CPSC 453 Course Notes

Faramarz Samavati *

Updated in September 2024

*Please notify samavati@ucalgary.ca of typos or mistakes.

Contents

Introduction

This document contains content from the University of Calgary's introductory computer graphics class, CPSC 453. It is designed to complement the material covered in lectures, although it may not cover all lecture topics in details. Originally created in 2008 with assistance from Luke Olsen, a former PhD student and the TA of the course, it has undergone multiple updates. In 2020, Benjamin Ulmer, a former MSc student, provided additional updates and writing for chapter 2. Throughout the semester, any necessary corrections and changes will be included and communicated via the course web page.

1 Graphics Systems

Computer graphics is concerned with modelling, rendering, and displaying synthetic images. A typical graphics system involves several key components that work together to process user inputs and generate visual outputs (See Fig. 1.1):

- **Interaction (Input Devices)**: This includes devices like keyboards and joysticks that users interact with to provide input.
- **Application Program**: The software that processes the input and generates the necessary graphics commands.
- **Graphics Hardware**: Specialized hardware that executes the graphics commands and processes the data.
- **Output Devices**: Various display devices such as *raster displays* (including CRT, LCD, and LED monitors), head-mounted stereo displays and 3D printers render the final visual output.

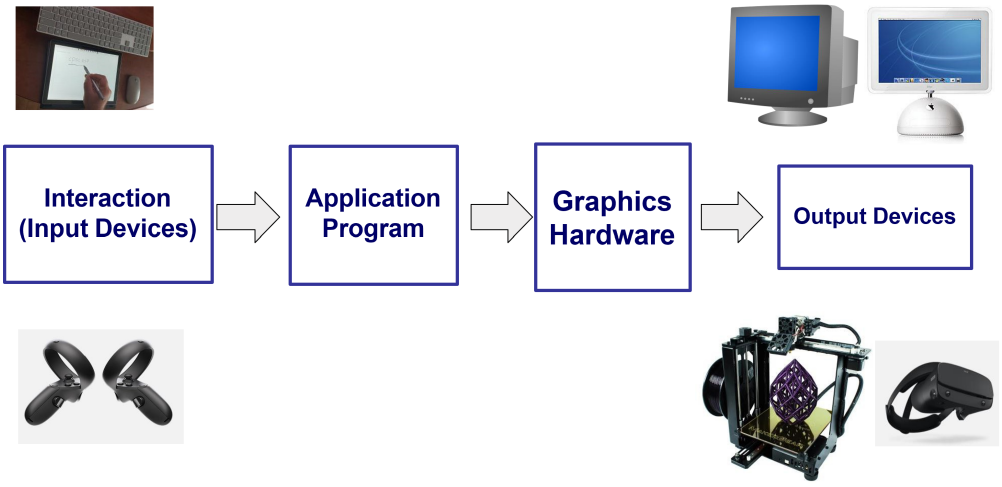


Figure 1.1: A typical graphics system

1.1 Raster Displays

Raster displays are among the most common output devices in computer graphics. They are capable of showing *raster images*, which consist of equal-length lines of picture elements, or *pixels*. These pixels are arranged in a grid pattern, and each pixel represents a tiny portion of the overall image with a constant color (see Fig. 1.2).

The **resolution** of a raster display (or raster image) refers to the number of pixels along each axis of the display grid. Higher resolution means more pixels are used to represent the image, resulting in finer detail and better image quality. Raster displays need to refresh the screen multiple times per second, typically between 60 and 120 times, to maintain a stable and flicker-free image.

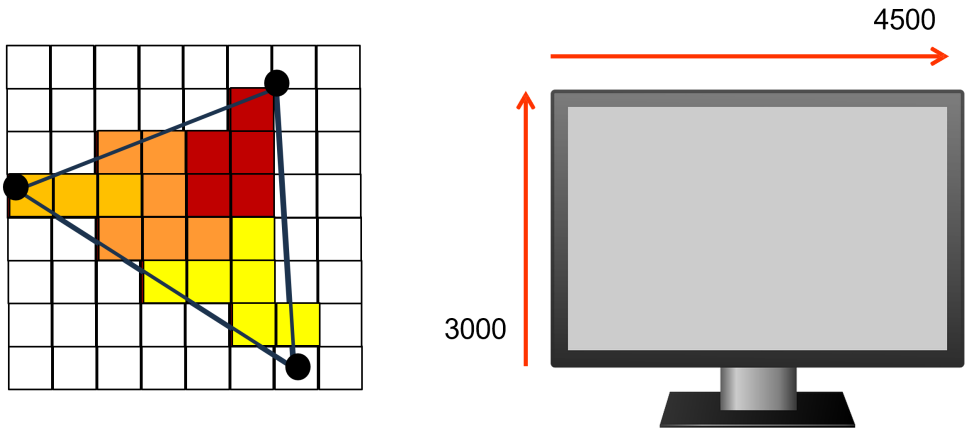


Figure 1.2: An eight by eight raster image and a typical raster display

An important question to consider is how the color of pixels is represented. The detailed answer to this question will be provided in the next chapter. However, here we will discuss the basic idea and provide a very high-level description of pixel colors. There are several methods for representing color digitally, with RGB being one of the most common.

The RGB model uses three primary colors:

- **R** $[0, 1] \rightarrow$ Represents the intensity of red (0 for no red, 1 for maximum red).
- **G** $[0, 1] \rightarrow$ Represents the intensity of green.
- **B** $[0, 1] \rightarrow$ Represents the intensity of blue.

In this model, each color can be thought of as a 3-dimensional real value, where each dimension corresponds to the intensity of one of the primary colors.

For storing raster display, instead of using floating points for each pixel's RGB values, it is more efficient to employ a bit encoding method s. For instance, using 8 bits per color or 24 bits for color can be a more effective approach. During encoding, not all possible intensity values can be represented, as a range of intensity values is grouped together and assigned to a sequence of bits (quantization). For instance, when using 8-bit color encoding, only 256 different colors can be represented.

1.2 Graphics Pipeline

In computer graphics, we typically develop a program to generate a scene that consists of geometric objects (such as polyhedrons) along with specifications for color, material, and light. Our program utilizes an Application Program Interface (API) like OpenGL to process the scene data. The API leverages the Graphics Hardware to process the data and transmit the visual output to the graphics hardware, usually raster displays. This pipeline (demonstrated in Fig. 1.3) is crucial for rendering 3D graphics by converting raw data into visual images.

Current graphics hardware usually includes a Graphics Processing Unit (GPU). This specialized processor, typically found on graphics cards, is designed to accelerate graphics rendering

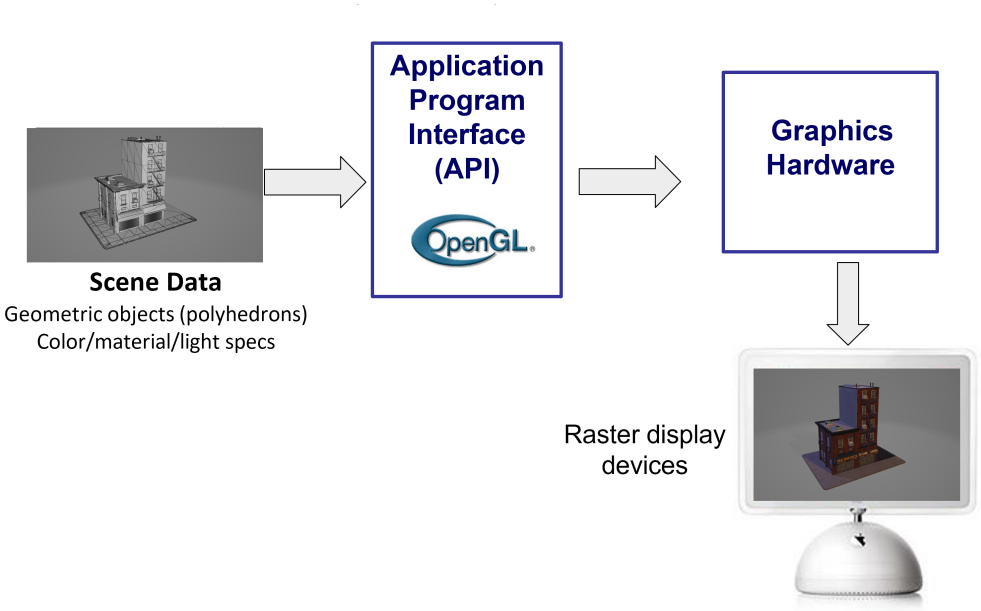


Figure 1.3: A typical graphics pipeline using OpenGL API

and perform parallel processing tasks. While GPUs were originally created to enhance computer graphics, they have since been optimized for AI applications as well.

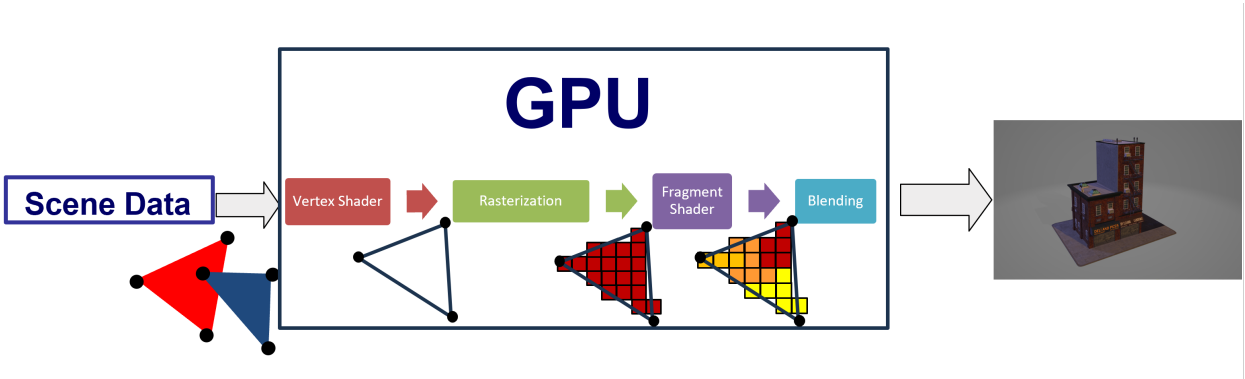


Figure 1.4: GPU Pipeline

The diagram in Fig. 1.4 illustrates the stages involved in rendering graphics through a GPU. It starts with scene data, which is processed through several stages:

1. **Vertex Processing**: This stage handles the transformation and creation of geometric primitives.

2. **Rasterization**: Converts the geometric primitives into fragments, which are essentially potential pixels. This process determines which pixels on the screen will be affected by each primitive.

3. **Fragment Shader**: Processes each fragment to determine its final color and other attributes,

such as texture and lighting effects. This stage allows for detailed and complex visual effects to be applied to each pixel.

4. **Blending**: Combines the processed fragments to form the final pixel colors.

Vertex and fragment shaders are types of computer programs that run on the GPU and can be modified for various purposes. More detailed learning about shaders and the pipeline will occur progressively in the tutorials.

Given that the final raster image needs to be sent to the displays multiple times (refreshing), it's more efficient to store the current raster image in a dedicated part of the graphics hardware (i.e., buffer) instead of repeating all of these steps. This saved buffer helps avoid rerunning the entire GPU pipeline stages to recreate the current image.

1.3 The Frame Buffer

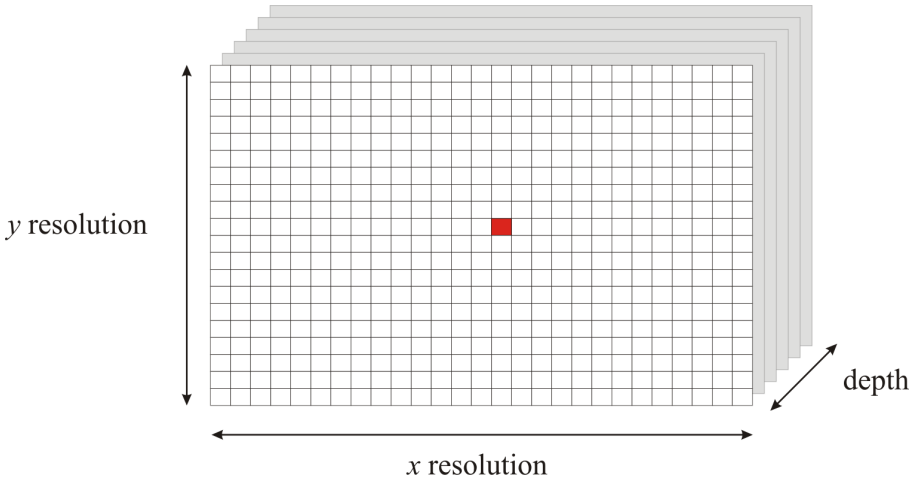


Figure 1.5: A frame buffer holds an intensity value for every pixel in a raster display, a total of *x resolution* $\times$ *y resolution* pixels. The *depth* of the frame buffer determines how many colors can be represented.

Raster displays are always connected to a frame buffer. A **frame buffer** is a region of memory that is large enough to hold a value (color or intensity) for every pixel in the display. The frame buffer may be physically located on the display, or it may reside in the host computer. For example, the frame buffer on early IBM PCs resided in a special region of main memory, while modern graphics cards contain several high-speed frame buffers in their on-board memory. Figure 1.5 illustrates the basic elements of a frame buffer.

The **depth** of a frame buffer is the number of bits available to each pixel. The depth– often referred to as the *color depth* or the number of *planes* – determines how many colors the frame buffer can represent. Note that the depth of the frame buffer should be related to the number of colors available on the connected raster display.

As we discussed, most raster displays separate the color of a particular pixel into three color channels or components: red, green, and blue color. The final color of a pixel is the sum of each

color channel. There are other schemes used in raster displays – such as CMYK (cyan, magenta, yellow, and black), commonly used in printers – but RGB color is the standard in computer graphics.

Example 1.1

How many unique colors can an RGB frame buffer with 6 planes represent? Also, what would be the frame buffer size for a display screen of eight by eigh pixels with a depth of six bits (see Fig. 1.6)?

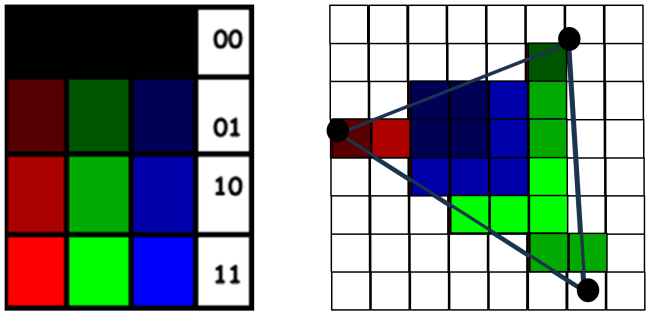


Figure 1.6: A small screen example

Answer:

Each RGB channel is allotted $6/3 = 2$ bits. These 2 bits can produce $2^2 = 4$ colors in each channel. So, the total number of colors that can be represented is

$$4 * 4 * 4 = 64 .$$

The beuffer size can be found by the size of screen times to the color depth. So, here we have $8 * 8 * 6 = 384$ bits $= 48$ bytes.

How about 24 bit planes?

Here, each RGB channel is given 8 bits and can represent 2^8 shades. All channels combines can represent $2^{8^3} = 2^{24} = 16,777,216$, or 16.7 million, colors.

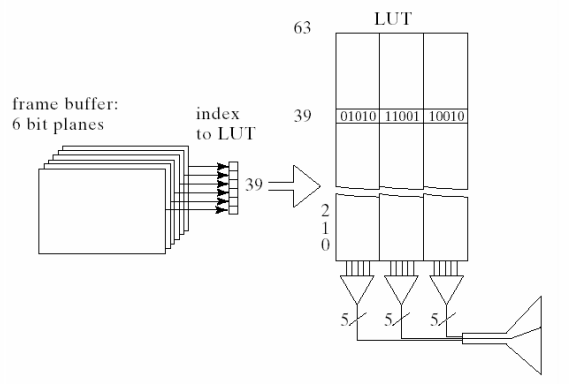


Figure 1.7: Indexed color system: each entry in the frame buffer is an index into a color table.

1.3.1 Indexed Color Systems

The number of colors that can be represented by a frame buffer is determined strictly by its depth, creating a tradeoff between memory usage and color range. Though memory in modern computers is cheap and plentiful, early systems were more concerned with limiting the frame buffer's memory footprint. This concern lead to the question, *Is it possible to have a large color range in a small frame buffer?*

Indexed color systems use an alternative method of associating pixel values with colors: instead of an entry in the frame buffer holding an RGB intensity, the frame buffer holds indices into a **lookup table** (LUT) (Figure 1.7).

Suppose that our display system has a color depth of k bits (k planes), and that each entry of the LUT holds m bits per color channel. Then, the system can represent 2^{3m} different colors, of which 2^k can be displayed at any one time. The set of possible colors available to the system at any one time (i.e. the colors held in the LUT) is called the *palette*.

Example 1.2

Consider a system with $k = 8$ and $m = 8$.

How many colors can be represented in total?

> Each entry of the LUT holds $m = 8$ bits per color channel, or 24 bits total. Thus, there are $2^{3m} = 2^{24} = 16.7$ `million` colors that can be represented.

How many colors can be held in the LUT?

> Each entry of the frame buffer holds an index with $k = 8$ bits. Thus, the frame buffer can reference up to $2^8 = 256$ LUT indices, which is the maximum size of the LUT.

The advantage of using LUTs is the reduced memory cost. The size of the frame buffer plus the palette in an indexed system is much smaller than the size of a frame buffer with the same

depth as the indexed palette. However, indexed color systems are slower than non-indexed because determining the color of a pixel requires two memory accesses instead of one: first the frame buffer is accessed to get the index, and then the LUT is accessed to get the color value.

1.4 Common Display Devices

This section details some of the most common display devices, both currently and historically.

1.4.1 CRT Monitors

While less common today, **Cathode-ray tube** (CRT) technology was once the primary technology used in television sets and computer displays. In order to produce a wide range of colors, there are three electron guns in a CRT monitor: one gun glows red, one glows green, and the other glows blue. A deflector coil focuses all three beams closely together so that our eyes perceive one composite color that is the sum of the three components.

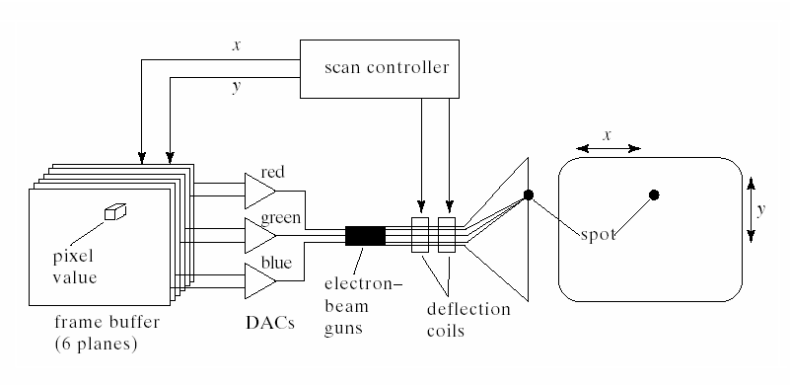


Figure 1.8: CRT monitors. Each color channel from the frame buffer is sent through a different DAC, and the output analog voltage determines the intensity of the corresponding electron gun. The guns are all focused on a single spot to create a composite color on the screen.

To display an image from a frame buffer, the monitor takes a series of steps (Figure 1.8):

1. Each RGB component of the pixel value is sent to a separate digital-to-analog converter (DAC), which converts a digital bit value to an actual voltage.

2. The voltage levels control a gun inside the display, which in turn excites three electron beams with intensity proportional to the voltages.

3. The deflection coils divert the three beams to the correct (x, y) pixel location.

Because the glow of the phosphor dots fades quickly when the stimulus is removed, the display of a CRT must be redrawn or *refreshed* constantly. To avoid a perceptible flicker, the refresh rate is usually greater than 60 times per second (60 Hz). During each "refresh interval," the entire image from the frame buffer is sent to the screen; this is called the *scanning* process.

1.4.2 Liquid-Crystal Displays

As technology has improved, *flat panel* displays have become the primary display devices in use today. There are several types of flat panel monitors, but one of the most common are **liquid crystal displays** (LCDs).

LCDs are used in laptop computers, calculators, phones, television sets, and desktop monitors. These devices produce a picture by passing polarized light from an internal source through a liquid crystal material that can be aligned to either block or transmit the light. The display still needs to be refreshed, because the crystal molecules return to their initial alignment.

Active matrix panels are a recent LCD display technology that employ a tiny transistor at each pixel location. The transistor responds to an electric field and adjusts the liquid crystals by an amount proportional to the field, meaning that the display does not need to be refreshed. This removes another source of radiation. The common technology used in active matrix flat panels is the *thin-film transistor*, or TFT.

1.4.3 3D Displays

The human ability to perceive depth is due to the fact that we have two eyes that are separated by approximately 7.5 cm. This separation allows each eye to get a slightly different view of the world, which are then combined together by the brain to create a single viewpoint as well as perceive depth.

Stereographic display systems are based on the human visual system. A 3D effect is created by using a pair of images from slightly different viewpoints that correspond to each eye of the viewer. The stereo images are then composited together in the display to create a three-dimensional image. The head-mounted displays in virtual reality (VR) systems typically use stereographic displays, for instance. Stereographic displays can typically be viewed by only one person at a time.

In a more recent technology called *holographic* monitors, several viewers can simultaneously view a 3D picture in front of the display, from different perspectives. In these displays, the pair of stereo images are provided by complicated lens configurations such as hexagonal lenses.

Sharp 3D LCDs use a technique called "the parallax barrier." This barrier controls the path traveled by the light between the display and the viewer, so that the left and right eye receives a slightly different image.

2 Light and Colour

The human sense of sight operates by gathering and interpreting light from the environment. Understanding human sight is essential in order to create synthetic images; therefore, it is also necessary to understand light. This section provides a brief background on what light and colour are, how humans perceive them, and how we represent them digitally.

2.1 Electromagnetic Radiation

From the name, electromagnetic radiation is composed of waves that propagate through the electric and magnetic fields. Light perceptible by humans is simply electromagnetic waves within a specific spectrum, known as visible light. The bounds of the visible spectrum are approximately 380–740 nm. Figure 2.1 shows the spectrum of visible light in the context of the full electromagnetic spectrum.

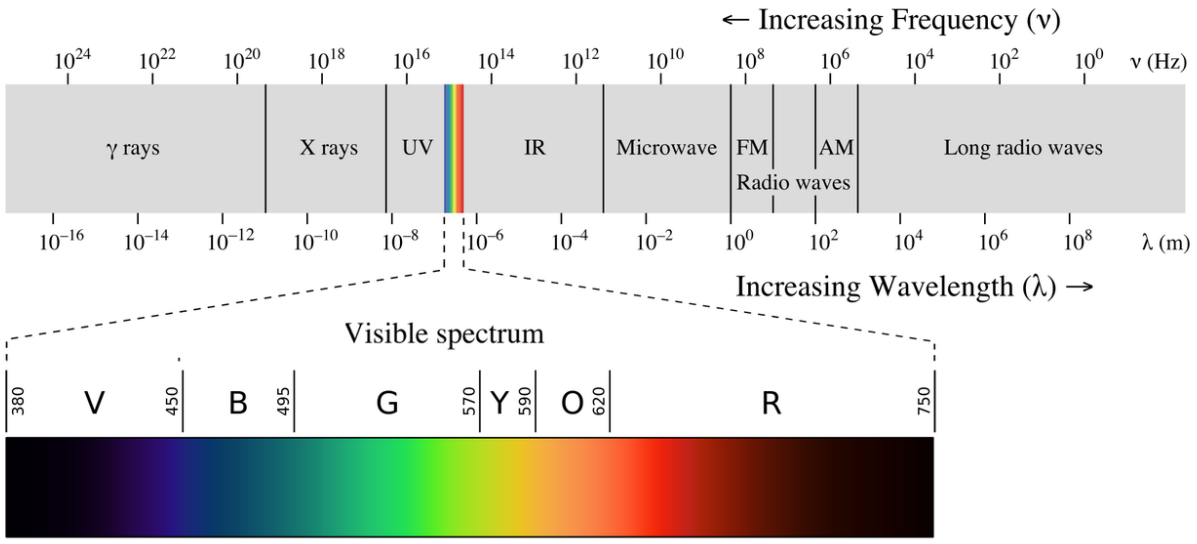


Figure 2.1: The electromagnetic spectrum with the visible portion of magnified for clarity. Source: https://commons.wikimedia.org/wiki/File:EM_spectrumrevised.png

Different wavelengths of light within the visible spectrum are distinguished by what we perceive as colour. Pure wavelengths produce what we call *monochromatic* colours; however, we also perceive different combinations of wavelengths as unique colours. For example, "white" light is an equal mixture of all wavelengths of light in the visible spectrum. The colour of objects that do not emit light is determined by the wavelengths of light they reflect versus the ones they absorb. Emissive objects, like display devices, can create different colours by emitting varying combinations of different wavelengths of light.

2.2 Human Light Perception

The eye is a complicated mechanism with many different organs responsible for various tasks. Briefly, light enters an opening in the eye—known as the pupil—and is focused by a lens, forming an image on the back of the eye. This part of the eye where images are formed is called the retina. The retina is home to approximately 100 million photoreceptors that produce neural signals in response to light stimulus. The brain then processes these neural signals, resulting in our sense of sight.

2.2.1 Photoreceptors and Colour Perception

There are two types of photoreceptors in the human eye: rod and cone cells. Rods are the more numerous of the two, with approximately 92 million in the retina compared to six-seven million cones. Rods are also much more sensitive than cones, able to produce a neural response from a single photon. Because of this, rods are almost entirely responsible for night vision. Rods are mostly concentrated on the retina's outer edges and are largely responsible for peripheral vision. Cones, on the other hand, are tightly packed in the centre of the retina. Cones also perceive finer details and have a faster response time, meaning they can detect changes in stimulus faster than rods. Finally, cone cells are what allow for the perception of colour.

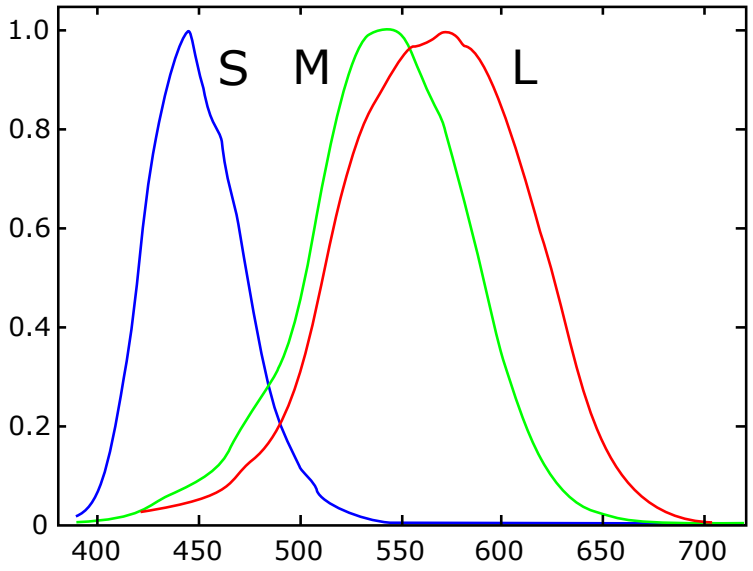


Figure 2.2: Normalized spectral sensitivity of short (S), medium (M), and long (L) cone cells. Source: https://commons.wikimedia.org/wiki/File:Cones_SMJ2_E.svg

There are three types of cone cells, each of which responds uniquely to different wavelengths of light (Figure 2.2). The three types are referred to as short, medium, and long, corresponding to the wavelengths of light they are most sensitive to. The terms blue, green, and red are also often used, corresponding to the colours associated with the respective wavelengths. Since each type of cell responds differently to different wavelengths (and combinations of wavelengths) of light, the amount of response from each type of cell is what our brains interpret as colour.

As humans have three types of cone cells, we have what is called trichromatic vision. However, some people have defects that change how they perceive colour. Damage or irregularity in any of

the types of cone cells affects how an individual perceives colour. Other species also have different numbers of cone cell types. For example, while it is a common misconception that dogs see in black and white, in actuality, they have two types of cone cells and see colour in a way that is similar to humans with red-green colour blindness (deuteranopia).

2.2.2 Perceptual Brightness

Another key aspect of human light perception is how we perceive different intensities of light. Increasing the amount of light that enters the eye from an object increases the perceived "brightness" of said object. However, like most human senses, the relationship between the intensity of the physical stimulus and the perceived magnitude is not linear. Instead, the relationship is modelled much better by a power-law expression, where *relative* changes in the stimulus result in proportional *relative* changes in perception. Simply put, perceived brightness (ψ) is a function of the physical stimulus (i) raised to some exponent a. Formally,

$$\psi(i) = ki^a, \tag{1}$$

where k is a constant determined by the units used. The exact value of a depends on the viewing conditions; however, under most conditions, the value is in the range $[0.4, 0.6]$.

2.3 Gamma Correction

Accounting for human perception of light when storing information about images is crucial for ensuring optimal memory usage. For a fixed data size, there are a fixed number of values that can be used for encoding information about the intensity of light—typically 8 bits yielding 256 values. Using these values to encode intensity linearly is not optimal: high-intensity values have too many values allocated, and low-intensity regions not enough. This issue is illustrated in Figure 2.3. Instead, values should be encoded using the same power-law expression that models perceptual brightness.

The process of using a power-law expression to encode and decode light intensity is known as gamma correction. The definition is the same as Equation 1, but typically with different variable names:

$$v_{\text{out}} = kv_{\text{in}}^{\gamma}.$$

Furthermore, instensity values are usually normalized in the range $[0, 1]$, in which case $k = 1$. A value of $\gamma < 1$ is called an encoding gamma, the application of which results in gamma compression. Conversely, a value of $\gamma > 1$ is a decoding gamma and results in gamma expansion.

Knowing the gamma encoding of an image is essential when performing any operations that blend or average light intensities. These types of operations are linear and therefore require linear values to be physically meaningful. Thus, gamma-encoded values must first be decoded before performing such operations. Only once a final value is obtained can the results be gamma-encoded again. As an aside, many commercial and popular applications do not handle gamma correction correctly, including old versions of photoshop and most web browsers. Since around 2005, graphics cards have had hardware support for encoding and decoding gamma (more accurately, the sRGB transfer function, discussed later) at no additional cost. Refer to Section 3.5 of the programming notes for information on how to do this with OpenGL and ensure your programs are "gamma correct."

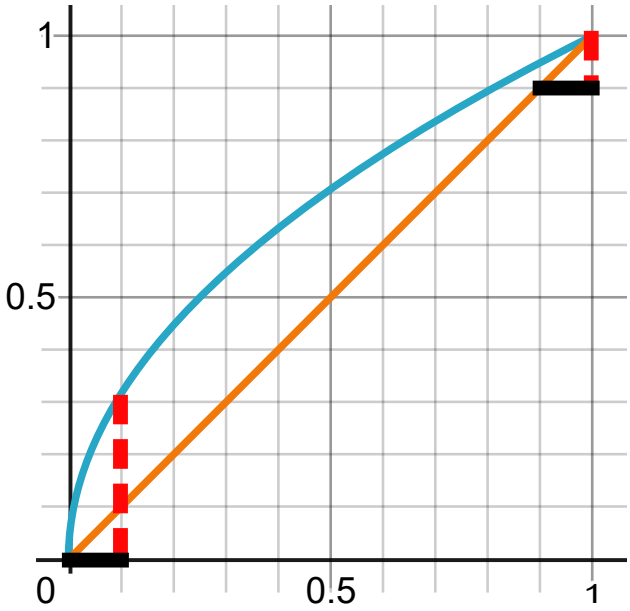


Figure 2.3: With linear brightness encoding, the distribution of light intensities does not align with human perception. For low values, a small increase in brightness results in a large perceived difference, whereas for high values, the same change results in a significantly smaller perceived difference.

It is important to note that when using floating-point numbers to store light intensities, gamma encoding is unnecessary and perhaps even detrimental. This is because the distribution of floating-point values already approximates a gamma curve, with more possible values for small numbers than larger ones.

2.4 Colour Models and Spaces

With an understanding of the mechanism behind human colour perception, we can now look at different methods for representing and quantifying colours. A colour *model* defines a general technique for describing colours, whereas a colour *space* provides a method for objectively specifying a set of colours, known as its gamut. This section briefly describes some of the most common and essential colour models and spaces in the context of computer graphics.

2.4.1 LMS Colour Space

The most straightforward method for encoding a colour is by describing the response it elicits in the three types of cone cells in the human eye (refer back to Figure 2.2). This method is known as the LMS colour space, named after the three types of cone cells. While intuitive, the LMS colour space is not of much use in computer graphics. Instead, it is mostly used for studying biologically driven processes, like colour blindness and chromatic adaptation.

2.4.2 CIE 1931

In 1931, the International Commission on Illumination (CIE) created two colour spaces that provided the first quantitative link between distributions of wavelengths and perceived colour. These are now known as CIE 1931 RGB and CIE 1931 XYZ.

CIE RGB was derived from a series of independent colour matching experiments. Participants were presented with a test colour and an adjustable colour—a mixture of three monochromatic (single wavelength) primary colours. Participants then had to match the adjustable colour to the test one by altering the primary colours' intensities. In cases where a match was not possible, a variable amount of one of the primary colours was added to the test colour; this corresponded to a negative value for the primary colour. The results of these experiments were used to define three colour matching functions for primary wavelengths of 435.8 nm, 546.1 nm, and 700 nm, shown in Figure 2.4.

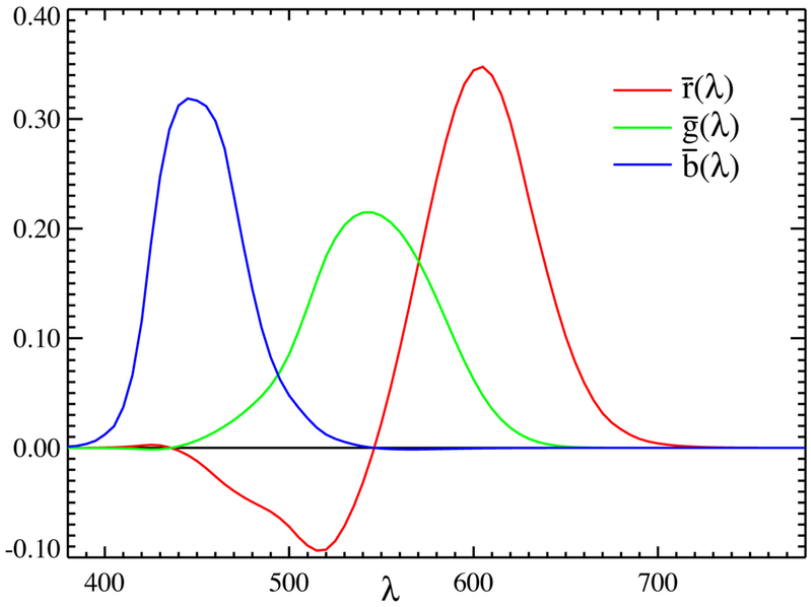


Figure 2.4: CIE RGB colour matching functions. Source: https://commons.wikimedia.org/wiki/File:CIE1931_RGBCMF.png

While CIE RGB has a gamut that covers all colours perceptible by humans, it requires negative values to achieve some colours. To address this, CIE created the XYZ space, which is a linear transformation of CIE RGB designed to ensure the colour matching functions were non-negative while also satisfying other technical constraints. The details of this transformation are beyond the scope of these notes; Figure 2.5 shows the resulting functions.

While a full plot of all colours is three dimensional—due to three types of cone cells, it can be visualized in 2D by dividing colour into brightness and chromaticity. CIE XYZ is already defined such that Y is a measure of luminance (brightness). The chromaticity, then, can be expressed with the derived parameters

$$x = \frac{X}{X+Y+Z} \quad \text{and} \quad y = \frac{Y}{X+Y+Z}.$$

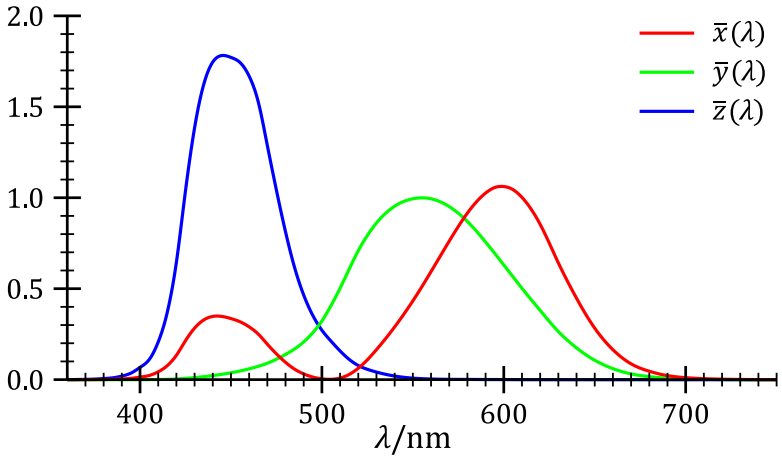


Figure 2.5: CIE XYZ colour matching functions. Source: https://commons.wikimedia.org/wiki/File:CIE_1931_XYZ_Color_Matching_Functions.svg

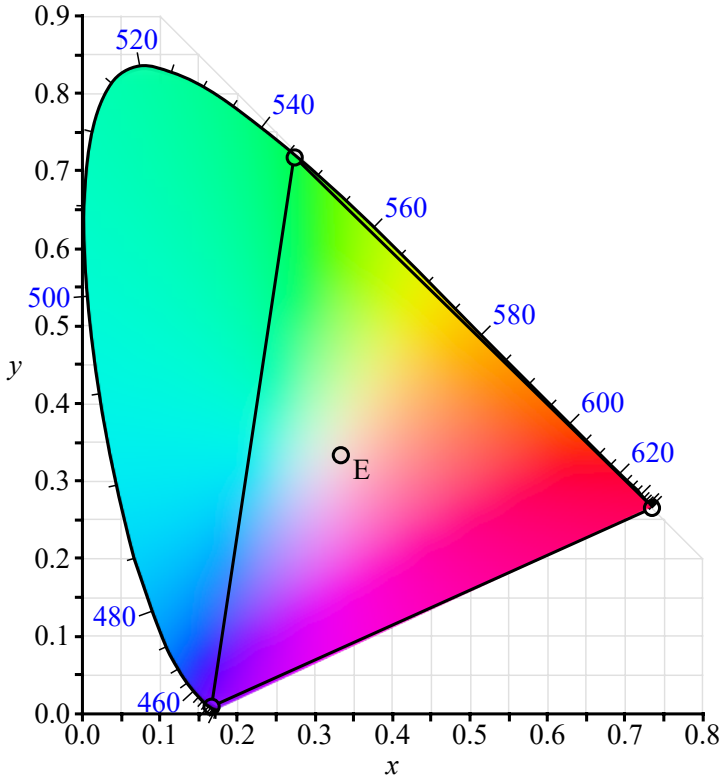


Figure 2.6: CIE 1931 chromaticity diagram. The vertices of the triangle show the primary colours for CIE RGB. Note that this image is specified in sRGB, and as such, does not actually represent all these colours properly (nor would your monitor be able to display them). Source: https://commons.wikimedia.org/wiki/File:CIE1931xy_CIERGB.svg

Plotting x and y, we get a visualization of all of the human-perceptible colours, controlled against varying brightness (Figure 2.6).

2.4.3 RGB Additive Model

While CIE XYZ is useful for objectively describing colours, it does not provide a convenient way to produce said colours artificially. While theoretically, one could manufacture a display device capable of producing any combination of wavelengths of light, such devices are not currently feasible. Instead, most display devices work by combing different combinations of a few (usually 2–4) primary colours to create their gamut. This leads us to the most common colour model used in computer graphics, the RGB additive.

Additive colour models start from black (little to no light) and create colour by *adding* different wavelengths of light that correspond to the primary colours of the model. As the name suggests, RGB additive models use red, green, and blue as their primary colours. While other additive models are possible, RGB closely matches the response of L, M, and S cone cells, resulting in gamuts that reasonably cover the full gamut of human colour perception. Defining the exact colours for the primary RGB colours defines an RGB colour *space*, one example being CIE RGB. As mentioned earlier, the primary colours in CIE RGB are monochromatic shades of red, green, and blue. However, manufacturing devices that display these colours is not necessarily practical. Because of this, other RGB colour spaces have been developed to match the gamut of display devices better, as well as for other purposes.

sRGB Created by HP and Microsoft in 1996, the sRGB colour space was designed for use with colour monitors, printers, and the web. Primary colours were chosen that closely matched the colour of CRT phosphors typical at the time. Figure 2.7 shows the gamut of the sRGB colour space, which covers 35.9% of visible colours.

In addition to specifying primary colours, sRGB also defines a special "gamma" transfer function. Like the primary colours, this function was defined to closely match the voltage response of CRTs of the time. This function is similar to normal gamma correction; however, there is a linear portion near zero. This piecewise definition is used to prevent an infinite slope at zero, which aids with certain numerical calculations. Overall, it is similar to an encoding gamma of 2.2, but is sufficiently different and should not be replaced by such an approximation.

sRGB is, by far, the most common colour space in use today. Most image and video files use this space. Furthermore, most consumer-grade cameras and scanners use sRGB by default or as the only available space. If no colour space information is provided with an image or colour, it is often safe to assume it is in sRGB.

Other RGB Spaces While sRGB is the most common, other RGB spaces are also used in certain applications. Adobe RGB has a larger gamut than sRGB, designed to cover most of the CMYK model (described below) used in printing. Wide-gamut RGB provides an even larger gamut, covering 77.6% percent of visible colours compared to 52.1% of Adobe RGB.

Transformations of RGB While RGB models are commonly used, they are not an intuitive way to think about colour. HSL (hue, saturation, lightness) and HSV (hue, saturation, value) are

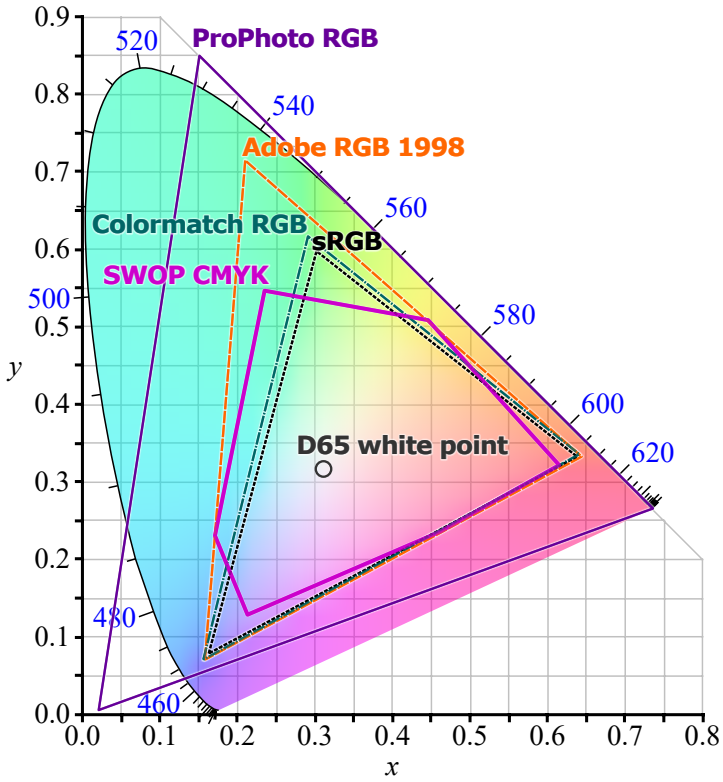


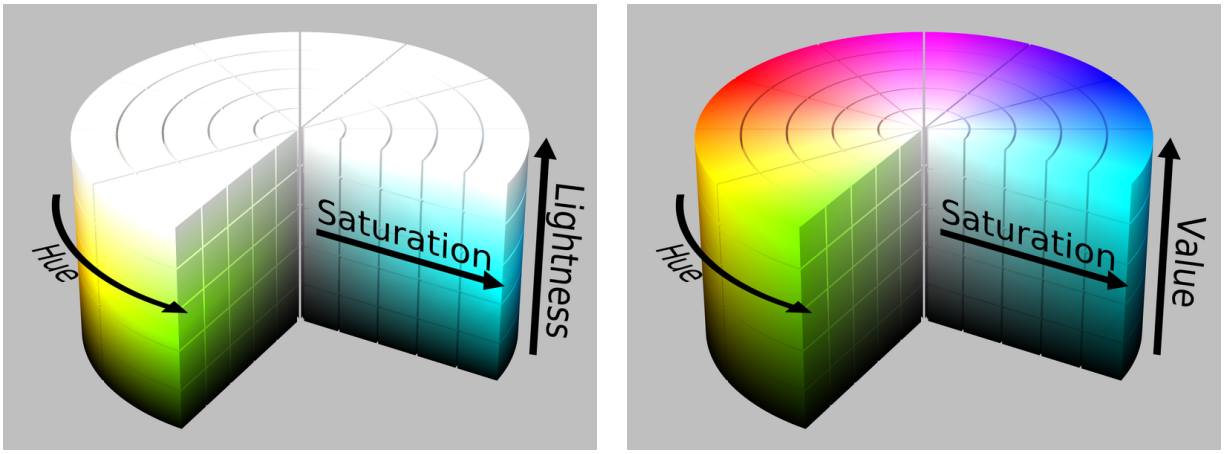
Figure 2.7: A comparison of several colour spaces with their gamuts overlaid on the CIE 1931 chromaticity diagram. Source: https://commons.wikimedia.org/wiki/File:CIE1931xy_gamut_comparison.svg

transformations of the RGB model designed to better align with how humans perceive attributes of colour (Figure 2.8a and 2.8b). HSV models colour in a way similar to mixing paints, whereas HSL has more resemblance to perceptual colour models. Most colour picking interfaces in software are based on the HSL or HSV model—or provide such an option alongside others.

2.4.4 Subtractive Models

As described earlier, additive models describe colour by combinations of light at different wavelengths. This type of model is useful for electronic displays that operate by emitting light; however, most objects we see in day to day life are not emissive. Instead, their colour is defined by the spectrum of light they reflect to our eyes. Specifically, illuminated by "white" light (relatively equal amounts of all wavelengths), an object absorbs some wavelengths and reflects the rest, which defines its colour. This process forms the basis of subtractive colour models, which operate by *subtracting* different wavelengths of light that correspond to the primary colours of the model.

RYB The primary colours most people are familiar with are red, blue, and yellow, which together form the RYB subtractive model. This model has been used for hundreds of years by artists to create a broad range of colours from a limited set of pigments; black and white are often used to supplement these primary colours further. Interestingly, mixing these three colours does not



(a) (b)

Figure 2.8: (a) HSL transformation of RGB (b) HSV transformation of RGB. Sources: https://commons.wikimedia.org/wiki/File:HSL_color_solid_cylinder_saturation_gray.png and https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder_saturation_gray.png

produce black, but instead a deep brown.

CMY and CMYK Another common subtractive colour model uses cyan, magenta, and yellow (CMY) as its primary colours. In some sense, this model is the complement of the RGB additive model, as primary colours in one space are secondary colours (the combination of two primary colours) in the other. CMY, with the addition of black (CMYK), is widely used throughout the printing industry, including in consumer-grade printers.

3 Basic Geometry

Ask a child to draw a house and you'll probably get something like Figure 3.1; a square, triangle, and rectangle together form the image of a house. Geometric objects are the basic building blocks used to create images. Even complex drawings can be broken down into a series of strokes: lines and curves with varying intensity.

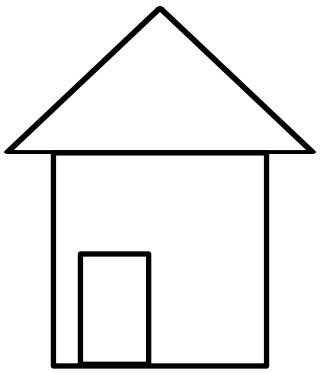


Figure 3.1: Simple image of a house created from three basic shapes: a square, triangle, and rectangle. These shapes can be further broken down into a series of line segments.

As computer graphics is concerned with using *computers* to generate images, it is essential to understand how to represent geometric objects mathematically—and as an extension—on a computer. Algorithms then turn these objects and other data into images. This section serves as a high-level introduction to a few basic types of geometry used in computer graphics. Subsequent sections will take a more in-depth look at some of these ideas.

3.1 Polygons and Polyhedra

Polygons are some of the most basic shapes we are introduced to in grade school and include familiar shapes such as the square, triangle, and rectangle. Formally, polygons are formed by connecting a series of points (vertices) via straight line segments (edges) to create a closed shape. While polygons can intersect themselves, typically, we are most interested in non-intersecting polygons (called simple polygons).

Because of their simplicity, specifying the shape of a polygon is straightforward. Given a cartesian plane, we specify vertices as (x, y) points on the plane; edges are simply pairs of vertices. Therefore, a list of vertices and edges defines a polygon. Alternatively, vertices specified in a specific order also define a polygon, with edges implicitly defined between each point.

Going to the third dimension, we extend polygons to get polyhedra, which under most definitions are a set of flat, polygonal faces that connect to form a closed object. This class of shapes includes the cube, pyramids, and prisms. In graphics, we use term polygonal mesh to refer to a collection of vertices, edges, and faces, which describe a polyhedral object (or a more broad class of shapes, depending on the definition of polyhedron used). Like polygons, polyhedra can be specified simply as a list of vertices and faces that connect vertices. Other representations are also possible, as will be explored further in Section 8.

3.2 Curves and Surfaces

While polygons and polyhedra are simple, they are jagged and do not describe "smooth" shapes well. A shape as simple as a circle cannot be perfectly represented by a polygon (unless an infinite number of edges are allowed). Instead, we need different representations for smooth objects, i.e. curves and surfaces.

Sticking with the circle, a common definition is $x^2 + y^2 = 1$, where a point (x, y) lies on the unit circle iff this relationship holds. Such a representation is an effective way to represent a circle, but it is not the only way. This definition provided a simple way to check if a given point is on a circle; however, it does not give any simple method for obtaining the set of *all* points on the circle. Because of this, we call this an implicit definition.

An alternative method for describing a circle comes from basic trigonometry, where the functions $\cos\theta$ and $\sin\theta$ specify the x and y coordinates of a point at angle θ on the unit circle. This definition converts values of θ into points on a circle; hence, we call this a parametric representation. Parametric curves and surfaces are explored more thoroughly in Section 7.

3.3 Generative Geometry

One of the main challenges with representing complex geometries is the amount of data required to store them. As opposed to fully representing objects, an alternative approach is to dynamically generate geometry algorithmically. This approach allows for detailed, high-resolution, objects to be created from a small memory footprint. Many advanced computer graphics techniques fall into this broad category, including subdivision surfaces, L-systems, and constructive solid geometry. However, one of the most straightforward demonstrations of this method is with fractals.

3.3.1 Fractals

The world around us is not made of perfect geometric objects. Clouds are not spheres. Mountains are not cones. A tree is not a cylinder. Yet Euclidean geometry describes only ideal shapes: spheres, circles, cubes, smoothly varying functions. **Fractal geometry** is the geometry of the natural world, capable of describing complex and non-ideal objects.

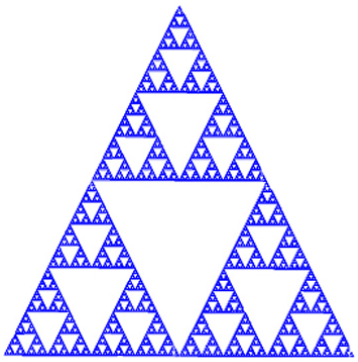


Figure 3.2: A fractal shape known as the Sierpinski Triangle.

The concept of Euclidean dimension is a familiar one. We know that lines and curves have a dimension of one, planes and surfaces have a dimension of two, volumes have a dimension of three, and so on. But can dimension only take on integer values?

Consider a piece of aluminum foil laid flat in a plane. The dimension of the foil is two. What happens as we gradually deform the foil, crumpling it into a dense ball? At the end of the deformation, the foil ball has a dimension of three. Is there some point where the dimension suddenly jumps from two to three? Or is the change in dimension a continuous process? Euclidean geometry says that the dimension is discontinuous and jumps from two to three, but can we define a new dimension concept that is continuous?

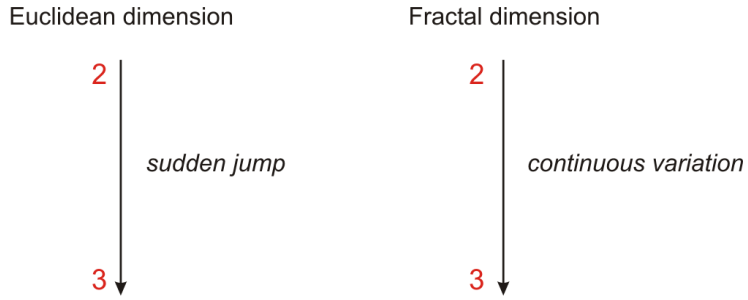


Figure 3.3: Euclidean versus fractal dimension.

In nature, and in complicated mathematical objects, there is often **self-similarity**, or the existence of small sub-objects that resemble the larger object. **Fractal dimension** is a measure of dimension that considers the self-similarity property, but is consistent with Euclidean dimension for simple objects. Figure 3.3 illustrates the difference between Euclidean and fractal dimension.

Fractal dimension D_F is defined as:

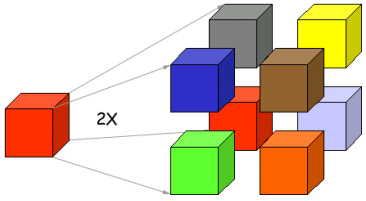
$$D_F = \log_e N \tag{2}$$

where

$$\begin{aligned} e &= magnification\ factor\ , \\ 1/e &= contraction\ factor\ , \\ N &= number\ of\ self\ similar\ objects\ . \end{aligned}$$

Example 3.1

What is the fractal dimension of a cube?



Answer:

A cube can be divided into 8 smaller cubes, each of which is half the size of the original cube. Thus,

$$\begin{aligned} N &= 8 \\ 1/e &= 0.5 \\ e &= 2 \end{aligned}$$

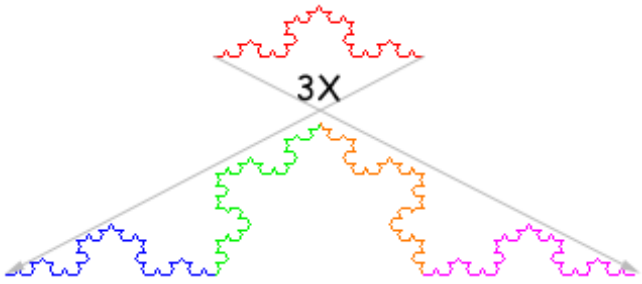
and therefore

$$\begin{aligned} D_F &= \log_2 8 \\ &= 3 \end{aligned}$$

So, for a simple object like a cube, the fractal dimension is the same as the Euclidean dimension. What about the dimension of complex objects such as the Sierpinski triangle in Figure 3.2?

Example 3.2

What is the fractal dimension of the Koch snowflake?



Answer:

As seen in the diagram, we can find four smaller self-similar copies of the snowflake, each of which is one-third the size of the full snowflake. Thus,

$$\begin{aligned} N &= 4 \\ 1/e &= 1/3 \\ e &= 3 \end{aligned}$$

and therefore

$$\begin{aligned} D_F &= \log_3 4 \\ &\approx 1.26 \end{aligned}$$

The Euclidean dimension of the snowflake is $D_E = 1$.

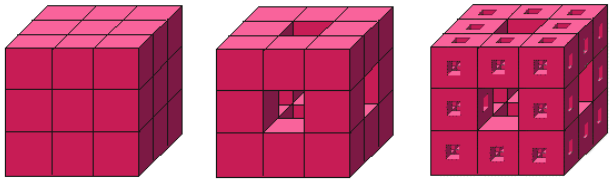
A **fractal** is an object that has a fractal dimension greater than its Euclidean dimension:

$$D_F > D_E$$

For example, the Koch snowflake has a fractal dimension of $D_F = 1.26$ and a Euclidean dimension of only $D_E = 1$. The Sierpinski Triangle is another example of a fractal.

Example 3.3

Is the Menger sponge (below) a fractal?



Answer:

We can find 20 self-similar copies in the third image of the Menger sponge, each of which is one-third the size of the original. Thus,

$$\begin{aligned} N &= 20 \\ 1/e &= 1/3 \\ e &= 3 \end{aligned}$$

and therefore

$$\begin{aligned} D_F &= \log_3 20 \\ &\approx 2.73 \end{aligned}$$

The fractal dimension $D_F = 2.73$ is greater than the Euclidean dimension $D_E = 2$. Therefore, the Menger sponge is a fractal.

The key property of fractals is their self-similar nature. In other words, a fractal can be broken down into smaller objects which are themselves fractals. Recursive algorithms solve a problem by breaking it into smaller instances of the same problem. Thus, recursive algorithms are ideal for programming fractals.

Algorithm 3.1 presents an algorithm for generating the Sierpinski Triangle (Figure 3.2).

Algorithm 3.1 Recursive algorithm for the Sierpinski Triangle fractal.

```
void divide_triangle(point a, point b, point c, int m)
{
    point v0, v1, v2;
    int j;

    if (m > 0) {
        for (j = 0; j < 2; j++)
            v0[j] = (a[j]+b[j])*0.5;
        for (j = 0; j < 2; j++)
            v1[j] = (a[j]+c[j])*0.5;
        for (j = 0; j < 2; j++)
            v2[j] = (b[j]+c[j])*0.5;

        divide_triangle(a, v0, v1, m-1);
        divide_triangle(c, v1, v2, m-1);
        divide_triangle(b, v2, v0, m-1);
    }
    else
        draw_triangle(a, b, c);
}
```

4 Representing Geometric Objects

Geometric objects such as points, lines, and curves are fundamental to computer graphics. Vector spaces and affine spaces provide the background for understanding these objects.

4.1 Vector Space

A **vector space** V is a set of scalars (real numbers) and vectors. Vectors have two possible operations: addition of two vectors, and multiplication of a vector and a scalar. Let u, v, and w be vectors in some vector space V, and let α and β be scalars.

4.1.1 Basic Operations

Vector addition is *closed*, so that $u+v \in V$. Vector addition is also *commutative* ($u+v = v+u$) and *associative* ($(u+v)+w = u+(v+w)$). There is a special *zero vector* $\mathbf{0}$ such that $u+\mathbf{0} = \mathbf{0}+u = u$. Finally, for every vector u, there is an *additive inverse* $-u$ such that $u + (-u) = \mathbf{0}$. Vector subtraction is just the addition of an additive inverse: $u - v = u + (-v)$.

Scalar-vector multiplication is *closed*, meaning that $\alpha u \in V$ if $u \in V$. It is also *commutative*: $\alpha(u + v) = \alpha u + \alpha v$ and $(\alpha + \beta)u = \alpha u + \beta u$.

An example of a vector space is $\Re^n$. A vector $v \in \Re^n$ is an n-tuple of scalars:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} .$$

In this vector space, vector-vector addition of v, w is defined as

$$v + w = \begin{bmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

and scalar-vector multiplication of vector v and scalar α is

$$\alpha v = \begin{bmatrix} \alpha v_1 \\ \vdots \\ \alpha v_n \end{bmatrix} .$$

Vectors in $\Re^n$ can be manipulated with matrix algebra, as we will see later.

4.1.2 Dimension and Bases

A **linear combination** of vectors is the vector sum of scalar-vector products:

$$\alpha_1 u_1 + \alpha_2 u_2 + \ldots + \alpha_n u_n .$$

Example 4.1

Write the vector $[3\ 1\ 1]^T$ as a linear combination of $[1\ 1\ 1]^T$ and $[0\ 1\ 1]^T$.

Answer:

Let $\alpha_1 = 3$ and $\alpha_2 = -2$. Then

$$\alpha_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + (-2) \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}$$

A set of vectors is *linearly independent* if

$$\alpha_1 u_1 + \alpha_2 u_2 + \ldots + \alpha_n u_n = 0$$

implies

$$\alpha_1 = \alpha_2 = \ldots = \alpha_n = 0\ .$$

In other words, a set of vectors S is linearly independent if we cannot express $v \in S$ as a linear combination of the other vectors in S.

Example 4.2

Consider the sets S_1 and S_2:

$$S_1 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$$

$$S_2 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

S_1 is linearly independent. S_2 is not linearly independent.

The **dimension** of a vector space is the maximum number of linearly independent vectors in the space. For example, the dimension of $\Re^2$ is $dim = 2$; there is no set of linearly independent vectors of size three in the space. For a vector space of dimension n, any set of linearly independent vectors forms a basis of the vector space.

Example 4.3

For the vector space $\Re^n$, there is a *standard basis* defined as

$$S_1 = \left\{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right\}$$

For example, the standard basis of $\Re^3$ is

$$S_1 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

A **basis** $\{u_1, \ldots, u_n\}$ of a vector space V is said to *span* V because any vector $v \in V$ can be written as a linear combination of the basis vectors:

$$v = \beta_1 u_1 + \ldots + \beta_n u_n \ .$$

The scalars β_i are called the *representation* of v with respect to the basis.

Example 4.4

What is the representation of $[5\ 6\ 2]^T$ with respect to the basis $\{[1\ 0\ 0],\ [1\ 1\ 0],\ [1\ 1\ 1]\}$.

Answer:

By inspection we can see that

$$\begin{bmatrix} 5 \\ 6 \\ 2 \end{bmatrix} = (-1) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 4 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \ .$$

Therefore, the representation is

$$\begin{bmatrix} -1 \\ 4 \\ 2 \end{bmatrix} \ .$$

4.1.3 Magnitude

The **magnitude** $|v|$ of a vector $v \in \Re^n$ is defined as

$$|v| = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2} \, .$$

A vector with a magnitude of 1 is called a **unit vector**. Any vector can be made into a unit vector by *normalizing* it; a **normalized vector** $\widehat{v}$ is defined as

$$\widehat{v} = \frac{v}{|v|} \, .$$

Sometimes a unit vector is referred to as a *direction*, because normalization essentially removes the magnitude of the vector.

Example 4.5

Compute the magnitude and normalized representation of $u = [3 \; 1 \; 4]^T$.

Answer:

First compute the magnitude:

$$|u| = \sqrt{3^2 + 1^2 + 4^2} = \sqrt{26} \, .$$

Then compute the normalized vector:

$$\begin{aligned} \widehat{u} &= \frac{1}{\sqrt{26}} \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} \frac{3}{\sqrt{26}} \\ \frac{1}{\sqrt{26}} \\ \frac{4}{\sqrt{26}} \end{bmatrix} \end{aligned}$$

4.1.4 More Operations

The **dot product**, or inner product, operator $\cdot$ is a powerful tool in computer graphics. It is defined between two vectors and produces a scalar result. For vectors $u, v \in \Re^n$, the dot product is:

$$u \cdot v = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

The dot product is commutative and associative, meaning that the following properties hold:

$$\begin{aligned} u \cdot v &= v \cdot u \\ (u + v) \cdot w &= u \cdot w + v \cdot w \\ \alpha(u \cdot v) &= (\alpha u) \cdot v \end{aligned}$$

The dot product has a useful geometric interpretation in $\Re^n$: it computes a measure of the angle between two vectors (Figure 4.1):

$$u \cdot v = |u||v| \cos\theta \; . \tag{3}$$

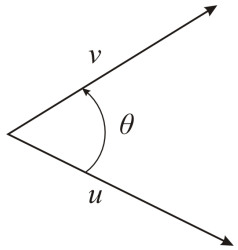


Figure 4.1: Geometric interpretation of the dot product.

From the sign of the dot product, we can determine a lot about the relationship between the vectors without explicitly computing the angle between them.

$$\begin{aligned} u \cdot v > 0 &\implies \theta < 90^\circ \\ u \cdot v < 0 &\implies \theta > 90^\circ \\ u \cdot v = 0 &\implies \theta = 0^\circ \end{aligned}$$

This provides an easy way to test if two vectors are perpendicular: the dot product will be 0 if they are perpendicular.

Example 4.6

Compute the angle between $[2\ 3\ 5]^T$ and $[-5\ 1\ \ -4]^T$.

Answer:

We can rewrite Equation 3 as

$$\theta = \cos^{-1}\left(\frac{u \cdot v}{|u||v|}\right) \; .$$

Therefore

$$\begin{aligned} \theta &= \cos^{-1}\left(\frac{[2\ 3\ 5]^T \cdot [-5\ 1\ \ -4]^T}{\sqrt{2^2+3^2+5^2}\sqrt{(-5)^2+1^2+(-4)^2}}\right) \\ &= \cos^{-1}\left(\frac{2*(-5)+3*1+5*(-4)}{\sqrt{38}\sqrt{42}}\right) \\ &= \cos^{-1}\left(\frac{-27}{39.95}\right) \\ &= \cos^{-1}(-0.676) \\ &= 132.5^\circ \end{aligned}$$

Note that the dot product of the two vectors is negative, so our result of $132.5° > 90°$ is expected.

The **cross product** $\times$ of two vectors $u, v \in \Re^3$ produces a vector result. It is defined as:

$$u \times v = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix}.$$

The cross product can also be computed as the determinant of a 3×3 matrix:

$$u \times v = \begin{vmatrix} i & j & k \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix},$$

where i, j, and k are the standard basis vectors of $\Re^3$. Note that the cross product is defined only in $\Re^3$.

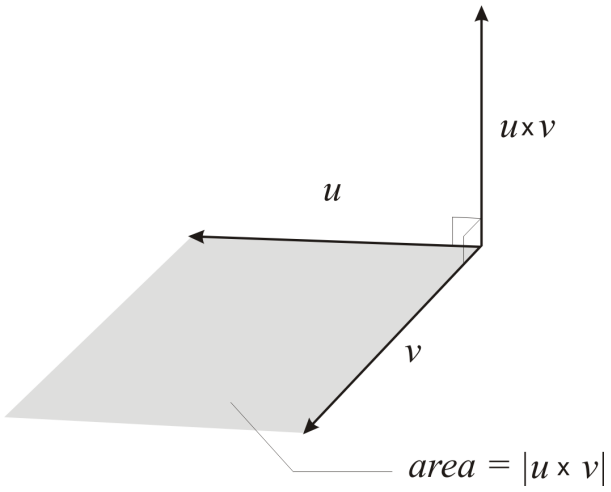


Figure 4.2: Geometric interpretation of the cross product.

Example 4.7

Compute the cross product of $[2\ 3\ 5]^T$ and $[-5\ 1\ -4]^T$.

Answer:

$$\begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \times \begin{bmatrix} -5 \\ 1 \\ -4 \end{bmatrix} = \begin{bmatrix} 3*(-4) - 5*1 \\ 5*(-5) - 2*(-4) \\ 2*1 - 3*(-5) \end{bmatrix}$$
$$= \begin{bmatrix} -17 \\ -17 \\ 17 \end{bmatrix}$$

The cross product is extremely useful in computer graphics because of its geometric interpretation (Figure 4.2).

- $u \times v$ is perpendicular to both u and v. In other words, $u \times v$ defines the normal direction of the plane containing u and v.
- The magnitude of $u \times v$ is equal to the area of the parallelogram defined by u and v.

4.2 Affine Space

Vector spaces provide rich mathematics for describing geometric objects, but they lack a crucial aspect: location. Vectors have no location in space, so concepts such as the distance between two vectors are meaningless.

An **affine space** adds a third entity to vector spaces: points. A **point** is a location in space and, unlike vectors, has no associated direction or magnitude. For instance, a vertex in an OpenGL program is a point, not a vector. Thus, points are crucial for representing geometric objects, while vectors are used to manipulate them.

4.2.1 Point-Vector Operations

Let u, v, and w be vectors, and let P, Q, and R be points. What type of operations are defined between these entities?

Point-vector addition is one type of operation: $Q = P + v$. This operation has a clear geometric interpretation. The point P is displaced by the magnitude of v in the direction of v to create a new point Q; see Figure 4.3.

Point-point subtraction is also a valid operation. In fact, it is just a different way of writing point-vector addition:

$$Q = P + v \implies v = Q - P\,.$$

The difference between two points yields a vector that points from the second point (the *subtrahend*) to the first (the *minuend*). This operation is also illustrated by Figure 4.3.

Example 4.8

Which of the following combinations of points and vectors are valid? Why or why not?

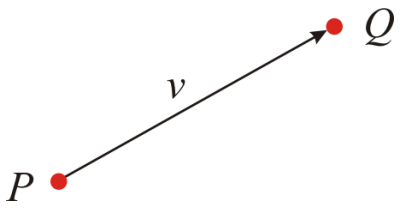


Figure 4.3: Point-vector operations. The vector v can be added to point P to produce Q. Alternatively, subtracting P from Q yields v.

(a) $P + 5v$

(b) $P + 2Q$

(c) $2P - Q + 3v$

Answer:

(a) is *valid*, because scalar-vector multiplication is closed. Therefore, $5v$ is a vector, and point-vector addition is defined.

(b) is *not valid*, because point-point addition is not an allowed operation.

(c) is *valid*. Rewrite the expression as

$$P + (P - (Q + 3v)) \, .$$

Then, $Q + 3v$ is another point R, $P - R$ yields a vector w, which is then added to P. All these operations are valid, so the original expression is valid.

4.2.2 Linear Combinations of Points

So, we have seen that we can subtract one point from another, but we cannot add two points together. In other words, we cannot take arbitrary linear combinations of points. However, there is a certain type of linear combination that is always defined for points.

Consider the linear combination $\frac{1}{2}P + \frac{1}{2}Q$. This is a valid operation on points, but why? We can rewrite this expression to clearly see why it is valid:

$$\begin{aligned} \frac{1}{2}P + \frac{1}{2}Q &= P + \frac{1}{2}Q - \frac{1}{2}P \\ &= P + \frac{1}{2}(Q - P) \\ &= P + \frac{1}{2}v \, , \end{aligned}$$

where $v = Q - P$ is a vector. In general, $\beta P + \alpha Q$ is a valid operation when $\alpha + \beta = 1$ because:

$$\begin{aligned} \beta P + \alpha Q &= P + (\beta - 1)P + \alpha Q \\ &= P + \alpha(Q - P)\,, \end{aligned} \tag{4}$$

where $\beta - 1 = -\alpha$.

This type of linear combination of points has a nice geometric interpretation. If we rewrite Equation 4 as a function $L(\alpha)$

$$L(\alpha) = (1 - \alpha)P + \alpha Q\,,$$

we get a parametric representation of a line (Figure 4.4). When $\alpha = 0$, $L(\alpha) = P$; when $\alpha = 1$, $L(\alpha) = Q$. For other values of α, we get some point on the line that passes between P and Q.

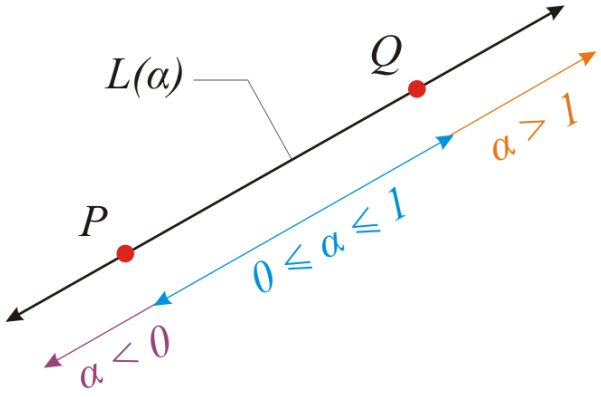


Figure 4.4: The parameterization of a line as a linear combination of two points.

We have seen that we add two points together in a certain type of linear combination where the coefficients of each point sum to unity. What about for more than two points?

Consider a linear combination of three points, with each coefficient $\frac{1}{3}$: $\frac{1}{3}P + \frac{1}{3}Q + \frac{1}{3}R$. We can manipulate this expression to get a familiar form:

$$\begin{aligned} \frac{1}{3}P + \frac{1}{3}Q + \frac{1}{3}R &= P - \frac{1}{3}P + \frac{1}{3}Q - \frac{1}{3}P + \frac{1}{3}R \\ &= P + \frac{1}{3}(Q - P) + \frac{1}{3}(R - P)\,. \end{aligned}$$

This last form is valid because $Q - P$ and $R - P$ are vectors and point-vector addition is a valid operation.

In general, a linear combination $\alpha_1 P + \alpha_2 Q + \alpha_3 R$ of three points is valid when $\alpha_1 + \alpha_2 + \alpha_3 = 1$. We can prove this as follows:

$$\begin{aligned} \alpha_1 P + \alpha_2 Q + \alpha_3 R &= (1 - \alpha_2 - \alpha_3)P + \alpha_2 Q + \alpha_3 R \\ &= P - \alpha_2 P + \alpha_2 Q - \alpha_3 P + \alpha_3 R \\ &= P + \alpha_2(Q - P) + \alpha_3(R - P) \end{aligned}$$

The coefficients α_1, α_2, and α_3 are known as **barycentric coordinates**, and they are used to parameterize the plane $G(\alpha_2, \alpha_3)$ that contains P, Q, and R; see Figure 4.5.

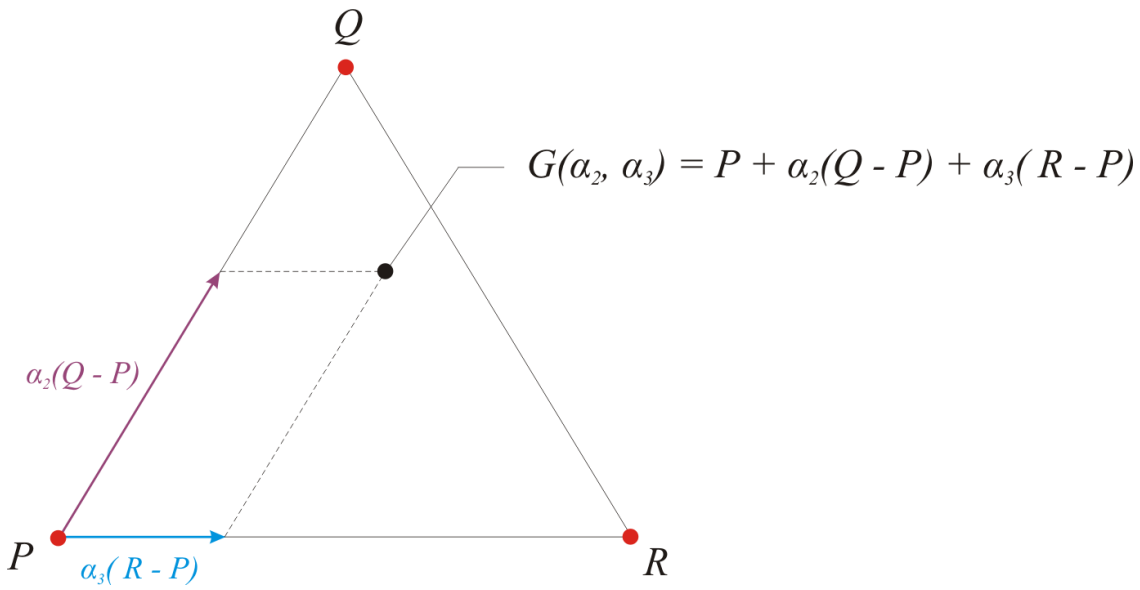


Figure 4.5: Barycentric coordinates parameterize a plane G containing three points.

In both cases – two and three points – linear combinations of the points are only defined when the sum of the coefficients is equal to one. This is known as an **affine combination**, and the condition holds in general.

For n points $P_1, \ldots, P_n$, the linear combination $\sum_{i=1}^{n} \alpha_i P_i$ is defined only when $\sum_{i=1}^{n} \alpha_i = 1$. The proof is an extension of the proof of the 2-point and 3-point cases:

$$\begin{aligned} \sum_{i=1}^{n} \alpha_i P_i &= \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \\ &= (1 - \alpha_2 - \ldots - \alpha_n) P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \\ &= P_1 + \alpha_2 (P_2 - P_1) + \ldots + \alpha_n (P_n - P_1) \\ &= P_1 + v \,, \end{aligned} \tag{5}$$

where $v = \alpha_2(P_2 - P_1) + \ldots + \alpha_n(P_n - P_1)$ is a vector.

Example 4.9

Which of the following expressions are valid affine combinations?

(a) $\frac{1}{4}P + \frac{1}{2}Q + \frac{1}{4}R$

(b) $\frac{1}{2}P + \frac{1}{8}Q$

(c) $3P - 2Q$

(d) $\frac{1}{6}P + \frac{3}{4}Q + \frac{1}{8}R$

Answer:

(a) is *valid*, because $\frac{1}{4} + \frac{1}{2} + \frac{1}{4} = 1$.

(b) is *invalid*, because $\frac{1}{2} + \frac{1}{8} = \frac{5}{8} \neq 1$.

(c) is *valid*, because $3 + (-2) = 1$.

(d) is *invalid*, because $\frac{1}{6} + \frac{3}{4} + \frac{1}{8} = \frac{25}{24} \neq 1$.

Are there other valid ways of combining points? We have seen that when $\sum \alpha_i = 1$, the expression simplifies to point-vector addition (Equation 5). Point-point subtraction is another valid operation between points and vectors:

$$1P + (-1)Q = v \; .$$

Here, $\sum \alpha_i = 1 + (-1) = 0$, and the result is a vector. Does this hold in general?

Let $\sum \alpha_i = 0$, i.e. $\alpha_1 = 0 - \alpha_2 - \ldots - \alpha_n$. Then

$$\begin{aligned} \sum_{i=1}^{n} \alpha_i P_i &= \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \\ &= (0 - \alpha_2 - \ldots - \alpha_n) P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \\ &= \alpha_2 (P_2 - P_1) + \ldots + \alpha_n (P_n - P_1) \\ &= v \; , \end{aligned} \tag{6}$$

where $v = \alpha_2 (P_2 - P_1) + \ldots + \alpha_n (P_n - P_1)$ is a vector. So, $\sum \alpha_i = 0$ produces a valid affine combination of points, and the result is a vector.

To summarize:

$$\begin{aligned} \sum \alpha_i = 1 \quad &\Longrightarrow \quad \sum \alpha_i P_i = a\ point \\ \sum \alpha_i = 0 \quad &\Longrightarrow \quad \sum \alpha_i P_i = a\ vector \\ \sum \alpha_i \notin \{0, 1\} \quad &\Longrightarrow \quad \sum \alpha_i P_i = undefined \end{aligned}$$

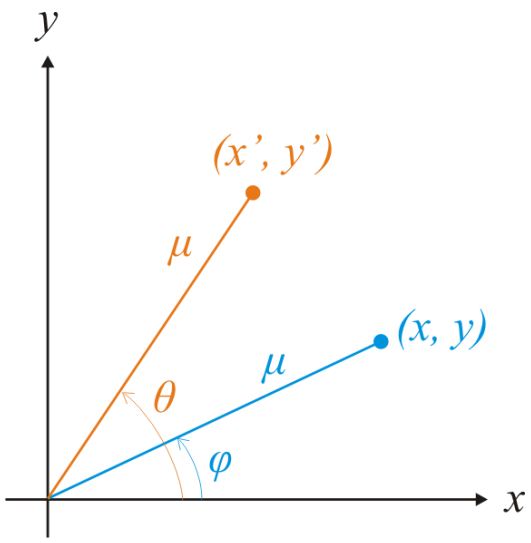


Figure 5.1: Two-dimensional rotation: an initial point (x, y) is rotated by θ to (x', y').

5 Affine Transformations

In Section 4, we were introduced to the concept of affine spaces. An affine space contains points, vectors, and scalars. These three entities together allow us to *represent* geometric objects, such as lines, curves, and polygons.

Beyond representation, in computer graphics, we also need tools to *manipulate* objects. Consider the animation of a character in a video game: the object as a whole moves throughout the game world, while the limbs may move independently of the body and the head looks around.

Affine transformations are used to manipulate objects in an affine space. They include such transformations as scaling, rotation, and translation. Transformations allow us to resize, reorient, and reposition objects in a world. They are necessary to compose a scene or world out of many different objects whose local coordinate systems may coincide. Transformations can also be composed in a hierarchical fashion to produce complex objects and behaviour.

Before we delve into affine transformations in general, let's first examine a simple example of 2D rotations.

5.1 2D Rotation

Consider a point $P = [x\ y]^T \in \Re^2$. We want to rotate P about the origin by angle θ to the new position $P' = [x'\ y']$. How can we compute P' from P and θ?

Figure 5.1 illustrates the situation. The angle between the x axis and the vector v from $(0, 0, 0)$ to P is ϕ. After rotation, the angle increases to $\phi + \theta$. Let the $\mu = \sqrt{x^2 + y^2}$ be the magnitude of v; this magnitude is unchanged by rotation, so the distance from the origin to P' is also μ. Then we can write

$$\begin{aligned} x &= \mu \cos \phi \\ y &= \mu \sin \phi\ , \end{aligned}$$

and

$$x' = \mu\cos(\phi+\theta) \tag{7}$$
$$y' = \mu\sin(\phi+\theta)\ . \tag{8}$$

Recall the trigonometric identities regarding the cosine and sine of a sum of angles:

$$\cos(\phi+\theta) = \cos\phi\cos\theta - \sin\phi\sin\theta \tag{9}$$
$$\sin(\phi+\theta) = \sin\phi\cos\theta + \cos\phi\sin\theta\ . \tag{10}$$

Substituting 9 into 7 and 10 into 8, we find

$$\begin{aligned} x' &= (\mu\cos\phi)\cos\theta - (\mu\sin\phi)\sin\theta \\ &= x\cos\theta - y\sin\theta \qquad (11) \\ y' &= (\mu\sin\phi)\cos\theta + (\mu\cos\phi)\sin\theta \\ &= x\sin\theta + y\cos\theta\ . \qquad (12) \end{aligned}$$

We can write this transformation as a matrix $\mathbf{R}(\theta)$ that transforms P to P':

$$\begin{aligned} P' &= \mathbf{R}(\theta)P \\ \begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}\ . \end{aligned}$$

The rotation transformation also affects vectors:

$$\begin{bmatrix} v_0' \\ v_1' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\begin{bmatrix} v_0 \\ v_1 \end{bmatrix}\ .$$

In computer graphics, shapes are typically represented as a collection of points (vertices) connected by line segments. For instance, take a rectangle, which consists of four points connected by four line segments. When it comes to rotating an object (such as a rectangle), the question arises: Do we only need to rotate the vertices of the shape, or do we also have to rotate all the points on the line segments?

The beauty of rotation lies in its ability to simply rotate the vertices and then connect the resulting vertices with new line segments (see Figure 5.2). This property is specific to rotation and a particular class of other transformations and does not apply to general mappings in 2D space. The key reason behind this lies in the fact that rotation preserves lines. In contrast, imagine a 2D transformation that distorts lines into some form of curve. 2D rotation serves as an example of affine transformations in 2D affine space. It not only preserves lines and maps points to points and vectors to vectors but also preserves the line parameter. This stronger form of line preservation is a result of preserving the affine combination.

In summary, A is an *affine transformation* in affine space if it maps point to points, vectors to vectors and preserves affine combinations:

$$A\left(\sum_{i=1}^{n}\alpha_i P_i\right) = \sum_{i=1}^{n}\alpha_i A(P_i)$$

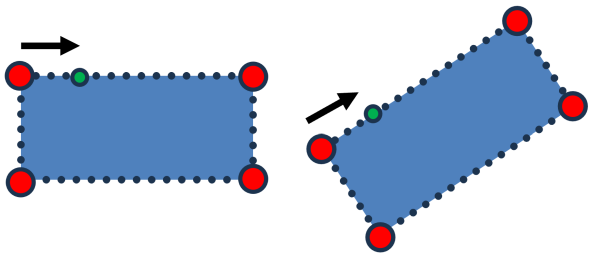


Figure 5.2: Rotation is a line-preserving transformation. Additionally, it preserves the line parameter (i.e. affine combinations' weights).

Note that a 2D rotation has a two-by-two matrix representation, and therefore, it is a linear transformation. By definition, linear transformations are affine transformations. However, not all affine transformations are linear. For example, a 2D translation is an affine transformation (you can verify this by checking the required properties of affine transformations), but it is not a linear transformation since it does not have a two-by-two matrix representation. Any two-by-two matrix maps zero to zero, but a 2D translation by vector v maps $(0, 0)$ to $(0, 0) + v$.

Itâs important to recognize that matrix representation is key to having a systematic way to combine transformations, but translation does not have a matrix form. This issue, combined with the challenge of distinguishing points and vectors in affine space, necessitates an elegant and unified way to represent points, vectors, and affine transformations, including translation.

5.2 Homogeneous Coordinates

We usually represent a point located at (x, y, z) as $P = [x\ y\ z]^T$. A vector is represented in the same way: $v = [v_1\ v_2\ v_3]^T$. This similarity in representation can cause a lot of confusion. How can one differentiate between the point $(1, 2, 3)$ and the vector that points from $(0, 0, 0)$ to $(1, 2, 3)$, where both the point and the vector would be represented by $[1\ 2\ 3]^T$?

Homogeneous coordinates distinguish between points and vectors by adding a third (in 2D) or fourth (in 3D) coordinate h to the representation. For points $h = 1$, and for vectors $h = 0$. Then,

the representation of point P and vector v are changed as follows:

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \implies \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \ (2D)$$

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \implies \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \ (3D)$$

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \implies \begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix} \ (2D)$$

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \implies \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix} \ (3D)$$

Note that OpenGL supports homogeneous coordinates (`glVertex4f`, for example).

In addition to allowing us to distinguish between point (x, y, z) and vector $(0, 0, 0) \rightarrow (x, y, z)$, homogeneous coordinates are a powerful representation for determining the validity and type of result of an operation. For example, homogeneous coordinates immediately tell us that point-point subtraction produces a vector:

$$P - Q = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ \mathbf{1} \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ \mathbf{1} \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \\ \mathbf{0} \end{bmatrix} \Bigg\} \mathbf{0} \ indicates \ result \ is \ a \ vector.$$

What about point-vector addition?

$$P + v = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ \mathbf{1} \end{bmatrix} - \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} x_1 - v_1 \\ y_1 - v_2 \\ z_1 - v_3 \\ \mathbf{1} \end{bmatrix} \Bigg\} \mathbf{1} \ indicates \ result \ is \ a \ point.$$

Clearly, homogeneous coordinates fit naturally into our previous concepts of vector and affine spaces and the permitted operations in them.

Example 5.1

Show that $\sum \alpha_i P_i$ produces a point when $\sum \alpha_i = 1$.

Answer:

$$\begin{aligned}
\sum \alpha_i P_i &= \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \\
&= \alpha_1 \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} + \ldots + \alpha_n \begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \sum x_i \\ \sum y_i \\ \sum z_i \\ \sum \alpha_i \end{bmatrix} \\
&= \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}
\end{aligned}$$

Having $\sum \alpha_i = 1$ ensures that the homogeneous coordinate is 1 in the result, indicating that the result is a point.

Show that *any* linear combination of vectors is valid.

Answer:

$$\begin{aligned}
\sum \alpha_i v_i &= \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \\
&= \alpha_1 \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 0 \end{bmatrix} + \ldots + \alpha_n \begin{bmatrix} x_n \\ y_n \\ z_n \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} \sum x_i \\ \sum y_i \\ \sum z_i \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix}
\end{aligned}$$

Regardless of the scalars α_i, the homogeneous coordinate in the result will always be 0, indicating that the result is a vector.

We use matrix notation to represent transformations. The key property of an affine transformation is that points are transformed to points, and vectors are transformed to vectors. With

homogeneous coordinates, this means that the last row of the transformation matrix must be the identity row, because this will ensure that the homogeneous coordinate h does not change.

Thus, a general 2D affine transformation matrix $\mathbf{A}$ has the form:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}.$$

The six entries a_{ij} can take on any value and the resulting matrix will still be affine. $\mathbf{A}$ is said to have six **degrees of freedom**.

In 3D, the matrix looks similar:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This matrix has 12 degrees of freedom.

5.3 Translation

Recall from Section 4.2 that point-vector addition $P + v$ moves P in the direction of v by the magnitude of v, i.e. P is *translated* by v:

$$P + v \quad \Rightarrow \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} x + v_1 \\ y + v_2 \end{bmatrix}.$$

The first question one might ask is, Is translation an affine transformation? Consider the situation depicted in Figure 5.3. Point P falls on the line between P_1 and P_2, so $P = (1 - \alpha)P_1 + \alpha P_2$ for some α. If translation is affine, then $P + v$ should fall on the line between $P_1 + v$ and $P_2 + v$. In particular, $P + v$ should have the same parameterization with respect to the line. We thus want to show that $(1 - \alpha)(P_1 + v) + \alpha(P_2 + v) = P + v$:

$$\begin{aligned} (1 - \alpha)(P_1 + v) + \alpha(P_2 + v) &= (1 - \alpha)P_1 + (1 - \alpha)v + \alpha P_2 + \alpha v \\ &= (1 - \alpha)P_1 + \alpha P_2 + v - \alpha v + \alpha v \\ &= P + v \end{aligned}$$

Thus we have shown that translation is an affine operation.

Knowing that translation is affine, we would expect it to have a matrix representation. Yet, it doesn't have a clear matrix form, because any 2D (or 3D) matrix will map $(0, 0)$ (or $(0, 0, 0)$) back to itself:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Translation does not map $\mathbf{0}$ to $\mathbf{0}$, so how can it be represented in matrix form?

Additionally, we know that translation affects points but not vectors. (More specifically, translation does not affect the *representation* of a vector; vectors can be moved around but maintain

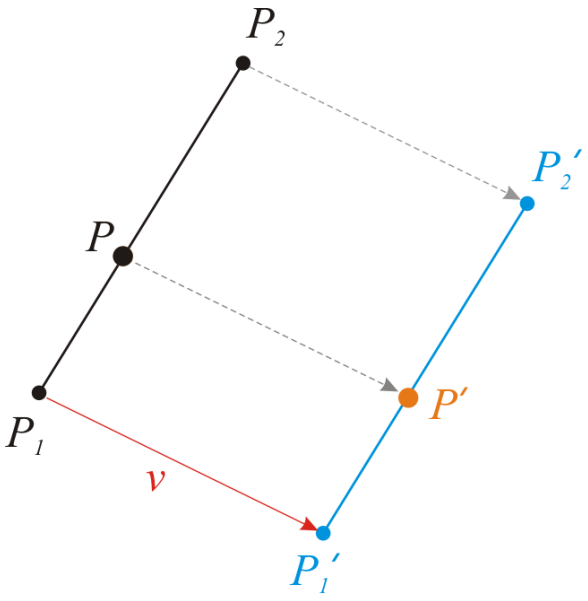


Figure 5.3: Translation is affine: a point P falling on the line between P_1 and P_2 also falls on the line between $P_1' = P_1 + v$ and $P_2' = P_2 + v$, such that $P' = P + v$.

the same representation regardless of their position.) How can a matrix representation distinguish between points and vectors?

Homogeneous coordinates allow us to address all of these problems and construct a matrix representation that affects points but not vectors. To translate a 2D point by a vector $v = [v_1\ v_2]^T$, define the translation matrix $\mathbf{T}(v)$ as

$$\mathbf{T}(v) = \begin{bmatrix} 1 & 0 & v_1 \\ 0 & 1 & v_2 \\ 0 & 0 & 1 \end{bmatrix} \ (2D)\ .$$

In 3D, the translation matrix is

$$\mathbf{T}(v) = \begin{bmatrix} 1 & 0 & 0 & v_1 \\ 0 & 1 & 0 & v_2 \\ 0 & 0 & 1 & v_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \ (3D)\ .$$

Example 5.2

Show that the translation represented by $\mathbf{T}(v)$ affects points but not vectors.

Answer:

Consider the 2D case. Let $P = [x\ y\ 1]^T$ be a point and $u = [u_1\ u_2\ 0]^T$ be a vector.

Then:

$$\begin{aligned}
\mathbf{T}(v)P &= \begin{bmatrix} 1 & 0 & v_1 \\ 0 & 1 & v_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1(x) + 0(y) + v_1(1) \\ 0(x) + 1(y) + v_2(1) \\ 0(x) + 0(y) + 1(1) \end{bmatrix} \\
&= \begin{bmatrix} x + v_1 \\ y + v_2 \\ 1 \end{bmatrix} \\
&= P + v\,.
\end{aligned}$$

So, *the translation matrix has the effect of adding the vector v to P.*

Now apply the translation matrix to a vector:

$$\begin{aligned}
\mathbf{T}(v)u &= \begin{bmatrix} 1 & 0 & v_1 \\ 0 & 1 & v_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 1(u_1) + 0(u_2) + v_1(0) \\ 0(u_1) + 1(u_2) + v_2(0) \\ 0(u_1) + 0(u_2) + 1(0) \end{bmatrix} \\
&= \begin{bmatrix} u_1 \\ u_2 \\ 0 \end{bmatrix} \\
&= u
\end{aligned}$$

So, *vectors are unchanged by translation*.

5.4 Rotation

5.4.1 2D Rotation

How does the rotation transformation change when dealing with homogeneous coordinates? Because rotation affects both points and vectors, our homogeneous rotation matrix must reflect this. By simply adding the identity row that an affine matrix must have, and setting the remaining free entries $a_{i,j}$ to zero, we achieve a new rotation matrix $\mathbf{R}(\theta)$ that rotates points and vectors in homogeneous coordinates:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\,.$$

Example 5.3

Show that $\mathbf{R}(\theta)$ rotates points to points and vectors to vectors.

Answer:

First consider a point $P = [x\ y\ 1]^T$:

$$\begin{aligned} P' \ = \mathbf{R}(\theta)P \ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} x\cos\theta - y\sin\theta + 0(1) \\ x\sin\theta + y\cos\theta + 0(1) \\ 0x + 0x + 1(1) \end{bmatrix} \\ P' \ &= \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \end{aligned}$$

So, P' is also a point.

Now consider a vector $v = [v_1\ v_2\ 0]^T$:

$$\begin{aligned} v' \ = \mathbf{R}(\theta)v \ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} v_1\cos\theta - v_2\sin\theta + 0(0) \\ v_1\sin\theta + v_2\cos\theta + 0(0) \\ 0x + 0x + 1(0) \end{bmatrix} \\ P' \ &= \begin{bmatrix} v_1' \\ v_2' \\ 0 \end{bmatrix} \end{aligned}$$

So, v' is also a vector.

5.4.2 3D Rotation

Rotation in 3D is a more complicated transformation than 2D rotation, because in 3D we must specify an axis about which to rotate. The axis of rotation in 2D is implicitly the z axis, but in 3D we may rotate about the x or y axes, or in fact any arbitrary axis.

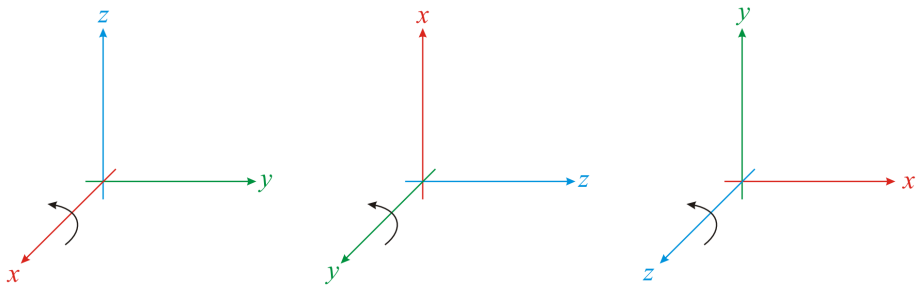


Figure 5.4: 3D rotation occurs about one of the primary axes. Rotation about the (a) x axis; (b) y axis; (c) z axis.

Let's first consider the most simple case, rotation about the z axis(Figure 5.4(c)). In this case, the z coordinate is unchanged by the rotation, and from Equations 11 and 12 we see that:

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z\ . \end{aligned}$$

In matrix notation we can write rotation about the z axis as

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\ .$$

We use a **right-handed** coordinate system to define a positive rotation: a positive rotation angle θ causes a counter-clockwise rotation about some axis when looking from a positive point on the axis towards the origin. Consider Figure 5.4(c), which illustrates rotation about the z axis. As we look down the z axis towards the origin, a counter-clockwise rotation is from x to y.

By our choice of coordinate system, a positive rotation about the x axis (Figure 5.4(a)) rotates from y to z. Thus,

$$\begin{aligned} x' &= x \\ y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta\ . \end{aligned}$$

and the rotation matrix becomes

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\ .$$

Rotation about the y axis (Figure 5.4(b)) is slightly different than the first two cases, due to our right-handed coordinate system. A positive rotation about the y axis rotates from z to x, which causes the sign of the sin terms to change:

$$\begin{aligned} x' &= z\sin\theta + x\cos\theta \\ y' &= y \\ z' &= z\sin\theta - x\cos\theta\ . \end{aligned}$$

The rotation matrix for this case is then:

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\ .$$

5.5 Scaling

So far we have looked at two types of transformation: translation and rotation. These are both examples of rigid-body transformations. A **rigid-body transformation** does not change the size or shape of an object, only its position and/or orientation.

There are important and useful transformations that are *not* rigid-body, however. **Scaling** changes the size of an object by moving points closer to or farther from a fixed point (usually the origin).

In two dimensions, scaling involves two factors s_x and s_y. A point $P = [x\ y]^T$ becomes $P' = [s_x x\ s_y y]^T$. In matrix form we can write the scaling transformation $\mathbf{S}(s_x, s_y)$ as:

$$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}\ .$$

Note that S is said to scale *about the origin*, because $(0, 0)$ is unchanged by it.

Scaling can be either **uniform** – when the scaling factors are equal – or **non-uniform**. Negative scaling factors cause a reflection about the associated axis. See Figure 5.5 for examples of uniform, non-uniform, and reflective scaling transformations.

The scaling matrix in 3D is very similar to the 2D matrix:

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\ .$$

5.6 Shear

The **shear** transformation distorts an object in one direction, usually along an axis. Figure 5.6 illustrates a shear in the x direction; note that the y coordinate of any given point in the cube do not change.

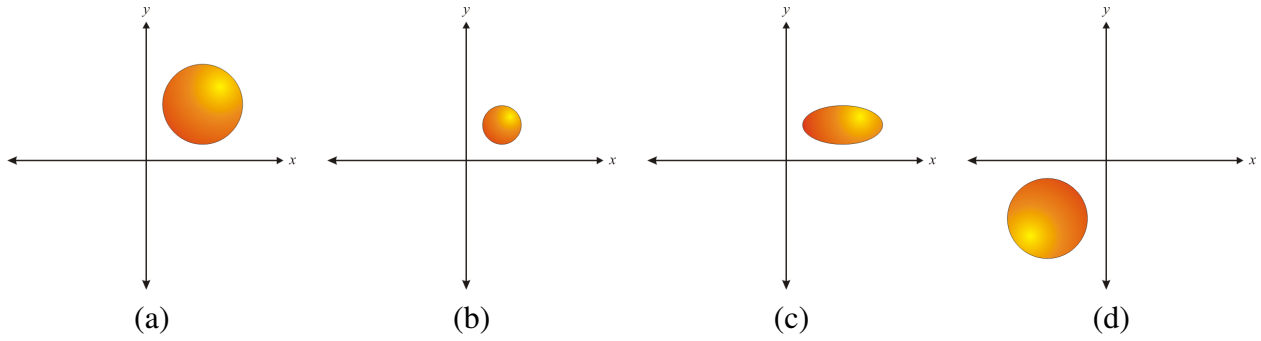


Figure 5.5: Scaling: (a) Original object; (b) Uniformly scaled by $s_x = s_y = 0.5$; (c) Non-uniformly scaled by $s_x = 1$, $s_y = 0.5$; (d) Reflected about both axes by $s_x = s_y = -1$.

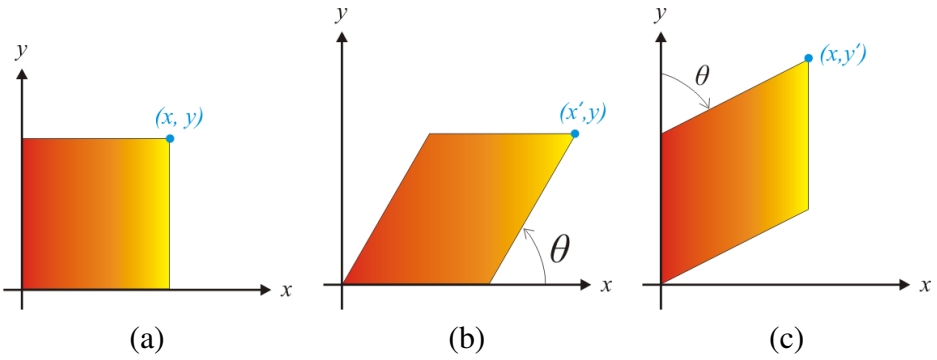


Figure 5.6: Shearing in the 2D can be quantified by the angle θ: (a) the original object; (b) after shearing in the x direction, the y coordinates are unchanged; (c) shearing in the y direction leaves the x coordinate unchanged.

To develop a mathematical description of shear, we can note from Figure 5.6 that the amount of displacement experienced by some point (x, y) depends only on the value of y; all points with the same y value are displaced by the same amount, call it hy:

$$\begin{aligned} x' &= x + hy \\ y' &= y \,, \end{aligned}$$

where h determines what fraction of the y coordinate is to be added to the x coordinate.

It might be more intuitive to quantify the amount of shear by some angle θ, as in Figure 5.6(b); $\theta = 90°$ corresponds to no shear, and lower angles produce more shear. Then, by trigonometry we can write:

$$x' = x + y \cot\theta \,.$$

In matrix notation, 2D shear in the x direction can be expressed by the matrix $\mathbf{H}_x$:

$$\mathbf{H}_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \,.$$

In the y direction, the matrix is similar:

$$\mathbf{H}_y(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ \cot\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

In three dimensions there are more possible ways to shear an object. For instance, when shearing in the x direction we could change the x coordinate based on the y coordinate (as in 2D):

$$\mathbf{H}_{x,y}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} ,$$

or based on the z coordinate:

$$\mathbf{H}_{x,z}(\theta) = \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Shearing is clearly an affine transformation, because the last row of $\mathbf{H}$ is always the identity row.

5.7 Inverse Transformations

Often we find it necessary to "undo" a transformation. The obvious way to undo a transformation $\mathbf{M}$ is to find its inverse $\mathbf{M}^{-1}$ via linear algebra techniques. However, matrix inversion algorithms for 4×4 matrices are relatively expensive.

We can instead use our knowledge of what the transformation matrices represent to find their inverse transformations.

A translation matrix $\mathbf{T}(v)$ adds the vector v to any point that it is applied to. The inverse of this translation, then, is simply to translate by $-v$:

$$\mathbf{T}^{-1}(v) = \mathbf{T}(-v) .$$

A rotation of θ about some axis can easily be undone by rotating about the same axis by $-\theta$:

$$\begin{aligned} \mathbf{R}_x^{-1}(\theta) &= \mathbf{R}_x(-\theta) \\ \mathbf{R}_y^{-1}(\theta) &= \mathbf{R}_y(-\theta) \\ \mathbf{R}_z^{-1}(\theta) &= \mathbf{R}_z(-\theta) \end{aligned}$$

Scaling transformations multiply coordinates by scaling factors s_x, s_y, and s_z. The inverse transformation re-scales by the reciprocal; for example, a scaling by s_x can be undone by a scale factor of $\frac{1}{s_x}$. Thus the inverse of $\mathbf{S}(s_x, s_y, s_z)$ is:

$$\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right) .$$

Example 5.4

What is the inverse transformation for shear?

Answer:

Recall the form of the shear matrix $\mathbf{H}(\theta)$:

$$\mathbf{H}_{x,y}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

We want a matrix $\mathbf{H}_{x,y}^{-1}$ such that $\mathbf{H}_{x,y}^{-1}\mathbf{H}_{x,y} = \mathbf{I}$. Let

$$\mathbf{H}_{x,y}^{-1} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} .$$

Then

$$\begin{aligned} \mathbf{H}_{x,y}^{-1}\mathbf{H}_{x,y} &= \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{11}\cot\theta + a_{12} & a_{13} & a_{14} \\ a_{21} & a_{21}\cot\theta + a_{22} & a_{23} & a_{24} \\ a_{31} & a_{31}\cot\theta + a_{32} & a_{33} & a_{34} \\ a_{41} & a_{41}\cot\theta + a_{42} & a_{43} & a_{44} \end{bmatrix} \end{aligned}$$

To satisfy $\mathbf{H}_{x,y}^{-1}\mathbf{H}_{x,y} = \mathbf{I}$, we can immediately set $a_{11} = a_{22} = a_{33} = a_{44} = 1$ and $a_{13} = a_{14} = a_{21} = a_{23} = a_{24} = a_{31} = a_{32} = a_{34} = a_{41} = a_{42} = a_{43} = 0$. Then:

$$\mathbf{H}_{x,y}^{-1}\mathbf{H}_{x,y} = \begin{bmatrix} 1 & \cot\theta + a_{12} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We need to choose a_{12} to satisfy $\cot\theta + a_{12} = 0$:

$$\begin{aligned} \cot\theta + a_{12} &= 0 \\ a_{12} &= -\cot\theta \\ a_{12} &= \cot(-\theta) \ . \end{aligned}$$

Therefore

$$\mathbf{H}_{x,y}^{-1} = \begin{bmatrix} 1 & \cot(-\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

In general, the inverse shear transformation is

$$\mathbf{H}^{-1}(\theta) = \mathbf{H}(-\theta) .$$

5.8 Concatenating Transformations

It is possible, but unnecessary, to derive single matrices for the most complex affine transformations one can imagine. The transformations we have developed to this point – translation, rotation, scaling, and shearing – constitute a set of "building blocks" that can be used to construct complex *composite* transformations.

Consider a seemingly simple problem: rather than rotating about the origin, we would like to rotate about the point $Q = (x_q, y_q)$. It is not as easy to derive the closed-form equations for $[x'\ y']^T = \mathbf{R}(\theta)[x\ y]^T$ as it was for rotations about the origin. How can we use a combination of our basic transformations to rotate about this arbitrary point?

To accomplish this rotation, we first need to move, or translate, our fixed point Q to the origin, via the transformation $\mathbf{T}(-Q)$:

$$P' = \mathbf{T}(-Q)P .$$

Then our fixed point corresponds to the origin, and we can rotate with our usual matrix $\mathbf{R}(\theta)$:

$$P'' = \mathbf{R}(\theta)P' .$$

After rotation, we need to restore everything to its original position by *un*-translating:

$$P''' = \mathbf{T}(Q)P'' .$$

These three transformations can be combined into one meta-transformation M by *concatenating* the transformation matrices:

$$\begin{aligned} P''' &= \mathbf{T}(Q)P'' \\ &= \mathbf{T}(Q)\mathbf{R}(\theta)P' \\ &= \mathbf{T}(Q)\mathbf{R}(\theta)\mathbf{T}(-Q)P \\ &= \mathbf{M}P \end{aligned} \tag{13}$$

where $\mathbf{M} = \mathbf{T}(Q)\mathbf{R}(\theta)\mathbf{T}(-Q)$ is a matrix that combines all three tranformations together.

In general, we **concatenate** transformations $\mathbf{M}_1, \mathbf{M}_2, \ldots \mathbf{M}_n$ as

$$Q = \mathbf{M}_n \cdots \mathbf{M}_2\mathbf{M}_1P .$$

Note that the order of specifying the transformations seems to be reversed; the reason for this is clear if we write the concatenation with brackets:

$$Q = (\mathbf{M}_n \cdots (\mathbf{M}_2 (\mathbf{M}_1 P))) \ .$$

Because matrix transformations are prefix operations, $\mathbf{M}_1$ first transforms P, then $\mathbf{M}_2$, and so on up to $\mathbf{M}_n$. The order of multiplication is very important, because *matrix multiplication is not commutative*; in select cases it may be that $\mathbf{M}_2\mathbf{M}_1 = \mathbf{M}_1\mathbf{M}_2$, but in general this will not hold.

Example 5.5

Find the matrix $\mathbf{M}$ that represents the transformation $\mathbf{M}_1$ followed by $\mathbf{M}_2$, where:

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{M}_2 = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

What kind of transformation does $\mathbf{M}$ represent?

Answer:

Remember to multiply the matrices together in reverse order:

$$\begin{aligned} \mathbf{M} &= \mathbf{M}_2\mathbf{M}_1 \\ &= \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 3 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & -4 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

$\mathbf{M}_1$ represents a translation $\mathbf{T}((3, -4))$, and $\mathbf{M}_2$ represents a rotation about the z axis by $30°$ ($\cos 30° = \frac{\sqrt{3}}{2}$). $\mathbf{M}$ therefore represents a translation followed by a rotation.

In the above example, we notice the concatenation of two affine transformations resulted in a new affine transformation, because the last row of $\mathbf{M}$ is the identity row. In general, *the concatenation of affine transformations is also affine*; the proof of this fact is left to the reader.

There are several important types of compound transformations that are frequently used in graphics. We will look at each one in detail below.

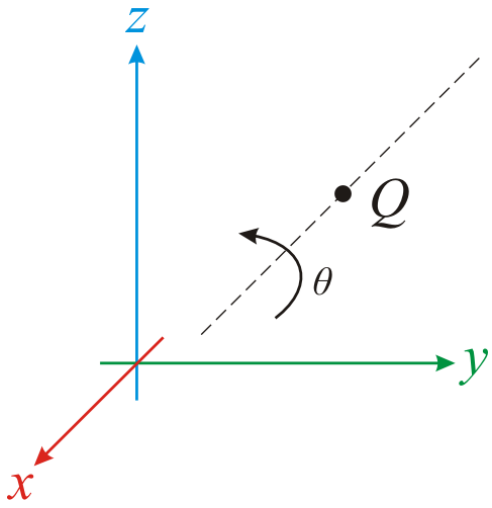


Figure 5.7: Rotation about an axis parallel to the z axis passing through point Q. First Q is translated to the origin, then the rotation takes place, and finally Q is translated to its original position.

5.8.1 Rotation About An Arbitrary Point

Rotation about an arbitrary point was described in the introduction to concatenation. Equation 13 summarizes the basic form of $\mathbf{M}$:

$$\mathbf{M} = \mathbf{T}(Q)\mathbf{R}(\theta)\mathbf{T}(-Q) \; .$$

Figure 5.7 illustrates a rotation about an axis parallel to the z axis that passes through point Q. In this case, the rotation matrix is $\mathbf{R}_z(\theta)$, and then we have:

$$\begin{aligned}
\mathbf{M} &= \mathbf{T}(Q)\mathbf{R}_z(\theta)\mathbf{T}(-Q) \\
&= \begin{bmatrix} 1 & 0 & 0 & x_q \\ 0 & 1 & 0 & y_q \\ 0 & 0 & 1 & z_q \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_q \\ 0 & 1 & 0 & -y_q \\ 0 & 0 & 1 & -z_q \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & x_q - x_q\cos\theta + y_q\sin\theta \\ \sin\theta & \cos\theta & 0 & y_q - x_q\sin\theta - y_q\cos\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} ,
\end{aligned}$$

where $Q = (x_q, y_q, z_q)$ is the fixed point. Note that z_q does not appear in $\mathbf{M}$.

5.8.2 Rotation About An Arbitrary Axis

An arbitrary rotation – i.e. a rotation that is not strictly about one of the main axes – can be achieved with several successive rotations about the main axes.

In Section 5.8.1, we derived a transformation matrix for rotating about an arbitrary point, but assumed that the axis of rotation is parallel to one of the main axes. We will now eliminate this

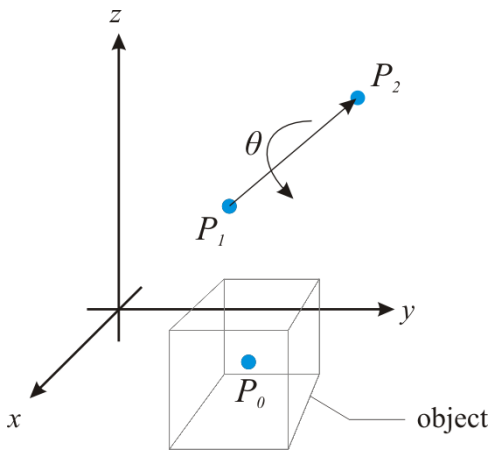


Figure 5.8: Rotation through angle θ about the axis defined by P_1 and P_2, with fixed point P_0,.

assumption, and derive a transformation about an arbitrary axis. Figure 5.8 illustrates the setting. The desired rotation is specified by the fixed point in the rotation P_0, the axis of rotation $u = P_2 - P_1$, and an angle of rotation θ.

From our experience rotating about an arbitrary point, we can surmise that the first step here will be to translate by $-P_0$ and the last step will be to translate by P_0. What happens in between these steps then?

The idea is to transform our object so that the rotation axis u is aligned with one of the main axes, because we know how to rotate about a main axis. Let's choose to align u with the z axis. Then our strategy for composing the final transformation $\mathbf{M}$ is:

1. Translate P_0 to the origin by $\mathbf{T}(-P_0)$.
2. Align u with the z axis with two rotations: $\mathbf{R}_x(\theta_x)$ and $\mathbf{R}_y(\theta_y)$.
3. Rotate by θ about the z axis.
4. Undo our alignment rotations: $\mathbf{R}_y(-\theta_y)$ and $\mathbf{R}_x(-\theta_x)$.
5. Translate back to P_0: $\mathbf{T}(P_0)$.

Having set out this strategy, we have reduced our problem to that of finding the proper angles θ_x and θ_y to align our axis of rotation with the z axis; everything else we know how to do. To approach this alignment problem, the magnitude of u is unimportant. We are only interested in the direction of u, so define v as a unit-length direction vector:

$$v = \frac{u}{|u|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_x \end{bmatrix} .$$

The vector v is depicted in Figure 5.9(a). We can find θ_x and θ_y by projecting v onto the $y-z$ and $x-z$ planes, respectively. From Figure 5.9(a), we see that:

$$\begin{aligned} \cos\theta_y &= \tfrac{d}{|v|} = d \\ \sin\theta_y &= \alpha_x \quad , \end{aligned}$$

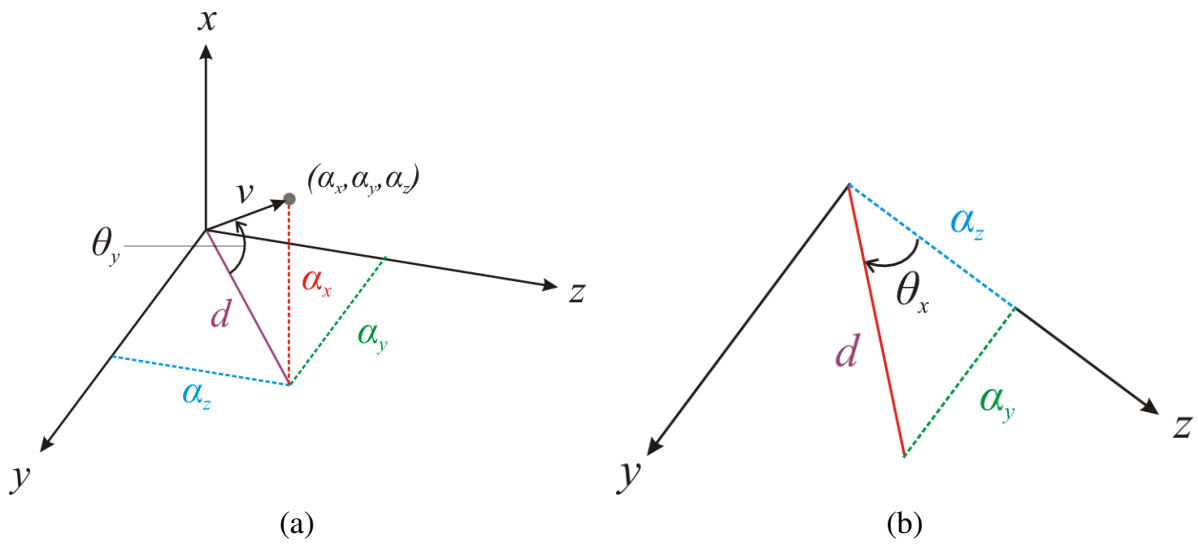


Figure 5.9: Determining θ_x and θ_y: (a) θ_y can be found from d and α_x; (b) θ_x follows from d, α_x, and α_z.

where $d = \sqrt{(\alpha_y^2 + \alpha_z^2)}$. Figure 5.9(b) shows the projection of v onto the $y - z$ plane. From trigonometry, we can determine the relevant functions of θ_x:

$$\begin{aligned} \cos\theta_x &= \frac{\alpha_z}{d} \\ \sin\theta_x &= \frac{\alpha_y}{d} \end{aligned}$$

It is not necessary to determine values for θ_x and θ_y; to build the rotation matrices $\mathbf{R}_x$ and $\mathbf{R}_y$, it is sufficient to have expressions for cosine and sine of the angles. The rotation matrices that will align v with the z axis are then:

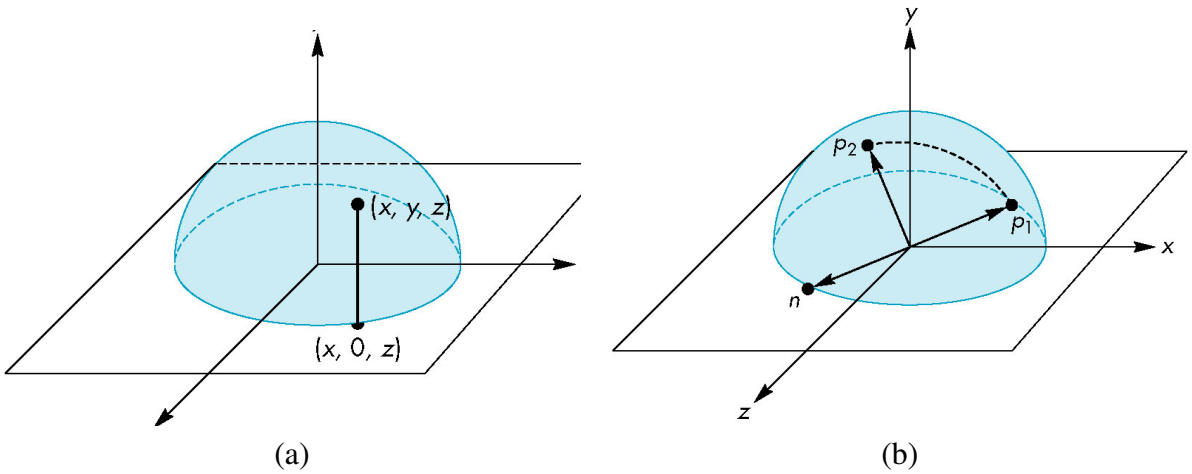
$$\mathbf{R}_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\alpha_z}{d} & -\frac{\alpha_y}{d} & 0 \\ 0 & \frac{\alpha_y}{d} & \frac{\alpha_z}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{R}_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

To undo the alignment rotations, we note that $\sin(-\theta) = -\sin\theta$ and $\cos(-\theta) = \cos\theta$. Therefore,

$$\mathbf{R}_x(-\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\alpha_z}{d} & \frac{\alpha_y}{d} & 0 \\ 0 & -\frac{\alpha_y}{d} & \frac{\alpha_z}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{R}_y(-\theta_y) = \begin{bmatrix} d & 0 & \alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ -\alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Now we can bring together everything in a single transformation $\mathbf{M}$ that rotates about an arbitrary axis. Following the strategy we set out above, and remembering to write the matrices in reverse order, we find:

$$\mathbf{M} = \mathbf{T}(P_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-P_0)\,.$$



(a) (b)

Figure 5.10: Virtual Trackball: (a) unproject a 2D point to a 3D point; (b)finding the axis and the angle of rotation.

5.8.3 Interface to arbitrary 3D rotation

For general 3D rotation, the user needs to provide the axis u and the angle θ. In order to use mouse movements for obtaining this kind of 3D information, we have to use the concept of *un-projection* that generates a 3D coordinate from a 2D coordinate. Notice that un-projection is not unique, and there are many ways to do this. A reasonable method is to use *virtual trackball* concept. In this method, we imagine 2D points resulting from mouse inputs are actually 3D and on the *unit hemisphere*(see (a) in Figure 5.10). To do this, assume that we have (x, y), the input 2D position, then we map it to (x, y, z) as the corresponding 3D points on the hemisphere (i.e $z = \sqrt{(1 - x^2 - y^2)}$). To ensure a real value for z, the condition $x^2 + y^2 \leq 1$ must be met. To satisfy this condition, we assume the mouse coordinates are inside of the unit sphere:

$$0 \leq x \leq \frac{\sqrt{(2)}}{2}$$
$$0 \leq y \leq \frac{\sqrt{(2)}}{2}.$$

For general coordinates (X, Y), a 2D transformation is necessary to map between the (X, Y) and (x, y). Now let P_1 and P_2 be two 3D points(the output of un-projection). We can form the vectors $v_1 = P_1 - O$ and $v_2 = P_2 - O$. The normal n to these vectors is a good selection for the axis of rotation or $u = n = v_1 \times v_2$ (see (b) in Figure 5.10). We also use the angle between v_1 and v_2 as θ.

6 Viewing

As mentioned in Section 1, most current display devices such as computer monitors and printers are two-dimensional raster devices. To obtain a planar view of a 3D object, it is necessary to somehow omit the third dimension; the sequence of operations that leads to the planar visualization of an object is called the *viewing pipeline*.

In the most general sense, **projection** transforms an object to a lower-dimensional space. In computer graphics, projection typically refers to a transformation that maps an object in 3D world coordinates to a flat representation on a specified plane known as the *viewplane*.

The steps in the viewing pipeline are somewhat analogous to the process of taking a photograph. The application must:

- Position the camera/viewer at a particular point;
- Decide on an orientation for the camera;
- Choose a suitable lens.

The pipeline must also perform other duties that a camera isn't burdened with, such as deciding which objects or portions thereof are visible to the camera.

One benefit of computer graphics over a traditional camera is that these specifications can be easily changed with little cost or effort. We can move the camera to infeasible locations, or move the camera along a path in 3D that may be impossible in the real world. These are contributing factors to the immense popularity of computer-aided special effects in the film industry, and also to the immersiveness of interactive games. Projections can also be manipulated to yield impossible scene renderings, such as the paintings of Escher; see Figure 6.1.

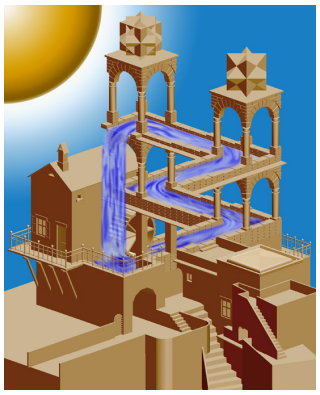


Figure 6.1: An M.C. Escher-like rendering of a computer graphics model. Source: http:/www.cs.technion.ac.il/gotsman/Escher/

6.1 Orthographic Projections

A very simple type of projection is orthographic projection. **Orthographic** projection does not account for the foreshortening that occurs in optical devices such as cameras or human eyes. That

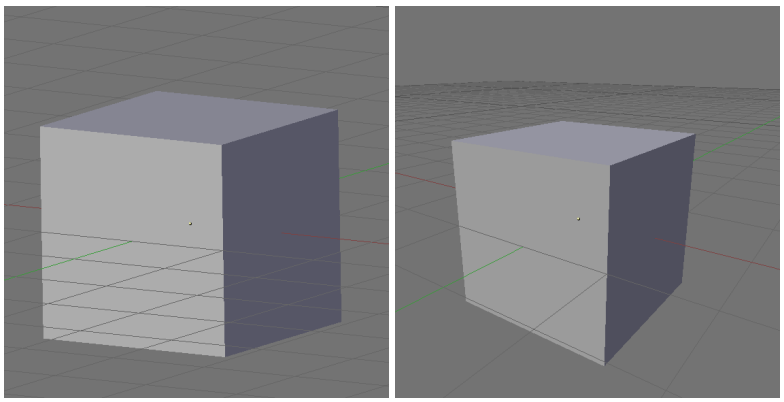


Figure 6.2: Orthographic (left) versus perspective (right) projections. Note that the gridlines remain parallel under orthographic projection, while they converge to a vanishing point in the perspective projection.

is, parallel lines do not converge to a vanishing point under an orthographic projection; they are always parallel. Figure 6.2 illustrates the difference between an orthographic and perspective projection.

Consider a simple tetrahedron, defined by the following four points in 3-space (Figure 6.3(a)):

$$A = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad B = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \quad D = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

Suppose we want to project this object onto the $x - y$ plane orthographically. This is done by simply setting the z component of each vertex to zero (Figure 6.3(b)):

$$A' = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad B' = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \quad C' = \begin{bmatrix} 1 \\ 3 \\ 0 \end{bmatrix} \quad D' = \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix}$$

This projection can be expressed in matrix notation, using homogeneous coordinates.

$$\begin{bmatrix} x' \\ y' \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

What do you notice about this matrix? The main characteristic is that there is a row of all zeros; therefore, the matrix is singular, or not invertible. Up to this point, all of the transformations we've looked at (rotation, translation, and so on) have been invertible, yet projection is not. This intuitively makes sense, though, because to move to a lower-dimensional space we have to discard some information. (Note that in rendering APIs such as OpenGL, projection *is* invertible because the discarded information is retained in an auxiliary structure, called the *depth buffer*.)

$$A = (1,1,1) \quad B = (2,2,2) \quad C = (1,3,1) \quad D = (3,2,1)$$

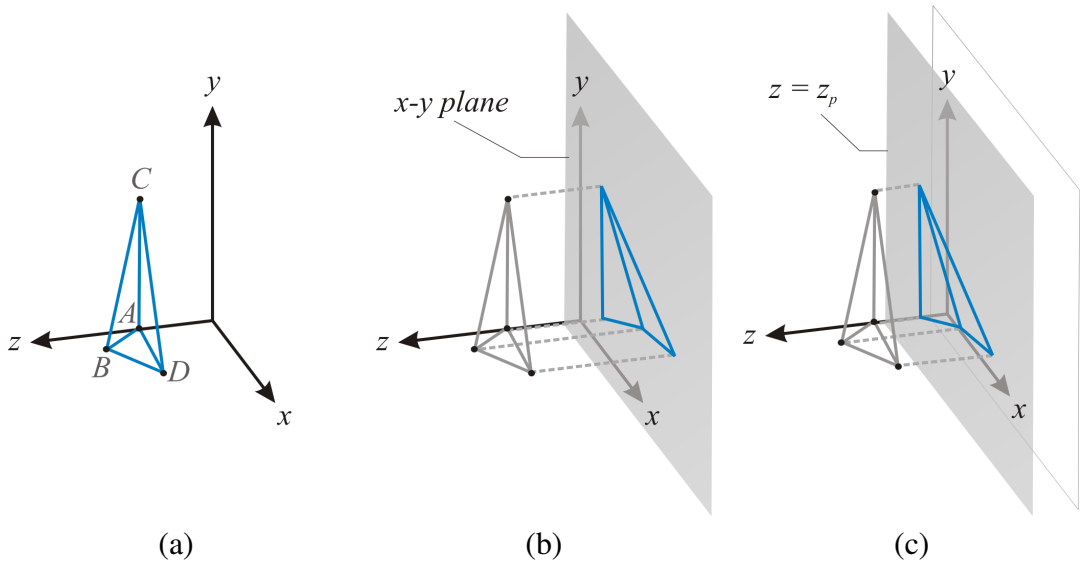


Figure 6.3: Setting for orthographic projection example: (a) a 3D object; (b) its projection onto the $x - y$ plane; (c) its projection onto a plane perpendicular to the z axis.

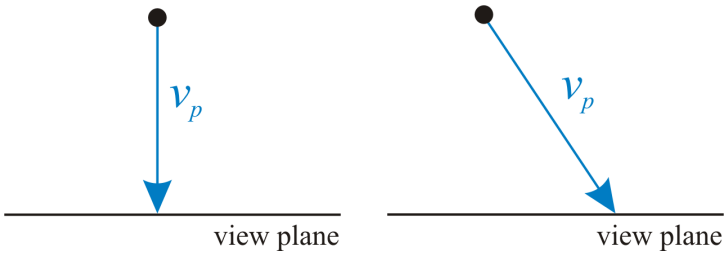


Figure 6.4: Orthogonal (left) versus oblique (right) projection.

Now supposed we move the viewplane along the z axis; for example, to z_p (Figure 6.3(c)). Now the projection matrix becomes

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & z_p \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Still, this projection is not invertible.

The previous projections are called *orthogonal*, because we use a projector, v_p, that is orthogonal to the view plane. If the projector is not orthogonal to the view plane, the projection is still orthographic but is instead known as an *oblique* projection.

Orthographic projections are most often used to produce front, side, and top views of an object. Because orthographic projections are length- and angle-preserving, they are well suited to

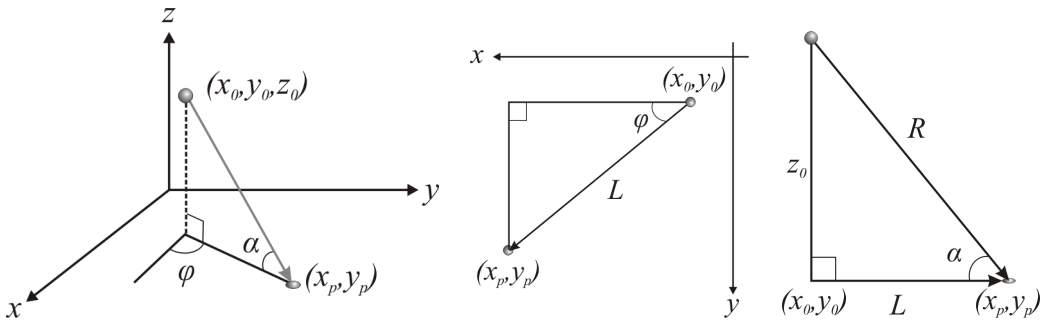


Figure 6.5: Oblique projection: (a) the point (x_0, y_0, z_0) is projected to (x_p, y_p); (b) top view (c) side view.

engineering and architectural applications; the lengths and angles can be accurately represented and even measured from a rendering. However, orthographic projections are not very effective for visualizing objects.

Let's derive the projection matrix for oblique projection. For this derivation, we will use the setting depicted in Figure 6.5(a). The point (x_0, y_0, z_0) is projected to $(x_p, y_p, 0)$. If we let

$$\begin{aligned} L &= (x_p, y_p, 0) - (x_0, y_0, 0) \\ R &= (x_p, y_p, 0) - (x_0, y_0, z_0) \ , \end{aligned}$$

then we can describe any oblique projection with two angles, α and ϕ, where α is the angle between L and R (Figure 6.5(b)), and ϕ is the angle between L and the x-axis (Figure 6.5(b)).

Given some values for α and ϕ, what is the projection matrix? By Figure 6.5(b), we can see that

$$\begin{aligned} \cos\phi &= \frac{(x_p - x_0)}{L} \ , \\ \sin\phi &= \frac{(y_p - y_0)}{L} \ , \end{aligned}$$

therefore

$$\begin{aligned} x_p &= x_0 + L\cos\phi \ , \\ y_p &= y_0 + L\sin\phi \ . \end{aligned}$$

To determine L, we can turn to Figure 6.5(c):

$$\cot\alpha = L/z_o \longrightarrow L = z_0 \cot\alpha$$

Substituting this result into the above expressions for x_p and y_p, we end up with the following matrix for oblique projection:

$$\begin{bmatrix} x_p \\ y_p \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cot\alpha\cos\phi & 0 \\ 0 & 1 & \cot\alpha\sin\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} .$$

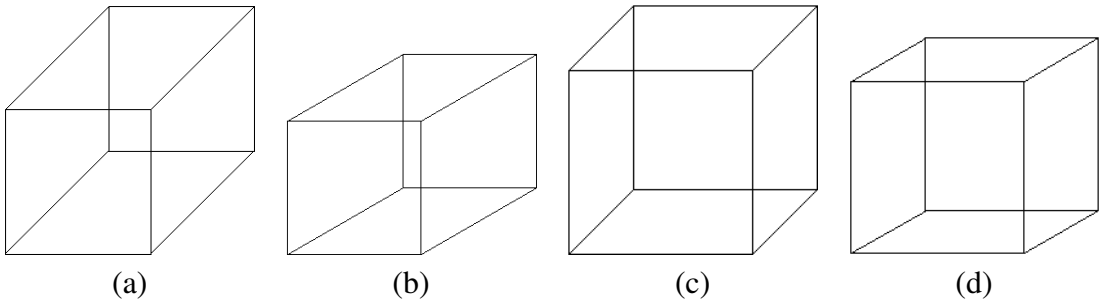


Figure 6.6: Cavalier and cabinet projection: (a) cavalier, $\alpha = 45°$, $\phi = 45°$; (b) cavalier, $\alpha = 45°$, $\phi = 30°$; (c) cabinet, $\alpha = 63.4°$, $\phi = 45°$; (d) cabinet, $\alpha = 63.4°$, $\phi = 30°$.

Note that orthogonal projection is a special case of oblique projection, with $\alpha = 90°$ (and therefore $L = 0$).

There are other values for α and ϕ that are often used. When $\phi \in \{30°, 45°\}$ and $\alpha = 45°$, it is called a *cavalier* projection; this is a useful method for visualizing a cube, for instance. *Cabinet* projection uses values of $\alpha = \tan^{-1} 2 \approx 63.4°$ and $\phi \in \{30°, 45°\}$; this is similar to cavalier projection, but with less depth. See Figure 6.6.

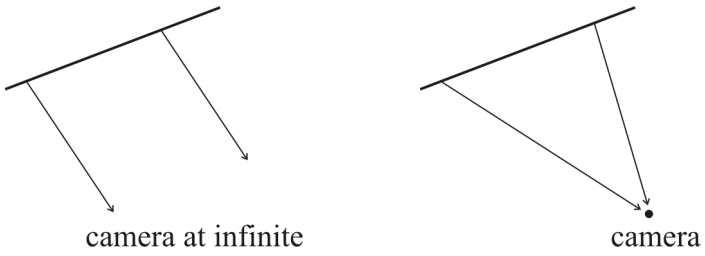


Figure 6.7: Left: in parallel projection, the camera is at infinite. Right: in perspective projection, the camera is close to the object and the incoming rays are no longer parallel.

6.2 Perspective Projection

In orthogonal projections, we are essentially assuming that the camera is infinitely far from the object being viewed, so that the incoming rays are parallel.

Perspective projection is what we are used to in most optical systems, such as a camera or human eye, where the camera is a finite distance from the object. In this case, rays from the object to the camera are no longer parallel, as in Figure 6.7. To derive the mathematical form of perspective projection, we must first formalize the setting (Figure 6.8(a)). Assume the view plane is perpendicular to the z-axis, positioned at $z = d$. Let the camera be positioned at the origin, $O = (0, 0, 0)$. We will consider the projection of some point $P = (x, y, z)$, for which the coordinates after projection are $P' = (x_p, y_p, z_p)$; of course, $z_p = d$.

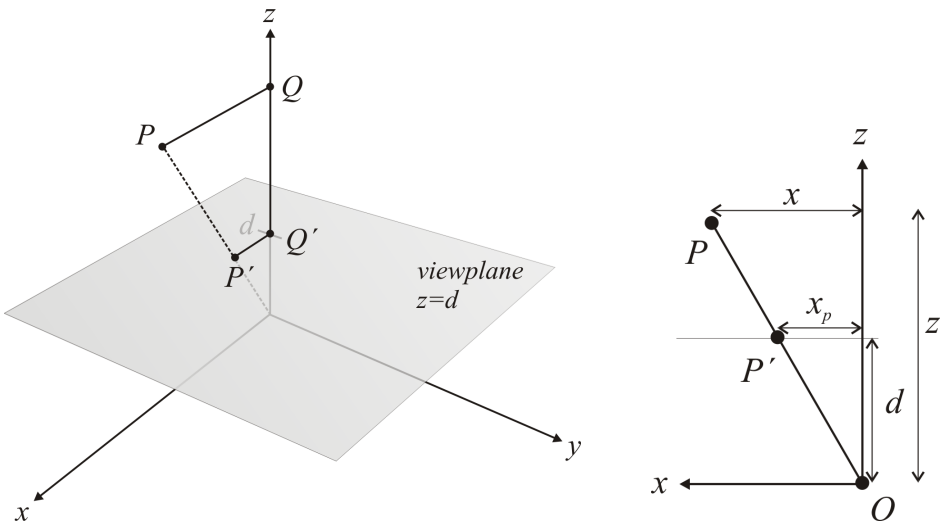


Figure 6.8: Setting for the derivation of perspective projection. (a) general setting; (b) Looking down the y-axis.

To discover expressions for x_p and y_p, we can consider the problem from two different angles: looking down the y-axis to the $x - z$ plane (Figure 6.8(b)); and, looking down the x-axis to the $y - z$ plane.

Consider Figure 6.8(b). Because $\triangle OQ'P'$ and $\triangle OQP$ are similar triangles, we know that the ratios $\frac{x_p}{x}$ and $\frac{d}{z}$ must be equal. Therefore:

$$x_p = x\frac{d}{z} .$$

When looking down the x-axis, the situation is very similar. In this case, we have:

$$y_p = y\frac{d}{z} .$$

And finally, $z_p = d$.

How can these expressions be encapsulated in a matrix? There is a problem, because the equations are non-linear. The projection is also not invertible, and is not affine. Therefore, there is no matrix form of perspective projection, at least not in the current framework.

The solution is to extend affine space in such a way that perspective projections have a matrix form. This new space is known as *projective space*. In projective space, homogeneous coordinates are modified to allow the last component to have a value other than 0 and 1. Recall that the 4D homogeneous coordinate $[x\ y\ z\ w]^T$ is interpreted as a point if $w = 1$ and a vector if $w = 0$. What happens if we allow w to take on other values?

Consider an example point $P = (x, y, z)$, whose homogeneous coordinate is $[x\ y\ z\ 1]^T$. In

projective space, the coordinate of P is extended to

$$P = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}.$$

Example 6.1

What is the 3D coordinate of $[15\ 24\ -6\ 3]^T$?

Answer:

Here, $w = 3$. Therefore, we can divide through by w to get the actual 3D coordinate.

$$\frac{1}{3}\begin{bmatrix} 15 \\ 24 \\ -6 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \\ -2 \\ 1 \end{bmatrix} = (5, 8, 2)$$

In general, we prefer to have $w = 1$ for points. However, by allowing w to change, we can represent a broader class of transformations; in particular, we can represent perspective projections. Consider the following matrix:

$$\mathbf{M}_P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}.$$

In our new projective space, this matrix performs a perspective projection.

Example 6.2

Show that $\mathbf{M}_P$ performs a perspective projection.

Answer:

Let $P = (x, y, z)$. Then:

$$M_P P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \begin{bmatrix} x\frac{d}{z} \\ y\frac{d}{z} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

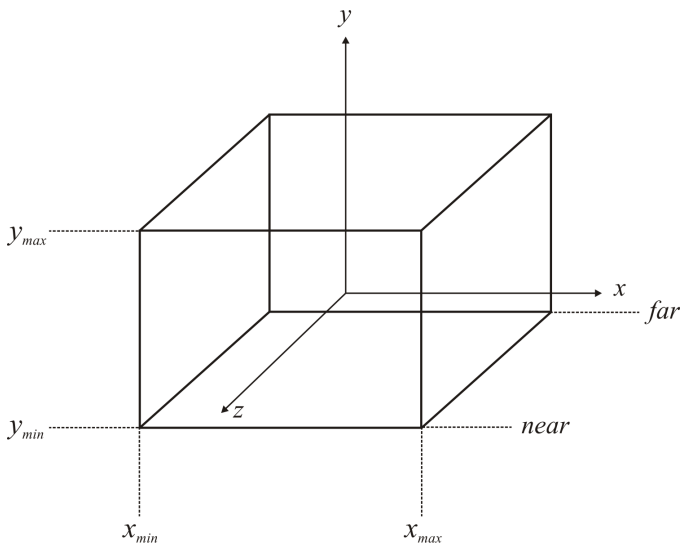


Figure 6.9: An orthographic view volume is defined by six scalars: x_{min}, x_{max}, y_{min}, y_{max}, *near*, and *far*.

> At first glance, it is not obvious that this is the correct result. However, after dividing through by $w = \frac{z}{d}$, we obtain the expected values for the projection of P onto $z = d$.

6.3 View Volumes

In reality, a camera can see everything that has an unobstructed path to it. However, when rendering a synthetic 3D scene, it can become very expensive to consider everything in the world. A *view volume* is used to define a bounded space relative to the camera, outside of which objects are immediately removed from consideration during rendering.

The classification of view volumes is the same as the classification of projections: view volumes are either orthographic or perspective.

An **orthographic** view volume is a right parallelepiped; see Figure 6.9. However, a **perspective** view volume is a portion of a rectangular pyramid whose apex is at the camera. Such a shape is known as a frustum, and can be defined by the same parameters as an orthographic projection (left, right, bottom, top, near, and far). It is often more natural to specify a frustum with only the near and far planes, and an angle representing the *field of view* (abbreviated *fov*). OpenGL supports both methods for specifying a frustum, with its glFrustum(x_{min}, x_{max}, y_{min}, y_{max}, *near*, *far*) and gluPerspective(*fov*, *aspect*, *near*, *far*) commands; see Figure 6.10 for an illustration of such a view volume.

In many applications, such as interactive simulations, we would like to allow a user to specify an arbitrary view volume. A view volume can be specified with a center of projection c, a normal vector n to the view plane, an up vector v, and a second axis u orthogonal to n and v.

Specifying the up vector v is often not intuitive for a user, because it has to satisfy $n \cdot v = 0$.

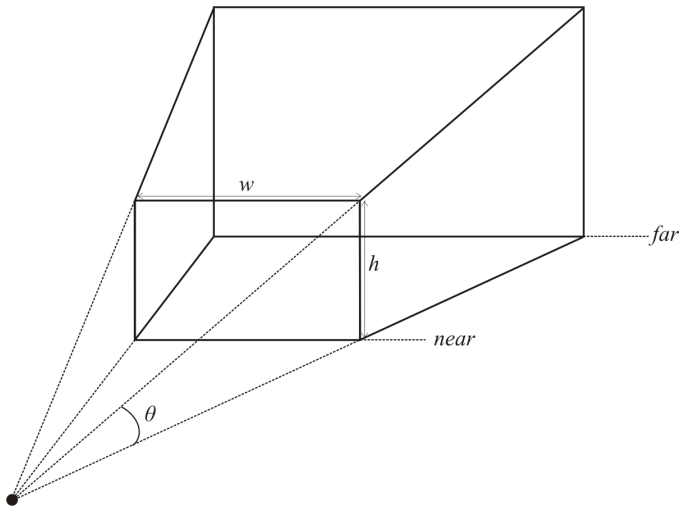


Figure 6.10: A perspective view volume (frustum) is defined by four scalars: *fov*, *aspect* $= \frac{w}{h}$, *near*, and *far*.

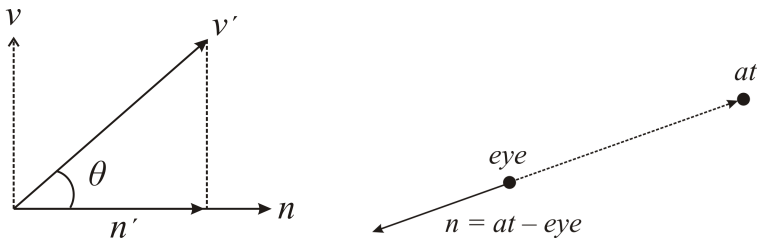


Figure 6.11: User specification of a view volume. Left: An orthogonal vector v can be computed from a normal vector n and an approximation vector v'. Right: A view volume in OpenGL is specified with an eye point and a look-at point; the normal vector can be computed from that.

A sensible strategy is to have the user specify only an approximate orientation of v, call it v', and have the application automatically calculate v to satisfy $n \cdot v = 0$.

Consider Figure 6.11. We can say that $v' = n' + v$, or:

$$v = v' - n' \ .$$

The vector n' is the projection of v' onto n, which from linear algebra we know to be $n' = (v' \cdot n) \cdot n$. Thus:

$$v = v' - (v' \cdot n) \cdot n$$

is an appropriate choice for the up vector, based on the approximation v'. Note that v is orthogonal to n as required: $(v \cdot n) = (v' \cdot n) - (v' \cdot n) \cdot (n \cdot n) = 0$ because $n \cdot n = 1$. And, with n and v, the third vector u folows easily as $u = v \times n$.

In OpenGL, a view volume is specified with a slightly different set of parameters: an eye/camera location *eye*, a look-at point *at*, and an up vector approximation up; see Figure 6.11. From the eye point and look-at point, the normal vector n can be easily computed, and the remaining orthogonal vectors can be computed as before.

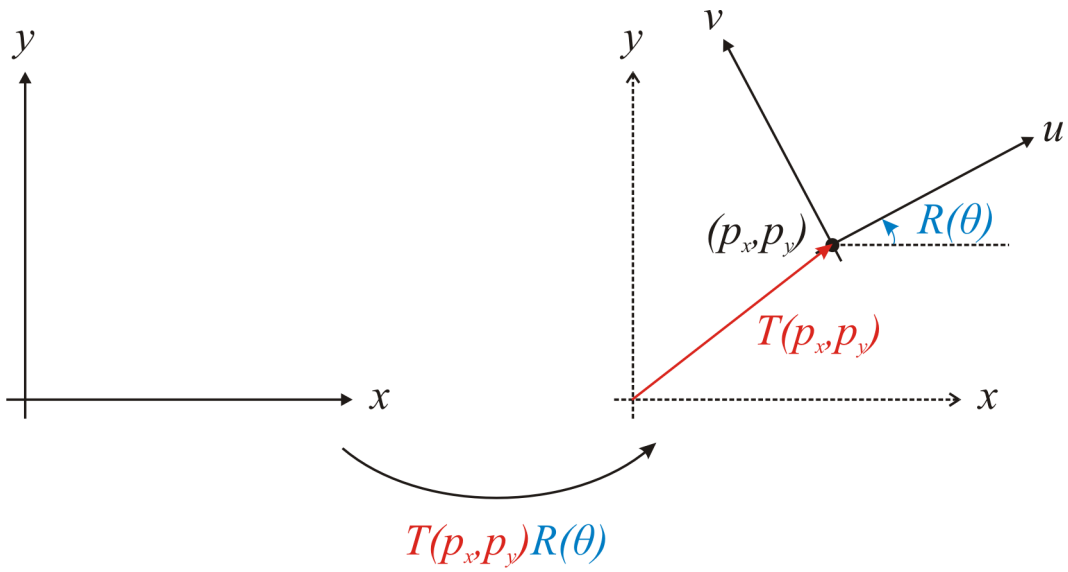


Figure 6.12: To change from the standard frame to the camera frame requires a rotation $R(\theta)$ followed by a translation $T(p_x, p_y)$.

6.4 View (Camera) Transformation

A camera frame can be defined with an origin plus some coordinate axes. Given such a definition, how is the view transformation matrix defined? The purpose of the view transformation matrix is to perform a change of frame, from the standard frame to the camera frame. Suppose the camera frame is defined by three variables P, n, and v, where $P = [p_x\ p_y\ p_z]^T$ is the origin point, $n = [n_x\ n_y\ n_z]^T$ is the view plane normal, and $v = [v_x\ v_y\ v_z]^T$ is a vector lying in the view plane and indicates the "up" direction of the frame (like the y axis in a standard Euclidean frame).

From this information, we can infer a second vector lying in the plane that is orthogonal to both n and v. Let u be the cross-product of n and v, $u = n \times v$, which is orthogonal to both by definition. As these vectors do not necessarily have unit length, let u', v', and n' be normalized versions of u, v, and n.

As illustrated in Figure 6.12, a frame transformation involves a rotation and a translation. The rotation $R(\theta)$ rotates the x and y axes to align with u and v, respectively. The translation $T(x, y)$ then moves the origin $(0, 0, 0)$ to the origin of the camera frame P.

The form of the translation is easily extracted from the frame parameters; the origin P is all we need, and the transformation has the form (in 3D):

$$\mathbf{T}(x, y, z) = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For the rotation transformation, first consider a simple 2D setting (Figure 6.13). From basic

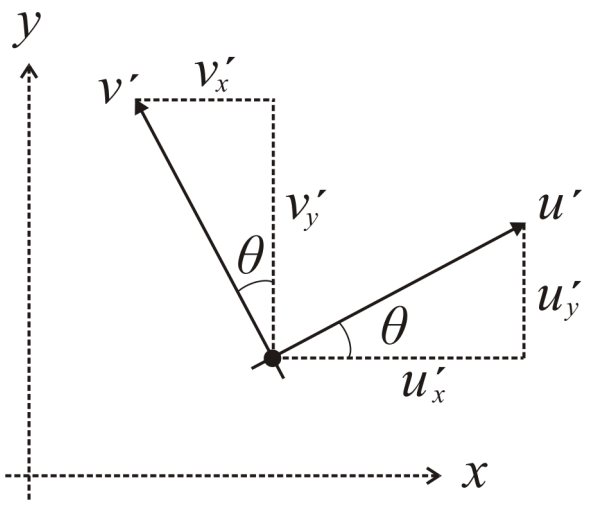


Figure 6.13: The angle of rotation can be derived from the u and v axes.

trigonometry, we have:

$$\cos\theta = u'_x = v'_y \,,$$
$$\sin\theta = u'_y = -v'_x \,,$$

because the length of the hypotenuse is unity. Therefore:

$$\mathbf{R}_{2D}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u'_x & v'_x & 0 \\ u'_y & v'_y & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

Extending this idea to 3D, we find:

$$\mathbf{R}(\theta) = \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We have now derived the appropriate matrices for a frame transformation. However, in practice we usually do not want to explicitly change frames in our graphics pipeline. Since we are working only with vertex data, it is more natural to perform an object transformation; that is, instead of changing to the camera frame (aligning x and y with u and v), we transform the objects such that the standard frame becomes the camera frame (align u and v with x and y). See Figure 6.14.

The necessary object transformation is just the inverse of the frame transformation: a translation by $(-p_x, -p_y, -p_z)$ followed by a rotation of $-\theta$. Thus the model-view matrix, denoted $\mathbf{V}$, is:

$$\mathbf{V} = \mathbf{R}(-\theta)\mathbf{T}(-P) = \begin{bmatrix} u'_x & u'_y & u'_z & 0 \\ v'_x & v'_y & v'_z & 0 \\ n'_x & n'_y & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

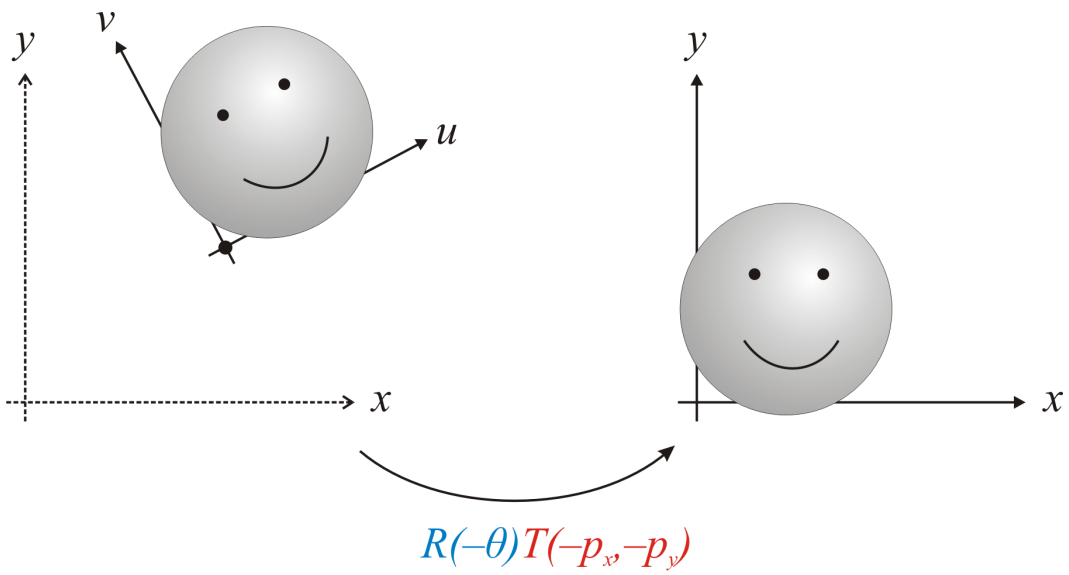


Figure 6.14: By transforming the objects appropriately, the camera frame becomes the standard frame.

6.5 The Viewing Pipeline

Now we can wrap up everything into a pipeline through which all 3D geometry passes through on its way to the rendering device. Each vertex of an object is multiplied by a sequence of matrices in a specific order. The viewing pipeline consists of the following matrix operations:

- Each vertex is multiplied by the modeling transformation matrix, $\mathbf{M}$, which transforms the object from its local model coordinates to world coordinates. For example, a model of a lamp might have local coordinates where the base of the lamp sits on the $x - z$ plane, and the modeling transformation would translate it to sit on a table.
- The view transformation $\mathbf{V}$ is then applied to convert world coordinates to camera coordinates.
- Then, the projection matrix $\mathbf{P}$ is applied, reducing 3D geometry to a 2D coordinate system.
- The clipping transformation removes geometry that is not visible in the rendering window, yielding normalized 2D coordinates.
- Finally, the viewport transformation maps all coordinates to proper display coordinates and the final image can be displayed.

See Figure 6.15. In OpenGL, the first two steps in the pipeline – modeling and viewing – are merged together in the so-called model-view matrix. It is helpful to remember that the modelview matrix is the product of two matrices, $\mathbf{VM}$ ($\mathbf{M}$ is applied first, and so appears last in the product).

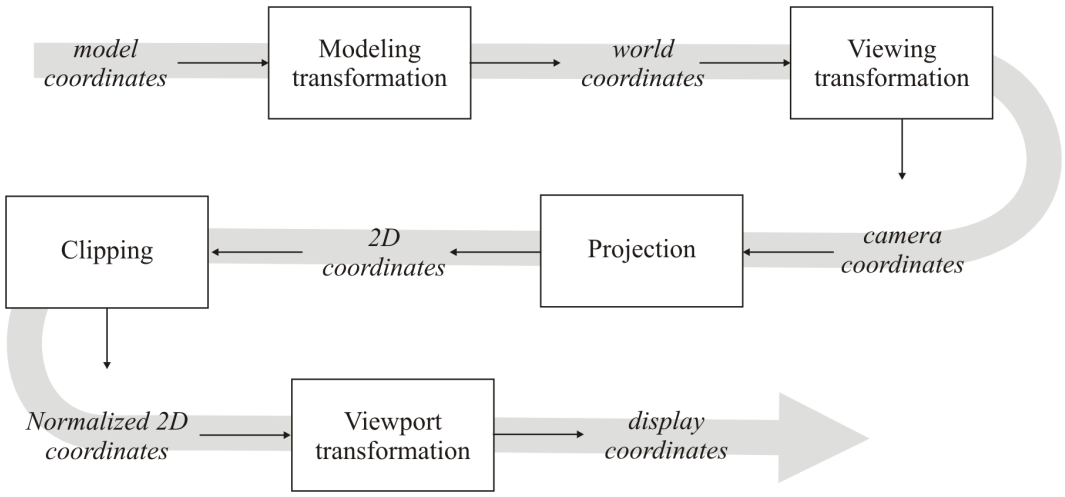


Figure 6.15: The viewing pipeline maps 3D model coordinates to 2D display coordinates.

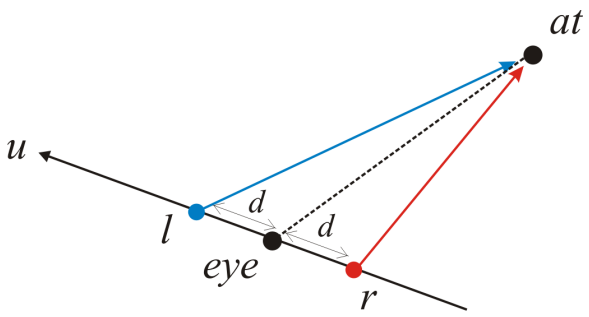


Figure 6.16: A simple approach to generating a stereo view is to display a single camera along each direction of the u axis of the camera frame.

6.6 Stereo Viewing

So far the camera models we have considered are all *cycloptic*, with only one optical sensor. A *stereo* view can make a much more realistic rendering by conveying the depth of the scene.

To make a stereo view of a scene, two pictures – a "left eye" and a "right eye" view – are rendered from slightly different camera positions, but with the same look-at location.

To render a stereo view of a scene, we must decide where to put the left and right cameras. A simple approach is to begin with a regular camera based on a single look-at point, establish the camera frame (u, v, n), and then define the left and right cameras as slight displacement d of the single camera along the u direction:

$$\begin{aligned} l &= \mathit{eye} - du \,, \\ r &= \mathit{eye} + du \,. \end{aligned}$$

For example, human eyes are typically three inches apart, so a reasonable choice for d might be 1.5 inches.

7 Modeling

At a very coarse level, computer graphics can be broken into two main disciplines: *modeling* and *rendering*. The former is concerned with the description of an object, while the latter is concerned with creating an image of the object based on its description.

There are many approaches to modeling objects, some of which will be presented in the following pages. For instance, suppose we want to render a scene containing a sphere. The sphere could be described with its mathematical definition, $r^2 = x^2 + y^2 + z^2$; alternatively, it could be described as a set of three-dimensional points that together approximate the sphere.

Which description is more desirable? The best description depends on the application, which is why there are so many different modeling methods in use today. In the sphere example, the discretized approximation is better for fast rendering, whereas the mathematical description is preferable for high quality rendering or easy collision detection.

7.1 Implicit Modeling

Modeling a sphere by its mathematical description is an instance of *implicit* modeling, so named because there is no explicit description of the sphere's surface. Instead, the surface must be inferred from the mathematical description by finding all points that satisfy $r^2 = x^2 + y^2 + z^2$.

Example 7.1

The implicit description of a sphere of radius r is $x^2 + y^2 + z^2 = r^2$.

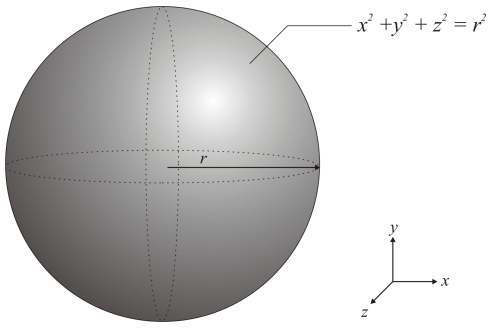


Figure 7.1: An implicit sphere.

7.2 Parametric Curves

As the name implies, parametric models are models defined by parametric functions. They have an associated parameter range, and rendering is accomplished by evaluating the function over some sampling of the parameter range. A nice benefit is that the quality of a rendering can be easily varied by changing the sampling rate of the parameter space.

Parametric models can describe a wide range of curves, surfaces, volumes, and so on. We will consider parametric curves (one-dimensional), which have the general form

$$Q(t) = (X(t), Y(t)) \ ,$$

for $t \in [t_{min}, t_{max}]$. It is helpful to think of a parametric curve as the continuous deformation of a line segment: the function $Q(t)$ deforms the line segment $[t_{min}, t_{max}]$. Another way to think about parametric curves is to regard t as *time*, and the curve generated by $Q(t)$ as the path of a particle over time.

Example 7.2

Consider the curve defined by

$$Q(t) = (R\cos t, R\sin t)\,, \quad 0 \le t \le \frac{3}{2}\pi.$$

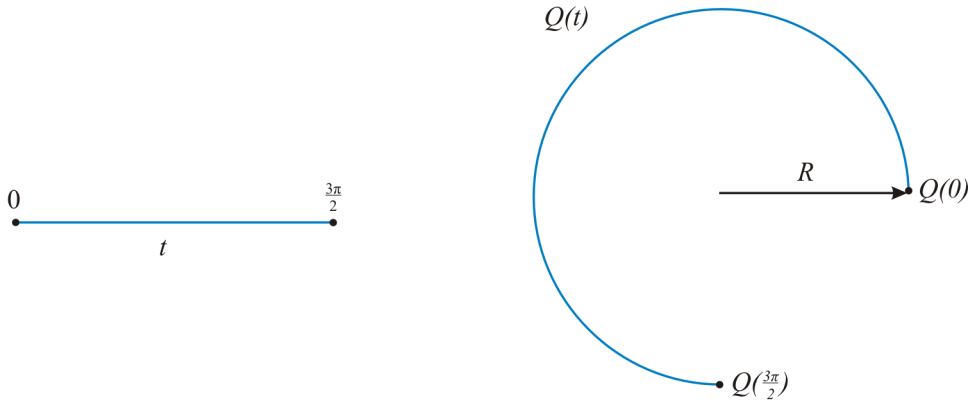


Figure 7.2: Left: the parameter space. Right: the curve $Q(t)$ evaluated over the parameter range.

In general, which modeling approach is best in terms of expressiveness and efficiency: implicit or parametric? Implicit modeling involves root-finding, which for general functions is very difficult and inefficient. For some specific applications, they may be suitable, but in general, parametric curves are more efficient to work with and able to model more complex objects.

Example 7.3

Consider the curve defined by

$$Q(t) = \left(t^2, 2t + 1\right), \quad 0 \le t \le 3\,.$$

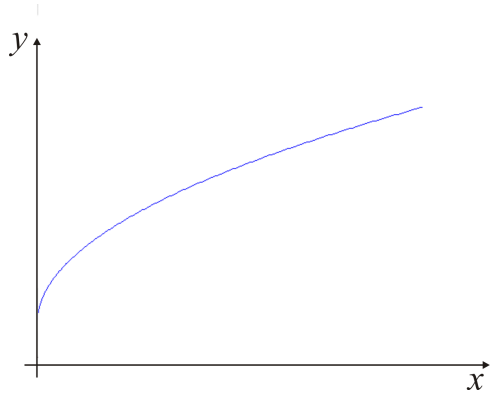


Figure 7.3: Plot of $Q(t) = (t^2, 2t + 1)$.

$Q(t)$ defines a parametric polynomial curve. In particular, it describes a quadractic (degree 2) curve.

Polynomials have many desirable properties from a computer graphics persepective. They are easy to evaluate, and have excellent mathematical properties, like differentiability.

However, polynomials are not easily visualized just by looking at the mathematical definition, and therefore are not directly suited to modeling: the user should not have to directly specify a polynomial that describes the desired object.

Instead, we prefer to use polynomial basis functions combined with user-input control points to create parametric curves. In the next section, we will investigate the earliest such curve.

7.2.1 Bézier Curves

Specifying a curve with a set of control points is far more intuitive for a user than giving a mathematical definition, even for a simple line segment. Consider the following example.

Example 7.4

Given two control points P_0 and P_1, how can we model the line segment connecting them?

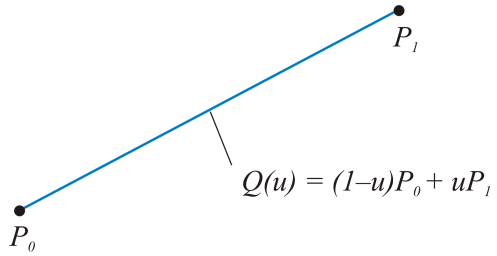


Figure 7.4: A Beziér curve defined over two control points.

Answer:

Let's use a parameter range of $0 \leq u \leq 1$. In this parameter range, we would like $Q(u = 0) = P_0$ and $Q(u = 1) = P_1$. A simple curve that satisfies these constraints and models the line between the points is

$$Q(u) = (1 - u)P_0 + uP_1, \qquad 0 \leq u \leq 1 \; .$$

Let $B_0 = 1 - u$ and $B_1 = u$; these *blending functions* are polynomials of degree 1, and therefore $Q(u)$ is a polynomial parametric curve. The basis functions B_0 and B_1 are said to blend P_0 and P_1.

What happens when the number of control points increases to three? How can we blend between the points and achieve a smooth curve? Consider the following basis functions:

$$\begin{aligned} B_0(u) &= (1 - u)^2 \; , \\ B_1(u) &= 2u(1 - u) \; , \\ B_2(u) &= u^2 \; . \end{aligned}$$

Then, the curve $Q(u)$ is defined as

$$Q(u) = B_0(u)P_0 + B_1(u)P_1 + B_2(u)P_2 \; .$$

Recall that points can only be combined affinely; that is, the weights of a linear combination of points must sum to unity. Does this property hold for B_0, B_1, and B_2?

$$\begin{aligned} B_0 + B_1 + B_2 &= (1 - u)^2 + 2u(1 - u) + u^2 \\ &= 1 - 2u + u^2 + 2u - 2u^2 + u^2 \\ &= 1 \; . \end{aligned}$$

So, the basis functions are indeed affine.

The preceeding examples involving two and three control points are particular instances of a more general formulation for a parametric curve invented by Pierre Bézier in 1972. The formal definition of a **Bézier curve** is:

$$Q(u) = \sum_{i=0}^{d} B_{i,d}(u)P_i, \qquad 0 \leq u \leq 1 \; ,$$

where d is the degree of the resulting curve and

$$B_{i,d} = \tbinom{d}{i} \, u^i (1 - u)^{d-i} \tag{14}$$

are known as the *Berenstein basis functions*.

Example 7.5

What are the basis functions for a 3^{rd}-degree Bézier curve?

Answer:

Here $d = 3$, and by Equation 14 we have:

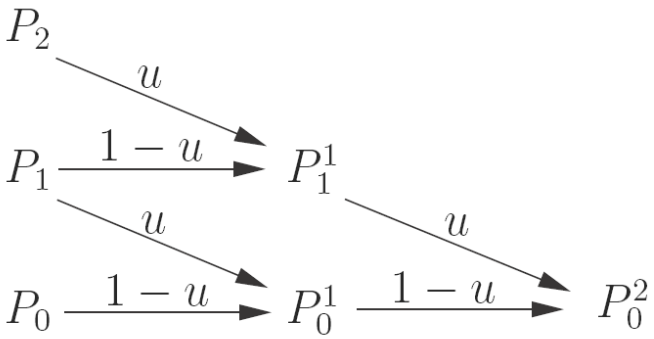
$$\begin{aligned} B_{0,3}(u) &= \binom{3}{0} u^0(1-u)^{3-0} &= (1-u)^3 \\ B_{1,3}(u) &= \binom{3}{1} u^1(1-u)^{3-1} &= 3u(1-u)^2 \\ B_{1,3}(u) &= \binom{3}{2} u^2(1-u)^{3-2} &= 3u^2(1-u) \\ B_{1,3}(u) &= \binom{3}{3} u^3(1-u)^{3-3} &= u^3 \end{aligned}$$

Bézier curves have the following properties:

- The first and last control points are interpolated.
- The direction of the tangent of the curve at $u = 0$ is given by $P_1 - P_0$.
- The direction of the tangent at $u = 1$ is defined by $P_d - P_{d-1}$.
- The degree of the curve is coupled with the number of control points.

7.2.2 de Casteljau Algorithm

Consider the second degree BÃ©zier curve $Q_2(u)$ for a given u and control points P_0, P_1, P_2. The value of the curve, P_0^2, can be found from figure 7.5.

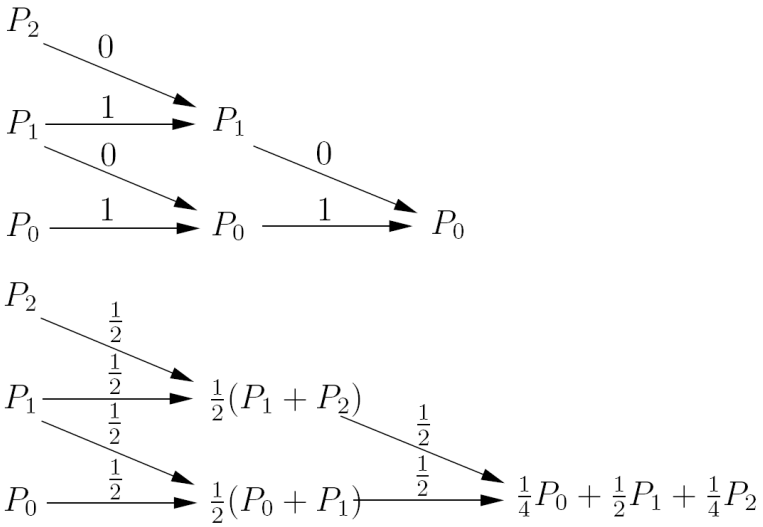


$$P_0^2 = (1-u)P_0^1 + uP_1^1 = (1-u)$$
$$P_0^2 = (1-u)\left[(1-u)P_0 + uP_1\right] + u\left[(1-u)P_1 + uP_2\right]$$
$$P_0^2 = (1-u)^2 P_0 + 2u(1-u)P_1 + u^2 P_2$$

Figure 7.5: de Casteljau algorithm with example of P_0^2

Example 7.6

Find $Q_2(0)$ and $Q_2(\frac{1}{2})$ using de Casteljau algorithm.



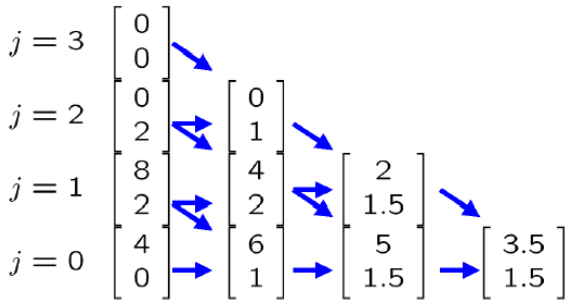
The de Casteljau algorithm is a column-by-column updating algorithm that starts from control points $P_0, P_1, \ldots, P_d$. The rule $P_j^{next} = (1-u)P_j^{curr} + uP_{j+1}^{curr}$ is used to create a new smaller column of control points. After $d-1$ repeats of this rule, P_0 will contain the curve value, $Q(u)$. Algorithm 7.1 shows the pseudo-code for the de Casteljau algorithm.

Algorithm 7.1 de Casteljau algorithm.

```
// input P[j], d, u
// P[j]:  control point, d:  degree, u:  parameter
//output will be Q(u)
for i = 1 to d
   for j = 0 to d-i
      P[j] = (1-u)*P[j] + u*P[j+1]
   end
end
output p[0]
```

Example 7.7

Compute $Q_3(\frac{1}{2})$ given the control points $P_0 = (4,0), P_1 = (8,2), P_2 = (0,2), P_3 = (0,0)$.



7.3 Geometric Interpretation of de Casteljau Algorithm

A more intuitive method of following the de Casteljau algorithm is as follows:

- Draw a new point between P_0 and P_1. This point will be $uP_0 + (1-u)P_1$.
- Draw a new point between P_1 and P_2 such that $P_{new} = uP_1 + (1-u)P_2$.
- Repeat until there is a new point between all the old points.
- Discard all the old points, and repeat the previous steps until one point is left.

- The remaining point is $Q(u)$.

Figure 7.6 shows the step by step process of finding P_0^3 at $u = \frac{1}{2}$.

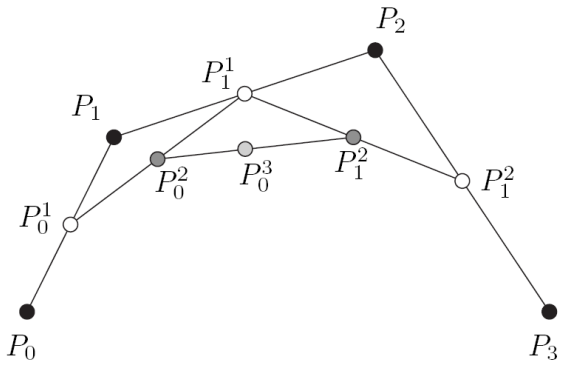


Figure 7.6: Interpretation of de Casteljau algorithm.

7.3.1 Hermite Curves

Recall from the previous section that Bézier curves interpolate the first and last control points, but not necessarily any others. Often it is more intuitive to have all control points interpolated; for some applications it may even be necessary.

Another issue with Bézier curves is that the degree of the curve is determined by the number of control points; for large curves, the degree becomes unreasonably high. An alternative way to build a Bézier curve with many control points is to use several small Béziers and join them together. However, to get a smooth transition between each sub-curve requires considerable effort.

Hermite curves [1] (Figure 7.7) address both issues: given a set of n control points and n corresponding tangent vectors, a Hermite curve will smoothly interpolate each control point P_i, and the tangent of the curve at each control point will match the desired tangent T_i.

A Hermite curve is a piecewise-cubic function. To see how they are constructed, let's focus on a single segment $Q_i(t)$, between P_i and P_{i+1}.

$$Q_i(t) = A + Bt + Ct^2 + Dt^3, \quad 0 \leq t \leq 1 \, .$$

For a 2D curve, the coefficients A, B, C, D will be 2-tuples; in general, for an n-D curve the coefficients will be n-tuples. From the conditions given – the control points and tangents – we can uniquely determine A, B, C, and D. Since $Q_i'(t) = B + 2Ct + 3Dt^2$, we have:

$$\begin{aligned} Q_i(0) = P_i &\implies P_i = A \\ Q_i'(0) = T_i &\implies T_i = B \\ Q_i(1) = P_{i+1} &\implies P_{i+1} = A + B + C + D \\ Q_i'(1) = T_{i+1} &\implies T_{i+1} = B + 2C + 3D \end{aligned}$$

[1] This subject is discussed in the last week of the class

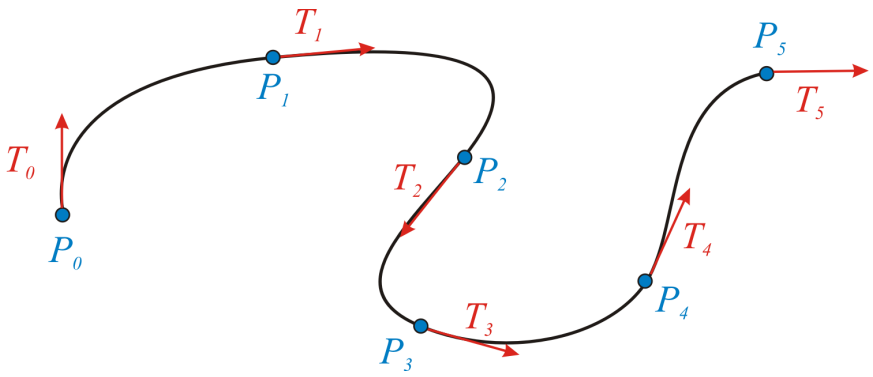


Figure 7.7: A Hermite curve is specified by a set of control points $P_0, \ldots, P_n$ and a corresponding set of tangent vectors $T_0, \ldots, T_n$.

Rearranging the equations to isolate the variables, we find:

$$\begin{aligned} A &= P_i \\ B &= T_i \\ C &= 3(P_{i+1} - P_i) - 2T_i - T_{i+1} \\ D &= 2(P_i - P_{i+1}) + T_i + T_{i+1} \end{aligned}$$

Example 7.8

Given $P_0 = (0, 0)$, $P_1 = (2, 0)$, $T_0 = (0, 1)$, and $T_1 = (0, -1)$, as in Figure 7.8, what is the equation for the Hermite segment $Q(t)$?

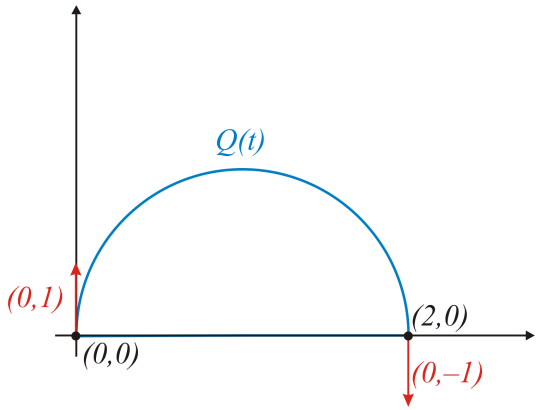


Figure 7.8: A segment of a Hermite curve.

Answer:

$$
\begin{aligned}
A &= P_0 \\
&= (0,0) \\
B &= T_0 \\
&= (0,1) \\
C &= 3(P_1 - P_0) - 2T_0 - T_1 \\
&= (6,-1) \\
D &= 2(P_0 - P_1) + T_0 + T_1 \\
&= (-4,0)
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
Q(t) &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} t + \begin{bmatrix} 6 \\ -1 \end{bmatrix} t^2 + \begin{bmatrix} -4 \\ 0 \end{bmatrix} t^3 \\
&= \begin{bmatrix} 6t^2 - 4t^3 \\ t - t^2 \end{bmatrix}
\end{aligned}
$$

For the general case, we can treat each segment independently; the final curve will be smooth because the tangent at each joint will match. For each segment, the parameter range will be $0 \leq t \leq 1$, so the parameter will have to be "reset" at each segment. Alternatively, each segment could be constructed with a different parameter range such that the whole curve is parameterized over the range $[0, 1]$.

For user interaction, there are a couple of ways that the tangent vectors T_i can be specified. With some suitable user interface, we could choose to have the users specify the control points *and* the tangent vectors. Or, the user could give only the control points, from which the application automatically determines the tangent vectors.

Automatic Tangent Vectors

Given only a set of control points $P_0, \ldots, P_n$, we can choose a set of suitable tangent vectors as follows. To begin, set the tangent vector T_0 to be the difference vector between the first and second control points; similarly for the last tangent vector.

$$
\begin{aligned}
T_0 &= P_1 - P_0 \\
T_n &= P_n - P_{n-1}
\end{aligned}
$$

For the remaining tangent vectors T_i, we can use the average of two difference vectors, from P_{i-1} to P_i and from P_i to P_{i+1}; see Figure 7.9.

7.4 Subdivision Curves

Properties of an Ideal Curve

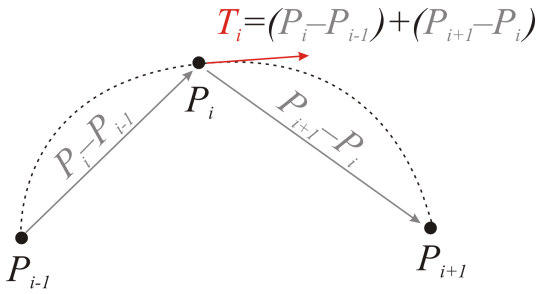


Figure 7.9: The tangent vectors for a Hermite curve can be automatically computed from a set of control points.

We would like to have a curve that is defined as sum of *blending functions* $B_{i,k}(u)$ given the control points $P_0, P_1, \ldots, P_d$:

$$Q(u) = \sum_{i=0}^{d} P_i B_{i,d}(u) \quad a \leq u \leq b.$$

The properties of the curve $Q(u)$ are the direct result of the blending functions $B_{i,k}(u)$. What would be the properties of an ideal basis function then?

- Easy and efficient computation.

  To generate curves rapidly, we require computationally simple and stable basis functions. These considerations lead us to choose low degree polynomials.

- Sum to unity at every u.

  The basis functions must sum to 1 at u so that the set of control points can be affinely combined. We require linear combination of control points to have invariance under affine transformations.

- Local support for $a \leq u \leq b$.

  In order to have the local control property for the curve, the basis functions should only be active (non-zero) within a small range. Polynomial functions cannot have local support because a polynomial of degree d is zero in at most $d + 1$ distinct points. Functions with local support must be zero at infinitely many points.

- Sufficient smoothness.

  The smoothness of curve $Q(u)$ depends on the smoothness of the basis functions. For example, if every basis function is $\mathbf{C}^1$ in $[a, b]$, then $Q(u)$ will also be $\mathbf{C}^1$ in the range $[a, b]$.

While polynomials do not satisfy our wish-list (why?), *piecewise polynomial* functions do. Figure 7.10 shows several examples of functions that are piecewise polynomial. Each function consists of several *polynomial segment*s. The points at which a pair of individual segments meet are called *joints*. The value of u at which two segments meet is called a *knot*.

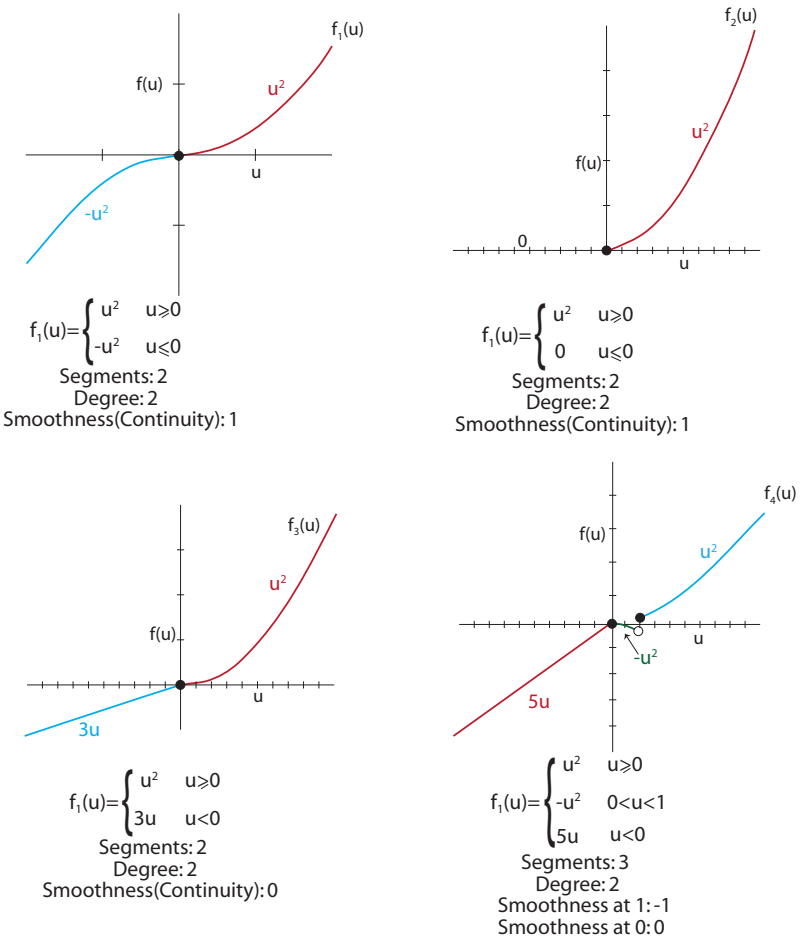


Figure 7.10: Piecewise polynomial functions.

An important subclass of piecewise polynomial functions are known as *splines*. Given a knot sequence, a nth-degree spline function is defined as a piecewise polynomial of degree n that is $(n-1)$ smooth (continuous) at each knot. Figure 7.11 shows a spline functions and its components.

B-splines form the basis functions for splines and satisfy all the desired properties. However, their proper definition is beyond the scope of this course. Nonetheless, there are simple and efficient subdivision methods for creating and rendering B-spline curves and surfaces. In this course, we will cover some of these methods.

Subdivision techniques provide an alternative to parametric curves and surfaces. Parametric curves, such as Bézier curves, are typically defined by a set of control points that are affinely combined according to some basis or blending functions. The evaluation of the basis functions over a large number of parameter samples is a relatively expensive operation, making the rendering and manipulation of parametric models less than ideal for real-time graphics applications.

Subdivision curves and surfaces are defined by a set of control points, just as their parametric counterparts. However, the control points are treated as a representation of the surface itself and rendered directly, rather than being considered as coefficients to a set of basis functions.

Consider Figure 7.12. The initial control mesh in (a) could be used as the control points for a

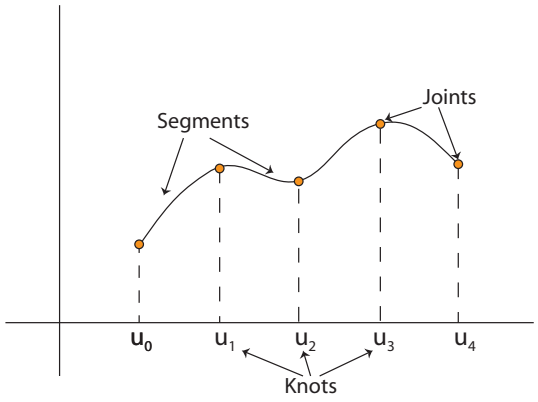


Figure 7.11: A spline function.

parametric surface, but in subdivision we consider it to be a low-resolution representation of the surface itself. One application of subdivision (here, bicubic B-spline subdivision is used) increases the number of points in the mesh as shown in (b), such that the new mesh is a smoother representation of the surface but at the same time, if the mesh is used to evaluate a parametric surface, it defines the exact same surface as the original control mesh. In (c) we see the result of another application of subdivision, and again in (d). After only three applications of subdivision, we can render the mesh directly as in (e), and have a very good representation of the parametric surface defined by the original mesh, but with no need to evaluate any parametric definition to render it.

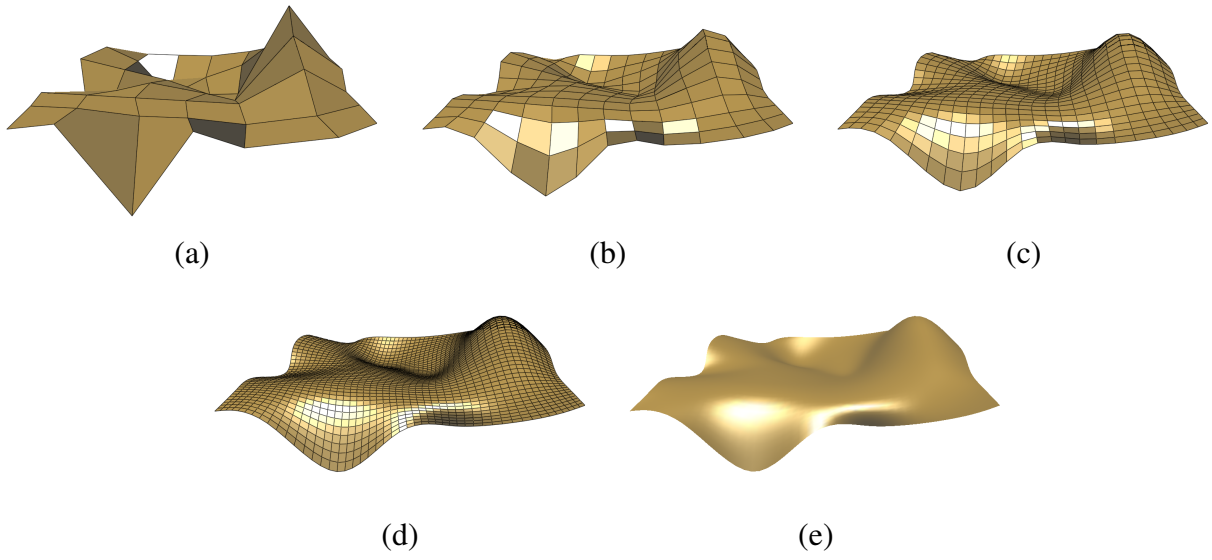


Figure 7.12: A bicubic B-spline subdivision surface: (a) the control mesh; (b) after 1 iteration; (c) 2 iterations; (d) 3 iterations; (e) shaded rendering of (d).

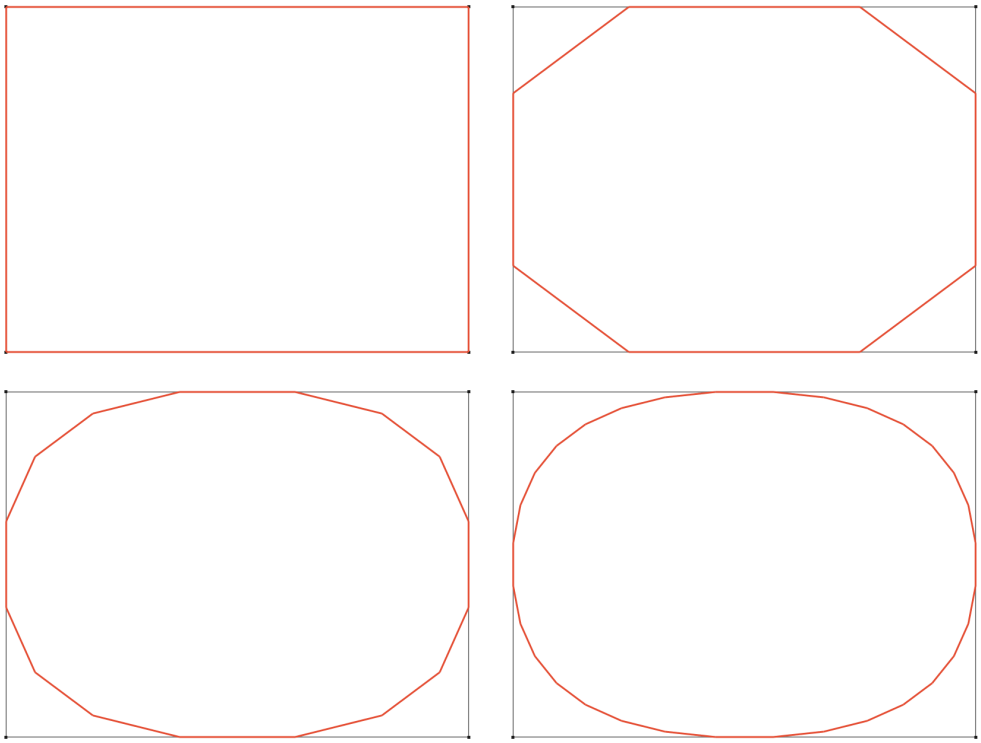


Figure 7.13: A few iterations of Chaikin subdivision on a simple control polygon: (*left to right, top to bottom*): the control polygon; after 1 iteration; 2 iterations; after 3 iterations, the polyline is a reasonable representation of the limit curve.

7.4.1 Chaikin

Chaikin subdivision is a subdivision scheme for curves or surfaces that produces objects with C^1 continuity. A subdivision scheme is defined by a set of *masks* or *filter values*; a mask simply defines how to affinely combine points in the control mesh to create the new control mesh.

For Chaikin subdivision, the subdivision mask is compactly expressed as $S = \{\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}\}$. What this means is that, starting with a set of coarse control points $c_0, \ldots, c_n$, each point c_i is replaced by two finer control points f_{2i} and f_{2i+1}:

$$\begin{aligned} f_{2i} &= \frac{3}{4}c_i + \frac{1}{4}c_{i+1} \\ f_{2i+1} &= \frac{1}{4}c_i + \frac{3}{4}c_{i+1} \end{aligned}$$

See Figure 7.13. Chaikin's scheme is often referred to as a *corner-cutting* scheme.

Subdivision is a linear operation, and so it is often useful to consider the matrix form of the subdivision operation. Of course, subdivision is also a *local* operation (each fine point depends on only a small number of coarse points) and so it is never implemented with $O(n^2)$ matrix operations.

A subdivision mask, such as $S = \{\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}\}$ for Chaikin, represents the non-zero entries in a regular column of its corresponding matrix form. Each regular column is a shifted version of

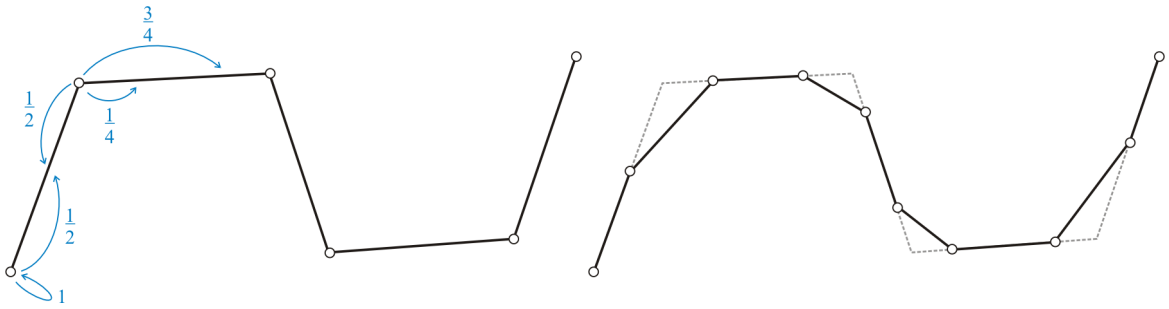


Figure 7.14: The non-periodic version of Chaikin passes through each endpoint of the polyline by using special masks at the beginning and end.

the previous column. For instance, the subdivision matrix that creates 8 fine points from 4 coarse vertices would be:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix} = \begin{bmatrix} \frac{3}{4} & \frac{1}{4} & 0 & 0 \\ \frac{1}{4} & \frac{3}{4} & 0 & 0 \\ 0 & \frac{3}{4} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & \frac{3}{4} & 0 \\ 0 & 0 & \frac{3}{4} & \frac{1}{4} \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} \\ \frac{1}{4} & 0 & 0 & \frac{3}{4} \\ \frac{3}{4} & 0 & 0 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

The Chaikin curve shown in Figure 7.13 is known as a *periodic* or *closed* curve, because the control points are treated as a polygon (no beginning or end) rather than a polyline.

There are special masks to handle non-periodic or *open* curves; see Figure 7.14. In matrix form, an open curve differs from the closed version only in the first two and last two columns

$$\begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{2n-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & & & \\ 0 & \frac{3}{4} & \frac{1}{4} & 0 & & & \\ 0 & \frac{1}{4} & \frac{3}{4} & 0 & & & \\ 0 & 0 & \frac{3}{4} & \frac{1}{4} & & & \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} & & & \\ & & & & \ddots & & \\ & & & & & \frac{1}{4} & \frac{3}{4} & 0 \\ & & & & & 0 & \frac{1}{2} & \frac{1}{2} \\ & & & & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}$$

As mentioned before, the matrix representation of subdivision is inefficient from an implementation standpoint: matrix-vector multiplicatoin is in $O(n^2)$. Because subdivision is a local operation, it can be easily implemented in $O(n)$. Algorithm 7.3 provides some pseudo-code for the implementation of open-curve Chaikin subdivision.

Algorithm 7.2 Chaikin subdivision for open curves.

```
/*
 * Input: coarse points C[], number of points N
 * Output: fine points F[]
 */
function Chaikin (Point C[], int N)

   // Special mask at the beginning
   F[0] = C[0]
   F[1] = 0.5*C[0] + 0.5*C[1]

   // Periodic mask for the interior points
   for i = 1 to N-2
      F[2*i  ] = 0.75*C[i] + 0.25*C[i+1]
      F[2*i+1] = 0.25*C[i] + 0.75*C[i+1]

   // Special mask at the end
   F[2*N-2] = 0.5*C[N-2] + 0.5*C[N-1]
   F[2*N-1] = C[N-1]

   return F[]
```

7.4.2 Cubic B-Spline

Chaikin subdivision is based on second-degree B-spline parametric curves. There are similar subdivision schemes for higher-degree B-splines, the most popular of which is the cubic (degree 3) case.

The subdivision mask for cubic B-spline subdivision is $S\{\frac{1}{8}, \frac{1}{2}, \frac{3}{4}, \frac{1}{2}, \frac{1}{8}\}$. This can be equivalently expressed in terms of the even and odd fine points:

$$\begin{aligned} f_{2i} &= \frac{1}{8}c_{i-1} + \frac{3}{4}c_i + \frac{1}{8}c_{i+1} \\ f_{2i+1} &= \frac{1}{2}c_i + \frac{1}{2}c_{i+1}\,. \end{aligned}$$

See Figure 7.15 for an example of this scheme being applied to a simple curve.

As in the case of Chaikin, there are different masks that must be used for non-periodic curves. For the cubic case, there are three non-regular columns at the beginning and the end of the subdivision matrix; in general, degree d B-spline subdivision schemes will have d non-regular columns.

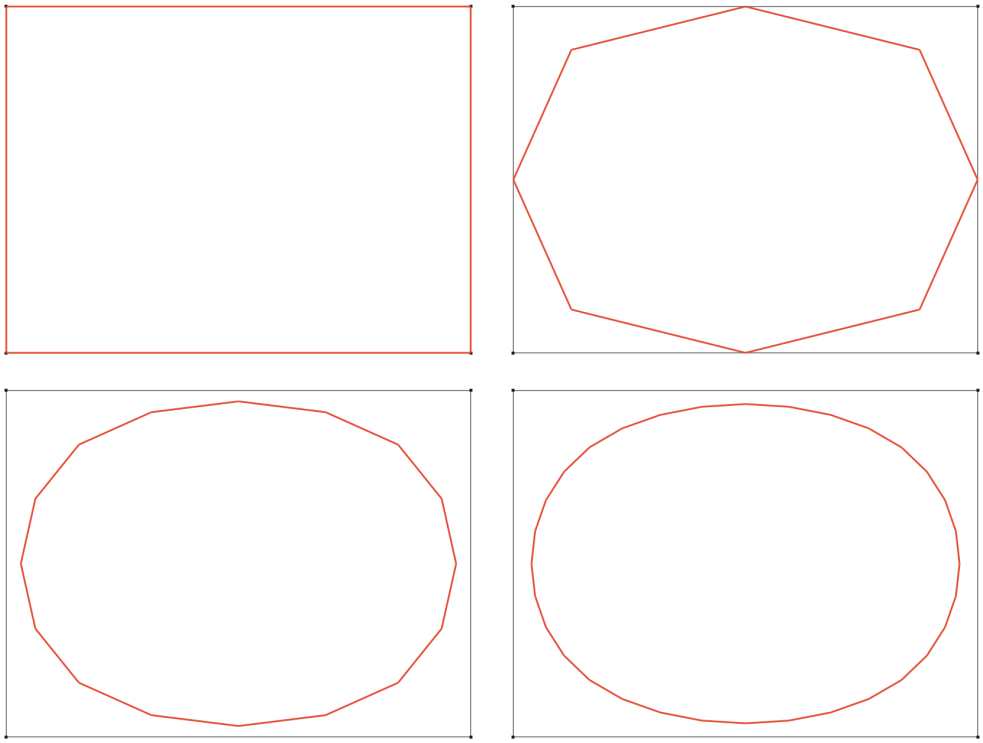


Figure 7.15: A few iterations of cubic B-spline subdivision on a simple control polygon: (*left to right, top to bottom*): the control polygon; after 1 iteration; 2 iterations; after 3 iterations, the polyline is a reasonable representation of the limit curve.

The general form of the matrix for open cubic B-spline subdivision curves is:

$$\begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{2n-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & \cdots \\ 0 & \frac{3}{4} & \frac{1}{4} & 0 & 0 & 0 & \cdots \\ 0 & \frac{3}{16} & \frac{11}{16} & \frac{1}{8} & 0 & 0 & \cdots \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & \cdots \\ 0 & 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 & \cdots \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & \cdots \\ 0 & 0 & 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & \cdots \\ & & & \vdots & & & \\ \cdots & 0 & 0 & \frac{1}{8} & \frac{11}{16} & \frac{3}{16} & 0 \\ \cdots & 0 & 0 & 0 & \frac{1}{4} & \frac{3}{4} & 0 \\ \cdots & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \cdots & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

7.5 Subdivision Surfaces

A great benefit of subdivision curves (and parametric curves, for that matter) is that they can trivially be extended to modeling of higher-dimensional objects.

The most common extension is to 2 dimensions, for the modeling of surfaces. The type of surfaces which we consider in this course, is called tensor product surfaces that can be covered by two sets of curves (denoted by u and v curves). A tensor-product subdivision surface is modeled by an array of control points, as in Figure 7.12(a). The critical aspect of the control points is that there is a regular arrangement, meaning each row has the same number of control points. This regularity provides an implicit structure for modeling the actual surface as quadrilateral faces.

To apply a curve-based subdivision scheme to a 2D grid of control points, we simply treat each row and column as an isolated curve. So, each row is subdivided, created a new grid of control points with the same number of rows but twice the number of columns. Then, each column is subdivided, resulting in a new grid with twice the rows and columns of the original grid. Figure 7.12 shows a few iterations of this process.

7.5.1 Parametric Surfaces

For surfaces, the parameter space becomes two dimensional, thus we use two parameters. One parameter can only cover a line, for a surface we must have a covering of 2D domains (two parameters) as demonstrated in the following figure.

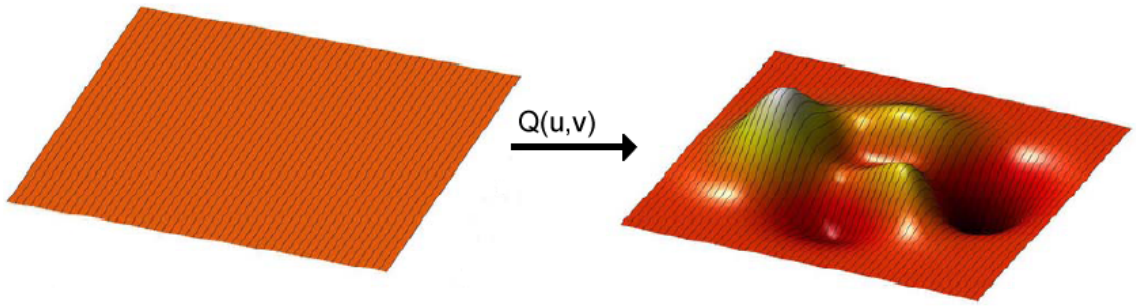


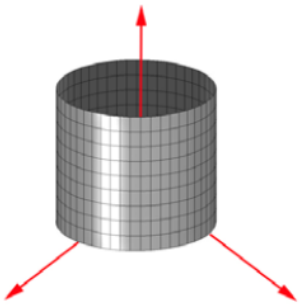
Figure 7.16: Paremetric surface is a function of a 2D domain(Left: usually a rectangle) to 3D space(Right).

$$Q(u,v) \ : \ A \Rightarrow \mathbf{E}^3, \ A \subseteq \mathbf{E}^2$$
$$Q(u,v) = \begin{bmatrix} X(u,v) \\ Y(u,v) \\ Z(u,v) \end{bmatrix} \quad (u,v) \in A$$

Example 7.9

A paremtric cylinder cab be defined as $Q(u,v) = \begin{bmatrix} \cos u \\ \sin u \\ v \end{bmatrix} \quad 0 \leq u < 2\pi, \ 0 \leq v < 1.$ As

demonstrated in the figure, surface is covered by two sets of curves: u-curves are circle and v-curves are straight vertical lines.



To display the cylinder or any other parametric surface we must, as with parametric curves, discretize our parametric domain. We choose a discrete sample of our parameters (usually in a uniform way). In our cylinder example we sampled 10 values for v from $[0, 1]$ and 50 values for u from $[0, 2\pi]$. For each u and v we have a 3D point on the surface. These points can then be used to polygonize the surface.

Algorithm 7.3 Discretized sample of a surface's parameter space.

```
for u=0 to 2*Pi step 0.1
   for v=0 to 1 step 0.1
     current_point= Q(u,v)
   endfor
endfor
```

Having $Q(u, v)$ we can define two important sets of embedded curves. By letting v vary while keeping u constant we generate a curve called a v-curve. Similarly, letting u vary while hold v constant produces a u-curve.

Figure 7.17: A visualization of v-curves on a parametric surface.

7.5.2 Curve Based Surfaces(Common Surfaces)

It is possible to create parametric surfaces using two approaches. One way is to use a grid of control points as discussed and demonstrated in Figure 7.12. This has a parametric representation too which is beyond of this course's targets. The other general approach is to use curves for defining parametric surfaces. Although, this approach can only create a limited sets of surfaces, it is easy and also more natural. Creating surfaces from the curves is not unique and there are several methods such as ruled surface, coons patch, sweep surface and surface of rotation. In this course, we just see surface of revolution as an important example.

7.5.3 Surfaces of Revolution

To create a surface of revolution we perform a rotation sweep of the curve. Figure 7.18 gives some examples of surfaces of revolution and their respective curves. So we take a two dimensional curve $C(v) = (x(v), z(v))$, which could be B-spline or Bezier curve, and we rotate it around the z axis. Then we simply form the surface as $Q(u, v) = (x(v)\cos u, x(v)\sin u, z(v)) \quad 0 \leq u \leq 2\pi$ (or a subrange such as $0 \leq u \leq \pi$), and the range of v is inherited from the curve. Notice in this description the axis of rotation is z axis. We can use this description as a generic form for surface of revolution and transform any other desired axis to the generic form using some rotations and translation.

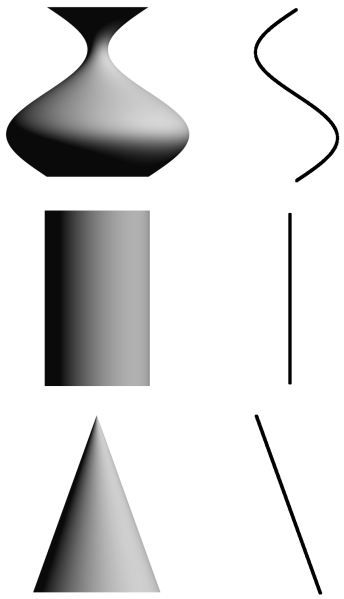
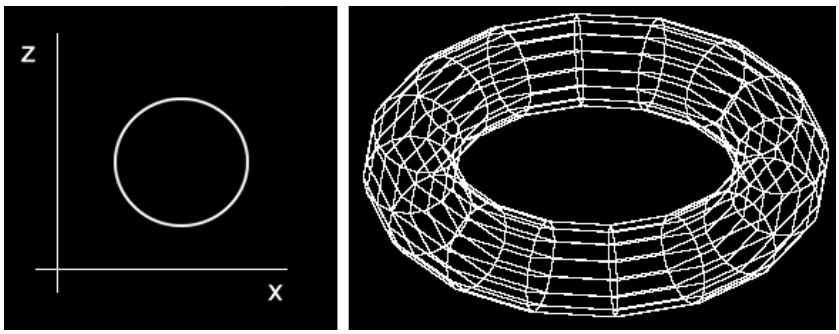


Figure 7.18: Surfaces of revolution and their generating curves.

Example 7.10
A Torus

$$C(v) = \begin{bmatrix} \cos v \\ \sin v \end{bmatrix} \quad Q(u, v) = \begin{bmatrix} \cos v \cos u \\ \cos v \sin u \\ \sin v \end{bmatrix}$$



7.6 Bézier Surface

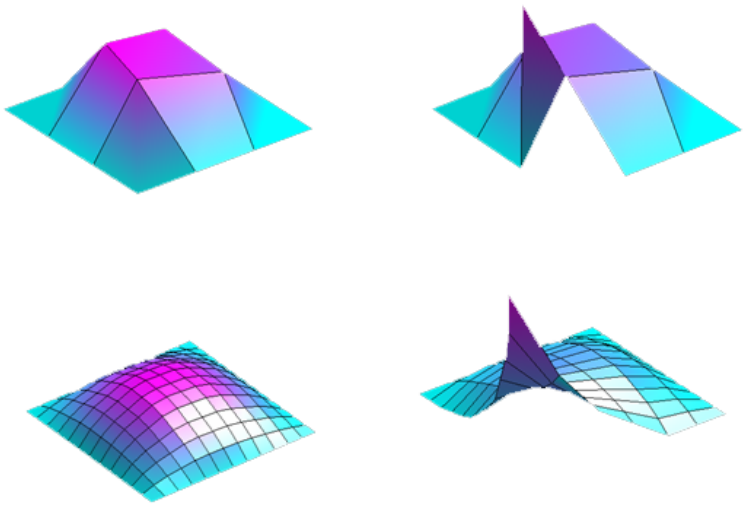


Figure 7.19: A Bézier surface.

If, instead of B-spline basis functions, we use Bézier basis functions in the above definition, then the obtained parametric surface is called a Bézier surface or patch, and its u- and v-curves are Bézier curves. Given $P_{i,j}$, a net of control points, we obtain a surface with the following equation:

$$Q(u,v) = \sum_{i=0}^{d} \sum_{j=0}^{d'} P_{i,j} B_{i,d}(u) B_{j,d'}(v) \qquad 0 \le u \le 1 \qquad 0 \le v \le 1.$$

Example 7.11

3^{th}-degree Bézier patch

$$Q(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} P_{i,j}B_{i,3}(u)B_{j,3}(v) \qquad 0 \le u \le 1 \qquad 0 \le v \le 1$$

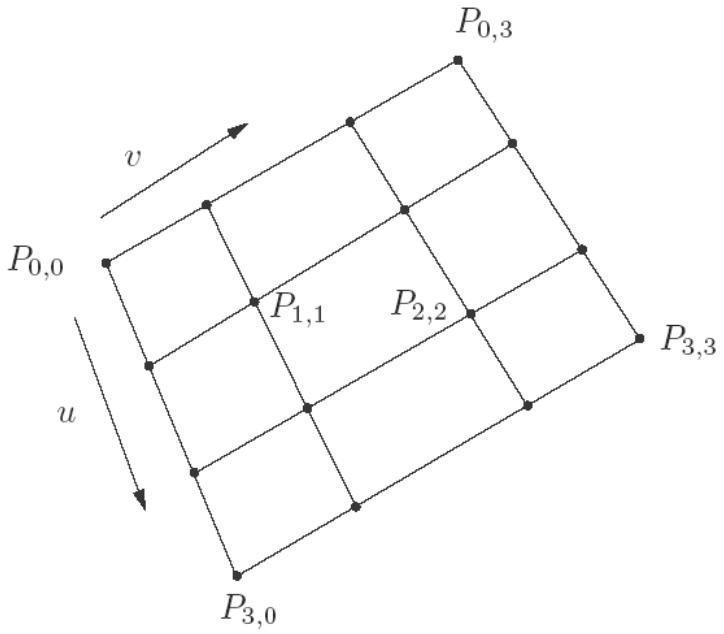


Figure 7.20: A 3^{rd} degree Bézier patch

7.7 Normals

For smooth surfaces, the normal is the unit vector, which is perpendicular to the surface at any given point on the surface. This vector is important in the shading of objects. The normal of a parametric surface is calculated from the tangents of the u- and v-curves:

- $T_u = \frac{\partial Q}{\partial u}$ is the tangent vector in the u direction,
- $T_v = \frac{\partial Q}{\partial v}$ is the tangent vector in the v direction.

Blue vectors show these tangents in the figure. The normal to the surface is defined by the normal to the tangent plane formed by T_u, T_v and the point $Q(u, v)$ (see the figure). Therefore, we can find the normal by taking the cross-product of the tangents:

$$N(u, v) = \frac{\partial Q}{\partial u} \times \frac{\partial Q}{\partial v}.$$

Example 7.12
Find the normal on the surface of a cylinder.

$$Q(u, v) = \begin{bmatrix} \cos u \\ \sin u \\ v \end{bmatrix} \qquad \begin{matrix} 0 \leq u \leq 2\pi \\ 0 \leq v \leq 1 \end{matrix}$$

$$\frac{\partial Q}{\partial u}(u, v) = \begin{bmatrix} -\sin u \\ \cos u \\ 0 \end{bmatrix} \qquad \frac{\partial Q}{\partial v}(u, v) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$N(u, v) = \begin{vmatrix} -\sin u & \cos u & 0 \\ 0 & 0 & 1 \\ i & j & k \end{vmatrix} = i \cos u + j \sin u + 0k$$

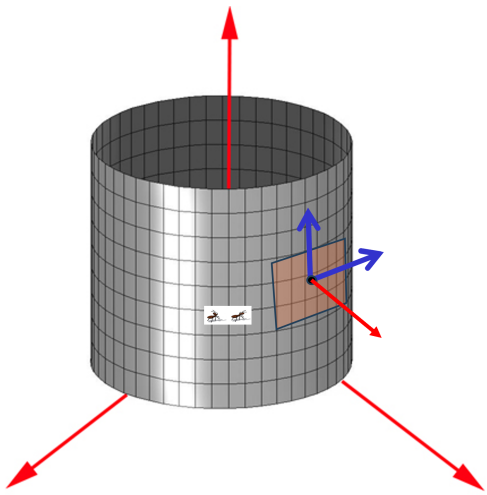


Figure 7.21: The normal to the surfaces can be found by the normal to the tangent plane.

---

8 Polygonal Meshes

Parametric surfaces provide a useful method to represent graphical objects. However, to generate a greater variety of objects, we need a different approach to represent general topological surfaces. *Polygonal meshes* can provide such an approach, in which the object is represented as a set of polygons that are attached to each other. Formally, a mesh M consists of a set of vertices V, which captures the geometry (locations) of discrete points on the object, and a set of faces denoted by F that represents the connectivity of the object. A face $f_i \in F$ connects n vertices $v_{i_0}, v_{i_1}, ..., v_{i_{n-1}}$ in which $v_{i_k} \in V$. If all faces connect only three vertices, it is called a *triangular* mesh. Another special case is that of a *quad* mesh, where all faces connect exactly four vertices.

In summary, a mesh M can be represented as the tuple $M = (V, F)$. However, to handle connectivity queries for the mesh, we typically also work with the set of edges that connect vertices. The set of all edges of M is denoted by E, and an edge $e_i \in E$ connects exactly two vertices v_{i_0} and v_{i_1} (again, $v_{i_k} \in V$). Figure 8.1 illustrates the set of vertices, edges and faces of a mesh that represents a bunny.

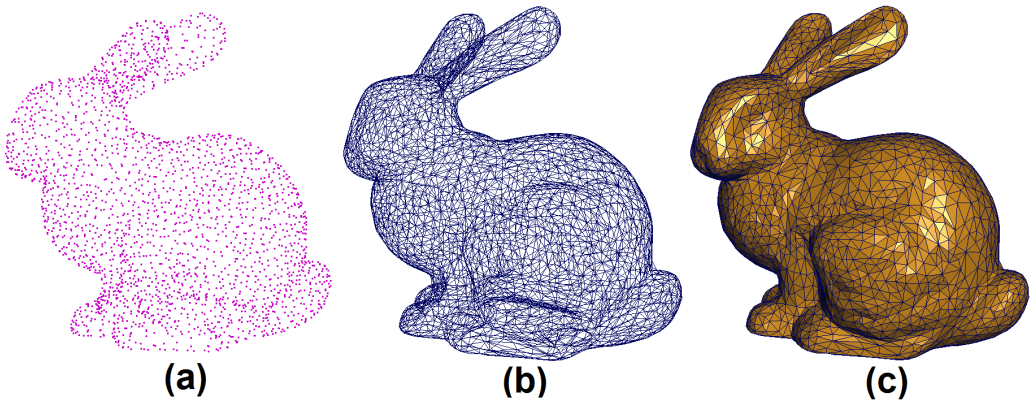


Figure 8.1: (a) The set of vertices, V. (b) The set of edges, E. (c) The set of faces, F.

8.1 Neighborhood

Each edge e in E connects two vertices v_i and v_j (Figure 8.2 (a)). These two vertices are called neighbors. In general, the set of all neighbors of vertex v_i is called its neighborhood and is denoted by $N(v_i)$ (Figure 8.2 (b)). $|N(v_i)|$ refers to the number of vertices in the neighborhood of v_i, or its *valence*. For instance, in Figure 8.2, the valence of v_i is seven, as it has seven vertices in its neighborhood. Two faces f_i and f_j are neighbors if they share an edge e (Figure 8.2 (a)).

8.2 Basic Operations

To manipulate the mesh, there are several basic operations that may be used to change the topology, connectivity, or geometry of the object. For instance, we can pick a point and change its position, or we can split an edge by inserting a vertex in the middle which is then connected to some nearby vertices (see Figure 8.3).

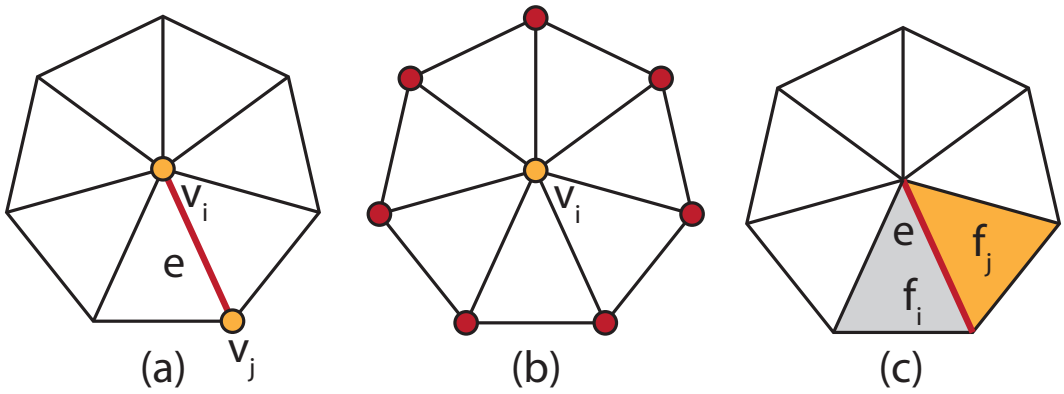


Figure 8.2: (a) v_i and v_j are neighbors. (b) Neighbors of v_i are drawn in red. (c) Faces f_i and f_j are neighbors.

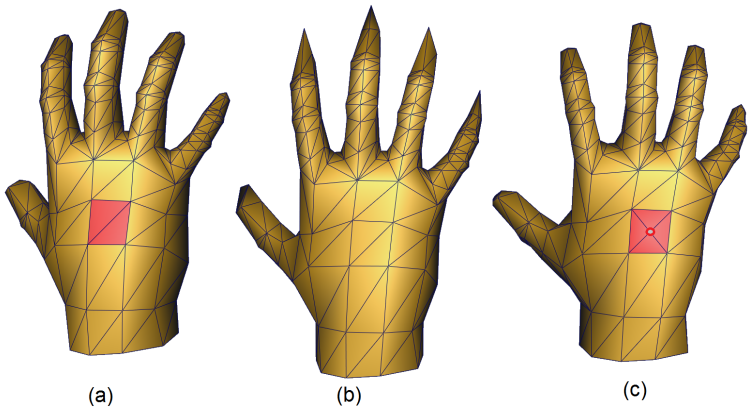


Figure 8.3: Basic modifications to a mesh. (a) A region of the mesh is selected. (b) Vertex repositioning. (c) Vertex and edge insertion.

8.3 Finding Normals

As we discussed in the previous chapter, for smooth surfaces, the normal is the unit vector perpendicular to the surface. For meshes, we also need to define and calculate normals. This can be done by finding a constant normal vector for each face. Suppose we want to find the normal vector n at vertex P_1 of the face P_0, P_1, P_2, as shown in Figure 8.4, using the cross product of $(P_2 - P_1)$ and $(P_0 - P_1)$.

Notice that if a face has more than three vertices, it might be non-planar. Consequently, the method mentioned above can produce different normals depending on which three vertices are used. For example, in Figure 8.4 (right), we have two normals at P_2 depending on the triangle used to define the normal. In this case, the average of all possible normals can be used as the normal of the face. Another strategy is to initially split the face into a set of triangles.

Using a constant normal vector per face produces *flat shading*, as shown in Figure 8.5 (b). For most applications, this kind of shading is not very desirable. We can generate a smooth result (see Figure 8.5 (a)) by defining a normal per vertex. One way to determine the normal of a vertex is to calculate the normal of each face connected to the vertex and then average them. That is, if faces f_0 to f_n are connected to vertex v, the normal vector of v denoted by N_v can be taken to be

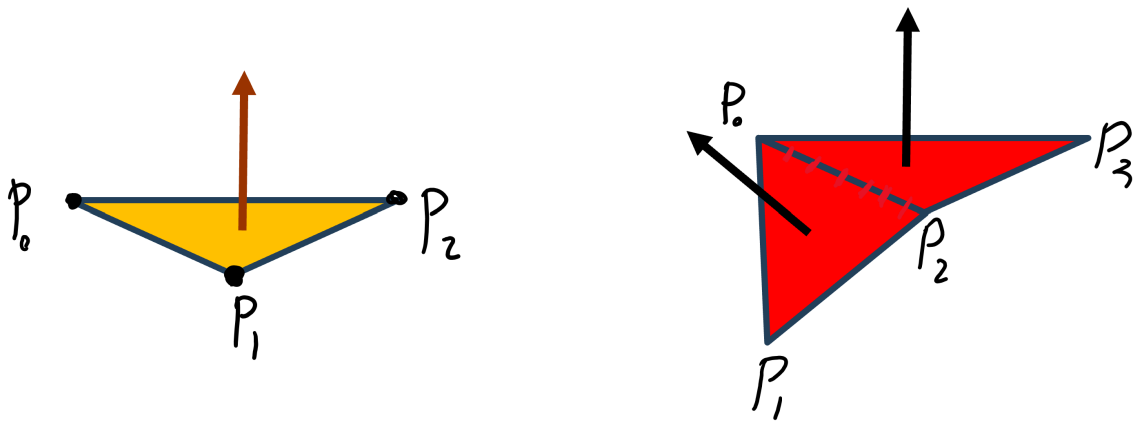


Figure 8.4: Face normals for triangles and quads

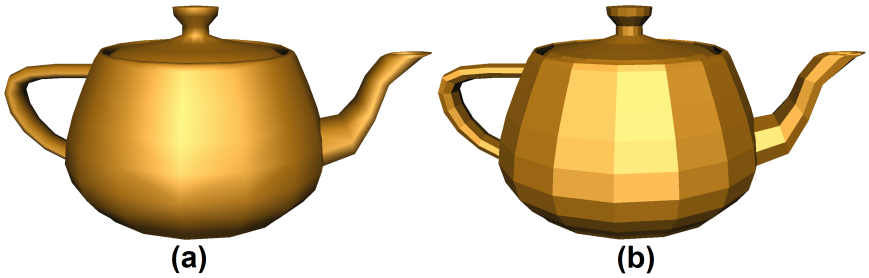


Figure 8.5: (a) Smooth shading. (b) Flat shading.

$N_v = \frac{\sum_{i=0}^{n} N_{f_i}}{|\sum_{i=0}^{n} N_{f_i}|}$ in which N_{f_i} is the normal vector of face f_i (see Figure 8.6).

However, computing vertex normals in this way disregards the different sizes of the faces connected to the vertex. It makes more sense to have a face with a larger area contribute more towards the vertex normals. Therefore, we can use a weighted average to compute the normal vectors for vertices. Suppose faces f_0 to f_n with areas a_0 to a_n are connected to vertex v. Using a weighted average, we can calculate the normal vector of v as $N_v = \frac{\sum_{i=0}^{n} a_i N_{f_i}}{|\sum_{i=0}^{n} a_i N_{f_i}|}$.

This method has a flaw because the area of the faces depends on the lengths of the triangle edges. One triangle can have larger edges, and another can have shorter edges, while their local contributions to the normal at v should be independent of the edge size. A better strategy is to use the angles of the triangles as the weights for the weighted average. In this method, we first calculate each face's angle α_i at the vertex v and then use that angle as the weight or contribution of that face (see Figure 8.7). Therefore, the normal at v can be computed as:

$$N_v = \frac{\sum_{i=0}^{n} \alpha_i N_{f_i}}{|\sum_{i=0}^{n} \alpha_i N_{f_i}|}$$

where $\alpha_i = \arccos(\frac{V_i \cdot V_{i+1}}{|V_i \cdot V_{i+1}|})$.

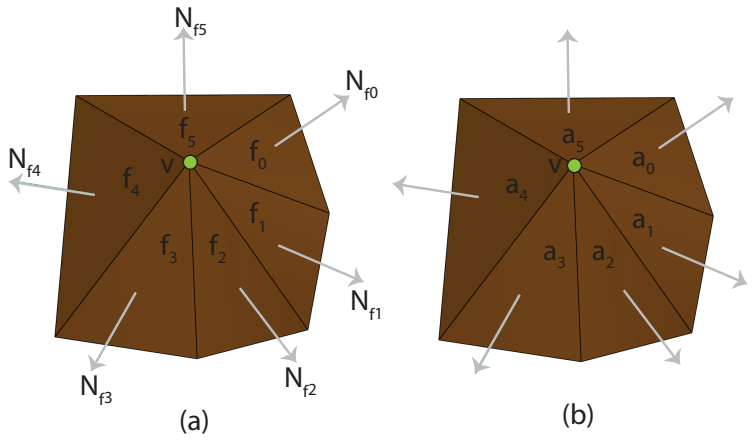


Figure 8.6: (a) Faces connected to vertex v and their normals. (b) Areas of the faces in (a).

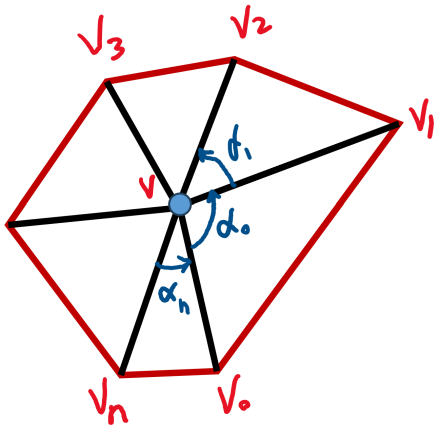


Figure 8.7: Faces connected to vertex v with various angles.

8.4 Face-Vertex List representation

This simple representation is the essence of some modern 3D formats and data-structures such as obj format from **Alias/Wavefront** and **MD2** format from **id** software. Fundamentally, it uses the following lists

- vertex list: reports the locations of the distinct vertices (positional or geometrical information)
- face list: indexes into the vertex list (connectivity or topological information).

The face list records the indices of the adjacent vertices by traversing them. The direction of this traverse is important for finding the normals to the faces which are used for shading. Figure 8.8 shows a quad face with its adjacent vertices. There are two ways to traverse a polygon

- clockwise (5,8,7,6)

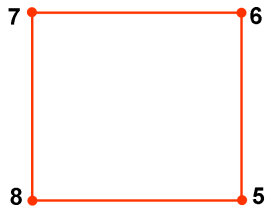


Figure 8.8: A simple quad face example

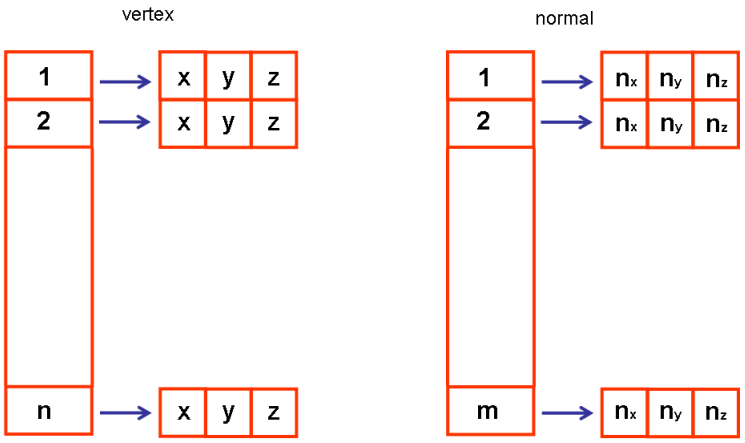


Figure 8.9: Vertex and normal list.

- counter-clockwise (5,6,7,8)

Either direction could be used, but the following convention is usually employed.
Right hand rule: Traverse the polygon counter-clockwise as seen from outside the object (or front side).

Because finding the normal is straightforward, the normal list is an optional structure for some 3d formats.

8.5 Half-edge data structure

Face-vertex list representation is efficient for OpenGL that needs a sequence of faces. However, it is hard to do operations such as followings

1. How can we add a certain vertex into the mesh?
2. How can we remove a certain vertex from the mesh?

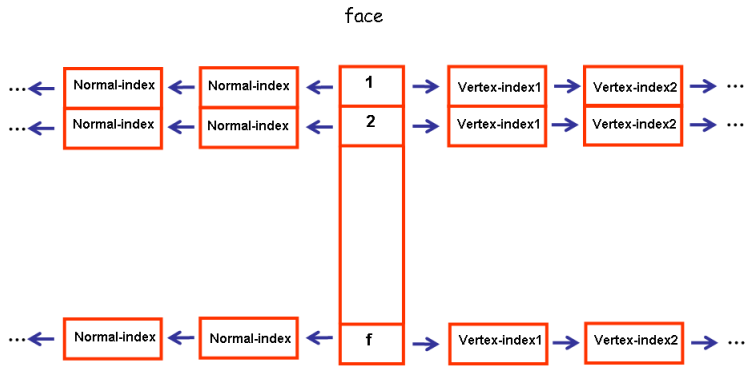


Figure 8.10: Face list, points to the vertex and normal lists.

3. How can we remove a certain face from the mesh?

For instance we need to traverse entire mesh to add just one vertex to a specific portion of the mesh. These kinds of operations are usually needed in mesh processing and editing (mesh surgery). Consequently, we need to use a data structure that allows us to traverse all neighbors of each vertex in an efficient way. There are several data structures that provide an efficient representation.

Half-edge is a very efficient data structure for meshes. In this data structure edges are first class citizens!! All connectivity information assign to the edges. Faces and vertexes have just pointer to one incident edge. Vertices have also a pointer to the 3D coordinates or perhaps normal vectors. For each edge there are two ways of traversing even using counter-clockwise direction. To reduce the symmetric ambiguity, each edge is split to half-edges and the necessary information is divided and attached to each half-edge separately as illustrated in Figure 8.11. A simple example is shown in Figure 8.12.

Initialization of the half-edge data structure is more complicated than Face-vertex list method. However, all necessary mesh processing operations can be done efficiently. As an example , the algorithm 8.5 shows how efficient we can find all the neighbors of a given vertex. This is a basic operation which is needed for inserting a new vertex or removing a vertex and etc. Let n be the number of vertices adjacent to the input vertex v. The loop in the algorithm is terminated at most after n iterations which is optimal for finding the neighbors.

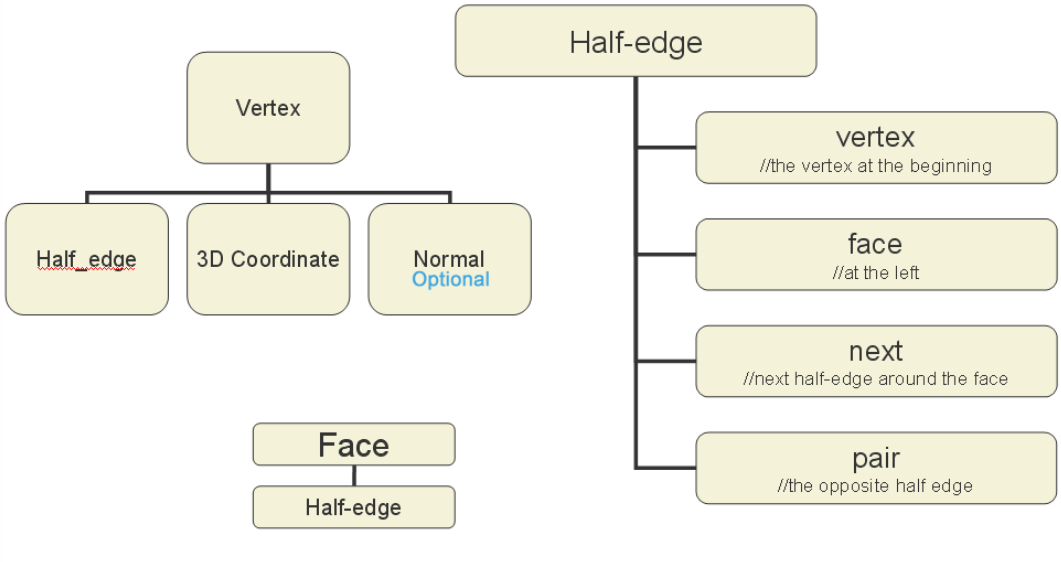


Figure 8.11: Half-edge data structure.

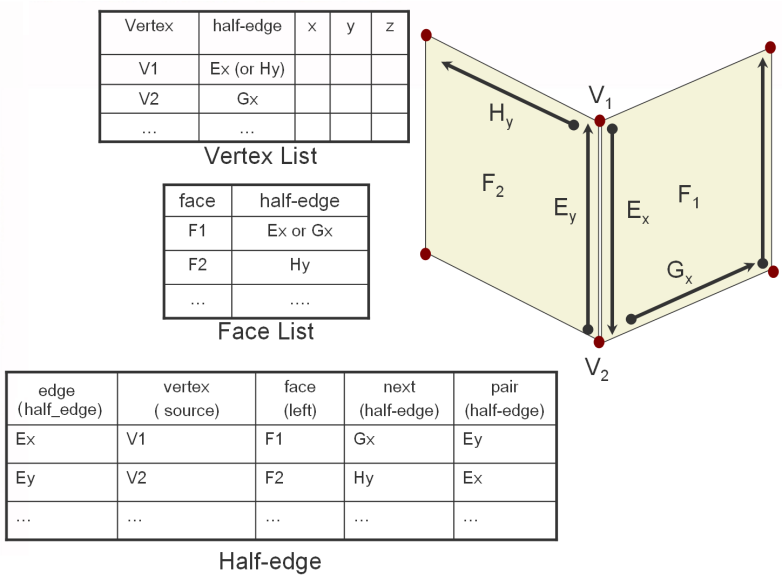


Figure 8.12: Half-edge data structure, a simple example.

Algorithm 8.1 A fast algorithm for finding all neighbors of a given vertex.

```
input v
e=v^.edge.^pair;
e0=e;
do
  w=e^.source;
  //do something with w
  e=e^.next.^pair;
while( e != e0)
```

9 Lighting & Shading

To add realism to the drawing of a 3D scene, we need to convey a sense of the shape and relative depths of objects in the scene. A lighting (equivalently *shading*) model is crucial to providing these cues; the lighting model describes how light emanating from sources in the scene interacts with the objects. It is impractical to simulate the physical laws and principles that govern lighting, so in computer graphics we use approximate models of varying complexity.

Most shading models used in graphics suppose that light sources are placed around the scene to illuminate the objects. There are several types of light sources one can model. *Point* light sources are simply points in space that emit light in all directions. *Directional* light sources, as the name implies, emit light concentrated along a particular vector. *Ambient* light is not really a light source, in that it has no location in 3D space; ambient light is simply a cheap method for simulating the complex reflections and refractions of light by modeling a base level of lighting to every object in the scene, regardless of whether the object is lit by another source in the scene.

Light sources shine on the surface of the objeects, and the incident light interacts with the surface in several ways:

- some light is absorbed by the surface (converted to heat);
- some light is reflected from the surface, and the reflected light can illuminate other objects in the scene;
- it may be transmitted into the surface, and may even leave the object at a later time (as in a piece of glass).

Absorption is an advanced interaction. Most real-time lighting models focus on the reflected light, while more advanced models, such as ray tracing, consider the transmitted rays as well.

Reflected or scattered light can be classified into two groups. *Diffuse* (or matte) lighting is light that is scattered from the surface in all directions uniformly. This type of scattering results from a strong interaction with the material of the surface, so the color of the surface is given by diffuse interactions. Diffuse lighting is independent of the viewer's position.

Specular reflection provides mirror-like reflections, and is highly view-dependent. It results from the indicent light not penetraing the object, but instead being reflected completely.

The color of a light source or a surface is described with a tri-variate intensity or luminance function I:

$$I = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix} .$$

Each component represents a color channel: red, green, and blue.

Because lighting computations involve three similar but independent calculations to handle each component, we will focus on one generic scalar value I that can represent any of the three components.

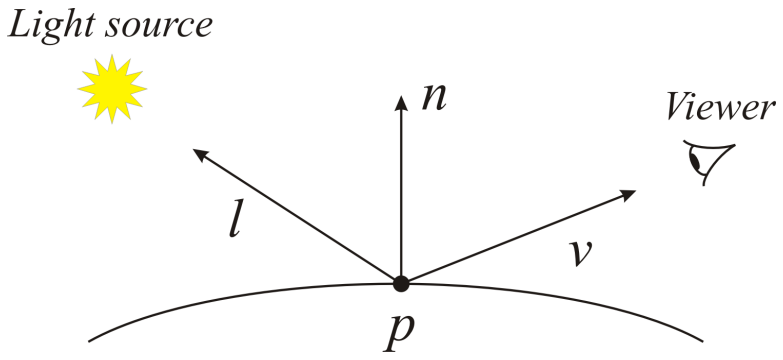


Figure 9.1: To compute the intensity at a point p on the surface of an object, we need a normal vector n, a vector l from the point to the light source, and a view vector v.

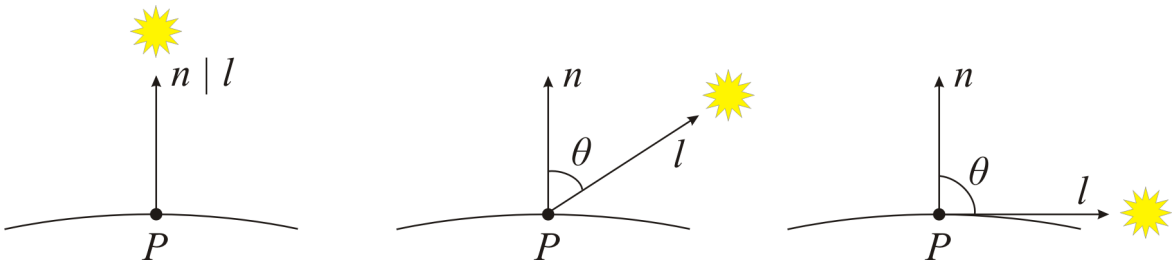


Figure 9.2: The amount of light scattered by a diffuse surface depends on the relative orientations of n and l. (Left to right) the maximum intensity occurs when l is parallel to n; as the angle between then increases, the intensity decreases; when n is perpendicular to l, there is no light scattered.

9.1 The Phong Reflection Model

9.1.1 Diffuse Lighting

To compute the amount of light that reflects from a surface, we need three vectors (Figure 9.1):

- the normal vector n of the surface at point p;
- the view vector v from p to the viewer/camera;
- and, the lighting vector l from p to the light source.

These are assumed to be normalized vectors.

In computing the intensity of diffuse light at p, there are two important factors to consider. The first factor is the position of the light source. As shown in Figure 9.2, the amount of diffuse reflection depends on the angle θ between n and l. The reflection is maximum when n and l are perpendicular, or $\cos\theta = n \cdot l = 1$. As θ increases, the amount of diffuse reflection decreases proportionally, eventually reaching zero when $n \cdot l = 0$.

The second factor is the material of the surface or more specifically, the roughness: a rough surface will have more diffuse interaction than a smooth surface. A perfectly diffuse (rough) surface is called a *Lambertian* surface. A perfect diffuse reflector scatters incident light equally in all directions, and therefore is not view-dependent. Thus diffuse lighting depends only on n and l.

Component	Gold	Black plastic	Silver
r	0.75	0.01	0.50
g	0.60	0.01	0.50
b	0.22	0.01	0.50

Table 1: k_d values for some material types.

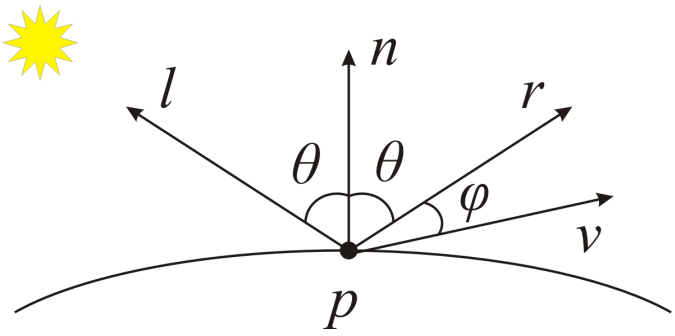


Figure 9.3: The Phong reflection model for specular highlights: the amount of light reflected is greatest in the direction of r, which is the reflection of l about n. The existence and intensity of the highlight depends on the angle ϕ between r and v.

If we let L_d represent the diffuse light from the source, and I_d represent the diffuse intensity at p, then

$$I_d \propto L_d \cos\theta = L_d(n \cdot l) \ .$$

If we add a reflection coefficient, call it k_d, that represents the fraction of incoming diffuse light that is reflected by the surface, then we can say:

$$I_d = k_d(n \cdot l)L_d \ , \qquad 0 \leq k_d \leq 1 \ .$$

The coefficient k_d is related to the material of the surface, and is usually determined by trial and error. Table 1 shows the value of k_d for a few materials.

9.1.2 Specular Lighting

With only diffuse lighting, there will be no highlights that you would expect from a shiny surface. Diffuse lighting models surfaces as being rough, so everything will have a dull, chalky appearance. Specular lighting is used to model highlights that result from incident light from the source being reflected in the direction of the viewer. As such, specular interactions are view-dependent.

The Phong model (Figure 9.3) of specular reflection is based on the observation that the amount of light reflected from a shiny surface is greatest in the direction of r, the reflection of the light source vector l about the surface normal n.

For a perfectly shiny mirror-like surface, a viewer could only see a highlight when the view vector v aligns with r. For surfaces that are shiny but not perfect mirrors, the intensity of the highlight decreases proportionally with ϕ, the angle between v and r.

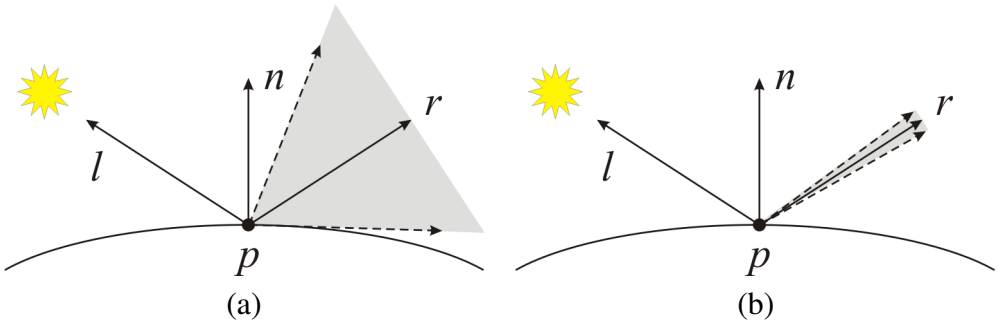


Figure 9.4: Dull versus shiny surfaces: (a) light incident on a dull surface will be specularly reflected in a large envelope about r; (b) for a shiny surface, the reflected light will be tightly concentrated around r.

Developing a physically accurate model of specular surfaces is very complex; the relationship between ϕ and the specular intensity is non-trivial. Phong proposed an approximate and empirical model for calculating the specular intensity I_s. Consider Figure 9.4. For a relatively dull surface (a), incident light will be scattered in many possible directions. For a very shiny surface, as in (b), the incident rays will be reflected primarily in the direction of r.

Based on these observations, Phong introduced a *shininess* parameter α that controls the size of the "ray envelope" that contains the reflected rays. Then the specular intensity I_s can be modeled as:

$$I_s \propto L_s(\cos\phi)^\alpha \ ,$$

where L_s is the specular intensity of the incident light.

As with the diffuse component, we can introduce a specular material coefficient $k_s \in [0, 1]$, which allows us to change from a proportionality to an equality:

$$I_s = k_s L_s(r \cdot v)^\alpha \ .$$

Given the unit vectors n and l, how can we determine the reflection vector r? As shown in Figure 9.3, the angle of incidence, θ, is equal to the angle of reflection. We also know that all three vectors must lie in the same plane.

Figure 9.5 indicates how to determine r from n and l. Because we are dealing with unit vectors, the projection of l onto n is just $(n \cdot l)n$. Also, we can see that $l + r = 2(n \cdot l)n$. Therefore, we can compute r as:

$$r = 2(n \cdot l)n - l \ .$$

9.1.3 Ambient Lighting

The diffuse and specular components of reflected light are determined by simplifying the rules of physical reflection. Neglected in this model is that reflected light is not reflected only to the viewer; light is also reflected onto other objects in the scene. In this way, any point on any surface can become a light source. This is known as *indirect lighting*.

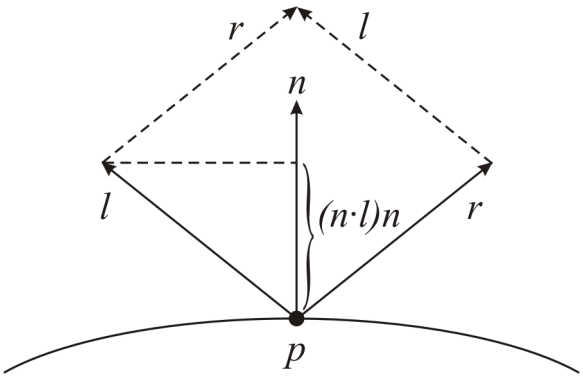


Figure 9.5: The reflection vector r can be computed from the unit vectors n and l, by noting that $l + r = 2\text{proj}_n l$.

With only our simple diffuse and specular models, however, there is no accounting for indirect light. It is too expensive to consider each point on a surface as a light source for real-time rendering. Thus, any portions of a surface that are not visible from a light source would be completely black. This results in an unrealistically dark rendering.

To account for indirect lighting in a manageable way, we imagine that a uniform back light called *ambient light* exists in the environment. Ambient light is not situated at any particular point, and it spreads in all directions uniformly.

If L_a is the amount of ambient light in the environment and I_a is the intensity of ambient light at a point, then we can model ambient light as

$$I_a = k_a L_a \ ,$$

where $k_a \in [0, 1]$ is the ambient reflection coefficient of the material.

9.1.4 Combining the Components

The three components of light that we've discussed so far – diffuse, specular, and ambient – together make up the Phong reflection model of lighting. The intensity I of reflected light at some point p is the sum of each independent lighting component:

$$\begin{aligned} I &= I_d + I_s + I_a \\ &= k_d L_d (n \cdot l) + k_s L_s (r \cdot v)^\alpha + k_a L_a \ . \end{aligned} \tag{15}$$

Note that it doesn't make sense to have negative contributions from a light source. If the angle θ between n and l is greater than $90°$, then $\cos\theta = n \cdot l < 0$. In this case, the light source contributes nothing to the intensity of reflected light at that point. Thus we should add a special condition to our model to avoid adding negative contributions. To accomplish this, let:

$$I = k_d L_d \max(0, n \cdot l) + k_s L_s (\max(0, r \cdot v))^\alpha + k_a L_a \ .$$

This is mostly an implementation detail; the form given in Equation 15 is used more often.

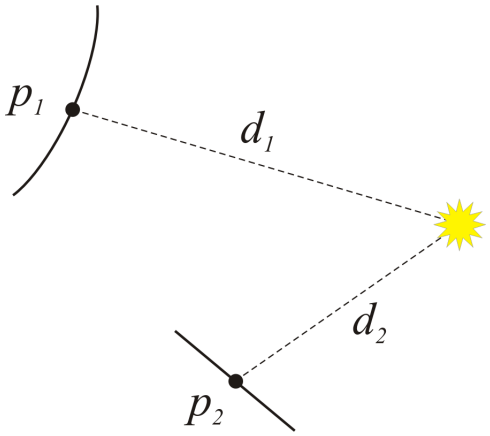


Figure 9.6: Distance attenuation is an important aspect of a lighting model. The light source contributes more to point p_2 than to p_1, because p_2 is closer.

We can easily extend the model to handle multiple light sources. Let $L^i = (L^i_d, L^i_s, L^i_a)$. Then we can model the total reflection at p by:

$$I = \sum_{i=0}^{n} \left(k_d L^i_d (n \cdot l^i) + k_s L^i_s (r^i \cdot v)^\alpha \right) + k_a L_a \ .$$

Note that the ambient term is not dependent on the number of light sources.

To complete the model, we must remember that the lighting calculations must be carried out for each color channel.

$$\begin{aligned} I_r &= k_{d,r} L_{d,r} (n \cdot l) + k_{s,r} L_{s,r} (r \cdot v)^{\alpha_r} + k_{a,r} L_{a,r} \\ I_g &= k_{d,g} L_{d,g} (n \cdot l) + k_{s,g} L_{s,g} (r \cdot v)^{\alpha_g} + k_{a,g} L_{a,g} \\ I_b &= k_{d,b} L_{d,b} (n \cdot l) + k_{s,b} L_{s,b} (r \cdot v)^{\alpha_b} + k_{a,b} L_{a,b} \end{aligned}$$

9.1.5 Distance Attenuation

One important aspect of the lighting model that should be addressed is distance attenuation. Consider Figure 9.6: clearly the incoming light at p_2 is greater than it would be at p_1. What is less clear, however, is precisely how rapidly the intensity of incoming light drops over a unit distance. We might consider an exponential drop-off, similar to other physical phenomenon such as magnetic fields. If S is the position of the light source and p is the position on some surface, we could model an exponential drop-off as $I(S,p) = \frac{I(p)}{d^2}$, where $d = \|S - p\|$. However, in practice this sort of attenuation function generates unrealistic results.

An experimental model that has proven to work well in practice is

$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d (n \cdot l) + k_s L_s (r \cdot v)^\alpha \right) + k_a L_a \ ,$$

where a, b, and c are some control parameters.

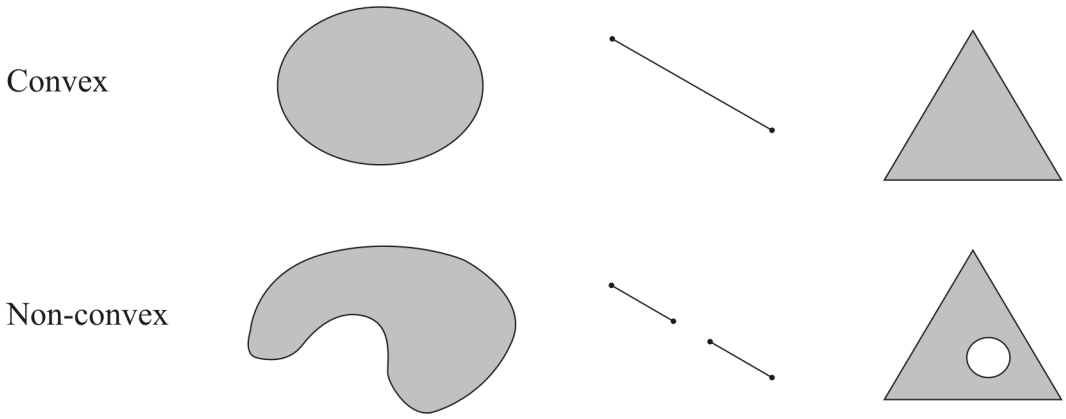


Figure 9.7: In a convex object, any point lying on the line segment between any pair of points in the object must also be in the object. Top: convex objects. Bottom: Non-convex objects.

9.2 Rendering Faces

In Section 9.1, we found out how to find the intensity at a given point on a 3D model, based on light source positions and a surface normal. However, for real-time graphics applications we can't perform a lighting computation for the required number of samples on a surface. Most often, a lighting computation is performed only at the vertices of the underlying mesh, and those values are then interpolated somehow to render the object.

Different objects require different shading effects. For instance, a cube should have a faceted appearance and a sphere should have a smooth appearance, though a cube could be considered as just a very discretized sampling of a sphere.

There are two broad types of face shading: *flat shading* and *smooth shading*:

- **Flat shading**: to visualize the individual faces, all vertices in a face are given the same normal, so that the color is uniform across the entire face. This type of shading is suitable for faceted objects, or for objects far from the viewer and light source.

- **Smooth shading**: to visualize the underlying surface, each vertex in a face is given its own normal vector. This type of shading is best for smooth objects. To interpolate the colors across a face, there are two main approaches: *Gouraud*, and *Phong*.

The rendering of a face is done with a *polygon fill routine* in the graphics pipeline. In order to have an efficient fill routine, each polygon must be convex. A convex object is one such that all points on a line between two points in the object also lie within the object; see Figure 9.7.

The benefit of having convex polygons to fill is that each scanline will be uninterrupted (Figure 9.8, because any line will intersect the edges of the polygon at most two times.

A simple polygon fill routine is described in Algorithm 9.1 and in Figure 9.9. Depending on the shading model (flat, Gouraud, or Phong), the complexity of the color computation in the inner loop will change.

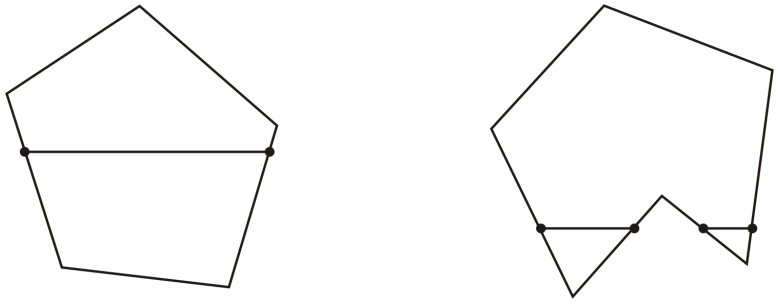


Figure 9.8: Left: when filling a convex polygon, each scanline is unbroken. Right: with a non-convex polygon, that may not be true.

9.2.1 Flat Shading

In the case of flat shading, the color computation can be moved outside of both loops. This can be done because all pixels of a flat-shaded face will have the same color. To compute this color, the normal of the face and the position of any vertex can be used to compute the color of that face. Because of this, flat shading is very efficient.

However, a scene consisting entirely of flat-shaded faces is not very visually compelling. More importantly, flat shading can even have a negative visual impact. First, specular highlights are rendered very poorly with flat shading, as the highlight will be either take over an entire face, or be invisible.

Further, flat-shaded effects suffer from a quirk of the human visual system known as the *Mach band* phenomenon. Because our vision system will blend discontinuous colors when the difference is below a certain threshold, we can see discrete representations as continuous (for example, a television image); on the other hand, when the difference is above that threshold, our vision system amplifies the boundary to make it more remarkable. In flat-shaded scenes, this causes the edges between faces to appear more pronounced than they actually are.

For these reasons, more complex polygon fill approaches are necessary.

9.2.2 Gouraud Shading

To flat-shade a face, we only need to perform one color computation per face. For smooth shading, the color computation must instead be done within the fill routine, once per pixel. The question is, how can we determine the color at each pixel such that we get a smooth transition across the face?

Gouraud shading is one particular approach to smooth shading that can be integrated into the graphics pipeline relatively cheaply. Enabling smooth shading in OpenGL invokes Gouraud shading, in fact.

Gouraud shading is based on the linear interpolation of the color values computed at each vertex of a face. The vertex colors are computed with the Phong reflection model, and each pixel in the polygon fill routine is computed by linearly interpolating these colors (Figure 9.10(a)).

Let's first consider the general case of linear interpolation (Figure 9.10(b)). Because of the properties of similar triangles, we can note that:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0},$$

Algorithm 9.1 A simple polygon fill routine.

```
for (y = y_bot; y <= y_top; y++) {
    // Find ends of the scanline
    find x_left, x_right;

    // Process the scanline
    for (x = x_left; x <= x_right; x++) {
        find color C of pixel at (x,y);
        put c into pixel (x, y);
    }
}
```

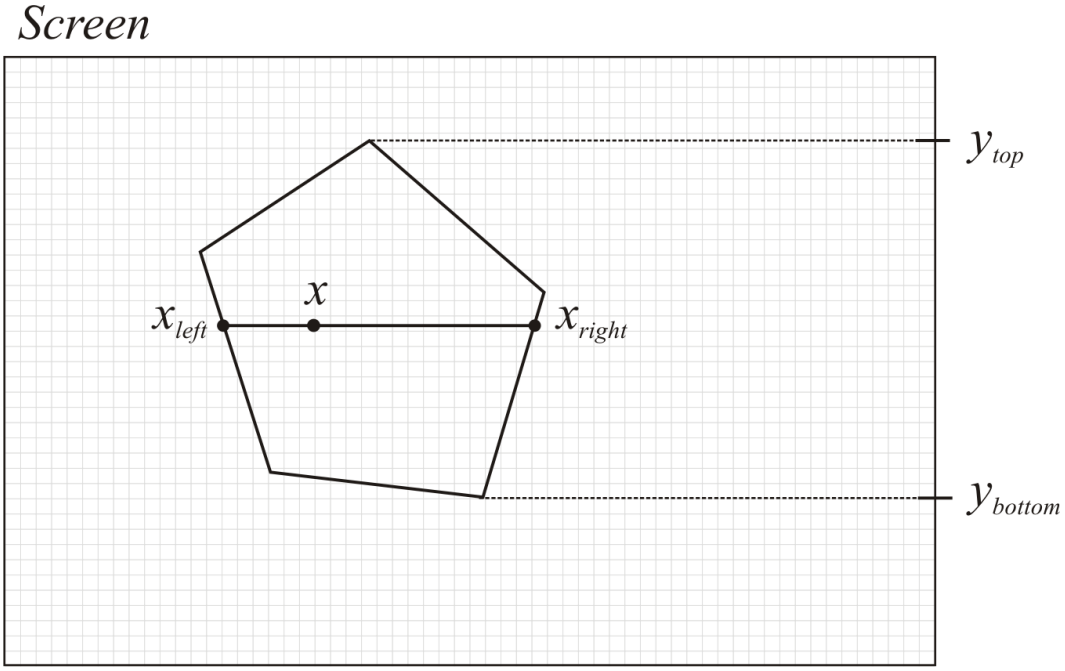


Figure 9.9: A polygon fill routine processes each face from top to bottom and left to right, and computes an intensity for each pixel based on a particular shading method such as flat or Gouraud.

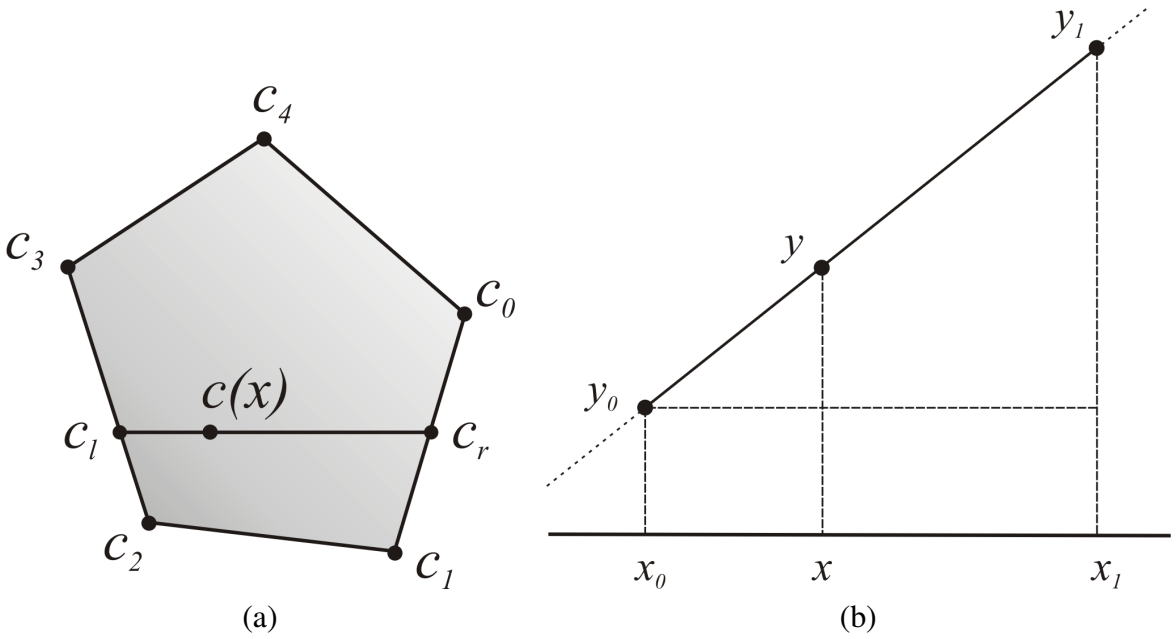


Figure 9.10: (a) Gouraud shading linearly interpolates the colors computed at each vertex of a polygon, first to compute the bounds of the scanline c_l and c_r, and again to compute the color $c(x)$ of the pixel. (b) the general setting for linear interpolation.

which can be rearranged as:

$$\begin{aligned} y - y_0 &= \frac{x - x_0}{x_1 - x_0}(y_1 - y_0) \\ y &= y_0 + \alpha(y_1 - y_0) \\ y &= (1 - \alpha)y_0 + \alpha y_1 \ , \end{aligned}$$

where

$$\alpha = \frac{x - x_0}{x_1 - x_0} \ .$$

Because $x_0 \leq x \leq x_1$, α is restricted to $0 \leq \alpha \leq 1$; therefore, we have an affine operation.

Linear interpolation can be used on many types of objects, such as scalars, points, and vectors. Or, in the case of Gouraud shading, even color values:

$$c(x) = (1 - \alpha)c_l + \alpha c_r \ ,$$

where

$$\alpha = \frac{x - x_l}{x_r - x_l} \ .$$

But, we only know the colors $c_0, \ldots, c_n$ at each vertex; c_l and c_r are not given. However, c_l and c_r can easily be computed with another linear interpolation of the appropriate vertex colors. For instance, in Figure 9.10(a), c_l lies on the edge between c_2 and c_3, so it can be computed as

$$c_l = (1 - \beta)c_2 + \beta c_3, \quad \beta = \frac{y - y_2}{y_3 - y_2} \ ,$$

and similarly

$$c_r = (1-\gamma)c_1 + \gamma c0, \quad \gamma = \frac{y - y_1}{y_0 - y_1} .$$

Example 9.1

Consider Figure 9.11. Given the coordinates of $P_0 = (0,3)$, $P_1 = (1,2)$, and $P_2 = (0,1)$, and normals n_1, n_2, and n_3, we can use the Phong reflection model to obtain a color c_i for each vertex. Let $c_0 = 60$, $c_1 = 50$, and $c_2 = 20$. Find c_l and c_r for a scanline with $P_l = (0, 1.8)$ and $P_r = (0.8, 1.8)$.

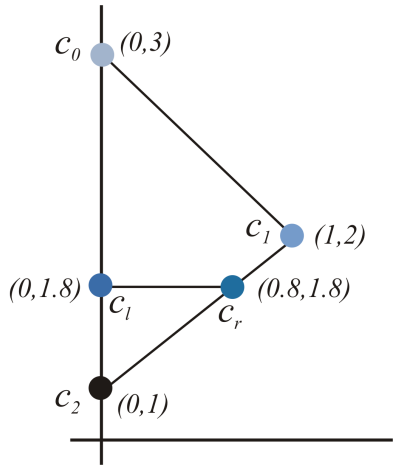


Figure 9.11: Gouraud shading uses linear interpolation to compute smoothly varying color values during the polygon fill routine.

Answer:

We know that

$$\begin{aligned} c_l &= (1-\beta)c_2 + \beta c_0 \\ c_r &= (1-\gamma)c_2 + \gamma c_1 . \end{aligned}$$

We can find β and γ from P_l and P_r, and then apply the interpolation.

$$\begin{aligned} \beta &= \frac{y_l - y_2}{y_0 - y_2} = \frac{1.8-1}{3-1} = 0.4 , \\ \gamma &= \frac{y_r - y_2}{y_1 - y_2} = \frac{1.8-1}{2-1} = 0.8 . \end{aligned}$$

So,

$$c_l = (1-0.4)c_2 + 0.4c_0 = 0.6(20) + 0.4(60) = 36 ,$$

and

$$c_r = (1-0.8)c_2 + 0.8c_1 = 0.2(20) + 0.8(50) = 42 .$$

Algorithm 9.2 Gouraud shading.

```
for (y = y_bot; y <= y_top; y++) {
    find x_l, x_r;
    find c_l, c_r;
    for (x = x_l; x <= x_r; x++) {
        find alpha
        pixel (x,y) = (1-alpha)*c_l + alpha*c_r;
    }
}
```

9.2.3 Phong Shading

With Gouraud shading, we compute color values for each vertex and interpolate those values across each face. One drawback of this approach is that because linear interpolation is an affine operation with non-negative weights, it is impossible to get an interpolated intensity that is higher than any vertex's intensity.

This fact impacts the realism of highlights, because the vertices will always have the highest intensity values; it is not possible to have a highlight centered within a face.

Phong shading addresses this issue by interpolating the per-vertex normals across the face, rather than the per-vertex colors. Then, at each pixel, the interpolated normal is used with the Phong reflection model to compute an intensity value. See Figure 9.12.

Interpolating normals is no different than interpolating a scalar or a color value:

$$n = (1-\beta)n_l + \beta n_r, \qquad \beta = \frac{x - x_l}{x_r - x_l},$$

where

$$\begin{aligned} n_l &= (1-\alpha_l)n_1 + \alpha_l n_2, \qquad \alpha_l = \frac{y - y_1}{y_2 - y_1}, \\ n_r &= (1-\alpha_r)n_3 + \alpha_r n_0, \qquad \alpha_r = \frac{y - y_3}{y_0 - y_3}. \end{aligned}$$

The per-pixel vector n must be normalized, and then it can be used in the Phong reflection model to compute an intensity at the pixel.

Under Phong interpolation, the normal vector varies smoothly from point to point. This allows for more realistic specular highlights which aren't necessarily centered on a vertex. Unfortunately, Phong shading is relatively slow, because a color is computed with the Phong reflection model for every pixel, compared to every vertex in Gouraud shading or every face in flat shading.

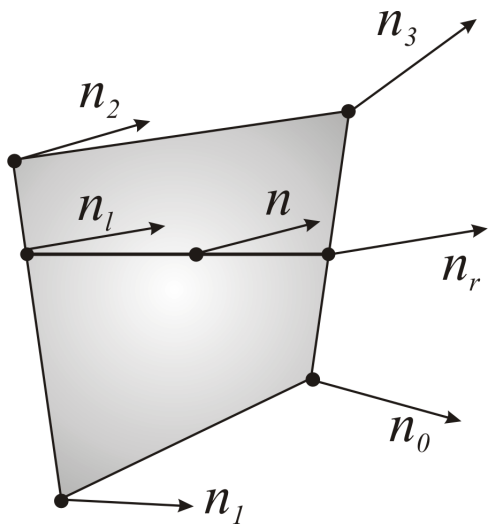


Figure 9.12: Phong shading uses linear interpolation to compute a normal for each pixel sample, which is then used to compute an intensity at that pixel.

OpenGL doesn't support Phong shading, though it can be implemented as a pixel shader on a modern video card.

9.3 Culling & Depth Buffering

Because smooth shading is non-trivial, a major consideration in the rendering pipeline is identifying which surfaces are visible and which are not, for a particular viewer/camera position. Hidden surfaces should not be considered during rendering, as any processing done on such surfaces is wasted.

There are many algorithms for hidden surface removal, which can be broadly classified into two groups:

- *Object-space* algorithms perform the hidden surface detection based on the real 3D coordinates of objects. These types of algorithms work best for scenes that contain relatively few polygons.
- *Image-space* algorithms operate on each pixel position on the viewplane.

9.3.1 Back-Face Culling

Culling means removing from a set, so in this context, back-face culling refers to the removal of certain faces from the rendering pipeline entirely. The criteria for back-face culling is this: if a polygon is facing away from a viewer, it is likely not going to be visible. This criteria assumes that the objects in the scene are closed ("water-tight").

There is a simple object-space test for whether a polygon is facing towards or away from the viewer; see Figure 9.13. If θ is the angle between the polygon's normal n and the view vector v, then the polygon is facing towards the viewer for $-90° \leq \theta \leq 90°$. Equivalently, the polygon is front-facing if $\cos\theta = n \cdot v \geq 0$.

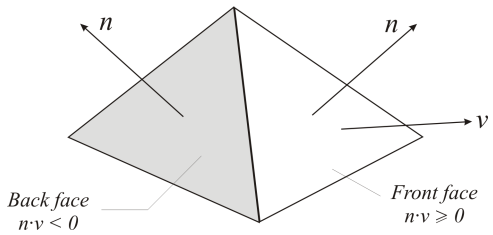


Figure 9.13: Back-face culling requires only a simple dot-product comparison per face to remove back-facing polygons. However, it is not always appropriate to remove back-facing polygons.

If back-face culling is appropriate for an application, then it can save a lot of rendering overhead. However, there are many times when it is not appropriate, such as when a polygon is used to model a two-sided planar object such as a piece of paper: the paper is still visible even if it is facing away from the viewer.

Removing back faces is also not enough to catch all hidden surfaces, as objects can fully or partially occlude each other. Thus we need more complex methods to handle these situations.

9.3.2 The Depth Buffer

A common image-space approach to hidden surface removal. The depth buffer is also referred to as the z-buffer moethod, because in a standard viewing frame, the viewplane is perpendicular to the z-axis and the z coordinate of an object thus implies its depth in the scene.

Each point p on an object is converted to a 2D point the view plane during the rendering pipeline. The color at each visible point is stored in a frame buffer and then drawn to screen. How can we do this in an efficient *and* correct way, such that more distant objects are properly occluded by closer objects? Note that we render a scene polygon by polygon, with no global depth information.

The solution is a z buffer, which has the same resolution as the frame buffer (Figure 9.14). Each location in the z buffer contains the depth of the closest intersection point on any polygon processed so far. The normalized view coordinate is used in the z-buffer algorithm.

Integrating a depth buffer into the rendering pipeline is fairly straightforward. First, both the frame buffer and depth buffer should be initialized with sensible values: the frame buffer with the background (clear) color, and the depth buffer with some maximum z value such that any actual face will definitely be closer.

Then, each face can be rendered with the polygon fill routine, with the added condition that pixels processed in the routine will only be written if the face is closer than the current pixel sample, according to the z buffer's value. See Algorithm 9.3.

After Algorithm 9.3 is executed, the frame buffer contains the intensity values of all visible surfaces. The z-buffer, similarly, contains the depth value corresponding to each surface rendered to the frame buffer.

The only missing piece of the z-buffer algorithm is the depth computation (the *ed line in Algorithm 9.3). Consider Figure 9.15. We know the depth values z_0, z_1, and z_2, and the coordinates (x, y) of the current pixel on the scanline. We can find the depth at (x, y) by linearly interpolating the z values of the vertices across the face.

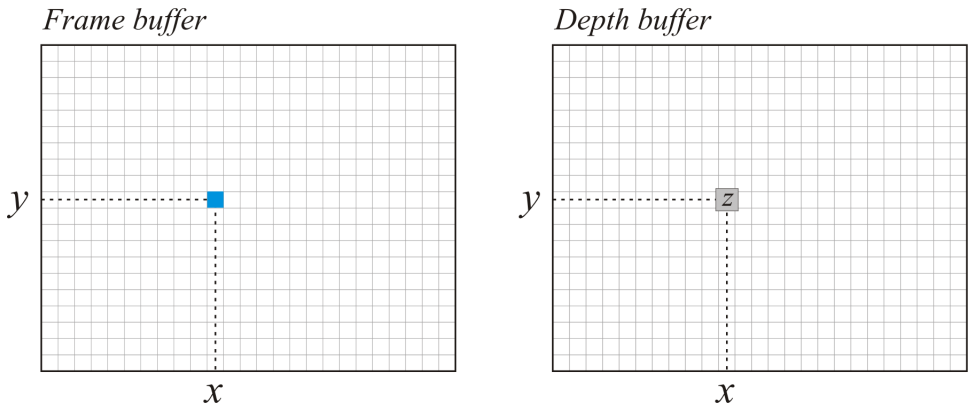


Figure 9.14: The depth buffer is the same size as the frame buffer, and stores the depth of the closest point rendered into each pixel. A new pixel is stored in the frame buffer only if its associated depth is less than the current pixel's depth.

Algorithm 9.3 The rendering pipeline with z-buffering.

```
for all (x,y):
    frameBuf[x,y] = I_background
    depthBuf[x,y] = z_max

for each polygon F in mesh M:
    for each point p = (x,y) in the polygon fill routine:
        z = distance from p to viewer (*)

        if depth[x,y] > z:
            depthBuf[x,y] = z
            frameBuf[x,y] = color at p
        endif
    endfor
endfor
```

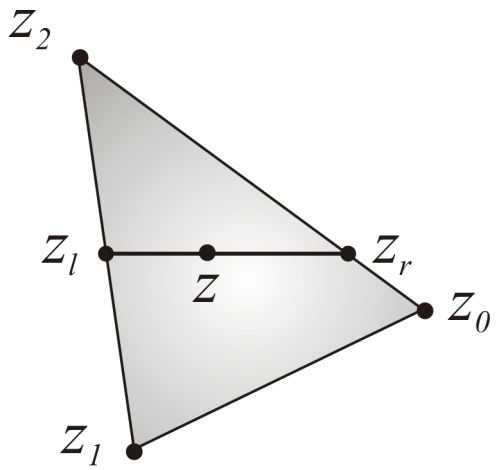


Figure 9.15: The depth of a sampled surface point is computed by linearly interpolating the depth value of each vertex, similar to Gouraud shading's interpolation of intensity values.

10 Ray Tracing

For real-time rendering, we take a scene specification including objects and light sources, and compute the intensity of each visible point based on a simple, local shading model. While this approach can produce very realistic renderings, there are still some global lighting interactions that can not be easily or accurately approximated with real-time techniques, such as translucency and indirect lighting.

Ray tracing is a rendering technique that can account for very complex global interactions, but in general it does not run at interactive rates. Ray tracing is based on the observation that an eye/camera can only see rays of light that emanate from a light source, travel through the scene, and enter the eye/camera device. Such rays may come directly from the light source, after reflecting from one or more surfaces in the scene, or after transmission through one or more surfaces.

However, a light source emits light in infinitely many directions and only very few of those emitted rays ever reach the viewpoint. Thus it doesn't make sense to follow each potential ray emitting from a light source, even if the potential number of rays is suitably discretized.

Ray tracing considers the opposite problem. By reversing the direction of the rays and following rays that *start* at the *center of projection* (COP), the number of rays to follow is determined by the resolution of the rendering output. Figure 10.1 shows the general setting for ray tracing: for each pixel (x, y) in the output image, we cast a ray from the COP through the pixel location. The intersection of the ray with each object in the scene is computed, and the intensity at the closest intersection is computed based on light sources and occlusion by other objects.

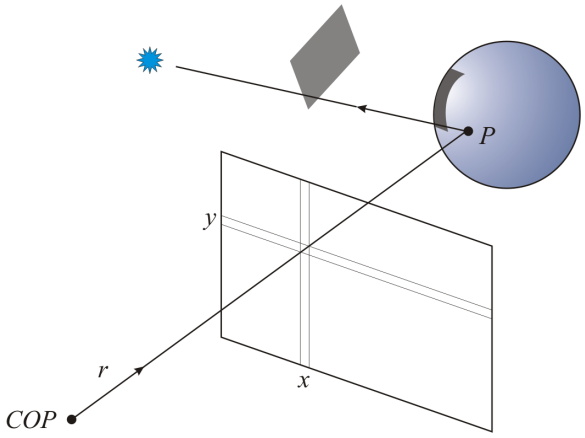


Figure 10.1: To ray-trace an image, a ray is cast from the COP through each pixel. When an intersection is found, the visible light sources are used to compute a pixel intensity.

The aim of ray tracing (or rendering in general) is to assign a color/intensity value to every pixel in the output image. In ray tracing, the ray cast from the COP to a pixel location may either intersect a surface, or travel off to infinity without striking any object. Because the depth of each intersection can easily be computed, we can easily test whether an intersection point is the closest intersection; thus, ray tracing does not require any additional effort for hidden surface removal. In the case where the ray intersects a surface, we can compute an intensity for that point based on the Phong reflection model, just as in the local setting.

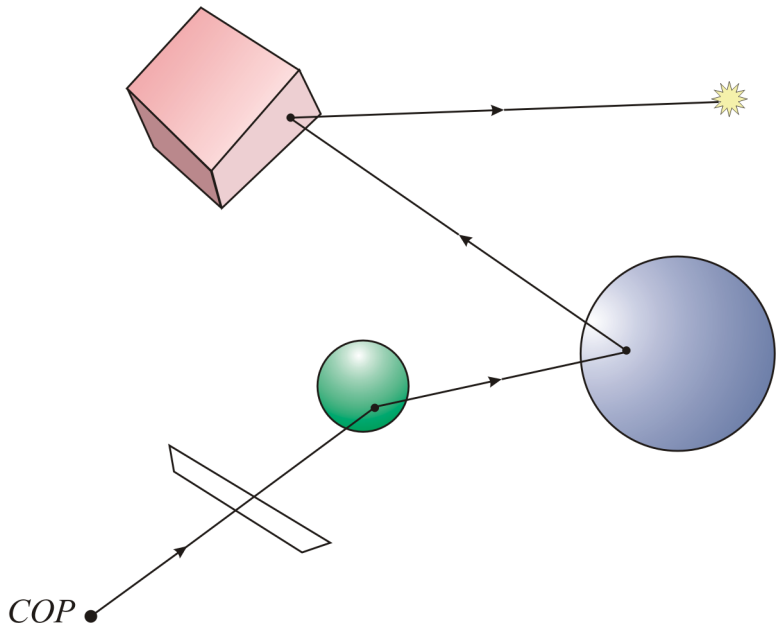


Figure 10.2: Ray tracing allows for global interactions such as indirect lighting. A ray cast from the center of projection intersects a surface, and a reflection ray is recursively cast until a light source is reached (or until a maximum recursion depth is reached.

The true benefit of ray tracing, however, goes beyond free hidden surface removal. We can get accurate shadows in a ray-traced scene by checking whether a point of intersection is illuminated by any light source. This is done by casting a *shadow ray* from the intersection point to a light source and performing the same intersection testing. If the shadow ray intersects another surface before reaching the light source, then the point is not illuminated by that light source. See Figure 10.1: part of the sphere will lie in the shadow of the planar object that lies between it and the light source.

10.1 Indirect Lighting

Ray tracing also allows for global illumination effects, such as *indirect lighting*. Recall that the ambient term in the Phong reflection model was introduced as a crude approximation of indirect lighting; without it, any part of a surface not directly visible from a light source would be completely black. In reality, each surface that a light ray is incident upon acts as a new point light source by reflecting some of the incident light. The shininess of a surface determines how much incident light is reflected.

We can consider the reflected ray as just another ray to be traced, and follow it as it bounces from surface to surface until it either goes off to infinity or intersects a light source. As not all incident light is reflected, we must take into account the absorbtion of light at each surface. See Figure 10.2.

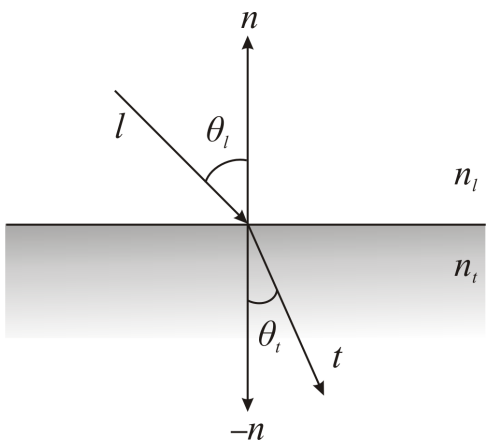


Figure 10.3: The direction of a transmitted ray depends on the indices of refraction of the two materials.

10.2 Transmission

Ray tracing can easily model translucent surfaces, for which incident light is partially absorbed (diffused), and the remaining light is split between

- a transmited ray, and
- a reflected ray.

Diffusion scatters incident light in all directions, so it is not feasible to trace diffuse reflection rays. The transmitted and reflected rays, however, can be traced efficiently.

The speed of light depends on the material through which it is traveling. When light transitions between two materials, the light is bent at the boundary by a certain amount. This phenomenon is well-studied in physics, and there is a law that describes the change in direction of the light ray known as *Snell's law*.

Consider the setting in Figure 10.3. The light ray l is incident to the surface at angle θ_l relative to the surface normal, and the incident material's *index of refraction* is n_l. The light is transmitted through the object along transmission ray t, at angle θ_t to the normal, where the material of the object has index of refraction n_t. Snell's law relates these quantites as

$$\frac{\sin\theta_l}{\sin\theta_t} = \frac{n_t}{n_l} = \eta \, .$$

10.3 Recursive Algorithm

Ray tracing involved three main steps at each intersection point:

1. Compute the contribution of all light sources at the intersection point using the Phong reflection model.
2. Cast and trace a ray in the direction of a perfect reflection.

3. Cast and trace a ray in the direction of a transmitted ray.

The latter two operations involve a recursive call to the ray-tracing algorithm. Thus we need to enforce robust termination conditions to keep the rendering time from running out of control. The tracing of a ray is terminated when:

- the ray goes off to infinity without intersecting anything;
- the ray reaches a light source, or;
- the ray becomes very weak.

At each surface that a ray intersects, there are two possible recursive calls, for the reflected and transmitted rays. Algorithm 10.1 provides pseudo-code for a recursive ray tracer.

Diffuse light interactions are ignored in the ray-tracing process, except when the Phong reflection model is used to evaluate the intensity at an intersection point. Thus ray tracing gives more precedence to specular interactions, making ray tracing most suitable to highly reflective environments. For primarily diffuse scenes, ray tracing can lead to an artificially clean and shiny appearance (see Figure 10.4). (*Radiosity* approaches attempt to model these diffuse interactions more accurately than ray tracing, drastically improving realism for most scenes.)

10.4 Intersection Testing

The bulk of processing time in a ray tracer is spent calculating ray-object intersections. Some object representations allow for easy intersection tests, while others require considerably more work.

10.4.1 Ray-Sphere Intersection

Ray-sphere intersection is perhaps the easiest intersection test. A sphere can be represented implicitly, as discussed in Section 7.1. The surface of a sphere of radius r centered at p_c is defined as all points p satisfying

$$\|p - p_c\|^2 = r^2 \; . \tag{16}$$

Equivalently, if $p_c = (x_c, y_c, z_c)$ we can write

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2 \; .$$

This setting is depicted in Figure 10.5.

A ray can be defined as a point p_0 and a direction vector d, such that any point q along the ray satisfies $q = p_0 + td$ for some t. The particular value of t can be used to find the nearest intersection (eg. $t_1 < t_2$ implies that object 1 is closer than object 2) and to discard invalid intersections (eg. $t < 0$ implies the object is "behind" the ray).

To intersect a ray with a sphere, we can substitute the expression for a point along the ray for a point on the surface of the sphere, and then solve for t. Let $q = p_0 + td$ be a point along the ray.

Algorithm 10.1 A recursive ray tracer.

```
Trace (Point P, Vector d, int depth)
{
    Color cLocal, cReflected, cTransmitted;
    Vector n, r, t;    // Normal, reflected/transmitted rays
    Intersection iHit; // Intersection information
    Point q;           // Intersection point

    // Terminate the trace at a certain depth
    if (depth > MAX_DEPTH)
        return (cBackground);  // Background color

    // Find the closest intersection of the ray
    iHit = intersect(p, d);

    // Check the type of intersection
    if (iHit.Type == None)
        return (cBackground);  // Background color

    else if (iHit.Type == LightSource)
        return (iHit.Object.color);  // Light source color

    else  // Intersected an object
    {
        q = iHit.IntersectionPt;
        n = normal(q);
        r = reflect(q,n);
        t = transmit(q,n);

        cLocal = Phong(q, n, r, iHit.Object);
        cReflected = Trace(q, r, depth+1);
        cTransmitted = Trace(q, t, depth+1);

        return (cLocal + cReflected + cTransmitted);
    }
}
```

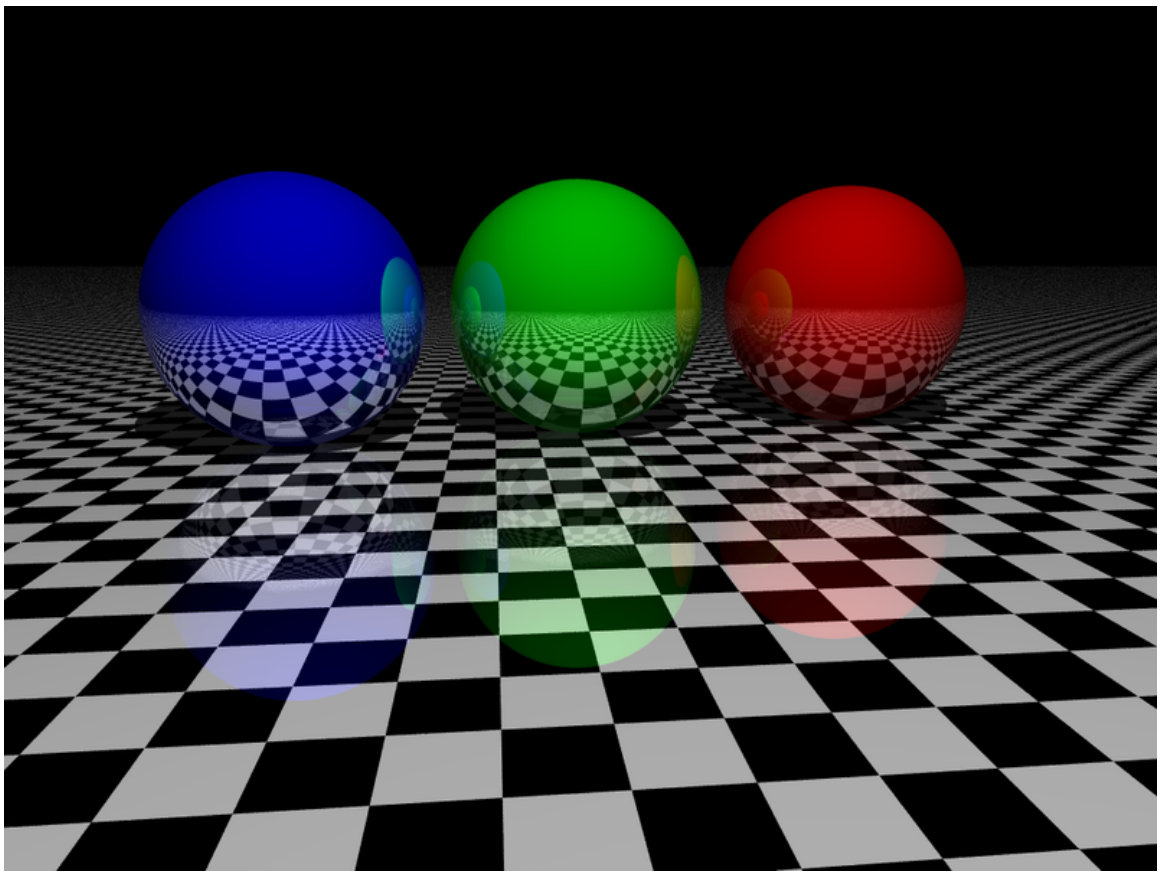


Figure 10.4: A ray-traced scene with three highly-reflective spheres. Source: https://commons.wikimedia.org/wiki/File:Raytracing_reflection.png

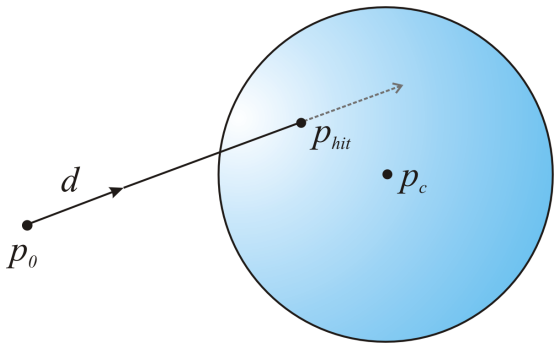


Figure 10.5: Ray-sphere intersection can be solved by substituting the ray description into the sphere definition.

Substituting into Equation 16, we get:

$$\begin{aligned} \|p_0 + td - p_c\|^2 &= r^2 \\ \|td + D\|^2 &= r^2 \\ (td + D) \cdot (td + D) &= r^2 \\ t^2(d \cdot d) + 2t(d \cdot D) + D \cdot D &= r^2 \\ t^2 + 2t(d \cdot D) + \|D\|^2 - r^2 &= 0 \,, \end{aligned}$$

where $D = p_0 - p_c$ and $d \cdot d = \|d\|^2 = 1$.

We have a quadratic expression in t, the unknown, which can be solved easily via the quadratic equation:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \,,$$

where $a = 1$, $b = 2(d \cdot D)$, and $c = \|D\|^2 - r^2$. We can simplify this expression:

$$\begin{aligned} t &= \frac{-2(d \cdot D) \pm \sqrt{4(d \cdot D)^2 - 4(\|D\|^2 - r^2)}}{2} \\ &= -(d \cdot D) \pm \sqrt{(d \cdot D)^2 - \|D\|^2 + r^2} \end{aligned}$$

The quantity under the square-root, $(d \cdot D)^2 - \|D\|^2 + r^2$, is known as the discriminant. If the discriminant is negative, then the ray does not intersect the sphere. Otherwise, the sphere is intersected at $p_0 + td$ for each t value; the smallest t value gives the closest intersection.

Example 10.1
Consider the unit sphere ($p_c = (0, 0, 0)$, $r = 1$) and a ray defined by $p_0 = (2, 0, 0)$ and $d = (-1, 0, 0)$. Find the intersection point.

Answer:

Here, $D = p_0 - p_c = (2, 0, 0)$. We can first check the discriminant to see if there is an intersection.

$$(d \cdot D)^2 - \|D\|^2 + r^2 = (-2)^2 - (2)^2 + 1^2 = 1 \,.$$

The discriminant is non-negative, so there is an intersection. The intersection t values are:

$$\begin{aligned} t &= -(d \cdot D) \pm \sqrt{1} \\ &= 2 \pm 1 \,. \end{aligned}$$

The two intersection points, q_{enter} and q_{exit}, are then:

$$\begin{aligned} q_{enter} &= p_0 + (2 - 1)d &= (1, 0, 0) \\ q_{exit} &= p_0 + (2 + 1)d &= (-1, 0, 0) \,. \end{aligned}$$

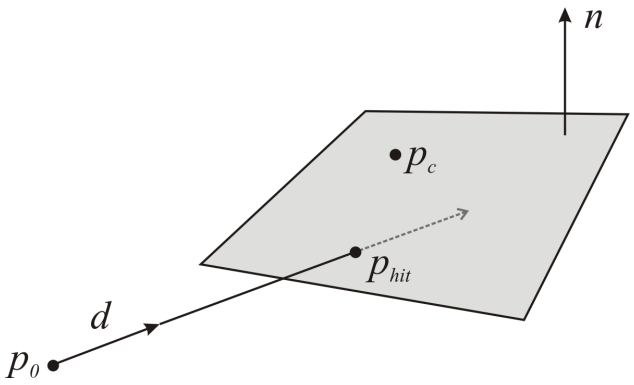


Figure 10.6: Ray-plane intersection.

To find the normal at the intersection point on a sphere, we can simply take the vector from the center p_c to the intersection point q and normalize it:

$$n = \frac{q - p_c}{\|q - p_c\|} .$$

10.4.2 Ray-Plane Intersection

For polygonal objects, we will need a fast intersection test for planar objects. We will first consider intersecting a ray with an infinite plane, and then consider how to determine if the intersection point is inside of a polygon (bounded plane).

A plane can be defined by a single point p_c lying in the plane, and a normal vector n. The plane then consists of all points p satisfying

$$(p - p_c) \cdot n = 0 .$$

In other words, the angle θ between a vector $p - p_c$ lying in the plane and the normal n will be $90°$. See Figure 10.6.

To intersect a ray $p_0 + td$ with a plane, we take a similar approach to ray-sphere intersection: substitute the ray description into the plane equation and solve for t:

$$\begin{aligned} (p_0 + td - p_c) \cdot n &= 0 \\ (p_0 - p_c) \cdot n - td \cdot n &= 0 \\ -td \cdot n &= (p_0 - p_c) \cdot n \\ t &= -\frac{(p_0 - p_c) \cdot n}{d \cdot n} . \end{aligned}$$

The normal of the intersection point $p_{hit} = p_0 + td$ is just the normal n of the plane.

Inside/Outside Testing

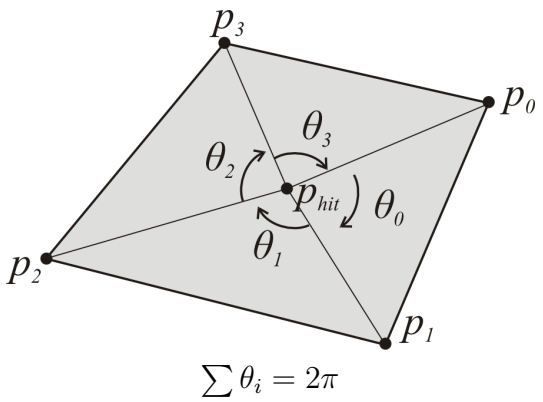


Figure 10.7: To test whether a ray-plane intersection lies within the bounds of a polygon in the plane, we can check the sum of the angles.

Finding an intersection with an infinite plane is not a problem that occurs often in practice. More often we are dealing with bounded planes, such as a polygon defined by a convex set of vertices $p_0, \ldots, p_n$.

To determine whether the point of intersection p_{hit} lies inside a polygon, we can use a simple test on convex polygons: the sum of the angles θ_i between $p_i - p_{hit}$ and $p_{i+1} - p_{hit}$ will sum to $2\pi = 360°$ if p_{hit} lies inside the polygon. See Figure 10.7.

Barycentric Coordinates for Triangle Meshes

The ray-polygon intersection routine discussed is sufficient for simple objects such as lone polygons or cubes. For complex polygonal meshes, however, we need a more robust and informative intersection test. We will consider triangle meshes, as non-triangle meshes can be and often are interpreted as triangle meshes.

For triangle mesh objects, we will be performing intersection tests on each individual triangle, but we also want to impart some global properties over the mesh during rendering, such as smooth shading. A simple plane-based intersection with the surface normal equal to the plane's normal isn't sufficient for this purpose.

Expressing an intersection point p in barycentric coordinates provides us with information that can be used to smoothly shade ray-traced triangle meshes. Barycentric coordinates allow us to express any point p lying in a triangle $\triangle ABC$ as an affine combination of the vertices $\{A, B, C\}$. The barycentric coordinates (α, β, γ) of p are defined such that:

$$p = \alpha A + \beta B + \gamma C \,, \quad \alpha + \beta + \gamma = 1 \,.$$

Note that only two of the three coordinates are necessary, as the third follows from $\alpha + \beta + \gamma = 1$.

Barycentric coordinates can be viewed as the translation of a point by two vectors. Consider:

$$\begin{aligned} p &= \alpha A + \beta B + \gamma C \\ &= (1 - \beta - \gamma)A + \beta B + \gamma C \\ &= A + \beta(B - A) + \gamma(C - A). \end{aligned}$$

A nice property of barycentric coordinates is that they provide a fast inside/outside test. A point p is inside of $\triangle ABC$ if and only if $0 \leq \alpha, \beta, \gamma \leq 1$.

The biggest benefit of this coordinate system is that they provide blending factors for combining per-vertex properties across the triangle. Barycentric coordinates actually describe a bilinear interpolation. For example, if $p = \frac{1}{2}A + \frac{1}{5}B + \frac{3}{10}C$, then the normal at p should be $n_p = \frac{1}{2}n_A + \frac{1}{5}n_B + \frac{3}{10}n_C$ (before normalization). Similarly, the color I_p can be computed with the same weightings: $I_p = \frac{1}{2}I_A + \frac{1}{5}I_B + \frac{3}{10}I_C$.

How can we find barycentric coordinates for a given point p and triangle $\triangle ABC$? One obvious approach is to set up a system of equations, as there are three equations and three unknowns:

$$\begin{aligned} x_p &= \alpha x_A + \beta x_B + \gamma x_C \\ y_p &= \alpha y_A + \beta y_B + \gamma y_C \\ z_p &= \alpha z_A + \beta z_B + \gamma z_C \end{aligned}$$

Another possible solution is depicted in Figure 10.8. If $S = \text{area}(\triangle ABC)$, then the relationship between the internal areas $\{S_1, S_2, S_3\}$ defined by p_{hit} can determine the barycentric coordinates. If α is the weight applied to A, then α is determined by the area S_1 of the opposite triangle. Why? Consider $S_1 = 0$: this implies that p lies on the line BC, which in turn implies $\alpha = 0$. In fact, we have:

$$\begin{aligned} \alpha &= \frac{S_1}{S} \\ \beta &= \frac{S_2}{S} \\ \gamma &= 1 - \alpha - \beta \,. \end{aligned}$$

The internal areas S_i can be determined by exploiting properties from linear algebra. Recall that the area of the parallelogram spanned by two vectors u and v is given by the length of the cross-product $u \times v$. Consider the parellologram shown in gray in Figure 10.8, which is defined by $A - p$ and $B - p$. The area is given by $\|(B - p) \times (A - p)\|$, and is exactly twice the area S_3. By the same reasoning, S, S_1 and S_2 are given by:

$$\begin{aligned} S &= \frac{1}{2}\|(A - B) \times (A - C)\| \\ S_1 &= \frac{1}{2}\|(B - p) \times (C - p)\| \\ S_2 &= \frac{1}{2}\|(p - C) \times (p - A)\| \,. \end{aligned}$$

Note that S and S_i are *signed* values, as the sign of the coordinates is important for the intersection test.

10.5 Scene Description

The input to a ray tracer is a scene description, including objects and light sources. Most ray tracing engines include a flexible scene description language, which allows an editor of some sort

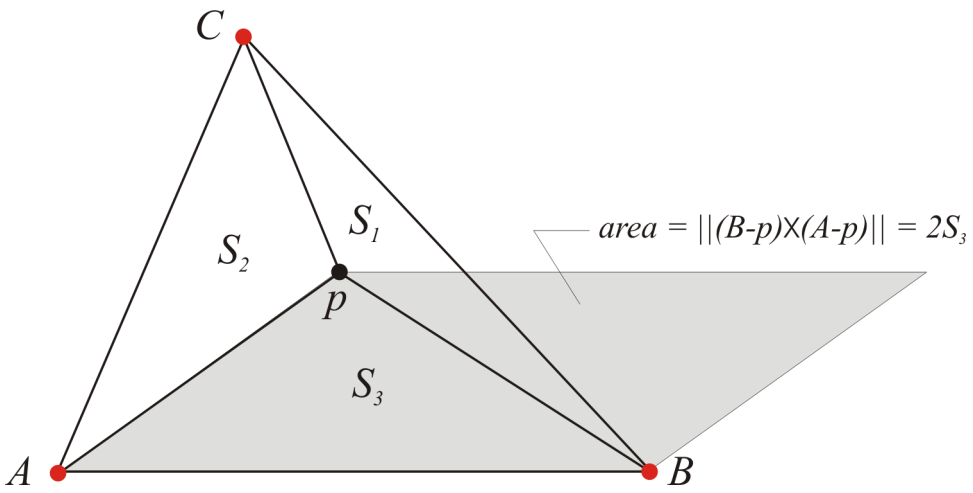


Figure 10.8: The relationships between $S_1 = \text{area}(\triangle PBC)$, $S_2 = \text{area}(\triangle PCA)$, $S_3 = \text{area}(\triangle PAB)$, and $S = \text{area}(\triangle ABC)$ yield barycentric coordinates for p.

to create objects, assign materials, and place them in the scene with affine transformations. The ray tracer then parses the file to create a list of objects to be used during the rendering phase.

As an example, consider the following scene description in the Scene Description Language described in *Computer Graphics Using OpenGL* (Hill, 2001, Prentice Hall).

```
! Sample scene in SDL
light 2 9 8 1 1 1          ! White light (1,1,1) at (2,9,8)
diffuse 0.9 0.1 0.1        ! Diffuse state - reddish
push                       ! Push transformation stack
translate 3 5 -2 sphere    ! Sphere centered at (3,5,-2)
translate 2 2 -1 sphere    ! Second sphere, at (2,2,-1)
pop                        ! Restore identity matrix
diffuse 0 1 0              ! Set following object to green
push scale .2 .3 .3 cube pop
! End of scene
```

One final piece missing from our ray tracer is the outer loop, in which rays are cast. Algorithm 10.2 provides a ray-casting outer loop that calls the `Trace()` function in Algorith 10.1 for each pixel on the viewplane.

Figure 10.9 depicts the setting for the outer loop. The viewplane is defined by a width `W` and height `H` in world coordinates, a desired number of pixel samples `xRes` and `yRes` in each direction, the eye position `eye`, and the distance from the eye to the viewplane, `N`. Algorithm 10.2 visits each pixel in a viewplane defined by these quantities, and casts a ray from the eye through each pixel.

10.6 Anti-Aliasing

Because of the discrete nature of ray tracing (a single ray is shot through each pixel in the image), we often encounter unpleasing aliasing effects. Aliasing is caused by undersampling of a "signal",

Algorithm 10.2 Ray tracer outer loop (see Figure 10.9).

```
Input: W, H, N, xRes, yRes, eye

Point pixelLoc;
Vector d;
float xInc = 2*W/xRes;
float yInc = 2*H/yRes;

for (x = -W; x <= W; x += xInc)
{
    for (y = -H; y <= H; y += yInc)
    {
        pixelLoc = (x, y, N);
        d = pixelLoc - eye;
        d.Normalize();

        // Trace the ray
        Trace (eye, d, 0);    // depth = 0
    }
}
```

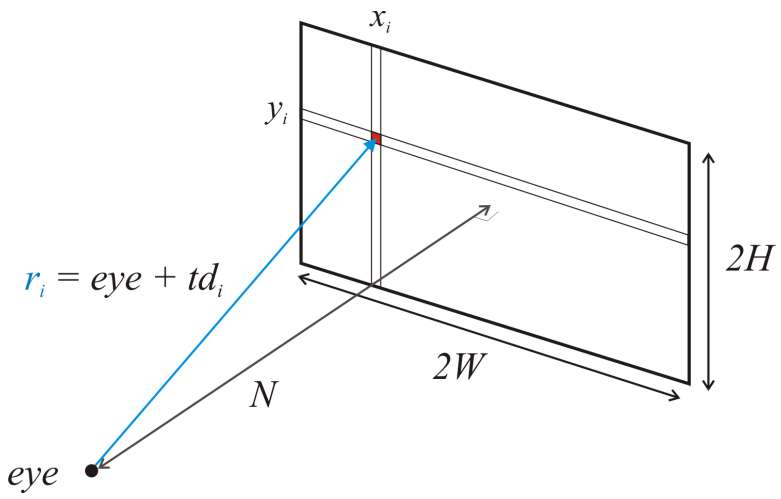
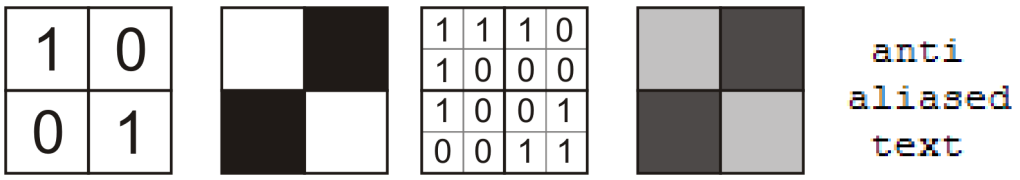


Figure 10.9: A ray tracer casts a ray through each pixel in a viewport defined by a width W and height H, x and y pixel resolution, and distance N from the eye.



(a) (b) (c)

Figure 10.10: Anti-aliasing by supersampling each pixel: (a) with no anti-aliasing, only one intensity per pixel is used, creating abrupt intensity gradients; (b) 4x supersampling uses four samples per pixel and averages them to create more gradual intensity changes. (c) an example of anti-aliased text.

or in this case the objects in the scene. The main idea behind anti-aliasing is to reduce the aliasing effect by producing gradual color changes rather than abrupt ones.

There are many techniques for anti-aliasing, but the simplest to understand and to implement is known as *supersampling*. As the name implies, supersampling is based on acquiring more than one color value sample per pixel. This is done by casting several rays through each pixel location, each slightly perturbed from the last. See Figure 10.10.

10.7 Acceleration Techniques

Ray-tracing is an expensive rendering technique for two reasons: it recursively traces rays to a maximum depth; and finding the closest intersection is expensive. A brute force approach to intersection testing would simply intersect the ray with every object in the scene and choose the nearest one. In practice, this is wildly expensive and impractical, as many objects can be eliminated without performing any intersection tests on it.

Consider a triangle mesh made up of 10,000 triangles. Intersecting a ray with each and every face is very expensive, especially if you do it for every ray to be traced. A simple technique for avoiding those 10,000 intersection tests whenever possible is to surround the entire mesh with a *proxy*, which in most cases is a *bounding box*. Proxy is a more general term for any bounding object whose intersection test is cheaper than the contained object. Figure 10.11 shows a complex triangle mesh surrounded by a simple axis-aligned bounding box. Before computing intersections with all triangles in the mesh, we can first see if the ray even intersects the bounding box, which only requires 6 or fewer ray-polygon intersections. If the bounding box isn't intersected by the ray, then certainly no triangle in the mesh will be.

There are many more advanced acceleration techniques, such as heirarchical bounding boxes, octree space partitioning, and ray marching. The reader is encouraged to investigate these techniques further.

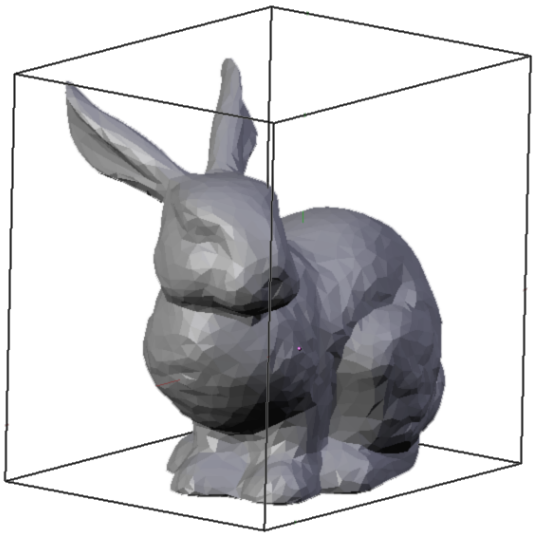


Figure 10.11: A bounding box can save a lot of unnecessary intersection tests.

11 Texture Mapping

Motivation: geometry and properties of many objects vary significantly across surface. Would need many many triangles to represent at high enough resolution: large memory and rendering requirement. Instead, augment properties of object using images. "Texture mapping is the use of an image to store and render spatially varying surface properties." Most obvious example is changing colour, but as we'll see, other properties can also be modified.

Three main subproblems of texture mapping. Mapping function: how points on surface are mapped to location in image. Sampling: how an image value is chosen for mapped coordinates. Modifying surface properties: different ways values from image can modify surface properties during rendering.

11.1 Mapping Functions

Good properties. bijective: has an inverse (go from point on image to object). Allows full control of properties, i.e. each point on object has point on image which can be independently modified. In cases where it is known parts of object should be identical, property can be dropped to save save. Even so, it may be desirable not to in case future needs necessitate bijectivity.

low distortion: little stretching, squashing, warping, twisting, etc. Types of distortion. Area: consistent level of detail (resolution) for whole object. Creating and interpreting texture also more intuitive. Shape/angle: shade drawn on texture would have same shape on object. Again, more easy to create and interpret textures.

continuity: adjacent positions on object are adjacent in map. For most objects, perfect continuity not possible. Instead, try to put seams in natural/inconspicuous places. Continuity and distortion are a tradeoff: more seams = less distortion

11.1.1 Defining Mapping Functions

How do we actually evaluate mapping function? Works just like any other property (e.g. normals) during rendering. For ray tracing geometric primitives (plane, sphere, cone) calculate map directly from intersection point. For triangle mesh (either raster or ray trace), evaluation of mapping done and stored at vertices. When rendering fragment, these values are interpolated across triangle with barycentric just like any other property (normal, colour, etc.).

Typically call texture coordinates UV coordinates. U corresponds to horizontal axis of image, V vertical. Normalized to the range 0-1. Values outside this range can be handled in several different ways, such as wrapping, clamping, or using a fix value. More details in programming notes section [ref].

So, we need function that takes points on object (vertices in case of mesh), and calculates UV coordinates. Many different ways to define texture mapping function. Look at a few common ones below.

Spherical Maps Brief recap on spherical coordinate will go here. For mapping object, make vector from centre to point. Calculate spherical coords of vector. Maps spherical coords to plane: many ways to do this with different properties (see geographic map projections)

Works well for objects that are (roughly) spherical (sports balls, cubes, round rocks, etc.) Not necessarily bijective, depending on geometry of object **figure**

Cylindrical Maps Similar to spherical, but use cylindrical coordinates instead. Brief recap on cylindrical coordinates here. Cylindrical coordinates map nicely to the plane by simply unrolling cylinder.

Again, works well for objects that are well approximated by a cylinder (pole, cucumber, snake) Also not necessarily bijective.

Custom Maps Above work well for simple objects, for complex geometry they often do not behave well. In practice, most objects will have a custom mapping function. Custom does not necessarily mean manual, however. Lots of research and good algorithms for creating custom maps for objects based on geometric properties. Also research on how to best pack regions of map most efficiently into rectangular texture. While the result of such algorithm may not be ideal, they at least provide a starting point for manual fine tuning. In most cases though, the automatic result is good enough.

11.2 Texture Sampling

Texture mapping function gives us continuous values in domain of texture. However, images have a fixed resolution, so we need way to sample the image to get a value for a given UV. In general, the value of points between pixels can be calculated with different interpolation functions to different effect. Common ones are nearest neighbour, bi linear, and bi quadratic.

The problem of texture sampling for rendering, however, is more complex than just pixel interpolation. Rendering is done at the pixel level, but a pixel covers more than a single point of a scene. To avoid ambiguity, we refer to pixels being rendered as pixels, and the pixels of the texture as texels. The size of this pixel, when mapped to texture space, may not be the same as the size of texels in the image; in fact, most of the time they are not the same. This leads to challenge of texture footprint, or the texel to pixel ratio.

There are two cases that need to be handled: 1) one texel covers many pixels (e.g. object is very close to camera) and 2) many texels are covered by a single pixel (e.g. object is very far from camera). These situations are addressed with texture magnification and minification, respectively. Texture magnification is fundamentally a reconstruction problem, where we don't have enough information from the texture and have to make a "guess" about what the value should be. Minification is fundamentally an anti-aliasing problem, where we need to blur or oversample the image to avoid jagged artifacts.

11.2.1 Mipmapping

Many ways to approach texture magnification and minification. Like many problems in computer science, a tradeoff can be made between execution time and memory footprint in implementing a solution to a problem. In graphics (especially real-time), often willing to trade memory for rendering speed gains. This leads to mipmapping, where we store multiple sizes of a texture and sample the appropriate one based on the texture footprint at a given pixel. This creates an image

pyramid. With mipmapping, each lower image has a resolution one half that above (for a total of one fourth the size), down to the final image which is just a single pixel. Because of this, the highest resolution image is usually chosen to be a power of two.

While storing all these copies of an image may seem like it would require a significant increase in the amount storage, mipmapping only increases the memory footprint of an image by one third. This is clearly demonstrated in **figure**.

Beyond sampling a single level of a mipmap, can also sample between levels. Details in chapter [ref] of programming notes on how to specify how mipmapping is handled in OpenGL.

11.2.2 Anisotropic Filtering

While mipmapping (and minification in general) work well in many cases, has built in assumption that square pixel maps to roughly square shape in texture space. In many cases this more or less holds; however, at oblique viewing angles pixels masp to increasingly skinny rectangle like shapes. Therefore, uniform minification results in overblurring in certain directions.

Anisotropic filtering addressing this, by sampling differently in different direction. **Figure of results**. This is accomplished by having mulitple texture samples per pixel, determined by the shape of the pixel in texture space. **Figure** Note that anisotropic filtering is not a replacement for traditional mipmapping, but an augmentation of it to get better results in certain situations. Smart implementations will only sample as many times as needed per pixel, usually up to a user specified maximum.

11.3 Modifying Surface Properties

With a sample from the texture obtained, the piece of texture mapping is using this value to modify the rendering result. With a programmable graphics pipeline, the only limits on how these values are used is your own creativity. However, there are several common uses that are worth exploring.

11.3.1 Colour (Diffuse) Maps

The most obvious way to use texture information is as the colour of the object. In our illuminations equations from section [ref], this is the diffuse (and often also ambient) coefficient of the object. No additional calculations are needed in this case, values from the texture are simply plugged directly into the lighting equations.

Clarify distinction between diffuse and albedo here.

11.3.2 Specular and Gloss Maps

Just as with diffuse maps, texture values can also be mapped to the specular component in the lighting equation. Can also map texture values to highlight exponent.

11.3.3 Bump and Normal Maps

I need to review this more before I can write this part.