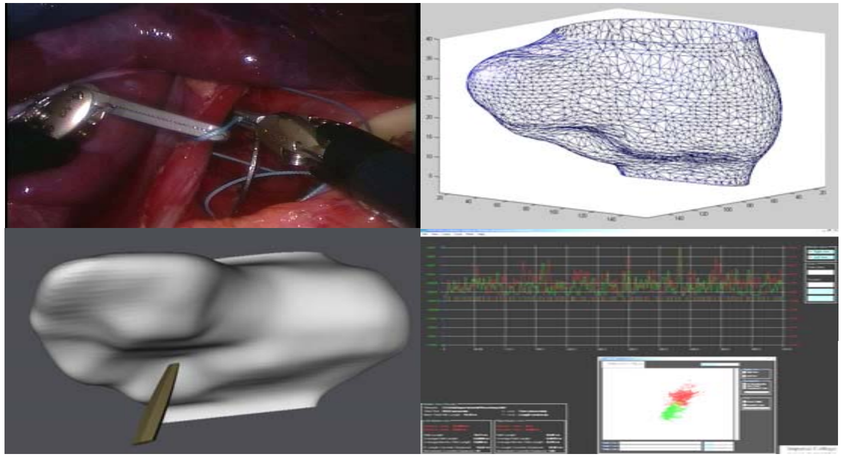
Introduction to Graphics



Spring 2006

Fernando Bello - F.Bello@ic.ac.uk

Surgical
Robotics &
Imaging

Imperial College
London

Course Aim

Give a practical introduction to various mathematical methods employed in Interactive Computer Graphics.

- Use of vectors (and matrices!)
- Dot product
- Cross product
- Unit vectors
- …

(Refer to Mathematical Methods vectors & matrices notes:

http://www.doc.ic.ac.uk/~jb/teaching/mathematical-methods/)

Course Structure

- 8 Lectures (Mon/Tue wks 5-8)
- 4 Tutorials (Tue after lecture)
- In-course assessment (Issued Wk 8 / hand-in 7 Mar)

Course Overview

1. Graphics Input and Output
2. Worlds in 2D and 3D
3. Transformations of 3D Worlds
4. Introduction to OpenGL
5. More Transforms and Homogeneous Coordinates
6. Manipulation of 3D Objects
7. Polygon Rendering
8. Hidden Line Removal

References

- **Interactive Computer Graphics**
  Peter Burger / Duncan Gillies

- Introduction to Computer Graphics
  J D Foley, A van Dam, S K Feiner, J F Hughes and R L Philips

- Computer Graphics, Principles and Practice
  J D Foley, A van Dam, S K Feiner, J F Hughes and R L Philips

- http://www.opengl.org/documentation/

- http://www.opengl.org/resources/

- **Slide presentations, Notes, Tutorials, etc: ~fernando/MMG/**

Mathematical Methods in Computer Graphics

Lecture 1:

Graphical Input and Output

Lecture Overview

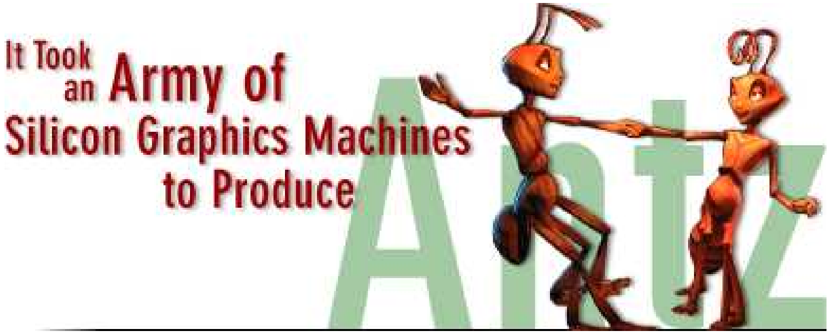
- Why Computer Graphics?
- Input Devices
- Graphics Output Devices
- Raster Graphics
- Device Dependent / Independent Graphics
- World Coordinate System
- Attributes
- Normalisation
- Viewports

What is computer graphics?

- *Creation*, *Storage* and *Manipulation* of models / images
  $\rightarrow$ using computers to generate and display images.
- Form, Appearance, Behaviour.
- Issues that arise:
  - Modelling (form)
  - Rendering (appearance)
  - Animation (behaviour)
  - …

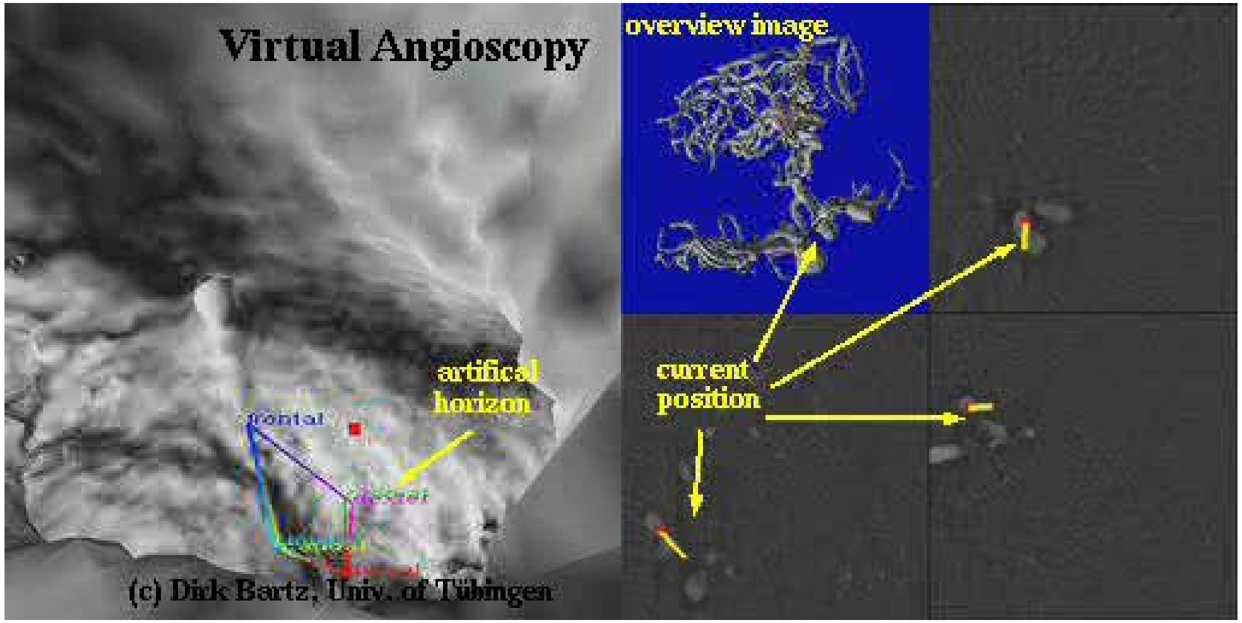
Applications

- Movies
- Games
- Simulation
- Analysis / Visualisation
- Design
- Etc



Number of frames in the movie	119,592
Number of times the movie was rendered during production	15 (approx.)
Number of feet of approved animation produced in a week	107 ft.
Total number of hours of rendering per week	275,000 hrs.
Average size of the frame rendered	6 MB
Total number of Silicon Graphics servers used for rendering	270
Number of desktop systems used in production	166
Total Number of processors used for rendering	700
Average amount of memory per processor	256 MB
Time it would have taken to render this movie on 1 processor	54 yrs., 222 days, 15 mins., 36 secs.
Amount of storage required for the movie	3.2 TB
Amount of frames kept online at any given time	75000 frames
Time to re-film out final cut beginning to end	41.5 days (997 hrs.)

Medical Example



Input Devices

- There are many input devices for computer graphics:

  Mouse
  Joystick
  Button Box
  Digitising Tablet
  Light Pen
  Haptic / Tactile Devices
  etc…

Mouse Position and Visible Markers

- The mouse is a device which supplies the computer with three bytes of information (minimum) at a time:

  Distance Moved in X direction (ticks)

  Distance Moved in Y direction (ticks)

  Button Status

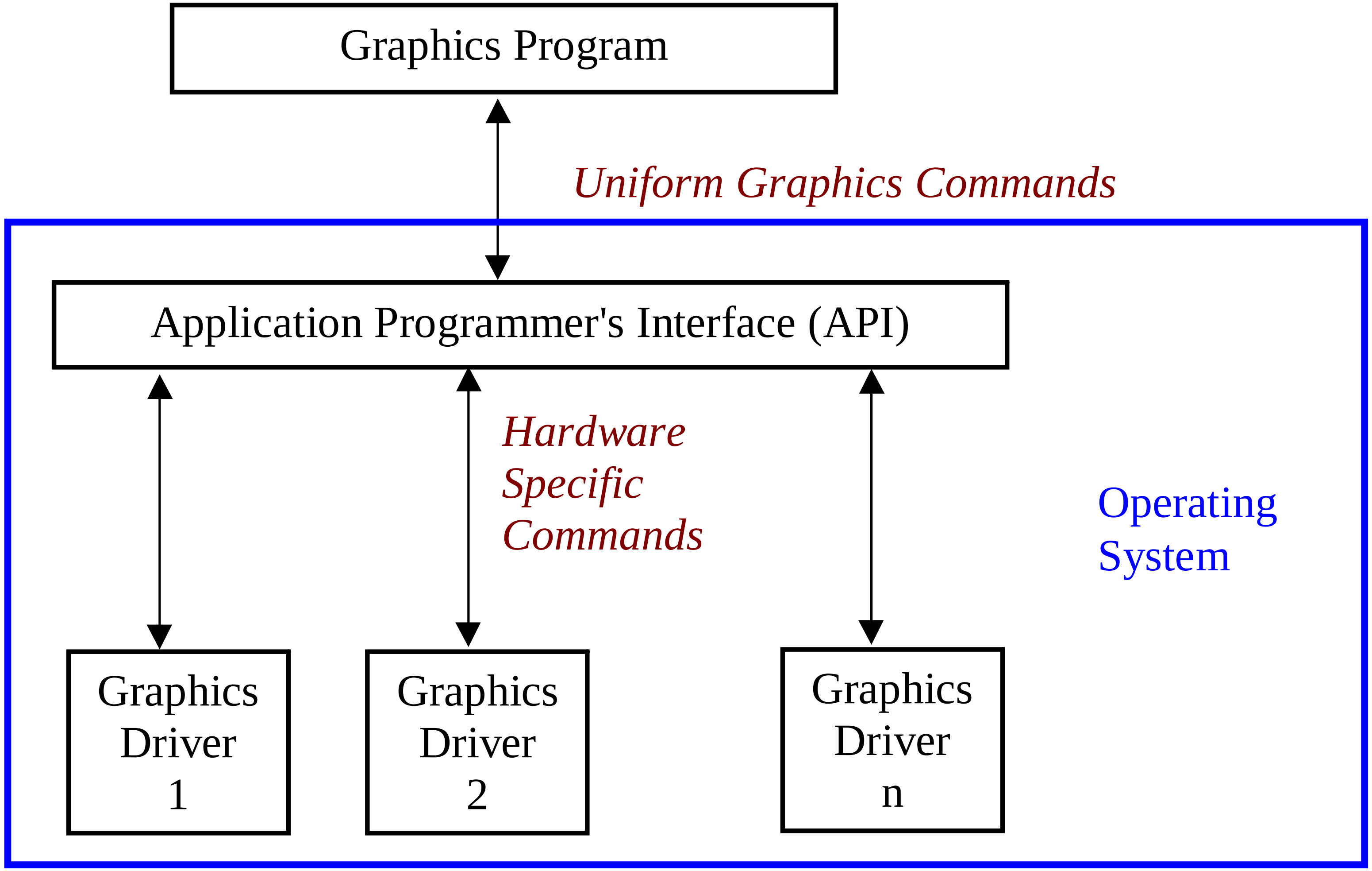
- The provision of a visible marker on the screen is done by software.

Graphics Output Devices

- Graphics output devices are many and diverse.
- Fortunately we don't need to worry too much about them

  $\Rightarrow$ the OS takes care of many of the details

- It provides us with an Application Programmer's Interface (API).
- An API is a set procedures for handling menus windows and, of course, graphics.


Graphics Program
Uniform Graphics Commands
Application Programmer's Interface (API)
Hardware
Specific
Commands
Operating
System
Graphics
Driver
1
Graphics
Driver
2
Graphics
Driver
n

Device Drivers and The API

- Each graphics adapter has a software driver which is loaded into the OS.
- The OS provides a set of graphics primitives (API) that are uniform across all cards
- Unfortunately the API is not standard across systems (but there are emerging standards e.g. OpenGL)

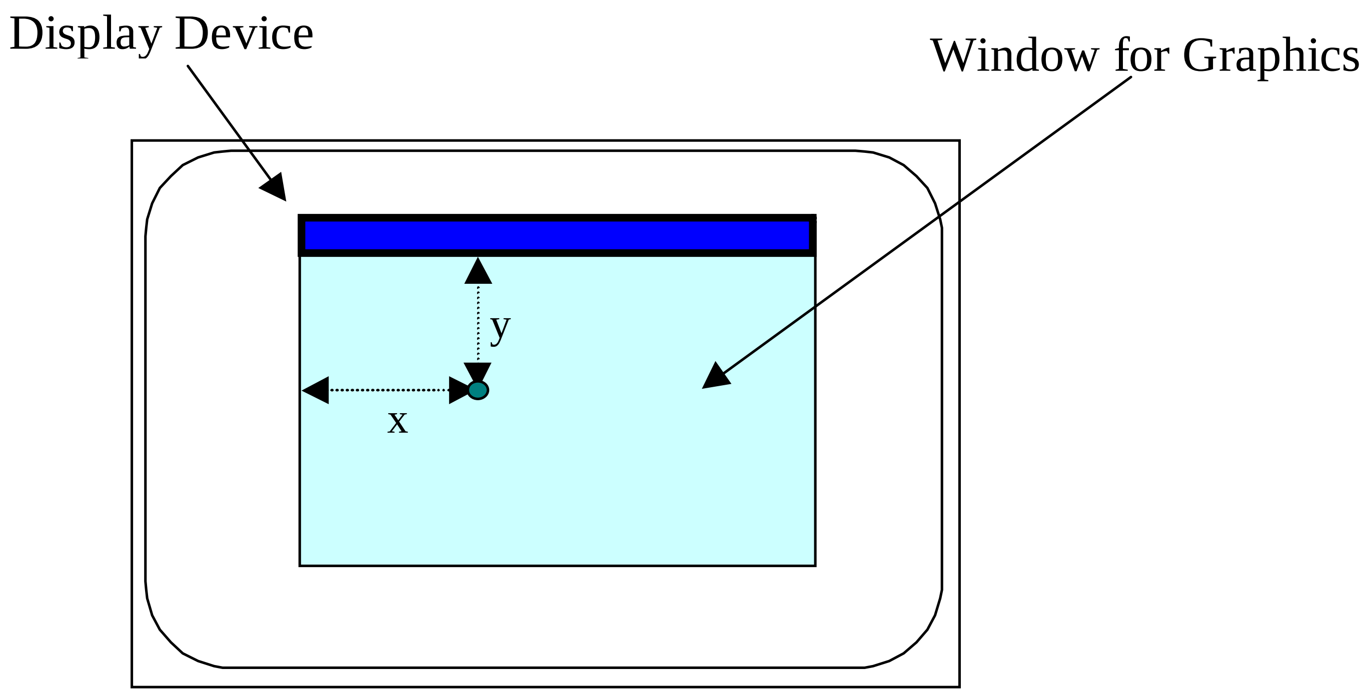
Raster Graphics

The most common graphics device is the raster terminal where the programmer plots points or pixels.

A typical (API) command might be:

SetPixel(x,y,colour) OpenGL - glColor3f(red,green,blue)

Where x and y are pixel coordinates.



Normal meaning for *SetPixel(x,y,green)*

Storing Images

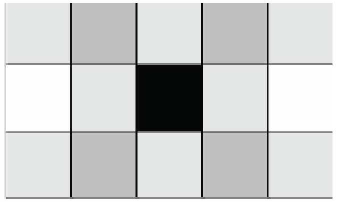
Raster Images

2D Array of memory

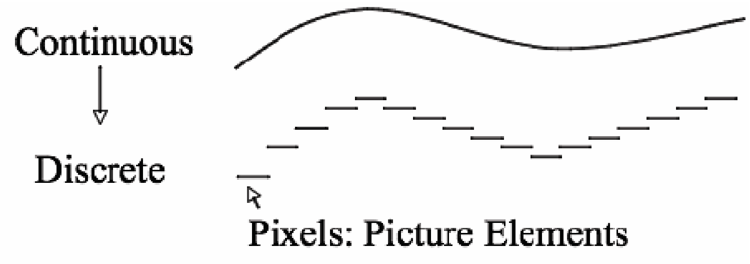
Pixels store different things:

- Intensity (scalar value – float, int)
- RGB Colour (vector value)
- Depth
- Others…

Have discrete pixel *locations* and discrete pixel *values*:



0.25	0.5	0.25	0.5	0.25
1	0.25	0	0.25	1
0.25	0.5	0.25	0.5	0.25



Bits per pixel

- In some cases only one bit is used to represent each pixel allowing it to be on or off.
- Old systems had only 8 bits per pixel allowing 256 different shades to be represented.
- Today, pixels have 24 or 32 bits allowing representation of millions of colours.

Pixel Addressing

- Unfortunately not all systems adopt the same pixel addressing conventions.
- Some have the origin at the top left corner, some have it at the bottom right hand corner…

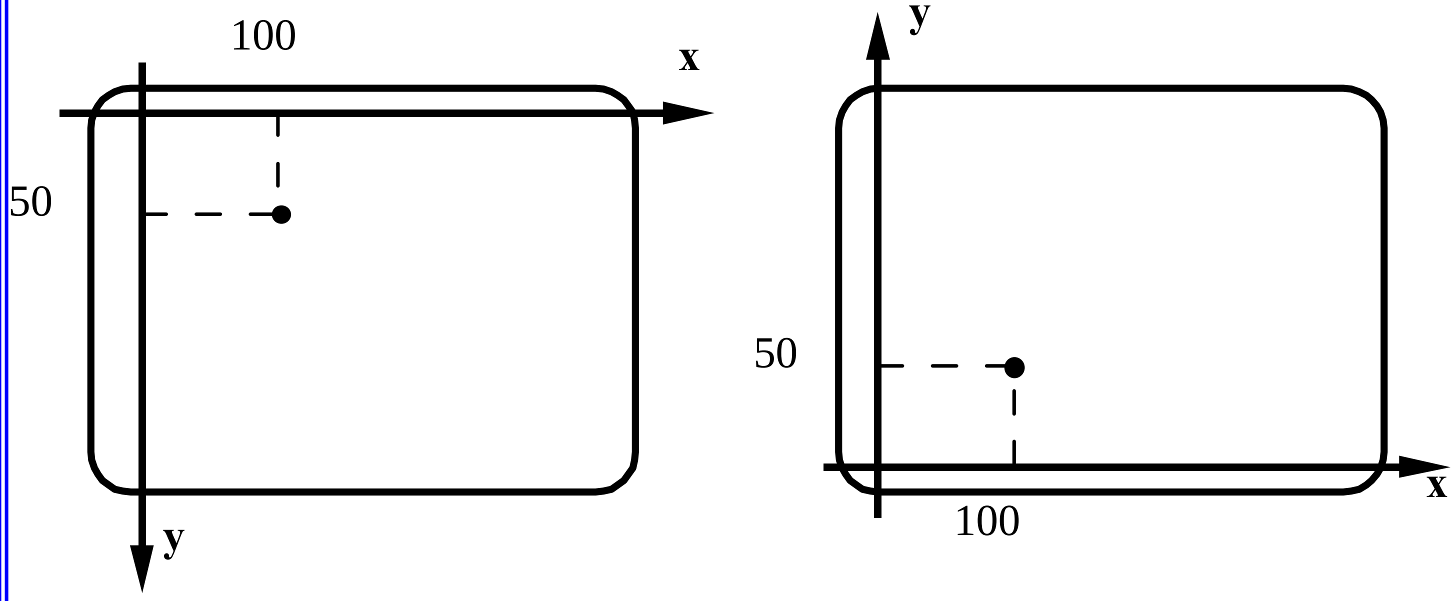


Diagram 1.2
Pixel addressing conventions vary from system to system

Device Independent Graphics

- As a general principle of programming it is best to minimize dependence on hardware.
- However, graphics programmers frequently use device features to accelerate their computations.
- Thus there is a conflict of interest between performance and good programming practice!

Device Dependent Drawing Primitives

Each OS (API) provides us with the possibility of drawing graphics at the pixel level.

In Windows we have:

MoveToEx(hdc xpix, ypix);

LineTo(hdc, xpix, ypix);

TextOut(hdc, xpix, ypix, message, length);

Hdc: an identifier for the window

Xpix, ypix: pixel coordinates

Why aim for better device independence?

1. In normal applications we want our pictures to adjust their size if the window is changed.

2. In graphics-only applications we want our pictures to be independent of resolution

3. Be able to move graphics applications between different systems (PC [Win/Linux], MAC, SUN etc.)

World Coordinate System

To achieve device independence we need to define a world coordinate system.

This will define our drawing area in units that are suited to the application:

- meters
- light years
- microns
- etc

Worlds and Windows

It is common, but not universal to define the world coordinates with the command:

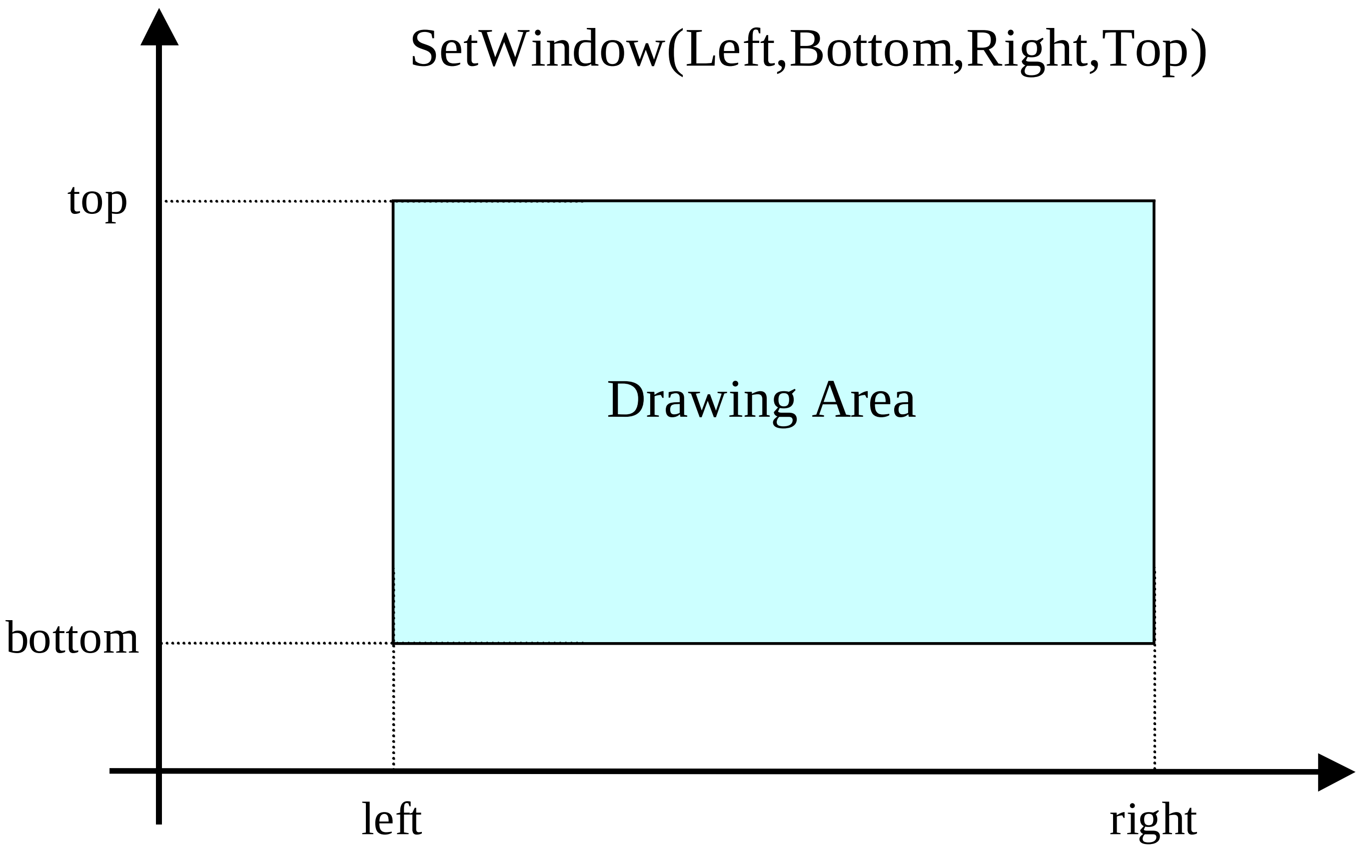
SetWindow(left,bottom,right,top)

glutInitWindowPosition(x,y)
glutWindowSize(width,size)
glutCreateWindow(win)

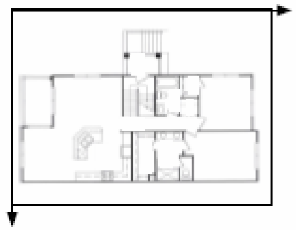
} OpenGL

Think of this as a window onto the world matching a window on the screen

World Coordinates



World Coordinates - Meters



Screen Space - Pixels

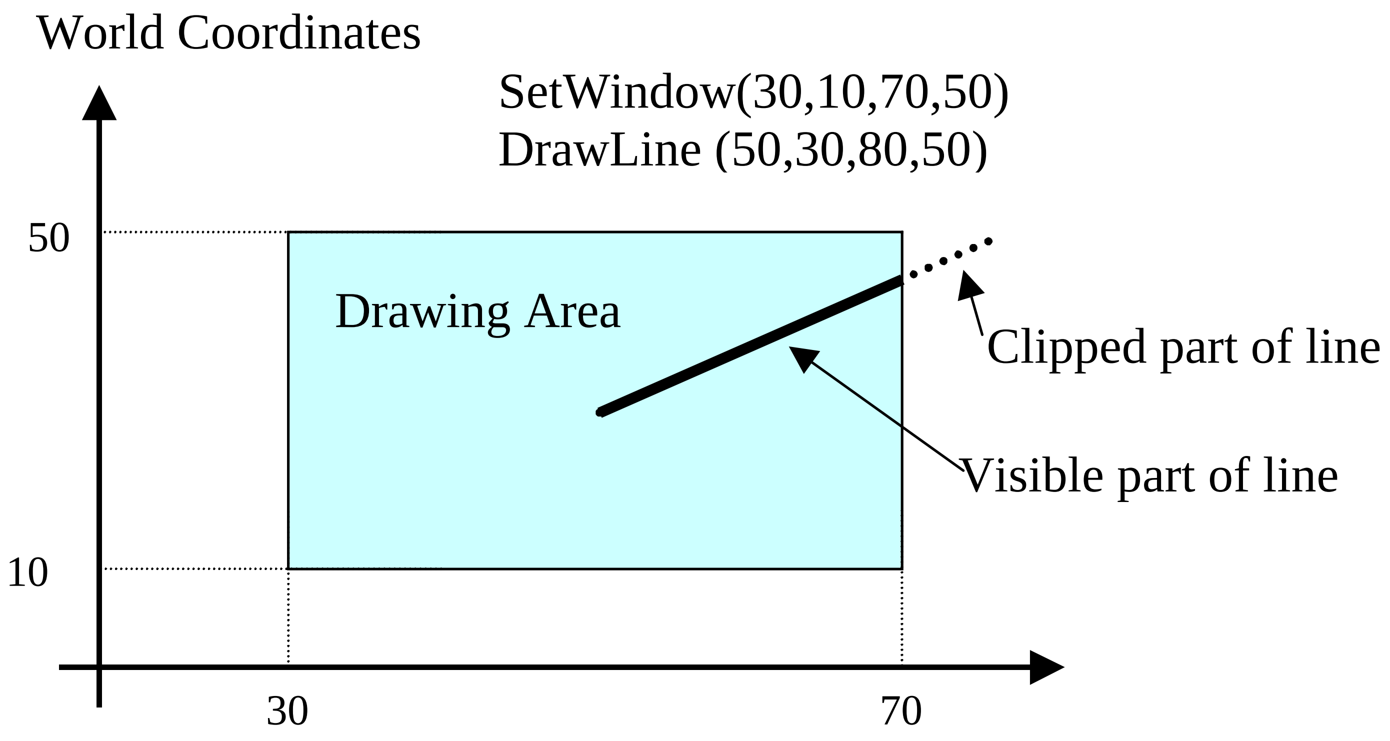
Note possible distortion issues…

Device independent Graphics Primitives

Once our world coordinate system is defined, we can implement drawing primitives to use with it:

```
DrawLine(x1,y1,x2,y2);
DrawCircle(x1,y1,r);
DrawPolygon(PointArray);
DrawText(x1,y1,"A Message");
```

Normally any part of a graphics object outside the window is clipped.


World Coordinates
SetWindow(30,10,70,50)
DrawLine (50,30,80,50)
50
Drawing Area
Clipped part of line
Visible part of line
10
30
70

Normalisation

- Need to connect device independent graphics primitives to device dependent drawing commands.

- This is done by the process of normalisation.

- Translate world coordinates into a set of coordinates that will be suitable for the OS API.

- This is done by a simple linear transformation.

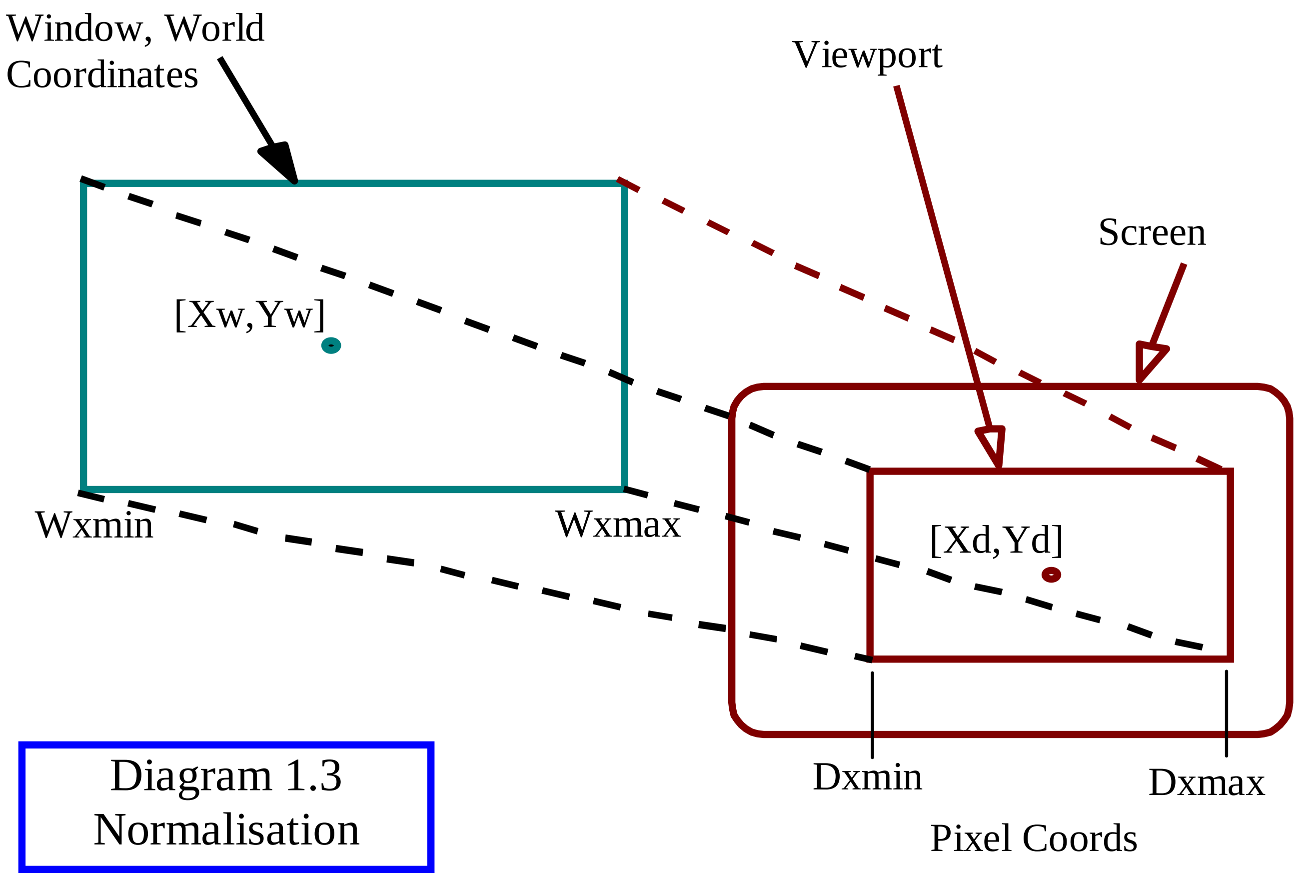


Diagram 1.3
Normalisation

Enquiry Functions

- The user may re-size a window independently of our program.
- Need to enquire the pixel size of our window before we can normalise the coordinates.

➢ Thus we need a command such as:

GetWindowPixelCoords(DXmin,DYmin,DXmax,DYmax)

(GetClientRect in Windows)

Normalisation

Relate world coordinates and device coordinates by simple ratios:

$$\frac{(Xw\text{-}WXmin)}{(WXMax\text{-}WXMin)} = \frac{(Xd - DXMin)}{(DXMax\text{-}DXMin)}$$

rearranging gives us

$$Xd = \frac{(Xw\text{-}WXmin) * (DXMax\text{-}DXMin)}{(WXMax\text{-}WXMin)} + DXmin$$

Normalisation

- A similar equation allows us to calculate the the Y pixel coordinate.
- The two can be combined into one matrix equation for simplicity:

Xd := Xw * A + B;

Yd := Yw * C + D;

Mapping the World Coordinates to the API

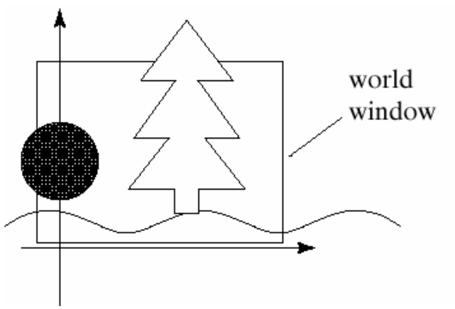
We can now implement in pseudo-code our device independent drawing primitives.

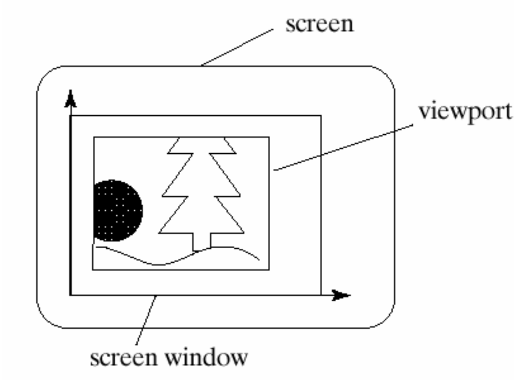
Here is an outline of an implementation in C:

```
void DrawLine(float xs, float ys, float xf, float yf)
{   /* Clip any part of the line outside the window */
    /* Normalise: Calculate the pixel coordinates */
    /* Draw the line using the API */
}
```

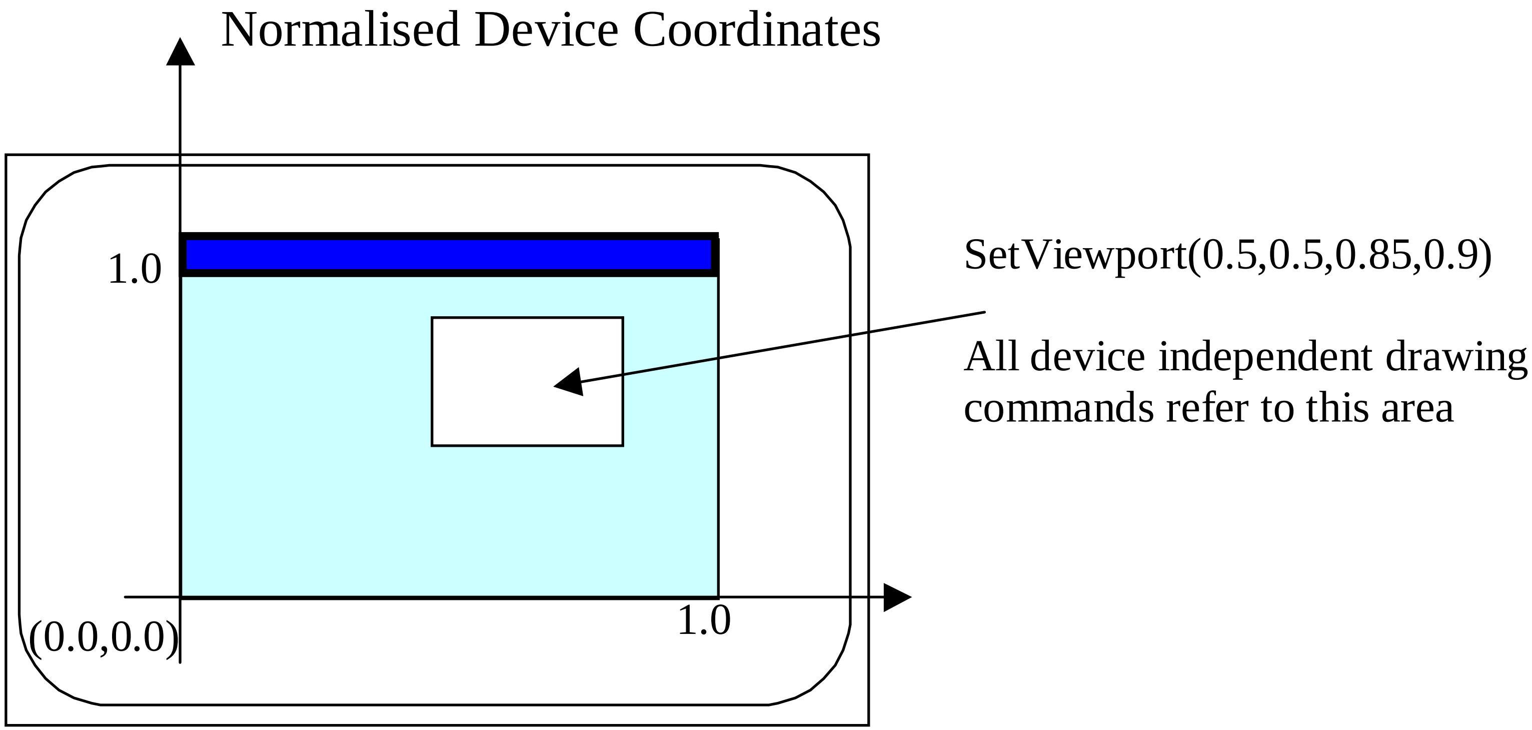
Viewports

- A Viewport is the rectangle on the raster graphics screen (or interface window for “window” displays) defining where the image will appear.
- If we select a viewport, the normal convention is that all world coordinates are mapped to the viewport rather than the whole drawing area.





- Viewports are defined in Normalised Device Coordinates where the whole drawing window has corners [0.0,0.0] and [1.0,1.0]



Normalisation with Viewports

- Using viewports simply changes our normalisation procedure.
- We now need to do the following:

1. Call the operating system API to find out the pixel addresses of the corners of the window.
2. Use the viewport setting to calculate the pixel addresses of the area where the drawing is to appear.
3. Compute the normalisation parameters A, B, C, D.