المملكة العربية السعودية
وزارة التعليم العالي
جامعة طيبة
قسم علوم الحاسب والمعلومات

الرسم بالحاسب
Computer Graphics

Chapter 1

What is Computer Graphics?

- Computer-generated images or sequences of images (i.e., animations, movies)
- The scientific study of techniques and methods for generating such images

Computer Graphics Applications

- Healthcare
- Education
- Building
- Army
- Business
- Geopgraphy
- Films
- Space
- TV
- Video games
- Simulation of natural phenomena
- …

Some Graphics Software Packages

- Early graphics libraries:
  - GKS (Graphical Kernel System)
  - PHIGS
- OpenGL (Silicon Graphics)
- Java2D (Sun Microsystems)
- Java3D (Sun Microsystems)
- VRML (Silicon Graphics)

Graphics: Main Components

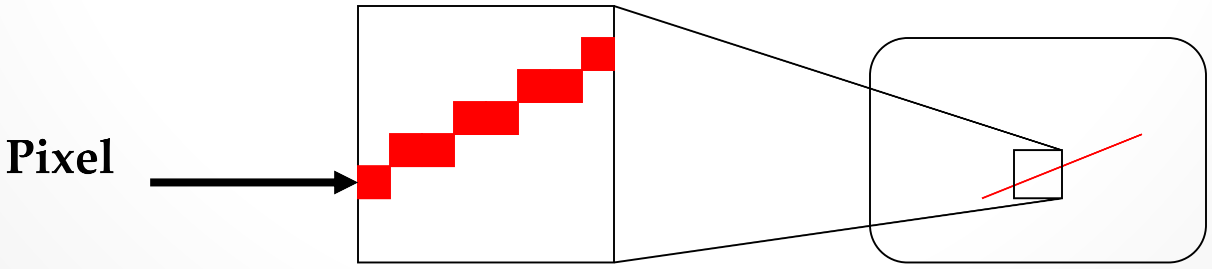
- Theory
  - Analytical Geometry
  - Vectors and Matrices
- Algorithms
  - Eg: Line drawing, Filling etc.
- Implementation
  - Programming (OpenGL)

Graphics Hardware

- Line Drawing Devices:
  - Eg. Pen Plotters
  - Advantages: Perfect lines, Sharp Diagrams.
  - Disadvantages: Not suitable for filled regions.

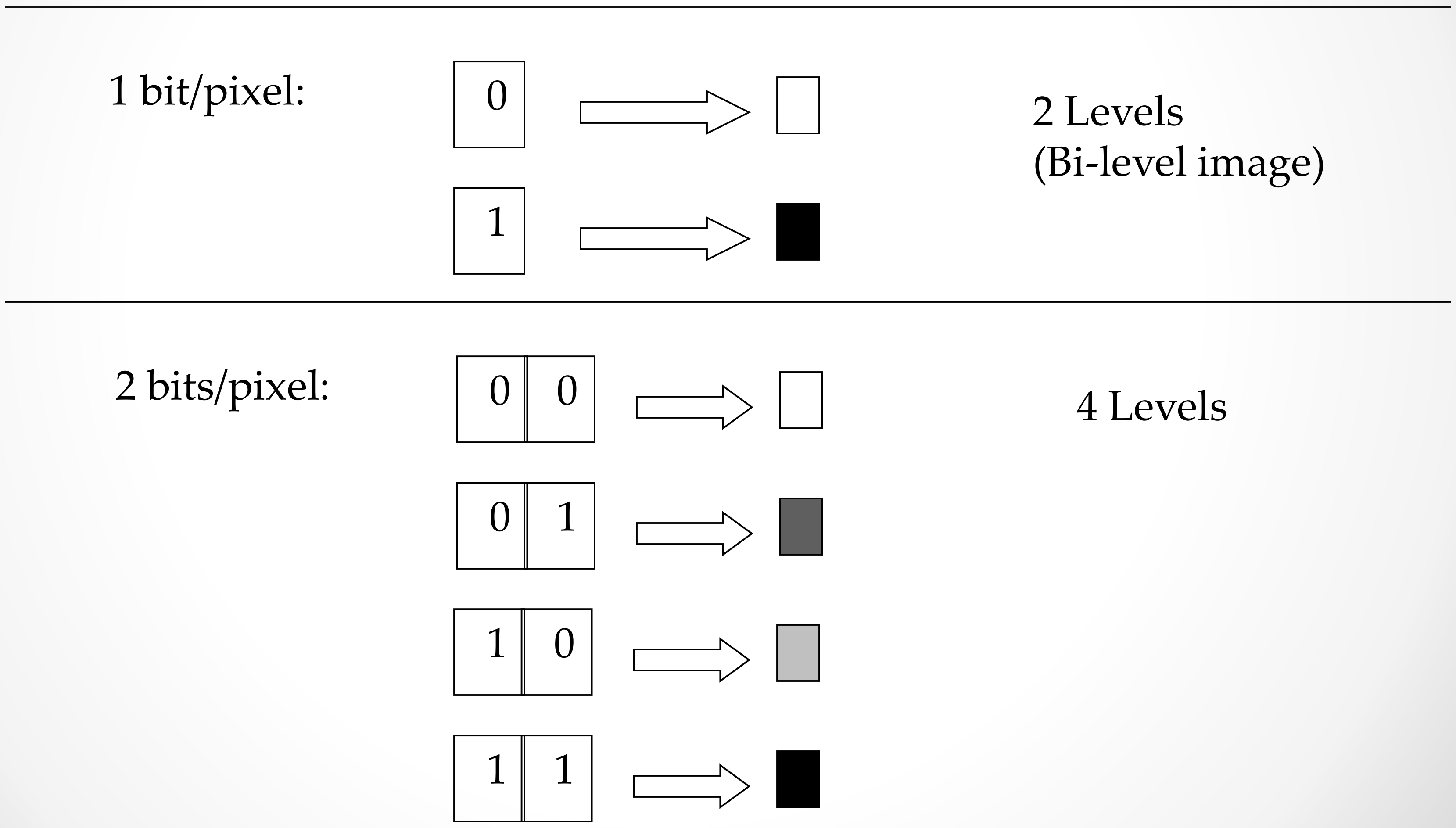
Graphics Hardware

- Raster Devices: Create pictures by displaying dots
  - Eg: Video monitor, dot-matrix printer, laser printer, ink-jet printer, film recorder
  - Advantages: Filled, shaded regions are easily displayed
  - Disadvantages: memory requirements,



Pixel Depth

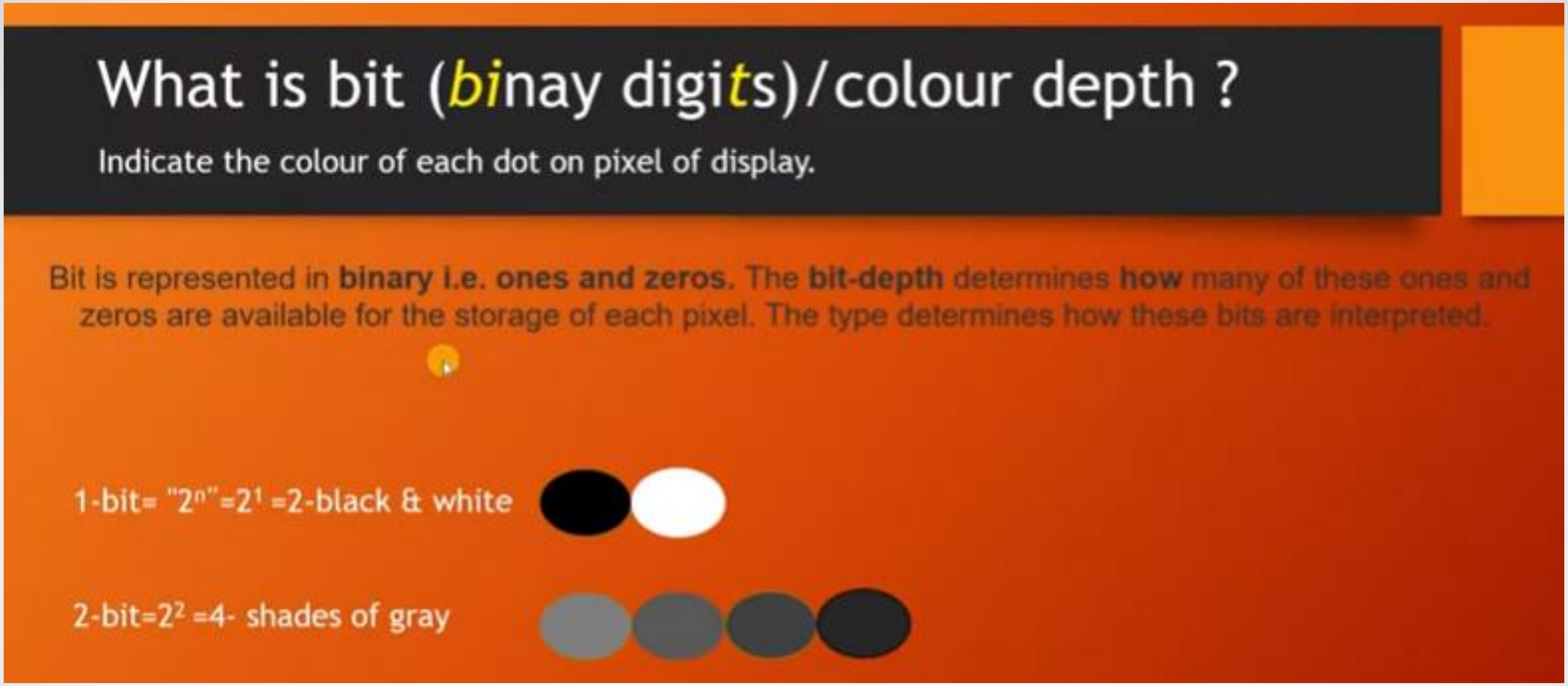
- Pixel depth is the number of bits used to represent a pixel value.



Pixel Depth

- 1 bit per pixel produce 2 levels (bi-level image).
- 2 bits per pixel produce 4 levels.
- 8 bits per pixel produce 256 levels.

- In general, if the pixel depth is n, then it is possible to have 2^n levels.



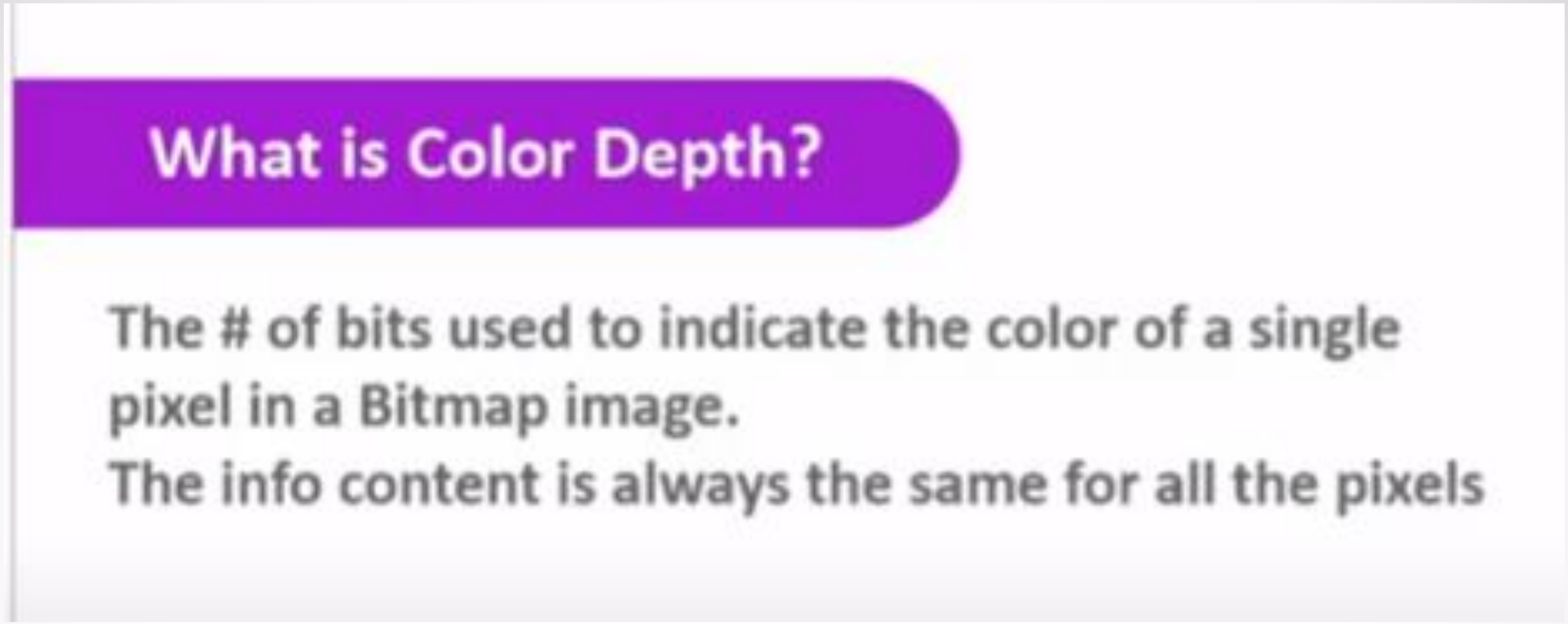
What is bit (*bi*nay digi*t*s)/colour depth ?

Indicate the colour of each dot on pixel of display.

Bit is represented in **binary i.e. ones and zeros.** The **bit-depth** determines **how** many of these ones and zeros are available for the storage of each pixel. The type determines how these bits are interpreted.

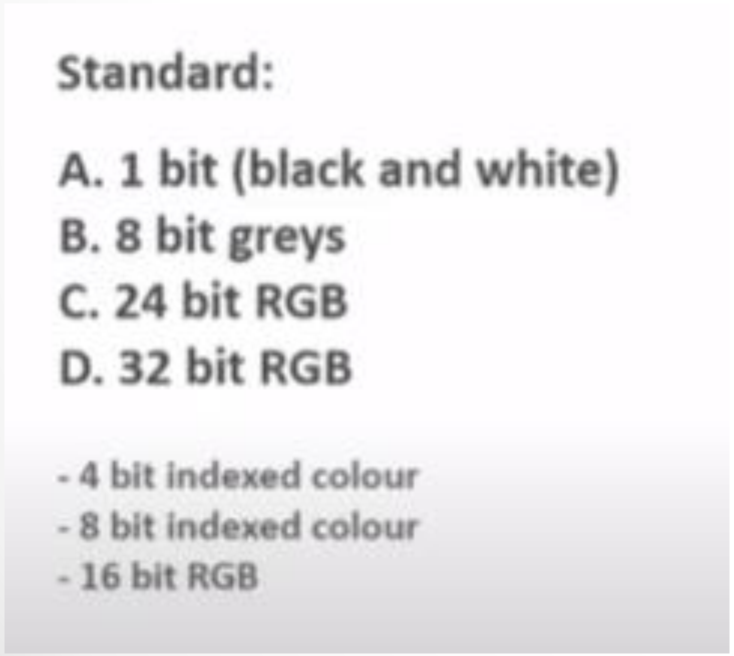
1-bit= "2^n"=2^1 =2-black & white

2-bit=2^2 =4- shades of gray



What is Color Depth?

The # of bits used to indicate the color of a single pixel in a Bitmap image.
The info content is always the same for all the pixels

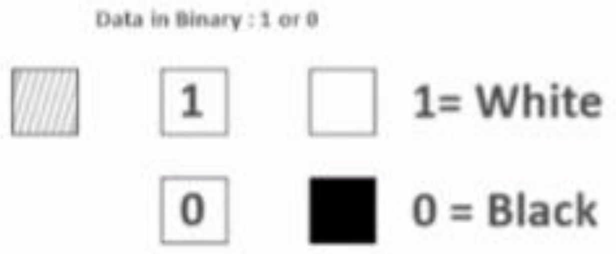


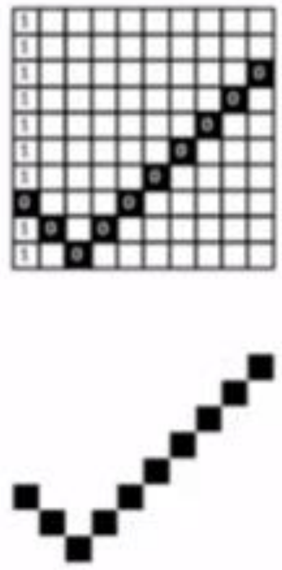
Standard:

A. 1 bit (black and white)
B. 8 bit greys
C. 24 bit RGB
D. 32 bit RGB

- 4 bit indexed colour
- 8 bit indexed colour
- 16 bit RGB

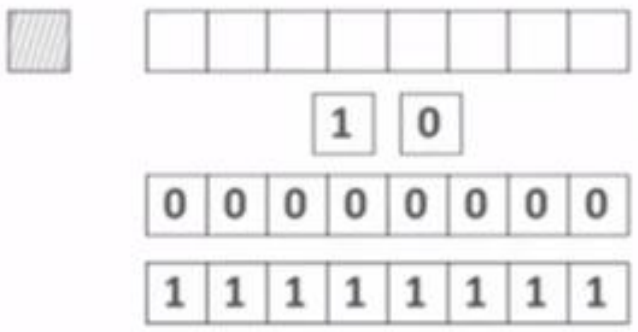
A. 1 bit (black and white)



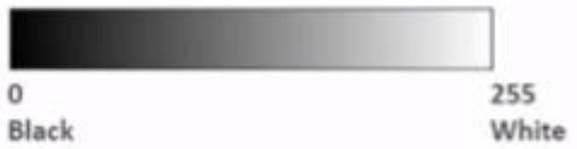


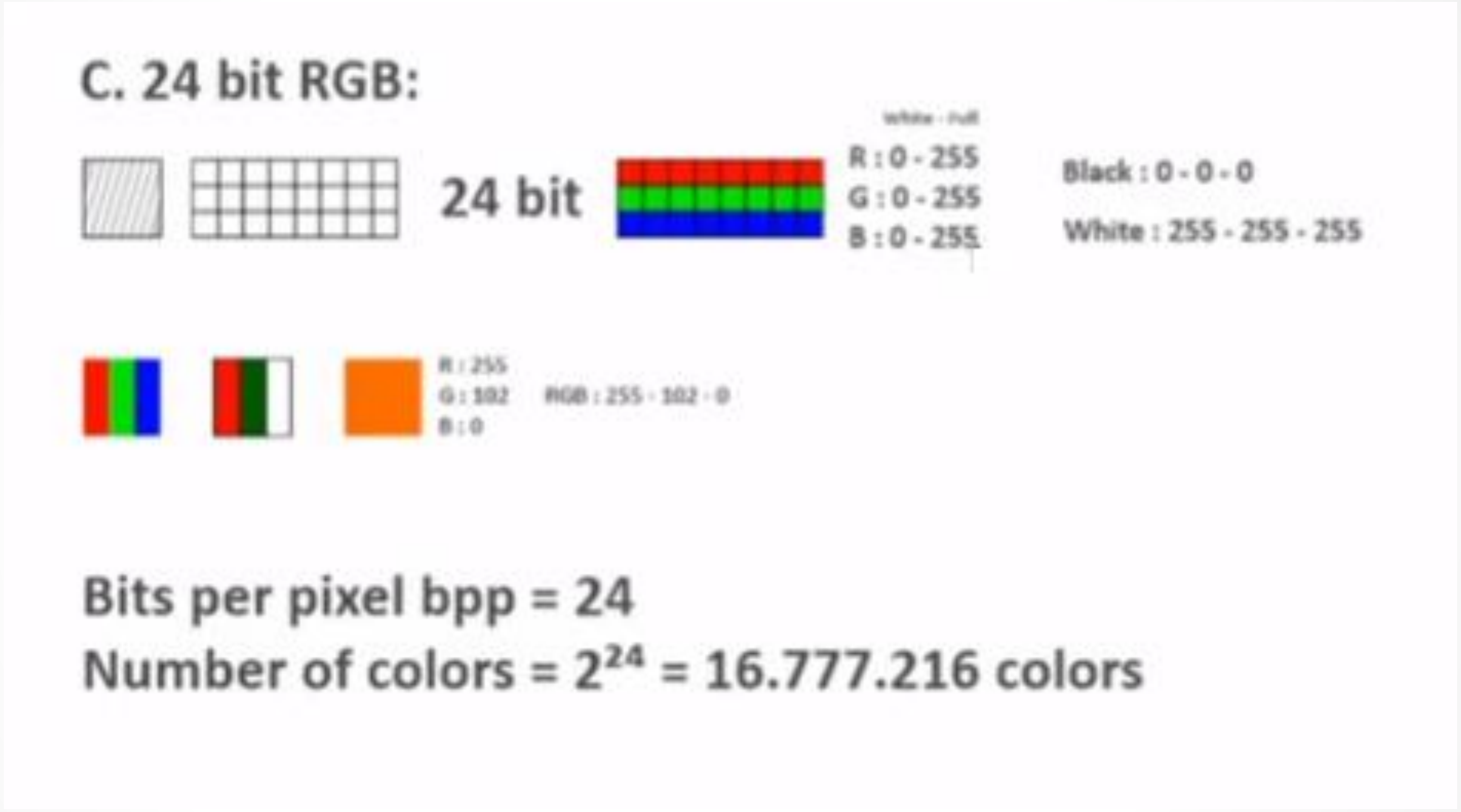
Bits per pixel bpp = 1
Number of colors = 2^1 = 2 colors

B. 8 bit greys



Bits per pixel bpp = 8
Number of colors = 2^8 = 256 colors





C. 24 bit RGB:

24 bit

white - null

R : 0 - 255

G : 0 - 255

B : 0 - 255

Black : 0 - 0 - 0

White : 255 - 255 - 255

R : 255

G : 102

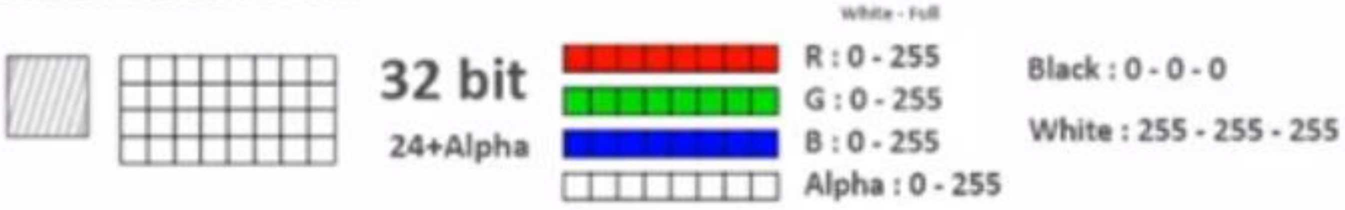
B : 0

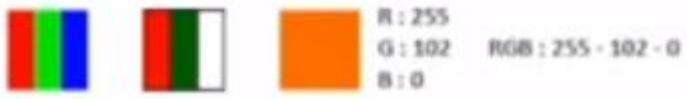
RGB : 255 - 102 - 0

Bits per pixel bpp = 24

Number of colors = 2^{24} = 16.777.216 colors

D. 32 bit RGB:





Alpha channel. This channel can be used to represent transparency.

Bits per pixel bpp = 32

Number of colors = 2^{32} = 4.294.967.296 colors

Raster Display

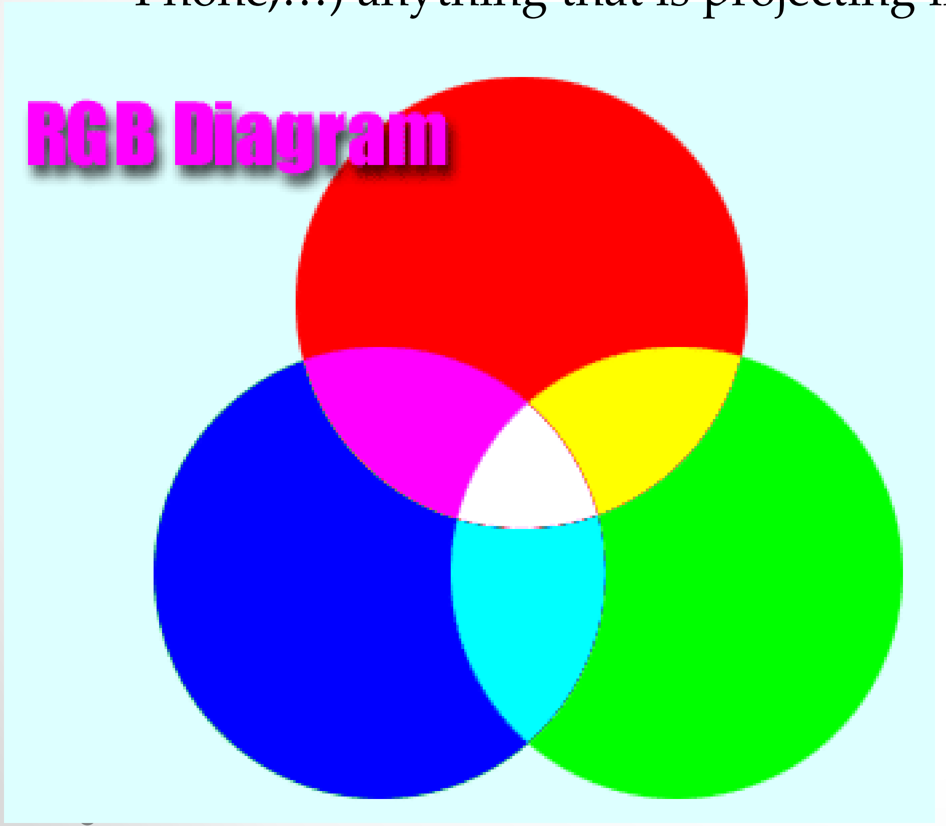
- Most display used for computer graphics nowadays are raster displays.
- Image presented in display surface that contains certain number of pixels. Eg. 480 x 640 .
- Frame buffer is a region of memory sufficiently large to hold all the pixel values for display.

Assignment- 1

- **HW: Search for Vector display and compare between raster and vector display?**

Color Models: RGB Diagram

RGB is called an additive model and used for anything that is to be displayed on a screen. Unlike CYMK, when we add the colors of RGB we get white.
It is using for any design that you will be viewing on a screen(laptop, desktop, TV, Phone,…) anything that is projecting light onto a screen.



<u>Basis colors: R, G, B</u>

R: Red=[1, 0, 0]

G: Green=[0, 1, 0]

B: Blue=[0, 0, 1]

C: Cyan=[0, 1, 1]

M: Magenta=[1, 0, 1]

Y:Yellow=[1, 1, 0]

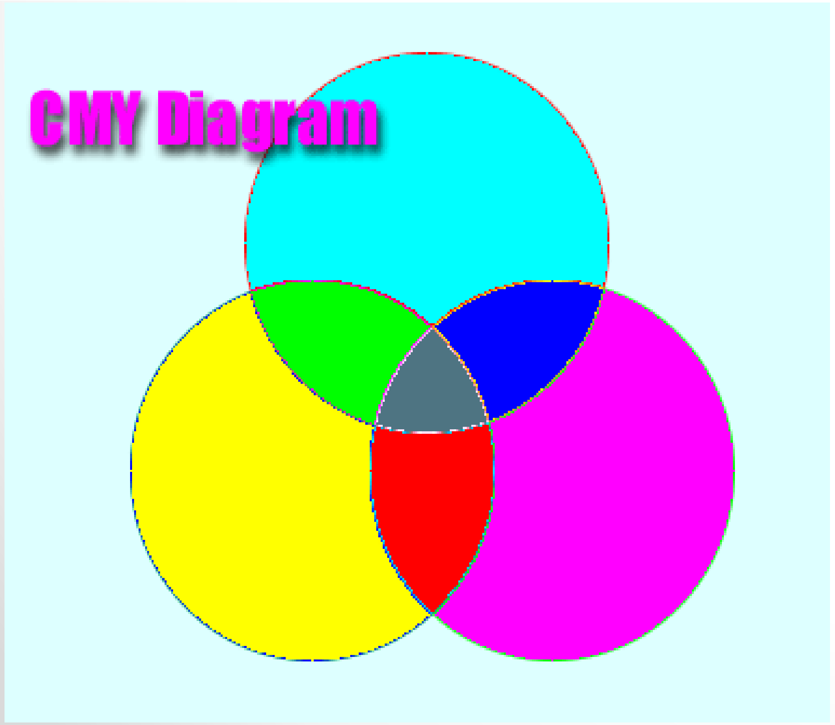
W:White=[1,1,1]

K: Black=[0,0,0]

Color Models: CMY Diagram

CMY is a subtractive color model . It is used for anything that is printed by ink.
It is called subtractive because we start with a white page and add ink to the page.
So, CYMK subtract, add the colors to get black

- Cyan, magneta & yellow are the secondary colors of light and primary colors of pigments



<u>Basis colors: C, M, Y</u>

C: Cyan=[1, 0, 0]

M: Magenta=[0, 1, 0]

Y: Yellow=[0, 0, 1]

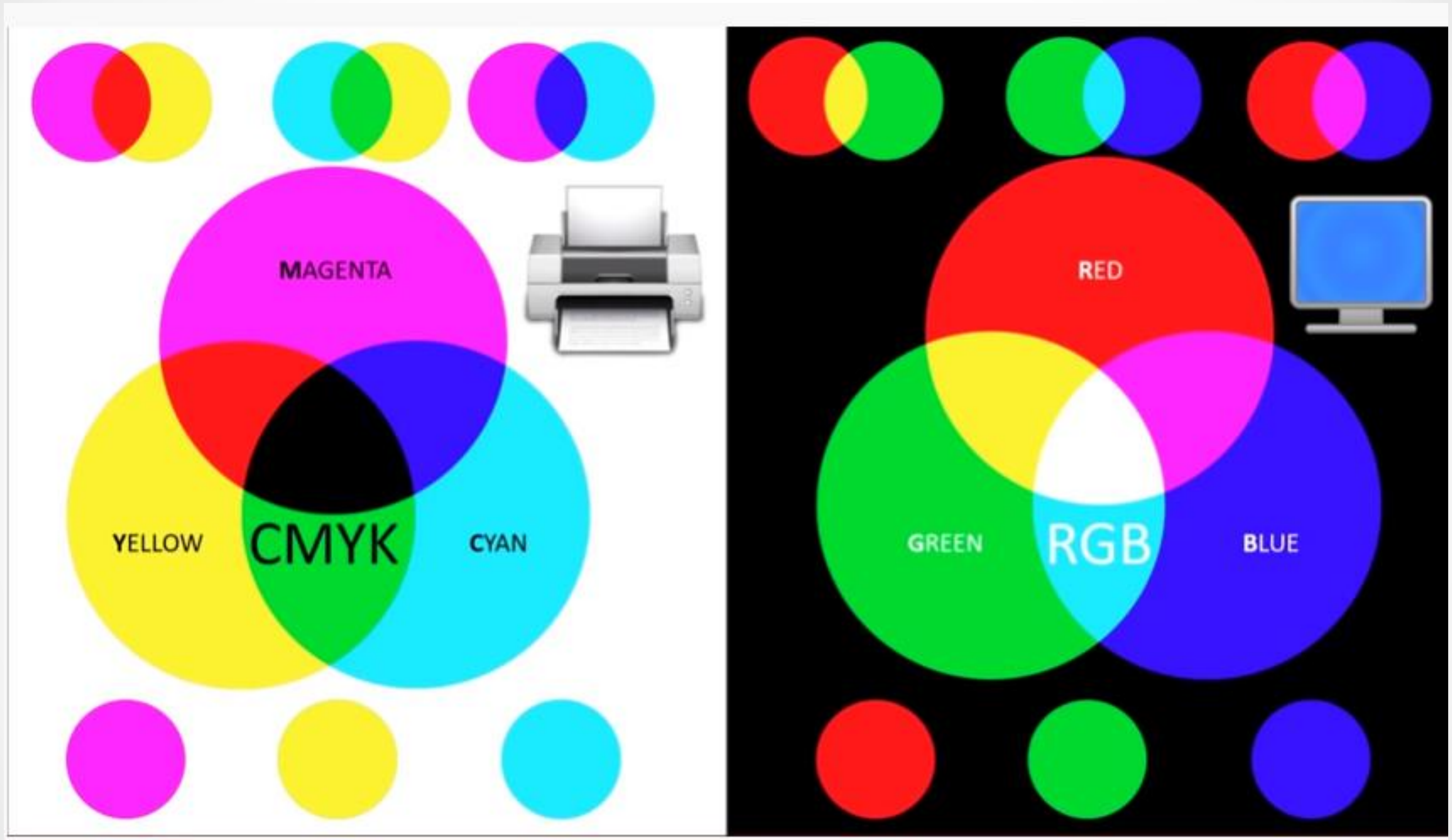
R: Red=[0, 1, 1]

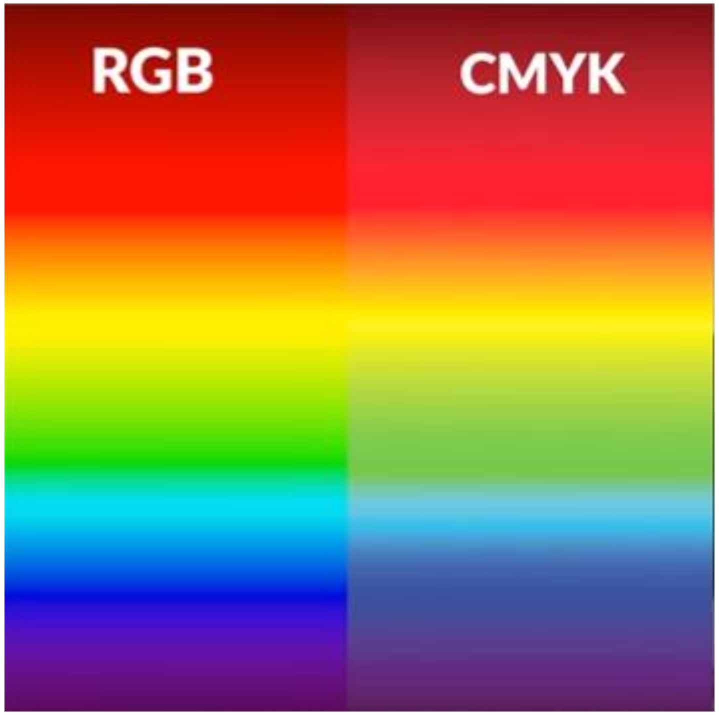
G: Green=[1, 0, 1]

B: Blue=[1, 1, 0]

W:White=[0,0,0]

K: Black=[1,1,1]


MAGENTA
YELLOW
CMYK
CYAN
RED
GREEN
RGB
BLUE


RGB
CMYK

Color Models: RGB <-> CMY

$$(r, g, b)_{\text{RGB}} = (1,1,1) - (c, m, y)_{\text{CMY}}$$

Red :

$(1, 0, 0)_{\text{RGB}} = (1,1,1) - (0, 1, 1)_{\text{CMY}}$

Green :

$(0, 1, 0)_{\text{RGB}} = (1,1,1) - (1, 0, 1)_{\text{CMY}}$

Blue :

$(0, 0, 1)_{\text{RGB}} = (1,1,1) - (1, 1, 0)_{\text{CMY}}$

Chapter 2
Introduction to OpenGL

2.1 What is OpenGL?

OpenGL has it origins in the earlier GL (“Graphics Library”) system which was invented by Silicon Graphics Inc. as the means for programming their high-performance specialised graphics workstations.

As time went on, people became interested in porting GL to other kinds of machine, and in 1992 a variation of GL – called OpenGL – was announced. Unlike GL, OpenGL was specifically designed to be **platform-independent**, so it would work across a whole range of computer hardware – not just Silicon Graphics machines. The combination of OpenGL’s power and portability led to its rapid acceptance as a **standard** for computer graphics programming.

OpenGL itself isn’t a programming language, or a software library. It’s the **specification** of an Application Programming Interface (API) for computer graphics programming. In other words, OpenGL defines a set of functions for doing computer graphics.

2.2 A whirlwind tour of OpenGL

What exactly can OpenGL do? Here are some of its main features:

- It provides 3D geometric objects, such as lines, polygons, triangle meshes, spheres, cubes, quadric surfaces, NURBS curves and surfaces;
- It provides 3D modelling transformations, and viewing functions to create views of 3D scenes using the idea of a **virtual camera**;
- It supports high-quality rendering of scenes, including hidden-surface removal, multiple light sources, material types, transparency, textures, blending, fog;

- It provides display lists for creating graphics caches and hierarchical models. It also supports the interactive “picking” of objects;
- It supports the manipulation of images as pixels, enabling frame-buffer effects such as antialiasing, motion blur, depth of field and soft shadows.

2.2.1 The support libraries: GLU and GLUT

- A key feature of the design of OpenGL is the separation of **interaction** (input and windowing functions) from **rendering**. OpenGL itself is concerned only with graphics rendering. You can always identify an OpenGL function: all OpenGL function names start with **“gl”**.

- Over time, two **utility libraries** have been developed which greatly extend the low-level (but very efficient) functionality of OpenGL. The first is the “OpenGL Utility Library”, or **GLU**. The second is the “OpenGL Utility Toolkit”, or **GLUT**:

- **GLU** provides functions for drawing more complex primitives than those of OpenGL, such as curves and surfaces, and also functions to help specify 3D views of scenes. All GLU function names start with **“glu”**.
- **GLUT** provides the facilities for interaction that OpenGL lacks. It provides functions for managing windows on the display screen, and handling input events from the mouse and keyboard. It provides some rudimentary tools for creating Graphical User Interfaces (GUIs). It also includes functions for conveniently drawing 3D objects like the platonic solids, and a teapot. All GLUT function names start with **“glut”**.

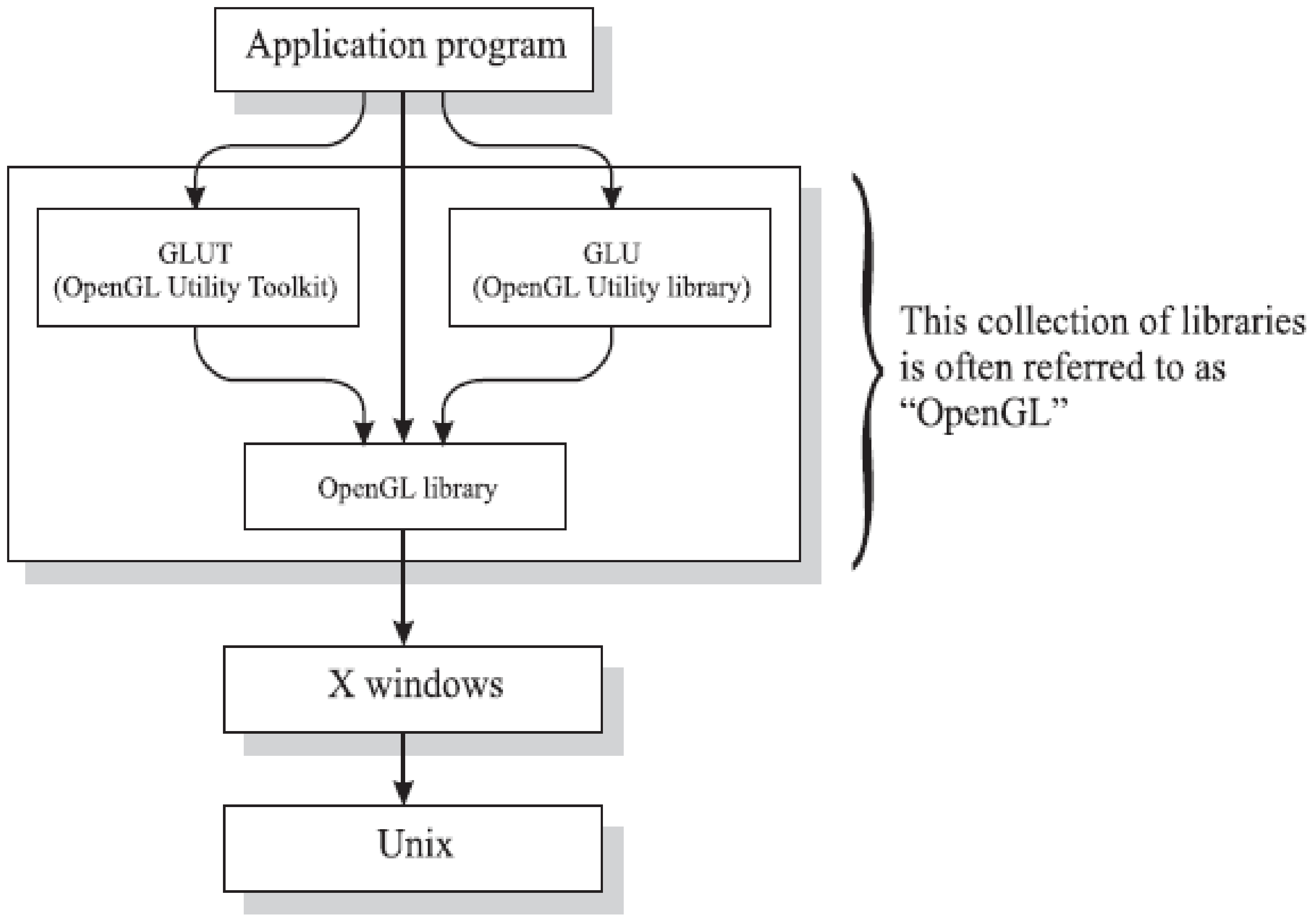


Figure 2.2: What is commonly called "OpenGL" is actually a set of three libraries: **OpenGL** itself, and the supporting libraries **GLU** and **GLUT**.

How to install GLUT?

- Download GLUT
  - http://www.opengl.org/resources/libraries/glut.html
- Copy the files to following folders:

glut.h	→	VC/include/gl/
glut32.lib	→	VC/lib/
glut32.dll	→	windows/system32/

- Header Files:
  - #include <GL/glut.h>
  - #include <GL/gl.h>
  - Include glut automatically includes other header files

Chapter 4
Beginning OpenGL programming

```
/* ex1.c */
#include <GL/glut.h>
void display (void) {
/* Called when OpenGL needs to update the display */
glClear (GL_COLOR_BUFFER_BIT); /* Clear the window
*/
glFlush(); /* Force update of screen */
{
int main (int argc, char **argv) {
glutInit (&argc, argv); /* Initialise OpenGL */
glutCreateWindow ("ex1"); /* Create the window */
glutDisplayFunc (display); /* Register the "display"
function */
glutMainLoop (); /* Enter the OpenGL main loop */
return 0}
```

The program begins with
#include <GL/glut.h>

All OpenGL programs must start with this line, which accesses all the OpenGL include files: it pulls in all the function prototypes and other definitions used by OpenGL. Miss it out,will flatly refuse to compile your program.
ex1.c contains two functions: display(), and main(). The execution of all C programs starts at main(), so we'll start there too.

Entering main()

- The first six lines use the OpenGL Utility Toolkit to configure and open window for us.
- **glutInit()**, initializes the GLUT library. It processes the command-line arguments provided to the program, and removes any that control how GLUT might operate (such as specifying the size of a window).
- **glutInit()** needs to be the first GLUT function that your application calls, as it sets up data structures required by subsequent GLUT routines.

Entering main() Cont.

- **glutInitDisplayMode()** configures the type of window we want to use with our application. In this case, we only request that the window use the RGBA color space
- more OpenGL features, such as depth buffers, or to enable animation.

Entering main() Cont.

- **glutInitWindowSize()** specifies the size of the window.
- **glutInitContextVersion()** and **glutInitContextProfile()** specify the type of OpenGL *context (vresion)*---OpenGL's internal data structure for keeping track of state settings and operations-
- Here, we request an OpenGL Version 4.3 *core*
- OpenGL versions all the way back to OpenGL Version 1.0.

Entering main() Cont.

- **glutCreateWindow()**, which does just what it

says.

- Only after GLUT has created a window for you can you use OpenGL functions.

Entering main() Cont.

- **glewInit()** initializes another help library we use: GLEW---the OpenGL Extension Wrangler. GLEW simplifies
- a considerable amount of additional work is required to get an application going.

Entering main() Cont.

- The **init()** initializes all of our relevant OpenGL data so we can use for rendering later.
- **glutDisplayFunc()**, sets up the *display callback*, which is the routine GLUT will call when it thinks the contents of the window need to be updated. Here, we provide the GLUT library a pointer to a function: **display()**, which we'll also discuss soon. GLUT uses a number of callback

Entering main() Cont.

- **glutMainLoop()**, which is an infinite loop
- that works with the window and operating systems to process user input
- Since **glutMainLoop()** is an infinite loop, any commands placed after it aren’t executed.

- void **glClear** (GLbitfield *mask*);
- **glClear()** clears one or more of OpenGL’s buffers, specified by mask. In this manual, we’ll only be concerned with one buffer, the **frame buffer**, which holds the pixels which will be copied to the window. This has the special name GL COLOR BUFFER BIT. When **glClear()** is called, each pixel
- in the buffer is set to the **current clear colour**, which is set to black by default. You set the current clear colour using the function **glClearColor()**

- void **glFlush** (void);
- The purpose of this function is to instruct OpenGL to make sure the screen is up to date – it causes the contents of any internal OpenGL buffers are “flushed” to the screen.

```
/* ex1.c */
#include <GL/glut.h>
void display (void) {
/* Called when OpenGL needs to update the display */
glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
glFlush(); /* Force update of screen */
}
int main (int argc, char **argv) {
glutInit (&argc, argv); /* Initialise OpenGL */
glutCreateWindow ("ex1"); /* Create the window */
glutDisplayFunc (display); /* Register the "display" function
*/
glutMainLoop (); /* Enter the OpenGL main loop */
return 0;
}
/* end of ex1.c */
/* ex2.c */
#include <GL/glut.h>
#include <stdio.h>
void display (void) {
/* Called when OpenGL needs to update the display */
glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
glFlush(); /* Force update of screen */
}
void keyboard (unsigned char key, int x, int y) {
/* Called when a key is pressed */
if (key == 27) exit (0); /* 27 is the Escape key */
else printf ("You pressed %c\n", key);
}
int main(int argc, char **argv) {
glutInit (&argc, argv); /* Initialise OpenGL */
glutCreateWindow ("ex2"); /* Create the window */
glutDisplayFunc (display); /* Register the "display" function */
glutKeyboardFunc (keyboard); /* Register the "keyboard" function */
glutMainLoop (); /* Enter the OpenGL main loop */
return 0;
}
/*end of ex2.c */
```

void **glutKeyboardFunc** (*void (*func)(unsigned char key, int x, int y)*);

- **glutKeyboardFunc()** registers the application function to call when OpenGL detects a key press generating an ASCII character. This can only occur when the mouse focus is inside the OpenGL window.
- Three values are passed to the callback function: key is the ASCII code of the key pressed; x and y give the pixel position of the mouse at the time.

Example 3: customizing the window

```
#include <GL/glut.h>
void display (void) {
/* Called when OpenGL needs to update the display */
glClearColor (1.0,1.0,1.0,0.0);
glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
glFlush(); /* Force update of screen */
}
void keyboard (unsigned char key, int x, int y) {
/* Called when a key is pressed */
if (key == 27) exit (0); /* 27 is the Escape key */
}
int main(int argc, char **argv) {
glutInit (&argc, argv); /* Initialise OpenGL */
glutInitWindowSize (500, 500); /* Set the window size */
glutInitWindowPosition (100, 100); /* Set the window position */
glutCreateWindow ("ex3"); /* Create the window */
glutDisplayFunc (display); /* Register the "display" function */
glutKeyboardFunc (keyboard); /* Register the "keyboard" function */
glutMainLoop (); /* Enter the OpenGL main loop */
return 0;
}
```

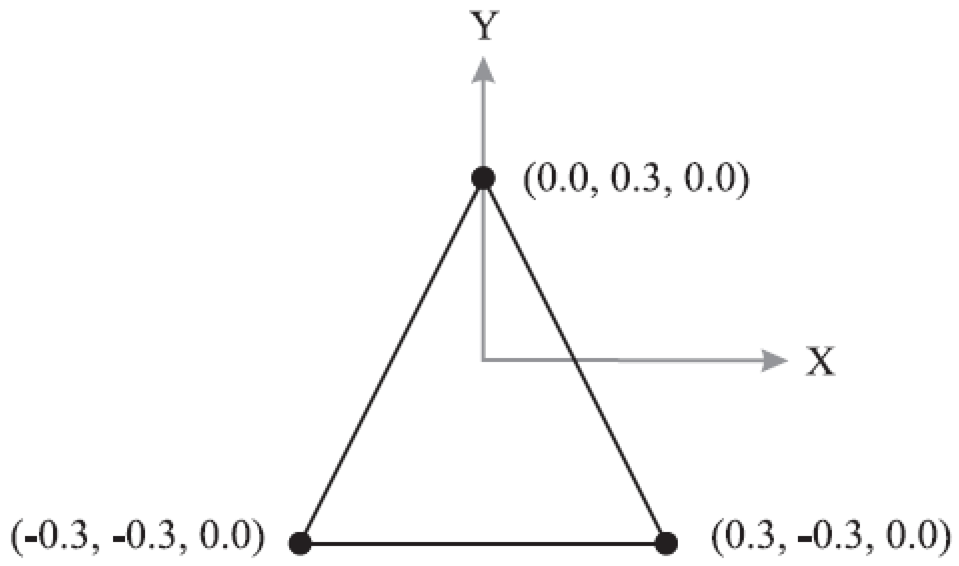
void **glutInitWindowSize** (int *width*, int *height*);

- **glutInitWindowSize()** sets the value of GLUT’s **initial window size** to the size specified by width and height, measured in pixels.

void **glutInitWindowPosition** (int *x*, int *y*);

- x and y give the position of the top left corner of the window measured in pixels from the **top left corner** of the X display.

Example 4: drawing a 2D triangle



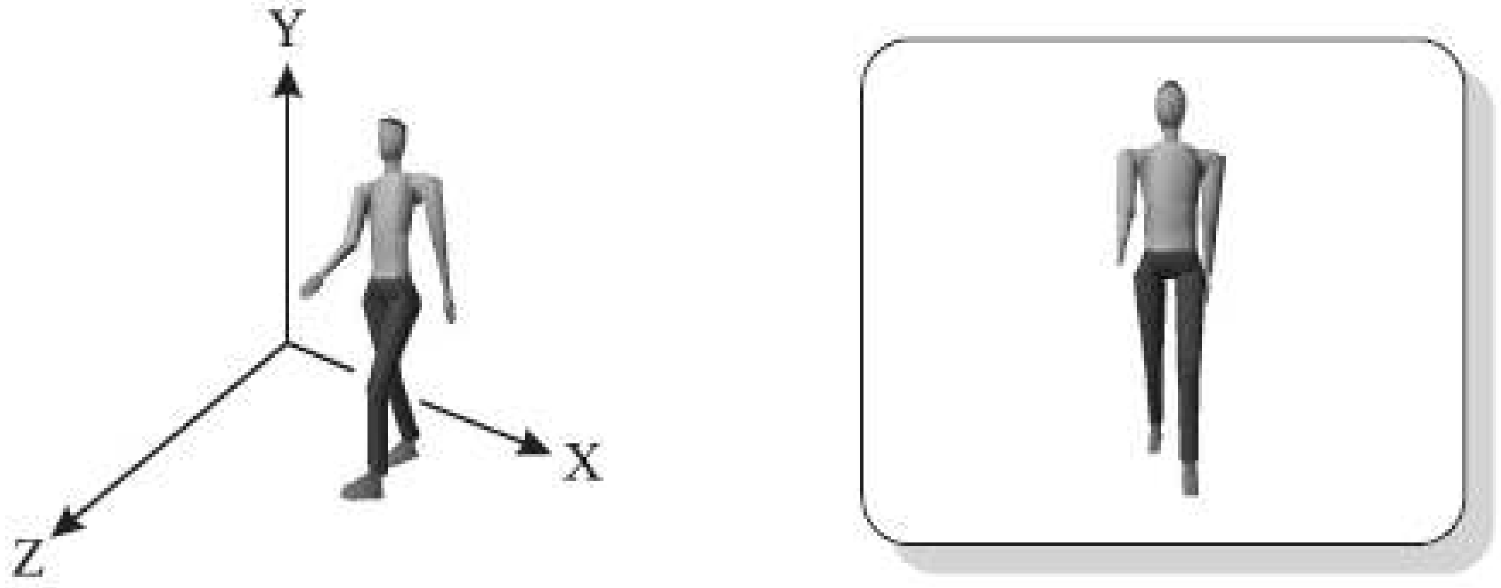
```
/* ex4.c */
#include <GL/glut.h>
void display (void) {
/* Called when OpenGL needs to update the display */
glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
glLoadIdentity ();
gluLookAt (0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glBegin (GL_LINE_LOOP); /* Draw a triangle */
glVertex3f(-0.3, -0.3, 0.0);
glVertex3f(0.0, 0.3, 0.0);
glVertex3f(0.3, -0.3, 0.0);
glEnd();
glFlush(); /* Force update of screen */
}
void keyboard (unsigned char key, int x, int y) {
/* Called when a key is pressed */
if (key == 27) exit (0); /* 27 is the Escape key */
}
```

```
void reshape (int width, int height)
{ /* Called when the window is created, moved or resized */
glViewport (0, 0, (GLsizei) width, (GLsizei) height);
glMatrixMode (GL_PROJECTION); /* Select the projection matrix */
glLoadIdentity (); /* Initialise it */
glOrtho(-1.0,1.0, -1.0,1.0, -1.0,1.0); /* The unit cube */
glMatrixMode (GL_MODELVIEW); /* Select the modelview matrix
*/}
int main(int argc, char **argv) {
glutInit (&argc, argv); /* Initialise OpenGL */
glutInitWindowSize (500, 500); /* Set the window size */
glutInitWindowPosition (100, 100); /* Set the window position */
glutCreateWindow ("ex4"); /* Create the window */
glutDisplayFunc (display); /* Register the "display" function */
glutReshapeFunc (reshape); /* Register the "reshape" function */
glutKeyboardFunc (keyboard); // Register the "keyboard" function
glutMainLoop (); /* Enter the OpenGL main loop */
return 0;}
```

Viewing using the camera

The idea of creating a 2D view of a 3D scene is simple: we “take a picture” of the scene using a **camera**, and display the camera’s picture in the window on the display screen. For convenience, OpenGL splits the process into three separate steps:

- **Step one**: First, we specify the position and orientation of the camera, using the function **gluLookAt()**;



OpenGL's 3D graphics "world" — The flat display screen

Figure 5.2: OpenGL's 3D "world", and the 2D display screen.

- **Step two**: Second, we decide what kind of projection we’d like the camera to create. We can choose an **orthographic** projection (also known as a **parallel projection**) using the function **glOrtho()** (page 56); or a **perspective** projection using the function **gluPerspective()** (page 56);
- **Step three**: Finally, we specify the size and shape of the camera’s image we wish to see in the window, using **glViewport()** (page 58). This last step is optional – by default the camera’s image is displayed using the whole window.

In OpenGL, the camera model described above is always active – you can’t switch it off. It’s implemented using **transformation matrices**, here’s a brief description of the process.

OpenGL keeps two transformation matrices:

the **modelview** matrix, M

The modelview matrix holds a transformation which composes the scene in world coordinates, and then takes a view of the scene using the camera (step one, above).

the **projection matrix**, P

The projection matrix applies the camera projection (step two, above).

Whenever the application program specifies a coordinate c for drawing, OpenGL transforms the coordinate in two stages, as follows, to give a new coordinate c′. First it transforms the coordinate c by the matrix M, and then by the matrix P, as follows:

$$c' = P \cdot M \cdot c$$

When an OpenGL application starts up, P and M are unit matrices – they apply the **identity transformation** to coordinates, which has no effect on the coordinates. It’s **entirely up to the application** to ensure that theM and P matrices always have suitable values. Normally, an application will set M in its display() function, and P in its reshape() function, as we shall now describe.

void **glutReshapeFunc** (*void)(*func)(int width, int height)* ;

- **glutReshapeFunc()** registers the application callback to call when the window is first created, and also if the window manager subsequently informs OpenGL that the user has reshaped the window.
- The new height and width of the window, in pixels, are passed to the callback. Typically, the callback will use these values to define the way that OpenGL's virtual camera projects its image onto the window,

- **glViewport()**, which specifies a rectangular portion of the window in which to display the camera’s image. As in this example, it’s common to use the whole of the window, so we set the viewport to be a rectangle of equal dimensions to the window.

- **glMatrixMode()** (page 47) selects which matrix subsequent functions will affect – in this case we select the projection matrix (P).
- Then we initialise it to the unit transformation with **glLoadIdentity()** (page 48). This is very important,
- Then, we select the orthographic projection using **glOrtho()** . The projection we've chosen maps a unit cube, centred on the origin, onto the viewport.

Section – III:

TRANSFORMATIONS

in 2-D

2D TRANSFORMATIONS AND MATRICES

Representation of Points:

2 x 1 matrix: $\begin{bmatrix} X \\ Y \end{bmatrix}$

General Problem: [B] = [T] [A]

[T] represents a generic operator to be applied to the points in A. T is the geometric transformation matrix.

If A & T are known, the transformed points are obtained by calculating B.

General Transformation of 2D points:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$x' = ax + cy$$

$$y' = bx + dy$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix}^T = \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$x' = ax + cy$$

$$y' = bx + dy$$

Solid body transformations – the above equation is valid for all set of points and lines of the object being transformed.

Special cases of 2D Transformations:

1) T = identity matrix:
a=d=1, b=c=0 => x'=x, y'=y

2) *Scaling & Reflections*:
b=0, c=0 => x' = a.x, y' = d.y;
This is scaling by a in x, d in y.

If, a = d > 1, we have enlargement;
If, 0 < a = d < 1, we have compression;

If a = d, we have uniform scaling,
else non-uniform scaling.

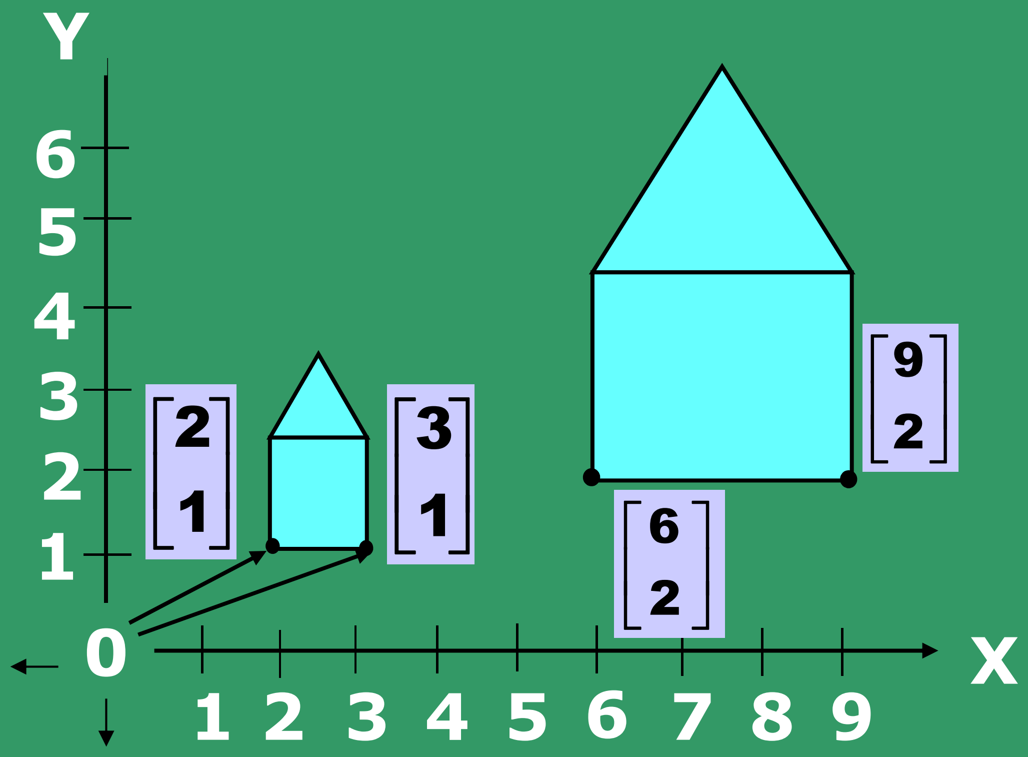
Scale matrix: let S_x = a, S_y = d:

$$\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

Example of Scaling

$S_x = 3$

$S_y = 2$



Y
X
0
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6
$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$
$\begin{bmatrix} 3 \\ 1 \end{bmatrix}$
$\begin{bmatrix} 6 \\ 2 \end{bmatrix}$
$\begin{bmatrix} 9 \\ 2 \end{bmatrix}$

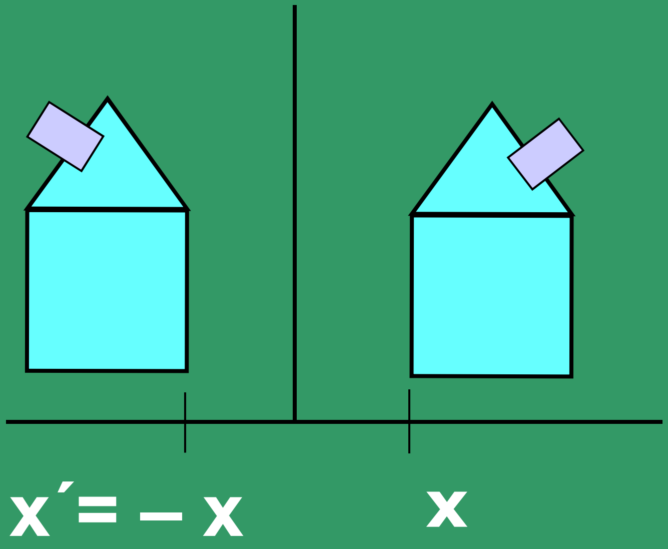
What if S_x and/ or S_y < 0 (are negative)?
Get reflections through an axis or plane.

Only diagonal terms are involved in scaling and reflections.

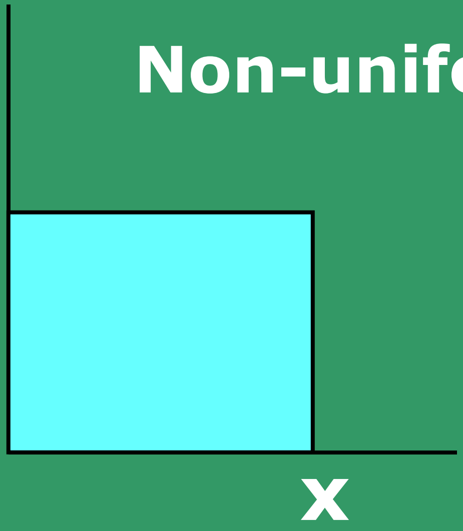
Note : House shifts position relative to origin

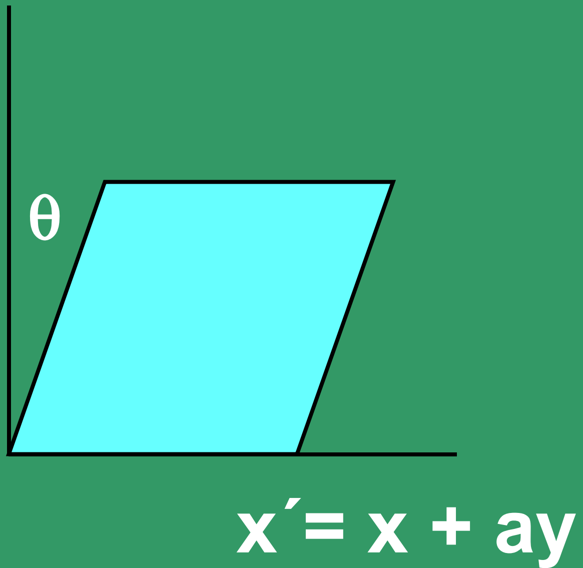
More examples of Scaling and reflection

Reflection
(about the Y-axis)



Non-uniform scaling





Special cases of Reflections (|T| = -1)

Matrix T	Reflection about
$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Y=0 Axis (or X-axis)
$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	X=0 Axis (or Y-axis)
$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	Y = X Axis
$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	Y = -X Axis

Off diagonal terms are involved in SHEARING;

a = d = 1;

let, c = 0, b = 2

x' = x
y' = 2x + y ;

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$x' = ax + cy$$

$$y' = bx + dy$$

y' depends linearly on x ; This effect is called shear.

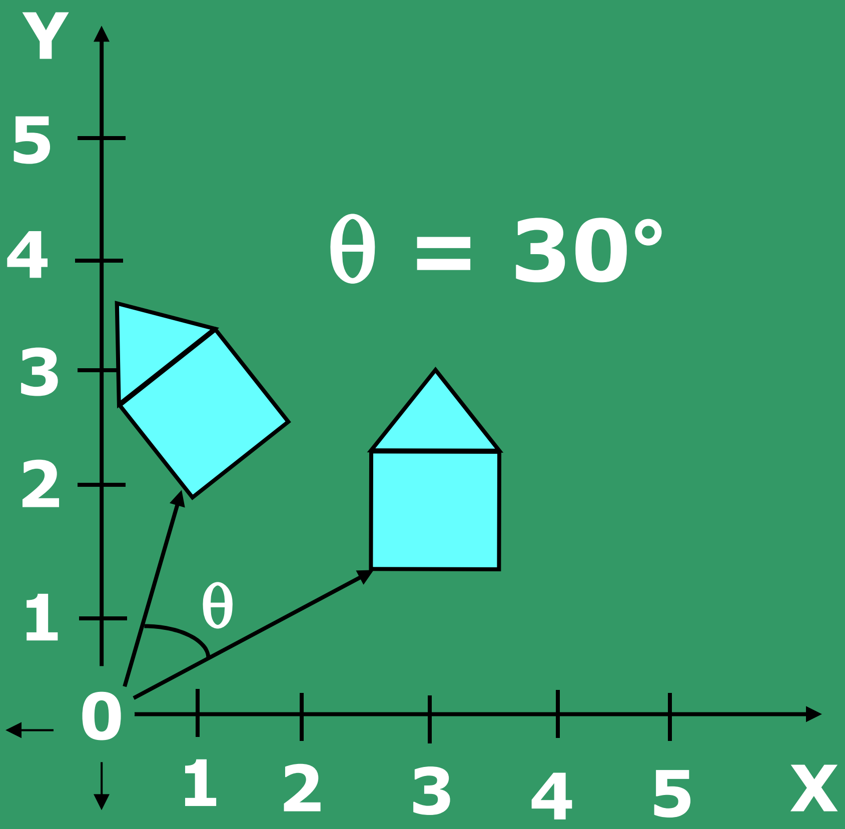
Similarly for b=0, c not equal to zero. The shear in this case is proportional to y-coordinate.

ROTATION

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

In matrix form, this is :

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$



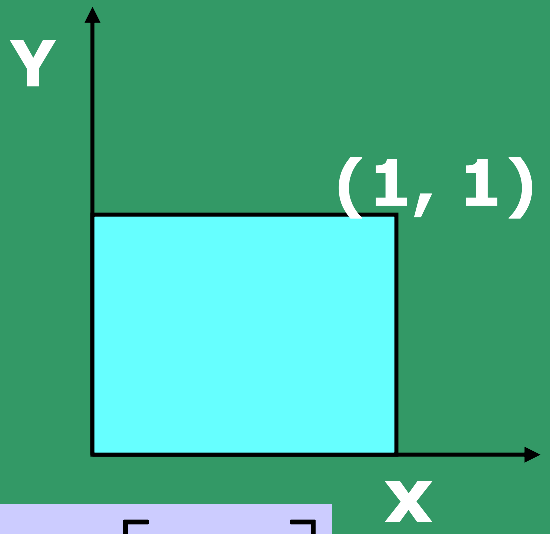
Positive Rotations: counter clockwise about the origin

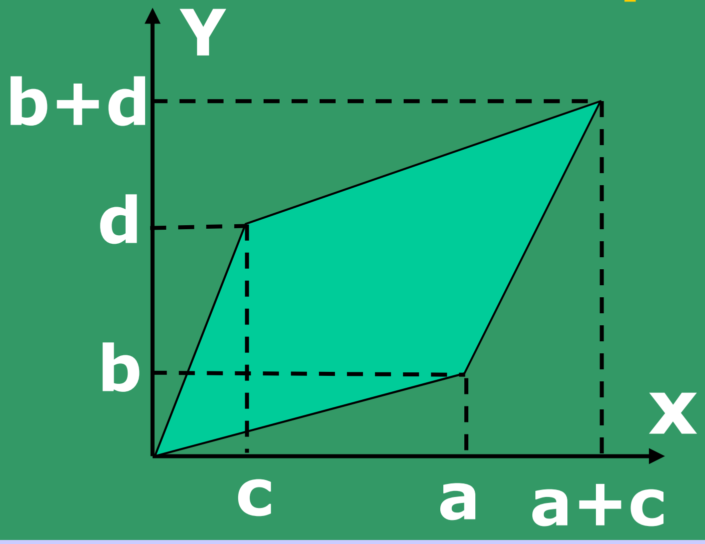
For rotations, $|T| = 1$ and $[T]^T = [T]^{-1}$. Rotation matrices are orthogonal.

Special cases of Rotations

θ (in degrees)	Matrix T
90	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
180	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
270 or -90	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
360 or 0	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Example - Transformation of a Unit Square





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

$$S' = S\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ a & b \\ a+c & b+d \\ c & d \end{bmatrix}$$

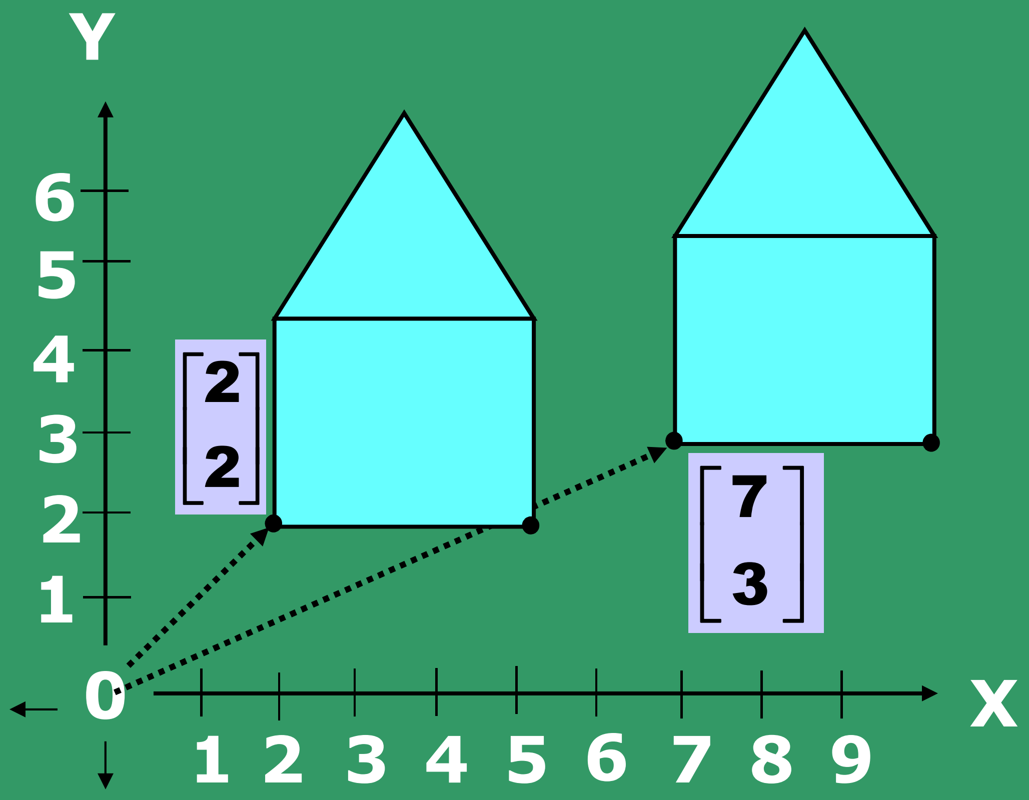
Area of the unit square after transformation

Extend this idea for any arbitrary area.

Translations

$t_x = 5$

$t_y = 1$



Y
X
0
1
2
3
4
5
6
7
8
9
$\begin{bmatrix}2\\2\end{bmatrix}$
$\begin{bmatrix}7\\3\end{bmatrix}$

Translations

$B = A + T_d$, where $T_d = [t_x \ t_y]^T$

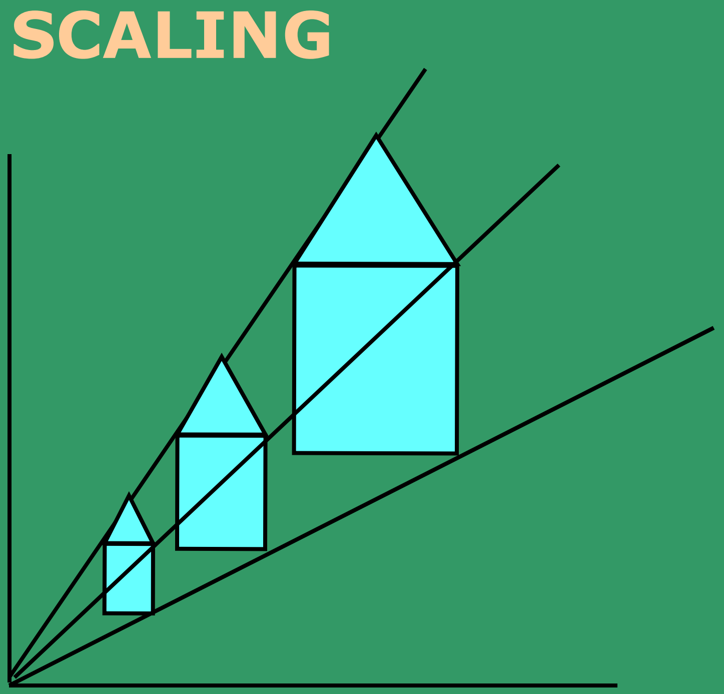
Where else are translations introduced?

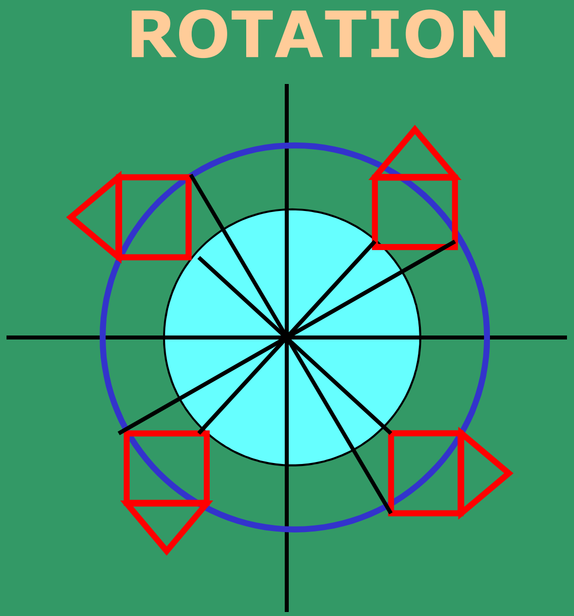
1) Rotations - when objects are not centered at the origin.

2) Scaling - when objects/lines are not centered at the origin - if line intersects the origin, no translation.

Origin is invariant to Scaling, reflection and Shear – not translation.

Note: we cannot directly represent translations as matrix multiplication, as we can for:





Can we represent translations in our general transformation matrix?

Yes, by using homogeneous coordinates

HOMOGENEOUS COORDINATES

Use a 3 x 3 matrix:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & c & t_x \\ b & d & t_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

We have:

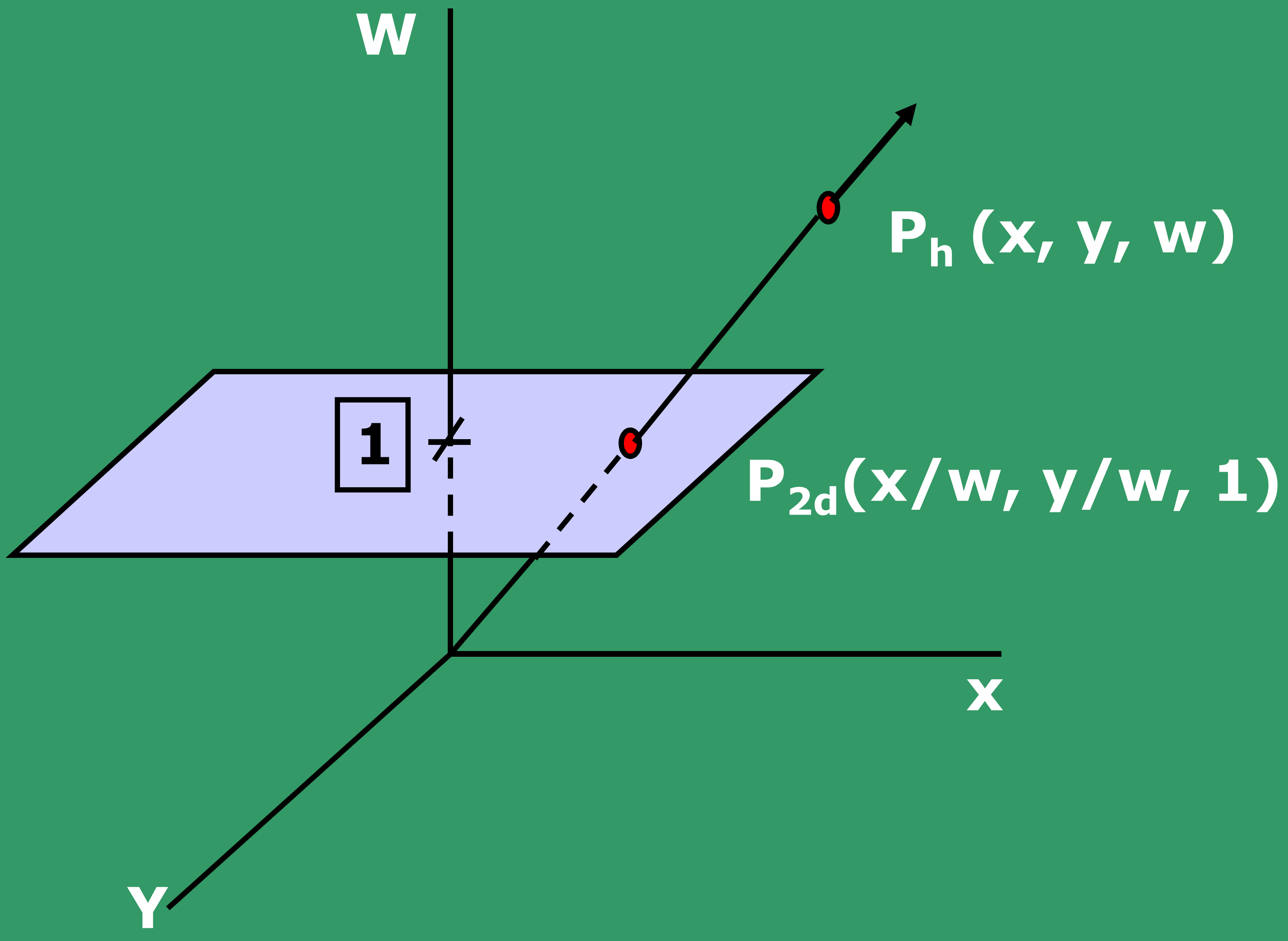
$x' = ax + cy + t_x$

$y' = bx + cy + t_y$

Each point is now represented by a triplet: (x, y, w).

$(x/w, y/w)$ are called the Cartesian coordinates of the homogeneous points.

Interpretation of Homogeneous Coordinates



Two homogeneous coordinates (x_1, y_1, w_1) & (x_2, y_2, w_2) may represent the same point, iff they are multiples of one another: say, (1,2,3) & (3,6,9).

There is no unique homogeneous representation of a point.

All triples of the form (t.x, t.y, t.W) form a line in x,y,W space.

Cartesian coordinates are just the plane w=1 in this space.

W=0, are the points at infinity

General Purpose 2D transformations in homogeneous coordinate representation

$$T = \begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix}$$

Parameters involved in scaling, rotation, reflection and shear are: **a, b, c, d**

If B = T.A, then

Translation parameters: (p, q)

If B = A.T, then

Translation parameters: (m, n)

What about S ?

COMPOSITE TRANSFORMATIONS

If we want to apply a series of transformations T_1, T_2, T_3 to a set of points, We can do it in two ways:

1) We can calculate $p'=T_1*p$, $p''= T_2*p'$, $p'''=T_3*p''$
2) Calculate $T= T_1*T_2*T_3$, then $p'''= T*p$.

Method 2, saves large number of additions and multiplications (computational time) – needs approximately 1/3 of as many operations. Therefore, we concatenate or compose the matrices into one final transformation matrix, and then apply that to the points.

Translations:

Translate the points by tx_1, ty_1, then by tx_2, ty_2:

$$\begin{bmatrix} 1 & 0 & (tx_1 + tx_2) \\ 0 & 1 & (ty_1 + ty_2) \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling:

Similar to translations

Rotations:

Rotate by θ_1, then by θ_2:

(i) stick the $(\theta_1 + \theta_2)$ in for θ, or

(ii) calculate T_1 for θ_1, then T_2 for θ_2 & multiply them.

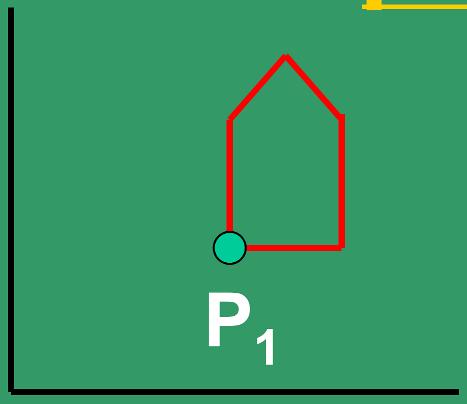
Exercise: Both gives the same result – work it out

Rotation about an arbitrary point P in space

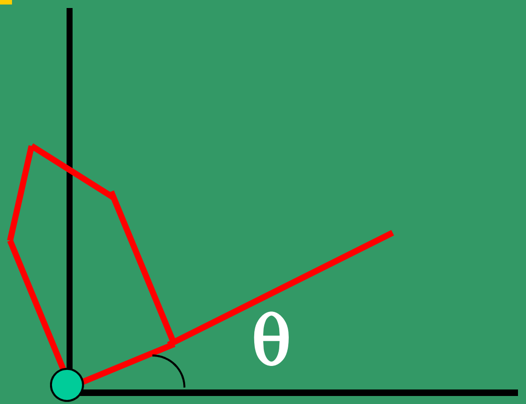
As we mentioned before, rotations are applied about the origin. So to rotate about any arbitrary point P in space, translate so that P coincides with the origin, then rotate, then translate back. Steps are:

- Translate by ($-P_x$, $-P_y$)
- Rotate
- Translate by (P_x, P_y)

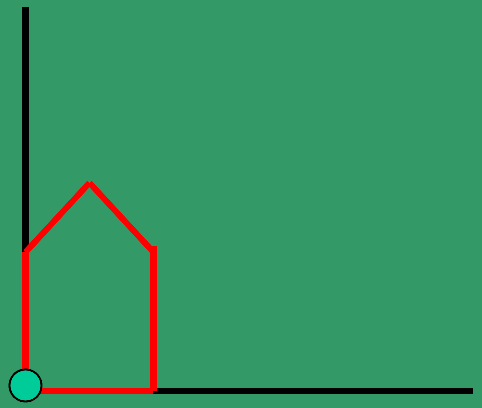
Rotation about an arbitrary point P in space



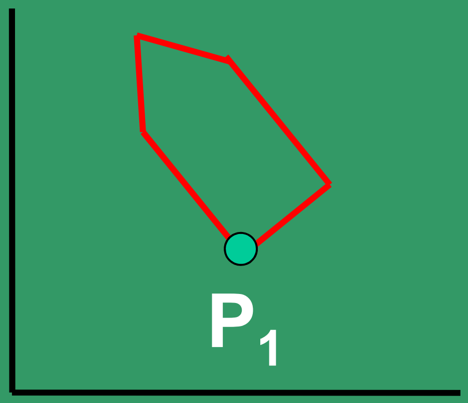
House at P_1



Rotation by θ



Translation of P_1 to Origin



Translation back to P_1

Rotation about an arbitrary point P in space

$$T = T_3(P_x, P_y) * T_2(\theta) * T_1(-P_x, -P_y)$$

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

Scaling about an arbitrary point in Space

Again,

- Translate P to the origin
- Scale
- Translate P back

$T = T_1(P_x, P_y) * T_2(S_x, S_y) * T_3(-P_x, -P_y)$

$$T = \begin{bmatrix} Sx & 0 & \{Px*(1-Sx)\} \\ 0 & Sy & \{Py*(1-Sy)\} \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection through an arbitrary line

Steps:

- **Translate line to the origin**
- **Rotation about the origin**
- **Reflection matrix**
- **Reverse the rotation**
- **Translate line back**

$$T_{GenRfl} = T_r \; R \; T_{rfl} \; R^T \; T_r^{-1}$$

Commutivity of Transformations

If we scale, then translate to the origin, and then translate back, is that equivalent to translate to origin, scale, translate back?

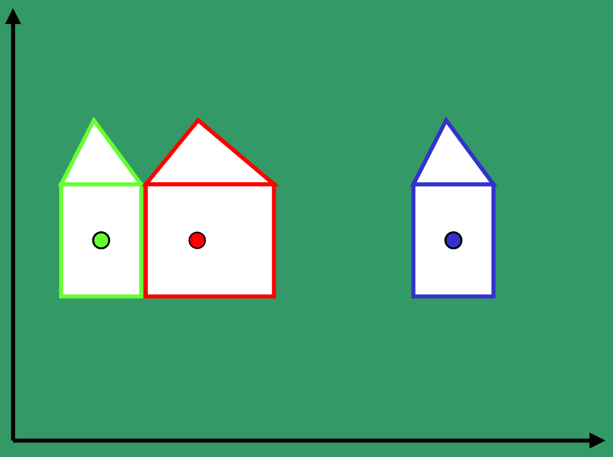
When is the order of matrix multiplication unimportant?

When does $T_1 * T_2 = T_2 * T_1$?

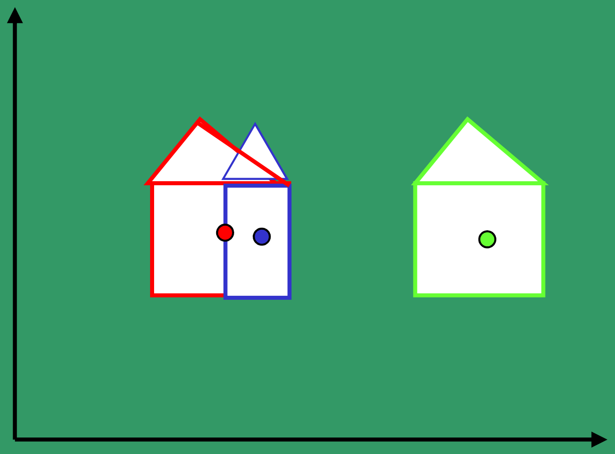
Cases where $T_1 * T_2 = T_2 * T_1$:

T_1	T_2
translation	translation
scale	scale
rotation	rotation
scale(uniform)	rotation

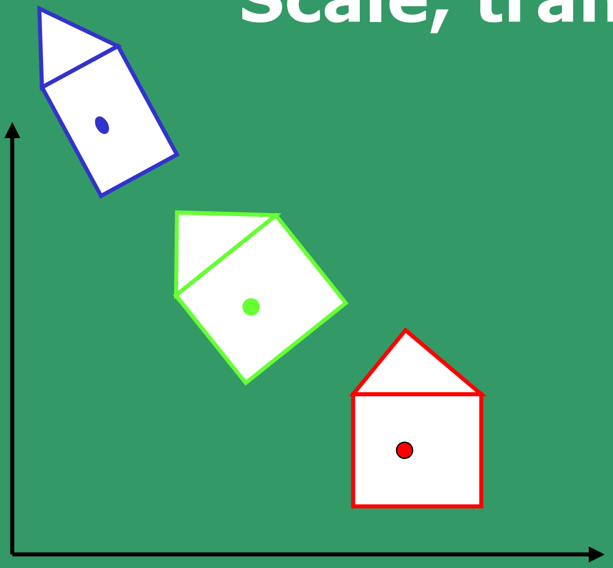
Order:
R-G-B



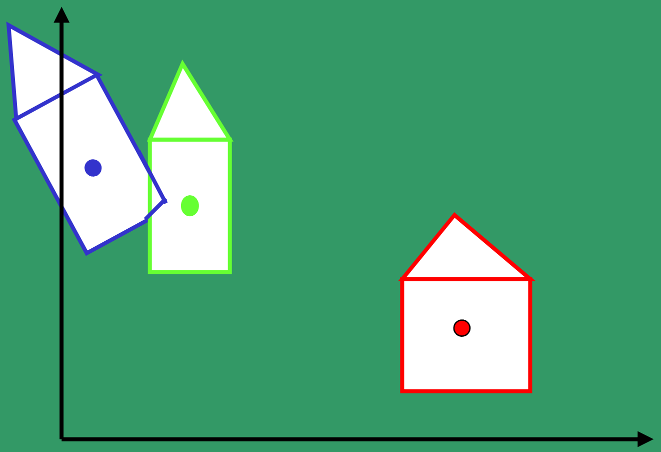
Scale, translate



Translate, scale



Rotate, differential scale



Differential scale, rotate

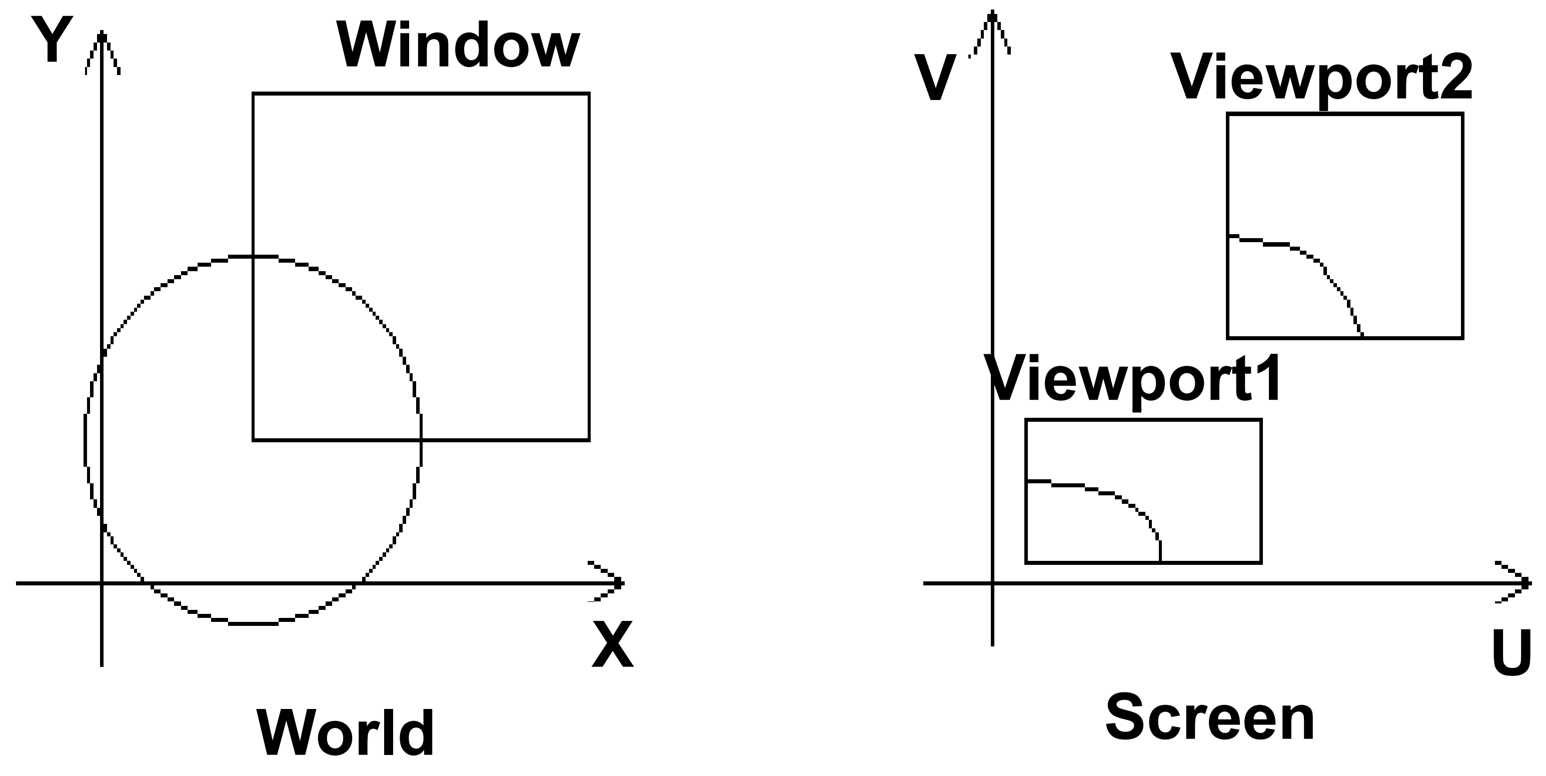
COORDINATE SYSTEMS

Screen Coordinates: The coordinate system used to address the screen (device coordinates)

World Coordinates: A user-defined application specific coordinate system having its own units of measure, axis, origin, etc.

Window: The rectangular region of the world that is visible.

Viewport: The rectangular region of the screen space that is used to display the window.


Y
Window
X
World
V
Viewport2
Viewport1
U
Screen

WINDOW TO VIEWPORT TRANSFORMATION

Purpose is to find the transformation matrix that maps the window in world coordinates to the viewport in screen coordinates.

Window: (x, y space) denoted by:

$x_{min}, y_{min}, x_{max}, y_{max}$

Viewport: (u, v space) denoted by:

$u_{min}, v_{min}, u_{max}, v_{max}$

The overall transformation:

- Translate the window to the origin
- Scale it to the size of the viewport
- Translate it to the viewport location

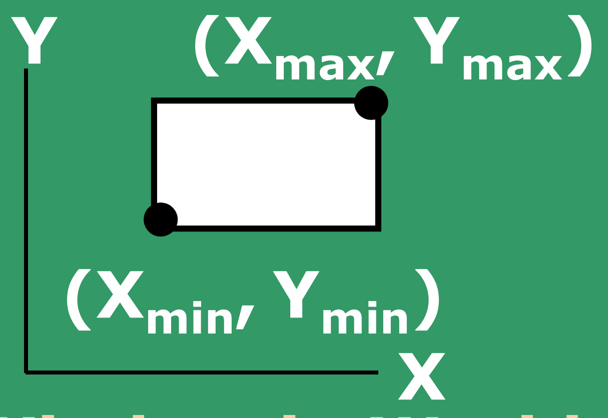
$$M_{WV} = T(U_{\min}, V_{\min}) * S(S_x, S_y) * T(-x_{\min}, -y_{\min});$$

$$S_x = (U_{\max} - U_{\min})/(x_{\max} - x_{\min});$$

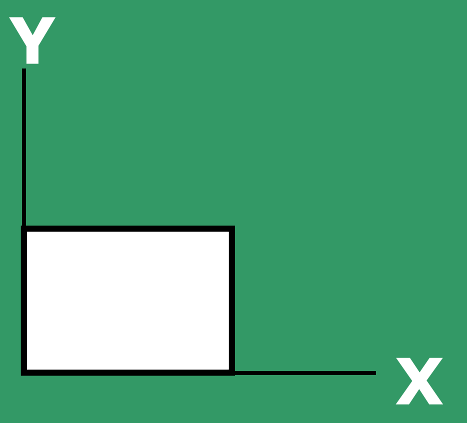
$$S_y = (V_{\max} - V_{\min})/(y_{\max} - y_{\min});$$

$$M_{WV} = \begin{bmatrix} S_x & 0 & (-x_{\min} * S_x + U_{\min}) \\ 0 & S_y & (-y_{\min} * S_y + V_{\min}) \\ 0 & 0 & 1 \end{bmatrix}$$

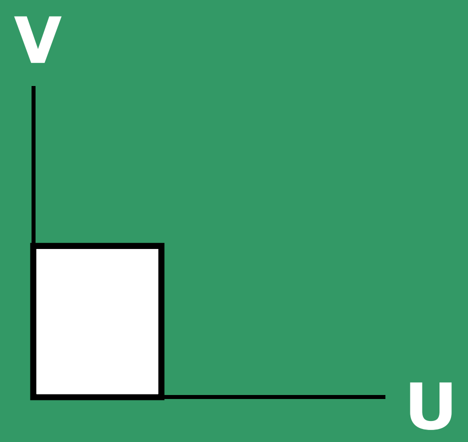
Window – Viewport Transformation



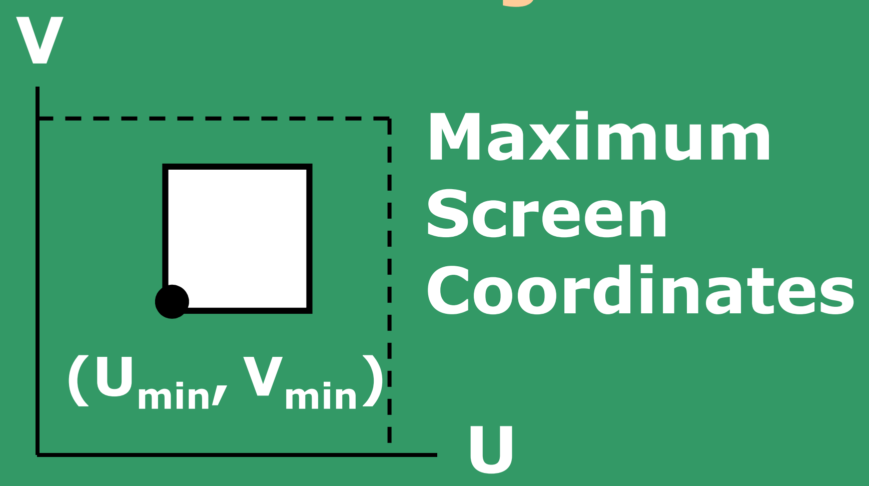
Window in World Coordinates



Window translated to origin



Window Scaled to size to Viewport



Viewport Translated to final position

Exercise -Transformations of Parallel Lines

Consider two parallel lines:

(i) A[X_1, Y_1] to B[X_2, Y_2] and

(ii) C[X_3, Y_3] to B[X_4, Y_4].

Slope of the lines:

$$m = \frac{Y_2 - Y_1}{X_2 - X_1} = \frac{Y_4 - Y_3}{X_4 - X_3}$$

Solve the problem:

If the lines are transformed by a matrix:

$$T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The slope of the transformed lines is:

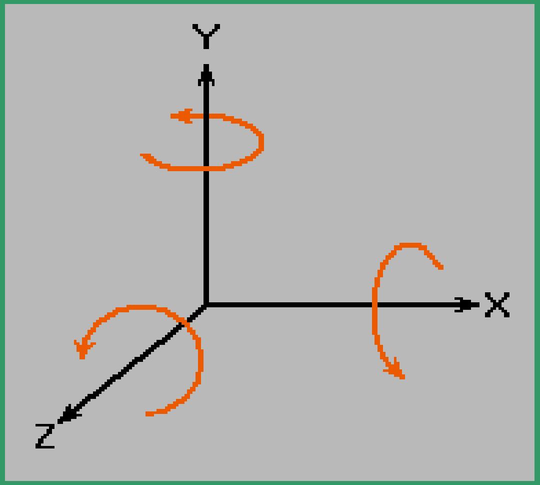
$$m' = \frac{b + dm}{a + cm}$$

Three - Dimensional

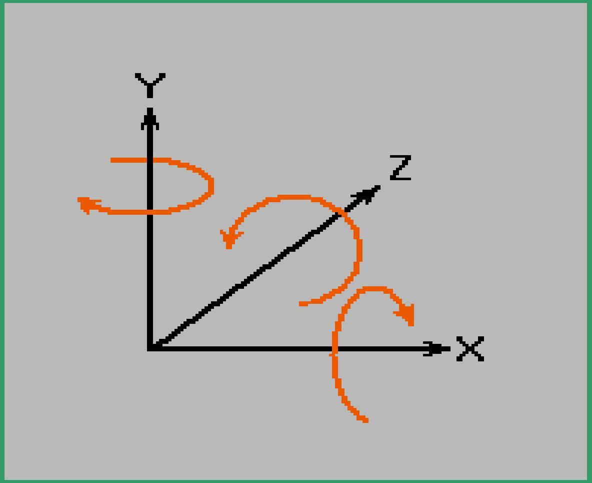
Graphics

Three-Dimensional Graphics

- Use of a right-handed coordinate system (consistent with math)
- Left-handed suitable to screens.
- To transform from right to left, negate the z values.



Right Handed Space



Left Handed Space

Homogeneous representation of a point in 3D space:

$$P = \left| \mathrm{x\ y\ z\ w} \right|^{\mathrm{T}}$$

$$(\mathrm{w} = 1, \text{for a 3D point})$$

Transformations will thus be represented by 4x4 matrices:

P′ = A.P

Transformation Matrix in 3D:

$$A = \begin{bmatrix} a & b & c & p \\ d & e & f & q \\ g & i & j & r \\ l & m & n & s \end{bmatrix} = \begin{bmatrix} \mathrm{T} & \mathrm{K} \\ \Gamma & \Theta \end{bmatrix}$$

where,

$$\mathrm{T} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & i & j \end{bmatrix}$$

produces linear transformations: scaling, shearing, reflection and rotation.

$\mathrm{K} = [p\ q\ r]^T$, produces translation

$\Gamma = [l\ m\ n]^T$, yields perspective transformation

while, $\Theta = s$, is responsible for uniform scaling

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

Translation

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} 1 & 0 & Sh_x & 0 \\ 0 & 1 & Sh_y & 0 \\ 0 & Sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear

Origin is unaffected by scale and shear

3D Reflection:

The following matrices:

$$T_{XY}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&0\\0&0&0&1\end{bmatrix} \quad T_{YZ}=\begin{bmatrix}-1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \quad T_{ZX}=\begin{bmatrix}1&0&0&0\\0&-1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

produce reflection about:

XY plane

YZ plane

ZX plane

respectively.

Rotation Matrices along an axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X-axis

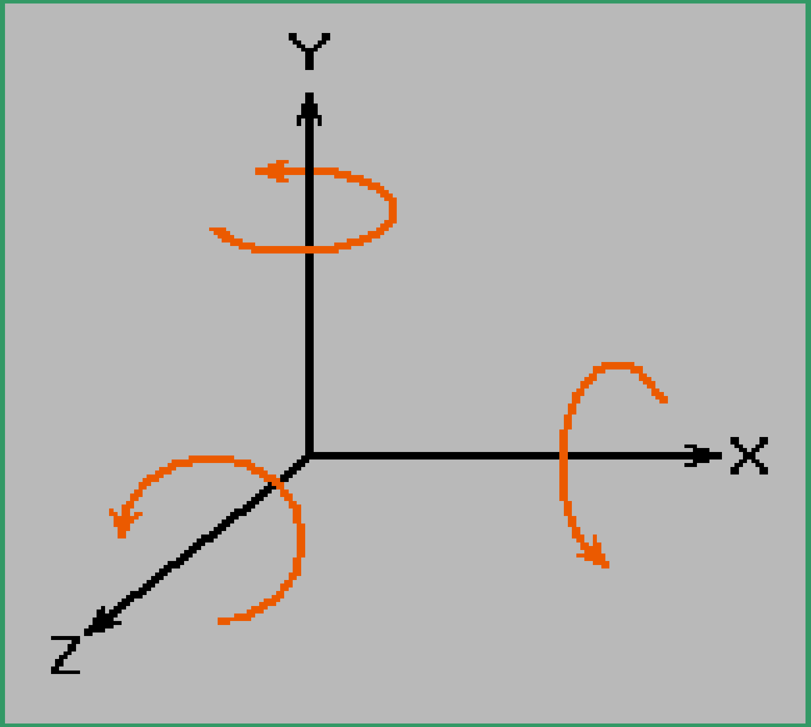
$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

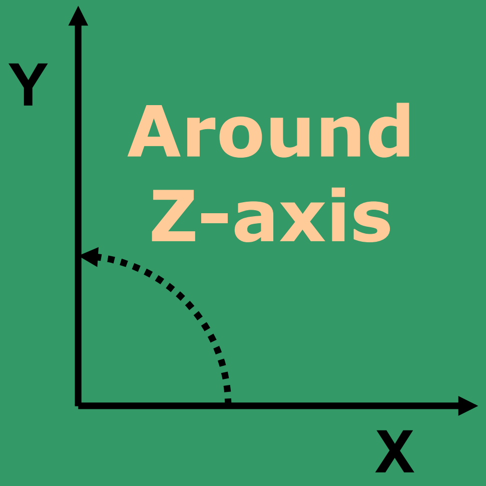
Y-axis

$$\begin{bmatrix} cos(\gamma) & -sin(\gamma) & 0 & 0 \\ sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

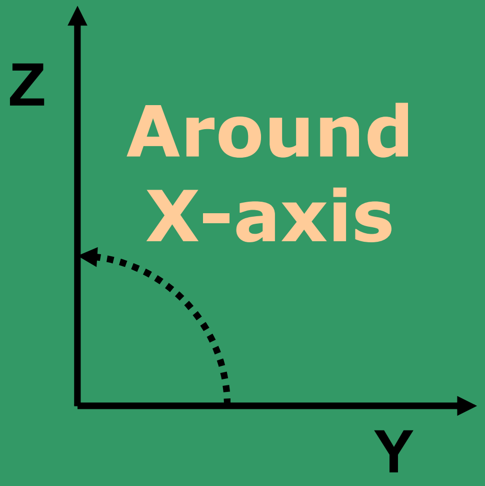
Z-axis

Why is the sign reversed in one case ?

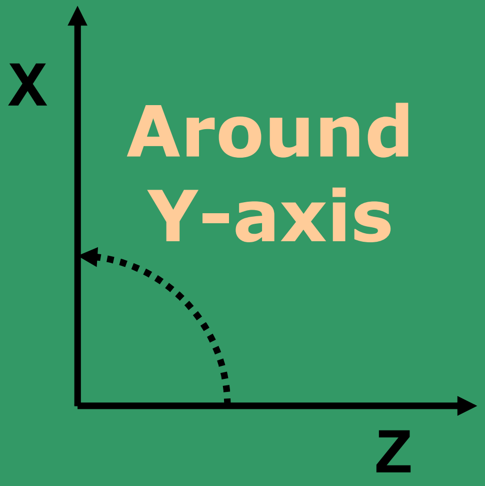




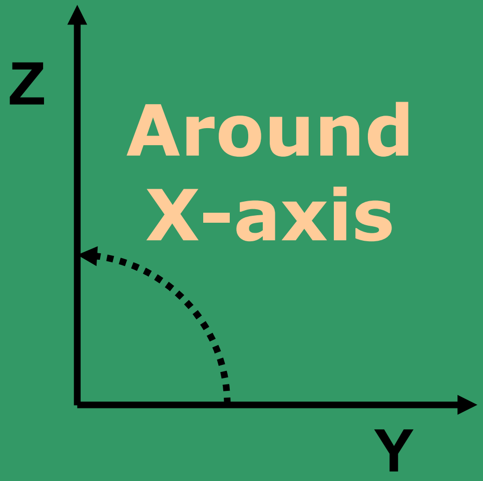
Around Z-axis



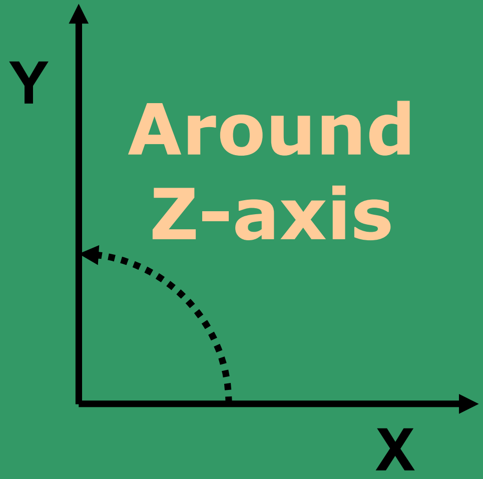
Around X-axis



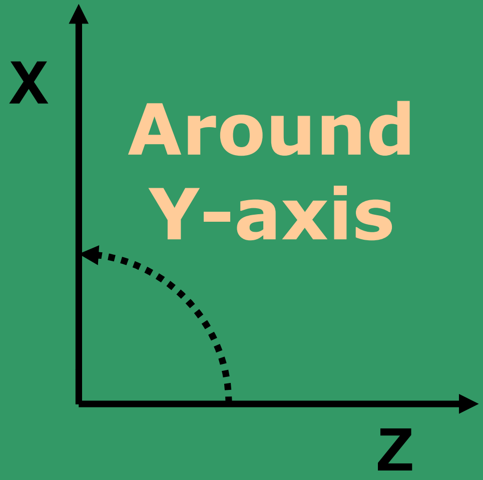
Around Y-axis



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{array}{l} \leftarrow X \\ \leftarrow Y \\ \leftarrow Z \\ \end{array}$$



$$\begin{bmatrix} cos(\gamma) & -sin(\gamma) & 0 & 0 \\ sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{array}{l} \leftarrow X \\ \leftarrow Y \\ \leftarrow Z \\ \end{array}$$



$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{array}{l} \leftarrow X \\ \leftarrow Y \\ \leftarrow Z \\ \end{array}$$

Rotation About an Arbitrary Axis in Space

Assume, we want to perform a rotation by θ degrees, about an axis in space passing through the point (x_0, y_0, z_0) with direction cosines (c_x, c_y, c_z).

1. First of all, translate by:

   $$|T| = -(x_0, y_0, z_0)^T$$

2. Next, we rotate the axis into one of the principle axes, let's pick, Z ($|R_x|$, $|R_y|$).
3. We rotate next by θ degrees in Z ($|R_z(\theta)|$).
4. Then we undo the rotations to align the axis.
5. We undo the translation: translate by $(-x_0, -y_0, -z_0)^T$

The tricky part of the algorithm is in step (2), as given before.

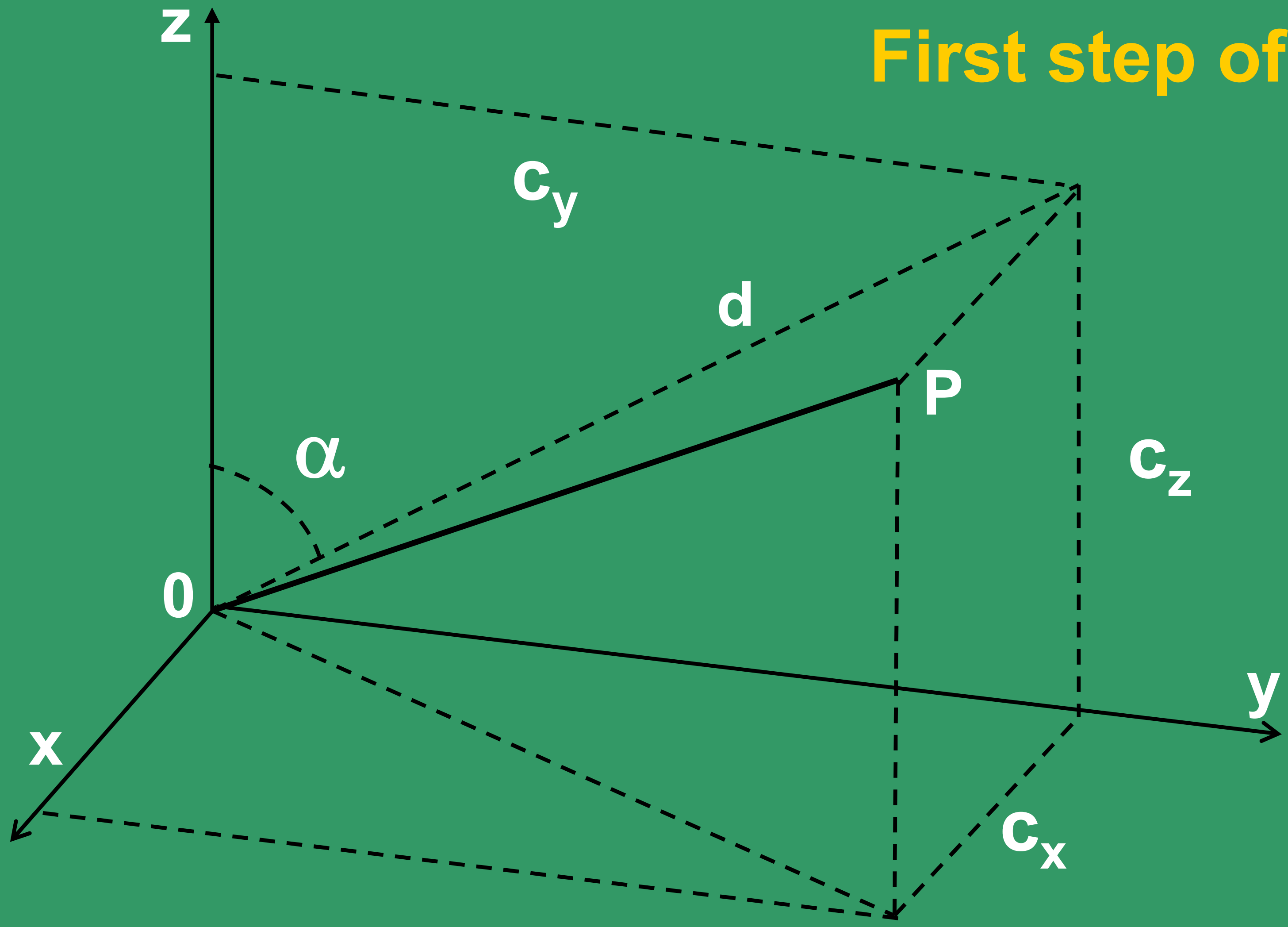
This is going to take 2 rotations:

i) About x-axis (to place the axis in the xz plane)

and

ii) About y-axis (to place the result coincident with the z-axis).

First step of Rotation:



Rotation about x by α:

How do we determine α?

Project the unit vector, along OP, into the yz plane.

The y and z components, c_y and c_z, are the direction cosines of the unit vector along the arbitrary axis.

It can be seen from the diagram, that :

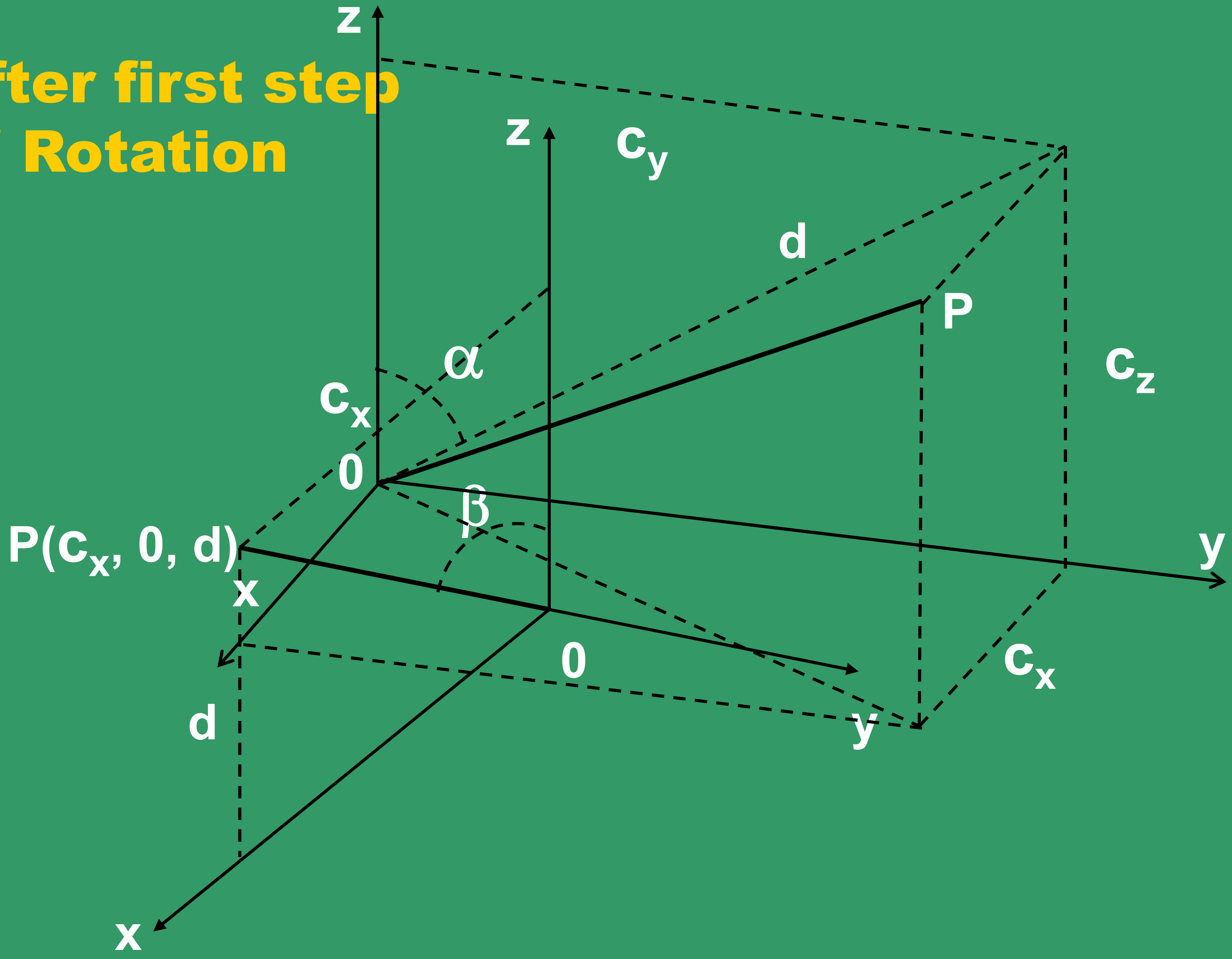
$$\mathrm{d} = \mathrm{sqrt}\left(C_y^2 + C_z^2\right)$$

$$\cos(\alpha) = C_z / d$$

$$\sin(\alpha) = C_y / d$$

$$\alpha = \sin^{-1}\left[\frac{c_y}{\sqrt{c_y^2 + c_z^2}}\right]$$

After first step of Rotation



Rotation by β about y:

How do we determine β?

Steps are similar to that done for α:

- Determine the angle β to rotate the result into the Z axis:
- The x component is c_x and the z component is d.

$\cos(\beta) = d = d/(\text{length of the unit vector})$

$\sin(\beta) = c_x = c_x/(\text{length of the unit vector})$.

Final Transformation for 3D rotation, about an arbitrary axis:

$$M = |T|\ |R_x|\ |R_y|\ |R_z|\ |R_y|^{-1}\ |R_x|^{-1}\ |T|^{-1}$$

Final Transformation matrix for 3D rotation, about an arbitrary axis:

$$M = |T|\ |R_x|\ |R_y|\ |R_z|\ |R_y|^{-1}\ |R_x|^{-1}\ |T|^{-1}$$

where:

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C_z/d & -C_y/d & 0 \\ 0 & C_y/d & C_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$R_y = \begin{bmatrix} d & 0 & -C_x & 0 \\ 0 & 1 & 0 & 0 \\ C_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

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

$$M = |T|\ |R_x|\ |R_y|\ |R_z|\ |R_y|^{-1}\ |R_x|^{-1}\ |T|^{-1}$$
$$= [T\ R_x\ R_y]\ [R_z]\ [T\ R_x\ R_y]^{-1}$$
$$= C\ [R_z]\ C^{-1}$$

A special case of 3D rotation:

Rotation about an axis parallel to a coordinate axis (say, parallel to X-axis):

$$M_X = |T|\ \ |R_X|\ \ |T|^{-1}$$

Rotation About an Arbitrary Axis in Space

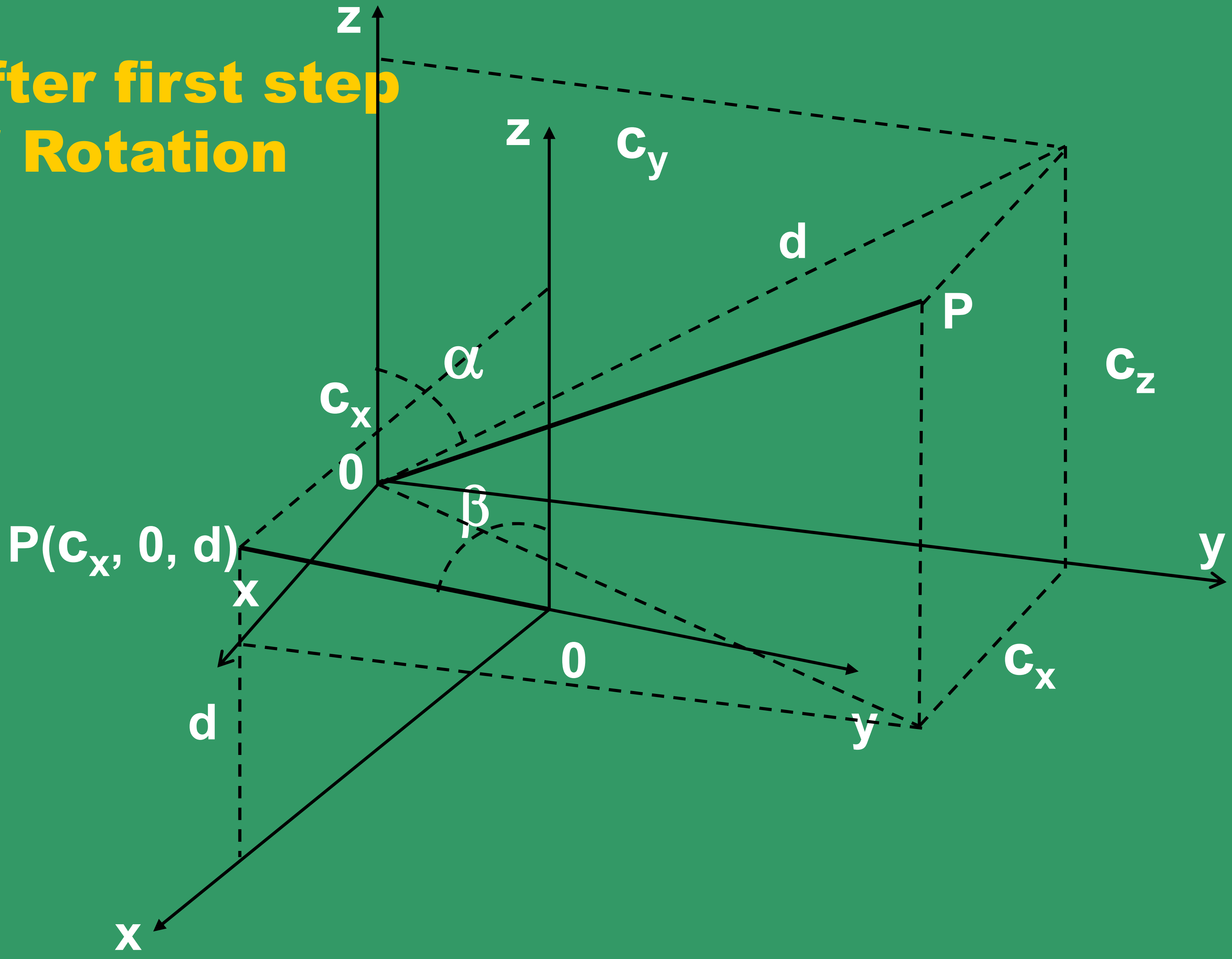
Assume, we want to perform a rotation by θ degrees, about an axis in space passing through the point (x_0, y_0, z_0) with direction cosines (c_x, c_y, c_z).

1. First of all, translate by:

   $$|T| = -(x_0, y_0, z_0)^T$$

2. Next, we rotate the axis into one of the principle axes, let's pick, Z ($|R_x|$, $|R_y|$).
3. We rotate next by θ degrees in Z ($|R_z(\theta)|$).
4. Then we undo the rotations to align the axis.
5. We undo the translation: translate by $(-x_0, -y_0, -z_0)^T$

After first step of Rotation



Final Transformation matrix for 3D rotation, about an arbitrary axis:

$$M = |T|\ |R_x|\ |R_y|\ |R_z|\ |R_y|^{-1}\ |R_x|^{-1}\ |T|^{-1}$$

where:

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C_z/d & -C_y/d & 0 \\ 0 & C_y/d & C_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$R_y = \begin{bmatrix} d & 0 & -C_x & 0 \\ 0 & 1 & 0 & 0 \\ C_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

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

If you are given 2 points instead (on the axis of rotation), you can calculate the direction cosines of the axis as follows:

$$V = \left| (x_1 - x_0) \; (y_1 - y_0) \; (z_1 - z_0) \right|^T$$

$$c_x = (x_1 - x_0) / |V|$$
$$c_y = (y_1 - y_0) / |V|$$
$$c_z = (z_1 - z_0) / |V|,$$

where $|V|$ is the lenght of the vector V.

Reflection through an arbitrary plane

Method is similar to that of rotation about an arbitrary axis.

$$M = |T|\ |R_x|\ |R_y|\ |R_{fl}|\ |R_y|^{-1}\ |R_x|^{-1}\ |T|^{-1}$$

T does the job of translating the origin to the plane.

R_x and R_y will rotate the vector normal to the reflection plane (at the origin), until it is coincident with the +Z axis.

R_{fl} is the reflection matrix about X-Y plane or Z=0 plane.

Spaces

Object Space:

definition of objects. Also called Modeling space.

World Space:

where the scene and viewing specification is made

***Eyespace* (Normalized Viewing Space):**

where eye point (COP) is at the origin looking down the Z axis.

3D Image Space:

A 3D Projective space.

Dimensions: [-1:1] in X & Y, [0:1] in Z.

This is where image space hidden surface algorithms work.

Screen Space (2D):

Range of Coordinates -
[0 : width], [0 : height]

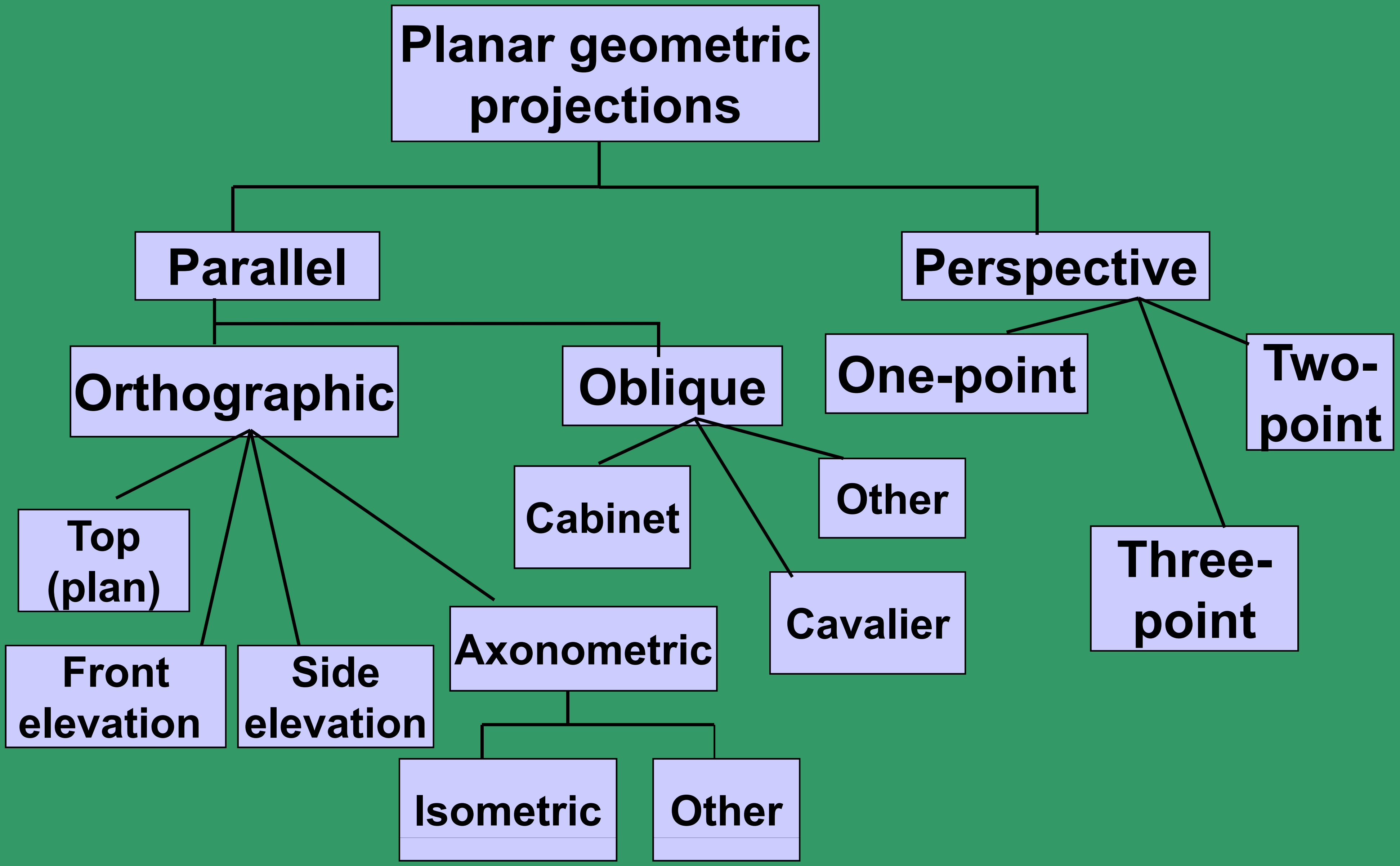
Projections

We will look at several planar geometric 3D to 2D projection:

- Parallel Projections
  - Orthographic
  - Oblique
- Perspective

Projection of a 3D object is defined by straight projection rays (projectors) emanating from the center of projection (COP) passing through each point of the object and intersecting the projection plane.

Classification of Geometric Projections



Perspective Projections

Distance from COP to projection plane is finite. The projectors are not parallel & we specify a center of projection (COP).

Center of Projection is also called the Perspective Reference Point

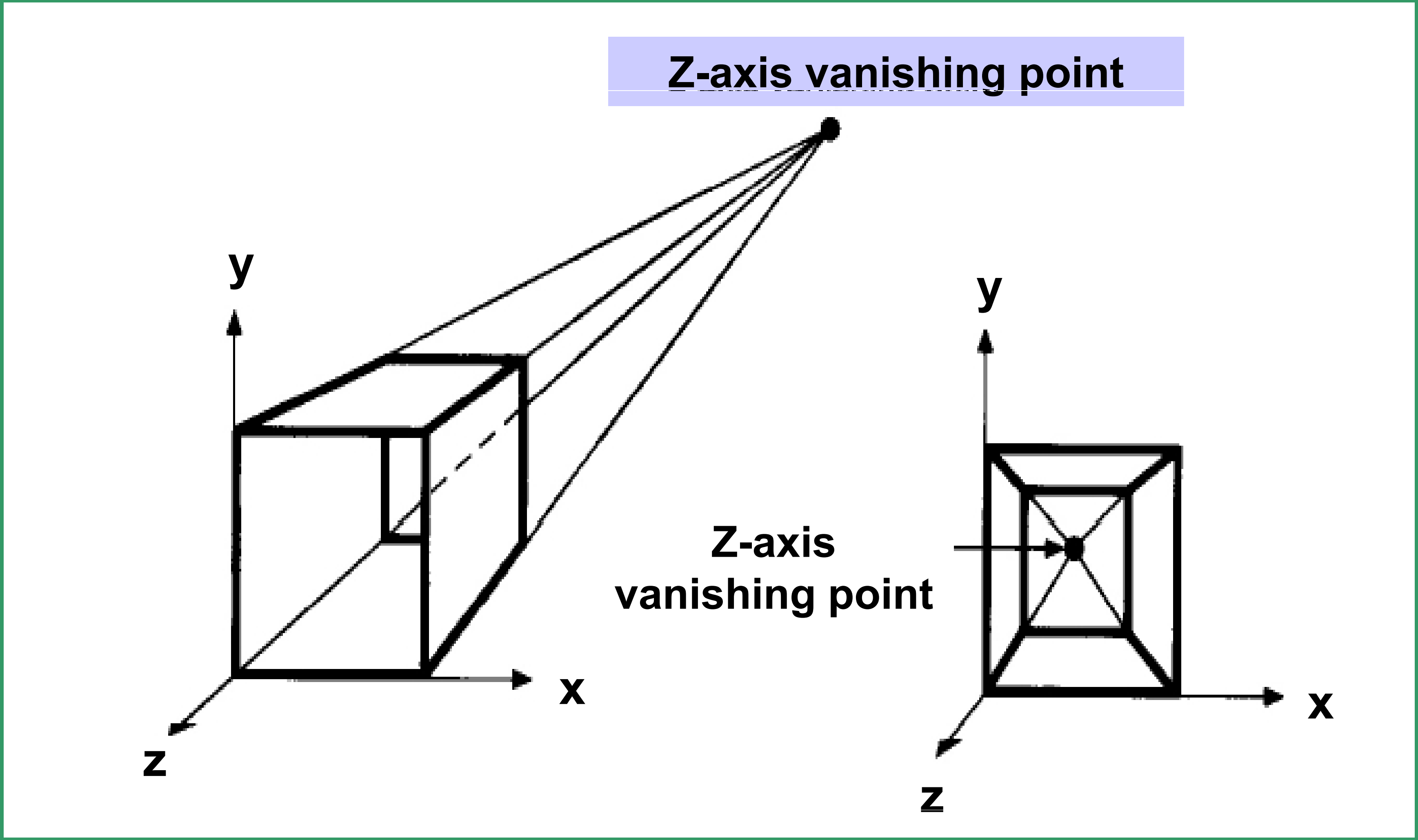
COP = PRP

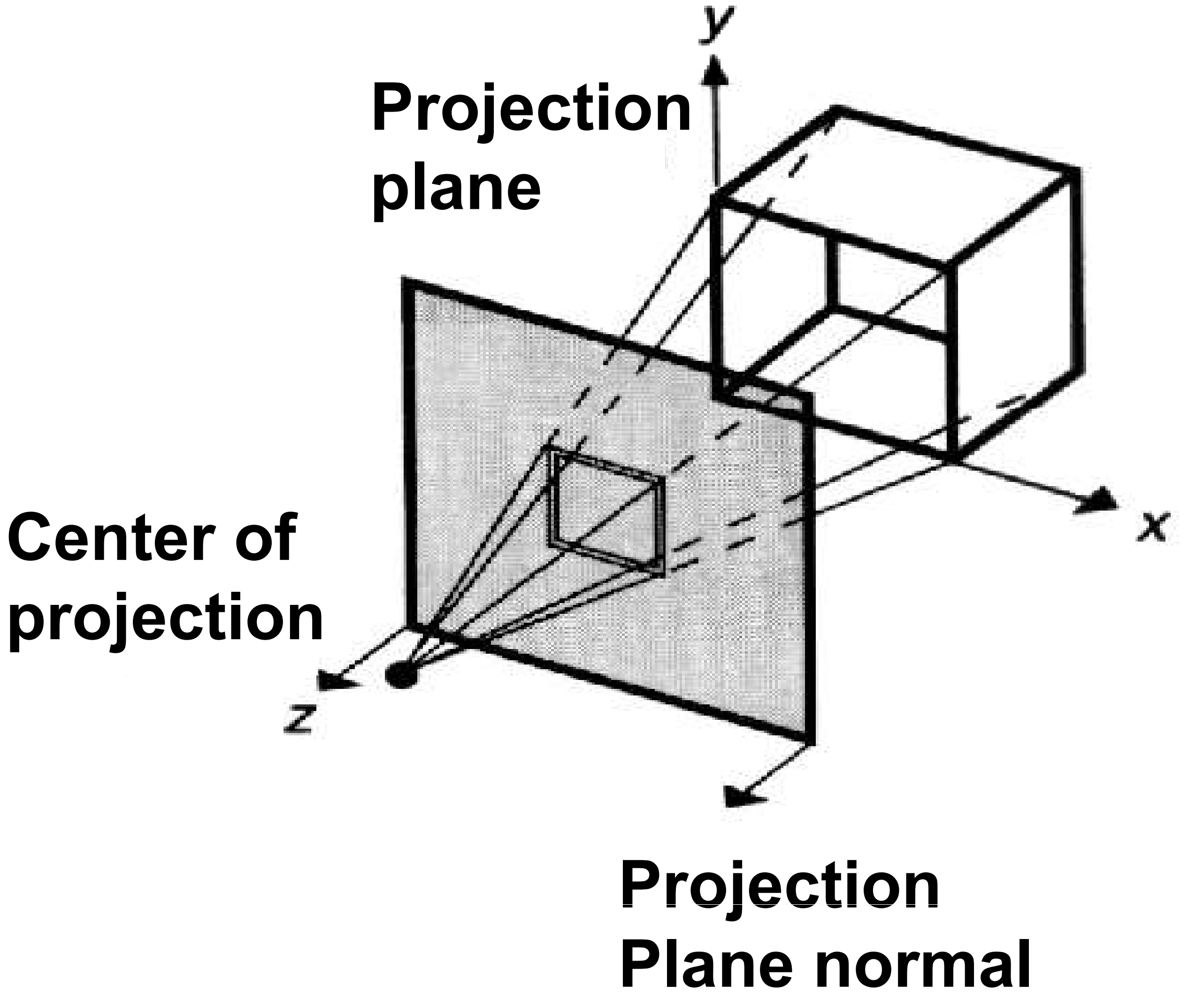
Perspective foreshortening:

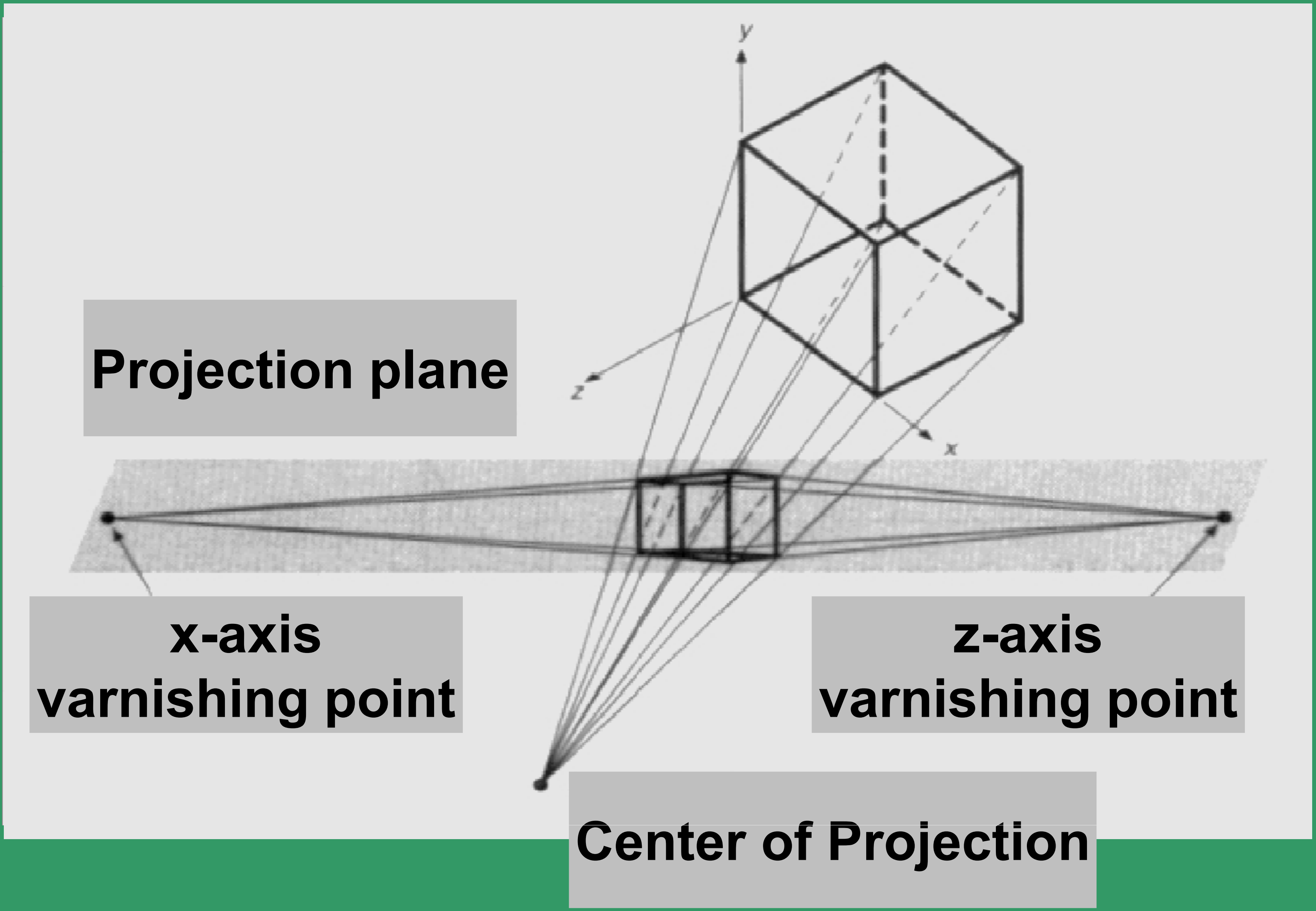
The size of the perspective projection of the object varies inversely with the distance of the object from the center of projection.

Vanishing Point:

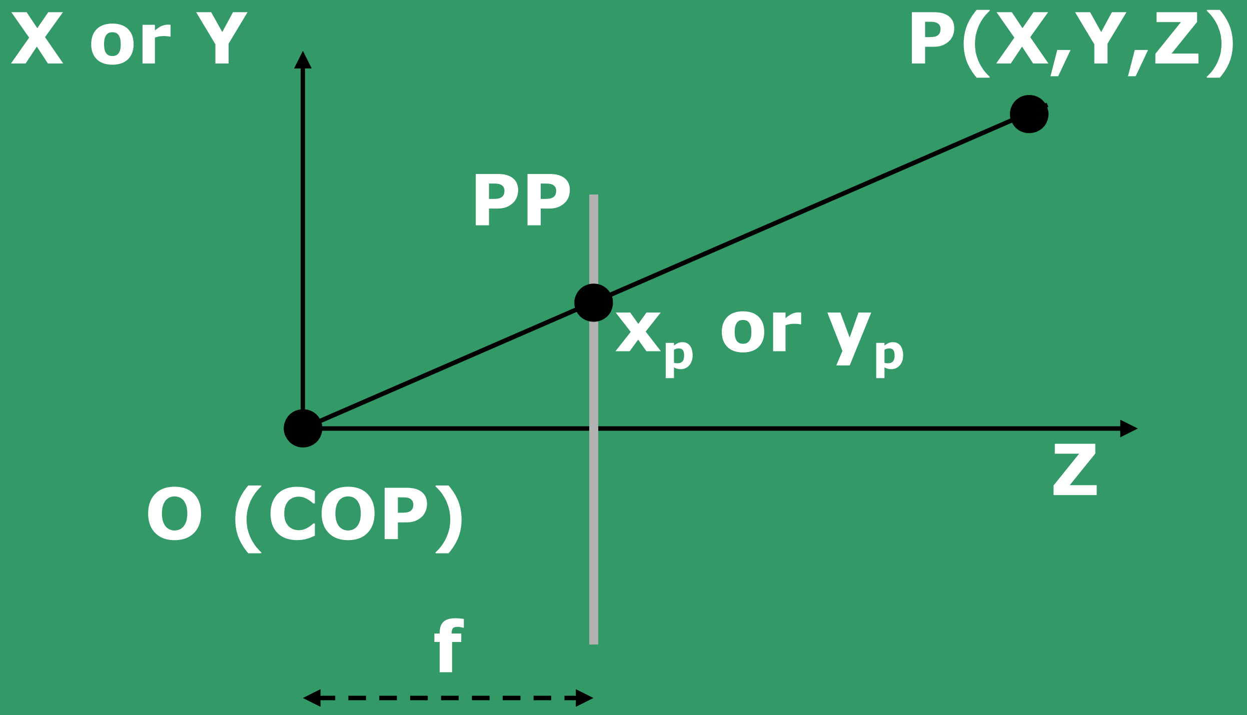
The perspective projections of any set of parallel lines that are not parallel to the projection plane converge to a vanishing point.


Z-axis vanishing point
y
x
z
y
Z-axis
vanishing point
x
z

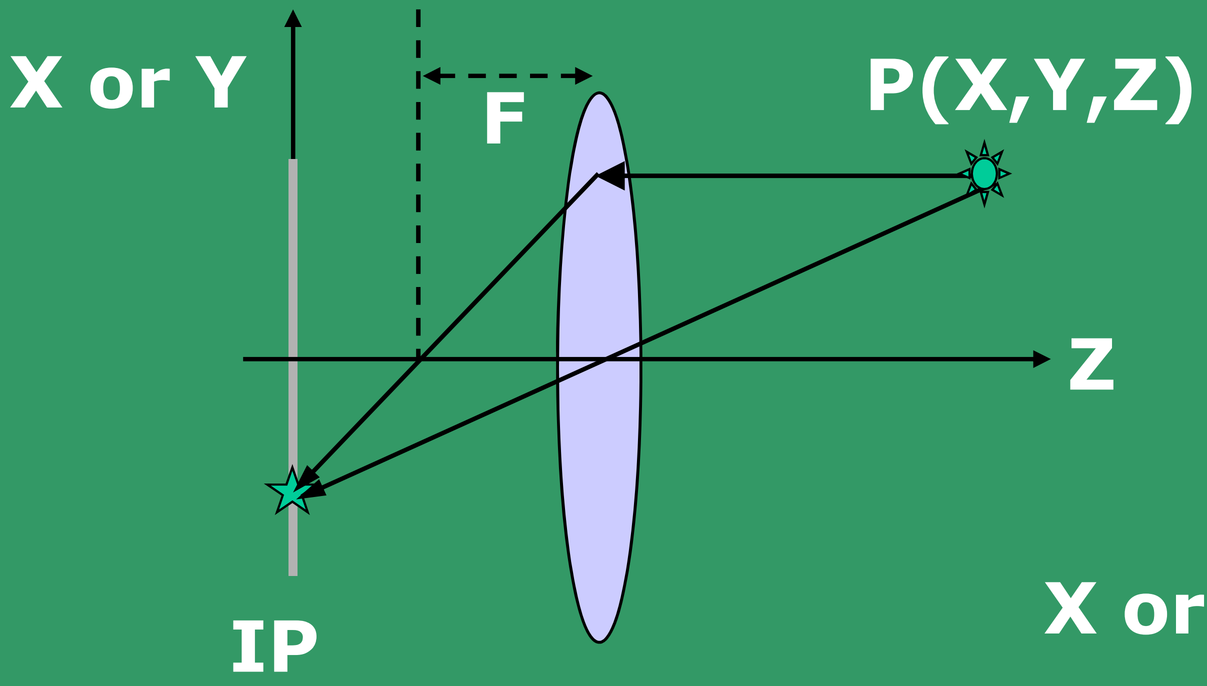

y
Projection plane
x
Center of projection
z
Projection Plane normal

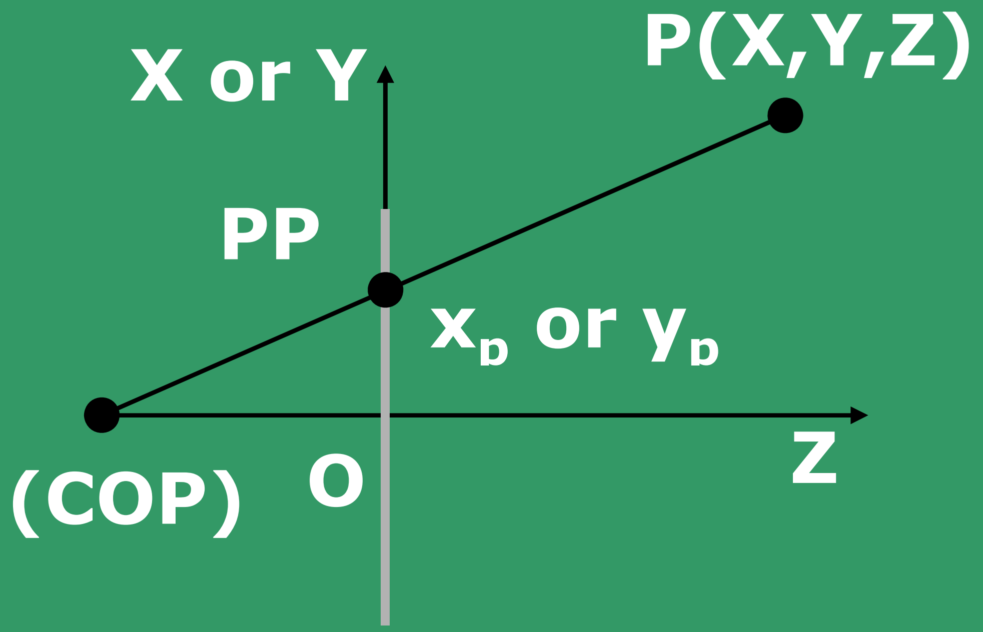

y
z
x
Projection plane
x-axis varnishing point
z-axis varnishing point
Center of Projection

Perspective Geometry and Camera Models

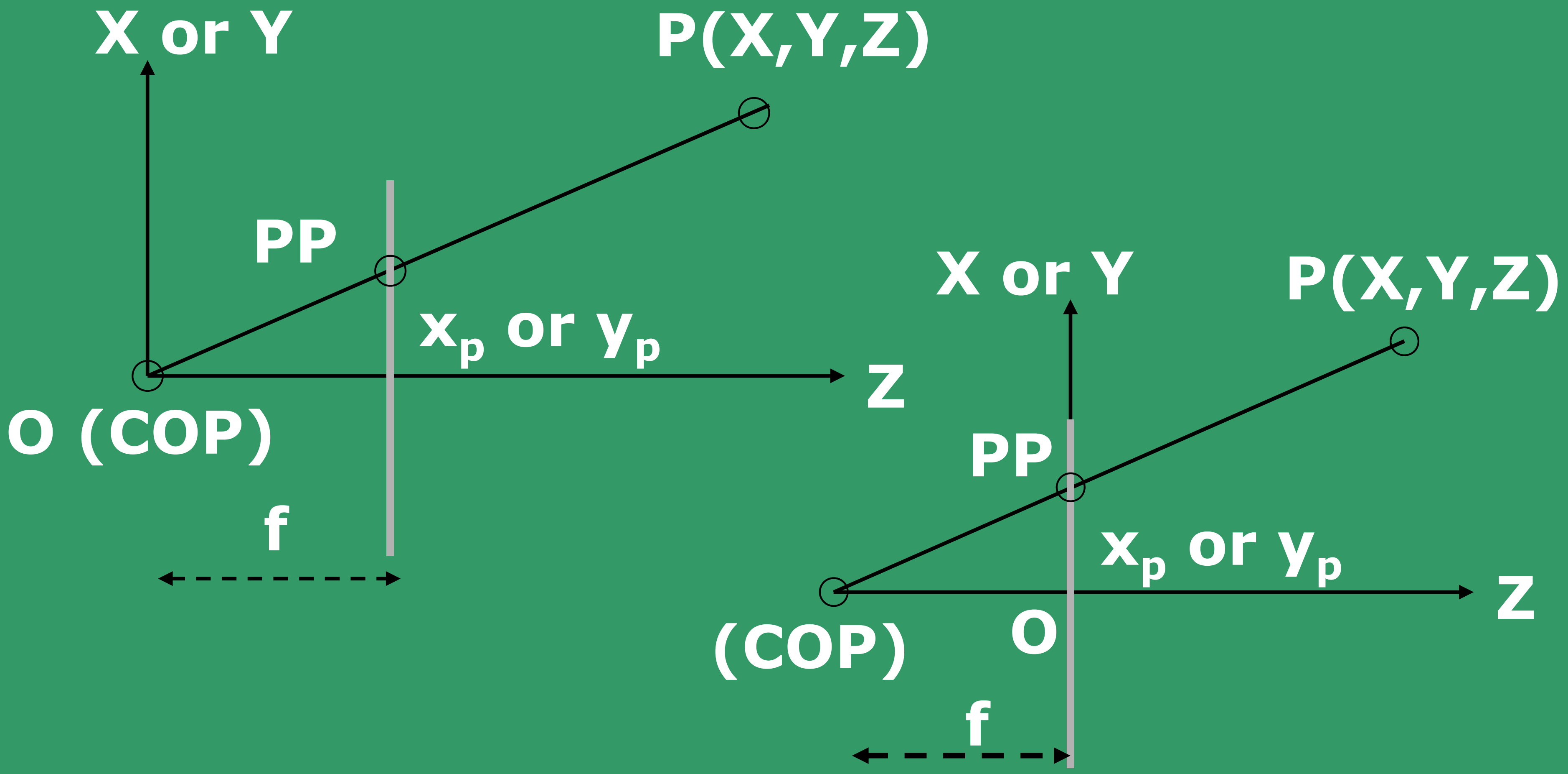


Perspective Geometry and Camera Models





Perspective Geometry and Camera Models



Equations of Perspective geometry, next ->

Equations of Perspective geometry

$$x_p/f = X/Z; \quad y_p/f = Y/Z;$$

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}$$

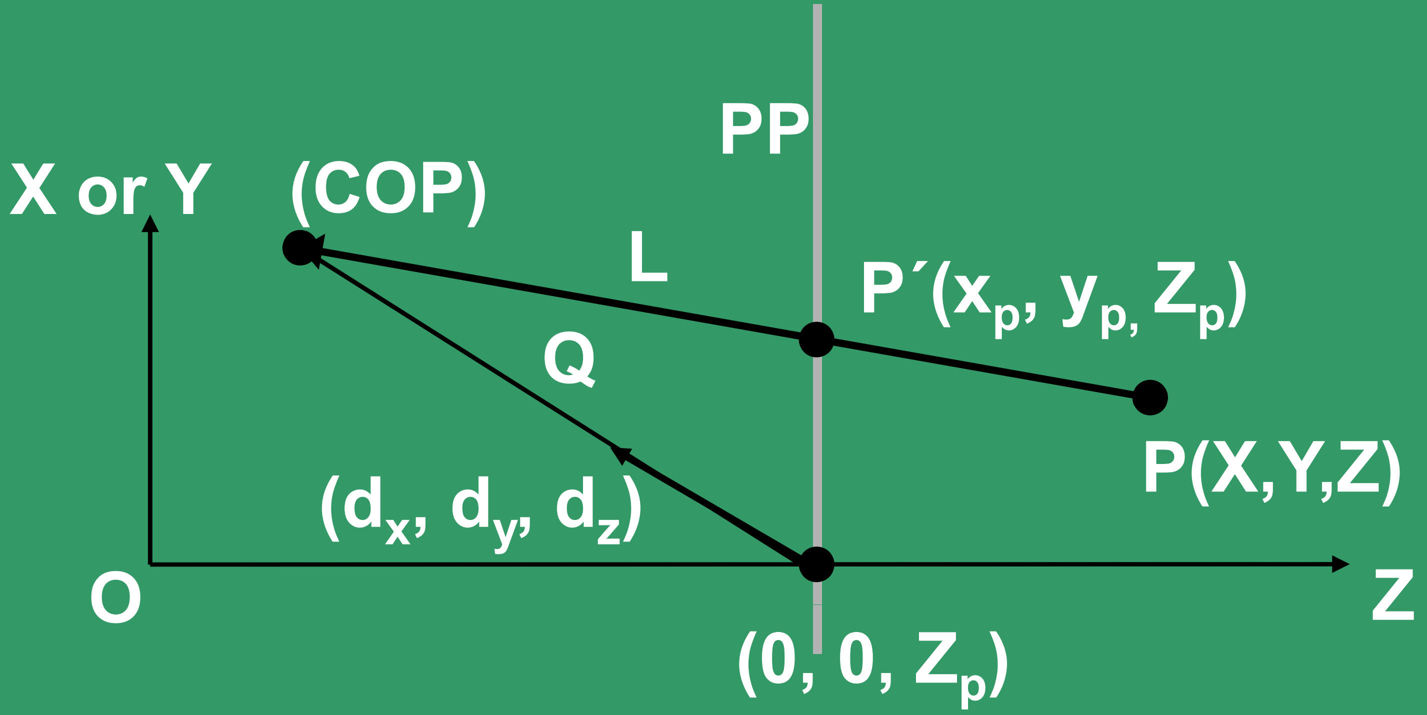
P´ = M_{per}.P;

where, P = $[X\ Y\ Z\ 1]^T$

$$\frac{x_p}{f} = \frac{X}{Z+f}; \quad \frac{y_p}{f} = \frac{Y}{Z+f};$$

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 1 \end{bmatrix}$$

Generalized formulation of perspective projection:



Parametric eqn. of the line L between COP and P:

COP + t(P-COP); $0 \leq t \leq 1$.

Let the direction vector from $(0, 0, Z_p)$ to COP be (d_x, d_y, d_z),
and Q be the distance from $(0, 0, Z_p)$ to COP.

Then COP = $(0, 0, Z_p) + Q(d_x, d_y, d_z)$.

The coordinates of any point on line L is:

$$X' = Qd_x + (X - Qd_x)t;$$

$$Y' = Qd_y + (Y - Qd_y)t;$$

$$Z' = (Z_p + Qd_z) + (Z - (Z_p + Qd_z))t;$$

Using the condition $Z' = Z_p$, at the intersection of line L and plane PP:

$$t = \frac{-Qd_z}{Z - (Z_p + Qd_z)}$$

Now subsitute to obtain, x_p and y_p.

$$x_p = \frac{X - Z\dfrac{d_x}{d_z} + Z_p\dfrac{d_x}{d_z}}{\dfrac{Z_p - Z}{Qd_z} + 1}$$

$$y_p = \frac{Y - Z\dfrac{d_y}{d_z} + Z_p\dfrac{d_y}{d_z}}{\dfrac{Z_p - Z}{Qd_z} + 1}$$

Generalized formula of perspective projection matrix:

$$M_{gen} = \begin{bmatrix} 1 & 0 & -\dfrac{d_x}{d_z} & z_p \dfrac{d_x}{d_z} \\ 0 & 1 & -\dfrac{d_y}{d_z} & z_p \dfrac{d_y}{d_z} \\ 0 & 0 & -\dfrac{Z_p}{Qd_z} & \dfrac{Z_p^2}{Qd_z} + Z_p \\ 0 & 0 & -\dfrac{1}{Qd_z} & \dfrac{Z_p}{Qd_z} + 1 \end{bmatrix}$$

Special cases from the generalized formulation of the perspective projection matrix

Matrix Type	Z_p	Q	$[d_x, d_y, d_z]$
M_{orth}	0	Infinity	[0, 0, -1]
M_{per}	d	d	[0, 0, -1]
M'_{per}	0	d	[0, 0, -1]

If Q is finite, M_{gen} defines a one-point perspective projection in the above two cases.

Parallel Projection

Distance from COP to projection plane is infinite.

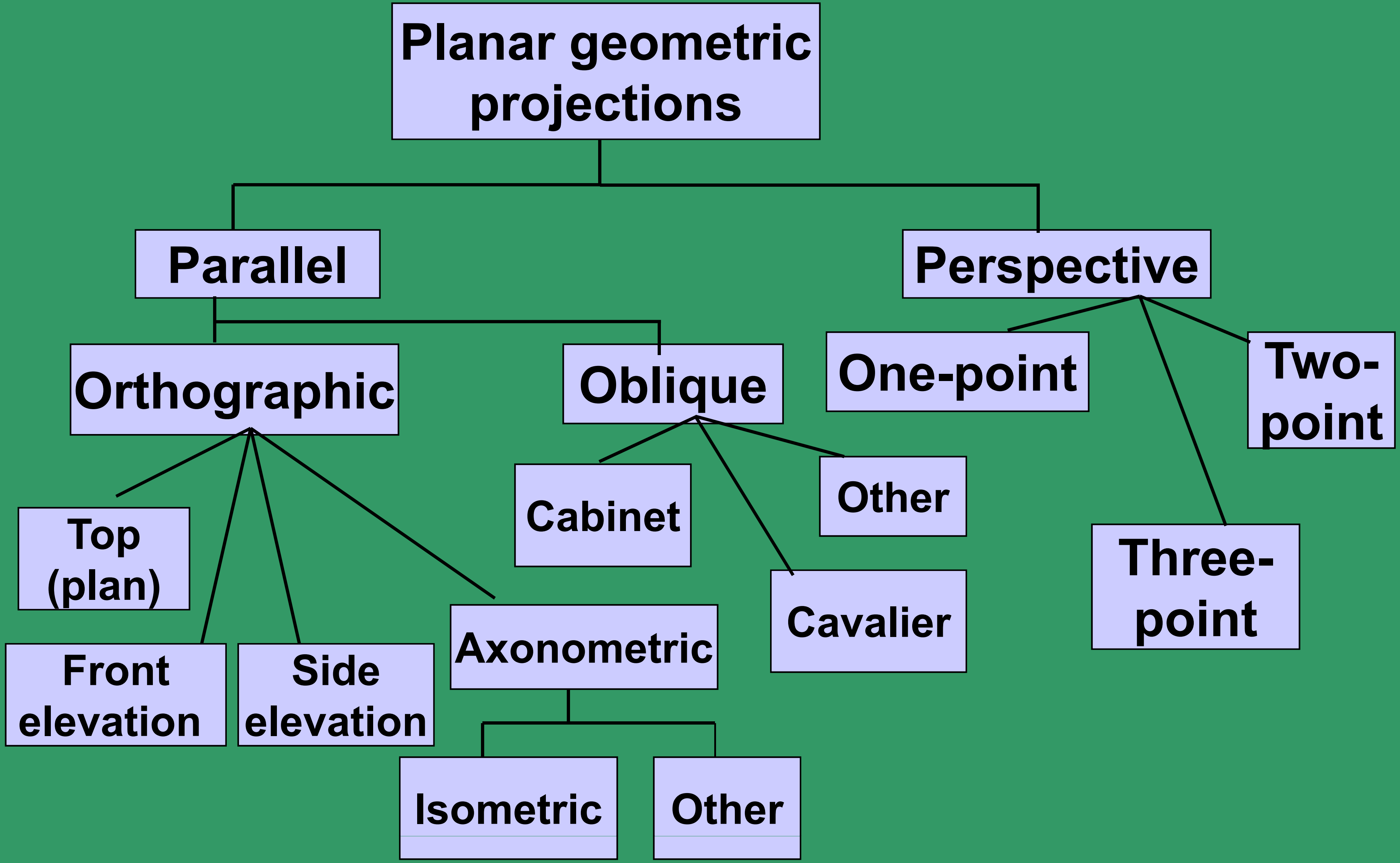
Therefore, the projectors are parallel lines & we need to specify a:

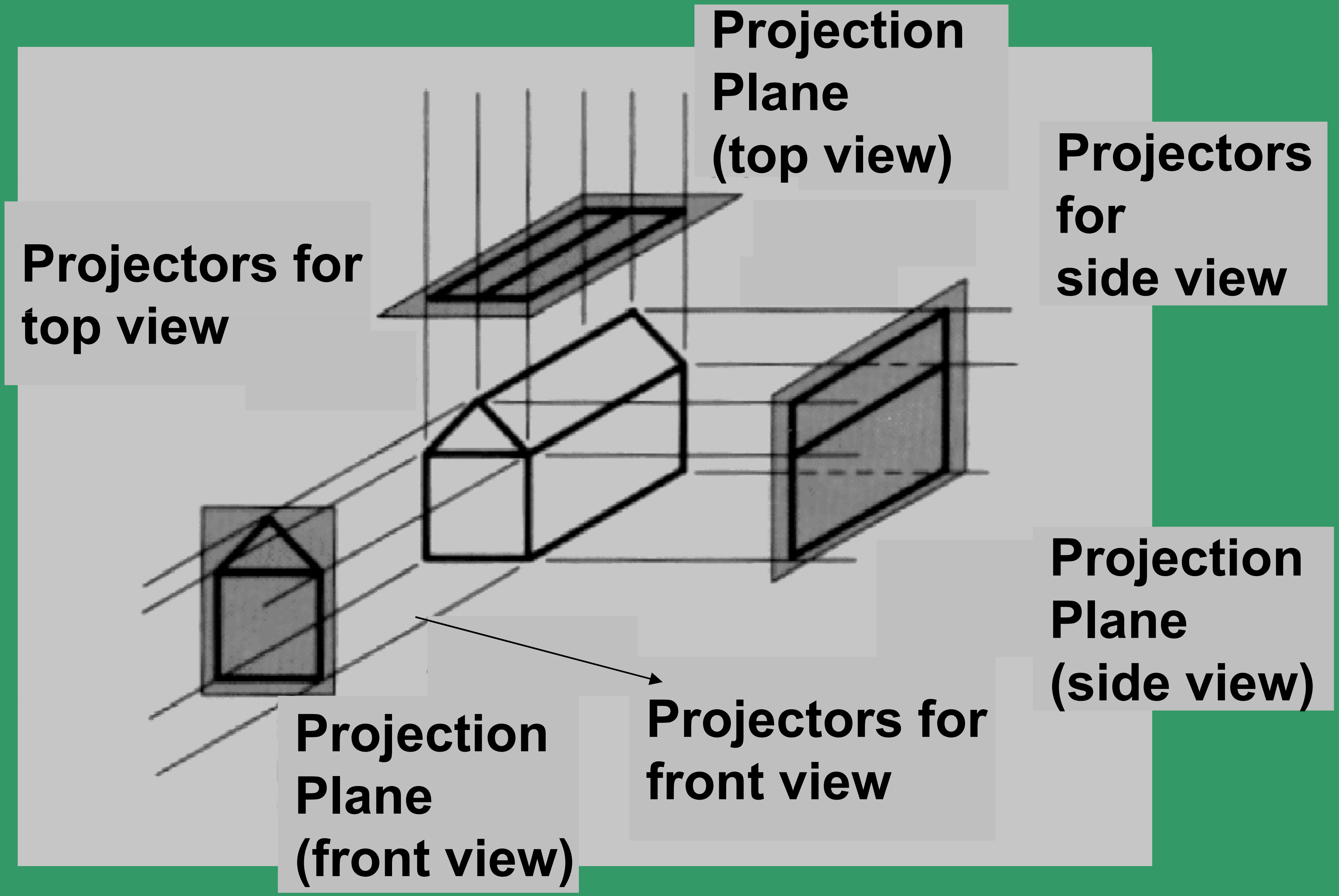
direction of projection (DOP)

Orthographic:

the direction of projection and the normal to the projection plane are the same. (direction of projection is normal to the projection plane).

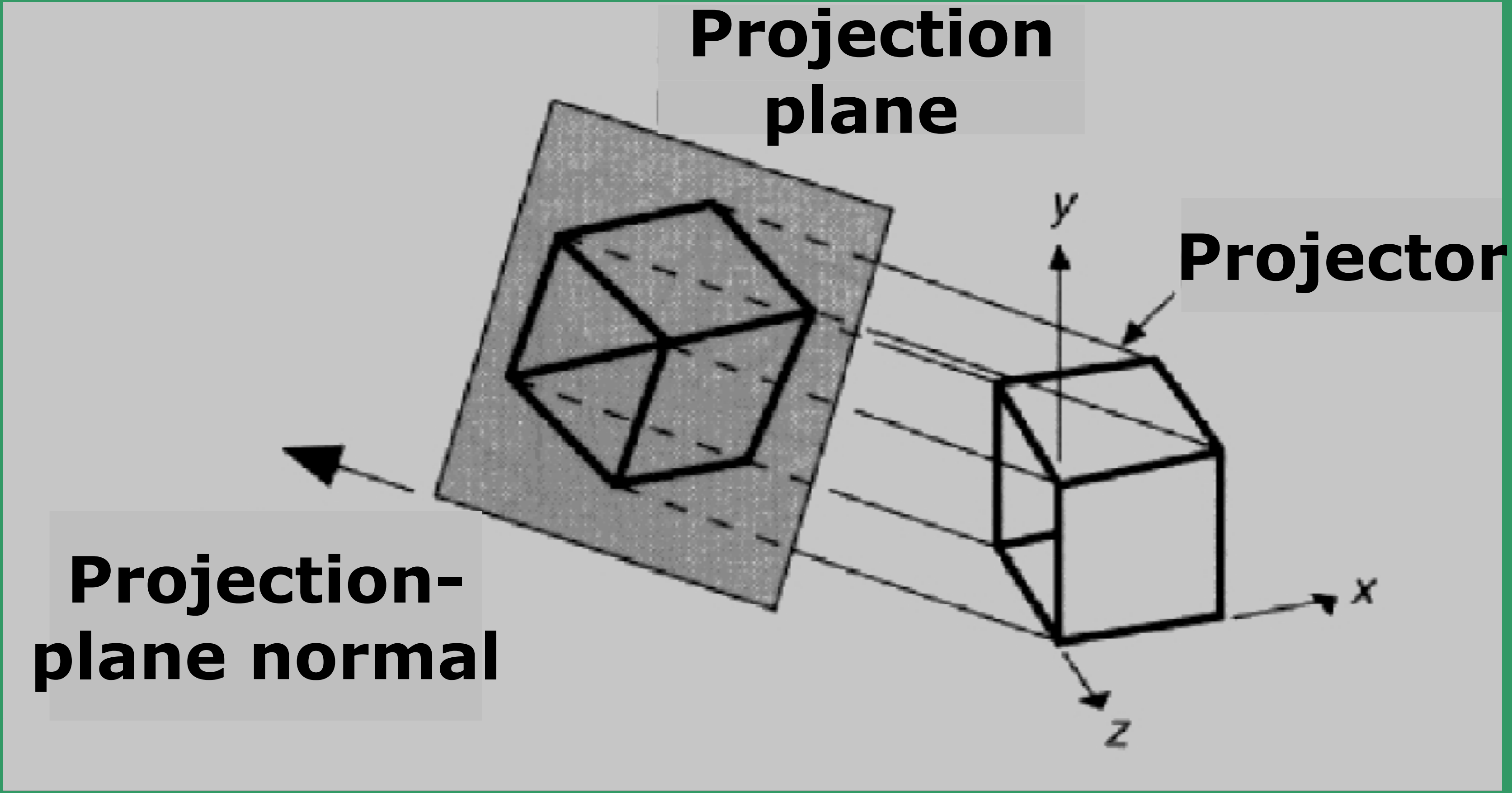
Classification of Geometric Projections




Projection Plane (top view)
Projectors for side view
Projectors for top view
Projection Plane (side view)
Projection Plane (front view)
Projectors for front view

Example of Orthographic Projection

Example of Isometric Projection:



Axonometric orthographic projections use planes of projection that are not normal to a principal axis (they therefore show multiple face of an object.)

Isometric projection: projection plane normal makes equal angles with each principle axis. DOP Vector: [1 1 1].

All 3 axis are equally foreshortened allowing measurements along the axes to be made with the same scale.

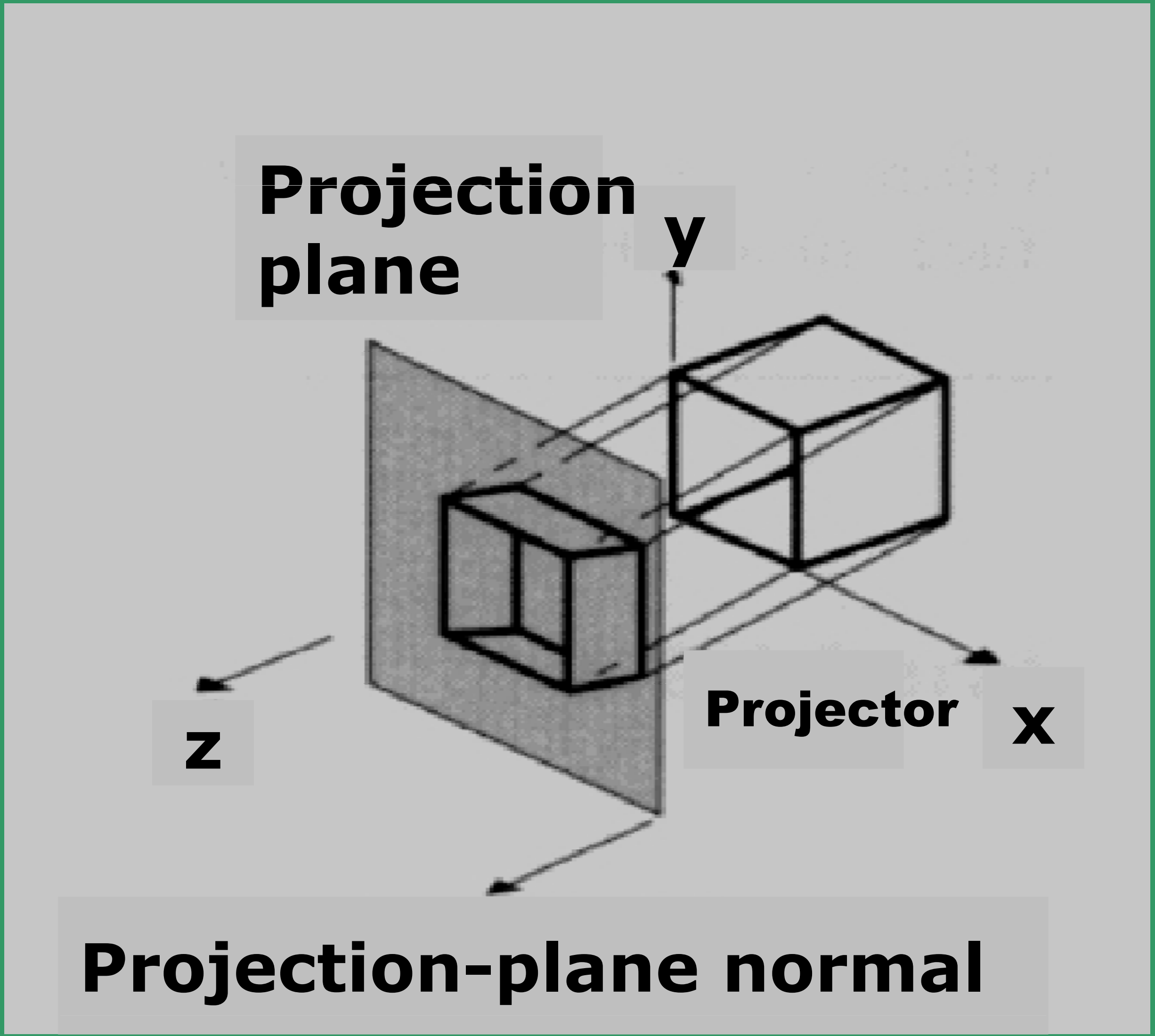
Oblique projections :

projection plane normal and the direction of projection differ.

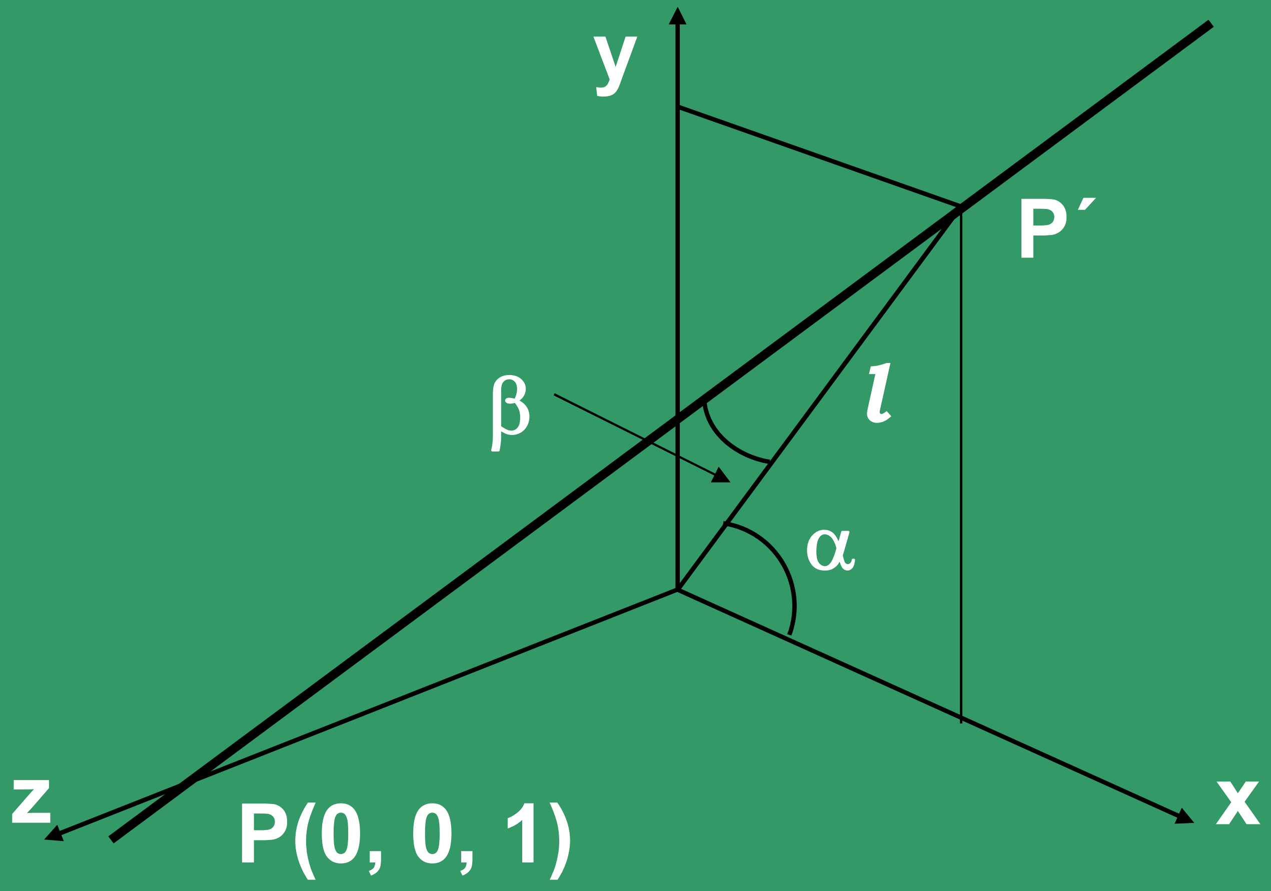
Plane of projection is normal to a Principle axis

Projectors are not normal to the projection plane

Example Oblique Projection



General oblique projection of a point/line:



General oblique projection of a point/line:

Projection Plane: x-y plane; P´ is the projection of P(0, 0, 1) onto x-y plane.

`l´ is the projection of the z-axis unit vector onto x-y plane and α is the angle the projection makes with the x-axis.

When DOP varies, both `l´ and α will vary.

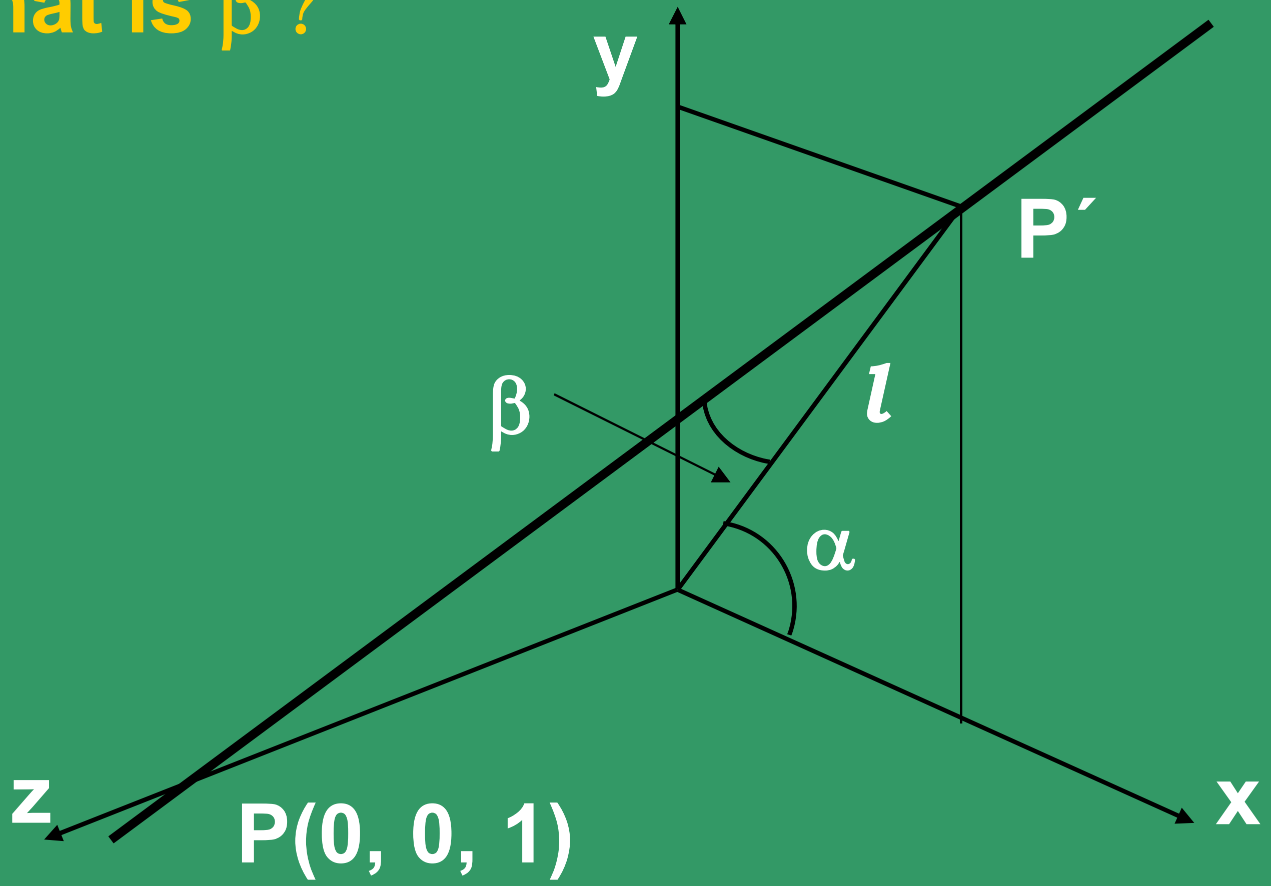
Coordinates of P´: (l *cos* α, l *sin* α, 0).

As given in the figure: DOP is:

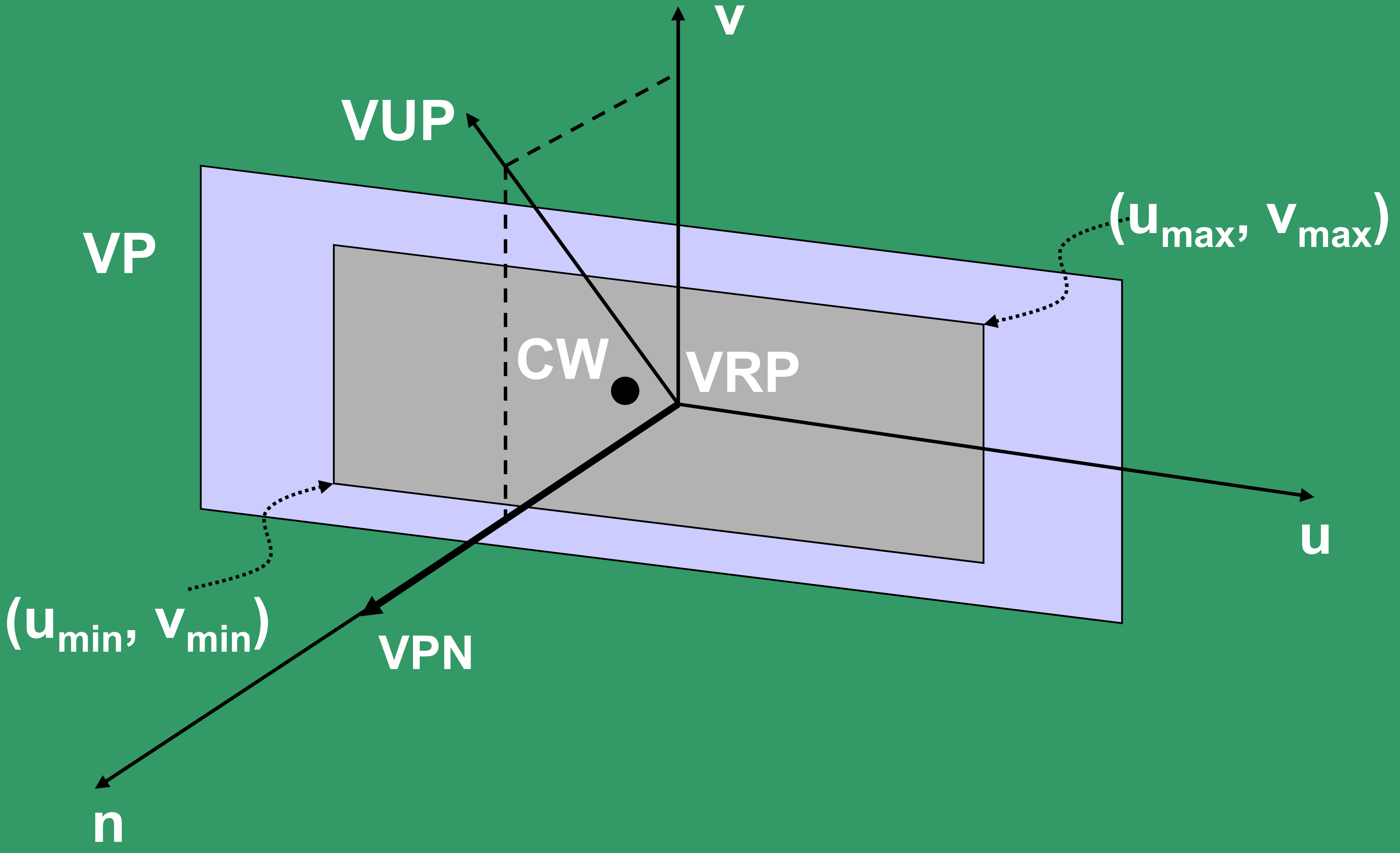
(d_x, d_y, -1) or (l *cos* α, l *sin* α, -1).

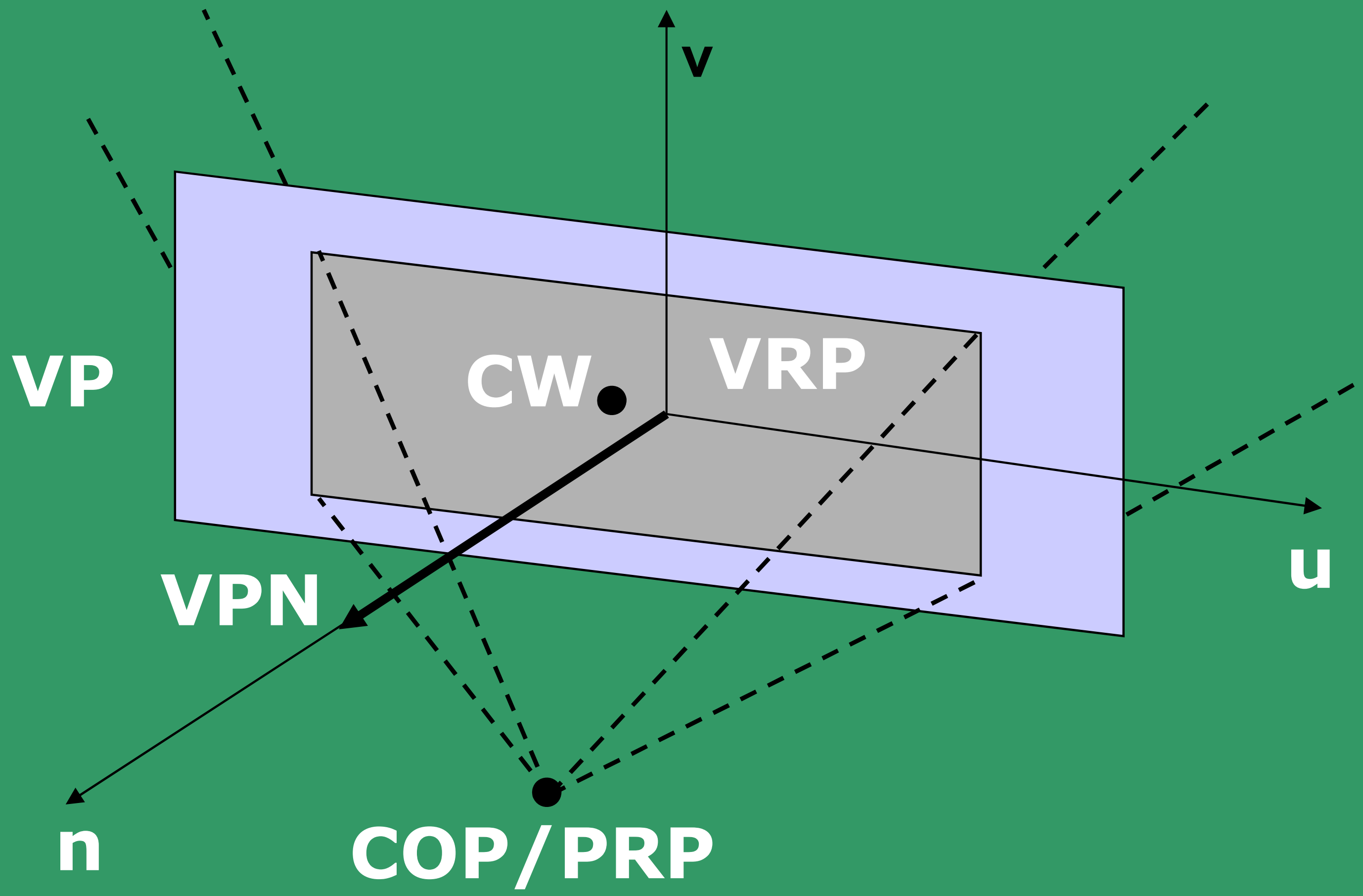
General oblique projection of a point/line:

What is β ?

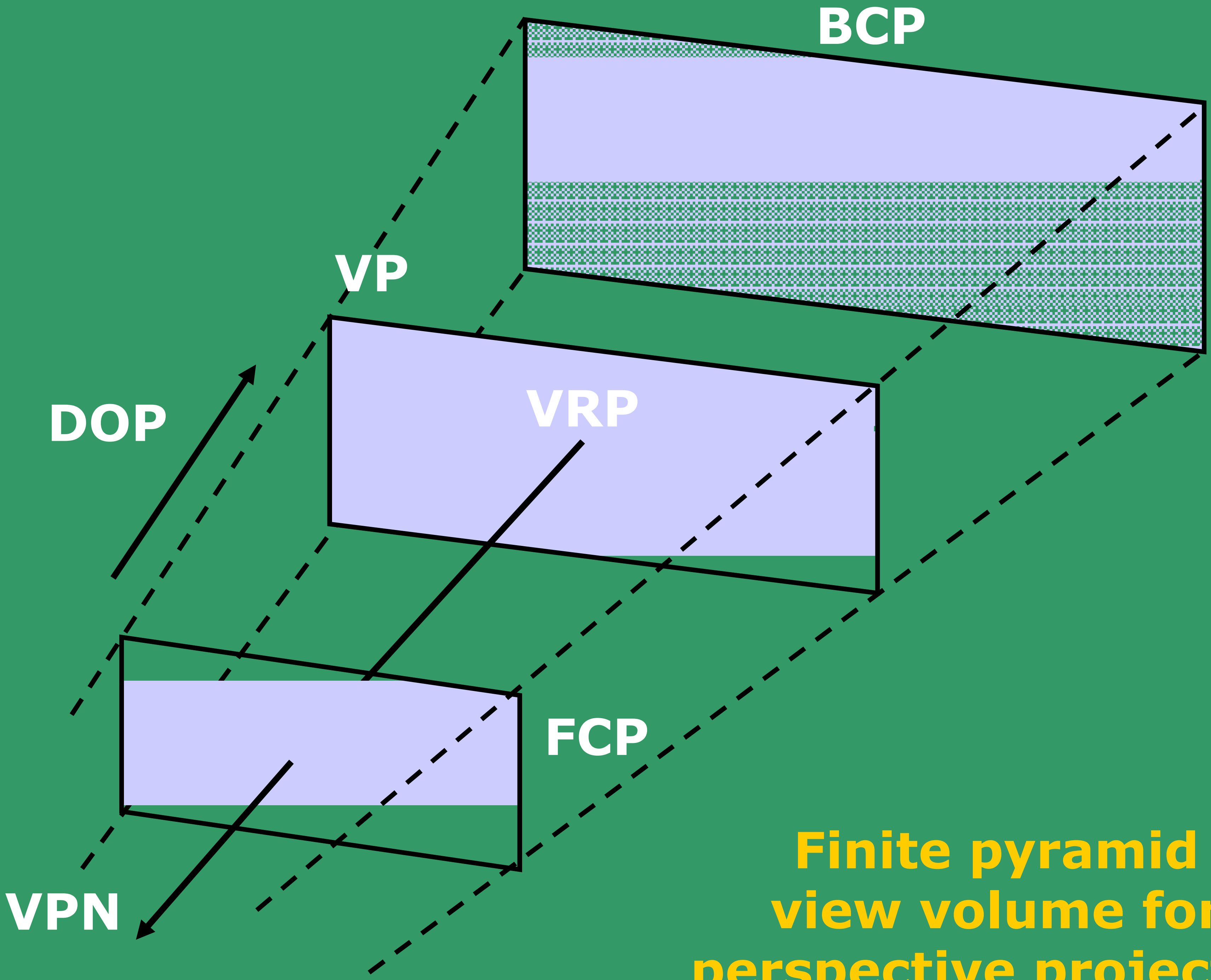


View Specifications:
VP, VRP, VUP, VPN, PRP, DOP, CW, VRC

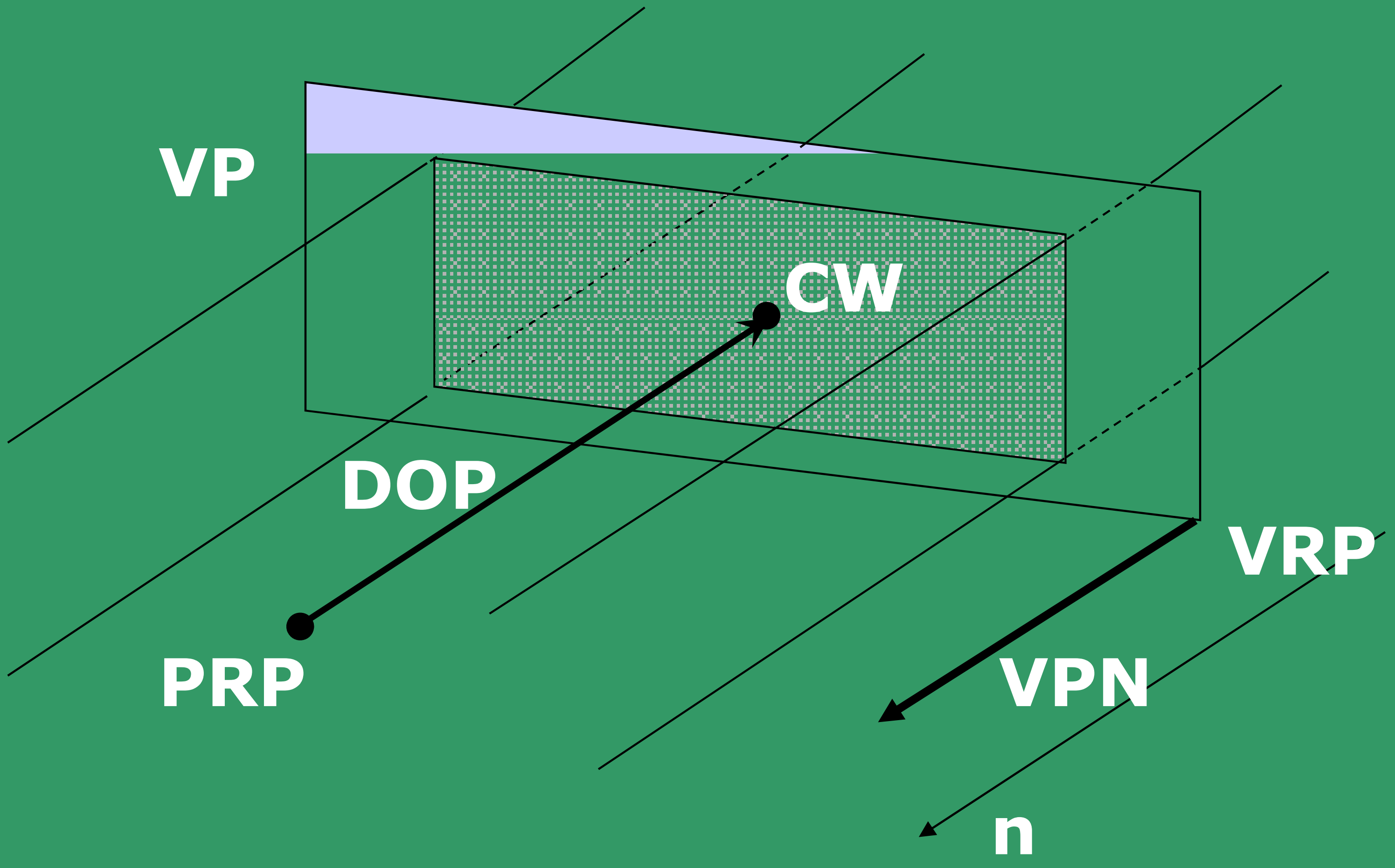




Semi-infinite pyramid view volume for perspective projection

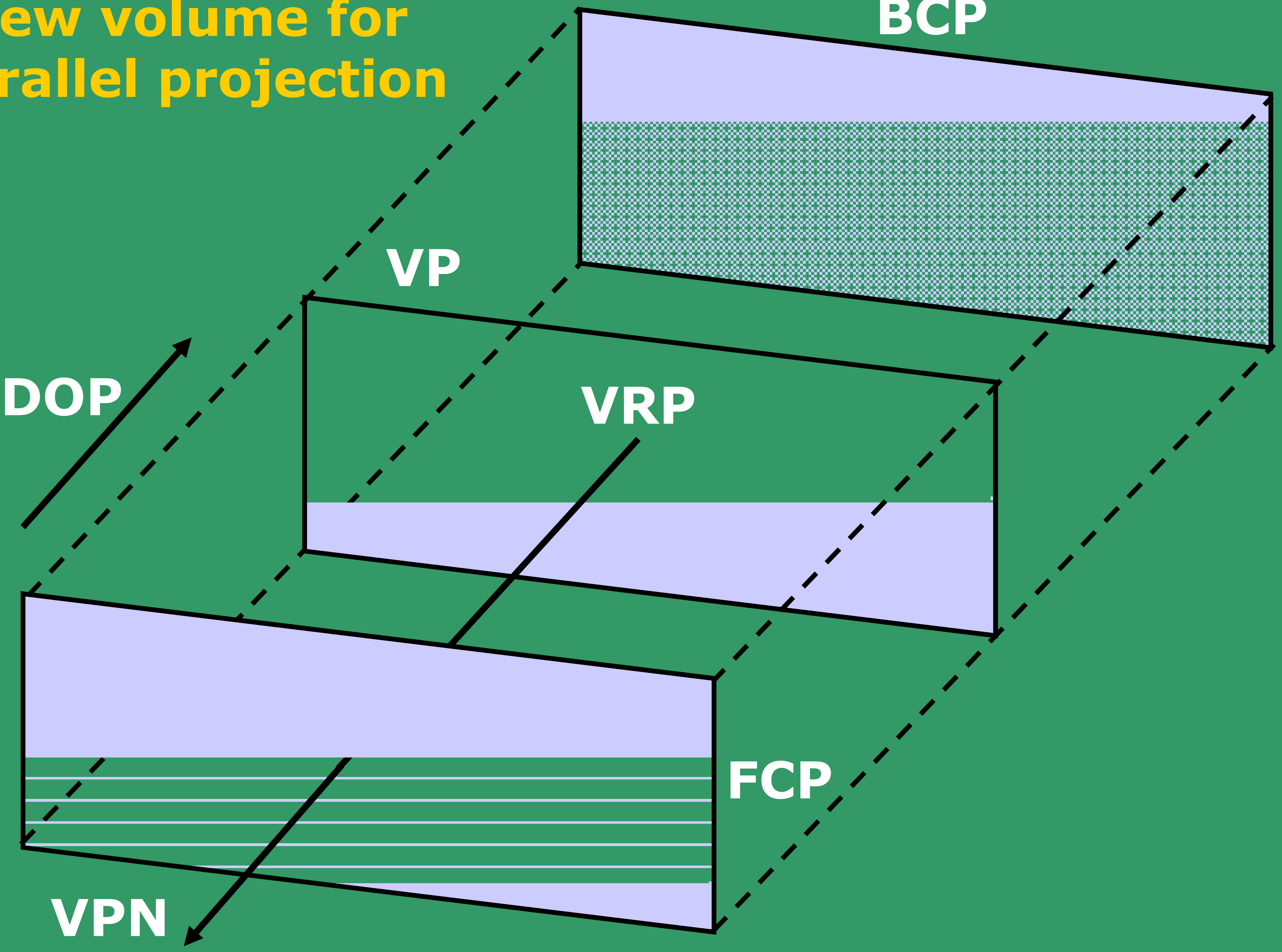


Finite pyramid view volume for perspective projection



Infinite parallelopiped view volume for parallel projection

Finite parallelopiped view volume for parallel projection



Scene Modeling

Lecture Objectives

- At the completion of this module, you will be able to Use the correct transformation order for a desired effect
- Composite modeling transformations
- Create dependent and independent models
- Use GLUT functions to create geometric models

Modeling Transformation Order

- Modeling transformations are post-multiplied together to produce the *current matrix*
  - All vertices are post-multiplied by the current matrix

    ```
    glTranslatef( 4.0, 2.0, 0.0 );
    glRotatef( 90.0, 0.0, 0.0, 1.0 );
    glVertex3f( 1.0, 2.0, 3.0 );
    ```

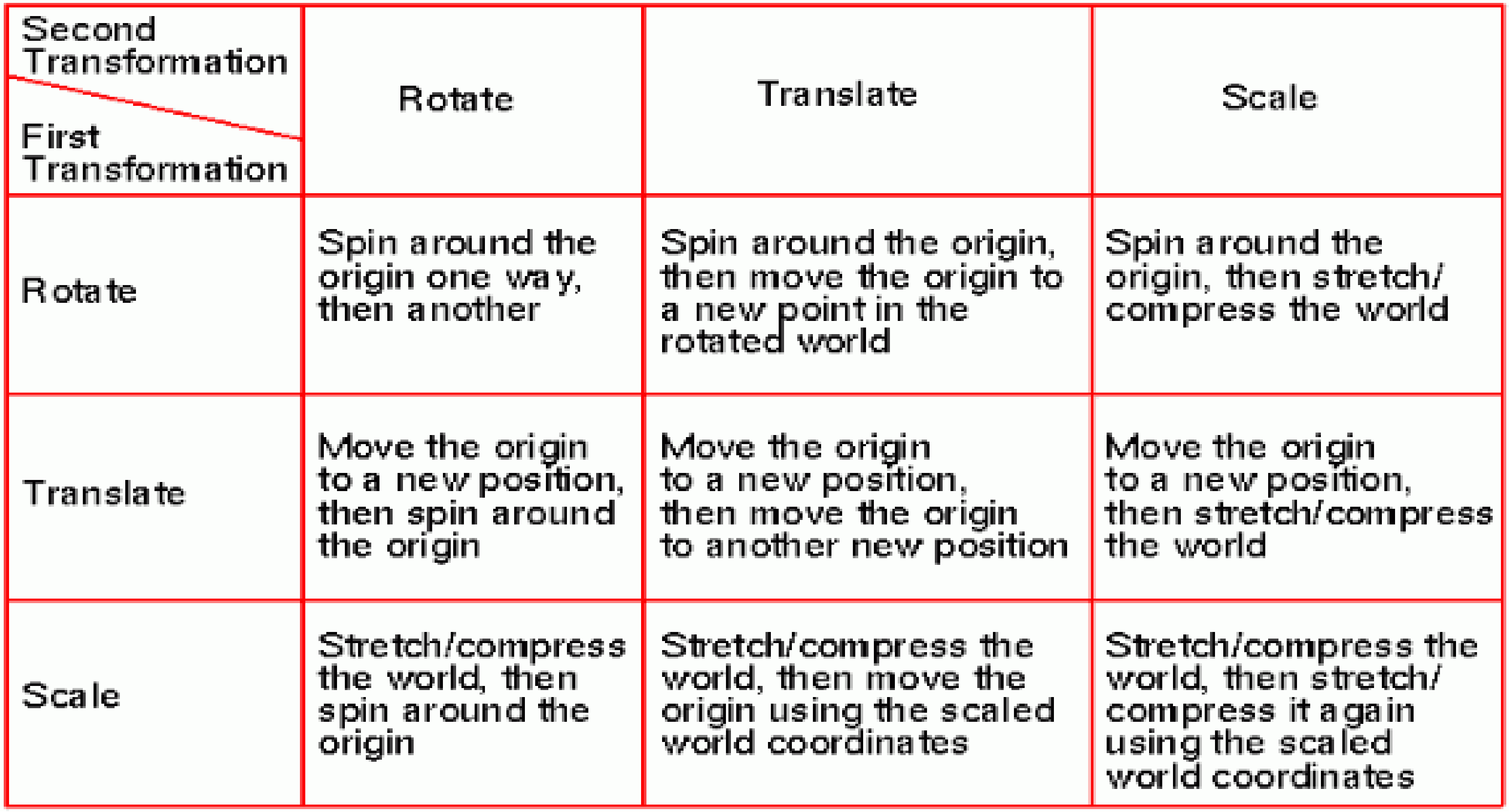
Why Is Order Important?

- Modeling transformations are post-multiplied together to produce the *current matrix*

- All vertices are post-multiplied by the current matrix

```
glTranslatef( 4.0, 2.0, 0.0 );
glRotatef( 90.0, 0.0, 0.0, 1.0 );
glVertex3f( 1.0, 2.0, 3.0 );
```

Effects of Transformation Order



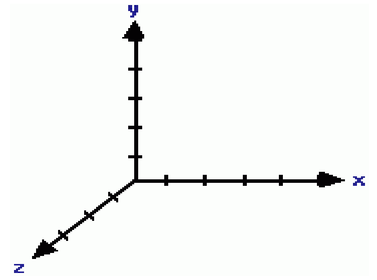
Second Transformation First Transformation	Rotate	Translate	Scale
Rotate	Spin around the origin one way, then another	Spin around the origin, then move the origin to a new point in the rotated world	Spin around the origin, then stretch/ compress the world
Translate	Move the origin to a new position, then spin around the origin	Move the origin to a new position, then move the origin to another new position	Move the origin to a new position, then stretch/compress the world
Scale	Stretch/compress the world, then spin around the origin	Stretch/compress the world, then move the origin using the scaled world coordinates	Stretch/compress the world, then stretch/ compress it again using the scaled world coordinates

Compositing Modeling Transformations

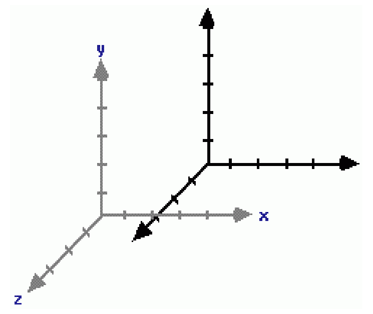
- Think of a moving coordinate system
- Modeling transformations affect the coordinate system, not the objects
- All models are drawn relative to the current coordinate system
- The current coordinate system is represented by the current matrix

Moving Coordinate System

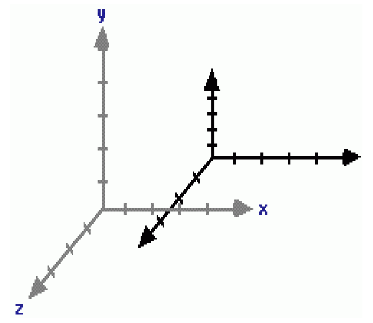
- glPushMatrix();



- glTranslatef(4.0, 2.0, 0.0);

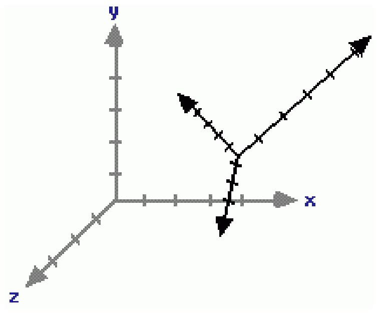


- glScalef(1.0, 0.5, 1.0);

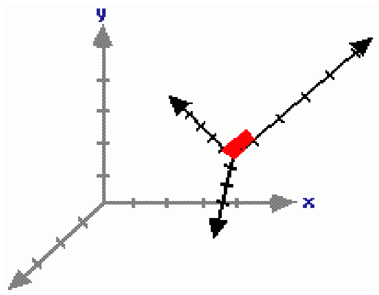


Moving Coordinate System (continued)

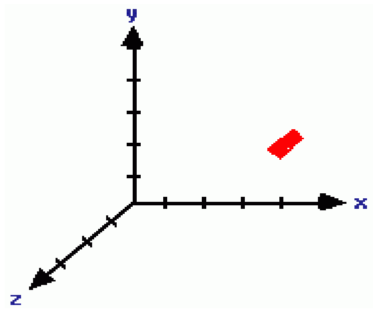
- glRotatef
(45.0, 0.0, 0.0, 1.0);



- draw_unit_square_box();



- glPopMatrix();



Matrix stack

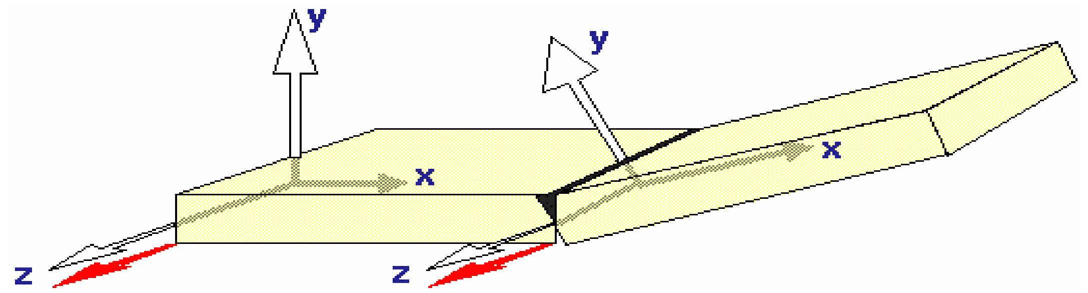
- glPushMatrix()
- glPopMatrix()

Matrix stack

```
Draw_wheel_bolts(){
 draw_wheel();
 for(int i=0;i<5;i++){
 glPushMatrix();
    glRotatef(72.0*i,0.0,0.0,1.0);
    glTranslatef(3.0,0.0,0.0);
    draw_bolt();
 glPopMatrix();
}
```

Composition Example 1: Robot Arm

Draw two boxes to represent the upper and lower segments of a robot arm.



Step 1: Move to the center of the upper arm

Step 2: Draw box

Step 3: Move to the elbow(It means move the coordinate system)

Step 4: Rotate 45 degrees about the elbow(Rotate the coordinates)

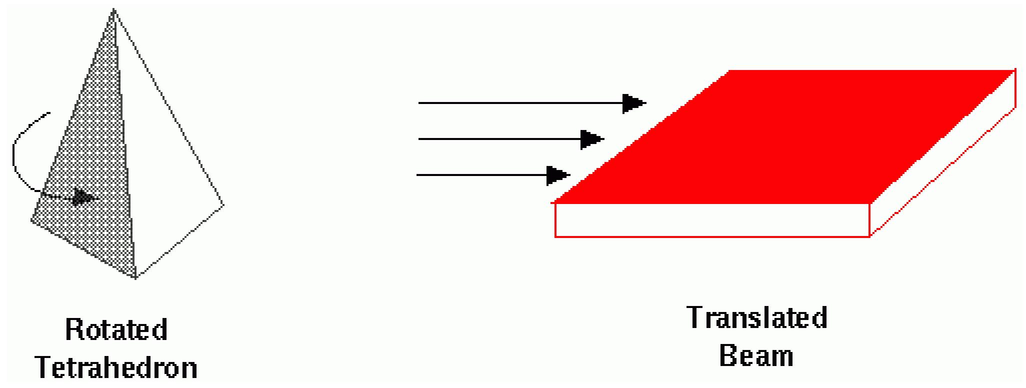
Step 5: Move to the center of the lower arm

Step 6: Draw box

Independent Models

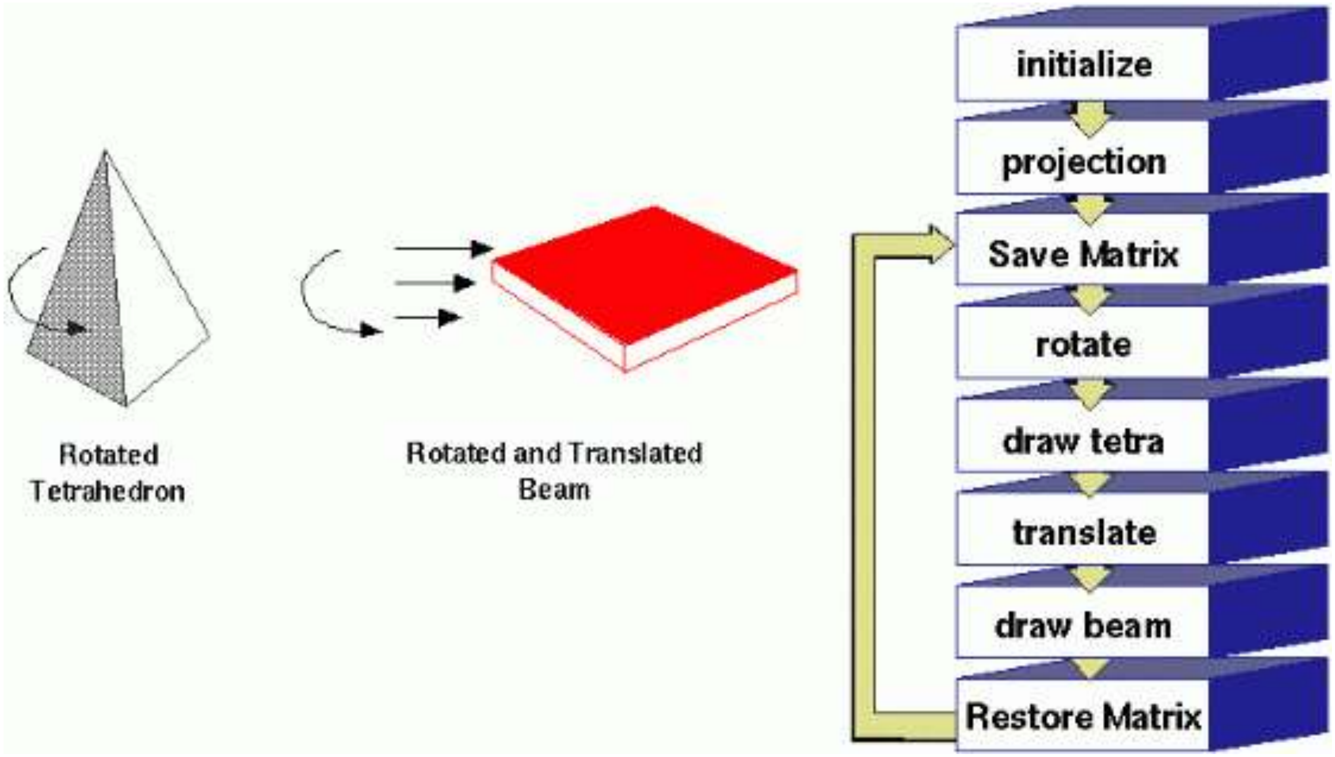
Independent models undergo independent transformations.

- Push to save a coordinate system
- Pop to return to the previously saved coordinate system



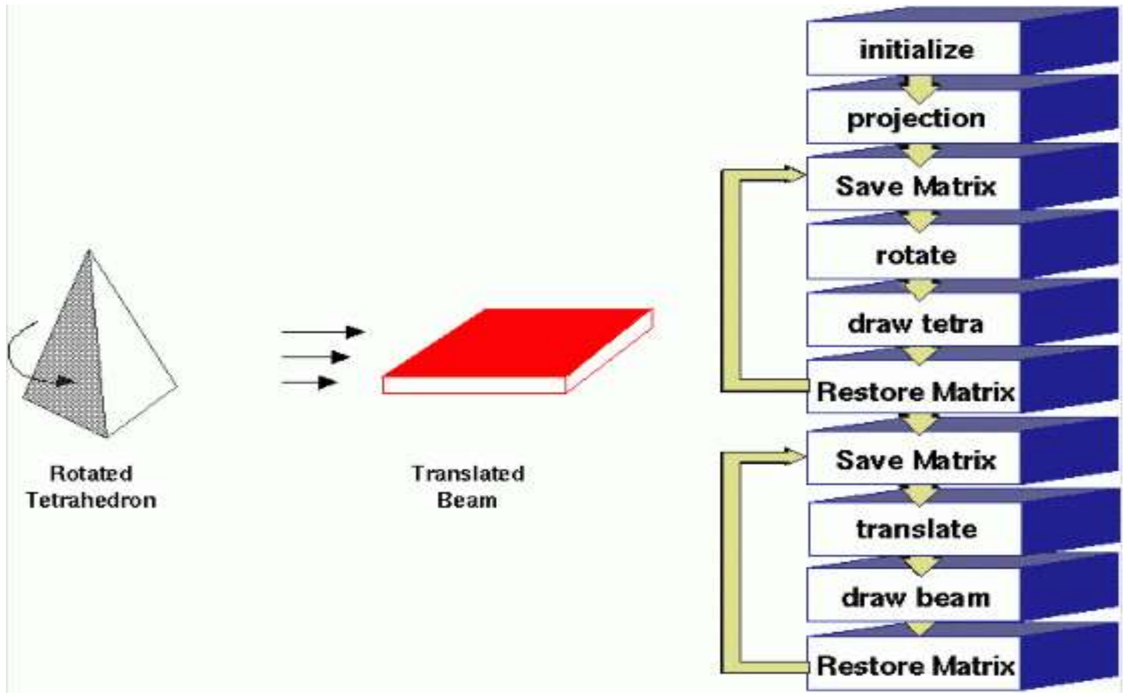
Example of Models That Are Not Independent

Transformations accumulate.

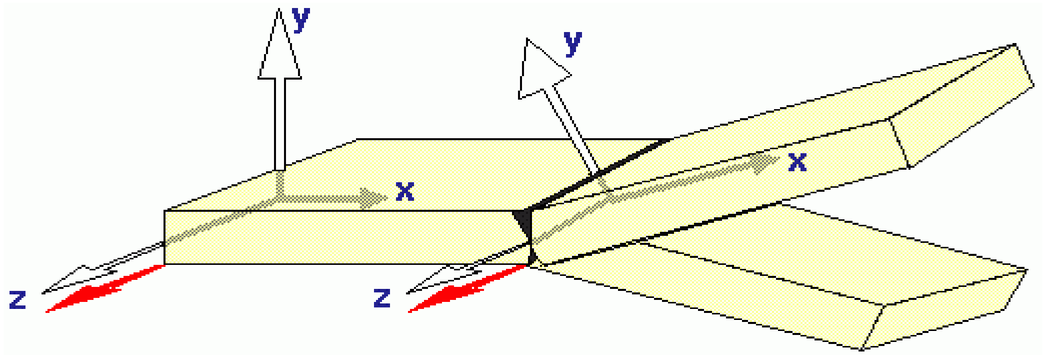


Example of Models That Are Independent

Each model is affected by separate transformations.



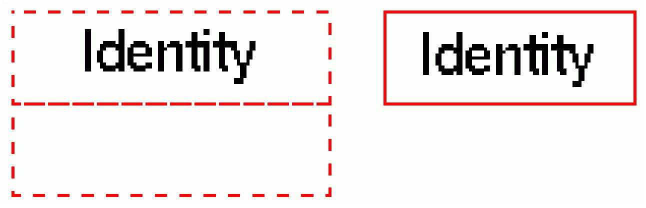
Composition Example 2: Robot Claw



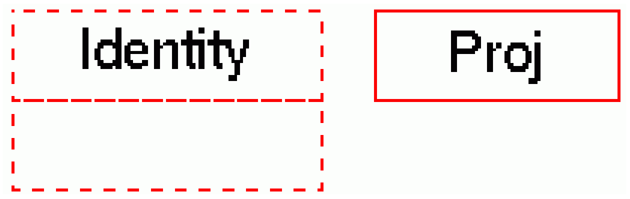
- Step 1: Move to the center of the upper arm
- Step 2: Draw box
- Step 3: Move to the elbow
- Step 4: Rotate 45 degrees about the elbow
- Step 5: Move to the center of the lower arm
- Step 6: Draw box
- Step 7: How do you position the third arm?

Robot Claw Example

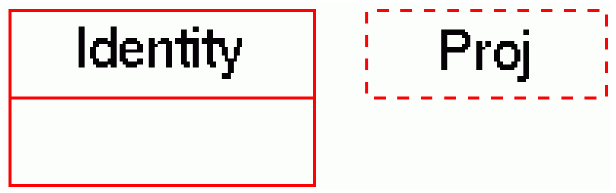
- glMatrixMode (GL_PROJECTION)



- gluPerspective (45.0,1.0,1.0,20.0)

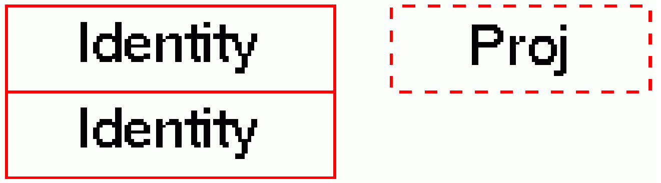


- glMatrixMode (GL_MODELVIEW)

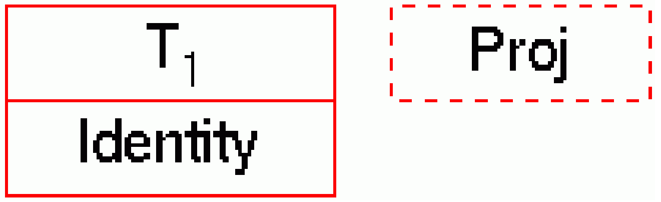


Robot Claw Example (continued)

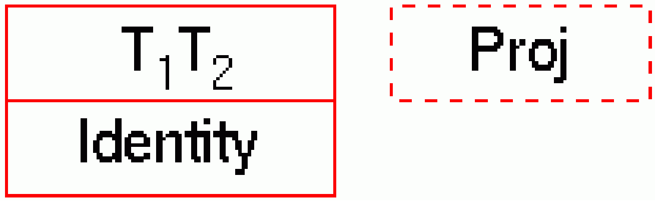
- glPushMatrix()



- glTranslatef (0.0, 0.0, -8.0)

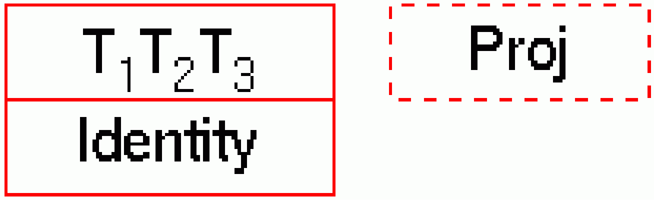


- glTranslatef (1.0, 0.0, 0.0)
/* Draw the upper arm */
WireBox(2.0, 0.4, 1.0);

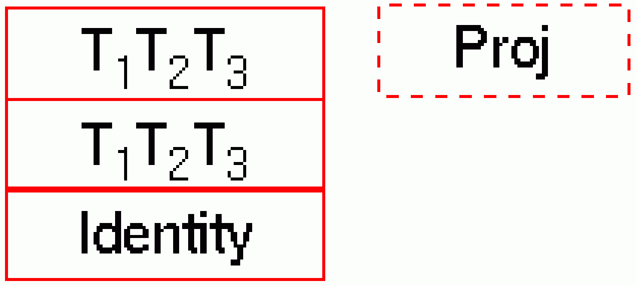


Robot Claw Example (continued)

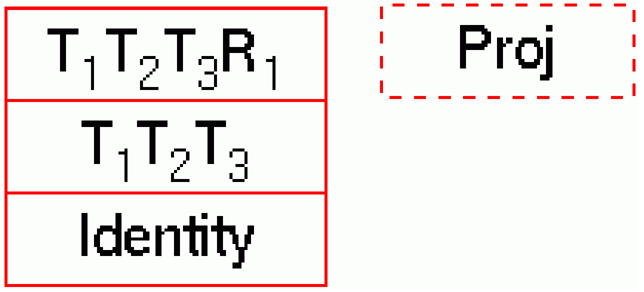
- glTranslatef (1.0, 0.0, 0.0)



- glPushMatrix();

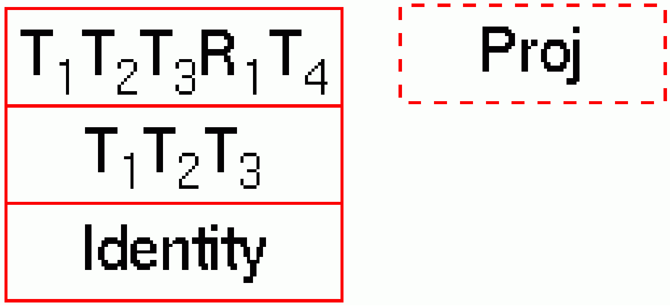


- glRotatef (45.0, 0.0, 0.0, 1.0)

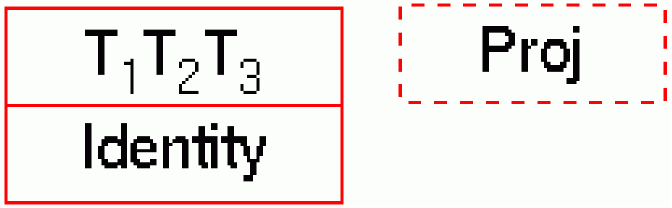


Robot Claw Example (continued)

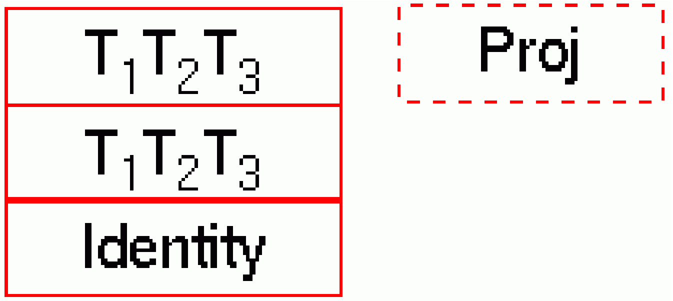
- glTranslatef (1.0, 0.0, 0.0);
  /* Draw the upper claw */
  WireBox(2.0, 0.4, 1.0);



- glPopMatrix()

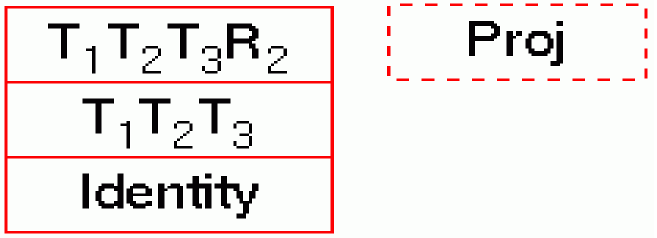


- glPushMatrix();

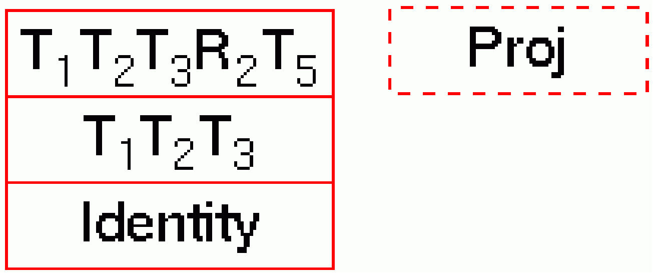


Robot Claw Example (continued)

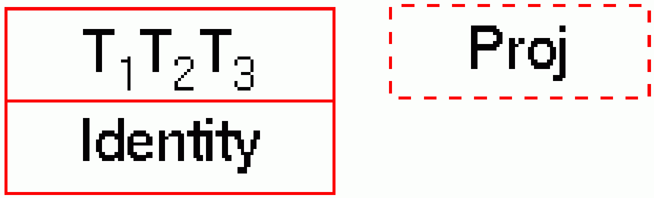
- glRotatef (-45.0, 0.0, 0.0, 1.0)



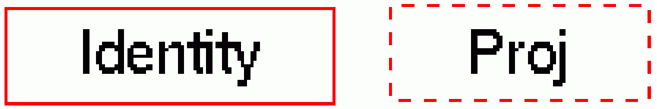
- glTranslatef(1.0, 0.0, 0.0);
  /* Draw the lower claw */
  WireBox(2.0, 0.4, 1.0);



- glPopMatrix()

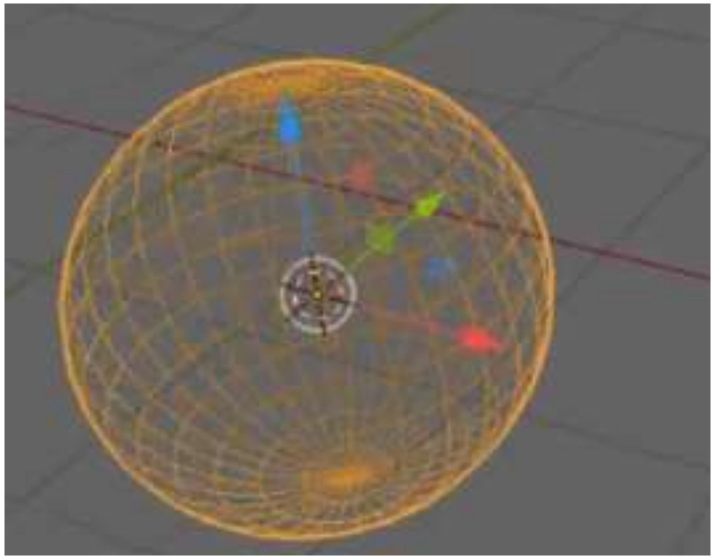


- glPopMatrix()

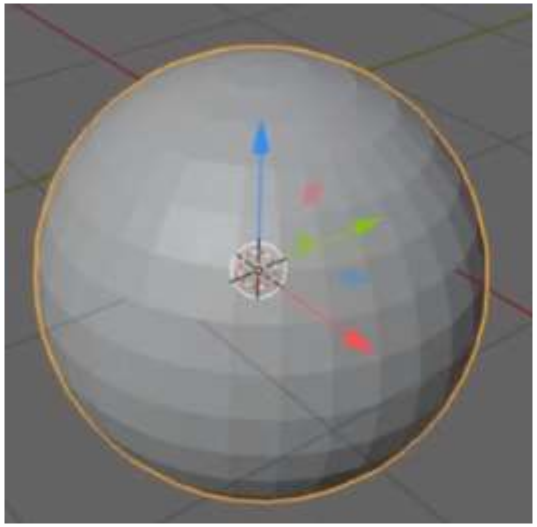


3D Models Provided by the GLUT Library

Function	**Description**
glutSolidSphere glutWireSphere	glutWireSphereRenders a sphere centered at the modeling coordinate origin
glutSolidCube glutWireCube	Renders a cube centered at the modeling coordinate origin
glutSolidCone glutWireCone	Renders a cone oriented along the z axis; the base is at z=0 and the top is at z=height
glutSolidTorus glutWireTorus	Renders a torus centered at the modeling coordinate origin
glutSolidDodecahedron glutWireDodecahedron	Renders a 12-sided regular object centered at the modeling coordinate origin; radius is 1.0



Wire



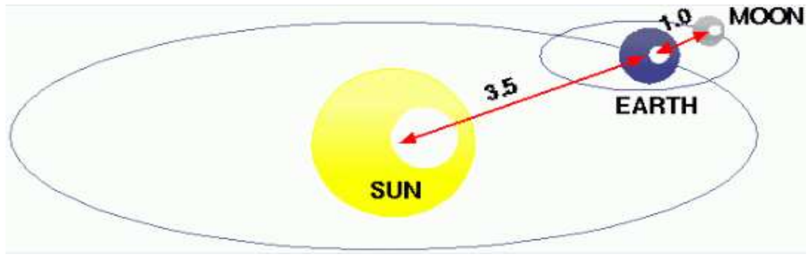
Solid

3D Models Provided by the GLUT Library(Continued)

Function	Description
glutSolidOctahedron glutWireOctahedron	Renders an 8-sided regular object centered at the modeling coordinate origin; radius is 1.0
glutSolidIcosahedron glutWireIcosahedron	Renders a 20-sided regular object centered at the modeling coordinate origin; radius is sqrt(3)
glutSolidTetrahedron glutWireTetrahedron	Renders a 4-sided regular object centered at the modeling coordinate origin; radius is sqrt(3)
glutSolidTeapot glutWireTeapot	Renders a teapot centered at the modeling coordinate origin

Lab: Solar System Matrix Stack Exercise

- Create a simple model of our solar system.



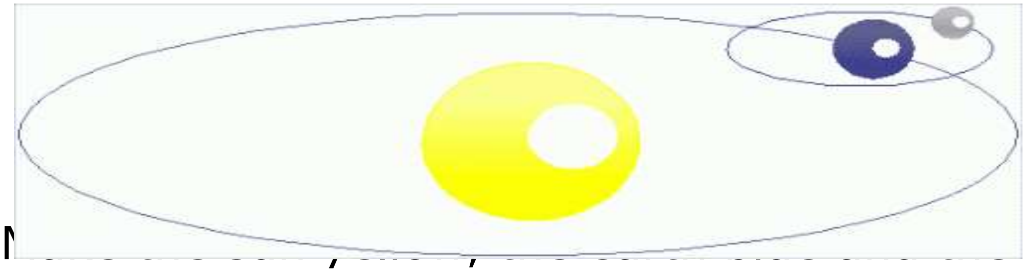
  Use **glutSolidSphere()** to draw the Sun, Earth and Moon. Use 0.7 for the radius of the sun, 0.4 for the earth, and 0.2 for the moon.

- Use modeling transformations to position the spheres as shown above.

  Using the *Worksheet for Solar System Matrix Stack*, list the commands to create the solar system and describe what the matrix stacks will look like.

Lab: Scene Modeling

- Create a program called solar.c that contains the code from the solar system exercise.



  moon gray.

  Add code to toggle between wireframe and solid spheres when the <**SPACE**> key is pressed.

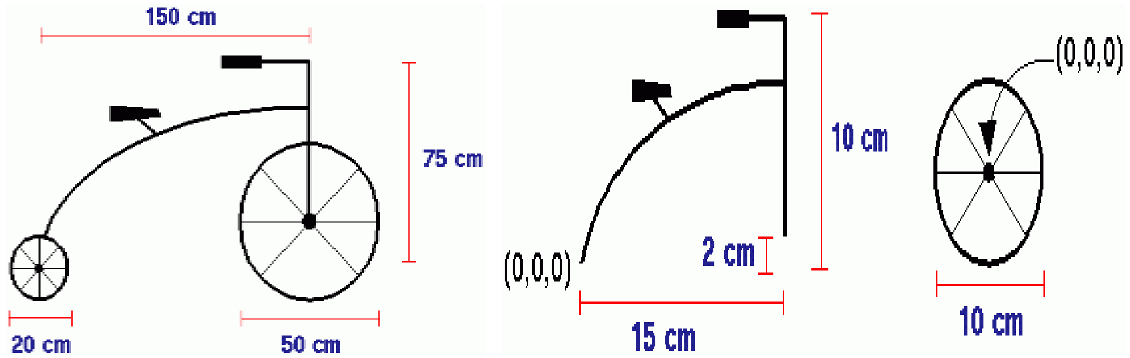
- * Copy your **perspective.c** program to **scene.c**, and add some 3D shapes. Use the shapes available from the GLUT library.

  * Apply modeling transformations to your shapes. Be sure to save and restore the matrix stack when you want independent transformations.

  * Model a simple 3D object, such as a cube or tetrahedron.

Lab: Bike Matrix Stack Exercise (Optional)

- Model an old-style bicycle.



  Models for the frame and a wheel are rendered using the functions drawFrame() and drawWheel().
- Using the *Worksheet for Bike Matrix Stack Exercise* , list the commands to create the bike and describe what the matrix stacks will look like. Include the following
  - Projection and modeling transformations
  - Matrix mode operations
  - Moving the model into the viewing frustum
  - Drawing routines

Order of the commands is more important than function parameters.

Lecture Summary

- Compositing Modeling Transformations
- Compositing Dissimilar Modeling Transformations
- Special Topic: Hierarchical Design

Chapter 6
Animated graphics

```
/* ex6.c */
#include <GL/glut.h>
GLfloat angle= 0.0;
void spin (void) {
angle+= 1.0;
glutPostRedisplay();
}
void display(void) {
glClear (GL_COLOR_BUFFER_BIT);
glLoadIdentity ();
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glRotatef(angle, 1, 0, 0);
glRotatef(angle, 0, 1, 0);
glRotatef(angle, 0, 0, 1);
glutWireCube(2.0);
glFlush(); /* Force update of screen */
}
```

```
void reshape (int w, int h) {
glViewport (0, 0, (GLsizei)w, (GLsizei)h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (60, (GLfloat) w / (GLfloat) h, 1.0, 100.0);
glMatrixMode (GL_MODELVIEW);
}
void keyboard(unsigned char key, int x, int y) {
if (key == 27) exit (0); /* escape key */
}
```

```
int main(int argc, char **argv) {
glutInit(&argc, argv);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("ex6: A rotating cube.");
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutIdleFunc(spin); /* Register the "idle" function */
glutMainLoop();
return 0;
}
/* end of ex6.c */
```
You

void **glutIdleFunc (** ***void (*func)(void));***
glutIdleFunc() registers a callback which will be automatically be called by OpenGL in each cycle of the event loop, **after OpenGL has checked for any events and called the relevant callbacks.**
In ex6.c, the idle function we've registered is called spin():

```
void spin (void) {
angle+= 1.0;
glutPostRedisplay();
}
```

First spin() increments the global variable angle. Then, it calls **glutPostRedisplay(), which tells** OpenGL that the window needs redrawing:

glutPostRedisplay() tells OpenGL that the application is asking for the display to be refreshed.

OpenGL will call the application's display() callback at the next opportunity, which will be during the next cycle of the event loop.

6.2 Double-buffering and animation

As we saw, the rotating cube looks horrible. Why?
The problem is that OpenGL is operating **asynchronously with the refreshing of the display. OpenGL** is pumping out frames too fast: it’s writing (into the frame-buffer) a new image of the cube in a slightly rotated position, **before the previous image has been completely displayed.**

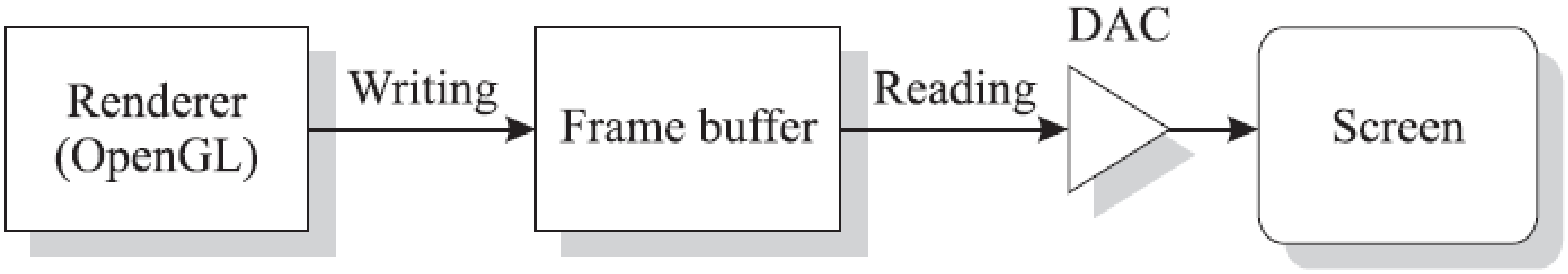


Figure 6.1: Single buffering.

the pixel data is stored in the frame buffer, which is repeatedly read (typically at 60 Hz) by the digital-to-analogue converter (DAC) to control the intensity of the electron beam as it sweeps across the screen, one scan-line at a time.

With a single frame-buffer, the renderer (OpenGL) is writing pixel information into the buffer **at the same time the DAC is reading the information out.**

If the writer and the reader are out of sync, the reader can never be guaranteed to read and display a complete frame

so the viewer always sees images which comprise part of one frame and part of another. This is very disturbing to the eye – and destroys any possibility of seeing smooth animation.

The idea here is that one buffer, called the “back buffer” is only ever **written to by the renderer. The other buffer – the “front buffer” –** is only ever **read by the DAC. The renderer writes its new frame into the back buffer, and when that’s** done, it then requests that the back and front buffers be swapped over.
The **trick is to perform the** swapping while the DAC is performing its **vertical retrace, which is when it’s finished a complete** sweep of its buffer, and is resetting to begin again. There’s enough slack time here to swap the contents of the two buffers over. This method will ensure that the DAC only ever reads and displays a complete frame.

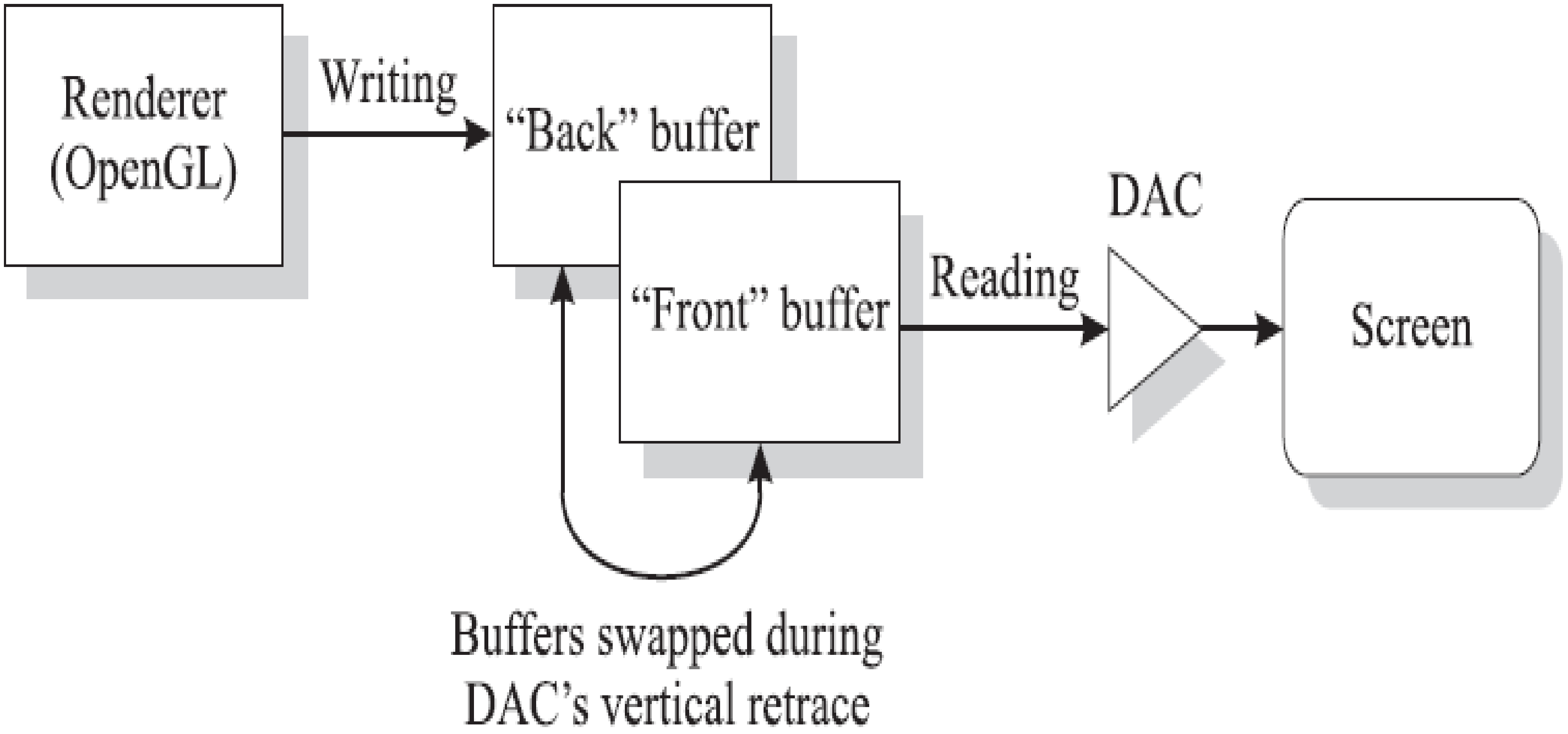


Figure 6.2: Double buffering.

void **glutInitDisplayMode (unsigned int *mode);***
glutInitDisplayMode() sets the current display mode, which which will be used for a window created
using **glutCreateWindow(). mode is:**
• GLUT SINGLE: selects a single-buffered window – which is the default if **glutInitDisplay-Mode isn't called;**
• GLUT DOUBLE: selects a double-buffered window;

void **glutSwapBuffers (void);**
glutSwapBuffers() swaps the back buffer with the front buffer, at the next opportunity, which is normally the next vertical retrace of the monitor. The contents of the new back buffer (which was the old front buffer) are undefined.

Exercise: smooth the cube

Edit your copy of ex6.c as follows:

- In main(), after the call to **glutInit(), insert a call to glutInitDisplayMode() to select a** double-buffered window;
- In display(), after the call to **glutWireCube(), insert a call to glutSwapBuffers().**
- Also, **remove the call to glFlush(). We don't need that anymore, since it gets called internally by glutSwapBuffers(). And if we leave glFlush() in the code, not only will its effect be redundant,** but it'll also slow the program down.

Chapter 9
Viewing

• First, we specify the position and orientation of the camera, using **gluLookAt().**

• Second, we decide what kind of picture we'd like the camera to create. Usually, for 2D graphics we'll use an orthographic (also known as "parallel") view using **glOrtho()). For 3D viewing,** we'll usually want a perspective view, using **gluPerspective().**

• Finally, we describe how to map the camera's image onto the display screen, using **glViewport()**.

9.1 Controlling the camera

Let’s look again at the OpenGL viewing pipeline, in Figure 9.1.

We set the position and orientation of the OpenGL camera, as shown in Figure 9.2, using **gluLookAt():**

```
void gluLookAt (    GLdouble eyex,
                    GLdouble eyey,
                    GLdouble eyez,
                    GLdouble centerx,
                    GLdouble centery,
                    GLdouble centerz,
                    GLdouble upx,
                    GLdouble upy,
                    GLdouble upz );
```

If you don’t call **gluLookAt(), the OpenGL camera is given some default settings:**

- it’s located at the origin, (0, 0, 0);
- it looks down the negative Z axis;
- its “up” direction is parallel to the Y axis.

This is the same as if you had called **gluLookAt() as follows:**

```
gluLookAt (0.0, 0.0, 0.0, /*camera position*/
0.0, 0.0, -1.0, /* point of interest */
0.0, 1.0, 0.0); /* up direction */
```

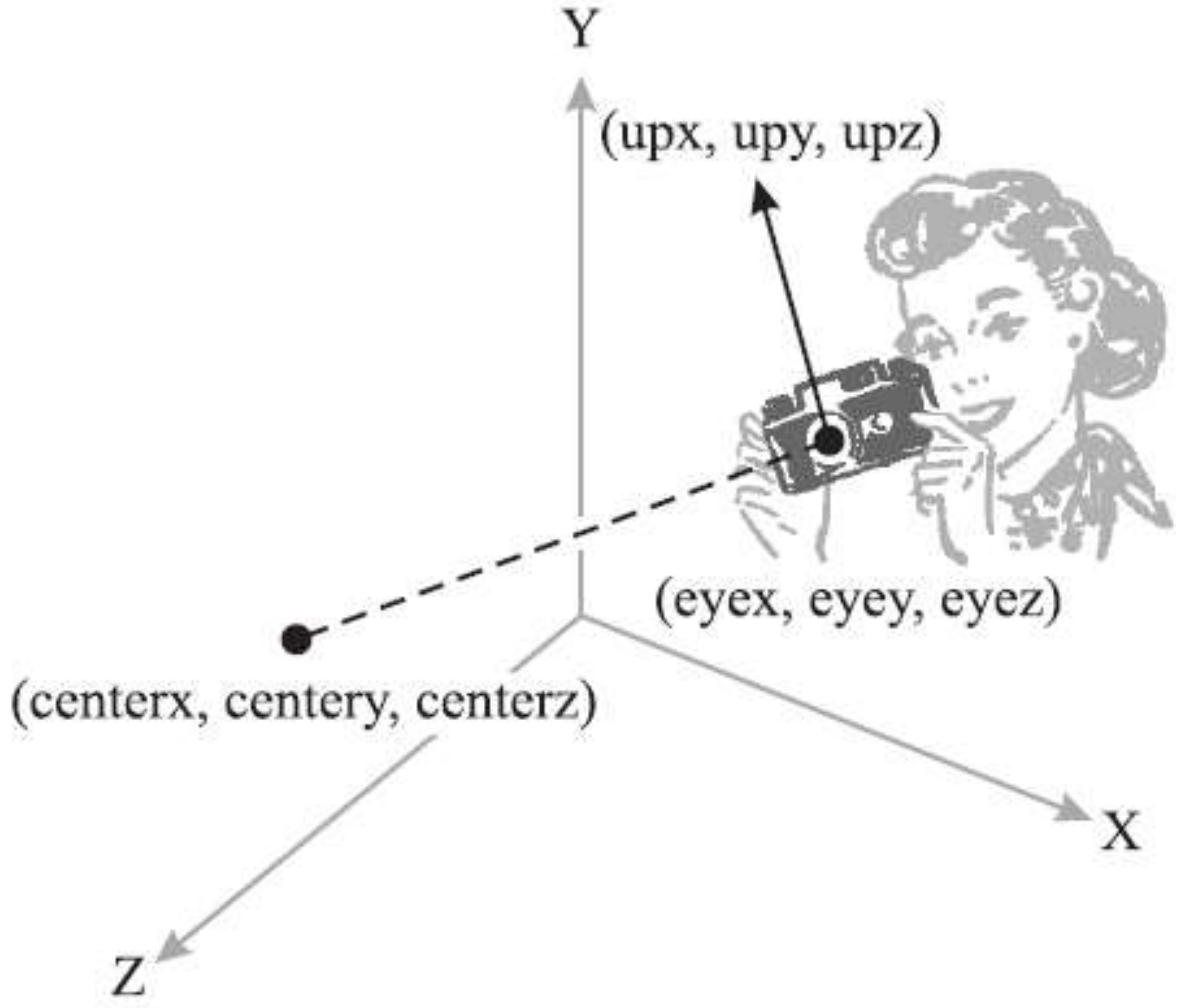


Figure 9.2: The OpenGL camera.

9.2 Projections

specify what kind of image we want. This is done using the **projection matrix, P. OpenGL applies the projection** transformation **after it has applied the modelview transformation.**

9.2.1 The view volume

Consider the real world camera analogy, in which we choose:

- the lens type (wide-angle, telephoto etc.).
- The lens affects the field of view
- what portion of the 3D world will appear within
- the bounds of final image.

The volume of space which eventually appears in the image is known as the **view volume (or view frustum).**

9.2.2 Orthographic projection

glOrtho() creates a matrix for an orthographic projection, and post-multiplies the current matrix (which is normally the projection matrix) by it:

```
void glOrtho (     GLdouble left,
                   GLdouble right,
                   GLdouble bottom,
                   GLdouble top,
                   GLdouble near,
                   GLdouble far );
```

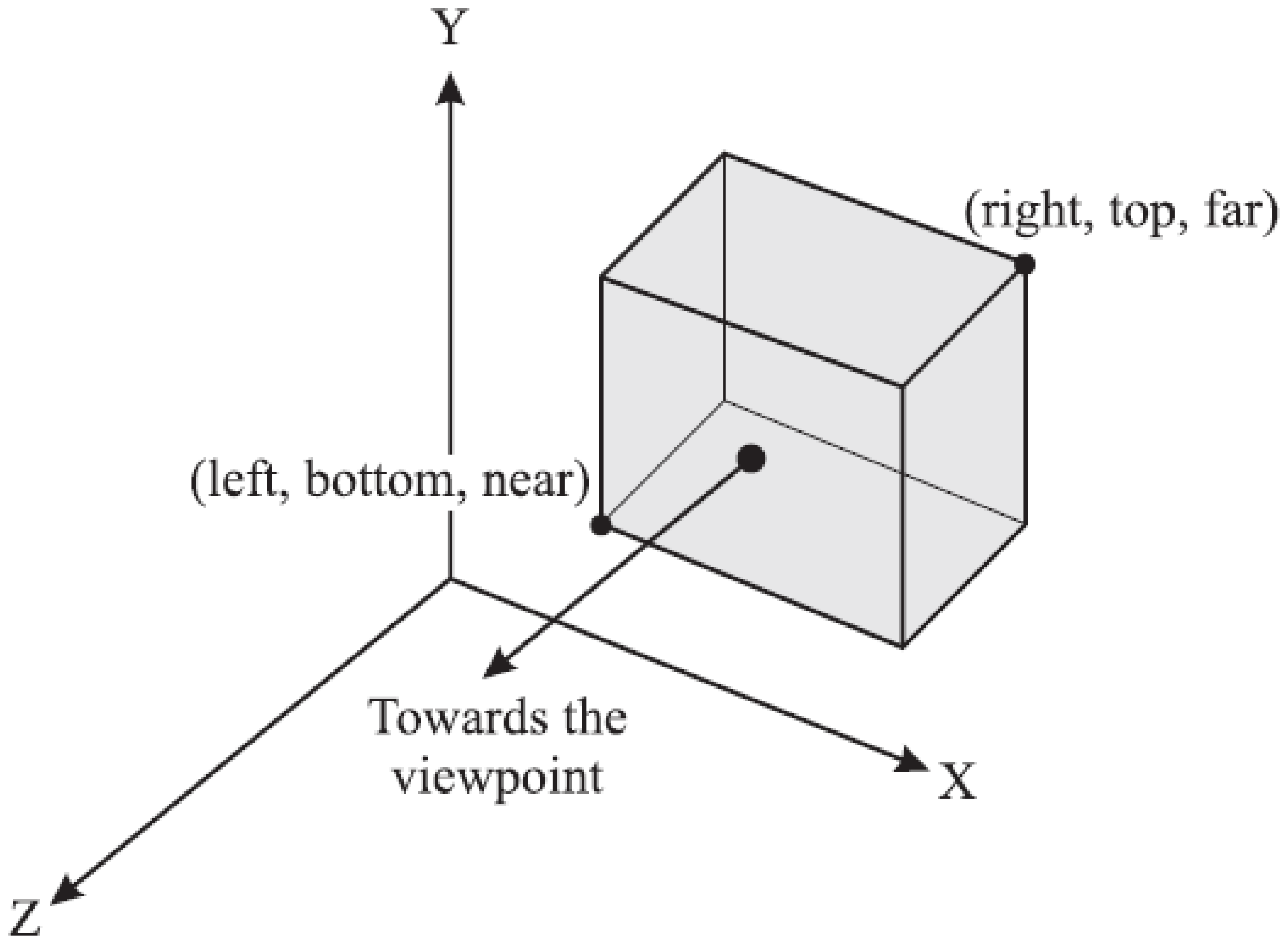


Figure 9.3: The orthographic viewing voulme specified by **glOrtho**.

9.2.3 Perspective projection
gluPerspective() creates a matrix for a perspctive projection, and post-multiplies the current matrix
(which will normally be the projection matrix) by it:
void **gluPerspective (GLdouble *fovy,***
GLdouble *aspect,*
GLdoublea *near,*
GLdouble *far);*

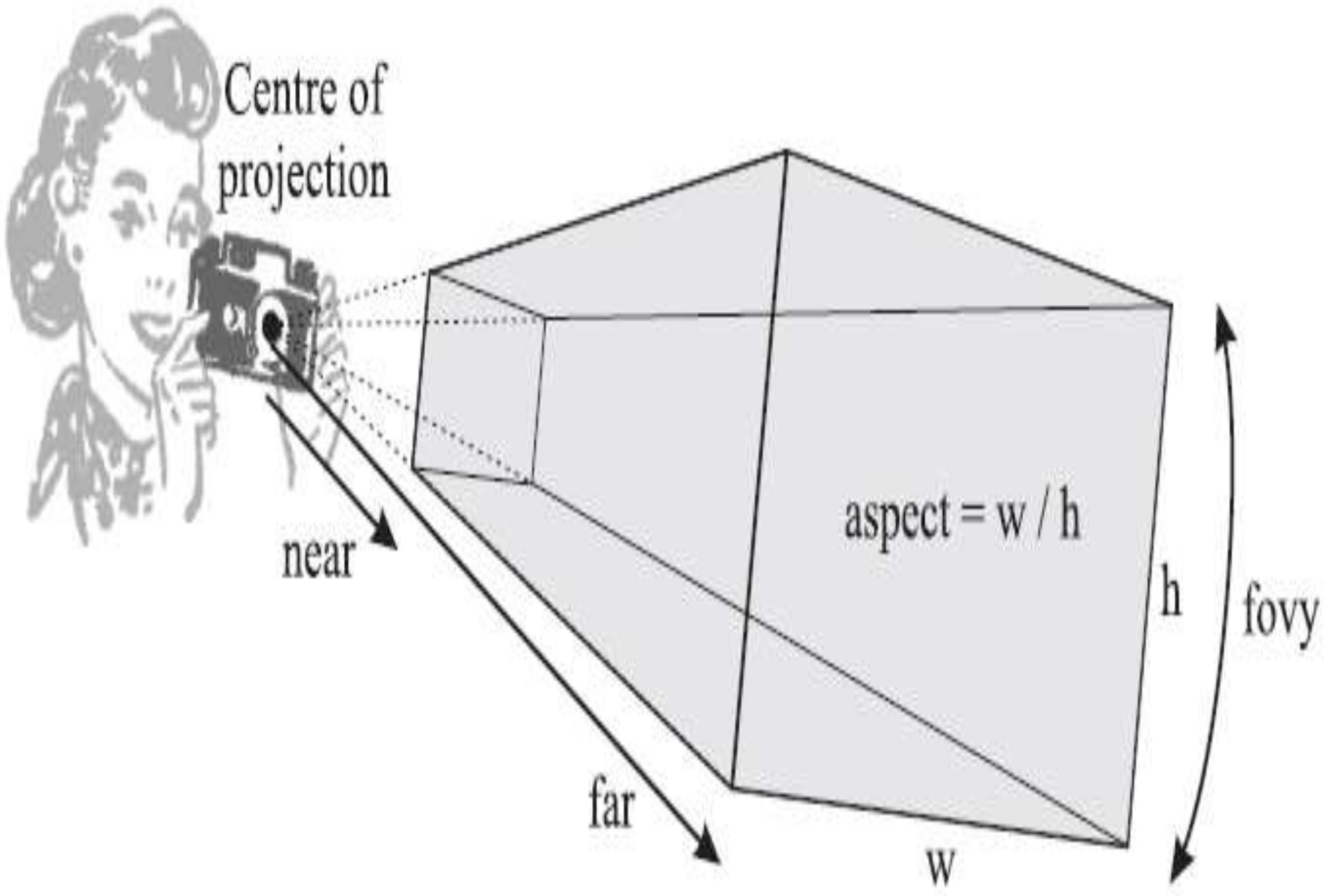


Figure 9.4: The perspective viewing frustum specified by **gluPerspective**.

9.3 Setting the viewport
glViewport() sets the position and size of the viewport – the rectangular area in the display window
in which the final image is drawn, as shown in Figure 9.5:

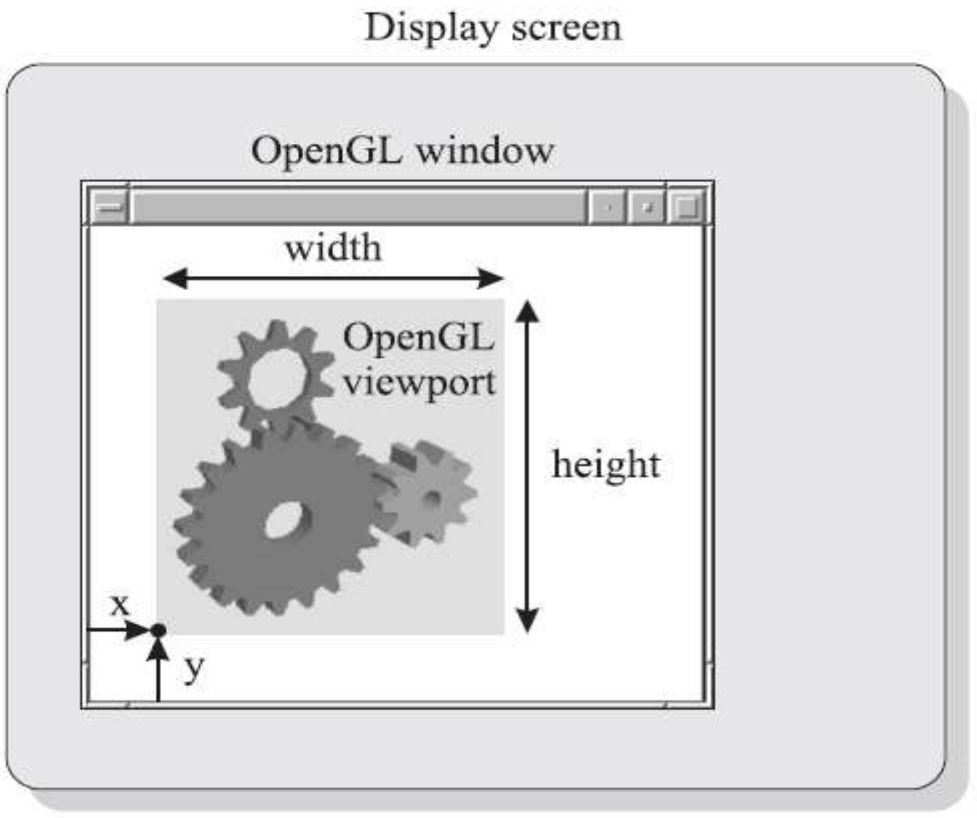


Figure 9.5: How the viewport is defined.

void **glViewport** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

- x and y specify the lower-left corner of the viewport, and width and height specify its width
- height. If a viewport is not set explicitly it defaults to fill the entire OpenGL window.

9.4 Using multiple windows
Most OpenGL programs use a single drawing window. GLUT does support the use of multiple windows simultaneously.
During execution of an OpenGL program, all rendering appears on the **current window**. By default, the current window is always the most recently created window (by **glutCreateWindow()**). If you want to use multiple windows,
first create each window and note the window identifier returned by each call to **glutCreateWindow()**. Then,

void **glutSetWindow** (int *window*);
select a window to render using **glutSetWindow()**:
To find out which window is currently selected, call **glutSetWindow()**:
int **glutGetWindow** (void);
You can also destroy windows, using:
void **glutDestroyWindow** (int *window*);
Obviously, you can’t refer to the window identifier for a window which has been destroyed.

Chapter 10
Drawing pixels and images

Using object coordinates as pixel coordinates

```
glutInitWindowSize (360, 335);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Pixel world");
```

It's usual to place the projection specification in the reshape function:

```
void reshape (int width, int height)
}
glViewport (0, 0, (GLsizei) width, (GLsizei) height);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (0.0, (GLfloat) width, 0.0, (GLfloat) height, -1.0, 1.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
```

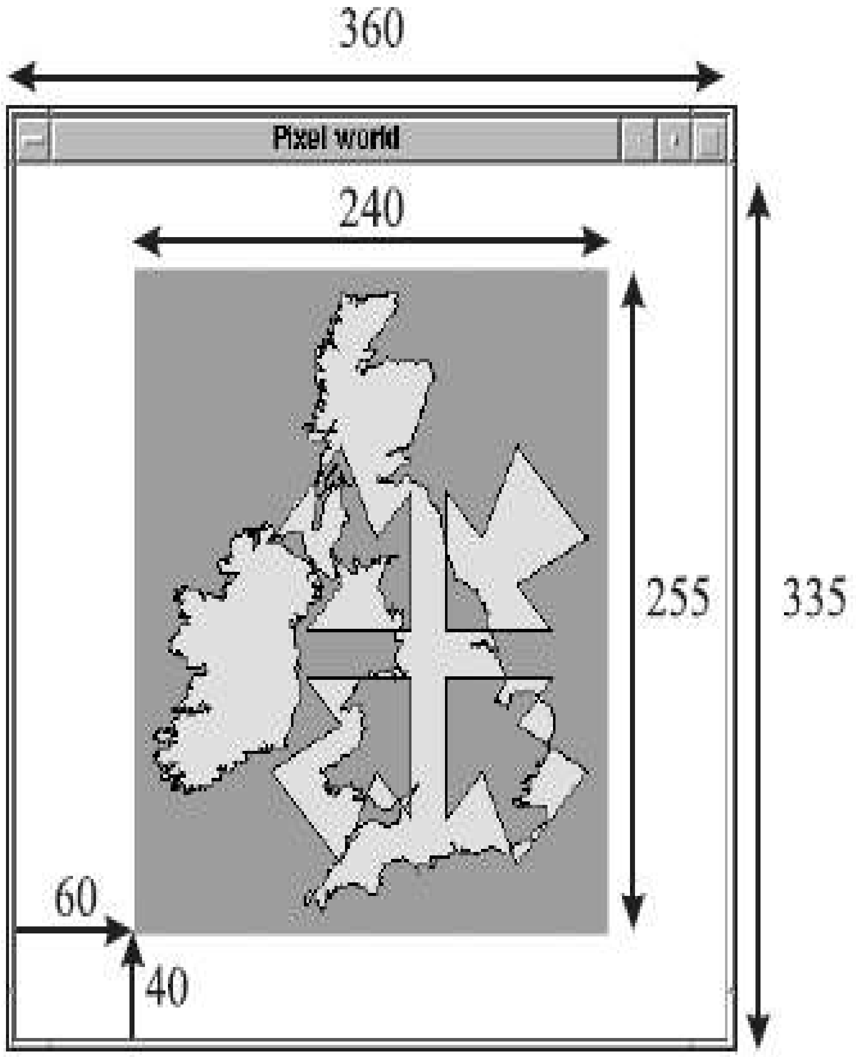


Figure 10.1: The pixel rectangle drawn by **glDrawPixels** at the current raster position.

10.2 Setting the pixel drawing position

The function **glRasterPos3f()** sets the **current raster position** – the pixel position at which the next pixel rectangle specified using **glDrawPixels()** will be drawn:

void **glRasterPos3f** (GLfloat *x*,GLfloat *y*,GLfloat *z*);

The position (x, y, z) is expressed in object coordinates, and is transformed in the normal way by the modelview and projection matrices.

10.3 Drawing pixels

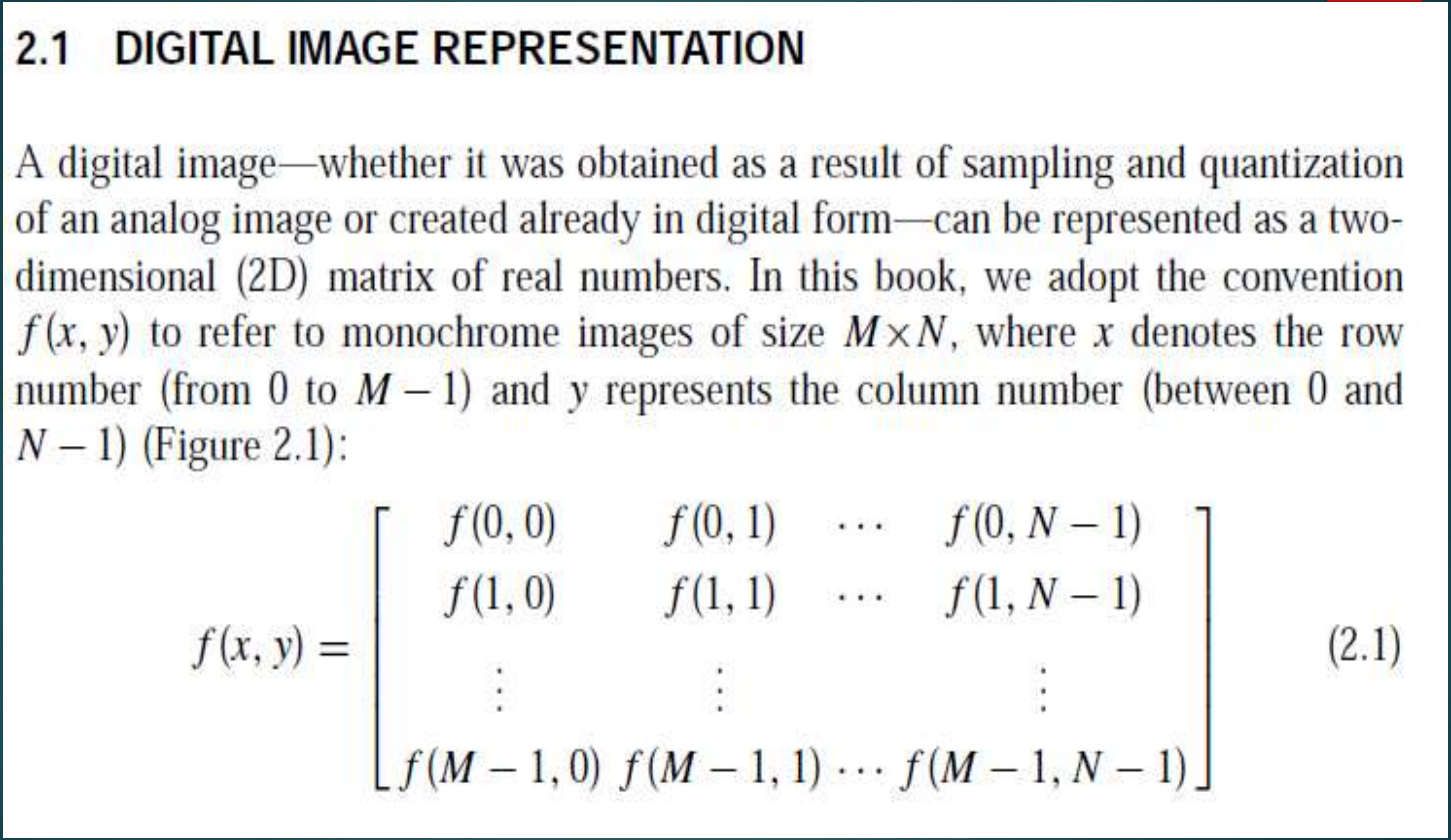
glDrawPixels draws a rectangle of pixels, with width pixels horizontally, and height pixels vertically.

void **glDrawPixels** (GLsizei *width*,GLsizei *height*,
                              GLenum *format*, GLenum *type*,
                               const GLvoid **pixels*);

pixels is a pointer to an array containing the actual pixel data. Because pixel data can be encoded in several different ways, the type of pixels is a (void *) pointer. format and type specify thepixel data encoding: normally format will be GL RGB,

```
#define WIDTH 240
#define HEIGHT 255
GLfloat image[WIDTH][HEIGHT][3]; /* pixel data, R,G,B
*/
/* code omitted to write pixel values into ’image’ */
void display (void)
{
glClear(GL_COLOR_BUFFER_BIT);
glRasterPos3f(60.0, 40.0, 0.0);
glDrawPixels(WIDTH, HEIGHT, GL_RGB, GL_FLOAT,
image);
}
```

Vector and raster graphics



2.1 DIGITAL IMAGE REPRESENTATION

A digital image—whether it was obtained as a result of sampling and quantization of an analog image or created already in digital form—can be represented as a two-dimensional (2D) matrix of real numbers. In this book, we adopt the convention $f(x, y)$ to refer to monochrome images of size $M \times N$, where x denotes the row number (from 0 to $M - 1$) and y represents the column number (between 0 and $N - 1$) (Figure 2.1):

$$f(x, y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0, N-1) \\ f(1,0) & f(1,1) & \cdots & f(1, N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1, N-1) \end{bmatrix} \qquad (2.1)$$

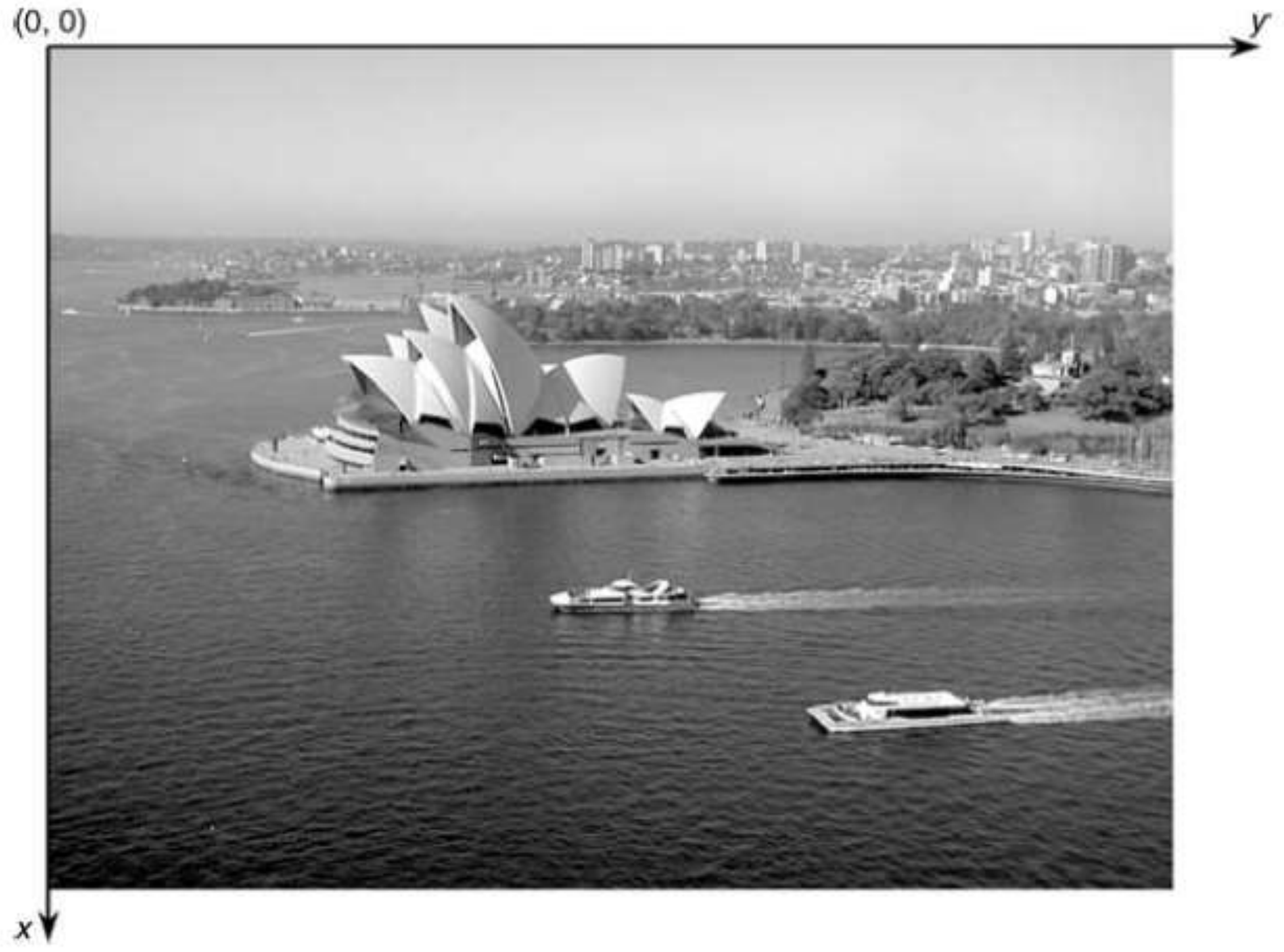
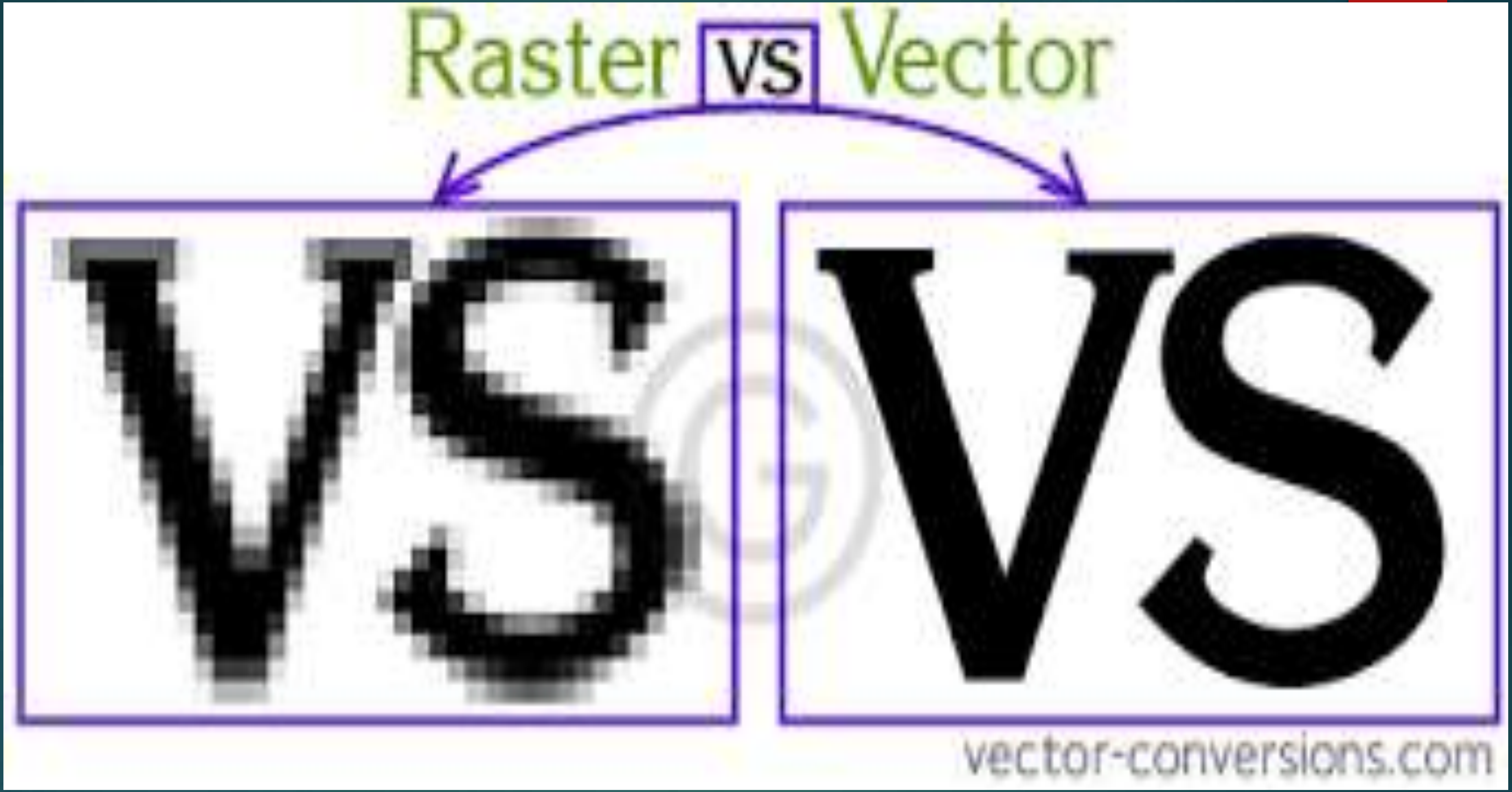


FIGURE 2.1 A monochrome image and the convention used to represent rows (x) and columns (y) adopted in this book.

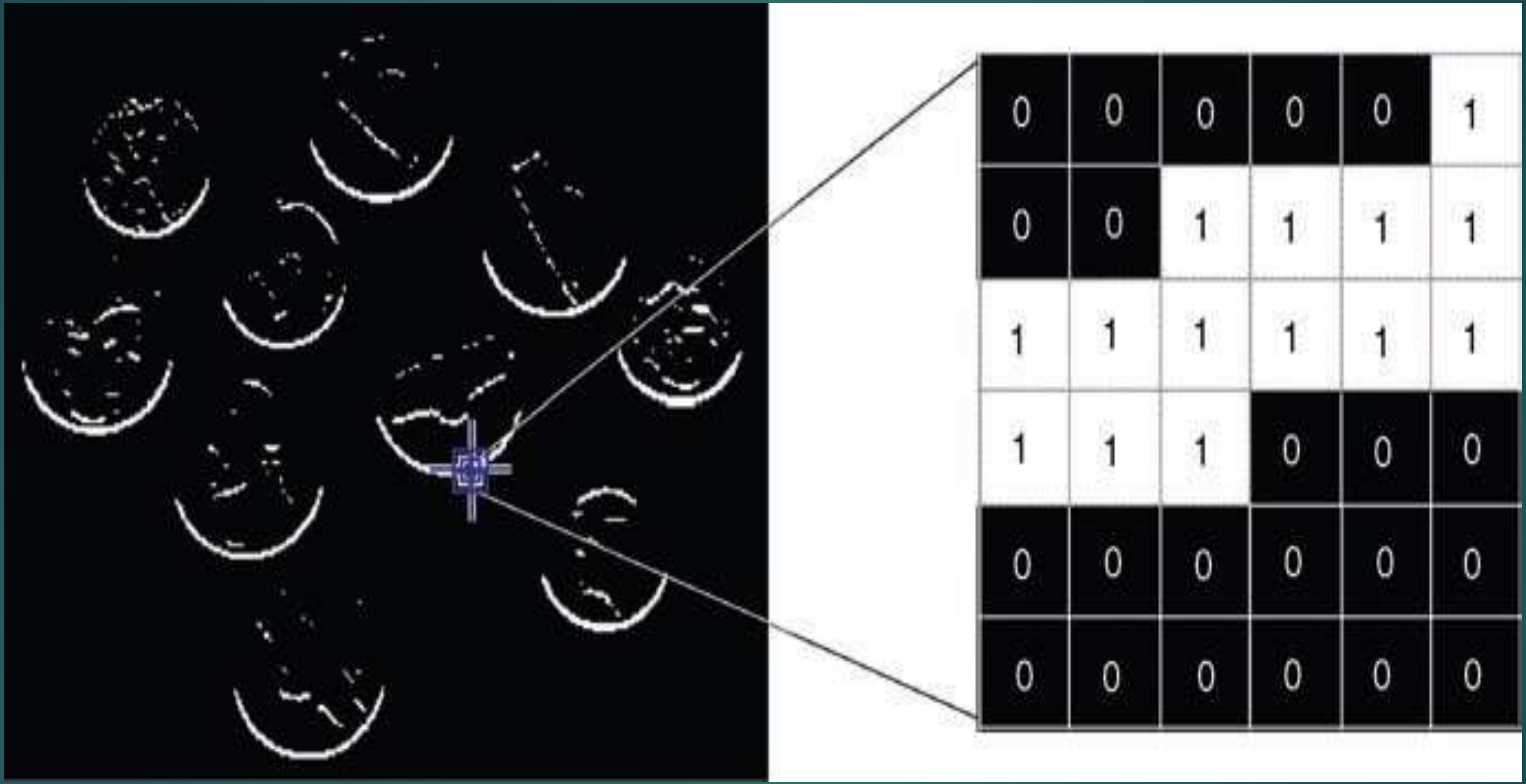
$$f(p,q) = \begin{bmatrix} f(1,1) & f(1,2) & \cdots & f(1,N) \\ f(2,1) & f(2,2) & \cdots & f(2,N) \\ \vdots & \vdots & & \vdots \\ f(M,1) & f(M,2) & \cdots & f(M,N) \end{bmatrix}$$

2D image in digital format
format: ***raster* (also known as *bitmap*) and *vector*.**
Bitmap representations use one or more two-dimensional arrays of pixels, advantages of bitmap graphics are:
 their quality and display speed;
Disadvantages
larger memory storage requirements
size dependence (e.g., enlarging a bitmap image may lead to noticeable artifacts).
vector representations use a series of drawing commands to represent an image.
advantages
Vector representations require less memory and allow resizing and geometric manipulations without introducing artifacts,
Disadvantages
need to be rasterized for most presentation devices.


Raster
vs
Vector
VS
VS
vector-conversions.com

Raster Graphics	Vector Graphics
They are composed of pixels.	They are composed of paths.
In Raster Graphics, refresh process is independent of the complexity of the image.	Vector displays flicker when the number of primitives in the image become too large.
Graphic primitives are specified in terms of end points and must be scan converted into corresponding pixels.	Scan conversion is not required.
Raster graphics can draw mathematical curves, polygons and boundaries of curved primitives only by pixel approximation.	Vector graphics draw continuous and smooth lines.
Raster graphics cost less.	Vector graphics cost more as compared to raster graphics.
They occupy more space which depends on image quality.	They occupy less space.
File extensions: .BMP, .TIF, .GIF, .JPG	File Extensions: .SVG, .EPS, .PDF, .AI, .DXF

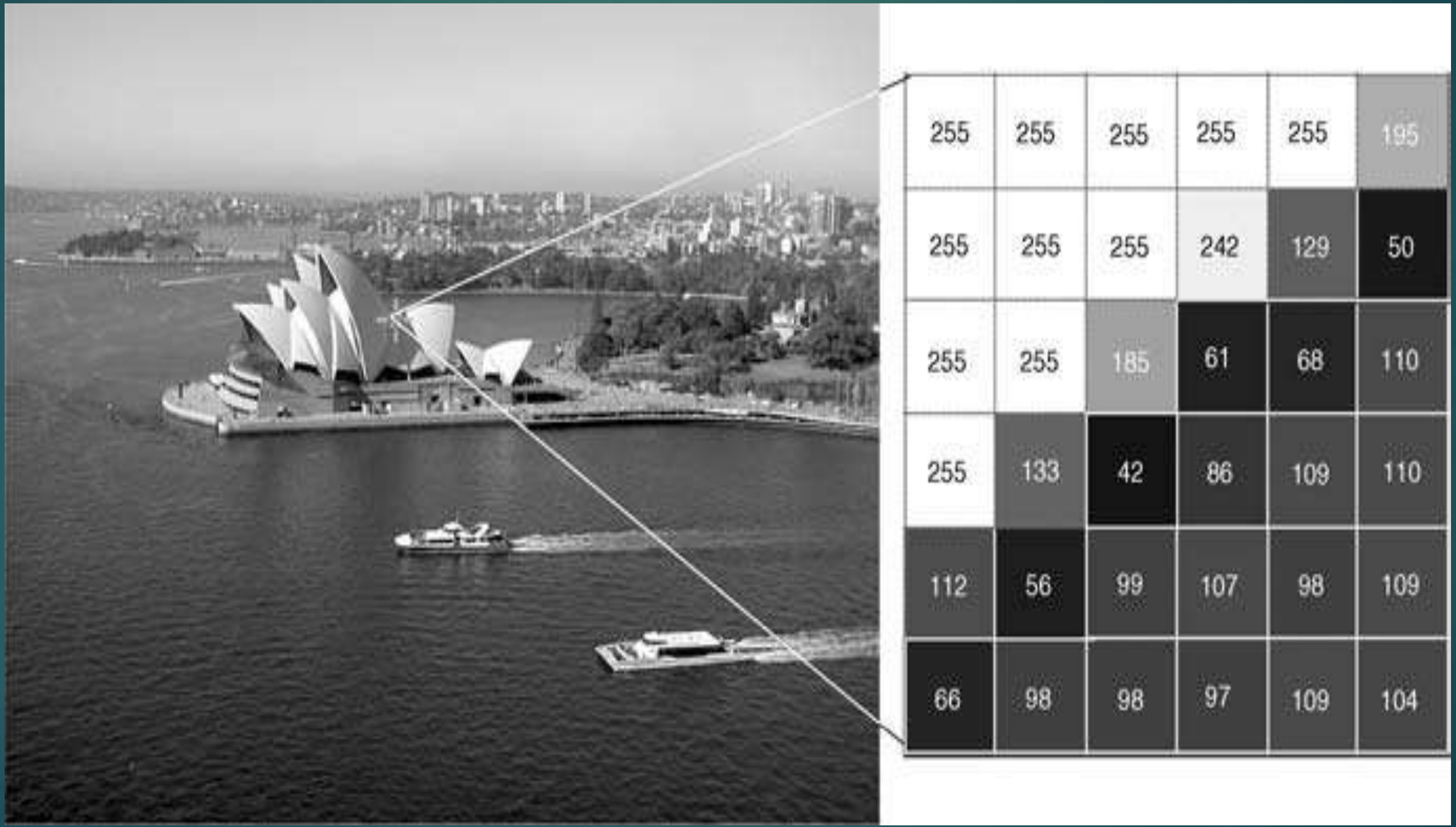
2.1.1 Binary (1-Bit) Images



0	0	0	0	0	1
0	0	1	1	1	1
1	1	1	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

2.1.2 Gray-Level (8-Bit) Images

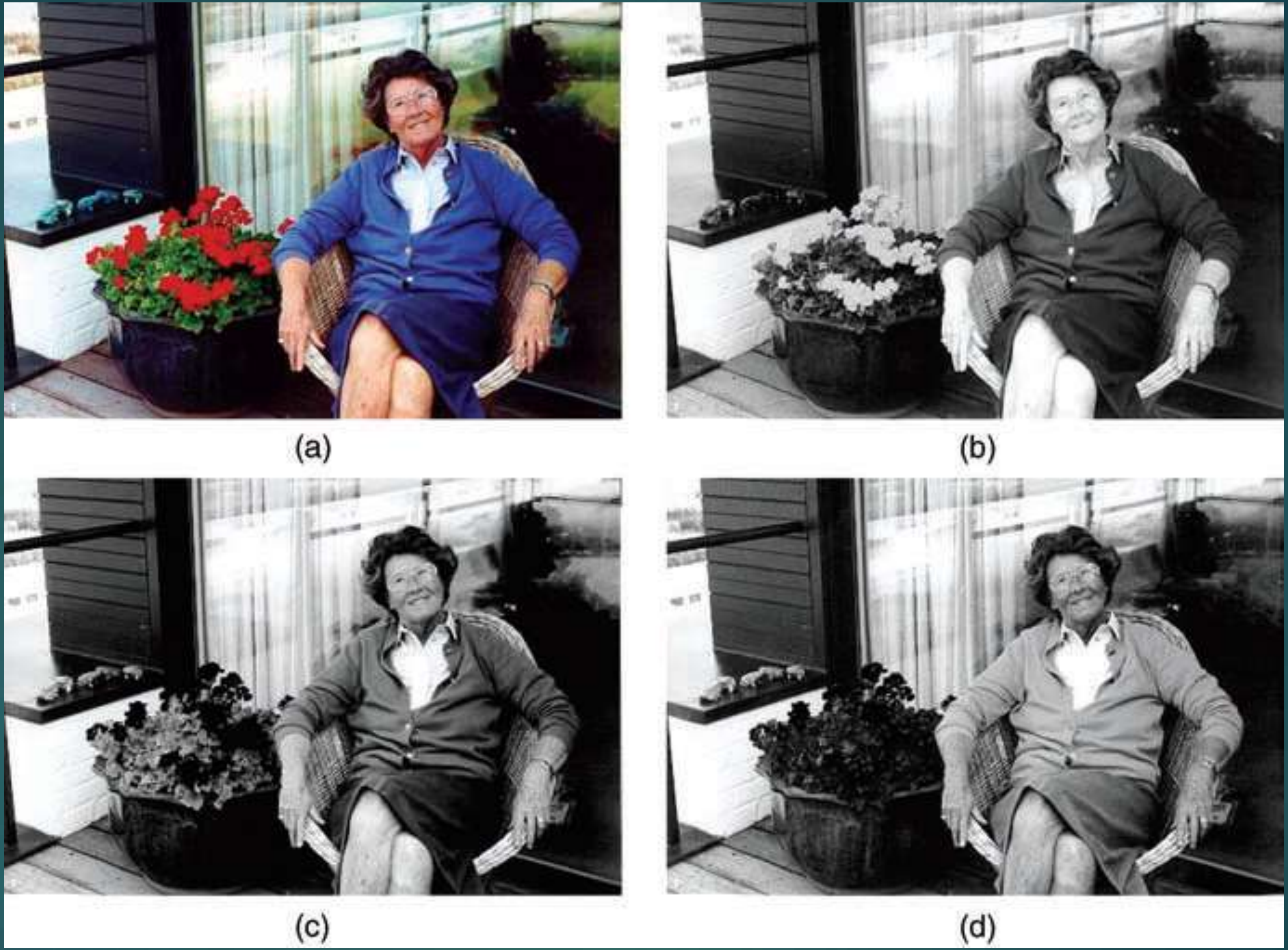
usually with 8 bits per pixel



255	255	255	255	255	195
255	255	255	242	129	50
255	255	185	61	68	110
255	133	42	86	109	110
112	56	99	107	98	109
66	98	98	97	109	104

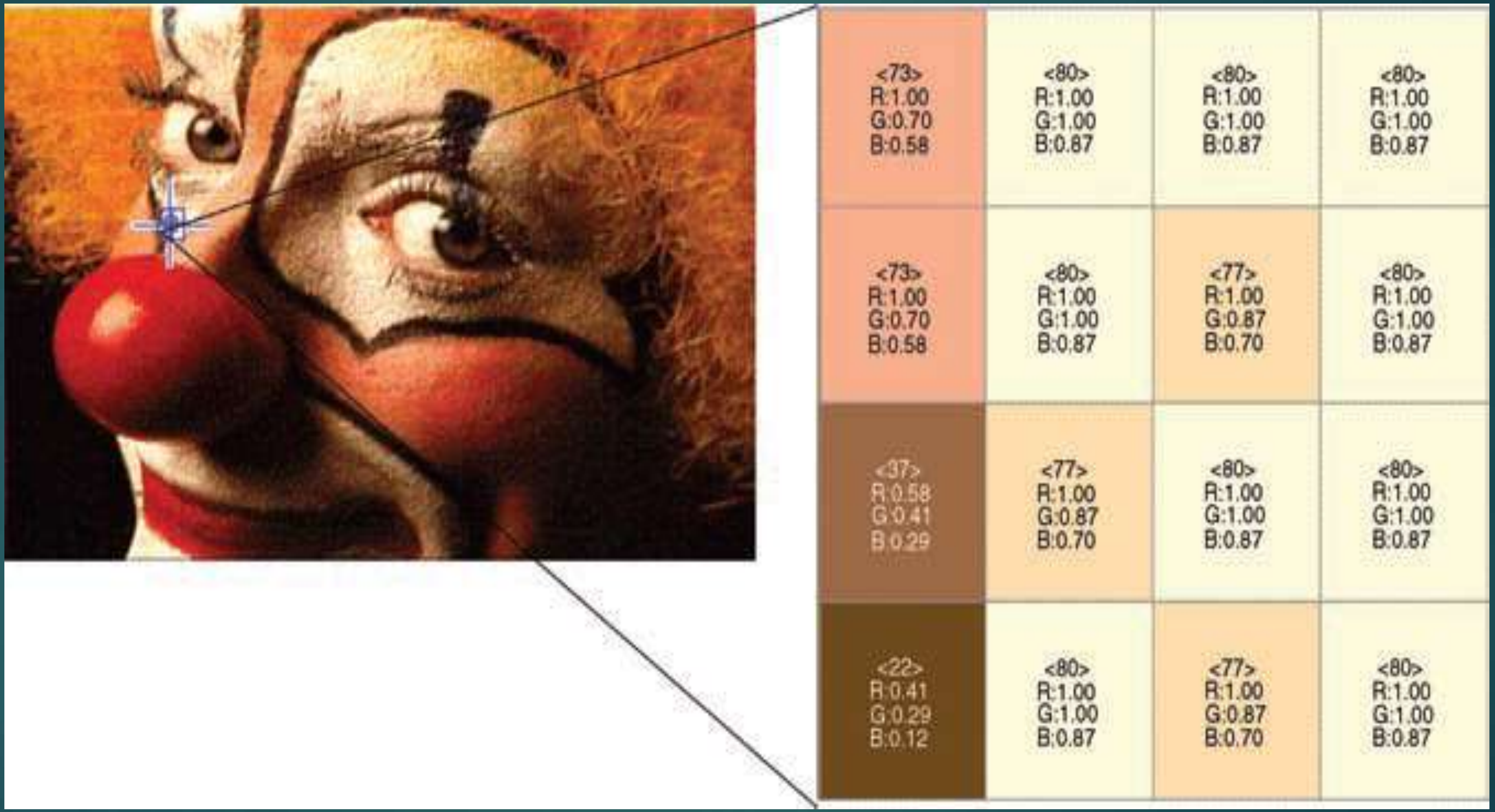
2.1.3 Color Images

24-Bit (RGB) Color Images Color images can be represented using three 2D arrays of same size, one for each color channel: red (R), green (G), and blue (B) (Figure 2.4).1 Each array element contains an 8-bit value, indicating the amount of red, green, or blue at that point in a [0, 255] scale.



Color image (a) and its R (b), G (c), and B (d) components

Indexed Color Images



<73> R:1.00 G:0.70 B:0.58	<80> R:1.00 G:1.00 B:0.87	<80> R:1.00 G:1.00 B:0.87	<80> R:1.00 G:1.00 B:0.87
<73> R:1.00 G:0.70 B:0.58	<80> R:1.00 G:1.00 B:0.87	<77> R:1.00 G:0.87 B:0.70	<80> R:1.00 G:1.00 B:0.87
<37> R:0.58 G:0.41 B:0.29	<77> R:1.00 G:0.87 B:0.70	<80> R:1.00 G:1.00 B:0.87	<80> R:1.00 G:1.00 B:0.87
<22> R:0.41 G:0.29 B:0.12	<80> R:1.00 G:1.00 B:0.87	<77> R:1.00 G:0.87 B:0.70	<80> R:1.00 G:1.00 B:0.87

Where are vector graphics used?

Outside of screen printing, Vector graphics are used in **text**, **logos**, **illustrations**, **symbols**, **infographics**, **charts**, **and graphs**.
They are created and edited in computer programs such as Adobe Illustrator and, Corel Draw. Typical formats for a vector file are .ai (Adobe Illustrator file), .cdr (Corel Draw file) .eps or .pdf. However, not all .eps or .pdf files are automatically vector-based. To understand why, we need to explore raster graphics.

Where are raster graphics used?

Photographs and scanned images are the most common examples of raster graphics. Raster graphics often **show more subtle changes in color, tone, and value than vector** graphics are able to **achieve.** Unlike a vector graphic, it is impossible to take a small raster graphic and scale it up **without losing image quality.**

Raster graphics or images are captured by a digital camera or scanned into the computer and edited by programs such as Adobe Photoshop. Typical file formats include .jpg, .psd, .png, .tiff, .bmp, and .gif. However, both raster and vector graphics can be saved as .eps and .pdf.

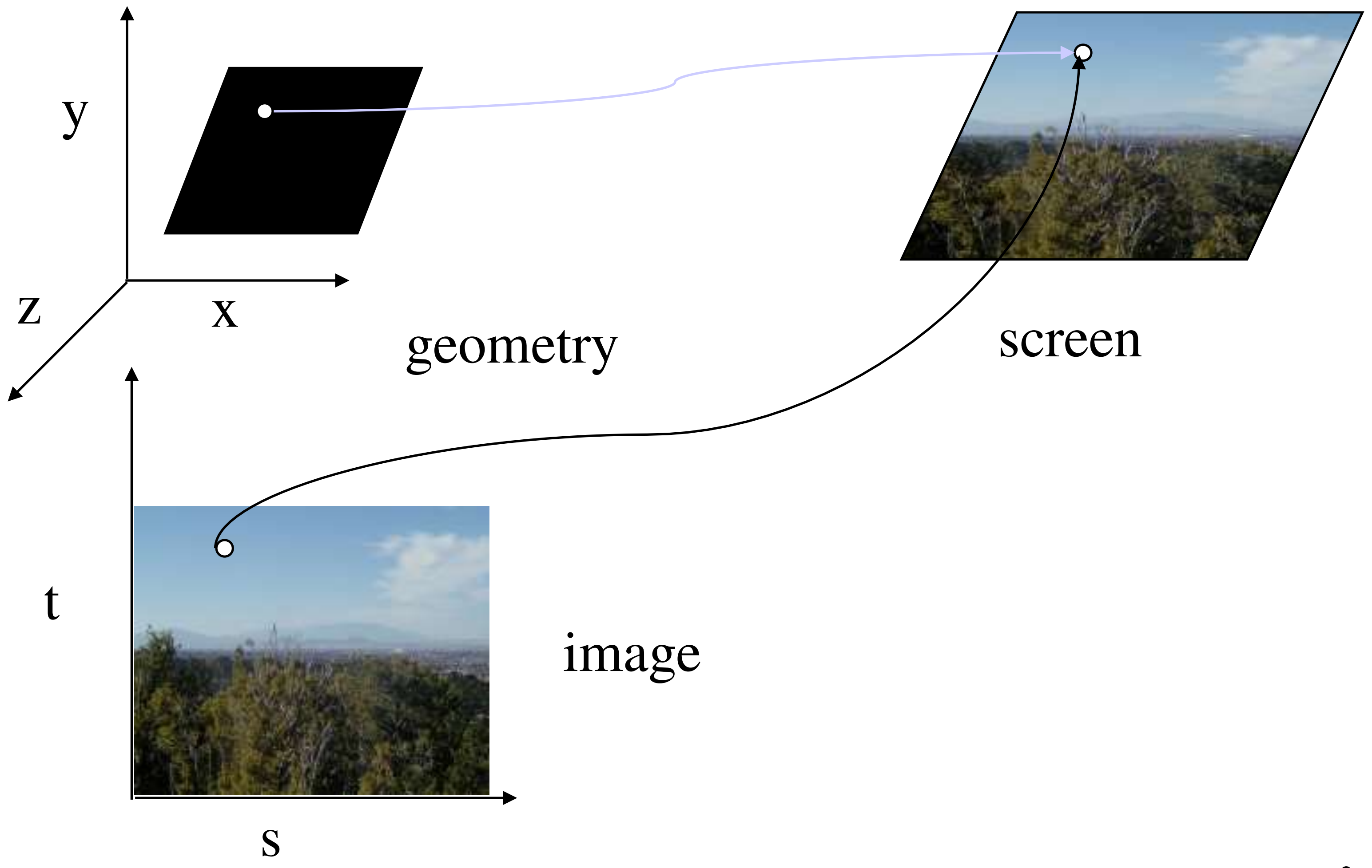
OpenGL Texture Mapping

Basic Stragegy

- Three steps to applying a texture

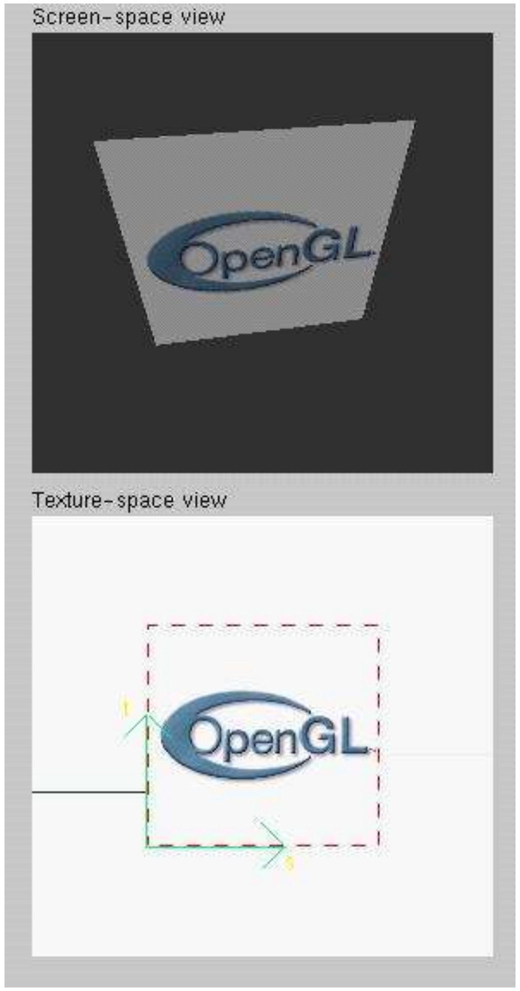
1. specify the texture
    - read or generate image
    - assign to texture
    - enable texturing
2. assign texture coordinates to vertices
    - Proper mapping function is left to application
3. specify texture parameters
    - wrapping, filtering

Texture Mapping



Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective



Specify Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory

```
Glubyte my_texels[512][512][4];
```

- Define as any other pixel map
  - Scan
  - Via application code
- Enable texture mapping
  - **`glEnable(GL_TEXTURE_2D)`**
  - OpenGL supports 1-4 dimensional texture maps

Define Image as a Texture

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

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

`level:` used for mipmapping (discussed later)

`components:` elements per texel

`w, h:` width and height of `texels` in pixels

`border:` used for smoothing (discussed later)

`format and type:` describe texels

`texels:` pointer to texel array

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0,
 GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

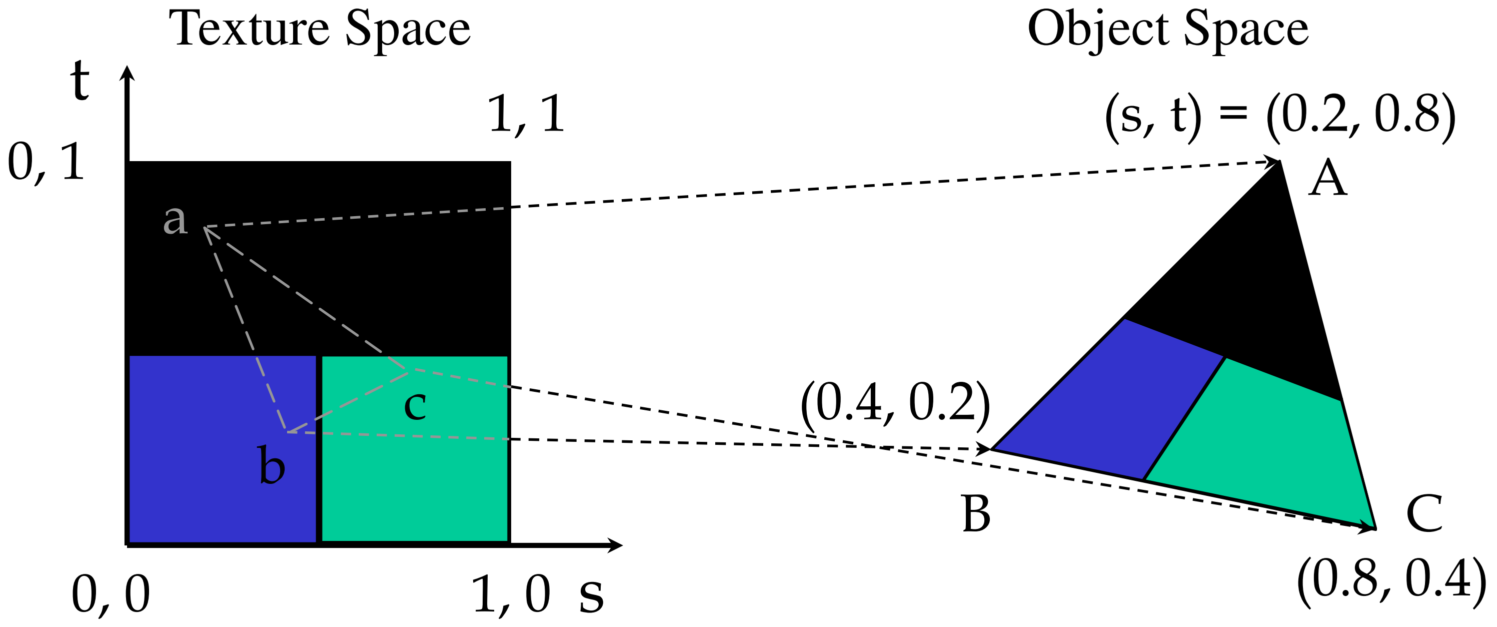
Converting A Texture Image

- OpenGL requires texture dimensions to be powers of 2
- If dimensions of image are not powers of 2

  -
    ```
    gluScaleImage( format, w_in, h_in,
        type_in, *data_in, w_out, h_out,
            type_out, *data_out );
    ```
  - `data_in` is source image
  - `data_out` is for destination image
- Image interpolated and filtered during scaling

Mapping a Texture

- Based on parametric texture coordinates
- **`glTexCoord*()`** specified at each vertex



Typical Code

```
glBegin(GL_POLYGON);
 glColor3f(r0, g0, b0);
 glNormal3f(u0, v0, w0);
 glTexCoord2f(s0, t0);
 glVertex3f(x0, y0, z0);
 glColor3f(r1, g1, b1);
 glNormal3f(u1, v1, w1);
 glTexCoord2f(s1, t1);
 glVertex3f(x1, y1, z1);
      .
      .
glEnd();
```

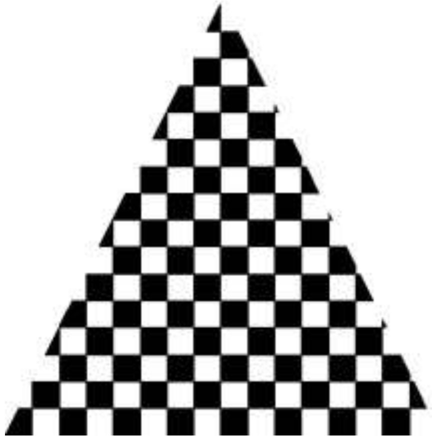
Note that we can use vertex arrays to increase efficiency

Interpolation

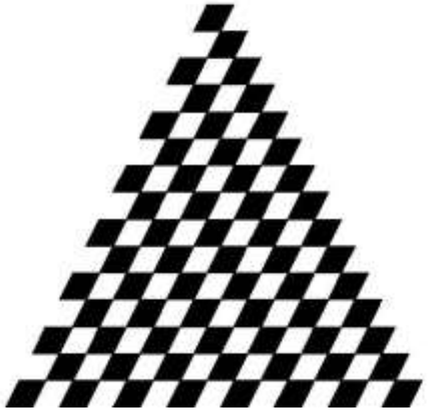
OpenGL uses bilinear interpolation to find proper texels from specified texture coordinates

Can be distortions

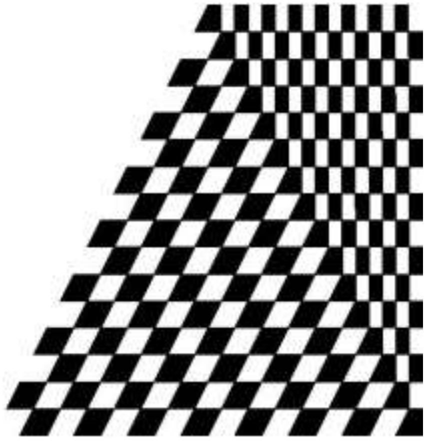
good selection of tex coordinates



poor selection of tex coordinates



texture stretched over trapezoid showing effects of bilinear interpolation



Texture Parameters

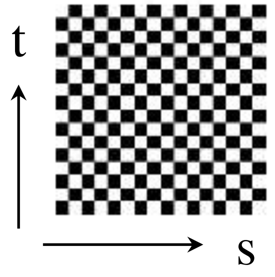
- OpenGL a variety of parameter that determine how texture is applied
  - **Wrapping parameters** determine what happens of s and t are outside the (0,1) range
  - **Filter modes** allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

Wrapping Mode

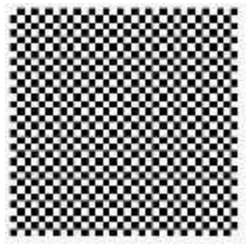
Clamping: if s,t > 1 use 1, if s,t <0 use 0

Wrapping: use s,t modulo 1

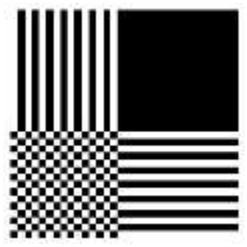
```
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_WRAP_T, GL_REPEAT )
```



texture



GL_REPEAT wrapping

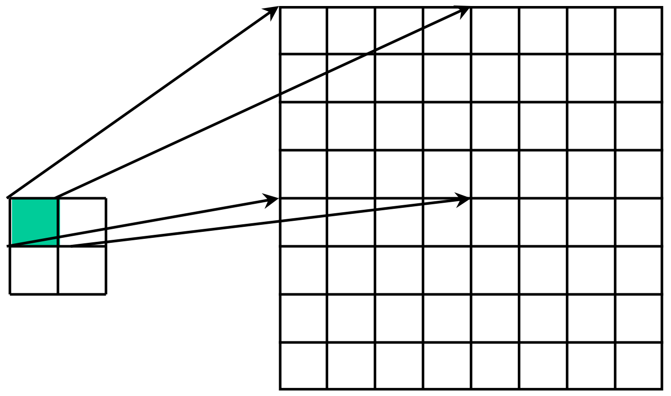


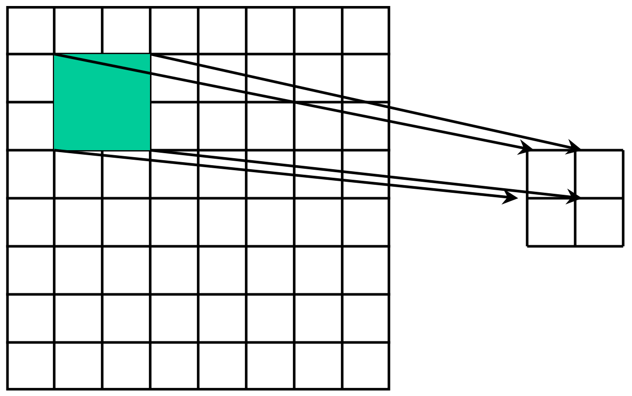
GL_CLAMP wrapping

Magnification and Minification

More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values





Filter Modes

Modes determined by

- `glTexParameteri( target, type, mode )`

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MAG_FILTER,
                GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER,
                GL_LINEAR);
```

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)

Mipmapped Textures

- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

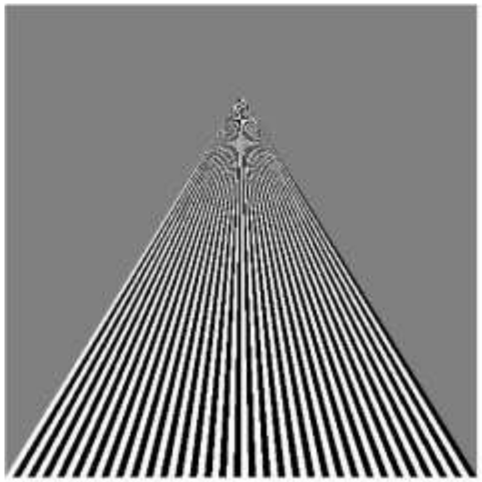
```
glTexImage2D( GL_TEXTURE_*D, level, … )
```

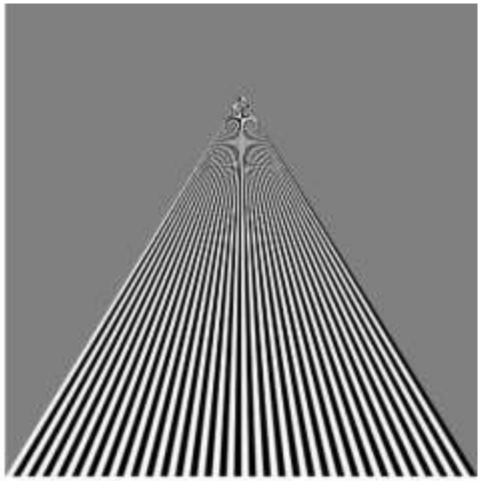
- GLU mipmap builder routines will build all the textures from a given image

```
gluBuild*DMipmaps( … )
```

Example

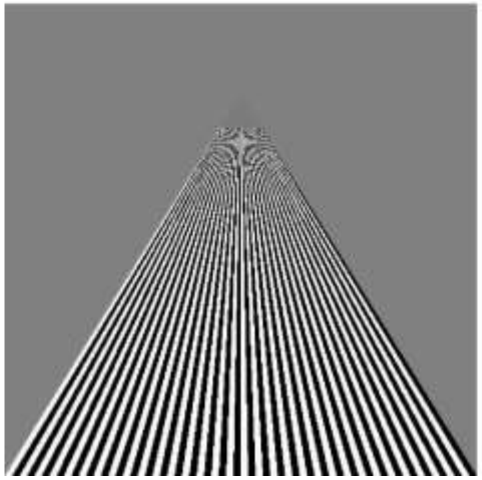
point sampling

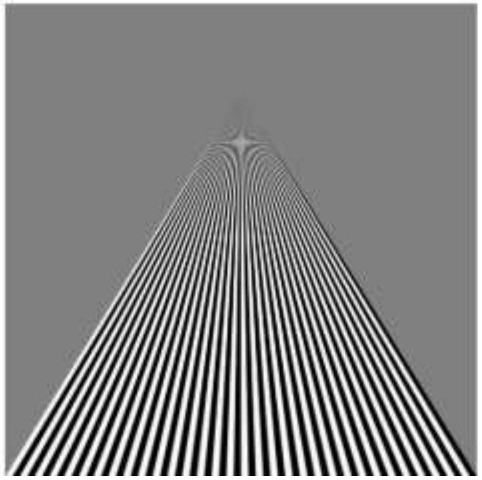




linear filtering

mipmapped point sampling





mipmapped linear filtering

Texture Functions

- Controls how texture is applied
  - **`glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop, param )`**
- **`GL_TEXTURE_ENV_MODE`** modes
  - **`GL_MODULATE:`** modulates with computed shade
  - **`GL_BLEND:`** blends with an environmental color
  - **`GL_REPLACE:`** use only texture color
  - `GL(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);`
- Set blend color with **`GL_TEXTURE_ENV_COLOR`**

Perspective Correction Hint

- Texture coordinate and color interpolation
  - either linearly in screen space
  - or using depth/perspective values (slower)
- Noticeable for polygons "on edge"
  - **`glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )`**

    where **`hint`** is one of
    - **`GL_DONT_CARE`**
    - **`GL_NICEST`**
    - **`GL_FASTEST`**

Generating Texture Coordinates

- OpenGL can generate texture coordinates automatically

  `glTexGen{ifd}[v]()`

- specify a plane
  - generate texture coordinates based upon distance from the plane
- generation modes
  - **`GL_OBJECT_LINEAR`**
  - **`GL_EYE_LINEAR`**
  - **`GL_SPHERE_MAP`** **(used for environmental maps)**

From Sec.2.4 of “Advanced Graphics Programming Using OpenGL” (electronic copy available in NTHU library) http://www.netLibrary.com/urlapi.asp?action=summary&v=1&bookid=130156

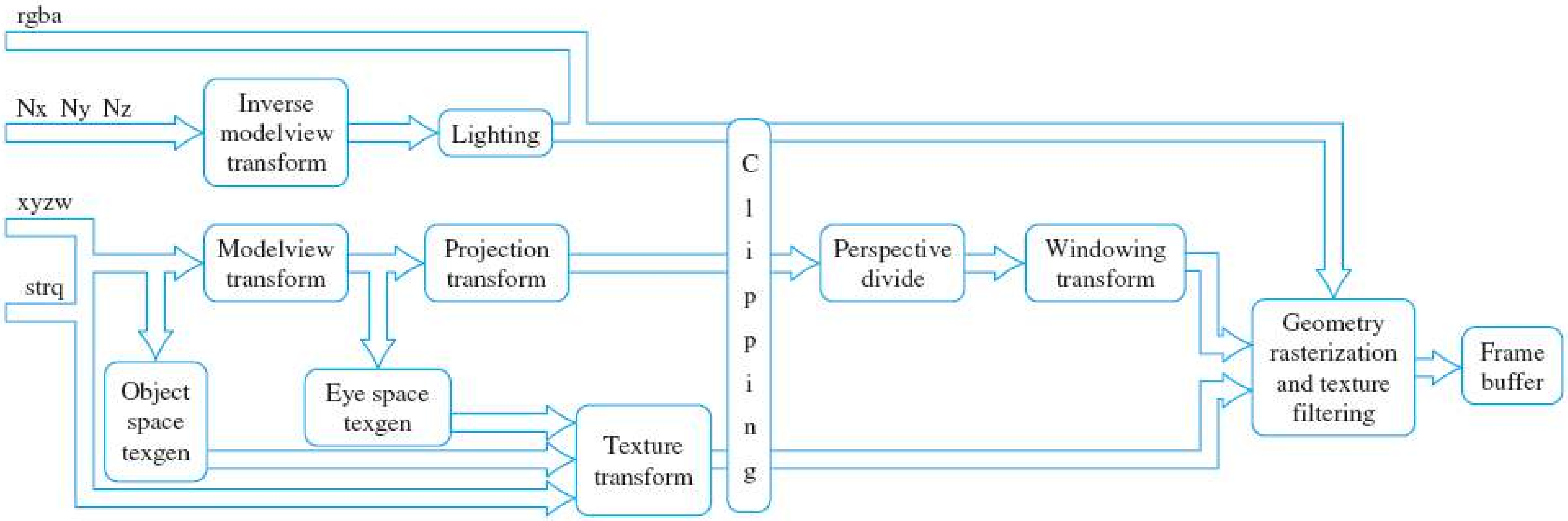


Figure 2.5 Texture coordinate transformation pipeline.

Texture Objects

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

Other Texture Features

- Environmental Maps
  - Start with image of environment through a wide angle lens
    - Can be either a real scanned image or an image created in OpenGL t
  - Use this texture to generate a spherical map
  - Use automatic texture coordinate generation
- Multitexturing
  - Apply a sequence of textures through cascaded texture units

Lighting (in OpenGL)

Hidden surface removal

OpenGL implements hidden-surface removal using a simple technique called depth buffering (also known as Z-buffering). This takes place during rasterization, using a “depth buffer” – an array which records a depth value corresponding to each pixel in the window.

1. Initially, each depth value is set to be a very large number.
2. a new pixel is generated, for example during the scan-conversion of a polygon P1,
3. the pixel's Z value is compared with the corresponding value in the depth-buffer.
4. If the pixel's depth is less than that in the buffer, the pixel is drawn and its depth recorded in the depth buffer, over-writing the previous value.
5. Otherwise, the pixel is not drawn and the depth buffer is not updated.

To tell OpenGL to perform hidden-surface removal using a depth buffer, you need:

First, in the call to glutInitDisplayMode(), instruct GLUT to create a depth buffer using:

```
glutInitDisplayMode (GLUT_DOUBLE |
GLUT_DEPTH);
```

Second, enable the depth test, which is switched off by default, using glEnable():

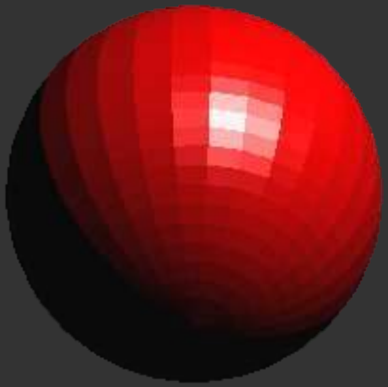
```
glEnable (GL_DEPTH_TEST);
```

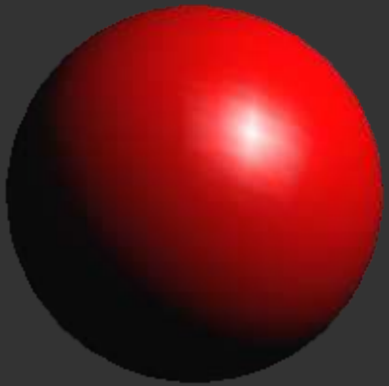
Finally, you need to explictly clear the depth buffer (in other words, re-load it with large depth values)
each time around the rendering loop:

```
void display () {
glClear (GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
/ * all your display code */
}
```

Importance of Lighting

- Important to bring out 3D appearance (compare teapot now to in previous demo)
- Important for correct shading under lights
- The way shading is done also important

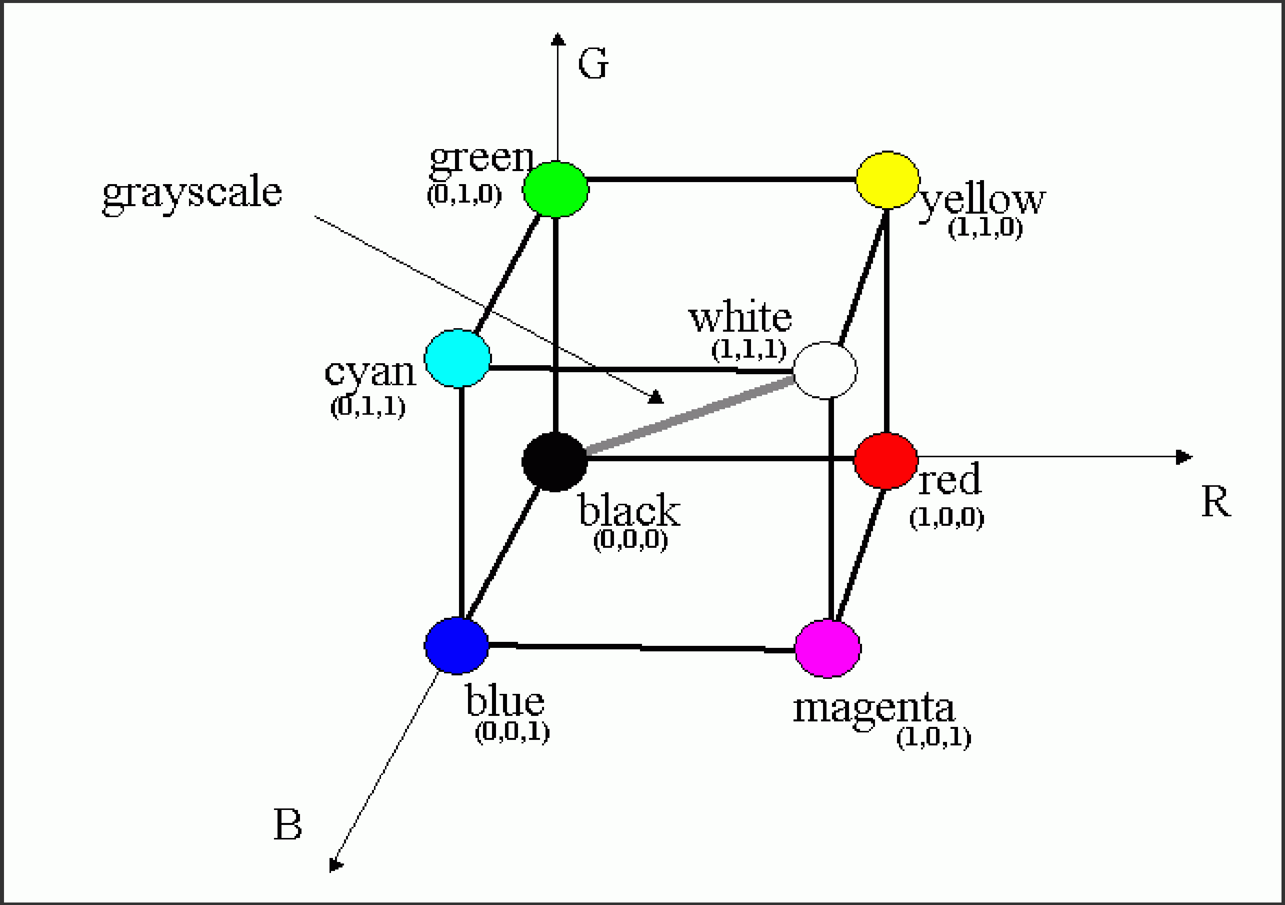

glShadeModel(GL_FLAT)


glShadeModel(GL_SMOOTH)

Brief primer on Color

- Red, Green, Blue primary colors
  - Can be thought of as vertices of a color cube
  - R+G = Yellow, B+G = Cyan, B+R = Magenta, R+G+B = White
  - Each color channel (R,G,B) treated separately
- RGBA 32 bit mode (8 bits per channel) often used
  - A is for alpha for transparency if you need it
- Colors normalized to 0 to 1 range in OpenGL
  - Often represented as 0 to 255 in terms of pixel intensities
- Also, color index mode (not so important)

RGB Color Space



CMY Color Model

- C: Cyan;   M: Magenta; Y: Yellow
- Subtractive primaries - Cyan, Magenta, and Yellow are the compliment of Red, Green Blue
- Specified by what is being removed from white
- Example: Cyan color = (1,0,0) means red is removed; CMY: (1,1,0) -> red and green is removed => what color?
- Sometimes CMYK - K: Black

CMY <-> RGB

$$\begin{vmatrix} C \\ M \\ Y \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix} - \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

If the color in RGB is (1,1,0) then the color in CMY is (0,0,1)

If the color is (50,100,80) then the color in CMY is (205,155,175)

Shading Models

- So far, lighting disabled: color explicit at each vertex
- This lecture, enable lighting
  - Calculate color at each vertex (based on shading model, lights and material properties of objects)
  - Rasterize and interpolate vertex colors at pixels
- Flat shading: single color per polygon (one vertex)
- Smooth shading: interpolate colors at vertices
- Wireframe: glPolygonMode (GL_FRONT, GL_LINE)
  - Also, polygon offsets to superimpose wireframe
  - Hidden line elimination? (polygons in black…)

Lighting

- Rest of this lecture considers lighting on vertices
- In real world, complex lighting, materials interact
- OpenGL is a hack that efficiently captures some qualitative lighting effects. But not physical
- Modern programmable shaders allow arbitrary lighting and shading models (not covered in class)

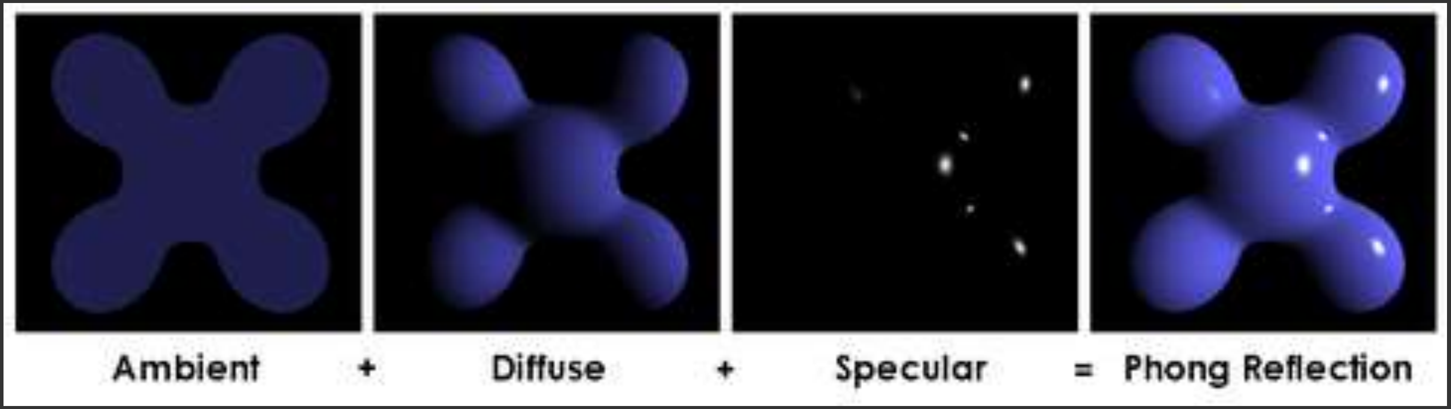
Types of Light Sources

- Point
  - Position, Color [separate diffuse/specular]
  - Attenuation (quadratic model)

$$atten = \frac{1}{k_c + k_l d + k_q d^2}$$

- Directional (w=0, infinitely far away, no attenuation)
- Spotlights
  - Spot exponent
  - Spot cutoff

ambient light is (in the classic lighting model) merely a constant value throughout a scene so is modelled by shading the object that colour,
diffuse light is modelled as the reflection of light in all directions at a given point,
specular light is modelled as the reflection of light in a single direction, which gives a shiny look. You can of course involve material properties that modify how an object is shaded, e.g. a low shininess value might make specular light less apparent.



Material Properties

- Need normals (to calculate how much diffuse, specular, find reflected direction and so on)
- Four terms: Ambient, Diffuse, Specular, Emissive

Ambient color : Ambient color is the color of an object where it is in shadow. This color is what the object reflects when illuminated by ambient light rather than direct light.

Diffuse color : Diffuse color is the most instinctive meaning of the color of an object. It is that essential color that the object reveals under pure white light. It is perceived as the color of the object itself rather than a reflection of the light.

Emissive color : This is the self-illumination color an object has.

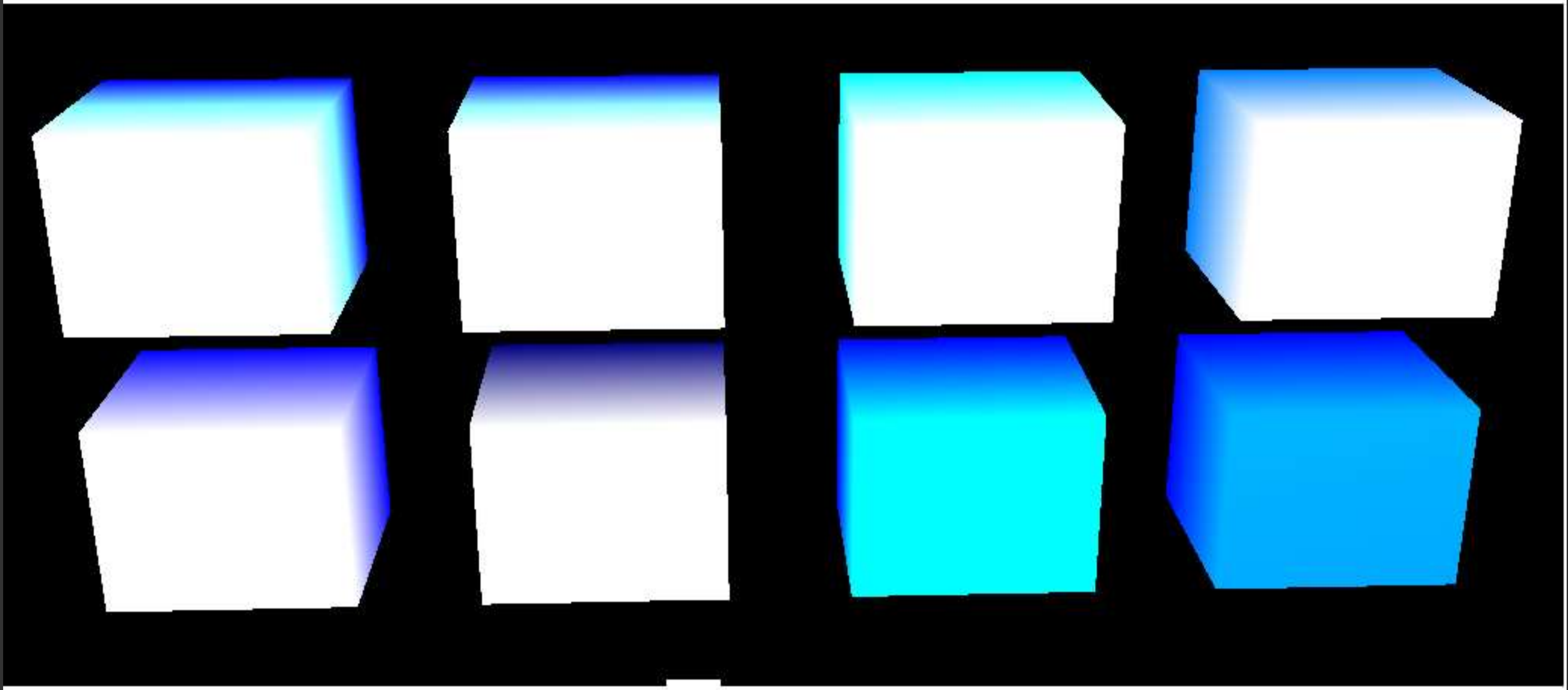
Specular color : Specular color is the color of the light of a specular reflection (specular reflection is the type of reflection that is characteristic of light reflected from a shiny surface).

ambient : 0 0 1
diffuse : 1 1 1
specular : 0 .5 0
emissive : 0 0 0

ambient : 0 0 1
diffuse : 1 1 1
specular : 0 .5 0
emissive : 0 0 0

ambient : 0 0 1
diffuse : 1 1 1
specular : 0 0 0
emissive : 0 .5 0

ambient : 0 0 1
diffuse : 1 1 1
specular : 0 0 0
emissive : 0 .5 0



ambient : 0 0 1
diffuse : 1 1 1
specular : 0 0 0
emissive : 0 0 0

ambient : 0 0 .5
diffuse : 1 1 1
specular : 0 0 0
emissive : 0 0 0

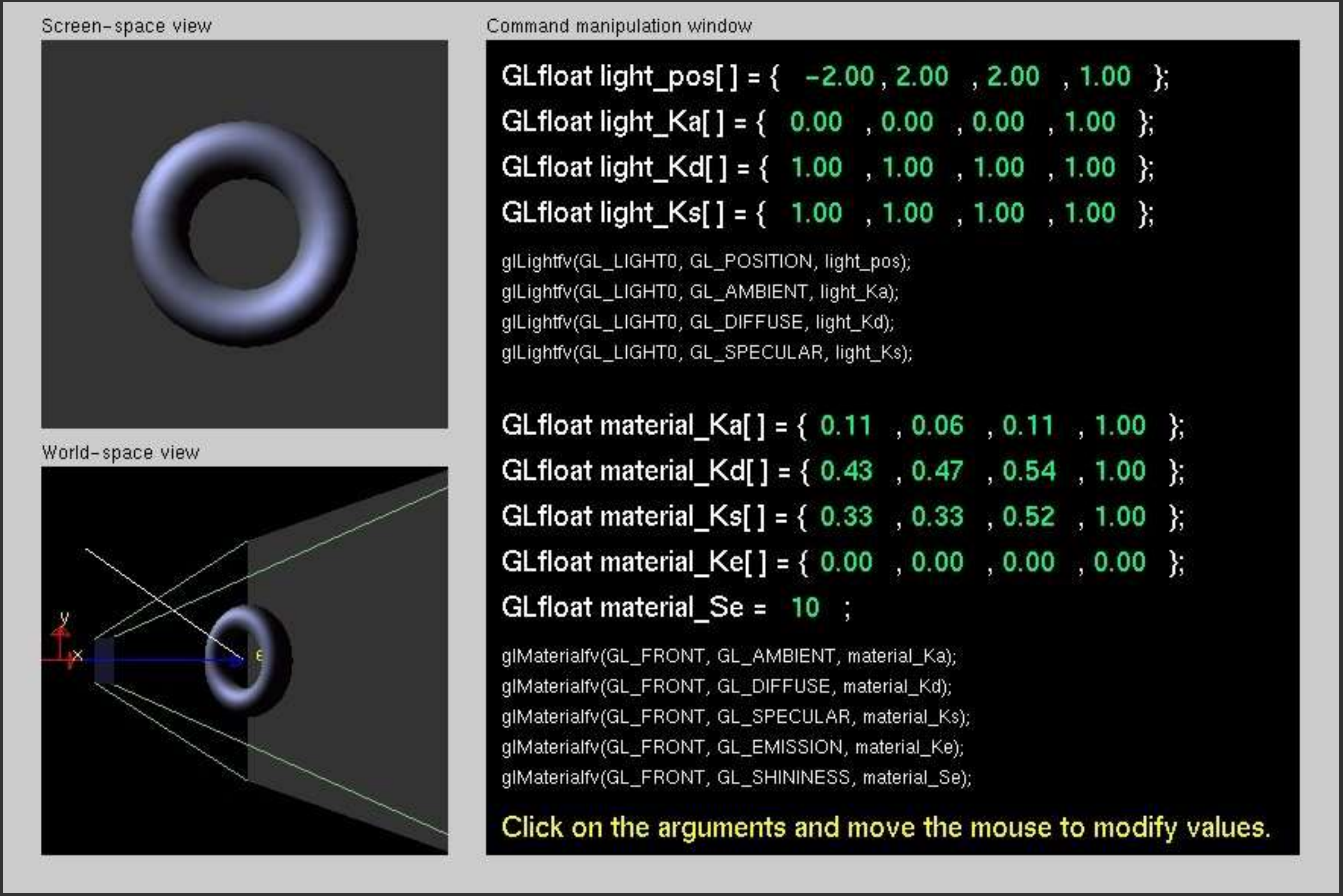
ambient : 0 0 1
diffuse : 0 1 0
specular : 0 0 0
emissive : 0 0 0

ambient : 0 0 1
diffuse : 0 .5 0
specular : 0 0 0
emissive : 0 0 0

Specifying Normals

- Normals are specified through glNormal
- Normals are associated with vertices
- Specifying a normal sets the *current* normal
  - Remains unchanged until user alters it
  - Usual sequence: glNormal, glVertex, glNormal, glVertex, glNormal, glVertex…
- Usually, we want unit normals for shading
  - glEnable(GL_NORMALIZE)
  - This is slow – either normalize them yourself or don't use glScale
- Evaluators will generate normals for curved surfaces
  - Such as splines. GLUT does it automatically for teapot, cylinder,…

LightMaterial



```
GLfloat light_pos[ ] = {  -2.00 , 2.00  , 2.00  , 1.00  };
GLfloat light_Ka[ ] = {  0.00  , 0.00  , 0.00  , 1.00  };
GLfloat light_Kd[ ] = {  1.00  , 1.00  , 1.00  , 1.00  };
GLfloat light_Ks[ ] = {  1.00  , 1.00  , 1.00  , 1.00  };

glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_Ka);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_Kd);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_Ks);

GLfloat material_Ka[ ] = { 0.11  , 0.06  , 0.11  , 1.00  };
GLfloat material_Kd[ ] = { 0.43  , 0.47  , 0.54  , 1.00  };
GLfloat material_Ks[ ] = { 0.33  , 0.33  , 0.52  , 1.00  };
GLfloat material_Ke[ ] = { 0.00  , 0.00  , 0.00  , 0.00  };
GLfloat material_Se =  10  ;

glMaterialfv(GL_FRONT, GL_AMBIENT, material_Ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, material_Kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, material_Ks);
glMaterialfv(GL_FRONT, GL_EMISSION, material_Ke);
glMaterialfv(GL_FRONT, GL_SHININESS, material_Se);
```

Click on the arguments and move the mouse to modify values.

Emissive Term

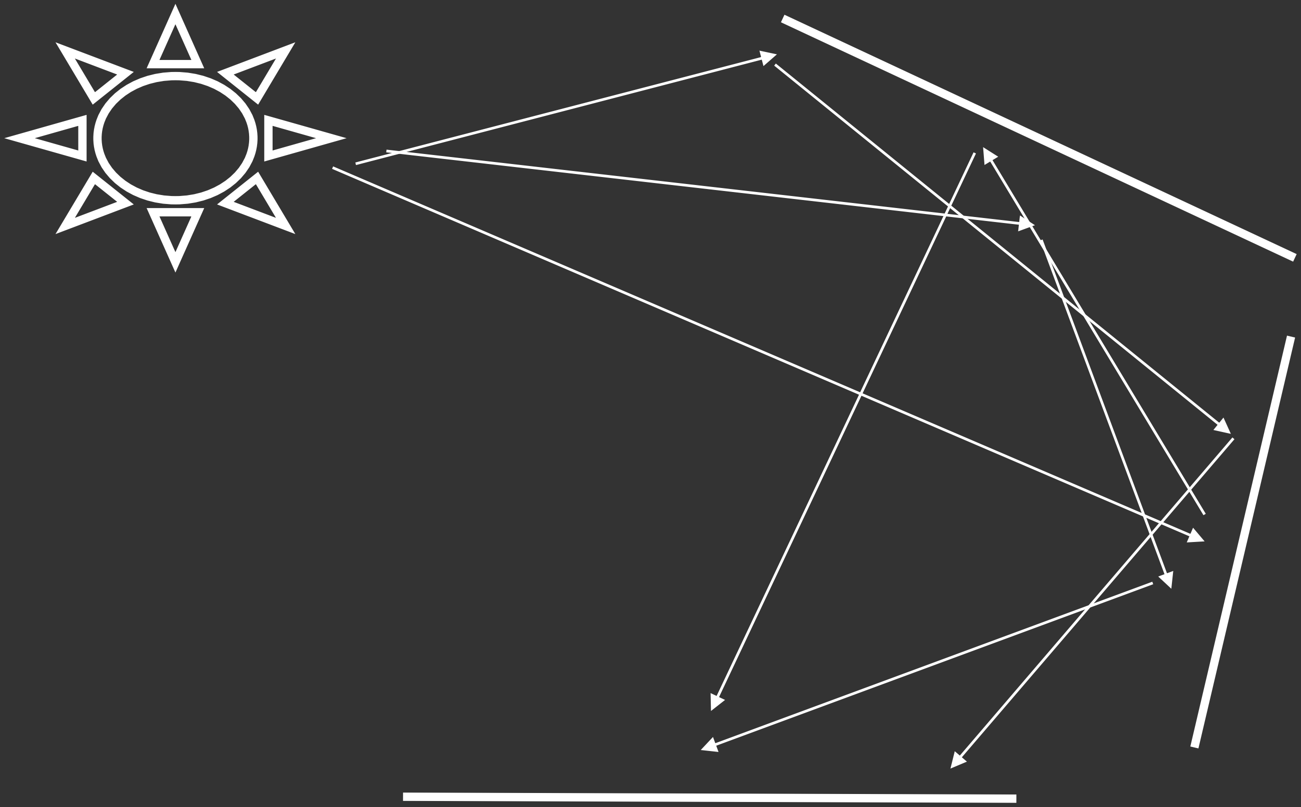
$$I = Emission_{material}$$

Only relevant for light sources when looking directly at them

- Gotcha: must create geometry to actually see light
- Emission does not in itself affect other lighting calculations

Ambient Term

- Hack to simulate multiple bounces, scattering of light
- Assume light equally from all directions



Ambient Term

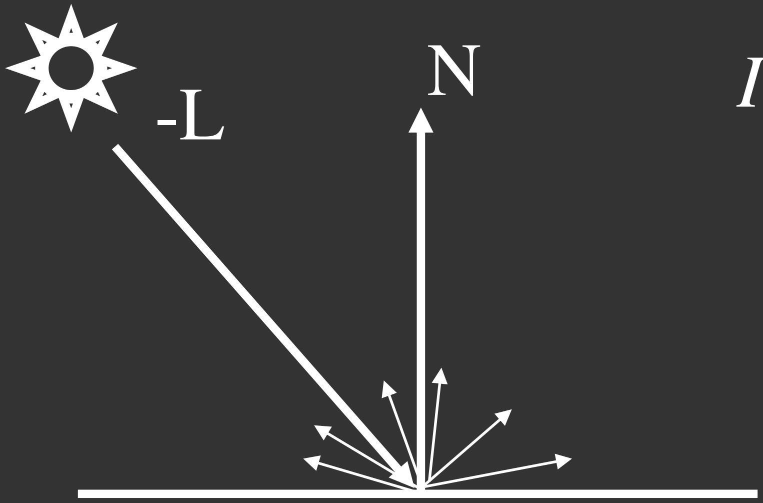
- Associated with each light and overall light
- E.g. skylight, with light from everywhere

$$I = ambient_{global} * ambient_{material} + \sum_{i=0}^{n} ambient_{light\ i} * ambient_{material} * atten_i$$

Most effects per light involve linearly combining effects of light sources

Diffuse Term

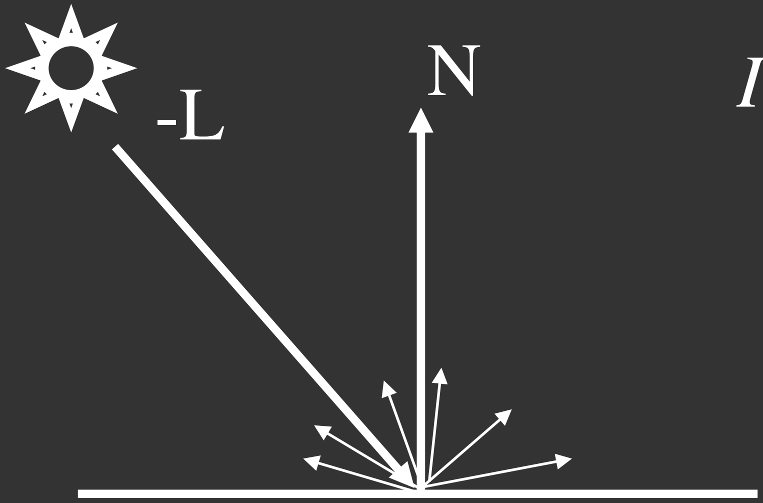
- Rough matte (technically Lambertian) surfaces
- Light reflects equally in all directions



$$I \propto N \bullet L$$

Diffuse Term

- Rough matte (technically Lambertian) surfaces
- Light reflects equally in all directions

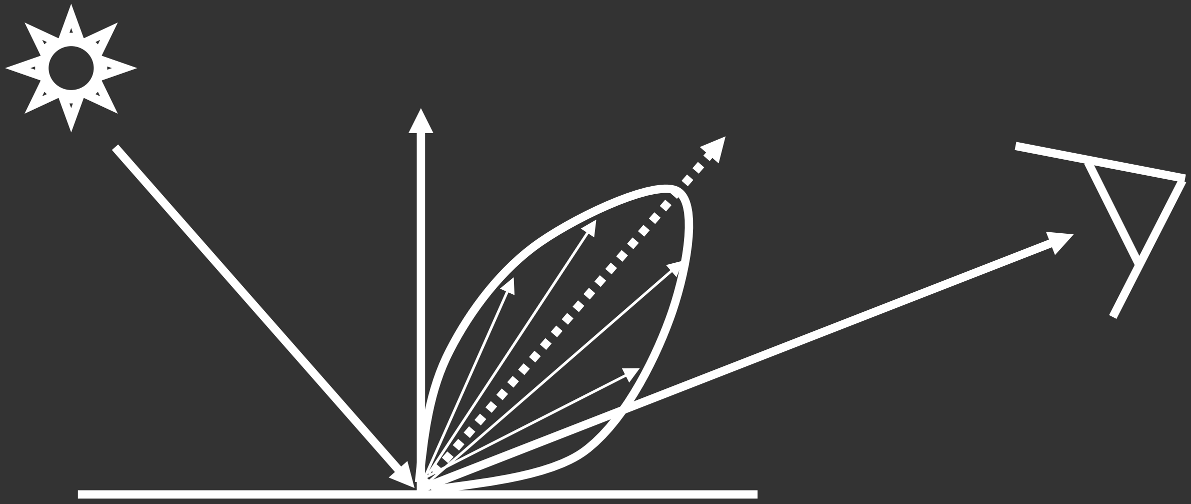


$I \propto N \bullet L$

$$I = \sum_{i=0}^{n} diffuse_{light\ i} * diffuse_{material} * atten_i * [\max(L\ N, 0)]$$

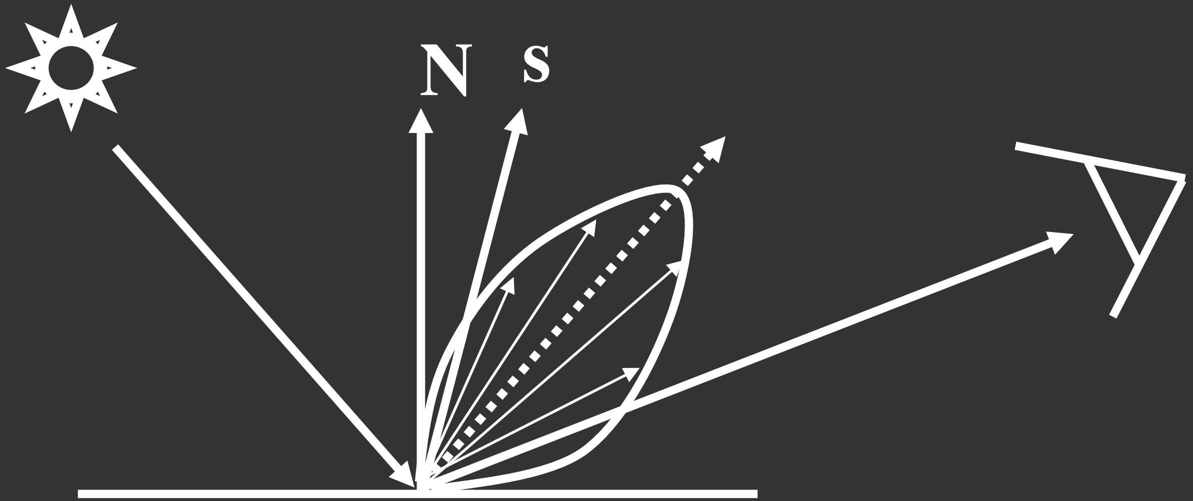
Specular Term

- Glossy objects, specular reflections
- Light reflects close to mirror direction

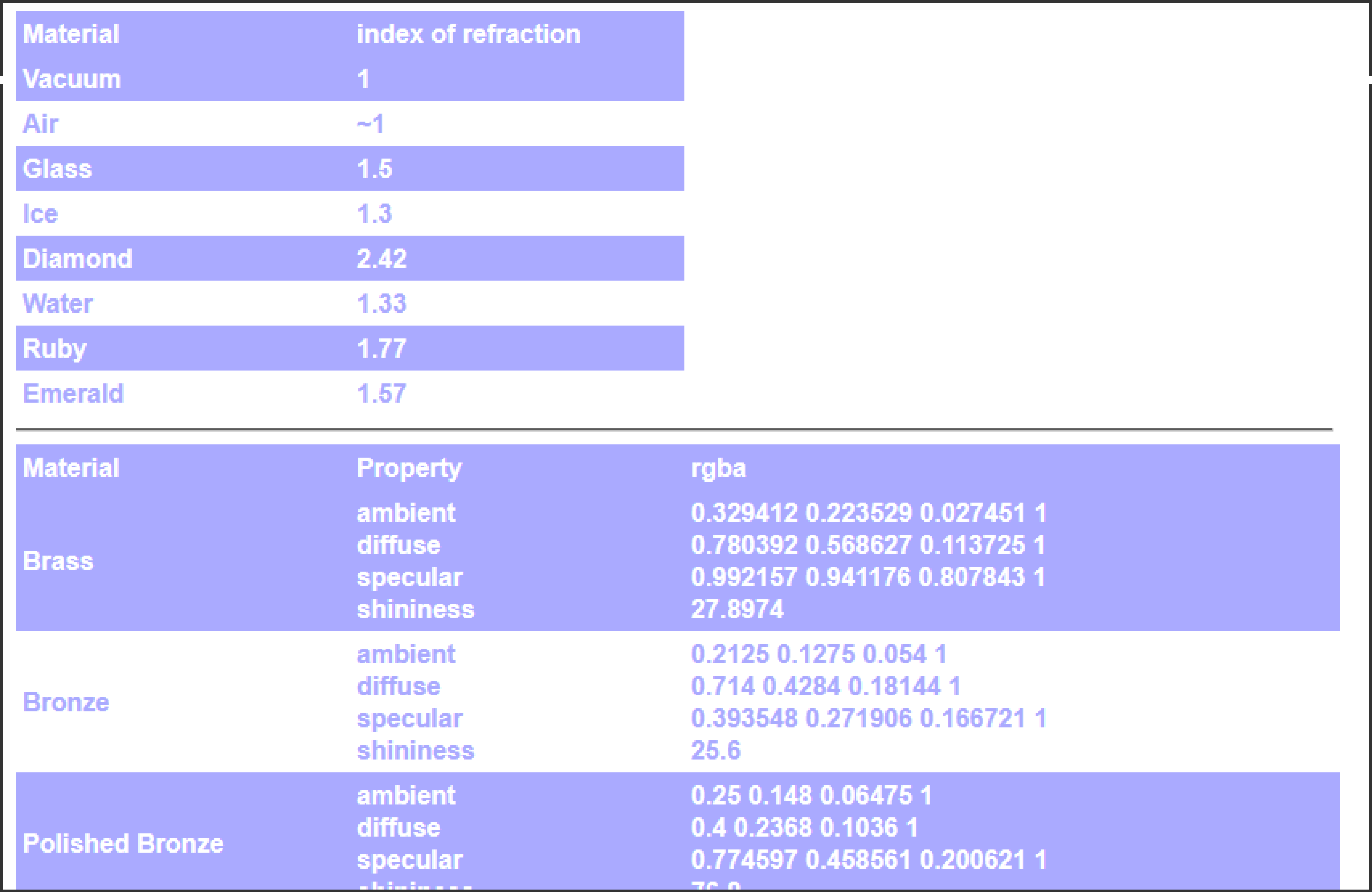


Specular Term

- Glossy objects, specular reflections
- Light reflects close to mirror direction
- Consider half-angle between light and viewer

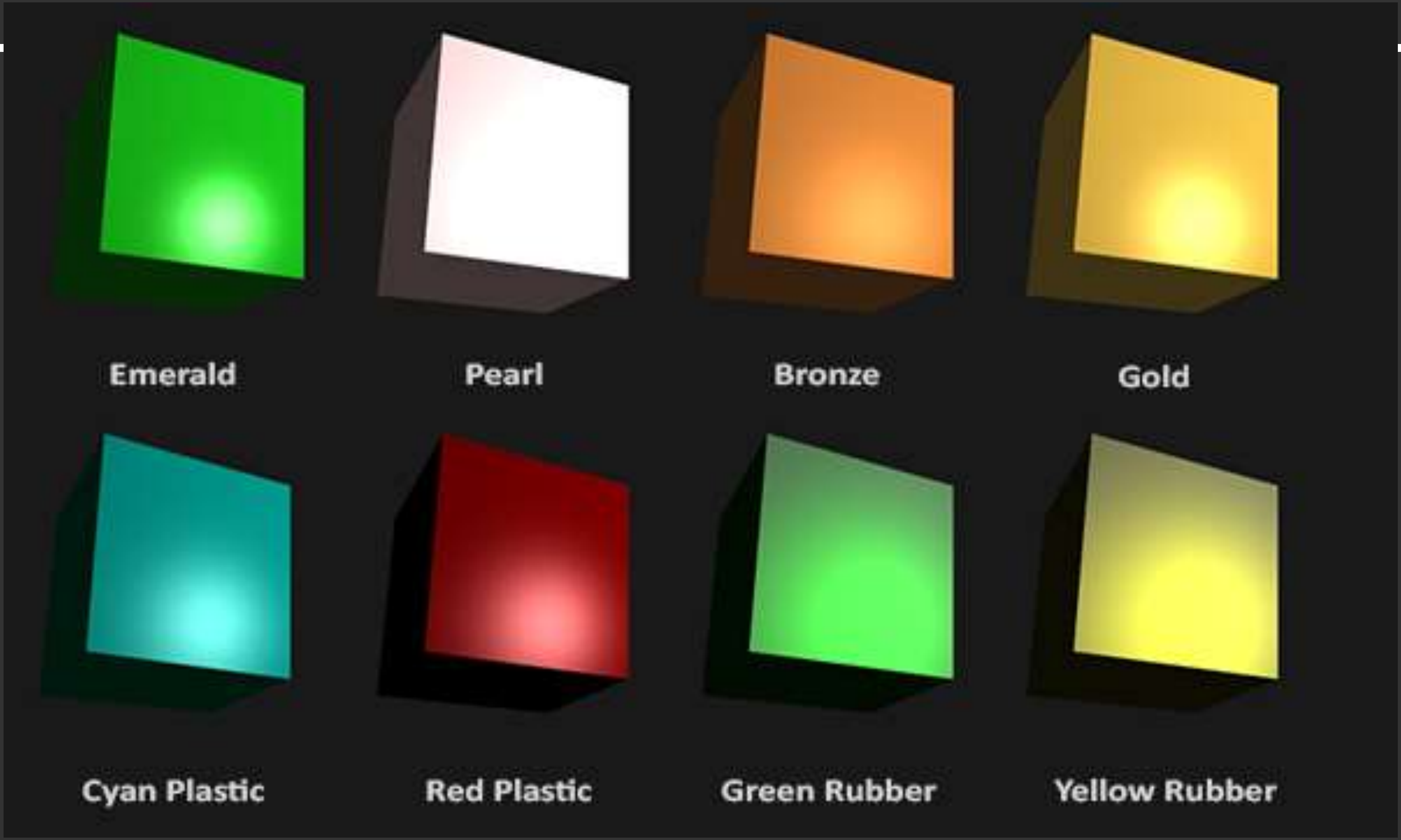


$$I = \sum_{i=0}^{n} specular_{light\ i} * specular_{material} * atten_i * [\max\,(N \bullet s, 0)]^{shininess}$$



Material	index of refraction
Vacuum	1
Air	~1
Glass	1.5
Ice	1.3
Diamond	2.42
Water	1.33
Ruby	1.77
Emerald	1.57

Material	Property	rgba
Brass	ambient	0.329412 0.223529 0.027451 1
	diffuse	0.780392 0.568627 0.113725 1
	specular	0.992157 0.941176 0.807843 1
	shininess	27.8974
Bronze	ambient	0.2125 0.1275 0.054 1
	diffuse	0.714 0.4284 0.18144 1
	specular	0.393548 0.271906 0.166721 1
	shininess	25.6
Polished Bronze	ambient	0.25 0.148 0.06475 1
	diffuse	0.4 0.2368 0.1036 1
	specular	0.774597 0.458561 0.200621 1


Emerald
Pearl
Bronze
Gold
Cyan Plastic
Red Plastic
Green Rubber
Yellow Rubber

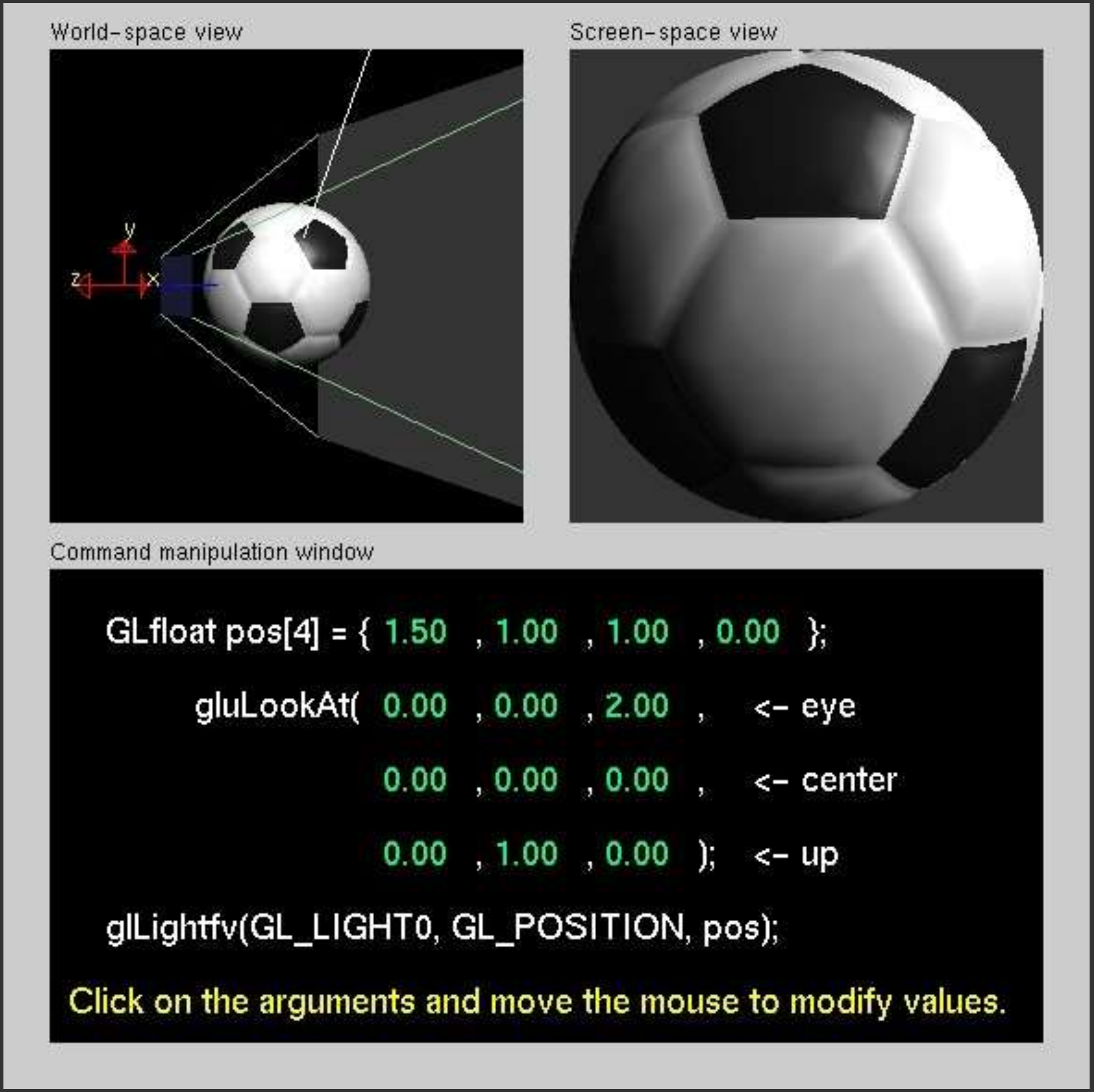
Source Code (in display)

```
/* New for Demo 3; add lighting effects */
  /* See hw1 and the red book (chapter 5) for details */
  {
   GLfloat one[] = {1, 1, 1, 1};
   //    GLfloat small[] = {0.2, 0.2, 0.2, 1};
   GLfloat medium[] = {0.5, 0.5, 0.5, 1};
   GLfloat small[] = {0.2, 0.2, 0.2, 1};
   GLfloat high[] = {100};
   GLfloat light_specular[] = {1, 0.5, 0, 1};
   GLfloat light_specular1[] = {0, 0.5, 1, 1};
   GLfloat light_position[] = {0.5, 0, 0, 1};
   GLfloat light_position1[] = {0, -0.5, 0, 1};

   /* Set Material properties for the teapot */
   glMaterialfv(GL_FRONT, GL_AMBIENT, one);
   glMaterialfv(GL_FRONT, GL_SPECULAR, one);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, medium);
   glMaterialfv(GL_FRONT, GL_SHININESS, high);
```

Source Code (contd)

```
/* Set up point lights, Light 0 and Light 1 */
  /* Note that the other parameters are default values */

  glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
  glLightfv(GL_LIGHT0, GL_DIFFUSE,  small);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);

  glLightfv(GL_LIGHT1, GL_SPECULAR, light_specular1);
  glLightfv(GL_LIGHT1, GL_DIFFUSE,  medium);
  glLightfv(GL_LIGHT1, GL_POSITION, light_position1);

  /* Enable and Disable everything around the teapot */
  /* Generally, we would also need to define normals etc. */
  /* But glut already does this for us */

  glEnable(GL_LIGHTING) ;
  glEnable(GL_LIGHT0) ;
  glEnable(GL_LIGHT1) ;
  if (smooth) glShadeModel(GL_SMOOTH) ; else glShadeModel(GL_FLAT)
 }
```

Lightposition demo



```
GLfloat pos[4] = { 1.50  , 1.00  , 1.00  , 0.00  };
      gluLookAt(  0.00  , 0.00  , 2.00  ,    <- eye
                  0.00  , 0.00  , 0.00  ,    <- center
                  0.00  , 1.00  , 0.00  );   <- up
glLightfv(GL_LIGHT0, GL_POSITION, pos);
```

Click on the arguments and move the mouse to modify values.

Clipping Algorithms

- Point Clipping
- Line Clipping
- Polygon Clipping

2D Viewing Pipe Line

What is CLIPPING WINDOW ?

→Section of 2D scene selected for display – clipping window.

→ the only part of the scene that shows up on the screen is inside the clipping window.

→ All the other parts of the scene outside the selected section is clipped.

→ Also called as world window or viewing window.

2D Viewing Pipe Line

What is VIEWPORT?

→ Another window to control the placement of the clipped scene within the display window.

→Objects inside the clipping window are mapped to the viewport.

→ Viewport is then positioned within the display window

Two-Dimensional Viewing

Co-ordinate Systems.

- **Cartesian – offsets along the x and y axis from (0,0)**
- **Graphic libraries mostly using Cartesian co-ordinates**
- **Four Cartesian co-ordinates systems in computer Graphics.**
  - **1. Modeling co-ordinates**
  - **2. World co-ordinates**
  - **3. Normalized device co-ordinates**
  - **4. Device co-ordinates**

Modeling Coordinates

- **Also known as local coordinate.**
- **Ex: where individual object in a scene within separate coordinate reference frames.**
- **Each object has an origin (0,0)**
- **So the part of the objects are placed with reference to the object's origin.**
- **In term of scale it is user defined, so, coordinate values can be any size.**

World Co-ordinates.

- **The world coordinate system describes the relative positions and orientations of every generated objects.**
- **The scene has an origin (0,0).**
- **The object in the scene are placed with reference to the scenes origin.**
- **World co-ordinate scale may be the same as the modeling co-ordinate scale or it may be different.**
- **However, the coordinates values can be any size (similar to MC)**

Normalized Device Co-ordinates

- **Output devices have their own co-ordinates.**
- **Co-ordinates values: The x and y axis range from 0 to 1**
- **All the x and y co-ordinates are floating point numbers in the range of 0 to 1**
- **This makes the system independent of the various devices coordinates.**
- **This is handled internally by graphic system without user awareness.**

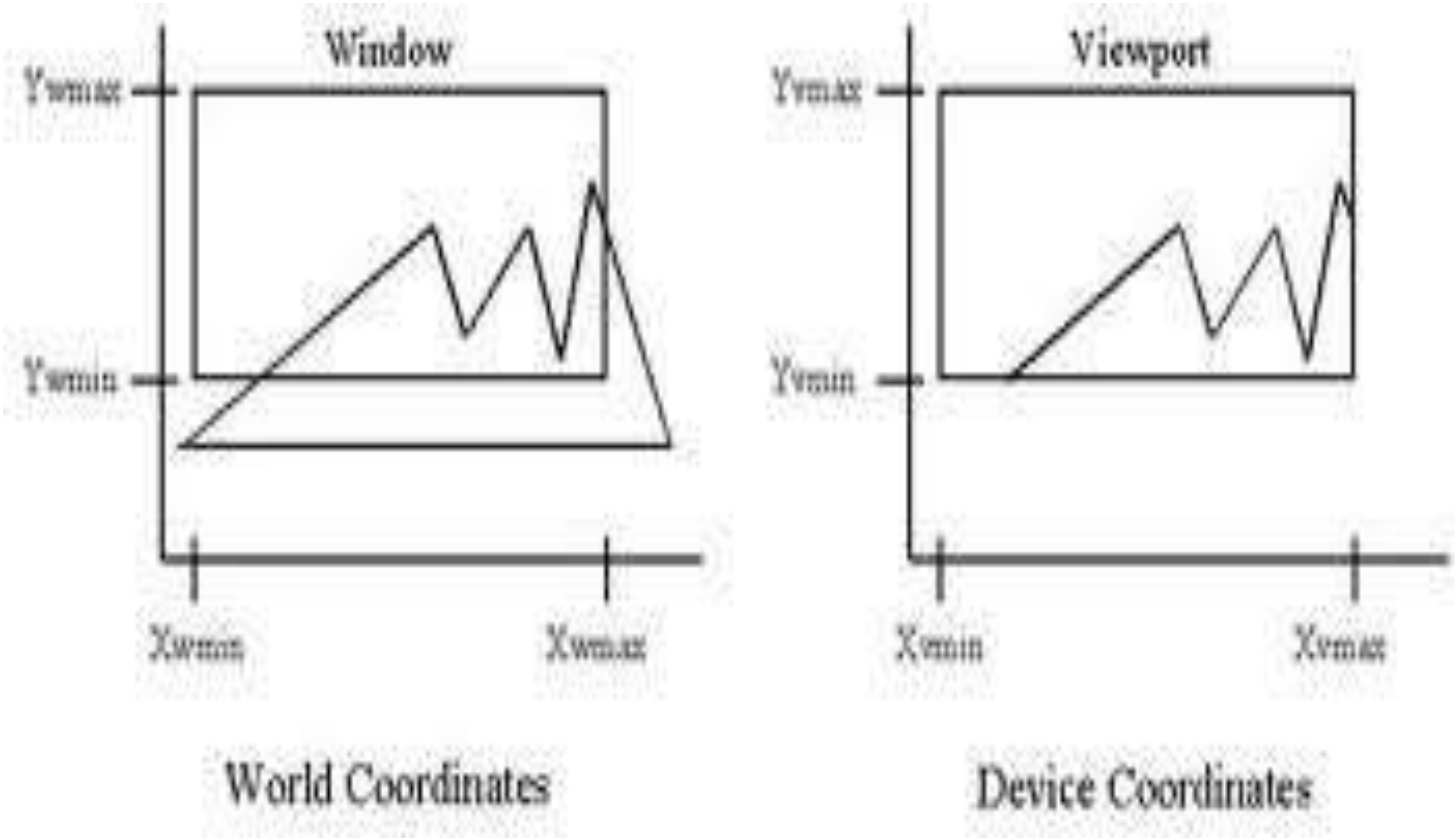
Device Co-ordinates

- **Specific co-ordinates used by a device.**
    - **Pixels on a monitor**
    - **Points on a laser printer.**
    - **mm on a plotter.**
- **The transformation based on the individual device is handled by computer system without user concern.**

Two-Dimensional Viewing

- When we interested to display certain portion of the drawing, enlarge the portion, *windowing* technique is used
- Technique for not showing the part of the drawing which one is not interested is called *clipping*
- An area on the device (ex. Screen) onto which the window will be mapped is called *viewport.*
- Window defines what to be displayed.
- A viewport defines where it is to be displayed.
- Most of the time, windows and viewports are usually rectangles in standard position(i.e aligned with the x and y axes). In some application, others such as general polygon shape and circles are also available
- However, other than rectangle will take longer time to process.

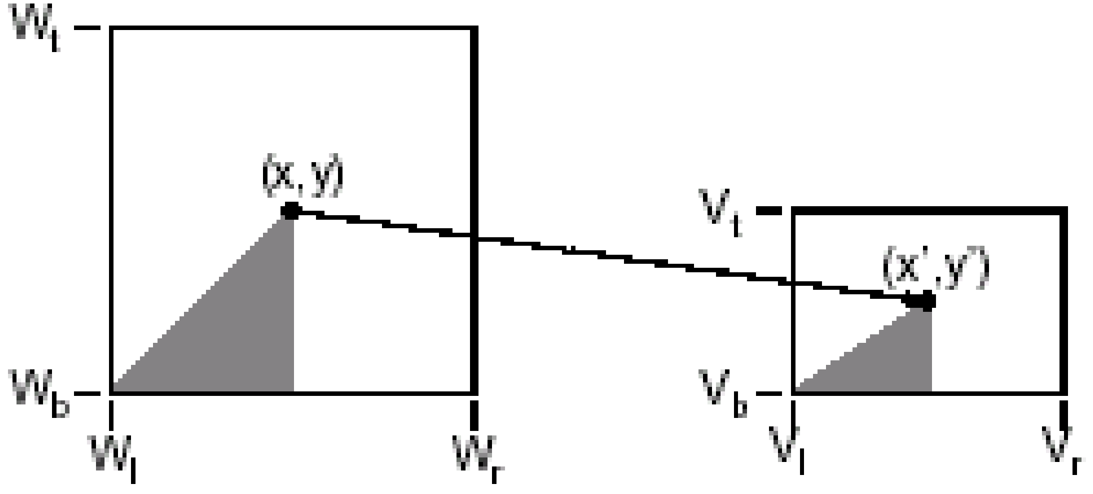
Two-Dimensional Viewing



Viewing Transformation

- ***Viewing transformation*** **is the mapping of a part of a world-coordinate scene to device coordinates.**
- **In 2D (two dimensional) viewing transformation is simply referred as the *window-to-viewport transformation* or the *windowing transformation*.**
- **Mapping a window onto a viewport involves converting from one coordinate system to another.**
- **If the window and viewport are in standard position, this just**
  - **involves translation and scaling.**
  - **if the window and/or viewport are not in standard, then extra transformation which is rotation is required.**

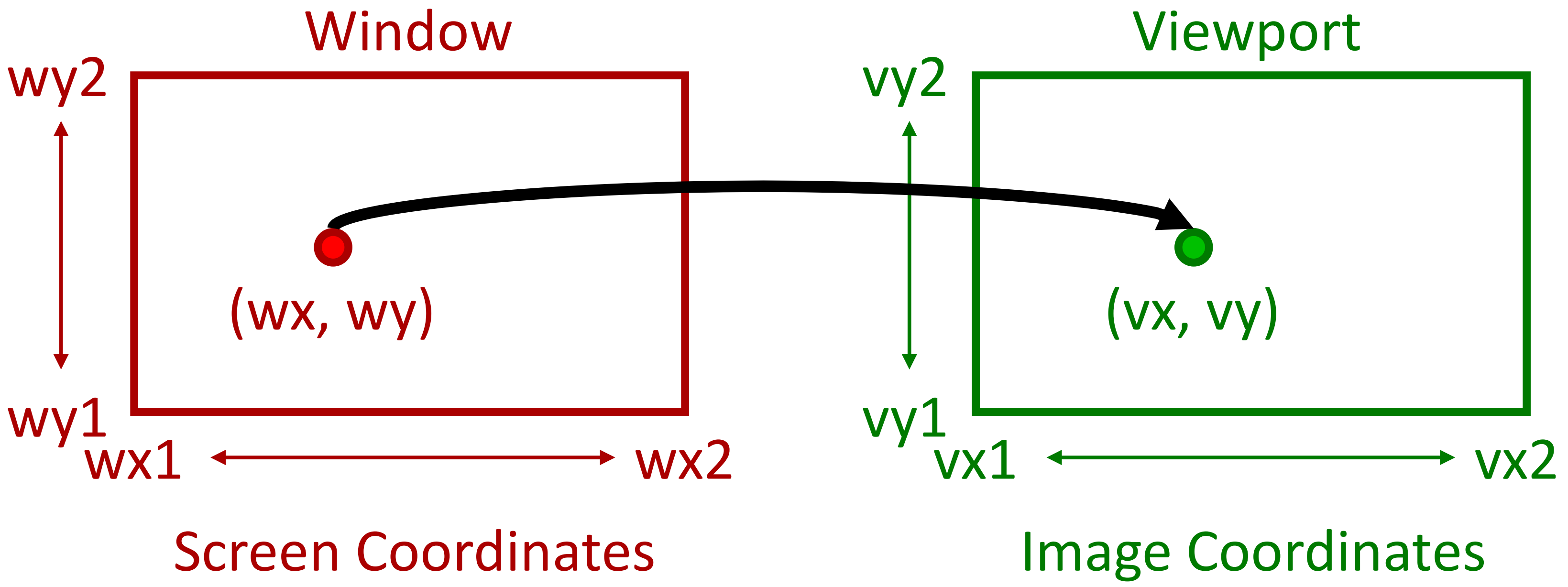
Window-To-Viewport Coordinate Transformation



Window-to-Viewport transformation

Viewport Transformation

- Window-to-Viewport Mapping



```
vx = vx1 + (wx - wx1) * (vx2 - vx1) / (wx2 - wx1);
vy = vy1 + (wy - wy1) * (vy2 - vy1) / (wy2 - wy1);
```

Window-To-Viewport Coordinate Transformation

The sequence of transformations are:

1. **Perform a scaling transformation using a fixed-point position of ($xw_{min,}yw_{min}$) that scales the window area to the size of the viewport.**

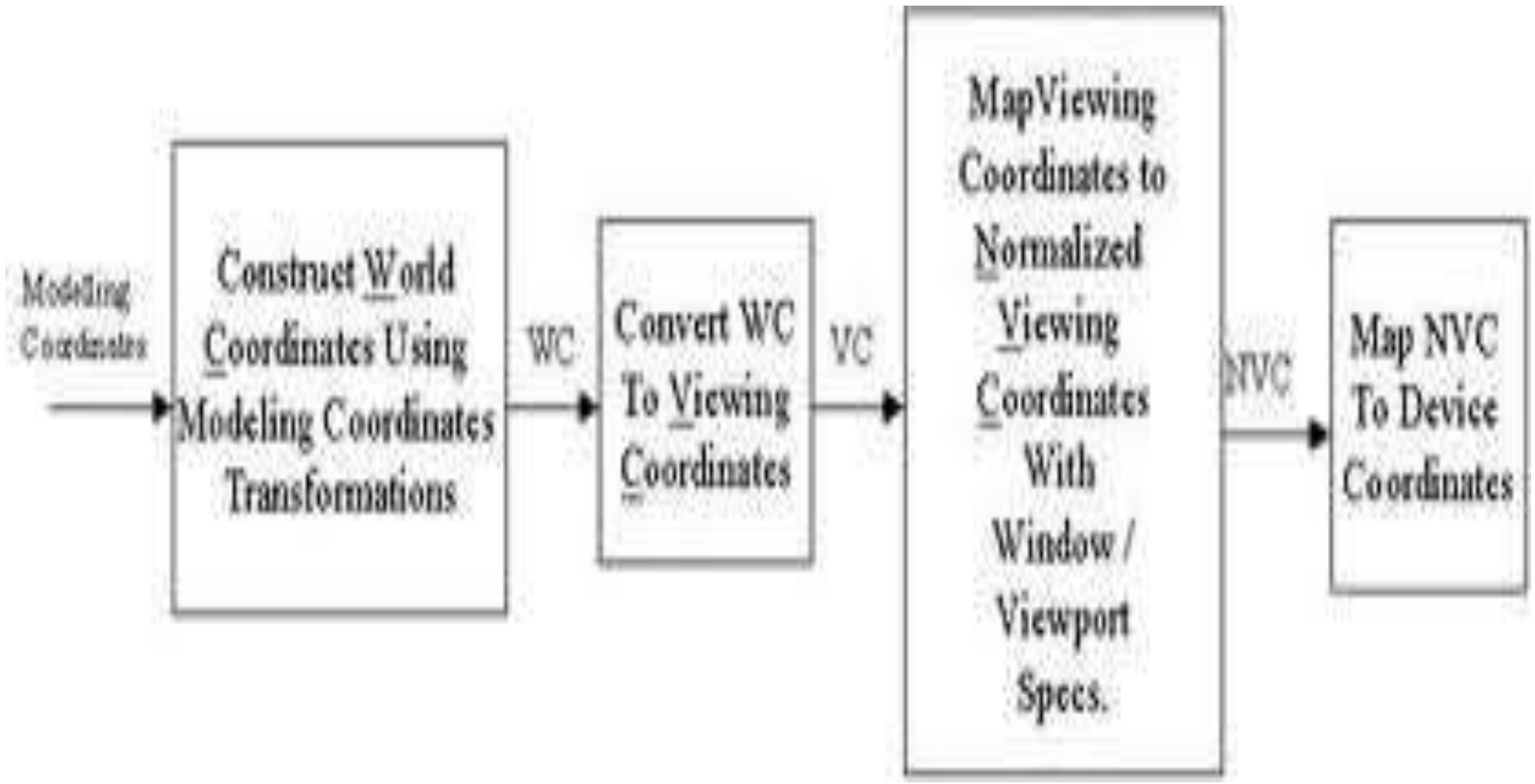
2. **Translate the scaled window area to the position of the viewport.**

Viewport-to-Normalized Device Coordinate Transformation

- **From normalized coordinates, object descriptions can be mapped to the various display devices**

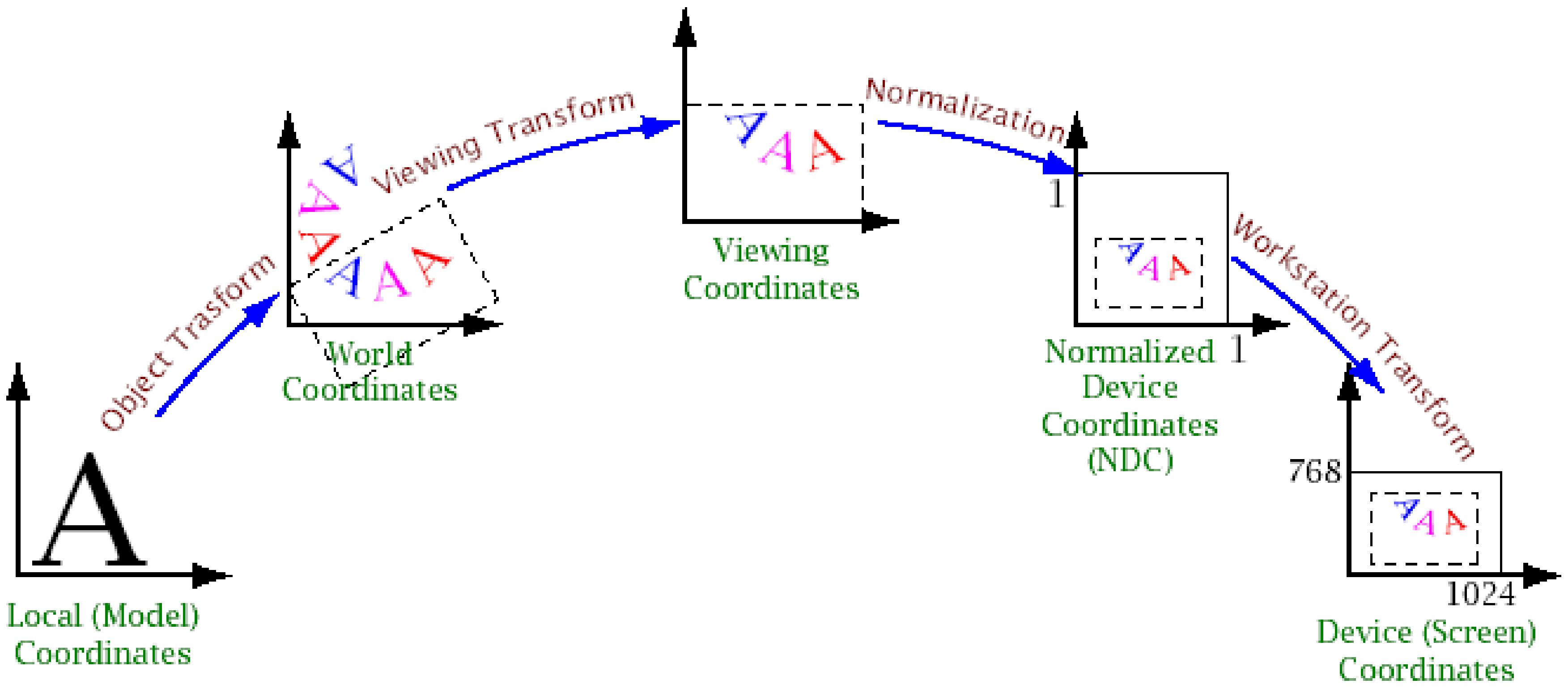
- **When mapping window-to-viewport transformation is done to different devices from one normalized space, it is called *workstation transformation*.**

Two-Dimensional Viewing Pipe Line


Modeling Coordinates
Construct World Coordinates Using Modeling Coordinates Transformations
WC
Convert WC To Viewing Coordinates
VC
MapViewing Coordinates to Normalized Viewing Coordinates With Window / Viewport Specs.
NVC
Map NVC To Device Coordinates

The Viewing Pipeline

Local C. $\longrightarrow$ World C. $\longrightarrow$ Viewing C. $\longrightarrow$ N.D.C. $\longrightarrow$ Device C.



Clipping Algorithm

The primary use of clipping in computer graphics is to remove objects, lines, or line segments that are outside the viewing pane. The viewing transformation is insensitive to the position of points relative to the viewing volume − especially those points behind the viewer − and it is necessary to remove these points before generating the view.

Point Clipping

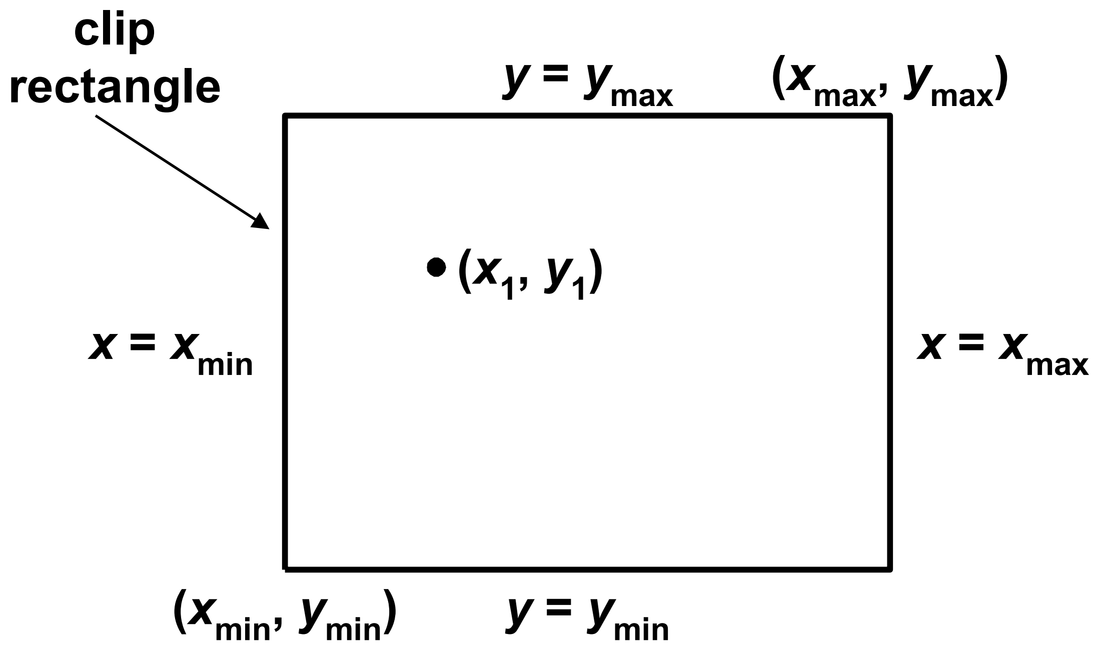
For a clipping rectangle in standard position, we save a 2d point P=(x,y) for display if the following inequalities are satisfied:

$$xwmin \le x \le xwmax$$

$$ywmin \le y \le ywmax$$

If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

Point Clipping



For a point (x,y) to be inside the clip rectangle:

$$x_{\min} \le x \le x_{\max}$$

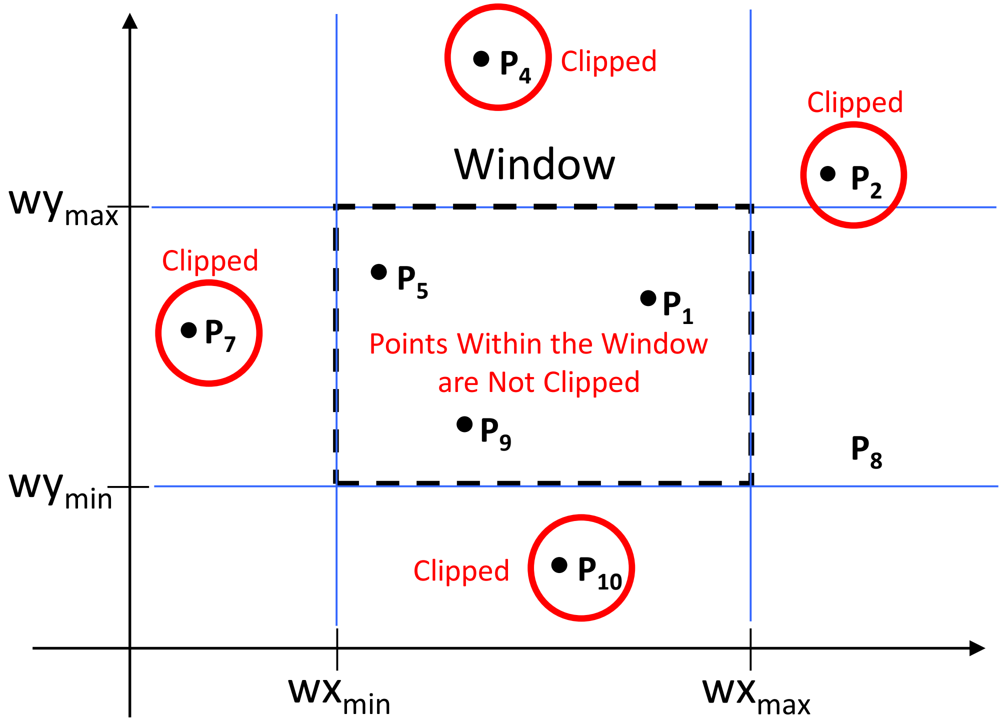
$$y_{\min} \le y \le y_{\max}$$

Point Clipping

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

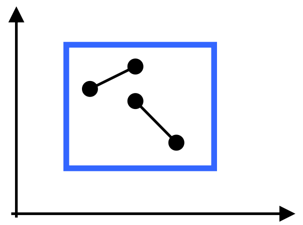
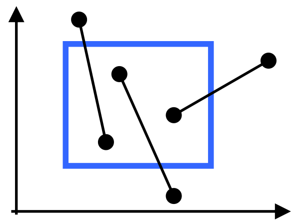
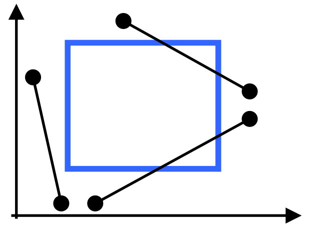
$$wx_{min} \leq x \leq wx_{max} \text{ AND } wy_{min} \leq y \leq wy_{max}$$

otherwise it is clipped



Line Clipping

Harder - examine the end-points of each line to see if they are in the window or not

Situation	**Solution**	**Example**
Both end-points inside the window	Don't clip	
One end-point inside the window, one outside	Must clip	
Both end-points outside the window	Don't know!	

Cohen-Sutherland Clipping Algorithm

- 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.

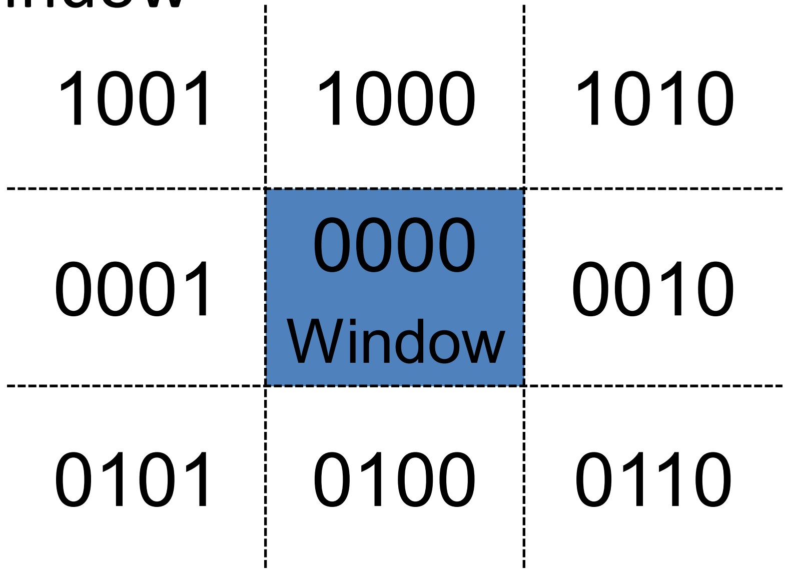
Cohen is something of a mystery – can anybody find out who he was?

Cohen-Sutherland: World Division

- 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

3	2	1	0
above	below	right	left

Region Code Legend



Cohen-Sutherland Algorithm

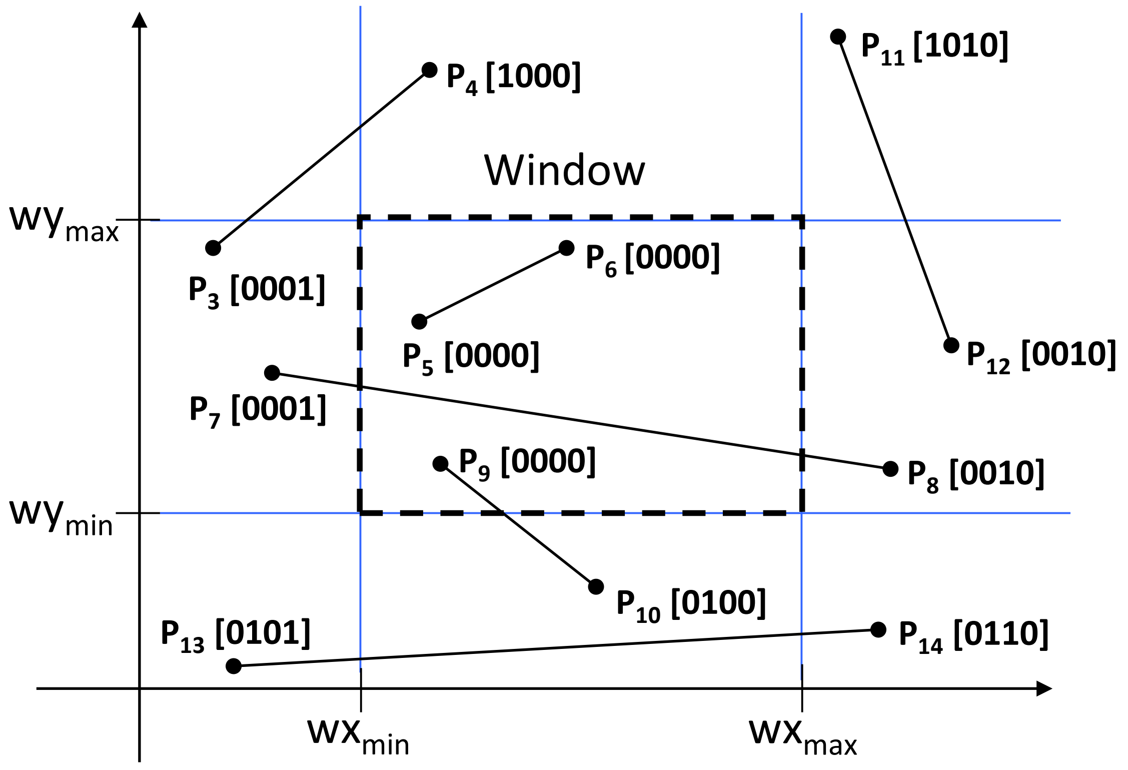
The Cohen-Sutherland Line-Clipping Algorithm performs initial tests on a line to determine whether intersection calculations can be avoided.

1. **First, end-point pairs are checked for Trivial Acceptance.**
2. **If the line cannot be trivially accepted, region checks are done for Trivial Rejection.**
3. **If the line segment can be neither trivially accepted or rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected.**

– **These three steps are performed iteratively until what remains can be trivially accepted or rejected.**

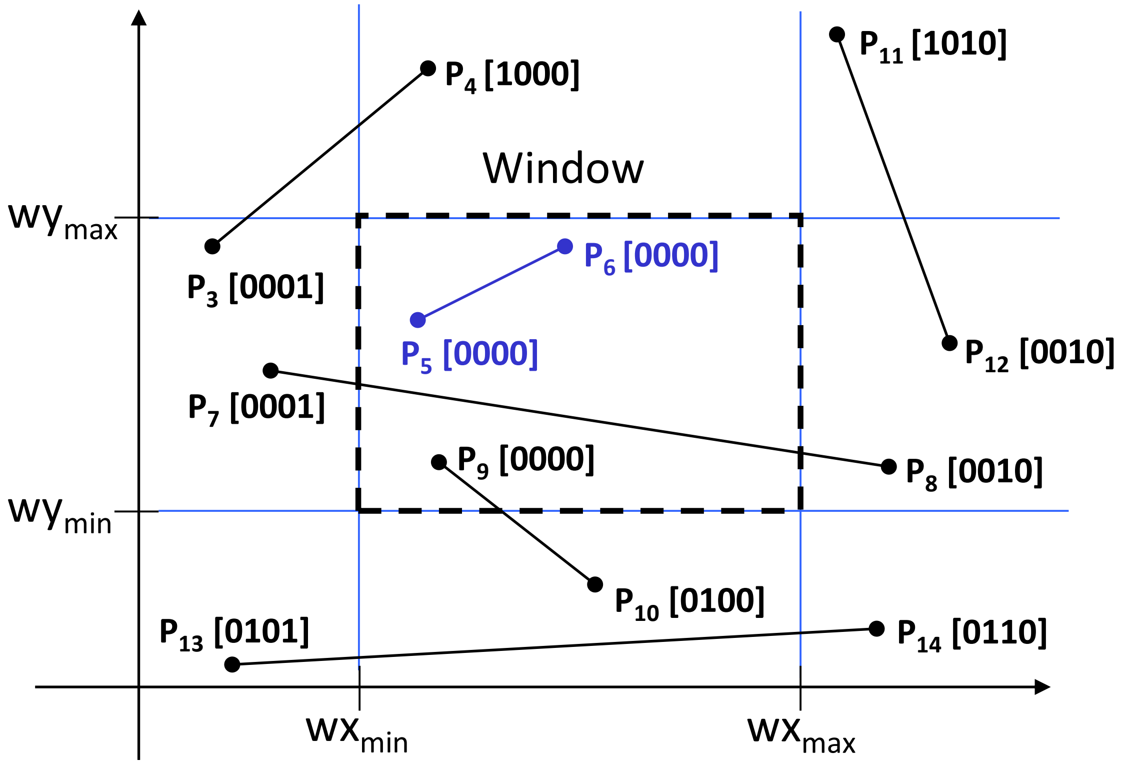
Cohen-Sutherland: Labelling

- Every end-point is labelled with the appropriate region code



Cohen-Sutherland: Lines In The Window

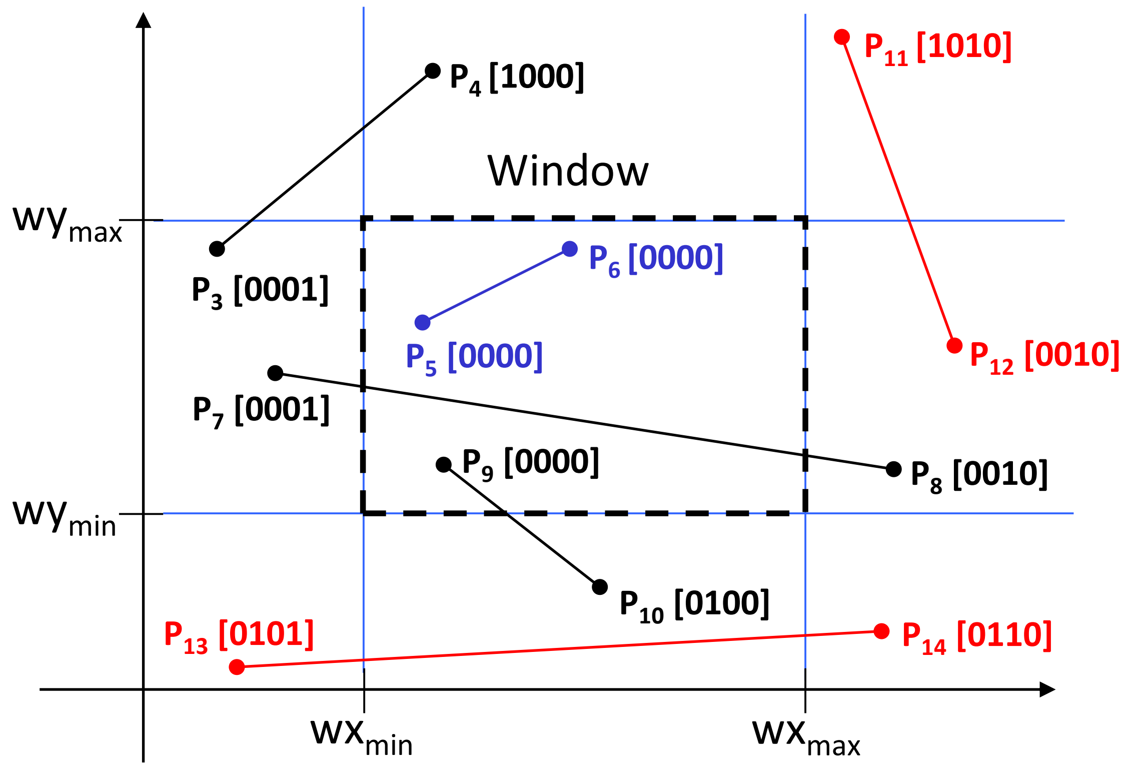
Lines completely contained within the window boundaries have region code [0000] for both end-points so are not clipped



Cohen-Sutherland: Lines Outside The Window

Any lines with a common set bit in the region codes of both end-points can be clipped

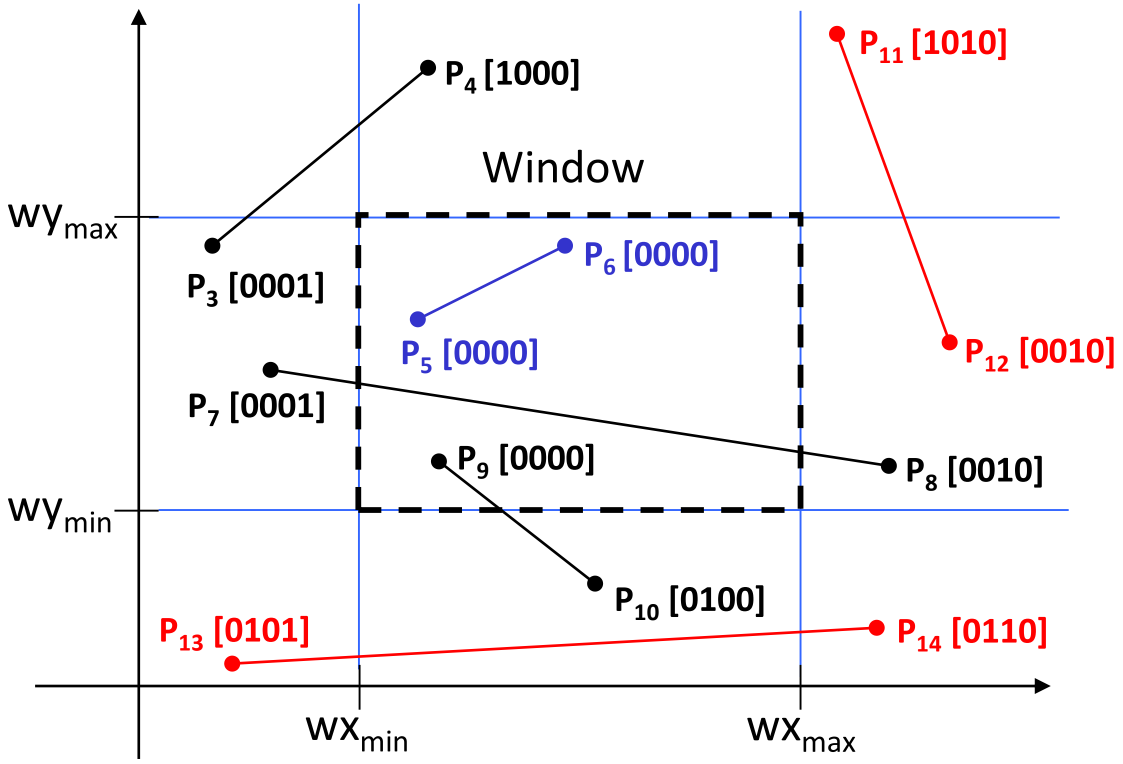
- The AND operation can efficiently check this



Cohen-Sutherland: Lines Outside The Window

Any lines with a common set bit in the region codes of both end-points can be clipped

- The AND operation can efficiently check this



Cohen-Sutherland: Other Lines

- **Lines that cannot be identified as completely inside or outside the window may or may not cross the window interior**
- **These lines are 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**

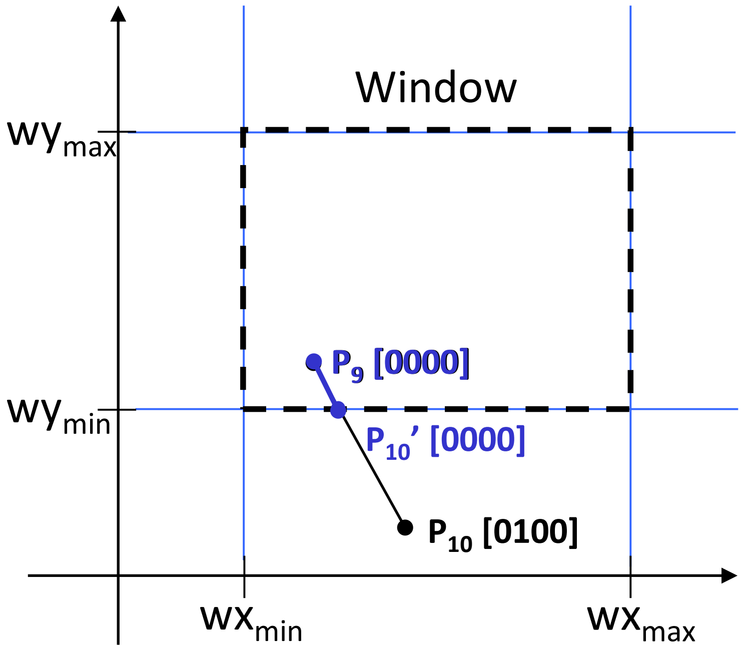
Cohen-Sutherland: Other Lines (cont…)

  - 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

- 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

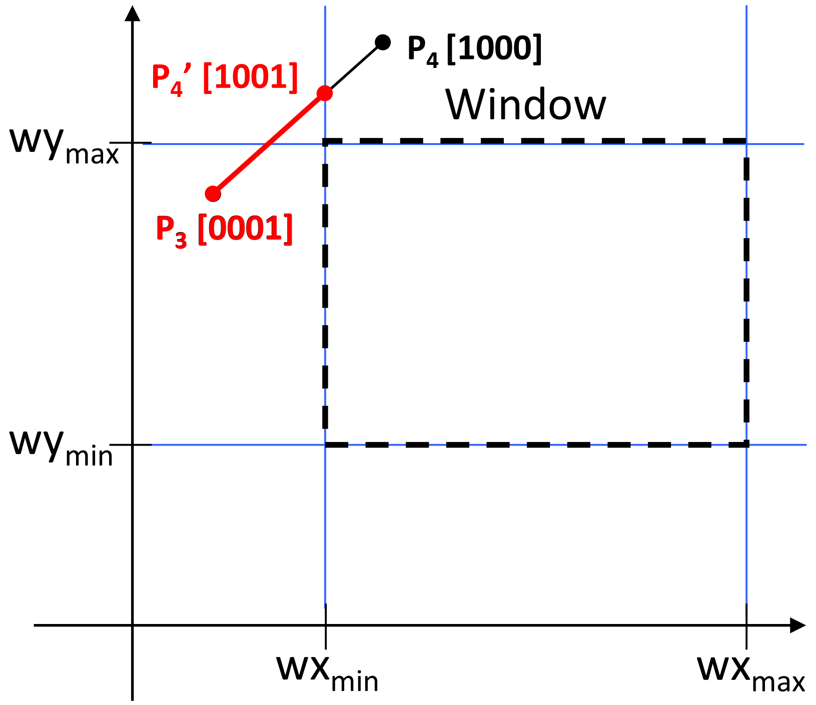
Cohen-Sutherland Examples

- 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 Examples (cont…)

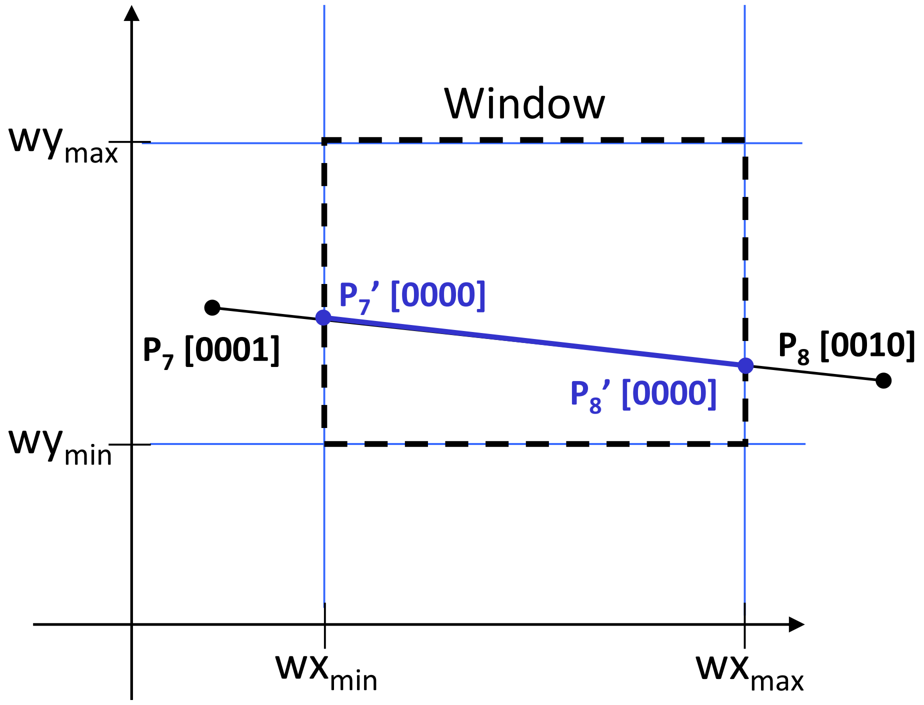
- Consider the line P_3 to P_4 below
  - Start at P_4
  - From the region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_4'



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

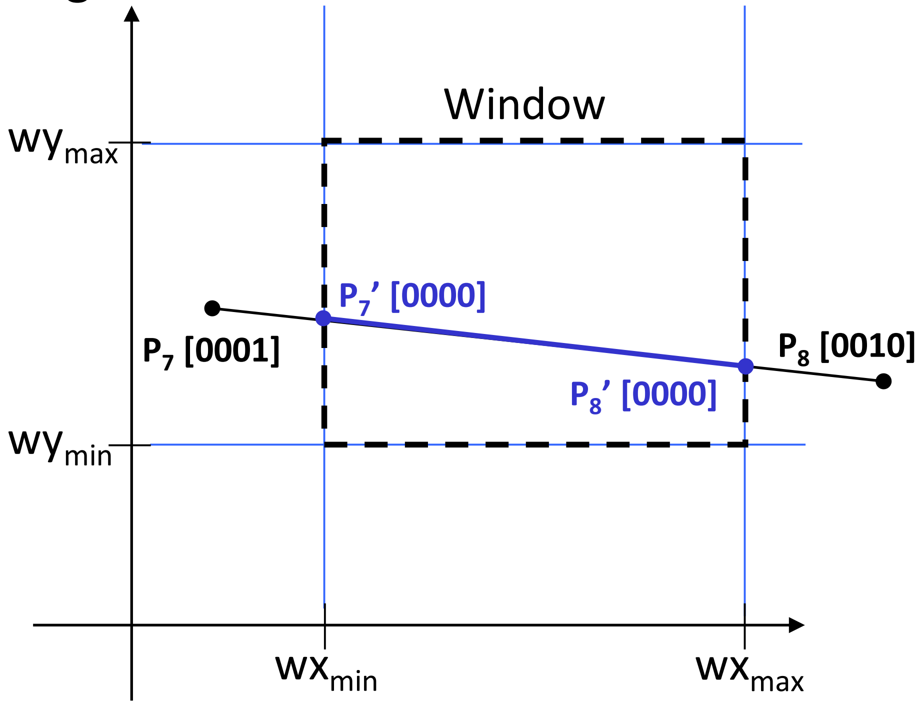
Cohen-Sutherland Examples (cont…)

- 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 Examples (cont…)

- 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



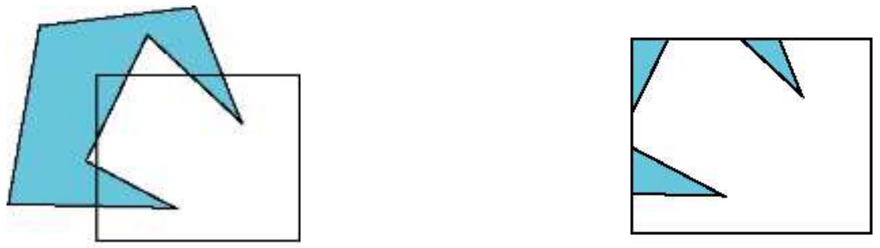
Cohen-Sutherland Algorithm

- Assumes the form:
  - $y = y_0 + slope * (x - x_0)$
  - $x = x_0 + (1/slope) * (y - y_0)$

```
Algorithm (float x0, y0, x1, y1)
    ComputeOutCode(x0, y0, outcode0);
    ComputeOutCode(x1, y1, outcode1);
    repeat
      check for trivial reject or trivial accept
      pick the point that is outside the clip rectangle

      if TOP then
          x = x0 + (x1 – x0) * (ymax – y0)/(y1 – y0); y = ymax;
      else if BOTTOM then
          x = x0 + (x1 – x0) * (ymin – y0)/(y1 – y0); y = ymin;
      else if RIGHT then
          y = y0 + (y1 – y0) * (xmax – x0)/(x1 – x0); x = xmax;
      else if LEFT then
          y = y0 + (y1 – y0) * (xmin – x0)/(x1 – x0); x = xmin;

      if (x0, y0 is the outer point) then
          x0 = x; y0 = y; ComputeOutCode(x0, y0, outcode0)
      else
          x1 = x; y1 = y; ComputeOutCode(x1, y1, outcode1)

    until done
    DrawRectangle(xmin, ymin, xmax, ymax)
    DrawLine (x0, y0, x1, y1)
```

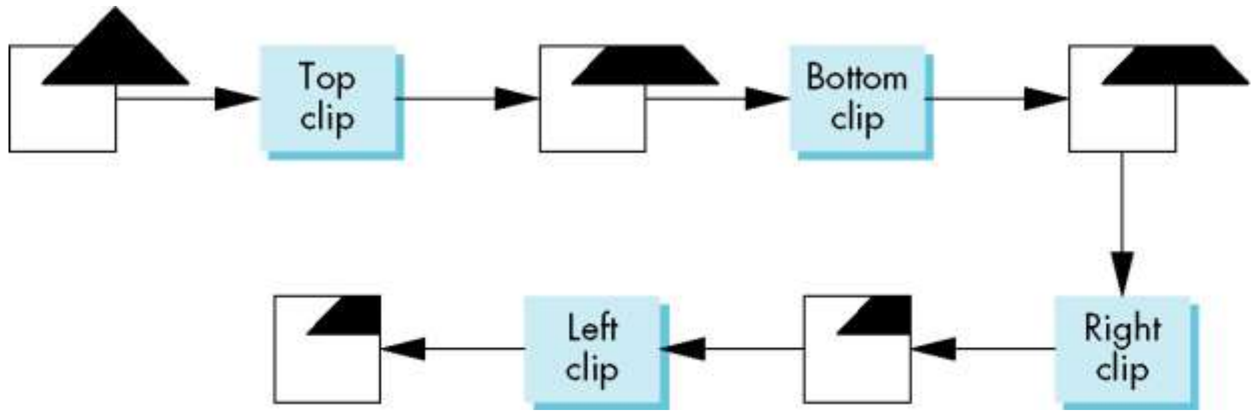
Polygon Clipping

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



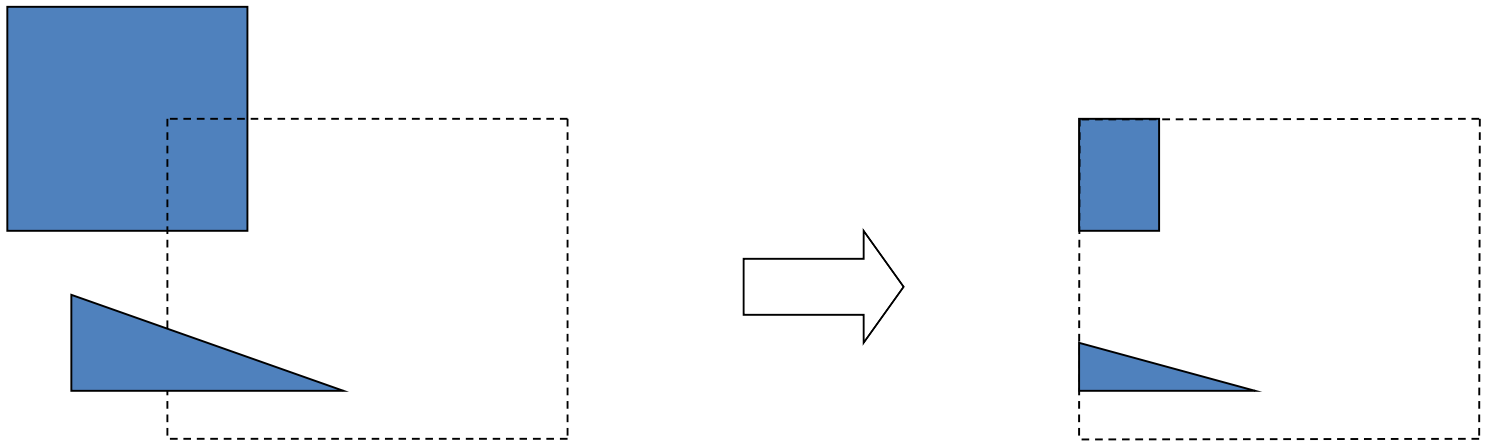
- However, clipping a convex polygon can yield at most one other polygon

Pipeline Clipping of Polygons

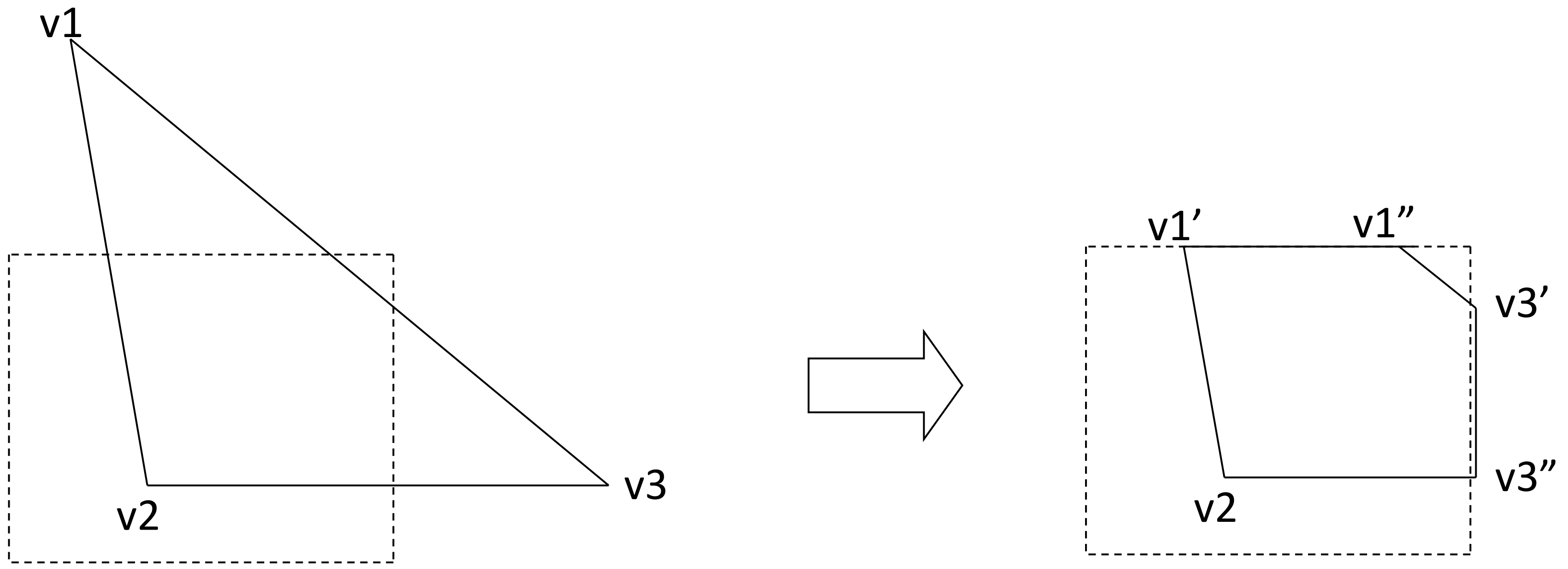


- Three dimensions: add front and back clippers

Polygon Fill-Area Clipping



Polygon Fill-Area Clipping

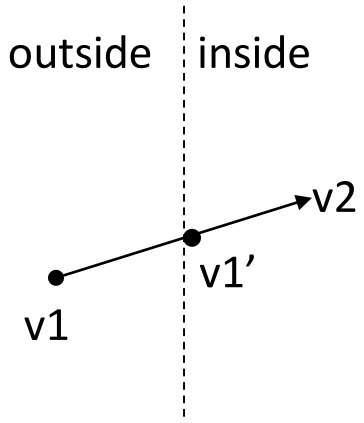


Note: Need to consider each of 4 edge boundaries

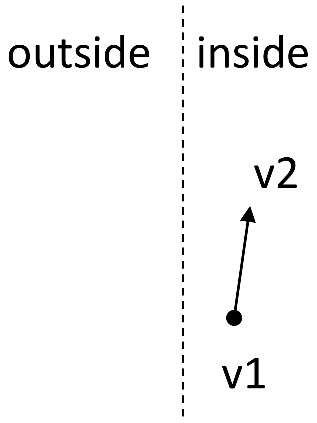
Sutherland-Hodgman Polygon Clipping

- Input each edge (vertex pair) successively.
- Output is a new list of vertices.
- Each edge goes through 4 clippers.
- The rule for each edge for each clipper is:
  - If first input vertex is outside, and second is inside, output the intersection and the second vertex
  - If first both input vertices are inside, then just output second vertex
  - If first input vertex is inside, and second is outside, output is the intersection
  - If both vertices are outside, output is nothing

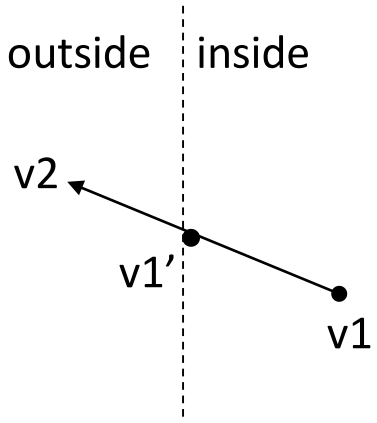
Sutherland-Hodgman Polygon Clipping: Four possible scenarios at each clipper



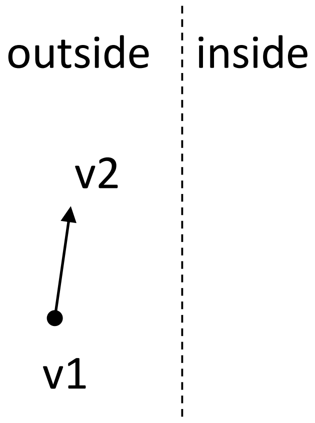
Outside to inside:
Output: v1' and v2



Inside to inside:
Output: v2



Inside to outside:
Output: v1'



Outside to outside:
Output: nothing

Sutherland-Hodgman Polygon Clipping

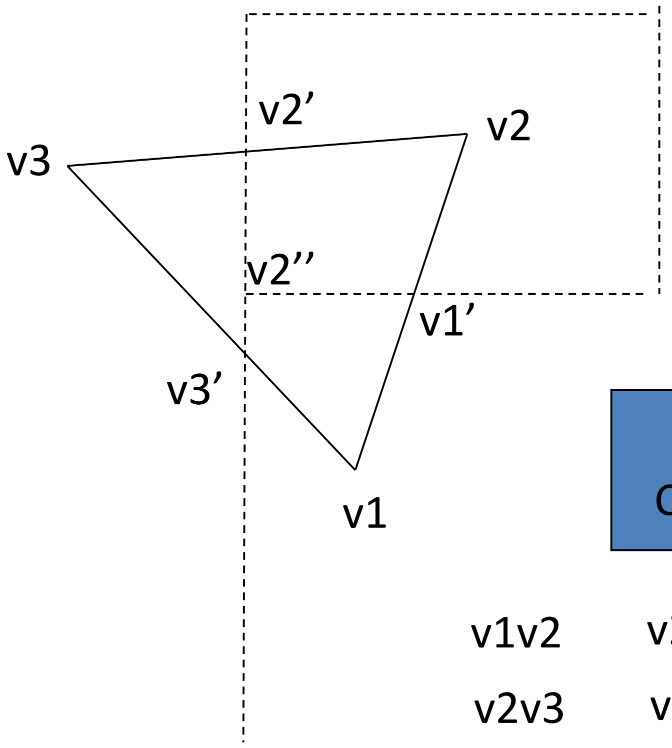
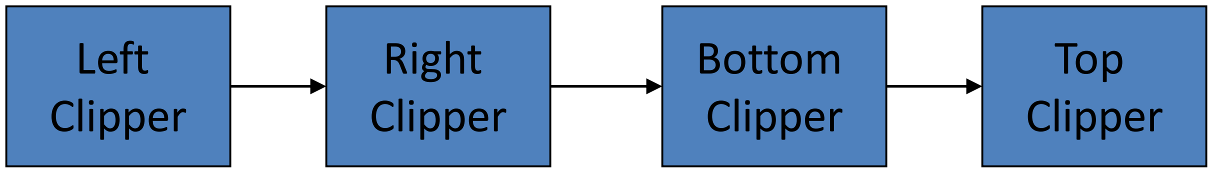


Figure 6-27, page 332



Edges	Output	Edges	Output	Edges	Output	Edges	Output
v1v2	v2	v2v2’	v2’	v2’v3’	v2”	v2”v1’	v1’
v2v3	v2’	v2’v3’	v3’	v3’v1		v1’v2	v2
v3v1	v3’v1	v3’v1	v1	v1v2	v1’v2	v2v2’	v2’
		v1v2	v2	v2v2’	v2’	v2’v2”	v2”

Final