Interactive Computer Graphics

Lecturers:

Duncan Gillies
dfg@doc.ic.ac.uk

Daniel Rueckert
dr@doc.ic.ac.uk

Webpage:
http:/www.doc.ic.ac.uk/~dfg/graphics/graphics.html

Interactive Computer graphics books

Interactive Computer Graphics – A top down approach with OpenGL, E. Angel, 2nd edition, Addison Wesley, 2000

well suited for this course, many examples in OpenGL

Computer Graphics: Principles and practice. J. D. Foley et al, 2nd edition, Addison Wesley, 1996

very comprehensive – "The Bible of Graphics"

Coursework : Shading and Texture Mapping Viewing transformations

NB We may change the exercise this year

Key elements of a graphics system

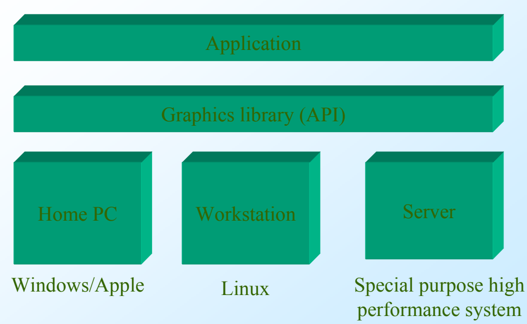
1. Processor
2. Memory
3. Framebuffer
4. Output devices:
   - monitor (CRT LCD)
   - printer
5. Input Devices:
   - keyboard, mouse, joystick, spaceball
   - data glove, eye tracker

Interactive Computer Graphics: APIs

Graphics output devices are many and diverse, but fortunately we don't need to worry too much about them since the operating system will generally take care of many of the details.

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

Interactive Computer Graphics: APIs



Interactive computer graphics: APIs

Problem: If speed is critical (i.e. computer games) it may be tempting to avoid using the API and access the graphics hardware directly.

➨ Device dependence

Existing APIs:

1. OpenGL
2. Direct3D
3. Java3D
4. VRML
5. Win32 API

Interactive Computer Graphics: OpenGL

OpenGL is hardware independent:
- PCs
- Workstations
- Supercomputers

OpenGL is operating system independent:
- Windows NT, Windows 2000, Windows XP
- Linux and Unix

OpenGL can perform rendering in
- software (i.e. processor)
- hardware (i.e. accelerated graphics card) if available

Interactive Computer Graphics: OpenGL

OpenGL can be used from
- C, C++
- Ada, Fortran
- Java

OpenGL supports
- polygon rendering
- texture mapping and anti-aliasing

OpenGL doesn't support
- ray tracing
- volume rendering

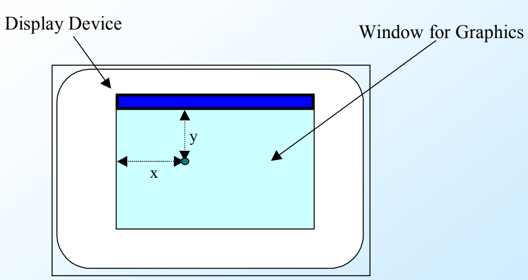
Raster Graphics

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

A typical (API) command might be:

SetPixel(x,y,colour)

Where x and y are pixel coordinates.



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

Bits per pixel

In some cases (laser printers) only one bit is used to represent each pixel allowing it to be on or off (black dot or white dot).

In old systems 8 bits are provided per pixel allowing 256 different shades to be represented.

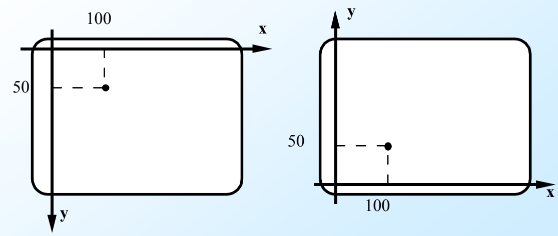
Most common today are pixels with 24 or 32 bit representation, allowing representation of millions of colours.

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.

Different pixel addressing conventions



Device Dependent Drawing Primitives

Each operating system provides us with the possibility of drawing graphics at the pixel level.

For example in the Windows 32 API we have:

MoveToEx(hdc xpix, ypix);
LineTo(hdc, xpix1, ypix1);
TextOut(hdc, xpix2, ypix2, message, length);

Where hdc is an identifier for the window, and xpix and ypix are 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. We want to be able to move graphics applications between different systems (PC, Workstation, Supercomputer 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:

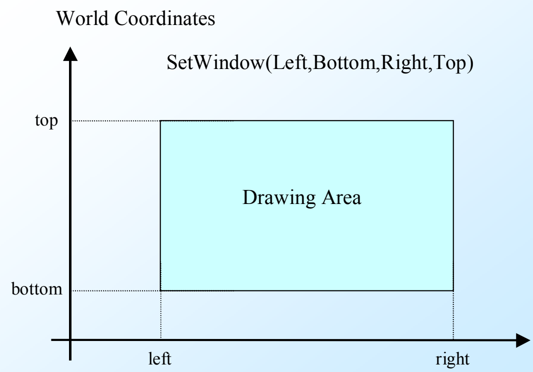
meters
light years
microns
etc

Worlds and Windows

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

SetWindow(left,bottom,right,top)

We can think of this as a window onto the world matching a window on the screen



Device independent Graphics Primitives

Having defined our world coordinate system we can implement drawing primitives to use with it. For example:

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

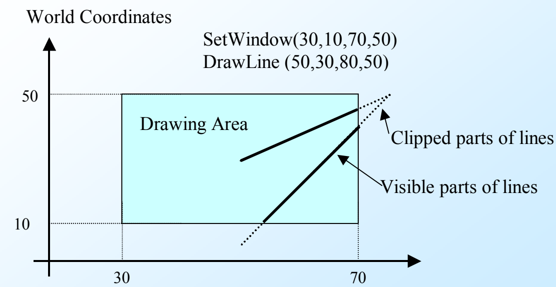
Normally any part of a graphics object outside the window is clipped.

Problem Break

What would you expect to be drawn in a graphics window by the following instructions:

SetWindow(30,10,70,50);
DrawLine(50,30,80,50);
DrawLine(80,50,50,5);

Solution



Attributes

In device independent graphics primitives we usually avoid having a comprehensive set of parameters. For example, a line will have:

Style (solid or dotted)
Thickness (points)
Colour

And text will have

Font
Size
Colour

These are called attributes

Normalisation

We need to connect our device independent graphics primitives to the device dependent drawing commands so that we can see something on the screen.

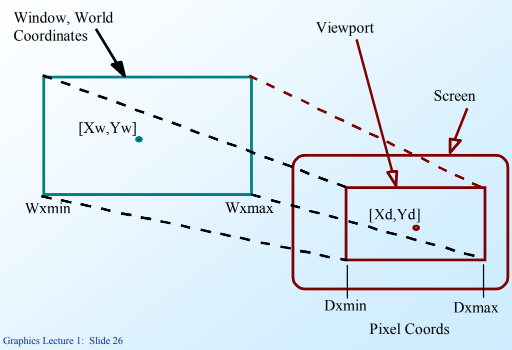
This is done by the process of normalisation.

Normalisation

In normalisation we need to translate our world coordinates into a set of coordinates that will be suitable for drawing using the API.

First we must call the API to find out from the operating system the pixel addresses of the corners of the area we are using

Normalisation



Normalisation

Having defined our world coordinates, and obtained our device coordinates we relate the two by simple ratios:

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

rearranging gives us

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

Normalisation

A similar equation allows us to calculate the Y pixel coordinate. The two can be combined into a simple pair of linear equations:

Xd := Xw * A + B;

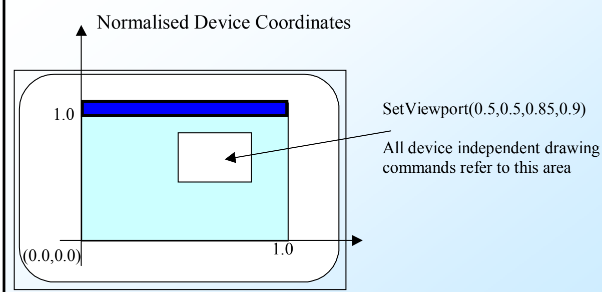
Yd := Yw * C + D;

Viewports

Viewports are smaller parts of the window where the drawing is being displayed.

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

Input Devices

There are many input devices for computer graphics:

Mouse
Joystick
Button Box
Digitising Tablet
Light Pen
etc.

We will only consider the mouse here.

Mouse Position and Visible Markers

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

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.

Mouse Events

A mouse event occurs when something changes, ie it is moved or a button is pressed.

The mouse interrupts the operating system to tell it that an event has occurred and sends it the new data.

The operating system normally updates the position of the marker on the screen.

Callback procedure

The operating system informs the application program of mouse events (and other events) which are relevant to it.

The program must receive this information in what is called a callback procedure (or event loop).

Simple Callback procedure

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

Input Methods

The mouse is commonly used to implement input methods:

Locator:

Identifies a point on the screen using a visible marker

Rubber Band:

Adjusts the size and position of a graphical object