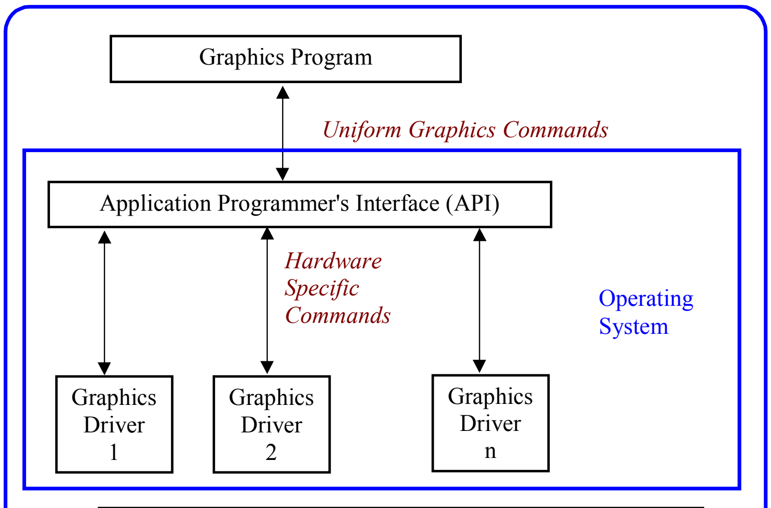
Lecture 1: Graphical Output and Input

Device Dependence

In keeping with good programming practice it is generally accepted that graphics programs should, as far as possible be device independent, that is to say we should write our programs avoiding where possible any specific hardware features. This of course is difficult to achieve in practice.



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

Diagram 1.1: The Application Programmer's Interface

Most graphics output devices are of the *Raster* type, and plot points. They include most visual display units and laser and ink jet printers. Each dot (or pixel) making up the picture is mapped directly into a random access memory, which may be accessed by the cpu either directly or through control registers. The number of bits utilised for each pixel determines the range of intensities that can be used. (1 bit means the pixel is either on or off, 8 bits allows 0..255 different intensities (or colours) to be chosen for the pixel). The number of pixels on the screen is called the resolution, and is normally quoted in terms of the x and y components [XRes,YRes]. In the case of a laser printer the resolution is quoted in dots per inch. At the lowest level, programming is device dependent. For raster devices we would use a system procedure like:

```
SetPixel(XCoord,YCoord,Colour)
```

to change the colour of a particular pixel. The coordinates used will be actual pixel addresses.

For the most part we do not need to worry about the details of each device, since the operating system takes care of that through what is called the applications programmer interface or API. This provides programmers with a set of uniform procedures to draw pictures regardless of the actual device being used as shown in diagram 1.1. However, the API does not really provide us with complete device independence. For example, if we were to write our program with all quantities specified in pixels, then it would not change if the user re-sized the window. In different systems there are different addressing conventions for pixels as illustrated in diagram 1.2. To write transportable software we need to remove this device dependence from the majority of our graphics software. For some applications, such as computer games, we may need to utilize the whole screen, in which case we need our software to cope with changes of resolution. All this means that we cannot effectively write a program using pixel addresses.

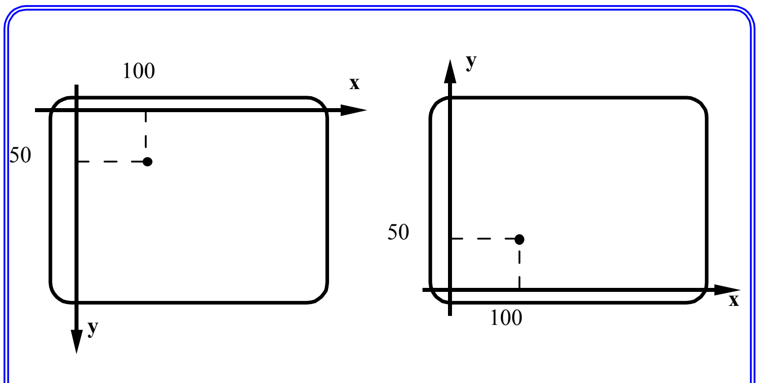


Diagram 1.2
Pixel addressing conventions vary from system to system

World Coordinate System

For real graphics applications, programmers require that anything that is drawn in a window, should be independent of the position of that window on the screen and its size. The world coordinate system provides this independence. It allows any coordinate values to be applied to the window to be drawn. It is defined through a procedure:

```
SetWindowWorldCoords(WXMin,WYMin,WXMax,WYMax)
```

We may think of a window in two ways. One is an area of the screen, in which the coordinate system is measured in pixels, the other is like a window in a room through which we view the outside world. The latter will have dimensions measured in some real world metric such as centimeters. The SetWindowWorldCoords command is simply defining the real world coordinates of the window which will be used by all the drawing procedures applied to that window. These world coordinates are chosen irrespective of how the screen window is to be moved or re-sized interactively. They will be chosen for the convenience of the applications programmer. If the task is to produce a visualisation of a house then the units could be meters. If it is to draw accurate molecular models the units will be μm. If the application program works for the most part in these units, and converts them to pixels at a late, well defined stage, then it will be easy to transport it to other systems or to upgrade it when new graphics hardware becomes available. Even using world coordinates, however, there will be a problem over the aspect ratio (Xlength/Ylength) distortion of a window. If the picture is created to fit a square window exactly, then inevitably it will be distorted if the window is resized and becomes rectangular.

Graphics Primitives

Having defined the real world coordinates of a window, the programmer can now make use of device independent procedures to draw pictures. Typical examples might be:

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

Any parts of a drawing which do not appear in the specified window are clipped so that a picture will not exceed its window.

Device independent graphics systems make extensive use of attributes (global variables) to simplify the parameter passing. In graphics a line for example can have a thickness, a style (dotted dashed or solid) and a colour. To specify all of these every time a line is drawn would be tedious, hence they are stored as attributes, and any drawing uses the current attibutes. Attributes would be manipulated by procedures such as:

```
SetLineColour(ColourNumber)
SetPolygonFillStyle(FillStyleNumber)
SetFont(FontName)
```

(Note: the names for these procedures will vary from system to system. The names given in these lectures are for illustrative purposes, and you should always read the manual carefully).

Normalisation

In order to implement a world coordinate system we need to be able to translate between world coordinates and the device or pixel coordinates. However, we do not necessarily know what the pixel coordinates of a window are, since the user can move and resize it without the program knowing. The first stage is therefore to find out what the pixel coordinates of a window are, which is done using an enquiry procedure.

```
GetWindowPixelCoords(DXmin, DYmin, DXmax, DY max)
```

In the Windows API this procedure is called GetClientRect.

Having established the user (or world) coordinate system, graphics procedures which use it must have their output data translated into the appropriate device coordinates. This is done by simple ratios, referring to Diagram 1.3 we have for the X direction:

(Xw-WXmin)/(WXMax-WXMin) = (Xd - DXMin)/(DXMax-DXMin)

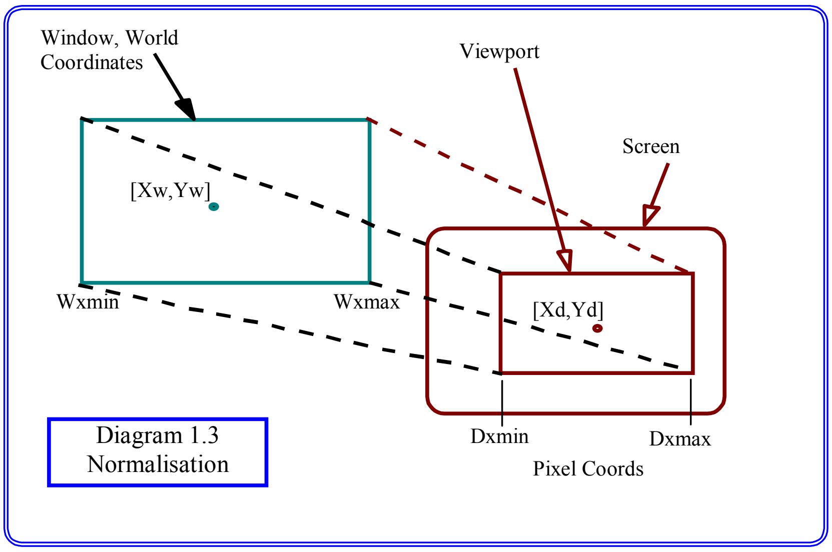
Which gives us the equations:

Xd = Xw * A + B;
Yd = Yw * C + D;

where

A = (DXmax-DXmin)/(WXmax-WXmin)
B = - WXmin (DXmax-DXmin)/(WXmax-WXmin) + DXmin

and a similar equation pair defines C and D. The normalisation is performed directly between the world coordinate system and the window pixel coordinates. Whenever a window is re-sized it is necessary to re-calculate the constants A,B,C and D.



We are now in a position to see how the world coordinate system is implemented. For example, the procedure to draw a line using world coordinates would be implemented like this.

```
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

Some graphics systems allow a further level or organisation to the applications programmer by providing viewports. These are sub areas of the window where the picture is being drawn. The normal convention is that the whole window is taken to have bottom left coordinate value (0.0,0.0) and top right coordinate (1.0,1.0). The primitive:

SetViewport(VXmin,VYmin,VXmax,VYmax)

simply defines the area where the window coordinates are to be drawn. Having obtained the values DXmax DXmin, DYmax, DYMin from the operating system, the pixel coordinates of the corners of the viewport can be simply obtained, and the normalisation transformation carried out as above.

Input Devices

The most important input device is the mouse, which records the distance moved in the X and Y directions. In the simplest form it provides at least three bytes of information (usually through a serial device); the x distance moved, the y distance moved and the button status. The mouse causes an interrupt every time it is moved, and it is up to the system software to keep track of the changes. Note that the mouse is not connected with the screen in any way. Either the operating system or the application program must achieve the connection by drawing the visible marker.

Callback

The operating system must share control of the mouse with the application, since it needs to act on mouse actions that take place outside the graphics window. For instance, processing a menu bar or launching a different application. It therefore traps all mouse events (ie changes in position or buttons) and informs the program whenever an event has taken place. The application program must, after every action carried out, return to a callback procedure (or event loop) to determine whether any mouse action (or other event such as a keystroke) has occurred. The callback is the main program part of any application, and, in simplified pseudo code, looks like this:

```
while (executing) do
{   if (menu event) ProcessMenuRequest();
    if (mouse event)
    {   GetMouseCoordinates();
        GetMouseButtons();
        PerformMouseProcess();
    }
    if (window resize event) RedrawGraphics();
}
```

The procedure *ProcessMenuRequest* will be used to launch all the normal actions, such as save and open and quit, together with all the application specific requests. The procedures *GetMouseCoordinates* and *PerformMouseProcess* will be used by the application writer to create whatever effect he wants, for example, moving an object with the mouse. This may well involve re-drawing the graphics. If the window is re-sized then the whole picture will be re-drawn.

Device Independent Input

Typically, a user will move a visible marker on the screen, and click the button when a position is chosen. This identifies a pixel, and for device independent input, its address can be inverse transformed into the users world coordinate system. Two forms of visible marker are common. The first is called a locator, and can be implemented by a cross hair or pointer. The second is called a rubber band, and is indicated by a line (or perhaps a box) from a fixed pixel to the pixel corresponding to the mouse register. In most cases the system software will provide these for a graphics application programmer. In the most usual case request mode is used meaning that the applications program calls a procedure such as:

```
RequestLocator(X,Y)
```

and then waits while the system sets up a visible marker and waits for a mouse button to be pressed, at which time it reads the mouse registers, inverse transforms them into world coordinates, and returns the values X and Y. Other input modes allow continuous sampling of devices or queuing of events by the system for processing later by the applications program.