NATIONAL OPEN UNIVERSITY OF NIGERIA

SCHOOL OF SCIENCE AND TECHNOLOGY

COURSE CODE: CIT 671

COURSE TITLE: INTRODUCTION TO COMPUTER GRAPHICS AND ANIMATION

MODULE 1 – Introduction to Computer Graphics and Animation

UNIT 1: Introduction to computer graphics and GPU

Contents **Pages**

1.0 Introduction to Computer Graphics

Today, we find computer graphics used routinely in such diverse areas as science, engineering, medicine, business, industry, government, art, entertainment, advertising, education, and training. A major use of computer graphics is in design processes, particularly for engineering and architectural systems, but almost all products are now computer designed. Computer-aided design (CAD) methods are now routinely used in the design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, and many, many other products.

There is virtually no area in which graphical displays cannot be used to some advantage, and so it is not surprising to find the use of computer graphics so widespread. Although early applications in engineering and science had to rely on expensive and cumbersome equipment, advances in computer technology have made interactive computer graphics a practical tool.

2.0 Objectives

On completing this unit, you would be able to:

1. Explain the various application areas of computer graphics
2. Understand the elements of a Graphic system.
3. Explain Graphics processing unit and its various forms

3.0 Main Content

3.1 A Graphics System

A computer graphics system is a computer system; as such, it must have all the components of a general-purpose computer system. Let us start with the high-level view of a graphics system, as shown in the block diagram in Figure 1.1(a). There are six major elements in the Graphic system:

1. Input devices
2. Central Processing Unit (CPU)
3. Graphics Processing Unit (GPU)
4. Memory
5. Frame buffer
6. Output devices

This model is general enough to include workstations and personal computers, interactive game systems, mobile phones, GPS systems, and sophisticated image generation systems. Although most of the components are present in a standard computer, it is the way each element is specialized for

computer graphics that characterizes this diagram as a portrait of a graphics system. A complete graphic system is shown in figure 1.1(a).

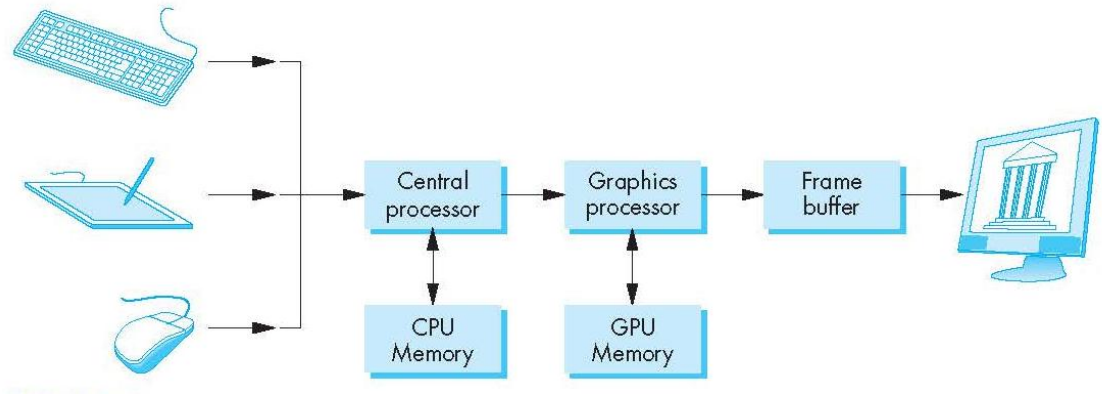


Figure 1.1(a): A graphic system (Engel et al., 1991)

3.2 Applications of Computer Graphics

The development of computer graphics has been driven both by the needs of the user community and by advances in hardware and software. The applications of computer graphics are many and varied; we can, however, divide them into four major areas:

a) Display of information

b) Design

c) Simulation and animation

d) User interfaces

Although many applications span two or more of these areas, the development of the field was based on separate work in each.

3.2.1 Display of Information

Classical graphics techniques arose as a medium to convey information among people. Although spoken and written languages serve a similar purpose, the human visual system is unrivaled both as a processor of data and as a pattern recognizer. More than 4000 years ago, the Babylonians displayed floor plans of buildings on stones. More than 2000 years ago, the Greeks were able to convey their architectural ideas graphically, even though the related mathematics was not developed until the Renaissance.

Today, the same type of information is generated by architects, mechanical designers, and drafts-people using computer-based drafting systems. For centuries, cartographers have developed maps to display celestial and geographical information. Such maps were crucial to navigators as these

people explored the ends of the earth; maps are no less important today in fields such as geographic information systems. Now, maps can be developed and manipulated in real time over the Internet.

Over the past 100 years, workers in the field of statistics have explored techniques for generating plots that aid the viewer in determining the information in a set of data. Now, we have computer plotting packages that provide a variety of plotting techniques and colour tools that can handle multiple large data sets. Nevertheless, it is still the human's ability to recognize visual patterns that ultimately allows us to interpret the information contained in the data. The field of information visualization is becoming increasingly more important as we have to deal with understanding complex phenomena from problems in bioinformatics to detecting security threats. Medical imaging poses interesting and important data-analysis problems. Modern imaging technologies in the field of medicine—such as computed tomography (CT), magnetic resonance imaging (MRI), ultrasound, and positron-emission tomography (PET)—generate three-dimensional data that must be subjected to algorithmic manipulation to provide useful information.

3.2.2 Design

Professions such as engineering and architecture are concerned with design. Starting with a set of specifications, engineers and architects seek a cost-effective and esthetic solution that satisfies the specifications. Design is an iterative process. Rarely in the real world is a problem specified such that there is a unique optimal solution. Design problems are either *overdetermined*, such that they possess no solution that satisfies all the criteria; much less an optimal solution, or *underdetermined*, such that they have multiple solutions that satisfy the design criteria. Thus, the designer works in an iterative manner. a possible design is generated, tested it, and then the results are used as the basis for exploring other solutions.

The power of the paradigm of humans interacting with images on the screen of a CRT was recognized by Ivan Sutherland over 40 years ago. Today, the use of interactive graphical tools in computer-aided design (CAD) pervades fields such as architecture and the design of mechanical parts and of very-large-scale integrated (VLSI) circuits. In many such applications, the graphics are used in a number of distinct ways. For example, in a VLSI design, the graphics provide an interface between the user and the design package, usually by means of such tools as menus and icons. In addition, after the user produces a possible design, other tools analyze the design and display the analysis graphically.

3.2.3 Simulation and Animation

Once graphics systems evolved to be capable of generating sophisticated images in real time, engineers and researchers began to use them as simulators. One of the most important uses has been in the training of pilots. Graphical flight simulators have proved both to increase safety and to reduce training expenses. The use of special VLSI chips has led to a generation of arcade games as sophisticated as flight simulators. Games and educational software for home computers are almost

as impressive as the flight simulators. The success of flight simulators led to the use of computer graphics for animation in the television, motion-picture, and advertising industries. Entire animated movies can now be made by computer at a cost less than that of movies made with traditional hand-animation techniques. The use of computer graphics with hand animation allows the creation of technical and artistic effects that are not possible with either alone. Whereas computer animations have a distinct look, we can also generate photorealistic images by computer. Images that we see on television, in movies, and in magazines often are so realistic that we cannot distinguish computer-generated or computer-altered images from photographs.

The field of virtual reality (VR) has opened up many new horizons. A human viewer can be equipped with a display headset that allows her to see separate images with her right eye and her left eye so that she has the effect of stereoscopic vision. In addition, her body location and position, possibly including her head and finger positions, are tracked by the computer. She may have other interactive devices available, including force-sensing gloves and sound. She can then act as part of a computer-generated scene, limited only by the image-generation ability of the computer. For example, a surgical intern might be trained to do an operation in this way, or an astronaut might be trained to work in a weightless environment.

3.2.4 User Interfaces

Our interaction with computers has become dominated by a visual paradigm that includes windows, icons, menus, and a pointing device, such as a mouse. From a user's perspective, windowing systems such as the X Window System, Microsoft Windows, and the Macintosh Operating System differ only in details. More recently, millions of people have become users of the Internet. Their access is through graphical network browsers, such as Firefox, Chrome, Safari, and Internet Explorer that use these same interface tools. We have become so accustomed to this style of interface that we often forget that what we are doing is working with computer graphics. Although we are familiar with the style of graphical user interface used on most workstations, advances in computer graphics have made possible other forms of interfaces.

3.3 The CPU and the GPU

In a simple system, there may be only one processor, the Central Processing Unit (CPU) of the system, which must do both the normal processing and the graphical processing. The main graphical function of the processor is to take specifications of graphical primitives (such as lines, circles, and polygons) generated by application programs and to assign values to the pixels in the frame buffer that best represent these entities. For example, a triangle is specified by its three vertices, but to display its outline by the three line segments connecting the vertices, the graphics system must generate a set of pixels that appear as line segments to the viewer. The conversion of geometric entities to pixel colours and locations in the frame buffer is known as rasterization, or scan conversion.

In early graphics systems, the frame buffer was part of the standard memory that could be directly addressed by the CPU. Today, virtually all graphics systems are characterized by special-purpose Graphics Processing Units (GPUs), custom-tailored to carry out specific graphics functions. The GPU can be either on the mother board of the system or on a graphics card. The frame buffer is accessed through the graphics processing unit and usually is on the same circuit board as the GPU. GPUs have evolved to where they are as complex as or even more complex than CPUs. They are characterized by both special-purpose modules geared toward graphical operations and a high degree of parallelism—recent GPUs contain over 100 processing units, each of which is user programmable. GPUs are so powerful that they can often be used as mini supercomputers for general purpose computing.

3.3.1 Graphics Processing Unit

A Graphics Processing Unit or GPU (also occasionally called visual processing unit or VPU) is a specialized circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. In a personal computer, a GPU can be on a video card, or it can be on the motherboard, or in certain CPUs, on the CPU die. An example is the GeForce 6600GT GPU shown in Figure 1.1(b). More than 90% of new desktop and notebook computers have integrated GPUs, which are usually far less powerful than those on a dedicated video card.

Figure 1.1(b): GeForce 6600GT GPU

3.4 GPU forms

There are various GPU forms characterized by their interfaces with the main board. The common ones are mentioned below.

3.4.1 Dedicated graphics cards

The GPUs of the most powerful class typically interface with the motherboard by means of an expansion slot such as PCI Express (PCIe) or Accelerated Graphics Port (AGP) and can usually be replaced or upgraded with relative ease, assuming the motherboard is capable of supporting the upgrade. A few graphics cards still use Peripheral Component Interconnect (PCI) slots, but their bandwidth is so limited that they are generally used only when a PCIe or AGP slot is not available.

A dedicated GPU is not necessarily removable, nor does it necessarily interface with the motherboard in a standard fashion. The term "dedicated" refers to the fact that dedicated graphics cards have RAM that is dedicated to the card's use, not to the fact that most dedicated GPUs are removable. Dedicated GPUs for portable computers are most commonly interfaced through a non-standard and often proprietary slot due to size and weight constraints. Such ports may still be considered PCIe or AGP in terms of their logical host interface, even if they are not physically interchangeable with their counterparts.

3.4.2 Integrated graphics solutions

Integrated graphics solutions, shared graphics solutions, or Integrated Graphics Processors (IGP) utilize a portion of a computer's system RAM rather than dedicated graphics memory. They are integrated into the motherboard. Exceptions are AMD's IGPs that use dedicated side-port memory on certain motherboards, and APUs, where they are integrated with the CPU die. Computers with integrated graphics account for 90% of all PC shipments. These solutions are less costly to implement than dedicated graphics solutions, but are less capable. Historically, integrated solutions were often considered unfit to play 3D games or run graphically intensive programs but could run less intensive programs such as Adobe Flash. Modern desktop motherboards often include an integrated graphics solution and have expansion slots available to add a dedicated graphics card later.

As a GPU is extremely memory intensive, an integrated solution may find itself competing for the already relatively slow system RAM with the CPU, as it has minimal or no dedicated video memory. System RAM may be 2 GB/s to 16 GB/s, yet dedicated GPUs enjoy between 10 GB/s to over 300 GB/s of bandwidth depending on the model (for instance the GeForce GTX 590 and Radeon HD 6990 provide approximately 320 GB/s between dual memory controllers). Older integrated graphics chipsets lacked hardware transform and lighting, but newer ones include it

3.4.3 Hybrid solutions

This newer class of GPUs competes with integrated graphics in the low-end desktop and notebook markets. The most common implementations of this are ATI's HyperMemory and NVIDIA's TurboCache. Hybrid graphics cards are somewhat more expensive than integrated graphics, but much less expensive than dedicated graphics cards. These share memory with the system and have a small dedicated memory cache, to make up for the high latency of the system RAM. Technologies within PCI Express can make this possible. While these solutions are sometimes advertised as having as much as 768MB of RAM, this refers to how much can be shared with the system memory.

3.5 The Graphics pipeline

In 3D computer graphics, the terms graphics pipeline or rendering pipeline most commonly refers to the current state of the art method of rasterization-based rendering as supported by commodity graphics hardware. The graphics pipeline typically accepts some representation of a three-dimensional primitive as an input and results in a 2D raster image as output. OpenGL and Direct3D are two notable 3D graphic standards, both describing very similar graphic pipeline.

The rendering pipeline is mapped onto current graphics acceleration hardware such that the input to the graphics card (GPU) is in the form of vertices. These vertices then undergo transformation and per-vertex lighting. At this point in modern GPU pipelines a custom vertex shader program can be used to manipulate the 3D vertices prior to rasterization. Once transformed and lit, the vertices undergo clipping and rasterization resulting in fragments as shown in figure 1.1(d). A second custom shader program can then be run on each fragment before the final pixel values are output to the frame buffer for display.

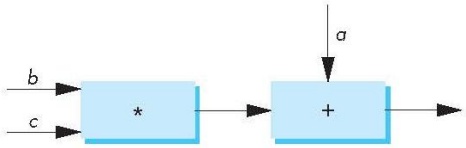


FIGURE 1.1(c) Arithmetic pipeline (Ed Angel (1991)

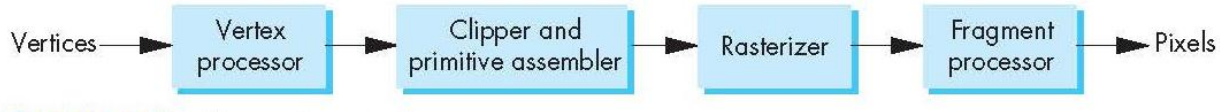


FIGURE 1.1(d) Geometric pipeline. (Ed Angel, 1991)

The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor since all vertices and fragments can be thought of as independent. This allows

all stages of the pipeline to be used simultaneously for different vertices or fragments as they work their way through the pipe. In addition to pipelining vertices and fragments, their independence allows graphics processors to use parallel processing units to process multiple vertices or fragments in a single stage of the pipeline at the same time.

4.0 Conclusion

A major use of computer graphics is in design processes, particularly for engineering and architectural systems, design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, and many, many other products.

5.0 Summary

In this unit, we have studied Computer Graphics, its application areas, computer graphics systems and also Graphic processing units and its various forms.

6.0 Tutor Marked Assignment

1. What do you understand by Computer Graphics?
2. Identify application areas of computer graphics.
3. Draw a graphic system.
4. Explain what GPU is meant for and write a short note and its various types.

7.0 References/Further Reading

1. Jeffrey J. McConnell (2006). *Computer Graphics: Theory into Practice*. Jones & Bartlett Publishers. ISBN:0-7637-2250-2
2. R. D. Parslow, R. W. Prowse, Richard Elliot Green (1969). *Computer Graphics: Techniques and Applications*. ISBN-13: 978-0306200168
3. Peter Shirley and others. (2005). *Fundamentals of computer graphics*. A.K. Peters, Ltd. ISBN-13: 978:1568814692
4. David Salomon (1999). *Computer Graphics and Geometric Modeling*, Springer ISBN 0-387-98682-0.
5. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL* Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
6. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247

MODULE 1 – Introduction to Computer Graphics and Animation

UNIT 2: Illumination: The BRDF.

Contents **Pages**

1.0 Introduction to Bi-directional Reflection Distribution Function (BRDF)

One of the most general means to characterize the reflection properties of a surface is by use of the bi-directional reflection distribution function (BRDF), a function which defines the spectral and spatial reflection characteristic of a surface. The BRDF of a surface is the ratio of reflected radiance to incident irradiance at a particular wavelength:

$$\rho(\Theta_i; \Theta_r; \lambda) = \frac{dL_r(\Theta_i; \Theta_r; \lambda)}{dE_i(\Theta_i; \lambda)}$$

where the subscripts i and r denote incident and reflected respectively, $\Theta = (\theta, \phi)$is the direction of light propagation, $\lambda$is the wavelength of light, *L* is radiance, and *E* is irradiance.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the BRDFs
2. Understand the application of BRDFs
3. Understand the features of BRDF models.

3.0 Main Content

3.1 An overview of the BRDF

To understand the concept of a BRDF and how BRDFs can be used to improve realism in interactive computer graphics, we begin by discussing what we know about light and how light interact with matter. In general, when light interacts with matter, a complicated light-matter dynamic occurs. This interaction depends on the physical characteristics of the light as well as the physical composition and characteristics of the matter. For example, a rough opaque surface such as sandpaper will reflect light differently than a smooth reflective surface such as a mirror. Figure 1.2(a) shows a typical light-matter interaction scenario.

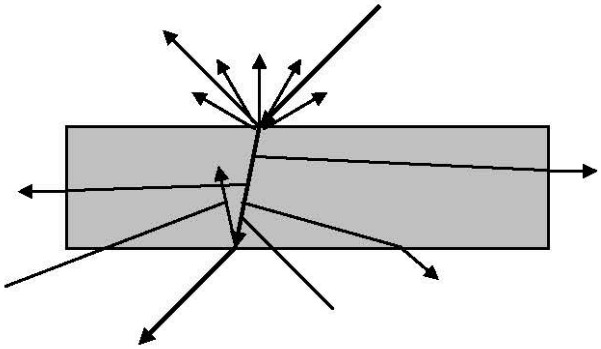


Figure 1.2(a): Light Interactions.

From this figure, we make a couple of observations about light. First, when light makes contact with a material, three types of interactions may occur: light reflection, light absorption, and light transmittance. That is, some of the incident light is reflected, some of the light is transmitted, and another portion of the light is absorbed by the medium itself.

Light incident at surface = light reflected + light absorbed + light transmitted

For opaque materials, the majority of incident light is transformed into reflected light and absorbed light. As a result, when an observer views an illuminated surface, what is seen is reflected light, i.e. the light that is reflected towards the observer from all visible surface regions. A BRDF describes how much light is *reflected* when light makes contact with a certain material. Similarly, a BTDF (Bi-directional Transmission Distribution Function) describes how much light is *transmitted* when light makes contact with a certain material.

In general, the degree to which light is reflected (or transmitted) depends on the viewer and light position relative to the surface normal and tangent. Consider, for example, a shiny plastic teapot illuminated by a white point light source. Since the teapot is made of plastic, some surface regions will show a shiny highlight when viewed by an observer. If the observer moves (i.e. changes view direction), the position of the highlight shifts. Similarly, if the observer and teapot both remain fixed, but the light source is moved, the highlight shifts. Since a BRDF a measure how light is reflected, it must capture this view and light-dependent nature of reflected light. Consequently, a BRDF is a function of incoming (light) direction and outgoing (view) direction relative to a local orientation at the light interaction point.

Additionally, when light interacts with a surface, different wavelengths (colours) of light may be absorbed, reflected, and transmitted to varying degrees depending upon the physical properties of the material itself. This means that a BRDF is also a function of wavelength.

Finally, light interacts differently with different regions of a surface. This property, known as *positional variance*, is most noticeably observed in materials such as wood that reflect light in a manner that produces surface detail. Both the ringing and striping patterns often found in wood are indications that the BRDF for wood varies with the surface spatial position. Many materials exhibit this positional variance because they are not entirely composed of a single material. Instead, most real world materials are heterogeneous and have unique material composition properties which vary with the density and stochastic characteristics of the sub-materials from which they are comprised.

Considering the dependence of a BRDF on the incoming and outgoing directions, the wavelength of light under consideration, and the positional variance, a general BRDF in functional notation can be written as

$$\mathrm{BRDF}_{\lambda}(\theta_i, \phi_i, \theta_o, \phi_o, u, v)$$

Where λ is used to indicate that the BRDF depends on the wavelength under consideration, the parameters $\phi i, \theta_{\mathrm{i}}$ represent the incoming light direction in spherical coordinates, the parameters ϕ_0, θ_0 represent the outgoing reflected direction in spherical coordinates, and *u* and *v* represent the surface position parameterized in texture space

Though a BRDF is truly a function of position, sometimes the positional variance is not included in a BRDF description. Instead, it is common to see a BRDF written as a function of incoming and outgoing directions and wavelength only (i.e. $\mathbf{BRDF}_{\boldsymbol{\lambda}}(\boldsymbol{\theta_i \phi_i \theta_0 \phi_0})$) Such BRDFs are often called *position-invariant* or *shift-invariant* BRDFs. When the spatial position is not included as a parameter to the function, an assumption is made that the reflectance properties of a material do not vary with spatial position. In general, this is only valid for homogenous materials. One way to introduce the positional variance is through the use of a detail texture. By adding or modulating the result of a BRDF lookup with a texture, it is possibly to reasonably approximate a spatially variant BRDF.

For the remainder of this unit, we will denote a position-invariant BRDF in functional notation as

$$\mathbf{BRDF}_{\boldsymbol{\lambda}}(\boldsymbol{\theta_i \phi_i \theta_0 \phi_0})$$

where $\phi i, \theta_{\mathrm{i}}$, ϕ_0, θ_0 have the same meaning as before.

When describing a BRDF in this functional notation, it is sometimes convenient to omit the λ subscript for the sake of notation simplicity. When this is done, keep in mind that the values produced by a BRDF do depend on the wavelength or colour channel under consideration. In practice what this means is that in terms of the RGB colour convention, the value of the BRDF function must be determined separately for each colour channel (i.e. R, G, and B separately). For convenience, it's usually preferred not to specify a particular colour channel in the subscript. The

implicit assumption is that the programmer knows that a BRDF value must be determined for each colour channel of interest separately. Given this slightly abbreviated form, the position-invariant BRDF associated with a single colour channel can be considered to be a function of 4 variables. When the RGB colour components are considered as a group, the BRDF is a three-component vector function.

3.2 The Definition of a BRDF

Up until this point, the exact definition of a BRDF has not been discussed. Suppose we are given an incoming light direction, w_i, and an outgoing reflected direction, w_o, each defined relative to a small surface element. A BRDF is defined as the ratio of the quantity of reflected light in direction w_o, to the amount of light that reaches the surface from direction w_i. To make this clear, let's call the quantity of light reflected from the surface in direction w_o, L_o, and the amount of light arriving from direction w_i, E_i. Then a BRDF is given by

$$BRDF = \frac{L_o}{E_i}. \qquad \text{Equation 1.2}$$

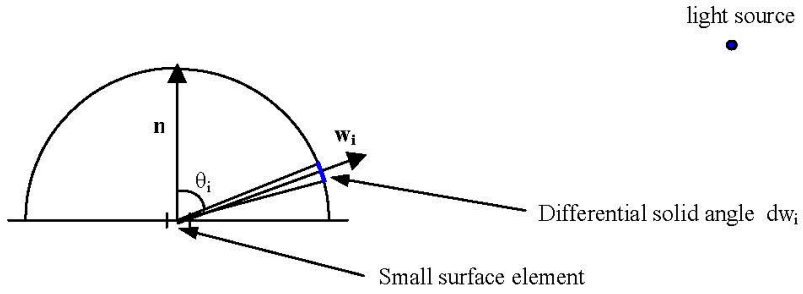


Figure 1.2(b): A surface element illuminated by a light source.

Now consider figure 1.2(b). The figure shows a small surface element (i.e. a pixel/surface point) that is being illuminated by a point light source. The amount of light arriving from direction w_i is proportional to the amount of light arriving at the differential solid angle. Suppose the light source in the figure has intensity L_i. Since the differential solid angle is small, it is essentially a flat region on the hemisphere. As a result, the region is uniformly illuminated as the same quantity of light, L_i, arrives for each position on the differential solid angle. So the total amount of incoming light arriving through the region is $Li*dw$. The only problem is that this amount of light is with respect to the differential solid angle and not the actual surface element under consideration. To determine the amount of light with respect to the surface element, the incoming light must be "spread out" or projected onto the surface element. This projection is similar to that which happens with diffuse Lambertian lighting and is accomplished by modulating that amount by $\cos\theta_i = N * w_i$. This means $E_i = L_i \cos\theta_i dw_i$.

As a result, a BRDF is given by

$$BRDF = \frac{L_o}{L_i \cos\theta_i \, dw_i}.$$

From this definition, observe two interesting results. First, a BRDF is not bounded to the range [0, 1] – a common misconception about BRDFs. Although the ratio L_o to L_i must be in [0, 1], the division by the cosine term in the denominator implies that a BRDF may have values larger than 1. Secondly, a BRDF is not a unit-less function. Since the BRDF definition above includes a division by the solid angle (which has units steradians (sr)), the units of a BRDF are inverse steradians (sr^{-1}).

3.3 Classes and Properties of BRDFs

There are two classes of BRDFs and two important properties. BRDFs can be classified into two classes: *isotropic* BRDFs and *anisotropic* BRDFs. The two important properties of BRDFs are *reciprocity* and *conservation of energy*.

The term 'isotropic' is used to describe BRDFs that represent reflectance properties that are invariant with respect to rotation of the surface around the surface normal vector. Consider a small relatively smooth surface element and fix the light and viewer positions. If we were to rotate the surface about its normal, the BRDF value (and consequently the resulting illumination) would remain unchanged. Materials with this characteristic such as smooth plastics have isotropic BRDFs.

Anisotropy, on the other hand, refers to BRDFs that describe reflectance properties that do exhibit change with respect to rotation of the surface around the surface normal vector. Some examples of materials that have anisotropic BRDFs are brushed metal, satin, and hair. In general, most real-world BRDFs are anisotropic to some degree, but the notion of isotropic BRDFs is useful because many classes of analytical BRDF models fall within this class. In general, most real-world BRDFs are probably more isotropic than anisotropic though many real-world surfaces have subtle anisotropy. Any material that exhibits even the slightest anisotropic reflection has a BRDF that is anisotropic. BRDFs based on physical laws and considered to be *physically plausible* have two properties: *reciprocity* and *conservation of energy*.

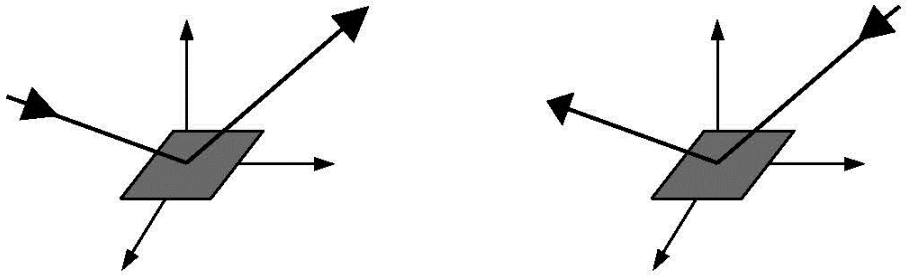


Figure 1.2(c): The Reciprocity Principle

The reciprocity property is illustrated in figure 1.2(c). Basically it says that if the sense of the traveling light is reversed, the value of the BRDF remains unchanged. That is, if the incoming and outgoing directions are swapped, the value of the BRDF does not change. Mathematically, this property is written as

$$\mathrm{BRDF}_\lambda(\theta_i, \phi_i, \theta_o, \phi_o) = \mathrm{BRDF}_\lambda(\theta_o, \phi_o, \theta_i, \phi_i).$$

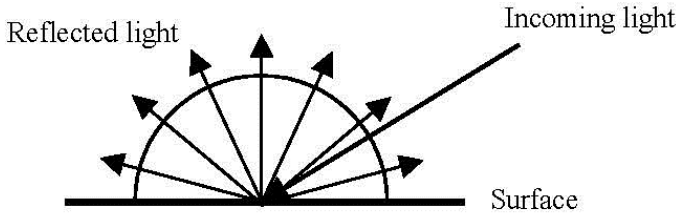


Figure 1.2(d): Conservation of Energy- The quantity of light reflected must be less than or equal to the quantity of incident light.

The conservation of energy constraint has to do with the scattering of light during the light-matter interaction. In general, this property states that when light from a single incoming direction makes contact with a surface and is reflected/scattered over the sphere of outgoing directions, the total quantity of light that is scattered cannot exceed the original quantity of light arriving at the surface. Figure 1.2(d) illustrates this property. For each one unit of light energy that arrives at a point, no more than one unit of light energy can be reflected in total to all possible outgoing directions.

By considering the definition of a BRDF (the ratio of the reflected light to incident light divided by the projected solid angle), this means the sum over all outgoing directions of the BRDF times the projected solid angle must be less than one in order for the ratio of the total amount of reflected light to the incident light to be less than one. Mathematically, this is written as

$$\sum_{out} \mathrm{BRDF}_\lambda(\theta_i, \phi_i, \theta_o, \phi_o) \cos\theta_o \, dw_o \leq 1.$$

When considering the continuous hemisphere of all outgoing reflected directions, the sum becomes an integral and this conservation property becomes

$$\int_\Omega \mathrm{BRDF}_\lambda(\theta_i, \phi_i, \theta_o, \phi_o) \cos\theta_o \, dw_o \leq 1.$$

The symbol $\int_\Omega$ indicates an integral over a hemisphere of all directions.

3.4 Related functions

1. The **Spatially Varying Bidirectional Reflectance Distribution Function** (SVBRDF) is a 6-dimensional function, $f_r(\omega_i, \omega_o, \mathbf{x})$, where $\mathbf{x}$describes a 2D location over an object's surface.

2. The **Bidirectional Texture Function** (BTF) is appropriate for modeling non-flat surfaces, and has the same parameterization as the SVBRDF; however in contrast, the BTF includes non-local scattering effects like shadowing, masking, inter-reflections or subsurface scattering. The functions defined by the BTF at each point on the surface are thus called **Apparent BRDFs**.

3. The **Bidirectional Surface Scattering Reflectance Distribution Function** (BSSRDF), is a further generalized 8-dimensional function $S(\mathbf{x}_i, \omega_i, \mathbf{x}_o, \omega_o)$ in which light entering the surface may scatter internally and exit at another location.

In all these cases, the dependence on wavelength has been ignored and binned into RGB channels. In reality, the BRDF is wavelength dependent, and to account for effects such as iridescence or luminescence the dependence on wavelength must be made explicit: $f_r(\lambda_i, \omega_i, \lambda_o, \omega_o)$.

3.5 Physically based BRDFs

Physically based BRDFs have additional properties, including,

1. positivity: $f_r(\omega_i, \omega_o) \geq 0$
2. Obeying Helmholtz reciprocity: $f_r(\omega_i,\omega_o) = f_r(\omega_o,\omega_i)$.
3. conserving energy: $\forall \omega_i, \int_\Omega f_r(\omega_i, \omega_o) \cos\theta_o d\omega_o \leq 1$

3.6 Applications of BRDF

The BRDF is a fundamental radiometric concept, and used in computer graphics for photorealistic rendering of synthetic scenes, as well as in computer vision for many inverse problems such as object recognition.

3.6 Features of BRDF models

BRDFs can be measured directly from real objects using calibrated cameras and light sources; however, many phenomenological and analytic models have been proposed including the Lambertian reflectance model frequently assumed in computer graphics. Some useful features of recent models include:

1. accommodating anisotropic reflection
2. editable using a small number of intuitive parameters
3. accounting for Fresnel effects at grazing angles
4. being well-suited to Monte Carlo methods.

4.0 Conclusion

This unit has presented some of the basic terminologies and concepts about BRDFs, its applications and useful features of recent models. The degree to which light is reflected (or transmitted) depends on the viewer and light position relative to the surface normal and tangent. BRDF is also a function of wavelength.

5.0 Summary

The bidirectional reflectance distribution function is a four-dimensional function that defines how light is reflected at an opaque surface and accordingly is used in computer graphics for photorealistic rendering of synthetic scenes, as well as in computer vision for many inverse problems such as object recognition.

6.0 Tutor Marked Assignment

1. What do you understand by BRDF?
2. Identify Application areas and features of BRDFs.
3. Explain the classes and properties of BRDFs
4. Highlight the features of BRDF models.

5. Differentiate between Isotropic BRDF and Anisotropic BRDF.

7.0 References/Further Reading

1. Jeffr Ward, Gregory J. (1992). "*Measuring and modeling anisotropic reflection*". Proceedings of SIGGRAPH. pp. 265–272.
2. S.K. Nayar and M. Oren, "*Generalization of the Lambertian Model and Implications for Machine Vision"*. International Journal on Computer Vision, Vol. 14, No. 3, pp. 227–251, Apr, 1995
3. Michael Ashikhmin, Peter Shirley (2000), *An Anisotropic Phong BRDF Model*, Journal of Graphics Tools 2000
4. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL* Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
5. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247

MODULE 1 – Introduction to Computer Graphics and Animation
UNIT 3: Vectors and dot Products

Contents **Pages**

1.0 Introduction to Vectors

Vectors are geometric objects that have a **length** and a **direction**. We can also talk about a vector's **tail** (where it begins) and **head** (where it ends up). A vector is like a point, in that it is described by a set of coordinates in a given dimension. But there are differences:

1. A point has an absolute position within a coordinate system. A vector has no position; the same vector can appear anywhere.
2. A point has no dimension to it. A vector has a length as well as a direction.

Vectors are very important in computer graphics. For example, they are needed to:

1. Analyze shapes: find the point at which two lines intersect, the distance of a point to a line, or whether a shape is convex or concave.
2. Determine visibility: find objects closest to the eye (ray tracing) or determine whether a plane is facing away from us (back-face culling).
3. Calculate lighting effects: determine how much light hits a surface (illumination), how much of that light is seen by the viewer (reflection), and what other objects are reflected in that surface (ray tracing).

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the what vectors are
2. Understand dot products and cross products
3. Understand vector operators and how to utilize them.
4. Understand the properties of Dot and Cross Products of vectors

3.0 Main Content

3.1 Adding vectors and points

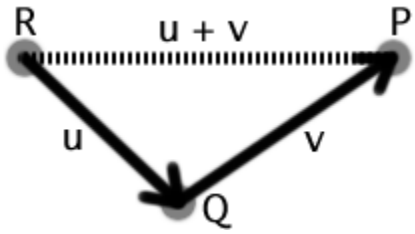
Points and vectors can be used to define one another by adding and subtracting the coordinates. Given that *P* and *Q* are points and *u* and *v* are vectors, then

1. $P - Q$ is a vector with its tail at *Q* and its head at *P*
2. $P + v$ is a new point (*P* displaced by the quantities in *v*)
3. $u + v$ is another vector

Coordinates are added and subtracted as follows:

If $a = (a_x, a_y, a_z)$ and $b = (b_x, b_y, b_z)$ then

$a + b = (a_x + b_x, a_y + b_y, a_z + b_z)$ and $a - b = (a_x - b_x, a_y - b_y, a_z - b_z)$.



For example, consider the illustration at left. Imagine that R = (2, 3, 1), Q = (4, 1, 1), and P = (7, 3, 1). Then

1. u = Q - R = (2, -2, 0) and Q = R + u
2. v = P - Q = (3, 2, 0) and P = Q + v

u + v = (Q - R) + (P - Q) = P - R = (5, 0, 0)

3.2 Other vector operations

You can change the length of a vector by multiplying it with a scalar value. Given a scalar value *s* and a vector $v = (v_x, v_y, v_z)$ then $sv = (sv_x, sv_y, sv_z)$. For example, if $s = 0.5$ and $v = (4, 3, 0)$ then $sv = (2, 1.5, 0)$.

You can find the length (or magnitude) of a vector using the Pythagorean Theorem. Given a vector $v = (v_x, v_y, v_z)$, the magnitude of v is $|v| = sqrt(v_x*v_x, v_y*v_y, v_z*v_z)$. For example, if $v = (4, 3, 0)$ then $|v| = 5$.

A unit vector is a vector of length 1. For any vector, you can find a corresponding unit vector (with the same direction) by dividing each of the coordinate values by the magnitude of the original vector. In other words, given a vector $v = (v_x, v_y, v_z)$, the unit vector is $(v_x / |v|, v_y / |v|, v_z / |v|)$. For example, if $v = (4, 3, 0)$ then the unit vector with the same direction is (4/5, 3/5, 0/5) = (0.8, 0.6, 0).

3.3 Dot product

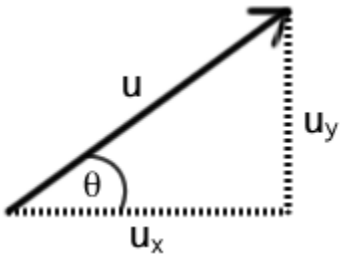
The dot (or inner) product of 2 vectors produces a scalar value. The dot product is used to solve a number of important geometric problems in graphics. The dot product for 3-dimensional vectors is solved as follows:

If $u = (u_x, u_y, u_z)$ and $v = (v_x, v_y, v_z)$ then
$u \bullet v = u_x v_x + u_y v_y + u_z v_z$.

3.4 Properties of Dot products

The dot product has the following properties:

1. Symmetry: $u \bullet v = v \bullet u$
2. Linearity: $(u + w) \bullet v = (u \bullet v) + (w \bullet v)$
3. Homogeneity: $(su) \bullet v = s(u \bullet v)$
4. $|v| = sqrt(v \bullet v)$

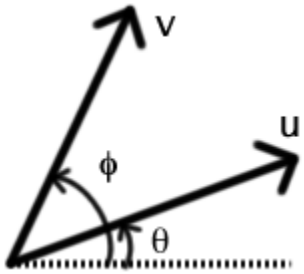


The dot product can be used to determine the angle between two vectors.

From the Pythagorean Theorem, we know that

$\cos \theta = u_x / |u|$ and $u_x = \cos \theta * |u|$

$\sin\theta = u_y / |u|$ and $u_y = \sin\theta * |u|$

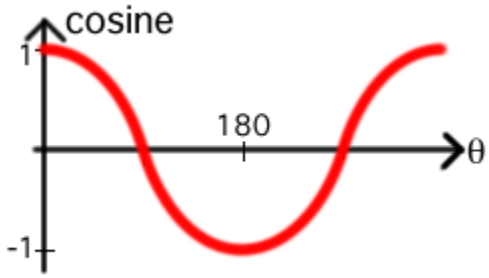


Therefore,

$u \bullet v = \cos\theta|u|\cos\varphi|v| + \sin\theta|u|\sin\varphi|v|$

$= |u||v|(\cos\theta\cos\varphi + \sin\theta\sin\varphi)$

$= |u||v|\cos(\theta-\varphi)$



And so,

$\cos(\theta-\varphi) = (u \bullet v) / (|u||v|)$

There is no need to calculate the exact cosine to know whether the angle is acute, obtuse, or a right angle. Because $|u||v|$ is always a positive value, the sign of $\cos(\theta-\varphi)$ will take on the sign of $u \bullet v$. So,

$u \bullet v > 0$ implies the angle is **acute** ($-90° < (\theta-\varphi) < 90°$);

$u \bullet v < 0$ implies the angle is **obtuse** ($90° < (\theta-\varphi) < 270°$); and

$u \bullet v = 0$ implies the angle is **right** ($(\theta-\varphi) = 90°$ or $(\theta-\varphi) = -90°$), i.e. the vectors are **perpendicular**.

3.5 Cross Product

The cross (or vector) product of 2 vectors produces another vector which is perpendicular (orthogonal) to both of the vectors used to find it.

The cross product is defined in terms of the standard unit vectors ***i***, ***j***, and ***k***, where

1. ***i*** = (1, 0, 0)
2. ***j*** = (0, 1, 0)
3. ***k*** = (0, 0, 1)

The cross product for 3-dimensional vectors is then solved as follows:

If $u = (u_x, u_y, u_z)$ and $v = (v_x, v_y, v_z)$ then

$u \times v = ((u_y v_z - u_z v_y)\boldsymbol{i} + (u_z v_x - u_x v_z)\boldsymbol{j} + (u_x v_y - u_y v_x)\boldsymbol{k})$.

This form can be hard to remember, and so we can also write the cross product as a determinant:

$$u \times v = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{K} \\ u_x & u_y & U_z \\ v_x & v_y & V_z \end{vmatrix}$$

3.6 Properties of Cross Products

The cross product has the following properties:

1. Antisymmetry: $u \times v = -v \times u$
2. Linearity: $u \times (v + w) = (u \times v) + (u \times w)$
3. Homogeneity: $(su) \times v = s(u \times v)$
4. $\boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k}$; $\boldsymbol{j} \times \boldsymbol{k} = \boldsymbol{i}$; $\boldsymbol{k} \times \boldsymbol{i} = \boldsymbol{j}$

The result of $u \times v$ is a vector that is perpendicular (orthogonal) to both u and v.

The result of $u \times v$ follows the right-hand rule:

1. Place your right hand at u and curl your fingers toward v. Your hand should be enclosing the smaller angle (<= 180°) between u and v.
2. Stick out your thumb: it points in the direction of $u \times v$.

The length of $u \times v$ equals the area of the parallelogram determined by u and v, which is

$|u \times v| = |u||v| * \sin\theta$

where θ is the angle from u to v or v to u (whichever is less).

4.0 Conclusion

Vector graphics editors typically allow rotation, movement, mirroring, stretching, skewing, affine transformations, changing of z-order and combination of primitives into more complex objects. More sophisticated transformations include set operations on closed shapes (union, difference, intersection, etc.).

5.0 Summary

In Vectors are geometric objects that have a **length** and a **direction**. Vectors are very important in computer graphics to analyze shapes, Determine visibility and Calculate lighting effects.

6.0 Tutor Marked Assignment

1. What do you understand by vectors? Differentiate between vectors and scalars.
2. Identify the properties of Cross products and Dot products of vector
3. Try these problems to test your understanding of this material.
1. For each of the following, calculate the **coordinates**. Indicate whether the result is a point or a vector.
    1. $v + u$, where $v = (-1, 0, 5)$ and $u = (2, 1, 1)$
    2. $P + v$, where $P = (1, 2, 3)$ and $v = (-1, -2, -3)$
    3. $P - Q$, where $P = (5, 5, 5)$ and $Q = (1, 2, 3)$
2. For each of the following, calculate sv and $|sv|$ when
    1. $s = 3$, $v = (1, 1, 1)$
    2. $s = 0.25$, $v = (-4, 8, 2)$
3. For each of the following vectors v and u, calculate the **dot product**. What does the result tell you about the angle between the vectors?
    1. v = (1, 0, 0) and u = (0, 1, 0)
    2. v = (1, 1, -1) and u = (2, 1, 0)
    3. v = (-2, 0, 0) and u = (1, 1, 1)
4. Calculate the **unit normal vector** for the polygon defined by points P0 = (1, 1, 1), P1 = (5, 1, 4) and P2 = (2, 1, 1).

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition*), Prentice Hall, 1997, ISBN: 0135309247
3. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, 4th edition, Academic Press, San Diego, 1996. ISBN-13: 978-0122490545
4. J.D. Foley et al., *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass., 1990. ISBN-13: 978-0201848403

MODULE 2 – Transformations, Camera models, Rasterization and Mapping techniques

UNIT 1: Transformations.

Contents **Pages**

1.0 Introduction to Transformations

Transformations are one of the primary vehicles used in computer graphics to manipulate objects in three-dimensional space. Their development is motivated by the process of converting coordinates between frames, which results in the generation of a 4x4 matrix. We can generalize this process and develop matrices that implement various transformations in space.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand how transformations work
2. Differentiate between 2D and 3D transformations

3. Identify classes of transformations.

3.0 Main Content

3.1 2D Transformations

Given a point cloud, polygon, or sampled parametric curve, we can use transformations for several purposes:

1. Change coordinate frames (world, window, viewport, device, etc).
2. Compose objects of simple parts with local scale/position/orientation of one part defined with regard to other parts. For example, articulated objects.
3. Use deformation to create new shapes.
4. Useful for animation.

There are three basic classes of transformations:

i. **Rigid body** - Preserves distance and angles.

Examples: translation and rotation.

ii. **Conformal** - Preserves angles.

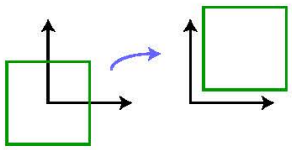
Examples: translation, rotation, and uniform scaling.

iii. **Affine** - Preserves parallelism. Lines remain lines.

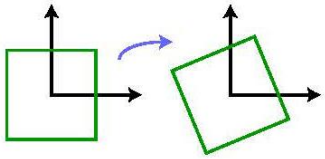
Examples: translation, rotation, scaling, shear, and reflection.

Examples of transformations:

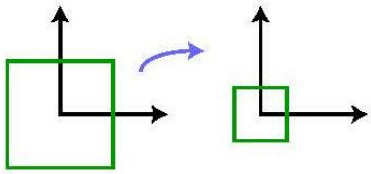
1 **Translation** by Vector $\vec{t}$: $P_1 = P_0 + \vec{t}$



1. **Rotation** by Clockwise θ: $\bar{p}_1 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \bar{p}_0.$

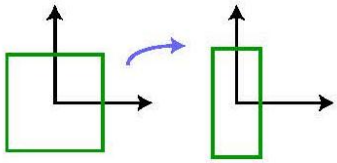


2. **Uniform Scaling** by Scalar : $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \bar{p}_0.$

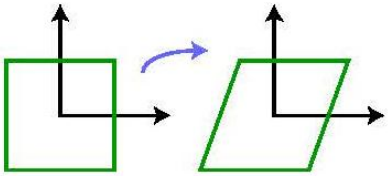


Type equation here.

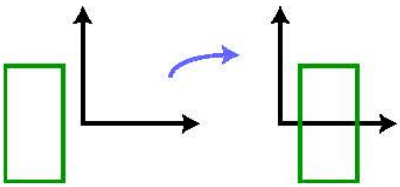
3. **Non-uniform Scaling** by a and b: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \bar{p}_0.$



4. **Shear** by Scalar *h*: $\bar{p}_1 = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \bar{p}_0.$



5. **Reflection** about the *y*-axis: $\bar{p}_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \bar{p}_0.$



3.2 Affine Transformations

An **affine transformation** takes a point $\bar{p}$ to $\bar{q}$ according to $\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t}$, a linear transformation followed by a translation. You should understand the following proof

- The inverse of an affine transformation is also affine, assuming it exists.

Proof:
Let $\bar{q} = A\bar{p} + \vec{t}$ and assume A^{-1} exists, i.e. $\det(A) \neq 0$.
Then $A\bar{p} = \bar{q} - \vec{t}$, so $\bar{p} = A^{-1}\bar{q} - A^{-1}\vec{t}$. This can be rewritten as $\bar{p} = B\bar{q} + \vec{d}$, where $B = A^{-1}$ and $\vec{d} = -A^{-1}\vec{t}$.

Note:

The inverse of a 2D linear transformation is

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

• Lines and parallelism are preserved under affine transformations.

Proof:

To prove lines are preserved, we must show that $\bar{q}(\lambda) = F(\bar{l}(\lambda))$ is a line, where $F(\bar{p}) = A\bar{p} + \vec{t}$ and $\bar{l}(\lambda) = \bar{p}_0 + \lambda\vec{d}$.

$$\begin{aligned} \bar{q}(\lambda) &= A\bar{l}(\lambda) + \vec{t} \\ &= A(\bar{p}_0 + \lambda\vec{d}) + \vec{t} \\ &= (A\bar{p}_0 + \vec{t}) + \lambda A\vec{d} \end{aligned}$$

This is a parametric form of a line through $A\bar{p}_0 + \vec{t}$ with direction $A\vec{d}$.

• Given a closed region, the area under an affine transformation $A\bar{p} + \vec{t}$ is scaled by $\det(A)$.

Note:

- Rotations and translations have $\det(A) = 1$.
- Scaling $A = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$ has $\det(A) = ab$.
- Singularities have $\det(A) = 0$.

Example:

The matrix $A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ maps all points to the x-axis, so the area of any closed region will become Zero. We have $\boldsymbol{det(A) = 0}$, which verifies that any closed region's area will be scaled by zero

• A composition of affine transformations is still affine.

Proof:

Let $F_1(\bar{p}) = A_1\bar{p} + \vec{t}_1$ and $F_2(\bar{p}) = A_2\bar{p} + \vec{t}_2$.

Then,

$$\begin{aligned} F(\bar{p}) &= F_2(F_1(\bar{p})) \\ &= A_2(A_1\bar{p} + \vec{t}_1) + \vec{t}_2 \\ &= A_2A_1\bar{p} + (A_2\vec{t}_1 + \vec{t}_2). \end{aligned}$$

Letting $A = A_2A_1$ and $\vec{t} = A_2\vec{t}_1 + \vec{t}_2$, we have $F(\bar{p}) = A\bar{p} + \vec{t}$, and this is an affine transformation.

3.3 Homogeneous Coordinates

Homogeneous coordinates are another way to represent points to simplify the way in which we express affine transformations. Normally, bookkeeping would become tedious when affine transformations of the form $A\bar{p} + \vec{t}$ are composed. With homogeneous coordinates, affine transformations become matrices, and composition of transformations is as simple as matrix multiplication. In future sections of the course we exploit this in much more powerful ways.

With homogeneous coordinates, a point $\bar{p}$ is augmented with a 1, to form $\hat{p} = \begin{bmatrix} \bar{p} \\ 1 \end{bmatrix}$

All points $(\alpha\bar{p}, \alpha)$ represent the same point $\bar{p}$ for real $\alpha \neq 0$

Given $\hat{p}$ in homogeneous coordinates, to get $\bar{p}$, we divide $\hat{p}$ by its last component and discard the last component.

> Example:
> The homogeneous points $(2, 4, 2)$ and $(1, 2, 1)$ both represent the Cartesian point $(1, 2)$. It's the orientation of $\hat{p}$ that matters, not its length.

Many transformations become linear in homogeneous coordinates, including affine transformations.

$$\begin{aligned} \begin{bmatrix} q_x \\ q_y \end{bmatrix} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ &= \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} A & \vec{t} \end{bmatrix} \hat{p} \end{aligned}$$

To produce $\hat{q}$ rather than $\bar{q}$, we can add a row to the matrix:

This is linear! Bookkeeping becomes simple under composition.

> *Example:*
> $F_3(F_2(F_1(\bar{p})))$, where $F_i(\bar{p}) = A_i(\bar{p}) + \vec{t_i}$ becomes $M_3M_2M_1\bar{p}$, where $M_i = \begin{bmatrix} A_i & \vec{t_i} \\ \vec{0}^T & 1 \end{bmatrix}$.

With homogeneous coordinates, the following properties of affine transformations become apparent:

1. Affine transformations are associative.

   For affine transformations F_1, F_2, and F_3,

$$(F_3 \circ F_2) \circ F_1 = F_3 \circ (F_2 \circ F_1).$$

2. Affine transformations are *not* commutative.

   For affine transformations F_1 and F_2,

$$F_2 \circ F_1 = F_1 \circ F_2.$$

3.4 Uses and Abuses of Homogeneous Coordinates

Homogeneous coordinates provide a different representation for Cartesian coordinates, and cannot be treated in quite the same way. For example, consider the midpoint between two points $\bar{p}_1 = (1, 1)$ and $\bar{p}_2 = (5, 5)$. The midpoint is $(\bar{p}_1 + \bar{p}_2)/2 = (3, 3)$. We can represent these points in homogeneous coordinates as $\hat{p}_1 = (1, 1, 1)$ and $\hat{p}_2 = (5, 5, 1)$. Directly applying the same computation as above gives the same resulting point: $(3, 3, 1)$.

However, we can *also* represent these points as $\hat{p}' = (2, 2, 2)$ and $\hat{p}' = (5, 5, 1)$. We then have $(\hat{p}' + \hat{p}')/2 = (7/2, 7/2, 3/2)$ which corresponds to the Cartesian point $(7/3, 7/3)$. This is a different point, and illustrates that we cannot blindly apply geometric operations to homogeneous coordinates. The simplest solution is to **always convert homogeneous coordinates to Cartesian coordinates.** That said, there are several important operations that can be performed correctly in terms of homogeneous coordinates, as follows.

3.4.1 Affine transformations:

An important case in the previous section is applying an affine trans- formation to a point in homogeneous coordinates:

$$\begin{aligned} \bar{q} &= F(\bar{p}) = A\bar{p} + \vec{t} \\ \hat{q} &= \hat{A}\hat{p} = (x', y', 1)^T \end{aligned}$$

It is easy to see that this operation is correct, since rescaling $\hat{p}$ does not change the result:

$$\hat{A}(\alpha\hat{p}) = \alpha(\hat{A}\hat{p}) = \alpha\hat{q} = (\alpha x', \alpha y', \alpha)^T$$

Which is the same geometric point as $\hat{q} = (x', y', 1)^T$

3.4.2 Vectors: We can represent a vector $\vec{v} = (x, y)$ in homogeneous coordinates by setting the last element of the vector to be zero: $\hat{v} = (x, y, 0)$. However, when adding a vector to a point, the point must have the third component to be 1.

$$\begin{aligned} \hat{q} &= \hat{p} + \hat{v} \\ (x', y', 1)^T &= (x_p, y_p, 1) + (x, y, 0) \end{aligned}$$

The result is clearly incorrect if the third component of the vector is not 1

Homogeneous coordinates are a representation of points in **projective geometry.**

3.5 Hierarchical Transformations

It is often convenient to model objects as hierarchically connected parts. For example, a robot arm might be made up of an upper arm, forearm, palm, and fingers. Rotating at the shoulder on the upper arm would affect all of the rest of the arm, but rotating the forearm at the elbow would affect the palm and fingers, but not the upper arm. A reasonable hierarchy, then, would have the upper arm at the root, with the forearm as its only child, which in turn connects only to the palm, and the palm would be the parent to all of the fingers.

Each part in the hierarchy can be modeled in its own local coordinates, independent of the other parts. For a robot, a simple square might be used to model each of the upper arm, forearm, and so on. Rigid body transformations are then applied to each part relative to its parent to achieve the proper alignment and pose of the object. For example, the fingers are positioned to be in the appropriate places in the palm coordinates, the fingers and palm together are positioned in forearm coordinates, and the process continues up the hierarchy. Then a transformation applied to upper arm coordinates is also applied to all parts down the hierarchy.

3.6 Transformations in OpenGL

OpenGL manages two 4×4 transformation matrices: the *modelview matrix*, and the *projection matrix*. Whenever you specify geometry (using `glVertex`), the vertices are transformed by the current modelview matrix and then the current projection matrix. Hence, you don't have to perform these transformations yourself. You can modify the entries of these matrices at any time. OpenGL provides several utilities for modifying these matrices. The modelview matrix is normally used to represent geometric transformations of objects; the projection matrix is normally used to store the camera transformation. For now, we'll focus just on the modelview matrix, and discuss the camera transformation later.

To modify the current matrix, first specify which matrix is going to be manipulated: use `glMatrixMode (GL MODE)` to modify the modelview matrix. The modelview matrix can then be initialized to the identity with `glLoadIdentity()`. The matrix can be manipulated by directly filling its values, multiplying it by an arbitrary matrix, or using the functions OpenGL provides to multiply the matrix by specific transformation matrices (`glRotate`, `glTranslate`, and `glScale`). Note that these transformations **right-multiply** the current matrix; this can be confusing since it means that you specify transformations in the reverse of the obvious order.

OpenGL provides a **stacks** to assist with hierarchical transformations. There is one stack for the modelview matrix and one for the projection matrix. OpenGL provides routines for pushing and popping matrices on the stack. The following example draws an upper arm and forearm with shoulder and elbow joints. The current model view matrix is pushed onto the stack and popped at the end of the rendering, so, for example, another arm could be rendered without the

transformations from rendering this arm affecting its model view matrix. Since each OpenGL transformation is applied by multiplying a matrix on the right-hand side of the modelview matrix, the transformations occur in reverse order. Here, the upper arm is translated so that its shoulder position is at the origin, then it is rotated, and finally it is translated so that the shoulder is in its appropriate world-space position. Similarly, the forearm is translated to rotate about its elbow position, and then it is translated so that the elbow matches its position in upper arm coordinates. Below is a program written in OpenGL that implements what has been illustrated above.

OpenGL Program:

```
glPus
hMatr
ix();

glTranslatef(worldShoulderX,  worldShoulderY,
0.0f);
drawShoulde
rJoint();
glRotatef(shoulderRotation, 0.0f, 0.0f,
1.0f);
glTranslatef(-upperArmShoulderX,   -upperArmShoulderY,
0.0f);
drawUpperAr
mShape();

glTranslatef(upperArmElbowX,  upperArmElbowY,
0.0f);

drawElbowJoint();
glRotatef(elbowRotation,     0.0f,     0.0f,     1.0f);
glTranslatef(-forearmElbowX,    -forearmElbowY,    0.0f);
drawForearmShape();

glPopMatrix();
```

4.0 Conclusion

Transformation can change vectors in a variety of ways that are useful. In particular, it can be used to scale, rotate, and shear. Every matrix can be decomposed via SVD into a rotation times a scale times another rotation. An important class of transforms is rigid-body transforms. These are composed only of translations and rotations, so they have no stretching or shrinking of the objects. Such transforms will have a pure rotation.

5.0 Summary

Transformations are used to manipulate objects in three-dimensional space. The three basic classes of transformations are rigid body, conformal and affine transformations.

6.0 Tutor Marked Assignment

1.0 Explain how transformations work

2.0 What is affine transformation?

3.0 Identify and explain various classes of transformations with diagrams

4.0 What is 3D transformation?

5.0 Explain projective transformations.

6.0 Explain the following terms in Transformation

i. Rotation
ii. Scaling
iii. Shearing
iv. Reflection and
v. Orthogonal projections.

7.0 References/Further Reading

1. http://en.wikipedia.org/wiki/Transformation_matrix
2. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
3. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition)*, Prentice Hall, 1997, ISBN: 0135309247

MODULE 2 – Transformations, camera models, rasterization and mapping techniques

UNIT 2: Camera Models

Contents **Pages**

1.0 Introduction to camera models

Most modern cameras use lens to focus light onto the view plane (i.e., the sensory surface). This is done so that one can capture enough light in a sufficiently short period of time that the objects do not move appreciably, and the image is bright enough to show significant detail over a wide range of intensities and contrasts.

In a conventional camera, the view plane contains either photo-reactive chemical; in a digital camera, the view plane contains a charge-coupled device (CCD) array. (Some cameras use a

CMOS-based sensor instead of a CCD). In the human eye, the view plane is a curved surface called the *retina,* and contains a dense array of cells with photo-reactive molecules.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the Thin lens and Pin-hole Camera Models.
2. Understand Projections
3. Understand projections of a triangle.

3.0 Main Content

3.1 Thin Lens Model

Lens models can be quite complex, especially for compound lens found in most cameras. Here we consider perhaps the simplest case, known widely as the thin lens model. In the thin lens model, rays of light emitted from a point travel along paths through the lens, converging at a point behind the lens. The key quantity governing this behaviour is called the *focal length* of the lens. The focal length, $|\mathbf{f}|$, can be defined as distance behind the lens to which rays from an infinitely distant source converge in focus.

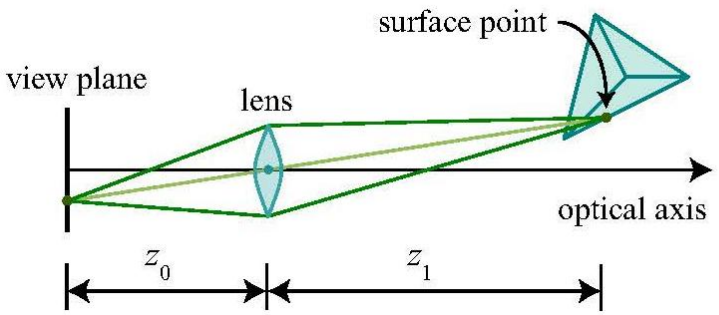


Figure 2.2(a): Thin lens models

More generally, for the thin lens model, if z_1 is the distance from the center of the lens (i.e., the nodal point) to a surface point on an object, then for a focal length $|\mathbf{f}|$, the rays from that surface point will be in focus at a distance z_0 behind the lens center, where z_1 and z_0 satisfy the thin lens equation

$$\frac{1}{|\mathbf{f}|} = \frac{1}{z_0} + \frac{1}{z_1}$$

3.2 Pinhole Camera Model

The **pinhole camera model** describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane of an *ideal* pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light. The model does not include, for example, geometric distortions or blurring of unfocused objects caused by lenses and finite sized apertures. It also does not take into account that most practical cameras have only discrete image coordinates. This means that the pinhole camera model can only be used as a first order approximation of the mapping from a 3D scene to a 2D image. Its validity depends on the quality of the camera and, in general, decreases from the center of the image to the edges as lens distortion effects increase.

Some of the effects that the pinhole camera model does not take into account can be compensated for, for example by applying suitable coordinate transformations on the image coordinates, and others effects are sufficiently small to be neglected if a high quality camera is used. This means that the pinhole camera model often can be used as a reasonable description of how a camera depicts a 3D scene, for example in computer vision and computer graphics.

A **pinhole** camera is an idealization of the thin lens as aperture shrinks to zero.

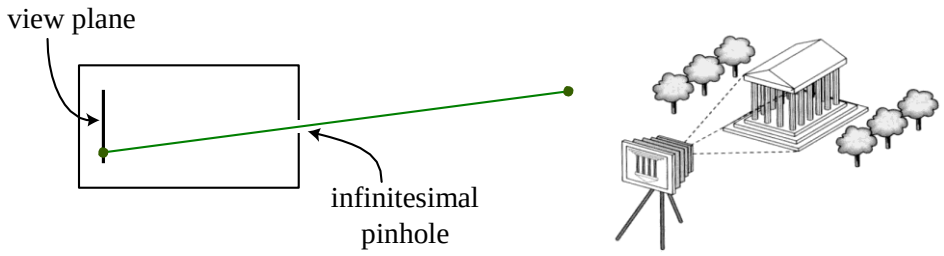
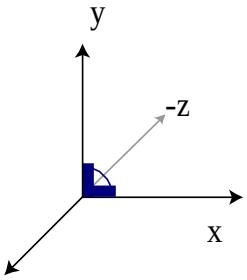


Figure 2.2(b): Light from a point travels along a single straight path through a pinhole onto the view plane. The object is imaged upside-down on the image plane.

We use a right-handed coordinate system for the camera, with the x-axis as the horizontal direction and the y-axis as the vertical direction shown in figure 2.2(c) . This means that the optical axis (gaze direction) is the negative z-axis.



z

Figure 2.2(c): the right-hand coordinate.

The image you would get corresponds to drawing a ray from the eye position and intersecting it with the window. This is equivalent to the pinhole camera model, except that the view plane is in front of the eye instead of behind it, and the image appears right side-up, rather than upside down. (The eye point here replaces the pinhole). To see this, consider tracing rays from scene points through a view plane behind the eye point and one in front of it.

The earliest cameras were room-sized pinhole cameras, called *camera obscura*s. You would walk in the room and see an upside-down projection of the outside world on the far wall. The word *camera* is Latin for "room;" *camera obscura* means "dark room."

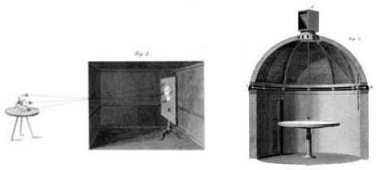


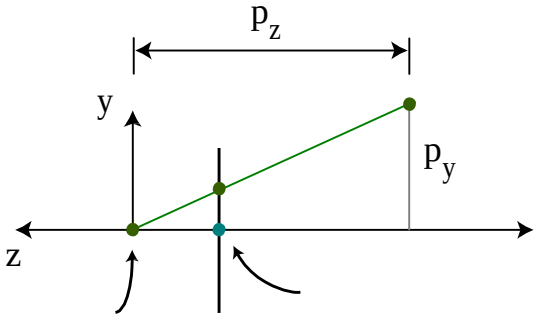
Figure 2.2(d): 18th-century camera obscuras. The camera on the right uses a mirror in the roof to project images of the world onto the table, and viewers may rotate the mirror.

3.3 Camera Projections

Consider a point $\bar{p}$ in 3D space oriented with the camera at the origin, which we want to project onto the view plane. To project p_y to y, we can use similar triangles to get $y = \frac{f}{p_z} p_y$. This is **perspective projection**.

Note that $f < 0$, and the focal length is $|f|$.

In perspective projection, distant objects appear smaller than near objects:



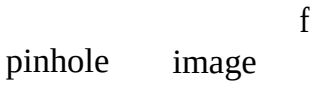


Figure 2.2 (e): Perspective Projection

3.4 Orthographic Projection

For objects sufficiently far away, rays are nearly parallel, and variation in p_z is insignificant.

Figure 2.2(f): Here, the baseball players appear to be about the same height in pixels, even though the batter is about 60 feet away from the pitcher. Although this is an example of perspective projection, the camera is so far from the players (relative to the camera focal length) that they appear to be roughly the same size.

In the limit, $y = \alpha p_y$ for some real scalar α. This is **orthographic projection**:

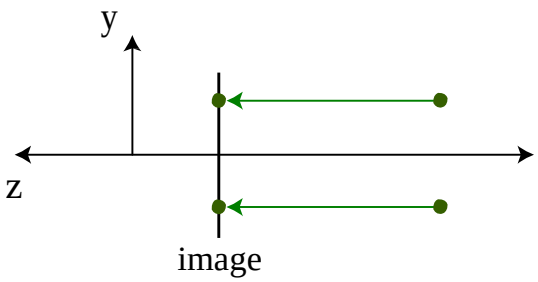
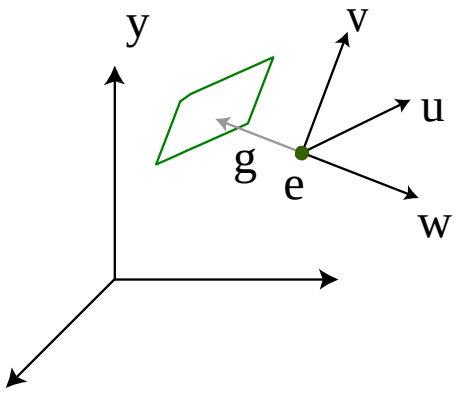


Figure 2.2(g): orthographic projection

3.5 Camera Position and Orientation

Assume camera coordinates have their origin at the "eye" (pinhole) of the camera, $\bar{e}$.



X

Z

Figure 2.2(h): camera positioning

Let g be the gaze direction, so a vector perpendicular to the view plane (parallel to the camera z-axis) is

$$\vec{w} = \frac{-\vec{g}}{\|\vec{g}\|}$$

We need two more orthogonal vectors u and v to specify a camera coordinate frame, with $\vec{u}$ and $\vec{v}$ parallel to the view plane. It may be unclear how to choose them directly. However, we can instead specify an "up" direction. Of course this up direction will not be perpendicular to the gaze direction.

Let $\vec{t}$ be the "up" direction (e.g., toward the sky so $\vec{t} = (0, 1, 0)$). Then, we want v to be the closest vector in the view-plane to $\vec{t}$. This is really just the projection of $\vec{t}$ onto the view plane. Therefore, u must be perpendicular to $\vec{v}$ and $\vec{w}$. In fact, with these definitions it is easy to show that u must also be perpendicular to $\vec{t}$, so one way to compute u and v from $\vec{t}$ and g is as follows:

$$\vec{u} = \frac{\vec{t} \times \vec{w}}{\|\vec{t} \times \vec{w}\|} \qquad \vec{v} = \vec{w} \times \vec{u}$$

Of course, we could have used many different "up" directions, so long as $t \times w = 0$.

Using these three basis vectors, we can define a **camera coordinate system**, in which 3D points are represented with respect to the camera's position and orientation. The camera coordinate system has its origin at the eye point e and has basis vectors u, v, and w, corresponding to the x, y, and z axes in the camera's local coordinate system. This explains why we chose w to point away from the image plane: the right-handed coordinate system requires that z (and, hence, w) point away from the image plane.

Now that we know how to represent the camera coordinate frame within the world coordinate frame we and need to explicitly formulate the rigid transformation from world to camera

coordinates. With this transformation and its inverse, we can easily express points either in world coordinates or camera coordinates (both of which are necessary).

To get an understanding of the transformation, it might be helpful to remember the mapping from points in camera coordinates to points in world coordinates. For example, we have the following correspondences between world coordinates and camera coordinates: Using such correspondences

Camera coordinates (x_c, y_c, z_c)	World coordinates (x, y, z)
$(0,0,0)$	$\bar{e}$
$(0,0,f)$	$\bar{e} + f\vec{w}$
$(0,1,0)$	$\bar{e} + \vec{v}$
$(0,1,f)$	$\bar{e} + \vec{v} + f\vec{w}$

Table 2.2: world coordinates and transformation coordinates

It is not hard to show that for a general point expressed in camera coordinates as the $\bar{p}^c = (x_c, y_c, z_c)$, corresponding point in world coordinates is given by

$$\begin{aligned} \bar{p}^w &= \bar{e} + x_c\vec{u} + y_c\vec{v} + z_c\vec{w} \\ &= \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix}\bar{p}^c + \bar{e} \\ &= M_{cw}\bar{p}^c + \bar{e}. \end{aligned}$$

Where $$M_{cw} = \left[\begin{array}{ccc} \vec{u} & \vec{v} & \vec{w} \end{array} \right] = \left[\begin{array}{ccc} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{array} \right]$$

Note: We can define the same transformation for points in homogeneous coordinates:

$$\hat{M}_{cw} = \left[\begin{array}{cc} M_{cw} & \bar{e} \\ \vec{0}^T & 1 \end{array} \right].$$

Now, we also need to find the inverse transformation, i.e., from world to camera coordinates. Toward this end, note that the matrix M_{cw} is orthonormal. To see this, note that vectors u, v and, w are all of unit length, and they are perpendicular to one another. You can also verify this by computing $M^T M_{cw}$. Because M_{cw} is orthonormal, we can express the inverse transformation (from camera coordinates to world coordinates) as

$$\begin{aligned} \bar{p}^c &= M_{cw}^T(\bar{p}^w - \bar{e}) \\ &= M_{wc}\bar{p}^w - \bar{d}\,, \end{aligned}$$

where $M_{wc} = M_{cw}^T = \left[\begin{array}{c} \vec{u}^T \\ \vec{v}^T \\ \vec{w}^T \end{array} \right]$. (why?), and $\bar{d} = M_{cw}^T \bar{e}$.

In homogeneous coordinates, $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, where

$$\begin{aligned} \hat{M}_v &= \left[\begin{array}{cc} M_{wc} & -M_{wc}\bar{e} \\ \vec{0}^T & 1 \end{array} \right] \\ &= \left[\begin{array}{cc} M_{wc} & \vec{0} \\ \vec{0}^T & 1 \end{array} \right] \left[\begin{array}{cc} I & -\bar{e} \\ \vec{0}^T & 1 \end{array} \right]. \end{aligned}$$

This transformation takes a point from world to camera-centered coordinates.

3.6 Perspective Projection

From above, we found that the form of the perspective projection using the idea of similar triangles. Here we consider a complementary algebraic formulation. To begin, we are given

1. A point $\bar{p}^c$ in camera coordinates (***UVW*** space),
2. Center of projection (eye or pinhole) at the origin in camera coordinates,
3. Image plane perpendicular to the z-axis, through the point $(0, 0, f)$, with $f < 0$, and
4. line of sight is in the direction of the negative z-axis (in camera coordinates),

We can find the intersection of the ray from the pinhole to $\bar{p}^c$ with the view plane.

The ray from the pinhole to $\bar{p}^c$ is $\bar{r}(\lambda) = \lambda(\bar{p}^c - \bar{0})$.

The image plane has normal $(0, 0, 1) = \vec{n}$ and contains the point $(0, 0, f) = \bar{f}$. So a point $\bar{x}^c$ is on the plane when $(\bar{x}^c - \bar{f}) \cdot \vec{n} = 0$. If $\bar{x}^c = (x^c, y^c, z^c)$, then the plane satisfies $z^c - f = 0$.

To find the intersection of the plane $z^c = f$ and ray $\vec{r}(\lambda) = \lambda\bar{p}^c$, substitute $\vec{r}$ into the plane equation. With $\bar{p}^c = (p_x^c, p_y^c, p_z^c)$, we have $\lambda p_z^c = f$, so $\lambda^* = f/p_z^c$, and the intersection is

$$\vec{r}(\lambda^*) = \left(f\frac{p_x^c}{p_z^c}, f\frac{p_y^c}{p_z^c}, f \right) = f\left(\frac{p_x^c}{p_z^c}, \frac{p_y^c}{p_z^c}, 1 \right) \equiv \bar{x}^*.$$

The first two coordinates of this intersection $\bar{x}^*$ determine the image coordinates.

2D points in the image plane can therefore be written as

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \frac{f}{p_z^c} \begin{bmatrix} p_x^c \\ p_y^c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \frac{f}{p_z^c} \bar{p}^c.$$

The mapping from $\bar{p}^c$ to $(x^*, y^*, 1)$ is called **perspective projection**.

Two important properties of perspective projection are:

1. Perspective projection preserves linearity. In other words, the projection of a 3D line is a line in 2D. This means that we can render a 3D line segment by projecting the endpoints to 2D, and then draw a line between these points in 2D.
2. Perspective projection does not preserve parallelism: two parallel lines in 3D do not necessarily project to parallel lines in 2D. When the projected lines intersect, the intersection is called a **vanishing point**, since it corresponds to a point infinitely far away.

Self Assessment Exercise:

1. When do parallel lines project to parallel lines and when do they not?

The discovery of linear perspective, including vanishing points, formed a cornerstone of Western painting beginning at the Renaissance. On the other hand, defying realistic perspective was a key feature of Modernist painting. To see that linearity is preserved, consider that rays from points on a line in 3D through a pinhole all lie on a plane, and the intersection of a plane and the image plane is a line. This implies that drawing of polygons only requires projecting the vertices to the image plane and draw lines between them.

3.7 Homogeneous Perspective

The mapping of $\bar{p}^c = (p_x^c, p_y^c, p_z^c)$ to $\bar{x}^* = \frac{f}{p_z^c}(p_x^c, p_y^c, p_z^c)$ is just a form of scaling transformation. However, the magnitude of the scaling depends on the depth p_z^c. So it's not linear.

Fortunately, the transformation can be expressed linearly (ie as a matrix) in homogeneous coordinates. To see this, remember that $\hat{p} = (\bar{p}, 1) = \alpha(\bar{p}, 1)$ in homogeneous coordinates. Using this property of homogeneous coordinates we can write $\bar{x}^*$ as

$$\hat{x}^* = \left(p_x^c, p_y^c, p_z^c, \frac{p_z^c}{f} \right).$$

As usual with homogeneous coordinates, when you scale the homogeneous vector by the inverse of the last element, when you get in the first three elements is precisely the perspective projection. Accordingly, we can express $\hat{x}^*$ as a linear transformation of $\hat{p}^c$:

$$\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c \equiv \hat{M}_p \hat{p}^c.$$

Try multiplying this out to convince yourself that this all works.
Finally, $\hat{M}_p$ is called the homogeneous perspective matrix, and since $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, we have $\hat{x}^* = \hat{M}_p \hat{M}_{wc} \hat{p}^w$.

3.8 Pseudodepth

After dividing by its last element, $\hat{x}^*$ has its first two elements as image plane coordinates, and its third element is f. We would like to be able to alter the homogeneous perspective matrix $\hat{M}_p$ so that the third element of $\frac{p_z^c}{f}\hat{x}^*$ encodes depth while keeping the transformation linear.

Idea: Let $\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c$, so $z^* = \frac{f}{p_z^c}(ap_z^c + b)$.

What should a and b be? We would like to have the following two constraints:

$$z^* = \begin{cases} -1 & \text{when } p_z^c = f \\ 1 & \text{when } p_z^c = F \end{cases},$$

where f gives us the position of the **near plane**, and F gives us the z coordinate of the **far plane**.

So $-1 = af + b$ and $1 = af + b\frac{f}{F}$. Then $2 = b\frac{f}{F} - b = b\left(\frac{f}{F} - 1\right)$, and we can find

$$b = \frac{2F}{f - F}.$$

Substituting this value for b back in, we get $-1 = af + \frac{2F}{f-F}$, and we can solve for a:

$$\begin{aligned} a &= -\frac{1}{f}\left(\frac{2F}{f-F} + 1\right) \\ &= -\frac{1}{f}\left(\frac{2F}{f-F} + \frac{f-F}{f-F}\right) \\ &= -\frac{1}{f}\left(\frac{f+F}{f-F}\right). \end{aligned}$$

These values of a and b give us a function $z^*(p_z^c)$ that increases monotonically as p_z^c decreases (since p_z^c is negative for objects in front of the camera). Hence, z^* can be used to sort points by depth.

Why did we choose these values for a and b? Mathematically, the specific choices do not matter, but they are convenient for implementation. These are also the values that OpenGL uses.

What is the meaning of the near and far planes? Again, for convenience of implementation, we will say that only objects between the near and far planes are visible. Objects in front of the near plane are behind the camera, and objects behind the far plane are too far away to be visible. Of course, this is only a loose approximation to the real geometry of the world, but it is very convenient for implementation. The range of values between the near and far plane has a number of subtle implications for rendering in practice. For example, if you set the near and far plane to be very far apart in OpenGL, then Z-buffering (discussed later in the course) will be very inaccurate due to numerical precision problems. On the other hand, moving them too close will make distant objects disappear. However, these issues will generally not affect rendering simple scenes. (For homework assignments, we will usually provide some code that avoids these problems).

3.9 Projecting a Triangle

Let's review the steps necessary to project a triangle from object space to the image plane.

1. A triangle is given as three vertices in an object-based coordinate frame: $\bar{p}_1^o, \bar{p}_2^o, \bar{p}_3^o$.

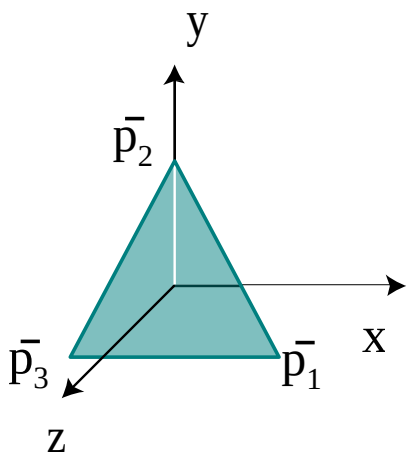
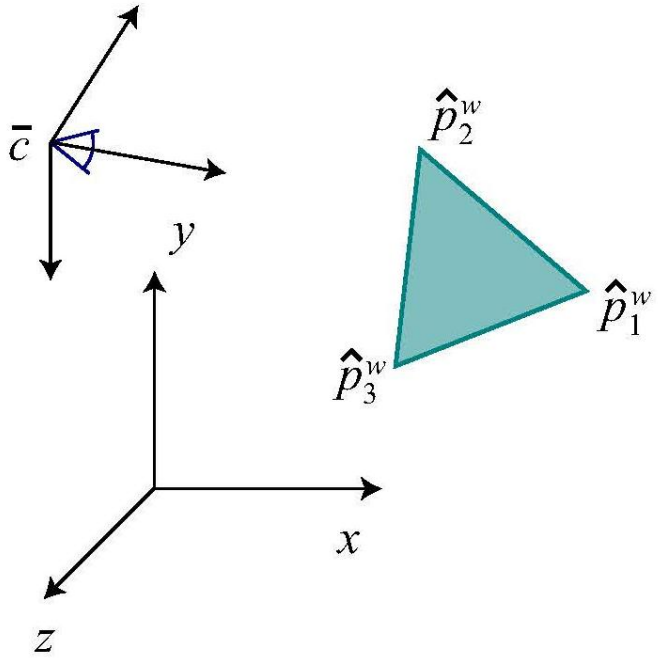


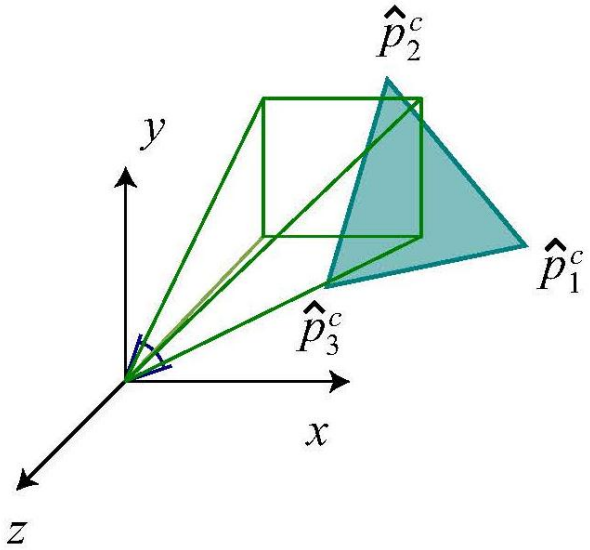
Figure 2.2 (i): A triangle in object coordinates.

2. Transform to world coordinates based on the object's transformation: $\hat{p}_1^w$, $\hat{p}_2^w$, $\hat{p}_3^w$, where $\hat{p}_i^w = \hat{M}_{ow}\hat{p}_i^o$.



The triangle projected to world coordinates, with a camera at $\bar{c}$.

3. Transform from world to camera coordinates: $\hat{p}_i^c = \hat{M}_{wc}\hat{p}_i^w$.



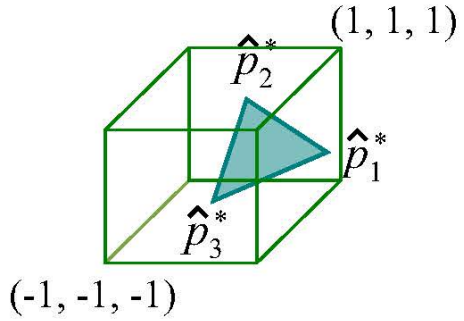
The triangle projected from world to camera coordinates.

4. Homogeneous perspective transformation: $\hat{x}_i^* = \hat{M}_p \hat{p}_i^c$, where

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix}, \text{ so } \hat{x}_i^* = \begin{bmatrix} p_x^c \\ p_y^c \\ ap_z^c + b \\ \frac{p_z^c}{f} \end{bmatrix}.$$

5. Divide by the last component:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = f \begin{bmatrix} \frac{p_x^c}{p_z^c} \\ \frac{p_y^c}{p_z^c} \\ \frac{ap_z^c + b}{p_z^c} \end{bmatrix}.$$



The triangle in normalized device coordinates after perspective division.

Now (x^*, y^*) is an image plane coordinate, and z^* is pseudodepth for each vertex of the triangle.

3.10 Camera Projections in OpenGL

OpenGL's modelview matrix is used to transform a point from object or world space to camera space. In addition to this, a *projection matrix* is provided to perform the homogeneous perspective transformation from camera coordinates to *clip coordinates* before performing perspective division. After selecting the projection matrix, the `glFrustum` function is used to specify a viewing volume, assuming the camera is at the origin:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(left, right, bottom, top, near, far);
```

For orthographic projection, `glOrtho` can be used instead:

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

The GLU library provides a function to simplify specifying a perspective projection viewing frustum:

```
gluPerspective(fieldOfView, aspectRatio, near, far);
```

The field of view is specified in degrees about the x-axis, so it gives the vertical visible angle. The aspect ratio should usually be the viewport width over its height, to determine the horizontal field.

4.0 Conclusion

Some of the effects that the pinhole camera model does not take into account can be compensated for, by applying suitable coordinate transformations on the image coordinates, and others effects are sufficiently small to be neglected if a high quality camera is used. This means that the pinhole camera model often can be used as a reasonable description of how a camera depicts a 3D scene, in computer vision and computer graphics.

5.0 Summary

Most modern cameras use a lens to focus light onto the view plane. This is done so that one can capture enough light in a sufficiently short period of time that the objects do not move appreciably, and the image is bright enough to show significant detail over a wide range of intensities and contrasts. Lens models can be quite complex, especially for compound lens found in most cameras. This means that the pinhole camera model can only be used as a first order approximation of the mapping from a 3D scene to a 2D image.

6.0 Tutor Marked Assignment

1. What do you understand by Thin lens Camera Model?
2. Identify Application areas of the pinhole camera model.
3. Explain camera projections.
4. Describe perspective projections.

7.0 References/Further Reading

1. David A. Forsyth and Jean Ponce (2003). *Computer Vision, A Modern Approach*. Prentice Hall.
2. Richard Hartley and Andrew Zisserman (2003). *Multiple View Geometry in computer vision*. Cambridge University Press. ISBN-13: 978-0521540513
3. Bernd Jähne (1997). *Practical Handbook on Image Processing for Scientific Applications*. CRC Press. ISBN-13: 978-0849319006
4. Linda G. Shapiro and George C. Stockman (2001). *Computer Vision*. Prentice Hall. ISBN-13: 978-0130307965
5. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
6. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition)*, Prentice Hall, 1997, ISBN: 0135309247

MODULE 2 – Transformations, camera models, rasterization and mapping techniques

UNIT 3: Rasterization and The z-Buffer

Contents **Pages**

1.0 Introduction to Raster Graphics

In computer graphics, a **raster graphics** image, or **bitmap**, is a data structure representing a generally rectangular grid of pixels, or points of colour, viewable via a monitor, paper, or other display medium. Raster images are stored in image files with varying formats. A bitmap corresponds bit-for-bit with an image displayed on a screen, generally in the same format used for storage in the display's video memory, or maybe as a device-independent bitmap. A bitmap is technically characterized by the width and height of the image in pixels and by the number of bits per pixel (a colour depth, which determines the number of colours it can represent).

The printing and prepress industries know raster graphics as **contones** (from "continuous tones") and refer to vector graphics as "line work".

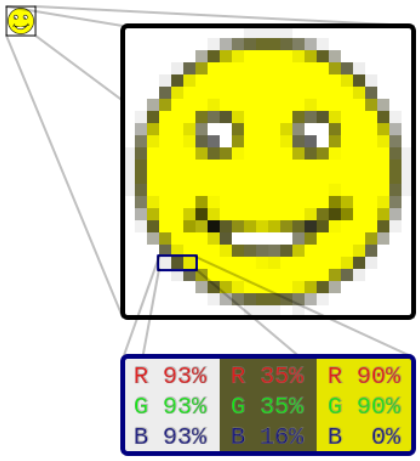


Figure 2.3(a): a bitmap smiley face image.

When this image is enlarged, individual pixels appear as squares. Zooming in further, they can be analyzed, with their colours constructed by adding the values for red, green and blue. Raster graphics are resolution dependent. They cannot scale up to an arbitrary resolution without loss of apparent quality. This property contrasts with the capabilities of vector graphics, which easily scale up to the quality of the device rendering them. Raster graphics deal more practically than vector graphics with photographs and photo-realistic images, while vector graphics often serve better for typesetting or for graphic design. Modern computer-monitors typically display about 72 to 130 pixels per inch (PPI), and some modern consumer printers can resolve 2400 dots per inch (DPI) or more; determining the most appropriate image resolution for a given printer-resolution can pose difficulties, since printed output may have a greater level of detail than a viewer can discern on a monitor. Typically, a resolution of 150 to 300 pixel per inch works well for 4-colour process Cyan Magenta Yellow Black (CMYK) printing.

However, for printing technologies that perform colour mixing through dithering rather than through overprinting (virtually all home and office, inkjet and laser printers included), printer DPI and image PPI have a very different meaning, and this can be misleading. Because, through the dithering process, the printer builds a single image pixel out of several printer dots to increase colour depth, the printer's DPI setting must be set far higher than the desired PPI to ensure sufficient colour depth without sacrificing image resolution. Thus, for instance, printing an image at 250 PPI may actually require a printer setting of 1200 DPI.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand how images are displayed.
2. Understand how imaging systems are organized (raster graphic Systems)
3. Understand the Z-buffering and its uses.

3.0 Main Content

3.1 Rasterization

Rasterization is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format. The term "*rasterization*" can in general be applied to any process by which vector information can be converted into a raster format.

In normal usage, the term refers to the popular rendering algorithm for displaying three-dimensional shapes on a computer. Rasterization is currently the most popular technique for producing real-time 3D computer graphics. Real-time applications need to respond immediately to user input, and generally need to produce frame rates of at least 24 frames per second to achieve smooth animation. Compared with other rendering techniques such as ray tracing, rasterization is extremely fast. However, rasterization is simply the process of computing the mapping from scene geometry to pixels and does not prescribe a particular way to compute the colour of those pixels. Shading, including programmable shading, may be based on physical light transport, or artistic intent.

The process of rasterizing 3D models onto a 2D plane for display on a computer screen is often carried out by fixed function hardware within the graphics pipeline. This is because there is no motivation for modifying the techniques for rasterization used at render time and a special-purpose system allows for high efficiency.

3.2 Z-buffering

Z-buffering is the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. The painter's algorithm is another common solution which, though less efficient, can also handle non-opaque scene elements. Z-buffering is also known as **depth buffering**.

When an object is rendered by a 3D graphics card, the depth of a generated pixel (z coordinate) is stored in a buffer (the **z-buffer** or **depth buffer**). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the graphics card compares the two depths and chooses the one closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one. In the end, the z-buffer will allow the graphics card to correctly reproduce the usual depth perception: a close object hides a farther one. This is called **z-culling**.

The granularity of a z-buffer has a great influence on the scene quality: a 16-bit z-buffer can result in artifacts (called "z-fighting") when two objects are very close to each other. A 24-bit or 32-bit z-buffer behaves much better, although the problem cannot be entirely eliminated without additional algorithms. An 8-bit z-buffer is almost never used since it has too little precision.

3.3 Uses for Z-buffering

Z-buffer data in the area of video editing permits one to combine 2D video elements in 3D space, permitting virtual sets, "ghostly passing through wall" effects, and complex effects like mapping of video on surfaces. An application for Maya, called IPR, permits one to perform post-rendering texturing on objects, utilizing multiple buffers like z-buffers, alpha, object id, UV coordinates and any data deemed as useful to the post-production process, saving time otherwise wasted in re-rendering of the video.

Z-buffer data obtained from rendering a surface from a light's POV permits the creation of shadows in a scanline renderer, by projecting the z-buffer data onto the ground and affected surfaces below the object. This is the same process used in non-ray tracing modes by the free and open sourced 3D application Blender.

3.4 Z-buffering Algorithm

Given: A list of polygons {P1, P2,.....Pn}
Output: A COLOUR array, which displays the intensity of the visible polygon surfaces.
Initialize:

```
note : z-depth and z-buffer(x,y) is positive........
 z-buffer(x,y)=max depth; and
 COLOUR(x,y)=background colour.
```

Begin:

```
for(each polygon P in the polygon list) do{
  for(each pixel(x,y) that intersects P) do{
      Calculate z-depth of P at (x,y)
       If (z-depth < z-buffer[x,y]) then{
            z-buffer[x,y]=z-depth;
            COLOUR(x,y)=Intensity of P at(x,y);
       }
   }
}
display COLOUR array.
```

4.0 Conclusion

Interactive raster graphics systems typically employ several processing units. In addition to the central processing unit, or CPU, a special-purpose processor, called the video controller or display controller, is used to control the operation of the display device.

5.0 Summary

Raster graphics are resolution dependent. They cannot scale up to an arbitrary resolution without loss of apparent quality. Raster graphics deal more practically than vector graphics with photographs and photo-realistic images, while vector graphics often serve better for typesetting or for graphic design. Rasterization is the task of taking an image described in a vector graphics format and converting it into a raster image for output on a video display or printer, or for storage in a bitmap file format. Z-buffering is the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden.

6.0 Tutor Marked Assignment

1. What do you understand by Rasterization?
2. Write a rasterization algorithm.
3. Explain Z-buffering and state its uses.
4. Compare the Z-buffering algorithm with the painter's algorithm
5. Write short notes on the following display devices
    1. Cathode ray tubes
    2. Liquid crystal display
    3. Plasma panels
    4. Light emitting diodes
    5. Thin-film electroluminescent displays.

7.0 References/Further Reading

1. 3D computer graphics – Compiled by H. Hees.
2. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
3. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247

MODULE 2 – Transformations, Camera models, Rasterization and Mapping techniques

UNIT 4: Mapping methods

Contents **Pages**

1.0 Introduction to Mapping Methods

One of the most powerful uses of discrete data is for surface rendering. There are three major techniques:

1. Texture mapping
2. Bump mapping
3. Environment mapping

Texture maps give detail by painting patterns onto smooth surfaces, while **bump maps** distort the normal vectors during the shading process to make the surface appear to have small variations in shape, such as the bumps on a real orange. **Reflection maps**, or **environment maps**, allow us to create images that have the appearance of reflected materials without having to trace reflected rays. In this technique, an image of the environment is painted onto the surface as that surface is being rendered.

The three methods have much in common. All of them alter the shading of individual fragments as part of fragment processing. They rely on the map being stored as a one-, two-, or three-dimensional digital image. They also keep the geometric complexity low while creating the illusion of complex geometry. However, they are subject to aliasing errors.

In virtual reality, visualization simulations, and interactive games, real-time performance is required. Hardware support for texture mapping in modern systems allows the detail to be added, without significantly degrading the rendering time. However, in terms of the standard pipeline, there are significant differences among the three techniques. Standard texture mapping is supported by the basic OpenGL pipeline and makes use of both the geometric and pixel pipelines. Environment maps are a special case of standard texture mapping but can be altered to create a variety of new effects if we can alter fragment processing. Bump mapping requires us to process each fragment independently, something we can do with a fragment shader.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the various mapping methods
2. Understand the Mapping algorithms.
3. Differentiate between the mapping techniques.

3.0 Main Content

3.1 Texture mapping

Texture mapping is a method for adding detail, surface texture (a bitmap or raster image), or colour to a computer-generated graphic or 3D model. A **texture map** is applied (mapped) to the surface of a shape or polygon. This process is akin to applying patterned paper to a plain white box. Every vertex in a polygon is assigned a texture coordinate (which in the 2d case is also known as a UV coordinate) either via explicit assignment or by procedural definition. Image sampling locations are then interpolated across the face of a polygon to produce a visual result that seems to have more richness than could otherwise be achieved with a limited number of polygons.

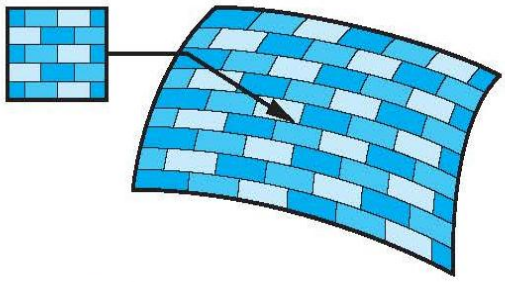


Figure 2.4(a): Texture mapping a pattern to a surface (Ed Angel, 1991)

Texture mapping uses an image (or texture) to influence the colour of a fragment. Textures can be specified using a fixed pattern, such as the regular patterns often used to fill polygons; by a procedural texture-generation method; or through a digitized image. We can characterize the resulting image as the mapping of a texture to a surface as part of the rendering of the surface.

Textures are patterns. They can range from regular patterns, such as stripes and checkerboards, to the complex patterns that characterize natural materials. In the real world, we can distinguish among objects of similar size and shape by their textures. Thus, if we want to create detailed virtual objects, we can extend our present capabilities by mapping a texture to the objects that we create.

Textures can be one, two, three, or four dimensional. For example, a one-dimensional texture might be used to create a pattern for colouring a curve. A three dimensional texture might describe a solid block of material from which an object could sculptured. The use of surfaces is so important in computer graphics, mapping two-dimensional textures to surfaces is by far the

most common use of texture. However, the processes by which these entities could be mapped is much the same regardless of the dimensionality of the texture.

3.2 Two-Dimensional Texture Mapping

Although there are multiple approaches to texture mapping, all require a sequence of steps that involve mappings among three or four different coordinate systems. Methods differ according to the types of surfaces being considered and the type of rendering architecture available. In most applications, textures start out as two-dimensional images of the sorts. Thus, they might be formed by application programs or scanned in from a photograph, but, regardless of their origin, they are eventually brought into processor memory as arrays. The elements of these arrays are called **texels**, or texture elements, rather than pixels to emphasize how they will be used. We prefer to think of this array as a continuous rectangular two-dimensional texture pattern $T(s, t)$. The independent variables s and t are known as **texture coordinates**. With no loss of generality, we can scale our texture coordinates to vary over the interval [0,1].

A **texture map** associates a texel with each point on a geometric object that is itself mapped to screen coordinates for display. If the object is represented in homogeneous or (x, y, z, w) coordinates, then there are functions such that

$$x = x(s, t),$$
$$y = y(s, t),$$
$$z = z(s, t),$$
$$w = w(s, t).$$

One of the difficulties confronted is that although these functions exist conceptually, finding them may not be possible in practice. In addition, there are concerns about the inverse problem: Having been given a point (x, y, z) or (x, y, z, w) on an object, how do we find the corresponding texture coordinates, or equivalently, how do we find the "inverse" functions to use to find the texel $T(s, t)$?

$$s = s(x, y, z , w),$$
$$t = t(x, y, z , w)$$

If we define the geometric object using parametric (u, v) surfaces, there is an additional mapping function that gives object coordinate values, (x, y, z) or (x, y, z , w) in terms of u and v. Although this mapping is known for simple surfaces, such as spheres and triangles, we also need the mapping from parametric coordinates (u, v) to texture coordinates and sometimes the inverse mapping from texture coordinates to parametric coordinates.

The projection process that take us from object coordinates to screen coordinates has to be considered, going through eye coordinates, clip coordinates, and window coordinates along the way. We can abstract this process through a function that takes a texture coordinate pair (s, t) and tells us where in the colour buffer the corresponding value of $T(s, t)$ will make its contribution to

the final image. Thus, there is a mapping of the form into coordinates, where (x_s , y_s) is a location in the colour buffer.

$$x_s=x_s(s, t),$$
$$y_s=y_s(s, t)$$

Depending on the algorithm and the rendering architecture, the function that takes us from a pixel in the colour buffer to the texel that makes a contribution to the colour of that pixel might also be required. One way to think about texture mapping is in terms of two concurrent mappings: the first from texture coordinates to object coordinates, and the second from parametric coordinates to object coordinates. A third mapping takes us from object coordinates to screen coordinates.

Conceptually, the texture-mapping process is simple. A small area of the texture pattern maps to the area of the geometric surface, corresponding to a pixel in the final image. Assume that the values of *T* are RGB colour values, these values can be used either to modify the colour of the surface that might have been determined by a lighting model or to assign a colour to the surface based on only the texture value. This colour assignment is carried out as part of the assignment of fragment colours. On closer examination, we face a number of difficulties.

First, the map from texture coordinates to object coordinates must be determined. A two-dimensional texture usually is defined over a rectangular region in texture space. The mapping from this rectangle to an arbitrary region in three-dimensional space may be a complex function or may have undesirable properties. For example, if we wish to map a rectangle to a sphere, we cannot do so without distortion of shapes and distances.

Second, owing to the nature of the rendering process, which works on a pixel-by-pixel basis, we are more interested in the inverse map from screen coordinates to texture coordinates. It is when we are determining the shade of a pixel that we must determine what point in the texture image to use—a calculation that requires us to go from screen coordinates to texture coordinates.

Third, because each pixel corresponds to a small rectangle on the display, we are interested in mapping not points to points, but rather areas to areas. Here again is a potential aliasing problem that we must treat carefully if we are to avoid artifacts, such as wavy sinusoidal or moir´e patterns.

3.3 Bump Mapping

Bump mapping is a texture-mapping technique that can give the appearance of great complexity in an image without increasing the geometric complexity. Unlike simple texture mapping, bump mapping will show changes in shading as the light source or object moves, making the object appear to have variations in surface smoothness. The technique of **bump mapping** varies the

apparent shape of the surface by perturbing the normal vectors as the surface is rendered; the colours that are generated by shading then show a variation in the surface properties. Unlike techniques such as environment mapping that can be implemented without programmable shaders, bump mapping cannot be done in real time without them.

Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object. This is achieved by perturbing the surface normals of the object and using the perturbed normal during lighting calculations. The result is an apparently bumpy surface rather than a smooth surface although the surface of the underlying object is not actually changed.

Bump mapping is limited in that it does not actually modify the shape of the underlying object. On the left, a mathematical function defining a bump map simulates a crumbling surface on a sphere, but the object's outline and shadow remain those of a perfect sphere. On the right, the same function is used to modify the surface of a sphere by generating an isosurface. This actually models a sphere with a bumpy surface with the result that both its outline and its shadow are rendered realistically.

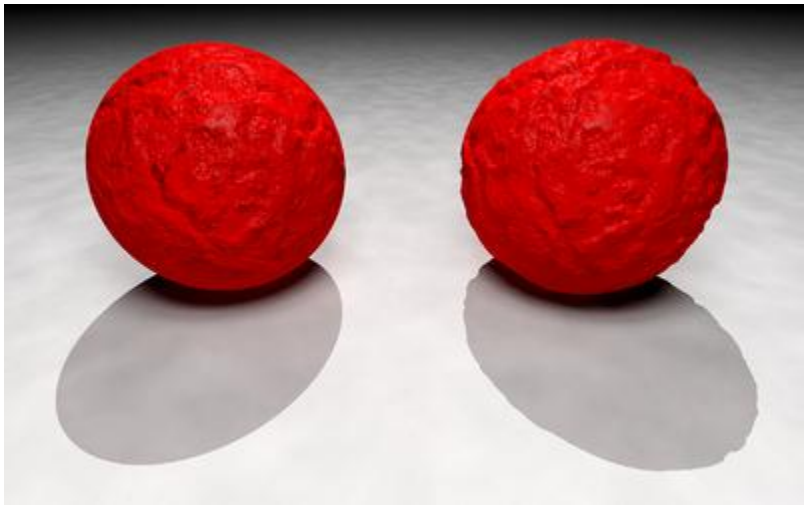


Figure 2.4(b): A bump map on a sphere (wikipedia)

Bump mapping is a technique in computer graphics to make a rendered surface look more realistic by simulating small displacements of the surface. However, unlike traditional displacement mapping, the surface geometry is not modified. Instead only the surface normal is modified *as if* the surface had been displaced. The modified surface normal is then used for lighting calculations as usual, typically using the Phong reflection model or similar, giving the appearance of detail instead of a smooth surface. Bump mapping is much faster and consumes fewer resources for the same level of detail compared to displacement mapping because the geometry remains unchanged.

There are primarily two methods to perform bump mapping. The first uses a height map for simulating the surface displacement yielding the modified normal. This is the method invented by Blinn and is usually what is referred to as bump mapping unless specified. The steps of this method are summarized as follows.

Before lighting a calculation is performed for each visible point (or pixel) on the object's surface:

1. Look up the height in the heightmap that corresponds to the position on the surface.
2. Calculate the surface normal of the heightmap, typically using the finite difference method.
3. Combine the surface normal from step two with the true ("geometric") surface normal so that the combined normal points in a new direction.
4. Calculate the interaction of the new "bumpy" surface with lights in the scene using, for example, the Phong reflection model.

The result is a surface that appears to have real depth. The algorithm also ensures that the surface appearance changes as lights in the scene are moved around.

The other method is to specify a normal map which contains the modified normal for each point on the surface directly. Since the normal is specified directly instead of derived from a height map, this method usually leads to more predictable results. This makes it easier for artists to work with, making it the most common method of bump mapping today. There are also extensions which modifies other surface features in addition to increase in the sense of depth. Parallax mapping is one such extension.

The primary limitation with bump mapping is that it perturbs only the surface normals without changing the underlying surface itself. Silhouettes and shadows therefore remain unaffected, which is especially noticeable for larger simulated displacements. This limitation can be overcome by techniques including the displacement mapping where bumps are actually applied to the surface or using an isosurface.

3.4 Environment Mapping

3.4.1 Introduction to Environment mapping

Environment mapping is a scheme that improves on the mapping techniques of chrome and refraction mapping. It is useful because of the fact that in a reflective environment, ray tracing can be very expensive. Environment mapping gives these reflections more cheaply with little loss of accuracy. Its computation is independent of the level of detail of the surroundings. In environment mapping, the object is surrounded by a closed three dimensional surface onto which the environment is projected. Reflected rays are traced from the object, hit the surface and then

index onto the map. It is essentially the same as chrome mapping except that the map consists of an image of the environment as seen from the center of the space to be environment mapped.

3.4.2 Factors affecting environment mapping

In all environment mapping techniques, the accuracy depends on the object being positioned in the center of the surface and that objects in the environment are distant from the objects receiving the environment map. As the object becomes larger with respect to the environment, or the distance from the map center increases, the geometric distortion increases. This is because of the fact that, the environment map is created as a projection from a single point at the center of the surface.

Geometric distortion can be reduced by ray tracing those objects in the environment that are too close to the reflective object. If the reflective object is to complex, it may have to be ray traced which environment mapping cannot handle.

3.4.3 Techniques of environment mapping

The first use of environment mapping was developed by Jim Blinn and Newell (1976). In this case, the object is deemed to be positioned at the center of a large sphere, onto the interior, of which the environment is projected. The mapping used a latitude-longitude map indexed by the reflected ray, similar to chrome mapping. The index function uses only the direction of R, leading to errors in planar surfaces on large objects that will tend to index onto the same point on the map. Also, mapping is essentially a spherical projection and contains a singularity at (0,0,Rz).

In chrome mapping, this gave rise to spikes whereas in this case, distortions arise in the map around the singularities. This is degradation and as it contributes to the final effect. The difference between longitude-latitude mapping and chrome mapping is that in the former, the map is the environment whereas in the latter, the map is an arbitrary image. In terms of implementation, both of them are identical.

The other environment technique is one in which the environment is projected onto the six sides of the cube. The mapping function R3 -> R2 is no longer spherical and so much less distortion. Environment maps are constructed by taking six images, from a fixed point, in mutually orthogonal directions either with a camera whose field of view is PI/2 or by using a renderer to construct the maps from a modeled scene. These six images are then converted into six mip-maps. The problem with using a real camera is that it is difficult to construct six component map without encountering both geometric and illumination discontinuities at the seams or boundaries. This technique is popular as it gives a neat way of blending computer generated objects and live action sets.

3.5 Multi-texturing

There are many surface rendering effects that can best be implemented by more than a single application of a texture. For example, suppose that we want to apply a shadow to an object whose surface shades are themselves determined by a texture map. One can use a texture map for the shadow, but if there were only a single texture application, this method would not work.

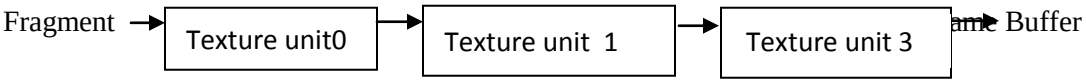


Figure 2.4(c): Sequence of Texture Units (Ed Angel, 1991)

If, instead, we have multiple texture units as in Figure 2.4(c), then one can accomplish this task. Each unit acts as an independent texturing stage starting with the results of the previous stage. This facility is supported in recent versions of OpenGL. Suppose that we want to use two texture units. We can define two texture objects as part of our initialization. Each in turn is then activated and decision on how its texture should be applied is determined. The usual code for multi-texturing is of the form

```
glActiveTexture(GL_TEXTURE0); /* unit 0 */
glBindTexture(GL_TEXTURE_2D, object0);
/* how to apply texture 0 */
glActiveTexture(GL_TEXTURE1); /* unit 1 */
glBindTexture(GL_TEXTURE_2D, object1);
/* how to apply texture 1 */
```

Each texture unit can use different texture coordinates, and the application needs to provide those texture coordinates for each unit.

3.6 Shadow mapping

Shadow mapping or projective shadowing is a process by which shadows are added to 3D computer graphics. This concept was introduced by Lance Williams in 1978. Since then, it has been used both in pre-rendered scenes and real-time scenes in many console and PC games.

Shadows are created by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or depth image of the light source's view, stored in the form of a texture.

3.6.1 Principle of a shadow and a shadow map

If you looked out from a source of light, all of the objects you can see would appear in light. Anything behind those objects, however, would be in shadow. This is the basic principle used to create a shadow map. The light's view is rendered; storing the depth of every surface it sees (the shadow map). Next, the regular scene is rendered comparing the depth of every point drawn (as if it were being seen by the light, rather than the eye) to this depth map.

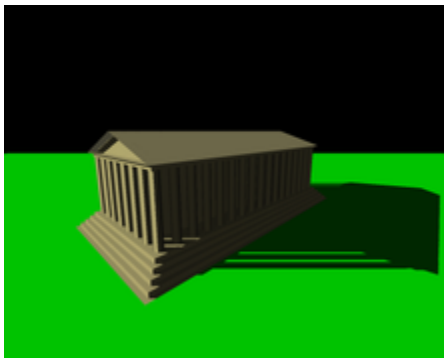


Figure 2.4(d): Shadow mapping (Wikipedia)

This technique is less accurate than shadow volumes, but the shadow map can be a faster alternative depending on how much fill time is required for either technique in a particular application and therefore may be more suitable to real time applications. In addition, shadow maps do not require the use of an additional stencil buffer, and can be modified to produce shadows with a soft edge. Unlike shadow volumes, however, the accuracy of a shadow map is limited by its resolution.

3.7 Shadow Mapping Algorithm overview

Rendering a shadowed scene involves two major drawing steps.

1. The first produces the shadow map itself.
2. The second applies it to the scene.

Depending on the implementation (and number of lights), this may require two or more drawing passes.

3.7.1 Creating the shadow map

The first step renders the scene from the light's point of view. For a point light source, the view should be a perspective projection as wide as its desired angle of effect (it will be a sort of square spotlight). For directional light (e.g., that from the Sun), an orthographic projection should be used. From this rendering, the depth buffer is extracted and saved. Because only the depth information is relevant, it is usual to avoid updating the colour buffers and disable all lighting

and texture calculations for this rendering, in order to save drawing time. This depth map is often stored as a texture in graphics memory.

This depth map must be updated any time there are changes to either the light or the objects in the scene, but can be reused in other situations, such as those where only the viewing camera moves. (If there are multiple lights, a separate depth map must be used for each light.) In many implementations it is practical to render only a subset of the objects in the scene to the shadow map in order to save some of the time it takes to redraw the map. Also, a depth offset which shifts the objects away from the light may be applied to the shadow map rendering in an attempt to resolve stitching problems where the depth map value is close to the depth of a surface being drawn (i.e., the shadow casting surface) in the next step. Alternatively, culling front faces and only rendering the back of objects to the shadow map is sometimes used for a similar result.

3.7.2 Shading the scene

The second step is to draw the scene from the usual camera viewpoint, applying the shadow map. This process has three major components, the first is to find the coordinates of the object as seen from the light, the second is the test which compares that coordinate against the depth map, and finally, once accomplished, the object must be drawn either in shadow or in light.

3.8 Light space coordinates

In order to test a point against the depth map, its position in the scene coordinates must be transformed into the equivalent position as seen by the light. This is accomplished by a matrix multiplication. The location of the object on the screen is determined by the usual coordinate transformation, but a second set of coordinates must be generated to locate the object in light space.

The matrix used to transform the world coordinates into the light's viewing coordinates is the same as the one used to render the shadow map in the first step (under OpenGL this is the product of the modelview and projection matrices). This will produce a set of homogeneous coordinates that need a perspective division (see 3D projection) to become normalized device coordinates, in which each component (x, y, or z) falls between −1 and 1 (if it is visible from the light view). Many implementations (such as OpenGL and Direct3D) require an additional scale and bias matrix multiplication to map those −1 to 1 values to 0 to 1, which are more usual coordinates for depth map (texture map) lookup. This scaling can be done before the perspective division, and is easily folded into the previous transformation calculation by multiplying that matrix with the following:

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If done with a shader, or other graphics hardware extension, this transformation is usually applied at the vertex level, and the generated value is interpolated between other vertices, and passed to the fragment level.

3.9 Depth map test

Once the light-space coordinates are found, the x and y values usually correspond to a location in the depth map texture, and the z value corresponds to its associated depth, which can now be tested against the depth map. If the z value is greater than the value stored in the depth map at the appropriate (x,y) location, the object is considered to be behind an occluding object, and should be marked as a failure, to be drawn in shadow by the drawing process. Otherwise it should be drawn lit. If the (x,y) location falls outside the depth map, the programmer must either decide that the surface should be lit or shadowed by default (usually lit).

In a shader implementation, this test would be done at the fragment level. Also, care needs to be taken when selecting the type of texture map storage to be used by the hardware: if interpolation cannot be done, the shadow will appear to have a sharp jagged edge (an effect that can be reduced with greater shadow map resolution).It is possible to modify the depth map test to produce shadows with a soft edge by using a range of values (based on the proximity to the edge of the shadow) rather than simply pass or fail.

The shadow mapping technique can also be modified to draw a texture onto the lit regions, simulating the effect of a projector. The picture above, captioned "visualization of the depth map projected onto the scene" is an example of such a process.

3.10 Drawing the scene

Drawing the scene with shadows can be done in several different ways. If programmable shaders are available, the depth map test may be performed by a fragment shader which simply draws the object in shadow or lighted depending on the result, drawing the scene in a single pass (after an initial earlier pass to generate the shadow map).

If shaders are not available, performing the depth map test must usually be implemented by some hardware extension (such as GL_ARB_shadow), which usually do not allow a choice between two lighting models (lighted and shadowed), and necessitate more rendering passes:
Render the entire scene in shadow. For the most common lighting models, this should technically be done using only the ambient component of the light, but this is usually adjusted to also include a dim diffuse light to prevent curved surfaces from appearing flat in shadow. Enable the

depth map test, and render the scene lit. Areas where the depth map test fails will not be overwritten, and remain shadowed.

An additional pass may be used for each additional light, using additive blending to combine their effect with the lights already drawn. (Each of these passes requires an additional previous pass to generate the associated shadow map.)

4.0 Conclusion

Environment mapping is a scheme that improves on the mapping techniques of chrome and refraction mapping. In all environment mapping techniques, the accuracy depends on the object being positioned in the center of the surface and that objects in the environment are distant from the objects receiving the environment map

5.0 Summary

There are three mapping techniques: texture, environment and bump mapping. Texture mapping is a method of adding detailed colour to a computer-generated graphic. Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object. Multi-texturing is the use of more than one texture at a time on a shape

6.0 Tutor Marked Assignment

1. What do you understand by texture mapping?
2. Identify application areas of texture mapping.
3. In what scenario is environmental mapping applicable?
4. Iterate factors affecting environmental mapping
5. What is multi-texturing?
6. Explain shadow mapping.
7. Compare bum mapping to a displacement mapping

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL,* Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Paul Rademacher (1997) *Graphics for the Masses*
4. High resolution textures resource366
5. http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html
6. http://www.mayang.com/textures/

7. Blinn, James F. "Simulation of Wrinkled Surfaces", Computer Graphics, Vol. 12 (3), pp. 286-292 SIGGRAPH-ACM (August 1978)

MODULE 3 – Hierarchical Modeling and Animation

UNIT 1: Modeling, Hierarchies and scene graphs

Contents **Pages**

1.0 Introduction

Models are abstractions of the world—both of the real world in which we live and of virtual worlds that is created with computers. In computer science, abstract data types are used to model organizations of objects; in computer graphics, worlds are modelled with geometric objects. We go through analogous processes in computer graphics, choosing which primitives to use in our models and how to show relationships among them. Often, as is true of choosing a mathematical model, there are multiple approaches, so we seek models that can take advantage of the capabilities of graphics systems discussed in this unit.

In this unit we will explore multiple approaches to developing and working with models of geometric objects. We consider models that use as building blocks a set of simple geometric objects: either the primitives supported by our graphics systems or a set of user-defined objects built from these primitives. We extend the use of transformations to include hierarchical relationships among the objects. The techniques are appropriate for applications, such as robotics and figure animation, where the dynamic behavior of the objects is characterized by relationships among the parts of the model. The notion of hierarchy is a powerful one and is an integral part of object oriented methodologies. We extend our hierarchical models of objects to hierarchical models of whole scenes, including cameras, lights, and material properties. Such models allow us to extend our graphics APIs to more object-oriented systems and also give us insight into using graphics over networks and distributed environments, such as the World Wide Web.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the hierarchical models
2. Understand the application hierarchical animations.
3. Understand the scene graphs and Scene trees.

3.0 Main Content

3.1 Hierarchical Models

Suppose that we wish to build a model of an automobile that we can animate. The model can compose from five parts—the chassis and the four wheels — each of which can be described by using our standard graphics primitives. Two frames of a simple animation of the model are shown in Figure 3.1(a).

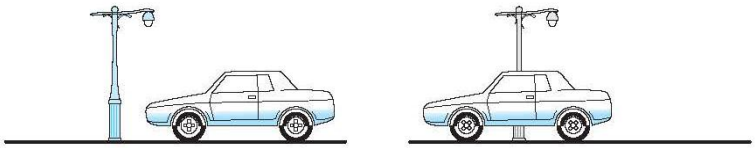


Figure 3.1(a): Simple Animation Model. (Ed Angel, 1991)

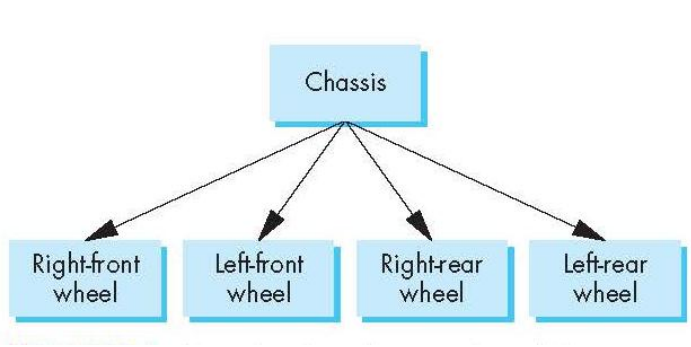
A program could be writen to generate this animation by noting that if each wheel has a radius *r*, then a 360-degree rotation of a wheel must correspond to the car moving forward (or backward) a distance of $2\pi r$. The program could then contain one function to generate each wheel and another to generate the chassis. All these functions could use the same input, such as the desired speed and direction of the automobile. The pseudo-code is presented as follows:

```
{
float s; /* speed */
float d[3]; /* direction */
float t; /* time */
/* determine speed and direction at time t*/
draw_right_front_wheel(s,d);
draw_left_front_wheel(s,d);
draw_right_rear_wheel(s,d);
draw_left_rear_wheel(s,d);
draw_chassis(s,d);
}
```

It is linear and shows none of the relationships among the components of the automobile. There are two types of relationships that would be exploited. First, the movement of the car cannot be separate from the movement of the wheels. If the car moves forward, the wheels must turn. Secondly, the fact that all the wheels of the automobile are identical is considered; they are

merely located in different places, with different orientations. The relationship can represented among parts of the models, both abstractly and visually, with graphs.

Mathematically, a **graph** consists of a set of **nodes** (or vertices) and a set of **edges**. Edges connect pairs of nodes or possibly connect a node to itself. Edges can have a direction associated with them; the graphs we use here are all **directed graphs**, which are graphs that have their edges leaving one node and entering another.



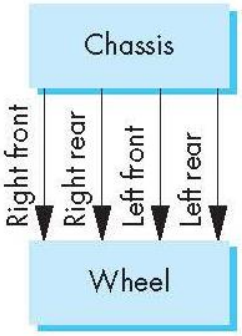


Figure 3.1 (b): Tree structure for an automobile. Fig 3.1(c): DAC model for and automobile (Ed Angel, 1991)

The most important type of graph that can be used is a tree. A (connected) **tree** is a directed graph without closed paths or loops. In addition, each node but one—the **root node**—has one edge entering it. Thus, every node except the root has a **parent node**, the node from which an edge enters, and can have one or more **child nodes**, nodes to which edges are connected. A node without children is called a **terminal node**, or **leaf**. Figure 3.1(b) shows a tree that represents the relationships in our car model. The chassis is the root node, and all four wheels are its children. Although the mathematical graph is a collection of set elements, in practice, both the edges and nodes can contain additional information. For the car example, each node can contain information defining the geometric objects associated with it. The information about the location and orientation of the wheels can be stored either in their nodes or in the edges connecting them with their parent.

In most cars the four wheels are identical, so storing the same information on how to draw each one at four nodes is inefficient. One can use the ideas behind the instance transformation to allow us to use a single prototype wheel in the model. If that is done, the tree structure can be replaced by the **directed acyclic graph** (**DAG**) in Figure 3.1(c). In a DAG, although there are loops, we cannot follow directed edges around any loop. Thus, if we follow any path of directed edges from a node, the path terminates at another node, and in practice, working with DAGs is no more difficult than working with trees. For our car, we can store the information that positions each

instance of the single prototype wheel in the chassis node, in the wheel node, or with the edges. Both forms—trees and DAGs—are **hierarchical** methods of expressing the relationships in the physical model. In each form, various elements of a model can be related to other parts—their parents and their children. We will explore how to express these hierarchies in a graphics program

Advantages of Hierarchies

1. Reasonable control knobs
2. Maintains structural constraints

Disadvantages of Hierarchies

1. Hierarchies does not always give the "right" control knobs trivially e.g. hand or foot position – Re-rooting may help
2. Hierarchical methods can't do closed kinematic chains easily (keep hand on hip)
3. Missing other constraints: do not walk through walls

Hierarchies are a vital tool for modelling and animation

.

3.2 A Robot ARM

Robotics provides many opportunities for developing hierarchical models. Consider the simple robot arm illustrated in Figure 3.1d(a). It can be modelled with three simple objects, or symbols, perhaps using only two parallelepipeds and a cylinder. Each of the symbols can be built up from our basic primitives.

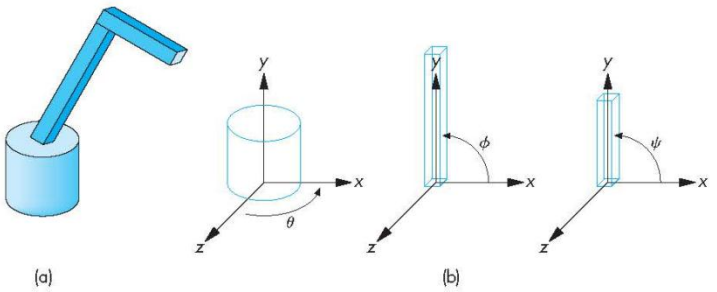


FIGURE 3.1(d) Robot arm. (a) Total model. (b) Components.

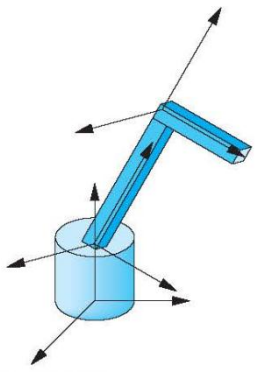


FIGURE 3.1(e) Movement of robot components and frames.

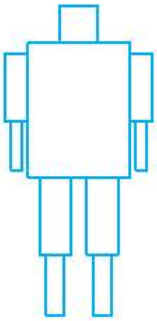
The robot arm consists of the three parts shown in Figure 3.1d(b). The mechanism has three degrees of freedom, two of which can be described by **joint angles** between components and the third by the angle the base makes with respect to a fixed point on the ground. In our model, each joint angle determines how to position a component with respect to the component to which it is

attached, or in the case of the base, the joint angle positions it relative to the surrounding environment. Each joint angle is measured in each component's own frame. We can rotate the base about its vertical axis by an angle θ. This angle is measured from the x-axis to some fixed point on the bottom of the base. The lower arm of the robot is attached to the base by a joint that allows the arm to rotate in the plane $z = 0$ in the arm's frame. This rotation is specified by an angle φ that is measured from the x-axis to the arm. The upper arm is attached to the lower arm by a similar joint, and it can rotate by an angle ψ, measured like that for the lower arm, in its own frame. As the angles vary, we can think of the frames of the upper and lower arms as moving relative to the base. By controlling the three angles, we can position the tip of the upper arm in three dimensions.

Suppose a program is to be written so as to render the simple robot model. Rather than specifying each part of the robot and its motion independently, we take an incremental approach. The base of the robot can rotate about the y-axis in its frame by the angle θ. Thus, we can describe the motion of any point **p** on the base by applying a rotation matrix $\mathbf{R}y(\theta)$ to it. The lower arm is rotated about the z-axis in its own frame, but this frame must be shifted to the top of the base by a translation matrix $\mathbf{T}(0, h1, 0)$, where $h1$ is the height above the base to the point where the joint between the base and the lower arm is located. However, if the base has rotated, then we must also rotate the lower arm, using $\mathbf{R}y(\theta)$. We can accomplish the positioning of the lower arm by applying $\mathbf{R}y(\theta)\mathbf{T}(0, h1, 0)\mathbf{Rz}(\varphi)$ to the arm's vertices. We can interpret the matrix $\mathbf{R}y(\theta)\mathbf{T}(0, h1, 0)$ as the matrix that positions the lower arm *relative* to the object or world frame and $\mathbf{Rz}(\varphi)$ as the matrix that positions the lower arm *relative* to the base. Equivalently, we can interpret these matrices as positioning the frames of the lower arm and base relative to some world frame, as shown in Figure 3.1(e)

3.3 Trees and Traversal

Figure 3.1(f) shows a boxlike representation of a humanoid that might be used for a robot model or in a virtual reality application. If the torso is taken as the root element, one can represent this figure with the tree shown in Figure 3.1(g). Once we the torso have been positioned, the position and orientation of the other parts of the model are determined by the set of joint angles. We can animate the figure by defining the motion of its joints. In a basic model, the knee and elbow joints might each have only a single degree of freedom, like the robot arm, whereas the joint at the neck might have two or three degrees of freedom.



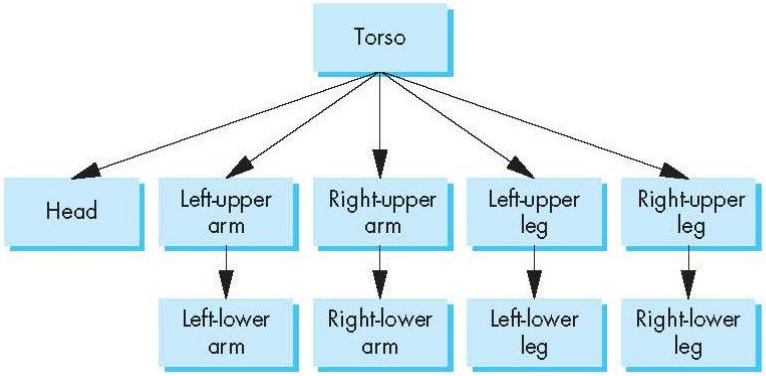


FIGURE 3.1(f) A humanoid figure FIGURE 3.1(g) Tree representation of Figure 5.1

(Ed Angel, 1991)

Let's assume that we have functions, such as head and left_upper_arm that draw the individual parts (symbols) in their own frames. We can now build a set of nodes for our tree by defining matrices that position each part relative to its parent, exactly as we did for the robot arm. If we assume that each body part has been defined at the desired size, each of these matrices is the concatenation of a translation matrix with a rotation matrix. We can show these matrices, as we do in Figure 3.1(h), by using the matrices to label the edges of the tree. Remember that each matrix represents the incremental change when we go from the parent to the child.

The interesting part of this example is how we do the traversal of the tree to draw the figure. In principle, we could use any tree-traversal algorithm, such as a depth-first or breadth-first search. Although in many applications it is insignificant which traversal algorithm is used, we will see that there are good reasons for always using the same algorithm for traversing our graphs. We will always traverse our trees left to right, depth first. That is, we start with the left branch, follow it to the left as deep as we can go, then go back up to the first right branch, and proceed recursively. This order of traversal is called a **pre-order traversal**.

We can write a tree-traversal function in one of two ways. We can do the traversal explicitly in the application code, using stacks to store the required matrices and attributes as we move through the tree. We can also do the traversal recursively. In this second approach, the code is simpler because the storage of matrices and attributes is done implicitly. We develop both approaches because both are useful and because their development yields further insights into how we can build graphics systems.

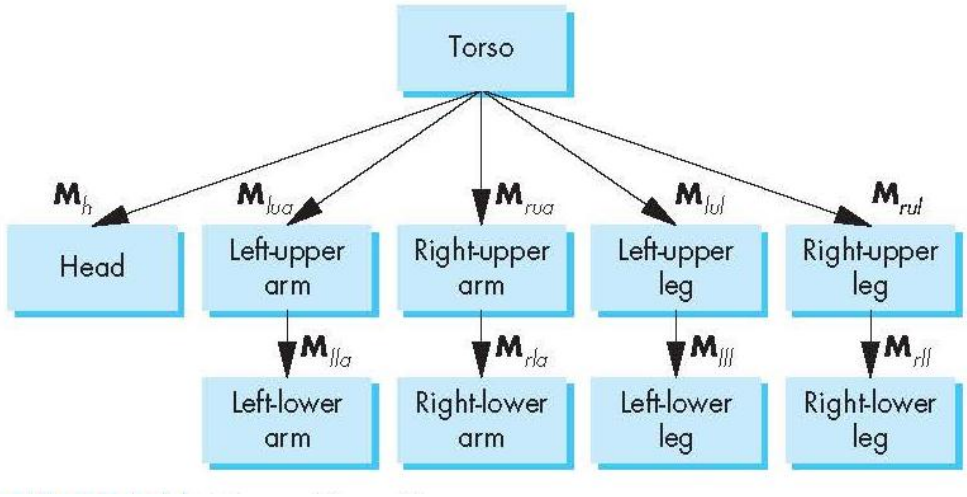


FIGURE 3.1(h) Tree with matrices. (Ed Angel, 1991)

3.4 Scene Graphs

If we think about what goes into describing a scene, we can see that in addition to our graphical primitives and geometric objects derived from these primitives, we have other objects, such as lights and a camera. These objects may also be defined by vertices and vectors and may have attributes, such as colour, that are similar to the attributes associated with geometric primitives. It is the totality of these objects that describes a scene, and there may be hierarchical relationships among these objects. For example, when a primitive is defined in a program, the camera parameters that exist at that time are used to form the image. If we alter the camera lens between the definitions of two geometric objects, we may produce an image in which each object is viewed differently. Although we cannot create such an image with a real camera, the example points out the power of our graphics systems.

The use of tree data structures can be extended to describe these relationships among geometric objects, cameras, lights, and attributes. Knowing that one can write a graphical application program to traverse a graph, we can expand our notion of the contents of a graph to describe an entire scene. One possibility is to use a tree data structure and to include various attributes at each node—in addition to the instance matrix and a pointer to the drawing function.

Another possibility is to allow new types of nodes, such as attribute-definition nodes and matrix-transformation nodes. Consider the tree in Figure 3.1(i). Here, we have set up individual nodes for the colours and for the model-view matrices. The place where there are branches at the top

can be considered a special type of node, a **group node** whose function is to isolate the two children.

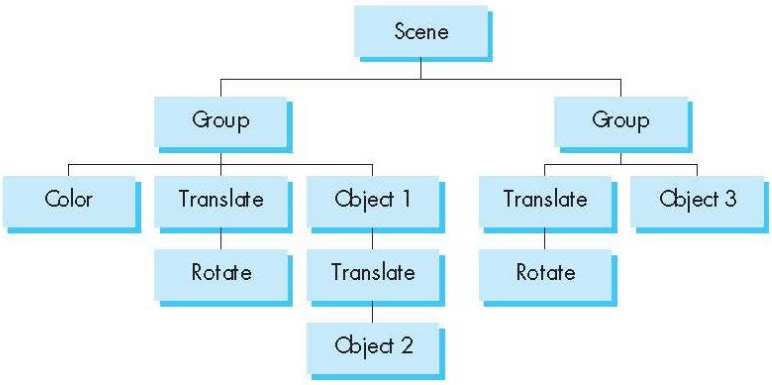


FIGURE 3.1(i) Scene tree. (Ed Angel, 1991)

The group node allows us to preserve the state that exists at the time that we enter a node and thus isolates the state of the subtree beginning at a group node from the rest of the tree. Using our preorder traversal algorithm, the corresponding application code is of the following form:

```
pushAttrib
pushMatrix
colour
translate
rotate
object1
translate
object2
popMatrix
pushMatrix
translate
rotate
object3
popMatrix
popAttrib
```

The group nodes correspond to the OpenGL push and pop functions. This code preserves and restores both the attributes and the model-view matrix before exiting. It sets a drawing colour that applies to the rest of the tree and traverses the tree in a manner similar to the figure example.

We can go further and note that we can use the attribute and matrix stacks to store the viewing conditions; thus, we can create a camera node in the tree. Although we probably do not want a scene in which individual objects are viewed with different cameras, we may want to view the same set of objects with multiple cameras, producing, for example, the multiview orthographic projections and isometric view that are used by architects and engineers. Such images can be created with a scene graph that has multiple cameras.

The scene graph we have just described is equivalent to an OpenGL program in the sense that we can use the tree to generate the program in a totally mechanical fashion. This approach was taken by Open Inventor and later by Open Scene Graph (OSG), both object-oriented APIs that were built on top of OpenGL. Open Inventor and OSG programs build, manipulate, and render a scene graph. Execution of a program causes traversal of the scene graph, which in turn executes graphics functions that are implemented in OpenGL.

The notion of scene graphs couples nicely with the object-oriented paradigm. We can regard all primitives, attributes, and transformations as software objects, and we can define classes to manipulate these entities. From this perspective, we can make use of concepts such as data encapsulation to build up scenes of great complexity with simple programs that use predefined software objects. We can even support animations through software objects that appear as nodes in the scene graph but cause parameters to change and the scene to be redisplayed.

Although, in Open Inventor, the software objects are rendered using OpenGL, the scene graph itself is a database that includes all the elements of the scene. OpenGL is the rendering engine that allows the database to be converted to an image, but it is not used in the specification of the scene. Game engines employ a similar strategy in which the game play modifies a scene graph that can be traversed and rendered at an interactive rate. Graphics software systems are evolving to the configuration shown in Figure 3.1(j). OpenGL is the rendering engine. It usually sits on top of another layer known as the **hardware abstraction layer** (**HAL**), which is a virtual machine that communicates with the physical hardware. Above OpenGL is an object-oriented layer that supports scene graphs and a storage mechanism. User programs can be written for any of the layers, depending on what facilities are required by the application.

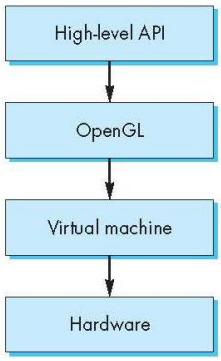


FIGURE 3.1(j) Modern graphics architecture. (Ed Angel, 1991)

4.0 Conclusion

The notion of hierarchy is a powerful one and is an integral part of object oriented methodologies.

5.0 Summary

Models are abstractions of the world; both of the real world in which we live and of virtual worlds that we create with computers. Its application is found in the field of robotics and figure animation where the dynamic behavior of the objects is characterized by relationships among the parts of the model.

6.0 Tutor Marked Assignment

1. What do you understand by hierarchical models?
2. State the advantages and disadvantages of hierarchies.
3. Explain its use in the field or robotics.
4. What are scene graphs?
5. Identify its application areas.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Leler, Wm and Merry, Jim (1996) *3D with HOOPS*, Addison-Wesley ISBN-13:978-0201870251
4. Wernecke, Josie (1994) *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, ISBN 0-201-62495-8 (Release 2)

MODULE 3 – Hierarchical Modeling and Animation

UNIT 2: Animation

Contents **Pages**

1.0 Introduction

Animation is the rapid display of a sequence of images of 2-D or 3-D artwork or model positions in order to create an illusion of movement. The effect is an optical illusion of motion due to the phenomenon of persistence of vision, and can be created and demonstrated in several ways. The most common method of presenting animation is as a motion picture or video

2.0 Objectives

On completing this unit, you would be able to:

1. Understand Animation Techniques; cel, stop animations.

2. Understand the processes flow of traditional animation.
3. Differentiate between 2D and 3D animations.

3.0 Main Content

3.1 Traditional animation

Traditional animation (also called cel animation or hand-drawn animation) was the process used for most animated films of the 20th century. Traditional animation is an animation technique where each frame is drawn by hand. The technique was the dominant form of animation in cinema until the advent of computer animation. The individual frames of a traditionally animated film are photographs of drawings, which are first drawn on paper. To create the illusion of movement, each drawing differs slightly from the one before it. The animators' drawings are traced or photocopied onto transparent acetate sheets called cels, which are filled in with paints in assigned colours or tones on the side opposite the line drawings. The completed character cels are photographed one-by-one onto motion picture film against a painted background by a rostrum camera.

3.1.1 Processes of Traditional Animations

1. **Storyboards**

Traditionally-animated productions, just like other forms of animation, usually begin life as a storyboard, which is a script of sorts written with images as well as words, similar to a giant comic strip. The images allow the animation team to plan the flow of the plot and the composition of the imagery. The storyboard artists will have regular meetings with the director, and may have to redraw or "re-board" a sequence many times before it meets final approval.

2. **Voice recording**

Before true animation begins, a preliminary soundtrack or "scratch track" is recorded, so that the animation may be more precisely synchronized to the soundtrack. Given the slow, methodical manner in which traditional animation is produced, it is almost always easier to synchronize animation to a pre-existing soundtrack than it is to synchronize a soundtrack to pre-existing animation. A completed cartoon soundtrack will feature music, sound effects, and dialogue performed by voice actors. However, the scratch track used during animation typically contains only the voices; any vocal songs the characters must sing along to, and

temporary musical score tracks; the final score and sound effects are added in post-production.

3. **Animatic**

Often, an animatic or story reel is made after the soundtrack is created, but before full animation begins. An animatic typically consists of pictures of the storyboard synchronized with the soundtrack. This allows the animators and directors to work out any script and timing issues that may exist with the current storyboard. The storyboard and soundtrack are amended if necessary, and a new animatic may be created and reviewed with the director until the storyboard is perfected. Editing the film at the animatic stage prevents the animation of scenes that would be edited out of the film since traditional animation is a very expensive and time-consuming process, creating scenes that will eventually be edited out of the completed cartoon is strictly avoided.

4. **Design and timing**

Once the animatic has been approved, it and the storyboards are sent to the design departments. Character designers prepare model sheets for all important characters and props in the film. These model sheets will show how a character or object looks from a variety of angles with a variety of poses and expressions, so that all artists working on the project can deliver consistent work. Sometimes, small statues known as maquettes may be produced, so that an animator can see what a character looks like in three dimensions. At the same time, the background stylists will do similar work for the settings and locations in the project, and the art directors and colour stylists will determine the art style and colour schemes to be used.

While design is going on, the timing director (who in many cases will be the main director) takes the animatic and analyzes exactly what poses, drawings, and lip movements will be needed on what frames. An exposure sheet (or X-sheet for short) is created; this is a printed table that breaks down the action, dialogue, and sound frame-by-frame as a guide for the animators. If a film is based more strongly in music, a bar sheet may be prepared in addition to or instead of an X-sheet. Bar sheets show the relationship between the on-screen action, the dialogue, and the actual musical notation used in the score.

5. **Layout**

Layout begins after the designs are completed and approved by the director. The layout process is the same as the blocking out of shots by a cinematographer on a live-action film. It is here that the background layout artists determine the camera angles, camera paths, lighting, and shading of the scene. Character layout artists will determine the major poses for the characters in the scene, and will make a drawing to indicate each pose. For short films, character layouts are often the responsibility of the director. The layout drawings and storyboards are then spliced, along with the audio and an animatic is formed (not to be

confused by its predecessor the leica reel). The term "animatic" was originally coined by Disney animation studios.

6. Animation

Once the animatic is finally approved by the director, animation begins. In the traditional animation process, animators will begin by drawing sequences of animation on sheets of transparent paper perforated to fit the peg bars in their desks, often using coloured pencils, one picture or "frame" at a time. A peg bar is an animation tool that is used in traditional (cel) animation to keep the drawings in place. The pins in the peg bar match the holes in the paper. It is attached to the animation desk or light table depending on which is being used. A key animator or lead animator will draw the key drawings in a scene, using the character layouts as a guide. The key animator draws enough of the frames to get across the major points of the action; in a sequence of a character jumping across a gap, the key animator may draw a frame of the character as he is about to leap, two or more frames as the character is flying through the air, and the frame for the character landing on the other side of the gap.

Timing is important for the animators drawing these frames; each frame must match exactly what is going on in the soundtrack at the moment the frame will appear, or else the discrepancy between sound and visual will be distracting to the audience. For example, in high-budget productions, extensive effort is given in making sure a speaking character's mouth matches in shape the sound that character's actor is producing as he or she speaks. While working on a scene, a key animator will usually prepare a pencil test of the scene. A pencil test is a preliminary version of the final animated scene; the pencil drawings are quickly photographed or scanned and synced with the necessary soundtracks. This allows the animation to be reviewed and improved upon before passing the work on to his assistant animators, who will go add details and some of the missing frames in the scene. The work of the assistant animators is reviewed, pencil-tested, and corrected until the lead animator is ready to meet with the director and have his scene sweatboxed, or reviewed by the director, producer, and other key creative team members. Similar to the storyboarding stage, an animator may be required to re-do a scene many times before the director will approve it.

This process is the same for both character animation and special effects animation, which on most high-budget productions are done in separate departments. Effects animators animate anything that moves and is not a character, including props, vehicles, machinery and phenomena such as fire, rain, and explosions. Sometimes, instead of drawings, a number of special processes are used to produce special effects in animated films; rain, for example, has been created in Disney animated films since the late-1930s by filming slow-motion footage of water in front of a black background, with the resulting film superimposed over the animation.

7. Pencil test

After all the drawings are cleaned-up, they are then photographed on an animation camera, usually on black and white film stock. Nowadays, pencil tests can be made using a video camera, and computer software.

8. **Backgrounds**

While the animation is being done, the background artists will paint the sets over which the action of each animated sequence will take place. These backgrounds are generally done in gouache or acrylic paint, although some animated productions have used backgrounds done in watercolour, oil paint, or even crayon. Background artists follow very closely the work of the background layout artists and colour stylists (which is usually compiled into a workbook for their use), so that the resulting backgrounds are harmonious in tone with the character designs.

9. **Traditional ink-and-paint and camera**

Once the clean-ups and in between drawings for a sequence are completed, they are prepared for photography, a process known as ink-and-paint. Each drawing is then transferred from paper to a thin, clear sheet of plastic called a cel, a contraction of the material name celluloid (the original flammable cellulose nitrate was later replaced with the more stable cellulose acetate). The outline of the drawing is inked or photocopied onto the cel, and gouache or a similar type of paint is used on the reverse sides of the cels to add colours in the appropriate shades. In many cases, characters will have more than one colour palette assigned to them; the usage of each one depends upon the mood and lighting of each scene. The transparent quality of the cel allows for each character or object in a frame to be animated on different cels, as the cel of one character can be seen underneath the cel of another; and the opaque background will be seen beneath all of the cels.

When an entire sequence has been transferred to cels, the photography process begins. Each cel involved in a frame of a sequence is laid on top of each other, with the background at the bottom of the stack. A piece of glass is lowered onto the artwork in order to flatten any irregularities, and the composite image is then photographed by a special animation camera, also called rostrum camera. The cels are removed, and the process repeats for the next frame until each frame in the sequence has been photographed. Each cel has small registration holes along the top or bottom edge of the cel, which allow the cel to be placed on corresponding peg bars before the camera to ensure that each cel aligns with the one before it; if the cels are not aligned in such a manner, the animation, when played at full speed, will appear "jittery." Sometimes, frames may need to be photographed more than once, in order to implement superimpositions and other camera effects.

10. **Digital ink and paint**

The current process, termed "digital ink and paint," is the same as traditional ink and paint until after the animation drawings are completed; instead of being transferred to cels, the animators' drawings are scanned into a computer, where they are coloured and processed using one or more of a variety of software packages. The resulting drawings are composited in the computer over their respective backgrounds, which have also been scanned into the computer (if not digitally painted), and the computer outputs the final film by either exporting a digital video file, using a video cassette recorder, or printing to film using a high-resolution output device. Use of computers allows for easier exchange of artwork between departments, studios, and even countries and continents.

11. **Computers and digital video cameras**

Computers and digital video cameras can also be used as tools in traditional cel animation without affecting the film directly, assisting the animators in their work and making the whole process faster and easier. Doing the layouts on a computer is much more effective than doing it by traditional methods. Additionally, video cameras give the opportunity to see a "preview" of the scenes and how they will look when finished, enabling the animators to correct and improve upon them without having to complete them first. This can be considered a digital form of pencil testing.

3.1.2 Types of Traditional Animation

Full animation refers to the process of producing high-quality traditionally animated films, which regularly use detailed drawings and plausible movement. Fully animated films can be done in a variety of styles, from more realistically animated works.

1. **Limited animation** involves the use of less detailed and/or more stylized drawings and methods of movement. Pioneered by the artists at the American studio United Productions of America, limited animation can be used as a method of stylized artistic expression.
2. **Rotoscoping** is a technique, patented by Max Fleischer in 1917, where animators trace live-action movement, frame by frame. The source film can be directly copied from actors' outlines into animated drawings.
3. **Live-action/animation** is a technique, when combining hand-drawn characters into live action shots. One of the earlier uses of it was Koko the Clown when Koko was drawn over live action footage.

3.2 Stop motion

Stop-motion animation is used to describe animation created by physically manipulating real-world objects and photographing them one frame of film at a time to create the illusion of movement. Computer software is widely available to create this type of animation. There are

many different types of stop-motion animation, usually named after the type of media used to create the animation. Examples are:

1. **Puppet animation** typically involves stop-motion puppet figures interacting with each other in a constructed environment, in contrast to the real-world interaction in model animation. The puppets generally have an armature inside of them to keep them still and steady as well as constraining them to move at particular joints.
2. **Puppetoon**, created using techniques developed by George Pal, are puppet-animated films which typically use a different version of a puppet for different frames, rather than simply manipulating one existing puppet.
3. **Clay animation**, or Plasticine animation often abbreviated as *claymation*, uses figures made of clay or a similar malleable material to create stop-motion animation. The figures may have an armature or wire frame inside of them, similar to the related puppet animation (below), that can be manipulated in order to pose the figures. Alternatively, the figures may be made entirely of clay, such as in the films of Bruce Bickford, where clay creatures morph into a variety of different shapes.
4. **Cutout animation** is a type of stop-motion animation produced by moving 2-dimensional
5. **Silhouette animation** is a variant of cutout animation in which the characters are backlit and only visible as silhouettes
6. **Model animation** refers to stop-motion animation created to interact with and exist as a part of a live-action world. Intercutting, matte effects, and split screens are often employed to blend stop-motion characters or objects with live actors and settings.
7. **Go motion** is a variant of model animation which uses various techniques to create motion blur between frames of film, which is not present in traditional stop-motion.
8. **Object animation** refers to the use of regular inanimate objects in stop-motion animation, as opposed to specially created items.
9. **Graphic animation** uses non-drawn flat visual graphic material (photographs, newspaper clippings, magazines, etc.) which are sometimes manipulated frame-by-frame to create movement. At other times, the graphics remain stationary, while the stop-motion camera is moved to create on-screen action.
10. **Pixilation** involves the use of live humans as stop motion characters. This allows for a number of surreal effects, including disappearances and reappearances, allowing people to appear to slide across the ground, and other such effects.

3.3 Computer animation

Computer animation is the process used for generating animated images by using computer graphics. Modern computer animation usually uses three-dimensional (3D) computer graphics, although two-dimensional (2D) computer graphics are still used for stylistic, low bandwidth, and

faster real-time renderings. Sometimes the target of the animation is the computer itself, but sometimes the target is another medium, such as film.

Computer animation is essentially a digital successor to the stop motion techniques used in traditional animation with 3D models and frame-by-frame animation of 2D illustrations. Computer generated animations are more controllable than other more physically based processes, such as constructing miniatures for effects shots or hiring extras for crowd scenes, and because it allows the creation of images that would not be feasible using any other technology. It can also allow a single graphic artist to produce such content without the use of actors, expensive set pieces, or props.

To create the illusion of movement, an image is displayed on the computer screen and repeatedly replaced by a new image that is similar to it, but advanced slightly in the time domain (usually at a rate of 24 or 30 frames/second). This technique is identical to how the illusion of movement is achieved with television and motion pictures.

3.3.1 2D animation

2D animation figures are created and/or edited on the computer using 2D bitmap graphics or created and edited using 2D vector graphics. This includes automated computerized versions of traditional animation techniques such as, interpolated morphing, onion skinning and interpolated rotoscoping. 2D animation has many applications, including analog computer animation, Flash animation and PowerPoint animation. Cinemagraphs are still photographs in the form of an animated GIF file of which part is animated.

3.3.2 3D animation

3D animation is digitally modeled and manipulated by an animator. In order to manipulate a mesh, it is given a digital skeletal structure that can be used to control the mesh. This process is called rigging. Various other techniques can be applied, such as mathematical functions (ex. gravity, particle simulations), simulated fur or hair, effects such as fire and water and the use of motion capture to name but a few. These techniques fall under the category of 3D dynamics. Well-made 3D animations can be difficult to distinguish from live action and are commonly used as visual effects for recent movies.

4.0 Conclusion

One open challenge in computer animation is a photorealistic animation of humans. Currently, most computer-animated movies show animal characters, fantasy characters machines or cartoon-like humans. The movie is often cited as the first computer-generated movie to attempt to show realistic-looking humans.

5.0 Summary

In this unit, we have surveyed computer animation and its application areas. Animation is the rapid display of a sequence of images of 2-D or 3-D artwork or model positions in order to create an illusion of movement. Traditional animation was the process used for most animated films of the 20th century. Stop-motion animation is used to describe animation created by physically manipulating real-world objects and photographing them one frame of film at a time to create the illusion of movement

6.0 Tutor Marked Assignment

1. What do you understand by Computer Animation?
2. Identify traditional animation processes.
3. Explain in details animation techniques.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Culhane, Shamus (1990). *Animation Script to Screen.* St Martin's Griffin, 1990. ISBN-13: 978-0312050528
4. Laybourne, Kit. *The Animation Book.* Three Rivers Press, 1998. ISBN-13:978-0517886021
5. Ledoux, Trish, Ranney, Doug, & Patten, Fred (Ed.), *Complete Anime Guide: Japanese Animation Film Directory and Resource Guide*, Tiger Mountain Press 1997
6. Lowe, Richard & Schnotz, Wolfgang (Eds) *Learning with Animation. Research implications for design* Cambridge University Press, 2008. ISBN-13:978-0521851893.

MODULE 3 – Hierarchical Modeling and Animation

UNIT 3: Keyframing and Motion capture.

Contents **Pages**

1.0 Introduction

Keyframing is the simplest form of animating an object. Based on the notion that an object has a beginning state or condition, and will be changing over time in; position, form, colour, luminosity, or any other property, to some different final form. Keyframing takes the stance that we only need to show the "key" frames, or conditions, that describe the transformation of this object, and that all other intermediate positions can be figured out from these.

The latter case also demonstrates that keyframing can effect more than simply an object's position in 3D space. Virtually any property of an object can be keyed at a given condition, and changed to a different through keyframing. Some (slightly) more robust examples of this might be a door opening and closing or a shock absorber compressing and decompressing In the case of the shock absorber, the spring is changing scale in the Z direction, compressing and expanding itself. Beyond scaling, we can also change the shape of an object over time.

2.0 Objectives

On completing this unit, you would be able to:

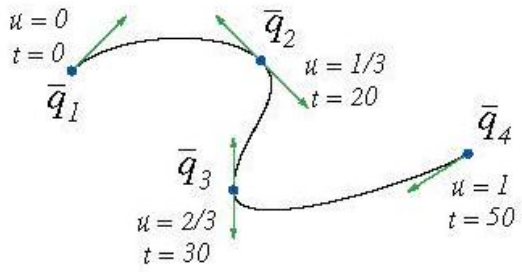
1. Understand the key concepts of keyframing.
2. Understand the applications of keyframing and its advantages over other animation techniques.
3. Understand the motion capture.

3.0 Main Content

3.1 Keyframing

Keyframing is an animation technique where motion curves are interpolated through states at times, (~q_1, ..., ~q_T), called keyframes, specified by a user. A key frame is a drawing that defines the starting and ending points of any smooth transition. They are called "frames" because their position in time is measured in frames on a strip of film. A sequence of keyframes defines *which* movement the viewer will see, whereas the position of the keyframes on the film, video or animation defines the timing of the movement, only two or three keyframes over the span of a second do not create the illusion of movement, the remaining frames are filled with in-betweens.

Catmull-Rom splines are well suited for keyframe animation because they pass through their control points.



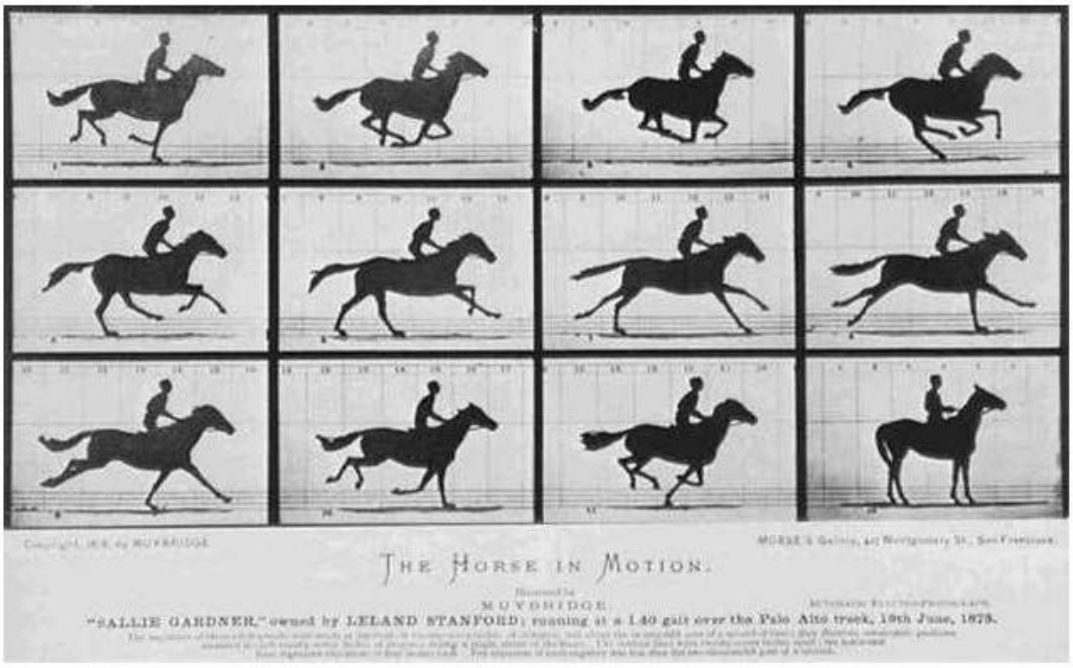


Figure: Horse in motion (Muybridge, 1978)

Advantages of Keyframing

– Very expressive

– Animator has complete control over all motion parameters

Disadvantages of Keyframing

– Very labor intensive

– Difficult to create convincing physical realism

Potentially everything except complex physical phenomena such as smoke, water, or Fire

3.2 Practical Applications

A means to change parameters

In software packages that support animation, especially 3D graphics packages, there are many parameters that can be changed for any one object. One example of such an object is a light. (In 3D graphics, lights function similarly to real-world lights: They cause illumination, cast shadows, and create specular highlights.) Lights have many parameters including light intensity, beam size, light colour, and the texture cast by the light. Supposing that an animator wants the beam size of the light to change smoothly from one value to another within a predefined period

of time, that could be achieved by using keyframes. At the start of the animation, a beam size value is set. Another value is set for the end of the animation. Thus, the software program automatically interpolates the two values, creating a smooth transition.

Video editing

In non-linear digital video editing as well as in video compositing software, a key frame is a frame used to indicate the beginning or end of a change made to the signal. For instance, a key frame could be set to indicate the point at which audio will have faded up or down to a certain level.

Video compression

In video compression, a keyframe, also known as an Intra Frame, is a frame in which a complete image is stored in the data stream. In video compression, only changes that occur from one frame to the next are stored in the data stream, in order to greatly reduce the amount of information that must be stored. This technique capitalizes on the fact that most video sources (such as a typical movie) have only small changes in the image from one frame to the next. Whenever a drastic change to the image occurs, such as when switching from one camera shot to another, or at a scene change, a keyframe must be created. The entire image for the frame must be output when the visual difference between the two frames is so great that representing the new image incrementally from the previous frame would be more complex and would require even more bits than reproducing the whole image.

3.3 Motion capture

Motion capture, motion tracking, or mocap are terms used to describe the process of recording movement and translating that movement on to a digital model. It is used in military, entertainment, sports, and medical applications, and for validation of computer vision and robotics. In filmmaking, it refers to recording actions of human actors, and using that information to animate digital character models in 2D or 3D computer animation. When it includes face and fingers or captures subtle expressions, it is often referred to as *performance capture*.

Figure 3.3(b): Motion capture of the animated movie ‘The Polar Express’, Weta)

In motion capture sessions, movements of one or more actors are sampled many times per second, although with most techniques (recent developments from Weta use images for 2D motion capture and project into 3D), motion capture records only the movements of the actor, not his or her visual appearance. This *animation data* is mapped to a 3D model so that the model performs the same actions as the actor. This is comparable to the older technique of rotoscope, such as the 1978 *The Lord of the Rings* animated film where the visual appearance of the motion of an actor was filmed, then the film used as a guide for the frame-by-frame motion of a hand-drawn animated character.

Camera movements can also be motion captured so that a virtual camera in the scene will pan, tilt, or dolly around the stage driven by a camera operator while the actor is performing, and the motion capture system can capture the camera and props as well as the actor's performance. This allows the computer-generated characters, images and sets to have the same perspective as the video images from the camera. A computer processes the data and displays the movements of the actor, providing the desired camera positions in terms of objects in the set. Retroactively obtaining camera movement data from the captured footage is known as match moving or camera tracking.

In motion capture, an actor has a number of small, round markers attached to his or her body that reflect light in frequency ranges that motion capture cameras are specifically designed to pick up. With enough cameras, it is possible to reconstruct the position of the markers accurately in 3D.
In practice, this is a laborious process. Markers tend to be hidden from cameras and 3D reconstructions fail, requiring a user to manually fix such drop outs. The resulting motion curves are often noisy, requiring yet more effort to clean up the motion data to more accurately match what an animator wants. Despite the labor involved, motion capture has become a popular technique in the movie and game industries, as it allows fairly accurate animations to be created

from the motion of actors. However, this is limited by the density of markers that can be placed on a single actor. Faces are still very difficult to convincingly reconstruct.

3.3.1 Advantages of Motion Capture

Motion capture offers several advantages over traditional computer animation of a 3D model:

1. More rapid, even real time results can be obtained. In entertainment applications this can reduce the costs of keyframe-based animation. For example: Hand Over.
2. The amount of work does not vary with the complexity or length of the performance to the same degree as when using traditional techniques. This allows many tests to be done with different styles or deliveries.
3. Complex movement and realistic physical interactions such as secondary motions, weight and exchange of forces can be easily recreated in a physically accurate manner.
4. The amount of animation data that can be produced within a given time is extremely large when compared to traditional animation techniques. This contributes to both cost effectiveness and meeting production deadlines.
5. Potential for free software and third party solutions reducing its costs.

3.3.2 Disadvantages of Motion Capture

1. Specific hardware and special programs are required to obtain and process the data.
2. The cost of the software, equipment and personnel required can potentially be prohibitive for small productions.
3. The capture system may have specific requirements for the space it is operated in, depending on camera field of view or magnetic distortion.
4. When problems occur, it is easier to reshoot the scene rather than trying to manipulate the data. Only a few systems allow real time viewing of the data to decide if the take needs to be redone.
5. The initial results are limited to what can be performed within the capture volume without extra editing of the data.
6. Movement that does not follow the laws of physics generally cannot be captured.
7. Traditional animation techniques, such as added emphasis on anticipation and follow through, secondary motion or manipulating the shape of the character, as with squash and stretch animation techniques, must be added later.
8. If the computer model has different proportions from the capture subject, artifacts may occur. For example, if a cartoon character has large, over-sized hands, these may intersect the character's body if the human performer is not careful with their physical motion.

3.4 Applications of Motion Capture

1. *Video games* often use motion capture to animate athletes, martial artists, and other in-game characters.
2. *Movies* use motion capture for CG effects, in some cases replacing traditional cel animation, and for completely computer-generated creatures, such as King Kong, *Avatar*, Motion capture has begun to be used extensively to produce films which attempt to simulate or approximate the look of live-action cinema, with nearly photorealistic digital character models.
3. *Virtual Reality and Augmented Reality* allows users to interact with digital content in real-time. This can be useful for training simulations, visual perception tests, or performing virtual walk-throughs in a 3D environment. Motion capture technology is frequently used in digital puppetry systems to drive computer generated characters in real-time.
4. *Gait analysis* is the major application of motion capture in clinical medicine. Techniques allow clinicians to evaluate human motion across several biometric factors, often while streaming this information live into analytical software.

3.5 Methods and systems

Motion tracking or motion capture started as a photogrammetric analysis tool in biomechanics research in the 1970s and 1980s, and expanded into education, training, sports and recently computer animation for television, cinema, and video games as the technology matured. A performer wears markers near each joint to identify the motion by the positions or angles between the markers. Acoustic, inertial, LED, magnetic or reflective markers, or combinations of any of these, are tracked, optimally at least two times the frequency rate of the desired motion, to sub-millimeter positions.

3.5.1 Optical systems

Optical systems utilize data captured from image sensors to triangulate the 3D position of a subject between one or more cameras calibrated to provide overlapping projections. Data acquisition is traditionally implemented using special markers attached to an actor; however, more recent systems are able to generate accurate data by tracking surface features identified dynamically for each particular subject. Tracking a large number of performers or expanding the capture area is accomplished by the addition of more cameras. These systems produce data with 3 degrees of freedom for each marker, and rotational information must be inferred from the relative orientation of three or more markers; for instance shoulder, elbow and wrist markers providing the angle of the elbow.

3.5.1.1 Passive Optical Systems

Passive optical system use markers coated with a retro-reflective material to reflect light that is generated near the cameras lens. The camera's threshold can be adjusted so only the bright reflective markers will be sampled, ignoring skin and fabric.

The centroid of the marker is estimated as a position within the 2-dimensional image that is captured. The grayscale value of each pixel can be used to provide sub-pixel accuracy by finding the centroid of the Gaussian. An object with markers attached at known positions is used to calibrate the cameras and obtain their positions and the lens distortion of each camera is measured.

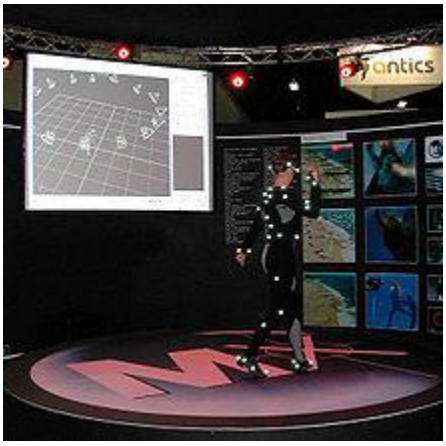


Figure3.3(c): optical motion capture system, Grookda Oger, 2010)

Several markers are placed at specific points on an actor's face during facial optical motion capture. If two calibrated cameras see a marker, a 3 dimensional fix can be obtained. Typically a system will consist of around 6 to 24 cameras. Systems of over three hundred cameras exist to try to reduce marker swap. Extra cameras are required for full coverage around the capture subject and multiple subjects.

Vendors have constraint software to reduce the problem of marker swapping since all markers appear identical. Unlike active marker systems and magnetic systems, passive systems do not require the user to wear wires or electronic equipment. Instead, hundreds of rubber balls are attached with reflective tape, which needs to be replaced periodically. The markers are usually attached directly to the skin (as in biomechanics), or they are velcroed to a performer wearing a full body spandex/lycra suit designed specifically for motion capture. This type of system can capture large numbers of markers at frame rates as high as 10000fps.

3.5.1.2 Active Optical Systems

Active optical systems triangulate positions by illuminating one Light-emitting diodes (LED) at a time very quickly or multiple LEDs with software to identify them by their relative positions, somewhat akin to celestial navigation. Rather than reflecting light back that is generated externally, the markers themselves are powered to emit their own light. Since Inverse Square law provides 1/4 the power at 2 times the distance, this can increase the distances and volume for capture.

The power to each marker can be provided sequentially in phase with the capture system providing a unique identification of each marker for a given capture frame at a cost to the resultant frame rate. The ability to identify each marker in this manner is useful in real-time applications. The alternative method of identifying markers is to do it algorithmically requiring extra processing of the data.

3.5.2 Time modulated active marker

Active marker systems can further be refined by strobing one marker on at a time, or tracking multiple markers over time and modulating the amplitude or pulse width to provide marker ID. 12 megapixel spatial resolution modulated systems show more subtle movements than 4 megapixel optical systems by having both higher spatial and temporal resolution. Directors can see the actors performance in real time, and watch the results on the mocap driven CG character. The unique marker IDs reduces the turnaround, by eliminating marker swapping and providing much cleaner data than other technologies. LEDs with onboard processing and a radio synchronization allow motion capture outdoors in direct sunlight, while capturing at 480 frames per second due to a high speed electronic shutter.

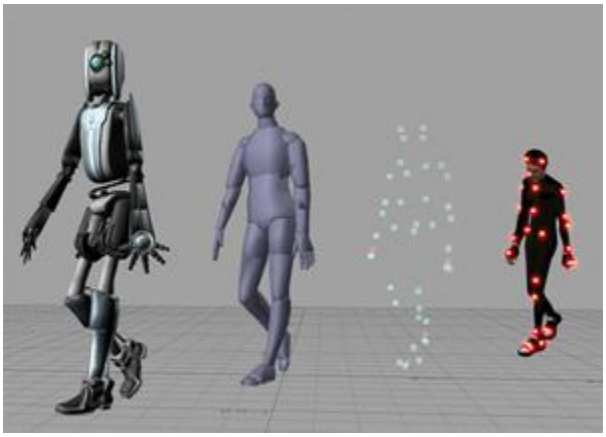


Figure 3.3(d): A high-resolution active marker system, Kameraad Pjotr (2007)

.

Computer processing of modulated IDs allows less hand cleanup or filtered results for lower operational costs. This higher accuracy and resolution requires more processing than passive technologies, but the additional processing is done at the camera to improve resolution via a sub-pixel or centroid processing, providing both high resolution and high speed.

3.5.3 Semi-passive imperceptible marker

One can reverse the traditional approach based on high speed cameras. Systems such as Prakash use inexpensive multi-LED high speed projectors. The specially built multi-LED IR projectors optically encode the space. Instead of retro-reflective or active light emitting diode (LED) markers, the system uses photosensitive marker tags to decode the optical signals. By attaching tags with photo sensors to scene points, the tags can compute not only their own locations of each point, but also their own orientation, incident illumination, and reflectance.

These tracking tags work in natural lighting conditions and can be imperceptibly embedded in attire or other objects. The system supports an unlimited number of tags in a scene, with each tag uniquely identified to eliminate marker reacquisition issues. Since the system eliminates a high speed camera and the corresponding high-speed image stream, it requires significantly lower data bandwidth. The tags also provide incident illumination data which can be used to match scene lighting when inserting synthetic elements. The technique appears ideal for on-set motion capture or real-time broadcasting of virtual sets but has yet to be proven.

3.5.4 Markerless Approach to Motion Capture

Emerging techniques and research in computer vision are leading to the rapid development of the markerless approach to motion capture. Markerless systems such as those developed at Stanford, University of Maryland, MIT, and Max Planck Institute, do not require subjects to wear special equipment for tracking. Special computer algorithms are designed to allow the system to analyze multiple streams of optical input and identify human forms, breaking them down into constituent parts for tracking. Applications of this technology extend deeply into popular imagination about the future of computing technology.

3.6 Non-optical systems

The following subsections discuss on types of non-optical systems

3.6.1 Inertial systems

Inertial Motion Capture technology is based on miniature inertial sensors, biomechanical models and sensor fusion algorithms. The motion data of the inertial sensors (inertial guidance system) is often transmitted wirelessly to a computer, where the motion is recorded or viewed. Most inertial systems use gyroscopes to measure rotational rates. These rotations are translated to a skeleton in the software. Much like optical markers, the more gyros the more natural the data. No external

cameras, emitters or markers are needed for relative motions. Inertial mocap systems capture the full six degrees of freedom body motion of a human in real-time. Benefits of using Inertial systems include: no solving, portability, and large capture areas. Disadvantages include lower positional accuracy and positional drift which can compound over time.

These systems are similar to the Wii (video game) controllers but are more sensitive and have greater resolution and update rates. They can accurately measure the direction to the ground to within a degree. The popularity of inertial systems is rising amongst independent game developers, mainly because of the quick and easy set up resulting in a fast pipeline.

3.6.2 Mechanical motion

Mechanical motion capture systems directly track body joint angles and are often referred to as exo-skeleton motion capture systems, due to the way the sensors are attached to the body. Performers attach the skeletal-like structure to their body and as they move so do the articulated mechanical parts, measuring the performer's relative motion. Mechanical motion capture systems are real-time, relatively low-cost, free-of-occlusion, and wireless (untethered) systems that have unlimited capture volume. Typically, they are rigid structures of jointed, straight metal or plastic rods linked together with potentiometers that articulate at the joints of the body.

3.6.3 Magnetic systems

Magnetic systems calculate position and orientation by the relative magnetic flux of three orthogonal coils on both the transmitter and each receiver. The relative intensity of the voltage or current of the three coils allows these systems to calculate both range and orientation by meticulously mapping the tracking volume. The sensor output is 6DOF, which provides useful results obtained with two-thirds the number of markers required in optical systems; one on upper arm and one on lower arm for elbow position and angle. The markers are not occluded by nonmetallic objects but are susceptible to magnetic and electrical interference from metal objects in the environment, like rebar (steel reinforcing bars in concrete) or wiring, which affect the magnetic field, and electrical sources such as monitors, lights, cables and computers. The sensor response is nonlinear, especially toward edges of the capture area. The wiring from the sensors tends to preclude extreme performance movements. The capture volumes for magnetic systems are dramatically smaller than they are for optical systems. With the magnetic systems, there is a distinction between "AC" and "DC" systems: one uses square pulses, the other uses sine wave pulse.

4.0 Conclusion

A keyframe is a drawing that defines the starting and ending points of any smooth transition. They are called "frames" because their position in time is measured in frames on a strip of film. A sequence of keyframes defines *which* movement the viewer will see, whereas the position of the keyframes on the film, video or animation defines the timing of the movement.

5.0 Summary

Keyframing is the simplest form of animating an object. Based on the notion that an object has a beginning state or condition, and will be changing over time, in position, form, colour, luminosity, or any other property, to some different final form. Its use is found in video editing, video compression as a means to change parameters. Motion capture is used to describe the process of recording movement and translating that movement on to a digital model.

6.0 Tutor Marked Assignment

1. What do you understand by keyframing?
2. Identify the advantages and disadvantages of the animation technique.
3. What is motion capture?
4. Describe the system used in motion capture.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Larry Greenemeier. "E-Motion: Next-Gen Simulators to Blur the Line between Person and Avatar". http://www.scientificamerican.com/article.cfm?id=next-generation-simulator.

MODULE 3 – Hierarchical Modeling and Animation

UNIT 4: Physical Simulation.

Contents **Pages**

1.0 Introduction

Simulation has been seen to cut across different field of endeavours. We will consider its application in education and training, clinical healthcare, automobile industry, biomechanics, engineering and technology. We will also consider various input and output hardware used in simulation.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand physical simulation.
2. Understand the benefits of physical simulation.
3. Identify application area of simulation.
4. Identify Simulation input and output hardware.

3.0 Main Content

3.1 Simulation

Simulation is the imitation of some real thing available, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system.

Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Training simulators include flight simulators for training aircraft pilots in order to provide them with a lifelike experience. Simulation is also used for scientific modeling of natural systems or human systems in order to gain insight into their functioning. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist.

3.2 Types of Simulation

1. *Physical simulation* refers to simulation in which physical objects are substituted for the real thing. These physical objects are often chosen because they are smaller or cheaper than the actual object or system.

2. *Interactive simulation* is a special kind of physical simulation, often referred to as a *human in the loop* simulation, in which physical simulations include human operators, such as in a flight simulator or a driving simulator. Human in the loop simulations can include a computer simulation as a so-called *synthetic environment*.

A computer simulation is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system. It is a tool to create a virtual environment of the real time system. Computer simulation has become a useful part of modeling many natural systems in physics, chemistry and biology, and human systems in economics and social science (the computational sociology) as well as in engineering to gain insight into the operation of those systems. A good example of the usefulness of using computers to simulate can be found in the field of network traffic simulation. In such simulations, the model behaviour will change each simulation according to the set of initial parameters assumed for the environment.

Traditionally, the formal modeling of systems has been via a mathematical model, which attempts to find analytical solutions enabling the prediction of the behaviour of the system from a set of parameters and initial conditions. Computer simulation is often used as an adjunct to, or substitution for, modeling systems for which simple closed form analytic solutions are not possible. There are many different types of computer simulation; the common feature they all share is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states would be prohibitive or impossible. Several software packages exist for running computer-based simulation modeling (e.g. Monte Carlo simulation, stochastic modeling, and multi-method modeling) that make all the modeling almost effortless.

3.3 Physical Simulation

It is possible to simulate the physics of the natural world to generate realistic motions, interactions, and deformations. Dynamics rely on the time evolution of a physical system in response to forces. Forward simulation has the advantage of being reasonably easy to simulate. However, a simulation is often very sensitive to initial conditions, and it is often difficult to predict paths without running a simulation—in other words, control is hard. With inverse dynamics, constraints on a path are specified. Then we attempt to solve for the forces required to produce the desired path. This technique can be very difficult computationally.

Advantages of Physically-based animation

1. Realistic motion i.e close to real life motion.
2. Long simulations are easy to create
3. Natural secondary effects such as wiggles, bending, and so on—materials behave naturally
4. Interactions between objects are also natural.

Disadvantages of Physically-based animation

1. The main disadvantage of physically-based animation is the lack of control, which can be critical, for example, when a complicated series of events needs to be modeled or when an artist needs precise control over elements in a scene.
2. Very slow

3. Not expressive

3.4 Data-Driven Animation

Data-driven animation uses information captured from the real world, such as video or captured motion data, to generate animation. The technique of video textures finds points in a video sequence that are similar enough that a transition may be made without appearing unnatural to a viewer, allowing for arbitrarily long and varied animation from video. A similar approach may be taken to allow for arbitrary paths of motion for a 3D character by automatically finding frames in motion capture data or keyframed sequences that are similar to other frames. An animator can then trace out a path on the ground for a character to follow, and the animation is automatically generated from a database of motion.

Advantages of Data-driven simulation

1. It captures specific style of real actors
2. It is very flexible
3. It can generate new motion in real-time

Disadvantages of Data-driven simulation

1. Requires good data, and possibly lots of it

Uses of Data-driven simulation

1. Character animation

3.5 Application Areas of Simulation

1. Simulation in education and training

Simulation is extensively used for educational purposes. It is frequently used by way of adaptive hypermedia. Simulation is often used in the training of civilian and military personnel. This usually occurs when it is prohibitively expensive or simply too dangerous to allow trainees to use the real equipment in the real world. In such situations they will spend time learning valuable lessons in a "safe" virtual environment yet living a lifelike experience. Often the convenience is to permit mistakes during training for a safety-critical system. Training simulations typically come in one of three categories:

1. "Live" simulation - where actual players use genuine systems in a real environment.
2. "virtual" simulation - where actual players use simulated systems in a synthetic environment, or
3. "Constructive" simulation - where simulated players use simulated systems in a synthetic environment. Constructive simulation is often referred to as "wargaming"

since it bears some resemblance to table-top war games in which players command armies of soldiers and equipment that move around a board.

In standardized tests, "live" simulations are sometimes called "high-fidelity", producing "samples of likely performance", as opposed to "low-fidelity", "pencil-and-paper" simulations producing only "signs of possible performance", but the distinction between high, moderate and low fidelity remains relative, depending on the context of a particular comparison.

Simulations in education are somewhat like training simulations. They focus on specific tasks. The term 'microworld' is used to refer to educational simulations which model some abstract concept rather than simulating a realistic object or environment, or in some cases model a real world environment in a simplistic way so as to help a learner develop an understanding of the key concepts. Normally, a user can create some sort of construction within the microworld that will behave in a way consistent with the concepts being modeled. The Logo programming environment developed by Papert is one of the most famous microworlds

4. **Management games (or business simulations)** have been finding favour in business education in recent years. Business simulations that incorporate a dynamic model enable experimentation with business strategies in a risk free environment and provide a useful extension to case study discussions.
5. **Social simulations** may be used in social science classrooms to illustrate social and political processes in anthropology, economics, history, political science, or sociology courses, typically at the high school or university level. These may, for example, take the form of civics simulations, in which participants assume roles in a simulated society, or international relations simulations in which participants engage in negotiations, alliance formation, trade, diplomacy, and the use of force. Such simulations might be based on fictitious political systems, or be based on current or historical events.

3.6 Common User Interaction Systems for Virtual Simulations

Virtual Simulations represent a specific category of simulation that utilizes simulation equipment to create a simulated world for the user. Virtual Simulations allow users to interact with a virtual world. Virtual worlds operate on platforms of integrated software and hardware components. In this manner, the system can accept input from the user (e.g., body tracking, voice/sound recognition, physical controllers) and produce output to the user (e.g., visual display, aural display, haptic display), Virtual Simulations use the aforementioned modes of interaction to produce a sense of immersion for the user.

3.7 Virtual Simulation Input Hardware

There is a wide variety of input hardware available to accept user input for virtual simulations. The following list briefly describes several of them:

Body Tracking: The motion capture method is often used to record the user's movements and translate the captured data into inputs for the virtual simulation. For example, if a user physically turns their head, the motion would be captured by the simulation hardware in some way and translated to a corresponding shift in view within the simulation.

1. **Capture Suits and/or gloves** may be used to capture movements of user's body parts. The systems may have sensors incorporated inside them to sense movements of different body parts (e.g., fingers). Alternatively, these systems may have exterior tracking devices or marks that can be detected by external ultrasound, optical receivers or electromagnetic sensors. Internal inertial sensors are also available on some systems. The units may transmit data either wirelessly or through cables.
2. **Eye trackers** can also be used to detect eye movements so that the system can determine precisely where a user is looking at any given instant.

Physical Controllers: they provide input to the simulation only through direct manipulation by the user. In virtual simulations, tactile feedback from physical controllers is highly desirable in a number of simulation environments.

1. **Omni directional treadmills** can be used to capture the user's locomotion as they walk or run.
2. **High fidelity instrumentation** such as instrument panels in virtual aircraft cockpits provides users with actual controls to raise the level of immersion. For example, pilots can use the actual global positioning system controls from the real device in a simulated cockpit to help them practice procedures with the actual device in the context of the integrated cockpit system.

Voice/Sound Recognition: This form of interaction may be used either to interact with agents within the simulation (e.g., virtual people) or to manipulate objects in the simulation (e.g., information). Voice interaction presumably increases the level of immersion for the user.

1. Users may use headsets with boom microphones, lapel microphones or the room may be equipped with strategically located microphones.

Research in future input systems hold a great deal of promise for virtual simulations. Systems such as brain-computer interfaces (BCIs) Brain-computer interface offer the ability to further increase the level of immersion for virtual simulation users. It is possible that these types of systems will become standard input modalities in future virtual simulation systems.

3.8 Virtual Simulation Output Hardware

There is a wide variety of output hardware available to deliver stimulus to users in virtual simulations. The following list briefly describes several of them:

a. Visual Display: they provide the visual stimulus to users.

1. **Stationary displays** can vary from a conventional desktop display to 360-degree wrap around screens to stereo three-dimensional screens. Conventional desktop displays can vary in size from 15 to 60+ inches. Wrap around screens are typically utilized in what is known as a Cave Automatic Virtual Environment (CAVE) Cave Automatic Virtual Environment. Stereo three-dimensional screens produce three-dimensional images either with or without special glasses—depending on the design.
2. **Head mounted displays (HMDs)** have small displays that are mounted on headgear worn by the user. These systems are connected directly into the virtual simulation to provide the user with a more immersive experience. Weight, update rates and field of view are some of the key variables that differentiate HMDs. Naturally, heavier HMDs are undesirable as they cause fatigue over time. If the update rate is too slow, the system is unable to update the displays fast enough to correspond with a quick head turn by the user. Slower update rates tend to cause simulation sickness and disrupt the sense of immersion..

b. Aural Display: several different types of audio systems exist to help the user hear and localize sounds spatially. Special software can be used to produce 3D audio effects to create the illusion that sound sources are placed within a defined three-dimensional space around the user.

1. **Stationary conventional speaker** systems may be used to provide dual or multi-channel surround sound. However, external speakers are not as effective as headphones in producing 3D audio effects.
2. **Conventional headphones** offer a portable alternative to stationary speakers. They also have the added advantages of masking real world noise and facilitate more effective 3D audio sound effects.

1. **Haptic Display**: these displays provide sense of touch to the user Haptic technology. This type of output is sometimes referred to as force feedback.

1. **Tactile tile displays** use different types of actuators such as inflatable bladders, vibrators, low frequency sub-woofers, pin actuators and/or thermo-actuators to produce sensations for the user.
2. **End effector displays** can respond to users' inputs with resistance and force. These systems are often used in medical applications for remote surgeries that employ robotic instruments.

d. Vestibular Display: these displays provide a sense of motion to the user motion simulator. They often manifest as motion bases for virtual vehicle simulation such as driving simulators or flight simulators. Motion bases are fixed in place but use actuators to move the simulator in ways that can produce the sensations pitching, yawing or rolling. The simulators can also move in such a way as to produce a sense of acceleration on all axes (e.g., the motion base can produce the sensation of falling).

3.9 Other forms of Simulation

3.9.1 Clinical healthcare simulators

Medical simulators are increasingly being developed and deployed to teach therapeutic and diagnostic procedures as well as medical concepts and decision making to personnel in the health professions. Simulators have been developed for training procedures ranging from the basics such as blood draw, to laparoscopic surgery and trauma care. They are also important to help on prototyping new devices for biomedical engineering problems. Currently, simulators are applied to research and development of tools for new therapies, treatments and early diagnosis in medicine.

Another important medical application of a simulator — although, perhaps, denoting a slightly different meaning of *simulator* — is the use of a placebo drug, a formulation that simulates the active drug in trials of drug efficacy.

3.9.2 Automobile simulator

An automobile simulator provides an opportunity to reproduce the characteristics of real vehicles in a virtual environment. It replicates the external factors and conditions with which a vehicle interacts enabling a driver to feel as if they are sitting in the cab of their own vehicle. Scenarios and events are replicated with sufficient reality to ensure that drivers become fully immersed in the experience rather than simply viewing it as an educational experience.

3.9.3 Biomechanics simulators

A biomechanics simulator is used to analyze walking dynamics, study sports performance, simulate surgical procedures, analyze joint loads, design medical devices, and animates human

and animal movement. A neuro-mechanical simulator that combines biomechanical and biologically realistic neural network simulation. It allows the user to test hypotheses on the neural basis of behavior in a physically accurate 3-D virtual environment.

3.9.4 City and urban simulation

A city simulator can be a city-building game but can also be a tool used by urban planners to understand how cities are likely to evolve in response to various policy decisions. AnyLogic is an example of modern, large-scale urban simulators designed for use by urban planners. City simulators are generally agent-based simulations with explicit representations for land use and transportation. UrbanSim and LEAM are examples of large-scale urban simulation models that are used by metropolitan planning agencies and military bases for land use and transportation planning.

3.9.5 Classroom of the future

The "classroom of the future" will probably contain several kinds of simulators, in addition to textual and visual learning tools. This will allow students to enter the clinical years better prepared, and with a higher skill level. The advanced student or postgraduate will have a more concise and comprehensive method of retraining — or of incorporating new clinical procedures into their skill set — and regulatory bodies and medical institutions will find it easier to assess the proficiency and competency of individuals.

The classroom of the future will also form the basis of a clinical skills unit for continuing education of medical personnel; and in the same way that the use of periodic flight training assists airline pilots, this technology will assist practitioners throughout their career.

3.9.6 Communication Satellite Simulation

Modern satellite communications systems (SatCom) are often large and complex with many interacting parts and elements. In addition, the need for broadband connectivity on a moving vehicle has increased dramatically in the past few years for both commercial and military applications. To accurately predict and deliver high quality of service, satcom system designers have to factor in terrain as well as atmospheric and meteorological conditions in their planning. To deal with such complexity, system designers and operators increasingly turn towards computer models of their systems to simulate real world operational conditions and gain insights in to usability and requirements prior to final product sign-off. Modeling improves the understanding of the system by enabling the SatCom system designer or planner to simulate real world performance by injecting the models with multiple hypothetical atmospheric and environmental conditions.

3.9.7 Digital Lifecycle Simulation

Simulation solutions are being increasingly integrated with CAx (CAD, CAM, CAE....) solutions and processes. The use of simulation throughout the product lifecycle, especially at the earlier concept and design stages, has the potential of providing substantial benefits. These benefits range from direct cost issues such as reduced prototyping and shorter time-to-market, to better performing products and higher margins. However, for some companies, simulation has not provided the expected benefits.

3.9.8 Engineering, technology or process simulation

Simulation is an important feature in engineering systems or any system that involves many processes. For example, in electrical engineering, delay lines may be used to simulate propagation delay and phase shift caused by an actual transmission line. Similarly, dummy loads may be used to simulate impedance without simulating propagation, and is used in situations where propagation is unwanted. A simulator may imitate only a few of the operations and functions of the unit it simulates.

Most engineering simulations entail mathematical modeling and computer assisted investigation. There are many cases, however, where mathematical modeling is not reliable. Simulation of fluid dynamics problems often requires both mathematical and physical simulations. In these cases the physical models require dynamic similitude. Physical and chemical simulations have also direct realistic uses, rather than research uses. In chemical engineering, for example, process simulations are used to give the process parameters immediately used for operating chemical plants, such as oil refineries.

4.0 Conclusion

Key issues in simulation include acquisition of valid source information about the relevant selection of key characteristics and behaviours, the use of simplifying approximations and assumptions within the simulation, and fidelity and validity of the simulation outcomes.

5.0 Summary

Simulation is the imitation of some real thing available, state of affairs, or process. Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Virtual Simulations allow users to interact with a virtual world. Virtual worlds operate on platforms of integrated software and hardware components. In this manner, the system can accept input from the user (e.g., body tracking, voice/sound recognition, physical controllers) and produce output to the user (e.g., visual display, aural display, haptic display)

6.0 Tutor Marked Assignment

1. What do you understand by Computer Simulation?
2. Highlight the merits and demerits of physically-based animation.
3. Identify the application areas of simulation in various fields.
4. Identify the input and output hardware devices used for simulation.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Encyclopedia of Computer Science, "designing a model of a real or imagined system and conducting experiments with that model".
4. Sokolowski, J.A., Banks, C.M. *Principles of Modeling and Simulation*. Hoboken, NJ: Wiley, 2009. ISBN-13:978-0470289433

MODULE 4 – Curves and surfaces, image and the human visual system

UNIT 1: Introduction to curves and surfaces: Bezier, B-splines and NURBS.

Contents **Pages**

1.0 Introduction

Curves are found in various areas of computer graphics. They are used when creating 3D models, vector images, animations, or for example in definition of TrueType fonts. There is a great variety of curves. Some are easy to use, some are flexible enough to describe a large variety of shapes, and some are simple enough to be implemented and accelerated by graphics hardware. NURBS and Bézier curves are ones of the most commonly used curves.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the concept of curves and surfaces in computer
2. Understand the application of curves and surfaces.
3. Understand Bezier, hermite, b-spline curves.
4. Understand NURBS.

3.0 Main Content

3.1 A Bézier curve

A Bézier curve is a parametric curve frequently used in computer graphics and related fields. Generalizations of Bézier curves to higher dimensions are called Bézier surfaces, of which the Bézier triangle is a special case.

Bézier curves are also used in the time domain, particularly in animation and interface design, e.g., a Bézier curve can be used to specify the velocity over time of an object such as an icon moving from A to B, rather than simply moving at a fixed number of pixels per step. When animators or interface designers talk about the "physics" or "feel" of an operation, they may be referring to the particular Bézier curve used to control the velocity over time of the move in question.

Bézier curves were widely publicized in 1962 by the French engineer Pierre Bézier, who used them to design automobile bodies. The curves were first developed in 1959 by Paul de Casteljau using de Casteljau's algorithm, a numerically stable method to evaluate Bézier curves.

3.2 Applications of Bezier curves

Bézier curves are widely used in computer graphics to model smooth curves. As the curve is completely contained in the convex hull of its control points, the points can be graphically displayed and used to manipulate the curve intuitively. Affine transformations such as translation

and rotation can be applied on the curve by applying the respective transform on the control points of the curve.

Quadratic and cubic Bézier curves are most common; higher degree curves are more expensive to evaluate.

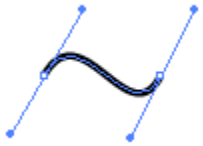


Figure 4.1(a): Bézier path in Adobe Illustrator

When more complex shapes are needed, low order Bézier curves are patched together. This is commonly referred to as a "path" in vector graphics standards (like SVG) and vector graphics programs. To guarantee smoothness, the control point at which two curves meet must be on the line between the two control points on either side.

The simplest method for scan converting (rasterizing) a Bézier curve is to evaluate it at many closely spaced points and scan convert the approximating sequence of line segments. However, this does not guarantee that the rasterized output looks sufficiently smooth, because the points may be spaced too far apart. Conversely, it may generate too many points in areas where the curve is close to linear. A common adaptive method is recursive subdivision, in which a curve's control points are checked to see if the curve approximates a line segment to within a small tolerance. If not, the curve is subdivided parametrically into two segments, $0 \le t \le 0.5$ and $0.5 \le t \le 1$, and the same procedure is applied recursively to each half. There are also forward differencing methods, but great care must be taken to analyze error propagation. Analytical methods where a spline is intersected with each scan line involve finding roots of cubic polynomials (for cubic splines) and dealing with multiple roots, so they are not often used in practice.

A Bézier curve is defined by its order (linear, quadratic, cubic, etc.) and a set of *control points* P_0 through P_n, the number n of which depends on the order ($n = 2$ for linear, 3 for quadratic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

3.2.1 Linear Bézier curves

Given points P_0 and P_1, a linear Bézier curve is simply a straight line between those two points. The curve is given by

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , \ t \in [0,1]$$

and is equivalent to linear interpolation.

3.2.2 Quadratic Bézier curves

A quadratic Bézier curve is the path traced by the function B(*t*), given points P_0, P_1, and P_2,

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \ , \ t \in [0,1],$$

which can be interpreted as the linear interpolant of corresponding points on the linear Bézier curves from P_0 to P_1 and from P_1 to P_2 respectively. More explicitly it can be written as:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \ , \ t \in [0,1].$$

It departs from P_0 in the direction of P_1, then bends to arrive at P_2 in the direction from P_1. In other words, the tangents in P_0 and P_2 both pass through P_1. This is directly seen from the derivative of the Bézier curve:

$$\mathbf{B}'(t) = 2(1-t)(\mathbf{P}_1 - \mathbf{P}_0) + 2t(\mathbf{P}_2 - \mathbf{P}_1)\,.$$

A quadratic Bézier curve is also a parabolic segment. As a parabola is a conic section, some sources refer to quadratic Béziers as "conic arcs".

3.2.3 Cubic Bézier curves

Four points P_0, P_1, P_2 and P_3 in the plane or in higher-dimensional space define a cubic Bézier curve. The curve starts at P_0 going toward P_1 and arrives at P_3 coming from the direction of P_2. Usually, it will not pass through P_1 or P_2; these points are only there to provide directional information. The distance between P_0 and P_1 determines "how long" the curve moves into direction P_2 before turning towards P_3.

Writing $B_{Pi,Pj,Pk}(t)$ for the quadratic Bézier curve defined by points P_i, P_j, and P_k, the cubic Bézier curve can be defined as a linear combination of two quadratic Bézier curves:

$$\mathbf{B}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2}(t) + t\mathbf{B}_{\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3}(t) \ , \ t \in [0,1].$$

The explicit form of the curve is:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \ , \ t \in [0,1].$$

For some choices of P_1 and P_2 the curve may intersect itself, or contain a cusp.

Bézier curves can be defined for any degree *n.*

3.2.4 Recursive definition of A Bézier curve

A recursive definition for the Bézier curve of degree *n* expresses it as a linear interpolation between two Bézier curves of degree $n - 1$.

Let $\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}$ denote the Bézier curve determined by the points P_0, P_1, ..., P_n. Then

$$\mathbf{B}_{\mathbf{P}_0}(t) = \mathbf{P}_0 \text{ to start, and}$$
$$\mathbf{B}(t) = \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t)$$

This recursion is elucidated in the animations below.

3.3 Explicit definition of Bezier curves

The formula can be expressed explicitly as follows:

$$\begin{aligned}\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i\mathbf{P}_i \\ &= (1-t)^n\mathbf{P}_0 + \binom{n}{1}(1-t)^{n-1}t\mathbf{P}_1 + \cdots \\ &\quad \cdots + \binom{n}{n-1}(1-t)t^{n-1}\mathbf{P}_{n-1} + t^n\mathbf{P}_n, \quad t \in [0,1],\end{aligned}$$

where $\binom{n}{i}$ is the binomial coefficient.

For example, for $n = 5$:

$$\begin{aligned}\mathbf{B}(t) = {} & (1-t)^5\mathbf{P}_0 + 5t(1-t)^4\mathbf{P}_1 + 10t^2(1-t)^3\mathbf{P}_2 \\ & + 10t^3(1-t)^2\mathbf{P}_3 + 5t^4(1-t)\mathbf{P}_4 + t^5\mathbf{P}_5, \quad t \in [0,1].\end{aligned}$$

Terminology

Some terminology is associated with these parametric curves. We have

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{b}_{i,n}(t)\mathbf{P}_i, \quad t \in [0,1]$$

where the polynomials

$$\mathbf{b}_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots n$$

are known as Bernstein basis polynomials of degree *n*, defining $t^0 = 1$ and $(1 - t)^0 = 1$. The binomial coefficient, $\binom{n}{i}$, has the alternative notation,

$${}^n\mathbf{C}_i = \binom{n}{i} = \frac{n!}{i!(n-i)!}.$$

The points P_i are called *control points* for the Bézier curve. The polygon formed by connecting the Bézier points with lines, starting with P_0 and finishing with P_n, is called the *Bézier polygon* (or *control polygon*). The convex hull of the Bézier polygon contains the Bézier curve.

3.2.5 Properties of Bezier Curves

1. The curve begins at P_0 and ends at P_n; this is the so-called *endpoint interpolation* property.
2. The curve is a straight line if and only if all the control points are collinear.
3. The start (end) of the curve is tangent to the first (last) section of the Bézier polygon.
4. A curve can be split at any point into two subcurves, or into arbitrarily many subcurves, each of which is also a Bézier curve.
5. Some curves that seem simple, such as the circle, cannot be described exactly by a Bézier or piecewise Bézier curve; though a four-piece cubic Bézier curve can approximate a circle (see Bézier spline), with a maximum radial error of less than one part in a thousand, when each inner control point (or offline point) is the distance

$$\frac{4\left(\sqrt{2}-1\right)}{3}$$

horizontally or vertically from an outer control point on a unit circle. More generally, an *n*-piece cubic Bézier curve can approximate a circle, when each inner control point is the distance

$$\frac{4}{3}\tan(t/4)$$

from an outer control point on a unit circle, where t is 360/n degrees, and $n > 2$.

6. The curve at a fixed offset from a given Bézier curve, often called an *offset curve* (lying "parallel" to the original curve, like the offset between rails in a railroad track), cannot be exactly formed by a Bézier curve (except in some trivial cases). However, there are heuristic methods that usually give an adequate approximation for practical purpose.
7. Every quadratic Bézier curve is also a cubic Bézier curve, and more generally, every degree n Bézier curve is also a degree m curve for any $m > n$. In detail, a degree n curve with control points $P_0, \ldots, P_n$ is equivalent (including the parameterization) to the degree $n + 1$ curve with control points $P'_0, \ldots, P'_{n+1}$, where

$$\mathbf{P}'_k = \tfrac{k}{n+1}\mathbf{P}_{k-1} + \left(1 - \tfrac{k}{n+1}\right)\mathbf{P}_k.$$

3.2.5 The Bézier surface

The Bézier surface is formed as the cartesian product of the blending functions of two orthogonal Bézier curves.

$$\mathbf{B}(u,v) = \sum_{i=0}^{Ni} \sum_{j=0}^{Nj} P_{i,j} \frac{Ni!}{i!\,(Ni-i)!} u^i (1-u)^{Ni-i} \frac{Nj!}{j!\,(Nj-j)!} v^j (1-v)^{Nj-j}$$

$$0 \le u \le 1$$
$$0 \le v \le 1$$

Where $P_{i,j}$ is the i,jth control point. There are N_{i+1} and N_{j+1} control points in the i and j directions respectively.

The corresponding properties of the Bézier curve apply to the Bézier surface.

- The surface does not in general pass through the control points except for the corners of the control point grid.
- The surface is contained within the convex hull of the control points.

C Source Example

The following source code generates the surface shown in the first example above. It is provided for illustration only, the headers and prototype files are not given.

```
#define NI 5
#define NJ 4
XYZ inp[NI+1][NJ+1];
#define RESOLUTIONI 10*NI
```

```
#define RESOLUTIONJ 10*NJ
XYZ outp[RESOLUTIONI][RESOLUTIONJ];

int main(argc,argv)
int argc;
char **argv;
{
   int i,j,ki,kj;
   double mui,muj,bi,bj;

   /* Create a random surface */
   srandom(1111);
   for (i=0;i<=NI;i++) {
      for (j=0;j<=NJ;j++) {
         inp[i][j].x = i;
         inp[i][j].y = j;
         inp[i][j].z = (random() % 10000) / 5000.0 - 1;
      }
   }

   for (i=0;i<RESOLUTIONI;i++) {
      mui = i / (double)(RESOLUTIONI-1);
      for (j=0;j<RESOLUTIONJ;j++) {
         muj = j / (double)(RESOLUTIONJ-1);
         outp[i][j].x = 0;
         outp[i][j].y = 0;
         outp[i][j].z = 0;
         for (ki=0;ki<=NI;ki++) {
            bi = BezierBlend(ki,mui,NI);
            for (kj=0;kj<=NJ;kj++) {
               bj = BezierBlend(kj,muj,NJ);
               outp[i][j].x += (inp[ki][kj].x * bi * bj);
               outp[i][j].y += (inp[ki][kj].y * bi * bj);
               outp[i][j].z += (inp[ki][kj].z * bi * bj);
            }
         }
      }
   }

   printf("LIST\n");

   /* Display the surface, in this case in OOGL format for GeomView */
   printf("{ = CQUAD\n");
   for (i=0;i<RESOLUTIONI-1;i++) {
      for (j=0;j<RESOLUTIONJ-1;j++) {
        printf("%g %g %g 1 1 1 1\n",
            outp[i][j].x,    outp[i][j].y,    outp[i][j].z);
        printf("%g %g %g 1 1 1 1\n",
            outp[i][j+1].x,  outp[i][j+1].y,  outp[i][j+1].z);
        printf("%g %g %g 1 1 1 1\n",
            outp[i+1][j+1].x,outp[i+1][j+1].y,outp[i+1][j+1].z);
        printf("%g %g %g 1 1 1 1\n",
            outp[i+1][j].x,  outp[i+1][j].y,  outp[i+1][j].z);
      }
   }
   printf("}\n");
```

```
   /* Control point polygon */
   for (i=0;i<NI;i++) {
      for (j=0;j<NJ;j++) {
         printf("{ = SKEL 4 1  \n");
         printf("%g %g %g \n",inp[i][j].x,inp[i][j].y,inp[i][j].z);
         printf("%g %g %g \n",inp[i][j+1].x,inp[i][j+1].y,inp[i][j+1].z);
         printf("%g %g %g
\n",inp[i+1][j+1].x,inp[i+1][j+1].y,inp[i+1][j+1].z);
         printf("%g %g %g \n",inp[i+1][j].x,inp[i+1][j].y,inp[i+1][j].z);
         printf("5 0 1 2 3 0\n");
         printf("}\n");
      }
   }
}
```

Bézier Blending Function

This function computes the blending function as used in the Bézier surface code above. It is written for clarity, not efficiency. Normally, if the number of control points is constant, the blending function would be calculated once for each desired value of mu.

```
double BezierBlend(k,mu,n)
int k;
double mu;
int n;
{
   int nn,kn,nkn;
   double blend=1;

   nn = n;
   kn = k;
   nkn = n - k;

   while (nn >= 1) {
      blend *= nn;
      nn--;
      if (kn > 1) {
         blend /= (double)kn;
         kn--;
      }
      if (nkn > 1) {
         blend /= (double)nkn;
         nkn--;
      }
   }
   if (k > 0)
      blend *= pow(mu,(double)k);
   if (n-k > 0)
      blend *= pow(1-mu,(double)(n-k));

   return(blend);
}
```

3.3 B-Splines

A B-Spline consists of multiple Bézier arcs and provides a unified mechanism on how to define continuity in the joins. Consider two cubic Bézier curves - that is 8 total control points (4 per curve).

figure 4.1(b): B-Splines consist of Bézier arcs.

Making the last point of the first (green) curve equal to the first point of the second (violet) curve - this saves us 1 point leaving us with 7 total control points. We have replaced one control point with an external condition.

The third (blue) curve and the fourth (yellow) curve share ending points just like in previous case, but and also share the same tangent direction at the junction point. There are two external conditions and only 6 control points are necessary to describe the curves.

B-Splines use external conditions to put multiple pieces together while keeping the original concept of control points. The neighbor curves share some control points. External conditions are either implicit (uniform curves) or explicitly given by a knot vector. Knot vector defines how much information should be shared by neighbor curves (segments).

Knot vector is a sequence of numbers, usually from 0 to 1, for example (0, 0.5, 0.5, 0.7, 1), and it holds the information about external conditions mentioned earlier. Number of intervals defines number of segments (3 in our case: 0-0.5, 0.5-0.7, 0.7-1). Numbers in knot vector are called knots and each knot has its multiplicity. Multiplicity of knot 0.7 is 1, while multiplicity of knot 0.5 is 2. The higher the multiplicity, the less information share the neighbor segments. When multiplicity is equal to the degree of used curves, there is a sharp edge (green and violet curves on the image).

3.4 NURBS

NURBS stands for Non-Uniform Rational B-Spline. It means NURBS uses rational Bézier curves and a non-uniform explicitly given knot vector. Therefore, degree, control points, weights, and knot vector is needed to specify a NURBS curve. So far, we were talking about curves - one-dimensional formations. The principles can be applied to higher-dimensional objects like surfaces or volumes. Surfaces are used when creating 3D objects, for example landscape while volumes can be used to define a non-linear transformation.

Development of NURBS began in the 1950s by engineers who were in need of a mathematically precise representation of freeform surfaces like those used for ship hulls, aerospace exterior surfaces, and car bodies, which could be exactly reproduced whenever technically needed. Prior representations of this kind of surface only existed as a single physical model created by a designer.

The pioneers of this development were Pierre Bézier who worked as an engineer at Renault, and Paul de Casteljau who worked at Citroën, both in France. Bézier worked nearly parallel to de Casteljau, neither knowing about the work of the other. But because Bézier published the results of his work, the average computer graphics user today recognizes splines — which are represented with control points lying off the curve itself — as Bézier splines, while de Casteljau's name is only known and used for the algorithms he developed to evaluate parametric surfaces. In the 1960s it became clear that non-uniform, rational B-splines are a generalization of Bézier splines, which can be regarded as uniform, non-rational B-splines.

At first NURBS were only used in the proprietary CAD packages of car companies. Later they became part of standard computer graphics packages. They allow representation of geometrical shapes in a compact form. They can be efficiently handled by the computer programs and yet allow for easy human interaction. NURBS surfaces are functions of two parameters mapping to a surface in three-dimensional space. The shape of the surface is determined by control points.

3.4.1 Examples of NURBS curves

Following screenshots demonstrate different uses of NURBS in 3D graphics.

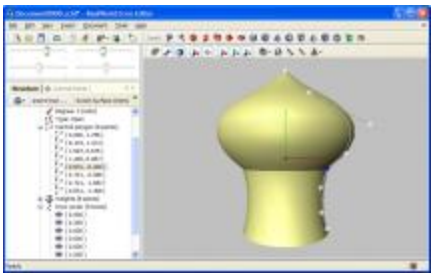


Figure 4.1(c): Surfaces or revolution can roughly approximate relatively large amount of different shapes.

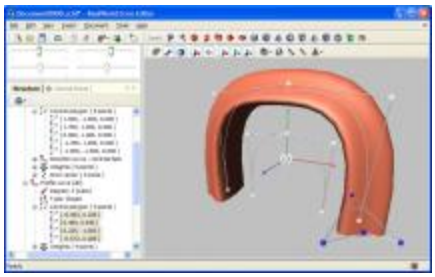


Figure 4.1(d): Surface was created by moving a 2D NURBS curve along a path defined by another 3D NURBS curve.

The left image demonstrates a surface created by revolving a 2D NURBS curve around Y axis. The curve itself consists of 3 pieces (knot vector: 0, 0.2, 0.6, 0.6, 0.6, 1). Join between the two upper pieces is smooth, because the multiplicity of knot 0.2 is 1 and curve degree is 3. On the other hand, knot 0.6 with multiplicity 3 causes a sharp edge.

The right image shows a surface created by sweeping a 2D curve along a 3D trajectory.

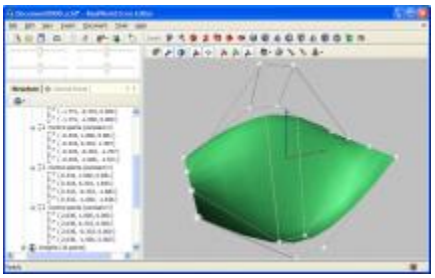


figure 4.1(e): NURBS surfaces need relatively large amount of control points, which makes them hard to control.

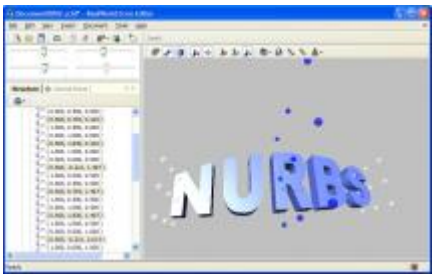


figure 4.1(f): The middle part of the text is magnified and the text is bent using a 2nd degree NURBS volume.

Left image shows a NURBS surface and its control points. NURBS surfaces are used rather rarely in their pure form because the number of control points is usually large (4x4 in our simple case) and the surface becomes hard to control.

Right image shows a 3D text that was transformed using a Bézier (or NURBS) volume of degree 2. The text is bent and its central part is larger - that effect was caused by the non-linear transformation defined by the NURBS volume (note the control points in the center of the model).

3.4.2 Operations with NURBS

When working with NURBS in their pure form, there is one very useful operation: inserting new knot. A knot can be inserted into a NURBS curve without changing the shape of the curve. The desired side effect of this operation is an additional control point that provides finer control of the related region of the NURBS curve or surface.

There are other operations with NURBS, like elevating degree, removing knots, or computing control point positions from points laying on a curve, but they do not reach the usefulness of knot insertion.

3.5 Subdivision surface

A subdivision surface is a method of representing a smooth surface via the specification of a coarser piecewise linear polygon mesh. The smooth surface can be calculated from the coarse mesh as the limit of a recursive process of subdividing each polygonal face into smaller faces that better approximate the smooth surface.

The subdivision surfaces are defined recursively. The process starts with a given polygonal mesh. A refinement scheme is then applied to this mesh. This process takes that mesh and subdivides it, creating new vertices and new faces. The positions of the new vertices in the mesh are computed based on the positions of nearby old vertices. In some refinement schemes, the positions of old vertices might also be altered (possibly based on the positions of new vertices).

This process produces a denser mesh than the original one, containing more polygonal faces. This resulting mesh can be passed through the same refinement scheme again and so on.

The limit subdivision surface is the surface produced from this process being iteratively applied infinitely many times. In practical use however, this algorithm is only applied a limited number of times. The limit surface can also be calculated directly for most subdivision surfaces using the technique of Jos Stam, which eliminates the need for recursive refinement.

3.5.1 Editing a subdivision surface

Subdivision surfaces can be naturally edited at different levels of subdivision. Starting with basic shapes you can use binary operators to create the correct topology. Then edit the coarse mesh to create the basic shape, then edit the offsets for the next subdivision step, then repeat this at finer and finer levels. You can always see how your edit effect the limit surface via GPU evaluation of the surface.

A surface designer may also start with a scanned in object or one created from a NURBS surface. The same basic optimization algorithms are used to create a coarse base mesh with the correct topology and then add details at each level so that the object may be edited at different levels. These types of surfaces may be difficult to work with because the base mesh does not have control points in the locations that a human designer would place them. With a scanned object this surface is easier to work with than a raw triangle mesh, but a NURBS object probably had well laid out control points which behave less intuitively after the conversion than before.

Non-uniform rational basis spline (NURBS) is a mathematical model commonly used in computer graphics for generating and representing curves and surfaces which offers great flexibility and precision for handling both analytic and freeform shapes.

A surface under construction, e.g. the hull of a motor yacht, is usually composed of several NURBS surfaces known as *patches*. These patches should be fitted together in such a way that the boundaries are invisible. This is mathematically expressed by the concept of geometric continuity.

3.5.3 Usefulness of NURBS curves and surfaces

1. They are invariant under affine as well as perspective transformations: operations like rotations and translations can be applied to NURBS curves and surfaces by applying them to their control points.
2. They offer one common mathematical form for both standard analytical shapes (e.g., conics) and free-form shapes.
3. They provide the flexibility to design a large variety of shapes.
4. They reduce the memory consumption when storing shapes (compared to simpler methods).
5. They can be evaluated reasonably quickly by numerically stable and accurate algorithms.

3.5.4 Control points of Bezier curves

The control points determine the shape of the curve. Typically, each point of the curve is computed by taking a weighted sum of a number of control points. The weight of each point varies according to the governing parameter. For a curve of degree d, the weight of any control point is only nonzero in d+1 intervals of the parameter space. Within those intervals, the weight changes according to a polynomial function (*basis functions*) of degree d. At the boundaries of the intervals, the basis functions go smoothly to zero, the smoothness being determined by the degree of the polynomial.

As an example, the bases function of degree one is a triangle function. It rises from zero to one, then falls to zero again. While it rises, the basis function of the previous control point falls. In that way, the curve interpolates between the two points, and the resulting curve is a polygon, which is continuous, but not differentiable at the interval boundaries, or knots. Higher degree polynomials have correspondingly more continuous derivatives. Note that within the interval the polynomial nature of the basis functions and the linearity of the construction make the curve perfectly smooth, so it is only at the knots that discontinuity can arise. The fact that a single control point only influences those intervals where it is active is a highly desirable property,

known as local support. In modeling, it allows the changing of one part of a surface while keeping other parts equal.

Adding more control points allows better approximation to a given curve, although only a certain class of curves can be represented exactly with a finite number of control points. NURBS curves also feature a scalar weight for each control point. This allows for more control over the shape of the curve without unduly raising the number of control points. In particular, it adds conic sections like circles and ellipses to the set of curves that can be represented exactly. The term *rational* in NURBS refers to these weights.

The control points can have any dimensionality. One-dimensional points just define a scalar function of the parameter. These are typically used in image processing programs to tune the brightness and colour curves. Three-dimensional control points are used abundantly in 3D modeling, where they are used in the everyday meaning of the word 'point', a location in 3D space. Multi-dimensional points might be used to control sets of time-driven values, e.g. the different positional and rotational settings of a robot arm. NURBS surfaces are just an application of this. Each control 'point' is actually a full vector of control points, defining a curve. These curves share their degree and the number of control points, and span one dimension of the parameter space. By interpolating these control vectors over the other dimension of the parameter space, a continuous set of curves is obtained, defining the surface.

3.5.5 The knot vector

The knot vector is a sequence of parameter values that determines where and how the control points affect the NURBS curve. The number of knots is always equal to the number of control points plus curve degree plus one. The knot vector divides the parametric space in the intervals mentioned before, usually referred to as *knot spans*. Each time the parameter value enters a new knot span, a new control point becomes active, while an old control point is discarded. It follows that the values in the knot vector should be in non-decreasing order, so (0, 0, 1, 2, 3, 3) is valid while (0, 0, 2, 1, 3, 3) is not.

Consecutive knots can have the same value. This then defines a knot span of zero length, which implies that two control points are activated at the same time (and of course two control points become deactivated). This has impact on continuity of the resulting curve or its higher derivatives; for instance, it allows the creation of corners in an otherwise smooth NURBS curve. A number of coinciding knots is sometimes referred to as a knot with a certain multiplicity. Knots with multiplicity two or three are known as double or triple knots. The multiplicity of a knot is limited to the degree of the curve; since a higher multiplicity would split the curve into disjoint parts and it would leave control points unused. For first-degree NURBS, each knot is paired with a control point.

The knot vector usually starts with a knot that has multiplicity equal to the order. This makes sense, since this activates the control points that have influence on the first knot span. Similarly, the knot vector usually ends with a knot of that multiplicity. Curves with such knot vectors start and end in a control point.

The individual knot values are not meaningful by themselves; only the ratios of the difference between the knot values matter. Hence, the knot vectors (0, 0, 1, 2, 3, 3) and (0, 0, 2, 4, 6, 6) produce the same curve. The positions of the knot values influences the mapping of parameter space to curve space. Rendering a NURBS curve is usually done by stepping with a fixed stride through the parameter range. By changing the knot span lengths, more sample points can be used in regions where the curvature is high. Another use is in situations where the parameter value has some physical significance, for instance if the parameter is time and the curve describes the motion of a robot arm. The knot span lengths then translate into velocity and acceleration, which are essential to get right to prevent damage to the robot arm or its environment. This flexibility in the mapping is what the phrase *non uniform* in NURBS refers to.

Necessary only for internal calculations, knots are usually not helpful to the users of modeling software. Therefore, many modeling applications do not make the knots editable or even visible. It's usually possible to establish reasonable knot vectors by looking at the variation in the control points. More recent versions of NURBS software (e.g., Autodesk Maya and Rhinoceros 3D) allow for interactive editing of knot positions, but this is significantly less intuitive than the editing of control points.

3.5.6 Order of NURBS

The *order* of a NURBS curve defines the number of nearby control points that influence any given point on the curve. The curve is represented mathematically by a polynomial of degree one less than the order of the curve. Hence, second-order curves (which are represented by linear polynomials) are called linear curves, third-order curves are called quadratic curves, and fourth-order curves are called cubic curves. The number of control points must be greater than or equal to the order of the curve.

In practice, cubic curves are the ones most commonly used. Fifth- and sixth-order curves are sometimes useful, especially for obtaining continuous higher order derivatives, but curves of higher orders are practically never used because they lead to internal numerical problems and tend to require disproportionately large calculation times.

3.5.6.1 Construction of the basis functions

The basis functions used in NURBS curves are usually denoted as $N_{i,n}(u)$, in which i corresponds to the i-th control point, and n corresponds with the degree of the basis function. The parameter

dependence is frequently left out, so we can write $N_{i,n}$. The definition of these basis functions is recursive in n. The degree-0 functions $N_{i,0}$ are piecewise constant functions. They are one on the corresponding knot span and zero everywhere else. Effectively, $N_{i,n}$ is a linear interpolation of $N_{i,n-1}$ and $N_{i+1,n-1}$. The latter two functions are non-zero for n knot spans, overlapping for $n-1$ knot spans. The function $N_{i,n}$ is computed as

From bottom to top: Linear basis functions $N_{1,1}$ (blue) and $N_{2,1}$ (green), their weight functions f and g and the resulting quadratic basis function. The knots are 0, 1, 2 and 2.5

$$N_{i,n} = f_{i,n}N_{i,n-1} + g_{i+1,n}N_{i+1,n-1}$$

f_i rises linearly from zero to one on the interval where $N_{i,n-1}$ is non-zero, while g_{i+1} falls from one to zero on the interval where $N_{i+1,n-1}$ is non-zero. As mentioned before, $N_{i,1}$ is a triangular function, nonzero over two knot spans rising from zero to one on the first, and falling to zero on the second knot span. Higher order basis functions are non-zero over corresponding more knot spans and have correspondingly higher degree. If u is the parameter, and k_i is the i-th knot, we can write the functions f and g as

$$f_{i,n}(u) = \frac{u - k_i}{k_{i+n} - k_i}$$

and

$$g_{i,n}(u) = \frac{k_{i+n} - u}{k_{i+n} - k_i}$$

The functions f and g are positive when the corresponding lower order basis functions are non-zero. By induction on n it follows that the basis functions are non-negative for all values of n and u. This makes the computation of the basis functions numerically stable.

Again by induction, it can be proved that the sum of the basis functions for a particular value of the parameter is unity. This is known as the partition of unity property of the basis functions.

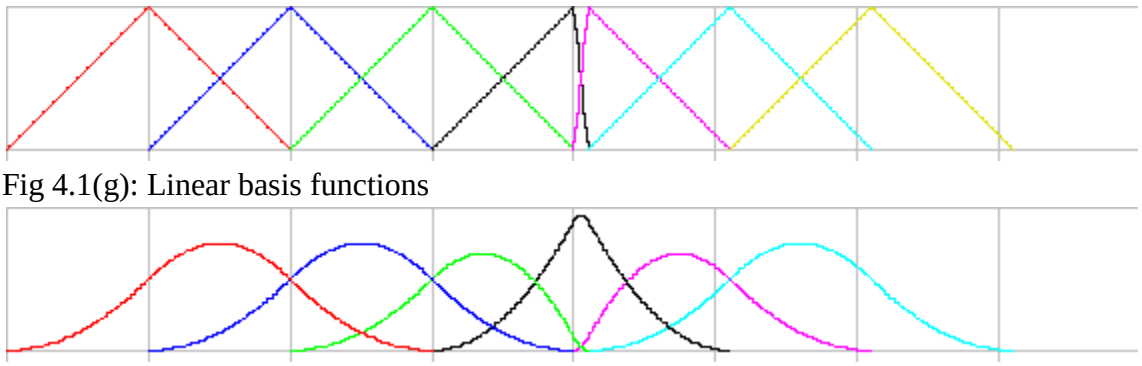


Fig 4.1(g): Linear basis functions

Fig 4.1(h): Quadratic basis functions

The figures show the linear and the quadratic basis functions for the knots {..., 0, 1, 2, 3, 4, 4.1, 5.1, 6.1, 7.1, ...}

One knot span is considerably shorter than the others. On that knot span, the peak in the quadratic basis function is more distinct, reaching almost one. Conversely, the adjoining basis functions fall to zero more quickly. In the geometrical interpretation, this means that the curve approaches the corresponding control point closely. In case of a double knot, the length of the knot span becomes zero and the peak reaches one exactly. The basis function is no longer differentiable at that point. The curve will have a sharp corner if the neighbour control points are not collinear.

3.5.7 General form of a NURBS curve

Using the definitions of the basis functions $N_{i,n}$ from the previous paragraph, a NURBS curve takes the following form:

$$C(u) = \sum_{i=1}^{k} \frac{N_{i,n} w_i}{\sum_{j=1}^{k} N_{j,n} w_j} \mathbf{P}_i = \frac{\sum_{i=1}^{k} N_{i,n} w_i \mathbf{P}_i}{\sum_{i=1}^{k} N_{i,n} w_i}$$

In this, k is the number of control points $\mathbf{P}_i$ and w_i are the corresponding weights. The denominator is a normalizing factor that evaluates to one if all weights are one. This can be seen from the partition of unity property of the basis functions. It is customary to write this as

$$C(u) = \sum_{i=1}^{k} R_{i,n} \mathbf{P}_i$$

in which the functions

$$R_{i,n} = \frac{N_{i,n} w_i}{\sum_{j=1}^{k} N_{j,n} w_j}$$

are known as the *rational basis functions*.

3.5.8. Manipulating NURBS objects

A number of transformations can be applied to a NURBS object. For instance, if some curve is defined using a certain degree and N control points, the same curve can be expressed using the same degree and N+1 control points. In the process a number of control points change position and a knot is inserted in the knot vector. These manipulations are used extensively during interactive design. When adding a control point, the shape of the curve should stay the same, forming the starting point for further adjustments. A number of these operations are discussed below.

3.5.9 Knot Operations

3.5.9.1 Knot insertion

As the term suggests, knot insertion inserts a knot into the knot vector. If the degree of the curve is *n*, then $n - 1$ control points are replaced by *n* new ones. The shape of the curve stays the same. A knot can be inserted multiple times, up to the maximum multiplicity of the knot. This is sometimes referred to as knot refinement and can be achieved by an algorithm that is more efficient than repeated knot insertion.

3.5.9.2 Knot removal

Knot removal is the reverse of knot insertion. Its purpose is to remove knots and the associated control points in order to get a more compact representation. Obviously, this is not always possible while retaining the exact shape of the curve. In practice, a tolerance in the accuracy is used to determine whether a knot can be removed. The process is used to clean up after an interactive session in which control points may have been added manually, or after importing a curve from a different representation, where a straightforward conversion process leads to redundant control points.

3.5.9.3 Degree elevation

A NURBS curve of a particular degree can always be represented by a NURBS curve of higher degree. This is frequently used when combining separate NURBS curves, e.g. when creating a NURBS surface interpolating between a set of NURBS curves or when unifying adjacent curves. In the process, the different curves should be brought to the same degree, usually the maximum degree of the set of curves. The process is known as degree elevation.

3.5.9.4 Curvature

The most important property in differential geometry is the curvature κ. It describes the local properties (edges, corners, etc.) and relations between the first and second derivative, and thus, the precise curve shape. Having determined the derivatives it is easy to compute the curvature *k* of shapes

$$\kappa = \frac{|r'(t) \times r''(t)|}{|r'(t)|^3}$$

or approximated as the arclength from the second derivate $\kappa = | r''(s_o) |$. The direct computation of the curvature κ with these equations is the big advantage of parameterized curves against their polygonal representations.

4.0 Conclusion

Editing NURBS curves and surfaces is highly intuitive and predictable. Control points are always either connected directly to the curve/surface or act as if they were connected by a rubber band. Depending on the type of user interface, editing can be realized via an element's control points, which are most obvious and common for Bézier curves, or via higher level tools such as spline modeling or hierarchical editing.

5.0 Summary

Curves are used when creating 3D models, vector images, animations. There is a great variety of curves. Some are easy to use, some are flexible enough to describe a large variety of shapes, and some are simple enough to be implemented and accelerated by graphics hardware. Bézier curves are widely used in computer graphics to model smooth curves. A B-Spline consists of multiple Bézier arcs and provides a unified mechanism how to define continuity in the joins.

6.0 Tutor Marked Assignment

1. What do you understand by Curves and surfaces?
2. Identify Applications of Bezier curves.
3. State the properties of Bezier curves
4. Highlight the usefulness of NURB curves and surfaces.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247

3. Paul Bourke: *Bézier Surfaces (in 3D)*, 1996 http://local.wasp.uwa.edu.au/~pbourke/geometry/bezier/index.html
4. Donald Knuth: *Metafont: the Program*, Addison-Wesley 1986, pp. 123–131.
5. Dr Thomas Sederberg, BYU *Bézier curves*, 2003 http://www.tsplines.com/resources/class_notes/Bezier_curves.pdf
6. David F. Rogers: An Introduction to NURBS with Historical Perspective ISBN-13:978-1558606692.
7. Demidov, Evgeny. "NonUniform Rational B-splines (NURBS) - Perspective projection". *An Interactive Introduction to Splines*. Ibiblio. http://www.ibiblio.org/e-notes/Splines/NURBS.htm. Retrieved 2010-02-14
8. Les Piegl and Wayne Tiller: The NURBS Book. Springer-Verlag. Newyork, NY. Second Edition, 1997.

MODULE 4 – Curves and surfaces, image and the human visual system

UNIT 2: Colour theory

Contents **Pages**

1.0 Introduction

Colour theory is a body of practical guidance to colour mixing and the visual impacts of specific colour combinations. A tradition of "colour theory" began in the 18th century, initially within a partisan controversy around Isaac Newton's theory of colour and the nature of so-called primary colours. From there it developed as an independent artistic tradition with only superficial reference to colourimetry and vision science.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the properties of light and the human vision
2. Understand the various colour theories.
3. Differentiate the various colour models.

3.0 Main Content

3.1 Definition of Light

Light is electromagnetic radiation that is visible to the human eye; and is responsible for the sense of sight. Light travels in a straight line under normal circumstances - i.e. travelling through a uniform medium. Our visual systems rely heavily on this fact, 'back-projecting' rays that enter our eyes, to the probable origin of the light rays. Objects that we see around us can usually be assumed to be where they appear to be, as long as the light from them has travelled to our eyes in a straight line. However, the following phenomena can alter the path or nature of the light. The remaining part of this section discusses the properties of light.

1. **Absorption of light**

Light falling on an object may be absorbed, transmitted, or reflected. What happens to it depends on the colour of the object: a red object reflects red light and absorbs much of the rest of the other colours that we see. The colour of an object is that colour which is reflected rather than absorbed.

2. **Reflection of light**

Reflection of light is the most familiar property of light, since it is what enables us to see objects around us. Light from a source, such as the Sun, or a lamp, travels in a straight line until it strikes an object, at which point it may be absorbed, transmitted, or reflected.

Reflection occurs when waves encounter a boundary that does not absorb the radiation's energy and bounces the waves off the surface. The incoming light wave is referred to as an *incident wave* and the wave that is bounced from the surface is called the *reflected wave*. Those surfaces which reflect the most light appear white, or silver. A highly polished, smooth and flat silver surface acts as a *mirror*, reflecting a perfect image of the world around it.

3. **Refraction of light**

Light that is transmitted through a medium will usually be deviated somewhat from the straight path it was previously following. This phenomenon is familiar with transparent objects such as glasses and lenses - objects seen through them appear larger, smaller, or distorted. Place a stick partially into water and it appears to be bent at the surface.

Refraction is an important characteristic of lenses that allows them to focus a beam of light onto a single point. Refraction occurs as light passes from a one medium to another when there is a difference in the *index of refraction* between the two materials.

4. **Interference of light**

Interference is the net effect of the combination of two or more wave trains moving on intersecting or coincident paths. The effect is that of the addition of the amplitudes of the individual waves at each point affected by more than one wave.

If two of the components are of the same frequency and phase (i.e., they vibrate at the same rate and are maximum at the same time), the wave amplitudes are reinforced, producing constructive interference; but, if the two waves are out of phase by 1/2 period (i.e., one is minimum when the other is maximum), the result is destructive interference, producing complete annulment if they are of equal amplitude. One of the best examples of interference is demonstrated by the light reflected from a film of oil floating on water or a soap bubble, which reflects a variety of beautiful colours when illuminated by natural or artificial light sources.

5. **Diffraction of light**

Diffraction occurs when a light wave passes by a corner or through an opening or slit that is physically the approximate size of, or even smaller than, that light's wavelength. This is a specialized case of light scattering in which an object with regularly repeating features (such as a diffraction grating) produces an orderly diffraction of light in a diffraction pattern. In the real world, most objects are very complex in shape and should be considered to be composed of many individual diffraction features that can collectively produce a random scattering of light.

6. **Polarization of light**

Natural sunlight and most forms of artificial illumination transmit light waves whose electric field vectors vibrate in all perpendicular planes with respect to the direction of propagation. When the electric field vectors are restricted to a single plane by filtration then the light is said to be polarized with respect to the direction of propagation and all waves vibrate in the same plane.

3.2 Colour concepts

Understanding colour on the computer can be a bit tricky since the computer makes use of more than one type of colour model. There are two types of colour spaces - additive and subtractive colours. Additive colours are the colours that are inherent in light. They are the colours indigenous to monitors, digital cameras, and scanners. The human eyes, also, sees these colours. They are red, green, and blue (RGB). They are called primary additive colours because when added together, they form white.

Subtractive colours are those used in the printing trades such as dyes, inks, and pigments. These colours are cyan, magenta, and yellow. These colours are called absorbing or subtractive colours since when light is absorbed by all of them, they produce black. These are the colours used in printers. They are commonly referred to as CMY colours. However, usually one sees CMYK. K stands for a truer black since when 100% of C, M, and Y are combined; the resulting colour is a muddy dark brown. All of these colours are related.

For example:

white = red + green + blue	black = cyan + magenta + yellow
cyan = green + blue	red = yellow + magenta
magenta = blue + red	green = yellow + cyan
yellow = red + green	blue = cyan + magenta

RGB and CMY colours form a complementary relationship.

Red.....Cyan
Green.....Magenta
Blue.....Yellow

Computers are capable of generating images of varying numbers of colours. These can range from 16 colours to 16 million colours. If an image is created with only 16 colours and this image is turned into a grayscale image, there are only a certain number of shades of gray possible. Likewise, if an image that has 16 million colours is turned into a grayscale image, there are many more shades of gray even though the human eye cannot discern them all. How can the computer vary the number of colours or shades of gray? The computer monitor is made up of red, green, and blue phosphors or light producing elements. Each of the colours associated with a pixel (picture element) can have attributed to it a certain number of colours. Bit-depth determines how many colours or levels of gray each pixel carries. In scanning for example, a bit-depth of 24 means that the red, green, and blue sources of light each have 8 bits of colour assigned to them. The more colour bits assigned to a pixel, the more colours can be displayed and, theoretically, the more shades of gray. However, the human eye can only see a certain number of shades of gray. Also, not all visual colours can be transmitted to the printed medium. Thus, for example, when a scanner advertises that it can produce a bit-depth of 36, this bit-depth will not necessarily produce a better image than one with a bit-depth of 30 or even 24.

There are other models for assigning colours to an image:

HSB = Hue, Saturation, & Brightness
HSL=Hue Saturation, & Lightness
HSV=Hue, Saturation, & Value Colour Space

Hue = The colour of something
Saturation = The strength of a colour
Value = How dark or light is a colour

3.2.1 Additive colour

An additive colour model involves light emitted directly from a source or illuminant of some sort. The additive reproduction process usually uses red, green and blue light to produce the other colours. Combining one of these additive primary colours with another in equal amounts produces the additive secondary colours cyan, magenta, and yellow. Combining all three primary lights (colours) in equal intensities produces white. Varying the luminosity of each light (colour) eventually reveals the full gamut of those three lights (colours).

Computer monitors and televisions are the most common form of additive light. The coloured pixels do not overlap on the screen, but when viewed from a sufficient distance, the light from the pixels diffuses to overlap on the retina. Another common use of additive light is the projected light used in theatrical lighting, such as plays, concerts, circus shows, and night clubs.

Results obtained when mixing additive colours are often counterintuitive for people accustomed to the more everyday subtractive colour system of pigments, dyes, inks and other substances which present colour to the eye by reflection rather than emission. For example, in subtractive colour systems green is a combination of yellow and blue; in additive colour, red + green = yellow and no simple combination will yield green. Additive colour is a result of the way the eye detects colour, and is not a property of light. There is a vast difference between yellow light, with a wavelength of approximately 580 nm, and a mixture of red and green light. However, both stimulate our eyes in a similar manner, so we do not detect that difference.

3.2.2 Subtractive colour

A subtractive colour model explains the mixing of paints, dyes, inks, and natural colourants to create a full range of colours, each caused by subtracting (that is, absorbing) some wavelengths of light and reflecting the others. The colour that a surface displays depends on which colours of the electromagnetic spectrum are reflected by it and therefore made visible. Subtractive colour systems start with light, presumably white light. Coloured inks, paints, or filters between the viewer and the light source or reflective surface *subtract* wavelengths from the light, giving it colour. If the incident light is other than white, our visual mechanisms are able to compensate well, but not perfectly, often giving a flawed impression of the "true" colour of the surface.

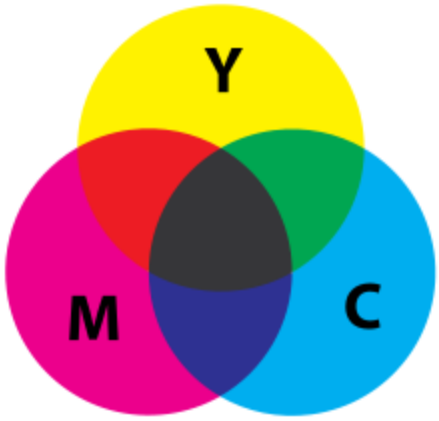


figure 4.2(a): Additive Colour system

Conversely, additive colour systems start without light (black). Light sources of various wavelengths combine to make a colour. In either type of system, three primary colours are combined to stimulate humans' trichromatic colour vision, sensed by the three types of cone cells in the eye, giving an apparently full range.

3.3 RGB colour model

The RGB colour model is an additive colour model in which red, green, and blue light is added together in various ways to reproduce a broad array of colours. The name of the model comes from the initials of the three additive primary colours, red, green, and blue.

The main purpose of the RGB colour model is for the sensing, representation, and display of images in electronic systems, such as televisions and computers, though it has also been used in conventional photography. Before the electronic age, the RGB colour model already had a solid theory behind it, based in human perception of colours.

RGB is a *device-dependent* colour model: different devices detect or reproduce a given RGB value differently, since the colour elements (such as phosphors or dyes) and their response to the individual R, G, and B levels vary from manufacturer to manufacturer, or even in the same device over time. Thus an RGB value does not define the same *colour* across devices without some kind of colour management.

Typical RGB input devices are colour TV and video cameras, image scanners, and digital cameras. Typical RGB output devices are TV sets of various technologies (CRT, LCD, plasma, etc.), computer and mobile phone displays, video projectors, multicolour LED displays, and large screens such as JumboTron, etc. Colour printers, on the other hand, are not RGB devices, but subtractive colour devices (typically CMYK colour model).

3.4 YIQ Colour space

YIQ is the colour space used by the NTSC colour TV system, employed mainly in North and Central America, and Japan. It is currently in use only for low-power television stations, as full-power analog transmission was ended by the U.S. Federal Communications Commission (FCC). It is still federally mandated for these transmissions as shown in this excerpt of the current FCC rules and regulations part 73 "TV transmission standard".

The YIQ system is intended to take advantage of human colour-response characteristics. The eye is more sensitive to changes in the orange-blue (I) range than in the purple-green range (Q) — therefore less bandwidth is required for Q than for I. Broadcast NTSC limits I to 1.3 MHz and Q to 0.4 MHz. I and Q are frequency interleaved into the 4 MHz Y signal, which keeps the bandwidth of the overall signal down to 4.2 MHz. In YUV systems, since U and V both contain information in the orange-blue range, both components must be given the same amount of bandwidth as I to achieve similar colour fidelity.

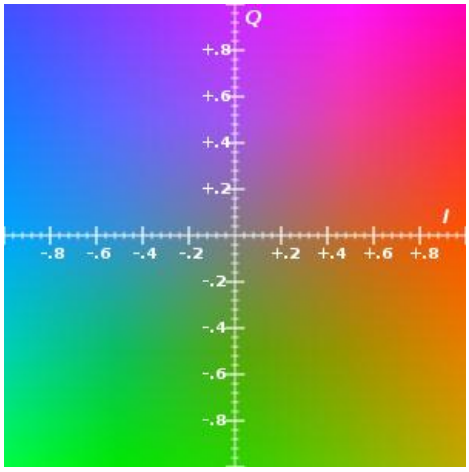


Figure 4.2(b): YIQ colour space

Very few television sets perform true I and Q decoding, due to the high costs of such an implementation. Compared to the cheaper R-Y and B-Y decoding which requires only one filter, I and Q each requires a different filter to satisfy the bandwidth differences between I and Q. These bandwidth differences also requires that the 'I' filter include a time delay to match the longer delay of the 'Q' filter. The Rockwell Modular Digital Radio (MDR) was one I and Q decoding set, which in 1997 could operate in frame-at-a-time mode with a PC or in realtime with the Fast IQ Processor (FIQP). Some RCA "Colourtrak" home TV receivers made circa 1985 not only used I/Q decoding, but also advertised its benefits along with its comb filtering benefits as full "100 percent processing" to deliver more of the original colour picture content. Earlier, more than one brand of colour TV (RCA, Arvin) used I/Q decoding in the 1954 or 1955 model year on models utilizing screens about 13 inches (measured diagonally). The original Advent projection television used I/Q decoding. Around 1990, at least one manufacturer (Ikegami) of professional studio picture monitors advertised I/Q decoding.

3.5 CMYK colour model

The CMYK colour model (process colour, four colour) is a subtractive colour model, used in colour printing, and is also used to describe the printing process itself. CMYK refers to the four inks used in some colour printing: cyan, magenta, yellow, and key (black). Though it varies by print house, press operator, press manufacturer and press run, ink is typically applied in the order of the abbreviation.

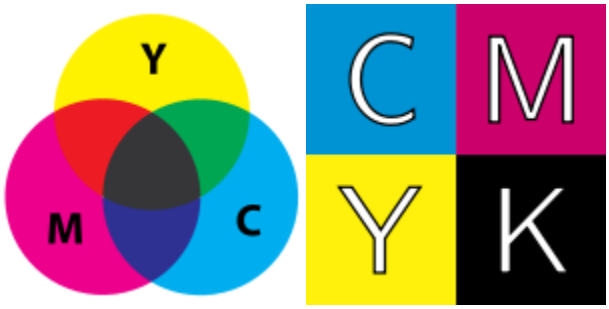


Figure 4.2(c): CYMK colour model

The "K" in CMYK stands for *key* since in four-colour printing cyan, magenta, and yellow printing plates are carefully *keyed* or aligned with the *key* of the black *key plate*. Some sources suggest that the "K" in CMYK comes from the last letter in "black" and was chosen because B already means blue. However, this explanation, though plausible and useful as a mnemonic, is incorrect.

The CMYK model works by partially or entirely masking colours on a lighter, usually white, background. The ink reduces the light that would otherwise be reflected. Such a model is called *subtractive* because inks "subtract" brightness from white.

In additive colour models such as RGB, white is the "additive" combination of all primary coloured lights, while black is the absence of light. In the CMYK model, it is the opposite: white is the natural colour of the paper or other background, while black results from a full combination of coloured inks. To save money on ink, and to produce deeper black tones, unsaturated and dark colours are produced by using black ink instead of the combination of cyan, magenta and yellow.

3.6 HSL and HSV

HSL stands for *hue*, *saturation*, and *lightness*, and is often also called HLS. HSV stands for *hue*, *saturation*, and *value*, and is also often called HSB (*B* for *brightness*). A third model, common in computer vision applications, is HSI, for *hue*, *saturation*, and *intensity*. Unfortunately, while typically consistent, these definitions are not standardized, and any of these abbreviations might be used for any of these three or several other related cylindrical models.

HSL and HSV are the two most common cylindrical-coordinate representations of points in an RGB colour model, which rearrange the geometry of RGB in an attempt to be more intuitive and perceptually relevant than the cartesian (cube) representation. They were developed in the 1970s for computer graphics applications, and are used for colour pickers, in colour-modification tools in image editing software, and less commonly for image analysis and computer vision.

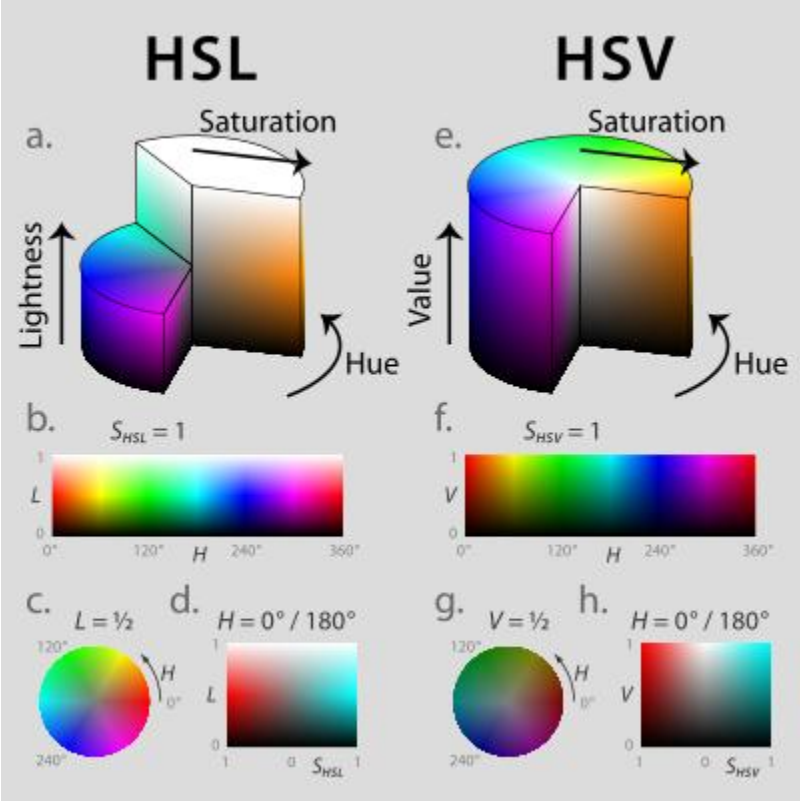


Figure 4.2 (d): HSL and HSV Colour models

In each cylinder, the angle around the central vertical axis corresponds to "hue", the distance from the axis corresponds to "saturation", and the distance along the axis corresponds to "lightness", "value" or "brightness". Note that while "hue" in HSL and HSV refers to the same attribute, their definitions of "saturation" differ dramatically. Because HSL and HSV are simple transformations of device-dependent RGB models, the physical colours they define depend on the colours of the red, green, and blue primaries of the device or of the particular RGB space, and on the gamma correction used to represent the amounts of those primaries. Each unique RGB device therefore has unique HSL and HSV spaces to accompany it, and numerical HSL or HSV values describe a different colour for each basis RGB space.

Both of these representations are used widely in computer graphics, and one or the other of them is often more convenient than RGB, but both are also criticized for not adequately separating colour-making attributes, or for their lack of perceptual uniformity. Other more computationally intensive models, such as CIELAB or CIECAM02 better achieve these goals.

4.0 Conclusion

Understanding colour on the computer can be a bit tricky since the computer makes use of more than one type of colour model. Therefore, a deep understanding of the various colour models is necessary in order to produce realistic colours on computer graphics

5.0 Summary

Colour theory is a body of practical guidance to colour mixing and the visual impacts of specific colour combinations. Subtractive colours are those used in the printing trades such as dyes, inks, and pigments while Additive colours are the colours that are inherent in light. . RGB, YIQ, HSL, HSV and CYMK are colour models discussed in the unit with varying applications.

6.0 Tutor Marked Assignment

1. What do you understand by colour theory?
2. Explain the following colour models
    1. RGB colour model
    2. YIQ colour space
    3. CYMK colour Model
    4. HSV and HSL colour models
3. Identify the properties of light and relate them to colour models discussed above.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. O'Connor, Z. (2010). Colour harmony revisited. Colour Research and Application, 35 (4), pp267-273.
4. Pointer, M. R. & Attridge, G.G. (1998). The number of discernible colours. Colour Research and Application, 23 (1), pp52-54.
5. Hard, A. & Sivik, L. (2001). A theory of colours in combination - A descriptive model related to the NCS colour-order system. Colour Research and Application, 26 (1), pp4-28.
6. Feisner, E. A. (2000). Colour: How to use colour in art and design. London: Laurence King.
7. Mahnke, F. (1996). Colour, Environment and Human response. New York: John Wiley & Sons. ISBN-13: 978-0471286677

MODULE 4 – Curves and surfaces, image and the human visual system

Unit 3 - Pixels and images, Vision and colour, HDR images and Optical illusion

Contents **Pages**

1.0 Introduction

A pixel (short for picture element) is one of the many tiny *dots* that make up the representation of a picture in a computer's memory. Each such information element is not really a dot, nor a square, but an abstract sample. With care, pixels in an image can be reproduced at any size

without the appearance of visible dots or squares; but in many contexts, they are reproduced as dots or squares and can be visibly distinct when not fine enough. The intensity of each pixel is variable; in colour systems, each pixel has typically three or four dimensions of variability such and Red, Green and Blue, or Cyan, Magenta, Yellow and Black.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand what makes up a pixel.
2. Understand the relationship between vision and colour.
3. Understand alpha compositing and Image anti-aliasing.

3.0 Main Content

3.1 Definition of a Pixel

A pixel is generally thought of as the smallest complete sample of an image. The definition is highly context sensitive. For example, we can speak of pixels in a visible image (e.g. a printed page) or pixels carried by one or more electronic signal(s), or represented by one or more digital value(s), or pixels on a display device, or pixels in a digital camera (photosensor elements). This list is not exhaustive and depending on context there are several synonyms which are accurate in particular contexts, e.g. pel, sample, bytes, bits, dots, spots, superset, triad, stripe set, window, etc. We can also speak of pixels in the abstract, in particular when using pixels as a measure of resolution, e.g. 2400 pixels per inch or 640 pixels per line. Dots is often used to mean pixels, especially by computer sales and marketing people, and gives rise to the abbreviation DPI or dots per inch.

The colour samples that form a digitized image (such as a JPG file used on a web page) are also called pixels. Depending on how a computer displays an image, these may not be in one-to-one correspondence with screen pixels. In areas where the distinction is important, the dots in the image file may be called texels.

In computer programming, an image composed of pixels is known as a *bitmapped image* or a *raster image*. The word *raster* originates from analogue television technology. Bitmapped images are used to encode digital video and to produce some types of computer-generated art. In digital imaging, a pixel, or pel, (picture element) is a single point in a raster image, or the smallest addressable screen element in a display device; it is the smallest unit of picture that can be represented or controlled.

Each pixel has its own address. The address of a pixel corresponds to its coordinates. Pixels are normally arranged in a two-dimensional grid, and are often represented using dots or squares. Each pixel is a sample of an original image; more samples typically provide more accurate representations of the original. The intensity of each pixel is variable. In colour image systems, a

colour is typically represented by three or four component intensities such as red, green, and blue, or cyan, magenta, yellow, and black.

In some contexts (such as descriptions of camera sensors), the term *pixel* is used to refer to a single scalar element of a multi-component representation (more precisely called a *photosite* in the camera sensor context, although the neologism *sensel* is sometimes used to describe the elements of a digital camera's sensor), while in others the term may refer to the entire set of such component intensities for a spatial position. In colour systems that use chroma subsampling, the multi-component concept of a pixel can become difficult to apply, since the intensity measures for the different colour components correspond to different spatial areas in such a representation.

3.2 Vision and Colour

Colour vision is the capacity of an organism or machine to distinguish objects based on the wavelengths (or frequencies) of the light they reflect, emit, or transmit. Colours can be measured and quantified in various ways; indeed, a human's perception of colours is a subjective process whereby the brain responds to the stimuli that are produced when incoming light reacts with the several types of cone photoreceptors in the eye.

3.3 Alpha Compositing

In computer graphics, alpha compositing is the process of combining an image with a background to create the appearance of partial or full transparency. It is often useful to render image elements in separate passes, and then combine the resulting multiple 2D images into a single, final image in a process called compositing. For example, compositing is used extensively when combining computer rendered image elements with live footage. In order to combine these image elements correctly, it is necessary to keep an associated *matte* for each element. This matte contains the coverage information—the shape of the geometry being drawn—making it possible to distinguish between parts of the image where the geometry was actually drawn and other parts of the image which are empty.

To store matte information, the concept of an alpha channel was introduced by Alvy Ray Smith in the late 1970s, and fully developed in a 1984 paper by Thomas Porter and Tom Duff. In a 2D image element, which stores a colour for each pixel, additional data is stored in the alpha channel with a value between 0 and 1. A value of 0 means that the pixel does not have any coverage information and is transparent; i.e. there was no colour contribution from any geometry because the geometry did not overlap this pixel. A value of 1 means that the pixel is opaque because the geometry completely overlapped the pixel. If an alpha channel is used in an image, it is common to also multiply the colour by the alpha value, to save on additional multiplications during compositing. This is usually referred to as *premultiplied alpha*. Assuming that the pixel colour is expressed using *straight* (non-premultiplied) RGBA tuples, a pixel value of (0.0, 0.5, 0.0, 0.5)

implies a pixel which has 50% of the maximum green intensity and 50% opacity. If the colour were fully green, its RGBA would be (0, 1, 0, 0.5).

However, if this pixel uses premultiplied alpha, all of the RGB values (0, 1, 0) are multiplied by 0.5 and then the alpha is appended to the end to yield (0, 0.5, 0, 0.5). In this case, the 0.5 value for the G channel actually indicates 100% green intensity (with 50% opacity). For this reason, knowing whether a file uses premultiplied or straight alpha is essential to correctly process or composite it. Premultiplied alpha has some practical advantages over normal alpha blending because premultiplied alpha blending is associative and linear interpolation gives better results, although premultiplication can cause a loss of precision and, in extreme cases, a noticeable loss of quality.

With the existence of an alpha channel, it is possible to express compositing image operations, using a *compositing algebra*. For example, given two image elements A and B, the most common compositing operation is to combine the images such that A appears in the foreground and B appears in the background. This can be expressed as A over B. In addition to over, Porter and Duff (1984) defined the compositing operators in, held out by (usually abbreviated out), atop, and xor (and the reverse operators rover, rin, rout, and ratop) from a consideration of choices in blending the colours of two pixels when their coverage is, conceptually, overlaid orthogonally:

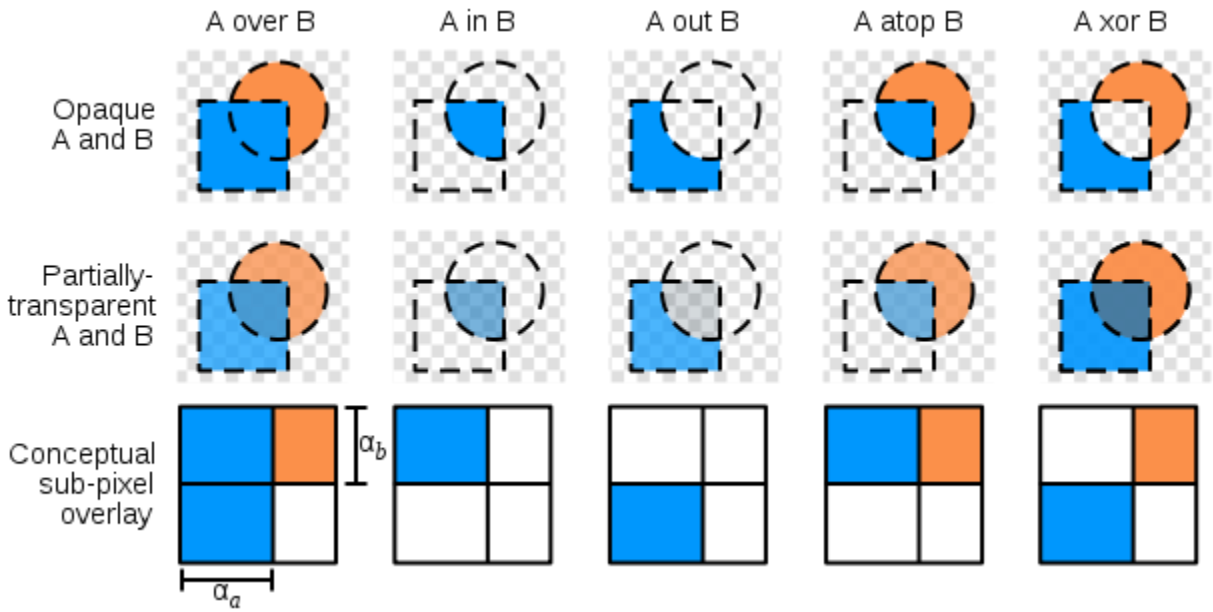


Figure 4.3(a): Alpha compositing of image A and B

The over operator is, in effect, the normal painting operation. The in operator is the alpha compositing equivalent of clipping.

As an example, the over operator can be accomplished by applying the following formula to each pixel value:

$$C_o = C_a \alpha_a + C_b \alpha_b \left(1 - \alpha_a\right)$$

where C_o is the result of the operation, C_a is the colour of the pixel in element A, C_b is the colour of the pixel in element B, and α_a and α_b are the alpha of the pixels in elements A and B respectively. If it is assumed that all colour values are premultiplied by their alpha values ($c_i = \alpha_i C_i$), we can rewrite the equation for output colour as:

$$C_o = c_a + \left(1 - \alpha_a\right) c_b$$

and resulting alpha channel value is

$$\alpha_o = \frac{c_o}{C_o} = \alpha_a + \alpha_b \left(1 - \alpha_a\right)$$

However, this operation may not be appropriate for all applications, since it is not associative. The associative version of this operation is very similar; simply take the newly computed colour value and divide it by its new alpha value, as follows:

$$C_o = \frac{1}{\alpha_o} \left[C_a \alpha_a + C_b \alpha_b \left(1 - \alpha_a\right)\right]$$

Image editing applications that allow merging of layers generally prefer this second approach.

3.4 Alpha blending

Alpha blending is a convex combination of two colours allowing for transparency effects in computer graphics. The value of alpha in the colour code ranges from 0.0 to 1.0, where 0.0 represents a fully transparent colour, and 1.0 represents a fully opaque colour. This corresponds to "SRC over DST" in Porter and Duff equations.

The value of the resulting colour is given by:

$$\begin{cases} out_A = src_A + dst_A(1 - src_A) \\ out_{RGB} = \left(src_{RGB} src_A + dst_{RGB} dst_A \left(1 - src_A\right)\right) \div out_A \\ out_A = 0 \Rightarrow out_{RGB} = 0 \end{cases}$$

If the destination background is opaque, then $dst_a = 1$, and if you enter it to the upper equation:

$$\begin{cases} out_A = 1 \\ out_{RGB} = src_{RGB} src_A + dst_{RGB}(1 - src_A) \end{cases}$$

The alpha component may be used to blend to red, green and blue components equally, as in 32-bit RGBA, or, alternatively, there may be three alpha values specified corresponding to each of the primary colours for spectral colour filtering.

Note that the RGB colour may be premultiplied, hence saving the additional multiplication before RGB in the equation above. This can be a considerable saving in processing time given that images are often made up of millions of pixels.

3.5 High dynamic range imaging

In image processing, computer graphics, and photography, high dynamic range imaging (HDRI or just HDR) is a set of techniques that allow a greater dynamic range between the lightest and darkest areas of an image than current standard digital imaging techniques or photographic methods. This wide dynamic range allows HDR images to more accurately represent the range of intensity levels found in real scenes, ranging from direct sunlight to faint starlight, and is often captured by way of a plurality of differently exposed pictures of the same subject matter.

The two main sources of HDR imagery are computer renderings and merging of multiple low-dynamic-range (LDR) or standard-dynamic-range (SDR) photographs. Tone-mapping techniques, which reduce overall contrast to facilitate display of HDR images on devices with lower dynamic range, can be applied to produce images with preserved or exaggerated local contrast for artistic effect.

3.5.1 Dynamic range in Photography

In photography, dynamic range is measured in Exposure value (EV) differences (known as *stops*) between the brightest and darkest parts of the image that show detail. An increase of one EV or one stop is a doubling of the amount of light.

Dynamic Ranges of Common Devices

	Device	Stops	Contrast
1.	LCD display	9.5	700:1 (250:1 - 1750:1)
2.	DSLR camera (Canon EOS-1D Mark II)	11	2048:1
3.	Print film	7	128:1
4.	Human eye	10–14	1024:1 – 16384:1

Table 4.3: Dynamic ranges of common devices

High-dynamic-range photographs are generally achieved by capturing multiple standard photographs, often using exposure bracketing, and then merging them into an HDR image. Digital photographs are often encoded in a camera's raw image format, because 8 bit JPEG encoding doesn't offer enough values to allow fine transitions (and also introduces undesirable effects due to the lossy compression).

Any camera that allows manual over- or under-exposure of a photo can be used to create HDR images. Some cameras have an auto exposure bracketing (AEB) feature with a far greater dynamic range than others, from the 3 EV of the Canon EOS 40D, to the 18 EV of the Canon EOS-1D Mark II. As the popularity of this imaging technique grows, several camera manufactures are now offering built in HDR features. For example, the Pentax K-7 DSLR has an HDR mode which captures an HDR image and then outputs (only) a tone-mapped JPEG file. The Canon PowerShot G12 and Canon PowerShot S95 offer similar features in a smaller format.

3.5.2 Comparison with traditional digital images

Information stored in high-dynamic-range images typically corresponds to the physical values of luminance or radiance that can be observed in the real world. This is different from traditional digital images, which represent colours that should appear on a monitor or a paper print. Therefore, HDR image formats are often called "scene-referred", in contrast to traditional digital images, which are "device-referred" or "output-referred". Furthermore, traditional images are usually encoded for the human visual system (maximizing the visual information stored in the fixed number of bits), which is usually called "gamma encoding" or "gamma correction". The values stored for HDR images are often gamma compressed (power law) or logarithmically encoded, or floating-point linear values, since fixed-point linear encodings are increasingly inefficient over higher dynamic ranges.

HDR images often use a higher number of bits per colour channel than traditional images to represent many more colours over a much wider dynamic range. 16-bit ("half precision") or 32-bit floating point numbers are often used to represent HDR pixels. However, when the appropriate transfer function is used, HDR pixels for some applications can be represented with as few as 10–12 bits for luminance and 8 bits for chrominance without introducing any visible quantization artifacts.

3.6 Optical Illusion (Visual Illusion)

An optical illusion (also called a visual illusion) is characterized by visually perceived images that differ from objective reality. The information gathered by the eye is processed in the brain to

give a perception that does not tally with a physical measurement of the stimulus source. There are three main types: literal optical illusions that create images that are different from the objects that make them, physiological ones that are the effects on the eyes and brain of excessive stimulation of a specific type (brightness, colour, size, position, tilt, movement), and cognitive illusions, the result of unconscious inferences.

3.6.1 Physiological illusions

Physiological illusions, such as the afterimages following bright lights, or adapting stimuli of excessively longer alternating patterns (contingent perceptual aftereffect), are presumed to be the effects on the eyes or brain of excessive stimulation or interaction with contextual or competing stimuli of a specific type—brightness, colour, position, tile, size, movement, etc. The theory is that a stimulus follows its individual dedicated neural path in the early stages of visual processing, and that intense or repetitive activity in that or interaction with active adjoining channels cause a physiological imbalance that alters perception

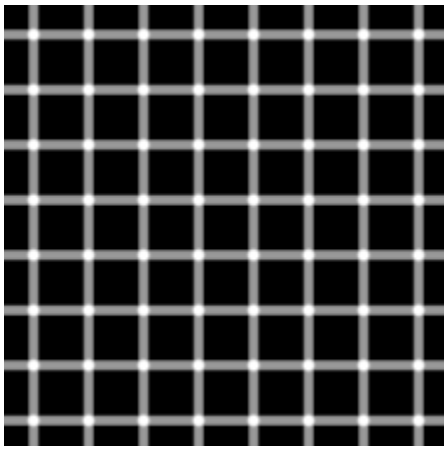


Figure 4.2 (c): A scintillating grid illusion. Shape, position, colour, and 3D contrast converge to produce the illusion of black dots at the intersections.

The Hermann grid illusion and Mach bands are two illusions that are best explained using a biological approach. Lateral inhibition, where in the receptive field of the retina light and dark receptors compete with one another to become active, has been used to explain why we see bands of increased brightness at the edge of a colour difference when viewing Mach bands. Once a receptor is active it inhibits adjacent receptors. This inhibition creates contrast, highlighting edges. In the Hermann grid illusion the gray spots appear at the intersection because of the inhibitory response which occurs as a result of the increased dark surround. Lateral inhibition has also been used to explain the Hermann grid illusion, but this has been disproved. More recent "empirical" approaches to optical illusions have had some success in explaining optical phenomena with which theories based on lateral inhibition have struggled

3.6.2 Cognitive illusions

Cognitive illusions are assumed to arise by interaction with assumptions about the world, leading to "unconscious inferences", an idea first suggested in the 19th century by Hermann Helmholtz. Cognitive illusions are commonly divided into ambiguous illusions, distorting illusions, paradox illusions, or fiction illusions.

1. Ambiguous illusions are pictures or objects that elicit a perceptual 'switch' between the alternative interpretations. The Necker cube is a well known example; another instance is the Rubin vase.
2. Distorting or geometrical-optical illusions are characterized by distortions of size, length, position or curvature. A striking example is the Café wall illusion. Other examples is the famous Müller-Lyer illusion and Ponzo illusion.
3. Paradox illusions are generated by objects that are paradoxical or impossible, such as the Penrose triangle or impossible staircases seen, for example, in M. C. Escher's *Ascending and Descending* and *Waterfall*. The triangle is an illusion dependent on a cognitive misunderstanding that adjacent edges must join.
4. Fictions are when a figure is perceived even though it is not in the stimulus.

To make sense of the world it is necessary to organize incoming sensations into information which is meaningful. Gestalt psychologists believe one way this is done is by perceiving individual sensory stimuli as a meaningful whole. Gestalt organization can be used to explain many illusions including the Duck-Rabbit illusion where the image as a whole switches back and forth from being a duck then being a rabbit and why in the figure-ground illusion the figure and ground are reversible.

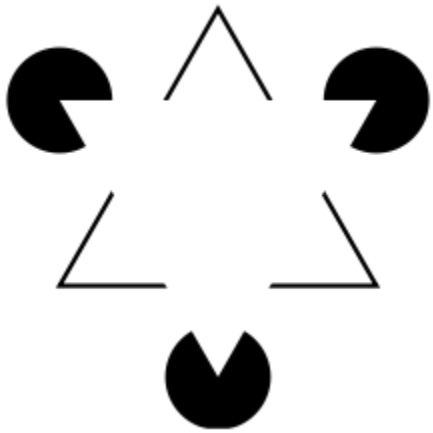


Figure 4.2 (d): Kanizsa triangle

In addition, Gestalt theory can be used to explain the illusory contours in the Kanizsa Triangle. A floating white triangle, which does not exist, is seen. The brain has a need to see familiar simple objects and has a tendency to create a "whole" image from individual elements. Gestalt means "form" or "shape" in German. However, another explanation of the Kanizsa Triangle is based in

evolutionary psychology and the fact that in order to survive it was important to see form and edges. The use of perceptual organization to create meaning out of stimuli is the principle behind other well-known illusions including impossible objects. Our brain makes sense of shapes and symbols putting them together like a jigsaw puzzle, formulating that which isn't there to that which is believable.

3.7 Depth and motion perception

Illusions can be based on an individual's ability to see in three dimensions even though the image hitting the retina is only two dimensional. The Ponzo illusion is an example of an illusion which uses monocular cues of depth perception to fool the eye.

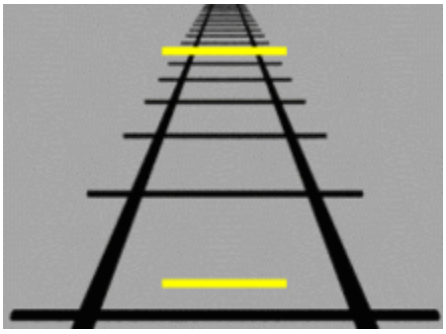


Figure 4.2 (e): Ponzo illusion

In the Ponzo illusion the converging parallel lines tell the brain that the image higher in the visual field is farther away therefore the brain perceives the image to be larger, although the two images hitting the retina are the same size. The Optical illusion seen in a diorama/false perspective also exploits assumptions based on monocular cues of depth perception. The M. C. Escher *Waterfall* painting (lithograph,1961) exploits rules of depth and proximity and our understanding of the physical world to create an illusion.

Like depth perception, motion perception is responsible for a number of sensory illusions. Film animation is based on the illusion that the brain perceives a series of slightly varied images produced in rapid succession as a moving picture. Likewise, when we are moving, as we would be while riding in a vehicle, stable surrounding objects may appear to move. We may also perceive a large object, like an airplane, to move more slowly than smaller objects, like a car, although the larger object is actually moving faster. The Phi phenomenon defined by Max Wertheimer (Gestalt psychology, (1912) is yet another example of how the brain perceives motion, which is most often created by blinking lights in close succession.

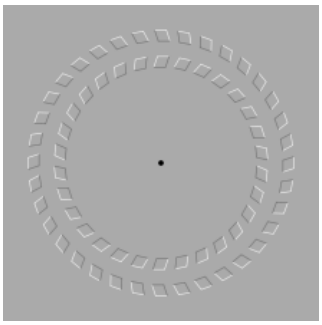


Figure 4.2(f): An optical illusion: the two circles seem to move when the viewer's head is moving forwards and backwards while looking at the black dot.

3.8 Cognitive processes hypothesis

The hypothesis claims that visual illusions occur because the neural circuitry in our visual system evolves by neural learning, to a system that makes very efficient interpretations of usual 3D scenes based in the emergence of simplified models in our brain that speed up the interpretation process but give rise to optical illusions in unusual situations. In this sense, the cognitive processes hypothesis can be considered a framework for an understanding of optical illusions as the signature of the empirical statistical way vision has evolved to solve the inverse problem.

Research indicates that 3D vision capabilities emerge and are learned jointly with the planning of movements. After a long process of learning, an internal representation of the world emerges that is well adjusted to the perceived data coming from closer objects. The representation of distant objects near the horizon is less "adequate". In fact, it is not only the Moon that seems larger when we perceive it near the horizon. In a photo of a distant scene, all distant objects are perceived as smaller than when we observe them directly using our vision.

The retinal image is the main source driving vision but what we see is a "virtual" 3D representation of the scene in front of us. We do not see a physical image of the world; we see objects, and the physical world is not itself separated into objects. We see it according to the way our brain organizes it. The names, colours, usual shapes and other information about the things we see pop up instantaneously from our neural circuitry and influence the representation of the scene. We "see" the most relevant information about the elements of the best 3D image that our neural networks can produce. The illusions arise when the "judgments" implied in the unconscious analysis of the scene are in conflict with reasoned considerations about it.

4.0 Conclusion

A human's perception of colours is a subjective process whereby the brain responds to the stimuli that are produced when incoming light reacts with the several types of cone photoreceptors in the eye, which makes it difficult to define its components.

5.0 Summary

In this unit, we have surveyed pixels and imaging, vision and colour perception in the human brain, alpha bending and optical illusion caused by excessive stimulation or interaction with contextual or competing stimuli of a specific type. An optical illusion is characterized by visually perceived images that differ from objective reality

6.0 Tutor Marked Assignment

1. What do you understand by alpha compositing?
2. Explain the human perception of colour
3. Identify the two sources of HDR imagery.
4. Mention the three main types of optical illusion with concrete examples.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. J.D Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, 1982
4. Rudolf F. Graf (1999). *Modern Dictionary of Electronics*. Oxford. ISBN-13:978-0750698665
5. Michael Goesele (2004). *New Acquisition Techniques for Real Objects and Light Sources in Computer Graphics*
6. Porter, Thomas; Tom Duff (1984). "Compositing Digital Images". *Computer Graphics* 18 (3): 253–259.

MODULE 5 – Ray tracing, illumination algorithms and GPGPU

UNIT 1: Ray tracing, BSP Trees and Monte-carlo raytracing

Contents **Pages**

1.0 Introduction

Although, three-dimensional computer graphics have been around for many decades, there has been a surge of general interest towards the field in the last couple of years. Just a quick glance at the latest blockbuster movies is enough to see the public's fascination with the new generation of graphics. As exciting as graphics are, however, there is a definite barrier which prevents most people from learning about them. For one thing, there is a lot of math and theory involved. Beyond that, just getting a window to display even simple 2D graphics can often be a daunting task. Simple 3D graphics method known as ray tracing, which can be understood and implemented without dealing with much math or the intricacies of windowing systems. The only math we assume is a basic knowledge of vectors, dot products, and cross products.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the raytracing
2. Identify the properties of raytracing
3. Build a simple raytracer

3.0 Main Content

3.1 Ray tracing

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television special effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration).

A serious disadvantage of ray tracing is low performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process afresh, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform anti-aliasing and improve image quality where needed. Although, it does handle inter-reflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required. The realism of all rendering methods, then, must be evaluated as an approximation to the equation, and in the case of ray tracing, it is not necessarily the most realistic. Other methods, including photon mapping, are based upon ray tracing for certain parts of the algorithm, yet give far better results.

3.2 Some Ray Terminologies

Before presenting the full description of ray tracing, we should agree on some basic terminology. When creating any sort of computer graphics, you must have a list of objects that you want your software to render. These objects are part of a scene or world (Fig. 5.1(a)), in graphics, this viewpoint is called the eye or camera. Following this camera analogy, just like a camera needs film onto which the scene is projected and recorded, in graphics we have a view window on which we draw the scene. The difference is that while in cameras the film is placed *behind* the aperture or focal point, in graphics the view window is in *front* of the focal point. So, the colour of each point on real film is caused by a light ray (actually, a group of rays) that passes through the aperture and hits the film, while in computer graphics each pixel of the final image is caused by a simulated light ray that hits the view window on its path towards the eye. The results, however, are the same

.

Our goal is find the colour of each point on the view window. We subdivide the view window

into small squares, where each square corresponds to one pixel in the final image. If you want to create an image at the resolution of 640x400, you would break up the view window into a grid of 640 squares across and 400 square down. The real problem, then, is assigning a colour to each square. This is what ray tracing does.

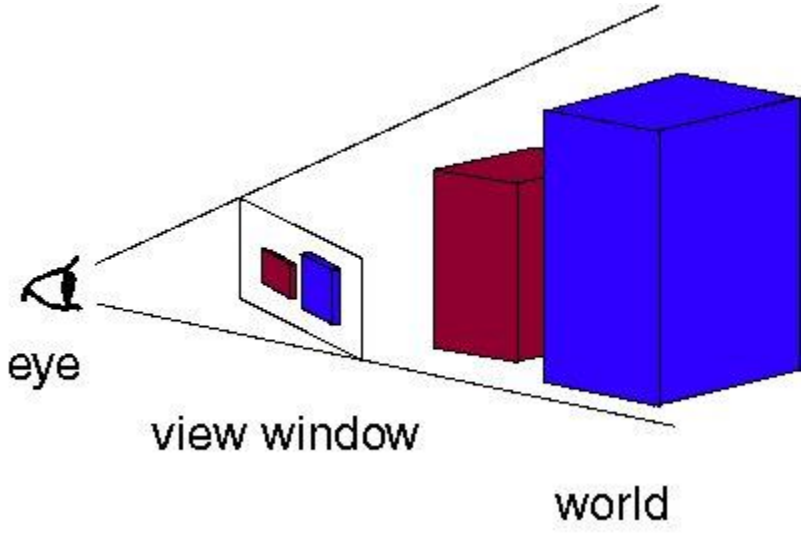


Figure 5.1(a). The eye, view window, and world.

Ray tracing is so named because it tries to simulate the path that light rays take as they bounce around within the world - they are *traced* through the scene. The objective is to determine the colour of each light ray that strikes the view window before reaching the eye. A light ray can best be thought of as a single photon (although this is not strictly accurate because light also has wave properties). The name ``ray tracing'' is a bit misleading because the natural assumption would be that rays are traced starting at their point of origin, the light source, and towards their destination, the eye (see Fig. 5.1(a)). This would be an accurate way to do it, but unfortunately it tends to be very difficult due to the sheer numbers involved. Consider tracing one ray in this manner through a scene with one light and one object, such as a table. We begin at the light bulb, but first need to decide how many rays to shoot out from the bulb. Then for each ray we have to decide in what direction it is going. There is really an infinity of directions in which it can travel - how do we know which to choose? Assuming these questions have been answered and a number of photons are being traced. Some will reach the eye directly, others will bounce around some and then reach the eye, and many, many more will probably never hit the eye at all. For all the rays that never reach the eye, the effort tracing them is wasted.

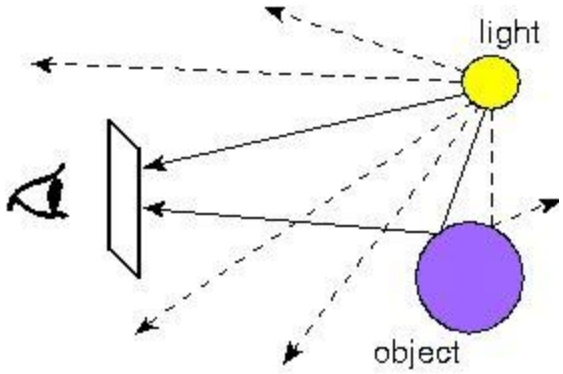


Figure 5.1(b). Tracing rays from the light source to the eye. Lots of rays are wasted because they never reach the eye.

In order to save ourselves this wasted effort, we trace only those rays that are *guaranteed* to hit the view window and reach the eye. It seems at first that it is impossible to know beforehand which rays reach the eye. After all, any given ray can bounce around the room many times before reaching the eye. However, if we look at the problem *backwards*, we see that it has a very simple solution. Instead of tracing the rays starting at the light source, we trace them backwards, starting at the eye. Consider any point on the view window whose colour we're trying to determine. Its colour is given by the colour of the light ray that passes through that point on the view window and reaches the eye. We can just as well follow the ray backwards by starting at the eye and passing through the point on its way out into the scene. The two rays will be identical, except for their direction: if the original ray came directly *from* the light source, then the backwards ray will go directly *to* the light source; if the original bounced off a table first, the backwards ray will also bounce off the table. You can see this by looking at Figure 5.1(b) again and just reversing the directions of the arrows. So the backwards method does the same thing as

the original method, except it does not waste any effort on rays that never reached the eye.

This, then, is how ray tracing works in computer graphics. For each pixel on the view window, we define a ray that extends from the eye to that point. We follow this ray out into the scene and as it bounces off of different objects. The final colour of the ray (and therefore of the corresponding pixel) is given by the colours of the objects hit by the ray as it travels through the scene.

Just as in the light-source-to-eye method it might take a very large number of bounces before the ray ever hits the eye, in backwards method it might take many bounces before the ray every hits the light. Since we need to establish some limit on the number of bounces to follow the ray on, we make the following approximation: every time a ray hits an object, we follow a single new ray from the point of intersection *directly towards* the light source (Fig. 5.1(c)).

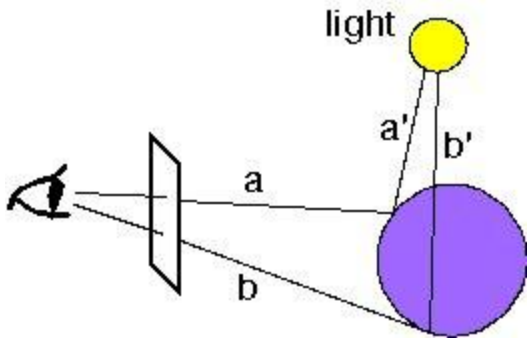


Figure 5.1(c). We trace a new ray from each ray-object intersection directly towards the light source

In the figure, there are two rays, a and b, which intersect the purple sphere. To determine the colour of a, we follow the new ray ‘a’ directly towards the light source. The colour of a will then depend on several factors, discussed in Colour and Shading discussed in section 3.4. As you can see, b will be shadowed because the ray b' towards the light source is blocked by the sphere itself. Ray a would have also been shadowed if another object blocked the ray a'.

3.3 Reflection and Refraction

Just as shadows are easily handled, so are reflection and refraction. In the above example, we only considered a single bounce from the eye to the light source. To handle reflection we also consider multiple bounces from objects, and to handle refraction we consider what happens when a ray passes *through* a partially- or fully-transparent object.

If an object is reflective we simply trace a new reflected ray from the point of intersection towards the direction of reflection. The reflected ray is the mirror image of the original ray, pointing away from the surface. If the object is to some extent transparent, then we also trace a refracted ray into the surface. If the materials on either side of the surface have different indices of refraction, such as air on one side and water on the other, then the refracted ray will be bent, like the image of a straw in a glass of water. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray and is not bent.

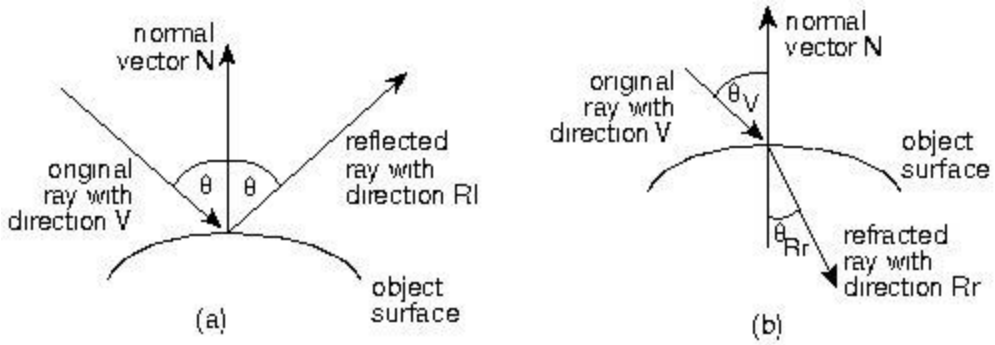


Figure 5.1d. (a) Reflected ray. (b) Refracted ray

The directions of the reflected and refracted rays are given by the following formulas. For a ray with direction V, and a surface with normal N (the normal is just the direction perpendicular to the surface - pointing directly away from it), the reflected ray direction Rl is given by

c1 = -dot_product(N, V)

Rl = V + (2 * N * c1)

Note that since V, N, and Rl are vectors with x, y, and z components, the arithmetic on them is performed per-component. The refracted ray direction Rr is given by

n1 = index of refraction of original medium

n2 = index of refraction of new medium

n = n1 / n2

$c2 = sqrt(1 - n^2 * (1 - c1^2))$

Rr = (n * V) + (n * c1 - c2) * N

3.3.1 Recursive Reflection and Refraction

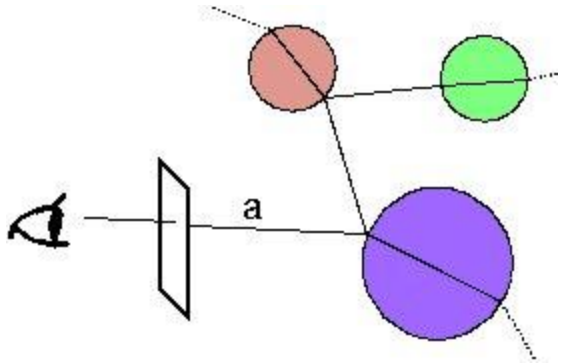


Figure 5.1(e). Recursive reflection and refraction

Figure 5.1(e) shows an example of objects that are both reflective and refractive (it does not show the rays from each intersection to the light source, for the sake of clarity). Note how much more complicated the rays become once reflection and refraction are added. Now, each reflected and refracted ray can strike an object which spawns another two rays, each of *these* may spawn another two, and so on. This arrangement of rays spawning subrays leads to the idea of a tree of rays. The root node of this tree is the original ray (a in the figure) from the eye, and each node in the tree is either a reflected or a refracted ray from the ray above it. Each leaf of the tree is either where the ray hit a non-reflective and non-transparent object, or where the tree depth reached a maximum.

As complicated as it seems, there is actually a very simple way to implement this tree of rays: with recursive function calls.

- For each point on the view window, we call a function trace_ray(), passing it the ray from the eye through the point.
- If the first object intersected by the ray is non-reflective and non- transparent, the function simply determines the colour at the intersection point and returns this colour.
- If this ray strikes a reflective object, however, the function invokes trace_ray() recursively, but with the reflected ray instead.
- If the intersected object is transparent, the function also calls trace_ray() with the refracted ray as a parameter.
- The colours returned by these two function calls are combined with the object colour, and the combined colour is returned.

This can be easily expressed in pseudo-code as:

```
Colour trace_ray( Ray original_ray )
{
       Colour point_colour, reflect_colour, refract_colour
       Object obj

       obj = get_first_intersection( original_ray )
       point_colour = get_point_colour( obj )

       if ( object is reflective )
              reflect_colour = trace_ray( get_reflected_ray( original_ray, obj ) )
       if ( object is refractive )
              refract_colour = trace_ray( get_refracted_ray( original_ray, obj ) )

       return ( combine_colours( point_colour, reflect_colour, refract_colour ))
}
```

where a Colour is the triple (R, G, B) for the red, green, and blue components of the colour, which can each vary between zero and one. This function, along with some intersection routines, is really all that is needed to write a ray tracer.

3.4 Ray Intersection

One of the basic computations needed by the ray tracer is an intersection routine for each type of object in the scene: one for spheres, one for cubes, one for cones, and so forth. If a ray intersects an object, the object's intersection routine returns the distance of the intersection point from the origin of the ray, the normal vector at the point of intersection, and, if texture mapping is being used, a coordinate mapping between the intersection point and the texture image. The distance to the intersection point is needed because if a ray intersects more than one object, we choose the one with the closest intersection point. The normal is used to determine the shade at the point, since it describes in which direction the surface is facing, and therefore affects how much light the point receives (see Colour and Shading in section 3.4).

3.4.1 Intersecting a Sphere

For each type of object that the ray tracer supports, you need a separate intersection routine. We will limit our ray tracer to handling spheres only, since the intersection routine for a sphere is among the simplest. The following derivation of a sphere intersection is based on [Glassner90].

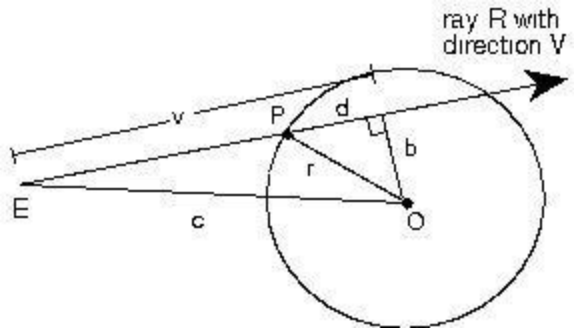


Figure 5.1(f). Intersection of a ray with a sphere.

Fig. 5.1(f) shows the geometry of a ray R (with origin at E and direction V) intersecting a sphere with center at O and radius r. According to the diagram,

$v^2 + b^2 = c^2$ and $d^2 + b^2 = r^2$ (by simple geometry)

and so

$d = sqrt(r^2 - (c^2 - v^2))$

To determine whether an intersection occurs, we compute the value of d. If d >= 0, then a valid intersection occurs. If the ray does not intersect, then d will be less than zero. After finding the value of d, the distance from E to the intersection point P is (v - d). The pseudocode for all this is:

```
v = dot_product( EO, V )
disc = r^2 - ((dot_product( EO, EO ) - v^2 )
if ( disc < 0 )
        no intersection
else
        d = sqrt( disc )
        P = E + (v - d) * V
```

3.5 Colour and Shading

There are many different ways to determine colour at a point of intersection. Some methods are very simple - as in flat shading, where every point on the object has the same colour. Some techniques - such as the Cook-Torrance method- are fairly complex, and take into account many physical attributes of the material in question. We will describe here a simple model known as Lambertian shading or cosine shading. It determines the brightness of a point based on the normal vector at the point and the vector from the point to the light source. If the two coincide (the surface directly faces the light source) then the point is at full intensity. As the angle between the two vectors increases, as when the surface is tilted away from the light, then the brightness diminishes. This model is known as ``cosine shading" because that mathematical function easily implements the above effect: it returns the value 1 when given an angle of zero, and returns zero when given a ninety degree angle (when the surface and light source are perpendicular). Thus, to find the brightness of a point, we simply take the cosine of the angle

between the two vectors. This value can be quickly computed because it is equal to the dot product of the two unit-length vectors. So by taking the dot product of the surface normal and the unit-length vector towards the light, we get a value between -1 and 1. The values from -1 to 0 indicate that the surface is *facing away* from the light source, so it doesn't receive any light. The value of 0 means the surface is directly perpendicular to the light source (it is at grazing incidence), and so again does not receive any light. Values above 0 indicate that the surface does receive some light, with maximum reception when the dot product is 1.

In case the result of the dot product is zero or below zero, we still may not want that part of the object to be pitch-black. After all, even when an object is completely blocked from a light source, there is still light bouncing around that illuminates it to some extent. For this reason we add a little bit of ambient light to every object, which means that the *minimum* amount of light that an object can receive is actually above zero. If we set the ambient coefficient to 20%, say, then even in total shadow an object will receive 20% illumination, and will thus be somewhat visible. If 20% illumination is always present, then the remaining 80% is determined by cosine shading. The value 80% in this case is known as the diffuse coefficient, which is just (1 – (ambient coefficient)). The final colour computation is then:

```
shade = dot_product( light_vector, normal_vector )
if ( shade < 0 )
      shade = 0
point_colour = object_colour * ( ambient_coefficient +
      diffuse_coefficient * shade )
```

3.6 Building a Simple Ray Tracer

The easiest way to describe a ray tracer is recursively, through a single function that traces a ray and calls itself for the reflected and transmitted rays.

Most of the work in ray tracing goes into the calculation of intersections between rays and surfaces. One reason is that it is difficult to implement a ray tracer that can handle a variety of objects in that as we add more complex objects, computing intersections become problematic. Consequently, most basic ray tracers support only flat and quadric surfaces.

We have seen the basic considerations that determine the ray-tracing process. Building a simple recursive ray tracer that can handle simple objects—quadrics and polyhedra—is quite easy. In this section, we will examine the basic structure and the functions that are required. We need two basic functions. The recursive function raytrace follows a ray, specified by a point and a direction, and returns the shade of the first surface that it intersects. It will use the function intersect to find the location of the closest surface that the specified ray intersects.

3.6.1 Recursive Ray Tracing

Let us consider the procedure trace in pseudocode. We give it a starting point p and a direction d, and it returns a colour c. In order to stop the ray tracer from recursing forever, we can specify a

maximum number of steps *max* that it can take. We will assume, for simplicity that we have only a single light source whose properties, as well as the description of the objects and their surface properties, are all available globally. If there are additional light sources, we can add in their contributions in a manner similar to the way in which we deal with the single source:

```
colour c = trace(point p, vector d, int step)
{
colour local, reflected, transmitted;
point q;
normal n;
if (step > max) return(background_colour);
q = intersect(p, d, status);

if (status == light_source) return(light_source_colour);
if (status == no_intersection) return(background_colour);
n = normal(q);
r = reflect(q, n);
t = transmit(q, n);
local = phong(q, n, r);
reflected = trace(q, r, step+1);
transmitted = trace(q, t, step+1);
return(local + reflected + transmitted);
}
```

Note that the calculation of reflected and transmitted colours must take into account how much energy is absorbed at the surface before reflection and transmission. If we have exceeded the maximum number of steps, we return a specified background colour. Otherwise, we use intersect to find the intersection of the given ray with the closest object. This function must have the entire database of objects available to it, and it must be able to find the intersections of rays with all types of objects supported. Consequently, most of the time spent in the ray tracer and the complexity of the code is hidden in this function.

If the ray does not intersect any object, we can return a status variable from intersect and return the background colour from trace. Likewise, if the ray intersects the light source, we return the colour of the source. If an intersection is returned, there are three components to the colour at this point: a local colour that can be computed using the modified Phong (or any other) model, a reflected colour, and, if the surface is translucent, a transmitted colour. Before computing these colours, we must compute the normal at the point of intersection, as well as the direction of reflected and transmitted rays.

The complexity of computing the normal depends on the class of objects supported by the ray tracer, and this calculation can be part of the function trace. The computation of the local colour requires a check to see if the light source is visible from the point of closest intersection. Thus, we cast a feeler or shadow ray from this point toward the light source and check whether it intersects any objects. We can note that this process can also be recursive because the shadow

ray might hit a reflective surface, such as a mirror, or a translucent surface, such as a piece of glass.

In addition, if the shadow ray hits a surface that itself is illuminated, some of this light should contribute to the colour at q. Generally, we ignore these possible contributions because they will slow the calculation significantly. Practical ray tracing requires that we make some compromises and is never quite physically correct. Next, we have two recursive steps that compute the contributions from the reflected and transmitted rays starting at q using trace. It is these recursions that make this code a ray tracer rather than a simple ray-casting rendering in which we find the first intersection and apply a lighting model at that point. Finally, we add the three colours to obtain the colour at p.

3.6.2 Calculating Intersections

Most of the time spent in a typical ray tracer is in the calculation of intersections in the function intersect. Hence, we must be very careful in limiting the objects to those for which we can find intersections easily. The general intersection problem can be expressed cleanly if we use an implicit representation of our objects. Thus, if an object is defined by the surface(s)

$$f(x, y, z) = f(\mathrm{p}) = 0,$$

and a ray from a point P_0 in the direction d is represented by the parametric form

$$\mathrm{p}(t) = P_0 + t\mathrm{d},$$

then the intersections are given for the values of t such that

$$f(P_0 + t\mathrm{d}) = 0,$$

which is a scalar equation in t. If f *is* an algebraic surface, then f is a sum of polynomial terms of the form $x^i y^j z^k$ and $f(\mathrm{P}_0 + t\mathrm{d})$ is a polynomial in t. finding the intersections reduces to finding all the roots of a polynomial. Unfortunately, there are only a few cases that do not require numerical methods. One is quadrics. All quadrics could be written as the quadratic form

$$\mathrm{P}^T\mathrm{A}\mathrm{p} + \mathrm{b}^T\mathrm{p} + c = 0.$$

Substituting in the equation for a ray, leaves us with a scalar quadratic equation to solve for the values of t that yield zero, one, or two intersections. Ray tracers can handle quadrics without difficulty because the solution of the quadratic equation requires only the taking of a single square root. In addition, we can eliminate those rays that miss a quadric object and those that are tangent to it before taking the square root, further simplifying the calculation.

Consider, for example, a sphere centered at pc with radius r, which can be written as

$$(\mathrm{p} - \mathrm{p}_c) \cdot (\mathrm{p} - \mathrm{p}_c) - r^2 = 0.$$

Substituting in the equation of the ray

$$\mathrm{P}(t) = \mathrm{P}_0 + t\mathrm{d},$$

we get the quadratic equation $\mathrm{d} \cdot \mathrm{d}t^2 + 2(\mathrm{P}_0 - \mathrm{P}c) \cdot \mathrm{d}t + (\mathrm{P}_0 - \mathrm{P}c) \cdot (\mathrm{P}_0 - \mathrm{p}c) - r2 = 0.$

Planes are also simple. We can take the equation for the ray and substitute it into the equation of a plane

$$p \cdot n + c = 0,$$

which yields a scalar equation that requires only a single division to solve. Thus, for the ray

$$p = P_0 + td,$$

we find

$$t = \frac{P_0 \cdot n + cn \cdot d}{}$$

.

However, planes by themselves have limited applicability in modeling scenes. We are usually interested either in the intersection of multiple planes that form convex objects (polyhedra) or a piece of a plane that defines a flat polygon. For polygons, we must decide whether the point of intersection lies inside or outside the polygon.

The difficulty of such a test depends on whether the polygon is convex and, if not convex, whether it is simple. For convex polygons, there are very simple tests that are similar to the tests for ray intersections with polyhedra that we consider next. Although we can define polyhedra by their faces, we can also define them as the convex objects that are formed by the intersection of planes. Thus, a parallelepiped is defined by six planes and a tetrahedron by four. For ray tracing, the advantage of this definition is that we can use the simple ray–plane intersection equation to derive a ray–polyhedron intersection test.

We develop the test as follows. Let us assume that all the planes defining our polyhedron have normals that are outward facing. Consider the ray in Figure 5.1(f) that intersects the polyhedron. It can enter and leave the polygon only once. It must enter through a plane that is facing the ray and leave through a plane that faces in the direction of the ray. However, this ray must also intersect all the planes that form the polyhedron (except those parallel to the ray).Consider the intersections of the ray with all the front-facing planes—that is, those whose normals point toward the starting point of the ray. The entry point must be the intersection farthest along the ray. Likewise, the exit point is the nearest intersection point of all the planes facing away from the origin of the ray, and the entry point must be closer to the initial point than the exit point.

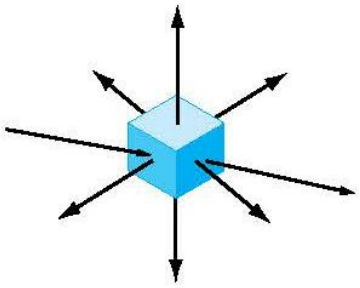


FIGURE 5.1(g) Ray intersecting a polyhedron with outwardfacing normals shown.

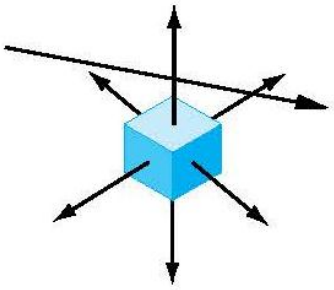


FIGURE 5.1(h) Ray missing a polyhedron with outwardfacing normals shown.

If we consider a ray that misses the same polyhedron, as shown in Figure 5.1(h), we see that the farthest intersection with a front-facing plane is farther from the initial point than the closest intersection with a back-facing plane. Hence, our test is to find these possible entry and exit points by computing the ray–plane intersection points, in any order, and updating the possible entry and exit points as we find the intersections. The test can be halted if we ever find a possible exit point closer than the present entry point or a possible entry point farther than the present exit point

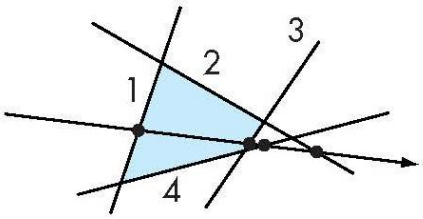


FIGURE 5.1(i) Ray intersecting a convex polygon

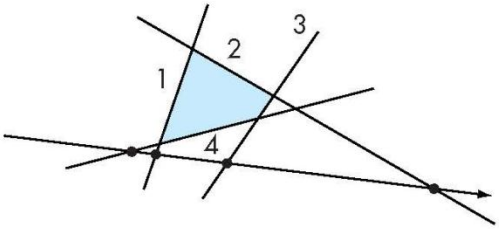


FIGURE 5.1 (j) Ray missing a convex polygon.

Consider the two-dimensional example illustrated in Figure 5.1(i) that tests for a ray–convex polygon intersection in a plane. Here lines replace planes, but the logic is the same. Suppose that we do the intersections with the lines in the order 1, 2, 3, 4. Starting with line 1, we find that this line faces the initial point by looking at the sign of the dot product of the normal with the direction of the ray. The intersection with line 1 then yields a possible entry point. Line 2 faces away from the initial point and yields a possible exit point that is farther away than our present estimate of the entry point. Line 3 yields an even closer exit point but still one that is farther than the entry point. Line 4 yields a farther exit point that can be discarded. At this point, we have tested all the lines and conclude the ray passes through the polygon.

Figure 5.1(j) has the same lines and the same convex polygon but shows a ray that misses the polygon. The intersection with line 1 still yields a possible entry point. The intersections with lines 2 and 3 still yield possible exit points that are farther than the entry point. But the

intersection with line 4 yields an exit point closer than the entry point, which indicates that the ray must miss the polygon.

3.7 BSP Trees

A Binary Space Partitioning (BSP) tree is a structure that, as the name suggests, subdivides the space into smaller sets. These days, given hardware accelerated Z-buffers; the benefit of this is that one has a smaller amount of data to consider given a location in space. But in the beginning of the 90's, the main reason BSP-trees were being used was that they sorted the polygons in the scene so that you always drew back-to-front, meaning that the polygon with the lowest Z-value was drawn last. There are other ways to sort the polygons so that the closest polygon is drawn last, for example the Painter's algorithm, but few are as cheap as BSP-trees, because the sorting of the polygons is done during the pre-processing of the map and not under run-time. The algorithm for generating a BSP-tree is actually an extension of Painter's algorithm (foley et al, 1990). Just as the original design of the BSP algorithm, the Painter's algorithm works by drawing the polygons in the scene in back-to-front order. However, there are some major drawbacks with Painter's algorithm:

1. Polygons will not be drawn correctly if they pass through any other polygon.
2. It is difficult and computationally expensive to calculate the order that the polygons should be drawn in for each frame.
3. The algorithm cannot handle cases of cyclic overlap as shown in the figure below.

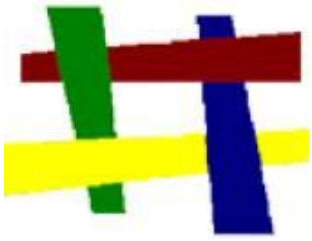


Figure 5.1(k): Cyclic Overlap

3.7.1 The BSP Algorithm

The original idea for the creation of a BSP-tree is that you take a set of polygons that is part of a scene and divide them into smaller sets, where each subset is a convex set of polygons. That is, each polygon in this subset is in front of every other polygon in the same set. Polygon 1 is in front of polygon 2 if each vertex in polygon 1 is on the positive side of the plane polygon 2 defines or in that plane that. A cube made of inward facing polygons is a convex set, whilst a cube made of outwards facing polygons is not.

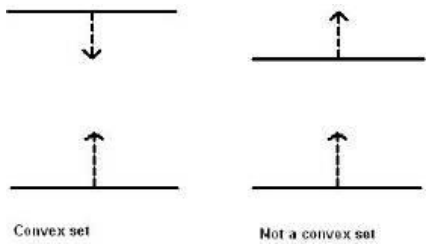


Figure 5.1(l): The difference between a convex set and a non-convex set

The functions needed to determine whether a set of polygons is a convex set would look like this:

► CLASSIFY-POINT
// *Indata*:
// Polygon – The polygon to classify the point versus.
* Point - 3D-point to classify versus the plane defined by the polygon.
* *Outdata*:Which side the point is of the polygon.
* *Effect*: Determines on which side of the plane defined by the polygon the point is located.

```
CLASSIFY-POINT (Polygon, Point)
1 SideValue = Polygon.Normal * Point
2 if (SideValue = Polygon.Distance)
3     then return COINCIDING
4 else if (SideValue < Polygon.Distance)
5     then return BEHIND
6 else return INFRONT
```

► POLYGON-INFRONT
* *Indata*:
* Polygon1 – The polygon to determine whether the other polygon is in front of or not
* Polygon2 – The polygon to check if it is in front of the first polygon or not
* *Outdata*:
* Whether the second is in front of the first polygon or not.
* *Effect*:
* Checks each point in the second polygon is in front of the first polygon. If so is the case it is considered to be in the front of it.

```
POLYGON-INFRONT (Polygon1, Polygon2)
1 for each point p in Polygon2
2     if (CLASSIFY-POINT (Polygon1, p) <> INFRONT)
3         then return false
4 return true
```

► IS-CONVEX-SET
* *Indata*:
* PolygonSet – The set of polygons to check for convexity
* *Outdata*:
* Whether the set is convex or not
* *Effect*:
* Checks each polygon against each other polygon, to see if they are

```
*   in front of each other, if any two polygons doesn't fulfill that
*   criteria the set isn't convex.

IS-CONVEX-SET (PolygonSet)
1 for i f 0 to PolygonSet.Length ()
2     for j f 0 to PolygonSet.Length ()
3         if(i <> j && not POLYGON-INFRONT(PolygonSet[i], PolygonSet[j]))
4             then return false
5 return true
```

The function POLYGON-INFRONT is a non-symmetric comparison, meaning that if Polygon2 is in front of Polygon1 it does not necessarily imply that Polygon1 is in front of Polygon2. This can easily be shown with the following example:

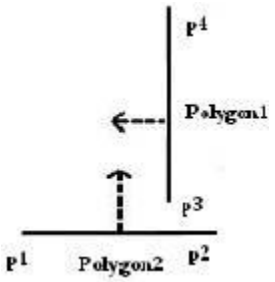


Figure 5.1(m): The non-symmetric nature of the comparison POLGON-INFRONT

In Figure 5.1(m), Polygon1 is in front of Polygon2 since both p3 and p4 is on the positive side of Polygon2, but Polygon2 is not in front of Polygon1 since p2 is behind Polygon1. The idea can be slightly modified as the need of convex sets is not as acute when you can use hardware accelerated Z-buffers. Later in this series it will be described how this can be solved.

The structures needed for a BSP-tree can be defined as follows:

```
class BSPTree
{
   BSPTreeNode RootNode          // The root node of the tree.
}
class BSPTreeNode
{
   BSPTree Tree                  // The tree this node belongs to.
   BSPTreePolygon Divider        // The polygon that lies in middle w of the two sub trees.
   BSPTreeNode *RightChild       // The right sub tree of this node.
   BSPTreeNode *LeftChild        // The left sub tree of this node.
   BSPTreePolygon PolygonSet[]   // The set of polygons in this node.
}
class BSPTreePolygon
{
```

```
   3DVector Point1          // Vertex 1 in the polygon.
   3DVector Point3          // Vertex 2 in the polygon.
   3DVector Point3          // Vertex 3 in the polygon.
}
```

As you can see each polygon is represented by only three points. This is because the hardware in graphic cards is designed to draw triangles. But the algorithm for generating BSP-trees is designed to take care of polygons with any number of points, as long as all points are in the same plane.
There are several ways to split up the set of polygons into smaller subsets. For example, you can choose an arbitrary plane in space and divide the polygons by putting the ones on the positive side of the plane in the right sub tree and the polygons on the negative side in the left sub tree. The problem with this approach is that it is very difficult to find a plane that divides the polygons into two approximately equally sized sets, since there are an infinite set of planes in space. So the most common way to do this is by taking one of the polygons in the scene and dividing the polygons according to the plane that polygon defines.

We have defined an algorithm, POLYGON-INFRONT, which can classify whether a polygon is on the positive side of another polygon. Now, we need to modify that algorithm to be able to also determine whether the polygon is spanning the plane defined by the other polygon. The algorithm is defined as follows:

```
► CALCULATE-SIDE
* Indata:
*   Polygon1 – The polygon to classify the other polygon against
*   Polygon2 – The polygon to classify
* Outdata:
*   Which side of polygon1 polygon 2 is located on.
* Effect:
*   Classifies each point in the second polygon versus the
*   first polygon. If there are points on the positive side but no
*   points on the negative side, Polygon2 is considered to be in front
*   of Polygon1. If there are points on the negative side but no
*   points on the positive side, Polygon2 is considered to be behind
*   Polygon1. If all points are coinciding polygon2 is considered to
*   be coinciding with Polygon1. The last possible case is that there
*   are points on both the positive and the negative side, then
*   polygon2 is considered to be spanning Polygon1.

CALCULATE-SIDE (Polygon1, Polygon2)
1  NumPositive = 0, NumNegative = 0
2  for each point p in Polygon2
3     if (CLASSIFY-POINT (Polygon1, p) = INFRONT)
4        then NumPositive = NumPositive + 1
5     if (CLASSIFY-POINT (Polygon1, p) = BEHIND)
```

```
6        then NumNegative = NumNegative + 1
7  if (NumPositive > 0 && NumNegative = 0)
8     then return INFRONT
9  else if(NumPositive = 0 && NumNegative > 0)
10    then return BEHIND
11 else if(NumPositive = 0 && NumNegative = 0)
12    then return COINCIDING
13 else return SPANNING
```

This gives us a problem when it comes to determining which subset a polygon that is spanning the plane should be placed in. The algorithm deals with this by splitting such a polygon into two polygons. This also solves two of the problems in Painter's algorithm, namely cyclic overlap and intersecting polygons. Below is example of how a polygon is splitted:

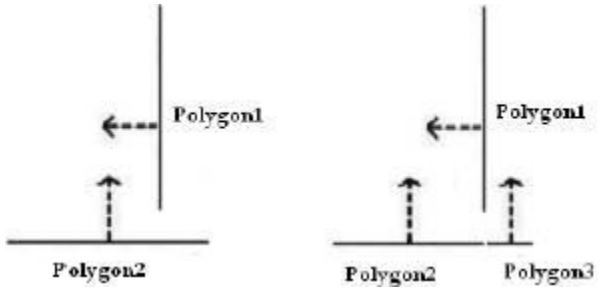


Figure 5.1(n): Splitting a polygon

In the figure above Polygon1 is the classifying polygon and Polygon2 is the polygon that is classified. Since Polygon2 is spanning the plane defined by Polygon1 it has to be splitted. The result is the picture to the right. Polygon2 is now completely in front of Polygon1 and Polygon3 is completely behind. The glitch between Polygon2 and Polygon3 is just there to illustrate that it is two separate polygons, after a split the two resulting polygons will be adjacent to each other.

When a BSP-tree is created, one has to decide whether the need is of a balanced tree, meaning that there should not be too big a difference in depth between the left and the right sub tree of each node, or try to limit the number of splits, since each split creates new polygons. If too many new polygons is created during the BSP-tree creation the graphic card will have a hard time rendering the map, thus reducing the frame rate, while a unbalanced tree will require more expensive traversal of the tree. We decided to accept a certain number of splits in order to get a more balanced tree. But the main concern was reducing the number of new polygons created. Below is our loop for choosing the best dividing polygon from a set of polygons:

► CHOOSE-DIVIDING-POLYGON
* *Indata*:
* PolygonSet – The set of polygons to search for the best dividing polygon.
* *Outdata*:
* The best dividing polygon
* *Effect*:
* Searches through the set of polygons and returns the polygons that
* splits the set into the two best resulting sets. If the set is
* convex no polygon can be returned.

```
CHOOSE-DIVIDING-POLYGON (PolygonSet)
1  if (IS-CONVEX-SET (PolygonSet))
2     then return NOPOLYGON
3  MinRelation = MINIMUMRELATION
4  BestPolygon = NOPOLYGON
5  LeastSplits = INFINITY
6  BestRelation = 0
   // Loop to find the polygon that best divides the set.
7  while(BestPolygon = NOPOLYGON)
8     for each polygon P1 in PolygonSet
9        if (Polygon P1 has not been used as divider previously
            during the creation of the tree)

         // Count the number of polygons on the positive side, negative side
            // of and spanning the plane defined by the current polygon.
10           NumPositive = 0, NumNegative = 0, NumSpanning = 0
11           for each polygon P2 in PolygonSet except P1
12              Value = CALCULATE-SIDE(P1, P2)
13              if(Value = INFRONT)
14                 NumPositive = NumPositive + 1
15              else if(Value = BEHIND)
16                 NumNegative = NumNegative + 1
17              else if(Value = SPANNING)
18                 NumSpanning = NumSpanning + 1

         // Calculate the relation between the number of polygons in the two
          // sets divided by the current polygon.
19        if (NumPositive < NumNegative)
20           Relation = NumPositive / NumNegative
21        else
22           Relation = NumNegative / NumPositive

        // Compare the results given by the current polygon to the best this
          // far. If the this polygon splits fewer polygons and the relation
        // between the resulting sets is acceptable this is the new candidate
        // polygon. If the current polygon splits the same amount of polygons
          // as the best polygon this far and the relation between the two
          // resulting sets is better -> this polygon is the new candidate
```

```
        // polygon.
23        if (Relation > MinRelation && (NumSpanning < LeastSplits ||
            NumSpanning = LeastSplits && Relation > BestRelation))
24          BestPolygon = P1
25          LeastSplits = NumSpanning
26          BestRelation = Relation

      // Decrease the number least acceptable relation by dividing it with
      // a predefined constant.
27    MinRelation = MinRelation / MINRELATIONSCALE
28 return BestPolygon
```

3.7.1.1 Complexity Analysis

Because of the while loop it is very hard to find a bound to this function. Depending of the structure of the scene the while loop might loop for a very long time. The MINRELATIONSCALE is what decides how much the acceptable relation decreases per iteration, thus how long it will take before the minimum relation will be small enough to accept the best possible solution. The worst case is that we have a set consisting of n polygons that is not a convex set and the best possible solution is a dividing polygon that splits the set into one part consisting of n-1 polygons and another set consisting of 1 polygon. This solution will only be acceptable when the minimal acceptable relation is less than 1/(n-1) (see line 19-23 in the algorithm). Meaning that *MinRelation* / $\text{MINRELATIONSCALE}^i < 1/(n-1)$ where i is the number of iterations in the loop, this is due the division by MINRELATIONSCALE at line 27 in the algorithm. Let us assume that the initial value for *MinRelation* is 1, which is the highest possible value since the relation is always between 0 and 1

(see lines 19-22 in the algorithm). We have:

$1 / \text{MINRELATIONSCALE}^i < 1/(n-1)$

$1 < \text{MINRELATIONSCALE}^i/(n-1)$

$(n-1) < \text{MINRELATIONSCALE}^i$

$\log_{\text{MINRELATIONSCALE}} (n-1) < i$

This is no upper bound for i, but since i will be very close to $\log_{\text{MINRELATIONSCALE}} (n-1)$ we will, for simplicity assume they are equal. Another practical assumption to make is that MINRELATIONSCALE always should be greater than or equal to 2. Thus giving us:

$\log_{\text{MINRELATIONSCALE}} (n-1) = i$ MINRELATIONSCALE >= 2

$i = \log_{\text{MINRELATIONSCALE}} (n-1) < \lg(n-1) = O(\lg n)$

Inside the while loop, there are two iterations over the set of polygons. Giving us that the worst case behavior of this algorithm is of order $O(n^2 \lg n)$, but the typical behavior is almost always closer to $O(n^2)$ as there tend to exist a polygon that will fulfill the requirements in the first iteration.

The loop in CHOOSE-DIVIDING-POLYGON might look as if there are cases where it will not terminate, but this is not true since if the set of polygons is a non-convex set there is always one polygon that can divide the polygons into two sets. CHOOSE-DIVIDING-POLYGON selects the polygon that splits the least number of polygons. To prevent from choosing polygons that would not divide the set, the relation between the sizes of the two resulting sets must be better than a threshold value. To better illustrate this we show an example where choosing the polygon that splits the fewest amount of polygons would render in an infinite loop:

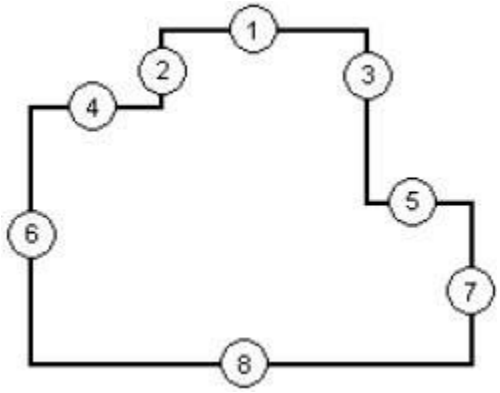


Figure 5.1(o): Problems when choosing dividing polygon

In the above example, choosing either polygon 1,6,7 or 8 would not result in the split of any polygon, but on the other hand each of the polygons in the set is on the positive side of those polygons, so in the next loop the same polygon would be chosen again, rendering in a infinite loop. As a matter of fact 1,2,3 and 4 is on the border of the least convex hull that can hold the polygon set, polygons for which this is true cannot be used as a dividing polygon since all other polygons in the set is on the positive side of them. Choosing polygon 2,3,4 or 5 would each cause one split but it would also divide the set into two smaller sets.

Another reason why a it is not always good to choose the polygon that splits the fewest polygons is that in most cases that heuristic will result in a unbalanced set. A balanced tree will perform better during runtime than an unbalanced one.

When the best polygon has been chosen the rest of the polygons is divided according to that polygon. There are two ways to do deal with the dividing polygon:

1. A leafy tree can be created, meaning that all polygons are put into the leaf nodes, thus the dividing polygons have to be categorized to be on one of the sides. In our example we count the polygons in the same plane as the dividing polygon as being on the positive side of the plane.
2. The other way is to store the dividing polygons in the internal nodes. This process is repeated for each sub tree until each leaf contains a convex set of polygons.

The algorithm for generating a leafy BSP-tree looks like this:

```
► GENERATE-BSP-TREE
* Indata:
*   Node - The sub tree to build of the type BSPTreeNode.
*   PolygonSet – The set of polygons to create a BSP-tree from.
* Outdata:
*   A BSP-tree stored in the incoming node.
* Effect:
*   Generates a BSP-tree out of a set of polygons.

GENERATE-BSP-TREE (Node, PolygonSet)
1  if (IS-CONVEX-SET (PolygonSet))
2     Tree = BSPTreeNode (PolygonSet)
3  Divider = CHOOSE-DIVIDING-POLYGON (PolygonSet)
4  PositiveSet = {}
5  NegativeSet = {}
6  for each polygon P1 in PolygonSet
7     Value f CALCULATE-SIDE (Divider, P1)
8     if(Value = INFRONT)
9        PositiveSet = PositiveSet U P1
10    else if (Value = BEHIND)
11       NegativeSet = NegativeSet U P1
12    else if (Value = SPANNING)
13       Split_Polygon (P1, Divider, Front, Back)
14       PositiveSet = PositiveSet U Front
15       NegativeSet = NegativeSet U Back
16 GENERATE-BSP-TREE (Tree.RightChild, PositiveSet)
17 GENERATE-BSP-TREE (Tree.LeftChild, NegativeSet)
```

The call to CHOOSE-DIVIDING-POLYGON is of order $O(n^2 \lg n)$, which dominates the rest of the function except for the recursive calls. If we assume that the division of the polygon set is fairly even we can formulate the following function to calculate the bounds of GENERATE-BSP-TREE:

$T(n) = 2T(n/2) + O(n^2 \lg n)$

Using Masters Theorem (Thomas, 2001) we get that the order of complexity is $\Theta(n^2 \lg n)$, where n is the number of polygons in the incoming set. Following there is an example of how a BSP-tree is generated. The structure below is the original set of polygons, we have numbered them to make the example easier to follow. This set of polygons is going to be divided into a BSP-tree.

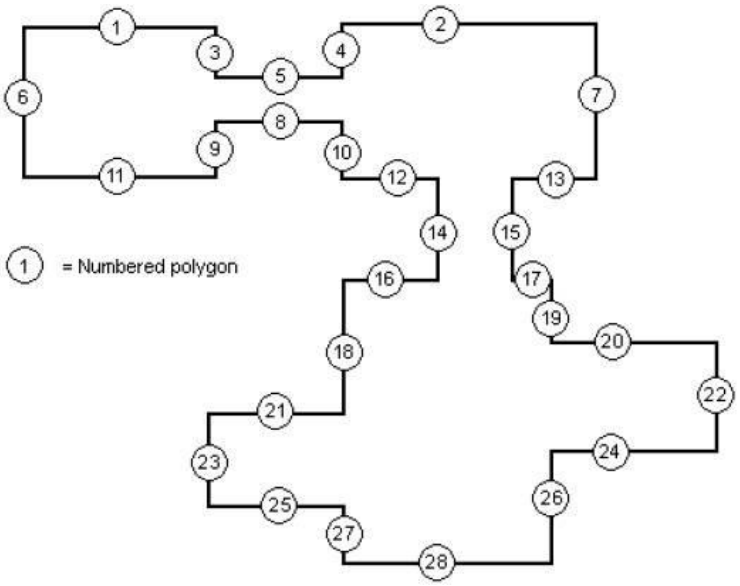


Figure 5.1(p): Example Structure

To be able to run the algorithm, we must choose a couple of constants, namely: MINIMUMRELATION and MINRELATIONSCALE. We found that choosing MINIMUMRELATION = 0.8 and MINRELATIONSCALE = 2 will give quite good result, but one can experiment we those numbers. The higher the MINIMUMRELATION is the better balanced the tree will be, but the number of splitted polygons will probably increase too.

The starting set of polygons is obviously not a convex set, so a dividing polygon will be chosen. After a quick glance at the structure, we can see that polygons {1,2,6,22,28} cannot be used as dividing polygons since they define convex hull that contains the whole set of polygons. But all the other polygons are candidates for being dividing polygon. The polygons that split the fewest number of polygons and give the best relation between the sizes of the two resulting sets are 16 and 17, these two polygons lie on the same line and do not split any other polygon. The two resulting sets is almost equally sized namely |negative|= 15 and |positive| = 13 polygons in each of the resulting sets. Let us choose polygon 16 as the divider. The result will look as follows:

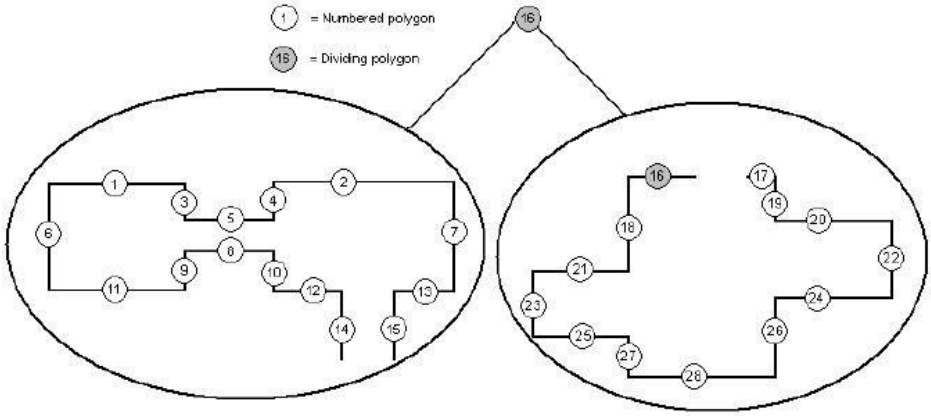


Figure 5.1(q): The result of a split at polygon 16

Neither the right nor the left subtree contains a convex set of polygons so a new dividing polygon must be chosen in both.

In the left sub tree {1,2,6,7} is on the convex hull and they cannot be used as dividers. Polygon 4 and 10 is on the same line and they do not split any other polygon. The sizes of the resulting sets is |negative|= 7 and |positive| = 8 which is very balanced. We choose 4 as the divider.

{16,17,22,23,28} contains the right sub tree, so they will not be dividers. The polygons that will not split any other polygons are {18,19,27,26} but the sizes of the resulting sets for all of them will be |negative|= 3 and |positive| = 11, 3/11 is below the minimum relation(0.5) so we will have to check the other polygons to see if they can provide us with a more balanced solution. Each of {20,21,24,25} splits exactly one polygon, but the most balanced set is attained by polygon 21, which after splitting polygon 22 produces resulting sets of size |negative|= 6 and |positive| = 8.
The following shows the result after these operations:

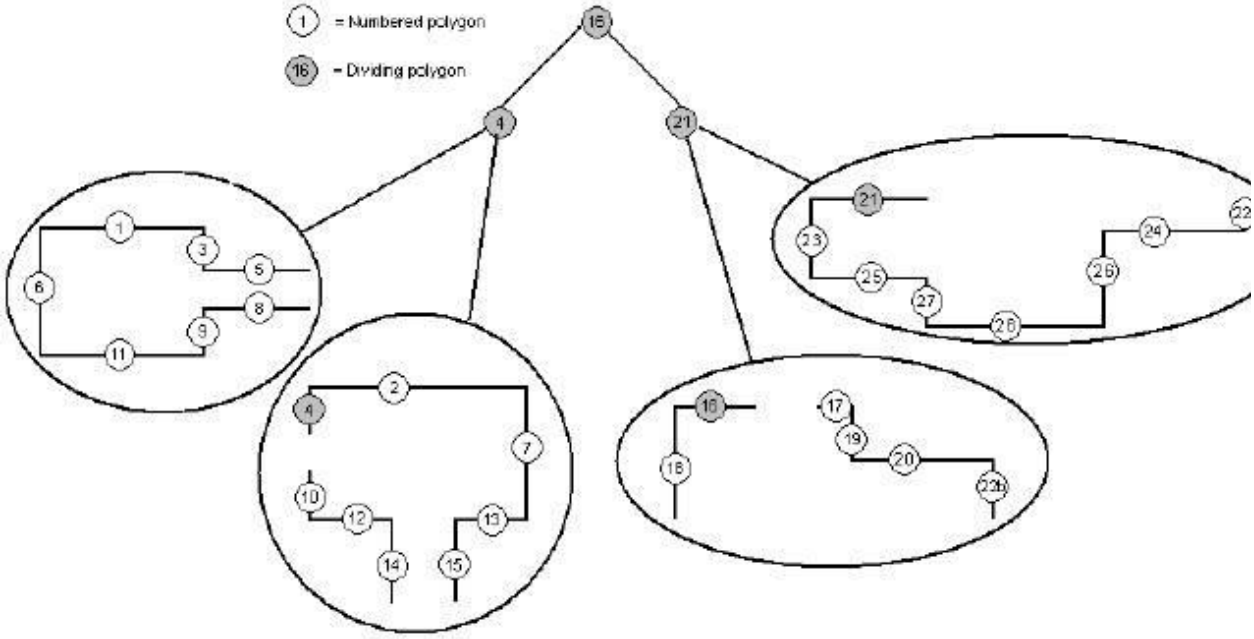


Figure 5.1(r): The second step

None of the sub trees contain a convex set of polygons so the algorithm will move on in the same manner; the resulting tree will look like this:

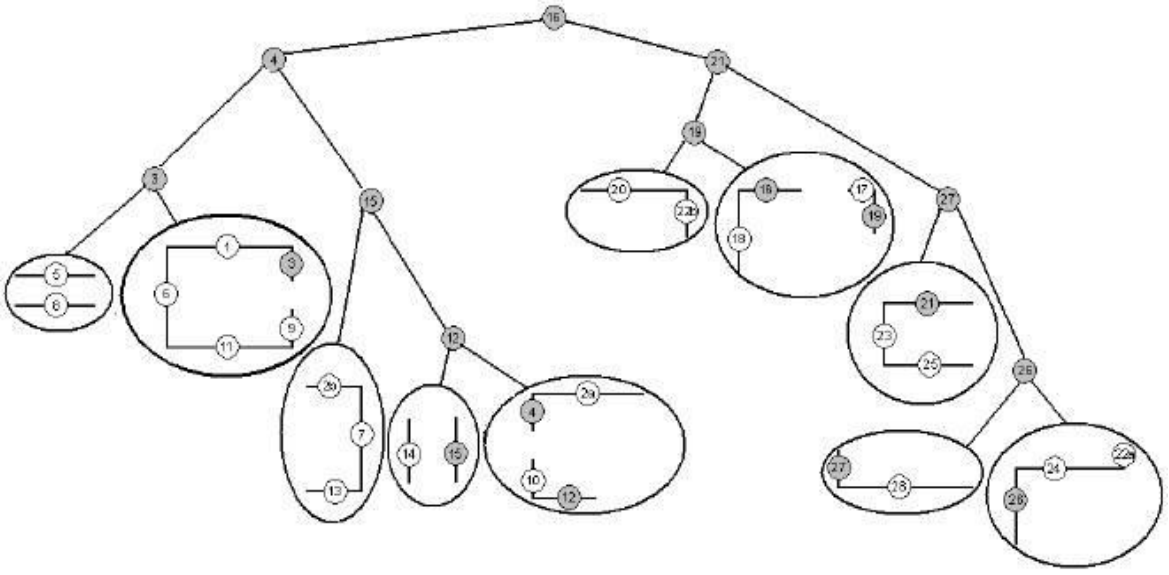


Figure 5.1(s): The final tree

Even though it is not the optimal solution it is quite close to it and it does not take that long time.

3.7.1.2 Drawing the BSP Tree

Now that the BSP-tree is created, it is very easy to draw the polygons the tree, with zero chance of failure in the drawing. The algorithm of that process is described below. We assume there is a function IS-LEAF that given a BSP-node, it returns true if that node is a leaf otherwise false.

```
► DRAW-BSP-TREE
* Indata:
*   Node – The node to draw.
*   Position – The viewer's position.
* Outdata:
*   None
* Effect:
*   Draws the polygons contained in the node and its sub trees.

DRAW-BSP-TREE (Node, Position)
1  if (IS-LEAF(Node))
2     DRAW-POLYGONS (Node.PolygonSet)
3     return

   // Calculate which sub tree the viewer is in.
4  Side = CLASSIFY-POINT (Node.Divider, Position)

   // If the viewer is in the right sub tree draw that tree before the
```

```
  // left.
5  if (Side = INFRONT || Side = COINCIDING)
6      DRAW-BSP-TREE (Node.RightChild, Position)
7      DRAW-BSP-TREE (Node.LeftChild, Position)

  // Otherwise draw the left first.
8  else if(Value = BEHIND)
9      DRAW-BSP-TREE (Node.LeftChild, Position)
10     DRAW-BSP-TREE (Node.RightChild, Position)
```

This way of drawing gives us no reduction in number of polygons that is drawn to the screen. Since a map can consist of hundreds of thousands of polygons, it is no good solution. In some way, nodes that are not visible or even polygons that are not visible should be discarded. This is called hidden surface removal.

3.8 Bounding volume

In computer graphics and computational geometry, a bounding volume for a set of objects is a closed volume that completely contains the union of the objects in the set. Bounding volumes are used to improve the efficiency of geometrical operations by using simple volumes to contain more complex objects. Normally, simpler volumes have simpler ways to test for overlap. A bounding volume for a set of objects is also a bounding volume for the single object consisting of their union, and the other way around. Therefore, it is possible to confine the description to the case of a single object, which is assumed to be non-empty and bounded (finite).

Uses of bounding volumes

1. Bounding volumes are most often used to accelerate certain kinds of tests.

2. In ray tracing, bounding volumes are used in ray-intersection tests, and in many rendering algorithms, they are used for viewing frustum tests. If the ray or viewing frustum does not intersect the bounding volume, it cannot intersect the object contained in the volume. These intersection tests produce a list of objects that must be displayed. Here, displayed means rendered or rasterized.

3. In collision detection, when two bounding volumes do not intersect, then the contained objects cannot collide.

Testing against a bounding volume is typically much faster than testing against the object itself, because of the bounding volume's simpler geometry. This is because an 'object' is typically composed of polygons or data structures that are reduced to polygonal approximations. In either case, it is computationally wasteful to test each polygon against the view volume if the object is not visible. (Onscreen objects must be 'clipped' to the screen, regardless of whether their surfaces are actually visible.)

To obtain bounding volumes of complex objects, a common way is to break the objects/scene down using a scene graph or more specifically bounding volume hierarchies like e.g. (Oriented Bounding Box) OBB trees. The basic idea behind this is to organize a scene in a tree-like structure where the root comprises the whole scene and each leaf contains a smaller subpart.

3.8.1 Common types of bounding volume

The choice of the type of bounding volume for a given application is determined by a variety of factors: the computational cost of computing a bounding volume for an object, the cost of updating it in applications in which the objects can move or change shape or size, the cost of determining intersections, and the desired precision of the intersection test. The precision of the intersection test is related to the amount of space within the bounding volume not associated with the bounded object, called void space. Sophisticated bounding volumes generally allow for less void space but are more computationally expensive. It is common to use several types in conjunction, such as a cheap one for a quick but rough test in conjunction with a more precise but also more expensive type.

A **bounding sphere** is a sphere containing the object. In 2-D graphics, this is a circle. Bounding spheres are represented by centre and radius. They are very quick to test for collision with each other: two spheres intersect when the distance between their centres does not exceed the sum of their radii. This makes bounding spheres appropriate for objects that can move in any number of dimensions.

A **bounding ellipsoid** is an ellipsoid containing the object. Ellipsoids usually provide tighter fitting than a sphere. Intersections with ellipsoids are done by scaling the other object along the principal axes of the ellipsoid by an amount equal to the multiplicative inverse of the radii of the ellipsoid, thus reducing the problem to intersecting the scaled object with a unit sphere. Care should be taken to avoid problems if the applied scaling introduces skew. Skew can make the usage of ellipsoids impractical in certain cases, for example collision between two arbitrary ellipsoids.

A bounding cylinder is a cylinder containing the object. In most applications the axis of the cylinder is aligned with the vertical direction of the scene. Cylinders are appropriate for 3-D objects that can only rotate about a vertical axis but not about other axes, and are otherwise constrained to move by translation only. Two vertical-axis-aligned cylinders intersect when, simultaneously, their projections on the vertical axis intersect – which are two line segments – as well their projections on the horizontal plane – two circular disks. Both are easy to test. In video games, bounding cylinders are often used as bounding volumes for people standing upright.

A **bounding capsule** is a swept sphere (i.e. the volume that a sphere takes as it moves along a straight line segment) containing the object. Capsules can be represented by the radius of the swept sphere and the segment that the sphere is swept across). It has traits similar to a cylinder, but is easier to use, because the intersection test is simpler. A capsule and another object intersect

if the distance between the capsule's defining segment and some feature of the other object is smaller than the capsule's radius. For example, two capsules intersect if the distance between the capsules' segments is smaller than the sum of their radii. This holds for arbitrarily rotated capsules, which is why they're more appealing than cylinders in practice.

A **bounding box** is a cuboid, or in 2-D a rectangle, containing the object. In dynamical simulation, bounding boxes are preferred to other shapes of bounding volume such as bounding spheres or cylinders for objects that are roughly cuboid in shape when the intersection test needs to be fairly accurate. The benefit is obvious, for example, for objects that rest upon other, such as a car resting on the ground: a bounding sphere would show the car as possibly intersecting with the ground, which then would need to be rejected by a more expensive test of the actual model of the car; a bounding box immediately shows the car as not intersecting with the ground, saving the more expensive test.

In many applications the bounding box is aligned with the axes of the co-ordinate system, and it is then known as an axis-aligned bounding box (AABB). To distinguish the general case from an AABB, an arbitrary bounding box is sometimes called an oriented bounding box (OBB). AABBs are much simpler to test for intersection than OBBs, but have the disadvantage that when the model is rotated they cannot be simply rotated with it, but need to be recomputed.

3.9 Monte Carlo Ray tracing

Realistic image synthesis is increasingly important in areas such as entertainment (movies, special effects and games), design, architecture and more. A common trend in all these areas is to request more realistic images of increasingly complex models. Monte Carlo ray tracing based techniques are the only methods that can handle this complexity. Recent advances in algorithms and computation power have made Monte Carlo ray tracing the natural choice for most problems. This is a significant change from just a few years back when the (finite element) radiosity method was the preferred algorithm for most graphics researchers.

3.9.1 Advantages of Monte Carlo ray tracing

• Geometry can be procedural
• No tessellation is necessary
• It is not necessary to pre-compute a representation for the solution
• Geometry can be duplicated using instancing
• Any type of BRDF can be handled
• Specular reflections (on any shape) are easy
• Memory consumption is low
• The accuracy is controlled at the pixel/image level
• Complexity has empirically been found to be O(logN) where N is number of scene elements. Compare this with O(N logN) for the fastest finite element methods.

In addition, one might add that Monte Carlo ray tracing methods can be very easy to implement. A basic path tracing algorithm which has all of the above advantages is a relatively straightforward extension to ray tracing. The main problem with Monte Carlo ray tracing is variance seen as noise in the rendered images. This noise can be eliminated by using more samples. Unfortunately, the convergence of Monte Carlo methods is quite slow, and a large number of samples can be necessary to reduce the variance to an acceptable level. Another way of reducing variance is to try to be cleverer; a large part of this course material is devoted to techniques and algorithms for making Monte Carlo ray tracing more efficient.

4.0 Conclusion

A serious disadvantage of ray tracing is low performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately.

5.0 Summary

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects.

6.0 Tutor Marked Assignment

1.0 What do you understand by raytracing?
2.0 What makes it different from other scanline rendering methods?
3.0 state the advantages of monte-carlo raytracing.
4.0 What is BSP trees?

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL,* Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),*Prentice Hall, 1997, ISBN: 0135309247
3. Sunsted, Tod. (1997) *3D computer graphics: Moving from wire-frame drawings to solid, shaded models*
4. Sobey, Anthony.(2007) *Software Engineering*
5. Southwick, Andrew R. *Quake Rendering Tutorial*
6. S. Mann and R. W. Picard. "On Being ‘Undigital’ With Digital Cameras: Extending Dynamic Range By Combining Differently Exposed Pictures". http://citeseer.ist.psu.edu/mann95being.html

MODULE 5 – Ray tracing, illumination algorithms and GPGPU
UNIT 2: Photon Mapping and Radiosity

Contents **Pages**

1.0 Introduction

In this unit we will be looking at two popular illumination algorithms: Photon mapping and radiosity which have been used in the field of computer graphics and also what makes it preferred above other illumination algorithms.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the illumination algorithms: Photon mapping and radiosity
2. Understand the interaction of light with surfaces
3. Understand the usage of photo mapping

3.0 Main Content

3.1 Photo mapping

Photon mapping is a two-pass global illumination algorithm developed by Henrik Wann Jensen (2001) that solves the rendering equation. Rays from the light source and rays from the camera are traced independently until some termination criterion is met, and then they are connected in a second step to produce a radiance value. It is used to realistically simulate the interaction of light with different objects. Specifically, it is capable of simulating the refraction of light through a transparent substance such as glass or water, diffuse inter-reflection between illuminated objects, the subsurface scattering of light in translucent materials, and some of the effects caused by particulate matter such as smoke or water vapor. It can also be extended to more accurate simulations of light such as spectral rendering.

Unlike path tracing, bidirectional path tracing and Metropolis light transport, photon mapping is a "biased" rendering algorithm, which means that averaging many renders using this method does not converge to a correct solution to the rendering equation. However, since it is a consistent method, a correct solution can be achieved by increasing the number of photons.

Figure 5.2(a): A model of a wine glass ray traced with photon mapping to show caustics.

Light refracted or reflected causes patterns called caustics, usually visible as concentrated patches of light on nearby surfaces. For example, as light rays pass through a wine glass sitting on a table, they are refracted and patterns of light are visible on the table. Photon mapping can trace the paths of individual photons to model where these concentrated patches of light will appear.

3.2 Diffuse inter-reflection

Diffuse inter-reflection is apparent when light from one diffuse object is reflected onto another. Photon mapping is particularly adept at handling this effect because the algorithm reflects photons from one surface to another based on that surface's bidirectional reflectance distribution function (BRDF), and thus light from one object striking another is a natural result of the method. Diffuse inter-reflection was first modeled using radiosity solutions. Photon mapping differs though in that it separates the light transport from the nature of the geometry in the scene. Colour bleed is an example of diffuse inter-reflection.

3.3 Subsurface scattering

Subsurface scattering is the effect evident when light enters a material and is scattered before being absorbed or reflected in a different direction. Subsurface scattering can accurately be modeled using photon mapping. This was the original way Jensen (2001) implemented it. However, the method becomes slow for highly scattering materials, and bidirectional surface scattering reflectance distribution functions (BSSRDFs) are more efficient in these situations.

3.4 Usage of Photon Mapping

3.4.1 Construction of the photon map (1st pass)

With photon mapping, light packets called *photons* are sent out into the scene from the light sources. Whenever a photon intersects with a surface, the intersection point and incoming direction are stored in a cache called the *photon map*. Typically, two photon maps are created for

a scene: one especially for caustics and a global one for other light. After intersecting the surface, a probability for reflecting, absorbing, or transmitting/refracting is given by the material. A Monte Carlo method called *Russian roulette* is used to choose one of these actions. If the photon is absorbed, no new direction is given, and tracing for that photon ends. If the photon reflects, the surface's bidirectional reflectance distribution function is used to determine a new direction. Finally, if the photon is transmitting, a different function for its direction is given depending upon the nature of the transmission.

Once the photon map is constructed (or during construction), it is typically arranged in a manner that is optimal for the k-nearest neighbor algorithm, as photon look-up time depends on the spatial distribution of the photons. Jensen advocates the usage of kd-trees. The photon map is then stored on disk or in memory for later usage.

3.4.2 Rendering (2nd pass)

In this step of the algorithm, the photon map created in the first pass is used to estimate the radiance of every pixel of the output image. For each pixel, the scene is ray traced until the closest surface of intersection is found.

At this point, the rendering equation is used to calculate the surface radiance leaving the point of intersection in the direction of the ray that struck it. To facilitate efficiency, the equation is decomposed into four separate factors: direct illumination, specular reflection, caustics, and soft indirect illumination.

For an accurate estimate of direct illumination, a ray is traced from the point of intersection to each light source. As long as a ray does not intersect another object, the light source is used to calculate the direct illumination. For an approximate estimate of indirect illumination, the photon map is used to calculate the radiance contribution.

Specular reflection can be, in most cases, calculated using ray tracing procedures (as it handles reflections well). The contribution to the surface radiance from caustics is calculated using the caustics photon map directly. The number of photons in this map must be sufficiently large, as the map is the only source for caustics information in the scene.

For soft indirect illumination, radiance is calculated using the photon map directly. This contribution, however, does not need to be as accurate as the caustics contribution and thus uses the global photon map.

3.5 Calculating radiance using the photon map

In order to calculate surface radiance at an intersection point, one of the cached photon maps is used. The steps are:

1. Gather the N nearest photons using the nearest neighbor search function on the photon map.
2. Let S be the sphere that contains these N photons.
3. For each photon, divide the amount of flux (real photons) that the photon represents by the area of S and multiply by the BRDF applied to that photon.
4. The sum of those results for each photon represents total surface radiance returned by the surface intersection in the direction of the ray that struck it.

3.5.1 Optimizations of Photon mapping

1. To avoid emitting unneeded photons, the initial direction of the outgoing photons is often constrained. Instead of simply sending out photons in random directions, they are sent in the direction of a known object that is a desired photon manipulator to either focus or diffuse the light. There are many other refinements that can be made to the algorithm: for example, choosing the number of photons to send, and where and in what pattern to send them. It would seem that emitting more photons in a specific direction would cause a higher density of photons to be stored in the photon map around the position where the photons hit, and thus measuring this density would give an inaccurate value for irradiance. This is true; however, the algorithm used to compute radiance does *not* depend on irradiance estimates.
2. For soft indirect illumination, if the surface is Lambertian, then a technique known as irradiance caching may be used to interpolate values from previous calculations.
3. To avoid unnecessary collision testing in direct illumination, shadow photons can be used. During the photon mapping process, when a photon strikes a surface, in addition to the usual operations performed, a shadow photon is emitted in the same direction the original photon came from that goes all the way through the object. The next object it collides with causes a shadow photon to be stored in the photon map. Then during the direct illumination calculation, instead of sending out a ray from the surface to the light that tests collisions with objects, the photon map is queried for shadow photons. If none are present, then the object has a clear line of sight to the light source and additional calculations can be avoided.
4. To optimize image quality, particularly of caustics, Jensen recommends use of a cone filter. Essentially, the filter gives weight to photons' contributions to radiance depending on how far they are from ray-surface intersections. This can produce sharper images.
5. Image space photon mapping achieves real-time performance by computing the first and last scattering using a GPU rasterizer.

3.5.2 Variations

Although photon mapping was designed to work primarily with ray tracers, it can also be extended for use with scanline renderers.

3.6 Radiosity

Radiosity is a global illumination algorithm used in 3D computer graphics rendering. Radiosity is an application of the finite element method to solving the rendering equation for scenes with purely diffuse surfaces. Unlike Monte Carlo algorithms (such as path tracing), which handle all types of light paths, typical radiosity methods only account for paths which leave a light source and are reflected diffusely some number of times (possibly zero) before hitting the eye. Such paths are represented as "LD*E". Radiosity calculations are viewpoint independent which increases the computations involved, but makes them useful for all viewpoints.

Radiosity methods were first developed in about 1950 in the engineering field of heat transfer. They were later refined specifically for application to the problem of rendering computer graphics in 1984 by researchers at Cornell University.

3.6.1 Visual characteristics of Radiosity

The inclusion of radiosity calculations in the rendering process often lends an added element of realism to the finished scene, because of the way it mimics real-world phenomena. Consider a simple room scene.

Figure 5.2(b): Difference between standard direct illumination and radiosity

The image on the left was rendered with a typical direct illumination renderer. There are *three types* of lighting in this scene which have been specifically chosen and placed by the artist in an attempt to create realistic lighting: spot lighting with shadows (placed outside the window to create the light shining on the floor), ambient lighting (without which any part of the room not lit

directly by a light source would be totally dark), and omnidirectional lighting without shadows (to reduce the flatness of the ambient lighting).

The image on the right was rendered using a radiosity algorithm. There is only one source of light: an image of the sky placed outside the window. The difference is marked. The room glows with light. Soft shadows are visible on the floor, and subtle lighting effects are noticeable around the room. Furthermore, the red colour from the carpet has bled onto the grey walls, giving them a slightly warm appearance. None of these effects were specifically chosen or designed by the artist.

3.7 The Radiosity algorithm

The surfaces of the scene to be rendered are each divided up into one or more smaller surfaces (patches). A view factor is computed for each pair of patches. View factors (also known as *form factors*) are coefficients describing how well the patches can see each other. Patches that are far away from each other, or oriented at oblique angles relative to one another, will have smaller view factors. If other patches are in the way, the view factor will be reduced or zero, depending on whether the occlusion is partial or total.

The view factors are used as coefficients in a linearized form of the rendering equation, which yields a linear system of equations. Solving this system yields the radiosity, or brightness, of each patch, taking into account diffuse inter-reflections and soft shadows.

Progressive radiosity solves the system iteratively in such a way that after each iteration, we have intermediate radiosity values for the patch. These intermediate values correspond to bounce levels. That is, after one iteration, we know how the scene looks after one light bounce, after two passes, two bounces, and so forth. Progressive radiosity is useful for getting an interactive preview of the scene. Also, the user can stop the iterations once the image looks good enough, rather than wait for the computation to numerically converge.

As the algorithm iterates, light can be seen to flow into the scene, as multiple bounces are computed. Individual patches are visible as squares on the walls and floor. Another common method for solving the radiosity equation is "shooting radiosity," which iteratively solves the radiosity equation by "shooting" light from the patch with the most error at each step. After the first pass, only those patches which are in direct line of sight of a light-emitting patch will be illuminated. After the second pass, more patches will become illuminated as the light begins to bounce around the scene. The scene continues to grow brighter and eventually reaches a steady state.

3.7.1 Mathematical formulation

The basic radiosity method has its basis in the theory of thermal radiation, since radiosity relies on computing the amount of light energy transferred among surfaces. In order to simplify computations, the method assumes that all scattering is perfectly diffuse. Surfaces are typically discretized into quadrilateral or triangular elements over which a piecewise polynomial function is defined.

After this breakdown, the amount of light energy transfer can be computed by using the known reflectivity of the reflecting patch, combined with the view factor of the two patches. This dimensionless quantity is computed from the geometric orientation of two patches, and can be thought of as the fraction of the total possible emitting area of the first patch which is covered by the second patch.

More correctly, radiosity B is the energy per unit area leaving the patch surface per discrete time interval and is the combination of emitted and reflected energy:

$$B(x)\,dA = E(x)\,dA + \rho(x)\,dA \int_S B(x')\frac{1}{\pi r^2}\cos\theta_x \cos\theta_{x'} \cdot \mathrm{Vis}(x,x')\,\mathrm{d}A'$$

where:

1. $B(x)_\mathrm{i}\ \mathrm{d}A_\mathrm{i}$ is the total energy leaving a small area $\mathrm{d}A_\mathrm{i}$ around a point x.
2. $E(x)_\mathrm{i}\ \mathrm{d}A_\mathrm{i}$ is the emitted energy.
3. $\rho(x)$ is the reflectivity of the point, giving reflected energy per unit area by multiplying by the incident energy per unit area (the total energy which arrives from other patches).
4. S denotes that the integration variable x' runs over all the surfaces in the scene
5. r is the distance between x and x'
6. θ_x and $\theta_{\mathrm{x}'}$ are the angles between the line joining x and x' and vectors normal to the surface at x and x' respectively.
7. Vis(x,x') is a visibility function, defined to be 1 if the two points x and x' are visible from each other, and 0 if they are not.

The geometrical form factor (or "projected solid angle") F_ij.

F_ij can be obtained by projecting the element A_j onto a the surface of a unit hemisphere, and then projecting that in turn onto a unit circle around the point of interest in the plane of A_i. The form factor is then equal to the proportion of the unit circle covered by this projection.

Form factors obey the reciprocity relation $A_\mathrm{i}F_\mathrm{ij} = A_\mathrm{j}F_\mathrm{ji}$

If the surfaces are approximated by a finite number of planar patches, each of which is taken to have a constant radiosity B_i and reflectivity ρ_i, the above equation gives the discrete radiosity equation,

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j$$

where F_{ij} is the geometrical view factor for the radiation leaving j and hitting patch i.

This equation can then be applied to each patch. The equation is monochromatic, so colour radiosity rendering requires calculation for each of the required colours.

3.7.1.1 Solution methods

The equation can formally be solved as matrix equation, to give the vector solution:

$$B = (I - \rho F)^{-1} E$$

This gives the full "infinite bounce" solution for B directly. However, the number of calculations to compute the matrix solution scales according to n^3, where n is the number of patches. This becomes prohibitive for realistically large values of n.

Instead, the equation can more readily be solved iteratively, by repeatedly applying the single-bounce update formula above. Formally, this is a solution of the matrix equation by Jacobi iteration. The reflectivities ρ_i are less than 1, this scheme converges quickly, typically requiring only a handful of iterations to produce a reasonable solution. Other standard iterative methods for matrix equation solutions can also be used, for example the Gauss–Seidel method (jefferys, 1988), where updated values for each patch are used in the calculation as soon as they are computed, rather than all being updated synchronously at the end of each sweep. The solution can also be tweaked to iterate over each of the sending elements in turn in its main outermost loop for each update, rather than each of the receiving patches. This is known as the *shooting* variant of the algorithm, as opposed to the *gathering* variant. Using the view factor reciprocity, $A_i F_{ij} = A_j F_{ji}$, the update equation can also be re-written in terms of the view factor F_{ji} seen by each *sending* patch A_j:

$$A_i B_i = A_i E_i + \rho_i \sum_{j=1}^{n} A_j B_j F_{ji}$$

This is sometimes known as the "power" formulation, since it is now the total transmitted power of each element that is being updated, rather than its radiosity.

The view factor F_{ij} itself can be calculated in a number of ways. Early methods used a *hemi-cube* (an imaginary cube centered upon the first surface to which the second surface was projected, devised by Cohen and Greenberg in 1985). The surface of the hemi-cube was divided into pixel-like squares, for each of which a view factor can be readily calculated analytically. The full form factor could then be approximated by adding up the contribution from each of the pixel-like squares. The projection onto the hemi-cube, which could be adapted from standard methods for determining the visibility of polygons, also solved the problem of intervening patches partially obscuring those behind.

However, all this was quite computationally expensive, because ideally form factors must be derived for every possible pair of patches, leading to a quadratic increase in computation as the number of patches increased. This can be reduced somewhat by using a binary space partitioning tree to reduce the amount of time spent determining which patches are completely hidden from others in complex scenes; but even so, the time spent to determine the form factor still typically scales as $n \log n$. New methods include adaptive integration

3.7.1.2 Sampling approaches

The form factors F_{ij} themselves are not in fact explicitly needed in either of the update equations; neither to estimate the total intensity $\sum_j F_{ij} B_j$ gathered from the whole view, nor to estimate how the power $A_j B_j$ being radiated is distributed. Instead, these updates can be estimated by sampling methods, without ever having to calculate form factors explicitly. Since the mid 1990s such sampling approaches have been the methods most predominantly used for practical radiosity calculations.

The gathered intensity can be estimated by generating a set of samples in the unit circle, lifting these onto the hemisphere, and then seeing what was the radiosity of the element that a ray incoming in that direction would have originated on. The estimate for the total gathered intensity is then just the average of the radiosities discovered by each ray. Similarly, in the power formulation, power can be distributed by generating a set of rays from the radiating element in the same way, and spreading the power to be distributed equally between each element a ray hits.

This is essentially the same distribution that a path-tracing program would sample in tracing back one diffuse reflection step; or that a bidirectional ray tracing program would sample to achieve one forward diffuse reflection step when light source mapping forwards. The sampling approach therefore to some extent represents a convergence between the two techniques, the key difference remaining that the radiosity technique aims to build up a sufficiently accurate map of the radiance of all the surfaces in the scene, rather than just a representation of the current view.

3.7.2 Reducing computation time

Although in its basic form radiosity is assumed to have a quadratic increase in computation time with added geometry (surfaces and patches), this need not be the case. The radiosity problem can be rephrased as a problem of rendering a texture mapped scene. In this case, the computation time increases only linearly with the number of patches (ignoring complex issues like cache use).

Following the commercial enthusiasm for radiosity-enhanced imagery, but prior to the standardization of rapid radiosity calculation, many architects and graphic artists used a technique referred to loosely as false radiosity. By darkening areas of texture maps corresponding to corners, joints and recesses, and applying them via self-illumination or diffuse mapping, a radiosity-like effect of patch interaction could be created with a standard scanline renderer (cf. ambient occlusion). Radiosity solutions may be displayed in realtime via lightmaps on current desktop computers with standard graphics acceleration hardware

3.7.3 Advantages of Radiosity Algorithm

One of the advantages of the radiosity algorithm is that it is relatively simple to explain and implement. This makes it a useful algorithm for teaching students about global illumination algorithms. A typical direct illumination renderer already contains nearly all of the algorithms (perspective transformations, texture mapping, hidden surface removal) required to implement radiosity. A strong grasp of mathematics is not required to understand or implement this algorithm.

3.7.4 Limitations of Radiosity Algorithm

Typical radiosity methods only account for light paths of the form LD*E, i.e., paths which start at a light source and make multiple diffuse bounces before reaching the eye. Although there are several approaches to integrating other illumination effects such as specular and glossy reflections, radiosity-based methods are generally not used to solve the complete rendering equation.

Basic radiosity also has trouble resolving sudden changes in visibility (e.g., hard-edged shadows) because coarse, regular discretization into piecewise constant elements corresponds to a low-pass box filter of the spatial domain. Discontinuity meshing uses knowledge of visibility events to generate a more intelligent discretization.

4.0 Conclusion

Photon mapping is a "biased" rendering algorithm, which means that averaging many renders using this method does not converge to a correct solution to the rendering equation. Radiosity calculations are viewpoint independent which increases the computations involved, but makes them useful for all viewpoints.

5.0 Summary

In this unit, we have surveyed tow global illumination algorithms, photon mapping and radiosity. Photon mapping is used to realistically simulate the interaction of light with different objects while Radiosity is an application of the finite element method to solving the rendering equation for scenes with purely diffuse surfaces.

6.0 Tutor Marked Assignment

1.0 What do you understand by photon mapping?
2.0 Discuss radiosity as an illumination algorithm
3.0 Differentiate between standard direct illumination and radiosity
4.0 What are the limitations of the radiosity algorithm?

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. Modeling the interaction of light between diffuse surfaces", C. Goral, K. E. Torrance, D. P. Greenberg and B. Battaile, *Computer Graphics*, Vol. 18, No. 3.
4. G Walton, *Calculation of Obstructed View Factors by Adaptive Integration*, NIST Report NISTIR-6925, see also http://view3d.sourceforge.net/
5. *Realistic Image Synthesis Using Photon Mapping* ISBN 1-56881-147-0
6. Jensen, Henrik W., Realistic Image Synthesis Using Photon Mapping, A K Peters, Ltd., Massachusetts, 2001. ISBN-13: 978-1568814629.

MODULE 5 – Ray tracing, illumination algorithms and GPGPU

UNIT 3: General Purpose – GPU Computing

Contents **Pages**

1.0 Introduction

General-Purpose computing on Graphics Processing Units (GPGPU, also referred to as GPGP and less often GP²U) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. It is made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines, which allows programmers to use stream processing on non-graphics data. Additionally, the use of multiple graphics cards in a single computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing

GPU functionality has, traditionally, been very limited. In fact, for many years the GPU was only used to accelerate certain parts of the graphics pipeline. Some improvements were needed before GPGPU became feasible.

2.0 Objectives

On completing this unit, you would be able to:

1. Understand the GPGPU
2. Understand programming concepts such as stream processing, computational resources and kernels
3. Identify application areas of GPU computing.

3.0 Main Content

3.1 GPGPU programming concepts

GPUs are designed specifically for graphics and thus are very restrictive in terms of operations and programming. As a result of their nature, GPUs are only effective at tackling problems that can be solved using stream processing and the hardware can only be used in certain ways. The following are some of the programming concepts in GPGPU computing.

1. **Stream processing**

GPUs can only process independent vertices and fragments, but can process many of them in parallel. This is especially effective when the programmer wants to process many vertices or fragments in the same way. In this sense, GPUs are stream processors – processors that can operate in parallel by running a single kernel on many records in a stream at once.

A stream is simply a set of records that require similar computation. Streams provide data parallelism. Kernels are the functions that are applied to each element in the stream. In the GPUs, vertices and fragments are the elements in streams and vertex and fragment shaders are the kernels to be run on them. Since GPUs process elements independently, there is no way to have shared or static data. For each element, we can only read from the input, perform operations on it, and write to the output. It is permissible to have multiple inputs and multiple outputs, but never a piece of memory that is both readable and writable.

Arithmetic intensity is defined as the number of operations performed per word of memory transferred. It is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speed. Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between data elements.

2. Computational resources

There are a variety of computational resources available on the GPU:

1. Programmable processors – Vertex, primitive, and fragment pipelines allow programmer to perform kernel on streams of data
2. Rasterizer – creates fragments and interpolates per-vertex constants such as texture coordinates and colour
3. Texture Unit – read only memory interface
4. Frame buffer – write only memory interface

In fact, the programmer can substitute a write only texture for output instead of the framebuffer. This is accomplished either through Render to Texture (RTT), Render-To-Backbuffer-Copy-To-Texture (RTBCTT), or the more recent stream-out.

5. Textures as stream

The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs. Many computations naturally map into grids: matrix algebra, image processing, physically based simulation, and so on.

Since textures are used as memory, texture lookups are then used as memory reads. Certain operations can be done automatically by the GPU because of this.

6. **Kernels**

Kernels can be thought of as the body of loops. For example, if the programmer were operating on a grid on the CPU, they might have code that looked like this:

```
// Input and output grids have 10000 x 10000 or 100 million elements.

void transform_10k_by_10k_grid(float in[10000][10000], float
out[10000][10000])
{
  for(int x = 0; x < 10000; x++)
  {
    for(int y = 0; y < 10000; y++)
    {
      // The next line is executed 100 million times
      out[x][y] = do_some_hard_work(in[x][y]);
    }
  }
}
```

On the GPU, the programmer only specifies the body of the loop as the kernel and what data to loop over by invoking geometry processing.

7. **Flow control**

In sequential code it is possible to control the flow of the program using if-then-else statements and various forms of loops. Such flow control structures have only recently been added to GPUs. Conditional writes could be accomplished using a properly crafted series of arithmetic/bit operations, but looping and conditional branching are not possible.

Recent GPUs allow branching, but usually with a performance penalty. Branching should generally be avoided in inner loops, whether in CPU or GPU code, and various techniques, such as static branch resolution, pre-computation, predication, loop splitting, and Z-cull can be used to achieve branching when hardware support does not exist.

3.2 GPU techniques:

The following are GPU techniques:

1. **Map**

The map operation simply applies the given function (the kernel) to every element in the stream. A simple example is multiplying each value in the stream by a constant (increasing the brightness of an image). The map operation is simple to implement on the GPU. The programmer generates a fragment for each pixel on screen and applies a fragment program to each one. The result stream of the same size is stored in the output buffer.

2. **Reduce**

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally a reduction can be accomplished in multiple steps. The results from the previous step are used as the input for the current step and the range over which the operation is applied is reduced until only one stream element remains.

3. **Stream filtering**

Stream filtering is essentially a non-uniform reduction. Filtering involves removing items from the stream based on some criteria.

4. **Scatter**

The scatter operation is most naturally defined on the vertex processor. The vertex processor is able to adjust the position of the vertex, which allows the programmer to control where information is deposited on the grid. Other extensions are also possible, such as controlling how large an area the vertex affects.

The fragment processor cannot perform a direct scatter operation because the location of each fragment on the grid is fixed at the time of the fragment's creation and cannot be altered by the programmer. However, a logical scatter operation may sometimes be recast or implemented with an additional gather step. A scatter implementation would first emit both an output value and an output address. An immediately following gather operation uses address comparisons to see whether the output value maps to the current output slot.

5. **Gather**

The fragment processor is able to read textures in a random access fashion, so it can gather information from any grid cell, or multiple grid cells, as desired.

6. **Sort**

The sort operation transforms an unordered set of elements into an ordered set of elements. The most common implementation on GPUs is using sorting networks.

7. **Search**

The search operation allows the programmer to find a particular element within the stream, or possibly find neighbors of a specified element. The GPU is not used to speed up the search for an individual element, but instead is used to run multiple searches in parallel.

8. **Data structures**

A variety of data structures can be represented on the GPU:

1. Dense arrays
2. Sparse arrays – static or dynamic
3. Adaptive structures

3.4 Applications

The following are some of the areas where GPUs have been used for general purpose computing:

1. Bitcoin peer-to-peer currency relies on a distributed computing network for performing SHA256 calculations where GPGPUs have become the dominant mode of calculation
2. MATLAB acceleration using the Parallel Computing Toolbox and MATLAB Distributed Computing Server, as well as 3rd party packages like Jacket.
3. k-nearest neighbor algorithm
4. Computer clusters or a variation of a parallel computing (utilizing GPU cluster technology) for highly calculation-intensive tasks
5. Physical based simulation and physics engines (usually based on Newtonian physics models)
6. Statistical physics
7. Lattice gauge theory
8. Segmentation– 2D and 3D
9. Level-set methods
10. CT reconstruction
11. Fast Fourier transform
12. Tone mapping
13. Audio signal processing
14. Digital image processing
15. Video Processing
16. Global illumination – ray tracing, photon mapping, radiosity among others, subsurface scattering
17. Geometric computing – constructive solid geometry, distance fields, collision detection, transparency computation, shadow generation
18. Scientific computing
19. Bioinformatics
20. Computational finance
21. Medical imaging
22. Computer vision

23. Digital signal processing / signal processing
24. Control engineering
25. Neural networks
26. Database operations
27. Lattice Boltzmann methods
28. Cryptography and cryptanalysis
29. Electronic Design Automation

4.0 Conclusion

GPU functionality has, traditionally, been very limited. In fact, for many years the GPU was only used to accelerate certain parts of the graphics pipeline. Some improvements were needed before GPGPU became feasible.

5.0 Summary

General-purpose computing on graphics processing units is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. There are a variety of computational resources available on the GPU: Programmable processors, Rasterizer, Texture Unit and Frame buffer.

6.0 Tutor Marked Assignment

1.0 What do you understand by GPGPU?
2.0 Identify computational resources available in a GPU.
3.0 Explain some of the GPU techniques discussed in this unit.

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition),* Prentice Hall, 1997, ISBN: 0135309247
3. GPGPU Programming in F# using the Microsoft Research Accelerator system.
4. GPGPU Review, Tobias Preis, European Physical Journal Special Topics 194, 87-119 (2011)

MODULE 5 – Ray tracing, Illumination algorithms and GPGPU

UNIT 4: Visualization, Facial modeling and Animation

Contents **Pages**

1.0 Introduction

Visualization today has ever-expanding applications in science, education, engineering (e.g., product visualization), interactive multimedia, medicine, etc. Typical of a visualization application is the field of computer graphics. The development of animation also helped advance visualization. The use of visualization to present information is not a new phenomenon. It has been used in maps, scientific drawings, and data plots for over a thousand years

2.0 Objectives

On completing this unit, you would be able to:

1. Understand Visualization
2. Understand the Applications of Visualization.
3. Understand Visualization techniques
4. Understand Facial modeling and animation.

3.0 Main Content

3.1 Visualization

Visualization is any technique for creating images, diagrams, or animations to communicate a message. Visualization through visual imagery, has been an effective way to communicate both abstract and concrete ideas since the dawn of man. Examples from history include cave paintings, Egyptian hieroglyphs, Greek geometry, and Leonardo da Vinci's revolutionary methods of technical drawing for engineering and scientific purposes.

Computer graphics has from its beginning been used to study scientific problems. However, in its early days the lack of graphics power often limited its usefulness. The recent emphasis on visualization started in 1987 with the special issue of Computer Graphics on Visualization in Scientific Computing. Since then, there have been several conferences and workshops, co-sponsored by the IEEE Computer Society and ACM SIGGRAPH, devoted to the general topic, and special areas in the field, for example volume visualization.

Most people are familiar with the digital animations produced to present meteorological data during weather reports on television, though few can distinguish between those models of reality and the satellite photos that are also shown on such programs. TV also offers scientific visualizations when it shows computer drawn and animated reconstructions of road or airplane accidents. Some of the most popular examples of scientific visualizations are computer-generated images that show real spacecraft in action, out in the void far beyond Earth, or on other planets. Dynamic forms of visualization, such as educational animation or timelines, have the potential to enhance learning about systems that change over time.

Apart from the distinction between interactive visualizations and animation, the most useful categorization is probably between abstract and model-based scientific visualizations. The abstract visualizations show completely conceptual constructs in 2D or 3D. These generated shapes are completely arbitrary. The model-based visualizations either place overlays of data on real or digitally constructed images of reality or make a digital construction of a real object directly from the scientific data.

Scientific visualization is usually done with specialized software, though there are a few exceptions, noted below. Some of these specialized programs have been released as Open source software, having very often its origins in universities, within an academic environment where sharing software tools and giving access to the source code is common. There are also many proprietary software packages of scientific visualization tools.

Models and frameworks for building visualizations include the data flow models popularized by systems such as AVS, IRIS Explorer, and VTK toolkit, and data state models in spreadsheet systems such as the Spreadsheet for Visualization and Spreadsheet for Images.

3.2 Applications of visualization

As a subject in computer science, scientific visualization is the use of interactive, sensory representations, typically visual, of abstract data to reinforce cognition, hypothesis building, and reasoning. Data visualization is a related subcategory of visualization dealing with statistical graphics and geographic or spatial data (as in thematic cartography) that is abstracted in schematic form.

1. Scientific visualization

Scientific visualization is the transformation, selection, or representation of data from simulations or experiments, with an implicit or explicit geometric structure, to allow the exploration, analysis, and understanding of the data. It is a very important part of visualization and maybe the first one, as the visualization of experiments and phenomena is as old as Science itself. Traditional areas of scientific visualization are flow visualization, medical visualization,

astrophysical visualization, and chemical visualization. There are several different techniques to visualize scientific data, with isosurface reconstruction and direct volume rendering being the more common.

2. **Educational visualization**

Educational visualization is using a simulation normally created on a computer to create an image of something so it can be taught about. This is very useful when teaching about a topic that is difficult to otherwise see, for example, atomic structure, because atoms are far too small to be studied easily without expensive and difficult to use scientific equipment. It can also be used to view past events, such as looking at dinosaurs, or looking at things that are difficult or fragile to look at in reality like the human skeleton.

3. **Information visualization**

Information visualization concentrates on the use of computer-supported tools to explore large amount of abstract data. The term "information visualization" was originally coined by the User Interface Research Group at Xerox PARC and included Dr. Jock Mackinlay. Practical application of information visualization in computer programs involves selecting, transforming, and representing abstract data in a form that facilitates human interaction for exploration and understanding. Important aspects of information visualization are dynamics of visual representation and the interactivity. Strong techniques enable the user to modify the visualization in real-time, thus affording unparalleled perception of patterns and structural relations in the abstract data in question.

4. **Knowledge visualization**

The use of visual representations to transfer knowledge between at least two persons aims to improve the transfer of knowledge by using computer and non-computer-based visualization methods complementarily. Examples of such visual formats are sketches, diagrams, images, objects, interactive visualizations, information visualization applications, and imaginary visualizations as in stories. While information visualization concentrates on the use of computer-supported tools to derive new insights, knowledge visualization focuses on transferring insights and creating new knowledge in groups. Beyond the mere transfer of facts, knowledge visualization aims to further transfer insights, experiences, attitudes, values, expectations, perspectives, opinions, and predictions by using various complementary visualizations.

5. **Visual communication**

Visual communication is the communication of ideas through the visual display of information. Primarily associated with two dimensional images, it includes: alphanumerics, art, signs, and

electronic resources. Recent research in the field has focused on web design and graphically-oriented usability.

3.3 Visualization techniques

The following are examples of some common visualization techniques:

1. **Constructing isosurfaces:** they are normally displayed using computer graphics, and are used as data visualization methods in computational fluid dynamics (CFD), allowing engineers to study features of a fluid flow (gas or liquid) around objects, such as aircraft wings. An isosurface may represent an individual shock wave in supersonic flight, or several isosurfaces may be generated showing a sequence of pressure values in the air flowing around a wing. Isosurfaces tend to be a popular form of visualization for volume datasets since they can be rendered by a simple polygonal model, which can be drawn on the screen very quickly.

2. **Direct volume rendering:** In scientific visualization and computer graphics, **volume rendering** is a set of techniques used to display a 2D projection of a 3D discretely sampled data set. A typical 3D data set is a group of 2D slice images acquired by a CT, MRI, or MicroCT scanner. Usually these are acquired in a regular pattern (e.g., one slice every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element, or voxel represented by a single value that is obtained by sampling the immediate area surrounding the voxel.

3. Streamlines, streaklines, and pathlines: Engineers often use dyes in water or smoke in air in order to see streaklines, from which pathlines can be calculated. Streaklines are identical to streamlines for steady flow. Further, dye can be used to create timelines.[4] The patterns guide their design modifications, aiming to reduce the drag. This task is known as *streamlining*, and the resulting design is referred to as being *streamlined*. Streamlined objects and organisms, like steam locomotives, streamliners, cars and dolphins are often aesthetically pleasing to the eye.

4. Euler diagram: Often, Euler diagrams are augmented with extra structures, such as dots, labels or graphs, showing information about what is contained in the various zones. One significant feature of Euler diagrams is their capacity to visualize complex hierarchies.

5. Chernoff face: display multivariate data in the shape of a human face. The individual parts, such as eyes, ears, mouth and nose represent values of the variables by their shape, size, placement and orientation. The idea behind using faces is that humans easily recognize faces and notice small changes without difficulty. Chernoff faces handle each variable differently. Because the features of the faces vary in perceived importance, the way in which variables are mapped to the features should be carefully chosen (e.g. eye size and eyebrow-slant have been found to carry significant weight)
6. Hyperbolic trees: a **hyperbolic tree** (often shortened as **hypertree**) defines a graph drawing method inspired by hyperbolic geometry. Displaying hierarchical data as a tree suffers from visual clutter as the number of nodes per level can grow exponentially. For a simple binary tree, the maximum number of nodes at a level *n* is 2^n, while the number of nodes for larger trees grows much more quickly. Drawing the tree as a node-link diagram thus requires exponential amounts of space to be displayed.

7. Table, matrix
8. Charts (pie chart, bar chart, histogram, function graph, scatter plot, etc.)
9. Graphs (tree diagram, network diagram, flowchart, existential graph, etc.)
10. Maps
11. parallel coordinates - a visualization technique aimed at multidimensional data
12. Treemap - a visualization technique aimed at hierarchical data
13. Venn diagram
14. Timeline
15. Brushing and linking
16. Cluster diagram or dendrogram
17. Ordinogram

3.4 Facial Modeling and Animation

Computer facial animation is primarily an area of computer graphics that encapsulates models and techniques for generating and animating images of the human head and face. Due to its subject and output type, it is also related to many other scientific and artistic fields from

psychology to traditional animation. The importance of human faces in verbal and non-verbal communication and advances in computer graphics hardware and software have caused considerable scientific, technological, and artistic interests in computer facial animation.

Although development of computer graphics methods for facial animation started in the early 1970s, major achievements in this field are more recent and happened since the late 1980s. Computer facial animation includes a variety of techniques from morphing to three-dimensional modeling and rendering. It has become well-known and popular through animated feature films and computer games but its applications include many more areas such as communication, education, scientific simulation, and agent-based systems (for example online customer service representatives).

Human facial expression has been the subject of scientific investigation for more than one hundred years. Study of facial movements and expressions started from a biological point of view. After some older investigations, for example by John Bulwer in late 1640s, Charles Darwin's book *The Expression of the Emotions in Men and Animals* can be considered a major departure for modern research in behavioural biology.

More recently, one of the most important attempts to describe facial activities (movements) was Facial Action Coding System (FACS). Introduced by Ekman and Friesen in 1978, FACS defines 46 basic facial Action Units (AUs). A major group of these Action Units represent primitive movements of facial muscles in actions such as raising brows, winking, and talking. Eight AUs are for rigid three-dimensional head movements, i.e. turning and tilting left and right and going up, down, forward and backward. FACS has been successfully used for describing desired movements of synthetic faces and also in tracking facial activities.

Computer based facial expression modelling and animation is not a new endeavour. The earliest work with computer based facial representation was done in the early 1970s. The first three-dimensional facial animation was created by Parke in 1972. In 1973, Gillenson developed an interactive system to assemble and edit line drawn facial images. And in 1974, Parke developed a parameterized three-dimensional facial model.

3.5 Facial Animation Techniques

3.5.1 2D Animation

Two-dimensional facial animation is commonly based upon the transformation of images, including both images from still photography and sequences of video. Image morphing is a technique which allows in-between transitional images to be generated between a pair of target still images or between frames from sequences of video. These morphing techniques usually consist of a combination of a geometric deformation technique, which aligns the target images,

and a cross-fade which creates the smooth transition in the image texture. An early example of image morphing can be seen in Michael Jackson's video for "Black or White”

Another form of animation from images consists of concatenating together sequences captured from video. In 1997, Bregler et al. described a technique called video-rewrite where existing footage of an actor is cut into segments corresponding to phonetic units which are blended together to create new animations of a speaker. Video-rewrite uses computer vision techniques to automatically track lip movements in video and these features are used in the alignment and blending of the extracted phonetic units. This animation technique only generates animations of the lower part of the face, these are then composited with video of the original actor to produce the final animation.

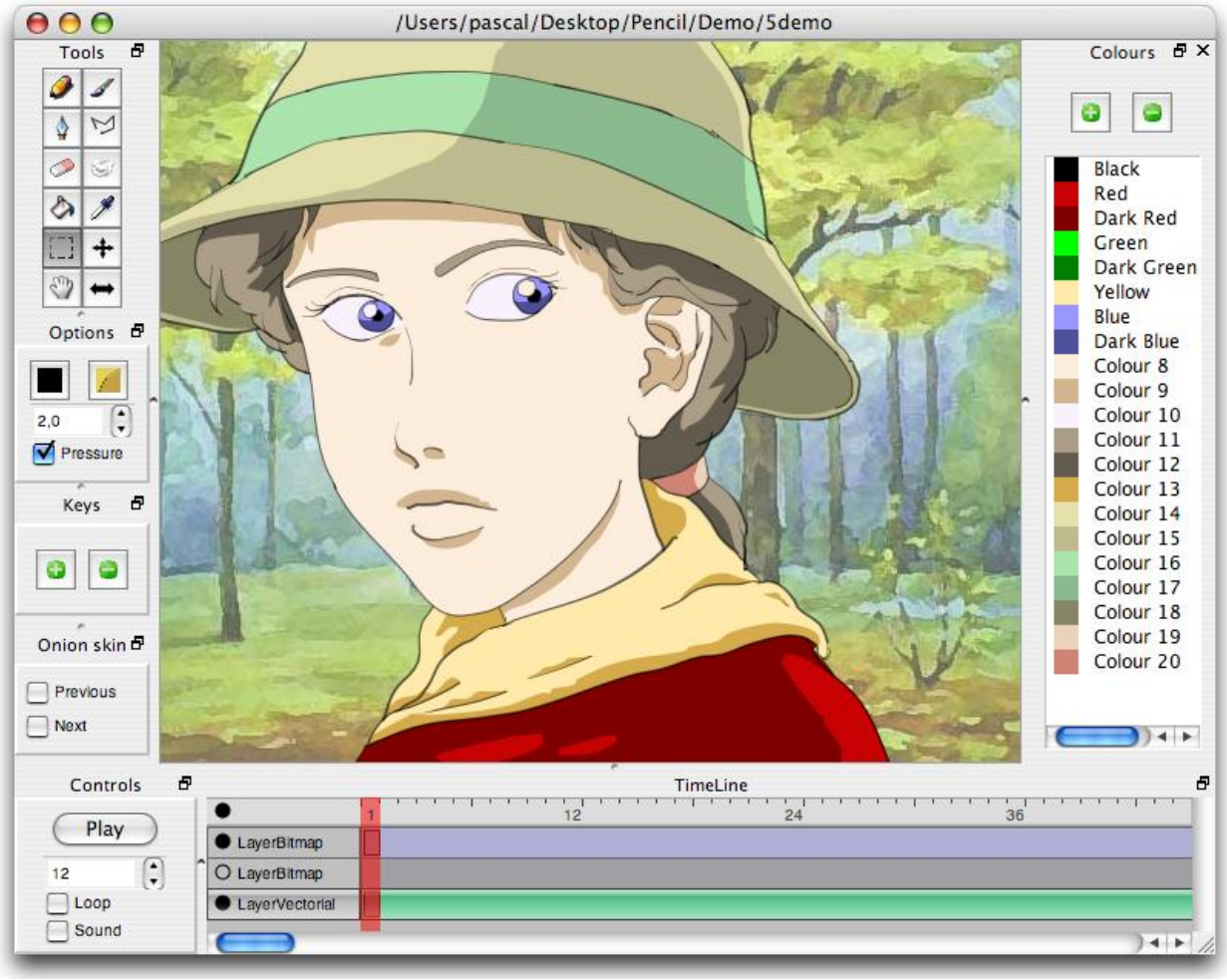


Figure 5.4(a): 2D traditional animation (pencilanimation.org)

3.5.2 3D Animation

Three-dimensional head models provide the most powerful means of generating computer facial animation. One of the earliest works on computerized head models for graphics and animation was done by Parke. The model was a mesh of 3D points controlled by a set of conformation and expression parameters. The former group controls the relative location of facial feature points such as eye and lip corners. Changing these parameters can re-shape a base model to create new heads. The latter group of parameters (expression) are facial actions that can be performed on face such as stretching lips or closing eyes. This model was extended by other researchers to include more facial features and add more flexibility. Different methods for initializing such "generic" model based on individual (3D or 2D) data have been proposed and successfully implemented. The parameterized models are effective ways due to use of limited parameters, associated to main facial feature points. The MPEG-4 standard defines a minimum set of parameters for facial animation.

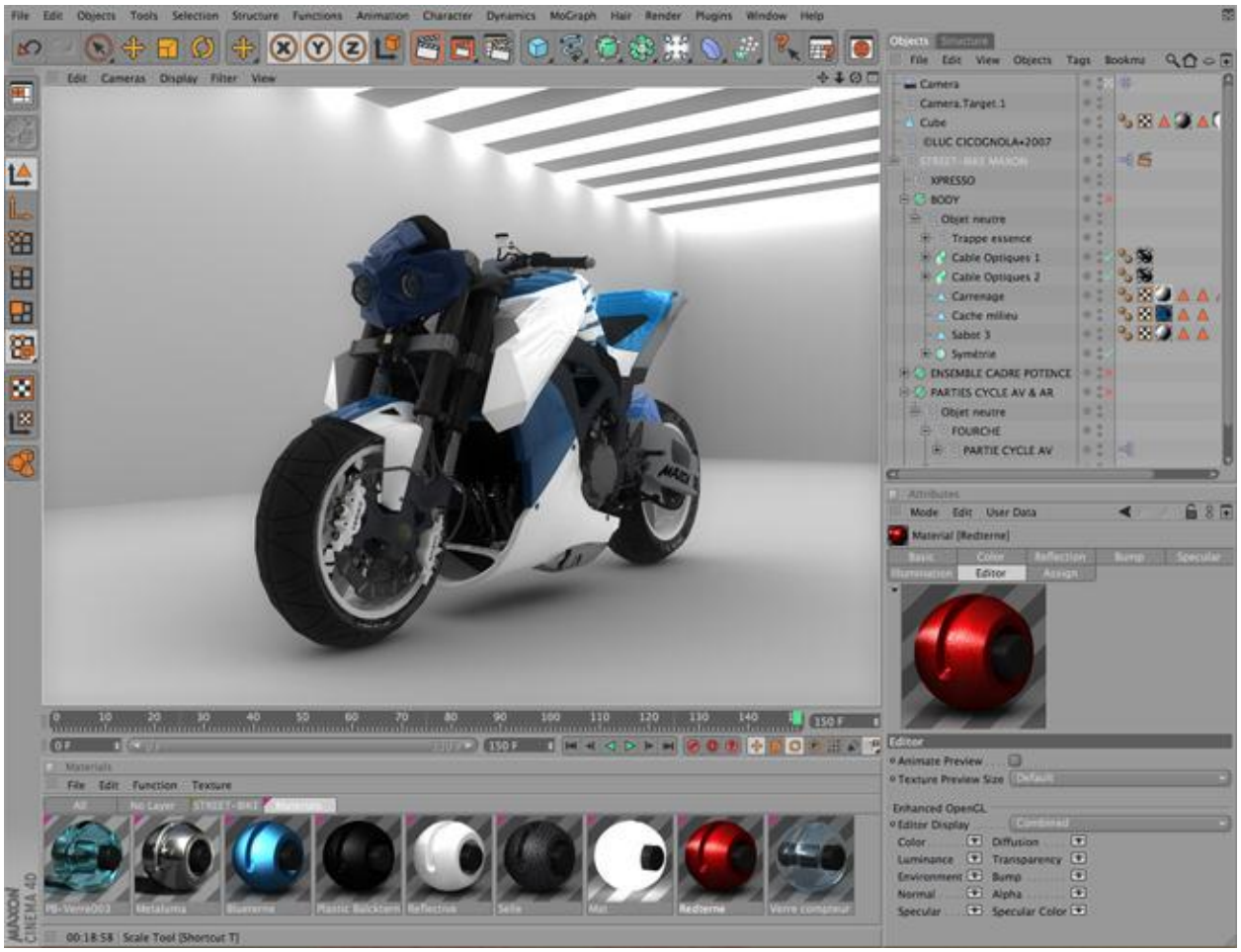


Figure 5.4(b): 3D animation (lawalstemescal.com)

Animation is done by changing parameters over time. Facial animation is approached in different ways, traditional techniques include

a) shapes/morph targets,
b) skeleton-muscle systems,
c) bones/cages,
d) motion capture on points on the face and
e) knowledge based solver deformations.

a. *Shape based systems* offer a fast playback as well as a high degree of fidelity of expressions. The technique involves modelling portions of the face mesh to approximate expressions and visemes and then blending the different sub meshes, known as morph targets or shapes. Perhaps the most accomplished character using this technique was Gollum, from *The Lord of the Rings*. Drawbacks of this technique are that they involve intensive manual labor, specific to each character and must be animated by slider parameter tables.
b. *Skeletal Muscle systems*, physically based head models form another approach in modeling the head and face. Here the physical and anatomical characteristics of bones, tissues, and skin are simulated to provide a realistic appearance (e.g. spring-like elasticity). Such methods can be very powerful for creating realism but the complexity of facial structures make them computationally expensive, and difficult to create. Considering the effectiveness of parameterized models for communicative purposes (as explained in the next section), it may be argued that physically based models are not a very efficient choice in many applications. This does not deny the advantages of physically based models and the fact that they can even be used within the context of parameterized models to provide local details when needed. Waters, Terzopoulos, Kahler, and Seidel (among others) have developed physically based facial animation systems.
c. 'Envelope Bones' or 'Cages' are commonly used in games. They produce simple and fast models, but are not prone to portray subtlety.
d. Motion capture uses cameras placed around a subject. The subject is generally fitted either with reflectors (passive motion capture) or sources (active motion capture) that precisely determine the subject's position in space. The data recorded by the cameras is then digitized and converted into a three-dimensional computer model of the subject. Until recently, the size of the detectors/sources used by motion capture systems made the technology inappropriate for facial capture. However, miniaturization and other advancements have made motion capture a viable tool for computer facial animation. The main difficulties of motion capture are the quality of the data which may include vibration as well as the retargeting of the geometry of the points. A recent technology developed at the Applied Geometry Group and Computer Vision Laboratory at ETH Zurich achieves real-time performance without the use of any markers using a high speed structured light scanner. The system is based on a robust offline face tracking stage which trains the system with different facial expressions. The matched sequences are used to build a person-specific linear face model that is subsequently used for online face tracking and expression transfer.
e. Deformation Solver Face Robot.

3.6 Speech Animation

Speech is usually treated in a different way to the animation of facial expressions, this is because simple keyframe-based approaches to animation typically provide a poor approximation to real speech dynamics. Often visemes are used to represent the key poses in observed speech (i.e. the position of the lips, jaw and tongue when producing a particular phoneme), however there is a great deal of variation in the realisation of visemes during the production of natural speech. The source of this variation is termed coarticulation which is the influence of surrounding visemes upon the current viseme (i.e. the effect of context). To account for coarticulation current systems either explicitly take into account context when blending viseme keyframes or use longer units such as diphone, triphone, syllable or even word and sentence-length units.

One of the most common approaches to speech animation is the use of dominance functions introduced by Cohen and Massaro (1993). Each dominance function represents the influence over time that a viseme has on a speech utterance. Typically the influence will be greatest at the center of the viseme and will degrade with distance from the viseme center. Dominance functions are blended together to generate a speech trajectory in much the same way that spline basis functions are blended together to generate a curve. The shape of each dominance function will be different according to both which viseme it represents and what aspect of the face is being controlled (e.g. lip width, jaw rotation etc.). This approach to computer-generated speech animation can be seen in the Baldi talking head.

Other models of speech use basis units which include context (e.g. diphones, triphones etc.) instead of visemes. As the basis units already incorporate the variation of each viseme according to context and to some degree the dynamics of each viseme, no model of coarticulation is required. Speech is simply generated by selecting appropriate units from a database and blending the units together. This is similar to concatenative techniques in audio speech synthesis. The disadvantage to these models is that a large amount of captured data is required to produce natural results, and whilst longer units produce more natural results the size of database required expands with the average length of each unit.

Finally, some models directly generate speech animations from audio. These systems typically use hidden markov models or neural nets to transform audio parameters into a stream of control parameters for a facial model. The advantage of this method is the capability of voice context handling, the natural rhythm, tempo, emotional and dynamics handling without complex approximation algorithms. The training database is not needed to be labeled since there are no phonemes or visemes needed; the only needed data is the voice and the animation parameters. An example of this approach is the Johnnie Talker system

3.7 Face Animation Languages

Many face animation languages are used to describe the content of facial animation. They can be input to a compatible "player" software which then creates the requested actions. Face animation languages are closely related to other multimedia presentation languages such as SMIL and VRML. Due to the popularity and effectiveness of XML as a data representation mechanism, most face animation languages are XML-based. For instance, this is a sample from Virtual Human Markup Language (VHML):

```
<vhml>
   <person disposition="angry">
     First I speak with an angry voice and look very angry,
     <surprised intensity="50">
       but suddenly I change to look more surprised.
     </surprised>
   </person>
 </vhml>
```

More advanced languages allow decision-making, event handling, and parallel and sequential actions. Following is an example from Face Modeling Language (FML):

```
 <fml>
   <act>
     <par>
        <hdmv type="yaw" value="15" begin="0" end="2000" />
        <expr type="joy" value="-60" begin="0" end="2000" />
     </par>
     <excl event_name="kbd" event_value="" repeat="kbd;F3_up" >
        <hdmv type="yaw" value="40" begin="0" end="2000"event_value="F1_up"/>
        <hdmv type="yaw" value="-40" begin="0" end="2000"
event_value="F2_up"/>
     </excl>
   </act>
 </fml>
```

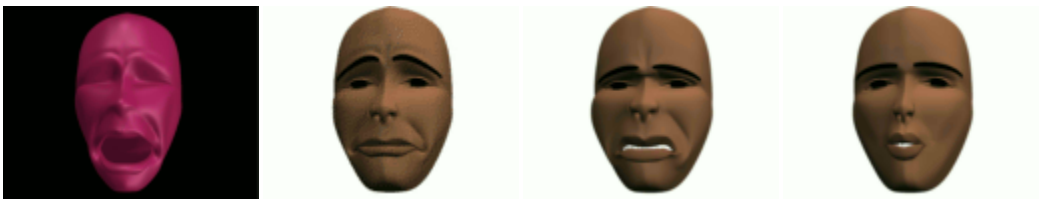


Figure 5.4(a): Examples of facial models

4.0 Conclusion

Model-based visualizations either place overlays of data on real or digitally constructed images of reality or make a digital construction of a real object directly from the scientific data.

5.0 Summary

In this unit, we have surveyed visualization, its application areas and facial animation. Visualization is said to be any technique for creating images, diagrams, or animations to communicate a message. Visualization today has ever-expanding applications in science, education, engineering, interactive multimedia, medicine

6.0 Tutor Marked Assignment

1. What do you understand by visualization?
2. Identify Application areas.
3. Identify and discuss some the visualization techniques mentioned in this unit.
4. What is facial modeling?

7.0 References/Further Reading

1. Ed Angel (1991) *Interactive Computer Graphics- A Top-Down Approach with OpenGL*, Fifth Edition, Addison-Wesley 2009 ISBN 0-321-53586-3
2. Donald Hearn and M. Pauline Baker (1997) *Computer Graphics, C Version (2nd Edition)* Prentice Hall, 1997, ISBN: 0135309247
3. Battiti, Roberto; Mauro Brunato (2011). *Reactive Business Intelligence. From Data to Models to Insight.*. Trento, Italy: Reactive Search Url: http://www.reactivebusinessintelligence.com/.
4. Matthew Ward, Georges Grinstein, Daniel Keim (2010), *Interactive Data Visualization: Foundations, Techniques, and Applications*, Hardcover (May, 2010). ISBN: 1568814735
5. Marty R. (2008), *Applied Security Visualization.* Pearson Education, 2008. ISBN: 0321510105
6. Visualization Handbook (Hardcover) by Charles D. Hansen, Chris Johnson, Academic Press (June, 2004). ISBN: 0123875822.
7. Cohen and Masoro (1993) Use of dominant functions.
8. Bregler et al (1997). Video-rewrite techniques.