Chapter 1 Handout – Introduction to Computer Graphics

Introduction

The aim of computer graphics is to produce realistic and/or useful images on a computer. For some applications the emphasis will be on realism (e.g. special effects in films), whereas for others it will simply be on usefulness (e.g. data visualisation). We will discuss some different applications of computer graphics in Section 4.

Computer graphics provides methods to generate images using a computer. The word "image" should be understood in a more abstract sense here. An image can represent a realistic scene from the real world, but graphics like histograms (statics diagram) or pie charts as well as the graphical user interface of a software tool are also considered as images. The development of **computer graphics** has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized animation, movies and the video game industry.

The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound". Typically, the term *computer graphics* refers to several different things:

- the representation and manipulation of image data by a computer
- the various technologies used to create and manipulate images
- the images so produced, and
- the sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content

Today, computers and computer-generated images touch many aspects of daily life. Computer imagery is found on television, in newspapers, for example in weather reports, or for example in all kinds of medical investigation and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to understand and interpret. In the media "such graphs are used to illustrate papers, reports, thesis", and other presentation material.

Many powerful tools have been developed to visualize data. Computer generated imagery can be categorized into several different types: 2D, 3D, 4D, 7D, and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still widely used. Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Over the past decade, other specialized fields have been developed like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component".

Whatever the application, the final image will be generated from some *model* of the scene we want to take a picture of. For example, in Figure 1 we have a representation of a 3-D tetrahedron (the *object model*), together with some model of how the scene is lit (the *light sources*), and we want to produce an image on the *image plane* based on the position of the viewer. The object model will typically represent the geometry of the object together with some information about what type of material it is made of (i.e. what it looks like). Normally we consider the viewer to be a *virtual camera*, which defines how the picture is taken. This process of starting with a model (often a 3-D model) of the world and taking a picture of it with a virtual camera is known in computer graphics as *rendering*. The rendering process can include many stages, such as hidden surface elimination, surface rendering and clipping, which we will deal with throughout this course.

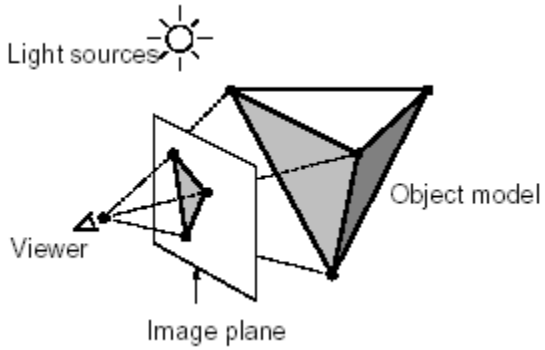


Figure 1 - Rendering a 2-D Image from a 3-D Model

Some graphics applications can be described as *interactive graphics applications*. In these cases, images need to be generated in real-time and as well as viewing the images the user can interact with the model of the world using specialised input hardware. For example, a mouse, keyboard, tablet and stylus, scanner or a virtual reality headset and gloves can all be used as interactive graphics input devices.

1. Initial Development of Computer Graphics

The advance in computer graphics was to come from one MIT (Massachusetts Institute of Technology: an engineering university in Cambridge) student, Ivan Sutherland. In 1961 Sutherland created another computer drawing program called Sketchpad. Using a light pen, Sketchpad allowed one to draw simple shapes on the computer screen, save them and even recall them later. The light pen itself had a small photoelectric cell in its tip. This cell emitted an electronic pulse whenever it was placed in front of a computer screen and the screen's electron gun fired directly at it. By simply timing the electronic pulse with the current location of the electron gun, it was easy to pinpoint exactly where the pen was on the screen at any given

moment. Once that was determined, the computer could then draw a cursor at that location. Even today, many standards of computer graphics interfaces got their start with this early Sketchpad program. One example of this is in drawing constraints. If one wants to draw a square for example, s/he doesn't have to worry about drawing four lines perfectly to form the edges of the box. One can simply specify that s/he wants to draw a box, and then specify the location and size of the box. The software will then construct a perfect box, with the right dimensions and at the right location.

2. Applications of Computer Graphics

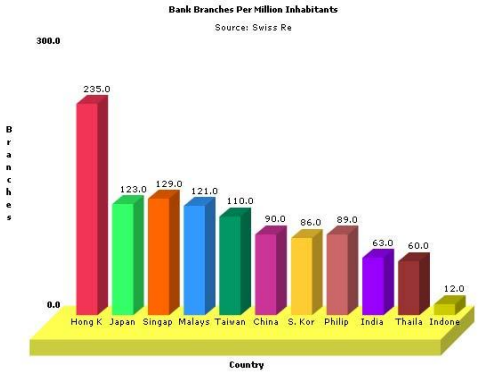
Before we look in any more detail at *how* computers can generate graphical images, let us consider *why* we would want to do this. Since the early days of computing, the field of computer graphics has become a very popular one because of its wide range of applications. The following sections summarise the main categories of application.

2.1. Graphs and Charts

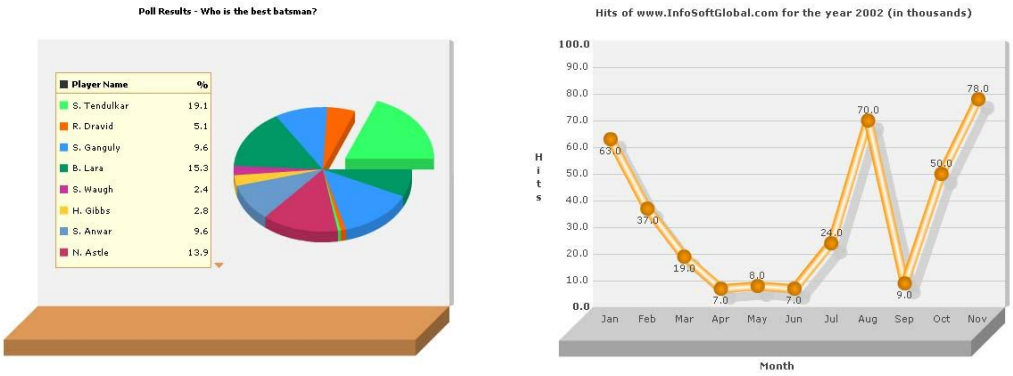
Graphs and charts have long been used for visualising data. They are particularly widely used for visualising relationships and trends in scientific, mathematical, financial and economic data.

Although, in principle, we do not need advanced graphics algorithms to display graphs and charts, many modern visualisation packages use 3-D effects such as shadowing to make the graph/chart more visually appealing. For example, Figure 2 shows a range of graphs and charts produced by the free *FusionCharts Lite* package (http://www.infosoftglobal.com/FusionCharts/Lite). Commercial packages are also available and include Microsoft Excel.

(a)



(b)



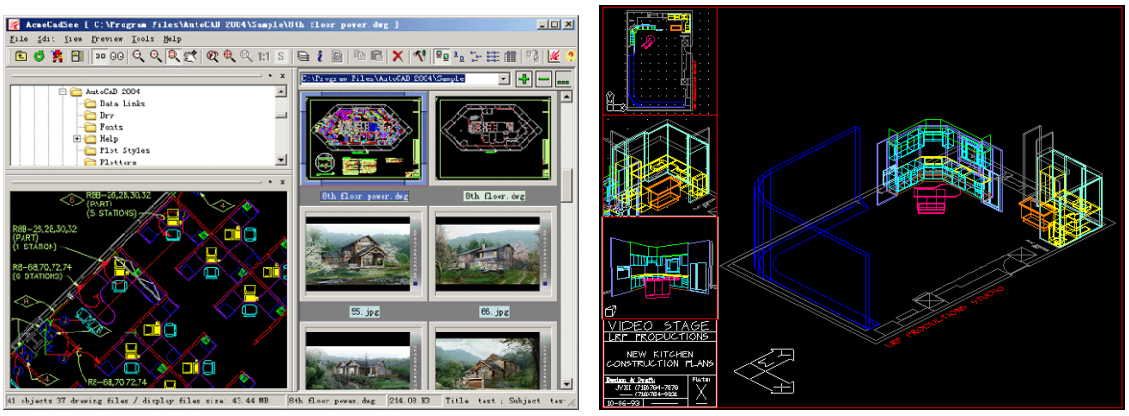
(c) **(d)**

Figure 2 - Computer Graphics used for Graphs and Charts

2.2. Computer-Aided Design

Computer-Aided Design (CAD) and Computer-Aided Drafting and Design (CADD) involve using a computer application to enable designers to construct and visualise 3-D models. They are commonly used in fields such as architecture, engineering and circuit design. For example, Figure 3 shows two screenshots from CAD applications being used for architecture. One of the most common CAD applications is AutoCAD.

Computer-Aided Manufacturing (CAM) is a similar concept to CAD, except that the design application is linked to the manufacturing process, i.e. the application will directly control, via a hardware communication link, the machine that manufactures the object being designed.



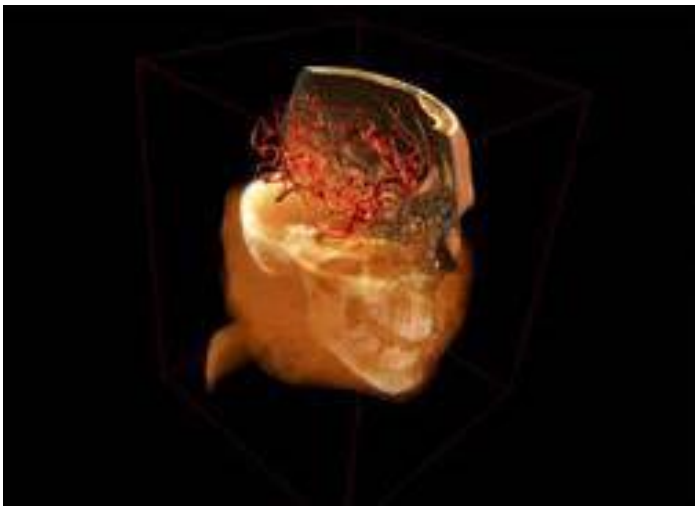
(a) **(b)**

Figure 3 - Computer Graphics used for Computer-Aided Design (CAD)

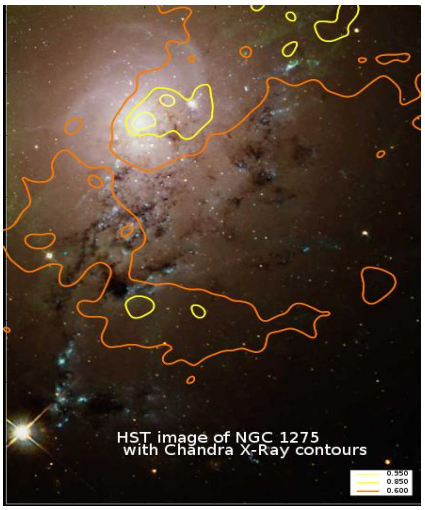
2.3. Data Visualisation

We already saw in Section 2.1 how graphics can be used for visualising data in the form of graphs and charts. More complex, often multidimensional, datasets can also be visualised using graphics techniques. In this case, it is often difficult to visualise such datasets without computer graphics. These complex datasets are particularly common in science (*scientific visualisation*) and business (*business visualisation*).

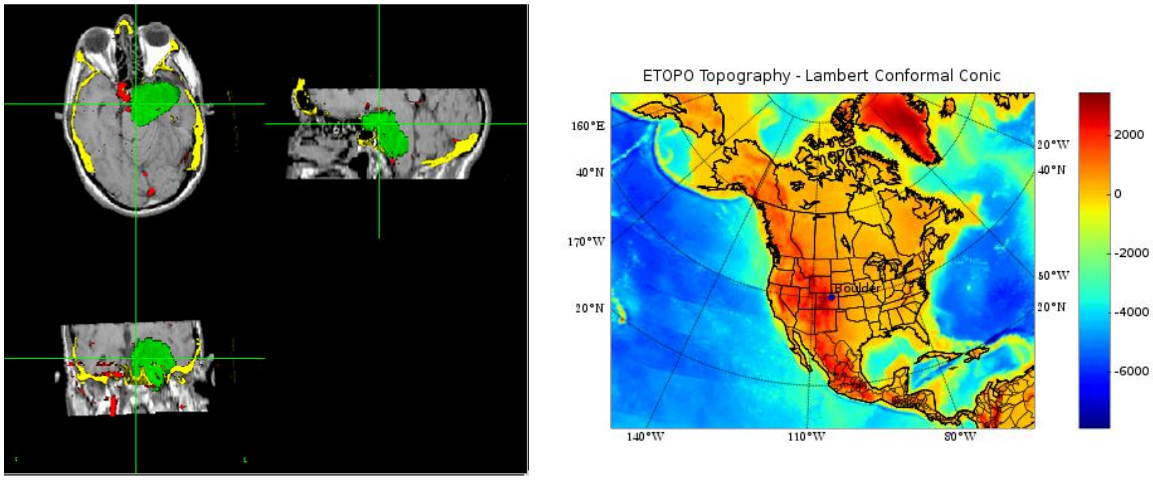
For example, Figure 4(a) shows a three-dimensional medical dataset of a patient. Here we use computer graphics to visualise the paths of blood vessels inside the patient's head. Figure 4(b) shows an image generated from data acquired by the Hubble Space Telescope. This image of a distant galaxy is allowing us to visualise a four-dimensional dataset – three dimensions are combined as the red, green and blue components of an optical image, and the fourth (an X-ray image) is displayed as overlaid contour plots. Figure 4(c) shows another medical dataset, this time acquired using a Magnetic Resonance Imaging (MRI) scanner. Computer graphics allow us to visualise this three-dimensional dataset as three *slices* through the 3-D volume. Finally, Figure 4(d) shows the use of colour to visualise altitude in a map of the United States.



(a)



(b)



(c) **(d)**

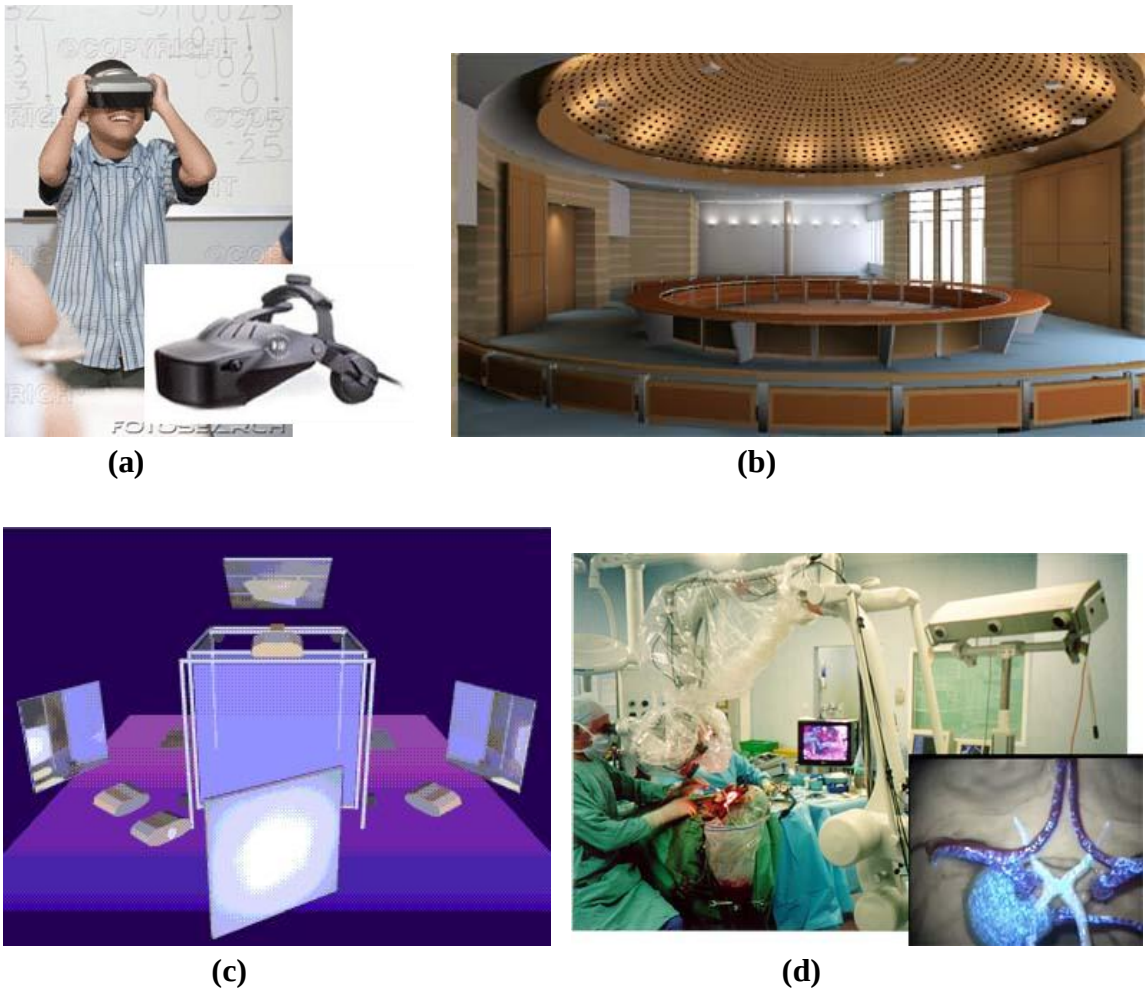
Figure 4 - Computer Graphics used for Visualisation of Complex Datasets

2.4. Virtual Reality and Augmented Reality

Virtual Reality (VR) allows users to be *immersed* in a computer generated world, and to interact with it as if they were interacting with the real world. Typically, the user will wear special hardware such as the VR-headset shown in Figure 5(a). This allows the computer to completely control the visual information received by the user, so if realistic computer-generated images such as that shown in Figure 5(b) are displayed in the headset in stereo (i.e. different images for the left and right eyes) the user will have a sense of physical *immersion* in the virtual world. In addition, other specialised hardware such as VR-gloves can be used to interact with the 'objects' in the virtual world.

An alternative way of experiencing virtual reality is by using a CAVE (which stands for Cave Automatic Virtual Environment). An example of this is shown in Figure 5(c). Here, the user stands inside a cubicle which has images projected onto the walls and ceiling. Often there will also be a means of 'moving' in the virtual world such as pressure pads.

Augmented Reality (AR) combines a computer-generated virtual world with the real world. In AR, computer-generated images are overlaid onto a user's view of the real world. For example, Figure 5(d) shows an AR system used for surgery, in which computer-generated images of hidden features such as blood vessels and tumours are overlaid on the surgeon's view of the patient through a surgical microscope.



(a) **(b)**

(c) **(d)**

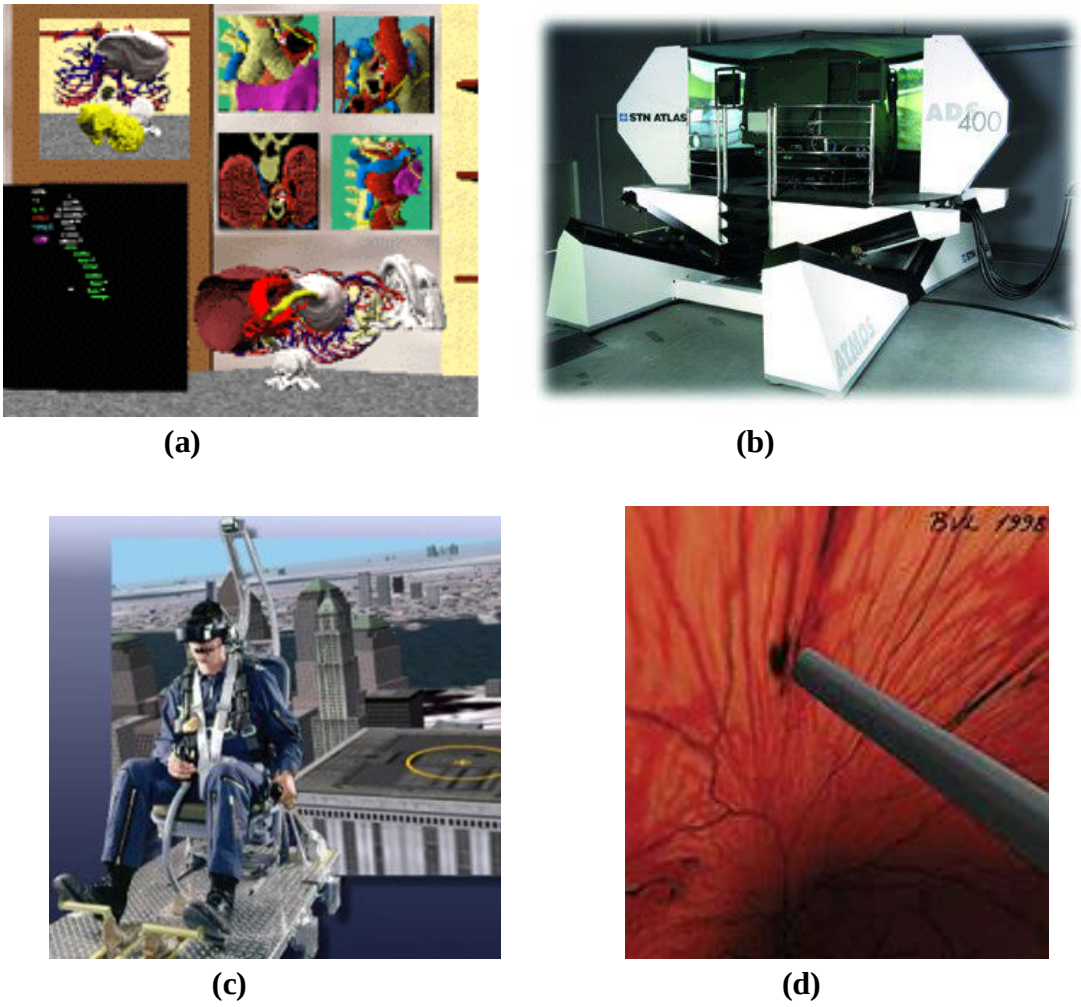
Figure 5 - Computer Graphics used for Virtual Reality and Augmented Reality

2.5. Education and Training

Computer graphics can be very beneficial in education and training. For example, Figure 6(a) shows a screenshot from some educational software that teaches students about human anatomy. Graphics are used to help students visualise the appearance and location of different organs inside the body.

Another common application of computer graphics is in training. In many jobs it is difficult for workers to get direct experience of the environment that they are expected to work in. This may be because the environment is inaccessible (e.g. space), dangerous (e.g. bomb disposal), expensive (e.g. flying aircraft) or high-risk (e.g. surgery). Computer graphics, combined with specialised hardware, can be used to simulate the working environment, so that workers can gain the skills required for their work. For example, Figure 6(b) shows a picture of a truck-driving simulator used by the German army to train their drivers in emergency situations; Figure 6(c)

shows a helicopter simulator; and Figure 6(d) shows a screenshot from a simulator for laser eye surgery.



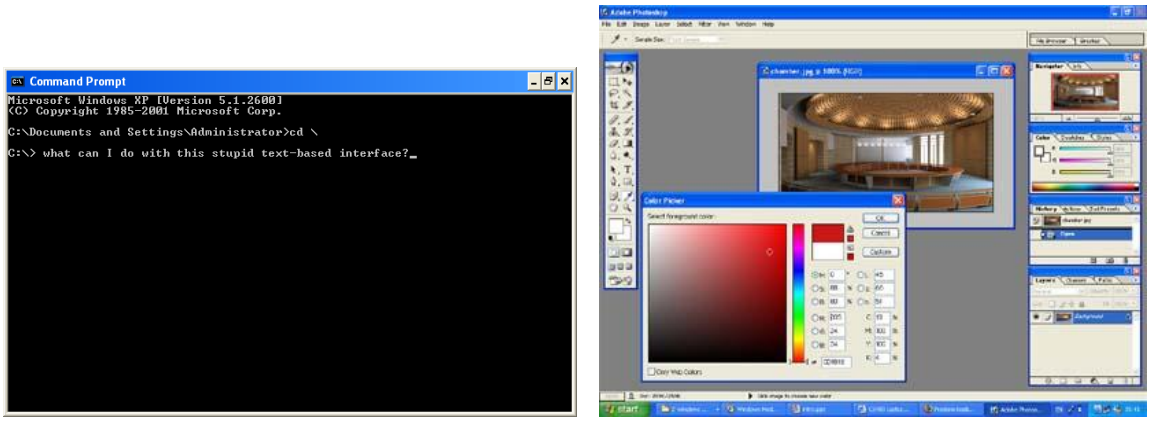
(a) **(b)**

(c) **(d)**

Figure 6 - Computer Graphics used for Education and Training

2.6. Graphical User Interfaces

Another possibility that has been opened up by the improvements in computer graphics technology is in the area of user interfaces. Figure 7 shows the difference in visual appeal between an old text-based user interface and a modern windows-based user interface. And the difference is not just in visual appeal: modern user interfaces have made *multitasking* (doing several things at once) much easier and applications such as word-processors have become much more powerful.



(a) **(b)**

Figure 7 - Text-Based and Graphical User Interfaces

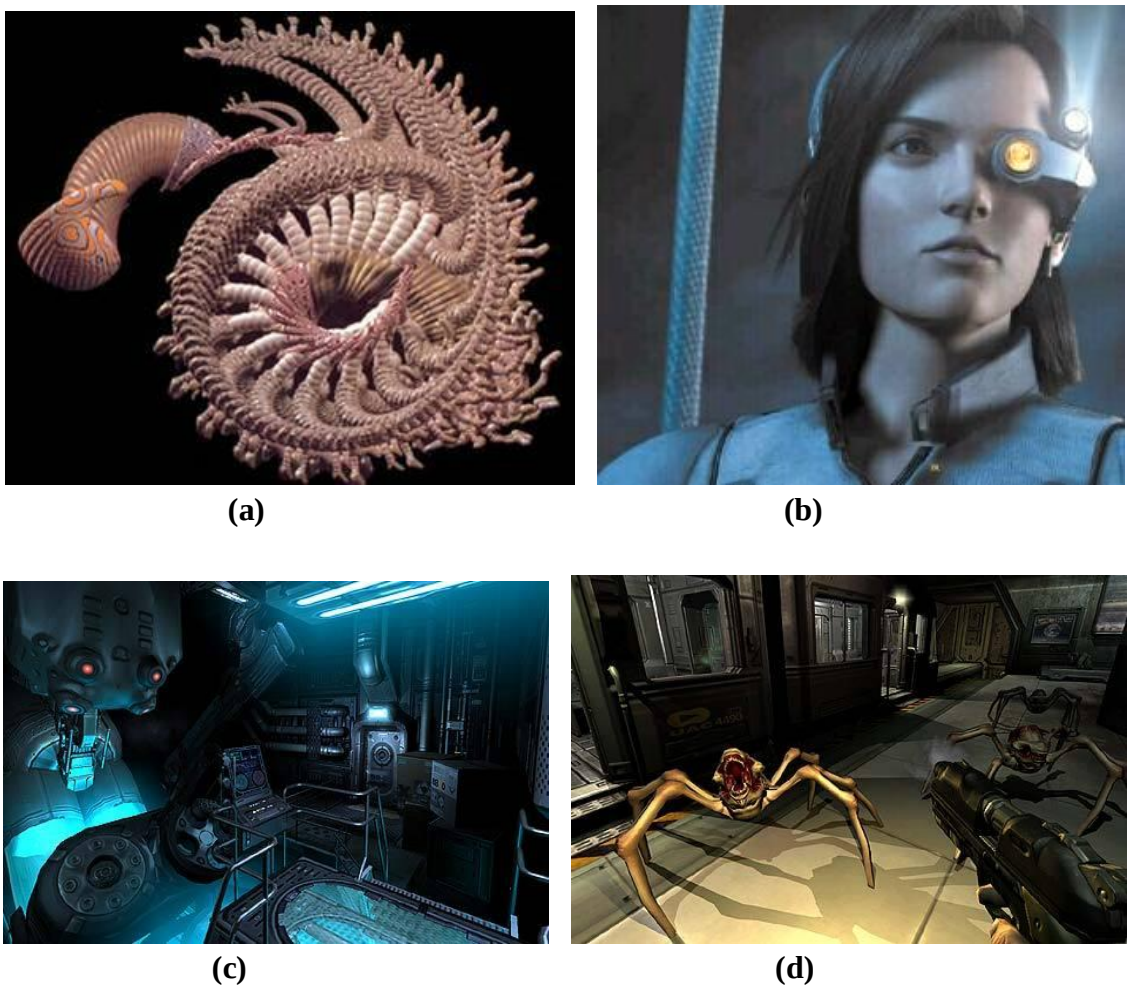
2.7. Entertainment

The applications of computer graphics in entertainment fall into three categories: computer art, special effects and animations, and games.

A number of artists have taken advantage of the possibilities offered by computer graphics in producing works of art. For example, Figure 8(a) shows a frame from an animated artwork by William Latham.

Computer-generated images (CGI) have been widely used in the entertainment industry for producing special effects for films, and also for producing completely computer-generated films. Figure 8(b) shows a frame from the film *Final Fantasy: the Spirits Within*, which featured 100% computer-generated images. For applications such as this, highly realistic images can be produced *off-line*, i.e. we do not need to generate the images in real-time, they need only be generated once, frame-by-frame, and then combined into the desired animated sequence.

For computer games, images must be generated in real-time in response to user actions. Therefore the graphics in games can be less realistic, since the emphasis is on speed of generation rather than realism. Figure 8(c) and (d) show screenshots from the popular computer game *Doom*.



(a) **(b)**

(c) **(d)**

Figure 8 - Computer Graphics used in the Entertainment Industry

3. Graphics Software

To generate graphical images like those we have seen above, we need some type of graphics software. We can divide graphics software into two categories:

- *Special-purpose* graphics packages
- *General-purpose* graphics packages

Special-purpose packages are designed for a specific purpose. For example a drawing package such as the *Paint* accessory in Microsoft Windows is an example of a special-purpose graphics package: it is designed to allow users to draw simple pictures consisting of lines, curves and other basic components. Another example is CAD packages such as AutoCAD: these are designed to allow users to build and visualise 3-D models. Special-purpose graphics packages tend to be used mostly by *end-users*, not programmers or computer professionals. In other words,

you do not need to know about computer graphics theory to use them. Other examples of special-purpose packages are the drawing applications *Adobe Photoshop*, *Xara* and *GIMP*; the 3-D modelling applications *3-D Studio Max* and *Design Works*; and the visualisation software *FusionCharts* and *Microsoft Excel*.

General-purpose packages are a pre-defined library of routines for drawing graphics primitives. They are used by computer programmers to add graphical capability to their programs. They provide a lot more flexibility than special-purpose packages but they require technical knowledge of computer graphics and programming and so are not suitable for end-users. Examples of general-purpose graphics packages are the standard low-level graphics library *OpenGL*, the Microsoft equivalent *DirectX* and the higher-level library *OpenInventor*. We can also think of *scene description languages* as a type of general-purpose graphics package. Scene description languages are languages that can be used to define 3-D models of scenes. Special viewing software is required to view and often interact with the scene. The most common example of a scene description language is *VRML*, the Virtual Reality Modelling Language.

4. Graphics Display Devices

Any computer-generated image must be displayed in some form. The most common graphics display device is the video monitor, and the most common technology for video monitors is the Cathode Ray Tube (CRT). Now we will examine the technology behind CRT displays, and how they can be used to display images.

4.1. CRT Displays

CRT display devices are very common in everyday life as well as in computer graphics: most television sets use CRT technology. Figure 9 illustrates the basic operation of a CRT display. Beams of electrons are generated by electron guns and fired at a screen consisting of thousands of tiny phosphor dots. When a beam hits a phosphor dot it emits light with brightness proportional to the strength of the beam. Therefore pictures can be drawn on the display by directing the electron beam to particular parts of the screen. The beam is directed to different parts of the screen by passing it through a magnetic field. The strength and direction of this field, generated by the *deflection yoke*, determines the degree of deflection of the beam on its way to the screen.

To achieve a colour display, CRT devices have three electron beams, and on the screen the phosphor dots are in groups of three, which give off red, green and blue light respectively. Because the three dots are very close together the light given off by the phosphor dots is combined, and the relative brightnesses of the red, green and blue components determines the colour of light perceived at that point in the display.

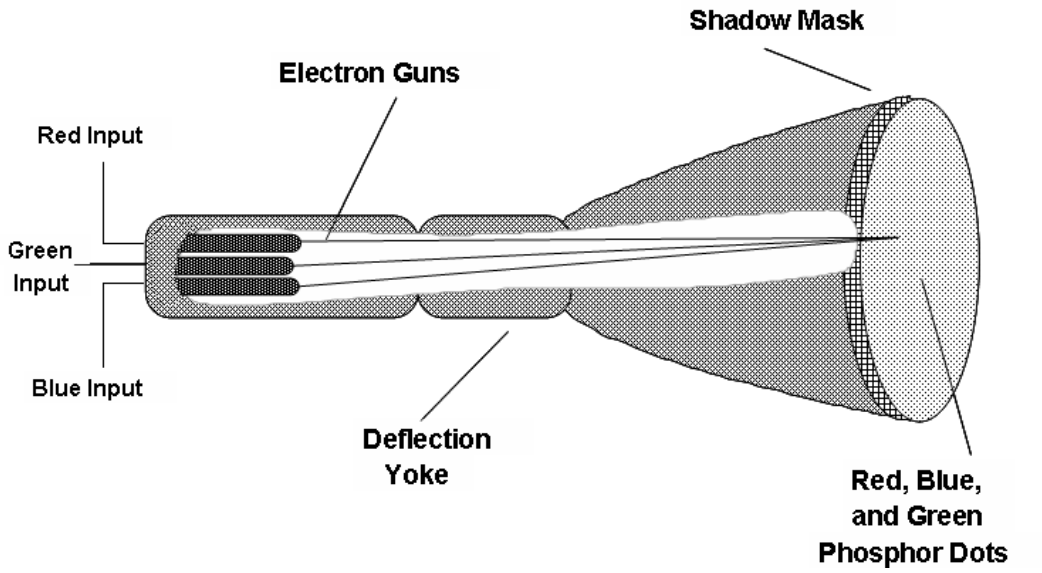


Figure 9 - A Cathode Ray Tube Display Device

4.2. Random Scan Devices

CRT displays can work in one of two ways: *random scan* devices or *raster scan* devices. In a random scan device the CRT beam is only directed to areas of the screen where parts of the picture are to be drawn. If a part of the screen is blank the electron beam will never be directed at it. In this case, we draw a picture as a set of *primitives*, for example lines or curves (see Figure 10). For this reason, random scan devices are also known as *vector graphics displays*.

Random scan displays are not so common for CRT devices, although some early video games did use random scan technology. These days random scan is only really used by some hard-copy plotters.

4.3. Raster Scan Devices

In a raster scan device the primitives to be drawn are first converted into a grid of dots (see Figure 10). The brightness's of these dots are stored in a data structure known as a *frame buffer*. The CRT electron beam then sweeps across the screen line-by-line, visiting every location on the screen, but it is only switched on when the frame buffer indicates that a dot is to be displayed at that location.

Raster scan CRT devices are the most common type of graphics display device in use today, although recently LCD (Liquid Crystal Display) devices have been growing in popularity. In this course, though, we will concentrate on the details of raster scan devices.

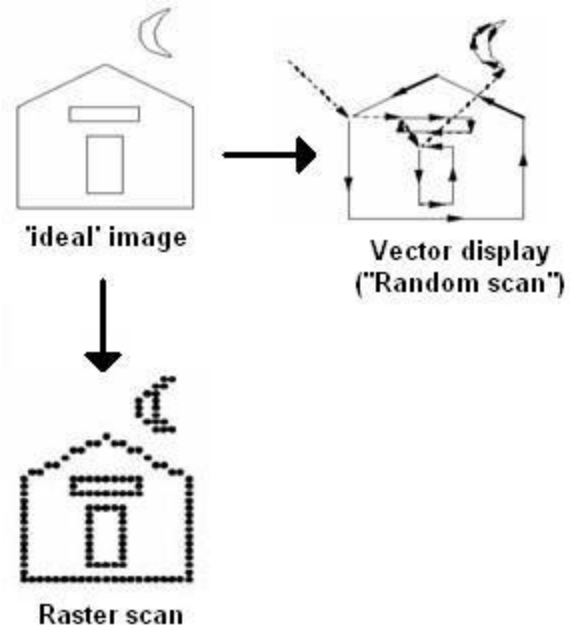


Figure 10 - Random Scan and Raster Scan CRT Displays

Before we proceed, we must introduce some important terminology used when discussing raster scan devices (see Figure 11). Whilst the frame buffer stores the image data ready for display, the actual display itself is known as the *raster*. The raster is a grid of phosphor dots on the display screen. Each of these dots is known as a *picture cell*, or *pixel* for short. Each row of pixels in the raster is known as a *scan-line*. The number of pixels in a scan-line is known as the *x-resolution* of the raster, and the number of scan-lines is known as the *y-resolution*. Finally, the ratio of the *y*-resolution to the *x*-resolution (or sometimes the other way around) is known as the *aspect ratio* of the display.

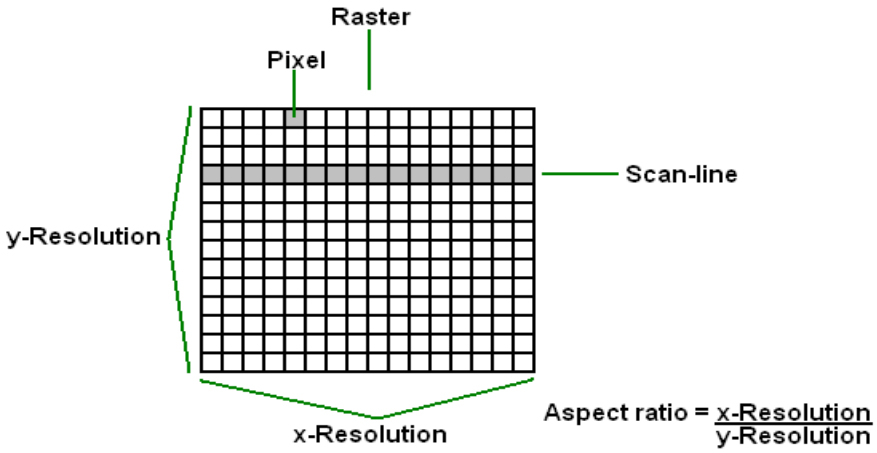


Figure 11 - Raster Scan Terminology

We mentioned in Section 4.1 that for colour CRT displays there are three electron beams: one for red light, one for green light and one for blue light. To ensure that these three beams precisely hit the appropriate phosphor dots on the display, after the beams have been deflected by the deflection yoke they pass through a mask containing many tiny holes – one hole for each pixel in the raster (see Figure 9 and Figure 12). This mask is known as the *shadow mask*. Because the three electron beams originated from slightly different locations (from three separate electron guns), if they all pass through the same hole in the shadow mask they will impact the screen at slightly different locations: the locations of the red, green and blue phosphor dots.

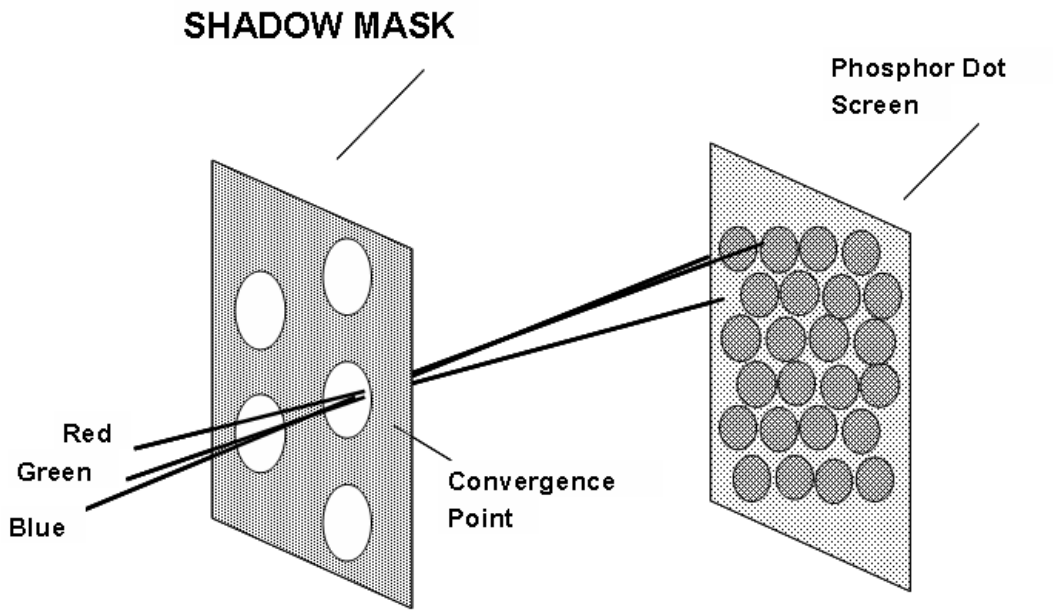


Figure 12 - The Shadow Mask in a Raster Scan CRT Display

The phosphor dots on a CRT display device only emit light for a very brief period of time after the electron beam has moved on (the length of time that the phosphor emits light is known as its *persistence*). Therefore to give the impression of a permanent image on the screen the raster must be continually updated. Raster scan systems perform this continual update by 'sweeping' the electron beams across the raster scan-line by scan-line, starting from the top and working towards the bottom. When the last scan-line is completed we start again from the top.

The number of times that the entire raster is *refreshed* (i.e. drawn) each second is known as the *refresh rate* of the device. For the display to appear persistent and not to flicker the display must update often enough so that we cannot perceive a gap between frames. In other words, we must refresh the raster when the persistence of the phosphor dots is beginning to wear off. In practise, if the refresh rate is more than 24 frames per second (*f/s*) the display will appear reasonably smooth and persistent.

Modern graphics displays have high refresh rates, typically in the region of 60 f/s. However, early graphics systems tended to have refresh rates of about 30 f/s. Consequently, some flicker was noticeable. To overcome this, a technique known as *interlaced scanning* was employed. Using interlaced scanning alternate scan-lines are updated in each raster refresh. For example, in the first refresh only odd numbered scan-lines may be refreshed, then on the next refresh only even-numbered scan-lines, and so on (see Figure 13). Because this technique effectively doubles the screen refresh rate, it has the effect of reducing flicker for displays with low refresh rates. Interlaced scanning was common in the early days of computer graphics, but these days displays have better refresh rates so it is not so common.

The following are the specifications of some common video formats that have been (and still are) used in computer graphics:

- *VGA*: resolution 640x480, 60 f/s refresh rate, non-interlaced scanning.
- *PAL*: resolution 625x480, 25 f/s refresh rate, interlaced scanning
- *NTSC*: resolution 525x480, 30 f/s refresh rate, interlaced scanning

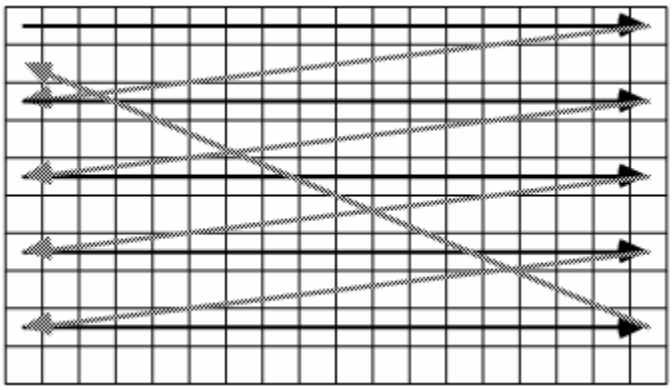


Figure 13 - Interlaced Scanning for Raster Scan Displays

4.3.1. Frame Buffers

In Section 4.3 we introduced the concept of a frame buffer. Frame buffers are used by raster scan display devices to store the pixel values of the image that will be displayed on the raster. It is a 2-D array of data values, with each data value corresponding to a pixel in the image. The number of bits used to store the value for each pixel is known as the *bit-planes* or *depth* of the frame buffer. For example, a 640x480x8 frame buffer has a resolution of 640x480 and a *depth* of 8 bits; a 1280x1024x24 frame buffer has a resolution of 1280x1024 and a *depth* of 24 bits. For colour displays we need to store a value for each component of the colour (red, green and blue), so the bit-planes will typically be a multiple of 3 (e.g. 8 bit-planes each for red, green and blue makes a total of 24 bit-planes)

4.3.2. Architecture of Raster Graphics Systems

Now we are in a position to examine the basic architecture of raster graphics systems, i.e. what components are required, and how do they interact? Figure 14 shows a block-diagram of a typical raster graphics system. Most (non-graphics) processing will occur in the CPU of the computer, which uses the system bus to communicate with the system memory and peripheral devices. When graphics routines are to be executed, instead of being executed by the CPU they are passed straight to the *display processor*, which contains dedicated hardware for drawing graphics primitives. The display processor writes the image data into the frame buffer, which is a reserved portion of the display processor memory. Finally the video controller reads the data in the frame buffer and uses it to control the electron beams in the CRT display.

The display processor is also known by a variety of other names: *graphics controller*, *display coprocessor*, *graphics accelerator* and *video card* are all terms used to refer to the display processor. Since the display processor contains dedicated hardware for executing graphics routines it must be dedicated to a particular set of routines. In other words, display processors will only be able to handle the graphics processing if the routines used are from a particular graphics package. This is known as *hardware rendering*. Most commercial video cards will support hardware rendering for the OpenGL graphics package, and many PC video cards will also support hardware rendering for DirectX. Hardware rendering is much faster than the alternative, *software rendering*, in which graphics routines are compiled and executed by the CPU just like any other code. For the raster graphics architecture to support software rendering the block-diagram shown in Figure 14 would need to be modified so that the frame buffer was connected directly to the system bus in order that it could be updated by the CPU.

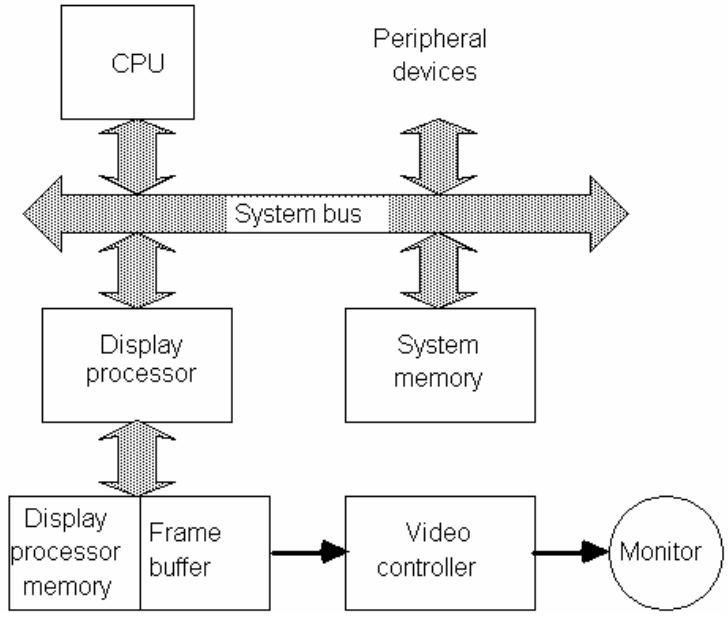


Figure 14 - The Architecture of a Raster Graphics System with a Display Processor

4.4. 3-D Display Devices

In recent years the popularity of 3-D graphics display devices has been growing, although currently they are still quite expensive compared with traditional 2-D displays. The aim of 3-D display devices is to provide a *stereo pair* of images, one to each eye of the viewer, so that the viewer can perceive the *depth* of objects in the scene as well as their position. The process of generating such 3-D displays is known as *stereoscopy*.

3-D displays can be divided into two types: head-mounted displays (HMDs) and head-tracked displays (HTDs). HMDs are displays that are mounted on the head of the viewer. For example, Figure 15 shows a HMD – the device fits on the head of the user and displays separate images to the left and right eyes, producing a sense of stereo *immersion*. Such devices are common in virtual reality applications. Normally a tracking device will be used to track the location of the display device so that the images presented to the user can be updated accordingly – giving the impression that the viewer is able to 'move' through the virtual world.

Whereas with a HMD the display moves with the viewers head, with a HTD the display remains stationary, but the head of the viewer is tracked so that the images presented in the display can be updated. The difficulty with HTDs is how to ensure that the left and right eyes of the viewer receive separate stereo images to give a 3-D depth effect. A number of technologies exist to achieve this aim. For example, the display can use polarised light filters to give off alternate vertically and horizontally polarised images; the viewer then wears special glasses with polarised filters to ensure that, for example, the left eye receives the vertically polarised images and the

right eye receives the horizontally polarised images. An alternative technique is to use colour filters: the display draws a left-eye image in, for example, blue, and a right eye image in green; then the viewer wears glasses with colour filters to achieve the stereo effect. Other technologies also exist.

Figure 15 - A Virtual Reality Headset

5. OpenGL

Now we will introduce the OpenGL graphics package. This part of the course is aimed to give you practical experience to reinforce the theory behind computer graphics. Code examples will be given in C++, and you are expected to experiment with the OpenGL routines that are introduced throughout the course.

In this course we will be using OpenGL together with the GL Utilities Toolkit (*glut*). You should make sure that you have access to a C++ development environment that supports these two libraries. *Note that Turbo C++ does not support either of them*. Two possibilities are:

- The free *Dev-C++* environment (www.bloodshed.net) has OpenGL built-in and *glut* can be easily added on as a separate package.
- Microsoft Visual C++ also has OpenGL built-in but not *glut*. The *glut* library is available as a free download from http://www.xmission.com/~nate/glut.html. Installation is fairly straightforward.

The examples and exercise solutions given in this course were coded with Dev-C++, but the same source code should compile with Visual C++. Also, if you use Dev-C++, it comes with a very useful OpenGL help file that you can use as a reference for different OpenGL routines (it will be found in the *Docs/OpenGL* folder inside the Dev-C++ installation folder).

5.1. Introduction

OpenGL is based on the GL graphics package developed by the graphics hardware manufacturer *Silicon Graphics*. GL was a popular package but it was specific to Silicon Graphics systems, i.e. code written using GL would only run on Silicon Graphics hardware. In an attempt to overcome this limitation, OpenGL was developed in the early 1990's as a free platform-independent version of GL. The package was developed, and is still maintained, by a consortium of graphics companies.

Although OpenGL is the core library that contains the majority of the functionality, there are a number of associated libraries that are also useful. The following is a summary of the most important of the OpenGL family of graphics libraries:

- *OpenGL*: the core library, it is platform (i.e. hardware system) independent, but not windows-system independent (i.e. the code for running on Microsoft Windows will be different to the code for running on the UNIX environments X-Windows or Gnome).
- *glut*: The GL Utilities Toolkit, it contains some extra routines for drawing 3-D objects and other primitives. Using *glut* with OpenGL enables us to write windows-system independent code.
- *glu*: The OpenGL Utilities, it contains some extra routines for projections and rendering complex 3-D objects.
- *glui*: Contains some extra routines for creating user-interfaces.

Every routine provided by OpenGL or one of the associated libraries listed above follows the same basic rule of syntax:

- The *prefix* of the function name is either *gl*, *glu*, or *glut*, depending on which of these three libraries the routine is from.
- The main part of the function name indicates the purpose of the function.
- The *suffix* of the function name indicates the number and type of arguments expected by the function. For example, the suffix *3f* indicates that 3 floating point arguments are expected.

For example, the function *glVertex2i* is an OpenGL routine that takes 2 integer arguments and defines a vertex.

Some function arguments can be supplied as predefined *symbolic constants*. (These are basically identifiers that have been defined using the C++ *#define* statement.) These symbolic constants are always in capital letters, and have the same prefix convention as function names. For example, *GL_RGB*, *GL_POLYGON* and *GLUT_SINGLE* are all symbolic constants used by OpenGL and its associated libraries.

Finally, OpenGL has a number of built-in data types to help make it into a platform-independent package. For example, the C++ *int* data type may be stored as a 16-bit number on some platforms but as a 32-bit number on others. Therefore if we use these standard C++ data types with OpenGL routines the resultant code will not be platform-independent. OpenGL provides its own data types to overcome this limitation. Mostly, they have the same names as C++ data types but with the prefix *GL* attached. For example, *GLshort*, *GLint*, *GLfloat* and *GLdouble* are all built-in OpenGL data types. Although you will find that your OpenGL code will still work on your computer if you do not use these data types, it will not be as portable to other platforms so it is recommended that you do use them.

5.2. Getting Started with OpenGL

The first thing we need to know to be able to start writing OpenGL programs is which header files to include. Exactly which header files are required depends upon which library(s) are being used. For example,

If we are only using the core OpenGL library, then the following line must be added near the beginning of your source code:

#include <GL/gl.h>

If we also want to use the GL Utilities library, we must add the following line:

#include <GL/glu.h>

For the *glui* user-interface library we must add:

#include <GL/glui.h>

If we want to use the *glut* library (and this makes using OpenGL a lot easier) we do not need to include the OpenGL or *glu* header files. All we need to do is include a single header file for *glut*:

#include <GL/glut.h>
This will automatically include the OpenGL and *glu* header files too.

As well as including the appropriate header files, we should also link in the appropriate libraries when compiling. For Dev-C++, all you need to do is select *Multimedia/glut* as the project type when creating your new project – the rest will be done for you.

Now we are ready to start writing OpenGL code. Refer to the code listing in Appendix A for the following discussion. Before displaying any graphics primitives we always need to perform some basic initialisation tasks, including creating a window for display. The following lines initialise the *glut* library, define a frame buffer and create a window for display.

```
glutInit(&argc, argv);
glutInitWindowSize(640,480);
glutInitWindowPosition(10,10);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutCreateWindow("GLUT Points demo");
```

Let us examine each of these routines in turn:

- *glutInit(&argc, argv);*
    - This line initialises the *glut* library. We must call this function before any others in our *glut*/OpenGL programs.
- *glutInitWindowSize(640,480);*
    - This function call sets the size of the display window in pixels.
- *glutInitWindowPosition(10,10);*
    - This line sets the position of the top-left corner of the display window, measured in pixels from the top-left corner of the monitor screen.
- *glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);*
    - This line defines the type of frame buffer we want to have. In this case, we are asking for a single RGB (i.e. colour) frame buffer.
- *glutCreateWindow("GLUT Points demo");*
    - Finally, this function call creates the display window with the attributes defined by the previous calls.

5.3. The OpenGL Event Loop

OpenGL is an *event-driven* package. This means that it always executes code in response to *events*. The code that is executed is known as a *callback function*, and it must be defined by the programmer. OpenGL will continually detect a number of standard events in an *event loop*, and execute the associated callback functions. For example, mouse clicks and keyboard presses are considered as OpenGL events.

The most important event is the *display event*. This occurs whenever OpenGL decides that it needs to redraw the display window. It is guaranteed to happen once at the beginning of the event loop; after this it will occur whenever the window needs to be redrawn, for example if it is hidden by another window and then uncovered. Any primitives that we want to draw should be drawn in the display event callback function; otherwise they will not be displayed.

To specify a callback function we must first write a programmer-defined function, and then specify that this function should be associated with a particular event. For example, the following code specifies that the programmer-defined function *myDisplay* will be executed in response to the *display* event:

glutDisplayFunc(myDisplay);

Therefore all OpenGL programs consist of a number of parts:

- Perform any initialisation tasks such as creating the display window.
- Define any required callback functions and associate them with the appropriate events.
- Start the event loop.

To start the event loop we write:
glutMainLoop();

Note that this should be the last statement in your program – after the event loop is started normal program execution will be suspended. Look at the listing in Appendix A and see if you can identify the three parts mentioned above.

5.4. OpenGL Initialisation Routines

Now we are ready to start drawing graphics primitives. The first thing we need to understand is that the OpenGL graphics package is a *state system*. This means that it maintains a list of *state variables*, or *state parameters*, which will be used to define how primitives will be drawn. These state variables are specified separately, before the primitives are drawn. For example, if we want to draw a blue point on our display window, we first set the *drawing colour* state variable to blue, and then we draw the point at the specified location.

Therefore, before we actually draw any primitives, we need to set the values of some state variables. The listing in Appendix A is for an OpenGL program that draws some points on the display window, so before we draw the points we assign values for three state variables (at the beginning of the *myInit* function):

```
glClearColor(1.0, 1.0, 1.0, 0.0);
glColor3f(0,0,0);
glPointSize(4.0);
```

In the first line we are defining the *background colour* to have an RGB value of (1,1,1), which means white. When we clear the screen later on it will be cleared to white. The fourth argument to *glClearColor* is the *alpha* value of the background colour. The alpha value indicates the *opacity* of the colour (i.e. is it a semi-transparent colour?): a value of 1 means that it is completely opaque, or the transparency is zero. The second line listed above defines the current drawing colour to be (0,0,0), or black. Any subsequently drawn primitive will be drawn in this colour. The final line sets the point size to four pixels. Any subsequent point primitives drawn will be drawn with this size.

5.5. OpenGL Coordinate Systems

The last three lines of the *myInit* function are:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
```

These lines define how the 3-D points will be projected onto the 2-D image plane. We will return to this important subject in Chapter 5, but for the moment we will give a simplified version of what really happens. We start off with our graphics primitives defined in *world coordinates*, then using a 3-D to 2-D *projection* transformation we draw a picture of how the primitives would appear if we took their picture using a *virtual camera*. The final primitives, drawn in the display window, are in *screen coordinates*. Therefore we must specify to OpenGL how it should map points in world coordinates to points in screen coordinates. The three lines given above define this mapping. In effect, we are saying that any primitive whose *x* coordinate lies between 0 and 640, and whose *y* coordinate lies between 0 and 480, will be seen by the virtual camera and therefore drawn in the display window.

See Appendix A for a full listing of the simple OpenGL program we have worked through in this section. In particular note the following routines in the *display* function:

- *glClear*: This function is used to clear the screen before drawing. The background colour set earlier will be used for clearing.
- *glBegin*, *glVertex2i*, *glEnd*: This sequence of commands draws a number of point primitives. Actually the *glBegin ... glEnd* pair of commands is used to draw many different types of primitive in OpenGL, with the symbolic constant argument to *glBegin* defining how the vertices in between should be interpreted. In this case, *GL_POINTS* means that each vertex should be considered to be a point primitive.
- *glFlush*: When we draw primitives in OpenGL it sometimes queues the routines in a buffer. This function tells OpenGL to 'flush' this buffer, i.e. to draw any primitives now to the frame buffer.
-

Appendix A – A Simple OpenGL Program

```
#include <GL/glut.h>
#include <stdlib.h>

void myInit(void) {
    glClearColor(1.0, 1.0, 1.0, 0.0); // white background
    glColor3f(0,0,0); // black foreground
    glPointSize(4.0); // size of points to be drawn

    // establish a coordinate system for the image
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
```

```
/* GLUT display callback handler */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT); // Clear Screen
    glBegin(GL_POINTS); // draw 3 points
      glVertex2i(100,50);
      glVertex2i(100,130);
      glVertex2i(150,130);
    glEnd();
    glFlush(); // send all output to the display
}

/* Program entry point */
int main(int argc, char *argv[])
{
    glutInit(&argc, argv); // initialise the glut library
    glutInitWindowSize(640,480); // set size of the window
    glutInitWindowPosition(10,10); // position of window
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("GLUT Points demo");
    glutDisplayFunc(display); // set display callback
    myInit(); // perform other initialisation
    glutMainLoop(); // enter the GL event loop
    return EXIT_SUCCESS;
}
```

Source: Computer Graphics with OpenGL,
Donald Hearn and M. Pauline Baker,
Prentice-Hall, 2004 (Third Edition)