

Graphics Systems and Models

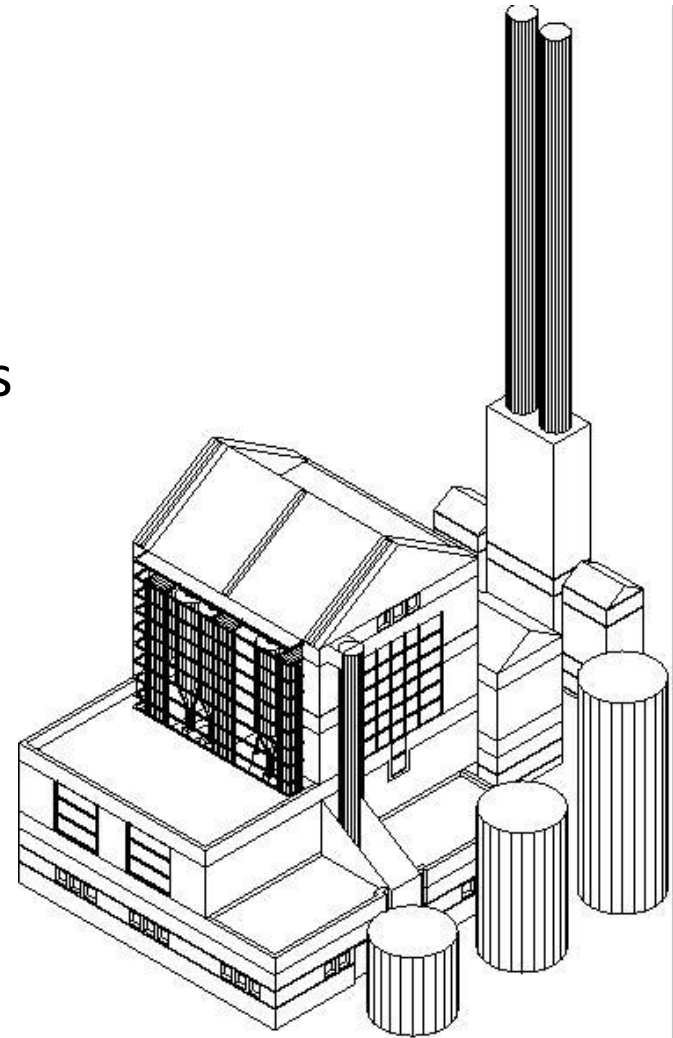
Chapter 1

- Introduction:
 - Computer Graphics
 - What is it?
 - Overview of what we will cover:
 - A graphics overview
 - Graphics Theory
 - A graphics Software System: OpenGL
 - Our approach will be top-down.
 - We want you to start writing application programs that generate graphical output as quickly as possible

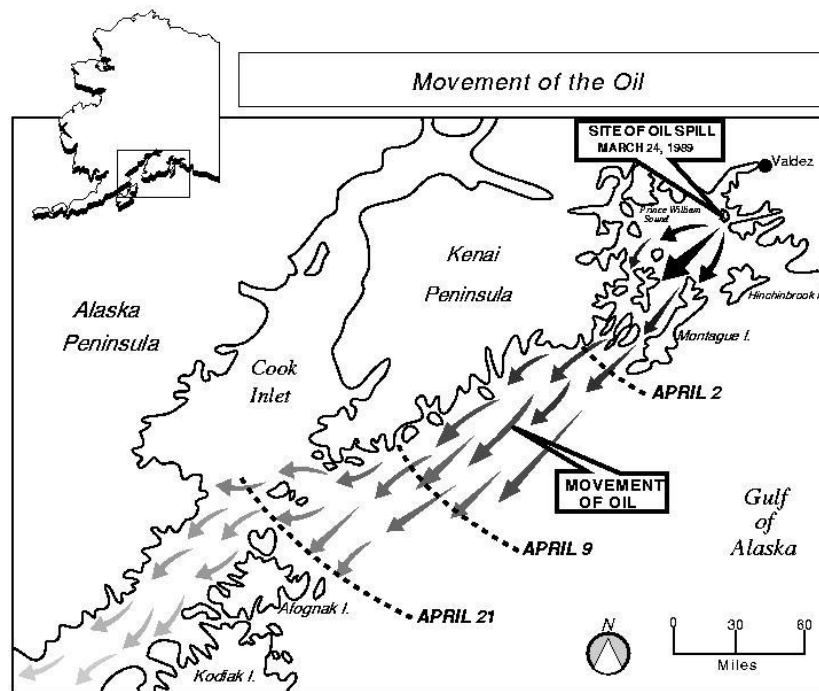
1. Applications of Computer Graphics

- The development of Computer Graphics has been driven by the needs of the user community and by the advances in hardware and software.
- Applications can be split into four major areas:
 - Display of information
 - Design
 - Simulation
 - User Interfaces

- 1.1 Display of Information
 - Classical graphics techniques arose as a medium to convey information among people:
 - 4,000 years ago -- Babylonians: floor plans of buildings on stones
 - 2,000 years ago -- Greeks: Architectural ideas
 - Now we have Computer-based drafting programs



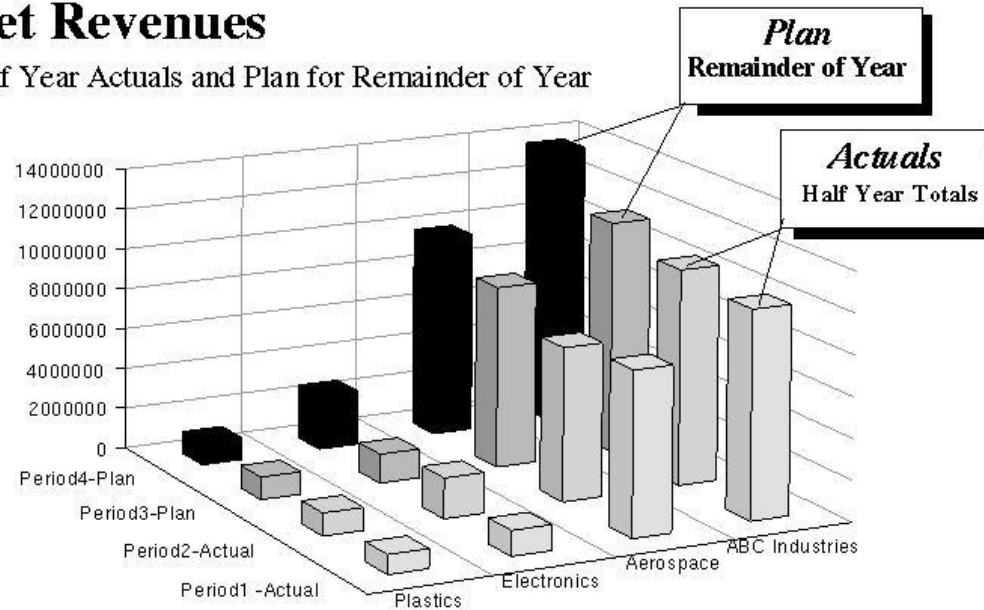
- For centuries -- Cartographers have developed maps to display celestial and geographical information.
 - Now maps can be developed and manipulated in real-time over the internet.



- Over the past 100 years -- workers in statistics have explored techniques for generating plots to convey information
 - Now we have computer plotting packages

Net Revenues

Half Year Actuals and Plan for Remainder of Year



- Medicine poses interesting and important data-analysis problems
 - CAT Scans, MRI's ultrasound, and other 3D data producing technologies



- The field of Scientific Visualization provides graphical tools that help these researchers and others interpret the vast quantities of data generated

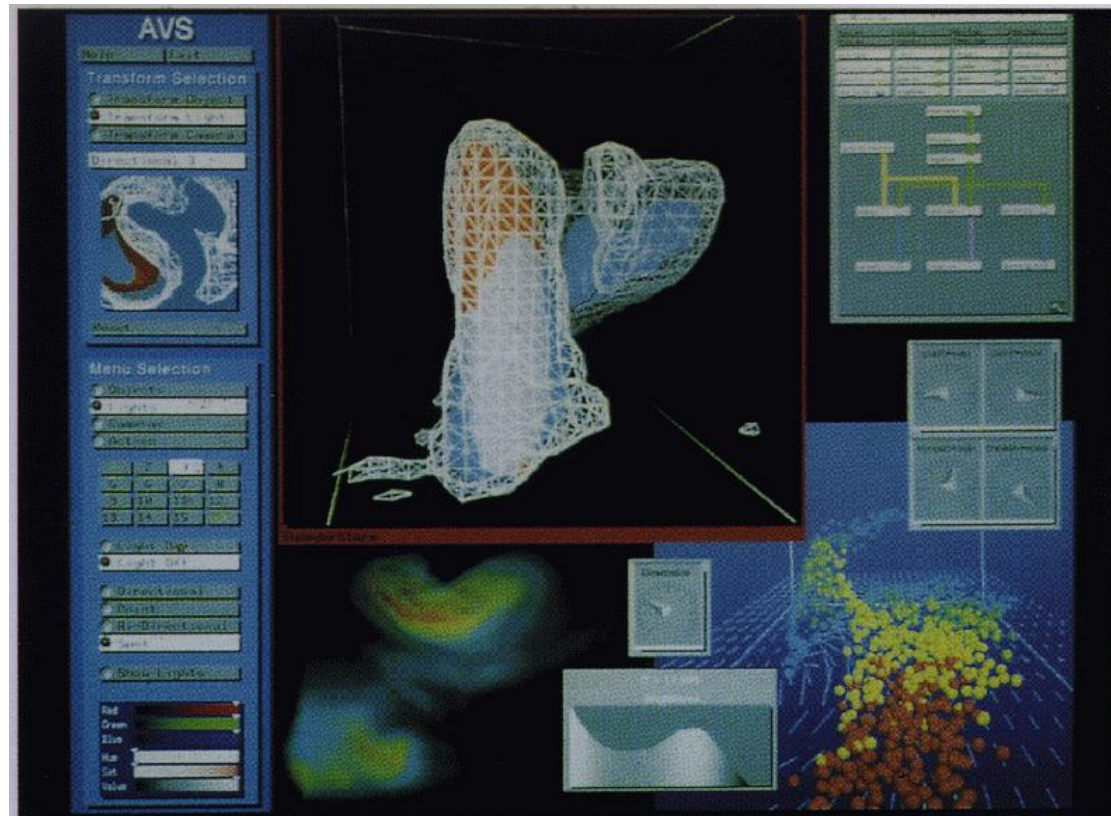


Plate 1. Severe tornadic storm, by R. Wilhelmson, L. Wicker, and C. Shaw, NCSA, University of Illinois. (Application Visualization System by Stardent Computer.)

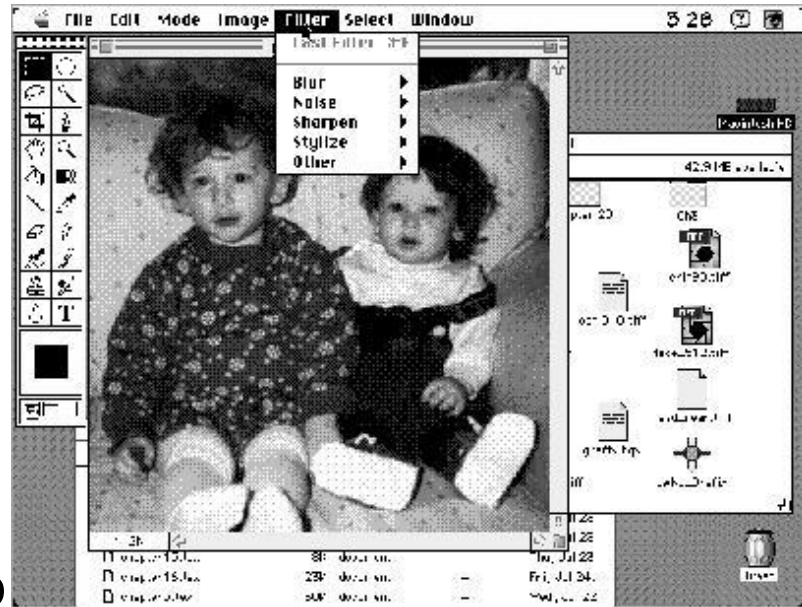
- 1.2 Design
 - Engineering and Architecture are concerned with design
 - starting with a set of specification
 - seek a cost-effective (and esthetic) solution
 - This is an iterative process
 - The power of interacting with images on the screen
 - has been known for at least 40 years.
 - and today the use of interactive tools pervades the CAD field in areas such as architecture and VLSI design

- 1.3 Simulation
 - When did graphics begin to be used?
 - Why?
 - The field of VR has opened up many new horizons.
 - Same (or different) image in each eye
 - position tracking
 - interactive devices
 - This has led to training capabilities
 - NASA research grant

- 1.4 User Interfaces

- Our interaction with computers has become dominated by a visual paradigm that includes:

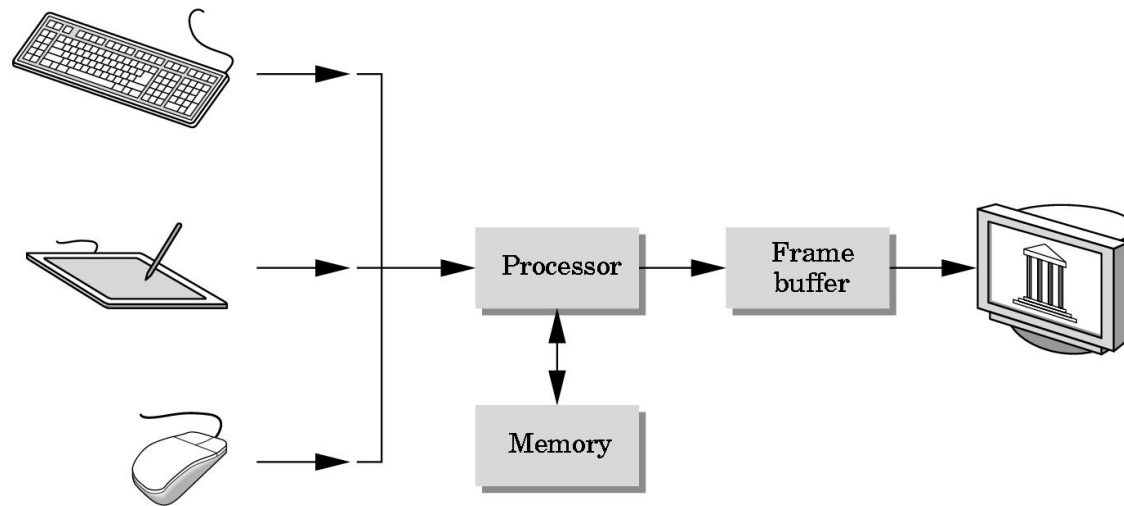
- windows,
 - icons,
 - menus, and
 - a pointing device



- We have become so used to this graphical user interface that we often forget that what we are doing is working with computer graphics.

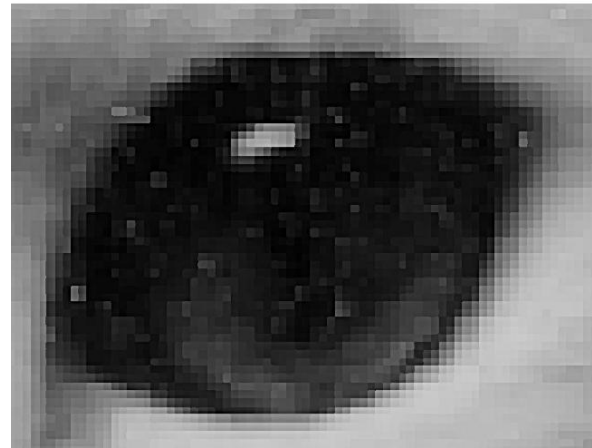
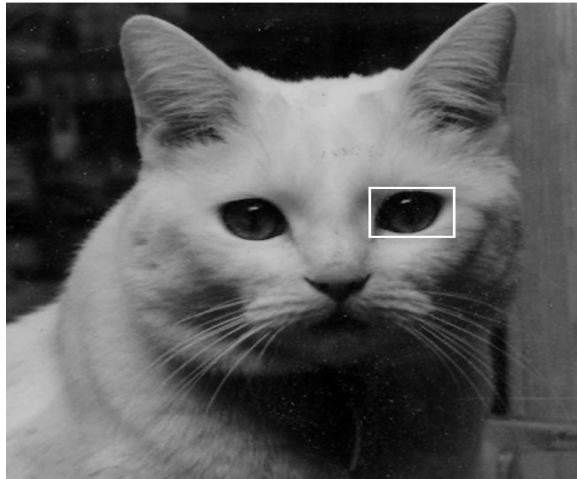
2. A Graphics System

– There are 5 major elements in our system



- Processor, Memory, Frame Buffer, Input Devices, Output Devices

- 2.1 Pixels and the Frame Buffer
 - At present, almost all graphics systems are raster-based
 - A picture is produced from an array (the raster) of picture elements (pixels)



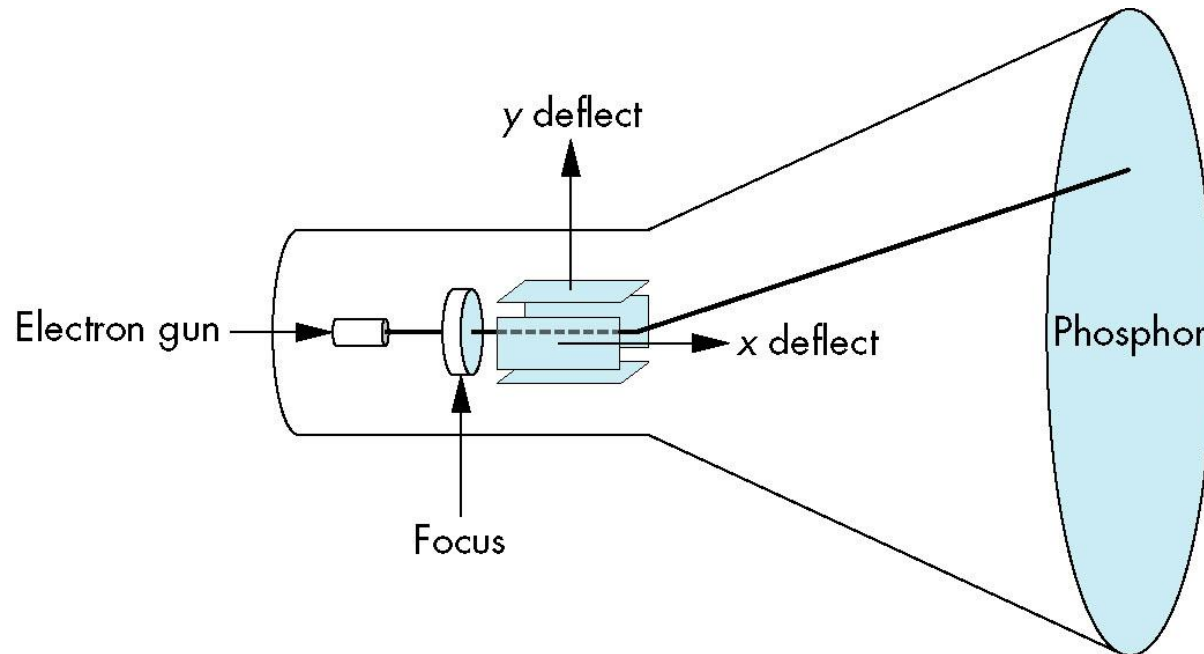
- Def: depth of the frame buffer

– Definitions:

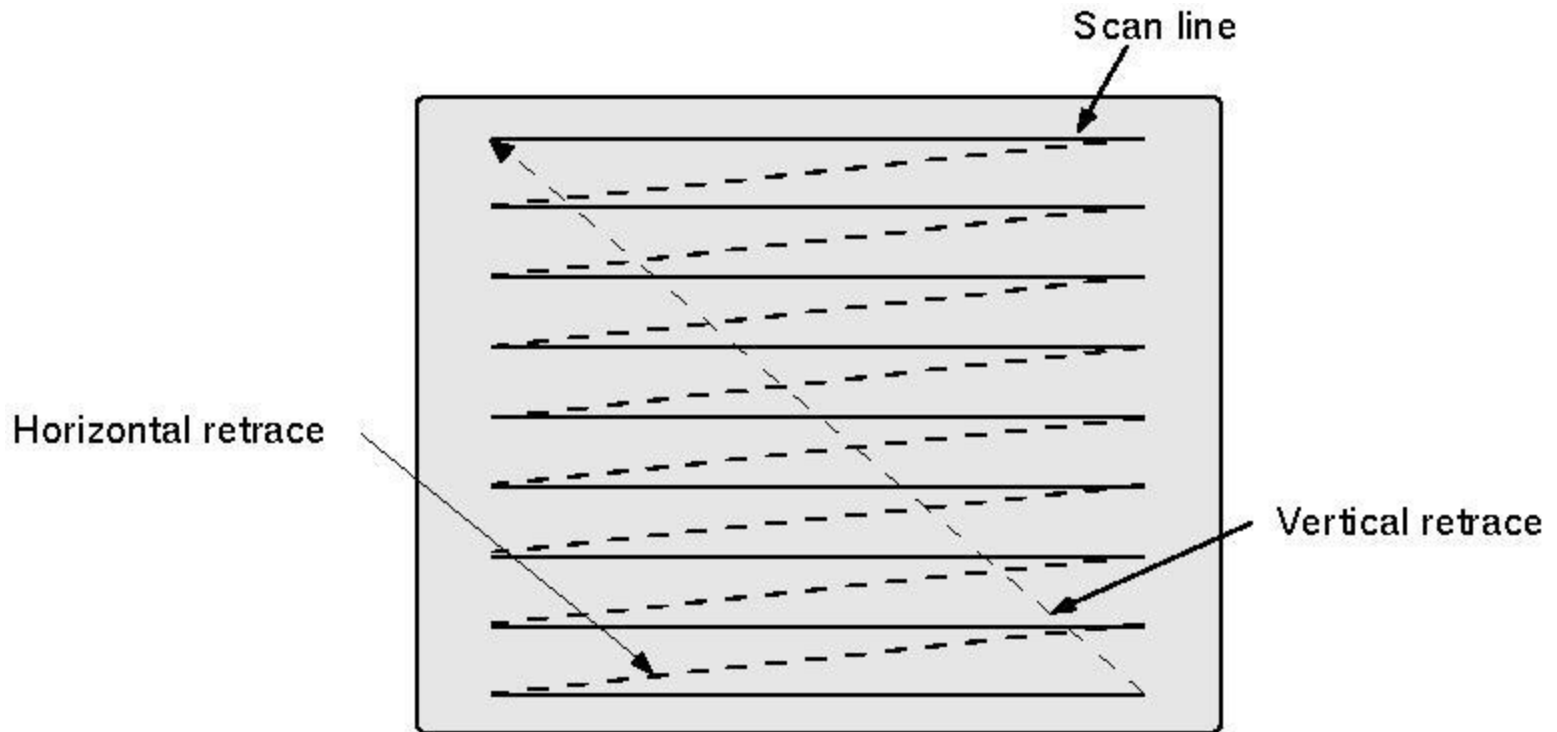
- Depth of the frame buffer
 - 1-bit
 - 8-bit deep
 - full-color is 24-bit or more
- Resolution
 - the number of pixels in the frame buffer
- Rasterization or scan conversion
 - The converting of geometric primitives into pixel assignments in the frame buffer

- 2.2 Output Devices

- The dominate type of display is the Cathode-ray tube (CRT)

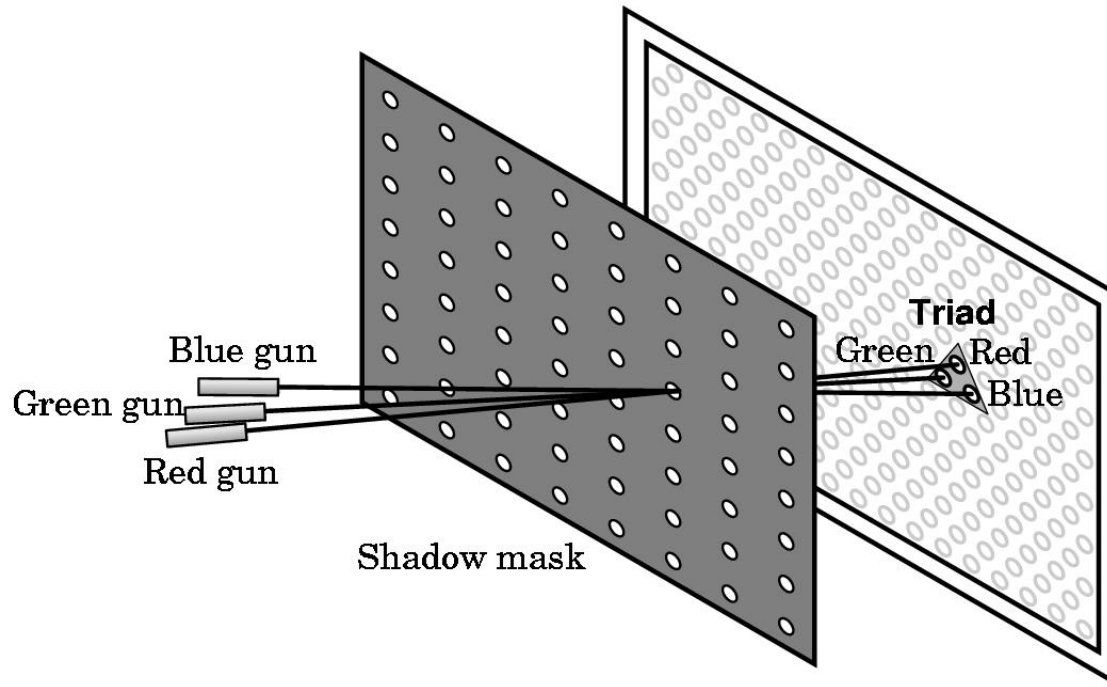


- Def: refresh rate



– Color CRT's

- have three different colored phosphors arranged in small groups (typically triads).



- 2.3 Input Devices

- Most graphics systems provide a keyboard and at least one other input device

- mouse

- joystick

- data tablet

- We will study these devices in Chapter 3

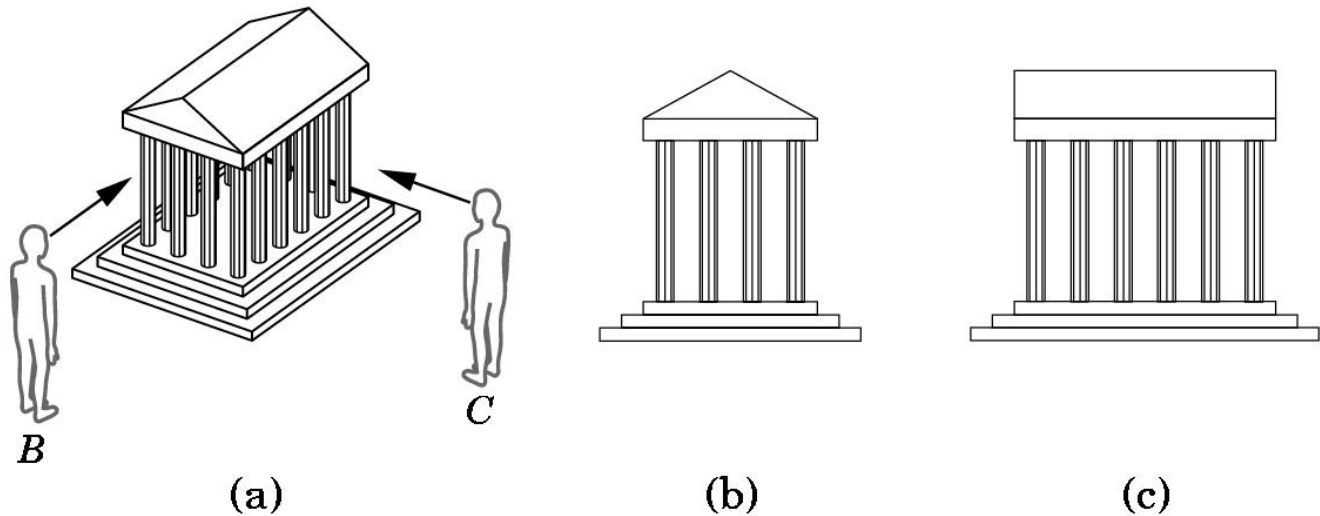
3. Images: Physical and Synthetic

- The Usual pedagogical approach:
 - construct raster images
 - simple 2D entities (points, lines, polygons)
 - Define objects based upon 2D
 - Because such functionality is supported by most present computer graphics systems, we are going to learn to create images here, rather than expand a limited model.

- 3.1 Objects and Viewers

- Two basic entities must be part of any image-formation process:

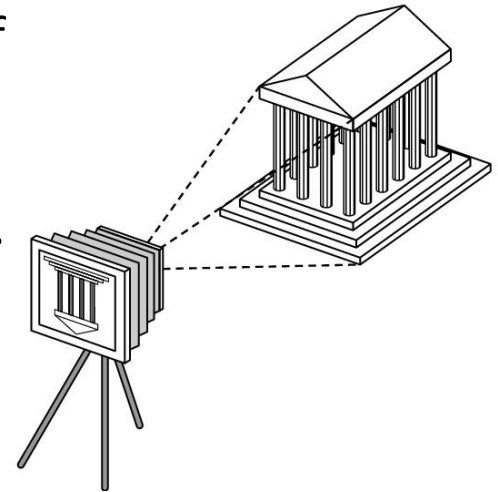
- the object
- the viewer



- The object exists in space independent of any image-formation process, and of any viewer.

- What we will study Now

- Both the object and the viewer exist in a 3D world. However, the image they define is 2D
- The process by which the specification of the object is combined with the specification of the viewer to produce an image is the essence of image formation.

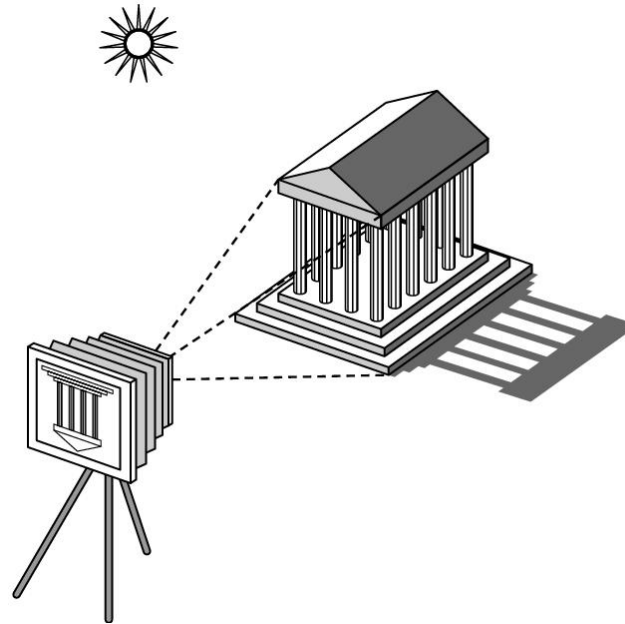


- Future:

- In Chapter 2, we show how OpenGL allows us to build simple objects
- In Chapter 9 we learn how to define objects in a manner that incorporates relationships among objects.

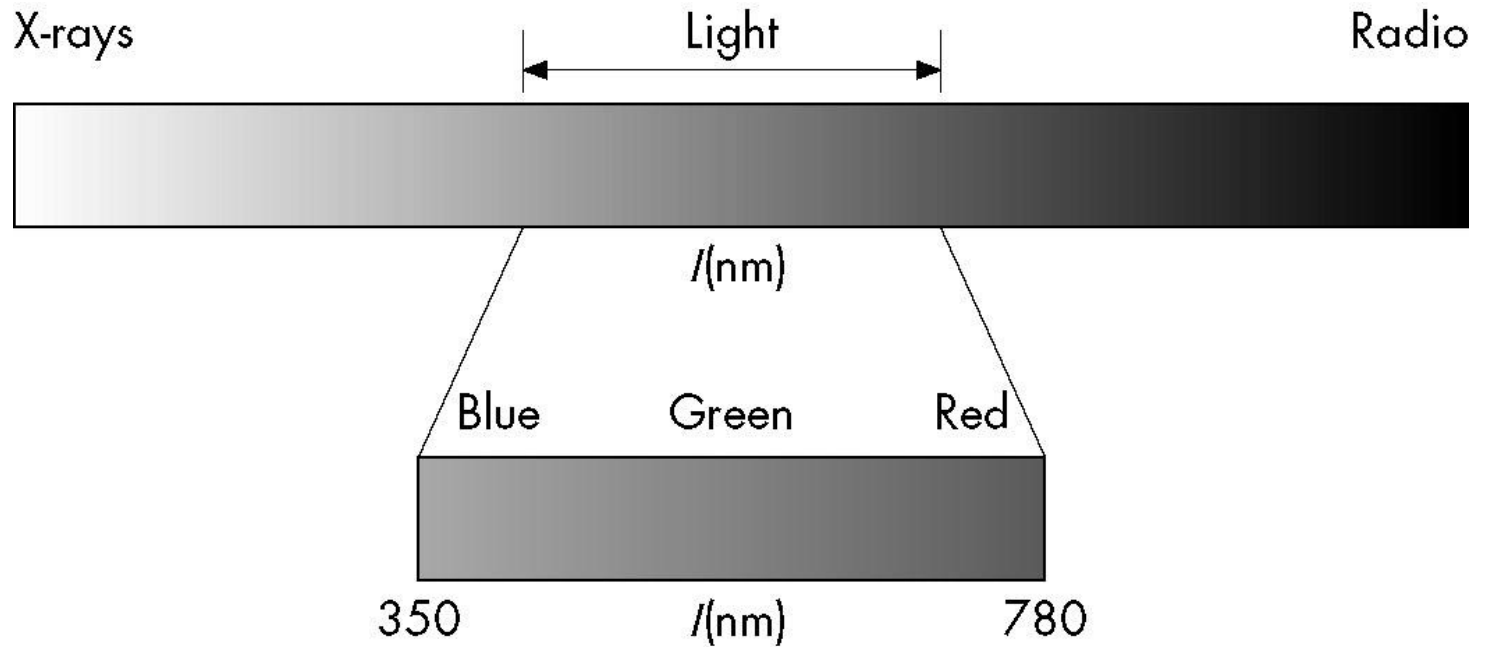
- 3.2 Light and Images
 - Much information was missing from the preceding picture:
 - We have yet to mention light!
 - If there were no light sources the objects would be dark, and there would be nothing visible in our image.
 - We have not mentioned how color enters the picture
 - Or, what are the effects of different kinds of surfaces on the objects.

- Taking a more physical approach, we can start with the following arrangement:

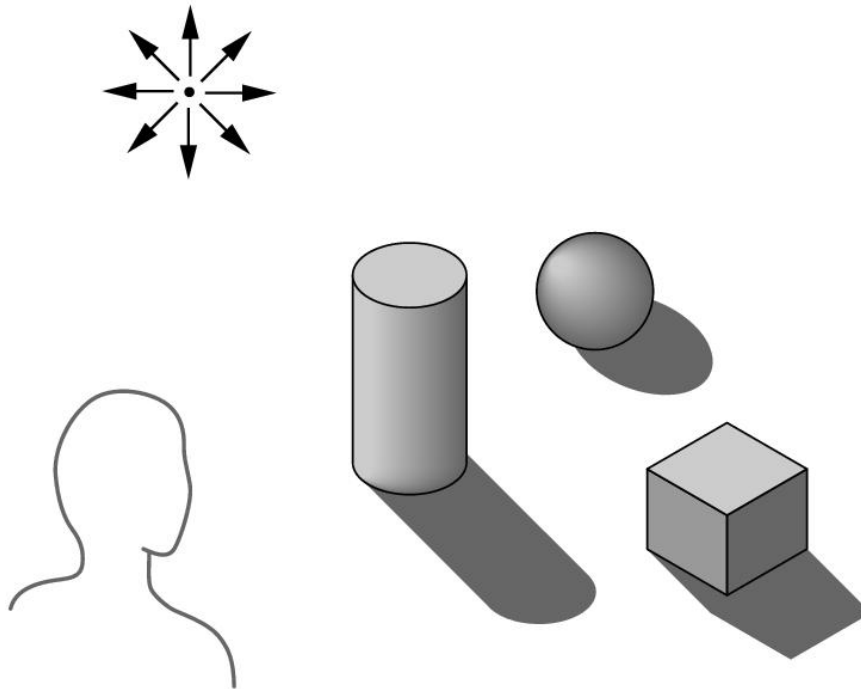


- The details of the interaction between light and the surfaces of the object determine how much light enters the camera.

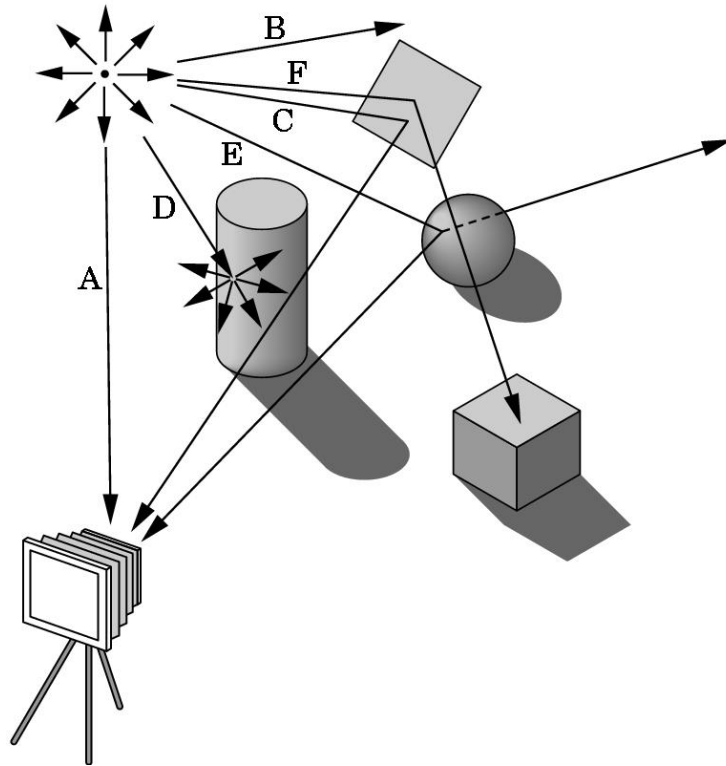
- Light is a form of electromagnetic radiation:



- 3.3 Ray Tracing
 - We can start building an imaging model by following light from a source.

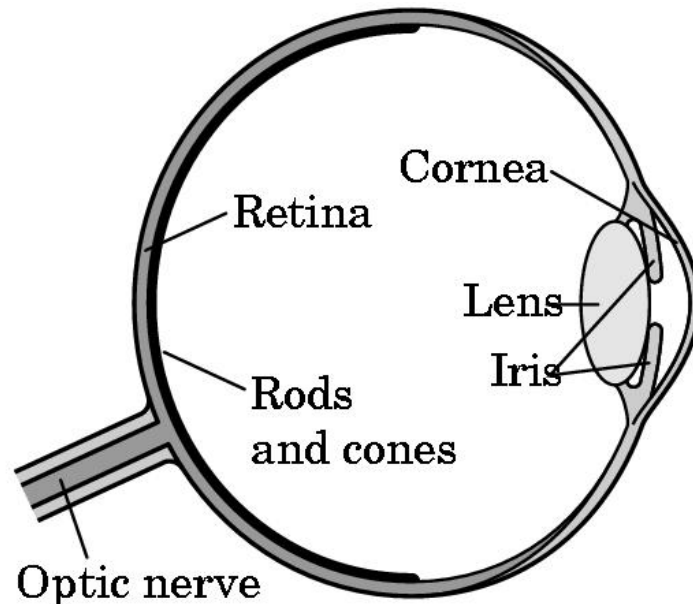


- Ray tracing is an image formation technique that is based on these ideas and that can form the basis for producing computer generated images.



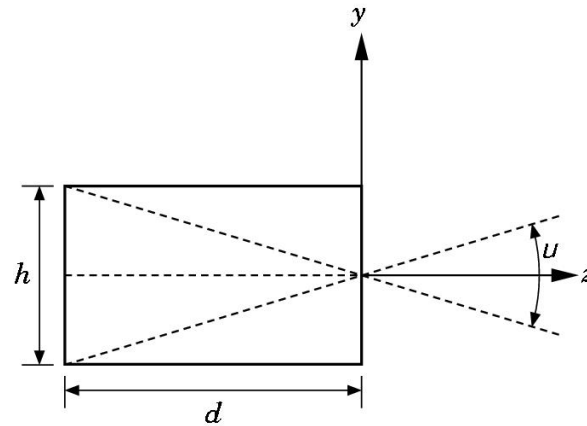
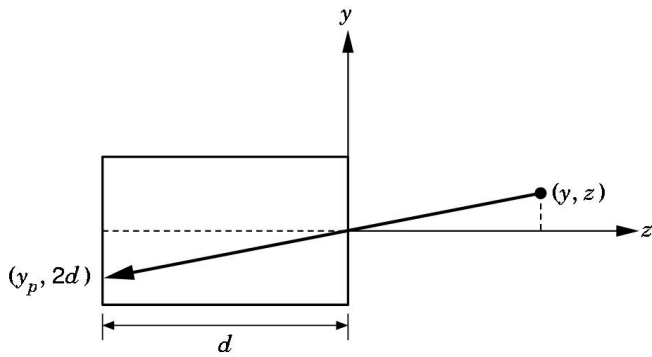
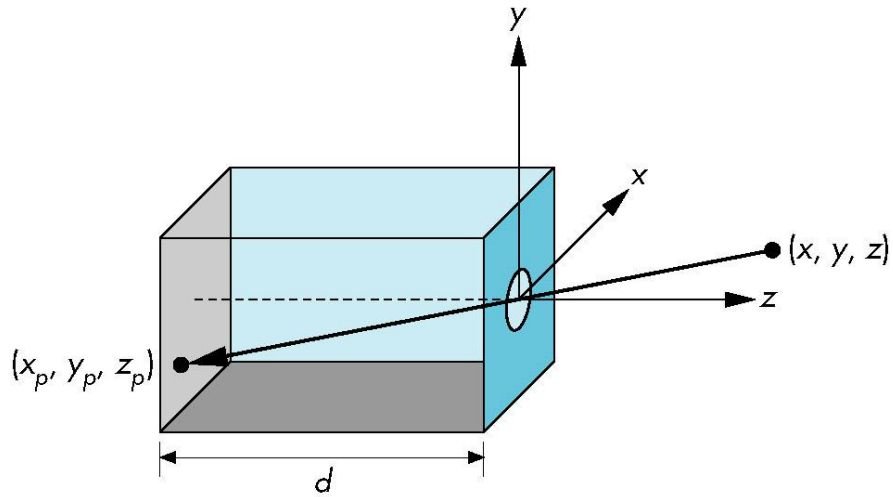
4. The Human Visual System

- Our extremely complex visual system has all the components of a physical imaging system, such as a camera or a microscope.



5. The Pinhole Camera

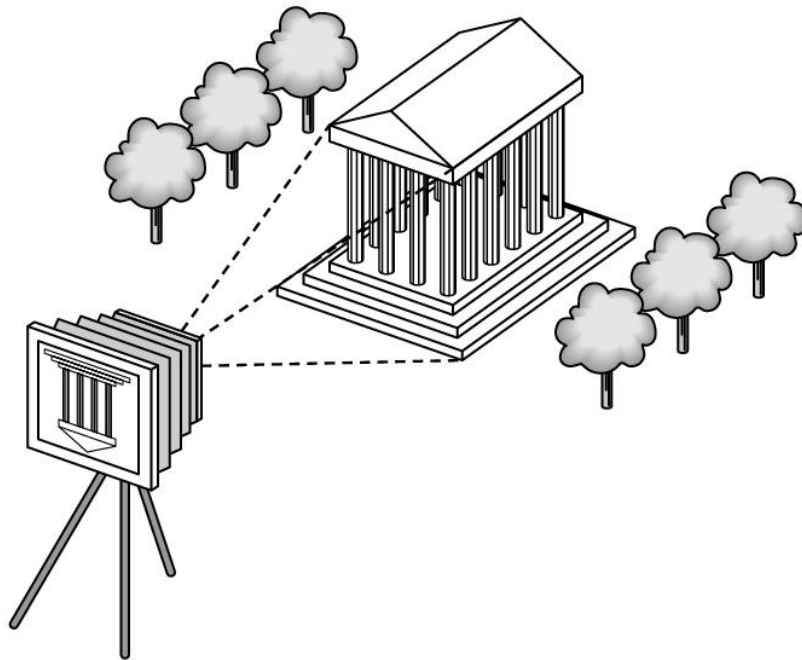
- What is one?
- A Pinhole camera is a box
 - with a small hole in the center on one side,
 - and the film on the opposite side



– Pinhole camera near Cliff House in San Francisco

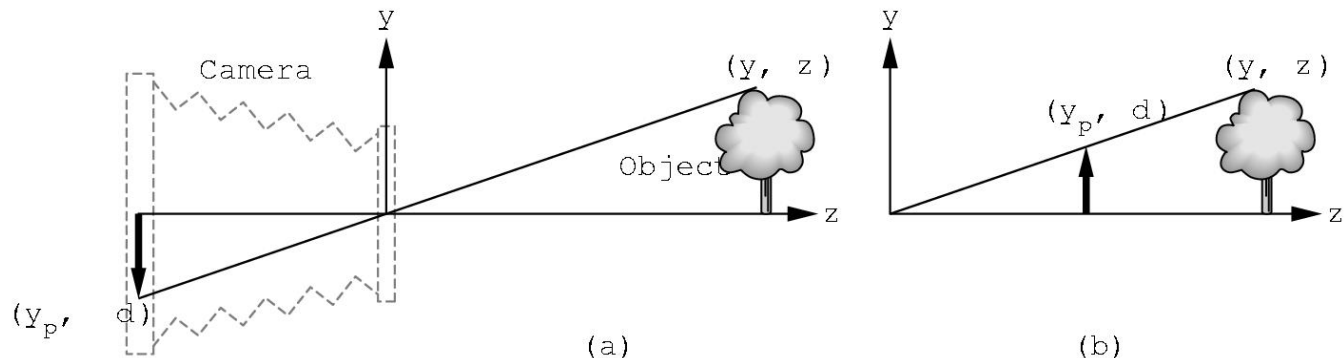
6. The Synthetic Camera Model

– Consider the imaging system shown here:

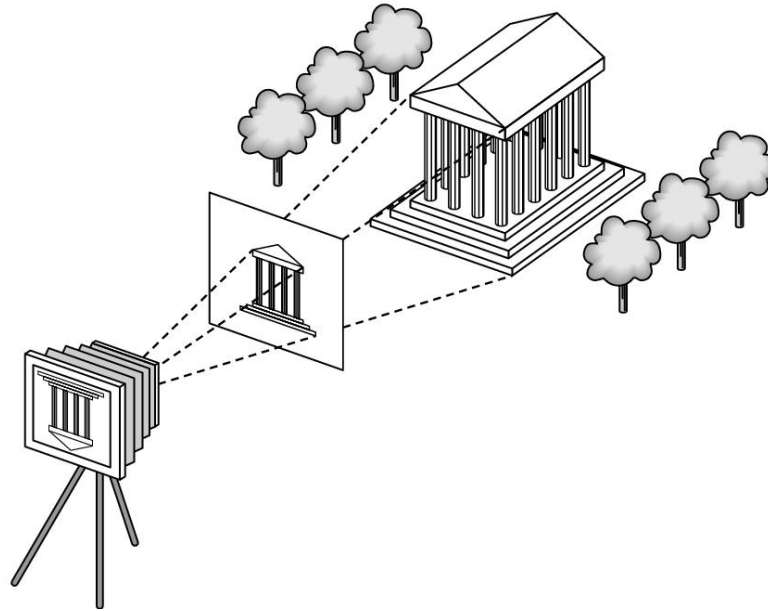


– A few Basic principles:

- First, the specification of the objects is independent of the specification of the viewer.
 - In a graphics library we would expect separate functions for specifying objects and the viewer.
- Second, we can compute the image using simple trigonometric calculations

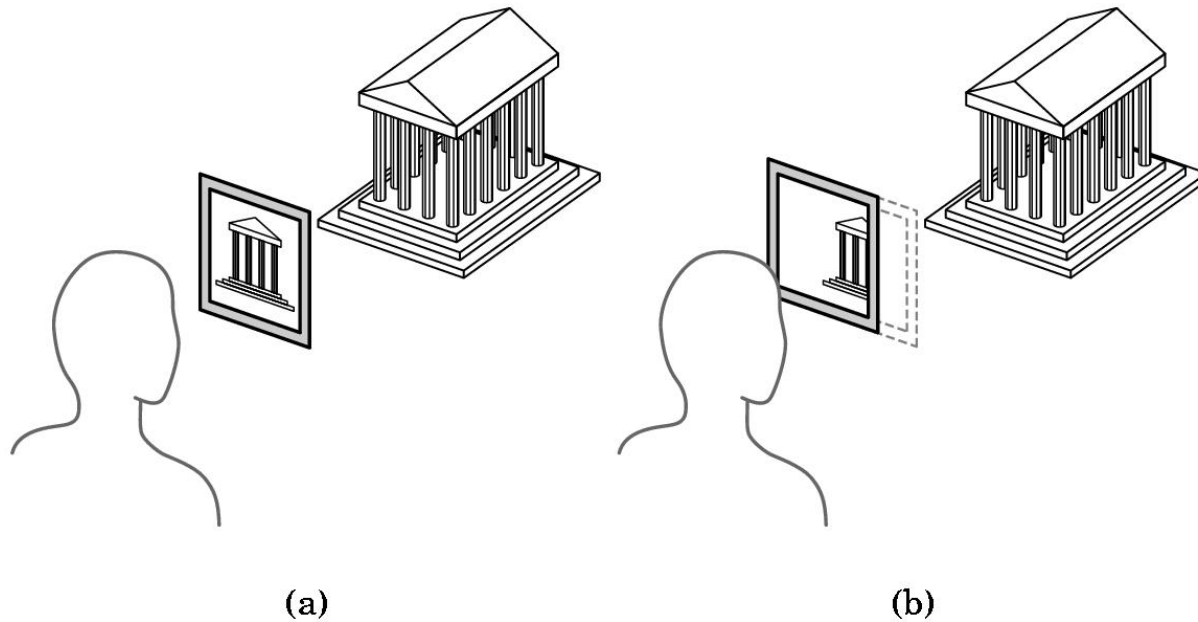


- Because of this, we can move the image plane in front of the lens (call this the projection plane) and end up with:



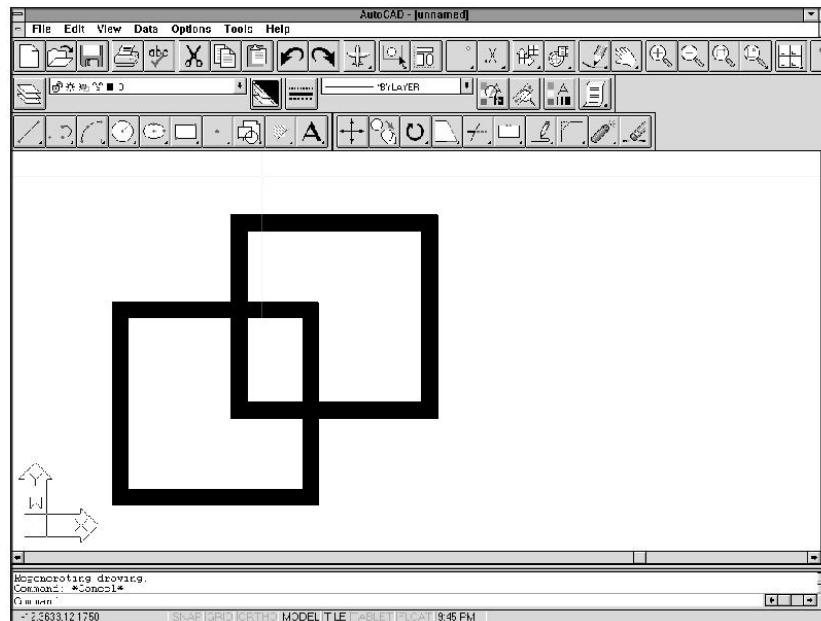
- In Chapter 5, we discuss this process in detail and derive the relevant mathematical formulas

- We must also consider the limited size of the image.
- Therefore, we must discuss clipping:

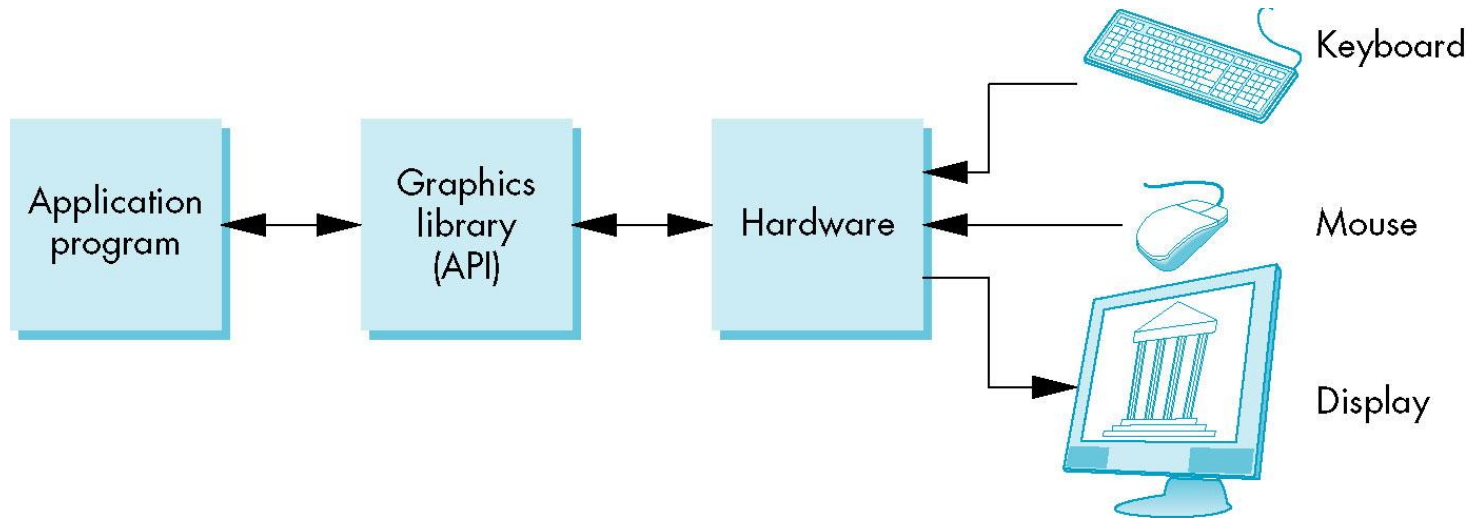


7. The Programmer's Interface

- There are numerous ways that a user can interact with a graphics system.
 - In a typical paint program it would look like:



- 7.1 Application Programmer's Interfaces
 - What is an API?

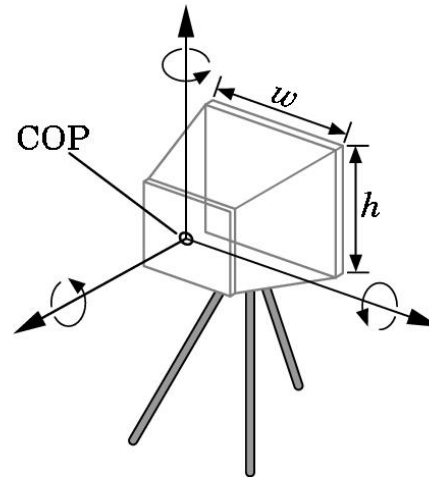


- Why do you want one?

- If we are to follow the synthetic camera model, we need functions in the API to specify:
 - Objects
 - Viewer
 - Light Sources
 - Material Properties

- Objects are usually defined by a set of vertices
 - The following code fragment defines a triangular polygon in OpenGL
 - 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();

- We can define a viewer or camera in a variety of ways



- We can identify four types of necessary specifications:
 - Position
 - Orientation
 - Focal length
 - Film plane

- Light sources can be defined by their location, strength, color, and directionality.
 - API's provide a set of functions to specify these parameters for each source.

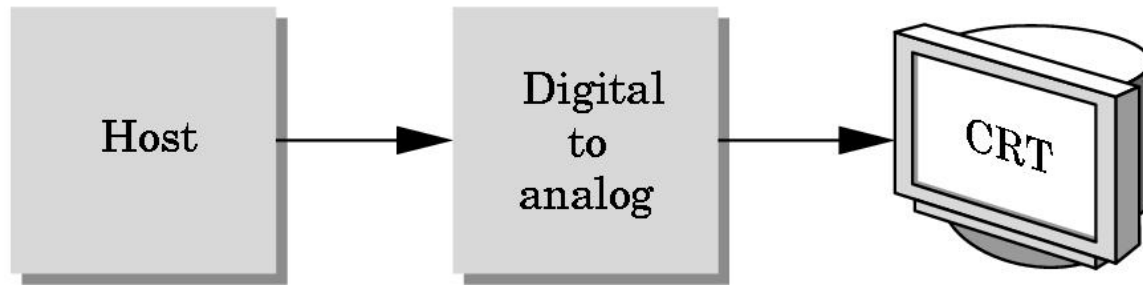
- Material properties are characteristics, or attributes, of the objects
 - such properties are usually defined through a series of function calls at the time that each object is defined.

- 7.2 A Sequence of Images
 - In Chapter 2 , we begin our detailed discussion of the OpenGL API
 - Color Plates 1 through 8 show what is possible with available hardware and a good API, but also they are not difficult to generate

8. Graphics Architectures

- On one side of the API is the application program. On the other is some combination of hardware and software that implements the functionality of the API
- Researchers have taken various approaches to developing architectures to support graphics APIs

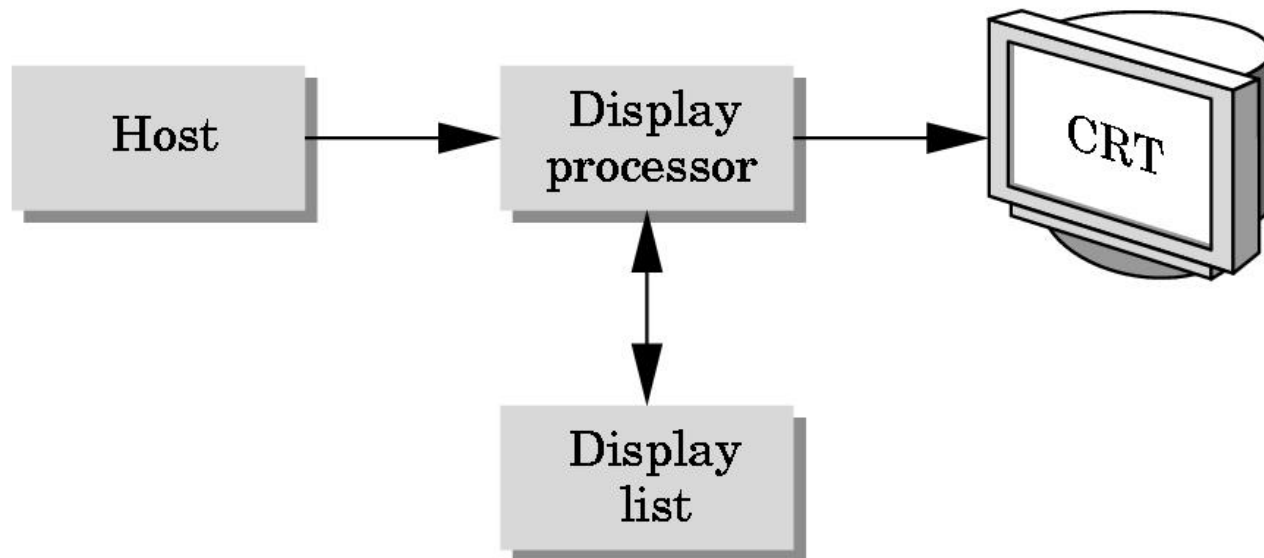
- Early systems used general purpose computers
 - single processing unit



- In the early days of computer graphics, computers were so slow that refreshing even simple images, containing a few hundred line segments, would burden an expensive computer

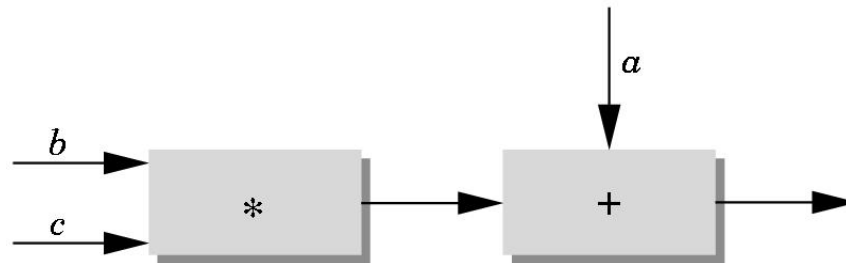
- 8.1 Display Processors

- The earliest attempts to build a special purpose graphics system were concerned primarily with relieving the task of refreshing the display

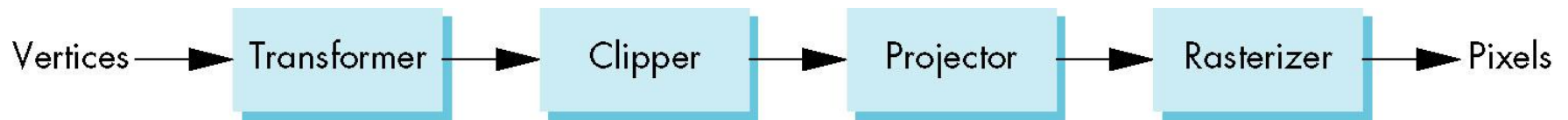


- ## 8.2 Pipeline Architectures

- The major advances in graphics architectures parallel closely the advances in workstations.
- For computer graphics applications, the most important use of custom VLSI circuits has been in creating pipeline architectures
- A simple arithmetic pipeline is shown here:



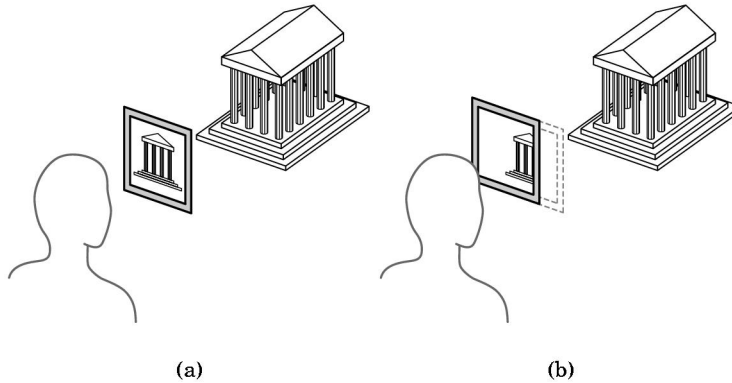
- If we think in terms of processing the geometry of our objects to obtain an image, we can use the following block diagram:



- We will discuss the details of these steps in subsequent chapters.

- 8.3 Transformations
 - Many of the steps in the imaging process can be viewed as transformations between representations of objects in different coordinate systems
 - for example: from the system in which the object was defined to the system of the camera
 - We can represent each change of coordinate systems by a matrix
 - We can represent successive changes by multiplying (or concatenating) the individual matrices into a single matrix.

- 8.4 Clipping
 - Why do we Clip?



- Efficient clipping algorithms are developed in Chapter 7

- 8.5. Projection

- In general three-dimensional objects are kept in three dimensions as long as possible, as they pass through the pipeline.
- Eventually though, they must be projected into two-dimensional objects.
- There are various projections that we can implement.
- We shall see in Chapter 5 that we can implement this step using 4 x 4 matrices, and, thus, also fit it into the pipeline.

- 8.6 Rasterization
 - Finally, our projected objects must be represented as pixels in the frame buffer.
 - We discuss this scan-conversion or rasterization process in Chapter 7

- 8.7 Performance Characteristics
 - There are two fundamentally different types of processing
 - Front end -- geometric processing, based on the processing of vertices
 - ideally suited for pipelining, and usually involves floating-point calculations.
 - The geometry engine developed by SGI was a VLSI implementation for many of these operations in a special purpose chip. These became the basis for a series of graphics workstations.
 - Back end -- involves direct manipulation of bits in the frame buffer.
 - Ideally suited for parallel bit processors.

9. Summary

- In this chapter we have set the stage for our top-down development of computer graphics.
 - Computer graphics is a method of image formation that should be related to classical methods -- in particular to cameras
 - Our next step is to explore the application side of Computer Graphics programming
 - We will use the OpenGL API

10. Suggested Readings

- Journals:
 - *Computer Graphics -- ACM*
 - *IEEE Compute Graphics and Applications*
- Textbooks:
 - Foley et.al.
 - Hearn and Baker
 - Hill



UNIT-2

Input and Interaction

Outlines

- ❖ Interaction
- ❖ Input Devices
- ❖ Clients & servers
- ❖ Display lists and modeling
- ❖ Programming Event-Driven Input
 - Menus, picking
- ❖ Graphical User Interface
- ❖ Animating Interactive Program
- ❖ Logical Operation

Introduction

❖ Basic Interactive Paradigm

- It is established by Ivan Sutherland (MIT 1963) to characterize **interactive computer graphics**
 - **Step 1:** User sees an *object* on the display
 - **Step 2:** User points to (*picks*) the object with an input device (light pen, mouse, trackball)
 - **Step 3:** Object changes (moves, rotates, morphs)
 - **Step 4:** Return to Step 1.

Interaction

- ❖ **In the field of computer graphics, interaction refers to the manner in which the application program communicates with input and output devices of the system.**
- ❖ **For e.g. Image varying in response to the input from the user.**
- ❖ **OpenGL doesn't directly support interaction in order to maintain portability. However, OpenGL provides the GLUT library. This library supports interaction with the keyboard, mouse etc and hence enables interaction. The GLUT library is compatible with many operating systems such as X windows, Current Windows, Mac OS etc and hence indirectly ensures the portability of OpenGL.**



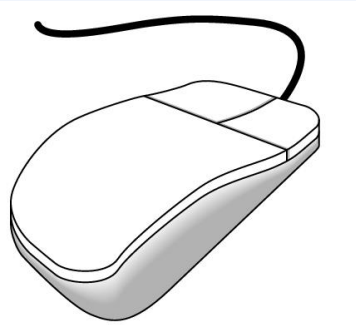
Introduction

❖ Input Devices

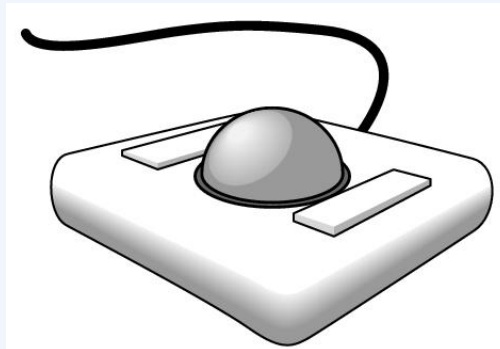
- The input devices can be considered in two distinct ways.
 - **Physical Devices:** is the way that we take to look
 - **Logical Devices:** is to treat the devices in terms of what they do from the perspective of the program.

Introduction

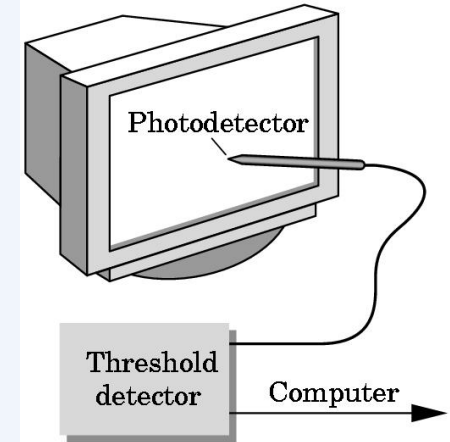
❖ Physical Devices



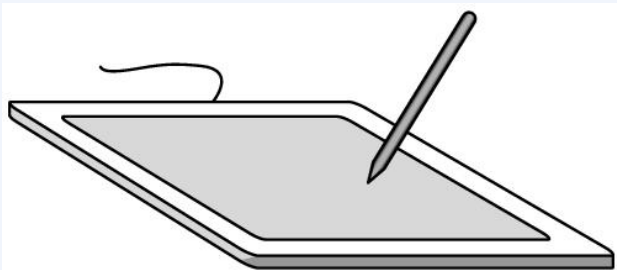
mouse



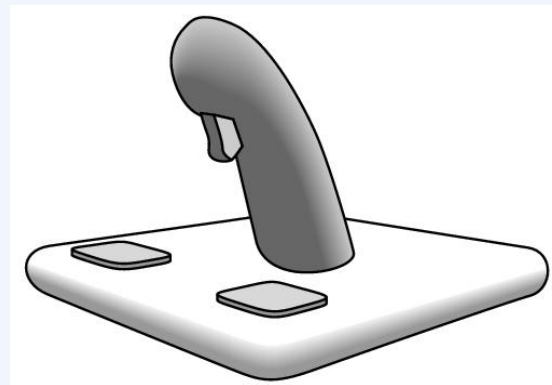
trackball



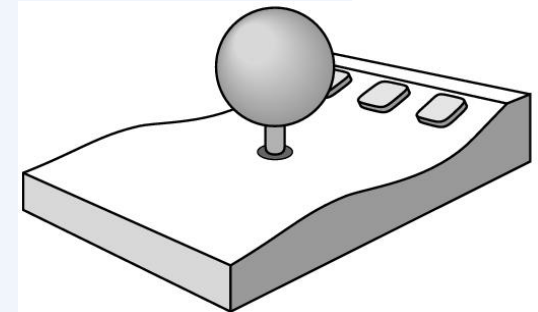
light pen



data tablet



joy stick



space ball

Introduction

❖ Logical Device

- The behavior of the logical devices is characterized by two factors.
 - Measurement.
 - Time to return the measurement.
- Six Types
 - **String**: ASCII strings
 - **Locator**: position
 - **Pick**: identifier of an object
 - **Choice**: one of the discrete items
 - **Valuator**: return floating point number
 - **Stroke**: return array of positions

Introduction

❖ Input Modes

- It can be described in terms of two entities
 - **Measure Process**: is what the device returns
 - **Trigger Process**: is a physical input device that the user can signal the computer.
- We can obtain the **measure** of a device in the three Modes
 - Request Mode
 - Sample Mode
 - Event Mode

Introduction

❖ Input Modes

■ Request Mode

- The measure of the device is not returned to the program until the device is triggered.
- Example: The user can erase, edit, or correct until enter (return) key (the trigger) is depressed

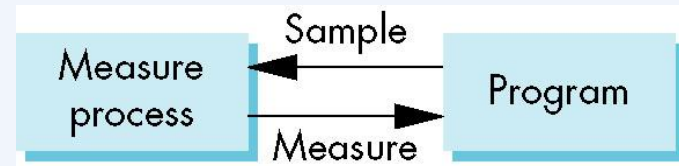


Introduction

❖ Input Modes

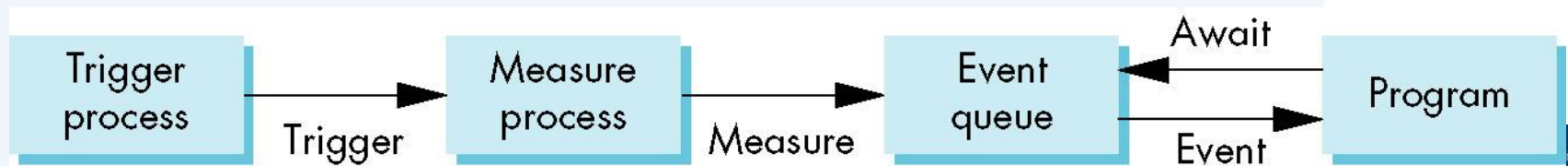
■ Sample Mode

- The measure is returned as soon as the function call is encountered.



■ Event Mode

- Each input device can be triggered at an arbitrary time by a user.
- Each trigger generates an event whose measure is put in an **event queue**.



Event-Driven Input Programming

❖ Introduction

- The event-driven input is developed through the **callback mechanism**
 - **Window:** resize
 - **Mouse:** click one or more buttons
 - **Motion:** move mouse
 - **Keyboard:** press or release a key
 - **Idle:** nonevent

Event-Driven Input Programming

❖ Pointing Device: Mouse Callback

- **Setting:** `glutMouseFunc(mymouse)`

- **Prototype:**

```
void myMouse(GLint button, GLint state,  
             GLint x, GLint y)
```

- **button:** GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON
- **state:** GLUT_UP, GLUT_DOWN
- **x, y:** position in window

Event-Driven Input Programming

❖ Keyboard Events: `glutKeyboardFunc()`

- Trigger Situations
 - The mouse is in the window
 - One of the keys is pressed.
- Measurements
 - ASCII Code of the Key
 - Mouse Location

```
void mykeyboard(unsigned char key,  
                 GLint x, GLint y)
```

Event-Driven Input Programming

❖ Keyboard Events:

- A function `glutGetModifiers()` can be used to get the **meta keys**, such as Control, Alt keys.

```
glutKeyboardFunc(myKeyboard);
```

```
void myKeyboard(unsigned char key, int x, int y){  
    if(key == 'q' || key == 'Q') exit(0);  
}/* End of myKeyboard */
```

Example: Exit the Program

Event-Driven Input Programming

❖ Window Resizing Event: `glutReshapeFunc()`

- It is triggered when the size of window is changed.
- Issues
 - Do we redraw all the objects?
 - What do we do if the **aspect ratio** of the new window is different from the old one?
 - Do we change the sizes of attributes of new primitives?

There is no single answer to any of these questions.

Event-Driven Input Programming

❖ Window Resizing Event: `glutReshapeFunc()`

```
void myReshape(int width, int height){  
    /* set the sub menu */  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    gluOrtho2D(0.0, width, 0.0, height);  
  
    glMatrixMode(GL_MODELVIEW);  
    glLoadIdentity();  
  
    /* set the main menu */  
    glViewport(0, 0, width, height);  
}/* End of myReshape */
```

Event-Driven Input Programming

❖ Display & Idle Callback

- Display Callback: `glutDisplayFunc()`
 - This callback can be invoked in different way to avoid unnecessary drawings.

```
glutPostRedisplay();
```

- Idle Callback
 - It is invoked when there are no other events.
 - The default of idle callback is `NULL`.
 - `glutIdleFunc(myIdle)`

Event-Driven Input Programming

❖ Callback Enable and Disable

- The callback function can be changed during the **program execution**.
- We can disable a callback function by setting its callback function to **NULL**

```
void myMouse(int button, int state, int x, int y){  
    if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)  
        glutIdleFunc(NULL);  
}/* End of myKeyboard */
```

Event-Driven Input Programming

```
int main(int argc, char** argv){
    /* initialize the interaction */
    glutInit(&argc, argv);

    /* specify the window properties and create window */
    .....
    glutDisplayFunc(myDisplay);
    glutIdleFunc(myIdle);
    glutKeyboardFunc(myKeyboard);
    glutMouseFunc(myMouse);
    glutReshapeFunc(myReshape);
    myInit();

    /* start to run the program */
    glutMainLoop();
}/* End of main */
```

Graphical User Interface: Menu

❖ Menu Description

- GLUT provides **pop-up menus** for users to create sophisticated interactive applications.
- The menus can be developed in a **hierarchical** manner.

❖ Development Steps

- Register a callback function to each menu
- Add the items to the menu
- Link the menu to a particular mouse button.

Graphical User Interface: Me

quit
resize

❖ Example:

```
void myInit(){  
    /* set the sub menu */  
    int sub_menu = glutCreateMenu(mySubMenu);  
    glutAddMenuEntry("Increase Square Size", 2);  
    glutAddMenuEntry("Decrease Square Size", 3);  
  
    /* set the main menu */  
    glutCreateMenu(myMenu);  
    glutAddMenuEntry("Quit", 1);  
    glutAddSubMenu("Resize", sub_menu);  
  
    glutAttachMenu(GLUT_RIGHT_BUTTON);  
}/* End of myInit */
```

Callback Function

Callback Function

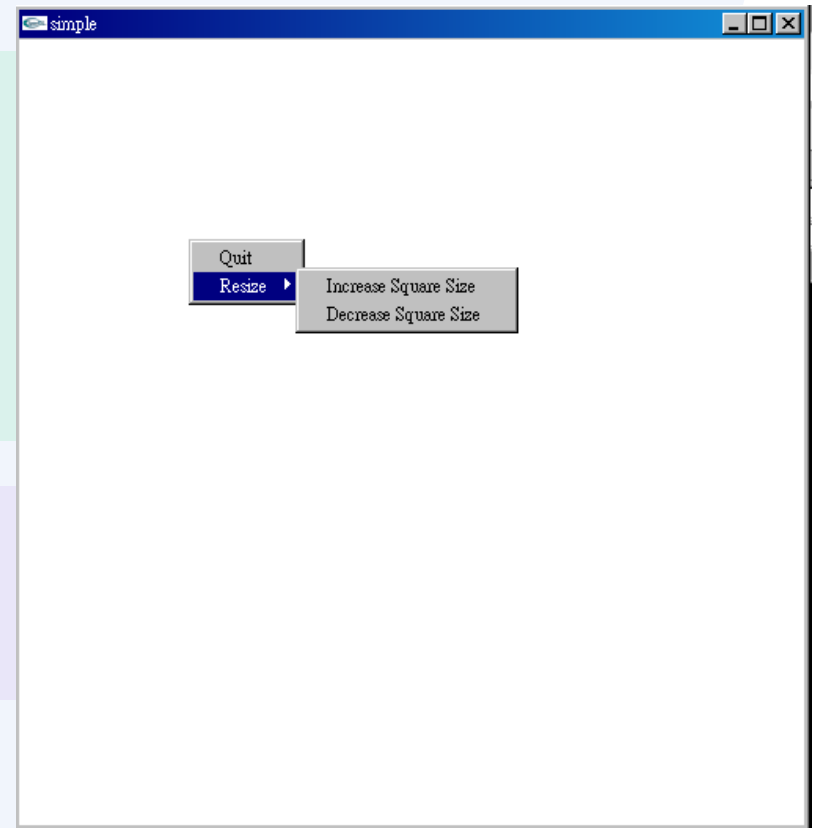
inc
dec

Graphical User Interface: Menu

❖ Example:

```
void myMenu(int id){  
    switch(id){  
        case 1: exit(0);  
    }/* End of switch */  
}/* End of myMenu */
```

```
void mySubMenu(int id){  
  
}/* End of myMenu */
```



Animating Interactive Program

❖ Description

- The function of animation is to create a picture where one or more objects are moving.
- It is an important part of computer graphics.
- Examples
 - An animated character walks across the display
 - We may move the view over time.

Animating Interactive Program

❖ Animation in Movie

- Motion is achieved by taking a sequence of pictures and projecting them at **24 frames/sec.**
 - **Step1:** Each frame is moved to into position behind the lens.
 - **Step2:** The shutter is opened and frame is displayed.
 - **Step3:** The shutter is momentarily closed.
 - **Step4:** Go to Step 1.

The key reason that motion picture projection works is that each frame **is complete** when it is displayed.

Animating Interactive Program

❖ Animation Problem

```
Open_Window();  
for(int i=0; i < 1000000; i++){  
    Clear_Window();  
    Draw_Frame(i);  
    Wait();  
}/* End of for-loop */
```

Problem: it doesn't display completely drawn frames; instead, you watch the drawing as it happens

Animating Interactive Program

❖ Double Buffering

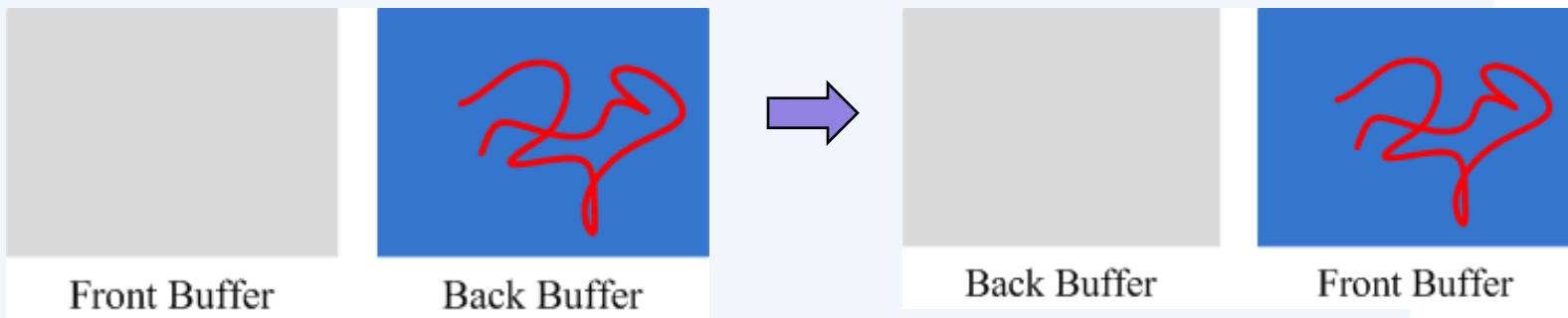
- Hardware/Software supplies two complete color buffers.
 - **Front Buffer**: the one for displaying
 - **Back Buffer**: the one for drawing.
- The objective is to redisplay efficiently so that we can't notice the **clearing** and **redrawing**.
 - One manifestation occurs if the display cannot be drawn in a single refresh cycle.

Animating Interactive Program

❖ Double Buffering

- **Remark:** It does not speed up the drawing process, it only ensures that we never see a partial display.

Motion = Redraw + Swap



Animating Interactive Program

❖ Double Buffering

```
Open_Window(Double_Buffer);  
for(int i=0; i < 1000000; i++){  
    Clear_Window();  
    Draw_Frame(i);  
    Swap_Buffer();  
}/* End of for-loop */
```

OpenGL:

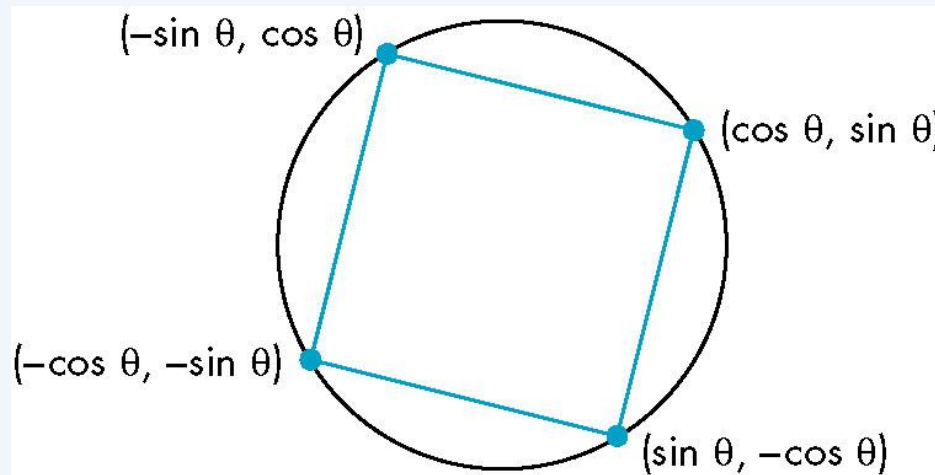
`glutInitDisplayMode (GLUT_RGB | GLUT_DOUBLE)`

`glutSwapBuffers():` swap these two buffers.

Animating Interactive Program

❖ Example: Rotating Square

- Draw a square rotating on the screen
- Rotation Start: press the left button of mouse.
- Rotation Stop: press the right button of mouse.



Animating Interactive Program

❖ Example: Rotating Square

/* callback functions */

```
void myDisplay();           /* display callback */
void myIdle();             /* idle callback */
void myKeyboard(unsigned char, int, int); /* keyboard callback */
void myMouse(int, int, int, int); /* mouse callback */
void myTimer(int);        /* timer callback */
```

/* member function */

```
void myInit();
void mySpin();
```

/* attribute */

```
float theta = 0.0;
```

Animating Interactive Program

```
int _tmain(int argc, char** argv){  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);  
  
    glutInitWindowSize(500, 500);  
    glutInitWindowPosition(0, 0);  
    glutCreateWindow("Double Buffer");  
  
    glutDisplayFunc(myDisplay);  
    glutMouseFunc(myMouse);  
  
    myInit();  
    glutMainLoop();  
    return 0;  
}/* End of main */
```

Animating Interactive Program

```
void myMouse(int button, int state, int x, int y){
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(mySpin);
            break;
        case GLUT_MIDDLE_BUTTON:
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(NULL);
            break;
        default:
            break;
    } /* End of switch */
} /* End of GasketDisplay */
```

Animating Interactive Program

```
void mySpin(void){  
    /* increase the theta */  
    theta += 2;           /* increase 2 degree */  
    if(theta >= 360.0) theta = 0.0;  
  
    glutPostRedisplay();  
}/* End of mySpin */
```

Animating Interactive Program

❖ Example: Rotating Square

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

    float thetar = theta * 3.14159 / (180);
    glBegin(GL_POLYGON);
        glVertex2f(cos(thetar), sin(thetar));
        glVertex2f(sin(thetar), -cos(thetar));
        glVertex2f(-cos(thetar), -sin(thetar));
        glVertex2f(-sin(thetar), cos(thetar));
        glVertex2f(cos(thetar), sin(thetar));
    glEnd();
    glFlush();
    glutSwapBuffers();
}/* End of GasketDisplay */
```

Animating Interactive Program

❖ Timer

- The objective is to avoid the **blur effect** of the display due to many redrawing.
- The timer is used to control the redrawing times per second.
- Timer Callback
 - Execution of this function starts a timer to delay the event loop for “**delay**” milliseconds.
 - When timer has counted down, callback is invoked.

```
glutTimerFunc(int delay,  
              void(*timer_func)(int), int value);
```

Animating Interactive Program

❖ Timer

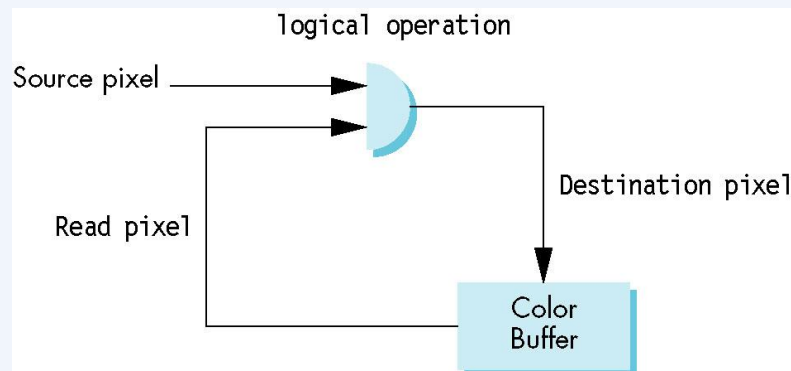
- Because GLUT allows only a single timer, the **recursive mechanism** should be used to execute the callback.

```
void myTimer(int value){  
    glutTimerFunc(1000/value, myTimer, value);  
}/* End of myTimer */
```


Logical Operation

❖ Description

- The logical operations are the ways to combine the source and destination pixels.
 - **Source Pixel:** the pixel that we want to write
 - **Destination Pixel:** the pixel in the drawing buffer that the source pixel will affect.



Logical Operation

❖ Description

- There are 16 possible functions between two bits.
- Each function defines a **writing mode**.

❖ Writing Mode

- Replacement/Copy Mode
 - It is the **default mode** of OpenGL
 - It replaces the destination pixel with source one.
- XOR Mode
 - It combines two bits through exclusive operation

Logical Operation

❖ Mode Change Operators

```
glEnable(GL_COLOR_LOGIC_OP);
```

```
glLogicOp(GL_XOR);
```

```
glEnable(GL_COLOR_LOGIC_OP);
```

```
glLogicOp(GL_COPY)
```

```
glBegin(GL_POLYGON);
```

```
glVertex2f(cos(thetar), sin(thetar));
```

```
glVertex2f(sin(thetar), -cos(thetar));
```

```
glVertex2f(-cos(thetar), -sin(thetar));
```

```
glEnd();
```

```
glLogicOp(GL_XOR);
```

Logical Operation

❖ XOR Mode

- We return to the original state, if we apply xor operation twice.
- It is generally used for **erasing** objects by simply drawing it a second time.

$$d' = d \oplus s$$

$$d = (d \oplus s) \oplus s$$



Thank You !

Geometric Objects and Transformations

Chapter 4

- Introduction:
 - We are now ready to concentrate on three-dimensional graphics
 - Much of this chapter is concerned with matters such as
 - how to represent basic geometric types
 - how to convert between various representations
 - and what statements we can make about geometric objects, independent of a particular representation.

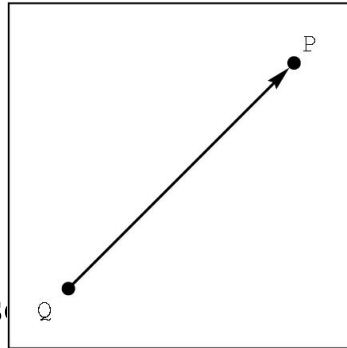
1. Scalars, Points, and Vectors

- We need three basic types to define most geometric objects
 - scalars
 - points
 - and vectors
- We can define each in many ways, so we will look at different ways and compare and contrast them.

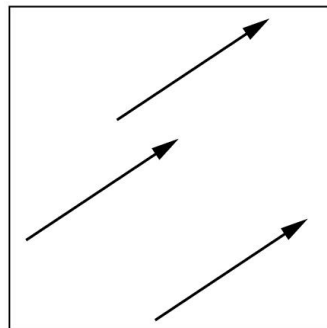
- 1.1 The Geometric View

- Our fundamental geometric object is a point.
 - In a three-dimensional geometric system, a point is a location in space
 - The only attribute a point possesses is its location in space.
- Our scalars are always real numbers.
 - Scalars lack geometric properties,
 - but we need scalars as units of measurement.

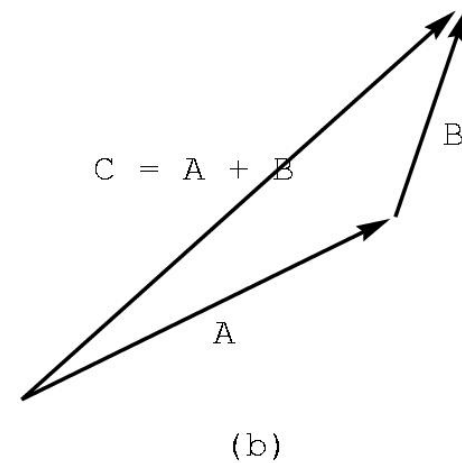
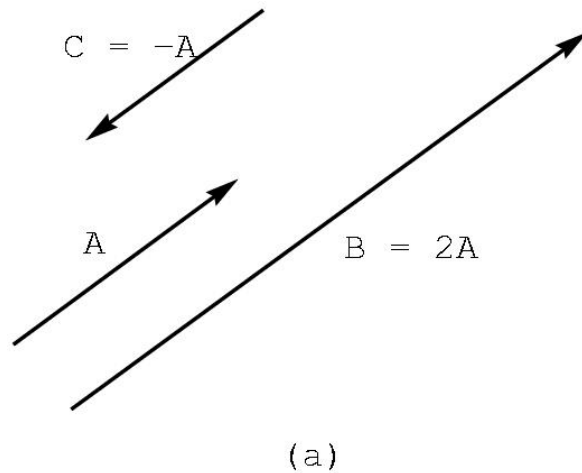
- In computer graphics, we often connect points with directed line segments



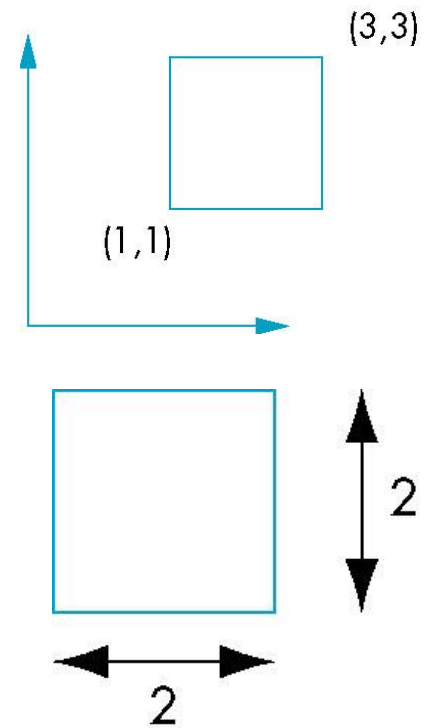
- These are our vectors.
- A vector does not have a fixed position
 - hence these segments are identical because their orientation and magnitude are identical.



- Directed line segments can have their lengths and directions changed by real numbers or by combining vectors



- 1.2 Coordinate-Free Geometry
 - Points exist in space regardless of any reference or coordinate system.
 - Here we see a coordinate system defined by two axis and a simple geometric primitive. We can refer to points by their coordinates
 - If we remove the axes, we can no longer specify where the points are, except for in a relative term



- 1.3 The Mathematical View: Vector and Affine Spaces
 - We can regard scalars, points and vectors as members of mathematical sets;
 - Then look at a variety of abstract spaces for representing and manipulating these sets of objects.
 - The formal definitions of interest to us -- vector spaces, affine spaces, and Euclidean spaces -- are given in Appendix B

- Perhaps the most important mathematical space is the vector space
 - A vector space contains two distinct entities: vectors and scalars.
 - There are rules for combining scalars through two operations: addition and multiplication, to form a scalar field
 - Examples of scalar are:
 - real numbers, complex numbers, and rational functions
 - You can combine scalars and vectors to form a new vector through
 - scalar-vector multiplication and vector-vector addition

- An affine space is an extension of the vector space that includes an additional type of object:
 - The point
- A Euclidean space is an extension that adds a measure of size or distance
- In these spaces, objects can be defined independently of any particular representation
 - But representation provides a tie between abstract objects and their implementation
 - And conversion between representations leads us to geometric transformations

- 1.4 The Computer Science View

- We prefer to see these objects as ADTs

- Def: ADT

- So, first we define our objects

- Then we look to certain abstract mathematical spaces to help us with the operations among them

- 1.5 Geometric ADTs

- Our next step is to show how we can use our types to perform geometrical operations and to form new objects.

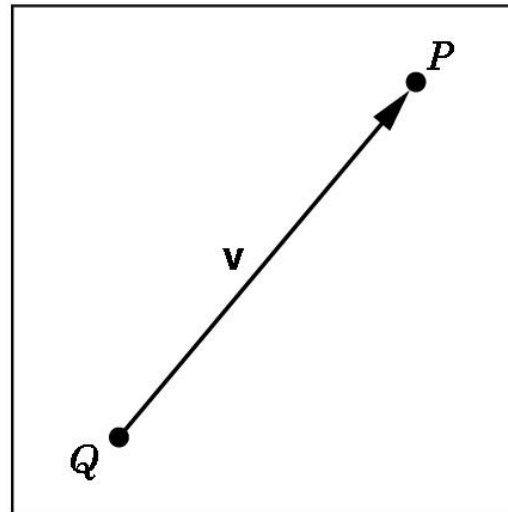
- Notation

- Scalars will be denoted α, β, γ
- Points will be denoted P, Q, R, \dots
- Vectors will be denoted u, v, w, \dots

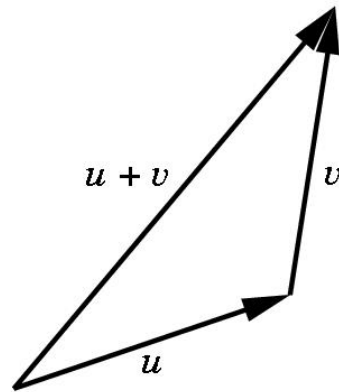
- The magnitude of a vector v is the real number that is denoted by $|v|$

- The operation of vector-scalar multiplication has the property that $|\alpha v| = |\alpha| |v|$

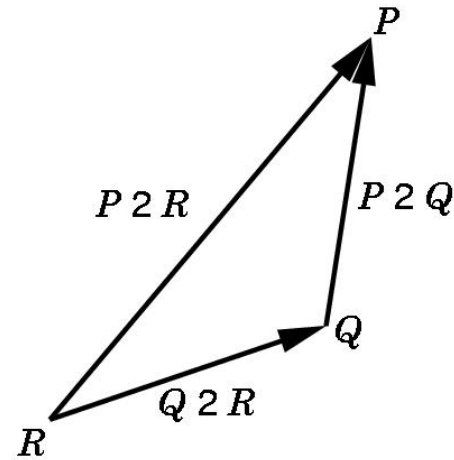
- The direction of αv is the same as the direction of v if α is positive.
- We have two equivalent operations that relate points and vectors:
 - First there is the subtraction of two points P and Q . This is an operation that yields a vector
 - $v = P - Q$
 - or $P = v + Q$



- Second there is the head-to-tail rule that gives us a convenient way of visualizing vector-vector addition.



(a)



(b)

- 1.6 Lines

- The sum of a point and a vector leads to the notion of a line in an affine space

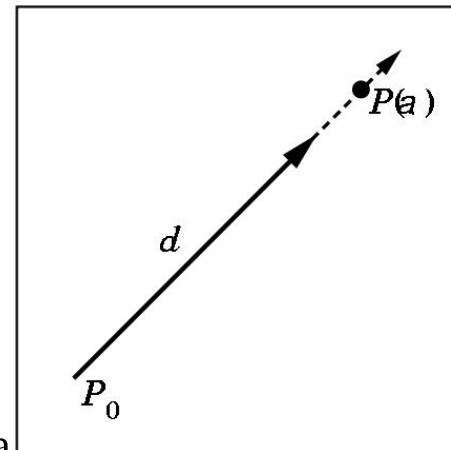
- Consider all points of the form

- $P(\alpha) = P_0 + \alpha d$

- P_0 is an arbitrary point

- d is an arbitrary vector

- α is a scalar



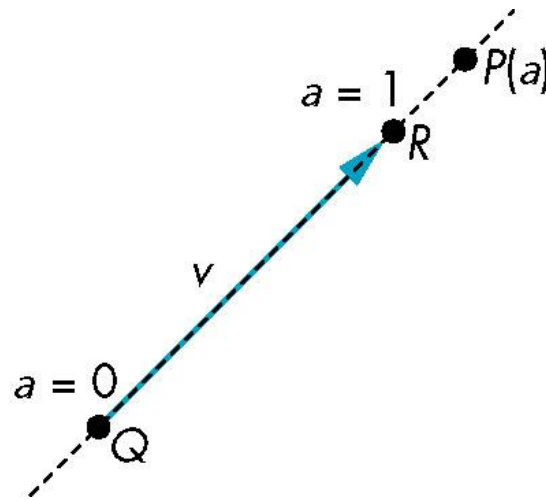
- This form is sometimes called the parametric form, because we generate points by varying the parameter α .

- 1.7 Affine Sums
 - In an Affine Space
 - the following are defined:
 - the addition of two vectors,
 - the multiplication of a vector by a scalar,
 - and the addition of a vector and a point
 - The following are not:
 - the addition of two arbitrary points
 - and the multiplication of a point by a scalar.
 - There is an operation called Affine Addition that is.

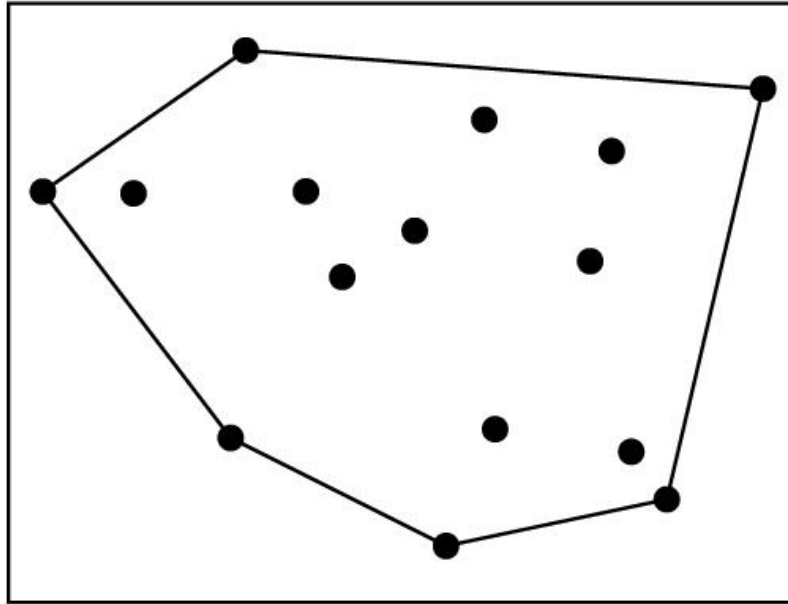
- Affine Addition:

- For any point Q , vector v , and positive scalar α

- $P = Q + \alpha v$ describes all the points on the line Q in the direction of v as shown



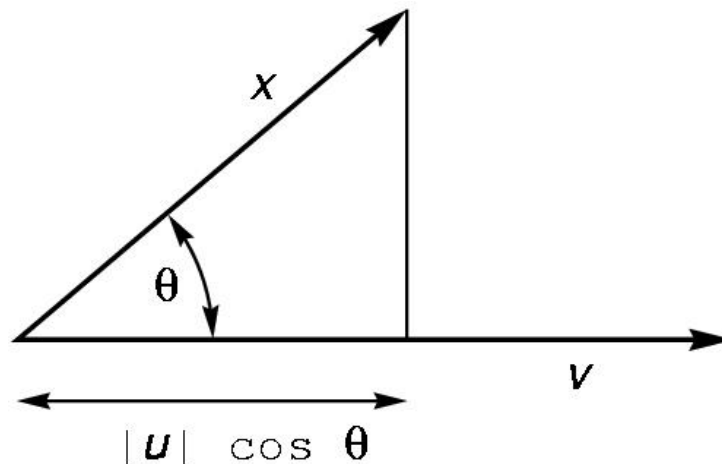
- 1.8 Convexity
 - Def: Convex object
 - Def: Convex Hull



- 1.9 Dot and Cross Products

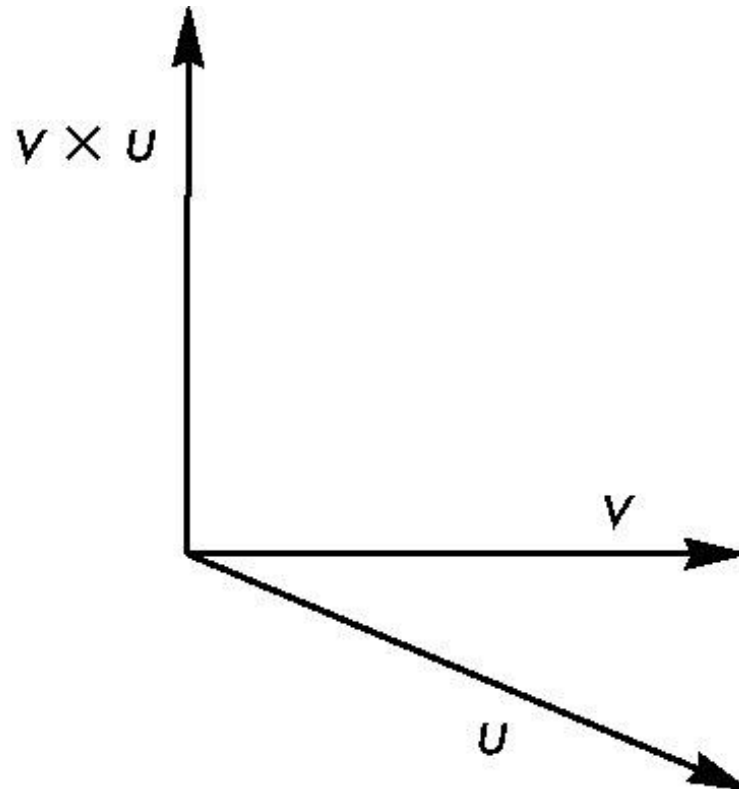
- Dot product

- the dot product of u and v is written $u \bullet v$
- If $u \bullet v = 0$, then u and v are orthogonal
- In euclidean space, $|u|^2 = u \bullet u$
- The angle between two vectors is given by
 - $\cos \theta = (u \bullet v) / (|u| |v|)$

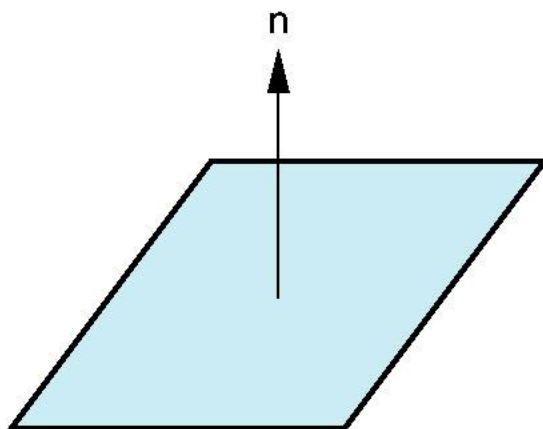
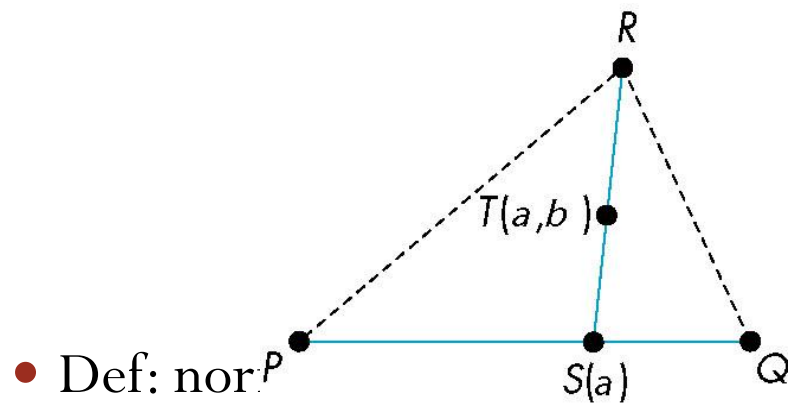


- Cross product

- We can use two non parallel vectors u and v to determine a third vector n that is orthogonal to them $n = u \times v$

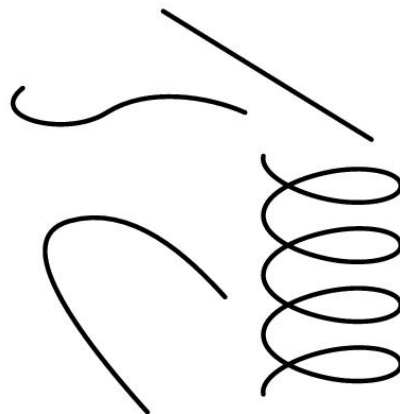


- 1.10 Planes
 - Def: Plane



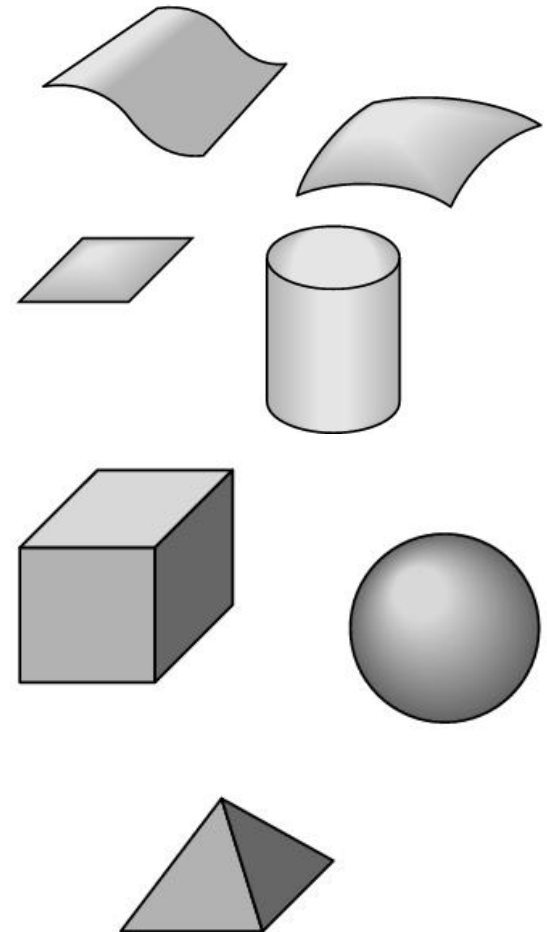
2. Three-Dimensional Primitives

- In a three-dimensional world, we can have a far greater variety of geometric objects than we could in two-dimensions.
 - In 2D we had curves,
 - Now we can have curves in space



- In 2D we had objects with interiors, such as polygons,
- Now we have surfaces in space

- In addition, now we can have objects that have volume



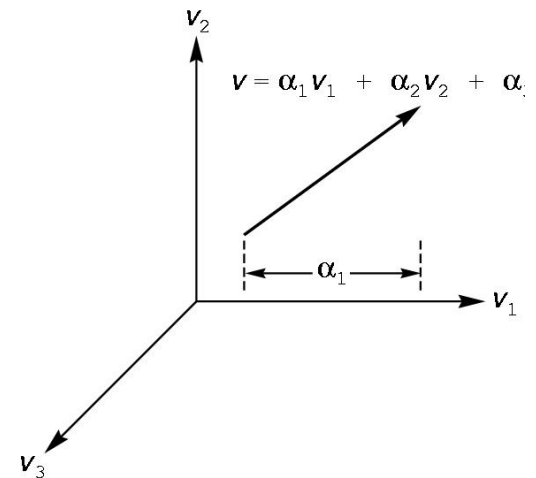
- We face two issues when moving from 2D to 3D.
 - First, the mathematical definitions of these objects becomes complex
 - Second, we are interested in only those objects that lead to efficient implementations in graphic systems
- Three features characterize 3D objects that fit well with existing graphics hardware and software:
 - 1. The objects are described by their surfaces and can be thought of as hollow.
 - 2. The objects can be specified through a set of vertices in 3D
 - 3. The objects either are composed of or can be approximated by flat convex polygons.

- We can understand why we set these conditions if we consider what most modern graphics systems do best:
 - They render triangles

- Def: tessellate

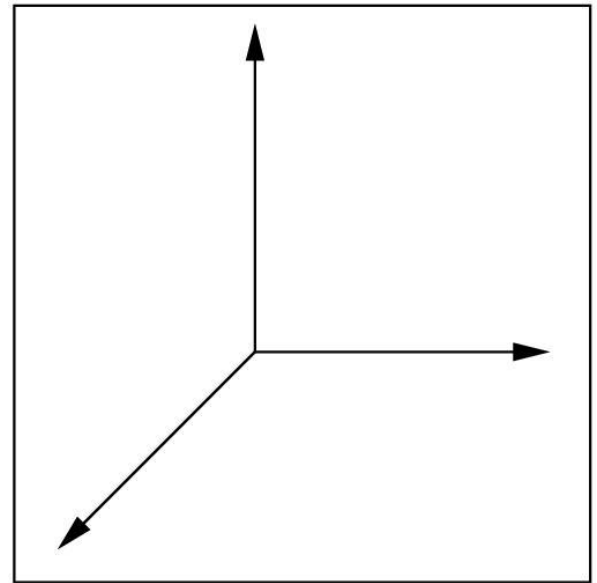
3. Coordinate Systems and Frames

- In a three-dimensional vector space, we can represent any vector w uniquely in terms of any three linearly independent vectors v_1 , v_2 , and v_3 , as
 - $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
- The scalars α_1 , α_2 , and α_3 are the components of w with respect to the basis functions

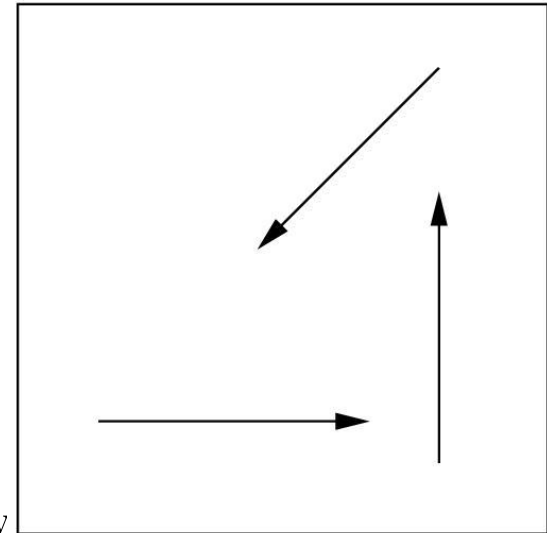


- We can write the representation of w with respect to this basis as the column matrix

- We usually think of basis vectors defining a coordinate system



- However, vectors have no position, so this would also be appropriate
 - But a little more confusing.
- Once we fix a reference point (the origin), we will feel more comfortable,
 - because the usual convention for drawing the coordinate axes as emerging from the origin



- This representation that requires both the reference point and the basis vectors is called a frame.
 - Loosely, this extension fixes the origin of the vector coordinate system at some point P_0 .
 - So, every vector is defined in terms of the basis vectors
 - and every point is defined in terms of the origin and the basis vectors.
 - Thus the representation of either just requires three scalars, so we can use matrix representations

• 3.1 Representations and N-tuples

- Suppose the vectors e_1 , e_2 , and e_3 form a basis.
- The representation of any vector, v , is given by the components $(\alpha_1, \alpha_2, \alpha_3)$ where
 - $v = \alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3$

• 3.2 Changes in Coordinate Systems

- Frequently, we are required to find how the representation of a vector changes when we change the basis vectors.
 - Suppose $\{v_1, v_2, v_3\}$ and $\{u_1, u_2, u_3\}$ are two bases
 - Each basis vector in the second can be represented in terms of the first basis (and vice versa)
 - Hence:
 - $u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$
 - $u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$
 - $u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$

- The 3x3 matrix is

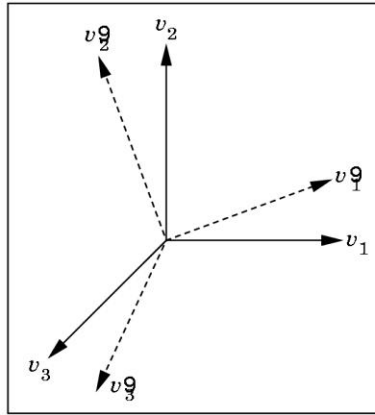
$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

- is defined by the scalars

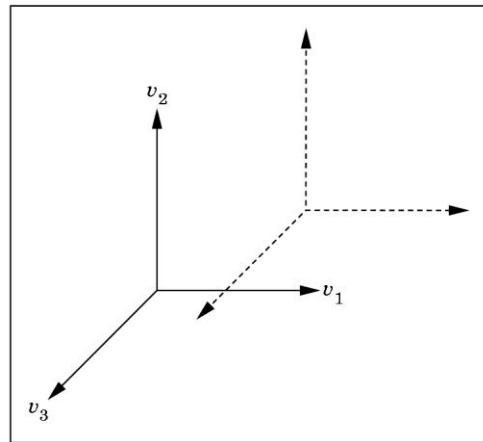
$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

- and M tells us how to go one way, and the inverse of M tells us how to go the other way.

- This change in basis leave the origin unchanged



- However, a simple translation, or change of frame, cannot be represented in this way



- 3.3 Example of Change of Representation

- Suppose we have some vector w

- whose representation in some basis is $a=[1,2,3]$

- We can denote 3 basis vectors $v_1, v_2,$ and v_3 .

- Hence $w = v_1 + 2v_2 + 3v_3$

- Now suppose we make a new basis from the three vectors $v_1, v_2,$ and v_3

- $u_1 = v_1, u_2 = v_1+v_2, u_3 = v_1+v_2+v_3$

- Then the matrix M is

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

- The matrix which converts a representation in $v_1, v_2,$ and v_3 to one with $u_1, u_2,$ and u_3 is

- $A = (M^T)^{-1}$

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

- In the new system $b = Aw$ $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$

- That is $w = -u_1 - u_2 + 3u_3$

$$b = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

- 3.4 Homogeneous Coordinates
 - In order to distinguish between points and vectors...and because matrix multiplication in 3D cannot represent a change in frames, we introduce homogenous coordinates. (4D data)
 - Homogenous coordinates allow us to tie a basis point (origin) with the basis vectors.

- If (v_1, v_2, v_3, P_0) and (u_1, u_2, u_3, Q_0) are two frames, then we can represent the second in terms of the first as

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$

$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$

- These equations can be written as:

$$Q_0 = \gamma_{41} v_1 + \gamma_{42} v_2 + \gamma_{43} v_3 + P_0$$

- Where M is now:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

- We can also use M to compute the changes in the representations directly.
 - Suppose a and b are homogenous-coordinate representations of either two points or two vectors in the two frames then
 - $a = M^T b$
- There are other advantages of using homogenous coordinate systems. Perhaps the most important is the fact that all affine transformations can be represented as matrix multiplications in homogenous coordinates.

- 3.5 Example of Change in Frames

- 3.6 Working with Representations

- 3.7 Frames and ADTs
 - Thus far, our discussion has been mathematical.
 - Now we begin to address the question of what this has to do with programming

• 3.8 Frames in OpenGL

- In OpenGL we use two frames:
 - The camera frame
 - The world frame.
- We can regard the camera frame as fixed
 - (or the world frame if we wish)
 - The model-view matrix positions the world frame relative to the camera frame.
- Thus, the model-view matrix converts the homogeneous-coordinate representations of points and vectors to their representations in the camera frame.

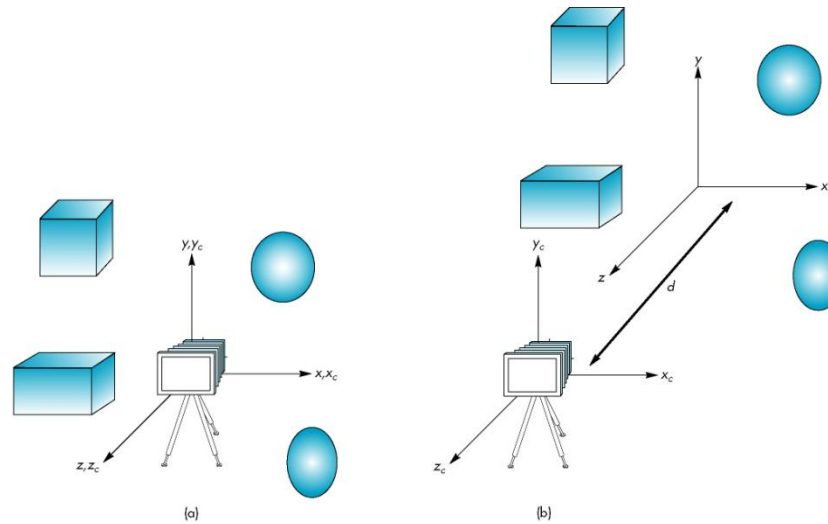
- Because the model-view matrix is part of the state of the system there is always a camera frame and a present-world frame.
- OpenGL provides matrix stacks, so we can store model-view matrices
 - (or equivalently, frames)

- As we saw in Chapter 2, the camera is at the origin of its frame
 - The three basis vectors correspond to:
 - The up direction of the camera (y)
 - The direction the camera is pointing (-z)
 - and a third orthogonal direction
 - We obtain the other frames (in which to place objects) by performing homogeneous coordinate transformations
 - We will learn how to do this in Section 4.5
 - In Section 5.2 we use them to position the camera relative to our objects.

- Let's look at a simple example:
 - In the default settings, the camera and the world frame coincide with the camera pointing in the negative z direction
 - In many applications it is natural to define objects near the origin.
- If we regard the camera frame as fixed, then the model-view matrix:

- moves a point (x,y,z) in the world frame to the point $(x,y,z-d)$ in the camera frame.
- $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

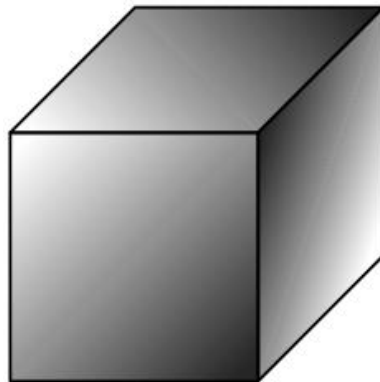
- Thus by making d a suitably large positive number, we move the objects in front of the camera



- Note that, as far as the user - who is working in world coordinates - is concerned, they are positioning objects as before
- The model-view matrix takes care of the relative positioning of the frames

4. Modeling a Colored Cube

- We now have the tools we need to build a 3D graphical application.
- Consider the problem of drawing a rotating cube on the screen

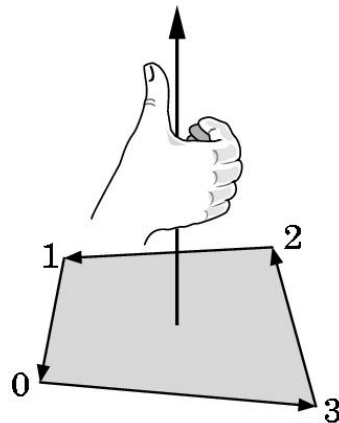


- 4.1 Modeling of a Cube

- `typedef GLfloat point3[3];`
- `point3 vertices[8] = { {-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} };`
- `glBegin(GL_POLYGON);`
- `glVertex3fv(vertices[0]);`
- `glVertex3fv(vertices[3]);`
- `glVertex3fv(vertices[2]);`
- `glVertex3fv(vertices[1]);`
- `glEnd();`

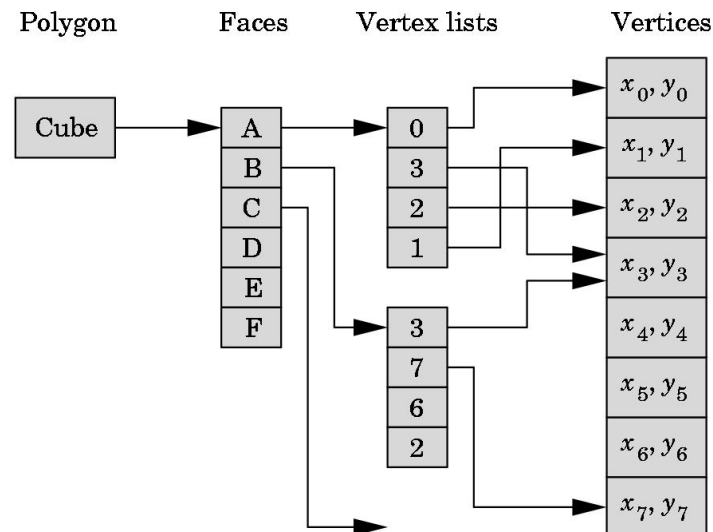
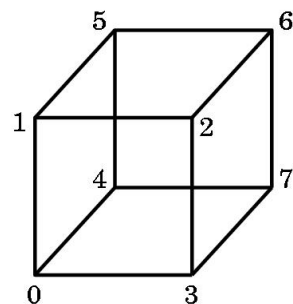
• 4.2 Inward- and Outward-Pointing Faces

- We have to be careful about the order in which we specify our vertices when we are defining a three-dimensional polygon
- We call a face **outward facing** if the vertices are traversed in a counterclockwise order when the face is viewed from the outside.
 - This is also known as the right-hand rule



4.3 Data Structures for Object Representation

- We could describe our cube
 - glBegin(GL_POLYGON)
 - six times, each followed by four vertices
 - glBegin(GL_QUADS)
 - followed by 24 vertices
- But these repeat data....
- Let's separate the topology from the geometry.



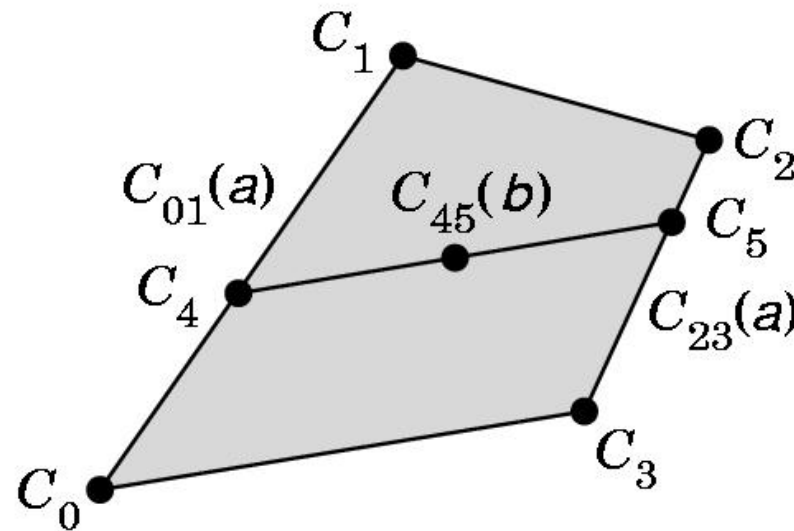
• 4.4 The Color Cube

```
typedef GLfloat point3[3];
point3 vertices[8] = { {-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[8][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}};

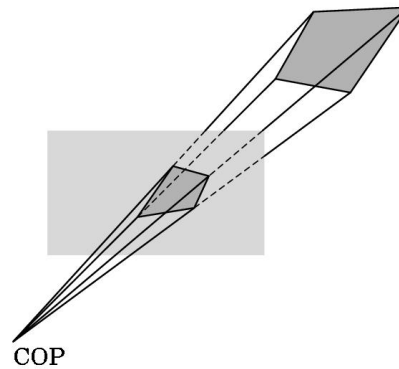
void quad(int a, int b, int c, int d)
{
    glBegin(GL_POLYGON);
        glColor3fv(colors[a]); glVertex3fv(vertices[a]);
        glColor3fv(colors[a]); glVertex3fv(vertices[b]);
        glColor3fv(colors[a]); glVertex3fv(vertices[c]);
        glColor3fv(colors[a]); glVertex3fv(vertices[d]);
    glEnd();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


• 4.5 Bilinear Interpolation

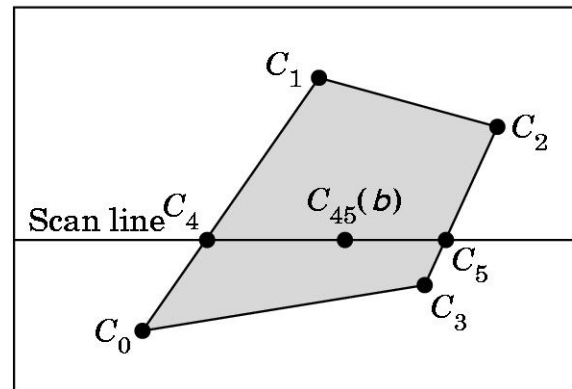
- Although we have specified colors for the vertices of the cube, the graphics system must decide how to use this information to assign colors to points inside the polygon.
- Probably the most common method is bilinear interpolation.



- Another method is Scan-line interpolation
 - pushes the decision off until rasterization.
 - OpenGL uses this method for this as well as other values.
 - First you project the polygon



- Then convert the colors with each scan line



• 4.6 Vertex Arrays

- Vertex arrays provide a method for encapsulating the information in our data structure such that we could draw polyhedral objects with only a few function calls.
 - Rather than the 60 OpenGL calls:
 - six faces, each of which needs a glBegin, a glEnd, four calls to glColor, and four calls to glVertex.
- There are 3 steps in using vertex arrays
 - First, enable the functionality of vertex arrays.
 - Second, we tell OpenGL where and in what format the arrays are.
 - Third, we render the object

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);
```

```
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};
```

- We now have a few options regarding how to draw the arrays

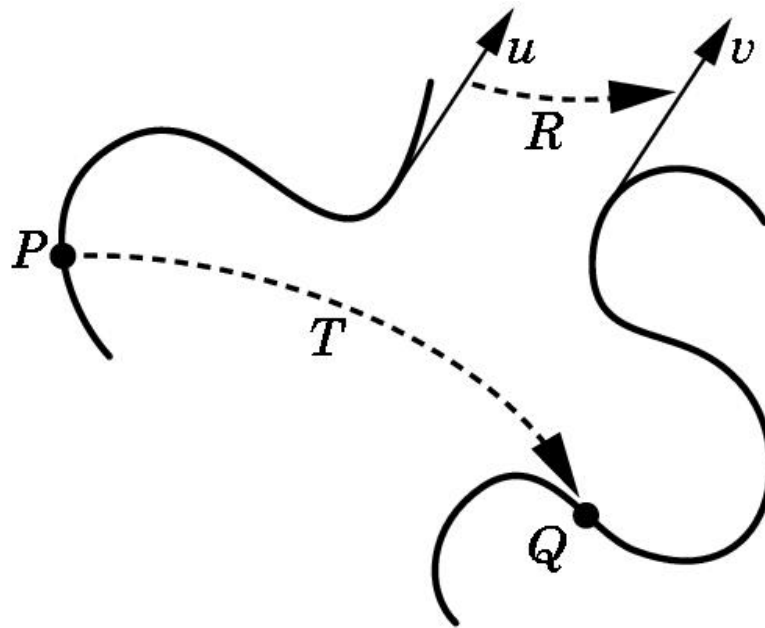
```
for(i=0;i<6;i++)
```

```
    glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE,
        &cubeIndices[4*i]);
```

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

5. Affine Transformations

- A transformation is a function that takes a point (or vector) and maps it into another .



- We can represent this as:
 - $Q=T(P)$ and $v=R(u)$
- If we write both the points and vectors in homogeneous coordinates, then we can represent both vectors and points in 4-D column matrices and define a single function
 - $q=f(p)$ and $v=f(u)$
- This is too general, but if we restrict it to linear functions, then we can always write it as
 - $v = Au$
 - where v and u are column vectors (or points) and A is a square matrix.

- Therefore,
 - we need only to transform the homogeneous coordinate representation of the endpoints of a line segment to determine completely a transformed line.
 - Thus, we can implement our graphics systems as a pipeline that passes endpoints through affine transformation units, and finally generate the line at rasterization stage.
- Fortunately, most of the transformations that we need in computer graphics are affine.
 - These transformations include rotation, translation, and scaling.
 - With slight modifications, we can also use these results to describe the standard and parallel projections.

6. Translation, Rotation, and Scaling

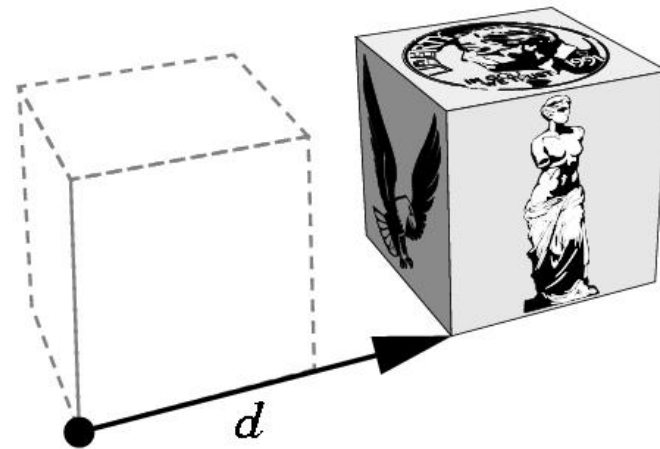
- In this section,
 - First, we show how we can describe the most important affine transformations independently of any representation.
 - Then we find matrices that describe these transformations
- In section 4.8 we shall see how these transformations are implemented in OpenGL

- 6.1 Translation

- Translation is an operation that displaces points by a fixed distance in a given direction



(a)



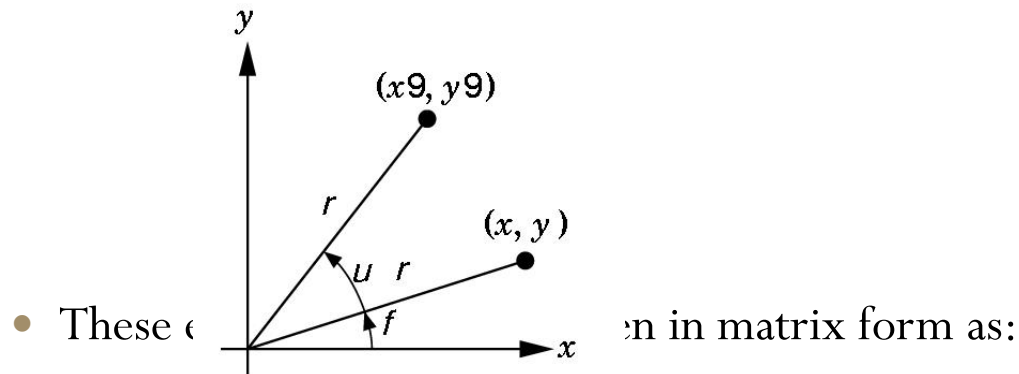
(b)

- To
ve

ment

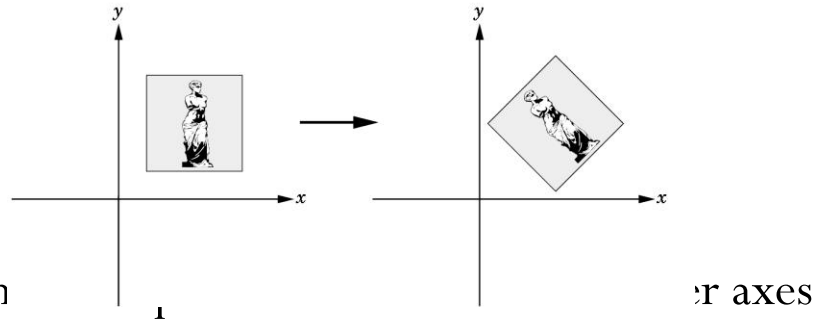
- 6.2 Rotation

- Rotation is more difficult to specify than translation, because more parameters are involved
- Let's start with 2D rotation about the origin

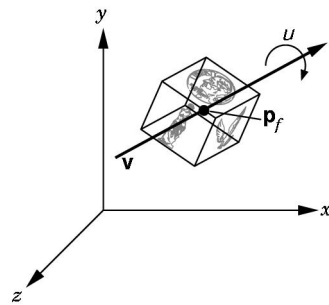


$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- We expand this to 3D in Section 4.7
- Note that there are three features of this transformation that extend to other rotations.
 - 1) There is one point - the origin - that is unchanged by the rotation. This is called a fixed point.



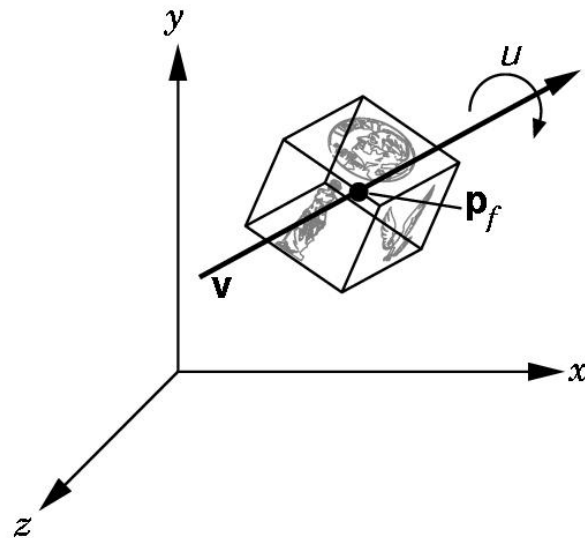
- 2) We can



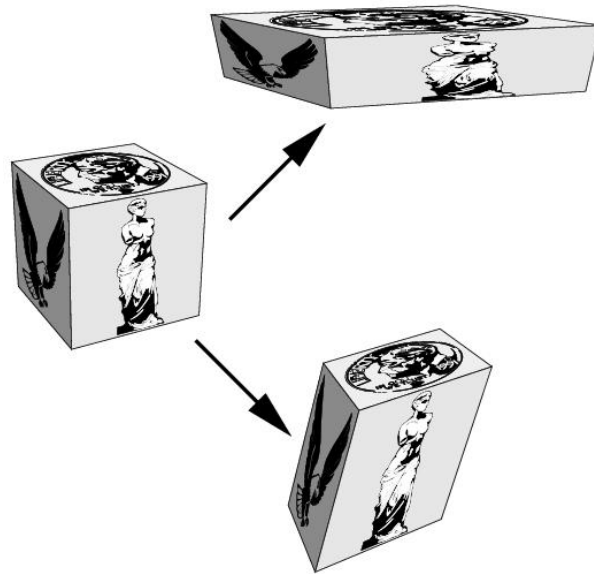
- 3) 2D rot
axis.

equivalent to 3D rotation about the x

- We can use these observations to define a general 3D rotation that is independent of the frame.
 - To do this we must specify:
 - a fixed point: P_f
 - a rotation angle: θ
 - a line or vector about which to rotate:

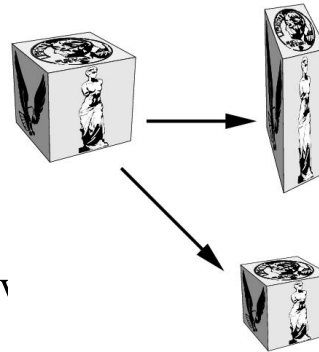


- Rotations and translation are known as rigid-body transformations.
 - No combination can alter the shape of an object,
 - They can alter only the object's location and orientation
- The transformation given in this figure are affine, but they are not rigid-body transformations

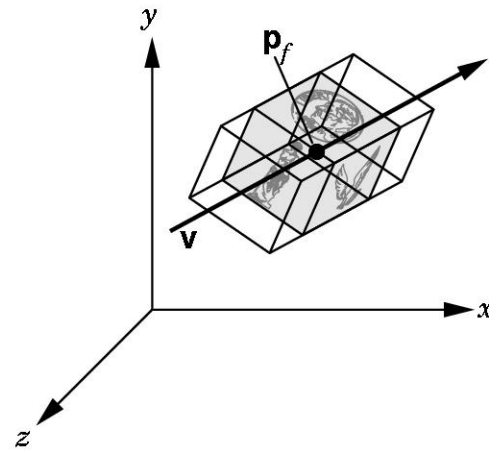


- 6.3 Scaling

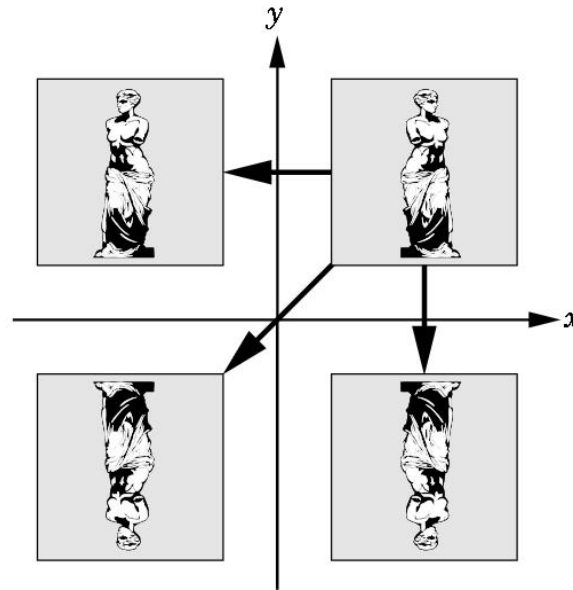
- Scaling is an affine non-rigid body transformation.



- Scaling transformations have



- Hence, to specify a scaling,
 - we can specify a fixed point,
 - a direction in which we wish to scale,
 - and a scale factor α .
 - For $\alpha > 1$, the object gets longer
 - for $0 \leq \alpha < 1$ the object get smaller
 - Negative values of α give us reflection



7. Transformations in Homogeneous Coordinates

- In most graphics APIs we work with a representation in homogeneous coordinates.
- And each affine transformation is represented by a 4x4 matrix of the form:

$$M = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• 7.1 Translation

- Translation displaces points to a new position defined by a displacement vector.
 - If we move p to p' by displacing by a distance d then
 - $p' = p + d$
 - or $p' = Tp$
 - where

$$T = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- T is called the translation matrix

• 7.2 Scaling

- A scaling matrix with a fixed point at the origin allows for independent scaling along the coordinate axes.
 - $x' = \beta_x x$
 - $y' = \beta_y y$
 - $z' = \beta_z z$
- These three can be combined in homogeneous for as
 - $p' = Sp$

$$S = S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• 7.3 Rotation

- We first look at rotation with a fixed point at the origin.
 - We can find the matrices for rotation about the individual axes directly from the results of the 2D rotation developed earlier.
 - Thus,
 - $x' = x \cos \theta - y \sin \theta$
 - $y' = x \sin \theta + y \cos \theta$
 - $z' = z$
 - of $p' = R_z p$

$$R_z = R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

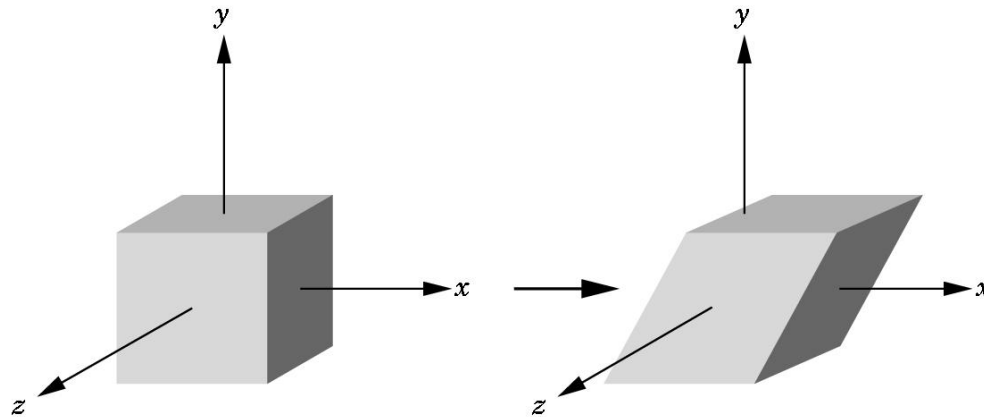
- We can derive the matrices for rotation about the x and y axes through an identical argument.
 - And we can come up with:

$$R_x = R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

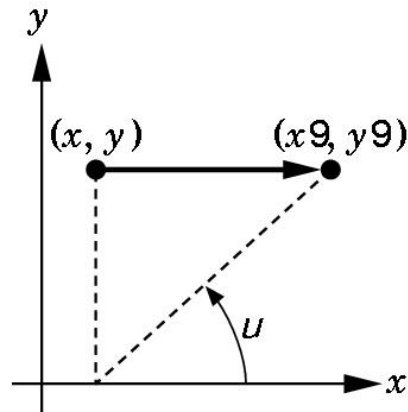
$$R_y = R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• 7.4 Shear

- Although we can construct any affine transformation from a sequence of rotations, translations, and scaling, there is one more affine transformation that is of such importance that we regard it as a basic type, rather than deriving it from others.



- Using simple trigonometry, we can see that each shear is characterized by a single θ



- The equation
 - $x' = x + y \cot \theta, y' = y, z' = z$
- Leading to the shear matrix :

$$H_x(\theta) = \begin{bmatrix} 1 & \cot \theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

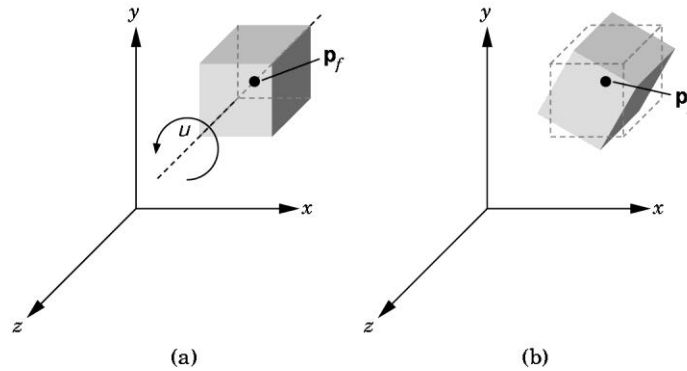
8. Concatenation of Transformations

- In this section, we create examples of affine transformations by multiplying together, or concatenating, sequences of basic transformations that we just introduced.
- This approach fits well with our pipeline architectures for implementing graphics systems.

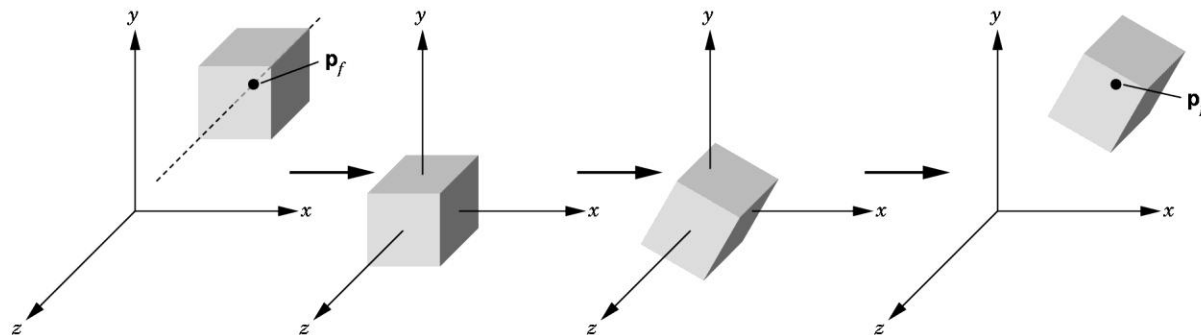


- 8.1 Rotation About a Fixed Point

- In order to do this:

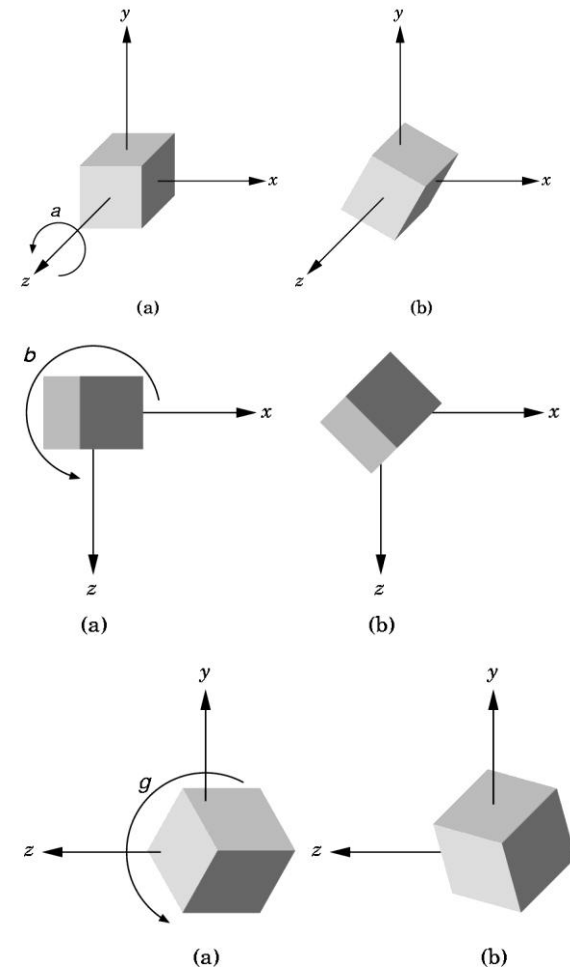


- You do this:



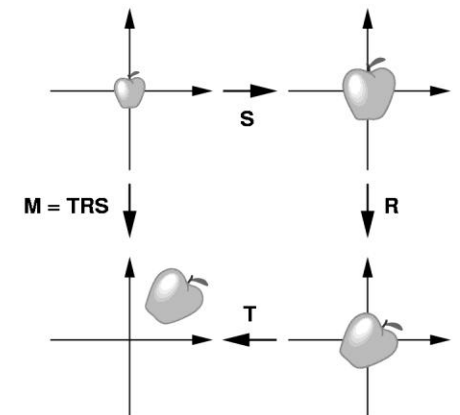
• 8.2 General Rotation

- An arbitrary rotation about the origin can be composed of three successive rotations about the three axes.



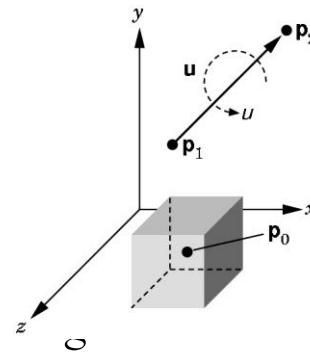
• 8.3 The Instance Transformation

- Objects are usually defined in their own frames, with the origin at the center of mass, and the sides aligned with the axes.
- The instance transformation is applied as follows:
 - First we scale
 - Then we orient it with a rotation matrix
 - Finally we translate it to the desired orientation.

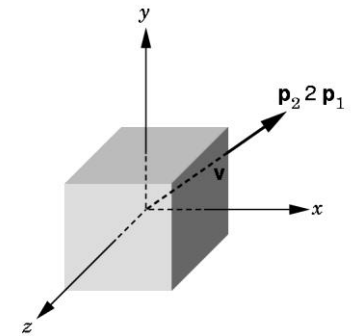


8.4 Rotation About an Arbitrary Axis

- In order to do this:

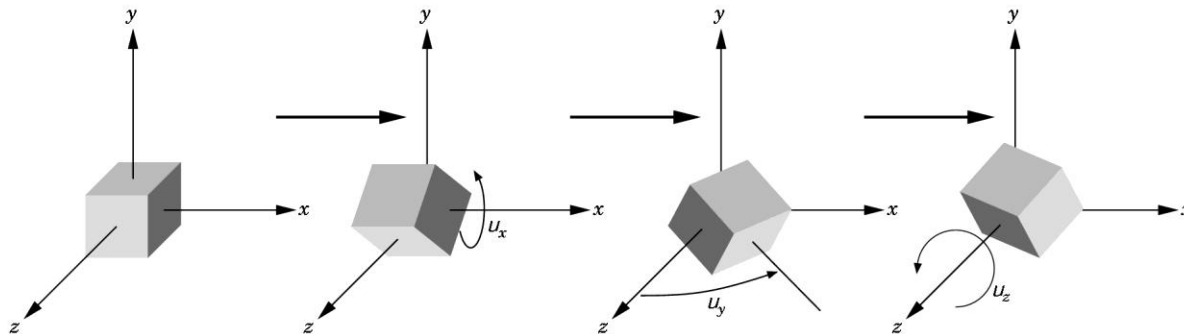


- We move the fixed point to the



- Do the rotations

- Tra



9. OpenGL Transformation Matrices

- In OpenGL there are three matrices that are part of the state.
 - We shall use only the model-view matrix in this chapter.
 - All three can be manipulated by a common set of functions,
 - And we use `glMatrixMode` to select the matrix to which the operations apply.

• 9.1 The Current Transformation Matrix

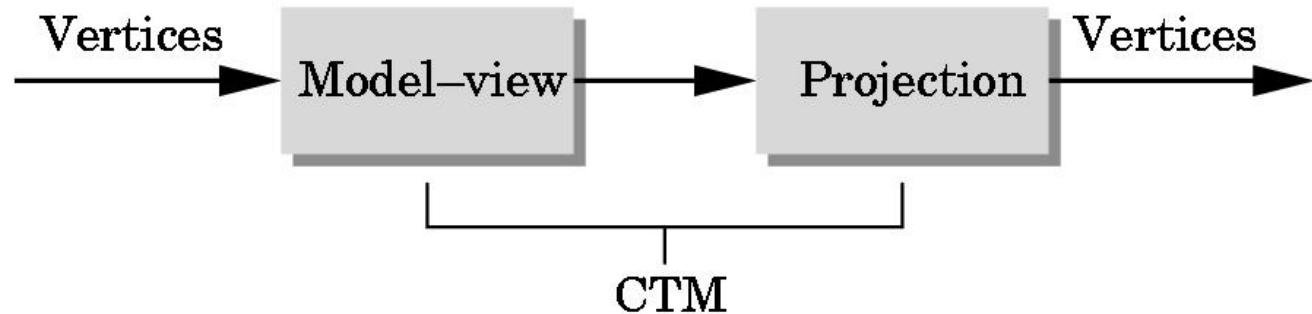
- This is the matrix that is applied to any vertex that is defined subsequent to its setting.
- If we change the CTM we change the state of the system.
- The CTM is part of the pipeline



- The functions that alter the CTM are:
 - Initialization ($C \leftarrow I$)
 - Post-Multiplication ($C \leftarrow CT$)

• 9.2 Rotation, Translation, and Scaling

- In OpenGL,
 - the matrix that is applied to all primitives is the product of the model-view matrix (GL_MODELVIEW) and the projection matrix (GL_PROJECTION)
 - We can think of the CTM as the product of these two.



- We can load a matrix with
 - `glLoadMatrixf(pointer_to_matrix);`
- or set it to the identity with
 - `glLoadIdentity();`
- Rotation, translation, and scaling are provided through three functions:
 - `glRotate(angle, vx, vy, vz);`
 - angle is in degrees
 - vx, vy, and vz are the components of a vector about which we wish to rotate.
 - `glTranslate(dx, dy, dz);`
 - `glScale(sx, sy, sz);`

• 9.3 Rotation About a Fixed Point in OpenGL

- In Section 4.8 we showed that we can perform a rotation about a fixed point other than the origin.
 - (translate, rotate, and translate back)
 - Here is how you do it in OpenGL
 - `glMatrixMode(GL_MODELVIEW);`
 - `glLoadIdentity();`
 - `glTranslatef(4.0, 5.0, 6.0);`
 - `glRotatef(45.0, 1.0, 2.0, 3.0);`
 - `glTranslatef(-4.0, -5.0, -6.0);`

• 9.4 Order of Transformations

- You might be bothered by the apparent reversal of the function calls.
- The rule in OpenGL is this:
 - The transformation specified most recently is the one applied first.
 - $C \leftarrow I$
 - $C \leftarrow CT$
 - $C \leftarrow CR$
 - $C \leftarrow CT$
 - each vertex that is specified after the model-view matrix has been set will be multiplied by C thus forming the new vertex
 - $q = Cp$

- 9.5 Spinning of the Cube

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity( );
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube( );
    glutSwapBuffers( );
}
```

```
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;
}
```

```
void spinCube( )
{
    theta[axis] += 2.0;
    if (theta[axis] > 360.0) theta[axis] -= 360.0;
    glutPostRedisplay( );
}
```

```
void mkey(char key, int mousex, int mousey)
{
    if(key=='q' || key=='Q')
        exit( );
}
```

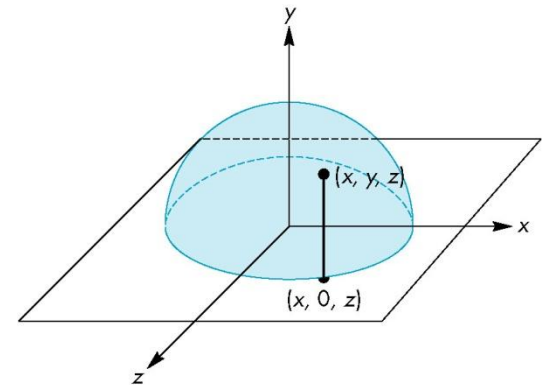
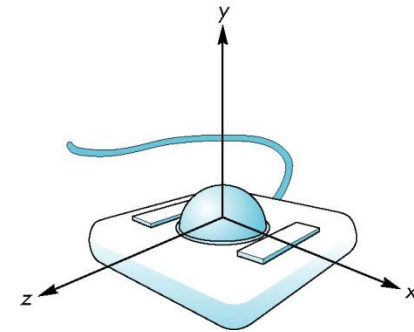
• 9.6 Loading, Pushing, and Popping Matrices

- For most purposes, we can use rotation, translation, and scaling to form a desired transformation matrix.
- In some circumstances, however, such as forming a shear matrix, it is easier to set up the matrix directly.
 - `glLoadMatrixf(myarray);`
 - Note: This is a column major array.
- We can also multiply on the right of the current matrix by using
 - `glMultMatrixf(myarray);`

- Sometimes we want to perform a transformation and then return to the same state as before its execution.
- We can push the current transformation matrix on a stack and recover it later.
- Thus we often see the sequence
 - `glPushMatrix();`
 - `glTranslatef(. . .);`
 - `glRotatef(. . .);`
 - `glScalef(. . .);`
 - `// draw object here`
 - `glPopMatrix();`

10 Interfaces to Three-Dimensional Applications

- This section describes combination of devices (keyboard and mouse) as well as trackballs and virtual trackballs to achieve better input.



11. Quaternions

- Quaternions are an extension of complex numbers that provide an alternative method for describing and manipulating rotations.
- Although less intuitive, they provide advantages for animation and hardware implementations of rotation

12. Summary and Notes

- In this chapter, we have presented two different-but ultimately complementary-points of view regarding the mathematics of Computer Graphics.
 - One is the mathematical abstraction of the objects with which we work in computer graphics is necessary if we are to understand the operations that we carry out in our programs
 - The other is that transformations (and homogeneous coordinates) are the basis for implementations of graphics systems
- Finally we provided the set of affine transformations supported by OpenGL, and discussed ways that we could concatenate them to provide all affine transformations.

13. Suggested Readings

- Homogeneous coordinates arose in Geometry(51) and were later discovered by the graphics community(81)
 - Their use in hardware started with the SGI Geometry Engine (82)
 - Modern hardware architectures use Application Specific Integrated Circuits (ASIC's) that include homogeneous coordinate transformations
- Quaternions were introduced to computer graphics by Shoemaker(85) for use in animation
- Software tools such as Mathematica(91) and Matlab(95) are excellent aids for learning to manipulate transformation matrices

Exercises -- Due next Monday

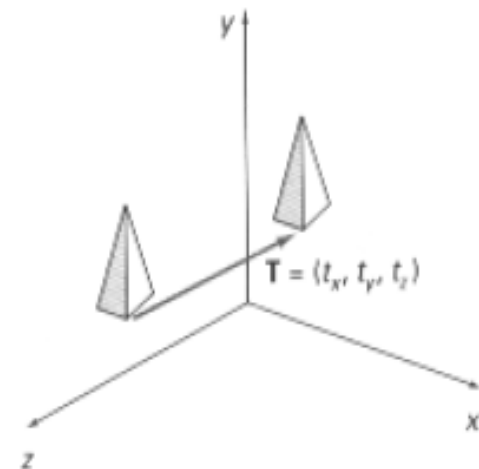
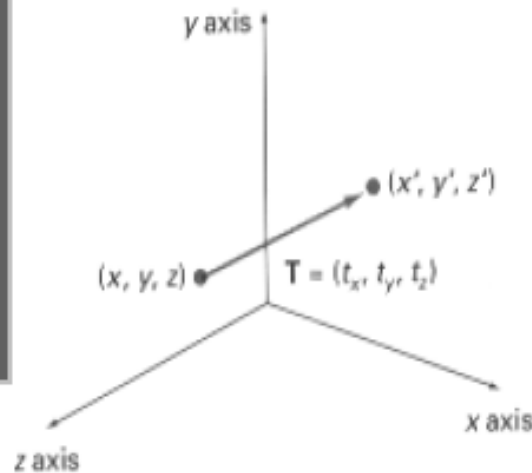
- 4.5
- 4.8
- 4.13
- 4.17

3D Graphics

Translation

In three-dimensional homogeneous coordinate representation, a point is transformed from position $P = (x, y, z)$ to $P' = (x', y', z')$ This can be written as:

$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y \\z' &= z + t_z\end{aligned}$$



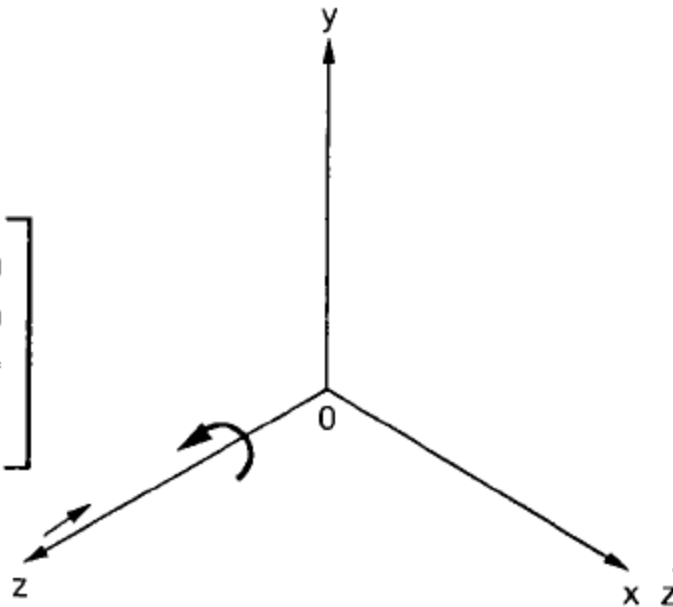
Translation

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

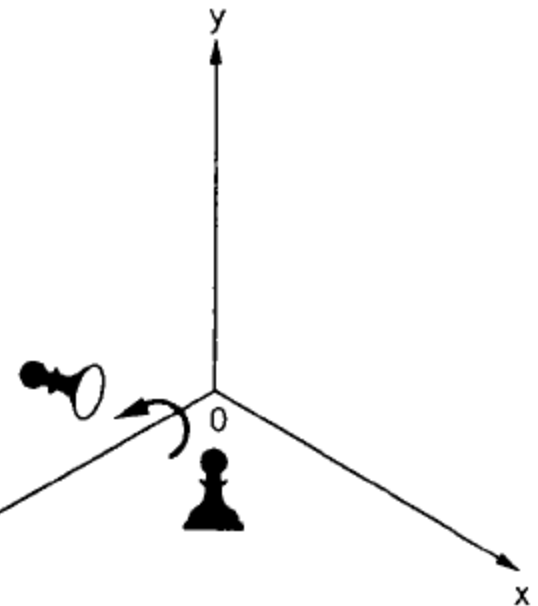
Rotation

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a)



(b)

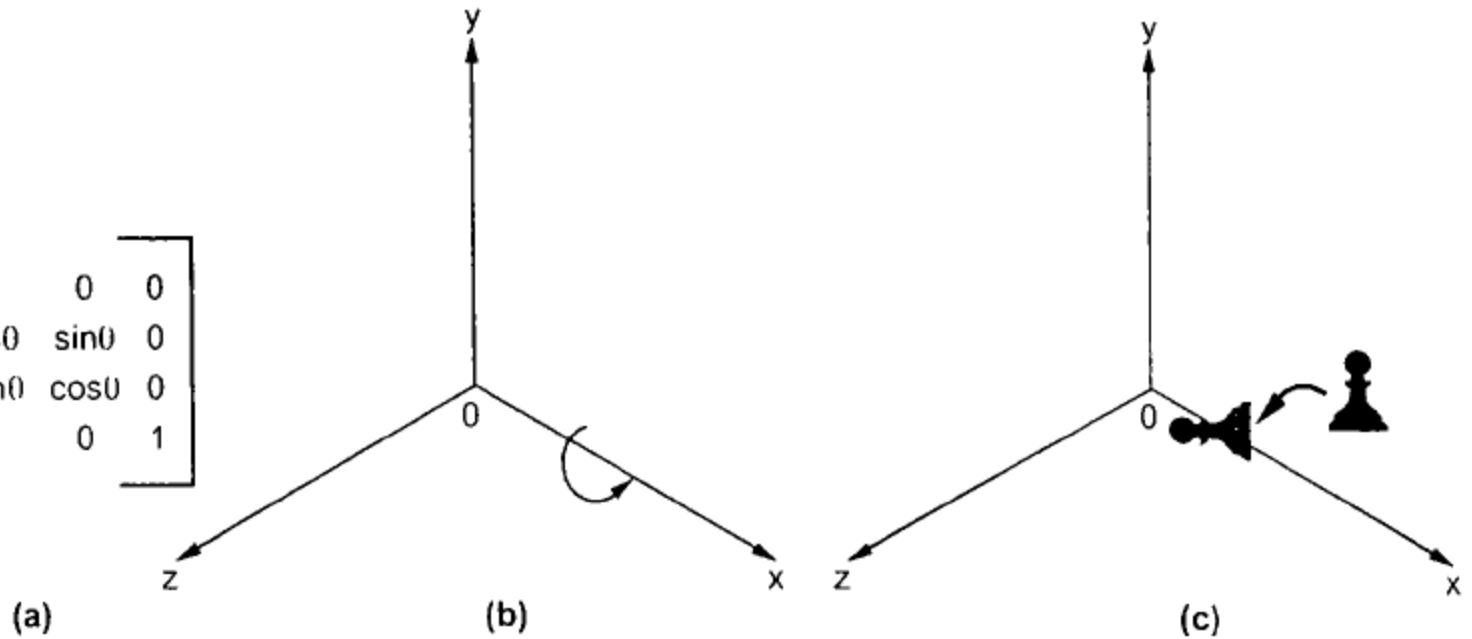


(c)

Rotation about z axis

Rotation

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

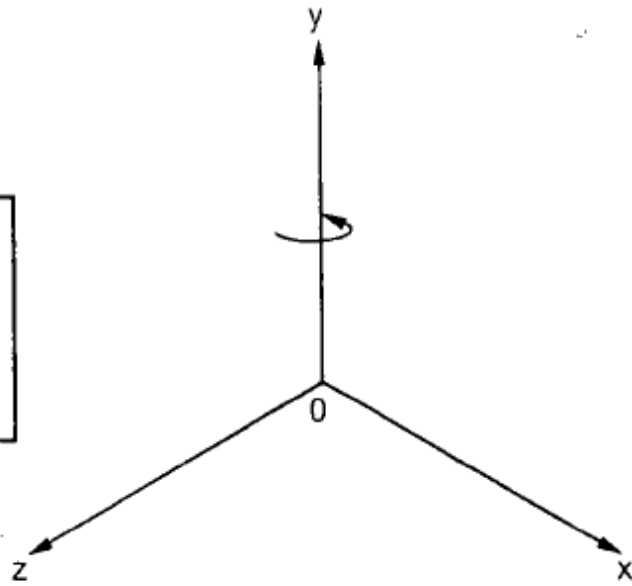


Rotation about x axis

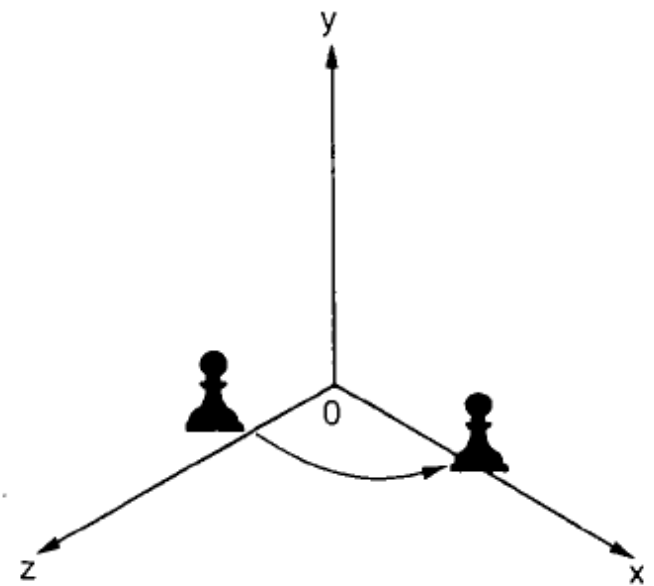
Rotation

$$R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a)



(b)



(c)

Rotation about y axis

Scaling

- The matrix expression for the scaling transformation of a position $P = (x_i, y_i, z_i)$ relative to coordinate origin can be written as:

$$\begin{bmatrix} x_i & y_i & z_i & w \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x x_i & S_y y_i & S_z z_i & w \end{bmatrix}$$

3D Viewing

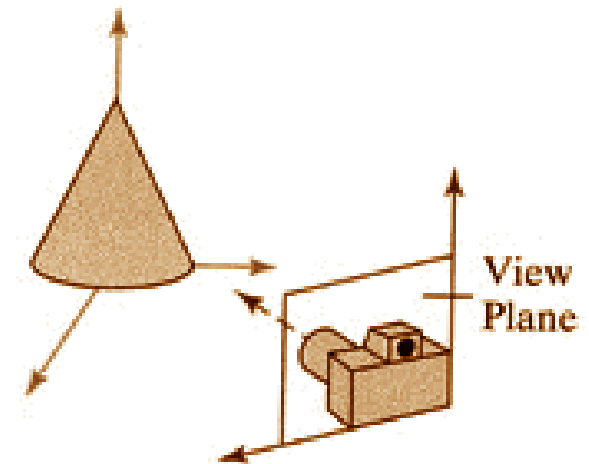
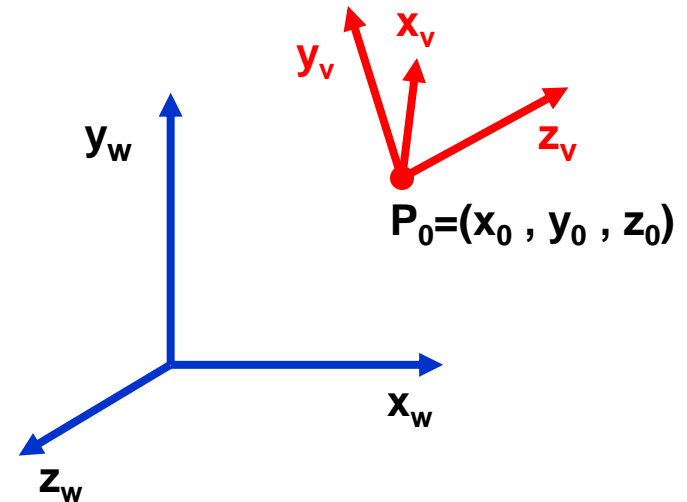
- Viewing in 3D involves the following considerations: -
 - We can view an object from any spatial position, eg. In front of an object, Behind the object, In the middle of a group of objects, Inside an object, etc.
- 3D descriptions of objects must be projected onto the flat viewing surface of the output device.

Viewing Coordinates

- Generating a view of an object in 3D is similar to photographing the object.
- Whatever appears in the viewfinder is projected onto the flat film surface.
- Depending on the position, orientation and aperture size of the camera corresponding views of the scene is obtained.

Specifying The View Coordinates

- For a particular view of a scene first we establish **viewing-coordinate system**.
- A **view-plane** (or **projection plane**) is set up perpendicular to the viewing z-axis.
- World coordinates are transformed to viewing coordinates, then viewing coordinates are projected onto the view plane.

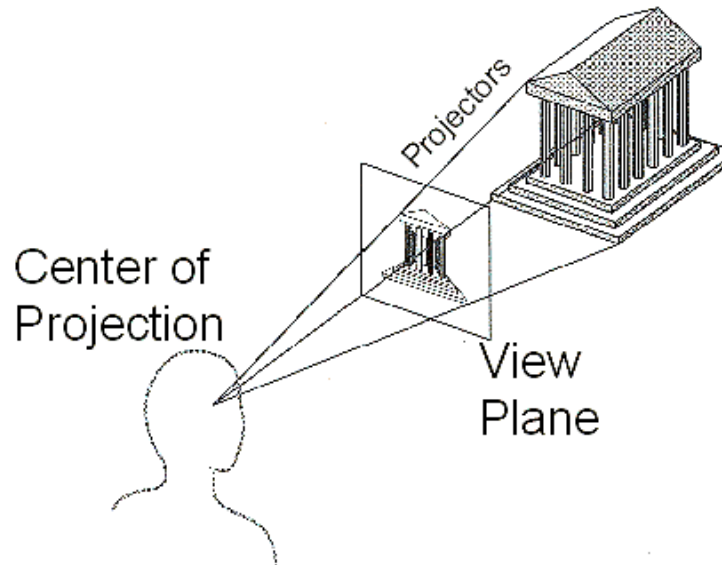


Specifying The View Coordinates

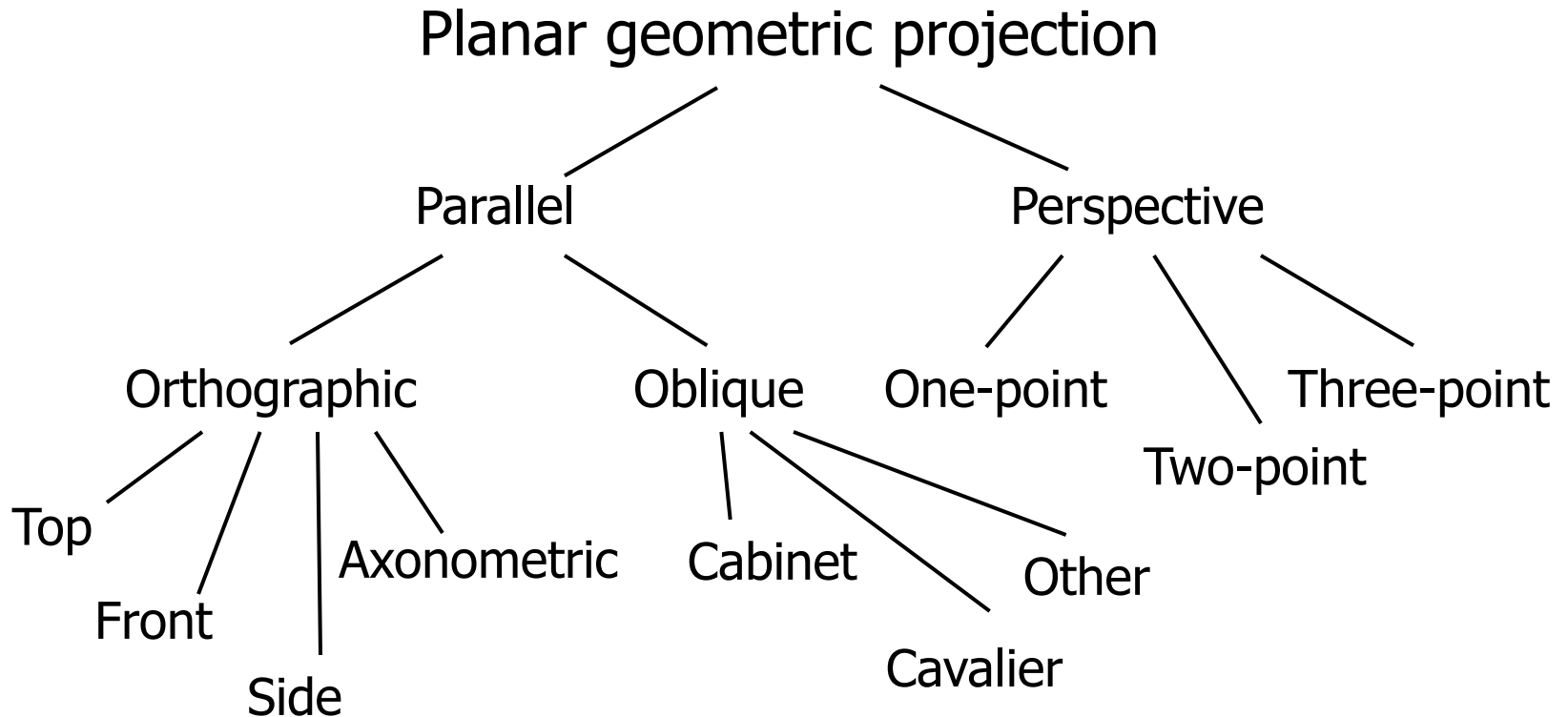
- To establish the viewing reference frame, we first pick a world coordinate position called the **view reference point**.
- This point is the origin of our viewing coordinate system. If we choose a point on an object we can think of this point as the position where we aim a camera to take a picture of the object.

Projection

- General definition
 - Transform points in n -space to m -space ($m < n$)
- In computer graphics
 - Map viewing coordinates to 2D screen coordinates

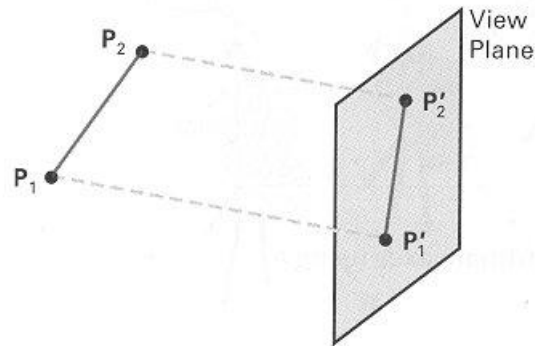


Taxonomy of Projections

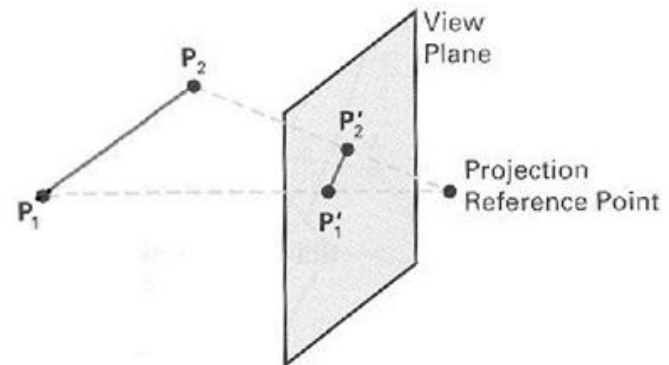


Parallel & Perspective

- Parallel Projection

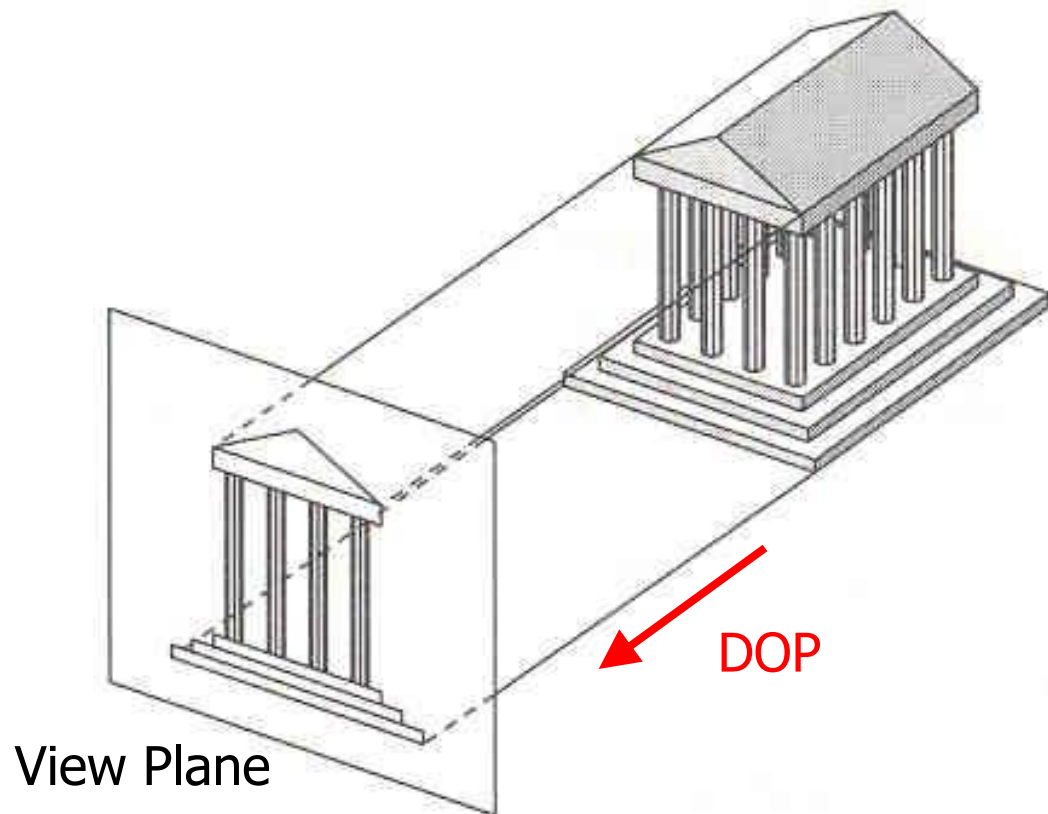


- Perspective Projection



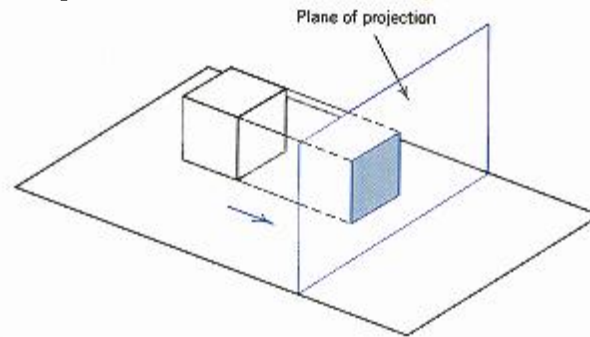
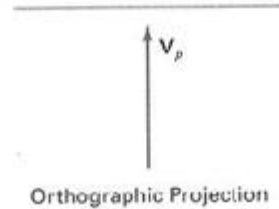
Parallel Projection

- Center of projection is at infinity
 - Direction of projection (DOP) same for all points

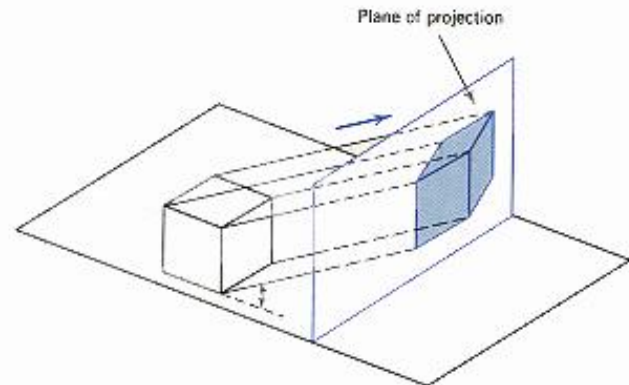
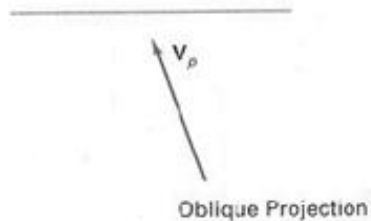


Orthographic & Oblique

- Orthographic parallel projection
 - the projection is perpendicular to the view plane

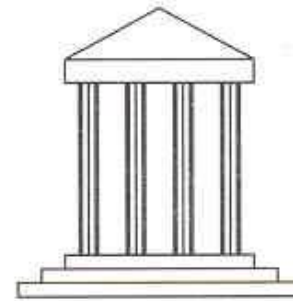
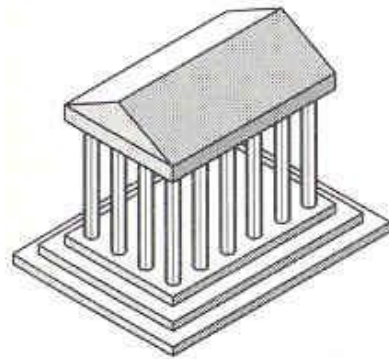


- Oblique parallel projection
 - The projectors are inclined with respect to the view plane

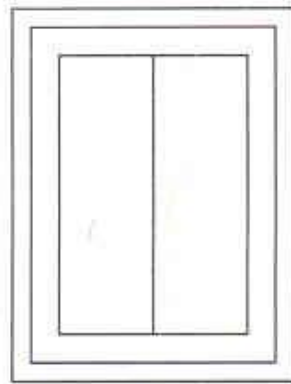


Orthographic Projections

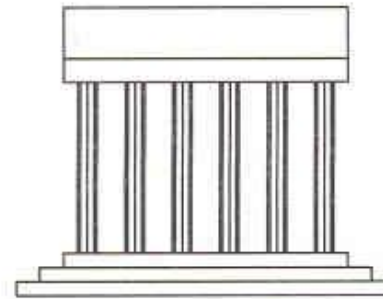
- DOP perpendicular to view plane



Front



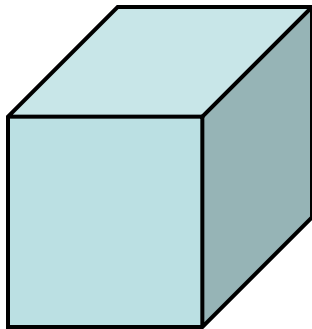
Top



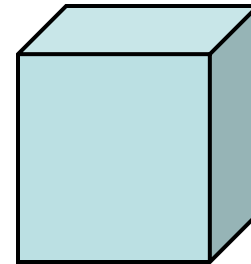
Side

Oblique Projections

- DOP not perpendicular to view plane



Cavalier
(DOP at 45°)



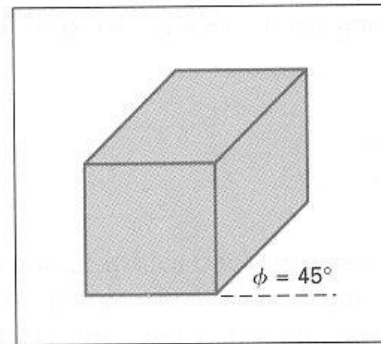
Cabinet
(DOP at 63.4°)

Oblique Projections

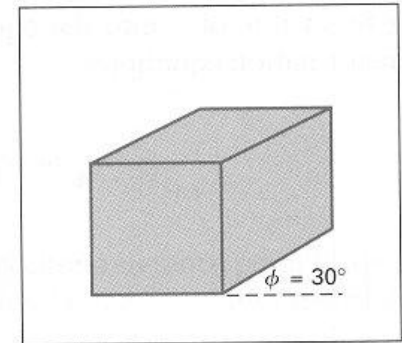
- DOP not perpendicular to view plane

- Cavalier projection

$$\tan \alpha = 1, \quad \alpha = 45^\circ$$



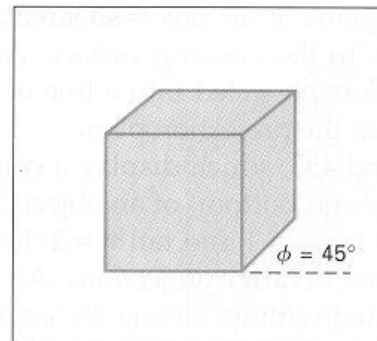
(a)



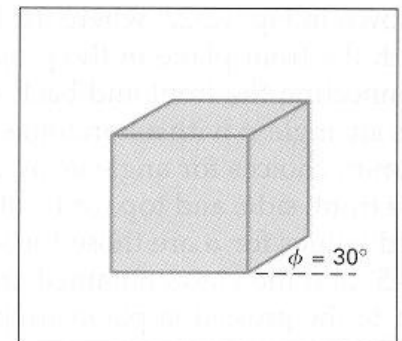
(b)

- Cabinet projection

$$\tan \alpha = 2, \quad \alpha = 63.4^\circ$$



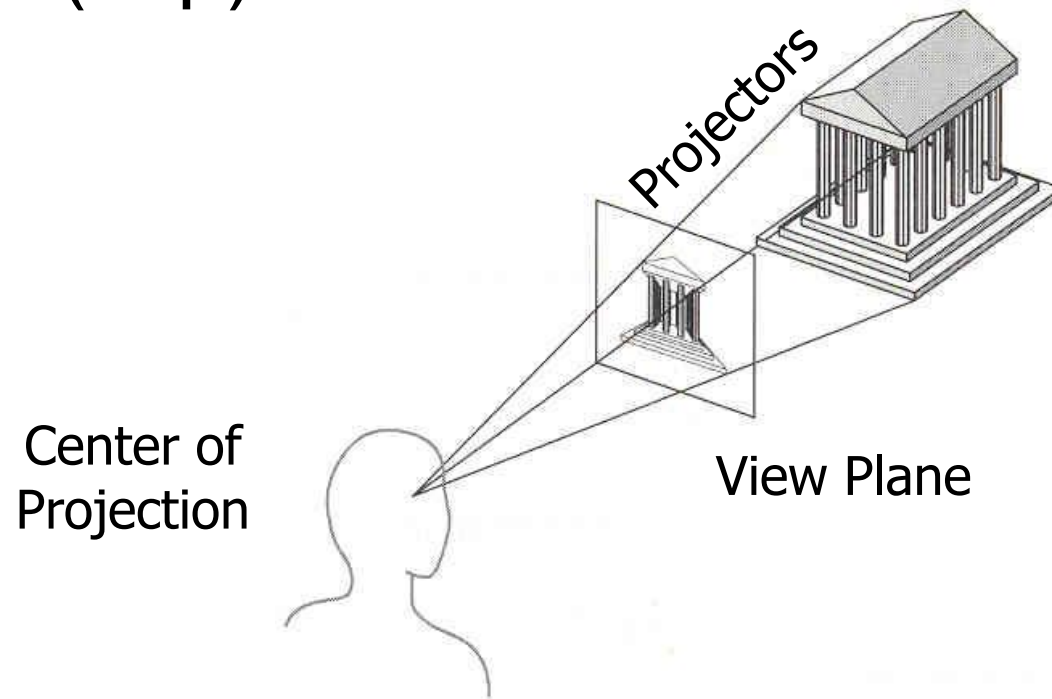
(a)



(b)

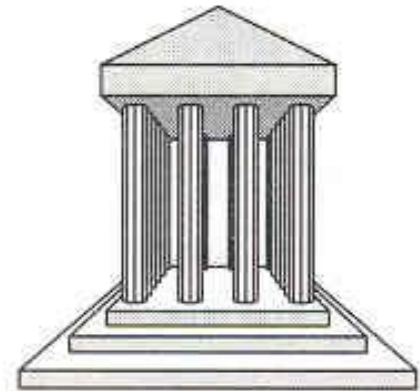
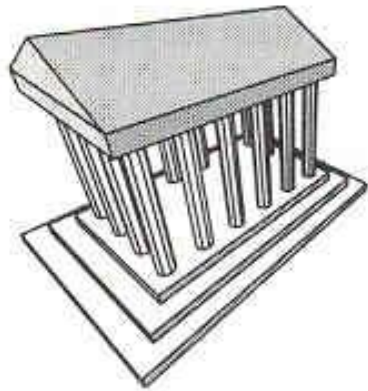
Perspective Projection

- Map points onto “view plane” along “projectors” emanating from “center of projection”(cop)



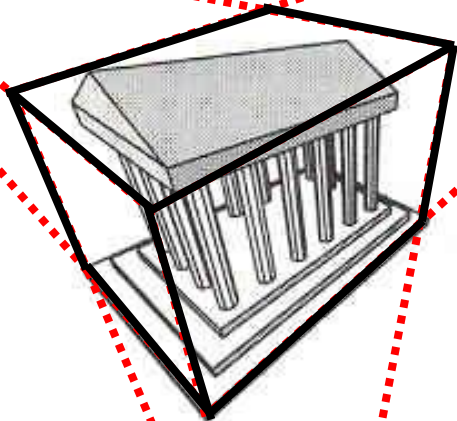
Perspective Projection

- How many vanishing point?



Perspective Projection

- How many vanishing point?

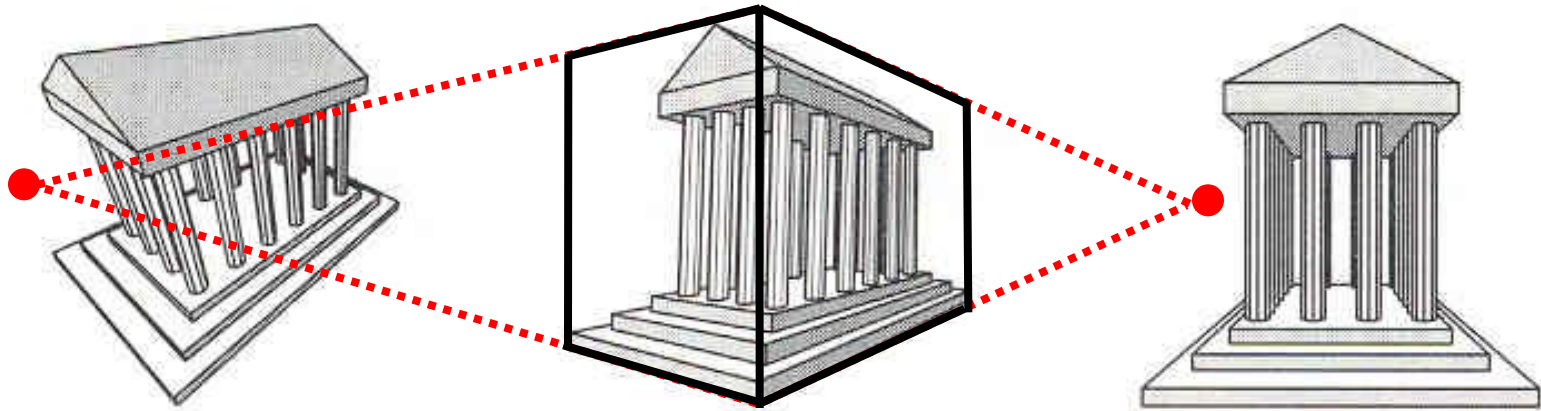


Three-point
perspective



Perspective Projection

- How many vanishing point?

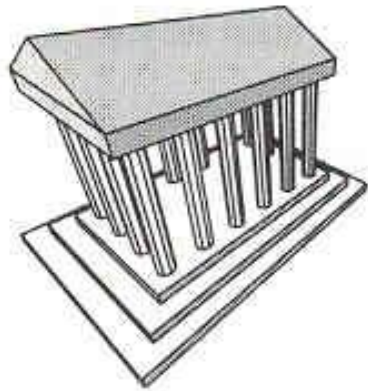


Three-point
perspective

Two-point
perspective

Perspective Projection

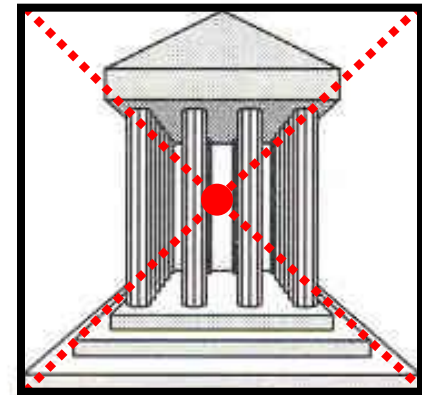
- How many vanishing point?



Three-point perspective



Two-point perspective



One-point perspective

2D Clipping

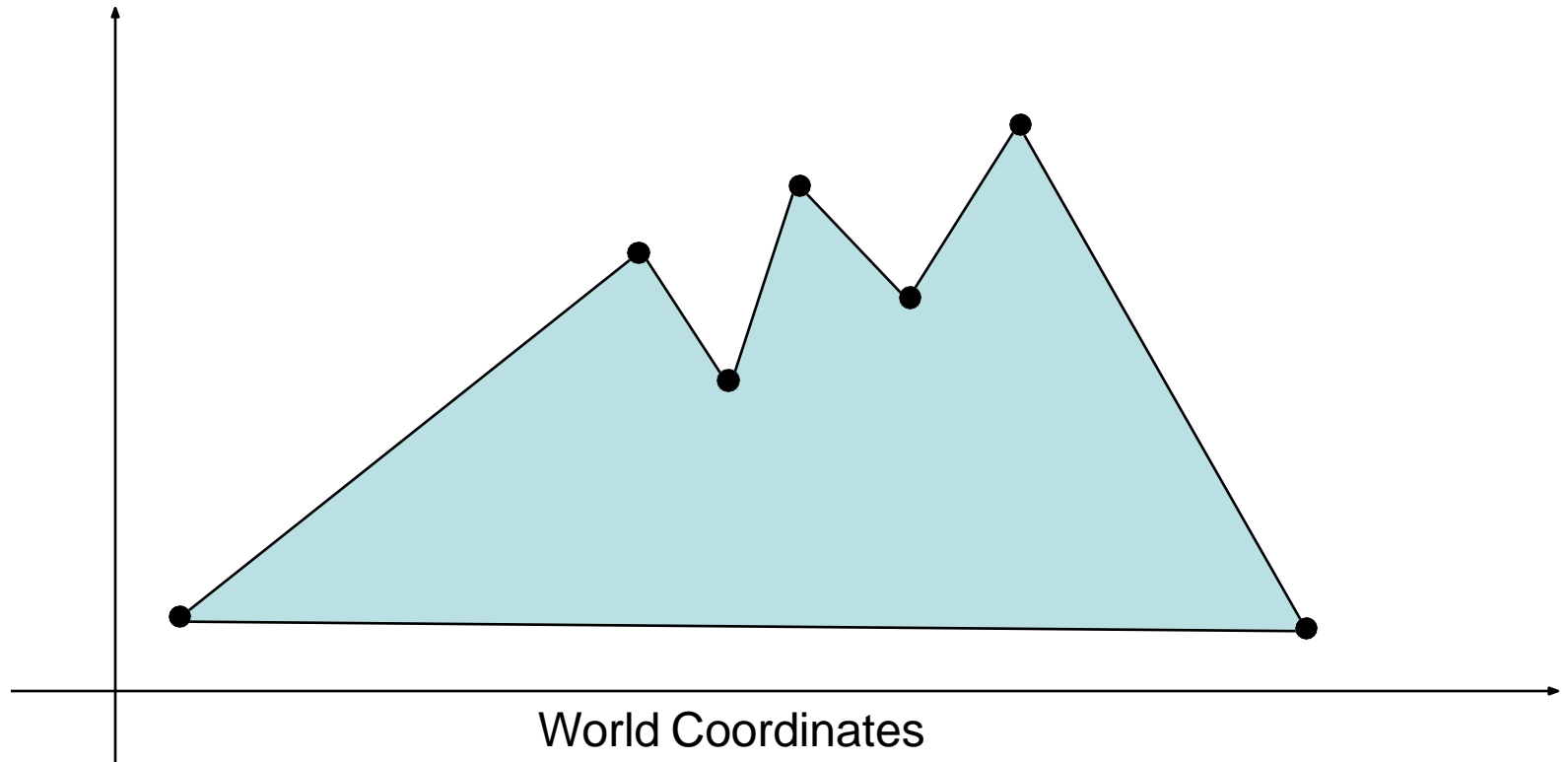
2D Clipping

- 1. Introduction**
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

2D Clipping

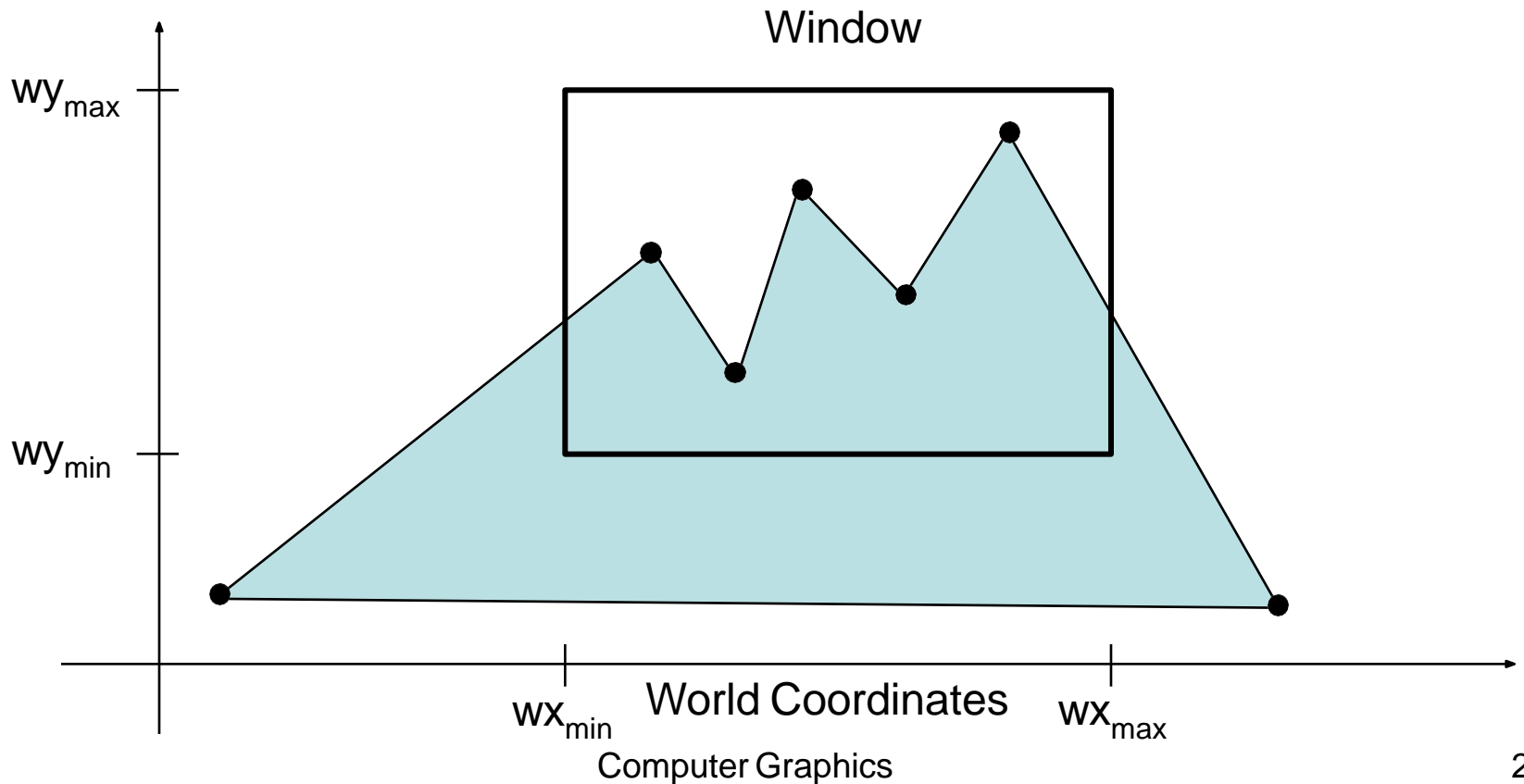
1. Introduction:

A scene is made up of a collection of objects specified in world coordinates



2D Clipping

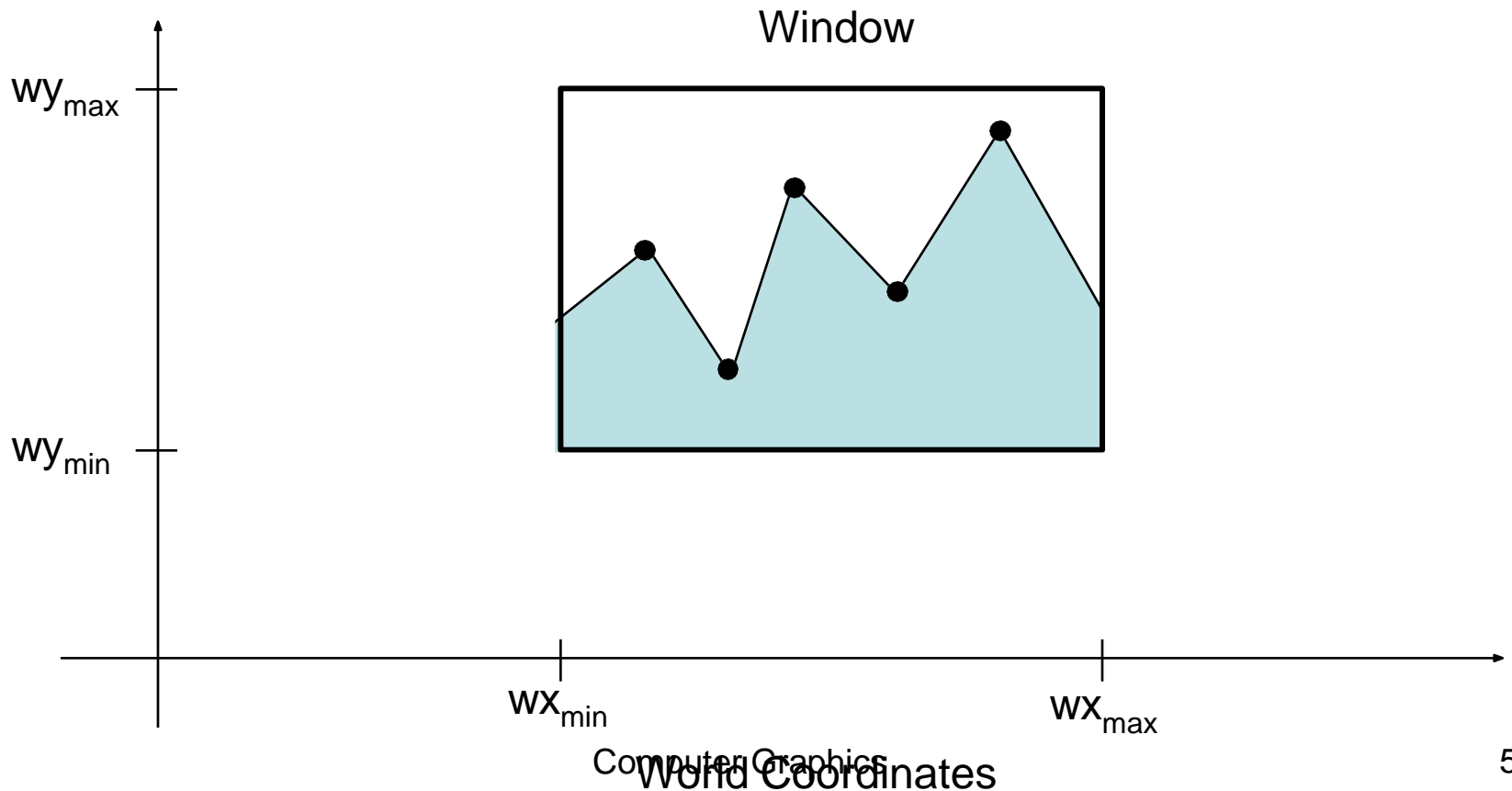
When we display a scene only those objects within a particular window are displayed



*

2D Clipping

Because drawing things to a display takes time we *clip* everything outside the window



2D Clipping

1.1 Definition:

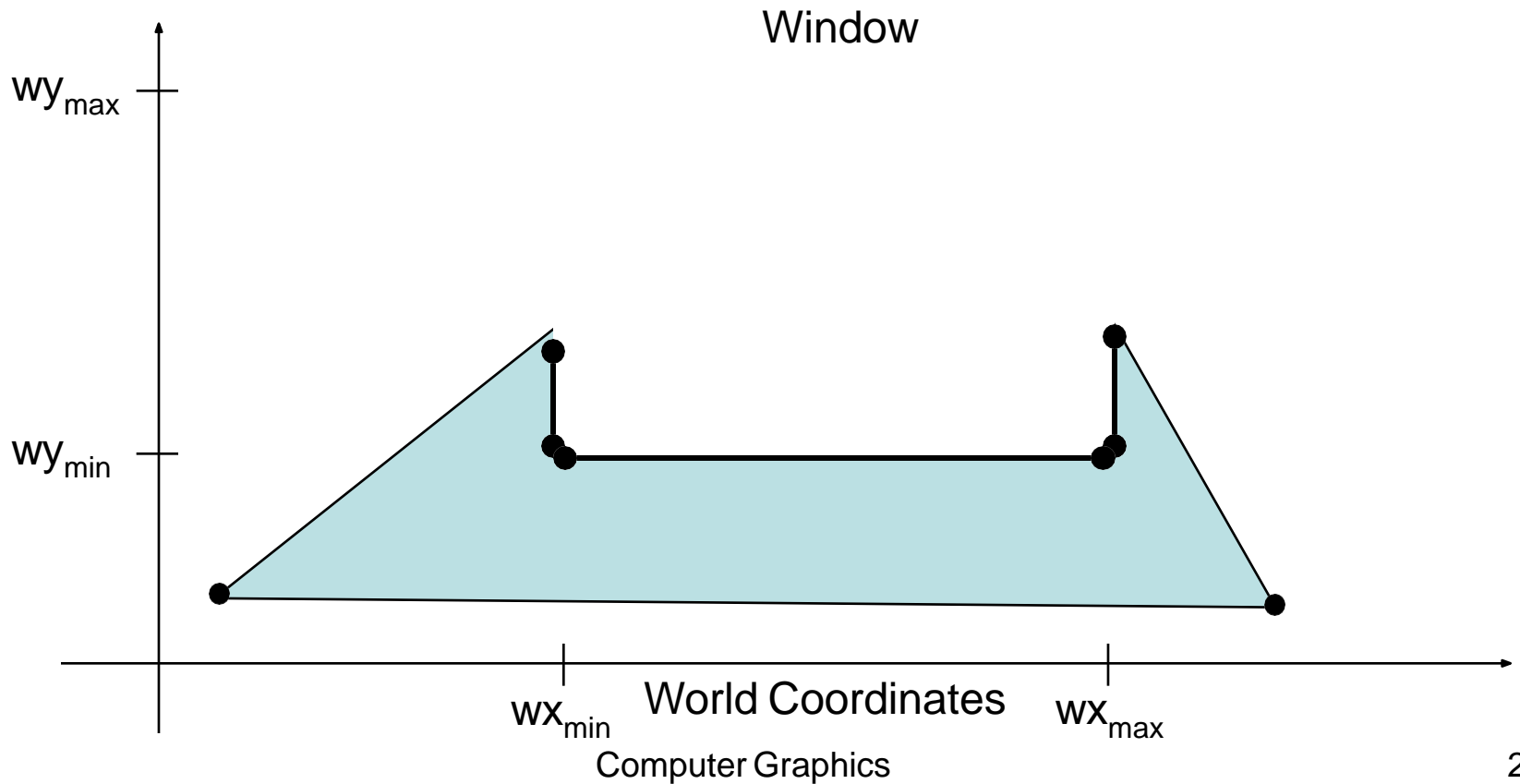
- *Clipping* is the process of determining which elements of the picture lie inside the window and are visible.
- By default, the “*clip window*” is the entire canvas
 - not necessary to draw outside the canvas
 - for some devices, it is damaging (plotters)
 - \
- Sometimes it is convenient to restrict the “clip window” to a smaller portion of the canvas
 - partial canvas redraw for menus, dialog boxes, other obscuration

2D Clipping

1.2 Shielding:

- *Shielding or exterior clipping* is the reverse operation of clipping where window act as the block used to abstract the view.
- Examples
 - A multi view window system
 - The design of page layouts in advertising or publishing applications or for adding labels or design patterns to picture.
 - Combining graphs, maps o schematics

2D Clipping

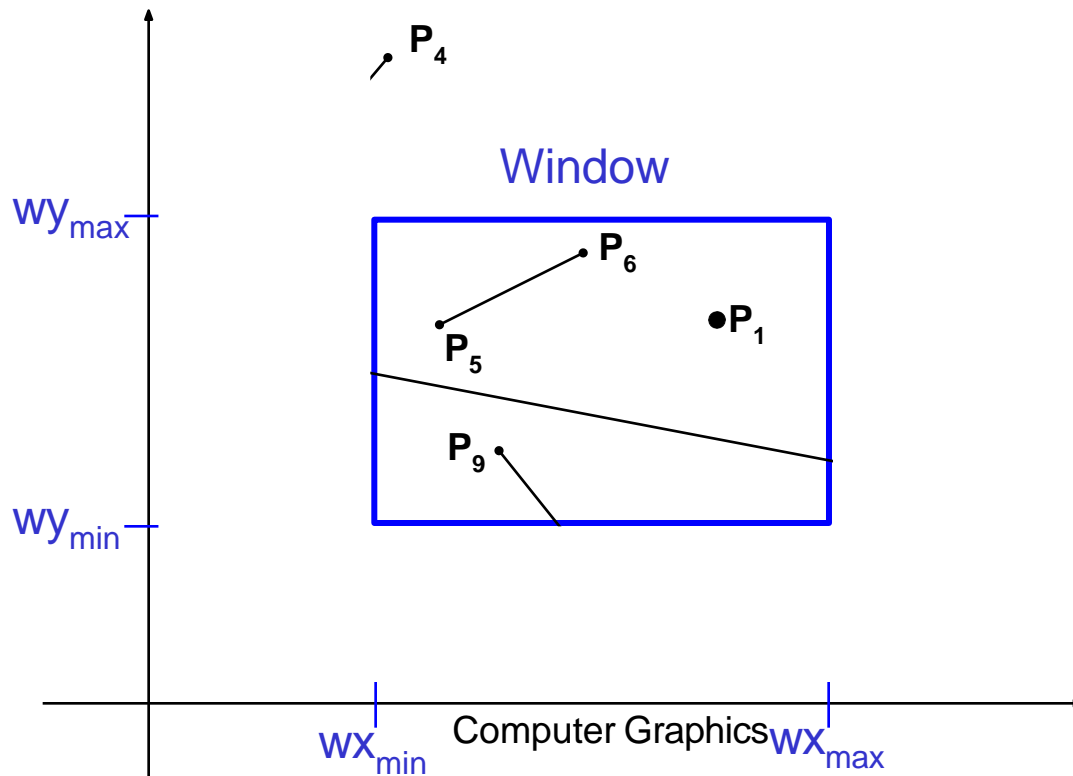


*

2D Clipping

1.3 Example:

For the image below consider which lines and points should be kept and which ones should be clipped against the clipping window



2D Clipping

4. Applications:

- Extract part of a defined scene for viewing.
- Drawing operations such as erase, copy, move etc.
- Displaying multi view windows.
- Creating objects using solid modeling techniques.
- Anti-aliasing line segments or object boundaries.
- Identify visible surfaces in 3D views.

2D Clipping

1.5 Types of clipping:

- Three types of clipping techniques are used depending upon when the clipping operation is performed

a. Analytical clipping

- Clip it before you scan convert it
- used mostly for lines, rectangles, and polygons, where clipping algorithms are simple and efficient

2D Clipping

b. Scissoring

- Clip it during scan conversion
- a brute force technique
 - scan convert the primitive, only write pixels if inside the clipping region
 - easy for thick and filled primitives as part of scan line fill
 - if primitive is not much larger than clip region, most pixels will fall inside
 - can be more efficient than analytical clipping.

2D Clipping

c. Raster Clipping

- Clip it after scan conversion
- render everything onto a temporary canvas and copy the clipping region
 - wasteful, but simple and easy,
 - often used for text

2D Clipping

6. Levels of clipping:

- Point Clipping
- Line Clipping
- Polygon Clipping
- Area Clipping
- Text Clipping
- Curve Clipping

2D Clipping

1. Introduction
2. **Point Clipping**
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

Point Clipping

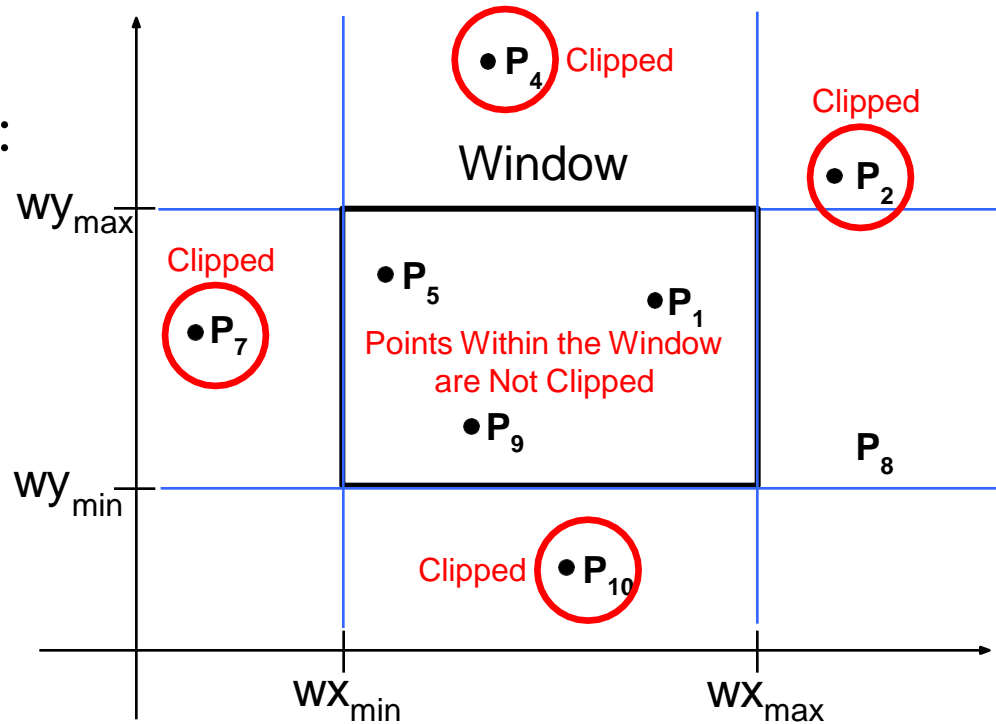
- Simple and Easy
- a point (x,y) is not clipped if:

$$wx_{min} \leq x \leq wx_{max}$$

&

$$wy_{min} \leq y \leq wy_{max}$$

- otherwise it is clipped

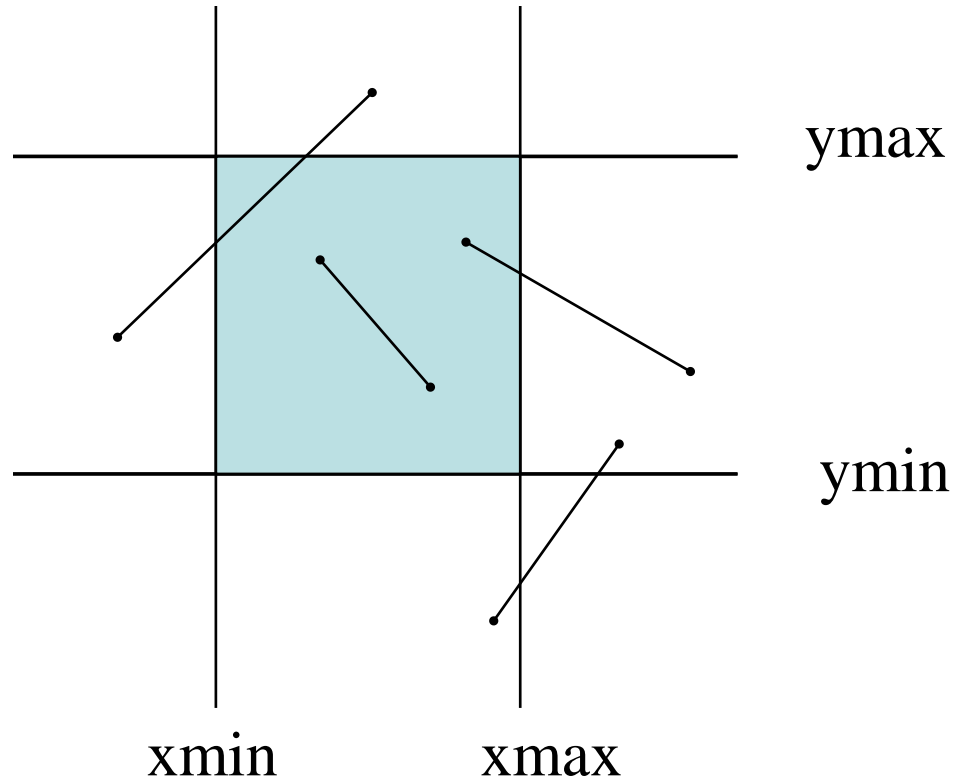


2D Clipping

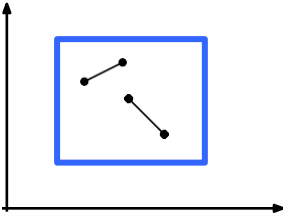
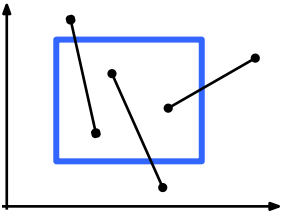
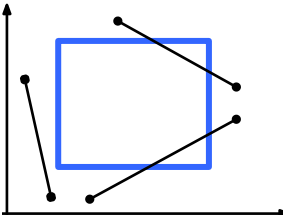
1. Introduction
2. Point Clipping
3. **Line Clipping**
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

Line Clipping

- It is Harder than point clipping
- We first examine the end-points of each line to see if they are in the window or not
 - Both endpoints inside, line trivially accepted
 - One in and one out, line is partially inside
 - Both outside, might be partially inside
 - What about trivial cases?



Line Clipping

Situation	Solution	Example
Both end-points inside the window	Don't clip	 A 2D coordinate system with x and y axes. A blue square window is centered in the first quadrant. A black line segment with dots at both ends is entirely contained within the square.
One end-point inside the window, one outside	Must clip	 A 2D coordinate system with x and y axes. A blue square window is centered in the first quadrant. A black line segment with dots at both ends passes through the square, with one end inside and one end outside.
Both end-points outside the window	Don't know!	 A 2D coordinate system with x and y axes. A blue square window is centered in the first quadrant. A black line segment with dots at both ends is entirely outside the square, passing through it without touching it.

2D Line Clipping Algorithms

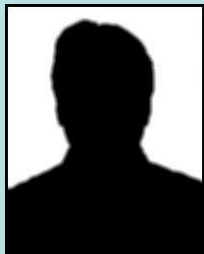
1. Analytical Line Clipping
2. **Cohen Sutherland Line Clipping**
3. Liang Barsky Line Clipping

Cohen-Sutherland Line Clipping

- An efficient line clipping algorithm
- The key advantage of the algorithm is that it vastly reduces the number of line intersections that must be calculated.



Dr. Ivan E. Sutherland co-developed the Cohen-Sutherland clipping algorithm. Sutherland is a graphics giant and includes amongst his achievements the invention of the head mounted display.



Cohen is something of a mystery – can anybody find out who he was?

Cohen-Sutherland Line Clipping

- Two phases Algorithm

Phase I: Identification Phase

All line segments fall into one of the following categories

1. Visible: Both endpoints lies inside
2. Invisible: Line completely lies outside
3. Clipping Candidate: A line neither in category 1 or 2

Phase II: Perform Clipping

Compute intersection for all lines that are candidate for clipping.

Cohen-Sutherland Line Clipping

Phase I: Identification Phase: World space is divided into regions based on the window boundaries

- Each region has a unique four bit region code
- Region codes indicate the position of the regions with respect to the window

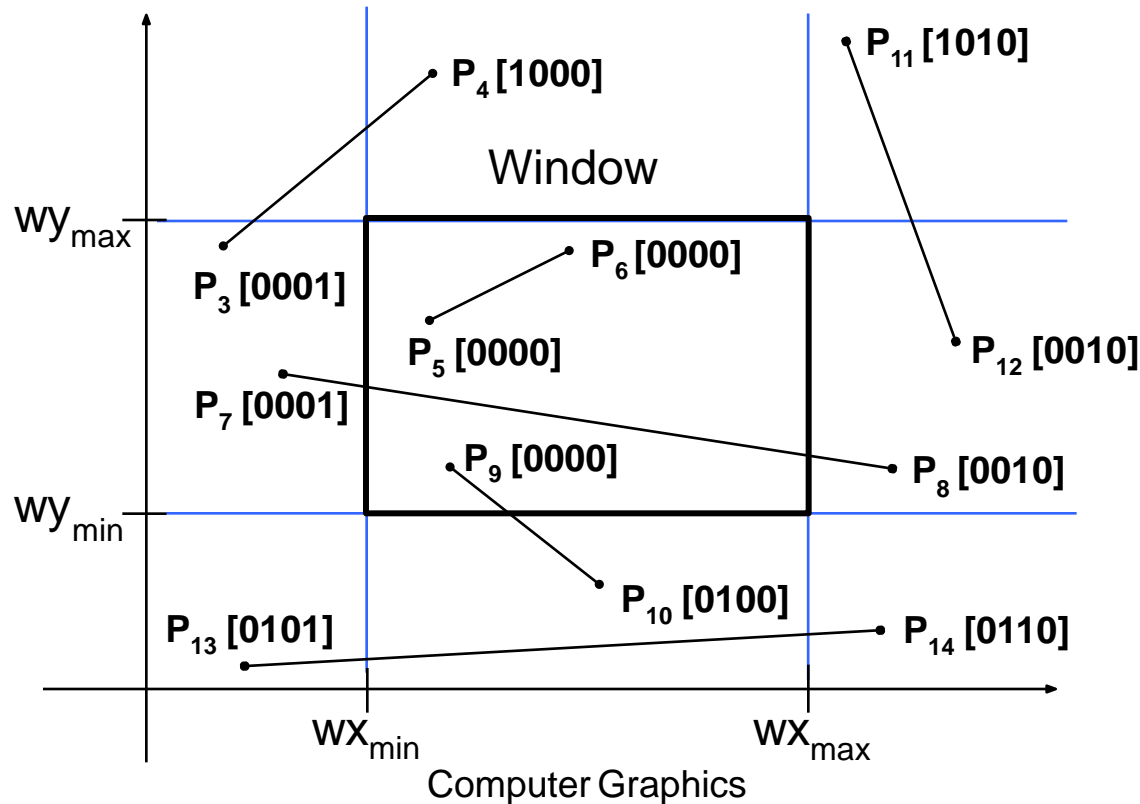
4	3	2	1
Top	below	right	left

Region Code Legend

1001	1000	1010
0001	0000 Window	0010
0101	0100	0110

Cohen-Sutherland Line Clipping

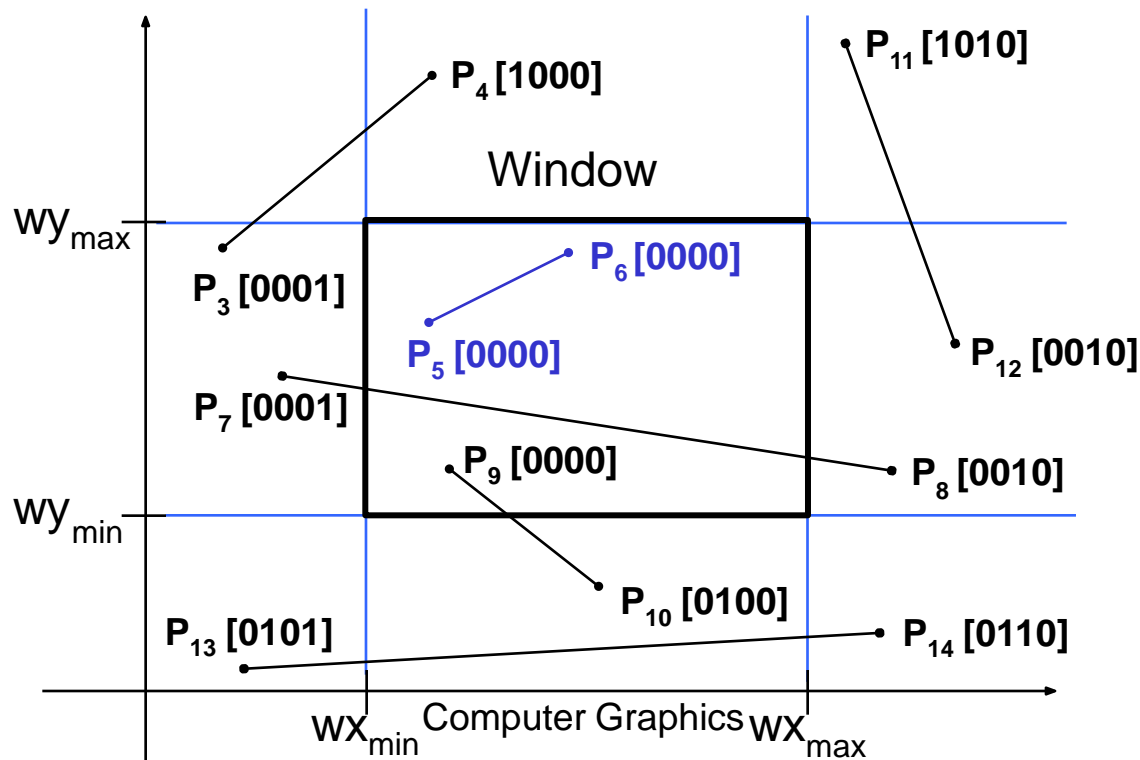
Every end-point is labelled with the appropriate region code



*

Cohen-Sutherland Line Clipping

Visible Lines: Lines completely contained within the window boundaries have region code [0000] for both end-points so are not clipped

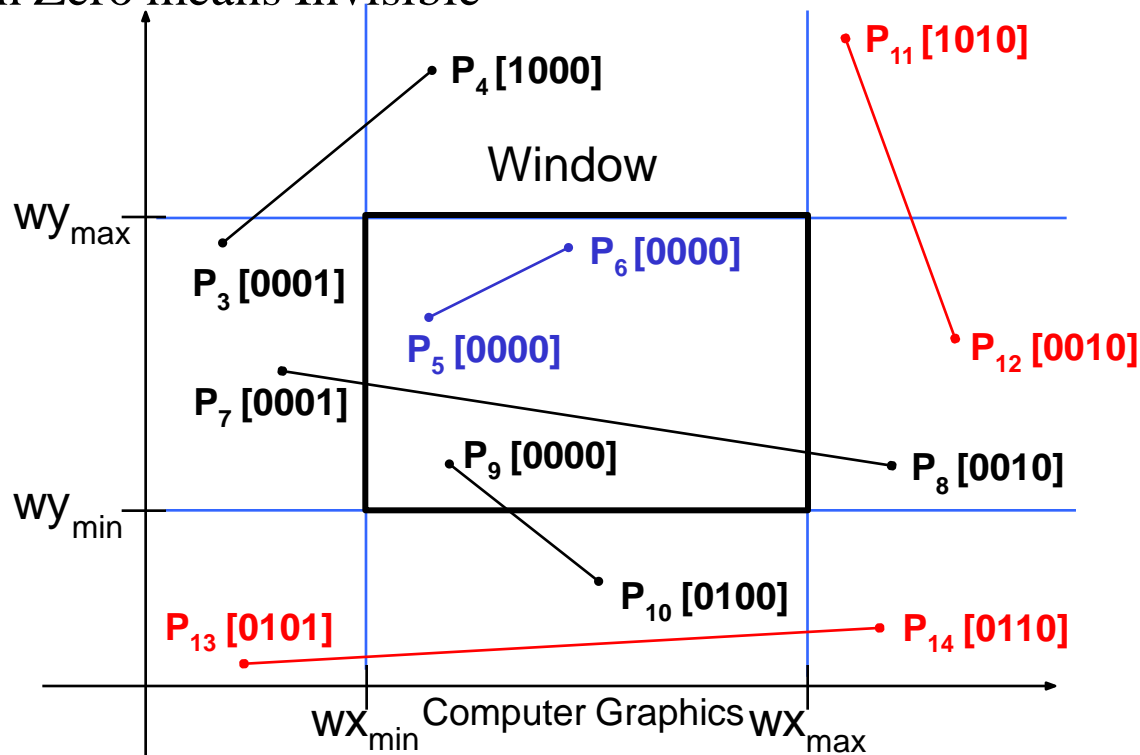


*

Cohen-Sutherland Line Clipping

Invisible Lines: Any line with a common set bit in the region codes of both end-points can be clipped completely

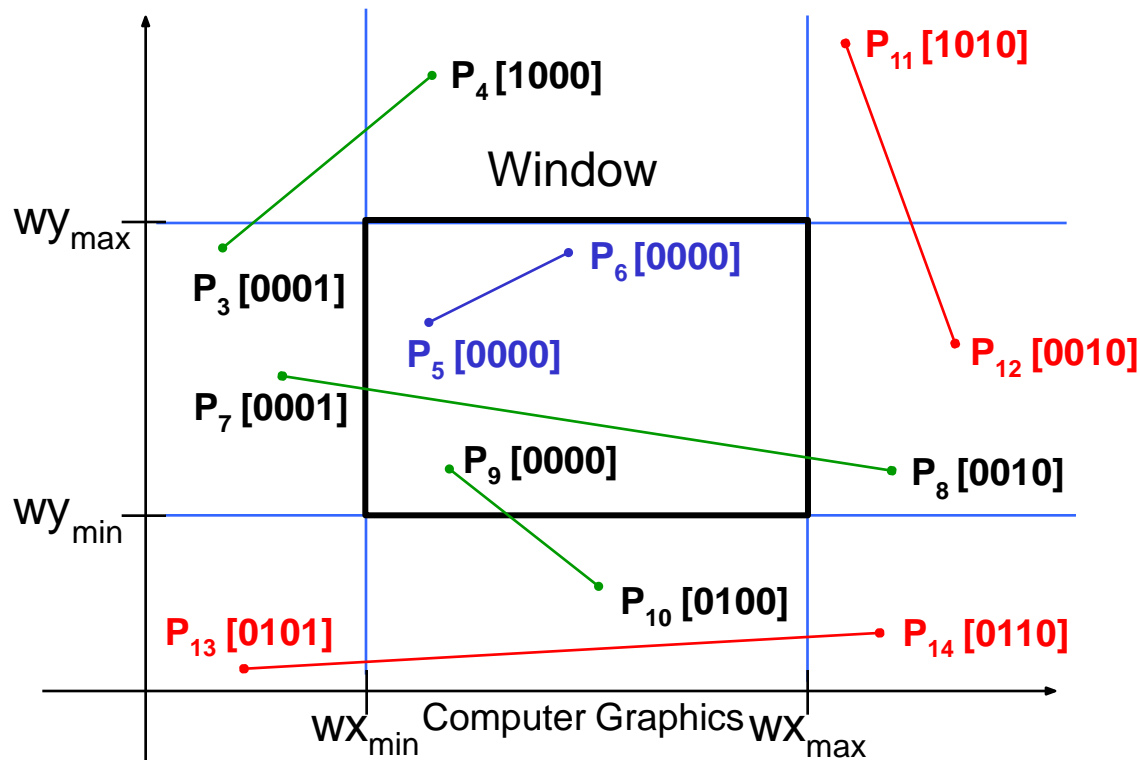
- The AND operation can efficiently check this
- Non Zero means Invisible



Cohen-Sutherland Line Clipping

Clipping Candidates: Lines that cannot be identified as completely inside or outside the window may or may not cross the window interior. These lines are processed in Phase II.

- If AND operation result in 0 the line is candidate for clipping



Cohen-Sutherland Line Clipping

Assigning Codes

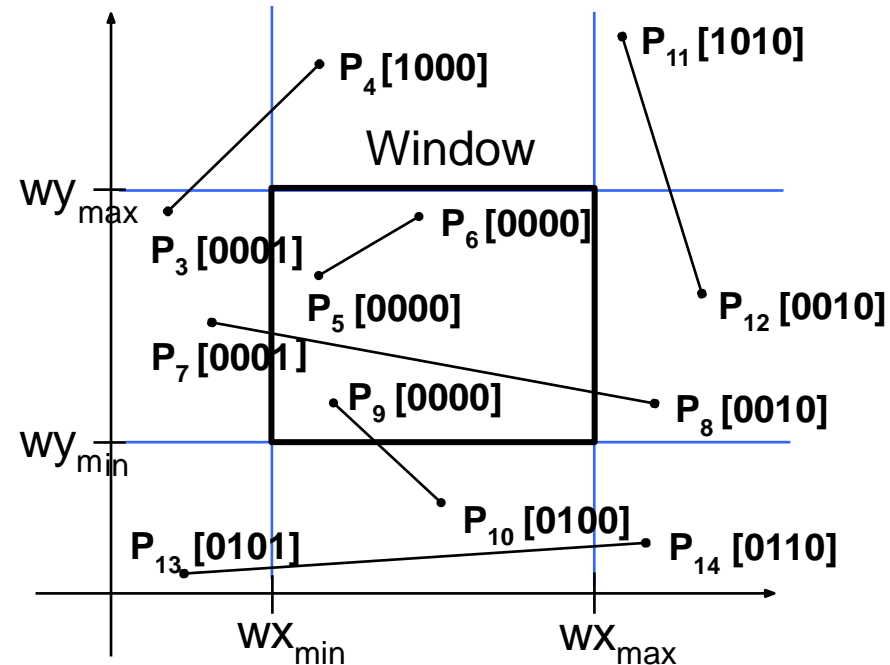
– Let point (x,y) is be given
code $b_3b_2b_1b_0$:

bit 3 = 1 if $wy_{max} - y \leq 0$

bit 2 = 1 if $y - wy_{min} \leq 0$

bit 1 = 1 if $wx_{max} - x \leq 0$

bit 0 = 1 if $x - wx_{min} \leq 0$



Cohen-Sutherland Clipping Algorithm

Phase II: Clipping Phase: Lines that are in category 3 are now processed as follows:

- Compare an end-point outside the window to a boundary (choose any order in which to consider boundaries e.g. left, right, bottom, top) and determine how much can be discarded
- If the remainder of the line is entirely inside or outside the window, retain it or clip it respectively
- Otherwise, compare the remainder of the line against the other window boundaries
- Continue until the line is either discarded or a segment inside the window is found

Cohen-Sutherland Line Clipping

- Intersection points with the window boundaries are calculated using the line-equation parameters
- Consider a line with the end-points (x_1, y_1) and (x_2, y_2)
 - The y-coordinate of an intersection with a vertical window boundary can be calculated using:

$$y = y_1 + m (x_{boundary} - x_1)$$

where $x_{boundary}$ can be set to either wx_{min} or wx_{max}

- The x-coordinate of an intersection with a horizontal window boundary can be calculated using:

$$x = x_1 + (y_{boundary} - y_1) / m$$

where $y_{boundary}$ can be set to either wy_{min} or wy_{max}

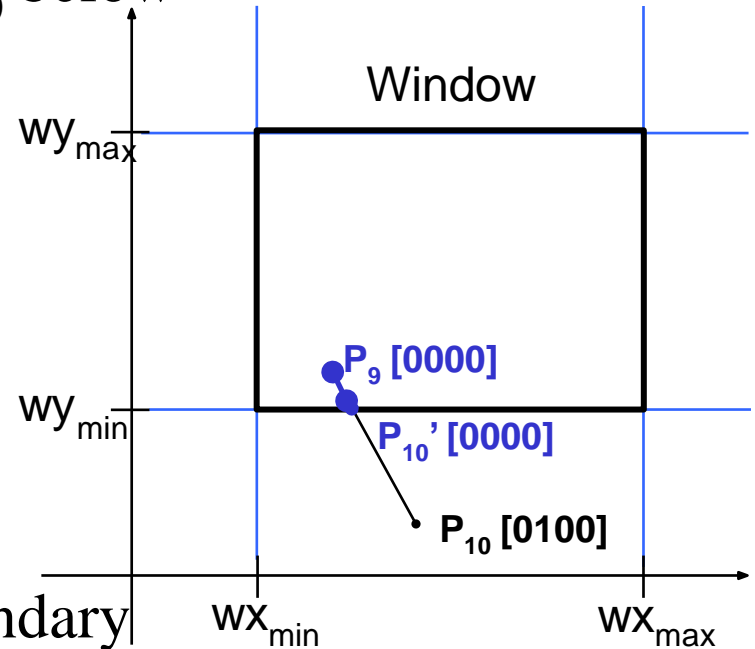
Cohen-Sutherland Line Clipping

- We can use the region codes to determine which window boundaries should be considered for intersection
 - To check if a line crosses a particular boundary we compare the appropriate bits in the region codes of its end-points
 - If one of these is a 1 and the other is a 0 then the line crosses the boundary.

Cohen-Sutherland Line Clipping

Example 1: Consider the line P_9 to P_{10} below

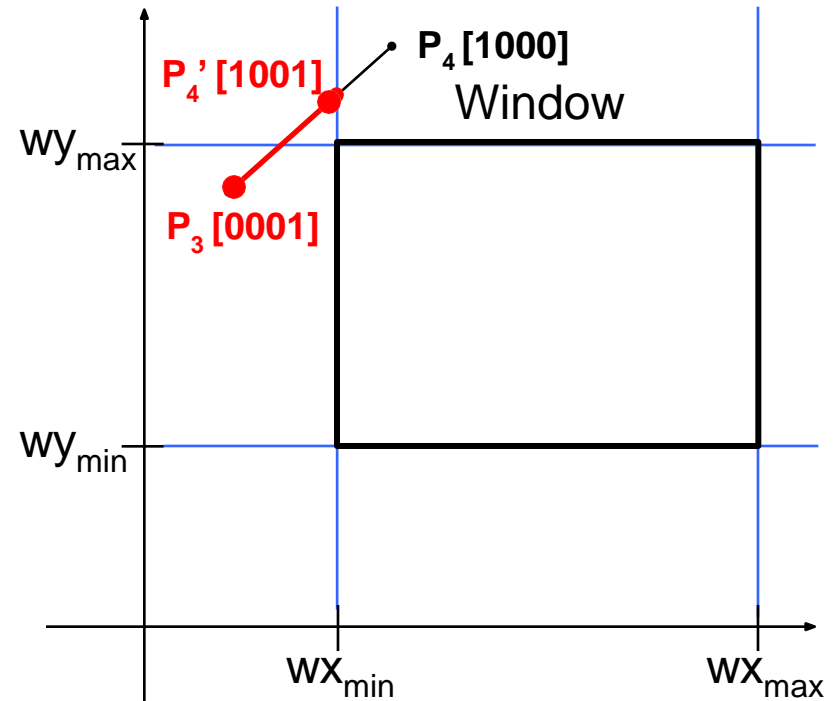
- Start at P_{10}
- From the region codes of the two end-points we know the line doesn't cross the left or right boundary
- Calculate the intersection of the line with the bottom boundary to generate point P_{10}'
- The line P_9 to P_{10}' is completely inside the window so is retained



Cohen-Sutherland Line Clipping

Example 2: Consider the line P_3 to P_4 below

- Start at P_4
- From the region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_4'

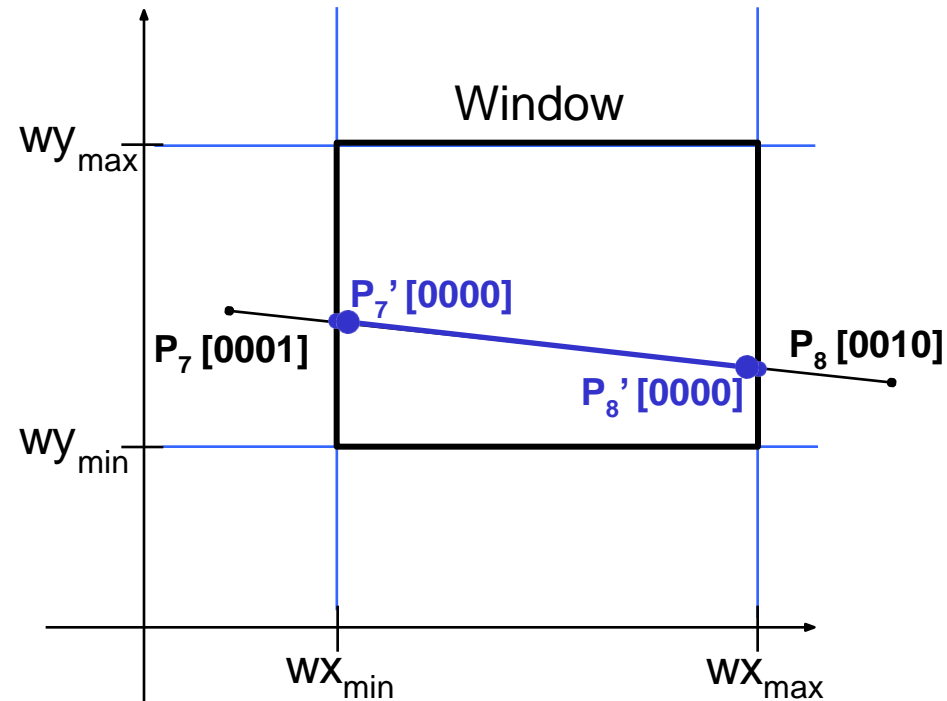


- The line P_3 to P_4' is completely outside the window so is clipped

Cohen-Sutherland Line Clipping

Example 3: Consider the line P_7 to P_8 below

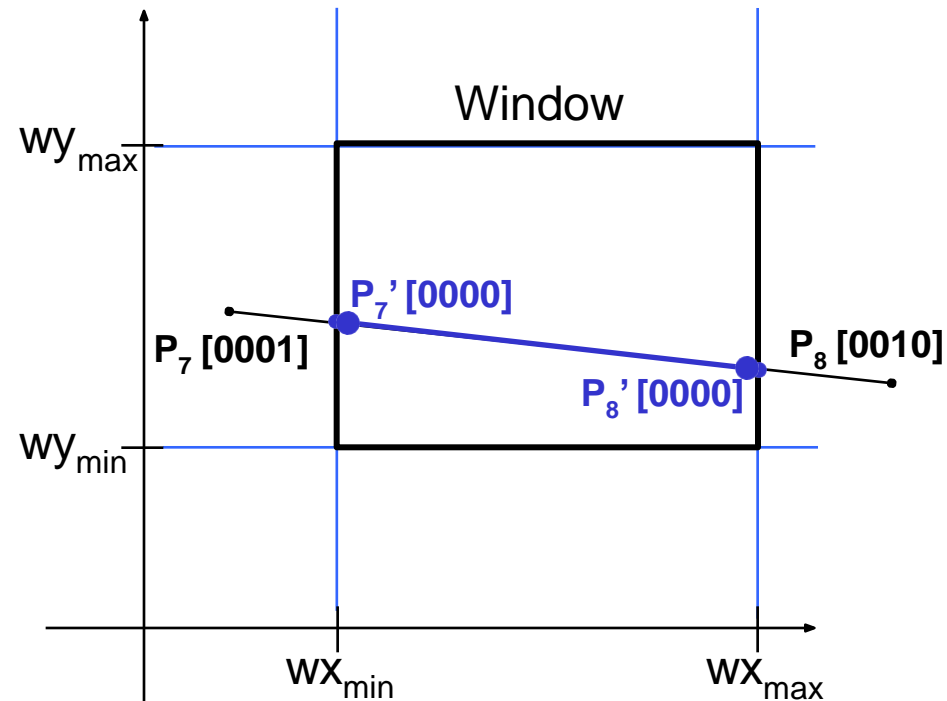
- Start at P_7
- From the two region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_7'



Cohen-Sutherland Line Clipping

Example 4: Consider the line P_7' to P_8

- Start at P_8
- Calculate the intersection with the right boundary to generate P_8'
- P_7' to P_8' is inside the window so is retained



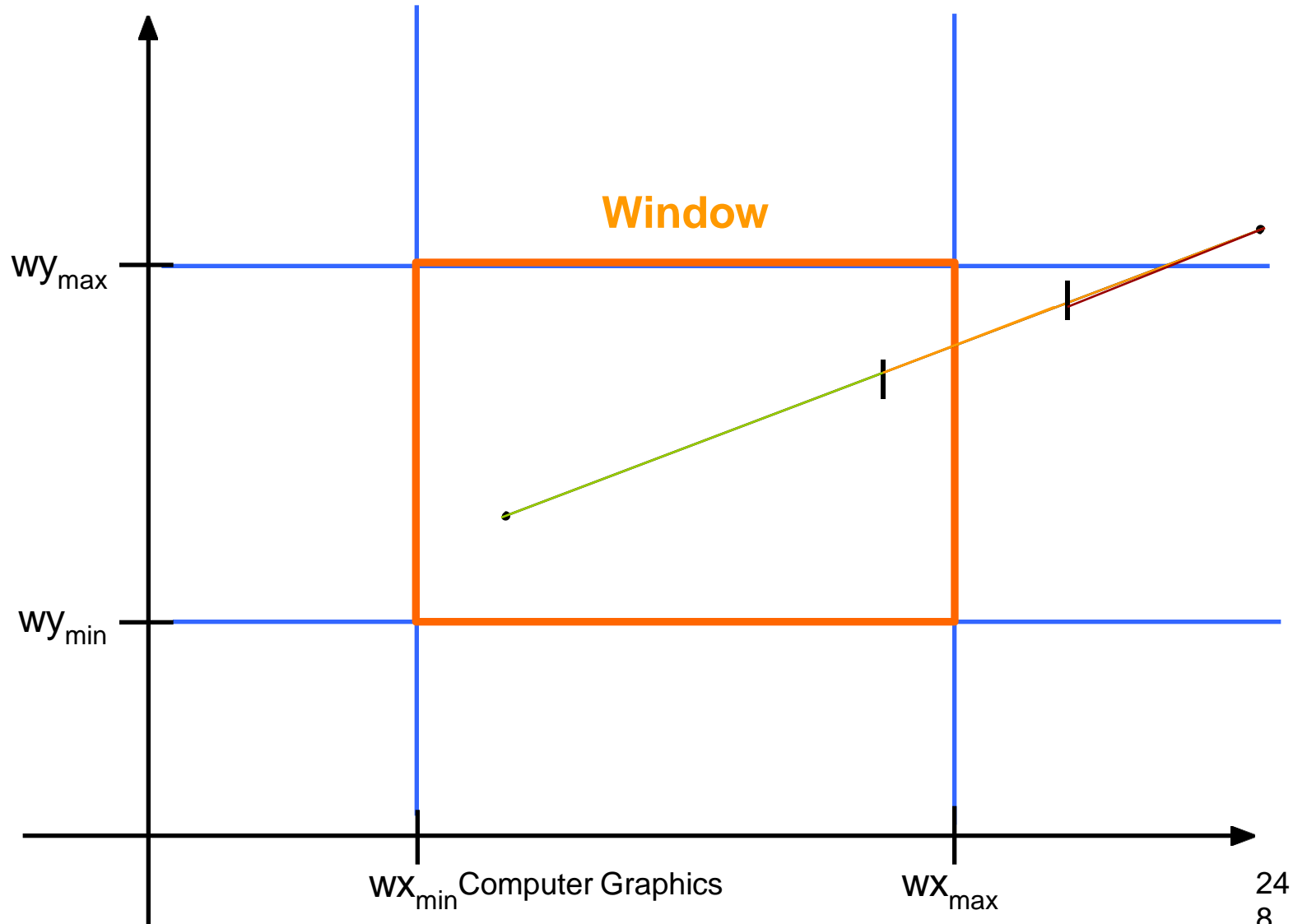
Cohen-Sutherland Line Clipping

Mid-Point Subdivision Method

– Algorithm

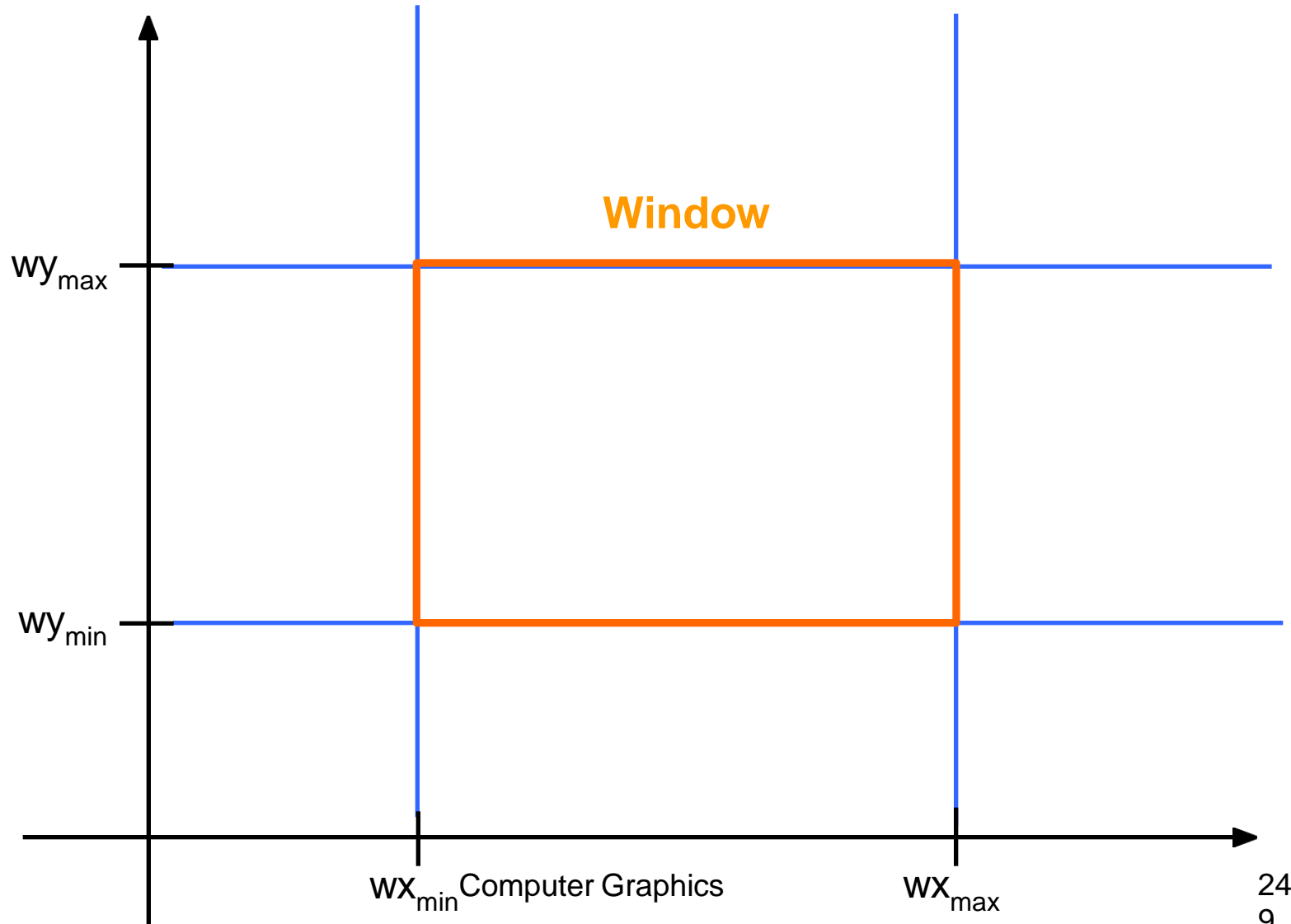
1. Initialise the list of lines to all lines
2. Classify lines as in *Phase I*
 - i. Assign 4 point bit codes to both end points $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$
 - ii. If $(a_3a_2a_1a_0 = b_3b_2b_1b_0 = 0)$ Line in category 1
 - iii. If $(a_3a_2a_1a_0) \text{ AND } (b_3b_2b_1b_0) \neq 0$ Line in category 2
 - iv. If $(a_3a_2a_1a_0) \text{ AND } (b_3b_2b_1b_0) = 0$ Line in category 3
3. Display all lines from the list in category 1 and remove;
4. Delete all lines from the list in category 2 as they are invisible;
5. Divide all lines of category 3 are into two smaller segments at mid-point (x_m, y_m) where $x_m = (x_1 + x_2)/2$ and $y_m = (y_1 + y_2)/2$
6. Remove the original line from list and enter its two newly created segments.
7. Repeat step 2-5 until list is null.

Cohen-Sutherland Line Clipping



*

Cohen-Sutherland Line Clipping



*

Cohen-Sutherland Line Clipping

Mid-Point Subdivision Method

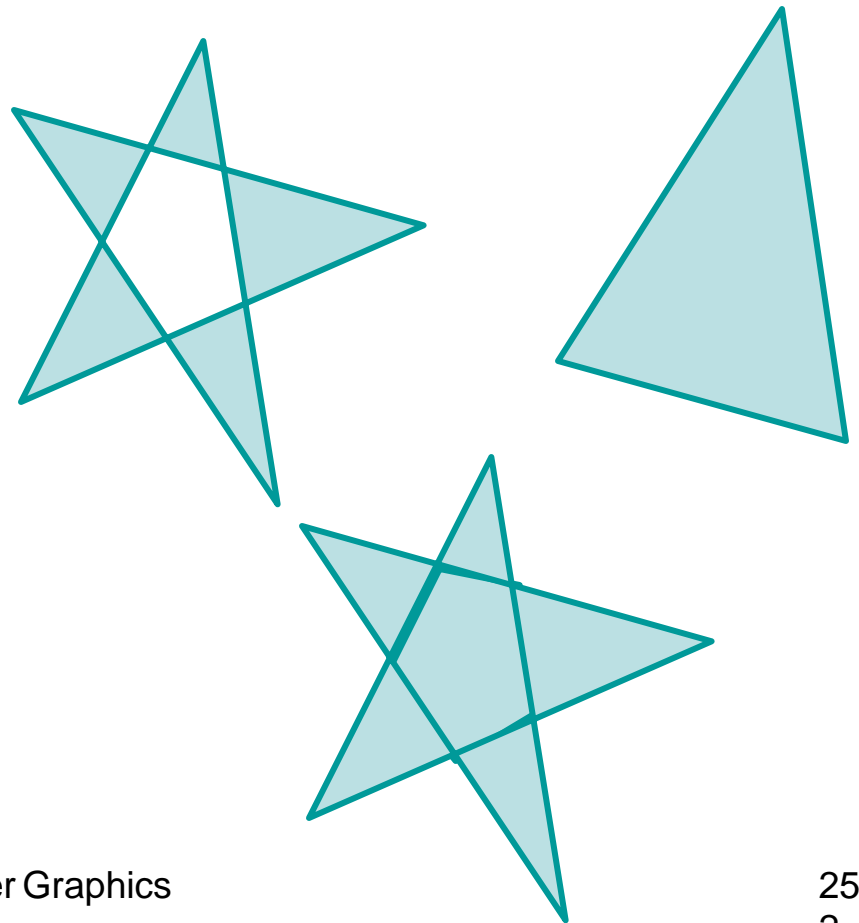
- Integer Version
- Fast as Division by 2 can be performed by simple shift right operation
- For NxN max dimension of line number of subdivisions required $\log_2 N$.
- Thus a 1024x1024 raster display require just 10 subdivisions.....

2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
- 4. Polygon / Area Clipping**
5. Text Clipping
6. Curve Clipping

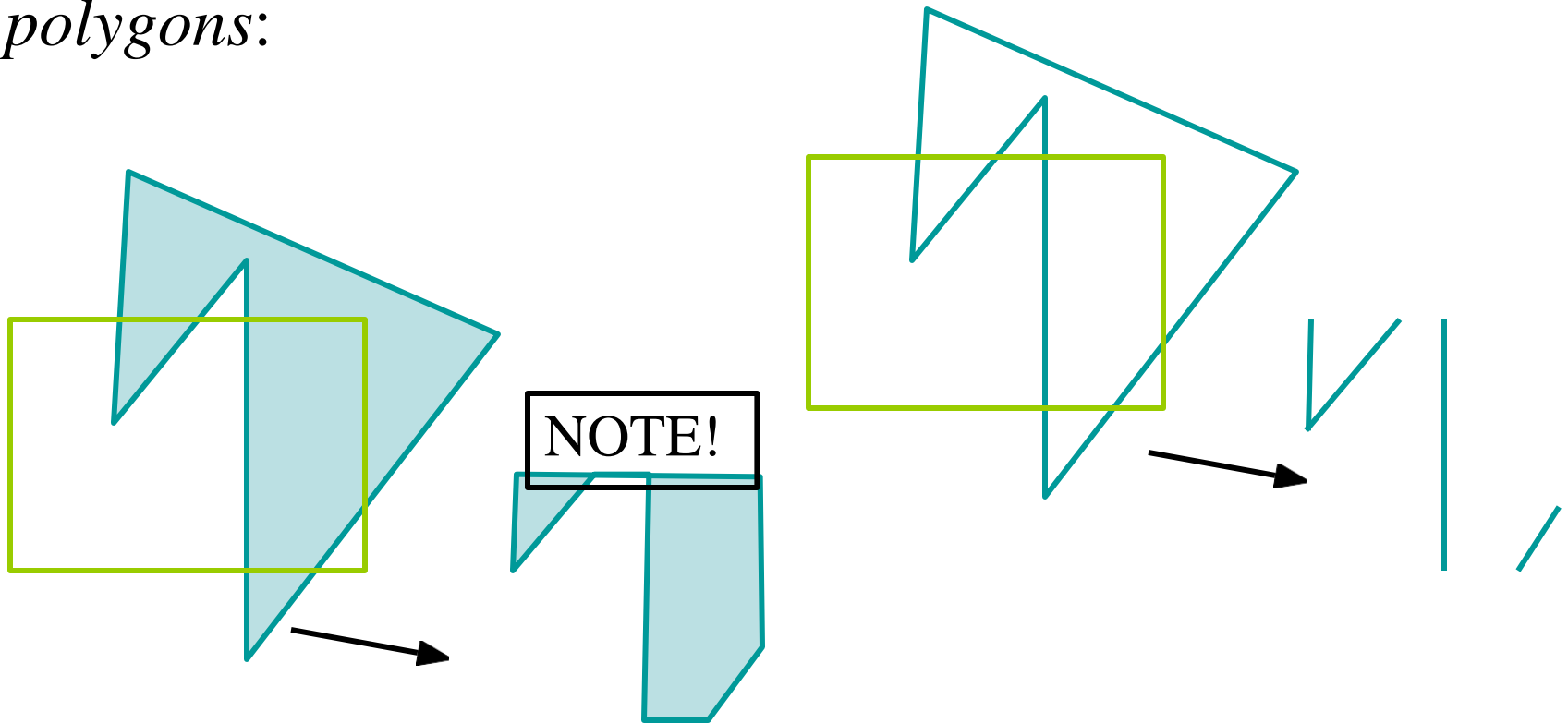
Polygon Clipping

- Polygons have a distinct *inside* and *outside*...
- Decided by
 - Even/Odd
 - Winding Number



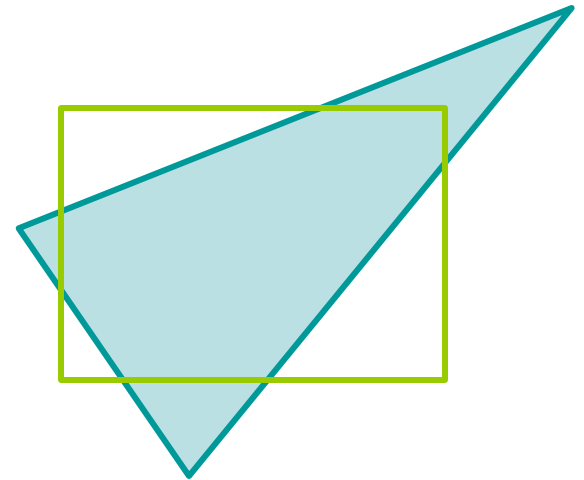
Polygon Clipping

- Note the difference between clipping *lines* and *polygons*:



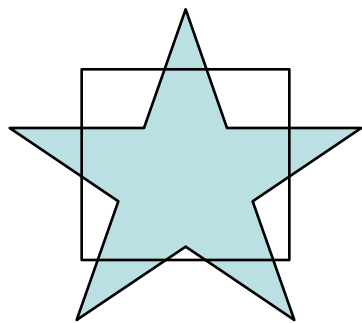
Polygon Clipping

- Some difficulties:
 - Maintaining correct inside/outside
 - Variable number of vertices
 - Handle screen corners correctly

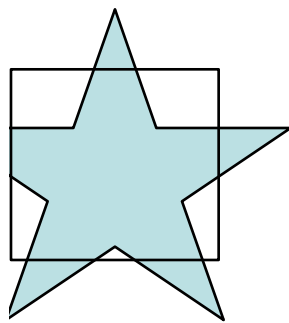


Sutherland-Hodgman Area Clipping

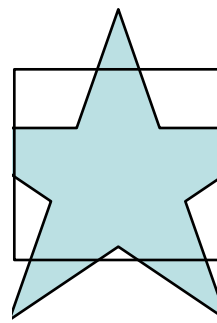
- A technique for clipping areas developed by Sutherland & Hodgman
- Put simply the polygon is clipped by comparing it against each boundary in turn



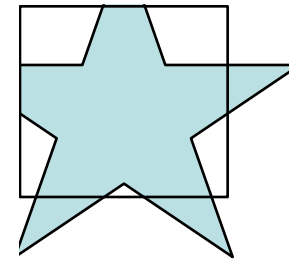
Original Area



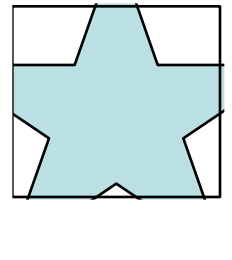
Clip Left



Clip Right

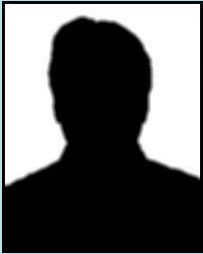


Clip Top



Clip Bottom

Sutherland turns up again. This time with Gary Hodgman with whom he worked at the first ever graphics company Evans & Sutherland



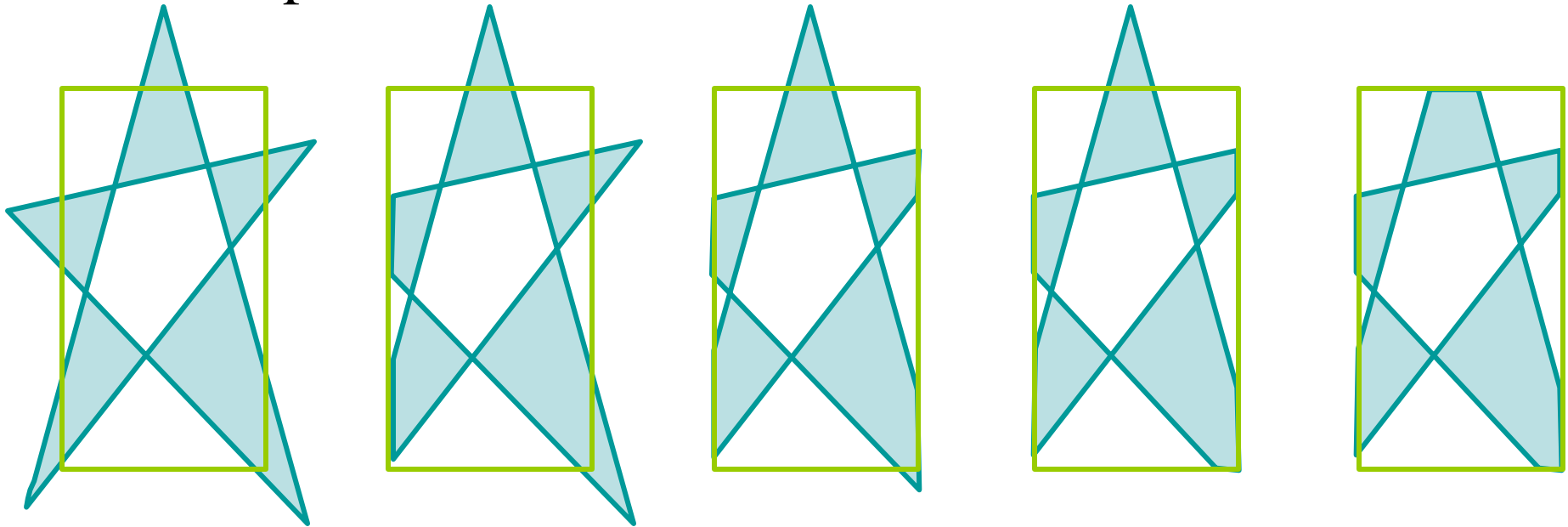
Sutherland-Hodgeman Polygon Clipping

1. Basic Concept:

- Simplify via separation
- Clip whole polygon against one edge
 - Repeat with output for other 3 edges
 - Similar for 3D
- You can create intermediate vertices that get thrown out

Sutherland-Hodgeman Polygon Clipping

- Example



Start

Left

Right

Bottom

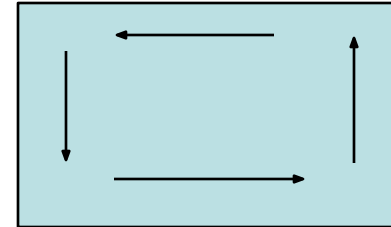
Top

Note that the point one of the points added when clipping on the right gets removed when we clip with bottom

Sutherland-Hodgeman Polygon Clipping

2. Algorithm:

Let (P_1, P_2, \dots, P_N) be the vertex list of the Polygon to be clipped and E be the edge of *counterclockwise oriented, convex clipping window*.



We clip each edge of the polygon in turn against each window edge E , forming a new polygon whose vertices are determined as follows:

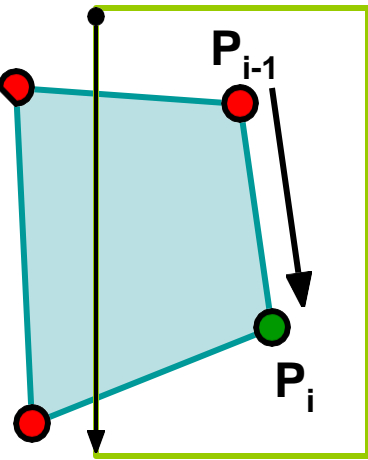
Sutherland-Hodgeman Polygon Clipping

Four cases

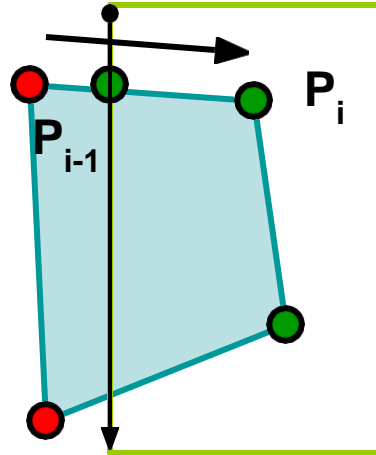
1. **Inside:** If both P_{i-1} and P_i are to the left of window edge vertex then P_i is placed on the output vertex list.
2. **Entering:** If P_{i-1} is to the right of window edge and P_i is to the left of window edge vertex then intersection (I) of P_{i-1} P_i with edge E and P_i are placed on the output vertex list.
3. **Leaving:** If P_{i-1} is to the left of window edge and P_i is to the right of window edge vertex then only intersection (I) of P_{i-1} P_i with edge E is placed on the output vertex list.
4. **Outside:** If both P_{i-1} and P_i are to the right of window edge nothing is placed on the output vertex list.

Sutherland-Hodgeman Polygon Clipping

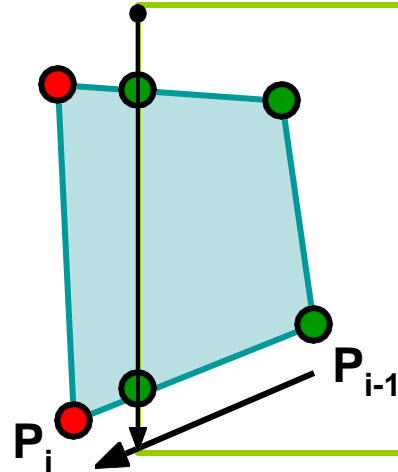
Creating New Vertex List



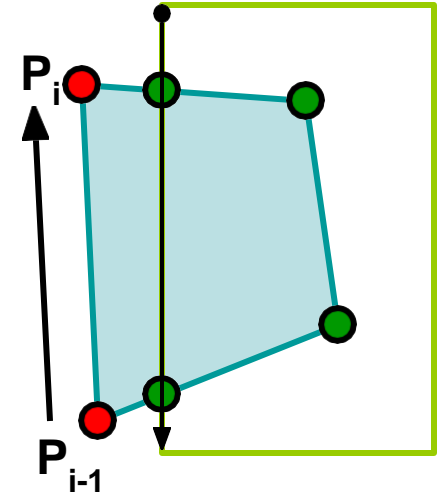
in \square in
save ending vert
Inside
(1 output)



out \square in
save new clip vert
and ending vert
Entering
(2 outputs)



in \square out
save new clip vert
Leaving
(1 output)

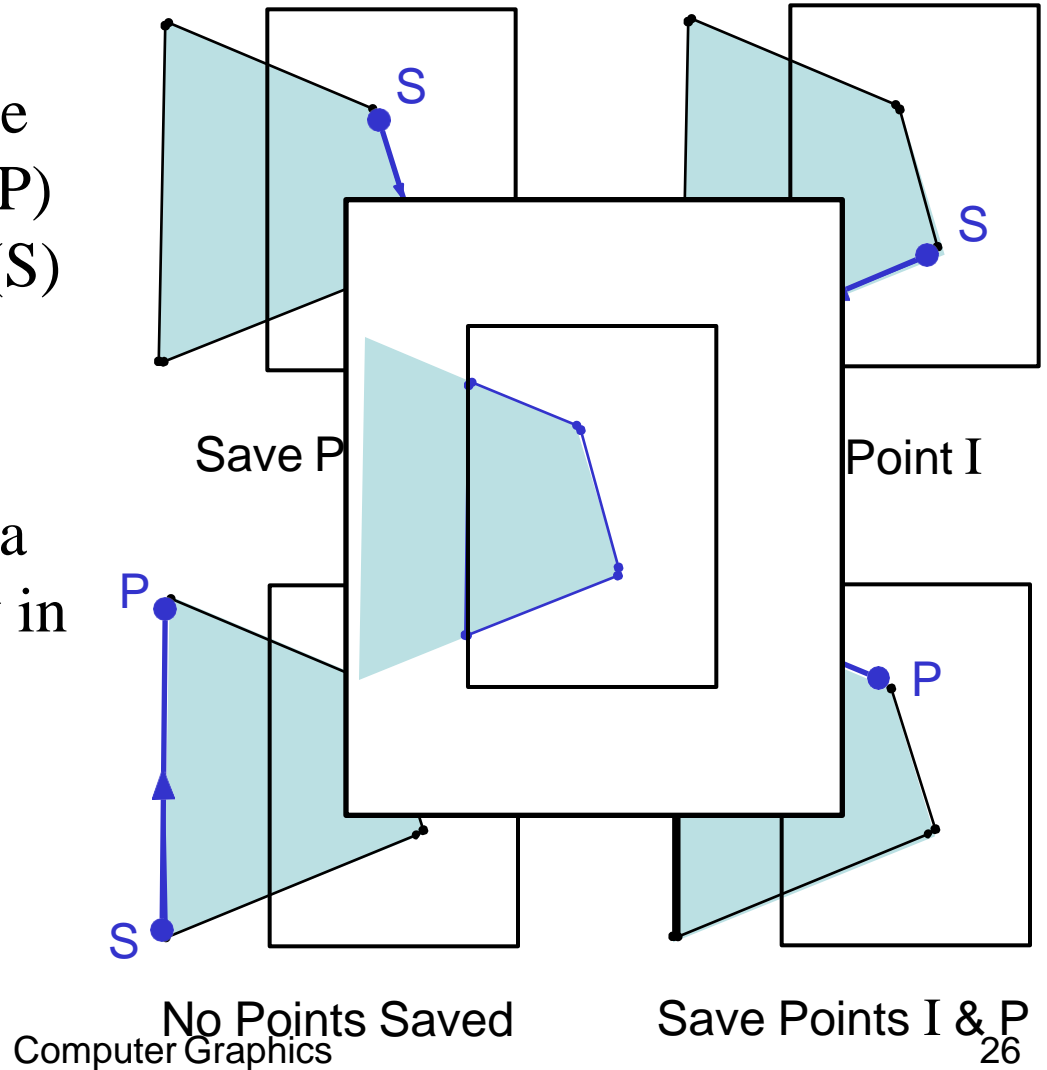


out \square out
save nothing
Outside
(0 output)

*

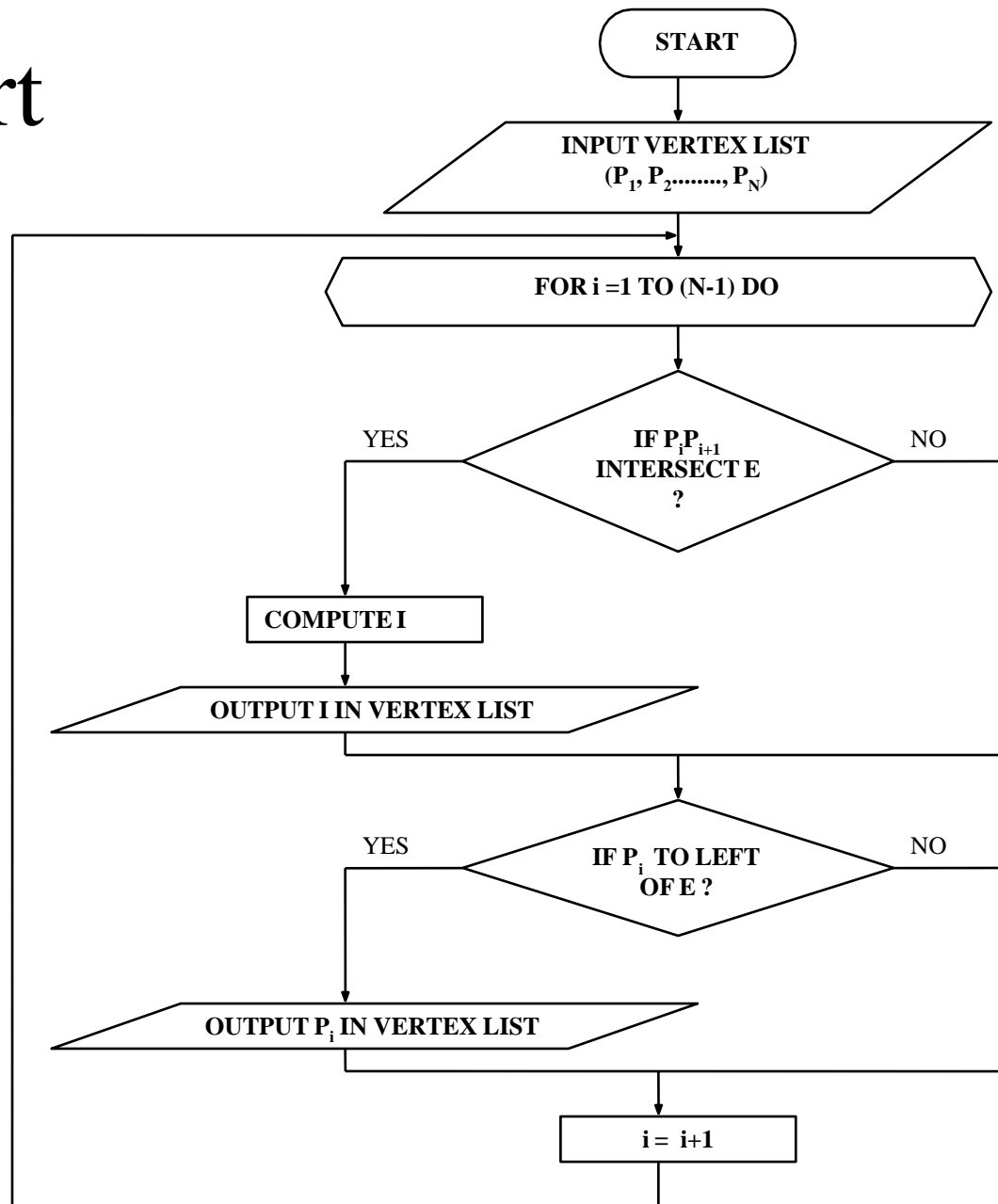
Sutherland-Hodgman Polygon Clipping

- Each example shows the point being processed (P) and the previous point (S)
- Saved points define area clipped to the boundary in question



*

Flow Chart

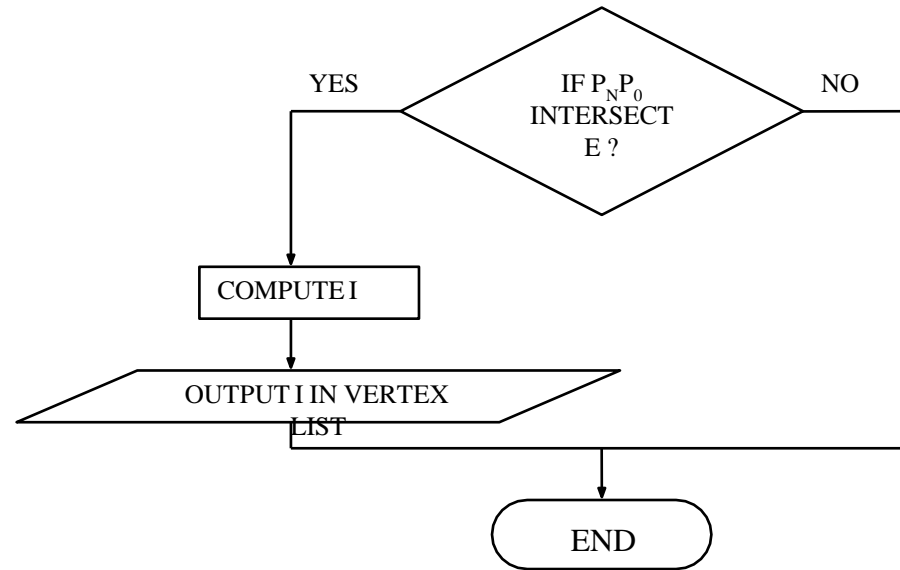


Special case for
first Vertex

*

Flow Chart

Special case for
first Vertex



YOU CAN ALSO APPEND AN ADDITIONAL VERTEX
 $P_{N+1} = P_1$ AND AVOID SPECIAL CASE FOR FIRST
VERTEX

Sutherland-Hodgeman Polygon Clipping

Inside/Outside Test:

Let $P(x,y)$ be the polygon vertex which is to be tested against edge E defined from $A(x_1, y_1)$ to $B(x_2, y_2)$. Point P is to be said to the left (inside) of E or AB iff

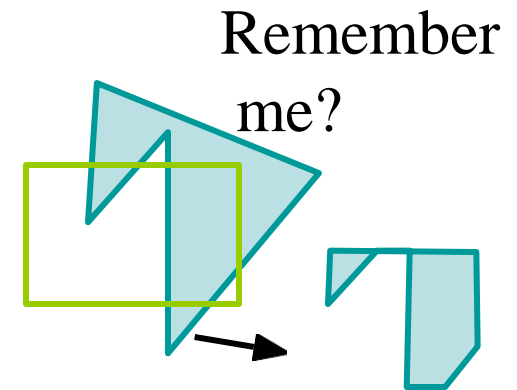
$$\frac{y - y_1}{y_2 - y_1} - \frac{x - x_1}{x_2 - x_1} > 0$$

or $C = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) > 0$

otherwise it is said to be the right/Outside of edge E

Weiler-Atherton Polygon Clipping

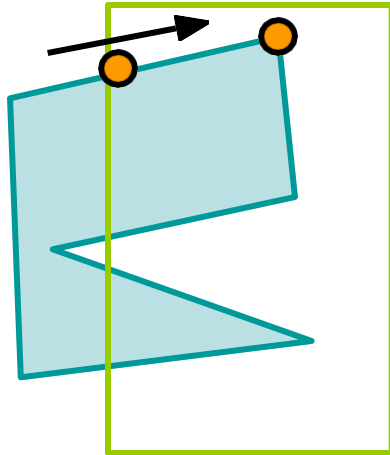
- Problem with Sutherland-Hodgeman:
 - Concavities can end up linked



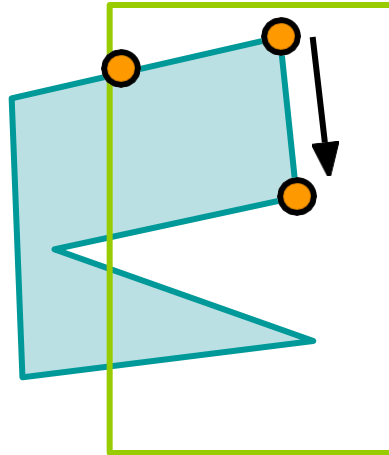
- Weiler-Atherton creates separate polygons in such cases

Weiler-Atherton Polygon Clipping

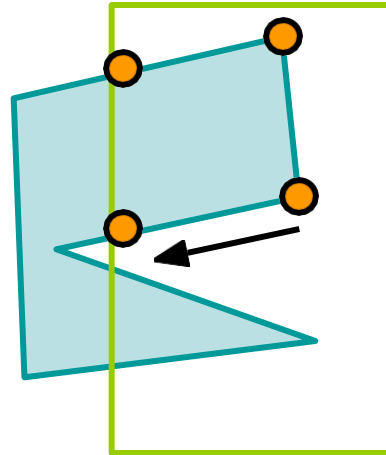
- Example



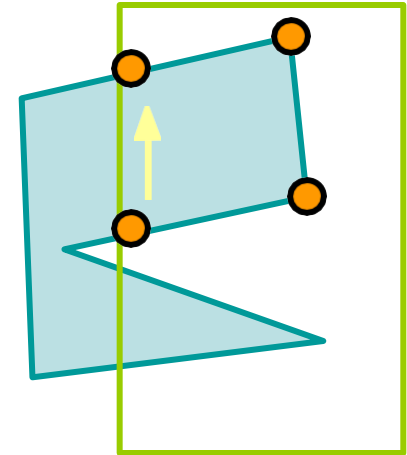
add clip pt.
and end pt.



add end pt.



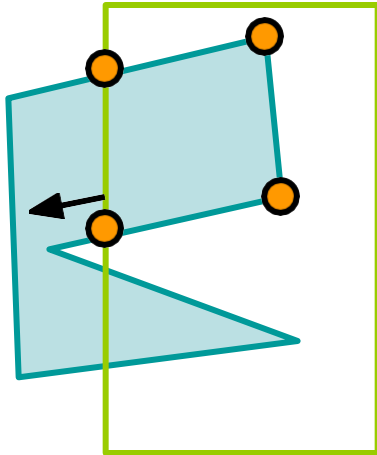
add clip pt.
cache old dir.



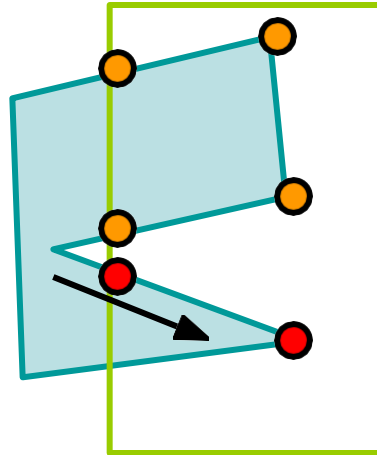
follow clip edge until
a) new crossing
found
b) reach pt. already
added

Weiler-Atherton Polygon Clipping

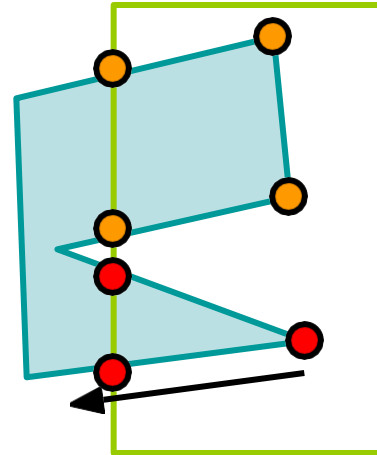
- Example (cont)



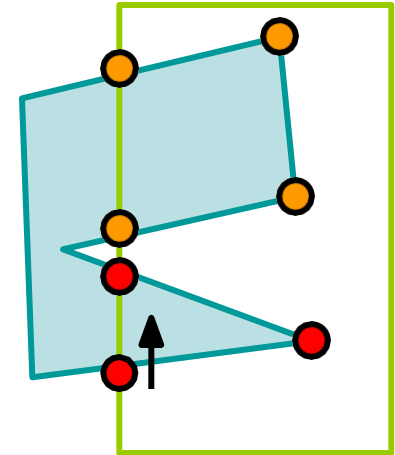
continue from
cached location



add clip pt.
and end pt.



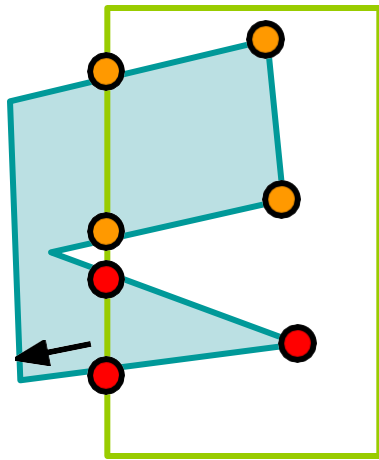
add clip pt.
cache dir.



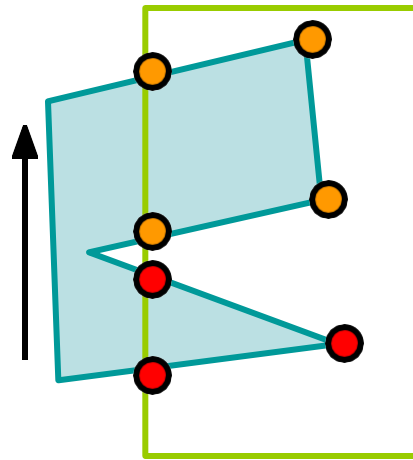
follow clip edge until
a) new crossing
found
b) reach pt. already
added

Weiler-Atherton Polygon Clipping

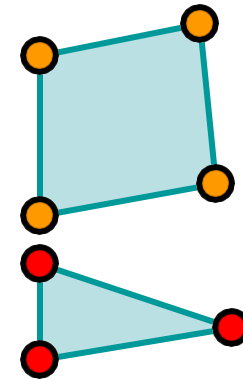
- Example (concluded)



continue from
cached location



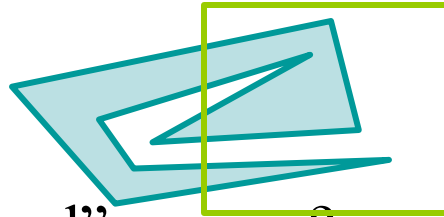
nothing added
finished



Final result:
Two *unconnected*
polygons

Weiler-Atherton Polygon Clipping

- Difficulties:
 - What if the polygon re-crosses edge?



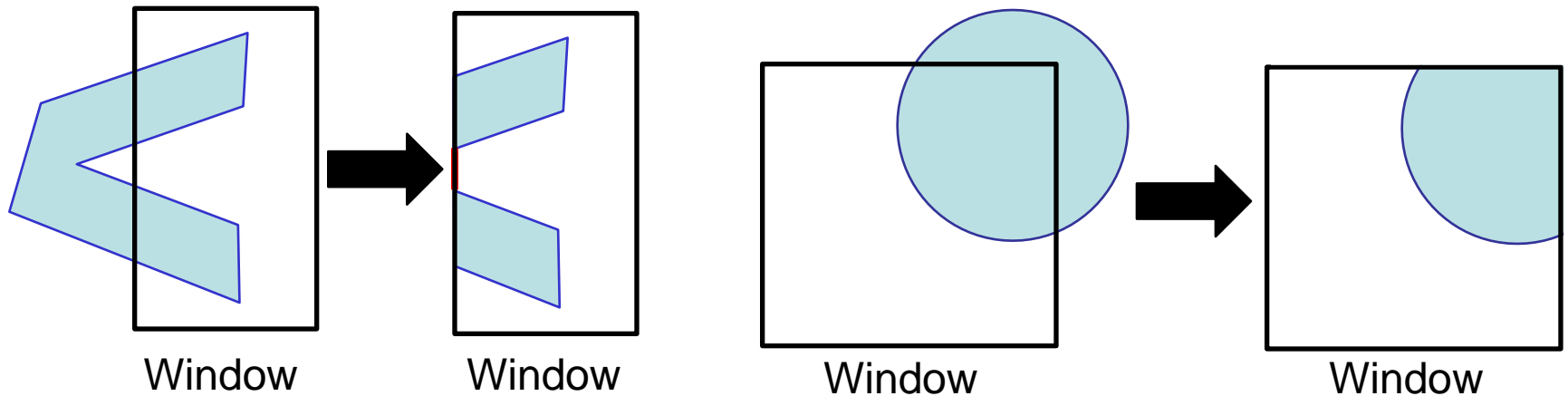
- How many “cached” crosses?



- Your geometry step must be able to *create* new polygons
 - Instead of 1-in-1-out

Other Area Clipping Concerns

- Clipping concave areas can be a little more tricky as often superfluous lines must be removed



- Clipping curves requires more work
 - For circles we must find the two intersection points on the window boundary

2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
- 5. Text Clipping**
6. Curve Clipping

Text Clipping

Text clipping relies on the concept of bounding rectangle

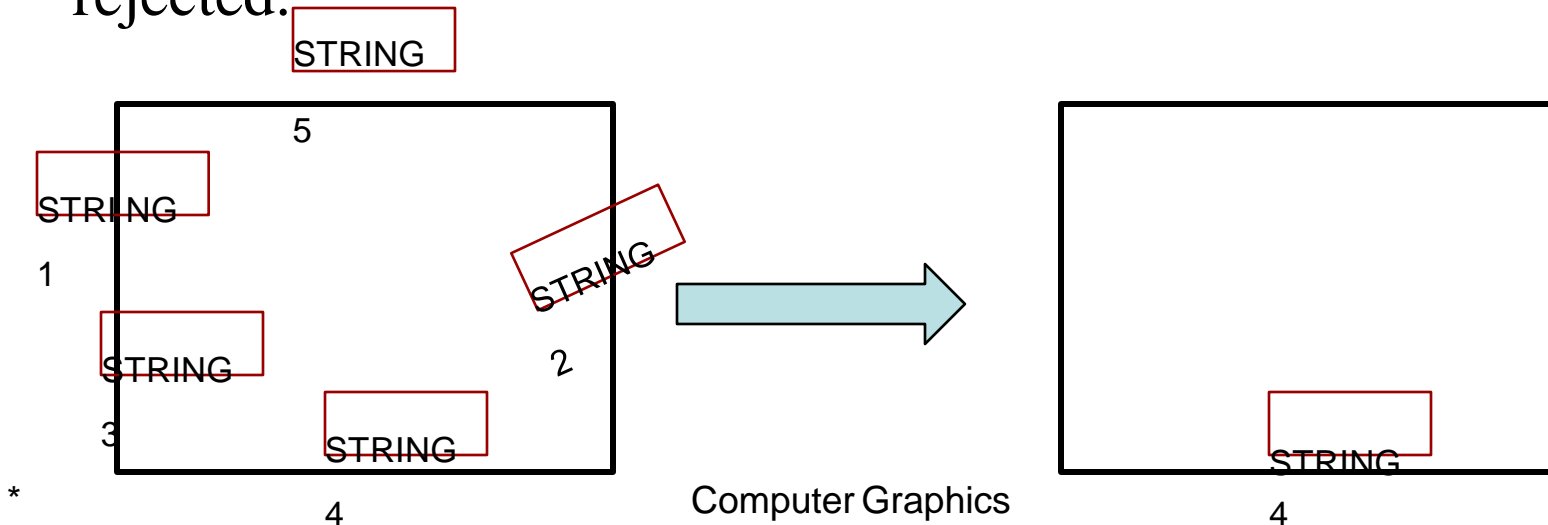
TYPES

1. All or None String Clipping
2. All or None Character Clipping
3. Component Character Clipping

Text Clipping

1. All or None String Clipping

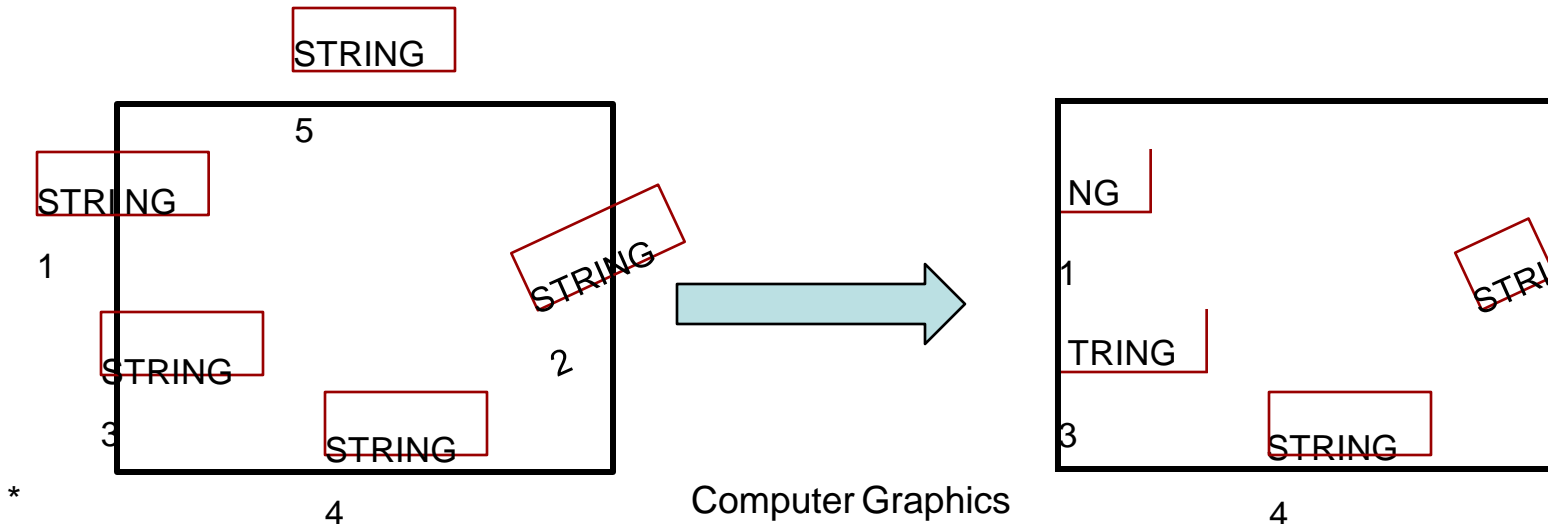
- In this scheme, if all of the string is inside window, we clip it, otherwise the string is discarded. **This is the fastest method.**
- The procedure is implemented by consider a *bounding rectangle* around the text pattern. The boundary positions are compared to the window boundaries. In case of overlapping the string is rejected.



Text Clipping

2. All or None Character Clipping

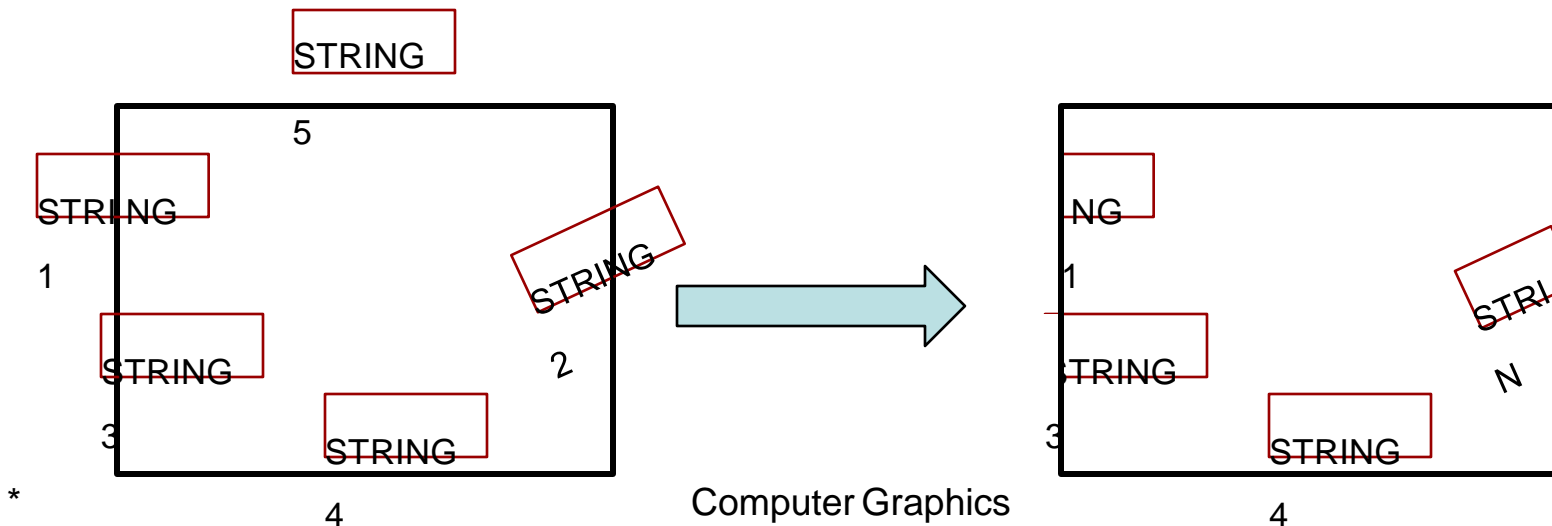
- In this scheme, we discard only those characters that are not completely inside window.
- Boundary limits of individual characters are compared against window. In case of overlapping the character is rejected.



Text Clipping

3. Component Character Clipping

- Characters are treated like graphic objects.
 - Bit Mapped Fonts : Point Clipping
 - Outlined Fonts : Line/Curve Clipping
- In case of overlapping the part of the character inside is displayed and the outside portion of the character is rejected.

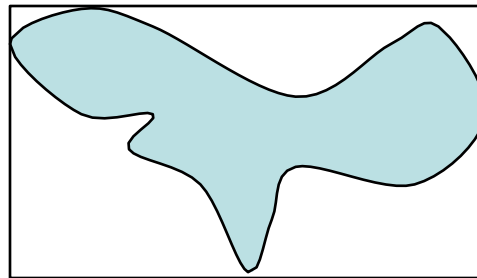


2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. **Curve Clipping**

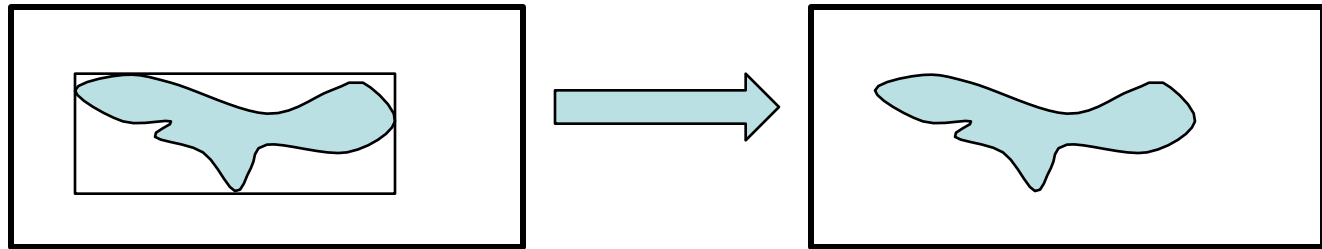
Curve Clipping

- Areas with curved boundaries can be clipped with methods similar to line and polygon clipping.
- Curve clipping requires more processing as it involve non linear equations.
- ***Bounding Rectangles*** are used to test for overlap with rectangular clip window.

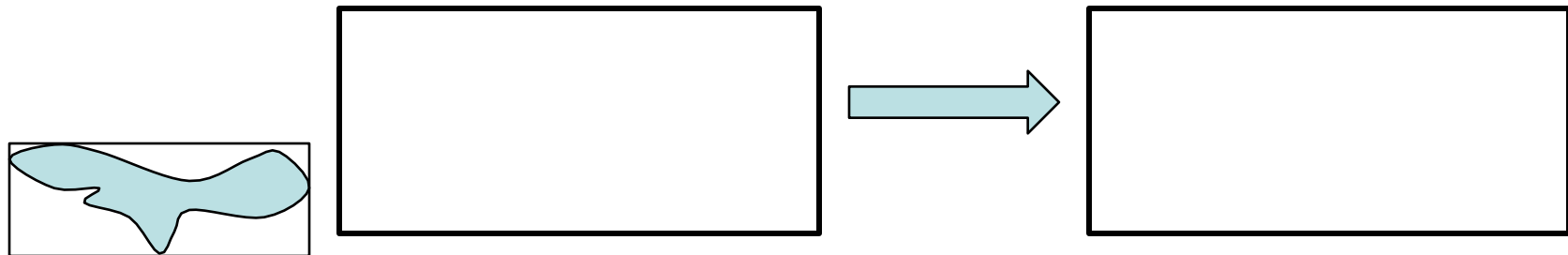


Curve Clipping

- If bounding rectangle is completely inside the object/curve is saved.



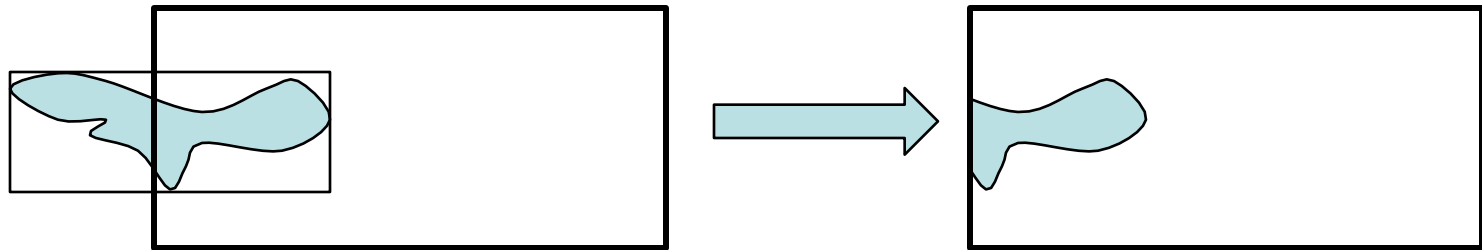
- If bounding rectangle is completely outside the object/curve is discarded.



*

Curve Clipping

- If both the above tests fails we use other computation saving approaches depending upon type of object
 - **Circle:** Use coordinate extent of individual quadrant, then octant if required.
 - **Ellipse:** Use coordinate extent of individual quadrant.
 - **Point:** Use point clipping



Any Question !