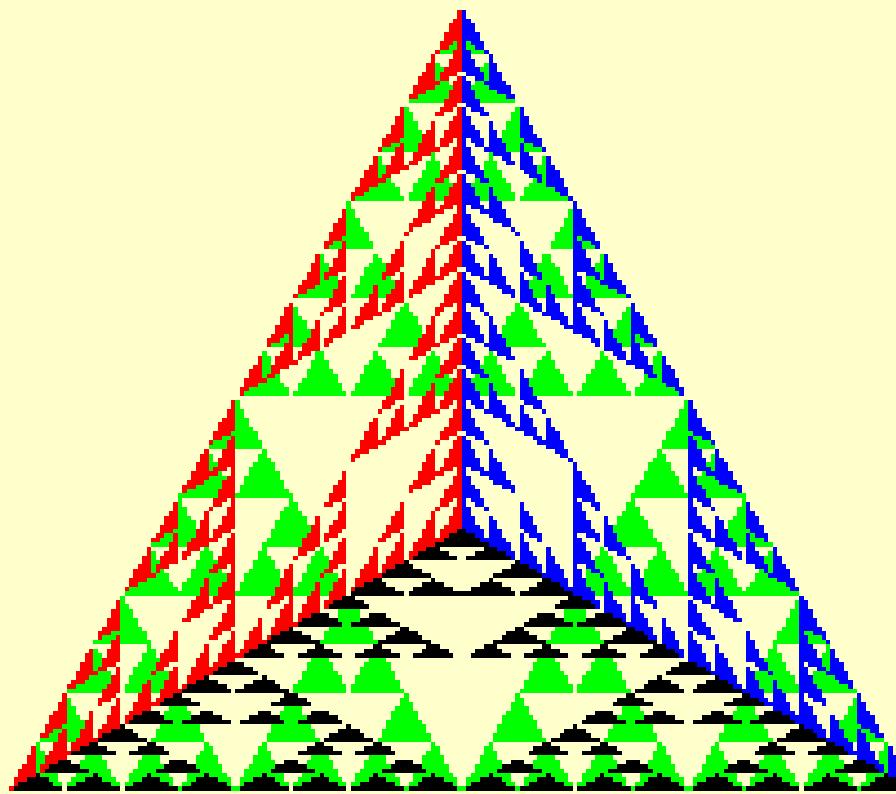


Almost Correct 2 – what we get





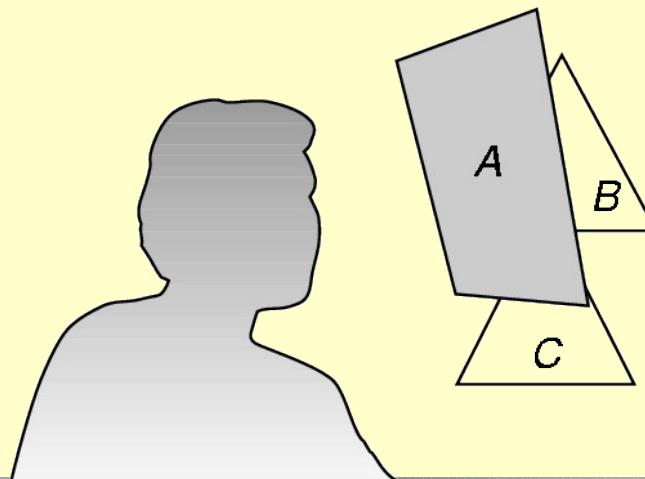
Almost Correct 3 – what we want





Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image





Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- It must be
 - Lwjgl have z-buffer enabled by default
 - Enabled in init
- ~~glEnable(GL_DEPTH_TEST)~~
 - Cleared in the display callback
- ~~glClear(GL_COLOR_BUFFER_BIT + GL_DEPTH_BUFFER_BIT)~~



Surface vs Volume Subdivision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a *volume* in the middle



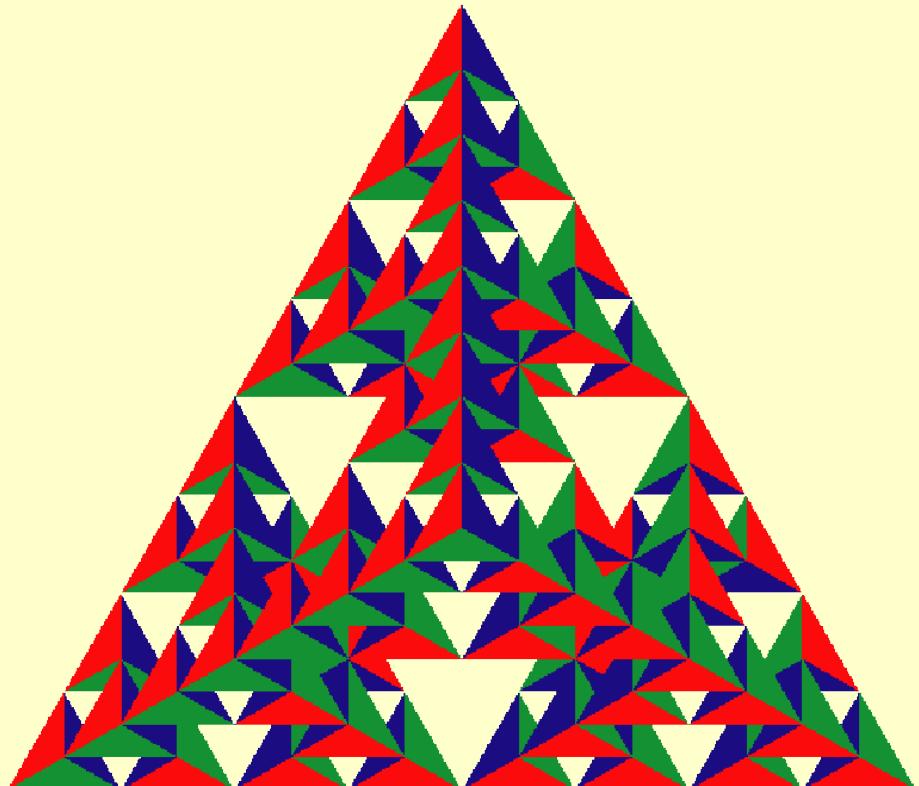
Volume Subdivision

- What functions do we need?



Volume Subdivision

- What functions do we need?
 - ~~drawTetrahedron~~
 - ~~divideTetrahedron~~





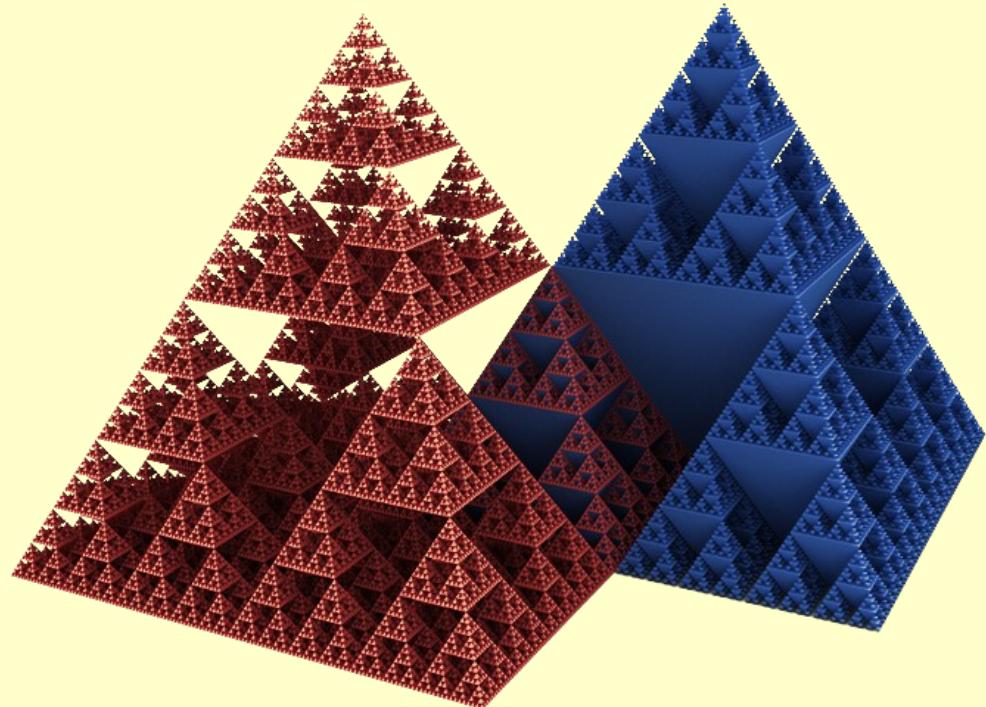
Back-face culling

- Determines whether a polygon of a graphical object is visible, depending on the position of the camera.
- Tests whether the points in the polygon appear in clockwise or counter-clockwise order when projected onto the screen.
- User specifies that front-facing polygons have a clockwise winding (or vice-versa), if the polygon projected on the screen has a counter-clockwise winding it has been rotated to face away from the camera and will not be drawn.
- Makes rendering objects quicker and more efficient by reducing the number of polygons for the program to draw



Square-based Sierpinski Gasket

- A Sierpiński square-based pyramid and its 'inverse'





Another way of making Sierpinski gasket

1. Take 3 points in a plane, and form a triangle
2. Randomly select any point inside the triangle and move half the distance from that point to any of the 3 vertex points. Plot the current position.
3. Repeat from step 2.

Write some pseudo code to draw the above

Would this work well in OpenGL?



Computer Graphics

Input and Interaction

Lecture 8

John Shearer

Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Introduce the basic input devices
 - ~~Physical Devices~~
 - ~~Logical Devices~~
 - ~~Input Modes~~
- Event-driven input
- Introduce double buffering for smooth animations



Project Sketchpad

- Ivan Sutherland (MIT 1963) established the basic interactive paradigm that characterizes interactive computer graphics:
 - User sees an *object* on the display
 - User points to (*picks*) the object with an input device (light pen, mouse, trackball)
 - Object changes (*moves, rotates, morphs*)
 - Repeat

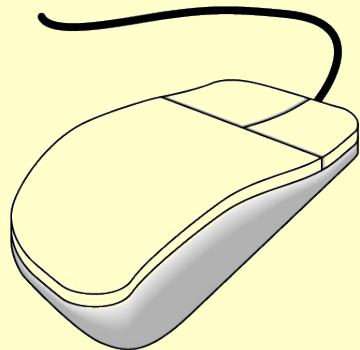


Graphical Input

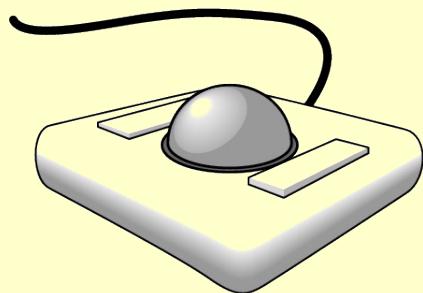
- Devices can be described either by
 - Physical properties
 - Mouse
 - Keyboard
 - Trackball
- Logical Properties
 - What is returned to program via API
 - A position
 - An object identifier
- Modes
 - How and when input is obtained
 - Request or event



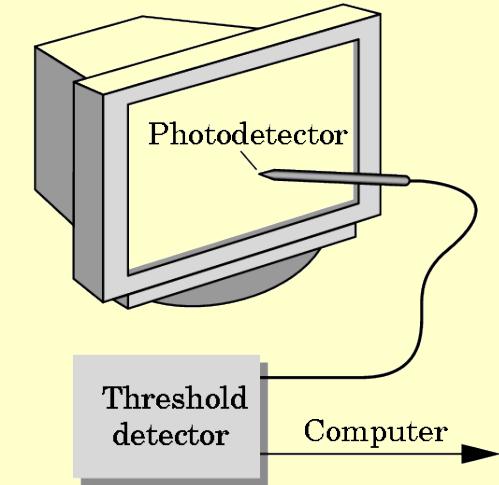
Physical Devices



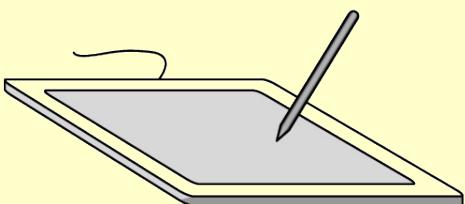
mouse



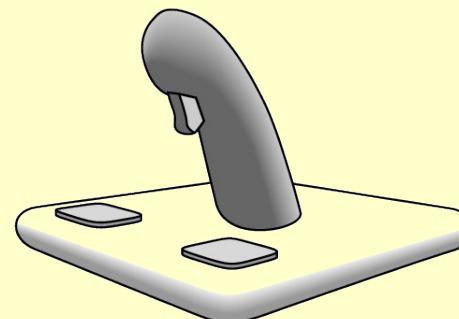
trackball



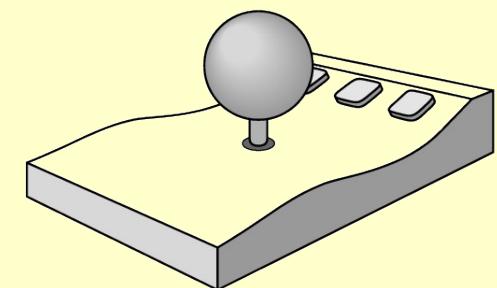
light pen



data tablet



joystick



space ball



Incremental (Relative) Devices

- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
 - Must integrate these inputs to obtain an absolute position
 - Rotation of cylinders in mouse
 - Roll of trackball
 - Difficult to obtain absolute position
 - Can get variable sensitivity



Logical Devices

- Consider the C and C++ code

- ~~- C++: `cin >> x;`~~

- ~~- C: `scanf ("%d", &x);`~~

- What is the input device?

- ~~- Can't tell from the code~~

- ~~- Could be keyboard, file, output from another program~~

- The code provides *logical input*

- ~~- A number (an `int`) is returned to the program regardless of the physical device~~



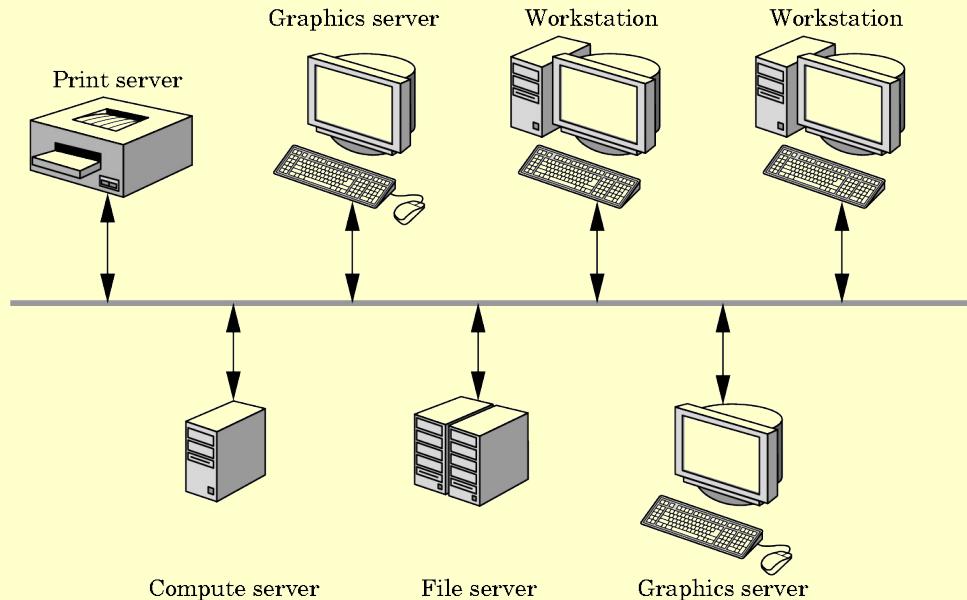
Graphical Logical Devices

- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
 - **Locator**: return a position
 - **Pick**: return ID of an object
 - **Keyboard**: return strings of characters
 - **Stroke**: return array of positions
 - **Valuator**: return floating point number
 - **Choice**: return one of n items



X Window Input

- The X Window System introduced a client-server model for a network of workstations
 - **Client:** OpenGL program
 - **Graphics Server:** bitmap display with a pointing device and a keyboard





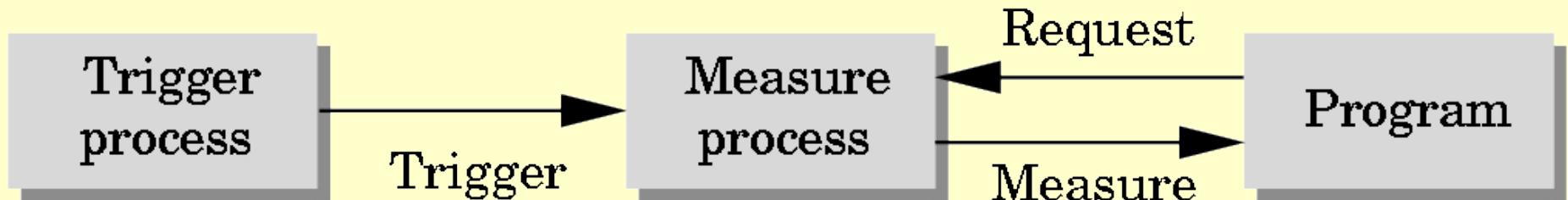
Input Modes

- Input devices contain a *trigger* which can be used to send a signal to the operating system
 - ~~Button on mouse~~
 - ~~Pressing or releasing a key~~
- When triggered, input devices return information (their *measure*) to the system
 - ~~Mouse returns position information~~
 - ~~Keyboard returns ASCII code~~



Request Mode

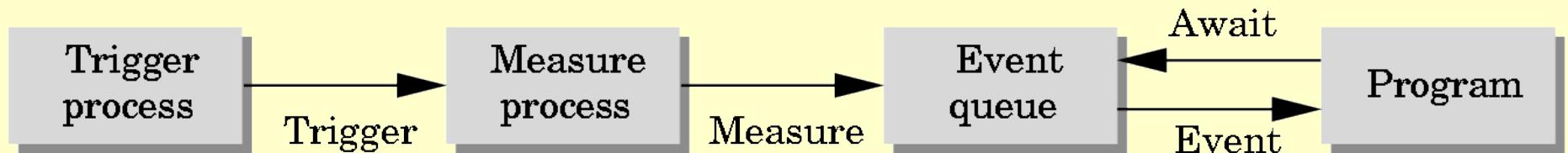
- Input provided to program only when user triggers the device
- Typical of keyboard input
 - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed





Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an *event* whose measure is put in an *event queue* which can be examined by the user program





Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: nonevent
 - Define what should be done if no other event is in queue



Callbacks

- Programming interface for event-driven input
- Define a *callback function* for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example: **glutMouseFunc (mymouse)**
- GLFW : <http://gpwiki.org/index.php/GLFW:Tutorials:Basics#Input>



Input in LWJGL

- Mouse and Keyboard can be accessed through their associated classes
 - `org.lwjgl.input.Keyboard` (used to poll the current state of the keys, or read all the keyboard presses / releases since the last read)
 - `org.lwjgl.input.Mouse` (used to poll the current state of the mouse buttons, and determine the mouse movement delta since the last poll)
- Controllers API in LWJGL is based on JInput
- While JInput provides access to controllers including keyboard and mouse that LWJGL does not support access to the keyboard and mouse via the controllers API.



Animating a Display

- When we redraw the display through the display callback, we usually start by clearing the window
 - ~~glClear()~~then draw the altered display
- Problem: the drawing of information in the frame buffer is decoupled from the display of its contents
 - ~~Graphics systems use dual ported memory (can be read while being written to)~~so we can see partially drawn display (except glfw & lwjgl defaults to double buffered so we don't see it, unless you force a single-buffered window)



Double Buffering

- Instead of one colour buffer, we use two
 - **Front Buffer**: one that is displayed but not written to
 - **Back Buffer**: one that is written to but not displayed
- `glutInitDisplayMode(GL_RGB | GL_DOUBLE)`
- LWJGL defaults to double buffered
- Could we have other numbers of colour buffers? 3, 4, 5, 6 ??? When? Why?
- At the end of the display callback buffers are swapped

```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT|....)

    /* draw graphics here */

    glutSwapBuffers() // C++ (GLUT)
    glfwSwapBuffers() // C++ (GLFW)
    GL11.update() // Java calls GL11.SwapBuffers() on our behalf
}
```



How might one animate a display?

- What would be involved?
- Can you think of any potential hazards?



Computer Graphics

Better Interactive Programs

Lecture 9

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Learn to build more sophisticated interactive programs using
 - Picking
 - Select objects from the display
 - Three methods
- Rubberbanding
 - Interactive drawing of lines and rectangles
- Display Lists
 - Retained mode graphics



Picking

- Identify a user-defined object on the display
- In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
- Practical difficulties
 - Pipeline architecture is feed forward, hard to go from screen back to world
 - Complicated by screen being 2D, world is 3D
 - How close do we have to come to object to say we selected it?



Three Approaches

- Hit list
 - Most general approach but most difficult to implement
- Use back or some other buffer to store object IDs as the objects are rendered
- Rectangular maps
 - Easy to implement for many applications
 - See paint program in text



Rendering Modes

- OpenGL can render in one of three modes selected by `glRenderMode(mode)`
 - `GL_RENDER`: normal rendering to the frame buffer (default)
 - `GL_FEEDBACK`: provides list of primitives rendered but no output to the frame buffer
 - `GL_SELECTION`: Each primitive in the view volume generates a *hit record* that is placed in a *name stack* which can be examined later



Selection Mode Functions

- **glSelectBuffer()**: specifies name buffer
- **glInitNames()**: initializes name buffer
- **glPushName(id)**: push id on name buffer
- **glPopName()**: pop top of name buffer
- **glLoadName(id)**: replace top name on buffer
- id is set by application program to identify objects



Using Selection Mode

- Initialize name buffer
- Enter selection mode (using mouse)
- Render scene with user-defined identifiers
- Reenter normal render mode
 - This operation returns number of hits
- Examine contents of name buffer (hit records)
- Hit records include id and depth information



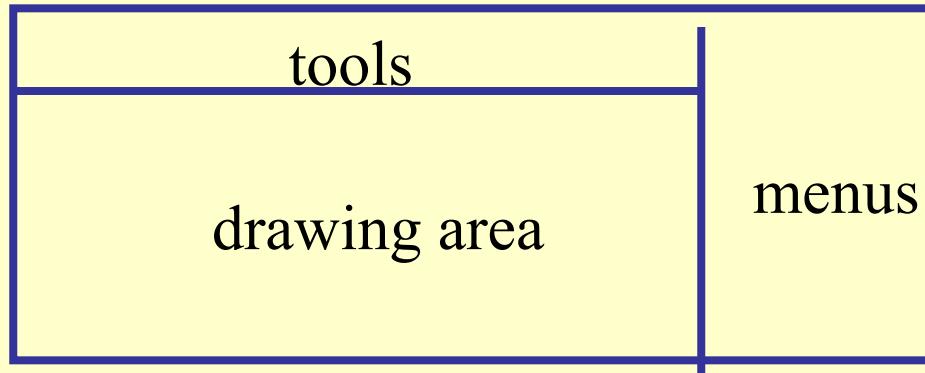
Selection Mode and Picking

- As we just described it, selection mode won't work for picking because every primitive in the view volume will generate a hit
- Change the viewing parameters so that only those primitives near the cursor are in the altered view volume
 - Use `gluPickMatrix` (see text for details)



Using Regions of the Screen

- Many applications use a simple rectangular arrangement of the screen
 - Example: paint/CAD program



- Easier to look at mouse position and determine which area of screen it is in than using selection mode picking



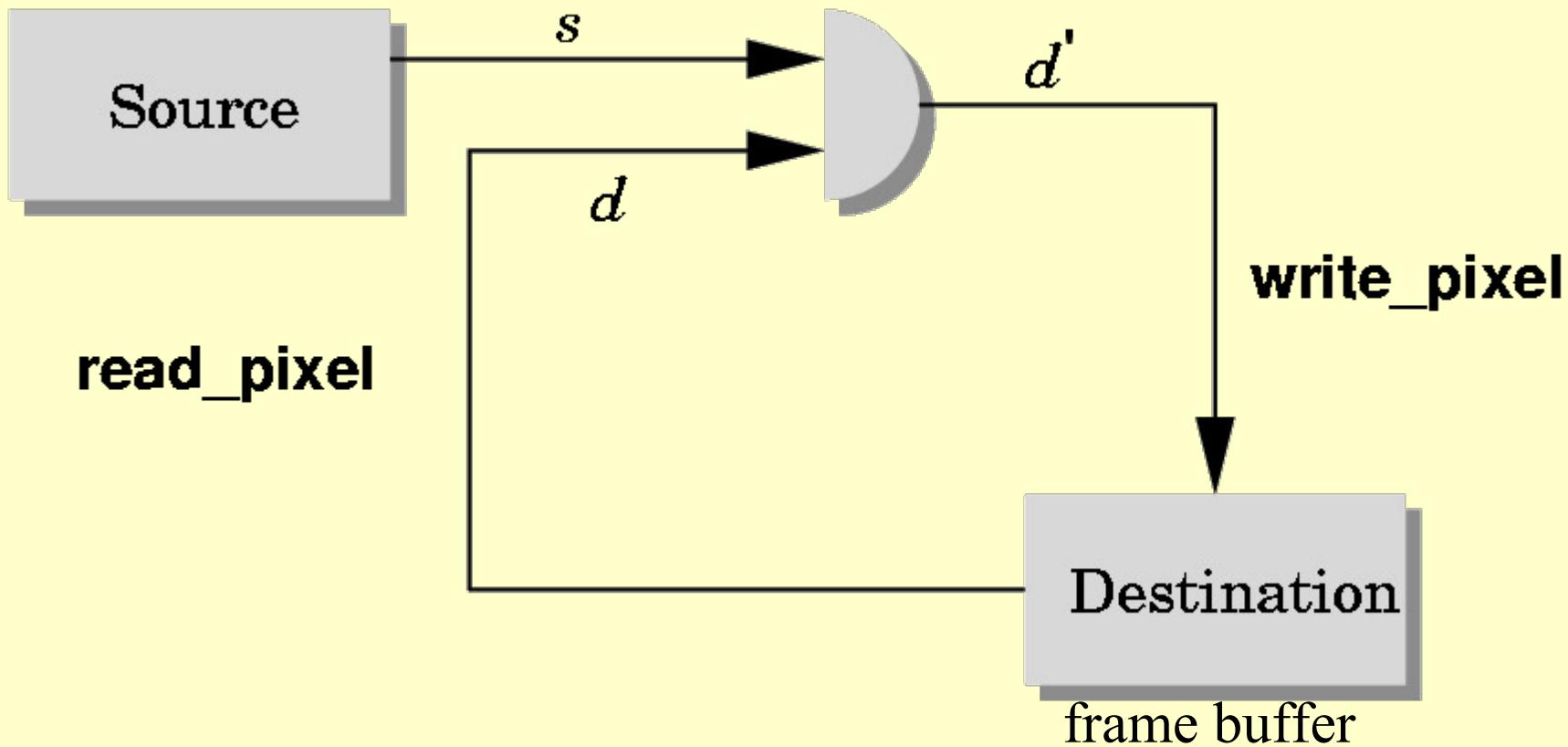
Using another buffer and colors for picking

- For a small number of objects, we can assign a unique color (often in color index mode) to each object
- We then render the scene to a color buffer other than the front buffer so the results of the rendering are not visible
- We then get the mouse position and use `glReadPixels()` to read the color in the buffer we just wrote at the position of the mouse
- The returned color gives the id of the object



Writing Modes

application





XOR write

- Usual (default) mode: source replaces destination ($d' = s$)
 - Cannot write temporary lines this way because we cannot recover what was “under” the line in a fast simple way
- Exclusive OR mode (XOR) ($d' = d \oplus s$)
 - $y \oplus x \oplus x = y$
 - Hence, if we use XOR mode to write a line, we can draw it a second time and line is erased!

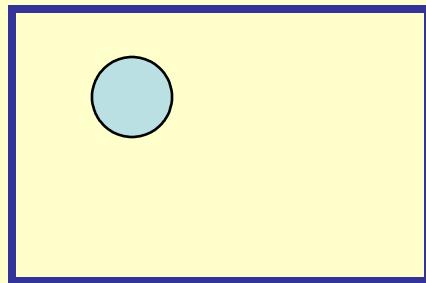


Rubberbanding

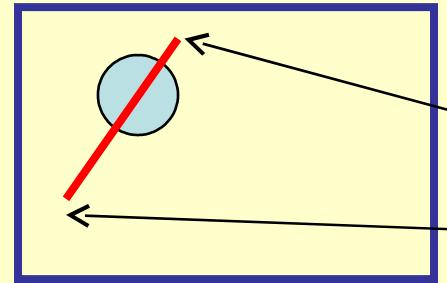
- Switch to XOR write mode
- Draw object
 - For line can use first mouse click to fix one endpoint and then use motion callback to continuously update the second endpoint
 - Each time mouse is moved, redraw line which erases it and then draw line from fixed first position to new second position
 - At end, switch back to normal drawing mode and draw line
 - Works for other objects: rectangles, circles



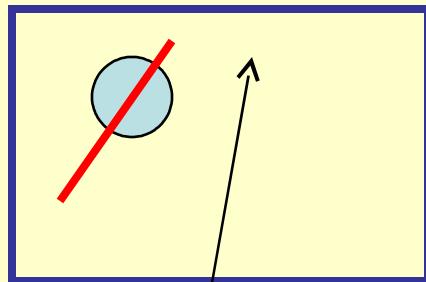
Rubberband Lines



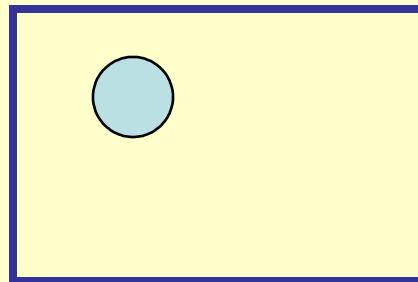
initial display



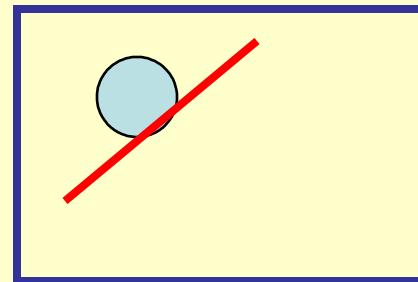
draw line with mouse
in XOR mode



mouse moved to
new position



original line redrawn
with XOR



new line drawn
with XOR



XOR in OpenGL

- There are 16 possible logical operations between two bits
- All are supported by OpenGL
 - Must first enable logical operations
 - `glEnable(GL_COLOR_LOGIC_OP)`
 - Choose logical operation
 - `glLogicOp(GL_XOR)`
 - `glLogicOp(GL_COPY)` (default)



A little more on glLogicOp

- glLogicOp specifies a logical operation that, when enabled, is applied between the incoming color index or RGBA color and the color index or RGBA color at the corresponding location in the frame buffer.
- Therefore, what ends up in the frame buffer is not necessarily the pixels we tried to draw –the frame buffer can be viewed as a form of memory, hence why one XOR followed by another has sufficient information to undo itself



A little more on glLogicOp 2



Immediate and Retained Modes

- Recall that in a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for it
 - Known as *immediate mode graphics*
 - Can be especially slow if the objects are complex and must be sent over a network
- Alternative is define objects and keep them in some form that can be redisplayed easily
 - *Retained mode graphics*
 - Accomplished in OpenGL via *display lists*



Display Lists

- Conceptually similar to a graphics file
 - Must define (name, create)
 - Add contents
 - Close
- In client-server environment, display list is placed on server
 - Can be redisplayed without sending primitives over network each time



Display List Functions

- Creating a display list

```
GLuint id;  
  
void init()  
{  
    id = glGenLists( 1 );  
  
    glNewList( id, GL_COMPILE );  
  
    /* other OpenGL routines */  
  
    glEndList();  
}
```

- Call a display list

```
void display()  
{  
    glCallList( id );  
}
```



Display Lists and State

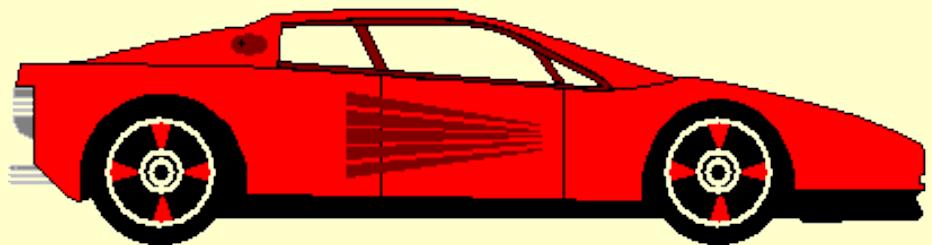
- Most OpenGL functions can be put in display lists
- State changes made inside a display list persist after the display list is executed
- Can avoid unexpected results by using `glPushAttrib` and `glPushMatrix` upon entering a display list and `glPopAttrib` and `glPopMatrix` before exiting



Hierarchy and Display Lists

- Consider model of a car
 - Create display list for chassis
 - Create display list for wheel

```
glNewList( CAR,  
GL_COMPILE );  
  
glCallList( CHASSIS );  
    glTranslatef( ... );  
    glCallList( WHEEL );  
    glTranslatef( ... );  
    glCallList( WHEEL );  
  
    ...  
glEndList();
```





Computer Graphics

Shading

Lecture 13

John Shearer

Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Learn to shade objects so their images appear three-dimensional
- Introduce the types of light-material interactions
- Build a simple reflection model---the Phong model--- that can be used with real time graphics hardware
- Introduce modified Phong model
- Consider computation of required vectors
- Introduce the OpenGL shading functions
- Discuss polygonal shading
 - Flat
 - Smooth
 - Gouraud



Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with `glColor`. We get something like
- But we want



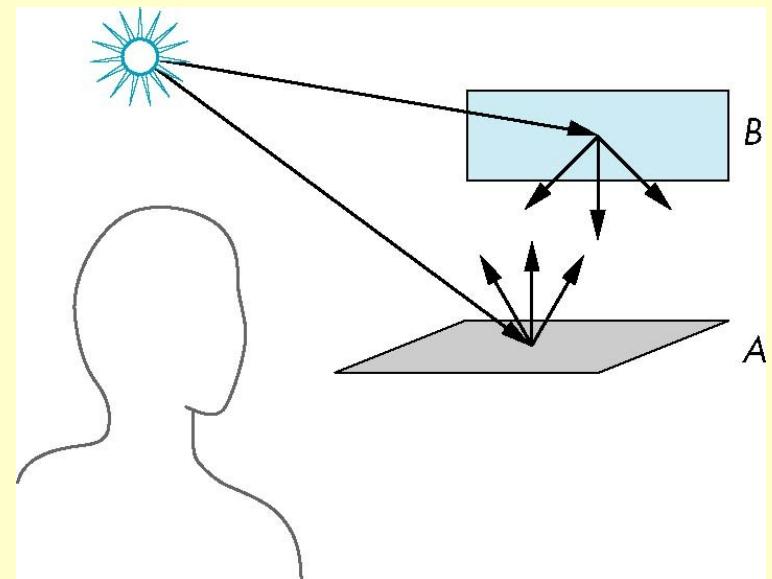
Shading

- Why does the image of a real sphere look like
- Light-material interactions cause each point to have a different color or shade
- Need to consider
 - Light sources
 - Material properties
 - Location of viewer
 - Surface orientation



Scattering

- Light strikes A
 - Some scattered
 - Some absorbed
- Some of scattered light strikes B
 - Some scattered
 - Some absorbed
- Some of this scattered light strikes A
 - and so on



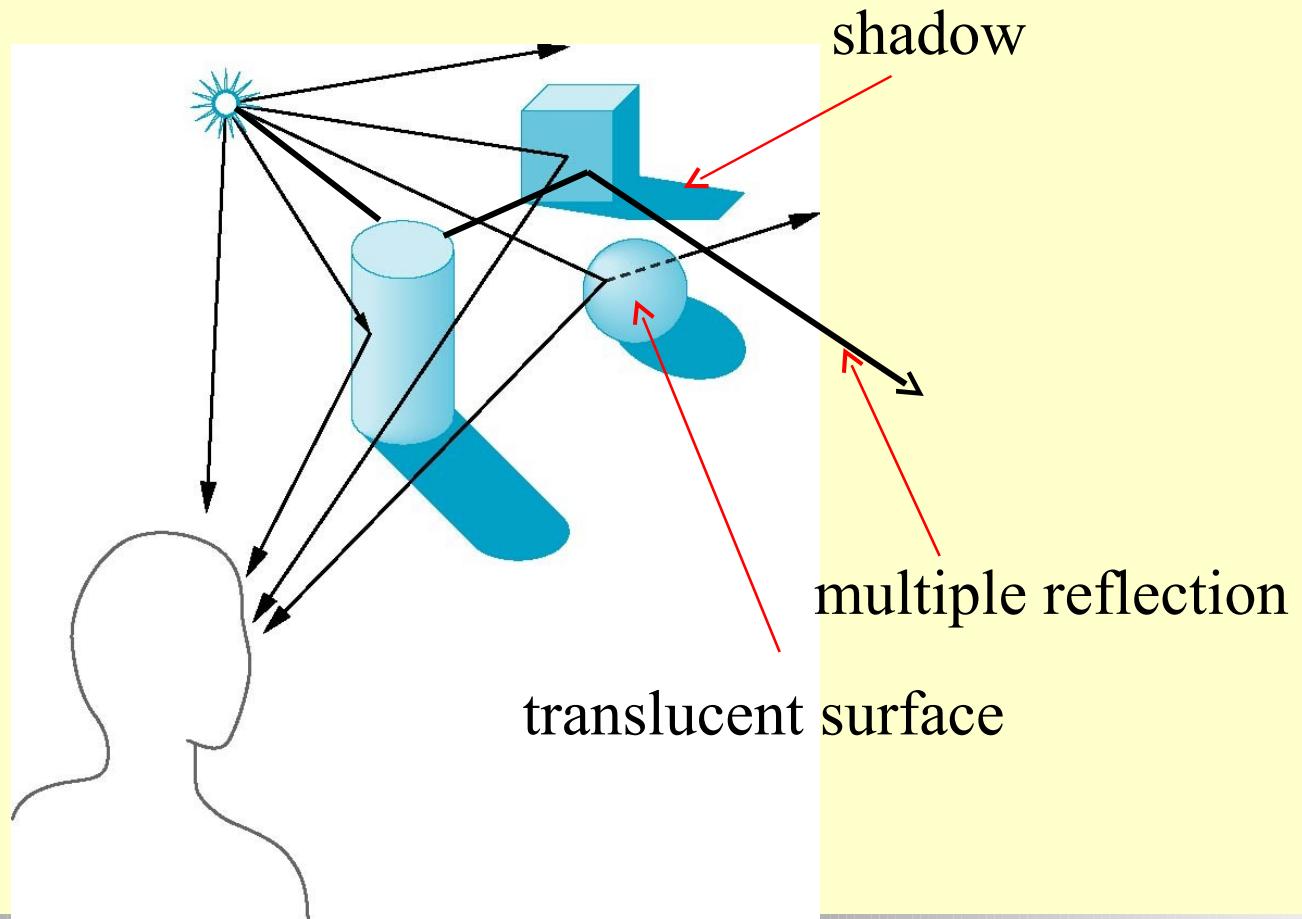


Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*
 - Cannot be solved in general
 - Ray tracing is a special case for perfectly reflecting surfaces
- Rendering equation is global and includes
 - Shadows
 - Multiple scattering from object to object



Global Effects





Local vs Global Rendering

- Correct shading requires a global calculation involving all objects and light sources
 - Incompatible with pipeline model which shades each polygon independently (local rendering)
- However, in computer graphics, especially real time graphics, we are happy if things “look right”
 - Exist many techniques for approximating global effects



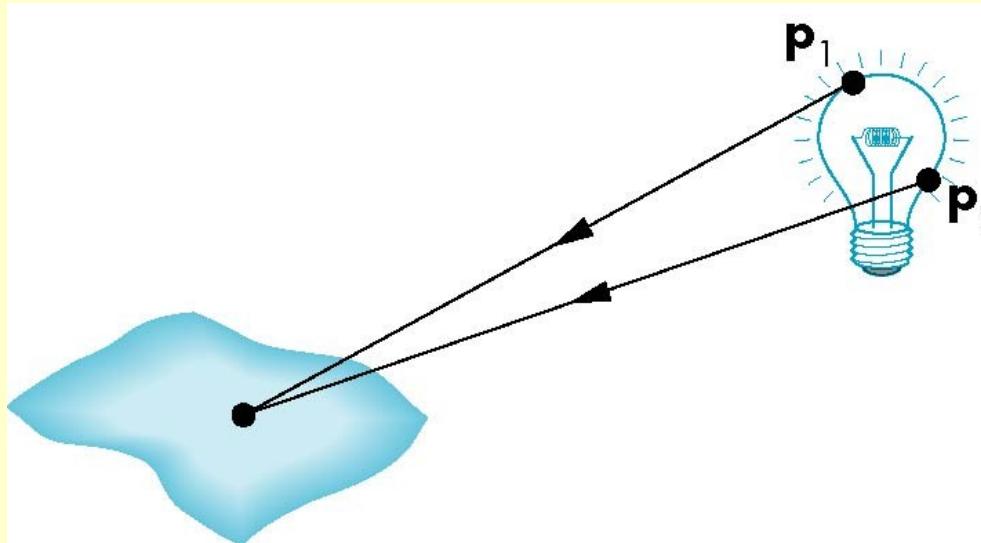
Light-Material Interaction

- Light that strikes an object is partially absorbed and partially scattered (reflected)
- The amount reflected determines the color and brightness of the object
 - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
- The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface



Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source





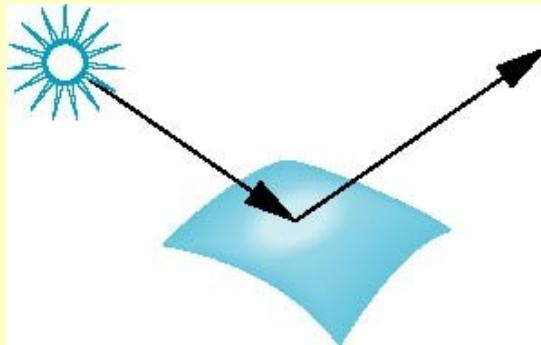
Simple Light Sources

- Point source
 - Model with position and color
 - Distant source = infinite distance away (parallel)
- Spotlight
 - Restrict light from ideal point source
- Ambient light
 - Same amount of light everywhere in scene
 - Can model contribution of many sources and reflecting surfaces

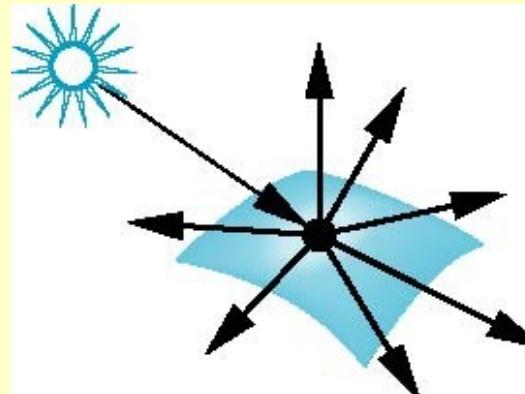


Surface Types

- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflect the light
- A very rough surface scatters light in all directions



smooth surface



rough surface



Phong Model

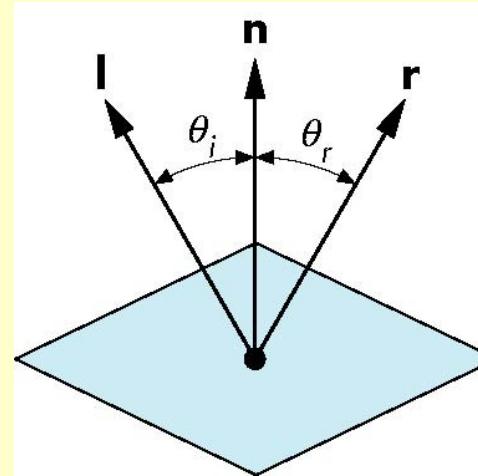
- A simple model that can be computed rapidly
- Has three components
 - Diffuse
 - Specular
 - Ambient
- Uses four vectors
 - To source
 - To viewer
 - Normal
 - Perfect reflector



Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of relection
- The three vectors must be coplanar

$$\mathbf{r} = 2 (\mathbf{l} \cdot \mathbf{n}) \mathbf{n} - \mathbf{l}$$





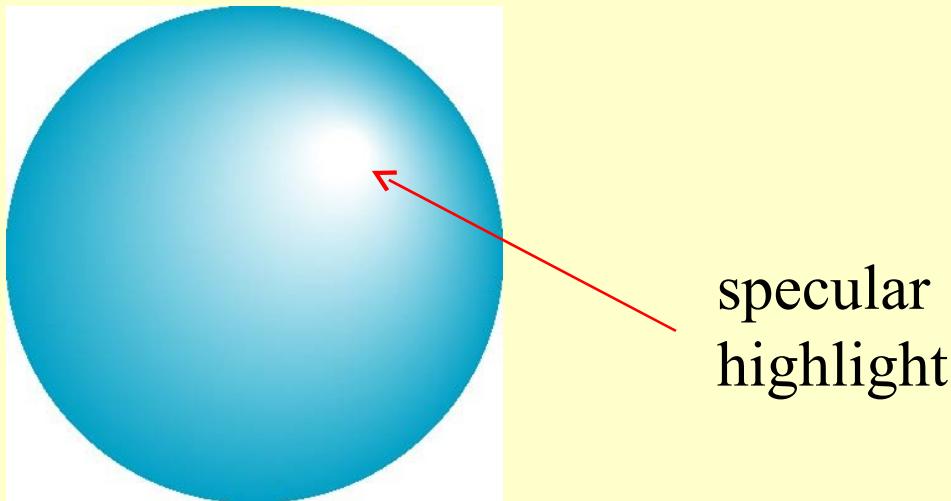
Lambertian Surface

- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
 - ~~reflected light $\sim \cos \theta_i$~~
 - ~~$\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized~~
 - ~~There are also three coefficients, k_r, k_b, k_g that show how much of each color component is reflected~~



Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



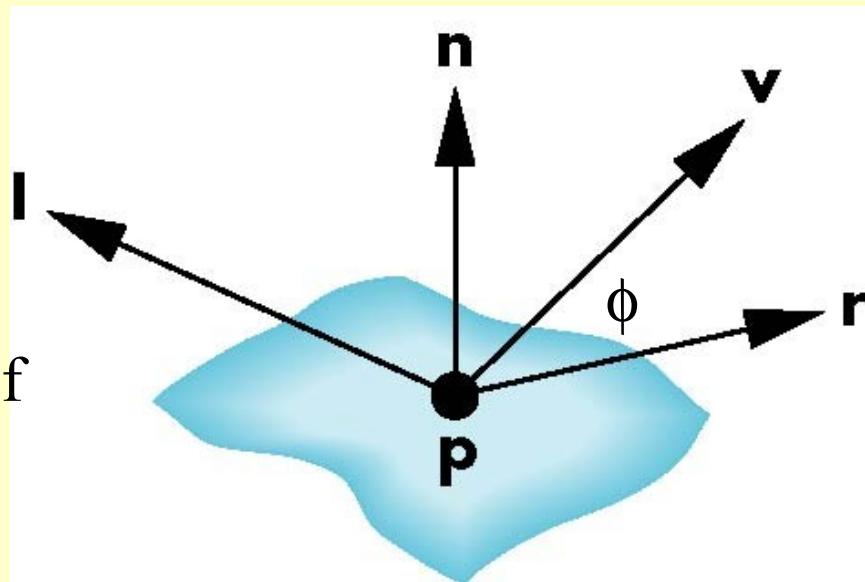


Modeling Specular Reflections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

$$I_r \sim k_s I \cos^\alpha \phi$$

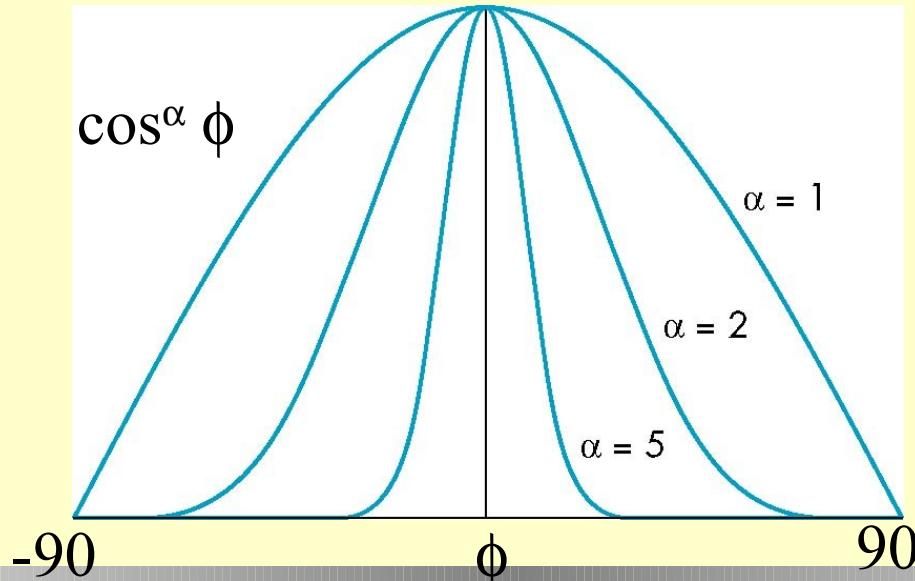
reflected intensity incoming intensity
absorption coef shininess coef





The Shininess Coefficient

- Values of α between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic

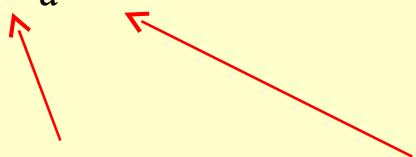




Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
- Amount and color depend on both the color of the light(s) and the material properties of the object
- Add $k_a I_a$ to diffuse and specular terms

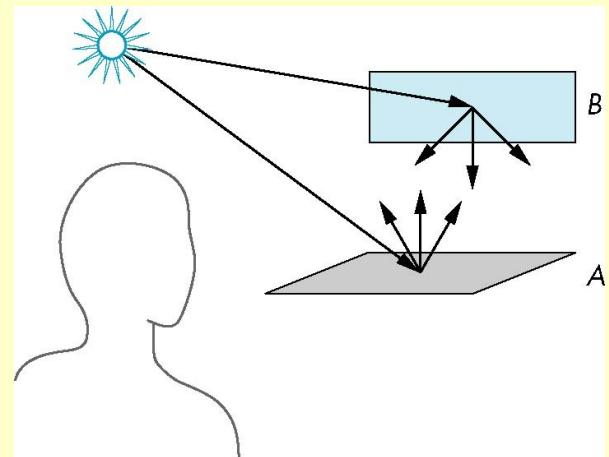
reflection coef intensity of ambient light





Distance Terms

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them
- We can add a factor of the form $1/(a + bd + cd^2)$ to the diffuse and specular terms
- The constant and linear terms soften the effect of the point source





Light Sources

- In the Phong Model, we add the results from each light source
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification
 - Separate red, green and blue components
 - Hence, 9 coefficients for each point source
 - I_{dr} , I_{dg} , I_{db} , I_{sr} , I_{sg} , I_{sb} , I_{ar} , I_{ag} , I_{ab}



Material Properties

- Material properties match light source properties
 - ~~Nine absorption coefficients~~
 - $k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}$
 - ~~Shininess coefficient α~~

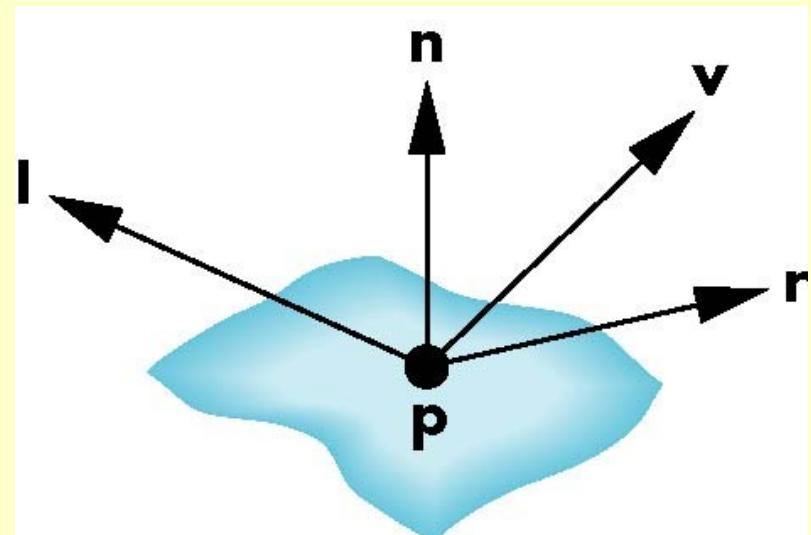


Adding up the Components

For each light source and each color 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 color component we add contributions from all sources





Modified Phong Model

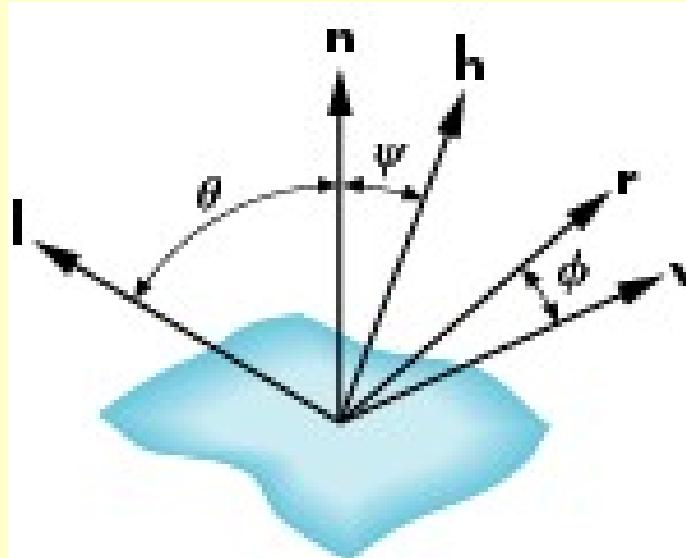
- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex
- Blinn suggested an approximation using the halfway vector that is more efficient



The Halfway Vector

- \mathbf{h} is normalized vector halfway between \mathbf{l} and \mathbf{v}

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$





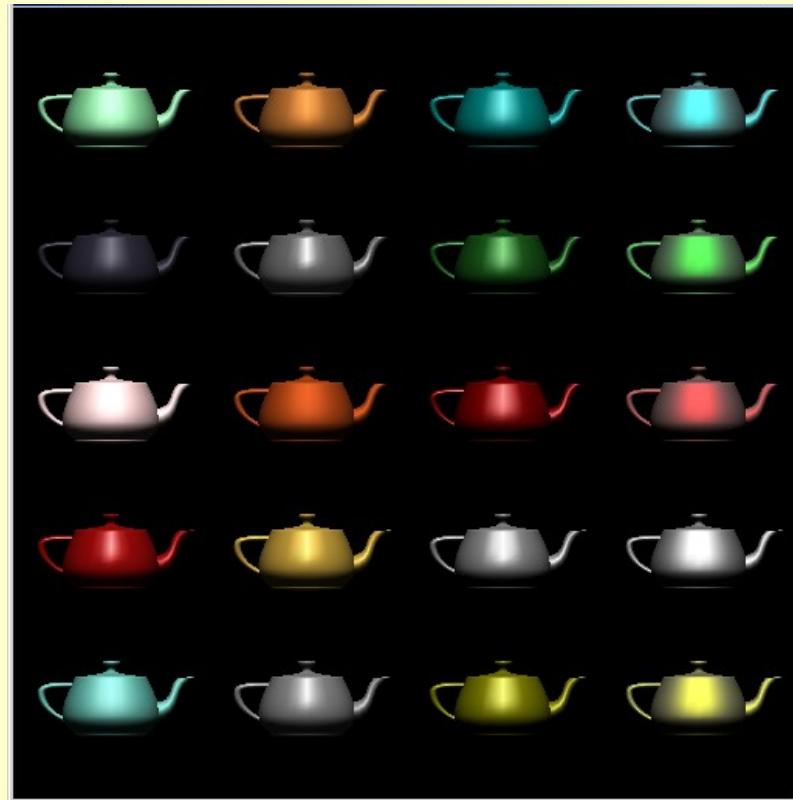
Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^\alpha$ by $(\mathbf{n} \cdot \mathbf{h})^\beta$
- β is chosen to match shininess
- Note that halfway angle is half of angle between \mathbf{r} and \mathbf{v} if vectors are coplanar
- Resulting model is known as the modified Phong or Blinn lighting model
- Specified in OpenGL standard



Example

Only differences in
these teapots are
the parameters
in the modified
Phong model





Computation of Vectors

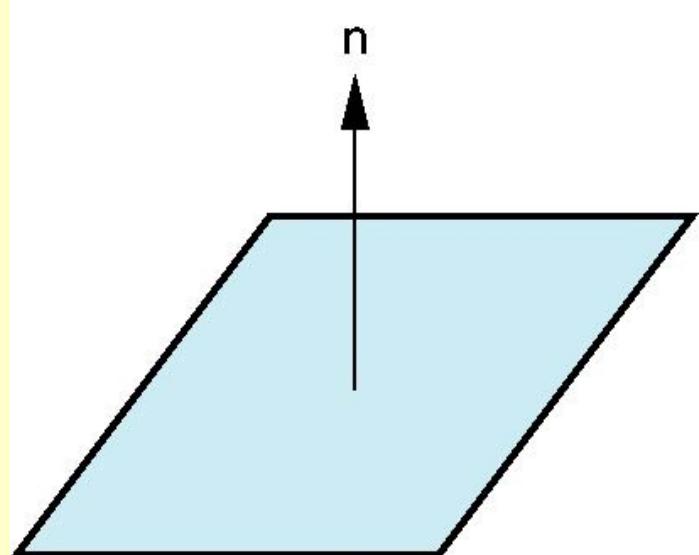
- I and v are specified by the application
- Can computer r from I and n
- Problem is determining n
- For simple surfaces n can be determined but how we determine n differs depending on underlying representation of surface
- OpenGL leaves determination of normal to application
 - ~~Exception for GLU quadrics and Bezier surfaces (Chapter 11)~~



Plane Normals

- Equation of plane: $ax+by+cz+d = 0$
- From Chapter 4 we know that plane is determined by three points p_0, p_1, p_2 or normal \mathbf{n} and p_0
- Normal can be obtained by

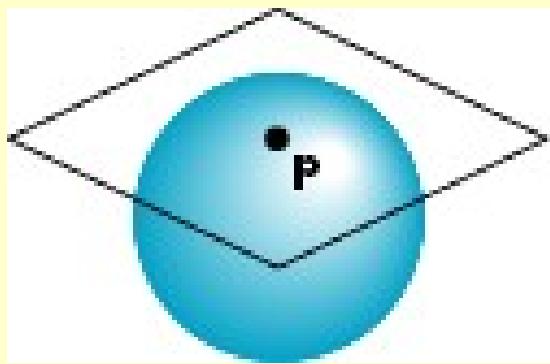
$$\mathbf{n} = (p_1 - p_0) \times (p_2 - p_0)$$





Normal to Sphere

- Implicit function $f(x,y,z)=0$
- Normal given by gradient
- Sphere $f(\mathbf{p})=\mathbf{p} \cdot \mathbf{p} - 1$
- $\mathbf{n} = [\partial f / \partial x, \partial f / \partial y, \partial f / \partial z]^T = \mathbf{p}$





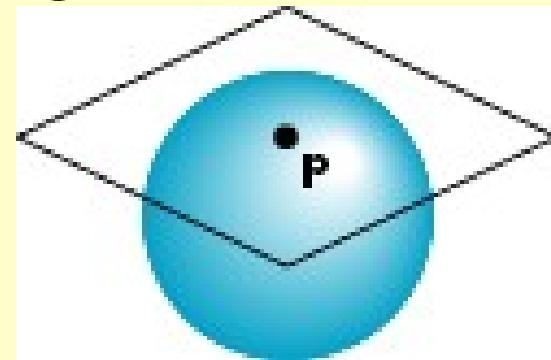
Parametric Form

- For sphere

$$x = x(u, v) = \cos u \sin v$$

$$y = y(u, v) = \cos u \cos v$$

$$z = z(u, v) = \sin u$$



- Tangent plane determined by vectors

$$\frac{\partial \mathbf{p}}{\partial u} = [\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}]^T$$

$$\frac{\partial \mathbf{p}}{\partial v} = [\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}]^T$$

- Normal given by cross product

$$\mathbf{n} = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}$$



General Case

- We can compute parametric normals for other simple cases
 - ~~Quadratics~~
 - ~~Parametric polynomial surfaces~~
- ~~Bezier surface patches (Chapter 11)~~



Steps in OpenGL shading

1. Enable shading and select model
2. Specify normals
3. Specify material properties
4. Specify lights



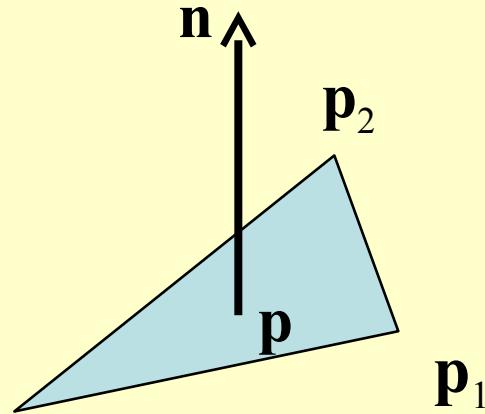
Normals

- In OpenGL the normal vector is part of the state
- Set by `glNormal*` ()
 - ~~`glNormal3f(x, y, z);`~~
 - ~~`glNormal3fv(p);`~~
- Usually we want to set the normal to have unit length so cosine calculations are correct
 - Length can be affected by transformations
 - Note that scaling does not preserve length
 - ~~`glEnable(GL_NORMALIZE)`~~ allows for autonormalization at a performance penalty



Normal for Triangle

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



$$\text{normalize } \mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$$

Note that right-hand rule determines outward face



Enabling Shading

- Shading calculations are enabled by
 - `glEnable(GL_LIGHTING)`
 - Once lighting is enabled, `glColor()` ignored
- Must enable each light source individually
 - `glEnable(GL_LIGHTi), i=0,1.....`
- Can choose light model parameters
 - `glLightModeli(parameter, GL_TRUE)`
 - `GL_LIGHT_MODEL_LOCAL_VIEWER` do not use simplifying distant viewer assumption in calculation
 - `GL_LIGHT_MODEL_TWO_SIDED` shades both sides of polygons independently



Defining a Point Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

```
GLfloat diffuse0[] = {1.0, 0.0, 0.0, 1.0};  
GLfloat ambient0[] = {1.0, 0.0, 0.0, 1.0};  
GLfloat specular0[] = {1.0, 0.0, 0.0, 1.0};  
GLfloat light0_pos[] = {1.0, 2.0, 3.0, 1.0};  
  
 glEnable(GL_LIGHTING);  
 glEnable(GL_LIGHT0);  
 glLightv(GL_LIGHT0, GL_POSITION, light0_pos);  
 glLightv(GL_LIGHT0, GL_AMBIENT, ambient0);  
 glLightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);  
 glLightv(GL_LIGHT0, GL_SPECULAR, specular0);
```



Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
 - If $w = 1.0$, we are specifying a finite location
 - If $w = 0.0$, we are specifying a parallel source with the given direction vector
- The coefficients in the distance terms are by default $a=1.0$ (constant terms), $b=c=0.0$ (linear and quadratic terms).
Change by

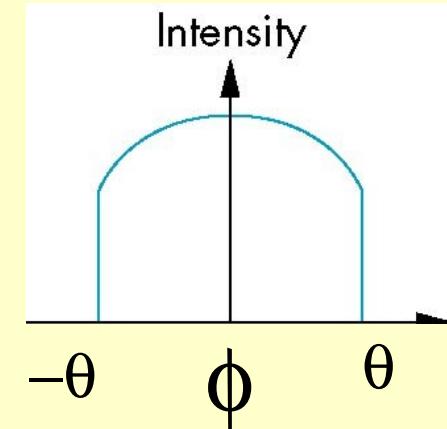
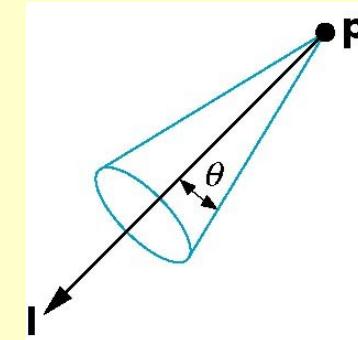
~~a = 0.80;~~

~~glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);~~



Spotlights

- Use `glLightv` to set
 - Direction `GL_SPOT_DIRECTION`
 - Cutoff `GL_SPOT_CUTOFF`
 - Attenuation `GL_SPOT_EXPONENT`
- Proportional to $\cos^{\alpha} \theta$





Global Ambient Light

- Ambient light depends on color of light sources
 - A red light in a white room will cause a red ambient term that disappears when the light is turned off
- OpenGL also allows a global ambient term that is often helpful for testing
 - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`



Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, we can
 - Move the light source(s) with the object(s)
 - Fix the object(s) and move the light source(s)
 - Fix the light source(s) and move the object(s)
 - Move the light source(s) and object(s) independently



Material Properties

- Material properties are also part of the OpenGL state and match the terms in the modified Phong model
- Set by `glMaterialfv()`

```
GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};  
GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};  
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat shine = 100.0  
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);  
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);  
glMaterialfv(GL_FRONT, GL_SHININESS, shine);
```

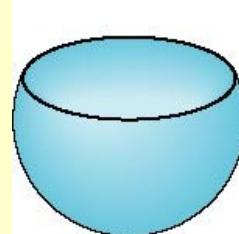
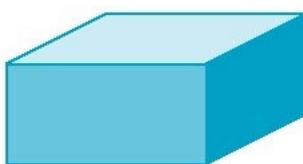


Front and Back Faces

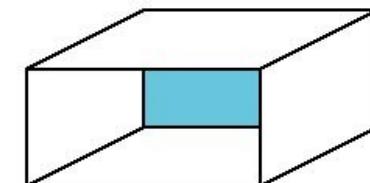
- The default is shade only front faces which works correctly for convex objects
- If we set two sided lighting, OpenGL will shade both sides of a surface
- Each side can have its own properties which are set by using `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK` in `glMaterialf`



back faces not visible



back faces visible





Emissive Term

- We can simulate a light source in OpenGL by giving a material an emissive component
- This component is unaffected by any sources or transformations

```
GLfloat emission[] = { 0.0, 0.3, 0.3, 1.0 };  
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```



Transparency

- Material properties are specified as RGBA values
- The A value can be used to make the surface translucent
- The default is that all surfaces are opaque regardless of A
- Later we will enable blending and use this feature



Efficiency

- Because material properties are part of the state, if we change materials for many surfaces, we can affect performance
- We can make the code cleaner by defining a material structure and setting all materials during initialization

```
typedef struct materialStruct {  
    GLfloat ambient[4];  
    GLfloat diffuse[4];  
    GLfloat specular[4];  
    GLfloat shininess;  
} MaterialStruct;
```

- We can then select a material by a pointer



Polygonal Shading

- Shading calculations are done for each vertex
 - ~~Vertex colors become vertex shades~~
- By default, vertex shades are interpolated across the polygon
 - ~~glShadeModel(GL_SMOOTH);~~
- If we use `glShadeModel(GL_FLAT)` ; the color at the first vertex will determine the shade of the whole polygon



Polygon Normals

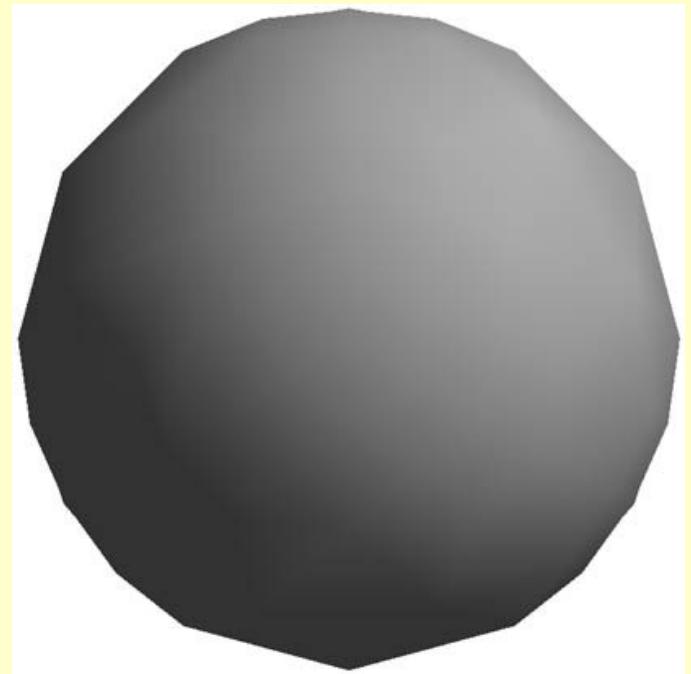
- Polygons have a single normal
 - Shaded at the vertices as computed by the Phong model can be almost same
 - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Want different normals at each vertex even though this concept is not quite correct mathematically





Smooth Shading

- We can set a new normal at each vertex
- Easy for sphere model
 - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*

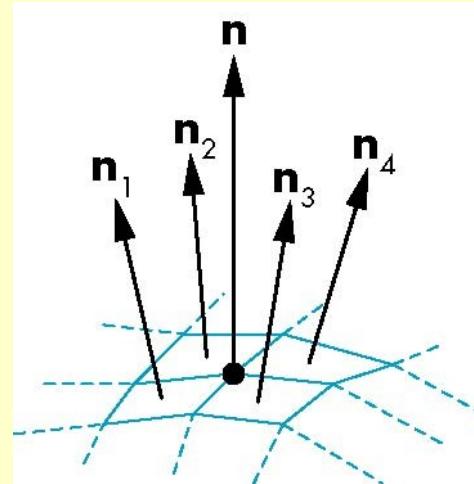




Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4) / |\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|$$





Gouraud and Phong Shading

- Gouraud Shading
 - Find average normal at each vertex (vertex normals)
 - Apply modified Phong model at each vertex
 - Interpolate vertex shades across each polygon
- Phong shading
 - Find vertex normals
 - Interpolate vertex normals across edges
 - Interpolate edge normals across polygon
 - Apply modified Phong model at each fragment



Comparison

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
 - Until recently not available in real time systems
 - Now can be done using fragment shaders (see Chapter 9)
- Both need data structures to represent meshes so we can obtain vertex normals



Computer Graphics

Building Models

John Shearer

Culture Lab – space 2

john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



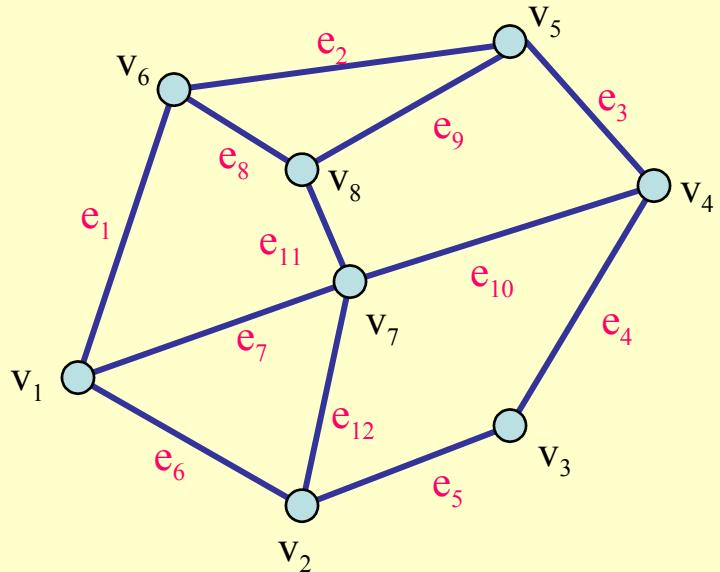
Objectives

- Introduce simple data structures for building polygonal models
 - Vertex lists
 - Edge lists
- OpenGL vertex arrays



Representing a Mesh

- Consider a mesh
- There are 8 nodes and 12 edges
 - 5 interior polygons
 - 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i \ y_i \ z_i)$





Simple Representation

- Define each polygon by the geometric locations of its vertices
- Leads to OpenGL code such as

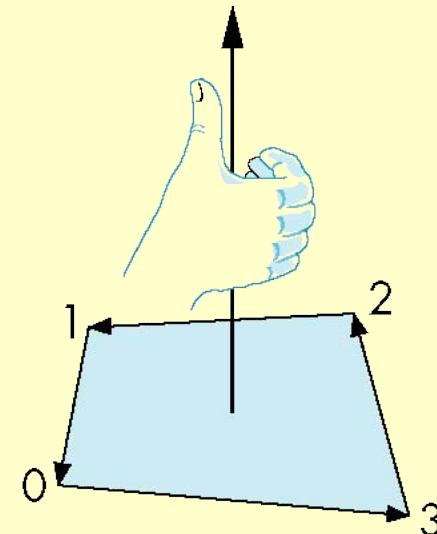
```
glBegin(GL_POLYGON);  
    glVertex3f(x1, y1, z1);  
    glVertex3f(x6, y6, z6);  
    glVertex3f(x8, y8, z8);  
    glVertex3f(x7, y7, z7);  
glEnd();
```

- Inefficient and unstructured
 - Consider moving a vertex to a new location
 - Must search for all occurrences



Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outward-facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently





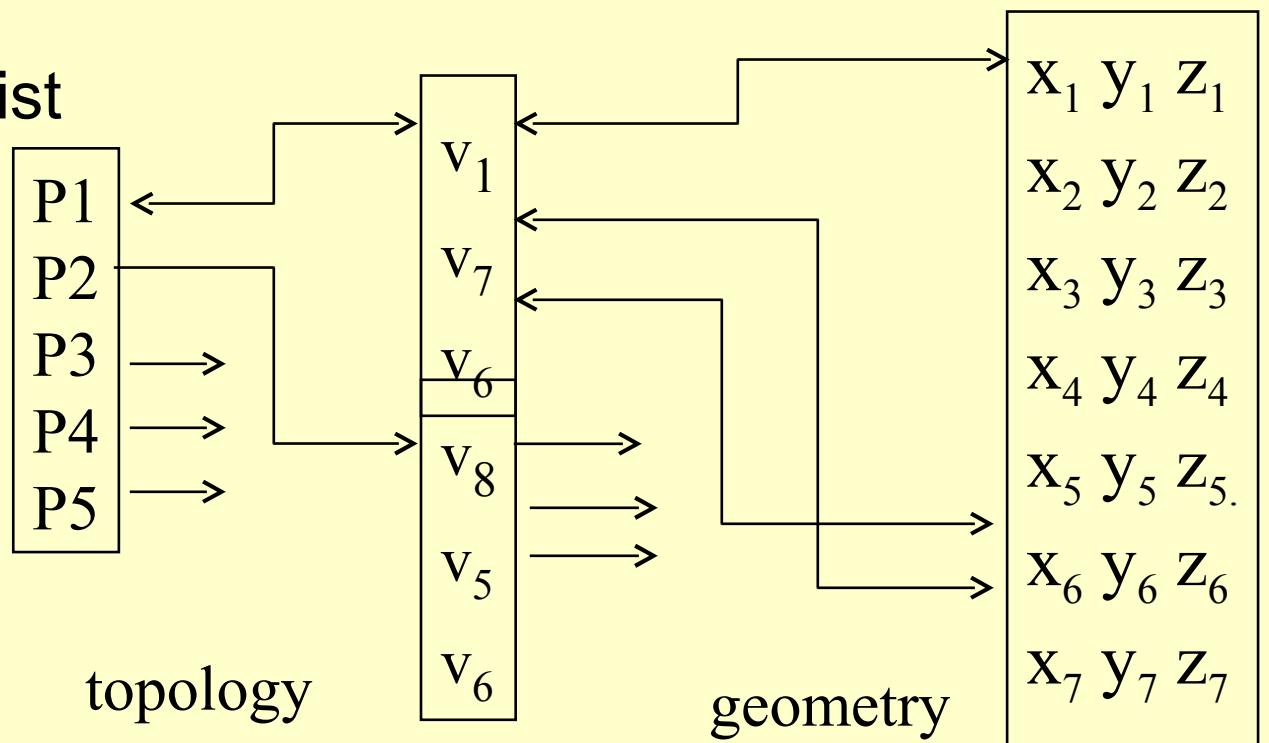
Geometry vs Topology

- Generally it is a good idea to look for data structures that separate the geometry from the topology
 - Geometry: locations of the vertices
 - Topology: organization of the vertices and edges
 - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
 - Topology holds even if geometry changes



Vertex Lists

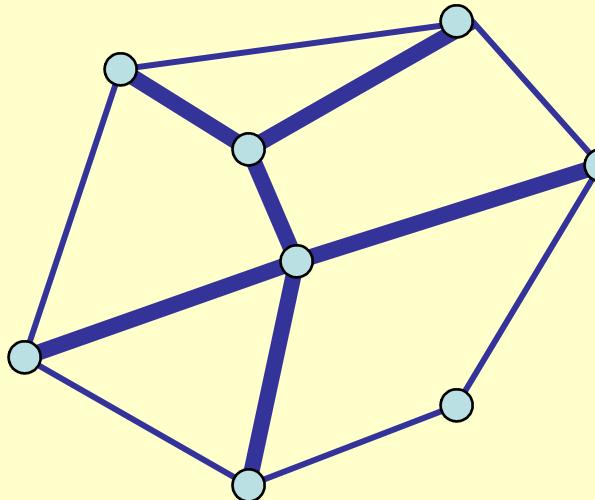
- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list





Shared Edges

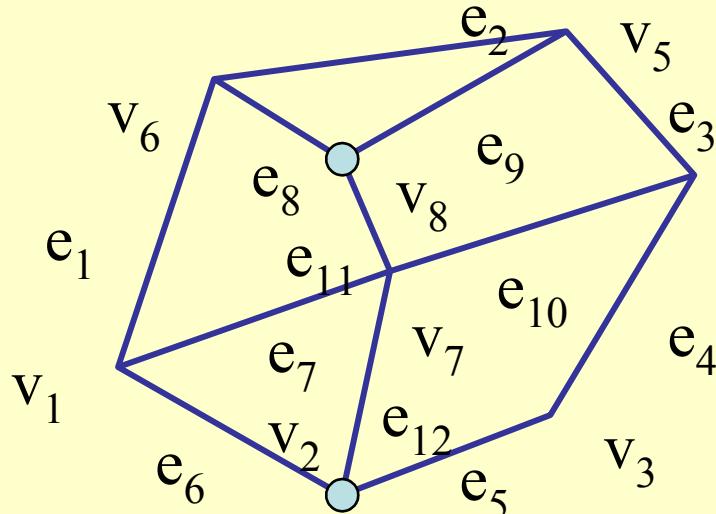
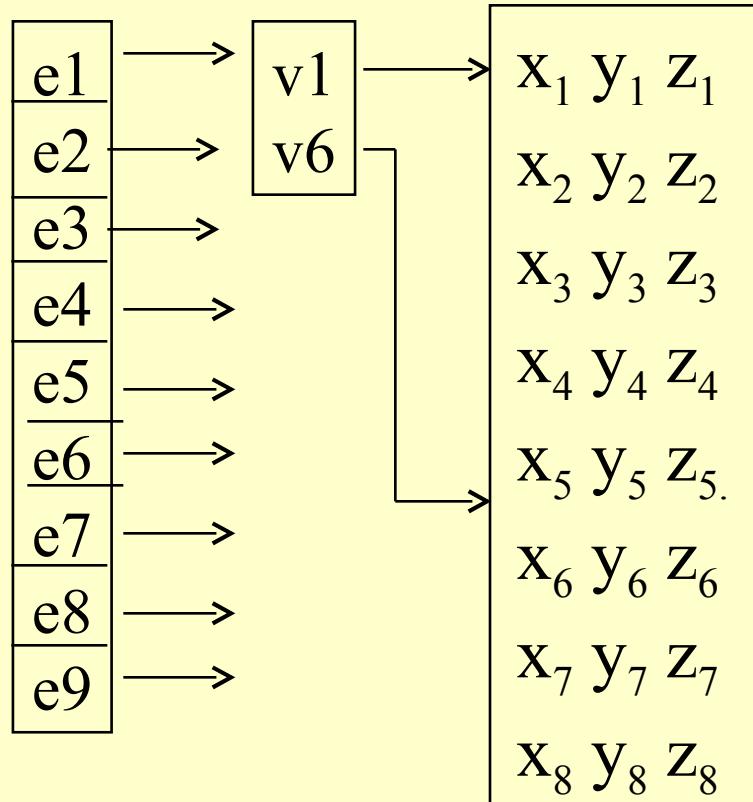
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*



Edge List



Note polygons are
not represented



Modeling a Cube

- Model a color cube for rotating cube program
- Define global arrays for vertices and colors

```
GLfloat vertices[][][3] = {{{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},  
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},  
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}}};
```

```
GLfloat colors[][][3] = {{ {0.0,0.0,0.0},{1.0,0.0,0.0},  
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},  
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}}};
```



Drawing a polygon from a list of indices

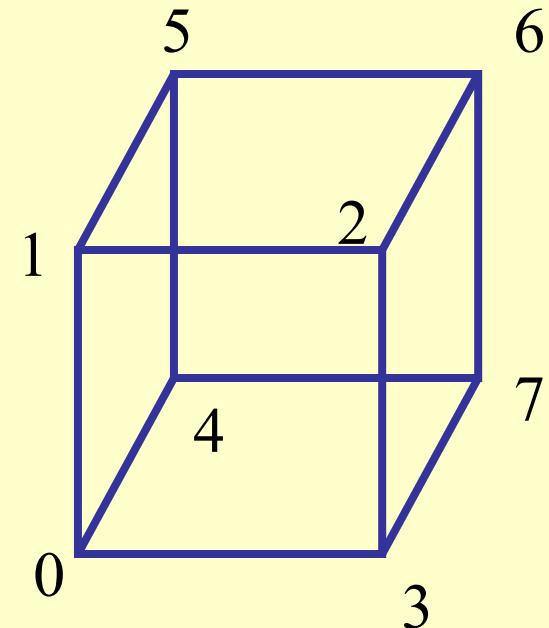
Draw a quadrilateral from a list of indices into the array `vertices` and use color corresponding to first index

```
void polygon(int a, int b, int c , int d)
{
    glBegin(GL_POLYGON);
        glColor3fv(colors[a]);
        glVertex3fv(vertices[a]);
        glColor3fv(colors[b]);
        glVertex3fv(vertices[b]);
        glColor3fv(colors[c]);
        glVertex3fv(vertices[c]);
        glColor3fv(colors[d]);
        glVertex3fv(vertices[d]);
    glEnd();
}
```



Draw cube from faces

```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



- Note that vertices are ordered so that we obtain correct outward facing normals



Efficiency

- The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube
- Drawing a cube by its faces in the most straight forward way requires
 - 6 glBegin, 6 glEnd
 - 6 glColor
 - 24 glVertex
 - More if we use texture and lighting



Vertex Arrays

- OpenGL provides a facility called *vertex arrays* that allows us to store array data in the implementation
- Six types of arrays supported
 - Vertices
 - Colors
 - Color indices
 - Normals
 - Texture coordinates
 - Edge flags
- We will need only colors and vertices



Initialization

- Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);  
glEnableClientState(GL_VERTEX_ARRAY);
```

- Identify location of arrays

```
glVertexPointer(3, GL_FLOAT, 0, vertices);  
           ↑  
           3d arrays       stored as floats      data contiguous  
                                                 ↑  
                                                 data array
```

```
glColorPointer(3, GL_FLOAT, 0, colors);
```



Mapping indices to faces

- Form an array of face indices

```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6  
 0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

- Each successive four indices describe a face of the cube
- Draw through **glDrawElements** which replaces all **glVertex** and **glColor** calls in the display callback



Drawing the cube

- Method 1:

```
for(i=0; i<6; i++) glDrawElements(GL_POLYGON, 4,  
GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

- Method 2:

```
glDrawElements(GL_QUADS, 24,  
GL_UNSIGNED_BYTE, cubeIndices);
```

format of index data

what to draw

number of indices

what to draw

number of indices to be rendered

start of index data

Draws cube with 1
function call!!



Computer Graphics

Implementation 1

Lecture 15

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Introduce basic implementation strategies
- Clipping
- Scan conversion



Overview

- At end of the geometric pipeline, vertices have been assembled into primitives
 - Must clip out primitives that are outside the view frustum
 - Algorithms based on representing primitives by lists of vertices
 - Must find which pixels can be affected by each primitive
 - Fragment generation
 - Rasterization or scan conversion



Required Tasks

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





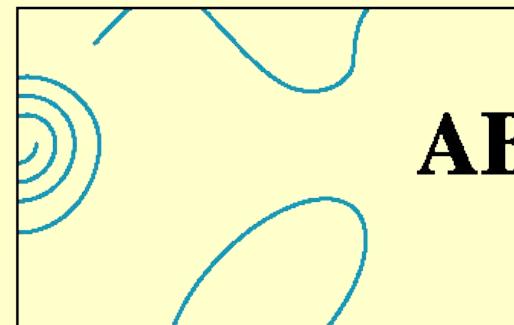
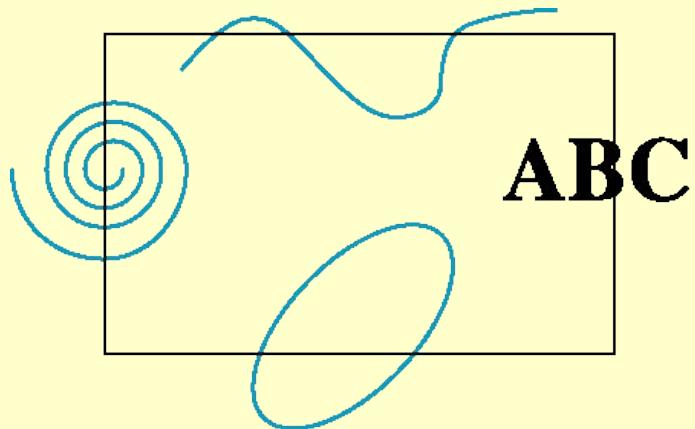
Rasterization Meta Algorithms

- 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~~



Clipping

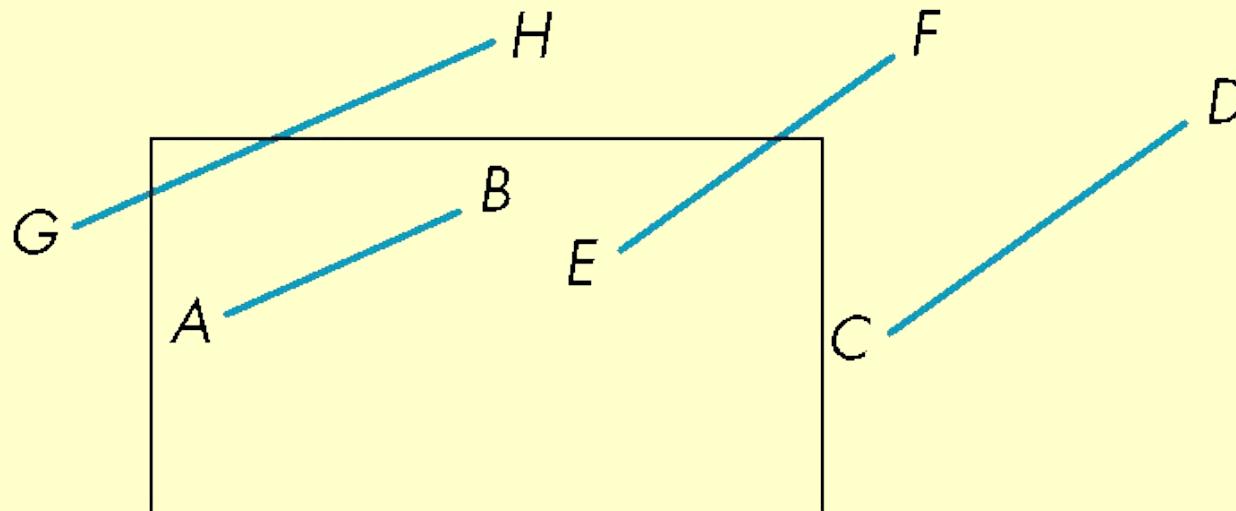
- 2D against clipping window
- 3D against clipping volume
- Easy for line segments polygons
- Hard for curves and text
- Convert to lines and polygons first





Clipping 2D Line Segments

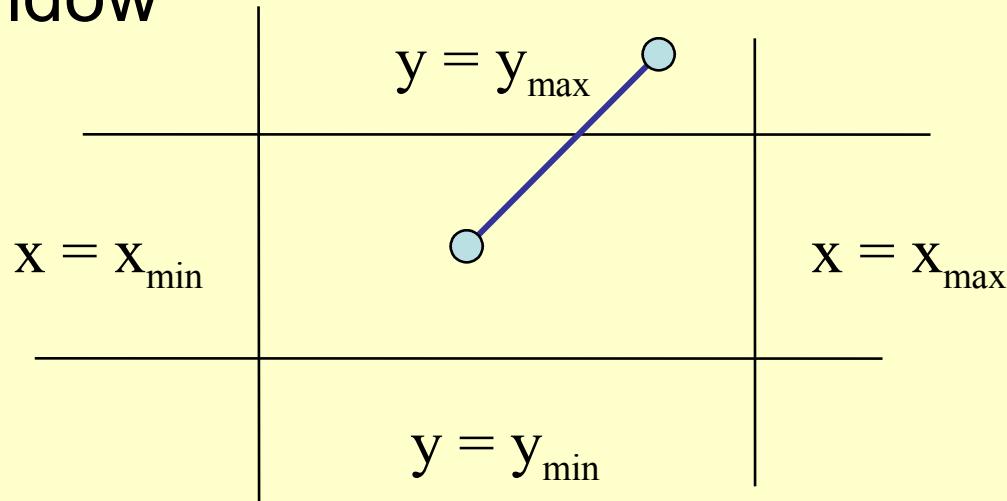
- Brute force approach: compute intersections with all sides of clipping window
 - Inefficient: one division per intersection





Cohen-Sutherland Algorithm

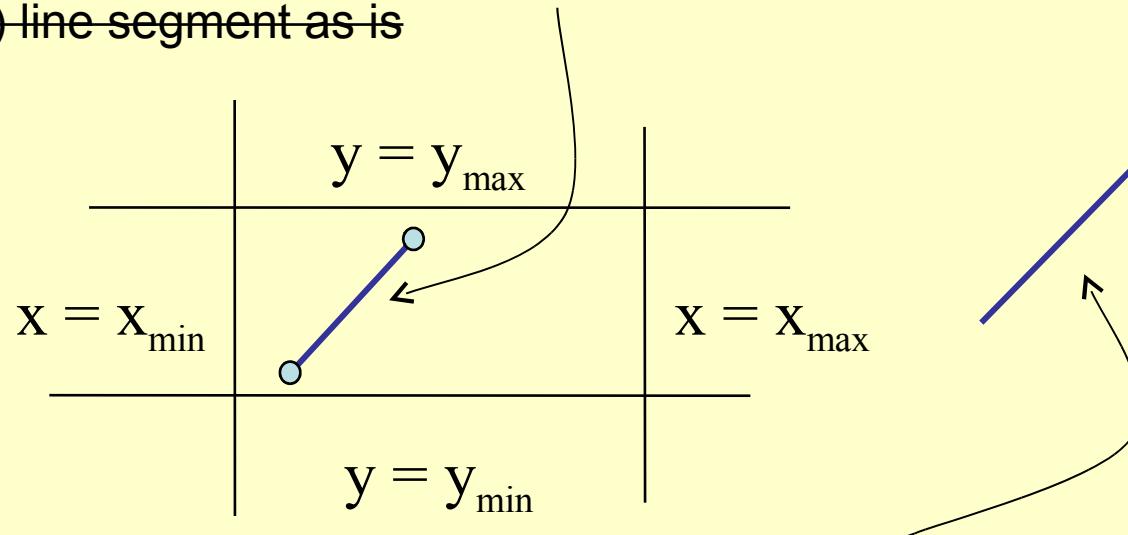
- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window





The Cases

- Case 1: both endpoints of line segment inside all four lines
 - ~~Draw (accept)~~ line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
 - ~~Discard (reject)~~ the line segment



The Cases

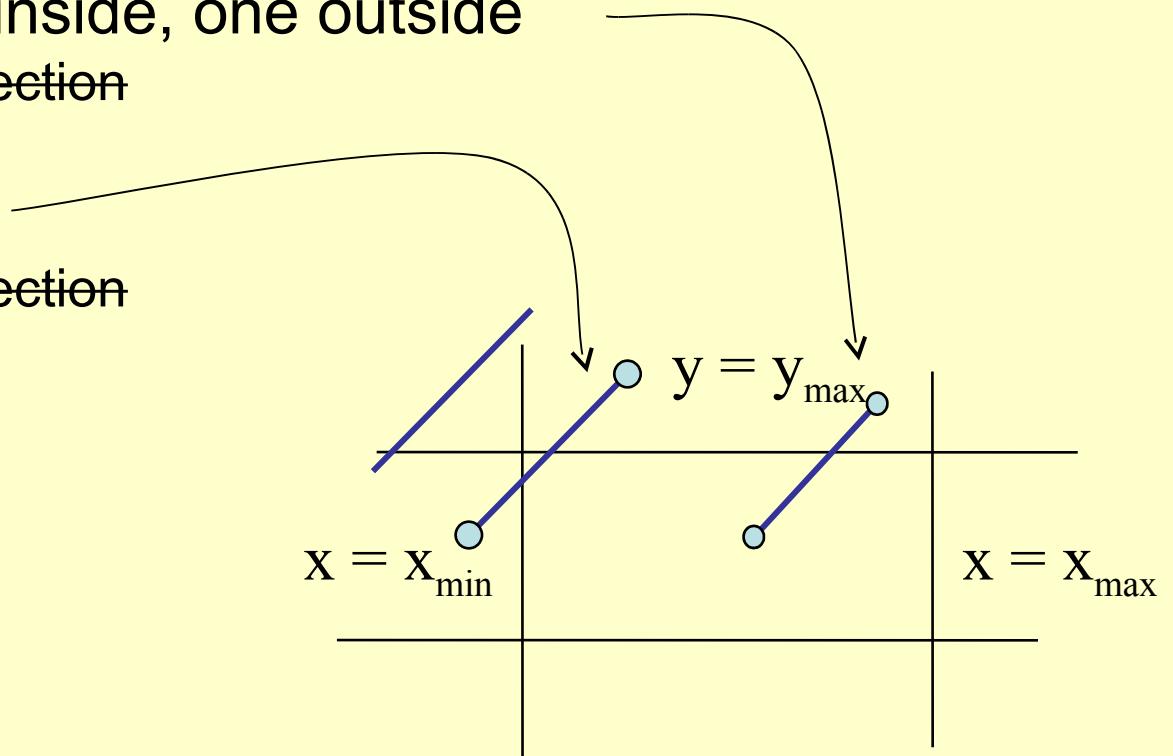
- Case 3: One endpoint inside, one outside

~~— Must do at least one intersection~~

- Case 4: Both outside

~~— May have part inside~~

~~— Must do at least one intersection~~





Defining Outcodes

- For each endpoint, define an outcode

$$b_0 b_1 b_2 b_3$$

$b_0 = 1$ if $y > y_{\max}$, 0 otherwise

$b_1 = 1$ if $y < y_{\min}$, 0 otherwise

$b_2 = 1$ if $x > x_{\max}$, 0 otherwise

$b_3 = 1$ if $x < x_{\min}$, 0 otherwise

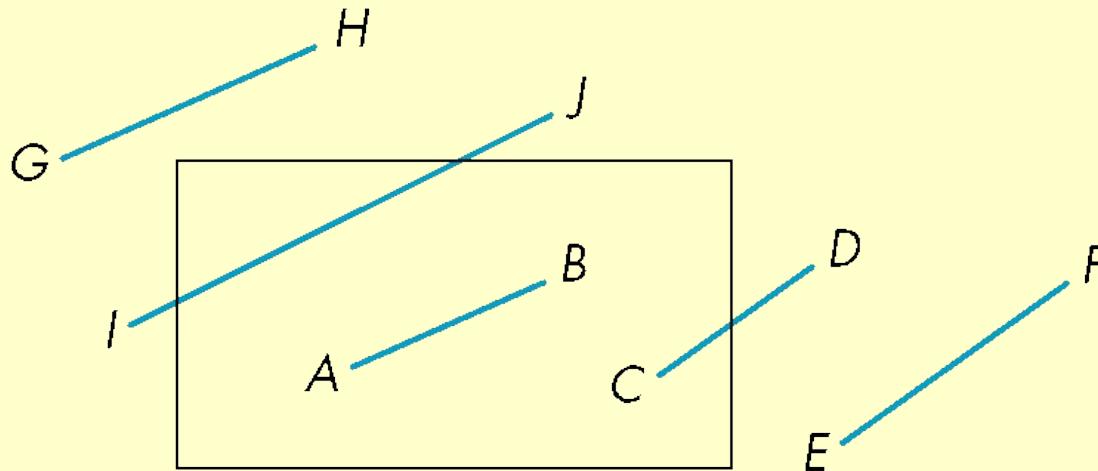
1001	1000	1010	$y = y_{\max}$
0001	0000	0010	$y = y_{\min}$
0101	0100	0110	
$x = x_{\min}$	$x = x_{\max}$		

- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions



Using Outcodes

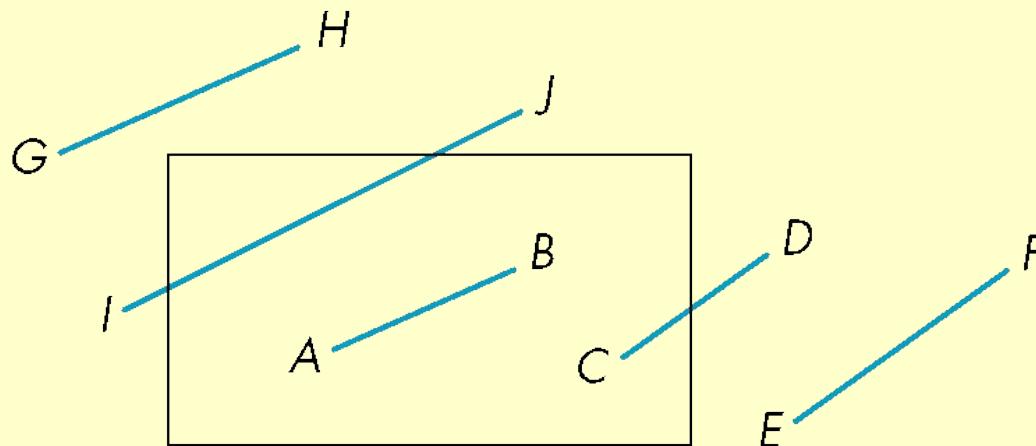
- Consider the 5 cases below
- AB: $\text{outcode}(A) = \text{outcode}(B) = 0$
 - Accept line segment





Using Outcodes

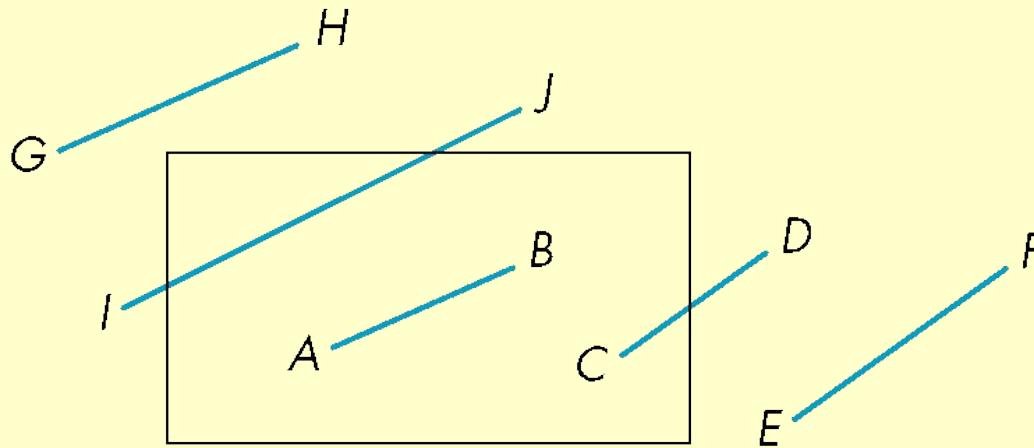
- CD: outcode(C) = 0, outcode(D) ≠ 0
 - Compute intersection
 - Location of 1 in outcode(D) determines which edge to intersect with
 - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two intersections





Using Outcodes

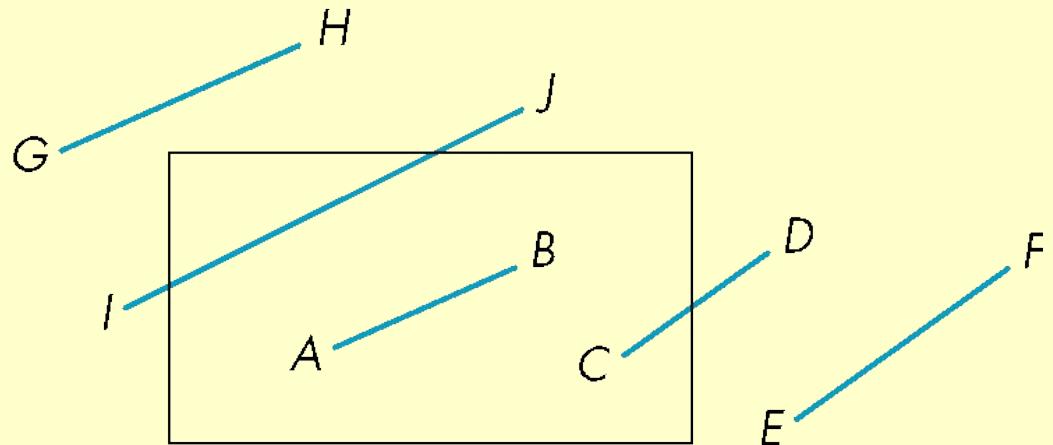
- EF: $\text{outcode}(E)$ logically ANDed with $\text{outcode}(F)$ (bitwise) $\neq 0$
 - Both outcodes have a 1 bit in the same place
 - Line segment is outside of corresponding side of clipping window
 - reject





Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm





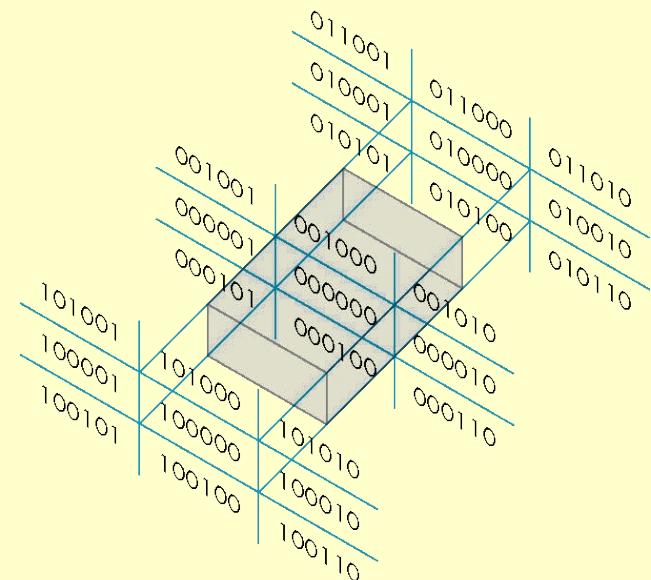
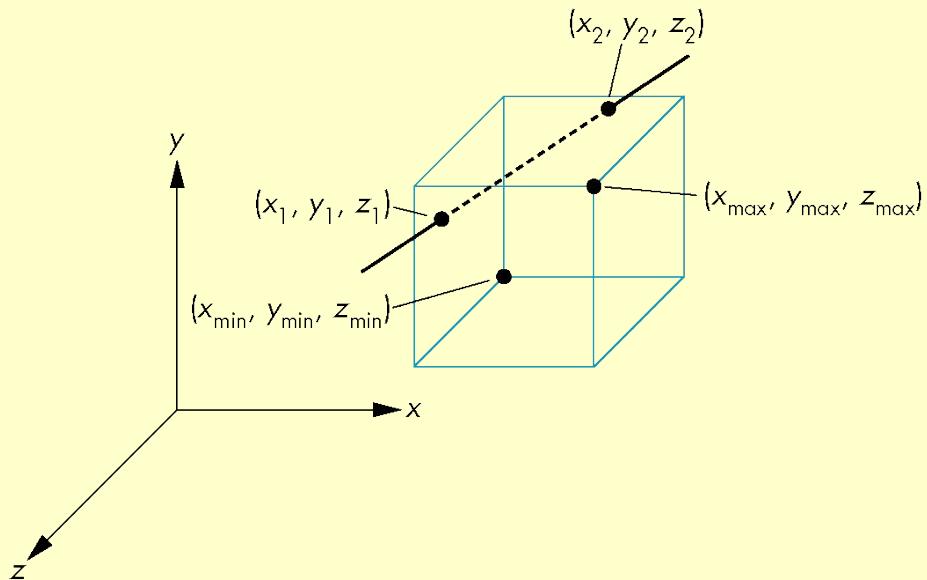
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
 - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step



Cohen Sutherland in 3D

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

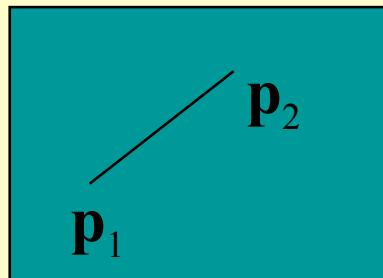




Liang-Barsky Clipping

- Consider the parametric form of a line segment

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

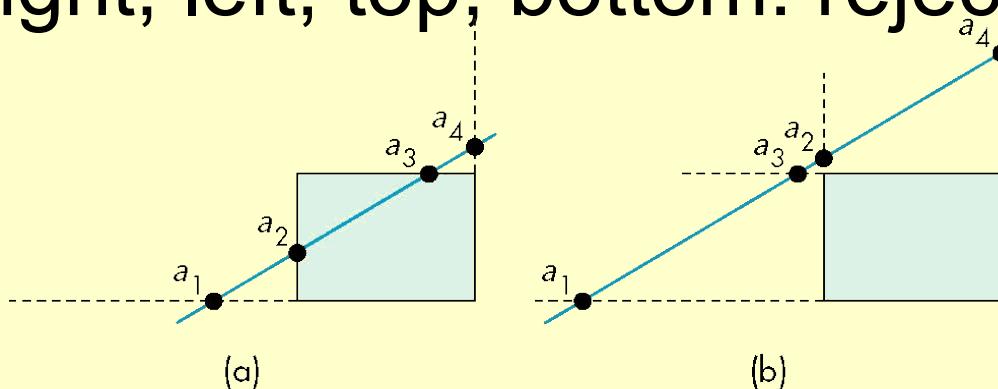


- We can distinguish between the cases by looking at the ordering of the values of α where the line determined by the line segment crosses the lines that determine the window



Liang-Barsky Clipping

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





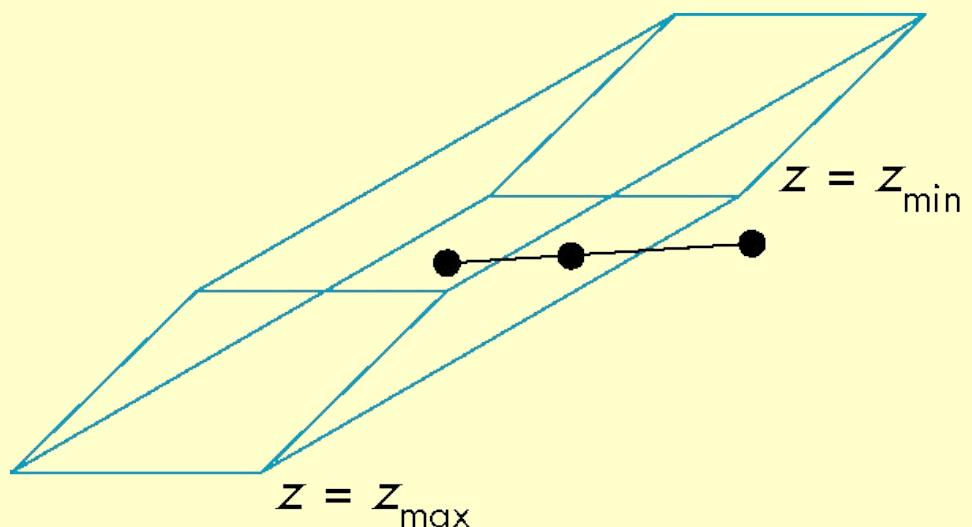
Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α , we do not have to use algorithm recursively as with C-S
- Extends to 3D



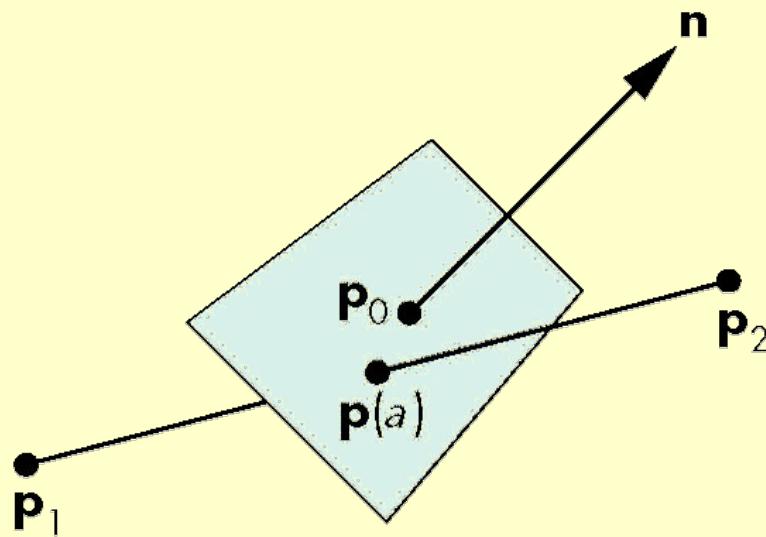
Clipping and Normalization

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



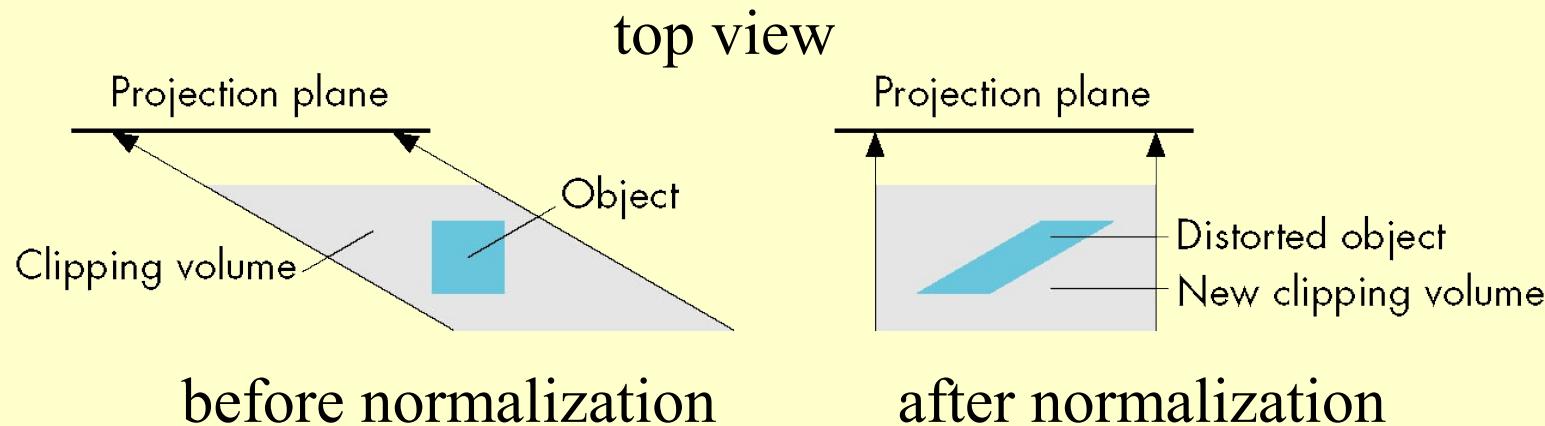


Plane-Line Intersections





Normalized Form



- Normalization is part of viewing (pre clipping)
- but after normalization, we clip against sides of
- right parallelepiped

- Typical intersection calculation now requires only
- a floating point subtraction, e.g. is $x > xmax$?



Computer Graphics

Implementation II

Lecture 16

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



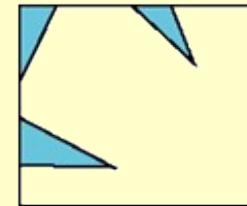
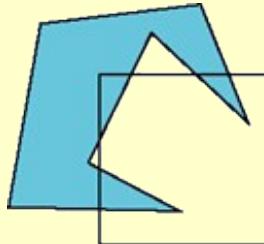
Objectives

- Introduce clipping algorithms for polygons
- Survey hidden-surface algorithms



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

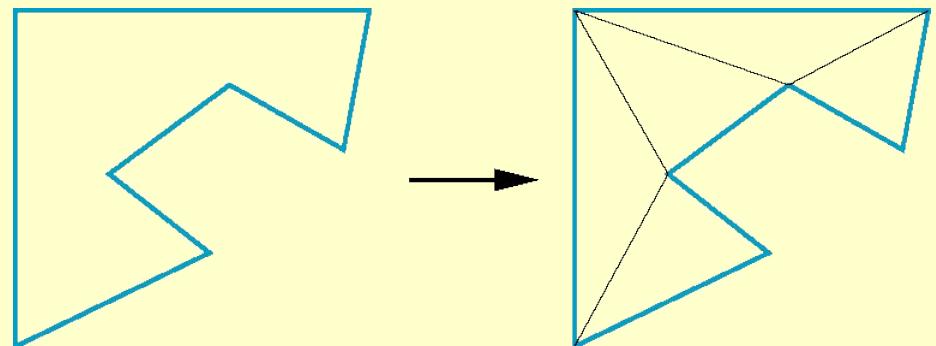


- However, clipping a convex polygon can yield at most one other polygon



Tessellation and Convexity

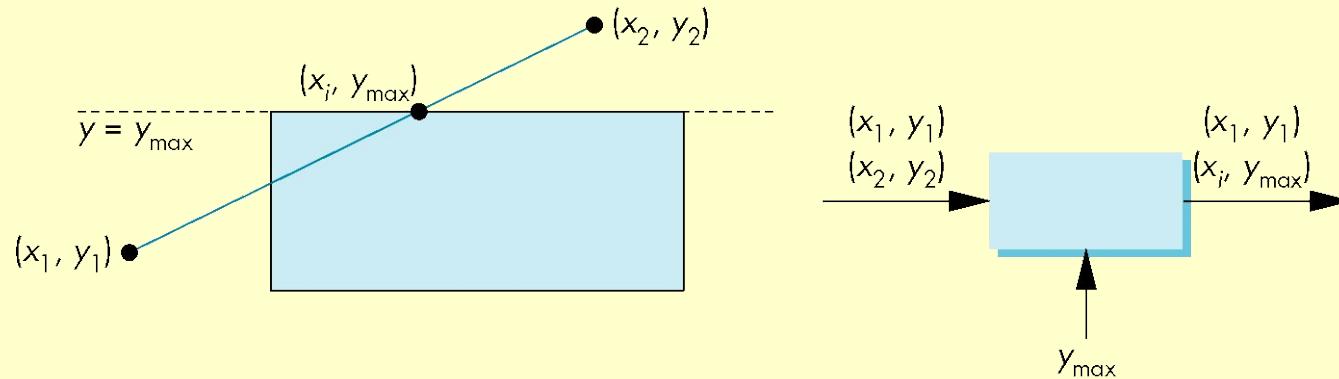
- One strategy is to replace nonconvex (concave) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation code in GLU library
 - GLU provides tessellation routines to let you render concave polygons, self-intersecting polygons, and polygons with holes. The tessellation routines break these complex primitives up into (possibly groups of) simpler, convex primitives that can be rendered by the OpenGL API





Clipping as a Black Box

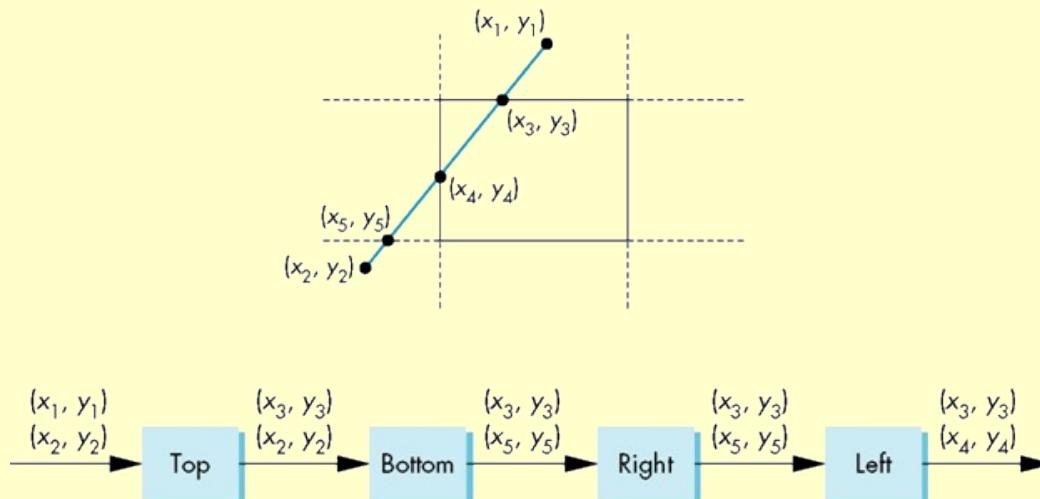
- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment





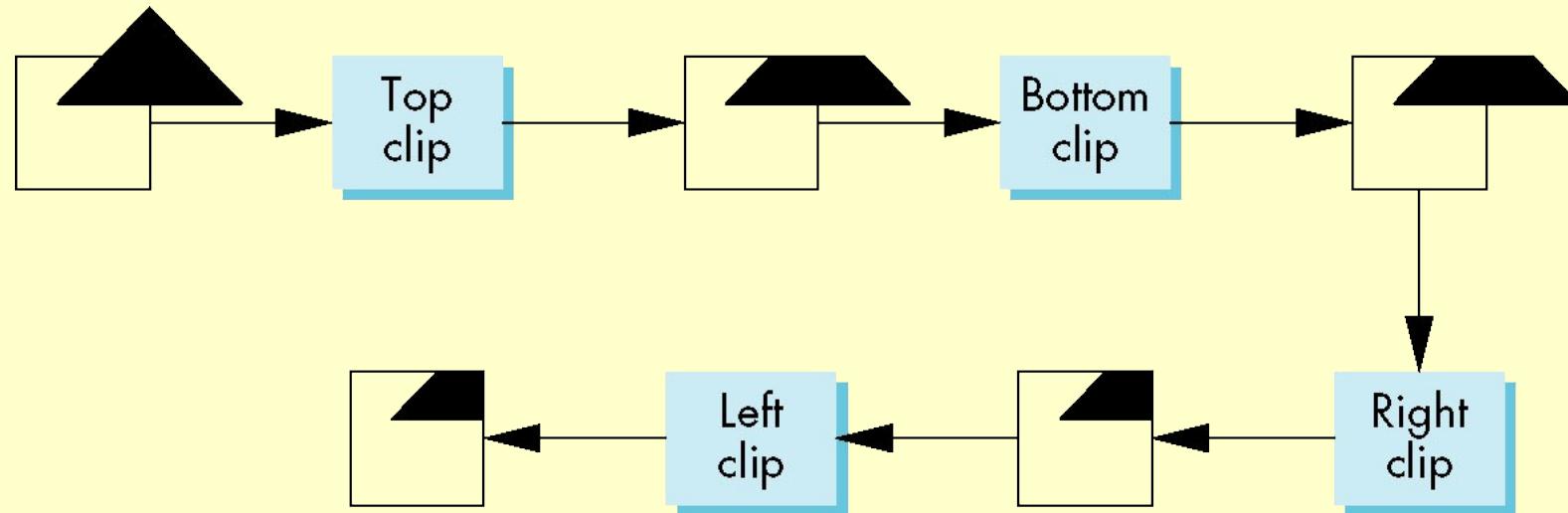
Pipeline Clipping of Line Segments

- Clipping against each side of window is independent of other sides
 - Can use four independent clippers in a pipeline





Pipeline Clipping of Polygons

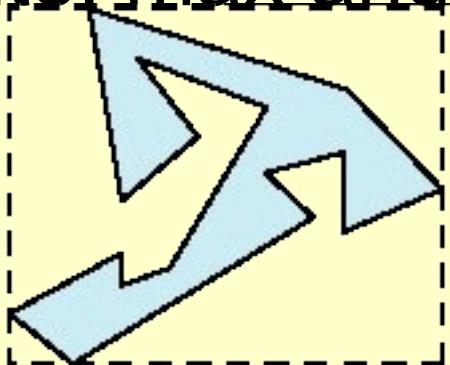


- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency



Bounding Boxes

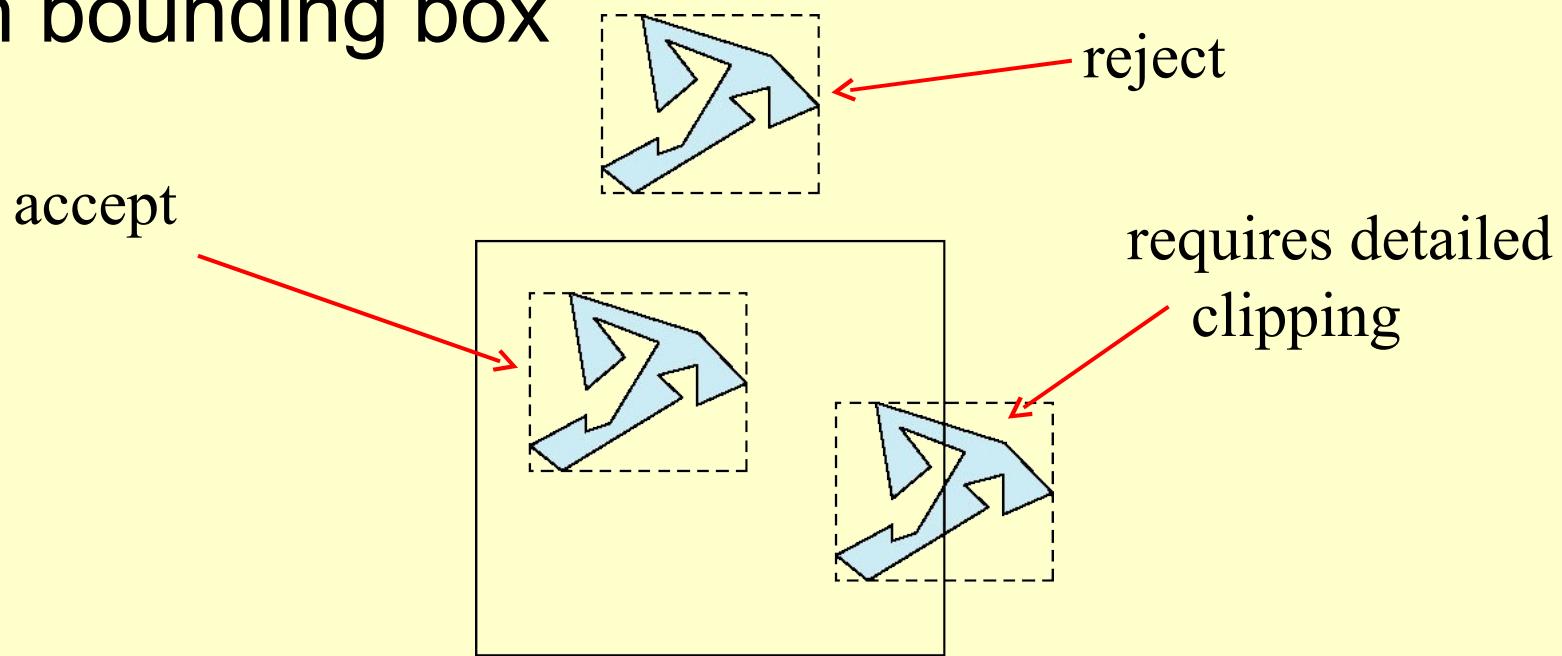
- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
 - Smallest rectangle aligned with axes that encloses the polygon
 - Simple to compute: max and min of x and y





Bounding boxes

Can usually determine accept/reject based only on bounding box





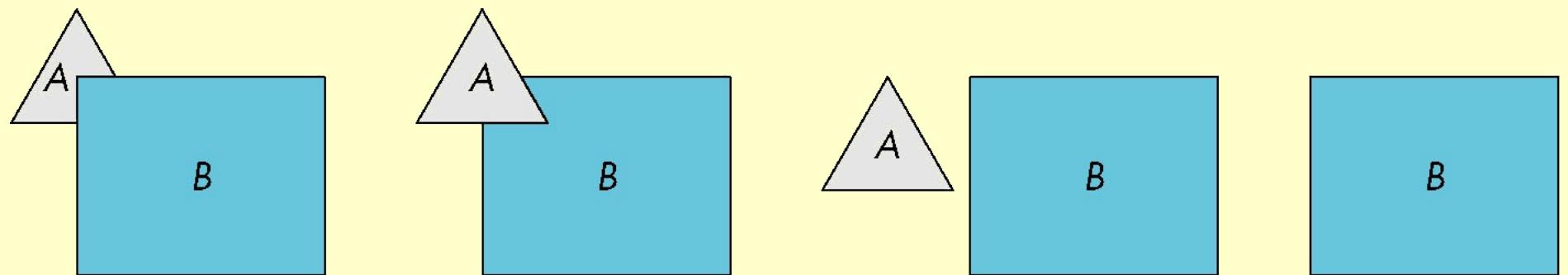
Clipping and Visibility

- Clipping has much in common with hidden-surface removal
- In both cases, we are trying 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



Hidden Surface Removal

- Object-space approach: use pairwise testing between polygons (objects)

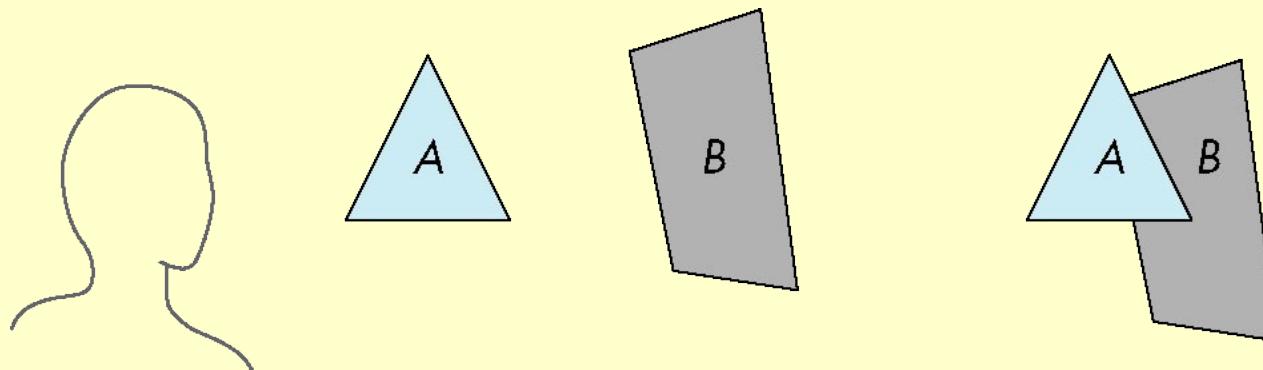


- Worst case complexity $O(n^2)$ for n polygons



Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

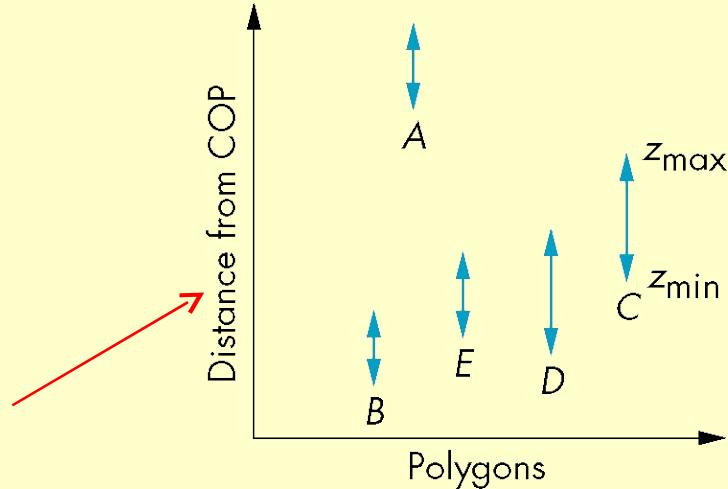
Fill B then A



Depth Sort

- Requires ordering of polygons first
 - $O(n \log n)$ calculation for ordering
 - Not every polygon is either in front or behind all other polygons
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

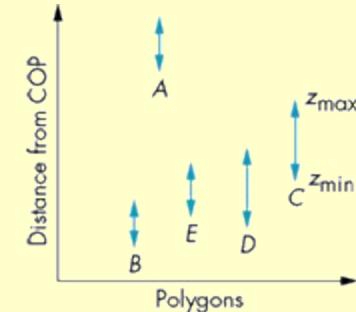




Easy Cases

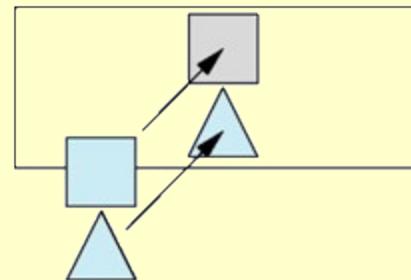
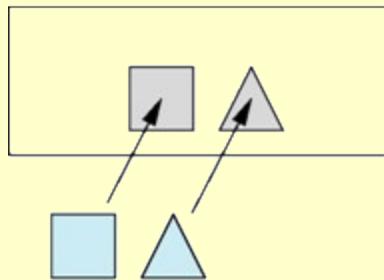
- A lies behind all other polygons

~~—Can render~~



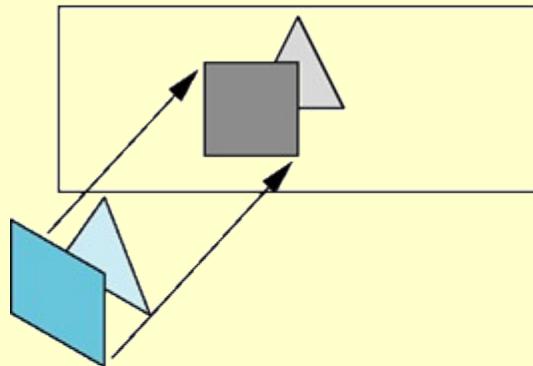
- Polygons overlap in z but not in either x or y

~~—Can render independently~~

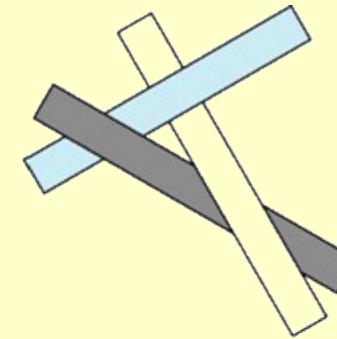




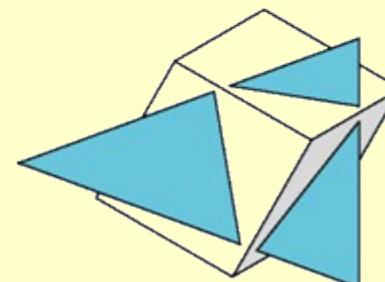
Hard Cases



Overlap in all directions
but can one is fully on
one side of the other



cyclic overlap



penetration



Back-Face Removal (Culling)

- face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos \theta \geq 0$
or $\mathbf{v} \cdot \mathbf{n} \geq 0$
- plane of face has form $ax + by +cz +d = 0$
but after normalization $\mathbf{n} = (0\ 0\ 1\ 0)^T$
- need only test the sign of c
- In OpenGL we can simply enable culling
but may not work correctly if we have nonconvex objects

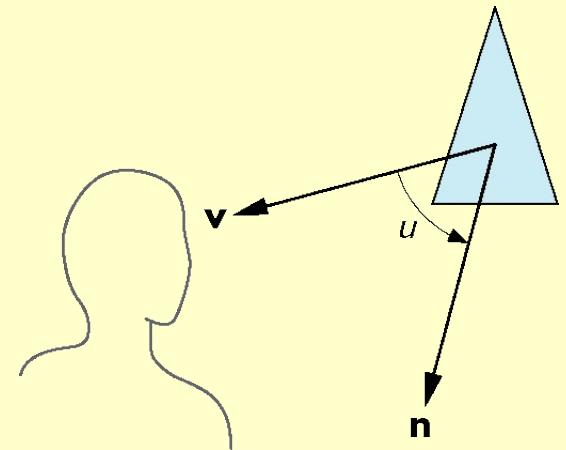
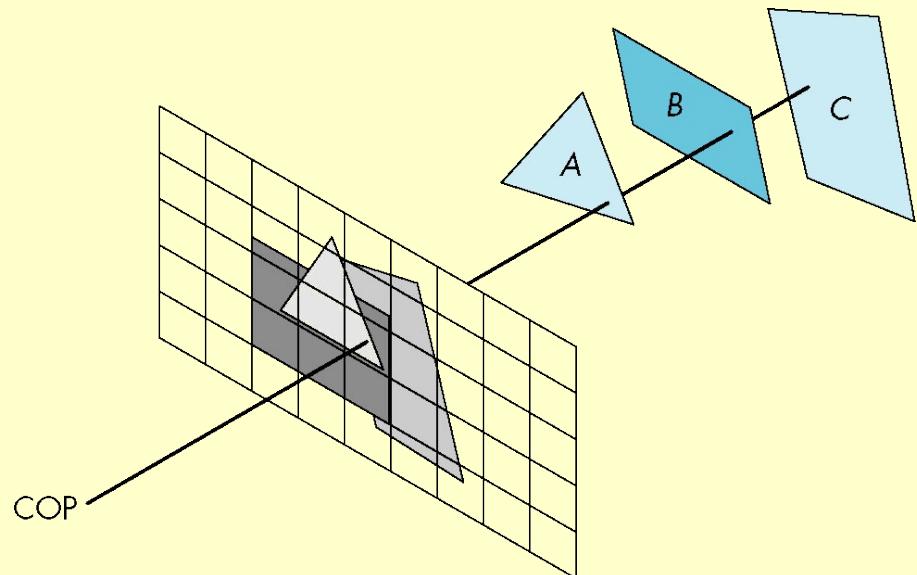




Image Space Approach

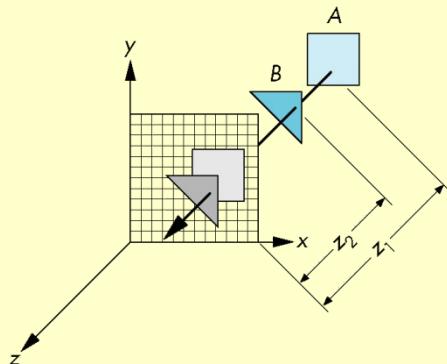
- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity $O(nmk)$
- Ray tracing
- z-buffer





z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer





Efficiency

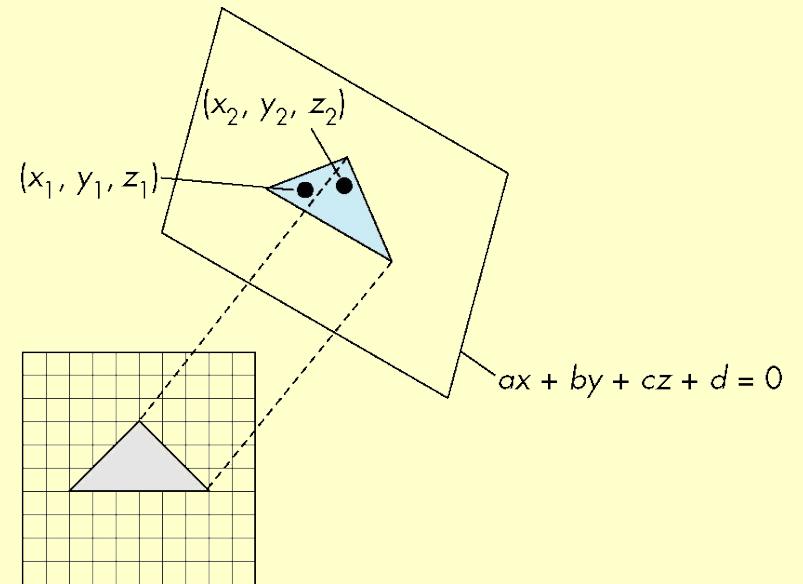
- If we work scan line by scan line as we move across a scan line, the depth changes satisfy
 $a\Delta x + b\Delta y + c\Delta z = 0$

Along scan line

$$\Delta y = 0$$

$$\Delta z = - \Delta x$$

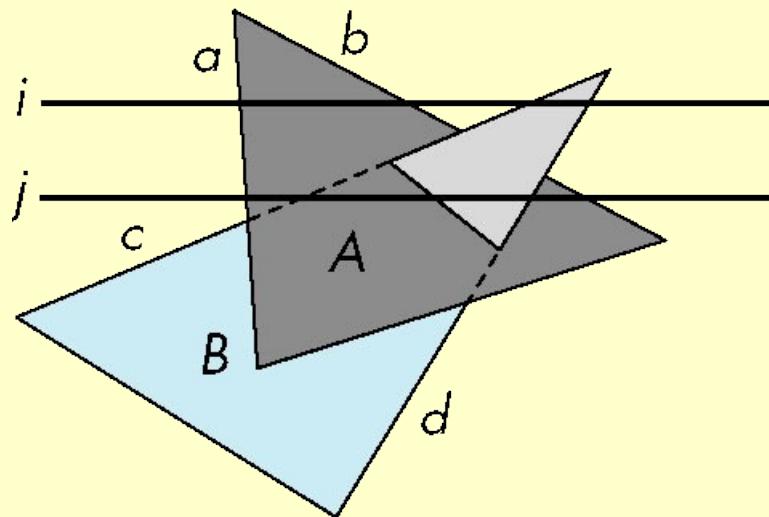
In screen space $\Delta x = 1$





Scan-Line Algorithm

- 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



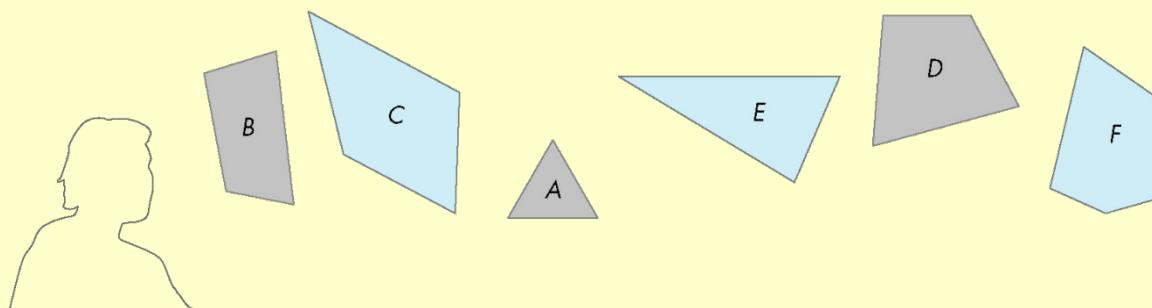
Implementation

- 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



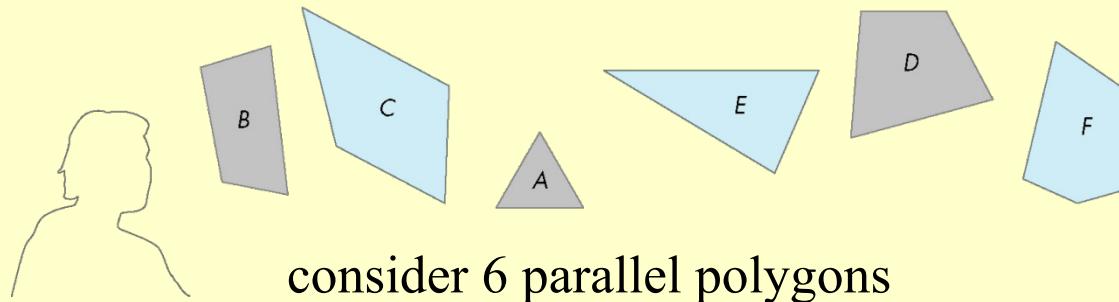
Visibility Testing

- In many realtime applications, such as games, we want to eliminate as many objects as possible within the application
 - Reduce burden on pipeline
 - Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree

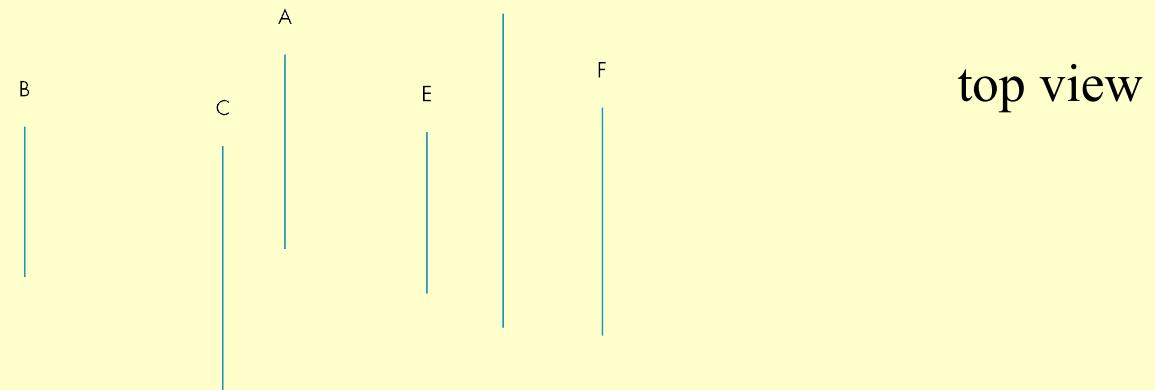




Simple Example



consider 6 parallel polygons

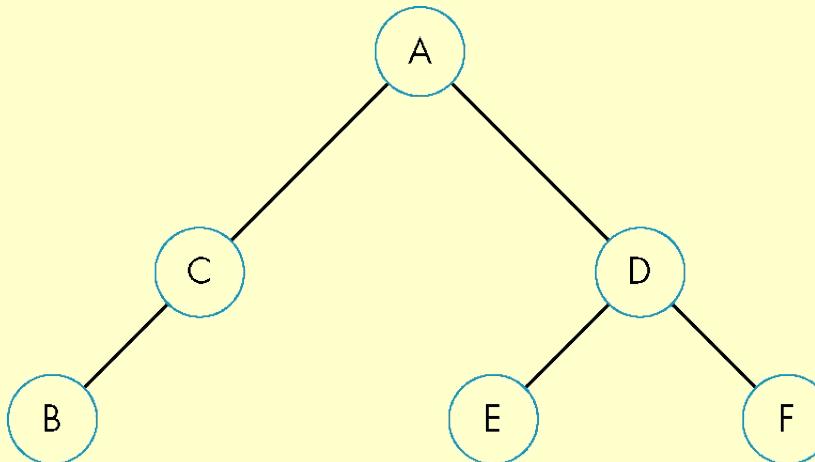


The plane of A separates B and C from D, E and F



BSP Tree

- Can continue recursively
 - Plane of C separates B from A
 - Plane of D separates E and F
- Can put this information in a BSP tree
 - Use for visibility and occlusion testing





Computer Graphics

Implementation III

Lecture 17

John Shearer
Culture Lab – space 2
john.shearer@ncl.ac.uk

<http://di.ncl.ac.uk/teaching/csc3201/>



Objectives

- Survey Line Drawing Algorithms
 - ~~DDA~~
 - ~~Bresenham~~



Rasterization

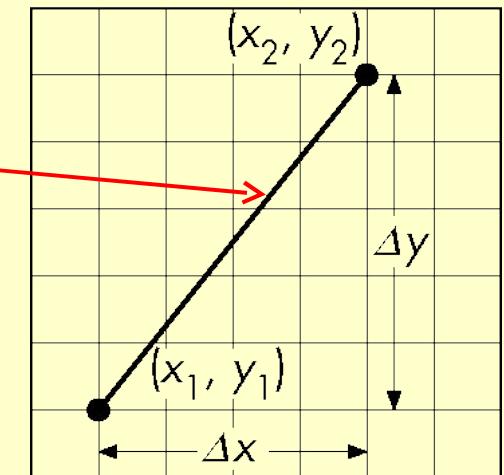
- Rasterization (scan conversion)
 - Determine which pixels that are inside primitive specified by a set of vertices
 - Produces a set of fragments
 - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties



Scan Conversion of Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **write_pixel** function

$$y = mx + h$$





DDA Algorithm

- Digital Differential Analyzer
- DDA was a mechanical device for numerical solution of differential equations
- Line $y = mx + b$ satisfies differential equation
- $\frac{dy}{dx} = m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$

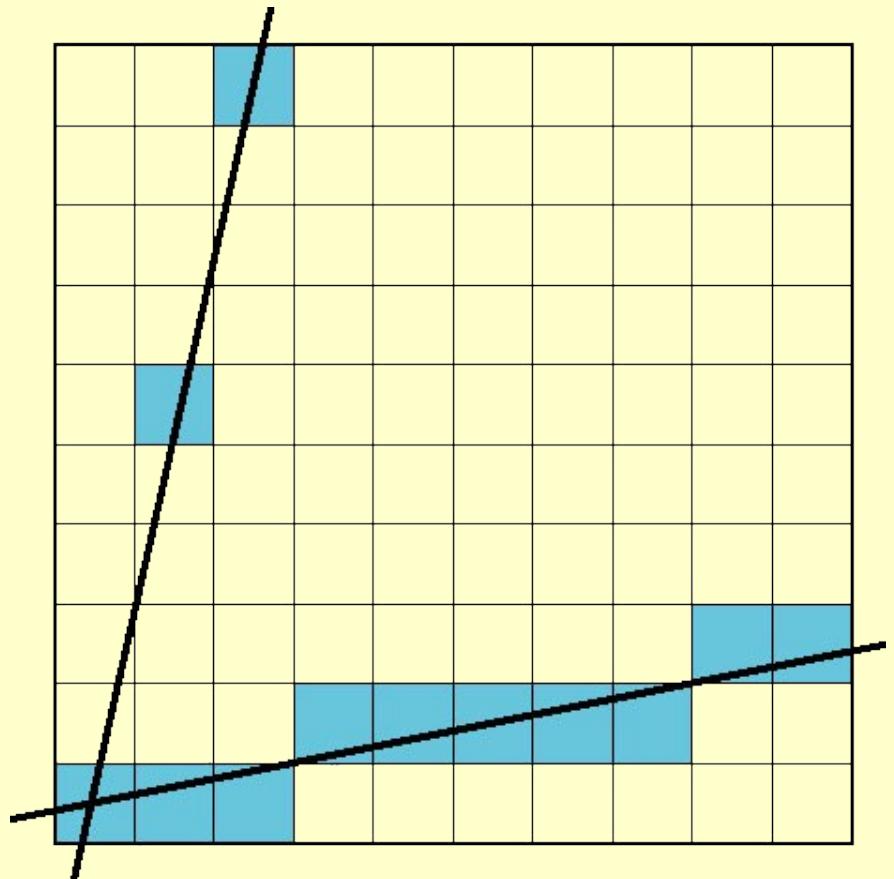
- Along scan line $\Delta x = 1$

```
for(x=x1; x<=x2, ix++) {
        y+=m;                                // Floating point addition
        write_pixel(x, round(y), line_color)
}
```



Problem

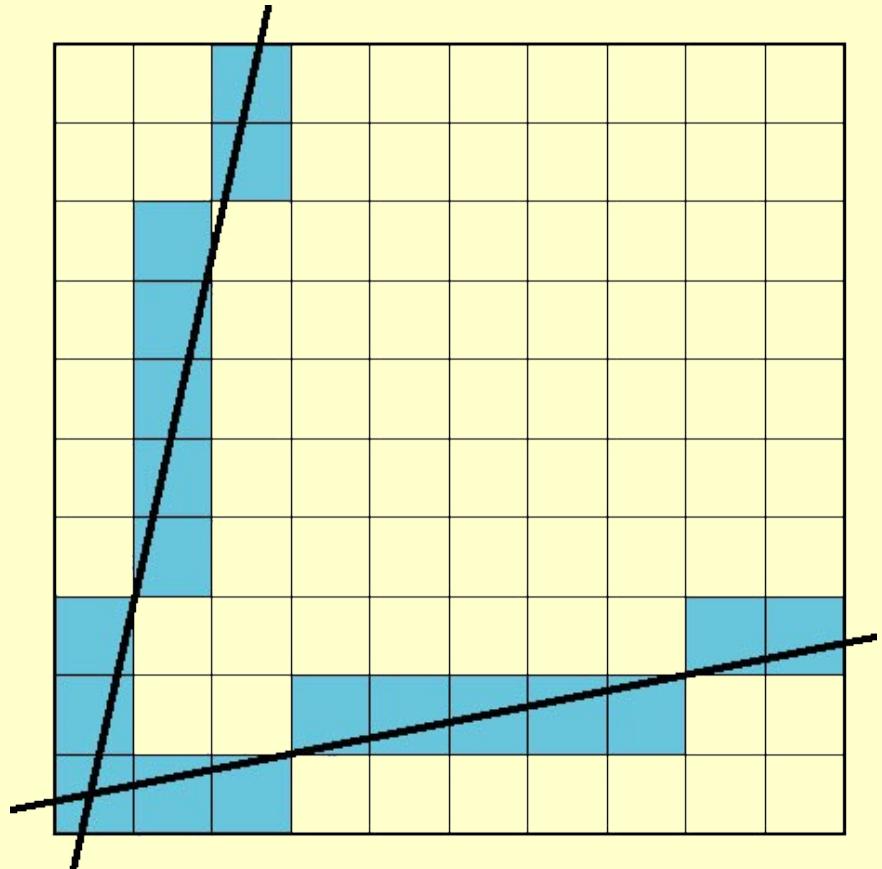
- DDA = for each x plot pixel at closest y
 - ~~Problems for steep lines~~





Using Symmetry

- Use for $1 \geq m \geq 0$
- For $m > 1$, swap role of x and y
 - For each y , plot closest x





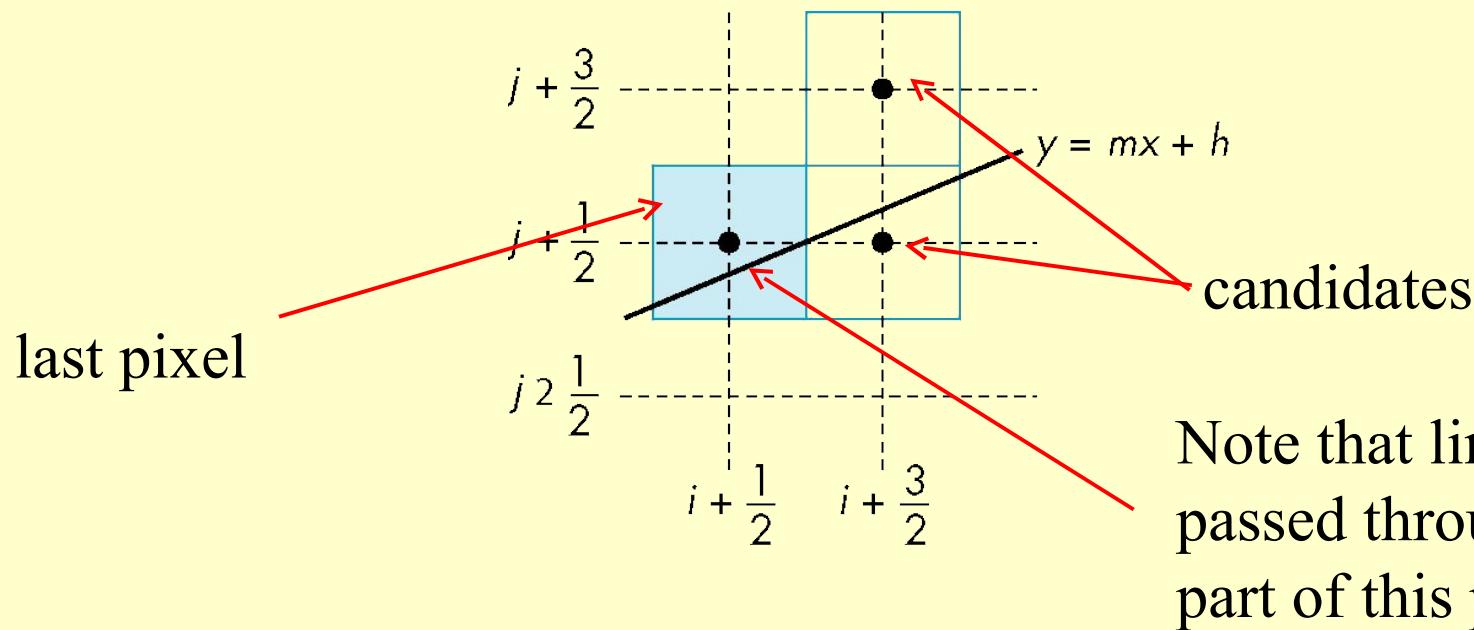
Bresenham's Algorithm

- DDA requires one floating point addition per step
- We can eliminate all fp through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
 - ~~Other cases by symmetry~~
 - Assume pixel centers are at half integers
 - If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer



Candidate Pixels

$$1 \geq m \geq 0$$



Note that line could have passed through any part of this pixel



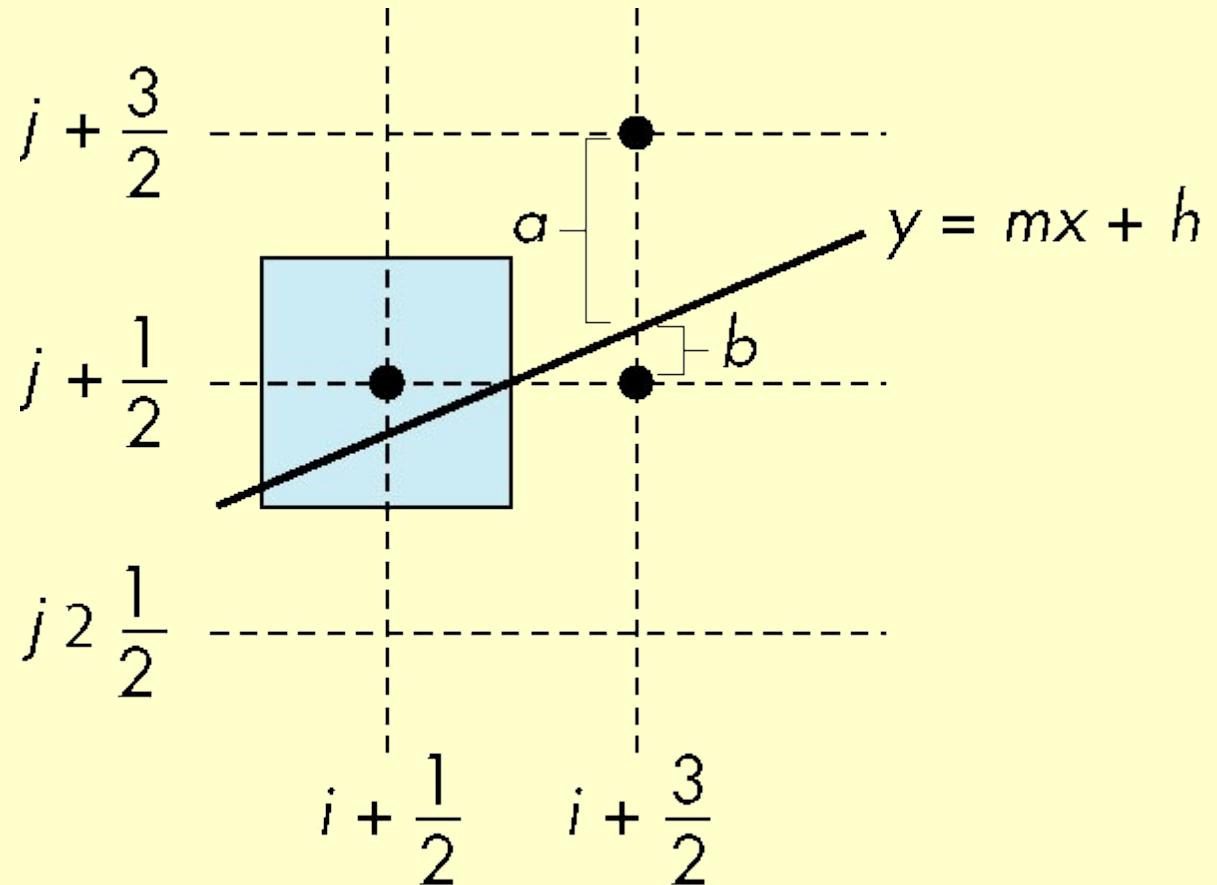
Decision Variable

$$d = \Delta x(b-a)$$

d is an integer

$d > 0$ use upper pixel

$d < 0$ use lower pixel





Incremental Form

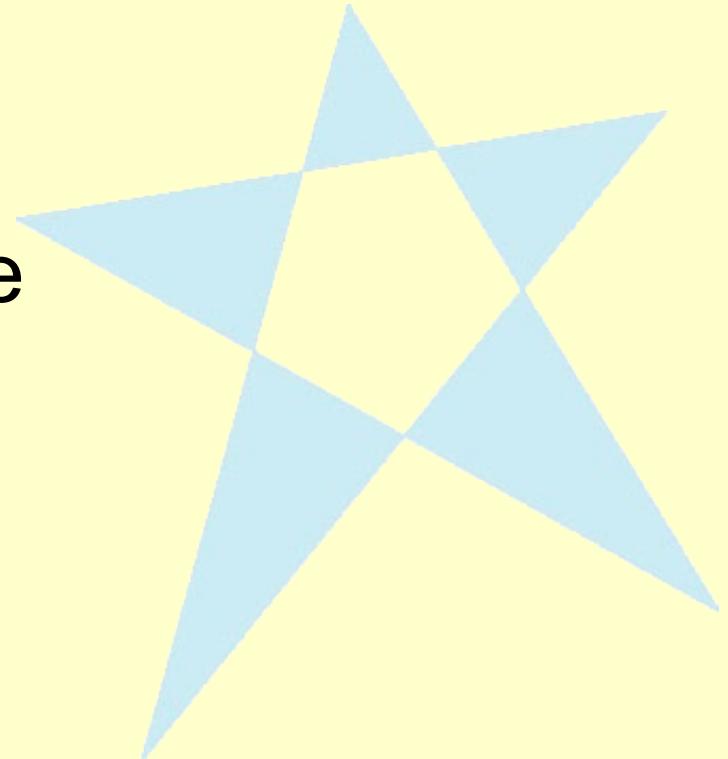
- More efficient if we look at d_k , the value of the decision variable at $x = k$
$$d_{k+1} = d_k - 2Dy, \text{ if } d_k < 0$$

$$d_{k+1} = d_k - 2(Dy - Dx), \text{ otherwise}$$
- For each x , we need do only an integer addition and a test
- Single instruction on graphics chips



Polygon Scan Conversion

- Scan Conversion = Fill
- How to tell inside from outside
 - Convex easy
 - Nonsimple difficult
 - Odd even test
- Count edge crossings



odd-even fill



Filling in the Frame Buffer

- Fill at end of pipeline
 - Convex Polygons only
 - Nonconvex polygons assumed to have been tessellated
 - Shades (colors) have been computed for vertices (Gouraud shading)
 - Combine with z-buffer algorithm
- March across scan lines interpolating shades
- Incremental work small



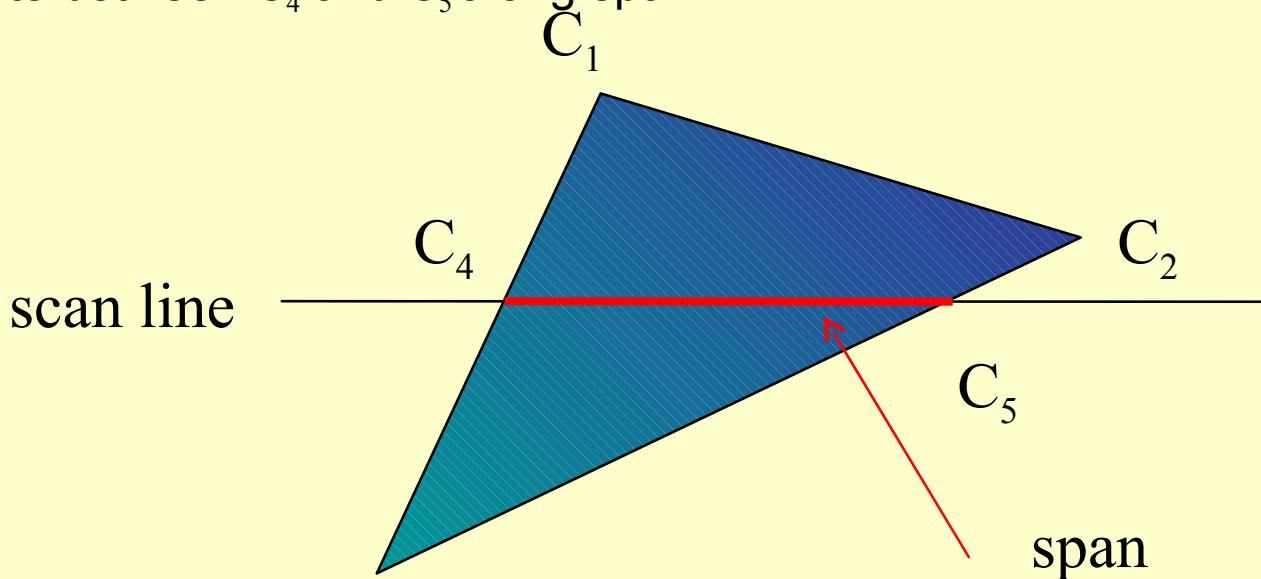
Using Interpolation

$C_1 C_2 C_3$ specified by `glColor` or by vertex shading

C_4 determined by interpolating between C_1 and C_2

C_5 determined by interpolating between C_2 and C_3

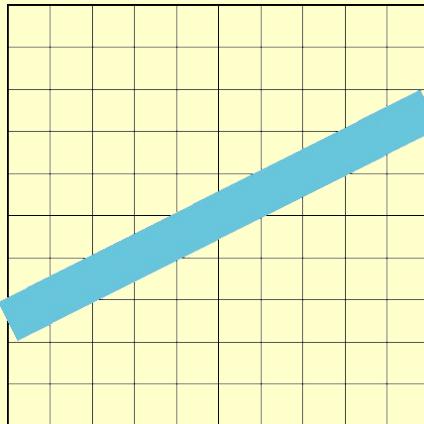
interpolate between C_4 and C_5 along span





Aliasing

- Ideal rasterized line should be 1 pixel wide

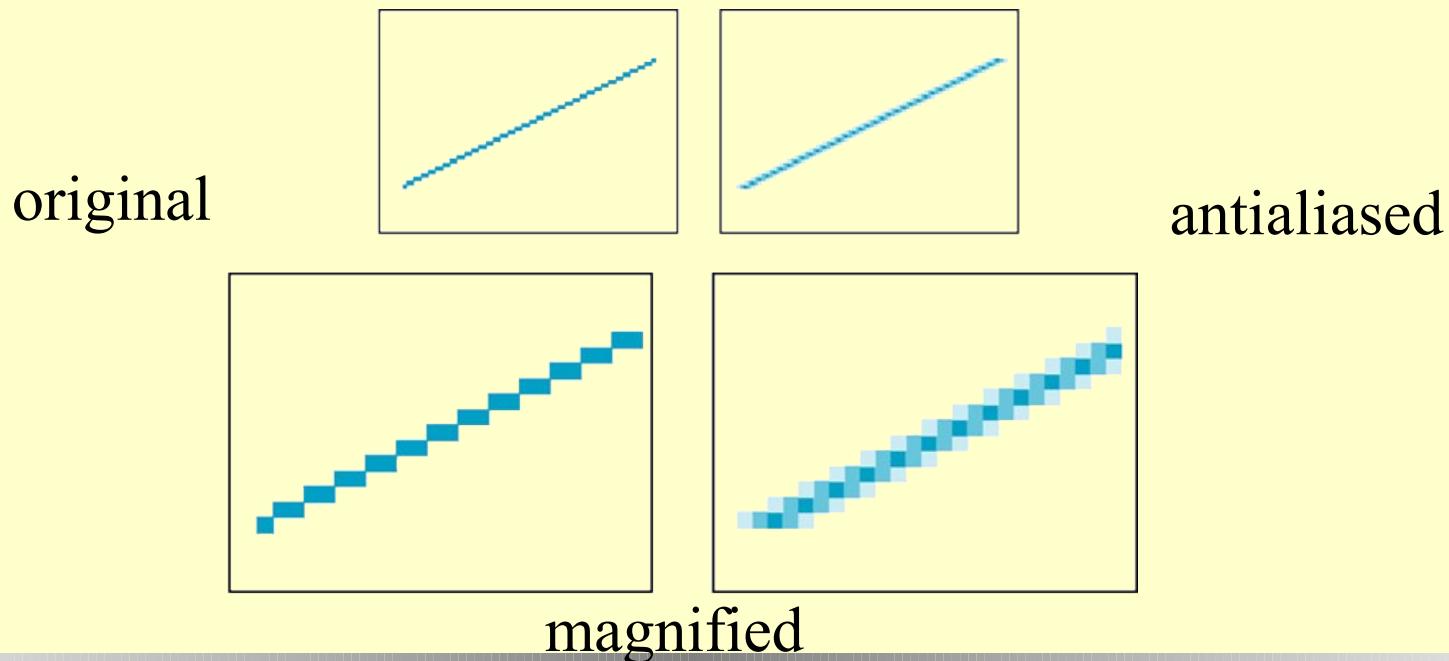


- Choosing best y for each x (or visa versa) produces aliased raster lines



Antialiasing by Area Averaging

- Color multiple pixels for each x depending on coverage by ideal line





Polygon Aliasing

- Aliasing problems can be serious for polygons
 - Jaggedness of edges
 - Small polygons neglected
 - Need compositing so color of one polygon does not totally determine color of pixel
- All three polygons should contribute to color

